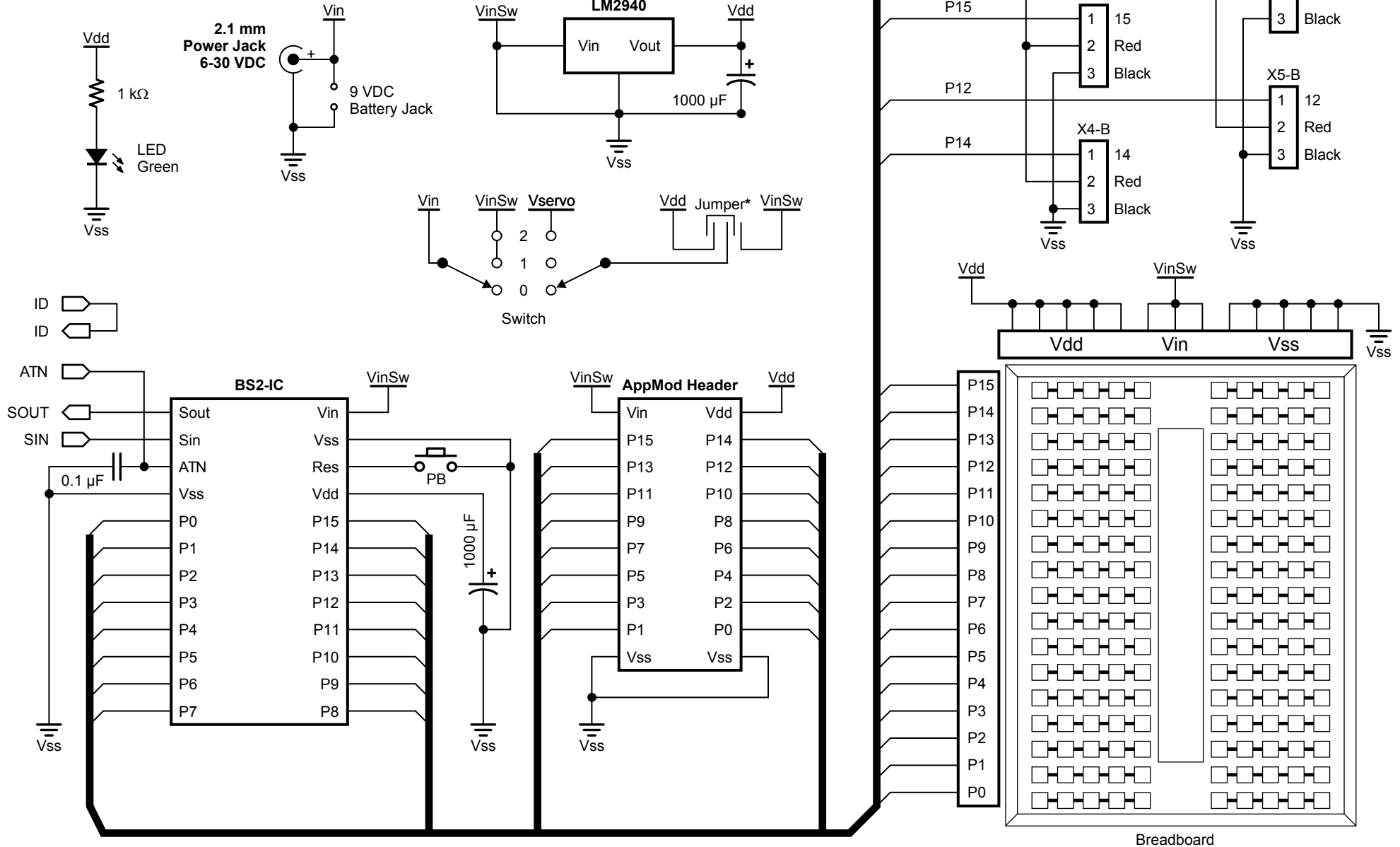
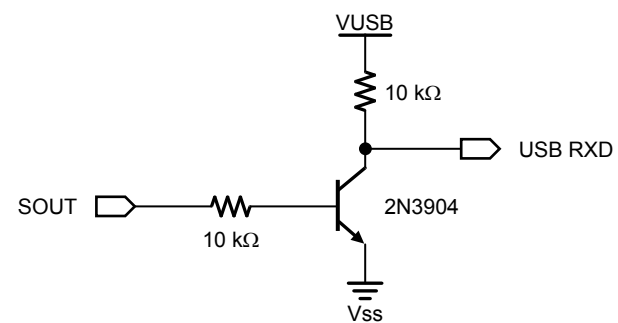
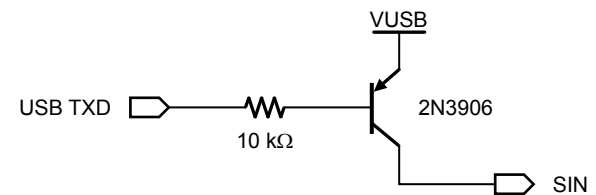
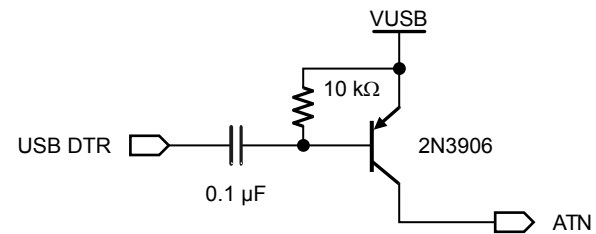
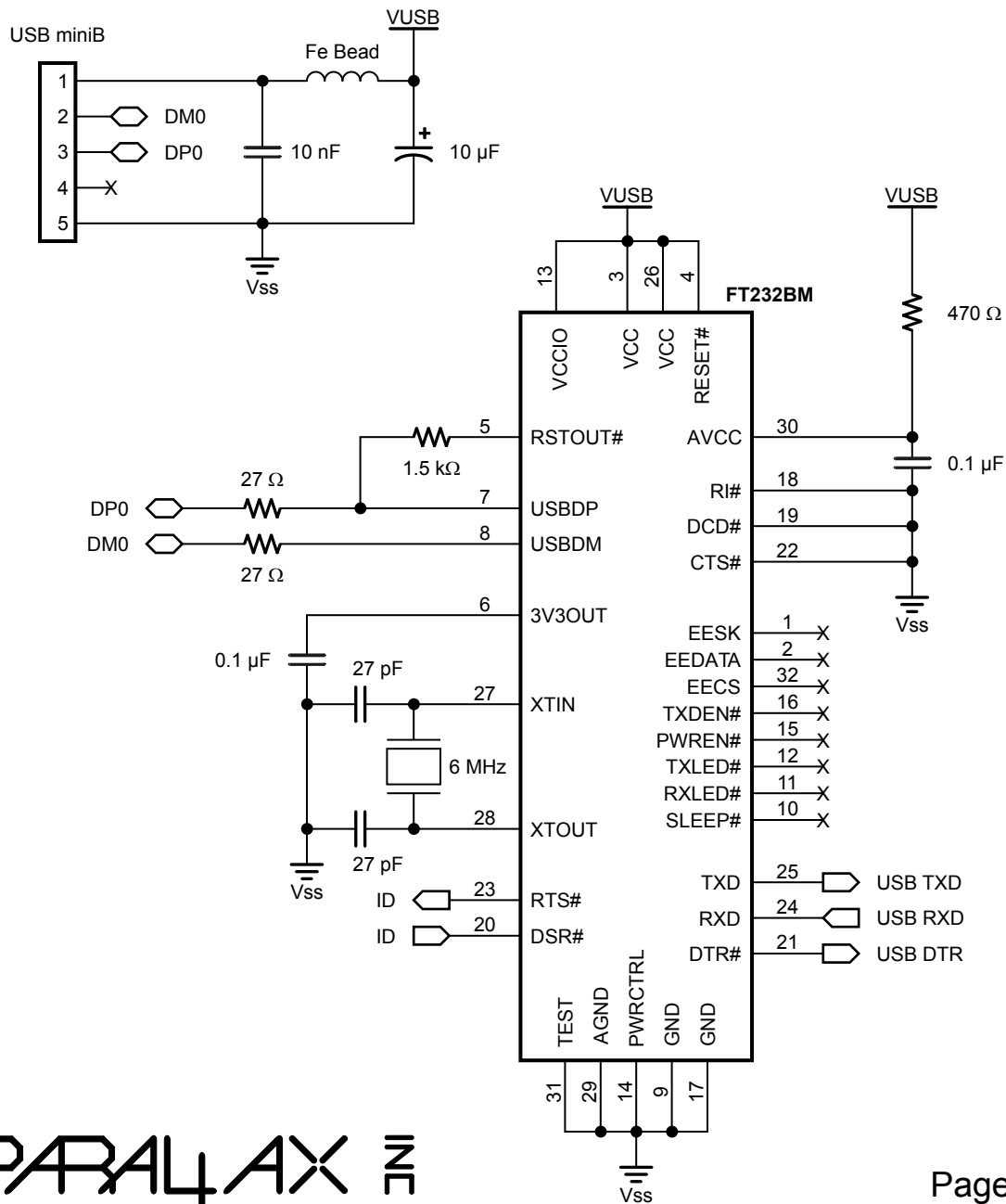


Circuitry Powered by External Source



Circuitry powered by USB



Robotics with the Boe-Bot

Student Guide

VERSION 2.2

PARALLAX 

WARRANTY

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

14-DAY MONEY BACK GUARANTEE

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

COPYRIGHTS AND TRADEMARKS

This documentation is copyright 2003-2004 by Parallax Inc. By downloading or obtaining a printed copy of this documentation or software you agree that it is to be used exclusively with Parallax products. Any other uses are not permitted and may represent a violation of Parallax copyrights, legally punishable according to Federal copyright or intellectual property laws. Any duplication of this documentation for commercial uses is expressly prohibited by Parallax Inc. Duplication for educational use is permitted, subject to the following Conditions of Duplication: Parallax Inc. grants the user a conditional right to download, duplicate, and distribute this text without Parallax's permission. This right is based on the following conditions: the text, or any portion thereof, may not be duplicated for commercial use; it may be duplicated only for educational purposes when used solely in conjunction with Parallax products, and the user may recover from the student only the cost of duplication.

This text is available in printed format from Parallax Inc. Because we print the text in volume, the consumer price is often less than typical retail duplication charges.

BASIC Stamp, Stamps in Class, Board of Education, SumoBot, and SX-Key are registered trademarks of Parallax, Inc. If you decide to use registered trademarks of Parallax Inc. on your web page or in printed material, you must state that "(registered trademark) is a registered trademark of Parallax Inc." upon the first appearance of the trademark name in each printed document or web page. Boe-Bot, HomeWork Board, Parallax, the Parallax logo, and Toddler are trademarks of Parallax Inc. If you decide to use trademarks of Parallax Inc. on your web page or in printed material, you must state that "(trademark) is a trademark of Parallax Inc.", "upon the first appearance of the trademark name in each printed document or web page. Other brand and product names are trademarks or registered trademarks of their respective holders.

ISBN 1-928982-03-4

DISCLAIMER OF LIABILITY

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

WEB SITE AND DISCUSSION LISTS

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

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

ERRATA

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

Table of Contents

Preface	5
Foreword.....	5
Audience.....	6
Support and Discussion Groups.....	6
The Stamps in Class Curriculum.....	7
Foreign Translations.....	8
Special Contributors.....	8
Chapter 1: Your Boe-Bot's Brain	1
Hardware and Software.....	2
Activity #1: Getting the Software.....	4
Activity #2: Installing the Software.....	10
Activity #3: Setting up the Hardware and Testing the System.....	13
Activity #4: Your First Program.....	22
Activity #5: Looking up Answers.....	30
Activity #6: Introducing ASCII Code.....	33
Activity #7: When You're Done.....	35
Summary.....	37
Chapter 2: Your Boe-Bot's Servo Motors	41
Introducing the Continuous Rotation Servo.....	41
Activity #1: How to Track Time and Repeat Actions.....	42
Activity #2: Tracking Time and Repeating Actions with a Circuit.....	45
Activity #3: Connecting the Servo Motors.....	58
Activity #4: Centering the Servos.....	66
Activity #5: How to Store Values and Count.....	71
Activity #6: Testing the Servos.....	75
Summary.....	86
Chapter 3: Assemble and Test Your Boe-Bot	91
Activity #1: Assembling the Boe-Bot.....	91
Activity #2: Re-Test the Servos.....	101
Activity #3: Start/Reset Indicator Circuit and Program.....	105
Activity #4: Testing Speed Control with the Debug Terminal.....	111
Summary.....	118
Chapter 4: Boe-Bot Navigation	123
Activity #1: Basic Boe-Bot Maneuvers.....	123
Activity #2: Tuning the Basic Maneuvers.....	129
Activity #3: Calculating Distances.....	132
Activity #4: Maneuvers – Ramping.....	137

Activity #5: Simplify Navigation with Subroutines	140
Activity #6: Building Complex Maneuvers in EEPROM	146
Summary	157
Chapter 5: Tactile Navigation with Whiskers	165
Tactile Navigation	165
Activity #1: Building and Testing the Whiskers	166
Activity #2: Field Testing the Whiskers	174
Activity #3: Navigation with Whiskers	177
Activity #4: Artificial Intelligence and Deciding When You're Stuck.....	182
Summary	188
Chapter 6: Light Sensitive Navigation with Photoresistors	193
Introducing the Photoresistor.....	193
Activity #1: Building and Testing Photoresistor Circuits	194
Activity #2: Roam and Avoid Shadows Like Objects	200
Activity #3: A More Responsive Shadow Controlled Boe-Bot.....	203
Activity #4: Getting More Information from Your Photoresistors.....	205
Activity #5: Flashlight Beam Following Boe-Bot	210
Activity #6: Roaming Toward the Light	219
Summary	227
Chapter 7: Navigating with Infrared Headlights.....	235
Using Infrared Headlights to See the Road	235
Activity #1: Building and Testing the IR Pairs	237
Activity #2: Field Testing for Object Detection and Infrared Interference	242
Activity #3: Infrared Detection Range Adjustments	247
Activity #4: Object Detection and Avoidance	249
Activity #5: High Performance IR Navigation.....	252
Activity #6: The Drop-Off Detector.....	255
Summary	262
Chapter 8: Robot Control with Distance Detection	269
Determining Distance with the Same IR LED/Detector Circuit	269
Activity #1: Testing the Frequency Sweep	269
Activity #2: Boe-Bot Shadow Vehicle	277
Activity #3: Following a Stripe.....	286
Summary	294
Appendix A: PC to BASIC Stamp Communication Trouble-Shooting.....	301
Appendix B: BASIC Stamp and Carrier Board Components and Features	305
Appendix C: Resistor Color Codes	309
Appendix D: Breadboarding Rules	311

Appendix E: Boe-Bot Parts Lists	317
Appendix F: Balancing Photoresistors	321
Appendix G: Tuning IR Distance Detection	329
Appendix H: Boe-Bot Navigation Contests	335
Index	339

Preface

FOREWORD

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

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

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



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

The activities and projects in this text begin with an introduction to your Boe-Bot's brain, the Parallax BASIC Stamp[®] 2 microcontroller, and then move on to construction, testing, and calibration of the Boe-Bot. After that, you will program the Boe-Bot for basic maneuvers, and then proceed to adding sensors and writing programs that make it react to its surroundings and perform autonomous tasks.

AUDIENCE

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

SUPPORT AND DISCUSSION GROUPS

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

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

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

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

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

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

THE STAMPS IN CLASS CURRICULUM

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

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

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

Stamps in Class Student Guides:

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

***“Applied Sensors”*, Student Guide, Version 1.3, Parallax Inc., 2003**

***“Basic Analog and Digital”*, Student Guide, Version 1.3, Parallax Inc., 2004**

***“Process Control”*, Student Guide, Version 2.0, Parallax Inc., 2004**

More Robotics Kits:

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

***“Advanced Robotics: with the Toddler”*, Student Guide, Version 1.2, Parallax Inc., 2003**

***“SumoBot” Manual*, Version 2.0, Parallax Inc., 2004**

Educational Project Kits:

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

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

Reference

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

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

FOREIGN TRANSLATIONS

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

SPECIAL CONTRIBUTORS

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

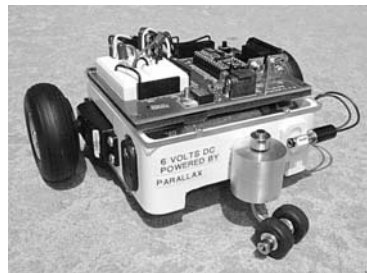


Figure P-2
Original Boe-Bot
Prototype

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

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

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

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

Chapter 1: Your Boe-Bot's Brain

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

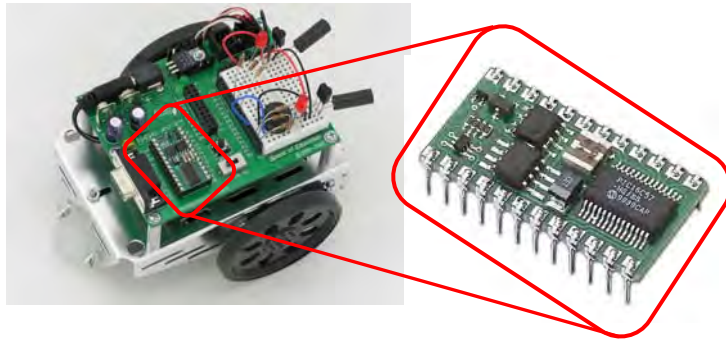


Figure 1-1
BASIC
Stamp® 2
module on a
Boe-Bot™
robot.

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

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



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

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



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

The *What's a Microcontroller?* Student Guide is the recommended first text for beginners. It is full of examples of how to use microcontrollers, and how to make the BASIC Stamp the brain of your own microcontrolled inventions. It's available for free download from www.parallax.com, and it's also included on the Parallax CD. Many electronics outlets carry the What's a Microcontroller Kits and printed Student Guides. If you have any difficulty finding them locally, they can also be purchased directly from Parallax, either on-line at www.parallax.com or by phone at (888) 512-1024.

HARDWARE AND SOFTWARE

Getting started with BASIC Stamp programming is similar to getting started with a brand-new PC or laptop. The first things that most people do when they get a new PC or laptop is take it out of the box, plug it in, install and test some software, and maybe even write some software of their own using a programming language. If this is your first time using BASIC Stamp microcontrollers, you will be doing all these same activities. This chapter shows you how to get up and running with BASIC Stamp programming as it guides you through:

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

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

- ✓ If you have already completed the *What's a Microcontroller?* Student Guide, skip to the next chapter.
- ✓ Likewise, if you are already familiar with your BASIC Stamp and Board of Education or BASIC Stamp HomeWork Board, skip to the next chapter.



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

Introducing the BASIC Stamp and Board of Education

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

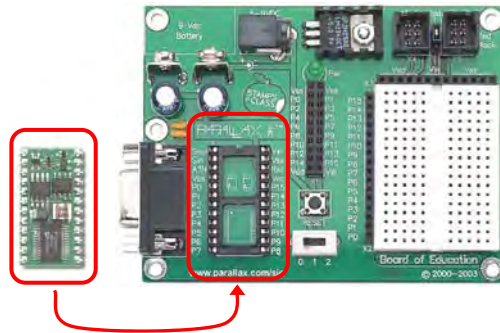


Figure 1-2
BASIC Stamp® 2 Module
(left)

Board of Education®
Carrier Board (right)

Introducing the BASIC Stamp HomeWork Board

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

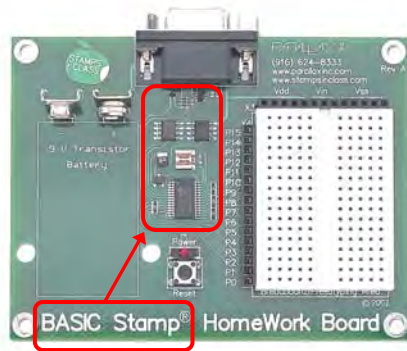


Figure 1-3
BASIC Stamp®
HomeWork Board™
project platform.



What's the difference?

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

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

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

ACTIVITY #1: GETTING THE SOFTWARE

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



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

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

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

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

Computer System Requirements

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

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



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

Downloading the Software from the Internet

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

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



Figure 1-4
The Parallax Web Site:

www.parallax.com

- ✓ When you get to the BASIC Stamp Software page, find a BASIC Stamp Windows Editor download with a version number of 2.0 or higher.
- ✓ Click the Download icon. In Figure 1-5, the Download icon looks like a file folder to the right of the description: “BASIC Stamp Windows Editor Version 2.0 Beta 1 (6MB)”.

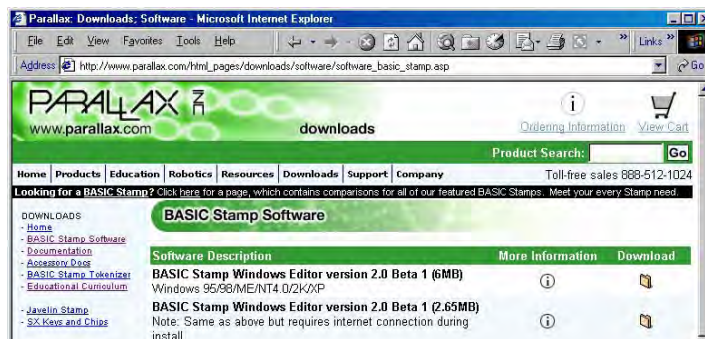


Figure 1-5
The Parallax Web Site Downloads Page

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

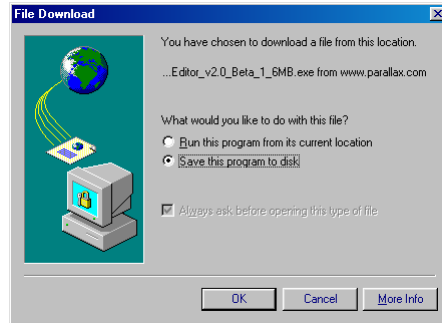


Figure 1-6
File Download
Window

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

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

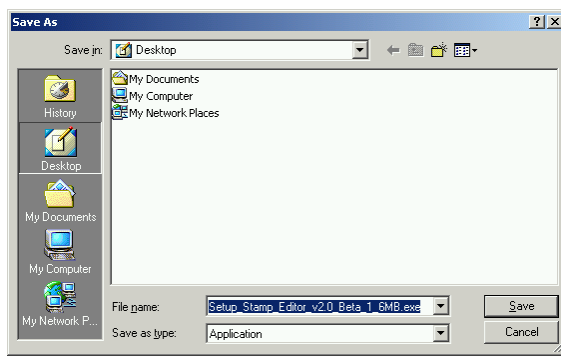


Figure 1-7
Save As Window

*Selecting a place
to save the file*

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

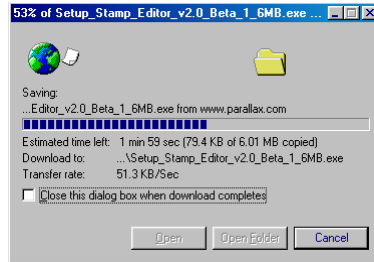


Figure 1-8
Download
Progress Window

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

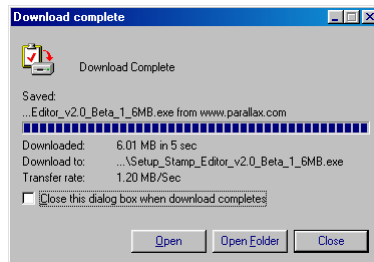


Figure 1-9
Download
Complete
Window

*Go to Activity #2:
Installing the
Software.*



Other free downloads at the Parallax web site include:

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

Finding the Software on the Parallax CD

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

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

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

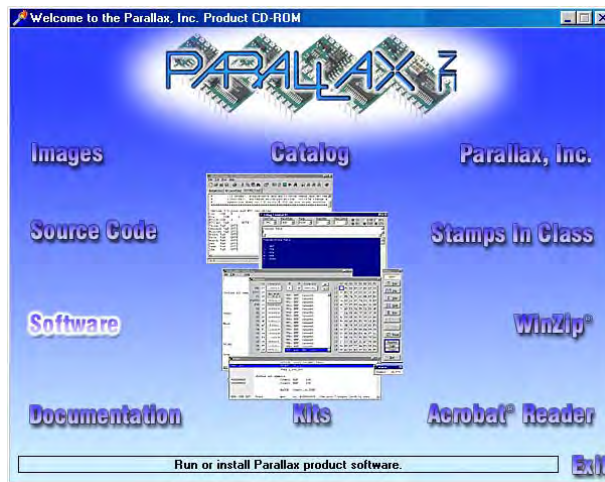


Figure 1-10
The Parallax CD
Browser

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



Figure 1-11
The Parallax CD
Browser

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



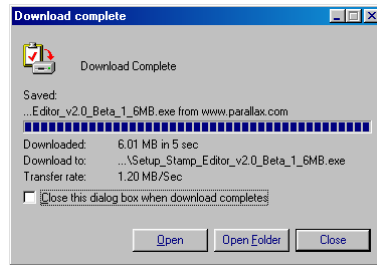
Free downloads at the Parallax web site are included in the Parallax CD, but only up to the date the CD was created. The date on the front of the CD indicates when it was created. If the CD is just a month or two old, you will probably have the most up-to-date material. If it's an older CD, consider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

ACTIVITY #2: INSTALLING THE SOFTWARE

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

Installing the Software Step by Step

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

**Figure 1-12**

Download
Complete
Window

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

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

**Figure 1-13**

The Parallax CD
Browser

*Install button
located near
bottom of window.*

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

**Figure 1-14**

InstallShield
Wizard for the
BASIC Stamp
Editor

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



Figure 1-15
Setup Type

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



Figure 1-16
Ready to Install

Click the Install button.

- ✓ When the InstallShield Wizard window tells you “InstallShield Wizard Completed” as shown in Figure 1-17, click Finish.

Congratulations! Your BASIC Stamp Editor is now installed.



Figure 1-17
InstallShield
Wizard
Completed

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

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

Computer Serial Cable Setup

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

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

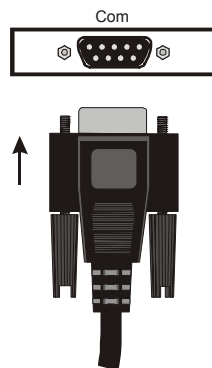


Figure 1-18
PC or Laptop
COM Port

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

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



FTDI's US232B/LC USB to Serial Adapter:

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

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



Figure 1-19

FTDI's US232B/LC USB to Serial Adapter

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

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

- ✓ If you have a BASIC Stamp and Board of Education, follow the instructions in the next section, Board of Education Connection Instructions.
- ✓ If you have a BASIC Stamp HomeWork board, skip to the BASIC Stamp HomeWork Board Connection Instructions on page 18.
- ✓ If your equipment is already hooked up, skip to the Testing for Communication section on page 21.

Board of Education Connection Instructions

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

Required Hardware

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

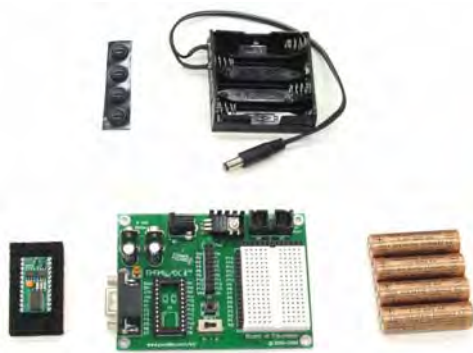


Figure 1-20
Getting Started Hardware
for the BASIC Stamp and
Board of Education

Connecting the Hardware

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

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

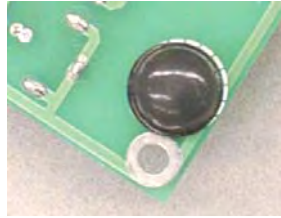


Figure 1-21
Rubber Feet (left)
Affixed to Underside
of the Board of
Education (right)

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

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

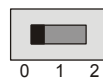


Figure 1-22
3-position Switch

*Set to position-0 to turn
off the power.*

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



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

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

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



Figure 1-23
Battery Pack

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

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



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

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

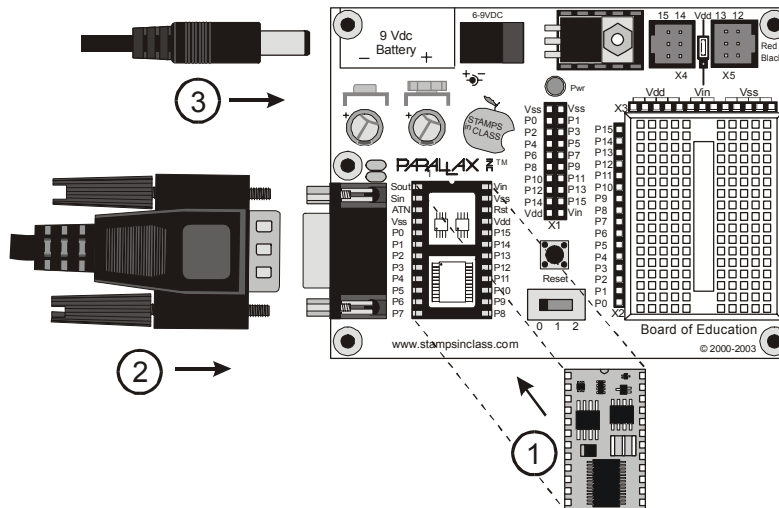


Figure 1-24
Board of Education, BASIC Stamp, Battery, and Serial Cable

Connect components in the order shown in the diagram.

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

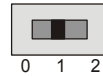


Figure 1-25
3-position Switch

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

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

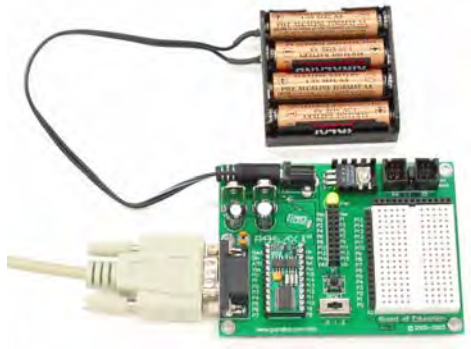


Figure 1-26
BASIC Stamp and Board of
Education Connected and
Ready to Program

- ✓ Skip to the Testing for Communication section on page 21.

BASIC Stamp HomeWork Board Connection Instructions

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

Required Hardware

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

(1) BASIC Stamp HomeWork Board



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

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



Figure 1-28
Rubber Feet

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

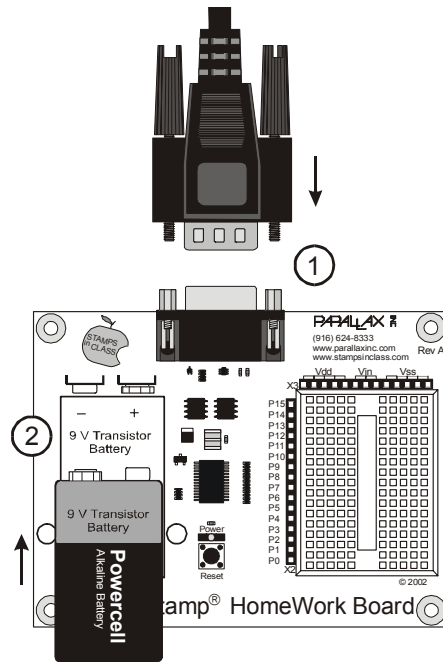


Figure 1-29
HomeWork Board
and Serial Cable

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

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



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

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



Figure 1-30
BASIC Stamp
HomeWork Board
Ready to Program

Testing for Communication

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



Figure 1-31
BASIC Stamp
Editor Shortcut

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



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

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



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

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

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

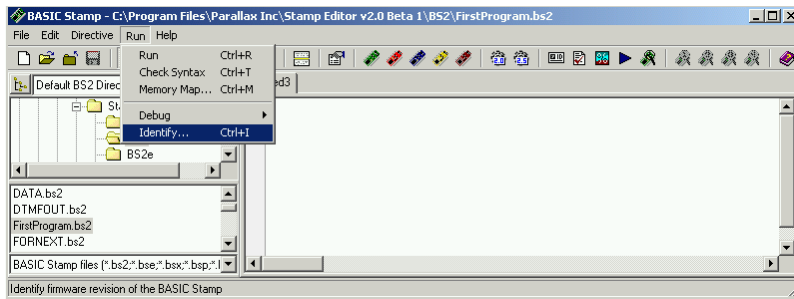


Figure 1-32
BASIC
Stamp Editor

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

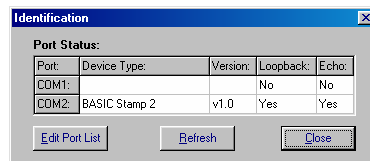


Figure 1-33
Identification
Window

*Example: BASIC
Stamp 2 found on
COM2.*

- ✓ Check the Identification window to make sure a BASIC Stamp 2 has been detected on one of the COM ports. If the BASIC Stamp 2 has been detected, then you are ready for Activity #4: Your First Program.
- ✓ If the Identification window does not detect a BASIC Stamp 2 on any of the COM ports, go to page 301 (Appendix A: PC to BASIC Stamp Communication Trouble-Shooting).

ACTIVITY #4: YOUR FIRST PROGRAM

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

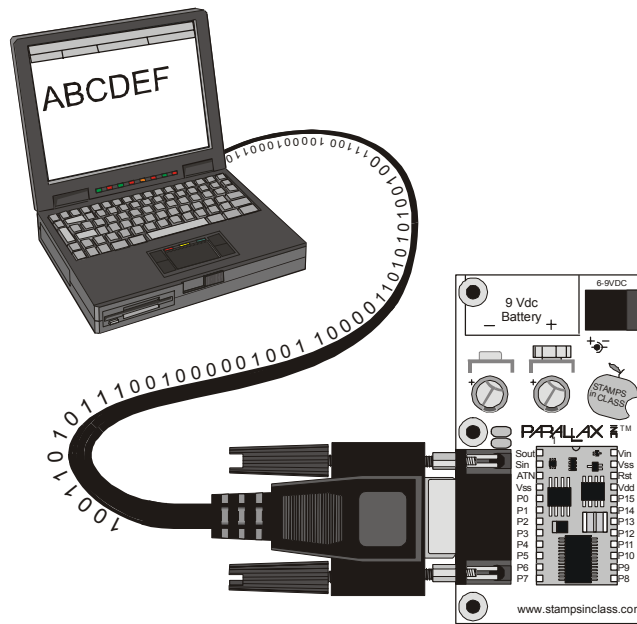


Figure 1-34
Messages from the
BASIC Stamp to Your
Computer

Your First Program

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

Example Program: HelloBoeBot.bs2

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

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

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

END
```

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

- ✓ Begin by clicking the BS2 icon (the green diagonal chip) on the toolbar, shown highlighted in Figure 1-35. If you hold your cursor over this button, its flyover help description “Stamp Mode: BS2” will appear.
- ✓ Next, click on the gear icon labeled “2.5” shown highlighted in Figure 1-36. Its flyover help description is “PBASIC Language: 2.5”.



Figure 1-35
BS2 Icon

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

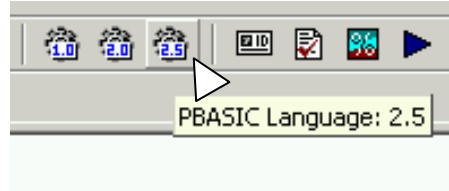


Figure 1-36
PBASIC 2.5 Icon

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



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

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



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

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

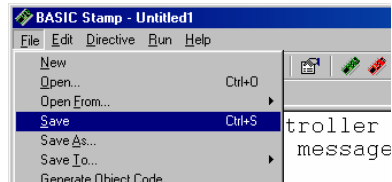


Figure 1-38
Saving the
Program
HelloBoeBot.bs2

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

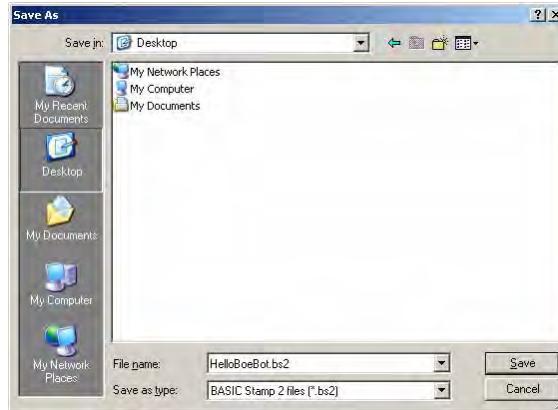


Figure 1-39
Entering the File
Name



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

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

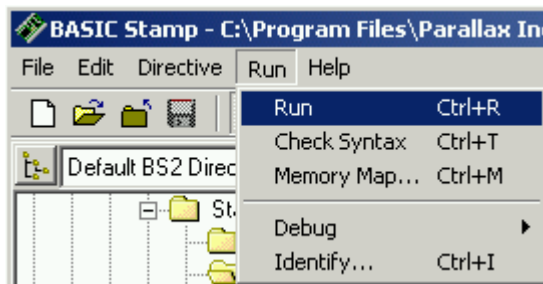


Figure 1-40
Running
Your First
Program
HelloBoeBot.
bs2

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

- ✓ Press and release the Reset button. Did you see a second “Hello...” message appear in the Debug Terminal?

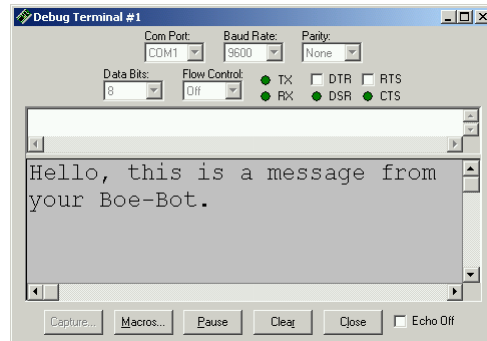


Figure 1-41
Debug Terminal

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



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



Figure 1-42
BASIC Stamp Editor
Shortcut Buttons

How HelloBoeBot.bs2 Works

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

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

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

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

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

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

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

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

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

```
END
```

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

Your Turn – DEBUG Formatters and Control Characters

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

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

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


```
' Robotics with the Boe-Bot - HelloBoeBotYourTurn.bs2
' BASIC Stamp does simple math, and sends the results
' to the Debug Terminal.
```
- ✓ Add these three lines between the first **DEBUG** command and the **END** command:


```
DEBUG CR, "What's 7 X 11?"
DEBUG CR, "The answer is: "
DEBUG DEC 7 * 11
```
- ✓ Save the changes you made by clicking File and selecting Save.

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

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

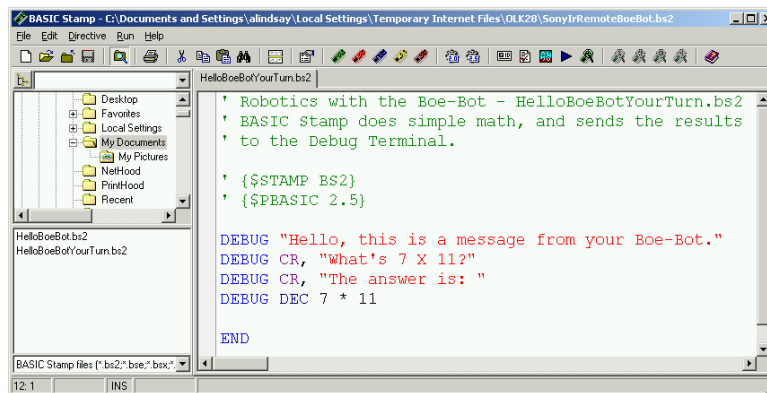


Figure 1-43
Modified
HelloBoeBot.bs2

*Check your work
against the
example program
shown here.*

Your Debug Terminal should now resemble Figure 1-44.

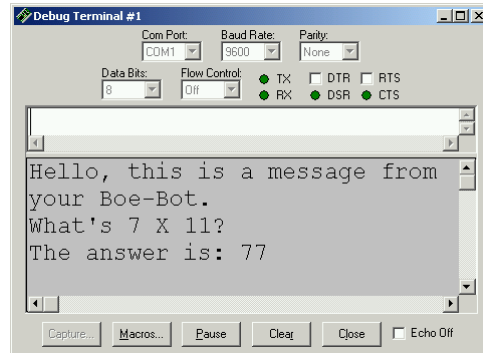


Figure 1-44
Modified
HelloBoeBot.bs2 Debug
Terminal Output

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

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

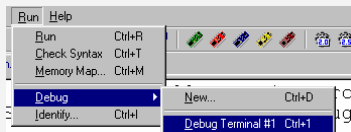


Figure 1-45
Debug Terminal 1 to
Foreground

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

ACTIVITY #5: LOOKING UP ANSWERS

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

Using the BASIC Stamp Editor's Help

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

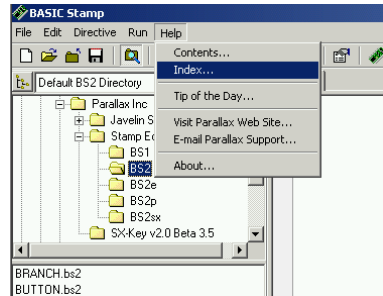


Figure 1-46
Selecting Index
from the Help
Menu

- ✓ Type **DEBUG** into the field labeled Type in the keyword to find: (shown in Figure 1-47).
- ✓ When the word **DEBUG** appears in the list below the field you are typing in, double-click it, then click the Display button.

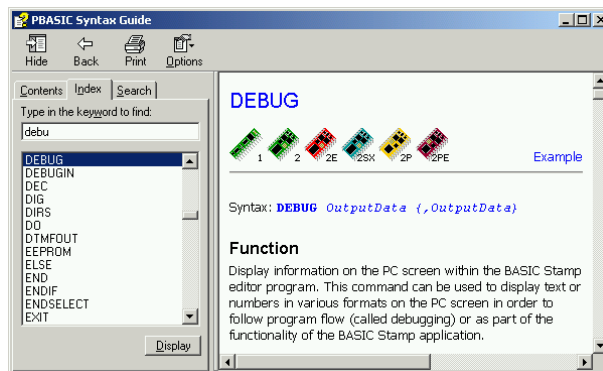


Figure 1-47
Looking up the
DEBUG Command
Using Help

Your Turn

- ✓ Use the scrollbar to review the **DEBUG** command article. Notice that it has lots of explanations and example programs you can try.
- ✓ Click the Contents tab, and find **DEBUG** there.

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

Getting and Using the BASIC Stamp Manual

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



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

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

Viewing the *BASIC Stamp Manual* on the Parallax CD

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

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

BASIC STAMP COMMAND REFERENCE	77
AUXIO	81
BRANCH	83
BUTTON	85
COUNT	89
DATA	91
DEBUG	97

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

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

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

5: BASIC Stamp Command Reference - DEBUG

DEBUG

BS1	BS2	BS2e	BS2sx	BS2p

DEBUG *OutputData* { *OutputData* }

Function

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

Figure 1-49

Reviewing the **DEBUG** Command in the *BASIC Stamp Manual*

Your Turn

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

ACTIVITY #6: INTRODUCING ASCII CODE

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

Programming with ASCII Code

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

Example Program – AsciiName.bs2

✓ Enter and run AsciiName.bs2.



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

' {\$STAMP BS2} - Use the diagonal green electronic chip icon .

' {\$PBASIC 2.5} - Use the gear icon labeled 2.5.

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

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

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

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

END
```

How AsciiName.bs2 Works

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

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

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

Your Turn – Exploring ASCII Code

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

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

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

ACTIVITY #7: WHEN YOU'RE DONE

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



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

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

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

Disconnecting Power

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

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

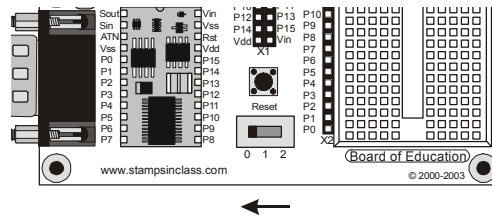


Figure 1-50
Switching Off the Power
for the Board of
Education Rev C



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

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

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

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

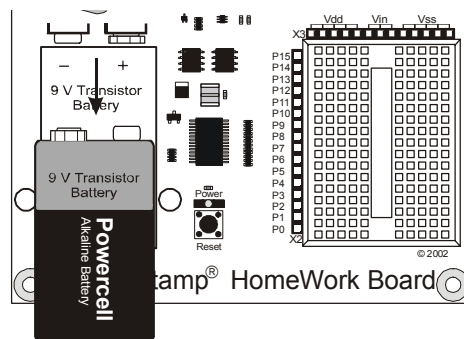


Figure 1-51
Disconnecting the
Power for the
HomeWork Board



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

Your Turn

- ✓ Disconnect power now.

SUMMARY

This chapter guided you through the following:

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

Questions

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

Exercises

1. Explain what you can do with each PBASIC command you learned in this chapter.
2. Explain what the asterisk does in this command:

```
DEBUG DEC 7 * 11
```
3. There is a problem with these two commands. When you run the code, the numbers they display are stuck together so that it looks like one large number

instead of two small ones. Modify these two commands so that the answers appear on different lines in the Debug Terminal.

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

Projects

1. Write a program that uses a **DEBUG** instruction to display the solution to the math problem: $1 + 2 + 3 + 4$.
2. Predict what you would expect to see if you removed the **DEC** formatter from this command. Use a PBASIC program to test your prediction.

```
DEBUG DEC 7 * 11
```

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

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

Solutions

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

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

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

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

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

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

END
```

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

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

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

DEBUG 7 * 11

END
```

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

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

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

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

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

END
```

The output from the Debug Terminal is:

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

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

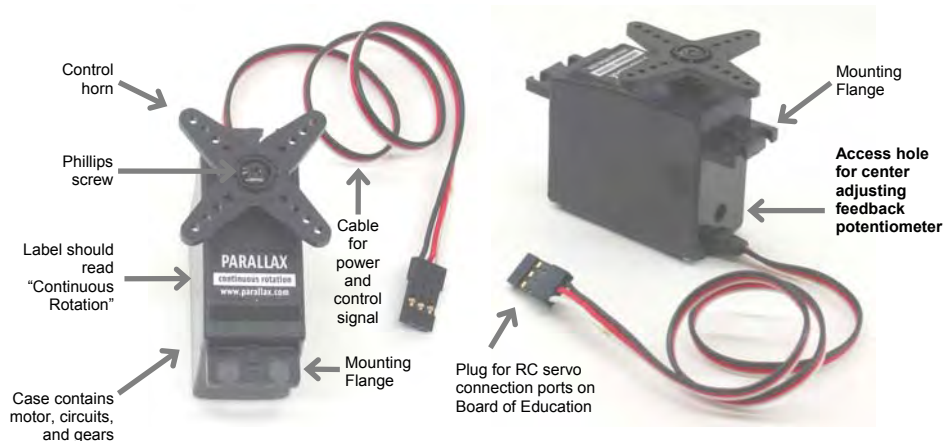
Chapter 2: Your Boe-Bot's Servo Motors

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

INTRODUCING THE CONTINUOUS ROTATION SERVO

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

Figure 2-1
Parallax Continuous Rotation Servo



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



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

ACTIVITY #1: HOW TO TRACK TIME AND REPEAT ACTIONS

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

Displaying Messages at Human Speeds

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

PAUSE Duration

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

```
PAUSE 1000
```

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

```
PAUSE 2000
```



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

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

Example Program: TimedMessages.bs2

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

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

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

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

DEBUG "Start timer..."

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

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

DEBUG CR, "Done."

END
```



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

Enter, save, and run TimedMessages.bs2.

Your Turn – Different Pause Durations

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

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

```
DEBUG "Start timer..."

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

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

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

Over and Over Again

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

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

Example Program: HelloOnceEverySecond.bs2

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

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

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

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

Your Turn – A Different Message

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

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

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

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

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

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



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

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

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

Introducing the LED and Resistor

A resistor is a component that ‘resists’ the flow of electricity. This flow of electricity is called current. Each resistor has a value that tells how strongly it resists current flow.

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

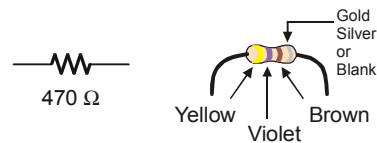


Figure 2-2
470 Ω Resistor Part
Drawing

*Schematic symbol (left)
and part drawing (right)*



The colored stripes indicate resistance values. See Appendix C: Resistor Color Codes for information on how to determine a resistor’s value from the colored stripes on its ceramic case.

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

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

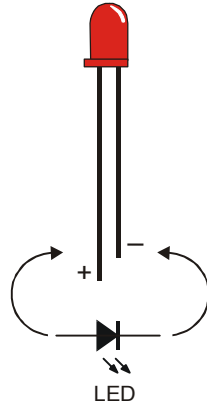


Figure 2-3
LED Part Drawing and
Schematic Symbol

*Part drawing (above) and
schematic symbol
(below).*

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

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



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

LED Test Circuit Parts

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

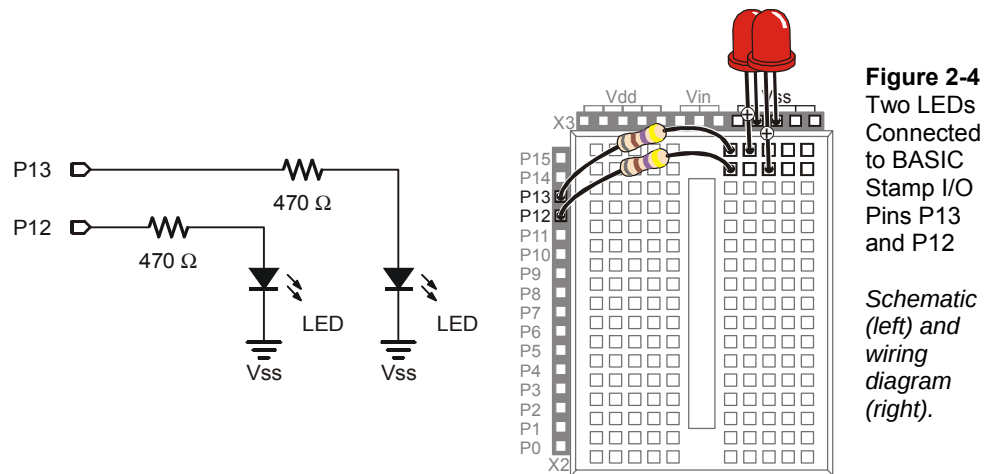


Always disconnect power to your board before building or modifying circuits! For the Board of Education Rev C, set the 3-position switch to position-0. For the BASIC Stamp HomeWork Board, disconnect the 9 V battery from the battery clip. Always double-check your circuit for errors before reconnecting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

LED Test Circuits

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

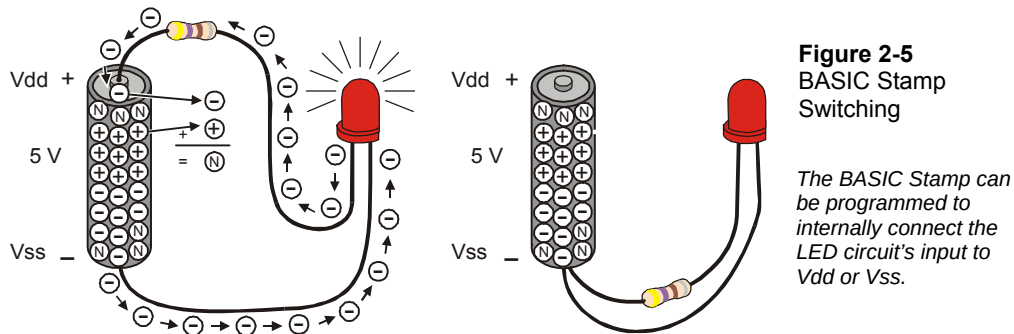
- ✓ Build the circuit shown in Figure 2-4.
- ✓ Make sure that the shorter pins on each LED (the cathodes) are plugged into black sockets labeled Vss.
- ✓ Make sure the longer pins (the anodes, marked with a $\oplus$ in the wiring diagram) are connected to the white breadboard sockets exactly as shown.



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

New to building circuits? See Appendix D: Breadboarding Rules.

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



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

Current refers to the rate at which electrons pass through a circuit. You will often see measurements of current expressed in amps, which is abbreviated A. The amount of current an electric motor draws is often measured in amps, for example 2 A, 5 A, etc. However, the currents you will use in the Board of Education are measured in thousandths of an amp, or milliamperes. For example, 10.3 mA passes through the circuit in Figure 2-5.

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

Programs that Control the LED Test Circuits

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

HIGH *Pin*

LOW *Pin*

For example, if you use the command

HIGH 13

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

Likewise, if you use the command

LOW 13

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

Example Program: HighLowLed.bs2

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

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

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

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

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

How HighLowLed.bs2 Works

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

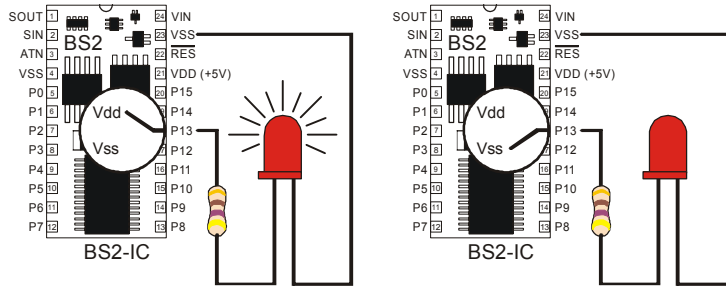


Figure 2-6
BASIC Stamp
Switching

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

A Diagnostic Test for your Computer

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

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

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

If the LED blinked on and off only once and then stopped, you have a computer with a non-standard serial port design. If you disconnect the serial cable from your board and

press the Reset button, the BASIC Stamp will run the program properly without freezing. In programs you write yourself, you should add a single command:

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

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

Introducing the Timing Diagram

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

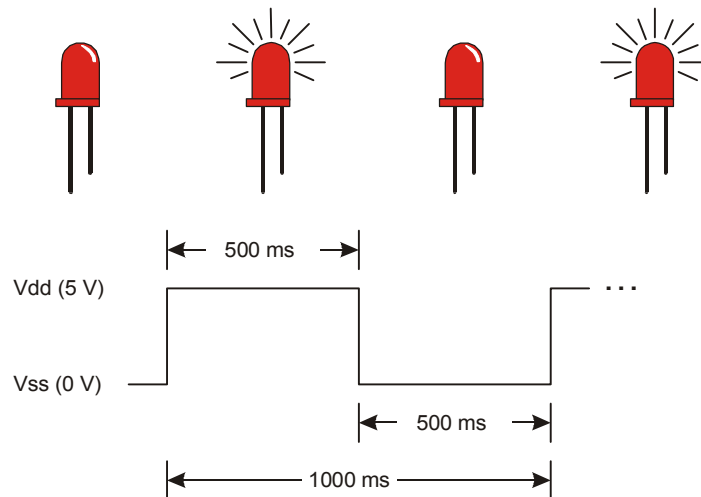


Figure 2-7
Timing Diagram
for
HighLowLed.bs2

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

Your Turn – Blink the Other LED

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

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

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

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

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

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

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

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

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

- ✓ Try it!

Viewing a Servo Control Signal with an LED

The high and low signals you will program the BASIC Stamp to send to the servo motors must last for very precise amounts of time. That's because the servo motors measure the amount of time the signal stays high, and use it as an instruction for where to turn. For

accurate servo motor control, the time these signals stay high must be much more precise than you can get with a **HIGH** and a **PAUSE** command. You can only change the **PAUSE** command's *Duration* argument by 1 ms (remember, that's 1/1000 of a second) at a time. There's a different command called **PULSOUT** that can deliver high signals for precise amounts of time. These amounts of time are values you use in the *Duration* argument, and they are measured in units that are two millionths of a second!

PULSOUT Pin, Duration



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

For example, 8 microseconds is abbreviated 8 μs .

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

```
PULSOUT 13, 1
```

This command would turn the LED on for 4 μs

```
PULSOUT 13, 2
```

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

```
PULSOUT 13, 65000
```

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

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

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



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

Example Program: PulseP13Led.bs2

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

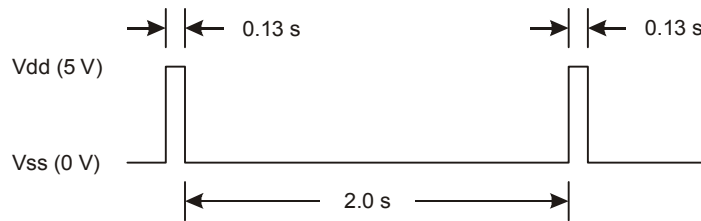


Figure 2-8
Timing Diagram
for
PulseP13Led.bs2

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

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

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

DEBUG "Program Running!"

DO
  PULSOUT 13, 65000
  PAUSE 2000
LOOP
```

Example Program: PulseBothLeds.bs2

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

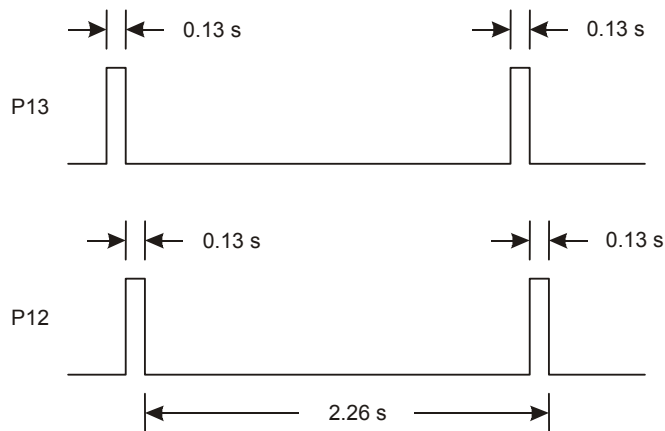


Figure 2-9
Timing Diagram for
PulseBothLeds.bs2

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



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

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

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

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

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

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

DEBUG "Program Running!"
```



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

Your Turn – Viewing the Full Speed Servo Signal

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

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

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

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

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

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

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

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

```
DO
```

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

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

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

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

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

ACTIVITY #3: CONNECTING THE SERVO MOTORS

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

Parts for Connecting the Servos

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

Finding the Connection Instructions for Your Carrier Board

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

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

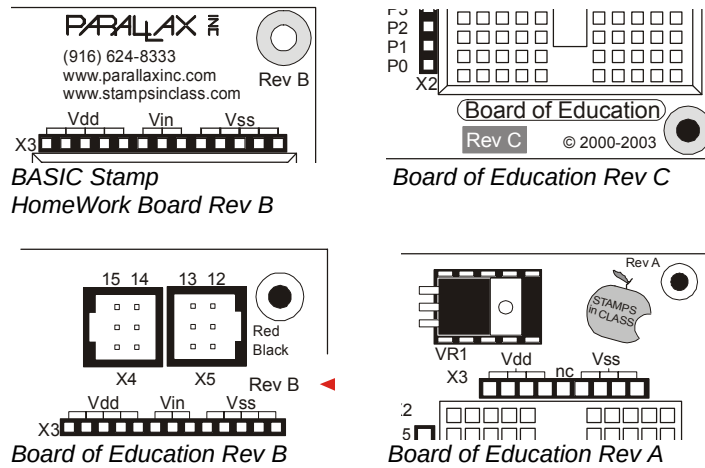


Figure 2-10
Examples of Rev
Labels on the
BASIC Stamp
HomeWork Board
and the Board of
Education

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

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

Board of Education Rev B

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

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

Board of Education Rev A

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

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

Connecting the Servos to the Board of Education Rev C

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

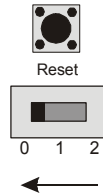


Figure 2-11
Disconnect
Power

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

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



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

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

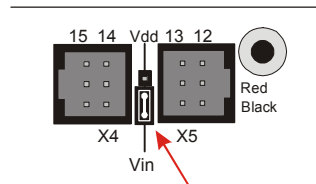


CAUTION – Misuse of AC powered DC supplies can damage your servos.

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

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

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



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

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

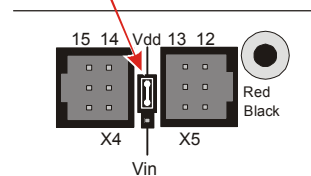


Figure 2-12
Selecting Your Servo's Power Supply on the Board of Education Rev C

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

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

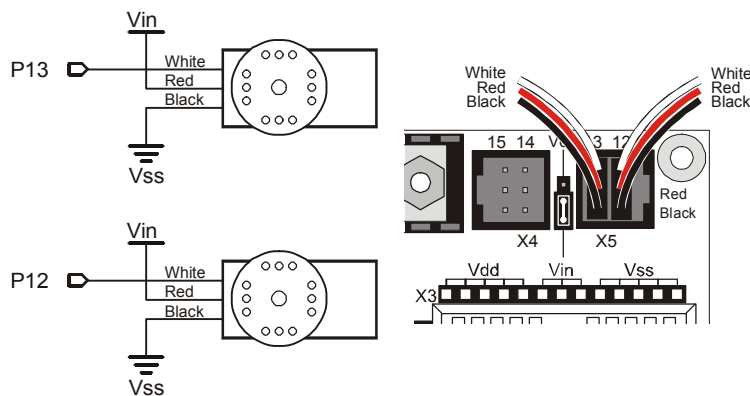


Figure 2-13
Servo Connection Schematic and Wiring Diagram for the Board of Education Rev C



How do I tell which servo is connected to P13 and which servo is connected to P12?

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

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

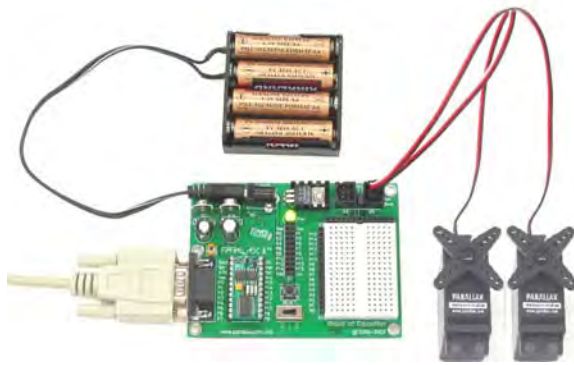


Figure 2-14
Board of Education
with Servos and
Battery Pack
Connected

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

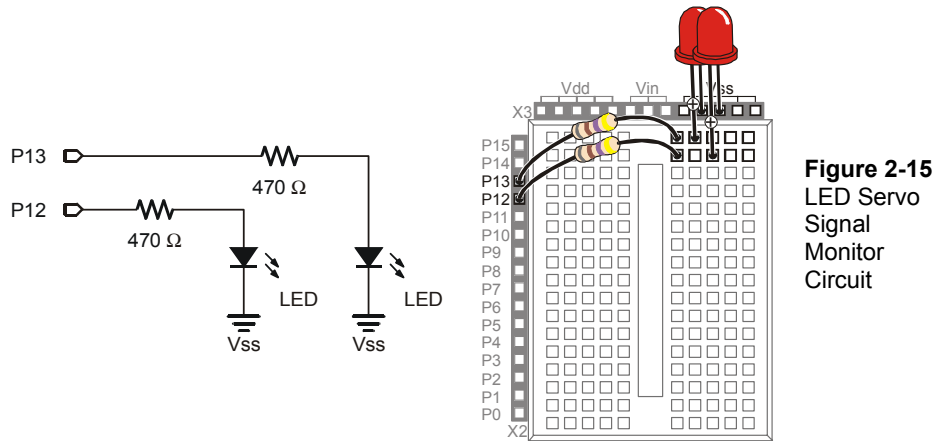


Figure 2-15
LED Servo
Signal
Monitor
Circuit



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

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

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

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

Connecting the Servos to the BASIC Stamp HomeWork Board

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

Parts List:

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

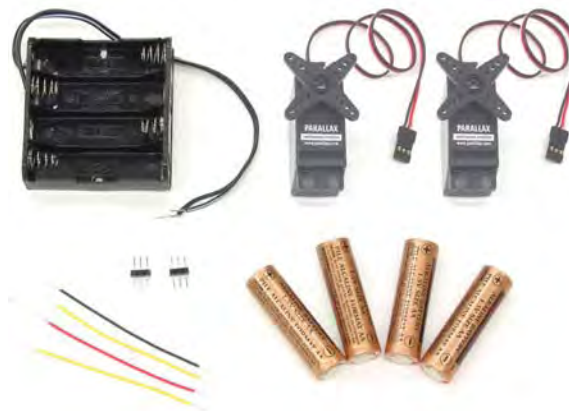


Figure 2-16
Servo Centering
Parts for the
HomeWork
Board

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

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

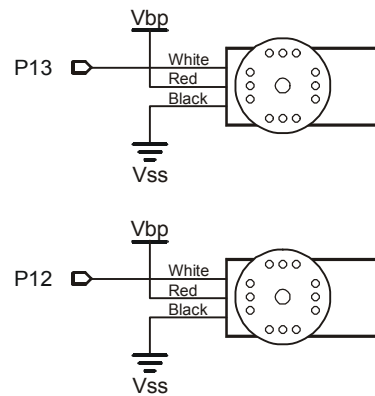


Figure 2-17
Servo Connection
Schematic for the
BASIC Stamp
HomeWork Board.

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



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

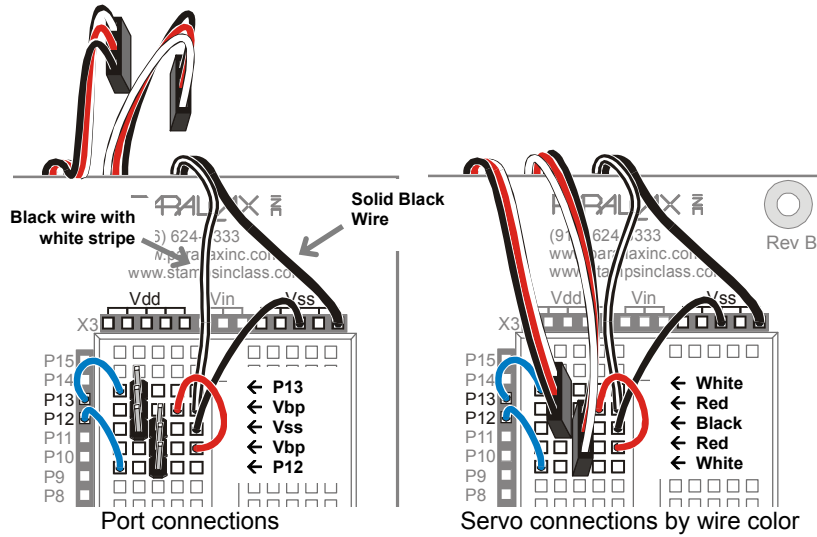


Figure 2-18
Servo
Connection
Wiring
Diagram for
the BASIC
Stamp
HomeWork
Board

2

*Left (build
the servo
ports).*

*Right
(connect
the servos).*

Your setup will then resemble Figure 2-19.

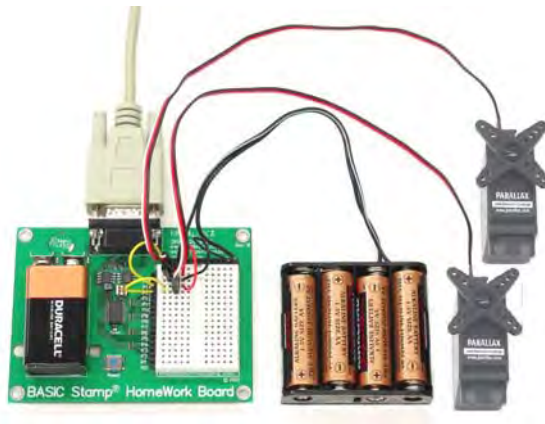
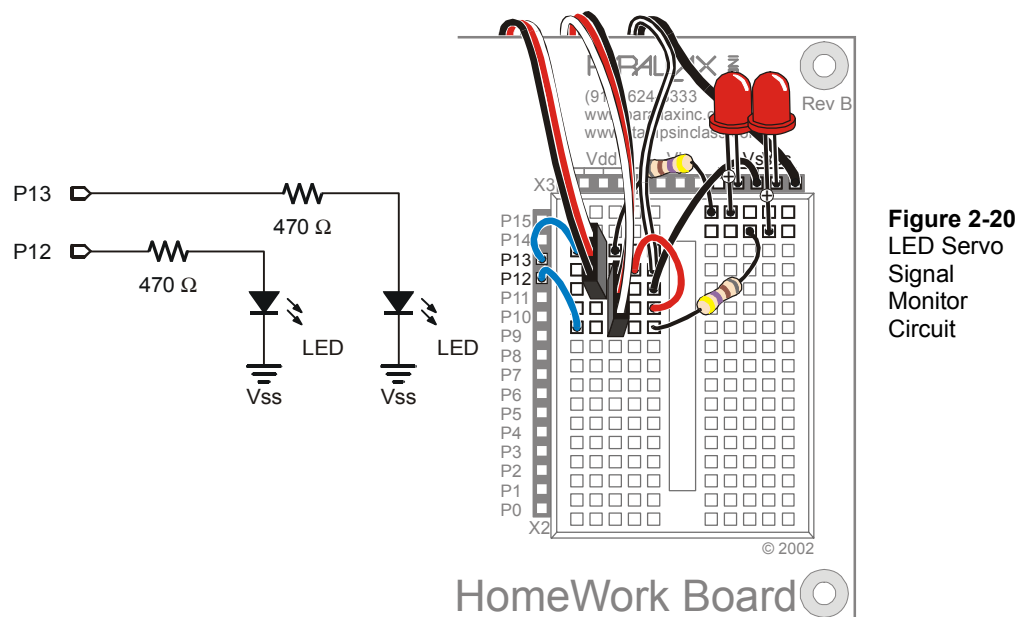


Figure 2-19
Dual Supplies
and Servos
Connected

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



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



Disconnecting Power – Special Instructions for the HomeWork Board

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

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

ACTIVITY #4: CENTERING THE SERVOS

In this activity, you will run a program that sends the servos a signal, instructing them to stay still. Because the servos are not pre-adjusted at the factory, they will instead start

turning. You will then use a screwdriver to adjust them so that they stay still. This is called centering the servos. After the adjustment, you will test the servos to make sure they are functioning properly. The test programs will send signals that make the servos turn clockwise and counterclockwise at various speeds.

Servo Tools and Parts

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



Figure 2-21
Parallax
Screwdriver

Sending the Center Signal

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

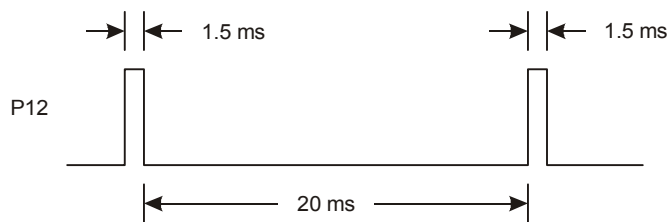


Figure 2-22
Timing Diagram for
CenterServoP12.bs2

*The 1.5 ms pulses
instruct the servo to
remain still.*

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

Figuring out the **PULSOUT** command's **Pin** argument isn't that hard either, it's going to be 12, for I/O pin P12. Next, let's figure out what the **PULSOUT** command's **Duration** argument has to be for 1.5 ms pulses. 1.5 ms is 1.5 thousandths of a second, or 0.0015 s. Remember whatever number is in the **PULSOUT** command's **Duration** argument, multiply that number by 2 μ s (2 millionths of a second = 0.000002 s), and you will know how long

the pulse will last. You can also figure out what the **PULSOUT** command's *Duration* argument has to be if you know how long you want the pulse to last. Just divide 2 μ s into the time you want the pulse to last. With this calculation:

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

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

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

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



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

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



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

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

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

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

Example Program: CenterServoP12.bs2

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

```
' P12 for manual centering.

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

DEBUG "Program Running!"

DO
  PULSOUT 12, 750
  PAUSE 20
LOOP
```

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

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



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



Insert tip of Phillips screwdriver into potentiometer access hole.



Gently turn screwdriver to adjust potentiometer

Figure 2-24
Center Adjusting a Servo

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

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

- ✓ If the servo does not turn, skip to the Your Turn section on page 70 so that you can test and center the other servo that's connected to P13.



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

Your Turn – Centering the Servo Connected to P13

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

Example Program: CenterServoP13.bs2

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

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

DEBUG "Program Running!"

DO
  PULSOUT 13, 750
  PAUSE 20
LOOP
```



Remember to completely disconnect power when you are done.

If you have a Board of Education Rev C.

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

If you have a BASIC Stamp HomeWork Board:

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

ACTIVITY #5: HOW TO STORE VALUES AND COUNT

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

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

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

Using Variables for Storing Values, Math Operations, and Counting

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

variableName* VAR *Size

You can declare four different sizes of variables in PBASIC:

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

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

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

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

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



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

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

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

Example Program: VariablesAndSimpleMath.bs2

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

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

```
' Robotics with the Boe-Bot - VariablesAndSimpleMath.bs2
' Declare variables and use them to solve a few arithmetic problems.

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

value          VAR      Word          ' Declare variables
anotherValue   VAR      Word

value = 500
anotherValue = 2000                    ' Initialize variables

DEBUG ? value
DEBUG ? anotherValue                  ' Display values

value = 10 * anotherValue              ' Perform operations

DEBUG ? value
DEBUG ? anotherValue                  ' Display values again

END
```

How VariablesAndSimpleMath.bs2 Works

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

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



```
anotherValue  VAR    Word
```

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

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

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

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



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

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

```
value = 10 * anotherValue                    ' Perform operations

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

Your Turn – Calculations with Negative Numbers

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

✓ Delete this portion of `VariablesAndSimpleMath.bs2`:

```
value = 10 * anotherValue                    ' Perform operations

DEBUG ? value                                ' Display values again
```

✓ Replace it with the following:

```

value = value - anotherValue      ' Answer = -1500

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

```

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

Counting and Controlling Repetitions

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

FOR Counter = StartValue TO EndValue {STEP StepValue}...NEXT

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

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

```

myCounter      VAR      Word

```

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

```

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

```

Example Program: CountToTen.bs2

✓ Enter, save, and run CountToTen.bs2.

```

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

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

myCounter      VAR      Word

```

```

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

DEBUG CR, "All done!"

END

```

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

You can use different values for the *startValue* and *endValue* arguments.

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

```

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

```

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

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

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

```

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

```

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

ACTIVITY #6: TESTING THE SERVOS

There's one last thing to do before assembling your Boe-Bot, and that's testing the servos. In this activity, you will run programs that make the servos turn at different speeds and directions. By doing this, you will verify that your servos are working properly before you assemble your Boe-Bot.

This is an example of subsystem testing. Subsystem testing is a worthwhile habit to develop, because it isn't any fun to take a robot back apart just to fix a problem that you could have otherwise caught before putting it together!



Subsystem testing is the practice of testing the individual components before they go into the larger device. It's a valuable strategy that can help you win robotics contests. It's also an essential skill used by engineers worldwide to develop everything from toys, cars, and video games to space shuttles and Mars roving robots. Especially in more complex devices, it can become nearly impossible to figure out a problem if the individual components haven't been tested beforehand. In aerospace projects, for example, disassembling a prototype to fix a problem can cost hundreds of thousands, or even millions of dollars. In those kinds of projects, subsystem testing is rigorous and thorough.

Pulse Width Controls Speed and Direction

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

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

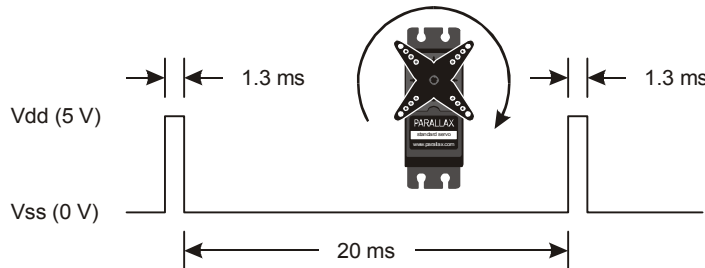


Figure 2-25

A 1.3 ms
Pulse Train
Turns the
Servo Full
Speed
Clockwise



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

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

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

Example Program: `ServoP13Clockwise.bs2`



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

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

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

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

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

DEBUG "Program Running!"

DO
  PULSOUT 13, 650
  PAUSE 20
LOOP
```

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

Example Program: `ServoP12Clockwise.bs2`

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

- ✓ Save `ServoP13Clockwise.bs2` as `ServoP12Clockwise.bs2`.
- ✓ Modify the program by updating the comments and the `PULSOUT` command's *Pin* argument to 12.

- ✓ Run the program and verify that the servo connected to P12 is now rotating between 50 and 60 RPM clockwise.

```
' Robotics with the Boe-Bot - ServoP12Clockwise.bs2
' Run the servo connected to P12 at full speed clockwise.

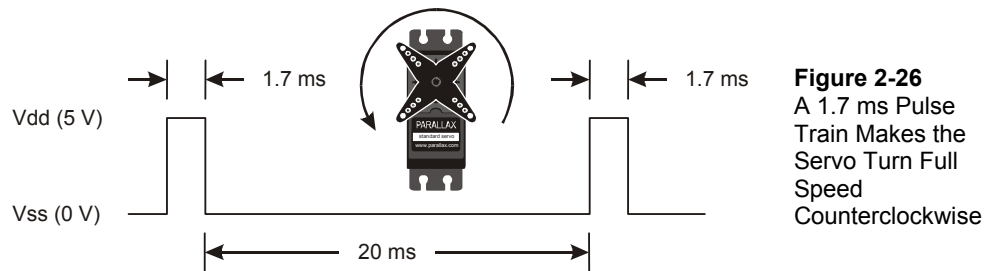
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

DO
  PULSOUT 12, 650
  PAUSE 20
LOOP
```

Example Program: ServoP12Counterclockwise.bs2

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



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

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

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

```

DEBUG "Program Running!"

DO
  PULSOUT 12, 850
  PAUSE 20
LOOP

```

Your Turn – P13Clockwise.bs2

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

Example Program: ServosP13CcwP12Cw.bs2

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

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

```

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

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

DEBUG "Program Running!"

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

```

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

- ✓ Hold your servos together back-to-back and re-run the program.

Your Turn – Adjusting the Speed and Direction

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

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

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

FOR...NEXT to Control Servo Run Time

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

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

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

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

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

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

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

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

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

Example Program: ControlServoRunTimes.bs2

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

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

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

DEBUG "Program Running!"

counter VAR Byte

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

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

END
```

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

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

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

$$\text{Number of pulses} = \text{Time } s / 0.0246s = \text{Time} / 0.0246$$

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

$$\text{Number of pulses} = 3 / 0.0246 = 122$$

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

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

Example Program: BothServosThreeSeconds.bs2

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

✓ Enter, save, and run BothServosThreeSeconds.bs2.

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

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

DEBUG "Program Running!"

counter VAR Byte

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

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

END
```

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

Your Turn – Predict Servo Run Time

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

- ✓ Remember to disconnect power from your system (board and servos) when you are done. That means setting the 3-position switch to position-0 if you have a Board of Education Rev C. If you have a HomeWork Board, disconnect the 9 V battery from the battery clip and remove one battery from the battery pack.

2



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

SUMMARY

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

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

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

Questions

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

PBASIC commands? What are the command and argument that you can adjust to control a continuous rotation servo's speed and direction?

Exercises

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

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

Projects

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

Solutions

- Q1. Instead of holding a certain position like a standard servo, the Parallax Continuous Rotation servos turn a certain direction at a certain speed.
- Q2. A millisecond lasts one thousandth of a second. Millisecond is abbreviated "ms".
- Q3. The **DO...LOOP** command is used to make other PBASIC commands execute over and over.
- Q4. **HIGH** connects I/O pin to Vdd, **LOW** connects I/O pin to Vss.
- Q5. The variable sizes are bit, nib, byte, and word.
 Bit – Stores 0 to 1
 Nib – Stores 0 to 15
 Byte – Stores 0 to 255
 Word – Stores 0 to 65535 or -32768 to +32767
- Q6. Pulse width controls servo speed and direction. As seen on a timing diagram, the pulse width is the high time. In PBASIC, the pulse can be generated with the **PULSOUT** command. The **PULSOUT** command's *Duration* argument adjusts the speed and direction.

E1. **PAUSE 10000**

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

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

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

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

DEBUG "Program Running!"

DO
```



```

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

```

- P2. First, calculate the number of loops needed to get the servos to run for three seconds, for each combination of rotation. As given on page 79, the code overhead is 1.6 ms.

Four combinations (1,2,3,4).

Each combination: Determine **PULSOUT Duration** arguments:

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

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

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

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

1. number of pulses = $3 \text{ s} / 0.025 \text{ s} = 120$
2. number of pulses = $3 \text{ s} / 0.0242 \text{ s} = 123.9 = 124$
3. number of pulses = $3 \text{ s} / 0.0246 \text{ s} = 121.9 = 122$
4. number of pulses = $3 \text{ s} / 0.0246 \text{ s} = 121.9 = 122$

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

```

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

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

```

```

DEBUG "Program Running!"

counter      VAR      Word

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

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

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

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

END

```

Chapter 3: Assemble and Test Your Boe-Bot

3

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

Activity	Summary

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

ACTIVITY #1: ASSEMBLING THE BOE-BOT

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

Servo Tools and Parts

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

Tools

- (1) Parallax screwdriver (Phillips #1 point screwdriver $\frac{1}{8}$ " (3.18 mm) shaft)
- (1) $\frac{1}{4}$ " Combination wrench (Optional)
- (1) Needle-nose pliers (Optional)



Figure 3-1
Boe-Bot
Assembly
Tools

Mounting the Topside Hardware

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

Parts List:

See Figure 3-2.

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

Instructions:

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

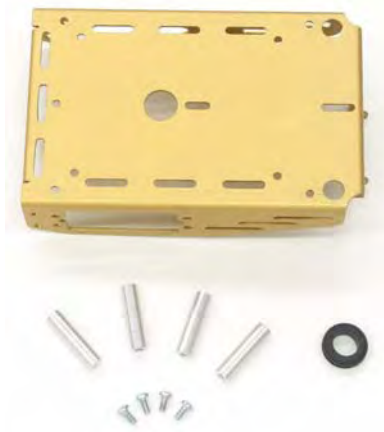


Figure 3-2
Chassis and
Topside
Hardware

*Parts (left);
assembled
(right).*

3



Boe-Bot Parts - The parts for the Boe-Bot are either included in the Boe-Bot full kit or in a combination of the Board of Education Full Kit and Robotics Parts Kit. See Appendix E: Boe-Bot Parts Lists for more information.

Removing the Servo Horns

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

Parts List:

See Figure 3-3.

(2) Parallax Continuous
Rotation servos, previously
centered

Instructions:

- ✓ Use a Phillips screwdriver to remove the screws that hold the servo control horns on the output shafts.
- ✓ Pull each horn upwards and off the servo output shaft.
- ✓ Save the screws; they will be used in a later step.



Stop!

✓ Before this next step, you must have completed these activities from Chapter 2: Your Boe-Bot's Servo Motors

- Activity #3: Connecting the Servo Motors
- Activity #4: Centering the Servos

Mounting the Servos on the Chassis

Parts List:

See Figure 3-4.

- (1) Boe-Bot chassis (partially assembled)
- (2) Parallax Continuous Rotation servos
- (8) Pan Head Screws, 3/8" 4-40
- (8) Nuts, 4-40

Instructions:

- ✓ Attach the servos to the chassis using the Phillips screws and nuts. Note that for best performance, you must place the face of each servo through the rectangular window from inside the chassis rather than dropping them in from the outside.
- ✓ Use pieces of masking tape to label the servos left (L) and right (R).

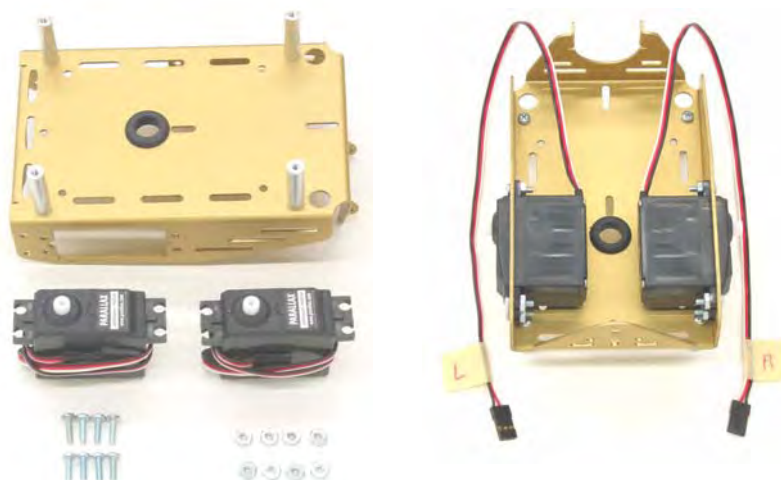


Figure 3-4
Mounting the
Servos on the
Chassis

*Parts (left);
assembled
(right).*

Mounting the Battery Pack

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

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

See Figure 3-5 (left side).

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

Parts List for Boe-Bot with a HomeWork Board:

See Figure 3-5 (right side).

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

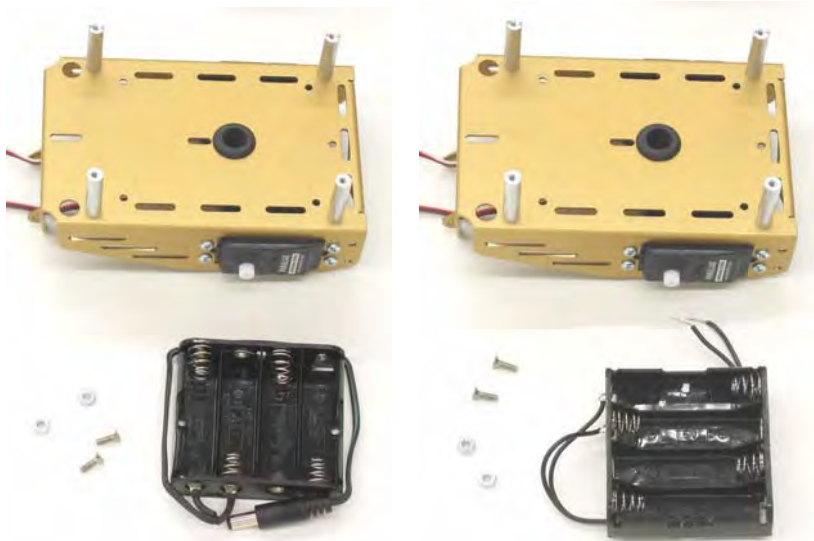


Figure 3-5
Battery Pack
Mounting
Hardware

For use with the Board of Education

For use with the HomeWork Board

Instructions:

- ✓ Use the flathead screws and nuts to attach the battery pack to underside of the Boe-Bot chassis as shown on the left side of Figure 3-6.
- ✓ Make sure to insert the screws through the battery pack, then tighten down the nuts on the topside of the chassis.
- ✓ As shown on the right side of Figure 3-6, pull the battery pack's power cord through the hole with the rubber grommet in the center of the chassis.
- ✓ Pull the servo lines through the same hole.
- ✓ Arrange the servo lines and supply cable as shown.

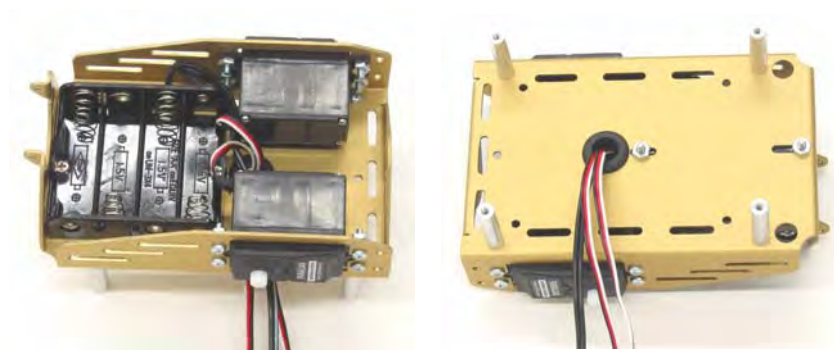


Figure 3-6
Battery Pack
Installed

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

3

Mounting the Wheels

Parts List:

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



Figure 3-7
Wheel
Hardware

Instructions:

The left side of Figure 3-8 shows the Boe-Bot's tail wheel mounted on the chassis. The tail wheel is merely a plastic ball with a hole through the center. A cotter pin holds it to the chassis and functions as an axle for the wheel.

- ✓ Line the hole in the tail wheel up with the holes in the tail portion of the chassis.
- ✓ Run the cotter pin through all three holes (chassis left, tail wheel, chassis right).
- ✓ Bend the ends of the cotter pin apart so that it can't slide back out of the hole.

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

- ✓ Stretch each rubber band tire and seat it on the outer edge of each wheel.

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

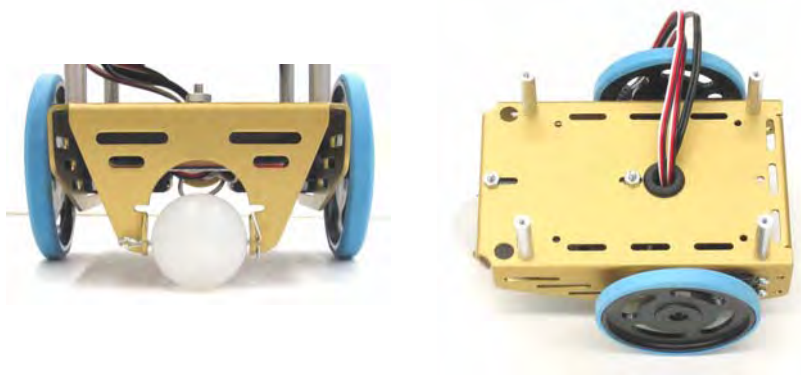


Figure 3-8
Mounting the
Wheels

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

Attaching Board to Chassis

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

See left side of Figure 3-9.

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

Parts List for a Boe-Bot with a HomeWork Board:

See right side of Figure 3-9.

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

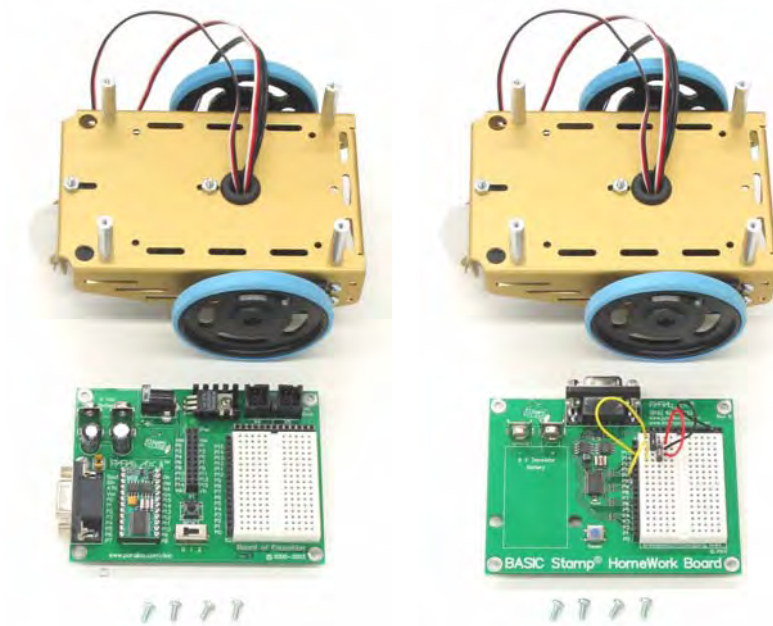


Figure 3-9
Boe-Bot Chassis
and Boards

With the Board of Education Rev C

With the HomeWork Board

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

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

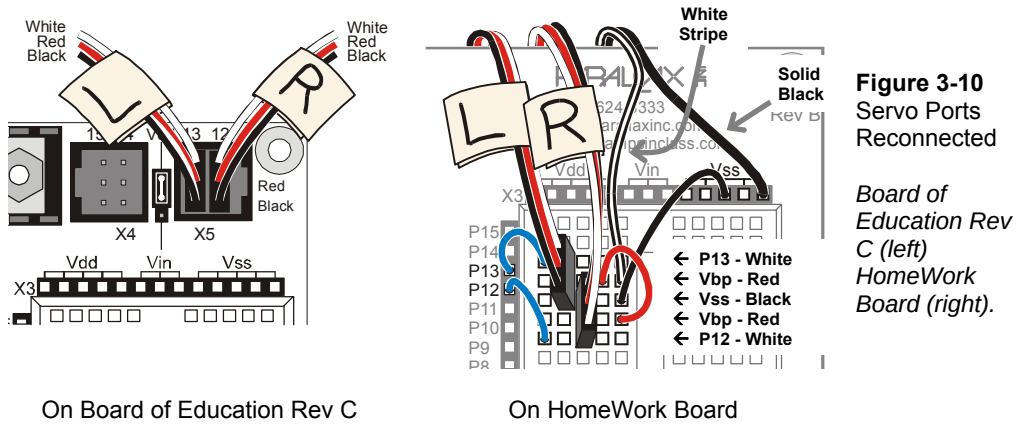


Figure 3-11 shows the Boe-Bot chassis with their respective boards attached.

- ✓ Set the board on the four standoffs so that they line up with the four holes on the outer corner of the board.
- ✓ Make sure the white breadboard is closer to the drive wheels, not the tail wheel.
- ✓ Attach the board to the standoffs with the pan head screws.

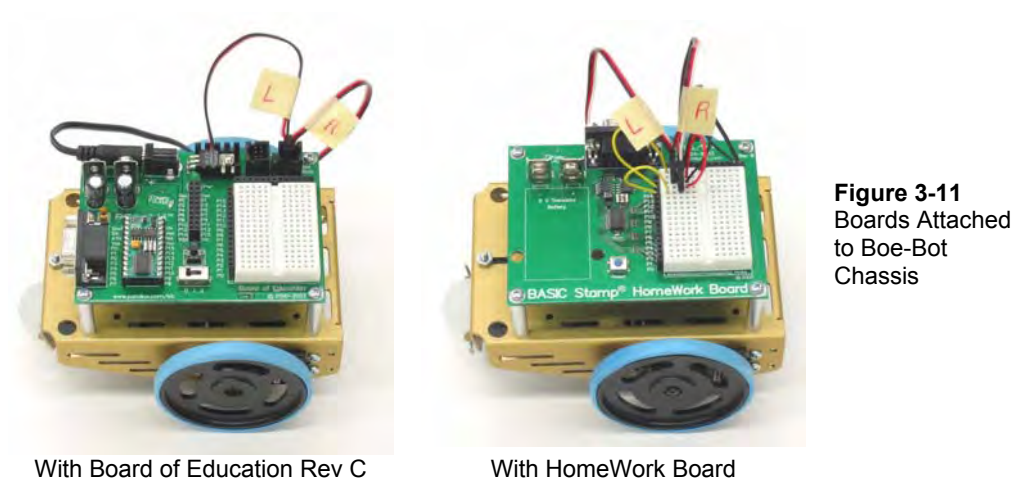
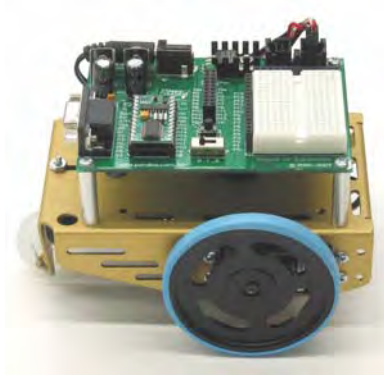


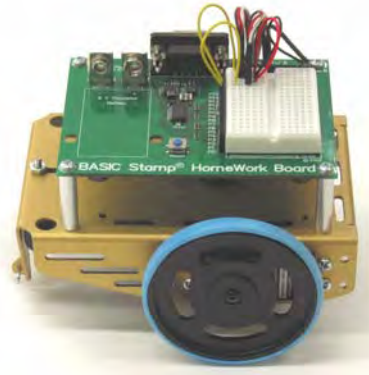
Figure 3-12 shows assembled Boe-Bots, the left built with a Board of Education Rev C and the right built with a HomeWork Board.

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

3



With Board of Education Rev C



With HomeWork Board

Figure 3-12
Assembled
Boe-Bots

ACTIVITY #2: RE-TEST THE SERVOS

In this activity, you will test to make sure that the electrical connections between your board and the servos are correct. Figure 3-13 shows your Boe-Bot's front, back, left, and right. We need to make sure that the servo on the right turns when it receives pulses from P12 and that the servo on the left turns when it receives pulses from P13.

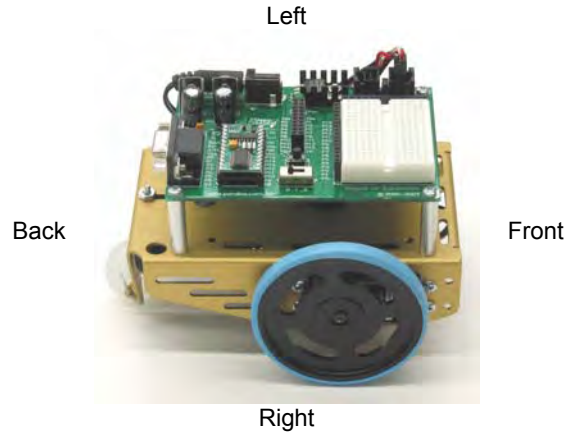


Figure 3-13
Your Boe-Bot
robot's Front, Back,
Left, and Right

Testing the Right Wheel

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

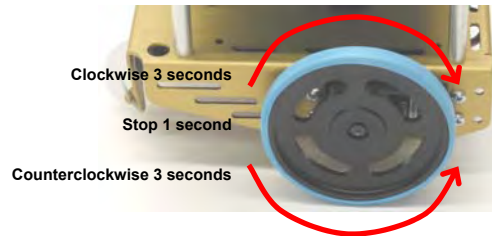


Figure 3-14
Testing the Right
Wheel

Example Program: RightServoTest.bs2

- ✓ Set the Boe-Bot on its nose so that the drive wheels are suspended above ground.
- ✓ Reload the batteries into the battery pack.
- ✓ If you have a Board of Education Rev C, set the 3-position switch to position-2.
- ✓ If you have a BASIC Stamp HomeWork Board, connect the 9 V battery to the battery clip.
- ✓ Enter, save, and run RightServoTest.bs2.
- ✓ Verify that the right wheel turns clockwise for three seconds, stops for one second, then turns counterclockwise for three seconds.

- ✓ If the right wheel/servo does not behave as predicted, see the Servo Trouble Shooting section. It comes right after RightServoTest.bs2.
- ✓ If the right wheel/servo does behave properly, then move on to the Your Turn section, where you will test the left wheel.

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

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

DEBUG "Program Running!"

counter      VAR      Word

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

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

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

END
```



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

The servo doesn't turn at all.

- √ If you are using a Board of Education Rev C, make sure the 3-position switch is set to position-2. You can then re-run the program by pressing and releasing the Reset button.
- √ If you are using a BASIC Stamp HomeWork Board, make sure the battery pack has batteries.
- √ Double-check your servo connections using Figure 3-10 on page 100 as a guide. If you are using a HomeWork Board, you may also want to take a second look at Figure 2-18 on page 65.
- √ Check and make sure you entered the program correctly.

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

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

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

The wheel does not fully stop; it turns slowly.

This means that the servo may not be exactly centered. You can often adjust the program to make the servo stay still. You can do this by modifying the `PULSOUT 12, 750` command.

- √ If the wheel turns slowly counterclockwise, use a value that's a little smaller than 750.
- √ If it's turning clockwise, use a value that's a little larger than 750.
- √ If you can find a value between 740 and 760 that fully stops your servo, then make sure to use it anywhere you see the command `PULSOUT 12, 750`.

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

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

- √ Remove the wheels, un-mount the servos and repeat the exercise in Chapter 2 Activity #4: Centering the Servos.

Your Turn – Testing the Left Wheel

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

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

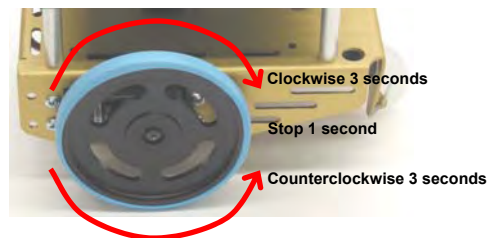


Figure 3-15
Testing the Left
Wheel

- ✓ Save RightServoTest.bs2 as LeftServoTest.bs2.
- ✓ Change the three **PULSOUT** commands so that they read **PULSOUT 13** instead of **PULSOUT 12**.
- ✓ Save and then run the program.
- ✓ Verify that it makes the left servo turn clockwise for 3 seconds, stops for 1 second, then makes the servo turn counterclockwise for 3 seconds.
- ✓ If the left wheel/servo does not behave as predicted, see the Servo Trouble Shooting section on page 104.
- ✓ If the left wheel/servo does behave properly, then your Boe-Bot is functioning properly, and you are ready to move on to the next activity.

ACTIVITY #3: START/RESET INDICATOR CIRCUIT AND PROGRAM

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

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

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

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



BASIC Stamp HomeWork Board Special Instructions

Although the reset indicator will tell you when the 9 V battery supplying the BASIC Stamp is running low, it will not tell you when the servo supply (the battery pack) is running low.

You can always tell when your battery pack is running low because the servos will gradually move slower and slower during normal operation. When you observe this symptom, replace the dead batteries with new 1.5 V alkaline batteries.

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

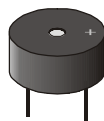
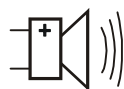


Figure 3-16
Piezospeaker

3



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

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

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

Parts Required

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



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

Building the Start/Reset Indicator Circuit

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



Always disconnect power before building or modifying circuits!

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

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

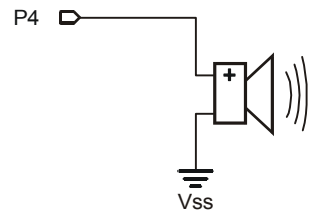


Figure 3-17
Program Start/Reset
Indicator Circuit

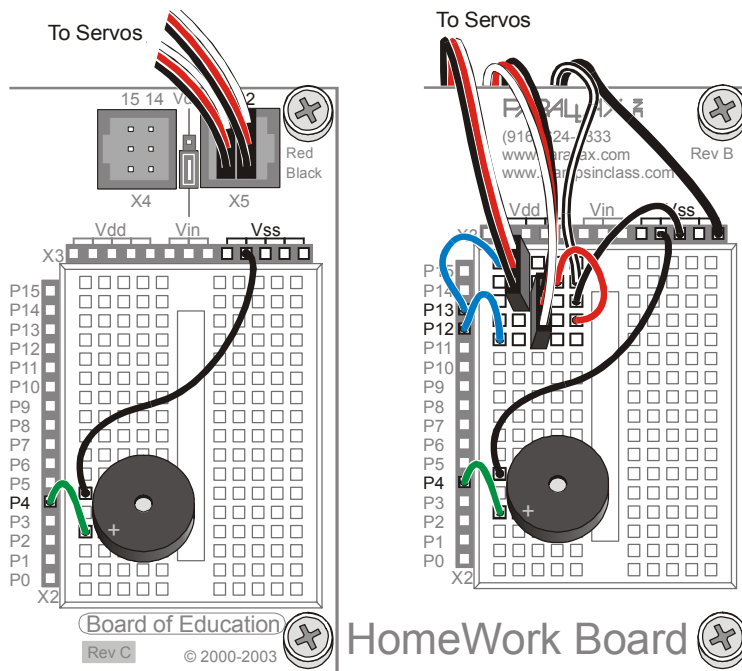


Figure 3-18
Wiring Diagrams for
the Program
Start/Reset Indicator
Circuit

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



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

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

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

Programming the Start/Reset Indicator

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

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

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

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

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



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

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

Example Program: StartResetIndicator.bs2

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

- ✓ Reconnect power to your board.
- ✓ Enter, save, and run StartResetIndicator.bs2.
- ✓ Verify that the piezospeaker made a clearly audible tone for two seconds before the "Waiting for reset..." messages started to display in the Debug Terminal.

- ✓ If you did not hear a tone, check your wiring and code for errors. Repeat until you get an audible tone from your speaker.
- ✓ If you did hear an audible tone, try simulating the brownout condition by pressing and releasing the Reset button on your board. Verify that the piezospeaker makes a clearly audible tone after each reset.
- ✓ Also try disconnecting and reconnecting your battery supply, and verify that this results in the reset warning tone as well.

```
' Robotics with the Boe-Bot - StartResetIndicator.bs2
' Test the piezospeaker circuit.

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

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

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

How StartResetIndicator.bs2 Works

StartResetIndicator.bs2 starts by displaying the message “Beep!!!” Then, immediately after printing the message, the **FREQOUT** command plays a 3 kHz tone on the piezoelectric speaker for 2 s. Because the instructions are executed so rapidly by the BASIC Stamp, it should seem as though the message is displayed at the same instant the piezospeaker starts to play the tone.

When the tone is done, the program enters a **DO...LOOP**, displaying the same “Waiting for reset...” message over and over again. Each time the reset button on the Board of Education is pressed or the power is disconnected and reconnected, the program starts over again, with the “Beep!!!” message and the 3 kHz tone.

Your Turn - Adding StartResetIndicator.bs2 to a Different Program

The lines of code in the battery indicator program will be used at the beginning of every example program from here onward. You could consider it part of the “initialization routine” or “boot routine” for every Boe-Bot program.



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

3

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

ACTIVITY #4: TESTING SPEED CONTROL WITH THE DEBUG TERMINAL

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

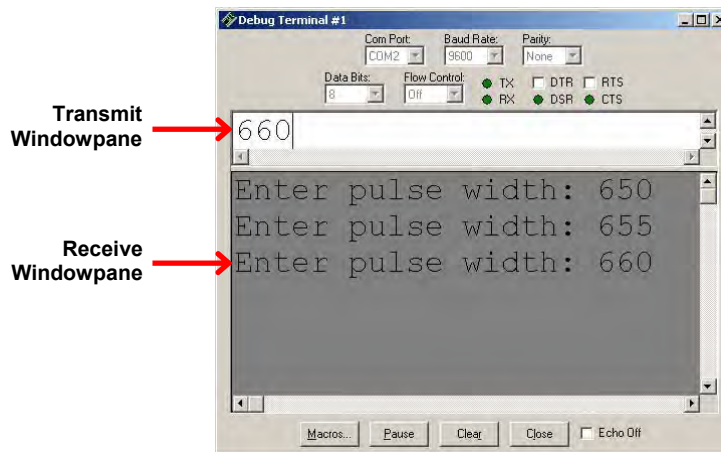


Figure 3-19
Debug Terminal
Windowpanes



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

Using the DEBUGIN Command

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

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

```
pulseWidth    VAR    Word
```

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

```
DEBUGIN DEC pulseWidth
```

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

```
PULSOUT 12, pulseWidth
```

Example Program: TestServoSpeed.bs2

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

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

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

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

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

- ✓ Place a mark on the wheel so that you can see how far it turns in 6 seconds.
- ✓ Use the Debug Terminal to test how far the wheel turns for each of these pulse widths: 650, 660, 670, 680, 690, 700, 710, 720, 730, 740, 750, 760, 770, 780, 790, 800, 810, 820, 830, 840, 850
- ✓ For each pulse width, multiply the number of turns by 10 to get the RPM. For example, if the wheel makes 3.65 full turns, it was rotating at 36.5 RPM.
- ✓ Explain in your own words how you can use pulse width to control Continuous Rotation servo speed.

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

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

counter      VAR      Word
pulseWidth   VAR      Word
pulseWidthComp VAR    Word

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

DO

  DEBUG "Enter pulse width: "

  DEBUGIN DEC pulseWidth

  pulseWidthComp = 1500 - pulseWidth
```

```

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

```

How TestServoSpeed.bs2 Works

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

```

counter      VAR      Word
pulseWidth   VAR      Word
pulseWidthComp VAR    Word

```

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

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

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

```

DEBUG "Enter pulse width: "

DEBUGIN DEC pulseWidth

```

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

This next command takes the pulse width you entered, and calculates a pulse width that will make 1500 when the two are added together. If you enter a pulse width of 650, `pulseWidthComp` will be 850. If you enter a pulse width of 850, `pulseWidthComp` will

be 650. If you enter a pulse width of 700, `pulseWidthComp` will be 800. Try a few other examples. They will all add up to 1500.

```
pulseWidthComp = 1500 - pulseWidth
```

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

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

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

Figure 3-20 shows an example of a transfer curve for a continuous rotation servo. The horizontal axis shows the pulse width in ms, and the vertical axis shows the rotational velocity in RPM. In this graph, clockwise is negative and counterclockwise is positive. This particular servo's transfer curve ranges from about -48 RPM to 48 RPM over the range of test pulse widths that range from 1.3 ms to 1.7 ms.

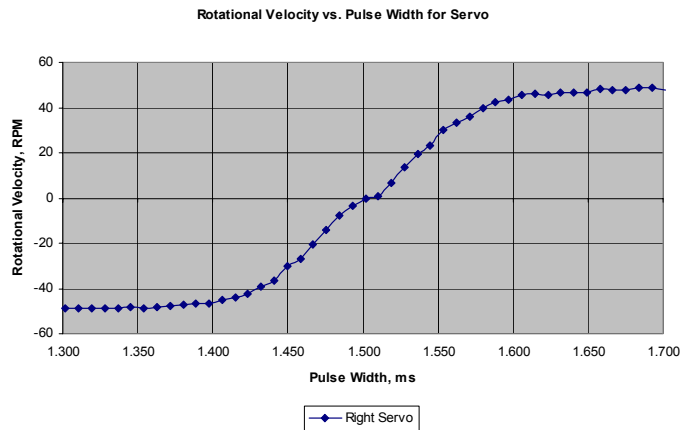


Figure 3-20
Transfer Curve
Example for
Parallax Servo

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

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

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

$$Duration = 650 \times 2 \mu s$$

$$= 650 \times 0.000002 s$$

$$= 0.0013 s$$

$$= 1.3 ms$$

$$Duration = 655 \times 2 \mu s$$

$$= 655 \times 0.000002 s$$

$$= 0.00131 s$$

$$= 1.31 ms$$

$$Duration = 660 \times 2 \mu s$$

$$= 660 \times 0.000002 s$$

$$= 0.00132 s$$

$$= 1.32 ms$$

- ✓ Mark your right wheel so that you have a reference point to count the revolutions.
- ✓ Run TestServoSpeed.bs2.
- ✓ Click the Debug Terminal's Transmit windowpane.
- ✓ Enter the value 650.
- ✓ Count how many turns the wheel made.

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

- ✓ Multiply this value by 10 and enter the result into Table 3-1 next to the 1.3 ms entry.
- ✓ Enter the value 655.
- ✓ Count how many turns the wheel made.
- ✓ Multiply this value by 10 and enter the result into Table 3-1 next to the 1.31 ms entry.
- ✓ Keep increasing your durations by 5 (0.01 ms) until you are up to 850 (1.7 ms).
- ✓ Use a spreadsheet, calculator, or graph paper to graph the data.
- ✓ Repeat this process for your other servo.

3

You can repeat these measurements for the left wheel. You will have to modify the **PULSOUT** commands so that pulses with a duration of **pulseWidth** are sent to P13 and pulses with a duration of **pulseWidthComp** are sent to P12.

SUMMARY

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

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

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

Questions

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

Exercises

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

3

Projects

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

Solutions

- Q1. Symptoms include erratic behavior such as going in unexpected directions or doing a confused dance.
- Q2. A **FREQOUT** command at the beginning of all Boe-Bot programs causes the piezospeaker to play a tone. This tone will therefore occur every time an accidental reset happens due to brownout conditions.
- Q3. A reset is when the power is interrupted and the BASIC Stamp program starts running again from the beginning of the program.
- Q4. An initialization routine consists of the lines of code that are used at the beginning of the program. These lines of code run each time the program starts from the beginning.
- Q5. 1. The servo lines P12 and P13 are swapped.
 2. One or both servos is plugged in backwards, so that the white-red-black color coding is incorrect.
 3. The power switch is not on position-2.
 4. The 9V or AA batteries are not installed.
 5. The servo centering potentiometer is out of adjustment.
- Q6. The **PULSOUT** commands must be changed to read **PULSOUT 13** instead of **PULSOUT 12**.
- E1. The key is to modify the **FREQOUT** command used for the StartResetIndicator.bs2 program, that is, **FREQOUT, 4, 2000, 3000**. For example: **FREQOUT, 4, 500, 3500** would work.
- E2. **FREQOUT 4, 100, 4000**
- P1. The key to solving this program is to add the line from Exercise 1 above the **END** command in the RightServoTest.bs2 program.

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

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

counter      VAR      Word
```



```

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

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

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

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

FREQOUT 4, 500, 3500      ' Signal end of program

END

```

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

```

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

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

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

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

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

  DEBUG "Enter number of pulses: "
  DEBUGIN DEC pulseCount

  FOR counter = 1 TO pulseCount          ' Send specific number of pulses

```

```
PULSOUT 13, ltPulseWidth      ' Left servo motion
PULSOUT 12, rtPulseWidth      ' Right servo motion
PAUSE 20
NEXT
LOOP
```

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

Chapter 4: Boe-Bot Navigation

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

4

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

Activity Summary

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

ACTIVITY #1: BASIC BOE-BOT MANEUVERS

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

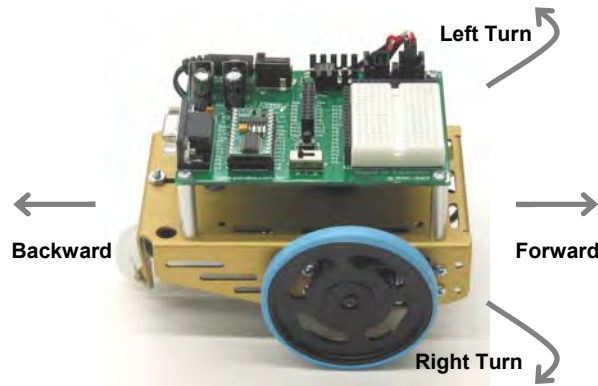


Figure 4-1
Your Boe-Bot and
Driving Directions

Moving Forward

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

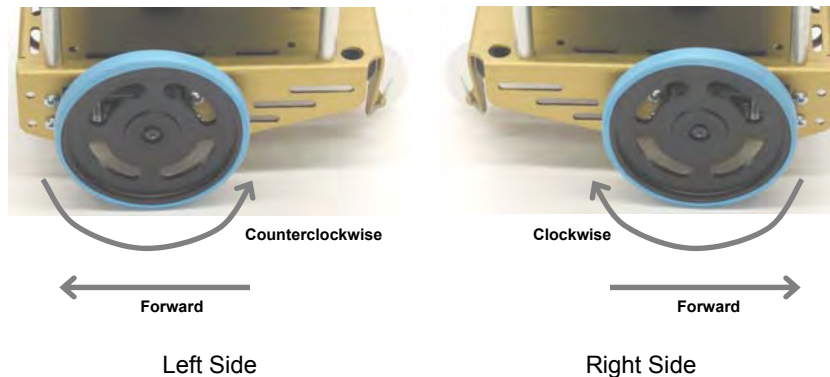


Figure 4-2
Wheel
Rotation for
Forward
Motion

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

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

Example Program: BoeBotForwardThreeSeconds.bs2

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

4

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

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

DEBUG "Program Running!"

counter      VAR      Word

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

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

    PULSOUT 13, 850
    PULSOUT 12, 650
    PAUSE 20

NEXT

END
```

How BoeBotForwardThreeSeconds.bs2 Works

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

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

```
counter VAR Word
```

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

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

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

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

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

Your Turn – Adjusting Distance and Speed

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

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

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


```
PULSOUT 13, 780
PULSOUT 12, 720
```
- ✓ Run the program, and verify that your Boe-Bot moves slower.

Moving Backward, Rotating, and Pivoting

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

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

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

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

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

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

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

Example Program: ForwardLeftRightBackward.bs2

✓ Enter, save, and run ForwardLeftRightBackward.bs2.



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

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

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

DEBUG "Program Running!"

counter      VAR      Word

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

FOR counter = 1 TO 64          ' Forward
```

```

PULSOUT 13, 850
PULSOUT 12, 650
PAUSE 20

NEXT

PAUSE 200

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

    PULSOUT 13, 650
    PULSOUT 12, 650
    PAUSE 20

NEXT

PAUSE 200

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

    PULSOUT 13, 850
    PULSOUT 12, 850
    PAUSE 20

NEXT

PAUSE 200

FOR counter = 1 TO 64                                ' Backward

    PULSOUT 13, 650
    PULSOUT 12, 850
    PAUSE 20

NEXT

END

```

Your Turn - Pivoting

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

```

PULSOUT 13, 750
PULSOUT 12, 650

```

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


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

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

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

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

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

4

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

ACTIVITY #2: TUNING THE BASIC MANEUVERS

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

Straightening the Boe-Bot's Path

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

Example Program: BoeBotForwardTenSeconds.bs2

- ✓ Open BoeBotForwardThreeSeconds.bs2.
- ✓ Rename and save it as BoeBotForwardTenSeconds.bs2.

✓ Change the **EndValue** of the **FOR counter** from 122 to 407, so it reads like this:

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

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

DEBUG "Program Running!"

counter      VAR      Word

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

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

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

NEXT

END
```

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

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



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

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

Remember that servo speed is determined by the **PULSOUT** command's *Duration* argument. The closer the *Duration* is to 750, the slower the servo turns. This means you should change the 650 in the command **PULSOUT 12,650** to something a little closer to 750. If the Boe-Bot is only just a little off course, maybe **PULSOUT 12,663** will do the trick. If the servos are severely mismatched, maybe it needs to be **PULSOUT 12,690**.

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



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

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

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

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

Tuning the Turns

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

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

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

PULSOUT 13, 650
```

```
PULSOUT 12, 650
PAUSE 20
```

```
NEXT
```

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

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

Your Turn - 90° Turns

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



Carpeting can cause navigation errors. If you are running your Boe-Bot on carpeting, don't expect perfect results! A carpet is a bit like a golf green – the way the carpet pile is inclined can affect the way your Boe-Bot travels, especially over long distances. For more precise maneuvers, use a smooth surface.

ACTIVITY #3: CALCULATING DISTANCES

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

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

“Are we there yet?”

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

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

Example – Time for English Distance

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

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

Example – Time for Metric Distance

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

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

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

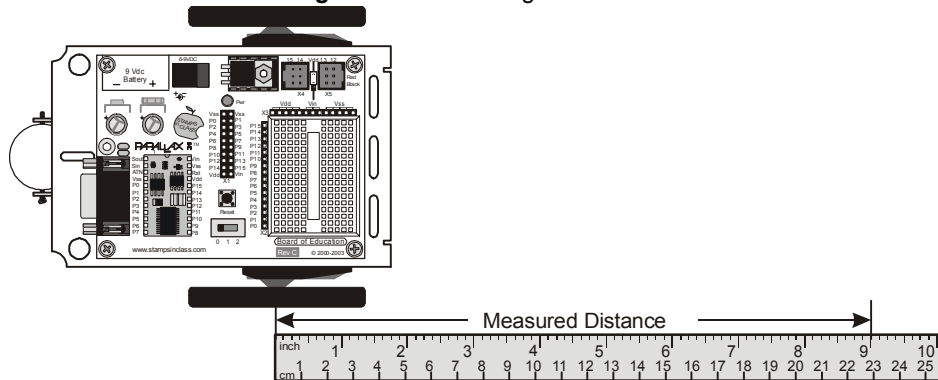
$$\text{servo run time} = \frac{\text{Boe - Bot distance}}{\text{Boe - Bot speed}}$$

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

✓ Enter, save, and run ForwardOneSecond.bs2.

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

Figure 4-3: Measuring Boe-Bot Distance



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

Example Program: ForwardOneSecond.bs2

```
' Robotics with the Boe-Bot - ForwardOneSecond.bs2
' Make the Boe-Bot roll forward for one second.

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

DEBUG "Program Running!"

counter      VAR      Word

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

FOR counter = 1 TO 41

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

NEXT

END

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

4



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

Example – Time for 20 in

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

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

Example – Time for 51 cm

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

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

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

Since you know the amount of time you want your Boe-Bot to move forward (2.22 s) and the number of pulses the BASIC Stamp sends to the servos each second (40.65 pulses/s),

you can use these values to calculate how many pulses to send to the servos. This is the number you will have to use for your **FOR...NEXT** loop's **EndValue** argument.

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

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

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

Example – Time for 20 in

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

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

Example – Time for 51 cm

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

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

Your Turn – Your Boe-Bot's Distance

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

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

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

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



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

You can increase the accuracy of your Boe-Bot distances with devices called encoders, which count the holes in the Boe-Bot's wheels as they pass. Encoders hardware, documentation and example programs are available in the Robotics → Accessories page at www.parallax.com.

ACTIVITY #4: MANEUVERS – RAMPING

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

Programming for Ramping

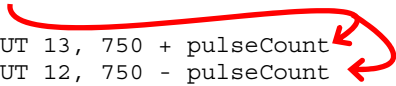
The key to ramping is to use variables along with constants for the `PULSOUT` command's *Duration* argument. Figure 4-4 shows a `FOR...NEXT` loop that can ramp the Boe-Bot's speed from full stop to full speed ahead. Each time the `FOR...NEXT` loop repeats itself, the `pulseCount` variable increases by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

```

pulseCount    VAR      Word

FOR pulseCount = 1 TO 100
    PULSOUT 13, 750 + pulseCount
    PULSOUT 12, 750 - pulseCount
    PAUSE 20
NEXT

```



1, 2, 3,
...100

Figure 4-4
Ramping
Example

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

Example Program: StartAndStopWithRamping.bs2

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

```

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

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

DEBUG "Program Running!"

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

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

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

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

' Ramp up forward.

FOR pulseCount = 1 TO 100
    PULSOUT 13, 750 + pulseCount
    PULSOUT 12, 750 - pulseCount
    PAUSE 20
NEXT

```

```

NEXT

' Continue forward for 75 pulses.

FOR pulseCount = 1 TO 75
    PULSOUT 13, 850
    PULSOUT 12, 650
    PAUSE 20
NEXT

' Ramp down from going forward to a full stop.

FOR pulseCount = 100 TO 1
    PULSOUT 13, 750 + pulseCount
    PULSOUT 12, 750 - pulseCount
    PAUSE 20
NEXT

END

```

Your Turn

You can also create routines to combine ramping up or down with the other maneuvers. Here's an example of how to ramp up to full speed going backward instead of forward. The only difference between this routine and the forward ramping routine is that the value of `pulseCount` is subtracted from 750 in the `PULSOUT 13` command, where before it was added. Likewise, `pulseCount` is added to the value of 750 in the `PULSOUT 12` command, where before it was subtracted.

```

' Ramp up to full speed going backwards

FOR pulseCount = 1 TO 100

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

NEXT

```

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

```

' Ramp up right rotate.

FOR pulseCount = 0 TO 30

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

NEXT

' Ramp down right rotate

FOR pulseCount = 30 TO 0

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

NEXT

```

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

ACTIVITY #5: SIMPLIFY NAVIGATION WITH SUBROUTINES

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

Inside the Subroutine

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

Figure 4-5 shows part of a PBASIC program that contains a subroutine call and a subroutine. The subroutine call is the `GOSUB My_Subroutine` command. The actual subroutine is everything from the `My_Subroutine:` label through the `RETURN` command. Here's how it works. When the program gets to the `GOSUB My_Subroutine` command, it looks for the `My_Subroutine:` label. As shown by arrow (1), the program jumps to the `My_Subroutine:` label and starts executing commands. The program keeps going down line by line from the label, so you'll see the message "Command in subroutine" in your Debug Terminal. `PAUSE 1000` causes a one second pause. Then, when the program gets to the `RETURN` command, arrow (2) shows how it jumps back to the command immediately after the `GOSUB` command. In this case, it's a `DEBUG` command that displays the message "After subroutine".

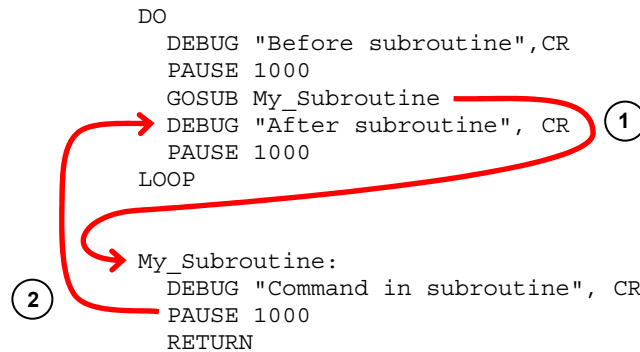


Figure 4-5
Subroutine
Basics

Example Program – OneSubroutine.bs2

✓ Enter, save, and run OneSubroutine.bs2

```

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

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

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

My_Subroutine:

```

```

DEBUG "Command in subroutine", CR
PAUSE 1000
RETURN

```

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

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

Example Program – TwoSubroutines.bs2

- ✓ Enter, save, and run TwoSubroutines.bs2

```

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

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

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

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

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

```

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

Example Program – MovementsWithSubroutines.bs2

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

```
' Robotics with the Boe-Bot - MovementsWithSubroutines.bs2
' Make forward, left, right, and backward movements in reusable subroutines.

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

DEBUG "Program Running!"

counter          VAR      Word

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

GOSUB Forward
GOSUB Left
GOSUB Right
GOSUB Backward

END

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

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

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

```

    PAUSE 200
    RETURN

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

```

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

Example Program – `MovementsWithVariablesAndOneSubroutine.bs2`

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

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

' Forward full speed	' Ramp down from full speed backwards
FOR counter = 1 TO 64	FOR pulseCount = 100 TO 1
PULSOUT 13, 850	PULSOUT 13, 750 - pulseCount
PULSOUT 12, 650	PULSOUT 12, 750 + pulseCount
PAUSE 20	PAUSE 20
NEXT	NEXT

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

✓ Enter, save, and run MovementWithVariablesAndOneSubroutine.bs2.

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

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

DEBUG "Program Running!"

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

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

' Forward

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

' Left turn

pulseLeft = 650: pulseRight = 650: pulseCount = 24: GOSUB Navigate

' Right turn

pulseLeft = 850: pulseRight = 850: pulseCount = 24: GOSUB Navigate

' Backward

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

END

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

Did your Boe-Bot perform the familiar forward-left-right-backward sequence? This program may be difficult to read at first, because the instructions are arranged in a new way. Instead of having each variable statement and each **GOSUB** command on a different line, they are grouped together on the same line and separated by colons. Here, the

colons function the same as a carriage return to separate each PBASIC instruction. Using colons this way allows all of the new variable values for a given maneuver to be stored together, and on the same line as the subroutine call.

Your Turn

Here is your "dead reckoning" contest mentioned earlier.

- ✓ Modify `MovementWithVariablesAndOneSubroutine.bs2` to make your Boe-Bot drive in a square, facing forwards on the first two sides and backwards on the second two sides. Hint: you will need to use your own `PULSOUT EndValue` argument that you determined in Activity #2, page 132.

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

When you download PBASIC program to your BASIC Stamp, the BASIC Stamp Editor converts your program to numeric values called tokens. These tokens are what the BASIC Stamp uses as instructions for executing the program. They are stored in one of the two smaller black chips on top of your BASIC Stamp, the one labeled "24LC16B." This chip is a special type of computer memory called EEPROM, which stands for electrically erasable programmable read only memory (EEPROM). The BASIC Stamp's EEPROM can hold 2048 bytes (2 kB) of information. What's not used for program storage (which builds from address 2047 toward address 0) can be used for data storage (which builds from address 0 toward address 2047).



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

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

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

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



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

4

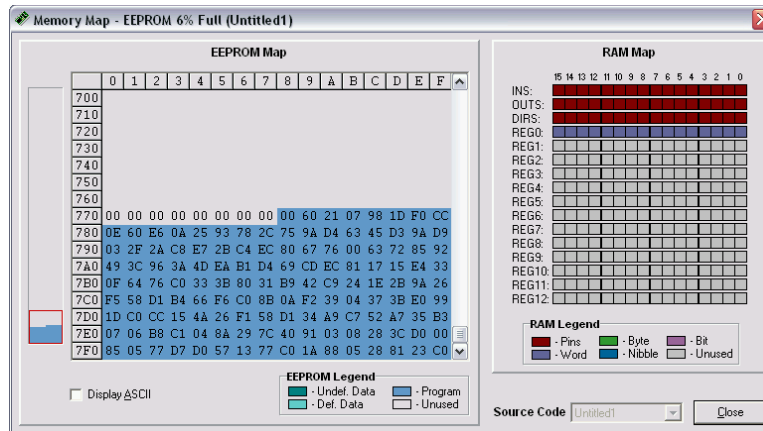


Figure 4-6
BASIC Stamp
Memory Map

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

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

EEPROM Navigation

Up to this point we have tried three different programming approaches to make your Boe-Bot drive forward, turn left, turn right, and drive back again. Each technique has its merits, but all would be cumbersome if you wanted your Boe-Bot to execute a longer, more complex set of maneuvers. The upcoming program examples will use the now-

familiar code blocks in subroutines for each basic maneuver. Each maneuver is given a one-letter code as a reference. Long lists of these code letters can be stored in EEPROM and then read and decoded during program execution. This avoids the tedium of repeating long lists of subroutines, or having to change the variables before each **GOSUB** command.

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

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

```
DATA          "FLFFRBLBBQ"
```

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

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

The **address** variable is the location of each byte in EEPROM that is holding a code letter. The **instruction** variable will hold the actual value of that byte, our code letter. Notice that each time through the loop, the value of the **address** variable is increased by one. This will allow each letter to be read from consecutive bytes in the EEPROM, starting at address 0.

The **DO...LOOP** command has optional conditions that are handy for different circumstances. The **DO UNTIL (condition)...LOOP** allows the loop to repeat until a certain condition occurs. **DO WHILE (condition)...LOOP** allows the loop to repeat only

while a certain condition exists. Our example program will use `DO...LOOP UNTIL (condition)`. In this case, it causes the `DO...LOOP` to keep repeating until the character “Q” is read from EEPROM.

A `SELECT...CASE...ENDSELECT` statement can be used to select a variable and evaluate it on a case-by-case basis and execute code blocks accordingly. Here is the code block that will look at each letter value held in the `instruction` variable and then call the appropriate subroutine for each instance, or case, of a given letter.

4

```
SELECT instruction
CASE "F": GOSUB Forward
CASE "B": GOSUB Backward
CASE "R": GOSUB Right_Turn
CASE "L": GOSUB Left_Turn
ENDSELECT
```

Here are these concepts, all together in a single program.

Example Program: EepromNavigation.bs2

- ✓ Carefully read the code instructions and comments in `EepromNavigation.bs2` to understand what each part of the program does.
- ✓ Enter, save, and run `EepromNavigation.bs2`.

```
' Robotics with the Boe-Bot - EepromNavigation.bs2
' Navigate using characters stored in EEPROM.

' {$STAMP BS2}                                ' Stamp directive.
' {$PBASIC 2.5}                                ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Variables ]-----
pulseCount    VAR    Word                    ' Stores number of pulses.
address       VAR    Byte                    ' Stores EEPROM address.
instruction    VAR    Byte                    ' Stores EEPROM instruction.

' -----[ EEPROM Data ]-----

'      Address: 0123456789                    ' These two commented lines show
'      |||||            ' EEPROM address of each datum.
DATA      "FLFFRBLBBQ"                    ' Navigation instructions.

' -----[ Initialization ]-----
```

[illegible]

```

' -----[ Subroutine - Right_Turn ]-----
Right_Turn:
  FOR pulseCount = 1 TO 24
    PULSOUT 13, 850
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN
' right turn subroutine.
' Send 24 right rotate pulses.
' 1.7 ms pulse to left servo.
' 1.7 ms pulse to right servo.
' Pause for 20 ms.
' Return to Main Routine section.

```

4

Did your Boe-Bot drive in a rectangle, going forward on the first two sides and backwards on the second two? If it looked more like a trapezoid, you may want to adjust the **FOR...NEXT** loop's *EndValue* arguments in the turning subroutines to make precise 90-degree turns.

Your Turn

- ✓ With EepromNavigation.bs2 active in the BASIC Stamp Editor, click Run and select Memory Map.

Your stored instructions will appear highlighted in blue at the beginning of the Detailed EEPROM Map as shown in Figure 4-7. The numbers shown are the hexadecimal ASCII (American Standard Code for Information Interchange) codes that correspond to the characters you entered in your **DATA** statement.

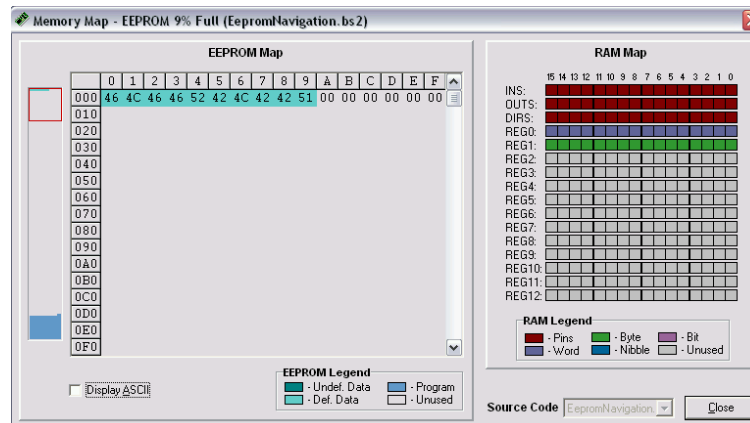


Figure 4-7
Memory Map
with Stored
Instructions
Visible in
EEPROM Map

- ✓ Click the Display ASCII checkbox near the lower left corner of the Memory Map window.

Now the direction instructions will appear in a more familiar format shown in Figure 4-8. Instead of ASCII codes, they appear as the actual characters you recorded using the **DATA** directive.



Figure 4-8
Close-up of the
Detailed
EEPROM Map
after Display
ASCII Box is
Checked

This program stored a total of 10 characters in EEPROM. These ten characters were accessed by the **READ** command's **address** variable. The **address** variable was declared as a byte, so it can access up to 256 locations, well over the 10 we needed. If the **address** variable is re-declared to be a word variable, you could theoretically access up to 65535, far more locations than are available. Keep in mind that if your program gets larger, the number of available EEPROM addresses for holding data gets smaller.

You can modify the existing data string to a new set of directions. You can also add additional **DATA** statements. The data is stored sequentially, so the first character in the second data string will get stored immediately after the last character in the first data string.

- ✓ Try changing, adding, and deleting characters in the **DATA** directive, and re-running the program. Remember that the last character in the **DATA** directive should always be a "Q."
- ✓ Modify the **DATA** directive to make your Boe-Bot perform the familiar forward-left-right-backward sequence of movements.
- ✓ Try adding a second **DATA** directive. Remember to remove the "Q" from the end of the first **DATA** directive and add it to the end of the second. Otherwise, the program will execute only the commands in the first **DATA** directive.

Example Program – EepromNavigationWithWordValues.bs2

This next example program looks complicated at first, but it is a very efficient way to design programs for custom Boe-Bot choreography. This example program uses EEPROM data storage, but does not use subroutines. Instead, a single code block is used, with variables in place of the **FOR...NEXT** loop's *EndValue* and **PULSOUT** *Duration* arguments.

By default, the **DATA** directive stores bytes of information in EEPROM. To store word-sized data items, you can add the **Word** modifier to the **DATA** directive, before each data item in your string. Each word-sized data item will use two bytes of EEPROM storage, so the data will be accessed via every other address location. When using more than one **DATA** directive, it is most convenient to assign a label to each one. This way, your **READ** commands can refer to the label to retrieve data items without you having to figure out at which EEPROM address each string of data items begins. Take a look at this code snippet:

```
' addressOffset  0          2          4          6          8
Pulses_Count DATA Word 64, Word 24, Word 24, Word 64, Word 0
Pulses_Left  DATA Word 850, Word 650, Word 850, Word 650
Pulses_Right DATA Word 650, Word 650, Word 850, Word 850
```

Each of the three **DATA** statements begins with its own label. The **Word** modifier goes before each data item, and the items are separated by commas. These three strings of data will be stored in EEPROM one after another. We don't have to do the math to figure out the address number of a given data item, because the labels and the **addressOffset** variable will do that automatically. The **READ** command uses each label to determine the EEPROM address where that string begins, and then adds the value of the **addressOffset** variable to know how many address numbers to shift over to find the correct *DataItem*. The *DataItem* found at the resulting **Address** will be stored in the **READ** command's *Variable* argument. Notice that the **Word** modifier also comes before the variable that stores the value fetched from EEPROM.

```
DO
  READ Pulses_Count + addressOffset, Word pulseCount
  READ Pulses_Left + addressOffset, Word pulseLeft
  READ Pulses_Right + addressOffset, Word pulseRight

  addressOffset = addressOffset + 2

' PBASIC code block omitted here.
```

```
LOOP UNTIL (pulseCount = 0)
```

The first time through the loop, `addressOffset = 0`. The first **READ** command will retrieve a value of 64 from the first address at the `Pulses_Count` label, and place it in the `pulseCount` variable. The second **READ** command retrieves a value of 850 from the first address specified by the `Pulses_Left` label, and places it in the `pulseLeft` variable. The third **READ** command retrieves a value of 650 from the first address specified by the `Pulses_Right` label and places it in the `pulseRight` variable. Notice that these are the three values in the “0” column of the code snippet on page 153. When the value of those variables are placed in the code block that follows, this:

FOR counter = 1 TO pulseCount		FOR counter = 1 TO 64
PULSOUT 13, pulseLeft		PULSOUT 13, 850
PULSOUT 12, pulseRight	becomes	PULSOUT 12, 650
PAUSE 20		PAUSE 20
NEXT		NEXT

Do you recognize the basic maneuver generated by this code block?

- ✓ Look at the other columns of the code snippet on page 153 and anticipate what the **FOR...NEXT** code block will look like on the second, third, and fourth times through the loop.
- ✓ Look at the **LOOP UNTIL (pulseCount = 0)** statement in the program below. The `<>` operator stands for “not equal to”. What will happen on the fifth time through the loop?
- ✓ Enter, save, and run `EepromNavigationWithWordValues.bs2`.

```
' Robotics with the Boe-Bot - EepromNavigationWithWordValues.bs2
' Store lists of word values that dictate.

' {$STAMP BS2}
' {$PBASIC 2.5}
' Stamp directive.
' PBASIC directive.

DEBUG "Program Running!"

' -----[ Variables ]-----

counter      VAR    Word
pulseCount   VAR    Word
addressOffset VAR    Byte
instruction   VAR    Byte
pulseRight   VAR    Word

' Stores number of pulses.
' Stores offset from label.
' Stores EEPROM instruction.
' Stores servo pulse widths.
```

```

pulseLeft      VAR      Word

' -----[ EEPROM Data ]-----

' addressOffset      0          2          4          6          8
Pulses_Count DATA   Word 64, Word 24, Word 24, Word 64, Word 0
Pulses_Left  DATA   Word 850, Word 650, Word 850, Word 650
Pulses_Right DATA   Word 650, Word 650, Word 850, Word 850

' -----[ Initialization ]-----

FREQOUT 4, 2000, 3000          ' Signal program start/reset.

' -----[ Main Routine ]-----

DO

  READ Pulses_Count + addressOffset, Word pulseCount
  READ Pulses_Left + addressOffset, Word pulseLeft
  READ Pulses_Right + addressOffset, Word pulseRight

  addressOffset = addressOffset + 2

  FOR counter = 1 TO pulseCount
    PULSOUT 13, pulseLeft
    PULSOUT 12, pulseRight
    PAUSE 20
  NEXT

LOOP UNTIL (pulseCount = 0)

END          ' Stop executing until reset.

```

Did your Boe-Bot perform the familiar forward-left-right-backwards movements? Are you thoroughly bored with it by now? Do you want to see your Boe-Bot do something else, or to choreograph your own routine?

Your Turn – Making Your Own Custom Navigation Routines

- ✓ Save EepromNavigationWithWordValues.bs2. under a new name.
- ✓ Replace the **DATA** directives with the ones below.
- ✓ Run the modified program and see what your Boe-Bot does.

```

Pulses_Count DATA Word 60, Word 80, Word 100, Word 110,
                  Word 110, Word 100, Word 80, Word 60, Word 0
Pulses_Left  DATA Word 850, Word 800, Word 785, Word 760, Word 750,

```

```
Word 740, Word 715, Word 700, Word 650, Word 750
Pulses_Right DATA Word 650, Word 700, Word 715, Word 740, Word 750,
Word 760, Word 785, Word 800, Word 850, Word 750
```

- ✓ Make a table with three rows, one for each **DATA** directive, and a column for each Boe-Bot maneuver you want to make, plus one for the **Word 0** item in the **Pulses_Count** row.
- ✓ Use the table to plan out your Boe-Bot choreography, filling in the **FOR...NEXT** loop's **EndValue** and **PULSOUT Duration** arguments you will need for each maneuver's code block.
- ✓ Modify your program with your newly charted **DATA** directives.
- ✓ Enter, save, and run your custom program. Did your Boe-Bot do what you wanted it to do? Keep working on it until it does.

SUMMARY

This chapter introduced the basic Boe-Bot maneuvers: forward, backward, rotating in place to turn to the right or left, and pivoting. The type of maneuver is determined by the **PULSOUT** commands' *Duration* arguments. How far the maneuver goes is determined by the **FOR...NEXT** loop's *StartValue* and *EndValue* arguments.

Chapter 2 included a hardware adjustment, physically centering the Boe-Bot's servos with a screwdriver. This chapter focused on fine tuning adjustments made by manipulating the software. Specifically, a difference in rotation speed between the two servos was compensated for by changing the **PULSOUT** command's *Duration* argument for the faster of the two servos. This changes the Boe-Bot's path from a curve to a straight line if the servos are not perfectly matched. To refine turning so that the Boe-Bot turns to the desired angle, the *StartValue* and *EndValue* arguments of a **FOR...NEXT** loop can be adjusted.

Programming the Boe-Bot to travel a pre-defined distance can be accomplished by measuring the distance it travels in one second, with the help of a ruler. Using this distance, and the number of pulses in one second of run time, you can calculate the number of pulses required to cover a desired distance.

Ramping was introduced as a way to gradually accelerate and decelerate. It's kinder to the servos, and we recommended that you use your own ramping routines in place of the abrupt start and stop routines shown in the example programs. Ramping is accomplished by taking the same variable that's used as the *Counter* argument in a **FOR...NEXT** loop and adding it to or subtracting it from 750 in the **PULSOUT** command's *Duration* argument.

Subroutines were introduced as a way to make pre-programmed maneuvers reusable by a PBASIC program. Instead of writing an entire **FOR...NEXT** loop for each new maneuver, a single subroutine that contains a **FOR...NEXT** loop can be executed as needed with the **GOSUB** command. A subroutine begins with a label, and ends with the **RETURN** command. A subroutine is called from the main program with a **GOSUB** command. When the subroutine is finished and it encounters the **RETURN** command, the next command to be executed is the one immediately following the **GOSUB** command.

The BASIC Stamp's EEPROM stores the program it runs, but you can take advantage of any unused portion of the program to store values. This is a great way to store custom navigation routines. The **DATA** directive can store values in EEPROM. Bytes are stored by default, but adding the **word** modifier to each data item allows you to store values up to 65535 in two bytes' worth of EEPROM memory space. You can read values back out of EEPROM using the **READ** command. If you are retrieving a word-sized variable, make sure to place a **word** modifier before the variable that will receive the value that **READ** fetches. **SELECT...CASE** was introduced as a way of evaluating a variable on a case by case basis, and executing a different code block depending on the case. Optional **DO...LOOP** conditions are helpful in certain circumstances; **DO UNTIL (Condition) ...LOOP** and **DO...LOOP UNTIL (Condition)** were demonstrated as ways to keep executing a **DO...LOOP** until a particular condition is detected.

Questions

1. What direction does the left wheel have to turn to make the Boe-Bot go forward? What direction does the right wheel have to turn?
2. When the Boe-Bot pivots to the left, what are the right and left wheels doing? What PBASIC commands do you need to make the Boe-Bot pivot left?
3. If your Boe-Bot veers slightly to the left when you are running a program to make it go straight ahead, how do you correct this? What command needs to be adjusted and what kind of adjustment should you make?
4. If your Boe-Bot travels 11 in/s, how many pulses will it take to make it travel 36 inches?
5. What's the relationship between a **FOR...NEXT** loop's *Counter* argument and the **PULSOUT** command's *Duration* argument that makes ramping possible?
6. What directive can you use to pre-store values in the BASIC Stamp's EEPROM before running a program?
7. What command can you use to retrieve a value stored in EEPROM and copy it to a variable?
8. What code block can you use to select a particular variable and evaluate it on a case by case basis and execute a different code block for each case?
9. What are the different conditions that can be used with **DO...LOOP**?

Exercises

1. Write a routine that makes the Boe-Bot back up for 350 pulses.

2. Let's say that you tested your servos and discovered that it takes 48 pulses to make a 180° turn with right-rotate. With this information, write routines to make the Boe-Bot perform 30, 45, and 60 degree turns.
3. Write a routine that makes the Boe-Bot go straight forward, then ramp in and out of a pivoting turn, and then continue straight forward.

Projects

4

1. It is time to fill in column 3 of Table 2-1: **PULSOUT Duration** Combinations on page 81. To do this, modify the **PULSOUT Duration** arguments in the program BoeBotForwardThreeSeconds.bs2 using each pair of values from column 1. Record your Boe-Bot's resultant behavior for each pair in column 3. Once completed, this table will serve as a reference guide when you design your own custom Boe-Bot maneuvers.
2. Figure 4-9 shows two simple obstacle courses. Write a program that will make your Boe-Bot navigate along each figure. Assume straight line distances (including the diameter of the circle) to be either 1 yd or 1 m.

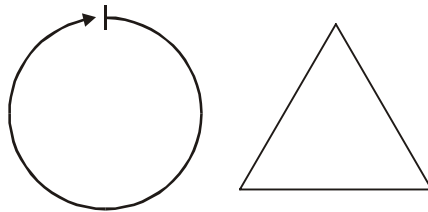


Figure 4-9
Simple Obstacle
Courses

Solutions

- Q1. Left wheel counterclockwise, right wheel clockwise.
 Q2. The right wheel is turning clockwise (forward), and the left wheel is not moving.

```
PULSOUT 13, 750
PULSOUT 12, 650
```

- Q3. You can slow down the right wheel to correct a veer to the left. The **PULSOUT** command for the right wheel needs to be adjusted.

```
PULSOUT 12, 650
```

- Adjust the 650 to something closer to 750 to slow the wheel down.

```
PULSOUT 12, 663
```

- Q4. Given:

Boe-Bot speed = 11 in/s

Boe-Bot distance = 36 in/s

pulses = (Boe-Bot distance / Boe-Bot speed) * (40.65 pulses / s)

= (36 / 11) * (40.65)

= 133.04

= 133

It should take 133 pulses to travel 36 inches.

- Q5. The **FOR...NEXT** loop's **pulseCount** variable can be used as an offset (plus or minus) to 750 (the center position) in the **Duration** argument.

```
FOR pulseCount = 1 to 100
  PULSOUT 13, 750 + pulseCount
  PULSOUT 12, 750 - pulseCount
  PAUSE 20
NEXT
```

- Q6. The **DATA** directive.

- Q7. The **READ** command.

- Q8. **SELECT...CASE...ENDSELECT**.

- Q9. **UNTIL** and **WHILE**.

```
E1.   FOR counter = 1 to 350 ' Backward
      PULSOUT 13, 650
      PULSOUT 12, 850
      PAUSE 20
      NEXT
```

```
E2.   FOR counter = 1 to 8 ' Rotate right 30 degrees
      PULSOUT 13, 850
```



```

        PULSOUT 12, 850
        PAUSE 20
    NEXT

    FOR counter = 1 to 12      ' Rotate right 45 degrees
        PULSOUT 13, 850
        PULSOUT 12, 850
        PAUSE 20
    NEXT

    FOR counter = 1 to 16      ' Rotate right 60 degrees
        PULSOUT 13, 850
        PULSOUT 12, 850
        PAUSE 20
    NEXT

E3.  FOR counter = 1 to 100      ' Forward
        PULSOUT 13, 850
        PULSOUT 12, 650
        PAUSE 20
    NEXT

    FOR counter = 0 TO 30      ' Ramping pivot turn
        PULSOUT 13, 750 + counter
        PULSOUT 12, 750
        PAUSE 20
    NEXT

    FOR counter = 30 TO 0
        PULSOUT 13, 750 + counter
        PULSOUT 12, 750
        PAUSE 20
    NEXT

    FOR counter = 1 to 100      ' Forward
        PULSOUT 13, 850
        PULSOUT 12, 650
        PAUSE 20
    NEXT

```

P1.

P13	P12	Description	Behavior
850	650	Full Speed P13 CCW, P12 CW	Forward
650	850	Full Speed P13 CW, P12 CCW	Backward
850	850	Full Speed P13 CCW, P12 CCW	Right rotate
650	650	Full Speed P13 CW, P12 CW	Left rotate
750	850	P13 Stopped P12 CCW Full speed	Pivot back left
650	750	P13 CW Full Speed P12 Stopped	Pivot back right
750	750	P13 Stopped P12 Stopped	Stopped
760	740	P13 CCW Slow P12 CW Slow	Forward slow
770	730	P13 CCW Med P12 CW Med	Forward medium
850	700	P13 CCW Full Speed P12 CW Medium	Veer right
800	650	P13 CCW Medium P12 CW Full Speed	Veer left

P2. The circle can be implemented by veering right continuously. Trial and error, a yard or meter stick, will help you arrive at the right **PULSOUT** value. Circle with a one-yard diameter:

```
' Robotics with the Boe-Bot - Chapter 4 - Circle.bs2
' Boe-Bot navigates a circle of 1 yard diameter.

'{$STAMP BS2}
'{$PBASIC 2.5}
DEBUG "Program running!"

pulseCount      VAR      Word          ' Pulse count to servos

FREQOUT 4, 2000, 3000                  ' Signal program start/reset.

' -----[ Main Routine ]-----
Main:
DO
    PULSOUT 13, 850                      ' Veer right
    PULSOUT 12, 716
    PAUSE 20
LOOP
```

To make the triangle, first calculate the number of pulses required for a one meter or yard straight line, as in Question 4. Then fine-tune your distances to

match your Boe-Bot and particular surface. For a triangle pattern, the Boe-Bot must travel 1 meter/yard forward, then make a 120 degree turn. This should be repeated three times for the three sides of the triangle. You may have to adjust the `pulseCount EndValue` in the `Right_Rotate120` subroutine to get a precise 120 degree turn.

```
' Robotics with the Boe-Bot - Chapter 4 - Triangle.bs2
' Boe-Bot navigates triangle shape with 1 yard sides.
' Go forward, then turn 120 degrees. Repeat three times.

'{$STAMP BS2}
'{$PBASIC 2.5}
DEBUG "Program running!"

counter      VAR    Nib      ' Triangle has 3 sides
pulseCount   VAR    Word     ' Pulse count to servos

FREQOUT 4, 2000, 3000        ' Signal program start/reset.

Main:
  FOR counter = 1 TO 3      ' Repeat 3 times for triangle
    GOSUB Forward
    GOSUB Right_Rotate120
  NEXT
END

Forward:
  FOR pulseCount = 1 TO 163 ' Forward 1 yard
    PULSOUT 13, 850
    PULSOUT 12, 650
    PAUSE 20
  NEXT
  RETURN

Right_Rotate120:
  FOR pulseCount = 1 TO 21  ' Rotate right 120 degrees
    PULSOUT 13, 850
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN
```


Chapter 5: Tactile Navigation with Whiskers

5

Many types of robotic machinery rely on a variety of tactile switches. For example, a tactile switch may detect when a robotic arm has encountered an object. The robot can be programmed to pick up the object and place it elsewhere. Factories use tactile switches to count objects on a production line, and also for aligning objects during industrial processes. In all these instances, the switches provide inputs that dictate some other form of programmed output. The inputs are electronically monitored by the product, be it a robot, or a calculator, or a production line. Based on the state of the switches, the robot arm grabs an object, or the calculator display updates, or the factory production line reacts with motors or servos to guide products.

In this chapter, you will build tactile switches, called whiskers, onto your Boe-Bot and test them. You will then program the Boe-Bot to monitor the state of these switches, and to decide what to do when it encounters an obstacle. The end result will be autonomous navigation by touch.

TACTILE NAVIGATION

The whiskers are so named because that is what these bumper switches look like, though some argue they look more like antennae. At any rate, these whiskers are shown mounted on a Boe-Bot in Figure 5-1. Whiskers give the Boe-Bot the ability to sense the world around it through touch, much like the antennae on an ant or the whiskers on a cat. The activities in this chapter use the whiskers by themselves, but they can also be combined with other sensors you will learn about in later chapters to increase your Boe-Bot's functionality.

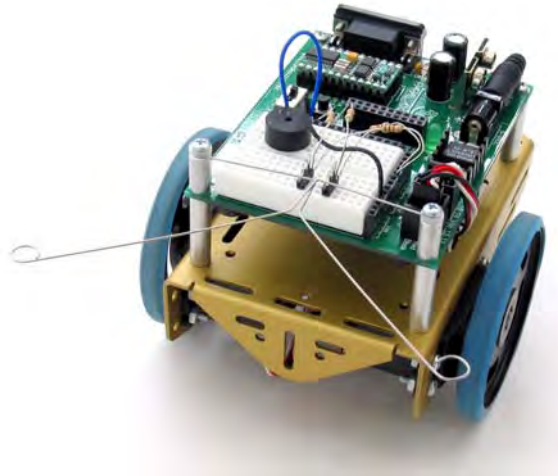


Figure 5-1
Boe-Bot with
Whiskers

ACTIVITY #1: BUILDING AND TESTING THE WHISKERS

Before moving on to programs that make the Boe-Bot navigate based on what it can touch, it's essential to build and test the whiskers first. This activity will guide you through building and testing the whiskers.

Whisker Circuit and Assembly

- ✓ Gather the whiskers hardware shown in Figure 5-2.
- ✓ Disconnect power from your board and servos.

Parts List:

- (2) Whisker wires
- (2) $\frac{7}{8}$ " pan head 4-40 Phillips screws
- (2) $\frac{1}{2}$ " round spacer
- (2) Nylon washers – size #4
- (2) 3-pin m/m headers
- (2) Resistors, 220 Ω (red-red-brown)
- (2) Resistors, 10 k Ω (brown-black-orange)



Figure 5-2
Whiskers
Hardware

5

Building the Whiskers

- ✓ Remove the two front screws that hold your board to the front standoffs.
- ✓ Refer to Figure 5-3 while following the remaining instructions.
- ✓ Thread a nylon washer and then a $\frac{1}{2}$ " round spacer on each of the $\frac{7}{8}$ " screws.
- ✓ Attach the screws through the holes in your board and into the standoffs below, but do not tighten them all the way yet.
- ✓ Slip the hooked ends of the whisker wires around the screws, one above the washer and the other below the washer, positioning them so they cross over each other without touching.
- ✓ Tighten the screws into the standoffs.

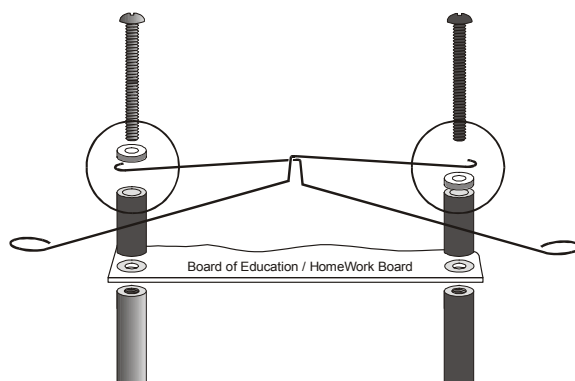


Figure 5-3
Mounting the
Whiskers

The next step is add the whiskers circuit shown in Figure 5-4 to the piezospeaker and servo circuits you built and tested in Chapter 2 and Chapter 3.

- ✓ If you have a Board of Education, build the whiskers circuit shown in Figure 5-4 using the wiring diagram in Figure 5-5 on page 169 as a reference.
- ✓ If you have a HomeWork Board, build the whiskers circuit shown in Figure 5-4 using the wiring diagram in Figure 5-6 on page 170 as a reference.
- ✓ Make sure to adjust each whisker so that it is close to, but not touching, the 3-pin header on the breadboard. A distance of about $\frac{1}{8}$ " (3 mm) is a recommended starting point.

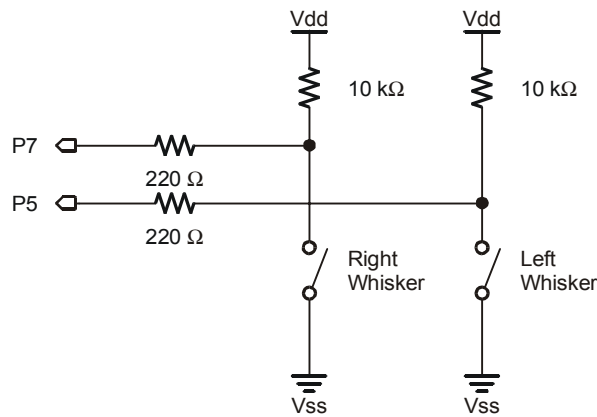
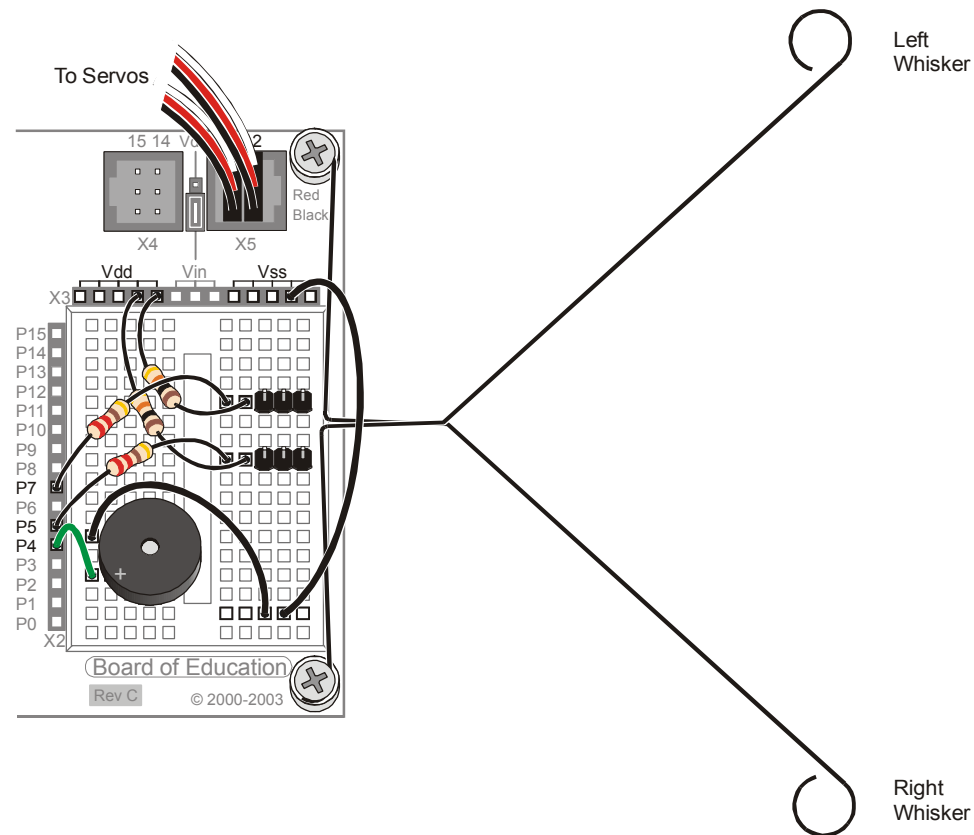


Figure 5-4
Whiskers
Schematic

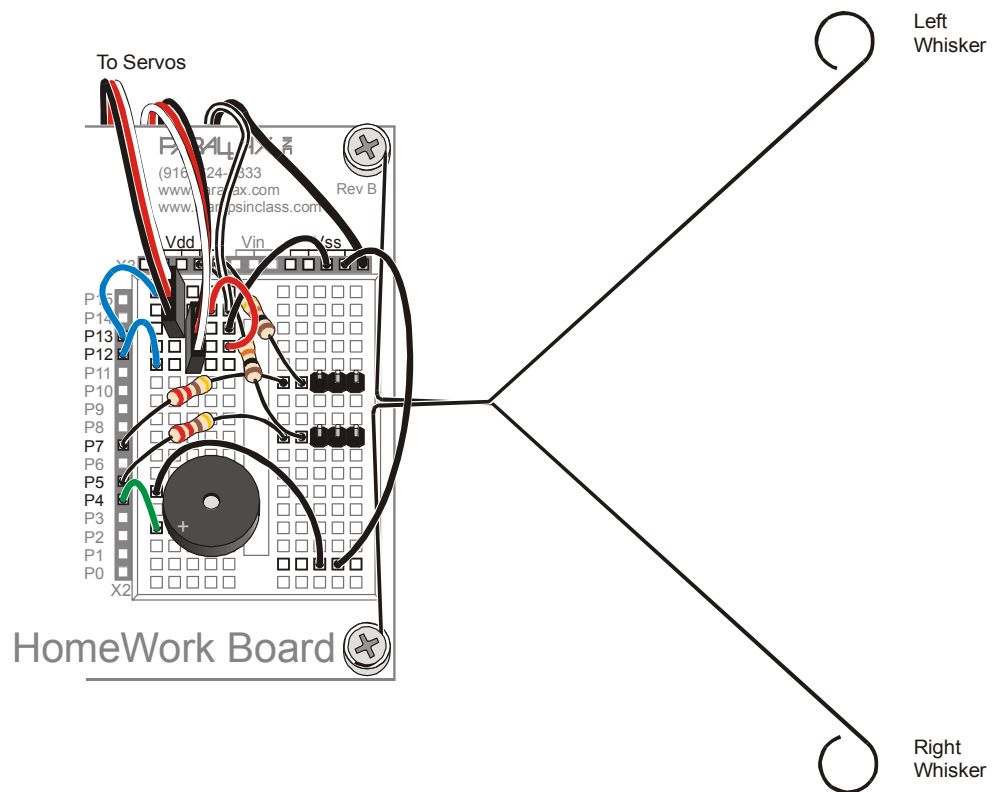
Figure 5-5: Whisker Wiring Diagram for the Board of Education

5



Use the 220 Ω resistors (red-red-brown color codes) to connect P5 and P7 to their corresponding 3-pin headers. Use the 10 k Ω resistors (brown-black-orange color codes) to connect Vdd to each 3-pin header.

Figure 5-6: Whisker Wiring Diagram for the HomeWork Board



Use the 220 Ω resistors (red-red-brown-color codes) to connect P5 and P7 to their corresponding 3-pin headers. Use the 10 k Ω resistors (brown-black-orange color codes) to connect Vdd to each 3-pin header.

Testing the Whiskers

Take a second look at the whiskers schematic (Figure 5-7). Each whisker is both the mechanical extension and the ground electrical connection of a normally open, single-pole, single-throw switch. The reason the whiskers are connected to ground (Vss) is because the plated holes at the outer edge of the board are all connected to Vss. This is true for both the Board of Education and the BASIC Stamp HomeWork Board. The metal standoffs and screw provide the electrical connection to each whisker.

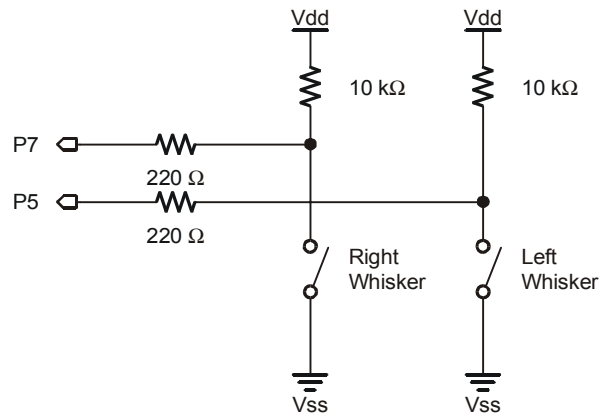


Figure 5-7
Whiskers
Schematic –
A Second
Look

The BASIC Stamp can be programmed to detect when a whisker is pressed. I/O pins connected to each switch circuit monitor the voltage at the 10 kΩ pull-up resistor. Figure 5-8 illustrates how this works. When a given whisker is not pressed, the voltage at the I/O pin connected to that whisker is 5 V. When a whisker is pressed, the I/O line is shorted to ground (Vss), so the I/O line sees 0 V.

All I/O pins default to input every time a PBASIC program starts. This means that the I/O pins connected to the whiskers will function as inputs automatically. As an input, an I/O pin connected to a whisker circuit will cause its input register to store a 1 if the voltage is 5 V (whisker not pressed) or a 0 if the voltage is 0 V (whisker pressed). The Debug Terminal can be used to display these values.



How do you get the BASIC Stamp to tell you whether it's reading a 1 or 0? Because the circuit is connected to P7, this 1 or 0 value will appear in a variable named **IN7**. **IN7** is called an input register. Input register variables are built-in and do not have to be declared in the beginning of your program. You can see the value this variable is storing by using the command **DEBUG BIN1 IN7**. The **BIN1** is a formatter that tells the Debug Terminal to display one binary digit (either 1 or 0).

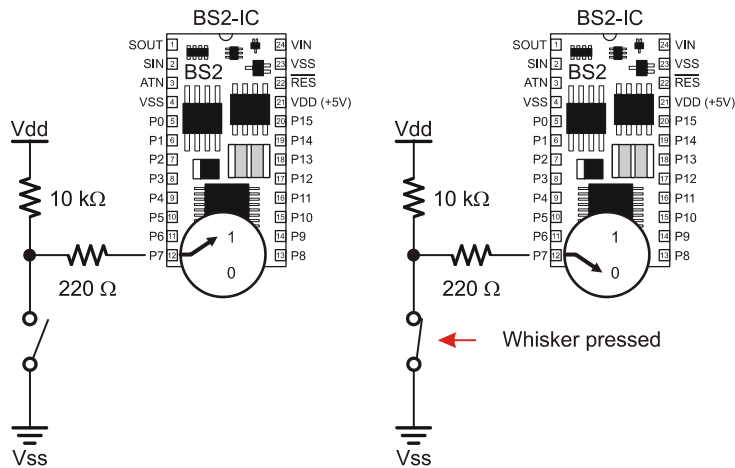


Figure 5-8
Detecting Electrical
Contacts

Example Program: TestWhiskers.bs2

This next example program is designed to test the whiskers to make sure they are functioning properly. By displaying the binary digits stored in the P7 and P5 input registers (**IN7** and **IN5**), the program will show you whether the BASIC Stamp detects contact with a whisker. When the value stored in a given input register is 1, the whisker is not pressed. When it is 0, the whisker is pressed.

- ✓ Reconnect power to your board and servos.
- ✓ Enter, save, and run TestWhiskers.bs2.
- ✓ This program makes use of the Debug Terminal, so leave the serial cable connected to the BASIC Stamp while the program is running.

```
' Robotics with the Boe-Bot - TestWhiskers.bs2
' Display what the I/O pins connected to the whiskers sense.

' {$STAMP BS2}                                ' Stamp directive.
```

```
' {$PBASIC 2.5}                                ' PBASIC directive.

DEBUG "WHISKER STATES", CR,
      "Left      Right", CR,
      "-----  -----"

DO
  DEBUG CRSRXY, 0, 3,
        "P5 = ", BIN1 IN5,
        "    P7 = ", BIN1 IN7
  PAUSE 50
LOOP
```

5

- ✓ Note the values displayed in the Debug Terminal; it should display that both P7 and P5 are equal to 1.
- ✓ Check Figure 5-5 on page 169 (or Figure 5-6 on page 170) so you know which whisker is the “left whisker” and which whisker is the “right whisker”.
- ✓ Press the right whisker into its three-pin header, and note the values displayed in the Debug Terminal. It should now read:
P5 = 1 P7 = 0
- ✓ Press the left whisker into its three-pin header, and note the value displayed in the Debug Terminal again. This time it should read:
P5 = 0 P7 = 1
- ✓ Press both whiskers against both three-pin headers. Now it should read
P5 = 0 P7 = 0
- ✓ If the whiskers passed all these tests, you’re ready to move on; otherwise, check your program and circuits for errors.

What is CRSRXY?

It is a formatter that allows you to conveniently arrange information your program sends to the Debug Terminal. The formatter **CRSRXY 0, 3**, in the command

```
DEBUG CRSRXY, 0, 3,
      "P5 = ", BIN1 IN5,
      "    P7 = ", BIN1 IN7
```

places the cursor at column 0, row 3 in the Debug Terminal. This makes it display nicely below the “Whisker States” table heading. Each time through the loop, the new values overwrite the old values because the cursor keeps going back to the same place.

ACTIVITY #2: FIELD TESTING THE WHISKERS

Assume that you may have to test the whiskers at some later time away from a computer. Since the Debug Terminal won't be available, what can you do? One solution would be to program the BASIC Stamp so that it sends an output signal that corresponds to the input signal it's receiving. This can be done with a pair of LED circuits and a program that turns the LEDs on and off based on the whisker inputs.

Parts List:

- (2) Resistors - 220 Ω (red-red-brown)
- (2) LEDs – Red

Building the LED Whisker Testing Circuits

- ✓ Disconnect power from your board and servos.
- ✓ If you have a Board of Education, add the circuit shown in Figure 5-9 with the help of the wiring diagram in Figure 5-10 (page 175).
- ✓ If you have a HomeWork Board, add the circuit shown in Figure 5-9 with the help of the wiring diagram in Figure 5-11 (page 176).

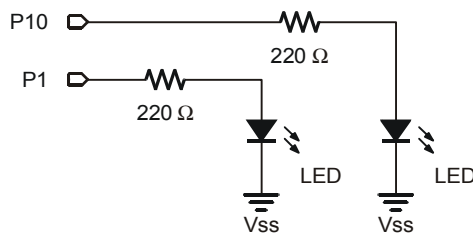


Figure 5-9
LED Whisker
Testing
Schematic

*Add this LED
circuit.*



Remember that an LED is a one way current valve. If it is plugged in backwards, it will not let current pass through, and so will not emit light. For the LED to emit light when the BASIC Stamp sends a high signal, the LED's anode must be connected to the 220 Ω resistor, and its cathode must be connected to Vss. See Figure 5-10 or Figure 5-11.

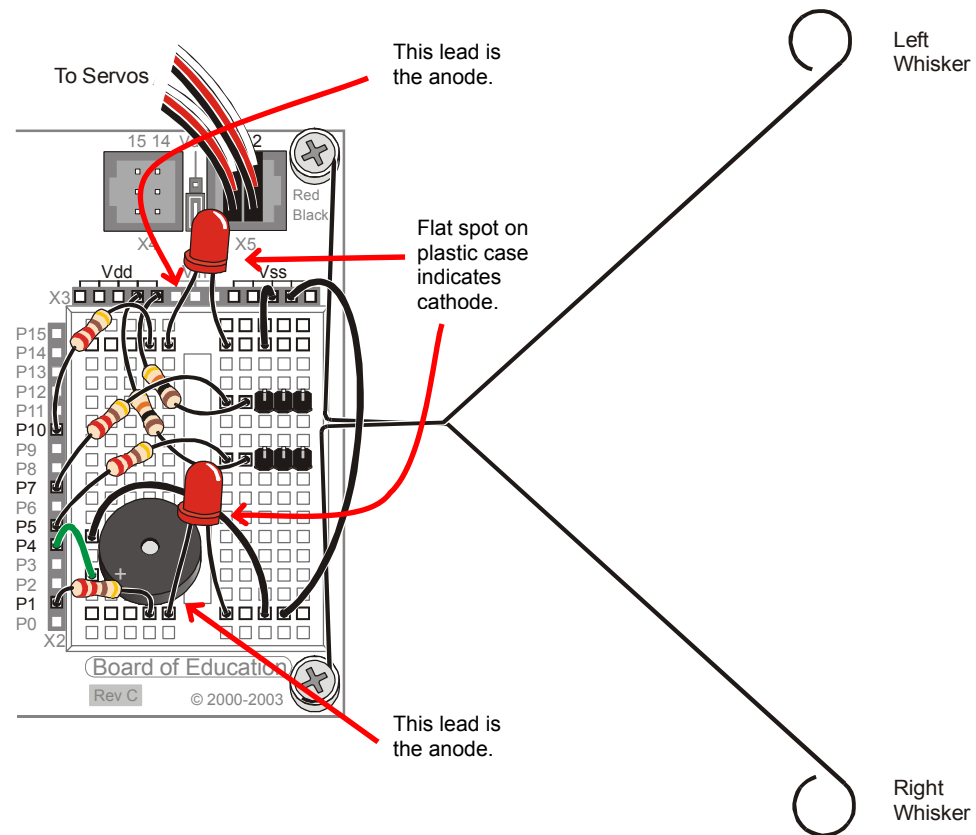
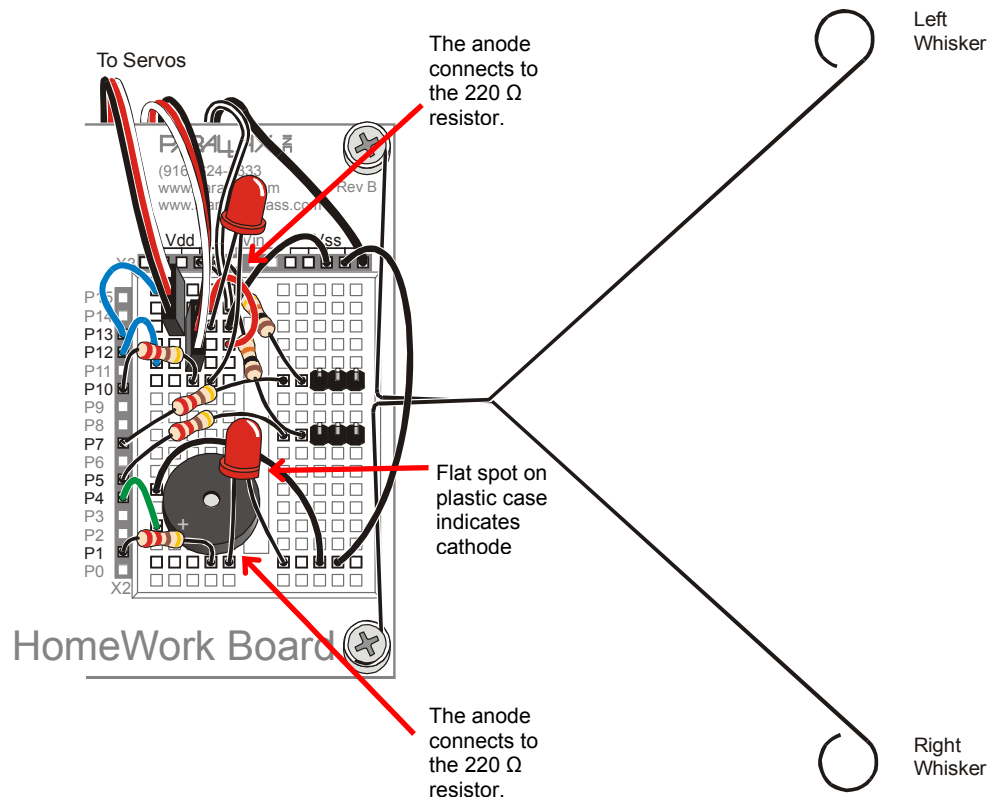
Figure 5-10: Whisker Plus LED Wiring Diagram for the Board of Education

Figure 5-11: Whisker Plus LED Wiring Diagram for the HomeWork Board



Programming the LED Whisker Testing Circuits

- ✓ Reconnect power to your board.
- ✓ Save TestWhiskers.bs2 as TestWhiskersWithLeds.bs2.
- ✓ Insert these two **IF...THEN** statements between the **PAUSE 50** and **LOOP** commands.

```

IF (IN7 = 0) THEN
  HIGH 1
ELSE
  LOW 1
ENDIF

```

```

IF (IN5 = 0) THEN
  HIGH 10
ELSE
  LOW 10
ENDIF

```

These are called **IF...THEN** statements, and they will be more fully introduced in the next activity. These statements are used to make decisions in PBASIC. The first of the two **IF...THEN** statements sets P1 high, which turns the LED on when the whisker connected to P7 is pressed (**IN7 = 0**). The **ELSE** portion of the statement makes P1 go low, which turns the LED off when the whisker is not pressed. The second **IF...THEN** statement does the same thing for the whisker connected to P5 and the LED connected to P10.

- ✓ RunTestWhiskersWithLeds.bs2.
- ✓ Test the program by gently pressing the whiskers. The red LEDs should light up when each whisker has made contact with its 3-pin header.

ACTIVITY #3: NAVIGATION WITH WHISKERS

In Activity #1, the BASIC Stamp was programmed to detect whether a given whisker was pressed. In this activity, the BASIC Stamp will be programmed to take advantage of this information to guide the Boe-Bot. When the Boe-Bot is rolling along and a whisker is pressed, it means the Boe-Bot bumped into something. A navigation program needs to take this input, decide what it means, and call a set of maneuvers that will make the Boe-Bot back up from the obstacle, turn, and go in a different direction.

Programming the Boe-Bot to Navigate Based on Whisker Inputs

This next program makes the Boe-Bot go forward until it encounters an obstacle. In this case, the Boe-Bot knows when it encounters an obstacle by bumping into it with one or both of its whiskers. As soon as the obstacle is detected by the whiskers, the navigation routines and subroutines developed in Chapter 4 will make the Boe-Bot back up and turn. Then, the Boe-Bot resumes forward motion until it bumps into another obstacle.

In order to do that, the Boe-Bot needs to be programmed to make decisions. PBASIC has a command called an **IF...THEN** statement that makes decisions. The syntax for **IF...THEN** statements is:

IF (condition) THEN...{ELSEIF (condition)}...{ELSE}...ENDIF

The “...” means you can place a code block (one or more commands) between the keywords. The next example program makes decisions based on the whisker inputs, and then calls subroutines to make the Boe-Bot take action. The subroutines are similar to the ones you developed in Chapter 4. Here is how **IF...THEN** is used.

```

IF (IN5 = 0) AND (IN7 = 0) THEN
    GOSUB Back_Up           ' Both whiskers detect obstacle,
    GOSUB Turn_Left         ' back up & U-turn (left twice)
    GOSUB Turn_Left
ELSEIF (IN5 = 0) THEN      ' Left whisker contacts
    GOSUB Back_Up           ' Back up & turn right
    GOSUB Turn_Right
ELSEIF (IN7 = 0) THEN      ' Right whisker contacts
    GOSUB Back_Up           ' Back up & turn left
    GOSUB Turn_Left
ELSE                        ' Both whiskers 1, no contacts
    GOSUB Forward_Pulse     ' Apply a forward pulse &
ENDIF                      ' check again

```

Example Program: RoamingWithWhiskers.bs2

This program demonstrates one way of evaluating the whisker inputs and deciding which navigation subroutine to call using **IF...THEN**.

- ✓ Reconnect power to your board and servos.
- ✓ Enter, save, and run RoamingWithWhiskers.bs2.

- ✓ Try letting the Boe-Bot roam. When it contacts obstacles in its path, it should back up, turn, and then roam in a new direction.

```
' -----[ Title ]-----
' Robotics with the Boe-Bot - RoamingWithWhiskers.bs2
' Boe-Bot uses whiskers to detect objects, and navigates around them.

' {$STAMP BS2}                                ' Stamp directive.
' {$PBASIC 2.5}                                ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Variables ]-----

pulseCount      VAR      Byte                  ' FOR...NEXT loop counter.

' -----[ Initialization ]-----

FREQOUT 4, 2000, 3000                          ' Signal program start/reset.

' -----[ Main Routine ]-----

DO
  IF (IN5 = 0) AND (IN7 = 0) THEN                ' Both whiskers detect obstacle
    GOSUB Back_Up                               ' Back up & U-turn (left twice)
    GOSUB Turn_Left
    GOSUB Turn_Left
  ELSEIF (IN5 = 0) THEN                          ' Left whisker contacts
    GOSUB Back_Up                               ' Back up & turn right
    GOSUB Turn_Right
  ELSEIF (IN7 = 0) THEN                          ' Right whisker contacts
    GOSUB Back_Up                               ' Back up & turn left
    GOSUB Turn_Left
  ELSE                                           ' Both whiskers 1, no contacts
    GOSUB Forward_Pulse                         ' Apply a forward pulse
  ENDIF                                         ' and check again
LOOP

' -----[ Subroutines ]-----

Forward_Pulse:                                ' Send a single forward pulse.
  PULSOUT 13,850
  PULSOUT 12,650
  PAUSE 20
  RETURN

Turn_Left:                                    ' Left turn, about 90-degrees.
  FOR pulseCount = 0 TO 20
    PULSOUT 13, 650
    PULSOUT 12, 650
```

```

    PAUSE 20
  NEXT
  RETURN

Turn_Right:
  FOR pulseCount = 0 TO 20          ' Right turn, about 90-degrees.
    PULSOUT 13, 850
    PULSOUT 12, 850

    PAUSE 20
  NEXT
  RETURN

Back_Up:
  FOR pulseCount = 0 TO 40          ' Back up.
    PULSOUT 13, 650
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN

```

How Roaming with Whiskers Works

The **IF...THEN** statements in the Main Routine section first check the whiskers for any states that require attention. If both whiskers are pressed (**IN5** = 0 and **IN7** = 0), a U-turn is executed by calling the **Back_Up** subroutine followed by calling the **Turn_Left** subroutine twice in a row. If just the left whisker is pressed (**IN5** = 0), then the program calls the **Back_Up** subroutine followed by the **Turn_Right** subroutine. If the right whisker is pressed (**IN7** = 0), the **Back_Up** subroutine is called, followed by the **Turn_Left** subroutine. The only possible combination that has not been covered is if neither whisker is pressed (**IN5** = 1 and **IN7** = 1). The **ELSE** command calls the **Forward_Pulse** subroutine in this case.

```

  IF (IN5 = 0) AND (IN7 = 0) THEN
    GOSUB Back_Up
    GOSUB Turn_Left
    GOSUB Turn_Left
  ELSEIF (IN5 = 0) THEN
    GOSUB Back_Up
    GOSUB Turn_Right
  ELSEIF (IN7 = 0) THEN
    GOSUB Back_Up
    GOSUB Turn_Left
  ELSE
    GOSUB Forward_Pulse
  ENDIF

```

The **Turn_Left**, **Turn_Right**, and **Back_Up** subroutines should look fairly familiar, but the **Forward_Pulse** subroutine has a twist. It just sends one pulse, then returns. This is really important, because it means the Boe-Bot can check its whiskers between each forward pulse. That means the Boe-Bot checks for obstacles roughly 40 times per second as it travels forward.

```
Forward_Pulse:
  PULSOUT 12,650
  PULSOUT 13,850
  PAUSE 20
  RETURN
```

Since each full speed forward pulse makes the Boe-Bot roll around half a centimeter, it's a really good idea to only send one pulse, then go back and check the whiskers again. Since the **IF...THEN** statement is inside a **DO...LOOP**, each time the program returns from a **Forward_Pulse**, it gets to **LOOP**, which sends the program back up to **DO**. What happens then? The **IF...THEN** statement checks the whiskers all over again.

Your Turn

The **FOR...NEXT** loop **EndValue** arguments in the **Back_Right** and **Back_Left** routines can be adjusted for more or less turn, and the **Back_Up** routine can have its **EndValue** adjusted to back up less for navigation in tighter spaces.

- ✓ Experiment with the **FOR...NEXT** loop **EndValue** arguments in the navigation routines in `RoamingWithWhiskers.bs2`.

You can also modify your **IF...THEN** statements to make the LED indicators from the previous activity broadcast what maneuver the Boe-Bot is in by adding **HIGH** and **LOW** commands to control the LED circuits. Here is an example.

```
IF (IN5 = 0) AND (IN7 = 0) THEN
  HIGH 10
  HIGH 1
  GOSUB Back_Up
  GOSUB Turn_Left
  GOSUB Turn_Left
ELSEIF (IN5 = 0) THEN
  HIGH 10
  GOSUB Back_Up
```

```

        GOSUB Turn_Right
    ELSEIF (IN7 = 0) THEN
        HIGH 1
        GOSUB Back_Up
        GOSUB Turn_Left
    ELSE
        LOW 10
        LOW 1
        GOSUB Forward_Pulse
    ENDIF

```

- ✓ Modify the **IF...THEN** statement in RoamingWithWhiskers.bs2 to make the Boe-Bot broadcast its maneuver using the LED indicators.

ACTIVITY #4: ARTIFICIAL INTELLIGENCE AND DECIDING WHEN YOU'RE STUCK

You may have noticed that the Boe-Bot gets stuck in corners. As the Boe-Bot enters the corner, its whisker touches the wall on the left, so it turns right. When the Boe-Bot moves forward again, its right whisker bumps the wall on the right, so it turns left. Then it turns and bumps the left wall again, and the right wall again, and so on, until somebody rescues it from its predicament.

Programming to Escape Corners

RoamingWithWhiskers.bs2 can be modified to detect this problem and act upon it. The trick is to count the number of times that alternate whiskers are contacted. One important thing about this trick is that the program has to remember what state each whisker was in during the previous contact. It has to compare that to the whisker states of the current contact. If they are opposite, then add one to the counter. If the counter goes over a threshold that you (the programmer) have determined, then, it's time to do a U-turn and reset that alternate whisker counter.

This next program also relies on the fact that you can “nest” **IF...THEN** statements. In other words, the program checks for one condition, and if that condition is true, it checks for another condition within the first condition. Here is a pseudo code example of how it can be used.

```

IF condition1 THEN
    Commands for condition1
    IF condition2 THEN

```

```

        Commands for both condition2 and condition1
    ELSE
        Commands for condition1 but not condition2
    ENDIF
    ELSE
        Commands for not condition1
    ENDIF

```

There is an example of nested **IF...THEN** statements in the routine that detects consecutive alternate whisker contacts in the next program.

5

Example Program: EscapingCorners.bs2

This program will cause your Boe-Bot to execute a U-turn at either the fourth or fifth alternate corner, depending on which whisker was pressed first.

- ✓ Enter, save, and run EscapingCorners.bs2.
- ✓ Test this program by pressing alternate whiskers as the Boe-Bot roams. Depending on which Whisker you started with, the Boe-Bot should execute its U-Turn maneuver after either the fourth or fifth consecutive whisker press.

```

' -----[ Title ]-----
' Robotics with the Boe-Bot - EscapingCorners.bs2
' Boe-Bot navigates out of corners by detecting alternating whisker presses.

' {$STAMP BS2}                ' Stamp directive.
' {$PBASIC 2.5}              ' PBASIC directive.

DEBUG "Program Running!"

' -----[ Variables ]-----

pulseCount    VAR    Byte      ' FOR...NEXT loop counter.
counter       VAR    Nib       ' Counts alternate contacts.
old7          VAR    Bit       ' Stores previous IN7.
old5          VAR    Bit       ' Stores previous IN5.

' -----[ Initialization ]-----

FREQOUT 4, 2000, 3000          ' Signal program start/reset.
counter = 1                    ' Start alternate corner count.
old7 = 0                      ' Make up old values.
old5 = 1

' -----[ Main Routine ]-----

```

```

DO

' --- Detect Consecutive Alternate Corners -----
' See the "How EscapingCorners.bs2 Works" section that follows this program.

IF (IN7 <> IN5) THEN
  IF (old7 <> IN7) AND (old5 <> IN5) THEN
    counter = counter + 1
    old7 = IN7
    old5 = IN5
    IF (counter > 4) THEN
      counter = 1
      GOSUB Back_Up
      GOSUB Turn_Left
      GOSUB Turn_Left
    ENDIF
  ELSE
    counter = 1
  ENDIF
ENDIF

' One or other is pressed.
' Different from previous.
' Alternate whisker count + 1.
' Record this whisker press
' for next comparison.
' If alternate whisker count = 4,
' reset whisker counter
' and execute a U-turn.
' ENDIF counter > 4.
' ELSE (old7=IN7) or (old5=IN5),
' not alternate, reset counter.
' ENDIF (old7<>IN7) and
' (old5<>IN5).
' ENDIF (IN7<>IN5).

' --- Same navigation routine from RoamingWithWhiskers.bs2 -----

IF (IN5 = 0) AND (IN7 = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Left
  GOSUB Turn_Left
ELSEIF (IN5 = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Right
ELSEIF (IN7 = 0) THEN
  GOSUB Back_Up
  GOSUB Turn_Left
ELSE
  GOSUB Forward_Pulse
ENDIF

' Both whiskers detect obstacle
' Back up & U-turn (left twice)
' Left whisker contacts
' Back up & turn right
' Right whisker contacts
' Back up & turn left
' Both whiskers 1, no contacts
' Apply a forward pulse
' and check again

LOOP

' -----[ Subroutines ]-----

Forward_Pulse:
  PULSOUT 13,850
  PULSOUT 12,650
  PAUSE 20
  RETURN
' Send a single forward pulse.

Turn_Left:
  FOR pulseCount = 0 TO 20
    PULSOUT 13, 650
  NEXT pulseCount
' Left turn, about 90-degrees.

```



```

    PULSOUT 12, 650
    PAUSE 20
  NEXT
  RETURN

Turn_Right:
  FOR pulseCount = 0 TO 20
    PULSOUT 13, 850
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN

Back_Up:
  FOR pulseCount = 0 TO 40
    PULSOUT 13, 650
    PULSOUT 12, 850
    PAUSE 20
  NEXT
  RETURN

```

How EscapingCorners.bs2 Works

Since this program is a modified version of `RoamingWithWhiskers.bs2`, only new features related to detecting and escaping corners are discussed here.

Three extra variables are created for detecting a corner. The nibble variable `counter` can store a value between 0 and 15. Since our target value for detecting a corner is 4, the size of the variable is reasonable. Remember that a bit variable can store a single bit, either a 1 or a 0. The next two variables (`old7` and `old5`) are both bit variables. These are also the right size for the job since they are used to store old values of `IN7` and `IN5`, which are also bit variables.

<code>counter</code>	VAR	Nib
<code>old7</code>	VAR	Bit
<code>old5</code>	VAR	Bit

These variables have to be initialized (given initial values). For the sake of making the program easier to read, `counter` is set to 1, and when it gets to 4 due to the fact that the Boe-Bot is stuck in a corner, it is reset to 1. The `old7` and `old5` variables have to be set so that it looks like one of the two whiskers was pressed some time before the program started. This has to be done because the routine for detecting alternate corners always compares an alternating pattern, either (`IN5 = 1` and `IN7 = 0`) or (`IN5 = 0` and `IN7 = 1`). Likewise, `old5` and `old7` have to be different from each other.

```

counter = 1
old7 = 0
old5 = 1

```

Now we get to the Detect Consecutive Alternate Corners section. The first thing we want to check for is if one or the other whisker is pressed. A simple way to do this is to ask “is **IN7** different from **IN5**?” In PBASIC, we can use the not-equal operator **<>** in an **IF** statement:

```
IF (IN7 <> IN5) THEN
```

If it is indeed one whisker that is pressed, the next thing to check for is whether or not it’s the exact opposite pattern as the previous time. In other words, is (**old7 <> IN7**) and is (**old5 <> IN5**)? If that’s true, then, it’s time to add one to the counter that tracks alternate whisker contacts. It’s also time to remember the current whisker pattern by setting **old7** equal to the current **IN7** and **old5** equal to the current **IN5**.

```

IF (old7 <> IN7) AND (old5 <> IN5) THEN
  counter = counter + 1
  old7 = IN7
  old5 = IN5

```

If it turns out that this is the fourth consecutive whisker contact, then it’s time to reset the counter to 1 and execute a U-turn.

```

IF (counter > 4) THEN
  counter = 1
  GOSUB Back_Up
  GOSUB Turn_Left
  GOSUB Turn_Left

```

This **ENDIF** ends the code block that is executed if **counter > 4**.

```
ENDIF
```

This **ELSE** statement is connected to the **IF (old7 <> IN7) AND (old5 <> IN5) THEN** statement. The **ELSE** statement covers what happens if the **IF** statement is not true. In other words, it must not be an alternate whisker that was pressed, so reset the counter because the Boe-Bot is not stuck in a corner.

```
ELSE
  counter = 1
```

This **ENDIF** statement ends the decision making process for the **IF (old7 <> IN7) AND (old5 <> IN5) THEN** statement.

```
ENDIF
ENDIF
```

5

The remainder of the program is the same as before.

Your Turn

One of the **IF...THEN** statements in `EscapingCorners.bs2` checks to see if `counter` has reached 4.

- ✓ Try increasing the value to 5 and 6 and note the effect.
- ✓ Try also reducing the value and see if it has any effect on normal roaming.

SUMMARY

In this chapter, instead of navigating from a pre-programmed list, the Boe-Bot was programmed to navigate based on sensory inputs. The sensory inputs used in this chapter were whiskers, which served as normally open contact switches. When properly wired, these switches can show one voltage (5 V) at the switch's contact point when it's open, and a different voltage (0 V) when it's closed. The BASIC Stamp I/O pin's input registers store "1" if they detect Vdd (5 V) and "0," if they detect Vss (0 V).

The BASIC Stamp was programmed to test the whisker sensors and display the test results using two different media, the Debug Terminal and LEDs. PBASIC programs were developed to make the BASIC Stamp check the whiskers between each servo pulse. Based on the state of the whiskers, **IF...THEN** statements in the program's Main Routine section called navigation subroutines similar to the ones developed in the previous chapter to guide the Boe-Bot away from obstacles. As an example of artificial intelligence, an additional routine was developed that enabled the Boe-Bot to detect when it got stuck in a corner. This routine involved storing old whisker states, comparing them against the current whisker states, and counting the number of alternate object detections.

This chapter introduced sensor-based Boe-Bot navigation. The next three chapters will focus on using different types of sensors to give the Boe-Bot vision. Both vision and touch open up lots of opportunities for the Boe-Bot to navigate in increasingly complex environments.

Questions

1. What kind of electrical connection is a whisker?
2. When a whisker is pressed, what voltage occurs at the I/O pin monitoring it? What binary value will occur in the input register? If I/O pin P8 is used to monitor the input pin, what value does **IN8** have when a whisker is pressed, and what value does it have when a whisker is not pressed?
3. If **IN7 = 1**, what does that mean? What does it mean if **IN7 = 0**? How about **IN5 = 1** and **IN5 = 0**?
4. What command is used to jump to different subroutines depending on the value of a variable? What command is used to decide which subroutine to jump to? What are these decisions based on?
5. What is the purpose of having nested **IF...THEN** statements?

Exercises

1. Write a **DEBUG** command for TestWhiskers.bs2 that updates each whisker state on a new line. Adjust the **PAUSE** command so that it is 250 instead of 50.
2. Using RoamingWithWhiskers.bs2 as a reference, write a **Turn_Away** subroutine that calls the **Back_Up** subroutine once and the **Turn_Left** subroutine twice. Write down the modifications you will have to make to the Main Routine section of RoamingWithWhiskers.bs2.

5

Projects

1. Modify RoamingWithWhiskers.bs2 so that the Boe-Bot makes a 4 kHz beep that lasts 100 ms before executing the evasive maneuver. Make it beep twice if both whisker contacts are detected during the same sample.
2. Modify RoamingWithWhiskers.bs2 so that the Boe-Bot roams in a 1 yard (or meter) diameter circle. When you touch one whisker, it will cause the Boe-Bot to travel in a tighter circle (smaller diameter). When you touch the other whisker, it will cause the Boe-Bot to navigate in a wider diameter circle.

Solutions

- Q1. A tactile switch.
- Q2. Zero (0) volts, resulting in Binary zero (0) at the input register.
IN8 = 0 when whisker is pressed. IN8 = 1 when whisker is not pressed.
- Q3. IN7 = 1 means the right whisker is not pressed.
IN7 = 0 means the right whisker is pressed.
IN5 = 1 means the left whisker is not pressed.
IN5 = 0 means the left whisker is pressed.
- Q4. The **GOSUB** command performs the actual jump. The **IF...THEN** command is used to decide which subroutine to jump to. That decision is based on conditions, which are logical statements that evaluate to true or false.
- Q5. The program can check for one condition, and if that condition is true, it can check for another condition within the first condition.
- E1. The key to solving this problem is to use a second **CRSRXY** command that will place the right whisker state in the proper place on the screen. To line up with the headings, the text should start on column 9 of row 3.

```
' Robotics with the Boe-Bot - TestWhiskers_UpdateEaOnNewLine.bs2
' Update each whisker state on a new line.

' {$STAMP BS2}                                ' Stamp directive.
' {$PBASIC 2.5}                                ' PBASIC directive.

DEBUG "WHISKER STATES", CR,
      "Left      Right", CR,
      "-----  -----"

DO
  DEBUG CRSRXY, 0, 3, "P5 = ", BIN1 IN5        ' Print in Column 0,Row 3
  DEBUG CRSRXY, 9, 3, "P7 = ", BIN1 IN7        ' Print in Column 9,Row 3
  PAUSE 250                                     ' Change from 50 to 250
LOOP
```

- E2. Turn_Away:
- ```
GOSUB Back_Up
GOSUB Turn_Left
GOSUB Turn_Left
RETURN
```

To modify the **Main** Routine, replace the three **GOSUB** commands under the first **IF** condition with this single line:

```
GOSUB Turn_Away
```

- P1. The key to solving this problem is to write a statement that makes a beep with the required parameters:

```
FREQOUT 4, 100, 4000 ' 4kHz beep for 100ms
```

This statement must be added to the Main Routine in the appropriate places, as shown below. The rest of the program is unchanged.

```
' -----[Main Routine]-----

DO
 IF (IN5 = 0) AND (IN7 = 0) THEN ' Both whiskers detect
 FREQOUT 4, 100, 4000 ' 4 kHz beep for 100 ms
 FREQOUT 4, 100, 4000 ' Repeat twice
 GOSUB Back_Up ' Back up & U-turn
 GOSUB Turn_Left
 GOSUB Turn_Left
 ELSEIF (IN5 = 0) THEN ' Left whisker contacts
 FREQOUT 4, 100, 4000 ' 4 kHz beep for 100 ms
 GOSUB Back_Up ' Back up & turn right
 GOSUB Turn_Right
 ELSEIF (IN7 = 0) THEN ' Right whisker contacts
 FREQOUT 4, 100, 4000 ' 4 kHz beep for 100 ms
 GOSUB Back_Up ' Back up & turn left
 GOSUB Turn_Left
 ELSE ' Both whiskers 1, no
 GOSUB Forward_Pulse ' contacts
 ENDIF ' Apply a forward pulse
LOOP ' and check again
```

- P2. We found from Chapter 4 Projects that a 1 yard circle can be achieved with **PULSOUT 13, 850** and **PULSOUT 12, 716**. Using these values as the 1y circle, the radius can be adjusted by slightly increasing or decreasing the pulse width from the starting value of 716. Each time a whisker is pressed the program will add or subtract a bit from the right wheel's pulse width.

In the solution below, an audible beeping indicator was added. This acts as feedback to verify that the whisker was pressed. This is entirely optional.

```

' Robotics with the Boe-Bot - CirclingWithWhiskerInput.bs2
' Move in 1 yard circle, increase/decrease radius in response
' to whisker presses, one whisker increases, one decreases.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.
DEBUG "Program Running!"

' -----[Variables/Initialization]-----

pulseWidth VAR Word ' Signal sent to servo
toneFreq VAR Word ' Frequency of beeping tone
pulseWidth = 716 ' Found in Ch4 to make 1y circle
toneFreq = 4000 ' Beginning tone is 4 kHz

' -----[Main Routine]-----

DO

 PULSOUT 13, 850 ' Pulse servos in circular path
 PULSOUT 12, pulseWidth ' 12 slower than 13 so it arcs
 PAUSE 20

 IF (IN5 = 0) THEN ' Left whisker makes circle
 IF (pulseWidth <= 845) THEN ' smaller, down to servo max
 pulseWidth = pulseWidth + 5 ' pulseWidth of 850.
 toneFreq = toneFreq + 100
 FREQOUT 4, 100, toneFreq ' Play tone as indicator.
 ENDIF
 ELSEIF (IN7 = 0) THEN ' Right whisker makes circle
 IF (pulseWidth >= 655) THEN ' larger, down to servo min
 pulseWidth = pulseWidth - 5 ' pulseWidth of 650.
 toneFreq = toneFreq - 100
 FREQOUT 4, 100, toneFreq ' Play tone as indicator.
 ENDIF
 ENDIF

LOOP

```



## Chapter 6: Light Sensitive Navigation with Photoresistors

Light has many applications in robotics and industrial control. Some examples include sensing the edge of a roll of fabric in the textile industry, determining when to activate streetlights at different times of the year, when to take a picture, or when to deliver water to a crop of plants.

There are many different light sensors that serve unique functions. The light sensor in your Boe-Bot kit is designed to detect visible light, and it can be used to make your Boe-Bot detect variations in light level. With this ability, your Boe-Bot can be programmed to recognize areas with light or dark perimeters, report the overall brightness and darkness level it sees, and seek out light sources such as flashlight beams and doorways letting light into dark rooms.

6

### INTRODUCING THE PHOTORESISTOR

The resistors you worked with in previous chapters had fixed values, such as 220  $\Omega$  and 10 k $\Omega$ . The photoresistor, on the other hand, is a light dependent resistor (LDR). This means that its resistance value depends on the brightness, or illuminance, of the light that shines on its light detecting surface. Figure 6-1 shows the schematic symbol and part drawing for the photoresistor you will use to make the Boe-Bot able to detect variations in light levels.

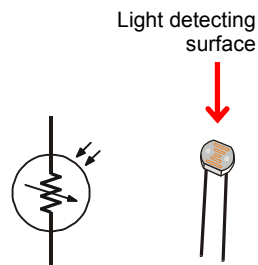


**Figure 6-1**  
Photoresistor Schematic and  
Part Drawing



A **photoresistor** is a light-dependent resistor (LDR) that covers the spectral sensitivity similar to that of the human eye. In other words, the kind of light that your eye detects is the same kind of light that affects the photoresistor's resistance. The active elements of these photoresistors are made of Cadmium Sulfide (CdS). Light enters into the semiconductor layer applied to a ceramic substrate and produces free charge carriers. A defined electrical resistance is produced that is inversely proportional to the illumination intensity. In other words, darkness causes more resistance, and brightness causes less resistance.

**Illuminance** is a scientific name for the measurement of incident light. One way to understand incident light is to think about shining a flashlight at a wall. The focused beam that you see shining on the wall is incident light. The unit of measurement of illuminance is commonly the "foot-candle" in the English system or the "lux" in the metric system. While using the photoresistors we won't be concerned about lux levels, just whether illuminance is higher or lower in certain directions. The Boe-Bot can be programmed to use the relative light intensity information to make navigation decisions.

## ACTIVITY #1: BUILDING AND TESTING PHOTORESISTOR CIRCUITS

In this activity, you will build and test light level sensor circuits with photoresistors. Your light level sensor circuits will be able to detect the difference between shade and no shade. The PBASIC commands for determining whether a shadow is cast over the photoresistor will be very similar to those used to determine whether or not a whisker has contacted an object.

### Parts List:

- (2) Photoresistors - CdS
- (2) Resistors – 2 k $\Omega$  (red-black-red)
- (2) Resistors – 220  $\Omega$  (red-red-brown)
- (4) Jumper wires
- (2) Resistors – 470  $\Omega$  (yellow-violet-brown)
- (2) Resistors – 1 k $\Omega$  (brown-black-red)
- (2) Resistors – 4.7 k $\Omega$  (yellow-violet-red)
- (2) Resistors – 10 k $\Omega$  (brown-black-orange)

### Building the Photosensitive Eyes

Figure 6-2 shows the schematic and Figure 6-3 shows the wiring diagram for the photoresistor circuits you will use in this and the next two activities.

- √ Disconnect power from your board and servos.
- √ Build the circuit shown in Figure 6-2, using Figure 6-3 as a reference.

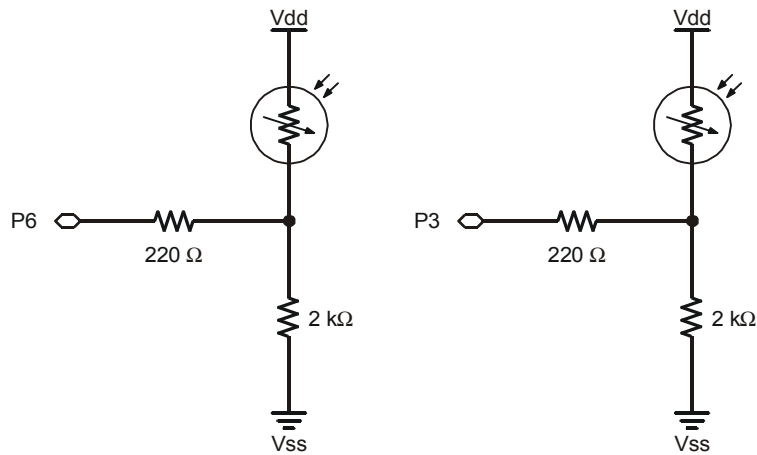


**Figure 6-2**  
Schematic –  
First Light  
Detection  
Circuit

6

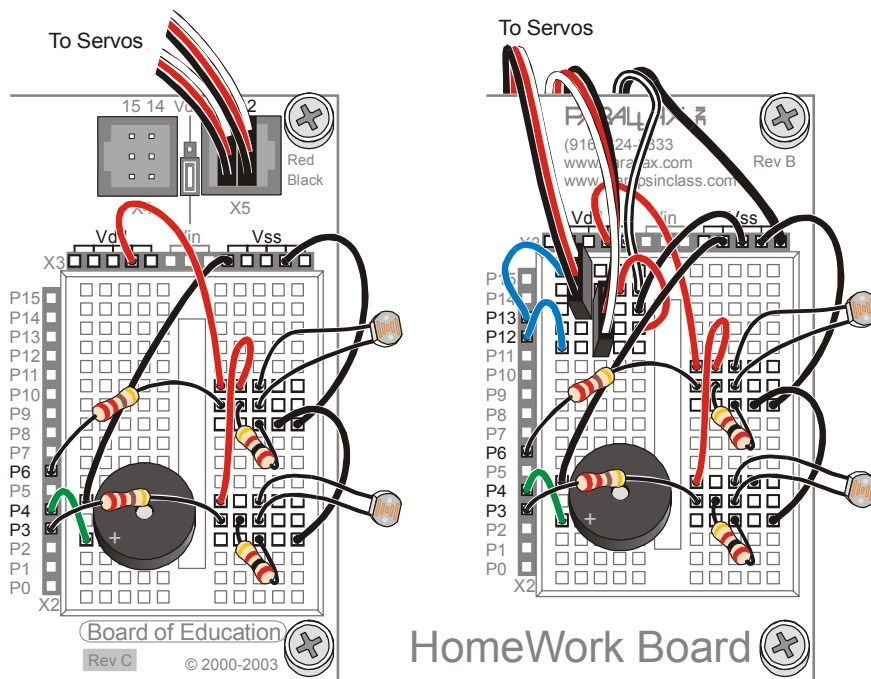


**Figure 6-3**  
Wiring  
Diagrams  
for the First  
Light  
Detection  
Circuit

*Board of  
Education  
(left) and  
HomeWork  
Board  
(right).*

### **How the Photoresistor Circuit Works**

A BASIC Stamp I/O pin can function as an output or an input. As an output, the I/O pin can send a high (5 V) or low (0 V) signal. Up to this point, high and low signals have been used to turn LED circuits on and off, control servos, and send tones to a speaker.

A BASIC Stamp I/O pin can also function as an input. As an input, the I/O pin does not apply any voltage to the circuit it is connected to. Instead, it just quietly listens without any actual effect on the circuit. In the previous chapter, these input registers stored values that indicated whether or not the whiskers were pressed. For example, the **IN7** input register stored a 1 when it sensed 5 V (whisker not pressed), or a 0 when it sensed 0 V (whisker pressed).

An I/O pin set to input doesn't actually need to have 5 V applied to it to make its input register store a 1. Anything above 1.4 V will make the input register for an I/O pin store a 1. Likewise, an I/O pin doesn't need 0 V to make its input register store a 0. Any voltage below 1.4 V will make an input register for an I/O pin store a 0.



**BASIC Stamp I/O pins are input by default.** When a BASIC Stamp program starts, all I/O pins start as inputs. When you use commands like **HIGH**, **LOW**, **PULSOUT** or **FREQOUT**, the I/O pin is changed from input to output so that the BASIC Stamp can send the high or low signals.

When a BASIC Stamp I/O pin is an input, the circuit behaves as though neither the I/O pin nor 220  $\Omega$  resistor is present. Figure 6-4 shows the equivalent circuit. The resistance of the photoresistor is shown as the letter R. It could be a few  $\Omega$  if the light is very bright, or it could be in the neighborhood of 50 k $\Omega$  in complete darkness. In a well lit room with fluorescent ceiling fixtures, the resistance could be as small as a 1 k $\Omega$  (full light exposure) or as large as 25 k $\Omega$  (shade cast around most of the object).

As the photoresistor's resistance changes with light exposure, so does the voltage at  $V_o$ ; as R gets larger,  $V_o$  gets smaller, and as R gets smaller,  $V_o$  gets larger.  $V_o$  is what the BASIC Stamp I/O pin is detecting when it is functioning as an input. If this circuit is connected to **IN6**, when the voltage at  $V_o$  is above 1.4 V, **IN6** will store a 1. If  $V_o$  falls below 1.4 V, **IN6** will store a 0.

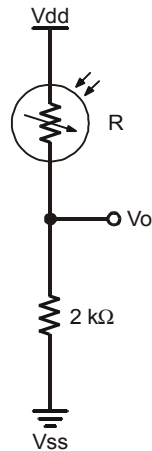


**Figure 6-4**  
Schematic –  
Voltage Divider  
Circuit

6

When resistors are connected end-to-end as shown in Figure 6-4 they are **connected in series**, and they can be referred to as series resistors.

When two resistors are connected in series to set a voltage at  $V_o$ , the circuit is called a **voltage divider**. In this circuit, the value of  $V_o$  can be anywhere between  $V_{dd}$  and  $V_{ss}$ . The exact value of  $V_o$  is determined by the ratio of  $R$  to  $2\text{ k}\Omega$ . When  $R$  is larger than  $2\text{ k}\Omega$ ,  $V_o$  will be closer to  $V_{ss}$ . When  $R$  is smaller than  $2\text{ k}\Omega$ ,  $V_o$  will be closer to  $V_{dd}$ . When  $R$  is equal to  $2\text{ k}\Omega$ ,  $V_o$  will be  $2.5\text{ V}$ . If you measure one of the two values ( $R$  or  $V_o$ ), you can calculate the other value using one of these two equations.



$$V_o = 5V \times \frac{2000\Omega}{2000\Omega + R}$$

$$R = \left( 5V \times \frac{2000\Omega}{V_o} \right) - 2000\Omega$$

$1.4\text{ V}$  is called the BASIC Stamp I/O pin's **threshold voltage**, also known as the I/O pin's **logic threshold**. When voltage sensed by an I/O pin is above that threshold, the I/O pin's input register will store a 1. If it is below that value, the I/O pin's input register will store a 0.

### Detecting Shadows

Casting a shadow makes the photoresistor's resistance value ( $R$ ) larger, which in turn makes  $V_o$  smaller. The  $2\text{ k}\Omega$  resistors were chosen to make the value of  $V_o$  reside slightly above the BASIC Stamp I/O pin's  $1.4\text{ V}$  threshold in a well lit room. When you cast a shadow over it with your hand, it should send  $V_o$  below the  $1.4\text{ V}$  threshold.

In a well lit room, both **IN6** and **IN3** will store the value 1. If you cast a shadow over the photoresistor divider connected to P6, it will then store a 0. Likewise, if you cast a shadow over the photoresistor divider connected to P3, it will cause **IN3** to store a 0.

**Example Program: TestPhotoresistorDividers.bs2**

This example program is TestWhiskers.bs2 adapted to the photoresistor dividers. Instead of monitoring P5 and P7 as we did with the whiskers, we are now monitoring P3 and P6, which are connected to the photoresistor divider circuits. This program should display a value of 1 on both sides in a well-lit room. When you cast a shadow over one or both of the photoresistors, their corresponding values should change to 0.

- ✓ Reconnect power to your board.
- ✓ Enter, save, and run TestPhotoresistorDividers.bs2.
- ✓ Verify that with no shade, both **IN6** and **IN3** store the value 1.
- ✓ Verify that you can use your hand to cast a shadow over each of the photoresistors and cause its input register to change from 1 to 0.
- ✓ If you are having difficulty, either with getting a shadow to change the input register to 0 or if the input registers store 0 regardless of whether or not you cast a shadow over them, see the Photoresistor Divider Troubleshooting box after the program listing. Work on it until your hand casting a shadow over the photoresistor reliably changes the state from 1 to 0.

```
' Robotics with the Boe-Bot - TestPhotoresistorDividers.bs2
' Display what the I/O pins connected to the photoresistor
' voltage dividers sense.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "PHOTORESISTOR STATES", CR,
 "Left Right", CR,
 "----- -----"

DO
 DEBUG CR$RXY, 0, 3,
 "P6 = ", BIN1 IN6,
 " P3 = ", BIN1 IN3
 PAUSE 100
LOOP
```

**Photoresistor Divider Troubleshooting**

General things to verify:

- ✓ Check your wiring and program for errors.
- ✓ Make sure that each component is firmly plugged into its socket.
- ✓ Check the color codes on your resistors. The resistors that connect between Vss and the photoresistors should be 2 k $\Omega$  (red-black-red). The resistors connecting P6 and P3 to the photoresistors should be 220  $\Omega$  (red-red-brown).

If either the IN3 or IN6 registers showed 0 regardless of whether or not you cast a shadow over them:



- ✓ If the room is dimly lit, consider bringing in some extra lamps. Alternately, you can replace the 2 k $\Omega$  resistors with 4.7 k $\Omega$  resistors (Yellow Violet Red). This will give your resistor divider better performance under lower lighting conditions. For really low lighting conditions, you can even use the 10 k $\Omega$  resistors (brown-black-orange).

If either the IN3 or IN6 registers showed 1 regardless of whether or not you cast a shadow over them:

- ✓ If the room is very brightly lit, and you find yourself having to cup your hand over the photoresistor's light collecting surface to make the 1 go to 0, you may need to substitute a lower value resistor in place of the 2 k $\Omega$ . Try 1 k $\Omega$  resistor (brown-black-red), or even a 470  $\Omega$  resistor (yellow-violet-brown) if you are outdoors.

6

**Your Turn – Experimenting with Different Voltage Dividers**

Depending on the lighting conditions in your robotics area, larger or smaller series resistors in place of the 2 k $\Omega$  resistors may improve the performance of your shadow detectors.

- ✓ Remember to disconnect power to your board during each circuit modification.
- ✓ Try replacing the 2 k $\Omega$  (red-black-red) resistors with each of the other resistor values you have gathered: 470  $\Omega$ , 1 k $\Omega$ , 4.7 k $\Omega$ , and 10 k $\Omega$ .
- ✓ Test each voltage divider combination with TestPhotoresistorDividers.bs2 and determine which resistors work best under your lighting conditions. The best combination is one that's not overly sensitive, but doesn't require you to cup your hand over the photoresistor either.
- ✓ Use the resistor combination that you think works best in the next two activities.

## ACTIVITY #2: ROAM AND AVOID SHADOWS LIKE OBJECTS

Since the photoresistor dividers behave similarly to whiskers, it's worth examining what's involved in adapting `RoamingWithWhiskers.bs2` so that it functions with the photoresistor dividers.

### Adapting `RoamingWithWhiskers.bs2` for the Photoresistor Dividers

All you really have to do is adjust the `IF...THEN` statements so that they monitor `IN6` and `IN3`, instead of `IN7` and `IN5`. Figure 6-5 demonstrates how to make these changes.

**Figure 6-5:** Modify `RoamingWithWhiskers.bs2` for Use with Photoresistor Dividers

|                                             |                                 |
|---------------------------------------------|---------------------------------|
|                                             | ' Modified for                  |
|                                             | ' RoamingWithPhotoresistor      |
| ' From <code>RoamingWithWhiskers.bs2</code> | ' <code>Dividers.bs2</code>     |
| IF (IN5 = 0) AND (IN7 = 0) THEN             | IF (IN6 = 0) AND (IN3 = 0) THEN |
| GOSUB Back_Up                               | GOSUB Back_Up                   |
| GOSUB Turn_Left                             | GOSUB Turn_Left                 |
| GOSUB Turn_Left                             | GOSUB Turn_Left                 |
| ELSEIF (IN5 = 0) THEN                       | ELSEIF (IN6 = 0) THEN           |
| GOSUB Back_Up                               | GOSUB Back_Up                   |
| GOSUB Turn_Right                            | GOSUB Turn_Right                |
| ELSEIF (IN7 = 0) THEN                       | ELSEIF (IN3 = 0) THEN           |
| GOSUB Back_Up                               | GOSUB Back_Up                   |
| GOSUB Turn_Left                             | GOSUB Turn_Left                 |
| ELSE                                        | ELSE                            |
| GOSUB Forward_Pulse                         | GOSUB Forward_Pulse             |
| ENDIF                                       | ENDIF                           |

### **Example Program – `RoamingWithPhotoresistorDividers.bs2`**

- ✓ Open the program `RoamingWithWhiskers.bs2` from page 179, and save it as `RoamingWithPhotoresistorDividers.bs2`.
- ✓ Make the modifications shown in Figure 6-5.
- ✓ Reconnect power to your board and servos.
- ✓ Run and test the program.





**Casting shadows over both photoresistors at the same time can be difficult.** When the Boe-Bot is going forward, it is checking the photoresistors around 40 times/second. You will have to move quickly to cast a shadow over both photoresistors between pulses. It helps to move your hand rapidly from no shade to full shade to trigger both photoresistors at once. Alternately, just leave your hand casting shade over both photoresistors while it executes a maneuver. When it returns from the maneuver and checks the photoresistors again, it should recognize that both photoresistors are in shade. □

- ✓ Verify that the Boe-Bot avoids shadows by using your hand to cast a shadow over the photoresistors. Try no shadow, a shadow over the right photoresistor divider (circuit connected to P3), a shadow over the left photoresistor divider (circuit connected to P7), and a shadow over both photoresistor dividers.
- ✓ Update the comments such as the title and descriptions of reactions to whisker presses to reflect the photoresistor circuit behavior. It should resemble the program below when you are done.

6

```
' -----[Title]-----
' Robotics with the Boe-Bot - RoamingWithPhotoresistorDividers.bs2
' Boe-Bot detects shadows photoresistors voltage divider circuit and turns
' away from them.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

DEBUG "Program Running!"

' -----[Variables]-----

pulseCount VAR Byte ' FOR...NEXT loop counter.

' -----[Initialization]-----

FREQOUT 4, 2000, 3000 ' Start/restart signal.

' -----[Main Routine]-----

DO
IF (IN6 = 0) AND (IN3 = 0) THEN ' Both photoresistors detects
 GOSUB Back_Up ' shadow, back up & U-turn
 GOSUB Turn_Left ' (left twice).
 GOSUB Turn_Left
ELSEIF (IN6 = 0) THEN ' Left photoresistor detects
 GOSUB Back_Up ' shadow, back up & turn right.
 GOSUB Turn_Right
ELSEIF (IN3 = 0) THEN ' Right photoresistor detects
 GOSUB Back_Up ' shadow, back up & turn left.
 GOSUB Turn_Left
```

```

ELSE ' Neither photoresistor detects
 GOSUB Forward_Pulse ' shadow, apply a forward pulse.
ENDIF

LOOP

' -----[Subroutines]-----

Forward_Pulse: ' Send a single forward pulse.
 PULSOUT 12,650
 PULSOUT 13,850
 PAUSE 20
 RETURN

Turn_Left: ' Left turn, about 90-degrees.
 FOR pulseCount = 0 TO 20
 PULSOUT 12, 650
 PULSOUT 13, 650
 PAUSE 20
 NEXT
 RETURN

Turn_Right: ' Right turn, about 90-degrees.
 FOR pulseCount = 0 TO 20
 PULSOUT 12, 850
 PULSOUT 13, 850
 PAUSE 20
 NEXT
 RETURN

Back_Up: ' Back up.
 FOR pulseCount = 0 TO 40
 PULSOUT 12, 850
 PULSOUT 13, 650
 PAUSE 20
 NEXT
 RETURN

```

### Your Turn – Improving performance

You can improve your Boe-Bot's performance by commenting some of the subroutine calls that were designed to help the Boe-Bot back away from obstacles and then turn to avoid them. Figure 6-6 shows an example where the two **Turn\_Left** subroutine calls are commented from the **IF...THEN** statement when the condition is that both photoresistors detect a shadow. Then, when only individual photoresistors detect shadows, the **Back\_Up** subroutine calls are commented so that the Boe-Bot only turns in response to a shadow.

**Figure 6-6: Modify RoamingWithPhotoresistorDividers.bs2**

|                                                                                                                                                                                                                                                                                                            |                                                                                                                                                                                                                                                                                                                             |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>' Excerpt from ' RoamingWithPhotoresistor ' Dividers.bs2  IF (IN6 = 0) AND (IN3 = 0) THEN   GOSUB Back_Up   GOSUB Turn_Left   GOSUB Turn_Left ELSEIF (IN6 = 0) THEN   GOSUB Back_Up   GOSUB Turn_Right ELSEIF (IN3 = 0) THEN   GOSUB Back_Up   GOSUB Turn_Left ELSE   GOSUB Forward_Pulse ENDIF</pre> | <pre>' Modified excerpt from ' RoamingWithPhotoresistor ' Dividers.bs2  IF (IN6 = 0) AND (IN3 = 0) THEN   GOSUB Back_Up   ' GOSUB Turn_Left   ' GOSUB Turn_Left ELSEIF (IN6 = 0) THEN   ' GOSUB Back_Up   GOSUB Turn_Right ELSEIF (IN3 = 0) THEN   ' GOSUB Back_Up   GOSUB Turn_Left ELSE   GOSUB Forward_Pulse ENDIF</pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

6

- ✓ Modify RoamingWithPhotoresistorDividers.bs2 as shown in the right side of Figure 6-6.
- ✓ Run the program, and compare the performance.

### ACTIVITY #3: A MORE RESPONSIVE SHADOW CONTROLLED BOE-BOT

By eliminating the **FOR...NEXT** loops in the navigation subroutines, you can make the Boe-Bot significantly more responsive. This wasn't really possible with the whiskers, because the Boe-Bot had to back up before turning since it had already made physical contact with the obstacle. When you are using shadows to guide the Boe-Bot, it can check between each pulse to see if the shadow is still detected regardless of whether it's moving forward or executing a maneuver.

#### A Simple Shadow Controlled Boe-Bot

One interesting form of remote control is to have the Boe-Bot sit still in normal light, then follow a shadow you cast over the photoresistors. It's kind of a user-friendly way of guiding the Boe-Bot's motion.

#### **Example Program – ShadowGuidedBoeBot.bs2**

When you run this next program, the Boe-Bot should stay still when no shadow is cast over its photoresistor dividers. When you cast a shadow over both photoresistors, the

Boe-Bot should move forward. If you cast a shadow over one of the photoresistors, the Boe-Bot should turn in the direction of the photoresistor that senses the shadow.

- ✓ Enter, save, and run ShadowGuidedBoeBot.bs2.
- ✓ Use your hand to cast shadows over the photoresistor dividers.
- ✓ Study this program carefully and make sure you understand how it works. It is very short, yet very powerful.

```
' Robotics with the Boe-Bot - ShadowGuidedBoeBot.bs2
' Boe-Bot detects shadows cast by your hand and tries to follow them.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

DEBUG "Program Running!"
FREQOUT 4, 2000, 3000 ' Start/restart signal.

DO

 IF (IN6 = 0) AND (IN3 = 0) THEN ' Both detect shadows, forward.
 PULSOUT 13, 850
 PULSOUT 12, 650
 ELSEIF (IN6 = 0) THEN ' Left detects shadow,
 PULSOUT 13, 750 ' pivot left.
 PULSOUT 12, 650
 ELSEIF (IN3 = 0) THEN ' Right detects shadow,
 PULSOUT 13, 850 ' pivot right.
 PULSOUT 12, 750
 ELSE
 PULSOUT 13, 750 ' No shadow, sit still
 PULSOUT 12, 750
 ENDIF

 PAUSE 20 ' Pause between pulses.

LOOP
```

### How ShadowGuidedBoeBot.bs2 Works

The **IF...THEN** statement in the **DO...LOOP** looks for one of the four possible shadow conditions: both, left, right, neither. Depending on which condition is detected, **PULSOUT** commands deliver pulses for one of the following maneuvers: forward, pivot right, pivot left, or sit still. Regardless of the condition, one of the four sets of pulses will be delivered each time through the **DO...LOOP**. After the **IF...THEN** statement, it's important to remember to include the **PAUSE 20** to ensure the low time between each pair of servo pulses.

### Your Turn – Condensing the Program

This program does not need the **ELSE** condition or the two **PULSOUT** commands that follow. If you deliver no pulses, the Boe-Bot will sit still, just as it should when you deliver pulses using 750 for the **PULSOUT Duration** argument.

- ✓ Try deleting (or commenting) this code block.  

```

ELSE
 PULSOUT 13, 750
 PULSOUT 12, 750

```
- ✓ Run the modified program.
- ✓ Can you detect any difference in the Boe-Bot's behavior?

6

### ACTIVITY #4: GETTING MORE INFORMATION FROM YOUR PHOTORESISTORS

The only information the BASIC Stamp was able to gather from the photoresistor divider circuits was whether the light level was above or below a threshold. This activity introduces a different circuit that the BASIC Stamp can monitor, and actually gather enough information from it to determine relative light levels. The value the BASIC Stamp gets from the circuit will range from small numbers, indicating bright light, to large numbers, indicating low light. This means no more manually replacing series resistors based on light levels. Instead, you will be able to adjust your program to look for different ranges of values.

#### Introducing the Capacitor

A capacitor is a device that stores charge, and it is a fundamental building block of many circuits. How much charge the capacitor tends to store is measured in farads (F). A farad is a very large value that's not practical for use with the Boe-Bot. The capacitors you will use in this activity store fractions of millionths of farads. A millionth of a farad is called a microfarad, and it's abbreviated  $\mu\text{F}$ . The capacitor you will use in this exercise stores one one-hundredth of a millionth of a farad. That's  $0.01 \mu\text{F}$ .



**Common capacitance measurements are:**

- Microfarads: (millionths of a Farad), abbreviated  $\mu\text{F}$       $1 \mu\text{F} = 1 \times 10^{-6} \text{ F}$
- Nanofarads: (billionths of a Farad), abbreviated nF      $1 \text{ nF} = 1 \times 10^{-9} \text{ F}$
- Picofarads: (trillionths of a Farad), abbreviated pF      $1 \text{ pF} = 1 \times 10^{-12} \text{ F}$

**The 103 on the 0.01  $\mu\text{F}$  capacitor's case** is a measurement picofarads or (pF). 103 is 10, with three zeros added, which is 10,000. Here is how to relate 103 to 0.01  $\mu\text{F}$ .

$10,000$  is  $10 \times 10^3$ .

$(10 \times 10^3) \times (1 \times 10^{-12}) \text{ F} = 10 \times 10^{-9} \text{ F}$

which is also  $0.01 \times 10^{-6} \text{ F}$

which is 0.01  $\mu\text{F}$ . □

Figure 6-7 shows the schematic symbol for a 0.01  $\mu\text{F}$  capacitor along with a drawing of the part in your Boe-Bot parts kit. The 103 marking on the capacitor indicates its value.

#### Parts List:

- (2) Photoresistors - CDS
- (2) Capacitors – 0.01  $\mu\text{F}$  (103)
- (2) Resistors - 220  $\Omega$   
(red-red-brown)
- (2) Jumper wires

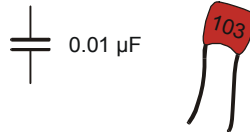


**Figure 6-7**  
Capacitor  
Schematic  
Symbol and  
Part Drawing



**There may also be 0.1  $\mu\text{F}$  capacitors marked 104 in your kit. Do not use them in these activities.**

✓ Make sure you have selected the 0.01  $\mu\text{F}$  capacitors (marked 103) for this activity.

The 0.1  $\mu\text{F}$  capacitors can be used in brightly lit areas, but they interfere with the Boe-Bot's performance in indoor and low lighting activities.

#### Rebuilding the Photosensitive Eyes

The circuit the BASIC Stamp can use to determine light levels is called a resistor/capacitor (RC) circuit. Figure 6-8 shows schematics of the Boe-Bot's RC light detection circuits and Figure 6-9 shows examples wiring diagrams for the Board of Education and the HomeWork Board.

- ✓ Disconnect power from your board and servos.
- ✓ Build the RC circuits shown in Figure 6-8 using Figure 6-9 as a reference.

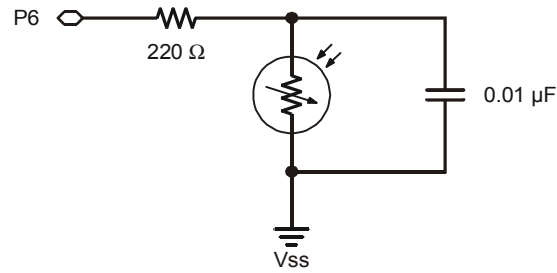
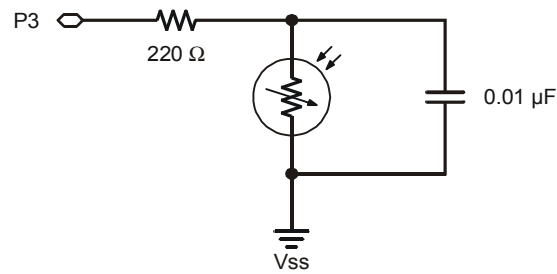


**Figure 6-8**  
Schematic - Two  
Photoresistor RC  
Circuits



*For measurement  
of resistance that  
varies with light.*

6

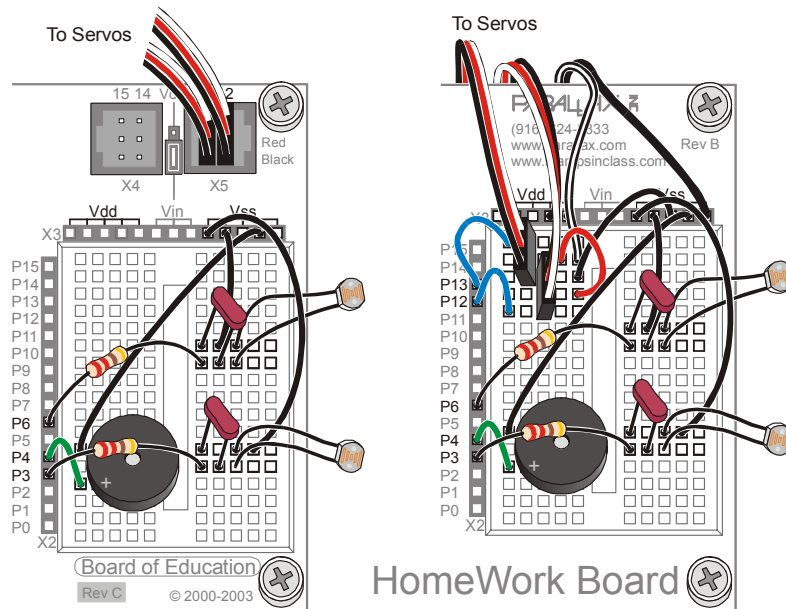


**Figure 6-9**  
Wiring  
Diagrams for  
Photoresistor  
Circuits

*Board of  
Education  
(left) and  
HomeWork  
Board (right).*

### About RC Decay Time and the Photoresistor Circuit

Think of a capacitor in the circuit shown in Figure 6-10 as a tiny rechargeable battery. When P6 sends the high signal, it essentially charges this capacitor-battery by applying 5 V to it. After a few ms, the capacitor charges up to almost 5 V. If the BASIC Stamp program then changes the I/O pin so that it just quietly listens, the capacitor loses its charge through the photoresistor. As the capacitor loses its charge through the photoresistor, its voltage decays, getting lower and lower as it loses charge. The amount of time it takes for the voltage that **IN6** senses to drop below 1.4 V depends on how strongly the photoresistor “resists” the flow of electric current supplied by the capacitor. If the photoresistor has a large resistance value due to very dim lighting conditions, the capacitor takes longer to discharge. If the photoresistor has a small resistance value because the light incident on its surface is very bright, it will not resist current very strongly, and the capacitor will lose its charge very rapidly.

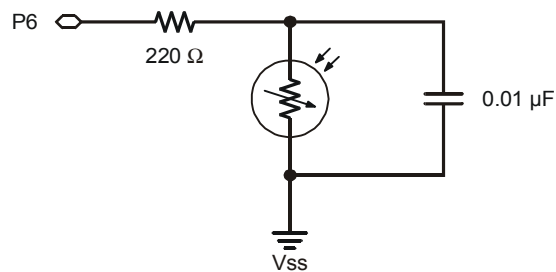


**Figure 6-10**  
RC Circuit  
Connected to I/O  
Pin



#### Connected in parallel

The photoresistor and capacitor shown in Figure 6-10 are connected in parallel. For two components to be connected in parallel, each of their leads must be connected to common terminals (also called nodes). The photoresistor and the capacitor each have one lead connected to Vss. They also each have one lead connected to the same 220 Ω resistor lead. □

### Measuring RC Decay Time with the BASIC Stamp

The BASIC Stamp can be programmed to charge the capacitor and then measure the time it takes the capacitor's voltage to decay to 1.4 V. This decay time measurement can be used to indicate the photoresistor's resistance. This resistance in turn indicates how bright the light detected by the photoresistor really is. This measurement requires a combination of the **HIGH** and **PAUSE** commands along with a new command called



**RCTIME.** The **RCTIME** command is designed to measure RC decay time on a circuit like the one in Figure 6-10. Here is the syntax for the **RCTIME** command:

**RCTIME Pin, State, Duration**

The **Pin** argument is the number of the I/O pin that you want to measure. For example, if you want to measure P6, the **Pin** argument should be 6. The **State** argument can either be 1 or 0. It should be 1 if the voltage across the capacitor starts above 1.4 V and decays downward. It should be 0 if the voltage across the capacitor starts below 1.4 V and grows upward. For the circuit in Figure 6-10, the voltage across the capacitor will start close to 5 V and decay to 1.4 V, so the **State** argument should be 1. The **Duration** argument has to be a variable that stores the time measurement, which is in 2  $\mu$ s units. In this next example program, we'll measure the RC decay time on the photoresistor circuit connected to P6, which is the photoresistor on the Boe-Bot's left.

6

To measure RC decay, the first thing you have to do is make sure you have declared a variable that will store the time measurement:

```
timeLeft VAR Word
```

These next three lines of code charge the capacitor, measure the RC decay time and then store it in the **timeLeft** variable.

```
HIGH 6
PAUSE 3
RCTIME 6,1,timeLeft
```

To get the measurement, the code implements these three steps:

1. Start charging the capacitor by connecting the circuit to 5 V (using the **HIGH** command).
2. Use **PAUSE** to give the **HIGH** command enough time to charge the capacitor in the RC circuit.
3. Execute the **RCTIME** command, which sets the I/O pin to input, measures the decay time (from almost 5 V to 1.4 V), and stores it in the **timeLeft** variable.

#### Example Program: TestP6Photoresistor.bs2

- ✓ Reconnect power to your board.
- ✓ Enter, save, and run TestP6Photoresistor.bs2.

- ✓ Cast a shadow over the photoresistor connected to P6 and verify that the time measurement gets larger as the environment gets darker.
- ✓ Point the photoresistor's light collecting surface directly at an overhead light, or shine flashlight directly at it. The time measurement should get very small. It should then get larger as you gradually direct the photoresistor further away from the light source. It should get even larger if you cast a shadow over it or turn out the lights.

```
' Robotics with the Boe-Bot - TestP6Photoresistor.bs2
' Test Boe-Bot photoresistor circuit connected to P6 and display
' the decay time.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

timeLeft VAR Word

DO

 HIGH 6
 PAUSE 2
 RCTIME 6,1,timeLeft

 DEBUG HOME, "timeLeft = ", DEC5 timeLeft
 PAUSE 100

LOOP
```

### Your Turn

- ✓ Save TestP6Photoresistor.bs2 as TestP3Photoresistor.bs2.
- ✓ Modify the program so that it performs the RC decay time measurement on the right photoresistor, the one connected to P3.
- ✓ Repeat the shadow and bright light tests with the P3 RC circuit and verify that it works correctly. You will need to modify the *Pin* arguments for both the **HIGH** and **RCTIME** commands, changing them from 6 to 3.

## **ACTIVITY #5: FLASHLIGHT BEAM FOLLOWING BOE-BOT**

In this activity, you will test and calibrate your Boe-Bot's light sensors so that they recognize the difference between ambient light and a directed flashlight beam. You will then program the Boe-Bot to follow the flashlight beam that is pointed at the surface in front of the Boe-Bot.

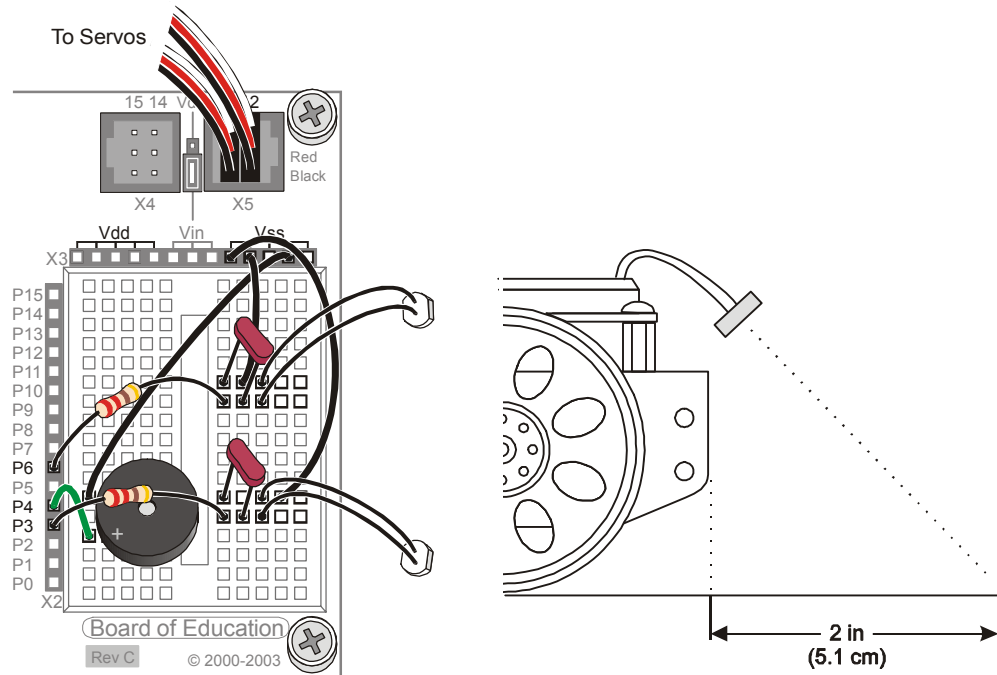
**Extra Equipment**

(1) Flashlight

**Adjust Sensors to Search for Flashlight Beam**

This activity works best if the photoresistors' light-collecting surfaces are pointing ahead at separate points on the ground about 2 in (5.1 cm) in front of the Boe-Bot.

- ✓ Point the light collecting surfaces of your photoresistors at the surface in front of the Boe-Bot as shown in Figure 6-11.

**Figure 6-11:** Photoresistor Orientation**6**



```

PAUSE 3
RCTYPE 6,1,timeLeft

HIGH 3 ' Right RC time measurement.
PAUSE 3
RCTYPE 3,1,timeRight

DEBUG CRSRXY, 0, 3, ' Display measurements.
 DEC5 timeLeft,
 " ",
 DEC5 timeRight

PAUSE 100

LOOP

```

6

### Your Turn

- ✓ Try facing the Boe-Bot in different directions, and repeat your measurements.
- ✓ For better results, you can average your measurements for "flashlight on" and "flashlight off" and replace the values in Table 6-1 with your average values.

### Following the Flashlight Beam

You have been using variable declarations up to this point. For example, **counter VAR nib** gives the name **counter** to a particular memory location in the BASIC Stamp's RAM. After you have declared the variable, every time you use **counter** in a PBASIC program, it uses the value stored at that particular location in the BASIC Stamp's RAM.

You can also declare constants. In other words, if you have a number you plan on using in your program, give it a useful name. Instead of the **VAR** directive, use the **CON** directive. Here are some **CON** directives from the next example program:

```

LeftAmbient CON 108
RightAmbient CON 114
LeftBright CON 20
RightBright CON 22

```

Now, everywhere in the program the name **LeftAmbient** is used, the BASIC Stamp will use the number 108. Everywhere **RightAmbient** is used, the BASIC Stamp will use the value 114. Likewise, everywhere **LeftBright** appears, it's really the value 20, and **RightBright** is 22. You will substitute your values from Table 6-1 before running the program.

Constants can even be used to calculate other constants. Here is an example of two constants, named `LeftThreshold` and `RightThreshold` that are calculated using the four constants just discussed. The `LeftThreshold` and `RightThreshold` constants are used in the program to figure out whether or not the flashlight beam has been detected.

|                                 | Average                                     | Scale factor         |
|---------------------------------|---------------------------------------------|----------------------|
| <code>LeftThreshold CON</code>  | <code>LeftBright + LeftAmbient / 2</code>   | <code>* 5 / 8</code> |
| <code>RightThreshold CON</code> | <code>RightBright + RightAmbient / 2</code> | <code>* 5 / 8</code> |

The math performed on these constants is an average, and then a scale. The average calculation for `LeftThreshold` is `LeftBright + LeftAmbient / 2`. That result is multiplied by 5 and divided by 8. This means that `LeftThreshold` is a constant whose value is the  $\frac{5}{8}$  of the average of `LeftBright` and `LeftAmbient`.

**Math expressions in PBASIC are executed from left to right.** First, `LeftBright` is added to `LeftAmbient`. This value is divided by 2. The result is then multiplied by 5 and divided by 8.

Let's try this: `LeftBright + LeftAmbient = 20 + 108 = 128`.

$$128 / 2 = 64.$$

$$64 * 5 = 320$$

$$320 / 8 = 40$$



**You can use parentheses to force a calculation that is further to the right in a line of PBASIC code to be completed first.** For example, you can rewrite this line of PBASIC code:

```
pulseRight = 2 - distanceRight * 35 + 750
```

like this:

```
pulseRight = 35 * (2 - distanceRight) + 750
```

In this expression, 35 is multiplied by the result of `(2 - distanceRight)`, then the product is added to 750.

### Example Program: FlashlightControlledBoeBot.bs2

- ✓ Enter `FlashlightControlledBoeBot.bs2` into the BASIC Stamp Editor.
- ✓ Substitute your `timeLeft` measurement with no flashlight beam (from Table 6-1) in place of the value 108 in the `LeftAmbient CON` directive.

- ✓ Substitute your `timeRight` measurement with no flashlight beam in place of the value 114 in the `RightAmbient CON` directive.
- ✓ Substitute your `timeLeft` measurement with focused flashlight beam in place of the value 20 in the `LeftBright CON` directive.
- ✓ Substitute your `timeRight` measurement with focused flashlight beam in place of the value 22 in the `RightBright CON` directive.
- ✓ Reconnect power to your board and servos.
- ✓ Save and then run `FlashlightControlledBoeBot.bs2`.
- ✓ Experiment and figure out exactly where to focus the light to get the forward, left turn, and right turn maneuvers to execute.
- ✓ Use the flashlight to guide your Boe-Bot through various obstacle courses and maneuvers.

```
' -----[Title]-----
' Robotics with the Boe-Bot - FlashlightControlledBoeBot.bs2
' Boe-Bot follows flashlight beam focused in front of it.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

DEBUG "Program Running!"

' -----[Constants]-----

' REPLACE THESE VALUES WITH THE VALUES YOU DETERMINED AND ENTERED INTO
' TABLE 6.1.

LeftAmbient CON 108
RightAmbient CON 114
LeftBright CON 20
RightBright CON 22

' Average Scale factor
LeftThreshold CON LeftBright + LeftAmbient / 2 * 5 / 8
RightThreshold CON RightBright + RightAmbient / 2 * 5 / 8

' -----[Variables]-----

' Declare variables for storing measured RC times of the
' left & right photoresistors.

timeLeft VAR Word
timeRight VAR Word

' -----[Initialization]-----
```

[illegible]



### How FlashlightControlledBoeBot.bs2 Works

These are the four constant declarations that you used with your own values from Table 6-1.

```
LeftAmbient CON 108
RightAmbient CON 114
LeftBright CON 20
RightBright CON 22
```

Now that the four constants have been declared, the next two lines average and scale the values to come up with threshold values for the program. These threshold values can be compared with the current **timeLeft** and **timeRight** measurements to determine whether the photoresistors are sensing ambient light or a focused beam.

```

' Average Scale
LeftThreshold CON LeftBright + LeftAmbient / 2 * 5 / 8
RightThreshold CON RightBright + RightAmbient / 2 * 5 / 8
```

These variables are used to store the **RCTIME** measurements.

```
timeLeft VAR Word
timeRight VAR Word
```

This is the reset indicator that has been used in most of the programs in this text.

```
FREQOUT 4, 2000, 3000
```

The Main Routine section contains just two subroutine calls. All the actual work in the program occurs in the two subroutines. **Test\_Photoresistors** takes the **RCTIME** measurements for both RC photoresistor circuits, and the **Navigate** subroutine makes the decisions and delivers the servo pulses.

```
DO

 GOSUB Test_Photoresistors
 GOSUB Navigate

LOOP
```

This is the subroutine that performs the **RCTIME** measurements on both photoresistor RC circuits. The measurement for the left circuit is stored in the **timeLeft** variable, and the measurement for the right circuit is stored in the **timeRight** variable.

```
Test_Photoresistors:

HIGH 6
PAUSE 3
RCTIME 6,1,timeLeft

HIGH 3
PAUSE 3
RCTIME 3,1,timeRight

RETURN
```

The Navigate subroutine uses an **IF...THEN** statement to compare the **timeLeft** variable against the **LeftThreshold** constant and the **timeRight** variable against the **RightThreshold** constant. Remember, when the **RCTIME** measurement is small, it means bright light is detected, and when it's large, it means the light is not as bright. So, when one of the variables that stores an **RCTIME** measurement is smaller than the threshold constant, it means the flashlight beam has been detected; otherwise, the flashlight beam has not been detected. Depending on which condition this subroutine detects (both, left, right or neither), the correct navigation pulses is applied, followed by a **PAUSE** before the **RETURN** command exits the subroutine.

```
Navigate:

IF (timeLeft<LeftThreshold) AND (timeRight<RightThreshold) THEN
 PULSOUT 13, 850
 PULSOUT 12, 650
ELSEIF (timeLeft < LeftThreshold) THEN
 PULSOUT 13, 700
 PULSOUT 12, 700
ELSEIF (timeRight < RightThreshold) THEN
 PULSOUT 13, 800
 PULSOUT 12, 800
ELSE
 PULSOUT 13, 750
 PULSOUT 12, 750
ENDIF
```

PAUSE 20

RETURN

**Your Turn – Adjusting the Performance and Changing the Behavior**

You can adjust the program's performance by adjusting the scale factor term in this constant declaration:

|                    | Average                        | Scale factor |
|--------------------|--------------------------------|--------------|
| LeftThreshold CON  | LeftBright + LeftAmbient / 2   | * 5 / 8      |
| RightThreshold CON | RightBright + RightAmbient / 2 | * 5 / 8      |

6

If you change the scale factor from  $\frac{5}{8}$  to  $\frac{1}{2}$ , it will make the Boe-Bot less sensitive to the flashlight, which may (or may not) lead to improved flashlight control.

- ✓ Try different scale factors, such as  $\frac{1}{4}$ ,  $\frac{1}{2}$ ,  $\frac{1}{3}$ ,  $\frac{2}{3}$ , and  $\frac{3}{4}$  and make notes about any differences in the way the Boe-Bot responded to the flashlight beam.

By modifying the **IF...THEN** statement in the example program, you can change the Boe-Bot's behavior so that it tries to get the light out of its eyes.

- ✓ Modify the **IF...THEN** statement so that the Boe-Bot backs up when it detects the flashlight beam with both photoresistor circuits and turns away if it detects the flashlight beam with only one of its photoresistor circuits.

**ACTIVITY #6: ROAMING TOWARD THE LIGHT**

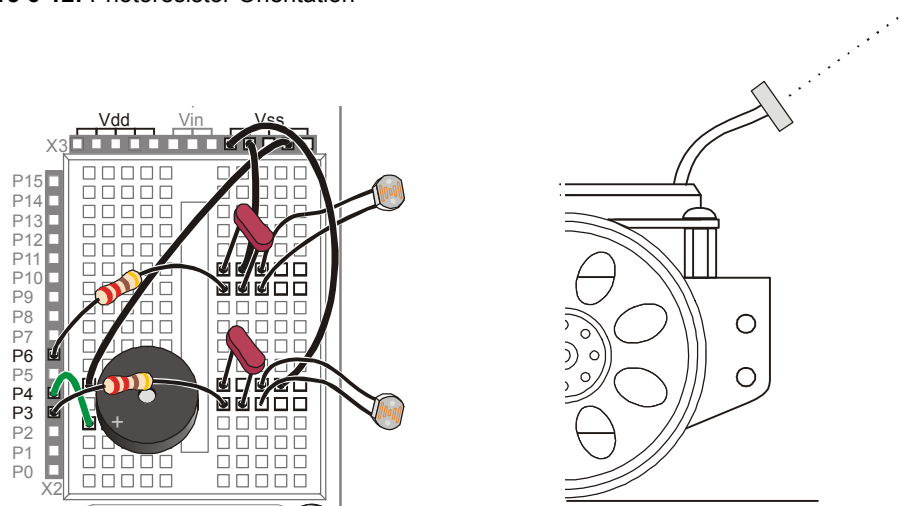
The example program in this activity can be used to guide the Boe-Bot through exiting a fairly dark room toward a doorway that's letting in brighter light. It also allows for much better control over the Boe-Bot's roaming by casting shadows over the photoresistors with your hand.

**Readjusting the Photoresistors**

This activity works best if the photoresistors' light collecting surfaces are pointing upwards and outwards.

- ✓ Point the light collecting surfaces of your photoresistors upward and outward shown in Figure 6-12.

Figure 6-12: Photoresistor Orientation



### Programming the Roaming Toward the Light Behavior

The key to roaming toward brighter light sources is going straight ahead when the differences between the photoresistor measurements are small, and turning toward the smaller photoresistor measurement when there is a large difference between the two measurements. In effect, this means the Boe-Bot will turn toward bright light.

Initially this seems like a simple enough programming task; **IF...THEN** reasoning like this example below should work. The problem is, it doesn't because the Boe-Bot gets stuck turning left and then right again because the change in `timeLeft` and `timeRight` is too large. Each time the Boe-Bot turns a little, the `timeRight` and `timeLeft` variables change so much that the Boe-Bot tries to correct and turn back. It never manages to get any forward pulses in.

```
IF (timeLeft > timeRight) THEN ' Turn right.
 PULSOUT 13, 850
 PULSOUT 12, 850
ELSEIF (timeRight > timeLeft) THEN ' Turn left.
```

```

PULSOUT 13, 650
PULSOUT 12, 650
ELSE ' Go forward.
PULSOUT 13, 850
PULSOUT 12, 650
ENDIF

```

Here is another code block that works a little better. This code block fixes the turning back and forth problem under certain conditions. The `timeLeft` variable now has to be larger than `timeRight` by a margin of 15 before the Boe-Bot will apply a left pulse. Likewise, `timeRight` has to be larger than `timeLeft` by 15 before the Boe-Bot adjusts to the left. This gives the Boe-Bot the opportunity to apply enough forward pulses before it has to correct with a turn, but only at certain light levels.

```

IF (timeLeft > timeRight + 15) THEN ' Turn right.
PULSOUT 13, 850
PULSOUT 12, 850
ELSEIF (timeRight > timeLeft + 15) THEN ' Turn left.
PULSOUT 13, 650
PULSOUT 12, 650
ELSE ' Go forward.
PULSOUT 13, 850
PULSOUT 12, 650
ENDIF

```

6

The problem with the code block above is that it works under medium dark conditions only. If you take it into a much darker area, the Boe-Bot starts turning back and forth again, and it never applies any forward pulses. If you take it into a brighter area, the Boe-Bot just goes forward, and never makes any adjustments to the left or right.

Why does that happen?

Here is the answer: When the Boe-Bot is in a dark part of a room, the measurement for each photoresistor will be large. For the Boe-Bot to decide to turn toward a light source, the difference between these two measurements has to be large. When the Boe-Bot is in a more brightly lit area, the measurement for each photoresistor will be smaller. For the Boe-Bot to decide to make a turn, the difference between photoresistor measurements also has to be much smaller than it was in the darker part of the room. The way to make this difference respond to the lighting conditions is to make it a variable that is a fraction of the average of `timeRight` and `timeLeft`. That way, it will always be the right value, regardless whether the lighting is bright or dim.

```

average = timeRight + timeLeft / 2
difference = average / 6

```

Now, the **difference** variable can be used in this **IF...THEN** statement, and it will be a large value when the lighting is low, and a small value when the lighting is bright.

```

IF (timeLeft > timeRight + difference) THEN ' Turn right.
 PULSOUT 13, 850
 PULSOUT 12, 850
ELSEIF (timeRight > timeLeft + difference) THEN ' Turn left.
 PULSOUT 13, 650
 PULSOUT 12, 650
ELSE
 ' Go forward.
 PULSOUT 13, 850
 PULSOUT 12, 650
ENDIF

```

### Example Program – RoamingTowardTheLight.bs2

Unlike RoamingWithPhotoresistorDividers.bs2 on page 201, this program will be very responsive to your hand casting a shadow over the photoresistor, regardless of whether the light is bright or dim. This program does not need to change resistors depending on the lighting conditions. Instead, it takes into account the lighting conditions and the sensitivity adjustment is made in software using the **average** and **difference** variables.



**For this program to work well, your photoresistors should respond similarly to similar light levels.** If the RC circuits are severely mismatched, your measurements from Table 6-1 will be very different under the same lighting conditions. You can correct these mismatched measurements using techniques discussed in Appendix F: Balancing Photoresistors.

This program measures the overall average of **timeLeft** and **timeRight** and uses it to set the **difference** between the **timeLeft** and **timeRight** measurements that's needed to justify delivering a turning pulse.

- ✓ Enter, save, and run RoamingTowardTheLight.bs2
- ✓ Take it to various areas, and let it roam, and verify that you can change its course by casting a shadow over one of the photoresistor RC circuits, regardless of the lighting conditions.

- ✓ Also try placing your Boe-Bot in a room that is poorly lit, but that has light streaming in through a doorway from an adjacent brightly lit room or hallway. See if the Boe-Bot can successfully find its way out the door.

```
' -----[Title]-----
' Robotics with the Boe-Bot - RoamingTowardTheLight.bs2
' Boe-Bot roams, and turns away from dark areas in favor of brighter areas.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

DEBUG "Program Running!"

' -----[Variables]-----

' Declare variables for storing measured RC times of the
' left & right photoresistors.

timeLeft VAR Word
timeRight VAR Word
average VAR Word
difference VAR Word

' -----[Initialization]-----

FREQOUT 4, 2000, 3000

' -----[Main Routine]-----

DO
 GOSUB Test_Photoresistors
 ' For mismatched photoresistors, use Appendix F, uncomment and use next line.
 ' timeLeft = (timeLeft */ 351) + 7 ' Replace 351 and 7 with your own values.
 GOSUB Average_And_Difference
 GOSUB Navigate
LOOP

' -----[Subroutine - Test_Photoresistors]-----

Test_Photoresistors:

 HIGH 6 ' Left RC time measurement.
 PAUSE 3
 RCTIME 6,1,timeLeft

 HIGH 3 ' Right RC time measurement.
 PAUSE 3
 RCTIME 3,1,timeRight

 RETURN
```

```

' -----[Subroutine - Average_And_Difference]-----
Average_And_Difference:

 average = timeRight + timeLeft / 2
 difference = average / 6

 RETURN

' -----[Subroutine - Navigate]-----
Navigate:

 ' Shadow significantly stronger on left detector, turn right.
 IF (timeLeft > timeRight + difference) THEN
 PULSOUT 13, 850
 PULSOUT 12, 850
 ' Shadow significantly stronger on right detector, turn left.
 ELSEIF (timeRight > timeLeft + difference) THEN
 PULSOUT 13, 650
 PULSOUT 12, 650
 ' Shadows in same neighborhood of intensity on both detectors.
 ELSE
 PULSOUT 13, 850
 PULSOUT 12, 650
 ENDIF

 PAUSE 10

 RETURN

```



**Why PAUSE 10 instead of PAUSE 20?** Because the **Test\_Photoresistors** subroutine has two **PAUSE** commands adding up to 6 ms plus some extra time to execute the **RCTIME** commands. Both these factors add to the amount of time between servo pulses, so the **PAUSE** in the **Navigate** subroutine has to be reduced. After some trial and error experiments, **PAUSE 10** appeared to give the servos the most reliable performance over the widest range of light levels.

### Your Turn – Adjusting the Sensitivity to Differences in Light

Right now, the **difference** variable is the **average** divided by 6. You can divide **average** by a smaller value if you want to make the Boe-Bot less sensitive to differences in light or divide it by a larger value if you want to make the Boe-Bot more sensitive to differences in light level.



- ✓ Instead of the value 6, try dividing the **average** variable by the values 3, 4, 5, 7, and 9.
- ✓ Run the program and test the Boe-Bot's ability to exit a dark room with each denominator value.
- ✓ Decide what the optimum denominator value is.

```
Average_And_Difference:
```

```
 average = timeRight + timeLeft / 2
 difference = average / 6
```

```
RETURN
```

You can also change the denominator into a constant like this:

```
Denominator CON 6
```

Then, in your `Average_And_Difference` subroutine, you can replace 6 (or the optimum value that you determined) with the **Denominator** constant, like this:

```
Average_And_Difference:
```

```
 average = timeRight + timeLeft / 2
 difference = average / Denominator
```

```
RETURN
```

- ✓ Make the changes just discussed, and verify that the program still works correctly.

You can also use one less variable in this program. Notice that the only time the **average** variable is used is to temporarily hold the average value, then it gets divided by **Denominator** and stored in the **difference** variable. The **difference** variable is needed later, but the **average** variable is not. One way to fix this problem would be to simply use the **difference** variable in place of the **average** variable. It will work fine, and you would no longer need the **average** variable. Here is how the subroutine would look:

```
Average_And_Difference:
```

```
 difference = timeRight + timeLeft / 2
```

```

 difference = difference / Denominator

 RETURN

```

There is a better way though.

✓ Leave the **Average\_And\_Difference** routine like this:

```

Average_And_Difference:

 average = timeRight + timeLeft / 2
 difference = average / Denominator

 RETURN

```

✓ Next, make this change in the variable declarations:

**Figure 6-13:** Modify `RoamingTowardTheLight.bs2` to Save a Word of RAM

| ' Unchanged code |     |      | ' Changed to save Word of RAM |     |         |
|------------------|-----|------|-------------------------------|-----|---------|
| average          | VAR | Word | average                       | VAR | Word    |
| difference       | VAR | Word | difference                    | VAR | average |

We don't really need the **average** variable, but the program will make more sense to someone trying to understand it if we use the word **average** in the first line and the word **difference** in the second line. Here is how to create an alias name **difference** for the **average** variable.

```

difference VAR average

```

Now, both **average** and **difference** refer to the same word of RAM.

✓ Test your modified program and make sure it still works properly.

## SUMMARY

This chapter focused on measuring differences in light intensity and programming the Boe-Bot to act on these differences. A pair of cadmium sulfide (CdS) photoresistors were used to measure differences in visible light. The CdS photoresistors were first connected to resistors to form voltage dividers, and the BASIC Stamp monitored the voltage at the connection between the photoresistor and the fixed resistor. When this voltage dropped below or rose above 1.4 V the input register for the I/O pin connected to the circuit stored either a 0 or 1. The Boe-Bot was programmed to make decisions using these binary values in a manner similar to the whiskers.

The photoresistor divider technique works so long as the right resistors are chosen and the lighting doesn't change. However, a much more versatile way of detecting light levels with the BASIC Stamp is to use the CdS photoresistor in an RC circuit, charge the capacitor, and then measure the decay time. RC stands for resistor capacitor, and the capacitor was introduced in this chapter along with a circuit that makes it possible for the BASIC Stamp to measure RC decay time. This is easily done with the BASIC Stamp using the RCTIME command, which is specifically designed for measuring RC decay and growth times.

Constants were introduced as a way to substitute meaningful names for numbers that are used in a PBASIC program. Scaling and averaging were also introduced. Scaling was used to set a threshold value to indicate whether or not a flashlight beam was detected. It was also used to determine the average value of the light levels in an area based on the two photoresistor RC time measurements. This was used to create a threshold that automatically self-adjusted to the overall lighting conditions, eliminating the need to change resistors when the light levels change.



### Watch the Boe-Bot in Action at [www.parallax.com](http://www.parallax.com)!

You can see the Boe-Bot solving Chapter 6 Projects 1 and 2 along with other Robotics video clips in the Robo Video Gallery under the Robotics Menu at [www.parallax.com](http://www.parallax.com).

## Questions

1. How does the resistance of a photoresistor respond to bright and dim light? What happens if the light levels are between bright and dim?

2. Does an I/O pin have any effect on the circuit when it's set to input? What causes the input register for an I/O pin to hold a 1 or 0 when it's set to input?
3. What does threshold voltage mean? What's the threshold voltage of a BASIC Stamp I/O pin?
4. Referring to Figure 6-4 on page 197, what causes  $V_o$  to rise above or fall below a BASIC Stamp I/O pin's threshold voltage? What is it about the circuit that causes  $V_o$  to change value?
5. How does the program `ShadowGuidedBoeBot.bs2` differ from the program `RoamingWithPhotoresistorDividers.bs2`? What does this change in the Boe-Bot's performance?
6. What is a constant declaration? What does it do? How can you use one in a program?
7. How are math expressions evaluated in PBASIC?
8. What are the two examples in this chapter where PBASIC was used to calculate an average? How are they different? How are they the same?

### **Exercises**

1. Calculate  $V_o$  for Figure 6-4 on page 197 if  $R$  is 10 k $\Omega$ . Repeat for  $R = 30$  k $\Omega$ .
2. If  $V_o$  in Figure 6-4 on page 197 is 1.4 V, what's the value of  $R$ ? Repeat for  $V_o = 1$  V and  $V_o = 3$  V.
3. Assume you have three variable values: `firstValue`, `secondValue`, and `thirdValue`. Write a command that takes the average of these three values in a variable named `myAverage`. Write a command that stores 7/8 of the average value in a variable named `myScaledAverage`. Write the variable declarations needed to make your command able to run in a program, first with `myAverage` and `myScaledAverage` as separate variables, then with one of these variable names aliased as the other.

### **Projects**

1. With your Boe-Bot's photoresistors looking down in front of it, develop a program that makes your Boe-Bot recognize the difference between black and white. Find a large white surface and place dark-black sheets of paper on it. Develop a program that makes the Boe-Bot avoid the black sheets of paper. Hints: Make sure to test and understand what the Boe-Bot sees when it is focused on a black sheet of paper and what it sees when it is focused on a white background. Use example programs from the last three activities in this chapter. The RC decay time circuit and programs will be much more helpful for making

the program work than the photoresistor divider techniques. Also, make sure this obstacle course is in a uniformly lit area. Bright sunlight from windows, and shadows cast by onlookers can make the demonstration fail.

2. If you succeeded with project 1, experiment with confining the Boe-Bot so that it can only roam in a space that is enclosed by black sheets of paper.

## Solutions

- Q1. The resistance is small, a few ohms, if the light is bright. The resistance is large, around 50 k $\Omega$ , in dim light. For light levels between bright and dim, its resistance will be somewhere between the bright and dim values.
- Q2. No. The I/O pin just quietly listens without any actual effect on the circuit. The value of the applied voltage causes the input register to change what it stores. If the applied voltage is less than 1.4 volts it stores a 0. Otherwise it stores a 1.
- Q3. The threshold voltage is a value above which is a logic 1, below which is a logic 0. The threshold voltage of BASIC Stamp modules is 1.4 volts.
- Q4. The value of  $V_o$  is determined by the ratio of the resistors.  $V_o$  changes value because the resistor,  $R$ , changes value.  $R$  is a photoresistor which changes value depending on the amount of light falling upon it.
- Q5. It checks the sensors between each pulse, instead of having fixed maneuvers of many pulses. This makes the Boe-Bot much more responsive.
- Q6. A constant declaration tells the compiler the value of your constant. For example, `MaxTemp con 212` is a constant declaration. A constant gives a useful name to a number or value used in a program. To use a constant, type the constant name anywhere in the program where the value is needed.
- Q7. Expressions are evaluated from left to right. This is different than standard algebraic evaluation, where multiplication and division are evaluated before addition and subtraction.
- Q8. In `FlashLightControlledBoeBot.bs2`, averages were used to calculate lighting thresholds. In `RoamingTowardTheLight.bs2`, an average of the left and right readings is calculated. They differ in that the first is calculated statically in a constant declaration, and the second is calculated dynamically as the program runs. They are similar in that they both add two values together and divide by 2.

E1.

a) **R = 10 kOhm**

$$\begin{aligned}
 V_o &= 5V * (2000 / (2000 + R)) \\
 &= 5 * (2000 / (2000 + 10000)) \\
 &= 5 * (2000 / (12000)) \\
 &= 5 * (2 / 12) \\
 &= 5 * (1 / 6) \\
 &= 5 * 0.17 \\
 &= 0.83 \text{ Volts}
 \end{aligned}$$

If  $R = 10 \text{ kOhm}$ ,  $V_o = 0.83 \text{ V}$

b) **R = 30 kOhm**

$$\begin{aligned}
 V_o &= 5V * (2000 / (2000 + R)) \\
 &= 5 * (2000 / (2000 + 30000)) \\
 &= 5 * (2000 / (32000)) \\
 &= 5 * (2 / 32) \\
 &= 5 * (1 / 16) \\
 &= 5 * 0.06 \\
 &= 0.31 \text{ Volts}
 \end{aligned}$$

If  $R = 30 \text{ kOhm}$ ,  $V_o = 0.31 \text{ V}$

E2.

a)  $V_o = 1.4 \text{ V}$ 

$$\begin{aligned}
 R &= (5 * (2000/V_o)) - 2000 \\
 &= (5 * (2000/1.4)) - 2000 \\
 &= (5 * 1428.57) - 2000 \\
 &= 7142.86 - 2000 \\
 &= 5142.86 \\
 &= 5143 \text{ Ohm} \\
 \text{When } V_o &= 1.4\text{V}, R = 5143 \Omega
 \end{aligned}$$

b)  $V_o = 1.0 \text{ V}$ 

$$\begin{aligned}
 R &= (5 * (2000/1)) - 2000 \\
 &= (5 * 2000) - 2000 \\
 &= (10000) - 2000 \\
 &= 8000 \\
 &= 8 \text{ kOhm} \\
 \text{When } V_o &= 1.4\text{V}, R = 8 \text{ k}\Omega
 \end{aligned}$$

c)  $V_o = 3.0 \text{ V}$ 

$$\begin{aligned}
 R &= (5 * (2000/3.0)) - 2000 \\
 &= (5 * 666.67) - 2000 \\
 &= 3333.33 - 2000 \\
 &= 1333.33 \\
 &= 1333 \text{ Ohm} \\
 \text{When } V_o &= 1.4\text{V}, R = 1333 \Omega
 \end{aligned}$$

6

E3. The average of these three values in a variable named **myAverage**, storing 7/8 in**myScaledAverage:**

```
myAverage = firstValue + secondValue + thirdValue / 3
myScaledAverage = myAverage * 7 / 8
```

Declarations as separate variables:

```
myAverage VAR Word
myScaledAverage VAR Word
```

Declarations using aliasing:

```
myAverage VAR Word
myScaledAverage VAR myAverage
```

P1. The first step is to use "TestBothPhotoresistors.bs2" and determine the values for the white surface and the black paper. Similar to "FlashlightControlledBoeBot.bs2", these values can be coded as constants. Then, **IF...THEN** statements can be used to determine whether the values are above or below the average readings. (For the author's Boe-Bot, scaling was not necessary). Here's a program solution that makes the Boe-Bot recognize the difference between black and white surfaces.

```

' -----[Title]-----
' Robotics with the Boe-Bot - TestBlackWhiteLogic.bs2
' Calculate whether Boe-Bot is over black or white surface, and print.

' {$STAMP BS2} ' Stamp directive
' {$PBASIC 2.5} ' PBASIC directive.

' -----[Constants]-----

LeftWhite CON 16
RightWhite CON 33
LeftBlack CON 26
RightBlack CON 45
LeftAvg CON LeftWhite + LeftBlack / 2
RightAvg CON RightWhite + RightBlack / 2

' -----[Variables]-----

timeLeft VAR Word ' Left photoresistor reading
timeRight VAR Word ' Right photoresistor reading

' -----[Main Routine]-----
DO
 GOSUB Test_Photoresistors
 IF (timeLeft > LeftAvg) THEN
 DEBUG CRSRXY, 0, 0, "Left Black "
 ELSE
 DEBUG CRSRXY, 0, 0, "Left White "
 ENDIF
 IF (timeRight > RightAvg) THEN
 DEBUG CRSRXY, 13, 0, "Right Black", CR
 ELSE
 DEBUG CRSRXY, 13, 0, "Right White", CR
 ENDIF
LOOP

' -----[Subroutine - Test_Photoresistors]-----

Test_Photoresistors:
 HIGH 6 ' Left RC time Measurement.
 PAUSE 3
 RCTIME 6,1,timeLeft
 HIGH 3 ' Right RC time measurement.
 PAUSE 3
 RCTIME 3,1,timeRight
RETURN

```

To develop a program that makes the Boe-Bot avoid the black sheets of paper, the decision and navigation steps required are very similar to



"FlashlightControlledBoeBot.bs2" and "RoamingTowardTheLight.bs2". A sample solution is shown below.

```
' -----[Title]-----
' Robotics with the Boe-Bot - AvoidBlackSpots.bs2
' Boe-Bot avoids black pieces of paper.

' {$STAMP BS2} ' Stamp directive
' {$PBASIC 2.5} ' PBASIC directive.

' -----[Constants]-----

LeftWhite CON 16
RightWhite CON 33
LeftBlack CON 26
RightBlack CON 45
LeftAvg CON LeftWhite + LeftBlack / 2
RightAvg CON RightWhite + RightBlack / 2

' -----[Variables]-----

timeLeft VAR Word ' Left photoresistor
reading
timeRight VAR Word ' Right photoresistor
reading

' -----[Initialization]-----

FREQUOT 4, 2000, 3000

' -----[Main Routine]-----
DO
 GOSUB Test_Photoresistors
 GOSUB Navigate
LOOP

' -----[Subroutines]-----

Test_Photoresistors:
 HIGH 6 ' Left RC time
Measurement.
 PAUSE 3
 RCTIME 6,1,timeLeft
 HIGH 3 ' Right RC time
measurement.
 PAUSE 3
 RCTIME 3,1,timeRight
RETURN

Navigate:
```

```

' Both detect black paper, back up and make a noise
IF (timeLeft > LeftAvg) AND (timeRight > RightAvg) THEN
 PULSOUT 13, 650
 PULSOUT 12, 850
 FREQOUT 4, 20, 4400
' Beep instead of pause
' Left detects black paper, turn away to right, make a noise
ELSEIF (timeLeft > LeftAvg) THEN
 PULSOUT 13, 850
 PULSOUT 12, 850
 FREQOUT 4, 20, 2200
' Right detects black paper, turn away to left, make a noise
ELSEIF (timeRight > RightAvg) THEN
 PULSOUT 13, 650
 PULSOUT 12, 650
 FREQOUT 4, 20, 3300
' Neither detects black paper, go forward one pulse.
ELSE
 PULSOUT 13,850
 PULSOUT 12,650
 PAUSE 20
ENDIF
RETURN

```

Hints: Make sure to test and understand what the Boe-Bot sees when it is focused on a black sheet of paper and what it sees when it is focused on a white background. Use example programs from the last three activities in this chapter. The RC decay time circuit and programs will be much more helpful for making the program work than the photoresistor divider techniques. Also, make sure this obstacle course is in a uniformly lit area. Bright sunlight from windows, and shadows cast by onlookers can make the demonstration fail.

- P2. The "AvoidBlackSpots.bs2" program solution, above, works quite well to keep the Boe-Bot confined in a black-bordered space. A video clip of a Boe-Bot doing just this can be viewed at [www.parallax.com](http://www.parallax.com). Under the Robotics menu, look for Robo Video Gallery.

## Chapter 7: Navigating with Infrared Headlights

Today's hottest products seem to have one thing in common: wireless communication. Personal organizers beam data into desktop computers, and wireless remotes let us channel surf. Many remote controls and PDA's use signals in the infrared frequency range to communicate, below the visible light spectrum. With a few inexpensive and widely available parts, the BASIC Stamp can also receive and transmit infrared light signals.

### USING INFRARED HEADLIGHTS TO SEE THE ROAD

Detecting objects without whiskers doesn't require anything as sophisticated as machine vision. Some robots use RADAR or SONAR (sometimes called SODAR when used in air instead of water). An even simpler system is to use infrared light to illuminate the robot's path and determine when the light reflects off an object. Thanks to the proliferation of infrared (IR) remote controls, IR illuminators and detectors are readily available and inexpensive.

7

**Infrared:** Infra means below, so Infra-red is light (or electromagnetic radiation) that has lower frequency, or longer wavelength than red light. Table 7-1 shows the wavelengths for common colors along with the infrared spectrum. Our IR LED and detector work at 980 nm (nanometers) which is considered near infrared. Night-vision goggles and IR temperature sensing use far infrared wavelengths of 2000-10,000 nm, depending on the application. Table 7-1 shows the wavelengths for common colors along with the infrared spectrum.



**Table 7-1:** Colors and Approximate Wavelengths

| Color  | Wavelength | Color         | Wavelength  |
|--------|------------|---------------|-------------|
| Violet | 400        | Red           | 780         |
| Blue   | 470        | Near infrared | 800-1000    |
| Green  | 565        | Infrared      | 1000-2000   |
| Yellow | 590        | Far infrared  | 2000-10,000 |
| Orange | 630        |               |             |

## Infrared Headlights

The infrared object detection system we'll build on the Boe-Bot is like a car's headlights in several respects. When the light from a car's headlights reflects off obstacles, your eyes detect the obstacles and your brain processes them and makes your body guide the car accordingly. The Boe-Bot uses infrared LEDs for headlights as shown in Figure 7-1. They emit infrared, and in some cases, the infrared reflects off objects and bounces back in the direction of the Boe-Bot. The eyes of the Boe-Bot are the infrared detectors. The infrared detectors send signals indicating whether or not they detect infrared reflected off an object. The brain of the Boe-Bot, the BASIC Stamp, makes decisions and operates the servo motors based on this sensor input.

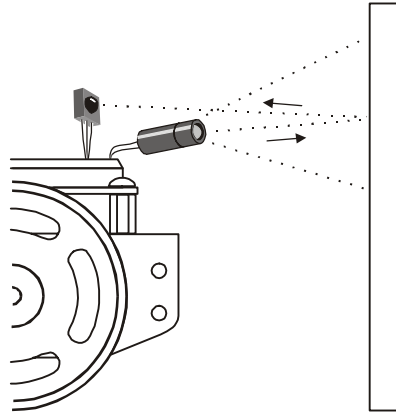


**Figure 7-1**  
Object Detection  
with IR Headlights

The IR detectors have built-in optical filters that allow very little light except the 980 nm infrared that we want to detect with its internal photodiode sensor. The infrared detector also has an electronic filter that only allows signals around 38.5 kHz to pass through. In other words, the detector is only looking for infrared that's flashing on and off 38,500 times per second. This prevents IR interference from common sources such as sunlight and indoor lighting. Sunlight is DC interference (0 Hz), and indoor lighting tends to flash on and off at either 100 or 120 Hz, depending on the main power source in the region. Since 120 Hz is outside the electronic filter's 38.5 kHz band pass frequency, it is completely ignored by the IR detectors.



**Some fluorescent lights do generate signals that can be detected by the IR detectors.** These lights can cause problems for your Boe-Bot's infrared headlights. One of the things you will do in this chapter is develop an infrared interference "sniffer" that you can use to test the fluorescent lights near your Boe-Bot courses.

## ACTIVITY #1: BUILDING AND TESTING THE IR PAIRS

In this activity, you will build and test the infrared transmitter/detector pairs.

### Parts List:

- (2) Infrared detectors
- (2) IR LEDs (clear case)
- (2) IR LED shield assemblies
- (2) Resistors - 220  $\Omega$  (red-red-brown)
- (2) Resistors - 1 k $\Omega$  (brown-black-red)

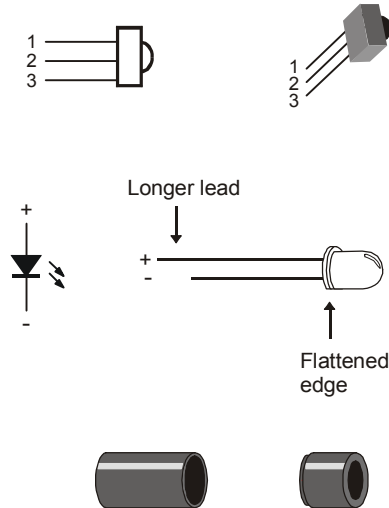


**Figure 7-2**  
New Parts  
Used in this  
Chapter

*IR detector  
(top)*

*IR LED  
(middle)*

*IR LED  
shield  
assembly  
(bottom)*

7

### Building the IR Headlights

- ✓ Insert the infrared LED into the shield assembly as shown in Figure 7-3.
- ✓ Make sure the LED snaps into the larger part of the housing.
- ✓ Snap the smaller part of the housing over the LED case and onto the larger part.

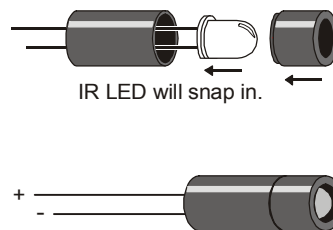


**Figure 7-3**  
Snapping the IR  
LED into the  
Shield Assembly

One IR pair (IR LED and detector) is mounted on each corner of the breadboard. Figure 7-4 shows the IR headlights circuit as a schematic and Figure 7-5 shows the circuit as a wiring diagram.

- ✓ Disconnect power from your board and servos.
- ✓ Build the circuit shown by the schematic in Figure 7-4, using the wiring diagram for your board in Figure 7-5 as a reference for parts placement.

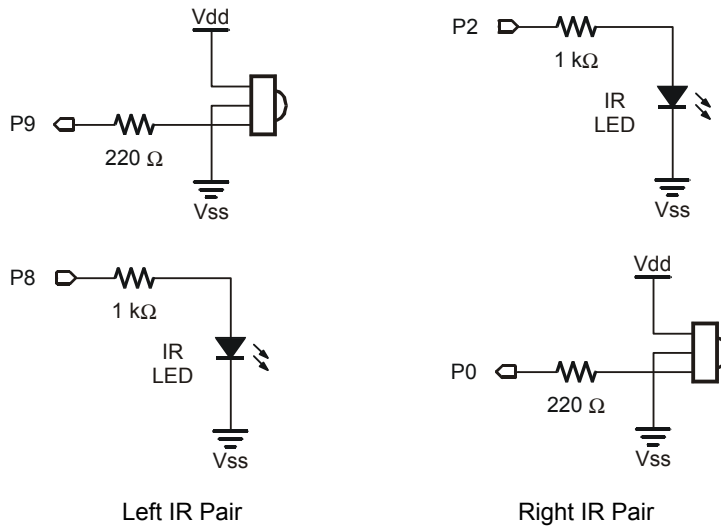
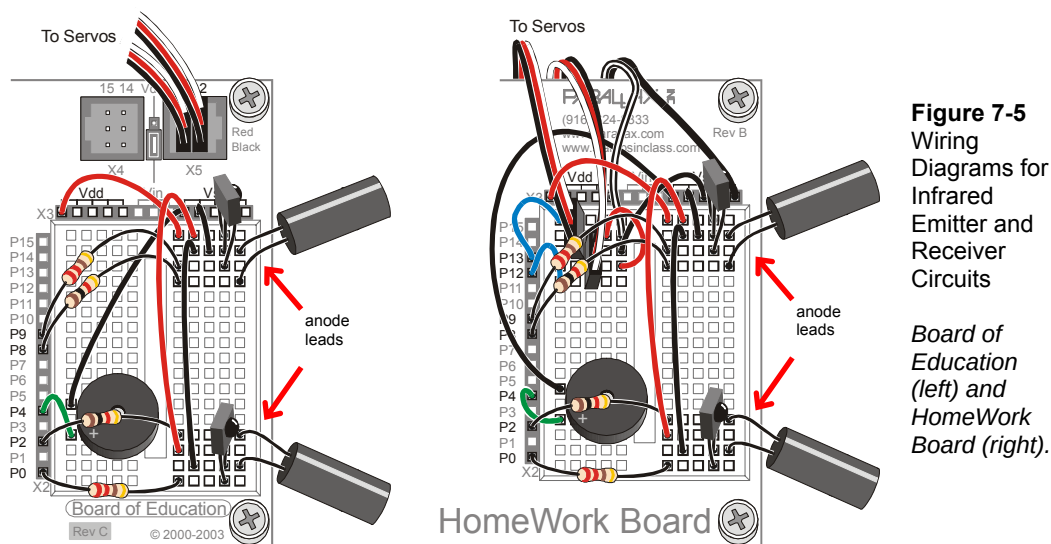


**Figure 7-4**  
Left and Right IR  
Pairs



**Watch your IR LED anodes and cathodes!**

Remember that the anode lead is the longer lead on an IR LED by convention, but that you need to check the LED's plastic case to make sure. The cathode lead is the one near the flat spot on the case. In Figure 7-5, the anode lead of each IR LED connects to a 1 kΩ resistor. The cathode lead plugs into the same breadboard row as an IR detector's center pin, and that row connects to Vss with a jumper wire.



## Testing the IR Pairs Using the FREQOUT Trick

The **FREQOUT** command was designed mainly to synthesize audio tones. The actual range of the **FREQOUT** command is 1 to 32768 Hz. One interesting phenomenon of digitally synthesized tones is that they contain signals called harmonics. A harmonic is a higher frequency tone that's mixed in with the tone you want to hear. These tones are outside human abilities to detect sound, which tend to range from 20 Hz to 20 kHz. The harmonics generated by the **FREQOUT** command start at 32769 Hz and go upward. You can directly control these harmonics using *Freq1* arguments above 32768. In this activity, you will use the command **FREQOUT 8, 1, 38500** to send a 38.5 kHz harmonic that lasts 1 ms to P8. The infrared LED circuit connected to P8 will broadcast this harmonic. If the infrared light is reflected back to the Boe-Bot by an object in its path, the infrared detector will send the BASIC Stamp a signal to let it know that the reflected infrared light was detected.

**FREQOUT** Command - Fundamentals and Harmonics

The fundamental frequency is the value of the **Freq1** argument when it's at or below 32768. For example, when you use the command **FREQOUT 4, 2000, 3000**, the fundamental frequency is 3000 Hz. That's the intended sound, but there is also a harmonic sound that accompanies it. This harmonic is a much higher frequency that the human ear can detect, in the neighborhood of 62.5 kHz. Here's how to calculate the harmonic frequency given the fundamental and visa versa.

Whenever you use the **FREQOUT** command to send a tone in this range, it contains that hidden (harmonic) tone as well. The equation for the harmonic is:

$$\text{harmonic frequency} = 65536 - \text{Freq1}, \quad \text{Freq1} \leq 32768$$

Whenever you use the **FREQOUT** command with a **Freq1** argument above 32768 to send a harmonic, it contains a fundamental tone. The equation for the fundamental is:

$$\text{fundamental frequency} = 65536 - \text{Freq1}, \quad \text{Freq1} > 32768$$

The key to making each IR LED/detector pair work is to send 1 ms of 38.5 kHz **FREQOUT** harmonic, and then, immediately store the IR detector's output in a variable. Here is an example that sends the 38.5 kHz signal to the IR LED connected to P8, then stores the IR detector's output, which is connected to P9, in a bit variable named **irDetectLeft**.

```
FREQOUT 8, 1, 38500
irDetectLeft = IN9
```

The IR detector's output state when it sees no IR signal is high. When the IR detector sees the 38500 Hz harmonic reflected by an object, its output is low. The IR detector's output only stays low for a fraction of a millisecond after the **FREQOUT** command is done sending the harmonic, so it's essential to store the IR detector's output in a variable immediately after sending the **FREQOUT** command. The value stored by the variable can then be displayed in the Debug Terminal or used for navigation decisions by the Boe-Bot.

**Example Program: TestLeftIrPair.bs2**

- ✓ Reconnect power to your board.
- ✓ Enter, save, and run TestLeftIrPair.bs2.

```
' Robotics with the Boe-Bot - TestLeftIrPair.bs2
' Test IR object detection circuits, IR LED connected to P8 and detector
' connected to P9.

' {$STAMP BS2}
```



```
' {$PBASIC 2.5}

irDetectLeft VAR Bit

DO

 FREQOUT 8, 1, 38500
 irDetectLeft = IN9

 DEBUG HOME, "irDetectLeft = ", BIN1 irDetectLeft
 PAUSE 100

LOOP
```

- ✓ Leave the Boe-Bot connected to the serial cable, because you will be using the Debug Terminal to test your IR pair.
- ✓ Place an object, such as your hand or a sheet of paper, about an inch from the left IR pair, in the manner shown in Figure 7-1 on page 236.
- ✓ Verify that when you place an object in front of the IR pair the Debug Terminal displays a 0, and when you remove the object from in front of the IR pair, it displays a 1.
- ✓ If the Debug Terminal displays the expected values for object not detected (1) and object detected (0), move on to the Your Turn section following the example program.
- ✓ If the Debug Terminal does not display the expected values, try the steps in the Trouble-Shooting box.

7

#### Trouble-Shooting

If the Debug Terminal does not display the expected values, check for circuit and program entry errors.



If you are always getting 0, even when an object is not placed in front of the Boe-Bot, there may be a nearby object that is reflecting the infrared. The surface of the table in front of the Boe-Bot is a common culprit. Move the Boe-Bot so that the IR LED and detector cannot possibly be reflecting off any nearby object.

If the reading is 1 most of the time when there is no object in front of the Boe-Bot, but flickers to 0 occasionally, it may mean you have interference from a nearby fluorescent light. Turn off any nearby fluorescent lights and repeat your tests.

#### Your Turn

- ✓ Save TestLeftIrPair.bs2 as TestRightIrPair.bs2.

- ✓ Change the **DEBUG** statement, title and comments to refer to the right IR pair.
- ✓ Change the variable name from **irDetectLeft** to **irDetectRight**. You will need to do this in four places in the program.
- ✓ Change the **FREQOUT** command's **Pin** argument from 8 to 2.
- ✓ Change the input register monitored by the **irDetectRight** variable from **IN9** to **IN0**.
- ✓ Repeat the testing steps in this activity for the right IR pair; with the IR LED circuit connected to P2 and the detector connected to P0.

## **ACTIVITY #2: FIELD TESTING FOR OBJECT DETECTION AND INFRARED INTERFERENCE**

In this activity, you will build and test indicator LEDs that will tell you if an object is detected without the help of the Debug Terminal. This is handy if you are not near a PC or laptop, and you need to trouble-shoot your IR detector circuits. You will also write a program to “sniff” for infrared interference from fluorescent lights. Some fluorescent lights send signals that resemble the signal sent by your infrared LEDs. The device inside a fluorescent light fixture that controls voltage for the lamp is called the ballast. Some ballasts operate in the same frequency range of your IR detector, 38.5 kHz, which in turn causes the lamp to emit a signal at this frequency. When you integrate IR object detection with navigation, this interference can cause some bizarre Boe-Bot behavior!

### **Rebuilding the LED Indicator Circuits**

These are the same LED indicator circuits that you used with the whiskers.

#### **Parts List:**

- (2) Red LEDs
- (2) Resistors – 220  $\Omega$  (red-red-brown)

- ✓ Disconnect power from your board and servos.
- ✓ Build the circuit shown in Figure 7-6 using Figure 7-7 as a reference.

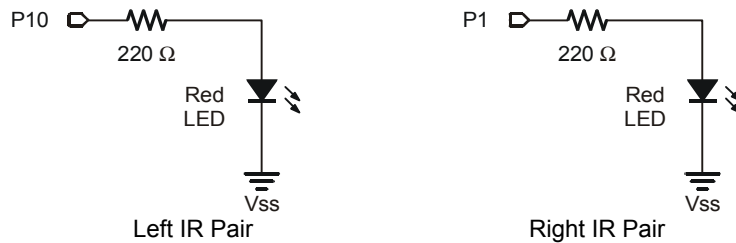


**Figure 7-6**  
Left and Right  
Indicator LEDs

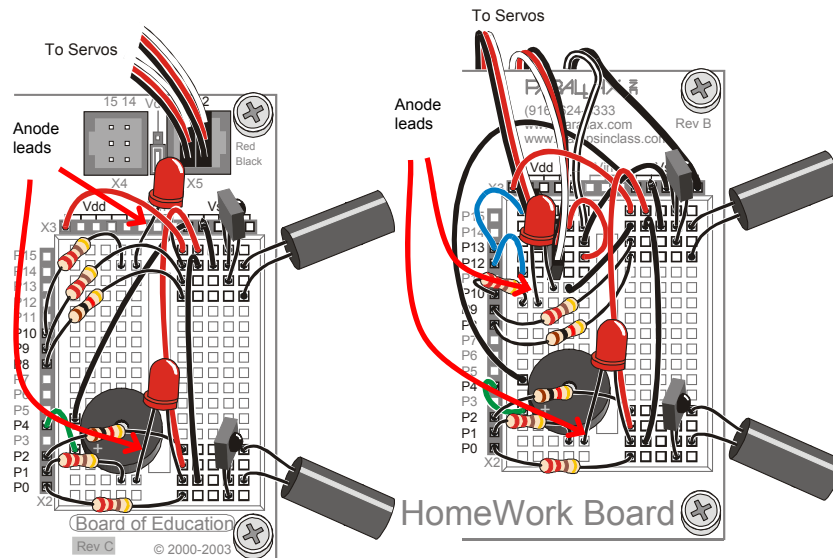


**Figure 7-7**  
Wiring  
Diagrams for  
Red LED  
Indicators  
with IR Object  
Detection  
Circuits

*Board of  
Education  
(left) and  
HomeWork  
Board (right).*

7

### Testing the System

There are quite a few components involved in this system, and this increases the likelihood of a wiring error. That's why it's important to have a test program that shows you what the infrared detectors are sensing. You can use this program to verify that all the circuits are working before unplugging the Boe-Bot from its serial cable and testing other objects.

### Example Program – TestIrPairsAndIndicators.bs2

- ✓ Reconnect power to your board.
- ✓ Enter, save, and run TestIrPairsAndIndicators.bs2.

- ✓ Verify that the speaker makes a clear, audible tone while the Debug Terminal displays “Testing piezospeaker...”.
- ✓ Use the Debug Terminal to verify that the BASIC Stamp still receives a zero from each IR detector when an object is placed in front of it.
- ✓ Verify that the LED next to each detector emits light when the detector detects an object. If one or both of the LEDs appear not to work, check your wiring and your program.

```
' Robotics with the Boe-Bot - TestIrPairsAndIndicators.bs2
' Test IR object detection circuits.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

' -----[Variables]-----
irDetectLeft VAR Bit
irDetectRight VAR Bit

' -----[Initialization]-----
DEBUG "Testing piezospeaker..."
FREQOUT 4, 2000, 3000

DEBUG CLS,
 "IR DETECTORS", CR,
 "Left Right", CR,
 "-----"

' -----[Main Routine]-----

DO

 FREQOUT 8, 1, 38500
 irDetectLeft = IN9

 FREQOUT 2, 1, 38500
 irDetectRight = IN0

 IF (irDetectLeft = 0) THEN
 HIGH 10
 ELSE
 LOW 10
 ENDIF

 IF (irDetectRight = 0) THEN
 HIGH 1
 ELSE
```

```

 LOW 1
 ENDIF

 DEBUG CRSXY, 2, 3, BIN1 irDetectLeft,
 CRSXY, 9, 3, BIN1 irDetectRight

 PAUSE 100

LOOP

```

### Your Turn – Remote Testing and Range Testing

You can now use your LED detectors to take your Boe-Bot and test your IR detectors on objects that might not otherwise be in reach of your computer's serial cable.

- ✓ Unplug your Boe-Bot from the serial cable, and take your Boe-Bot to a variety of objects and test the range of the IR detectors.
- ✓ Try the detection range of different colored objects. What color is detected at the furthest range? What color is detected at the closest range?

7

### Sniffing for IR Interference

If you happened to notice that your Boe-Bot let you know it detected something even though nothing was in range, it may mean that a nearby light is generating some IR light at a frequency close to 38.5 kHz. If you try to have a Boe-Bot contest or demonstration under one of these lights, your infrared systems might end up performing very poorly. The last thing anybody wants is to have their robot not perform as advertised during a public demonstration, so make sure to check any prospective demo area with this IR interference “sniffer” program before-hand.

The concept behind this program is simple, don't transmit any IR through the IR LEDs, just monitor to see if any IR is detected. If IR is detected, sound the alarm using the piezospeaker.



You can use a handheld remote for just about any piece of equipment to generate IR interference. TVs, VCRs, CD/DVD players, and projectors all use the same IR detectors you have on your Boe-Bot right now. Likewise, the remotes you use to control these devices all use the same kind of IR LED that's on your Boe-Bot to transmit messages to the IR detector in your TV, VCR, CD/DVD player, etc.

**Example Program – IrInterferenceSniffer.bs2**

- ✓ Enter, save, and run IrInterferenceSniffer.bs2.
- ✓ Test to make sure the Boe-Bot sounds the alarm when it detects IR interference. You can do this with a separate Boe-Bot that's running TestIrPairsAndIndicators.bs2. If you don't have a second Boe-Bot, just use a handheld remote for a TV, VCR, CD/DVD player, or projector. Simply point the remote at the Boe-Bot and press a button. If the Boe-Bot responds by sounding the alarm, you know your IR interference sniffer is working.

```
' Robotics with the Boe-Bot - IrInterferenceSniffer.bs2
' Test fluorescent lights, infrared remotes, and other sources
' of 38.5 kHz IR interference.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

counter VAR Nib

DEBUG "IR interference not detected, yet...", CR

DO
 IF (IN0 = 0) OR (IN9 = 0) THEN
 DEBUG "IR Interference detected!!!", CR
 FOR counter = 1 TO 5
 HIGH 1
 HIGH 10
 FREQOUT 4, 50, 4000
 LOW 1
 LOW 10
 PAUSE 20
 NEXT
 ENDIF
LOOP
```

**Your Turn – Testing for Fluorescent Lights that Interfere**

- ✓ Disconnect your Boe-Bot from its serial cable, and point it at any fluorescent light near where you plan to operate it. Especially if you get frequent alarms, turn off that fluorescent light before trying to use IR object detection under it.



**Always use this IrInterferenceSniffer.bs2 to make sure that any area where you are using the Boe-Bot is free of infrared interference.**

### ACTIVITY #3: INFRARED DETECTION RANGE ADJUSTMENTS

You may have noticed that brighter car headlights (or a brighter flashlight) can be used to see objects that are further away when it's dark. By making the Boe-Bot's infrared LED headlights brighter, you can also increase its detection range. By resisting electric current less, a smaller resistor allows more current to flow through an LED. More current through an LED is what causes it to glow more brightly. In this activity, you will examine the effect of different resistance values with both the red and infrared LEDs.

#### Parts List:

You will need some extra parts for this activity.

- (2) Resistors – 470  $\Omega$  (yellow-violet-brown)
- (2) Resistors – 220  $\Omega$  (red-red-brown)
- (2) Resistors – 2 k $\Omega$  (red-black-red)
- (2) Resistors – 4.7 k $\Omega$  (yellow-violet-red)

7

#### Series Resistance and LED Brightness

First, let's use one of the red LEDs to "see" the difference that a resistor makes in how brightly an LED glows. All we need to test the LED is a program that sends a high signal to the LED.

#### Example Program – P1LedHigh.bs2

- ✓ Enter, save and run P1LedHigh.bs2.
- ✓ Run the program and verify that the LED in the circuit connected to P1 emits light.

```
' Robotics with the Boe-Bot - P1LedHigh.bs2
' Set P1 high to test for LED brightness testing with each of
' these resistor values in turn: 220 ohm , 470 ohm, 1 k ohm.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

HIGH 1

STOP
```

The command **STOP** is used here rather than **END**, since **END** would put the BASIC Stamp into low power mode.

### Your Turn – Testing LED Brightness



**Remember to disconnect power before you make changes to a circuit.** Remember also that the same program will run again when you reconnect power, so you can pick up right where you left off with each test.

- ✓ Note how brightly the LED in the circuit connected to P1 is glowing with the 220  $\Omega$  resistor.
- ✓ Replace the 220  $\Omega$  resistor connected to P1 and the right LED's cathode with a 470  $\Omega$  resistor.
- ✓ Note now how brightly the LED glows.
- ✓ Repeat for a 2 k $\Omega$  resistor.
- ✓ Repeat once more with a 4.7 k $\Omega$  resistor.
- ✓ Replace the 4.7 k $\Omega$  resistor with the 220  $\Omega$  resistor before moving on to the next portion of this activity.
- ✓ Explain in your own words the relationship between LED brightness and series resistance.

### Series Resistance and IR Detection Range

We now know that less series resistance will make an LED glow more brightly. A reasonable hypothesis would be that brighter IR LEDs can make it possible to detect objects that are further away.

- ✓ Open and run TestIrPairsAndIndicators.bs2 (from page 244).
- ✓ Verify that both detectors are working properly.

### Your Turn – Testing IR LED Range

- ✓ With a ruler, measure the furthest distance from the IR LED that a sheet of paper facing the IR LED can be detected, with the 1 k $\Omega$  resistors in place, and record your data in Table 7-2.
- ✓ Replace the 1 k $\Omega$  resistors that connect P2 and P8 to the IR LED anodes with 4.7 k $\Omega$  resistors.
- ✓ Determine the furthest distance at which the same sheet of paper is detected, and record your data.



- ✓ Repeat with 2 k $\Omega$  resistors.
- ✓ Repeat with 470  $\Omega$  resistors.
- ✓ Repeat with 220  $\Omega$  resistors.

| <b>Table 7-2: Detection Distance vs. Resistance</b>    |                                                            |
|--------------------------------------------------------|------------------------------------------------------------|
| <b>IR LED Series Resistance, (<math>\Omega</math>)</b> | <b>Maximum Detection Distance, Circle one: ( in / cm )</b> |
| 4700                                                   |                                                            |
| 2000                                                   |                                                            |
| 1000                                                   |                                                            |
| 470                                                    |                                                            |
| 220                                                    |                                                            |

7

- ✓ Before moving on to the next activity, restore your IR pairs to their original configuration (with 1 k $\Omega$  resistors in series with each IR LED).
- ✓ Also, before moving on, make sure to test this last change with TestIrPairsAndIndicators.bs2 to verify that both IR LED/detector pairs are working properly.

#### ACTIVITY #4: OBJECT DETECTION AND AVOIDANCE

An interesting thing about the IR detectors is that their outputs are just like the whiskers. When no object is detected, the output is high; when an object is detected, the output is low. In this activity, RoamingWithWhiskers.bs2 from page 178 is modified so that it works with the IR detectors.

##### Converting the Whiskers Program for IR Object Detection/Avoidance

This next example program started as RoamingWithWhiskers.bs2. Aside from adjusting the name and description, two bit variables were added to store the states of the IR detectors.

```
irDetectLeft VAR Bit
irDetectRight VAR Bit
```

A routine was also added to read the IR pairs.

```
FREQOUT 8, 1, 38500
```

```
irDetectLeft = IN9
```

The **IF...THEN** statements were modified so that they look at the variables that store the IR pair detections instead of the whisker inputs.

```
IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
 GOSUB Back_Up
 GOSUB Turn_Left
 GOSUB Turn_Left
ELSEIF (irDetectLeft = 0) THEN
 GOSUB Back_Up
 GOSUB Turn_Right
ELSEIF (irDetectRight = 0) THEN
 GOSUB Back_Up
 GOSUB Turn_Left
ELSE
 GOSUB Forward_Pulse
ENDIF
```

### Example Program – RoamingWithIr.bs2

- ✓ Open RoamingWithWhiskers.bs2
- ✓ Modify it so that it matches the program below.
- ✓ Reconnect power to your board and servos.
- ✓ Save and run it.
- ✓ Verify that, aside from the fact that there's no contact required, it behaves like RoamingWithWhiskers.bs2.

```
' -----[Title]-----
' Robotics with the Boe-Bot - RoamingWithIr.bs2
' Adapt RoamingWithWhiskers.bs2 for use with IR pairs.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

DEBUG "Program Running!"

' -----[Variables]-----

irDetectLeft VAR Bit
irDetectRight VAR Bit
pulseCount VAR Byte

' -----[Initialization]-----
```

```

FREQOUT 4, 2000, 3000 ' Signal program start/reset.

' -----[Main Routine]-----
DO

 FREQOUT 8, 1, 38500 ' Store IR detection values in
 irDetectLeft = IN9 ' bit variables.

 FREQOUT 2, 1, 38500
 irDetectRight = IN0

 IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
 GOSUB Back_Up ' Both IR pairs detect obstacle
 GOSUB Turn_Left ' Back up & U-turn (left twice)
 GOSUB Turn_Left
 ELSEIF (irDetectLeft = 0) THEN ' Left IR pair detects
 GOSUB Back_Up ' Back up & turn right
 GOSUB Turn_Right
 ELSEIF (irDetectRight = 0) THEN ' Right IR pair detects
 GOSUB Back_Up ' Back up & turn left
 GOSUB Turn_Left
 ELSE
 GOSUB Forward_Pulse ' Both IR pairs 1, no detects
 ' Apply a forward pulse
 ' and check again
 ENDIF

LOOP

' -----[Subroutines]-----

Forward_Pulse: ' Send a single forward pulse.
 PULSOUT 13,850
 PULSOUT 12,650
 PAUSE 20
 RETURN

Turn_Left: ' Left turn, about 90-degrees.
 FOR pulseCount = 0 TO 20
 PULSOUT 13, 650
 PULSOUT 12, 650
 PAUSE 20
 NEXT
 RETURN

Turn_Right: ' Right turn, about 90-degrees.
 FOR pulseCount = 0 TO 20
 PULSOUT 13, 850
 PULSOUT 12, 850
 PAUSE 20
 NEXT
 RETURN

```

```

Back_Up: ' Back up.
 FOR pulseCount = 0 TO 40
 PULSOUT 13, 650
 PULSOUT 12, 850
 PAUSE 20
 NEXT
 RETURN

```

### **Your Turn**

- ✓ Modify RoamingWithIr.bs2 so that the IR pairs are checked in a subroutine.

## **ACTIVITY #5: HIGH PERFORMANCE IR NAVIGATION**

The style of pre-programmed maneuvers that were used in the previous activity were fine for whiskers, but are unnecessarily slow when using the IR LEDs and detectors. You can greatly improve the Boe-Bot's roaming performance by checking for obstacles before delivering each set of pulses to the servos. The program can use the sensor inputs to select the best maneuver for each moment of navigation. That way, the Boe-Bot never turns further than it has to, and it can neatly find its way around obstacles and successfully navigate more complex courses.

### **Sampling Between Every Pulse to Avoid Collisions**

The great thing about detecting an obstacle before bumping into it is that it gives the Boe-Bot some room to navigate around it. The Boe-Bot can apply a pulse to turn away from an object, check again and if the object is still there, apply another pulse to avoid it. The Boe-Bot can keep applying pulses and checking, until it steers clear of the obstacle. Then, it can resume forward pulses. After experimenting with this next example program, you'll likely agree that it's a much better way for the Boe-Bot to roam.

### **Example Program – FastIrRoaming.bs2**

- ✓ Enter, save, and run FastIrRoaming.bs2.

```

' Robotics with the Boe-Bot - FastIrRoaming.bs2
' Higher performance IR object detection assisted navigation

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

irDetectLeft VAR Bit ' Variable Declarations

```

```

irDetectRight VAR Bit
pulseLeft VAR Word
pulseRight VAR Word

FREQOUT 4, 2000, 3000 ' Signal program start/reset.

DO ' Main Routine

 FREQOUT 8, 1, 38500 ' Check IR Detectors
 irDetectLeft = IN9
 FREQOUT 2, 1, 38500
 irDetectRight = IN0

 ' Decide how to navigate.
 IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
 pulseLeft = 650
 pulseRight = 850
 ELSEIF (irDetectLeft = 0) THEN
 pulseLeft = 850
 pulseRight = 850
 ELSEIF (irDetectRight = 0) THEN
 pulseLeft = 650
 pulseRight = 650
 ELSE
 pulseLeft = 850
 pulseRight = 650
 ENDIF

 PULSOUT 13,pulseLeft ' Apply the pulse.
 PULSOUT 12,pulseRight
 PAUSE 15

LOOP ' Repeat main routine

```

7

### How FastIrRoaming.bs2 Works

This program takes a slightly different approach to applying pulses. Aside from the two bits used to store the IR detector outputs, it uses two word variables to set the pulse durations delivered by the **PULSOUT** command.

```

irDetectLeft VAR Bit
irDetectRight VAR Bit
pulseLeft VAR Word
pulseRight VAR Word

```

Inside the **DO...LOOP**, the **FREQOUT** commands are used to send a 38.5 kHz IR signal to each IR LED. Immediately after the 1 ms burst of IR is sent, a bit variable stores the output state of the IR detector. This is necessary, because if you wait any longer than a

command's worth of time, the IR detector will return to the not detected (1 state), regardless of whether or not it detected an object.

```
FREQOUT 8, 1, 38500
irDetectLeft = IN9
FREQOUT 2, 1, 38500
irDetectRight = IN0
```

In the **IF...THEN** statements, instead of delivering pulses or calling navigation routines, this program sets variable values that will be used in **PULSOUT** commands' *Duration* arguments.

```
IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
 pulseLeft = 650
 pulseRight = 850
ELSEIF (irDetectLeft = 0) THEN
 pulseLeft = 850
 pulseRight = 850
ELSEIF (irDetectRight = 0) THEN
 pulseLeft = 650
 pulseRight = 650
ELSE
 pulseLeft = 850
 pulseRight = 650
ENDIF
```

Before the **DO...LOOP** repeats, the last thing to do is to deliver pulses to the servos. Notice that the **PAUSE** command is no longer 20. Instead, it's 15 since roughly 5 ms is taken checking the IR LEDs.

```
PULSOUT 13,pulseLeft ' Apply the pulse.
PULSOUT 12,pulseRight
PAUSE 15
```

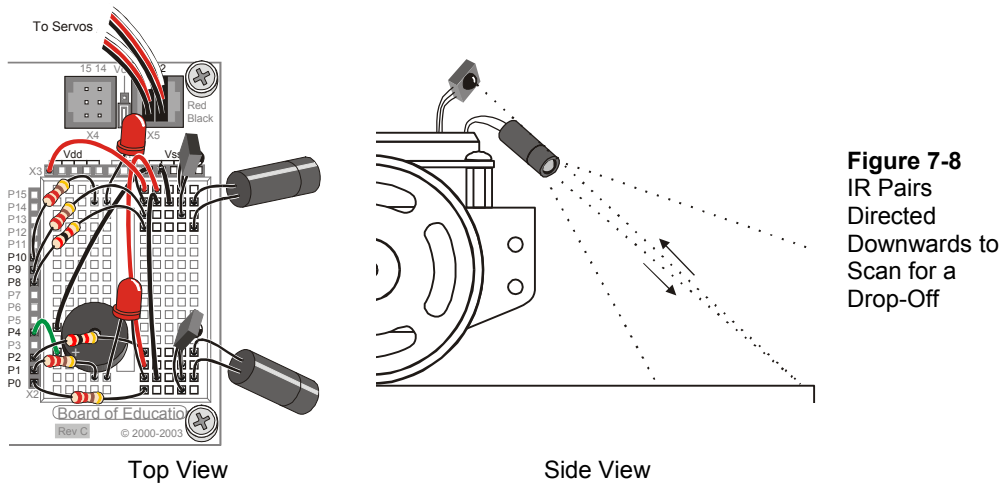
### **Your Turn**

- ✓ Save FastIrRoaming.bs2 as FastIrRoamingYourTurn.bs2.
- ✓ Use the LEDs to broadcast that the Boe-Bot has detected an object.
- ✓ Try modifying the values that pulseLeft and pulseRight are set to so that the Boe-Bot does everything at half speed.

## ACTIVITY #6: THE DROP-OFF DETECTOR

Up until now, the Boe-Bot has mainly been programmed to take evasive maneuvers when an object is detected. There are also applications where the Boe-Bot must take evasive action when an object is not detected. For example, if the Boe-Bot is roaming on a table, its IR detectors might be looking down at the table surface as shown in Figure 7-8. The program should make it continue forward so long as both IR detectors can “see” the surface of the table. In other words, the the Boe-Bot can continue forward so long as the table top it’s navigating on is detected.

- ✓ Disconnect power from your board and servos.
- ✓ Point your IR pairs downward and outward as shown in Figure 7-8.



### Recommended Materials:

- (1) Roll of black vinyl electrical tape –  $\frac{3}{4}$ " (19 mm) wide.
- (1) Sheet of white poster board – 22 x 28 in (56 x 71 cm).

### Simulating a Drop-Off with Electrical Tape

A sheet of white poster board with a border made of electrical tape makes for a handy way to simulate the drop-off presented by a table edge, with much less risk to your Boe-Bot.

- ✓ Build a course similar to the electrical tape delimited course shown in Figure 7-9. Use at least three strips of electrical tape, edge to edge with no paper visible between the strips.
- ✓ Replace your 1 k $\Omega$  resistors with 2 k $\Omega$  resistors (red-black-red) to connect P2 to its IR LED and P8 to its IR LED. We want the Boe-Bot to be nearsighted for this activity.
- ✓ Reconnect power to your board.
- ✓ Run the program IrInterferenceSniffer.bs2 (page 246) to make sure that nearby fluorescent lighting will not interfere with your Boe-Bot's IR detectors.
- ✓ Use the TestIrPairsAndIndicators.bs2 (page 244) to make sure that the Boe-Bot detects the poster board but does not detect the electrical tape.



If the Boe-Bot still "sees" the electrical tape too clearly, here are a few remedies:

- ✓ Try adjusting the IR detectors and LEDs downward at various angles.
- ✓ Try a different brand of vinyl electrical tape.
- ✓ Try replacing the 2 k $\Omega$  resistors with 4.7 k $\Omega$  (yellow-violet-red) resistors to make the Boe-Bot more nearsighted.
- ✓ Adjust the **FREQOUT** command with different **Freq1** arguments. Here are some arguments that will make the Boe-Bot more nearsighted: 38250, 39500, 40500

If you are using older IR LEDs, the Boe-Bot might actually be having problems with being **too nearsighted**. Here are some remedies that will increase the Boe-Bot's sensitivity to objects and make it more far sighted:

- ✓ Try 1 k $\Omega$  (brown-black-red) or 470  $\Omega$  (yellow-violet-brown) or even 220  $\Omega$  (red-red-brown) resistors in series with the IR LEDs instead of 2 k $\Omega$ .



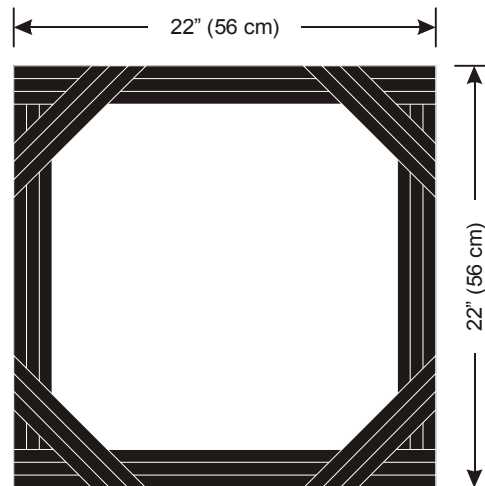


**Figure 7-9**  
Electrical Tape  
Outline  
Simulates  
Tabletop Edge

**If you try a tabletop after success with the electrical tape course:**



- ✓ Remember to follow the same steps you followed before running the Boe-Bot in the electrical tape delimited course!

Make sure to be the spotter for your Boe-Bot. Be ready as your Boe-Bot roams the tabletop:

- ✓ Always be ready to pick your Boe-Bot up from above as it approaches the edge of the table it's navigating. If the Boe-Bot tries to drive off the edge, pick it up before it takes the plunge. Otherwise, your Boe-Bot might become a Not-Bot!
- ✓ Your Boe-Bot may detect you if you are standing in its line of sight. Its current program has no way to differentiate you from the table below it, so it might try to continue forward and off the edge of the table. So, stay out of its detector's line of sight as you spot.

### **Programming for Drop-Off Detection**

For the most part, programming your Boe-Bot to navigate around a table top without going over the edge is a matter of adjusting the **IF...THEN** statements from `FastIrNavigation.bs2`. The main adjustment is that the servos should be directed to make the Boe-Bot roll forward when `irDetectLeft` and `irDetectRight` are both 0, indicating that an object (the table's surface) has been detected. The Boe-Bot also has to turn away from a detector that indicates it has not detected an object. For example, if `irDetectLeft` is 1, the Boe-Bot had better turn right.

A second feature of a program for turning away from drop-offs is adjustable distance. You may want your Boe-Bot to only take one pulse forward between checking the detectors, but as soon as a drop-off is detected, you may want your Boe-Bot to take several pulses worth of turn before checking the detectors again.

Just because you are taking multiple pulses in an evasive maneuver, it doesn't mean you have to return to whiskers-style navigation. Instead, you can add a `pulseCount` variable that you can use to set to the number of pulses to deliver for a maneuver. The `PULSOUT` command can be placed inside a `FOR...NEXT` loop that executes `FOR 1 TO pulseCount` pulses. For one pulse forward, `pulseCount` can be 1, for ten pulses left, `pulseCount` can be set to 10, and so on.

### Example Program – AvoidTableEdge.bs2

- ✓ Open FastIrNavigation.bs2 and save it as AvoidTableEdge.bs2.
- ✓ Modify the program so that it matches the example program. This will involve adding variables, modifying the `IF...THEN` statements, and nesting the `PULSOUT` commands inside a `FOR...NEXT` loop. Be careful to make sure that all the `pulseLeft` and `pulseRight` variable values inside the `IF...THEN` statement are properly adjusted. Their values are different from the ones in FastIrNavigation.bs2 because the rules of the course are different.
- ✓ Reconnect your board and servos.
- ✓ Test the program on your electrical tape delimited course.
- ✓ If you decide to try a tabletop, remember to follow the testing and spotting tips discussed earlier.

```
' Robotics with the Boe-Bot - AvoidTableEdge.bs2
' IR detects object edge and navigates to avoid drop-off.

' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Program Running!"

irDetectLeft VAR Bit ' Variable declarations.
irDetectRight VAR Bit
pulseLeft VAR Word
pulseRight VAR Word
loopCount VAR Byte
pulseCount VAR Byte
```

```

FREQOUT 4, 2000, 3000 ' Signal program start/reset.

DO ' Main Routine.

 FREQOUT 8, 1, 38500 ' Check IR detectors.
 irDetectLeft = IN9
 FREQOUT 2, 1, 38500
 irDetectRight = IN0

 ' Decide navigation.

 IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
 pulseCount = 1 ' Both detected,
 pulseLeft = 850 ' one pulse forward.
 pulseRight = 650
 ELSEIF (irDetectRight = 1) THEN
 pulseCount = 10 ' Right not detected,
 pulseLeft = 650 ' 10 pulses left.
 pulseRight = 650
 ELSEIF (irDetectLeft = 1) THEN
 pulseCount = 10 ' Left not detected,
 pulseLeft = 850 ' 10 pulses right.
 pulseRight = 850
 ELSE
 pulseCount = 15 ' Neither detected,
 pulseLeft = 650 ' back up and try again.
 pulseRight = 850
 ENDIF

 FOR loopCount = 1 TO pulseCount ' Send pulseCount pulses
 PULSOUT 13,pulseLeft
 PULSOUT 12,pulseRight
 PAUSE 20
 NEXT

LOOP

```

### How AvoidTableEdge.bs2 Works



Since this program is a modified version of FastIrRoaming.bs2, only changes to the program are discussed here.

A **FOR...NEXT** loop is added to the program to control how many pulses are delivered each time through the main (**DO...LOOP**) routine. Two variables are added, **loopCount** functions as an index for a **FOR...NEXT** loop and **pulseCount** is used as the *EndValue* argument.

```

loopCount VAR Byte

```

```
pulseCount VAR Byte
```

The **IF...THEN** statements now set the value of **pulseCount** as well as the values of **pulseRight** and **pulseLeft**. If both detectors can see the table, take one cautious pulse forward.

```
IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
 pulseCount = 1
 pulseLeft = 850
 pulseRight = 650
```

Else, if the right IR detector does not see the tabletop, rotate left 10 pulses.

```
ELSEIF (irDetectRight = 1) THEN
 pulseCount = 10
 pulseLeft = 650
 pulseRight = 650
```

Else, if the left IR detector does not see the tabletop, rotate right 10 pulses.

```
ELSEIF (irDetectLeft = 1) THEN
 pulseCount = 10
 pulseLeft = 850
 pulseRight = 850
```

Else, if neither detector can see the table top, back up 15 pulses and try again, hoping that one of the detectors will see the drop-off before the other.

```
ELSE
 pulseCount = 15
 pulseLeft = 650
 pulseRight = 850
ENDIF
```

Now that the value of **pulseCount**, **pulseLeft**, and **pulseRight** are set, this **FOR...NEXT** loop delivers the specified number of pulses for the maneuver determined by the **pulseLeft** and **pulseRight** variable.

```
FOR loopCount = 1 TO pulseCount
 PULSOUT 13,pulseLeft
 PULSOUT 12,pulseRight
 PAUSE 20
```

NEXT

**Your Turn**

You can experiment with setting different `pulseLeft`, `pulseRight`, and `pulseCount` values inside the **IF...THEN** statement. For example, if the Boe-Bot doesn't turn as far, it may actually track the edge of the electrical tape delimited course. Pivoting backward instead of rotating in place may also lead to some interesting behaviors.

- ✓ Modify `AvoidTableEdge.bs2` so that it follows the edge of the electrical tape delimited course by adjusting the `pulseCount` values so that the Boe-Bot doesn't turn too far away from the edge.
- ✓ Experiment with pivoting as a way to make the Boe-Bot roam inside the perimeter instead of following the edge.

## SUMMARY

This chapter covered a unique technique for infrared object detection that uses the infrared LED found in common handheld remotes, and the infrared detector found in TVs, CD/DVD players, and other appliances that are controlled by these remotes. By shining infrared into the Boe-Bot's path and looking for its reflection, object detection can be accomplished without physically contacting the object. Infrared LED circuits are used to send a 38.5 kHz signal with the help of a property of the **FREQOUT** command called a harmonic, which is inherent to digitally synthesized signals.

An infrared detection indicator program was introduced for remote (not connected to the PC) testing of the IR LED/detector pairs. An infrared interference sniffer program was also introduced to help detect interference that can be generated by some fluorescent light fixtures. Since the signals sent by the IR detectors are so similar to the signals sent by the whiskers, *RoamingWithWhiskers.bs2* was adapted to the infrared detectors. A program that checks the IR detectors between each servo pulse was introduced to demonstrate a higher performance way of roaming without colliding into objects. This program was then modified to avoid the edge of an electrical tape delimited area. Since electrical tape absorbs infrared, framing a large sheet of construction paper emulates the drop-off that is seen at a table edge without the danger to the actual Boe-Bot.

## Questions

1. What is the frequency of the harmonic sent by **FREQOUT 2, 1, 38500**? What is the value of the fundamental frequency sent by that command? How long are these signals sent for? What I/O pin does the IR LED circuit have to be connected to in order to broadcast this signal?
2. What command has to immediately follow the **FREQOUT** command in order to accurately determine whether or not an object has been detected?
3. What does it mean if the IR detector sends a low signal? What does it mean when the detector sends a high signal?
4. What happens if you change the value of a resistor in series with a red LED? What happens if you change the value of a resistor in series with an infrared LED?

**Exercises**

1. Modify a line of code in `IrInterferenceSniffer.bs2` so that it only monitors one of the IR LED/detector pairs.
2. Explain the function of `pulseCount` in `AvoidTableEdge.bs2`. How does this relate to your answer to Exercise 3?

**Projects**

1. Design a Boe-Bot application that sits still until you wave your hand in front of it, then it starts roaming.
2. Design a Boe-Bot application that slowly rotates in place until it detects an object. As soon as it detects an object, it locks onto and chases the object. This is a classic SumoBot behavior.
3. Design a Boe-Bot application that roams, but if it detects infrared interference, it sounds the alarm briefly, then continues roaming. This alarm should be different from the low battery alarm.

## Solutions

- Q1. 38.5 kHz is the frequency of the harmonic. Its fundamental frequency =  $65536 - 38500 = 27036$  Hz. The signals are sent for 1 millisecond, and the IR LED must be connected to I/O Pin 2.
- Q2. The command which stores the detector's output in a variable. For example, `irDetectLeft = IN9`.
- Q3. A low signal means IR at 38.5 kHz was detected, thus, an object was detected. A high signal means no IR at 38.5kHz was detected, so, no object.
- Q4. Electrically speaking, for both red and infrared LEDs, a smaller resistor will cause the LED to glow more brightly. A bigger resistor results in dimmer LEDs. In terms of results, brighter IR LEDs make it possible to detect objects that are farther away.
- E1. Change the **IF...THEN** to read:  
`IF (IN0 = 0) THEN`  
 This will only monitor the right detector.
- E2. The `pulseCount` variable allows the Boe-Bot to have adjustable distance of motion depending on the situation.
- P1. The `FastIrRoaming.bs2` program can be combined with a **DO...UNTIL** loop that does nothing until it detects an object. A sample solution is shown below.

```
' -----[Title]-----
' Robotics with the Boe-Bot - MotionActivatedBoeBot.bs2
' Boe-Bot starts roaming when hand is waved in front of IR detectors.

' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[Variables]-----

irDetectLeft VAR Bit ' Variable Declarations
irDetectRight VAR Bit
pulseLeft VAR Word
pulseRight VAR Word

' -----[Initialization]-----

DEBUG "Program Running!"
FREQOUT 4, 2000, 3000 ' Signal program
start/reset.
```



```

' -----[Main Routine]-----
Main:
' Loop until something is detected
DO
 GOSUB Check_IRs
 LOOP UNTIL (irDetectLeft = 0) OR (irDetectRight = 0)
' Now start roaming -- this code from FastIrRoaming.bs2
DO
 IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
 pulseLeft = 650 ' Both detect
 pulseRight = 850 ' Back up
 ELSEIF (irDetectLeft = 0) THEN
 pulseLeft = 850 ' Left detect
 pulseRight = 850 ' Turn right
 ELSEIF (irDetectRight = 0) THEN
 pulseLeft = 650 ' Right detect
 pulseRight = 650 ' Turn left
 ELSE
 pulseLeft = 850 ' Nothing detected
 pulseRight = 650 ' Go forward
 ENDIF

 PULSOUT 13, pulseLeft ' Apply the pulse.
 PULSOUT 12, pulseRight
 PAUSE 15

 GOSUB Check_IRs ' Check IRs again
LOOP

' -----[Subroutines] -----
Check_IRs:
 FREQOUT 8, 1, 38500 ' Check IR Detectors
 irDetectLeft = IN9
 FREQOUT 2, 1, 38500
 IrDetectRight = IN0
 RETURN

```

- P2. This behavior is in many ways the opposite of the roaming behavior covered in this chapter. Instead of avoiding objects, the Boe-Bot tries to go toward the objects. For this reason, the main code can be derived from "FastIrRoaming.bs2", with a bit added that spins the Boe-Bot slowly until an object is detected.

In the solution below, once the Boe-Bot has spied an object, it will continue forward even if the detectors both read "no object" (1) for a few loops. This is because, as the Boe-Bot is maneuvering toward the object, sometimes the



```

ENDIF ' its position.

PULSOUT 13,pulseLeft ' Apply the pulse.
PULSOUT 12,pulseRight
PAUSE 15 ' 5 ms for detectors

' Check IRs again in case object is moving
GOSUB Check_IRs
LOOP

' -----[Subroutines] -----

Check_IRs:
 FREQOUT 8, 1, 38500 ' Check IR Detectors
 irDetectLeft = IN9
 FREQOUT 2, 1, 38500
 IrDetectRight = IN0
 RETURN

```

P3. The key to solving this problem is to combine "FastIrRoaming.bs2" and "IrInterferenceSniffer.bs2" in a single program.

```

' -----[Title]-----
' Robotics with the Boe-Bot - RoamAndSniffBoeBot.bs2
' Boe-Bot roams around and sounds alarm when IR detected.

' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[Variables]-----

irDetectLeft VAR Bit ' Left IR sensor reading
irDetectRight VAR Bit ' Right IR sensor reading
pulseLeft VAR Word ' Pulses sent to servos
pulseRight VAR Word
counter VAR Nib ' Loop counter

' -----[Initialization]-----

DEBUG "Program Running!"
FREQOUT 4, 2000, 3000 ' Signal program
start/reset.

' -----[Main Routine]-----

Main:
DO
 GOSUB Roam
 GOSUB Sniff
LOOP

```

```

' -----[Subroutines] -----
Sniff: ' From IrInterferenceSniffer.bs2
 IF (IN0 = 0) OR (IN9 = 0) THEN
 FOR counter = 1 TO 5 ' Beep 5 times
 HIGH 1 ' and flash LEDs
 HIGH 10
 FREQOUT 4, 50, 4000
 LOW 1
 LOW 10
 PAUSE 20
 NEXT
 ENDIF
 RETURN

Roam: ' From FastIrRoaming.bs2
 FREQOUT 8, 1, 38500 ' Check IR Detectors
 irDetectLeft = IN9
 FREQOUT 2, 1, 38500
 irDetectRight = IN0

 ' Decide how to navigate.
 IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
 pulseLeft = 650
 pulseRight = 850
 ELSEIF (irDetectLeft = 0) THEN
 pulseLeft = 850
 pulseRight = 850
 ELSEIF (irDetectRight = 0) THEN
 pulseLeft = 650
 pulseRight = 650
 ELSE
 pulseLeft = 850
 pulseRight = 650
 ENDIF

 PULSOUT 13,pulseLeft ' Apply the pulse.
 PULSOUT 12,pulseRight
 PAUSE 15
 RETURN

```

## Chapter 8: Robot Control with Distance Detection

In Chapter 7, we used the infrared sensors to detect whether an object is in the Boe-Bot's way without actually touching it. Wouldn't it be nice to also know how far away the object is? This is usually a task for sonar, which sends a pulse of sound out and records how long it takes for the echo to come back. The time it takes for the echo to come back can then be used to calculate how far away the object is. There is, however, a way to accomplish distance detection with the very same circuit you used in the previous chapter. With your Boe-Bot able to determine the distance of an object, it can be programmed to follow a moving object without colliding into it. The Boe-Bot can also be programmed to follow black tracks on a white background.

### DETERMINING DISTANCE WITH THE SAME IR LED/DETECTOR CIRCUIT

You will use the same circuit from the previous chapter to detect distance.

- ✓ If the circuit is still built on your Boe-Bot, make sure your IR LED's have 1 k $\Omega$  resistors in series.
- ✓ If you already disassembled the circuit from the previous chapter, repeat the steps in Chapter 7, Activity #1 on page 237.

8

#### Recommended Equipment and Materials:

- (1) Ruler
- (1) Sheet of paper

### ACTIVITY #1: TESTING THE FREQUENCY SWEEP

Figure 8-1 shows an excerpt from one specific brand of IR detector's datasheet (Panasonic PNA4602M). This excerpt is a graph that shows how much less sensitive this IR detector becomes if the IR signal it receives flashes on/off at a frequency other than 38.5 kHz. For example, if you send it IR flashed on/off at 40 kHz, it's only 50% as sensitive as it would be at 38.5 kHz. If the IR is flashed on/off at 42 kHz, the detector is only 20% as sensitive. Especially for frequencies that make the detector less sensitive, the object has to be closer to make the reflected IR brighter for the detector to detect it.

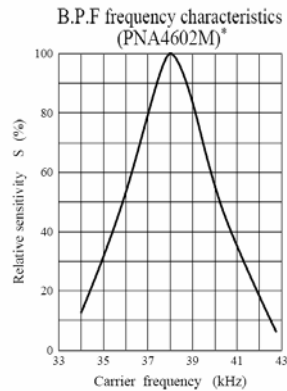


**Figure 8-1**  
Filter Sensitivity  
Depends on  
Carrier Frequency

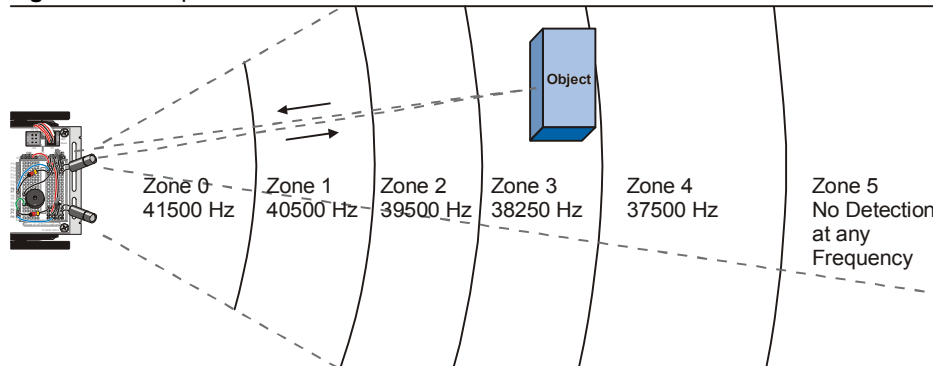
Another way to think about it is that the most sensitive frequency will detect the objects that are the farthest away, while less sensitive frequencies can only be used to detect closer objects. This makes distance detection simple. Pick 5 frequencies, then test them from most sensitive to least sensitive. Try at the most sensitive frequency first. If an object is detected, check and see if the next most sensitive frequency detects it. Depending on which frequency makes the reflected infrared no longer visible to the IR detector, you can infer the distance.



**Frequency Sweep** is the technique of testing a circuit's output using a variety of input frequencies.

### **Programming Frequency Sweep for Distance Detection**

**Figure 8-2** shows an example of how the Boe-Bot can test for distance using frequency. In this example, the object is in Zone 3. That means that the object can be detected when 37500 and 38250 Hz is transmitted, but it cannot be detected with 39500, 40500, and 41500 Hz. If you were to move the object into Zone 2, then the object can be detected when 37500, 38250, and 39500 Hz are transmitted, but not when 40500 and 41500 Hz are transmitted.

**Figure 8-2: Frequencies and Zones for the Boe-Bot**

**You might be wondering why the value of zone 4 is 37.5 kHz and not 38.5 kHz.** The reason they are not the values that you would expect based on the % sensitivity graph is because the **FREQOUT** command transmits a slightly more powerful (harmonic) signal at 37.5 kHz than it does at 38.5 kHz. The frequencies listed in Figure 8-2 are frequencies you will program the BASIC Stamp to use to determine the distance of an object. These frequencies were determined using tests similar to the ones outlined in Appendix G: Tuning IR Distance Detection.

8

In order to test the IR detector at each frequency, you will need to use **FREQOUT** to send five different frequencies and test at each frequency to find out whether the IR detector could see the object. The steps between each frequency are not quite even enough to use the **FOR...NEXT** loop's **STEP** operator. You could use **DATA** and **READ**, but that would be cumbersome. You could use five different **FREQOUT** commands, but that would be a waste of code space. Instead, the best approach for storing a short list of values that you want to use in sequence is a command called **LOOKUP**. The syntax for the **LOOKUP** command is:

**LOOKUP** *Index*, [*Value0*, *Value1*, ...*ValueN*], *Variable*

If the *Index* argument is 0, *value0* from the list inside the square braces will be placed in *variable*. If *Index* is 1, *value1* from the list will be placed in *variable*. There could be up to 256 values in the list, but for the next example program, we will only need 5. Here is how it will be used:

```
FOR freqSelect = 0 TO 4
```

```
 LOOKUP freqSelect, [37500, 38250, 39500, 40500, 41500], irFrequency
```

```

FREQOUT 8,1, irFrequency
irDetect = IN9
' Commands not shown...

```

```

NEXT

```

The first time through the **FOR...NEXT** loop, **freqSelect** is 0, so the **LOOKUP** command places the value 37500 in the **irFrequency** variable. Since **irFrequency** contains 37500 after the **LOOKUP** command, the **FREQOUT** command sends that frequency to the IR LED connected to P8. As in the previous chapter, the value of **IN9** is then saved in the **irDetect** variable. The second time through the **FOR...NEXT** loop, the value of **freqSelect** is now 1, which means the **LOOKUP** command places 38250 into the **irFrequency** variable, and the process is repeated for this higher frequency. The third time through, it's repeated again with 39500, and so on. The result is remarkable, especially considering you are using parts that were designed for a completely different purpose, to make IR communication between a handheld remote and a television possible.

### Example Program – TestLeftFrequencySweep.bs2

TestLeftFrequencySweep.bs2 does two things. First, it tests the left IR LED/detector pair (connected to P8 and P9) to make sure they are functioning properly for distance detection. However, it also demonstrates how the frequency sweep illustrated in Figure 8-2 is accomplished.

When you run the program, the Debug Terminal will display your zone measurement. There are many possible yes-no patterns that can be generated; two are shown in Figure 8-3. The test patterns will vary depending on the characteristics of the filter inside the IR detector.

The program determines which zone the detected object is in by counting the number of “No” occurrences. Notice that even though the two Debug Terminal test patterns in Figure 8-3 are different, they both have three “Yes” and two “No” occurrences. Therefore, “Zone 2” is the location of the object detected in both examples.



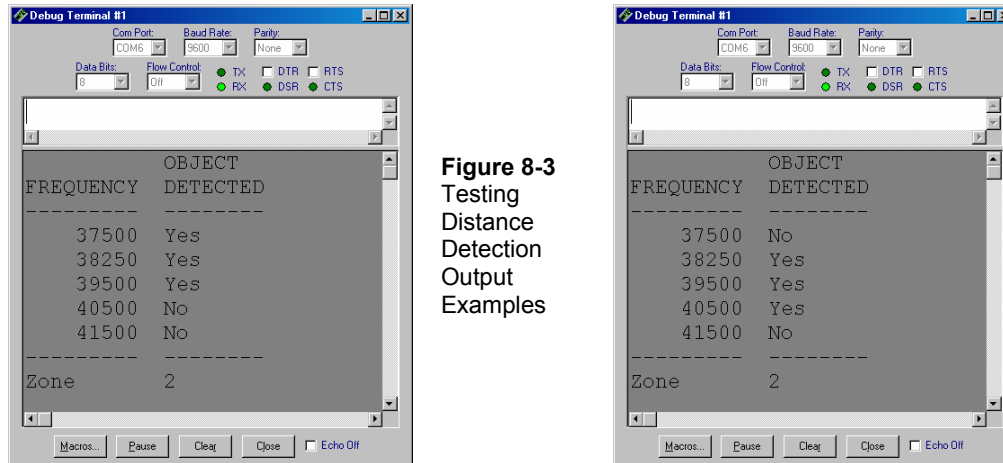


**Figure 8-3**  
Testing  
Distance  
Detection  
Output  
Examples



**Keep in mind that these distance measurements are relative and not necessarily precise or evenly spaced.** However, they will give the Boe-Bot a good enough sense of object distance for following, tracking, and other activities.

8

- ✓ Enter, save, and run TestLeftFrequencySweep.bs2.
- ✓ Use a sheet of paper or card facing the IR LED/detector to test the distance detection.
- ✓ Start with the sheet very close to the IR LED, perhaps ¼ in (or 1 cm) away from the IR LED. Your Zone in the Debug Terminal should either be 0 or 1.
- ✓ Gradually move the sheet of paper away from the IR LED and make a note of each distance that causes the zone value to get larger.



**Zones 1-4** typically fall in the range of 6 to 12 in (15 to 30 cm) for the shielded LEDs with a 1 kΩ resistor. Older shrink wrap LED distances will be less. As long as objects can be detected up to 4 in (10 cm) away, the experiments in this chapter will work. If the distance detector range is less than that, which is likely if you have shrink wrapped IR LEDs, try reducing your series resistance from 1 kΩ to 470 Ω or 220 Ω.

```
' -----[Title]-----
' Robotics with the Boe-Bot - TestLeftFrequencySweep.bs2
' Test IR detector distance responses to frequency sweep.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.
```

```

' -----[Variables]-----
freqSelect VAR Nib
irFrequency VAR Word
irDetect VAR Bit
distance VAR Nib

' -----[Initialization]-----
DEBUG CLS,
 " OBJECT", CR,
 "FREQUENCY DETECTED", CR,
 "-----"

' -----[Main Routine]-----
DO
 distance = 0

 FOR freqSelect = 0 TO 4

 LOOKUP freqSelect,[37500,38250,39500,40500,41500], irFrequency
 FREQOUT 8,1, irFrequency
 irDetect = IN9
 distance = distance + irDetect

 DEBUG CRSRXY, 4, (freqSelect + 3), DEC5 irFrequency
 DEBUG CRSRXY, 11, freqSelect + 3

 IF (irDetect = 0) THEN DEBUG "Yes" ELSE DEBUG "No "

 PAUSE 100

 NEXT

 DEBUG CR,
 "-----", CR,
 "Zone ", DEC1 distance
LOOP

```

### Your Turn – Testing the Right IR LED/Detector Pair

Although there's some labeling involved, you can modify this program to test the right IR LED and detector by changing these two lines:

```

 FREQOUT 8,1, irFrequency
 irDetect = IN9

```

so that they read

```
FREQOUT 2,1, irFrequency
irDetect = IN0
```

- ✓ Modify TestLeftFrequencySweep.bs2 for testing the distance measurement of the right IR LED/detector pair.
- ✓ Run the program and verify that this pair can measure a similar distance.

### **Displaying Both Distances**

It's useful at times to have a quick program you can run to test both the Boe-Bot's distance detectors at the same time. This program is organized into subroutines, which can be handy for copying and pasting into other programs that require distance detection.

#### **Example Program – DisplayBothDistances.bs2**

- ✓ Enter, save, and run DisplayBothDistances.bs2.
- ✓ Repeat the distance measurement exercise with a sheet of paper on each LED, then on both LEDs at the same time.

8

```
' -----[Title]-----
' Robotics with the Boe-Bot - DisplayBothDistances.bs2
' Test IR detector distance responses of both IR LED/detector pairs to
' frequency sweep.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

' -----[Variables]-----

freqSelect VAR Nib
irFrequency VAR Word
irDetectLeft VAR Bit
irDetectRight VAR Bit
distanceLeft VAR Nib
distanceRight VAR Nib

' -----[Initialization]-----

DEBUG CLS,
 "IR OBJECT ZONE", CR,
 "Left Right", CR,
 "-----"

' -----[Main Routine]-----
```

```

DO

 GOSUB Get_Distances
 GOSUB Display_Distances

LOOP

' -----[Subroutine - Get_Distances]-----
Get_Distances:

 distanceLeft = 0
 distanceRight = 0

 FOR freqSelect = 0 TO 4

 LOOKUP freqSelect, [37500,38250,39500,40500,41500], irFrequency

 FREQOUT 8,1,irFrequency
 irDetectLeft = IN9
 distanceLeft = distanceLeft + irDetectLeft

 FREQOUT 2,1,irFrequency
 irDetectRight = IN0
 distanceRight = distanceRight + irDetectRight

 PAUSE 100

 NEXT

 RETURN

' -----[Subroutine - Display_Distances]-----
Display_Distances:

 DEBUG CRSRXY,2,3, DEC1 distanceLeft,
 CRSRXY,9,3, DEC1 distanceRight
 RETURN

```

### Your Turn – More Distance Tests

- √ Try measuring the distance of different objects and find out if the color and/or texture make any difference to the distance measurement.

## ACTIVITY #2: BOE-BOT SHADOW VEHICLE

For one Boe-Bot to follow another, the Boe-Bot that follows, a.k.a. the shadow vehicle, has to know how far ahead the lead vehicle is. If the shadow vehicle is lagging behind, it has to detect this and speed up. If the shadow vehicle is too close to the lead vehicle, it has to detect this as well and slow down. If it's the right distance, it can wait until the measurements indicate it's too far or too close again.

Distance is just one kind of value that robots and other automated machinery are responsible for. When a machine is designed to automatically maintain a value, such as distance, pressure, or fluid level, it generally involves a control system. These systems sometimes consist of sensors and valves, or sensors and motors, or, in the case of the Boe-Bot, sensors and continuous rotation servos. There is also some kind of processor that takes the sensor measurements and converts them to mechanical action. The processor has to be programmed to make decisions based on the sensor inputs, and then control the mechanical outputs accordingly. In the case of the Boe-Bot, the processor is the BASIC Stamp 2.

Closed loop control is a common method of maintaining levels, and it works very well for helping the Boe-Bot maintain its distance from an object. There are lots of different kinds of closed loop control. Some of the most common are hysteresis, proportional, integral, and derivative control. All of these types of control are introduced in detail in the Stamps in Class text *Process Control*, listed in the Preface.

Most control techniques can be implemented with just a few lines of code in PBASIC. In fact, the majority of the proportional control loop shown in Figure 8-4 reduces to just one line of PBASIC code. This diagram is called a block diagram, and it describes the steps of the proportional control process that the Boe-Bot will use to measure distance with its right IR LED and detector and adjust position to maintain distance with its right servo.

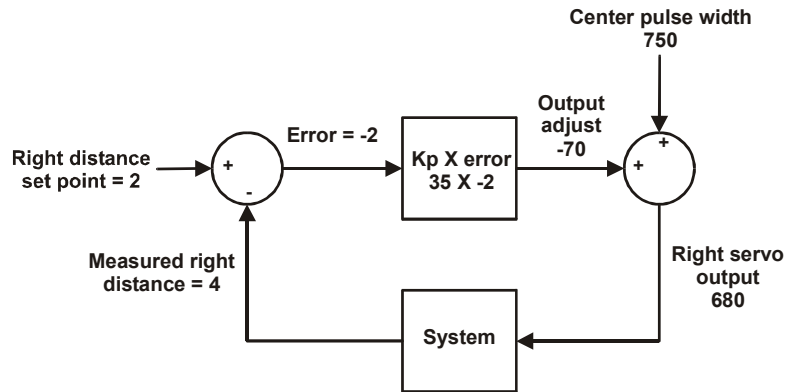


**Figure 8-4**  
Proportional  
Control Block  
Diagram for  
Right Servo  
and IR LED  
and Detector  
Pair

Let's take a closer look at the numbers in Figure 8-4 to learn how proportional control works. This particular example is for the right IR LED/detector and right servo. The set point is 2, which means we want the Boe-Bot to maintain a distance of 2 between itself and any object it detects. The measured distance is 4, which is too far away. The error is the set point minus the measured distance which is  $2 - 4 = -2$ . This is indicated by the symbols inside the circle on the left. This circle is called a summing junction. Next, the error feeds into an operator block. This block shows that error will be multiplied by a value called a proportional constant (Kp). The value of Kp is 35. The block's output shows the result of  $-2 \times 35 = -70$ , which is called the output adjust. This output adjust goes into another summing junction, and this time it is added to the servo's center pulse width of 750. The result is a 680 pulse width that will make the servo turn about  $\frac{3}{4}$  speed clockwise. That makes the Boe-Bot's right wheel roll forward, toward the object. This correction goes into the overall system, which consists of the Boe-Bot, and the object, that was at a measured distance of 4.

The next time through the loop, the measured distance might change, but that's OK because regardless of the measured distance, this control loop will calculate a value that will cause the servo to move to correct any error. The correction is always proportional to the error, which is the difference between the set point and measured distances.

A control loop always has a set of equations that govern the system. The block diagram in Figure 8-4 is a way of visually describing this set of equations. Here are the equations that can be taken from this block diagram, along with solutions.

$$\text{Error} = \text{Right distance set point} - \text{Measured right distance}$$

$$\begin{aligned}
 &= 2 - 4 \\
 \text{Output adjust} &= \text{error} \times K_p \\
 &= -2 \times 35 \\
 &= -70 \\
 \text{Right servo output} &= \text{Output adjust} + \text{Center pulse width} \\
 &= -70 + 750 \\
 &= 680
 \end{aligned}$$

By making some substitutions, the three equations above can be reduced to this one, which will give you the same result.

$$\begin{aligned}
 \text{Right servo output} &= (\text{Right distance set point} - \text{Measured right distance}) \times K_p \\
 &\quad + \text{Center pulse width}
 \end{aligned}$$

By substituting the values from the example, we can see that the equation still works:

$$\begin{aligned}
 &= ((2 - 4) \times 35) + 750 \\
 &= 680
 \end{aligned}$$

8

The left servo and IR pair have a similar algorithm shown in Figure 8-5. The difference is that  $K_p$  is  $-35$  instead of  $+35$ . Assuming the same measured value at the right IR pair, the output adjust results is a pulse width of 820. Here is the equation and calculations for this block diagram:

$$\begin{aligned}
 \text{Left servo output} &= (\text{Left distance set point} - \text{Measured left distance}) \times K_p \\
 &\quad + \text{Center pulse width} \\
 &= ((2 - 4) \times -35) + 750 \\
 &= 820
 \end{aligned}$$

The result of this control loop is a pulse width that makes the left servo turn about  $\frac{3}{4}$  of full speed counterclockwise. This is also a forward pulse for the left wheel. The idea of feedback is that the system's output is re-sampled, by the shadow Boe-Bot taking another distance measurement. Then the control loop repeats itself again and again and again...roughly 40 times per second.

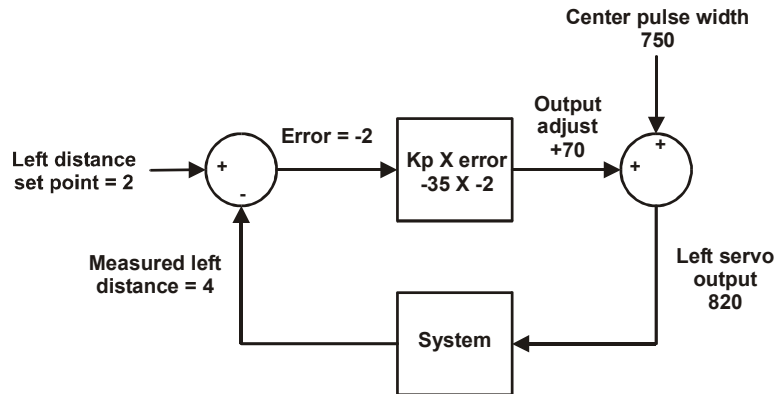


**Figure 8-5**  
Proportional  
Control Block  
Diagram for  
Left Servo and  
IR LED and  
Detector Pair

### Programming the Boe-Bot Shadow Vehicle

Remember that the equation for the right servo's output was:

$$\text{Right servo output} = (\text{Right distance set point} - \text{Measured right distance}) \times Kp + \text{Center pulse width}$$

Here is an example of solving this same equation in PBASIC. The right distance set point is 2, the measured distance is a variable named `distanceRight` that will store the IR distance measurement, `Kp` is 35, and the center pulse width is 750:

```
pulseRight = 2 - distanceRight * 35 + 750
```



**Remember that in PBASIC math expressions are executed from left to right.** First, `distanceRight` is subtracted from 2. The result of this subtraction is then multiplied by `Kp`, and after that, the product is added to the center pulse width.

**You can use parentheses to force a calculation that is further to the right in a line of PBASIC code to be completed first.** Recall this example: you can rewrite this line of PBASIC code:

```
pulseRight = 2 - distanceRight * 35 + 750
```

like this:

```
pulseRight = 35 * (2 - distanceRight) + 750
```

In this expression, 35 is multiplied by the result of `(2 - distanceRight)`, then the product is added to 750.



The left servo is different because  $K_p$  for that system is -35

```
pulseLeft = 2 - distanceLeft * (-35) + 750
```

Since the values -35, 35, 2, and 750 all have names, it's definitely a good place for some constant declarations.

```
Kpl CON -35
Kpr CON 35
SetPoint CON 2
CenterPulse CON 750
```

With these constant declarations in the program, you can use the name `Kpl` in place of -35, `Kpr` in place of 35, `SetPoint` in place of 2, and `CenterPulse` in place of 750. After these constant declarations, the proportional control calculations now look like this:

```
pulseLeft = SetPoint - distanceLeft * Kpl + CenterPulse
pulseRight = SetPoint - distanceRight * Kpr + CenterPulse
```

8

The convenient thing about declaring constants for these values is that you can change them in one place, at the beginning of the program. The changes you make at the beginning of the program will be reflected everywhere these constants are used. For example, by changing the `Kpl CON` directive from -35 to -40, every instance of `Kpl` in the entire program changes from -35 to -40. This is exceedingly useful for experimenting with and tuning the right and left proportional control loops.

### Example Program – FollowingBoeBot.bs2

FollowingBoeBot.bs2 repeats the proportional control loop just discussed with every servo pulse. In other words, before each pulse, the distance is measured and the error signal is determined. Then the error is multiplied by  $K_p$ , and the resulting value is added/subtracted to/from the pulse widths that are sent to the left/right servos.

- ✓ Enter, save, and run FollowingBoeBot.bs2.
- ✓ Point the Boe-Bot at an  $8\frac{1}{2} \times 11$ " sheet of paper held in front of it as though it's a wall-obstacle. The Boe-Bot should maintain a fixed distance between itself and the sheet of paper.
- ✓ Try rotating the sheet of paper slightly. The Boe-Bot should rotate with it.
- ✓ Try using the sheet of paper to lead the Boe-Bot around. The Boe-Bot should follow it.

- ✓ Move the sheet of paper too close to the Boe-Bot, and it should back up, away from the paper.

```
' -----[Title]-----
' Robotics with the Boe-Bot - FollowingBoeBot.bs2
' Boe-Bot adjusts its position to keep objects it detects in zone 2.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

DEBUG "Program Running!"

' -----[Constants]-----

Kpl CON -35
Kpr CON 35
SetPoint CON 2
CenterPulse CON 750

' -----[Variables]-----

freqSelect VAR Nib
irFrequency VAR Word
irDetectLeft VAR Bit
irDetectRight VAR Bit
distanceLeft VAR Nib
distanceRight VAR Nib
pulseLeft VAR Word
pulseRight VAR Word

' -----[Initialization]-----

FREQOUT 4, 2000, 3000

' -----[Main Routine]-----

DO

 GOSUB Get_Ir_Distances

 ' Calculate proportional output.

 pulseLeft = SetPoint - distanceLeft * Kpl + CenterPulse
 pulseRight = SetPoint - distanceRight * Kpr + CenterPulse

 GOSUB Send_Pulse

LOOP

' -----[Subroutine - Get IR Distances]-----
```

```

Get_Ir_Distances:
 distanceLeft = 0
 distanceRight = 0
 FOR freqSelect = 0 TO 4
 LOOKUP freqSelect, [37500,38250,39500,40500,41500], irFrequency

 FREQOUT 8,1,irFrequency
 irDetectLeft = IN9
 distanceLeft = distanceLeft + irDetectLeft

 FREQOUT 2,1,irFrequency
 irDetectRight = IN0
 distanceRight = distanceRight + irDetectRight
 NEXT
 RETURN

' -----[Subroutine - Get Pulse]-----

Send_Pulse:
 PULSOUT 13,pulseLeft
 PULSOUT 12,pulseRight
 PAUSE 5
 RETURN

```

### How the FollowingBoeBot.bs2 Works

FollowingBoeBot.bs2 declares four constants, **Kpr**, **Kpl**, **SetPoint**, and **CenterPulse** using the **CON** directive. Everywhere you see **SetPoint**, it's actually the number 2 (a constant). Likewise, everywhere you see either **Kpl**, it's actually the number -35. **Kpr** is actually 35, and **CenterPulse** is 750.

|             |     |     |
|-------------|-----|-----|
| Kpl         | CON | -35 |
| Kpr         | CON | 35  |
| SetPoint    | CON | 2   |
| CenterPulse | CON | 750 |

The first thing the main routine does is call the **Get\_Ir\_Distances** subroutine. After the **Get\_Ir\_Distances** subroutine is finished, **distanceLeft** and **distanceRight** each contain a number corresponding to the zone in which an object was detected for both the left and right IR pairs.

```

DO

 GOSUB Get_Ir_Distances

```

The next two lines of code implement the proportional control calculations for each servo.

```
' Calculate proportional output.

pulseLeft = SetPoint - distanceLeft * Kpl + CenterPulse
pulseRight = SetPoint - distanceRight * Kpr + CenterPulse
```

Now that the `pulseLeft` and `pulseRight` calculations are done, the `Send_Pulse` subroutine can be called.

```
GOSUB Send_Pulse
```

The `LOOP` portion of the `DO...LOOP` sends the program back to the command immediately following the `DO` at the beginning of the main loop.

```
LOOP
```

### **Your Turn**

Figure 8-6 shows a lead Boe-Bot followed by a shadow Boe-Bot. The lead Boe-Bot is running a modified version of `FastIrRoaming.bs2`, and the shadow Boe-Bot is running `FollowingBoeBot.bs2`. Proportional control makes the shadow Boe-Bot a very faithful follower. One lead Boe-Bot can string along a chain of 6 or 7 shadow Boe-Bots. Just add the lead Boe-Bot's side panels and tailgate to the rest of the shadow Boe-Bots in the chain.

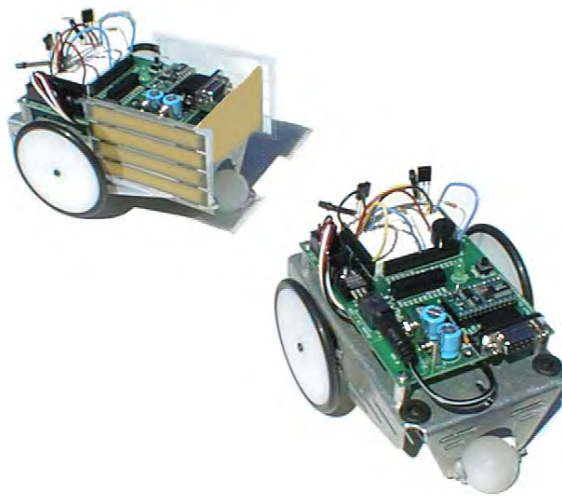


**Figure 8-6**  
Lead Boe-Bot (left)  
and Shadow Boe-  
Bot (right)

- ✓ If you are part of a class, mount paper panels on the tail and both sides of the lead Boe-Bot as shown in Figure 8-6.
- ✓ If you are not part of a class (and only have one Boe-Bot) the shadow vehicle will follow a piece of paper or your hand just as well as it follows a lead Boe-Bot.
- ✓ Replace the 1 k $\Omega$  resistors that connect the lead Boe-Bot's P2 and P8 to the IR LEDs with 470  $\Omega$  or 220  $\Omega$  resistors.
- ✓ Program the lead Boe-Bot for object avoidance using a modified version of FastIrRoaming.bs2. Open FastIrRoaming.bs2, and rename it SlowerIrRoamingForLeadBoeBot.bs2.
- ✓ Make these modifications to SlowerIrRoamingForLeadBoeBot.bs2:
  - ✓ Increase all **PULSOUT Duration** arguments that are now 650 to 710.
  - ✓ Reduce all **PULSOUT Duration** arguments that are now 850 to 790.
- ✓ The shadow Boe-Bot should be running FollowingBoeBot.bs2 without any modifications.
- ✓ With both Boe-Bots running their respective programs, place the shadow Boe-Bot behind the lead Boe-Bot. The shadow Boe-Bot should follow at a fixed distance, so long as it is not distracted by another object such as a hand or a nearby wall.

You can adjust the set points and proportionality constants to change the shadow Boe-Bot's behavior. Use your hand or a piece of paper to lead the shadow Boe-Bot while doing these exercises:

- ✓ Try running `FollowingBoeBot.bs2` using values of `kpr` and `kpl` constants, ranging from 15 to 50. Note the difference in how responsive the Boe-Bot is when following an object.
- ✓ Try making adjustments to the value of the `SetPoint` constant. Try values from 0 to 4.

### ACTIVITY #3: FOLLOWING A STRIPE

Figure 8-7 shows an example of a course you can build and program your Boe-Bot to follow. Each stripe in this course is three long pieces of  $\frac{3}{4}$  in (19 mm) vinyl electrical tape placed edge to edge on white poster board. No paper should be visible between the strips of electrical tape.

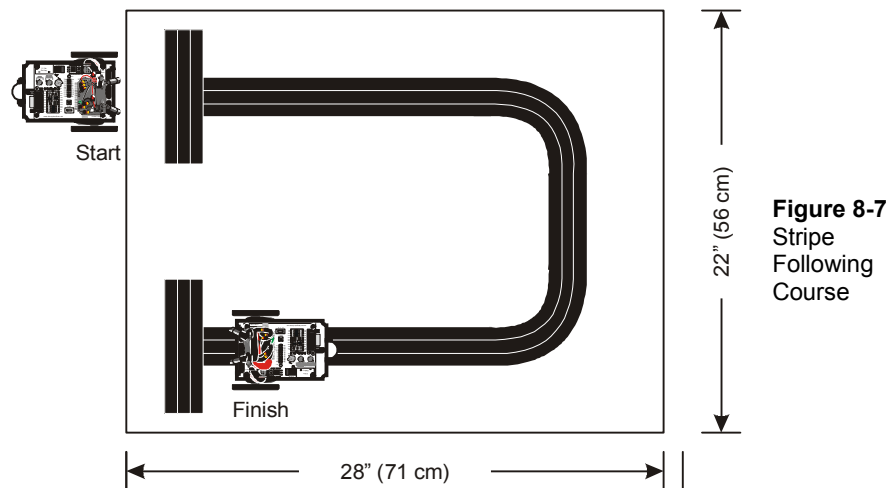


**Figure 8-7**  
Stripe  
Following  
Course

#### Building and Testing the Course

For successful navigation of this course, some testing and Boe-Bot adjustment will be required.

### Materials Required

- (1) Sheet of poster board – Approximate dimensions: 22 X 28 in (56 X 71 cm)
- (1) Roll of Black Vinyl Electrical Tape –  $\frac{3}{4}$ " (19 mm) wide.

- ✓ Use your poster board and electrical tape to build the course shown in Figure 8-7.

### Testing the Stripe

- ✓ Point your IR pairs downward and outward as shown in Figure 8-8 (Figure 7-8 from page 255 repeated here for convenience).

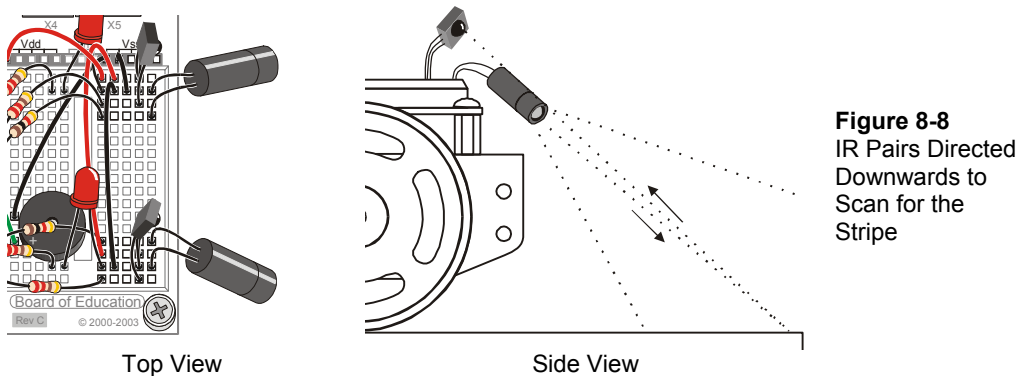


**Figure 8-8**  
IR Pairs Directed  
Downwards to  
Scan for the  
Stripe

8

- ✓ Make sure your electrical tape course is free of fluorescent light interference. See Sniffing for IR Interference (page 245).
- ✓ Replace the 1 k $\Omega$  resistors in series with the IR LEDs with 2 k $\Omega$  resistors to make the Boe-Bot more nearsighted.
- ✓ Run DisplayBothDistances.bs2 from page 275. Keep your Boe-Bot connected to its serial cable so that you can see the displayed distances.
- ✓ Start by placing your Boe-Bot so that it is looking directly at the white background of your poster board as shown in Figure 8-9.
- ✓ Verify that your zone readings indicate that an object is detected in a very close zone. Both sensors should give you a 1 or 0 reading.

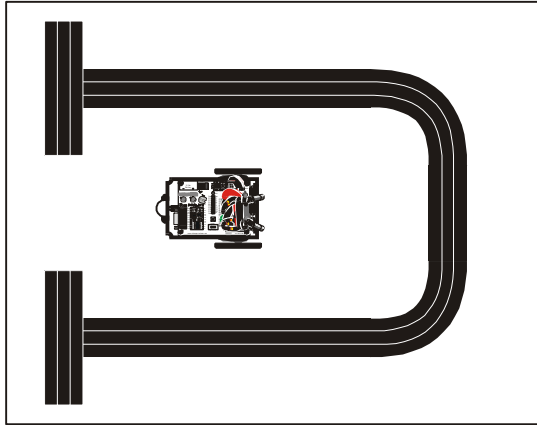


**Figure 8-9**  
Test for Low Zone  
Number – Top View

- ✓ Place your Boe-Bot so that both IR LED/detector pairs are focused directly at the center of your electrical tape stripe (see Figure 8-10 and Figure 8-11).
- ✓ Then, adjust your Boe-Bot's position (toward and away from the tape) until both zone values reach the 4 or 5 level indicating that either a far away object is detected, or no object is detected.
- ✓ If you are having difficulties getting the higher readings with your electrical tape course, see Trouble Shooting the Electrical Tape Course on page 289.

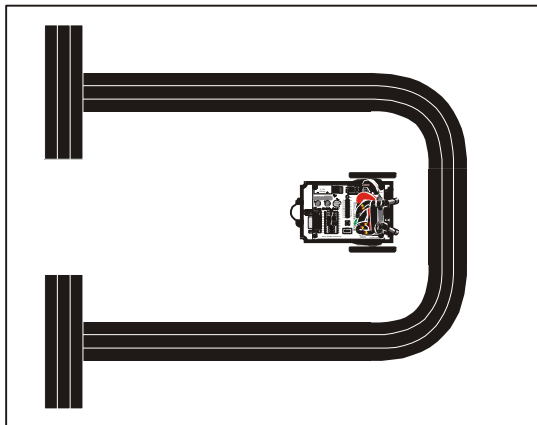


**Figure 8-10**  
Test for High Zone  
Number – Top View



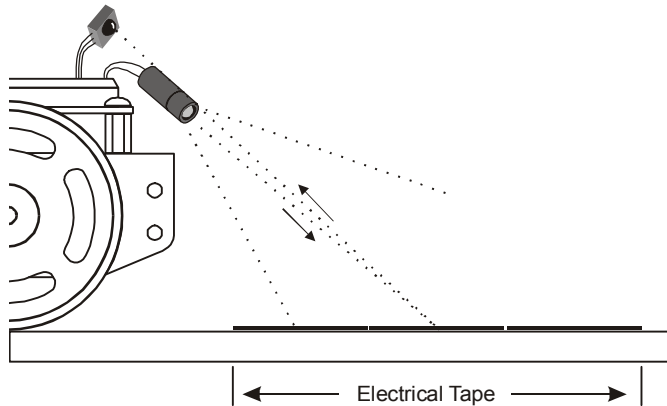


**Figure 8-11**  
Test for High  
Zone Number  
– Side View

#### Trouble Shooting the Electrical Tape Course

If you are unable to get a high zone value when the IR detectors are focused on the electrical tape, take a separate piece of paper, and make a stripe that's four strips wide instead of three. If the zone numbers are still low, make sure that you are using 2 k $\Omega$  resistors (red-black-red) in series with your IR LEDs. You can also try a 4.7 k $\Omega$  resistor to make the Boe-Bot more nearsighted. If none of this works, try a different brand of black vinyl electrical tape. Adjusting the IR LED/detector so that it is focused closer to or further from the front of the Boe-Bot (see Figure 8-11) may also help.



If you are having trouble with low zone measurements when reading the white surface, try pointing the IR LEDs and detectors further downward and toward the front of the Boe-Bot, but be careful not to cause reflection off the chassis. You can also try a lower-value resistor like 1 k $\Omega$  (brown-black-red).

If you are using the older shrink wrapped IR LEDs instead of the ones with the 2-piece plastic shields, you may be having trouble with getting a low zone number when the IR LED/detectors are focused on the white background. These LEDs may need 470  $\Omega$  (yellow-violet-brown) or 220  $\Omega$  (red-red-brown) resistors in series. Also, make sure that the leads of the IR LEDs are not touching each other.

- ✓ Now, place the Boe-Bot on the course so that its wheels straddle the black line. The IR detectors should be facing slightly outward. See close-up in Figure 8-12. Verify that the distance reading for both IR pairs is 0 or 1 again. If the readings are higher, it means they need to be pointed slightly further outward, away from the edge of the stripe.

When you move the Boe-Bot in either direction indicated by the double-arrow, one or the other IR pair will become focused on the electrical tape. When you do this, the readings

for the pair that is now over the electrical tape should increase to 4 or 5. Keep in mind that if you move the Boe-Bot toward its left, the right detectors should increase in value, and if you move the Boe-Bot toward its right, the left detectors should show the higher value.

- ✓ Adjust your IR LED/detector pairs until the Boe-Bot passes this last test. Then you will be ready to try following the stripe.

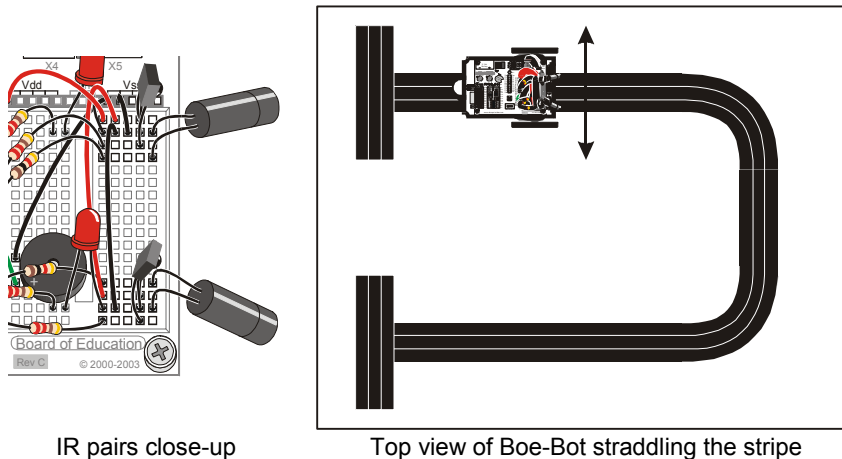


**Figure 8-12**  
Stripe Scan  
Test

### **Programming for Stripe Following**

You will only need to make a few small adjustments to `FollowingBoeBot.bs2` from page 282 to make it work for following a stripe. First, the Boe-Bot should move toward objects closer than the `SetPoint` and away from objects further from the `SetPoint`. This is the opposite of how `FollowingBoeBot.bs2` behaved. To reverse the direction the Boe-Bot moves when it senses that the object is not at the `SetPoint` distance, simply change the signs of `Kp1` and `Kpr`. In other words, change `Kp1` from -35 to 35, and change `Kpr` from 35 to -35. You will need to experiment with your `SetPoint`. Values from 2 to 4 tend to work best. This next example program will use a `SetPoint` of 3.

### **Example Program: `StripeFollowingBoeBot.bs2`**

- ✓ Open `FollowingBoeBot.bs2` and save it as `StripeFollowingBoeBot.bs2`.
- ✓ Change the `SetPoint` declaration from `SetPoint CON 2` to `SetPoint CON 3`.
- ✓ Change `Kp1` from -35 to 35.

- ✓ Change **Kpr** from 35 to -35.
- ✓ Run the program (shown below).
- ✓ Place your Boe-Bot at the “Start” location shown in Figure 8-13. The Boe-Bot should wait there until you place your hand in front of its IR pairs. It will then roll forward. When it clears the starting stripe, take your hand away, and it should start tracking the stripe. When it sees the “Finish” stripe, it should stop and wait there.
- ✓ Assuming that you can get distance readings of 5 from the electrical tape and 0 from the poster board, **SetPoint** constant values of 2, 3, and 4 should work. Try different **SetPoint** values and make notes of your Boe-Bot’s performance on the track.

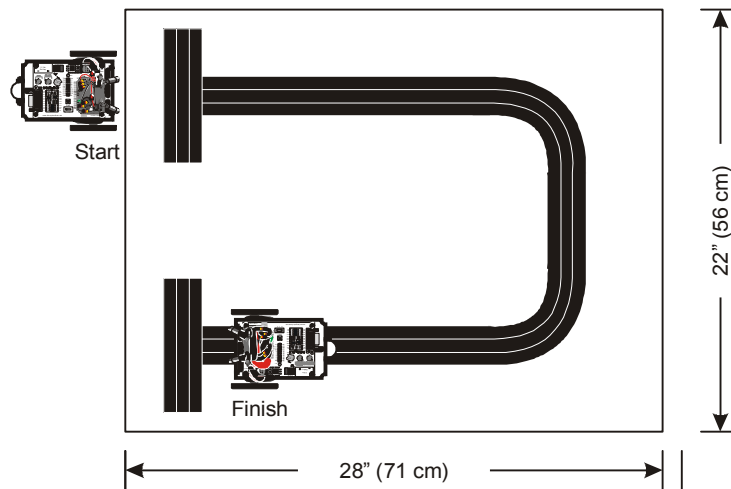


**Figure 8-13**  
Stripe  
Following  
Course

8

```
' -----[Title]-----
' Robotics with the Boe-Bot - StripeFollowingBoeBot.bs2
' Boe-Bot adjusts its position to move toward objects that are closer than
' zone 3 and away from objects further than zone 3. Useful for following a
' 2.25 inch wide vinyl electrical tape stripe.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

DEBUG "Program Running!"

' -----[Constants]-----

Kpl CON 35 ' Change from -35 to 35
```

```

Kpr CON -35 ' Change from 35 to -35
SetPoint CON 3 ' Change from 2 to 3.
CenterPulse CON 750

' -----[Variables]-----

freqSelect VAR Nib
irFrequency VAR Word
irDetectLeft VAR Bit
irDetectRight VAR Bit
distanceLeft VAR Nib
distanceRight VAR Nib
pulseLeft VAR Word
pulseRight VAR Word

' -----[Initialization]-----

FREQOUT 4, 2000, 3000

' -----[Main Routine]-----

DO

 GOSUB Get_Ir_Distances

 ' Calculate proportional output.

 pulseLeft = SetPoint - distanceLeft * Kpl + CenterPulse
 pulseRight = SetPoint - distanceRight * Kpr + CenterPulse

 GOSUB Send_Pulse

LOOP

' -----[Subroutine - Get IR Distances]-----

Get_Ir_Distances:
 distanceLeft = 0
 distanceRight = 0
 FOR freqSelect = 0 TO 4
 LOOKUP freqSelect, [37500,38250,39500,40500,41500], irFrequency

 FREQOUT 8,1,irFrequency
 irDetectLeft = IN9
 distanceLeft = distanceLeft + irDetectLeft

 FREQOUT 2,1,irFrequency
 irDetectRight = IN0
 distanceRight = distanceRight + irDetectRight
 NEXT
 RETURN

```

```
' -----[Subroutine - Get Pulse]-----
Send_Pulse:
 PULSOUT 13,pulseLeft
 PULSOUT 12,pulseRight
 PAUSE 5
 RETURN
```

### Your Turn – Stripe Following Contest

You can turn this into a contest with the lowest course time winning, provided the Boe-Bot faithfully waits at the “Start” and “Finish” stripes. You can make up other courses too. For best performance, experiment with different `SetPoint`, `Kp1`, and `Kpr` values.

## SUMMARY

Frequency sweep was introduced as a way of determining distance using the Boe-Bot's IR LED and detector. **FREQOUT** was used to send IR signals at frequencies ranging from 37.5 kHz (most sensitive) to 41.5 kHz (least sensitive). The distance was determined by tracking which frequencies caused the IR detector to report that an object was detected and which did not. Since not all of the frequencies were separated by the same value, the **LOOKUP** command was introduced as simple way to use the counting sequence supplied by a **FOR...NEXT** loop to index sequential lists of numbers.

Control systems were introduced along with closed loop control. Proportional control in a closed-loop system is an algorithm where the error is multiplied by a proportionality constant to determine the system's output. The error is the measured system output subtracted from the set point. For the Boe-Bot, both system output and set point were in terms of distance. The BASIC stamp was programmed in PBASIC to operate control loops for the both the left and right servos and distance detectors. By re-sampling distance and adjusting the servo output before sending pulses to the servos, the control loop made the Boe-Bot responsive to object motion. The Boe-Bot was able to use proportional control to lock onto and follow objects, and it also used it to track and follow a stripe of black electrical tape.



**Watch the Boe-Bot in Action at [www.parallax.com](http://www.parallax.com)!**

You can see the Boe-Bot solving Chapter 8 Project 2 and other Robotics video clips in the Robo Video Gallery under the Robotics Menu at [www.parallax.com](http://www.parallax.com).

## Questions

1. What would the relative sensitivity of the IR detector be if you use **FREQOUT** to send a 35 kHz harmonic? What is the relative sensitivity with a 36 kHz harmonic?
2. Consider the code snippet below. If the **index** variable is 4, which number will be placed in the **prime** variable in this **LOOKUP** command? What values will **prime** store when **index** is 0, 1, 2, and 7?

```
LOOKUP index, [2, 3, 5, 7, 11, 13, 17, 19], prime
```

3. In what order are PBASIC math expressions evaluated? How can you override that order?

4. What PBASIC directive do you use to declare a constant? How would you give the number 100 the name “BoilingPoint”?

### **Exercises**

1. List the sensitivity of the IR detector for each kHz frequency shown in Figure 8-1.
2. Write a segment of code that does the frequency sweep for just four frequencies instead of five.
3. Make a condensed checklist for the tests that should be performed to ensure faithful stripe following.

### **Projects**

- P1. Create different types of electrical tape intersections and program the Boe-Bot to navigate through them. The intersections could be 90° left, 90° right, three-way, and four-way. This will involve the Boe-Bot recognizing it is at an intersection. When the Boe-Bot executes `StripeFollowingBoeBot.bs2`, the Boe-Bot will stay still at intersections. The goal is to have the Boe-Bot realize it's not doing anything and break from its proportional control loop.

Hints: You can do this by creating two counters, one that increments by 1 each time through the `DO...LOOP`, and the other that only increments when the Boe-Bot delivers a forward pulse. When the counter that increments each time through the `DO...LOOP` gets to 60, use `IF...THEN` to check how many forward pulses were applied. If less than 30 forward pulses were applied, the Boe-Bot is probably stuck. Remember to reset both counters to zero each time the loop counter gets to 60. After the Boe-Bot recognizes that it is at an intersection, it needs to move to the top edge of the intersection, then back up and figure out whether it sees electrical tape or white background on the left and right, then make the correct 90° turn. Use a preprogrammed motion for turning 90°, without proportional control. For three-way and four-way intersections, the Boe-Bot may turn either right or left.

- P2. *Advanced Project* - Design a maze-solving contest of your own, and program the Boe-Bot to solve it!

## Questions

Q1. The relative sensitivity at 35 kHz is 30%. For 36 kHz, it's 50%

Q2. When `index = 4`, `prime = 11`

`index = 0`, `prime = 2`

`index = 1`, `prime = 3`

`index = 2`, `prime = 5`

`index = 7`, `prime = 19`

Q3. Expressions are evaluated left to right. To override, use parentheses to change the order.

Q4. Use the `CON` directive.

```
BoilingPoint CON 100
```

E1. Frequency (kHz): 34 35 36 37 38 39 40 41 42

Sensitivity : 14% 30% 50% 76% 100% 80% 55% 35% 16%

E2. To solve this problem, put only four frequencies in the `LOOKUP` list, and decrease the `FOR...NEXT` index by one.

```
FOR freqSelect = 0 TO 3
 LOOKUP freqSelect, [37500, 38750, 39500, 40500],
 irFrequency
 FREQOUT 8, 1, irFrequency
 irDetect = IN9
 ... commands not shown
NEXT
```

E3. • Sniff for IR interference with "IrInterferenceSniffer.bs2".

- Run Display BothDistances.bs2.
- White readings should be 0-1 in both sensors.
- Black readings should be 4-5 in both sensors.
- Straddle the line, both sensors should read 0-1.
- Move Boe-Bot back and forth over line, sensor over black line should read 4-5.

P1. In the solution below, the `Check_For_Intersection` subroutine implements the algorithm outlined. The left servo was arbitrarily chosen for counting the forward pulses. A bit-sized variable named `isStuck` is used as a flag to let the Main program know whether an intersection has been reached. In the `Navigate_Intersection` subroutine, the Boe-Bot goes forward past the intersection and then backs up, checking the sensors, using `DO...LOOP...UNTIL`. Then it makes a preprogrammed 90 degree turn in the correct direction. If the



intersection is a 3-way or 4-way intersection, the Boe-Bot will arbitrarily turn in the direction that black is first detected. A constant, **Turn90Degree**, is provided to tune the 90 degree turn. Some audible and visual indicators are included, which aid in troubleshooting and understanding what the Boe-Bot is seeing and deciding, as well as adding a bit of personality and fun.

```
' -----[Title]-----
' Robotics with the Boe-Bot - IntersectionsBoeBot.bs2
' Navigate 90 degree left/right, 3-way, and 4-way intersections.
' Based on StripeFollowingBoeBot.bs2

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.
DEBUG "Program Running!"

' -----[Constants]-----

Kpl CON 35 ' Left proportional constant
Kpr CON -35 ' Right proportional constant
SetPoint CON 3 ' 0-1 is White, 4-5 is Black
CenterPulse CON 750
Turn90Degree CON 30 ' Pulses needed for 90 turn

RightLED PIN 1 ' LED Indicators
LeftLED PIN 10

' -----[Variables]-----

freqSelect VAR Nib ' Sweep through 5 frequencies
irFrequency VAR Word ' Freq sent to IR emitter
irDetectLeft VAR Bit ' Store results from detectors
irDetectRight VAR Bit
distanceLeft VAR Nib ' Calculate distance zones
distanceRight VAR Nib
pulseLeft VAR Word ' Servo pulseWidths
pulseRight VAR Word
numPulses VAR Byte ' Count total pulses
fwdPulses VAR Byte ' Count forward pulses
counter VAR Byte
isStuck VAR Bit ' Boolean variable, is bot stuck?

' -----[Initialization]-----

FREQOUT 4, 2000, 3000

' -----[Main Routine]-----

DO
 GOSUB Get_Ir_Distances ' Read IR sensors
```

```

GOSUB Update_LEDs ' Indicate white/black line

' Calculate proportional output and move accordingly.
pulseLeft = SetPoint - distanceLeft * Kpl + CenterPulse
pulseRight = SetPoint - distanceRight * Kpr + CenterPulse
GOSUB Send_Pulse

GOSUB Check_For_Intersection ' Are we stuck at intersection?
IF (isStuck = 1) THEN
 GOSUB Make_Noise ' Audible indication
 GOSUB Navigate_Intersection ' Navigate through it
ENDIF

LOOP

' -----[Subroutines]-----

Navigate_Intersection:
' Go forward until both sensors read white, through the intersection.
DO
 pulseLeft = 850: pulseRight = 650 ' Forward
 GOSUB Send_Pulse
 GOSUB Get_Ir_Distances
 GOSUB Update_LEDs
LOOP UNTIL (distanceLeft <=2) AND (distanceRight <=2)

GOSUB Stop_Quickly ' Don't coast forward

' Now back up until one detector sees the black. L & R turn will see
' black on one detector. 3- or 4-way will see both black, turn toward
' whichever the bot sees first (random).
DO
 pulseLeft = 650: pulseRight = 850 ' Backward
 GOSUB Send_Pulse
 GOSUB Get_Ir_Distances
 GOSUB Update_LEDs
LOOP UNTIL (distanceLeft >=4) OR (distanceRight >=4)

GOSUB Stop_Quickly ' Don't coast backward

' Make 90 degree turn in direction of the detector which sees black
IF (distanceLeft >=4) THEN
 ' Left detector reads black
 FOR counter = 1 TO Turn90Degree
 ' Turn 90 degrees left
 ' without proportional control
 PULSOUT 13, 750
 PULSOUT 12, 650
 PAUSE 20
 ' so use PAUSE 20
 NEXT
ELSEIF (distanceRight >=4) THEN
 ' Right detector reads black
 ' Turn 90 degrees right
 FOR counter = 1 TO Turn90Degree
 PULSOUT 13, 850
 PULSOUT 12, 750
 NEXT

```

```

 PAUSE 20
 NEXT
ENDIF

' That's it. At this point the Boe-Bot should have turned 90 degrees
' to follow the intersection. Continue following the black line.

RETURN

Check_For_Intersection:
' Keep track of no. of pulses vs the forward pulses. If there are less
' than 30 forward pulses per total of 60 pulses, robot is likely stuck
' at an intersection.

isStuck = 0 ' Initialize Boolean variable
numPulses = numPulses + 1 ' Count total pulses sent

SELECT numPulses
CASE < 60
 IF (pulseLeft > CenterPulse) THEN
 fwdPulses = fwdPulses + 1 ' Count forward pulses
 ENDIF ' (forward is any pulse > 750)

CASE = 60 ' If we have sent 60 pulses
 IF (fwdPulses < 30) THEN ' how many were forward?
 isStuck = 1 ' If < 30, robot is stuck
 ENDIF

CASE > 60
 numPulses = 0 ' Reset counters back to zero
 fwdPulses = 0 ' (Could reset in =60 case but
ENDSELECT ' it spoils cool Make_Noise)
RETURN

Make_Noise:
' Makes an increasing tone, proportional to number of forward pulses
FOR counter = 1 TO fwdPulses STEP 3
 FREQOUT 4, 100, 3800 + (counter * 10)
NEXT
RETURN

Update_LEDs:
' Use LEDs to indicate whether detectors are seeing black or white.
' White = Off, Black = On. Black is a distance reading > or = 4 .
IF (distanceLeft >= 4) THEN HIGH LeftLED ELSE LOW LeftLED
IF (distanceRight >= 4) THEN HIGH RightLED ELSE LOW RightLED
RETURN

Stop_Quickly:
' This stops the wheels so the Boe-Bot does not "coast" forward.
PULSOUT 13, 750

```

```

PULSOUT 12, 750
PAUSE 20
RETURN

Get_Ir_Distances:
' Read both IR pairs and calculate the distance. Black line gives 4-5
' reading. White surface give 0-1 reading.
distanceLeft = 0
distanceRight = 0
FOR freqSelect = 0 TO 4
 LOOKUP freqSelect, [37500,38250,39500,40500,41500], irFrequency

 FREQOUT 8,1,irFrequency
 irDetectLeft = IN9
 distanceLeft = distanceLeft + irDetectLeft

 FREQOUT 2,1,irFrequency
 irDetectRight = IN0
 distanceRight = distanceRight + irDetectRight
NEXT
RETURN

Send_Pulse:
' Send a single pulse to the servos in between IR readings.
PULSOUT 13,pulseLeft
PULSOUT 12,pulseRight
PAUSE 5 ' PAUSE reduced due to IR readings
RETURN

```

- P2. If you create an interesting Boe-Bot maze project and you want to share it with others, you may want to join the StampsInClass Yahoo! Group, listed behind the title page of Robotics with the Boe-Bot. Or, you can email the parallax Educational Team directly at [stampsinclass@parallax.com](mailto:stampsinclass@parallax.com).

## Appendix A: PC to BASIC Stamp Communication Trouble-Shooting

---

Here is a list of things to try to quickly fix any difficulties getting the BASIC Stamp Editor to communicate with your BASIC Stamp:

- ✓ If you are using a Board of Education Rev C, make sure the power switch is set to position-1.
- ✓ Rule out dead batteries and incorrect or malfunctioning power supplies by using a new 9 V battery or four new 1.5 V AA alkaline batteries in the battery pack.
- ✓ Make sure the serial cable is firmly connected to both the computer's COM port and the DB9 connector on the Board of Education or BASIC Stamp HomeWork Board.
- ✓ Make sure that your serial cable is a normal (straight-through) serial cable. DO NOT USE A NULL MODEM CABLE. Most null modem cables are labeled NULL or Null Modem; visually inspect the cable for any such labeling.
- ✓ Disable any palmtop communication software.

If you are using a BASIC Stamp and Board of Education, also check the following:

- ✓ Make sure the BASIC Stamp was inserted into the socket right-side-up as shown in Figure 1-24 on page 17.
- ✓ If you are using a DC power supply that plugs into the wall, make sure it is plugged in to both the wall and the Board of Education. Verify that the green Pwr light on the Board of Education emits light when the DC supply is plugged in.
- ✓ Make sure the BASIC Stamp is firmly inserted into the socket.
- ✓ Visually inspect the BASIC Stamp module to make sure that none of the pins folded under the module instead of sinking into their sockets on the Board of Education.

If your Identification window looks similar to Figure A-1, it means that the BASIC Stamp Editor cannot find your BASIC Stamp on any COM port. If you have this problem, try the following:

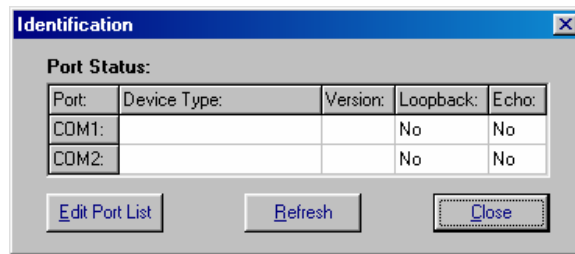


**Figure A-1**  
Identification Window

*Example: BASIC Stamp 2 not found on COM ports.*

If you know the number of the COM port, but it does not appear in the Identification Window:

- ✓ Use the Edit Port List button to add that COM port. When you return to the Identification window, click the Refresh button to find out if the BASIC Stamp 2 is now detected.
- ✓ Close the Identification window.
- ✓ In the BASIC Stamp Editor, Click Edit and select Preferences. Click the Editor Operation tab, and set the Default COM Port to AUTO.
- ✓ Try the Run → Identify test again.

If you are unsure of which COM port your BASIC Stamp is connected to, or if you are using a USB to serial port adaptor, you may need to look in your computer's Device Manager to find the list of COM ports in use.

- ✓ Click on your computer desktop's Start button.
- ✓ To view the list of COM ports in use, make the selections listed next to your operating system :

**Windows® 98:** Control Panel → System → Hardware → Device Manager → Ports(COM & LPT1).

**Windows® 2000:** Settings → Control Panel → System → Hardware → Device Manager → Ports (COM & LPT).

**Windows® XP:** Control Panel → Printers and Other Hardware. In the See Also box select System.

Select Hardware → Device Manager → Ports

**Windows® XP Pro:** Settings → Control Panel → System → Hardware → Device Manager → Ports (COM & LPT).



- ✓ If you are using a serial port (no USB to serial adaptor), make a note of the COM ports listed. If one or more of these COM ports do not appear in your BASIC Stamp Editor's list, make a note of the numbers for each COM port that doesn't appear in the list now.
- ✓ If you are using an FTDI USB to Serial adaptor, look for the COM port that reads FTDI USB to Serial COM...
- ✓ Repeat the Run → Identify test.
- ✓ Click the Edit Ports List button and add the missing COM port numbers.
- ✓ Repeat the Run → Identify test again, this time, the Identification window should "find" your BASIC Stamp 2.

#### Still no BASIC Stamp Detected?

- ✓ If you have more than one COM port, try connecting your Board of Education or BASIC Stamp HomeWork Board to a different COM port and see if Run → Identify works then.
- ✓ If you have a second computer, try it on the different computer.

If you get the error message "No BASIC Stamp Found" but the Run → Identify test shows a "Yes" in both columns for one of the COM ports, you may need to change a setting to your FIFO Buffers. This happens occasionally with Microsoft Windows® 98 and XP users. Make a note of the COM port with the "Yes" messages, and try this:

#### Windows® 98:

- ✓ Click on your computer desktop's Start button.
- ✓ Select Settings → Control Panel → System → Device Manager → Ports (COM & LPT).
- ✓ Select the COM port that was noted by the Run → Identify test.
- ✓ Select Properties → Port Settings → Advanced.
- ✓ Uncheck the box labeled "Use FIFO Buffers" then click OK.
- ✓ Click OK as needed to close each window and return to the BASIC Stamp Editor.
- ✓ Try downloading a program once more.

#### Windows® 2000:

- ✓ Click on your computer desktop's Start button.
- ✓ Select Settings → Control Panel → System → Hardware → Device Manager → Ports (COM & LPT).

- ✓ Select the COM port that was noted by the Run → Identify test.
- ✓ Select → Port Settings → Advanced.
- ✓ Uncheck the box labeled “Use FIFO Buffers” then click OK.
- ✓ Click OK as needed to close each window and return to the BASIC Stamp Editor.
- ✓ Try downloading a program once more.

Windows® XP:

- ✓ Click on your computer desktop’s *Start* button.
- ✓ Select Control Panel → Printers and Other Hardware.
- ✓ In the See Also box select System.
- ✓ Select Hardware → Device Manager → Ports.
- ✓ Enter the COM port number noted by the Run → Identify test.
- ✓ Select Port Settings → Advanced.
- ✓ Uncheck the box labeled “Use FIFO Buffers” then click OK.
- ✓ Click OK to close each window as needed and return to the BASIC Stamp Editor.
- ✓ Try downloading a program once more.

Windows® XP Pro:

- ✓ Click on your computer desktop’s *Start* button.
- ✓ Select Control Panel → System → Hardware → Device Manager → Ports(COM & LPT1).
- ✓ Select the Communications Port number noted by the Run → Identify test.
- ✓ Select Properties → Port Settings → Advanced.
- ✓ Uncheck the box labeled “Use FIFO Buffers” then click OK.
- ✓ Click OK to close each window as needed and return to the BASIC Stamp Editor.
- ✓ Try downloading a program once more.
- ✓

If none of these solutions work, you may go to [www.parallax.com](http://www.parallax.com) and follow the Support link. Or, email [support@parallax.com](mailto:support@parallax.com) or call Tech Support toll free at 1-888-99-STAMP.



## Appendix B: BASIC Stamp and Carrier Board Components and Features

### The BASIC STAMP® 2 Microcontroller Module

Figure B-1 shows a close-up of the BASIC Stamp® 2 microcontroller module. Its major components and their functions are indicated by labels.

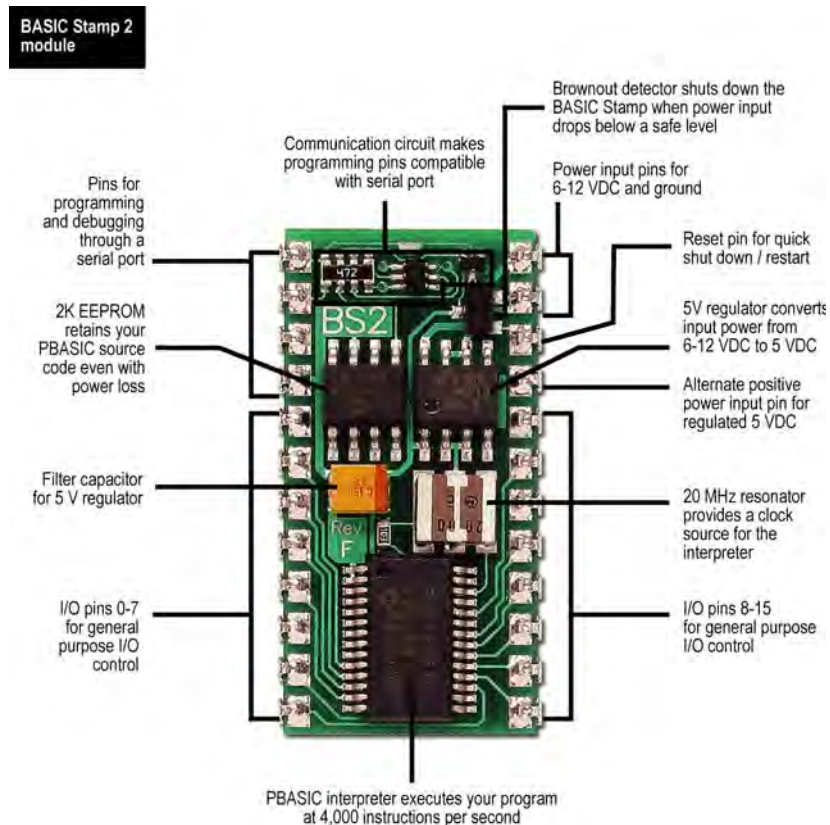


**Figure B-1:** BASIC Stamp® 2 Microcontroller Module Components and Their Functions

### The Board of Education® Rev C Carrier Board

The Board of Education® Rev C carrier board for BASIC Stamp® 24-pin microcontroller modules is shown in Figure B-2. Its major components and their functions are indicated by labels.

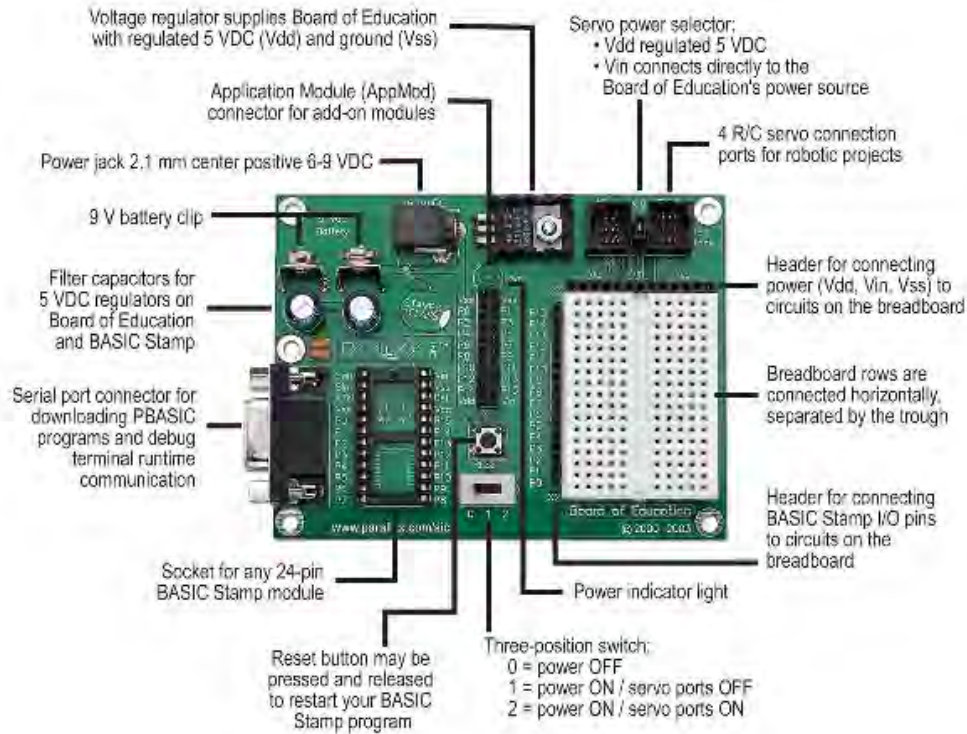


**Figure B-2:** Board of Education® Rev C Carrier Board

### The BASIC Stamp® HomeWork Board™ Project Platform

The BASIC Stamp® HomeWork Board™ project platform is shown in Figure B-3. Its major components and their functions are indicated by labels.

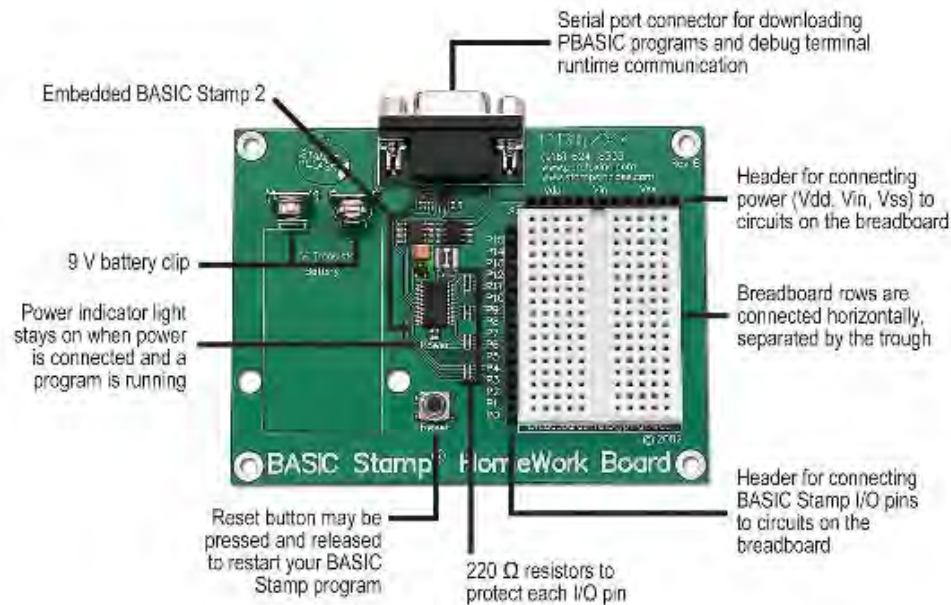


**Figure B-3:** BASIC Stamp® HomeWork Board™ Project Platform

### The Board of Education® Rev B Carrier Board

Figure B-4 shows the Board of Education® Rev B carrier board for BASIC Stamp® 24-pin microcontroller modules. Its major components and their functions are indicated by labels.

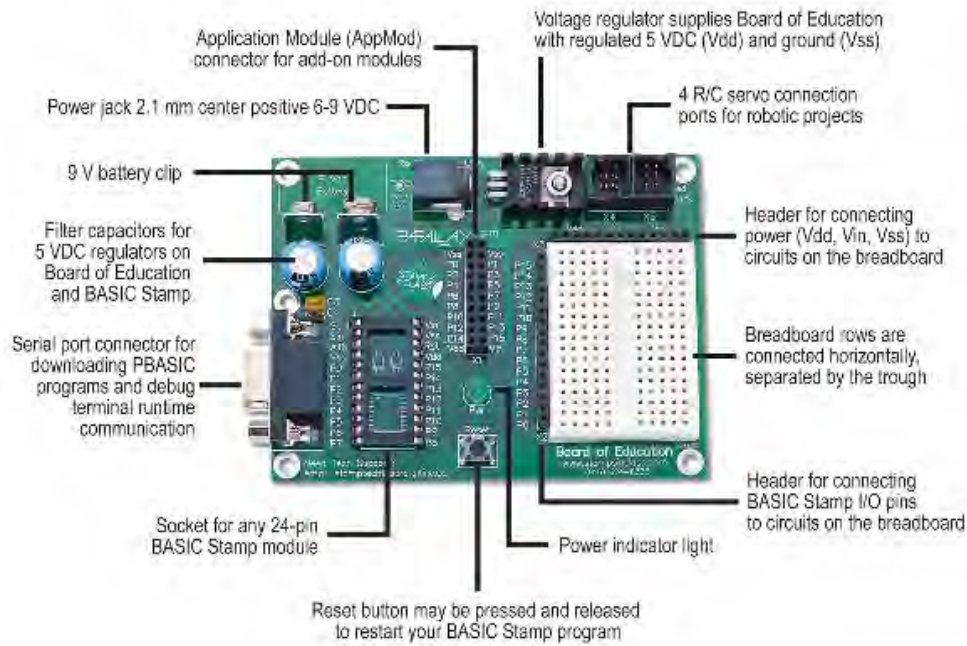


**Figure B-4:** Board of Education® Rev B Carrier Board

## Appendix C: Resistor Color Codes

Resistors like the ones we are using in this student guide have colored stripes that tell you what their resistance values are. There is a different color combination for each resistance value. For example, the color code for the 470  $\Omega$  resistor is yellow-violet-brown.



There may be a fourth stripe that indicates the resistor's tolerance. Tolerance is measured in percent, and it tells how far off the part's true resistance might be from the labeled resistance. The fourth stripe could be gold (5%), silver (10%), or no stripe (20%). For the activities in this book, a resistor's tolerance does not matter, but its value does.

Each color bar that tells you the resistor's value corresponds to a digit, and these colors/digits are listed in Table C-1. Figure C-1 shows how to use each color bar with the table to determine the value of a resistor.

| Table C-1:<br>Resistor Color Code<br>Values |        |
|---------------------------------------------|--------|
| Digit                                       | Color  |
| 0                                           | Black  |
| 1                                           | Brown  |
| 2                                           | Red    |
| 3                                           | Orange |
| 4                                           | Yellow |
| 5                                           | Green  |
| 6                                           | Blue   |
| 7                                           | Violet |
| 8                                           | Gray   |
| 9                                           | White  |

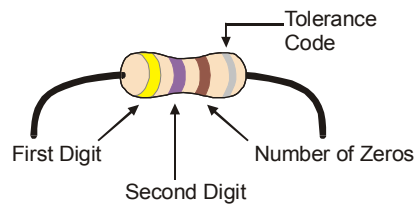


**Figure C-1**  
Resistor Color  
Codes

Here is an example that shows how Table C-1 and Figure C-1 can be used to figure out a resistor value by proving that yellow-violet-brown is really 470  $\Omega$ :

- First stripe is yellow, which means leftmost digit is a 4.
- Second stripe is violet, which means next digit is a 7.

- Third stripe is brown. Since brown is 1, it means add one zero to the right of the first two digits.

Yellow-Violet-Brown = 4-7-0.

## Appendix D: Breadboarding Rules

Look at your Board of Education or HomeWork Board. The white square with lots of holes, or sockets, in it is called a solderless breadboard. This breadboard, combined with the black strips of sockets along two of its sides, is called the prototyping area (shown in Figure D-1).

The example circuits in this text are built by plugging components such as resistors, LEDs, speakers, and sensors into these small sockets. Components are connected to each other with the breadboard sockets. You will supply your circuit with electricity from the power terminals, which are the black sockets along the top labeled Vdd, Vin, and Vss. The black sockets on the left are labeled P0, P1, up through P15. These sockets allow you to connect your circuit to the BASIC Stamp's input/output pins.

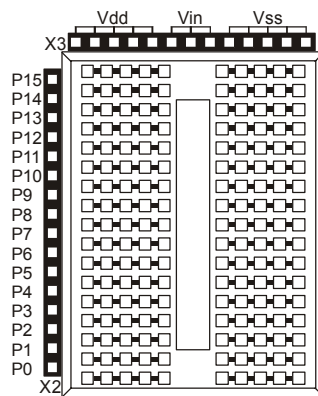


**Figure D-1**  
Prototyping Area

*Power terminals (black sockets along top), I/O pin access (black sockets along the side), and solderless breadboard (white sockets)*

The breadboard has 17 rows of sockets separated into two columns by a trough. The trough splits each of the seventeen rows of sockets into two rows of five. Each row of five sockets is electrically connected inside the breadboard. You can use these rows of sockets to connect components together as dictated by a circuit schematic. If you insert two wires into any two sockets in the same 5-socket row, they are electrically connected to each other.

A circuit schematic is a roadmap that shows how to connect components together. It uses unique symbols each representing a different component. These component symbols are connected by lines to indicate an electrical connection. When two circuit symbols are

connected by lines on a schematic, the line indicates that an electrical connection is made. Lines can also be used to connect components to voltage supplies. Vdd, Vin, and Vss all have symbols. Vss corresponds to the negative terminal of the battery supply for the Board of Education or BASIC Stamp HomeWork Board. Vin is the battery's positive terminal, and Vdd is regulated to +5 volts.

Let's take a look at an example that uses a schematic to connect the parts shown in Figure D-2. For each of these parts, the part drawing is shown above the schematic symbol.

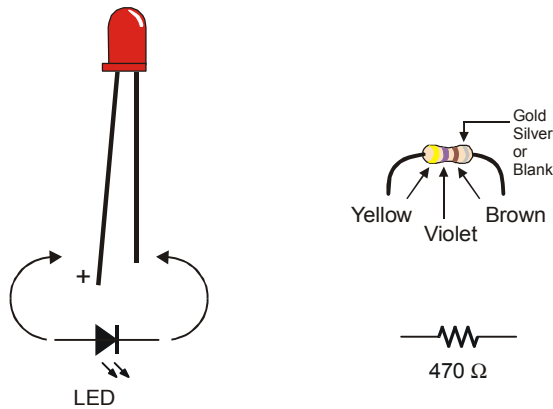


**Figure D-2**  
Part Drawings and  
Schematic Symbols

*LED(left) and  
470  $\Omega$  resistor (right)*

Figure D-3 shows an example of a circuit schematic on the left and a drawing of a circuit that can be built using this schematic on the right. Notice how the schematic shows that one end of the jagged line that denotes a resistor is connected to the symbol for Vdd. In the drawing, one of the resistor's two leads is plugged into one of the sockets labeled Vdd. In the schematic, the other terminal of the resistor symbol is connected by a line to the + terminal of the LED symbol. Remember, the line indicates the two parts are electrically connected. In the drawing, this is accomplished by plugging the other resistor lead into the same row of 5 sockets as the + lead on the LED. This electrically connects the two leads. The other terminal of the LED is shown connected to the Vss symbol in the schematic. In the drawing, the other lead of the LED is plugged into one of the sockets labeled Vss.



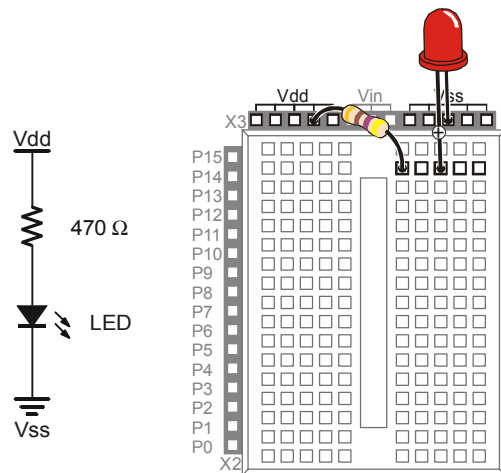


**Figure D-3**  
Example Schematic and  
Wiring Diagram

*Schematic (left) and wiring  
diagram (right)*

**D**

Figure D-4 shows a second example of a schematic and wiring diagram. This schematic shows P14 connected to one end of a resistor, with the other end connected to the + terminal of an LED, and the – terminal of the LED is connected to Vss. The schematic only differs by one connection. The resistor lead that used to be connected to Vdd is now connected to BASIC Stamp I/O pin P14. The schematic might look more different than that, mainly because the resistor is shown drawn horizontally instead of vertically. But in terms of connections, it only differs by one, P14 in place of Vdd. The wiring diagram shows how this difference is handled with the resistor lead that used to be plugged into Vdd, now plugged into P14.

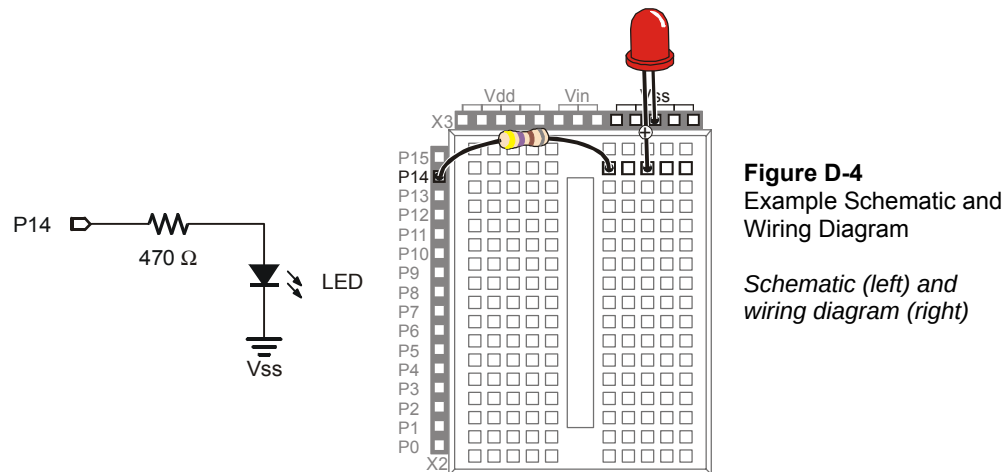


**Figure D-4**  
Example Schematic and  
Wiring Diagram

*Schematic (left) and  
wiring diagram (right)*

Here is a more complex example that involves two additional parts, a photoresistor and a capacitor. The schematic symbols and part drawings for these components are shown in Figure D-5.

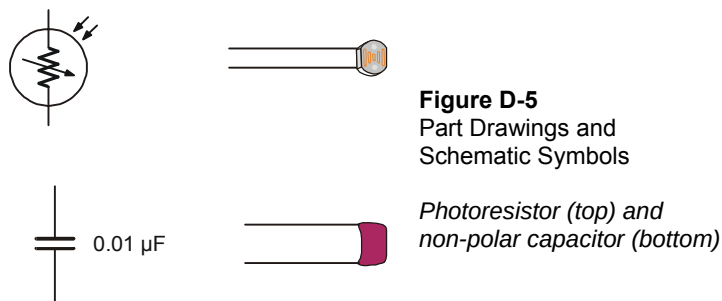


**Figure D-5**  
Part Drawings and  
Schematic Symbols

*Photoresistor (top) and  
non-polar capacitor (bottom)*

Since this schematic shown in Figure D-6 calls for a 220 Ω resistor, the first step is to consult Appendix C: Resistor Color Codes to determine the color code for a 220 Ω resistor. The color code is Red, Red, Brown. This resistor is connected to P6 in the schematic, which corresponds to the resistor lead plugged into the socket labeled P6 in

the prototyping area (Figure D-7). In the schematic, the other lead of the resistor is connected to not one, but two other component terminals. A terminal from the photoresistor and capacitor both share this connection. On the breadboard, the other resistor lead is plugged into one of the rows of 5 sockets. This row also has leads from the capacitor and photoresistor plugged into it. In the schematic, the other terminals of the photoresistor and capacitor are connected to Vss. Here is a trick to keep in mind when building circuits on a breadboard. You can use a wire to connect an entire row on the breadboard to another row, or even to I/O pins or power terminals such as Vdd or Vss. In this case, a wire was used to connect Vss to a row on the breadboard. Then, the leads for the capacitor and photoresistor were plugged into the same row, completing the circuit.

D

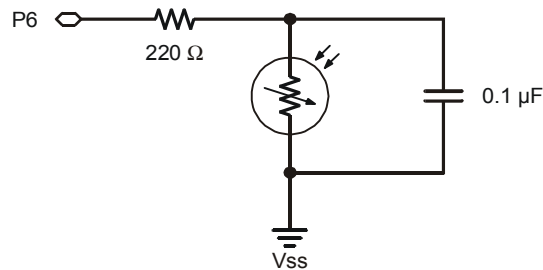


**Figure D-6**  
Resistor, Photoresistor, and  
Capacitor Schematic

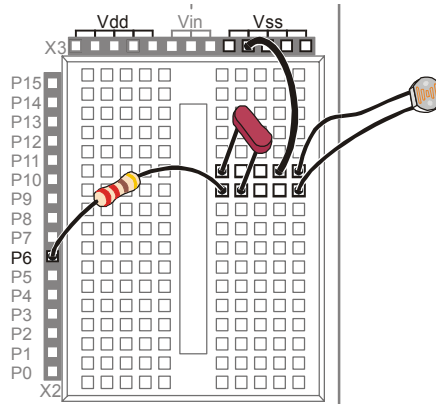


**Figure D-7**  
Resistor, Photoresistor,  
and Capacitor Wiring  
Diagram

Keep in mind that the wiring diagrams presented here as solutions to the schematics are not the **ONLY** solutions to those schematics. For example, Figure D-8 shows another solution to the schematic just discussed. Follow the connections and convince yourself that it does satisfy the schematic.

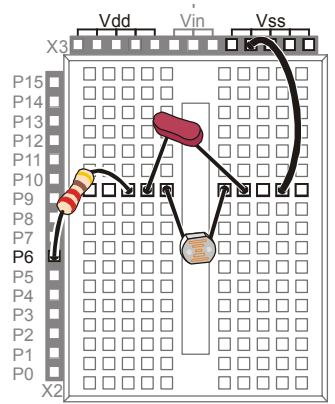


**Figure D-8**  
Resistor, Photoresistor, and  
Capacitor Wiring Diagram

*Note the alternative parts  
placement.*

## Appendix E: Boe-Bot Parts Lists

To complete the activities in this text, you will need a complete Boe-Bot and the electronic components necessary to build the example circuits. There are several options for ordering these items from Parallax, which are described on the following pages.

All of the information in this appendix was current at the time of printing. Parallax may make part substitutions at our discretion, out of necessity or to upgrade the quality of our products. For the latest information and free downloads about your Boe-Bot and the Robotics with the Boe-Bot Student Guide, check their individual product pages at [www.parallax.com](http://www.parallax.com).



### **Boe-Bot Robot Kit (also known as the Boe-Bot Full Kit)**

Aside from a PC with a serial port and a few common household items, the Boe-Bot Robot Kit contains all the parts and documentation you'll need to complete the experiments in this text.

| <b>Table E-1: Boe-Bot Robot (Full) Kit (#28132)</b><br>Parts and quantities subject to change without notice |                                             |                 |
|--------------------------------------------------------------------------------------------------------------|---------------------------------------------|-----------------|
| <b>Parallax Stock Code</b>                                                                                   | <b>Description</b>                          | <b>Quantity</b> |
| BS2-IC                                                                                                       | BASIC Stamp 2 microcontroller module        | 1               |
| 27000                                                                                                        | Parallax CD with software and documentation | 1               |
| 28124                                                                                                        | Robotics with the Boe-Bot Parts Kit         | 1               |
| 28125                                                                                                        | Robotics with the Boe-Bot Student Guide     | 1               |
| 28150                                                                                                        | Board of Education Rev C                    | 1               |
| 700-00064                                                                                                    | Parallax Screwdriver                        | 1               |
| 800-00003                                                                                                    | Serial cable                                | 1               |

All of these items may also be ordered separately, using the individual part numbers. You can contact the Parallax Sales Team toll free at 1-888-512-1024 or order online at [www.parallax.com](http://www.parallax.com). For technical questions or assistance call our Technical Support team at 1-888-99-STAMP.

**Robotics with the Boe Bot Parts Kit**

If you already have a Board of Education and BASIC Stamp, you may purchase the Robotics with the Boe-Bot Parts kit, with or without the printed text *Robotics with the Boe-Bot Student Guide*.

| <b>Table E-2: Robotics with the Boe-Bot Parts &amp; Text, #28154</b><br><b>Robotics with the Boe-Bot Parts only, #28124</b><br>Parts and quantities subject to change without notice |                                                      |                 |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------|-----------------|
| <b>Parallax Stock Code</b>                                                                                                                                                           | <b>Description</b>                                   | <b>Quantity</b> |
| 150-01020                                                                                                                                                                            | 1 k $\Omega$ resistor                                | 2               |
| 150-01030                                                                                                                                                                            | 10 k $\Omega$ resistor                               | 2               |
| 150-02020                                                                                                                                                                            | 2 k $\Omega$ resistor                                | 2               |
| 150-02210                                                                                                                                                                            | 220 $\Omega$ resistor                                | 8               |
| 150-04710                                                                                                                                                                            | 470 $\Omega$ resistor                                | 4               |
| 150-04720                                                                                                                                                                            | 4.7 k $\Omega$ resistor                              | 2               |
| 200-01031                                                                                                                                                                            | 0.01 $\mu$ F capacitor                               | 2               |
| 200-01040                                                                                                                                                                            | 0.1 $\mu$ F capacitor                                | 2               |
| 350-00003                                                                                                                                                                            | Infrared LED                                         | 2               |
| 350-00006                                                                                                                                                                            | Red LED                                              | 2               |
| 350-00009                                                                                                                                                                            | Photoresistors (EG&G Vactec VT935G group B)          | 2               |
| 350-00014                                                                                                                                                                            | Infrared receiver (Panasonic PNA4602M or equivalent) | 2               |
| 350-90000                                                                                                                                                                            | LED standoff for infrared LED                        | 2               |
| 350-00001                                                                                                                                                                            | LED light shield for infrared LED                    | 2               |
| 451-00303                                                                                                                                                                            | 3-Pin Header                                         | 2               |
| 700-00056                                                                                                                                                                            | Whisker wire                                         | 2               |
| 700-00015                                                                                                                                                                            | #4 screw-size nylon washer                           | 2               |
| 710-00007                                                                                                                                                                            | 7/8" 4-40 pan-head screw, Phillips                   | 2               |
| 713-00007                                                                                                                                                                            | 1/2" Spacer, aluminum, #4 round                      | 2               |
| 800-00016                                                                                                                                                                            | Jumper wires (bag of 10)                             | 2               |
| 900-00001                                                                                                                                                                            | Piezospoker                                          | 1               |
| 28133                                                                                                                                                                                | Boe-Bot Hardware Pack                                | 1               |

**Building a Boe-Bot with a HomeWork Board**

If you already have a BASIC Stamp HomeWork Board that you wish to use with a Boe Bot, you will need the Robotics with the Boe-Bot Parts kit **and these additional items:**

**(2) 3-pin male/male headers, #451-00303**

**(1) Tinned-lead battery pack, #753-00001**

**Boe-Bot Hardware Pack**

All of the Boe-Bot hardware parts can be purchased individually, as found in our on-line Robot Component Shop if you find you need a replacement part. Please note that the Hardware Pack is not sold as a unit separately from the Boe-Bot Robot (Full) Kit or the Boe-Bot Parts Kit.

**E****Table E-3: Boe-Bot Hardware Pack (#28133)**

Parts and quantities subject to change without notice

| Parallax Stock Code | Description                                  | Quantity |
|---------------------|----------------------------------------------|----------|
| 700-00002           | 4-40 x 3/8" machine screw, Phillips          | 8        |
| 700-00003           | Hex nut, 4-40 zinc plated                    | 10       |
| 700-00009           | Tail wheel ball                              | 1        |
| 700-00016           | 4-40 x 3/8" flathead machine screw, Phillips | 2        |
| 700-00022           | Boe-Bot aluminum chassis                     | 1        |
| 700-00023           | 1/16" x 1.5" long cotter pin                 | 1        |
| 700-00025           | 13/32" rubber grommet                        | 2        |
| 700-00028           | 4-40 x 1/4" machine screw, Phillips          | 8        |
| 700-00038           | Battery holder with cable and barrel plug    | 1        |
| 700-00060           | Standoff, threaded aluminum, round 4-40      | 4        |
| 721-00001           | Parallax plastic wheel                       | 2        |
| 721-00002           | Rubber band tire                             | 4        |
| 900-00008           | Parallax Continuous Rotation Servo           | 2        |

**Board of Education Kits**

Almost all of the titles in the Stamps in Class curriculum feature different hardware and component packages that depend on the BASIC Stamp and Board of Education as a core. The Board of Education can be purchased separately or in its own kit, as listed in Table E-4 below.

| <b>Table E-4: Board of Education – Full Kit (#28102)</b><br>Parts and quantities subject to change without notice |                                      |                 |
|-------------------------------------------------------------------------------------------------------------------|--------------------------------------|-----------------|
| <b>Parallax<br/>Stock Code</b>                                                                                    | <b>Description</b>                   | <b>Quantity</b> |
| 28150                                                                                                             | Board of Education Rev C             | 1               |
| 800-00016                                                                                                         | Jumper wires – pack of 10            | 1               |
| BS2-IC                                                                                                            | BASIC Stamp 2 microcontroller module | 1               |
| 750-00008                                                                                                         | 300 mA 9 VDC power supply            | 1               |
| 800-00003                                                                                                         | Serial cable                         | 1               |



## Appendix F: Balancing Photoresistors

In this appendix, you will test the photoresistors to find out if they respond similarly to the same incident light levels. If the measurements they report are different for the same incident light levels, you can modify your programs to scale the values reported by your photoresistors. The values will then be similar for similar incident light levels, which can help the Boe-Bot recognize different incident light levels more reliably. This technique can in turn assist the Boe-Bot in exiting dark rooms, even with mismatched photoresistor circuits.



**RC circuits with photoresistors can report different values for the same light level** for many reasons: The stated value of the capacitors is 0.01  $\mu\text{F}$ , but the actual value of capacitors can be very different. Many common ceramic capacitors are rated with a tolerance of +80/-20%, meaning that the actual value of the capacitor could be up to 80% larger or 20% smaller than 0.01  $\mu\text{F}$ . This means that your measured decay time could also be between 80% larger and 20% smaller. The photoresistors themselves can also behave differently if they come from different manufacturing batches or if they have smudged or chipped light collecting surfaces.



### Testing for Well Matched Photoresistor Circuits

This next example program displays the decay time of both photoresistors in the Debug Terminal. It makes it easy to judge the differences between the two readings for similar light levels.



**For best results, eliminate sources of direct sunlight:** In general, uniform lighting conditions improve the Boe-Bot's performance with photoresistors. Draw the blinds to eliminate sources of direct sunlight. Rooms with distributed light sources such as fluorescent lights or ceiling lamps work well.

### Example Program: TestPhotoresistors.bs2

- ✓ Enter, save, and run TestPhotoresistors.bs2.
- ✓ Cast a shadow over the Boe-Bot's photoresistors with a white sheet of paper. Find a level of shade that gives you readings between 20 and 100.
- ✓ Record the values of both time measurements in the first row of Table F-1.

- ✓ Cup your hand over the photoresistors, making sure that you are casting equal shade over both. For best results, the measurements should be in the 200 to 400 range.
- ✓ Record the values of both time measurements in the second row of Table F-1.

| Table F-1: RC-Time Measurements in Ambient and Low Light |           |                                         |
|----------------------------------------------------------|-----------|-----------------------------------------|
| Duration Values                                          |           | Description                             |
| timeLeft                                                 | timeRight |                                         |
|                                                          |           | Photoresistors in uniform ambient light |
|                                                          |           | Photoresistors in uniform low light     |

```
' Robotics with the Boe-Bot - TestPhotoresistors.bs2
' Test Boe-Bot photoresistor circuits.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

timeLeft VAR Word ' Variable declarations.
timeRight VAR Word

DEBUG "PHOTORESISTOR VALUES", CR, ' Initialization.
 "timeLeft timeRight", CR,
 "----- -----"

DO ' Main routine.

 HIGH 6 ' Left RC time measurement.
 PAUSE 3
 RCTIME 6,1,timeLeft

 HIGH 3 ' Right RC time measurement.
 PAUSE 3
 RCTIME 3,1,timeRight

 DEBUG CR$RXY, 0, 3, ' Display measurements.
 DEC5 timeLeft,
 " ",
 DEC5 timeRight

 PAUSE 200

LOOP
```

**Calibrating Using a Linear Approximation**

The photoresistor is often referred to as a non-linear device. In other words, if it returns one measurement at one brightness, that doesn't mean that the measurement will be five times as large when the light is five times as bright. The math is more complicated and involves logarithms. However, in cases where the measurements are confined over a narrow range of the sensors overall detection abilities, the sensor can be treated like it's a linear device. You can take a couple of measurements, and then figure out how the device will react if other measurements in its range could be plotted in a straight line. The technique is called linear approximation.

Another thing you can do with linear devices is assume the difference between the two lines can also be plotted as a line. In fact, if you have one linear device that has larger measurements than the other for ambient and low light, you can use a linear approximation for making the sensors return approximately the same values for the same light levels. For every reading from one sensor, (we'll call that one x), you can multiply it by a scale factor (m), and add it to a constant (b) to get a value in the same range the other sensor would report (y).

$y = mx + b$

Here is an example of how to get the values of m and b to match the left photoresistor circuit to the right. First, assign X<sub>1</sub> and X<sub>2</sub> to the left photoresistor values and Y<sub>1</sub> and Y<sub>2</sub> to the right photoresistor values. (Table F-2) shows some example values for mismatched photoresistors. Your values will be different.

| Table F-2: RC-Time Measurements in Ambient and Low Light |                      |                                         |
|----------------------------------------------------------|----------------------|-----------------------------------------|
| Duration Values                                          |                      | Description                             |
| timeLeft                                                 | timeRight            |                                         |
| X <sub>1</sub> = 36                                      | Y <sub>1</sub> = 56  | Photoresistors in uniform ambient light |
| X <sub>2</sub> = 152                                     | Y <sub>2</sub> = 215 | Photoresistors in uniform low light     |

Now, solve for m and b using two equations in two unknowns. One of the simpler approaches is to write two y = mx + b equations, one in terms of X<sub>1</sub> and Y<sub>1</sub> and the other in terms of X<sub>2</sub> and Y<sub>2</sub>. Then, subtract one from the other to eliminate b. Then, solve for m.



$$\begin{array}{r}
 y_2 = mx_2 + b \\
 - (y_1 = mx_1 + b) \\
 \hline
 (y_2 - y_1) = m(x_2 - x_1) \\
 \\
 m = \frac{(y_2 - y_1)}{(x_2 - x_1)}
 \end{array}$$

Once you've solved for m, you can plug m back into either of the two  $y = mx + b$  equations you started with to get b.

$$\begin{array}{l}
 y_2 = mx_2 + b \\
 b = y_2 - mx_2
 \end{array}$$

So, the two equations for solving for m and b turn out to be:

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \quad \text{and} \quad b = y_2 - mx_2$$

Let's plug our sample values from Table F-2 into the equations and see what the scale factor (m) and offset constant (b) will be for the left photoresistor. First, calculate m:

$$\begin{array}{l}
 m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \\
 m = \frac{(215 - 56)}{(152 - 36)} \\
 m = \frac{159}{116} = 1.37
 \end{array}$$

Then, use m,  $y_2$ , and  $x_2$  to calculate b:

$$\begin{array}{l}
 b = 215 - (1.37 \times 152) \\
 b = 6.76 \\
 b \approx 7
 \end{array}$$

Now, we know how to correct the `timeLeft` variable so that it reports values similar to the `timeRight` variable in this narrow range of light levels:

$$y = mx + b$$

$$y = 1.37x + 7$$

$$timeLeft_{(adjusted)} = 1.37 \times timeLeft + 7$$

### **A Linear Equation in PBASIC**

In most programming languages for PCs, this equation could be entered as-is. The BASIC Stamp is a very tiny processor compared to a PC. Because of this, it takes an extra step to multiply by a fractional value. You have to use the `*/` operator (it's called the "star-slash" operator). For the `timeLeft` equation, the PBASIC code to adjust the `timeLeft` variable can be done like this:

```
timeLeft = (timeLeft */ 351) + 7
```

The adjusted value of `timeLeft` after this line of code is executed is 1.37 times the old `timeLeft`, plus 7.



Why did 1.37 become 351? The way the `*/` operator works is that you have multiply your fractional value by 256, and place it to the right of the `*/` operator. Since  $1.37 \times 256 = 350.72 \approx 351$ , the value 351 goes to the right of the `*/` operator.

You can find out more about the `*/` operator in the BASIC Stamp Editor by clicking Help and selecting Index. Type in `*/` in the field labeled "Type in keyword to find". You can also look up `*/` in the Binary operators section of the *BASIC Stamp Manual*. □

### **Your Turn – Balance Your Photoresistors with m and b**

- ✓ In Table F-1, label the first `timeLeft` entry X1 and the second `timeLeft` entry X2.
- ✓ Label the first `timeRight` entry Y1 and the second `timeRight` entry Y2.
- ✓ Use these equations and your X1, X2, Y1, and Y2 values to solve for m and b.
- ✓

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \quad \text{and} \quad b = y_2 - mx_2$$

- ✓ Calculate the values of m you will use with the `*/` operator by multiplying m by 256.
- ✓ Substitute your value of m and b in this line of code from `BalancePhotoresistors.bs2`:

```
timeLeft = (timeLeft */ 351) + 7
```

- ✓ Enter, save, and run your adjusted version of `BalancePhotoresistors.bs2`.
- ✓ Expose both photoresistors to the same light level.
- ✓ Verify that the “after” values are similar and corrected for differences in the “before” values.
- ✓ Choose a different light level and again, expose both photoresistors to it.
- ✓ Check the “after” values again for similarity.
- ✓ When you have determined your values for m and b, you can modify `RoamingTowardTheLight.bs2` by uncommenting the equation between `GOSUB Test_Photoresistors` and `GOSUB Average_And_Difference`. Your m value will replace 351 and your b value will replace 7.

```
' Robotics with the Boe-Bot - BalancePhotoresistors.bs2
' Test adjustments to Boe-Bot photoresistor circuits.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

timeLeft VAR Word ' Variable declarations.
timeRight VAR Word

DEBUG "PHOTORESISTOR VALUES", CR, ' Initialization.
 "timeLeft timeRight", CR,
 "----- -----"

DO ' Main routine.

 HIGH 6 ' Left RC time measurement.
 PAUSE 3
 RCTIME 6,1,timeLeft

 HIGH 3 ' Right RC time measurement.
 PAUSE 3
 RCTIME 3,1,timeRight

DEBUG CRSRXY, 0, 3, ' Display measurements.
 DEC5 timeLeft,
 " ",
```

```
 DEC5 timeRight,
 " Before"

 timeLeft = (timeLeft */ 351) + 7

 DEBUG CRSRXY, 0, 5, ' Display measurements.
 DEC5 timeLeft,
 " ",
 DEC5 timeRight,
 " After"

 PAUSE 200
LOOP
```







## Appendix G: Tuning IR Distance Detection

### Finding the Right Frequency Sweep Values

Fine tuning the Boe-Bot's distance detection involves determining which frequency is most reliable for each zone for each IR pair.



**Note:** This appendix features a method of determining the best frequencies for determining given distances using spreadsheets. This activity takes time and patience, and is only recommended if your distance sensing is severely out of calibration. It involves collecting frequency sweep data and using it to determine the most reliable values for detecting particular distances. □

- ✓ Point both the IR LEDs and detectors straight forward.
- ✓ Place the Boe-Bot in front of a wall with a white sheet of paper as the IR target.
- ✓ Place the Boe-Bot so that its IR LEDs are 2.5 cm away from the paper target. Make sure the front of the Boe-Bot is facing the paper target. Both IR LEDs and detectors should be pointed directly at the paper.

G

### IR Fine Tuning Program

FrequencySweep.bs2 performs a frequency sweep on the IR detector and displays the data. Although the techniques used are similar to other programs, it has one unique feature. The BASIC Stamp is programmed to wait for you to press the Enter key.

- ✓ Enter and run FrequencySweep.bs2, but do not disconnect the Boe-Bot from the serial cable.

```
' -----[Title]-----
' Robotics with the Boe-Bot - FrequencySweep.bs2
' Test IR LED/detector response to frequency sweep.

' {$STAMP BS2} ' Stamp directive.
' {$PBASIC 2.5} ' PBASIC directive.

' -----[Variables]-----

crsrPosRow VAR Byte
irFrequency VAR Word
irDetect VAR Bit
distance VAR Nib
dummy VAR crsrPosRow
```

```

' -----[Initialization]-----
DEBUG CLS,
 "Click transmit windowpane,", CR,
 "then press enter to begin", CR,
 "frequency sweep...", CR, CR,

 " OBJECT", CR,
 "FREQUENCY DETECTED", CR,
 "-----", CR

' -----[Main Routine]-----

DO

 DEBUGIN dummy

 crsrPosRow = 6

 FOR irFrequency = 30500 TO 46500 STEP 1000

 crsrPosRow = crsrPosRow + 1

 FREQOUT 8,1, irFrequency
 irDetect = IN9

 DEBUG CRSRXY, 4, crsrPosRow, DEC5 irFrequency
 DEBUG CRSRXY, 11, crsrPosRow

 IF (irDetect = 0) THEN DEBUG "Yes" ELSE DEBUG "No "

 PAUSE 100

 NEXT

LOOP

```

- ✓ Click the upper of the two window panes shown in Figure G-1.
- ✓ Press the Enter key. The frequency response data will appear as shown in the figure.

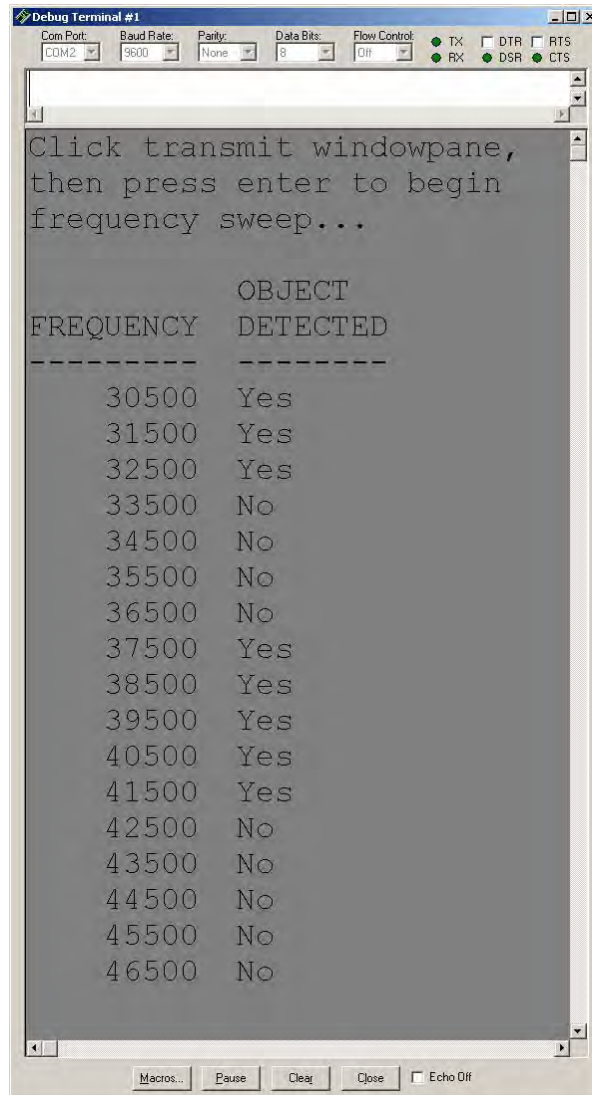


**Figure G-1**  
Debug of  
Frequency  
Data

G

The BASIC Stamp has been programmed to make the Debug Terminal display a “Yes” if an object was detected and a “No” if an object was not detected. Figure G-1 shows that the left sensor’s region of good signal response is between 36500 and 42500.

- √ Modify the **FOR...NEXT** loop in Program Listing FrequencySweep.bs2 so that it steps in increments of 250 and includes the upper and lower limits of both detectors. Based on the data in the example shown in Figure G-1, the start, end, and step values of the **FOR...NEXT** loop would be modified as follows:

```
FOR irFrequency = 36500 to 42500 STEP 250
```

- √ Re-run your modified FrequencySweep.bs2, and press Enter again.
- √ Record the data for left and right sides in separate spreadsheets.
- √ Press the Enter key again and record the next set of data points.
- √ Repeat this process three more times. When finished, you will have five sets of data points for each sensor in separate spreadsheets for this one frequency.
- √ Back the Boe-Bot up 2.5 cm. Now your Boe-Bot's IR detectors will be 5 cm from the paper target.
- √ Record five more data sets at this distance.
- √ Keep on backing up the Boe-Bot by 2.5 cm at a time and recording the five frequency sweep data sets between each distance adjustment.
- √ When the Boe-Bot has been backed up by 20 cm, the frequency sweep will display mostly, if not all "No" regions. When the frequency sweep is all "No", it means no object is detected at any frequency within the sweep.

By careful scrutiny of the spreadsheets and process of elimination, you can determine the optimum frequency for each IR pair for each zone. Customizing for up to eight zones can be done without any restructuring of the Boe-Bot navigational routines. If you were to customize for 15 zones, this would entail 30 one millisecond **FREQOUT** commands. That won't gracefully fit between servo pulses. One solution would be to take 15 measurements every other pulse.

How to determine the best frequencies for the left sensor is discussed here. Keep in mind you'll have to repeat this process for the right sensor. This example assumes you are looking for six zones (zero through five).

- √ Start by examining the data points taken when the Boe-Bot was furthest from the paper target. There probably won't be any sets of data points that are all "Yes" readings at the same frequency. Check the data points for the next 2.5 cm towards the paper target. Presumably, you will see a set of four or five "Yes"

readings at a particular frequency. Note this frequency as a reliable measurement for the dividing line between Zone 0 and Zone 1.

- ✓ At each of the remaining five distances, find a frequency for which the output values have just become stable.

For example, at 15 cm, three different frequencies might show five "Yes" readings. If you look back to the 17.5 cm mark, two of these frequencies were stable, but the other was not. Take the frequency that was not stable at 17.5 cm but was stable at 15 cm as your most reliable frequency for this distance. Now, this example has determined the frequencies that can be used to separate Zones 5 and 4 and Zones 4 and 3. Repeat this process for the remaining zone partitions.

### **Your Turn**

- ✓ If you succeeded in fine tuning five measurements and time permits, try increasing the resolution to eight measurements. Save your data for both methods.





## Appendix H: Boe-Bot Navigation Contests

---

If you're planning a competition for autonomous robots, these rules are provided courtesy of Seattle Robotics Society.

### CONTEST#1: ROBOT FLOOR EXERCISE

#### Purpose

The floor exercise competition is intended to give robot inventors an opportunity to show off their robots or other technical contraptions.

#### Rules

The rules for this competition are quite simple. A 10-foot-by-10-foot flat area is identified, preferably with some physical boundary. Each contestant will be given a maximum of five minutes in this area to show off what their robot can do. The robot's contestant can talk through the various capabilities and features of the robot. As always, any robot that could damage the area or pose a danger to the public will not be allowed. Robots need not be autonomous, but it is encouraged. Judging will be determined by the audience, either indicated by clapping (the loudest determined by the judge), or some other voting mechanism.



### CONTEST#2: LINE FOLLOWING

#### Objective

To build an autonomous robot that begins in Area "A" (at position "S"), travels to Area "B" (completely via the line), then travels to the Area "C" (completely via the line), then returns to the Area "A" (at position "F"). The robot that does this in the least amount of time (including bonuses) wins. The robot must enter areas "B" and "C" to qualify. The exact layout of the course will not be known until contest day, but it will have the three areas previously described.

#### Skills Tested

The ability to recognize a navigational aid (the line) and use it to reach the goal.

### **Maximum Time to Complete Course**

Four minutes.

### **Example Course**

All measurements in the example course are approximate. There is a solid line dividing Area "A" from Area "T" at position "F." This indicates where the course ends. The line is black, approximately 2.25 inches wide and spaced approximately two feet from the walls. All curves have a radius of at least one foot and at most three feet. The walls are 3 1/2 inches high and surround the course. The floor is white and made of either paper or Dupont Tyvek®. Tyvek is a strong plastic used in mailing envelopes and house construction.

Positions "S" and "F" are merely for illustration and are not precise locations. A Competitor may place the robot anywhere in Area "A," facing in any direction when starting. The robot must be completely within Area "A." Areas "A," "B" and "C" are not colored red on the actual course.

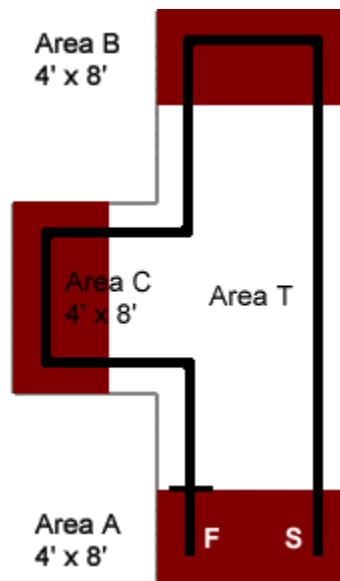


**Figure H-1**  
Sample Contest  
Course



### **Scoring**

Each contestant's score is calculated by taking the time needed to complete the course (in seconds) minus 10% for each "accomplishment." The contestant with the lowest score wins.

| <b>Table H-1: Line Following Scoring</b> |                         |
|------------------------------------------|-------------------------|
| <b>Accomplished</b>                      | <b>Percent Deducted</b> |
| Stops in area A after reaching B and C   | 10%                     |
| Does not touch any walls                 | 10%                     |
| Starts on command                        | 10%                     |

("Starts on command" means the robot starts with an external, non-tactile command. This could, for example, be a sound or light command.)



## **CONTEST#3: MAZE FOLLOWING**

### **Purpose**

The grand maze is intended to present a test of navigational skills by an autonomous robot. The scoring is done in such a way as to favor robots which are either brutally fast or which can learn the maze after one pass. The object is for a robot, which is set down at the entrance of the maze, to find its way through the maze and reach the exit in the least amount of time.

### **Physical Characteristics**

The maze is constructed of 3/4" shop-grade plywood. The walls are approximately 24 inches high, and are painted in primary colors with glossy paint. The walls are set on a grid with 24-inch spacing. Due to the thickness of the plywood and limitations in accuracy, the hallways may be as narrow as 22 inches. The maze can be up to 20-feet square, but may be smaller, depending on the space available for the event.

The maze will be set up on either industrial-type carpet or hard floor (depending on where the event is held). The maze will be under cover, so your robot does not have to be rain proof; however, it may be exposed to various temperatures, wind, and lighting

conditions. The maze is a classical two-dimensional proper maze: there is a single path from the start to the finish and there are no islands in the maze. Both the entrance and exit are located on outside walls. Proper mazes can be solved by following either the left wall or the right wall. The maze is carefully designed so that there is no advantage if you follow the left wall or the right wall.

### **Robot Limitations**

The main limit on the robot is that it be autonomous: once started by the owner or handler, no interaction is allowed until the robot emerges from the exit, or it becomes hopelessly stuck. Obviously the robot needs to be small enough to fit within the walls of the maze. It may touch the walls, but may not move the walls to its advantage -no bulldozers. The judges may disqualify a robot which appears to be moving the walls excessively. The robot must not damage either the walls of the maze, nor the floor. Any form of power is allowed as long as local laws do not require hearing protection in its presence or place any other limitations on it.

### **Scoring**

Each robot is to be run through the maze three times. The robot with the lowest single time is the winner. The maximum time allowed per run is 10 minutes. If a robot cannot finish in that amount of time, the run is stopped and the robot receives a time of 10 minutes. If no robot succeeds in finding the exit of the maze, the one that made it the farthest will be declared the winner, as determined by the contest's judge.

### **Logistics**

Each robot will make one run, proceeding until all robots have attempted the maze. Each robot then does a second run through the maze, then the robots all do the third run. The judge will allow some discretion if a contestant must delay their run due to technical difficulties. A robot may remember what it found on a previous run to try to improve its time (mapping the maze on the first run), and can use this information in subsequent runs-as long as the robot does this itself. It is not allowed to manually "configure" the robot through hardware or software as to the layout of the maze.

# Index

---

- \* -
- \*/, 325
- < -
- ◇, 186
- ... -
- ..., 52
- 3 -
- 3-position switch, 16
- 3-position switch, 35
- 9 -
- 90° turns, 132
- A -
- alarm circuit, 107
- American Standard Code for  
Information Interchange, 33, 151
- amps, 49
- anode, 46
- artificial intelligence, 182
- ASCII, 33, 151
- B -
- backwards motion, 127
- ballast, 242
- band pass frequency, 236
- Basic Analog and Digital, 56
- BASIC Stamp
  - components, 305
  - insertion, 17
  - low power mode, 28
  - preventing damage, 36
- BASIC Stamp Editor
  - Identification window, 22
  - Identify, 302, 303, 304
  - installation, 10
  - Software, 4
  - Trouble-Shooting, 301
- BASIC Stamp Editor's Help, 30
- BASIC Stamp HomeWork Board, 3
- BASIC Stamp HomeWork Board, 4
- BASIC Stamp HomeWork Board
  - connecting power, 20
- BASIC Stamp HomeWork Board
  - disconnect power, 36
- BASIC Stamp HomeWork Board
  - components, 307
- BASIC Stamp Manual, 32
- batteries, 60
- battery pack, 96
- battery pack with tinned leads, 63
- BIN1, 172
- binary numbers, 22
- Bit, 71
- block diagram, 277
- Board of Education, 3, 4
  - components, 306
  - servo header, 60
- Board of Education Rev A, 59
- Board of Education Rev A or B

- disconnect power, 36
- Board of Education Rev B, 59
  - components, 308
- Board of Education Rev C
  - connecting power, 16
  - disconnect power, 35
- breadboard. See prototyping area
- brownout, 105, 109
- brownout detector, 105
- Byte, 71
  - C -
- Cadmium Sulfide, 194
- capacitor, 205
  - part drawing, 206
  - schematic symbol, 206
- carriage return, 28
- carrier board, 3
- cathode, 46
- CdS, 194
- centering the servos, 67
- chassis, 92
- closed loop control, 277
- code block, 140
- color code, 309
- COM port, 301
- COM Port, 13
- command, 28
- comment, 27
- Compiler directives, 24
- components
  - BASIC Stamp, 305
  - BASIC Stamp HomeWork Board, 307
  - Board of Education, 306

- Board of Education Rev B, 308
- computer system requirements, 5
- CON, 213
- condensed EEPROM Map, 147
- constants, 213
- control character
  - CR, 28
- control system, 277
- cotter pin, 97
- CR, 28
- CRSRXY, 173
- crystal, 107
- current, 45, 49
  - D -
- DATA, 148
  - Word modifier, 153
- DATA directive, 152
- data storage, 146
- DC interference, 236
- DC power supply, 301
- dead reckoning, 132
- DEBUG, 28
- DEBUG formatters
  - ?, 73
  - BIN, 172
  - DEC, 28
  - SDEC, 73
- Debug Terminal, 26
- DEBUGIN, 112
- DEC, 28
- declare, 72, 213
- decrement, 138
- derivative control, 277

- Detailed EEPROM Map, 151
- disconnect power, 35
- distance calculation, 133
- DO WHILE, 148
- DO...LOOP, 44
- Download Progress window, 26
- Duration argument, 54, 109
  - maximum value, 54
- E -
- EEPROM, 146
- electrical tape, 255, 286
- electrically erasable programmable read
  - only memory, 146
- electromagnetic radiation, 235
- electronic filter, 236
- ELSE, 178
- ELSEIF, 178
- END, 28
- ENDIF, 178
- EndValue, 74
- F -
- F, 205
- farad, 205
- feedback, 279
- filter sensitivity, 270
- flashlight, 210
- fluorescent light, 242
- fluorescent light interference, 287
- fluorescent lights, 237
- foot-candle, 194
- FOR...NEXT, 74
  - counting backward, 75
  - decrement, 75
  - EndValue, 74
  - StartValue, 74
  - STEP StepValue, 75
- forward motion, 124
- Freq1 argument, 109
- FREQOUT, 109
  - Duration argument, 109
  - Freq1 argument, 109
  - Pin argument, 109
- frequency, 107
- frequency sweep, 270
- fundamental frequency, 240
- G -
- GOSUB, 141
- Guarantee, 2
- H -
- hardware adjustment, 129
- harmonic frequency, 240
- hertz, 109
- hexadecimal, 151
- HIGH, 50
  - PIN argument, 50
- hysteresis, 277
- I -
- I/O pins
  - as inputs or outputs, 196
  - default to input, 171
- Identification window, 22, 301
- Identify, 302, 303, 304
- IF...THEN, 178
  - nesting statements, 182
- illuminance, 193, 194
- incident light, 194

Index argument, 271  
*Industrial Control*, 277  
infrared detector, 237  
infrared interference, 242  
infrared led, 237  
infrared spectrum, 235  
initialize, 72  
input register, 172  
integral control, 277  
IR interference, 236  
    - J -  
jumper, 60  
    - K -  
kilohertz, 109  
Kp, 278  
    - L -  
label  
    subroutine, 141  
LDR, 194  
lead vehicle, 277  
LED, 46  
LED light shield assembly, 237  
light dependent resistor, 193  
light emitting diode, 46  
    anode, 46  
    cathode, 46  
    schematic symbol, 46  
    terminals, 46  
linear approximation, 323  
logic threshold, 197  
LOOKUP, 271  
    Index argument, 271

ValueN argument, 271  
Variable argument, 271  
LOW, 50  
    PIN argument, 50  
low power mode, 28  
lux, 194  
    - M -  
math order of operations, 214, 280  
measuring distance, 133  
Memory Map, 147  
microfarad, 205, 206  
milliamps, 49  
millisecond, 42  
    - N -  
nanofarad, 206  
negative numbers, 73  
Nib, 71  
nodes, 208  
null modem cable, 301  
nylon washer, 167  
    - O -  
ohm, 46  
omega, 46  
operator, 72  
operator block, 278  
output adjust, 278  
    - P -  
Parallax Continuous Rotation servos, 41  
part drawing  
    capacitor, 206  
    LED, 46  
    photoresistor, 193

- piezoelectric speaker, 106
- resistor, 46
- PAUSE, 42
  - Duration argument, 42
- PBASIC, 1
  - variables, 71
- PBASIC commands
  - DEBUG, 28
  - DEBUGIN, 112
  - DO WHILE, 148
  - DO...LOOP, 44
  - ELSE, 178
  - ELSEIF, 178
  - END, 28
  - ENDIF, 178
  - FOR...NEXT, 74
  - FREQOUT, 109
  - GOSUB, 141
  - HIGH, 50
  - IF...THEN, 178
  - LOOKUP, 271
  - LOW, 50
  - PAUSE, 42
  - PULSOUT, 54
  - RCTIME, 209
  - READ, 148
  - RETURN, 140
  - SELECT...CASE...ENDSELECT, 149
  - STOP, 248
- PBASIC directive, 28
- PBASIC directives
  - CON, 213
  - DATA, 148
  - PBASIC, 28
  - Stamp, 28
- PBASIC operators
  - \*/, 325
  - <>, 186
- photoresistor, 193, 194
  - calibration, 323
  - part drawing, 193
  - schematic symbol, 193
- photoresistor voltage divider
  - troubleshooting, 199
- picofarad, 206
- piezoelectric crystal, 107
- piezoelectric element, 107
- piezoelectric speaker, 106
  - part drawing, 106
  - schematic symbol, 106
- piezospeaker, 106
  - alarm circuit, 107
- Pin argument, 50, 109
- pivoting motion, 128
- plastic wheel, 98
- poster board, 255
- potentiometer, 70
- program storage, 146

programs

- saving, 26, 27

proportional constant, 278

proportional control, 277

prototyping area

- input/output pins, 311

prototyping areas

- socket, 311

PULSOUT, 54

- Duration argument, 54

- R -

RADAR, 235

RAM, 146

ramping, 137

random access memory, 146

RC decay time, 208

RCTIME, 209

- Duration argument, 209

- Pin argument, 209

- State argument, 209

READ, 148

Reset button, 28

Reset button, 26

resistor, 45

- color code, 309

- leads, 46

- light dependent resistor, 193

- series resistors, 197

- tolerance, 309

- voltage divider, 197

RETURN, 140

rotational velocity, 115

RPM, 115

rubber band tire, 97

rubber grommet, 92, 96

- S -

saving programs, 26, 27

schematic symbol

- capacitor, 206

- LED, 46

- photoresistor, 193

- piezoelectric speaker, 106

- resistor, 46

screwdriver, 67, 91

screws, 7/8", 167

SDEC, 73

second, 42

SELECT...CASE...ENDSELECT, 149

serial cable, 13

servo

- header, 60

- output shafts, 98

servos

- avoiding damage, 60, 69

- labeling, 94

- troubleshooting, 104

shadow vehicle, 277

SODAR, 235

software adjustment, 129

SONAR, 235

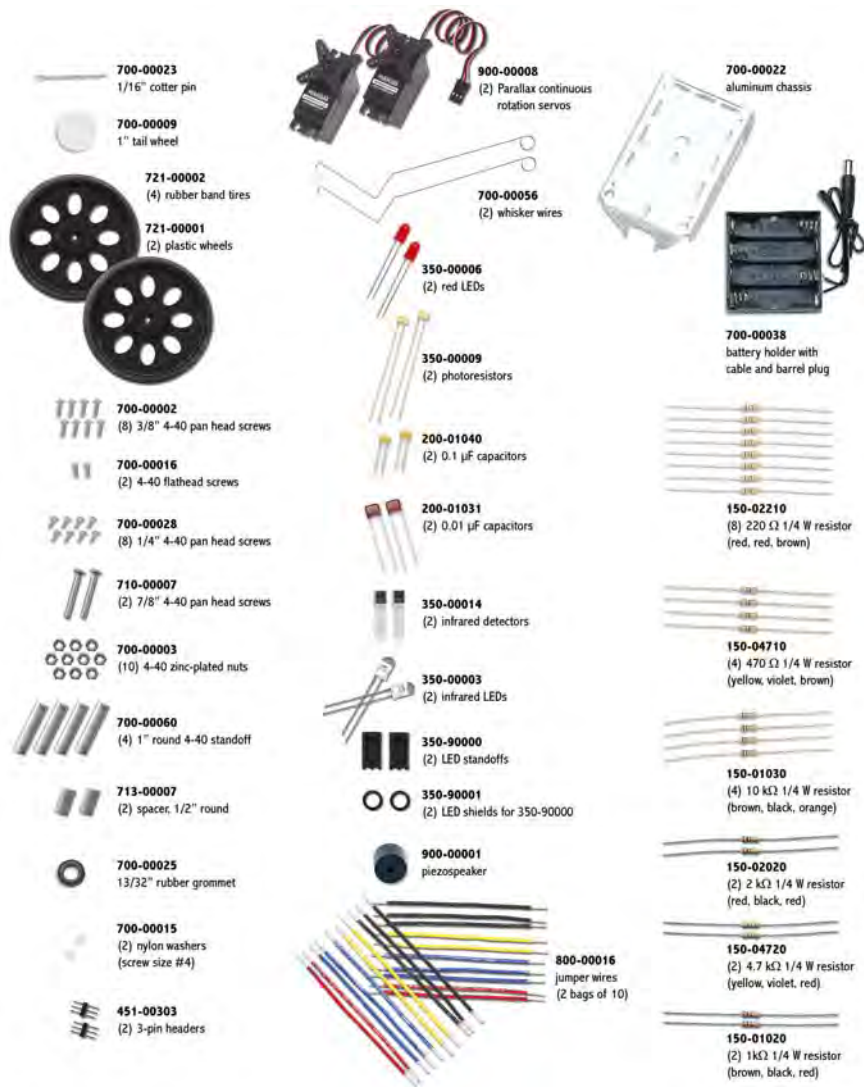
spacer, 167

Stamp Directive, 28



standoffs, 92, 100, 167  
 start/reset indicator, 106  
 StartValue, 74  
 STEP *StepValue*, 75  
*stepValue*, 75  
 STOP, 248  
 straightening the trajectory, 130  
 subroutine call, 140, 141  
 subroutine label, 141  
 subroutines, 140  
 summing junction, 278  
 - T -  
 tactile switches, 165  
 tail wheel, 97  
 threshold voltage, 197  
 timing diagram, 52, 56  
 tokens, 146  
 tolerance, 309  
 tones, 106  
 tools required, 91  
 transfer curve, 115  
 Transmit windowpane, 111  
 troubleshooting  
     BASIC Stamp to PC communication, 301  
     electrical tape course, 289  
     IR detectors, 241  
     photoresistor voltage divider, 199  
     servos, 103, 104, 105  
 Tyvek, 336

- U -  
 US232B, 14  
 USB to Serial Adapter, 13, 14  
 USB to Serial Adaptor, 5  
 - V -  
 VAR, 71  
 variable, 71  
     declare, 72  
     default value, 72  
     initialize, 72  
     VAR, 71  
 variable sizes, 71  
 Vbp, 64  
 Vdd, 49, 311  
 Vin, 311  
 voltage, 49  
 voltage divider, 197  
 Vss, 311  
 - W -  
 What's a Microcontroller? Student  
     Guide, 2  
 whisker wires, 167  
 Word, 71  
 Word modifier, 153  
 - M -  
 $\mu$ F, 205



Parts and quantities in the various Boe-Bot Robot kits are subject to change without notice. Parts may differ from what is shown in this picture. Please contact [stampsinclsss@parallax.com](mailto:stampsinclsss@parallax.com) if you have any questions about your kit.

***BASIC Stamp Syntax and Reference Manual***  
**Version 2.2**

## Warranty

Parallax Inc. warrants its products against defects in materials and workmanship for a period of 90 days from receipt of product. If you discover a defect, Parallax Inc. will, at its option, repair or replace the merchandise, or refund the purchase price. Before returning the product to Parallax, call for a Return Merchandise Authorization (RMA) number. Write the RMA number on the outside of the box used to return the merchandise to Parallax. Please enclose the following along with the returned merchandise: your name, telephone number, shipping address, and a description of the problem. Parallax will return your product or its replacement using the same shipping method used to ship the product to Parallax.

## 14-Day Money-Back Guarantee

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a full refund. Parallax Inc. will refund the purchase price of the product, excluding shipping/handling costs. This guarantee is void if the product has been altered or damaged. See the Warranty section above for instructions on returning a product to Parallax.

## Copyrights and Trademarks

This documentation is copyright 1994-2005 by Parallax Inc. By downloading or obtaining a printed copy of this documentation or software you agree that it is to be used exclusively with Parallax products. Any other uses are not permitted and may represent a violation of Parallax copyrights, legally punishable according to Federal copyright or intellectual property laws. Any duplication of this documentation for commercial uses is expressly prohibited by Parallax Inc. Duplication for educational use is permitted, subject to the following Conditions of Duplication: Parallax Inc. grants the user a conditional right to download, duplicate, and distribute this text without Parallax's permission. This right is based on the following conditions: the text, or any portion thereof, may not be duplicated for commercial use; it may be duplicated only for educational purposes when used solely in conjunction with Parallax products, and the user may recover from the student only the cost of duplication.

This text is available in printed format from Parallax Inc. Because we print the text in volume, the consumer price is often less than typical retail duplication charges.

BASIC Stamp, Stamps in Class, Board of Education, Boe-Bot, Todder, SumoBot, and SX-Key are registered trademarks of Parallax, Inc. If you decide to use registered trademarks of Parallax Inc. on your web page or in printed material, you must state that "(registered trademark) is a registered trademark of Parallax Inc." upon the first appearance of the trademark name in each printed document or web page. HomeWork Board, Parallax, and the Parallax logo are trademarks of Parallax Inc. If you decide to use trademarks of Parallax Inc. on your web page or in printed material, you must state that "(trademark) is a trademark of Parallax Inc.", "upon the first appearance of the trademark name in each printed document or web page. Other brand and product names are trademarks or registered trademarks of their respective holders.

## ISBN #1-928982-32-8

## Errata

While great effort is made to assure the accuracy of our texts, errors may still exist. If you find an error, please let us know by sending an email to [editor@parallax.com](mailto:editor@parallax.com). We continually strive to improve all of our educational materials and documentation, and frequently revise our texts. Occasionally, an errata sheet with a list of known errors and corrections for a given text will be posted to our web site, [www.parallax.com](http://www.parallax.com). Please check the individual product page's free downloads for an errata file.

## Disclaimer of Liability

Parallax Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, or any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax Inc. is also not responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your BASIC Stamp application, no matter how life-threatening it may be.

## Access Parallax via Internet

We maintain very a active web site for your convenience. These may be used to obtain software, communicate with members of Parallax, and communicate with other customers. Access information is shown below:

|                 |                                                                |
|-----------------|----------------------------------------------------------------|
| Web:            | <a href="http://www.parallax.com">http://www.parallax.com</a>  |
| General e-mail: | <a href="mailto:info@parallax.com">info@parallax.com</a>       |
| Tech. e-mail:   | <a href="mailto:support@parallax.com">support@parallax.com</a> |

## Internet BASIC Stamp Discussion List

We maintain active web-based discussion forums for people interested in Parallax products. These lists are accessible from [www.parallax.com](http://www.parallax.com) via the Support → Discussion Forums menu. These are the forums that we operate from our web site:

- [BASIC Stamps](#) – This list is widely utilized by engineers, hobbyists and students who share their BASIC Stamp projects and ask questions.
- [Stamps in Class®](#) – Created for educators and students, subscribers discuss the use of the Stamps in Class curriculum in their courses. The list provides an opportunity for both students and educators to ask questions and get answers.
- [Parallax Educators](#) – Exclusively for educators and those who contribute to the development of Stamps in Class. Parallax created this group to obtain feedback on our curricula and to provide a forum for educators to develop and obtain Teacher's Guides.
- [Translators](#) – The purpose of this list is to provide a conduit between Parallax and those who translate our documentation to languages other than English. Parallax provides editable Word documents to our translating partners and attempts to time the translations to coordinate with our publications.
- [Robotics](#) – Designed exclusively for Parallax robots, this forum is intended to be an open dialogue for a robotics enthusiasts. Topics include assembly, source code, expansion, and manual updates. The Boe-Bot®, Toddler®, SumoBot®, HexCrawler and QuadCrawler robots are discussed here.
- [SX Microcontrollers and SX-Key](#) – Discussion of programming the SX microcontroller with Parallax assembly language SX – Key® tools and 3<sup>rd</sup> party BASIC and C compilers.
- [Javelin Stamp](#) – Discussion of application and design using the Javelin Stamp, a Parallax module that is programmed using a subset of Sun Microsystems' Java® programming language.

## Supported Hardware, Firmware and Software

This manual is valid with the following software and firmware versions:

| BASIC Stamp Model | Firmware | Windows Interface |
|-------------------|----------|-------------------|
| BASIC Stamp 1     | 1.4      | 2.2               |
| BASIC Stamp 2     | 1.0      | 2.2               |
| BASIC Stamp 2e    | 1.1      | 2.2               |
| BASIC Stamp 2sx   | 1.1      | 2.2               |
| BASIC Stamp 2p    | 1.4      | 2.2               |
| BASIC Stamp 2pe   | 1.1      | 2.2               |
| BASIC Stamp 2px   | 1.0      | 2.2               |

The information herein will usually apply to newer versions but may not apply to older versions. New software can be obtained free on web site ([www.parallax.com](http://www.parallax.com)). If you have any questions about what you need to upgrade your product, please contact Parallax.

## Credits

Authorship and Editorial Review Team: Jeff Martin, Jon Williams, Ken Gracey, Aristides Alvarez, and Stephanie Lindsay;  
Cover Art: Jen Jacobs; Technical Graphics, Rich Allred; with many thanks to everyone at Parallax Inc.



# Contents

---

|                                                        |           |
|--------------------------------------------------------|-----------|
| <b>PREFACE .....</b>                                   | <b>5</b>  |
| <b>INTRODUCTION TO THE BASIC STAMP .....</b>           | <b>7</b>  |
| BASIC STAMP MODEL COMPARISON TABLE .....               | 8         |
| BASIC STAMP 1 HARDWARE .....                           | 10        |
| BASIC STAMP 2 HARDWARE .....                           | 13        |
| BASIC STAMP 2E HARDWARE .....                          | 15        |
| BASIC STAMP 2SX HARDWARE .....                         | 17        |
| BASIC STAMP 2P HARDWARE .....                          | 19        |
| BASIC STAMP 2PE HARDWARE .....                         | 21        |
| BASIC STAMP 2PX HARDWARE .....                         | 23        |
| GUIDELINES AND PRECAUTIONS .....                       | 25        |
| BASIC STAMP PROGRAMMING CONNECTIONS .....              | 27        |
| <b>QUICK START GUIDE .....</b>                         | <b>29</b> |
| <b>USING THE BASIC STAMP EDITOR .....</b>              | <b>35</b> |
| THE PROGRAMMING ENVIRONMENT .....                      | 35        |
| COMPILER DIRECTIVES .....                              | 43        |
| SPECIAL FUNCTIONS .....                                | 46        |
| SETTING PREFERENCES .....                              | 55        |
| ADVANCED COMPILATION TECHNIQUES .....                  | 68        |
| FEATURES FOR DEVELOPERS .....                          | 75        |
| <b>BASIC STAMP ARCHITECTURE .....</b>                  | <b>81</b> |
| RAM ORGANIZATION (BS1) .....                           | 81        |
| RAM ORGANIZATION (BS2, BS2E, BS2sx, BS2P, BS2PE) ..... | 82        |
| DEFINING AND USING VARIABLES .....                     | 84        |
| The Rules of Symbol Names .....                        | 86        |
| Defining Array Variables .....                         | 87        |
| Aliases and Variable Modifiers .....                   | 89        |
| CONSTANTS AND COMPILE-TIME EXPRESSIONS .....           | 94        |
| NUMBER REPRESENTATIONS .....                           | 96        |
| ORDER OF OPERATIONS .....                              | 102       |
| INTEGER MATH RULES .....                               | 103       |
| UNARY OPERATORS .....                                  | 105       |
| Absolute Value (ABS) .....                             | 105       |
| Cosine (COS) .....                                     | 106       |
| Decoder (DCD) .....                                    | 106       |

# Contents

---

|                                            |            |
|--------------------------------------------|------------|
| Negative (-).....                          | 106        |
| Encoder (NCD) .....                        | 107        |
| Sine (SIN) .....                           | 107        |
| Square Root (SQR) .....                    | 108        |
| BINARY OPERATORS .....                     | 109        |
| Add (+).....                               | 109        |
| Subtract (-).....                          | 110        |
| Multiply (*).....                          | 110        |
| Multiply High (**).....                    | 111        |
| Multiply Middle (*/).....                  | 112        |
| Divide (/) .....                           | 113        |
| Modulus (/).....                           | 113        |
| Arctangent (ATN).....                      | 114        |
| Hypotenuse (HYP).....                      | 115        |
| Minimum (MIN) .....                        | 115        |
| Maximum (MAX).....                         | 116        |
| Digit (DIG).....                           | 117        |
| Shift Left (<<).....                       | 117        |
| Shift Right (>>) .....                     | 117        |
| Reverse (REV) .....                        | 118        |
| And (&) .....                              | 118        |
| Or ( ) .....                               | 118        |
| Xor (^) .....                              | 119        |
| And Not (&/).....                          | 120        |
| Or Not ( /).....                           | 120        |
| Xor Not (^/) .....                         | 121        |
| <b>BASIC STAMP COMMAND REFERENCE .....</b> | <b>123</b> |
| PBASIC LANGUAGE VERSIONS .....             | 123        |
| CATEGORICAL LISTING OF COMMANDS .....      | 124        |
| SYNTAX CONVENTIONS.....                    | 128        |
| AUXIO.....                                 | 129        |
| BRANCH.....                                | 133        |
| BUTTON .....                               | 137        |
| COMPARE.....                               | 141        |
| CONFIGPIN .....                            | 143        |
| COUNT .....                                | 149        |
| DATA .....                                 | 153        |
| DEBUG .....                                | 159        |
| DEBUGIN.....                               | 171        |
| DO...LOOP.....                             | 175        |



## Contents

---

|                 |     |
|-----------------|-----|
| DTMFOUT.....    | 179 |
| EEPROM.....     | 183 |
| END.....        | 187 |
| EXIT.....       | 189 |
| FOR...NEXT..... | 191 |
| FREQOUT.....    | 199 |
| GET.....        | 203 |
| GOSUB.....      | 209 |
| GOTO.....       | 213 |
| HIGH.....       | 215 |
| I2CIN.....      | 217 |
| I2COUT.....     | 225 |
| IF...THEN.....  | 231 |
| INPUT.....      | 243 |
| IOTERM.....     | 247 |
| LCDCMD.....     | 249 |
| LCDIN.....      | 257 |
| LCDOUT.....     | 263 |
| LET.....        | 269 |
| LOOKDOWN.....   | 271 |
| LOOKUP.....     | 277 |
| LOW.....        | 281 |
| MAINIO.....     | 283 |
| NAP.....        | 285 |
| ON.....         | 289 |
| OUTPUT.....     | 293 |
| OWIN.....       | 295 |
| OWOUT.....      | 303 |
| PAUSE.....      | 311 |
| POLLIN.....     | 313 |
| POLLMODE.....   | 319 |
| POLLOUT.....    | 325 |
| POLLRUN.....    | 331 |
| POLLWAIT.....   | 335 |
| POT.....        | 339 |
| PULSIN.....     | 343 |
| PULSOUT.....    | 347 |
| PUT.....        | 351 |
| PWM.....        | 355 |
| RANDOM.....     | 359 |
| RCTIME.....     | 363 |
| READ.....       | 369 |

# Contents

---

|                                                 |            |
|-------------------------------------------------|------------|
| RETURN.....                                     | 375        |
| REVERSE.....                                    | 377        |
| RUN.....                                        | 381        |
| SELECT...CASE.....                              | 387        |
| SERIN.....                                      | 393        |
| SEROUT.....                                     | 415        |
| SHIFTIN.....                                    | 431        |
| SHIFTOUT.....                                   | 435        |
| SLEEP.....                                      | 441        |
| SOUND.....                                      | 445        |
| STOP.....                                       | 447        |
| STORE.....                                      | 449        |
| TOGGLE.....                                     | 455        |
| WRITE.....                                      | 459        |
| XOUT.....                                       | 465        |
| <b>APPENDIX A: ASCII CHART .....</b>            | <b>471</b> |
| <b>APPENDIX B: RESERVED WORDS.....</b>          | <b>473</b> |
| <b>APPENDIX C: CONVERSION FORMATTERS .....</b>  | <b>477</b> |
| <b>APPENDIX D: BASIC STAMP SCHEMATICS .....</b> | <b>481</b> |

## ***Preface***

---

**Thank you for purchasing a Parallax BASIC Stamp® microcontroller module.** We have done our best to produce several full-featured, easy to use development systems for BASIC Stamp microcontrollers. Depending on the Starter Kit you purchased, your BASIC Stamp model, development board and other contents will vary.

This manual is written for the latest available BASIC Stamp modules and software as of February 2005. As the product-line evolves, new information may become available. It is always recommended to visit the Parallax web site, [www.parallax.com](http://www.parallax.com), for the latest information.

This manual is intended to be a complete reference manual to the architecture and command structure of the various BASIC Stamp models. This manual is not meant to teach BASIC programming or electrical design; though a person can learn a lot by paying close attention to the details in this book.

If you have never programmed in the BASIC language or are unfamiliar with electronics, it would be best to locate one or more of the books listed on the following page for assistance. All are available, either to order or to download, from [www.parallax.com](http://www.parallax.com).

Books available in Adobe's PDF format are published for free download on the Parallax web site or on the CD-ROM which ships with our different Starter Kits. Books available in print may be purchased directly from Parallax or other distributors.

In addition, there are hundreds of great examples available on the Parallax CD and web site ([www.parallax.com](http://www.parallax.com)). Also, Nuts & Volts Magazine ([www.nutsvolts.com](http://www.nutsvolts.com) / 1-800-783-4624) is a national electronic hobbyist's magazine that features monthly articles featuring BASIC Stamp applications. This is an excellent resource for beginners and experts alike!

# Preface

---

| Book                                                                | Part #                           | Author and Publisher                                                                                                                           | Availability |                       |
|---------------------------------------------------------------------|----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|--------------|-----------------------|
|                                                                     |                                  |                                                                                                                                                | PDF          | In Print              |
| <i>What's a Microcontroller?</i>                                    | 28123                            | Andy Lindsay; Parallax Inc.; ISBN 1-928982-02-6                                                                                                | Yes          | Yes                   |
| <i>Robotics with the Boe-Bot</i>                                    | 28125                            | Andy Lindsay; Parallax Inc.; ISBN 1-928982-03-4                                                                                                | Yes          | Yes                   |
| <i>IR Remote for the Boe-Bot</i>                                    | 70016                            | Andy Lindsay; Parallax Inc.; ISBN 1-928982-31-X                                                                                                | Yes          | Yes                   |
| <i>Basic Analog and Digital</i>                                     | 28129                            | Andy Lindsay; Parallax Inc.; ISBN 1-928982-04-2                                                                                                | Yes          | Yes                   |
| <i>Applied Sensors</i>                                              | 28127                            | Tracy Allen, PhD.; Parallax Inc.; ISBN 1-928982-21-2                                                                                           | Yes          | Yes                   |
| <i>Understanding Signals</i>                                        | 28119<br>(With Full Kit)         | Doug Pientak; Parallax Inc.; ISBN 1-928982-23-9                                                                                                | Yes          | Yes                   |
| <i>Industrial Control</i>                                           | 27341                            | Marty Hebel / Will Devenport; Parallax Inc.; ISBN 1-928982-08-5                                                                                | Yes          | Yes                   |
| <i>Elements of Digital Logic</i>                                    | 70008                            | John Barrowman; Parallax Inc.; ISBN 1-928982-20-4                                                                                              | Yes          | Yes                   |
| <i>The Microcontroller Application Cookbook Volumes 1 and 2</i>     | Vol. 1&2: 28113<br>Vol. 2: 28112 | Matt Gilliland; Woodglen Press; ISBN 0-616-11552-7 and 0-972-01590-6                                                                           | No           | Yes                   |
| <i>AI's "World Famous" Stamp Project of the Month Anthology</i>     | 70013                            | AI Williams; Parallax Inc.; ISBN 1-928982-25-5                                                                                                 | Portions     | Yes                   |
| <i>The Nuts and Volts of BASIC Stamps Volumes 1, 2, 3, 4, and 5</i> | Vol. 4: 70010<br>Vol. 5: 70015   | Jon Williams, Scott Edwards and Lon Glazner; Parallax, Inc.; ISBN 1-928982-10-7, 1-928982-11-5, 1-928982-17-4, 1-928982-24-7 and 1-928982-30-1 | Yes (all)    | Yes (Vol 4 and Vol 5) |
| <i>StampWorks</i>                                                   | 27220                            | Jon Williams; Parallax, Inc.; ISBN 1-928982-07-7                                                                                               | Yes          | Yes                   |
| <i>Stamp 2 Communication and Control Projects</i>                   | 70004                            | Thomas Petruzzellis; McGraw-Hill; ISBN 0-071411-97-6                                                                                           | No           | Yes                   |
| <i>Programming and Customizing the BASIC Stamp Computer</i>         | 27956                            | Scott Edwards; McGraw-Hill; ISBN 0-071371-92-3                                                                                                 | No           | Yes                   |
| <i>BASIC Stamp 2p</i>                                               | 70001                            | Claus Kuehnel and Klaus Zahnert; Parallax, Inc.; ISBN 1-928982-19-0                                                                            | Yes          | No                    |

# ***1: Introduction to the BASIC Stamp***

---

Welcome to the wonderful world of BASIC Stamp<sup>®</sup> microcontrollers. BASIC Stamp microcontrollers have been in use by engineers and hobbyists since we first introduced them in 1992. As of November 2004, Parallax customers have put well over three million BASIC Stamp modules into use. Over this 12-year period, the BASIC Stamp line of controllers has evolved into six models and many physical package types, explained below.

## **General Operation Theory**

BASIC Stamp modules are microcontrollers (tiny computers) that are designed for use in a wide array of applications. Many projects that require an embedded system with some level of intelligence can use a BASIC Stamp module as the controller.

Each BASIC Stamp comes with a BASIC Interpreter chip, internal memory (RAM and EEPROM), a 5-volt regulator, a number of general-purpose I/O pins (TTL-level, 0-5 volts), and a set of built-in commands for math and I/O pin operations. BASIC Stamp modules are capable of running a few thousand instructions per second and are programmed with a simplified, but customized form of the BASIC programming language, called PBASIC.

## **PBASIC Language**

We developed PBASIC specifically for the BASIC Stamp as a simple, easy to learn language that is also well suited for this architecture, and highly optimized for embedded control. It includes many of the instructions featured in other forms of BASIC (GOTO, FOR...NEXT, IF...THEN...ELSE) as well as some specialized instructions (SERIN, PWM, BUTTON, COUNT and DTMFOUT). This manual includes an extensive section devoted to each of the available instructions.

## **Hardware**

At the time of this writing, there are currently seven models of the BASIC Stamp; the BS1, BS2, BS2e, BS2sx, BS2p, BS2pe, and the BS2px. The tables below are provided to easily compare their specifications, followed by diagrams that detail the various package types of these modules. Schematics for the SIP/DIP packages of all models can be found in Appendix D.

# Introduction to the BASIC Stamp

---

## BASIC Stamp Model Comparison Table

| Products                     | BS1                              | BS2                              | BS2e                            |
|------------------------------|----------------------------------|----------------------------------|---------------------------------|
| Environment                  | 0° - 70° C<br>(32° - 158° F) **  | 0° - 70° C (32° - 158° F) **     | 0° - 70° C (32° - 158° F) **    |
| Microcontroller              | Microchip PIC16C56a              | Microchip PIC16C57c              | Ubicom SX28AC                   |
| Processor Speed              | 4 MHz                            | 20 MHz                           | 20 MHz                          |
| Program Execution Speed      | ~2,000 instructions/sec.         | ~4,000 instructions/sec          | ~4,000 instructions/sec         |
| RAM Size                     | 16 Bytes (2 I/O, 14 Variable)    | 32 Bytes<br>(6 I/O, 26 Variable) | 32 Bytes (6 I/O, 26 Variable)   |
| Scratch PadRam               | N/A                              | N/A                              | 64 Bytes                        |
| EEPROM (Program) Size        | 256 Bytes,<br>~80 instructions   | 2K Bytes,<br>~500 instructions   | 8 x 2K Bytes,<br>~4,000 inst    |
| Number of I/O Pins           | 8                                | 16 + 2 Dedicated Serial          | 16 + 2 Dedicated Serial         |
| Voltage Requirements         | 5 - 15 vdc                       | 5 - 15 vdc                       | 5 - 12 vdc                      |
| Current Draw@ 5 volts        | 1 mA Run,<br>25 µA Sleep         | 3 mA Run,<br>50 µA Sleep         | 25 mA Run,<br>200 µA Sleep      |
| Source/Sink Current per I/O  | 20 mA / 25 mA                    | 20 mA / 25 mA                    | 30 mA / 30 mA                   |
| Source/Sink Current per unit | 40 mA / 50 mA                    | 40 mA / 50 mA<br>per 8 I/O pins  | 60 mA / 60 mA<br>per 8 I/O pins |
| PBASIC Commands*             | 32                               | 42                               | 45                              |
| PC Interface                 | Serial<br>(w/BS1 Serial Adapter) | Serial<br>(9600 baud)            | Serial (9600 baud)              |
| Windows Text Editor Version  | Stampw.exe (v2.1 and up)         | Stampw.exe (v1.04 and up)        | Stampw.exe<br>(v1.096 and up)   |

\* PBASIC Command count totals include PBASIC 2.5 commands on all BS2 models.

\*\* See below for industrial rated module information.

## Industrial-Rated BASIC Stamp Modules

Some BASIC Stamp models come in Industrial-rated versions, with an environmental temperature tolerance range of -40°C to +85°C . Contact the Parallax Sales Team directly for the latest information regarding industrial-rated product availability and specifications.

# 1: Introduction to the BASIC Stamp

| BS2sx                            | BS2p24                            | BS2p40                            | BS2pe                              | BS2px                             |
|----------------------------------|-----------------------------------|-----------------------------------|------------------------------------|-----------------------------------|
| 0° - 70° C<br>(32° - 158° F) **  | 0° - 70° C<br>(32° - 158° F) **   | 0° - 70° C<br>(32° - 158° F) **   | 0° - 70° C<br>(32° - 158° F) **    | 0° - 70° C<br>(32° - 158° F) **   |
| Ubicom SX28AC                    | Ubicom SX48AC                     | Ubicom SX48AC                     | Ubicom SX48AC                      | Ubicom SX48AC                     |
| 50 MHz                           | 20 MHz Turbo                      | 20 MHz Turbo                      | 8 MHz Turbo                        | 32 MHz Turbo                      |
| ~10,000 instructions/sec.        | ~12,000 instructions/sec.         | ~12,000 instructions/sec.         | ~6000 instructions/sec.            | ~19,000 instructions/sec.         |
| 32 Bytes<br>(6 I/O, 26 Variable) | 38 Bytes<br>(12 I/O, 26 Variable) | 38 Bytes<br>(12 I/O, 26 Variable) | 38 Bytes<br>(12 I/O, 26 Variable)  | 38 Bytes<br>(12 I/O, 26 Variable) |
| 64 Bytes                         | 128 Bytes                         | 128 Bytes                         | 128 Bytes                          | 128 Bytes                         |
| 8 x 2K Bytes,<br>~4,000 inst.    | 8 x 2K Bytes,<br>~4,000 inst.     | 8 x 2K Bytes,<br>~4,000 inst.     | 16 x 2K Bytes<br>(16 K for source) | 8 x 2K Bytes,<br>~4,000 inst.     |
| 16 + 2 Dedicated Serial          | 16 + 2 Dedicated Serial           | 32 + 2 Dedicated Serial           | 16 + 2 Dedicated Serial            | 16 + 2 Dedicated Serial           |
| 5 - 12 vdc                       | 5 - 12 vdc                        | 5 - 12 vdc                        | 5 - 12 vdc                         | 5 - 12 vdc                        |
| 60 mA Run,<br>500 µA Sleep       | 40 mA Run,<br>350 µA Sleep        | 40 mA Run,<br>350 µA Sleep        | 15 mA Run,<br>150 µA Sleep         | 55 mA Run,<br>450 µA Sleep        |
| 30 mA / 30 mA                    | 30 mA /<br>30 mA                  | 30 mA / 30 mA                     | 30 mA / 30 mA                      | 30 mA / 30 mA                     |
| 60 mA / 60 mA<br>per 8 I/O pins  | 60 mA / 60 mA<br>per 8 I/O pins   | 60 mA / 60 mA<br>per 8 I/O pins   | 60 mA / 60 mA<br>per 8 I/O pins    | 60 mA / 60 mA<br>per 8 I/O pins   |
| 45                               | 61                                | 61                                | 61                                 | 63                                |
| Serial (9600 baud)               | Serial (9600 baud)                | Serial<br>(9600 baud)             | Serial (9600 baud)                 | Serial (19200 baud)               |
| Stampw.exe<br>(v1.091 and up)    | Stampw.exe<br>(v1.1 and up)       | Stampw.exe<br>(v1.1 and up)       | Stampw.exe<br>(v1.33 and up)       | Stampw.exe<br>(v2.2 and up)       |

Phone: (916) 624-8333  
Toll free in the US or Canada: 1-888-512-1024  
Email: sales@parallax.com

# Introduction to the BASIC Stamp

## BASIC Stamp 1

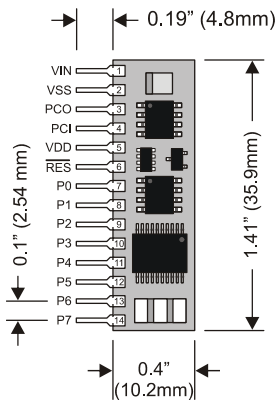


**Figure 1.1:** BASIC Stamp 1 (Rev B) (Stock# BS1-IC).

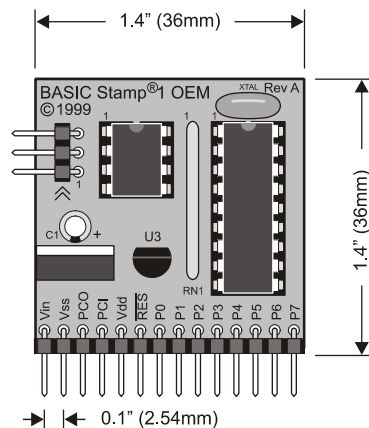
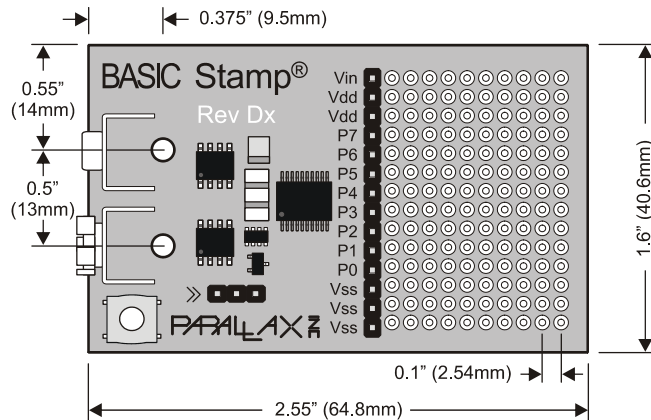


**Figure 1.2:** BASIC Stamp 1 OEM (Rev. A) (Stock# 27295).



## 1: Introduction to the BASIC Stamp

**Figure 1.3:** BASIC Stamp 1  
(Rev Dx) (Stock# 27100).



The BASIC Stamp 1 is available several physical packages. The BS1-IC (Figure 1.1) uses surface mount components to fit in a small 14-pin SIP package. The preassembled BASIC Stamp 1 OEM (Figure 1.2) features an easier-to-trace layout meant to aid customers who wish to integrate the BASIC Stamp 1 circuit directly into their design (as a lower-cost solution). The BASIC Stamp 1 Rev. Dx (simply called the Rev. Dx), see Figure 1.3, includes a prototyping area suitable for soldering electronic components. These three packages are functionally equivalent, except that the Rev. Dx does not have an available reset pin.

In addition to the packages shown, there are prototyping boards available that feature a surface mounted BS1 and programming cable connector. Please check [www.parallax.com](http://www.parallax.com) → Products → Development Boards for product descriptions.

## Introduction to the BASIC Stamp

---

| Pin  | Name  | Description                                                                                                                                                                                                                                                  |
|------|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | VIN   | Unregulated power in: accepts 5.5 - 15 VDC (6-40 VDC on BS1-IC rev. b), which is then internally regulated to 5 volts. May be left unconnected if 5 volts is applied to the VDD (+5V) pin.                                                                   |
| 2    | VSS   | System ground: connects to BS1 Serial Adapter ground for programming.                                                                                                                                                                                        |
| 3    | PCO   | PC Out: 4800 baud serial output (TTL level) to PC.                                                                                                                                                                                                           |
| 4    | PCI   | PC In: 4800 baud serial input (TTL level) from PC.                                                                                                                                                                                                           |
| 5    | VDD   | 5-volt DC input/output: (Also called +5V) if an unregulated voltage is applied to the VIN pin, then this pin will output 5 volts. If no voltage is applied to the VIN pin, then a regulated voltage between 4.5V and 5.5V should be applied to this pin.     |
| 6    | RES   | Reset input/output: goes low when power supply is less than approximately 4.2 volts, causing the BASIC Stamp to reset. Can be driven low to force a reset. This pin is internally pulled high and may be left disconnected if not needed. Do not drive high. |
| 7-14 | P0-P7 | General-purpose I/O pins: each can sink 25 mA and source 20 mA. However, the total of all pins should not exceed 50 mA (sink) and 40 mA (source).                                                                                                            |

**Table 1.1:** BASIC Stamp 1 Pin Descriptions.

See the "BASIC Stamp Programming Connections" section on page 27 for more information on the required programming connections between the PC and the BASIC Stamp.

# 1: Introduction to the BASIC Stamp

## BASIC Stamp 2

**Figure 1.4:** BASIC Stamp 2  
(Rev. G) (Stock# BS2-IC).

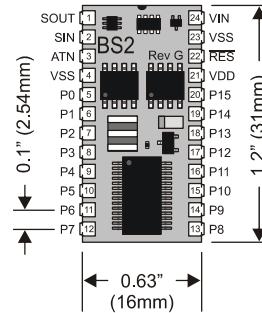
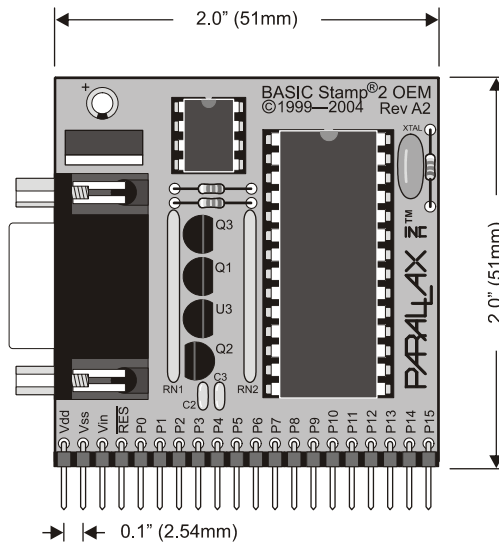


**Figure 1.5:** BASIC Stamp 2 OEM  
(Rev. A2) (Stock# 27290 assembled,  
or #27291 in kit form).



The BASIC Stamp 2 is available in several physical packages. The BS2-IC (Figure 1.4) uses surface mount components to fit in a small 24-pin DIP package. The BASIC Stamp 2 OEM (Figure 1.5) features an easier-to-trace layout meant to aid customers who wish to integrate the BASIC Stamp 2 circuit directly into their design (as a lower-cost solution). The BASIC

## Introduction to the BASIC Stamp

---

Stamp 2 OEM is available in either an assembled form or a kit form. These three packages are functionally equivalent.

In addition to the dual-inline and OEM packages, there are prototyping boards available that feature a surface mounted BS2. Please check [www.parallax.com](http://www.parallax.com) → Products → Development Boards for product descriptions.

| Pin  | Name   | Description                                                                                                                                                                                                                                                                                                                       |
|------|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | SOUT   | Serial Out: connects to PC serial port RX pin (DB9 pin 2 / DB25 pin 3) for programming.                                                                                                                                                                                                                                           |
| 2    | SIN    | Serial In: connects to PC serial port TX pin (DB9 pin 3 / DB25 pin 2) for programming.                                                                                                                                                                                                                                            |
| 3    | ATN    | Attention: connects to PC serial port DTR pin (DB9 pin 4 / DB25 pin 20) for programming.                                                                                                                                                                                                                                          |
| 4    | VSS    | System ground: (same as pin 23) connects to PC serial port GND pin (DB9 pin 5 / DB25 pin 7) for programming.                                                                                                                                                                                                                      |
| 5-20 | P0-P15 | General-purpose I/O pins: each can sink 25 mA and source 20 mA. However, the total of all pins should not exceed 50 mA (sink) and 40 mA (source) if using the internal 5-volt regulator. The total per 8-pin groups (P0 – P7 or P8 – 15) should not exceed 50 mA (sink) and 40 mA (source) if using an external 5-volt regulator. |
| 21   | VDD    | 5-volt DC input/output: if an unregulated voltage is applied to the VIN pin, then this pin will output 5 volts. If no voltage is applied to the VIN pin, then a regulated voltage between 4.5V and 5.5V should be applied to this pin.                                                                                            |
| 22   | RES    | Reset input/output: goes low when power supply is less than approximately 4.2 volts, causing the BASIC Stamp to reset. Can be driven low to force a reset. This pin is internally pulled high and may be left disconnected if not needed. Do not drive high.                                                                      |
| 23   | VSS    | System ground: (same as pin 4) connects to power supply's ground (GND) terminal.                                                                                                                                                                                                                                                  |
| 24   | VIN    | Unregulated power in: accepts 5.5 - 15 VDC (6-40 VDC on BS2-IC Rev. e, f, and g), which is then internally regulated to 5 volts. Must be left unconnected if 5 volts is applied to the VDD (+5V) pin.                                                                                                                             |

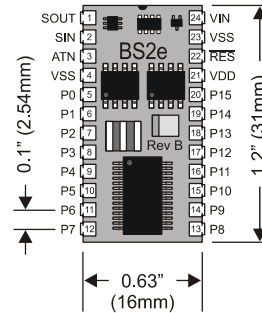
**Table 1.2:** BASIC Stamp 2 Pin Descriptions.

See the "BASIC Stamp Programming Connections" section on page 27 for more information on the required programming connections between the PC and the BASIC Stamp.

# 1: Introduction to the BASIC Stamp

## BASIC Stamp 2e

**Figure 1.6:** BASIC Stamp 2e (Rev. B) (Stock# BS2E-IC).



The BASIC Stamp 2e is available in the above 24-pin DIP package.

**Table 1.3:** BASIC Stamp 2e Pin Descriptions.

| Pin  | Name   | Description                                                                                                                                                                                                                                                                                                |
|------|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | SOUT   | Serial Out: connects to PC serial port RX pin (DB9 pin 2 / DB25 pin 3) for programming.                                                                                                                                                                                                                    |
| 2    | SIN    | Serial In: connects to PC serial port TX pin (DB9 pin 3 / DB25 pin 2) for programming.                                                                                                                                                                                                                     |
| 3    | ATN    | Attention: connects to PC serial port DTR pin (DB9 pin 4 / DB25 pin 20) for programming.                                                                                                                                                                                                                   |
| 4    | VSS    | System ground: (same as pin 23) connects to PC serial port GND pin (DB9 pin 5 / DB25 pin 7) for programming.                                                                                                                                                                                               |
| 5-20 | P0-P15 | General-purpose I/O pins: each can source and sink 30 mA. However, the total of all pins should not exceed 75 mA (source or sink) if using the internal 5-volt regulator. The total per 8-pin groups (P0 – P7 or P8 – 15) should not exceed 100 mA (source or sink) if using an external 5-volt regulator. |
| 21   | VDD    | 5-volt DC input/output: if an unregulated voltage is applied to the VIN pin, then this pin will output 5 volts. If no voltage is applied to the VIN pin, then a regulated voltage between 4.5V and 5.5V should be applied to this pin.                                                                     |
| 22   | RES    | Reset input/output: goes low when power supply is less than approximately 4.2 volts, causing the BASIC Stamp to reset. Can be driven low to force a reset. This pin is internally pulled high and may be left disconnected if not needed. Do not drive high.                                               |
| 23   | VSS    | System ground: (same as pin 4) connects to power supply's ground (GND) terminal.                                                                                                                                                                                                                           |
| 24   | VIN    | Unregulated power in: accepts 5.5 - 12 VDC (7.5 recommended), which is then internally regulated to 5 volts. Must be left unconnected if 5 volts is applied to the VDD (+5V) pin.                                                                                                                          |

## ***Introduction to the BASIC Stamp***

---

See the "BASIC Stamp Programming Connections" section on page 27 for more information on the required programming connections between the PC and the BASIC Stamp.

# 1: Introduction to the BASIC Stamp

## BASIC Stamp 2sx

**Figure 1.7:** BASIC Stamp 2sx  
(Rev. E) (Stock# BS2sx-IC)

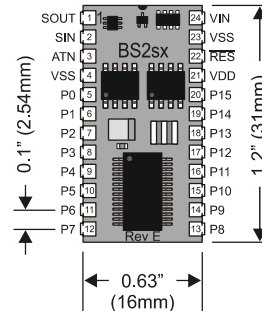
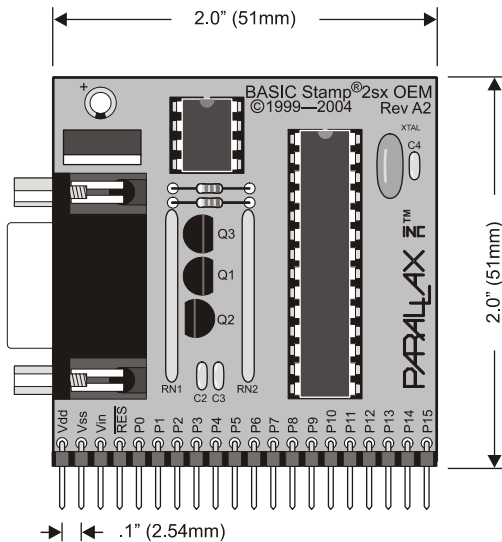


**Figure 1.8:** BASIC Stamp 2sx OEM  
(Rev. A2) (Stock# 27294)



The BASIC Stamp 2sx is available in the above two physical packages. The BS2sx-IC (Figure 1.7) uses surface mount components to fit in a small 24-pin DIP package. The preassembled BASIC Stamp 2sx OEM (Figure 1.8) features an easier-to-trace layout meant to aid customers who wish to integrate the BASIC Stamp 2sx circuit directly into their design (as a lower-cost solution). The BASIC Stamp 2sx OEM is available in assembled form only.

## Introduction to the BASIC Stamp

---

| Pin  | Name   | Description                                                                                                                                                                                                                                                                                                |
|------|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | SOUT   | Serial Out: connects to PC serial port RX pin (DB9 pin 2 / DB25 pin 3) for programming.                                                                                                                                                                                                                    |
| 2    | SIN    | Serial In: connects to PC serial port TX pin (DB9 pin 3 / DB25 pin 2) for programming.                                                                                                                                                                                                                     |
| 3    | ATN    | Attention: connects to PC serial port DTR pin (DB9 pin 4 / DB25 pin 20) for programming.                                                                                                                                                                                                                   |
| 4    | VSS    | System ground: (same as pin 23) connects to PC serial port GND pin (DB9 pin 5 / DB25 pin 7) for programming.                                                                                                                                                                                               |
| 5-20 | P0-P15 | General-purpose I/O pins: each can source and sink 30 mA. However, the total of all pins should not exceed 75 mA (source or sink) if using the internal 5-volt regulator. The total per 8-pin groups (P0 – P7 or P8 – 15) should not exceed 100 mA (source or sink) if using an external 5-volt regulator. |
| 21   | VDD    | 5-volt DC input/output: if an unregulated voltage is applied to the VIN pin, then this pin will output 5 volts. If no voltage is applied to the VIN pin, then a regulated voltage between 4.5V and 5.5V should be applied to this pin.                                                                     |
| 22   | RES    | Reset input/output: goes low when power supply is less than approximately 4.2 volts, causing the BASIC Stamp to reset. Can be driven low to force a reset. This pin is internally pulled high and may be left disconnected if not needed. Do not drive high.                                               |
| 23   | VSS    | System ground: (same as pin 4) connects to power supply's ground (GND) terminal.                                                                                                                                                                                                                           |
| 24   | VIN    | Unregulated power in: accepts 5.5 - 12 VDC (7.5 recommended), which is then internally regulated to 5 volts. Must be left unconnected if 5 volts is applied to the VDD (+5V) pin.                                                                                                                          |

**Table 1.4:** BASIC Stamp 2sx Pin Descriptions

See the "BASIC Stamp Programming Connections" section on page 27 for more information on the required programming connections between the PC and the BASIC Stamp.



# 1: Introduction to the BASIC Stamp

## BASIC Stamp 2p

**Figure 1.9:** BASIC Stamp 2p24  
(Rev. C) (Stock# BS2p24-IC)

This module is identical in function to the BS2p40-IC, except that it has 16 I/O pins.

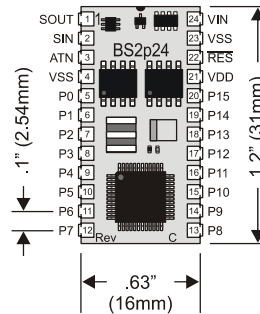
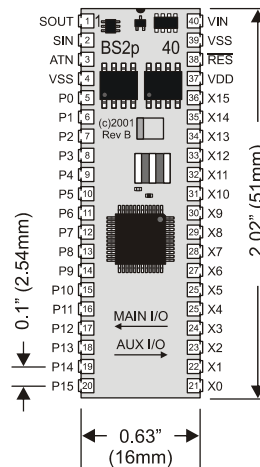


**Figure 1.10:** BASIC Stamp 2p40  
(Rev. B) (Stock# BS2p40-IC)

This module is identical in function to the BS2p24-IC, except that it has 32 I/O pins.



The BASIC Stamp 2p is available in the above two physical packages. Both packages use surface mount components to fit in a small package. The BS2p24-IC (Figure 1.9) is a 24-pin DIP package. The BS2p40-IC (Figure 1.10) is a 40-pin DIP package. Both packages are functionally equivalent except that the BS2p40 has 32 I/O pins instead of 16.

## Introduction to the BASIC Stamp

| Pin     | Name   | Description                                                                                                                                                                                                                                                                                                                                                                         |
|---------|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1       | SOUT   | Serial Out: connects to PC serial port RX pin (DB9 pin 2 / DB25 pin 3) for programming.                                                                                                                                                                                                                                                                                             |
| 2       | SIN    | Serial In: connects to PC serial port TX pin (DB9 pin 3 / DB25 pin 2) for programming.                                                                                                                                                                                                                                                                                              |
| 3       | ATN    | Attention: connects to PC serial port DTR pin (DB9 pin 4 / DB25 pin 20) for programming.                                                                                                                                                                                                                                                                                            |
| 4       | VSS    | System ground: (same as pin 23 on BS2p24, or pin 39 on BS2p40) connects to PC serial port GND pin (DB9 pin 5 / DB25 pin 7) for programming.                                                                                                                                                                                                                                         |
| 5-20    | P0-P15 | General-purpose I/O pins: each can source and sink 30 mA. However, the total of all pins (including X0-X15, if using the BS2p40) should not exceed 75 mA (source or sink) if using the internal 5-volt regulator. The total per 8-pin groups (P0 – P7, P8 – 15, X0 – X7 or X8 – X15) should not exceed 100 mA (source or sink) if using an external 5-volt regulator.               |
| {21-36} | X0-X15 | (BS2p40 Only!) Auxiliary Bank of General-purpose I/O pins: each can source and sink 30 mA. However, the total of all pins (including P0 – P15) should not exceed 75 mA (source or sink) if using the internal 5-volt regulator. The total per 8-pin groups (P0 – P7, P8 – 15, X0 – X7 or X8 – X15) should not exceed 100 mA (source or sink) if using an external 5-volt regulator. |
| 21 {37} | VDD    | 5-volt DC input/output: if an unregulated voltage is applied to the VIN pin, then this pin will output 5 volts. If no voltage is applied to the VIN pin, then a regulated voltage between 4.5V and 5.5V should be applied to this pin.                                                                                                                                              |
| 22 {38} | RES    | Reset input/output: goes low when power supply is less than approximately 4.2 volts, causing the BASIC Stamp to reset. Can be driven low to force a reset. This pin is internally pulled high and may be left disconnected if not needed. Do not drive high.                                                                                                                        |
| 23 {39} | VSS    | System ground: (same as pin 4) connects to power supply's ground (GND) terminal.                                                                                                                                                                                                                                                                                                    |
| 24 {40} | VIN    | Unregulated power in: accepts 5.5 - 12 VDC (7.5 recommended), which is then internally regulated to 5 volts. Must be left unconnected if 5 volts is applied to the VDD (+5V) pin.                                                                                                                                                                                                   |

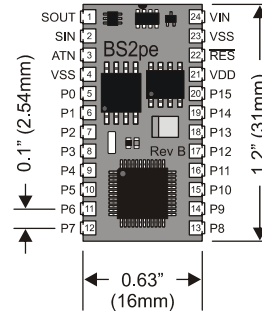
**Table 1.5:** BASIC Stamp 2p Pin Connections

See the "BASIC Stamp Programming Connections" section on page 27 for more information on the required programming connections between the PC and the BASIC Stamp.

# 1: Introduction to the BASIC Stamp

## Basic Stamp 2pe

**Figure 1.11:** BASIC Stamp 2pe (Rev. B) (Stock# BS2pe-IC)



The BASIC Stamp 2pe is available in the above 24-pin DIP physical package.

**Table 1.6:** BASIC Stamp 2pe Pin Descriptions.

| Pin  | Name   | Description                                                                                                                                                                                                                                                                                              |
|------|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | SOUT   | Serial Out: connects to PC serial port RX pin (DB9 pin 2 / DB25 pin 3) for programming.                                                                                                                                                                                                                  |
| 2    | SIN    | Serial In: connects to PC serial port TX pin (DB9 pin 3 / DB25 pin 2) for programming.                                                                                                                                                                                                                   |
| 3    | ATN    | Attention: connects to PC serial port DTR pin (DB9 pin 4 / DB25 pin 20) for programming.                                                                                                                                                                                                                 |
| 4    | VSS    | System ground: (same as pin 23), connects to PC serial port GND pin (DB9 pin 5 / DB25 pin 7) for programming.                                                                                                                                                                                            |
| 5-20 | P0-P15 | General-purpose I/O pins: each can source and sink 30 mA. However, the total of all pins should not exceed 75 mA (source or sink) if using the internal 5-volt regulator. The total per 8-pin groups P0 – P7 or P8 – 15 should not exceed 100 mA (source or sink) if using an external 5-volt regulator. |
| 21   | VDD    | 5-volt DC input/output: if an unregulated voltage is applied to the VIN pin, then this pin will output 5 volts. If no voltage is applied to the VIN pin, then a regulated voltage between 4.5V and 5.5V should be applied to this pin.                                                                   |
| 22   | RES    | Reset input/output: goes low when power supply is less than approximately 4.2 volts, causing the BASIC Stamp to reset. Can be driven low to force a reset. This pin is internally pulled high and may be left disconnected if not needed. Do not drive high.                                             |
| 23   | VSS    | System ground: (same as pin 4) connects to power supply's ground (GND) terminal.                                                                                                                                                                                                                         |
| 24   | VIN    | Unregulated power in: accepts 5.5 - 12 VDC (7.5 recommended), which is then internally regulated to 5 volts. Must be left unconnected if 5 volts is applied to the VDD (+5V) pin.                                                                                                                        |

## ***Introduction to the BASIC Stamp***

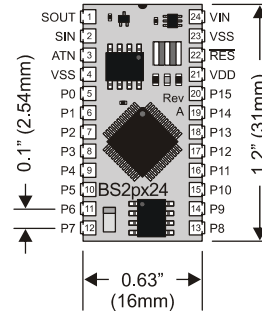
---

See the "BASIC Stamp Programming Connections" section on page 27 for more information on the required programming connections between the PC and the BASIC Stamp.

# 1: Introduction to the BASIC Stamp

## Basic Stamp 2px

**Figure 1.12:** BASIC Stamp 2px (Rev. A) (Stock# BS2px-IC)



The BASIC Stamp 2px is available in the above 24-pin DIP physical package.

**Table 1.7:** BASIC Stamp 2px Pin Descriptions.

| Pin  | Name   | Description                                                                                                                                                                                                                                                                                              |
|------|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | SOUT   | Serial Out: connects to PC serial port RX pin (DB9 pin 2 / DB25 pin 3) for programming.                                                                                                                                                                                                                  |
| 2    | SIN    | Serial In: connects to PC serial port TX pin (DB9 pin 3 / DB25 pin 2) for programming.                                                                                                                                                                                                                   |
| 3    | ATN    | Attention: connects to PC serial port DTR pin (DB9 pin 4 / DB25 pin 20) for programming.                                                                                                                                                                                                                 |
| 4    | VSS    | System ground: (same as pin 23), connects to PC serial port GND pin (DB9 pin 5 / DB25 pin 7) for programming.                                                                                                                                                                                            |
| 5-20 | P0-P15 | General-purpose I/O pins: each can source and sink 30 mA. However, the total of all pins should not exceed 75 mA (source or sink) if using the internal 5-volt regulator. The total per 8-pin groups P0 – P7 or P8 – 15 should not exceed 100 mA (source or sink) if using an external 5-volt regulator. |
| 21   | VDD    | 5-volt DC input/output: if an unregulated voltage is applied to the VIN pin, then this pin will output 5 volts. If no voltage is applied to the VIN pin, then a regulated voltage between 4.5V and 5.5V should be applied to this pin.                                                                   |
| 22   | RES    | Reset input/output: goes low when power supply is less than approximately 4.2 volts, causing the BASIC Stamp to reset. Can be driven low to force a reset. This pin is internally pulled high and may be left disconnected if not needed. Do not drive high.                                             |
| 23   | VSS    | System ground: (same as pin 4) connects to power supply's ground (GND) terminal.                                                                                                                                                                                                                         |
| 24   | VIN    | Unregulated power in: accepts 5.5 - 12 VDC (7.5 recommended), which is then internally regulated to 5 volts. Must be left unconnected if 5 volts is applied to the VDD (+5V) pin.                                                                                                                        |

## ***Introduction to the BASIC Stamp***

---

See the "BASIC Stamp Programming Connections" section on page 27 for more information on the required programming connections between the PC and the BASIC Stamp.

# ***1: Introduction to the BASIC Stamp***

---

## **Guidelines and Precautions**

When using the BASIC Stamp, or any IC chip, please follow the guidelines below.

1. **Be alert to static sensitive devices and static-prone situations.**
  - a. The BASIC Stamp, like other IC's, can be damaged by static discharge that commonly occurs touching grounded surfaces or other conductors. Environmental conditions (humidity changes, wind, static prone surfaces, etc) play a major role in the presence of random static charges. It is always recommended to use grounding straps and anti-static or static dissipative mats when handling devices like the BASIC Stamp. If the items above are not available, be sure to touch a grounded surface after you have approached the work area and before you handle static sensitive devices.
2. **Verify that all power is off before connecting/disconnecting.**
  - a. If power is connected to the BASIC Stamp or any device it is connected to while inserting or removing it from a circuit, damage to the BASIC Stamp or circuit could result.
3. **Verify BASIC Stamp orientation before connection to development boards and other circuits.**
  - a. Like other IC's, the BASIC Stamp should be inserted in a specific orientation in relation to the development board or circuit. Powering the circuit with an IC connected backwards will likely damage the IC and/or other components in the circuit. Most IC's have some form of a "pin 1 indicator" as do most IC sockets. This indicator usually takes the form of a dot, a half-circle, or the number 1 placed at or near pin 1 of the device.

The BS1-IC has a "1" and a half-circle indicator on the backside of the module. Additionally, Figure 1.1 above indicates the pin numbering and labels.

All BS2 series modules have a half-circle indicator on the topside of the module (see Figure 1.13). This indicates that pin number one is the first pin counterclockwise from the notch. The socket that accepts this 24-pin module also

*BASIC Stamp Syntax and Reference Manual 2.2 • [www.parallax.com](http://www.parallax.com) • Page 25*

## Introduction to the BASIC Stamp

has a half-circle or notch on one end, indicating the correct orientation. See Figure 1.14 for other examples.

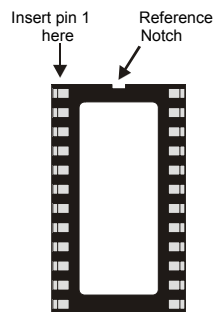
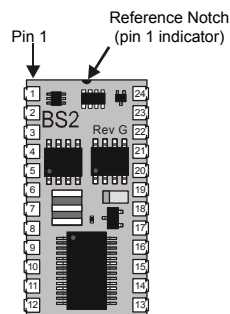


**Figure 1.13:** Pin 1 Indicators  
BS2-IC shown in the correct  
orientation in relation to a 24-pin  
socket.

Note: The Half-Circle indicator is  
also known as a Reference Notch

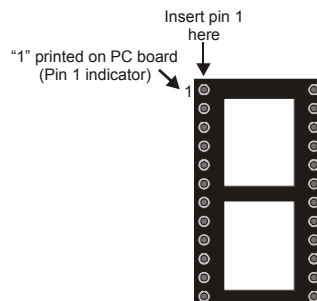
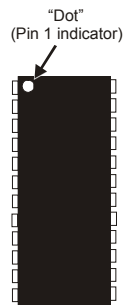


**Figure 1.14:** Additional  
Examples of Pin 1 Indicators  
(chip and socket shown in the  
correct orientation in relation to  
each other)



# 1: Introduction to the BASIC Stamp

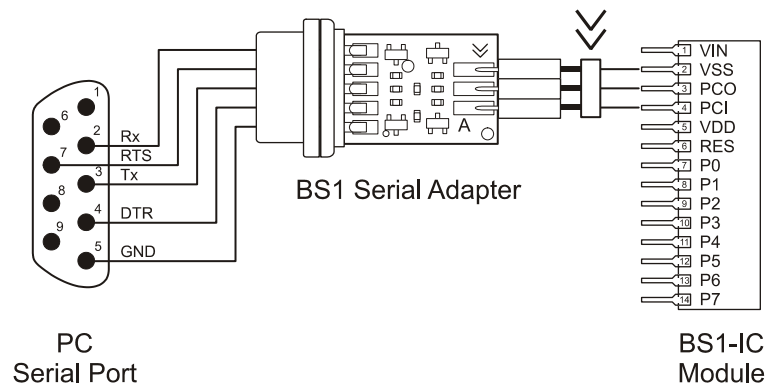
## BASIC Stamp Programming Connections

We suggest using a Parallax development board and cable for programming BASIC Stamp modules. When these items are not available, you may create your own board by duplicating the following diagrams with your own circuits and cables.

Be very careful to follow these diagrams closely; it is quite common for programming problems with the BASIC Stamp to be a result of a poorly made custom cable or programming connections on your applications board. With the programming connections for all the BS2 models, it is possible to reverse a couple of wires and still get positive results using some of the "connection" tests our Tech. Support team tries and yet you still will not be able to communicate with the BASIC Stamp. It is vital that you check your connections with a meter and verify the pin numbering to avoid problems like this.

**Figure 1.15:** BS1 Programming Connections with BS1 Serial Adapter

Note: Though it is not shown, power must be connected to the BS1 to program it.



## Introduction to the BASIC Stamp

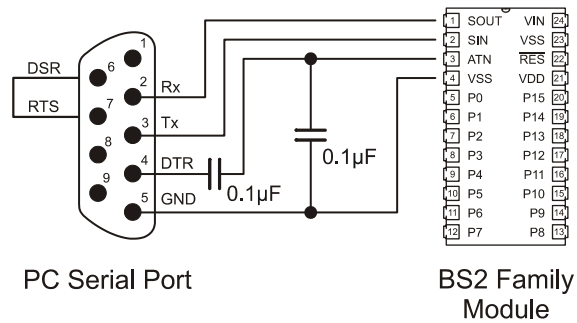


**Figure 1.16:** Programming and Run-time Communication Connections for all BS2 models.

Note: Though it is not shown, power must be connected to the BASIC Stamp to program it.

Also, the programming connections are the same for the BS2p40.

## ***2: Quick Start Guide***

---

### **Quick Start Introduction**

This chapter is a quick start guide to connecting the BASIC Stamp to the PC and programming it. Without even knowing how the BASIC Stamp functions, you should be able to complete the exercise below. This exercise assumes you have a BASIC Stamp and an appropriate development board. For the latest Parallax development board selection and documentation, go to [www.parallax.com](http://www.parallax.com) → Products → Development Boards. For a more detailed introduction to the BASIC Stamp Editor software, see Chapter 3.

### **Equipment Needed**

- BASIC Stamp module
- Compatible carrier board and programming cable
- Power supply (wall mount or battery) rated for your carrier board
- PC running Windows® 2000/XP, with
  - Quantity of RAM recommended for the OS
  - 3 MB of hard drive space
  - CD-ROM drive or Internet access
  - Available port compatible with your carrier board and cable (serial or USB)

### **Connecting and Downloading**

- 1) If the BASIC Stamp isn't already plugged into your development board, insert it into the socket. Refer to Figure 1.13 and Figure 1.14 on page 26 to make sure that you orient it correctly. For a complete listing of Parallax development boards for the various BASIC Stamp modules, go to [www.parallax.com](http://www.parallax.com) and look for Development Boards on the Products menu.
- 2) If you are using a Parallax development board, follow the directions that came with it to connect the board to the appropriate port in your computer. Figure 2.1 below shows the proper sequence for setting up with a BS1 and a BS1 Carrier Board. Figure 2.2 on page 31 shows the proper sequence to connect any BS2. Note: if you are using your own

## Quick Start Guide

---

development board or a breadboard, carefully follow the Programming Connections guidelines on page 27 before proceeding.

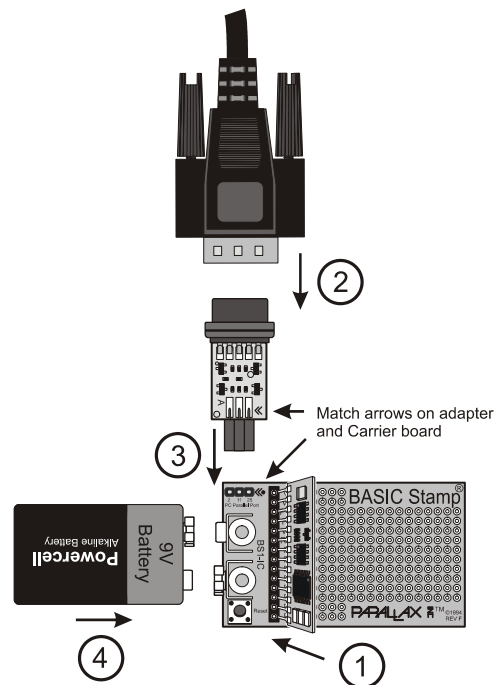


**Figure 2.1:** BS1-IC, BS1 Carrier Board, and BS1 Serial Adapter

- 1) Insert the BASIC Stamp module into its socket, being careful to orient it properly.
- 2) Connect the 9-pin female end of the serial cable to an available serial port on your computer, then attach the male end to the BS1 Serial Adapter, Note: you cannot use a null modem cable.
- 3) Plug the BS1 Serial Adapter into the programming header on the BS1 Carrier Board.
- 4) Plug a 9 volt battery into the 9 VDC battery clip.

## 2: Quick Start Guide

**Figure 2.2:** BS2-IC and Board of Education

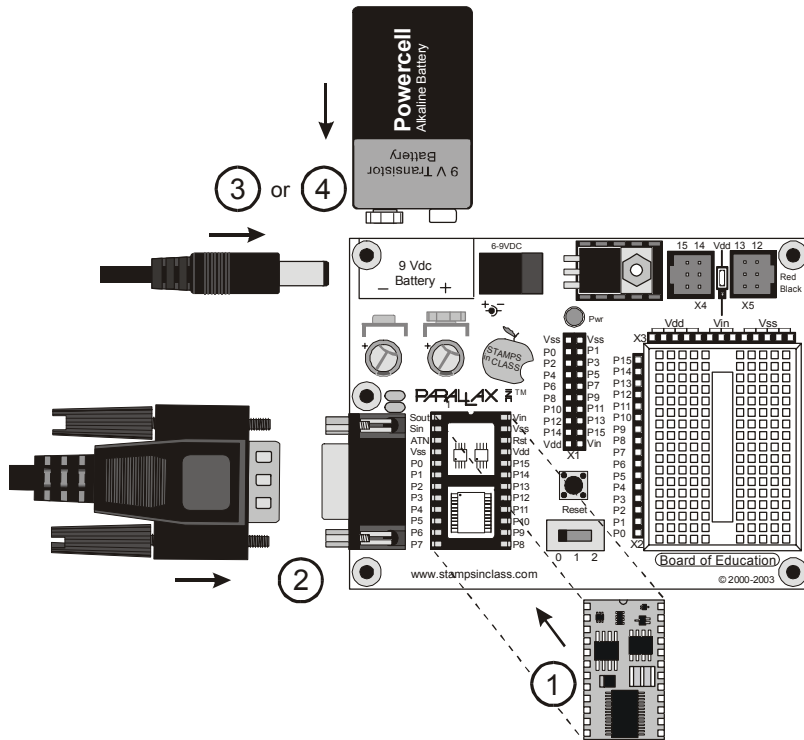
1) Insert the BASIC Stamp module into its socket, being careful to orient it properly.

2) Connect the 9-pin female end of the serial cable to an available serial port on your computer, and then connect the male end to the Board of Education. Note: you cannot use a null modem cable.

3) Plug in the 6-9 V 300mA center-positive power supply into the barrel jack.

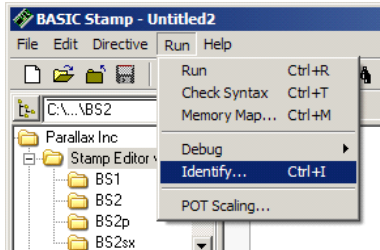
OR

4) Plug a 9 volt battery into the 9 VDC battery clip.



- 3) Install and run the BASIC Stamp Editor software.
  - a) If using the Parallax CD, go to the Software → BASIC Stamp → Windows section to locate the latest version. Click the Install button and follow the prompts to install and run.
  - b) If using the Parallax website, go to [www.parallax.com](http://www.parallax.com) → Downloads → Basic Stamp Software and look in the Software for Windows section for the latest version. Click the Download icon and follow the prompts to install and run.
  - c) Test your PC's connection to the BASIC Stamp by selecting Run → Identify from the menu bar, as shown in Figure 2.3. If the BASIC Stamp module is not found, check your power and cable connections and retry.

## Quick Start Guide



Select Run → Identify

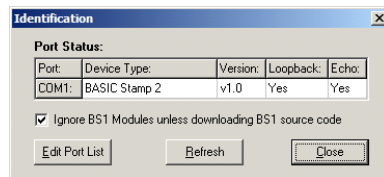


**Figure 2.3:** Test your PC connection to the BASIC Stamp.

Verify that the BASIC Stamp was detected on one of the COM ports.

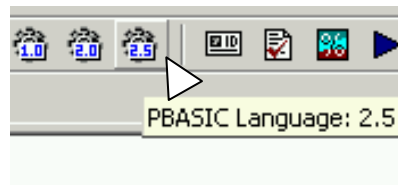
- 4) Enter a \$STAMP Directive into the Editor window by clicking on the toolbar icon for the BASIC Stamp module you are using. (Hold the cursor over the icons for flyover help labels.) The example below shows the Stamp Directive that would be inserted for the BS2.

```
' {$STAMP BS2}
```

- 5) Enter a \$PBASIC Directive into the Editor window with the toolbar icon. For a BS1, you must use PBASIC 1.0. All BS2 series modules can use PBASIC 2.0 or 2.5. The command set differences between PBASIC 2.0 and 2.5 are covered in Chapter 5.



Click on the icon that corresponds to your BASIC Stamp model to automatically place the \$STAMP directive in your program.



Click on the icon for the PBASIC language version that is compatible with your BASIC Stamp model.

**Figure 2.4:** Entering the \$STAMP and \$PBASIC directives from the toolbar

The examples shown would be used for programming a BS2 module in PBASIC 2.5

You should now see both a \$STAMP directive and \$PBASIC directive on your PC screen:

```
' {$STAMP BS2}
' {$PBASIC 2.5}
```

- a) Note: These directives may be typed in from the keyboard, but failure to type this line properly may cause the editor to fail to recognize your BASIC Stamp during the next step.

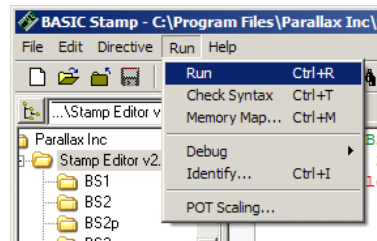
## 2: Quick Start Guide

- 6) Type the line DEBUG "Hello World!" below the compiler directives:

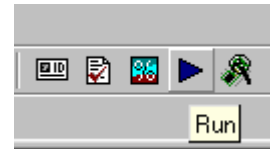
```
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Hello World!"
```

- 7) Download this program into the BASIC Stamp. You may select Run → Run from the menu bar, press CTRL-R from the keyboard, or click on the Run ► icon on the toolbar.

**Figure 2.5:** To run your program, you may use the task bar menu or the Run icon.



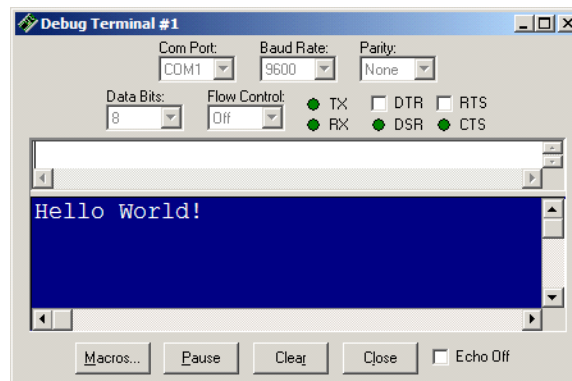
*Selecting Run → Run*



*Using the Run toolbar icon*

- a) If the program is typed correctly, a progress bar window should appear (perhaps very briefly) showing the download progress. Then a Debug Terminal window should appear and display "Hello World!"

**Figure 2.6:** Debug Terminal displaying program output



- b) If there is a syntax error in the program, the editor will highlight the text in question and display an error message. Review the error, fix the code and then try downloading again.

## ***Quick Start Guide***

---

- c) If the error reported a connection problem with the BASIC Stamp, make sure the first line of code indicates the proper module name and verify the programming cable connections, module orientation (in the socket) and that it is properly powered, then try downloading again.
- 8) Congratulations! You've just written and downloaded your first BASIC Stamp program! The "Hello World!" text that appeared on the screen was sent from the BASIC Stamp, back up the programming cable, to the PC.



## 3: Using the BASIC Stamp Editor

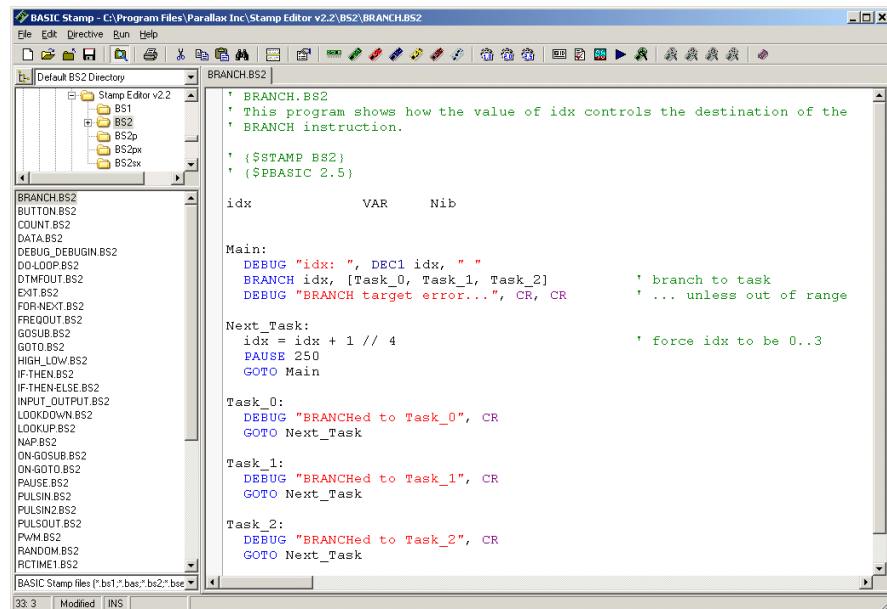
### Introducing the BASIC Stamp Editor

This section describes the BASIC Stamp Editor for Windows version 2.2. This software supports all 7 BASIC Stamp modules available as of February 2005, and all 3 versions of the PBASIC programming language, PBASIC 1.0, PBASIC 2.0, and PBASIC 2.5.

### The Programming Environment

The BASIC Stamp Windows Editor, shown in Figure 3.1, was designed to be easy to use and mostly intuitive. Those that are familiar with standard Windows software should feel comfortable using the BASIC Stamp Windows Editor.

**Figure 3.1:** BASIC Stamp Windows Editor.



THE EDITOR WINDOW.

The editor window consists of the main edit pane with an integrated explorer panel to its left, as shown above.

THE MAIN EDIT PANE.

The main edit pane can be used to view and modify up to 16 different source code files at once. Each source code file that is loaded into the

## Using the BASIC Stamp Editor

editor will have its own tab at the top of the page labeled with the name of the file, as seen in Figure 3.2. The full file path of the currently displayed source code appears in the title bar. Source code that has never been saved to disk will default to “Untitled#”; where # is an automatically generated number. A user can switch between source code files by simply pointing and clicking on a file’s tab or by pressing Ctrl+Tab or Ctrl+Shift+Tab while the main edit pane is active.

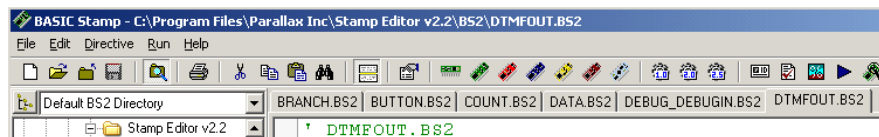


**Figure 3.2:** Example Editor Tabs. Shown with 6 separate files open; Title Bar shows current code’s file path.

The status of the active source code is indicated in the status bar below the main edit pane and integrated explorer panel. The status bar contains information such as cursor position, file save status, download status and syntax error/download messages. The example in Figure 3.3 indicates that the source code tokenized successfully.

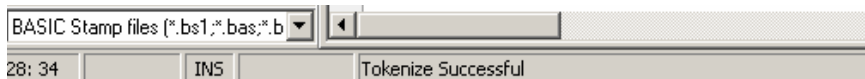


**Figure 3.3:** Status Bar beneath the Main Edit Pane.

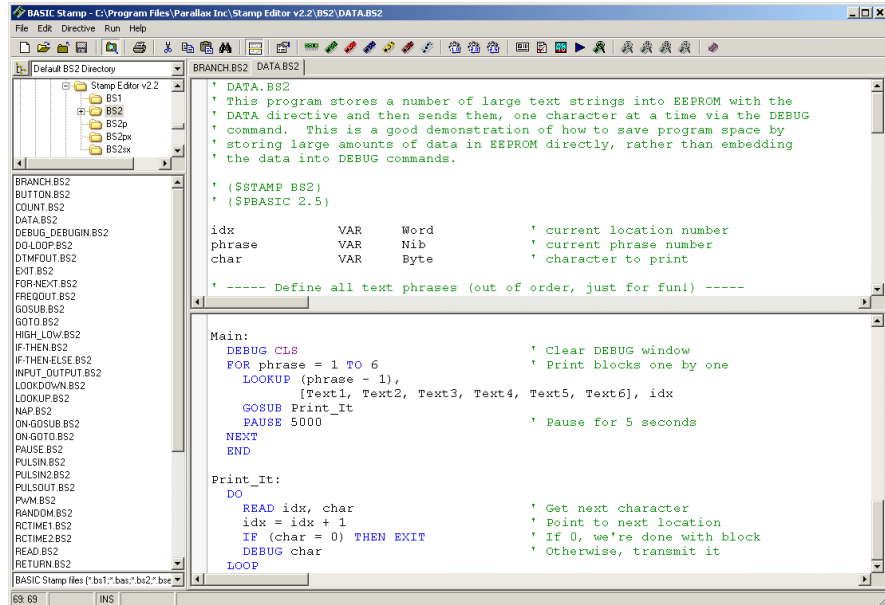
Each editor pane can be individually split into two views of the same source code. This can be done via the Split button on the toolbar, pressing Ctrl-L, or clicking and dragging the top or bottom border of the editor pane with the mouse.

SPLIT WINDOW VIEW.

Once split, the top and bottom edit controls allow viewing of different areas of the same source code; this can be handy when needing to keep variable declarations or a particular routine in view while modifying a related section of code elsewhere. Note that the Split button and Ctrl+L shortcut act like a toggle function, splitting or un-splitting the edit pane.

## 3: Using the BASIC Stamp Editor

**Figure 3.4:** The Split Edit Pane Feature displaying the beginning and end of a long program at the same time.



SYNTAX HIGHLIGHTING.

NOTE: a complete list of reserved words can be found in Appendix B.

Within the edit pane, BASIC Stamp source code files are displayed with syntax highlighting. Syntax Highlighting applies designated colors and character case (upper, lower, capitalized) to reserved words in the PBASIC language. This happens automatically as you type. Table 3.1 shows the default syntax highlighting settings for each syntax element.

If you copy and paste a program into a blank edit pane, select Run → Syntax Check or click on the toolbar checkmark icon to activate the syntax highlighting in that file. The syntax highlighting settings can be changed or customized via the Preferences → Editor Appearance tab; for details see the Setting Preferences section which begins on page 55. Source code can be printed to paper with the active syntax highlighting (and in color if using a color printer).

## Using the BASIC Stamp Editor

| Syntax Element                 | Text Color    | Character Case |
|--------------------------------|---------------|----------------|
| Command                        | Blue          | Upper Case     |
| Comment                        | Green         | No Change      |
| Constant - Binary              | Default       | No Change      |
| Constant - Decimal             | Default       | No Change      |
| Constant - Hexadecimal         | Default       | No Change      |
| Constant – Predefined          | Purple        | Upper case     |
| Constant – String              | Red           | No Change      |
| Operators                      | Default       | Upper case     |
| Declaration                    | Default       | Upper Case     |
| Directive, Conditional Compile | Gray(Bold)    | Upper case     |
| Directive, Editor              | Teal (Bold)   | Upper case     |
| Directive, Target module       | Teal (Bold)   | Upper case     |
| Input/Output Formatter         | Navy          | Upper case     |
| Selection                      | White on Navy | No change      |
| Search match                   | Lime on black | No change      |
| Variable modifier              | Default       | Upper case     |
| Variable – predefined          | Purple        | Upper case     |
| Variable, type                 | Default       | Capitalize     |

**Table 3.1:** Syntax Highlighting Defaults for the PBASIC Scheme.

NOTE: The default edit pane has a white background with black characters.

Automatic line numbers can be enabled or disabled via the “Show Line Numbers” checkbox on the Preferences → Editor Appearance tab. Line numbers, when enabled, appear in a gutter (the gray area on the left of the edit pane as shown in Figure 3.5). When printing, the line numbers may be included if desired.

AUTOMATIC LINE NUMBERING.

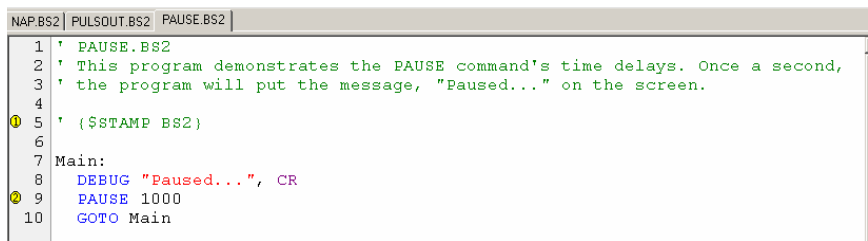


**Figure 3.5:** Automatic Line Numbering appears in the gutter to the left of the edit pane. Yellow Bookmarks are visible on lines 5 and 9.

Bookmarks can be enabled or disabled via the “Show Bookmarks” checkbox on the Preferences → Editor Appearance tab. The bookmarks appear in the gutter as small numbered icons, providing a way to mark lines or sections of code that you need to navigate to quickly or repeatedly. You can define up to nine bookmarks by clicking on the gutter where you want one placed, or by pressing Ctrl+B when the cursor is on the desired line. You can instantly navigate to any defined bookmark by pressing Ctrl+(#) (where # can be the 1 through 9 keys) or by selecting Go To

BOOKMARKS.

---

## 3: Using the BASIC Stamp Editor

---

Bookmark from either the Edit menu or from the shortcut menu (right-click) in the edit pane.

### EDITING YOUR CODE.

You can navigate through and edit your code in the edit pane with keyboard shortcuts, most of which will be familiar to Windows users.

**Table 3.2:** Keyboard Shortcuts for Editing and Navigation Functions.

| Editing and Navigation Functions |                                                      |
|----------------------------------|------------------------------------------------------|
| Shortcut Key                     | Function                                             |
| Ctrl+A                           | Select all text in current source code               |
| Ctrl+B                           | Set or clear bookmark on current source code line    |
| Ctrl+(#)                         | Go to bookmark #, where # can be 1 through 9         |
| Ctrl+C                           | Copy selected text to the clipboard                  |
| Ctrl+F                           | Find or replace text                                 |
| Ctrl+L                           | Split or un-split edit pane                          |
| Ctrl+N                           | Insert line                                          |
| Ctrl+V                           | Paste text from clipboard to selected area           |
| Ctrl+X                           | Cut selected text to the clipboard                   |
| Ctrl+Y                           | Delete current line of code                          |
| Ctrl+Shift+Y                     | Delete from cursor to end of current line            |
| Ctrl+Z                           | Undo last action (unlimited)                         |
| Ctrl+Shift+Z                     | Redo last action (unlimited)                         |
| Tab                              | Indent block (Inserts tab or space characters)       |
| Shift+Tab                        | Outdent block (Deletes tab or space characters)      |
| F3                               | Find text again                                      |
| F4                               | Replace current found selection                      |
| Ctrl+F4                          | Perform replace and find next                        |
| Ctrl+Home                        | Jump to top of file                                  |
| Ctrl+End                         | Jump to end of file                                  |
| Ctrl+PageUp                      | Jump to top of screen                                |
| Ctrl+PageDown                    | Jump to bottom of screen                             |
| Ctrl+CursorUp                    | Move source view up one line without moving cursor   |
| Ctrl+CursorDown                  | Move source view down one line without moving cursor |
| F5                               | Open Preferences window                              |

Some editing functions, specifically Cut, Copy, Paste, and Find/Replace, can also be accessed from the edit pane's shortcut menu (by right-clicking in the edit pane).

### THE FIND/REPLACE WINDOW.

The Find/Replace window allows you to set several search parameters. Match whole or partial words, match case, and match with wildcard options can be used singly or together. You can begin your search at the cursor or at the top or bottom of the selection or the entire file, and search in the forward (downward) or backward (upward) direction. You may replace a single instance of a given item or all instances at once. Recent

## Using the BASIC Stamp Editor

---

Find and Replace items are saved in the Find: and Replace: field's drop-down lists.

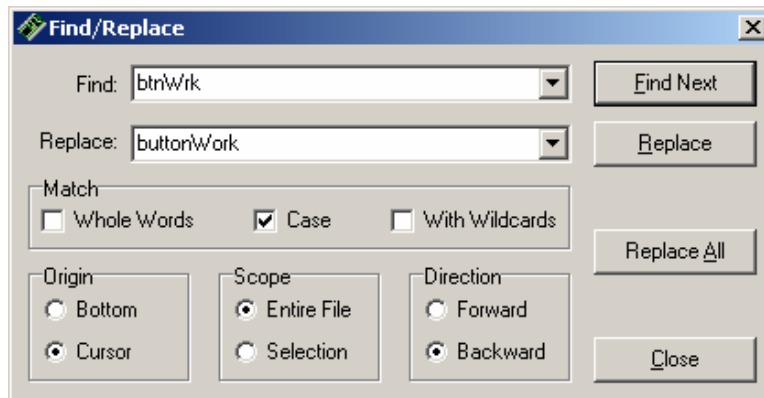


**Figure 3.6:** The Find/Replace Window.

The Find/Replace window will stay visible when using the Find Next and Replace options for quick and convenient source code editing. Using the Replace All function, however, will close the Find/Replace window and perform the designated find/replace operation.

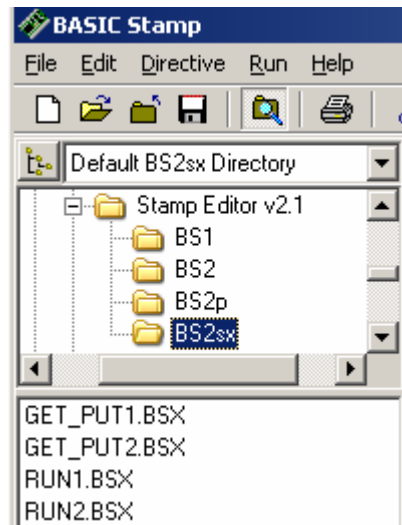
The integrated explorer panel to the left of the main edit pane is divided in to four portions: the Recent, Directory, File and Filter lists. The upper portion is the Recent list, a drop-down list of default, favorite, and recently visited directories.

THE INTEGRATED EXPLORER PANEL.

If you select a directory from the Recent list drop down field, the integrated explorer will automatically navigate to that directory. The button to the left of the Recent list allows you to limit the Directory list display below it to only the directories that are in the Recent list. This makes it easy to find your commonly used source code directories among a large set of directories and local and network hard drives. The Recent list button behaves like a toggle switch: 1) selecting it switches to the "Show Recent folders only" mode, 2) selecting it again switches back to the "Show all folders" mode.

## 3: Using the BASIC Stamp Editor

**Figure 3.7:** The Integrated Explorer Panel's Recent list (top), Directory list (middle), and File list (bottom).



### THE DIRECTORY LIST.

The Directory list, right below the Recent list, displays drives and directories in a hierarchical tree fashion. If a directory is selected, the Folders list displays the files in that directory.

### THE FILE LIST.

The File list, below the Directory list, displays all the files in the selected directory that match the selected filter (from the Filter list at the bottom, see Figure 3.8). You can select one or more files from this list and double-click, or drag-and-drop them over the editor pane, to open those files.

### OPEN FROM... AND SAVE TO... OPTIONS.

You may also open files with the Open From... option by selecting File → Open From, or by pressing Ctrl+Shift+O. This allows quick access to any directory for the default and favorite directories set within Preferences (see page 60) as well as any recently used directory. The Save To... option works similarly; select File → Save To or press Ctrl+Shift+S. These features can be very helpful if you organize your files in many different directories.

### THE FILTERS LIST.

The Filter list at the bottom of the explorer panel (Figure 3.8), is a drop-down list of file extension filters to apply to the File list. It works just like the "Save as type:" field of a standard Open or Save dialog.

# Using the BASIC Stamp Editor

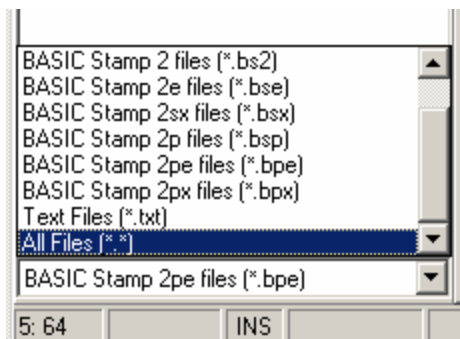


**Figure 3.8:** The Filter List found at the bottom of the Integrated Explorer Panel.

The BASIC Stamp Editor automatically associates BASIC Stamp source code file types (.bs1, .bs2, .bse, .bsx, .bsp, .bpe, and .bpx) with itself. This feature can be configured through automatic prompts or through the Preferences → Files & Directories tab. Also, when using any Explorer-shell for file browsing, right-clicking on a BASIC Stamp source code file provides you with an Open With Stamp Editor option.

The integrated explorer panel can be resized via the vertical splitter bar that separates it and the edit pane. The Directory list and File list can be resized via the horizontal splitter bar that separates them. The integrated explorer can also be hidden or shown via the Explorer toolbar button, by pressing Ctrl+E, or by resizing it to zero width using the vertical splitter bar.

Table 3.3 lists keyboard shortcuts for several file functions.

| File Functions |                                                                |
|----------------|----------------------------------------------------------------|
| Shortcut Key   | Function                                                       |
| Ctrl+E         | Show/hide explorer panel                                       |
| Ctrl+L         | Show/hide split view in edit pane                              |
| Ctrl+O         | Open a source code file into edit pane                         |
| Ctrl+Shift+O   | Open a source code file from a recent directory into edit pane |
| Ctrl+S         | Save current source code file to its current location on disk  |
| Ctrl+Shift+S   | Save current source code file to a recent directory on disk    |
| Ctrl+P         | Print current source code                                      |
| Ctrl+Tab       | Switch to next open file page                                  |
| Ctrl+Shift+Tab | Switch to previous open file page                              |

**Table 3.3:** Keyboard Shortcuts for File Functions.



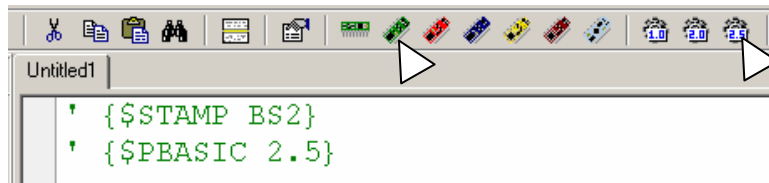
## 3: Using the BASIC Stamp Editor

### Compiler Directives

#### COMPILER DIRECTIVES .

The BASIC Stamp Editor supports all of the BASIC Stamp models, and all versions of the PBASIC programming language. Compiler directives must be placed in each program to indicate the desired BASIC Stamp model and language version. In addition, it is sometimes useful to target a given program to a particular communication port. The Directive menu contains options for setting the \$STAMP, \$PBASIC, and \$PORT directives. Since the \$STAMP and \$PBASIC directives are used most often, they are most easily inserted or modified with the toolbar buttons, as shown in Figure 3.9.

**Figure 3.9:** Toolbar icons make it easy to insert or modify \$STAMP and \$PBASIC directives directly in your program.



#### THE \$STAMP DIRECTIVE.

A \$STAMP directive is required in each PBASIC program. The editor determines which BASIC Stamp model to target for compiling and downloading based on this directive. Any code that is missing the \$STAMP directive, but whose filename contains a known BASIC Stamp extension (.bs1, .bs2, .bse, .bsx, .bsp, .bpe, .bpx) will be recognized by that extension and an appropriate \$STAMP directive will be added automatically when you run, tokenize, view the memory map or download the program. If there is no file extension present, an error message will prompt you to enter a \$STAMP directive.

#### FORMAT OF THE \$STAMP DIRECTIVE.

You may choose to manually type the \$STAMP directive into the program from the keyboard. This line should be entered into your code on a line by itself, usually near the top. Note that the directive appears on a comment line, as indicated by the apostrophe (').

|                   |                                                 |
|-------------------|-------------------------------------------------|
| ' {\$STAMP BS1}   | 'This indicates to use a BASIC Stamp 1 module   |
| ' {\$STAMP BS2}   | 'This indicates to use a BASIC Stamp 2 module   |
| ' {\$STAMP BS2e}  | 'This indicates to use a BASIC Stamp 2e module  |
| ' {\$STAMP BS2sx} | 'This indicates to use a BASIC Stamp 2sx module |
| ' {\$STAMP BS2p}  | 'This indicates to use a BASIC Stamp 2p module  |
| ' {\$STAMP BS2pe} | 'This indicates to use a BASIC Stamp 2pe module |
| ' {\$STAMP BS2px} | 'This indicates to use a BASIC Stamp 2px module |

## Using the BASIC Stamp Editor

---

If you choose to type the \$STAMP directive, care must be taken, or it will not be recognized. The directive itself must be enclosed in curly braces, {...}, not parentheses (...) or square brackets [...]. There should not be any spaces between the dollar sign (\$) and the word STAMP; however, the directive may contain additional spaces in certain other areas. For example:

```
' { $STAMP BS2 }
```

-- or --

```
' {$STAMP BS2}
```

-- and --

```
' {$STAMP BS2 }
```

are all acceptable variations. However:

```
' {$ STAMP BS2}
```

-- and --

```
' {$STAMPBS2}
```

are not acceptable and will be ignored. If one of the above two lines were entered into the source code, the editor would ignore it and, instead, rely on the extension of the filename to determine the appropriate model.

The \$STAMP directive is read and acted upon by the BASIC Stamp Windows Editor any time a source code file is loaded, tokenized, downloaded (run) or viewed in the Memory Map.

In some cases you may wish to write a program that can run on multiple BASIC Stamp models. In this case, conditional compile directives can be employed that will cause the editor to determine which Basic Stamp model is detected, and then download only those program elements applicable to that model. Many of the demo programs in Chapter 5 use this technique. To read about conditional compilation, see the Advanced Compilation Techniques section which begins on page 68.

EXTRA SPACES ARE ALLOWED IN CERTAIN AREAS.

PROGRAMS FOR MULTIPLE BASIC STAMP MODELS—CONDITIONAL COMPILE.

The \$PBASIC directive allows you to indicate which version of the PBASIC language to use. At the time of this printing, the options are 1.0, 2.0 and 2.5. If no \$PBASIC directive is present in the program, version 1.0 is assumed for BS1 module source code, and version 2.0 is assumed for

THE \$PBASIC DIRECTIVE.

## 3: Using the BASIC Stamp Editor

any BS2 model source code. A \$PBASIC directive is required to use version 2.5, which is compatible with all BS2 models.

PBASIC 2.5 has enhanced syntax options for several commands, as well as some additional commands not available in PBASIC 2.0. Table 3.4 shows the number of PBASIC commands that are available in each version of the PBASIC language, on each BASIC Stamp model. Details about the syntax differences among the three versions of PBASIC are denoted by icons in the margins of Chapters 4 and 5; also refer to Table 5.1 on page 124 and individual command syntax descriptions.

**Table 3.4:** Number of Available Commands for each BASIC Stamp Model with each version of the PBASIC language .

|                   | BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-------------------|-----|-----|------|-------|------|-------|-------|
| <b>PBASIC 1.0</b> | 32  | -   | -    | -     | -    | -     | -     |
| <b>PBASIC 2.0</b> | -   | 37  | 40   | 40    | 56   | 56    | 58    |
| <b>PBASIC 2.5</b> | -   | 42  | 45   | 45    | 61   | 61    | 63    |

A categorical listing of all PBASIC commands is included at the beginning of Chapter 5, followed by detailed descriptions of each command in alphabetical order.

Note that the syntax-highlighting feature of the BASIC Stamp Editor will also adjust to the language version indicated by the \$PBASIC directive. The best way to select the \$PBASIC directive is to use the toolbar icons, as was shown in Figure 3.9. Like the \$STAMP directive, you must use care if you choose to type it in by hand. The syntax is:

```
' {$PBASIC 1.0} 'Default when a BASIC Stamp 1 module is detected
' {$PBASIC 2.0} 'Default when any BASIC Stamp 2 module is detected
' {$PBASIC 2.5} 'Required for PBASIC 2.5 command set & enhanced syntax
```

If you try to run a program that contains command syntax specific to PBASIC 2.5 without including the corresponding compiler directive, you will probably get an error message. In this case, insert a \$PBASIC 2.5 directive and try running the program again.

### THE \$PORT DIRECTIVE.

The optional \$PORT directive allows you to indicate a specific PC communications port through which to download a program to a BASIC Stamp module. The syntax is as follows:

```
' {$PORT COM#}
```

where # is a valid port number. When any PBASIC program containing this directive is downloaded, all other ports will be ignored. This directive is especially convenient when using two of the same BASIC Stamp models

## Using the BASIC Stamp Editor

---

(such as two BS2s) on two ports and you have two different PBASIC programs to download (one to each BS2). Without this directive, developing and downloading in this case would be a tedious task of always answering the "which BASIC Stamp?" prompt.

The \$PORT directive can be automatically inserted or modified by selecting the appropriate port from the Directive → Port menu. The COM ports listed in the Directive → Port menu are automatically updated any time a change is made to the exiting computer hardware or to the available ports list. See the Setting Preferences section which begins on page 55 for more information.

### Special Functions

The Identify function will identify which BASIC Stamp model, if any, is detected on any available communications port. This information is displayed in the Identification window (Figure 3.10), which can greatly aid in troubleshooting your connection to your BASIC Stamp module. Activate this function by selecting Run → Identify, by pressing Ctrl-I, or pressing F6.

THE IDENTIFICATION FUNCTION.

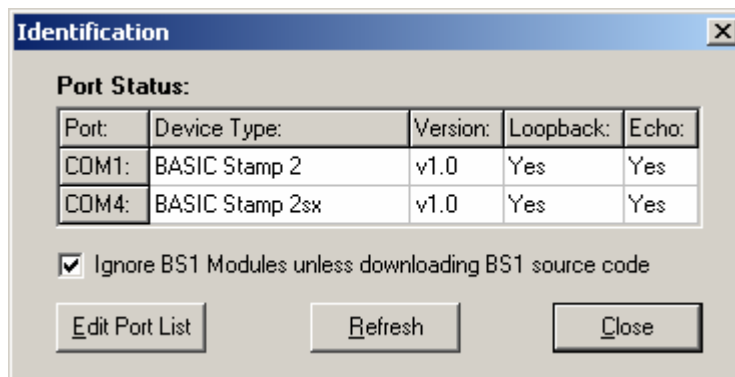


**Figure 3.10:** The Identification Window.

The Port column shows the available ports (those that the BASIC Stamp Editor is trying to access). You can modify the available Port List by clicking on the Edit Port List button. Modifying this list only affects which ports the BASIC Stamp Editor tries to use; it does not affect which serial ports are installed on your computer. It is recommended that you delete all known modem ports and any problematic ports from this list.

---

## 3: Using the BASIC Stamp Editor

The Device Type column shows the model of BASIC Stamp found on the respective port. For example, in Figure 3.10 above, the BASIC Stamp Editor found a BS2 on COM port 1 and a BS2sx on COM port 4.

The Version column displays the firmware version number of the BASIC Stamp module that was found.

NOTE: when using a BS1 Serial Adapter, the Loopback column of the Identification window should always indicate "No".

The Loopback column indicates whether or not a loopback connection was found on the port. The loopback connection is created by BASIC Stamp development boards, such as the Board of Education, across serial port pins 6 and 7 (of a DB9). A "Yes" in this column is an indication that the serial port and serial cable are properly connected to a BASIC Stamp development board. Note that the Loopback column should always indicate "No" when using a BS1 Serial Adapter, regardless of whether or not the adapter is properly connected to a BASIC Stamp development board.

The Echo column indicates whether or not a communication echo was detected on the port's transmit and receive pins (pins 2 and 3). All BASIC Stamp 2 models create this echo naturally, even without power. BASIC Stamp 1 modules do not create this echo. A "Yes" in this column is an indication that the serial port and serial cable are properly connected to a BASIC Stamp 2 (or higher) module, and if using a BASIC Stamp development board, it's an indication that the module is properly connected to the development board.

For all BASIC Stamp 2 models, the Loopback and Echo columns are great for doing some simple connection diagnosis when using a serial port. For example, a Yes in both columns indicates the serial port and serial cable are properly connected and that the BASIC Stamp is properly inserted into its socket. See Table 3.5 below. Note that the Loopback column does not give reliable results when using a USB to serial adapter, or a USB-based development board. Usually this is not an error, and the Loopback status can simply be ignored.

## Using the BASIC Stamp Editor

| Loopback | Echo | Interpretation                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Yes      | Yes  | Serial port and serial cable properly connected. BASIC Stamp properly inserted into socket. If no BASIC Stamp is detected, it is probably because the BASIC Stamp is not connected to power. Other causes could be: 1) low battery, 2) Reset pin of BASIC Stamp is connected to Vdd (it should be left disconnected), 3) the BASIC Stamp is damaged or 4) there is some other type of communication error (software or hardware). |
| Yes      | No   | Serial port and serial cable properly connected to the development board. BASIC Stamp improperly inserted into socket (i.e.: inserted backwards or not inserted at all).                                                                                                                                                                                                                                                          |
| No       | Yes  | Serial port and serial cable may be improperly connected, or you may not be using a standard BASIC Stamp development board. The Echo indicates there may be a BASIC Stamp properly connected to the port (the Loopback is not required for successful connection) or there may be another device connected to the serial port.                                                                                                    |
| No       | No   | Serial port and serial cable are not properly connected, or not connected at all, to the BASIC Stamp 2 (and higher) modules. Could also be an indication of a serial port hardware/software problem. When using a BS1 Serial Adapter, this Loopback and Echo is normal and expected.                                                                                                                                              |

**Table 3.5:** Using Loopback and Echo to troubleshoot your serial port (DB9) connection.

NOTE: When using a USB port, the Loopback column does not give reliable results. Usually this is not an error and the Loopback status can be ignored.

Selecting the “Ignore BS1 Modules unless downloading BS1 source code” checkbox at the bottom of the Identification window optimizes identification speed. All BS2 models can be identified very quickly. For BASIC Stamp 1 modules, the identification process can take as much as five seconds per communications port. Since the Identification function checks all available serial ports for any possible model of BASIC Stamp, the five-second timeout for BS1’s can be very inconvenient, especially if you are not using a BS1.

SPEED UP IDENTIFICATION WITH THE “IGNORE BS1 MODULES” CHECKBOX.

When this checkbox is checked, the Identification function will not attempt to locate BS1 modules, and thus saves time. If, however, you are downloading BASIC Stamp 1 code, the Download function will attempt to locate BS1 modules regardless of the setting of this checkbox. This feature can also be found and modified via the Preferences → Editor Operation tab.

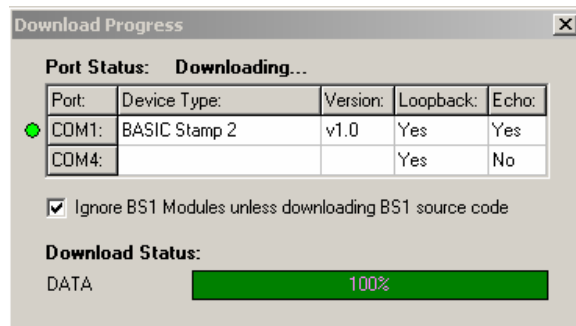
Like the Identification function, the Download function provides information to help guide you through the downloading process. After entering the desired source code in the editor window, you may run it in one of three ways: select Run → Run, press Ctrl+R on the keyboard, or click on the “▶” toolbar icon. This will tokenize and download the code

THE DOWNLOAD FUNCTION.

### 3: Using the BASIC Stamp Editor

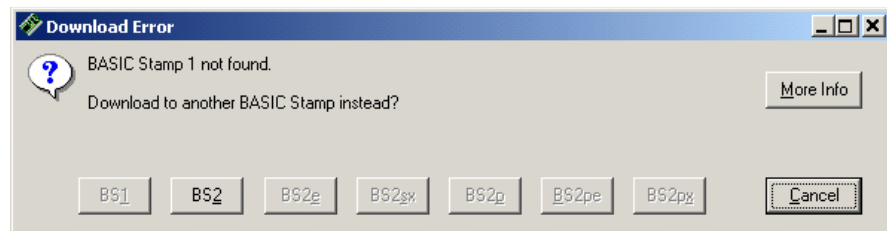
to the BASIC Stamp module (assuming the code is correct and the BASIC Stamp is properly connected). The Download Progress window looks similar to the Identify window with the exception of the additional Download Status progress bar, and the indicator LED by the port transmitting the data.

**Figure 3.11:** The Download Progress Window.



If any errors occur, such as communication failure or inability to detect a BASIC Stamp module, you will be prompted appropriately. One possible error occurs when the BASIC Stamp your PBASIC program is targeting does not appear to be connected to the PC (see Figure 3.12). This may be caused, for example, by opening up a BASIC Stamp 1 program (usually has a .bas or .bs1 extension) and trying to download it to a BASIC Stamp 2 module, instead.

**Figure 3.12:** A Download Error message.



When this happens, you'll be prompted to correct the situation, quickly done by clicking on the BS2 button (if you really intended to download to the BS2 in the first place). Keep in mind that programs written for one BASIC Stamp model may not function properly on a different BASIC Stamp model. Click on the More Info button for more detail. NOTE: If you select the BS2 button, as in this example, the editor will modify the

# Using the BASIC Stamp Editor

\$STAMP directive in the program, notify you of this change and what it means, and then will try to download to the BS2.

Another possibility is having two or more of the same BASIC Stamp model connected to the PC. In this case, the editor will prompt you for clarification as to which BASIC Stamp module you want to download to. In this case using a \$PORT directive in your code will save you some tedium in repeatedly responding to such prompts.

The BASIC Stamp Editor also features a Memory Map that displays the layout of the current PBASIC program. Type Ctrl+M, or press F7, to activate this window.

MEMORY MAP FUNCTION.

When you activate the Memory Map, the editor will check your program for syntax errors and, if the program's syntax is okay, will present you with a color-coded map of the RAM and EEPROM. You'll be able to tell at a glance how much memory you have used and how much remains.

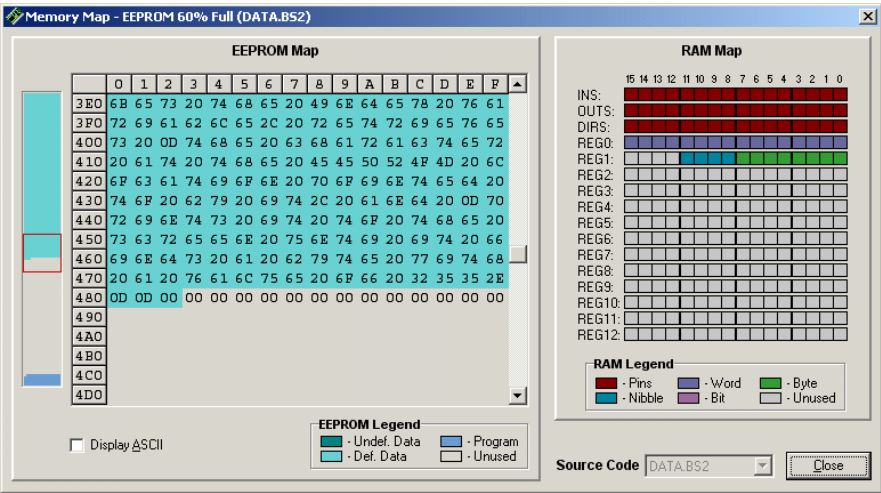


Figure 3.13: Memory Map for Demo Program DATA.bs2.

The Memory Map is divided into two sections, the RAM map and the EEPROM map. The RAM map shows how much of each register has been allotted to program variables. The RAM legend details how much is used by I/O Pins, Word, Byte, Nibble and Bit variables, and how much is unused.

THE RAM MAP.



## 3: Using the BASIC Stamp Editor

---

### THE EEPROM MAP.

The EEPROM map is shown in two scales. The main view is the detailed EEPROM map, which displays the data in hexadecimal format in each location. The condensed EEPROM map is the vertical region on the left that shows a small-scale view of the entire EEPROM; the red square over it corresponds to the scroll bar handle in the detailed EEPROM map and indicates the portion of the EEPROM that is currently visible in the detailed EEPROM map.

Checking the Display ASCII checkbox switches the detailed EEPROM display from hexadecimal to ASCII. In this program, the textual data can be read right off the EEPROM map when using this option.

Two important points to remember about this map are: 1) it only indicates how your program will be downloaded to the BASIC Stamp module; it does not "read" the BASIC Stamp memory, and 2) for all BS2 models, fixed variables like B3 and W1 and any aliases do not show up on the memory map as memory used. The editor ignores fixed variables when it arranges automatically allocated variables in memory. Remember, fixed and allocated variables can overlap.

### THE DEBUG TERMINAL.

The Debug Terminal window provides a convenient display for data received from a BASIC Stamp during run-time, and also allows for the transmission of characters from the PC keyboard to the BASIC Stamp. The Debug Terminal is automatically opened and configured when a PBASIC program, containing a DEBUG command, is downloaded. You can manually open a Debug Terminal one of three ways: select Run → Debug → New, press Ctrl+D on the keyboard, or click on the Debug Terminal toolbar button. Up to four (4) Debug Terminals can be open at once (on four different ports) and all can be left open while editing and downloading source code.

Figure 3.14 below shows the demo program DEBUG\_DEBUGIN.bs2 in the edit pane, and the Debug Terminal that opens when this program is run.

## Using the BASIC Stamp Editor

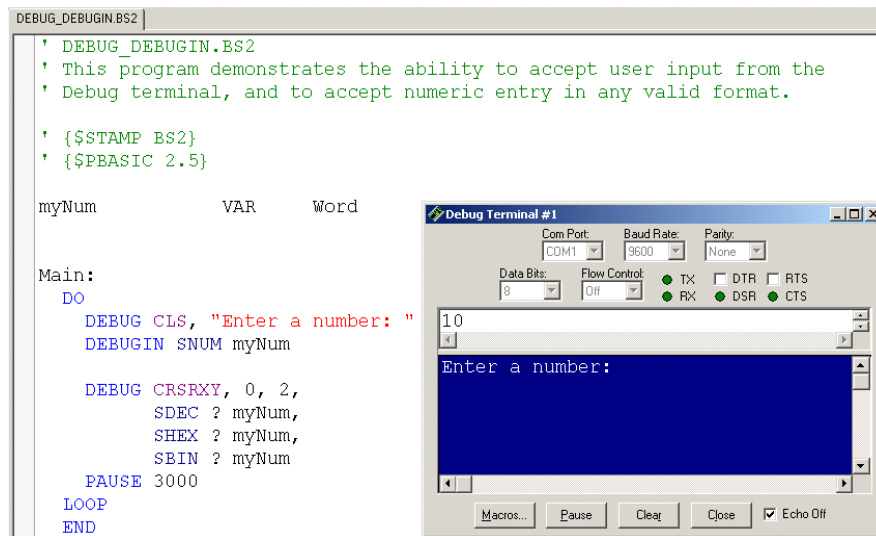


**Figure 3.14:** Demo program using the Debug Terminal

The text in the Debug Terminal's Receive pane (blue area) prompts the user to enter a number into the Transmit pane (white area). After typing the number 10 and pressing Enter, the Receive pane displays the number in decimal, hexadecimal, and binary format as dictated by the program (Figure 3.15).

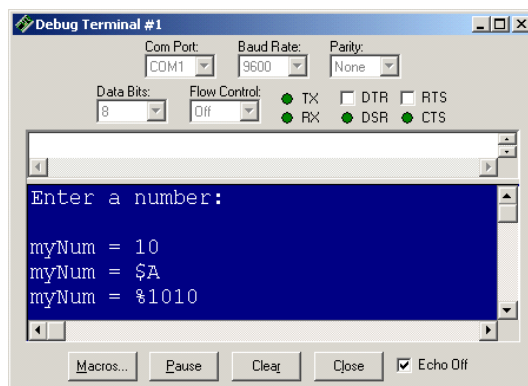


**Figure 3.15:** Debug Terminal output after entering a number.

The fields across the top of the Debug Terminal window allow configuration of the communication port settings. These fields will be automatically configured and disabled if the Debug Terminal was

PORT SETTINGS AND STATUS.

## 3: Using the BASIC Stamp Editor

automatically opened by the editor, however, if manually opened, these fields will be enabled to allow manual configuration. The signal status LEDs turn bright green when activity on the indicated port line is detected. The signal checkboxes (DTR and RTS) can be selected to set or clear the respective output line on the port.

The Echo Off checkbox (bottom of window) causes the Receive pane to throw away the characters that arrive in the port's receive buffer immediately after transmitting characters from the transmit buffer. This produces a cleaner Receive pane display for interactive programs such as the example above. Keep in mind, however, that this feature does not verify that the character it throws away is actually a match to a character that was just transmitted (because data collisions on the port can cause echoed characters to be garbled). You should only use the Echo Off feature in situations where it is required, as it may result in a strange display in certain applications.

### KEYBOARD SHORTCUTS FOR CODING FUNCTIONS.

**Table 3.6:** Coding Function Keyboard Shortcuts.

There are keyboard shortcuts for several coding functions, some of which are unique to the BASIC Stamp Editor.

| Coding Functions               |                                                                                       |
|--------------------------------|---------------------------------------------------------------------------------------|
| Shortcut Key(s)                | Function                                                                              |
| Ctrl+J                         | Show code templates.                                                                  |
| F6 or Ctrl+I                   | Identify BASIC Stamp firmware.                                                        |
| F7 or Ctrl+T                   | Perform a syntax check on the code and display any error messages.                    |
| F8 or Ctrl+M                   | Open Memory Map window.                                                               |
| F9 or Ctrl+R                   | Tokenize code, download to the BASIC Stamp and open Debug window if necessary.        |
| F11 or Ctrl+D                  | Open a new Debug window.                                                              |
| F12                            | Switch to next window (Editor, Debug #1, Debug #2, Debug #3 or Debug #4)              |
| Ctrl+1, Ctrl+2, Ctrl+3, Ctrl+4 | Switch to Debug Terminal #1, Debug Terminal #2, etc. if that Terminal window is open. |
| Ctrl+`                         | Switch to Editor window.                                                              |
| ESC                            | Close current window.                                                                 |

### HELP FILES.

The BASIC Stamp Editor includes searchable, indexed help files. Access Help by selecting Help → Contents or Help → Index. Context sensitive help (highlighting a word in the editor and pressing F1 key) is also supported. The help file can remain open in a separate window while using the BASIC Stamp Editor; simply press Alt+Tab to toggle back and forth between the editor and the Help window.

## Using the BASIC Stamp Editor

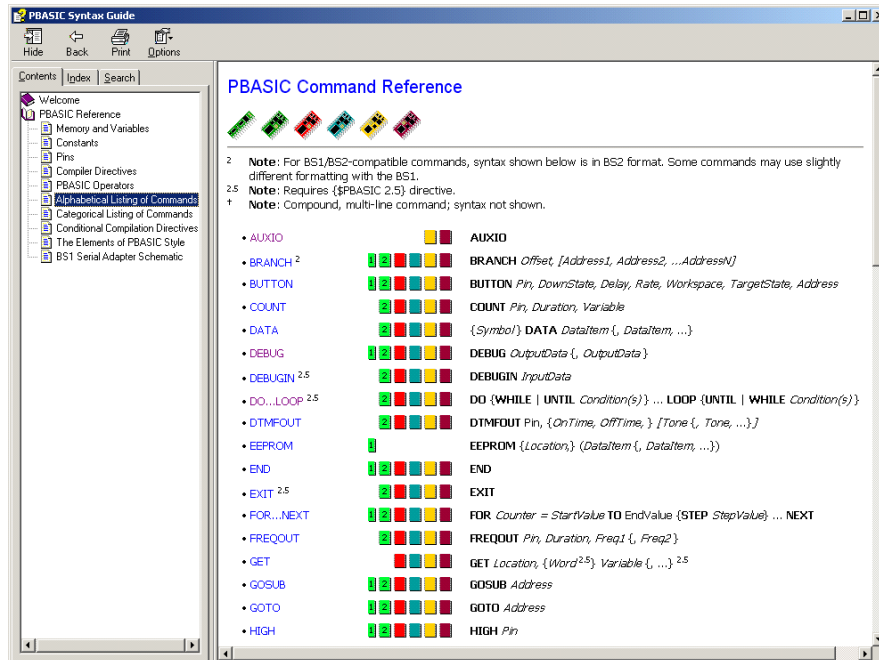


**Figure 3.16:** The Help file contains the complete PBASIC syntax documentation.

The current help files contain the entire PBASIC syntax documentation. In addition, the example demo code programs that appear after most command descriptions in Chapter 5 are automatically placed in default directories during the BASIC Stamp Editor v2.2 installation. These programs can be accessed via hyperlinks within the help file.

NOTE: The BASIC Stamp Editor Help file requires Microsoft's HTML Help utility and Internet Explorer 4.0 or above (IE 6.0 recommended). The proper version of HTML Help is included with Windows 2000 and Windows XP. On other versions of Windows you may have to install or upgrade your HTML Help utility to properly view the Stamp Editor's on-line help. The HTML Help upgrade program (hhupd.exe) is included as part of the BASIC Stamp Editor setup program and the editor will automatically prompt you to run it if it determines you need to upgrade. You can download the latest version of Internet Explorer from Microsoft's web site at [www.microsoft.com](http://www.microsoft.com).

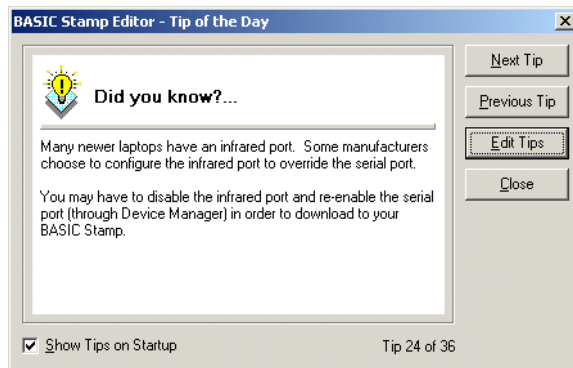
HELP FILES REQUIRE MICROSOFT'S HTML HELP UTILITY.

## 3: Using the BASIC Stamp Editor

### TIP OF THE DAY.

The Tip of the Day function displays a new message each time you run the BASIC Stamp Editor. There are many useful tips, and you may browse through them any time with the Next Tip and Previous Tip buttons. You may also use the Edit Tips option to change the contents of any tip. All tips are contained in a single file, named Stamp\_Tips.txt, that is stored in the editor's installation directory, usually a path similar to C:\Program Files\Parallax Inc\Stamp Editor v2.2.

**Figure 3.17:** Tip of the Day #24.



You can turn this feature off by unchecking the Show Tips on Startup box at the bottom of the window. To access it again, choose Help → Tip of the Day from the menu bar.

### Setting Preferences

The BASIC Stamp Editor allows the user to set preferences for the appearance and operation of many aspects of the application. Select Edit → Preferences, press F5 or click on the Preferences toolbar button to open the Preferences window, where you will see these options organized under 6 tabs. Each tab has a Restore Defaults button in case you make a royal mess of things.

### EDITOR APPEARANCE PREFERENCES.

Under the Editor Appearance tab (Figure 3.18), you can set the font size in the edit pane. The other text attributes, such as background and foreground color, character case, and bold, italic and underline, are controlled by the Syntax Highlighting scheme. There are 3 predefined schemes, and you may also create a custom scheme. Please note that the

## Using the BASIC Stamp Editor

preferences are specific to the editor, and are not saved as part of any BASIC Stamp program that you may have open while setting preferences.

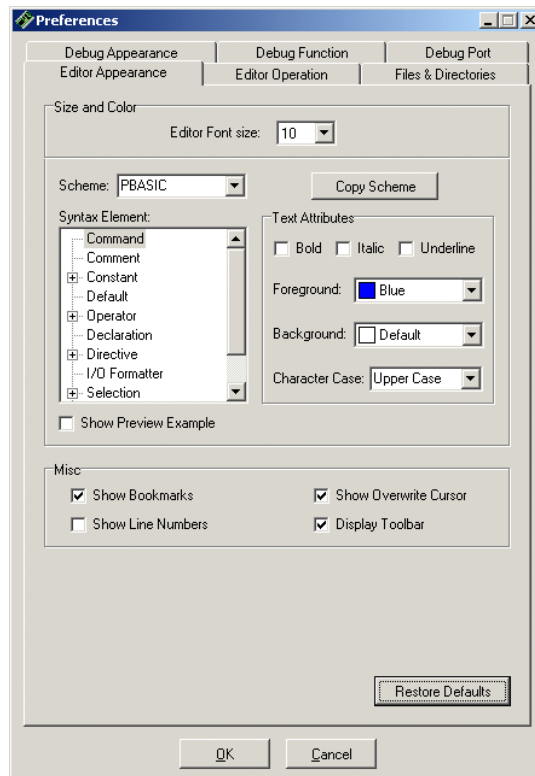


**Figure 3.18:** The Editor Appearance Tab under Edit → Preferences.

The default font size for the edit pane is 10 point, but there are 12 fixed options ranging from 8 to 40 point. The Editor Font size setting and all the other text attribute settings under this tab will not affect the text in the Debug Terminal.

The default scheme is the “PBASIC” scheme, with the syntax highlighting text attributes described above in Table 3.1. The “plain text” scheme is just that – the default foreground and background for all entered text, with no other attributes applied. The “simple” scheme is the same as the “plain text” scheme, except comments appear in green. Both the plain text and simple schemes use the PBASIC scheme defaults for selected text.

SYNTAX HIGHLIGHTING SCHEMES.

## 3: Using the BASIC Stamp Editor

---

### CUSTOMIZED SYNTAX HIGHLIGHTING.

To create a custom scheme, select a default scheme you wish to modify, and click on the Copy Scheme button. Then, select (highlight) an element within the Syntax Element list, and apply new Text Attributes with the checkboxes and drop-down menus to the right. As you try various text attributes and color combinations, the Show Preview Example checkbox lets you audition your custom scheme without closing the Preferences window.

The BASIC Stamp Editor supports one custom scheme at a time. It can be modified indefinitely, but it cannot be copied. If you again copy a default scheme, you will be asked to confirm that you wish to overwrite your current custom scheme.

Under this tab, you will also find checkboxes that allow you to show or hide bookmarks, line numbers, the overwrite cursor, and the toolbar.

### EDITOR OPERATION PREFERENCES.

Under the Editor Operation tab (Figure 3.19), you may set preferences for automatic indentation and tab behavior.

### AUTO INDENTING / UNINDENTING.

The Auto Indent on Enter option makes it easy to indent nested loops to make code easier to read. The Auto Unindent option enables quick reversal of an indented line by simply using the backspace key, provided that the cursor is to the left of the first character on the line.

### TAB CHARACTER.

The editor lets you choose whether a tab character or spaces are inserted into source code whenever you press the Tab key. The default setting, insert space characters upon Tab key presses, is recommended because it enforces the intended formatting regardless of what editor you use to view the code later.

## Using the BASIC Stamp Editor

In addition to the actual character used for the Tab key, there are three behaviors of tabbing employed by the editor: Smart Tabs, Fixed Tabs and Fixed plus Smart Tabs.

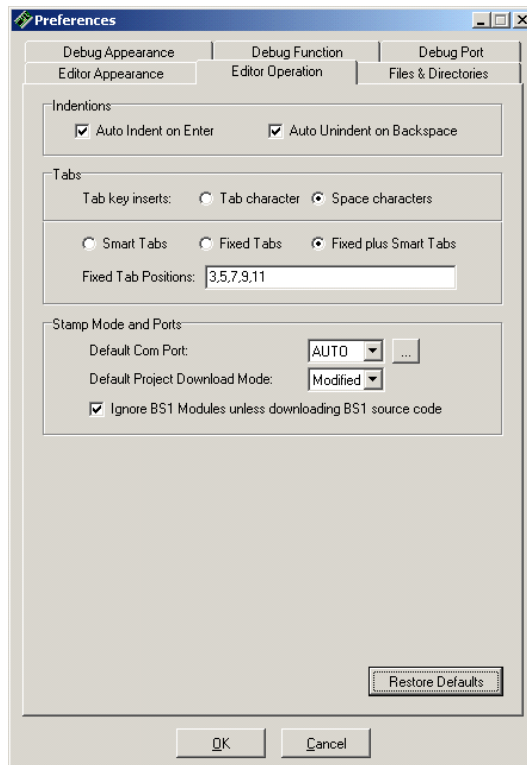


**Figure 3.19:** The Editor Operation Tab under Edit → Preferences.

Smart Tabs (Figure 3.20) cause the tab key to move the cursor to a position that is aligned with the nearest break between words in nearby lines above the current line. It has the effect of providing a somewhat intuitive, auto adjusting behavior based entirely on how you have aligned previous lines.

SMART TABS.

Fixed Tabs (Figure 3.21) cause the tab key to move the cursor to the position indicated by the Fixed Tab Positions field. If the position is already beyond the end of the Fixed Tab Positions list, it moves by a multiple of the distance between the last two positions in that list. For example, with Fixed Tabs set, the default Fixed Tab Positions list will

FIXED TABS.



## FIXED PLUS SMART TABS.

**Figure 3.20: Smart Tabs.**

**Figure 3.21: Fixed Tabs.**

**Figure 3.22:** Fixed plus Smart Tabs.

BASIC Stamp Syntax and Reference Manual 2.2 • [www.parallax.com](http://www.parallax.com) • Page 59

## Using the BASIC Stamp Editor

---

The Fixed Tab Positions list is used to provide a list of desired fixed tab positions (used with Fixed Tabs or Fixed plus Smart Tabs options). The list can be a single number, or a list of comma separated numbers in ascending order. The allowable range is 2 to 512 and the list size is virtually unlimited. When multiple values are entered, the difference between the last two values will be used to set tab positions beyond the last position. For example, in the default list, the last two positions are 9 and 11; resulting in further tab positions of 13, 15, 17, etc. (multiples of 2 after the last specified position). Since source code is usually indented by multiples of two (2) spaces, the default list of 3, 5, 7, 9 and 11 is recommended.

THE FIXED TAB POSITIONS LIST.

The Default Com Port setting allows you to specify which COM port to download through. If you specify a specific port here, the Identification window will report that it is “ignoring” other known ports. This can be selectively overridden by placing a \$PORT directive in the program. If this setting is left on “AUTO”, the default, the editor will open and scan all known ports for the correct BASIC Stamp. The button to the right, labeled ‘...’, opens the a window allowing the known port list to be edited. Modifying the known port list only affects which ports the BASIC Stamp Editor tries to use; it does not affect which serial ports are installed on your computer. It is recommended that you delete all known modem ports and any problematic ports from this list.

DEFAULT COM PORT.

For an explanation of the Default Project Download Modes, see Table 3.7 on page 70. This is part of a discussion on BASIC Stamp Projects in the Advanced Compilation Techniques beginning on page 68, below.

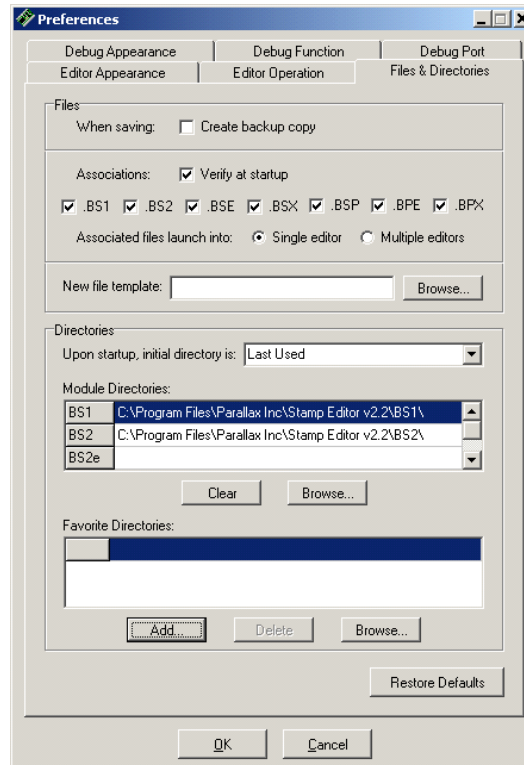
Selecting the “Ignore BS1 Modules unless downloading BS1 source code” checkbox optimizes identification speed by attempting to locate BS1 modules only if you are downloading BASIC Stamp 1 code. This feature can also be activated via the Identification or Download window.

Under the Files and Directories tab (Figure 3.23), you can set preferences for saving and accessing files, as well as automatically creating backup copies.

THE FILES AND DIRECTORIES TAB.

## 3: Using the BASIC Stamp Editor

**Figure 3.23:** The Files and Directories Tab under Edit → Preferences.



### BACKUP COPY.

Check the “Create backup copy” option to cause the editor to automatically create a backup copy of any file that is being re-saved under the same name. The backup file will be stored in the same directory and named the same as the existing file, but with a .bak extension appended to the existing extension. For example, “test.bs2” becomes “test.bs2.bak” and then the new file called “test.bs2” is created from the source code being saved. Note: the .bak files will not appear in the integrated explorer’s file window unless you change the Filter list to show All Files (\*.\*)).

### FILE ASSOCIATIONS.

BASIC Stamp source code file types (.bs1, .bs2, .bse, .bsx, .bsp, .bpe, .bpx) can be associated with the BASIC Stamp Editor. Check the “Verify at startup” option to have the editor verify the proper associations each time it is started. The “Associated files launch into” option changes the way Windows behaves when you open BASIC Stamp source code from any Explorer-shell. Choosing “Single Editor” causes all programs to open up

## Using the BASIC Stamp Editor

---

into a single BASIC Stamp Editor, including an editor that is already running. The “Multiple Editors” option will cause a new BASIC Stamp Editor to open each time you open an associated BASIC Stamp file from any Explorer-shell.

Also, by associating BASIC Stamp source code with the editor, Windows will provide an “Open With Stamp Editor” option when right-clicking on that source code from any Explorer-shell. OPEN WITH STAMP EDITOR OPTION.

The “New file template” field allows you to specify a file to load each time the File → New function is selected. The file will be loaded into the new edit page, but the name will be set to “Untitled#”; where # is an automatically generated number. This feature provides a convenient way to start every new source code project with a specified code template of your choosing. Note that once this feature is set, you may hold down the Shift key while selecting File → New, or clicking the New File toolbar button, to suppress the loading of the code template and thus end up with a blank edit page. NEW FILE TEMPLATE FEATURE.

The “Upon startup, initial directory is” field affects what directory is initially selected in the integrated explorer and the Open and Save As dialog boxes when the editor is started. The default is “Last Used,” meaning the initial directory will be that which was most recently used by the editor. If this setting is changed to “Set Via Shortcut” the editor will initially view the directory indicated by the “Start in” field of the Windows shortcut that launched the editor. The other options include the default module directories and favorite directories. INITIAL DIRECTORY ON STARTUP.

The Module Directories list contains a list of module-specific directories that are called the “default module directories.” Upon installation of the BASIC Stamp Editor software, the PBASIC source code examples in this text are copied to the installation folder and organized into appropriate subfolders. If the default installation folder is used during software installation, the source code files will be copied to a path similar to: C:\Program Files\Parallax Inc\Stamp Editor 2.2\BS1, BS2, BS2sx..., etc. The BASIC Stamp Editor automatically sets its default directories to point to these source code examples, making them immediately available via the File → Open From... and File → Save To... menus as well as the Recent list in the integrated explorer panel. MODULE DIRECTORIES.

## 3: Using the *BASIC Stamp Editor*

---

You may use the Clear and Browse... buttons under the Module Directories list to select new default directory locations for each model of BASIC Stamp. These new folders will then appear as options when you use the File → Open From... and File → Save To... menus as well as the Recent list in the integrated explorer panel.

Note that if you are upgrading from a previous version of the BASIC Stamp Editor and you have set your own default directories, they will not be replaced with the new source code example directories. Upon opening the editor, only default directory options that are blank will be redirected to the source code examples.

### FAVORITE DIRECTORIES.

The Favorite Directories list allows you to add and delete folder locations that will appear as additional options in the File → Open From... and File → Save To... menus as well as the Recent list in the integrated explorer. It works in a similar way as the Module Directories list; however, you can set your own descriptive names for those folders.

### DEBUG APPEARANCE PREFERENCES.

Under the Debug Appearance tab (Figure 3.24) you can set the color and size of the various Debug Terminal elements; settings apply to all the Debug Terminal windows at once. A Debug Terminal itself can be resized and/or moved by simply clicking and dragging the window; each window's size and position is remembered even after closing the editor.

### THE CHOOSE... BUTTONS.

The Choose... buttons allow you to change the background and font color of both the Transmitter and Receiver panes, independently. The font size of both panes can be changed to one of 8 sizes: 6, 8, 10, 12, 14, 18, 24, and 36. The Debug Terminal font size is independent of the font size in the main editor window.

## Using the BASIC Stamp Editor

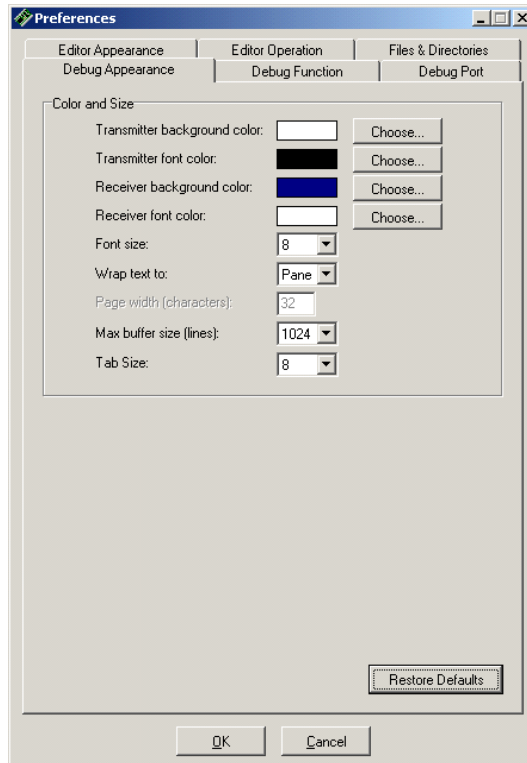


**Figure 3.24:** The Debug Appearance Tab under Edit → Preferences.

The “Wrap Text to” field gives two options, Pane and Page. Wrapping to Pane is the default, and causes text to wrap at the right edge of the Receiver pane, reflecting the current visible size that the user happens to have set for the Debug Terminal’s window. Wrapping to Page, however, causes text to wrap at a specific line width, regardless of the user’s current Debug Terminal window size. The “Page width (characters)” field is enabled when wrap mode is set to Page. The default page width is 32, characters and the range is 32 to 128. Note: wrapping to page can be handy to maintain formatting of formatted tabular information, but could lead to information being displayed off the edge of the Receive pane if the Debug Terminal is sized too small.

TEXT WRAPPING IN THE DEBUG TERMINAL.

The maximum Receive pane buffer size is defined in terms of lines. It can be set to any power of two between 256 and 8192; 1024 is the default. Data received by the Debug Terminal is maintained in this buffer for display on

MAXIMUM BUFFER SIZE.

### ***3: Using the BASIC Stamp Editor***

---

the screen. If the default is used, for example, you could receive 1024 lines worth of text from a BASIC Stamp, and still be able to scroll back and view the first line that was received. Upon receiving the 1025<sup>th</sup> line of text, the first line of text is pushed out of the buffer and is lost for good, making the first visible line in the Receive pane actually be the 2<sup>nd</sup> line of text that was received. Larger buffer sizes consume more PC memory ( $256 * \text{buffer\_size} * \text{Number\_of\_Open\_Debug\_Terminals}$  bytes), so it is best to set it only as high as you need it for your application.

The Tab size can be adjusted as well, anywhere from 3 to 16 character spaces. The default is 8. Keep in mind that most people don't change this value, so writing code that relies on a particular setting other than 8 may display improperly on other user's Debug Terminals.

## Using the BASIC Stamp Editor

---

Under the Debug Function tab (Figure 3.25), checkboxes allow enabling or disabling of special processing for 16 different control characters. The default is for all 16 control characters to be processed, but you may disable one or more of them if you are using the Debug Terminal to view data coming from a device other than a BASIC Stamp.

DEBUG FUNCTION PREFERENCES.

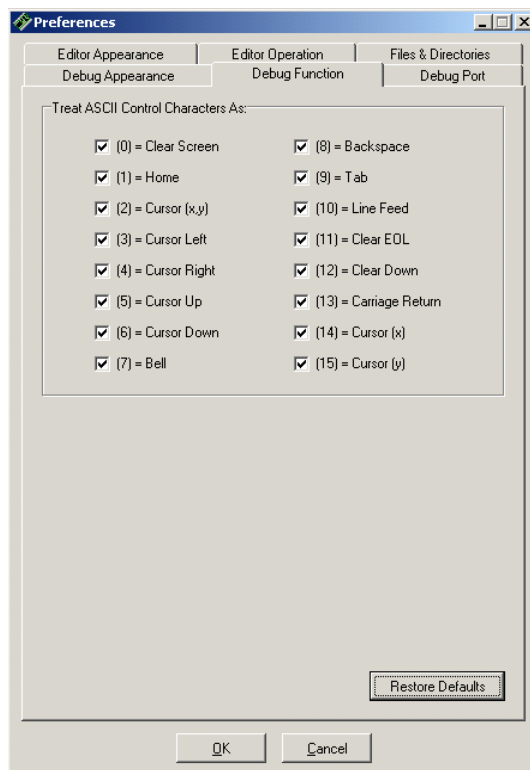


**Figure 3.25:** The Debug Function Tab under Edit → Preferences.

For example, a device that sends out a 0 to indicate something other than Clear Screen will cause unintentional clearing of the Receive pane; unchecking the checkbox for “(0) = Clear Screen” will prevent this from happening.

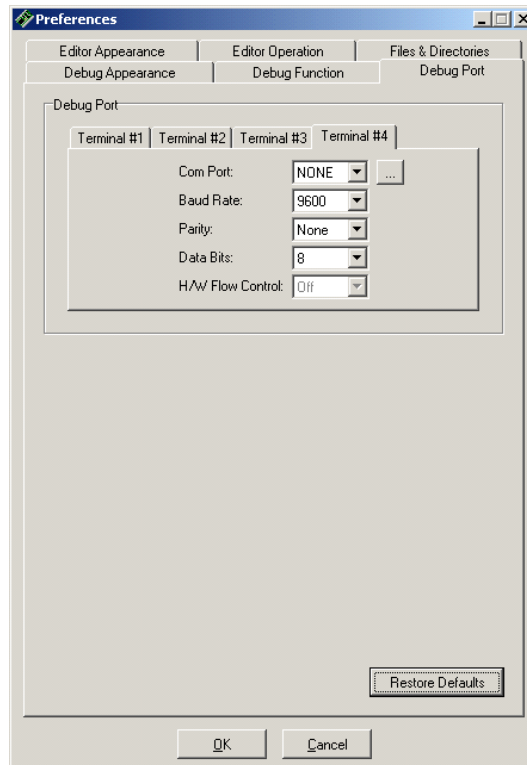


## 3: Using the BASIC Stamp Editor

DEBUG PORT PREFERENCES.

Under the Debug Port tab (Figure 3.26), each of the four (4) Debug Terminal's default COM port settings may be configured separately. These settings are only used when the Debug Terminal is manually opened.

**Figure 3.26:** The Debug Port Tab under Edit → Preferences.



You may assign a specific COM port from the available drop-down list; this list can be changed by clicking on the (...) button to the right. Note that any Debug Terminals that are opened automatically after a PBASIC program is downloaded will always default to the COM port and settings used during download. If NONE is selected as the COM port, the manually opened Debug Terminal will not open any port upon startup, so you will have to manually select the desired COM port from the Debug Terminal window each and every time you open that Debug Terminal.

# Using the BASIC Stamp Editor

---

## Advanced Compilation Techniques

For BS2e, BS2sx, BS2p, BS2pe and BS2px modules, each editor page can be a separate project, or part of a single project. A project is a set of up to eight files that should all be downloaded to the BASIC Stamp for a single application. Each of the files within the project is downloaded into a separate "program slot". Only the BASIC Stamp 2e, 2sx, 2p, 2pe, and 2px modules support multi-file projects.

INTRODUCTION TO BASIC STAMP PROJECTS.

For BASIC Stamp projects (consisting of multiple programs), the \$STAMP directive has an option to specify additional filenames. The syntax below demonstrates this form of the \$STAMP directive:

USING THE \$STAMP DIRECTIVE TO DEFINE MULTI-FILE PROJECTS.

```
' { $STAMP BS2e, file2, file3, ..., file8 }
```

Use this form of the \$STAMP directive if a project, consisting of multiple files, is desired. This form of the directive must be entered only into the first program (to be downloaded into program slot 0). The *file2*, *file3*, etc. items should be the actual name (and optionally the path) of the other files in the project. *File2* refers to the program that should be downloaded into program slot 1, *file3* is the program that should be downloaded into program slot 2, etc. If no path is given, the filename is given the path of program 0 when loading them into the editor.

Up to seven filenames can be included, bringing the total to eight files in the project all together. Upon loading, tokenizing, running or viewing program 0 in the Memory Map, the editor will read the \$STAMP directive, determine if the indicated files exist, will load them if necessary and change their captions to indicate the project they belong to and their associated program number. After the directive is tokenized properly, and all associated files are labeled properly, tokenizing, running or viewing any program in the Memory Map will result in that program's entire project being tokenized, downloaded or viewed.

When program #0 of a multi-file project is opened from diskette, the entire project will be loaded (all referenced files) as well. When a file that is part of a multi-file project is closed, the entire project (all the associated files) will be closed as well.

---

## 3: Using the BASIC Stamp Editor

---

### EASY STEPS TO CREATING MULTI-FILE PROJECTS.

To create a project consisting of multiple files, follow these steps:

1. Create the first file in the editor and save it (we'll call it Sample.bsx). This will be the program that is downloaded into program slot 0.
2. Create at least one other file in the editor and save it also (we'll call it NextProgram.bsx).

Note: At this point the editor tabs will be:

0:Sample.bsx      and      0:NextProgram.bsx.

indicating that there are two unrelated files open "Sample.bsx" and "NextProgram.bsx" and each will be downloaded into program slot 0.

3. Go back to the first program and enter or modify the \$STAMP directive using the project format. Use "NextProgram" as the *File2* argument. For example:

```
' {$STAMP BS2sx, NextProgram.bsx}
```

4. Then tokenize the code by pressing F7 or selecting Run → Check Syntax from the menu.

At this point, the BASIC Stamp Editor will see the \$STAMP directive and realize that this file (Sample.bsx) is the first file in a project and that the second file should be NextProgram.bsx. It will then search for the file on the hard drive (to verify its path is correct), will see that it is already loaded, and then will change the editor tabs to indicate the project relationship. At this point the editor tabs will be:

0:Sample.bsx      and      [Sample] 1:NextProgram.bsx.

indicating that there are two related files open; "Sample.bsx" and "NextProgram.bsx". NextProgram.bsx belongs to the "Sample" project and it will be downloaded into program slot 1 and Sample.bsx will be downloaded into program slot 0.

## Using the BASIC Stamp Editor

The editor has the ability to treat projects as one logical unit and can download each of the associated source code files at once. In order to minimize download time for large projects a Project Download Mode is available in the Preferences window. The available modes are: “Modified” (the default), “All” or “Current” and are explained below. This item only affects download operations for the BS2e, BS2sx, BS2p and BS2pe. See Table 3.7.

PROJECT DOWNLOAD MODES.

| Download Mode      | Function                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Modified (default) | This mode will cause only the source code files that were modified since the last download to be downloaded next time. If no files have been modified since the last download, or the entire project has just been loaded into the editor, all the files will be downloaded next time. This mode decreases the delay during downloading projects and should help speed development and testing. |
| All                | This mode will cause all the source code files to be downloaded each time. This will be noticeably slow with large projects.                                                                                                                                                                                                                                                                    |
| Current            | This mode will cause only the current source code file to be downloaded, ignoring all the others. This mode can be helpful, but can lead to development errors if you forget to download a required program.                                                                                                                                                                                    |

**Table 3.7:** Project Download Modes.

Regardless of the download mode selected, the programs will be downloaded into the program slot indicated in their tab.

Some source code may be suitable for multiple uses but requires changing a set of constants as needed for each case. For example, you may want to run the same program on a BS2 and a BS2sx, but the resolution of time-sensitive commands is different, requiring slight code modifications. Several conditional compile directives exist in PBASIC 2.5 to assist with this situation. Table 3.8 lists the available directives.

CONDITIONAL COMPILE DIRECTIVES.

| Directive           | Function                                                                                                                                                                                                                      |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| #DEFINE             | Allows the programmer to create custom symbols for use within conditional compilation control structures.                                                                                                                     |
| #IF...#THEN...#ELSE | Evaluate Condition and, if it is True, compile the statement(s) following #THEN, otherwise compile the statements following #ELSE.                                                                                            |
| #SELECT...#CASE     | Evaluate Expression and then conditionally compile a block of code based on comparison to Condition(s). If no conditions are found True and a #CASE ELSE block is included, the #CASE #ELSE code statements will be compiled. |
| #ERROR              | Allows the programmer to create a custom error dialog.                                                                                                                                                                        |

**Table 3.8:** Conditional Compile Directives.

NOTE: These directives require PBASIC 2.5.

---

## 3: Using the BASIC Stamp Editor

Lets look at the syntax and examples for each conditional compile directive. For an explanation of syntax conventions, see page 128.

### #DEFINE SYNTAX.

#### #DEFINE *Symbol* { = *Value* }

#DEFINE allows the programmer to create custom, compile-time, symbols for use within conditional compile control structures.

- ***Symbol*** is a unique symbol name that will optionally represent a *Value*.
- ***Value*** is an optional constant/expression specifying the value of *Symbol*. If the value parameter is omitted, *Symbol* is defined as true (-1).

Example:

```
' {$PBASIC 2.5}

#DEFINE DebugMode

#IF DebugMode #THEN DEBUG "Debugging."
STOP
```

In the example above, the #DEFINE statement defines *DebugMode* to be “true” (-1), since there is no *Value* argument provided. The second line is another conditional compile statement, #IF...#THEN (see below for more information) which evaluates the state of *DebugMode*, determines it is true and then allows the following DEBUG statement to be compiled into the program. The last line, STOP, is compiled into the program afterwards. The result of compiling this example is a program with only two executable statements, DEBUG "Debugging", CR and STOP. The real power of this example, however, is more obvious when you comment out, or remove, the #DEFINE line. Look at the next example, below:

```
' {$PBASIC 2.5}

' #DEFINE DebugMode

#IF DebugMode #THEN DEBUG "Dubugging."
STOP
```

Here we commented out the #DEFINE line, effectively removing that line from the program. This means that the symbol *DebugMode* will be undefined, and undefined conditional compile symbols are treated as

## Using the BASIC Stamp Editor

---

False (0). Upon compiling this example, the #IF...#THEN statement will evaluate *DebugMode*, which is False (because it is undefined) and then will not allow the DEBUG statement to be compiled. Only the STOP command will be compiled into the program in this example. This is a very powerful feature for quickly removing many DEBUG statements (or other statements) from a program when you're done developing it, but leaving the possibility of re-enabling all those statements should further maintenance be required at a later time.

The optional *Value* argument can be used, for example, to select modes of operation:

```
' {$PBASIC 2.5}

#DEFINE SystemMode = 2

#IF SystemMode = 1 #THEN
 HIGH 1
#ELSE
 LOW 1
#ENDIF
```

In the example above, the first line defines *SystemMode* to be equal to 2. The #IF...#THEN statement evaluates the state of *SystemMode*, determines it is 2, so the condition is false, and then it skips the statement after #THEN and allows the statement following #ELSE to be compiled into the program.

Note, conditional compile directives are evaluated just before the program is compiled, so variables and named constants cannot be referenced within a conditional compile definition. Compile-time symbols created with #DEFINE can, however, be referenced by conditional compile commands.

**#IF *Condition(s)* #THEN**  
    ***Statement(s)***  
**{ #ELSE**  
    ***Statement(s)***  
**#ENDIF**

#IF...#THEN SYNTAX.

---

## 3: Using the BASIC Stamp Editor

#IF...#THEN is a conditional compile structure similar to the run-time IF...THEN command except that, at compile time, #IF...#THEN evaluates *Condition(s)* and, if it is True, compiles the *Statement(s)* following #THEN, otherwise it compiles the *Statement(s)* following #ELSE.

- **Condition** is a statement that can be evaluated as True or False during compile-time.
- **Statement** is any valid PBASIC instruction.

Example:

```
' {$PBASIC 2.5}

' set Baud for 9600-N81

#IF ($STAMP = BS2sx) OR ($STAMP = BS2p) #THEN
 Baud CON 16624
#ELSE

 #IF ($STAMP = BS2px)
 Baud CON 16780
 #ELSE
 Baud CON 16468
 #ENDIF

#ENDIF
```

In this example, the constant *Baud* is set to an appropriate value for the BASIC Stamp that is specified in the \$STAMP directive (not shown). This code will work with the BS2, BS2e, BS2sx, BS2p, BS2pe, and BS2px.

One important thing to note is that the \$STAMP directive is used here as a compile-time symbol, as if it were defined by #DEFINE. The compiler treats all the editor directives, \$STAMP, \$PBASIC and \$PORT as “defined” compile-time symbols set equal to the respective value used in their declaration. At the time of this writing, using \$PBASIC in this fashion is pointless since the conditional-compile directives are only supported in PBASIC 2.5, and would cause an error if compiled in any other version of the language.

## Using the BASIC Stamp Editor

---

### #SELECT Expression

#SELECT...#CASE SYNTAX.

**#CASE Condition(s)**

**Statement(s)**

**{ #CASE Condition(s)**

**Statement(s)**

**#CASE #ELSE**

**Statement(s) }**

### #ENDSELECT

#SELECT...#CASE is a conditional compile structure similar to the run-time SELECT...CASE command except that, at compile time, #SELECT...#CASE evaluates *Expression* and then conditionally compiles a block of code based on comparison to *Condition(s)*. If no *Conditions* are found to be True and a #CASE #ELSE block is included, the *Statement(s)* in the #CASE #ELSE block will be compiled.

- **Expression** is a statement that can be evaluated as True or False during compile-time.
- **Condition** is a statement, that when compared to *Expression*, can be evaluated as True or False. Multiple conditions within the same CASE can be separated by commas ( , ).
- **Statement** is any valid PBASIC instruction.

Example:

```
' { $PBASIC 2.5 }

#SELECT $STAMP
 #CASE BS2, BS2e, BS2sx
 GOSUB LCD_Write
 #CASE #ELSE
 LCDOUT LCDpin, cmd, [char]
#ENDSELECT
```

This example checks the \$STAMP directive at compile-time and either compiles

```
GOSUB LCD_Write
```

- or -

LCDOUT LCDpin, cmd, [char] into the program.



## 3: Using the BASIC Stamp Editor

#ERROR SYNTAX.

### #ERROR Message

#ERROR displays a compile-time error. This allows the programmer to flag fatal errors during compilation.

- **Message** is the error message string, enclosed in quotes.

Example:

```
' {$PBASIC 2.5}

#DEFINE I2CReady = (($STAMP = BS2p) OR ($STAMP = BS2pe) OR ($STAMP = BS2px))

#IF NOT I2CReady #THEN
 #ERROR "BS2p, BS2pe, or BS2px is required for this program."
#ENDIF
```

When compiled, this example will cause the editor to halt compilation and display the dialog below if you attempt to compile for a BASIC Stamp model other than the BS2p, BS2pe, or BS2px:

**Figure 3.27:** Custom Error Message using the #ERROR directive.



### Features for Developers

The BASIC Stamp Editor has several features that are designed to support the needs of developers. Note: when installing the BASIC Stamp editor, you can instruct the installer to include additional developer resources by selecting the “Custom” option from the “Setup Type” prompt.

GENERATE OBJECT CODE FEATURE.

The Generate Object Code feature allows you to tokenize a PBASIC program and save it to a file in the tokenized form. This allows you to send your BASIC Stamp object code (the actual binary data that is downloaded to the BASIC Stamp module) to other people without having to reveal your PBASIC source code. If you are a developer who has

## ***Using the BASIC Stamp Editor***

---

customers using BASIC Stamp-based products, you can release firmware updates to them in this manner.

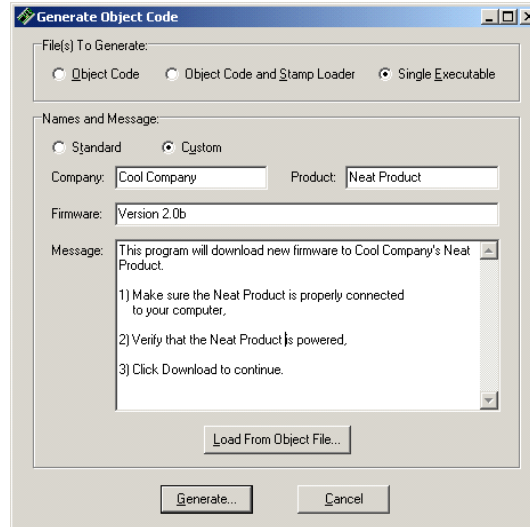
Object code can be saved as a separate .obj file (downloadable with the StampLoader.exe program) or as a single executable (integrated with the StampLoader.exe inside of it). The single executable method provides a simpler way to pass your firmware update on to your customers.

Any syntactically correct PBASIC source code can be used with the Generate Object Code feature; this includes BS1 and BS2 code as well as BS2e, BS2sx, BS2p, BS2pe, and BS2px code that is either a single file or a multi-file project. Note: The original DOS-based software for the BS1 included a directive called BSAVE; when used it would cause the software to generate an object file. In the BASIC Stamp Windows Editor, the Generate Object Code feature replaces and enhances the BSAVE feature; the reserved word BSAVE is still accepted in BS1 source code, but is simply ignored. Old BS1 object code saved via the BSAVE option is not compatible with the StampLoader.exe program so you must regenerate the object file using the BASIC Stamp Windows Editor.

If you don't have the StampLoader.exe program, it can be automatically generated for you by selecting the second output file option, "Object Code and Stamp Loader", in the Generate Object Code window. Additionally, firmware, product, company and related info can be embedded in the object code or single executable file for your customers to view before downloading.

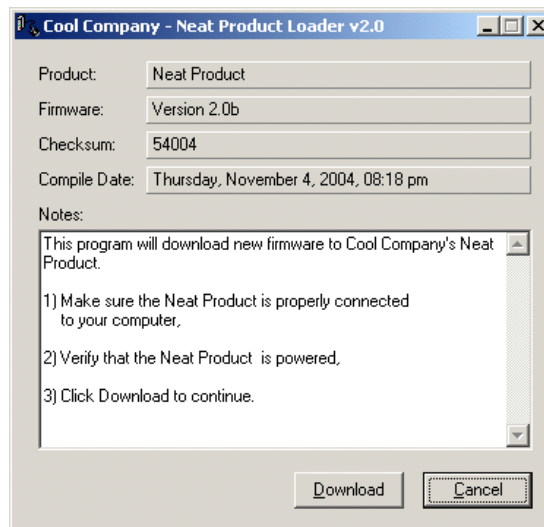
### 3: Using the BASIC Stamp Editor

**Figure 3.28:** The Generate Object Code Window.



In the example above, we chose to generate a single executable with custom names and messages as shown. Then we clicked the Generate... button (which prompted us for a file name) and the file was created. When a user runs the file we just generated, they see a screen similar to the figure below:

**Figure 3.29:** Example customized StampLoader.exe file.



## Using the BASIC Stamp Editor

---

Another feature of interest to some developers is the BASIC Stamp Editor's Command Line interface. This interface provides for command line, batch file or third-party driven control of the editor.

The Stampw.exe program is a Win32 application that can be run through any standard method. When run with the command-line options, however, it provides special features that developers and product manufacturing managers may find useful. While this program can be run from a command prompt on a Windows system, it will not work on a DOS-only system.

The Stampw.exe supports redirection of its input and output via the standard pipe mechanisms. If its output is redirected via the command-line (ex: Stampw.exe myfile.bs2 > Test1.txt) the designated output file, *Test1.txt* in this case, will be created and various information about the processing of the source file will be stored there. This information directly reflects the information available on the GUI prompts, interactions with the user and downloading status. This feature can be combined with the /NoDebug and /NoPrompts switches for various levels of GUI interaction with the user; including completely hidden operation.

The following is the syntax of the BASIC Stamp Editor's command-line switches.

```
Stampw.exe {/Com#} {{/ReadOnly} source_file}
Stampw.exe {/Com#} /Download {/Updates}{/NoDebug}{/NoPrompts} source_file > output_file
Stampw.exe {/Com#} /Identify {/NoPrompts} > output_file
Stampw.exe /Tokenize source_file > output_file
Stampw.exe /Pipe master_file
Stampw.exe /Help
```

Table 3.9 gives a function description for each command-line switch.

## 3: Using the BASIC Stamp Editor

**Table 3.9:** Command Line Switches

| Command              | Function                                                                                                                                                                                                                                                                                                             |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| /Com#                | Specify com port (serial port) to download to. # is a valid com port number. NOTE: must be one word, i.e.: Com2 indicates com port 2.                                                                                                                                                                                |
| /ReadOnly            | Open <i>source_file</i> in read-only mode. The Ctrl key acts as a download key when in read-only mode. Requires <i>source_file</i> argument. This command option is not available if double-piped communication is established.                                                                                      |
| /Identify            | Identify BASIC Stamp modules on COM ports. Requires redirection to <i>output_file</i> .                                                                                                                                                                                                                              |
| /Tokenize            | Tokenize source code. No prompts will be displayed. Requires <i>source_file</i> argument and redirection to <i>output_file</i> .                                                                                                                                                                                     |
| /Download            | Tokenize source code, and download it (if tokenization successful). Requires <i>source_file</i> argument and redirection to <i>output_file</i> .                                                                                                                                                                     |
| /Updates             | Provides program slot number (if applicable) and download-percentage-complete status updates during download.                                                                                                                                                                                                        |
| /NoDebug             | No Debug Terminal opens after downloading (even if code contains DEBUGs) and COM port is immediately closed after downloading. This option requires /Download switch. Note: This switch will have no effect if Debug Terminal is already open from a previous operation.                                             |
| /NoPrompts           | No screen prompts at all (except for Debug Terminal). This option requires /Download switch.                                                                                                                                                                                                                         |
| /Pipe<br>master_file | Start up <i>master_file</i> (must be .exe) and establish bi-directional communication pipes (double-piped communication) for master-program-controlled execution. Stampw.exe remains open until <i>master_file</i> breaks pipe. This command option is not available once double-piped communication is established. |
| /Help                | Display command-line help. This command option is not available if double-piped communication is established.                                                                                                                                                                                                        |

When the output of the BASIC Stamp Editor is piped to a file or a master program, it displays all of its messages in a specific, predefined format. Each message has a unique 3-digit number. Detailed information about the use of command-line options, including a table of all messages with their ID numbers, can be found in the “What’s New in Stamp.exe” document included with the typical installation of the Stamp Editor.

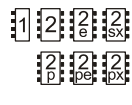
## ***Using the BASIC Stamp Editor***

---

## 4: BASIC Stamp Architecture – Memory Organization

**BASIC Stamp Architecture Introduction** This chapter provides detail on the architecture (RAM usage) and math functions of the BS1, BS2, BS2e, BS2sx, BS2p, BS2pe, and BS2px.

The following icons will appear to indicate where there are differences among the various BASIC Stamp models:

 One or more of these icons indicates the item applies only to the BS1, BS2, BS2e, BS2sx, BS2p, BS2pe, or BS2px respectively.

 If an item applies to all of the models in the BS2 family, this icon is used.

### MEMORY ORGANIZATION

The BASIC Stamp has two kinds of memory; RAM (for variables used by your program) and EEPROM (for storing the program itself). EEPROM may also be used to store long-term data in much the same way that desktop computers use a hard drive to hold both programs and files.

An important distinction between RAM and EEPROM is this:

- RAM loses its contents when the BASIC Stamp loses power; when power returns, all RAM locations are cleared to 0s.
- EEPROM retains the contents of memory, with or without power, until it is overwritten (such as during the program-downloading process or with a WRITE instruction.)

### RAM ORGANIZATION (BS1)

 The BS1 has 16 bytes (8 words) of RAM space arranged as shown in Table 4.1. The first word, called PORT, is used for I/O pin control. It consists of two bytes, PINS and DIRS. The bits within PINS correspond to each of the eight I/O pins on the BS1. Reading PINS effectively reads the I/O pins directly, returning an 8-bit set of 1's and 0's corresponding to the high and low state of the respective I/O pin at that moment. Writing to PINS will store a high or low value on the respective I/O pins (though only on pins that are set to outputs).

### THE INPUT/OUTPUT VARIABLES.

The second byte of PORT, DIRS, controls the direction of the I/O pins. Each bit within DIRS corresponds to an I/O pin's direction. A high bit (1)

## BASIC Stamp Architecture – Memory Organization

sets the corresponding I/O pin to an output direction and a low bit (0) sets the corresponding I/O pin to an input direction.

The remaining words (W0 – W6) are available for general-purpose use. Each word consists of separately addressable bytes and the first two bytes (B0 and B1) are bit addressable as well.

You may assign other names (symbols) to these RAM registers as shown in section "Defining and Using Variables", below.

When the BS1 is powered up, or reset, all memory locations are cleared to 0, so all pins are inputs (DIRS = %00000000). Also, if the PBASIC program sets all the I/O pins to outputs (DIRS = %11111111), then they will initially output low, since the output latch (PINS) is cleared to all zeros upon power-up or reset, as well.

| Word Name | Byte Names   | Bit Names                   | Special Notes                                                       |
|-----------|--------------|-----------------------------|---------------------------------------------------------------------|
| PORT      | PINS<br>DIRS | PIN0 – PIN7<br>DIR0 – DIR7  | I/O pins; bit addressable.<br>I/O pins directions; bit addressable. |
| W0        | B0<br>B1     | BIT0 – BIT7<br>BIT8 – BIT15 | Bit addressable.<br>Bit addressable.                                |
| W1        | B2<br>B3     |                             |                                                                     |
| W2        | B4<br>B5     |                             |                                                                     |
| W3        | B6<br>B7     |                             |                                                                     |
| W4        | B8<br>B9     |                             |                                                                     |
| W5        | B10<br>B11   |                             |                                                                     |
| W6        | B12<br>B13   |                             | Used by GOSUB instruction.<br>Used by GOSUB instruction.            |

**Table 4.1:** BS1 RAM Organization.

Note: There are eight words, consisting of two bytes each for a total of 16 bytes. The bits within the upper two words are individually addressable.

The BS2, BS2e, and BS2sx models have 32 bytes of Variable RAM space arranged as shown in Table 4.2. Of these, the first six bytes are reserved for input, output, and direction control of the I/O pins. The remaining 26 bytes are available for general-purpose use as variables.

**Table 4.2:** RAM ORGANIZATION.



The BS2p, BS2pe, and BS2px models have an extra set of INS, OUTS, and DIRS registers for a total of 38 bytes of variable RAM. These are “shadow” registers that are switched in and out of the memory map with the AUXIO, MAINIO, and IOTERM commands. While this feature exists in





## 4: BASIC Stamp Architecture – Memory Organization

the variable RAM for these models, only the BS2p40 module has the extra 16 I/O pins for which this feature is intended.

### THE INPUT/OUTPUT VARIABLES.

The word variable INS is unique in that it is read-only. The 16 bits of INS reflect the state of I/O pins P0 through P15. It may only be read, not written. OUTS contains the states of the 16 output latches. DIRS controls the direction (input or output) of each of the 16 I/O pins.

A 0 in a particular DIRS bit makes the corresponding pin an input and a 1 makes the corresponding pin an output. So if bit 5 of DIRS is 0 and bit 6 of DIRS is 1, then I/O pin 5 (P5) is an input and I/O pin 6 (P6) is an output. A pin that is an input is at the mercy of circuitry outside the BASIC Stamp; the BASIC Stamp cannot change its state. A pin that is an output is set to the state indicated by the corresponding bit of the OUTS register.

When the BASIC Stamp is powered up, or reset, all memory locations are cleared to 0, so all pins are inputs (DIRS = %0000000000000000). Also, if the PBASIC program sets all the I/O pins to outputs (DIRS = %1111111111111111), then they will initially output low, since the output latch (OUTS) is cleared to all zeros upon power-up or reset, as well.

**Table 4.2:** RAM Organization for all BS2 models.

NOTE: There are 16 words, of two bytes each for a total of 32 bytes\*. All bits are individually addressable through variable modifiers; the bits within the upper three words are also individually addressable through the pre-defined names shown. All registers are word, byte, nibble and bit addressable.

\*The BS2p, BS2pe, and BS2px have an additional set of INS, OUTS, and DIRS registers that are switched in and out of the memory map in place of the main INS, OUTS, and DIRS registers by using AUXIO, MAINIO, and IOTERM. Only the BS2p40 has the required extra I/O pins this feature is intended for.

| Word Name | Byte Names | Nibble Names             | Bit Names                   | Special Notes             |
|-----------|------------|--------------------------|-----------------------------|---------------------------|
| INS*      | INL, INH   | INA, INB<br>INC, IND     | IN0 – IN7<br>IN8 – IN15     | Input pins                |
| OUTS*     | OUTL, OUTH | OUTA, OUTB<br>OUTC, OUTD | OUT0 – OUT7<br>OUT8 – OUT15 | Output pins               |
| DIRS*     | DIRL, DIRH | DIRA, DIRB<br>DIRC, DIRD | DIR0 – DIR7<br>DIR8 – DIR15 | I/O pin direction control |
| W0        | B0, B1     |                          |                             |                           |
| W1        | B2, B3     |                          |                             |                           |
| W2        | B4, B5     |                          |                             |                           |
| W3        | B6, B7     |                          |                             |                           |
| W4        | B8, B9     |                          |                             |                           |
| W5        | B10, B11   |                          |                             |                           |
| W6        | B12, B13   |                          |                             |                           |
| W7        | B14, B15   |                          |                             |                           |
| W8        | B16, B17   |                          |                             |                           |
| W9        | B18, B19   |                          |                             |                           |
| W10       | B20, B21   |                          |                             |                           |
| W11       | B22, B23   |                          |                             |                           |
| W12       | B24, B25   |                          |                             |                           |

## ***BASIC Stamp Architecture – Memory Organization***

---

The INS variable always shows the state of the I/O pins themselves, regardless of the direction of each I/O pin. We call this, "reading the pins". If a pin was set to an input mode (within DIRS) and an external circuit connected the I/O pin to ground, the corresponding bit of INS would be low. If a pin was set to an output mode and the pin's state was set to a high level (within OUTS), the corresponding bit of INS would be high. If, however, that same pin was externally connected directly to ground, the corresponding bit of INS would be low; since we're reading the state of the pin itself and the BASIC Stamp cannot override a pin that is driven to ground or 5 volts externally. Note: The last example is an error, is a direct short and can cause damage to the BASIC Stamp! Do not intentionally connect output pins directly to an external power source or you risk destroying your BASIC Stamp.

To summarize: DIRS determines whether a pin's state is set by external circuitry (input, 0) or by the state of OUTS (output, 1). INS always matches the actual states of the I/O pins, whether they are inputs or outputs. OUTS holds bits that will only appear on pins whose DIRS bits are set to output.

SUMMARY OF THE FUNCTION OF DIRS, INS AND OUTS.

In programming the BASIC Stamp, it's often more convenient to deal with individual bytes, nibbles or bits of INS, OUTS and DIRS rather than the entire 16-bit words. PBASIC has built-in names for these elements, shown in Table 4.2.

Here's an example of what is described in Table 4.2. The INS register is 16-bits (corresponding to I/O pins 0 through 15). The INS register consists of two bytes, called INL (the Low byte) and INH (the High byte). INL corresponds to I/O pins 0 through 7 and INH corresponds to I/O pins 8 through 15. INS can also be thought of as containing four nibbles, INA, INB, INC and IND. INA is I/O pins 0 through 3, INB is I/O pins 4 through 7, etc. In addition, each of the bits of INS can be accessed directly using the names IN0, IN1, IN2... IN15.

The same naming scheme holds true for the OUTS and DIRS variables as well.

As Table 4.2 shows, the BASIC Stamp module's memory is organized into 16 words of 16 bits each. The first three words are used for I/O. The remaining 13 words are available for use as general-purpose variables.

PREDEFINED "FIXED" VARIABLES.

## 4: BASIC Stamp Architecture – Defining Variables

 The 40-pin BS2p uses the first three words for I/O even though it has twice as many I/O pins. This is done with the AUXIO, MAINIO, and IOTERM commands, which effectively switch the auxiliary I/O registers in and out of the INS, OUTS, and DIRS locations.

Just like the I/O variables, the general-purpose variables have predefined names: W0 through W12 and B0 through B25. B0 is the low byte of W0; B1 is the high byte of W0; and so on through W12 (B24=low byte, B25=high byte). Unlike I/O variables, there's no reason that your program variables have to be stuck in a specific position in the BASIC Stamp's physical memory. A byte is a byte regardless of its location. And if a program uses a mixture of variables of different sizes, it can be difficult to logically dole them out or allocate storage.

More importantly, mixing fixed variables with automatically allocated variables (discussed in the next section) is an invitation to bugs. A fixed variable can overlap an allocated variable, causing data meant for one variable to show up in another! The fixed variable names (of the general-purpose variables) are only provided for power users who require absolute access to a specific location in RAM.

We recommend that you avoid using the fixed variables in most situations. Instead, let PBASIC allocate variables as described in the next section. The editor software will organize your storage requirements to make optimal use of the available memory.

### DEFINING AND USING VARIABLES (VAR).

 Before you can use a variable in a PBASIC program you must declare it. "Declare" means letting the BASIC Stamp know that you plan to use a variable, what you want to call it, and how big it is. Although PBASIC does have predefined variables that you can use without declaring them first (see previous sections), the preferred way to set up variables is to use the directive SYMBOL (for the BS1) or VAR (for all BS2 models). Here is the syntax for a variable declaration:

 SYMBOL     name   =   RegisterName

-- OR --

 name VAR   Size

where *name* is the name by which you will refer to the variable, *RegisterName* is the "fixed" name for the register and *size* indicates the

## BASIC Stamp Architecture – Defining Variables

number of bits of storage for the variable. NOTE: The top example is for the BS1 and the bottom example is for all BS2 models.

There are certain rules regarding symbol names. Symbols must start with a letter or underscore, can contain a mixture of letters, numbers, and underscore ( ) characters, and must not be the same as PBASIC reserved words, or labels used in your program. Additionally, symbols can be up to 32 characters long. See Appendix B for a list of PBASIC reserved words. PBASIC does not distinguish between upper and lower case, so the names MYVARIABLE, myVariable, and MyVaRiAbLe are all equivalent.

THE RULES OF SYMBOL NAMES.

For the BS1, the register name is one of the predefined "fixed" variable names, such as W0, W1, B0, B1, etc. Here are a few examples of variable declarations on the BS1:

```
SYMBOL temporary = W0 ' value can be 0 to 65535
SYMBOL counter = B1 ' value can be 0 to 255
SYMBOL result = B2 ' value can be 0 to 255
```

The above example will create a variable called *temporary* whose contents will be stored in the RAM location called W0. Also, the variable *counter* will be located at RAM location B1 and *result* at location B2. Note that *temporary* is a word-sized variable (because that's what size W0 is) while the other two are both byte-sized variables. Throughout the rest of the program, we can use the names *temporary*, *counter*, and *result* instead of W0, B1 and B2, respectively. This makes the code much more readable; it's easier to determine what *counter* is used for than it would be to figure out what the name B1 means. Please note that *counter* resides at location B1, and B1 happens to be the high byte of W0. This means that changing *counter* will also change *temporary* since they overlap. A situation like this usually is a mistake and results in strange behavior, but is also a powerful feature if used carefully.

For all BS2 models, the *Size* argument has four choices: 1) Bit (1 bit), 2) Nib (nibble; 4 bits), 3) Byte (8 bits), and 4) Word (16 bits). Here are some examples of variable declarations on the BS2 models:

```
mouse VAR BIT ' Value can be 0 or 1.
cat VAR NIB ' Value can be 0 to 15.
dog VAR BYTE ' Value can be 0 to 255.
rhino VAR WORD ' Value can be 0 to 65535.
```

## 4: BASIC Stamp Architecture – Defining Arrays

The above example will create a bit-sized variable called *mouse*, and nibble-sized variable called *cat*, a byte-sized variable called *dog* and a word-sized variable called *rhino*. Unlike in the BS1, these variable declarations don't point to a specific location in RAM. Instead, we only specified the desired size for each variable; the BASIC Stamp will arrange them in RAM as it sees fit. Throughout the rest of the program, we can use the names *mouse*, *cat*, *dog* and *rhino* to set or retrieve the contents of these variables.

A variable should be given the smallest size that will hold the largest value that will ever be stored in it. If you need a variable to hold the on/off status (1 or 0) of switch, use a bit. If you need a counter for a FOR...NEXT loop that will count from 1 to 100, use a byte. And so on.

If you assign a value to a variable that exceeds its size, the excess bits will be lost. For example, suppose you use the byte variable *dog*, from the example above, and write *dog* = 260 (%100000100 binary). What will *dog* contain? It will hold only the lowest 8 bits of 260: %00000100 (4 decimal).

### DEFINING ARRAYS.



On all BS2 models, you can also define multipart variables called arrays. An array is a group of variables of the same size, and sharing a single name, but broken up into numbered cells, called elements. You can define an array using the following syntax:

```
name VAR Size (n)
```

where *name* and *Size* are the same as described earlier. The new argument, (n), tells PBASIC how many elements you want the array to have. For example:

```
myList VAR Byte(10) ' Create a 10-byte array.
```

Once an array is defined, you can access its elements by number. Numbering starts at 0 and ends at n-1. For example:

```
myList(3) = 57
DEBUG ? myList(3)
```

This code will display "myList(3) = 57" on the PC screen. The real power of arrays is that the index value can be a variable itself. For example:

## BASIC Stamp Architecture – Defining Arrays

---

```
myBytes VAR Byte(10) ' Define 10-byte array
idx VAR Nib ' Define 4-bit var

FOR idx = 0 TO 9 ' Repeat with idx = 0, 1, 2...9
 myBytes(idx) = idx * 13 ' Write idx * 13 to each cell
NEXT

FOR idx = 0 TO 9 ' Repeat with idx = 0, 1, 2...9
 DEBUG ? myBytes(idx) ' Show contents of each cell
NEXT
STOP
```

If you run this program, DEBUG will display each of the 10 values stored in the elements of the array: myBytes(0) = 0\*13 = 0, myBytes(1) = 1\*13 = 13, myBytes(2) = 2\*13 = 26 ... myBytes(9) = 9\*13 = 117.

A word of caution about arrays: If you're familiar with other BASICs and have used their arrays, you have probably run into the "subscript out of range" error. Subscript is another term for the index value. It is out-of-range when it exceeds the maximum value for the size of the array. For instance, in the example above, myBytes is a 10-cell array. Allowable index numbers are 0 through 9. If your program exceeds this range, PBASIC will not respond with an error message. Instead, it will access the next RAM location past the end of the array. If you are not careful about this, it can cause all sorts of bugs.

If accessing an out-of-range location is bad, why does PBASIC allow it? Unlike a desktop computer, the BASIC Stamp doesn't always have a display device connected to it for displaying error messages. So it just continues the best way it knows how. It's up to the programmer (you!) to prevent bugs. Clever programmers, can take advantage of this feature, however, to perform tricky effects.

Another unique property of PBASIC arrays is this: You can refer to the 0th cell of the array by using just the array's name without an index value. For example:

```
myBytes VAR Byte(10) ' Define 10-byte array

myBytes(0) = 17 ' Store 17 to 0th cell
DEBUG ? myBytes(0) ' Display contents of 0th cell
DEBUG ? myBytes ' Also displays 0th cell
```



## 4: BASIC Stamp Architecture – Aliases and Modifiers

This feature is how the "string" capabilities of the DEBUG and SEROUT command expect to work. A string is simply a byte array used to store text. See "Displaying Strings (Byte Arrays)" in the DEBUG command description on page 166 for more information.

### ALIASES AND VARIABLE MODIFIERS.

An *alias* is an alternative name for an existing variable. For example:

```
1 SYMBOL cat = B0 ' Create a byte-sized variable
 SYMBOL tabby = cat ' Create alias for cat
```

-- OR --

```
All 2 cat VAR Byte ' Create a byte-sized variable
 tabby VAR cat ' Create alias for cat
```

In this example, *tabby* is an alias to the variable *cat*. Anything stored in *cat* shows up in *tabby* and vice versa. Both names refer to the same physical piece of RAM. This kind of alias can be useful when you want to reuse a temporary variable in different places in your program, but also want the variable's name to reflect its function in each place. Use caution, because it is easy to forget about the aliases; during debugging, you might end up asking 'How did that value get here?!' The answer is that it was stored in the variable's alias.

All 2 On all the BS2 models, an alias can also serve as a window into a portion of another variable. This is done using "modifiers." Here the alias is assigned with a modifier that specifies what part to reference:

```
rhino VAR Word ' A 16-bit variable
head VAR rhino.HIGHBYTE ' Highest 8 bits of rhino
tail VAR rhino.LOWBYTE ' Lowest 8 bits of rhino
```

Given that example, if you write the value %1011000011111101 to *rhino*, then *head* would contain %10110000 and *tail* would contain %11111101.

Table 4.3 lists all the variable modifiers. PBASIC 2.0 and 2.5 lets you apply these modifiers to any variable name and to combine them in any fashion that makes sense. For example, it will allow:

```
rhino VAR Word ' A 16-bit variable
eye VAR rhino.HIGHBYTE.LOWNIB.BIT1 ' A bit
```

## BASIC Stamp Architecture – Aliases and Modifiers

| Symbol         | Definition                          |
|----------------|-------------------------------------|
| LOWBYTE        | low byte of a word                  |
| HIGHBYTE       | high byte of a word                 |
| BYTE0          | byte 0 (low byte) of a word         |
| BYTE1          | byte 1 (high byte) of a word        |
| LOWNIB         | low nibble of a word or byte        |
| HIGHNIB        | high nibble of a word or byte       |
| NIB0           | nib 0 of a word or byte             |
| NIB1           | nib 1 of a word or byte             |
| NIB2           | nib 2 of a word                     |
| NIB3           | nib 3 of a word                     |
| LOWBIT         | low bit of a word, byte, or nibble  |
| HIGHBIT        | high bit of a word, byte, or nibble |
| BIT0           | bit 0 of a word, byte, or nibble    |
| BIT1           | bit 1 of a word, byte, or nibble    |
| BIT2           | bit 2 of a word, byte, or nibble    |
| BIT3           | bit 3 of a word, byte, or nibble    |
| BIT4 ... BIT7  | bits 4 through 7 of a word or byte  |
| BIT8 ... Bit15 | bits 8 through 15 of a word         |

**Table 4.3:** Variable Modifiers for all BS2 models.

The common sense rule for combining modifiers is that they must get progressively smaller from left to right. It would make no sense to specify, for instance, the low byte of a nibble, because a nibble is smaller than a byte! And just because you can stack up modifiers doesn't mean that you should unless it is the clearest way to express the location of the part you want get at. The example above might be improved:

ⓘ 2

```
rhino VAR Word ' A 16-bit variable
eye VAR rhino.BIT9 ' A bit
```

Although we've only discussed variable modifiers in terms of creating alias variables, you can also use them within program instructions:

```
rhino VAR Word ' A 16-bit variable
head VAR rhino.HIGHBYTE ' Highest 8 bits of rhino

rhino = 13567
DEBUG ? head ' Display alias variable head
DEBUG ? rhino.HIGHBYTE ' rhino.HIGHBYTE works too
STOP
```

Modifiers also work with arrays. For example:

```
myBytes VAR Byte(10) ' Define 10-byte array

myBytes(0) = $AB ' Hex $AB into 0th byte
DEBUG HEX ? myBytes.LOWNIB(0) ' Show low nib ($B)
DEBUG HEX ? myBytes.LOWNIB(1) ' Show high nib ($A)
```



## 4: BASIC Stamp Architecture – Aliases and Modifiers

---

If you looked closely at that example, you probably thought it was a misprint. Shouldn't `myBytes.LOWNIB(1)` give you the low nibble of byte 1 of the array rather than the high nibble of byte 0? Well, it doesn't. The modifier changes the meaning of the index value to match its own size. In the example above, when `myBytes()` is addressed as a byte array, it has 10 byte-sized cells numbered 0 through 9. When it is addressed as a nibble array, using `myBytes.LOWNIB()`, it has 20 nibble-sized cells numbered 0 through 19. You could also address it as individual bits using `myBytes.LOWBIT()`, in which case it would have 80 bit-sized cells numbered 0 through 79.

What if you use something other than a "low" modifier, say `myBytes.HIGHNIB()`? That will work, but its effect will be to start the nibble array with the high nibble of `myBytes(0)`. The nibbles you address with this nib array will all be contiguous, one right after the other, as in the previous example.

```
myBytes VAR Byte(10) ' Define 10-byte array.

myBytes(0) = $AB ' Hex $AB into 0th byte
myBytes(1) = $CD ' Hex $CD into next byte
DEBUG HEX ? myBytes.HIGHNIB(0) ' Show high nib of cell 0 ($A)
DEBUG HEX ? myBytes.HIGHNIB(1) ' Show next nib ($D)
```

This property of modified arrays makes the names a little confusing. If you prefer, you can use the less-descriptive versions of the modifier names; `BIT0` instead of `LOWBIT`, `NIB0` instead of `LOWNIB`, and `BYTE0` instead of `LOWBYTE`. These have exactly the same effect, but may be less likely to be misconstrued.

You may also use modifiers with the 0th cell of an array by referring to just the array name without the index value in parentheses. It's fair game for aliases and modifiers, both in `VAR` directives and in instructions.

### THE MEMORY MAP

On all BS2 models, if you're working on a program and wondering how much variable space you have left, you can use the Memory Map feature of the editor (CTRL-M). See the "Memory Map" section of Chapter 3 on page 50.

## ***BASIC Stamp Architecture – Scratch Pad RAM***

---

The BS2e, BS2sx, BS2p, BS2pe, and BS2px have some additional RAM called Scratch Pad RAM. The BS2e and BS2sx have 64 bytes of Scratch Pad RAM (0 to 63) and the BS2p, BS2pe, and BS2px have 136 bytes of Scratch Pad RAM (0-135). Scratch Pad RAM can only be accessed with the GET and PUT commands and cannot have variable names assigned to it. Table 4.4 shows the layout of all SPRAM registers.

SCRATCH PAD RAM



Notice that the highest locations in Scratch Pad RAM (location 63 in the BS2e and BS2sx, locations 127-135 in the BS2p, BS2pe, and BS2px) are special-purpose, read-only locations that always contain special run-time information. For example, the lowest nibble of location 63 (BS2e and BS2sx) or 127 (BS2p, BS2pe, and BS2px) contains the number of the currently running program slot. This is handy for programs that need to know which program slot they exist in. In the BS2p, BS2pe, and BS2px, the high nibble of location 127 holds the slot designated for READ and WRITE; see the STORE command on page 449 for more information.

## 4: BASIC Stamp Architecture – Scratch Pad RAM

**Table 4.4:** Layout of SPRAM Registers.

NOTE: Scratch Pad RAM can only be accessed with the GET and PUT commands. Scratch Pad RAM cannot have variable names assigned to it.

| Location | BS2e and BS2sx                        | BS2p, BS2pe and BS2px                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|----------|---------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0...62   | General Purpose RAM                   | General Purpose RAM                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 63       | Bits 0-3: Active program slot number. | General Purpose RAM                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 64...126 | n/a                                   | General Purpose RAM                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 127      | n/a                                   | Bits 0-3, Active program slot #. Bits 4-7, program slot for READ and WRITE operations.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 128      | n/a                                   | Polled input trigger status of Main I/O pins 0-7 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| 129      | n/a                                   | Polled input trigger status of Main I/O pins 8-15 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| 130      | n/a                                   | Polled input trigger status of Auxiliary I/O pins 0-7 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 131      | n/a                                   | Polled input trigger status of Auxiliary I/O pins 8-15 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 132      | n/a                                   | Bits 0-3: Polled-interrupt mode, set by POLLMODE.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| 133      | n/a                                   | Bits 0-2: Polled-interrupt "run" slot, set by POLLRUN.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 134      | n/a                                   | Bit 0: Active I/O group; 0 = Main I/O, 1 = Auxiliary I/O.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 135      | n/a                                   | Bit 0: Polled-output status (set by POLLMODE); 0 = disabled, 1 = enabled.<br>Bit 1: Polled-input status; 0 = none defined, 1 = at least one defined.<br>Bit 2: Polled-run status (set by POLLMODE); 0 = disabled, 1 = enabled.<br>Bit 3: Polled-output latch status; 0 = real-time mode, 1 = latch mode.<br>Bit 4: Polled-input state; 0 = no trigger, 1 = triggered.<br>Bit 5: Polled-output latch state; 0 = nothing latched, 1 = signal latched.<br>Bit 6: Poll-wait state; 0 = No Event, 1 = Event Occurred. (Cleared by POLLMODE only).<br>Bit 7: Polling status; 0 = not active, 1 = active. |

## BASIC Stamp Architecture – Constants and Expressions

---

Suppose you're working on a program called "Three Cheers" that flashes LEDs, makes hooting sounds, and activates a motor that crashes cymbals together, all in sets of three. A portion of your PBASIC program might contain something like:

```
FOR counter = 1 TO 3
 GOSUB Make_Cheers
NEXT
...
FOR counter = 1 TO 3
 GOSUB Blink_LEDs
NEXT
...
FOR counter = 1 TO 3
 GOSUB Crash_Cymbals
NEXT
```

The numbers 1 and 3 in the code above are called *constants*. They are constants because, while the program is running, nothing can happen to change those numbers. This distinguishes constants from variables, which can change while the program is running.

Constants are not limited to the decimal number system; PBASIC allows you to use several numbering systems. See "Number Representations" on page 96.

You can assign names to constants in a fashion similar to how variables are declared. On a BS1, it is identical to variable declarations. For all BS2 models, use the CON directive. Here is the syntax:

SYMBOL     Name   =   ConstantValue

-- or --

Name       CON    ConstantValue

CONSTANTS AND COMPILE-TIME EXPRESSIONS.



DEFINING AND USING CONSTANTS (CON).



## 4: BASIC Stamp Architecture – Constants and Expressions

---

Once created, named constants may be used in place of the numbers they represent. For example:

```
1 SYMBOL Cheers = 3 ' Number of cheers.
FOR counter = 1 TO Cheers
 GOSUB Make_Cheers
NEXT
```

-- OR --

```
All 2 Cheers CON 3 ' Number of cheers.
FOR counter = 1 TO Cheers
 GOSUB Make_Cheers
NEXT
```

That code works exactly the same as the corresponding FOR...NEXT loop in the previous example. The editor software substitutes the number 3 for the symbol named *Cheers* throughout your program. Like variables, labels and instructions, constant names are not case sensitive; *CHEERS*, and *ChEErs* are identical to *Cheers*.

Using named constants does not increase the amount of code downloaded to the BASIC Stamp, and it often improves the clarity of the program. Weeks after a program is written, you may not remember what a particular number was supposed to represent—using a name may jog your memory (or simplify the detective work needed to figure it out).

Named constants also have another benefit. Suppose the “Three Cheers” program had to be upgraded to “Five Cheers.” In the original example you would have to change all of the 3s to 5s. Search and replace would help, but you might accidentally change some 3s that weren’t numbers of cheers, too. However, if you had made smart use of a named constant, all you would have to do is change 3 to 5 in one place, the constant’s declaration:

```
1 SYMBOL Cheers = 5 ' Number of cheers.
```

-- OR --

```
All 2 Cheers CON 5 ' Number of cheers.
```

## BASIC Stamp Architecture – Number Representations

Now, assuming that you used the constant *Cheers* wherever your program needed ‘the number of cheers,’ your upgrade would be done.

On all BS2 models, you can take this idea a step further by defining constants with expressions; groups of math and/or logic operations that the editor software solves (evaluates) at compile-time (the time right after you start the download and before the BASIC Stamp starts running your program). For example, suppose the “Cheers” program also controls a pump to fill glasses with champagne. Perhaps the number of glasses to fill is always twice the number of cheers, minus 1 (another constant):

```
Cheers CON 5 ' # of cheers
Glasses CON Cheers*2-1 ' # of glasses
```

As you can see, one constant can be defined in terms of another. That is, the number glasses depends on the number cheers.

The expressions used to define constants must be kept fairly simple. The editor software solves them from left to right, and doesn’t allow you to use parentheses to change the order of evaluation. The operators that are allowed in constant expressions are shown in Table 4.5.

| Operator Symbol | Description |
|-----------------|-------------|
| +               | Add         |
| -               | Subtract    |
| *               | Multiply    |
| /               | Divide      |
| <<              | Shift Left  |
| >>              | Shift Right |
| &               | Logical AND |
|                 | Logical OR  |
| ^               | Logical XOR |

**Table 4.5:** Operators allowed in constant expressions for all BS2 models.

The BASIC Stamp, like any computer, excels at math and logic. However, being designed for control applications, the BASIC Stamp does math a little differently than a calculator or spreadsheet program. This section will help you understand BASIC Stamp numbers, math, and logic.

RUN-TIME MATH AND LOGIC.



In your programs, you may express a number in various ways, depending on how the number will be used and what makes sense to you. By default, the BASIC Stamp recognizes numbers like 0, 99 or 62145 as being in our

NUMBER REPRESENTATIONS.

## 4: BASIC Stamp Architecture – Number Representations

everyday decimal (base-10) system. However, you may also use hexadecimal (base-16; also called hex) or binary (base-2).

Since the symbols used in decimal, hex and binary numbers overlap (e.g., 1 and 0 are used by all; 0 through 9 apply to both decimal and hex) the editor software needs prefixes to tell the numbering systems apart, as shown below:

```
99 ' Decimal (no prefix)
$1A6 ' Hex (prefix '$' required)
%1101 ' Binary (prefix '%' required)
```

The BASIC Stamp also automatically converts quoted text into ASCII codes, and allows you to apply names (symbols) to constants from any of the numbering systems. For example:

```
1 SYMBOL LetterA = "A" ' ASCII code for A (65)
 SYMBOL Cheers = 3
 SYMBOL Hex128 = $80
 SYMBOL FewBits = %1101
```

-- OR --

```
2 LetterA CON "A" ' ASCII code for A (65)
 Cheers CON 3
 Hex128 CON $80
 FewBits CON %1101
```

### HEX TO BCD CONVERSION

2 Binary Coded Decimal (BCD) is a way to encode decimal digits that is easier to display or manipulate in some devices. Each digit of the decimal number (0 – 9) requires 4 bits (a nibble) to encode. For this reason, a BCD byte is always two decimal digits and a BCD word is always four decimal digits. The BASIC Stamp does not support BCD natively, however, because of the way that BCD is encoded the BS2 models' hexadecimal prefix, and Conversion Formatters can be used as a shortcut for most BCD input/output operations as long as the digits used do not exceed valid decimal digits (0 – 9). For example:

```
BCDValue CON $4096

DEBUG HEX BCDValue
```

The first line creates a symbol, *BCDValue*, that contains the binary form of the hexadecimal value \$4096, which means the upper nibble contains the binary value for the decimal digit 4, the next nibble is 0, the next nibble is 9

## BASIC Stamp Architecture – Number Representations

---

and the last nibble is 6; this corresponds exactly to the BCD form of the decimal number 4096. The second line in the above example uses the HEX Conversion Formatter within the DEBUG command (see DEBUG, page 159) to output the BCD value 4096 to the Debug Terminal. The HEX Conversion Formatter can also be used for input operations to convert a decimal value to BCD, as long as that decimal value is no greater than 2 digits for a Byte-sized variable or 4 digits for a Word-sized variable.

For more information on constants, see the section "Constants and Compile-Time Expressions", above.

With all BS2 models, some of the math or logic operations in a program are solved by the BASIC Stamp. The editor software solves operations that define constants before the program is downloaded to the BASIC Stamp. The preprocessing that takes place before the program is downloaded is referred to as "compile-time."

WHEN IS RUN-TIME?



After the download is complete, the BASIC Stamp starts executing your program; this is referred to as "run-time." At run-time the BASIC Stamp processes math and logic operations involving variables, or any combination of variables and constants.

Because compile-time and run-time expressions appear similar, it can be hard to tell them apart.

A few examples will help:

| result                 | VAR | Byte           | ' Compile-time assignment           |
|------------------------|-----|----------------|-------------------------------------|
| Cheers                 | CON | 3              | ' Compile-time                      |
| Glasses                | CON | Cheers * 2 - 1 | ' Compile-time                      |
| OneNinety              | CON | 100 + 90       | ' Compile-time                      |
| NotWorking             | CON | 3 * result     | ' ERROR: Variables not allowed here |
|                        |     |                |                                     |
| result = Glasses       |     |                | ' Run-time                          |
| result = 99 + Glasses  |     |                | ' Run-time                          |
| result = OneNinety + 1 |     |                | ' "100 + 90" solved at compile-time |
|                        |     |                | ' OneNinety + 1 solved at run-time  |
| result = 100 + 90      |     |                | ' 100 + 90 solved at run-time       |



Notice that the last example is solved at run-time, even though the math performed could have been solved at compile-time since it involves two constants. If you find something like this in your own programs, you can save some program space in the EEPROM by converting the run-time



## 4: BASIC Stamp Architecture – PIN Symbols

---

expression `100+90` into a compile-time expression like `OneNinety CON 100+90`.

To sum up: compile-time expressions are those that involve only constants; once a variable is involved, the expression must be solved at run-time. That's why the line `"NotWorking CON 3 * result"` would generate an error message. The `CON` directive works only at compile-time and `result` is a variable; variables are not allowed in compile-time expressions.

DEFINING AND USING PINS WITH THE PIN  
DIRECTIVE.



Now we know how to create variables and constants (with `VAR` and `CON`) but there is a third option if you're using PBASIC 2.5; pin-type symbols (with `PIN`). `PIN` is like `VAR` and `CON` put together and represents an I/O pin.

There are some situations where it is handy to refer to a pin using a variable (like `IN2` or `OUT2`) and also as a constant (`2`, in this case). The `PIN` directive lets you define a context-sensitive symbol representing an I/O pin. Depending on where and how this pin-type symbol is used determines whether it is treated as an I/O pin input variable, and I/O pin output variable or as a constant representing the pin number.

Let's explore a simple example to see where this is useful. It is common practice to define constants for any number used in many places so that changing that number doesn't create a maintenance hassle later on. If we were to use a constant symbol to represent an I/O pin, we might do something like this:

```
' {$PBASIC 2.5}

signal CON 1 ' constant-type symbol representing I/O 1

INPUT signal ' set signal pin to input

Wait:
 IF signal = 0 THEN Wait ' wait until signal pin = 1
```

Here we define *signal* to represent our desired I/O pin, then we use the `INPUT` command to set it to the input direction and later we check the state of the signal pin and loop (wait) while it is equal to logic 0. This code has a common bug, however; the `INPUT` command works as expected, because its *Pin* argument requires a number representing the I/O pin, but

## BASIC Stamp Architecture – PIN Symbols

---

the *Condition* argument in the IF...THEN statement will always evaluate to false because *signal* is a constant equal to 1, and “1 = 0” is false. What the user really meant to happen is something like: IF IN1 = 0 THEN Wait because *IN1* is the input variable representing the current state of I/O pin 1. This situation is perfect for the PIN directive:

```
' {$PBASIC 2.5}

signal PIN 1 ' pin-type symbol representing I/O 1

INPUT signal ' set signal pin to input

Wait:
 IF signal = 0 THEN Wait ' wait until signal = 1
```



We only changed one thing in this code: the CON directive was changed to PIN. Now *signal* is a context-sensitive symbol that will be treated as a constant or as a variable depending on where it is used. In the INPUT command *signal* is used as the *Pin* argument, and since that argument requires a number representing the pin number, *signal* is treated as a constant equal to 1. In the IF...THEN statement, *signal* is compared to another value (which implies that what *signal* represents is expected to change at run-time; i.e.: *signal's* value is “variable”) so *signal* is treated as a variable equal to the input variable for the defined pin (IN1 in this case).

As another example, consider the following code:

```
' {$PBASIC 2.5}

signal CON 2 ' constant-type symbol representing I/O 2

OUTPUT signal ' set signal pin to output
signal = 1 ' set signal high
```

Here, again, this is a common bug; the OUTPUT command will work as expected, but the `signal = 1` statement generates a syntax error at compile-time. Why the error? This is an assignment statement, meant to assign the value 1 to the item on the left, but the item on the left is a constant, not a variable, so it can not be changed at run-time. What the user was thinking when writing this was: `OUT2 = 1` which sets the value of the output variable representing I/O pin 2 to logical 1 (high). Here's the solution:

## 4: BASIC Stamp Architecture – PIN Symbols

```

All 2 ' {$PBASIC 2.5}

signal PIN 2 ' pin-type symbol representing I/O 2

OUTPUT signal ' set signal pin to output
signal = 1 ' set signal high

```

The OUTPUT command treats *signal* as a constant equal to 2 and the *signal = 1* statement treats *signal* as a variable equal to the output variable for the defined pin (OUT2 in this case).

You might be wondering why “*signal = 0*” in the IF...THEN statement of our first example treats *signal* as the input variable *IN1* and yet “*signal = 1*” in our last example treats *signal* as the output variable *OUT2*. The distinction is that the first example is a comparison and the second example is an assignment. Comparisons need to “read” expressions and then evaluate the comparison while assignments need to read expressions and then “write” the results. Since *signal* is to the left of the equal sign (=) in our assignment statement, it must be a variable we can write to, thus it must be treated as OUT2, in this case.

What happens if our pin-type symbol is to the right of the equal sign in an assignment statement? Example:

```

All 2 ' {$PBASIC 2.5}

signal1 PIN 1 ' pin-type symbol representing I/O 1
signal2 PIN 2 ' pin-type symbol representing I/O 2

INPUT signal1 ' set signal1 pin to input
OUTPUT signal2 ' set signal2 pin to output
signal2 = signal1 ' set signal2 pin to signal1 pin's state

```

In this case *signal2* is treated as OUT2 and *signal1* is treated as IN1; left side must be written to and right side must be read from.

If a pin-type symbol is used in a command, but not in the *Pin* argument of that command, it will be treated as an input variable (i.e.: INx). NOTE: It is very rare that you’ll need to use a pin-type symbol in this way.

The following is a summary of behaviors and uses of pin-type symbols.

## BASIC Stamp Architecture – PIN Symbols

---

*PIN\_Symbol* behaves like a constant:

1. when used in a command's *Pin* argument. Example:

```
OUTPUT PIN_Symbol
```

2. when used in the index of an array. Example:

```
myArray(PIN_Symbol) = 25
```

*PIN\_Symbol* behaves like an input variable (INx):

1. when used in a command's non-*Pin* argument that expects to read a variable/constant/expression. Example:

```
DEBUG BIN PIN_Symbol
```

2. when used in a command's *Condition* argument. Example:

```
IF PIN_Symbol = 1 THEN..
```

3. when used to the right of the equal sign (=) in an assignment statement. Example:

```
ex: myVariable = PIN_Symbol + 1
```

*PIN\_Symbol* behaves like an output variable (OUTx):

1. when used in a command's non-*Pin* argument that expects to write a result to a variable. Example:

```
LOOKUP index, [0, 1, 1, 0, 1], PIN_Symbol
```

2. when used to the left of the equal sign (=) in an assignment statement. Example:

```
PIN_Symbol = 1
```

Let's talk about the four basic operations of arithmetic: addition (+), subtraction (-), multiplication (\*), and division (/).

BASIC ARITHMETIC OPERATIONS



You may recall that the order in which you do a series of additions and subtractions doesn't affect the result. The expression 12+7-3+22 works out the same as 22-3+12+7. However, when multiplication or division are involved, it's a different story; 12+3\*2/4 is not the same as 2\*12/4+3. In fact, you may have the urge to put parentheses around portions of those equations to clear things up.

## 4: BASIC Stamp Architecture – Order of Operations

---

### ORDER OF OPERATIONS.

The BASIC Stamp solves math problems in the order they are written: from left to right. The result of each operation is fed into the next operation. So to compute  $12+3*2/4$ , the BASIC Stamp goes through a sequence like this:

$$\begin{aligned}12 + 3 &= 15 \\15 * 2 &= 30 \\30 / 4 &= 7\end{aligned}$$

Since the BASIC Stamp performs integer math (whole numbers only)  $30 / 4$  results in 7, not 7.5. We'll talk more about integers in the next section.

Some other dialects of BASIC would compute that same expression based on their precedence of operators, which requires that multiplication and division be done before addition. So the result would be:

$$\begin{aligned}3 * 2 &= 6 \\6 / 4 &= 1 \\12 + 1 &= 13\end{aligned}$$

Once again, because of integer math, the fractional portion of  $6 / 4$  is dropped, so we get 1 instead of 1.5.

 The BS1 does not allow parenthesis in expressions. Unfortunately, all expressions have to be written so that they evaluate as intended strictly from left to right.

 All BS2 models, however, allow parentheses to be used to change the order of evaluation. Enclosing a math operation in parentheses gives it priority over other operations. To make the BASIC Stamp compute the previous expression in the conventional way, you would write it as  $12 + (3*2/4)$ . Within the parentheses, the BASIC Stamp works from left to right. If you wanted to be even more specific, you could write  $12 + ((3*2)/4)$ . When there are parentheses within parentheses, the BASIC Stamp works from the innermost parentheses outward. Parentheses placed within parentheses are called "nested parentheses."

### INTEGER MATH.

The BASIC Stamp performs all math operations by the rules of positive integer math. That is, it handles only whole numbers, and drops any fractional portions from the results of computations. The BASIC Stamp handles negative numbers using two's complement rules.

## ***BASIC Stamp Architecture – Math and Operators***

---

All BS2 models can interpret two's complement negative numbers correctly in DEBUG and SEROUT instructions using formatters like SDEC (for signed decimal). In calculations, however, it assumes that all values are positive. This yields correct results with two's complement negative numbers for addition, subtraction, and multiplication, but not for division.



The standard operators we just discussed: +, -, \*, and / all work on two values; as in  $1 + 3$  or  $26 * 144$ . The values that operators process are referred to as arguments. So we say that the add, subtract, multiply and divide operators take two arguments.

UNARY AND BINARY OPERATORS.

Operators that take two arguments are called “binary” operators, and those that take only one argument are called “unary” operators. Please note that the term “binary operator” has nothing to do with binary numbers; it's just an inconvenient coincidence that the same word, meaning ‘involving two things’ is used in both cases.

The minus sign (-) is a bit of a hybrid. It can be used as a binary operator, as in  $8 - 2 = 6$ , or it can be used as a unary operator to represent negative numbers, such as -4.

Unary operators take precedence over binary operators; the unary operation is always performed first. For example, on all BS2 models, SQR is the unary operator for square root. In the expression  $10 - \text{SQR } 16$ , the BASIC Stamp first takes the square root of 16, then subtracts it from 10.

Most of the descriptions that follow say something like “computes (some function) of a 16-bit value.” This does not mean that the operator does not work on smaller byte or nibble values, but rather that the computation is done in a 16-bit workspace. If the value is smaller than 16 bits, the BASIC Stamp pads it with leading 0s to make a 16-bit value. If the 16-bit result of a calculation is to be packed into a smaller variable, the higher-order bits are discarded (truncated).

NOTES ABOUT THE 16-BIT WORKSPACE.

Keep this in mind, especially when you are working with two's complement negative numbers, or moving values from a larger variable to a smaller one. For example, look at what happens when you move a two's complement negative number into a byte (rather than a word):

## 4: BASIC Stamp Architecture – Math and Operators, ABS

```

All 2 value VAR Byte
value = - 99
DEBUG SDEC ? value ' Show signed decimal result (157)
```

All 2 We expected -99 to be displayed but what we got was 157. How did -99 become 157? Let's look at the bits: 99 is %01100011 binary. When the BASIC Stamp negates 99, it converts the number to 16 bits %0000000001100011, and then takes the two's complement, %1111111110011101. Since we've asked for the result to be placed in an 8-bit (byte) variable, the upper eight bits are truncated and the lower eight bits stored in the byte: %10011101.

Now for the second half of the story. DEBUG's SDEC modifier (for all BS2 models) expects a 16-bit, two's complement value, but we've only given it a byte to work with. As usual, it creates a 16-bit value by padding the leading eight bits with 0s: %00000000010011101. And what's that in signed decimal? 157.

To fix this problem, always store values that are intended to be signed into a word-sized variable.

### UNARY OPERATORS.

Table 4.1 lists the available Unary Operators. Note: the BS1 only supports negative (-).

**Table 4.1:** Unary Operators.  
Note: the BS1 only supports the negative (-) unary operator.

| Operator | Description                                      | Supported By:  |
|----------|--------------------------------------------------|----------------|
| ABS      | Returns absolute value                           | All except BS1 |
| COS      | Returns cosine in twos complement binary radians | All except BS1 |
| DCD      | 2 <sup>n</sup> -power decoder                    | All except BS1 |
| ~        | Inverse                                          | All except BS1 |
| -        | Negative                                         | All            |
| NCD      | Priority encoder of a 16-bit value               | All except BS1 |
| SIN      | Returns sine in twos complement binary radians   | All except BS1 |
| SQR      | Returns square root of value                     | All except BS1 |

### ABSOLUTE VALUE: ABS

All 2 The Absolute Value operator (ABS) converts a signed (two's complement) 16-bit number to its absolute value. The absolute value of a number is a positive number representing the difference between that number and 0. For example, the absolute value of -99 is 99. The absolute value of 99 is also 99. ABS works on two's complement negative numbers. Examples of ABS at work:

## BASIC Stamp Architecture – COS, DCD, ~, -

---

```
result VAR Word
```

```
result = -99 ' Put -99 into result
 ' ... (2's complement format)
DEBUG SDEC ? result ' Display as a signed #
DEBUG SDEC ? ABS result ' Display as a signed #
```

The Cosine operator (COS) returns the two's complement, 16-bit cosine of an angle specified as an 8-bit "binary radian" (0 to 255) angle. COS is the same as SIN in all respects, except that the cosine function returns the x distance instead of the y distance. See "Sine: SIN", below, for a code example and more information.

COSINE: COS



The Decoder operator (DCD) is a 2<sup>n</sup>-power decoder of a four-bit value. DCD accepts a value from 0 to 15, and returns a 16-bit number with the bit, described by value, set to 1. For example:

DECODER: DCD



```
result VAR Word
```

```
result = DCD 12 ' Set bit 12
DEBUG BIN16 ? result ' Display result (%0001000000000000)
```

The Inverse operator (~) complements (inverts) the bits of a number. Each bit that contains a 1 is changed to 0 and each bit containing 0 is changed to 1. This process is also known as a "bitwise NOT" and "ones complement". For example:

INVERSE: ~

```
result VAR Byte
```

```
result = %11110001 ' Store bits in byte result.
DEBUG BIN8 ? result ' Display in binary (%11110001)
result = ~ result ' Complement result
DEBUG BIN8 ? Result ' Display in binary (%00001110)
```

The Negative operator (-) negates a 16-bit number (converts to its twos complement).

NEGATIVE: -

```
SYMBOL result = W1
```



```
result = -99 ' Put -99 into result
 ' ... (2's complement format)
result = result + 100 ' Add 100 to it
DEBUG result ' Display result (1)
```

-- or --



## 4: BASIC Stamp Architecture – NCD, SIN

|              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <div><div>All 2</div><div><pre>result          VAR      Word  result = 99                      ' Put -99 into result                                 ' ... (2's complement format) DEBUG SDEC ? result              ' Display as a signed # result = -result                 ' Negate the value DEBUG SDEC ? result              ' Display as a signed #</pre></div></div>                                                                                                                                                                                                                                                                        |
| ENCODER: NCD | <div><div>All 2</div><div><p>The Encoder operator (NCD) is a "priority" encoder of a 16-bit value. NCD takes a 16-bit value, finds the highest bit containing a 1 and returns the bit position plus one (1 through 16). If the input value is 0, NCD returns 0. NCD is a fast way to get an answer to the question "what is the largest power of two that this value is greater than or equal to?" The answer NCD returns will be that power, plus one. Example:</p><pre>result          VAR      Word  result = %1101                  ' Highest bit set is bit 3 DEBUG ? NCD result              ' Show the NCD of result (4)</pre></div></div> |
| SINE: SIN    | <div><div>All 2</div><div><p>The Sine operator (SIN) returns the two's complement, 16-bit sine of an angle specified as an 8-bit binary radian (0 to 255) angle.</p></div></div>                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

To understand the SIN operator more completely, let's look at a typical sine function. By definition: given a circle with a radius of 1 unit (known as a unit circle), the sine is the y-coordinate distance from the center of the circle to its edge at a given angle. Angles are measured relative to the 3-o'clock position on the circle, increasing as you go around the circle counterclockwise.

At the origin point (0 degrees) the sine is 0, because that point has the same y (vertical) coordinate as the circle center. At 45 degrees the sine is 0.707. At 90 degrees, sine is 1. At 180 degrees, sine is 0 again. At 270 degrees, sine is -1.

The BASIC Stamp SIN operator breaks the circle into 0 to 255 units instead of 0 to 359 degrees. Some textbooks call this unit a "binary radian" or "brad." Each brad is equivalent to 1.406 degrees. And instead of a unit circle, which results in fractional sine values between 0 and 1, BASIC Stamp SIN is based on a 127-unit circle. Results are given in two's complement form in order to accommodate negative values. So, at the origin, SIN is 0. At 45 degrees (32 brads), sine is 90. At 90 degrees (64 brads), sine is 127. At 180 degrees (128 brads), sine is 0. At 270 degrees (192 brads), sine is -127.

## BASIC Stamp Architecture – SIN, SQR

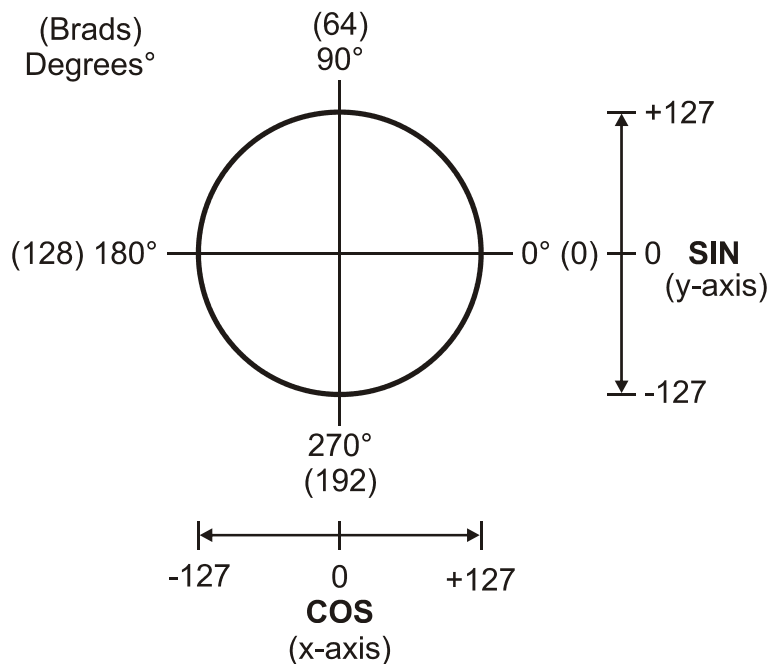


Figure 4.1: 127-Unit Circle

To convert brads to degrees, multiply by 180 then divide by 128, or simply multiply with `"*/ 360"`. To convert degrees to brads, multiply by 128, then divide by 180. Here's a small program that demonstrates the SIN and COS operators:

```
degr VAR Word
brads VAR Byte

DEBUG 2, 4, 0, "ANGLE", TAB, "COS", TAB, "SIN", CR
DEBUG "DEGREES", TAB, "BRADS", TAB, "(X)", TAB, "(Y)", CR

FOR degr = 0 TO 359 STEP 45 ' Increment degrees
 brads = degr * 128 / 180 ' Convert to brads
 DEBUG CR, DEC3 degr, TAB, DEC3 brads, TAB ' Display angle
 DEBUG SDEC COS brads, TAB, SDEC SIN brads ' Display COS & SIN
NEXT
```

The Square Root operator (SQR) computes the integer square root of an unsigned 16-bit number. (The number must be unsigned since the square root of a negative number is an 'imaginary' number.) Remember that most square roots have a fractional part that the BASIC Stamp discards when

SQUARE ROOT: SQR



## 4: BASIC Stamp Architecture – +

doing its integer-only math. So it computes the square root of 100 as 10 (correct), but the square root of 99 as 9 (the actual is close to 9.95). Example:

```
DEBUG ? SQR 100 ' Display square root of 100 (10)
DEBUG ? SQR 99 ' Display of square root of 99
 ' ...(9 due to truncation)
```

BINARY OPERATORS.

Table 4.6 lists the available Binary (two-argument) Operators.

**Table 4.6:** Binary Operators. Note: some binary operators are not supported by all BASIC Stamp models.

| Operator | Description                               | Supported By:  |
|----------|-------------------------------------------|----------------|
| +        | Addition                                  | All            |
| -        | Subtraction                               | All            |
| *        | Multiplication                            | All            |
| **       | Multiplication (returns upper 16-bits)    | All            |
| */       | Multiply by 8-bit integer, 8-bit fraction | All BS2 models |
| /        | Division                                  | All            |
| //       | Modulus (Remainder of division)           | All            |
| ATN      | Returns arctangent of X/Y vector          | All BS2 models |
| HYP      | Returns hypotenuse of X/Y vector          | All BS2 models |
| MIN      | Limits a value to a specified low         | All            |
| MAX      | Limits a value to a specified high        | All            |
| DIG      | Returns specified digit of number         | All BS2 models |
| <<       | Shift bits left by specified amount       | All BS2 models |
| >>       | Shift bits right by specified amount      | All BS2 models |
| REV      | Reverse specified number of bits          | All BS2 models |
| &        | Bitwise AND                               | All            |
|          | Bitwise OR                                | All            |
| ^        | Bitwise XOR                               | All            |
| &/       | Logical AND NOT                           | BS1 Only       |
| /        | Logical OR NOT                            | BS1 Only       |
| ^/       | Logical XOR NOT                           | BS1 Only       |

ADD: +



The Addition operator (+) adds variables and/or constants, returning a 16-bit result. It works exactly as you would expect with unsigned integers from 0 to 65535. If the result of addition is larger than 65535, the carry bit will be lost. If the values added are signed 16-bit numbers and the destination is a 16-bit variable, the result of the addition will be correct in both sign and value.

## BASIC Stamp Architecture – +, -, \*

---

For example:

```
SYMBOL value1 = W0
SYMBOL value2 = W1

value1 = - 99
value2 = 100
value1 = value1 + value2 ' Add the numbers
DEBUG value1 ' Show the result (1)
```



-- Or --

```
value1 VAR Word
value2 VAR Word

value1 = - 1575
value2 = 976
value1 = value1 + value2 ' Add the numbers
DEBUG SDEC ? value1 ' Show the result (-599)
```



The Subtraction operator (-) subtracts variables and/or constants, returning a 16-bit result. It works exactly as you would expect with unsigned integers from 0 to 65535. If the result is negative, it will be correctly expressed as a signed 16-bit number. For example:



```
SYMBOL value1 = W0
SYMBOL value2 = W1

value1 = 199
value2 = 100
value1 = value1 - value2 ' Subtract the numbers
DEBUG value1 ' Show the result (99)
```



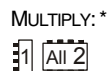
-- Or --

```
value1 VAR Word
value2 VAR Word

value1 = 1000
value2 = 1999
value1 = value1 - value2 ' Subtract the numbers
DEBUG SDEC ? value1 ' Show the result (-999)
```



The Multiply operator (\*) multiplies variables and/or constants, returning the low 16 bits of the result. It works exactly as you would expect with unsigned integers from 0 to 65535. If the result of multiplication is larger than 65535, the excess bits will be lost. Multiplication of signed variables will be correct in both number and sign, provided that the result is in the range -32767 to +32767.



## 4: BASIC Stamp Architecture – \*, \*\*

```
1 SYMBOL value1 = W0
 SYMBOL value2 = W1

value1 = 1000
value2 = 19
value1 = value1 * value2 ' Multiply value1 by value2
DEBUG value1 ' Show the result (19000)
```

-- OR --

```
All 2 value1 VAR Word
 value2 VAR Word
 value1 = 1000
 value2 = - 19
 value1 = value1 * value2 ' Multiply value1 by value2
 DEBUG SDEC ? value1 ' Show the result (-19000)
```

MULTIPLY HIGH: \*\*

All 2 The Multiply High operator (\*\*) multiplies variables and/or constants, returning the high 16 bits of the result. When you multiply two 16-bit values, the result can be as large as 32 bits. Since the largest variable supported by PBASIC is a word (16 bits), the highest 16 bits of a 32-bit multiplication result are normally lost. The \*\* (double-star) instruction gives you these upper 16 bits. For example, suppose you multiply 65000 (\$FDE8) by itself. The result is 4,225,000,000 or \$FBD46240. The \* (star, or normal multiplication) instruction would return the lower 16 bits, \$6240. The \*\* instruction returns \$FBD4.

```
1 SYMBOL value1 = W0
 SYMBOL value2 = W1

value1 = $FDE8
value2 = value1 ** value1 ' Multiply $FDE8 by itself
DEBUG $value2 ' Return high 16 bits ($FBD4)
```

-- OR --

```
All 2 value1 VAR Word
 value2 VAR Word

value1 = $FDE8
value2 = value1 ** value1 ' Multiply $FDE8 by itself
DEBUG HEX ? value2 ' Return high 16 bits ($FBD4)
```

An interesting application of the \*\* operator allows you to multiply a number by a fractional value less than one. The fraction must be expressed in units of 1/65536. To find the fractional \*\* argument,

## BASIC Stamp Architecture – \*\*, \*/

---

multiply the fraction part by 65536. For example, 0.72562 is represented by 47554, which is  $0.72562 \times 65536$ .

```
SYMBOL Frac = 47554 ' = 0.72562 x 65536
SYMBOL value = W0

value = 10000
value = value ** Frac ' Multiply 10000 by 0.72562
DEBUG value ' Show result (7256)
```



-- OR --

```
Frac CON 47554 ' = 0.72562 x 65536
value VAR Word

value = 10000
value = value ** Frac ' Multiply 10000 by 0.72562
DEBUG ? value ' Show result (7256)
```



The Multiply Middle operator (**\*/**) multiplies variables and/or constants, returning the middle 16 bits of the 32-bit result. This has the effect of multiplying a value by a whole number and a fraction. The whole number is the upper byte of the multiplier (0 to 255 whole units) and the fraction is the lower byte of the multiplier (0 to 255 units of 1/256 each). The **\*/** (star-slash) instruction gives you an excellent workaround for the BASIC Stamp's integer-only math. Suppose you want to multiply a value by 1.5. The whole number, and therefore the upper byte of the multiplier, would be 1, and the lower byte (fractional part) would be 128, since  $128/256 = 0.5$ . It may be clearer to express the **\*/** multiplier in hex—as \$0180—since hex keeps the contents of the upper and lower bytes separate. Here's an example:

MULTIPLY MIDDLE: \*/

```
value1 VAR Word

value1 = 100
value1 = value1 */ $0180 ' Multiply by 1.5 [1 + (128/256)]
DEBUG ? value1 ' Show result (150)
```



To calculate the constant for use with the **\*/** operator, multiply the target (mixed) value by 256 and convert to an integer. For instance, take Pi ( $\pi$ , 3.14159). The **\*/** constant would be  $\text{INT}(3.14159 \times 256) = 804$  (\$0324). So the constant Pi for use with **\*/** would be \$0324. This isn't a perfect match for Pi, but the error is only about 0.1%. Note that the **\*/** operator can be used to multiply by mixed values up to about 255.996.

## 4: BASIC Stamp Architecture – /, //

DIVIDE: /



The Divide operator (/) divides variables and/or constants, returning a 16-bit result. Works exactly as you would expect with unsigned integers from 0 to 65535. Use / only with positive values; signed values do not provide correct results. Here's an example of unsigned division:



```
SYMBOL value1 = W0
SYMBOL value2 = W1
value1 = 1000
value2 = 5

value1 = value1 / value2 ' Divide the numbers
DEBUG value1 ' Show the result (200)
```

-- OR --



```
value1 VAR Word
value2 VAR Word

value1 = 1000
value2 = 5
value1 = value1 / value2 ' Divide the numbers
DEBUG DEC ? value1 ' Show the result (200)
```

A workaround to the inability to divide signed numbers is to have your program divide absolute values, then negate the result if one (and only one) of the operands was negative. All values must lie within the range of -32767 to +32767. Here is an example:



```
sign VAR Bit ' Bit to hold the result sign
value1 VAR Word
value2 VAR Word

value1 = 100
value2 = -3200

sign = value1.BIT15 ^ value2.BIT15 ' Determine sign of result
value2 = ABS value2 / ABS value1 ' Divide absolute values
IF (sign = 1) THEN value2 = -value2 ' Negate result if sign = 1
DEBUG SDEC ? value2 ' Show the result (-32)
```

MODULUS: //



The Modulus operator (//) returns the remainder left after dividing one value by another. Some division problems don't have a whole-number result; they return a whole number and a fraction. For example,  $1000/6 = 166.667$ . Integer math doesn't allow the fractional portion of the result, so  $1000/6 = 166$ . However, 166 is an approximate answer, because  $166*6 = 996$ . The division operation left a remainder of 4. The // (double-slash)

## ***BASIC Stamp Architecture – //, ATN***

---

returns the remainder, 4 in this example. Naturally, numbers that divide evenly, such as 1000/5, produce a remainder of 0.

Example:

```
SYMBOL value1 = W0
SYMBOL value2 = W1

value1 = 1000
value2 = 6
value1 = value1 // value2 ' Get remainder of value1 / value2
DEBUG value1 ' Show the result (4)
```

-- Or --

```
value1 VAR Word
value2 VAR Word

value1 = 1000
value2 = 6
value1 = value1 // value2 ' Get remainder of value1 / value2
DEBUG ? value1 ' Show the result (4)
```

The Arctangent operator (ATN) returns the angle to the vector specified by X and Y coordinate values. The syntax of ATN is:

ARCTANGENT: ATN



xCoord ATN yCoord

where xCoord and yCoord are the coordinates of the target vector point.

In the BASIC Stamp, the angle is returned in binary radians (0 to 255) instead of degrees (0 to 359). See the explanation of the SIN operator for more information about binary radians. Coordinate input values are limited to -127 to 127 (signed bytes) as shown in the diagram of the PBASIC Unit Circle (Figure 4.2).

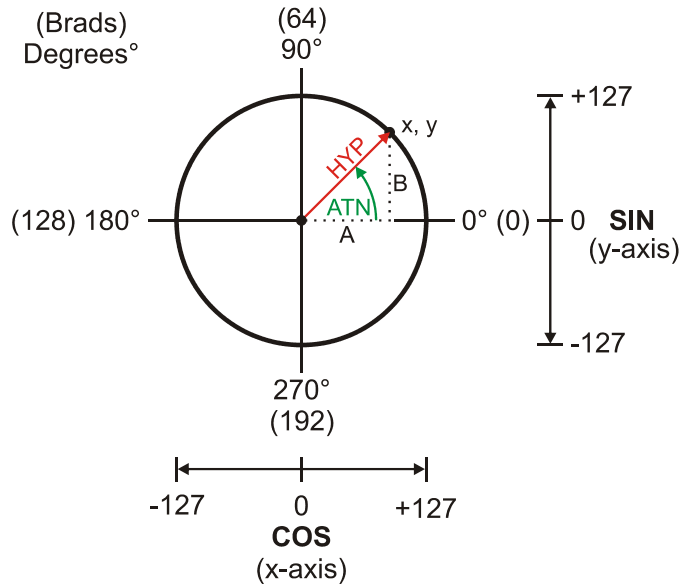
```
brads VAR Word ' angle in brads
degr VAR Word ' angle in degrees

brads = 4 ATN 4 ' get angle
degr = brads * / 360 ' convert to degrees
DEBUG SDEC ? brads ' display brads (32)
DEBUG SDEC ? degr ' display degrees (45)
```



## 4: BASIC Stamp Architecture – ATN, HYP, MIN

**Figure 4.2:** ATN and HYP operators in the PBASIC unit circle



HYPOTENUSE: HYP



The Hypotenuse operator (HYP) returns the length of the hypotenuse of a right triangle with sides of length A and B. The syntax of HYP is:

```
SideA HYP SideB
```

where SideA and SideB are the side lengths of a right-triangle (the order isn't important). Another application of HYP is to calculate the distance between the origin (0, 0) and a point (X, Y) in a Cartesian coordinate system. Side length (vector) input values are limited to -127 to 127 (signed bytes). See diagram with ATN operator, Figure 4.2.

```
DEBUG ? 3 HYP 4
```

' hypotenuse of 3x4 triangle (5)

MINIMUM: MIN



The Minimum operator (MIN) limits a value to a specified 16-bit positive minimum. The syntax of MIN is:

```
value MIN limit
```

where *value* is a constant or variable value to perform the MIN function upon and *limit* is the minimum value that *value* is allowed to be. Its logic is, 'if *value* is less than *limit*, then make result = *limit*; if *value* is greater than or equal to *limit*, make result = *value*.'

## BASIC Stamp Architecture – MIN, MAX

---

MIN works in positive math only; its comparisons are not valid when used on two's complement negative numbers, since the positive-integer representation of a number like -1 (\$FFFF or 65535 in unsigned decimal) is larger than that of a number like 10 (\$000A or 10 decimal). Use MIN only with unsigned integers. Because of the way fixed-size integers work, you should be careful when using an expression involving MIN 0. For example, 0-1 MIN 0 will result in 65535 because of the way fixed-size integers wrap around.

```
SYMBOL value1 = W0
SYMBOL value2 = W1

FOR value1 = 100 TO 0 STEP -10
 value2 = value1 MIN 50
 DEBUG value2
NEXT
-- OR --
```



```
value VAR Word

FOR value = 100 TO 0 STEP 10
 DEBUG ? value MIN 50
NEXT
```



The Maximum operator (MAX) limits a value to a specified 16-bit positive maximum. The syntax of MAX is:

```
value MAX Limit
```

Where *value* is a constant or variable value to perform the MAX function upon and *limit* is the maximum value that *value* is allowed to be. Its logic is, 'if *value* is greater than *limit*, then make result = *limit*; if *value* is less than or equal to *limit*, make result = *value*.' MAX works in positive math only; its comparisons are not valid when used on two's complement negative numbers, since the positive-integer representation of a number like -1 (\$FFFF or 65535 in unsigned decimal) is larger than that of a number like 10 (\$000A or 10 decimal). Use MAX only with unsigned integers. Because of the way fixed-size integers work, you should be careful when using an expression involving MAX 65535. For example, 65535+1 MAX 65535 will result in 0 because of the way fixed-size integers wrap around.

## 4: BASIC Stamp Architecture – MAX, DIG, <<, >>

```
1 SYMBOL value1 = W0
 SYMBOL value2 = W1

FOR value1 = 0 TO 100 STEP 10 ' Walk value1 from 0 to 100
 value2 = value1 MAX 50 ' Use MAX to clamp at 50
 DEBUG value2 ' Show "clamped" value
NEXT

-- OR --
```

```
All 2 value VAR Word
 FOR value = 0 TO 100 STEP 10 ' Walk value from 0 to 100
 DEBUG ? value MAX 50 ' Show value, use MAX clamp at 50
 NEXT
```

DIGIT: DIG

All 2 The Digit operator (DIG) returns the specified decimal digit of a 16-bit positive value. Digits are numbered from 0 (the rightmost digit) to 4 (the leftmost digit of a 16-bit number; 0 to 65535).

Example:

```
value VAR Word
idx VAR Byte

value = 9742
DEBUG ? value DIG 2 ' Show digit 2 (7)
FOR idx = 4 TO 0
 DEBUG ? value DIG idx ' Show digits 0 through 4 (09742)
NEXT
```

SHIFT LEFT: <<

All 2 The Shift Left operator (<<) shifts the bits of a value to the left a specified number of places. Bits shifted off the left end of a number are lost; bits shifted into the right end of the number are 0s. Shifting the bits of a value left  $n$  number of times has the same effect as multiplying that number by 2 to the  $n^{\text{th}}$  power. For instance  $100 \ll 3$  (shift the bits of the decimal number 100 left three places) is equivalent to  $100 * 2^3$ . Here's an example:

```
value VAR Word
idx VAR Byte

value = %1111111111111111
FOR idx = 1 TO 16 ' Repeat with idx = 1 to 16
 DEBUG BIN16 ? value << idx ' Shift value left idx places
NEXT
```

SHIFT RIGHT: >>

All 2 The Shift Right operator (>>) shifts the bits of a value to the right a specified number of places. Bits shifted off the right end of a number are

## BASIC Stamp Architecture – >>, REV, &, |

lost; bits shifted into the left end of the number are 0s. Shifting the bits of a value right  $n$  number of times has the same effect as dividing that number by 2 to the  $n^{\text{th}}$  power. For instance  $100 \gg 3$  (shift the bits of the decimal number 100 right three places) is equivalent to  $100 / 2^3$ . Here's an example:

```
value VAR Word
idx VAR Byte
```

```
value = %1111111111111111
FOR idx = 1 TO 16 ' Repeat with idx = 1 to 16
 DEBUG BIN16 ? value >> idx ' Shift value right idx places
NEXT
```

The Reverse operator (REV) returns a reversed (mirrored) copy of a specified number of bits of a value, starting with the right-most bit (least significant bit or “lsb”). For instance,  $\%10101101 \text{ REV } 4$  would return  $\%1011$ , a mirror image of the right-most four bits of the value. Example:

```
DEBUG BIN4 ? %10101101 REV 4 ' Mirror 1st 4 bits (%1011)
```

The And operator (&) returns the bitwise AND of two values. Each bit of the values is subject to the following logic:

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

The result returned by & will contain 1s in only those bit positions in which both input values contain 1s. Example:

```
SYMBOL value1 = B2
SYMBOL value2 = B3
SYMBOL result = B4
```

```
value1 = %00001111
value2 = %10101101
result = value1 & value2
DEBUG %result ' Show result of AND (%00001101)
-- or --
```

```
DEBUG BIN8 ? %00001111 & %10101101 ' Show result of AND (%00001101)
```

The OR operator (|) returns the bitwise OR of two values. Each bit of the values is subject to the following logic:

## 4: BASIC Stamp Architecture – |, ^

0 OR 0 = 0  
0 OR 1 = 1  
1 OR 0 = 1  
1 OR 1 = 1

The result returned by | will contain 1s in any bit positions in which one or the other (or both) input values contain 1s. Example:

```
1 SYMBOL value1 = B2
 SYMBOL value2 = B3
 SYMBOL result = B4

value1 = %00001111
value2 = %10101001
result = value1 | value2
DEBUG %result ' Show result of OR (%10101111)

-- or --

DEBUG BIN ? %00001111 | %10101001 ' Show result of OR (%10101111)
```

XOR: ^

**All 2** The Xor operator (^) returns the bitwise XOR of two values. Each bit of the values is subject to the following logic:

0 XOR 0 = 0  
0 XOR 1 = 1  
1 XOR 0 = 1  
1 XOR 1 = 0

The result returned by ^ will contain 1s in any bit positions in which one or the other (but not both) input values contain 1s. Example:

```
1 SYMBOL value1 = B2
 SYMBOL value2 = B3
 SYMBOL result = B4

value1 = %00001111
value2 = %10101001
result = value1 ^ value2
DEBUG %result ' Show result or XOR (%10100110)

-- or --
```

## BASIC Stamp Architecture – &/, |/

---

```
DEBUG BIN8 ? %00001111 ^ %10101001 ' Show result of XOR (%10100110)
```



The And Not operator (&/) returns the bitwise AND NOT of two values. Each bit of the values is subject to the following logic:



0 AND NOT 0 = 0  
0 AND NOT 1 = 0  
1 AND NOT 0 = 1  
1 AND NOT 1 = 0

The result returned by &/ will contain 1s in any bit positions in which the first value is 1 and the second value is 0. Example:

```
SYMBOL value1 = B2
SYMBOL value2 = B3
SYMBOL result = B4
```



```
value1 = %00001111
value2 = %10101001
result = value1 &/ value2
DEBUG %result ' Show result of AND NOT (%00000110)
```

The Or Not operator (|/) returns the bitwise OR NOT of two values. Each bit of the values is subject to the following logic:



0 OR NOT 0 = 1  
0 OR NOT 1 = 0  
1 OR NOT 0 = 1  
1 OR NOT 1 = 1

The result returned by |/ will contain 1s in any bit positions in which the first value is 1 or the second value is 0. Example:

```
SYMBOL value1 = B2
SYMBOL value2 = B3
SYMBOL result = B4
```



```
value1 = %00001111
value2 = %10101001
result = value1 |/ value2
DEBUG %result ' Show result of OR NOT (%01011111)
```

## 4: BASIC Stamp Command Reference – ^/

---

XOR NOT: ^/

 The Xor Not operator (^/) returns the bitwise XOR NOT of two values. Each bit of the values is subject to the following logic:

```
0 XOR NOT 0 = 1
0 XOR NOT 1 = 0
1 XOR NOT 0 = 0
1 XOR NOT 1 = 1
```

The result returned by ^/ will contain 1s in any bit positions in which the first value and second values are equal.

Example:

```
 SYMBOL value1 = B2
SYMBOL value2 = B3
SYMBOL result = B4

value1 = %00001111
value2 = %10101001
result = value1 ^/ value2
DEBUG %result ' Show result of OR NOT (%01011001)
```

## ***BASIC Stamp Architecture***

---



## **5: BASIC Stamp Command Reference**

---

### **Introduction**

This chapter provides details on all three versions of the PBASIC Programming Language. A categorical listing of all available PBASIC commands is followed by an alphabetized command reference with syntax, functional descriptions, and example code for each command.

### **PBASIC LANGUAGE VERSIONS**

There are three forms of the PBASIC language: PBASIC 1.0 (for the BS1), PBASIC 2.0 (for all BS2 models) and PBASIC 2.5 (for all BS2 models). You may use any version of the language that is appropriate for your BASIC Stamp module; however, when using any BS2 model, we suggest you use PBASIC 2.5 for any new programs you write because of the advanced control and flexibility it allows. PBASIC 2.5 is backward compatible with almost every existing PBASIC 2.0-based program, and code that is not 100% compatible can easily be modified to work in PBASIC 2.5.

This chapter gives details on every command for every BASIC Stamp model. Be sure to pay attention to any notes in the margins and body text regarding supported models and PBASIC language versions wherever they apply.

The BASIC Stamp Editor for Windows defaults to using PBASIC 1.0 (for the BS1) or PBASIC 2.0 (for all BS2 models). If you wish to use the default language for your BASIC Stamp model you need not do anything special. If you wish to use PBASIC 2.5, you must specify that fact, using the \$PBASIC directive in your source code, for example:

```
' {$PBASIC 2.5}
```

Review the Compiler Directives section of Chapter 3 for more details on this directive. Note: you may also specify either 1.0 or 2.0 using the \$PBASIC directive if you wish to explicitly state those desired languages.

Please note that the reserved word set will vary with each version of PBASIC, with additional reserved words for some BASIC Stamp models. Please see the reserved words tables in Appendix B for the complete lists. PBASIC 2.5 features many enhancements. Table 5.1 gives a brief summary of these items, with references to more information given elsewhere.

# BASIC Stamp Command Reference

| Feature Categories              | New Items                                                                           | Description                                                                                                                                                   |
|---------------------------------|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Additional Commands             | DEBUGIN,<br>DO...LOOP, EXIT,<br>ON. SELECT...CASE                                   | Allows easier user input and program control. See individual command descriptions.                                                                            |
| Enhanced Commands               | IF...THEN,<br>GET, PUT,<br>READ, WRITE                                              | Improves program control, and SPRAM and EEPROM access. See individual command descriptions.                                                                   |
| Additional Directive            | PIN                                                                                 | Provides flexible, context-sensitive I/O pin references; see page 99.                                                                                         |
| Conditional Compile Directives  | #DEFINE, #ERROR,<br>#IF...#THEN...#ELSE,<br>#SELECT...CASE                          | Encourages development of source code that is compatible with multiple BASIC Stamp models and helpful user hints; see page 70.                                |
| Additional Predefined Constants | CLRDN, CLREOL,<br>CRSRDN, CRSRLF,<br>CRSRRT, CRSRUP,<br>CRSRX, CRSRXY,<br>CRSRY, LF | Symbols for control characters supported by the Debug Terminal. See Table 5.13 in the DEBUG command description, page 168.                                    |
| Syntax Enhancements             | , and :                                                                             | Any line can be continued to the next line after the comma (,) character wherever commas are normally used.<br><br>Colons are required on label declarations. |

**Table 5.1:** PBASIC 2.5 Enhancements.

## CATEGORICAL LISTING OF COMMANDS

This section lists all available PBASIC commands for all BASIC Stamp models, grouped by category. Commands with PBASIC 2.5 enhanced syntax options are marked with (\*); commands that exist only in PBASIC 2.5 are indicated with (<sup>2.5</sup>).

    One or more of these icons indicates the item applies to the BS1, BS2, BS2e, BS2sx, BS2p, BS2pe, or BS2px respectively.

 If an item applies to all the models in the BS2 family, this icon is used.

## BRANCHING / PROGRAM CONTROL

-   **BRANCH**      Jump to address specified by offset.
-   **IF...THEN\***      Conditionally execute one or more blocks of code.
-   **GOTO**      Jump to address.
-   **GOSUB**      Jump to subroutine at address.

## 5: BASIC Stamp Command Reference

|                                                                                                     |                                                                                                           |
|-----------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
|  ON <sup>2.5</sup> | Jump to address or subroutine specified by an offset.                                                     |
|  RETURN            | Return from subroutine.                                                                                   |
|  RUN               | Switch execution to another program slot.                                                                 |
|  POLLRUN           | Switch execution to another program page upon the occurrence of a polled interrupt.                       |
|  SELECT            |                                                                                                           |
| ...CASE <sup>2.5</sup>                                                                              | Evaluate expression and conditionally execute a block of code based on comparison to multiple conditions. |
|  STOP              | Halt program execution until BASIC Stamp is reset.                                                        |

### LOOPING STRUCTURES

|                                                                                                             |                                                                            |
|-------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|
|  DO...LOOP <sup>2.5</sup> | Execute code block repetitively, with optional, conditional exit.          |
|  EXIT <sup>2.5</sup>     | Terminate execution of a looping code block (DO...LOOP or FOR...NEXT).     |
|  FOR...NEXT              | Execute code block repetitively, a finite number of times using a counter. |

### EEPROM Access

|                                                                                            |                                                     |
|--------------------------------------------------------------------------------------------|-----------------------------------------------------|
|  EEPROM | Store data in EEPROM during program download.       |
|  DATA   | Store data in EEPROM during program download.       |
|  READ*  | Read EEPROM value into variable.                    |
|  WRITE* | Write value into EEPROM.                            |
|  STORE  | Switch READ/WRITE access to different program slot. |

### RAM Access

|                                                                                          |                                           |
|------------------------------------------------------------------------------------------|-------------------------------------------|
|  GET* | Read Scratch Pad RAM value into variable. |
|  PUT* | Write value into Scratch Pad RAM.         |

# ***BASIC Stamp Command Reference***

---

## **NUMERICS**

-  **LET** Optional instruction to perform variable assignments.
-   **LOOKUP** Look up data specified by offset and store in variable. This instruction provides a means to make a lookup table.
-   **LOOKDOWN** Find target's matching value in table and store match number (0-N) in variable.
-   **RANDOM** Generate a pseudo-random number.

## **DIGITAL I/O**

-  **CONFIGPIN** Configure pin properties.
-   **INPUT** Make pin an input.
-   **OUTPUT** Make pin an output.
-   **REVERSE** Reverse direction of a pin.
-   **LOW** Make pin output low.
-   **HIGH** Make pin output high.
-   **TOGGLE** Make pin an output and toggle state.
-   **PULSIN** Measure width of an input pulse.
-   **PULSOUT** Output a pulse by inverting a pin for a given amount of time.
-   **BUTTON** Debounce button, perform auto-repeat, and branch to address if button is in target state.
-  **COUNT** Count cycles on a pin for a given amount of time.
-  **XOUT** Generate X-10 power line control codes.
-    **AUXIO** Switch control to auxiliary I/O pin group.
-    **MAINIO** Switch control to main I/O pin group.
-    **IOTERM** Switch control to specified I/O pin group.
-    **POLLIN** Specify pin and state for a polled-interrupt.
-    **POLLOUT** Specify pin and state for output upon a polled-interrupt.
-    **POLLMODE** Specify the polled-interrupt mode.

---

## 5: BASIC Stamp Command Reference

### ASYNCHRONOUS SERIAL I/O

|  |        |                                               |
|--|--------|-----------------------------------------------|
|  | SERIN  | Input data in an asynchronous serial stream.  |
|  | SEROUT | Output data in an asynchronous serial stream. |
|  | OWIN   | Input data from a 1-wire device.              |
|  | OWOUT  | Output data to a 1-wire device.               |

### SYNCHRONOUS SERIAL I/O

|  |          |                                                 |
|--|----------|-------------------------------------------------|
|  | SHIFTIN  | Shift data in from synchronous serial device.   |
|  | SHIFTOUT | Shift data out to synchronous serial device.    |
|  | I2CIN    | Input data from I <sup>2</sup> C serial device. |
|  | I2COUT   | Output data to I <sup>2</sup> C serial device.  |

### PARALLEL I/O

|  |        |                            |
|--|--------|----------------------------|
|  | LDCMD  | Write a command to an LCD. |
|  | LCDIN  | Read data from an LCD.     |
|  | LCDOUT | Write data to an LCD.      |

### ANALOG I/O

|  |         |                                                                     |
|--|---------|---------------------------------------------------------------------|
|  | PWM     | Output using pulse width modulation, then return pin to input.      |
|  | POT     | Read a 5 k $\Omega$ - 50 k $\Omega$ potentiometer and scale result. |
|  | RCTIME  | Measure a variable resistance or capacitance.                       |
|  | COMPARE | Compare two 0-5 V analog voltages.                                  |

### TIME

|  |          |                                           |
|--|----------|-------------------------------------------|
|  | PAUSE    | Pause execution for 0–65535 milliseconds. |
|  | POLLWAIT | Pause until a polled-interrupt occurs.    |

### SOUND

|  |         |                                                          |
|--|---------|----------------------------------------------------------|
|  | SOUND   | Generate tones or white noise.                           |
|  | FREQOUT | Generate one or two sine waves of specified frequencies. |
|  | DTMFOUT | Generate DTMF telephone tones.                           |

# ***BASIC Stamp Command Reference***

---

## **POWER CONTROL**

-   NAP      Nap for a short period. Power consumption is reduced.
-   SLEEP      Sleep for 1-65535 seconds. Power consumption is reduced.
-   END      Sleep until the power cycles or the PC connects. Power consumption is reduced.

## **PROGRAM DEBUGGING**

-   DEBUG      Send information to the PC for viewing in the Debug Terminal's Receive windowpane.
-  DEBUGIN<sup>2.5</sup>      Retrieve information from the user via the PC, entered into the Debug Terminal's Transmit windowpane.

## **SYNTAX CONVENTIONS**

**BOLD UPPER CASE** – any word that appears bold with all capital letters must be typed exactly as shown. These are all reserved words.

***Italics*** – italicized words must be replaced with your content.

**[ ]** – square brackets must be typed, in the position shown around the given syntax element. Only used with PBASIC 2.0 and 2.5.

**( )** – parentheses must be typed in the position shown around the given syntax element; only used this way with PBASIC 1.0.\*\*

**{ }** – curly braces indicate optional syntax items. They are not typed as part of any syntax other than compiler directives.

**|** – vertical line separates mutually exclusive syntax elements.

**, \ # =** – where they appear, commas, backslashes, pound signs, and equal signs must be typed in the positions shown.

\*\*NOTE: You may use parentheses to enclose expressions in PBASIC 2.0 and 2.5, but they are not necessary. Used within an expression, parentheses will change the order of evaluation. See page 103 for details and examples.

## 5: BASIC Stamp Command Reference – AUXIO

### **AUXIO**

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|

#### Function

Switch from control of main I/O pins to auxiliary I/O pins (on the BS2p40 only).

#### Quick Facts

Table 5.2: AUXIO Quick Facts.

|                  | BS2p, BS2pe, and BS2px                                                                                                          |
|------------------|---------------------------------------------------------------------------------------------------------------------------------|
| I/O pin IDs      | 0 – 15 (just like main I/O, but after AUXIO command, all references affect physical pins 21 – 36).                              |
| Special Notes    | The BS2p, BS2pe, and BS2px 24-pin modules accept this command, however, only the BS2p40 gives access to the auxiliary I/O pins. |
| Related Commands | MAINIO and IOTERM                                                                                                               |

#### Explanation

The BS2p, BS2pe, and BS2px are available as 24-pin modules that are pin compatible with the BS2, BS2e and BS2sx. Also available is a 40-pin module called the BS2p40, with an additional 16 I/O pins (for a total of 32). The BS2p40's extra, or auxiliary, I/O pins can be accessed in the same manner as the main I/O pins (by using the IDs 0 to 15) but only after issuing an AUXIO or IOTERM command. The AUXIO command causes the BASIC Stamp to affect the auxiliary I/O pins instead of the main I/O pins in all further code until the MAINIO or IOTERM command is reached, or the BASIC Stamp is reset or power-cycled. AUXIO is also used when setting the DIRS register for auxiliary I/O pins on the BS2p40.

When the BASIC Stamp module is reset, all RAM variables including DIRS and OUTS are cleared to zero. This affects both main and auxiliary I/O pins. On the BS2p24, BS2pe, and BS2px, the auxiliary I/O pins from the interpreter chip are not connected to physical I/O pins on the BASIC Stamp module. While not connected to anything, these pins do have internal pull-up resistors activated, effectively connecting them to Vdd. After reset, reading the auxiliary I/O from a BS2p24, BS2pe24, or BS2px24 will return all 1s.

## AUXIO - BASIC Stamp Command Reference

---

Here is a simple AUXIO example:

A SIMPLE AUXIO EXAMPLE.

```
HIGH 0 ' make P0 high
AUXIO ' select auxiliary pins
LOW 0 ' make X0 low
```

The first line of the above example will set I/O pin 0 of the main I/O pins (P0, physical pin 5) high. Afterward, the AUXIO command tells the BASIC Stamp that all commands following it should affect the auxiliary I/O pins. The following LOW command will set I/O pin 0 of the auxiliary I/O pins (X0, physical pin 21) low.

Note that the main I/O and auxiliary I/O pins are independent of each other; the states of the main I/O pins remain unchanged while the program affects the auxiliary I/O pins, and vice versa.

MAIN I/O AND AUXILIARY I/O PINS ARE INDEPENDENT AND UNAFFECTED BY CHANGES IN THE OPPOSITE GROUP.

Other commands that affect I/O group access are MAINIO and IOTERM.

### Demo Program (AUX\_MAIN\_TERM.bsp)

```
' AUX_MAIN_TERM.bsp
' This program demonstrates the use of the AUXIO, MAINIO and IOTERM
' commands to affect I/O pins in the auxiliary and main I/O groups.

' {$STAMP BS2p}
' {$PBASIC 2.5}

#SELECT $STAMP
#CASE BS2, BS2E, BS2SX
#ERROR "Program requires BS2p40"
#CASE BS2P, BS2PE, BS2PX
 DEBUG "Note: This program designed for the BS2p40.", CR
#ENDSELECT

port VAR Bit

Main:
DO
 MAINIO ' Switch to main I/O pins
 TOGGLE 0 ' Toggle state of I/O pin P0
 PWM 1, 100, 40 ' Generate PWM on I/O pin P1

 AUXIO ' Switch to auxiliary I/O pins
 TOGGLE 0 ' Toggle state of I/O pin X0
 PULSOUT 1, 1000 ' Generate a pulse on I/O pin X1
 PWM 2, 100, 40 ' Generate PWM on I/O pin X2
```



NOTE: This example program will tokenize with the 24-pin BS2p, BS2pe, and BS2px, but its effects can only be seen on the BS2p40. This program uses conditional compilation techniques; see Chapter 3 for more information.



## ***5: BASIC Stamp Command Reference – AUXIO***

---

```
IOTERM port ' Switch to main or aux I/Os
 ' -- depending on port
TOGGLE 3 ' Toggle state of I/O pin 3
 ' -- on main and aux, alternately
port = ~port ' Invert port
PAUSE 1000 ' 1 second delay
LOOP
END
```

## ***AUXIO - BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – BRANCH

### BRANCH

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**BRANCH** *Offset*, ( *Address0*, *Address1*, ...*AddressN* )



**BRANCH** *Offset*, [ *Address0*, *Address1*, ...*AddressN* ]

### Function

Go to the address specified by offset (if in range).

- **Offset** is a variable/constant/expression (0 – 255) that specifies the index of the address, in the list, to branch to (0 – N).
- **Addresses** are labels that specify where to go. BRANCH will ignore any list entries beyond offset 255.



NOTE: Expressions are not allowed as arguments on the BS1.

### Quick Facts

Table 5.3: BRANCH Quick Facts.

|                          | BS1                    | All BS2 Models |
|--------------------------|------------------------|----------------|
| Limit of Address Entries | Limited only by memory | 256            |
| Related Commands         | None                   | ON...GOTO      |

### Explanation

The BRANCH instruction is useful when you want to write something like this:

```
IF value = 0 THEN Case_0 ' when value is 0, jump to Case_0
IF value = 1 THEN Case_1 ' when value is 1, jump to Case_1
IF value = 2 THEN Case_2 ' when value is 2, jump to Case_2
```



You can use BRANCH to organize this into a single statement:

```
BRANCH value, [Case_0, Case_1, Case_2]
```

BS1 syntax not shown here.

This works exactly the same as the previous IF...THEN example. If the value isn't in range (in this case if *value* is greater than 2), BRANCH does nothing and the program continues with the next instruction after BRANCH.

BRANCH can be teamed with the LOOKDOWN instruction to create a simplified SELECT...CASE statement. See LOOKDOWN for an example.

## BRANCH – BASIC Stamp Command Reference

### Demo Program (BRANCH.bs1)



```
' BRANCH.bs1
' This program shows how the value of idx controls the destination of the
' BRANCH instruction.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL idx = B2

Main:
 DEBUG "idx: ", #idx, " "
 BRANCH idx, (Task_0, Task_1, Task_2) ' branch to task
 DEBUG "BRANCH target error...", CR, CR ' ... unless out of range

Next_Task:
 idx = idx + 1 // 4 ' force idx to be 0..3
 GOTO Main

Task_0:
 DEBUG "BRANCHed to Task_0", CR
 GOTO Next_Task

Task_1:
 DEBUG "BRANCHed to Task_1", CR
 GOTO Next_Task

Task_2:
 DEBUG "BRANCHed to Task_2", CR
 GOTO Next_Task
```

### Demo Program (BRANCH.bs2)



```
' BRANCH.bs2
' This program shows how the value of idx controls the destination of the
' BRANCH instruction.

' {$STAMP BS2}
' {$PBASIC 2.5}

idx VAR Nib

Main:
 DEBUG "idx: ", DEC1 idx, " "
 BRANCH idx, [Task_0, Task_1, Task_2] ' branch to task
 DEBUG "BRANCH target error...", CR, CR ' ... unless out of range

Next_Task:
 idx = idx + 1 // 4 ' force idx to be 0..3
```

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## ***5: BASIC Stamp Command Reference – BRANCH***

---

```
PAUSE 250
GOTO Main

Task_0:
 DEBUG "BRANChed to Task_0", CR
 GOTO Next_Task

Task_1:
 DEBUG "BRANChed to Task_1", CR
 GOTO Next_Task

Task_2:
 DEBUG "BRANChed to Task_2", CR
 GOTO Next_Task
```

## ***BRANCH – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – BUTTON

### BUTTON

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**BUTTON** *Pin, DownState, Delay, Rate, Workspace, TargetState, Address*

#### Function

Monitor and process a pushbutton input, perform auto-repeat, and branch to address if button is in target state. Button circuits may be active-low or active-high.



NOTE: Expressions are not allowed as arguments on the BS1. The range of the *Pin* argument on the BS1 is 0 – 7.

- **Pin** is a variable/constant/expression (0–15) that specifies the I/O pin to use. This pin will be set to input mode.
- **DownState** is a variable/constant/expression (0 or 1) that specifies which logical state occurs when the button is pressed.
- **Delay** is a variable/constant/expression (0 – 255) that specifies how long the button must be pressed before auto-repeat starts. The delay is measured in cycles of the BUTTON routine. Delay has two special settings: 0 and 255. If Delay is 0, BUTTON performs no debounce or auto-repeat. If Delay is 255, BUTTON performs debounce, but no auto-repeat.
- **Rate** is a variable/constant/expression (0 – 255) that specifies the number of cycles between auto-repeats. The rate is expressed in cycles of the BUTTON routine.
- **Workspace** is a byte variable used by BUTTON for workspace. It must be cleared to 0 before being used by BUTTON for the first time and should not be adjusted outside of the BUTTON command.  
NOTE: All RAM is cleared to 0 by default upon power-up or reset of the BASIC Stamp module.
- **TargetState** is a variable/constant/expression (0 or 1) that specifies which state the button should be in for a branch to occur. (0=not pressed, 1=pressed)
- **Address** is a label that specifies where to branch if the button is in the target state.

#### Explanation

When you press a button or flip a switch, the contacts make or break a connection. A brief (1 to 20-ms) burst of noise occurs as the contacts scrape and bounce against each other. By scanning an input within a loop to

## ***BUTTON – BASIC Stamp Command Reference***

---

ensure that the contact remains in a specified state for a minimum duration, spurious multiple inputs caused by contact noise can be eliminated. The **BUTTON** instruction helps prevent this noise from being interpreted as more than one switch action; this is the function of the *Delay* parameter. For a demonstration of switch bounce, see the demo program for the **COUNT** instruction. *Delay*, combined with the *Rate* argument, allows the programmer to control the rate at which multiple inputs are accepted by the BASIC Stamp.

**BUTTON** also lets PBASIC react to a button press the way your computer keyboard does to a key press. When you press a key, a character immediately appears on the screen. If you hold the key down, there's a delay, then a rapid-fire stream of characters appears on the screen. **BUTTON**'s auto-repeat function can be set up to work much the same way.

**BUTTON** is designed for use inside a program loop. Each time through the loop, **BUTTON** checks the state of the specified pin. When it first matches *DownState*, **BUTTON** begins the *Delay* countdown for auto-repeat. Then, in accordance with *TargetState*, it either branches to *Address* (*TargetState* = 1) or doesn't (*TargetState* = 0).

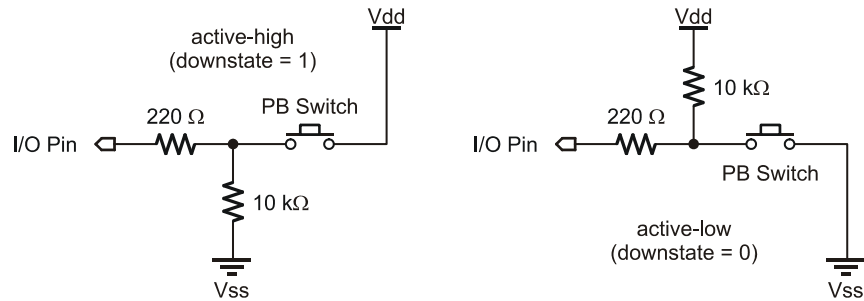
If the switch stays in *DownState*, **BUTTON** counts the number of program loops that execute. When this count equals *Delay*, **BUTTON** once again triggers the action specified by *TargetState* and *Address*. Hereafter, if the switch remains in *DownState*, **BUTTON** waits *Rate* number of cycles between actions. The *Workspace* variable is used by **BUTTON** to keep track of how many cycles have occurred since the *Pin* switched to *TargetState* or since the last auto-repeat.

**BUTTON** does not stop program execution. In order for its delay and auto-repeat functions to work properly, **BUTTON** must be executed from within a program loop.



## 5: BASIC Stamp Command Reference – BUTTON

**Figure 5.1:** Sample BUTTON circuits. Active-high (left) and active-low (right).



### Demo Program (BUTTON.bs1)

```
' BUTTON.bs1
' Connect an active-low circuit to pin P0 of the BS1. When you press the
' button, the DEBUG screen will display an asterisk (*). The program, as
' shown below, will print an asterisk at the first button press, then
' delay approximately one second (200 x 5 ms PAUSE) before auto-repeating
' at a rate of approximately 100 ms (5 x 20 ms). Feel free to modify the
' program to see the effects of your changes on the way BUTTON responds.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL Btn = 0
SYMBOL btnWrk = B2

Main:
 ' Try changing the Delay value (200) in BUTTON to see the effect of
 ' its modes: 0 = no delay; 1-254 = varying delays before auto-repeat;
 ' 255 = no auto-repeat (only one action per button press)
 '
 ' The BUTTON instruction will cause the program to branch to
 ' No_Press unless P0 = 0

 PAUSE 5
 BUTTON Btn, 0, 200, 20, btnWrk, 0, No_Press
 DEBUG "*"

No_Press:
 GOTO Main
```

## BUTTON – BASIC Stamp Command Reference

---

### Demo Program (BUTTON.bs2)

```
' BUTTON.bs2
' Connect an active-low circuit to pin P0 of the BS2. When you press the
' button, the DEBUG screen will display an asterisk (*). The program, as
' shown below, will print an asterisk at the first button press, then
' delay approximately one second (200 x 5 ms PAUSE) before auto-repeating
' at a rate of approximately 100 ms (5 x 20 ms). Feel free to modify the
' program to see the effects of your changes on the way BUTTON responds.

' {$STAMP BS2}
' {$PBASIC 2.5}

Btn PIN 0

btnWrk VAR Byte

Main:
' Try changing the Delay value (200) in BUTTON to see the effect of
' its modes: 0 = no delay; 1-254 = varying delays before auto-repeat;
' 255 = no auto-repeat (only one action per button press)
'
' The BUTTON instruction will cause the program to branch to
' No_Press unless P0 = 0

PAUSE 5
BUTTON Btn, 0, 200, 20, btnWrk, 0, No_Press
DEBUG "***

No_Press:
GOTO Main
```



NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## 5: BASIC Stamp Command Reference – COMPARE

### COMPARE

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



#### COMPARE Mode, Variable

#### Function

Enable or disable comparator, compare voltages on P1 and P2 and retrieve comparison result to store in *Variable*.

- **Mode** is a variable/constant/expression (0 – 2) that enables or disables the comparator (input pins P1 and P2) and determines if the optional comparator output pin (pin P0) is enabled or not. See Table 5.4 for an explanation of the *Mode* values.
- **Variable** is a variable (usually a bit) in which the comparison result is stored.

#### Quick Facts

Table 5.4: COMPARE Quick Facts.

|                        | BS2px                                             |
|------------------------|---------------------------------------------------|
| <b>Mode Values</b>     | 0: Disables comparator                            |
|                        | 1: Enables comparator with P0 as result output    |
|                        | 2: Enables comparator without P0 as result output |
| <b>Variable Values</b> | 0: Voltage P1 > P2; P0 optionally outputs 0       |
|                        | 1: Voltage P1 < P2; P0 optionally outputs 1       |

#### Explanation

The COMPARE command enables or disables the built-in comparator hardware on the BS2px's I/O pins P0, P1, and P2. I/O pins P1 and P2 are the comparator inputs and P0 is optionally the comparator result output pin.

By default, the comparator feature is disabled. Using the COMPARE command with a *Mode* argument of 1 or 2 enables the comparator feature (using input pins P1 and P2) and returns the result of the comparison in *Variable*. If *Mode* is 1, the result of the comparison is also output on I/O pin P0. The following is an example of the COMPARE command:

```
Result VAR Bit
COMPARE 1, Result
```

This example enables the comparator (setting P0 to output the result, with P1 and P2 as the comparator inputs) and writes the result of the comparison into *Result*. Both *Result* and the output pin P0 will be 0 if the

## COMPARE – BASIC Stamp Command Reference

---

input voltage on P1 was greater than that of P2. *Result* and the output pin P0 will be 1 if the input voltage on P1 was less than that of P2.

Note that the comparator hardware operates independently of the execution speed of the BS2px and will continue to run and update P0 if *Mode* = 1, even during sleep mode (execution of END, NAP, POLLWAIT 8, or SLEEP commands). To avoid spurious current draw during sleep mode, disable the comparator first.

### Demo Program (COMPARE.bpx)



NOTE: This example program can be used only with the BS2px.

```
' COMPARE.bpx
' This example demonstrates the use of the COMPARE command.
' Connect two variable voltage sources (0 to 5 volts) on I/O pins
' P1 and P2 (or a button on each pin connected to ground). Run the program
' and watch the Debug Terminal display as you adjust the variable voltage
' or press the buttons.

' {$STAMP BS2px}
' {$PBASIC 2.5}

Result VAR Bit

#IF $STAMP <> BS2PX #THEN
 #ERROR "This program requires a BS2px."
#ENDIF

Setup:
 CONFIGPIN DIRECTION, %0000000000000001 'P0 = output, all others = input
 CONFIGPIN PULLUP, %0000000000000110 'Enable pull-ups on P1 and P2
 DEBUG "BS2px COMPARATOR DEMONSTRATION", CR,
 "=====", CR,
 "Input Voltage: P1 > P2", CR,
 "Output State: P0 = 0"

Main:
DO
 COMPARE 1, Result 'Display P1/P2 comparison
 IF Result = 0 THEN
 DEBUG CR$RXY,18,2,">"
 ELSE
 DEBUG CR$RXY,18,2,"<"
 ENDIF
 DEBUG CR$RXY,20,3,BIN1 Result
LOOP
```

## 5: BASIC Stamp Command Reference – CONFIGPIN

### CONFIGPIN

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



### CONFIGPIN Mode, PinMask

#### Function

Configure special properties of I/O pins.

- **Mode** is a variable/constant/expression (0 – 3), or one of four predefined symbols, that specifies the I/O pin property to configure: Schmitt Trigger, Logic Threshold, Pull-up Resistor or Output Direction. See Table 5.5 for an explanation of *Mode* values.
- **PinMask** is a variable/constant/expression (1 – 65535) that indicates how *Mode* is applied to I/O pins. Each bit of *PinMask* corresponds to an individual I/O pin. A high bit (1) enables the *Mode* and a low bit (0) disables the *Mode* on the corresponding I/O pin.

#### Quick Facts

Table 5.5: CONFIGPIN Quick Facts.

|                                              | BS2px                                                   |
|----------------------------------------------|---------------------------------------------------------|
| <b>Mode Values</b>                           | 0 (or SCHMITT): Schmitt Trigger                         |
|                                              | 1 (or THRESHOLD): Logic Threshold                       |
|                                              | 2 (or PULLUP): Pull-up Resistor                         |
|                                              | 3 (or DIRECTION): Output Direction                      |
| <b>Related Commands (For DIRECTION Mode)</b> | INPUT and OUTPUT, and the DIRx = # assignment statement |

#### Explanation

The CONFIGPIN command enables or disables special I/O pin properties on all 16 I/O pins at once. There are four properties, or modes, available: Schmitt Trigger, Logic Threshold, Pull-up Resistor, and Output Direction. Each I/O pin on the BS2px contains special hardware dedicated to each of these properties.

OUTPUT DIRECTION.

By default, all BASIC Stamp I/O pins are set to inputs. Enabling the Output Direction mode sets an I/O pin's direction to output. Disabling the Output Direction mode sets an I/O pin's direction to input. This has the same effect as using the OUTPUT or INPUT commands, or the DIRx = # assignment statement to configure I/O pin directions. The following is an example of the CONFIGPIN command using the Output Direction mode:

```
CONFIGPIN DIRECTION, %0000000100010011
```

## CONFIGPIN – BASIC Stamp Command Reference

Every high bit (1) in the *PinMask* argument enables the output direction for the corresponding I/O pin while every low bit (0) disables the output direction. In the above example, I/O pins 8, 4, 1, and 0 are set to the output direction and all other I/O pins are set to the input direction. This is similar to the following statement:

```
DIRS = %0000000100010011
```

Pull-up resistors are commonly used in circuitry where a component, such as a button, provides an open/drain signal; the signal is either floating (open) or is driven to ground (drain). Since the BASIC Stamp input pins must always be connected to either 5 volts or ground (0 volts) in order to read a reliable logic state with them, a pull-up resistor is required on circuitry, such as the button circuit mentioned above, so that the signal is never left floating (electrically disconnected).

The following example enables internal pull-up resistors on I/O pins 15, 12, 6, and 3, and disables internal pull-up resistors on all other I/O pins:

```
CONFIGPIN PULLUP, %1001000001001000
```

Note that the internal pull-up resistors are intentionally weak, about 20 k $\Omega$ . Additionally, the internal pull-up resistors can be activated for all pins, regardless of pin direction, but really matter only when the associated pin is set to input mode.

An input pin's logic threshold determines the voltage levels that are interpreted as logic high (1) and logic low (0). Most microcontrollers, and other integrated circuits use one of two types of logic threshold: TTL Level or CMOS Level. The BASIC Stamp I/O pins are, by default, configured for TTL level logic thresholds. Figure 5.2 is an illustration of the difference between TTL and CMOS logic levels.

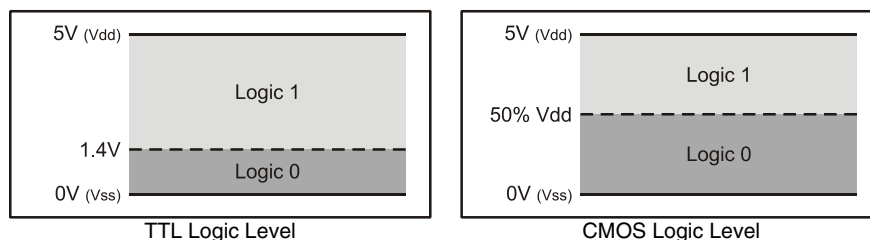


**Figure 5.2:** TTL and CMOS Logic Level Threshold Voltages

## 5: BASIC Stamp Command Reference – CONFIGPIN

The logic threshold for TTL is 1.4 volts; a voltage below 1.4 is considered to be a logic 0 while a voltage above 1.4 is considered to be a logic 1. The logic threshold for CMOS is 50% of Vdd; a voltage below  $\frac{1}{2}$  Vdd is considered a logic 0 while a voltage above  $\frac{1}{2}$  Vdd is considered a logic 1.

For the CONFIGPIN command's THRESHOLD mode, a high bit (1) in the *PinMask* argument sets the corresponding I/O pin to CMOS threshold level, and a low bit sets it to a TTL threshold level. The following example sets CMOS threshold level on I/O pins 3, 2, 1, and 0, and TTL threshold level on all other I/O pins.

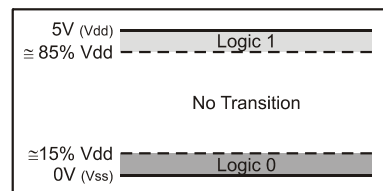
```
CONFIGPIN THRESHOLD, %0000000000001111
```

The threshold level can be set for all pins, regardless of pin direction, but really matters only when the associated pin is set to input mode.

### SCHMITT TRIGGER

Normally, if a signal on an input pin is somewhat noisy (the voltage level randomly rises and falls beyond the logic threshold boundary) then reading that pin's input value will result in spurious highs and lows (1s and 0s). Schmitt Triggers are circuits that make inputs more steady and reliable by adding a region of hysteresis around the logic threshold that the signal must completely traverse before the logic level is interpreted as being changed. By default BASIC Stamp I/O pins are set to normal input mode, but the BS2px can be configured for Schmitt Trigger mode as well. Figure 5.3 illustrates Schmitt Trigger characteristics.

**Figure 5.3:** Schmitt Trigger Characteristics



In Schmitt Trigger mode, the threshold for a logic 0 is approximately 15% of Vdd and the threshold for a logic 1 is approximately 85% of Vdd. The input pin defaults to an unknown state until the initial voltage crosses a logic 0 or logic 1 boundary. Thereafter, the voltage must cross above 85% of Vdd to be interpreted as a logic 1 and must cross below 15% of Vdd to be interpreted as a logic 0. If the voltage transitions somewhere between the two thresholds, the interpreted logic state remains the same as the previous state.

## CONFIGPIN – BASIC Stamp Command Reference

For the CONFIGPIN command's SCHMITT mode, a high bit (1) in the *PinMask* argument enables the Schmitt Trigger on the corresponding I/O pin and a low bit (0) disables the Schmitt Trigger. The following example sets Schmitt Triggers on I/O pins 7, 6, 5, and 4, and sets all other I/O pins to normal mode.

```
CONFIGPIN SCHMITT, %0000000011110000
```

Schmitt Trigger mode can be activated for all pins, regardless of pin direction, but really matters only when the associated pin is set to input mode.

### Demo Program (CONFIGPIN.bpx)



NOTE: This example program can be used only with the BS2px.

```
' CONFIGPIN.BPX
' This example demonstrates the use of the CONFIGPIN command.
' All I/O pins are set to inputs with various combinations of
' Pull-Up Resistor, Logic Threshold and Schmitt-Trigger properties.
' While running, this program will constantly display the state of all
' input pins along with an indication of the configuration for each group
' of pins. Try connecting different input signals to the I/O pins (such as
' buttons, a function generator with a slowing sweeping signal (0 to 5
' VDC)) or simply running your fingers across the I/O pins and note how
' they react based upon their configured property.

' {$STAMP BS2px}
' {$PBASIC 2.5}

#IF $STAMP <> BS2PX #THEN
 #ERROR "This program requires a BS2px."
#ENDIF

Setup:
CONFIGPIN DIRECTION, %0000000000000000 'Set all I/O pins to inputs
CONFIGPIN PULLUP, %111111111110000 'Enable pull-ups on pins 4 - 15
CONFIGPIN THRESHOLD, %0000111100000000 'Set P8-P11 to CMOS, others TTL
CONFIGPIN SCHMITT, %1111000000000000 'Enable Schmitt-Triggers P12-P15

DEBUG CLS
DEBUG " BS2px INPUT PIN CONFIGURATION TEST", CR,
"=====", CR,
" P15-P12: Pull-Up Resistors, TTL & Schmitt-Triggers", CR,
" /", CR,
" / P11-P8: Pull-Up Resistors & CMOS", CR,
" / /", CR,
" | / P7-P4: Pull-Up Resistors & TTL", CR,
" | | /", CR,
" | | P3-P0: Normal", CR,
" | | /", CR,
```



## ***5: BASIC Stamp Command Reference – CONFIGPIN***

---

```
 " | | | |", CR,
 "-----"

Main:
DO
 'Display input pin states
 DEBUG CRSRXY,0,12, BIN4 IND, " ", BIN4 INC, " ", BIN4 INB, " ", BIN4 INA
LOOP
```

## ***CONFIGPIN – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – COUNT

### COUNT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**COUNT** *Pin, Duration, Variable*

#### Function

Count the number of cycles (0-1-0 or 1-0-1) on the specified pin during the *Duration* time frame and store that number in *Variable*.

- **Pin** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This pin will be set to input mode.
- **Duration** is a variable/constant/expression (1 – 65535) specifying the time during which to count. The unit of time for *Duration* is described in Table 5.6.
- **Variable** is a variable (usually a word) in which the count will be stored.

#### Quick Facts

Table 5.6: COUNT Quick Facts.

**NOTE:** All timing values are approximate.

|                                        | BS2, BS2e        | BS2sx                   | BS2p                    | BS2pe                  | BS2px                   |
|----------------------------------------|------------------|-------------------------|-------------------------|------------------------|-------------------------|
| <b>Units in Duration</b>               | 1 ms             | 400 $\mu$ s             | 287 $\mu$ s             | 720 $\mu$ s            | 287 $\mu$ s             |
| <b>Duration range</b>                  | 1 ms to 65.535 s | 400 $\mu$ s to 26.214 s | 287 $\mu$ s to 18.809 s | 720 $\mu$ s to 47.18 s | 287 $\mu$ s to 18.809 s |
| <b>Minimum pulse width</b>             | 4.16 $\mu$ s     | 1.66 $\mu$ s            | 1.20 $\mu$ s            | 3.0 $\mu$ s            | 1.20 $\mu$ s            |
| <b>Maximum frequency (square wave)</b> | 120,000 Hz       | 300,000 Hz              | 416,700 Hz              | 166,667 Hz             | 416,700 Hz              |
| <b>Related Command</b>                 | PULSIN           |                         |                         |                        |                         |

#### Explanation

The COUNT instruction makes the *Pin* an input, then for the specified *Duration* of time, counts cycles on that pin and stores the total in *Variable*. A cycle is a change in state from 1 to 0 to 1, or from 0 to 1 to 0.

According to Table 5.6, COUNT on the BS2 can respond to transitions (pulse widths) as small as 4.16 microseconds ( $\mu$ s). A cycle consists of two transitions (e.g., 0 to 1, then 1 to 0), so COUNT (on the BS2) can respond to square waves with periods as short as 8.32  $\mu$ s; up to 120 kilohertz (kHz) in frequency. For non-square waves (those whose high time and low time are unequal), the shorter of the high and low times must be at least 4.16  $\mu$ s in

## COUNT – BASIC Stamp Command Reference

width (on the BS2). Refer to Table 5.6 for data on other BASIC Stamp models.

If you use COUNT on slowly changing analog waveforms like sine waves, you may find that the value returned is higher than expected. This is because the waveform may pass through the BASIC Stamp module's logic threshold slowly enough that noise causes false counts. You can fix this by passing the signal through a Schmitt Trigger, like one of the inverters of a 74HCT14. Or, you may use the BS2px's built-in Schmitt-Trigger pin property; see the CONFIGPIN section beginning on page 143 for details.

### Demo Program (COUNT.bs2)

```
' COUNT.bs2
' Connect an active-low button circuit shown in the BUTTON command
' description to pin P0 of the BS2. The DEBUG screen will prompt you to
' press the button as quickly as possible for a 1-second count. When the
' count is done, the screen will display your "score," the total number of
' cycles registered by COUNT. Note that this score will almost always
' be greater than the actual number of presses because of switch contact
' bounce.

' {$STAMP BS2}
' {$PBASIC 2.5}

PushBtn PIN 0 ' pushbutton on P0

#SELECT $STAMP
#CASE BS2, BS2E
 DurAdj CON $100 ' / 1
#CASE BS2SX
 DurAdj CON $280 ' / 0.400
#CASE BS2P, BS2PX
 DurAdj CON $37B ' / 0.287
#CASE BS2PE
 DurAdj CON $163 ' / 0.720
#ENDSELECT

Capture CON 1000 ' 1 second

cycles VAR Word ' counted cycles

Main:
 DO
 DEBUG CLS,
 "How many times can you press the button in 1 second?", CR
 PAUSE 1000
 DEBUG "Ready, set... "
```



NOTE: This example program can be used with all BS2 models. This program uses conditional compilation techniques; see Chapter 3 for more information.

## ***5: BASIC Stamp Command Reference – COUNT***

---

```
PAUSE 500
DEBUG "GO!", CR
COUNT PushBtn, (Capture */ DurAdj), cycles
DEBUG CR, "Your score: ", DEC cycles, CR
PAUSE 3000
DEBUG "Press button to go again."
DO : LOOP UNTIL (PushBtn = 0) ' wait for button press
LOOP
END
```

## ***COUNT – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – DATA

### DATA

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|



(See EEPROM)



{ *Symbol* } DATA *Dataltem* { , *Dataltem*... }

### Function

Write data to the EEPROM during program download.

- ***Symbol*** is an optional, unique symbol name that will be automatically defined as a constant equal to the location number of the first data item.
- ***Dataltem*** is a constant/expression (0 – 65535) indicating a value, and optionally how to store the value.

### Quick Facts

Table 5.7: DATA Quick Facts.

|                  | All BS2 Models                                                                                                                         |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| Special Notes    | Writes values to EEPROM during download in blocks of 16 bytes. Writes byte or word-sized values. Can be used to decrease program size. |
| Related Commands | READ and WRITE                                                                                                                         |

### Explanation

When you download a program into the BASIC Stamp, it is stored in the EEPROM starting at the highest address (2047) and working towards the lowest address. Most programs don't use the entire EEPROM, so the lower portion is available for other uses. The DATA directive allows you to define a set of data to store in the available EEPROM locations. It is called a "directive" rather than a "command" because it performs an activity at compile-time rather than at run-time (i.e.: the DATA directive is not downloaded to the BASIC Stamp, but the data it contains is downloaded).

WRITING SIMPLE, SEQUENTIAL DATA.

The simplest form of the DATA directive is something like the following:

```
DATA 100, 200, 52, 45
```

This example, when downloaded, will cause the values 100, 200, 52 and 45 to be written to EEPROM locations 0, 1, 2 and 3, respectively. You can then use the READ and WRITE commands in your code to access these locations and the data you've stored there.

## DATA – BASIC Stamp Command Reference

DATA uses a counter, called a pointer, to keep track of available EEPROM addresses. The value of the pointer is initially 0. When a program is downloaded, the DATA directive stores the first byte value at the current pointer address, then increments (adds 1 to) the pointer. If the program contains more than one DATA directive, subsequent DATAs start with the pointer value left by the previous DATA. For example, if the program contains:

```
DATA 72, 69, 76, 76, 79
DATA 104, 101, 108, 108, 111
```

The first DATA directive will start at location 0 and increment the pointer for each data value it stores (1, 2, 3, 4 and 5). The second DATA directive will start with the pointer value of 5 and work upward from there. As a result, the first 10 bytes of EEPROM will look like the following:

|          | EEPROM Location (address) |    |    |    |    |     |     |     |     |     |
|----------|---------------------------|----|----|----|----|-----|-----|-----|-----|-----|
|          | 0                         | 1  | 2  | 3  | 4  | 5   | 6   | 7   | 8   | 9   |
| Contents | 72                        | 69 | 76 | 76 | 79 | 104 | 101 | 108 | 108 | 111 |

THE DATA POINTER (COUNTER).

**Table 5.8:** Example EEPROM Storage.

What if you don't want to store values starting at location 0? Fortunately, the DATA directive has an option to specify the next location to use. You can specify the next location number (to set the pointer to) by inserting a *DatalItem* in the form @x ;where x is the location number. The following code writes the same data in Table 5.8 to locations 100 through 109:

WRITING DATA TO OTHER LOCATIONS.

```
DATA @100, 72, 69, 76, 76, 79, 104, 101, 108, 108, 111
```

In this example, the first *DatalItem* is @100. This tells the DATA directive to store the following *DatalItem*(s) starting at location 100. All the *DatalItems* to the right of the @100 are stored in their respective locations (100, 101, 102... 109).

In addition, the DATA directive allows you to specify new starting locations at any time within the *DatalItem* list. If, for example, you wanted to store 56 at location 100 and 47 at location 150 (while leaving every other location intact), you could type the following:

```
DATA @100, 56, @150, 47
```

If you have multiple DATA directives in your program, it may be difficult to remember exactly what locations contain the desired data. For this reason, the DATA directive can optionally be prefixed with a unique

AUTOMATIC CONSTANTS FOR DEFINED DATA.



## 5: BASIC Stamp Command Reference – DATA

---

symbol name. This symbol becomes a constant that is set equal to the location number of the first byte of data within the directive. For example,

```
MyNumbers DATA @100, 72, 73
```

This would store the values 72 and 73 starting with location 100 and will create a constant, called *MyNumbers*, which is set equal to 100. Your program can then use the *MyNumbers* constant as a reference to the start of the data within a READ or WRITE command. Each DATA directive can have a unique symbol preceding it, allowing you to reference the data defined at different locations.

### RESERVING EEPROM LOCATIONS.

There may be a time when you wish to reserve a section of EEPROM for use by your BASIC code, but not necessarily store data there to begin with. To do this, simply specify a *DataItem* within parentheses, as in:

```
DATA @100, (20)
```

The above DATA directive will reserve 20 bytes of EEPROM, starting with location 100. It doesn't store any values there, rather it simply leaves the data as it is and increments DATA's location pointer by 20. A good reason to do this is when you have a program already downloaded into the BASIC Stamp that has created or manipulated some data in EEPROM. To protect that section of EEPROM from being overwritten by your next program (perhaps a new version of the same program) you can reserve the space as shown above. The EEPROM's contents from locations 100 to 119 will remain intact. NOTE: This only "reserves" the space for the program you are currently downloading; the BASIC Stamp does not know to "reserve" the space for future programs. In other words, make sure use this feature of the DATA directive in every program you download if you don't want to risk overwriting valuable EEPROM data.

### IMPORTANT CONCEPT: HOW DATA AND PROGRAMS ARE DOWNLOADED INTO EEPROM.

It is important to realize that EEPROM is not overwritten during programming unless it is needed for program storage, or is filled by a DATA directive specifying data to be written. **During downloading, EEPROM is always written in 16-byte sections if, and only if, any location within that section needs writing.**

### WRITING A BLOCK OF THE SAME VALUE.

DATA can also store the same number in a block of consecutive locations. This is similar to reserving a block of EEPROM, above, but with a value added before the first parenthesis.

## DATA – BASIC Stamp Command Reference

---

For example,

```
DATA @100, 0 (20)
```

This statement writes the value 0 in all the EEPROM locations from 100 to 119.

A common use for DATA is to store strings; sequences of bytes representing text. PBASIC converts quoted text like "A" into the corresponding ASCII character code (65 in this case). To make data entry easier, you can place quotes around a whole chunk of text used in a DATA directive, and PBASIC will understand it to mean a series of bytes (see the last line of code below). The following three DATA directives are equivalent: WRITING TEXT STRINGS.

```
DATA 72, 69, 76, 76, 79
DATA "H", "E", "L", "L", "O"
DATA "HELLO"
```

All three lines of code, above, will result in the numbers 72, 69, 76, 76, and 79 being stored into EEPROM upon downloading. These numbers are simply the ASCII character codes for "H", "E", "L", "L", and "O", respectively. See the Demo program, below, for an example of storing and reading multiple text strings.

The EEPROM is organized as a sequential set of byte-sized memory locations. By default, the DATA directive stores bytes into EEPROM. If you try to store a word-sized value (ex: DATA 1125) only the lower byte of the value will be stored. This does not mean that you can't store word-sized values, however. A word consists of two bytes, called a low-byte and a high-byte. If you wanted to store the value 1125 using the DATA directive, simply insert the prefix "word" before the number, as in: WRITING WORD VALUES VS. BYTE VALUES.

```
DATA Word 1125
```

The directive above will automatically break the word-sized value into two bytes and store them into two sequential EEPROM locations (the low-byte first, followed by the high-byte). In this case, the low-byte is 101 and the high byte is 4 and they will be stored in locations 0 and 1, respectively. If you have multiple word-sized values, you must prefix each value with "word", as in:

```
DATA Word 1125, Word 2000
```

## 5: BASIC Stamp Command Reference – DATA

To retrieve a word-sized value, you'll need to use the WORD modifier in the READ command and a word-sized variable.

Finally, a *DataItem* may be defined using a simple expression with the binary operators shown in Table 4.5. For example,

```
MinLvl CON 10

myLvl VAR Byte

Level1 DATA MinLvl + 10
Level2 DATA MinLvl * 5 + 21

READ Level2, myLvl ' read EE location Level2
DEBUG DEC myLvl ' show value of myLvl (71)
```

All 2

### Demo Program (DATA.bs2)

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

```
' DATA.bs2
' This program stores a number of large text strings into EEPROM with the
' DATA directive and then sends them, one character at a time via the DEBUG
' command. This is a good demonstration of how to save program space by
' storing large amounts of data in EEPROM directly, rather than embedding
' the data into DEBUG commands.

' {$STAMP BS2}
' {$PBASIC 2.5}

idx VAR Word ' current location number
phrase VAR Nib ' current phrase number
char VAR Byte ' character to print

' ----- Define all text phrases (out of order, just for fun!) -----
'
Text1 DATA "Here is the first part of a large chunk of textual "
 DATA "data ", CR, "that needs to be transmitted. There's "
 DATA "a 5 second delay", CR, "between text paragraphs. ", CR
 DATA CR, 0

Text3 DATA "The alternative (having multiple DEBUGs or SEROUTs, "
 DATA "each ", CR, "with their own line of text) consumes "
 DATA "MUCH more EEPROM ", CR, "(program) space. ", CR
 DATA CR, 0

Text6 DATA "The 0 is used by this program to indicate we've "
 DATA "reached the ", CR, "End of Text. The Main routine "
 DATA "pauses in between each block of", CR, "text, and then "
 DATA "uses a LOOKUP command to retrieve the location ", CR
 DATA "of the next desired block of text to print. ", 0
```

## DATA – BASIC Stamp Command Reference

---

```
Text4 DATA CLS, "This program also demonstrates retrieving data "
 DATA "out of order ", CR, "in relation to the way it is "
 DATA "stored in EEPROM. Additionally,", CR, "control codes "
 DATA "(like carriage-returns, clear-screens, etc) can ", CR
 DATA "be embedded right in the data, as it is here. ", CR
 DATA CR, 0

Text2 DATA "This is an example of a good way to save space in "
 DATA "your ", CR, "BASIC Stamp's program by storing data "
 DATA "into EEPROM and ", CR, "retrieving it, one byte at a "
 DATA "time, and transmitting it ", CR, "with just a single "
 DATA "DEBUG (or SEROUT) command.", CR, CR, 0

Text5 DATA "The Print_It routine simply takes the idx variable, "
 DATA "retrieves", CR, "the character at the EEPROM location "
 DATA "pointed to by it, and ", CR, "prints it to the screen "
 DATA "until it finds a byte with a value of 0.", CR, CR, 0

Main:
 DEBUG CLS ' Clear DEBUG window
 FOR phrase = 1 TO 6 ' Print blocks one by one
 LOOKUP (phrase - 1),
 [Text1, Text2, Text3, Text4, Text5, Text6], idx
 GOSUB Print_It
 PAUSE 5000 ' Pause for 5 seconds
 NEXT
END

Print_It:
 DO
 READ idx, char ' Get next character
 idx = idx + 1 ' Point to next location
 IF (char = 0) THEN EXIT ' If 0, we're done with block
 DEBUG char ' Otherwise, transmit it
 LOOP
 RETURN ' Return to the main routine
```

## 5: BASIC Stamp Command Reference – DEBUG

### DEBUG

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**DEBUG** *OutputData* { , *OutputData* }

#### Function

Display information on the PC screen within the BASIC Stamp Editor's Debug Terminal. This command can be used to display text or numbers in various formats on the PC screen in order to follow program flow (called debugging) or as part of the functionality of the BASIC Stamp application.

- **OutputData** is a variable/constant/expression (0 – 65535) that specifies the information to output. Valid data can be ASCII characters (text strings and control characters), decimal numbers (0 - 65535), hexadecimal numbers (\$0000 - \$FFFF) or binary numbers (up to %1111111111111111). Data can be modified with special formatters as explained below.



NOTE: Expressions are not allowed as arguments on the BS1. The only constant allowed for the BS1 DEBUG command is a text string.

#### Quick Facts

Table 5.9: DEBUG Quick Facts.

|                         | BS1                                                                        | BS2, BS2e, BS2sx<br>BS2p, BS2pe                                | BS2px                                                           |
|-------------------------|----------------------------------------------------------------------------|----------------------------------------------------------------|-----------------------------------------------------------------|
| <b>Serial Protocol</b>  | Asynchronous<br>4800, N, 8, 1<br>True polarity<br>Custom packetized format | Asynchronous<br>9600, N, 8, 1<br>Inverted polarity<br>Raw data | Asynchronous<br>19200, N, 8, 1<br>Inverted polarity<br>Raw data |
| <b>Related Commands</b> | None                                                                       | SEROUT and DEBUGIN                                             |                                                                 |

#### Explanation

DEBUG provides a convenient way for your BASIC Stamp to send messages to the PC screen while running. The name “debug” suggests its most popular use; debugging programs by showing you the value of a variable or expression, or by indicating what portion of a program is currently executing. DEBUG is also a great way to rehearse programming techniques. Throughout this manual, we use DEBUG to give you immediate feedback on the effects of instructions. The following example demonstrates using the DEBUG command to send the text string message “Hello World!”.

```
DEBUG "Hello, World!"
```

After you download this one-line program, the BASIC Stamp Editor will open a Debug Terminal on your PC screen and wait for a response from

## ***DEBUG – BASIC Stamp Command Reference***

---

the BASIC Stamp. A moment later, the phrase "Hello World!" will appear. Note that if you close the Debug Terminal, your program keeps executing, but you can't see the DEBUG data anymore.

Multiple pieces of data can be sent with one DEBUG command by separating the data with commas (.). The following example produces exactly the same results as the example above.

```
DEBUG "Hello ", "World!"
```

DEBUG can also print and format numbers (values) from both constants and variables. The formatting methods for DEBUG are very different for the BS1, than for any other BASIC Stamp. Please read the appropriate sections, below, carefully.

### **BASIC Stamp 1 Formatting**

On the BS1, the DEBUG command, by default, displays numbers in the format "symbol = value" (followed by a carriage return), using the decimal number system. For example,

```
SYMBOL x = B2
```

```
x = 75
DEBUG x
```

displays "x = 75" on the screen. To display the value, in decimal, without the "x =" text, use the value formatter (#) before the variable name. For example, the following code displays "75" on the screen.

```
SYMBOL x = B2
```

```
x = 75
DEBUG #x
```

To display numbers in hexadecimal or binary form, use the \$ or % formatter, respectively. The code below displays the same number in its hexadecimal and binary forms.

```
SYMBOL x = B2
```

```
x = 75
DEBUG $x, %x
```



DISPLAYING DECIMAL NUMBERS (BS1).

DISPLAYING HEXADECIMAL OR BINARY NUMBERS (BS1).

## 5: BASIC Stamp Command Reference – DEBUG

After running the above code, "x = \$4B" and "x = %01001011" should appear on the screen. To display hexadecimal or binary values without the "symbol = " preface, use the value formatter (#) before the \$ and %, as shown below:

```
SYMBOL x = B2

x = 75
DEBUG #x, "as HEX is ", #$x ' displays "75 as HEX is $4B"
DEBUG #x, "as BINARY is ", %#x ' displays "75 as BINARY is %01001011"
```

DISPLAYING ASCII CHARACTERS (BS1).

To display a number as its ASCII character equivalent, use the ASCII formatter (@).

```
SYMBOL x = B2

x = 75
DEBUG @x
```

**Table 5.10:** DEBUG Formatters for the BASIC Stamp 1.

| Formatter | Description                                                                                                                                                                         |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| #         | Suppresses the "symbol = x" format and displays only the 'x' value. The default format is decimal but may be combined with any of the formatters below (ex: #x to display: x value) |
| @         | Displays "symbol = 'x'" + carriage return; where x is an ASCII character.                                                                                                           |
| \$        | Hexadecimal text.                                                                                                                                                                   |
| %         | Binary text.                                                                                                                                                                        |

USING CR AND CLS (BS1).

Two pre-defined symbols, CR and CLS, can be used to send a carriage-return or clear-screen command to the Debug Terminal. The CR symbol will cause the Debug Terminal to start a new line and the CLS symbol will cause the Debug Terminal to clear itself and place the cursor at the top-left corner of the screen. The following code demonstrates this.

```
DEBUG "You can not see this.", CLS, "Here is line 1", CR, "Here is line 2"
```

When the above is run, the final result is "Here is line 1" on the first line of the screen and "Here is line 2" on the second line. You may or may not have seen "You can not see this." appear first. This is because it was immediately followed by a clear-screen symbol, CLS, which caused the display to clear the screen before displaying the rest of the information.

NOTE: The rest of this discussion does not apply to the BASIC Stamp 1.

## DEBUG – BASIC Stamp Command Reference

---

### BASIC Stamp 2, 2e, 2sx, 2p, 2pe, and 2px Formatting



On the all BASIC Stamp models 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:

DISPLAYING ASCII CHARACTERS.

```
x VAR Byte
```

```
x = 65
```

```
DEBUG x ' Try to show decimal value of x
```

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.

DISPLAYING DECIMAL NUMBERS.

```
x VAR Byte
```

```
x = 65
```

```
DEBUG DEC x ' Show decimal value of x
```

In addition to decimal (DEC), DEBUG can display numbers in hexadecimal (HEX) and binary (BIN). See Table 5.11 and Table 5.12 for a complete list of formatters.

DISPLAYING HEXADECIMAL AND BINARY NUMBERS.

Expressions are allowed within the DEBUG command arguments as well. In the above code, `DEBUG DEC x+25` would yield “90” and `DEBUG DEC x*10/2-3` would yield “322”.

EXPRESSIONS IN DEBUG COMMANDS.



## 5: BASIC Stamp Command Reference – DEBUG

**Table 5.11:** DEBUG Special Formatters for all BS2 models.

| Special Formatter        | Action                                                                                                                                                                                                                |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ?                        | Displays "symbol = x' + carriage return; where x is a number. Default format is decimal, but may be combined with conversion formatters (ex: BIN ? x to display "x = binary_number").                                 |
| ASC ?                    | Displays "symbol = 'x'" + carriage return; where x is an ASCII character.                                                                                                                                             |
| STR <i>ByteArray</i> {L} | Send character string from an array. The optional \L argument can be used to limit the output to L characters, otherwise, characters will be sent up to the first byte equal to 0 or the end of RAM space is reached. |
| REP <i>Byte</i> \L       | Send a string consisting of <i>Byte</i> repeated L times (ex: REP "X"\10 sends "XXXXXXXXXX").                                                                                                                         |

**Table 5.12:** DEBUG Conversion Formatters for all BS2 models.

| Conversion Formatter | Type of Number                                                              | Notes |
|----------------------|-----------------------------------------------------------------------------|-------|
| DEC{1..5}            | Decimal, optionally fixed to 1 – 5 digits                                   | 1     |
| SDEC{1..5}           | Signed decimal, optionally fixed to 1 – 5 digits                            | 1,2   |
| HEX{1..4}            | Hexadecimal, optionally fixed to 1 – 4 digits                               | 1,3   |
| SHEX{1..4}           | Signed hexadecimal, optionally fixed to 1 – 4 digits                        | 1,2   |
| IHEX{1..4}           | Indicated hexadecimal, optionally fixed to 1 – 4 digits (\$ prefix)         | 1     |
| ISHEX{1..4}          | Signed, indicated hexadecimal, optionally fixed to 1 – 4 digits (\$ prefix) | 1,2   |
| BIN{1..16}           | Binary, optionally fixed to 1 – 16 digits                                   | 1     |
| SBIN{1..16}          | Signed binary, optionally fixed to 1 – 16 digits                            | 1,2   |
| IBIN{1..16}          | Indicated binary, optionally fixed to 1 – 16 digits (% prefix)              | 1     |
| ISBIN{1..16}         | Signed, indicated binary, optionally fixed to 1 – 16 digits (% prefix)      | 1,2   |

1 Fixed-digit formatters like DEC4 will pad the number with leading 0s if necessary; ex: DEC4 65 sends 0065. If a number is larger than the specified number of digits, the leading digits will be dropped; ex: DEC4 56422 sends 6422.

2 Signed modifiers work under two's complement rules.

3 The HEX modifier can be used for BCD to Decimal Conversion. See "Hex to BCD Conversion" on page 97.

DISPLAYING "INDICATED" NUMBERS.

As seen in Table 5.12, special versions of the DEC, HEX and BIN formatters allow for the display of indicated, signed and fixed-width numbers. The term "indicated" simply means that a special symbol is displayed, before the number, indicating what number system it belongs to. For example,

```
x VAR Byte
```

```
x = 65
```

```
DEBUG HEX x
```

```
' Show hexadecimal value of x
```

displays "41" (65, in decimal, is 41, in hexadecimal). You might see a problem here... unless you knew the number was supposed to be

## DEBUG – BASIC Stamp Command Reference

---

hexadecimal, you might think it was 41, in decimal... a totally different number. To help avoid this, use the IHEX formatter (the "I" stands for indicated). Changing the DEBUG line to read: `DEBUG IHEX x` would print "\$41" on the screen. A similar formatter for binary also exists, IBIN, which prints a "%" before the number.

Signed numbers are preceded with a space ( ) or a minus sign (-) to indicate a positive or negative number, respectively. Normally, any number displayed by the BASIC Stamp is shown in its unsigned (positive) form without any indicator. The signed formatters allow you to display the number as a signed (rather than unsigned) value. **NOTE: Only Word-sized variables can be used for signed number display.** The code below demonstrates the difference in all three numbering schemes.

DISPLAYING SIGNED VS. UNSIGNED NUMBERS.

```
x VAR Word

x = -65
DEBUG "Signed: ", SDEC x, " ", ISHEX x, " ", ISBIN x, CR
DEBUG "Unsigned: ", DEC x, " ", IHEX x, " ", IBIN x
```

This code will generate the display shown below:

```
Signed: -65 -$41 -%1000001
Unsigned: 65471 $FFBF %111111110111111
```

The signed form of the number -65 is shown in decimal, hexadecimal and then in binary on the top line. The unsigned form, in all three number systems, is shown on the bottom line. If the unsigned form looks strange to you, it's because negative numbers are stored in twos complement format within the BASIC Stamp.

Suppose that your program contained several DEBUG instructions showing the contents of different variables. You would want some way to tell them apart. One possible way is to do the following:

AUTOMATIC NAMES IN THE DISPLAY.

```
x VAR Byte
y VAR Byte

x = 100
y = 250
DEBUG "X = ", DEC x, CR ' Show decimal value of x
DEBUG "Y = ", DEC y, CR ' Show decimal value of y
```

---

## 5: BASIC Stamp Command Reference – DEBUG

---

but typing the name of the variables in quotes (for the display) can get a little tedious. A special formatter, the question mark (?), can save you a lot of time. The code below does exactly the same thing (with less typing):

```
x VAR Byte
y VAR Byte

x = 100
y = 250
DEBUG DEC ? x ' Show decimal value of x
DEBUG DEC ? y ' Show decimal value of y
```

The display would look something like this:

```
x = 100
y = 250
```

The ? formatter always displays data in the form "symbol = value" (followed by a carriage return). In addition, it defaults to displaying in decimal, so we really only needed to type: `DEBUG ? x` for the above code. You can, of course, use any of the three number systems. For example: `DEBUG HEX ? x` or `DEBUG BIN ? y`.

It's important to note that the "symbol" it displays is taken directly from what appears to the right of the ?. If you were to use an expression, for example: `DEBUG ? x*10/2+3` in the above code, the display would show: "x\*10/2+3 = 503".

A special formatter, ASC, is also available for use only with the ? formatter to display ASCII characters, as in: `DEBUG ASC ? x`.

### DISPLAYING FIXED-WIDTH NUMBERS.

What if you need to display a table of data; multiple rows and columns? The Signed/Unsigned code (above) approaches this but, if you notice, the columns don't line up. The number formatters (DEC, HEX and BIN) have some useful variations to make the display fixed-width (see Table 5.12). Up to 5 digits can be displayed for decimal numbers. To fix the value to a specific number of decimal digits, you can use DEC1, DEC2, DEC3, DEC4 or DEC5. For example:

```
x VAR Byte

x = 165
DEBUG DEC5 x ' Show decimal value of x in 5 digits
```

## DEBUG – BASIC Stamp Command Reference

---

displays "00165". Notice that leading zeros? The display is "fixed" to 5 digits, no more and no less. Any unused digits will be filled with zeros.

Using DEC4 in the same code would display "0165". DEC3 would display "165". What would happen if we used DEC2? Regardless of the number, the BASIC Stamp will ensure that it is always the exact number of digits you specified. In this case, it would truncate the "1" and only display "65".

Using the fixed-width version of the formatters in the Signed/Unsigned code above, may result in the following code:

```
x VAR Word

x = -65
DEBUG "Signed: ", SDEC5 x, " ", ISHEX4 x, " ", IBIN16 x, CR
DEBUG "Unsigned: ", DEC5 x, " ", IHEX4 x, " ", IBIN16 x
```

and displays:

```
Signed: -00065 -$0041 -%0000000001000001
Unsigned: 65471 $FFBF %1111111110111111
```

Note: The columns don't line up exactly (due to the extra "sign" characters in the first row), but it certainly looks better than the alternative.

If you have a string of characters to display (a byte array), you can use the STR formatter to do so. The STR formatter has two forms (as shown in Table 5.11) for variable-width and fixed-width data. The example below is the variable-width form.

DISPLAYING STRINGS (BYTE ARRAYS).

VARIABLE-WIDTH STRINGS.

```
x VAR Byte (5)

x(0) = "A"
x(1) = "B"
x(2) = "C"
x(3) = "D"
x(4) = 0
DEBUG STR x
```

This code displays "ABCD" on the screen. In this form, the STR formatter displays each character contained in the byte array until it finds a character that is equal to 0 (value 0, not "0"). This is convenient for use with the SERIN command's STR formatter, which appends 0's to the end of variable-width character string inputs. NOTE: If your byte array

## 5: BASIC Stamp Command Reference – DEBUG

---

doesn't end with 0, the BASIC Stamp will read and output all RAM register contents until it finds a 0 or until it cycles through all RAM locations.

### FIXED-WIDTH STRINGS.

To specify a fixed-width format for the STR formatter, use the form STR x\n; where x is the byte array and n is the number of characters to print. Changing the DEBUG line in the example above to: DEBUG STR x\2 would display "AB" on the screen.

### REPEATING CHARACTERS.

If you need to display the same ASCII character multiple times, the REP (repeat) formatter can help. REP takes the form: REP x\n; where x is the character and n is the number of times to repeat it. For example:

```
DEBUG REP "-"\10
```

would display 10 hyphens on the screen, "-----".

### SPECIAL CONTROL CHARACTERS.

Since individual DEBUG instructions can grow to be fairly complicated, and since a program can contain many DEBUGs, you'll probably want to control the character positioning of the Debug Terminal screen. DEBUG supports a number of different control characters, some with pre-defined symbols (see Table 5.13).

All of the control characters have pre-defined symbols associated with them. In your DEBUG commands, you can use those symbols, for example: DEBUG "Hello", CR displays "Hello" followed by a carriage return. You can always use the ASCII value for any of the control characters, however. For example: DEBUG "Hello", 13 is exactly the same as the code above.

The Move To (x,y) control character allows positioning to a specific column and row of the display. If the Debug Terminal receives this character, it expects to see an x and y position value to follow (in the next two characters received). The following line moves the cursor to column number 4 in row number 5 and displays "Hello":

```
' {$PBASIC 2.5}
```

```
DEBUG CRSRXY, 4, 5, "Hello"
```

## DEBUG – BASIC Stamp Command Reference

The upper-left cursor position is 0,0 (that is column 0, row 0). The right-most cursor positions depend on the size of the Debug Terminal window (which is user adjustable). If a character position that is out of range is received, the Debug Terminal wraps back around to the opposite side of the screen.

The Move To Column (x) and Move To Row (y) control characters work similarly to Move To (x,y) except they only expect a single position value to follow.

The Clear Right (CLREOL) control character clears the characters that appear to the right of, and on, the cursor's current position. The cursor is not moved by this action.

The Clear Down (CRLDN) control character clears the characters that appear below, and on, the cursor's current line. The cursor is not moved by this action.

| Name             | Symbol                | ASCII Value | Description                                                                                                     |
|------------------|-----------------------|-------------|-----------------------------------------------------------------------------------------------------------------|
| Clear Screen     | CLS                   | 0           | Clear the screen and place cursor at home position.                                                             |
| Home             | HOME                  | 1           | Place cursor at home in upper-left corner of the screen.                                                        |
| Move To (x,y)    | CRSRXY <sup>2.5</sup> | 2           | Move cursor to specified location. Must be followed by two values (x and then y)                                |
| Cursor Left      | CRSRLF <sup>2.5</sup> | 3           | Move cursor one character to left.                                                                              |
| Cursor Right     | CRSRRT <sup>2.5</sup> | 4           | Move cursor one character to right.                                                                             |
| Cursor Up        | CRSRUP <sup>2.5</sup> | 5           | Move cursor one character up.                                                                                   |
| Cursor Down      | CRSRDN <sup>2.5</sup> | 6           | Move cursor one character down.                                                                                 |
| Bell             | BELL                  | 7           | Beep the PC speaker.                                                                                            |
| Backspace        | BKSP                  | 8           | Back up cursor to left one space.                                                                               |
| Tab              | TAB                   | 9           | Tab to the next column.                                                                                         |
| Line Feed        | LF <sup>2.5</sup>     | 10          | Move cursor down one line.                                                                                      |
| Clear Right      | CLREOL <sup>2.5</sup> | 11          | Clear line contents to the right of cursor.                                                                     |
| Clear Down       | CLRDN <sup>2.5</sup>  | 12          | Clear screen contents below cursor.                                                                             |
| Carriage Return  | CR                    | 13          | Move cursor to the first column of the next line (shift any data on the right down to that line as well).       |
| Move To Column X | CRSRX <sup>2.5</sup>  | 14          | Move cursor to specified column. Must be followed by byte value (x) for the column (0 is the left-most column). |
| Move To Row Y    | CRSRY <sup>2.5</sup>  | 15          | Move cursor to specified row. Must be followed by byte value (y) for the row (0 is the top-most row).           |

**Table 5.13:** Special DEBUG Control Characters for all BS2 models.

NOTE: ( <sup>2.5</sup> ) indicates this control character requires the PBASIC 2.5 compiler directive.

## 5: BASIC Stamp Command Reference – DEBUG

---

### TECHNICAL BACKGROUND



On all the BS2 models, DEBUG is actually a special case of the SEROUT instruction. It is set for inverted (RS-232-compatible) serial output through the programming connector (the SOUT pin) at 9600 baud, no parity, 8 data bits, and 1 stop bit. For example,

```
DEBUG "Hello"
```

is exactly like:



```
' {$STAMP BS2}
```

```
SEROUT 16, $4054, ["Hello"]
```

in terms of function on a BS2. The DEBUG line actually takes less program space, and is obviously easier to type.

Another method to decrease program space is to reduce the number of DEBUG instructions by spreading DEBUG data across multiple lines. To do this, each line that wraps around must end with a comma as in the example below:

```
' {$PBASIC 2.5}
```

```
DEBUG "This is line 1", CR,
 "This is line 2"
```

The example above works identically to, but uses less program space than this version:

```
DEBUG "This is line 1", CR
DEBUG "This is line 2"
```

Note that spreading a DEBUG statement across multiple lines requires the declaration of PBASIC 2.5 syntax.

You may view DEBUG's output using a terminal program set to the above parameters, but you may have to modify either your development board or the serial cable to temporarily disconnect pin 3 of the BASIC Stamp (pin 4 of the DB-9 connector). See the SEROUT command for more detail.

A demo program for all BS2 models that uses DEBUG and DEBUGIN commands can be found at the end of the DEBUGIN section, next.

## ***DEBUG – BASIC Stamp Command Reference***

---



## 5: BASIC Stamp Command Reference – DEBUGIN

### DEBUGIN

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**DEBUGIN** *InputData* { , *InputData* }

NOTE: DEBUGIN requires the \$PBASIC 2.5 compiler directive.

### Function

Accept information from the user via the Debug Terminal within the BASIC Stamp Editor program.

- **InputData** is list of variables and formatters that tells DEBUGIN what to do with incoming data. DEBUGIN can store data in a variable or array, interpret numeric text (decimal, binary, or hex) and store the corresponding value in a variable, wait for a fixed or variable sequence of bytes, or ignore a specified number of bytes. These actions can be combined in any order in the *InputData* list.

### Quick Facts

Table 5.14: DEBUGIN Quick Facts.

|                  | BS2, BS2e, BS2sx, BS2p, BS2pe                                 | BS2px                                                          |
|------------------|---------------------------------------------------------------|----------------------------------------------------------------|
| Serial Protocol  | Asynchronous 9600 baud N, 8, 1<br>Inverted Polarity, Raw Data | Asynchronous 19200 baud N, 8, 1<br>Inverted Polarity, Raw Data |
| Related Commands | SERIN and DEBUG                                               |                                                                |

### Explanation

DEBUGIN provides a convenient way for your BASIC Stamp to accept input from the user via the Debug Terminal. DEBUGIN can wait for, filter and convert incoming data in powerful ways, using the same techniques and modifiers as SERIN.

DEBUGIN is actually a special case of the SERIN instruction. It is set for inverted (RS-232-compatible) serial input through the programming connector (the SIN pin) at 9600 baud (19200 baud on BS2px), no parity, 8 data bits, and 1 stop bit.

For example:

```
DEBUGIN DEC1 myNum
```

## DEBUGIN – BASIC Stamp Command Reference

is exactly like:

```
' {$STAMP BS2}
```



```
SERIN 16, $4054, [DEC1 myNum]
```

in terms of function on a BS2. The DEBUGIN line actually takes less program space, and is obviously easier to type. Example:

```
' {$PBASIC 2.5}
```

```
myNum VAR Nib
```

```
DEBUG CLS, "Enter a number (1 - 5)? --> "
DEBUGIN DEC1 myNum
IF ((myNum >= 1) AND (myNum <= 5)) THEN
 DEBUG CLS, "You entered: ", DEC1 myNum
ELSE
 DEBUG CLS, "Sorry, number out of range"
ENDIF
END
```

The tables below list all the special formatters and conversion formatters available to the DEBUGIN command. See the SERIN instruction for additional information and examples of their use.

| Special Formatter            | Action                                                                                                                                                                                                                              |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| STR <i>ByteArray</i> \L {E}  | Input a character string of length L into an array. If specified, an end character E causes the string input to end before reaching length L. Remaining bytes are filled with 0s (zeros).                                           |
| WAIT ( <i>Value</i> )        | Wait for a sequence of bytes specified by value. Value can be numbers separated by commas or quoted text (ex: 65, 66, 67 or "ABC"). The WAIT formatter is limited to a maximum of six characters.                                   |
| WAITSTR <i>ByteArray</i> {L} | Wait for a sequence of bytes matching a string stored in an array variable, optionally limited to L characters. If the optional L argument is left off, the end of the array-string must be marked by a byte containing a zero (0). |
| SKIP <i>Length</i>           | Ignore <i>Length</i> bytes of characters.                                                                                                                                                                                           |

**Table 5.15:** DEBUGIN Special Formatters.

There is an additional special formatter for the BS2p, BS2pe, and BS2px:



| Special Formatter | Action                                                                                                                                   |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| SPSTR L           | Input a character string of length L bytes (up to 126) into Scratch Pad RAM, starting at location 0. Use GET to retrieve the characters. |

**Table 5.16:** DEBUGIN Additional Special Formatter for the BS2p, BS2pe, and BS2px.

## 5: BASIC Stamp Command Reference – DEBUGIN

**Table 5.17:** DEBUGIN Conversion Formatters.

| Conversion Formatter | Type of Number                                                                                         | Numeric Characters Accepted        | Notes   |
|----------------------|--------------------------------------------------------------------------------------------------------|------------------------------------|---------|
| DEC{1..5}            | Decimal, optionally limited to 1 – 5 digits                                                            | 0 through 9                        | 1       |
| SDEC{1..5}           | Signed decimal, optionally limited to 1 – 5 digits                                                     | -, 0 through 9                     | 1,2     |
| HEX{1..4}            | Hexadecimal, optionally limited to 1 – 4 digits                                                        | 0 through 9, A through F           | 1,3,5   |
| SHEX{1..4}           | Signed hexadecimal, optionally limited to 1 – 4 digits                                                 | -, 0 through 9, A through F        | 1,2,3   |
| IHEX{1..4}           | Indicated hexadecimal, optionally limited to 1 – 4 digits                                              | \$, 0 through 9, A through F       | 1,3,4   |
| ISHEX{1..4}          | Signed, indicated hexadecimal, optionally limited to 1 – 4 digits                                      | -, \$, 0 through 9, A through F    | 1,2,3,4 |
| BIN{1..16}           | Binary, optionally limited to 1 – 16 digits                                                            | 0, 1                               | 1       |
| SBIN{1..16}          | Signed binary, optionally limited to 1 – 16 digits                                                     | -, 0, 1                            | 1,2     |
| IBIN{1..16}          | Indicated binary, optionally limited to 1 – 16 digits                                                  | %, 0, 1                            | 1,4     |
| ISBIN{1..16}         | Signed, indicated binary, optionally limited to 1 – 16 digits                                          | -, %, 0, 1                         | 1,2,4   |
| NUM                  | Generic numeric input (decimal, hexadecimal or binary); hexadecimal or binary number must be indicated | \$, %, 0 through 9, A through F    | 1, 3, 4 |
| SNUM                 | Similar to NUM with value treated as signed with range -32768 to +32767                                | -, \$, %, 0 through 9, A through F | 1,2,3,4 |

- 1 All numeric conversions will continue to accept new data until receiving either the specified number of digits (ex: three digits for DEC3) or a non-numeric character.
- 2 To be recognized as part of a number, the minus sign (-) must immediately precede a numeric character. The minus sign character occurring in non-numeric text is ignored and any character (including a space) between a minus and a number causes the minus to be ignored.
- 3 The hexadecimal formatters are not case-sensitive; “a” through “f” means the same as “A” through “F”.
- 4 Indicated hexadecimal and binary formatters ignore all characters, even valid numerics, until they receive the appropriate prefix (\$ for hexadecimal, % for binary). The indicated formatters can differentiate between text and hexadecimal (ex: ABC would be interpreted by HEX as a number but IHEX would ignore it unless expressed as \$ABC). Likewise, the binary version can distinguish the decimal number 10 from the binary number %10. A prefix occurring in non-numeric text is ignored, and any character (including a space) between a prefix and a number causes the prefix to be ignored. Indicated, signed formatters require that the minus sign come before the prefix, as in -\$1B45.
- 5 The HEX modifier can be used for Decimal to BCD Conversion. See “Hex to BCD Conversion” on page 97.

For examples of all conversion formatters and how they process incoming data, see Appendix C.

## ***DEBUGIN – BASIC Stamp Command Reference***

---

### **Demo Program (DEBUGIN.bs2)**

```
' DEBUGIN.bs2
' This program demonstrates the ability to accept user input from the
' Debug Terminal, and to accept numeric entry in any valid format.

' {$STAMP BS2}
' {$PBASIC 2.5}

myNum VAR Word

Main:
DO
 DEBUG CLS, "Enter any number: " ' prompt user
 DEBUGIN SNUM myNum ' retrieve number in any format

 DEBUG CLS, ' display number in all formats
 SDEC ? myNum,
 SHEX ? myNum,
 SBIN ? myNum
 PAUSE 3000
LOOP ' do it again
END
```



NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## 5: BASIC Stamp Command Reference – DO...LOOP

NOTE: DO...LOOP requires the PBASIC 2.5 compiler directive.

All 2

### DO...LOOP

BS1

BS2

BS2e

BS2sx

BS2p

BS2pe

BS2px

**DO { WHILE | UNTIL *Condition(s)* }**

***Statement(s)***

**LOOP { WHILE | UNTIL *Condition(s)* }**

### Function

Create a repeating loop that executes the *Statement(s)*, one or more program lines that form a code block, between DO and LOOP, optionally testing *Condition(s)* before or after the *Statement(s)*.

- ***Condition*** is an optional variable/constant/expression (0 - 65535) which determines whether the loop will run or terminate. *Condition* must follow WHILE or UNTIL.
- ***Statement*** is any valid PBASIC instruction.

### Quick Facts

**Table 5.18:** DO...LOOP Quick Facts.

|                                   | All BS2 Models                                       |
|-----------------------------------|------------------------------------------------------|
| Maximum Nested Loops              | 16                                                   |
| WHILE <i>Condition</i> Evaluation | Run loop if <i>Condition</i> evaluates as true       |
| UNTIL <i>Condition</i> Evaluation | Terminate loop if <i>Condition</i> evaluates as true |
| Related Commands                  | FOR...NEXT and EXIT                                  |

### Explanation

DO...LOOP loops let a program execute a series of instructions indefinitely or until a specified condition terminates the loop. The simplest form is shown here:

```
' {$PBASIC 2.5}

DO
 DEBUG "Error...", CR
 PAUSE 2000
LOOP
```

In this example the error message will be printed on the Debug screen every two seconds until the BASIC Stamp is reset. Simple DO...LOOP loops can be terminated with EXIT.

## ***DO...LOOP – BASIC Stamp Command Reference***

---

For example:

```
' {$PBASIC 2.5}

AckPin PIN 0
Pressed CON 1

DO
 DEBUG "Error...", CR
 IF (AckPin = Pressed) THEN EXIT ' wait for user button press
 PAUSE 2000
LOOP
GOTO Initialize ' re-initialize system
```

In this case the DO...LOOP will continue until the pin called *AckPin* is equal to *Pressed* (1), and then the loop will terminate and continue at the line GOTO Initialize.

More often than not, you will want to test some condition to determine whether the code block should run or continue to run. A loop that tests the condition before running code block is constructed like this:

```
' {$PBASIC 2.5}

reps VAR Nib

DO WHILE (reps < 3) ' test before loop statements
 DEBUG "*"
 reps = reps + 1
LOOP
```

In this program the instructions DEBUG "\*" and reps = reps + 1 will not run unless the WHILE condition evaluates as True. Another way to write the loop is like this:

```
' {$PBASIC 2.5}

reps VAR Nib

DO
 DEBUG "*"
 reps = reps + 1
LOOP UNTIL (reps >= 3) ' test after loop statements
```

The difference is that with this loop, the code block will always be run at least once before the condition is tested and will continue to run as long as the UNTIL condition evaluates as False.

## 5: BASIC Stamp Command Reference – DO...LOOP

Note that the WHILE test (loop runs WHILE *Condition* is True) and UNTIL test (loop runs UNTIL *Condition* is True) can be interchanged, but they are generally used as illustrated above.



NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

### Demo Program (DO-LOOP.bs2)

```
' DO-LOOP.bs2
' This program creates a little guessing game. It starts by creating
' a (pseudo) random number between 1 and 10. The inner loop will run
' until the answer is guessed or 10 tries have been attempted. The
' outer loop has no condition and will cause the inner loop code to
' run until the BASIC Stamp is reprogrammed.

' {$STAMP BS2}
' {$PBASIC 2.5}

rVal VAR Word ' random value
answer VAR Byte ' game answer
guess VAR Byte ' player guess
tries VAR Nib ' number of tries

Main:
DO
 RANDOM rVal
 answer = rVal.LOWBYTE */ 10 + 1 ' create 1 - 10 answer
 tries = 0

 DO ' get answer until out of tries
 DEBUG CLS,
 "Guess a number (1 - 10): "
 DEBUGIN DEC guess ' get new guess
 tries = tries + 1 ' update tries count
 LOOP UNTIL ((tries = 10) OR (guess = answer))

 IF (guess = answer) THEN ' test reason for loop end
 DEBUG CR, "You got it!"
 ELSE
 DEBUG CR, "Sorry ... the answer was ", DEC answer, "."
 ENDIF
 PAUSE 1000
LOOP ' run again
END
```

## ***DO...LOOP – BASIC Stamp Command Reference***

---



## 5: BASIC Stamp Command Reference – DTMFOUT

### DTMFOUT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**DTMFOUT** *Pin*, { *OnTime*, *OffTime*, } [ *Tone*{, *Tone*...} ]

#### Function

Generate dual-tone, multifrequency tones (DTMF, i.e., telephone “touch” tones).

- ***Pin*** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This pin will be set to output mode during generation of tones and set to input mode afterwards.
- ***OnTime*** is an optional variable/constant/expression (0 – 65535) specifying a duration of the tone. The unit of time and the default time for *OnTime* is described in Table 5.19.
- ***OffTime*** is an optional variable/constant/expression (0 – 65535) specifying the length of silent pause after a tone (or between tones, if multiple tones are specified). The unit of time and the default time for *OffTime* is described in Table 5.19.
- ***Tone*** is a variable/constant/expression (0 – 15) specifying the DTMF tone to generate. Tones 0 through 11 correspond to the standard layout of the telephone keypad, while 12 through 15 are the fourth-column tones used by phone test equipment and in ham-radio applications.

#### Quick Facts

Table 5.19: DMTFOUT Quick Facts.

|                                | BS2, BS2e | BS2sx  | BS2p     | BS2pe  | BS2px    |
|--------------------------------|-----------|--------|----------|--------|----------|
| <b>Default <i>OnTime</i></b>   | 200 ms    | 80 ms  | 55 ms    | 196 ms | 34 ms    |
| <b>Default <i>OffTime</i></b>  | 50 ms     | 50 ms  | 50 ms    | 50 ms  | 50 ms    |
| <b>Units in <i>OnTime</i></b>  | 1 ms      | 0.4 ms | 0.265 ms | 1 ms   | 0.166 ms |
| <b>Units in <i>OffTime</i></b> | 1 ms      | 1 ms   | 1 ms     | 1 ms   | 1 ms     |
| <b>Related Command</b>         | FREQOUT   |        |          |        |          |

#### Explanation

DTMF tones are used to dial the phone or remotely control certain radio equipment. The BASIC Stamp can generate these tones digitally using the DTMFOUT instruction. Figure 5.4 shows how to connect a speaker or audio amplifier to hear these tones and Figure 5.5 shows how to connect the BASIC Stamp to the phone line.

## DTMFOUT – BASIC Stamp Command Reference

The following DTMFOUT instruction will generate DTMF tones on I/O pin 10:

```
DTMFOUT 10, [6, 2, 4, 8, 3, 3, 3] ' Call Parallax
```

If the BASIC Stamp is connected to the phone line properly, the above command would be equivalent to dialing 624-8333 from a phone keypad. If you wanted to slow the pace of the dialing to accommodate a noisy phone line or radio link, you could use the optional *OnTime* and *OffTime* values:

```
DTMFOUT 10, 500, 100, [6, 2, 4, 8, 3, 3, 3] ' Call Parallax, slowly
```

In this example, on a BS2 the *OnTime* is set to 500 ms (1/2 second) and *OffTime* to 100 ms (1/10th second).

| Tone Value | Corresponding Telephone Key     |
|------------|---------------------------------|
| 0 – 9      | Digits 0 through 9              |
| 10         | Star (*)                        |
| 11         | Pound (#)                       |
| 12 – 15    | Fourth column tones A through D |

**Table 5.20:** DTMF Tones and Corresponding Telephone Keys.

### Driving an Audio Amplifier

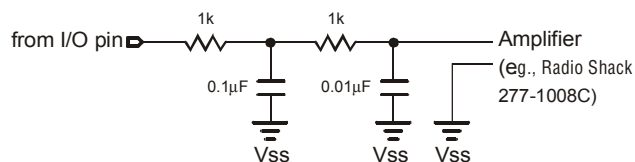
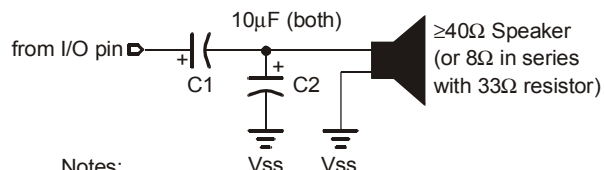


**Figure 5.4:** Example RC Filter Circuits for Driving an Audio Amplifier (top) or a Speaker (bottom).

### Driving a Speaker



Notes:  
C1 may be omitted for piezo speakers  
C2 is optional, but reduces high-frequency noise

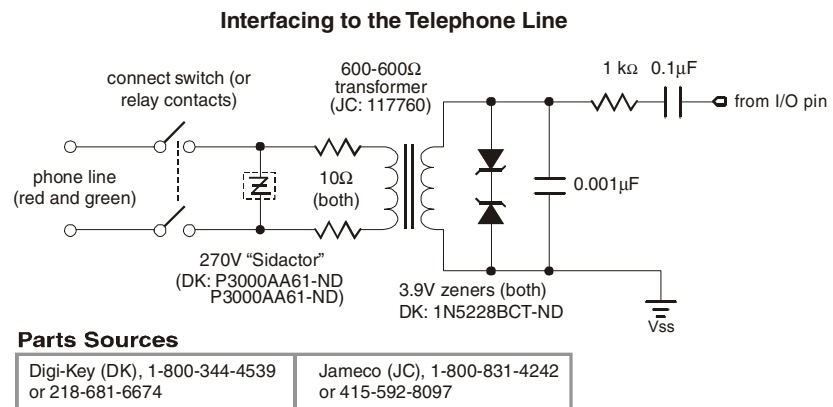
## 5: BASIC Stamp Command Reference – DTMFOUT

### TECHNICAL BACKGROUND.

The BASIC Stamp microcontroller is a purely digital device. DTMF tones are analog waveforms, consisting of a mixture of two sine waves at different audio frequencies. So how does a digital device generate analog output? The BASIC Stamp creates and mixes the sine waves mathematically, then uses the resulting stream of numbers to control the duty cycle of a very fast pulse-width modulation (PWM) routine. So what's actually coming out of the I/O pin is a rapid stream of pulses. The purpose of the filtering arrangements shown in Figure 5.4 and Figure 5.5 is to smooth out the high-frequency PWM, leaving only the lower frequency audio behind.

Keep this in mind if you want to interface BASIC Stamp's DTMF output to radios and other equipment that could be adversely affected by the presence of high-frequency noise on the input. Make sure to filter the DTMF output thoroughly. The circuits in Figure 5.4 are only a starting point; you may want to use an active low-pass filter with a roll-off point around 2 kHz.

**Figure 5.5:** Example DAA Circuit to Interface to a Standard Telephone Line.



### Demo Program (DTMFOUT.bs2)

NOTE: This example program can be used with all BS2 models. This program uses conditional compilation techniques; see Chapter 3 for more information.

```
' DTMFOUT.bs2
' This demo program is a rudimentary memory dialer. Since DTMF digits fit
' within a nibble (four bits), the program below packs two DTMF digits into
' each byte of three EEPROM data tables. The end of phone number is marked
' by the nibble $F, since this is not a valid phone-dialing digit.
' Conditional compilation sets the timing adjustment factor so that the
' output will sound the same on any BS2 model.
```

## DTMFOUT – BASIC Stamp Command Reference

---

```
' {$STAMP BS2}
' {$PBASIC 2.5}

Spkr PIN 10 ' DTMF output on pin 10

#SELECT $STAMP
#CASE BS2, BS2E, BS2PE
 TmAdj CON $100 ' x 1.0 (time adjust)
#CASE BS2SX
 TmAdj CON $280 ' x 2.5
#CASE BS2P
 TmAdj CON $3C5 ' x 3.77
#CASE BS2PX
 TmAdj CON $607 ' x 6.03
#ENDSELECT

eeLoc VAR Byte ' EEPROM address of stored number
eeByte VAR Byte ' Byte containing two DTMF digits
dtDig VAR eeByte.NIB1 ' Digit to dial
phone VAR Nib ' Pick a phone #
hiLo VAR Bit ' Bit to select upper and lower nib

Parallax DATA $19,$16,$62,$48,$33,$3F ' Phone: 1-916-624-8333
ParallaxFax DATA $19,$16,$62,$48,$00,$3F ' Phone: 1-916-624-8003
Information DATA $15,$20,$55,$51,$21,$2F ' Phone: 1-520-555-1212

Main:
 FOR phone = 0 TO 2
 ' retrieve address
 LOOKUP phone, [Parallax, ParallaxFax, Information], eeLoc
 GOSUB Dial_Number
 PAUSE 2000
 NEXT
END

Dial_Number:
 DO
 READ eeLoc, eeByte ' Retrieve byte from EEPROM
 eeLoc = eeLoc + 1 ' point to next pair of digits
 FOR hiLo = 0 TO 1 ' Dial upper and lower digits
 IF (dtDig = $F) THEN EXIT ' Hex $F is end-of-number flag
 DTMFOUT Spkr, ' dial digit
 150 */ TmAdj, 25, [dtDig] ' 150 ms on, 25 ms off
 eeByte = eeByte << 4 ' Shift in next digit
 NEXT
 LOOP UNTIL (dtDig = $F)
 RETURN
```

## 5: BASIC Stamp Command Reference – EEPROM

### EEPROM

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**EEPROM { Location, } ( Dataltem {, Dataltem...} )**



(See DATA)

### Function

Write data to the EEPROM during program download.

- **Location** is an optional variable/constant (0 – 255) that specifies the starting location in the EEPROM at which data should be stored. If no location is given, data is written starting at the next available location.
- **Dataltem** is a constant (0 – 255) to be stored in EEPROM.

### Quick Facts

**Table 5.21:** EEPROM Quick Facts.

|                         | BS1                                                                               |
|-------------------------|-----------------------------------------------------------------------------------|
| <b>Special Notes</b>    | Writes values to EEPROM during download.<br>Can be used to decrease program size. |
| <b>Related Commands</b> | READ and WRITE                                                                    |

### Explanation

When you download a program into the BASIC Stamp 1, it is stored in the EEPROM starting at the highest address (255) and working towards the lowest address. Most programs don't use the entire EEPROM, so the lower portion is available for other uses. The EEPROM directive allows you to define a set of data to store in the available EEPROM locations. It is called a "directive" rather than a "command" because it performs an activity at compile-time rather than at run-time (i.e.: the EEPROM directive is not downloaded to the BASIC Stamp 1, but the data it contains is downloaded).

WRITING SIMPLE, SEQUENTIAL DATA.

The simplest form of the EEPROM directive is something like the following:

```
EEPROM (100, 200, 52, 45)
```

This example, when downloaded, will cause the values 100, 200, 52 and 45 to be written to EEPROM locations 0, 1, 2 and 3, respectively. You can then use the READ and WRITE commands in your code to access these locations and the data you've stored there.

## EEPROM – BASIC Stamp Command Reference

The EEPROM directive uses a counter, called a pointer, to keep track of available EEPROM addresses. The value of the pointer is initially 0. When a program is downloaded, the EEPROM directive stores the first byte value at the current pointer address, then increments (adds 1 to) the pointer. If the program contains more than one EEPROM directive, subsequent EEPROM directives start with the pointer value left by the previous EEPROM directive. For example, if the program contains:

```
EEPROM (72, 69, 76, 76, 79)
EEPROM (104, 101, 108, 108, 111)
```

The first EEPROM directive will start at location 0 and increment the pointer for each data value it stores (1, 2, 3, 4 and 5). The second EEPROM directive will start with the pointer value of 5 and work upward from there. As a result, the first 10 bytes of EEPROM will look like the following:

|          | EEPROM Location (address) |    |    |    |    |     |     |     |     |     |
|----------|---------------------------|----|----|----|----|-----|-----|-----|-----|-----|
|          | 0                         | 1  | 2  | 3  | 4  | 5   | 6   | 7   | 8   | 9   |
| Contents | 72                        | 69 | 76 | 76 | 79 | 104 | 101 | 108 | 108 | 111 |

**Table 5.22:** Example EEPROM Storage.

What if you don't want to store values starting at location 0? Fortunately, the EEPROM directive has an option to specify the next location to use. You can specify the next location number (to set the pointer to) by using the optional *Location* argument before the list of *DataItems*. The following code writes the same data in Table 5.22 to locations 50 through 59:

```
EEPROM 50, (72, 69, 76, 76, 79, 104, 101, 108, 108, 111)
```

WRITING DATA TO OTHER LOCATIONS.

In this example, the *Location* argument is given and tells the EEPROM directive to store the following *DataItem(s)* starting at location 50. The *DataItems* in the list are stored in their respective locations (50, 51, 52... 59).

It is important to realize that the entire BASIC Stamp 1 EEPROM is overwritten during programming. Any EEPROM location not containing a PBASIC program or *DataItems* from an EEPROM directive is written with a 0.

**IMPORTANT CONCEPT:** HOW DATA AND PROGRAMS ARE DOWNLOADED INTO EEPROM.

A common use for EEPROM is to store strings; sequences of bytes representing text. PBASIC converts quoted text like "A" into the corresponding ASCII character code (65 in this case). To make data entry easier, you can place quotes around a whole chunk of text used in an

WRITING TEXT STRINGS.

## 5: BASIC Stamp Command Reference – EEPROM

---

EEPROM directive, and PBASIC will understand it to mean a series of bytes (see the last line of code below). The following three EEPROM directives are equivalent:

```
EEPROM (72, 69, 76, 76, 79)
EEPROM ("H", "E", "L", "L", "O")
EEPROM ("HELLO")
```

All three lines of code, above, will result in the numbers 72, 69, 76, 76, and 79 being stored into EEPROM upon downloading. These numbers are simply the ASCII character codes for "H", "E", "L", "L", and "O", respectively. See the demo program, below, for an example of storing and reading multiple text strings.

### WRITING WORD VALUES VS. BYTE VALUES.

The EEPROM is organized as a sequential set of byte-sized memory locations. The EEPROM directive only stores bytes into EEPROM. If you try to store a word-sized value, for example: EEPROM (1125), only the lower byte of the value will be stored (in this case, 101). This does not mean that you can't store word-sized values, however. A word consists of two bytes, called a low-byte and a high-byte. If you wanted to store the value 1125 using the EEPROM directive you'll have to calculate the low-byte and the high-byte and insert them in the list in the proper order, as in:

```
EEPROM (101, 4)
```

The directive above will store the two bytes into two sequential EEPROM locations (the low-byte first, followed by the high-byte). We calculated this in the following manner: 1) high-byte is INT(value / 256) and 2) low-byte is value – (high-byte \* 256).

To retrieve a word-sized value, you'll need to use two READ commands and a word-sized variable. For example,

```
SYMBOL result = W0 ' word-sized variable
SYMBOL resultLo = B0 ' B0 is the low-byte of W0
SYMBOL resultHi = B1 ' B1 is the high-byte of W0

EEPROM (101, 4)

READ 0, resultLo
READ 1, resultHi
DEBUG #result
```

This code would write the low-byte and high-byte of the number 1125 into locations 0 and 1 during download. When the program runs, the two

## EEPROM – BASIC Stamp Command Reference

---

READ commands will read the low-byte and high-byte out of EEPROM (reconstructing it in a word-sized variable) and then display the value on the screen. See the READ and WRITE commands for more information.

### Demo Program (EEPROM.bs1)



```
' EEPROM.bs1
' This program stores a couple of text strings into EEPROM with the EEPROM
' directive and then sends them, one character at a time via the SEROUT
' command. This is a good demonstration of how to save program space by
' storing large amounts of data in EEPROM directly, rather than embedding
' the data into SEROUT commands.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL SOut = 0 ' serial output

SYMBOL idx = B2 ' Holds current location number
SYMBOL phrase = B3
SYMBOL char = B4 ' Holds current character to print

Phrases:
 EEPROM ("Here is a long message that needs to be transmitted.", 13, 0)
 EEPROM ("Here is some more text to be transmitted.", 13, 0)

Main:
 idx = 0
 FOR phrase = 1 TO 2 ' select phrase
 GOSUB Print_It ' print the phrase
 PAUSE 3000 ' Pause for 3 seconds
 NEXT
 END

Print_It:
 READ idx, char ' get next character
 idx = idx + 1 ' point to next EEPROM location
 IF char = 0 THEN Print_Done ' if 0, we're done with this block
 SEROUT SOut, N2400, (char) ' otherwise, transmit it
 'DEBUG #@char ' -- for demo with DEBUG (slower)
 GOTO Print_It

Print_Done:
 RETURN ' return to caller
```



## 5: BASIC Stamp Command Reference – END

|                                                                                   |            |            |            |             |              |             |              |              |
|-----------------------------------------------------------------------------------|------------|------------|------------|-------------|--------------|-------------|--------------|--------------|
|                                                                                   | <b>END</b> | <b>BS1</b> | <b>BS2</b> | <b>BS2e</b> | <b>BS2sx</b> | <b>BS2p</b> | <b>BS2pe</b> | <b>BS2px</b> |
|  | <b>END</b> |            |            |             |              |             |              |              |

### Function

End the program, placing the BASIC Stamp into low-power mode indefinitely. This is equivalent to having a program that does not loop continuously; once the BASIC Stamp reaches the end of the PBASIC program, it enters low-power mode indefinitely. The END command is optional.

### Quick Facts

**Table 5.23:** END Quick Facts.

Note: Current measurements are based on 5-volt power, no extra loads, and 75°F ambient temperature.

|                                  | BS1           | BS2                 | BS2e        | BS2sx       | BS2p                          | BS2pe      | BS2px       |
|----------------------------------|---------------|---------------------|-------------|-------------|-------------------------------|------------|-------------|
| <b>Current Draw During Run</b>   | 1 mA          | 3 mA                | 25 mA       | 60 mA       | 40 mA                         | 15 mA      | 55 mA       |
| <b>Current Draw During Sleep</b> | 25 $\mu$ A    | 50 $\mu$ A          | 200 $\mu$ A | 500 $\mu$ A | 350 $\mu$ A                   | 36 $\mu$ A | 450 $\mu$ A |
| <b>Related Commands</b>          | NAP and SLEEP | NAP, SLEEP and STOP |             |             | NAP, SLEEP, STOP and POLLWAIT |            |             |

### Explanation

END puts the BASIC Stamp into its inactive, low-power mode. In this mode the Stamp's current draw (excluding loads driven by the I/O pins) is reduced to the amount shown in Table 5.23. END keeps the BASIC Stamp inactive until the reset line is activated, the power is cycled off and back on or the PC downloads another program.

Just as with the SLEEP command, pins will retain their input or output settings after the BASIC Stamp is deactivated by END. For example, if the BASIC Stamp is powering an LED when END executes, the LED will stay lit after END, but every 2.3 seconds, there will be a visible wink of the LED as the output pin switches to the input direction for 18 ms (60  $\mu$ s on the BS2pe). (See the SLEEP command for more information).

## ***END – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – EXIT

### EXIT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



### EXIT

NOTE: EXIT requires the PBASIC 2.5 compiler directive.

### Function

Causes the immediate termination of a loop construct (DO...LOOP, FOR...NEXT).

### Quick Facts

Table 5.24: EXIT Quick Facts.

|                        | All BS2 Models           |
|------------------------|--------------------------|
| Maximum EXITs per Loop | 16                       |
| Related Commands       | DO...LOOP and FOR...NEXT |

### Explanation

The EXIT instruction allows a program to terminate a loop structure before the loop limit test is executed. While not required, EXIT is usually used as part of an IF...THEN construct to test a secondary condition for terminating a loop, or for testing a termination condition of an unconditional DO...LOOP structure.

For example, the following subroutine will send characters from a DATA statement to a serial port until a 0 byte is encountered in the data:

```
' {$PBASIC 2.5}

DO
 READ eeAddr, char ' get character from DATA statement
 eeAddr = eeAddr + 1 ' update address pointer
 IF (char = 0) THEN EXIT ' if 0, end of string
 DEBUG char ' otherwise, transmit the character
LOOP ' get next character
RETURN
```



### Demo Program (EXIT.bs2)

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

```
' EXIT.bs2
' This program demonstrates the early termination of DO...LOOP and
' FOR...NEXT loop structures. IF...THEN is used to test a condition
' and when true, EXIT will terminate the loop.

' {$STAMP BS2}
' {$PBASIC 2.5}

col VAR Nib
```

## ***EXIT – BASIC Stamp Command Reference***

---

```
row VAR Nib

Setup:
 col = 0

Main:
 DO WHILE (col < 10) ' attempt 10 iterations
 FOR row = 0 TO 15 ' attempt 16 iterations
 IF (row > 9) THEN EXIT ' terminate when row > 9
 DEBUG CR$RXY, (col * 8), row, ' print col/row at location
 DEC col, "/", DEC row, CR
 NEXT
 col = col + 1 ' update column
 IF (col = 3) THEN EXIT ' terminate when col = 3
 LOOP
END
```

## 5: BASIC Stamp Command Reference – FOR...NEXT

**FOR...NEXT**

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|

 **FOR** *Counter* = *StartValue* **TO** *EndValue* { **STEP** {-} *StepValue* }  
*Statement(s)*  
**NEXT** { *Counter* }

 **FOR** *Counter* = *StartValue* **TO** *EndValue* { **STEP** *StepValue* }  
*Statement(s)*  
**NEXT** { *Counter* }

### Function

Create a repeating loop that executes the *Statement(s)*, one or more program lines that form a code block, between FOR and NEXT, incrementing or decrementing *Counter* according to *StepValue* until the value of the *Counter* variable passes the *EndValue*.

- **Counter** is a variable (usually a byte or a word) used as a counter.
- **StartValue** is a variable/constant/expression (0 – 65535) that specifies the initial value of the variable (*Counter*).
- **EndValue** is a variable/constant/expression (0 – 65535) that specifies the end value of the variable (*Counter*). When the value of *Counter* is outside of the range *StartValue* to *EndValue*, the FOR...NEXT loop stops executing and the program goes on to the instruction after NEXT.
- **StepValue** is an optional variable/constant/expression (0 – 65535) by which the *Counter* increases or decreases with each iteration through the FOR...NEXT loop. On the BS1, use a minus sign (-) in front of the *StepValue* to indicate a negative step. On all BS2 models, if *StartValue* is larger than *EndValue*, PBASIC understands *StepValue* to be negative, even though no minus sign is used.
- **Statement** is any valid PBASIC instruction.



NOTE: Expressions are not allowed as arguments on the BS1.



NOTE: Use a minus sign to indicate negative *StepValues* on the BS1.

# FOR...NEXT – BASIC Stamp Command Reference

## Quick Facts

|                               | BS1                                                                           | All BS2 Models                                                                        |
|-------------------------------|-------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| Max. Nested Loops             | 8                                                                             | 16                                                                                    |
| To Decrement Counter Variable | Set <i>StartValue</i> > <i>EndValue</i> and enter negative <i>StepValue</i> * | Set <i>StartValue</i> > <i>EndValue</i>                                               |
| Counter Comparison            | Exit loop if <i>Counter</i> exceeds <i>EndValue</i>                           | Exit loop if <i>Counter</i> outside range set by <i>StartValue</i> to <i>EndValue</i> |
| Related Commands              | None                                                                          | DO...LOOP and EXIT                                                                    |

\*NOTE: For the BS1, direction (increment or decrement) cannot be changed at run-time.

Table 5.25: FOR...NEXT Quick Facts.

## Explanation

FOR...NEXT loops let your program execute a series of instructions for a specified number of repetitions (called iterations). By default, each time through the loop, the *Counter* variable is incremented by 1. It will continue to loop until the value of *Counter* is outside of the range set by *StartValue* and *EndValue*. Also, FOR...NEXT loops always execute at least once. The simplest form is shown here:

```
reps VAR Nib ' counter for the FOR/NEXT loop
FOR reps = 1 TO 3 ' repeat with reps = 1, 2, 3
 DEBUG "*" ' put * on screen for each repetition
NEXT
```

SIMPLEST FORM OF FOR...NEXT.



NOTE: On the BS1, the loop will continue until *Counter* has gone past *EndValue*.



NOTE: Replace the first line with **SYMBOL reps = B0** on the BS1.

In the above code, the FOR command sets *reps* = 1. Then the DEBUG line (within the FOR...NEXT loop) is executed; printing an asterisk (\*) on the screen. When the BASIC Stamp sees the NEXT command, it goes back to the previous FOR command, adds 1 to *reps* and compares the result to the range set by *StartValue* and *EndValue*. If *reps* is still within range, it executes the code in the loop again. Each time the FOR...NEXT loop executes, the value of *reps* is updated (incremented by 1) and the code within the loop (the DEBUG line) is executed; printing another asterisk on the screen. This code will run through the loop three times; setting *reps* to 1, 2 and 3, and printing three asterisks on the screen. After the third loop, again the BASIC Stamp goes back up to the FOR command, adds 1 to *reps* and compares the result (4 in this case) to the range. Since the range is 1 to 3 and the value is 4 (outside the range) the FOR...NEXT loop is done and the BASIC Stamp will jump down to the first line of code following the NEXT command.

PROCESSING A FOR...NEXT LOOP.

You can view the changing values of *reps* by including the *reps* variable in a DEBUG command within the loop:

## 5: BASIC Stamp Command Reference – FOR...NEXT

NOTE: Change the first line as noted above and replace line 3 with  
**DEBUG #Reps, CR**

```
1 reps VAR Nib ' counter for the FOR/NEXT loop
FOR reps = 1 TO 3 ' repeat with reps = 1, 2, 3
 DEBUG DEC reps, CR ' print rep number
NEXT
```

Running this example should display "1" , "2", and "3" on the screen.

DECREMENTING THE COUNTER INSTEAD  
OF INCREMENTING IT.

FOR...NEXT can also be made to decrement (rather than increment) the *Counter* variable. The BS1 does this when you specify a negative *StepValue* (as well as a *StartValue* that is greater than the *EndValue*). All other BASIC Stamp models do this automatically when the *StartValue* is greater than the *EndValue*. Examples of both are shown below:

```
1 SYMBOL reps = B0 ' counter for the FOR/NEXT loop
FOR reps = 3 TO 1 STEP -1 ' repeat with reps = 3, 2, 1
 DEBUG #reps, CR ' print reps number
NEXT
```

-- OR --

```
2 reps VAR Nib ' counter for the FOR/NEXT loop
FOR reps = 3 TO 1 ' repeat with reps = 3, 2, 1
 DEBUG DEC reps, CR ' print reps number
NEXT
```

2 Note that the code for all the BS2 models did not use the optional STEP argument. This is because we wanted to decrement by positive 1 anyway (the default unit) and the BASIC Stamp realizes it needs to decrement because the *StartValue* is greater than the *EndValue*. A negative *StepValue* on any BS2 model would be treated as its positive, twos complement counterpart. For example, -1 in twos complement is 65535. So the following code executes only once:

```
2 reps VAR Nib ' counter for the FOR/NEXT loop
FOR reps = 3 TO 1 STEP -1 ' try to decrement 3 by 65535
 DEBUG DEC reps, CR ' print reps number
NEXT
```

The above code would run through the loop once with *reps* set to 3. The second time around, it would decrement *reps* by 65535 (-1 is 65535 in twos complement) effectively making the number -65532 (4 in twos complement) which is outside the range of the loop.

## FOR...NEXT – BASIC Stamp Command Reference

All the arguments in the FOR...NEXT command can be constants, variables or expressions on all BS2 models. This leads to some interesting uses. For example, if you make the *StartValue* and *EndValue* a variable, and change their values within the loop, you'll change the behavior of the loop itself. Try the following:

```
{' $PBASIC 2.5}
```

```
reps VAR Byte ' counter for the FOR/NEXT loop
startVal VAR Byte
endVal VAR Byte
```

```
startVal = 1 ' initialize startVal to 1
endVal = 3 ' initialize endVal to 3
```

```
FOR reps = startVal TO endVal ' repeat for 1 to 3
 DEBUG DEC reps, CR
 IF (reps = 3) THEN ' if reps =3, swap startVal/endVal
 startVal = 3 ' otherwise continue loop
 endVal = 1
 ENDIF
NEXT
```

Here the loop starts with a range of 1 to 3. First, the DEBUG line prints the value of *reps*. Then the IF...THEN line makes a decision; if *reps* is equal to 3, then swap the order of *startVal* and *endVal*, otherwise continue the loop execution. The next time through the loop (after *startVal* and *endVal* have been swapped), *reps* will be decremented instead of incremented because *startVal* is greater than *endVal*. The result is a display on the screen of the numbers 1, 2, 3, 2, 1.

The following example uses the value of *reps* as the *StepValue*. This creates a display of power's of 2 (1, 2, 4, 8, 16, 32, 64, etc):

```
reps VAR Word ' counter for the loop

FOR reps = 1 TO 256 STEP reps ' each loop add current value of reps
 DEBUG DEC ? reps ' show reps in Debug window
NEXT
```

There is a potential bug that you should be careful to avoid. The BASIC Stamp uses unsigned 16-bit integer math for any math operation it performs, regardless of the size of values or variables. The maximum value the BASIC Stamp can internally calculate is 65535 (the largest 16-bit

USING VARIABLES AS ARGUMENTS.



NOTE: The increment/decrement direction of the FOR...NEXT loop cannot be changed on the BS1.



NOTE: For BS1's, change line 1 to **SYMBOL reps = W0** and line 3 to **DEBUG reps**

WATCH OUT FOR 16-BIT ROLLOVER, OR VARIABLE RANGE, ERRORS.



## 5: BASIC Stamp Command Reference – FOR...NEXT

number). If you add 1 to 65535, you get 0 as the 16-bit register rolls over (like a car's odometer does when you exceed the maximum mileage it can display). Similarly, if you subtract 1 from 0, you'll get 65535 as the 16-bit register rolls under (a rollover in the opposite direction).

If you write a FOR...NEXT loop who's *StepValue* would cause *Counter* to go past 65535, this rollover may cause the loop to execute more times than you expect. Try the following example:



NOTE: For BS1's, change line 1 to  
**SYMBOL reps = W0**  
and line 3 to  
**DEBUG reps**

```
reps VAR Word ' counter for the loop
FOR reps = 0 TO 65535 STEP 3000 ' each loop add 3000
 DEBUG DEC ? reps ' show reps in Debug window
NEXT
```

The value of *reps* increases by 3000 each trip through the loop. As it approaches the *EndValue*, an interesting thing happens; *reps* is: 57000, 60000, 63000, 464, 3464... It passes the *EndValue*, rolls over and keeps going. That's because the result of the calculation 63000 + 3000 exceeds the maximum capacity of a 16-bit number and then rolls over to 464. When the result of 464 is tested against the range ("Is Reps > 0 and is Reps < 65535?") it passes the test and the loop continues.

A similar symptom can be seen in a program who's *EndValue* is mistakenly set higher than what the counter variable can hold. The example below uses a byte-sized variable, but the *EndValue* is set to a number greater than what will fit in a byte:

```
SYMBOL reps = B2 ' counter for the loop
FOR reps = 0 TO 300 ' each loop add 1
 DEBUG reps ' show reps in Debug window
NEXT
```



-- OR --



```
reps VAR Byte ' counter for the loop
FOR reps = 0 TO 300 ' each loop add 1
 DEBUG DEC ? reps ' show reps in Debug window
NEXT
```

Here, *reps* is a byte variable; which can only hold the number range 0 to 255. The *EndValue* is set to 300, however; greater than 255. This code will loop endlessly because when *reps* is 255 and the FOR...NEXT loop adds 1,

## FOR...NEXT – BASIC Stamp Command Reference

---

*reps* becomes 0 (bytes will rollover after 255 just like words will rollover after 65535). The result, 0, is compared against the range (0 – 255) and it is found to be within the range, so the FOR...NEXT loop continues.

It's important to realize that on all the BS2 models, the test is against the entire range, not just the *EndValue*. The code below is a slight modification of the previous example (the *StartValue* is 10 instead of 0) and will not loop endlessly.

NOTE: On the BS1, the loop will continue until *Counter* has gone past *EndValue*. The rollover error will still occur if the BS1 cannot determine if *Counter* went past *EndValue*.

```
reps VAR Byte ' counter for the loop
FOR reps = 10 TO 300 ' each loop add 1
 DEBUG DEC ? reps ' show reps in Debug window
NEXT
```



*reps* still rolls over to 0, as before, however, this time it is outside the range of 10 to 255. The loop stops, leaving *reps* at 0. Note that this code is still in error since *reps* will never reach 300 until it is declared as a Word.

### Demo Program (FOR-NEXT.bs1)



```
' FOR-NEXT.bs1
' This example uses a FOR...NEXT loop to churn out a series of sequential
' squares (numbers 1, 2, 3, 4... raised to the second power) by using a
' variable to set the FOR...NEXT StepValue, and incrementing StepValue
' within the loop. Sir Isaac Newton is generally credited with the
' discovery of this technique.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL square = B2 ' FOR/NEXT counter
SYMBOL stepSize = B3 ' step size increases by 2 each loop

Setup:
 stepSize = 1
 square = 1

Main:
 FOR square = 1 TO 250 STEP stepSize ' show squares up to 250
 DEBUG square ' display on screen
 stepSize = stepSize + 2 ' add 2 to stepSize
 NEXT ' loop until square > 250
END
```

## 5: BASIC Stamp Command Reference – FOR...NEXT



NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

### Demo Program (FOR-NEXT.bs2)

```
' FOR-NEXT.bs2
' This example uses a FOR...NEXT loop to churn out a series of sequential
' squares (numbers 1, 2, 3, 4... raised to the second power) by using a
' variable to set the FOR...NEXT StepValue, and incrementing StepValue
' within the loop. Sir Isaac Newton is generally credited with the
' discovery of this technique.

' {$STAMP BS2}
' {$PBASIC 2.5}

square VAR Byte ' FOR/NEXT counter
stepSize VAR Byte ' step size increases by 2 each loop

Setup:
 stepSize = 1
 square = 1

Main:
 FOR square = 1 TO 250 STEP stepSize ' show squares up to 250
 DEBUG DEC ? square ' display on screen
 stepSize = stepSize + 2 ' add 2 to stepSize
 NEXT ' loop until square > 250
END
```

## ***FOR...NEXT – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – FREQOUT

### FREQOUT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



(See SOUND)



**FREQOUT** *Pin, Duration, Freq1 { , Freq2 }*

### Function

Generate one or two sine-wave tones for a specified *Duration*.

- **Pin** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This pin will be set to output mode.
- **Duration** is a variable/constant/expression (0 - 65535) specifying the amount of time to generate the tone(s). The unit of time for *Duration* is described in Table 5.26.
- **Freq1** is a variable/constant/expression (0 – 32767) specifying frequency of the first tone. The unit of *Freq1* is described in Table 5.26.
- **Freq2** is an optional argument exactly like *Freq1*. When specified, two frequencies will be mixed together on the specified I/O pin.

### Quick Facts

**Table 5.26:** FREQOUT Quick Facts.

|                                 | BS2, BS2e       | BS2sx         | BS2p           | BS2pe         | BS2px          |
|---------------------------------|-----------------|---------------|----------------|---------------|----------------|
| <b>Units in Duration</b>        | 1 ms            | 0.4 ms        | 0.265 ms       | 1 ms          | 0.166 ms       |
| <b>Units in Freq1 and Freq2</b> | 1 Hz            | 2.5 Hz        | 3.77 Hz        | 1.51 Hz       | 6.03 Hz        |
| <b>Range of Frequency</b>       | 0 to 32767 Hz   | 0 to 81917 Hz | 0 to 123531 Hz | 0 to 49478 Hz | 0 to 197585 Hz |
| <b>Related Commands</b>         | DTMFOUT and PWM |               |                |               |                |

### Explanation

FREQOUT generates one or two sine waves using a pulse-width modulation algorithm. The circuits shown in Figure 5.6 will filter the signal in order to play the tones through a speaker or audio amplifier. Here's a simple FREQOUT command:

SIMPLEST FORM OF FREQOUT.

```
FREQOUT 2, 1000, 2500
```

## FREQOUT – BASIC Stamp Command Reference

On the BS2, this command generates a 2500 Hz tone for 1 second (1000 ms) on I/O pin 2. See Table 5.26 for timing data on other BASIC Stamp models.

To play two tones on the same I/O pin at once:

```
FREQOUT 2, 1000, 2500, 3000
```

GENERATING TWO TONES AT ONCE.

This will generate a 2500 Hz and 3000 Hz tone for 1 second. The frequencies will mix together for a chord- or bell-like sound. To generate a silent pause, specify frequency value(s) of 0.

### Driving an Audio Amplifier

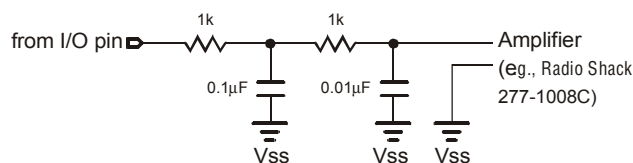
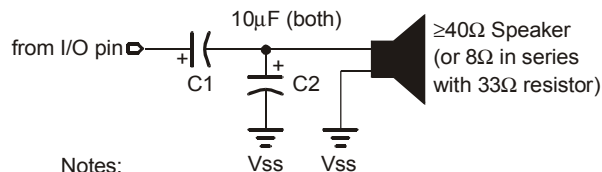


**Figure 5.6:** Example RC filter circuits for driving an audio amplifier(top) or a speaker (bottom).

### Driving a Speaker



Notes:  
C1 may be omitted for piezo speakers  
C2 is optional, but reduces high-frequency noise

The circuits in Figure 5.6 work by filtering out the high-frequency PWM used to generate the sine waves. FREQOUT works over a very wide range of frequencies (as shown in Table 5.26) so at the upper end of its range, those PWM filters will also filter out most of the desired frequency. You may find it necessary to reduce values of the parallel capacitors shown in the circuit, or to devise a custom active filter for your application.

FREQUENCY CONSIDERATIONS.

## 5: BASIC Stamp Command Reference – FREQOUT

All 2

NOTE: This example program can be used with all BS2 models. This program uses conditional compilation techniques; see Chapter 3 for more information.

### Demo Program (FREQOUT.bs2)

```
' FREQOUT.bs2
' This program demonstrates sound-effects generation by the BASIC Stamp.
' Conditional compilation sets timing and frequency adjustment factors so
' that the output will sound the same on any BS2 model.

' {$STAMP BS2}
' {$PBASIC 2.5}

Spkr PIN 10 ' output pin for FREQOUT

#SELECT $STAMP
#CASE BS2, BS2E
 TmAdj CON $100 ' x 1.0 (time adjust)
 FrAdj CON $100 ' x 1.0 (freq adjust)
#CASE BS2SX
 TmAdj CON $280 ' x 2.5
 FrAdj CON $066 ' x 0.4
#CASE BS2P
 TmAdj CON $3C5 ' x 3.77
 FrAdj CON $044 ' x 0.265
#CASE BS2PE
 TmAdj CON $100 ' x 1.0
 FrAdj CON $0A9 ' x 0.662
#CASE BS2PX
 TmAdj CON $607 ' x 6.03
 FrAdj CON $2A ' x 0.166
#ENDSELECT

Main:
DEBUG "Let's make a call...", CR
' combine 350 Hz & 440 Hz
FREQOUT Spkr, 2000 */ TmAdj, 350 */ FrAdj, 440 */ FrAdj
' dial number (digits 150 ms on, 25 ms off)
DTMFOUT Spkr, 150 */ TmAdj, 25, [5, 5, 5, 1, 2, 1, 2]
PAUSE 500

' bad connection (SIT sequence)
FREQOUT Spkr, 375 */ TmAdj, 985 */ FrAdj
FREQOUT Spkr, 375 */ TmAdj, 1371 */ FrAdj
FREQOUT Spkr, 375 */ TmAdj, 1777 */ FrAdj

DEBUG "Oops! -- try again...", CR
PAUSE 1000
DTMFOUT Spkr, 150 */ TmAdj, 25, [5, 5, 5, 2, 2, 2, 2]
DEBUG "Ringing"
FREQOUT Spkr, 2000 */ TmAdj, 440 */ FrAdj, 480 */ FrAdj
PAUSE 4000
FREQOUT Spkr, 2000 */ TmAdj, 440 */ FrAdj, 480 */ FrAdj
INPUT Spkr
END
```

## ***FREQOUT – BASIC Stamp Command Reference***

---



2e 2sx 2p 2pe 2px

**GET** *Location*, { **WORD** } *Variable* { , { **WORD** } *Variable*... }

## Function

- **Location** is a variable/constant/expression (0 – 63 for BS2e and BS2sx and 0 – 131 for BS2p, BS2pe, and BS2px) that specifies the SPRAM location to read from.
- **Variable** is a variable (usually a byte, or word if using the optional WORD modifier) in which to store the value.

**Table 5.27: GET Quick Facts.**

|                                              | BS2e and BS2sx                                                                                                                             | BS2p, BS2pe, and BS2px                                                                                                        |
|----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <b>Scratch Pad RAM Size and Organization</b> | 64 bytes (0 – 63). Organized as bytes only.                                                                                                | 136 bytes (0 – 135). Organized as bytes only.                                                                                 |
| <b>General Purpose Locations</b>             | 0 - 62                                                                                                                                     | 0 – 126                                                                                                                       |
| <b>Special Use Location</b>                  | Location 63: Active program slot number (read only).                                                                                       | Location 127: READ/WRITE slot and Active Program slot (read only).<br>Locations 128-135: Polled Interrupt status (read only). |
| <b>Related Commands</b>                      | PUT                                                                                                                                        | PUT and STORE,<br>and SPSTR formatter.                                                                                        |
| <b>PBASIC 2.5 Syntax Options</b>             | Multiple sequential variables may be read from the Scratch Pad RAM. The optional WORD modifier may be specified to retrieve 16-bit values. |                                                                                                                               |

The GET command reads a value from the specified Scratch Pad RAM location and stores it into *Variable*. All values in all locations can be retrieved from within any of the 8 program slots.

SPRAM is useful for passing data to programs in other program slots and for additional workspace. It is different than regular RAM in that symbol names cannot be assigned directly to locations and each location is always configured as a byte only. The following code will read the value at location 25, store it in a variable called *temp* and display it:

## GET – BASIC Stamp Command Reference

---

temp     VAR     Byte

```
GET 25, temp ' retrieve byte from location 25
DEBUG DEC temp
```

When using the PBASIC 2.5 directive, multiple sequential variables may be read from the Scratch Pad RAM, starting at *Location*, and the WORD modifier may be specified to retrieve 16-bit values.

```
' {$PBASIC 2.5}
```

temp     VAR     Byte  
temp2    VAR     Word

```
GET 25, temp, Word temp2 ' retrieve byte from location 25
 ' and word from locations 26 and 27
DEBUG DEC temp, CR
DEBUG DEC temp2
```

The low nibble of location 63 (BS2e and BS2sx) and location 127 (BS2p, BS2pe, and BS2px) is a special, read-only location that always contains the number of the currently running program slot. On the BS2p, BS2pe, and BS2px, the high nibble of location 127 also contains the current program slot that will be used for the READ and WRITE commands. See the demo program below for an example of use.

Table 5.28 shows the layout of all SPRAM registers.

## 5: BASIC Stamp Command Reference – GET

**Table 5.28:** Layout of SPRAM Registers.

NOTE: Scratch Pad RAM can only be accessed with the GET and PUT commands. Scratch Pad RAM cannot have variable names assigned to it.

| Location | BS2e and BS2sx                        | BS2p, BS2pe, and BS2px                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|----------|---------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0...62   | General Purpose RAM                   | General Purpose RAM                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 63       | Bits 0-3: Active program slot number. | General Purpose RAM                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 64...126 | n/a                                   | General Purpose RAM                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 127      | n/a                                   | Bits 0-3, Active program slot #. Bits 4-7, program slot for READ and WRITE operations.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 128      | n/a                                   | Polled input trigger status of Main I/O pins 0-7 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| 129      | n/a                                   | Polled input trigger status of Main I/O pins 8-15 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| 130      | n/a                                   | Polled input trigger status of Auxiliary I/O pins 0-7 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 131      | n/a                                   | Polled input trigger status of Auxiliary I/O pins 8-15 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| 132      | n/a                                   | Bits 0-3: Polled-interrupt mode, set by POLLMODE                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| 133      | n/a                                   | Bits 0-2: Polled-interrupt "run" slot, set by POLLRUN.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 134      | n/a                                   | Bit 0: Active I/O group; 0 = Main I/O, 1 = Auxiliary I/O.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| 135      | n/a                                   | Bit 0: Polled-output status (set by POLLMODE); 0 = disabled, 1 = enabled.<br>Bit 1: Polled-input status; 0 = none defined, 1 = at least one defined.<br>Bit 2: Polled-run status (set by POLLMODE); 0 = disabled, 1 = enabled.<br>Bit 3: Polled-output latch status; 0 = real-time mode, 1 = latch mode.<br>Bit 4: Polled-input state; 0 = no trigger, 1 = triggered.<br>Bit 5: Polled-output latch state; 0 = nothing latched, 1 = signal latched.<br>Bit 6: Poll-wait state; 0 = No Event, 1 = Event Occurred. (Cleared by POLLMODE only).<br>Bit 7: Polling status; 0 = not active, 1 = active. |



NOTE: This is written for the BS2sx but can be used with the BS2e, BS2p, BS2pe and BS2px also. This program uses conditional compilation techniques; see Chapter 3 for more information.

### Demo Program (GET\_PUT1.bsx)

```
' GET_PUT1.bsx
' This example demonstrates the use of the GET and PUT commands. First,
' slot location is read using GET to display the currently running program
' number. Then a set of values are written (PUT) into locations 0 TO 9.
' Afterwards, program number 1 is RUN. This program is a BS2SX project
' consisting of GET_PUT1.BSX and GET_PUT2.BSX, but will run on the BS2e,
' BS2p, BS2pe and BS2px without modification.

' {$STAMP BS2sx, GET_PUT2.BSX}
' {$PBASIC 2.5}
```

## GET – BASIC Stamp Command Reference

```
#SELECT $STAMP
#CASE BS2
 #ERROR "BS2e or greater required."
#CASE BS2E, BS2SX
 Slot CON 63
#CASE BS2P, BS2PE, BS2PX
 Slot CON 127
#ENDSELECT

value VAR Byte
idx VAR Byte

Setup:
 GET Slot, value
 DEBUG "Program Slot #", DEC value.NIB0, CR

Main:
 FOR idx = 0 TO 9
 value = (idx + 3) * 8
 PUT idx, value
 DEBUG " Writing: ", DEC2 value, " to location: ", DEC2 idx, CR
 NEXT
 DEBUG CR
 RUN 1
END
```

### Demo Program (GET\_PUT2.bsx)

```
' GET_PUT2.bsx
' This example demonstrates the use of the GET and PUT commands. First,
' the Slot location is read using GET to display the currently running
' program number. Then a set of values are read (GET) from locations
' 0 to 9 and displayed on the screen for verification. This program is a
' BS2SX project consisting of GET_PUT1.BSX and GET_PUT2.BSX, but will run
' on the BS2e, BS2p, BS2pe, and BS2px without modification.

' {$STAMP BS2sx}
' {$PBASIC 2.5}

#SELECT $STAMP
#CASE BS2
 #ERROR "BS2e or greater required."
#CASE BS2E, BS2SX
 Slot CON 63
#CASE BS2P, BS2PE, BS2PX
 Slot CON 127
#ENDSELECT

value VAR Byte
idx VAR Byte
```



NOTE: This is written for the BS2sx but can be used with the BS2e, BS2p, BS2pe and BS2px also. This program uses conditional compilation techniques; see Chapter 3 for more information.

## ***5: BASIC Stamp Command Reference – GET***

---

```
Setup:
 GET Slot, value
 DEBUG "Program Slot #", DEC value.NIB0, CR

Main:
 FOR idx = 0 TO 9
 GET idx, value
 DEBUG " Reading: ", DEC2 value, " from location: ", DEC2 idx, CR
 NEXT
END
```

## ***GET – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – GOSUB

### GOSUB

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



#### GOSUB Address

#### Function

Store the address of the next instruction after GOSUB, then go to the point in the program specified by *Address*; with the intention of returning to the stored address.

- **Address** is a label that specifies where to go.

#### Quick Facts

Table 5.29: GOSUB Quick Facts.

|                         | BS1  | All BS2 Models      |
|-------------------------|------|---------------------|
| Max. GOSUBs per program | 16   | 255                 |
| Max. nested GOSUBs      | 4    | 4                   |
| Related Commands        | GOTO | ON...GOSUB and GOTO |

#### Explanation

GOSUB is a close relative of GOTO, in fact, its name means, "GO to a SUBroutine". When a PBASIC program reaches a GOSUB, the program executes the code beginning at the specified address label. Unlike GOTO, GOSUB also stores the address of the instruction immediately following itself. When the program encounters a RETURN command, it interprets it to mean, "go to the instruction that follows the most recent GOSUB." In other words, a GOSUB makes the BASIC Stamp do a similar operation as you do when you see a table or figure reference in this manual; 1) you remember where you are, 2) you go to the table or figure and read the information there, and 3) when you've reached the end of it, you "return" to the place you were reading originally.

GOSUB CAN SAVE EEPROM  
(PROGRAM) SPACE.

GOSUB is mainly used to execute the same piece of code from multiple locations. If you have, for example, a block of three lines of code that need to be run from 10 different locations in your entire program you could simply copy and paste those three lines to each of those 10 locations. This would amount to a total of 30 lines of repetitive code (and extra space wasted in the program memory). A better solution is to place those three lines in a separate routine, complete with it's own label and followed by a

## ***GOSUB – BASIC Stamp Command Reference***

---

RETURN command, then just use a GOSUB command at each of the 10 locations to access it. This technique can save a lot of program space.

Try the example below:

```
Main:
 GOSUB Hello
 DEBUG "How are you?"
 END

Hello:
 DEBUG "Hello, my friend!", CR
 RETURN
```

The above code will start out by GOSUB'ing to the section of code beginning with the label *Hello*. It will print "Hello, my friend!" on the screen then RETURN to the line after the GOSUB... which prints "How are you?" and ENDS. Note: colons ( : ) are placed after labels, as in "Main: " and "Hello:" but the colon is not used on references to these labels such as in the "GOSUB Hello" line.

There's another interesting lesson here; what would happen if we removed the END command from this example? Since the BASIC Stamp reads the code from left to right / top to bottom (like the English language) once it had returned to and run the "How are you?" line, it would naturally "fall into" the *Hello* routine again. Additionally, at the end of the *Hello* routine, it would see the RETURN again (although it didn't GOSUB to that routine this time) and because there wasn't a previous place to return to, the BASIC Stamp will start the entire program over again. This would cause an endless loop. The important thing to remember here is to always make sure your program doesn't allow itself to "fall into" a subroutine.

WATCH OUT FOR SUBROUTINES THAT  
YOUR PROGRAM CAN "FALL INTO."



NOTE: On the BS1, a RETURN without a GOSUB will return the program to the last GOSUB (or will end the program if no GOSUB was executed).

A limited number of GOSUBs are allowed per program (as shown in Table 5.29), and they may be nested only four levels deep. In other words, the subroutine that's the destination of a GOSUB can contain a GOSUB to another subroutine, and so on, to a maximum depth (total number of GOSUBs before the first RETURN) of four. Any deeper, and the program will "forget" its way back to the starting point (the instruction following the very first GOSUB).

GOSUB LIMITATIONS.

When GOSUBS are nested, each RETURN takes the program back to the instruction after the most-recent GOSUB. As is mentioned above, if the



## 5: BASIC Stamp Command Reference – GOSUB

BASIC Stamp encounters a RETURN without a previous GOSUB, the entire program starts over from the beginning. Take care to avoid these phenomena.



### Demo Program (GOSUB.bs1)

```
' GOSUB.bs1
' This program is a guessing game that generates a random number in a
' subroutine called Pick_A_Number. It is written to stop after ten
' guesses. To see a common bug associated with GOSUB, delete or comment
' out the line beginning with END after the FOR-NEXT loop. This means
' that after the loop is finished, the program will wander into the
' Pick_A_Number subroutine. When the RETURN at the end executes, the
' program will go back to the beginning of the program. This will cause
' the program to execute endlessly. Make sure that your programs can't
' accidentally execute subroutines!

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL rounds = B2 ' number of reps
SYMBOL numGen = W0 ' random number holder
SYMBOL myNum = B3 ' random number, 1-10

Setup:
 numGen = 11500 ' initialize random "seed"

Main:
 FOR rounds = 1 TO 10
 DEBUG CLS, "Pick a number from 1 to 10", CR
 GOSUB Pick_A_Number
 PAUSE 2000 ' dramatic pause
 DEBUG "My number was: ", #myNum ' show the number
 PAUSE 1000 ' another pause.
 NEXT
 DEBUG CLS, "Done"
 END ' end program

' Random-number subroutine. A subroutine is just a piece of code with
' the RETURN instruction at the end. Always make sure your program enters
' subroutines with a GOSUB. If you don't, the RETURN won't have the
' correct address, and your program will have a bug!

Pick_A_Number:
 RANDOM numGen ' stir up the bits of NumGen.
 DEBUG numGen, CR
 myNum = numGen / 6550 MIN 1 ' scale to fit 1-10 range.
 RETURN ' go back to 1st instruction
 ' after GOSUB that got us here
```

## ***GOSUB – BASIC Stamp Command Reference***

---

### **Demo Program (GOSUB.bs2)**

```
' GOSUB.bs2
' This program is a guessing game that generates a random number in a
' subroutine called Pick_A_Number. It is written to stop after ten
' guesses. To see a common bug associated with GOSUB, delete or comment
' out the line beginning with END after the FOR-NEXT loop. This means
' that after the loop is finished, the program will wander into the
' Pick_A_Number subroutine. When the RETURN at the end executes, the
' program will go back to the beginning of the program. This will cause
' the program to execute endlessly. Make sure that your programs can't
' accidentally execute subroutines!

' {$STAMP BS2}
' {$PBASIC 2.5}

rounds VAR Byte ' number of reps
numGen VAR Word ' random number holder
myNum VAR Byte ' random number, 1-10

Setup:
 numGen = 11500 ' initialize random "seed"

Main:
 FOR rounds = 1 TO 10
 DEBUG CLS, "Pick a number from 1 to 10", CR
 GOSUB Pick_A_Number
 PAUSE 2000 ' dramatic pause
 DEBUG "My number was: ", DEC myNum ' show the number
 PAUSE 1000 ' another pause.
 NEXT
 DEBUG CLS, "Done"
 END ' end program

' Random-number subroutine. A subroutine is just a piece of code with
' the RETURN instruction at the end. Always make sure your program enters
' subroutines with a GOSUB. If you don't, the RETURN won't have the
' correct address, and your program will have a bug!

Pick_A_Number:
 RANDOM numGen ' stir up the bits of NumGen.
 DEBUG DEC ? numGen
 myNum = numGen / 6550 MIN 1 ' scale to fit 1-10 range.
 RETURN ' go back to 1st instruction
 ' after GOSUB that got us here
```



NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## 5: BASIC Stamp Command Reference – GOTO

### GOTO

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



#### GOTO Address

### Function

Go to the point in the program specified by *Address*.

- **Address** is a label that specifies where to go.

### Quick Facts

Table 5.30: GOTO Quick Facts.

|                        | BS1                                                                                | All BS2 Models              |
|------------------------|------------------------------------------------------------------------------------|-----------------------------|
| Related Commands       | BRANCH and GOSUB                                                                   | ON...GOTO, BRANCH and GOSUB |
| Max. GOTOs per Program | Unlimited, but good programming practices suggest using the least amount possible. |                             |

### Explanation

The GOTO command makes the BASIC Stamp execute the code that starts at the specified *Address* location. The BASIC Stamp reads PBASIC code from left to right / top to bottom, just like in the English language. The GOTO command forces the BASIC Stamp to jump to another section of code.

A common use for GOTO is to create endless loops; programs that repeat a group of instructions over and over. For example:

```
Start:
 DEBUG "Hi", CR
GOTO Start
```

The above code will print "Hi" on the screen, over and over again. The GOTO Start line simply tells it to go back to the code that begins with the label *Start*. Note: colons ( : ) are placed after labels, as in "Start:" to further indicate that they are labels, but the colon is not used on references to labels such as in the "GOTO Start" line.



### Demo Program (GOTO.bs2)

NOTE: This is written for the BS2 but can be used for the BS1 and all other BS2 models as well, by modifying the \$STAMP directive accordingly.

```
' GOTO.bs2
' This program is not very practical, but demonstrates the use of GOTO to
' jump around the code. This code jumps between three different routines,
' each of which print something different on the screen. The routines are
' out of order for this example.
```

## ***GOTO – BASIC Stamp Command Reference***

---

```
'{$STAMP BS2}

GOTO Routine1

Routine2:
 DEBUG "We're in routine #2",CR
 PAUSE 1000
GOTO Routine3

Routine1:
 DEBUG "We're in routine #1",CR
 PAUSE 1000
GOTO Routine2

Routine3:
 DEBUG "We're in routine #3",CR
 PAUSE 1000
GOTO Routine1
```

## 5: BASIC Stamp Command Reference – HIGH

### HIGH



#### HIGH Pin

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px | BS2px |
|-----|-----|------|-------|------|-------|-------|-------|
|-----|-----|------|-------|------|-------|-------|-------|



NOTE: Expressions are not allowed as arguments on the BS1. The range of the *Pin* argument on the BS1 is 0 – 7.

### Function

Make the specified pin output high.

- **Pin** is a variable/constant/expression (0 – 15) that specifies which I/O pin to set high. This pin will be placed into output mode.

### Quick Facts

Table 5.31: HIGH Quick Facts.

|                  | BS1 and all BS2 Models |
|------------------|------------------------|
| Related Commands | LOW and TOGGLE         |

### Explanation

The HIGH command sets the specified pin to 1 (a +5 volt level) and then sets its mode to output. For example,

```
HIGH 6
```

does exactly the same thing as:

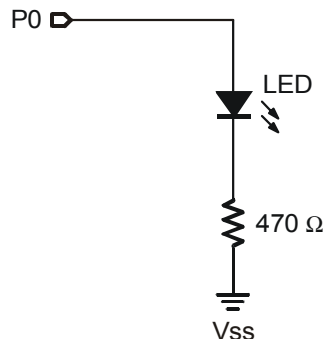


```
OUT6 = 1
DIR6 = 1
```

Using the HIGH command is faster and more concise, in this case.

Connect an LED and a resistor as shown in Figure 5.7 for demo program HIGH.bs2, below.

Figure 5.7: Example LED Circuit.



## ***HIGH – BASIC Stamp Command Reference***

---

### **Demo Program (HIGH.bs2)**

```
' HIGH.bs2
' This simple program sets I/O pin 0 high for 1/2 second and low for
' 1/2 second in an endless loop. Connect an LED to P0 for a simple
' blinker.

' {$STAMP BS2}

Main:
 HIGH 0
 PAUSE 500
 LOW 0
 PAUSE 500
 GOTO Main
END
```



NOTE: This is written for the BS2 but can be used for the BS1 and all other BS2 models as well, by modifying the \$STAMP directive accordingly.

## 5: BASIC Stamp Command Reference – I2CIN

### I2CIN

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**I2CIN** *Pin*, *SlaveID*, { *Address* { \ *LowAddress* }, } [ *InputData* ]

#### Function

Receive data from a device using the I<sup>2</sup>C protocol.

- ***Pin*** is a variable/constant/expression (0 or 8) that specifies which I/O pins to use. I<sup>2</sup>C devices require two I/O pins to communicate. The *Pin* argument serves a double purpose; specifying the first pin (for connection to the chip's SDA pin) and, indirectly, the other required pin (for connection to the chip's SCL pin). See explanation below. Both I/O pins will be toggled between output and input mode during the I2CIN command and both will be set to input mode by the end of the I2CIN command.
- ***SlaveID*** is a variable/constant/expression (0 – 255) indicating the unique ID of the I<sup>2</sup>C chip.
- ***Address*** is an optional variable/constant/expression (0 – 255) indicating the desired address within the I<sup>2</sup>C chip to receive data from. The *Address* argument may be used with the optional *LowAddress* argument to indicate a word-sized address value.
- ***LowAddress*** is an optional variable/constant/expression (0 – 255) indicating the low-byte of the word-sized address within the I<sup>2</sup>C chip to receive data from. This argument must be used along with the *Address* argument.
- ***InputData*** is a list of variables and modifiers that tells I2CIN what to do with incoming data. I2CIN can store data in a variable or array, interpret numeric text (decimal, binary, or hex) and store the corresponding value in a variable, wait for a fixed or variable sequence of bytes, or ignore a specified number of bytes. These actions can be combined in any order in the *InputData* list.

## I2CIN – BASIC Stamp Command Reference

### Quick Facts

|                     | BS2p, BS2pe, and BS2px                                                                                                                                                                                          |                                                       |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------|
| Values for Pin      | Pin = 0                                                                                                                                                                                                         | Pin = 8                                               |
| I/O Pin Arrangement | 0: Serial Data (SDA) pin<br>1: Serial Clock (SCL) pin                                                                                                                                                           | 8: Serial Data (SDA) pin<br>9: Serial Clock (SCL) pin |
| Transmission Rate   | Approximately 81 kbits/sec on a BS2p, 45 kbits/sec on a BS2pe, and 83 kbits/sec on a BS2px (not including overhead).                                                                                            |                                                       |
| Special Notes       | Both the SDA and SCL pins must have 1 k $\Omega$ - 4.7 k $\Omega$ pull-up resistors. The I2CIN command does not allow for multiple masters. The BASIC Stamp cannot operate as an I <sup>2</sup> C slave device. |                                                       |
| Related Command     | I2COUT                                                                                                                                                                                                          |                                                       |

Table 5.32: I2CIN Quick Facts.

### Explanation

The I<sup>2</sup>C protocol is a form of synchronous serial communication developed by Phillips Semiconductors. It only requires two I/O pins and both pins can be shared between multiple I<sup>2</sup>C devices. The I2CIN command allows the BASIC Stamp to receive data from an I<sup>2</sup>C device.

The following is an example of the I2CIN command:

A SIMPLE I2CIN EXAMPLE.

```
result VAR Byte
I2CIN 0, $A1, 0, [result]
```

This code will transmit a "read" command to an I<sup>2</sup>C device (connected to I/O pins 0 and 1) and then will receive one byte and store it in the variable *result*. Though it may seem strange, the I2CIN command first transmits some data and then receives data. It must first transmit information (ID, read/write and address) in order to tell the I<sup>2</sup>C device what information it would like to receive. The exact information transmitted (\$A1, 0) depends on the I<sup>2</sup>C device that is being used.

NOTE: The I2C command will make up to 8 attempts to connect to the addressed device. If the device does not properly respond, the I2C command will timeout and the *InputData* will remain unchanged..

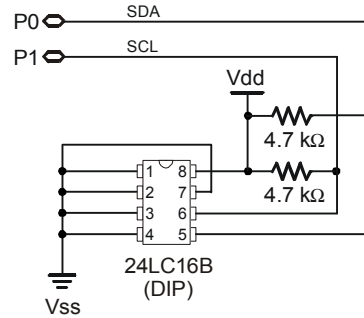
The above example will read a byte of data from location 0 of a 24LC16B EEPROM from Microchip. Figure 5.8 shows the proper wiring for this example to work. The *SlaveID* argument (\$A1) is both the ID of the chip and the command to read from the chip; the 1 means read. The *Address* argument (0) is the EEPROM location to read from.



## 5: BASIC Stamp Command Reference – I2CIN

**Figure 5.8:** Example Circuit for the I2CIN command and a 24LC16B EEPROM.

**Note:** The 4.7 k $\Omega$  resistors are required for the I2CIN command to function properly.



### RECEIVING FORMATTED DATA.

The I2CIN command's *InputData* argument is similar to the SERIN command's *InputData* argument. This means data can be received as ASCII character values, decimal, hexadecimal and binary translations and string data as in the examples below. (Assume the 24LC16B EEPROM is used and it has the string, "Value: 3A:101" stored, starting at location 0).

```
value VAR Byte(13)

I2CIN 0, $A1, 0, [value] ' receive the ASCII value for "V"
I2CIN 0, $A1, 0, [DEC value] ' receive the number 3
I2CIN 0, $A1, 0, [HEX value] ' receive the number $3A
I2CIN 0, $A1, 0, [BIN value] ' receive the number %101
I2CIN 0, $A1, 0, [STR value\13] ' receive the string "Value: 3A:101"
```

Table 5.33 and Table 5.34 below list all the available special formatters and conversion formatters available to the I2CIN command. See the SERIN command for additional information and examples of their use.

**Table 5.33:** I2CIN Special Formatters.

| Special Formatter                     | Action                                                                                                                                                                                                                                            |
|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SKIP <i>Length</i>                    | Ignore <i>Length</i> bytes of characters.                                                                                                                                                                                                         |
| SPSTR <i>L</i>                        | Input a character stream of length <i>L</i> bytes (up to 126) into Scratch Pad RAM, starting at location 0. Use GET to retrieve the characters.                                                                                                   |
| STR <i>ByteArray</i> \L { <i>E</i> }  | Input a character string of length <i>L</i> into an array. If specified, an end character <i>E</i> causes the string input to end before reaching length <i>L</i> . Remaining bytes are filled with 0s (zeros).                                   |
| WAITSTR <i>ByteArray</i> { <i>L</i> } | Wait for a sequence of bytes matching a string stored in an array variable, optionally limited to <i>L</i> characters. If the optional <i>L</i> argument is left off, the end of the array-string must be marked by a byte containing a zero (0). |

## I2CIN – BASIC Stamp Command Reference

| Conversion Formatter | Type of Number                                                                                         | Numeric Characters Accepted        | Notes   |
|----------------------|--------------------------------------------------------------------------------------------------------|------------------------------------|---------|
| DEC{1..5}            | Decimal, optionally limited to 1 – 5 digits                                                            | 0 through 9                        | 1       |
| SDEC{1..5}           | Signed decimal, optionally limited to 1 – 5 digits                                                     | -, 0 through 9                     | 1,2     |
| HEX{1..4}            | Hexadecimal, optionally limited to 1 – 4 digits                                                        | 0 through 9, A through F           | 1,3,5   |
| SHEX{1..4}           | Signed hexadecimal, optionally limited to 1 – 4 digits                                                 | -, 0 through 9, A through F        | 1,2,3   |
| IHEX{1..4}           | Indicated hexadecimal, optionally limited to 1 – 4 digits                                              | \$, 0 through 9, A through F       | 1,3,4   |
| ISHEX{1..4}          | Signed, indicated hexadecimal, optionally limited to 1 – 4 digits                                      | -, \$, 0 through 9, A through F    | 1,2,3,4 |
| BIN{1..16}           | Binary, optionally limited to 1 – 16 digits                                                            | 0, 1                               | 1       |
| SBIN{1..16}          | Signed binary, optionally limited to 1 – 16 digits                                                     | -, 0, 1                            | 1,2     |
| IBIN{1..16}          | Indicated binary, optionally limited to 1 – 16 digits                                                  | %, 0, 1                            | 1,4     |
| ISBIN{1..16}         | Signed, indicated binary, optionally limited to 1 – 16 digits                                          | -, %, 0, 1                         | 1,2,4   |
| NUM                  | Generic numeric input (decimal, hexadecimal or binary); hexadecimal or binary number must be indicated | \$, %, 0 through 9, A through F    | 1, 3, 4 |
| SNUM                 | Similar to NUM with value treated as signed with range -32768 to +32767                                | -, \$, %, 0 through 9, A through F | 1,2,3,4 |

**Table 5.34:** I2CIN Conversion Formatters.

- 1 All numeric conversions will continue to accept new data until receiving either the specified number of digits (ex: three digits for DEC3) or a non-numeric character.
- 2 To be recognized as part of a number, the minus sign (-) must immediately precede a numeric character. The minus sign character occurring in non-numeric text is ignored and any character (including a space) between a minus and a number causes the minus to be ignored.
- 3 The hexadecimal formatters are not case-sensitive; “a” through “f” means the same as “A” through “F”.
- 4 Indicated hexadecimal and binary formatters ignore all characters, even valid numerics, until they receive the appropriate prefix (\$ for hexadecimal, % for binary). The indicated formatters can differentiate between text and hexadecimal (ex: ABC would be interpreted by HEX as a number but IHEX would ignore it unless expressed as \$ABC). Likewise, the binary version can distinguish the decimal number 10 from the binary number %10. A prefix occurring in non-numeric text is ignored, and any character (including a space) between a prefix and a number causes the prefix to be ignored. Indicated, signed formatters require that the minus sign come before the prefix, as in -\$1B45.
- 5 The HEX modifier can be used for Decimal to BCD Conversion. See “Hex to BCD Conversion” on page 97.

For examples of all conversion formatters and how they process incoming data, see Appendix C.

## 5: BASIC Stamp Command Reference – I2CIN

### THE I<sup>2</sup>C PROTOCOL FORMAT.

The I<sup>2</sup>C protocol has a well-defined standard for the information passed at the start of each transmission. First of all, any information sent must be transmitted in units of 1 byte (8-bits). The first byte, we call the *SlaveID*, is an 8-bit pattern whose upper 7-bits contain the unique ID of the device you wish to communicate with. The lowest bit indicates whether this is a write operation (0) or a read operation (1). Figure 5.9 shows this format.

Figure 5.9: Slave ID Format.

|                |                |                |                |                |                |                |     |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----|
| 7              | 6              | 5              | 4              | 3              | 2              | 1              | 0   |
| A <sub>6</sub> | A <sub>5</sub> | A <sub>4</sub> | A <sub>3</sub> | A <sub>2</sub> | A <sub>1</sub> | A <sub>0</sub> | R/W |

The second byte, immediately following the *SlaveID*, is the optional *Address*. It indicates the 8-bit address (within the device) containing the data you would like to receive. Note that the *Address* argument is optional and may be left unspecified for devices that don't require an *Address* argument.

### USING LONG ADDRESSES.

Some devices require more than 8 bits of address. For this case, the optional *LowAddress* argument can be used for the low-byte of the required address. When using the *LowAddress* argument, the *Address* argument is effectively the high-byte of the address value. For example, if the entire address value is 2050, use 8 for the *Address* argument and 2 for the *LowAddress* argument ( $8 * 256 + 2 = 2050$ ).

Following the last address byte is the first byte of data. This data byte may be transmitted or received by the BASIC Stamp. In the case of the I2CIN command, this data byte is transmitted by the device and received by the BASIC Stamp. Additionally, multiple data bytes can follow the address, depending on the I<sup>2</sup>C device. Note that every device has different limitations regarding how many contiguous bytes they can receive or transmit in one session. Be aware of these device limitations and program accordingly.

### START AND STOP CONDITIONS AND ACKNOWLEDGMENTS.

Every I<sup>2</sup>C transmission session begins with a Start Condition and ends with a Stop Condition. Additionally, immediately after every byte is transmitted, an extra clock cycle is used to send or receive an acknowledgment signal (ACK). All of these operations are automatically taken care of by the I2CIN command so that you need not be concerned with them. The general I<sup>2</sup>C transmission format is shown in Figure 5.10.

## I2CIN – BASIC Stamp Command Reference

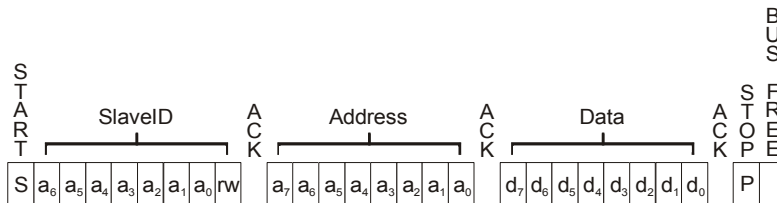


**Figure 5.10:** I<sup>2</sup>C Transmission Format.

### NOTES:

S = Start Condition

P = Stop Condition

a = id or address bit

d = data bit (transmitted by the BASIC Stamp or the I<sup>2</sup>C device)

ACK = Acknowledge signal. (Most acknowledge signals are generated by the I<sup>2</sup>C device)

Since the I2CIN command is intended for input only, it actually overrides the "R/W" bit (bit 0) in the *SlaveID* argument. This is done so that it can use the I<sup>2</sup>C protocol's "Combined Format" for receiving data. Put simply, this means a command such as: I2CIN 0, \$A1, 10, [Result] actually transmits \$A0, then 10, then \$A1 and then it reads the data back from the device. The \$A0 means "write", the 10 is the address to write to, and finally, the \$A1 indicates a change of direction; to "read" the location, instead. Even though the I2CIN command really doesn't care what the value of the *SlaveID*'s LSB is, it is suggested that you still set it appropriately for clarity.

**SPECIAL NOTE ABOUT I2CIN IMPLEMENTATION.**

Also note that the I2CIN command does not support multiple I<sup>2</sup>C masters and the BASIC Stamp cannot operate as an I<sup>2</sup>C slave device.

### Demo Program (I2C.bsp)

```
' I2C.bsp
' This program demonstrates writing and reading every location in a 24LC16B
' EEPROM using the BS2p/BS2pe's I2C commands. Connect the BS2p, BS2pe, or
' BS2px to the 24LC16B DIP EEPROM as shown in the diagram in the I2CIN or
' I2COUT command description.

' {$STAMP BS2p}
' {$PBASIC 2.5}

#IF ($STAMP < BS2P) #THEN
 #ERROR "Program requires BS2p, BS2pe, or BS2px."
#ENDIF

SDA PIN 0 ' I2C SDA pin
SCL PIN SDA + 1
```



**NOTE:** This example program can be used with the BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

## 5: BASIC Stamp Command Reference – I2CIN

```
addr VAR Word ' internal address
block VAR Nib ' block address in 24LC16
value VAR Byte ' value to write
check VAR Nib ' for checking returned values
result VAR Byte(16) ' array for returned value

Write_To_EEPROM:
 DEBUG "Writing...", CR
 PAUSE 2000
 FOR addr = 0 TO 2047 STEP 16 ' loop through all addresses
 block = addr.NIB2 << 1 ' calculate block address
 value = addr >> 4 ' create value from upper 8 bits
 ' write 16 bytes
 I2COUT SDA, $A0 | block, addr, [REP value\16]
 PAUSE 5
 DEBUG "Addr: ", DEC4 addr, "-", DEC4 addr + 15, " ",
 "Value: ", DEC3 value, CR
 NEXT
 PAUSE 2000

Read_From_EEPROM:
 DEBUG CR, "Reading...", CR
 PAUSE 2000
 FOR addr = 0 TO 2047 STEP 16
 block = addr.NIB2 << 1
 value = addr >> 4
 I2CIN SDA, $A1 | block, addr, [STR result\16]
 FOR check = 0 TO 15
 IF (result(check) <> value) THEN Error
 NEXT
 DEBUG "Addr: ", DEC4 addr, "-", DEC4 addr + 15, " ",
 "Value: ", DEC3 result, CR
 NEXT
 PAUSE 100
 DEBUG CR, "All locations passed"
 END

Error:
 DEBUG "Error at location: ", DEC4 addr + check, CR,
 "Found: ", DEC3 result(check), ", Expected: ", DEC3 value
 END
```

## ***I2CIN – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – I2COUT

### I2COUT

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|



**I2COUT** *Pin*, *SlaveID*, { *Address* { \ *LowAddress* }, } [ *OutputData* ]

#### Function

Send data to a device using the I<sup>2</sup>C protocol.

- ***Pin*** is a variable/constant/expression (0 or 8) that specifies which I/O pins to use. I<sup>2</sup>C devices require two I/O pins to communicate. The *Pin* argument serves a double purpose; specifying the first pin (for connection to the chip's SDA pin) and, indirectly, the other required pin (for connection to the chip's SCL pin). See explanation below. Both I/O pins will be toggled between output and input mode during the I2COUT command and both will be set to input mode by the end of the I2COUT command.
- ***SlaveID*** is a variable/constant/expression (0 – 255) indicating the unique ID of the I<sup>2</sup>C chip.
- ***Address*** is an optional variable/constant/expression (0 – 255) indicating the desired address within the I<sup>2</sup>C chip to send data to. The *Address* argument may be used with the optional *LowAddress* argument to indicate a word-sized address value.
- ***LowAddress*** is an optional variable/constant/expression (0 – 255) indicating the low-byte of the word-sized address within the I<sup>2</sup>C chip to receive data from. This argument must be used along with the *Address* argument.
- ***OutputData*** is a list of variables, constants, expressions and formatters that tells I2COUT how to format outgoing data. I2COUT can transmit individual or repeating bytes, convert values into decimal, hexadecimal or binary text representations, or transmit strings of bytes from variable arrays. These actions can be combined in any order in the *OutputData* list.

# I2COUT – BASIC Stamp Command Reference

## Quick Facts

|                     | BS2p, BS2pe, and BS2px                                                                                                                                                                     |                                                       |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------|
| Values for Pin      | Pin = 0                                                                                                                                                                                    | Pin = 8                                               |
| I/O Pin Arrangement | 0: Serial Data (SDA) pin<br>1: Serial Clock (SCL) pin                                                                                                                                      | 8: Serial Data (SDA) pin<br>9: Serial Clock (SCL) pin |
| Transmission Rate   | Approximately 81 kbits/sec on a BS2p, 45 kbits/sec on a BS2pe, and 83 kbits/sec on a BS2px (not including overhead).                                                                       |                                                       |
| Special Notes       | The SDA and SCL pins must have 1 kΩ - 4.7 kΩ pull-up resistors. The I2CIN command does not allow for multiple masters. The BASIC Stamp cannot operate as an I <sup>2</sup> C slave device. |                                                       |
| Related Command     | I2CIN                                                                                                                                                                                      |                                                       |

Table 5.35: I2COUT Quick Facts.

## Explanation

The I<sup>2</sup>C protocol is a form of synchronous serial communication developed by Phillips Semiconductors. It only requires two I/O pins and both pins can be shared between multiple I<sup>2</sup>C devices. The I2COUT command allows the BASIC Stamp to send data to an I<sup>2</sup>C device.

The following is an example of the I2COUT command:

A SIMPLE I2COUT EXAMPLE.

```
I2COUT 0, $A0, 5, [100]
```

This code will transmit a "write" command to an I<sup>2</sup>C device (connected to I/O pins 0 and 1), followed by an address of 5 and finally will transmit the number 100.

The above example will write a byte of data to location 5 of a 24LC16B EEPROM from Microchip. Figure 5.11 shows the proper wiring for this example to work. The *SlaveID* argument (\$A0) is both the ID of the chip and the command to write to the chip; the 0 means write. The *Address* argument (5) is the EEPROM location to write to.

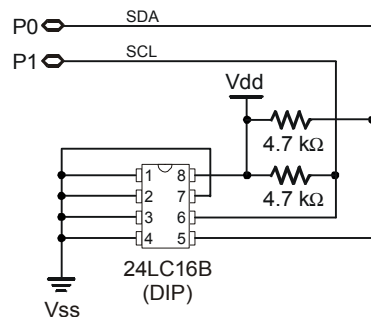


Figure 5.11: Example Circuit for the I2COUT command and a 24LC16B EEPROM.

**Note:** The 4.7 kΩ resistors are required for the I2COUT command to function properly.



## 5: BASIC Stamp Command Reference – I2COUT

### SENDING AND FORMATTING DATA.

The I2COUT command's *OutputData* argument is similar to the DEBUG and SEROUT command's *OutputData* argument. This means data can be sent as literal text, ASCII character values, repetitive values, decimal, hexadecimal and binary translations and string data as in the examples below. (Assume the 24LC16B EEPROM is being used).

```
value VAR Byte
value = 65

I2COUT 0, $A0, 0, [value] ' send "A"
I2COUT 0, $A0, 0, [REP value\5] ' send "AAAAA"
I2COUT 0, $A0, 0, [DEC value] ' send "6" and "5"
I2COUT 0, $A0, 0, [HEX value] ' send "4" and "1"
I2COUT 0, $A0, 0, [BIN value] ' send "1000001"
```

Table 5.36 and Table 5.37 list all the available formatters for the I2COUT command. See the DEBUG and SEROUT commands for additional information and examples of their use.

**Table 5.36:** I2COUT Conversion Formatters.

| Conversion Formatter | Type of Number                                                              | Notes |
|----------------------|-----------------------------------------------------------------------------|-------|
| DEC{1..5}            | Decimal, optionally fixed to 1 – 5 digits                                   | 1     |
| SDEC{1..5}           | Signed decimal, optionally fixed to 1 – 5 digits                            | 1,2   |
| HEX{1..4}            | Hexadecimal, optionally fixed to 1 – 4 digits                               | 1,3   |
| SHEX{1..4}           | Signed hexadecimal, optionally fixed to 1 – 4 digits                        | 1,2   |
| IHEX{1..4}           | Indicated hexadecimal, optionally fixed to 1 – 4 digits (\$ prefix)         | 1     |
| ISHEX{1..4}          | Signed, indicated hexadecimal, optionally fixed to 1 – 4 digits (\$ prefix) | 1,2   |
| BIN{1..16}           | Binary, optionally fixed to 1 – 16 digits                                   | 1     |
| SBIN{1..16}          | Signed binary, optionally fixed to 1 – 16 digits                            | 1,2   |
| IBIN{1..16}          | Indicated binary, optionally fixed to 1 – 16 digits (% prefix)              | 1     |
| ISBIN{1..16}         | Signed, indicated binary, optionally fixed to 1 – 16 digits (% prefix)      | 1,2   |

1 Fixed-digit formatters like DEC4 will pad the number with leading 0s if necessary; ex: DEC4 65 sends 0065. If a number is larger than the specified number of digits, the leading digits will be dropped; ex: DEC4 56422 sends 6422.

2 Signed modifiers work under two's complement rules.

3 The HEX modifier can be used for BCD to Decimal Conversion. See "Hex to BCD Conversion" on page 97.

## I2COUT – BASIC Stamp Command Reference

| Special Formatter                 | Action                                                                                                                                                                                                                             |
|-----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ?                                 | Displays "symbol = x' + carriage return; where x is a number. Default format is decimal, but may be combined with conversion formatters (ex: BIN ? x to display "x = binary_number").                                              |
| ASC ?                             | Displays "symbol = 'x' + carriage return; where x is an ASCII character.                                                                                                                                                           |
| STR <i>ByteArray</i> { <i>L</i> } | Send character string from an array. The optional <i>L</i> argument can be used to limit the output to <i>L</i> characters, otherwise, characters will be sent up to the first byte equal to 0 or the end of RAM space is reached. |
| REP <i>Byte</i> \L                | Send a string consisting of <i>Byte</i> repeated <i>L</i> times (ex: REP "X"\10 sends "XXXXXXXXXX").                                                                                                                               |

**Table 5.37:** I2COUT Special Formatters

The I<sup>2</sup>C protocol has a well-defined standard for the information passed at the start of each transmission. First of all, any information sent must be transmitted in units of 1 byte (8-bits). The first byte, we call the *SlaveID*, is an 8-bit pattern whose upper 7-bits contain the unique ID of the device you wish to communicate with. The lowest bit indicates whether this is a write operation (0) or a read operation (1). Figure 5.12 shows this format.

THE I<sup>2</sup>C PROTOCOL FORMAT.

|                |                |                |                |                |                |                |     |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|-----|
| 7              | 6              | 5              | 4              | 3              | 2              | 1              | 0   |
| A <sub>6</sub> | A <sub>5</sub> | A <sub>4</sub> | A <sub>3</sub> | A <sub>2</sub> | A <sub>1</sub> | A <sub>0</sub> | R/W |

**Figure 5.12:** Slave ID Format

The second byte, immediately following the *SlaveID*, is the optional *Address*. It indicates the 8-bit address (within the device) containing the data you would like to receive. Note that the *Address* argument is optional and may be left unspecified for devices that don't require an *Address* argument.

Some devices require more than 8 bits of address. For this case, the optional *LowAddress* argument can be used for the low-byte of the required address. When using the *LowAddress* argument, the *Address* argument is effectively the high-byte of the address value. For example, if the entire address value is 2050, use 8 for the *Address* argument and 2 for the *LowAddress* argument ( $8 * 256 + 2 = 2050$ ).

USING LONG ADDRESSES.

Following the last address byte is the first byte of data. This data byte may be transmitted or received by the BASIC Stamp. In the case of the I2COUT command, this data byte is transmitted by the BASIC Stamp and received by the device. Additionally, multiple data bytes can follow the address,

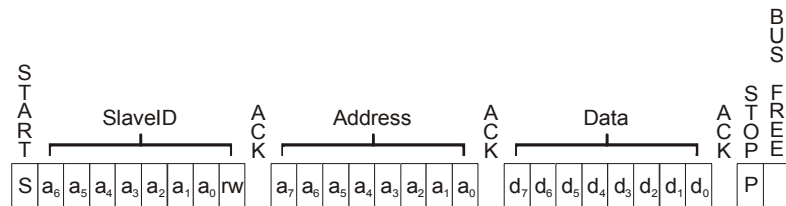
## 5: BASIC Stamp Command Reference – I2COUT

depending on the I<sup>2</sup>C device. Note that every device has different limitations regarding how many contiguous bytes they can receive or transmit in one session. Be aware of this, and program accordingly.

START AND STOP CONDITIONS AND  
ACKNOWLEDGMENTS.

Every I<sup>2</sup>C transmission session begins with a Start Condition and ends with a Stop Condition. Additionally, immediately after every byte is transmitted, an extra clock cycle is used to send or receive an acknowledgment signal (ACK). All of these operations are automatically taken care of by the I2COUT command so that you need not be concerned with them. The general I<sup>2</sup>C transmission format is shown in Figure 5.13.

**Figure 5.13:** I<sup>2</sup>C Transmission Format



### NOTES:

S = Start Condition

P = Stop Condition

a = id or address bit

d = data bit (transmitted by the BASIC Stamp or the I<sup>2</sup>C device)

ACK = Acknowledge signal. (Most acknowledge signals are generated by the I<sup>2</sup>C device)

SPECIAL NOTE ABOUT I2COUT  
IMPLEMENTATION.

Since the I2COUT command is intended for output only, it actually overrides the "R/W" bit (bit 0) in the *SlaveID* argument. This is done to avoid device conflicts should the value be mistyped. Put simply, this means commands such as: I2COUT 0, \$A0, 10, [0] and I2COUT 0, \$A1, 10, [0] both transmit the same thing (\$A0, then 10, then the data). Even though the I2COUT command really doesn't care what the value of the *SlaveID*'s LSB is, it is suggested that you still set it appropriately for clarity.

Also note that the I2COUT command does not support multiple I<sup>2</sup>C masters and the BASIC Stamp cannot operate as an I<sup>2</sup>C slave device.



### Demo Program (I2C.bsp)

NOTE: This example program can be used with the BS2p, BS2pe and BS2px. This program uses conditional compilation techniques; see Chapter 3.

```
' I2C.bsp
' This program demonstrates writing and reading every location in a 24LC16B
' EEPROM using the BS2p/BS2pe's I2C commands. Connect the BS2p, BS2pe, or
' BS2px to the 24LC16B DIP EEPROM as shown in the diagram in the I2CIN or
```

## I2COUT – BASIC Stamp Command Reference

---

```
' I2COUT command description.

' {$STAMP BS2p}
' {$PBASIC 2.5}

#IF ($STAMP < BS2P) #THEN
 #ERROR "Program requires BS2p, BS2pe or BS2px."
#ENDIF

SDA PIN 0 ' I2C SDA pin
SCL PIN SDA + 1

addr VAR Word ' internal address
block VAR Nib ' block address in 24LC16
value VAR Byte ' value to write
check VAR Nib ' for checking returned values
result VAR Byte(16) ' array for returned value

Write_To_EEPROM:
 DEBUG "Writing...", CR
 PAUSE 2000
 FOR addr = 0 TO 2047 STEP 16 ' loop through all addresses
 block = addr.NIB2 << 1 ' calculate block address
 value = addr >> 4 ' create value from upper 8 bits
 ' write 16 bytes
 I2COUT SDA, $A0 | block, addr, [REP value\16]
 PAUSE 5
 DEBUG "Addr: ", DEC4 addr, "-", DEC4 addr + 15, " ",
 "Value: ", DEC3 value, CR
 NEXT
 PAUSE 2000

Read_From_EEPROM:
 DEBUG CR, "Reading...", CR
 PAUSE 2000
 FOR addr = 0 TO 2047 STEP 16
 block = addr.NIB2 << 1
 value = addr >> 4
 I2CIN SDA, $A1 | block, addr, [STR result\16]
 FOR check = 0 TO 15
 IF (result(check) <> value) THEN Error
 NEXT
 DEBUG "Addr: ", DEC4 addr, "-", DEC4 addr + 15, " ",
 "Value: ", DEC3 result, CR
 NEXT
 PAUSE 100
 DEBUG CR, "All locations passed"
 END

Error:
 DEBUG "Error at location: ", DEC4 addr + check, CR,
 "Found: ", DEC3 result(check), ", Expected: ", DEC3 value
 END
```

## 5: BASIC Stamp Command Reference – IF...THEN

NOTE: PBASIC 1.0 and PBASIC 2.0 only support IF *Condition(s)* THEN *Address*. PBASIC 2.5 supports all syntax variations.



All 2

### IF...THEN

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|

IF *Condition(s)* THEN *Address*



All 2

IF *Condition(s)* THEN *Statement(s)* { ELSE *Statement(s)* }



All 2

IF *Condition(s)* THEN  
*Statement(s)*  
 { ELSEIF *Condition(s)* THEN  
*Statement(s)...* }  
 { ELSE  
*Statement(s)* }  
 ENDIF

### Function

Evaluate *Condition(s)* and, if true, go to the *Address* or execute the *Statement(s)* following THEN, otherwise process the ELSEIF/ELSE block(s), if provided. ELSEIF is optional and works just like IF, but is only evaluated if the *Condition(s)* in the preceding IF is false. The ELSE block is optional and is executed if all *Condition(s)* in all preceding IF/ELSEIFs are false. The program will continue at the next line of code (single-line syntax) or the line that follows ENDIF (multi-line syntax) unless *Address* or *Statement(s)* are executed that cause the program to jump.

- **Condition** is a statement, such as “x = 7” that can be evaluated as true or false. *Condition* can be a very simple or very complex relationship, as described below.
- **Address** is a label that specifies where to go in the event that *Condition(s)* is true.
- **Statement** is any valid PBASIC instruction. Multiple statements may be placed on the same line (though not recommended) by separating each statement with a colon ( : ).

# IF...THEN – BASIC Stamp Command Reference

## Quick Facts

|                             | BS1                                                                                | All BS2 Models                                                                                                            |
|-----------------------------|------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| Comparison Operators        | =, <>, >, <, >=, <=                                                                | =, <>, >, <, >=, <=                                                                                                       |
| Conditional Logic Operators | AND, OR                                                                            | NOT, AND, OR, XOR                                                                                                         |
| Format of Condition         | <i>Variable Comparison Value</i> ;<br>where <i>Value</i> is a variable or constant | <i>Value1 Comparison Value2</i> ;<br>where <i>Value1</i> and <i>Value2</i> can be any of variable, constant or expression |
| Parentheses                 | Not Allowed                                                                        | Allowed                                                                                                                   |
| Max nested IF...THENs       | n/a                                                                                | 16                                                                                                                        |
| Max ELSEIFs per IF          | n/a                                                                                | 16                                                                                                                        |
| Max ELSEs per IF            | n/a                                                                                | 1                                                                                                                         |
| Related Command             | None                                                                               | SELECT...CASE                                                                                                             |

Table 5.38: IF...THEN Quick Facts.

## Explanation

IF...THEN is PBASIC's decision maker that allows one block of code or another to run based on the value (True or False) of a condition. The condition that IF...THEN tests is written as a mixture of comparison and logic operators. The available comparison operators are:

| Comparison Operator Symbol | Definition               |
|----------------------------|--------------------------|
| =                          | Equal                    |
| <>                         | Not Equal                |
| >                          | Greater Than             |
| <                          | Less Than                |
| >=                         | Greater Than or Equal To |
| <=                         | Less Than or Equal To    |

Table 5.39: IF...THEN Comparison Operators.

Comparisons are always written in the form: Value1 Comparison Value2. The values to be compared can be any combination of variables (any size), constants, or expressions.

NOTE: On the BS1, expressions are not allowed as arguments. Also, the Value1 (to the left of comparison) must be a variable.

The following example is an IF...THEN command with a simple condition:

```
IF value < 4000 THEN Main
```



A SIMPLE FORM OF IF...THEN

This code will compare the value of *value* to the number 4000. If *value* is less than 4000, the condition is true and the program will jump (implied

## 5: BASIC Stamp Command Reference – IF...THEN

GOTO) to the label called *Main*. This is the simplest form of IF...THEN and is the only form supported by PBASIC 1.0 and PBASIC 2.0.

ALL ABOUT *CONDITION(S)*.

The *Condition(s)* argument is very powerful and flexible. In the next few pages we'll demonstrate this flexibility in great detail and afterwards we'll discuss the additional, optional arguments allowed by PBASIC 2.5.

Here's a complete example of IF...THEN in action:

NOTE: For BS1's, change line 1 to  
**SYMBOL value = W0**  
and line 4 to  
**DEBUG #value, CR**

```
value VAR Word

Main:
 PULSIN 0, 1, value
 DEBUG DEC value, cr
 IF value < 4000 THEN Main
 DEBUG "Pulse value was greater than 3999!"
```

Here, the BASIC Stamp will look for and measure a pulse on I/O pin 0, then compare the result, *value*, against 4000. If *value* is less than (<) 4000, it will jump back to Main. Each time through the loop, it displays the measured value and once it is greater than or equal to 4000, it displays, "Value was greater than 3999!"



On all BS2 models, the values can be expressions as well. This leads to very flexible and sophisticated comparisons. The IF...THEN statement below is functionally the same as the one in the program above:

```
IF value < (45 * 100 - (25 * 20)) THEN Val_Low
```



Here the BASIC Stamp evaluates the expression:  $45 * 100 = 4500$ ,  $25 * 20 = 500$ , and  $4500 - 500 = 4000$ . Then the BASIC Stamp performs the comparison: is *value* < 4000? Another example that is functionally the same:

```
IF (value / 100) < 40 THEN Val_Low
```

WATCH OUT FOR UNSIGNED MATH  
COMPARISON ERRORS



It's important to realize that all comparisons are performed using unsigned, 16-bit math. This can lead to strange results if you mix signed and unsigned numbers in IF...THEN conditions. Watch what happens here when we include a signed number (-99):

## IF...THEN – BASIC Stamp Command Reference

---

```
Test:
 IF -99 < 100 THEN Is_Less
 DEBUG "Greater than or equal to 100"
END
```

```
Is_Less:
 DEBUG "Less than 100"
END
```

Although -99 is obviously less than 100, the program will say it is greater. The problem is that -99 is internally represented as the two's complement value 65437, which (using unsigned math) is greater than 100. This phenomena will occur whether or not the negative value is a constant, variable or expression.

IF...THEN supports the conditional logic operators NOT, AND, OR, and XOR to allow for more sophisticated conditions, such as multi-part conditions. See Table 5.38 for a list of the operators and Table 5.40 for their effects.

LOGICAL OPERATORS (NOT, AND, OR AND XOR).

The NOT operator inverts the outcome of a condition, changing false to true, and true to false. The following IF...THENs are equivalent:

NOTE: The NOT and XOR operators are not available on the BS1.

```
IF x <> 100 THEN Not_Equal
IF NOT x = 100 THEN Not_Equal
```

The operators AND, OR, and XOR can be used to join the results of two conditions to produce a single true/false result. AND and OR work the same as they do in everyday speech. Run the example below once with AND (as shown) and again, substituting OR for AND:

```
value1 VAR Byte
value2 VAR Byte
```

```
Setup:
 value1 = 5
 value2 = 9
```

```
Main:
 IF value1 = 5 AND value2 = 10 THEN Is_True
 DEBUG "Statement is False"
END
```

```
Is_True:
 DEBUG "Statement is True"
END
```



NOTE: For BS1's, change lines1 and 2 to:

**SYMBOL value1 = B2**  
**SYMBOL value2 = B3**



## 5: BASIC Stamp Command Reference – IF...THEN

The condition “value1 = 5 AND value2 = 10” is not true. Although value1 is 5, value2 is not 10. The AND operator works just as it does in English; both conditions must be true for the statement to be true. The OR operator also works in a familiar way; if one or the other or both conditions are true, then the statement is true. The XOR operator (short for exclusive-OR) may not be familiar, but it does have an English counterpart: If one condition or the other (but not both) is true, then the statement is true.



NOTE: On the BS1, parentheses are not allowed within arguments.

Table 5.40 below summarizes the effects of the conditional logic operators. On all BS2 models you can alter the order in which comparisons and logical operations are performed by using parentheses. Operations are normally evaluated left-to-right. Putting parentheses around an operation forces PBASIC 2.0 and PBASIC 2.5 to evaluate it before operations that are not in parentheses.

**Table 5.40:** Conditional Logic Operators Truth Tables.



NOTE: The NOT and XOR operators are not available on the BS1.

| Truth Table for Logical Operator: NOT |       |
|---------------------------------------|-------|
| Condition A                           | NOT A |
| False                                 | True  |
| True                                  | False |

| Truth Table for Logical Operator: AND |             |         |
|---------------------------------------|-------------|---------|
| Condition A                           | Condition B | A AND B |
| False                                 | False       | False   |
| False                                 | True        | False   |
| True                                  | False       | False   |
| True                                  | True        | True    |

| Truth Table for Logical Operator: OR |             |        |
|--------------------------------------|-------------|--------|
| Condition A                          | Condition B | A OR B |
| False                                | False       | False  |
| False                                | True        | True   |
| True                                 | False       | True   |
| True                                 | True        | True   |

| Truth Table for Logical Operator: XOR |             |         |
|---------------------------------------|-------------|---------|
| Condition A                           | Condition B | A XOR B |
| False                                 | False       | False   |
| False                                 | True        | True    |
| True                                  | False       | True    |
| True                                  | True        | False   |

## IF...THEN – BASIC Stamp Command Reference

---

Internally, the BASIC Stamp defines “false” as 0 and “true” as any value other than 0. Consider the following instructions:

INTERNAL REPRESENTATION OF BOOLEAN VALUES (TRUE VS. FALSE).

```
flag VAR Bit

Setup:
 flag = 1

Test:
 IF flag THEN Is_True
 DEBUG "False"
END

Is_True:
 DEBUG "True"
END
```



Since *flag* is 1, IF...THEN would evaluate it as true and print the message “True” on the screen. Suppose you changed the IF...THEN command to read “IF NOT flag THEN Is\_True.” That would also evaluate as true. Whoa! Isn’t NOT 1 the same thing as 0? No, at least not in the 16-bit world of the BASIC Stamp.

Internally, the BASIC Stamp sees a bit variable containing 1 as the 16-bit number %0000000000000001. So it sees the NOT of that as %1111111111111110. Since any non-zero number is regarded as true, NOT 1 is true. Strange but true.

The easiest way to avoid the kinds of problems this might cause is to always use a conditional operator with IF...THEN. Change the example above to read IF flag = 1 THEN is\_True. The result of the comparison will follow IF...THEN rules. Also, the logical operators will work as they should; IF NOT Flag = 1 THEN is\_True will correctly evaluate to false when *flag* contains 1.

AVOIDING ERRORS WITH BOOLEAN RESULTS.

This also means that you should only use the “named” conditional logic operators NOT, AND, OR, and XOR with IF...THEN. The conditional logic operators format their results correctly for IF...THEN instructions. The other logical operators, represented by symbols ~, &, |, and ^ do not; they are binary logic operators.

The remainder of this discussion only applies to the extended IF...THEN syntax supported by PBASIC 2.5.



## 5: BASIC Stamp Command Reference – IF...THEN

---

### IF...THEN WITH A SINGLE STATEMENT

In addition to supporting everything discussed above, PBASIC 2.5 provides enhancements to the IF...THEN command that allow for more powerful, structured programming. In prior examples we've only used the first syntax form of this command: IF *Condition(s)* THEN *Address*. That form, while handy in some situations, can be quite limiting in others. For example, it is common to need to perform a single instruction based on a condition. Take a look at the following code:

```
' {$PBASIC 2.5}

x VAR Byte

FOR x = 1 TO 20 ' count to 20
 DEBUG CR, DEC x ' display num
 IF (x // 2) = 0 THEN DEBUG " EVEN" ' even num?
NEXT
```

This example prints the numbers 1 through 20 on the screen but every even number is also marked with the text " EVEN." The IF...THEN command checks to see if *x* is even or odd and, if it is even (i.e.:  $x // 2 = 0$ ), then it executes the statement to the right of THEN: DEBUG " EVEN." If it was odd, it simply continued at the following line, NEXT.

Suppose you also wanted to mark the odd numbers. You could take advantage of the optional ELSE clause, as in:

```
' {$PBASIC 2.5}

x VAR Byte

FOR x = 1 TO 20 ' count to 20
 DEBUG CR, DEC x ' display num
 IF (x // 2) = 0 THEN DEBUG " EVEN" ELSE DEBUG " ODD"
NEXT
```

This example prints the numbers 1 through 20 with " EVEN" or " ODD" to the right of each number. For each number (each time through the loop) IF...THEN asks the question, "Is the number even?" and if it is it executes DEBUG " EVEN" (the instruction after THEN) or, if it is not even it executes DEBUG " ODD" (the instruction after ELSE). It's important to note that this form of IF...THEN always executes code as a result of *Condition(s)*; it either does "this" (THEN) or "that" (ELSE).

## IF...THEN – BASIC Stamp Command Reference

---

The IF...THEN in the previous example is called a “single-line” syntax. It is most useful where you only have one instruction to run as the result of a *Condition*. Sometimes this form may be a little hard to read, like in our above example. For these cases, it would be better to use the “multi-line” syntax of IF...THEN. The multi-line format allows the same flexibility as the single-line format, but is easier to read in most cases and requires an ENDIF statement at the end. For example, our IF...THEN line above could be re-written as: SINGLE-LINE VS. MULTI-LINE IF...THENS

```
IF (x // 2) = 0 THEN
 DEBUG " EVEN" ' even number
ELSE
 DEBUG " ODD" ' odd number
ENDIF
```

This example runs exactly the same way, is much easier to read and also leaves extra room to add some helpful comments on the right. We also indented the *Statement(s)* for clarity and suggest you do the same.

Did you notice that multi-line syntax requires ENDIF to mark the end of the IF...THEN...ELSE construct? That is because the *Statement(s)* argument can be multiple instructions on multiple lines, so without ENDIF there is no way to know just where the IF...THEN...ELSE ends.

Occasionally, it may be necessary to have compound IF statements. One way to achieve this is through nested IF...THEN...END constructs: NESTED IF...THENS

```
' {$PBASIC 2.5}

value VAR Word

DO
 PULSIN 0, 1, value ' measure pulse input
 DEBUG DEC value, CR
 IF (value > 4000) THEN ' evaluate duration
 DEBUG "Value was greater than 4000"
 ELSE
 IF (value = 4000) THEN
 DEBUG "Value was equal to 4000"
 ELSE
 DEBUG "Value was less than 4000"
 ENDIF
 ENDIF
 DEBUG CR, CR
 PAUSE 1000
LOOP
```

## 5: BASIC Stamp Command Reference – IF...THEN

---

Here, the BASIC Stamp will look for and measure a pulse on I/O pin 0, then compare the result, *value*, against 4000. Based on this comparison, a message regarding the pulse width value will be printed.

If *value* is greater than 4000, "Value was greater than 4000" is printed to the screen. Look what happens if *value* is not greater than 4000... the code in the ELSE block is run, which is another IF...THEN...ELSE statement. This "nested" IF...THEN statement checks if *value* is equal to 4000 and if it is, it prints "Value was equal to 4000" or, if it was not equal, the last ELSE code block executes, printing "Value was less than 4000." Up to sixteen (16) IF...THENs can be nested like this.

### USING THE ELSEIF CLAUSE

The nesting option is great for many situations, but, like single-line syntax, may be a little hard to read, especially if there are multiple nested statements or there is more than one instruction in each of the *Statement(s)* arguments. Additionally, every multi-line IF...THEN construct must end with ENDIF, resulting in two ENDIFs right near each other in our example; one for the innermost IF...THEN and one for the outermost IF...THEN. For this reason, IF...THEN supports an optional ELSEIF clause. The ELSEIF clause takes the place of a nested IF...THEN and removes the need for multiple ENDIFs. Our IF...THEN construction from the example above could be rewritten to:

```
' {$PBASIC 2.5}

IF (value > 4000) THEN ' evaluate duration
 DEBUG "Value was greater than 4000"
ELSEIF (value = 4000) THEN
 DEBUG "Value was equal to 4000"
ELSE
 DEBUG "Value was less than 4000"
ENDIF
```

This IF...THEN construct does the same thing as in the previous example:

- 1) if value is greater than 4000:  
it displays "Value was greater than 4000"
- 2) else, if value is equal to 4000 (the ELSEIF part):  
it displays "Value was equal to 4000"
- 3) and finally (ELSE) if none of the above were true:  
it displays "Value was less than 4000"

## IF...THEN – BASIC Stamp Command Reference

---

Note that an IF...THEN construct can have as many as sixteen (16) ELSEIF clauses, but a maximum of only one (1) ELSE clause.

There are three demo programs below. The first two demonstrate the PBASIC 1.0 (BS1) and PBASIC 2.0 (all BS2 models) form of the IF...THEN command. The last example demonstrates the PBASIC 2.5 (all BS2 models) form of IF...THEN.

### Demo Program (IF-THEN.bs1)



```
' IF-THEN.bs1
' The program below generates a series of 16-bit random numbers and tests
' each to determine whether they're evenly divisible by 3. If a number is
' evenly divisible by 3, then it is printed, otherwise, the program
' generates another random number. The program counts how many numbers it
' prints, and quits when this number reaches 10.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL sample = W0 ' Random number to be tested
SYMBOL samps = B2 ' Number of samples taken
SYMBOL temp = B3 ' Temporary workspace

Setup:
 sample = 11500

Mult3:
 RANDOM sample ' Put a random number into sample
 temp = sample // 3
 IF temp <> 0 THEN Mult3 ' Not multiple of 3? -- try again
 DEBUG #sample, "divides by 3", CR ' show sample divisible by 3
 samps = samps + 1 ' Count multiples of 3
 IF samps = 10 THEN Done ' Quit with 10 samples
 GOTO Mult3 ' keep checking

Done:
 DEBUG CR, "All done."
 END
```

### Demo Program (IF-THEN.bs2)



```
' IF-THEN.bs2
' The program below generates a series of 16-bit random numbers and tests
' each to determine whether they're evenly divisible by 3. If a number is
' evenly divisible by 3, then it is printed, otherwise, the program
' generates another random number. The program counts how many numbers it
' prints, and quits when this number reaches 10.
```

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## 5: BASIC Stamp Command Reference – IF...THEN

```
' {$STAMP BS2}
' {$PBASIC 2.0}

sample VAR Word ' Random number to be tested
samps VAR Nib ' Number of samples taken
temp VAR Nib ' Temporary workspace

Setup:
 sample = 11500

Mult3:
 RANDOM sample ' Put a random number into sample
 temp = sample // 3
 IF temp <> 0 THEN Mult3 ' Not multiple of 3? -- try again
 DEBUG DEC5 sample, " divides by 3", CR
 samps = samps + 1 ' Count multiples of 3
 IF samps = 10 THEN Done ' Quit with 10 samples
 GOTO Mult3 ' keep checking

Done:
 DEBUG CR, "All done."
 END
```



### Demo Program (IF-THEN-ELSE.bs2)

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

```
' IF-THEN-ELSE.bs2
' The program below generates a series of 16-bit random numbers and tests
' each to determine whether they're evenly divisible by 3. If a number is
' evenly divisible by 3, then it is printed, otherwise, the program
' generates another random number. The program counts how many numbers it
' prints, and quits when this number reaches 10.

' {$STAMP BS2}
' {$PBASIC 2.5} ' version 2.5 required

sample VAR Word ' Random number to be tested
hits VAR Nib ' Number of hits
misses VAR Word ' Number of misses

Setup:
 sample = 11500

Main:
 DO
 RANDOM sample ' Put a random number into sample
 IF ((sample // 3) = 0) THEN ' divisible by 3?
 DEBUG DEC5 sample, ' - yes, print value and message
 " is divisible by 3", CR

```

## ***IF...THEN – BASIC Stamp Command Reference***

---

```
 hits = hits + 1 ' count hits (divisible by 3)
 ELSE
 misses = misses + 1 ' count misses
 ENDIF
LOOP UNTIL (hits = 10) ' quit after 10 hits

DEBUG CR,
 "All done.", CR, CR, ' display results
 "Hits: ", DEC hits, CR,
 "Misses: ", DEC misses, CR,
 "Samples: ", DEC (hits + misses)
END
```



# 5: BASIC Stamp Command Reference – INPUT

|                                                                                                |                  |            |             |              |             |              |              |
|------------------------------------------------------------------------------------------------|------------------|------------|-------------|--------------|-------------|--------------|--------------|
|                                                                                                | <b>BS1</b>       | <b>BS2</b> | <b>BS2e</b> | <b>BS2sx</b> | <b>BS2p</b> | <b>BS2pe</b> | <b>BS2px</b> |
|  <b>All 2</b> | <b>INPUT Pin</b> |            |             |              |             |              |              |

1

NOTE: Expressions are not allowed as arguments on the BS1. The range of the PIN argument on the BS1 is 0 – 7.

## Function

Make the specified pin an input.

- **Pin** is a variable/constant/expression (0 – 15) that specifies which I/O pin to set to input mode.

## Quick Facts

Table 5.41: INPUT Quick Facts.

|                     | BS1                     | All BS2 Models        |
|---------------------|-------------------------|-----------------------|
| Input Pin Variables | PINS; PIN0 through PIN7 | INS; IN0 through IN15 |
| Related Commands    | OUTPUT and REVERSE      |                       |

## Explanation

There are several ways to make a pin an input. When a program begins, all of the BASIC Stamp's pins are inputs. Commands that rely on input pins, like PULSIN and SERIN, automatically change the specified pin to input. Writing 0s to particular bits of the variable DIRS makes the corresponding pins inputs. And then there's the INPUT command.

When a pin is an input, your program can check its state by reading the corresponding INS variable (PINS on the BS1). For example:

 **All 2**

```
INPUT 4
Hold:
 IF IN4 = 0 THEN Hold ' stay here until P4 = 1
```

The code above will read the state of P4 as set by external circuitry. If nothing is connected to P4, it will alternate between states (1 or 0) apparently at random.

What happens if your program writes to the OUTS bit (PINS bit on the BS1) of a pin that is set up as an input? The value is stored in OUTS (PINS on the BS1), but has no effect on the outside world. If the pin is changed to output, the last value written to the corresponding OUTS bit (or PINS bit

## INPUT – BASIC Stamp Command Reference

---

on the BS1) will appear on the pin. The demo program shows how this works.

### Demo Program (INPUT.bs1)



```
' INPUT.bs1
' This program demonstrates how the input/output direction of a pin is
' determined by the corresponding bit of DIRS. It also shows that the
' state of the pin itself (as reflected by the corresponding bit of PINS)
' is determined by the outside world when the pin is an input, and by the
' corresponding bit of OUTS when it's an output. To set up the demo,
' connect a 10k resistor from +5V to P7 on the BASIC Stamp. The resistor
' to +5V puts a high (1) on the pin when it's an input. The BASIC Stamp
' can override this state by writing a low (0) to bit 7 of OUTS and
' changing the pin to output.

' {$STAMP BS1}
' {$PBASIC 1.0}

Main:
 INPUT 7 ' Make P7 an input
 DEBUG "State of P7: ", #PIN7, CR

 PIN7 = 0 ' Write 0 to output latch
 DEBUG "After 0 written to OUT7: "
 DEBUG #PIN7, CR

 OUTPUT 7 ' Make P7 an output
 DEBUG "After P7 changed to output: "
 DEBUG #PIN7
```

### Demo Program (INPUT.bs2)



```
' INPUT.bs2
' This program demonstrates how the input/output direction of a pin is
' determined by the corresponding bit of DIRS. It also shows that the
' state of the pin itself (as reflected by the corresponding bit of INS)
' is determined by the outside world when the pin is an input, and by the
' corresponding bit of OUTS when it's an output. To set up the demo,
' connect a 10k resistor from +5V to P7 on the BASIC Stamp. The resistor
' to +5V puts a high (1) on the pin when it's an input. The BASIC Stamp
' can override this state by writing a low (0) to bit 7 of OUTS and
' changing the pin to output.

' {$STAMP BS2}
' {$PBASIC 2.5}

Main:
 INPUT 7 ' Make P7 an input
 DEBUG "State of P7: ",
```

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## ***5: BASIC Stamp Command Reference – INPUT***

---

```
BIN1 IN7, CR

OUT7 = 0 ' Write 0 to output latch
DEBUG "After 0 written to OUT7: ",
 BIN1 IN7, CR

OUTPUT 7 ' Make P7 an output
DEBUG "After P7 changed to output: ",
 BIN1 IN7
```

## ***INPUT – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – IOTERM

|                                                                                   |                    |            |            |             |              |             |              |              |
|-----------------------------------------------------------------------------------|--------------------|------------|------------|-------------|--------------|-------------|--------------|--------------|
| <b>IOTERM</b>                                                                     |                    | <b>BS1</b> | <b>BS2</b> | <b>BS2e</b> | <b>BS2sx</b> | <b>BS2p</b> | <b>BS2pe</b> | <b>BS2px</b> |
|  | <b>IOTERM Port</b> |            |            |             |              |             |              |              |

### Function

Switch control to main I/O pins or auxiliary I/O pins (on the BS2p40 only) depending on state of *Port*.

- **Port** is a variable/constant/expression (0 – 1) that specifies which I/O port to use.

### Quick Facts

Table 5.42: IOTERM Quick Facts.

|                         | BS2p, BS2pe, and BS2px                                                                                                       |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <b>Values for Port</b>  | 0 = switch to main I/O group, 1 = switch to auxiliary I/O group.                                                             |
| <b>I/O pin IDs</b>      | 0 – 15 (after IOTERM command, all references affect physical pins 5 – 20 or 21 – 36 depending on state of <i>Port</i> ).     |
| <b>Special Notes</b>    | Both the BS2p and BS2pe 24-pin modules accept this command, however, only the BS2p40 gives access to the auxiliary I/O pins. |
| <b>Related Commands</b> | AUXIO and MAINIO                                                                                                             |

### Explanation

The BS2p, BS2pe and BS2px are available as 24-pin modules that are pin compatible with the BS2, BS2e and BS2sx. Also available is a 40-pin module called the BS2p40, with an additional 16 I/O pins (for a total of 32). The BS2p40's I/O pins are organized into two groups, called main and auxiliary. The I/O pins in each group can be accessed in the same manner (by referencing I/O pins 0 – 15) but access is only possible within one group at a time. The IOTERM command causes the BASIC Stamp to affect either the main or auxiliary I/O pins in all further code until the MAINIO, AUXIO or another IOTERM command is reached, or the BASIC Stamp is reset or power-cycled. The value of *Port* determines which group of I/O pins will be referenced. Using 0 for *Port* will switch to the main I/O group and using 1 for *Port* will switch to the auxiliary group.

A SIMPLE IOTERM EXAMPLE.

The following example illustrates this:

```
HIGH 0 ' make P0 high
IOTERM 1 ' Port = 1, so switch to auxiliary pins
LOW 0 ' make X0 low
```

The first line of the above example will set I/O pin 0 of the main I/O pins (physical pin 5) high. Afterward, the IOTERM command tells the BASIC

## IOTERM – BASIC Stamp Command Reference

Stamp that all commands following it should affect the auxiliary I/O pins (*Port* = 1). The following LOW command will set I/O pin 0 of the auxiliary I/O pins (physical pin 21) low.

Note that the main I/O and auxiliary I/O pins are independent of each other; the states of the main I/O pins remain unchanged while the program affects the auxiliary I/O pins, and vice versa.

MAIN I/O AND AUXILIARY I/O PINS ARE INDEPENDENT AND UNAFFECTED BY CHANGES IN THE OPPOSITE GROUP.

Other commands that affect I/O group access are AUXIO and MAINIO.

### Demo Program (AUX\_MAIN\_TERM.bsp)

```
' AUX_MAIN_TERM.bsp
' This program demonstrates the use of the AUXIO, MAINIO and IOTERM
' commands to affect I/O pins in the auxiliary and main I/O groups.

' {$STAMP BS2p}
' {$PBASIC 2.5}

#SELECT $STAMP
#CASE BS2, BS2E, BS2SX
#ERROR "Program requires BS2p40"
#CASE BS2P, BS2PE, BS2PX
 DEBUG "Note: This program designed for the BS2p40.", CR
#ENDSELECT

port VAR Bit

Main:
DO
 MAINIO ' Switch to main I/O pins
 TOGGLE 0 ' Toggle state of I/O pin P0
 PWM 1, 100, 40 ' Generate PWM on I/O pin P1

 AUXIO ' Switch to auxiliary I/O pins
 TOGGLE 0 ' Toggle state of I/O pin X0
 PULSOUT 1, 1000 ' Generate a pulse on I/O pin X1
 PWM 2, 100, 40 ' Generate PWM on I/O pin X2

 IOTERM port ' Switch to main or aux I/Os
 ' -- depending on port
 TOGGLE 3 ' Toggle state of I/O pin 3
 ' -- on main and aux, alternately
 port = ~port ' Invert port
 PAUSE 1000 ' 1 second delay
LOOP
END
```



NOTE: This example program will tokenize with the 24-pin BS2p, BS2pe, and BS2px but its effects can only be seen on the BS2p40. This program uses conditional compilation techniques; see Chapter 3 for more information.

## 5: BASIC Stamp Command Reference – LCDCMD

### LCDCMD

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|



#### LCDCMD *Pin, Command*

### Function

Send a command to an LCD display.

- **Pin** is a variable/constant/expression (0 – 1 or 8 – 9) that specifies which I/O pins to use. The LCD requires, at most, seven I/O pins to operate. Note that LCDCMD, LCDIN and LCDOUT use a 4-bit interface to the LCD which requires a specific initialization sequence before LCDIN and LCDOUT can be used. Specifics on the initialization sequence will follow. The *Pin* argument serves a double purpose; specifying the first pin and, indirectly, the group of other required pins. See explanation below. All I/O pins will be set to output mode.
- **Command** is a variable/constant/expression (0 – 255) indicating the LCD command to send.

### Quick Facts

Table 5.43: LCDCMD Quick Facts.

|                                               | BS2p, BS2pe, and BS2px                                                                                                                                                     |
|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Values for <i>Pin</i>                         | 0, 1, 8 or 9                                                                                                                                                               |
| I/O Pin Arrangement when <i>Pin</i> is 0 or 1 | 0 or 1 (depending on pin): LCD Enable (E) pin<br>2: LCD Read/Write (R/W) pin<br>3: LCD Register Select (RS) pin<br>4 – 7: LCD Data Buss (DB4 – DB7, respectively) pins     |
| I/O Pin Arrangement when <i>Pin</i> is 8 or 9 | 8 or 9 (depending on pin): LCD Enable (E) pin<br>10: LCD Read/Write (R/W) pin<br>11: LCD Register Select (RS) pin<br>12 – 15: LCD Data Buss (DB4 – DB7, respectively) pins |
| Special Notes                                 | LCDCMD is designed to use the LCD's 4-bit mode only.                                                                                                                       |
| Related Commands                              | LCDIN and LCDOUT                                                                                                                                                           |

### Explanation

NOTE: LCDCMD, LCDIN and LCDOUT use a 4-bit interface to the LCD which requires a specific initialization sequence before LCDIN and LCDOUT can be used; read more below.

The three LCD commands (LCDCMD, LCDIN and LCDOUT) allow the BS2p, BS2pe, and BS2px to interface directly to standard LCD displays that feature a Hitachi 44780 controller (part #HD44780A). This includes many 1 x 16, 2 x 16 and 4 x 20 character LCD displays.

The Hitachi 44780 LCD controller supports a number of special instructions for initializing the display, moving the cursor, changing the default layout, etc. The LCDCMD command is used to send one of these

## LCDCMD – BASIC Stamp Command Reference

instructions to the LCD. It is most commonly used to initialize the display upon a power-up or reset condition.

The following is an example of the LCDCMD command:

A SIMPLE LCDCMD EXAMPLE.

```
LCDOUT 0, 24
```

The above code will send the Scroll Left command (represented by the number 24) to the LCD whose enable pin is connected to I/O pin 0. This will cause the LCD display to scroll, or shift, the entire display one character to the left.

You may have noticed that the *Pin* argument in the example above was 0. The LCDCMD command actually uses more than just this I/O pin, however. The LCDCMD command requires seven I/O pins. This is because the standard LCD displays have a parallel interface, rather than a serial one. The *Pin* argument can be the numbers 0, 1, 8 or 9 and will result in the use of the I/O pins shown in Table 5.43. Figure 5.14 shows the required wiring for the above command to work.

WIRING THE BASIC STAMP TO AN LCD.

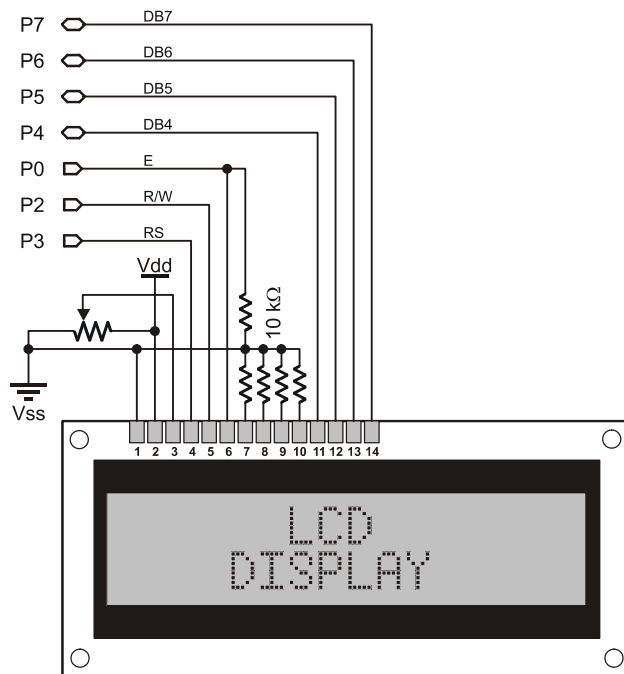


**Figure 5.14:** Example LCD Circuit. Shown with all connections necessary for the LCDCMD, LCDIN and LCDOUT commands.



## 5: BASIC Stamp Command Reference – LCDCMD

INITIALIZING THE LCD; THE MOST  
IMPORTANT STEP!

Note that we could have used 1 for the *Pin* argument and moved the LCD's Enable pin (pin 6) to I/O pin 1. Similarly, using 9 for the *Pin* argument would have required us to wire the LCD's pins to I/O pins 9 through 15, rather than I/O pins 0 and 2 through 7.

When the LCD is first powered-up, it defaults to an 8-bit interface and must be configured for a 4-bit buss before sending commands like the one shown above. This process is known as initializing the LCD and is the first thing your program should do upon starting up. The following code is a good example of LCD initialization.

```
Init_LCD:
 PAUSE 1000 ' allow LCD to self-initialize first
 LCDCMD 0, %00110000 ' send wakeup sequence to LCD
 PAUSE 5 ' pause required by LCD specs
 LCDCMD 0, %00110000
 PAUSE 0 ' pause required by LCD specs
 LCDCMD 0, %00110000
 PAUSE 0 ' pause required by LCD specs
 LCDCMD 0, %00100000 ' set data bus to 4-bit mode
 LCDCMD 0, %00101000 ' set to 2-line mode with 5x8 font
 LCDCMD 0, %00001100 ' display on without cursor
 LCDCMD 0, %00000110 ' auto-increment cursor
```

This initialization code is the most commonly used sequence for a 2 x 16 and 4 x 20 LCD display (the 2-line mode instruction sets the 4 x 20 to 4-line mode). The PAUSE 1000 command is optional, but only if your program takes more than approximately 700 ms before it executes the InitLCD code above. Without it, upon powering your circuit, the BASIC Stamp may talk to the LCD too early, the LCD will then miss some of the commands and the display will operate strangely, or not at all.

Do not change the "wake-up" and "4-bit mode" sequence commands. However, the commands below the line that says, "Set data bus to 4-bit mode" may be modified to set other desired modes.

COMMON LCD COMMANDS.

Table 5.44 shows the most commonly used LCD commands. Here's an example:

```
LCDCMD 0, 16
```

This will make the LCD's cursor move left by one character (16 is the Cursor Left command), even if the cursor is not visible. The next character

## LCDCMD – BASIC Stamp Command Reference

printed on the display (with the LCDOUT command) will appear at the current cursor's location. Here's another example:

```
LCDCMD 0, 128 + 64
```

The above command will move the cursor to the first character position on the second line (on a 2 x 16 display). 128 is the Move To Display Address command and 64 is the location number. See the "Character Positioning" section, below, for more information.

|                                  | Command<br>(in decimal) | Description                                             |
|----------------------------------|-------------------------|---------------------------------------------------------|
| <b>Do Nothing</b>                | 0                       | Don't perform any special operation.                    |
| <b>Clear Display</b>             | 1                       | Clear the display and move cursor to home position.     |
| <b>Home Display</b>              | 2                       | Move cursor and display to home position.               |
| <b>Inc Cursor</b>                | 6                       | Set cursor direction to right, without a display shift. |
| <b>Display Off</b>               | 8                       | Turn off display (display data is retained).            |
| <b>Display On</b>                | 12                      | Turn on display without cursor (display is restored).   |
| <b>Blinking Cursor</b>           | 13                      | Turn on display with blinking cursor.                   |
| <b>Underline Cursor</b>          | 14                      | Turn on display with underline cursor.                  |
| <b>Cursor Left</b>               | 16                      | Move cursor left one character.                         |
| <b>Cursor Right</b>              | 20                      | Move cursor right one character.                        |
| <b>Scroll Left</b>               | 24                      | Scroll display left one character.                      |
| <b>Scroll Right</b>              | 28                      | Scroll display right one character.                     |
| <b>Move To CGRAM<br/>Address</b> | 64 + address            | Move pointer to character RAM location.                 |
| <b>Move To DDRAM<br/>Address</b> | 128 + address           | Move cursor to Display Data RAM location.               |

**Table 5.44:** Common LCD Commands. These are supported by LCDs with the Hitachi 44780 controller.

While most users will only need the commands shown in Table 5.44 above, Table 5.45 below details all of the instructions supported by the LCD (for advanced users). Many instructions are multipurpose, depending on the state of special bits. Clever manipulation of the instruction bits will allow for powerful control of the LCD.

A NOTE ABOUT ADVANCED LCD COMMANDS.

The last command shown above (Move To DDRAM Address) is used to move the cursor to a specific position on the LCD. The LCD's DDRAM (Display Data RAM) is a fixed size with a unique position number for each character cell. The viewable portion of the DDRAM depends on the LCD's logical view position (which can be altered with the Scroll Display command). The default view position is called the Home position; it means that the display's upper left character corresponds to DDRAM

CHARACTER POSITIONING: MOVING THE CURSOR.

## 5: BASIC Stamp Command Reference – LCDCMD

location 0. Figure 5.15 indicates the position numbers for characters on the LCD screen.

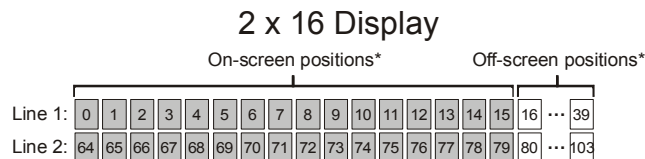
**Table 5.45:** All LCD Commands (for advanced users). These are supported by LCDs with the Hitachi 44780 controller.

|                                      | Command Code (in binary) |   |   |   |   |   |   |   | Description                                                                                                       |
|--------------------------------------|--------------------------|---|---|---|---|---|---|---|-------------------------------------------------------------------------------------------------------------------|
|                                      | 7                        | 6 | 5 | 4 | 3 | 2 | 1 | 0 |                                                                                                                   |
| <b>Clear Display</b>                 | 0                        | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Clear entire display and move cursor home (address 0).                                                            |
| <b>Home Display</b>                  | 0                        | 0 | 0 | 0 | 0 | 0 | 1 | 0 | Move cursor home and return display to home position.                                                             |
| <b>Entry Mode</b>                    | 0                        | 0 | 0 | 0 | 0 | 1 | M | S | Sets cursor direction (M: 0=left, 1=right) and display scrolling (S: 0=no scroll, 1=scroll)                       |
| <b>Display/Cursor</b>                | 0                        | 0 | 0 | 0 | 1 | D | U | B | Sets display on/off (D), underline cursor (U) and blinking block cursor (B). (0=off, 1=on)                        |
| <b>Scroll Display / Shift Cursor</b> | 0                        | 0 | 0 | 1 | C | M | 0 | 0 | Shifts display or cursor (C: 0=cursor, 1=display) left or right (M: 0=left, 1=right).                             |
| <b>Function Set</b>                  | 0                        | 0 | 1 | B | L | F | 0 | 0 | Sets buss size (B: 0=4-bits, 1=8-bits), number of lines (L: 0=1-line, 1=2-lines) and font size (F: 0=5x8, 1=5x10) |
| <b>Move To CGRAM Address</b>         | 0                        | 1 | A | A | A | A | A | A | Move pointer to character RAM location specified by address (A)                                                   |
| <b>Move To DDRAM Address</b>         | 1                        | A | A | A | A | A | A | A | Move cursor to display RAM location specified by address (A)                                                      |

Note that Figure 5.15 shows the most common DDRAM mapping, though some LCD's may have organized the DDRAM differently. A little experimentation with your LCD may reveal this.

**Figure 5.15:** LCD Character Positions.

NOTE: Many 1 x 16 displays conform to the position numbers shown on Line 1 of the 2 x 16 display.



\*Assuming the display is in the home position.

## ***LCDCMD – BASIC Stamp Command Reference***

---

On a standard 2 x 16 character display, the following command would move the cursor to the third column of the second line:

```
LCDCMD 0, 128 + 66
```

The number 128 tells the LCD we wish to move the cursor and 66 is the location number of the desired position. Similarly, sending just 128 (128 + 0) would move the cursor to the first character of the first line (the upper left character if the display is at the home position).

You may have noticed that the 2 x 16 display has many locations that are not visible; they are beyond the right edge of the screen. These locations (16 – 39 and 80 to 103) become important for scrolling operations. For example, it is possible to move the cursor to location 16, print some text there and then issue a number of Scroll Left instructions (LCDCMD 0, 24) to slowly scroll the text onto the display from right to left. If you did so, the DDRAM positions that were on the left of the screen would now be past the left edge of the screen. For example,

SCROLLING THE DISPLAY.

```
LCDCMD 0, 24
LCDCMD 0, 24
```

would cause the screen to scroll to the left by two characters. At this point, the upper-left character in the display would actually be DDRAM location 2 and the lower-left character would be DDRAM location 66. Locations 0, 1, 64 and 65 would be off the left edge of the LCD and would no longer be visible. Some interesting effects can be achieved by taking advantage of this feature.

The 4 x 20 LCD has a strange DDRAM map. The upper-right character is location 19 and the next location, 20, appears as the first character of the third line. This strange mapping is due to constraints in the LCD controller and the manufacturers design, and unfortunately makes the scrolling features virtually useless on the 4 x 20 displays.

NOTES ON DDRAM MAPPING FOR 4 X 20 LCDs.

Even though the LCD requires many pins to talk to it, only the Enable pin needs to remain dedicated to the LCD and all the other pins can be multiplexed (shared) with certain other devices (if wired carefully). In addition, the I/O pin connected to the LCD's R/W pin is only necessary if the LCDIN command will be used in the application. If the LCDIN command will not be used, LCD pin 5 (R/W pin) can be connected to

DETAILS ON LCD WIRING.

## 5: BASIC Stamp Command Reference – LCDCMD

ground, the 4 resistors on LCD pins 7-10 may be removed (connecting pins 7-10 directly to ground). Note that even though this change will leave BASIC Stamp I/O pin 2 disconnected, it will still be set to output mode for each LCDCMD and LCDOUT command executed.



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the \$STAMP directive accordingly.

### Demo Program (LCDINIT.bsp)

```
' LCDINIT.bsp
' This program demonstrates initialization and printing on a 2 x 16
' character LCD display. The set of "LCD constants", below, are provided
' as pre-defined and useful LCD commands, though not all are actually
' used in this program.

' {$STAMP BS2p}
' {$PBASIC 2.5}

#IF ($STAMP < BS2P) #THEN
 #ERROR "Program requires BS2p, BS2pe or BS2px."
#ENDIF

Lcd PIN 0

LcdCls CON $01 ' clear the LCD
LcdHome CON $02 ' move cursor home
LcdCrsrL CON $10 ' move cursor left
LcdCrsrR CON $14 ' move cursor right
LcdDispL CON $18 ' shift chars left
LcdDispR CON $1C ' shift chars right
LcdDDRam CON $80 ' Display Data RAM
LcdCGRam CON $40 ' Character Generator RAM
LcdLine1 CON $80 ' DDRAM address of line 1
LcdLine2 CON $C0 ' DDRAM address of line 2

Init_LCD:
 PAUSE 1000 ' allow LCD to self-initialize first
 LCDCMD Lcd, %00110000 ' send wakeup sequence to LCD
 PAUSE 5 ' pause required by LCD specs
 LCDCMD Lcd, %00110000
 PAUSE 0 ' pause required by LCD specs
 LCDCMD Lcd, %00110000
 PAUSE 0 ' pause required by LCD specs
 LCDCMD Lcd, %00100000 ' set data bus to 4-bit mode
 LCDCMD Lcd, %00101000 ' set to 2-line mode with 5x8 font
 LCDCMD Lcd, %00001100 ' display on without cursor
 LCDCMD Lcd, %00000110 ' auto-increment cursor

Main:
DO
 LCDOUT Lcd, LcdCls, ["Hello, World!"]
 LCDOUT Lcd, LcdLine2, ["How are you?"]
```

## ***LCDCMD – BASIC Stamp Command Reference***

---

```
PAUSE 3000
LCDCMD Lcd, LcdCls
PAUSE 500
LOOP
END
```

## 5: BASIC Stamp Command Reference – LCDIN

### LCDIN

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**LCDIN** *Pin, Command, [ InputData ]*

#### Function

Receive data from an LCD display.

- **Pin** is a variable/constant/expression (0 – 1 or 8 – 9) that specifies which I/O pins to use. The LCD requires, at most, seven I/O pins to operate. The *Pin* argument serves a double purpose; specifying the first pin and, indirectly, the group of other required pins. See explanation below. All I/O pins will be set to output mode initially and the upper I/O pins (4 – 7 or 12 – 15) will be set to input mode by the end of the LCDIN command.
- **Command** is a variable/constant/expression (0 – 255) indicating the LCD command to send.
- **InputData** is a list of variables and formatters that tells LCDIN what to do with incoming data. LCDIN can store data in a variable or array, interpret numeric text (decimal, binary, or hex) and store the corresponding value in a variable, wait for a fixed or variable sequence of bytes, or ignore a specified number of bytes. These actions can be combined in any order in the *InputData* list.

#### Quick Facts

Table 5.46: LCDIN Quick Facts.

|                                               | BS2p, BS2pe, and BS2px                                                                                                                                                     |
|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Values for <i>Pin</i>                         | 0, 1, 8 or 9                                                                                                                                                               |
| I/O Pin Arrangement when <i>Pin</i> is 0 or 1 | 0 or 1 (depending on pin): LCD Enable (E) pin<br>2: LCD Read/Write (R/W) pin<br>3: LCD Register Select (RS) pin<br>4 – 7: LCD Data Buss (DB4 – DB7, respectively) pins     |
| I/O Pin Arrangement when <i>Pin</i> is 8 or 9 | 8 or 9 (depending on pin): LCD Enable (E) pin<br>10: LCD Read/Write (R/W) pin<br>11: LCD Register Select (RS) pin<br>12 – 15: LCD Data Buss (DB4 – DB7, respectively) pins |
| Special Notes                                 | LCDIN is designed to use the LCD's 4-bit mode only.                                                                                                                        |
| Related Commands                              | LCDCMD and LCDOUT                                                                                                                                                          |

## LCDIN – BASIC Stamp Command Reference

---

### Explanation

The three LCD commands (LCDCMD, LCDIN and LCDOUT) allow the BS2p, BS2pe, and BS2px to interface directly to standard LCD displays that feature a Hitachi 44780 controller (part #HD44780A). This includes many 1 x 16, 2 x 16 and 4 x 20 character LCD displays.

NOTE: LCDCMD, LCDIN and LCDOUT use a 4-bit interface to the LCD which requires a specific initialization sequence before LCDIN and LCDOUT can be used; read more below.

The LCDIN command is used to send one instruction and then receive at least one data byte from the LCD's Character Generator RAM or Display Data RAM. The following is an example of the LCDIN command:

A SIMPLE LCDIN EXAMPLE.

```
char VAR Byte
LCDIN 0, 128, [char]
```

The above code will read the character value at location 0 of the DDRAM. See the "Character Positioning" section, of the LCDCMD command description on page 252 for more information.

The LCDIN command actually uses more than just the I/O pin specified by the *Pin* argument. The LCDIN command requires seven I/O pins. This is because the standard LCD displays have a parallel interface, rather than a serial one. The *Pin* argument can be the numbers 0, 1, 8 or 9 and will result in the use of the I/O pins shown in Table 5.46. Please refer to the LCDCMD command description for information on properly wiring the LCD display.

TWO VERY IMPORTANT STEPS:  
1) WIRING THE BASIC STAMP TO AN LCD.  
2) INITIALIZING THE LCD.

When the LCD is first powered-up, it defaults to an 8-bit interface and must be properly configured for a 4-bit buss before sending commands like the one shown above. This process is known as initializing the LCD and is the first thing your program should do upon starting up. Please refer to the LCDCMD command description for information on properly initializing the LCD display.

The LCDIN command's *InputData* argument is similar to the SERIN command's *InputData* argument. This means data can be received as ASCII character values, decimal, hexadecimal and binary translations and string data as in the examples below (assume the LCD display has "Value: 3A:101" starting at the first character of the first line on the screen).

RECEIVING FORMATTED DATA.



## 5: BASIC Stamp Command Reference – LCDIN

```

value VAR Byte(13)
LCDIN 0, 128, [value] 'receive the ASCII value for "V"
LCDIN 0, 128, [DEC value] 'receive the number 3.
LCDIN 0, 128, [HEX value] 'receive the number $3A.
LCDIN 0, 128, [BIN value] 'receive the number %101.
LCDIN 0, 128, [STR value\13] 'receive the string "Value: 3A:101"

```

Table 5.47 and Table 5.48 list all the special formatters and conversion formatters available to the LCDIN command. See the SERIN command for additional information and examples of their use.

Some possible uses of the LCDIN command are 1) in combination with the LCDOUT command to store and read data from the unused DDRAM or CGRAM locations (as extra variable space), 2) to verify that the data from a previous LCDOUT command was received and processed properly by the LCD, and 3) to read character data from CGRAM for the purposes of modifying it and storing it as a custom character.

**Table 5.47:** LCDIN Special Formatters.

| Special Formatter            | Action                                                                                                                                                                                                                              |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SPSTR L                      | Input a character string of length L bytes (up to 126) into Scratch Pad RAM, starting at location 0. Use GET to retrieve the characters.                                                                                            |
| STR <i>ByteArray</i> \L {E}  | Input a character string of length L into an array. If specified, an end character E causes the string input to end before reaching length L. Remaining bytes are filled with 0s (zeros).                                           |
| WAIT ( <i>Value</i> )        | Wait for a sequence of bytes specified by value. Value can be numbers separated by commas or quoted text (ex: 65, 66, 67 or "ABC"). The WAIT formatter is limited to a maximum of six characters.                                   |
| WAITSTR <i>ByteArray</i> {L} | Wait for a sequence of bytes matching a string stored in an array variable, optionally limited to L characters. If the optional L argument is left off, the end of the array-string must be marked by a byte containing a zero (0). |
| SKIP <i>Length</i>           | Ignore <i>Length</i> bytes of characters.                                                                                                                                                                                           |

## LCDIN – BASIC Stamp Command Reference

| Conversion Formatter | Type of Number                                                                                         | Numeric Characters Accepted        | Notes   |
|----------------------|--------------------------------------------------------------------------------------------------------|------------------------------------|---------|
| DEC{1..5}            | Decimal, optionally limited to 1 – 5 digits                                                            | 0 through 9                        | 1       |
| SDEC{1..5}           | Signed decimal, optionally limited to 1 – 5 digits                                                     | -, 0 through 9                     | 1,2     |
| HEX{1..4}            | Hexadecimal, optionally limited to 1 – 4 digits                                                        | 0 through 9, A through F           | 1,3,5   |
| SHEX{1..4}           | Signed hexadecimal, optionally limited to 1 – 4 digits                                                 | -, 0 through 9, A through F        | 1,2,3   |
| IHEX{1..4}           | Indicated hexadecimal, optionally limited to 1 – 4 digits                                              | \$, 0 through 9, A through F       | 1,3,4   |
| ISHEX{1..4}          | Signed, indicated hexadecimal, optionally limited to 1 – 4 digits                                      | -, \$, 0 through 9, A through F    | 1,2,3,4 |
| BIN{1..16}           | Binary, optionally limited to 1 – 16 digits                                                            | 0, 1                               | 1       |
| SBIN{1..16}          | Signed binary, optionally limited to 1 – 16 digits                                                     | -, 0, 1                            | 1,2     |
| IBIN{1..16}          | Indicated binary, optionally limited to 1 – 16 digits                                                  | %, 0, 1                            | 1,4     |
| ISBIN{1..16}         | Signed, indicated binary, optionally limited to 1 – 16 digits                                          | -, %, 0, 1                         | 1,2,4   |
| NUM                  | Generic numeric input (decimal, hexadecimal or binary); hexadecimal or binary number must be indicated | \$, %, 0 through 9, A through F    | 1, 3, 4 |
| SNUM                 | Similar to NUM with value treated as signed with range -32768 to +32767                                | -, \$, %, 0 through 9, A through F | 1,2,3,4 |

**Table 5.48:** LCDIN Conversion Formatters

- 1 All numeric conversions will continue to accept new data until receiving either the specified number of digits (ex: three digits for DEC3) or a non-numeric character.
- 2 To be recognized as part of a number, the minus sign (-) must immediately precede a numeric character. The minus sign character occurring in non-numeric text is ignored and any character (including a space) between a minus and a number causes the minus to be ignored.
- 3 The hexadecimal formatters are not case-sensitive; “a” through “f” means the same as “A” through “F”.
- 4 Indicated hexadecimal and binary formatters ignore all characters, even valid numerics, until they receive the appropriate prefix (\$ for hexadecimal, % for binary). The indicated formatters can differentiate between text and hexadecimal (ex: ABC would be interpreted by HEX as a number but IHEX would ignore it unless expressed as \$ABC). Likewise, the binary version can distinguish the decimal number 10 from the binary number %10. A prefix occurring in non-numeric text is ignored, and any character (including a space) between a prefix and a number causes the prefix to be ignored. Indicated, signed formatters require that the minus sign come before the prefix, as in -\$1B45.
- 5 The HEX modifier can be used for Decimal to BCD Conversion. See “Hex to BCD Conversion” on page 97.

For examples of all conversion formatters and how they process incoming data see Appendix C.

## 5: BASIC Stamp Command Reference – LCDIN



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

### Demo Program (LCDIN.bsp)

```
' LCDIN.bsp
' This program demonstrates initialization, printing and reading
' from a 2 x 16 character LCD display.

' {$STAMP BS2p}
' {$PBASIC 2.5}

#IF ($STAMP < BS2P) #THEN
 #ERROR "Program requires BS2p, BS2pe or BS2px."
#ENDIF

Lcd PIN 0

LcdCls CON $01 ' clear the LCD
LcdHome CON $02 ' move cursor home
LcdCrsrL CON $10 ' move cursor left
LcdCrsrR CON $14 ' move cursor right
LcdDispL CON $18 ' shift chars left
LcdDispR CON $1C ' shift chars right
LcdDDRam CON $80 ' Display Data RAM
LcdCGRam CON $40 ' Character Generator RAM
LcdLine1 CON $80 ' DDRAM address of line 1
LcdLine2 CON $C0 ' DDRAM address of line 2

char VAR Byte(16)

Init_LCD:
 PAUSE 1000 ' allow LCD to self-initialize first
 LCDCMD Lcd, %00110000 ' send wakeup sequence to LCD
 PAUSE 5 ' pause required by LCD specs
 LCDCMD Lcd, %00110000
 PAUSE 0 ' pause required by LCD specs
 LCDCMD Lcd, %00110000
 PAUSE 0 ' pause required by LCD specs
 LCDCMD Lcd, %00100000 ' set data bus to 4-bit mode
 LCDCMD Lcd, %00101000 ' set to 2-line mode with 5x8 font
 LCDCMD Lcd, %00001100 ' display on without cursor
 LCDCMD Lcd, %00000110 ' auto-increment cursor

Main:
 DO
 LCDOUT Lcd, LcdCls, ["Hello!"]
 GOSUB Read_LCD_Screen
 PAUSE 3000
 LCDOUT Lcd, LcdCls, ["I'm a 2x16 LCD!"]
 GOSUB Read_LCD_Screen
 PAUSE 3000
 LOOP
```

## ***LCDIN – BASIC Stamp Command Reference***

---

```
Read_LCD_Screen:
 DEBUG "LCD now says: "
 LCDIN Lcd, LcdLine1, [STR char\16]
 DEBUG STR char\16, CR
 RETURN
```

## 5: BASIC Stamp Command Reference – LCDOUT

### LCDOUT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**LCDOUT** *Pin, Command, [ OutputData ]*

#### Function

Send data to an LCD display.

- **Pin** is a variable/constant/expression (0 – 1 or 8 – 9) that specifies which I/O pins to use. The LCD requires, at most, seven I/O pins to operate. The *Pin* argument serves a double purpose; specifying the first pin and, indirectly, the group of other required pins. See explanation below. All I/O pins will be set to output mode.
- **Command** is a variable/constant/expression (0 – 255) indicating an LCD command to send.
- **OutputData** is a list of variables, constants, expressions and formatters that tells LCDOUT how to format outgoing data. LCDOUT can transmit individual or repeating bytes, convert values into decimal, hex or binary text representations, or transmit strings of bytes from variable arrays. These actions can be combined in any order in the *OutputData* list.

#### Quick Facts

Table 5.49: LCDOUT Quick Facts.

|                                               | BS2p, BS2pe, and BS2px                                                                                                                                                     |
|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Values for <i>Pin</i>                         | 0, 1, 8 or 9                                                                                                                                                               |
| I/O Pin Arrangement when <i>Pin</i> is 0 or 1 | 0 or 1 (depending on pin): LCD Enable (E) pin<br>2: LCD Read/Write (R/W) pin<br>3: LCD Register Select (RS) pin<br>4 – 7: LCD Data Buss (DB4 – DB7, respectively) pins     |
| I/O Pin Arrangement when <i>Pin</i> is 8 or 9 | 8 or 9 (depending on pin): LCD Enable (E) pin<br>10: LCD Read/Write (R/W) pin<br>11: LCD Register Select (RS) pin<br>12 – 15: LCD Data Buss (DB4 – DB7, respectively) pins |
| Special Notes                                 | LCDOUT is designed to use the LCD's 4-bit mode only.                                                                                                                       |
| Related Commands                              | LCDCMD and LCDIN                                                                                                                                                           |

#### Explanation

NOTE: LCDCMD, LCDIN and LCDOUT use a 4-bit interface to the LCD which requires a specific initialization sequence before LCDIN and LCDOUT can be used; read more below.

The three LCD commands (LCDCMD, LCDIN and LCDOUT) allow the BS2p, BS2pe, and BS2px to interface directly to standard LCD displays that feature a Hitachi 44780 controller (part #HD44780A). This includes many 1 x 16, 2 x 16 and 4 x 20 character LCD displays.

## LCDOUT – BASIC Stamp Command Reference

---

The LCDOUT command is used to send one instruction followed by at least one data byte to the LCD. The data that is output is written to the LCD's Character Generator RAM or Display Data RAM. The following is an example of the LCDOUT command:

```
LCDOUT 0, 1, ["Hello World!"]
```

The above code will clear the LCD screen and then send "Hello World!" to the screen. The first argument (0) is the starting I/O pin number and the second argument (1) is the LCD's instruction for Clear Screen.

The LCDOUT command actually uses more than just the I/O pin specified by the *Pin* argument. The LCDOUT command requires seven I/O pins. This is because the standard LCD displays have a parallel interface, rather than a serial one. The *Pin* argument can be the numbers 0, 1, 8 or 9 and will result in the use of the I/O pins shown in Table 5.49. Please refer to the LCDCMD command description for information on properly wiring the LCD display.

TWO VERY IMPORTANT STEPS:  
1) WIRING THE BASIC STAMP TO AN LCD.  
2) INITIALIZING THE LCD.

When the LCD is first powered-up, it defaults to an 8-bit interface and must be properly configured for a 4-bit buss before sending commands like the one shown above. This process is known as initializing the LCD and is the first thing your program should do upon starting up. Please refer to the LCDCMD command description for information on properly initializing the LCD display.

The LCDOUT command's *OutputData* argument is exactly like that of the DEBUG and SEROUT command's *OutputData* argument. This means data can be sent as literal text, ASCII character values, repetitive values, decimal, hexadecimal and binary translations and string data as in the examples below.

```
value VAR Byte
```

```
value = 65
```

```
LCDOUT 0, 0, [value] ' send "A"
LCDOUT 0, 0, [REP value\5] ' send "AAAAA"
LCDOUT 0, 0, [DEC value] ' send "6" and "5"
LCDOUT 0, 0, [HEX value] ' send "4" and "1"
LCDOUT 0, 0, [BIN value] ' send "1000001"
```

SENDING AND FORMATTING DATA.

## 5: BASIC Stamp Command Reference – LCDOUT

Table 5.50 and Table 5.51 list all the available conversion formatters and special formatters available to the LCDOUT command. See the DEBUG and SEROUT commands for additional information and examples of their use.

**Table 5.50:** LCDOUT Conversion Formatters.

| Conversion Formatter | Type of Number                                                              | Notes |
|----------------------|-----------------------------------------------------------------------------|-------|
| DEC{1..5}            | Decimal, optionally fixed to 1 – 5 digits                                   | 1     |
| SDEC{1..5}           | Signed decimal, optionally fixed to 1 – 5 digits                            | 1,2   |
| HEX{1..4}            | Hexadecimal, optionally fixed to 1 – 4 digits                               | 1,3   |
| SHEX{1..4}           | Signed hexadecimal, optionally fixed to 1 – 4 digits                        | 1,2   |
| IHEX{1..4}           | Indicated hexadecimal, optionally fixed to 1 – 4 digits (\$ prefix)         | 1     |
| ISHEX{1..4}          | Signed, indicated hexadecimal, optionally fixed to 1 – 4 digits (\$ prefix) | 1,2   |
| BIN{1..16}           | Binary, optionally fixed to 1 – 16 digits                                   | 1     |
| SBIN{1..16}          | Signed binary, optionally fixed to 1 – 16 digits                            | 1,2   |
| IBIN{1..16}          | Indicated binary, optionally fixed to 1 – 16 digits (% prefix)              | 1     |
| ISBIN{1..16}         | Signed, indicated binary, optionally fixed to 1 – 16 digits (% prefix)      | 1,2   |

- 1 Fixed-digit formatters like DEC4 will pad the number with leading 0s if necessary; ex: DEC4 65 sends 0065. If a number is larger than the specified number of digits, the leading digits will be dropped; ex: DEC4 56422 sends 6422.
- 2 Signed modifiers work under two's complement rules.
- 3 The HEX modifier can be used for BCD to Decimal Conversion. See "Hex to BCD Conversion" on page 97.

**Table 5.51:** LCDOUT Special Formatters.

| Special Formatter                 | Action                                                                                                                                                                                                                |
|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ?                                 | Displays "symbol = x' + carriage return; where x is a number. Default format is decimal, but may be combined with conversion formatters (ex: BIN ? x to display "x = binary_number").                                 |
| ASC ?                             | Displays "symbol = 'x'" + carriage return; where x is an ASCII character.                                                                                                                                             |
| STR <i>ByteArray</i> { <i>L</i> } | Send character string from an array. The optional \L argument can be used to limit the output to L characters, otherwise, characters will be sent up to the first byte equal to 0 or the end of RAM space is reached. |
| REP <i>Byte</i> \L                | Send a string consisting of <i>Byte</i> repeated L times (ex: REP "X"\10 sends "XXXXXXXXXX").                                                                                                                         |

USING THE *COMMAND* ARGUMENT.

The *Command* argument is useful for proceeding a set of data with a special LCD instruction. For example, the code below will move the cursor to location 64 (the first character on the second line) and print "Hi":

```
LCDOUT 0, 128 + 64, ["Hi"]
```

The next example, below, will turn on the blinking block cursor and print "Yo!":

# LCDOUT – BASIC Stamp Command Reference

```
LCDOUT 0, 13, ["Yo!"]
```

Most of the time, you will want to send data without preceding it with a command. To do this, simply use 0 for the *Command* argument, as in:

```
LCDOUT 0, 0, ["Hello there!"]
```

Another use for the LCDOUT command is to access and create custom characters. The Hitachi 44780 controller has a built-in character set that is similar to the ASCII character set (at least for the first 128 characters). Most of these characters are stored in ROM and are not changeable, however, the first eight characters (ASCII 0 through 7) are programmable.

CREATING CUSTOM CHARACTERS.

Each of the programmable characters is five pixels wide and eight pixels tall. It takes eight bytes to describe each character; one byte per row (the left-most three bits are ignored). For example, the character at ASCII location 0 is defined by the bit patterns stored in bytes 0 through 7 of Character Generator RAM (CGRAM). The character at ASCII location 1 is defined by the bit patterns stored in bytes 8 through 15 of CGRAM, and so on.

To create a custom character, use some graph paper to plot out the bit pattern (on and off pixels) in a 5 x 8 pattern, as shown in Figure 5.16. Then calculate the corresponding binary value of the bit pattern for each of the eight rows of character data.

Character Cell Structure and Data

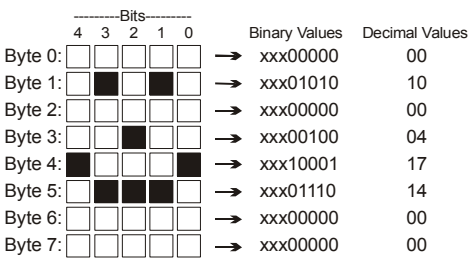


Figure 5.16: LCD Character Structure.

After the data is calculated for each character (8 byte values per character), use the LCDOUT command with the "Move To CGRAM Address" instruction to insert the data into the character's CGRAM locations. For example, the code below will store the character shown in Figure 5.16 into



## 5: BASIC Stamp Command Reference – LCDOUT

character 0's CGRAM data locations. Then it will place the cursor back on the display (DDRAM) and print the character on the screen.

```
LCDOUT 0, 64+0, [00, 10, 00, 04, 17, 14, 00, 00]
LCDOUT 0, 128+0, ["Custom Char: ", 0]
```

The number 64 in the *Command* argument is the LCD's "Move to CGRAM Address" instruction and the 0 that is added to it is the location of the first row of data for the character 0. The LCDOUT command will write the first *OutputData* value (00) to this location, the second *OutputData* value (10) to location 1, etc. If we wanted this custom character to affect character 1, instead of 0, we'd have to adjust the value of the "Move To..." command, i.e.: 64+8. To affect character 2, we'd use 64+16.

To try the example above, don't forget to execute the LCD initialization code (shown in the LDCMD description) first and never forget to move the cursor back to the screen (as with the last command, above) when you're done writing the character data to CGRAM.



### Demo Program (LCDOUT.bsp)

NOTE: This example program can be used with the BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

```
' LCDOUT.bsp
' This program demonstrates initialization and printing on a 2x16
' character LCD display.

' {$STAMP BS2p}
' {$PBASIC 2.5}

#IF ($STAMP < BS2P) #THEN
 #ERROR "Program requires BS2p, BS2pe or BS2px."
#ENDIF

Lcd PIN 0

LcdCls CON $01 ' clear the LCD
LcdHome CON $02 ' move cursor home
LcdCrsrL CON $10 ' move cursor left
LcdCrsrR CON $14 ' move cursor right
LcdDispL CON $18 ' shift chars left
LcdDispR CON $1C ' shift chars right
LcdDDRam CON $80 ' Display Data RAM
LcdCGRam CON $40 ' Character Generator RAM
LcdLine1 CON $80 ' DDRAM address of line 1
LcdLine2 CON $C0 ' DDRAM address of line 2

Init_LCD:
 PAUSE 1000 ' allow LCD to self-initialize first
```

## ***LCDOUT – BASIC Stamp Command Reference***

---

```
LDCMD Lcd, %00110000 ' send wakeup sequence to LCD
PAUSE 5 ' pause required by LCD specs
LDCMD Lcd, %00110000
PAUSE 0 ' pause required by LCD specs
LDCMD Lcd, %00110000
PAUSE 0 ' pause required by LCD specs
LDCMD Lcd, %00100000 ' set data bus to 4-bit mode
LDCMD Lcd, %00101000 ' set to 2-line mode with 5x8 font
LDCMD Lcd, %00001100 ' display on without cursor
LDCMD Lcd, %00000110 ' auto-increment cursor

LCDOUT Lcd, LcdCGRam, ' load custom character map
 [$00, $0A, $0A, $00, $11, $0E, $06, $00]
```

Main:

```
DO
 LCDOUT Lcd, LcdCls, ["Hello my friend."]
 PAUSE 750
 LCDOUT Lcd, LcdLine2, ["How are you?"]
 PAUSE 1500
 LDCMD Lcd, LcdCls
 LCDOUT Lcd, LcdLine1 + 1, ["I'm doing just"]
 LCDOUT Lcd, LcdLine2 + 4, ["fine! ", 0]
 PAUSE 2000
LOOP
END
```

## 5: BASIC Stamp Command Reference – LET

### LET

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|

 { LET } *Variable* = *Value*

### Function

Sets *Variable* equal to the value of *Value*.

- **Value** is a variable/constant/expression. (0-65535).
- **Variable** is a variable which will be set equal to *Value*.

### Explanation

LET is an optional instruction for the BASIC Stamp 1 that can be used with variable assignment statements, such as `A = 5` and `B = A + 2`, etc. This instruction is not required and only exists on the BASIC Stamp 1. LET was a commonly used command in early forms of BASIC, and was originally included in the BS1 command set to accommodate programmers from that generation. Parallax recommends that all new BASIC Stamp 1 programs use assignment statements without the LET command.

### Demo Program (LET.bs1)

```
' LET.bs1
' This example demonstrates the use of LET in assignment statements, which
' is optional and not recommended. Note that the Lunchtime and
' Dinnertime routines do essentially the same thing, but the Dinnertime
' method is recommended(even though the pricing scheme is not!)

' {$PBASIC 1.0}
' {$STAMP BS1}

SYMBOL salad = B1
SYMBOL bread = B2
SYMBOL soup = B3
SYMBOL lunch = B4
SYMBOL dinner = B5

Lunchtime:
 LET salad = 3
 LET bread = 1
 LET soup = 4
 LUNCH = salad + bread + soup
 DEBUG "lunch = $", #lunch, "plus local tax.", CR

Dinnertime:
 salad = 4
 bread = 2
 soup = 5
```

## ***LET – BASIC Stamp Command Reference***

---

```
dinner = salad + bread + soup
DEBUG "Dinner = $",# lunch, "plus local tax.", CR
```

## 5: BASIC Stamp Command Reference – LOOKDOWN

### LOOKDOWN

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**LOOKDOWN** *Target*, ( *Value0*, *Value1*, ...*ValueN* ), *Variable*



**LOOKDOWN** *Target*, { *ComparisonOp* } [ *Value0*, *Value1*, ...*ValueN* ], *Variable*

### Function

Compare *Target* value to a list of values and store the index number of the first value that matches into *Variable*. If no value in the list matches, *Variable* is left unaffected. On all BS2 models, the optional *ComparisonOp* is used as criteria for the match; the default criteria is "equal to."



NOTE: Expressions are not allowed as arguments on the BS1.



- **Target** is a variable/constant/expression (0 – 65535) to be compared to the values in the list.
- **ComparisonOp** is an optional comparison operator (as described in Table 5.53) to be used as the criteria when comparing values. When no *ComparisonOp* is specified, equal to (=) is assumed. This argument is not available on the BS1.
- **Values** are variables/constants/expressions (0 – 65535) to be compared to *Target*.
- **Variable** is a variable (usually a byte) that will be set to the index (0 – 255) of the matching value in the *Values* list. If no matching value is found, *Variable* is left unaffected.

### Quick Facts

Table 5.52: LOOKDOWN Quick Facts.

|                                    | BS1 and all BS2 Models      |
|------------------------------------|-----------------------------|
| Limit of <i>Value</i> Entries      | 256                         |
| Starting Index Number              | 0                           |
| If value list contains no match... | Variable is left unaffected |
| Related Command                    | LOOKUP                      |

### Explanation

LOOKDOWN works like the index in a book. In an index, you search for a topic and get the page number. LOOKDOWN searches for a target value in a list, and stores the index number of the first match in a variable. For example:

## LOOKDOWN – BASIC Stamp Command Reference

---

```
SYMBOL value = B0
SYMBOL result = B1
value = 17
result = 15
```



```
LOOKDOWN value, (26, 177, 13, 1, 0, 17, 99), result
DEBUG "Value matches item ", #result, "in list"
```

-- or --

```
value VAR Byte
result VAR Nib
value = 17
result = 15
```



```
LOOKDOWN value, [26, 177, 13, 1, 0, 17, 99], result
DEBUG "Value matches item ", DEC result, " in list"
```

DEBUG prints, “Value matches item 5 in list” because the value (17) matches item 5 of [26, 177, 13, 1, 0, 17, 99]. Note that index numbers count up from 0, not 1; that is, in this list, 26 is item 0.

THE INDEX NUMBER OF THE FIRST ITEM IS 0, NOT 1.

What happens if the value doesn’t match any of the items in the list? Try changing “Value = 17” to “Value = 2.” Since 2 is not on the list, LOOKDOWN leaves *result* unaffected. Since *result* contained 15 before LOOKDOWN executed, DEBUG prints “Value matches item 15 in list.” By strategically setting the initial value of *result*, as we have here, your program can be written to detect when an item was not found in the list.

Don’t forget that text phrases are just lists of byte values, so they too are eligible for LOOKDOWN searches, as in this example:

USING TEXT IN THE VALUE LIST.

```
SYMBOL value = B0
SYMBOL result= B1
```



```
value = "f"
result = 255
```

```
LOOKDOWN value, ("The quick brown fox"), result
DEBUG "Value matches item ", #result, "in list"
```

-- or --

## 5: BASIC Stamp Command Reference – LOOKDOWN



| value  | VAR | Byte |
|--------|-----|------|
| result | VAR | Nib  |

```
value = "f"
result = 255
```

```
LOOKDOWN value, ["The quick brown fox"], result
DEBUG "Value matches item ", DEC result, " in list"
```

DEBUG prints, "Value matches item 16 in list" because the character at index item 16 is "f" in the phrase, "The quick brown fox".

LOOKDOWN CAN USE VARIABLES AND EXPRESSIONS IN THE VALUE LIST.

The examples above show LOOKDOWN working with lists of constants, but it also works with variables and expressions also. Note, however, that expressions are not allowed as argument on the BS1.

USING LOOKDOWN'S COMPARISON OPERATORS.



On all BS2 models, the LOOKDOWN command can also use another criteria (other than "equal to") for its list. All of the examples above use LOOKDOWN's default comparison operator, =, that searches for an exact match. The entire list of *ComparisonOps* is shown in Table 5.53. The "greater than" comparison operator (>) is used in the following example:

| value  | VAR | Byte |
|--------|-----|------|
| result | VAR | Nib  |

```
value = 17
result = 15
```

```
LOOKDOWN value, >[26, 177, 13, 1, 0, 17, 99], result
DEBUG "Value greater than item ", DEC result, " in list"
```

DEBUG prints, "Value greater than item 2 in list" because the first item the value 17 is greater than is 13 (which is item 2 in the list). *Value* is also greater than items 3 and 4, but these are ignored, because LOOKDOWN only cares about the first item that matches the criteria. This can require a certain amount of planning in devising the order of the list. See the demo program below.

WATCH OUT FOR UNSIGNED MATH ERRORS WHEN USING THE COMPARISON OPERATORS.

LOOKDOWN comparison operators (Table 5.53) use unsigned 16-bit math. They will not work correctly with signed numbers, which are represented internally as two's complement (large 16-bit integers). For example, the two's complement representation of -99 is 65437. So although -99 is certainly less than 0, it would appear to be larger than zero

## LOOKDOWN – BASIC Stamp Command Reference

to the LOOKDOWN comparison operators. The bottom line is: Don't use signed numbers with LOOKDOWN comparisons.

| ComparisonOp Symbol | Description                                             |
|---------------------|---------------------------------------------------------|
| =                   | Find the first value Target is equal to                 |
| <>                  | Find the first value Target is not equal to             |
| >                   | Find the first value Target is greater than             |
| <                   | Find the first value Target is less than                |
| >=                  | Find the first value Target is greater than or equal to |
| <=                  | Find the first value Target is less than or equal to    |

**Table 5.53:** LOOKDOWN Comparison Operators.

A common application for LOOKDOWN is to use it in conjunction with the BRANCH, ON...GOTO, or ON...GOSUB commands to create selective jumps based on a simple variable input:

USING LOOKDOWN WITH BRANCH, ON...GOTO OR ON...GOSUB TO JUMP BASED ON VALUES.

```
' {$PBASIC 2.5}
```

All 2

```
cmd VAR Byte
```

```
DO
```

```
 DEBUG CLS, "Enter cmd (SLMH): "
```

```
 DEBUGIN cmd
```

```
 LOOKDOWN cmd, ["SLMH"], cmd
```

```
 ON cmd GOSUB _Stop, _Low, _Medium, _High
```

```
 IF cmd > 3 THEN DEBUG CLS, "Command not in list"
```

```
 PAUSE 2000
```

```
 END
```

```
LOOP
```

```
_Stop:
```

```
 DEBUG CLS, "Stop"
```

```
 RETURN
```

```
_Low:
```

```
 DEBUG CLS, "Low"
```

```
 RETURN
```

```
_Medium:
```

```
 DEBUG CLS, "Medium"
```

```
 RETURN
```

```
_High:
```

```
 DEBUG CLS, "High"
```

```
 RETURN
```

In this example, the program waits for a key. Here's what happens when "M" is pressed and *cmd* contains "M" (ASCII 77). LOOKDOWN finds that this is item 2 of a list of one-character commands and stores 2 into *cmd*. ON...GOSUB then goes to item 2 of its list, which is the program label



## 5: BASIC Stamp Command Reference – LOOKDOWN

“\_Medium” at which point DEBUG prints “Medium” on the PC screen then returns to the main loop. This is a powerful method for interpreting user input, and a lot neater than the alternative IF...THEN instructions.

USING LOOKDOWN WITH LOOKUP TO  
"MAP" NON-CONTIGUOUS SETS OF  
NUMBERS.

Another great use of LOOKDOWN is in combination with LOOKUP to "map" non-contiguous sets of numbers together. For example, you may have an application where certain numbers are received by the BASIC Stamp and, in response, the BASIC Stamp needs to send a specific set of numbers. This may be easy to code if the numbers are contiguous, or follow some known algebraic equations... but what if they don't? The table below shows some sample, non-contiguous inputs and the corresponding outputs the BASIC Stamp needs to respond with:

**Table 5.54:** Non-Contiguous Number Example.

| Each of these values received (inputs): | Needs to result in each of these values sent (outputs): |
|-----------------------------------------|---------------------------------------------------------|
| 5                                       | 16                                                      |
| 14                                      | 17                                                      |
| 1                                       | 18                                                      |
| 43                                      | 24                                                      |
| 26                                      | 10                                                      |
| 22                                      | 12                                                      |
| 30                                      | 11                                                      |

So, if we receive the number 5, we need to output 16. If we received 43, we need to output 24, and so on. These numbers are not contiguous and they don't appear to be derived from any simple algorithm. We can solve this problem with two lines of code, as follows:

```
LOOKDOWN value, [5, 14, 1, 43, 26, 22, 30], value
LOOKUP value, [16, 17, 18, 24, 10, 12, 11], value
```

Assuming our received number is in *value*, the first line (LOOKDOWN) will find the value in the list and store the index of the location that matches back into *value*. (This step "maps" the non-contiguous numbers: 5, 14, 1, etc, to a contiguous set of numbers: 0, 1, 2, etc). The second line (LOOKUP) takes our new *value*, finds the number at that location and stores it back into *value*. If the received value was 14, LOOKDOWN stores 1 into *value* and LOOKUP looks at the value at location 1 and stores 17 in *value*. The number 43 gets mapped to 3, 3 gets mapped to 24, and so on. This is a quick and easy fix for a potentially messy problem!



### Demo Program (LOOKDOWN.bs1)

## LOOKDOWN – BASIC Stamp Command Reference

---

```
' LOOKDOWN.bs1
' This program uses LOOKDOWN followed by LOOKUP to map the numbers:
' 0, 10, 50, 64, 71 and 98 to 35, 40, 58, 62, 79, and 83,
' respectively. All other numbers are mapped to 255.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL num = W0 ' holds current number
SYMBOL result = W1 ' holds mapped result

Main:
 FOR num = 0 TO 100
 result = 255 ' default value for no match
 LOOKDOWN num, (0, 10, 50, 64, 71, 98), result
 LOOKUP result, (35, 40, 58, 62, 79, 83), result
 DEBUG "Num = ", #num, "Result = ", #result, CR
 PAUSE 100
 NEXT
END
```

### Demo Program (LOOKDOWN.bs2)

All 2

```
' LOOKDOWN.bs2
' This program uses LOOKDOWN to determine the number of decimal
' digits in a number. Since LOOKDOWN uses a zero-indexed table, the
' output will be the number of digits minus one, so this gets
' corrected in the following line. Note that zero is considered a
' valid number and has one digit.

' {$STAMP BS2}
' {$PBASIC 2.5}

aNum VAR Word ' the number to study
stpSz VAR Word ' FOR-NEXT step size
numDig VAR Nib ' digits in aNum

Setup:
 stpSz = 2

Main:
 FOR aNum = 0 TO 15000 STEP stpSz
 LOOKDOWN aNum, <[0, 10, 100, 1000, 10000, 65535], numDig
 ' right-justify output
 DEBUG "aNum = ", REP " "(5-numDig), DEC aNum, TAB,
 "Digits = ", DEC numDig, CR
 PAUSE 250
 LOOKDOWN aNum, <[0, 10, 100, 1000, 10000, 65535], stpSz
 LOOKUP stpSz, [2, 2, 5, 25, 250, 500, 1000], stpSz
 NEXT
END
```

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## 5: BASIC Stamp Command Reference – LOOKUP

### LOOKUP

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**LOOKUP** *Index*, ( *Value0*, *Value1*, ...*ValueN* ), *Variable*



**LOOKUP** *Index*, [ *Value0*, *Value1*, ...*ValueN* ], *Variable*

### Function

Find the value at location *Index* and store it in *Variable*. If *Index* exceeds the highest index value of the items in the list, *Variable* is left unaffected.

- **Index** is a variable/constant/expression (0 – 255) indicating the list item to retrieve.
- **Values** are variables/constants/expressions (0 – 65535).
- **Variable** is a variable that will be set to the value at the *Index* location. If *Index* exceeds the highest location number, *Variable* is left unaffected.



NOTE: Expressions are not allowed as arguments on the BS1.

### Quick Facts

Table 5.55: LOOKUP Quick Facts.

|                                          | BS1 and all BS2 Models      |
|------------------------------------------|-----------------------------|
| Limit of <i>Value</i> Entries            | 256                         |
| Starting Index Number                    | 0                           |
| If index exceeds the highest location... | Variable is left unaffected |
| Related Command                          | LOOKDOWN                    |

### Explanation

LOOKUP retrieves an item from a list based on the item's position, *Index*, in the list. For example:



```
SYMBOL index = B2
SYMBOL result = B3
```

```
index = 3
result = 255
```

```
LOOKUP index, (26, 177, 13, 1, 0, 17, 99), result
DEBUG "Item ", #index, "is: ", #result
```

```
-- OR --
```

## LOOKUP – BASIC Stamp Command Reference

---

```
index VAR Byte
result VAR Byte
```



```
index = 3
result = 255
```

```
LOOKUP index, [26, 177, 13, 1, 0, 17, 99], result
DEBUG "Item ", DEC index, " is: ", DEC result
```

In this example, DEBUG prints “Item 3 is: 1.” Note that the first location number is 0. In the list above, item 0 is 26, item 1 is 177, etc.

THE INDEX NUMBER OF THE FIRST ITEM IS 0, NOT 1.

If *index* is beyond the end of the list, the result variable is unchanged. In the example above, if *index* were greater than 6, the message would have reported the result to be 255, because that’s what *result* contained before LOOKUP executed.

Don’t forget that text phrases are just lists of byte values, so they too are eligible for LOOKUP searches, as in this example:

USING TEXT IN THE VALUE LIST.

```
SYMBOL index = B2
SYMBOL result = B3
```



```
index = 16
result = " "
```

```
LOOKUP index, ("The quick brown fox"), result
DEBUG @result
```

-- Or --

```
index VAR Byte
result VAR Byte
```



```
index = 16
result = " "
```

```
LOOKUP index, ["The quick brown fox"], result
DEBUG ASC ? result
```

DEBUG prints, “Result = 'f ' ” because the character at index item 16 is “f” in the phrase, “The quick brown fox”.

The examples above show LOOKUP working with lists of constants, but it also works with variables and expressions also. Note, however, that expressions are not allowed as argument on the BS1.

LOOKUP CAN USE VARIABLES AND EXPRESSIONS IN THE VALUE LIST.

## 5: BASIC Stamp Command Reference – LOOKUP

USING LOOKUP WITH LOOKDOWN TO  
"MAP" NON-CONTIGUOUS SETS OF  
NUMBERS.

A great use of LOOKUP is in combination with LOOKDOWN to "map" non-contiguous sets of numbers together. For example, you may have an application where certain numbers are received by the BASIC Stamp and, in response, the BASIC Stamp needs to send a specific set of numbers. This may be easy to code if the numbers are contiguous, or follow some known algebraic equations... but what if they don't? The table below shows some sample, non-contiguous inputs and the corresponding outputs the BASIC Stamp needs to respond with:

**Table 5.56:** Non-Contiguous Number Example.

| Each of these values received (inputs): | Needs to result in each of these values sent (outputs): |
|-----------------------------------------|---------------------------------------------------------|
| 5                                       | 16                                                      |
| 14                                      | 17                                                      |
| 1                                       | 18                                                      |
| 43                                      | 24                                                      |
| 26                                      | 10                                                      |
| 22                                      | 12                                                      |
| 30                                      | 11                                                      |

So, if we receive the number 5, we need to output 16. If we received 43, we need to output 24, and so on. These numbers are not contiguous and they don't appear to be derived from any simple algorithm. We can solve this problem with two lines of code, as follows:

```
LOOKDOWN value, [5, 14, 1, 43, 26, 22, 30], value
LOOKUP value, [16, 17, 18, 24, 10, 12, 11], value
```

Assuming our received number is in *value*, the first line (LOOKDOWN) will find the value in the list and store the index of the location that matches back into *value*. (This step "maps" the non-contiguous numbers: 5, 14, 1, etc, to a contiguous set of numbers: 0, 1, 2, etc). The second line (LOOKUP) takes our new *value*, finds the number at that location and stores it back into *value*. If the received value was 14, LOOKDOWN stores 1 into *value* and LOOKUP looks at the value at location 1 and stores 17 in *value*. The number 43 gets mapped to 3, 3 gets mapped to 24, and so on. This is a quick and easy fix for a potentially messy problem!



### Demo Program (LOOKUP.bs1)

```
' LOOKUP.bs1
' This program uses Lookup to create a Debug-window animation of a spinning
' propeller. The animation consists of the four ASCII characters | / - \
' which, when printed rapidly in order at a fixed location, appear to spin.
' A little imagination helps a lot here....
```

## LOOKUP – BASIC Stamp Command Reference

---

```
' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL idx = B2
SYMBOL frame = B3

Spinner:
 LOOKUP idx, ("|/-\"), frame ' lookup current frame character
 DEBUG CLS, "Spinner: ", #@frame ' display
 idx = idx + 1 // 4 ' update frame index (0..3)
 GOTO Spinner ' loop forever
END
```

### Demo Program (LOOKUP.bs2)

```
' LOOKUP.bs2
' This program uses LOOKUP to create a Debug-window animation of a spinning
' propeller. The animation consists of the four ASCII characters | / - \
' which, when printed rapidly in order at a fixed location, appear to spin.
' A little imagination helps a lot here....

' {$STAMP BS2}
' {$PBASIC 2.5}

idx VAR Nib
frame VAR Byte

Spinner:
 DO
 LOOKUP idx, ["|/-\"], frame ' lookup current frame character
 DEBUG HOME, "Spinner: ", frame ' display
 PAUSE 150 ' pause between frames
 idx = idx + 1 // 4 ' update frame index (0..3)
 LOOP ' loop forever
END
```

All 2

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

# 5: BASIC Stamp Command Reference – LOW

## LOW

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



## LOW Pin

### Function

Make the specified pin output low.

- **Pin** is a variable/constant/expression (0 – 15) that specifies which I/O pin to set low. This pin will be placed into output mode.

### Quick Facts

Table 5.57: LOW Quick Facts.

|                  | BS1 and all BS2 Models |
|------------------|------------------------|
| Related Commands | HIGH and TOGGLE        |

### Explanation

The LOW command sets the specified pin to 0 (a 0 volt level) and then sets its mode to output. For example,

```
LOW 6
```

does exactly the same thing as:

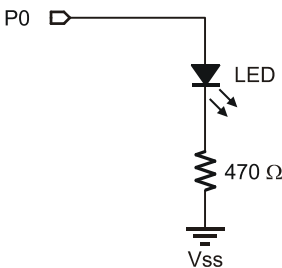


```
OUT6 = 0
DIR6 = 1
```

Using the LOW command is faster, in this case.

Connect an LED and a resistor as shown in Figure 5.17 for the demo program below.

Figure 5.17: Example LED Circuit.



## ***LOW – BASIC Stamp Command Reference***

---

### **Demo Program (LOW.bs2)**

```
' LOW.bs2
' This simple program sets I/O pin 0 high for 1/2 second and low for
' 1/2 second in an endless loop. Connect an LED to P0 for a simple
' blinker.

' {$STAMP BS2}

Main:
 HIGH 0
 PAUSE 500
 LOW 0
 PAUSE 500
 GOTO Main
END
```



NOTE: This example program can be used with the BS1 and all BS2 models by changing the \$STAMP directive accordingly.



## 5: BASIC Stamp Command Reference – MAINIO

|                                                                                   |               |            |            |             |              |             |              |              |
|-----------------------------------------------------------------------------------|---------------|------------|------------|-------------|--------------|-------------|--------------|--------------|
|  | <b>MAINIO</b> | <b>BS1</b> | <b>BS2</b> | <b>BS2e</b> | <b>BS2sx</b> | <b>BS2p</b> | <b>BS2pe</b> | <b>BS2px</b> |
|                                                                                   | <b>MAINIO</b> |            |            |             |              |             |              |              |

### Function

Switch from control of auxiliary I/O pins to main I/O pins (on the BS2p40 only).

### Quick Facts

Table 5.58: MAINIO Quick Facts.

|                  | BS2p, BS2pe, and BS2px                                                                                                  |
|------------------|-------------------------------------------------------------------------------------------------------------------------|
| I/O pin IDs      | 0 – 15 (just like auxiliary I/O, but after MAINIO command, all references affect physical pins 5 – 20).                 |
| Special Notes    | The 24-pin BS2p, BS2pe, and BS2px accept this command, however, only the BS2p40 gives access to the auxiliary I/O pins. |
| Related Commands | AUXIO and IOTERM                                                                                                        |

### Explanation

The BS2p, BS2pe and BS2px are available as 24-pin modules that are pin compatible with the BS2, BS2e and BS2sx. Also available is a 40-pin module called the BS2p40, with an additional 16 I/O pins (for a total of 32). The BS2p40's extra, or auxiliary, I/O pins can be accessed in the same manner as the main I/O pins (by using the IDs 0 to 15) but only after issuing AUXIO or IOTERM commands. The MAINIO command causes the BASIC Stamp to affect the main I/O pins (the default) instead of the auxiliary I/O pins in all further code until the AUXIO or IOTERM command is reached, or the BASIC Stamp is reset or power-cycled.

A SIMPLE MAINIO EXAMPLE.

The following example illustrates this:

```
AUXIO ' switch to auxiliary pins
HIGH 0 ' make X0 high
MAINIO ' switch to main pins
LOW 0 ' make P0 low
```

The first line of the above example will tell the BASIC Stamp to affect the auxiliary I/O pins in the commands following it. Line 2, sets I/O pin 0 of the auxiliary I/O pins (physical pin 21) high. Afterward, the MAINIO command tells the BASIC Stamp that all commands following it should affect the main I/O pins. The last command, LOW, will set I/O pin 0 of the main I/O pins (physical pin 5) low.

## MAINIO – BASIC Stamp Command Reference

Note that the main I/O and auxiliary I/O pins are independent of each other; the states of the main I/O pins remain unchanged while the program affects the auxiliary I/O pins, and vice versa.

MAIN I/O AND AUXILIARY I/O PINS ARE INDEPENDENT AND UNAFFECTED BY CHANGES IN THE OPPOSITE GROUP.

Other commands that affect I/O group access are AUXIO and IOTERM.

### Demo Program (AUX\_MAIN\_TERM.bsp)

```
' AUX_MAIN_TERM.bsp
' This program demonstrates the use of the AUXIO, MAINIO and IOTERM
' commands to affect I/O pins in the auxiliary and main I/O groups.

' {$STAMP BS2p}
' {$PBASIC 2.5}

#SELECT $STAMP
 #CASE BS2, BS2E, BS2SX
 #ERROR "Program requires BS2p40"
 #CASE BS2P, BS2PE, BS2PX
 DEBUG "Note: This program designed for the BS2p40.", CR
#ENDSELECT

port VAR Bit

Main:
 DO
 MAINIO ' Switch to main I/O pins
 TOGGLE 0 ' Toggle state of I/O pin P0
 PWM 1, 100, 40 ' Generate PWM on I/O pin P1

 AUXIO ' Switch to auxiliary I/O pins
 TOGGLE 0 ' Toggle state of I/O pin X0
 PULSOUT 1, 1000 ' Generate a pulse on I/O pin X1
 PWM 2, 100, 40 ' Generate PWM on I/O pin X2

 IOTERM port ' Switch to main or aux I/Os
 ' -- depending on port
 TOGGLE 3 ' Toggle state of I/O pin 3
 ' -- on main and aux, alternately
 port = ~port ' Invert port
 PAUSE 1000 ' 1 second delay
 LOOP
END
```



NOTE: This example program will tokenize with the 24-pin BS2p, BS2pe, and BS2px, but its effects can only be seen on the BS2p40. This program uses conditional compilation techniques; see Chapter 3 for more information.

## 5: BASIC Stamp Command Reference – NAP

### NAP

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



#### NAP Duration

### Function

Enter sleep mode for a short time. Power consumption is reduced as indicated in Table 5.59 assuming no loads are being driven.

- **Duration** is a variable/constant/expression (0 – 7) that specifies the duration of the reduced-power nap. The duration is  $(2^{\text{Duration}}) * 18 \text{ ms}$ . Table 5.60 indicates the nap length for any given *Duration*.

### Quick Facts

**Table 5.59:** NAP Quick Facts.

Note: Current measurements are based on 5-volt power, no extra loads, and 75°F ambient temperature.

|                         | BS1                                                      | BS2        | BS2e        | BS2sx       | BS2p                     | BS2pe      | BS2px       |
|-------------------------|----------------------------------------------------------|------------|-------------|-------------|--------------------------|------------|-------------|
| Current Draw During Run | 1 mA                                                     | 3 mA       | 25 mA       | 60 mA       | 40 mA                    | 15 mA      | 55 mA       |
| Current Draw During NAP | 25 $\mu$ A                                               | 50 $\mu$ A | 200 $\mu$ A | 500 $\mu$ A | 350 $\mu$ A              | 36 $\mu$ A | 450 $\mu$ A |
| Related Commands        | END and SLEEP                                            |            |             |             | END, SLEEP, and POLLWAIT |            |             |
| Accuracy of Nap         | -50 to 100% ( $\pm$ 10% @ 75°F with stable power supply) |            |             |             |                          |            |             |

### Explanation

NAP uses the same shutdown/startup mechanism as SLEEP, with one big difference. During SLEEP, the BASIC Stamp automatically compensates for variations in the speed of the watchdog timer oscillator that serves as its alarm clock. As a result, longer SLEEP intervals are accurate to approximately  $\pm 1$  percent.

**Table 5.60:** *Duration* and Resulting Length of NAP.

| <i>Duration</i> | Length of Nap           |
|-----------------|-------------------------|
| 0               | 18 ms                   |
| 1               | 36 ms                   |
| 2               | 72 ms                   |
| 3               | 144 ms                  |
| 4               | 288 ms                  |
| 5               | 576 ms                  |
| 6               | 1152 ms (1.152 seconds) |
| 7               | 2304 ms (2.304 seconds) |

NAP ACCURACY NOTES.

NAP intervals are directly controlled by the watchdog timer without compensation. Variations in temperature, supply voltage, and manufacturing tolerance of the BASIC Stamp's interpreter chip can cause

## ***NAP – BASIC Stamp Command Reference***

---

the actual timing to vary by as much as -50, +100 percent (i.e., a *Duration* of 0, NAP can range from 9 to 36 ms). At room temperature with a fresh battery or other stable power supply, variations in the length of a NAP will be less than  $\pm 10$  percent.

One great use for NAP is in a battery-powered application where at least a small amount of time is spent doing nothing. For example, you may have a program that loops endlessly, performing some task, that pauses for approximately 100 ms each time through the loop. You could replace your PAUSE 100 with NAP 3, as long as the timing of the 100 ms pause was not critical. The NAP 3 would effectively pause your program for about 144 ms and, at the same time, would place the BASIC Stamp in low-power mode, which would extend your battery life.

A GREAT USE FOR NAP; FREE POWER SAVINGS.

If your application is driving loads (sourcing or sinking current through output-high or output-low pins) during a NAP, current will be interrupted for about 18 ms (60  $\mu$ s on the BS2pe) when the BASIC Stamp wakes up. The reason is that the watchdog-timer reset that awakens the BASIC Stamp also causes all of the pins to switch to input mode for approximately 18 ms (60  $\mu$ s on the BS2pe). When the interpreter firmware regains control of the processor, it restores the I/O direction dictated by your program.

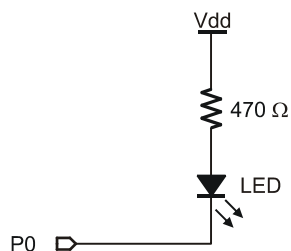
TIPS FOR DRIVING LOADS DURING NAP.

If you plan to use END, NAP, POLLWAIT or SLEEP in your programs, make sure that your loads can tolerate these power outages. The simplest solution is often to connect resistors high or low (to +5V or ground) as appropriate to ensure a continuing supply of current during the reset glitch.

The demo program can be used to demonstrate the effects of the NAP glitch with an LED and resistor as shown in Figure 5.18.

## 5: BASIC Stamp Command Reference – NAP

Figure 5.18: Example LED Circuit.



All 2

NOTE: This example program can be used with the BS1 and all BS2 models by changing the \$STAMP directive accordingly.

### Demo Program (NAP.bs2)

```
' NAP.bs2
' The program below lights an LED by placing a low on pin 0. This completes
' the circuit from +5V, through the LED and resistor, to ground. During the
' NAP interval, the LED stays lit, but blinks off for a fraction of a sec.
' This blink is caused by the NAP wakeup mechanism during wakeup, all pins
' briefly slip into input mode, effectively disconnecting them from loads.

' {$STAMP BS2}

Setup:
 LOW 0 ' turn LED on

Snooze:
 NAP 4 ' nap for 288 ms
 GOTO Snooze
END
```

## ***NAP – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – ON

### ON

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|

NOTE: ON requires PBASIC 2.5.

All 2

**ON Offset GOTO Address1, Address2, ...AddressN**

All 2

**ON Offset GOSUB Address1, Address2, ...AddressN**

### Function

GOTO or GOSUB to the *Address* specified by *Offset* (if in range). ON is similar in operation to BRANCH with the exception that program execution can optionally return to the line following ON (if using ON...GOSUB).

- **Offset** is a variable/constant/expression (0 - 255) that specifies the index (0 - N) of the address, in the list, to GOTO or GOSUB to.
- **Address** is a label that specifies where to go for a given *Offset*. ON will ignore any list entries beyond offset 255.

### Quick Facts

Table 5.61: ON Quick Facts.

|                            | All BS2 Models                                                                          |
|----------------------------|-----------------------------------------------------------------------------------------|
| Limit of Address Entries   | 256                                                                                     |
| Maximum GOSUBs per Program | 255 (each ON...GOSUB counts as one GOSUB, regardless of number of address list entries) |
| Maximum Nested GOSUBS      | 4                                                                                       |
| Related Commands           | BRANCH, GOTO and GOSUB                                                                  |

### Explanation

The ON instruction is like saying, "Based ON the value of *Offset*, GOTO or GOSUB to one of these *Addresses*." ON is useful when you want to write something like this:

```
IF (value = 0) THEN GOTO Case_0 ' "GOTO" jump table
IF (value = 1) THEN GOTO Case_1
IF (value = 2) THEN GOTO Case_2
```

- or -

```
IF (value = 0) THEN GOSUB Case_0 ' "GOSUB" jump table
IF (value = 1) THEN GOSUB Case_1
IF (value = 2) THEN GOSUB Case_2
```

## ON – BASIC Stamp Command Reference

---

You can use ON to organize each of these two examples into single statements:

```
ON value GOTO Case_0, Case_1, Case_2 ' "GOTO" jump table
```

- or -

```
ON value GOSUB Case_0, Case_1, Case_2 ' "GOSUB" jump table
```

This works like the previous IF...THEN example. If the value isn't in range (in this case if value is greater than 2), ON does nothing and the program continues with the next instruction after ON.

See the GOTO and GOSUB command descriptions for more information.

### Demo Program (ON-GOTO.bs2)

```
' ON-GOTO.bs2
' This program shows how the value of idx controls the destination of the
' ON...GOTO instruction.

' {$STAMP BS2}
' {$PBASIC 2.5}

idx VAR Byte

Main:
 DEBUG "idx: ", DEC idx, " "
 ON idx GOTO Case_0, Case_1, Case_2 ' if idx = 0..2 goto label
 DEBUG "ON..GOTO target error.", CR ' message if idx is out of range

Update:
 idx = idx + 1 // 4 ' force idx to be 0..3
 PAUSE 1000
 GOTO Main

Case_0:
 DEBUG "Running Case_0 routine", CR
 GOTO Update

Case_1:
 DEBUG "Running Case_1 routine", CR
 GOTO Update

Case_2:
 DEBUG "Running Case_2 routine", CR
 GOTO Update
```

All 2

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.



## 5: BASIC Stamp Command Reference – ON

All 2

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

### Demo Program (ON-GOSUB.bs2)

```
' ON-GOSUB.bs2
' This program demonstrates a simple task manager that can be used
' in a variety of applications. It is particularly useful in
' robotics and industrial applications. The advantage of this
' design is that task code modules may be called from other places
' in the program, including other tasks, and the overall program flow
' is maintained.

' {$STAMP BS2}
' {$PBASIC 2.5}

task VAR Nib

Main:
DO
 ON task GOSUB Task_0, Task_1, Task_2 ' run current task
 task = task + 1 // 3 ' update task pointer
 PAUSE 1000
LOOP
END

Task_0:
 DEBUG "Running Task 0", CR
 RETURN

Task_1:
 DEBUG "Running Task 1", CR
 RETURN

Task_2:
 DEBUG "Running Task 2", CR
 RETURN
```

## ***ON – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – OUTPUT

### OUTPUT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



#### OUTPUT *Pin*

#### Function

Make the specified pin an output.

- ***Pin*** is a variable/constant/expression (0 – 15) that specifies which I/O pin to set to output mode.

#### Quick Facts

|                  | BS1 and all BS2 Models |
|------------------|------------------------|
| Related Commands | INPUT and REVERSE      |

#### Explanation

There are several ways to make a pin an output. Commands that rely on output pins, like PULSOUT and SEROUT, automatically change the specified pin to output. Writing 1s to particular bits of the variable DIRS makes the corresponding pins outputs. And then there's the OUTPUT command.

When a pin is an output, your program can change its state by writing to the corresponding bit in the OUTS variable (PINS on the BS1). For example:



```
OUTPUT 4
OUT4 = 1
```

EFFECTS OF SETTING AN INPUT PIN TO AN OUTPUT.

When your program changes a pin from input to output, whatever state happens to be in the corresponding bit of OUTS (PINS on the BS1) sets the initial state of the pin. To simultaneously make a pin an output and set its state use the HIGH and LOW commands.



#### Demo Program (INPUT\_OUTPUT.bs1)

```
' INPUT_OUTPUT.bs1
' This program demonstrates how the input/output direction of a pin is
' determined by the corresponding bit of DIRS. It also shows that the
' state of the pin itself (as reflected by the corresponding bit of PINS)
' is determined by the outside world when the pin is an input, and by the
' corresponding bit of PINS when it's an output. To set up the demo,
' connect a 10k resistor from +5V to P7 on the BASIC Stamp. The resistor
```

## OUTPUT – BASIC Stamp Command Reference

---

```
' to +5V puts a high (1) on the pin when it's an input. The BASIC Stamp
' can override this state by writing a low (0) to bit 7 of OUTS and
' changing the pin to output.

' {$STAMP BS1}
' {$PBASIC 1.0}

Main:
 INPUT 7 ' Make P7 an input
 DEBUG "State of P7: ", #PIN7, CR

 PIN7 = 0 ' Write 0 to output latch
 DEBUG "After 0 written to OUT7: "
 DEBUG #PIN7, CR

 OUTPUT 7 ' Make P7 an output
 DEBUG "After P7 changed to output: "
 DEBUG #PIN7
END
```

### Demo Program (INPUT\_OUTPUT.bs2)

```
' INPUT_OUTPUT.bs2
' This program demonstrates how the input/output direction of a pin is
' determined by the corresponding bit of DIRS. It also shows that the
' state of the pin itself (as reflected by the corresponding bit of INS)
' is determined by the outside world when the pin is an input, and by the
' corresponding bit of OUTS when it's an output. To set up the demo,
' connect a 10k resistor from +5V to P7 on the BASIC Stamp. The resistor
' to +5V puts a high (1) on the pin when it's an input. The BASIC Stamp
' can override this state by writing a low (0) to bit 7 of OUTS and
' changing the pin to output.

' {$STAMP BS2}
' {$PBASIC 2.5}

Main:
 INPUT 7 ' Make P7 an input
 DEBUG "State of P7: ",
 BIN1 IN7, CR

 OUT7 = 0 ' Write 0 to output latch
 DEBUG "After 0 written to OUT7: ",
 BIN1 IN7, CR

 OUTPUT 7 ' Make P7 an output
 DEBUG "After P7 changed to output: ",
 BIN1 IN7
END
```



NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## 5: BASIC Stamp Command Reference – OWIN

|                                                                                                                                                                                                                                                       |                                      |     |     |      |       |      |       |       |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|-----|-----|------|-------|------|-------|-------|
| OWIN                                                                                                                                                                                                                                                  |                                      | BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|    | OWIN <i>Pin, Mode, [ InputData ]</i> |     |     |      |       |      |       |       |

### Function

Receive data from a device using the 1-Wire protocol.

- **Pin** is a variable/constant/expression (0 – 15) that specifies which I/O pin to use. 1-Wire devices require only one I/O pin (called DQ) to communicate. This I/O pin will be toggled between output and input mode during the OWIN command and will be set to input mode by the end of the OWIN command.
- **Mode** is a variable/constant/expression (0 – 15) indicating the mode of data transfer. The *Mode* argument controls placement of reset pulses (and detection of presence pulses) as well as byte vs. bit input and normal vs. high speed. See explanation below.
- **InputData** is a list of variables and modifiers that tells OWIN what to do with incoming data. OWIN can store data in a variable or array, interpret numeric text (decimal, binary, or hex) and store the corresponding value in a variable, wait for a fixed or variable sequence of bytes, or ignore a specified number of bytes. These actions can be combined in any order in the *InputData* list.

Table 5.63: OWIN Quick Facts.

### Quick Facts

|                  | BS2p, BS2pe, and BS2px                                                                                                               |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| Receive Rate     | Approximately 20 kbits/sec (low speed, not including reset pulse)                                                                    |
| Special Notes    | The DQ pin (specified by <i>Pin</i> ) must have a 4.7 K $\Omega$ pull-up resistor. The BS2pe is not capable of high-speed transfers. |
| Related Commands | OWOUT                                                                                                                                |

### Explanation

The 1-Wire protocol is a form of asynchronous serial communication developed by Dallas Semiconductor. It only requires one I/O pin and that pin can be shared between multiple 1-Wire devices. The OWIN command allows the BASIC Stamp to receive data from a 1-wire device.

A SIMPLE OWIN EXAMPLE.

The following is an example of the OWIN command:

```
result VAR Byte
OWIN 0, 1, [result]
```

## OWIN – BASIC Stamp Command Reference

This code will transmit a "reset" pulse to a 1-Wire device (connected to I/O pin 0) and then will detect the device's "presence" pulse and then receive one byte and store it in the variable *result*.

The *Mode* argument is used to control placement of reset pulses (and detection of presence pulses) and to designate byte vs. bit input and normal vs. high speed. Figure 5.19 shows the meaning of each of the 4 bits of Mode. Table 5.64 shows just some of the 16 possible values and their effect.

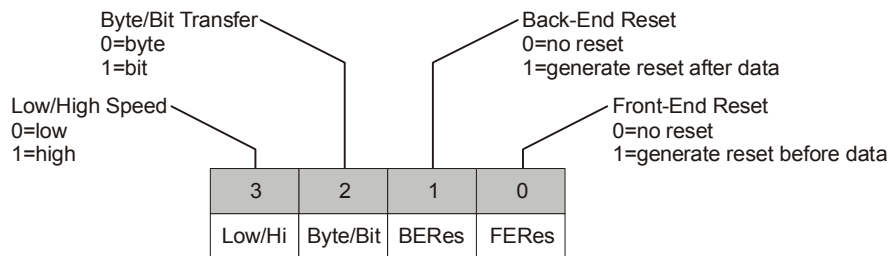


Figure 5.19: Mode Format.

| Mode | Effect                                            |
|------|---------------------------------------------------|
| 0    | No Reset, Byte mode, Low speed                    |
| 1    | Reset before data, Byte mode, Low speed           |
| 2    | Reset after data, Byte mode, Low speed            |
| 3    | Reset before and after data, Byte mode, Low speed |
| 4    | No Reset, Bit mode, Low speed                     |
| 5    | Reset before data, Bit mode, Low speed            |
| 8    | No Reset, Byte mode, High speed                   |
| 9    | Reset before data, Byte mode, High speed          |

Table 5.64: OWIN Mode Values.

**NOTE:** The BS2pe is not capable of high-speed transfers.

The proper value for *Mode* depends on the 1-Wire device and the portion of the communication you're working on. Please consult the data sheet for the device in question to determine the correct value for *Mode*. In many cases, however, when using the OWIN command, *Mode* should be set for either No Reset (to receive data from a transaction already started by a OWOUT command) or a Back-End Reset (to terminate the session after data is received). This may vary due to device and application requirements, however.

When using the Bit (rather than Byte) mode of data transfer, all variables in the *InputData* argument will only receive one bit. For example, the following code could be used to receive two bits using this mode:

## 5: BASIC Stamp Command Reference – OWIN

```
bitOne VAR Byte
bitTwo VAR Byte

OWIN 0, 6, [bitOne, bitTwo]
```

In the code above, we chose the value “6” for *Mode*. This sets Bit transfer and Back-End Reset modes. Also, we could have chosen to make the *bitOne* and *bitTwo* variables each a byte in size, but they still would only have received one bit each in the OWIN command (due to the *Mode* we chose).

### RECEIVING FORMATTED DATA.

The OWIN command’s *InputData* argument is similar to the SERIN command’s *InputData* argument. This means data can be received as ASCII character values, decimal, hexadecimal and binary translations and string data as in the examples below. (Assume a 1-Wire device is used and that it transmits the string, “Value: 3A:101” every time it receives a Front-End Reset pulse).

```
value VAR Byte(13)

OWIN 0, 1, [value] ` receive ASCII value for "V"
OWIN 0, 1, [DEC value] ` receive the number 3
OWIN 0, 1, [HEX value] ` receive the number $3A
OWIN 0, 1, [BIN value] ` receive the number %101
OWIN 0, 1, [STR value\13] ` receive "Value: 3A:101"
```

Table 5.65 and Table 5.66 list all the special formatters and conversion formatters available to the OWIN command. See the SERIN command for additional information and examples of their use.

**Table 5.65:** OWIN Special Formatters.

| Special Formatter     | Action                                                                                                                                                                                                                              |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SPSTR L               | Input a character string of length L bytes (up to 126) into Scratch Pad RAM, starting at location 0. Use GET to retrieve the characters.                                                                                            |
| STR ByteArray \L {E}  | Input a character string of length L into an array. If specified, an end character E causes the string input to end before reaching length L. Remaining bytes are filled with 0s (zeros).                                           |
| WAITSTR ByteArray {L} | Wait for a sequence of bytes matching a string stored in an array variable, optionally limited to L characters. If the optional L argument is left off, the end of the array-string must be marked by a byte containing a zero (0). |
| SKIP Length           | Ignore Length bytes of characters.                                                                                                                                                                                                  |

## OWIN – BASIC Stamp Command Reference

| Conversion Formatter | Type of Number                                                                                         | Numeric Characters Accepted        | Notes   |
|----------------------|--------------------------------------------------------------------------------------------------------|------------------------------------|---------|
| DEC{1..5}            | Decimal, optionally limited to 1 – 5 digits                                                            | 0 through 9                        | 1       |
| SDEC{1..5}           | Signed decimal, optionally limited to 1 – 5 digits                                                     | -, 0 through 9                     | 1,2     |
| HEX{1..4}            | Hexadecimal, optionally limited to 1 – 4 digits                                                        | 0 through 9, A through F           | 1,3,5   |
| SHEX{1..4}           | Signed hexadecimal, optionally limited to 1 – 4 digits                                                 | -, 0 through 9, A through F        | 1,2,3   |
| IHEX{1..4}           | Indicated hexadecimal, optionally limited to 1 – 4 digits                                              | \$, 0 through 9, A through F       | 1,3,4   |
| ISHEX{1..4}          | Signed, indicated hexadecimal, optionally limited to 1 – 4 digits                                      | -, \$, 0 through 9, A through F    | 1,2,3,4 |
| BIN{1..16}           | Binary, optionally limited to 1 – 16 digits                                                            | 0, 1                               | 1       |
| SBIN{1..16}          | Signed binary, optionally limited to 1 – 16 digits                                                     | -, 0, 1                            | 1,2     |
| IBIN{1..16}          | Indicated binary, optionally limited to 1 – 16 digits                                                  | %, 0, 1                            | 1,4     |
| ISBIN{1..16}         | Signed, indicated binary, optionally limited to 1 – 16 digits                                          | -, %, 0, 1                         | 1,2,4   |
| NUM                  | Generic numeric input (decimal, hexadecimal or binary); hexadecimal or binary number must be indicated | \$, %, 0 through 9, A through F    | 1, 3, 4 |
| SNUM                 | Similar to NUM with value treated as signed with range -32768 to +32767                                | -, \$, %, 0 through 9, A through F | 1,2,3,4 |

**Table 5.66:** OWIN Conversion Formatters

- 1 All numeric conversions will continue to accept new data until receiving either the specified number of digits (ex: three digits for DEC3) or a non-numeric character.
- 2 To be recognized as part of a number, the minus sign (-) must immediately precede a numeric character. The minus sign character occurring in non-numeric text is ignored and any character (including a space) between a minus and a number causes the minus to be ignored.
- 3 The hexadecimal formatters are not case-sensitive; “a” through “f” means the same as “A” through “F”.
- 4 Indicated hexadecimal and binary formatters ignore all characters, even valid numerics, until they receive the appropriate prefix (\$ for hexadecimal, % for binary). The indicated formatters can differentiate between text and hexadecimal (ex: ABC would be interpreted by HEX as a number but IHEX would ignore it unless expressed as \$ABC). Likewise, the binary version can distinguish the decimal number 10 from the binary number %10. A prefix occurring in non-numeric text is ignored, and any character (including a space) between a prefix and a number causes the prefix to be ignored. Indicated, signed formatters require that the minus sign come before the prefix, as in -\$1B45.
- 5 The HEX modifier can be used for Decimal to BCD Conversion. See “Hex to BCD Conversion” on page 97.

For examples of all conversion formatters and how they process incoming data, see Appendix C.



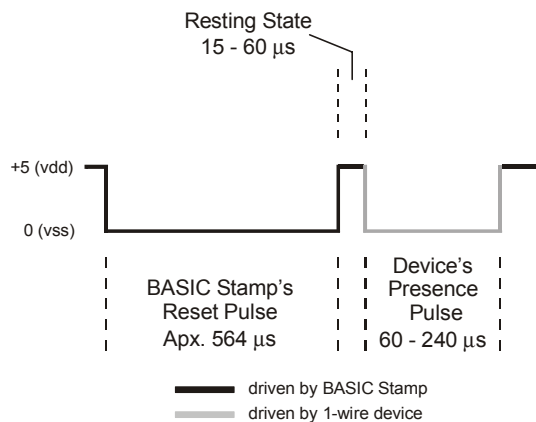
## 5: BASIC Stamp Command Reference – OWIN

### THE 1-WIRE PROTOCOL FORMAT.

The 1-Wire protocol has a well-defined standard for transaction sequences. Every transaction sequence consists of four parts: 1) Initialization, 2) ROM Function Command, 3) Memory Function Command, and 4) Transaction/Data. Additionally, the ROM Function Command and Memory Function Command are always 8 bits wide (1 byte in size) and is sent least-significant-bit (LSB) first.

The Initialization part consists of a reset pulse (generated by the master) and will be followed by a presence pulse (generated by all slave devices). Figure 5.20 details the reset pulse generated by the BASIC Stamp and a typical presence pulse generated by a 1-wire slave, in response.

**Figure 5.20:** OWIN Reset and Presence Pulse.



This reset pulse is controlled by the lowest two bits of the *Mode* argument in the OWIN command. It can be made to appear before the ROM Function Command (ex: *Mode* = 1), after the Transaction/Data portion (ex: *Mode* = 2), before and after the entire transaction (ex: *Mode* = 3) or not at all (ex: *Mode* = 0). See the section on *Mode*, above, for more information.

Following the Initialization part is the ROM Function Command. The ROM Function Command is used to address the desired 1-Wire device. Table 5.67 shows common ROM Function Commands. If only a single 1-Wire device is connected, the Skip ROM command may be used to address it. If more than one 1-Wire device is attached, the BASIC Stamp will ultimately have to address them individually using the Match ROM command.

## OWIN – BASIC Stamp Command Reference

| Command    | Value (in Hex) | Action                                                                                                                          |
|------------|----------------|---------------------------------------------------------------------------------------------------------------------------------|
| Read ROM   | \$33           | Reads the 64-bit ID of the 1-Wire device. This command can only be used if there is a single 1-wire device on the line.         |
| Match ROM  | \$55           | This command, followed by a 64-bit ID, allows the BASIC Stamp to address a specific 1-Wire device.                              |
| Skip ROM   | \$CC           | Address a 1-Wire device without its 64-bit ID. This command can only be used if there is a single 1-wire device on the line.    |
| Search ROM | \$F0           | Reads the 64-bit IDs of all the 1-Wire devices on the line. A process of elimination is used to distinguish each unique device. |

**Table 5.67:** 1-Wire ROM Function Commands.

The third part, the Memory Function Command, allows the BASIC Stamp to address specific memory locations, or features, of the 1-Wire device. Refer to the 1-Wire device's data sheet for a list of the available Memory Function Commands.

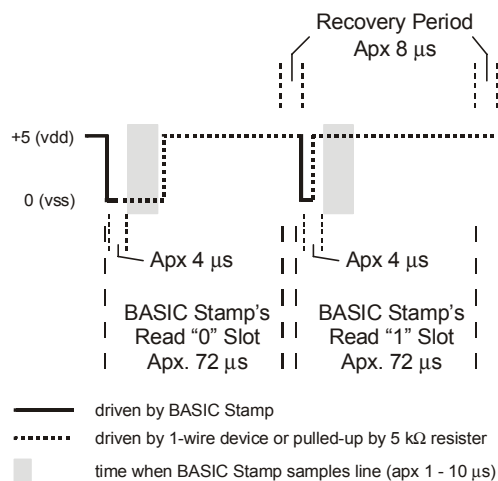


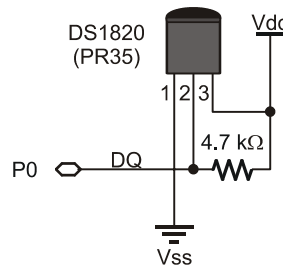
**Figure 5.21:** Example Read Slot.

Finally, the Transaction/Data section is used to read or write data to the 1-Wire device. The OWIN command will read data at this point in the transaction. A read is accomplished by generating a brief low-pulse and sampling the line within 15  $\mu$ s of the falling edge of the pulse. This is called a "Read Slot." Figure 5.21 shows typical Read Slots performed by the BASIC Stamp. See the OWOUT command for information on Write Slots.

## 5: BASIC Stamp Command Reference – OWIN

The demo program uses a Dallas Semiconductor DS1820 Digital Thermometer device connected as follows. Note that the 4.7 kΩ pull-up resistor is required for proper operation.

**Figure 5.22:** DS1820 Circuit.  
NOTE: The 4.7 kΩ resistor is required for proper operation.



### Demo Program (OWIN\_OWOUT.bsp)

NOTE: This example program can be used with the BS2p, BS2pe and BS2px by changing the \$STAMP directive accordingly.

```
' OWIN_OWOUT.bsp
' This program demonstrates interfacing to a Dallas Semiconductor DS1822
' 1-Wire Digital Thermometer chip using the BS2p's 1-Wire commands. Connect
' the BS2p, BS2pe, or BS2px to the DS1822 as shown in the diagram in the
' OWIN or OWOUT command description. This program uses a simplified
' approach that ignores the fractional portion of the temperature.

' {$STAMP BS2p}
' {$PBASIC 2.5}

DQ PIN 0 ' 1-Wire buss pin

RdROM CON $33 ' read serial number
MatchROM CON $55 ' match SN -- for multiple devices
SkipROM CON $CC ' ignore SN -- use for one device
CvrtTmp CON $44 ' start temperature conversion
RdSP CON $BE ' read DS1822 scratch pad

tempIn VAR Word ' raw temperature
sign VAR tempIn.BIT11 ' 1 = negative temperature
tLo VAR tempIn.BYTE0
tHi VAR tempIn.BYTE1
tSign VAR Bit ' saved sign bit
tempC VAR Word ' final Celsius temp
tempF VAR Word ' final Fahrenheit temp

Main:
DO
 GOSUB Get_Temperature ' read temperature from DS1822
 DEBUG HOME, ' display
 "DS1822", CR,
 "-----", CR,
```

## ***OWIN – BASIC Stamp Command Reference***

---

```
 SDEC tempC, " C ", CR,
 SDEC tempF, " F "
 PAUSE 1000
LOOP
END

Get_Temperature:
 OWOUT DQ, 1, [SkipROM, CvrtTmp] ' send convert temperature command
 DO ' wait on conversion
 PAUSE 25 ' small loop pad
 OWIN DQ, 4, [tempIn] ' check status (bit transfer)
 LOOP UNTIL (tempIn) ' 1 when complete
 OWOUT DQ, 1, [SkipROM, RdSP] ' read DS1822 scratch pad
 OWIN DQ, 2, [tLo, tHi] ' get raw temp data
 tSign = sign ' save sign bit
 tempC = tempIn >> 4 ' round to whole degrees
 tempC.BYTE1 = $FF * tSign ' correct twos complement bits
 tempF = (ABS tempC) * 9 / 5 ' start F conversion
 IF (tSign) THEN ' finish F conversion
 tempF = 32 - tempF ' C was negative
 ELSE
 tempF = tempF + 32 ' C was positive
 ENDIF
RETURN
```

## 5: BASIC Stamp Command Reference – OWOUT

### OWOUT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|

 **OWOUT** *Pin, Mode, [ OutputData ]*

#### Function

Send data to a device using the 1-Wire protocol.

- **Pin** is a variable/constant/expression (0 – 15) that specifies which I/O pin to use. 1-Wire devices require only one I/O pin (called DQ) to communicate. This I/O pin will be toggled between output and input mode during the OWOUT command and will be set to input mode by the end of the OWOUT command.
- **Mode** is a variable/constant/expression (0 – 15) indicating the mode of data transfer. The *Mode* argument controls placement of reset pulses (and detection of presence pulses) as well as byte vs. bit input and normal vs. high speed. See explanation below.
- **OutputData** is a list of variables and modifiers that tells OWOUT how to format outgoing data. OWOUT can transmit individual or repeating bytes, convert values into decimal, hexadecimal or binary text representations, or transmit strings of bytes from variable arrays. These actions can be combined in any order in the *OutputData* list.

#### Quick Facts

|                          | BS2p, BS2pe, and BS2px                                                                                                               |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <b>Transmission Rate</b> | Approximately 20 kbits/sec (low speed, not including reset pulse)                                                                    |
| <b>Special Notes</b>     | The DQ pin (specified by <i>Pin</i> ) must have a 4.7 K $\Omega$ pull-up resistor. The BS2pe is not capable of high-speed transfers. |
| <b>Related Command</b>   | OWIN                                                                                                                                 |

#### Explanation

The 1-Wire protocol is a form of asynchronous serial communication developed by Dallas Semiconductor. It only requires one I/O pin and that pin can be shared between multiple 1-Wire devices. The OWOUT command allows the BASIC Stamp to send data to a 1-Wire device.

**Table 5.68:** OWOUT Quick Facts.

A SIMPLE OWOUT EXAMPLE.

The following is an example of the OWOUT command:

```
OWOUT 0, 1, [$4E]
```

## OWOUT – BASIC Stamp Command Reference

This code will transmit a "reset" pulse to a 1-Wire device (connected to I/O pin 0) and then will detect the device's "presence" pulse and then transmit one byte (the value \$4E).

The *Mode* argument is used to control placement of reset pulses (and detection of presence pulses) and to designate byte vs. bit input and normal vs. high speed. Figure 5.23 shows the meaning of each of the 4 bits of *Mode*. Table 5.69 shows just some of the 16 possible values and their effect.

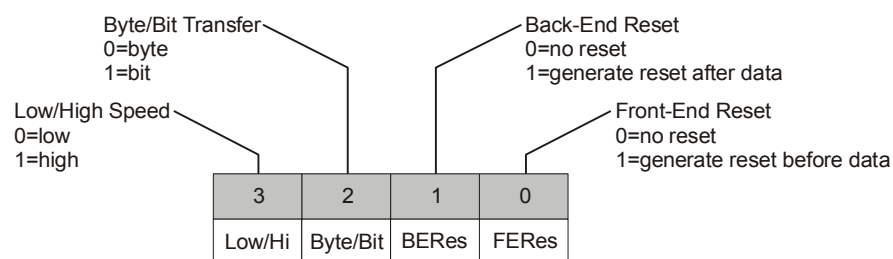


Figure 5.23: MODE Format.

| Mode | Effect                                            |
|------|---------------------------------------------------|
| 0    | No Reset, Byte mode, Low speed                    |
| 1    | Reset before data, Byte mode, Low speed           |
| 2    | Reset after data, Byte mode, Low speed            |
| 3    | Reset before and after data, Byte mode, Low speed |
| 4    | No Reset, Bit mode, Low speed                     |
| 5    | Reset before data, Bit mode, Low speed            |
| 8    | No Reset, Byte mode, High speed                   |
| 9    | Reset before data, Byte mode, High speed          |

Table 5.69: OWOUT Mode Values.

**NOTE:** The BS2pe is not capable of high-speed transfers.

The proper value for *Mode* depends on the 1-Wire device and the portion of the communication you're working on. Please consult the data sheet for the device in question to determine the correct value for *Mode*. In many cases, however, when using the OWOUT command, *Mode* should be set for a Front-End Reset (to initialize the transaction). This may vary due to device and application requirements, however.

When using the Bit (rather than Byte) mode of data transfer, all variables in the *OutputData* argument will only transmit one bit. For example, the following code could be used to send two bits using this mode:

## 5: BASIC Stamp Command Reference – OWOUT

```
bitOne VAR Bit
bitTwo VAR Bit

bitOne = 0
bitTwo = 1
OWOUT 0, 5, [bitOne, bitTwo]
```

In the code above, we chose the value "5" for *Mode*. This sets Bit transfer and Front-End Reset modes. Also, we could have chosen to make the *bitOne* and *bitTwo* variables each a byte in size, but the BASIC Stamp would still only use the their lowest bit (BIT0) as the value to transmit in the OWOUT command (due to the *Mode* we chose).

### SENDING AND FORMATTING DATA.

The OWOUT command's *OutputData* argument is similar to the DEBUG and SEROUT command's *OutputData* argument. This means data can be sent as literal text, ASCII character values, repetitive values, decimal, hexadecimal and binary translations and string data as in the examples below. (Assume a 1-wire device is used and that it transmits the string, "Value: 3A:101" every time it receives a Front-End Reset pulse).

```
value VAR Byte
value = 65

OWOUT 0, 1, [value] ' send "A"
OWOUT 0, 1, [REP value\5] ' send "AAAAA"
OWOUT 0, 1, [DEC value] ' send "6" and "5"
OWOUT 0, 1, [HEX value] ' send "4" and "1"
OWOUT 0, 1, [BIN value] ' send "1000001"
```

Table 5.70 and Table 5.71 list all the special formatters and conversion formatters available to the OWOUT command. See the DEBUG and SEROUT commands for additional information and examples of their use.

**Table 5.70:** OWOUT Special Formatters.

| Special Formatter | Action                                                                                                                                                                                                                |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ?                 | Displays "symbol = x" + carriage return; where x is a number. Default format is decimal, but may be combined with conversion formatters (ex: BIN ? x to display "x = binary_number").                                 |
| ASC ?             | Displays "symbol = 'x'" + carriage return; where x is an ASCII character.                                                                                                                                             |
| STR ByteArray {L} | Send character string from an array. The optional \L argument can be used to limit the output to L characters, otherwise, characters will be sent up to the first byte equal to 0 or the end of RAM space is reached. |
| REP Byte \L       | Send a string consisting of <i>Byte</i> repeated L times (ex: REP "X"\10 sends "XXXXXXXXXX").                                                                                                                         |

## OWOUT – BASIC Stamp Command Reference

| Conversion Formatter | Type of Number                                                              | Notes |
|----------------------|-----------------------------------------------------------------------------|-------|
| DEC{1..5}            | Decimal, optionally fixed to 1 – 5 digits                                   | 1     |
| SDEC{1..5}           | Signed decimal, optionally fixed to 1 – 5 digits                            | 1,2   |
| HEX{1..4}            | Hexadecimal, optionally fixed to 1 – 4 digits                               | 1,3   |
| SHEX{1..4}           | Signed hexadecimal, optionally fixed to 1 – 4 digits                        | 1,2   |
| IHEX{1..4}           | Indicated hexadecimal, optionally fixed to 1 – 4 digits (\$ prefix)         | 1     |
| ISHEX{1..4}          | Signed, indicated hexadecimal, optionally fixed to 1 – 4 digits (\$ prefix) | 1,2   |
| BIN{1..16}           | Binary, optionally fixed to 1 – 16 digits                                   | 1     |
| SBIN{1..16}          | Signed binary, optionally fixed to 1 – 16 digits                            | 1,2   |
| IBIN{1..16}          | Indicated binary, optionally fixed to 1 – 16 digits (% prefix)              | 1     |
| ISBIN{1..16}         | Signed, indicated binary, optionally fixed to 1 – 16 digits (% prefix)      | 1,2   |

**Table 5.71:** OWOUT Conversion Formatters.

- 1 Fixed-digit formatters like DEC4 will pad the number with leading 0s if necessary; ex: DEC4 65 sends 0065. If a number is larger than the specified number of digits, the leading digits will be dropped; ex: DEC4 56422 sends 6422.
- 2 Signed modifiers work under two's complement rules.
- 3 The HEX modifier can be used for BCD to Decimal Conversion. See "Hex to BCD Conversion" on page 97.

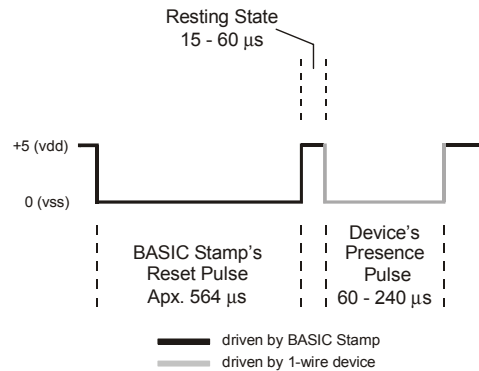
The 1-Wire protocol has a well-defined standard for transaction sequences. Every transaction sequence consists of four parts: 1) Initialization, 2) ROM Function Command, 3) Memory Function Command, and 4) Transaction/Data. Additionally, the ROM Function Command and Memory Function Command are always 8 bits wide (1 byte in size) and is sent least-significant-bit (LSB) first.

The Initialization part consists of a reset pulse (generated by the master) and will be followed by a presence pulse (generated by all slave devices). Figure 5.24 details the reset pulse generated by the BASIC Stamp and a typical presence pulse generated by a 1-Wire slave, in response.



## 5: BASIC Stamp Command Reference – OWOUT

**Figure 5.24:** OWOUT Reset and Presence Pulse.



This reset pulse is controlled by the lowest two bits of the *Mode* argument in the OWOUT command. It can be made to appear before the ROM Function Command (ex: *Mode* = 1), after the Transaction/Data portion (ex: *Mode* = 2), before and after the entire transaction (ex: *Mode* = 3) or not at all (ex: *Mode* = 0). See the section on *Mode*, above, for more information.

Following the Initialization part is the ROM Function Command. The ROM Function Command is used to address the desired 1-Wire device. Table 5.72 shows common ROM Function Commands. If only a single 1-wire device is connected, the Skip ROM command may be used to address it. If more than one 1-wire device is attached, the BASIC Stamp will ultimately have to address them individually using the Match ROM command.

**Table 5.72:** OWOUT ROM Function Commands.

| Command    | Value (in Hex) | Action                                                                                                                          |
|------------|----------------|---------------------------------------------------------------------------------------------------------------------------------|
| Read ROM   | \$33           | Reads the 64-bit ID of the 1-Wire device. This command can only be used if there is a single 1-Wire device on the line.         |
| Match ROM  | \$55           | This command, followed by a 64-bit ID, allows the BASIC Stamp to address a specific 1-Wire device.                              |
| Skip ROM   | \$CC           | Address a 1-Wire device without its 64-bit ID. This command can only be used if there is a single 1-wire device on the line.    |
| Search ROM | \$F0           | Reads the 64-bit IDs of all the 1-Wire devices on the line. A process of elimination is used to distinguish each unique device. |

The third part, the Memory Function Command, allows the BASIC Stamp to address specific memory locations, or features, of the 1-wire device.

## OWOUT – BASIC Stamp Command Reference

Refer to the 1-Wire device's data sheet for a list of the available Memory Function Commands.

Finally, the Transaction/Data section is used to read or write data to the 1-Wire device. The OWOUT command will write data at this point in the transaction. A write is accomplished by generating a low-pulse of a varying width to indicate a 0 or a 1. This is called a "Write Slot" and must be at least 60  $\mu\text{s}$  wide. Figure 5.25 shows typical Write Slots performed by the BASIC Stamp. See the OWIN command for information on Read Slots.

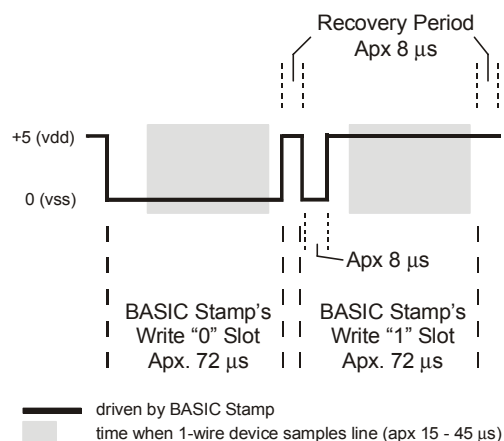


Figure 5.25: Example Write Slots.

The demo program uses a Dallas Semiconductor DS1820 Digital Thermometer device connected as follows. Note that the 4.7  $\text{k}\Omega$  pull-up resistor is required for proper operation.

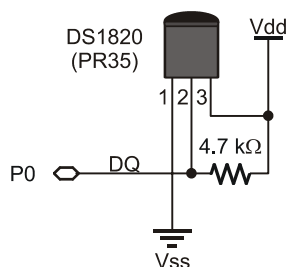


Figure 5.26: DS1820 Circuit.  
NOTE: The 4.7  $\text{k}\Omega$  resistor is required for proper operation.

## 5: BASIC Stamp Command Reference – OWOUT



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the \$STAMP directive accordingly.

### Demo Program (OWIN\_OWOUT.bsp)

```
' OWIN_OWOUT.bsp
' This program demonstrates interfacing to a Dallas Semiconductor DS1822
' 1-Wire Digital Thermometer chip using the BS2p's 1-Wire commands. Connect
' the BS2p, BS2pe or BS2px to the DS1822 as shown in the diagram in the
' OWIN or OWOUT command description. This program uses a simplified
' approach that ignores the fractional portion of the temperature.

' {$STAMP BS2p}
' {$PBASIC 2.5}

DQ PIN 0 ' 1-Wire buss pin

RdROM CON $33 ' read serial number
MatchROM CON $55 ' match SN -- for multiple devices
SkipROM CON $CC ' ignore SN -- use for one device
CvrtTmp CON $44 ' start temperature conversion
RdSP CON $BE ' read DS1822 scratch pad

tempIn VAR Word ' raw temperature
sign VAR tempIn.BIT11 ' 1 = negative temperature
tLo VAR tempIn.BYTE0
tHi VAR tempIn.BYTE1
tSign VAR Bit ' saved sign bit
tempC VAR Word ' final Celsius temp
tempF VAR Word ' final Fahrenheit temp

Main:
DO
 GOSUB Get_Temperature ' read temperature from DS1822
 DEBUG HOME, ' display
 "DS1822", CR,
 "-----", CR,
 SDEC tempC, " C ", CR,
 SDEC tempF, " F "
 PAUSE 1000
LOOP
END

Get_Temperature:
OWOUT DQ, 1, [SkipROM, CvrtTmp] ' send convert temperature command
DO
 PAUSE 25 ' wait on conversion
 PAUSE 25 ' small loop pad
 OWIN DQ, 4, [tempIn] ' check status (bit transfer)
LOOP UNTIL (tempIn) ' 1 when complete
OWOUT DQ, 1, [SkipROM, RdSP] ' read DS1822 scratch pad
OWIN DQ, 2, [tLo, tHi] ' get raw temp data
tSign = sign ' save sign bit
tempC = tempIn >> 4 ' round to whole degrees
tempC.BYTE1 = $FF * tSign ' correct twos complement bits
```

## ***OWOUT – BASIC Stamp Command Reference***

---

```
tempF = (ABS tempC) * 9 / 5 ' start F conversion
IF (tSign) THEN ' finish F conversion
 tempF = 32 - tempF ' C was negative
ELSE
 tempF = tempF + 32 ' C was positive
ENDIF
RETURN
```

## 5: BASIC Stamp Command Reference – PAUSE

### PAUSE

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



#### PAUSE Duration

#### Function

Pause the program (do nothing) for the specified *Duration*.

- **Duration** is a variable/constant/expression (0 – 65535) that specifies the duration of the pause. The unit of time for *Duration* is 1 millisecond.

#### Explanation

PAUSE delays the execution of the next program instruction for the specified number of milliseconds. For example:

```
Flash:
 LOW 0
 PAUSE 100
 HIGH 0
 PAUSE 100
 GOTO Flash
```

This code causes pin 0 to go low for 100 ms, then high for 100 ms. The delays produced by PAUSE are as accurate as the ceramic-resonator time base (on the BASIC Stamp modules),  $\pm 1$  percent. When you use PAUSE in timing-critical applications, keep in mind the relatively low speed of the PBASIC interpreter. This is the time required for the BASIC Stamp to read and interpret an instruction stored in the EEPROM.



#### Demo Program (PAUSE.bs2)

NOTE: This example program can be used with the BS1 and all BS2 models by changing the \$STAMP directive accordingly.

```
' PAUSE.bs2
' This program demonstrates the PAUSE command's time delays. Once a second,
' the program will put the message "Paused..." on the screen.
' {$STAMP BS2}

Main:
 DEBUG "Paused...", CR
 PAUSE 1000
 GOTO Main
```

## ***PAUSE – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – POLLIN

### POLLIN

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|



**POLLIN** *Pin, State*

#### Function

Specify a polled-input pin and active state.

- **Pin** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This I/O pin will be set to input mode.
- **State** is a variable/constant/expression (0 – 1) that specifies whether to poll the I/O pin for a low (0) or a high (1) level.

#### Quick Facts

Table 5.73: POLLIN Quick Facts.

|                                                                    | BS2p, BS2pe, and BS2px                                                                                                                                                                                                                                                                                                                                              |
|--------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Available actions in response to reaching the desired State</b> | 1) Nothing,<br>2) Set polled-output pins to a specified state,<br>3) Run another program (in a specified program-slot),<br>4) Wait (pause program execution) until desired <i>State</i> is reached,<br>5) Any combination of 2, 3 and 4, above.                                                                                                                     |
| <b>Special notes</b>                                               | <ul style="list-style-type: none"> <li>• The polled-input pins are monitored (polled) in-between each command within the PBASIC code.</li> <li>• On the BS2p40, polled-input pins can be defined on both Main I/O and Auxiliary I/O pins. These are all active regardless of which group the program happens to be using at the time of a polling event.</li> </ul> |
| <b>Useful SPRAM locations</b>                                      | Locations 128 – 135 hold polled interrupt status. See Table 5.77 in the POLLMODE command section for more information.                                                                                                                                                                                                                                              |
| <b>Related commands</b>                                            | POLLMODE, POLLOUT, POLLRUN and POLLWAIT                                                                                                                                                                                                                                                                                                                             |

#### Explanation

The POLLIN command is used to specify an input pin to monitor, or "poll", in-between instructions during the rest of the PBASIC program. The BASIC Stamp will then perform some activity (in-between instructions) when the specified *State* is detected. The activity performed depends on the POLLMODE, POLLOUT and POLLRUN commands.

The "polling" commands allow the BASIC Stamp to respond to certain I/O pin events at a faster rate than what is normally possible through manual PBASIC programming. The term "poll" comes from the fact that the BASIC Stamp module's interpreter periodically checks the state of the designated polled-input pins. It "polls" these pins after the end of each PBASIC command and before it reads the next PBASIC command from the

## ***POLLIN – BASIC Stamp Command Reference***

---

user program; giving the appearance that it is polling "in the background". This feature should not be confused with the concept of interrupts, as the BASIC Stamp *does not support true interrupts*.

The following is an example of the POLLIN command:

A SIMPLE POLLIN EXAMPLE.

```
POLLIN 0, 0
POLLMODE 2
```

The POLLIN command in the above code will cause the BASIC Stamp to set I/O pin 0 to an input mode and get ready to poll it for a low (0) state. The BASIC Stamp will not actually start polling until it is set to the appropriate mode, however. The second line, POLLMODE, initiates the polling process (see the POLLMODE description for more information). From then on, as the BASIC Stamp executes the rest of the program, it will check for a low level (logic 0) on I/O pin 0, in-between instructions.

In the code above, no obvious action will be noticed since we didn't tell the BASIC Stamp what to do when it detects a change on the I/O pin. One possible action the BASIC Stamp can be instructed to take is to change the state of an output, called a polled-output. Take a look at the next example:

SETTING ONE OF THE POSSIBLE ACTIONS:  
POLLED-OUTPUTS

```
POLLIN 0, 0
POLLOUT 1, 1
POLLMODE 2
```

```
Main:
 DEBUG "Looping...", CR
 GOTO Main
```

In this example, in addition to an endless loop, we've added another polling command called POLLOUT (see the POLLOUT description for more information). Our POLLOUT command tells the BASIC Stamp to set I/O pin 1 to an output mode and set it high (1) when it detects the desired poll state. The poll state is the low (0) level on I/O pin 0 that POLLIN told it to look for. If the polled-input pin is high, it will set polled-output pin 0 to low (0), instead.

Once the program reaches the endless loop, at *Main*, it will continuously print "Looping..." on the PC screen. In between reading the DEBUG command and the GOTO command (and vice versa) it will check polled-input pin 0 and set polled-output pin 1 accordingly. In this case, when I/O pin 0 is set low, the BASIC Stamp will set I/O pin 1 high. When I/O



## 5: BASIC Stamp Command Reference – POLLIN

---

pin 0 is set low, the BASIC Stamp will set I/O pin 1 high. It will continue to perform this operation, in-between each command in the loop, endlessly.

THE BASIC STAMP "REMEMBERS" THE POLLING CONFIGURATION FOR THE DURATION OF THE PBASIC PROGRAM.

It's important to note that, in this example, only the DEBUG and GOTO commands are being executed over and over again. The first three lines of code are only run once, yet their effects are "remembered" by the BASIC Stamp throughout the rest of the program.

FOR COMPARISON: ACHIEVING THE SAME EFFECTS WITHOUT THE POLLING COMMANDS.

If the polling commands were not used, the program would have to look like the one below in order to achieve the same effect.

```
INPUT 0
OUTPUT 1

Main:
 OUT1 = ~IN0
 DEBUG "Looping...", CR
 OUT1 = ~IN0
 GOTO Main
```

In this example, we create the inverse relationship of input pin 0 and output pin 1 manually, in-between the DEBUG and GOTO lines. Though the effects are the same as when using the polling commands, this program actually takes a little longer to run and consumes 7 additional bytes of program (EEPROM) space. Clearly, using the polling commands is more efficient.

USING MULTIPLE POLLED-INPUT AND POLLED-OUTPUT PINS.

You can have as many polled-input and polled-output pins as you have available. If multiple polled-input pins are defined, any one of them can trigger changes on the polled-output pins that are also defined. For example:

```
POLLIN 0, 0
POLLIN 1, 0
POLLOUT 2, 1
POLLOUT 3, 1
POLLMODE 2

Main:
 DEBUG "Looping...", CR
 GOTO Main
```

This code sets I/O pins 0 and 1 to polled-input pins (looking for a low (0) state) and sets I/O pins 2 and 3 to polled-output pins (with a high-active

## POLLIN – BASIC Stamp Command Reference

state). If either I/O pin 0 or 1 goes low, the BASIC Stamp will set I/O pins 2 and 3 high. This works similar to a logical OR operation. The truth table below shows all the possible states of these two polled-input pins and the corresponding states the BASIC Stamp will set the polled-output pins to.

| Polled-Inputs |   | Polled-Outputs |   |
|---------------|---|----------------|---|
| 0             | 1 | 2              | 3 |
| 0             | 0 | 0              | 0 |
| 0             | 1 | 1              | 1 |
| 1             | 0 | 1              | 1 |
| 1             | 1 | 1              | 1 |

**Table 5.74:** Polled-Inputs / Polled-Outputs Truth Table.

Normally, any polled-output pins reflect the state changes continuously, as described above. The POLLMODE command supports another feature, however, where the polled-output pins will latch the active state; they will change only once (when the poll state is reached) and stay in the new state until the PBASIC program tells it to change again. See the POLLMODE description for more information.

POLLED-OUTPUTS CAN BE "LATCHED" ALSO.

Other possible actions in response to polled-input states are: 1) Running another program (in a specified program slot), 2) Waiting (pausing program execution with or without low-power mode) until the poll state is reached, or 3) Any combination of the above-mentioned actions.

### Demo Program (POLL.bsp)

```
' POLL.bsp
' This program demonstrates POLLIN, POLLOUT, and the use of the POLLMODE
' instruction. Connect active-low inputs to pins 0, 1, 2, and 3. Then
' connect an LED to pin 7. The program will print "." to the Debug
' window until one of the alarm buttons are pressed. This will cause
' the termination of the main loop. At this point the program will
' save the latched bits, clear them (and the polling process), then
' report the input(s) that triggered the alarm.

' {$STAMP BS2p}
' {$PBASIC 2.5}

FDoor PIN 0
BDoor PIN 1
Patio PIN 2
Rst PIN 3
AlarmLed PIN 7

alarms VAR Byte ' alarm bits
```



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the \$STAMP directive accordingly.

## 5: BASIC Stamp Command Reference – POLLIN

---

```
idx VAR Nib ' loop control

Setup:
 POLLIN FDoor, 0 ' define alarm inputs
 POLLIN BDoor, 0
 POLLIN Patio, 0
 POLLOUT AlarmLed, 1 ' alarm indicator
 POLLMODE 10 ' activate latched polling
 DEBUG CLS,
 "Alarms Activated", CR

Main:
 DO
 DEBUG " ." ' foreground activity
 PAUSE 50
 LOOP UNTIL (AlarmLed = 1) ' loop until LED is on
 GET 128, alarms ' get alarm bits
 POLLMODE 0 ' deactivate polling

Report:
 DEBUG CLS, ' alarms report
 "Front Door : ", CR,
 "Back Door : ", CR,
 "Patio : ", CR

 FOR idx = 0 TO 2 ' scan alarm bits
 DEBUG CRSRXY, 13, idx ' move cursor
 IF (alarms.LOWBIT(idx)) THEN ' report each bit status
 DEBUG "Alarm", CR
 ELSE
 DEBUG "- ", CR
 ENDIF
 NEXT
 DEBUG CR, "Press RESET to clear..."
 DO : LOOP UNTIL (Rst = 0) ' wait until Rst pressed
 GOTO Setup
END
```

## ***POLLIN – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – POLLMODE

### POLLMODE

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|



### POLLMODE Mode

#### Function

Specify a polled command mode.

- **Mode** is a variable/constant/expression (0 – 15) that indicates the mode in which to process the polled command configuration.

#### Quick Facts

**Table 5.75:** POLLMODE Quick Facts.

|                               | BS2p, BS2pe, and BS2px                                                                                                                                                                                                                                                                                                                                                               |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Special Notes</b>          | <ul style="list-style-type: none"> <li>• Polled-output pins will either change states continuously, just once or not at all, depending on the POLLMODE command.</li> <li>• A poll-mode of 2 or 4 is required for a POLLWAIT command to work.</li> <li>• If both polled-outputs and polled-run are active, the polled-output event will occur before the polled-run event.</li> </ul> |
| <b>Useful SPRAM Locations</b> | Locations 128 – 135 hold polled interrupt status. See Table 5.77 for more information.                                                                                                                                                                                                                                                                                               |
| <b>Related Commands</b>       | POLLIN, POLLOUT, POLLRUN and POLLWAIT                                                                                                                                                                                                                                                                                                                                                |

#### Explanation

The POLLMODE command is used to specify the mode in which polling events and activities are processed. This activity will occur in-between instructions during the rest of the PBASIC program.

The "polling" commands allow the BASIC Stamp to respond to certain I/O pin events at a faster rate than what is normally possible through manual PBASIC programming. The term "poll" comes from the fact that the BASIC Stamp's interpreter periodically checks the state of the designated polled-input pins. It "polls" these pins after the end of each PBASIC command and before it reads the next PBASIC command from the user program; giving the appearance that it is polling "in the background". This feature should not be confused with the concept of interrupts, as the BASIC Stamp *does not support true interrupts*.

The POLLMODE command sets one of 15 possible modes for the polling commands. It is used mainly before and/or after any POLLIN, POLLOUT or POLLRUN commands to disable and enable the polling features as desired. Table 5.76 shows the mode values and their effect.

## POLLMODE – BASIC Stamp Command Reference

| Mode           | Effect                                                                  |
|----------------|-------------------------------------------------------------------------|
| 0              | Deactivate polling, clear polled-input and output configuration.        |
| 1              | Deactivate polling, save polled-input and output configuration.         |
| 2              | Activate polling with polled-output action (and polled-wait) only.      |
| 3 <sup>1</sup> | Activate polling with polled-run action only.                           |
| 4 <sup>2</sup> | Activate polling with polled-output/polled-wait and polled-run actions. |
| 5 <sup>3</sup> | Clear polled-input configuration.                                       |
| 6 <sup>3</sup> | Clear polled-output configuration.                                      |
| 7 <sup>3</sup> | Clear polled-input and output configuration.                            |
| 8 – 15         | Same as 0 – 7 except polled-output states are latched.                  |

**Table 5.76:** POLLMODE Mode Values.

<sup>1</sup> After the polled-run action occurs, the mode switches to 1 (deactivated, saved)

<sup>2</sup> After the polled-run action occurs, the mode switches to 2 (activated, outputs)

<sup>3</sup> These modes do not override the previous mode. Also, the output state of polled-outputs does not change as a result of these modes.

The polled-run modes, 3 and 4, are unique. As soon as the polled-run action occurs, the mode switches to 1 (deactivated, saved) or 2 (activated, outputs), respectively. This is so that the BASIC Stamp doesn't continuously go to the start of the designated program slot while the polled-inputs are in the desired poll state. Without this "one shot" feature, your program would appear to lock-up as long as the polled-inputs are in the designated state.

The clear configuration modes, 5, 6 and 7, are also unique. These modes do not override the previous mode. For example, if polled-inputs, polled-outputs and a polled-run configuration was set and the mode was set to 4 (activated, outputs and run) and later the program issued a POLLMODE 6 command, the polled-output configuration would be cleared but the mode would switch back to 4... still allowing the run action. This also means if, later still, the program issues a POLLOUT command, this polled-output would take effect immediately (since the mode is still 4). Also note that these modes do not change the output state of previously defined polled-output pins.

The POLLMODE command determines what action, if any, will occur in response to a polled-input event. This command works in conjunction with the POLLIN, POLLOUT and POLLRUN commands. The following is an example of the POLLMODE command:

A SIMPLE POLLMODE EXAMPLE.

## 5: BASIC Stamp Command Reference – POLLMODE

---

```
POLLIN 0, 0
POLLOUT 1, 1
POLLMODE 2

Main:
 DEBUG "Looping...", CR
 GOTO Main
```

In this example, the first two lines configure I/O pin 0 as a polled-input (looking for a low state) and I/O pin 1 as a polled-output (going high if I/O pin 0 goes low, and vice versa). The third line, POLLMODE, initiates the polling process and configures polled-outputs to be active. From then on, as the BASIC Stamp executes the rest of the program, it will check for a low level (logic 0) on I/O pin 0, in-between instructions and will set I/O pin 1 accordingly.

If, in the above example, the poll mode was set to 1 (which means deactivate polling but save configuration) I/O pins 0 and 1 would still be defined the same way, and I/O pin 1 would still be set to output mode, but no polling would take place during the rest of the program.

Here's another example that demonstrates mode 1 (deactivate but save configuration).

```
POLLIN 0, 0
POLLOUT 1, 1
POLLMODE 2

DEBUG "Polling configured.", CR

Main:
 POLLMODE 1
 DEBUG "No polling allowed here...", CR
 PAUSE 1000
 POLLMODE 2

Poll_Now:
 DEBUG "Polling now...", CR
 GOTO Poll_Now
```

In this case, polling is configured and activated before "Polling configured" is printed on the screen. Once we reach the *Main* routine, however, polling is disabled (via the POLLMODE 1 command) and no polling occurs during the printing of "No polling allowed here..." or during the 1 second pause afterward. Finally, polling is activated again, and since the configuration was saved (because of mode 1, before) the polling activity

## ***POLLMODE – BASIC Stamp Command Reference***

---

acts just like it did initially for the remainder of the program. The ability to temporarily disable polling, without changing the configuration, can be a powerful feature for certain "critical" parts of a program.

The following example contains two programs. The first should be downloaded into program slot 0 and the second into program slot 1.

```
' Pgm #1 (Slot 0) -----
```

```
POLLIN 0, 0
POLLOUT 1, 1
POLLRUN 1
POLLMODE 4
```

```
Main:
```

```
 DEBUG "Running Program 1", CR
 GOTO Main
```

```
' Pgm #2 (Slot 1) -----
```

```
Main:
```

```
 DEBUG "Running Program 2", CR
 GOTO Main
```

In this example (containing two programs; one is slot 0 and the other in slot 1) program 1 (slot 0) will configure polled-input pin 0 to detect a low state and polled-output 1 to go high in response. Program 1 also configures a polled-run activity (see the POLLRUN description for more information) to run the program in slot 1. The POLLMODE setting activates the polled-output and the polled-run. Then, program 1 continuously prints "Running Program 1" on the PC screen.

Once I/O pin 0 goes low, however, the BASIC Stamp will set I/O pin 1 high, then execution will be switched to the program in slot 1 (program 2). Program 2 will continuously print "Running Program 2" on the screen. From this point forward, I/O pin 1 will continue to be set low and high in response to changes occurring on I/O pin 0, but the polled-run activity is disabled and the BASIC Stamp endlessly runs the code in program 2's *Main* routine.

The highest locations of Scratch Pad RAM contain run-time information about polled interrupts. See Table 5.77 below.



**Table 5.77:** Special Purpose Scratch Pad RAM Locations.

| Location | BS2p and BS2pe                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 127      | Bits 0-3, Active program slot #. Bits 4-7, program slot for READ and WRITE operations.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 128      | Polled input trigger status of Main I/O pins 0-7 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| 129      | Polled input trigger status of Main I/O pins 8-15 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 130      | Polled input trigger status of Auxiliary I/O pins 0-7 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 131      | Polled input trigger status of Auxiliary I/O pins 8-15 (0 = not triggered, 1 = triggered).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| 132      | Bits 0-3: Polled-interrupt mode, set by POLLMODE                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| 133      | Bits 0-2: Polled-interrupt "run" slot, set by POLLRUN.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| 134      | Bit 0: Active I/O group; 0 = Main I/O, 1 = Auxiliary I/O.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 135      | Bit 0: Polled-output status (set by POLLMODE); 0 = disabled, 1 = enabled.<br>Bit 1: Polled-input status; 0 = none defined, 1 = at least one defined.<br>Bit 2: Polled-run status (set by POLLMODE); 0 = disabled, 1 = enabled.<br>Bit 3: Polled-output latch status; 0 = real-time mode, 1 = latch mode.<br>Bit 4: Polled-input state; 0 = no trigger, 1 = triggered.<br>Bit 5: Polled-output latch state; 0 = nothing latched, 1 = signal latched.<br>Bit 6: Poll-wait state; 0 = No Event, 1 = Event Occurred.<br>(Cleared by POLLMODE only).<br>Bit 7: Polling status; 0 = not active, 1 = active. |



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the \$STAMP directive accordingly.

## Demo Program (POLL.bsp)

```
' POLL.bsp
' This program demonstrates POLLIN, POLLOUT, and the use of the POLLMODE
' instruction. Connect active-low inputs to pins 0, 1, 2, and 3. Then
' connect an LED to pin 7. The program will print "." to the Debug
' window until one of the alarm buttons are pressed. This will cause
' the termination of the main loop. At this point the program will
' save the latched bits, clear them (and the polling process), then
' report the input(s) that triggered the alarm.

' {$STAMP BS2p}
' {$PBASIC 2.5}

FDoor PIN 0
BDoor PIN 1
Patio PIN 2
Rst PIN 3
AlarmLed PIN 7

alarms VAR Byte ' alarm bits
idx VAR Nib ' loop control
```

## ***POLLMODE – BASIC Stamp Command Reference***

---

```
Setup:
 POLLIN FDoor, 0 ' define alarm inputs
 POLLIN BDoor, 0
 POLLIN Patio, 0
 POLLOUT AlarmLed, 1 ' alarm indicator
 POLLMODE 10 ' activate latched polling
 DEBUG CLS,
 "Alarms Activated", CR

Main:
 DO
 DEBUG ".", ' foreground activity
 PAUSE 50
 LOOP UNTIL (AlarmLed = 1) ' loop until LED is on
 GET 128, alarms ' get alarm bits
 POLLMODE 0 ' deactivate polling

Report:
 DEBUG CLS, ' alarms report
 "Front Door : ", CR,
 "Back Door : ", CR,
 "Patio : ", CR

 FOR idx = 0 TO 2 ' scan alarm bits
 DEBUG CRSRXY, 13, idx ' move cursor
 IF (alarms.LOWBIT(idx)) THEN ' report each bit status
 DEBUG "Alarm", CR
 ELSE
 DEBUG "-", CR
 ENDIF
 NEXT
 DEBUG CR, "Press RESET to clear..."
 DO : LOOP UNTIL (Rst = 0) ' wait until Rst pressed
 GOTO Setup
END
```

## 5: BASIC Stamp Command Reference – POLLOUT

### POLLOUT

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|



**POLLOUT** *Pin, State*

#### Function

Specify a polled-output pin and active state.

- **Pin** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This I/O pin will be set to output mode.
- **State** is a variable/constant/expression (0 – 1) that specifies whether to set the I/O pin low (0) or high (1) when a polled-input pin changes to its poll state.

#### Quick Facts

**Table 5.78:** POLLOUT Quick Facts.

|                               | BS2p, BS2pe, and BS2px                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Special Notes</b>          | <ul style="list-style-type: none"> <li>• The POLLOUT command will immediately change the I/O pin to an output mode and set its level opposite to that of <i>State</i>, regardless of the polled-input states or the polled mode.</li> <li>• Polled-output pins will either change states continuously, just once or not at all, depending on the POLLMODE command.</li> <li>• On the BS2p40, polled-output pins can be defined on both main I/O and auxiliary I/O pins. These are all active regardless of which group the program happens to be using at the time of a polling event.</li> <li>• If both polled-outputs and polled-run are active, the polled-output event will occur before the polled-run event.</li> </ul> |
| <b>Useful SPRAM Locations</b> | Locations 128 – 135 hold polled interrupt status. See Table 5.77 in the POLLMODE command section for more information.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Related Commands</b>       | POLLMODE, POLLIN, POLLRUN and POLLWAIT                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

#### Explanation

The POLLOUT command is used to specify an output pin that changes states in response to changes on any of the defined polled-input pins. This activity will occur in-between instructions during the rest of the PBASIC program.

The "polling" commands allow the BASIC Stamp to respond to certain I/O pin events at a faster rate than what is normally possible through manual PBASIC programming. The term "poll" comes from the fact that the BASIC Stamp's interpreter periodically checks the state of the designated polled-input pins. It "polls" these pins after the end of each PBASIC command and before it reads the next PBASIC command from the user program; giving the appearance that it is polling "in the background".

## ***POLLOUT – BASIC Stamp Command Reference***

---

This feature should not be confused with the concept of interrupts, as the BASIC Stamp *does not support true interrupts*.

The POLLOUT command achieves one of three possible actions in response to a polled-input event. This command works in conjunction with the POLLIN and POLLMODE commands. The following is an example of the POLLOUT command: A SIMPLE POLLOUT EXAMPLE.

```
POLLIN 0, 0
POLLOUT 1, 1
POLLMODE 2

Main:
 DEBUG "Looping...", CR
 GOTO Main
```

In this example, the POLLOUT command tells the BASIC Stamp to set I/O pin 1 to an output mode and set it high (1) when it detects the desired poll state. The poll state is the low (0) level on I/O pin 0 that POLLIN told it to look for. If the polled-input pin is high, the BASIC Stamp will set polled-output pin 1 to low (0), instead. The BASIC Stamp will not actually start polling until it is set to the appropriate mode, however. The third line, POLLMODE, initiates the polling process (see the POLLMODE description for more information). From then on, as the BASIC Stamp executes the rest of the program, it will check for a low level (logic 0) on I/O pin 0, in-between instructions.

Once the program reaches the endless loop, it will continuously print "Looping..." on the PC screen. In between reading the DEBUG command and the GOTO command (and vice versa) it will check polled-input pin 0 and set polled-output pin 1 accordingly. In this case, when I/O pin 0 is set high, the BASIC Stamp will set I/O pin 1 low. When I/O pin 0 is set low, the BASIC Stamp will set I/O pin 1 high. It will continue to perform this operation, in-between each command in the loop, endlessly.

It's important to note that in this example only the DEBUG and GOTO commands are being executed over and over again. The first three lines of code are only run once, yet their effects are "remembered" by the BASIC Stamp throughout the rest of the program.

THE BASIC STAMP "REMEMBERS" THE POLLING CONFIGURATION FOR THE DURATION OF THE PBASIC PROGRAM.

If the polling commands were not used, the program would have to look like the one below in order to achieve the same effect.

FOR COMPARISON: ACHIEVING THE SAME EFFECTS WITHOUT THE POLLING COMMANDS.

## 5: BASIC Stamp Command Reference – POLLOUT

```
INPUT 0
OUTPUT 1

Main:
 OUT1 = ~IN0
 DEBUG "Looping...", CR
 OUT1 = ~IN0
 GOTO Main
```

In this example, we create the inverse relationship of input pin 0 and output pin 1 manually, in-between the DEBUG and GOTO lines. Though the effects are the same as when using the polling commands, this program actually takes a little longer to run and consumes 7 additional bytes of program (EEPROM) space. Clearly, using the polling commands is more efficient.

USING MULTIPLE POLLED-INPUT AND  
POLLED-OUTPUT PINS.

You can have as many polled-input and polled-output pins as you have available. If multiple polled-output pins are defined, all of them change in response to changes on the polled-input pins. For example:

```
POLLIN 0, 0
POLLOUT 1, 0
POLLOUT 2, 1
POLLOUT 3, 1
POLLMODE 2

Main:
 DEBUG "Looping...", CR
 GOTO Main
```

This code sets up I/O pin 0 as a polled-input pin (looking for a low (0) state) and sets I/O pins 1, 2 and 3 to polled-output pins. Polled-output pin 1 is set to a low-active state and pins 2 and 3 are set to a high-active state. If I/O pin 0 goes low, the BASIC Stamp will set I/O pin 1 low and I/O pins 2 and 3 high. The table below shows the two possible states of the polled-input pin and the corresponding states the BASIC Stamp will set the polled-output pins to.

**Table 5.79:** POLLOUT Truth Table.

| Polled-Input | Polled-Outputs |   |   |
|--------------|----------------|---|---|
| 0            | 1              | 2 | 3 |
| 1            | 1              | 0 | 0 |
| 0            | 0              | 1 | 1 |

POLLED-OUTPUTS CAN BE "LATCHED"  
ALSO.

Normally, any polled-output pins reflect the state changes continuously, as described above. The POLLMODE command supports another feature,

## ***POLLOUT – BASIC Stamp Command Reference***

---

however, where the polled-output pins will latch the active state; they will change only once (when the poll state is reached) and stay in the new state until the PBASIC program tells it to change again. See the POLLMODE description for more information.

A clever use of the "latched" feature is to set a polled input on the same pin as a polled output. This leaves the pin in an input state, and the "latched" state of that input pin will be saved in the pin's output latch bit (which is otherwise not used because the pin was last set to input mode). This gives us the advantage of having a run-time readable bit that indicates whether the input pin changed state while we were busy. For example, suppose an application needed to respond in some way if an input pin goes high, but it doesn't need to respond immediately, and the other tasks should not be interrupted. In essence, we need a way to know if the pin has gone high since the last time we checked it. Look at this example:

A CLEVER TRICK WITH POLLOUT AND THE "LATCHED" FEATURE.

```
alarm VAR OUT0
idx VAR Byte
```

Setup:

```
POLLOUT 0, 1
POLLIN 0, 0
POLLMODE 10
```

Work:

```
FOR idx = 1 TO 25
 DEBUG "Working...", CR
NEXT
IF (Alarm = 0) THEN Work
```

Respond:

```
DEBUG CR, "Hey, the Alarm was tripped!", CR
PAUSE 1000
POLLMODE 10
GOTO Work
```

Here, we set I/O pin 0 to a polled-output, then immediately set it to a polled-input. Then we set the polled-mode to latch the polled-outputs. Afterwards, the program performs some work, and once in a while, checks the state of OUT0 (named *Alarm* in the program). If *Alarm* is 0, I/O pin 0 was never seen to go high. If, however, *Alarm* is 1, I/O pin 0 must have gone high while the program was doing other work, and now it can respond in the proper manner. This even works if the input pin had gone high and then low again before we check it (as long as it was high at some point in between the instructions in our *Work* routine).

## 5: BASIC Stamp Command Reference – POLLOUT

It is important to note that during the time between the POLLOUT and POLLIN commands, I/O pin 0 will be set to an output direction. This can cause a temporary short within the circuitry connected to I/O pin 0, so it is vital that a large enough series resistor (perhaps 100 ohms or greater) be inserted on that pin to protect the external device and the BASIC Stamp.



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the \$STAMP directive accordingly.

### Demo Program (POLL.bsp)

```
' POLL.bsp
' This program demonstrates POLLIN, POLLOUT, and the use of the POLLMODE
' instruction. Connect active-low inputs to pins 0, 1, 2, and 3. Then
' connect an LED to pin 7. The program will print "." to the Debug
' window until one of the alarm buttons are pressed. This will cause
' the termination of the main loop. At this point the program will
' save the latched bits, clear them (and the polling process), then
' report the input(s) that triggered the alarm.

' {$STAMP BS2p}
' {$PBASIC 2.5}

FDoor PIN 0
BDoor PIN 1
Patio PIN 2
Rst PIN 3
AlarmLed PIN 7

alarms VAR Byte ' alarm bits
idx VAR Nib ' loop control

Setup:
 POLLIN FDoor, 0 ' define alarm inputs
 POLLIN BDoor, 0
 POLLIN Patio, 0
 POLLOUT AlarmLed, 1 ' alarm indicator
 POLLMODE 10 ' activate latched polling
 DEBUG CLS,
 "Alarms Activated", CR

Main:
 DO
 DEBUG "." ' foreground activity
 PAUSE 50
 LOOP UNTIL (AlarmLed = 1) ' loop until LED is on
 GET 128, alarms ' get alarm bits
 POLLMODE 0 ' deactivate polling

Report:
 DEBUG CLS, ' alarms report
```

## ***POLLOUT – BASIC Stamp Command Reference***

---

```
 "Front Door : ", CR,
 "Back Door : ", CR,
 "Patio : ", CR

FOR idx = 0 TO 2 ' scan alarm bits
 DEBUG CRSRXY, 13, idx ' move cursor
 IF (alarms.LOWBIT(idx)) THEN ' report each bit status
 DEBUG "Alarm", CR
 ELSE
 DEBUG "-", CR
 ENDIF
NEXT
DEBUG CR, "Press RESET to clear..."
DO : LOOP UNTIL (Rst = 0) ' wait until Rst pressed
GOTO Setup
END
```



## 5: BASIC Stamp Command Reference – POLLRUN

### POLLRUN

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|



#### POLLRUN ProgramSlot

#### Function

Specify a program to run upon a polled-input event.

- **ProgramSlot** is a variable/constant/expression (0 – 7) that specifies the program slot to run when a polled-input event occurs.

#### Quick Facts

Table 5.80: POLLRUN Quick Facts.

|                               | BS2p, BS2pe, and BS2px                                                                                                                                                 |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Default ProgramSlot</b>    | The default polled-run slot is 0. If no POLLRUN command is given and a poll mode of 3 or 4 is set, the program in slot 0 will run in response to a polled-input event. |
| <b>Special Notes</b>          | <ul style="list-style-type: none"> <li>• If both polled-outputs and polled-run are active, the polled-output event will occur before the polled-run event.</li> </ul>  |
| <b>Useful SPRAM locations</b> | Locations 128 – 135 hold polled interrupt status. See Table 5.77 in the POLLMODE command section for more information.                                                 |
| <b>Related commands</b>       | POLLMODE, POLLIN, POLLOUT, POLLWAIT and RUN                                                                                                                            |

#### Explanation

The POLLRUN command is used to specify a program slot to run in response to a polled event. This activity can occur in between any two instructions within the rest of the PBASIC program.

The "polling" commands allow the BASIC Stamp to respond to certain I/O pin events at a faster rate than what is normally possible through manual PBASIC programming. The term "poll" comes from the fact that the BASIC Stamp's interpreter periodically checks the state of the designated polled-input pins. It "polls" these pins after the end of each PBASIC command and before it reads the next PBASIC command from the user program; giving the appearance that it is polling "in the background". This feature should not be confused with the concept of interrupts, as the BASIC Stamp *does not support true interrupts*.

## POLLRUN – BASIC Stamp Command Reference

---

The following is a simple example of the POLLRUN command.

A SIMPLE POLLRUN EXAMPLE.

```
POLLIN 0, 0
POLLRUN 1
POLLMODE 3
```

Main:

```
 DEBUG "Waiting in Program Slot 0...", CR
 GOTO Main
```

The first line of the above code will set up I/O pin 0 as a polled-input pin looking for a low (0) state. The second line, POLLRUN, tells the BASIC Stamp that when I/O pin 0 goes low, it should switch execution over to the program residing in program slot 1. The third line, POLLMODE, activates the polled-run configuration.

Once the BASIC Stamp reaches the *Main* routine, it will continuously print "Waiting in Program Slot 0..." on the PC screen. In between reading the DEBUG and GOTO commands, however, the BASIC Stamp will poll I/O pin 0 and check for a high or low state. If the state of pin 0 is high, it will do nothing and continue as normal. If the state of pin 0 is low, it will switch execution over to the program in slot 1 (the second program is not shown in this example). The switch to another program slot works exactly like with the RUN command; the designated program is run and the BASIC Stamp does not "return" to the previous program (similar to a GOTO command).

Note that in order for the polled-run activity to occur, the poll mode must be set to either 3 or 4 (the two modes that activate polled-run). Also note, that the polled-run modes, 3 and 4, are unique. As soon as the polled-run action occurs, the mode switches to 1 (deactivated, saved) or 2 (activated, outputs), respectively. This is so that the BASIC Stamp doesn't continuously go to the start of the designated program slot while the polled-inputs are in the desired poll state. Without this "one shot" feature, your program would appear to lock-up as long as the polled-inputs are in the designated state.

After the program switch takes place, the *ProgramSlot* value is maintained. Any future change to poll mode 3 or 4, without another POLLRUN command, will result in the previously defined program slot being used.

## 5: BASIC Stamp Command Reference – POLLRUN



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the \$STAMP directive accordingly.

### Demo Program (POLLRUN0.bsp)

```
' POLLRUN0.bsp
' This program demonstrates the POLLRUN command. It is intended to be
' downloaded to program slot 0, and the program called POLLRUN1.bsp
' should be downloaded to program slot 1. I/O pin 0 is set to watch for
' a low signal. Once the Main routine starts running, the program
' continuously prints it's program slot number to the screen. If I/O
' pin 0 goes low, the program in program slot 1 (which should be
' POLLRUN1.bsp) is run.

' {$STAMP BS2p, POLLRUN1.BSP}
' {$PBASIC 2.5}

pgmSlot VAR Byte

Setup:
 POLLIN 0, 0 ' polled-input, look for 0
 POLLRUN 1 ' run slot 1 on polled activation
 POLLMODE 3 ' enable polling

Main:
 GET 127, pgmSlot
 DEBUG "Running Program #", DEC pgmSlot.LOWNIB, CR
 GOTO Main
END
```



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the \$STAMP directive accordingly.

### Demo Program (POLLRUN1.bsp)

```
' POLLRUN1.bsp
' This program demonstrates the POLLRUN command. It is intended to be
' downloaded to program slot 1, and the program called POLLRUN0.bsp
' should be downloaded to program slot 0. This program is run when
' program 0 detects a low on I/O pin 0 via the polled commands.

' {$STAMP BS2p}
' {$PBASIC 2.5}

pgmSlot VAR Byte

Main:
 GET 127, pgmSlot
 DEBUG "Running Program #", DEC pgmSlot.LOWNIB, CR
 GOTO Main
END
```

## ***POLLRUN – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – POLLWAIT

### POLLWAIT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



### POLLWAIT *Duration*

#### Function

Pause program execution, in a low-power mode, in units of *Duration* until any polled-input pin reaches the desired poll state.

- **Duration** is a variable/constant/expression (0 – 8) that specifies the duration of the low-power state. The duration is  $(2^{\text{Duration}}) * 18$  ms. Table 5.82 on page 336 indicates the low-power length for any given *Duration*. Using 8 as the *Duration* is a special case; the BS2p, BS2pe, and BS2px will not go into low-power mode and will respond quicker to polled-inputs.

#### Quick Facts

**Table 5.81:** POLLWAIT Quick Facts.

|                                                    | BS2p                                                                                                                                                                                                                                                                                                                            | BS2pe                 | BS2px                 |
|----------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|-----------------------|
| <b>Current draw during POLLWAIT</b>                | 350 $\mu$ A                                                                                                                                                                                                                                                                                                                     | 36 $\mu$ A            | 450 $\mu$ A           |
| <b>Response Time with <i>Duration</i> set to 8</b> | Less than 160 $\mu$ s                                                                                                                                                                                                                                                                                                           | Less than 250 $\mu$ s | Less than 100 $\mu$ s |
| <b>Accuracy of NAP</b>                             | –50 to 100% ( $\pm 10\%$ @ 75°F with stable power supply)                                                                                                                                                                                                                                                                       |                       |                       |
| <b>Special Notes</b>                               | <ul style="list-style-type: none"> <li>• Poll mode must be 2 or 4 and at least one polled-input must be set to activate POLLWAIT (POLLWAIT will be ignored otherwise).</li> <li>• If both polled-wait and polled-run are active, the polled-run event will occur immediately after the polled-wait detects an event.</li> </ul> |                       |                       |
| <b>Useful SPRAM Locations</b>                      | Locations 128 – 135 hold polled interrupt status. See Table 5.77 in the POLLMODE command section for more information.                                                                                                                                                                                                          |                       |                       |
| <b>Related Commands</b>                            | POLLMODE, POLLIN, POLLOUT, POLLRUN, END, NAP and SLEEP                                                                                                                                                                                                                                                                          |                       |                       |

#### Explanation

The POLLWAIT command is used to pause program execution and go into a low-power state until any polled-input pin reaches the desired poll state.

The "polling" commands allow the BASIC Stamp to respond to certain I/O pin events at a faster rate than what is normally possible through manual PBASIC programming. The term "poll" comes from the fact that the BASIC Stamp's interpreter periodically checks the state of the designated

## POLLWAIT – BASIC Stamp Command Reference

---

polled-input pins. It "polls" these pins after the end of each PBASIC command and before it reads the next PBASIC command from the user program; giving the appearance that it is polling "in the background". This feature should not be confused with the concept of interrupts, as the BASIC Stamp *does not support true interrupts*.

The POLLWAIT command is unique among the polling commands in that it actually causes execution to halt, until a polled-input pin event occurs. The *Duration* argument is similar to that of the NAP command; using the values 0 to 7 specifies the duration of the low-power period. After the low-power period is over, the BASIC Stamp polls the polled-input pins and determines if any meet the desired poll state. If no polled-input is in the desired state (as set by POLLIN command) the BASIC Stamp goes back into low-power mode, again, for the same duration as before. If any polled-input is in the desired state, however, the BASIC Stamp will continue execution with the next line of code.

A *Duration* of 8 makes the BASIC Stamp pause execution in normal running mode (not low-power mode) until a polled-input event occurs. The response time is indicated in Table 5.81 on page 335. Since the response time is so fast, this feature can be used to synchronize a portion of PBASIC code to an incoming pulse.

| <i>Duration</i> | <i>Length of Low-Power Mode</i> |
|-----------------|---------------------------------|
| 0               | 18 ms                           |
| 1               | 36 ms                           |
| 2               | 72 ms                           |
| 3               | 144 ms                          |
| 4               | 288 ms                          |
| 5               | 576 ms                          |
| 6               | 1152 ms (1.152 seconds)         |
| 7               | 2304 ms (2.304 seconds)         |
| 8               | No power-down                   |

**Table 5.82:** *Duration* values and associated low-power modes.

The following is a simple example of the POLLWAIT command.

A SIMPLE POLLWAIT EXAMPLE.

```
POLLIN 0, 0
POLLMODE 2
```

```
Main:
 POLLWAIT 2
 TOGGLE 1
 GOTO Main
```

## 5: BASIC Stamp Command Reference – POLLWAIT

In this example, the POLLIN command sets I/O pin 0 to be a polled-input pin looking for a low (0) state. The *Main* routine immediately runs a POLLWAIT command and specifies a *Duration* of 2 (which results in a low-power state of 72 ms). This means that every 72 ms, the BASIC Stamp wakes-up and checks I/O pin 0 for a low. If I/O pin 0 is high, it goes back to sleep for another 72 ms. If I/O pin 0 is low, it runs the next line of code, which toggles the state of I/O pin 1. Then the loop starts all over again. Note: Due to the nature of low-power mode, I/O pin 1 may toggle between high and low (at 72 ms intervals in this case) even if I/O pin 0 stays high. This is an artifact of the "reset" condition in the interpreter chip that occurs when the chip wakes up from a low-power state. Upon this "reset" condition, all the I/O pins are switched to inputs for approximately 18 ms. It is the switching to inputs that will cause I/O pin 1 to appear to toggle. See the NAP or SLEEP commands for more information.

If low-power mode is not required, change the POLLWAIT command in the example above to "POLLWAIT 8" instead. This will have the effect of keeping the BASIC Stamp in normal running mode (i.e.: no low-power glitches) and will also cause the TOGGLE command to execute in a much shorter amount of time after a polled-input event occurs.



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the \$STAMP directive accordingly.

### Demo Program (POLLWAIT.bsp)

```
' POLLWAIT.bsp
' This program demonstrates the POLLWAIT command. I/O pin 0 is set to
' watch for a low signal. Once the Main routine starts running, the
' POLLWAIT command causes the program to halt until the polled event
' happens (I/O pin is low) then it prints a message on the PC screen.
' It will do nothing until I/O pin is low.

' {$STAMP BS2p}
' {$PBASIC 2.5}

Setup:
 POLLIN 0, 0 ' polled-input, look for 0
 POLLMODE 2 ' enable polling

Main:
 POLLWAIT 8 ' Wait for polled event
 DEBUG "I/O pin 0 is LOW!", CR ' Print message
 GOTO Main
END
```

## ***POLLWAIT – BASIC Stamp Command Reference***

---



## 5: BASIC Stamp Command Reference – POT

### POT

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|



**POT Pin, Scale, Variable**

(See RCTIME)

### Function

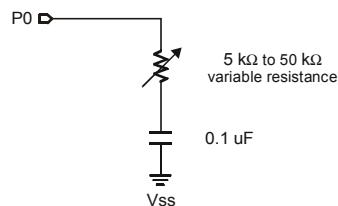
Read a 5 k $\Omega$  to 50 k $\Omega$  potentiometer, thermistor, photocell, or other variable resistance.

- **Pin** is a variable/constant (0 – 7) that specifies the I/O pin to use. This pin will be set to output mode initially, then to input mode.
- **Scale** is a variable/constant (0 – 255) used to scale the command's internal 16-bit result. See explanation below for steps to finding the scale value to use for your circuit.
- **Variable** is a variable (usually a byte) where the final result of the reading will be stored. Internally, the POT command calculates a 16-bit value, which is scaled down to an 8-bit value.

### Explanation

POT reads a variable resistance and returns a value (0 – 255) representing the amount of time it took to discharge the capacitor through the resistance. *Pin* must be connected to one side of the variable resistance, whose other side is connected through a capacitor to ground, as shown in Figure 5.27.

**Figure 5.27:** Example Variable Resistance Circuit.



#### HOW POT REALLY WORKS.

POT works by first setting the specified I/O pin to an output and setting its state high. This step places +5 volts on one side of the capacitor (see Figure 5.27) and ground (0 volts) on the other side, which charges the capacitor. POT waits for 10 ms and then sets the I/O pin to an input mode and starts its timer. Initially the I/O pin will see a high (1) that will eventually drop to a low (0) when the capacitor discharges past the 1.4-volt threshold. The timer stops once the low is seen. The value of the

## POT – BASIC Stamp Command Reference

---

variable resistor affects the time it takes to discharge the capacitor from 5 volts to approximately 1.4 volts.

The 16-bit reading is multiplied by ( $Scale/256$ ), so a scale value of 128 would reduce the range by approximately 50%, a scale of 64 would reduce to 25%, and so on. The amount by which the internal value must be scaled varies with the size of the resistor being used.

Finding the best Scale value:

1. Build the circuit shown in Figure 5.27 and plug the BS1 into the PC.
2. In the BASIC Stamp editor select *Pot Scaling* from the *Run* menu. A special calibration window appears, allowing you to find the best value.
3. The window asks for the number of the I/O pin to which the variable resistor is connected. Select the appropriate pin (0-7) from the drop-down.
4. The editor downloads a short program to the BS1 (this overwrites any program already stored in the BS1).
5. The window will now show the Scale Factor. Adjust the resistor until the smallest number is shown for scale (assuming you can adjust the resistor, as with a potentiometer).
6. Once you've found the smallest number for scale, you're done. This number should be used for the Scale in the POT command.
7. Optionally, you can verify the scale number found above by selecting the POT Value checkbox (so it's checked). This locks the scale and causes the BS1 to read the resistor continuously. The window displays the value. If the scale is good, you should be able to adjust the resistor, achieving a 0–255 reading for the value (or as close as possible). To change the scale value and repeat this step, deselect the POT Value checkbox. Continue this process until you find the best scale.

### Demo Program (POT.bs1)

```
' POT.bs1
' This program demonstrates the use of the POT command. Connect one side
' of a 5-50K potentiometer to P0. To the other side of the potentiometer
' connect a 0.1 uF capacitor, and then connect the other side of the
' capacitor to Vss (ground). Before running demo program,
' use the Run | POT Scaling dialog to determine the best Scale factor.
```



## ***5: BASIC Stamp Command Reference – POT***

---

```
' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL PotPin = 0 ' pot connected to P0
SYMBOL Scale = 111 ' scale value for test circuit

SYMBOL level = B2 ' storage of pot "level"

Main:
 POT PotPin, Scale, level ' read pot level
 DEBUG CLS, "Level = ", #level ' display
 PAUSE 50 ' short delay
 GOTO Main ' repeat forever
END
```

## ***POT – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – PULSIN

### PULSIN

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**PULSIN** *Pin, State, Variable*

#### Function

Measure the width of a pulse on *Pin* described by *State* and store the result in *Variable*.

- **Pin** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This pin will be set to input mode.
- **State** is a variable/constant/expression (0 – 1) that specifies whether the pulse to be measured is low (0) or high (1). A low pulse begins with a 1-to-0 transition and a high pulse begins with a 0-to-1 transition.
- **Variable** is a variable (usually a word) in which the measured pulse duration will be stored. The unit of time for *Variable* is described in Table 5.83.



NOTE: Expressions are not allowed as arguments on the BS1. The range of the *Pin* argument on the BS1 is 0 – 7.

#### Quick Facts

Table 5.83: PULSIN Quick Facts.

|                          | BS1               | BS2       | BS2e      | BS2sx       | BS2p        | BS2pe     | BS2px        |
|--------------------------|-------------------|-----------|-----------|-------------|-------------|-----------|--------------|
| Units in <i>Variable</i> | 10 $\mu$ s        | 2 $\mu$ s | 2 $\mu$ s | 0.8 $\mu$ s | 0.8 $\mu$ s | 2 $\mu$ s | 0.81 $\mu$ s |
| Maximum Pulse Width      | 655.35 ms         | 131.07 ms | 131.07 ms | 52.428 ms   | 52.428 ms   | 123.6 ms  | 53.08 ms     |
| Related Commands         | PULSOUT and COUNT |           |           |             |             |           |              |

#### Explanation

PULSIN is like a fast stopwatch that is triggered by a change in state (0 or 1) on the specified pin. The entire width of the specified pulse (high or low) is measured, in units shown in Table 5.83, and stored in *Variable*.

Many analog properties (voltage, resistance, capacitance, frequency, duty cycle) can be measured in terms of pulse durations. This makes PULSIN a valuable form of analog-to-digital conversion.

#### SPECIFICS OF PULSIN'S OPERATION.

PULSIN will wait, for the desired pulse, for up to the maximum pulse width it can measure, shown in Table 5.83. If it sees the desired pulse, it measures the time until the end of the pulse and stores the result in *Variable*. If it never sees the start of the pulse, or the pulse is too long

## PULSIN – BASIC Stamp Command Reference

(greater than the Maximum Pulse Width shown in Table 5.83) PULSIN "times out" and store 0 in *Variable*. This operation keeps your program from locking-up should the desired pulse never occur.

Regardless of the size of *Variable*, PULSIN internally uses a 16-bit timer. Unless the pulse widths are known to be short enough to fit in an 8-bit result, it is recommended to use a word-sized variable. Not doing so may result in strange and misleading results as the BASIC Stamp will only store the lower 8-bits into a byte variable.

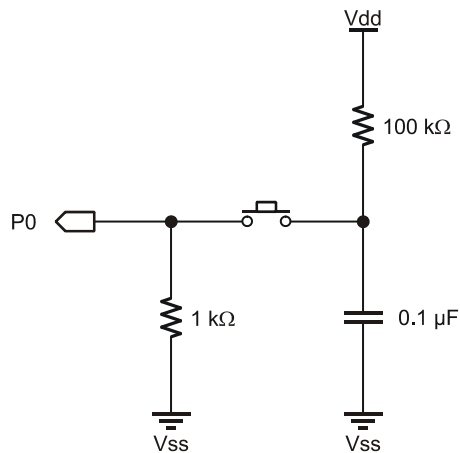


Figure 5.28: R/C Pulse Generator.

### Demo Program (PULSIN.bs1)

```
' PULSIN.bs1
' This program uses PULSIN to measure a pulse generated by discharging a
' 0.1 uF capacitor through a 1k resistor. Pressing the switch generates
' the pulse, which should ideally be approximately 120 us (12 PULSIN units
' of 10 us) long. Variations in component values may produce results that
' are up to 10 units off from this value. For more information on
' calculating resistor-capacitor timing, see the RCTIME command.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL Pulse = 7 ' pulse input pin
SYMBOL time = W1 ' pulse width (10 uS units)
```

## 5: BASIC Stamp Command Reference – PULSIN

```
Main:
 PULSIN Pulse, 1, time ' measure positive pulse
 IF time = 0 THEN Main ' if 0, try again
 DEBUG CLS, time ' else display result
 GOTO Main
END
```

All 2

NOTE: This example program can be used with all BS2 models. This program uses conditional compilation techniques; see Chapter 3 for more information.

### Demo Program (PULSIN.bs2)

```
' PULSIN.bs2
' This program uses PULSIN to measure a pulse generated by discharging a
' 0.1 uF capacitor through a 1K resistor. Pressing the switch generates
' the pulse, which should ideally be approximately 120 us (60 PULSIN units
' of 2 us) long (for BS2 and BS2e). Variations in component values may
' produce results that are up to 10 units off from this value. For more
' information on calculating resistor-capacitor timing, see the RCTIME
' command.

' {$STAMP BS2}
' {$PBASIC 2.5}

Pulse PIN 7 ' pulse input pin

#SELECT $STAMP
 #CASE BS2, BS2E, BS2PE
 Scale CON $200 ' 2.0 us per unit
 #CASE BS2SX, BS2P
 Scale CON $0CC ' 0.8 us per unit
 #CASE BS2PX
 Scale CON $0CF ' 0.81 us per unit
#ENDSELECT

time VAR Word

Main:
 PULSIN Pulse, 1, time ' measure positive pulse
 IF (time > 0) THEN ' if not 0
 DEBUG HOME,
 DEC time, " units ", CLREOL ' display raw input
 time = time */ Scale ' adjust for Stamp
 DEBUG CR,
 DEC time, " us " ' display microseconds
 ELSE
 DEBUG CLS, "Out of Range" ' else error message
 ENDIF
 PAUSE 200
 GOTO Main
END
```

## ***PULSIN – BASIC Stamp Command Reference***

---



## 5: BASIC Stamp Command Reference – PULSOUT

### PULSOUT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**PULSOUT** *Pin, Duration*

#### Function

Generate a pulse on *Pin* with a width of *Duration*.

- ***Pin*** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This pin will be set to output mode.
- ***Duration*** is a variable/constant/expression (0 – 65535) that specifies the duration of the pulse. The unit of time for *Duration* is described in Table 5.84.

#### Quick Facts

|                            | BS1        | BS2       | BS2e      | BS2sx       | BS2p        | BS2pe     | BS2px       |
|----------------------------|------------|-----------|-----------|-------------|-------------|-----------|-------------|
| <b>Duration units</b>      | 10 $\mu$ s | 2 $\mu$ s | 2 $\mu$ s | 0.8 $\mu$ s | 0.8 $\mu$ s | 2 $\mu$ s | 0.8 $\mu$ s |
| <b>Maximum Pulse Width</b> | 655.35 ms  | 131.07 ms | 131.07 ms | 52.428 ms   | 52.428 ms   | 131.07 ms | 52.428 ms   |
| <b>Related Command</b>     | PULSIN     |           |           |             |             |           |             |

#### Explanation

PULSOUT sets *Pin* to output mode, inverts the state of that pin; waits for the specified *Duration*; then inverts the state of the pin again; returning the bit to its original state. The unit of *Duration* is described in Table 5.84. The following example will generate a 100  $\mu$ s pulse on I/O pin 7 (of the BS2):

```
PULSOUT 7, 50 ' generate 100 us pulse on P7
```

CONTROLLING THE POLARITY OF THE PULSE.

The polarity of the pulse depends on the state of the pin before the command executes. In the example above, if pin 7 was low, PULSOUT would produce a positive (high) pulse. If the pin was high, PULSOUT would produce a negative (low) pulse.

WATCH OUT FOR UNDESIRABLE PULSE GLITCHES.

If the pin is an input, the output state bit, OUT7 (PIN7 on the BS1) won't necessarily match the state of the pin. What happens then? For example: pin 7 is an input (DIR7 = 0) and pulled high by a resistor as shown in Figure 5.29a. Suppose that pin 7 is low when we execute the instruction:

```
PULSOUT 7, 5 ' generate pulse on P7
```

## PULSOUT – BASIC Stamp Command Reference

Figure 5.29b shows the sequence of events on that pin. Initially, pin 7 is high. Its output driver is turned off (because it is in input mode), so the 10 kΩ resistor sets the state on the pin. When PULSOUT executes, it turns on the output driver, allowing OUT7 (PIN7 on the BS1) to control the pin.

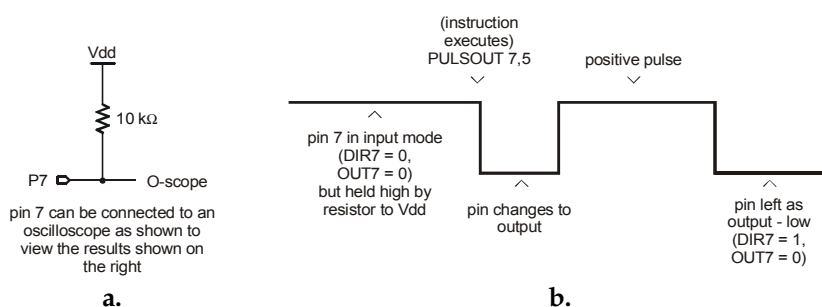


**Figure 5.29:** Example Pulse Diagram.

Since OUT7 (PIN7 on the BS1) is low, the pin goes low. After a few microseconds of preparation, PULSOUT inverts the state of the pin; from low to high. It leaves the pin in that state for the specified time (10μs if using a BS2) and then inverts it again, leaving the pin in its original state.

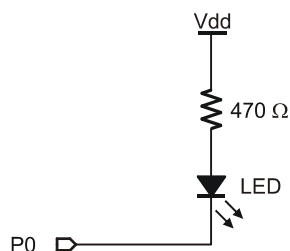


**Figure 5.30:** LED Circuit for PULSOUT Demo Programs.

### Demo Program (PULSOUT.bs1)

```
' PULSOUT.bs1
' This program blinks an LED on for 25 ms at 1-second intervals. Connect an
' LED (active-low) to I/O pin 0.

' {$STAMP BS1}
' {$PBASIC 1.0}
```

## 5: BASIC Stamp Command Reference – PULSOUT

```
Setup:
 HIGH 0 ' make P0 high (LED off)

Main:
 PULSOUT 0, 2500 ' flash LED for 25 ms
 PAUSE 1000 ' one second delay
 GOTO Main
END
```



NOTE: This example program can be used with all BS2 models. This program uses conditional compilation techniques; see Chapter 3 for more information.

### Demo Program (PULSOUT.bs2)

```
' PULSOUT.bs2
' This program blinks an LED on for 25 ms at 1-second intervals. Connect an
' LED (active-low) to I/O pin 0.

' {$STAMP BS2}
' {$PBASIC 2.5}

#SELECT $STAMP
#CASE BS2, BS2E, BS2PE
 Scale CON 500 ' to ms for 2 us per unit
#CASE BS2SX, BS2P, BS2PX
 Scale CON 1250 ' to ms for 0.8 us per unit
#ENDSELECT

Flash CON 25 * Scale ' 25 milliseconds

Setup:
 HIGH 0 ' make P0 high (LED off)

Main:
 PULSOUT 0, Flash ' flash LED
 PAUSE 1000 ' one second delay
 GOTO Main
END
```

## ***PULSOUT – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – PUT

**PUT** BS1 BS2 BS2e BS2sx BS2p BS2pe BS2px  
**PUT** *Location*, { **WORD** } *Value* { , { **WORD** } *Value*... }

NOTE: The optional arguments require PBASIC 2.5.

### Function

Write one or more values to the Scratch Pad RAM (SPRAM), starting at *Location*.

- **Location** is a variable/constant/expression (0 – 62 for BS2e and BS2sx, and 0 – 126 for BS2p, BS2pe, and BS2px) that specifies the SPRAM location to write to.
- **Value** is a variable/constant/expression (0 – 255, or 0 – 65535 if using the optional WORD modifier) to store in the SPRAM.

### Quick Facts

**Table 5.85:** PUT Quick Facts.

NOTE: When the WORD modifier is used, the low byte of the value is written to *Location*, the high byte to *Location* + 1.

|                                              | BS2e and BS2sx                                                                                                                           | BS2p, BS2pe, and BS2px                                                                                                        |
|----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <b>Scratch Pad RAM Size and Organization</b> | 64 bytes (0 – 63). Organized as bytes only.                                                                                              | 136 bytes (0 – 135). Organized as bytes only.                                                                                 |
| <b>General Purpose Locations</b>             | 0 - 62                                                                                                                                   | 0 – 126                                                                                                                       |
| <b>Special Use Location</b>                  | Location 63: Active program slot number (read only).                                                                                     | Location 127: READ/WRITE slot and Active Program slot (read only).<br>Locations 128-135: Polled Interrupt status (read only). |
| <b>Related Commands</b>                      | GET                                                                                                                                      | GET and STORE, and SPSTR formatter.                                                                                           |
| <b>PBASIC 2.5 Syntax Options</b>             | Multiple sequential variables may be written to the Scratch Pad RAM. The optional WORD modifier may be specified to store 16-bit values. |                                                                                                                               |

### Explanation

The PUT command writes value into the specified Scratch Pad RAM (SPRAM) location. All values in the general-purpose locations can be written to from within any of the 8 program slots.

USES FOR SCRATCH PAD RAM.

SPRAM is useful for passing data to programs in other program slots and for additional workspace. It is different than regular RAM in that symbol names cannot be assigned directly to locations and each location is always configured as a byte only. The following code will write the value 100 to location 25, read it back out with GET, and display it:

## PUT – BASIC Stamp Command Reference

```
temp VAR byte

PUT 25, 100 ' put low byte
GET 25, temp ' read byte value
DEBUG DEC temp ' display byte value
```

When using the \$PBASIC 2.5 directive, multiple sequential values may be stored to SPRAM, starting at *Location*, and the WORD modifier may be specified to store 16-bit values.

```
' {$PBASIC 2.5}

temp VAR Word

PUT 25, Word 2125 ' write word value
GET 25, Word temp ' read word value
DEBUG DEC temp ' display 2125
```

Most Scratch Pad RAM locations are available for general use. The highest locations have a special, read-only purpose; see the GET command for more information.

SCRATCH PAD RAM LOCATIONS AND THEIR PURPOSE.

### Demo Program (GET\_PUT1.bsx)

```
' GET_PUT1.bsx
' This example demonstrates the use of the GET and PUT commands. First,
' slot location is read using GET to display the currently running program
' number. Then a set of values are written (PUT) into locations 0 TO 9.
' Afterwards, program number 1 is RUN. This program is a BS2SX project
' consisting of GET_PUT1.BSX and GET_PUT2.BSX, but will run on the BS2e,
' BS2p, BS2pe, and BS2px without modification.

' {$STAMP BS2sx, GET_PUT2.BSX}
' {$PBASIC 2.5}

#SELECT $STAMP
#CASE BS2
 #ERROR "BS2e or greater required."
#CASE BS2E, BS2SX
 Slot CON 63
#CASE BS2P, BS2PE, BS2PX
 Slot CON 127
#ENDSELECT

value VAR Byte
idx VAR Byte

Setup:
 GET Slot, value
```



NOTE: This is written for the BS2sx but can be used with the BS2e, BS2p, BS2pe and BS2px also. This program uses conditional compilation techniques; see Chapter 3 for more information.

## 5: BASIC Stamp Command Reference – PUT

```
DEBUG "Program Slot #", DEC value.NIB0, CR

Main:
 FOR idx = 0 TO 9
 value = (idx + 3) * 8
 PUT idx, value
 DEBUG " Writing: ", DEC2 value, " to location: ", DEC2 idx, CR
 NEXT
DEBUG CR
RUN 1
END
```



NOTE: This is written for the BS2sx but can be used with the BS2e, BS2p, BS2pe, and BS2px also. This program uses conditional compilation techniques; see Chapter 3 for more information.

### Demo Program (GET\_PUT2.bsx)

```
' GET_PUT2.bsx
' This example demonstrates the use of the GET and PUT commands. First,
' the Slot location is read using GET to display the currently running
' program number. Then a set of values are read (GET) from locations
' 0 to 9 and displayed on the screen for verification. This program is a
' BS2SX project consisting of GET_PUT1.BSX and GET_PUT2.BSX, but will run
' on the BS2e, BS2p, BS2pe, and BS2px without modification.

' {$STAMP BS2sx}
' {$PBASIC 2.5}

#SELECT $STAMP
#CASE BS2
 #ERROR "BS2e or greater required."
#CASE BS2E, BS2SX
 Slot CON 63
#CASE BS2P, BS2PE, BS2PX
 Slot CON 127
#ENDSELECT

value VAR Byte
idx VAR Byte

Setup:
 GET Slot, value
 DEBUG "Program Slot #", DEC value.NIB0, CR

Main:
 FOR idx = 0 TO 9
 GET idx, value
 DEBUG " Reading: ", DEC2 value, " from location: ", DEC2 idx, CR
 NEXT
END
```

## ***PUT – BASIC Stamp Command Reference***

---



## 5: BASIC Stamp Command Reference – PWM

### PWM

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



#### PWM *Pin, Duty, Cycles*

#### Function

Convert a digital value to analog output via pulse-width modulation.

- **Pin** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This pin will be set to output mode initially then set to input mode when the command finishes.
- **Duty** is a variable/constant/expression (0 - 255) that specifies the analog output level (0 to 5V).
- **Cycles** is a variable/constant/expression (0 - 255) that specifies the duration of the PWM signal.

#### Quick Facts

Table 5.86: PWM Quick Facts

|                                                | BS1                                                     | BS2                 | BS2e | BS2sx  | BS2p    | BS2pe   | BS2px  |
|------------------------------------------------|---------------------------------------------------------|---------------------|------|--------|---------|---------|--------|
| Units in <i>Cycles</i>                         | 5 ms                                                    | 1 ms                | 1 ms | 0.4 ms | 0.65 ms | 1.62 ms | 0.4 ms |
| Average Voltage Equation                       | Average Voltage = (Duty / 255) * 5 volts                |                     |      |        |         |         |        |
| Required ChargeTime ( <i>Cycles</i> ) Equation | Charge time = 5 * R * C                                 |                     |      |        |         |         |        |
| Special Notes                                  | Pin is set to output initially, and set to input at end |                     |      |        |         |         |        |
| Related Commands                               | none                                                    | FREQOUT and DTMFOUT |      |        |         |         |        |

#### Explanation

Pulse-width modulation (PWM) allows the BASIC Stamp (a purely digital device) to generate an analog voltage. The basic idea is this: If you make a pin output high, the voltage at that pin will be close to 5 V. Output low is close to 0 V. What if you switched the pin rapidly between high and low so that it was high half the time and low half the time? The average voltage over time would be halfway between 0 and 5 V (2.5 V). PWM emits a burst of 1s and 0s whose ratio is proportional to the duty value you specify, making the average voltage over time somewhere between 0 and 5 V.

DETERMINING AVERAGE VOLTAGE FOR A PARTICULAR DUTY CYCLE.

The proportion of 1s to 0s in PWM is called the duty cycle. The duty cycle controls the analog voltage in a very direct way; the higher the duty cycle the higher the voltage. In the case of the BASIC Stamp, the duty cycle can range from 0 to 255. Duty is literally the proportion of 1s to 0s output by the PWM command. To determine the proportional PWM output voltage,

## PWM – BASIC Stamp Command Reference

use this formula:  $(Duty/255) * 5V$ . For example, if *Duty* is 100,  $(100/255) * 5V = 1.96V$ ; PWM outputs a train of pulses whose average voltage is 1.96V.

In order to convert PWM into an analog voltage we have to filter out the pulses and store the average voltage. The resistor/capacitor combination in Figure 5.31 will do the job. The capacitor will hold the voltage set by PWM even after the instruction has finished. How long it will hold the voltage depends on how much current is drawn from it by external circuitry, and the internal leakage of the capacitor. In order to hold the voltage relatively steady, a program must periodically repeat the PWM instruction to give the capacitor a fresh charge.

FILTERING THE PWM SIGNAL.

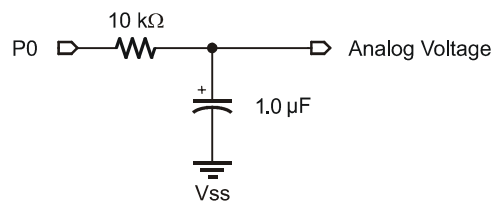


**Figure 5.31:** Example PWM Filter Circuit.

Just as it takes time to discharge a capacitor, it also takes time to charge it in the first place. The PWM command lets you specify the charging time in terms of PWM cycles. The period of each cycle is shown in Table 5.86. So, on the BS2, to charge a capacitor for 5ms, you would specify 5 cycles in the PWM instruction.

DETERMINING THE APPROPRIATE CYCLE TIME FOR YOUR CIRCUIT.

How do you determine how long to charge a capacitor? Use this rule-of-thumb formula: Charge time =  $5 * R * C$ . For instance, Figure 5.31 uses a 10 kΩ ( $10 \times 10^3$  ohm) resistor and a 1 μF ( $1 \times 10^{-6}$  F) capacitor:

Charge time =  $5 * 10 \times 10^3 * 1 \times 10^{-6} = 50 \times 10^{-3}$  seconds, or 50 ms.

Since, on the BS2, each cycle is approximately a millisecond, it would take at least 50 cycles to charge the capacitor. Assuming the circuit is connected to pin 0, here's the complete PWM instruction:

```
PWM 0, 100, 50 ' charge to 1.96 V
```

After outputting the PWM pulses, the BASIC Stamp leaves the pin in input mode (0 in the corresponding bit of DIRS). In input mode, the pin's output driver is effectively disconnected. If it were not, the steady output

## 5: BASIC Stamp Command Reference – PWM

---

state of the pin would change the voltage on the capacitor and undo the voltage setting established by PWM. Keep in mind that leakage currents of up to 1  $\mu\text{A}$  can flow into or out of this “disconnected” pin. Over time, these small currents will cause the voltage on the capacitor to drift. The same applies for leakage current from an op-amp’s input, as well as the capacitor’s own internal leakage. Executing PWM occasionally will reset the capacitor voltage to the intended value.

PWM charges the capacitor; the load presented by your circuit discharges it. How long the charge lasts (and therefore how often your program should repeat the PWM command to refresh the charge) depends on how much current the circuit draws, and how stable the voltage must be. You may need to buffer PWM output with a simple op-amp follower if your load or stability requirements are more than the passive circuit of Figure 5.31 can handle.

HOW PULSE-WIDTH-MODULATION IS  
GENERATED.

The term “PWM” applies only loosely to the action of the BASIC Stamp's PWM command. Most systems that output PWM do so by splitting a fixed period of time into an on time (1) and an off time (0). Suppose the interval is 1 ms and the duty cycle is 100 / 255. Conventional PWM would turn the output on for 0.39 ms and off for 0.61 ms, repeating this process each millisecond. The main advantage of this kind of PWM is its predictability; you know the exact frequency of the pulses (in this case, 1 kHz), and their widths are controlled by the duty cycle.

BASIC Stamp's PWM does not work this way. It outputs a rapid sequence of on/off pulses, as short as 1.6  $\mu\text{s}$  in duration, whose overall proportion over the course of a full PWM cycle of approximately a millisecond is equal to the duty cycle. This has the advantage of very quickly zeroing in on the desired output voltage, but it does not produce the neat, orderly pulses that you might expect. All BS2 models also use this high-speed PWM to generate pseudo-sine wave tones with the DTMFOUT and FREQOUT instructions.



### Demo Program (PWM.bs1)

```
' PWM.BS1
' Connect a voltmeter (such as a digital multimeter set to its voltage
' range) to the output of the circuit shown in the figure for the PWM
' command (in the manual). Run the program and observe the readings on
' the meter. They should come very close to 1.96V, then decrease slightly
```

## PWM – BASIC Stamp Command Reference

---

```
' as the capacitor discharges. Try varying the interval between PWM bursts
' (by changing the PAUSE value) and the number of PWM cycles to see their
' effect.

' {$STAMP BS1}
' {$PBASIC 1.0}

Main:
 PWM 0, 100, 10 ' PWM at 100/255 duty (~50 ms)
 PAUSE 1000 ' wait one second
 GOTO Main
END
```

### Demo Program (PWM.bs2)



NOTE: This example program can be used with all BS2 models. This program uses conditional compilation techniques; see Chapter 3 for more information.

```
' PWM.bs2
' Connect a voltmeter (such as a digital multimeter set to its voltage
' range) to the output of the circuit shown in the figure for the PWM
' command (in the manual). Run the program and observe the readings on
' the meter. They should come very close to 1.96V, then decrease slightly
' as the capacitor discharges. Try varying the interval between PWM bursts
' (by changing the PAUSE value) and the number of PWM cycles to see their
' effect.

' {$STAMP BS2}
' {$PBASIC 2.5}

#SELECT $STAMP
#CASE BS2, BS2E
 CycAdj CON $100 ' x 1.0, cycle adjustment (for ms)
#CASE BS2SX
 CycAdj CON $280 ' x 2.5
#CASE BS2P
 CycAdj CON $187 ' x 1.53
#CASE BS2PE
 CycAdj CON $09E ' x 0.62
#CASE BS2PX
 CycAdj CON $280 ' x 2.5
#ENDSELECT

Cycles CON 50

Main:
 PWM 0, 100, (Cycles */ CycAdj) ' PWM at 100/255 duty (~50 ms)
 PAUSE 1000 ' wait one second
 GOTO Main
END
```

## 5: BASIC Stamp Command Reference – RANDOM

### RANDOM

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|



#### RANDOM Variable

#### Function

Generate a pseudo-random number.

- **Variable** is a variable (usually a word) whose bits will be scrambled to produce a random number. *Variable* acts as RANDOM's input and its result output. Each pass through RANDOM stores the next number, in the pseudorandom sequence, in *Variable*.

#### Explanation

RANDOM generates pseudo-random numbers ranging from 0 to 65535. They're called "pseudo-random" because they appear random, but are generated by a logic operation that uses the initial value in *Variable* to "tap" into a sequence of 65535 essentially random numbers. If the same initial value, called the "seed", is always used, then the same sequence of numbers is generated. The following example demonstrates this:

 SYMBOL    result                      = W1

```
Main:
 result = 11000
 RANDOM result
 DEBUG result
 GOTO Main
```

-- OR --



result                      VAR            Word

```
Main:
 result = 11000
 RANDOM result
 DEBUG DEC ? result
 GOTO Main
```

In this example, the same number would appear on the screen over and over again. This is because the same seed value was used each time; specifically, the first line of the loop sets *result* to 11,000. The RANDOM command really needs a different seed value each time. Moving the "result =" line out of the loop will solve this problem, as in:

## ***RANDOM – BASIC Stamp Command Reference***

---

```
SYMBOL result = W1
```



```
Setup:
 result = 11000
```

```
Main:
 RANDOM result
 DEBUG result
 GOTO Main
```

-- OR --

```
result VAR Word
```



```
Setup:
 result = 11000
```

```
Main:
 RANDOM result
 DEBUG DEC ? result
 GOTO Main
```

Here, *result* is only initialized once, before the loop. Each time through the loop, the previous value of *result*, generated by RANDOM, is used as the next seed value. This generates a more desirable set of pseudorandom numbers.

In applications requiring more apparent randomness, it's necessary to "seed" RANDOM with a more random value every time. For instance, in the demo program below, RANDOM is executed continuously (using the previous resulting number as the next seed value) while the program waits for the user to press a button. Since the user can't control the timing of button presses very accurately, the results approach true randomness.

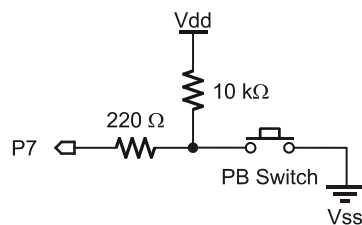


**Figure 5.32:** RANDOM Button Circuit.

## 5: BASIC Stamp Command Reference – RANDOM



### Demo Program (RANDOM.bs1)

```
' RANDOM.bs1
' Connect a button to I/O pin 7 as shown in the figure in the RANDOM
' command description and run this program. This program uses RANDOM to
' simulate a coin toss. After 100 trials, it reports the total number of
' heads and tails thrown.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL Btn = 7 ' button input

SYMBOL flip = W0 ' a random number
SYMBOL coin = BIT0 ' a bit from random number
SYMBOL trials = B2 ' number of flips
SYMBOL heads = B3 ' throws that come up heads
SYMBOL tails = B4 ' throws that come up tails
SYMBOL btnWrk = B5 ' workspace for BUTTON

Start:
 DEBUG CLS, "Press the button to toss coin.", CR

Main:
 FOR trials = 1 TO 100 ' flip coin 100 times

Hold:
 RANDOM flip ' randomize while waiting
 BUTTON Btn, 0, 250, 100, btnWrk, 0, Hold ' wait for button press
 BRANCH coin, (Head, Tail) ' 0 = heads, 1 = tails

Head:
 DEBUG CR, "Heads!"
 heads = heads + 1 ' increment heads counter
 GOTO Next_Toss

Tail:
 DEBUG CR, "Tails..."
 tails = tails + 1 ' increment heads counter

Next_Toss:
 NEXT

 DEBUG CR, CR, "Heads: ", #heads, CR, "Tails: ", #tails
 END
```



### Demo Program (RANDOM.bs2)

```
' RANDOM.BS2
' Connect a button to I/O pin 7 as shown in the figure in the RANDOM
' command description and run this program. This program uses RANDOM to
' simulate a coin toss. After 100 trials, it reports the total number of
```

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## ***RANDOM – BASIC Stamp Command Reference***

---

```
' heads and tails thrown.

' {$STAMP BS2}
' {$PBASIC 2.5}

Btn PIN 7 ' button input

flip VAR Word ' a random number
coin VAR flip.BIT0 ' Bit0 of the random number
trials VAR Byte ' number of flips
heads VAR Byte ' throws that come up heads
tails VAR Byte ' throws that come up tails
btnWrk VAR Byte ' workspace for BUTTON

Start:
 DEBUG CLS, "Press button to start"

Main:
 FOR trials = 1 TO 100 ' flip coin 100 times

Hold:
 RANDOM flip ' randomize while waiting
 BUTTON Btn, 0, 250, 100, btnWrk, 0, Hold ' wait for button press
 IF (coin = 0) THEN ' 0 = heads, 1 = tails
 DEBUG CR, "Heads!"
 heads = heads + 1 ' increment heads counter
 ELSE
 DEBUG CR, "Tails..."
 tails = tails + 1 ' increment tails counter
 ENDIF
NEXT

Done:
 DEBUG CR, CR, "Heads: ", DEC heads, " Tails: ", DEC tails
END
```



## 5: BASIC Stamp Command Reference – RCTIME

### RCTIME

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



(See POT)



**RCTIME** *Pin, State, Variable*

#### Function

Measure time while *Pin* remains in *State*; usually to measure the charge/discharge time of resistor/capacitor (RC) circuit.

- ***Pin*** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This pin will be placed into input mode.
- ***State*** is a variable/constant/expression (0 - 1) that specifies the desired state to measure. Once *Pin* is not in *State*, the command ends and stores the result in *Variable*.
- ***Variable*** is a variable (usually a word) in which the time measurement will be stored. The unit of time for *Variable* is described in Table 5.87.

#### Quick Facts

Table 5.87: RCTIME Quick Facts.

|                          | BS2       | BS2e      | BS2sx       | BS2p         | BS2pe     | BS2px        |
|--------------------------|-----------|-----------|-------------|--------------|-----------|--------------|
| Units in <i>Variable</i> | 2 $\mu$ s | 2 $\mu$ s | 0.8 $\mu$ s | 0.75 $\mu$ s | 2 $\mu$ s | 0.75 $\mu$ s |
| Maximum Pulse Width      | 131.07 ms | 131.07 ms | 52.428 ms   | 49.151 ms    | 131.07 ms | 49.151 ms    |

#### Explanation

RCTIME can be used to measure the charge or discharge time of a resistor/capacitor circuit. This allows you to measure resistance or capacitance; use R or C sensors such as thermistors or capacitive humidity sensors or respond to user input through a potentiometer. In a broader sense, RCTIME can also serve as a fast, precise stopwatch for events of very short duration.

#### HOW RCTIME'S TIMER WORKS.

When RCTIME executes, it makes *Pin* an input, then starts a counter (who's unit of time is shown in Table 5.87). It stops this counter as soon as the specified pin is no longer in *State* (0 or 1). If pin is not in *State* when the instruction executes, RCTIME will return 1 in *Variable*, since the instruction requires one timing cycle to discover this fact. If pin remains in *State* longer than 65535 timing cycles RCTIME returns 0.

## RCTIME – BASIC Stamp Command Reference

Figure 5.33 shows suitable RC circuits for use with RCTIME. The circuit in Figure 5.33a is preferred, because the BASIC Stamp's logic threshold is approximately 1.4 volts. This means that the voltage seen by the pin will start at 5V then fall to 1.4V (a span of 3.6V) before RCTIME stops. With the circuit of Figure 5.33b, the voltage will start at 0V and rise to 1.4V (spanning only 1.4V) before RCTIME stops. For the same combination of R and C, the circuit shown in Figure 5.33a will yield a higher count, and therefore more resolution than Figure 5.33b.

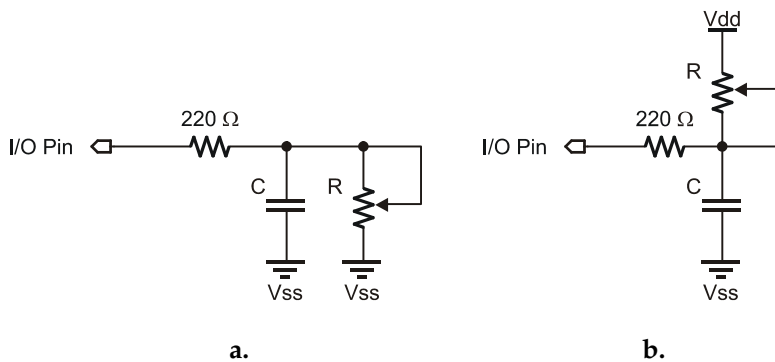


**Figure 5.33:** Example RC Circuits. Use **a** (left) with *State* = 1. Use **b** (right) with *State* = 0.

Before RCTIME executes, the capacitor must be put into the state specified in the RCTIME instruction. For example, with Figure 5.33a, the capacitor must be charged until the top plate is at 5V, then a *State* value of 1 will be used to monitor the discharge of the capacitor through the variable resistance.

DON'T FORGET TO DISCHARGE THE CAPACITOR BEFORE EXECUTING RCTIME.

Here's a typical sequence of instructions for Figure 5.33a (assuming I/O pin 7 is used):

```
result VAR Word

HIGH 7 ' charge the cap
PAUSE 1 ' for 1 ms
RCTIME 7, 1, result ' measure RC discharge time
DEBUG DEC ? result ' display result
```

Using RCTIME is very straightforward, except for one detail: For a given R and C, what value will RCTIME return? It's easy to figure, based on a value called the RC time constant, or tau ( $\tau$ ) for short. Tau represents the time required for a given RC combination to charge or discharge by 63

PREDICTING THE RETURNED VALUE.

## 5: BASIC Stamp Command Reference – RCTIME

---

percent of the total change in voltage that they will undergo. More importantly, the value  $\tau$  is used in the generalized RC timing calculation. Tau's formula is just R multiplied by C:

$$\tau = R \times C$$

CALCULATING CHARGE AND DISCHARGE TIME.

The general RC timing formula uses  $\tau$  to tell us the time required for an RC circuit to change from one voltage to another:

$$\text{time} = -\tau * (\ln(V_{\text{final}} / V_{\text{initial}}))$$

In this formula  $\ln$  is the natural logarithm; it's a key on most scientific calculators. Let's do some math. Assume we're interested in a 10 k resistor and 0.1  $\mu\text{F}$  cap. Calculate  $\tau$ :

$$\tau = (10 \times 10^3) \times (0.1 \times 10^{-6}) = 1 \times 10^{-3}$$

The RC time constant is  $1 \times 10^{-3}$  or 1 millisecond. Now calculate the time required for this RC circuit to go from 5V to 1.4V (as in Figure 5.33a):

$$\text{time} = -1 \times 10^{-3} \times (\ln(1.4\text{v} / 5.0\text{v})) = 1.273 \times 10^{-3}$$

THE RC TIME EQUATION.

On the BS2, the unit of time is  $2\mu\text{s}$  (See Table 5.87), that time ( $1.273 \times 10^{-3}$ ) works out to 636 units. With a 10 k $\Omega$  resistor and 0.1  $\mu\text{F}$  cap, RCTIME would return a value of approximately 635. Since  $V_{\text{initial}}$  and  $V_{\text{final}}$  doesn't change, we can use a simplified rule of thumb to *estimate* RCTIME results for circuits like Figure 5.33a:

$$\text{RCTIME units} = 635 \times R (\text{in k}\Omega) \times C (\text{in } \mu\text{F})$$

DETERMINING HOW LONG TO CHARGE OR DISCHARGE THE CAPACITOR BEFORE EXECUTING RCTIME.

Another handy rule of thumb can help you calculate how long to charge/discharge the capacitor before RCTIME. In the example above that's the purpose of the HIGH and PAUSE commands. A given RC charges or discharges 98 percent of the way in 5 time constants ( $5 \times R \times C$ ). In Figure 5.33, the charge/discharge current passes through the 220  $\Omega$  series resistor and the capacitor. So if the capacitor were 0.1  $\mu\text{F}$ , the minimum charge/discharge time should be:

$$\text{Charge time} = 5 \times 220 \times (0.1 \times 10^{-6}) = 110 \times 10^{-6}$$

## RCTIME – BASIC Stamp Command Reference

---

So it takes only 110  $\mu$ s for the cap to charge/discharge, meaning that the 1 ms charge/discharge time of the example is plenty.

A final note about Figure 5.33: You may be wondering why the 220  $\Omega$  resistor is necessary at all. Consider what would happen if resistor R in Figure 5.33a were a pot, and were adjusted to 0  $\Omega$ . When the I/O pin went high to charge the cap, it would see a short direct to ground. The 220  $\Omega$  series resistor would limit the short circuit current to  $5V/220\ \Omega = 23\text{ mA}$  and protect the BASIC Stamp from damage. (Actual current would be quite a bit less due to internal resistance of the pin's output driver, but you get the idea.)

NOTES ABOUT 220  $\Omega$  RESISTOR IN THE RC CIRCUITS.

### Demo Program (RCTIME1.bs2)

```
' RCTIME1.BS2
' This program shows the standard use of the RCTIME instruction measuring
' an RC charge/discharge time. Use the circuit in the RCTIME description
' (in the manual) with R = 10K pot and C = 0.1 uF. Connect the circuit to
' pin 7 and run the program. Adjust the pot and watch the value shown on
' the Debug screen change.

' {$STAMP BS2}
' {$PBASIC 2.5}

RC PIN 7

result VAR Word

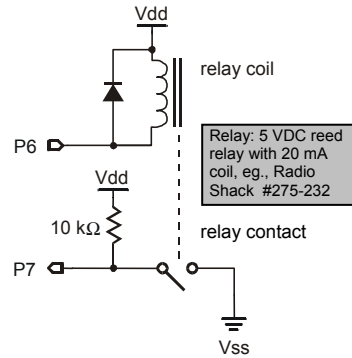
Main:
 DO
 HIGH RC ' charge the cap
 PAUSE 1 ' for 1 ms
 RCTIME RC, 1, result ' measure RC discharge time
 DEBUG HOME, DEC result ' display value
 PAUSE 50
 LOOP
END
```

All 2

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## 5: BASIC Stamp Command Reference – RCTIME

**Figure 5.34:** Relay Circuit for Demo Program RCTIME2.bs2.



All 2

NOTE: This example program can be used with all BS2 models. This program uses conditional compilation techniques; see Chapter 3 for more information.

### Demo Program (RCTIME2.bs2)

```
' RCTIME2.BS2
' This program illustrates the use of RCTIME as a fast stopwatch. The
' program energizes a relay coil, then measures how long it takes for the
' relay contacts to close. The circuit for this program can be found in
' the manual. Note that RCTIME doesn't start timing instantly -- as with
' all PBASIC instructions, it must be fetched from program EEPROM before
' it can execute.

' {$STAMP BS2}
' {$PBASIC 2.5}

Coil PIN 6
RC PIN 7

#SELECT $STAMP
#CASE BS2, BS2E, BS2PE
 Adjust CON $200 ' x 2 us per unit
#CASE BS2SX
 Adjust CON $0CC ' x 0.8 us per unit
#CASE BS2P, BS2PX
 Adjust CON $0C0 ' x 0.75 us per unit
#ENDSELECT

result VAR Word

Main:
DO
 LOW Coil ' energize relay coil
 RCTIME RC, 1, result ' measure time to contact closure
 result = result */ Adjust ' adjust for device
 DEBUG "Time to close: ",
```

## ***RCTIME – BASIC Stamp Command Reference***

---

```
 DEC Result, CR
HIGH Coil ' release relay
PAUSE 1000 ' wait one second
LOOP
END
```

## 5: BASIC Stamp Command Reference – READ

### READ

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|

NOTE: Optional arguments require PBASIC 2.5.



**READ** *Location, Variable*

**READ** *Location, { WORD } Variable {, { WORD } Variable... }*

### Function

Read the value at *Location* in EEPROM and store the result in *Variable*.

- **Location** is a variable/constant/expression (0 – 255 on BS1, 0 – 2047 on all BS2 models) that specifies the EEPROM address to read from.
- **Variable** is a variable (usually a byte, or word when the optional WORD modifier is used) in which to store the value.



NOTE: Expressions are not allowed as arguments on the BS1.

### Quick Facts

Table 5.88: READ Quick Facts.

|                                  | BS1              | BS2, BS2e, BS2sx                                                                                                                               | BS2p, BS2pe, BS2px                                            |
|----------------------------------|------------------|------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------|
| <b>Range of EEPROM Locations</b> | 0 to 255         | 0 to 2047                                                                                                                                      | 0 to 2047<br>(see notes below)                                |
| <b>Special Notes</b>             | n/a              | READ only works with current program slot on BS2e and BS2sx.                                                                                   | READ works with any program slot as set by the STORE command. |
| <b>Related Commands</b>          | WRITE and EEPROM | WRITE and DATA                                                                                                                                 | WRITE, DATA and STORE                                         |
| <b>PBASIC 2.5 Syntax Options</b> | n/a              | Multiple sequential variables may be read from the Scratch Pad RAM, and the optional WORD modifier may be specified to retrieve 16-bit values. |                                                               |

### Explanation

The EEPROM is used for both program storage (which builds downward from address 255 on BS1, 2047 on all BS2 models) and data storage (which builds upward from address 0). The READ instruction retrieves a value from any EEPROM address and stores that value in *Variable*. When the optional WORD modifier is used, the low byte of *Variable* is read from *Location*, the high byte of *Variable* from *Location* + 1. Any location within the EEPROM can be read (including your PBASIC program's tokens) at run-time. This feature is mainly used to retrieve long-term data from EEPROM; data stored in EEPROM is not lost when the power is removed.

## READ – BASIC Stamp Command Reference

---

The following READ command retrieves the value at location 100 and stores it into the variable called *result*: A SIMPLE READ COMMAND.

```
SYMBOL result = B2
```



```
READ 100, result
```

--Or--

```
result VAR Byte
```



```
READ 100, result
```

The EEPROM is organized as a sequential set of byte-sized memory locations. The READ command normally only retrieves byte-sized values from EEPROM. This does not mean that you can't read word-sized values, however. A word consists of two bytes, called a low-byte and a high-byte. With PBASIC 1.0 or 2.0, if you want to read a word-sized value, you'll need to use two READ commands and a word-sized variable.

READING WORD VALUES VS. BYTE VALUES.

```
SYMBOL result = W0
SYMBOL resultLo = B0
SYMBOL resultHi = B1
```



```
EEPROM (101, 4)
```

```
READ 0, resultLo 'read low byte
READ 1, resultHi 'read high byte
DEBUG #result
```

--Or--

```
result VAR Word
```



```
DATA Word 1125
```

NOTE: this method is required only if using PBASIC 2.0. See section below for PBASIC 2.5 method.

```
READ 0, result.LOWBYTE 'read low byte
READ 1, result.HIGHBYTE 'read high byte
```

On all BS2 models, by using the \$PBASIC 2.5 directive, you can use a single READ command with the WORD modifier. For example,



## 5: BASIC Stamp Command Reference – READ

```
All 2 ' {$PBASIC 2.5}

result VAR Word

DATA Word 1125

READ 0, Word result
DEBUG DEC ? result
```

This code uses the EEPROM or DATA directive to write the low-byte and high-byte of the number 1125 into locations 0 and 1 during download. The BS1 code the uses two READ commands to read each of the bytes out of EEPROM and reconstructs them into a single word-sized variable.

PBASIC 2.5 uses READ's optional WORD modifier to read both bytes from EEPROM and into a word-sized variable. Both programs then display the value on the screen.

When using PBASIC 2.5, a single READ command can retrieve multiple bytes and words from sequential EEPROM locations. For example:

```
All 2 ' {$PBASIC 2.5}

temp VAR Byte
temp2 VAR Word

READ 25, temp, Word temp2 ' retrieve byte from location 25
 ' and word from locations 26 and 27
DEBUG DEC temp, CR ' display value of temp, carriage return
DEBUG DEC temp2 ' display value of temp2
```

### SPECIAL NOTES FOR EEPROM USAGE.

Note that the EEPROM and DATA directives store data in the EEPROM before the program runs, however, the WRITE command can be used to store data while the program is running. Additionally, the EEPROM locations can be read an unlimited number of times, but EEPROM locations can be worn out by excessive writes. See the WRITE command for more information.

When using the READ and WRITE commands, take care to ensure that your program doesn't overwrite itself. On the BS1, location 255 holds the address of the last instruction in your program. Therefore, your program can use any space below the address given in location 255. For example, if location 255 holds the value 100, then your program can use locations 0–99 for data. You can read location 255 at run-time or simply view the

## READ – BASIC Stamp Command Reference

---

Memory Map of the program before you download it. On all BS2 models you will need to view the Memory Map of the program before you download it, to determine the last EEPROM location used. See the "Memory Map Function" section in Chapter 3.

On the BS2p, BS2pe, and BS2px, the READ and WRITE commands can  affect locations in any program slot as set by the STORE command. See the STORE command for more information.

### Demo Program (READ.bs1)

```
' READ.bs1
' This program reads a string of data stored in EEPROM. The EEPROM data is
' downloaded to the BS1 at compile-time and remains there (even with the
' power off) until overwritten. Put ASCII characters into EEPROM, followed
' by 0, which will serve as the end-of-message marker. For programs with
' multiple strings, use the Memory Map window to find the starting
character
' address.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL strAddr = B2
SYMBOL char = B3

Msg1:
 EEPROM ("BS1", 13, "EEPROM Storage!", 0)

Main:
 strAddr = 0 ' set to start of message
 GOSUB String_Out
 END

String_Out:
 READ strAddr, char ' read byte from EEPROM
 strAddr = strAddr + 1 ' point to next character
 IF char = 0 THEN StrOut_Exit ' if 0, exit routine
 DEBUG #@char ' otherwise print char
 GOTO String_Out ' get next character

StrOut_Exit:
 RETURN
```

## 5: BASIC Stamp Command Reference – READ

All 2

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

### Demo Program (READ.bs2)

```
' READ.bs2
' This program reads a string of data stored in EEPROM. The EEPROM data is
' downloaded to the BS2 at compile-time and remains there (even with the
' power off) until overwritten. Put ASCII characters into EEPROM, followed
' by 0, which will serve as the end-of-message marker.

' {$STAMP BS2}
' {$PBASIC 2.5}

strAddr VAR Word
char VAR Byte

Msg1 DATA "BS2", CR, "EEPROM Storage!", 0

Main:
 strAddr = Msg1 ' set to start of message
 GOSUB String_Out
 END

String_Out:
 DO
 READ strAddr, char ' read byte from EEPROM
 strAddr = strAddr + 1 ' point to next character
 IF (char = 0) THEN EXIT ' if 0, exit routine
 DEBUG char ' otherwise print char
 LOOP
 RETURN
```

## ***READ – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – RETURN

### RETURN

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



### RETURN

#### Function

Return from a subroutine, assuming there was a previous GOSUB executed.

#### Quick Facts

Table 5.89: RETURN Quick Facts.

| Related Commands                      | BS1                                                                                   | BS2 Models   |
|---------------------------------------|---------------------------------------------------------------------------------------|--------------|
|                                       | GOSUB                                                                                 | GOSUB and ON |
| Maximum Number of RETURNS per Program | Unlimited. However, the number of GOSUBs are limited. See GOSUB for more information. |              |



NOTE: On the BS1, a RETURN without a GOSUB will return the program to the last GOSUB (or will end the program if no GOSUB was executed).

#### Explanation

RETURN sends the program back to the address (instruction) immediately following the most recent GOSUB. If RETURN is executed without a prior GOSUB, the BASIC Stamp will return to the first executable line of the program; usually resulting in a logical bug in the code. See the GOSUB command for more information.

The example below will start out by GOSUB'ing to the section of code beginning with the label *Hello*. It will print "Hello, my friend!" on the screen then RETURN to the line after the GOSUB... which prints "How are you?" and ENDS.

```
Main:
 GOSUB Hello
 DEBUG "How are you?"
 END
```

```
Hello:
 DEBUG "Hello, my friend!", CR
 RETURN
```

WATCH OUT FOR SUBROUTINES THAT YOUR PROGRAM CAN "FALL INTO."

There's another interesting lesson here; what would happen if we removed the END command from this example? Since the BASIC Stamp reads the code from left to right / top to bottom (like the English language) once it had returned to and run the "How are you?" line, it would naturally "fall into" the *Hello* routine again. Additionally, at the end of the *Hello* routine,

## RETURN – BASIC Stamp Command Reference

---

it would see the RETURN again (although it didn't GOSUB to that routine this time) and because there wasn't a previous place to return to, the BASIC Stamp will start the entire program over again. This would cause an endless loop. The important thing to remember here is to always make sure your program doesn't allow itself to "fall into" a subroutine.

### Demo Program (RETURN.bs2)

```
' RETURN.BS2
' This program demonstrates a potential bug caused by allowing a program to
' "fall into" a subroutine. The program was intended to indicate that it
' is "Starting...", then "Executing Subroutine,", then "Returned..." from
' the subroutine and stop. Since we left out the END command (indicated in
' the comments), the program then falls into the subroutine, displays
' "Executing..." again and then RETURNS to the start of the program and
' runs continuously in an endless loop.

' {$STAMP BS2}

Reset:
 DEBUG "Starting Program", CR ' show start-up

Main:
 PAUSE 1000
 GOSUB Demo_Sub ' call the subroutine
 PAUSE 1000
 DEBUG "Returned from Subroutine", CR ' show that we're back
 PAUSE 1000
 ' <-- Forgot to put END here

Demo_Sub:
 DEBUG " Executing Subroutine", CR ' show subroutine activity
 RETURN
```



NOTE: This example program can be used with the BS1 and all BS2 models by changing the \$STAMP directive accordingly.

## 5: BASIC Stamp Command Reference – REVERSE

### REVERSE

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



#### REVERSE Pin

### Function

Reverse the data direction of the specified pin.

- **Pin** is a variable/constant/expression (0 – 15) that specifies the I/O pin to use. This pin will be placed into the mode opposite its current input/output mode.

### Quick Facts

**Table 5.90:** REVERSE Quick Facts.

|                  | BS1 and all BS2 models |
|------------------|------------------------|
| Related Commands | INPUT and OUTPUT       |

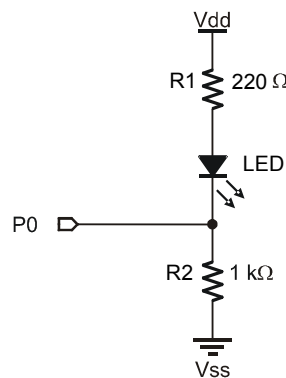
### Explanation

REVERSE is a convenient way to switch the I/O direction of a pin. If the pin is an input, REVERSE makes it an output; if it's an output, REVERSE makes it an input.

Remember that “input” really has two meanings: (1) Setting a pin to input makes it possible to check the state (1 or 0) of external circuitry connected to that pin. The current state is in the corresponding bit of the INS register (PINS on the BS1). (2) Setting a pin to input also disconnects the output driver, the corresponding bit of OUTS (PINS on the BS1).

The demo program below illustrates this second fact with a two-tone LED blinker.

**Figure 5.35:** LED Circuit for Demo Programs.



## REVERSE – BASIC Stamp Command Reference

### Demo Program (REVERSE.bs1)



```
' REVERSE.bs1
' Connect the circuit shown in the REVERSE command description to I/O pin
' 0 and run this program. The LED will alternate between two states, dim
' and bright. The BASIC Stamp is using the REVERSE command to toggling I/O
' pin 0 between input and output states. When pin 0 is an input, current
' flows through R1, through the LED, through R2 to ground. Pin 0 is
' effectively disconnected and doesn't play a part in the circuit. The
' total resistance encountered by current flowing through the LED is R1 +
' R2 = 1220 ohms. When pin 0 is reversed to an output, current flows
' through R1, through the LED, and into pin 0 to ground (because of the 0
' written to PIN0). The total resistance encountered by current flowing
' through the LED is R1,220 ohms. With only 20% of the resistance, the LED
' glows brighter.

' {$STAMP BS1}
' {$PBASIC 1.0}

Setup:
 PIN0 = 0 ' Put a low in the pin 0
 ' output driver

Main:
 PAUSE 250 ' 1/4th second pause
 REVERSE 0 ' reverse pin 0 I/O direction
 GOTO Main ' do forever
```

### Demo Program (REVERSE.bs2)



```
' REVERSE.bs2
' Connect the circuit shown in the REVERSE command description to I/O pin
' 0 and run this program. The LED will alternate between two states, dim
' and bright. The BASIC Stamp is using the REVERSE command to toggling I/O
' pin 0 between input and output states. When pin 0 is an input, current
' flows through R1, through the LED, through R2 to ground. Pin 0 is
' effectively disconnected and doesn't play a part in the circuit. The
' total resistance encountered by current flowing through the LED is R1 +
' R2 = 1220 ohms. When pin 0 is reversed to an output, current flows
' through R1, through the LED, and into pin 0 to ground (because of the 0
' written to OUT0). The total resistance encountered by current flowing
' through the LED is R1,220 ohms. With only 20% of the resistance, the LED
' glows brighter.

' {$STAMP BS2}
' {$PBASIC 2.5}

Setup:
 OUT0 = 0 ' Put a low in the pin 0
 ' output driver
```

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.



## ***5: BASIC Stamp Command Reference – REVERSE***

---

```
Main:
 PAUSE 250 ' 1/4th second pause
 REVERSE 0 ' reverse pin 0 I/O direction
 GOTO Main ' do forever
```

## ***REVERSE – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – RUN

|                        |            |            |             |              |             |              |              |
|------------------------|------------|------------|-------------|--------------|-------------|--------------|--------------|
| <b>RUN</b>             | <b>BS1</b> | <b>BS2</b> | <b>BS2e</b> | <b>BS2sx</b> | <b>BS2p</b> | <b>BS2pe</b> | <b>BS2px</b> |
| <b>RUN ProgramSlot</b> |            |            |             |              |             |              |              |

### Function

Switches execution to another BASIC Stamp program (in a different program slot).

- **ProgramSlot** is a variable/constant/expression (0 – 7) that specifies the program slot to run.

### Quick Facts

Table 5.91: RUN Quick Facts.

|                                            | BS2e                                                        | BS2sx  | BS2p    | BS2pe  | BS2px  |
|--------------------------------------------|-------------------------------------------------------------|--------|---------|--------|--------|
| Time Delay to Switch Between Program Slots | 770 μs                                                      | 300 μs | 250 μs  | 736 μs | 195 μs |
| Related Commands                           | n/a                                                         |        | POLLRUN |        |        |
| Special Notes                              | RUN is similar to a GOTO... you can not "return" from a RUN |        |         |        |        |

### Explanation

The BS2e, BS2sx, BS2p, BS2pe and BS2px have a total of 16 k bytes of code space. This code space is organized into eight slots of 2 k bytes each. The BS2pe has 32 k of EEPROM but only the first 16 k (eight slots) can be used for programs. Up to eight different programs can be downloaded to the BASIC Stamp (one program per code slot). When the BASIC Stamp powers up, or is reset, the program in slot 0 is executed.

The RUN command allows you to activate another program and causes the BASIC Stamp to stay in the newly activated program until it receives another RUN command, or until a power-down or reset condition occurs. The RUN command is similar to a GOTO command in that it allows you to "goto" another program. Normally a master-type program will be used in program slot 0 (since slot 0 runs first) and will control initial execution of the other programs.

A SIMPLE EXAMPLE OF RUN.

Look at the following example (there are two programs here, make sure to download them into program slots 0 and 1, respectively. See the Compiler Directives section of Chapter 3 for more information):

## ***RUN – BASIC Stamp Command Reference***

---

```
' Download this program to Slot 0
```

```
DEBUG "Hello "
RUN 1
```

```
' Download this program to Slot 1
```

```
DEBUG "World!"
PAUSE 1000
RUN 0
```

The above two programs (assuming they have been downloaded into program slots 0 and 1, respectively) will display "Hello World!" on the screen. Program 0 is the first to run and it displays "Hello ", then issues a RUN 1 command. The BASIC Stamp then starts execution of program 1, from its first line of code, which causes "World!" to be displayed. Program 1 then pauses for 1 second and then runs program 0 again.

The I/O pins retain their current state (directions and output latches) and all RAM and SPRAM locations retain their current data during a transition between programs with the RUN command. If sharing data between programs within RAM, make sure to keep similar variable declarations (defined in the same order) in all programs so that the variables align themselves on the proper word, byte, nibble and bit boundaries across programs. The following programs illustrate what happens with mismatched variable declarations:

WHAT HAPPENS TO I/O PINS AND RAM WHEN USING RUN?

```
' Download this program to Slot 0
cats VAR Byte
dogs VAR Byte
```

```
Setup:
 cats = 3
 dogs = 1
 DEBUG ? cats
 DEBUG ? dogs
 RUN 1
```

```
' Download this program to Slot 1
cats VAR Byte
dogs VAR Byte
fleas VAR Word
```

```
Main:
 DEBUG ? cats
 DEBUG ? dogs
 DEBUG ? fleas
 END
```

---

## 5: BASIC Stamp Command Reference – RUN

When the Slot 1 program runs you may be surprised to see that cats and dogs are now zero and fleas are up to 259! – even though we didn't explicitly define them. What happened? The key to remember is that variable names are simply pointers to RAM addresses, and the PBASIC compiler assigns variable names to RAM in descending order by size. This means that in the Slot 1 program, fleas was assigned to RAM locations 0 and 1 which are holding the values 3 and 1 respectively. Since words are stored low-byte first, the value 259 for fleas makes sense ( $3 + (1 * 256)$ ).

There may be occasions when you need to preserve the RAM space in a program slot before calling on another slot that has different variable requirements. You can use the following subroutines to save your RAM space to the SPRAM and restore it on returning from the other program slot.

```
Save_RAM:
 PUT 0, B0 ' move RAM 0 value to SP
 FOR B0 = 1 TO 25 ' loop through other RAM bytes
 PUT B0, B0(B0) ' move RAM value to SP location
 NEXT
 RETURN

Restore_RAM:
 FOR B0 = 1 TO 25 ' loop through RAM
 GET B0, B0(B0) ' retrieve RAM value from SP
 NEXT
 GET 0, B0 ' retrieve RAM 0 value from SP
 RETURN
```

While the use of internal variable names is usually discouraged, these subroutines demonstrate a valid opportunity for their use, as well as the ability to take advantage of the BASIC Stamp's unique memory architecture.

The Save\_RAM routine starts by saving the first byte of RAM (internal name: B0) to location 0 in the SPRAM. This is done so that B0 can be used as a loop index for the other locations. The FOR...NEXT loop provides control of that index. The following line is probably the most difficult to comprehend, but works due to the nature of the BASIC Stamp module's RAM organization

```
PUT B0, B0(B0) ' move RAM value to SP location
```

## RUN – BASIC Stamp Command Reference

In the BASIC Stamp, the variable RAM is implicitly considered an array. What this line is doing, then, is moving the value in each RAM address (B0(1), B0(2), B0(3), ...) to the SPRAM address that corresponds with its byte index. The process is simply reversed to retrieve the RAM variable space.

Any *ProgramSlot* specified above 7 will wrap around and result in running one of the 8 programs (RUN 8 will run program 0, RUN 9 will run program 1, etc).

Review the Advanced Compilation Techniques section beginning on page 68 for more information on downloading multiple programs.

### Demo Program (RUN1.bsx)

```
' RUN1.bsx
' This example demonstrates the use of the RUN command. First, the SPRAM
' location that holds the current slot is read using the GET command to
' display the currently running program number. Then a set of values
' (based on the program number) are displayed on the screen. Afterwards,
' program number 1 is run. This program is a BS2sx project consisting of
' RUN1.BSX and RUN2.BSX, but will run on all multi-slot BASIC Stamp models.

' {$STAMP BS2sx, RUN2.BSX}
' {$PBASIC 2.5}

#SELECT $STAMP ' set SPRAM of slot number
#CASE BS2
 #ERROR "Multi-slot BASIC Stamp required."
#CASE BS2E, BS2SX
 Slot CON 63
#CASE BS2P, BS2PE, BS2PX
 Slot CON 127
#ENDSELECT

slotNum VAR Nib ' current slot
idx VAR Nib ' loop counter
value VAR Byte ' value from EEPROM

EETable DATA 100, 40, 80, 32, 90
 DATA 200, 65, 23, 77, 91

Setup:
 GET Slot, slotNum ' read current slot
 DEBUG "Program #", DEC slotNum, CR ' display

Main:
```



NOTE: This example program was written for the BS2sx but can be used with the BS2e, BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

## 5: BASIC Stamp Command Reference – RUN

```
FOR idx = 0 TO 4 ' read/display table values
 READ (slotNum * 5) + idx, value
 DEBUG DEC3 value, " "
NEXT
DEBUG CR
PAUSE 1000

RUN 1 ' run Slot 1 pgm
```



NOTE: This example program was written for the BS2sx but can be used with the BS2e, BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

### Demo Program (RUN2.bsx)

```
' RUN2.bsx
' This example demonstrates the use of the RUN command. First, the SPRAM
' location that holds the current slot is read using the GET command to
' display the currently running program number. Then a set of values
' (based on the program number) are displayed on the screen. Afterwards,
' program number 0 is run. This program is a BS2sx project consisting of
' RUN1.BSX and RUN2.BSX, but will run on all multi-slot BASIC Stamp models.

' {$STAMP BS2sx}
' {$PBASIC 2.5}

#SELECT $STAMP ' set SPRAM of slot number
#CASE BS2
 #ERROR "Multi-slot BASIC Stamp required."
#CASE BS2E, BS2SX
 Slot CON 63
#CASE BS2P, BS2PE, BS2PX
 Slot CON 127
#ENDSELECT

slotNum VAR Nib ' current slot
idx VAR Nib ' loop counter
value VAR Byte ' value from EEPROM

EETable DATA 100, 40, 80, 32, 90
 DATA 200, 65, 23, 77, 91

Setup:
 GET Slot, slotNum ' read current slot
 DEBUG "Program #", DEC slotNum, CR ' display

Main:
 FOR idx = 0 TO 4 ' read/display table values
 READ (slotNum * 5) + idx, value
 DEBUG DEC3 value, " "
 NEXT
 DEBUG CR
 PAUSE 1000

 RUN 0 ' back to Slot 0 pgm
```

## ***RUN – BASIC Stamp Command Reference***

---



## 5: BASIC Stamp Command Reference – SELECT...CASE

NOTE: SELECT...CASE requires PBASIC 2.5.

All 2

### SELECT...CASE

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|

**SELECT** *Expression*

**CASE** *Condition(s)*

*Statement(s)*

{ **CASE** *Condition(s)*

*Statement(s)... }*

{ **CASE ELSE**

*Statement(s)* }

**ENDSELECT**

### Function

Evaluate *Expression* and conditionally execute a code block based on comparison to *Condition(s)* then continue execution with line following ENDSELECT.

- **Expression** is a variable, a constant, or an expression.
- **Condition** is a statement that, when compared to *Expression*, can be evaluated as True or False. The *Condition* can be a very simple or very complex relationship, as described below. Multiple conditions within the same CASE can be separated by commas ( , ).
- **Statement** is any valid PBASIC instruction.

### Quick Facts

**Table 5.92:** SELECT...CASE Quick Facts.

|                                         | All BS2 Models                                                                                                |
|-----------------------------------------|---------------------------------------------------------------------------------------------------------------|
| <b>Comparison Operators</b>             | =, <>, >, <, >=, <=                                                                                           |
| <b>Conditional Logic Operators</b>      | Not allowed, however multiple conditions can be separated by commas (,) which act like a logical OR operator. |
| <b>Format of Condition</b>              | Comparison <i>Value</i> where <i>Value</i> can be any of variable, constant or expression                     |
| <b>Parentheses</b>                      | Allowed to specify order of execution within expressions.                                                     |
| <b>Maximum Nested SELECT Statements</b> | 16                                                                                                            |
| <b>Maximum CASEs per SELECT</b>         | 16                                                                                                            |
| <b>Related Commands</b>                 | IF...THEN, ON, and BRANCH                                                                                     |

## SELECT...CASE – BASIC Stamp Command Reference

---

### Explanation

SELECT...CASE is an advanced decision-making structure that is often used to replace compound IF...THEN...ELSE structures. SELECT...CASE statements are good for performing one of many possible actions depending on the value of a single expression.

Upon reaching a SELECT...CASE statement, the BASIC Stamp will evaluate *Expression* once and then compare it to the *Condition(s)* of each CASE until it finds a “case” that evaluates to True, or it runs out of cases to compare to. As soon as a True case is found, the BASIC Stamp executes that CASE’s *Statement(s)* and then continues execution on the program line following ENDSELECT.

To understand how SELECT...CASE statements work, it helps to review how IF...THEN statements behave. The condition argument of IF...THEN takes the form:

*Value1 Comparison Value2*

and *Value1* is “compared” to *Value2* using the *Comparison* operator.

In SELECT...CASE statements, the *Value1* component is always *Expression* and so the format of *Condition(s)* is simplified to:

{ *Comparison* } *Value*

*Comparison* is optional and can be any of the comparison operators shown in Table 5.93. If *Comparison* is not specified, it is an implied Equal ( = ) operator. *Value* can be a variable, constant or expression.

| Comparison Operator Symbol | Definition               |
|----------------------------|--------------------------|
| =                          | Equal                    |
| <>                         | Not Equal                |
| >                          | Greater Than             |
| <                          | Less Than                |
| >=                         | Greater Than or Equal To |
| <=                         | Less Than or Equal To    |

**Table 5.93:** Comparison Operators for SELECT...CASE.

*Condition(s)* also has a special, additional format that can be used to indicate a range of sequential values:

## 5: BASIC Stamp Command Reference – SELECT...CASE

---

*Value1 TO Value2*

This indicates a range of Value1 to Value2, inclusive. For example, 20 TO 23 means 20, 21, 22 and 23. Similarly, "A" TO "F" means all the characters in the range "A" through "F".

Finally, multiple conditions can be included in a single CASE by separating them with commas ( , ). For example,

```
CASE 20, 25 TO 30, >100
```

will evaluate to True if the *Expression* (from the SELECT statement) is equal to 20, or is in the range 25 through 30, or is greater than 100.

An example will help clarify this function.

```
'{$PBASIC 2.5}

guess VAR WORD

DEBUG "Guess my favorite number: " ' prompt user

DO
 DEBUGIN DEC Guess ' get answer

 SELECT guess
 CASE < 100 ' less than 100?
 DEBUG CR, "Not even close. Higher."
 CASE > 140 ' greater than 140?
 DEBUG CR, "Too high."
 CASE 100 TO 120, 126 TO 140 ' 100-120 or 126-140?
 DEBUG CR, "Getting closer..."
 CASE 123 ' 123? Got it!
 DEBUG CR, "That's it! 123!"
 DEBUG CR, "Good Guessing!"
 STOP
 CASE 121 TO 125 ' close to 123?
 DEBUG CR, "You're so close!"
 ENDSELECT

 DEBUG CR, "Try again: " ' encourage another try
LOOP
```

This program will ask the user to guess a number, store that value in *guess* and check the results in the SELECT statement. If *guess* is less than 100, the first CASE is true and BASIC Stamp will display "Not even close."

## ***SELECT...CASE – BASIC Stamp Command Reference***

---

Higher.” then continues after the ENDSELECT, displaying “Try again: “ on the screen and giving the user another guess. If *guess* is greater than 140, the second CASE is true, “Too high.” will be displayed followed by “Try again: “, etc. If the user guesses 123, they are congratulated and the program stops.

You may have noticed a potential error in the code: both of the last two CASEs are true if *guess* is 123. So why didn’t the second CASE execute? The answer is because only the first “true” case is executed and any further cases are ignored. If we had swapped the order of the last two cases, we’d have a bug in the program and the user would never find out that 123 was the correct number because “CASE 123” would never have been evaluated. It’s not recommend to write code this way; “CASE 121 TO 122, 124 TO 125” or even “CASE 121, 122, 124, 125” would have been more clear and would prevent potential bugs like this one.

Many situations call for special handling of a few cases and every other case is handled another way. There is a special form of CASE to handle this as well, called CASE ELSE. In our example above, we could have replaced “CASE 121 TO 125” with “CASE ELSE” and it would behave exactly the same. CASE ELSE is a way to ensure that every possible *Expression* value is handled by a case; it’s like saying, “If none of the previous cases were true, run this case.” For a SELECT...CASE statement to work properly, it must have no more than one CASE ELSE statement, and that CASE ELSE statement must be the very last CASE.

The *Condition(s)* have a format similar to that of the IF...THEN commands except, in SELECT...CASE statements, *Expression* is always implied as the first part of the condition.

### **Demo Program (SELECT-CASE.bs2)**

```
' SELECT-CASE.bs2
' This program generates a series of 16-bit random numbers and tests each
' to determine odd or even, and where it falls in the possible range:
' lower third, middle third, or upper third. The program is useful for
' testing various seed values for RANDOM.

' {$STAMP BS2}
' {$PBASIC 2.5}
```



NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## 5: BASIC Stamp Command Reference – SELECT...CASE

```
test VAR Byte ' counter for tests
sample VAR Word ' random number to be tested
odd VAR Byte ' odd throws
even VAR Byte ' even throws
isLo VAR Byte ' sample in lower third
isMid VAR Byte ' in middle thrid
isHi VAR Byte ' in upper third

Main:
 sample = 11000 ' initialize seed
 FOR test = 1 TO 100 ' "throw" 100 times
 RANDOM sample ' randomize

 IF (sample.BIT0) THEN ' check odd/even bit
 odd = odd + 1 ' increment odd count
 ELSE
 even = even + 1 ' increment even count
 ENDIF

 SELECT sample
 CASE <= 21845 ' test lower third
 isLo = isLo + 1

 CASE 21846 TO 43691 ' test middle third
 isMid = isMid + 1

 CASE ELSE ' otherwise upper third
 isHi = isHi + 1
 ENDSELECT
 NEXT

Show_Results:
 DEBUG CLS,
 "Odd Throws.... ", DEC odd, "%", CR,
 "Even Throws... ", DEC even, "%", CR,
 "Low..... ", DEC isLo, "%", CR,
 "Mid..... ", DEC isMid, "%", CR,
 "High..... ", DEC isHi, "%", CR
 END
```

## ***SELECT...CASE – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – SERIN

### SERIN

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**SERIN** *Rpin*, *Baudmode*, { ( *Qualifier* ), } { # } *InputData*



**SERIN** *Rpin* { \Fpin }, *Baudmode*, { *Plabel*, } { *Timeout*, *Tlabel*, } [ *InputData* ]

### Function

Receive asynchronous serial data (e.g., RS-232 data).

- **Rpin** is a variable/constant/expression (0 – 16) that specifies the I/O pin through which the serial data will be received. This pin will be set to input mode. On all BS2 models, if Rpin is set to 16, the BASIC Stamp uses the dedicated serial-input pin (SIN, physical pin 2), which is normally used by the Stamp Editor during the download process.
- **Fpin** is an optional variable/constant/expression (0 – 15) that specifies the I/O pin to indicate flow control status on. This pin will be set to output mode.
- **Baudmode** is variable/constant/expression (0 – 7 on the BS1, 0 – 65535 on all BS2 models) that specifies serial timing and configuration.
- **Qualifier** is an optional variable/constant (0 – 255) indicating data that must be received before execution can continue. Multiple qualifiers can be indicated with commas separating them.
- **Plabel** is an optional label indicating where the program should go in the event of a parity error. This argument should only be provided if *Baudmode* indicates 7 bits, and even parity.
- **Timeout** is an optional variable/constant/expression (0 – 65535) that tells SERIN how long to wait for incoming data. If data does not arrive in time, the program will jump to the address specified by *Tlabel*.
- **Tlabel** is an optional label that must be provided along with *Timeout*, indicating where the program should go in the event that data does not arrive within the period specified by *Timeout*.
- **InputData** is list of variables and formatters that tells SERIN what to do with incoming data. SERIN can store data in a variable or array, interpret numeric text (decimal, binary, or hex) and store the



NOTE: Expressions are not allowed as arguments on the BS1. The range of the *Rpin* argument on the BS1 is 0 – 7.



NOTE: The BS1's *InputData* argument can only be a list of variables and the optional decimal modifier (#).

## SERIN - BASIC Stamp Command Reference

corresponding value in a variable, wait for a fixed or variable sequence of bytes, or ignore a specified number of bytes. These actions can be combined in any order in the *InputData* list.

### Quick Facts

|                              | BS1                           | BS2                                            | BS2e       | BS2sx         | BS2p                          | BS2pe                         | BS2px                         |
|------------------------------|-------------------------------|------------------------------------------------|------------|---------------|-------------------------------|-------------------------------|-------------------------------|
| Units in Timeout             | n/a                           | 1 ms                                           | 1 ms       | 0.4 ms        | 0.4 ms                        | 1 ms                          | 0.4 ms                        |
| Baud range                   | 300, 600, 1200, and 2400 only | 243 to 50K                                     | 243 to 50K | 608 to 115.2K | 608 to 115.2K                 | 243 to 50K                    | 972 to 115.2K                 |
| Baud limit with flow control | n/a                           | 19.2K                                          | 19.2K      | 19.2K         | 19.2K                         | 19.2K                         | 19.2K                         |
| Limit to qualifiers          | Unlimited                     | 6 (in WAIT formatter)                          |            |               |                               |                               |                               |
| I/O pins available           | 0 - 7                         | 0 - 15                                         | 0 - 15     | 0 - 15        | 0 - 15 (in current I/O block) | 0 - 15 (in current I/O block) | 0 - 15 (in current I/O block) |
| Other serial port pins       | n/a                           | SIN pin (physical pin 2) when <i>Rpin</i> = 16 |            |               |                               |                               |                               |
| Related Command              | SEROUT                        | SEROUT and DEBUGIN                             |            |               |                               |                               |                               |

Table 5.94: SERIN Quick Facts.

### Explanation

One of the most popular forms of communication between electronic devices is serial communication. There are two major types of serial communication; asynchronous and synchronous. The SERIN and SEROUT commands are used to receive and send asynchronous serial data. See the SHIFTIN and SHIFTOUT command for information on the synchronous method.

SERIAL COMMUNICATION BACKGROUND.

SERIN can wait for, filter and convert incoming data in powerful ways. SERIN deserves some lengthy discussion, below, since all this power brings some complexity.

The term asynchronous means "no clock." More specifically, "asynchronous serial communication" means data is transmitted and received without the use of a separate "clock" wire. Data can be sent using as little as two wires; one for data and one for ground. The PC's serial ports (also called COM ports or RS-232 ports) use asynchronous serial communication. Note: the other kind of serial communication, synchronous, uses at least three wires; one for clock, one for data and one for ground.

PHYSICAL AND ELECTRICAL DETAILS.



## 5: BASIC Stamp Command Reference – SERIN

RS-232 is the electrical specification for the signals that PC serial ports use. Unlike normal logic, where 5 volts is a logic 1 and 0 volts is logic 0, RS-232 uses -12 volts for logic 1 and +12 volts for logic 0. This specification allows communication over longer wire lengths without amplification.

Most circuits that work with RS-232 use a line driver/receiver. This component does two things: (1) it converts the  $\pm 12$  volts of RS-232 to TTL-compatible 0 to 5-volt levels and (2) it inverts the relationship of the voltage levels, so that 5 volts = logic 1 and 0 volts = logic 0.

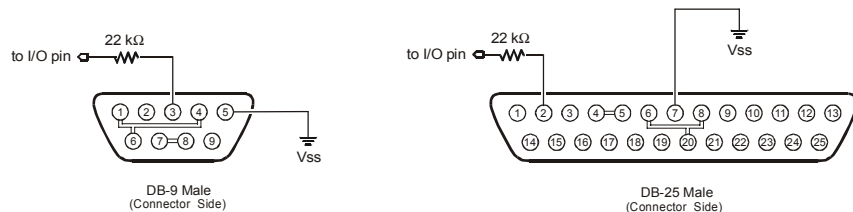
**ALL 2**  
USING THE BUILT-IN SERIAL PORT ON ALL  
BS2 MODELS.

All BS2 models have a line receiver on its SIN pin ( $Rpin = 16$ ). See the "Introduction to the BASIC Stamp" chapter. The SIN pin goes to a PC's serial data-out pin on the DB9 connector built into BASIC Stamp development boards. The connector is wired to allow both programming and run-time serial communication (unless you are using the BASIC Stamp 2 Carrier Board (#27120) which is designed for programming only). For the built-in serial port set the  $Rpin$  argument to 16 in the SERIN command.

**1** **ALL 2**

All BASIC Stamp models (including the BS1) can also receive RS-232 data through any of their I/O pins ( $Rpin = 0 - 7$  for BS1,  $Rpin = 0 - 15$  on all BS2 models). The I/O pins don't need a line receiver, just a 22 k $\Omega$  resistor. The resistor limits current into the I/O pins' built-in clamping diodes, which keep input voltages within a safe range. See Figure 5.36.

**Figure 5.36:** Serial Port Diagram Showing Correct Connections to a BASIC Stamp's I/O pin. NOTE: The 22 k $\Omega$  resistor is not required if connecting to the SIN pin.



| Function                  | DB9 | DB25 |
|---------------------------|-----|------|
| Data Carrier Detect (DCD) | 1   | 8    |
| Receive Data (RD)         | 2   | 3    |
| Transmit Data (TD)        | 3   | 2    |
| Data Terminal Ready (DTR) | 4   | 20   |
| Signal Ground (SG)        | 5   | 7    |
| Data Set Ready (DSR)      | 6   | 6    |
| Request to Send (RTS)     | 7   | 4    |
| Clear to Send (CTS)       | 8   | 5    |

NOTE: The connections shown with double-lines are normally not necessary. They indicate optional connections to disable hardware handshaking (DTR-DSR-DCD and RTS-CTS). This is only necessary if you are using software or hardware that expects hardware handshaking.

## SERIN - BASIC Stamp Command Reference

Figure 5.36 shows the pinouts of the two styles of PC serial ports and how to connect them to the BASIC Stamp's I/O pin (the 22 kΩ resistor is not needed if connecting to the SIN pin). Though not normally needed, the figure also shows loop back connections that defeat hardware handshaking used by some PC software. Note that PC serial ports are always male connectors. The 25-pin style of serial port (called a DB25) looks similar to a printer (parallel) port except that it is male, whereas a parallel port is female.

Asynchronous serial communication relies on precise timing. Both the sender and receiver must be set for identical timing, usually expressed in bits per second (bps) called baud. SERIAL TIMING AND MODE (BAUDMODE).

On all BASIC Stamp models, SERIN requires a value called *Baudmode* that tells it the important characteristics of the incoming serial data; the bit period, number of data and parity bits, and polarity. 1 All 2

On the BS1, serial communication is limited to: no-parity, 8-data bits and 1-stop bit at one of four different speeds: 300, 600, 1200 or 2400 baud. 1 Table 5.95 indicates the *Baudmode* value or symbols to use when selecting the desired mode.

| Baudmode Value | Symbol | Baud Rate | Polarity |
|----------------|--------|-----------|----------|
| 0              | T2400  | 2400      | TRUE     |
| 1              | T1200  | 1200      | TRUE     |
| 2              | T600   | 600       | TRUE     |
| 3              | T300   | 300       | TRUE     |
| 4              | N2400  | 2400      | INVERTED |
| 5              | N1200  | 1200      | INVERTED |
| 6              | N600   | 600       | INVERTED |
| 7              | N300   | 300       | INVERTED |

Table 5.95: BS1 Baudmode Values.

On all BS2 models, serial communication is very flexible. The *Baudmode* argument for SERIN accepts a 16-bit value that determines its characteristics: 1-stop bit, 8-data bits/no-parity or 7-data bits/even-parity and virtually any speed from as low as 300 baud to greater than 100K baud (depending on the BASIC Stamp). All 2 Table 5.96 shows how *Baudmode* is calculated, while Table 5.97, Table 5.98, and Table 5.99 show common baud modes for standard serial baud rates.

## 5: BASIC Stamp Command Reference – SERIN

**Table 5.96:** *Baudmode* calculation for all BS2 models. Add the results of steps 1, 2 and 3 to determine the proper value for the *Baudmode* argument.

|                                                        |                                                                                                                                                                                                                                                      |
|--------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Step 1: Determine the bit period (bits 0 – 11).</b> | BS2, BS2e and BS2pe: = INT(1,000,000 / baud rate) – 20<br>BS2sx and BS2p: = INT(2,500,000 / baud rate) – 20<br>BS2px: = INT(4,000,000 / baud rate) – 20<br>Note: INT means 'convert to integer;' drop the numbers to the right of the decimal point. |
| <b>Step 2: Set data bits and parity (bit 13).</b>      | 8-bit/no-parity = 0<br>7-bit/even-parity = 8192                                                                                                                                                                                                      |
| <b>Step 3: Select polarity (bit 14).</b>               | True (noninverted) = 0<br>Inverted = 16384                                                                                                                                                                                                           |

**Table 5.97:** BS2, BS2e, and BS2pe common baud rates and corresponding *Baudmodes*.

| Baud Rate | 8-bit no-parity inverted | 8-bit no-parity true | 7-bit even-parity inverted | 7-bit even-parity true |
|-----------|--------------------------|----------------------|----------------------------|------------------------|
| 300       | 19697                    | 3313                 | 27889                      | 11505                  |
| 600       | 18030                    | 1646                 | 26222                      | 9838                   |
| 1200      | 17197                    | 813                  | 25389                      | 9005                   |
| 2400      | 16780                    | 396                  | 24972                      | 8588                   |
| 4800*     | 16572                    | 188                  | 24764                      | 8380                   |
| 9600*     | 16468                    | 84                   | 24660                      | 8276                   |

\*The BS2, BS2e and BS2pe may have trouble synchronizing with the incoming serial stream at this rate and higher due to the lack of a hardware input buffer. Use only simple variables and no formatters to try to solve this problem.

**Table 5.98:** BS2sx and BS2p common baud rates and corresponding *Baudmodes*.

| Baud Rate | 8-bit no-parity inverted | 8-bit no-parity true | 7-bit even-parity inverted | 7-bit even-parity true |
|-----------|--------------------------|----------------------|----------------------------|------------------------|
| 1200      | 18447                    | 2063                 | 26639                      | 10255                  |
| 2400      | 17405                    | 1021                 | 25597                      | 9213                   |
| 4800*     | 16884                    | 500                  | 25076                      | 8692                   |
| 9600*     | 16624                    | 240                  | 24816                      | 8432                   |

\*The BS2sx and BS2p may have trouble synchronizing with the incoming serial stream at this rate and higher due to the lack of a hardware input buffer. Use only simple variables and no formatters to try to solve this problem.

**Table 5.99:** BS2px common baud rates and corresponding *Baudmodes*.

| Baud Rate | 8-bit no-parity inverted | 8-bit no-parity true | 7-bit even-parity inverted | 7-bit even-parity true |
|-----------|--------------------------|----------------------|----------------------------|------------------------|
| 1200      | 19697                    | 3313                 | 27889                      | 11505                  |
| 2400      | 18030                    | 1646                 | 26222                      | 9838                   |
| 4800      | 17197                    | 813                  | 25389                      | 9005                   |
| 9600      | 16780                    | 396                  | 24792                      | 8588                   |

### CHOOSING THE PROPER BAUD MODE.

If you're communicating with existing software or hardware, its speed(s) and mode(s) will determine your choice of baud rate and mode. In general, 7-bit/even-parity (7E) mode is used for text, and 8-bit/no-parity (8N) for byte-oriented data. Note: the most common mode is 8-bit/no-parity, even when the data transmitted is just text. Most devices

## SERIN - BASIC Stamp Command Reference

---

that use a 7-bit data mode do so in order to take advantage of the parity feature. Parity can detect some communication errors, but to use it you lose one data bit. This means that incoming data bytes transferred in 7E (even-parity) mode can only represent values from 0 to 127, rather than the 0 to 255 of 8N (no-parity) mode.

Usually a device requires only 1 stop bit per byte. Occasionally, however, you may find a device that requires 2 or more stop bits. Since a stop bit is really just a delay between transmitted bytes (leaving the line in a resting state) the BASIC Stamp can receive transmissions with multiple stop bits per byte without any trouble. In fact, sometimes it is desirable to have multiple stop bits (see the "SERIN Troubleshooting" section, below, for more information).

The example below will receive a single byte through I/O pin 1 at 2400 baud, 8N1, inverted.

```
SYMBOL serData = B2
```

```
SERIN 1, N2400, serData
```

--Or--

```
serData VAR Byte
```

```
SERIN 1, 16780, [serData]
```



Note: This is written with the BS2's *Baudmode* value. Be sure to adjust the value for your BASIC Stamp model.

Here, SERIN will wait for and receive a single byte of data through pin 1 and store it in the variable *serData*. If the BASIC Stamp were connected to a PC running a terminal program (set to the same baud rate) and the user pressed the "A" key on the keyboard, after the SERIN command executed, the variable *serData* would contain 65, the ASCII code for the letter "A" (see the ASCII character chart in Appendix A).

What would happen if, using the example above, the user pressed the "1" key? The result would be that *serData* would contain the value 49 (the ASCII code for the character "1"). This is a critical point to remember: every time you press a character on the keyboard, the computer receives the ASCII value of that character. It is up to the receiving side (in serial communication) to interpret the values as necessary. In this case, perhaps we actually wanted *serData* to end up with the value 1, rather than the ASCII code 49.

A SIMPLE NUMERIC CONVERSION; ASCII TEXT TO DECIMAL.

## 5: BASIC Stamp Command Reference – SERIN

The SERIN command provides a formatter, called the decimal formatter, which will interpret this for us. Look at the following code:

```
1 SYMBOL serData = B2
SERIN 1, N2400, #serData
-- OF --

2 serData VAR Byte
SERIN 1, 16780, [DEC serData]
```

Note: This is written with the BS2's *Baudmode* value. Be sure to adjust the value for your BASIC Stamp.

Notice the decimal formatter in the SERIN command. It is the “#” (for the BS1) or “DEC” (for all BS2 models) that appears just to the left of the *serData* variable. This tells SERIN to convert incoming text representing decimal numbers into true-decimal form and store the result in *serData*. If the user running the terminal software pressed the “1”, “2” and then “3” keys followed by a space or other non-numeric text, the value 123 will be stored in *serData*. Afterwards, the program can perform any numeric operation on the number just like with any other number. Without the decimal formatter, however, you would have been forced to receive each character (“1”, “2” and “3”) separately, and then would still have to do some manual conversion to arrive at the number 123 (one hundred twenty three) before you can do the desired calculations on it.

### DECIMAL FORMATTER SPECIFICS.

The decimal formatter is designed to seek out text that represents decimal numbers. The characters that represent decimal numbers are the characters “0” through “9”. Once the SERIN command is asked to use the decimal formatter for a particular variable, it monitors the incoming serial data, looking for the first decimal character. Once it finds the first decimal character, it will continue looking for more (accumulating the entire multi-digit number) until it finds a non-decimal numeric character. Keep in mind that it will not finish until it finds at least one decimal character followed by at least one non-decimal character.

To further illustrate this, consider the following examples (assuming we’re using the same code example as above):

#### 1) **Serial input:** ABC

**Result:** The BASIC Stamp halts at the SERIN command, continuously waiting for decimal text.

## ***SERIN - BASIC Stamp Command Reference***

---

- 2) **Serial input:** 123 (with no characters following it)  
**Result:** The BASIC Stamp halts at the SERIN command. It recognizes the characters "1", "2" and "3" as the number one hundred twenty three, but since no characters follow the "3", it waits continuously, since there's no way to tell whether 123 is the entire number or not.
- 3) **Serial input:** 123 (followed by a space character)  
**Result:** Similar to example 2, above, except once the space character is received, the BASIC Stamp knows the entire number is 123, and stores this value in *serData*. The SERIN command then ends, allowing the next line of code, if any, to run.
- 4) **Serial input:** 123A  
**Result:** Same as example 3, above. The "A" character, just like the space character, is the first non-decimal text after the number 123, indicating to the BASIC Stamp that it has received the entire number.
- 5) **Serial input:** ABCD123EFGH  
**Result:** Similar to examples 3 and 4 above. The characters "ABCD" are ignored (since they're not decimal text), the characters "123" are evaluated to be the number 123 and the following character, "E", indicates to the BASIC Stamp that it has received the entire number.

For examples of all formatters and how they process incoming data, see Appendix C.

Of course, as with all numbers in the BASIC Stamp, the final result is limited to 16 bits (up to the number 65535). If a number larger than this is received by the decimal formatter, the end result will look strange because the result rolled-over the maximum 16-bit value. **WATCH OUT FOR ROLLOVER ERRORS.**

The BS1 is limited to the decimal formatter shown above, however all the BS2 models have many more conversion formatters available for the SERIN command. If not using a BS1, see the "Additional Conversion Formatters" section below for more information.

## 5: BASIC Stamp Command Reference – SERIN

USING SERIN TO WAIT FOR SPECIFIC DATA BEFORE PROCESSING.

The SERIN command can also be configured to wait for specified data before it retrieves any additional input. For example, suppose a device that is attached to the BASIC Stamp is known to send many different sequences of data, but the only data you desire happens to appear right after the unique characters, “XYZ”. The BS1 has optional *Qualifier* arguments for this purpose. On all BS2 models, a special formatter called WAIT can be used for this.



```
SYMBOL serData = B2
```

```
SERIN 1, N2400, ("XYZ"), #serData
```



-- OR --

```
serData VAR Byte
```

```
SERIN 1, 16780, [WAIT("XYZ"), DEC serData]
```

This is written with the BS2's *Baudmode* value. Be sure to adjust the value for your BASIC Stamp.

The above code waits for the characters “X”, “Y” and “Z” to be received, in that order, and then it looks for a decimal number to follow. If the device in this example were to send the characters “XYZ100” followed by a carriage return or some other non-decimal numeric character, the *serData* variable would end up with the number 100 after the SERIN line finishes. If the device sent some data other than “XYZ” followed by a number, the BASIC Stamp would continue to wait at the SERIN command.

The BS1 will accept an unlimited number of *Qualifiers*. All BS2 models will only accept up to six bytes (characters) in the WAIT formatter.

USING ASCII CODES AND CASE SENSITIVITY.

Keep in mind that when we type “XYZ” into the SERIN command, the BASIC Stamp actually uses the ASCII codes for each of those characters for its tasks. We could also have typed: 88, 89, 90 in place of “XYZ” and the code would run the same way since 88 is the ASCII code for the “X” character, 89 is the ASCII code for the “Y” character, and so on. Also note, serial communication with the BASIC Stamp is case sensitive. If the device mentioned above sent, “xYZ” or “xyZ”, or some other combination of lower and upper-case characters, the BASIC Stamp would have ignored it because we told it to look for “XYZ” (all capital letters).



The BS1's SERIN command is limited to above-mentioned features. If you are not using a BS1, please continue reading about the additional features below.

## ***SERIN - BASIC Stamp Command Reference***

---

The decimal formatter is only one of a whole family of conversion formatters available with SERIN on all the BS2 models. See Table 5.100 for a list of available conversion formatters. All of the conversion formatters work similar to the decimal formatter (as described in the “Decimal Formatter Specifics” section, above). The formatters receive bytes of data, waiting for the first byte that falls within the range of characters they accept (e.g., “0” or “1” for binary, “0” to “9” for decimal, “0” to “9” and “A” to “F” for hex, and “-” for signed variations of any type). Once they receive a numeric character, they keep accepting input until a non-numeric character arrives or (in the case of the fixed length formatters) the maximum specified number of digits arrives.

ADDITIONAL CONVERSION FORMATTERS.





## 5: BASIC Stamp Command Reference – SERIN

**Table 5.100:** Conversion Formatters for all BS2 models.

| Conversion Formatter | Type of Number                                                                                         | Numeric Characters Accepted        | Notes   |
|----------------------|--------------------------------------------------------------------------------------------------------|------------------------------------|---------|
| DEC{1..5}            | Decimal, optionally limited to 1 – 5 digits                                                            | 0 through 9                        | 1       |
| SDEC{1..5}           | Signed decimal, optionally limited to 1 – 5 digits                                                     | -, 0 through 9                     | 1,2     |
| HEX{1..4}            | Hexadecimal, optionally limited to 1 – 4 digits                                                        | 0 through 9, A through F           | 1,3,5   |
| SHEX{1..4}           | Signed hexadecimal, optionally limited to 1 – 4 digits                                                 | -, 0 through 9, A through F        | 1,2,3   |
| IHEX{1..4}           | Indicated hexadecimal, optionally limited to 1 – 4 digits                                              | \$, 0 through 9, A through F       | 1,3,4   |
| ISHEX{1..4}          | Signed, indicated hexadecimal, optionally limited to 1 – 4 digits                                      | -, \$, 0 through 9, A through F    | 1,2,3,4 |
| BIN{1..16}           | Binary, optionally limited to 1 – 16 digits                                                            | 0, 1                               | 1       |
| SBIN{1..16}          | Signed binary, optionally limited to 1 – 16 digits                                                     | -, 0, 1                            | 1,2     |
| IBIN{1..16}          | Indicated binary, optionally limited to 1 – 16 digits                                                  | %, 0, 1                            | 1,4     |
| ISBIN{1..16}         | Signed, indicated binary, optionally limited to 1 – 16 digits                                          | -, %, 0, 1                         | 1,2,4   |
| NUM                  | Generic numeric input (decimal, hexadecimal or binary); hexadecimal or binary number must be indicated | \$, %, 0 through 9, A through F    | 1, 3, 4 |
| SNUM                 | Similar to NUM with value treated as signed with range -32768 to +32767                                | -, \$, %, 0 through 9, A through F | 1,2,3,4 |

- 1 All numeric conversions will continue to accept new data until receiving either the specified number of digits (ex: three digits for DEC3) or a non-numeric character.
- 2 To be recognized as part of a number, the minus sign (-) must immediately precede a numeric character. The minus sign character occurring in non-numeric text is ignored and any character (including a space) between a minus and a number causes the minus to be ignored.
- 3 The hexadecimal formatters are not case-sensitive; “a” through “f” means the same as “A” through “F”.
- 4 Indicated hexadecimal and binary formatters ignore all characters, even valid numerics, until they receive the appropriate prefix (\$ for hexadecimal, % for binary). The indicated formatters can differentiate between text and hexadecimal (ex: ABC would be interpreted by HEX as a number but IHEX would ignore it unless expressed as \$ABC). Likewise, the binary version can distinguish the decimal number 10 from the binary number %10. A prefix occurring in non-numeric text is ignored, and any character (including a space) between a prefix and a number causes the prefix to be ignored. Indicated, signed formatters require that the minus sign come before the prefix, as in -\$1B45.
- 5 The HEX modifier can be used for Decimal to BCD Conversion. See “Hex to BCD Conversion” on page 97.

For examples of all conversion formatters and how they process incoming data, see Appendix C.

## SERIN - BASIC Stamp Command Reference

While very effective at filtering and converting input text, the formatters aren't completely foolproof. As mentioned before, many conversion formatters will keep accepting text until the first non-numeric text arrives, even if the resulting value exceeds the size of the variable. After SERIN, a byte variable will contain the lowest 8 bits of the value entered and a word would contain the lowest 16 bits. You can control this to some degree by using a formatter that specifies the number of digits, such as DEC2, which would accept values only in the range of 0 to 99.

ONCE AGAIN, PAY ATTENTION TO POTENTIAL ROLLOVER ERRORS.

All BS2 models also have special formatters for handling a string of characters, a sequence of characters and undesirable characters. See Table 5.101 for a list of these special formatters. Also, see Appendix C for example serial inputs and the result of using these formatters.

| Special Formatter            | Action                                                                                                                                                                                                                              |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| STR <i>ByteArray</i> \L {E}  | Input a character string of length L into an array. If specified, an end character E causes the string input to end before reaching length L. Remaining bytes are filled with 0s (zeros).                                           |
| WAIT ( <i>Value</i> )        | Wait for a sequence of bytes specified by value. Value can be numbers separated by commas or quoted text (ex: 65, 66, 67 or "ABC"). The WAIT formatter is limited to a maximum of six characters.                                   |
| WAITSTR <i>ByteArray</i> {L} | Wait for a sequence of bytes matching a string stored in an array variable, optionally limited to L characters. If the optional L argument is left off, the end of the array-string must be marked by a byte containing a zero (0). |
| SKIP <i>Length</i>           | Ignore <i>Length</i> bytes of characters.                                                                                                                                                                                           |

**Table 5.101:** SERIN Special Formatters for all BS2 Models.

There is an additional special formatter for the BS2p, BS2pe, and BS2px, shown below.



| Special Formatter | Action                                                                                                                            |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| SPSTR L           | Input a character string of L bytes (up to 126) into Scratch Pad RAM, starting at location 0. Use GET to retrieve the characters. |

**Table 5.102:** Additional SERIN Special Formatter for the BS2p, BS2pe, and BS2px.

The string formatter is useful for receiving a string of characters into a byte array variable. A string of characters is a set of characters that are arranged or accessed in a certain order. The characters "ABC" could be stored in a string with the "A" first, followed by the "B" and then followed by the "C." A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the same size. The string "ABC" could be stored in a byte array containing

THE STR (STRING) FORMATTER.

## 5: BASIC Stamp Command Reference – SERIN

three bytes (elements). See the "Defining Arrays" section in Chapter 4 for more information on arrays.

Here is an example that receives nine bytes through I/O pin 1 at 9600 bps, N81/inverted and stores them in a 10-byte array:

ALL 2

```
serStr VAR Byte(10) ' make 10-byte array
serStr(9) = 0 ' put 0 in last byte of array
SERIN 1, 16468, [STR serStr\9] ' get nine bytes
DEBUG STR serStr ' display
```

NOTE: The rest of the code examples for this section are written for the BS2, using the BS2's *BaudMode* and *Timeout* values. Be sure to adjust the value for your BASIC Stamp.

Why store only 9 bytes in a 10-byte array? We want to reserve space for the 0 byte that many BASIC Stamp string-handling routines regard as an end-of-string marker. This becomes important when dealing with variable-length arrays. For example, the STR formatter (see Table 5.101) can accept an additional parameter telling it to end the string when a particular byte is received, or when the specified length is reached, whichever comes first. An example:

ALL 2

```
serStr VAR Byte(10) ' make 10-byte array
serStr(9) = 0 ' put 0 in last byte of array
SERIN 1, 16468, [STR serStr\9\"]] ' stop at "" or nine bytes
DEBUG STR serStr ' display
```

If the serial input were "hello\*" DEBUG would display "hello" since it collects bytes up to (but not including) the end character. It fills the unused bytes up to the specified length with 0s. DEBUG's normal STR formatter understands a 0 to mean end-of-string. However, if you use DEBUG's fixed-length string modifier, STR *ByteArray*\L, you will inadvertently clear the DEBUG screen. The fixed-length specification forces DEBUG to read and process the 0s at the end of the string, and 0 is equivalent to DEBUG's CLS (clear-screen) control character! Be alert for the consequences of mixing fixed- and variable-length string operations.

MATCHING A SEQUENCE OF CHARACTERS WITH WAIT.

As shown before, SERIN can compare incoming data with a predefined sequence of bytes using the WAIT formatter. The simplest form waits for a sequence of up to six bytes specified as part of the *InputData* list, like so:

ALL 2

```
SERIN 1, 16468, [WAIT("SESAME")]
DEBUG "Password accepted."
```

SERIN will wait for that word, and the program will not continue until it is received. Since WAIT is looking for an exact match for a sequence of

## SERIN - BASIC Stamp Command Reference

---

bytes, it is case-sensitive—"sesame" or "SESAME" or any other variation from "SESAME" would be ignored.

SERIN can also wait for a sequence that matches a string stored in an array variable with the WAITSTR formatter. In the example below, we'll capture a string with STR then have WAITSTR look for an exact match:

MATCHING A SEQUENCE OF CHARACTERS WITH WAITSTR.

```
serStr VAR Byte(10) ' make 10-byte array
serStr(9) = 0 ' put 0 in last byte of array
SERIN 1, 16468, [STR serStr\9\"!"] ' get the string
DEBUG "Waiting for:", STR serStr, CR
SERIN 1, 16468, [WAITSTR serStr] ' wait for match
DEBUG "Password accepted."
```

Ali 2

You can also use WAITSTR with fixed-length strings as in the following example:

```
serStr VAR Byte(4) ' make 4-byte array
DEBUG "Enter 4-character password"
SERIN 1, 16468, [STR serStr\4] ' get the string
DEBUG "Waiting for:", STR serStr\4, CR
SERIN 1, 16468, [WAITSTR serStr\4] ' wait for match
DEBUG "Password accepted."
```

SERIN's *InputData* can be structured as a sophisticated list of actions to perform on the incoming data. This allows you to process incoming data in powerful ways. For example, suppose you have a serial stream that contains "pos: xxxx yyyy" (where xxxx and yyyy are 4-digit numbers) and you want to capture just the decimal y value. The following code would do the trick:

BUILDING COMPOUND *INPUTDATA* STATEMENTS.

Ali 2

```
yOffset VAR Word
SERIN 1, 16468, [WAIT("pos: "), SKIP 4, DEC4 yOffset]
DEBUG ? yOffset
```

Ali 2

The items of the *InputData* list work together to locate the label "pos: ", skip over the four-byte x data, then convert and capture the decimal y data. This sequence assumes that the x data is always four digits long; if its length varies, the following code would be more appropriate:

```
yOffset VAR Word
SERIN 1, 16468, [WAIT("pos: "), DEC yOffset, DEC4 yOffset]
DEBUG ? yOffset
```

Ali 2

## 5: BASIC Stamp Command Reference – SERIN

### USING PARITY AND HANDLING PARITY ERRORS.

The unwanted *x* data is stored in *yOffset* then replaced by the desired *y* data. This is a sneaky way to filter out a number of any size without using an extra variable. With a little creativity, you can combine the *InputData* modifiers to filter and extract almost any data.

Parity is a simple error-checking feature. When a serial sender is set for even parity (the mode the BASIC Stamp supports) it counts the number of 1s in an outgoing byte and uses the parity bit to make that number even. For instance, if it is sending the 7-bit value: %0011010, it sets the parity bit to 1 in order to make an even number of 1s (four).

The receiver also counts the data bits to calculate what the parity bit should be. If it matches the parity bit received, the serial receiver assumes that the data was received correctly. Of course, this is not necessarily true, since two incorrectly received bits could make parity seem correct when the data was wrong, or the parity bit itself could be bad when the rest of the data was OK.

Many systems that work exclusively with text use (or can be set for) 7-bit/even-parity mode. Table 5.97 and Table 5.98 show appropriate *Baudmode* settings for different BASIC Stamp models. For example, with the BS2, to receive one data byte through pin 1 at 9600 baud, 7E, inverted:

```

[All 2] serData VAR Byte
SERIN 1, 24660, [serData]
```

That instruction will work, but it doesn't tell the BS2 what to do in the event of a parity error. Here's an improved version that uses the optional *Plabel* argument:

```

[All 2] serData VAR Byte

Main:
 SERIN 1, 24660, Bad_Data, [serData]
 DEBUG ? serData
 STOP

Bad_Data:
 DEBUG "Parity error."
 STOP
```

If the parity matches, the program continues at the DEBUG instruction after SERIN. If the parity doesn't match, the program goes to the label

## SERIN - BASIC Stamp Command Reference

---

*Bad\_Data*. Note that a parity error takes precedence over other *InputData* specifications (as soon as an error is detected, SERIN aborts and goes to the *Plabel* routine).

In all the examples above, the only way to end the SERIN instruction (other than RESET or power-off) is to give SERIN the serial data it wants. If no serial data arrives, the program is stuck. However, you can tell the BASIC Stamp to abort SERIN if it doesn't receive a byte within a specified number of milliseconds. For instance, to receive a decimal number through pin 1 at 9600 baud, 8N, inverted and abort SERIN after 2 seconds (2000 ms) of inactivity on the serial input:

USING THE SERIAL TIME-OUT FEATURE.

```
result VAR Word
```



```
Main:
```

```
 SERIN 1, 16468, 2000, No_Data, [DEC result]
 DEBUG CLS, ? result
 STOP
```

```
No_Data:
```

```
 DEBUG CLS, "Timeout error"
 STOP
```

If no data arrives within 2 seconds, the program aborts SERIN and continues at the label *No\_Data*. Note that on multi-byte input, the timeout timer is reset after the receipt of any valid data byte; with long timeout values this factor could have an adverse affect on program operation if data packets are transmitted with gaps between individual data bytes. Finally, be cautious when using very short timeout values. Without external flow control, very short timeout values may cause the program to branch to the Tlabel address unnecessarily.

Here's a very important concept: this timeout feature is not picky about the kind of data SERIN receives; if any serial data is received, it prevents the timeout. In the example above, SERIN wants a decimal number. But even if SERIN received letters "ABCD..." at intervals of less than two seconds, it would never abort.

REMEMBER: *TIMEOUT* DOES NOT CARE WHAT KIND OF DATA IS RECEIVED, ONLY THAT DATA IS RECEIVED OR NOT!

You can combine parity and serial timeouts. Here is an example for the BS2 designed to receive a decimal number through pin 1 at 9600 baud, 7E, inverted with a 10-second timeout:

COMBINING PARITY AND TIME-OUT.

## 5: BASIC Stamp Command Reference – SERIN

```

All 2 ' {$PBASIC 2.5}

result VAR Word

Main:
 DO
 SERIN 1, 24660, Bad_Data, 10000, No_Data, [DEC result]
 DEBUG CLS, ? result
 LOOP

Bad_Data:
 DEBUG CLS, "Parity error"
 GOTO Main

No_Data:
 DEBUG CLS, "Timeout error"
 GOTO Main
```

CONTROLLING DATA FLOW.

When you design an application that requires serial communication between BASIC Stamp modules, you have to work within these limitations:

- When the BASIC Stamp is sending or receiving data, it can't execute other instructions.
- When the BASIC Stamp is executing other instructions, it can't send or receive data. *The BASIC Stamp does not have a serial buffer as there is in PCs.* At most serial rates, the BASIC Stamp cannot receive data via SERIN, process it, and execute another SERIN in time to catch the next chunk of data, unless there are significant pauses between data transmissions.

These limitations can sometimes be addressed by using flow control; the *Fpin* option for SERIN and SEROUT (at baud rates of up to the limitation shown in Table 5.94). Through *Fpin*, SERIN can tell a BASIC Stamp sender when it is ready to receive data. (For that matter, *Fpin* flow control follows the rules of other serial handshaking schemes, but most computers other than the BASIC Stamp cannot start and stop serial transmission on a byte-by-byte basis. That's why this discussion is limited to communication between BASIC Stamp modules.)

Here's an example using flow control on the BS2 (data through I/O pin 1, flow control through I/O pin 0, 9600 baud, N8, noninverted):

```

All 2 serData VAR Byte

SERIN 1\0, 84, [serData]
```

## SERIN - BASIC Stamp Command Reference

When SERIN executes, I/O pin 1 (*Rpin*) is made an input in preparation for incoming data, and I/O pin 0 (*Fpin*) is made output low, to signal “go” to the sender. After SERIN finishes receiving, I/O pin 0 goes high to tell the sender to stop. If an inverted *BaudMode* had been specified, the *Fpin*’s responses would have been reversed. Here’s the relationship of serial polarity to *Fpin* states.

|              | Ready to Receive ("Go") | Not Ready to Receive ("Stop") |
|--------------|-------------------------|-------------------------------|
| Inverted     | Fpin is High (1)        | Fpin is Low (0)               |
| Non-inverted | Fpin is Low (0)         | Fpin is High (1)              |

**Table 5.103:** Flow control pin states in relation to polarity (inverted or non-inverted) for all BS2 models.

See the demo program, below, for a flow control example using two BS2s. In the demo program example, without flow control, the sender would transmit the whole word “Hello!” in about 6 ms. The receiver would catch the first byte at most; by the time it got back from the first 1-second PAUSE, the rest of the data would be long gone. With flow control, communication is flawless since the sender waits for the receiver to catch up.

In Figure 5.37, I/O pin 0, *Fpin*, is pulled to ground through a 10k resistor. This is to ensure that the sender sees a stop signal (0 for inverted communications) when the receiver is being programmed.

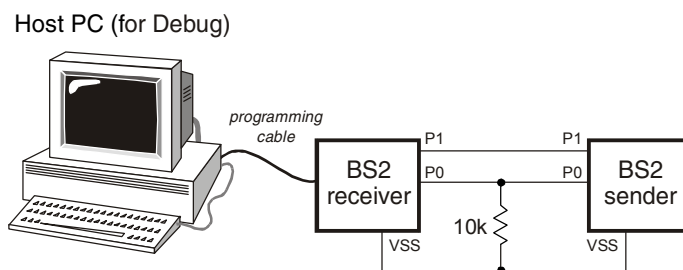


**Figure 5.37:** Flow-Control Example Circuit.

Serial communication, because of its complexity, can be very difficult to work with at times. Please follow these guidelines when developing a project using the SERIN and SEROUT commands:

1. Always build your project in steps.
  - a. Start with small, manageable pieces of code, that deals with serial communication) and test them, one at a time.



## ***5: BASIC Stamp Command Reference – SERIN***

---

- b. Add more and more small pieces, testing them each time, as you go.
  - c. Never write a large portion of code that works with serial communication without testing its smallest workable pieces first.
- 2. Pay attention to timing.
  - a. Be very careful to calculate and overestimate the amount of time operations should take within the BASIC Stamp. Misunderstanding the timing constraints is the source of most problems with code that communicate serially.
  - b. If the serial communication in your project is bi-directional, the above statement is even more critical.
- 3. Pay attention to wiring.
  - a. Take extra time to study and verify serial communication wiring diagrams. A mistake in wiring can cause strange problems in communication, or no communication at all. Make sure to connect the ground pins (Vss) between the devices that are communicating serially.
- 4. Verify port setting on the PC and in the SERIN/SEROUT commands.
  - a. Unmatched settings on the sender and receiver side will cause garbled data transfers or no data transfers. If the data you receive is unreadable, it is most likely a baud rate setting error.
- 5. If receiving data from another device that is not a BASIC Stamp, try to use baud rates of 4800 and below.
  - a. Because of additional overhead in the BASIC Stamp, and the fact that the BASIC Stamp has no hardware receive buffer for serial communication, received data may sometimes be missed or garbled. If this occurs, try lowering the baud rate (if possible), adding extra stop bits, and not using formatters in the SERIN command. Using simple variables (not arrays) and no formatters will increase the chance that the BASIC Stamp can receive the data properly.
- 6. Be sure to study the effects of SERIN formatters.
  - a. Some formatters have specific requirements that may cause problems in received data. For example, the DEC formatter requires a non-decimal-numeric character to follow the received number before it will allow the BASIC

## SERIN - BASIC Stamp Command Reference

---

Stamp to continue. See Appendix C for example input data and the effects on formatters.

### Demo Program (SERIN.bs1)



```
' SERIN.bs1
' This program waits for the characters "A", "B", "C", and "D" to arrive
' at the serial input (inverted, 2400 baud, N81), followed by a number,
' then a carriage-return or some other non-number character. The number is
' then displayed in the Debug window.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL SIn = 0
SYMBOL Baud = N2400

SYMBOL result = W1

Main:
 SERIN SIn, Baud, ("ABCD"), #result
 DEBUG #result, CR
 GOTO Main
END
```

### Demo Program (SERIN\_SEROUT1.bs2)



```
' SERIN_SEROUT1.bs2
' Using two BS2-IC's, connect the circuit shown in the SERIN command
' description and run this program on the BASIC Stamp designated as the
' Sender. This program demonstrates the use of Flow Control (FPin).
' Without flow control, the sender would transmit the whole word "Hello!"
' in about 1.5 ms. The receiver would catch the first byte at most; by the
' time it got back from the first 1-second PAUSE, the rest of the data
' would be long gone. With flow control, communication is flawless since
' the sender waits for the receiver to catch up.

' {$STAMP BS2}
' {$PBASIC 2.5}

SO PIN 1 ' serial output
FC PIN 0 ' flow control pin

#SELECT $STAMP
#CASE BS2, BS2E, BS2PE
 T1200 CON 813
 T2400 CON 396
 T9600 CON 84
 T19K2 CON 32
```

NOTE: This example program was written for the BS2 but it can be used with the BS2e, BS2sx, BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

## 5: BASIC Stamp Command Reference – SERIN

```

T38K4 CON 6
#CASE BS2SX, BS2P
 T1200 CON 2063
 T2400 CON 1021
 T9600 CON 240
 T19K2 CON 110
 T38K4 CON 45
#CASE BS2PX
 T1200 CON 3313
 T2400 CON 1646
 T9600 CON 396
 T19K2 CON 188
 T38K4 CON 84
#ENDSELECT

Inverted CON $4000
Open CON $8000
Baud CON T38K4 + Inverted

Main:
DO
 SEROUT SO\FC, Baud, ["Hello!", CR] ' send the greeting
 PAUSE 2500 ' wait 2.5 seconds
LOOP ' repeat forever
END

```



### Demo Program (SERIN\_SEROUT2.bs2)

NOTE: This example program was written for the BS2 but it can be used with the BS2e, BS2sx, BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

```

' SERIN_SEROUT2.bs2
' Using two BS2-IC's, connect the circuit shown in the SERIN command
' description and run this program on the BASIC Stamp designated as the
' Receiver. This program demonstrates the use of Flow Control (FPin).
' Without flow control, the sender would transmit the whole word "Hello!"
' in about 1.5 ms. The receiver would catch the first byte at most; by the
' time it got back from the first 1-second PAUSE, the rest of the data
' would be long gone. With flow control, communication is flawless since
' the sender waits for the receiver to catch up.

' {$STAMP BS2}
' {$PBASIC 2.5}

SI PIN 0 ' serial input
FC PIN 1 ' flow control pin

#SELECT $STAMP
#CASE BS2, BS2E, BS2PE
 T1200 CON 813
 T2400 CON 396
 T9600 CON 84
 T19K2 CON 32
 T38K4 CON 6

```

## ***SERIN - BASIC Stamp Command Reference***

---

```
#CASE BS2SX, BS2P
 T1200 CON 2063
 T2400 CON 1021
 T9600 CON 240
 T19K2 CON 110
 T38K4 CON 45
#CASE BS2PX
 T1200 CON 3313
 T2400 CON 1646
 T9600 CON 396
 T19K2 CON 188
 T38K4 CON 84
#ENDSELECT

Inverted CON $4000
Open CON $8000
Baud CON T38K4 + Inverted

letter VAR Byte

Main:
DO
 SERIN SI\FC, Baud, [letter] ' receive one byte
 DEBUG letter ' display on screen
 PAUSE 1000 ' wait one second
LOOP ' repeat forever
END
```

## 5: BASIC Stamp Command Reference – SEROUT

### SEROUT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



SEROUT *Tpin*, *Baudmode*, ( *#* ) *OutputData* )



SEROUT *Tpin* { *\Fpin* }, *Baudmode*, { *Pace*, } { *Timeout*, *Tlabel*, } [ *OutputData* ]

### Function

Transmit asynchronous serial data (e.g., RS-232 data).



NOTE: Expressions are not allowed as arguments on the BS1. The range of the *Rpin* argument on the BS1 is 0 – 7.

- ***Tpin*** is a variable/constant/expression (0 – 16) that specifies the I/O pin through which the serial data will be transmitted. This pin will be set to output mode. On all BS2 models, if *Tpin* is set to 16, the BASIC Stamp uses the dedicated serial-output pin (SOUT, physical pin 1), which is normally used by the Stamp Editor during the download process.
- ***Fpin*** is an optional variable/constant/expression (0 – 15) that specifies the I/O pin to monitor for flow control status. This pin will be set to input mode. NOTE: *Fpin* must be specified to use the optional *Timeout* and *Tlabel* arguments in the SEROUT command.
- ***Baudmode*** is variable/constant/expression (0 – 7 on the BS1, 0 – 65535 on all BS2 models) that specifies serial timing and configuration.
- ***Pace*** is an optional variable/constant/expression (0 – 65535) that determines the length of the pause between transmitted bytes. NOTE: *Pace* cannot be used simultaneously with *Timeout* and *Fpin*.
- ***Timeout*** is an optional variable/constant/expression (0 – 65535) that tells SEROUT how long to wait for *Fpin* permission to send. If permission does not arrive in time, the program will jump to the address specified by *Tlabel*. NOTE: *Fpin* must be specified to use the optional *Timeout* and *Tlabel* arguments in the SEROUT command.
- ***Tlabel*** is an optional label that must be provided along with *Timeout*. *Tlabel* indicates where the program should go in the event that permission to send data is not granted within the period specified by *Timeout*.
- ***OutputData*** is list of variables, constants, expressions and formatters that tells SEROUT how to format outgoing data. SEROUT can transmit individual or repeating bytes, convert values into decimal,



NOTE: The BS1's *OutputData* argument can only be a list of variables and the optional decimal modifier (#).

## SEROUT – BASIC Stamp Command Reference

hex or binary text representations, or transmit strings of bytes from variable arrays. These actions can be combined in any order in the *OutputData* list.

### Quick Facts

|                              | BS1                           | BS2, BS2e                                                                                                                                    | BS2sx         | BS2p                          | BS2pe                         | BS2px                         |
|------------------------------|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|---------------|-------------------------------|-------------------------------|-------------------------------|
| Units in <i>Pace</i>         | n/a                           | 1 ms                                                                                                                                         | 1 ms          | 1 ms                          | 1 ms                          | 1 ms                          |
| Units in <i>Timeout</i>      | n/a                           | 1 ms                                                                                                                                         | 0.4 ms        | 0.4 ms                        | 1 ms                          | 0.4 ms                        |
| Baud range                   | 300, 600, 1200, and 2400 only | 243 to 50K                                                                                                                                   | 608 to 115.2K | 608 to 115.2K                 | 243 to 50K                    | 972 to 115.2K                 |
| Baud limit with flow control | n/a                           | 19.2K                                                                                                                                        | 19.2K         | 19.2K                         | 19.2K                         | 19.2K                         |
| I/O Pins Available           | 0 - 7                         | 0 - 15                                                                                                                                       | 0 - 15        | 0 - 15 (in current I/O block) | 0 - 15 (in current I/O block) | 0 - 15 (in current I/O block) |
| Other serial port pins       | n/a                           | SOUT pin (physical pin 1) when <i>Tpin</i> = 16                                                                                              |               |                               |                               |                               |
| Special cases                | n/a                           | <i>Fpin</i> must be specified to use <i>Timeout</i> and <i>Tlabel</i> . <i>Pace</i> cannot be specified at the same time as <i>Timeout</i> . |               |                               |                               |                               |
| Related Commands             | SERIN                         | SERIN and DEBUG                                                                                                                              |               |                               |                               |                               |

Table 5.104: SEROUT Quick Facts.

### Explanation

One of the most popular forms of communication between electronic devices is serial communication. There are two major types of serial communication; asynchronous and synchronous. The SERIN and SEROUT commands are used to receive and send asynchronous serial data. See the SHIFTIN and SHIFTOUT command for information on the synchronous method.

SERIAL COMMUNICATION BACKGROUND.

The following information is supplemental to what is discussed in the SERIN command section. Please read through the SERIN command section for additional information.

All BS2 models have a line driver on its SOUT pin (*Tpin* = 16). The SOUT pin goes to a PC's serial data-in pin on the DB9 connector built into BASIC Stamp development boards. The connector is wired to allow both programming and run-time serial communication (unless you are using the BASIC Stamp 2 Carrier Board (#27120) which is only designed for programming). For the built-in serial port set the *Tpin* argument to 16 in the SEROUT command.

USING THE BUILT-IN SERIAL PORT FOUND ON ALL BS2 MODELS.

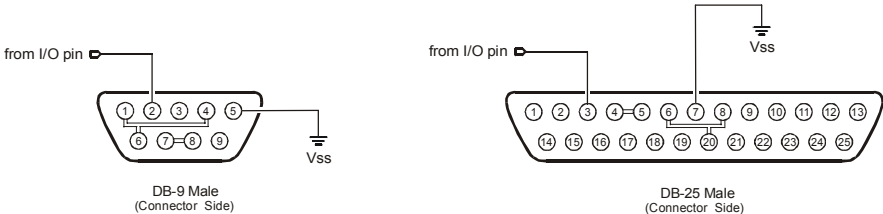


# 5: BASIC Stamp Command Reference – SEROUT

 All BASIC Stamp models (including the BS1) can also transmit RS-232 data through any of their I/O pins ( $Tpin = 0 - 7$  for BS1,  $Tpin = 0 - 15$  on all other BASIC Stamp models). The I/O pins only provide a 0 to +5 volt swing (outside of RS-232 specs) and may need to be connected through a line driver for proper operation with all serial ports. Most serial ports are able to recognize a 0 to +5 volt swing, however. See Figure 5.38 for sample wiring.

**Figure 5.38:** Serial port diagram showing correct connections to a BASIC Stamp's I/O pin.

NOTE: A line driver may have to be used between the I/O pin and the receiving serial port to ensure proper communication.



| Function                  | DB9 | DB25 |
|---------------------------|-----|------|
| Data Carrier Detect (DCD) | 1   | 8    |
| Receive Data (RD)         | 2   | 3    |
| Transmit Data (TD)        | 3   | 2    |
| Data Terminal Ready (DTR) | 4   | 20   |
| Signal Ground (SG)        | 5   | 7    |
| Data Set Ready (DSR)      | 6   | 6    |
| Request to Send (RTS)     | 7   | 4    |
| Clear to Send (CTS)       | 8   | 5    |

NOTE: The connections shown with double-lines are normally not necessary. They indicate optional connections to disable hardware handshaking (DTR-DSR-DCD and RTS-CTS). This is only necessary if you are using software or hardware that expects hardware handshaking.

Figure 5.38 shows the pinouts of the two styles of PC serial ports and how to connect them to the BASIC Stamp's I/O pin. Though not normally needed, the figure also shows loop back connections that defeat hardware handshaking used by some PC software. Note that PC serial ports are always male connectors. The 25-pin style of serial port (called a DB25) looks similar to a printer (parallel) port except that it is male, whereas a parallel port is female.

## SERIAL TIMING AND MODE (BAUDMODE).

Asynchronous serial communication relies on precise timing. Both the sender and receiver must be set for identical timing, usually expressed in bits per second (bps) called baud.

SEROUT requires a value called *Baudmode* that tells it the important characteristics of the outgoing serial data; the bit period, number of data and parity bits, and polarity.

## SEROUT – BASIC Stamp Command Reference

On the BS1, serial communication is limited to: no-parity, 8-data bits and 1-stop bit at one of four different speeds: 300, 600, 1200 or 2400 baud. Table 5.105 indicates the Baudmode value or symbols to use when selecting the desired mode.

| Baudmode Value | Symbol | Baud Rate | Polarity and Output Mode           |
|----------------|--------|-----------|------------------------------------|
| 0              | T2400  | 2400      | TRUE (always driven)               |
| 1              | T1200  | 1200      | TRUE (always driven)               |
| 2              | T600   | 600       | TRUE (always driven)               |
| 3              | T300   | 300       | TRUE (always driven)               |
| 4              | N2400  | 2400      | INVERTED (always driven)           |
| 5              | N1200  | 1200      | INVERTED (always driven)           |
| 6              | N600   | 600       | INVERTED (always driven)           |
| 7              | N300   | 300       | INVERTED (always driven)           |
| 8              | OT2400 | 2400      | TRUE (open drain, driven high)     |
| 9              | OT1200 | 1200      | TRUE (open drain, driven high)     |
| 10             | OT600  | 600       | TRUE (open drain, driven high)     |
| 11             | OT300  | 300       | TRUE (open drain, driven high)     |
| 12             | ON2400 | 2400      | INVERTED (open source, driven low) |
| 13             | ON1200 | 1200      | INVERTED (open source, driven low) |
| 14             | ON600  | 600       | INVERTED (open source, driven low) |
| 15             | ON300  | 300       | INVERTED (open source, driven low) |

**Table 5.105:** BS1 *Baudmode* Values.

On all BS2 models, serial communication is very flexible. The *Baudmode* argument for SEROUT accepts a 16-bit value that determines its characteristics: 1-stop bit, 8-data bits/no-parity or 7-data bits/even-parity and virtually any speed from as low as 300 baud to greater than 100K baud (depending on the BASIC Stamp model). Table 5.106 shows how *Baudmode* is calculated, while Table 5.107, Table 5.108, and Table 5.109 show common baud modes for standard serial baud rates.

|                                                       |                                                                                                                                                                                                                                                                     |
|-------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Step 1: Determine the bit period (bits 0 – 11)</b> | BS2, BS2e and BS2pe: = INT(1,000,000 / baud rate) – 20<br>BS2sx and BS2p: = INT(2,500,000 / baud rate) – 20<br>BS2px: = INT(4,000,000 / baud rate) – 20<br><small>Note: INT means 'convert to integer;' drop the numbers to the right of the decimal point.</small> |
| <b>Step 2: Set data bits and parity (bit 13)</b>      | 8-bit/no-parity = 0<br>7-bit/even-parity = 8192                                                                                                                                                                                                                     |
| <b>Step 3: Select polarity (bit 14)</b>               | True (noninverted) = 0<br>Inverted = 16384                                                                                                                                                                                                                          |
| <b>Step 4: Select driven or open output (bit 15)</b>  | Driven = 0<br>Open = 32768                                                                                                                                                                                                                                          |

**Table 5.106:** *Baudmode* calculation for all BS2 models. Add the results of steps 1, 2, 3 and 4 to determine the proper value for the *Baudmode* argument.



## 5: BASIC Stamp Command Reference – SEROUT

**Table 5.107:** BS2, BS2e and BS2pe common baud rates and corresponding *Baudmodes*.

| Baud Rate | 8-bit no-parity inverted | 8-bit no-parity true | 7-bit even-parity inverted | 7-bit even-parity true |
|-----------|--------------------------|----------------------|----------------------------|------------------------|
| 300       | 19697                    | 3313                 | 27889                      | 11505                  |
| 600       | 18030                    | 1646                 | 26222                      | 9838                   |
| 1200      | 17197                    | 813                  | 25389                      | 9005                   |
| 2400      | 16780                    | 396                  | 24972                      | 8588                   |
| 4800      | 16572                    | 188                  | 24764                      | 8380                   |
| 9600      | 16468                    | 84                   | 24660                      | 8276                   |

NOTE: For "open" baudmodes used in networking, add 32768 to the values from the table above. If the dedicated serial port (*Tpin*=16) is used, the data is inverted and driven regardless of the baudmode setting.

**Table 5.108:** BS2sx and BS2p common baud rates and corresponding *Baudmodes*.

| Baud Rate | 8-bit no-parity inverted | 8-bit no-parity true | 7-bit even-parity inverted | 7-bit even-parity true |
|-----------|--------------------------|----------------------|----------------------------|------------------------|
| 1200      | 18447                    | 2063                 | 26639                      | 10255                  |
| 2400      | 17405                    | 1021                 | 25597                      | 9213                   |
| 4800      | 16884                    | 500                  | 25076                      | 8692                   |
| 9600      | 16624                    | 240                  | 24816                      | 8432                   |

NOTE: For "open" baudmodes used in networking, add 32768 to the values from the table above. If the dedicated serial port (*Tpin*=16) is used, the data is inverted and driven regardless of the baudmode setting.

**Table 5.109:** BS2px common baud rates and corresponding *Baudmodes*.

| Baud Rate | 8-bit no-parity inverted | 8-bit no-parity true | 7-bit even-parity inverted | 7-bit even-parity true |
|-----------|--------------------------|----------------------|----------------------------|------------------------|
| 1200      | 19697                    | 3313                 | 27889                      | 11505                  |
| 2400      | 18030                    | 1646                 | 26222                      | 9838                   |
| 4800      | 17197                    | 813                  | 25389                      | 9005                   |
| 9600      | 16780                    | 396                  | 24792                      | 8588                   |

### CHOOSING THE PROPER BAUD MODE.

If you're communicating with existing software or hardware, its speed(s) and mode(s) will determine your choice of baud rate and mode. See the SERIN command description for more information.

### A SIMPLE FORM OF SEROUT.

The example below will transmit a single byte through I/O pin 1 at 2400 baud, 8N1, inverted:

 SEROUT 1, N2400, (65)

--or--

 SEROUT 1, 16780, [65]

This is written with the BS2's *Baudmode* value. Be sure to adjust the value for your BASIC Stamp model.

## SEROUT – BASIC Stamp Command Reference

---

Here, SEROUT will transmit a byte equal to 65 (the ASCII value of the character "A") through pin 1. If the BASIC Stamp were connected to a PC running a terminal program (set to the same baud rate) the character "A" would appear on the screen (see the ASCII character chart in Appendix A).

What if you really wanted the value 65 to appear on the screen? If you remember from the discussion in the SERIN command, "It is up to the receiving side (in serial communication) to interpret the values..." In this case, the PC is interpreting the byte-sized value to be the ASCII code for the character "A". Unless you're also writing the software for the PC, you can't change how the PC interprets the incoming serial data, so to solve this problem, the data needs to be translated before it is sent.

A SIMPLE NUMERIC CONVERSION; DECIMAL TO ASCII NUMERIC TEXT.

The SEROUT command provides a formatter, called the decimal formatter, which will translate the value 65 to two ASCII codes for the characters "6" and "5" and then transmit them. Look at the following code:

```
SEROUT 1, N2400, (#65)
```



--OR--

```
SEROUT 1, 16780, [DEC 65]
```



Notice the decimal formatter in the SEROUT command. It is the "#" (for the BS1) or "DEC" (for all BS2 models) that appears just to the left of the number 65. This tells SEROUT to convert the number into separate ASCII characters which represent the value in decimal form. If the value 65 in the code were changed to 123, the SEROUT command would send three bytes (49, 50 and 51) corresponding to the characters "1", "2" and "3".

This is written with the BS2's *Baudmode* value. Be sure to adjust the value for your BASIC Stamp model.

All BS2 models have many more conversion formatters available for the SEROUT command. See the "Additional Conversion Formatters" section below for more information.



The SEROUT command sends quoted text exactly as it appears in the *OutputData* list:

```
SEROUT 1, N2400, ("Hello" CR)
SEROUT 1, N2400, ("Num = ", #100)
```



--OR--

## 5: BASIC Stamp Command Reference – SEROUT



```
SEROUT 1, 16780, ["Hello", CR]
SEROUT 1, 16780, ["Num = ", DEC 100]
```

This is written with the BS2's *Baudmode* value. Be sure to adjust the value for your BASIC Stamp model.

The above code will display "HELLO" on one line and "Num = 100" on the next line. Notice that you can combine data to output in one SEROUT command, separated by commas. In the example above, we could have written it as one line of code, with "HELLO", CR, "Num = ", DEC 100 in the *OutputData* list.



The BS1's SEROUT command is limited to above-mentioned features. If you are not using a BS1, please continue reading about the additional features below.

USING SEROUT'S *Pace* ARGUMENT TO INSERT DELAYS BETWEEN TRANSMITTED BYTES.



NOTE: The rest of the code examples for this section are written for the BS2, using the BS2's *Baudmode* and *Timeout* values. Be sure to adjust the value for your BASIC Stamp model.

The SEROUT command can also be configured to pause between transmitted bytes. This is the purpose of the optional *Pace* argument. For example (9600 baud N8, inverted):

```
SEROUT 1, 16780, 1000, ["Slowly..."]
```

Here, the BASIC Stamp transmits "Slowly..." with a 1 second delay between each character. See Table 5.104 for units of the *Pace* argument. One good reason to use the *Pace* feature is to support devices that require more than one stop bit. Normally, the BASIC Stamp sends data as fast as it can (with a minimum of 1 stop bit between bytes). Since a stop bit is really just a resting state in the line (no data transmitted), using the *Pace* option will effectively add multiple stop bits. Since the requirement for 2 or more stop bits (on some devices) is really just a "minimum" requirement, the receiving side should receive this data correctly.

USING ASCII CODES.

Keep in mind that when we type something like "XYZ" into the SEROUT command, the BASIC Stamp actually uses the ASCII codes for each of those characters for its tasks. We could also typed: 88, 89, 90 in place of "XYZ" and the program would run the same way since 88 is the ASCII code for the "X" character, 89 is the ASCII code for the "Y" character, and so on.

ADDITIONAL CONVERSION FORMATTERS.

The decimal formatter is only one of a whole family of conversion formatters available with SERIN on all BS2 models. See Table 5.110 for a list of available conversion formatters. All of the conversion formatters work similar to the decimal formatter. The formatters translate the value into separate bytes of data until the entire number is translated or until the

## SEROUT – BASIC Stamp Command Reference

indicated number of digits (in the case of the fixed length formatters) is translated.

All BS2 models also have special formatters for outputting a string of characters, and repeated characters. See Table 5.111 for a list of these special formatters. All 2

| Conversion Formatter | Type of Number                                                              | Notes |
|----------------------|-----------------------------------------------------------------------------|-------|
| DEC{1..5}            | Decimal, optionally fixed to 1 – 5 digits                                   | 1     |
| SDEC{1..5}           | Signed decimal, optionally fixed to 1 – 5 digits                            | 1,2   |
| HEX{1..4}            | Hexadecimal, optionally fixed to 1 – 4 digits                               | 1,3   |
| SHEX{1..4}           | Signed hexadecimal, optionally fixed to 1 – 4 digits                        | 1,2   |
| IHEX{1..4}           | Indicated hexadecimal, optionally fixed to 1 – 4 digits (\$ prefix)         | 1     |
| ISHEX{1..4}          | Signed, indicated hexadecimal, optionally fixed to 1 – 4 digits (\$ prefix) | 1,2   |
| BIN{1..16}           | Binary, optionally fixed to 1 – 16 digits                                   | 1     |
| SBIN{1..16}          | Signed binary, optionally fixed to 1 – 16 digits                            | 1,2   |
| IBIN{1..16}          | Indicated binary, optionally fixed to 1 – 16 digits (% prefix)              | 1     |
| ISBIN{1..16}         | Signed, indicated binary, optionally fixed to 1 – 16 digits (% prefix)      | 1,2   |

**Table 5.110:** Conversion Formatters for all BS2 models.

- 1 Fixed-digit formatters like DEC4 will pad the number with leading 0s if necessary; ex: DEC4 65 sends 0065. If a number is larger than the specified number of digits, the leading digits will be dropped; ex: DEC4 56422 sends 6422.
- 2 Signed modifiers work under two's complement rules.
- 3 The HEX modifier can be used for BCD to Decimal Conversion. See "Hex to BCD Conversion" on page 97.

| Special Formatter | Action                                                                                                                                                                                                                |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ?                 | Displays "symbol = x' + carriage return; where x is a number. Default format is decimal, but may be combined with conversion formatters (ex: BIN ? x to display "x = binary_number").                                 |
| ASC ?             | Displays "symbol = 'x'" + carriage return; where x is an ASCII character.                                                                                                                                             |
| STR ByteArray {L} | Send character string from an array. The optional \L argument can be used to limit the output to L characters, otherwise, characters will be sent up to the first byte equal to 0 or the end of RAM space is reached. |
| REP Byte \L       | Send a string consisting of Byte repeated L times (ex: REP "X"\10 sends "XXXXXXXXXX").                                                                                                                                |

**Table 5.111:** Special Formatters for all BS2 models.

The string formatter is useful for transmitting a string of characters from a byte array variable. A string of characters is a set of characters that are arranged or accessed in a certain order. The characters "ABC" could be stored in a string with the "A" first, followed by the "B" and then followed by the "C." A byte array is a similar concept to a string; it contains data that is arranged in a certain order. Each of the elements in an array is the

THE STR (STRING) FORMATTER.

## 5: BASIC Stamp Command Reference – SEROUT

same size. The string "ABC" could be stored in a byte array containing three bytes (elements). See the “Defining Arrays” section in Chapter 4 for more information on arrays.

Here is an example that transmits five bytes (from a byte array) through I/O pin 1 at 9600 bps, N81/inverted:

```

All 2 serStr VAR Byte(5) ' create 5-byte array

Main:
 serStr(0) = "H" ' fill array
 serStr(1) = "E"
 serStr(2) = "L"
 serStr(3) = "L"
 serStr(4) = "O"
 SEROUT 1, 16468, [STR serStr\5] ' transmit
```

Note that we use the optional \L argument of STR. If we didn't specify this, the BASIC Stamp would try to keep sending characters until it found a byte equal to 0. Since we didn't specify a last byte of 0 in the array, we chose to tell it explicitly to only send 5 characters.

### USING PARITY AND HANDLING PARITY ERRORS.

Parity is a simple error-checking feature. When the SEROUT command's *Baudmode* is set for even parity it counts the number of 1s in the outgoing byte and uses the parity bit to make that number even. For instance, if it is sending the 7-bit value: %0011010, it sets the parity bit to 1 in order to make an even number of 1s (four).

The receiver also counts the data bits to calculate what the parity bit should be. If it matches the parity bit received, the serial receiver assumes that the data was received correctly. Of course, this is not necessarily true, since two incorrectly received bits could make parity seem correct when the data was wrong, or the parity bit itself could be bad when the rest of the data was OK. Parity errors are only detected on the receiver side. Generally, the receiver determines how to handle the error. In a more robust application, the receiver and transmitter might be set up such that the receiver can request a re-send of data that was received with a parity error.

### CONTROLLING DATA FLOW.

When you design an application that requires serial communication between BASIC Stamp modules, you have to work within these limitations:

## SEROUT – BASIC Stamp Command Reference

---

- When the BASIC Stamp is sending or receiving data, it can't execute other instructions.
- When the BASIC Stamp is executing other instructions, it can't send or receive data. *The BASIC Stamp does not have a serial buffer* as there is in PCs. At most serial rates, the BASIC Stamp cannot receive data via SERIN, process it, and execute another SERIN in time to catch the next chunk of data, unless there are significant pauses between data transmissions.

These limitations can sometimes be addressed by using flow control; the *Fpin* option for SERIN and SEROUT (at baud rates of up to the limitation shown in Table 5.94). Through *Fpin*, SERIN can tell a BASIC Stamp sender when it is ready to receive data and SEROUT (on the sender) will wait for permission to send. (For that matter, *Fpin* flow control follows the rules of other serial handshaking schemes, but most computers other than the BASIC Stamp cannot start and stop serial transmission on a byte-by-byte basis. That's why this discussion is limited to communication between BASIC Stamp modules.)

Here's an example using flow control on the BS2 (data through I/O pin 1, flow control through I/O pin 0, 9600 baud, N8, noninverted):

```
SerData VAR BYTE
SEROUT 1\0, 84, [SerData]
```

When SEROUT executes, I/O pin 1 (*Tpin*) is made an output, and I/O pin 0 (*Fpin*) is made an input, to wait for the "go" signal from the receiver. Here's the relationship of serial polarity to *Fpin* states.

|              | Ready to Receive ("Go") | Not Ready to Receive ("Stop") |
|--------------|-------------------------|-------------------------------|
| Inverted     | Fpin is High (1)        | Fpin is Low (0)               |
| Non-inverted | Fpin is Low (0)         | Fpin is High (1)              |

**Table 5.112:** Flow control pin states in relation to polarity (inverted or non-inverted) for all BS2 models.

See the demo program, below, for a flow control example using two BS2s. In the demo program example, without flow control, the sender would transmit the whole word "Hello!" in about 6 ms. The receiver would catch the first byte at most; by the time it got back from the first 1-second PAUSE, the rest of the data would be long gone. With flow control, communication is flawless since the sender waits for the receiver to catch up.

## 5: BASIC Stamp Command Reference – SEROUT

In Figure 5.39 below, I/O pin 0, *Fpin*, is pulled to ground through a 10k resistor. This is to ensure that the sender sees a stop signal (0 for inverted communications) when the receiver is being programmed.

### USING THE SERIAL TIME-OUT FEATURE.

In the flow control examples above, the only way the SEROUT instruction will end (other than RESET or power-off) is if the receiver allows it to send the entire *OutputData* list. If *Fpin* permission never occurs, the program is stuck. However, you can tell the BASIC Stamp to abort SEROUT if it doesn't receive *Fpin* permission within a specified time period. For instance, to transmit a decimal number through pin 1 at 9600 baud, 8N, inverted and abort SEROUT after 2 seconds (2000 ms) if no *Fpin* permission arrives on I/O pin 0:

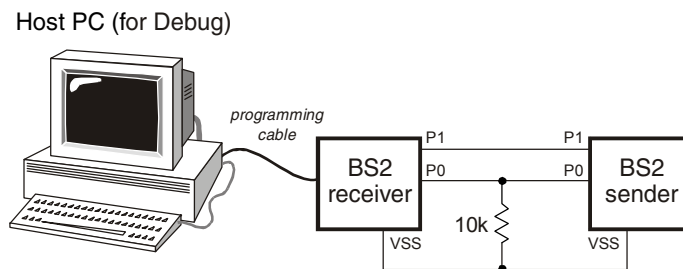


```
SEROUT 1\0, 16468, 2000, No_Permission, [DEC 150]
STOP
```

```
No_Permission:
 DEBUG "Timeout error", CR
```

If no *Fpin* permission arrives within 2 seconds, the program aborts SEROUT and continues at the label *No\_Permission*.

**Figure 5.39:** Flow-Control Example Circuit.



### USING OPEN BAUDMODES FOR NETWORKING BASIC STAMPS.

The SEROUT command supports open-drain and open-source output, which makes it possible to network multiple BASIC Stamp modules on a single pair of wires. These "open baudmodes" only actively drive the *Tpin* in one state (for the other state, they simply disconnect the pin; setting it to an input mode). If two BASIC Stamp modules in a network had their SEROUT lines connected together (while a third device listened on that line) and the BASIC Stamp modules were using always-driven baudmodes, they could simultaneously output two opposite states (i.e.: +5 volts and ground). This would create a short circuit. The heavy current flow would likely damage the I/O pins or the BASIC Stamp modules.

## SEROUT – BASIC Stamp Command Reference

themselves. Since the open baudmodes only drive in one state and float in the other, there's no chance of this kind of short.

The polarity selected for SEROUT determines which state is driven and which is open as in Table 5.113.

|              | State (0) | State (1) | Resistor Pulled to |
|--------------|-----------|-----------|--------------------|
| Inverted     | Open      | Driven    | Gnd (Vss)          |
| Non-inverted | Driven    | Open      | +5V (Vdd)          |

**Table 5.113:** Open Baudmode States for all BS2 models.

Since open baudmodes only drive to one state, they need a resistor to pull the networked line into the other state, as shown in Table 5.113 and in Figure 5.40 and Figure 5.41.

Open baudmodes allow the BASIC Stamp to share a line, but it is up to your program to resolve other networking issues such as who talks when and how to detect, prevent and fix data errors.

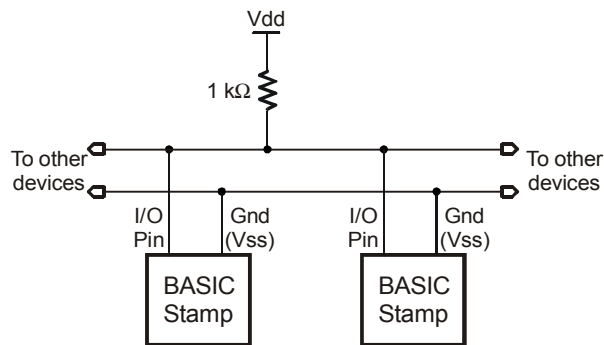
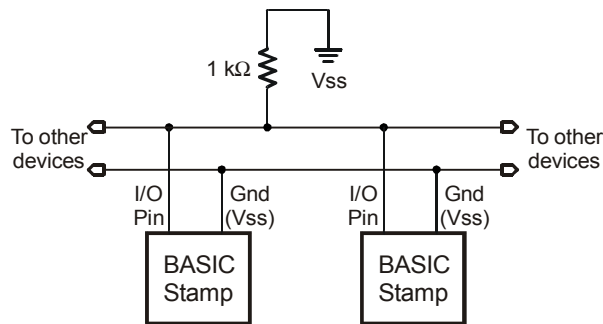


**Figure 5.40:** SEROUT Open-Drain Circuit. This circuit is for use with the Open, Non-inverted baudmode.



## 5: BASIC Stamp Command Reference – SEROUT

**Figure 5.41:** SEROUT Open-Source Circuit. This circuit is for use with the Open, Inverted baudmode.



### SEROUT TROUBLESHOOTING.

Serial communication, because of its complexity, can be very difficult to work with at times. Please follow these guidelines (and those in the "SERIN Troubleshooting" section of the SERIN command description) when developing a project using the SERIN and SEROUT commands:

1. Always build your project in steps.
  - a. Start with small, manageable pieces of code, that deals with serial communication) and test them, one at a time.
  - b. Add more and more small pieces, testing them each time, as you go.
  - c. Never write a large portion of code that works with serial communication without testing its smallest workable pieces first.
2. Pay attention to timing.
  - a. Be very careful to calculate and overestimate the amount of time operations should take within the BASIC Stamp. Misunderstanding the timing constraints is the source of most problems with code that communicate serially.
  - b. If the serial communication in your project is bi-directional, the above statement is even more critical.
3. Pay attention to wiring.
  - a. Take extra time to study and verify serial communication wiring diagrams. A mistake in wiring can cause strange problems in communication, or no communication at all. Make sure to connect the ground pins (Vss) between the devices that are communicating serially.
4. Verify port setting on the PC and in the SERIN/SEROUT commands.

## ***SEROUT – BASIC Stamp Command Reference***

---

- a. Unmatched settings on the sender and receiver side will cause garbled data transfers or no data transfers. If the data you receive is unreadable, it is most likely a baud rate setting error.
5. If data transmitted to the Stamp Editor's Debug Terminal is garbled, verify the output format.
  - a. A common mistake is to send data with SEROUT in ASCII format. For example, SEROUT 16, 84, [ 0 ] instead of SEROUT 16, 84, [ DEC 0 ]. The first example will send a byte equal to 0 to the PC, resulting in the Debug Terminal clearing the screen (since 0 is the control character for a clear-screen action).

### **Demo Program (SEROUT.bs1)**



```
' SEROUT.bs1
' This program transmits the string "ABCD" followed by a number and a
' carriage-return at 2400 baud, inverted, N81 format.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL SOut = 1
SYMBOL Baud = N2400

SYMBOL value = W1

Setup:
 value = 1

Main:
 SEROUT SOut, Baud, ("ABCD", #value)
 value = value + 1
 PAUSE 250
 GOTO Main
END
```

### **Demo Program (SERIN\_SEROUT1.bs2)**



```
' SERIN_SEROUT1.bs2
' Using two BS2-IC's, connect the circuit shown in the SERIN command
' description and run this program on the BASIC Stamp designated as the
' Sender. This program demonstrates the use of Flow Control (FPin).
' Without flow control, the sender would transmit the whole word "Hello!"
' in about 1.5 ms. The receiver would catch the first byte at most; by the
' time it got back from the first 1-second PAUSE, the rest of the data
```

NOTE: This example program was written for BS2's but it can be used with the BS2e, BS2sx, BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

## 5: BASIC Stamp Command Reference – SEROUT

```
' would be long gone. With flow control, communication is flawless since
' the sender waits for the receiver to catch up.

' {$STAMP BS2}
' {$PBASIC 2.5}

SO PIN 1 ' serial output
FC PIN 0 ' flow control pin

#SELECT $STAMP
#CASE BS2, BS2E, BS2PE
 T1200 CON 813
 T2400 CON 396
 T9600 CON 84
 T19K2 CON 32
 T38K4 CON 6
#CASE BS2SX, BS2P
 T1200 CON 2063
 T2400 CON 1021
 T9600 CON 240
 T19K2 CON 110
 T38K4 CON 45
#CASE BS2PX
 T1200 CON 3313
 T2400 CON 1646
 T9600 CON 396
 T19K2 CON 188
 T38K4 CON 84
#ENDSELECT

Inverted CON $4000
Open CON $8000
Baud CON T38K4 + Inverted

Main:
DO
 SEROUT SO\FC, Baud, ["Hello!", CR] ' send the greeting
 PAUSE 2500 ' wait 2.5 seconds
LOOP ' repeat forever
END
```



### Demo Program (SERIN\_SEROUT2.bs2)

NOTE: This example program was written for BS2's but it can be used with the BS2e, BS2sx, BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

```
' SERIN_SEROUT2.bs2
' Using two BS2-IC's, connect the circuit shown in the SERIN command
' description and run this program on the BASIC Stamp designated as the
' Receiver. This program demonstrates the use of Flow Control (FPin).
' Without flow control, the sender would transmit the whole word "Hello!"
' in about 1.5 ms. The receiver would catch the first byte at most; by the
' time it got back from the first 1-second PAUSE, the rest of the data
' would be long gone. With flow control, communication is flawless since
```

## ***SEROUT – BASIC Stamp Command Reference***

---

```
' the sender waits for the receiver to catch up.

' {$STAMP BS2}
' {$PBASIC 2.5}

SI PIN 0 ' serial input
FC PIN 1 ' flow control pin

#SELECT $STAMP
#CASE BS2, BS2E, BS2PE
 T1200 CON 813
 T2400 CON 396
 T9600 CON 84
 T19K2 CON 32
 T38K4 CON 6
#CASE BS2SX, BS2P
 T1200 CON 2063
 T2400 CON 1021
 T9600 CON 240
 T19K2 CON 110
 T38K4 CON 45
#CASE BS2PX
 T1200 CON 3313
 T2400 CON 1646
 T9600 CON 396
 T19K2 CON 188
 T38K4 CON 84
#ENDSELECT

Inverted CON $4000
Open CON $8000
Baud CON T38K4 + Inverted

letter VAR Byte

Main:
DO
 SERIN SI\FC, Baud, [letter] ' recieve one byte
 DEBUG letter ' display on screen
 PAUSE 1000 ' wait one second
LOOP ' repeat forever
END
```

## 5: BASIC Stamp Command Reference – SHIFTIN

### SHIFTIN

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**SHIFTIN** *Dpin, Cpin, Mode, [ Variable { \Bits } {, Variable { \Bits }...} ]*

### Function

Shift data in from a synchronous serial device.

- **Dpin** is a variable/constant/expression (0 – 15) that specifies the I/O pin that will be connected to the synchronous serial device's data output. This pin will be set to input mode.
- **Cpin** is a variable/constant/expression (0 – 15) that specifies the I/O pin that will be connected to the synchronous serial device's clock input. This pin will be set to output mode.
- **Mode** is a variable/constant/expression (0 – 3), or one of four predefined symbols, that tells SHIFTIN the order in which data bits are to be arranged and the relationship of clock pulses to valid data. See Table 5.115 for value and symbol definitions.
- **Variable** is a variable in which incoming data bits will be stored.
- **Bits** is an optional variable/constant/expression (1 – 16) specifying how many bits are to be input by SHIFTIN. If no *Bits* argument is given, SHIFTIN defaults to 8 bits.

### Quick Facts

|                                                        | BS2/BS2e                | BS2sx/BS2p               | BS2pe                   | BS2px                      |
|--------------------------------------------------------|-------------------------|--------------------------|-------------------------|----------------------------|
| <b>Timing of <math>T_h</math> and <math>t_l</math></b> | 14 $\mu$ s / 46 $\mu$ s | 5.6 $\mu$ s / 18 $\mu$ s | 14 $\mu$ s / 46 $\mu$ s | 3.6 $\mu$ s / 11.8 $\mu$ s |
| <b>Transmission Rate</b>                               | ~16 kbits/sec.          | ~42 kbits/sec.           | ~16 kbits/sec.          | ~ 65 kbits/sec.            |
| <b>Related Command</b>                                 | SHIFTOUT                |                          |                         |                            |

Table 5.114: SHIFTIN Quick Facts.

### Explanation

SHIFTIN and SHIFTOUT provide an easy method of acquiring data from synchronous serial devices. Synchronous serial differs from asynchronous serial (like SERIN and SEROUT) in that the timing of data bits (on a data line) is specified in relationship to clock pulses (on a clock line). Data bits may be valid after the rising or falling edge of the clock line. This kind of serial protocol is called Synchronous Peripheral Interface (SPI) and is commonly used by controller peripherals like ADCs, DACs, clocks, memory devices, etc.

## SHIFTIN – BASIC Stamp Command Reference

At their heart, synchronous-serial devices are essentially shift-registers; trains of flip-flops that pass data bits along in a bucket brigade fashion to a single data output pin. Another bit is output each time the appropriate edge (rising or falling, depending on the device) appears on the clock line.

The SHIFTIN instruction first causes the clock pin to output low and the data pin to switch to input mode. Then, SHIFTIN either reads the data pin and generates a clock pulse (PRE mode) or generates a clock pulse then reads the data pin (POST mode). SHIFTIN continues to generate clock pulses and read the data pin for as many data bits as are required.

SHIFTIN OPERATION.

Making SHIFTIN work with a particular device is a matter of matching the mode and number of bits to that device's protocol. Most manufacturers use a timing diagram to illustrate the relationship of clock and data. Items to look for include: 1) which bit of the data arrives first; most significant bit (MSB) or least significant bit (LSB) and 2) is the first data bit ready before the first clock pulse (PRE) or after the first clock pulse (POST). Table 5.115 shows the values and symbols available for *Mode*, and Figure 5.42 shows SHIFTIN's timing.

| Symbol | Value | Meaning                                           |
|--------|-------|---------------------------------------------------|
| MSBP   | 0     | Data is msb-first; sample bits before clock pulse |
| LSBP   | 1     | Data is lsb-first; sample bits before clock pulse |
| MSBP   | 2     | Data is msb-first; sample bits after clock pulse  |
| LSBP   | 3     | Data is lsb-first; sample bits after clock pulse  |

**Table 5.115:** SHIFTIN Mode Values and Symbols.

(Msb is most-significant bit; the highest or leftmost bit of a nibble, byte, or word. Lsb is the least-significant bit; the lowest or rightmost bit of a nibble, byte, or word.)

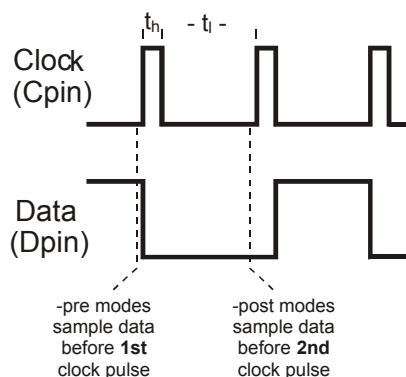


**Figure 5.42:** SHIFTIN Timing Diagram. Refer to the SHIFTIN Quick Facts table for timing information on  $t_h$  (t high) and  $t_l$  (t low).

## 5: BASIC Stamp Command Reference – SHIFTIN

---

### A SIMPLE SHIFTIN EXAMPLE.

Here is a simple example:

```
result VAR Byte

SHIFTIN 0, 1, MSBPRES, [result]
```

Here, the SHIFTIN command will read I/O pin 0 (*Dpin*) and will generate a clock signal on I/O 1 (*Cpin*). The data that arrives on *Dpin* depends on the device connected to it. Let's say, for example, that a shift register is connected and has a value of \$AF (10101111) waiting to be sent. Additionally, let's assume that the shift register sends out the most significant bit first, and the first bit is on *Dpin* before the first clock pulse (MSBPRES). The SHIFTIN command above will generate eight clock pulses and sample the data pin (*Dpin*) eight times. Afterward, the *result* variable will contain the value \$AF.

### CONTROLLING THE NUMBER OF BITS RECEIVED.

By default, SHIFTIN acquires eight bits, but you can set it to shift any number of bits from 1 to 16 with the *Bits* argument. For example:

```
result VAR Byte

SHIFTIN 0, 1, MSBPRES, [result\4]
```

Will only input the first 4 bits. In the example discussed above, the *result* variable will be left with %1010.

Some devices return more than 16 bits. For example, most 8-bit shift registers can be daisy-chained together to form any multiple of 8 bits; 16, 24, 32, 40... To solve this, you can use a single SHIFTIN instruction with multiple variables. Each variable can be assigned a particular number of bits with the *Bits* argument. As in:

```
resultLo VAR Word
resultHi VAR Nib

SHIFTIN 0, 1, MSBPRES, [resultHi\4, resultLo\16]
```

The above code will first shift in four bits into *resultHi* and then 16 bits into *resultLo*. The two variables together make up a 20 bit value.

## SHIFTIN – BASIC Stamp Command Reference

---

### Demo Program (SHIFTIN.bs2)

```
' SHIFTIN.bs2
' This program uses the SHIFTIN instruction to interface with the ADC0831
' 8-bit analog-to-digital converter from National Semiconductor.

' {$STAMP BS2}
' {$PBASIC 2.5}

CS PIN 0 ' chip select
AData PIN 1 ' data pin
Clk PIN 2 ' clock pin

adcRes VAR Byte ' ADC result

Setup:
 HIGH CS ' deselect ADC

' In the loop below, just three lines of code are required to read the
' ADC0831. The SHIFTIN command does most of the work. The mode argument in
' the SHIFTIN command specifies MSB or LSB-first and whether to sample data
' before or after the clock. In this case, we chose MSB-first, post-clock.
' The ADC0831 precedes its data output with a dummy bit, which we take care
' of by specifying 9 bits of data instead of 8.

Main:
 DO
 LOW CS ' activate the ADC0831
 SHIFTIN AData, Clk, MSBPOST, [adcRes\9] ' shift in the data
 HIGH CS ' deactivate ADC0831
 DEBUG ? adcRes ' show conversion result
 PAUSE 1000 ' wait one second
 LOOP ' repeat
END
```



NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.



## 5: BASIC Stamp Command Reference – SHIFTOUT

### SHIFTOUT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**SHIFTOUT** *Dpin, Cpin, Mode, [ OutputData {\Bits} { ,OutputData {\Bits}...} ]*

#### Function

Shift data out to a synchronous serial device.

- **Dpin** is a variable/constant/expression (0 – 15) that specifies the I/O pin that will be connected to the synchronous serial device's data input. This pin will be set to output mode.
- **Cpin** is a variable/constant/expression (0 – 15) that specifies the I/O pin that will be connected to the synchronous serial device's clock input. This pin will be set to output mode.
- **Mode** is a variable/constant/expression (0 – 1), or one of two predefined symbols, that tells SHIFTOUT the order in which data bits are to be arranged. See Table 5.117 for value and symbol definitions.
- **OutputData** is a variable/constant/expression containing the data to be sent.
- **Bits** is an optional variable/constant/expression (1 – 16) specifying how many bits are to be output by SHIFTOUT. When the *Bits* argument is given, the BASIC Stamp transmits the rightmost number of bits specified, regardless of the *Mode*. If no *Bits* argument is given, SHIFTOUT defaults to 8 bits.

#### Quick Facts

|                           | BS2, BS2e               | BS2sx, BS2p                | BS2pe                   | BS2px                      |
|---------------------------|-------------------------|----------------------------|-------------------------|----------------------------|
| Timing of $t_h$ and $t_p$ | 14 $\mu$ s / 46 $\mu$ s | 5.6 $\mu$ s / 18 $\mu$ s   | 14 $\mu$ s / 46 $\mu$ s | 3.6 $\mu$ s / 11.8 $\mu$ s |
| Timing of $t_a$ and $t_b$ | 15 $\mu$ s / 30 $\mu$ s | 6.3 $\mu$ s / 12.5 $\mu$ s | 15 $\mu$ s / 30 $\mu$ s | 4 $\mu$ s / 7.8 $\mu$ s    |
| Transmission Rate         | ~16 kbits/sec.          | ~42 kbits/sec.             | ~16 kbits/sec.          | ~65 kbits/sec.             |
| Related Command           | SHIFTOUT                |                            |                         |                            |

Table 5.116: SHIFTOUT Quick Facts.

#### Explanation

SHIFTOUT and SHIFTOIN provide an easy method of acquiring data from synchronous serial devices. Synchronous serial differs from asynchronous serial (like SERIN and SEROUT) in that the timing of data bits (on a data line) is specified in relationship to clock pulses (on a clock line). Data bits may be valid after the rising or falling edge of the clock line. This kind of

## SHIFTOUT – BASIC Stamp Command Reference

serial protocol is commonly called Synchronous Peripheral Interface (SPI) and is used by controller peripherals like ADCs, DACs, clocks, memory devices, etc.

At their heart, synchronous-serial devices are essentially shift-registers; trains of flip-flops that receive data bits in a bucket brigade fashion from a single data input pin. Another bit is input each time the appropriate edge (rising or falling, depending on the device) appears on the clock line.

The SHIFTOUT instruction first causes the clock pin to output low and the data pin to switch to output mode. Then, SHIFTOUT sets the data pin to the next bit state to be output and generates a clock pulse. SHIFTOUT continues to generate clock pulses and places the next data bit on the data pin for as many data bits as are required for transmission.

SHIFTOUT OPERATION.

Making SHIFTOUT work with a particular device is a matter of matching the mode and number of bits to that device's protocol. Most manufacturers use a timing diagram to illustrate the relationship of clock and data. One of the most important items to look for is which bit of the data should be transmitted first; most significant bit (MSB) or least significant bit (LSB). Table 5.117 shows the values and symbols available for *Mode* and Figure 5.43 shows SHIFTOUT's timing.

| Symbol   | Value | Meaning                       |
|----------|-------|-------------------------------|
| LSBFIRST | 0     | Data is shifted out lsb-first |
| MSBFIRST | 1     | Data is shifted out msb-first |

**Table 5.117:** SHIFTOUT Mode Values and Symbols.

(Msb is most-significant bit; the highest or leftmost bit of a nibble, byte, or word. Lsb is the least-significant bit; the lowest or rightmost bit of a nibble, byte, or word.)

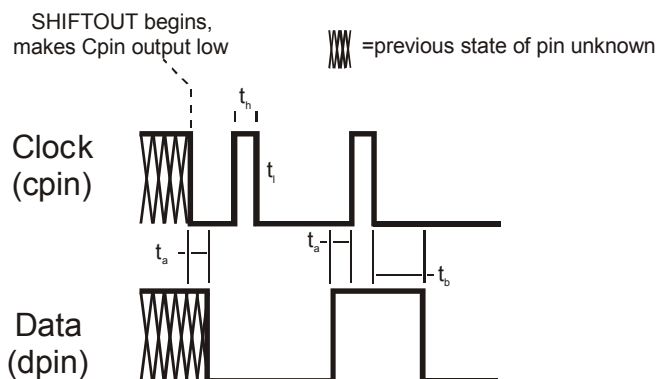


**Figure 5.43:** SHIFTOUT Timing Diagram. Refer to the SHIFTOUT Quick Facts table for timing information on  $t_n$ ,  $t_i$ ,  $t_a$  and  $t_b$ .

## 5: BASIC Stamp Command Reference – SHIFTOUT

---

### A SIMPLE SHIFTOUT EXAMPLE.

Here is a simple example:

```
SHIFTOUT 0, 1, MSBFIRST, [250]
```

Here, the SHIFTOUT command will write to I/O pin 0 (*Dpin*) and will generate a clock signal on I/O pin 1 (*Cpin*). The SHIFTOUT command will generate eight clock pulses while writing each bit (of the 8-bit value 250) onto the data pin (*Dpin*). In this case, it will start with the most significant bit first as indicated by the *Mode* value of MSBFIRST.

### CONTROLLING THE NUMBER OF BITS TRANSMITTED.

By default, SHIFTOUT transmits eight bits, but you can set it to shift any number of bits from 1 to 16 with the *Bits* argument. For example:

```
SHIFTOUT 0, 1, MSBFIRST, [250\4]
```

Will output only the lowest (rightmost) four bits (%1010 in this case). But what if you want to output the leftmost bits of a given value? By adding the right-shift operator (>>) to the code you can adjust the output as required:

```
SHIFTOUT 0, 1, MSBFIRST, [(250 >> 2)\6]
```

will output the upper six bits (%111110 in this case).

Some devices require more than 16 bits. To solve this, you can use a single SHIFTOUT command with multiple values. Each value can be assigned a particular number of bits with the *Bits* argument. As in:

```
SHIFTOUT 0, 1, MSBFIRST, [250\4, 1045\16]
```

The above code will first shift out four bits of the number 250 (%1010) and then 16 bits of the number 1045 (%0000010000010101). The two values together make up a 20 bit value.

### SHIFTOUT ACCEPTS VARIABLES AND EXPRESSIONS FOR *OUTPUTDATA* AND *BITS* ARGUMENTS.

In the examples above, specific numbers were entered as the data to transmit, but, of course, the SHIFTOUT command will accept variables and expressions for the *OutputData* and even for the *Bits* argument.

## SHIFTOUT – BASIC Stamp Command Reference

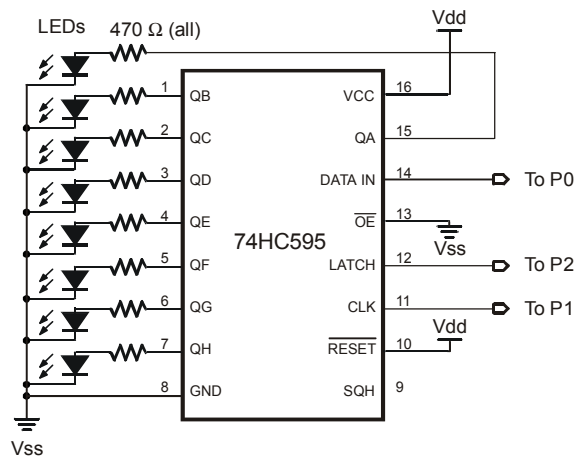


**Figure 5.44:** Circuit for Demo Program SHIFTOUT.bs2.

### Demo Program (SHIFTOUT.bs2)

```
' SHIFTOUT.bs2
' This program uses the SHIFTOUT command to interface to the 74HC595 shift
' register as an 8-bit output port. The '595 requires a minimum of three
' inputs: data, clock, and latch. See the figure in the SHIFTOUT command
' description in the manual for wiring information. SHIFTOUT automatically
' handles the data and clock, pulsing the clock to shift data bits into the
' '595. An extra step (pulsing the latch input) is required to move the
' shifted bits in parallel onto the '595's output pins. Note: this code
' does not control the output-enable or reset lines of the '595. This means
' that before the BASIC Stamp first sends, the '595's output latches are
' turned on and may contain random data. In critical applications, you
' should hold output-enable high (disabled) until the BASIC Stamp can take
' control.

' {$STAMP BS2}
' {$PBASIC 2.5}

Dpin PIN 0 ' data pin to 74HC595
Clk PIN 1 ' shift clock to 74HC595
Latch PIN 2 ' latch 74HC595 outputs

counter VAR Byte

Setup:
 LOW Latch ' initialize latch output

' This loop moves the 8-bit value 'counter' onto the output lines of the
' '595, pauses, then increments counter and repeats. The data is shifted
```

Alt 2

**NOTE:** This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## ***5: BASIC Stamp Command Reference – SHIFTOUT***

---

```
' msb first so that the msb appears on pin QH and the lsb on QA. Changing
' MSBFIRST to LSBFIRST causes the data to appear backwards on the outputs.
```

```
Main:
```

```
 DO
 SHIFTOUT Dpin, Clk, MSBFIRST, [counter] ' send the bits
 PULSOUT Latch, 1 ' transfer to outputs
 PAUSE 100 ' Wait 0.1 seconds
 counter = counter + 1 ' increment counter
 LOOP
END
```

## ***SHIFTOUT – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – SLEEP

### SLEEP

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



#### SLEEP Duration

#### Function

Put the BASIC Stamp into low-power mode for a specified time.

- **Duration** is a variable/constant/expression (1 – 65535) that specifies the duration of sleep. The unit of time for *Duration* is 1 second, though the BASIC Stamp rounds up to the nearest multiple of 2.3 seconds.

#### Quick Facts

|                           | BS1                                 | BS2   | BS2e   | BS2sx  | BS2p                  | BS2pe | BS2px  |
|---------------------------|-------------------------------------|-------|--------|--------|-----------------------|-------|--------|
| Current Draw during Run   | 1 mA                                | 3 mA  | 25 mA  | 60 mA  | 40 mA                 | 15 mA | 55 mA  |
| Current Draw during SLEEP | 25 μA                               | 50 μA | 200 μA | 500 μA | 350 μA                | 36 μA | 450 μA |
| Related Commands          | END and NAP                         |       |        |        | END, NAP and POLLWAIT |       |        |
| Accuracy of SLEEP         | ±1% @ 75°F with stable power supply |       |        |        |                       |       |        |

#### Explanation

SLEEP allows the BASIC Stamp to turn itself off, then turn back on after a programmed period of time. The length of SLEEP can range from 2.3 seconds to slightly over 18 hours. Power consumption is reduced to the amount described in Table 5.118, assuming no loads are being driven. The resolution of the SLEEP instruction is 2.304 seconds. SLEEP rounds the specified number of seconds up to the nearest multiple of 2.304. For example, SLEEP 1 causes 2.304 seconds of sleep, while SLEEP 10 causes 11.52 seconds (5 x 2.304) of sleep.

Pins retain their previous I/O directions during SLEEP. However, outputs are interrupted every 2.3 seconds during SLEEP due to the way the chip keeps time. The alarm clock that wakes the BASIC Stamp up is called the watchdog timer. The watchdog is a resistor/capacitor oscillator built into the interpreter chip. During SLEEP, the chip periodically wakes up and adjusts a counter to determine how long it has been asleep. If it isn't time to wake up, the chip "hits the snooze bar" and goes back to sleep.



NOTE: Expressions are not allowed as arguments on the BS1.

Table 5.118: SLEEP Quick Facts.

NOTE: Current measurements are based on 5-volt power, no extra loads and 75° F ambient temperature.

## SLEEP – BASIC Stamp Command Reference

To ensure accuracy of SLEEP intervals, the BASIC Stamp periodically compares the watchdog timer to the more-accurate resonator time base. It calculates a correction factor that it uses during SLEEP. As a result, longer SLEEP intervals are accurate to approximately  $\pm 1$  percent.

If your application is driving loads (sourcing or sinking current through output-high or output-low pins) during SLEEP, current will be interrupted for about 18 ms (60  $\mu$ s on the BS2pe) when the BASIC Stamp wakes up every 2.3 seconds. The reason is that the watchdog-timer reset that awakens the BASIC Stamp also causes all of the pins to switch to input mode for approximately 18 ms. When the interpreter firmware regains control of the processor, it restores the I/O directions dictated by your program.

If you plan to use END, NAP, POLLWAIT or SLEEP in your programs, make sure that your loads can tolerate these periodic power outages. The simplest solution is often to connect resistors high or low (to +5V or ground) as appropriate to ensure a continuing supply of current during the reset glitch. The demo program demonstrates the effects of this glitch.

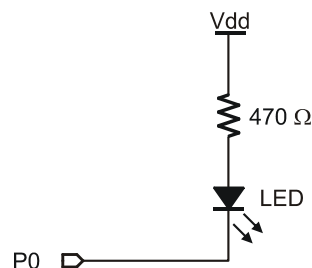


**Figure 5.45:** SLEEP Example LED Circuit.

### Demo Program (SLEEP.bs2)

```
' SLEEP.bs2
' This program lights an LED and then goes to sleep. Connect an LED to pin
' 0 as shown in the description of SLEEP in the manual and run the program.
' The LED will turn on, then the BASIC Stamp will go to sleep. During
' sleep, the LED will remain on, but will blink at intervals of
' approximately 2.3 seconds due to the watchdog timeout and reset.

' {$STAMP BS2}

Setup:
 LOW 0 ' turn LED on
```



**NOTE:** This example program is written for the BS2, but it also can be used with the BS1 and all other BS2 models by changing the \$STAMP directive accordingly.



## ***5: BASIC Stamp Command Reference – SLEEP***

---

```
Snooze:
 SLEEP 10 ' sleep for 10 seconds
 GOTO Snooze
END
```

## ***SLEEP – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – SOUND

### SOUND

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|



**SOUND** *Pin*, ( *Note*, *Duration* { , *Note*, *Duration*... } )

(See FREQOUT)

### Function

Generate square-wave tones for a specified period.

- **Pin** is a variable/constant (0 – 7) that specifies the I/O pin to use. This pin will be set to output mode.
- **Note** is a variable/constant (0 – 255) specifying the type and frequency of the tone. 1 – 127 are ascending tones and 128 – 255 are ascending white noises ranging from buzzing (128) to hissing (255).
- **Duration** is a variable/constant (1 - 255) specifying the amount of time to generate the tone(s). The unit of time for *Duration* is 12 ms.

### Quick Facts

**Table 5.119:** SOUND Quick Facts.

|                          | BS1                  |
|--------------------------|----------------------|
| Units in <i>Duration</i> | 12 ms                |
| Available Sounds         | 256                  |
| Frequency Range          | 94.8 Hz to 10,550 Hz |

### Explanation

SOUND generates one of 256 square-wave frequencies on an I/O pin. The output pin should be connected as shown in Figure 5.46.

The tones produced by SOUND can vary in frequency from 94.8 Hz (1) to 10,550 Hz (127). If you need to determine the frequency corresponding to a given note value, or need to find the note value that will give you best approximation for a given frequency, use the equations below.

$$\text{Note} = 127 - ((1/\text{Frequency}) - 0.000095) / 0.000083$$

--and--

$$\text{Frequency} = (1 / (0.000095 + ((127 - \text{Note}) * 0.000083))$$

In the above equations, Frequency is in Hertz (Hz).

## SOUND – BASIC Stamp Command Reference

### Driving an Audio Amplifier

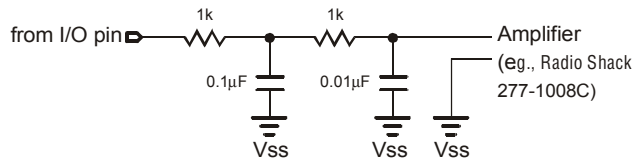
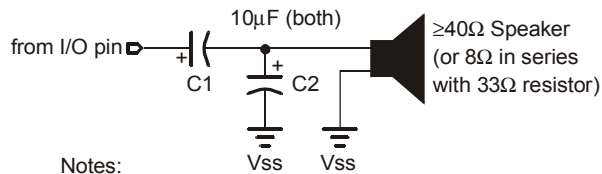


**Figure 5.46:** Example RC filter circuits for driving an audio amplifier (*top*) or a speaker (*bottom*).

### Driving a Speaker



Notes:  
C1 may be omitted for piezo speakers  
C2 is optional, but reduces high-frequency noise

### Demo Program (SOUND.bs1)

```
' SOUND.bs1
' This program generates a constant tone 25 followed by an ascending tones.
' Both the tones have the same duration.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL tone = B2

Main:
 FOR tone = 0 TO 255
 SOUND 0, (25, 10, tone, 10)
 NEXT
 END
```

## 5: BASIC Stamp Command Reference – STOP

### STOP

All 2

### STOP

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|

### Function

Stop program execution.

### Quick Facts

Table 5.120: STOP Quick Facts.

|                 | All BS2 Models |
|-----------------|----------------|
| Related Command | END            |

### Explanation

STOP prevents the BASIC Stamp from executing any further instructions until it is reset. The following actions will reset the BASIC Stamp:

1. Pressing and releasing the RESET button on the development board.
2. Driving the RES pin low then letting it float (high).
3. Downloading a new program
4. Disconnecting then reconnecting the power.

STOP differs from END in two respects:

1. Stop does not put the BASIC Stamp into low-power mode. The BASIC Stamp draws just as much current as if it were actively running program instructions.
2. The output glitch that occurs after a program has "ended" does not occur after a program has "stopped."

All 2

### Demo Program (STOP.bs2)

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

```
' STOP.bs2
' This program is similar to SLEEP.BS2 except that the LED will not blink
' since the BASIC Stamp does not go into low power mode. Use the circuit
' shown in the description of the SLEEP command for this example.

' {$STAMP BS2}
' {$PBASIC 2.5}

Main:
 LOW 0 ' turn LED on
 STOP ' stop program
```

## ***STOP – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – STORE

### STORE

|     |     |      |       |      |       |       |
|-----|-----|------|-------|------|-------|-------|
| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|



#### STORE ProgramSlot

#### Function

Designate a program slot for the READ/WRITE instructions to operate upon.

- **ProgramSlot** is a variable/constant/expression (0 – 7 on BS2p and BS2px, 0-15 on BS2pe) that specifies the program slot to use for READ and WRITE instructions.

#### Quick Facts

Table 5.121: STORE Quick Facts.

|                    | BS2p and BS2px | BS2pe |
|--------------------|----------------|-------|
| Program Slot Range | 0 —7           | 0 —15 |
| Related Commands   | READ and WRITE |       |

#### Explanation

STORE tells the BS2p, BS2pe, and BS2px which program slot to use when a READ or WRITE instruction is executed. The STORE command only affects the READ and WRITE instructions.

The STORE command allows a program to access all EEPROM locations that exist on the BS2p, BS2pe, and BS2px regardless of which program is running. The READ and WRITE commands can only access locations 0 to 2047 within a single program slot. The STORE command switches the program slot that the READ and WRITE commands operate on.

The default program slot that the READ and WRITE instructions operate on is that of the currently running program. The STORE command can be used to temporarily change this, to any program slot. The change will remain in effect until another STORE command is issued, or until another program slot is executed.

## STORE – BASIC Stamp Command Reference

### Demo Program (STORE0.bsp)

```
' STORE0.bsp
' This program demonstrates the STORE command and how it affects the READ
' and WRITE commands. This program "STORE0.BSP" is intended to be down-
' loaded into program slot 0. It is meant to work with STORE1.BSP and
' STORE2.BSP. Each program is very similar (they display the current
' Program Slot and READ/WRITE Slot numbers and the values contained in the
' first five EEPROM locations. Each program slot will have different data
' due to different DATA commands in each of the programs downloaded.

' {$STAMP BS2p, STORE1.BSP, STORE2.BSP}
' {$PBASIC 2.5}

#IF ($STAMP < BS2P) #THEN
 #ERROR "This program requires BS2p, BS2pe, or BS2px."
#ENDIF

idx VAR Word ' index
value VAR Byte
LocalData DATA @0, 1, 2, 3, 4, 5

Main:
 GOSUB Show_Slot_Info ' show slot info/data
 PAUSE 2000
 STORE 1 ' point READ/WRITE to Slot 1
 GOSUB Show_Slot_Info
 PAUSE 2000
 RUN 1 ' run program in Slot 1
 END

Show_Slot_Info:
 GET 127, value
 DEBUG CR, "Pgm Slot: ", DEC value.NIB0,
 CR, "R/W Slot: ", DEC value.NIB1,
 CR, CR

 FOR idx = 0 TO 4
 READ idx, value
 DEBUG "Location: ", DEC idx, TAB,
 "Value: ", DEC3 value, CR
 NEXT
 RETURN
```



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.



## 5: BASIC Stamp Command Reference – STORE



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the \$STAMP directive accordingly.

### Demo Program (STORE1.bsp)

```
' STORE1.bsp
' {$STAMP BS2p}
' {$PBASIC 2.5}

idx VAR Word ' index
value VAR Byte

LocalData DATA @0, 6, 7, 8, 9, 10

Main:
 GOSUB Show_Slot_Info ' show slot info/data
 PAUSE 2000
 STORE 0 ' point READ/WRITE to Slot 0
 GOSUB Show_Slot_Info
 PAUSE 2000
 RUN 2 ' run program in Slot 2
 END

Show_Slot_Info:
 GET 127, value
 DEBUG CR, "Pgm Slot: ", DEC value.NIB0,
 CR, "R/W Slot: ", DEC value.NIB1,
 CR, CR

 FOR idx = 0 TO 4
 READ idx, value
 DEBUG "Location: ", DEC idx, TAB,
 "Value: ", DEC3 value, CR
 NEXT
 RETURN
```



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px by changing the \$STAMP directive accordingly.

### Demo Program (STORE2.bsp)

```
' STORE2.bsp
' {$STAMP BS2p}
' {$PBASIC 2.5}

idx VAR Word ' index
value VAR Byte

LocalData DATA @0, 11, 12, 13, 14, 15

Main:
 GOSUB Show_Slot_Info ' show slot info/data
 PAUSE 2000
 STORE 0 ' point READ/WRITE to Slot 0
```

## STORE – BASIC Stamp Command Reference

```
GOSUB Show_Slot_Info
END

Show_Slot_Info:
 GET 127, value
 DEBUG CR, "Pgm Slot: ", DEC value.NIB0,
 CR, "R/W Slot: ", DEC value.NIB1,
 CR, CR

 FOR idx = 0 TO 4
 READ idx, value
 DEBUG "Location: ", DEC idx, TAB,
 "Value: ", DEC3 value, CR
 NEXT
RETURN
```

The next Demo program, STOREALL.bsp, is not related to the previous three programs. STOREALL.bsp demonstrates the use of the STORE command to treat contiguous program slots as one block of memory (14 kBytes on the BS2p and BS2px, 30 kBytes on the BS2pe). This illustrates one of the most powerful uses of the STORE command.

### Demo Program (STOREALL.bsp)

```
' STOREALL.bsp
' This program demonstrates the STORE command and how it can be used to
' "flatten" the EEPROM space for applications requiring a lot of storage.
' This program writes to EEPROM locations within program slots 1 through 7
' on the BS2p and BS2px, and 1 through 15 on the BS2pe, thus, has access to
' 14- or 30-kBytes of space.

' {$STAMP BS2p}
' {$PBASIC 2.5}

#SELECT $STAMP
#CASE BS2, BS2E, BS2SX
 #ERROR "This program requires BS2p, BS2pe, or BS2px."
#CASE BS2P, BS2PX
 HiSlot CON 7
#CASE BS2PE
 HiSlot CON 15
#ENDSELECT

LoSlot CON 1 ' first slot for "flat" EE
MemSize CON HiSlot - LoSlot + 1 * 2048

eeAddr VAR Word ' address pointer
value VAR Word ' cell value
slot VAR Byte ' current R/W slot
```



NOTE: This example program can be used with the BS2p, BS2pe, and BS2px. This program uses conditional compilation techniques; see Chapter 3 for more information.

## 5: BASIC Stamp Command Reference – STORE

```
Main:
 DEBUG "Flat Memory", CR,
 "-----", CR,
 "First Slot.... ", DEC LoSlot, CR,
 "Last Slot..... ", DEC HiSlot, CR,
 "Flat EE Size... ", DEC MemSize, CR, CR

 PAUSE 2000
 DEBUG "Writing to flat Memory...", CR
 PAUSE 1000
 FOR eeAddr = 0 TO (MemSize - 1) STEP 128 ' step through "flat" EE
 value = eeAddr * 2 ' generate value
 GOSUB Write_Word ' write it
 GET 127, slot ' get R/W slot
 DEBUG "--> Location: ", DEC5 eeAddr, " ", ' show "flat" address
 "Value: ", DEC5 value, " ", ' show value
 "(", DEC slot.NIB1, ")", CR ' show slot
 NEXT
 DEBUG CR

 DEBUG "Reading from flat Memory...", CR
 PAUSE 1000
 FOR eeAddr = 0 TO (MemSize - 1) STEP 128 ' read value from EE
 GOSUB Read_Word ' get W/R slot
 GET 127, slot
 DEBUG "<-- Location: ", DEC5 eeAddr, " ", '
 "Value: ", DEC5 value, " ", '
 "(", DEC slot.NIB1, ")", " '
 IF (value <> (2 * eeAddr)) THEN ' verify location
 DEBUG "- Error"
 ENDIF
 DEBUG CR
 NEXT
 END

Write_Word:
' NOTE: only use even-byte eeAddr with this routine
STORE (eeAddr >> 11) + LoSlot ' set slot
WRITE eeAddr, Word value ' write value
RETURN

Read_Word:
' NOTE: only use even-byte eeAddr with this routine
STORE (eeAddr >> 11) + LoSlot ' set slot
READ eeAddr, Word value ' read value
RETURN
```

## ***STORE – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – TOGGLE

### TOGGLE

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



#### TOGGLE Pin

### Function

Invert the state of an output pin.

- **Pin** is a variable/constant/expression (0 – 15) that specifies which I/O pin to switch logic state. This pin will be placed into output mode.

### Quick Facts

|                   | BS1          | All BS2 Models |
|-------------------|--------------|----------------|
| Affected Register | PINS         | OUTS           |
| Related Commands  | HIGH and LOW |                |

**Table 5.122:** TOGGLE Quick Facts.

### Explanation

TOGGLE sets a pin to output mode and inverts the output state of the pin, changing 0 to 1 and 1 to 0.

In some situations TOGGLE may appear to have no effect on a pin's state. For example, suppose pin 2 is in input mode and pulled to +5V by a 10k resistor. Then the following code executes:

```

1 DIR2 = 0 ' make P2 an input
 PIN2 = 0 ' make P2 output driver low
 DEBUG PIN2 ' show P2 state (1 due to pull-up)
 TOGGLE 2 ' toggle P2
 DEBUG PIN2 ' show P2 state (1 again)

```

- or -

```

All 2 OUT2 = 0 ' make P2 an input
 OUT2 = 0 ' make P2 output driver low
 DEBUG ? IN2 ' show P2 state (1 due to pull-up)
 TOGGLE 2 ' toggle P2
 DEBUG ? IN2 ' show P2 state (1 again)

```

The state of pin 2 doesn't change; it's high (due to the resistor) before TOGGLE, and it's high (due to the pin being output high) afterward. The point is that TOGGLE works on the OUTS register (PINS on the BS1), which may not match the pin's state when the pin is initially an input. To

## TOGGLE – BASIC Stamp Command Reference

guarantee that the state actually changes, regardless of the initial input or output mode, do this:

```
PIN2 = PIN2 ' make output driver match input
TOGGLE 2 ' then toggle
```

- or -

```
OUT2 = IN2 ' make output driver match input
TOGGLE 2 ' then toggle
```

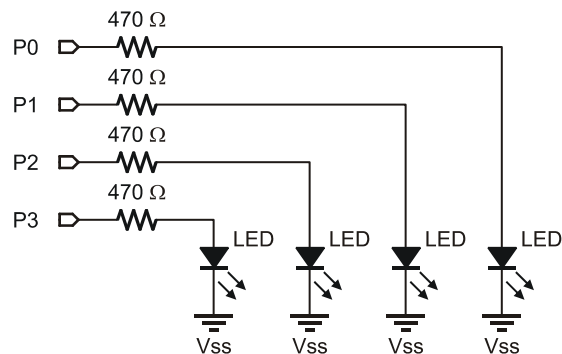


**Figure 5.47:** Example LED Circuit for TOGGLE Demo Programs.

### Demo Program (TOGGLE.bs1)



```
' TOGGLE.bs1
' Connect LEDs to pins 0 through 3 as shown in the TOGGLE command descrip-
' tion in the manual and run this program. The TOGGLE command will treat
' you to a light show. You may also run the demo without LEDs. The Debug
' window will show you the states of pins 0 through 3.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL thePin = B0 ' pin 0 - 3

Setup:
 DIRS = %1111 ' make LEDs output, low

Main:
 FOR thePin = 0 TO 3 ' loop through pins
 TOGGLE thePin ' toggle current pin
 DEBUG CLS, %PINS ' show on Debug
 PAUSE 100 ' short delay
 NEXT
 GOTO Main ' repeat forever
END
```

## 5: BASIC Stamp Command Reference – TOGGLE



NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

### Demo Program (TOGGLE.bs2)

```
' TOGGLE.bs2
' Connect LEDs to pins 0 through 3 as shown in the TOGGLE command descrip-
' tion in the manual and run this program. The TOGGLE command will treat
' you to a light show. You may also run the demo without LEDs. The Debug
' window will show you the states of pins 0 through 3.

' {$STAMP BS2}
' {$PBASIC 2.5}

thePin VAR Nib ' pin 0 - 3

Setup:
 DIRA = %1111 ' make LEDs output, low

Main:
 DO
 FOR thePin = 0 TO 3 ' loop through pins
 TOGGLE thePin ' toggle current pin
 DEBUG HOME, BIN4 OUTA ' show on Debug
 PAUSE 250 ' short delay
 NEXT
 LOOP ' repeat forever
END
```

## ***TOGGLE – BASIC Stamp Command Reference***

---



## 5: BASIC Stamp Command Reference – WRITE

### WRITE

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|

NOTE: Optional arguments require PBASIC 2.5.



**WRITE** *Location, Value*

**WRITE** *Location, { WORD } Value {, { WORD } Value... }*



NOTE: Expressions are not allowed as arguments on the BS1.

### Function

Write *Value* into *Location* in EEPROM.

- **Location** is a variable/constant/expression (0 – 255 on BS1, 0 – 2047 on all BS2 models) that specifies the EEPROM address to write to.
- **Value** is a variable/constant/expression (0 – 255, or 0 – 65535 if using the optional WORD modifier) to store in the EEPROM.

### Quick Facts

Table 5.123: WRITE Quick Facts.

|                                              | BS1             | BS2                                                                                                                                          | BS2e and BS2sx                                                | BS2p, BS2pe, BS2px                                             |
|----------------------------------------------|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------|----------------------------------------------------------------|
| <b>Range of EEPROM Locations</b>             | 0 to 255        | 0 to 2047                                                                                                                                    | 0 to 2047                                                     | 0 to 2047<br>(see notes below)                                 |
| <b>Maximum Number of Writes per Location</b> | 10 million      | 10 million                                                                                                                                   | 100,000                                                       | 100,000                                                        |
| <b>Special Notes</b>                         | n/a             | n/a                                                                                                                                          | WRITE only works with current program slot on BS2e and BS2sx. | WRITE works with any program slot as set by the STORE command. |
| <b>Related Commands</b>                      | READ and EEPROM | READ and DATA                                                                                                                                | READ and DATA                                                 | READ, DATA, and STORE                                          |
| <b>PBASIC 2.5 Syntax Options</b>             | n/a             | Multiple sequential variables may be written to the Scratch Pad RAM, and the optional WORD modifier may be specified to store 16-bit values. |                                                               |                                                                |

### Explanation

The EEPROM is used for both program storage (which builds downward from address 255 on BS1, 2047 on all BS2 models) and data storage (which builds upward from address 0). The WRITE instruction stores a value to any EEPROM address. Any location within the EEPROM can be written to (including your PBASIC program's locations) at run-time. This feature is mainly used to store long-term data to EEPROM; data stored in EEPROM is not lost when the power is removed.

## WRITE – BASIC Stamp Command Reference

---

The following WRITE command stores the value 245 at location 100:

```
WRITE 100, 245
```

--or--

```
WRITE 100, 245
```

A SIMPLE WRITE COMMAND.



The EEPROM is organized as a sequential set of byte-sized memory locations. The WRITE command normally only stores byte-sized values into EEPROM. This does not mean that you can't write word-sized values, however. A word consists of two bytes, called a low-byte and a high-byte. If you wanted to write a word-sized value, you'll need to use two WRITE commands and a word-sized value or variable. For example,

WRITING WORD VALUES VS. BYTE VALUES.

```
SYMBOL value = W0
SYMBOL valLo = B0
SYMBOL valHi = b1
```



```
value = 1125
```

```
WRITE 0, valLo ' write low byte
WRITE 1, valHi ' write high byte
```

- or -

```
value VAR Word
```

```
value = 1125
```

```
WRITE 0, value.LOWBYTE ' write low byte
WRITE 1, value.HIGHBYTE ' write high byte
```



NOTE: this method is required only if using PBASIC 2.0. See section below for PBASIC 2.5 method.

When this program runs, the two WRITE commands will store the low-byte and high-byte of the number 1125 into EEPROM.

On all BS2 models, with PBASIC 2.5 you can use a single WRITE command with the WORD modifier to write a 16-bit value. The low-byte of the value will be written to *Location*, the high byte will be written to *Location + 1*.

## 5: BASIC Stamp Command Reference – WRITE

```
All 2 ' {$PBASIC 2.5}
value VAR Word

value = 1125

WRITE 0, Word value ' write two bytes
```

When using PBASIC 2.5, a single WRITE command can write multiple bytes and words to sequential EEPROM locations. For example:

```
All 2 ' {$PBASIC 2.5}
value VAR Byte
value2 VAR Word

value = 18
value2 = 1125

WRITE 25, value, Word value2 ' write byte to location 25
 ' and word to locations 26 and 27
```

### SPECIAL NOTES FOR EEPROM USAGE.

EEPROM differs from RAM, the memory in which variables are stored, in several respects:

1. Writing to EEPROM takes more time than storing a value in a variable. Depending on many factors, it may take several milliseconds for the EEPROM to complete a write. RAM storage is nearly instantaneous.
2. The EEPROM can only accept a finite number of write cycles per location before it wears out. Table 5.123 indicates the guaranteed number of writes before failure. If a program frequently writes to the same EEPROM location, it makes sense to estimate how long it might take to exceed the guaranteed maximum. For example, on the BS2, at one write per second (86,400 writes/day) it would take nearly 116 days of continuous operation to exceed 10 million.
3. The primary function of the EEPROM is to store programs (data is stored in leftover space). If data overwrites a portion of your program, the program will most likely crash.

Check the program's memory map to determine what portion of memory your program occupies and make sure that EEPROM writes cannot stray into this area. You may also use the DATA directive on all BS2 models to set aside EEPROM space.

## WRITE – BASIC Stamp Command Reference

---

On the BS1, location 255 holds the address of the last instruction in your program. Therefore, your program can use any space below the address given in location 255. For example, if location 255 holds the value 100, then your program can use locations 0–99 for data. You can read location 255 at run-time or simply view the Memory Map of the program before you download it. On all BS2 models, you will need to view the Memory Map of the program before you download it to determine the last EEPROM location used. See the Memory Map section of Chapter 3.

On the BS2p, BS2pe, and BS2px, the READ and WRITE commands can  affect locations in any program slot as set by the STORE command. See the STORE command for more information.

### Demo Program (WRITE.bs1)

```
' WRITE.bs1
' This program writes a few bytes to EEPROM and then reads them back out
' and displays them in the Debug window.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL addr = B2 ' address
SYMBOL value = B3 ' value

Main:
 WRITE 0, 100 ' write some data to locations 0 - 3
 WRITE 1, 200
 WRITE 2, 45
 WRITE 3, 28

Read_EE:
 FOR addr = 0 TO 3
 READ addr, value ' read value from address
 DEBUG #addr, ": ", #value, CR ' display address and value
 NEXT
END
```

## 5: BASIC Stamp Command Reference – WRITE



NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

### Demo Program (WRITE.bs2)

```
' WRITE.bs2
' This program writes some data to EEPROM and then reads them back out
' and displays the data in the Debug window.

' {$STAMP BS2}
' {$PBASIC 2.5}

idx VAR Byte ' loop control
value VAR Word(3) ' value(s)

Main:
 WRITE 0, 100 ' single byte
 WRITE 1, Word 1250 ' single word
 WRITE 3, 45, 90, Word 725 ' multi-value write

Read_EE:
 FOR idx = 0 TO 6 ' show raw bytes in EE
 READ idx, value
 DEBUG DEC1 idx, " : ", DEC value, CR
 NEXT
 DEBUG CR

 ' read values as stored

 READ 0, value
 DEBUG DEC value, CR
 READ 1, Word value
 DEBUG DEC value, CR
 READ 3, value(0), value(1), Word value(2)
 FOR idx = 0 TO 2
 DEBUG DEC value(idx), CR
 NEXT
END
```

## ***WRITE – BASIC Stamp Command Reference***

---

## 5: BASIC Stamp Command Reference – XOUT

### XOUT

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|-----|-----|------|-------|------|-------|-------|
|-----|-----|------|-------|------|-------|-------|



**XOUT** *Mpin*, *Zpin*, [ *House* \ *Command* { \ *Cycles* } {, *House* \ *Command* { \ *Cycles* } ... } ]

#### Function

Send an X-10 power-line control command (through the appropriate power-line interface).

- **Mpin** is a variable/constant/expression (0 – 15) that specifies the I/O pin to output X-10 signals (modulation) to the power-line interface device. This pin will be set to output mode.
- **Zpin** is a variable/constant/expression (0 – 15) that specifies the I/O pin that inputs the zero-crossing signal from the power-line interface device. This pin will be set to input mode.
- **House** is a variable/constant/expression (0 – 15) that specifies the X-10 house code (values 0 - 15 representing letters A through P).
- **Command** is a variable/constant/expression (0 – 31) that specifies the command to send. Values 0 – 15 correspond to unit codes 1 – 16. Other commands are shown in Table 5.125.
- **Cycles** is an optional variable/constant/expression (1 – 255) specifying the number of times to transmit a given key or command. If no *Cycles* argument is used, XOUT defaults to two. The *Cycles* argument should be used only with the DIM and BRIGHT command codes

#### Quick Facts

Table 5.124: XOUT Quick Facts.

|                                  | All BS2 Models                                                                                                                                                                                |
|----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Compatible Power-line Interfaces | PL-513 and TW-523                                                                                                                                                                             |
| Special Notes                    | The XOUT command will stop the BASIC Stamp program until it is able to send the transmission. If there is no AC power to the power-line interface, the BASIC Stamp program will halt forever. |

#### Explanation

XOUT lets you control appliances via signals sent through household AC wiring to X-10 modules. The appliances plugged into these modules can be switched on or off; lights may also be dimmed. Each module is

## ***XOUT – BASIC Stamp Command Reference***

---

assigned a house code and unit code by setting dials or switches on the module. To talk to a particular module, XOUT sends the appropriate house code and unit code. The module with the corresponding code listens for its house code again followed by a command (on, off, dim, or bright).

X-10 signals are digital codes imposed on a 120 kHz carrier that is transmitted during zero crossings of the AC line. To send X-10 commands, a controller must synchronize to the AC line frequency with 50  $\mu$ s precision, and transmit an 11-bit code sequence representing the command.

X-10 PROTOCOL DETAILS.

XOUT interfaces to the AC power-line through an approved interface device such as a PL-513 or TW-523, available from X-10 dealers. The hookup requires a length of four-conductor phone cable and a standard modular phone-base connector (6P4C type). Connections are shown in Figure 5.48.

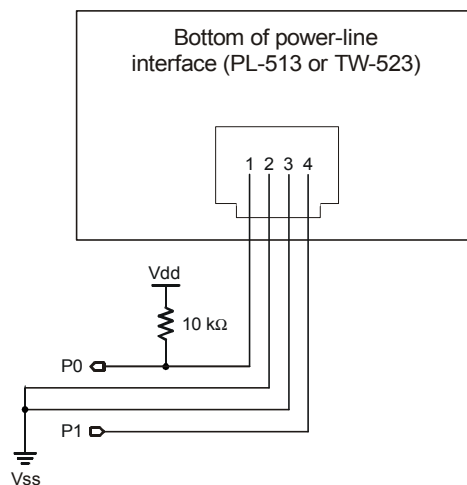


**Figure 5.48:** XOUT Power-Line Interface Circuit.



## 5: BASIC Stamp Command Reference – XOUT

Table 5.125 lists the XOUT command codes and their functions:

**Table 5.125:** XOUT Commands and Their Function.

| Command  | Value  | Function                                        |
|----------|--------|-------------------------------------------------|
| UNITON   | %10010 | Turn on the currently selected unit.            |
| UNITOFF  | %11010 | Turn off the currently selected unit.           |
| UNITSONf | %11100 | Turn off all modules in this house code.        |
| LIGHTSON | %10100 | Turn on all lamp modules in this house code.    |
| DIM      | %11110 | Reduce brightness of currently selected lamp.   |
| BRIGHT   | %10110 | Increase brightness of currently selected lamp. |

Note: In most applications, it's not necessary to know the code for a given X-10 instruction. Just use the command constant (UnitOn, Dim, etc.) instead. But knowing the codes leads to some interesting possibilities. For example, XORing a UnitOn command with the value %1000 turns it into a UnitOff command, and vice-versa. This makes it possible to write the equivalent of an X-10 "toggle" instruction.

A SIMPLE XOUT EXAMPLE: TURNING AN APPLIANCE ON.

Here is an example of the XOUT instruction:

```
Mpin PIN 0 ' modulation pin
Zpin PIN 1 ' zero-cross input

HouseA CON 0 ' House code A = 0
Unit1 CON 0 ' Unit code 1 = 0

XOUT Mpin, Zpin, [HouseA\Unit1] ' get Unit1's attention
XOUT Mpin, Zpin, [HouseA\UNITON] ' turn it on
```

COMBINING MULTIPLE COMMANDS.

You can combine those two XOUT instructions into one like so:

```
XOUT Mpin, Zpin, [HouseA\Unit1\2, HouseA\UNITON] ' Unit 1 on.
```

Note that to complete the attention-getting code HouseA\Unit1 we tacked on the normally optional cycles entry \2 to complete the command before beginning the next one. Always specify two cycles in multiple commands unless you're adjusting the brightness of a lamp module.

DIMMING LIGHTS.

Here is an example of a lamp-dimming instruction:

```
Mpin PIN 0 ' modulation pin
Zpin PIN 1 ' zero-cross input

HouseA CON 0 ' House code A = 0
Unit1 CON 0 ' Unit code 1 = 0

XOUT Mpin, Zpin, [HouseA\Unit1] ' get Unit1's attention
XOUT Mpin, Zpin, [HouseA\UNITOFF\2] ' turn it off
XOUT Mpin, Zpin, [HouseA\DIM\10] ' dim half way
```

## XOUT – BASIC Stamp Command Reference

---

The dim/bright commands support 19 brightness levels. Lamp modules may also be turned on and off using the standard UnitOn and UnitOff commands. In the example instruction above, we dimmed the lamp by first turning it completely off, then sending 10 cycles of the Dim command. This may seem odd, but it follows the peculiar logic of the X-10 system.

### Demo Program (X10.bs2)

```
' XOUT.BS2
' This program--really two program fragments--demonstrates the syntax and
' use of the XOUT command. XOUT works like pressing the buttons on an X-10
' control box; first you press one of 16 keys to identify the unit you want
' to control, then you press the key for the action you want that unit to
' take (turn ON, OFF, Bright, or Dim). There are also two group-action
' keys, Lights ON and All OFF. Lights ON turns all lamp modules on without
' affecting appliance modules. All OFF turns off all modules, both lamp and
' appliance types. Connect the BASIC Stamp to a power-line interface as
' shown in the XOUT command description in the manual.

' {$STAMP BS2}
' {$PBASIC 2.5}

Mpin PIN 0 ' modulation pin
Zpin PIN 1 ' zero-cross input

HouseA CON 0 ' House code A = 0
Unit1 CON 0 ' Unit code 1 = 0
Unit2 CON 1 ' Unit code 2 = 1

' This first example turns a standard (appliance or non-dimmer lamp) module
' ON, then OFF. Note that once the Unit code is sent, it need not be
' repeated
' --subsequent instructions are understood to be addressed to that unit.

Main:
 XOUT Mpin, Zpin, [HouseA\Unit1\2] ' select Unit1 (appliance module)
 XOUT Mpin, Zpin, [HouseA\UNITON] ' turn it on

 PAUSE 1000 ' wait one second

 XOUT Mpin, Zpin, [HouseA\UNITOFF] ' then turn it off

' The next example talks to a lamp module using the dimmer feature. Dimmers
' go from full ON to dimmed OFF in 19 steps. Because dimming is relative to
' the current state of the lamp, the only guaranteed way to set a
' predefined brightness level is to turn the dimmer fully OFF, then ON,
' then dim to the desired level.

 XOUT Mpin, Zpin, [HouseA\Unit2\2] ' select Unit2 (lamp module)
```

All 2

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

## ***5: BASIC Stamp Command Reference – XOUT***

---

```
' This example shows the use of the optional Cycles argument. Here we DIM
' for 10 cycles.
```

```
XOUT Mpin, Zpin, [HouseA\UNITOFF\2, HouseA\DIM\10]
STOP
```

## ***XOUT – BASIC Stamp Command Reference***

---

## Appendix A: ASCII Chart

### ASCII Chart (first 128 characters)

| Dec | Hex | Char | Name / Function    | Dec | Hex | Char  | Dec | Hex | Char | Dec | Hex | Char   |
|-----|-----|------|--------------------|-----|-----|-------|-----|-----|------|-----|-----|--------|
| 0   | 00  | NUL  | Null               | 32  | 20  | space | 64  | 40  | @    | 96  | 60  | `      |
| 1   | 01  | SOH  | Start Of Heading   | 33  | 21  | !     | 65  | 41  | A    | 97  | 61  | a      |
| 2   | 02  | STX  | Start Of Text      | 34  | 22  | "     | 66  | 42  | B    | 98  | 62  | b      |
| 3   | 03  | ETX  | End Of Text        | 35  | 23  | #     | 67  | 43  | C    | 99  | 63  | c      |
| 4   | 04  | EOT  | End Of Transmit    | 36  | 24  | \$    | 68  | 44  | D    | 100 | 64  | d      |
| 5   | 05  | ENQ  | Enquiry            | 37  | 25  | %     | 69  | 45  | E    | 101 | 65  | e      |
| 6   | 06  | ACK  | Acknowledge        | 38  | 26  | &     | 70  | 46  | F    | 102 | 66  | f      |
| 7   | 07  | BEL  | Bell               | 39  | 27  | '     | 71  | 47  | G    | 103 | 67  | g      |
| 8   | 08  | BS   | Backspace          | 40  | 28  | (     | 72  | 48  | H    | 104 | 68  | h      |
| 9   | 09  | HT   | Horizontal Tab     | 41  | 29  | )     | 73  | 49  | I    | 105 | 69  | i      |
| 10  | 0A  | LF   | Line Feed          | 42  | 2A  | *     | 74  | 4A  | J    | 106 | 6A  | j      |
| 11  | 0B  | VT   | Vertical Tab       | 43  | 2B  | +     | 75  | 4B  | K    | 107 | 6B  | k      |
| 12  | 0C  | FF   | Form Feed          | 44  | 2C  | ,     | 76  | 4C  | L    | 108 | 6C  | l      |
| 13  | 0D  | CR   | Carriage Return    | 45  | 2D  | -     | 77  | 4D  | M    | 109 | 6D  | m      |
| 14  | 0E  | SO   | Shift Out          | 46  | 2E  | .     | 78  | 4E  | N    | 110 | 6E  | n      |
| 15  | 0F  | SI   | Shift In           | 47  | 2F  | /     | 79  | 4F  | O    | 111 | 6F  | o      |
| 16  | 10  | DLE  | Data Line Escape   | 48  | 30  | 0     | 80  | 50  | P    | 112 | 70  | p      |
| 17  | 11  | DC1  | Device Control 1   | 49  | 31  | 1     | 81  | 51  | Q    | 113 | 71  | q      |
| 18  | 12  | DC2  | Device Control 2   | 50  | 32  | 2     | 82  | 52  | R    | 114 | 72  | r      |
| 19  | 13  | DC3  | Device Control 3   | 51  | 33  | 3     | 83  | 53  | S    | 115 | 73  | s      |
| 20  | 14  | DC4  | Device Control 4   | 52  | 34  | 4     | 84  | 54  | T    | 116 | 74  | t      |
| 21  | 15  | NAK  | Non Acknowledge    | 53  | 35  | 5     | 85  | 55  | U    | 117 | 75  | u      |
| 22  | 16  | SYN  | Synchronous Idle   | 54  | 36  | 6     | 86  | 56  | V    | 118 | 76  | v      |
| 23  | 17  | ETB  | End Transmit Block | 55  | 37  | 7     | 87  | 57  | W    | 119 | 77  | w      |
| 24  | 18  | CAN  | Cancel             | 56  | 38  | 8     | 88  | 58  | X    | 120 | 78  | x      |
| 25  | 19  | EM   | End Of Medium      | 57  | 39  | 9     | 89  | 59  | Y    | 121 | 79  | y      |
| 26  | 1A  | SUB  | Substitute         | 58  | 3A  | :     | 90  | 5A  | Z    | 122 | 7A  | z      |
| 27  | 1B  | ESC  | Escape             | 59  | 3B  | ;     | 91  | 5B  | [    | 123 | 7B  | {      |
| 28  | 1C  | FS   | File Separator     | 60  | 3C  | <     | 92  | 5C  | \    | 124 | 7C  |        |
| 29  | 1D  | GS   | Group Separator    | 61  | 3D  | =     | 93  | 5D  | ]    | 125 | 7D  | }      |
| 30  | 1E  | RS   | Record Separator   | 62  | 3E  | >     | 94  | 5E  | ^    | 126 | 7E  | ~      |
| 31  | 1F  | US   | Unit Separator     | 63  | 3F  | ?     | 95  | 5F  | _    | 127 | 7F  | delete |

Note that the control codes (lowest 32 ASCII characters) have no standardized screen symbols. The characters listed for them are just names used in referring to these codes. For example, to move the cursor to the beginning of the next line of a printer or terminal often requires sending line feed and carriage return codes. This common pair is referred to as "LF/CR."

## ***ASCII Chart***

---

## Appendix B: Reserved Words

### Reserved Words

This appendix contains complete listings of the reserved words for PBASIC 1.0, PBASIC 2.0, and PBASIC 2.5, current with the BASIC Stamp Editor v2.1.

The reserved word lists have been organized into 4 tables, because it varies with each BASIC Stamp model and version of PBASIC. Table B.1 shows the reserved words for the BASIC Stamp 1, using the required PBASIC 1.0.

**Table B.1:** BS1 Reserved Words.

| BS1         |          |        |            |        |
|-------------|----------|--------|------------|--------|
| AND         | GOSUB    | N2400  | PIN0..PIN7 | SOUND  |
| B0..B13     | GOTO     | NAP    | PINS       | STEP   |
| BIT0..BIT15 | HIGH     | NEXT   | PORT       | SYMBOL |
| BRANCH      | IF       | ON300  | POT        | T300   |
| BSAVE       | INPUT    | ON600  | PULSIN     | T600   |
| BUTTON      | LET      | ON1200 | PULSOUT    | T1200  |
| CLS         | LOOKDOWN | ON2400 | PWM        | T2400  |
| CR          | LOOKUP   | OR     | RANDOM     | THEN   |
| DEBUG       | LOW      | OT300  | READ       | TO     |
| DIR0..DIR7  | MAX      | OT600  | RETURN     | TOGGLE |
| DIRS        | MIN      | OT1200 | REVERSE    | W0..W6 |
| EEPROM      | N300     | OT2400 | SERIN      | WRITE  |
| END         | N600     | OUTPUT | SEROUT     |        |
| FOR         | N1200    | PAUSE  | SLEEP      |        |

Table B.2 on the following page lists the reserved words common to all BS2 models, including those for PBASIC 2.0 and PBASIC 2.5. Words listed that are only reserved when using PBASIC 2.5 are marked with ( <sup>2.5</sup> ).

## Reserved Words

| All BS2 Models            |                          |                     |                       |
|---------------------------|--------------------------|---------------------|-----------------------|
| #CASE <sup>2.5</sup>      | CRSRX <sup>2.5</sup>     | INC                 | PULSIN                |
| #DEFINE <sup>2.5</sup>    | CRSRXY <sup>2.5</sup>    | IND                 | PULSOUT               |
| #ELSE <sup>2.5</sup>      | CRSRY <sup>2.5</sup>     | INH                 | PWM                   |
| #ENDIF <sup>2.5</sup>     | DATA                     | INL                 | RANDOM                |
| #ENDSELECT <sup>2.5</sup> | DCD                      | INPUT               | RCTIME                |
| #ERROR <sup>2.5</sup>     | DEBUG                    | INS                 | READ                  |
| #IF <sup>2.5</sup>        | DEBUGIN <sup>2.5</sup>   | ISBIN               | REP                   |
| #SELECT <sup>2.5</sup>    | DEC                      | ISBIN1...ISBIN16    | RETURN                |
| #THEN <sup>2.5</sup>      | DEC1...DEC5              | ISHEX               | REV                   |
| \$PBASIC                  | DIG                      | ISHEX1...ISHEX4     | REVERSE               |
| \$PORT                    | DIM                      | LF <sup>2.5</sup>   | SBIN                  |
| \$STAMP                   | DIR0...DIR15             | LIGHTSON            | SBIN1...SBIN16        |
| ABS                       | DIRA                     | LOOKDOWN            | SDEC                  |
| AND                       | DIRB                     | LOOKUP              | SDEC1...SDEC5         |
| ASC                       | DIRC                     | LOOP <sup>2.5</sup> | SELECT <sup>2.5</sup> |
| ATN                       | DIRD                     | LOW                 | SERIN                 |
| B0...B25                  | DIRH                     | LOWBIT              | SEROUT                |
| BELL                      | DIRL                     | LOWBYTE             | SHEX                  |
| BIN                       | DIRS                     | LOWNIB              | SHEX1...SHEX4         |
| BIN1...BIN16              | DO <sup>2.5</sup>        | LSBFIRST            | SHIFTIN               |
| BIT                       | DTMFOUT                  | LSBPOST             | SHIFTOUT              |
| BIT0...BIT15              | ELSE <sup>2.5</sup>      | LSBPRES             | SIN                   |
| BKSP                      | ELSEIF <sup>2.5</sup>    | MAX                 | SKIP                  |
| BRANCH                    | END                      | MIN                 | SLEEP                 |
| BRIGHT                    | ENDIF <sup>2.5</sup>     | MSBFIRST            | SNUM                  |
| BS1                       | ENDSELECT <sup>2.5</sup> | MSBPOST             | SQR                   |
| BS2                       | EXIT <sup>2.5</sup>      | MSBPRES             | STEP                  |
| BS2E                      | FOR                      | NAP                 | STOP                  |
| BS2P                      | FREQOUT                  | NCD                 | STR                   |
| BS2PE                     | GOSUB                    | NEXT                | TAB                   |
| BS2SX                     | GOTO                     | NIB                 | THEN                  |
| BUTTON                    | HEX                      | NIB0...NIB3         | TO                    |
| BYTE                      | HEX1...HEX4              | NOT                 | TOGGLE                |
| BYTE0                     | HIGH                     | NUM                 | UNITOFF               |
| BYTE1                     | HIGHBIT                  | ON <sup>2.5</sup>   | UNITON                |
| CASE <sup>2.5</sup>       | HIGHBYTE                 | OR                  | UNITSOFF              |
| CLRDN <sup>2.5</sup>      | HIGHNIB                  | OUT0...OUT15        | UNTIL <sup>2.5</sup>  |
| CLREOL <sup>2.5</sup>     | HOME                     | OUTA                | VAR                   |
| CLS                       | HYP                      | OUTB                | W0...W12              |
| CON                       | IBIN                     | OUTC                | WAIT                  |
| COS                       | IBIN1...IBIN16           | OUTD                | WAITSTR               |
| COUNT                     | IF                       | OUTH                | WHILE <sup>2.5</sup>  |
| CR                        | IHEX                     | OUTL                | WORD                  |
| CRSRDN <sup>2.5</sup>     | IHEX1...IHEX4            | OUPUT               | WRITE                 |
| CRSRLF <sup>2.5</sup>     | IN0...IN15               | OUTS                | XOR                   |
| CRSRRT <sup>2.5</sup>     | INA                      | PAUSE               | XOUT                  |
| CRSRUP <sup>2.5</sup>     | INB                      | PIN <sup>2.5</sup>  |                       |

**Table B.2** Reserved Words common to all BS2 Models.

NOTE: This list includes reserved words for both PBASIC 2.0 and PBASIC 2.5. Words indicated with the symbol ( <sup>2.5</sup> ) are only reserved if used with PBASIC 2.5.



## Appendix B: Reserved Words

There are some reserved words unique to specific BS2 models.

The BS2e and BS2sx have all the reserved words shown in Table B.2, plus those shown in Table B.3. These additional words are reserved in both PBASIC 2.0 and PBASIC 2.5.

**Table B.3:** Additional Reserved Words for the BS2e and BS2sx.

| BS2e and BS2sx |     |     |
|----------------|-----|-----|
| GET            | PUT | RUN |

The BS2p and BS2pe have all the reserved words shown in Table B.2, plus those shown in Table B.4. These additional words are reserved in both PBASIC 2.0 and PBASIC 2.5.

**Table B.4:** Additional Reserved Words for the BS2p and BS2pe.

| BS2p and BS2pe |        |          |          |
|----------------|--------|----------|----------|
| AUXIO          | LCDCMD | OWOUT    | POLLWAIT |
| GET            | LCDIN  | POLLIN   | PUT      |
| I2CIN          | LCDOUT | POLLMODE | RUN      |
| I2COUT         | MAINIO | POLLOUT  | SPSTR    |
| IOTERM         | OWIN   | POLLRUN  | STORE    |

The BS2px has all the reserved words shown in Table B.2, plus those shown in Table B.5. These additional words are reserved in both PBASIC 2.0 and PBASIC 2.5.

**Table B.5:** Additional Reserved Words for the BS2px.

| BS2px     |        |          |           |
|-----------|--------|----------|-----------|
| AUXIO     | IOTERM | POLLIN   | RUN       |
| COMPARE   | LCDCMD | POLLMODE | SCHMITT   |
| CONFIGPIN | LCDIN  | POLLOUT  | SPSTR     |
| DIRECTION | LCDOUT | POLLRUN  | STORE     |
| GET       | MAINIO | POLLWAIT | THRESHOLD |
| I2CIN     | OWIN   | PULLUP   |           |
| I2COUT    | OWOUT  | PUT      |           |

## ***Reserved Words***

---

## Appendix C: Conversion Formatters

### Conversion Formatters

This appendix lists the Conversion Formatters available for the commands DEBUGIN, I2CIN, LCDIN, OWIN and SERIN and demonstrates, through various input/output data examples, exactly what will be received when using these formatters.

The following tables show data examples (characters received) across the top and formatters across the left side, with the results of that combination shown in the target cell.

For example, with the following code:

```
Value VAR WORD
```

```
DEBUGIN SDEC Value
```

the Decimal Formatters table shows us that if the characters -123 (followed by a carriage return) are received, the SDEC formatter will translate that to the word-sized decimal number -123 and will store that value into the *Value* variable.

NOTE: In all tables below, values in target cells represent the number base of the formatter (decimal for DEC, hexadecimal for HEX, etc) except where noted. Additionally, "--" means no valid data (or not enough valid data) was received so the command will halt forever unless additional data is received or SERIN's *Timeout* argument is used.

**Table C.1:** Decimal Formatters.

| Decimal Formatters | Characters Received |     |      |       |       |        |        |         |
|--------------------|---------------------|-----|------|-------|-------|--------|--------|---------|
|                    | ⊗                   | 123 | 123⊗ | -123⊗ | ⊗123⊗ | 12345⊗ | 65536⊗ | 255255⊗ |
| DEC                | --                  | --  | 123  | 123   | 123   | 12345  | 0      | 58647   |
| DEC1               | --                  | 1   | 1    | 1     | 1     | 1      | 6      | 2       |
| DEC2               | --                  | 12  | 12   | 12    | 12    | 12     | 65     | 25      |
| DEC3               | --                  | 123 | 123  | 123   | 123   | 123    | 655    | 255     |
| DEC4               | --                  | --  | 123  | 123   | 123   | 1234   | 6553   | 2552    |
| DEC5               | --                  | --  | 123  | 123   | 123   | 12345  | 0      | 25525   |
| SDEC               | --                  | --  | 123  | -123  | 123   | 12345  | 0      | -6889   |
| SDEC1              | --                  | 1   | 1    | -1    | 1     | 1      | 6      | 2       |
| SDEC2              | --                  | 12  | 12   | -12   | 12    | 12     | 65     | 25      |
| SDEC3              | --                  | 123 | 123  | -123  | 123   | 123    | 655    | 255     |
| SDEC4              | --                  | --  | 123  | -123  | 123   | 1234   | 6553   | 2552    |

⊗ Means any non-decimal-numeric characters such as letters, spaces, minus signs, carriage returns, control characters, etc. (Decimal numerics are: 0,1,2,3,4,5,6,7,8 and 9).

## Conversion Formatters

| Hexadecimal Formatters | Characters Received |    |     |      |      |       |        |        |
|------------------------|---------------------|----|-----|------|------|-------|--------|--------|
|                        | ⊗                   | 1F | 1F⊗ | -1F⊗ | ⊗1F⊗ | 15AF⊗ | 10000⊗ | 3E517⊗ |
| HEX                    | --                  | -- | 1F  | 1F   | 1F   | 15AF  | 0      | E517   |
| HEX1                   | --                  | 1  | 1   | 1    | 1    | 1     | 1      | 3      |
| HEX2                   | --                  | 1F | 1F  | 1F   | 1F   | 15    | 10     | 3E     |
| HEX3                   | --                  | -- | 1F  | 1F   | 1F   | 15A   | 100    | 3E5    |
| HEX4                   | --                  | -- | 1F  | 1F   | 1F   | 15AF  | 1000   | 3E51   |
| SHEX                   | --                  | -- | 1F  | -1F  | 1F   | 15AF  | 0      | -1AE9  |
| SHEX1                  | --                  | 1  | 1   | -1   | 1    | 1     | 1      | 3      |
| SHEX2                  | --                  | 1F | 1F  | -1F  | 1F   | 15    | 10     | 3E     |
| SHEX3                  | --                  | -- | 1F  | -1F  | 1F   | 15A   | 100    | 3E5    |

**Table C.2:** Hexadecimal Formatters.

NOTE: The HEX formatters are not case sensitive. For example, 1F is the same as 1f.

⊗ Means any non-hexadecimal-numeric characters such as letters (greater than F), spaces, minus signs, carriage returns, control characters, etc. (Hexadecimal numerics are: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F).

| Additional Hexadecimal Formatters | Characters Received |    |     |      |       |        |        |         |
|-----------------------------------|---------------------|----|-----|------|-------|--------|--------|---------|
|                                   | ⊗                   | 1F | 1F⊗ | \$1F | \$1F⊗ | -\$1F⊗ | ⊗\$1F⊗ | \$15AF⊗ |
| IHEX                              | --                  | -- | --  | --   | 1F    | 1F     | 1F     | 15AF    |
| IHEX1                             | --                  | -- | --  | 1    | 1     | 1      | 1      | 1       |
| IHEX2                             | --                  | -- | --  | 1F   | 1F    | 1F     | 1F     | 15      |
| IHEX3                             | --                  | -- | --  | --   | 1F    | 1F     | 1F     | 15A     |
| IHEX4                             | --                  | -- | --  | --   | 1F    | 1F     | 1F     | 15AF    |
| ISHEX                             | --                  | -- | --  | --   | 1F    | -1F    | 1F     | 15AF    |
| ISHEX1                            | --                  | -- | --  | 1    | 1     | -1     | 1      | 1       |
| ISHEX2                            | --                  | -- | --  | 1F   | 1F    | -1F    | 1F     | 15      |
| ISHEX3                            | --                  | -- | --  | --   | 1F    | -1F    | 1F     | 15A     |
| ISHEX4                            | --                  | -- | --  | --   | 1F    | -1F    | 1F     | 15AF    |

**Table C.3:** Additional Hexadecimal Formatters.

NOTE: The HEX formatters are not case sensitive. For example, 1F is the same as 1f.

⊗ Means any non-hexadecimal-numeric characters such as letters (greater than F), spaces, minus signs, carriage returns, control characters, etc. (Hexadecimal numerics are: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F).

| Binary Formatters | Characters Received |    |     |      |      |      |        |
|-------------------|---------------------|----|-----|------|------|------|--------|
|                   | ⊗                   | 11 | 11⊗ | -11⊗ | ⊗11⊗ | 101⊗ | 3E517⊗ |
| BIN               | --                  | -- | 11  | 11   | 11   | 101  | 1      |
| BIN1              | --                  | 1  | 1   | 1    | 1    | 1    | 1      |
| BIN2              | --                  | 11 | 11  | 11   | 11   | 10   | 1      |
| BIN3 – BIN16      | --                  | -- | 11  | 11   | 11   | 101  | 1      |
| SBIN              | --                  | -- | 11  | -11  | 11   | 101  | 1      |
| SBIN1             | --                  | 1  | 1   | -1   | 1    | 1    | 1      |
| SBIN2             | --                  | 11 | 11  | -11  | 11   | 10   | 1      |

**Table C.4:** Binary Formatters.

⊗ Means any non-binary-numeric characters such as letters, spaces, minus signs, carriage returns, control characters, etc. (Binary numerics are: 0 and 1).

## Appendix C: Conversion Formatters

**Table C.5:** Additional Binary Formatters.

| Additional Binary Formatters | Characters Received |    |     |     |      |       |       |       |
|------------------------------|---------------------|----|-----|-----|------|-------|-------|-------|
|                              | ⊗                   | 11 | 11⊗ | %11 | %11⊗ | -%11⊗ | ⊗%11⊗ | %101⊗ |
| IBIN                         | --                  | -- | --  | --  | 11   | 11    | 11    | 101   |
| IBIN1                        | --                  | -- | --  | 1   | 1    | 1     | 1     | 1     |
| IBIN2                        | --                  | -- | --  | 11  | 11   | 11    | 11    | 10    |
| IBIN3 – IBIN16               | --                  | -- | --  | --  | 11   | 11    | 11    | 101   |
| ISBIN                        | --                  | -- | --  | --  | 11   | -11   | 11    | 101   |
| ISBIN1                       | --                  | -- | --  | 1   | 1    | -1    | 1     | 1     |
| ISBIN2                       | --                  | -- | --  | 11  | 11   | -11   | 11    | 10    |
| ISBIN3 – ISBIN16             | --                  | -- | --  | --  | 11   | -11   | 11    | 101   |

⊗ Means any non-binary-numeric characters such as letters, spaces, minus signs, carriage returns, control characters, etc. (Binary numerics are: 0 and 1).

**Table C.6:** NUM and SNUM with Decimal Data.

| General (Dec. Data) | Characters Received |     |      |       |       |        |        |         |
|---------------------|---------------------|-----|------|-------|-------|--------|--------|---------|
|                     | ⊗                   | 123 | 123⊗ | -123⊗ | ⊗123⊗ | 12345⊗ | 65536⊗ | 255255⊗ |
| NUM                 | --                  | --  | 123  | 123   | 123   | 12345  | 0      | 58647   |

⊗ Means any non-decimal-numeric characters such as letters, spaces, minus signs, carriage returns, control characters, etc. (Decimal numerics are: 0,1,2,3,4,5,6,7,8 and 9).

**Table C.7:** NUM and SNUM with Hexadecimal Data.

| General (Hex. Data) | Characters Received |    |     |      |       |        |        |         |
|---------------------|---------------------|----|-----|------|-------|--------|--------|---------|
|                     | ⊗                   | 1F | 1F⊗ | \$1F | \$1F⊗ | -\$1F⊗ | ⊗\$1F⊗ | \$15AF⊗ |
| NUM                 | --                  | 1* | 1*  | --   | 1F    | 1F     | 1F     | 15AF    |

NOTE: Hexadecimal data is not case sensitive. For example, 1F is the same as 1f.

⊗ Means any non-hexadecimal-numeric characters such as letters (greater than F), spaces, minus signs, carriage returns, control characters, etc. (Hexadecimal numerics are: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F).

\* Invalid data; treated as decimal number *one* because no \$ preceded it and non-decimal digit followed it.

**Table C.8:** NUM and SNUM with Binary Data.

| General (Bin. Data) | Characters Received |    |     |     |      |       |       |       |
|---------------------|---------------------|----|-----|-----|------|-------|-------|-------|
|                     | ⊗                   | 11 | 11⊗ | %11 | %11⊗ | -%11⊗ | ⊗%11⊗ | %101⊗ |
| NUM                 | --                  | -- | 11* | --  | 11   | 11    | 11    | 101   |

⊗ Means any non-binary-numeric characters such as letters, spaces, minus signs, carriage returns, control characters, etc. (Binary numerics are: 0 and 1).

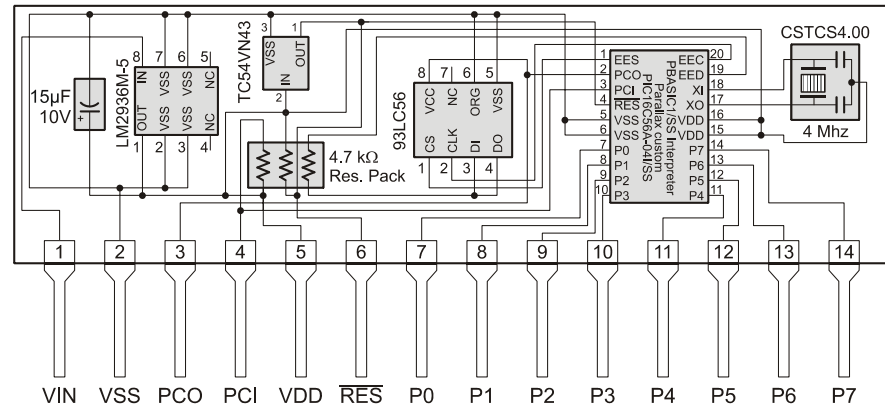
\* Invalid data; treated as decimal number *eleven* because no % preceded it.

## ***Conversion Formatters***

---

## Appendix D: BASIC Stamp Schematics

### BASIC Stamp 1 Schematic (Rev B)



#### Notes:

The 15µF, 10V capacitor may be a 10-22µF, 6.3-16V tantalum capacitor.

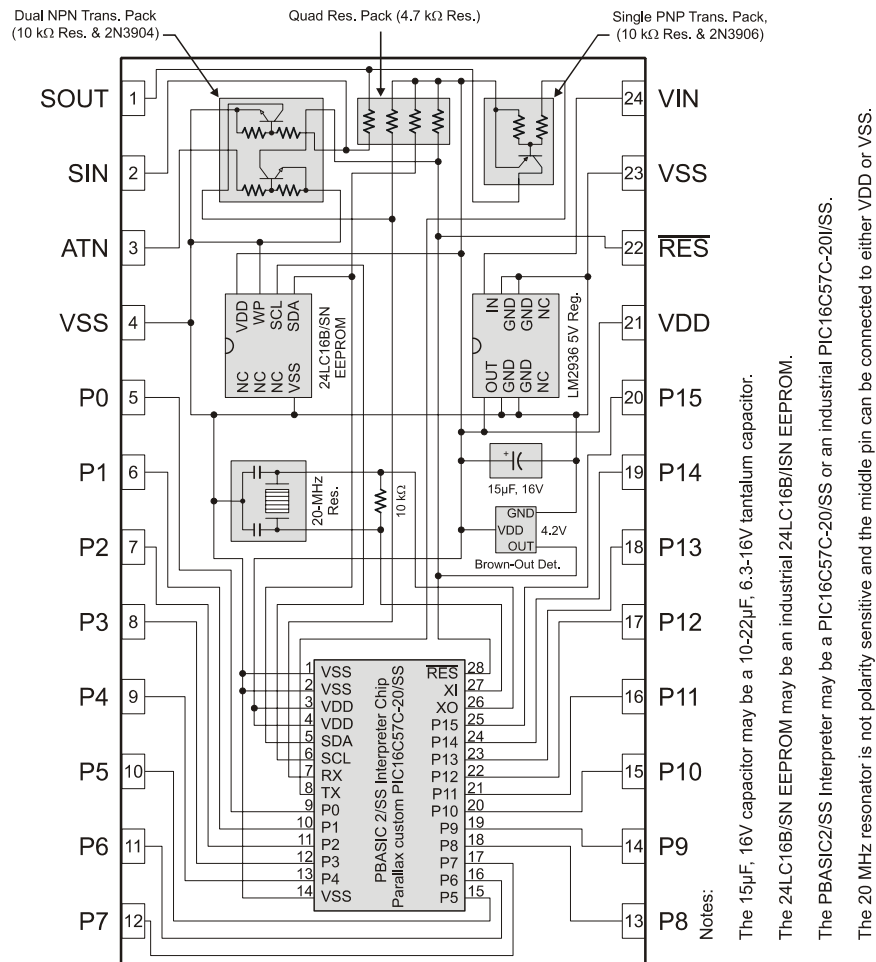
The 93LC56 EEPROM may be a 93LC56A, 93LC66 or 93LC66A.

The 4 MHz resonator is not polarity sensitive and the middle pin can be connected to either VDD or VSS.

The PBASIC/SS Interpreter chip may be a commercial PIC16C56A-04/SS or an industrial PIC16C56A-04I/SS

# BASIC Stamp Schematics

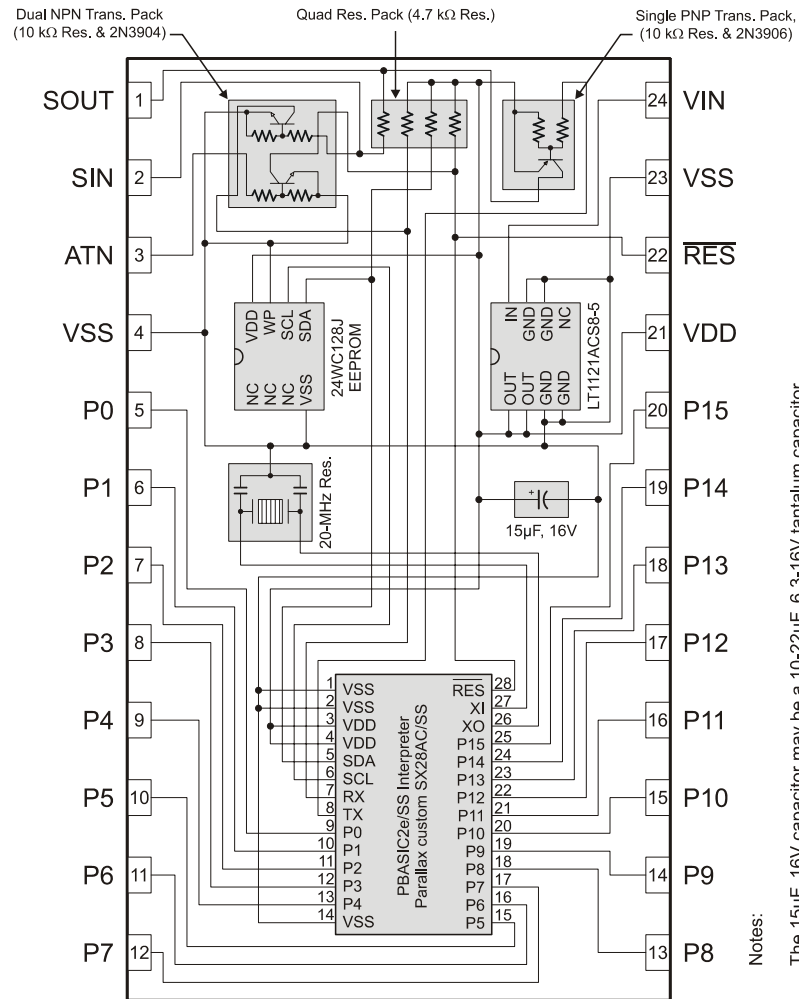
## BASIC Stamp 2 Schematic (Rev G)





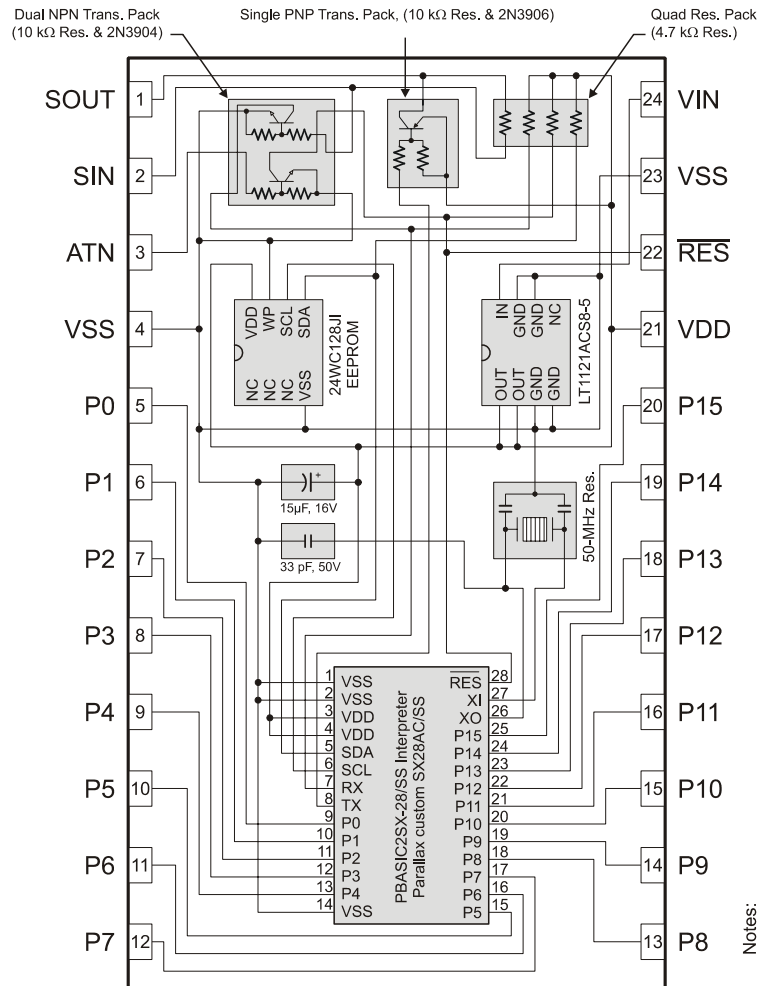
## Appendix D: BASIC Stamp Schematics

### BASIC Stamp 2e Schematic (Rev B)



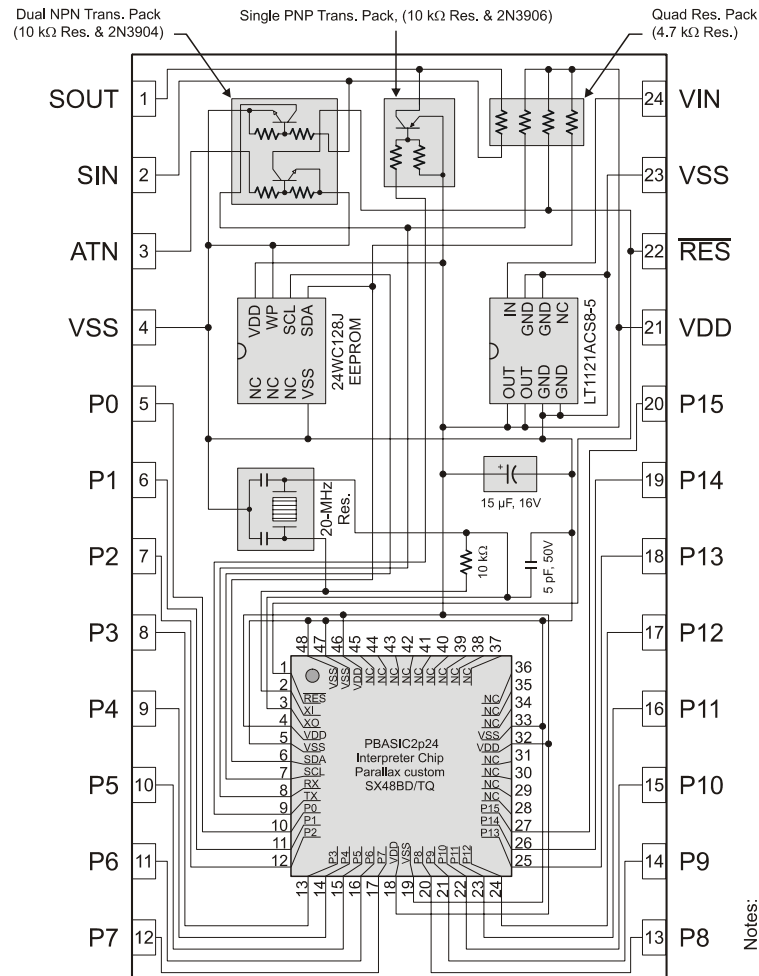
# BASIC Stamp Schematics

## BASIC Stamp 2sx Schematic (Rev E)



## Appendix D: BASIC Stamp Schematics

### BASIC Stamp 2p24 Schematic (Rev C)



# BASIC Stamp Schematics

## BASIC Stamp 2p40 Schematic (Rev B)

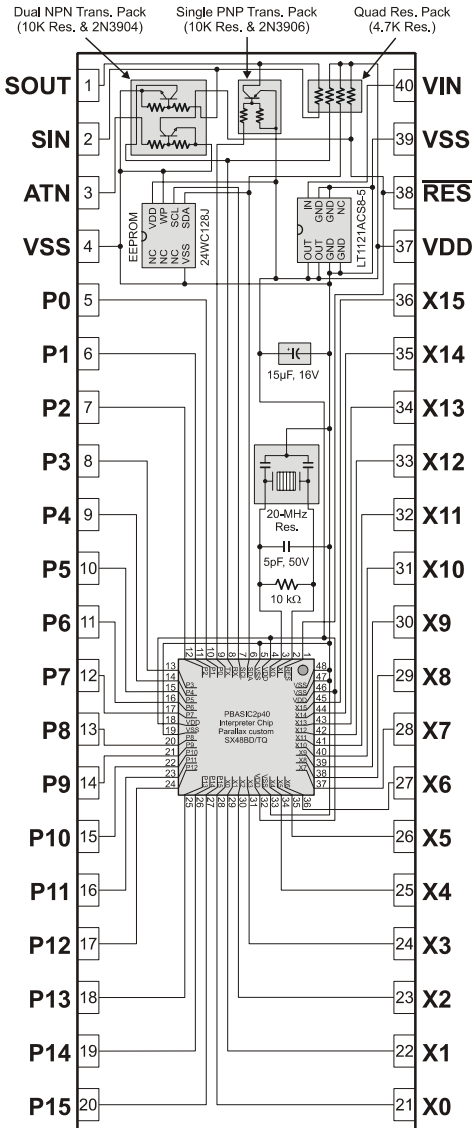
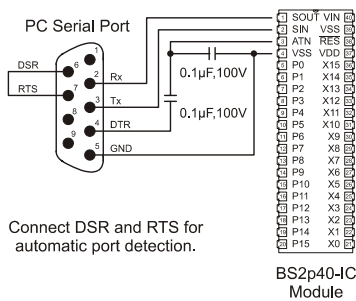
### Notes:

The 15 $\mu$ F, 16V capacitor may be a 10-22 $\mu$ F, 6.3-16V tantalum capacitor.

The 24WC128J EEPROM may be an industrial 24WC128JI EEPROM.

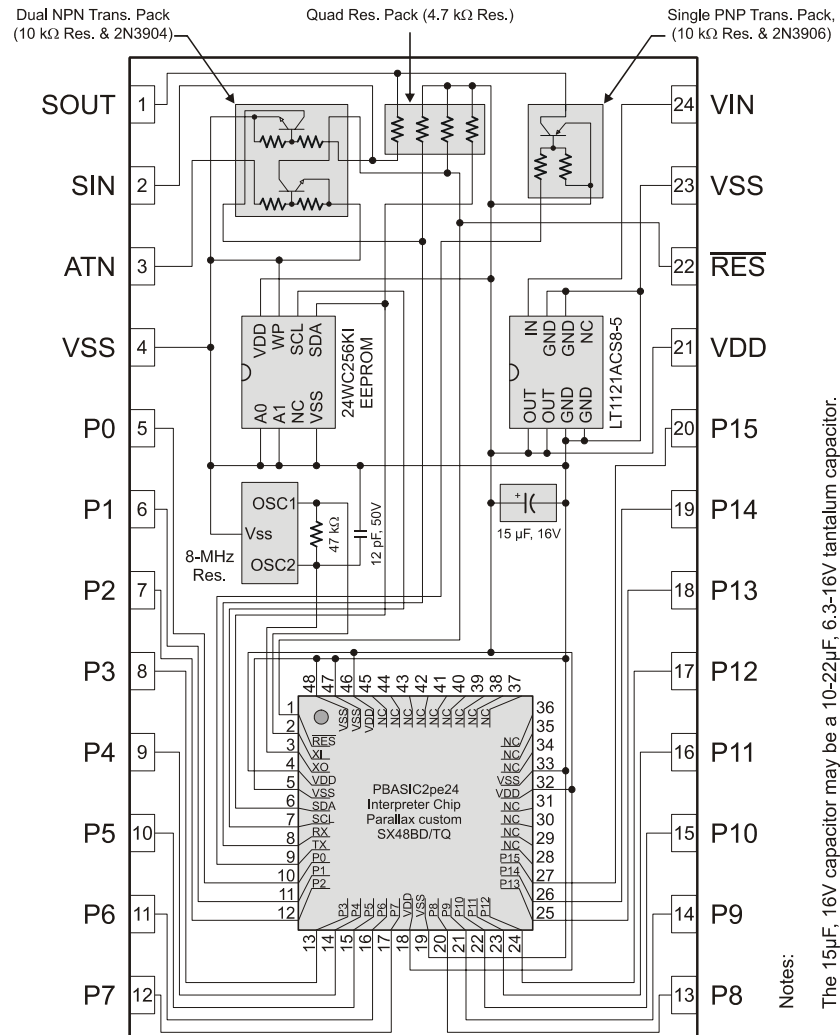
The 20 MHz resonator is not polarity sensitive and the middle pin can be connected to either VDD or VSS.

### PC to BS2p40-IC Connection



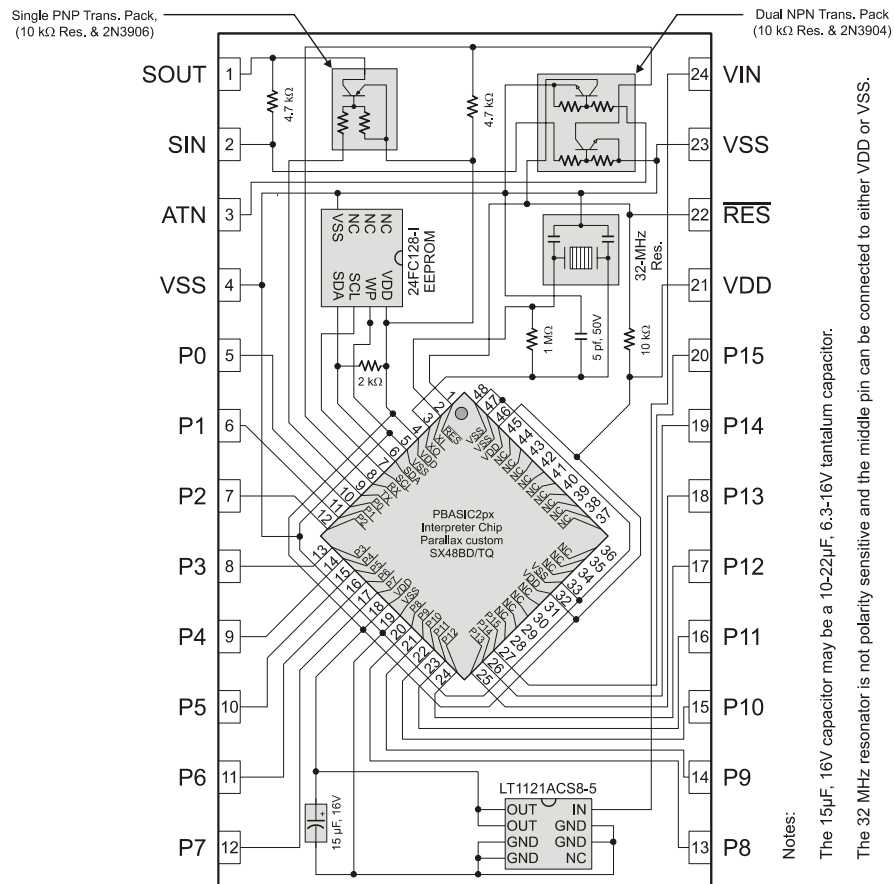
## Appendix D: BASIC Stamp Schematics

### BASIC Stamp 2pe Schematic (Rev B)



# BASIC Stamp Schematics

## BASIC Stamp 2px Schematic (Rev A)



## — # —

#DEFINE, 71  
#ERROR, 75  
#IF...#THEN, 72  
#SELECT...CASE, 74

## — \$ —

\$PBASIC Directive, 32, 44–45  
    Defaults, 44  
    required with syntax enhancements, 124  
    Toolbar Icons, 43  
\$PORT Directive, 45–46, 50  
\$STAMP Directive, 32, 43–44  
    Format, 43  
    Toolbar Icons, 43  
    with Multi-File Projects, 68

## — . —

.obj File, 76

## — 1 —

1-Wire Protocol, 295–301, 303–10

## — 2 —

24LC16B EEPROM, 218, 226

## — A —

ABS, 105  
Absolute Value (ABS), 105  
Accessing I/O Pins, 81, 83, 84  
Add (+), 109  
Aliases, 89–91

Aliases and Variable Modifiers, 89–91

Amplifier, 180, 200, 446

ANALOG I/O, 179, 199, 339, 355, 363, 445, *See* PWM, POT, RCTIME

AND, 235

AND (&), 109, 118

AND NOT (&/), 109, 120

Architecture, 29, 81

Arctangent (ATN), 109, 114

Arrays, 87–89

ASC, 163

ASC ?, 165, 228, 305, 422

ASCII

    Control Characters (Debug Terminal), 66

    Display in Debug Terminal, 161, 163

    Display in Memory Map, 51

ASCII Chart, 471

ASCII Notation, 96

ASYNCHRONOUS SERIAL, 394, 416,

*See also* SERIN, SEROUT, OWIN, OWOUT

ATN, 109, 114

ATN (Pin), 14, 15, 18, 20, 21, 23

Auto Indent, 57

Auxiliary I/O Pins, 129, 247

AUXIO, 129–30, 247, 283

## — B —

B0-B13, 82

Backup Copy, 61

BASIC Stamp

    Architecture, 29, 81

    Hardware, 7

    Memory, 29, 81

    Networking, 425

    Pinouts, 8–20

    Reserved Words, 473

BASIC Stamp 1 Rev. Dx, 11

BASIC Stamp Projects. *See* Multi-file Projects

# Index

---

**BASIC Stamp Schematics**, 481  
**BASIC Stamp Windows Editor**, 35  
**Baud Mode (table)**, 396, 418  
**Baud Mode, Choosing**, 397, 419  
**BELL**, 168  
**BIN**, 162, 163, 173, 220, 227, 260, 265, 298, 306, 403, 422  
**Binary Coded Decimal**, 97  
**Binary Notation**, 96  
**Binary Operators**, 104, 109–21  
    Add (+), 109  
    AND (&), 109, 118  
    AND NOT (&/), 109, 120  
    Arctangent (ATN), 109, 114  
    Digit (DIG), 109, 117  
    Divide (/), 109, 113  
    Hypotenuse (HYP), 109, 115  
    Maximum (MAX), 109, 116  
    Minimum (MIN), 109, 115  
    Modulus (/), 109, 113  
    Multiply (\*), 109, 110  
    Multiply High (\*\*), 109, 111  
    Multiply Middle (\*//), 109, 112  
    OR (|), 109, 118  
    OR NOT (|/), 109, 120  
    Reverse (REV), 109, 118  
    Shift Left (<<), 109, 117  
    Shift Right (>>), 109, 117  
    Subtract (-), 109, 110  
    XOR (^), 109, 119  
    XOR NOT (^/), 109, 121  
**Binary Radian**, 106, 107, 114  
**BKSP**, 168  
**Board of Education**, 31  
**Bookmarks**, 38, 57  
**Brad**, 106, 107, 114  
**BRANCH**, 133–35, 289, 387  
**BRANCHING**, 133–35, 137–40, 209–12, 213–14, 231–40, 289–90, 331–34, 375–76, 381–85, 387–90, *See also* **BRANCH, IF...THEN, GOTO, GOSUB, ON, RETURN, RUN, POLLRUN, SELECT...CASE**

**BRIGHT**, 467, *See* **XOUT**  
**BS1 Carrier Board**, 30  
**BS1 Serial Adapter**, 27, 30  
**BS1-IC**, 11  
**BS2-IC**, 13  
**BS2p24-IC**, 19  
**BS2p40-IC**, 19  
**BS2px pin configuration function**, 143  
**BS2px voltage comparison function**, 141  
**BS2px24**, 23  
**BS2sx-IC**, 17  
**BSAVE**, 76  
**BUTTON**, 137–40

## — C —

**Cable (Programming)**, 27  
**CASE**. *See* **SELECT...CASE**  
**CGRAM**, 252, 259, 266  
**Character Generator RAM**. *See* **CGRAM**  
**CLRDN**, 168  
**CLREOL**, 168  
**CLS**, 168  
**Colons ( : )**, 213  
**Command Line Interface**, 78–79  
**Command Line Switches**  
    Syntax, 78  
**Comparator**  
    on BS2px, 141  
**COMPARE**, 141–42  
**Comparison Operators**, 232, *See* **Operators, Comparison**  
**Compiler Directives**, 43–46  
    and Conditional Compilation, 70–75  
    *See* **\$STAMP, \$PBASIC, \$PORT**, 43  
**Compile-time Expressions**, 94–96, 98  
**CON Directive**, 94, 99  
**Conditional Compilation**, 70–75  
**Conditional Compile Directives**  
    #DEFINE, 71  
    #ERROR, 75  
    #IF...#THEN, 72



#SELECT...CASE, 74  
**Conditional Logic Operators, 234, 235**  
**CONFIGPIN, 143–46**  
    Logic Threshold mode, 144  
    Output Direction mode, 143  
    Pull-up Resistors mode, 144  
    Schmitt Trigger mode, 145  
**Connecting & Downloading, 29**  
**Connecting/Disconnecting, 25, 27**  
**Constants, 94–96**  
    Defining and Using, 94  
    Operators in Constant Expressions, 96  
**Control Characters, 167**  
**Conversion Formatters, 477–79**  
    DEBUGIN, 173  
    I2CIN, 220  
    I2COUT, 227  
    LCDIN, 260  
    OWIN, 298  
    OWOUT, 306  
    SERIN, 403  
    SEROUT, 422  
**COS, 105, 106**  
**Cosine (COS), 105, 106**  
**COUNT, 149–50**  
**Counter, 191**  
**Counting Pulses, 149–50**  
**CR, 168**  
**CRSRDN, 168**  
**CRSRLF, 168**  
**CRSRRT, 168**  
**CRSRUP, 168**  
**CRSRX, 168**  
**CRSRXY, 168**  
**CRSRY, 168**  
**Current Limit, 12, 14, 15, 18, 20, 21, 23**  
**Cycle Counting, 149**

## — D —

**DATA, 153–58, 459**  
    Writing Block, 155

    Writing Text Strings, 156, 184  
    Writing Word Values, 156, 185  
**Data and Program Downloading, 155, 184**  
**Data Input through Debug Terminal.**  
    *See* DEBUGIN  
**DCD, 105, 106**  
**DDRAM, 252, 258, 259**  
**Debounce Buttons, 137**  
**DEBUG, 166**  
**DEBUG Formatting**  
    (all BS2 models), 162  
    BS1, 160  
**Debug Port, 67**  
**Debug Port Preferences, 67**  
**Debug Terminal, 51–53**  
    Appearance Preferences, 65  
    Buffer Size, 64  
    Choose...Buttons, 63  
    Control Characters, 167  
    Debug Function Preferences, 66  
    Displaying Strings, 166  
    Echo Off, 53  
    Font Size, 63  
    Input Data via Transmit pane. *See* DEBUGIN  
    Port Preferences, 67  
    Receive Pane, 52  
    Repeating Characters, 167  
    Resizing, 63  
    Tabs, 65  
    Text Wrapping, 64  
    Transmit Pane, 52  
**DEBUGIN, 159, 171–73**  
    Special Formatters, 172  
**DEC, 162, 163, 165, 173, 220, 227, 260, 265, 298, 306, 398, 403, 420, 422**  
**Decimal Notation, 96**  
**Decoder (DCD), 105, 106**  
**Default Com Port, 60**  
**Default State, I/O Pins, 83**  
**Default State, I/O Pins (BS1), 82**

# Index

---

Defining Constants, 94–96  
Defining Symbols, 85–91  
Delaying Execution, 311  
Developer Features, 75  
DIG, 109, 117  
Digit (DIG), 109, 117  
Digital I/O, 129, 137, 143, 149, 215, 217, 225, 243, 247, 249, 257, 263, 281, 283, 293, 295, 303, 313, 325, 377, 393, 415, 431, 435, 455, 465  
DIM, 467, *See* XOUT  
Direction (setting of I/O pins), 81, 83, 243, 293, 377  
Directives. *See also* Conditional Compile Directives, *See also* \$PBASIC, \$PORT, \$STAMP, *See* CON, PIN, VAR, DATA  
Directory List, 40, 41  
DIRS, 83  
DIRS (BS1), 81  
Display Data RAM. *See* DDRAM  
Displaying Information, 159  
Displaying Unsigned Numbers, 164  
Divide (/), 109, 113  
Division Remainder, 109, 113  
DO...LOOP, 175–77, 189  
Download Function, 48–50  
Download Time, Saving, 60  
DS1820 Temperature Sensor, 301, 308  
DTMFOUT, 179–82

## — E —

Echo, 47  
Echo Off, 53  
Edit Preferences  
    Debug Function, 66  
    Debug Port, 67  
    Debug Terminal Appearances, 63–65  
    Editor Appearance, 55–57  
    Editor Operation, 57–60  
    Files and Directories, 60–63  
Editing your Code, 39

Editor  
    Bookmarks, 38  
    Described, 35  
    Editor Screen (diagram), 35  
    Editor Tabs, 35  
    Find/Replace Function, 39  
    Font Size, 56  
    Line Numbers, 38  
    Split Window View, 36  
    Status, 36  
    Syntax Highlighting, 37  
    Windows, 35  
Editor Appearance  
    Font Size, 56  
    Syntax Highlighting, 56  
Editor Functions  
    Command Line Interface, 78–79  
    Debug Terminal, 51–53  
    Download Function, 48–50  
    Help Files, 54  
    Identification, 46–48  
    Keyboard Shortcuts, 53  
    Memory Map, 50–51  
    Tip of the Day, 55  
EEPROM, 183–86, 459  
EEPROM ACCESS. *See* EEPROM, DATA, READ, WRITE, STORE  
EEPROM Map, 51  
EEPROM Usage, 153, 369–73, 447  
    Maximizing, 209  
EEPROM Usage (BS1), 183  
ELSE. *See* SELECT...CASE, *See* IF...THEN  
ELSEIF, 231, *See* IF...THEN  
Encoder (NCD), 105, 107  
END, 187, 335, 447  
ENDIF. *See* IF...THEN  
ENDSELECT. *See* SELECT...CASE  
EXIT, 189  
Expressions, 94–96  
Expressions in Constants, 96

## — F —

- Favorite Directories**, 63
- Features for Developers**, 75
- File Associations**, 42, 61
- File List**, 40, 41
- File Management**
  - .obj file, 76
  - Backup Copy, 61
  - Directory List, 41
  - Favorite Directories, 63
  - File Associations, 42, 61
  - File List, 41
  - Files and Directories Preferences, 63
  - Filter List, 40, 41
  - Initial Directory, 62
  - Keyboard Shortcuts, 42
  - Module Directories, 62
  - Open From, 41
  - Recent List, 40
  - Save To, 41
  - Single Executable File, 76
  - Templates, 62
- Filter List**, 40, 41
- Find/Replace Function**, 39
- Firmware**, 3
- Fixed plus Smart Tabs**, 59
- Fixed Tabs**, 58
- Flow Control**, 409, 423
- Font Size**
  - Debug Terminal, 63
  - Editor Pane, 56
- FOR...NEXT**, 189
  - Increment/Decrement, 193
  - Variables as Arguments, 194
- FOR...NEXT**, 191–97
- Formatters, Conversion. See**  
    **Conversion Formatters**
- Formatters, DEBUG. See** DEBUG
- Formatters**
- Formatters, Special. See** Special
- Formatters**
- FPin**, 409, 423

**FREQOUT**, 199–201

## — G —

- Generating Pulses**, 347–49
- Generating Random Numbers**, 359–61
- Generating Sound (BS1)**, 445–46
- Generating Sound (Non-BS1)**, 199–201
- GET**, 203–6
- GOSUB**, 209–12, 289, 375
- GOTO**, 209, 213–14, 213, 289
- GUI Interface Development**, 78
- Guidelines and Precautions**, 25

## — H —

- Hardware**
  - BASIC Stamp, 7
  - BS1, 10
  - BS2, 13
  - BS2e, 15
  - BS2p, 19
  - BS2pe, 21
  - BS2px, 23
  - BS2sx, 17
- Help Files**, 53–54
- HEX**, 162, 163, 173, 220, 227, 260, 265, 298, 306, 403, 422
- Hex to BCD Conversion**, 97
- Hexadecimal Notation**, 96
- HIGH**, 215–16, 281, 455
- Hitachi 44780 Controller**, 249, 258, 263
- HOME**, 168
- HYP**, 109, 115
- Hypotenuse (HYP)**, 109, 115

## — I —

- I/O pin**
  - Voltage comparator (BS2px), 141
- I/O pin properties (BS2px)**, 143
- I/O Pins**

# Index

---

Accessing, 81, 83, 84  
Auxiliary, 129, 247  
Default State, 83  
Default State (BS1), 82  
Defining with PIN, 99  
Directions, 81, 83, 243, 293, 377  
Main, 247, 283  
PIN Directive, 99–102  
Pinouts  
    BS1, 12  
    BS2, 14  
    BS2e, 15  
    BS2p, 20  
    BS2pe, 21  
    BS2px, 23  
    BS2sx, 18  
Reading, 84  
Reading (BS1), 81  
Source/Sink  
    BS1, 12  
    BS2, 14  
    BS2e, 15  
    BS2p, 20  
    BS2pe, 21  
    BS2px, 23  
    BS2sx, 18  
Writing (BS1), 81  
**I<sup>2</sup>C Protocol, 217–23, 225–30**  
**I2CIN, 217–23**  
**I2COUT, 225–30**  
**IBIN, 163, 173, 220, 227, 260, 265, 298, 306, 403, 422**  
**Identification Function, 46–48**  
**IF...THEN, 231–40, 387**  
    Nested, 238  
    Single vs Multi-Line Syntax, 238  
**IF...THEN, 133**  
**Ignore BS1 Modules, 60**  
**IHEX, 163, 173, 220, 227, 260, 265, 298, 306, 403, 422**  
**Indent**  
    Preferences, 57  
**Initial Directory on Startup, 62**

**INPUT, 243–44, 293, 377**  
**INS, 83**  
**Integer Math Rules, 103, 104**  
**Integrated Explorer Panel, 40–42**  
    Resizing, 42  
**Interface to Telephone Line, 181**  
**Interrupts**  
    Not supported by BASIC Stamp, 331  
**Inverse (~), 105, 106**  
**IOTERM, 129, 247–48, 283**  
**ISBIN, 163, 173, 220, 227, 260, 265, 298, 306, 403, 422**  
**ISHEX, 163, 173, 220, 227, 260, 265, 298, 306, 403, 422**

## — K —

**Keyboard Shortcuts**  
    Coding Functions, 53  
    Editing and Navigation, 39  
    Editor Functions, 53  
    File Functions, 42

## — L —

**Labels, 213**  
**Language**  
    PBASIC, 7  
**LCD Commands, 251**  
**LCD Customer Characters, 266**  
**LCD Displays, 249, 257, 263**  
**LCD Initialization, 251, 258, 264**  
**LCDCMD, 249–56**  
**LCDIN, 257–61**  
**LCDOUT, 263–70**  
**LET, 269**  
**LF, 168**  
**LIGHTSON, 467, See XOUT**  
**Line Numbering, 57**  
**Line Numbers, 38**  
**Logic**  
    Run-time, 96

**Logic Level, TTL and CMOS**  
(diagram), 144

**Logic Operators**

AND, 235

NOT, 235

OR, 235

XOR, 235

**Logic Threshold**

and BS2px, 143

CMOS, 144

TTL, 144

**Logic Truth Tables, 235**

**LOOKDOWN, 133, 271–76**

**LOOKUP, 271, 277–80**

**LOOP. *See* DO...LOOP**

**Loopback, 47**

**Loops, 175–77, 191–97, 213**

Conditional, 175–77

Terminating, 189

**LOW, 281–82, 455**

**Low-Power Mode, 187, 285, 441–42**

**LSBFIRST, 436**

**LSBPOST, 432**

**LSBPRES, 432**

## — M —

**Main I/O Pins, 247, 283**

**MAINIO, 129, 247, 283–84**

**Mapping Non-Contiguous Numbers,**  
275, 279

**Math**

Integer Math Rules, 103

Run-time, 96

**MAX, 109, 116**

**Maximum (MAX), 109, 116**

**Measuring Pulses, 343–45**

**Measuring Variable Resistance, 339–**  
40, 363–68

**Memory, 29, 81**

**Memory Map, 44, 50–51, 91**

(Diagram), 50

ASCII Format, 51

EEPROM Map, 51

Fixed Variables, 51

Hexadecimal Format, 51

RAM Map, 50

**Memory Organization, 81**

**MIN, 109, 115**

**Minimum (MIN), 109, 115**

**Modifiers, 89–91**

**Module Directories, 62**

**Modulus (/), 109, 113**

**MSBFIRST, 436**

**MSBPOST, 432**

**MSBPRES, 432**

**Multi-file Projects**

Project Download Modes, 70

**Multi-File Projects, 68–70**

\$STAMP Directive, 68

Creating, 69

**Multiply (\*), 109, 110**

**Multiply High (\*\*), 109, 111**

**Multiply Middle (\* /), 109, 112**

## — N —

**NAP, 285–87, 335**

**NCD, 105, 107**

**Negative (-), 105, 106**

**Networking BASIC Stamps, 425**

**NEXT. *See* FOR...NEXT**

**Non-volatile Storage, 153**

**NOT, 235**

**NUM, 173, 220, 260, 298, 403**

**Numbering Systems, 94, 96**

**Numbers, 94, 96**

**Numbers, Random. *See* RANDOM**

**NUMERICS. *See* LET, LOOKUP,**  
LOOKDOWN, RANDOM

## — O —

**OEMBS1, 11**

**OEMBS2, 13**

# Index

---

**OEMBS2sx**, 17  
**ON**, 289–90, 375, 387  
**Open Baud Modes**, 425  
**Open From**, 41  
**Open With**, 62  
**Operation, General Theory**, 7  
**Operators**  
    Binary, 104, 109–21  
    Comaprison (SELECT...CASE), 388  
    Comparison (IF...THEN), 232  
    Comparison (LOOKDOWN), 273  
    Conditional Logic, 234  
    in Constant Expressions, 96  
    Unary, 104, 105–9  
**OR**, 235  
**OR (|)**, 109, 118  
**OR NOT (|/)**, 109, 120  
**Order of Operations**, 102–3  
**Orientation**, 25  
**OUTPUT**, 243, 293–94, 377  
**Output Direction**  
    and BS2px, 143  
**OUTS**, 83  
**OWIN**, 295–301  
**OWOUT**, 295, 303–10

## — P —

**P0-P15**, 14, 15, 18, 20, 21, 23  
**P0-P7**, 12  
**Pace**, 421  
**Pacing**, 415  
**Package Types**  
    BS1, 10  
    BS2, 13  
    BS2e, 15  
    BS2p, 19  
    BS2pe, 21  
    BS2px, 23  
    BS2sx, 17  
**PARALLEL I/O. See** LCDCMD, LCDIN,  
    LCDOUT, *See* LCDCMD, LCDIN,  
    LCDOUT

**Parentheses**  
    and Order of Evaluation, 103, 235  
    Nested, 103  
    with expressions, 128  
**Parity and Parity Errors**, 407, 423  
**Parity and Timeout**, 408  
**PAUSE**, 311  
**PBASIC Language**, 7  
**PBASIC Language Versions**, 124  
    # of commands, 45  
    PBASIC 2.5 Enhancements., 124  
    Reserved Words, 473  
**PCI**, 12  
**PCO**, 12  
**PIN Directive**, 99–102  
**Pin-1 Indicators**, 25  
**PinMask**, 143  
**Pinouts**  
    BS1, 10  
    BS2, 13  
    BS2e, 15  
    BS2p, 19  
    BS2pe, 21  
    BS2px, 23  
    BS2sx, 17  
**PINS**, 81  
**POLLIN**, 313–17, 319, 331, 335  
**POLLMODE**, 313, 319–23, 331, 335  
**POLLOUT**, 313, 319, 325–28, 331, 335  
**POLLRUN**, 313, 319, 331–34, 335, 381  
**POLLWAIT**, 313, 319, 331, 335–37  
**PORT**, 81  
**Port Communication**  
    Debug Terminal Preferences, 67  
    Default Com Port, 60  
**Port Connection. See** Identify Function  
**POT**, 339–40  
**Potentiometer**, 313–17, 363–68  
**POWER CONTROL**, 187, 285–87, 335–  
    37, 441–42, *See* END, NAP, SLEEP  
**Power Supply**  
    BS1, 12  
    BS2, 14

BS2e, 15  
BS2p, 20  
BS2pe, 21  
BS2px, 23  
BS2sx, 18  
**Powerline Interface**, 466  
**Program Debugging**. *See* **DEBUG**,  
**DEBUGIN**  
**Program Slot**, 68, 93, 203, 316, 320,  
331, 351, 372, 381, 449, 459  
**Programming Connections**, 27  
**Programming Environment**, 35  
**Projects**. *See* **Multi-file Projects**  
**Protocol**  
1-Wire, Dallas, 295–301, 303–10  
Asynchronous RS-232, 393–412,  
415–28  
I<sup>2</sup>C, 217–23, 225–30  
Synchronous (SPI), 431–34, 435–40  
X10, 465–68  
**Pull-up Resistor**  
and BS2px, 143  
**Pull-up resistors (BS2px)**, 144  
**Pulse Width Modulation**, 355–58  
**Pulses, Generating**, 347–49  
**Pulses, Measuring**, 343–45  
**PULSIN**, 343–45  
**PULSOUT**, 347–49  
**PUT**, 351–52  
**PWM**, 355–58

## — Q —

**Quick Start Guide**, 29

## — R —

**Radian, Binary**, 106, 107, 114  
**RAM ACCESS**. *See* **GET**, **PUT**  
**RAM Map**, 50  
**RAM Organization**  
BS1, 81

BS2, 82  
BS2e, 82  
BS2p, 82  
BS2p40, 82  
BS2pe, 82  
BS2sx, 82  
**RANDOM**, 359–61  
**Random Numbers**, 359–61  
**RCTIME**, 363–68  
**RCTime Equation**, 365  
**READ**, 153, 369–73, 449, 459  
**Reading Potentiometers**, 339–40, 363–  
68  
**Reading Pulses**, 149–50, 343–45  
**Reading Word Values**, 370  
**Receive Pane**, 52  
**Recent List**, 40  
**Reference Notch**, 26  
**Registers**  
Memory Map, 50  
**Remainder of Division**, 109, 113  
**REP**, 163, 167, 228, 305, 422  
**RES**, 12, 14, 15, 18, 20, 21, 23  
**Reserved Words**  
additional  
BS2e and BS2sx, 475  
BS2p and BS2pe, 475  
all BASIC Stamp 2 models, 474  
BASIC Stamp 1, 473  
**Reserving EEPROM Locations**, 155  
**Reset**, 12, 14, 15, 18, 20, 21, 23  
**RETURN**, 375–76  
**REV**, 109, 118  
**REVERSE**, 243, 293, 377–78  
**Reverse (REV)**, 109, 118  
**Rules of Integer Math**, 103, 104  
**Rules of Symbols**, 86  
**RUN**, 331, 381–85  
**Run-time**, 98  
**Run-time Math and Logic**, 96

# Index

---

## — S —

- Save To, 41
- SBIN, 163, 173, 220, 227, 260, 265, 298, 306, 403, 422
- Schematic
  - BS1, 481
  - BS2, 482
  - BS2e, 483
  - BS2p24, 485
  - BS2p40, 486
  - BS2pe, 487
  - BS2px, 488
  - BS2sx, 484
- Schmitt Trigger, 143, 145, 150
  - (diagram), 145
- Scratch Pad Ram
  - Registers, 93
- Scratch Pad RAM, 92, 203, 351–52
  - Registers, 205
  - Special Purpose Locations (POLLMODE), 323
- SDEC, 163, 173, 220, 227, 260, 265, 298, 306, 403, 422
- SELECT...CASE, 387–90
- SELECT...CASE, 387
- Serial Port Diagram, 395
- Serial Timeout, 408, 425
- Serial Troubleshooting, 410, 427
- SERIN, 171, 393–412
- SEROUT, 415–28
- SHEX, 163, 173, 220, 227, 260, 265, 298, 306, 403, 422
- Shift Left (<<), 109, 117
- Shift Right (>>), 109, 117
- SHIFTIN, 431–34
- SHIFTOUT, 435–40
- Shortcuts. *See* Keyboard Shortcuts
- SIN, 105, 107
- SIN (pin), 14, 15, 18, 20, 21, 23
- Sine (SIN), 105, 107
- Single Executable File, 76
- SKIP, 172, 219, 259, 297, 404
- SLEEP, 187, 335, 441–42
- SNUM, 173, 220, 260, 298, 403
- SOUND. *See also* SOUND, FREQOUT, DTMFOUT
- SOUND, 445–46
- Sound, Generation (BS1), 445–46
- Sound, Generation (Non-BS1), 199–201
- SOUT, 14, 15, 18, 20, 21, 23
- Speaker, 180, 200, 446
- Special Formatters
  - DEBUGIN, 172
  - I2CIN, 219
  - I2COUT, 228
  - LCDIN, 259
  - OWIN, 297
  - OWOUT, 305
  - SERIN, 404
  - SEROUT, 422
- Split Window View, 36
- SPRAM. *See* Scratch Pad RAM
- SPSTR, 219, 297, 404
- SPSTR L, 172
- SQR, 105, 108
- Square Root (SQR), 105, 108
- STAMP Directive. *See* \$STAMP Directive
- StampLoader.exe program, 76
- Static Sensitive Devices, 25
- STEP. *See* FOR...NEXT
- STOP, 447
- STORE, 449, 459
- STR, 163, 166, 172, 219, 228, 259, 297, 305, 404, 422
- Strings
  - Displaying, 166
- Subroutines, 209, 375
- Subtract (-), 109, 110
- Switching Program Slots, 381–85
- Symbol Name Rules, 86
- Symbols (Characters). *See* +
  - #, 161
  - \$, 161
  - %, 161



- &, 118
- &/, 120
- \*, 110
- \*, 109
- \*\*, 109, 111
- \*/, 109, 112
- /, 109, 113
- //, 109, 113
- ?, 163, 165, 228, 305, 422
- @, 154, 161
- ^, 119
- ^/, 121
- |, 118
- |/, 120
- ~, 105, 106
- +, 96, 109
- <, 232
- <<, 117
- <=, 232
- <>, 232
- =, 232
- >, 232
- >=, 232
- >>, 117
- SYNCHRONOUS SERIAL, 431–34, 435–40, *See also* SHIFTIN, SHIFTOUT< I2CIN, I2COUT**
- Syntax Conventions, 128**
- Syntax Enhancements for PBASIC 2.5, 124**
- Syntax Highlighting, 37, 56**
  - Customized, 57
  - PBASIC versions, 45
- T —**
- TAB, 168**
- Tables, 153–58, 183–86, 271–76, 277–80**
- Tabs**
  - (diagram), 59
  - Character, 57
  - Fixed plus Smart Tabs, 59
  - Fixed Tab Positions List, 60
  - Fixed Tabs, 58
  - in Debug Terminal, 65
  - Smart Tabs, 58
  - Tab Behavior, 58–59
- Telephone Touch Tones, 179**
- Templates, 62**
- Text Wrapping**
  - Debug Terminal, 64
- Theory of Operation, 7**
- TIME. *See* PAUSE, POLLWAIT**
- Timeout, 393, 408, 415, 425**
- Tip of the Day, 55**
- TO. *See* FOR...NEXT**
- TOGGLE, 281, 455–57**
- Tone Generation, 179–82, 199–201, 445–46**
- Transmit Pane, 52**
- Troubleshooting Serial, 410, 427**
- Truth Table**
  - IF...THEN, 235
  - POLLIN, 316
  - POLLOUT, 327
- Two's Complement, 104**
- U —**
- Unary Operators, 104, 105–9**
  - Absolute Value (ABS), 105
  - Cosine (COS), 105, 106
  - Decoder (DCD), 105, 106
  - Encoder (NCD), 105, 107
  - Inverse (~), 105, 106
  - Negative (-), 105, 106
  - Sine (SIN), 105, 107
  - Square Root (SQR), 105, 108
- Unit Circle, 107, 114**
- UNITOFF, 467, *See* XOUT**
- UNITON, 467, *See* XOUT**
- UNITSONf, 467, *See* XOUT**
- UNTIL. *See* DO...LOOP**
- Untitled#, 36**
- USB Port**

# ***Index***

---

and Identify Function, 48

## **— V —**

**Variable Resistance, Measuring, 339–40, 363–68**

**Variables**

Aliases, 89–91

Arrays, 87–89

Defining, 85–91, 269

Fixed, 84

Modifiers, 89–91

Sizes, 86

**VDD, 12, 14, 15, 18, 20, 21, 23**

**Versions, 3**

**VIN, 12, 14, 15, 18, 20, 21, 23**

**Voltage comparison function, of BS2px, 141**

**VSS, 12, 14, 15, 18, 20, 21, 23**

## **— W —**

**W0-W6, 82**

**WAIT, 172, 259, 401, 404**

**WAITSTR, 172, 219, 259, 297, 404, 406**

**Warranty, 2**

**WHILE. *See* DO...LOOP**

**WRITE, 153, 449, 459–63**

## **— X —**

**X0-X15, 20**

**X10 Control, 465–68**

**XOR, 235**

**XOR (^), 109, 119**

**XOR NOT (^/), 109, 121**

**XOUT, 465–68**

**XOUT Command Codes (Table), 467**

# **BASIC STAMP FAQ**

**(FREQUENTLY ASKED QUESTIONS AND ANSWERS)**

---

## GENERAL INFORMATION 4

|                                                                                                                    |   |
|--------------------------------------------------------------------------------------------------------------------|---|
| WHAT IS THE BASIC STAMP?                                                                                           | 4 |
| DOES THE BASIC STAMP LOSE ITS PROGRAM WHEN I REMOVE THE BATTERY OR POWER SUPPLY?                                   | 4 |
| DOES THE BASIC STAMP COME IN DIFFERENT VERSIONS?                                                                   | 5 |
| WHAT IS THE DIFFERENCE BETWEEN THE BASIC STAMP REV. D (SOMETIMES JUST CALLED “BASIC STAMP”) AND THE BASIC STAMP I? | 5 |
| HOW BIG IS THE BASIC STAMP?                                                                                        | 6 |
| HOW BIG ARE THE BASIC STAMP I AND BASIC STAMP II CARRIER BOARDS?                                                   | 6 |
| HOW DO I POWER THE BASIC STAMP?                                                                                    | 6 |
| HOW MUCH CURRENT DOES THE BASIC STAMP CONSUME?                                                                     | 6 |
| IS THE BASIC STAMP SENSITIVE TO STATIC ELECTRICITY?                                                                | 6 |
| ARE THERE ANY PRODUCTION POSSIBILITIES WITH THE BASIC STAMP?                                                       | 7 |
| WHAT ITEMS DO I NEED TO GET STARTED WITH THE BASIC STAMP?                                                          | 7 |
| HOW CAN I MAKE MY OWN BASIC STAMP CIRCUIT?                                                                         | 7 |
| WHAT KIND OF MICROCHIPS ARE USED IN THE BASIC STAMP?                                                               | 7 |

---

## FUNCTIONAL INFORMATION 7

|                                                                                                                                 |    |
|---------------------------------------------------------------------------------------------------------------------------------|----|
| WHAT KIND OF OPERATING ENVIRONMENT DOES THE BASIC STAMP REQUIRE?                                                                | 7  |
| WHAT ARE THE MAIN DIFFERENCES BETWEEN THE BASIC STAMPS?                                                                         | 8  |
| IS THE BASIC STAMP II BETTER THAN THE BASIC STAMP I?                                                                            | 8  |
| IS THE BASIC STAMP IISX BETTER THAN THE BASIC STAMP II?                                                                         | 8  |
| HOW MANY I/O PINS DOES THE BASIC STAMP HAVE?                                                                                    | 9  |
| WHAT CAN I USE THE I/O PINS FOR?                                                                                                | 9  |
| CAN THE I/O PINS BE USED TO CONTROL RELAYS, SOLENOIDS AND OTHER SIMILAR DEVICES?                                                | 9  |
| CAN I CONTROL LEDs WITH THE I/O PINS?                                                                                           | 9  |
| HOW MUCH CURRENT CAN THE I/O PINS HANDLE?                                                                                       | 9  |
| WHAT IS THE INPUT IMPEDANCE OF THE BASIC STAMP’S I/O PINS?                                                                      | 9  |
| HOW FAST DOES THE BASIC STAMP EXECUTE ITS PBASIC CODE?                                                                          | 9  |
| WHAT IS THE VIN PIN USED FOR?                                                                                                   | 10 |
| HOW DOES THE VDD PIN WORK?                                                                                                      | 10 |
| HOW DOES THE RESET PIN (RES) WORK?                                                                                              | 10 |
| HOW DOES THE ATN (ATTENTION) PIN WORK (BASIC STAMP II, BASIC STAMP IIE AND BASIC STAMP IISX)?                                   | 10 |
| HOW MUCH CURRENT CAN THE BASIC STAMP’S ON-BOARD REGULATOR PROVIDE?                                                              | 11 |
| CAN THE BASIC STAMP GENERATE A SINE WAVE?                                                                                       | 11 |
| CAN THE BASIC STAMP GENERATE A SINE WAVE ON TWO PINS AT ONCE?                                                                   | 11 |
| CAN THE BASIC STAMP GENERATE MORE THAN ONE SINE WAVE ON A PIN AT THE SAME TIME?                                                 | 11 |
| CAN THE BASIC STAMP GENERATE SQUARE WAVES?                                                                                      | 11 |
| DOES THE BASIC STAMP INCLUDE A REAL TIME CLOCK FUNCTION?                                                                        | 11 |
| DOES THE BASIC STAMP HAVE A BUILT-IN TIMER?                                                                                     | 11 |
| DOES THE BASIC STAMP SUPPORT INTERRUPTS?                                                                                        | 12 |
| DOES THE BASIC STAMP II, BASIC STAMP IIE OR BASIC STAMP IISX HAVE THE SAME BSAVE/BSLOAD FEATURE AVAILABLE IN THE BASIC STAMP I? | 12 |

|                                                                                                                                                                      |           |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|
| <b>PROGRAMMING INFORMATION</b>                                                                                                                                       | <b>12</b> |
| <b>HOW DO I PROGRAM THE BASIC STAMP?</b>                                                                                                                             | <b>12</b> |
| <b>HOW DO I CONNECT THE BASIC STAMP TO MY COMPUTER FOR PROGRAMMING?</b>                                                                                              | <b>12</b> |
| <b>HOW DOES MY PROGRAM GET STORED IN THE BASIC STAMP?</b>                                                                                                            | <b>12</b> |
| <b>HOW DO I ERASE THE BASIC STAMP'S PROGRAM SPACE?</b>                                                                                                               | <b>12</b> |
| <b>HOW DO I REPROGRAM THE BASIC STAMP?</b>                                                                                                                           | <b>13</b> |
| <b>HOW BIG OF A PROGRAM CAN I STORE IN THE BASIC STAMP?</b>                                                                                                          | <b>13</b> |
| <b>CAN I EXPAND THE PROGRAM MEMORY?</b>                                                                                                                              | <b>13</b> |
| <b>CAN I EXPAND THE DATA MEMORY?</b>                                                                                                                                 | <b>13</b> |
| <b>HOW DIFFICULT IS IT TO PROGRAM THE BASIC STAMP?</b>                                                                                                               | <b>13</b> |
| <b>CAN I PROGRAM THE BASIC STAMP IN VISUAL BASIC OR QBASIC?</b>                                                                                                      | <b>13</b> |
| <b>WHAT IS PBASIC?</b>                                                                                                                                               | <b>13</b> |
| <b>CAN I IMBED ASSEMBLY LANGUAGE ROUTINES IN MY PBASIC CODE?</b>                                                                                                     | <b>13</b> |
| <b>HOW MUCH SPACE DOES EACH PBASIC COMMAND TAKE?</b>                                                                                                                 | <b>14</b> |
| <b>HOW DO I KNOW HOW MUCH SPACE MY PBASIC PROGRAM CONSUMES ?</b>                                                                                                     | <b>14</b> |
| <b>WHY DOESN'T THE MEMORY MAP FEATURE SHOW ME DATA I HAVE ALREADY STORED IN THE BASIC STAMP II, IIE OR IISX'S EEPROM?</b>                                            | <b>14</b> |
| <b>HOW DO I DEBUG MY PBASIC PROGRAMS ?</b>                                                                                                                           | <b>14</b> |
| <b>HOW DO I PRINT OUT MY PBASIC PROGRAMS ?</b>                                                                                                                       | <b>15</b> |
| <b>CAN I READ-OUT, OR UPLOAD, THE PBASIC PROGRAM WHICH IS ALREADY STORED IN THE BASIC STAMP?</b>                                                                     | <b>15</b> |
| <b>SINCE THE BASIC STAMP II, IIE AND IISX USE THE SERIAL PORT FOR PROGRAMMING, CAN I USE A SIMPLE TERMINAL PROGRAM TO DOWNLOAD MY PBASIC CODE?</b>                   | <b>15</b> |
| <b>WILL A PROGRAM AND CIRCUITRY DESIGNED FOR THE BASIC STAMP I WORK WITH THE BASIC STAMP II, IIE OR IISX?</b>                                                        | <b>15</b> |
| <b>CAN I PROGRAM THE BASIC STAMP II, IIE OR IISX WITH THE BASIC STAMP I EDITOR (STAMP.EXE)?</b>                                                                      | <b>15</b> |
| <b>HOW DO I MAKE MY PBASIC PROGRAM START OVER OR CONTINUE RUNNING FOREVER?</b>                                                                                       | <b>15</b> |
| <b>HOW DO I SET AN I/O PIN TO INPUT OR OUTPUT MODE?</b>                                                                                                              | <b>16</b> |
| <b>HOW DO I MAKE AN I/O PIN OUTPUT A HIGH OR A LOW?</b>                                                                                                              | <b>17</b> |
| <b>HOW DO I INPUT OR OUTPUT DATA IN PARALLEL ON THE I/O PINS?</b>                                                                                                    | <b>18</b> |
| <b>WHY DO THE BASIC STAMP'S OUTPUT PINS TOGGLE BRIEFLY WHEN I USE THE SLEEP, NAP OR END COMMANDS OR ALLOW MY PROGRAM TO END NATURALLY?</b>                           | <b>18</b> |
| <b>THE LET COMMAND IS NOT AVAILABLE ON THE BASIC STAMP II, IIE AND IISX. DOES THIS MEAN THE BASIC STAMP II, IIE AND IISX CANNOT PERFORM MATHEMATICAL OPERATIONS?</b> | <b>19</b> |
| <b>CAN THE BASIC STAMP STORE DATA, SUCH AS TEMPERATURE READINGS, FOR REVIEW OR USE AT A LATER TIME?</b>                                                              | <b>19</b> |
| <b>HOW CAN I STORE A WORD VALUE INTO THE INTERNAL EEPROM?</b>                                                                                                        | <b>19</b> |
| <b>HOW DOES THE FPIN (FLOW CONTROL PIN) WORK IN THE SERIN AND SEROUT COMMANDS ON THE BASIC STAMP II, IIE AND IISX?</b>                                               | <b>20</b> |
| <b>HOW ARE ARITHMETIC EXPRESSIONS EVALUATED WITHIN THE BASIC STAMP?</b>                                                                                              | <b>20</b> |
| <b>DOES THE BASIC STAMP HANDLE SIGNED NUMBERS AND ARITHMETIC?</b>                                                                                                    | <b>20</b> |
| <b>HOW CAN I DEFINE AN ALIAS TO AN I/O PIN OR ANOTHER VARIABLE?</b>                                                                                                  | <b>21</b> |
| <b>HOW DO I REFERENCE A SPECIFIC BIT WITHIN A BYTE OR WORD VARIABLE?</b>                                                                                             | <b>21</b> |
| <b>HOW DO I DEFINE A STRING VARIABLE?</b>                                                                                                                            | <b>21</b> |
| <b>HOW DO I DEFINE ALIASES TO SPECIFIC BYTES WITHIN A WORD ARRAY ON THE BASIC STAMP II, IIE AND IISX?</b>                                                            | <b>22</b> |

## **What is the BASIC Stamp?**

The BASIC Stamp is a microcontroller developed by Parallax, Inc. which is easily programmed using a form of the BASIC programming language. It is called a “Stamp” simply because it is close to the size of an average postage stamp.

## **Does the BASIC Stamp lose its program when I remove the battery or power supply?**

No, your PBASIC code is stored inside a serial EEPROM on-board the BASIC Stamp. EEPROMs provide non-volatile storage; they retain memory even without power. The EEPROM used in the BASIC Stamp 1 and 2 is guaranteed to function properly for 40 years and for 10,000,000 write cycles per memory location. The EEPROM used in the BASIC Stamp 2e and 2sx is guaranteed for 1,000,000 write cycles per memory location.

## Does the BASIC Stamp come in different versions?

Yes. Currently there are four functional versions of the BASIC Stamp and seven physical versions. The BASIC Stamp line consists of the BASIC Stamp I, the BASIC Stamp II, the BASIC Stamp IIe and the BASIC Stamp IIsx. The BASIC Stamp I is available in three package types (physical versions) as shown in figures 1a, 1b and 1c below. The BASIC Stamp 1 Rev. D is a through-hole, socketed package (with prototyping area). The BS1-IC is a 14-pin SIP with surface-mounted components. The OEMBS1 is a 14-pin SIP in a larger board layout meant for OEM use (for recreating the entire BASIC Stamp 1 circuitry). It's important to note the three versions are functionally equivalent (with the exception of the lack of a Reset pin on the BASIC Stamp 1 Rev. D) but are just in physically different packages. The BASIC Stamp II, BASIC Stamp IIe and BASIC Stamp IIsx are available in a 24-pin DIP, with surface-mounted components (see Figures 2a, 3 and 4. They are called the BS2-IC, BS2E-IC and BS2SX-IC, respectively. The BS2-IC is also available in an OEM version (Figure 2b) that is a 20-pin SIP package.

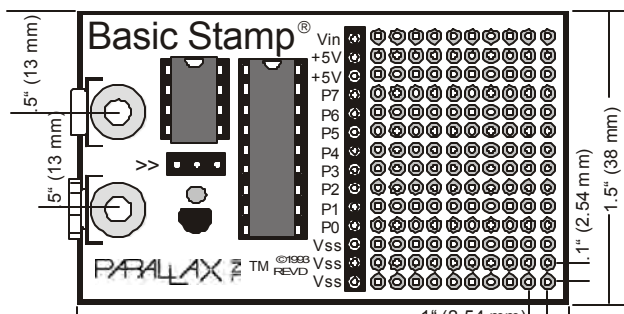


Figure 1a –BASIC Stamp 1 Rev. D

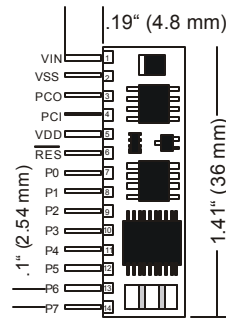


Figure 1b –BS1-IC

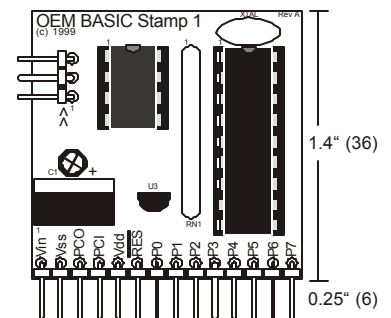


Figure 1c –OEMBS1

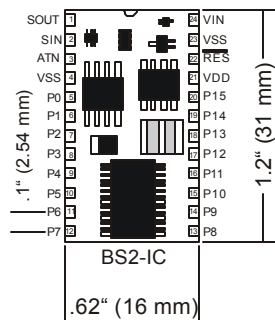


Figure 2a – BS2-IC

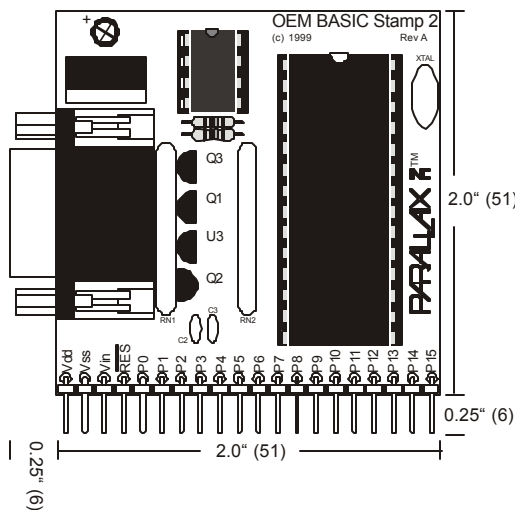


Figure 2b – OEMBS2

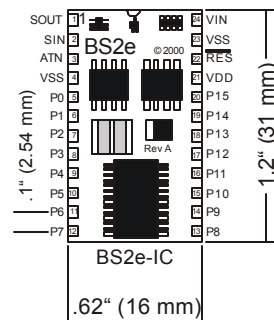


Figure 3 – BS2e-IC

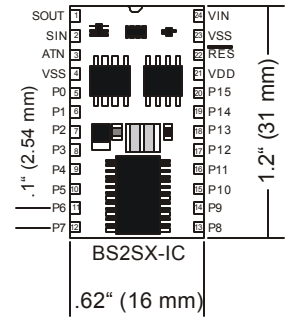


Figure 4 – BS2sx-IC

## What is the difference between the BASIC Stamp Rev. D (sometimes just called “BASIC

The original version of the BASIC Stamp I was simply called, “BASIC Stamp”. That name was sufficient until the arrival of the new physical package type, as well as the second model in the Stamp line, the BASIC Stamp II. At that point, the original version was split into two package types, the Rev. D and the BS1-IC; both of which are considered to be the “BASIC Stamp I”. The “BASIC Stamp II” is known as the BS2-IC, the “BASIC Stamp IIe” is known as the BS2E-IC, and the “BASIC Stamp IIsx” is known as the BS2SX-IC. Today, the term “BASIC Stamp” is too generic and really could be applied to any one of the units in the product line. Long-time Parallax customers, however, still frequently refer to the BASIC Stamp I as the BASIC Stamp, thus, in many cases, there is no difference between the term BASIC Stamp and the BASIC Stamp I. To keep things clear, we prefer the names: “BASIC Stamp I”, “BASIC Stamp II”, “BASIC Stamp IIe”, and “BASIC Stamp IIsx”, or more specifically, “-IC”, “BS2-IC”, “BS2E-IC” and “BS2SX-IC” and try to use the term “BASIC Stamp” to refer to the entire line of microcontrollers.

## How big is the BASIC Stamp?

The BASIC Stamp rev. D measures 2.5” (65 mm) L x 1.52” (39 mm) W x 0.4” (10 mm) D.

The BS1-IC measures 1.4” (35 mm) L x 0.58” (15 mm) W x 0.13” (3 mm) D.

The BS2-IC measures 1.2” (30 mm) L x 0.62” (16 mm) W x 0.35” (9 mm) D.

The BS2E-IC measures 1.2” (30 mm) L x 0.62” (16 mm) W x 0.35” (9 mm) D.

The BS2SX-IC measures 1.2” (30 mm) L x 0.62” (16 mm) W x 0.35” (9 mm) D.

## How big are the BASIC Stamp I and BASIC Stamp II Carrier Boards?

The BASIC Stamp I Carrier Board measures 2.5” (65 mm) L x 1.5” (38 mm) W x 0.5” (13 mm) D

The BASIC Stamp II Carrier Board measures 2.8” (71 mm) L x 3.1” (79 mm) W x 0.6” (15 mm) D.

## How do I power the BASIC Stamp?

The BASIC Stamp runs on 5 to 15 volts DC. All BASIC Stamps feature an on-board 5-volt regulator which will convert an input 6 to 15 volts (on the VIN pin) down to the 5 volts that its components require. If your power supply is 6 to 15 volts, you should connect it directly to the VIN and GND pins or to the battery clips on the development board. The VIN and GND pins are pin number 1 and 2 on the BASIC Stamp rev. D or BS1-IC or pin 24 and 23 on the BS2-IC, BS2E-IC and BS2SX-IC. If your power supply delivers a regulated 5 volts, you should connect it directly to the +5V and GND pins (14 and 2 on the BASIC Stamp rev. D, 5 and 2 on the BS1-IC, and 21 and 23 on the BS2-IC or BS2SX-IC). NOTE: When using battery or wall-pack power supplies, it is recommended to limit the voltage to 9 volts on the Stamp 1 and Stamp 2, and 7.5 volts on the Stamp 2e and Stamp 2sx.

## How much current does the BASIC Stamp consume?

The BASIC Stamp I consumes 2 mA in running mode and 20  $\mu$ A in sleep mode, not including any circuitry on the I/O pins. The BS2-IC consumes 8 mA in running mode and 100  $\mu$ A in sleep mode, not including any circuitry on the I/O pins. The BS2e-IC consumes 20 mA in running mode and 100  $\mu$ A in sleep mode, not including any circuitry on the I/O pins. The BS2SX-IC consumes 65 mA in running mode and 200  $\mu$ A in sleep mode, not including any circuitry on the I/O pins. A full comparison of the Stamps can be found at [the end of this document](#).

## Is the BASIC Stamp sensitive to static electricity?

While many electronic devices, including BASIC Stamps, can be damaged by static electricity, the BASIC Stamp is generally less sensitive to static. We do, however, recommend taking all the usual precautions when handling BASIC Stamps in static prone environments.



## **Are there any production possibilities with the BASIC Stamp?**

Yes. We offer the major components of the BASIC Stamp circuit (the interpreter, the EEPROM and the resonator) separately at a discounted price for tight integration into your products. For initial development purposes, request the OEMBS1 or OEMBS2 product. These are “OEM” versions of the modules with an easy to follow circuit layout and all the components provided in through-hole format. The “OEM” versions provide a great “base” unit, which you can build yourself, to test the circuit and help in your own designs.

## **What items do I need to get started with the BASIC Stamp?**

The items you need are: 1) the programming software, 2) the programming cable, 3) the manual, 4) the BASIC Stamp module itself, and optionally 5) the appropriate development board. If you are interested in using a particular version of the BASIC Stamp, it is best to purchase the BASIC Stamp I, BASIC Stamp II, BASIC STAMP Iie or BASIC Stamp IIsx Starter Kits. These kits include all five of the items listed above at a lower cost than if you were to purchase each of the components separately. If you choose not to purchase a starter kit, you can retrieve all the documentation and software from our web site and simply purchase the BASIC Stamp module you desire.

## **How can I make my own BASIC Stamp circuit?**

Parallax sells the BASIC Stamp’s interpreter chip in three different package type: DIP, SOIC and SSOP. Any of these can be used to create your own BASIC Stamp circuit for purposes of deeply embedding it within an application. For initial development purposes, request the [OEMBS1](#) or [OEMBS2](#) product. These are “OEM” versions of the modules with an easy to follow circuit layout and all the components provided in through-hole format. The “OEM” versions provide a great “base” unit, which you can build yourself, to test the circuit and help in your own designs. Please call Parallax Technical Support to receive information and pointers on other package types (particularly SSOP) as the pin configurations of the interpreter chips are different and are not properly shown in the 1995/96 Microchip documentation.

## **What kind of microchips are used in the BASIC Stamp?**

The BASIC Stamp I uses a PIC16C56 from Microchip Technology Inc. The BASIC Stamp II uses a PIC16C57 Microchip Technology Inc. The BASIC Stamps Iie and IIsx use an SX28AC from Scenix.

## **What kind of operating environment does the BASIC Stamp require?**

The BASIC Stamp modules will work in 0° to 70° C temperatures with up to 70%, non-condensing humidity. While the modules may continue to function outside these ranges, it is not guaranteed or recommended. Parallax also sells the BASIC Stamp I and BASIC Stamp II in industrial (-40° to 85° C) and extended (-40° to 125° C) ranges. (At the time of this writing, this option is not available for the BS2E-IC or BS2SX-IC) Additionally, it is best to keep the BASIC Stamps away from, or shielded from, any nearby RF interference as this may impact the accuracy of its I/O functions.

## **What are the main differences between the BASIC Stamps?**

The BASIC Stamp I has 8 I/O pins, room for 80 to 100 lines of code, executes approximately 2000 instructions per second and requires a parallel interface for programming. The BASIC Stamp II has 16 I/O pins, 2 dedicated serial port pins (1 input, 1 output), room for 500 to 600 lines of code, executes approximately 4000 instructions per second and requires a serial interface for programming. The BASIC Stamp IIsx has 16 I/O pins, 2 dedicated serial port pins (1 input, 1 output), room for 4000 lines of code, executes approximately 10000 instructions per second and requires a serial interface for programming. For a more detailed comparison, refer to the language differences page and review the BASIC Stamp I to BASIC Stamp II Conversion document on our web site:

<http://www.parallaxinc.com/stamps/langdiff>. There is also a comparison chart at [the end of this document](#).

## **Is the BASIC Stamp II better than the BASIC Stamp I?**

Yes and no, depending on your application. The BASIC Stamp II has twice the number of I/O pins, twice the execution speed, 5 times the memory (code) space, a serial port and 5 times the resolution on time sensitive commands. In many cases, the BASIC Stamp II fits applications better than the BASIC Stamp I. In some cases, however, this is not true. For example, you may have a need for only a couple of inputs and outputs with which you need to perform relatively simple, non speed-critical tasks. This is a perfect application for the BASIC Stamp I; the BASIC Stamp II would be overkill if used in this way. A slightly different example would be if you have a variable pulse width signal you need to monitor. The BASIC Stamp I can detect and measure a pulse as little as 10  $\mu$ S wide and as long as 0.65535 seconds wide. The BASIC Stamp II has 5 times the resolution, so it can measure a pulse as little as 2  $\mu$ S wide, however, only as long as 0.13107 seconds wide. While the better resolution may be helpful, the smaller maximum pulse width measuring capability may prove disadvantageous to your application.

## **Is the BASIC Stamp IIsx better than the BASIC Stamp II?**

Yes and no, depending on your application. The BASIC Stamp IIsx executes instructions 2.5 times as fast, has 8 times the memory (code) space and 2.5 times the resolution on time sensitive commands. In some cases, the BASIC Stamp IIsx fits applications better than the BASIC Stamp II. For example, you may have a need for additional program space that doesn't exist in the Stamp II. Another example would be if you have a variable pulse width signal you need to monitor. The BASIC Stamp II can detect and measure a pulse as little as 2  $\mu$ S wide and as long as 0.13107 seconds wide. The BASIC Stamp IIsx has 2.5 times the resolution, so it can measure a pulse as little as 800 nS wide, however, only as long as 0.05243 seconds wide. While the better resolution may be helpful, the smaller maximum pulse width measuring capability may prove disadvantageous to your application. A full comparison of the Stamps can be found at [the end of this document](#).

## **How many I/O pins does the BASIC Stamp have?**

The BASIC Stamp I has 8 I/O pins while the BASIC Stamp II, BASIC STAMP Iie and BASIC Stamp IIsx has 16 I/O pins plus 2 special serial port pins (1 input, 1 output).

## **What can I use the I/O pins for?**

The BASIC Stamp's I/O pins are perfectly suited for digital input and output with TTL/CMOS level (0 to 5 volt) signals. However, you can use some special commands and techniques to input and output limited analog signals. For example, the RCTIME and PWM commands can be used to read a variable resistance or output a variable voltage from 0 to 5 volts. The I/O pins can not be used to read analog voltages by themselves, however, but this can be done by interfacing the I/O pins to an A to D converter chip.

## **Can the I/O pins be used to control relays, solenoids and other similar devices?**

Yes, however, due to the demanding current and voltage requirements of some of these components, driver circuitry will need to be used to properly isolate the I/O pins from harmful effects. For examples of this, refer to BASIC Stamp I Application Note #6 and BASIC Stamp Article #6, "Silicon Steroids for Stamps".

## **Can I control LEDs with the I/O pins?**

Yes. Simply use a 470 ohm resistor in series with the LED to limit the current draw through the I/O pin (see "How much current can the I/O pins handle?"). Also, keep in mind that most LEDs require a lot of current in relation to what the BASIC Stamp can provide. If you attach and power 3 or more LEDs at one time from the BASIC Stamp's I/O pins, you are likely to see flaky and unpredictable results caused by voltage sag, I/O pin damage and/or hardware resets. Driver circuitry or low current LEDs will need to be used if you require such an application.

## **How much current can the I/O pins handle?**

On the BASIC Stamp I and BASIC Stamp II, each I/O pin is capable of sourcing 20 mA and sinking 25 mA. The total across all I/O pins should not exceed 40 mA source or 50 mA sink at any given time, however. One exception to this rule exists with the BS2-IC. If the BS2-IC is powered by an external 5-volt regulator capable of delivering at least 100 mA, the total across each group of 8 I/O pins (0 - 7 and 8 - 15) can reach 40 mA source or 50 mA sink providing a total of 80 mA source or 100 mA sink overall. The BASIC Stamp IIsx's I/O pins can source and sink 30 mA, and the total across all I/O pins should not exceed 150 mA.

## **What is the input impedance of the BASIC Stamp's I/O pins?**

Approximately 1 Meg Ohm.

## **How fast does the BASIC Stamp execute its PBASIC code?**

On average, the BASIC Stamp I executes 2000 instructions per second, the BASIC Stamp II executes 4000 instructions per second, the BASIC Stamp Iie executes apx 3800 instructions per second and the BASIC Stamp IIsx executes 10000 instructions per second. This is only an estimate as the true execution speed depends upon many factors including, the particular set of instructions used and the type of arguments provided to the instructions. Some instructions execute at a much faster rate while others, especially those that have to deal with external signal measurement or specific protocol speeds, will execute much slower.

## **What is the VIN pin used for?**

The VIN (Voltage Input) pin is used to power the BASIC Stamp from a 6 to 15 volt source. The VIN pin is the positive connection while the VSS pin is the negative, or ground, connection. When powered from the VIN pin the BASIC Stamp regulates the voltage down and outputs +5 volts on the VDD pin. For optimal operation, it is recommended to keep the voltage below 12 volts for the BASIC Stamp I and II, IIe, and around 7.5 volts for the BASIC Stamp IIsx.

## **How does the VDD pin work?**

The VDD pin (+5V) outputs 5 volts when the BASIC Stamp is powered by a 6 to 15 volt source using the VIN and VSS (ground) pins. The VDD pin can be used to power other circuitry if the overall current consumption is within the capabilities of the BASIC Stamp's regulator. The VDD pin can also be used to power the BASIC Stamp from an external 5-volt regulator. In this case, the VDD pin is the +5 volt connection and the VSS (ground) pin is the negative connection of the regulator.

## **How does the reset pin (RES) work?**

The reset pin is internally controlled by the BASIC Stamp's brownout detector. It is normally high (+5V), which allows the BASIC Stamp to run its program normally, and is pulled low when the power supply voltage drops below 4 volts (safely putting the BASIC Stamp to sleep). This pin can be monitored to detect when a reset condition occurs, or, you can pull the line to ground to force a reset. After the reset pin is allowed to rise to +5 volts, the BASIC Stamp wakes up and starts executing its program from the first line of code. Do not drive this pin high, it should be left electrically disconnected (floating) when you want the BASIC Stamp to run normally. This pin has an inverse relationship to the ATN (Attention) pin on the BASIC Stamp II, BASIC STAMP IIe and BASIC Stamp IIsx.

## **How does the ATN (Attention) pin work (BASIC Stamp II, BASIC STAMP IIe and BASIC Stamp IIsx)?**

The ATN pin is used strictly for programming the BASIC Stamp II, BASIC STAMP IIe, or BASIC Stamp IIsx, and should not be connected to anything other than the DTR pin of the programming connector and/or filtering capacitors. The ATN pin is an input and has an inverse relationship with the reset (RES) pin. When the ATN pin is left unconnected, or is driven low, it causes the RES pin to go high (allowing the BASIC Stamp II, IIe or IIsx to function properly). When the ATN pin is driven high it causes the RES pin to go low (performing a hardware reset on the BASIC Stamp II, IIe or IIsx). The BASIC Stamp editor software toggles this pin from low to high and then low again at the start of the programming process to perform a hardware reset on the module and begin the bi-directional programming communication. It is important to be aware of a particular artifact of the DTR pin on a standard serial port: upon port selection by standard software (terminal packages, development tools, etc.), the DTR pin is driven high by default. If the same connections for programming the BASIC Stamp II, IIe or IIsx are used to communicate with it via a serial port, the default state of the DTR pin will cause the BASIC Stamp to be held in permanent reset and its PBASIC program will not execute. (This is the case on the Stamp II Carrier Board). This problem can be fixed in a number of ways: 1) disconnect the DTR to ATN pin connection when trying to communicate serially to the BASIC Stamp, or 2) explicitly set the DTR pin to low after selecting the port if the terminal package or development tool allows this, or 3) modify the programming connection to include a 0.1  $\mu$ F capacitor in series with the DTR to ATN pin connection and a 0.1  $\mu$ F capacitor from ATN to ground (VSS). The last two options will allow you to use the same connection for both programming and serial communications. (The Super Carrier Board, BASIC Stamp Activity Board and Board of Education use the last method).

## **How much current can the BASIC Stamp's on-board regulator provide?**

The built in 5-volt regulator on the BASIC Stamp I and BASIC Stamp II can supply 50 mA of current if powered by a 12-volt source. The BASIC Stamp IIsx's regulator can supply 150 mA if powered by a 7.5-volt source. The BASIC Stamp I consumes 2 mA in running mode and the BASIC Stamp II consumes 7 mA. This leaves 48 mA and 43 mA respectively for use with other circuitry via the VDD pin and the I/O pins. The BASIC Stamp IIsx consumes 65 mA, leaving 85 mA for use with other circuitry via the VDD pin and the I/O pins.

## **Can the BASIC Stamp generate a sine wave?**

Only the BASIC Stamp II, BASIC STAMP Iie and BASIC Stamp IIsx can generate sine waves. The BASIC Stamp II uses the `FREQOUT` command to generate from 0 Hz to 32,767 Hz. The BASIC Stamp IIsx uses the `FREQOUT` command to generate from 0 Hz to 81,917 Hz. Note that the output from the `FREQOUT` command appears to be static when viewed on an oscilloscope. This is due to the nature of the underlying algorithm that generates the frequency, however, when averaged out over time, the result is a sine wave. You may achieve a cleaner signal by attaching the I/O pin to an appropriate RC filter circuit.

## **Can the BASIC Stamp generate a sine wave on two pins at once?**

No. The BASIC Stamp is a single-task device and thus will execute only one `FREQOUT` command at a time.

## **Can the BASIC Stamp generate more than one sine wave on a pin at the same time?**

Yes, the BASIC Stamp II, Iie and IIsx's `FREQOUT` command can generate two frequencies at the same time, on the same pin. This can be used to create harmonics, which greatly improve the sound output.

## **Can the BASIC Stamp generate square waves?**

Yes, however, it is limited. The `SOUND` command in the BASIC Stamp I generates square waves which vary from 94.8 Hz to 10,550 Hz (non-linearly). The BASIC Stamp II, Iie and IIsx, however, does not include any functions to generate an accurately timed square wave, however, it can be done manually as long as fast timing is not an issue. This can be done with multiple lines of code that simply set a pin high or low and pause for the appropriate duration in-between transitions. The timing will vary somewhat depending upon your loop execution speeds.

## **Does the BASIC Stamp include a real time clock function?**

No. If you need to keep track of time, especially dates and time, it is best to interface a real time clock to the BASIC Stamp. Many are available from various manufacturers. The real time clocks with a serial interface are usually the best choice, as they require fewer I/O pins from the BASIC Stamp to function properly. Parallax sells the DS1302 Application Kit to address this issue.

## **Does the BASIC Stamp have a built-in timer?**

Yes, however, it is used for timing functions within the BASIC Stamp interpreter and there is no direct access to it through any PBASIC commands. Refer to BASIC Stamp I Application Note #20, "An Accurate Timebase" for one possible solution to timing problems.

## **Does the BASIC Stamp support interrupts?**

No, the interpreter chip used in the BASIC Stamp does not support interrupts. In many cases, a fast polling routine may be used to accomplish the same effect, however, depending on the number and size of the tasks involved in some applications, this will not be fast enough and the BASIC Stamp may not be a plausible solution.

## **Does the BASIC Stamp II, BASIC STAMP IIe or BASIC Stamp IIsx have the same bsave/bsload feature available in the BASIC Stamp I?**

No. It is intended to provide this feature in the future.

## **How do I program the BASIC Stamp?**

You can write your PBASIC program using the BASIC Stamp I, II, IIe or IIsx editor on a standard IBM compatible PC. Both DOS and Windows versions are available for the BASIC Stamp II, IIe and IIsx. After you write the code for your application, you simply connect the BASIC Stamp to the computer's parallel port (BASIC Stamp I) or serial port (BASIC Stamp II, IIe and IIsx), provide power to the BASIC Stamp and press ALT-R (DOS version) or CTRL-R (Windows version) within the appropriate BASIC Stamp editor to download your program into the BASIC Stamp's EEPROM. As soon as the program has been downloaded successfully, it begins executing its new program from the first line of code.

## **How do I connect the BASIC Stamp to my computer for programming?**

The BASIC Stamp I (rev. D or BS1-IC) requires a three-wire connection to any available parallel port on your PC. The BASIC Stamp II, IIe and IIsx (BS2-IC and BS2SX-IC) require a standard, straight-through, serial cable connection to a 9-pin or 25-pin serial port on your PC. Only 6 wires are used for programming the BASIC Stamp II, IIe or IIsx, 4 of which are absolutely required. You can not use a null-modem cable to program the BASIC Stamp II, IIe or IIsx. Refer to the schematics on our web site for the details of the cable connections, <http://www.parallaxinc.com/stamps/stmpsche.htm>.

## **How does my program get stored in the BASIC Stamp?**

Your PBASIC code is tokenized (compressed) and stored within the Stamp's EEPROM memory. This memory is non-volatile, meaning it retains its program even without power. Every time the BASIC Stamp receives power, it starts running its PBASIC code starting with the first executable line. This EEPROM can be rewritten immediately, without any lengthy erase cycle or procedure, by simply re-downloading the code from the BASIC Stamp editor again. Each location in the EEPROM is guaranteed for 10,000,000 write cycles before it wears out. The code is tokenized before downloading. Source code elements like comments and constant and variable names are not stored in the BASIC Stamp, thus you may feel free to use as many comments and descriptive symbol names in your code as you like without worrying about increasing your code size.

## **How do I erase the BASIC Stamp's program space?**

An erase cycle is performed at the beginning of every programming process (in the BASIC Stamp I) or during the programming process (in the BASIC Stamp II, IIe and IIsx), thus, by downloading from the BASIC Stamp editor the necessary areas of the EEPROM are erased and then reprogrammed with the current PBASIC program in the editor. The BASIC Stamp II, IIe and IIsx only erase data in 16-byte increments. It is possible to completely erase the EEPROM by downloading the following line of code (in a program by itself):

```
DATA 0(2048)
```

## **How do I reprogram the BASIC Stamp?**

Simply re-connect it to the computer, run the BASIC Stamp editor and press ALT-R (DOS software) or CTRL-R (Windows software).

## **How big of a program can I store in the BASIC Stamp?**

The BASIC Stamp I has 256 bytes of program storage; enough for 80 to 100 lines of PBASIC1 code. The BASIC Stamp II has 2048 bytes of program storage; enough for 500 to 600 lines of PBASIC2 code. The BASIC Stamps IIe and IIsx each have 16,384 bytes of program storage separated into 8 pages of 2048 bytes. Each page (or code section) can hold 500 to 600 lines of PBASIC2e, or PBASIC2sx code, respectively, for an overall total of approx. 4000 instructions.

## **Can I expand the program memory?**

No, the program memory is not expandable as the interpreter chip expects the memory to be a specific, fixed size.

## **Can I expand the data memory?**

Yes, you may interface EEPROMs, or other memory devices, the BASIC Stamp's I/O pins to gain more data storage area. You will have to include the appropriate code within your PBASIC program to interface to the particular device you choose.

## **How difficult is it to program the BASIC Stamp?**

When compared to other programmable microcontrollers, the BASIC Stamps are perhaps the easiest to use because of their simple, but powerful, language structure and straightforward method of downloading and debugging. If you have some experience programming in BASIC, C or Pascal, you should find the learning process with the BASIC Stamp to be a simple one. If, however, you have never had any programming experience, you may have to spend some extra time studying the examples and application notes before you feel comfortable with the BASIC Stamp.

## **Can I program the BASIC Stamp in Visual BASIC or QBASIC?**

No. The BASIC Stamp is a microcontroller, not a miniature PC. It does not have a graphical user interface, a hard drive or a lot of RAM. The BASIC Stamp must only be programmed with PBASIC, which has been specifically designed to exploit all of the BASIC Stamp's capabilities.

## **What is PBASIC?**

PBASIC (Parallax BASIC) is a hybrid form of the BASIC programming language in which many people are familiar. Currently there are three versions of PBASIC: PBASIC1 for the BASIC Stamp I, PBASIC2 for the BASIC Stamp II and PBASIC2sx for the BASIC Stamp IIsx. Each version is specifically tailored to take advantage of the inherent features of the hardware it runs on. PBASIC is called a hybrid because, while it contains some simplified forms of standard BASIC commands, it also has special commands to efficiently control the I/O pins.

## **Can I imbed assembly language routines in my PBASIC code?**

No. The BASIC Stamp is a PBASIC interpreter only and cannot except assembly language routines.

## How much space does each PBASIC command take?

Each command takes a variable amount of space within the BASIC Stamp's EEPROM. It varies due to complexities of the command itself and the types of arguments you supply each command. Generally, each command takes 2 to 4 bytes of memory space, however, commands like SERIN, SEROUT, LOOKUP and LOOKDOWN, which accept a variable length list of arguments, may take tens of bytes or more.

## How do I know how much space my PBASIC program consumes?

On the BASIC Stamp I, enter the following code at the start of your PBASIC1 program:

```
READ 255,B0
DEBUG #B0
```

Upon running the program, a number will display in the debug window of the editor. Use the following equation to determine how many bytes are used by your PBASIC1 code:  $255 - \# - 6$ ; where # is the number displayed on the debug window. Note, the "- 6" in the equation results from the fact that the above two lines of code take 6 bytes of program space, thus without those two lines, your program takes 6 fewer bytes of space.

On the BASIC Stamp II, IIe or IIsx, from within the editor and with your PBASIC program on the screen, press ALT-M (DOS software) or CTRL-M (Windows software) to view the memory map screens. In the DOS software, the first screen shows your RAM space, the second screen shows an expanded view of the program (EEPROM) space and the third screen shows a detailed view of the program space. Press the spacebar to alternate between the three screens. In the Windows software, all EEPROM and RAM is shown on one window.

## Why doesn't the Memory Map feature show me data I have already stored in the BASIC Stamp II, IIe or IIsx's EEPROM?

The memory map feature (ALT-M, DOS software or CTRL-M, Windows software) shows only what the memory space will look like immediately after you program the Stamp II, IIe or IIsx. The editor determines this by examining your PBASIC source code within the editor, not the PBASIC code in the Stamp. It does not read the BASIC Stamp's EEPROM. It will only display data in the EEPROM that you've explicitly defined with DATA commands; no data which is operated upon by READ and WRITE commands can be shown since those are run-time, rather than compile-time, functions.

## How do I debug my PBASIC programs?

You may use two features of the BASIC Stamp editor to debug your PBASIC program: 1) Syntax checking, and 2) the DEBUG command. Syntax checking alerts you to any syntactical errors and is automatically performed on your code the moment you try to download to the BASIC Stamp. Any syntax errors will cause the download process to abort and the editor will display an error message with a highlight cursor pointing to the source of the error in your code. Logical errors are sometimes more difficult to find but may be more easily discovered by using the DEBUG command. Debug operates similar to the PRINT command in the BASIC language and can be used to print the current status of specific variables within your PBASIC program as it is executed within the BASIC stamp. If your PBASIC code includes a DEBUG command, the editor opens up a special window at the end of the download process to display the results for you. In the Windows editor, the Debug window (called the Debug Terminal) can be left open, while you edit your code.



## **How do I print out my PBASIC programs?**

The BASIC Stamp DOS editors do not have a print option built-in, however, the source file created by the editor is simply a DOS text file. This means it can be printed with the standard PRINT or TYPE commands as in:

```
PRINT source.bas
```

-or-

```
TYPE source.bas > LPT1
```

where source.bas is the name of your PBASIC program. The Windows software features built-in printing support.

## **Can I read-out, or upload, the PBASIC program which is already stored in the BASIC Stamp?**

No. For security reasons, this feature was not made available. It is possible to read the tokenized form of the PBASIC code out of the BASIC Stamp's EEPROM, but there is no easy way to "reassemble" it into usable source code.

## **Since the BASIC Stamp II, IIe and IIsx use the serial port for programming, can I use a simple terminal program to download my PBASIC code?**

No. The download process requires a very fast, bi-directional communication between the BASIC Stamp and PC to program and verify the contents of the BASIC Stamp's memory.

## **Will a program and circuitry designed for the BASIC Stamp I work with the BASIC Stamp II, IIe or IIsx?**

Yes, however, some changes, usually in the PBASIC code only, will have to be made for it to run properly or to take advantage of some of the new features of the BASIC Stamp II, IIe or IIsx. For help in performing these modifications or for insight into how the PBASIC1 language differs from PBASIC2, review Appendix C in the BASIC Stamp Manual.

## **Can I program the BASIC Stamp II, IIe or IIsx with the BASIC Stamp I editor (stamp.exe)?**

No. The BASIC Stamp II, IIe and IIsx requires the BASIC Stamp II, IIe and IIsx editor (stamp2.exe, stamp2e.exe or stamp2sx.exe in DOS, stampw.exe in Windows) due to the enhancements in the PBASIC2 language and the serial, rather than parallel, programming interface. Similarly, you cannot program the BASIC Stamp I with the BASIC Stamp II, IIe or IIsx editors.

## **How do I make my PBASIC program start over or continue running forever?**

All BASIC programs will simply end if one or more of their executable paths don't lead them to a continuous loop. In the simplest case, a program can be made to continue endlessly by adding a label (a unique name followed by a colon ':') to the start of code and adding a GOTO statement at the end of code directing logical execution back to that label. Without this, the BASIC Stamp will run the program only once and then will remain unresponsive until the power is cycled or a reset condition is created.

## How do I set an I/O pin to input or output mode?

There are many ways to set the I/O pin directions. The most readable and intuitive method is to use the INPUT and OUTPUT commands. For example:

```
INPUT 5
OUTPUT 6
```

will set I/O pin 5 to an input and I/O pin 6 to an output. By default, upon reset or startup, all I/O pins are inputs, thus only the second line of code above is really necessary.

Another method for setting the pin directions is to use the predefined variable DIRS or any one of its derivatives. The following code is functionally equivalent to the first example:

```
DIR5 = 0
DIR6 = 1
```

DIR0 through DIR15 (or DIR7 on the BASIC Stamp I) are bit variables which control the direction of the corresponding I/O pin. DIRS is a 16-bit variable (or 8-bit variable on the BASIC Stamp I) in which each bit represents the direction of the corresponding I/O pin. A '1' bit means output and a '0' bit means input. The advantage of this method of declaring pin directions is that multiple pin directions can be defined at once:

```
DIRS = %00101110
```

The above line of code defines I/O pins 7, 6, 4 and 0 as inputs and 5, 3, 2 and 1 as outputs. Review the BASIC Stamp manual for more examples of this.

## How do I make an I/O pin output a high or a low?

As with setting I/O pin directions, there are many ways to set the I/O pin states. The most readable and intuitive method is to use the HIGH and LOW commands. For example:

```
HIGH 5
LOW 6
```

will first set I/O pin 5 to output mode and then set it to a high (+5V) state. The second line will first set I/O pin 6 to output mode and then set it to a low (0V) state. Notice that these commands automatically set the pin direction for you; it's not necessary to manually set the pins to output when using the HIGH and LOW commands. Keep in mind that if a pin is set to input, its logical state, high or low, remains the same internally within the BASIC Stamp but will no longer affect the voltage on the I/O pin.

Another method for setting the pin state is to use the predefined variable OUTS (or PINS in the BASIC Stamp I) or any one of its derivatives. The following code is functionally equivalent to the first example:

```
DIR5 = 1
DIR6 = 1
OUT5 = 1
OUT6 = 0
```

Notice here that we had to explicitly define the I/O pins as outputs first. If we had left the pins as inputs, the OUT5 and OUT6 lines would have changed the state of the pins internally, but those changes would not appear externally from the BASIC Stamp. OUT0 through OUT15 (or PIN0 through PIN7 on the BASIC Stamp I) are bit variables which control the logical state of the corresponding I/O pin. OUTS is a 16-bit variable (PINS is an 8-bit variable on the BASIC Stamp I) in which each bit represents the state of the corresponding I/O pin. A '1' bit means high and a '0' bit means low. The advantage of this method of declaring pin states is that multiple pin states can be defined at once:

```
DIRS = %00101110
OUTS = %00001010
```

The above line of code defines I/O pins 7, 6, 4 and 0 as inputs and 5, 3, 2 and 1 as outputs. And sets I/O pins 7, 6, 5, 4, 2 and 0 to logical 0's and I/O pins 3 and 1 to logical 1's. Only pins 5, 3, 2 and 1 will output the voltage set in the OUTS variable because they are the only pins defined as outputs by the DIRS variable. Review the BASIC Stamp manual for more examples of this.

## How do I input or output data in parallel on the I/O pins?

All BASIC Stamps have predefined variables that refer to the I/O pins. On the BASIC Stamp I, the PINS variable refers to all eight I/O pins. Thus code such as:

```
SYMBOL Snapshot = B0
Snapshot = PINS
```

would copy the current state of all I/O pins (an 8-bit byte) into the variable Snapshot. Similarly:

```
DIRS = %11111111 'Set all eight pins as outputs
PINS = 150
```

would set the eight output pins to the binary form of 150, or %10010110.

On the BASIC Stamp II, IIe and IIsx, the PINS variable was replaced with the INS and OUTS variables. Both INS and OUTS refer to all sixteen pins, while their derivatives, INL, INH, OUTL and OUTH refer to the low and high bytes (groups of 8 pins). For example, the following code will input 8-bits in parallel from the lower 8 pins:

```
Snapshot VAR BYTE
Snapshot = INL
```

The following code will output 8-bits in parallel from the upper 8 pins:

```
DIRS = %1111111100000000 'Set upper 8 pins to outputs
OUTH = 255
```

## Why do the BASIC Stamp's output pins toggle briefly when I use the SLEEP, NAP or END commands or allow my program to end naturally?

Inside the BASIC Stamp's interpreter chip is a watchdog timer whose main purpose is to reset the interpreter chip if, for some reason, it should stop functioning properly. The SLEEP and NAP commands also utilize the watchdog timer to periodically (every 2.3 seconds to be exact) "wake-up" the BASIC Stamp from its low-power mode. Upon reset, the I/O pins are set to inputs for approximately 18 mS before returning to their previous directions and states. If you have an output pin set to a logical 1 state (+5V) and you use the SLEEP command, every 2.3 seconds during sleep mode that I/O pin will switch to an input for 18 mS causing a momentary signal loss. This "power glitch" is easily viewable with an LED and a 470 ohm resistor tied to an I/O pin and switched on just before entering sleep mode. In many cases this problem can be remedied by tying a pull-up or pull-down resistor to the I/O pin in question to provide a constant source of power should the I/O pin change directions. Allowing a PBASIC program to end naturally, or using the END command, will exhibit the same "power glitch" behavior because the interpreter chip enters a low-power state. The BASIC Stamp II, IIe and IIsx includes the STOP command which acts just like END, except the output pins remain driven.

## **The LET command is not available on the BASIC Stamp II, IIe and IIsx. Does this mean the BASIC Stamp II, IIe and IIsx cannot perform mathematical operations?**

No. The LET command on the BASIC Stamp I was optional; i.e.: you could leave it out of the variable assignment statement or equation and it would work normally. To save some space in the interpreter code of the BASIC Stamp II, IIe and IIsx, the LET command was removed completely, however, the mathematical ability was not removed and, in fact, it is greatly improved on the BASIC Stamp II, IIe and IIsx.

## **Can the BASIC Stamp store data, such as temperature readings, for review or use at a later time?**

Yes. The READ and WRITE commands allow the BASIC Stamp to manipulate memory locations in its EEPROM. Any space in the built-in EEPROM that is not occupied by PBASIC code may be used for data storage and is just as non-volatile as the PBASIC code itself. You may also attach external EEPROMs of various sizes to gain an even greater data storage area. The READ and WRITE commands will not function on external EEPROMs so you must write the appropriate code to interface with the device.

## **How can I store a word value into the internal EEPROM?**

The internal EEPROM of the BASIC Stamp (used for both program and data storage) is organized and accessed in units of bytes only. Since a word value is actually two bytes it requires two consecutive memory locations to store it in EEPROM. You must break the word value into its lower and upper byte parts and use two WRITE commands to store the value, or two READ commands to retrieve it.

The following code demonstrates this for the BASIC Stamp I:

```
W0 = 1250
WRITE 0, B0
WRITE 1, B1
```

The following code demonstrates this for the BASIC Stamp II:

```
Temp VAR WORD

Temp = 1250
WRITE 0, Temp.LOWBYTE
WRITE 1, Temp.HIGHBYTE
```

**How does the fpin (flow control pin) work in the SERIN and SEROUT commands on the BASIC Stamp II, IIE and IISx?**

The optional fpin argument in the SERIN and SEROUT commands allows the use of an I/O pin as a hardware flow control line between two BASIC Stamp II, IIE and IISx's at baud rates of up to 19.2 kbps. The flow control pin is always controlled by the receiving device (SERIN) and monitored by the sending device (SEROUT). The logical state of this pin and its meaning depend on the data mode, true or inverted, as specified within the baudmode parameter. If true data mode is selected, the receiving BASIC Stamp II, IIE or IISx holds the fpin low to indicate it is ready to receive or high to indicate it is not ready to receive. If inverted data mode is selected, the receiving BASIC Stamp II, IIE or IISx holds the fpin high to indicate it is ready to receive or low to indicate it is not ready to receive.

Assuming true data mode: the fpin is set to output high the moment the SERIN command is first encountered and remains high until the SERIN command is complete at which point the fpin is set to low and remains there until the SERIN command is executed again. The transmitting BASIC Stamp II, IIE or IISx's SEROUT command monitors this line and halts transmission whenever a low on the fpin is detected.

**How are arithmetic expressions evaluated within the BASIC Stamp?**

All expressions are evaluated using 16-bit, integer arithmetic. Even if byte and bit variables are used in expressions they are first expanded to 16-bits and then the expression is evaluated.

In the BASIC Stamp I, mathematical expressions are evaluated strictly from left to right. No order-of-precedence is utilized, parentheses are not allowed and expressions must be on a line by themselves; i.e.: they can not be used as an argument for a command. For example, the expression:  $W0 = 10 - 2 * 3$  results in the value 24 being stored into W0 (not the value 4 as you might think). To evaluate this expression properly, specify it as:  $W0 = 0 - 2 * 3 + 10$ .

In the BASIC Stamp II, IIE and IISx, mathematical expressions are evaluated from left to right, as in the BASIC Stamp I. No order-of-precedence is utilized, however, unlike the BASIC Stamp I, parentheses are allowed and expressions may be used as arguments within commands. For example, the expression:  $Answer = 10 - 2 * 3$  results in the value 24 being stored into Answer, however:  $Answer = 10 - (2 * 3)$  results in the value 4.

**Does the BASIC Stamp handle signed numbers and arithmetic?**

Yes. The BASIC Stamp uses twos-compliment notation for signed numbers. This means that the expression:  $0 - 10 + 5$  will result in -5 if viewed as a signed number, however, most instructions see the number as a positive value, in this case 65531 (the twos-compliment value for -5). All mathematical operators, except division, will work properly with signed numbers in the BASIC Stamp and signed numbers can be formatted for output properly using the SDEC, SHEX and SBIN formatters within DEBUG and SEROUT statements on the BASIC Stamp II, IIE and IISx. Be careful how you use signed numbers elsewhere. For example, if the value -5 is stored in a variable called Temp, and you use the following statement:

```
IF Temp < 0 THEN Loop
```

it will evaluate to false and will not branch to Loop because -5 is actually 65531 in twos-compliment form and thus 65531 is not less than 0.

### How can I define an alias to an I/O pin or another variable?

In the BASIC Stamp I you could specify the following:

|                |   |      |
|----------------|---|------|
| SYMBOL Counter | = | B0   |
| SYMBOL Index   | = | B0   |
| SYMBOL LED     | = | PIN0 |

to designate the symbol Index as an alias to the Counter variable and the symbol LED as an alias to I/O pin 0. Since Counter and Index use the same register, B0, anytime Counter is changed, Index will change in the same way. The symbol LED will always contain either a 1 or a 0 depending on the logical state of I/O pin 0.

In the BASIC Stamp II, IIE or IISx you could specify the following:

|         |     |         |
|---------|-----|---------|
| Counter | VAR | BYTE    |
| Index   | VAR | Counter |
| LED     | VAR | IN0     |

to designate the symbol Index as an alias to the Counter variable and the symbol LED as an alias to I/O pin 0. Since Index uses the same memory space as Counter, anytime Counter is changed, Index will change in the same way. The symbol LED will always contain either a 1 or a 0 depending on the logical state of I/O pin 0.

### How do I reference a specific bit within a byte or word variable?

On the BASIC Stamp I there is only one general purpose word register, W0, and two general purpose byte registers, B0 and B1, which are bit addressable. The predefined symbols Bit0 through Bit15 refer to the corresponding bits within W0 as well as B0 and B1 since those two byte variables correspond to the lower and upper bytes of W0 respectively. Thus Bit0 is the lowest bit of both W0 and B0 while Bit8 is the lowest bit of B1 (and the 9th bit of W0).

On the BASIC Stamp II, IIE and IISx all the variables are bit addressable, as well as nibble addressable. You can reference a specific part of a variable by specifying the variable name followed by a dot '.' and then the name of the portion you're interested in. For example, bit 2 of a word variable called Temp can be referenced with the notation: Temp.BIT2. Additionally, the second nibble of Temp can be referenced with: Temp.NIB1. The valid modifiers are: BIT0 through BIT15, LOWBIT, HIGHBIT, NIB0 through NIB3, LOWNIB, HIGHNIB, BYTE0 through BYTE3, LOWBYTE and HIGHBYTE.

### How do I define a string variable?

A string variable is simply an array of bytes. Only the BASIC Stamp II, IIE and IISx allows the explicit definition of arrays. If, for example, you want to define a string variable called Text to hold 6 characters, use the following code:

```
Text VAR BYTE(6)
```

The elements of an array can be accessed by specifying the array name followed by an open parenthesis, an index value and a close parenthesis. The first element is always index 0. For example, the second element, or character in this case, can be referenced with: Text(1) and the last element with: Text(5). Byte arrays are useful for receiving a string of characters via the SERIN command using the STR modifier. The BASIC Stamp II, IIE and IISx allows you to define arrays of bits, nibbles and words as well.

## How do I define aliases to specific bytes within a word array on the BASIC Stamp II, IIe and IIsx?

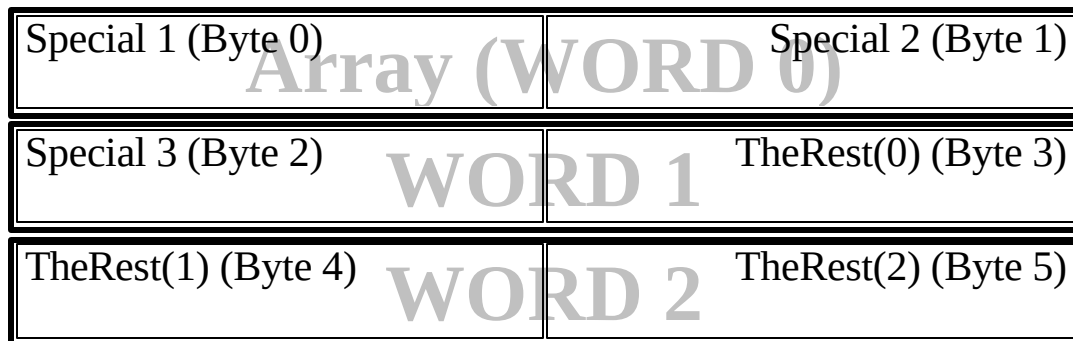
The BASIC Stamp II, IIe and IIsx uses a very relaxed and flexible indexing scheme when it comes to arrays of RAM space. When trying to define aliases in this way it is best to view the entire RAM space as one array of word, byte, nibble and bit registers. The BASIC Stamp II, IIe and IIsx organizes variables in the RAM space from largest entities (word declarations) to smallest entities (bit declarations), regardless of the order you define them, to be most efficient. The relaxed nature of indexing allows a user to cross array boundaries at will, or by accident, quite easily. For example, the following code defines two distinct byte variables:

```
Result VAR BYTE
Temp VAR BYTE
```

Normally you would reference the Result and Temp variables by name, as in: Result = 10. Even though this variable is not an array, you can still access it as such. The code: Result(0) = 10 is equivalent to the previous example. The most interesting, and powerful, feature is that you can index beyond the boundaries of this byte variable, as in Result(1) = 20. This statement actually modifies the byte of RAM immediately following the Result variable. The next byte happens to have been defined as Temp, in this case, thus the variable Temp will now equal 20. By carefully defining variables we can take advantage of this flexible feature to define byte sized aliases to specific locations within a word array. Suppose we have the need for an array of three words in which the first three bytes have a special meaning; we'd like to be able to easily manipulate those bytes as well as access the three distinct word elements.

```
Array VAR WORD
Special1 VAR Array.LOWBYTE
Special2 VAR Array.HIGHBYTE
Special3 VAR BYTE
TheRest VAR BYTE(3)
```

Remember: the BASIC Stamp II, IIe and IIsx will always organize RAM space from biggest elements to smallest. In the example above we defined a single word variable called array (this becomes assigned to the very first word in RAM space, word 0). The next two definitions create aliases to the first and second bytes of Array (no additional RAM space is allocated here since we're pointing at previously allocated space, byte 0 and byte 1). Next we define a single byte variable (this becomes assigned to the first byte of RAM immediately following the Array variable, byte 2). Finally, we define an array of three bytes (which are assigned to the next three bytes of RAM immediately following the previous byte declaration, byte 3, byte 4 and byte 5). Visually, our RAM space now appears like this:



Now, Array(0), Array(1) and Array(2) refer to the three words of our array and Special1, Special2 and Special3 refer to the first three bytes within our three byte array.



## Stamp Specifications (revised 6/00)

| Released Products              | Rev.D / BS1-IC                       | BS2-IC                        | BS2e-IC                       | BS2SX-IC                      |
|--------------------------------|--------------------------------------|-------------------------------|-------------------------------|-------------------------------|
| Package                        | PCB w/Proto / 14-pin SIP             | 24-pin DIP                    | 24-pin DIP                    | 24-pin DIP                    |
| Package Size (L x W x H)       | 2.5" x 1.5" x .5" / 1.4" x .6" x .1" | 1.2" x 0.6" x 0.4"            | 1.2" x 0.6" x 0.4"            | 1.2" x 0.6" x 0.4"            |
| Environment                    | 0° - 70° C* (32° - 158° F) **        | 0° - 70° C* (32° - 158° F) ** | 0° - 70° C* (32° - 158° F) ** | 0° - 70° C* (32° - 158° F) ** |
| Microcontroller                | Microchip PIC16C56                   | Microchip PIC16C57            | Scenix SX28AC                 | Scenix SX28AC                 |
| Processor Speed                | 4 MHz                                | 20 MHz                        | 20 MHz                        | 50 MHz                        |
| Program Execution Speed        | ~2,000 instructions/sec.             | ~4,000 instructions/sec.      | ~4,000 instructions/sec.      | ~10,000 instructions/sec.     |
| RAM Size                       | 16 Bytes (2 I/O, 14 Variable)        | 32 Bytes (6 I/O, 26 Variable) | 32 Bytes (6 I/O, 26 Variable) | 32 Bytes (6 I/O, 26 Variable) |
| Scratch Pad RAM                | N/A                                  | N/A                           | 64 Bytes                      | 64 Bytes                      |
| EEPROM (Program) Size          | 256 Bytes, ~80 instructions          | 2K Bytes, ~500 instructions   | 8 x 2K Bytes, ~4,000 inst.    | 8 x 2K Bytes, ~4,000 inst.    |
| Number of I/O pins             | 8                                    | 16 + 2 Dedicated Serial       | 16 + 2 Dedicated Serial       | 16 + 2 Dedicated Serial       |
| Voltage Requirements           | 5 - 15 vdc                           | 5 - 15 vdc                    | 5 - 12 vdc                    | 5 - 12 vdc                    |
| Current Draw @ 5V              | 2mA Run / 20µA Sleep                 | 8 mA Run / 100 µA Sleep       | 20 mA Run / 100 µA Sleep      | 60 mA Run / 200 µA Sleep      |
| Source / Sink Current per I/O  | 20 mA / 25 mA                        | 20 mA / 25 mA                 | 30 mA / 30 mA                 | 30 mA / 30 mA                 |
| Source / Sink Current per unit | 40 mA / 50 mA                        | 40 mA / 50 mA per 8 I/O pins  | 60 mA / 60 mA per 8 I/O pins  | 60 mA / 60 mA per 8 I/O pins  |
| PBASIC Commands                | 32                                   | 36                            | 39                            | 39                            |
| PC Programming Interface       | Parallel Port                        | Serial Port (9600 baud)       | Serial Port (9600 baud)       | Serial Port (9600 baud)       |
| DOS Text Editor                | STAMP.EXE                            | STAMP2.EXE                    | STAMP2E.EXE                   | STAMP2SX.EXE                  |
| Windows Text Editor            | N/A                                  | Stampw.exe (v1.04 and up)     | Stampw.exe (v1.096 and up)    | Stampw.exe (v1.091 and up)    |

\* Industrial Models Available - Environment is -40° - 85° C (-40° - 185° F)\*\*

\*\* 70% Non-Condensing Humidity