

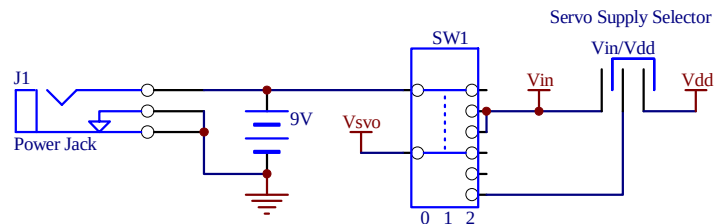
Distributed by:



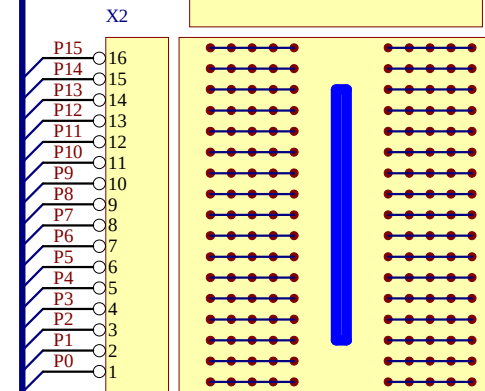
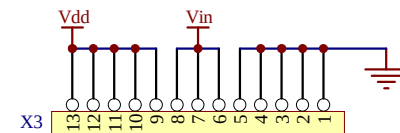
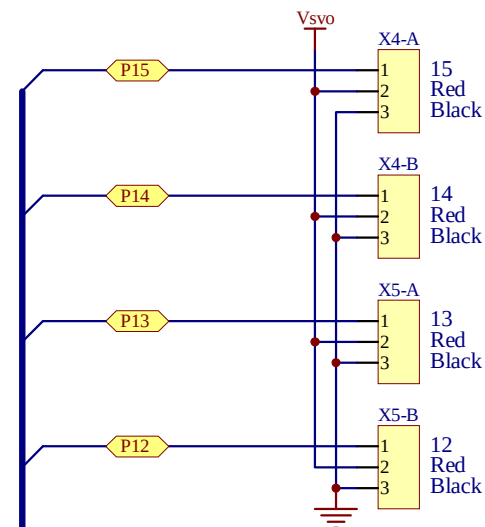
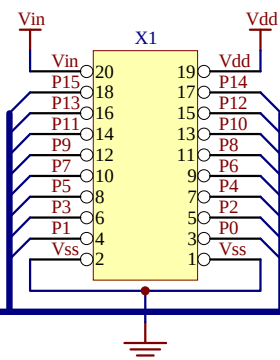
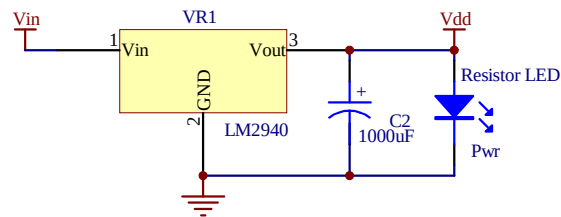
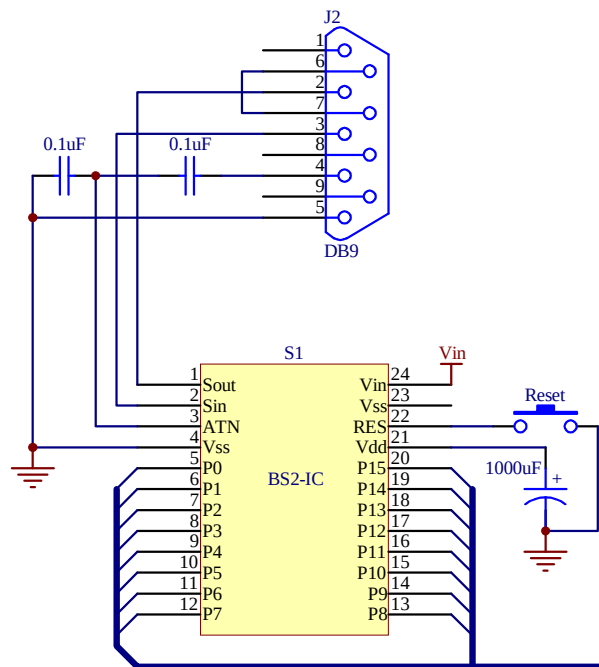
www.Jameco.com ♦ 1-800-831-4242

The content and copyrights of the attached
material are the property of its owner.

Jameco Part Number 282829



Note: The power jack and the battery are mechanically interlocked to ensure they are used mutually exclusively.



Prototype Area

Title		
Board of Education (Serial Version)		
Size	Number	Revision
A		C
Date:	13-Mar-2007	Sheet 1 of 1
File:	C:\sysadm\BOE_C Sch.DDB	Drawn By: John Barrowman

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. 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.

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:	http://www.parallax.com
General e-mail:	info@parallax.com
Tech. e-mail:	support@parallax.com

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 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 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.

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). 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

PREFACE	5
INTRODUCTION TO THE BASIC STAMP	7
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
QUICK START GUIDE	29
USING THE BASIC STAMP EDITOR	35
THE PROGRAMMING ENVIRONMENT	35
COMPILER DIRECTIVES	43
SPECIAL FUNCTIONS	46
SETTING PREFERENCES	55
ADVANCED COMPILATION TECHNIQUES	68
FEATURES FOR DEVELOPERS	75
BASIC STAMP ARCHITECTURE	81
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
BASIC STAMP COMMAND REFERENCE	123
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
APPENDIX A: ASCII CHART	471
APPENDIX B: RESERVED WORDS.....	473
APPENDIX C: CONVERSION FORMATTERS	477
APPENDIX D: BASIC STAMP SCHEMATICS	481

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, 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.

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). Also, Nuts & Volts Magazine (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 (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 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 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 Kuehnelt 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[®] 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 (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 (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, ~80 instructions	2K Bytes, ~500 instructions	8 x 2K Bytes, ~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, 25 µA Sleep	3 mA Run, 50 µA Sleep	25 mA Run, 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 per 8 I/O pins	60 mA / 60 mA per 8 I/O pins
PBASIC Commands*	32	42	45
PC Interface	Serial (w/BS1 Serial Adapter)	Serial (9600 baud)	Serial (9600 baud)
Windows Text Editor Version	Stampw.exe (v2.1 and up)	Stampw.exe (v1.04 and up)	Stampw.exe (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 (32° - 158° F) **	0° - 70° C (32° - 158° F) **	0° - 70° C (32° - 158° F) **	0° - 70° C (32° - 158° F) **	0° - 70° C (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 (6 I/O, 26 Variable)	38 Bytes (12 I/O, 26 Variable)	38 Bytes (12 I/O, 26 Variable)	38 Bytes (12 I/O, 26 Variable)	38 Bytes (12 I/O, 26 Variable)
64 Bytes	128 Bytes	128 Bytes	128 Bytes	128 Bytes
8 x 2K Bytes, ~4,000 inst.	8 x 2K Bytes, ~4,000 inst.	8 x 2K Bytes, ~4,000 inst.	16 x 2K Bytes (16 K for source)	8 x 2K Bytes, ~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, 500 µA Sleep	40 mA Run, 350 µA Sleep	40 mA Run, 350 µA Sleep	15 mA Run, 150 µA Sleep	55 mA Run, 450 µA Sleep
30 mA / 30 mA	30 mA / 30 mA	30 mA / 30 mA	30 mA / 30 mA	30 mA / 30 mA
60 mA / 60 mA per 8 I/O pins	60 mA / 60 mA per 8 I/O pins	60 mA / 60 mA per 8 I/O pins	60 mA / 60 mA per 8 I/O pins	60 mA / 60 mA per 8 I/O pins
45	61	61	61	63
Serial (9600 baud)	Serial (9600 baud)	Serial (9600 baud)	Serial (9600 baud)	Serial (19200 baud)
Stampw.exe (v1.091 and up)	Stampw.exe (v1.1 and up)	Stampw.exe (v1.1 and up)	Stampw.exe (v1.33 and up)	Stampw.exe (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

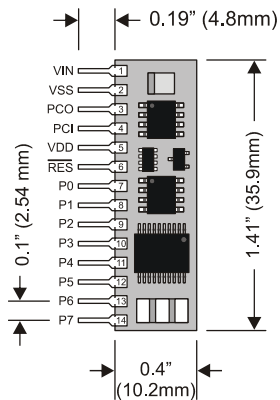


Figure 1.1: BASIC Stamp 1 (Rev B) (Stock# BS1-IC).

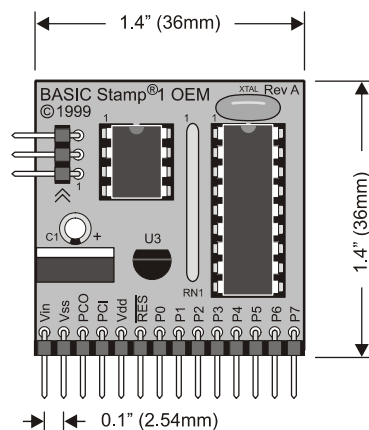
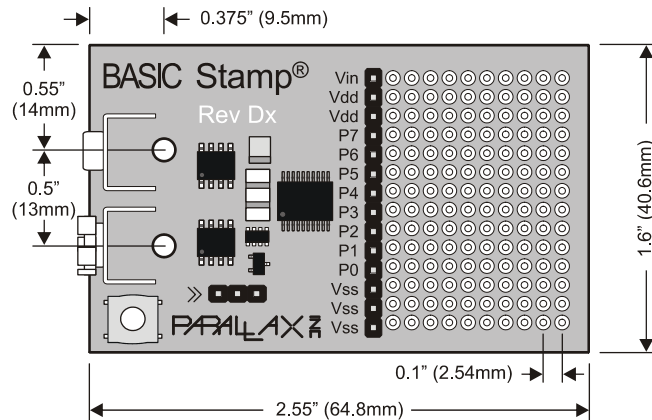


Figure 1.2: BASIC Stamp 1 OEM (Rev. A) (Stock# 27295).

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 → 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).

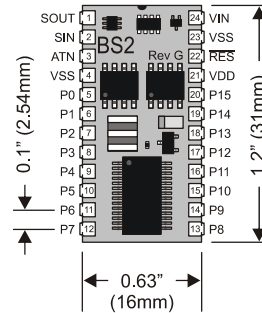
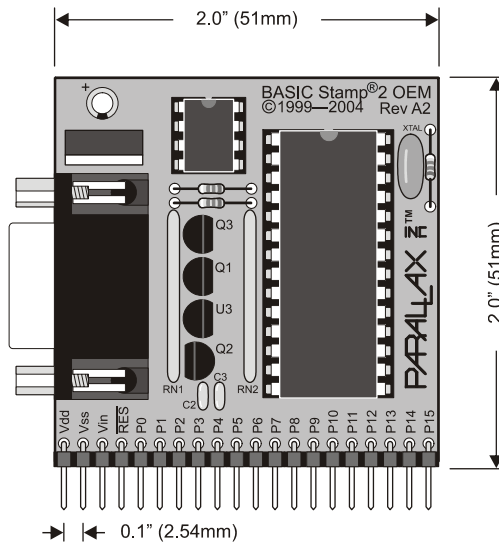


Figure 1.5: BASIC Stamp 2 OEM
(Rev. A2) (Stock# 27290 assembled,
or #27291 in kit form).



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 → 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.

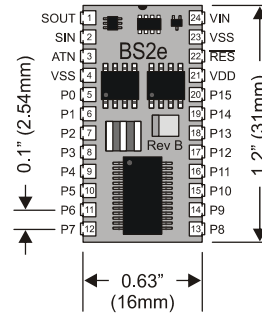
Table 1.2: BASIC Stamp 2 Pin Descriptions.

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)

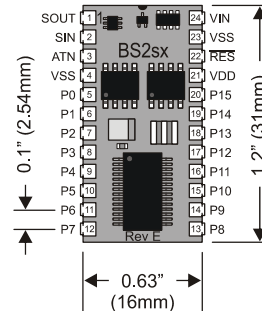
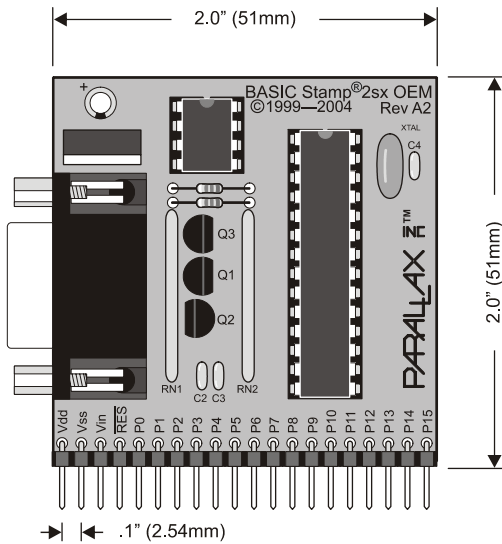


Figure 1.8: BASIC Stamp 2sx OEM
(Rev. A2) (Stock# 27294)



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.

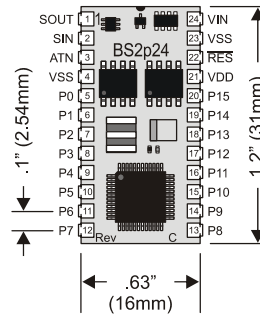
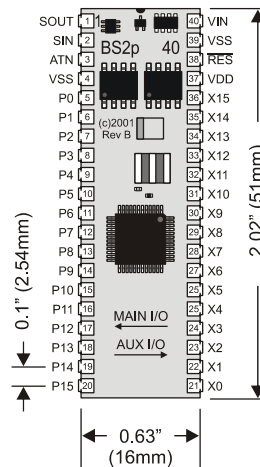


Figure 1.10: BASIC Stamp 2p40
(Rev. B) (Stock# BS2p40-IC)

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.

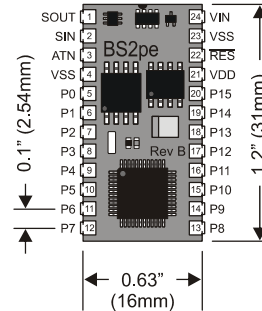
Table 1.5: BASIC Stamp 2p Pin Connections

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.

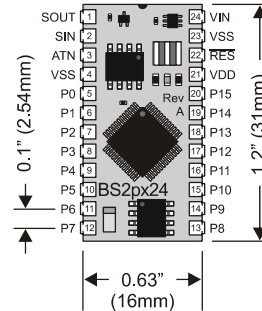
Introduction to the BASIC Stamp

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

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.

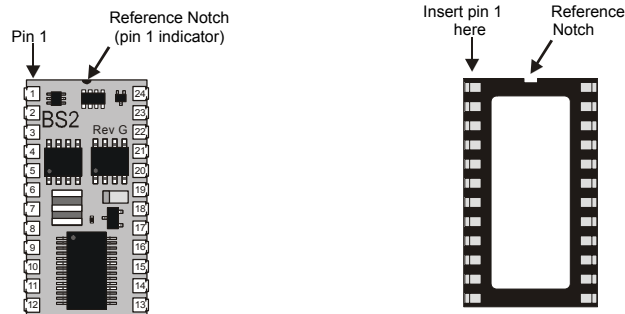


Figure 1.13: Pin 1 Indicators
BS2-IC shown in the correct orientation in relation to a 24-pin socket.

Note: The Half-Circle indicator is also known as a Reference Notch

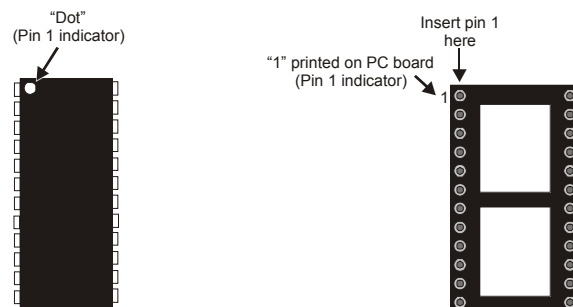


Figure 1.14: Additional Examples of Pin 1 Indicators
(chip and socket shown in the correct orientation in relation to each other)

1: Introduction to the BASIC Stamp

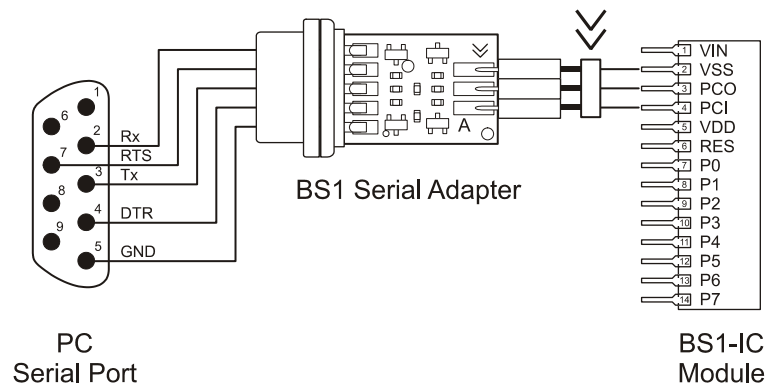
BASIC Stamp Programming Connections

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

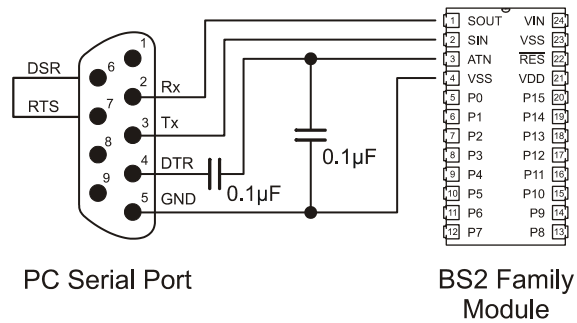


Figure 1.16: Programming and Run-time Communication Connections for all BS2 models.

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 → 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 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.

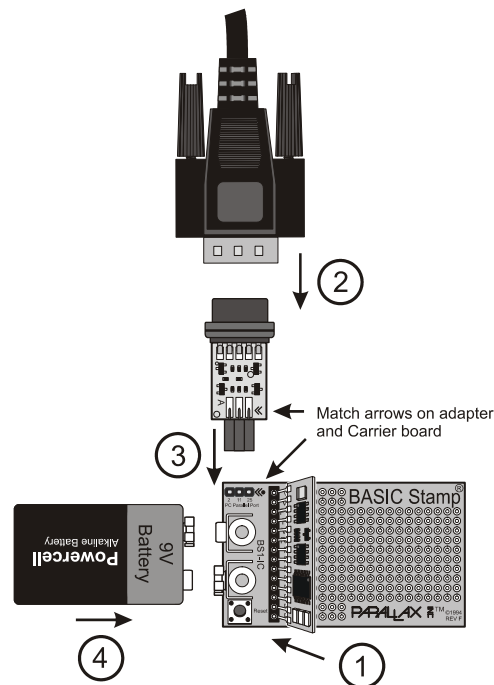


Figure 2.1: BS1-IC, BS1 Carrier Board, and BS1 Serial Adapter

- 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

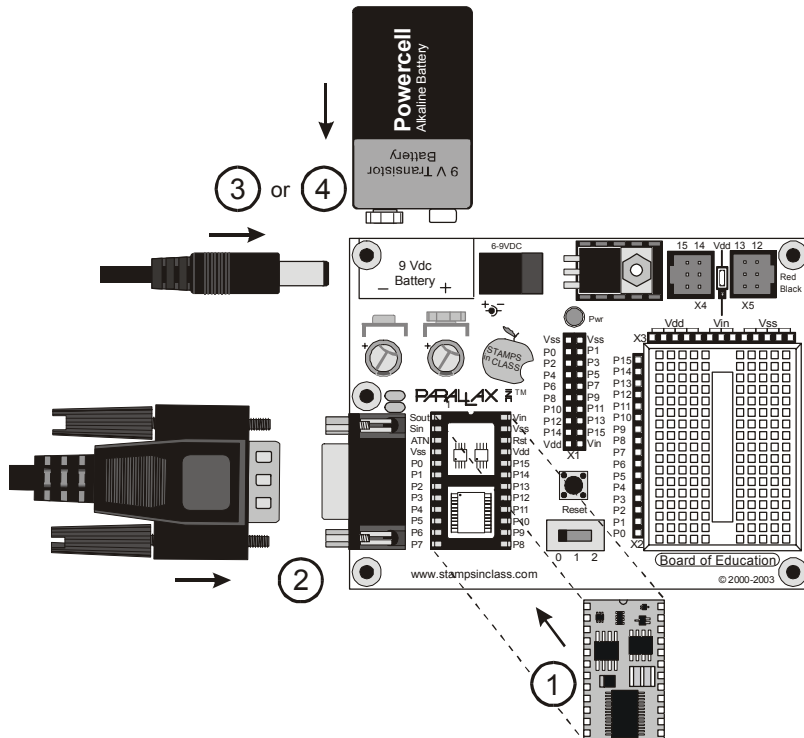
1) Insert the BASIC Stamp module into its socket, being careful to orient it properly.

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.

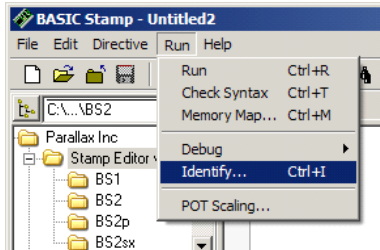
OR

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 → 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

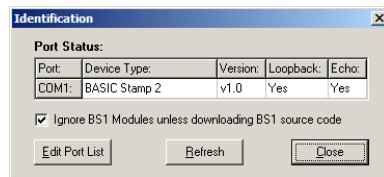


Figure 2.3: Test your PC connection to the BASIC Stamp.

Verify that the BASIC Stamp was detected on one of the COM ports.

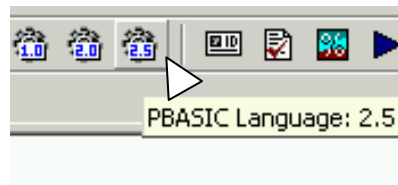
- 4) Enter a \$STAMP Directive into the Editor window by clicking on the toolbar icon for the BASIC Stamp module you are using. (Hold the cursor over the icons for flyover help labels.) The example below shows the Stamp Directive that would be inserted for the BS2.

```
' {$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.

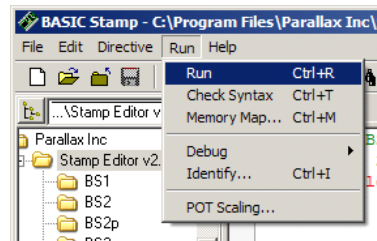
2: Quick Start Guide

- 6) Type the line DEBUG "Hello World!" below the compiler directives:

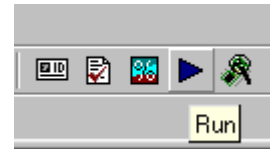
```
' {$STAMP BS2}  
' {$PBASIC 2.5}  
DEBUG "Hello World!"
```

- 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.



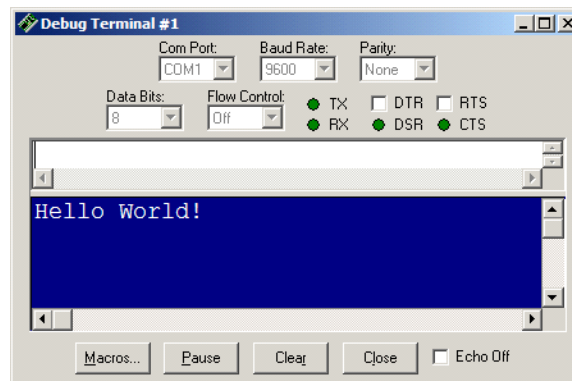
Selecting Run → Run



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

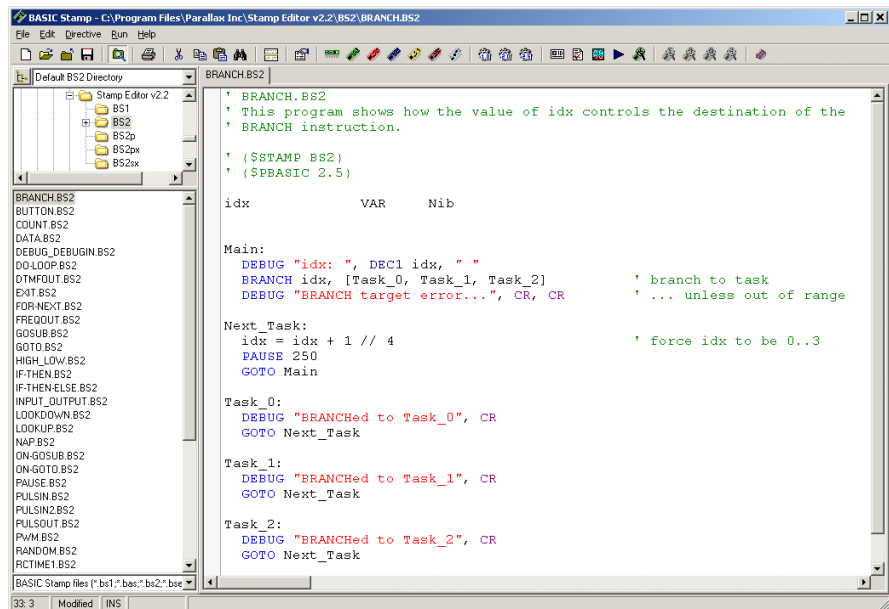
Introducing the BASIC Stamp Editor

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.

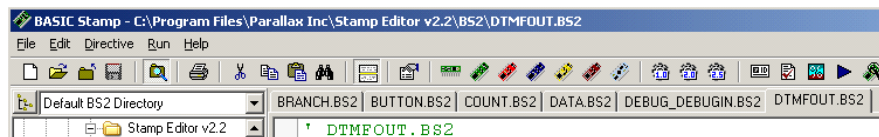


Figure 3.2: Example Editor Tabs. Shown with 6 separate files open; Title Bar shows current code’s file path.

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.

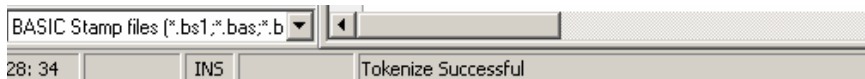


Figure 3.3: Status Bar beneath the Main Edit Pane.

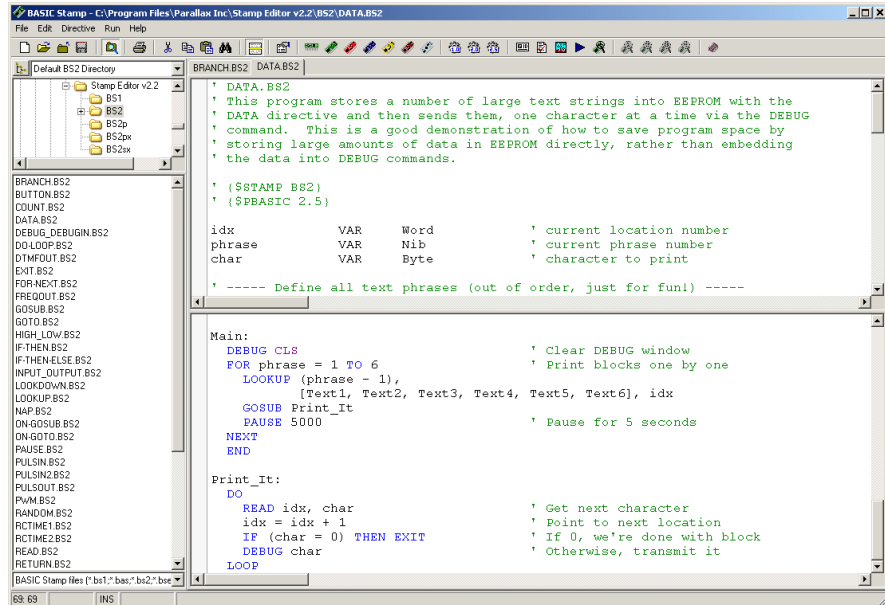
Each editor pane can be individually split into two views of the same source code. This can be done via the Split button on the toolbar, pressing Ctrl-L, or clicking and dragging the top or bottom border of the editor pane with the mouse.

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.

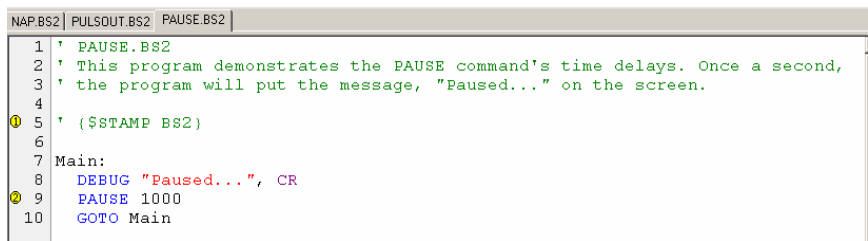


Figure 3.5: Automatic Line Numbering appears in the gutter to the left of the edit pane. Yellow Bookmarks are visible on lines 5 and 9.

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.

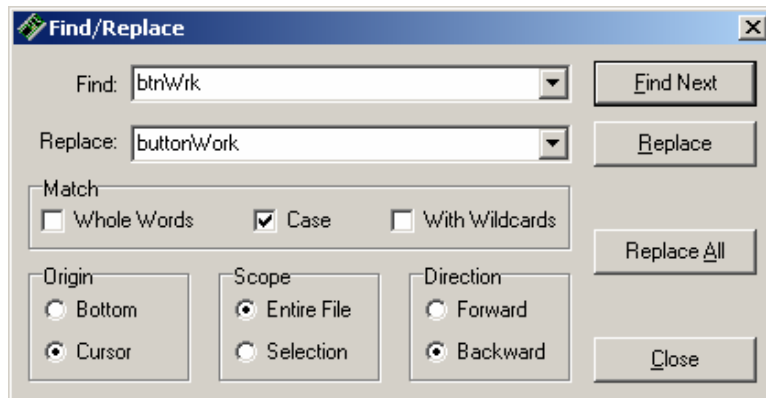


Figure 3.6: The Find/Replace Window.

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.

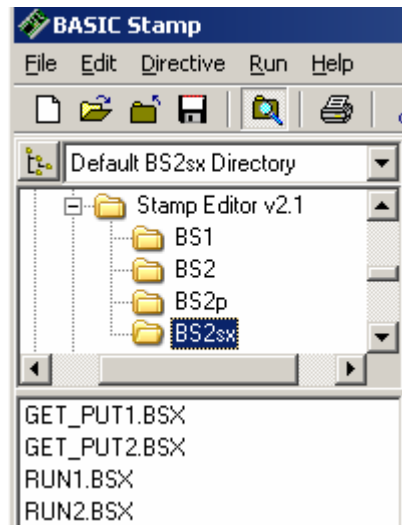
The integrated explorer panel to the left of the main edit pane is divided in to four portions: the Recent, Directory, File and Filter lists. The upper portion is the Recent list, a drop-down list of default, favorite, and recently visited directories.

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

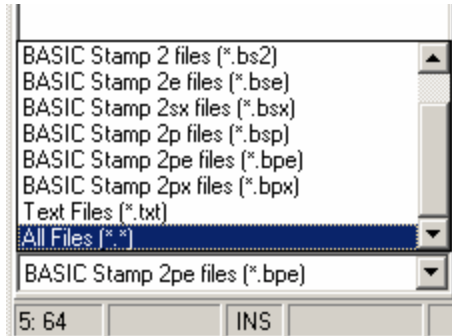


Figure 3.8: The Filter List found at the bottom of the Integrated Explorer Panel.

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.

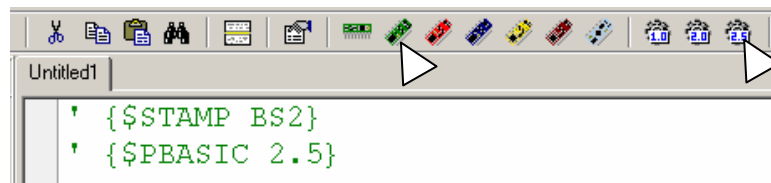
3: Using the BASIC Stamp Editor

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
PBASIC 1.0	32	-	-	-	-	-	-
PBASIC 2.0	-	37	40	40	56	56	58
PBASIC 2.5	-	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.

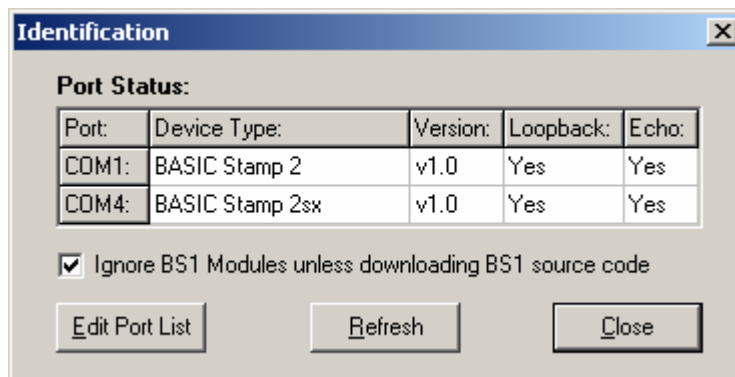


Figure 3.10: The Identification Window.

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.

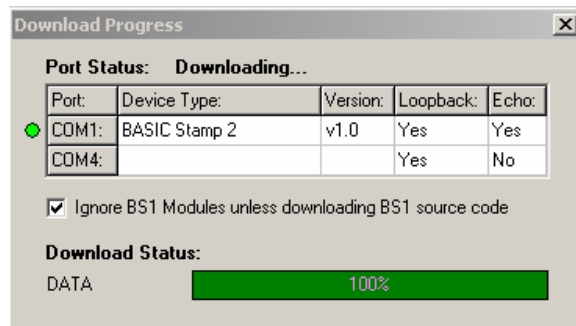
Like the Identification function, the Download function provides information to help guide you through the downloading process. After entering the desired source code in the editor window, you may run it in one of three ways: select Run → Run, press Ctrl+R on the keyboard, or click on the “▶” toolbar icon. This will tokenize and download the code

THE DOWNLOAD FUNCTION.

3: Using the BASIC Stamp Editor

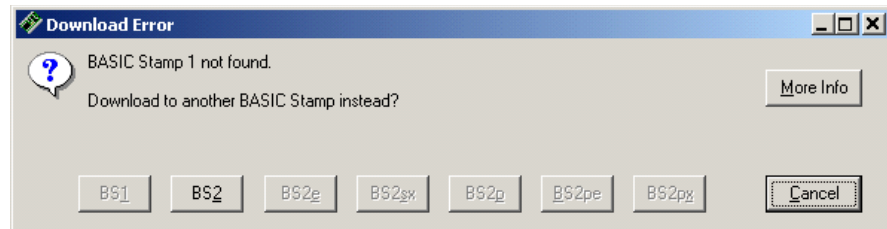
to the BASIC Stamp module (assuming the code is correct and the BASIC Stamp is properly connected). The Download Progress window looks similar to the Identify window with the exception of the additional Download Status progress bar, and the indicator LED by the port transmitting the data.

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.

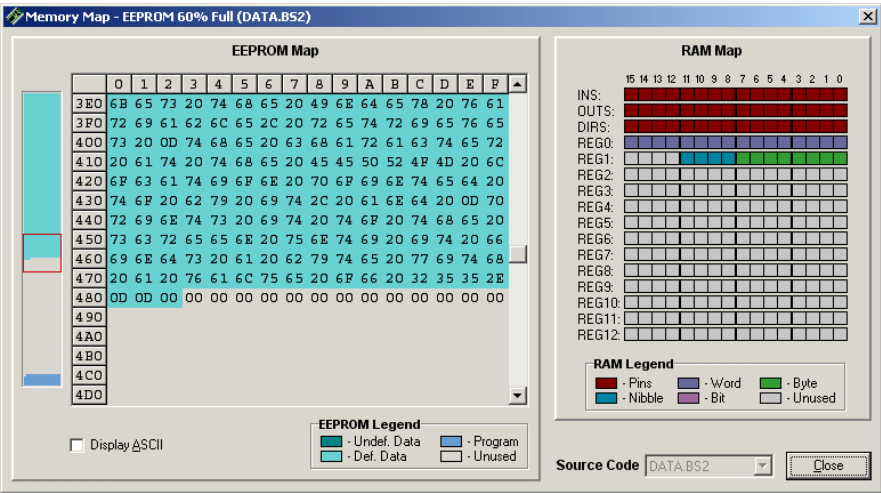


Figure 3.13: Memory Map for Demo Program DATA.bs2.

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

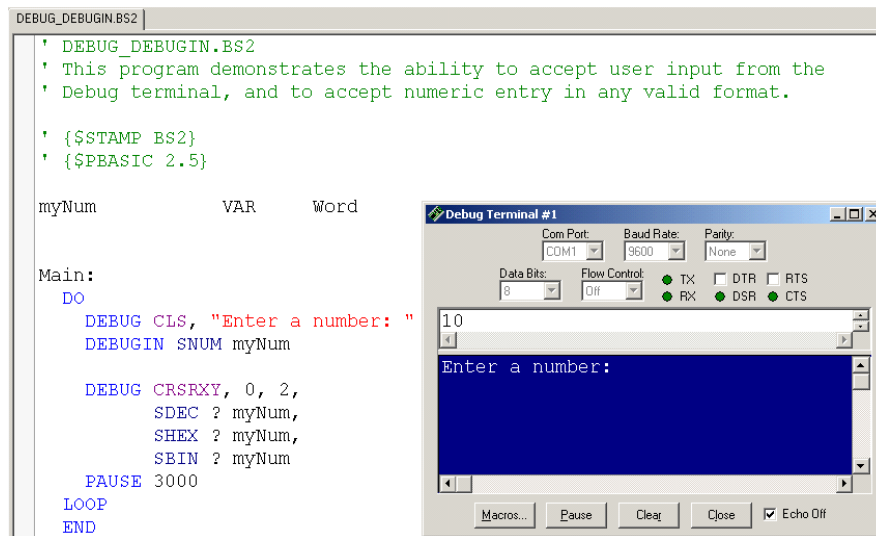


Figure 3.14: Demo program using the Debug Terminal

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).

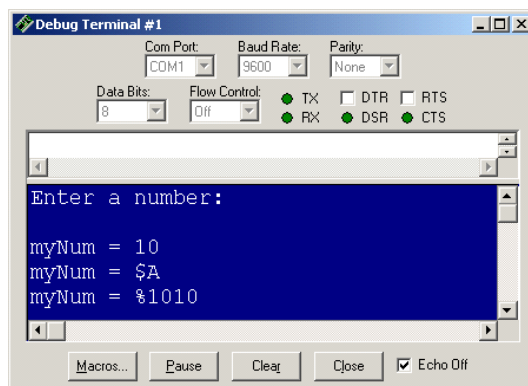


Figure 3.15: Debug Terminal output after entering a number.

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.

There are keyboard shortcuts for several coding functions, some of which are unique to the BASIC Stamp Editor.

Table 3.6: Coding Function Keyboard Shortcuts.

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

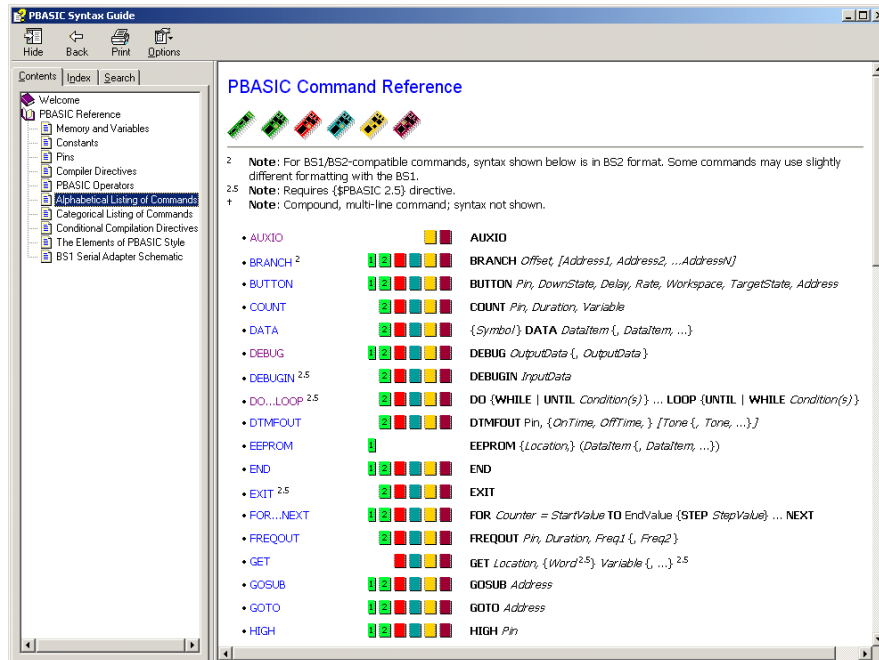


Figure 3.16: The Help file contains the complete PBASIC syntax documentation.

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.

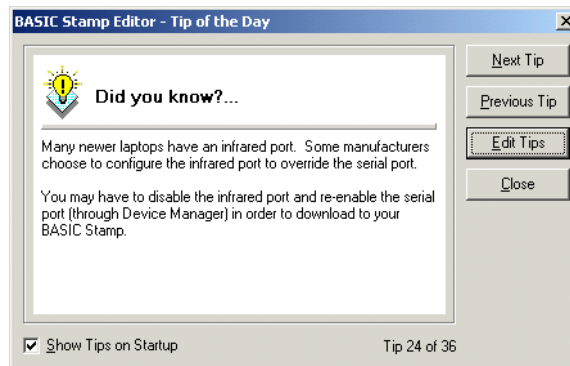
HELP FILES REQUIRE MICROSOFT'S HTML HELP UTILITY.

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.

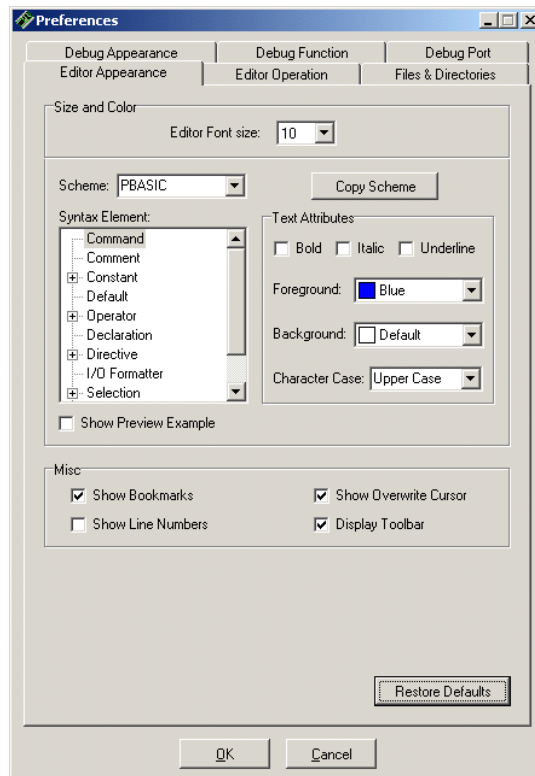


Figure 3.18: The Editor Appearance Tab under Edit → Preferences.

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.

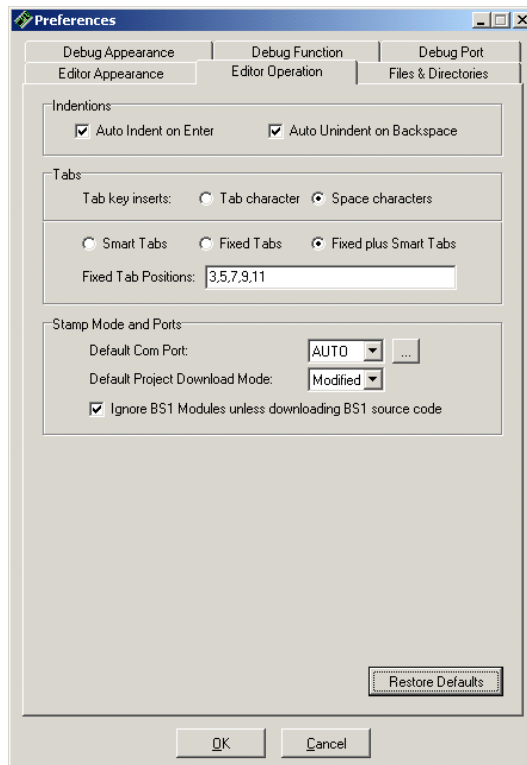


Figure 3.19: The Editor Operation Tab under Edit → Preferences.

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 • 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.

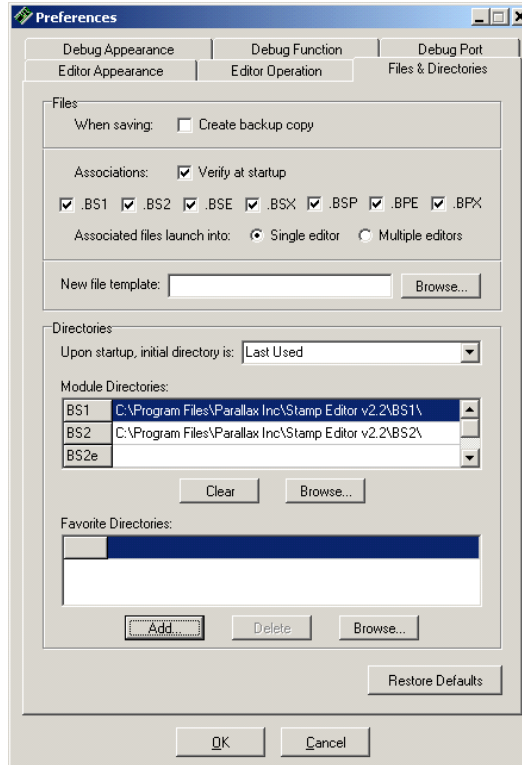
Selecting the “Ignore BS1 Modules unless downloading BS1 source code” checkbox optimizes identification speed by attempting to locate BS1 modules only if you are downloading BASIC Stamp 1 code. This feature can also be activated via the Identification or Download window.

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

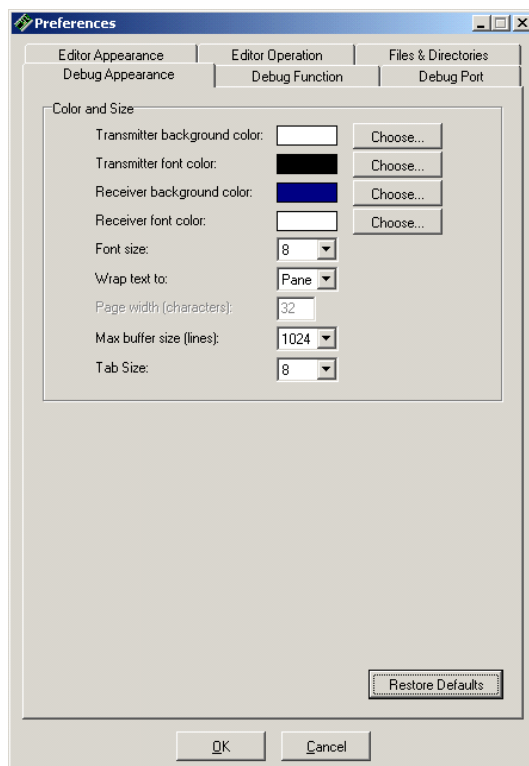


Figure 3.24: The Debug Appearance Tab under Edit → Preferences.

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 1025th 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 2nd line of text that was received. Larger buffer sizes consume more PC memory (256 * buffer_size * 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.

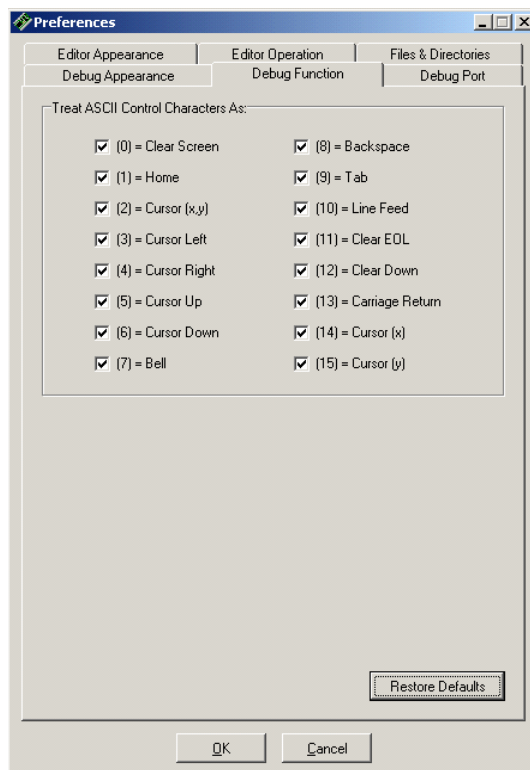


Figure 3.25: The Debug Function Tab under Edit → Preferences.

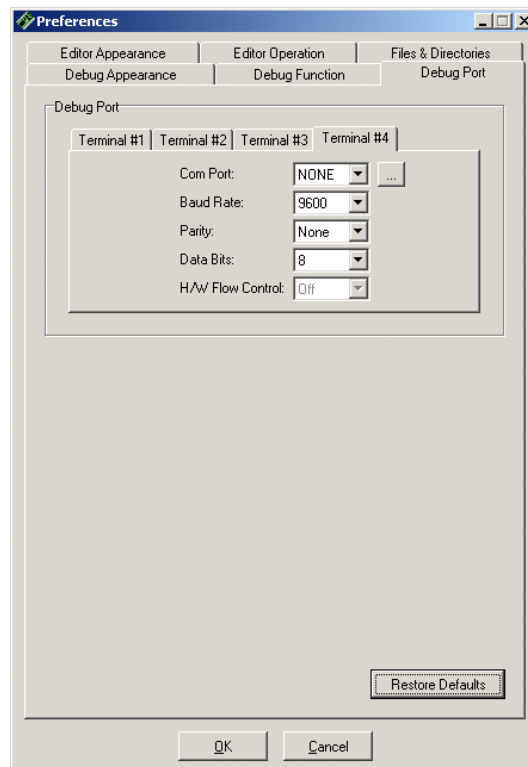
For example, a device that sends out a 0 to indicate something other than Clear Screen will cause unintentional clearing of the Receive pane; unchecking the checkbox for “(0) = Clear Screen” will prevent this from happening.

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:

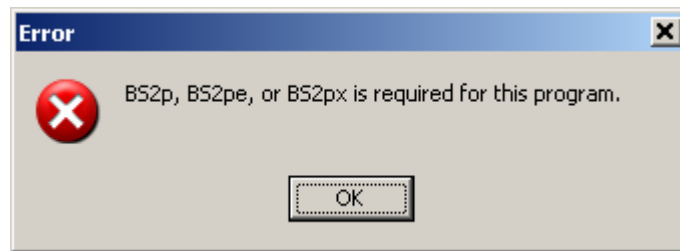
```
' {$PBASIC 2.5}

#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.

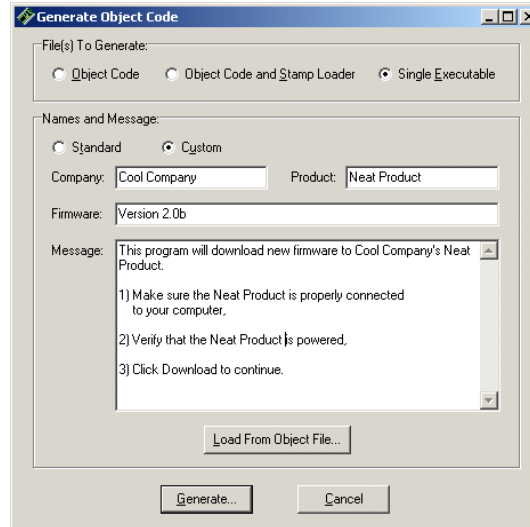
Object code can be saved as a separate .obj file (downloadable with the StampLoader.exe program) or as a single executable (integrated with the StampLoader.exe inside of it). The single executable method provides a simpler way to pass your firmware update on to your customers.

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.

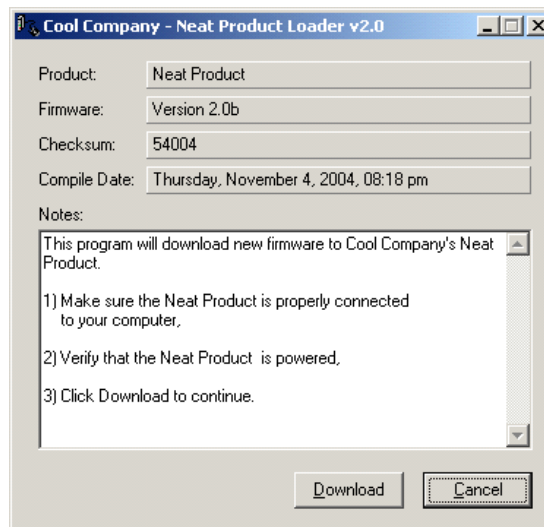
3: Using the BASIC Stamp Editor

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 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.

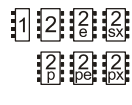
When the output of the BASIC Stamp Editor is piped to a file or a master program, it displays all of its messages in a specific, predefined format. Each message has a unique 3-digit number. Detailed information about the use of command-line options, including a table of all messages with their ID numbers, 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 DIRS	PIN0 – PIN7 DIR0 – DIR7	I/O pins; bit addressable. I/O pins directions; bit addressable.
W0	B0 B1	BIT0 – BIT7 BIT8 – BIT15	Bit addressable. Bit addressable.
W1	B2 B3		
W2	B4 B5		
W3	B6 B7		
W4	B8 B9		
W5	B10 B11		
W6	B12 B13		Used by GOSUB instruction. 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 INC, IND	IN0 – IN7 IN8 – IN15	Input pins
OUTS*	OUTL, OUTH	OUTA, OUTB OUTC, OUTD	OUT0 – OUT7 OUT8 – OUT15	Output pins
DIRS*	DIRL, DIRH	DIRA, DIRB DIRC, DIRD	DIR0 – DIR7 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. Bit 1: Polled-input status; 0 = none defined, 1 = at least one defined. Bit 2: Polled-run status (set by POLLMODE); 0 = disabled, 1 = enabled. Bit 3: Polled-output latch status; 0 = real-time mode, 1 = latch mode. Bit 4: Polled-input state; 0 = no trigger, 1 = triggered. Bit 5: Polled-output latch state; 0 = nothing latched, 1 = signal latched. Bit 6: Poll-wait state; 0 = No Event, 1 = Event Occurred. (Cleared by POLLMODE only). 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
```

CONSTANTS AND COMPILE-TIME EXPRESSIONS.



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:

DEFINING AND USING CONSTANTS (CON).

SYMBOL Name = ConstantValue



-- or --

Name CON ConstantValue



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)

```

 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 ⁿ -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

 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 2ⁿ-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

	All 2 <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>
ENCODER: NCD	All 2 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: <pre>result VAR Word result = %1101 ' Highest bit set is bit 3 DEBUG ? NCD result ' Show the NCD of result (4)</pre>
SINE: SIN	All 2 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.

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

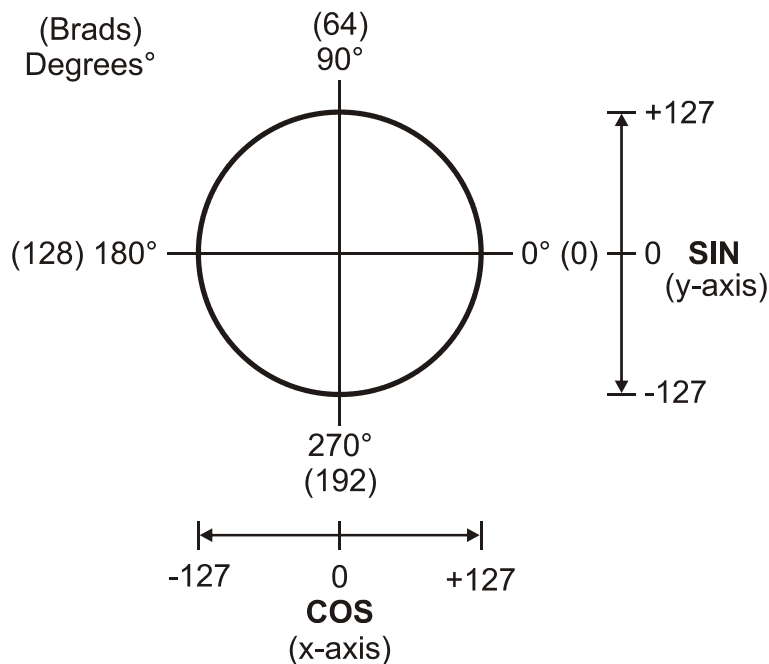


Figure 4.1: 127-Unit Circle

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)
```



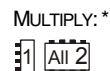
-- Or --

```
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 0.72562×65536 .

```
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:

```
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 (π , 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.

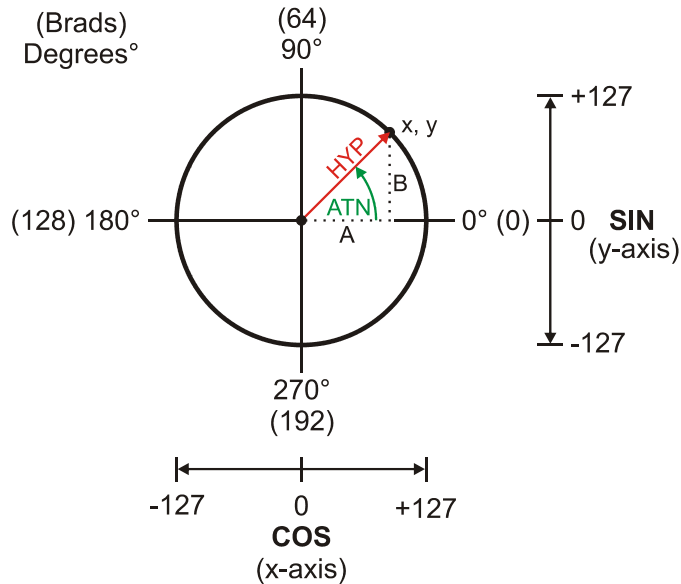
In the BASIC Stamp, the angle is returned in binary radians (0 to 255) instead of degrees (0 to 359). See the explanation of the SIN operator for more information about binary radians. Coordinate input values are limited to -127 to 127 (signed bytes) as shown in the diagram of the PBASIC Unit Circle (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:

MAXIMUM: MAX

```
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^{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^{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 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, DO...LOOP, EXIT, ON. SELECT...CASE	Allows easier user input and program control. See individual command descriptions.
Enhanced Commands	IF...THEN, GET, PUT, 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, #IF...#THEN...#ELSE, #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, CRSRDN, CRSRLF, CRSRRT, CRSRUP, CRSRX, CRSRXY, 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. 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 (^{2.5}).

    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 ^{2.5}	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 ^{2.5}	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 ^{2.5}	Execute code block repetitively, with optional, conditional exit.
 EXIT ^{2.5}	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 ² C serial device.
	I2COUT	Output data to I ² 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 Ω - 50 k Ω 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^{2.5} 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.

- **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



NOTE: Expressions are not allowed as arguments on the BS1. The range of the *Pin* argument on the BS1 is 0 – 7.

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.

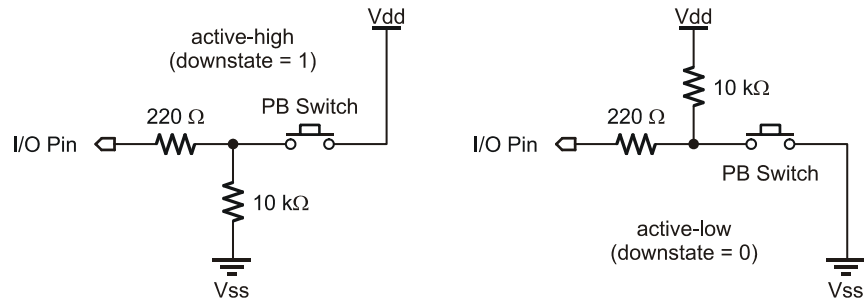
BUTTON is designed for use inside a program loop. Each time through the loop, **BUTTON** checks the state of the specified pin. When it first matches *DownState*, **BUTTON** begins the *Delay* countdown for auto-repeat. Then, in accordance with *TargetState*, it either branches to *Address* (*TargetState* = 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
Mode Values	0: Disables comparator
	1: Enables comparator with P0 as result output
	2: Enables comparator without P0 as result output
Variable Values	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
Mode Values	0 (or SCHMITT): Schmitt Trigger
	1 (or THRESHOLD): Logic Threshold
	2 (or PULLUP): Pull-up Resistor
	3 (or DIRECTION): Output Direction
Related Commands (For DIRECTION Mode)	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 Ω . 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.

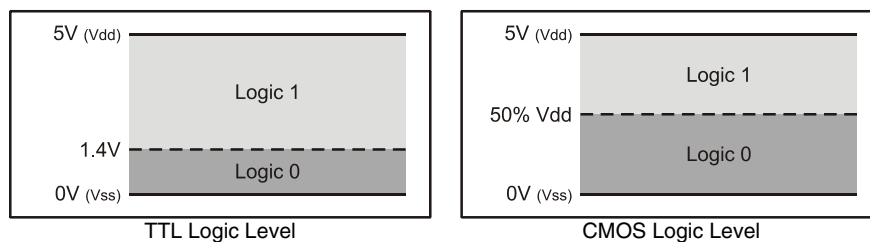


Figure 5.2: TTL and CMOS Logic Level Threshold Voltages

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.

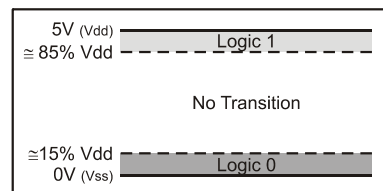
```
CONFIGPIN THRESHOLD, %0000000000001111
```

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,    %1111111111110000 '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
Units in Duration	1 ms	400 μ s	287 μ s	720 μ s	287 μ s
Duration range	1 ms to 65.535 s	400 μ s to 26.214 s	287 μ s to 18.809 s	720 μ s to 47.18 s	287 μ s to 18.809 s
Minimum pulse width	4.16 μ s	1.66 μ s	1.20 μ s	3.0 μ s	1.20 μ s
Maximum frequency (square wave)	120,000 Hz	300,000 Hz	416,700 Hz	166,667 Hz	416,700 Hz
Related Command	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 (μ 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 μ 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 μ 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 BS2p, BS2pe	BS2px
Serial Protocol	Asynchronous 4800, N, 8, 1 True polarity Custom packetized format	Asynchronous 9600, N, 8, 1 Inverted polarity Raw data	Asynchronous 19200, N, 8, 1 Inverted polarity Raw data
Related Commands	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 ^{2.5}	2	Move cursor to specified location. Must be followed by two values (x and then y)
Cursor Left	CRSRLF ^{2.5}	3	Move cursor one character to left.
Cursor Right	CRSRRT ^{2.5}	4	Move cursor one character to right.
Cursor Up	CRSRUP ^{2.5}	5	Move cursor one character up.
Cursor Down	CRSRDN ^{2.5}	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 ^{2.5}	10	Move cursor down one line.
Clear Right	CLREOL ^{2.5}	11	Clear line contents to the right of cursor.
Clear Down	CLRDN ^{2.5}	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 ^{2.5}	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 ^{2.5}	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: (^{2.5}) 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



NOTE: DEBUGIN requires the \$PBASIC 2.5 compiler directive.

DEBUGIN *InputData* { , *InputData* }

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 Inverted Polarity, Raw Data	Asynchronous 19200 baud N, 8, 1 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
```

All 2

NOTE: This example program can be used with all BS2 models by changing the \$STAMP directive accordingly.

5: BASIC Stamp Command Reference – DO...LOOP

All 2

NOTE: DO...LOOP requires the PBASIC 2.5 compiler directive.

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
Default <i>OnTime</i>	200 ms	80 ms	55 ms	196 ms	34 ms
Default <i>OffTime</i>	50 ms	50 ms	50 ms	50 ms	50 ms
Units in <i>OnTime</i>	1 ms	0.4 ms	0.265 ms	1 ms	0.166 ms
Units in <i>OffTime</i>	1 ms	1 ms	1 ms	1 ms	1 ms
Related Command	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

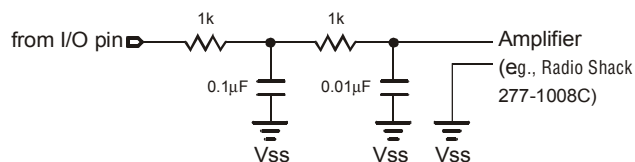
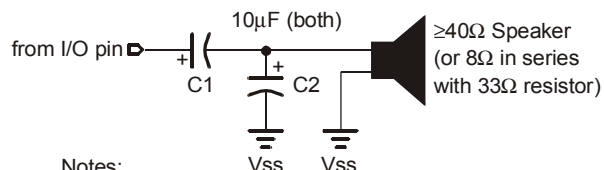


Figure 5.4: Example RC Filter Circuits for Driving an Audio Amplifier (top) or a Speaker (bottom).

Driving a Speaker



Notes:
C1 may be omitted for piezo speakers
C2 is optional, but reduces high-frequency noise

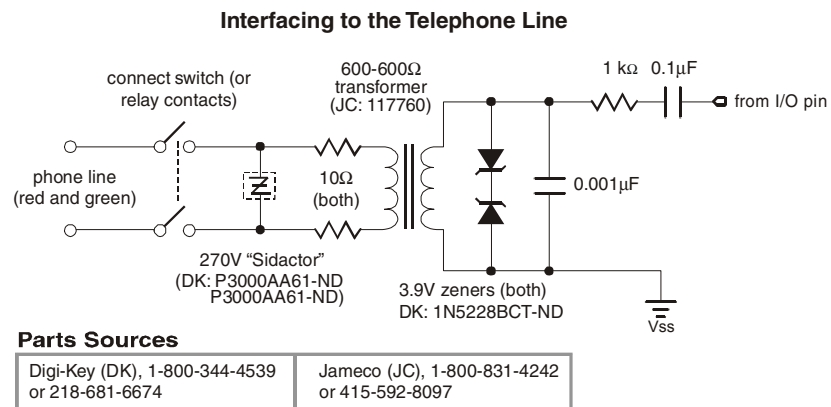
5: BASIC Stamp Command Reference – DTMFOUT

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
Special Notes	Writes values to EEPROM during download. Can be used to decrease program size.
Related Commands	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:

WRITING DATA TO OTHER LOCATIONS.

```
EEPROM    50, (72, 69, 76, 76, 79, 104, 101, 108, 108, 111)
```

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 $\text{INT}(\text{value} / 256)$ and 2) low-byte is $\text{value} - (\text{high-byte} * 256)$.

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

 END	BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

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
Current Draw During Run	1 mA	3 mA	25 mA	60 mA	40 mA	15 mA	55 mA
Current Draw During Sleep	25 μ A	50 μ A	200 μ A	500 μ A	350 μ A	36 μ A	450 μ A
Related Commands	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 μ 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 --

```
All 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
```

All 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:

```
All 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
Units in Duration	1 ms	0.4 ms	0.265 ms	1 ms	0.166 ms
Units in Freq1 and Freq2	1 Hz	2.5 Hz	3.77 Hz	1.51 Hz	6.03 Hz
Range of Frequency	0 to 32767 Hz	0 to 81917 Hz	0 to 123531 Hz	0 to 49478 Hz	0 to 197585 Hz
Related Commands	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

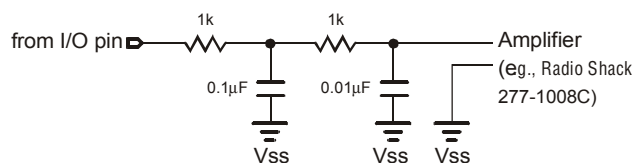
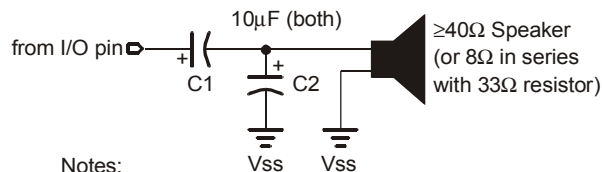


Figure 5.6: Example RC filter circuits for driving an audio amplifier(top) or a speaker (bottom).

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

2 2 2 2 2
e sx p pe px

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.

Explanation

USES FOR SCRATCH PAD RAM.

BASIC Stamp Syntax and Reference Manual 2.2 • www.parallax.com • Page 203

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. Bit 1: Polled-input status; 0 = none defined, 1 = at least one defined. Bit 2: Polled-run status (set by POLLMODE); 0 = disabled, 1 = enabled. Bit 3: Polled-output latch status; 0 = real-time mode, 1 = latch mode. Bit 4: Polled-input state; 0 = no trigger, 1 = triggered. Bit 5: Polled-output latch state; 0 = nothing latched, 1 = signal latched. Bit 6: Poll-wait state; 0 = No Event, 1 = Event Occurred. (Cleared by POLLMODE only). 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:

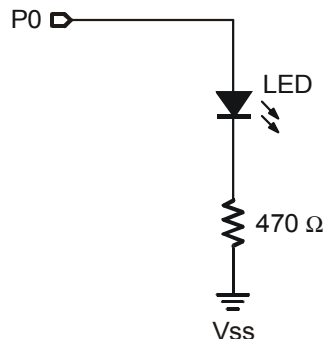


```
OUT6 = 1
DIR6 = 1
```

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 I²C protocol.

- ***Pin*** is a variable/constant/expression (0 or 8) that specifies which I/O pins to use. I²C 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 I²C chip.
- ***Address*** is an optional variable/constant/expression (0 – 255) indicating the desired address within the I²C 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 I²C 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 1: Serial Clock (SCL) pin	8: Serial Data (SDA) pin 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 Ω - 4.7 k Ω pull-up resistors. The I2CIN command does not allow for multiple masters. The BASIC Stamp cannot operate as an I ² C slave device.	
Related Command	I2COUT	

Table 5.32: I2CIN Quick Facts.

Explanation

The I²C 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 I²C devices. The I2CIN command allows the BASIC Stamp to receive data from an I²C 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 I²C 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 I²C device what information it would like to receive. The exact information transmitted (\$A1, 0) depends on the I²C device that is being used.

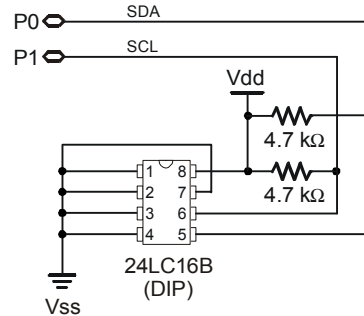
NOTE: The I2C command will make up to 8 attempts to connect to the addressed device. If the device does not properly respond, the I2C command will timeout and the *InputData* will remain unchanged..

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 Ω 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 I²C PROTOCOL FORMAT.

The I²C 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 ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	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 I²C 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 I²C 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 I²C transmission format is shown in Figure 5.10.

I2CIN – BASIC Stamp Command Reference

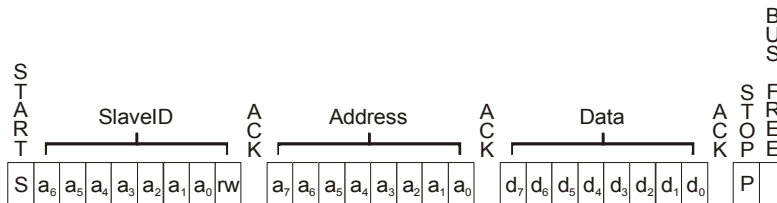


Figure 5.10: I²C Transmission Format.

NOTES:

S = Start Condition

P = Stop Condition

a = id or address bit

d = data bit (transmitted by the BASIC Stamp or the I²C device)

ACK = Acknowledge signal. (Most acknowledge signals are generated by the I²C 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 I²C 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 I²C masters and the BASIC Stamp cannot operate as an I²C 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 I²C protocol.

- **Pin** is a variable/constant/expression (0 or 8) that specifies which I/O pins to use. I²C 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 I²C chip.
- **Address** is an optional variable/constant/expression (0 – 255) indicating the desired address within the I²C 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 I²C 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 1: Serial Clock (SCL) pin	8: Serial Data (SDA) pin 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 ² C slave device.	
Related Command	I2CIN	

Table 5.35: I2COUT Quick Facts.

Explanation

The I²C 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 I²C devices. The I2COUT command allows the BASIC Stamp to send data to an I²C 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 I²C 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.

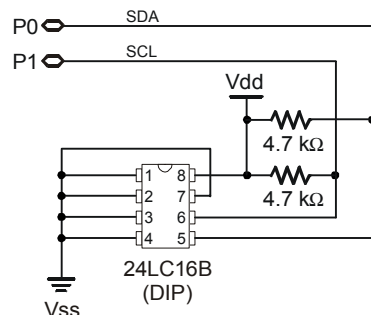


Figure 5.11: Example Circuit for the I2COUT command and a 24LC16B EEPROM.

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> {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 5.37: I2COUT Special Formatters

The I²C 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 I²C PROTOCOL FORMAT.

7	6	5	4	3	2	1	0
A ₆	A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	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,

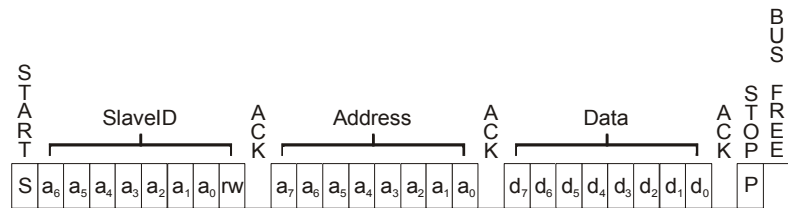
5: BASIC Stamp Command Reference – I2COUT

depending on the I²C 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 I²C 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 I²C transmission format is shown in Figure 5.13.

Figure 5.13: I²C Transmission Format



NOTES:
S = Start Condition
P = Stop Condition
a = id or address bit
d = data bit (transmitted by the BASIC Stamp or the I²C device)
ACK = Acknowledge signal. (Most acknowledge signals are generated by the I²C 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 I²C masters and the BASIC Stamp cannot operate as an I²C 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> ; where <i>Value</i> is a variable or constant	<i>Value1 Comparison Value2</i> ; 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:

```
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:

```
' {$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
    ELSE
      ' generate another random number
    END IF
  LOOP UNTIL hits = 10
```

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
```

INPUT *Pin*

1

Make the specified pin an input.

- ## Quick Facts

	BS1	All BS2 Models
Input Pin Variables	PINS; PIN0 through PIN7	INS; IN0 through IN15
Related Commands	OUTPUT and REVERSE	

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.

All 2

Hold:

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.

BASIC Stamp Syntax and Reference Manual 2.2 • www.parallax.com • Page 243

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

2

p

2

pe

2

px

IOTERM

BS1

BS2

BS2e

BS2sx

BS2p

BS2pe

BS2px

IOTERM Port

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
Values for Port	0 = switch to main I/O group, 1 = switch to auxiliary I/O group.
I/O pin IDs	0 – 15 (after IOTERM command, all references affect physical pins 5 – 20 or 21 – 36 depending on state of <i>Port</i>).
Special Notes	Both the BS2p and BS2pe 24-pin modules accept this command, however, only the BS2p40 gives access to the auxiliary I/O pins.
Related Commands	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 2: LCD Read/Write (R/W) pin 3: LCD Register Select (RS) pin 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 10: LCD Read/Write (R/W) pin 11: LCD Register Select (RS) pin 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.

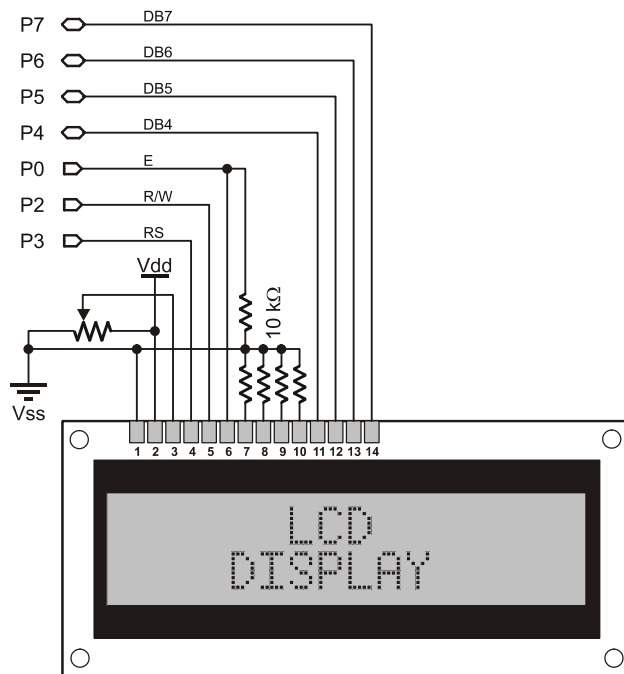


Figure 5.14: Example LCD Circuit. Shown with all connections necessary for the LCDCMD, LCDIN and LCDOUT commands.

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

LDCMD – BASIC Stamp Command Reference

printed on the display (with the LCDOUT command) will appear at the current cursor's location. Here's another example:

```
LDCMD 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 (in decimal)	Description
Do Nothing	0	Don't perform any special operation.
Clear Display	1	Clear the display and move cursor to home position.
Home Display	2	Move cursor and display to home position.
Inc Cursor	6	Set cursor direction to right, without a display shift.
Display Off	8	Turn off display (display data is retained).
Display On	12	Turn on display without cursor (display is restored).
Blinking Cursor	13	Turn on display with blinking cursor.
Underline Cursor	14	Turn on display with underline cursor.
Cursor Left	16	Move cursor left one character.
Cursor Right	20	Move cursor right one character.
Scroll Left	24	Scroll display left one character.
Scroll Right	28	Scroll display right one character.
Move To CGRAM Address	64 + address	Move pointer to character RAM location.
Move To DDRAM Address	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.

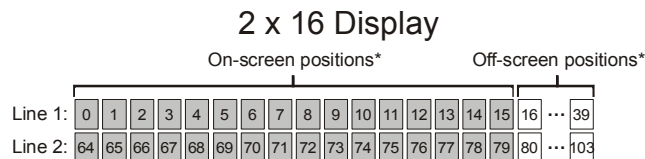
Table 5.45: All LCD Commands (for advanced users). These are supported by LCDs with the Hitachi 44780 controller.

	Command Code (in binary)								Description
	7	6	5	4	3	2	1	0	
Clear Display	0	0	0	0	0	0	0	1	Clear entire display and move cursor home (address 0).
Home Display	0	0	0	0	0	0	1	0	Move cursor home and return display to home position.
Entry Mode	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)
Display/Cursor	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)
Scroll Display / Shift Cursor	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).
Function Set	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)
Move To CGRAM Address	0	1	A	A	A	A	A	A	Move pointer to character RAM location specified by address (A)
Move To DDRAM Address	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 2: LCD Read/Write (R/W) pin 3: LCD Register Select (RS) pin 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 10: LCD Read/Write (R/W) pin 11: LCD Register Select (RS) pin 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 2: LCD Read/Write (R/W) pin 3: LCD Register Select (RS) pin 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 10: LCD Read/Write (R/W) pin 11: LCD Register Select (RS) pin 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 <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> <i>L</i>	Send a string consisting of <i>Byte</i> repeated <i>L</i> 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

	Bits						
	4	3	2	1	0	Binary Values	Decimal Values
Byte 0:	☐	☐	☐	☐	☐	→ xxx00000	00
Byte 1:	☐	☒	☐	☒	☐	→ xxx01010	10
Byte 2:	☐	☐	☐	☐	☐	→ xxx00000	00
Byte 3:	☐	☐	☒	☐	☐	→ xxx00100	04
Byte 4:	☒	☐	☐	☐	☒	→ xxx10001	17
Byte 5:	☐	☒	☒	☒	☐	→ xxx01110	14
Byte 6:	☐	☐	☐	☐	☐	→ xxx00000	00
Byte 7:	☐	☐	☐	☐	☐	→ xxx00000	00

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

```
LCD CMD Lcd, %00110000      ' send wakeup sequence to LCD
PAUSE 5                    ' pause required by LCD specs
LCD CMD Lcd, %00110000
PAUSE 0                    ' pause required by LCD specs
LCD CMD Lcd, %00110000
PAUSE 0                    ' pause required by LCD specs
LCD CMD Lcd, %00100000      ' set data bus to 4-bit mode
LCD CMD Lcd, %00101000      ' set to 2-line mode with 5x8 font
LCD CMD Lcd, %00001100      ' display on without cursor
LCD CMD 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
    LCD CMD 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}

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
```

All 2

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)

```
' 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
```

All 2

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
  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:

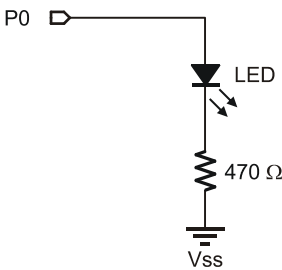


```
OUT6 = 0  
DIR6 = 1
```

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.

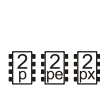
' {$STAMP BS2}

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



MAINIO

MAINIO

BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

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

	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 μA	50 μA	200 μA	500 μA	350 μA	36 μA	450 μA
Related Commands	END and SLEEP				END, SLEEP, and POLLWAIT		
Accuracy of Nap	-50 to 100% (±10% @ 75°F with stable power supply)						

Table 5.59: NAP Quick Facts.

Note: Current measurements are based on 5-volt power, no extra loads, and 75°F ambient temperature.

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 ± 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 ± 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 μ 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 μ s on the BS2pe). When the interpreter firmware regains control of the processor, it restores the I/O direction dictated by your program.

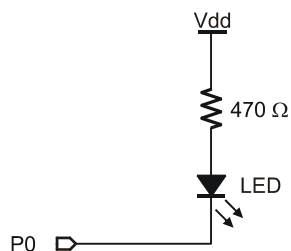
TIPS FOR DRIVING LOADS DURING NAP.

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

Table 5.62: OUTPUT 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
2	2	2	OWIN <i>Pin</i> , <i>Mode</i> , [<i>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 Ω 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.

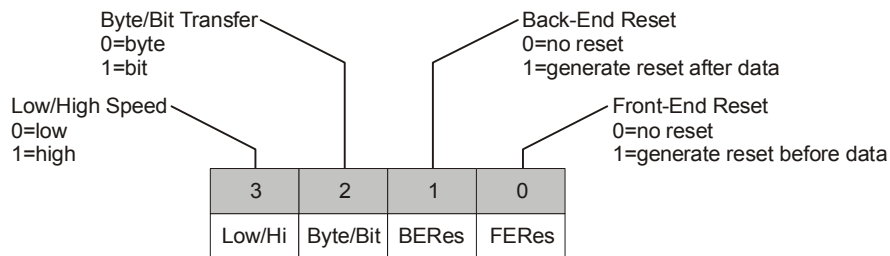


Figure 5.19: Mode Format.

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.

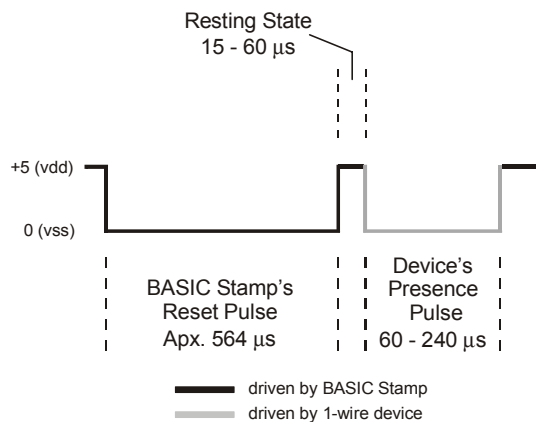
5: BASIC Stamp Command Reference – OWIN

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.

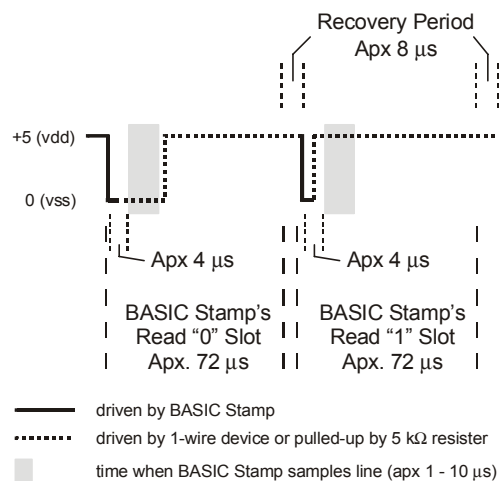


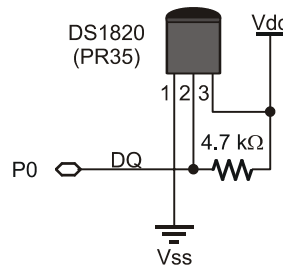
Figure 5.21: Example Read Slot.

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 μ 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, CvtTmp]      ' 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
Transmission 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 Ω pull-up resistor. The BS2pe is not capable of high-speed transfers.
Related Command	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.

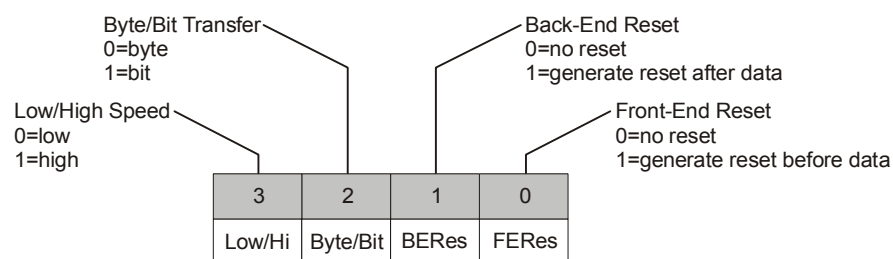


Figure 5.23: MODE Format.

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.

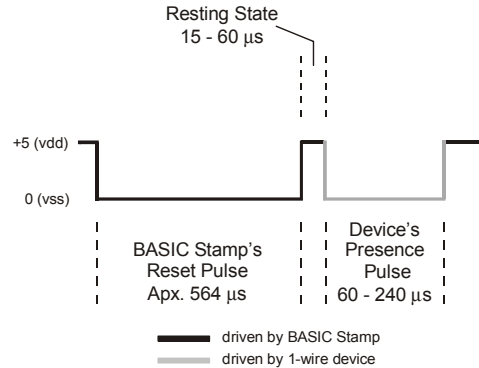
- 1 Fixed-digit formatters like DEC4 will pad the number with leading 0s if necessary; ex: DEC4 65 sends 0065. If a number is larger than the specified number of digits, the leading digits will be dropped; ex: DEC4 56422 sends 6422.
- 2 Signed modifiers work under two's complement rules.
- 3 The HEX 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 μs wide. Figure 5.25 shows typical Write Slots performed by the BASIC Stamp. See the OWIN command for information on Read Slots.

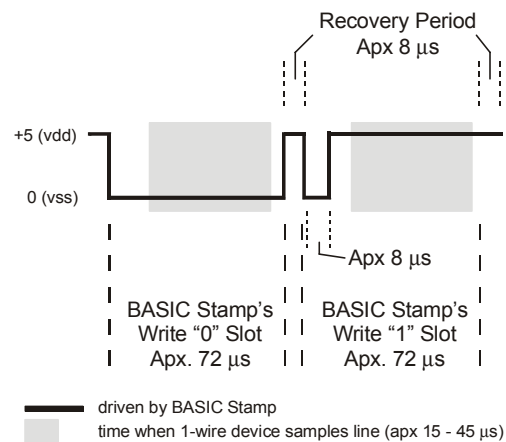


Figure 5.25: Example Write Slots.

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.

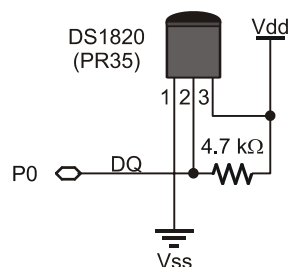


Figure 5.26: DS1820 Circuit.
NOTE: The 4.7 $\text{k}\Omega$ resistor is required for proper operation.

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), ± 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
Available actions in response to reaching the desired State	1) Nothing, 2) Set polled-output pins to a specified state, 3) Run another program (in a specified program-slot), 4) Wait (pause program execution) until desired <i>State</i> is reached, 5) Any combination of 2, 3 and 4, above.
Special notes	• The polled-input pins are monitored (polled) in-between each command within the PBASIC code. • 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.
Useful SPRAM locations	Locations 128 – 135 hold polled interrupt status. See Table 5.77 in the POLLMODE command section for more information.
Related commands	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
Special Notes	• Polled-output pins will either change states continuously, just once or not at all, depending on the POLLMODE command. • A poll-mode of 2 or 4 is required for a POLLWAIT command to work. • If both polled-outputs and polled-run are active, the polled-output event will occur before the polled-run event.
Useful SPRAM Locations	Locations 128 – 135 hold polled interrupt status. See Table 5.77 for more information.
Related Commands	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 ¹	Activate polling with polled-run action only.
4 ²	Activate polling with polled-output/polled-wait and polled-run actions.
5 ³	Clear polled-input configuration.
6 ³	Clear polled-output configuration.
7 ³	Clear polled-input and output configuration.
8 – 15	Same as 0 – 7 except polled-output states are latched.

Table 5.76: POLLMODE Mode Values.

¹ After the polled-run action occurs, the mode switches to 1 (deactivated, saved)

² After the polled-run action occurs, the mode switches to 2 (activated, outputs)

³ 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. Bit 1: Polled-input status; 0 = none defined, 1 = at least one defined. Bit 2: Polled-run status (set by POLLMODE); 0 = disabled, 1 = enabled. Bit 3: Polled-output latch status; 0 = real-time mode, 1 = latch mode. Bit 4: Polled-input state; 0 = no trigger, 1 = triggered. Bit 5: Polled-output latch state; 0 = nothing latched, 1 = signal latched. Bit 6: Poll-wait state; 0 = No Event, 1 = Event Occurred. (Cleared by POLLMODE only). 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
Special Notes	• 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. • Polled-output pins will either change states continuously, just once or not at all, depending on the POLLMODE command. • 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. • If both polled-outputs and polled-run are active, the polled-output event will occur before the polled-run event.
Useful SPRAM Locations	Locations 128 – 135 hold polled interrupt status. See Table 5.77 in the POLLMODE command section for more information.
Related Commands	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
Default ProgramSlot	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.
Special Notes	• If both polled-outputs and polled-run are active, the polled-output event will occur before the polled-run event.
Useful SPRAM locations	Locations 128 – 135 hold polled interrupt status. See Table 5.77 in the POLLMODE command section for more information.
Related commands	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
Current draw during POLLWAIT	350 μ A	36 μ A	450 μ A
Response Time with <i>Duration</i> set to 8	Less than 160 μ s	Less than 250 μ s	Less than 100 μ s
Accuracy of NAP	–50 to 100% ($\pm 10\%$ @ 75°F with stable power supply)		
Special Notes	• Poll mode must be 2 or 4 and at least one polled-input must be set to activate POLLWAIT (POLLWAIT will be ignored otherwise). • If both polled-wait and polled-run are active, the polled-run event will occur immediately after the polled-wait detects an event.		
Useful SPRAM Locations	Locations 128 – 135 hold polled interrupt status. See Table 5.77 in the POLLMODE command section for more information.		
Related Commands	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

1

POT Pin, Scale, Variable

All 2

(See RCTIME)

Function

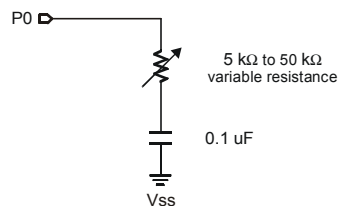
Read a 5 k Ω to 50 k Ω potentiometer, thermistor, photocell, or other variable resistance.

- **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 μ s	2 μ s	2 μ s	0.8 μ s	0.8 μ s	2 μ s	0.81 μ 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.

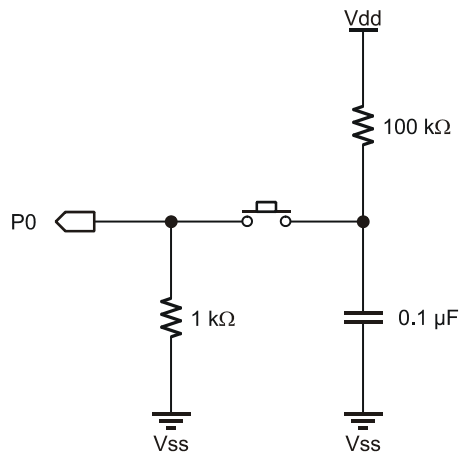


Figure 5.28: R/C Pulse Generator.

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
Duration units	10 μ s	2 μ s	2 μ s	0.8 μ s	0.8 μ s	2 μ s	0.8 μ s
Maximum Pulse Width	655.35 ms	131.07 ms	131.07 ms	52.428 ms	52.428 ms	131.07 ms	52.428 ms
Related Command	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 μ 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 k Ω 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.

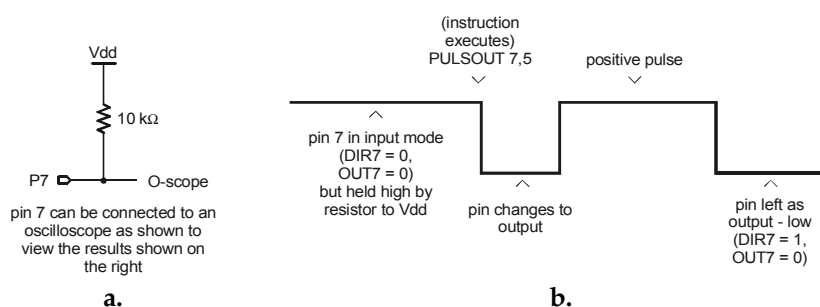


Figure 5.29: Example Pulse Diagram.

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 (10 μ s if using a BS2) and then inverts it again, leaving the pin in its original state.

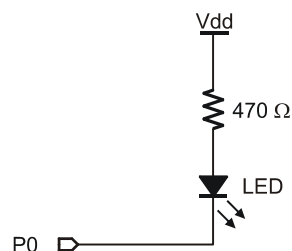


Figure 5.30: LED Circuit for PULSOUT Demo Programs.

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
Scratch Pad RAM Size and Organization	64 bytes (0 – 63). Organized as bytes only.	136 bytes (0 – 135). Organized as bytes only.
General Purpose Locations	0 - 62	0 – 126
Special Use Location	Location 63: Active program slot number (read only).	Location 127: READ/WRITE slot and Active Program slot (read only). Locations 128-135: Polled Interrupt status (read only).
Related Commands	GET	GET and STORE, and SPSTR formatter.
PBASIC 2.5 Syntax Options	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.

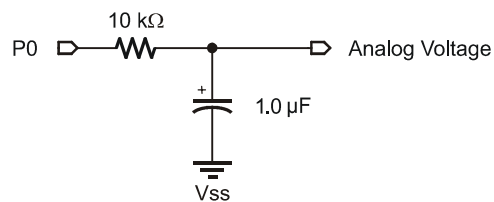


Figure 5.31: Example PWM Filter Circuit.

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 (10×10^3 ohm) resistor and a 1 μF (1×10^{-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 μA 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 μs 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.

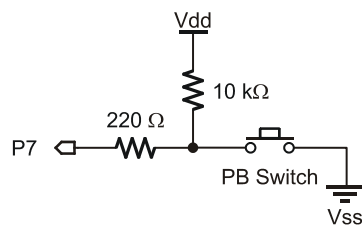


Figure 5.32: RANDOM Button Circuit.

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 μ s	2 μ s	0.8 μ s	0.75 μ s	2 μ s	0.75 μ 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.

SUITABLE RCTIME CIRCUITS.

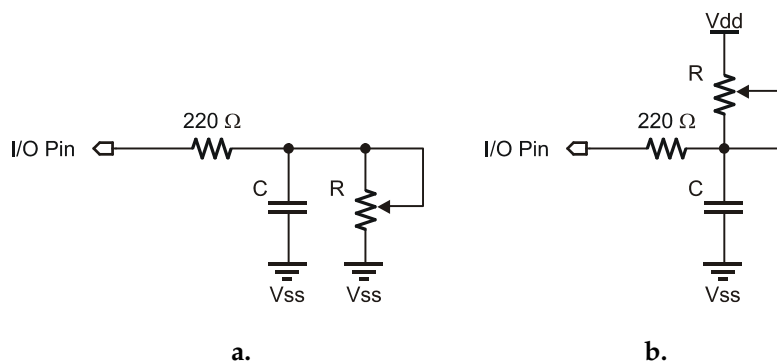


Figure 5.33: Example RC Circuits. Use **a** (left) with *State* = 1. Use **b** (right) with *State* = 0.

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
PAUSE 1
RCTIME 7, 1, result
DEBUG DEC ? result
```

' charge the cap
' for 1 ms
' measure RC discharge time
' 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 (τ) 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 τ 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 τ 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 μF cap. Calculate τ :

$$\tau = (10 \times 10^3) \times (0.1 \times 10^{-6}) = 1 \times 10^{-3}$$

The RC time constant is 1×10^{-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 (1.273×10^{-3}) works out to 636 units. With a 10 k Ω resistor and 0.1 μF cap, RCTIME would return a value of approximately 635. Since V_{initial} and V_{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 Ω series resistor and the capacitor. So if the capacitor were 0.1 μF , 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 μ 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 Ω resistor is necessary at all. Consider what would happen if resistor R in Figure 5.33a were a pot, and were adjusted to 0 Ω . When the I/O pin went high to charge the cap, it would see a short direct to ground. The 220 Ω 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 Ω 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

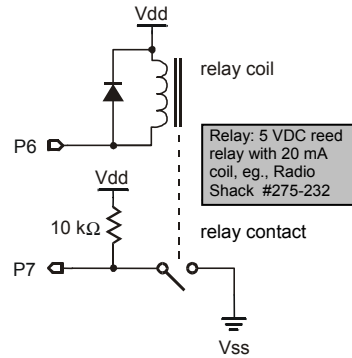
Main:
  DO
    HIGH RC          ' charge the cap
    PAUSE 1           ' for 1 ms
    RCTIME RC, 1, result ' measure RC discharge time
    DEBUG HOME, DEC result ' display value
    PAUSE 50
  LOOP
END
```

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
Range of EEPROM Locations	0 to 255	0 to 2047	0 to 2047 (see notes below)
Special Notes	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.
Related Commands	WRITE and EEPROM	WRITE and DATA	WRITE, DATA and STORE
PBASIC 2.5 Syntax Options	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
```

```
READ 0, result.LOWBYTE      'read low byte  
READ 1, result.HIGHBYTE     'read high byte
```



NOTE: this method is required only if using PBASIC 2.0. See section below for PBASIC 2.5 method.

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

 All	RETURN						
	BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px

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

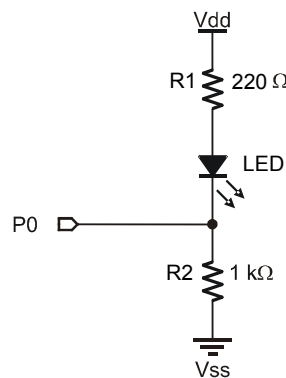
Explanation

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

RUN	BS1	BS2	BS2e	BS2sx	BS2p	BS2pe	BS2px
 RUN ProgramSlot							

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
Comparison Operators	=, <>, >, <, >=, <=
Conditional Logic Operators	Not allowed, however multiple conditions can be separated by commas (,) which act like a logical OR operator.
Format of <i>Condition</i>	Comparison <i>Value</i> where <i>Value</i> can be any of variable, constant or expression
Parentheses	Allowed to specify order of execution within expressions.
Maximum Nested SELECT Statements	16
Maximum CASEs per SELECT	16
Related Commands	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 ± 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.

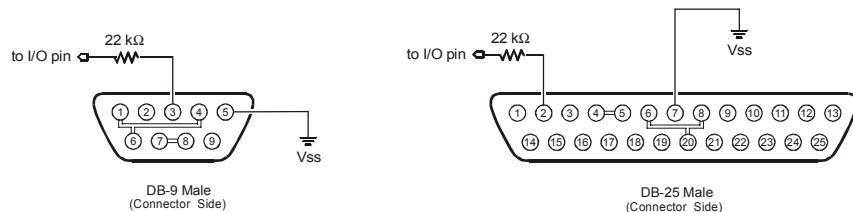
ALL 2
USING THE BUILT-IN SERIAL PORT ON ALL
BS2 MODELS.

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.

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 Ω 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 Ω 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. Table 5.95 indicates the *Baudmode* value or symbols to use when selecting the desired mode. 1

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). 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. All 2

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.

Step 1: Determine the bit period (bits 0 – 11).	BS2, BS2e and BS2pe: = INT(1,000,000 / baud rate) – 20 BS2sx and BS2p: = INT(2,500,000 / baud rate) – 20 BS2px: = INT(4,000,000 / baud rate) – 20 Note: INT means 'convert to integer;' drop the numbers to the right of the decimal point.
Step 2: Set data bits and parity (bit 13).	8-bit/no-parity = 0 7-bit/even-parity = 8192
Step 3: Select polarity (bit 14).	True (noninverted) = 0 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.

```
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:



```
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:



```
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.

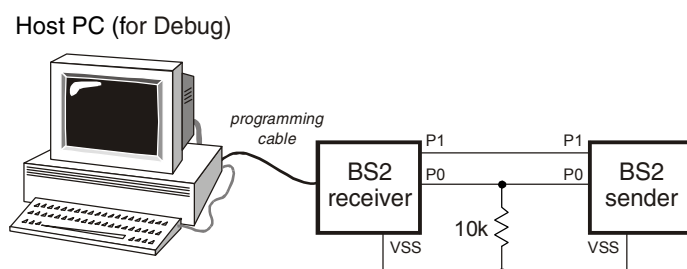


Figure 5.37: Flow-Control Example Circuit.

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.

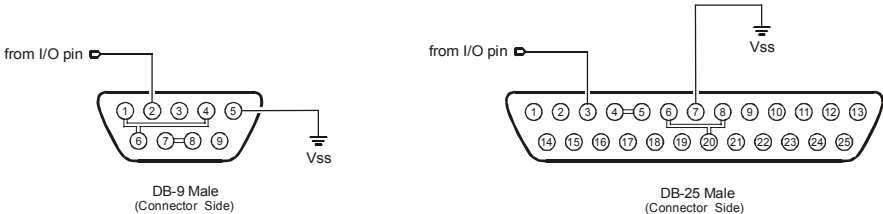


5: BASIC Stamp Command Reference – SEROUT

 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.

Step 1: Determine the bit period (bits 0 – 11)	BS2, BS2e and BS2pe: = INT(1,000,000 / baud rate) – 20 BS2sx and BS2p: = INT(2,500,000 / baud rate) – 20 BS2px: = INT(4,000,000 / baud rate) – 20 Note: INT means 'convert to integer;' drop the numbers to the right of the decimal point.
Step 2: Set data bits and parity (bit 13)	8-bit/no-parity = 0 7-bit/even-parity = 8192
Step 3: Select polarity (bit 14)	True (noninverted) = 0 Inverted = 16384
Step 4: Select driven or open output (bit 15)	Driven = 0 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:

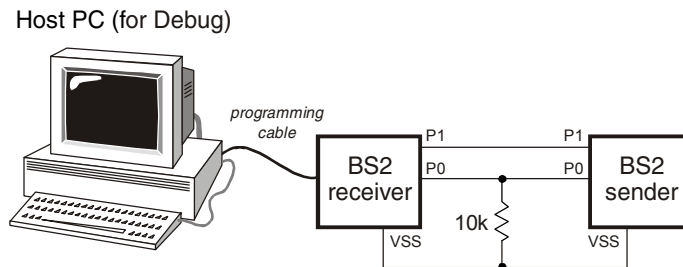


```
SEROUT 1\0, 16468, 2000, No_Permission, [DEC 150]  
STOP
```

```
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.

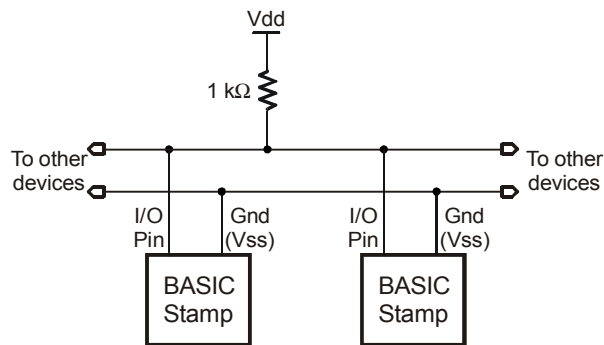
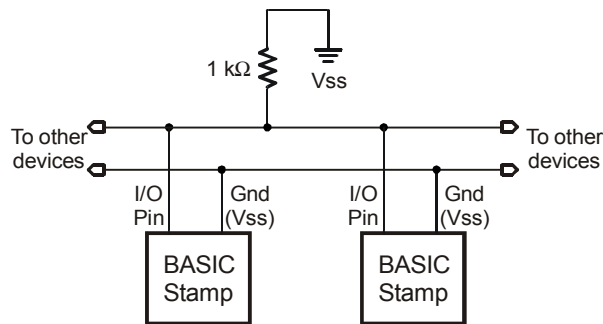


Figure 5.40: SEROUT Open-Drain Circuit. This circuit is for use with the Open, Non-inverted baudmode.

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
Timing of T_h and t_l	14 μ s / 46 μ s	5.6 μ s / 18 μ s	14 μ s / 46 μ s	3.6 μ s / 11.8 μ s
Transmission Rate	~16 kbits/sec.	~42 kbits/sec.	~16 kbits/sec.	~ 65 kbits/sec.
Related Command	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.)

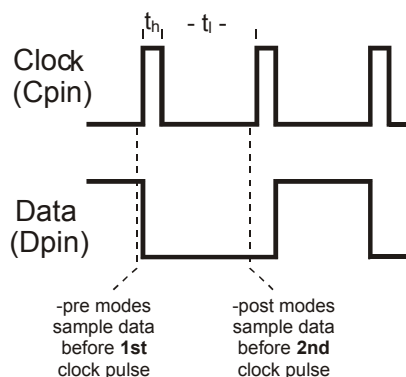


Figure 5.42: SHIFTIN Timing Diagram. Refer to the SHIFTIN Quick Facts table for timing information on t_h (t high) and t_l (t low).

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 μ s / 46 μ s	5.6 μ s / 18 μ s	14 μ s / 46 μ s	3.6 μ s / 11.8 μ s
Timing of t_a and t_b	15 μ s / 30 μ s	6.3 μ s / 12.5 μ s	15 μ s / 30 μ s	4 μ s / 7.8 μ 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.)

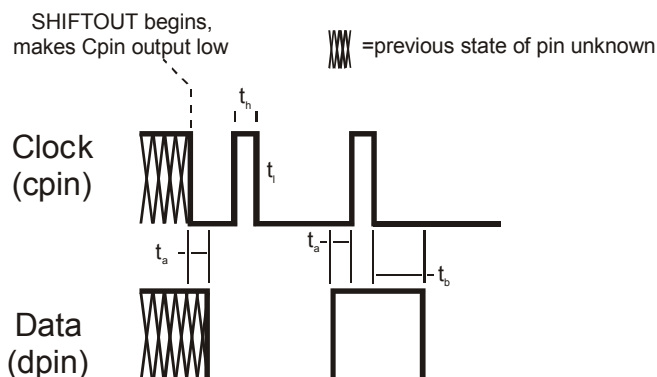


Figure 5.43: SHIFTOUT Timing Diagram. Refer to the SHIFTOUT Quick Facts table for timing information on t_n , t_i , t_a and t_b .

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

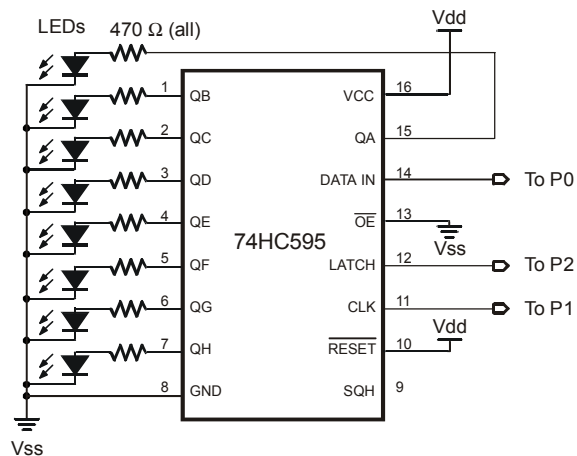


Figure 5.44: Circuit for Demo Program SHIFTOUT.bs2.

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 ± 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 μ 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.

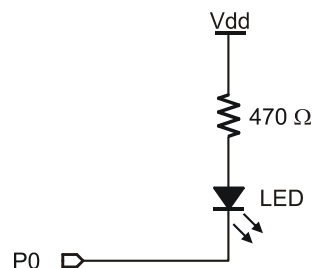


Figure 5.45: SLEEP Example LED Circuit.

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

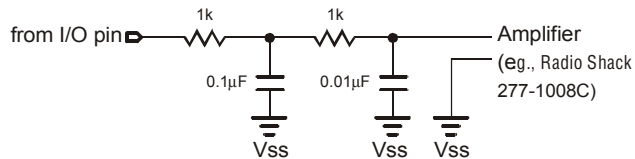
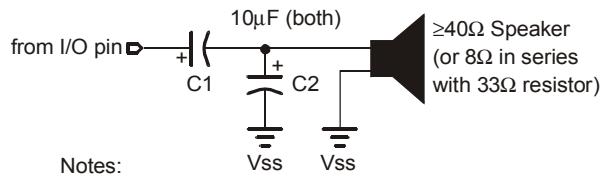


Figure 5.46: Example RC filter circuits for driving an audio amplifier (*top*) or a speaker (*bottom*).

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	<i>ProgramSlot</i>						

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
```

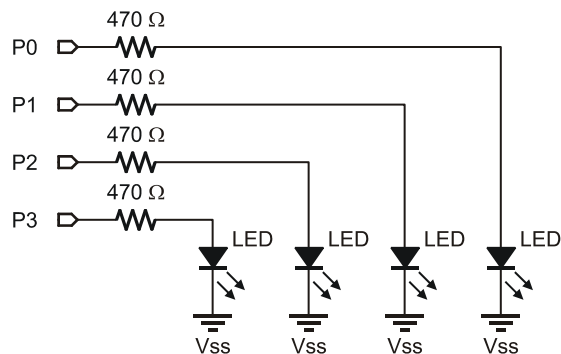


Figure 5.47: Example LED Circuit for TOGGLE Demo Programs.

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
Range of EEPROM Locations	0 to 255	0 to 2047	0 to 2047	0 to 2047 (see notes below)
Maximum Number of Writes per Location	10 million	10 million	100,000	100,000
Special Notes	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.
Related Commands	READ and EEPROM	READ and DATA	READ and DATA	READ, DATA, and STORE
PBASIC 2.5 Syntax Options	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 μ 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.

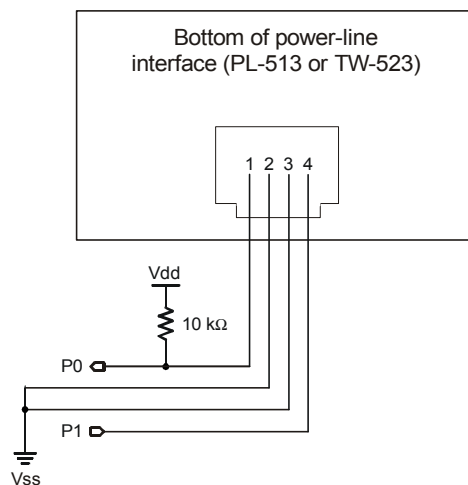


Figure 5.48: XOUT Power-Line Interface Circuit.

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 (^{2.5}).

Reserved Words

All BS2 Models			
#CASE ^{2.5}	CRSRX ^{2.5}	INC	PULSIN
#DEFINE ^{2.5}	CRSRXY ^{2.5}	IND	PULSOUT
#ELSE ^{2.5}	CRSRY ^{2.5}	INH	PWM
#ENDIF ^{2.5}	DATA	INL	RANDOM
#ENDSELECT ^{2.5}	DCD	INPUT	RCTIME
#ERROR ^{2.5}	DEBUG	INS	READ
#IF ^{2.5}	DEBUGIN ^{2.5}	ISBIN	REP
#SELECT ^{2.5}	DEC	ISBIN1...ISBIN16	RETURN
#THEN ^{2.5}	DEC1...DEC5	ISHEX	REV
\$PBASIC	DIG	ISHEX1...ISHEX4	REVERSE
\$PORT	DIM	LF ^{2.5}	SBIN
\$STAMP	DIR0...DIR15	LIGHTSON	SBIN1...SBIN16
ABS	DIRA	LOOKDOWN	SDEC
AND	DIRB	LOOKUP	SDEC1...SDEC5
ASC	DIRC	LOOP ^{2.5}	SELECT ^{2.5}
ATN	DIRD	LOW	SERIN
B0...B25	DIRH	LOWBIT	SEROUT
BELL	DIRL	LOWBYTE	SHEX
BIN	DIRS	LOWNIB	SHEX1...SHEX4
BIN1...BIN16	DO ^{2.5}	LSBFIRST	SHIFTIN
BIT	DTMFOUT	LSBPOST	SHIFTOUT
BIT0...BIT15	ELSE ^{2.5}	LSBPRES	SIN
BKSP	ELSEIF ^{2.5}	MAX	SKIP
BRANCH	END	MIN	SLEEP
BRIGHT	ENDIF ^{2.5}	MSBFIRST	SNUM
BS1	ENDSELECT ^{2.5}	MSBPOST	SQR
BS2	EXIT ^{2.5}	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 ^{2.5}	UNITON
CASE ^{2.5}	HIGHBYTE	OR	UNITSOFF
CLRDN ^{2.5}	HIGHNIB	OUT0...OUT15	UNTIL ^{2.5}
CLREOL ^{2.5}	HOME	OUTA	VAR
CLS	HYP	OUTB	W0...W12
CON	IBIN	OUTC	WAIT
COS	IBIN1...IBIN16	OUTD	WAITSTR
COUNT	IF	OUTH	WHILE ^{2.5}
CR	IHEX	OUTL	WORD
CRSRDN ^{2.5}	IHEX1...IHEX4	OUPUT	WRITE
CRSRLF ^{2.5}	IN0...IN15	OUTS	XOR
CRSRRT ^{2.5}	INA	PAUSE	XOUT
CRSRUP ^{2.5}	INB	PIN ^{2.5}	

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 (^{2.5}) 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

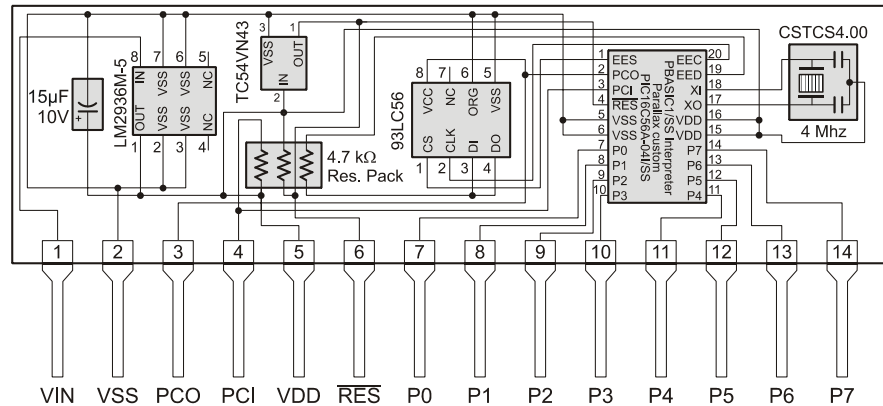
⊗ Means any non-binary-numeric characters such as letters, spaces, minus signs, carriage returns, control characters, etc. (Binary numerics are: 0 and 1).

* 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.

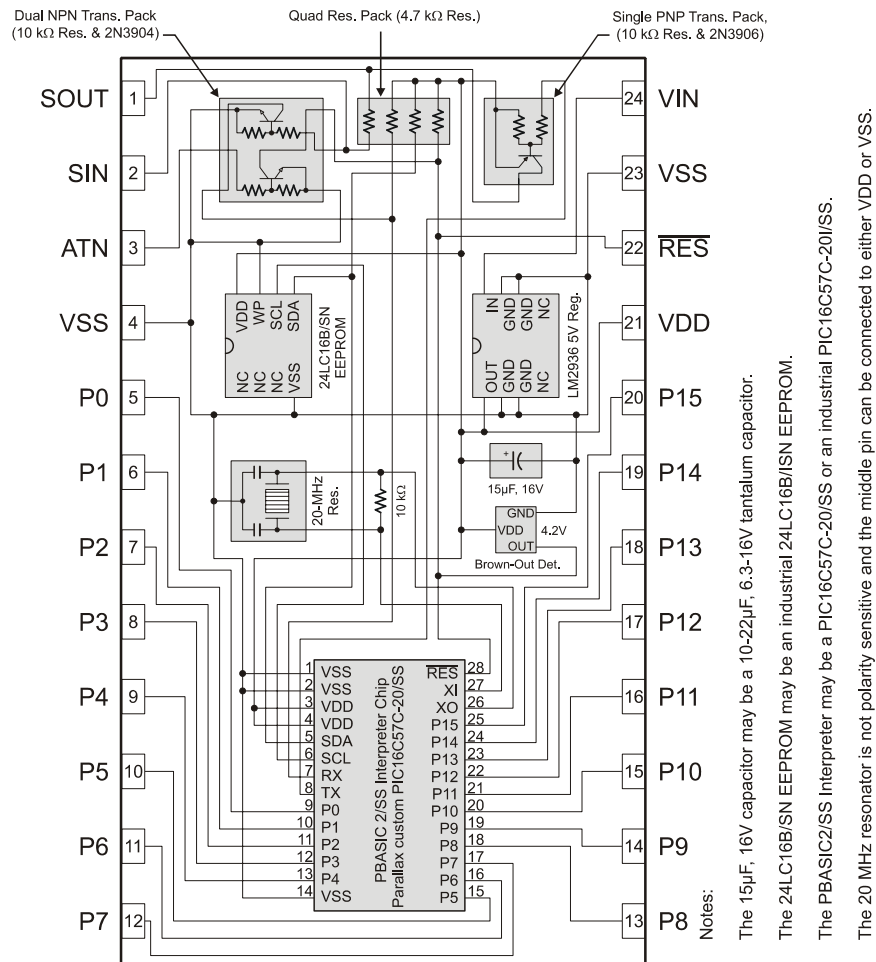
The 93LC56 EEPROM may be a 93LC56A, 93LC66 or 93LC66A.

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

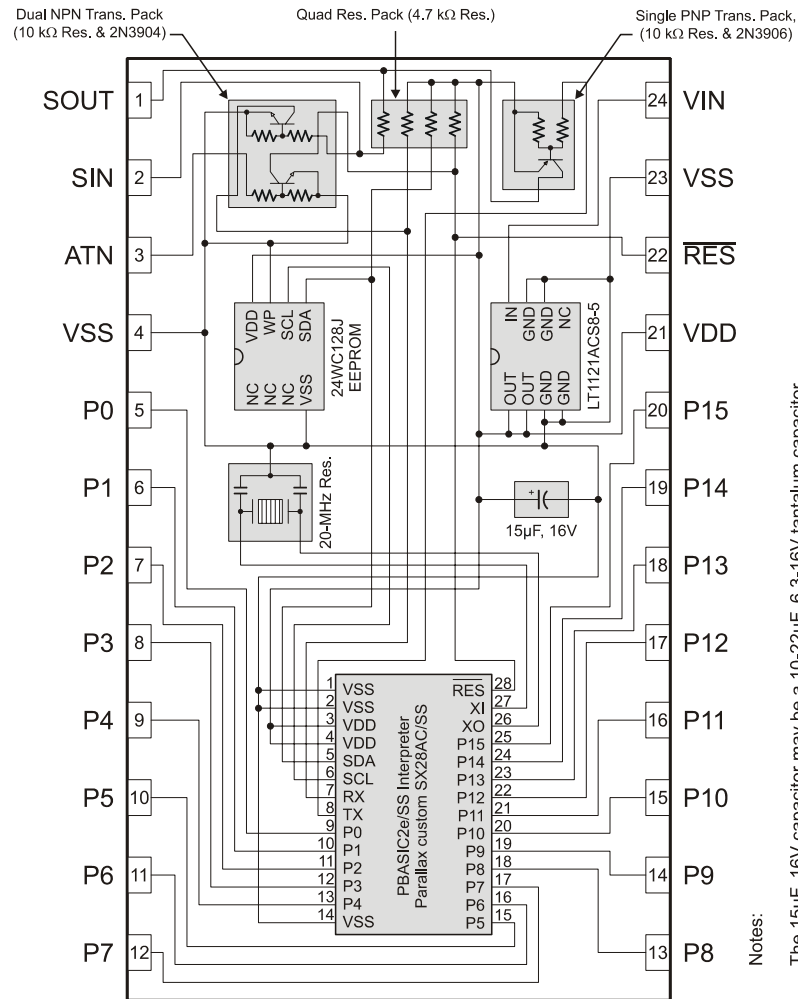
BASIC Stamp Schematics

BASIC Stamp 2 Schematic (Rev G)



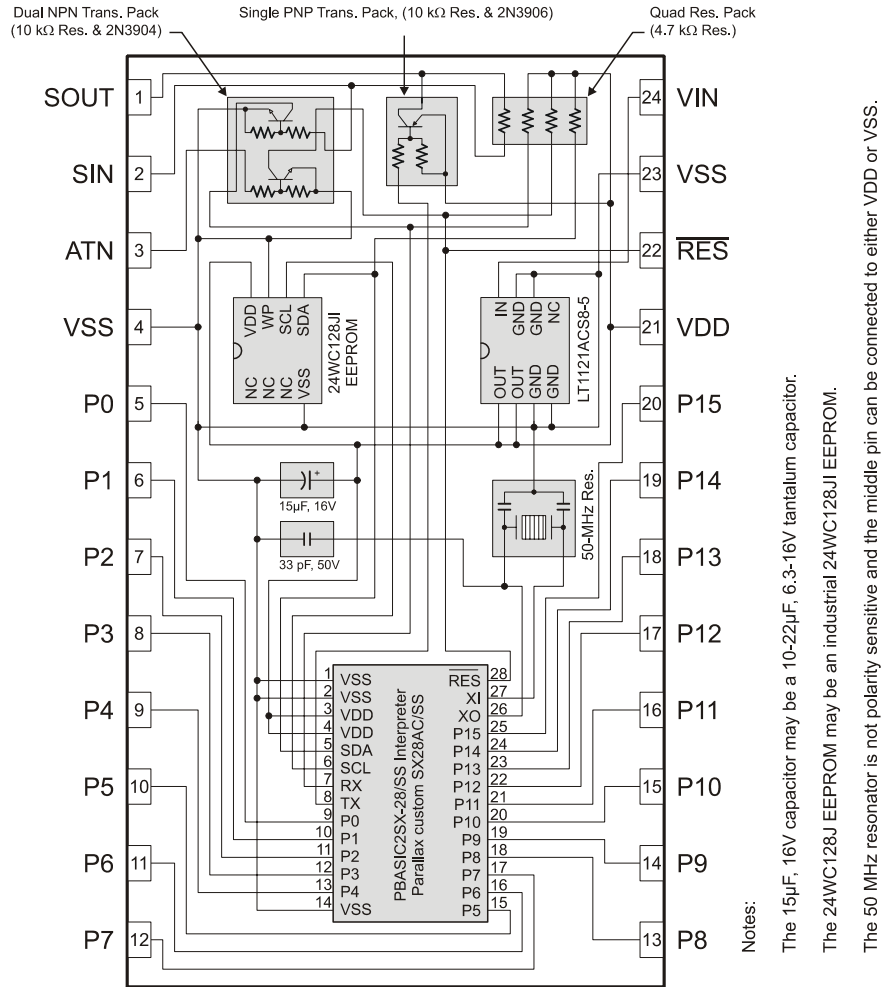
Appendix D: BASIC Stamp Schematics

BASIC Stamp 2e Schematic (Rev B)



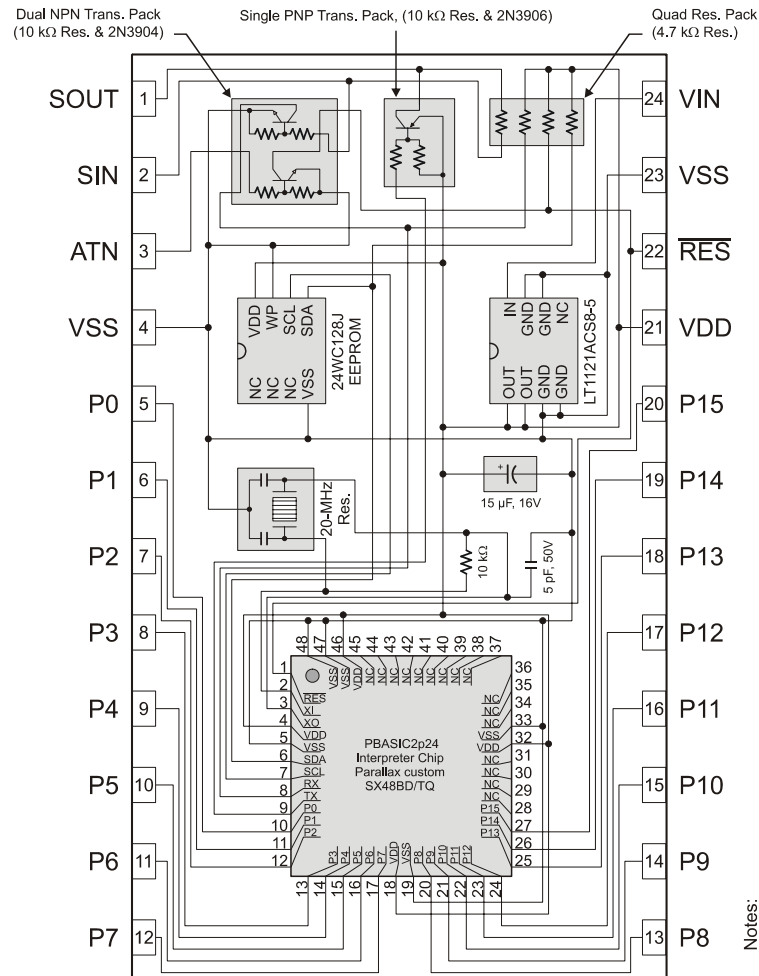
BASIC Stamp Schematics

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)

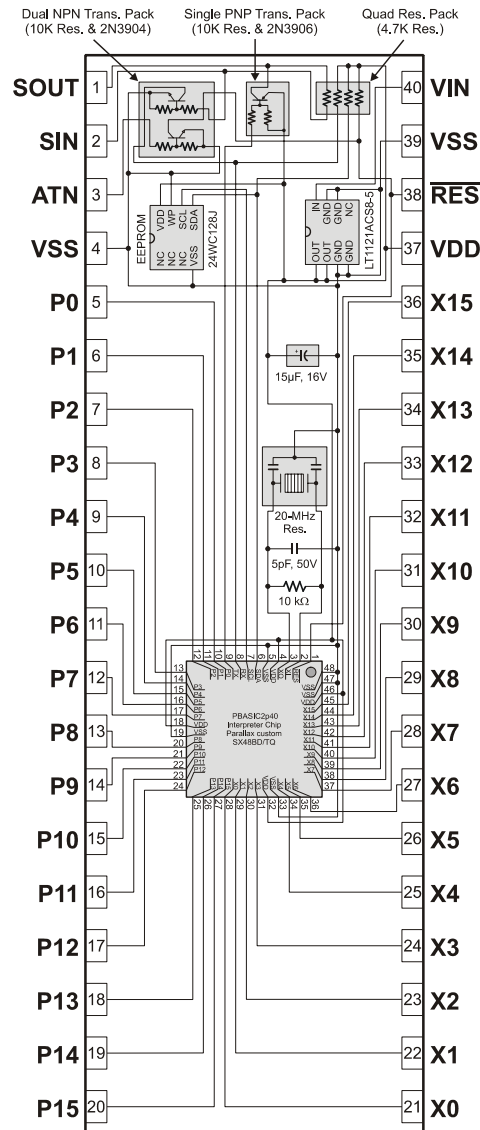
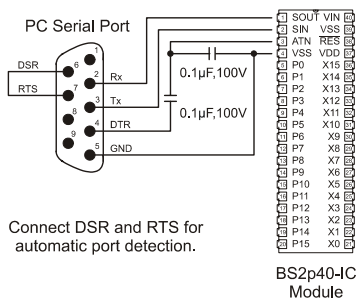
Notes:

The 15 μ F, 16V capacitor may be a 10-22 μ F, 6.3-16V tantalum capacitor.

The 24WC128J EEPROM may be an industrial 24WC128JI EEPROM.

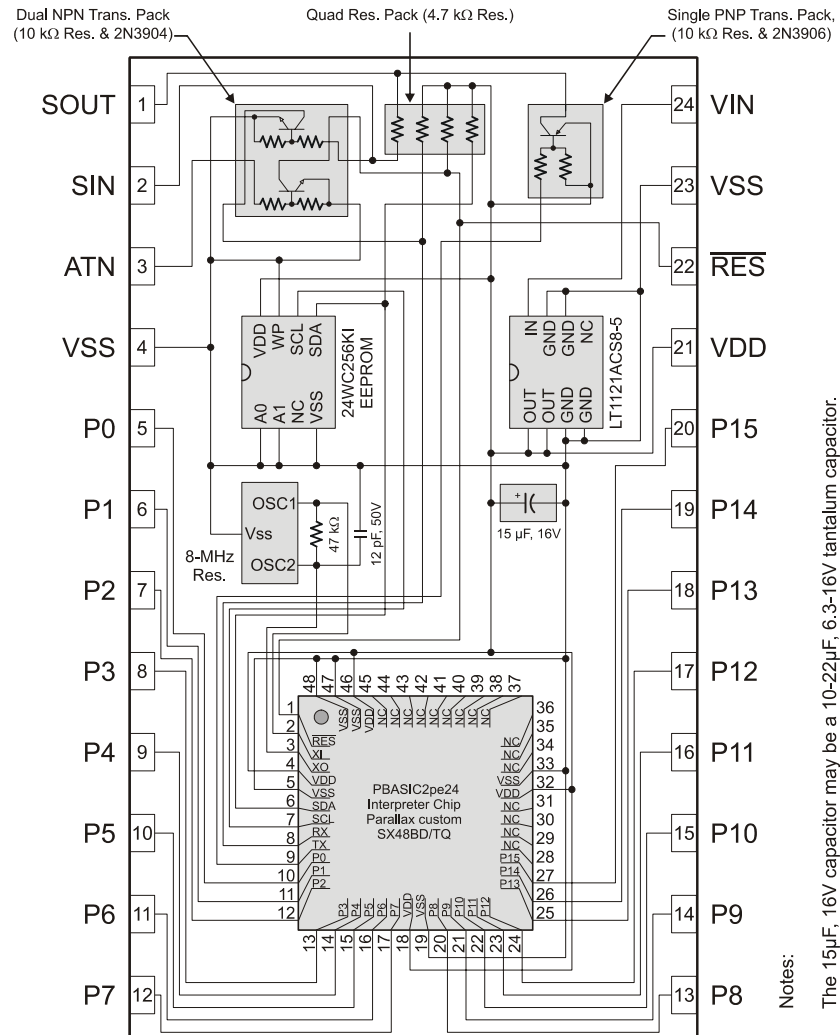
The 20 MHz resonator is not polarity sensitive and the middle pin can be connected to either VDD or VSS.

PC to BS2p40-IC Connection



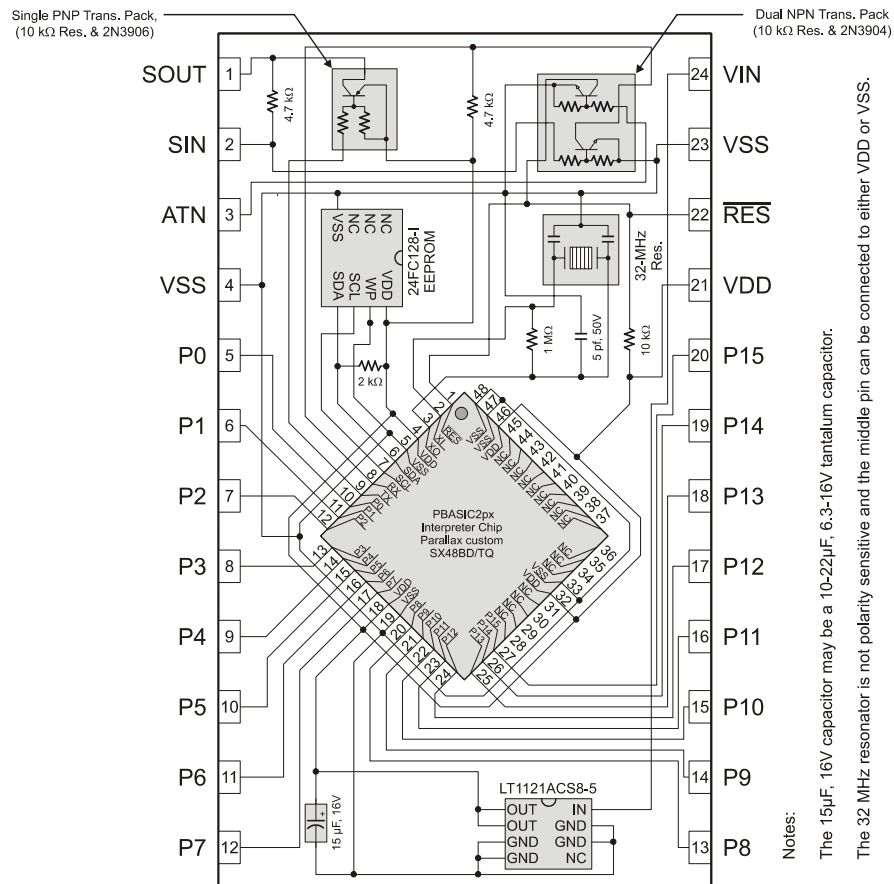
Appendix D: BASIC Stamp Schematics

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
I²C 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
 I²C, 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
- 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**

— T —

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

What's a Microcontroller?

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

ISBN 1-928982-02-6

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 – With over 4,000 subscribers, 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, this list has about 650 subscribers who 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 – Consisting of about 40 people, 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. Approximately 600 members.
- 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. Approximately 250 members.

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	v
Audience.....	v
Support and Discussion Groups.....	v
Teacher's Guide.....	vi
The Stamps in Class Curriculum.....	vi
Foreign Translations.....	viii
Special Contributors.....	viii
Chapter #1: Getting Started	1
How Many Microcontrollers Did You Use Today?.....	1
The BASIC Stamp 2 - Your New Microcontroller.....	1
Amazing Inventions with BASIC Stamp Microcontrollers.....	2
Hardware and Software.....	5
Activity #1: Getting the Software.....	5
Activity #2: Installing the Software.....	10
Activity #3: Setting Up the Hardware and Testing the System.....	13
Activity #4: First Program.....	20
Activity #5: Looking Up Answers.....	27
Activity #6: Introducing ASCII Code.....	30
Activity #7: When You're Done.....	31
Summary.....	33
Chapter #2: Lights On – Lights Off	37
Indicator Lights.....	37
Making a Light Emitting Diode (LED) Emit Light.....	37
Activity #1: Building and Testing the LED Circuit.....	38
Activity #2: On/Off Control with the BASIC Stamp.....	46
Activity #3: Counting and Repeating.....	52
Activity #4: Building and Testing a Second LED Circuit.....	56
Activity #5: Using Current Direction to Control a Bi-Color LED.....	60
Summary.....	67
Chapter #3: Digital Input - Pushbuttons	71
Found on Calculators, Hand Held Games, and Appliances.....	71
Receiving vs. Sending High and Low Signals.....	71
Activity #1: Testing a Pushbutton with an LED Circuit.....	71
Activity #2: Reading a Pushbutton with the BASIC Stamp.....	75
Activity #3: Pushbutton Control of an LED Circuit.....	80
Activity #4: Two Pushbuttons Controlling Two LED Circuits.....	83
Activity #5: Reaction Timer Test.....	88
Summary.....	97

Chapter #4: Controlling Motion	103
Microcontrolled Motion	103
On/Off Signals and Motor Motion	103
Activity #1: Connecting and Testing the Servo	103
Activity #2: Controlling Position with Your Computer	119
Activity #3: Converting Position to Motion	125
Activity #4: Pushbutton Controlled Servo	128
Summary	134
Chapter #5: Measuring Rotation.....	139
Adjusting Dials and Monitoring Machines.....	139
The Variable Resistor under the Dial – A Potentiometer	139
Activity #1: Building and Testing the Potentiometer Circuit.....	141
Activity #2: Measuring Resistance by Measuring Time	143
Activity #3: Reading the Dial with the BASIC Stamp	149
Activity #4: Controlling a Servo with a Potentiometer.....	152
Summary	160
Chapter #6: Digital Display.....	165
The Every-Day Digital Display	165
What's a 7-Segment Display?	165
Activity #1: Building and Testing the 7-Segment LED Display	167
Activity #2: Controlling the 7-Segment LED Display.....	171
Activity #3: Displaying Digits.....	174
Activity #4: Displaying the Position of a Dial.....	181
Summary	186
Chapter #7: Measuring Light.....	189
Devices that Contain Light Sensors	189
Introducing the Photoresistor.....	189
Activity #1: Building and Testing the Light Meter.....	190
Activity #2: Graphing Light Measurements.....	193
Activity #3: Tracking Light Events.....	197
Activity #4: Simple Light Meter	203
Summary	215
Chapter #8: Frequency and Sound	219
Your Day and Electronic Beeps.....	219
Microcontrollers, Speakers, Beeps and On/Off Signals	219
Activity #1: Building and Testing the Speaker	220
Activity #2: Action Sounds.....	222
Activity #3: Musical Notes and Simple Songs	227
Activity #4: Microcontroller Music	233
Activity #5: Cell Phone RingTones	245

Summary	258
Chapter #9: Electronic Building Blocks	263
Those Little Black Chips	263
Expand Your Projects with Peripheral Integrated Circuits	264
Activity #1: Control Current Flow with a Transistor	265
Activity #2: Introducing the Digital Potentiometer	267
Summary	278
Chapter #10: Running the Whole Show	281
Subsystem Integration	281
Activity #1: Building and Testing Each Pushbutton Circuit	282
Activity #2: Building and Testing Each RC-Time Circuit	285
Activity #3: Subsystem Integration Example	287
Activity #4: Developing and Adding a Software Subsystem	291
Summary	297
Appendix A: USB to Serial Adapter	301
Appendix B: Equipment and Parts Lists	303
Appendix C: BASIC Stamp and Carrier Board Components and Functions....	307
Appendix D: Batteries and Power Supplies.....	311
Appendix E: Trouble-Shooting.....	315
Appendix F: More about Electricity	319
Appendix G: RTTTL Format Summary	327
Index	329

Preface

This text answers the question “What’s a microcontroller?” by showing students how they can design their own customized, intelligent inventions with Parallax, Inc.’s BASIC Stamp® microcontroller module. The activities in this text incorporate a variety of fun and interesting experiments designed to appeal to a student’s imagination by using motion, light, sound, and tactile feedback to introduce new concepts. These activities introduce students to a variety of basic principles in the fields of computer programming, electricity and electronics, mathematics and physics. Many of the activities facilitate hands-on presentation of design practices used by engineers and technicians in the creation of modern machines and appliances, using common inexpensive parts.

AUDIENCE

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

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.

TEACHER'S GUIDE

Each chapter summary contains a set of questions, exercises and projects with solutions provided. A Teacher's Guide is also available for this text. It contains an additional set of solved questions, exercises and projects, as well as some expanded and alternative solutions to the material in the text. The Teacher's Guide is available free in both Word and PDF file formats by joining the Parallax Educators Yahoo Group or by emailing stampsinclass@parallax.com. To obtain these files, you must provide proof of your status as an educator.

THE STAMPS IN CLASS CURRICULUM

What's a Microcontroller? is the gateway text to our Stamps in Class curriculum. After completing this text, you can continue your studies with any of the Student Guides listed below. All of the books listed are available for free download from www.parallax.com. The versions cited below were current at the time of this printing. Please check our web sites www.parallax.com or www.stampsinclass.com for the latest revisions; we continually strive to improve our educational program.

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**
***“Industrial Control”*, Student Guide, Version 1.1, Parallax Inc., 1999**
***“Robotics with the Boe-Bot”*, Student Guide, Version 2.0, Parallax Inc., 2003**

More Robotics Kits:

Some enter the Stamps in Class curriculum through the *Robotics with the Boe-Bot* Student Guide. After completing it, 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., 2001**
***“Understanding Signals”*, Student Guide, Version 1.0, Parallax Inc., 2003**

Reference

This book is an essential reference for all Stamps in Class Student Guides. It is packed with information on the BASIC Stamp series of microcontroller modules, our BASIC Stamp Editor, and our 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 after the Title page.

SPECIAL CONTRIBUTORS

The Parallax team assembled to write this text includes: curriculum design and technical writing by Andy Lindsay, illustration by Rich Allred, cover design by Jen Jacobs and Larissa Crittenden, general consulting by Aristides Alvarez and Jeff Martin, electromechanical consulting by John Barrowman, technical review and solutions by Kris Magri, technical editing by Stephanie Lindsay, and committee review by Rich Allred, Gabe Duran, Stephanie Lindsay, and Kris Magri.

What's a Microcontroller? Student Guide Version 2.2 was written by Andy Lindsay after collecting observations and educator feedback while traveling the nation teaching Parallax Educators Courses. Andy studied Electrical and Electronic Engineering at California State University, Sacramento, and this is his third Stamps in Class Student Guide. He is also a contributing author of several papers that address the topic of microcontrollers in pre-engineering curricula. When he's not writing educational material, Andy does product engineering for Parallax.

Parallax wishes to thank StampsInClass Yahoo Group member Robert Ang for his thorough draft review and detailed input, and veteran engineer and esteemed customer Sid Weaver for his insightful review. Thanks also to Stamps in Class authors Tracy Allen (*Applied Sensors*), and Martin Hebel (*Industrial Control*) for their review and recommendations. Andy Lindsay wishes to thank his father Marshall and brother-in-law Kubilay for their expert musical advice and suggestions. Stamps in Class was founded by Ken Gracey, and Ken wishes to thank the Parallax staff for the great job they do. Each and every Parallaxian has made contributions to this and every Stamps in Class text.

Chapter #1: Getting Started

HOW MANY MICROCONTROLLERS DID YOU USE TODAY?

A microcontroller is a kind of miniature computer that you can find in all kinds of gizmos. Some examples of common, every-day products that have microcontrollers built-in are shown in Figure 1-1. If it has buttons and a digital display, chances are it also has a programmable microcontroller brain.



Figure 1-1
Every-Day Examples of
Devices that Contain
Microcontrollers

Try making a list and counting how many devices with microcontrollers you use in a typical day. Here are some examples: if your clock radio goes off, and you hit the snooze button a few times in the morning, the first thing you do in your day is interact with a microcontroller. Heating up some food in the microwave oven and making a call on a cell phone also involve operating microcontrollers. That's just the beginning. Here are a few more examples: turning on the television with a handheld remote, playing a handheld game, using a calculator, and checking your digital wristwatch. All those devices have microcontrollers inside them that interact with you.

THE BASIC STAMP 2 - YOUR NEW MICROCONTROLLER

Parallax, Inc.'s BASIC Stamp® 2 module shown in Figure 1-2 has a microcontroller built onto it. It's the black chip with lettering on it that reads "PIC16C57". The rest of the components on the BASIC Stamp module are also found in consumer appliances you use every day. All together, they are correctly called an embedded computer system. This name is almost always shortened to just "embedded system". Frequently, such modules are commonly just called "microcontrollers."

The activities in this text will guide you through building circuits similar to the ones found in consumer appliances and high-tech gizmos. You will also write computer programs that the BASIC Stamp module will run. These programs will make the BASIC Stamp module monitor and control these circuits so that they perform useful functions.

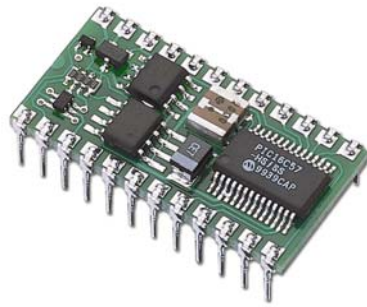


Figure 1-2
The BASIC Stamp® 2
Microcontroller Module

*BASIC Stamp 2
modules are the most
popular
microcontrollers made
by Parallax, Inc.*

In this text, “BASIC Stamp” refers to Parallax Inc.’s BASIC Stamp® 2 microcontroller module. There are other BASIC Stamp modules, some of which are shown in Figure 1-3. Each BASIC Stamp module is color coded. The BASIC Stamp 2 is green. The BASIC Stamp 2e is red. The BASIC Stamp 2SX is blue, and the BASIC Stamp 2p is gold. Each variation on the BASIC Stamp 2 is slightly different, featuring higher speed, more memory, additional functionality, or some combination of these extra features.

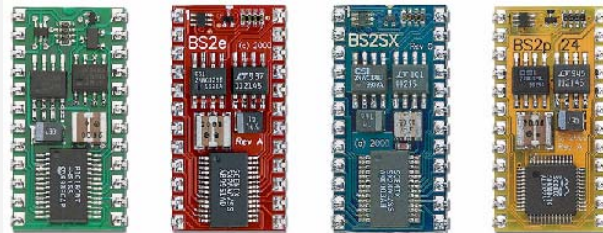


Figure 1-3
BASIC Stamp®
Modules

From Left to Right:
BASIC Stamp® 2, 2e,
2SX, and 2p

AMAZING INVENTIONS WITH BASIC STAMP MICROCONTROLLERS

Consumer appliances aren’t the only things that contain microcontrollers. Robots, machinery, aerospace designs and other high-tech devices are also built with microcontrollers. Let’s take a look at some examples that were created with BASIC Stamp modules.

Robots have been designed to do everything from helping students learn more about microcontrollers, to mowing the lawn, to solving complex mechanical problems. Figure 1-4 shows two example robots. On each of these robots, students use the BASIC Stamp 2 to read sensors, control motors, and communicate with other computers. The robot on

the left is Parallax Inc.'s Boe-Bot™. The projects in the *Robotics with the Boe-Bot* text can be tackled using the Boe-Bot after you've worked through the activities in this one. The robot on the right was built by a group of students and entered into a First Robotics competition. The goal of the contest is different each year. In the example shown, the goal was to see which group's robot could sort colored hoops the fastest.



Figure 1-4
Educational Robots

Parallax Boe-Bot™ (left)
First Competition Robot (right)

Other robots solve complex problems, such as the autonomous remote flight robot shown at the left of Figure 1-5. This robot was built and tested by mechanical engineering students at the University of California, Irvine. They used a BASIC Stamp module to help it communicate with a satellite global positioning system (GPS) so that the robot could know its position and altitude. The BASIC Stamp also read level sensors and controlled the motor settings to keep the robot flying properly. The mechanical millipede robot on the right was developed by a professor at Nanyang Technical University, Singapore. It has more than 50 BASIC Stamp modules, and they all communicate with each other in an elaborate network that helped control and orchestrate the motion of each set of legs. Robots like this not only help us better understand designs in nature, but they may eventually be used to explore remote locations, or even other planets.



Figure 1-5
Examples of Research
Robots that Contain
Microcontrollers

*Autonomous flying
robot at UC Irvine (left)*
*and Millipede Project at
Nanyang University
(right)*

With the help of microcontrollers, robots will also take on day-to-day tasks, such as mowing the lawn. The BASIC Stamp module inside the robotic lawn mower shown in Figure 1-6 helps it stay inside the boundaries of the lawn, and it also reads sensors that detect obstacles and controls the motors that make it move.



Figure 1-6
Robotic Lawn Mower

Prototype by the Robot Shop

Microcontrollers are also used in scientific, high technology, and aerospace projects. The weather station shown on the left of Figure 1-7 is used to collect environmental data related to coral reef decay. The BASIC Stamp module inside it gathers this data from a variety of sensors and stores it for later retrieval by scientists. The submarine in the center is an undersea exploration vehicle, and its thrusters, cameras and lights are all controlled by BASIC Stamp microcontrollers. The rocket shown on the right is one that was part of a competition to launch a privately owned rocket into space. Nobody won the competition, but this rocket almost made it! The BASIC Stamp controlled just about every aspect of the launch sequence.



Figure 1-7
High-tech and Aerospace
Microcontroller Examples

*Ecological data collection
by EME Systems (left),
undersea research by
Harbor Branch Institute
(center), and JP Aerospace
test launch (right)*

From common household appliances all the way through scientific and aerospace applications, the microcontroller basics you will need to get started on projects like these are introduced here. By working through the activities in this book, you will get to

experiment with and learn how to use a variety of building blocks found in all these high-tech inventions. You will build circuits for displays, sensors, and motion controllers. You will learn how to connect these circuits to the BASIC Stamp 2 module, and then write computer programs that make it control displays, collect data from the sensors, and control motion. Along the way, you will learn many important electronic and computer programming concepts and techniques. By the time you're done, you might find yourself well on the way to inventing a gizmo of your own design.

HARDWARE AND SOFTWARE

Getting started with BASIC Stamp microcontroller modules is similar to getting started with a brand-new PC or laptop. The first things that most people have to do 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 a BASIC Stamp module, you will be doing all these same activities. If you are in a class, your hardware may already be all set up for you. If this is the case, your teacher may have other instructions. If not, this chapter will take you through all the steps of getting your new BASIC Stamp microcontroller up and running.

ACTIVITY #1: GETTING THE SOFTWARE

The BASIC Stamp Editor (version 2.0 or higher) is the software you will use in most of the activities and projects in this text. You will use this software to write programs that the BASIC Stamp module will run. You can also use this software to display messages sent by the BASIC Stamp that help you understand what it senses.



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 May 2003 or newer.

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 the installation procedure.

Computer System Requirements

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

- Microsoft Windows 95® 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 Appendix A: USB to Serial Adapter for details and installation instructions.

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 1-8 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 1-8).
- ✓ Point at the *Downloads* menu to display the options.
- ✓ Point at the *BASIC Stamp Software* link and click to select it.



Figure 1-8
The Parallax Web Site:

www.parallax.com

- ✓ When you get to the BASIC Stamp Software page, find the most recent version of the BASIC Stamp Windows Editor download, with a version number of 2.0 or higher.
- ✓ Click the *Download* icon. In Figure 1-9, the download icon looks like a file folder to the right of the description: “BASIC Stamp Windows Editor version 2.0 Beta 1 (6MB)”.

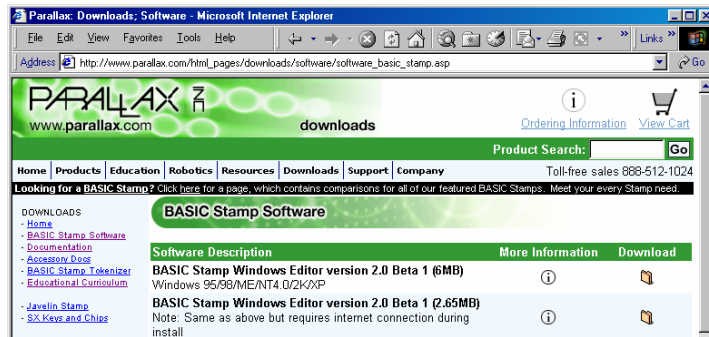


Figure 1-9
The Parallax Web Site
Downloads Page

- ✓ When the File Download window shown in Figure 1-10 appears, select: *Save this program to disk*.
- ✓ Click the *OK* button.

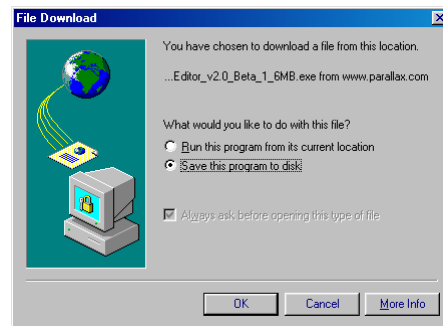


Figure 1-10
File Download Window

Figure 1-11 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.

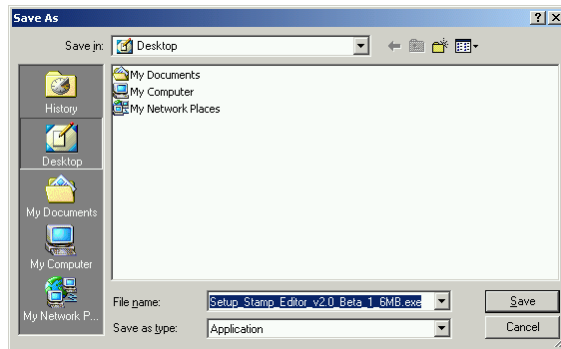


Figure 1-11
Save As Window

*Selecting a place to
save the file*

- ✓ Wait while the BASIC Stamp Editor installation program downloads (shown in Figure 1-12). This may take a while if you are using a modem connection.
- ✓ When the download is complete, leave the window shown in Figure 1-13 open while you skip to the next section - Activity #2: Installing the Software.

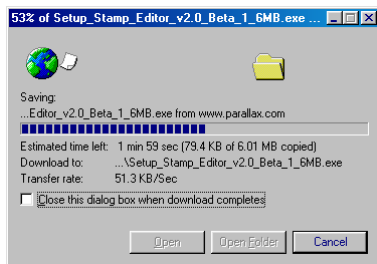


Figure 1-12: Download Progress Window

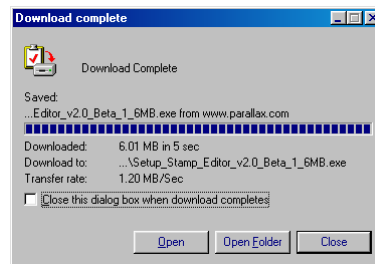


Figure 1-13: Download Complete

Finding the Software on the Parallax CD

You can also install the BASIC Stamp Editor from the Parallax CD, but it has to be May 2003 or newer 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 Welcome application shown in Figure 1-14 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, then double-click *Welcome*.

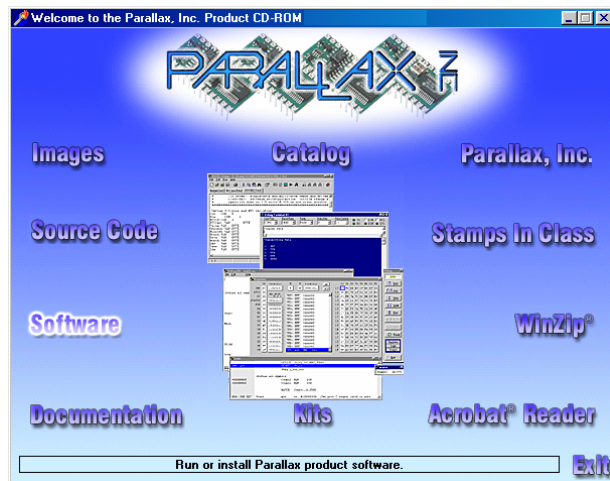


Figure 1-14
The Parallax CD
Browser

- ✓ Click the *Software* link shown in Figure 1-14.
- ✓ Click the + next to the BASIC Stamps folder shown in Figure 1-15.
- ✓ Click the + next to the Windows folder.
- ✓ Click the floppy diskette icon labeled “Stamp 2/2e/2sx/2p/2pe (stampw.exe)”.
- ✓ Move on to Activity #2: Installing the Software.

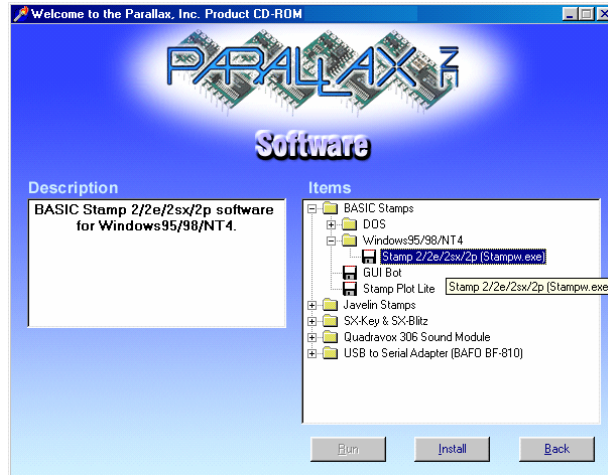


Figure 1-15
The Parallax CD
Browser

*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 few months 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 let's 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 1-16.

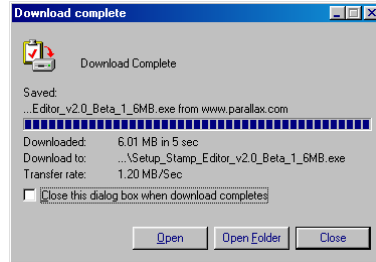


Figure 1-16
Download Complete Window

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

- ✓ If you located the software on the Parallax CD, click the *Install* button shown in Figure 1-17.



Figure 1-17
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 1-18.

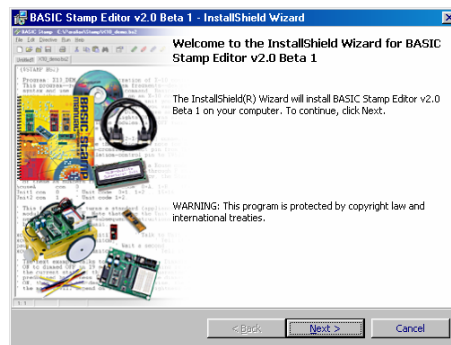


Figure 1-18
InstallShield Wizard for the BASIC Stamp Editor

Click Next.

- ✓ Select *Typical* for your setup type as shown in Figure 1-19.

- ✓ Click the *Next* button.

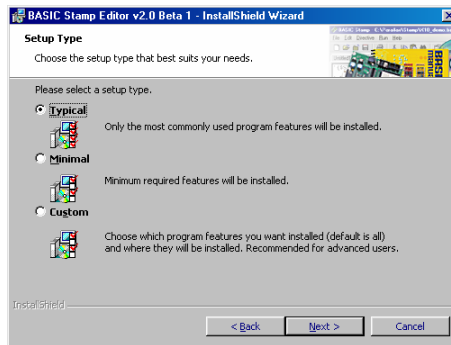


Figure 1-19
Setup Type

Click Typical, then click the Next button.

- ✓ When the InstallShield Wizard tells you it is “Ready to Install the Program”, click the *Install* button shown in Figure 1-20.

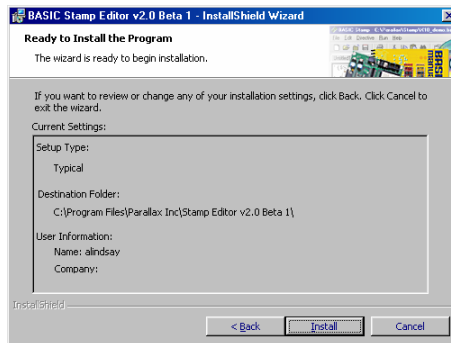


Figure 1-20
Ready to Install.

Click the Install button.

- ✓ When the InstallShield Wizard window tells you “InstallShield Wizard Completed”, as shown in Figure 1-21, click *Finish*.

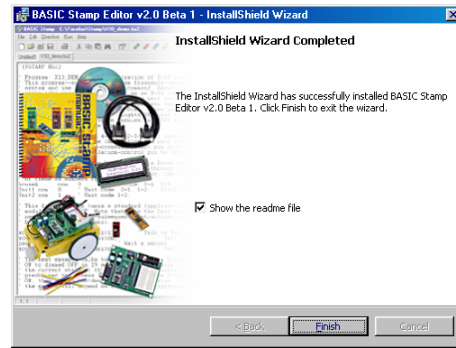


Figure 1-21
InstallShield Wizard
Completed:

Click the Finish button.

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

The BASIC Stamp module 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.

Introducing the BASIC Stamp®, Board of Education®, and HomeWork Board™

Parallax Inc.'s Board of Education® carrier board shown in Figure 1-22 next to a BASIC Stamp module. As mentioned earlier, the BASIC Stamp is a type of 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 your BASIC Stamp module.

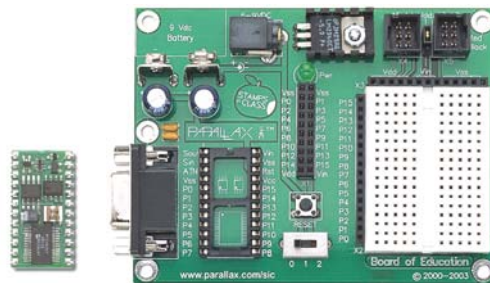


Figure 1-22
BASIC Stamp® 2
Microcontroller Module (left)
and Board of Education®
Carrier Board (right)

Parallax, Inc.'s BASIC Stamp HomeWork Board™ is shown in Figure 1-23. This board is like a Board of Education with the BASIC Stamp 2 module built-in. Its surface-mounted components are visible to the left of the white breadboard area. You can use either the Board of Education with a BASIC Stamp module or the BASIC Stamp HomeWork Board as your project platform for the activities in this text.

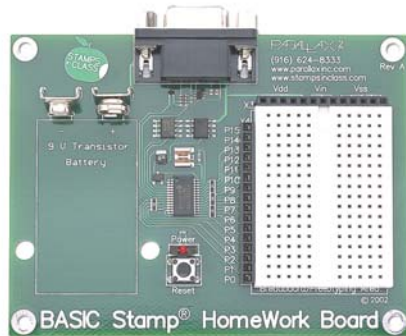


Figure 1-23
BASIC
Stamp®
HomeWork
Board™
Project
Platform.



Learn more about the features, parts and functions of BASIC Stamp modules, Board of Education carrier boards, and the HomeWork Board project platform. See Appendix C: BASIC Stamp and Carrier Board Components and Functions on page 307.

Required Hardware

- (1) BASIC Stamp 2 module AND
- (1) Board of Education
- or -
- (1) BASIC Stamp HomeWork Board

- (1) 9 V battery
- (1) strip of 4 adhesive rubber feet
- (1) Serial cable



Start with a new or fully charged 9 V battery. Avoid all the confusion a dead battery can cause. Start with a new alkaline battery or a rechargeable battery that has recently been fully recharged.

**CAUTION! Before using an AC adapter, “battery replacer”, or DC supply:**

- ✓ Consult Appendix D: Batteries and Power Supplies on page 311 to make sure the supply you use is appropriate for the activities in this text.

Connecting the Hardware

Both the Board of Education and the BASIC Stamp HomeWork Board come with a strip that has four adhesive rubber feet. These rubber feet are shown in Figure 1-24, and they should be affixed to the underside of your Board of Education or BASIC Stamp HomeWork Board.



Figure 1-24
Rubber Feet

- ✓ Remove each rubber foot from the adhesive strip and affix it to the underside of your board as shown in Figure 1-25. If you are using the Board of Education, it has circles on its underside that show where each rubber foot should be attached. For the HomeWork Board, just place a rubber foot next to each plated hole at each corner.

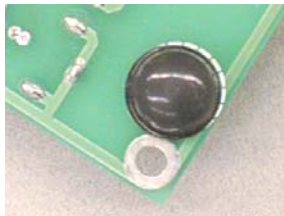


Figure 1-25
Rubber Foot
Affixed to
Underside of Board
of Education (left)
and HomeWork
Board (right)



Next, the Board of Education or BASIC Stamp HomeWork Board should be connected to your PC or laptop by a serial cable.

- ✓ Connect your serial cable to an available COM port on the back of your computer as shown in Figure 1-26.



USB Port Adapter If you are using a USB to Serial Adapter:

- ✓ Connect the USB end to your PC's USB port.
- ✓ Connect the COM port adapter either directly to your Board of Education or HomeWork Board, or connect it to the serial cable as shown in Figure 1-26.

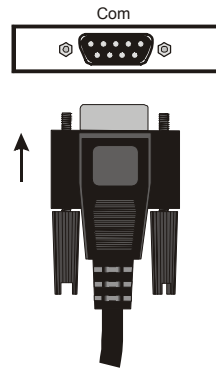


Figure 1-26: PC or Laptop COM Port

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



Figure 1-27: 3-position Switch

Set to the 0 position to turn off the power.

If you are using the BASIC Stamp 2 module and Board of Education:

- ✓ Set the 3-position switch on the Board of Education to position-0 as shown in Figure 1-27.



Only the Board of Education Rev C has a 3-position switch. To turn off power on a Board of Education Rev B, simply disconnect the power source by either unplugging the DC supply or the battery. These are shown in Figure 1-28, step 3 or 4.

- ✓ If your BASIC Stamp module is not already plugged into your Board of Education, insert it into the socket, oriented as shown in Figure 1-28, step-1. Make sure the pins are lined up properly with the holes in the socket and not folded under, then press down firmly.
- ✓ Plug the serial cable into the Board of Education as shown in step-2.

- ✓ Plug a DC power supply into the 6-9 VDC jack as shown in step-3, or plug a 9-V battery into the 9 VDC battery jack as shown in step-4.
- ✓ Move the 3-position switch from position-0 to position-1. The green light labeled Pwr on the Board of Education should now be on.

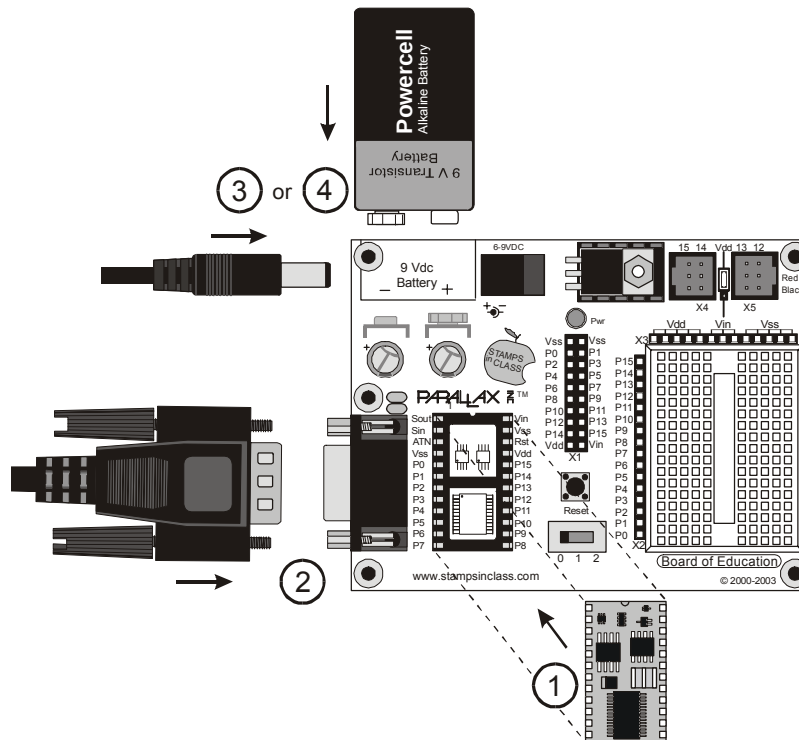


Figure 1-28
Board of
Education,
BASIC Stamp
Module,
Battery and
Serial Cable.

Connect components in the order shown in the diagram. Make sure to properly orient your BASIC Stamp module right side up, matching the notch on its top edge to notch on the socket.

If you are using the BASIC Stamp HomeWork Board:

- ✓ Connect the serial cable to the HomeWork Board (Figure 1-29, step-1).
- ✓ Connect a 9 V battery to the battery clip as shown in step-2.

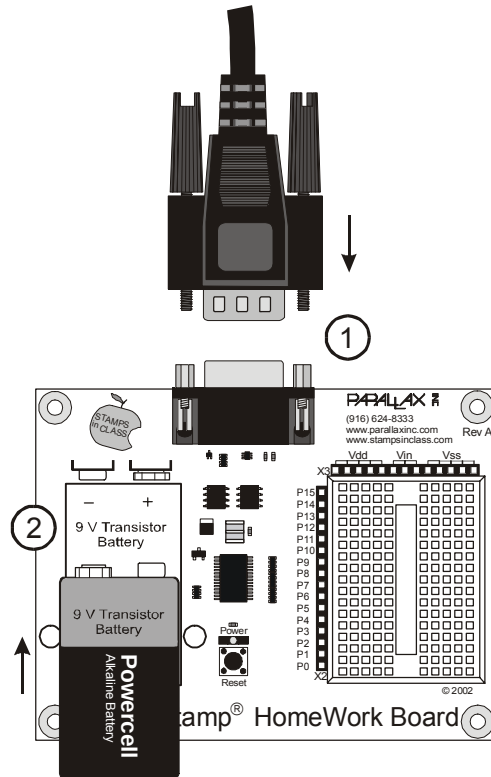


Figure 1-29
HomeWork Board and
Serial Cable

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

Testing for Communication

The BASIC Stamp Editor has a feature for testing to make sure your PC or laptop can communicate with your BASIC Stamp module.

- ✓ Double-click the BASIC Stamp Editor shortcut on your desktop. It should look similar to the one shown in Figure 1-30.



Figure 1-30
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.* → *BASIC Stamp Editor 2...*, then click the *BASIC Stamp Editor* icon.

Your BASIC Stamp Editor window should look similar to the one shown in Figure 1-31.



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.

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

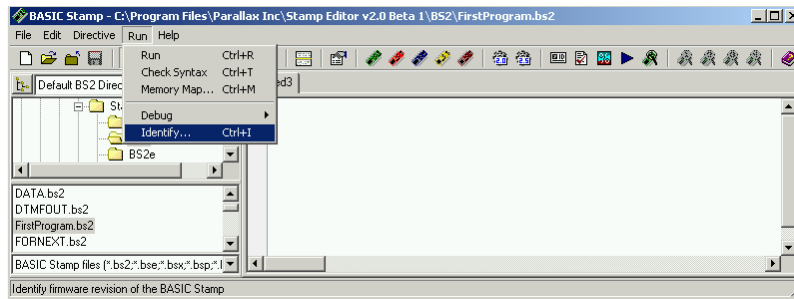


Figure 1-31
BASIC Stamp Editor

Select Identify from the Run menu.

An Identification window similar to the one shown in Figure 1-32 will appear. The example in the figure shows that a BASIC Stamp 2 has been detected on COM2.

- ✓ Check the Identification window to make sure a BASIC Stamp 2 module has been detected on one of the COM ports. If it has been detected, then you are ready for Activity #4: First Program.
- ✓ If the Identification window does not detect a BASIC Stamp 2 module on any of the COM ports, go to Appendix E: Trouble-Shooting on page 315.

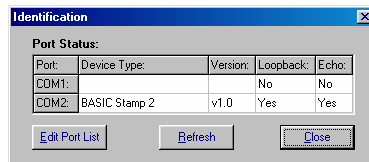


Figure 1-32
Identification Window

Example: BASIC Stamp 2 found on COM2.

ACTIVITY #4: 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 1-33 shows how it 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.

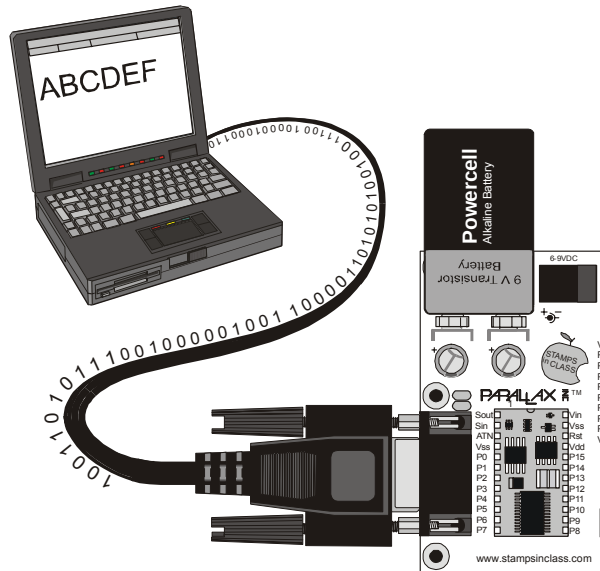


Figure 1-33
Messages from the
BASIC Stamp module
to your Computer

The BASIC Stamp module sends characters to your PC or laptop by transmitting a stream of binary ones and zeros. The BASIC Stamp Editor can detect and convert these binary codes to characters and display them.

First Program

The program listings that you will type into the BASIC Stamp Editor and download to the BASIC Stamp module will always be shown with a gray background like this:

Example Program: FirstProgram.bs2

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

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

DEBUG "Hello, it's me, your BASIC Stamp!"
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 1-34. 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 1-35. Its flyover help description is “PBASIC Language: 2.5”.

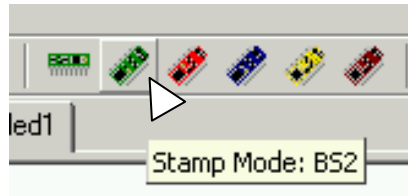


Figure 1-34
BS2 Icon

Clicking on this button will automatically place “ ‘{\$STAMP BS2}” at the beginning of your program.

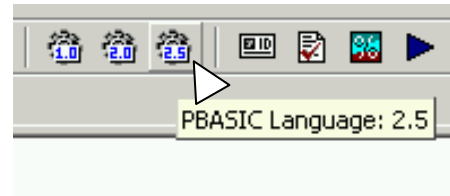


Figure 1-35
PBASIC 2.5 Icon

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 in the remaining 4 lines of the program exactly as shown in Figure 1-36.

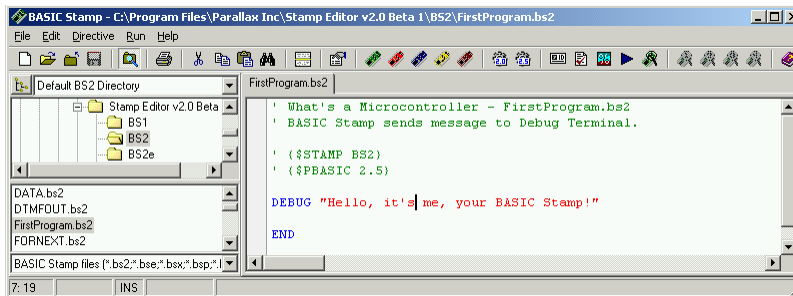


Figure 1-36
First Program
Entered into
Editor

*Enter your first
program into the
BASIC Stamp
Editor as shown
here.*

- ✓ Save your work by clicking *File* and selecting *Save* as shown in Figure 1-37.
- ✓ Enter the name *FirstProgram* into the *File name* field near the bottom of the *Save As* window as shown in Figure 1-38.
- ✓ Click the *Save* button.

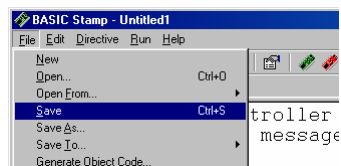


Figure 1-37: Saving the First Program

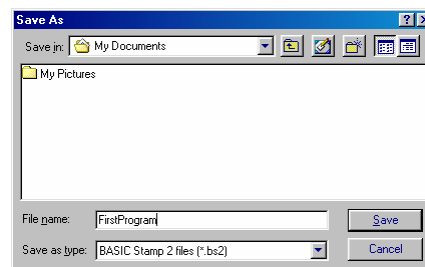


Figure 1-38: Entering the File Name



The next time you save, the BASIC Stamp Editor will automatically save to the same filename (FirstProgram.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 1-39.

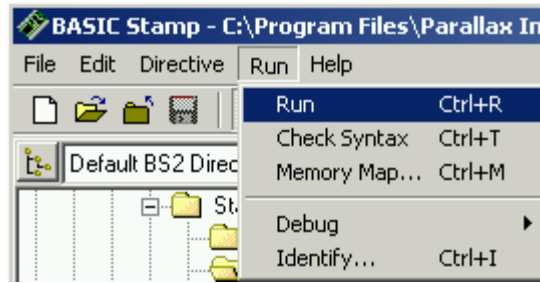


Figure 1-39
Running Your First
Program

A Download Progress window will appear briefly as the program is transmitted from the PC or laptop to your BASIC Stamp module. Figure 1-40 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. 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?

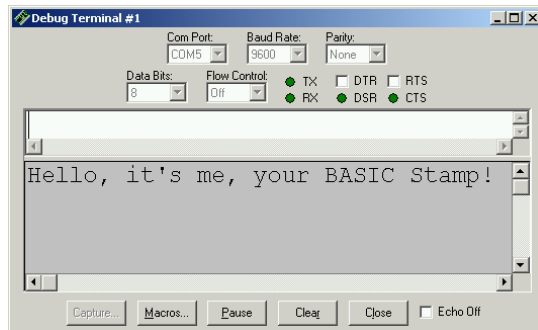
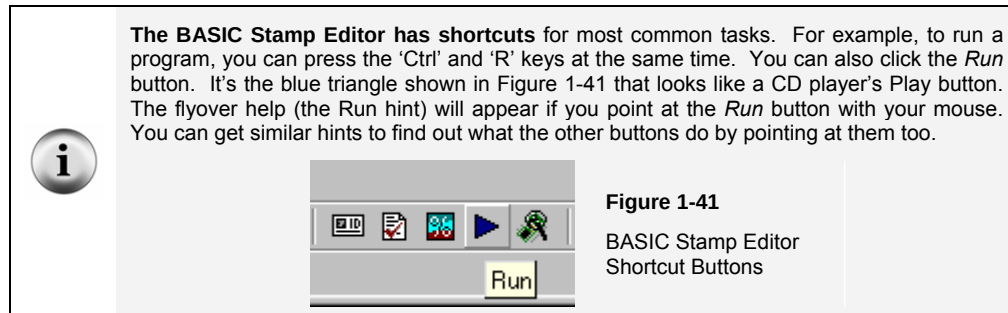


Figure 1-40
Debug Terminal

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



How FirstProgram.bs2 Works

The first two lines in the example 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 module. 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 the program's filename. The second comment contains a handy, one-line description that explains what the program does.

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

Although comments are ignored most of the time, the BASIC Stamp Editor does search through comments for special directives. 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 specifically to a BASIC Stamp 2 module. 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. Note that these compiler directives are enclosed in braces { } not parentheses (). You should always use the toolbar icons to place these compiler directives in your program to avoid typing errors. Also, entering the compiler directives by hand may not activate the syntax highlighting in the BASIC Stamp Editor. That function is what causes various letters, characters and words in your program to appear in different colors and capitalizations. Syntax highlighting makes your programs easier to read, understand, and correct if there are any bugs in them.

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

```
DEBUG "Hello, it's me, your BASIC Stamp!"
```

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 re-run the program you loaded into it. 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 is used to send 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 *FirstProgramYourTurn.bs2*
- ✓ Modify the comments at the beginning of the program so that they read:


```
' What's a Microcontroller - FirstProgramYourTurn.bs2
' BASIC Stamp sends messages to 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 1-42.

- ✓ Run your modified program. You will have to either select *Run* from the Run menu again, like in Figure 1-39 or click the *Run* button, like in Figure 1-41.

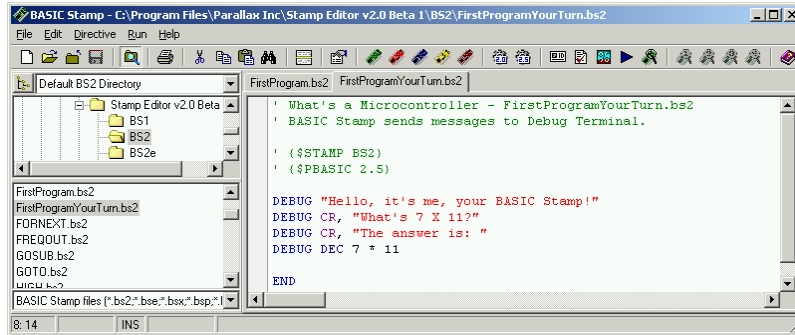


Figure 1-42
Modified
FirstProgram

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

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 1-43, the *Debug Terminal 1* shortcut button shown at the right of the figure, or the F12 key on your keyboard.

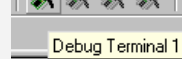
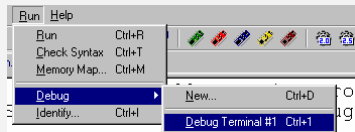


Figure 1-43

Debug Terminal 1 to
Foreground

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

Your Debug Terminal should now resemble Figure 1-44.

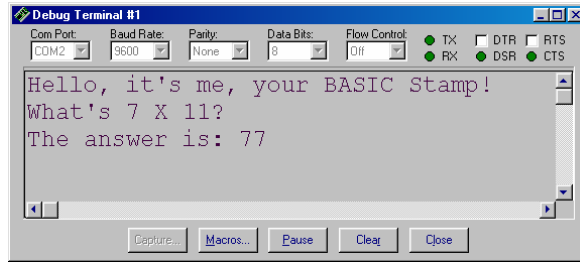


Figure 1-44
Modified
FirstProgram.bs2
Debug Terminal Output

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

ACTIVITY #5: LOOKING UP ANSWERS

The two activities 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* (Figure 1-45).

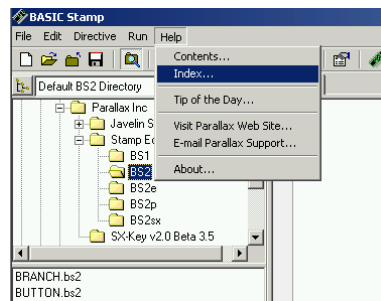


Figure 1-45
Selecting Index
from the Help
Menu

- ✓ Type **DEBUG** in the field labeled *Type in the keyword to find:* (Figure 1-46).
- ✓ When the word **DEBUG** appears in the list below where you are typing, click it, then click the *Display* button.

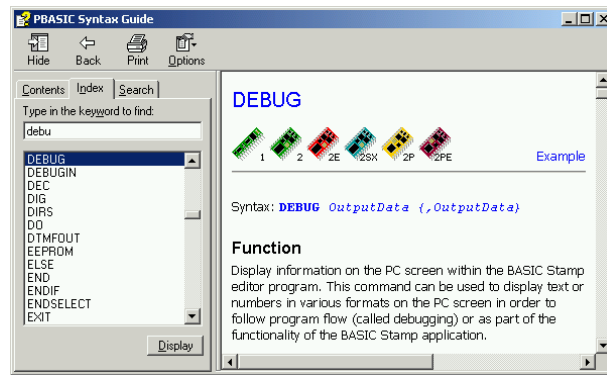


Figure 1-46
Looking up
the DEBUG
Command
Using Help.

Your Turn

- ✓ Use the scrollbar to review the **DEBUG** command's write-up. 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 printed book.

	Downloading the BASIC Stamp Manual from the Parallax Web Site
	✓ Using a web browser, go to www.parallax.com .
	✓ Point at the <i>Downloads</i> menu to display the options.
	✓ Point at the <i>Documentation</i> link and click to select it.
	✓ On the BASIC Stamp Documentation page, find The BASIC Stamp Users Manual.
	✓ Click the <i>Download</i> icon that looks like a file folder to the right of the description: "BASIC Stamp User's 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 1-47 shows an excerpt from the *BASIC Stamp Manual* v2.0 Contents section, showing that information on the **DEBUG** command is found on page 97.

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

Figure 1-47
Finding the
DEBUG
Command in
the Table of
Contents

Figure 1-48 shows an excerpt from page 97 in the BASIC Stamp Manual v2.0. The **DEBUG** command is explained in detail here along with example programs to demonstrate how the **DEBUG** command can be used.

- ✓ Look over the *BASIC Stamp Manual*'s 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 1-48
Reviewing
the **DEBUG**
Command
in the
BASIC
Stamp
Manual

Your Turn

- ✓ Use the BASIC Stamp Manual's 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, 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 21.

```
'What's a Microcontroller - 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). First, your batteries will last longer if the system is not drawing power when you're not using it. Second, soon you will build circuits on the Board of Education or HomeWork Board prototyping area.



Circuit prototypes should never be left unattended with a battery or power supply connected. **Always disconnect the power** from your Board of Education or HomeWork Board before you walk away, even if you only plan on leaving it alone for a minute or two.

Disconnecting Power

With the Board of Education Rev C, disconnecting power is easy. If you are using the Board of Education Rev C, power is disconnected by moving the 3-position switch to position-0 by pushing it to the left as shown in Figure 1-49.

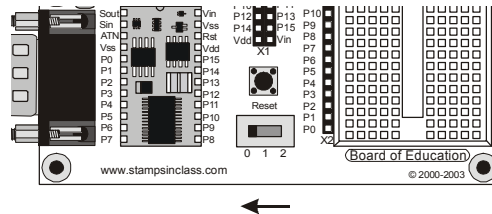


Figure 1-49
Turning the Power off

*Board of Education
Rev C*



Do not remove the BASIC Stamp module from its socket in the Board of Education! Every time the BASIC Stamp is removed and re-inserted into the socket on the Board of Education, you risk damaging it. You do not need to remove it for storage.

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 1-50.

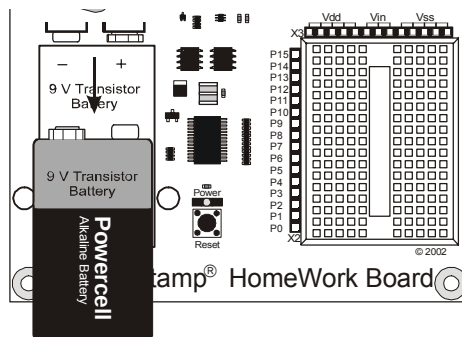


Figure 1-50
Disconnecting the
power to the HomeWork
Board

If you are using a Board of Education Rev B, you will not have a 3-position switch. Either unplug power supply, or remove the 9 V battery, whichever you are using.

Your Turn

- ✓ Disconnect the power to your board now.

SUMMARY

This chapter guided you through the following:

- An introduction to some devices that contain microcontrollers
- An introduction to the BASIC Stamp module
- A tour of some interesting inventions made with BASIC Stamp modules.
- 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 module, Board of Education, and HomeWork Board
- How to set up your BASIC Stamp hardware
- How to test your software and hardware
- How to write and run a PBASIC program
- Using the **DEBUG** and **END** commands
- Using the **CR** control character and **DEC** formatter
- How to use the BASIC Stamp Editor's Help and the BASIC Stamp Manual
- A brief introduction to ASCII code
- How to disconnect the power to your Board of Education or HomeWork Board when you're done.

Questions

1. What is a microcontroller?
2. Is the BASIC Stamp module a microcontroller, or does it contain one?
3. What clues would you look for to figure out whether or not an appliance like a clock radio or a cell phone contains a microcontroller?
4. What does an apostrophe at the beginning of a line of PBASIC program code signify?
5. What PBASIC commands did you learn in this chapter?
6. Let's say you want to take a break from your BASIC Stamp project to go get a snack, or maybe you want to take a longer break and return to the project in a couple days. What should you always do before you take your break?

Exercises

1. Explain what the asterisk does in this command:

```
DEBUG DEC 7 * 11
```

2. Guess what the Debug Terminal would display if you ran this command:

```
DEBUG DEC 7 + 11
```

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

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

Projects

1. Use **DEBUG** to display the solution to the math problem: $1 + 2 + 3 + 4$.
2. Which lines can you delete in FirstProgramYourTurn.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 FirstProgramYourTurn.bs2 with a new name, like FirstProgramCh01Project05.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. A microcontroller is a kind of miniature computer found in electronic products.
- Q2. The BASIC Stamp module contains a microcontroller called the PIC16C57.
- Q3. If the appliance has buttons and a digital display, these are good clues that it has a microcontroller inside.
- Q4. A comment.
- Q5. **DEBUG** and **END**
- Q6. Disconnect the power from the BASIC Stamp project.
- E1. It multiplies the two operands 7 and 11, resulting in a product of 77. The asterisk is the multiply operator.
- E2. The Debug Terminal would display: 18
- E3. To fix the problem, add a carriage return, the **CR**, control character.
- P1. Here is a program to display a solution to the math problem: $1+2+3+4$

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

```

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

END

```

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

```

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

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

DEBUG "Hello, it's me, your BASIC Stamp!", CR

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

END

```

The output from the Debug Terminal is:

```

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

```

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

Further Investigation

In this chapter, you visited the Software section of either the Parallax web site or the Parallax CD to get a copy of the BASIC Stamp Editor. You can go to the Documentation sections of either the Parallax web site or the Parallax CD to get a free copy of this text and of the *BASIC Stamp Manual*. Printed copies can also be purchased from Parallax.

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

You can learn much more about the **DEBUG** and **END** commands by looking them up in the *BASIC Stamp Manual*. You can find them using the Table of Contents. The *BASIC Stamp Manual* has many more examples you can try, along with lessons similar to those in the Projects section you just completed.

Chapter #2: Lights On – Lights Off

INDICATOR LIGHTS

Indicator lights are so common that most people tend not to give them much thought. Figure 2-1 shows three indicator lights on a laser printer. Depending on which light is on, the person using the printer knows if it is running properly or needs attention. Here are just a few examples of devices with indicator lights: car stereos, televisions, VCRs, disk drives, printers, and alarm system control panels.



Figure 2-1
Indicator
Lights

Turning an indicator light on and off is a simple matter of connecting and disconnecting it from a power source. In some cases, the indicator light is connected directly to the battery or power supply, like the power indicator light on the Board of Education. Other indicator lights are switched on and off by a microcontroller inside the device. These are usually status indicator lights that tell you what the device is up to.

MAKING A LIGHT EMITTING DIODE (LED) EMIT LIGHT

Most of the indicator lights you see on devices are called light emitting diodes. You will often see a light emitting diode referred to in books and circuit diagrams by the letters LED. The name is usually pronounced as three letters: “L-E-D”. You can build an LED circuit and connect power to it, and the LED emits light. You can disconnect the power from an LED circuit, and the LED stops emitting light.

An LED circuit can be connected to the BASIC Stamp, and the BASIC Stamp can be programmed to connect and disconnect the LED circuit's power. This is much easier than manually changing the circuit's wiring or connecting and disconnecting the battery. The BASIC Stamp can also be programmed to do the following:

- Turn an LED circuit on and off at different rates
- Turn an LED circuit on and off a certain number of times
- Control more than one LED circuit
- Control the color of a bi-color (two color) LED circuit

ACTIVITY #1: BUILDING AND TESTING THE LED CIRCUIT

It's important to test components individually before building them into a larger system. This activity focuses on building and testing two different LED circuits. The first circuit is the one that makes the LED emit light. The second circuit is the one that makes it not emit light. In the activity that comes after this one, you will build the LED circuit into a larger system by connecting it to the BASIC Stamp. You will then write programs that make the BASIC Stamp cause the LED to emit light, then not emit light. By first testing each LED circuit to make sure it works, you can be more confident that it will work when you connect it to a BASIC Stamp.

Introducing the Resistor

A resistor is a component that 'resists' the flow of electricity. This flow of electricity is called current. Each resistor has a value that tells how strongly it resists current flow. This resistance value is called the ohm, 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 jagged line symbol on the left 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 identify the resistor in your kit.

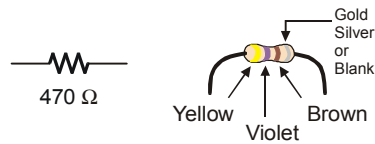


Figure 2-2
470 Ω Resistor Part
Drawing

*Schematic symbol (left)
and Part Drawing (right)*

Resistors like the ones we are using in this activity have colored stripes that tell you what their resistance values are. There is a different color combination for each resistance value. For example, the color code for the 470 Ω resistor is yellow-violet-brown.

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

Each color bar that tells you the resistor's value corresponds to a digit, and these colors/digits are listed in Table 2-1. Figure 2-3 shows how to use each color bar with the table to determine the value of a resistor.

Table 2-1: Resistor Color Code Values	
Digit	Color
0	Black
1	Brown
2	Red
3	Orange
4	Yellow
5	Green
6	Blue
7	Violet
8	Gray
9	White

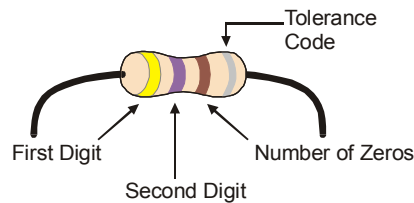


Figure 2-3
Resistor Color
Codes

Here is an example that shows how Table 2-1 and Figure 2-3 can be used to figure out a resistor value by proving that yellow-violet-brown is really 470 Ω :

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

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

Introducing the LED

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 2-4 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, paying attention to make sure the leads connected to the anode and cathode 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 the part drawing, the cathode lead is the unlabeled pin, and on the schematic symbol, the cathode is the line across the point of the triangle.

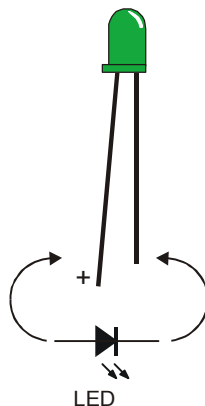


Figure 2-4
LED Part Drawing
and Schematic
Symbol

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

*The LED's 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. For the part drawing, note that the LED's leads are different lengths. The longer lead is connected to the LED's anode, and the shorter lead is connected to its cathode. Also, if you look closely at the LED's plastic case, it's mostly round, but there is a small flat spot right near the shorter lead that tells you it's the cathode. This really comes in handy if the leads have been clipped to the same length.

LED Test Circuit Parts

- (1) LED – Green
- (1) Resistor – 470 Ω (yellow-violet-brown)



Identifying the parts: In addition to the part drawings in Figure 2-2 and Figure 2-4, you can use the photo on the last page of the book to help identify the parts in the kit needed for this and all other activities. For more information on the parts in this photo, see Appendix B: Equipment and Parts Lists.

Building the LED Test Circuit

You will build a circuit by plugging the LED and resistor leads into small holes called sockets on the prototyping area shown in Figure 2-5. This prototyping area has black sockets along the top and along the left. The black sockets along the top have labels above them: Vdd, Vin, and Vss. These are called the power terminals, and they will be used to supply your circuits with electricity. The black sockets on the left have labels like P0, P1, up through P15. These are sockets that you can use to connect your circuit to the BASIC Stamp module's input/output pins. The white board with lots of holes in it is called a solderless breadboard. You will use this breadboard to connect components to each other and build circuits.

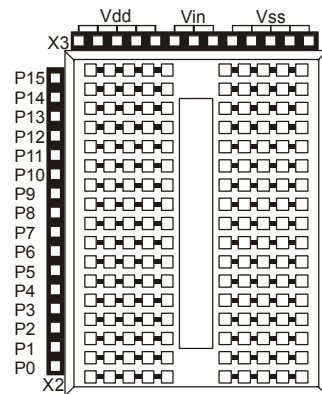


Figure 2-5
Prototyping Area

Power terminals (black sockets along top), I/O pin access (black sockets along the side), and solderless breadboard (white sockets)



Input/output pins are usually called I/O pins, and after connecting your circuit to one or more of these I/O pins, you can program your BASIC Stamp to monitor the circuit (input) or send on or off signals to the circuit (output). You will try this in the next activity.

Figure 2-6 shows a circuit schematic, and a picture of how that circuit will look when it is built on the prototyping area. The breadboard is separated into rows of five sockets. Each row can connect up to five leads, or wires, to each other. For this circuit, the resistor and the LED are connected because each one has a lead plugged into the same 5-socket row. Note that one lead of the resistor is plugged into Vdd so the circuit can draw power. The other resistor lead connects to the LED's anode lead. The LED's cathode lead is connected to Vss, or ground, completing the circuit.

You are now ready to build the circuit shown in Figure 2-6 (below) by plugging the LED and resistor leads into sockets on the prototyping area. Follow these steps:

- ✓ Disconnect power from your Board of Education or HomeWork Board.
- ✓ Use Figure 2-4 to decide which lead is connected to the LED's cathode. Look for the shorter lead and the flat spot on the plastic part of the LED.
- ✓ Plug the LED's cathode into one of the black sockets labeled Vss on the prototyping area.
- ✓ Plug the LED's anode (the other, longer lead) into the socket shown on the breadboard portion of the prototyping area.
- ✓ Plug one of the resistor's leads into the same breadboard row as the LED's anode. This will connect those two leads together.

- ✓ Plug the resistor's other lead into one of the sockets labeled Vdd.



Direction does matter for the LED, but not for the resistor. If you plug the LED in backward, the LED will not emit light when you connect power. The resistor just resists the flow of current. There is no backwards or forwards for a resistor.

2

- ✓ Reconnect power to your Board of Education or HomeWork Board.
- ✓ Check to make sure your green LED is emitting light. It should glow green.

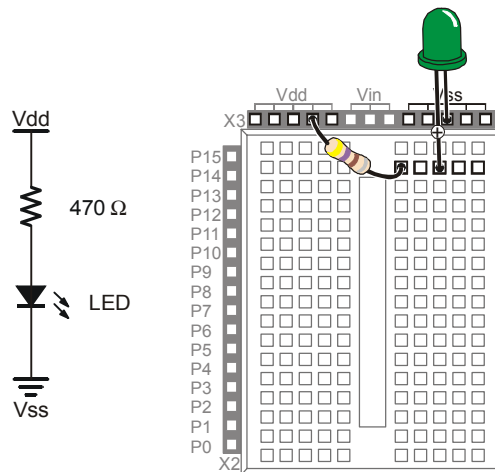


Figure 2-6
LED On

*Schematic (left) and
Wiring Diagram (right)*

If your green LED does not emit light when you connect power to the board:

- ✓ Some LEDs are brightest when viewed from above. Try looking straight down onto the dome part of the LED's plastic case from above.
- ✓ If the room is bright, try turning off some of the lights, or use your hands to cast a shadow on the LED.

If you still do not see any green glow, try these steps:

- ✓ Double check to make sure your cathode and anode are connected properly. If not, simply remove the LED, give it a half-turn, and plug it back in. It will not hurt the LED if you plug it in backwards, it just doesn't emit light. When you have it plugged in the right direction, it should emit light.

- ✓ Double check to make sure you built your circuit exactly as shown in Figure 2-6.
- ✓ If you are using a What's a Microcontroller kit that somebody used before you, the LED may be damaged, so try a different one.
- ✓ If you are in a lab class, check with your instructor.



Still stuck? If you don't have an instructor or friend who can help, you can always check with the Stamps in Class discussion group. The first pages of this book has Internet Access information on where to find the Stamps in Class discussion group. If the group is unable to help you solve the problem, you can contact the Parallax Technical Support department by following the Support link at www.parallax.com.

How the LED Test Circuit Works

The Vdd and Vss terminals supply electrical pressure in the same way that a battery would. The Vdd sockets are like the battery's positive terminal, and the Vss sockets are like the battery's negative terminal. Figure 2-7 shows how applying electrical pressure to a circuit using a battery causes electrons to flow through it. This flow of electrons is called electric current, or often just current. Electric current is limited by the resistor. This current is what causes the diode to emit light.

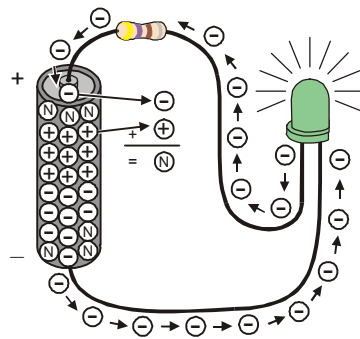


Figure 2-7
LED On Circuit Electron
Flow

The minus signs with the circles around them are used to show electrons flowing from the battery's negative terminal to its positive terminal.



Chemical reactions inside the battery supply the circuit with current. The battery's negative terminal contains a compound that has molecules with extra electrons (shown in Figure 2-7 by minus-signs). The battery's positive terminal has a chemical compound with molecules that are missing electrons (shown by plus-signs). When an electron leaves a molecule in the negative terminal and travels through the wire, it is called a free electron (also shown by minus-signs). The molecule that lost that extra electron no longer has an extra negative charge; it is now called neutral (shown by an N). When an electron gets to the positive terminal, it joins a molecule that was missing an electron, and now that molecule is neutral too.

Figure 2-8 shows how the flow of electricity through the LED circuit is described using schematic notation. The electric pressure across the circuit is called voltage. The + and – signs are used to show the voltage applied to a circuit. The arrow shows the current flowing through the circuit. This arrow is almost always shown pointing the opposite direction of the actual flow of electrons. Benjamin Franklin is credited with not having been aware of electrons when he decided to represent current flow as charge passing from the positive to negative terminal of a circuit. By the time physicists discovered the true nature of electric current, the convention was already well established.

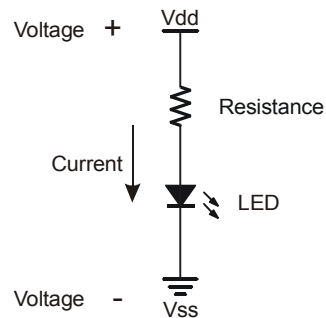


Figure 2-8
LED-On Circuit
Schematic Showing
Conventional Voltage
and Current Flow

*The + and – signs show
voltage applied to the
circuit, and the arrow
shows current flow
through the circuit.*



A schematic drawing (like Figure 2-8) is a picture that explains how one or more circuits are connected. Schematics are used by students, electronics hobbyists, electricians, engineers, and just about everybody else who works with circuits.

Appendix F: More about Electricity: This appendix contains some glossary terms and an activity you can try to get more familiar with measurements of voltage, current and resistance.

Your Turn – Modifying the LED Test Circuit

In the next activity, you will program the BASIC Stamp to turn the LED on, then off, then on again. The BASIC Stamp will do this by switching the LED circuit between two different connections, Vdd and Vss. You just finished working with the circuit where the resistor is connected to Vdd, and the LED emits light. Make the changes shown in Figure 2-9 to verify that the LED will turn off (not emit light) when the resistor's lead is disconnected from Vdd and connected to Vss.

- ✓ Disconnect power from your Board of Education or HomeWork Board.
- ✓ Unplug the resistor lead that's plugged into the Vdd socket, and plug it into a socket labeled Vss as shown in Figure 2-9.
- ✓ Reconnect power to your Board of Education or HomeWork Board.
- ✓ Check to make sure your green LED is **not emitting light**. It should not glow green.

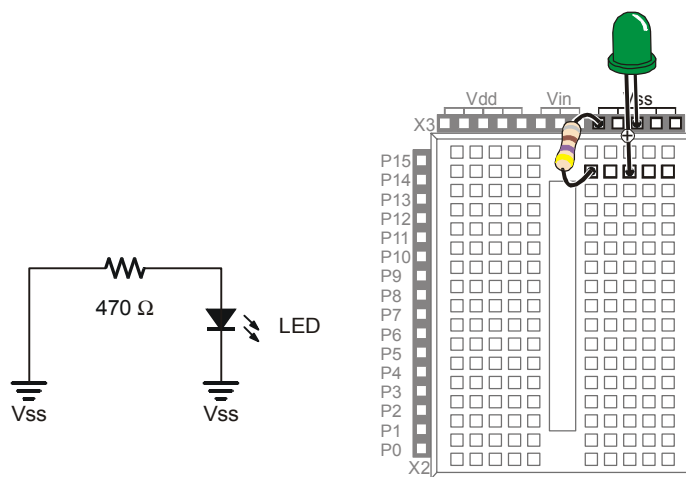


Figure 2-9
LED Off Circuit

*Schematic (left) and
wiring diagram (right).*

ACTIVITY #2: ON/OFF CONTROL WITH THE BASIC STAMP

In Activity #1, two different circuits were built and tested. One circuit made the LED emit light while the other did not. Figure 2-10 shows how the BASIC Stamp can do the same thing if you connect an LED circuit to one of its I/O pins. In this activity, you will connect the LED circuit to the BASIC Stamp and program it to turn the LED on and off.

You will also experiment with programs that make the BASIC Stamp do this at different speeds.

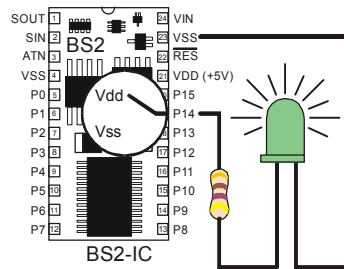
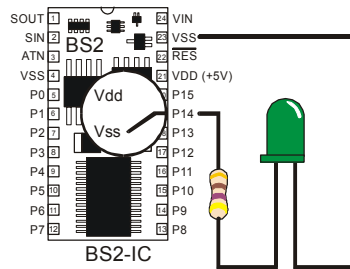


Figure 2-10
BASIC Stamp
Switching

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

There are two big differences between changing the connection manually and having the BASIC Stamp do it. First, the BASIC Stamp doesn't have to cut the power when it changes the LED circuit's supply from Vdd to Vss. Second, while a human can make that change several times a minute, the BASIC Stamp can do it thousands of times per second!

LED Test Circuit Parts

Same as Activity #1.

Connecting the LED Circuit to the BASIC Stamp

The LED circuit shown in Figure 2-11 is wired almost the same as the circuit in the previous exercise. The difference is that the resistor's lead that was manually switched between Vdd and Vss is now plugged into a BASIC Stamp I/O pin.

- ✓ Disconnect power from your Board of Education or HomeWork Board.
- ✓ Modify the circuit you were working with in Activity #1 so that it matches Figure 2-11.

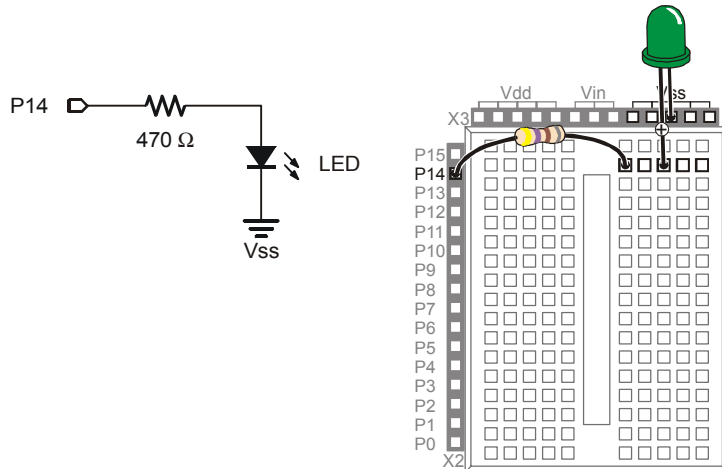


Figure 2-11
BASIC Stamp
Controlled LED
Circuit

*The LED circuit's
input is now
connected to a
BASIC Stamp I/O
pin instead of Vdd
or Vss.*



Resistors are essential. Always remember to use a resistor. Without it, too much current will flow through the circuit, and it could damage any number of parts in your circuit, BASIC Stamp, or Board of Education or HomeWork Board.

Turning the LED On/Off with a Program

The example program makes the LED blink on and off one time per second. It introduces several new programming techniques at once. After running it, you will experiment with different parts of the program to better understand how it works.

Example Program: LedOnOff.bs2

- ✓ Enter the LedOnOff.bs2 code into the BASIC Stamp Editor.
- ✓ Reconnect power to your Board of Education or HomeWork Board.
- ✓ Run the program.
- ✓ Verify that the LED flashes on and off once per second.
- ✓ Disconnect power when you are done with the program.

```
'What's a Microcontroller - LedOnOff.bs2
'Turn an LED on and off. Repeat 1 time per second indefinitely.

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

DEBUG "The LED connected to Pin 14 is blinking!"
```

```
DO
  HIGH 14
  PAUSE 500
  LOW 14
  PAUSE 500
LOOP
```

How LedOnOff.bs2 Works

The command **DEBUG** "The LED connected to Pin 14 is blinking!" makes this statement appear in the Debug Terminal. The command **HIGH 14** causes the BASIC Stamp to internally connect I/O pin P14 to Vdd. This turns the LED on.

The command **PAUSE 500** causes the BASIC Stamp to do nothing for ½ a second while the LED stays on. The number 500 tells the **PAUSE** command to wait for 500/1000 of a second. The number that follows **PAUSE** is called an argument. If you look up **PAUSE** in the BASIC Stamp Manual, you will discover that it calls this number the *Duration* argument. The name duration was chosen for this argument to show that the **PAUSE** command pauses for a certain 'duration' of time, in milliseconds.



What's a Millisecond? A millisecond is 1/1000 of a second. It is abbreviated as ms. It takes 1000 ms to equal one second.

The command **LOW 14** causes the BASIC Stamp to internally connect I/O pin P14 to Vss. This turns the LED off. Since **LOW 14** is followed by another **PAUSE 500**, the LED stays off for half a second.

The reason the code repeats itself over and over again is because it is nested between the PBASIC keywords **DO** and **LOOP**. Figure 2-12 shows how a **DO...LOOP** works. By placing the code segment that turns the LED on and off with pauses between **DO** and **LOOP**, it tells the BASIC Stamp to execute those four commands over and over again. The result is that the LED flashes on and off, over and over again. It will keep flashing until you disconnect power, press and hold the Reset button, or until the battery runs out.

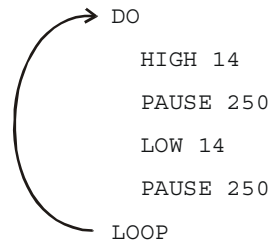


Figure 2-12
DO...LOOP

The code between the keywords DO and LOOP get executed over and over again.

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 `LedOnOff.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.

Your Turn – Timing and Repetitions

By changing the **PAUSE** command's *Duration* argument you can change the amount of time the LED stays on and off. For example, by changing both the *Duration* arguments to 250, it will cause the LED to flash on and off twice per second. The **DO...LOOP** in your program will now look like this:

```
DO

    HIGH 14
    PAUSE 250
    LOW 14
    PAUSE 250

LOOP
```

- ✓ Open LedOnOff.bs2
- ✓ Change the **PAUSE** command's *Duration* arguments from 500 to 250, and re-run the program.

If you want to make the LED blink on and off once every three seconds, with the low time twice as long as the high time, you can program the **PAUSE** command after the **HIGH 14** command so that it takes one second using **PAUSE 1000**. The **PAUSE** command after the **LOW 14** command will have to be **PAUSE 2000**.

```
DO

    HIGH 14
    PAUSE 1000
    LOW 14
    PAUSE 2000

LOOP
```

- ✓ Modify and re-run the program using the code snippet above.

A fun experiment is to see how short you can make the pauses and still see that the LED is flashing. When the LED is flashing very fast, but it looks like it's just on, it's called

persistence of vision. Here is how to test to see what your persistence of vision threshold is:

- ✓ Try modifying both of your **PAUSE** command's *Duration* arguments so that they are 100.
- ✓ Re-run your program and check for flicker.
- ✓ Reduce both *Duration* arguments by 5 and try again.
- ✓ Keep reducing the *Duration* arguments until the LED appears to be on all the time with no flicker. It will be dimmer than normal, but it should not appear to flicker.

One last thing to try is to create a one-shot LED flasher. When the program runs, the LED flashes only once. This is a way to look at the functionality of the **DO...LOOP**. You can temporarily remove the **DO...LOOP** from the program by placing an apostrophe to the left of both the **DO** and **LOOP** keywords as shown below.

```
' DO
HIGH 14
PAUSE 1000
LOW 14
PAUSE 2000

' LOOP
```

- ✓ Modify and re-run the program using the code snippet above.
- ✓ Explain what happened, why did the LED only flash once?



Commenting a line of code: Placing an apostrophe to the left of a command changes it into a comment. This is a useful tool because you don't actually have to delete the command to see what happens if you remove it from the program. It is much easier to add and remove an apostrophe than it is to delete and re-type the commands.

ACTIVITY #3: COUNTING AND REPEATING

In the previous activity, the LED circuit either flashed on and off all the time, or it flashed once and then stopped. What if you only want the LED to flash on and off ten times? Computers (including the BASIC Stamp) are great at keeping running totals of how many times something happens. Computers can also be programmed to make

decisions based on a variety of conditions. In this activity, you will program the BASIC Stamp to stop flashing the LED on and off after ten repetitions.

Counting Parts and Test Circuit

Use the example circuit shown in Figure 2-11 on page 48.

How Many Times?

There are many ways to make the LED blink on and off ten times. The simplest way is to use a **FOR...NEXT** loop. The **FOR...NEXT** loop is similar to the **DO...LOOP**. Although either loop can be used to repeat commands a fixed number of times, **FOR...NEXT** is easier to use.

The **FOR...NEXT** loop depends on a variable to track how many times the LED has blinked on and off. A variable is a word of your choosing that is used to store a value. The next example program uses the word **counter** to ‘count’ how many times the LED has been turned on and off.



Picking words for variable names has several rules:

1. The name cannot be a word that is already used by PBASIC. These words are called reserved words, and some examples that you should already be familiar with are: **DEBUG**, **PAUSE**, **HIGH**, **LOW**, **DO**, and **LOOP**.
2. The name cannot contain a space.
3. Even though the name can contain letters, numbers, or underscores, it must begin with a character.
4. The name must be less than 33 characters long.

Example Program: LedOnOffTenTimes.bs2

The program LedOnOffTenTimes.bs2 demonstrates how to use a **FOR...NEXT** loop to blink an LED on and off ten times.

- ✓ Your test circuit from Activity #2 should be built (or rebuilt) and ready to use.
- ✓ Enter the LedOnOffTenTimes.bs2 code into the BASIC Stamp Editor.
- ✓ Connect power to your Board of Education or HomeWork Board.
- ✓ Run the program.
- ✓ Verify that the LED flashes on and off ten times.

- ✓ Run the program a second time, and verify that the value of **counter** shown in the Debug Terminal accurately tracks how many times the LED blinked. Hint: instead of clicking *Run* a second time, you can press and release the Reset button on your Board of Education or HomeWork Board.

```
' What's a Microcontroller - LedOnOffTenTimes.bs2
' Turn an LED on and off. Repeat 10 times.

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

counter VAR Byte

FOR counter = 1 TO 10

    DEBUG ? counter

    HIGH 14
    PAUSE 500
    LOW 14
    PAUSE 500

NEXT

DEBUG "All done!"

END
```

How LedOnOffTenTimes.bs2 Works

This PBASIC statement

```
counter VAR Byte
```

tells the BASIC Stamp Editor that your program will use the word **counter** as a variable that can store a byte's worth of information.

What's a Byte? A byte is enough memory to store a number between 0 and 255. The BASIC Stamp has four different types of variables, and each can store a different range of numbers:

Table 2-2: Variable Types and Values They Can Store

Variable type	Range of Values
Bit	0 to 1
Nib	0 to 15
Byte	0 to 255
Word	0 to 65535



The question mark formatter before a variable in a **DEBUG** command tells the Debug Terminal to display the name of the variable and its value. This is how the command

```
DEBUG ? counter
```

displays both the name and the value of the **counter** variable in the Debug Terminal.

The **FOR...NEXT** loop and all the commands inside it are shown below. The statement **FOR counter = 1 to 10** tells the BASIC Stamp that it will have to set the **counter** variable to 1, then keep executing commands until it gets to the **NEXT** statement. When the BASIC Stamp gets to the **NEXT** statement, it jumps back to the **FOR** statement. The **FOR** statement adds one to the value of **counter**. Then, it checks to see if **counter** is greater than ten yet. If not, it repeats the process. When the value of **counter** finally reaches eleven, the program skips the commands between the **FOR** and **NEXT** statements and moves on to the command that comes after the **NEXT** statement.

```
FOR counter = 1 to 10

  DEBUG ? counter, CR

  HIGH 14
  PAUSE 500
  LOW 14
  PAUSE 500

NEXT
```

The command that comes after the **NEXT** statement is:

```
DEBUG "All done!"
```

This command is included just to show what the program does after ten times through the **FOR...NEXT** loop. It moves on to the command that comes after the **NEXT** statement.

Your Turn – Other Ways to Count

✓ Replace the statement

```
FOR counter = 1 to 10      with this:      FOR counter = 1 to 20
```

in `LedOnOffTenTimes.bs2` and re-run the program. What did the program do differently, and was this expected?

✓ Try a second modification to the **FOR** statement. This time, change it to

```
FOR counter = 20 to 120 STEP 10
```

How many times did the LED flash? What values displayed in the Debug Terminal?

ACTIVITY #4: BUILDING AND TESTING A SECOND LED CIRCUIT

Indicator LEDs can be used to tell the machine's user many things. Many devices need two, three, or more LEDs to tell the user if the machine is ready or not, if there is a malfunction, if it's done with a task, and so on.

In this activity, you will repeat the LED circuit test in Activity #1 for a second LED circuit. Then you will adjust the example program from Activity #2 to make sure the LED circuit is properly connected to the BASIC Stamp. After that, you will modify the example program from Activity #2 to make the LEDs operate in tandem.

Extra Parts

In addition to the parts you used in Activities 1 and 2, you will need these parts:

- (1) LED - yellow
- (1) Resistor – 470 Ω (yellow-violet-brown)

Building and Testing the Second LED Circuit

In Activity #1, you manually tested the first LED circuit to make sure it worked before connecting it to the BASIC Stamp. Before connecting the second LED circuit to the BASIC Stamp, it's important to test it too.

- ✓ Disconnect power from your Board of Education or HomeWork Board.
- ✓ Construct the second LED circuit as shown in Figure 2-13.
- ✓ Reconnect power to your Board of Education or HomeWork Board.
- ✓ Did the LED circuit you just added turn on? If yes, then continue. If no, Activity #1 has some trouble-shooting suggestions that you can repeat for this circuit.

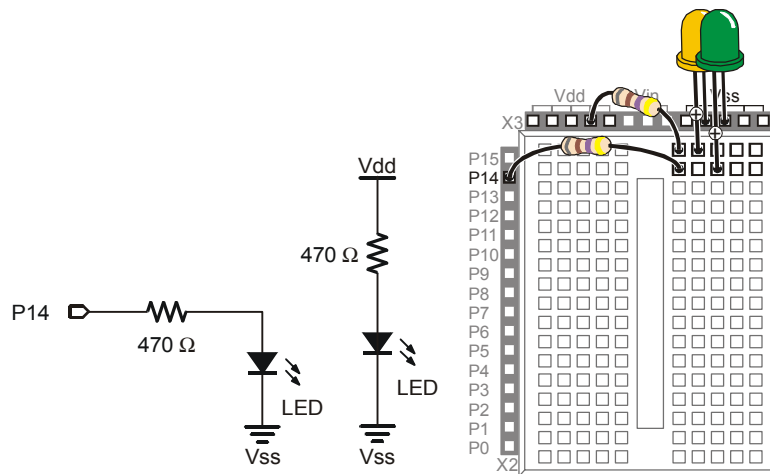


Figure 2-13
Manual Test
Circuit for
Second LED

- ✓ Disconnect power to your Board of Education or HomeWork Board.
- ✓ Modify the second LED circuit you just tested by connecting the LED circuit's resistor lead (input) to P15 as shown in Figure 2-14.

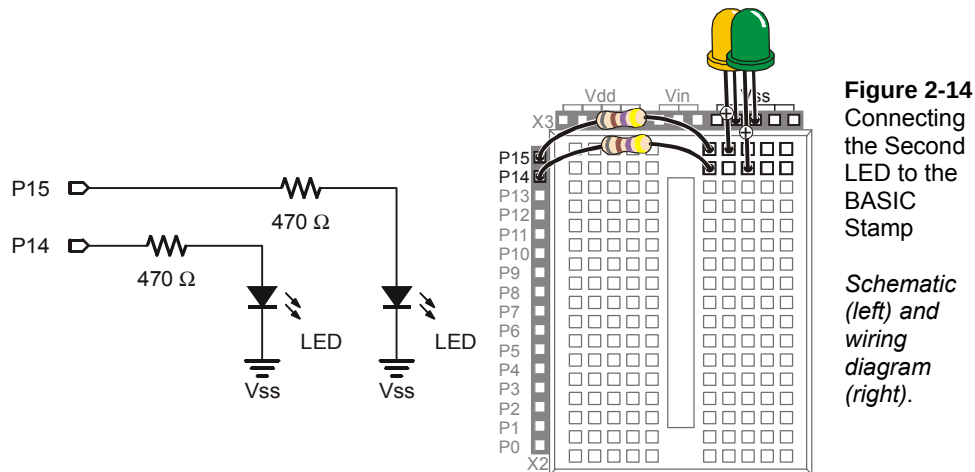


Figure 2-14
Connecting
the Second
LED to the
BASIC
Stamp

*Schematic
(left) and
wiring
diagram
(right).*

Using a Program to Test the Second LED Circuit

In Activity #2, you used an example program and the **HIGH** and **LOW** commands to control the LED circuit connected to P14. These commands will have to be modified to control the LED circuit connected to P15. Instead of using **HIGH 14** and **LOW 14**, you will use **HIGH 15** and **LOW 15**.

Example Program: TestSecondLed.bs2

- ✓ Enter TestSecondLed.bs2 into the BASIC Stamp Editor.
- ✓ Connect power to your Board of Education or HomeWork Board.
- ✓ Run TestSecondLED.bs2.
- ✓ Make sure the LED circuit connected to P15 is flashing. If the LED connected to P15 flashes, move on to the next example (Controlling Both LEDs). If the LED circuit connected to P15 is not flashing, check your circuit for wiring errors and your program for typing errors and try again.

```
' What's a Microcontroller - TestSecondLed.bs2
' Turn LED connected to P15 on and off.
' Repeat 1 time per second indefinitely.

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

DEBUG "Program Running!"
```

```
DO
```

```
  HIGH 15
  PAUSE 500
  LOW 15
  PAUSE 500
```

```
LOOP
```

Controlling Both LEDs

Yes, you can flash both LEDs at once. One way you can do this is to use two **HIGH** commands before the first **PAUSE** command. One **HIGH** command sets P14 high, and the next **HIGH** command sets P15 high. You will also need two **LOW** commands to turn both LEDs off. It's true that both LEDs will not turn on and off at exactly the same time because one is turned on or off after the other. However, there is no more than a millisecond's difference between the two changes, and the human eye will not detect it.

Example Program: FlashBothLeds.bs2

- ✓ Enter the FlashBothLeds.bs2 code into the BASIC Stamp Editor.
- ✓ Run the program.
- ✓ Verify that both LEDs appear to flash on and off at the same time.

```
' What's a Microcontroller - FlashBothLeds.bs2
' Turn LEDs connected to P14 and P15 on and off.

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

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

```
DO
```

```
  HIGH 14
  HIGH 15
  PAUSE 500
  LOW 14
  LOW 15
  PAUSE 500
```

```
LOOP
```

Your Turn – Alternate LEDs

You can cause the LEDs to alternate by swapping the **HIGH** and **LOW** commands that control one of the I/O pins. This means that while one LED is on, the other will be off.

- ✓ Modify FlashBothLeds.bs2 so that the commands between the **DO** and **LOOP** keywords look like this:

```
HIGH 14  
LOW 15  
PAUSE 500  
LOW 14  
HIGH 15  
PAUSE 500
```

- ✓ Run the modified version of FlashBothLeds.bs2 and verify that the LEDs flash alternately on and off.

ACTIVITY #5: USING CURRENT DIRECTION TO CONTROL A BI-COLOR LED

The device shown in Figure 2-15 is a security monitor for electronic keys. When an electronic key with the right code is used, the LED changes color, and a door opens. This kind of LED is called a bi-color LED. This activity answers two questions:

1. How does the LED change color?
2. How can you run one with the BASIC Stamp?



Figure 2-15
Bi-color LED in a
Security Device

When the door is locked, this bi-color LED glows red. When the door is unlocked by an electronic key with the right code, the LED turns green.

Introducing the Bi-Color LED

The bi-color LED's schematic symbol and part drawing are shown in Figure 2-16.

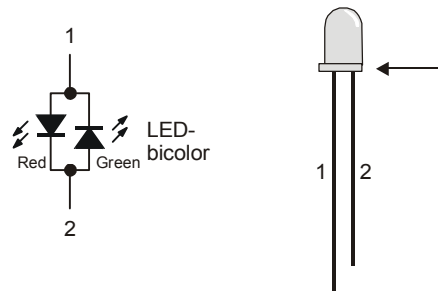
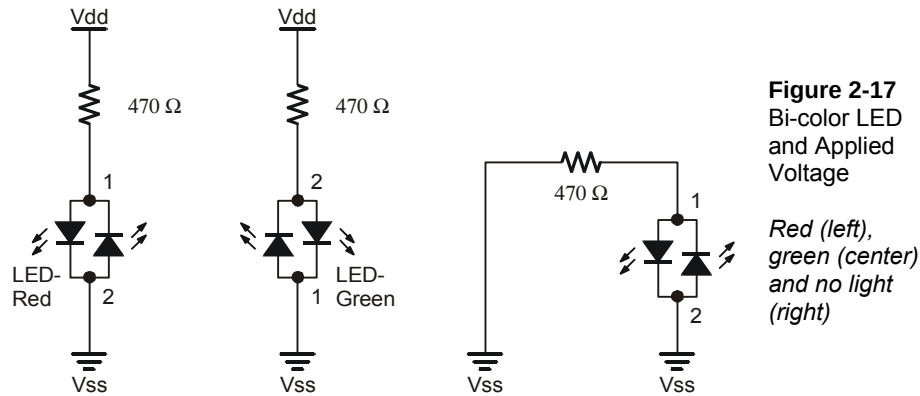


Figure 2-16
Bi-color LED

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

The bi-color LED is really just two LEDs in one package. Figure 2-17 shows how you can apply voltage in one direction and the LED will glow red. By disconnecting the LED and plugging it back in reversed, the LED will then glow green. As with the other LEDs, if you connect both terminals of the circuit to Vss, the LED will not emit light.



Bi-Color LED Circuit Parts

- (1) LED – bi-color
- (1) Resistor 470 Ω (yellow-violet-brown)
- (1) Jumper wire

Building and Testing the Bi-Color LED Circuit

Figure 2-18 shows the manual test for the bi-color LED.

- √ Disconnect power from your Board of Education or HomeWork Board.
- √ Build the circuit shown on the left side of Figure 2-18.
- √ Reconnect power and verify that the bi-color LED is emitting red light.
- √ Disconnect power again.
- √ Modify your circuit so that it matches the right side of Figure 2-18.
- √ Reconnect power.
- √ Verify that the bi-color LED is now emitting green light.
- √ Disconnect power.

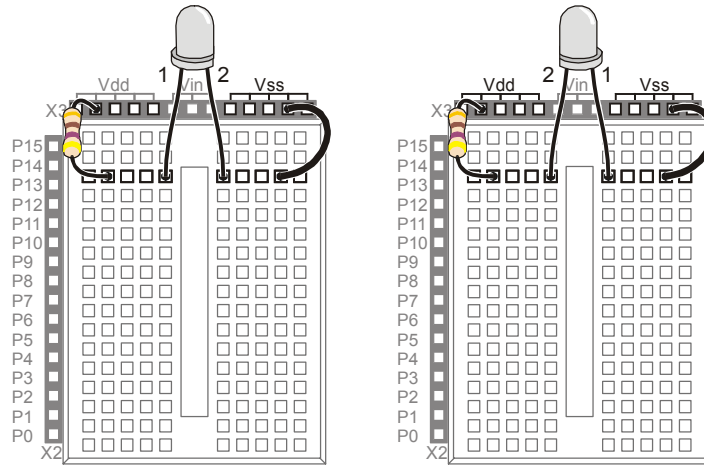


Figure 2-18
Manual bi-color LED Test

Bi-color LED red (left) and green (right).

Controlling a bi-color LED with the BASIC Stamp requires two I/O pins. After you have manually verified that the bi-color LED works using the manual test, you can connect the circuit to the BASIC Stamp as shown in Figure 2-19.

- ✓ Connect the bi-color LED circuit to the BASIC Stamp as shown in Figure 2-19.

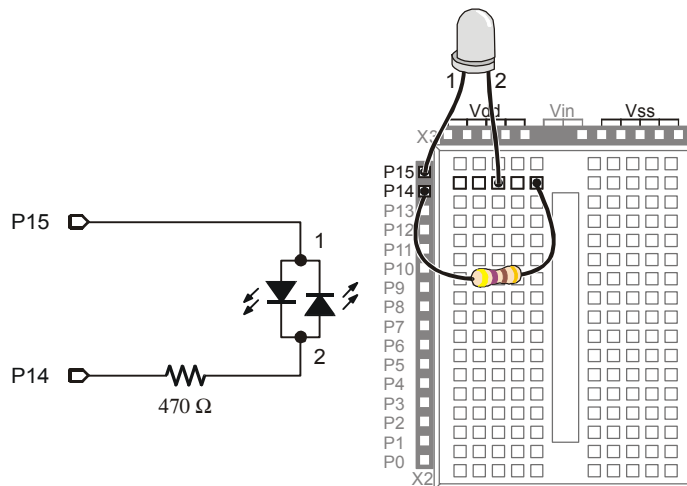


Figure 2-19
Bi-color LED Connected to BASIC Stamp

BASIC Stamp Bi-Color LED Control

Figure 2-20 shows how you can use P15 and P14 to control the current flow in the bi-color LED circuit. The upper schematic shows how current flows through the red LED when P15 is set to Vdd and P14 is set to Vss. This is because the red LED will let current flow through it when electrical pressure is applied as shown, but the green LED acts like a closed valve and does not let current through it. The bi-color LED glows red.

The lower schematic shows what happens when P15 is set to Vss and P14 is set to Vdd. The electrical pressure is now reversed. The red LED shuts off and does not allow current through. Meanwhile, the green LED turns on, and current passes through the circuit in the opposite direction.

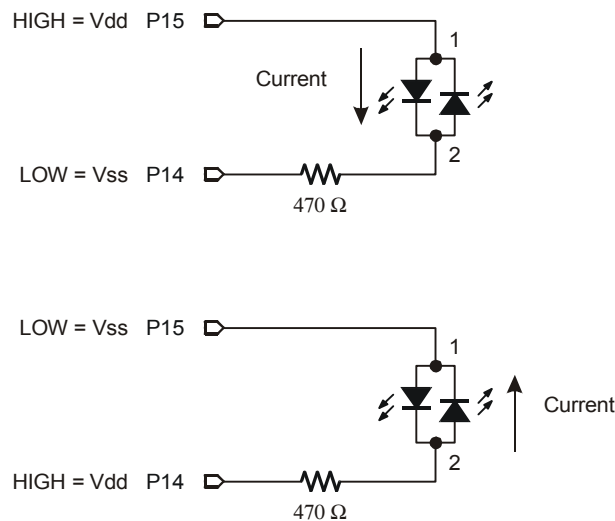


Figure 2-20
Manual bi-color
LED Test

*Current through
Red LED (above)
and Green LED
(below).*

Figure 2-20 also shows the key to programming the BASIC Stamp to make the bi-color LED glow two different colors. The upper schematic shows how to make the bi-color LED red using **HIGH 15** and **LOW 14**. The lower schematic shows how to make the bi-color LED glow green by using **LOW 15** and **HIGH 14**. To turn the LED off, send low signals to both P14 and P15 using **LOW 15** and **LOW 14**. In other words, use **LOW** on both pins.



The bi-color LED will also turn off if you send high signals to both P14 and P15. Why? Because the electrical pressure is the same at P14 and P15 regardless of whether you set both I/O pins high or low.

2

Example Program: TestBiColorLED.bs2

- ✓ Reconnect power.
- ✓ Enter the TestBiColorLed.bs2 code into the BASIC Stamp Editor.
- ✓ Run the program.
- ✓ Verify that the LED cycles through the red, green, and off states.

```
' What's a Microcontroller - TestBiColorLed.bs2
' Turn bi-color LED red, then green, then off in a loop.

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

DEBUG "Program Running!"

DO

  HIGH 15          ' Red
  LOW 14
  PAUSE 500

  LOW 15           ' Green
  HIGH 14
  PAUSE 500

  LOW 15           ' Off
  LOW 14
  PAUSE 500

LOOP
```

Your Turn – Lights Display

In Activity #3, a variable named **counter** was used to control how many times the LED blinked. What happens if you use the value **counter** to control the **PAUSE** command's *Duration* argument?

- ✓ Rename and save TestBiColorLed.bs2 as TestBiColorLedYourTurn.bs2.
- ✓ Add a counter variable declaration before the **DO** statement:


```
counter VAR BYTE
```

✓ Nest the **FOR...NEXT** loop below within the **DO...LOOP**.

```
FOR counter = 1 to 50
```

```
    HIGH 15
    LOW 14
    PAUSE counter
```

```
    LOW 15
    HIGH 14
    PAUSE counter
```

```
NEXT
```

When you are done, your code should look like this:

```
counter VAR BYTE
```

```
DO
```

```
    FOR counter = 1 to 50
```

```
        HIGH 15
        LOW 14
        PAUSE counter
```

```
        LOW 15
        HIGH 14
        PAUSE counter
```

```
    NEXT
```

```
LOOP
```

At the beginning of each pass through the **FOR...NEXT** loop, the **PAUSE** value (*Duration* argument) is only one millisecond. Each time through the **FOR...NEXT** loop, the pause gets longer by one millisecond at a time until it gets to 50 milliseconds. The **DO...LOOP** causes the **FOR...NEXT** loop to execute over and over again.

✓ Run the modified program and observe the effect.

SUMMARY

The BASIC Stamp can be programmed to switch a circuit with a light emitting diode (LED) indicator light on and off. LED indicators are useful in a variety of places including many computer monitors, disk drives, and other devices. The LED was introduced along with a technique to identify its anode and cathode terminals. An LED circuit must have a resistor to limit the current passing through it. Resistors were introduced along with one of the more common coding schemes you can use to figure out a resistor's value.

The BASIC Stamp switches an LED circuit on and off by internally connecting an I/O pin to either Vdd or Vss. The **HIGH** command can be used to make the BASIC Stamp internally connect one of its I/O pins to Vdd, and the **LOW** command can be used to internally connect an I/O pin to Vss. The **PAUSE** command is used to cause the BASIC Stamp to not execute commands for an amount of time. This was used to make LEDs stay on and/or off for certain amounts of time. This amount of time is determined by the number used in the **PAUSE** command's *Duration* argument.

The **DO...LOOP** can be used to create an infinite loop. The commands between the **DO** and **LOOP** keywords will execute over and over again. Even though this is called an infinite loop, the program can still be re-started by disconnecting and reconnecting power or pressing and releasing the Reset button. A new program can also be downloaded to the BASIC Stamp, and this will erase the program with the infinite loop.

Current direction and voltage polarity were introduced using a bi-color LED. If voltage is applied across the LED circuit, current will pass through it in one direction, and it glows a particular color. If the voltage polarity is reversed, current travels through the circuit in the opposite direction and it glows a different color.

Questions

1. What is the name of this Greek letter: Ω , and what measurement does Ω refer to?
2. Which resistor would allow more current through the circuit, a 470 Ω resistor or a 1000 Ω resistor?
3. How do you connect two wires using a breadboard? Can you use a breadboard to connect four wires together?

4. What do you always have to do before modifying a circuit that you built on a breadboard?
5. How long would **PAUSE 10000** last?
6. How would you cause the BASIC Stamp to do nothing for an entire minute?
7. What are the different types of variables?
8. Can a byte hold the value 500?
9. What will the command **HIGH 7** do?

Exercises

1. Draw the schematic of an LED circuit like the one you worked with in Activity #2, but connect the circuit to P13 instead of P14. Explain how you would modify LedOnOff.bs2 on Page 48 so that it will make your LED circuit flash on and off four times per second.
2. Explain how to modify LedOnOffTenTimes.bs2 so that it makes the LED circuit flash on and off 5000 times before it stops. Hint: you will need to modify just two lines of code.

Project

1. Make a 10-second countdown using one yellow LED and one bi-color LED. Make the bi-color LED start out red for 3 seconds. After 3 seconds, change the bi-color LED to green. When the bi-color LED changes to green, flash the yellow LED on and off once every second for ten seconds. When the yellow LED is done flashing, the bi-color LED should switch back to red and stay that way.

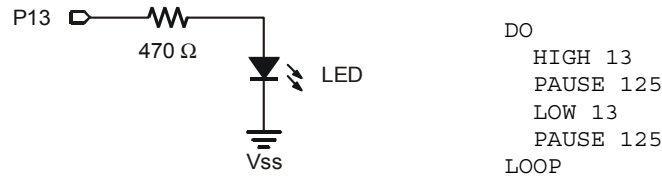
Solutions

- Q1. Omega refers to the ohm which measures how strongly something resists current flow.
- Q2. A 470 Ω resistor: higher values resist more strongly than lower values, therefore lower values allow more current to flow.
- Q3. To connect 2 wires, plug the 2 wires into the same row of 5 sockets. You can connect 4 wires by plugging all 4 wires into the same row of 5 sockets.
- Q4. Disconnect the power.
- Q5. 10 seconds.
- Q6. PAUSE 60000
- Q7. Bit, Nib, Byte, and Word

Q8. No. The largest value a byte can hold is 255. The value 500 is out of range for a byte.

Q9. **HIGH 7** will cause the BASIC Stamp to internally connect I/O pin P7 to Vdd.

E1. The **PAUSE Duration** must be reduced to $500\text{ms} / 4 = 125\text{ms}$. To use I/O pin P13, **HIGH 14** and **LOW 14** have been replaced with **HIGH 13** and **LOW 13**.

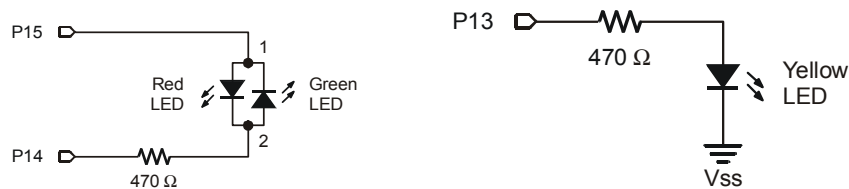


E2. The **counter** variable has to be changed to **Word** size, and the **FOR** statement has to be modified to count from 1 to 5000).

```

counter VAR Word
FOR counter = 1 to 5000
  DEBUG ? counter, CR
  HIGH 14
  PAUSE 500
  LOW 14
  PAUSE 500
NEXT
  
```

P1. The bi-color LED schematic, on the left, is unchanged from Figure 2-20 on page 64. The yellow LED schematic is based on Figure 2-11 on page 48. For this project P14 was changed to P13.



```

' What's a Microcontroller - Ch02Prj01_Countdown.bs2
' 10 Second Countdown with Red, Yellow, Green LED
' Red/Green: Bicolor LED on P15, P14. Yellow: P13
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Program Running!"

counter VAR Byte
  
```

```

' Red for three seconds
HIGH 15
LOW 14
PAUSE 3000

' Green for 10 seconds...
LOW 15
HIGH 14

' ...while the yellow LED is flashing
FOR counter = 1 TO 10
  HIGH 13
  PAUSE 500
  LOW 13
  PAUSE 500
NEXT

' Red stays on
HIGH 15
LOW 14

' Bi-color LED Red

' Bi-color LED Green

' Yellow LED on
' Yellow LED off

' Bi Color LED Red

```

Further Investigation

The resources listed here are available for free download from the Parallax web site and are also included on the Parallax CD.

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

The *BASIC Stamp Manual* has more examples you can try and information that further explains the following commands: **HIGH**, **LOW**, **PAUSE**, the **DEBUG ?** formatter, and **FOR...NEXT**.

“Basic Analog and Digital”, Student Guide, Version 2.0, Parallax Inc., 2003

Basic Analog and Digital uses LEDs to describe counting in binary, describes analog conditions, and it introduces new ways to adjust an LED's brightness.

“BASIC Stamp Editor Help File”, PBASIC 2.5 Version 2.0 Parallax Inc., 2003

The PBASIC 2.5 Help File has information on **DO...LOOP**, which is new to PBASIC 2.5 and not included in the BASIC Stamp Manual. You can find this information by clicking the book icon on your BASIC Stamp Editor task bar, then selecting PBASIC Reference from the menu in the left sidebar window. This will open the PBASIC Command Reference alphabetical listing in the main window. Detailed information can be found by clicking on each command.

Chapter #3: Digital Input - Pushbuttons

FOUND ON CALCULATORS, HAND HELD GAMES, AND APPLIANCES

How many devices with pushbuttons do you use on a daily basis? Here are a few examples that might appear in your list: computer, mouse, calculator, microwave oven, handheld remote, handheld games, and VCR. In each device, there is a microcontroller scanning the pushbuttons and waiting for the circuit to change. When the circuit changes, the microcontroller detects the change and takes action. By the end of this chapter, you will have experience with designing pushbutton circuits and programming the BASIC Stamp to monitor them and take action when changes occur.

RECEIVING VS. SENDING HIGH AND LOW SIGNALS

In Chapter #2, you programmed the BASIC Stamp to send high and low signals, and you used LED circuits to display these signals. Sending high and low signals means you used a BASIC Stamp I/O pin as an output. In this chapter, you will use a BASIC Stamp I/O pin as an input. As an input, an I/O pin listens for high/low signals instead of sending them. You will send these signals to the BASIC Stamp using a pushbutton circuit, and you will program the BASIC Stamp to recognize whether the pushbutton is pressed or not pressed.



Other terms that mean send, high/low, and receive: Sending high/low signals is described in different ways. You may see sending referred to as transmitting, controlling, or switching. Instead of high/low, you might see it referred to as binary, TTL, CMOS, or Boolean signals. Another term for receiving is sensing.

ACTIVITY #1: TESTING A PUSHBUTTON WITH AN LED CIRCUIT

If you can use a pushbutton to send a high or low signal to the BASIC Stamp, can you also control an LED with a pushbutton? The answer is yes, and you will use it to test a pushbutton in this activity.

Introducing the Pushbutton

Figure 3-1 shows the schematic symbol and the part drawing of a normally open pushbutton. Two of the pushbutton's pins are connected to each terminal. This means that connecting a wire or part lead to pin 1 of the pushbutton is the same as connecting it to pin 4. The same rule applies with pins 2 and 3. The reason the pushbutton doesn't just

have two pins is because it needs stability. If the pushbutton only had two pins, those pins would eventually bend and break from all the pressure that the pushbutton receives when people press it.

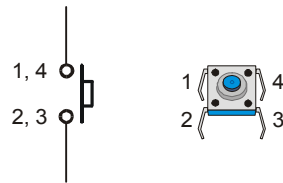


Figure 3-1
Normally Open
Pushbutton

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

The left side of Figure 3-2 shows how a normally open pushbutton looks when it's not pressed. When the button is not pressed, there is a gap between the 1,4 and 2,3 terminals. This gap makes it so that the 1,4 terminal can not conduct current to the 2,3 terminal. This is called an open circuit. The name "normally open" means that the pushbutton's normal state (not pressed) forms an open circuit. When the button is pressed, the gap between the 1,4 and 2,3 terminals is bridged by a conductive metal. This is called closed, and current can flow through the pushbutton.

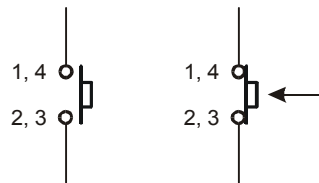


Figure 3-2
Normally Open
Pushbutton

*Not pressed (left) and
pressed (right)*

Test Parts for the Pushbutton

- (1) LED – pick a color
- (1) Resistor - 470 Ω (yellow-violet-brown)
- (1) Pushbutton - normally open
- (1) Jumper wire

Building the Pushbutton Test Circuit

Figure 3-3 shows a circuit you can build to manually test the pushbutton.



Always disconnect power from your Board of Education or BASIC Stamp HomeWork Board before making any changes to your test circuit. From here onward, the instructions will no longer say “Disconnect power...” between each circuit modification. It is up to you to remember to do this.

Always reconnect power to your Board of Education or BASIC Stamp HomeWork Board before downloading a program to the BASIC Stamp.

3

✓ Build the circuit shown in Figure 3-3.

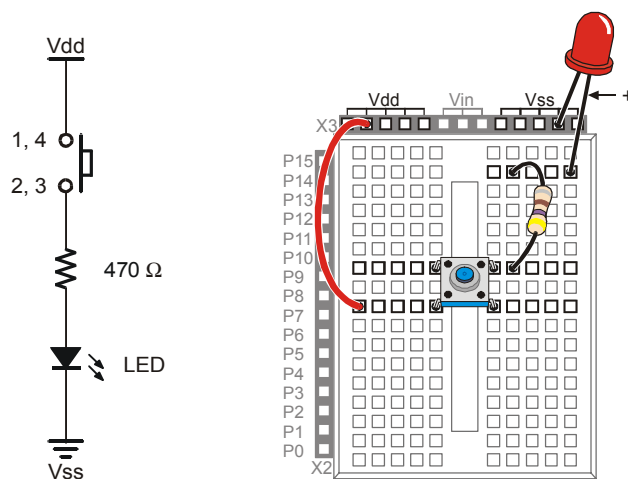


Figure 3-3
Pushbutton Test Circuit

Testing the Pushbutton

When the pushbutton is not pressed, the LED will be off. If the wiring is correct, when the pushbutton is pressed, the LED should be on (emitting light).



Warning signs: If the Pwr LED on the Board of Education flickers, goes dim, or goes out completely when you plug the power supply back in, it means you have a short circuit. If this happens to you, disconnect power immediately and find and correct the mistake in your circuit.

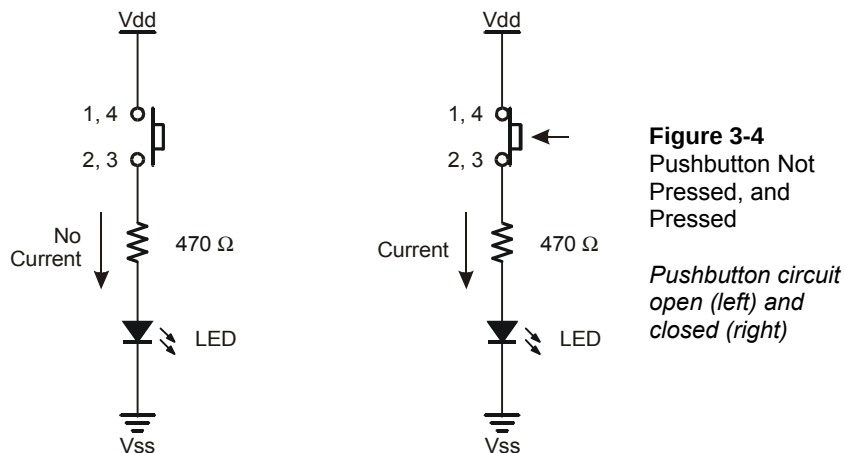
The Power LED on the HomeWork Board is different. It glows only while a program is running. If a program ends (using the **END** command), the Power LED will also turn off.

✓ Verify that the LED in your test circuit is off.

- ✓ Press and hold the pushbutton, and verify that the LED emits light while you are holding the pushbutton down.

How the Pushbutton Circuit Works

The left side of Figure 3-4 shows what happens when the pushbutton is not pressed. The LED circuit is not connected to Vdd. It is an open circuit that cannot conduct current. By pressing the pushbutton, you close the connection between the terminals with conductive metal making a pathway for electrons to flow through the circuit.



Your Turn – A Short-Circuit

Figure 3-5 shows a different circuit that will cause the LED to behave differently. When the button is not pressed, the LED stays on. When the button is pressed, the LED turns off. The reason the LED turns off when the pushbutton is pressed is because current always follows the path of least resistance. When the pushbutton is pressed, terminals 1,4 and 2,3, have almost no resistance between them, so all the current passes through the pushbutton (short circuit) instead of the LED.

- ✓ Build the circuit shown in Figure 3-5.
- ✓ Repeat the tests you performed on the first pushbutton circuit you built with this new circuit.

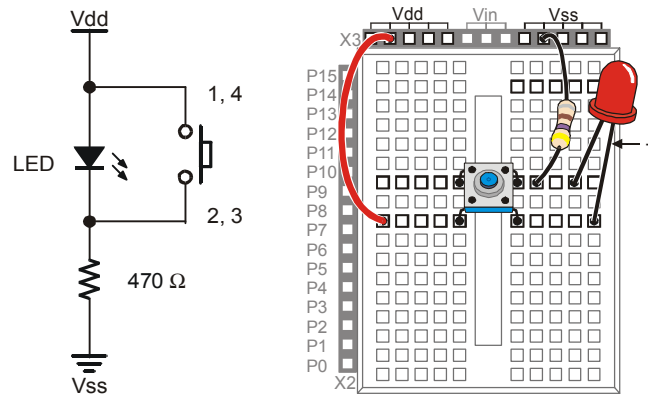


Figure 3-5
LED that Gets
Shorted by
Pushbutton



Can you really do that with the LED? Up until now, the LED's cathode has always been connected to Vss. Now, the LED is in a different place in the circuit, with its anode connected to Vdd. People often ask if this breaks any circuit rules, and the answer is no. The electrical pressure supplied by Vdd and Vss is 5 volts. The diode will always use 1.6 volts, and the resistor will always use 3.4 volts, regardless of their order.

ACTIVITY #2: READING A PUSHBUTTON WITH THE BASIC STAMP

In this activity, you will connect a pushbutton circuit to the BASIC Stamp and display whether or not the pushbutton is pressed. You will do this by writing a PBASIC program that checks the state of the pushbutton and displays it in the Debug Terminal.

Parts for a Pushbutton Circuit

- (1) Pushbutton - normally open
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Resistor – 10 k Ω (brown-black-orange)
- (2) Jumper wires

Building a Pushbutton Circuit for the BASIC Stamp

Figure 3-6 shows a pushbutton circuit that is connected to BASIC Stamp I/O pin P3.

✓ Build the circuit shown in Figure 3-6.

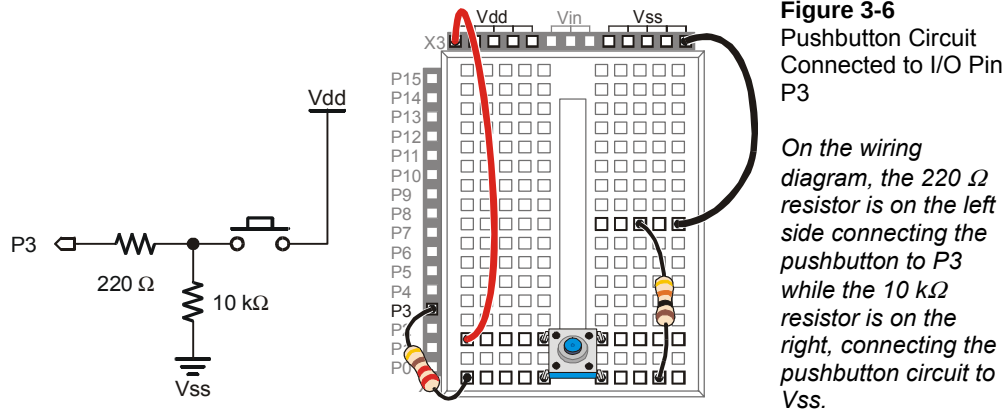


Figure 3-7 shows what the BASIC Stamp sees when the button is pressed, and when it's not pressed. When the pushbutton is pressed, the BASIC Stamp senses that Vdd is connected to P3. Inside the BASIC Stamp, this causes it to place the number 1 in a part of its memory that stores information about its I/O pins. When the pushbutton is not pressed, the BASIC Stamp cannot sense Vdd, but it can sense Vss through the 10 k Ω and 220 Ω resistors. This causes it to store the number 0 in that same memory location that stored a 1 when the pushbutton was pressed.

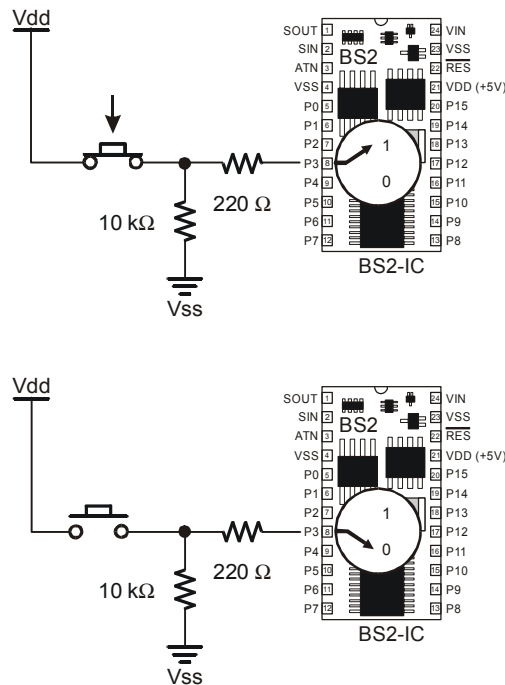


Figure 3-7
BASIC Stamp Reading
a Pushbutton

When the pushbutton is pressed, the BASIC Stamp reads a 1 (above). When the pushbutton is not pressed, the BASIC Stamp reads a 0 (below).



Binary and Circuits: The base-2 number system uses only the digits 1 and 0 to make numbers, and these binary values can be transmitted from one device to another. The BASIC Stamp interprets Vdd (5 V) as binary-1 and Vss (0 V) as binary-0. Likewise, when the BASIC Stamp sets an I/O pin to Vdd using **HIGH**, it sends a binary-1. When it sets an I/O pin to Vss using **LOW**, it sends a binary-0. This is a very common way of communicating binary numbers used by many computer chips and other devices.

Programming the BASIC Stamp to Monitor the Pushbutton

The BASIC Stamp stores the one or zero it senses at I/O pin P3 in a memory location called **IN3**. Here is an example program that shows how this works:

Example Program: ReadPushbuttonState.bs2

This program makes the BASIC Stamp check the pushbutton every $\frac{1}{4}$ second and send the value of **IN3** to the Debug Terminal. Figure 3-8 shows the Debug Terminal while the program is running. When the pushbutton is pressed, the Debug Terminal displays

the number 1, and when the pushbutton is not pressed, the Debug Terminal displays the number 0.

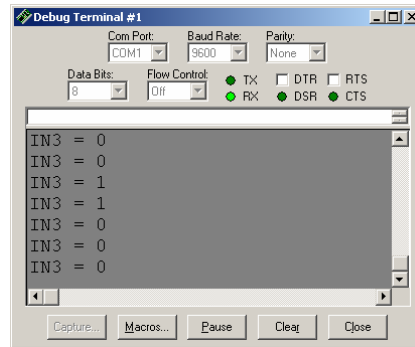


Figure 3-8
Debug Terminal
Displaying
Pushbutton States

The Debug Terminal displays 1 when the pushbutton is pressed and 0 when it is not pressed.

- ✓ Enter the ReadPushbuttonState.bs2 program into the BASIC Stamp Editor.
- ✓ Run the program.
- ✓ Verify that the Debug Terminal displays the value 0 when the pushbutton is not pressed.
- ✓ Verify that the Debug Terminal displays the value 1 when the pushbutton is pressed and held.

```
' What's a Microcontroller - ReadPushbuttonState.bs2
' Check and send pushbutton state to Debug Terminal every 1/4 second.

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

DO

    DEBUG ? IN3
    PAUSE 250

LOOP
```

How ReadPushbuttonState.bs2 Works

The **DO...LOOP** in the program repeats every $\frac{1}{4}$ second because of the command **PAUSE 250**. Each time through the **DO...LOOP**, the command **DEBUG ? IN3** sends the value of **IN3** to the Debug Terminal. The value of **IN3** is the state that I/O pin P3 senses at the instant the **DEBUG** command is executed.

Your Turn – A Pushbutton with a Pull-up Resistor

The circuit you just finished working with has a resistor connected to Vss. This resistor is called a pull-down resistor because it pulls the voltage at P3 down to Vss (0 volts) when the button is not pressed. Figure 3-9 shows a pushbutton circuit that uses a pull-up resistor. This resistor pulls the voltage up to Vdd (5 volts) when the button is not pressed. The rules are now reversed. When the button is not pressed, **IN3** stores the number 1, and when the button is pressed, **IN3** stores the number 0.



The 220 Ω resistor is used in the pushbutton example circuits to protect the BASIC Stamp I/O pin. Although it's a good practice for prototyping, in most products, this resistor is replaced with a wire (since wires cost less than resistors).

- ✓ Modify your circuit as shown in Figure 3-9.
- ✓ Re-run ReadPushbuttonState.bs2.
- ✓ Using the Debug Terminal, verify that **IN3** is 1 when the button is not pressed and 0 when the button is pressed.

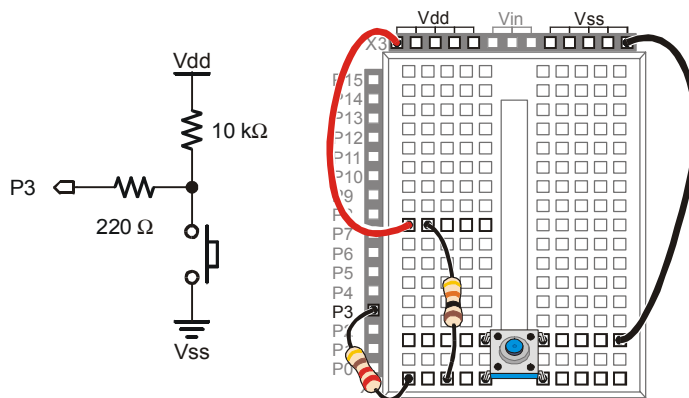


Figure 3-9
Modified Pushbutton
Circuit



Active-low vs. Active-high: This pushbutton Figure 3-9 is called active-low because it sends the BASIC Stamp a low signal (Vss) when the button is active (pressed). The pushbutton circuit in Figure 3-6 the main activity is active-high because it sends a high signal (Vdd) when the button is active (pressed).

ACTIVITY #3: PUSHBUTTON CONTROL OF AN LED CIRCUIT

Figure 3-10 shows a zoomed in view of a pushbutton and LED used to adjust the settings on a computer monitor. This is just one of many devices that have a pushbutton that you can press to adjust the device and an LED to show you the device's status.

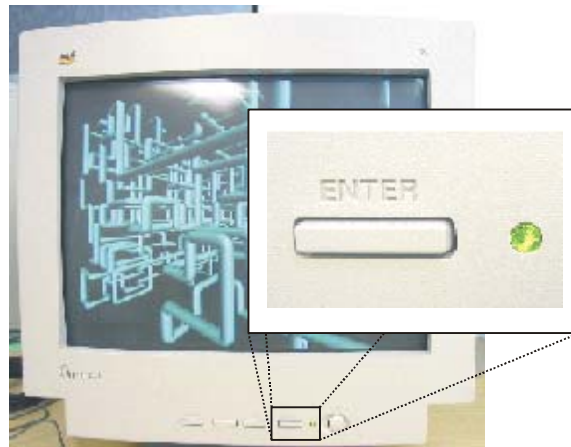


Figure 3-10
Button and LED on
a Computer Monitor

The BASIC Stamp can be programmed to make decisions based on what it senses. For example, it can be programmed to decide to flash the LED on/off ten times per second when the button is pressed.

Pushbutton and LED Circuit Parts

- (1) Pushbutton – normally open
- (1) Resistor - 10 k Ω (brown-black-orange)
- (1) LED – any color
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Resistor – 470 Ω (yellow-violet-brown)
- (2) Jumper wires

Building the Pushbutton and LED Circuits

Figure 3-11 shows the pushbutton circuit used in the activity you just finished along with the LED circuit used in Chapter #2, Activity #2.

✓ Build the circuit shown in Figure 3-11.

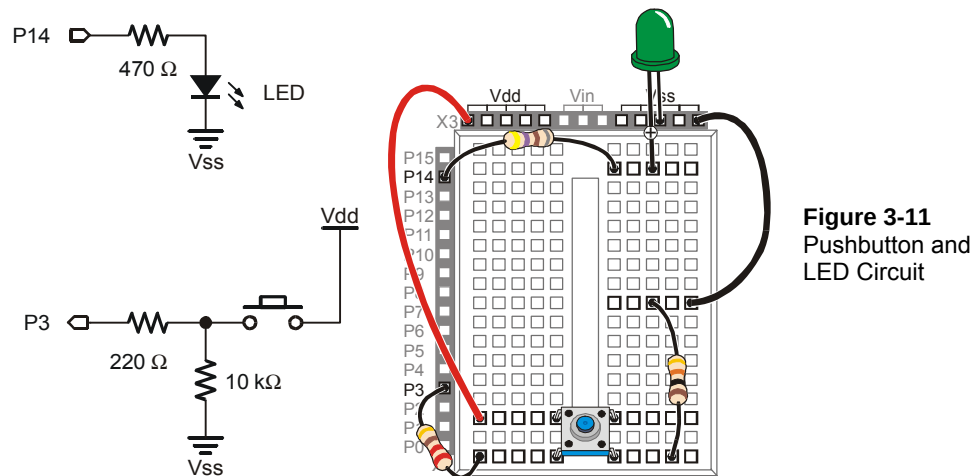


Figure 3-11
Pushbutton and
LED Circuit

Programming Pushbutton Control

The BASIC Stamp can be programmed to make decisions using an **IF...THEN...ELSE** statement. The example program you are about to run will flash the LED on and off when the pushbutton is pressed using an **IF...THEN...ELSE** statement. Each time through the **DO...LOOP**, the **IF...THEN...ELSE** statement checks the state of the pushbutton and decides whether or not to flash the LED.

Example Program: PushbuttonControlledLed.bs2

- ✓ Enter PushbuttonControlledLed.bs2 into the BASIC Stamp Editor and run it.
- ✓ Verify that the LED flashes on and off while the pushbutton is pressed and held down.
- ✓ Verify that the LED does not flash when the pushbutton is not pressed down.

```
' What's a Microcontroller - PushbuttonControlledLed.bs2
' Check pushbutton state 10 times per second and blink LED when pressed.

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

DO
```



```

DEBUG ? IN3

IF (IN3 = 1) THEN
  HIGH 14
  PAUSE 50
  LOW 14
  PAUSE 50

ELSE
  PAUSE 100

ENDIF

LOOP

```

How PushbuttonControlledLed.bs2 Works

This program is a modified version of ReadPushbuttonState.bs2 from the previous activity. The **DO...LOOP** and **DEBUG ? IN3** commands are the same. The **PAUSE 250** was replaced with an **IF...THEN...ELSE** statement. When the condition after the **IF** is true (**IN3 = 1**), the commands that come after the **THEN** statement are executed. They will be executed until the **ELSE** statement is reached, at which point the program skips to the **ENDIF** and moves on. When the condition after the **IF** is not true (**IN3 = 0**), the commands after the **ELSE** statement are executed until the **ENDIF** is reached.

You can make a detailed list of what a program should do, to either help you plan the program or to describe what it does. This kind of list is called pseudo code, and the example below uses pseudo code to describe how PushbuttonControlledLed.bs2 works.

- *Do the commands between here and the Loop statement over and over again*
 - *Display the value of IN3 in the Debug Terminal*
 - *If the value of IN3 is 1, Then*
 - *Turn the LED on*
 - *Wait for 1/20 of a second*
 - *Turn the LED off*
 - *Wait for 1/20 of a second*
 - *Else, (if the value of IN3 is 0)*
 - *do nothing, but wait for the same amount of time it would have taken to briefly flash the LED (1/10 of a second).*
- *Loop*

Your Turn – Faster/Slower

- ✓ Save the example program under a different name.
- ✓ Modify the program so that the LED flashes twice as fast when you press and hold the pushbutton.
- ✓ Modify the program so that the LED flashes half as fast when you press and hold the pushbutton.

3

ACTIVITY #4: TWO PUSHBUTTONS CONTROLLING TWO LED CIRCUITS

Let's add a second pushbutton into the project and see how it works. To make things a little more interesting, let's also add a second LED circuit and use the second pushbutton to control it.

Pushbutton and LED Circuit Parts

- (2) Pushbuttons – normally open
- (2) Resistors - 10 k Ω (brown-black-orange)
- (2) Resistors – 470 Ω (yellow-violet-brown)
- (2) Resistors – 220 Ω (red-red-brown)
- (2) LEDs – any color

Adding a Pushbutton and LED Circuit

Figure 3-12 shows a second LED and pushbutton circuit added to the circuit you tested in the previous activity.

- ✓ Build the circuit shown in Figure 3-12. If you need help building the circuit shown in the schematic, use the wiring diagram in Figure 3-13 as a guide.
- ✓ Modify ReadPushbuttonState.bs2 so that it reads **IN4** instead of **IN3**, and use it to test your second pushbutton circuit.

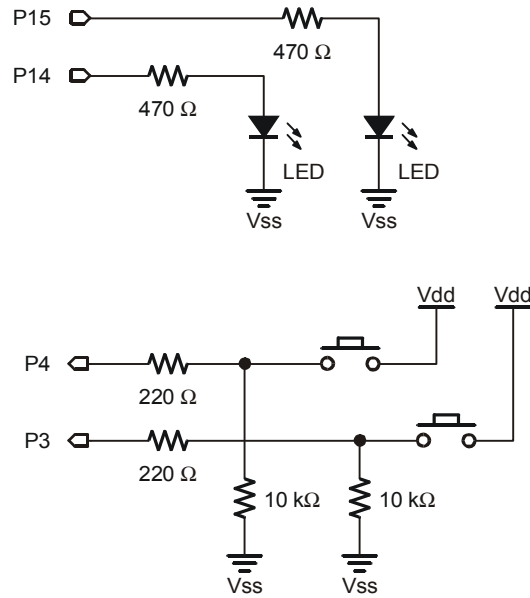


Figure 3-12
Schematic: Two
Pushbuttons and
LEDs



Connecting wires with dots: There are three places where wires intersect in Figure 3-12, but only two dots. Wires only connect if there is a dot at the intersection. The wire that connects the P4 pushbutton to the 10 k Ω resistor does not connect to the P3 pushbutton circuit because there is no dot.

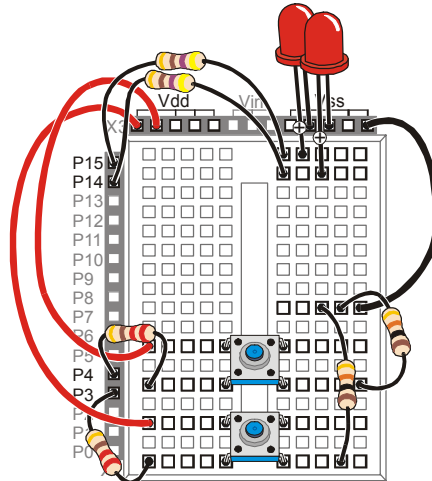


Figure 3-13
Wiring Diagram:
Two Pushbuttons
and LEDs

Programming Pushbutton Control

In the previous activity, you experimented with making decisions using an **IF...THEN...ELSE** statement. There is also such a thing as an **IF...ELSEIF...ELSE** statement. It works great for deciding which LED to flash on and off. The next example program shows how it works.

Example Program: PushbuttonControlOfTwoLeds.bs2

- ✓ Enter PushbuttonControlOfTwoLeds.bs2 into the BASIC Stamp Editor and run it.
- ✓ Verify that the LED in the circuit connected to P14 flashes on and off while the pushbutton in the circuit connected to P3 is held down.
- ✓ Also check to make sure the LED in the circuit connected to P15 flashes while the pushbutton in the circuit connected to P4 is held down

```
' What's a Microcontroller - PushbuttonControlOfTwoLeds.bs2
' Blink P14 LED if P3 pushbutton is pressed, and blink P15 LED if
' P4 pushbutton is pressed.
```

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

```
DO
```

```

DEBUG HOME
DEBUG ? IN4
DEBUG ? IN3

IF (IN3 = 1) THEN
  HIGH 14
  PAUSE 50

ELSEIF (IN4 = 1) THEN
  HIGH 15
  PAUSE 50

ELSE
  PAUSE 50

ENDIF

LOW 14
LOW 15

PAUSE 50

LOOP

```

How PushbuttonControlOfTwoLeds.bs2 Works

If the display of **IN3** and **IN4** scrolled down the Debug Terminal as they did in the previous example, it would be difficult to read. One way to fix this is to always send the cursor to the top-left position in the Debug Terminal using the **HOME** formatter:

```
DEBUG HOME
```

By sending the cursor to the home position each time through the **DO...LOOP**, the commands:

```

DEBUG ? IN4
DEBUG ? IN3

```

display the values of **IN4** and **IN3** in the same part of the Debug Terminal each time.

The **DO** keyword begins the loop in this program:

```
DO
```

These commands in the **IF** statement are the same as the ones in the example program from the previous activity:

```

IF (IN3 = 1) THEN
  HIGH 14

```

```
PAUSE 50
```

This is where the **ELSEIF** keyword helps. If **IN3** is not 1, but **IN4** is 1, we want to turn the LED connected to P15 on instead of the one connected to P14.

```
ELSEIF (IN4 = 1) THEN
  HIGH 15
  PAUSE 50
```

If neither statement is true, we still want to pause for 50 ms without changing the state of any LED circuits.

```
ELSE
  PAUSE 50
```

When you're finished with all the decisions, don't forget the **ENDIF**.

```
ENDIF
```

It's time to turn the LEDs off and pause again. You could try to decide which LED you turned on and turn it back off. PBASIC commands execute pretty quickly, so why not just turn them both off and forget about more decision making?

```
LOW 14
LOW 15

PAUSE 50
```

The **LOOP** statement sends the program back up to the **DO** statement, and the process of checking the pushbuttons and changing the states of the LEDs starts all over again.

```
LOOP
```

Your Turn – What about Pressing Both Pushbuttons?

The example program has a flaw. Try pressing both pushbuttons at once, and you'll see the flaw. You would expect both LEDs to flash on and off, but they don't because only one code block in an **IF...ELSEIF...ELSE** statement gets executed before it skips to the **ENDIF**.

Here is how you can fix this problem:

- ✓ Save PushbuttonControlOfTwoLeds.bs2 under a new name.
- ✓ Replace this **IF** statement and code block:

```
IF (IN3 = 1) THEN
```

```
HIGH 14
PAUSE 50
```

with this **IF...ELSEIF** statement:

```
IF (IN3 = 1) AND (IN4 = 1) THEN
  HIGH 14
  HIGH 15
  PAUSE 50

ELSEIF (IN3 = 1) THEN
  HIGH 14
  PAUSE 50
```



A **code block** is a group of commands. The **IF** statement above has a code block with three commands (**HIGH**, **HIGH**, and **PAUSE**). The **ELSEIF** statement has a code block with two commands (**HIGH**, **PAUSE**).

- ✓ Run your modified program and see if it handles both pushbutton and LED circuits as you would expect.



The **AND** keyword can be used in an **IF...THEN** statement to check if more than one condition is true. All conditions with **AND** have to be true for the **IF** statement to be true.

The **OR** keyword can also be used to check if at least one of the conditions are true.

You can also modify the program so that the LED that's flashing stays on for different amounts of time. For example, you can reduce the *Duration* of the **PAUSE** for both pushbuttons to 10, increase the **PAUSE** for the P14 LED to 100, and increase the **PAUSE** for the P15 LED to 200.

- ✓ Modify the **PAUSE** commands in the **IF** and the two **ELSEIF** statements as discussed.
- ✓ Run the modified program.
- ✓ Observe the difference in the behavior of each light.

ACTIVITY #5: REACTION TIMER TEST

You are the embedded systems engineer at a video game company. The marketing department recommends that a circuit to test the player's reaction time be added to the

next hand held game controller. Your next task is to develop a proof of concept for the reaction timer test.

The solution you will build and test in this activity is an example of how to solve this problem, but it's definitely not the only solution. Before continuing, take a moment to think about how you would design this reaction timer.

3

Reaction Timer Game Parts

- (1) LED – bi-color
- (1) Resistor – 470 Ω (yellow-violet-brown)
- (1) Pushbutton – normally open
- (1) Resistor – 10 k Ω (brown-black-orange)
- (1) Resistor – 220 Ω (red-red-brown)
- (2) Jumper wires

Building the Reaction Timer Circuit

Figure 3-14 shows a schematic and wiring diagram for a circuit that can be used with the BASIC Stamp to make a reaction timer game.

- √ Build the circuit shown in Figure 3-14.
- √ Run TestBiColorLED.bs2 from Chapter #2, Activity #5 to test the bi-color LED circuit and make sure your wiring is correct.
- √ If you re-built the pushbutton circuit for this activity, run ReadPushbuttonState.bs2 from Activity #2 in this chapter to make sure your pushbutton is working properly.

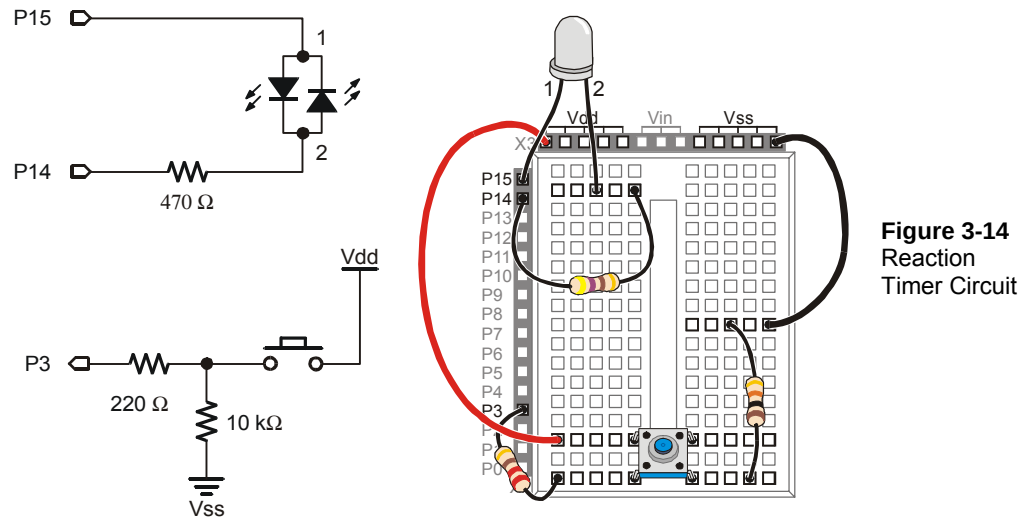


Figure 3-14
Reaction
Timer Circuit

Programming the Reaction Timer

This next example program will leave the bi-color LED off until the game player presses and holds the pushbutton. When the pushbutton is held down, the LED will turn red for a short period of time. When it turns green, the player has to let go of the pushbutton as fast as he or she can. The time between when the LED turns green and when the pushbutton is tracked by the program is used as a measure of reaction time.

The example program also demonstrates how polling and counting work. Polling is the process of checking something over and over again very quickly to see if it has changed. Counting is the process of adding a number to a variable each time something does (or does not) happen. In this program, the BASIC Stamp will poll from the time the bi-color LED turns green until the pushbutton is released. It will wait 1/1000 of a second by using the command **PAUSE 1**. Each time it polls and the pushbutton is not yet released, it will add 1 to the counting variable named **timeCounter**. When the pushbutton is released, the program stops polling and sends a message to the Debug Terminal that displays the value of the **timeCounter** variable.

Example Program: ReactionTimer.bs2

- ✓ Enter and run ReactionTimer.bs2.
- ✓ Follow the prompts on the Debug Terminal (see Figure 3-15).

The inner **DO...LOOP** deserves a closer look. A **DO...LOOP** can use a condition to decide whether or not to break out of the loop and move on to more commands that come afterwards. This **DO...LOOP** will repeat itself as long as the button is not pressed (**IN3 = 0**). The **DO...LOOP** will execute over and over again, until **IN3 = 1**. Then, the program moves on to the next command after the **LOOP UNTIL** statement. This is an example of polling. The **DO...LOOP UNTIL** polls until the pushbutton is pressed.

```
DO                                ' Nested loop repeats...
LOOP UNTIL IN3 = 1                ' until pushbutton press.
```

The commands that come immediately after the **LOOP UNTIL** statement turn the bi-color LED red, delay for one second, then turn it green.

```
LOW 14                            ' Bi-color LED red.
HIGH 15

PAUSE 1000                        ' Delay 1 second.

HIGH 14                           ' Bi-color LED green.
LOW 15
```

As soon as the bi-color LED turns green, it's time to start counting to track how long until the player releases the button. The **timeCounter** variable is set to zero, then another **DO...LOOP** with an **UNTIL** condition starts repeating itself. It repeats itself until the player releases the button (**IN3 = 0**). Each time through the loop, the BASIC Stamp delays for 1 ms using **PAUSE 1**, and it also adds 1 to the value of the **timeCounter** variable.

```
timeCounter = 0                    ' Set timeCounter to zero.

DO                                ' Nested loop, count time...

    PAUSE 1
    timeCounter = timeCounter + 1

LOOP UNTIL IN3 = 0                ' until pushbutton is released.
```

The bi-color LED is turned off.

```
LOW 14
```

The results are displayed in the Debug Terminal.

```
DEBUG "Your time was ", DEC timeCounter,
      " ms.", CR, CR,
      "To play again, hold the ", CR,
```

```
"button down again.", CR, CR
```

The last statement in the program is `LOOP`, which sends the program back to the very first `DO` statement.

Your Turn – Revising the Design

The marketing department gave your prototype to some game testers. When the game testers were done, the marketing department came back to you with an itemized list of three problems that have to be fixed before your prototype can be built into the game controller.

- √ Save `ReactionTimer.bs2` under a new name (like `ReactionTimerYourTurn.bs2`).

The “itemized list” of problems and their solutions are discussed below.

Item-1

When a player holds the button for 30 seconds, his score is actually 14000 ms, a measurement of 14 seconds. This has to be fixed!

It turns out that executing the loop itself along with adding one to the `timeCounter` variable takes about 1 ms without the `PAUSE 1` command. This is called code overhead, and it’s the amount of time it takes for the BASIC Stamp to execute the commands. A quick fix that will improve the accuracy is to simply comment out the `PAUSE 1` command by deleting it or adding an apostrophe to the left of it.

```
' PAUSE 1
```

- √ Try commenting `PAUSE 1` and test to see how accurate the program is.

Instead of commenting the delay, another way you can fix the program is to multiply your result by two. For example, just before the `DEBUG` command that displays the number of ms, you can insert a command that multiplies the result by two:

```
timeCounter = timeCounter * 2           ' <- Add this
DEBUG "Your time was ", DEC timeCounter, " ms.", CR, CR
```

- √ Uncomment the `PAUSE` command by deleting the apostrophe, and try the multiply by two solution instead.



For precision, you can use the `*/` operator to multiply by a value with a fraction to make your answer more precise. The `*/` operator is not hard to use; here's how:

- 1) Place the value or variable you want to multiply by a fractional value before the `*/` operator.
- 2) Take the fractional value that you want to use and multiply it by 256.
- 3) Round off to get rid of anything to the right of the decimal point.
- 4) Place that value after the `*/` operator.

Example: Let's say you want to multiply the `timeCounter` variable by 3.69.

- 1) Start by placing `timeCounter` to the left of the `*/` operator:

```
timeCounter = timeCounter */
```

- 2) Multiply your fractional value by 256: $3.69 \times 256 = 944.64$.
- 3) Round off: $944.64 \approx 945$.
- 4) Place that value to the right of the `*/` operator:

```
timeCounter = timeCounter */ 945
```

Item-2

Players soon figure out that the delay from red to green is 1 second. After playing it several times, they get better at predicting when to let go, and their score no longer reflects their true reaction time.

The BASIC Stamp has a **RANDOM** command. Here is how to modify your code for a random number:

- ✓ At the beginning of your code, add a declaration for a new variable called **value**, and set it to 23. The value 23 is called the seed because it starts the pseudo random number sequence.

```
timeCounter VAR Word
value VAR Byte           ' <- Add this
value = 23                ' <- Add this
```

- ✓ Just before the pause command, use the **RANDOM** command to give **value** a new “random” value from the pseudo random sequence that started with 23.

```
RANDOM value              ' <- Add this.
DEBUG "Delay time ", ? 1000 + value, CR ' <- Add this.
```

- ✓ Modify the **PAUSE** command so that the random value is added to 1000 (for one second) in the **PAUSE** command's *Duration* argument.

`PAUSE 1000 + value`

' <- Modify this.



What's an algorithm? An algorithm is a sequence of mathematical operations.

What's pseudo random? Pseudo random means that it seems random, but it isn't really. Each time you start the program over again, you will get the same sequence of values.

What's a seed? A seed is a value that is used to start the pseudo random sequence. If you use a different value (change value from 23 to some other number), you will get a different pseudo random sequence.

Item-3

A player that lets go of the button before the light turns green gets an unreasonably good score (1 ms). Your microcontroller needs to figure out if a player is cheating.

Pseudo code was introduced near the end of Activity #3 in this chapter. Here is some pseudo code to help you apply an **IF...THEN...ELSE** statement to solve the problem.

- *If the value of timeCounter equals 1*
 - *Display a message telling the player he or she has to wait until after the light turns green to let go of the button.*
 - *Else, (if the value of timeCounter is greater than 1)*
 - *Display the value of timeCounter (just like in ReactionTimer.bs2) time in ms.*
- ✓ Modify your program by implementing this pseudo code in PBASIC to fix the cheating player problem.

SUMMARY

This chapter introduced the pushbutton and common pushbutton circuits. This chapter also introduced how to build and test a pushbutton circuit and how to use the BASIC Stamp to read the state of one or more pushbuttons. The BASIC Stamp was programmed to make decisions based on the state(s) of the pushbutton(s) and this information was used to control LED(s). A reaction timer game was built using these concepts. In addition to controlling LEDs, the BASIC Stamp was programmed to poll a pushbutton and take time measurements.

Reading individual pushbutton circuits using the special I/O variables built into the BASIC Stamp (**IN3**, **IN4**, etc.) was introduced. Making decisions based on these values using **IF...THEN...ELSE** statements, **IF...ELSEIF...ELSE** statements, and code blocks were also introduced. For evaluating more than one condition, the **AND** and **OR** operators were introduced. Adding a condition to a **DO...LOOP** using the **UNTIL** keyword was introduced along with nesting **DO...LOOP** code blocks.

Questions

1. What is the difference between sending and receiving **HIGH** and **LOW** signals using the BASIC Stamp?
2. What does “normally open” mean in regards to a pushbutton?
3. What happens between the terminals of a normally open pushbutton when you press it?
4. What is the value of **IN3** when a pushbutton connects it to Vdd? What is the value of **IN3** when a pushbutton connects it to Vss?
5. What does the command **DEBUG ? IN3** do?
6. What kind of code blocks can be used for making decisions based on the value of one or more pushbuttons?
7. What does the **HOME** formatter do in the statement **DEBUG HOME**?

Exercises

1. Explain how to modify ReadPushbuttonState.bs2 on page 77 so that it reads the pushbutton every second instead of every $\frac{1}{4}$ second.
2. Explain how to modify ReadPushbuttonState.bs2 so that it reads a normally open pushbutton circuit with a pull-up resistor connected to I/O pin P6.

Project

1. Modify ReactionTimer.bs2 so that it is a two player game. Add a second button wired to P4 for the second player.

Solutions

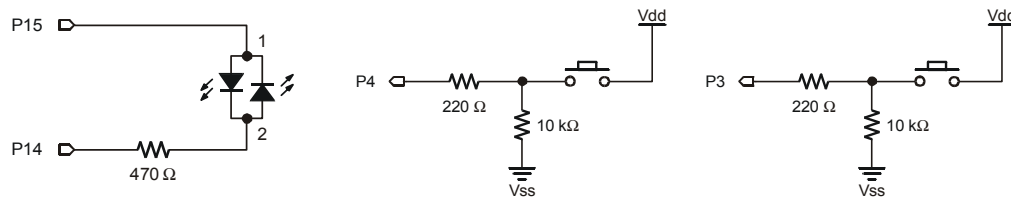
- Q1. Sending uses the BASIC Stamp I/O pin as an output, whereas receiving uses the I/O pin as an input.
- Q2. Normally open means the pushbutton's normal state (not pressed) forms an open circuit.
- Q3. When pressed, the gap between the terminals is bridged by a conductive metal. Current can then flow through the pushbutton.
- Q4. **IN3** = 1 when pushbutton connects it to Vdd. **IN3** = 0 when pushbutton connects it to Vss.
- Q5. **DEBUG ? IN3** sends the value of **IN3** to the Debug Terminal.
- Q6. **IF...THEN...ELSE** and **IF...ELSEIF...ELSE**.
- Q7. The **HOME** formatter sends the cursor to the top left position in the Debug Terminal.
- E1. The **DO...LOOP** in the program repeats every $\frac{1}{4}$ second because of the command **PAUSE 250**. To repeat every second, change the **PAUSE 250** ($250\text{ms} = 0.25\text{ s} = \frac{1}{4}\text{ s}$), to **PAUSE 1000** ($1000\text{ms} = 1\text{ s}$).

```
DO
  DEBUG ? IN3
  PAUSE 1000
LOOP
```

- E2. Replace **IN3** with **IN6**, to read I/O pin P6. The program only displays the pushbutton state, and does not use the value to make decisions, it does not matter whether the resistor is a pull-up or a pull-down. The **DEBUG** statement will display the button state either way.

```
DO
  DEBUG ? IN6
  PAUSE 250
LOOP
```

- P1. First, a button was added for the second player, wired to Stamp I/O pin P4. The schematic is based on Figure 3-14 on page 90.



3

Snippets from the solution program are included below, but keep in mind solutions may be coded a variety of different ways. However, most solutions will include the following modifications:

Use two variables to keep track of two player's times.

```
timeCounterA VAR Word ' Time score of player A
timeCounterB VAR Word ' Time score of player B
```

Change instructions to reflect two pushbuttons:

```
DEBUG "Press and hold pushbuttons", CR,
DEBUG "buttons down again.", CR, CR
```

Wait for both buttons to be pressed before turning LED red, by using the **AND** operator.

```
LOOP UNTIL (IN3 = 1) AND (IN4 = 1)
```

Wait for both buttons to be released to end timing, again using the **AND** operator.

```
LOOP UNTIL (IN3 = 0) AND (IN4 = 0)
```

Add logic to decide which player's time is incremented.

```
IF (IN3 = 1) THEN
timeCounterA = timeCounterA + 1
ENDIF
IF (IN4 = 1) THEN
timeCounterB = timeCounterB + 1
ENDIF
```

Change time display to show times of both players.

```
DEBUG "Player A Time: ", DEC timeCounterA, " ms. ", CR
DEBUG "Player B Time: ", DEC timeCounterB, " ms. ", CR, CR
```

Add logic to show which player had the faster reaction time.

```
IF (timeCounterA < timeCounterB) THEN
```

```

DEBUG "Player A is the winner!", CR
ELSEIF (timeCounterB < timeCounterA) THEN
DEBUG "Player B is the winner!", CR
ELSE
DEBUG "It's a tie!", CR
ENDIF

```

The complete solution is shown below.

```

' What's a Microcontroller - Ch03Prj03_TwoPlayerReactionTimer.bs2
' Test reaction time with a pushbutton and a bi-color LED.
' Add a second player with a second pushbutton. Both players
' play at once using the same LED. Quickest to release wins.
' Pin P3: Player A Pushbutton, Active High
' Pin P4: Player B Pushbutton, Active High

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

timeCounterA VAR          Word          ' Time score of player A
timeCounterB VAR          Word          ' Time score of player B

DEBUG "Press and hold pushbuttons", CR, ' Display reaction
      "to make light turn red.", CR, CR, ' instructions.
      "When light turns green, let", CR,
      "go as fast as you can.", CR, CR

DO                                     ' Begin main loop.

  DO                                  ' Loop until both press
  ' Nothing
  LOOP UNTIL (IN3 = 1) AND (IN4 = 1)

  LOW 14                              ' Bi-color LED red.
  HIGH 15

  PAUSE 1000                          ' Delay 1 second.

  HIGH 14                              ' Bi-color LED green.
  LOW 15

  timeCounterA = 0                    ' Set timeCounters to zero
  timeCounterB = 0

DO

  PAUSE 1
  IF (IN3 = 1) THEN                   ' If button is still down,

```

```

timeCounterA = timeCounterA + 1      ' increment counter
ENDIF
IF (IN4 = 1) THEN
    timeCounterB = timeCounterB + 1
ENDIF

LOOP UNTIL (IN3 = 0) AND (IN4 = 0)    ' Loop until both buttons
                                      ' released.

LOW 14                                ' Bi-color LED off.

DEBUG "Player A Time: ", DEC timeCounterA, " ms. ", CR
DEBUG "Player B Time: ", DEC timeCounterB, " ms. ", CR, CR
IF (timeCounterA < timeCounterB) THEN
    DEBUG "Player A is the winner!", CR
ELSEIF (timeCounterB < timeCounterA) THEN
    DEBUG "Player B is the winner!", CR
ELSE
    ' A & B times are equal
    DEBUG "It's a tie!", CR
ENDIF
DEBUG CR

DEBUG "To play again, hold the ", CR    ' Play again instructions.
DEBUG "buttons down again.", CR, CR

LOOP                                    ' Back to "Begin main

```

Further Investigation

The resources listed here are available for free download from the Parallax web site and are also included on the Parallax CD.

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

The *BASIC Stamp Manual* has more examples you can try and information that further explains the following: The **DEBUG HOME** and **CLS** formatters, input pin variables such **IN3**, **IN4**, and the **RANDOM** command.

“Basic Analog and Digital”, Student Guide, Version 2.0, Parallax Inc., 2003

Basic Analog and Digital explains binary counting using pushbuttons. It also uses pushbuttons to introduce a technique for transmitting numbers from one system to another called synchronous serial communication.

“BASIC Stamp Editor Help File”, PBASIC 2.5 Version 2.0 Parallax Inc., 2003

The PBASIC 2.5 Help File has information on the **WHILE** and **UNTIL** conditions used with **DO...LOOP**, and information on nesting and **IF...THEN...ELSE** code blocks, which is new to PBASIC 2.5. You can find this information by clicking the book icon on your BASIC Stamp Editor task bar, then selecting PBASIC Reference from the menu in the left sidebar window. This will open the PBASIC Command Reference alphabetical listing in the main window. Detailed information can be found by clicking on each command.

Chapter #4: Controlling Motion

4

MICROCONTROLLED MOTION

Microcontrollers make sure things move to the right place all around you every day. If you have an inkjet printer, the print head that goes back and forth across the page as it prints is moved by a stepper motor that is controlled by a microcontroller. The automatic grocery store doors that you walk through are controlled by microcontrollers, and the automatic eject feature in your VCR or DVD player is also controlled by a microcontroller.

ON/OFF SIGNALS AND MOTOR MOTION

Just about all microcontrolled motors receive sequences of high and low signals similar to the ones you've been sending to LEDs. The difference is that the microcontroller has to send these signals at rates that are usually faster than the eye can detect. The timing and number of separate high/low signals differ from one motor to the next, but they can all be controlled by microcontrollers capable of delivering the high/low signals.

Some of these motors require lots of circuitry to help the microcontroller make them work. Other motors require extra mechanical parts to make them work right in machines. Of all the different types of motors to start with, the hobby servo that you will experiment with in this chapter is probably the simplest. As you will soon see, it is easy to control with the BASIC Stamp, requires little or no additional circuitry, and has a mechanical output that is easy to connect to things to make them move.

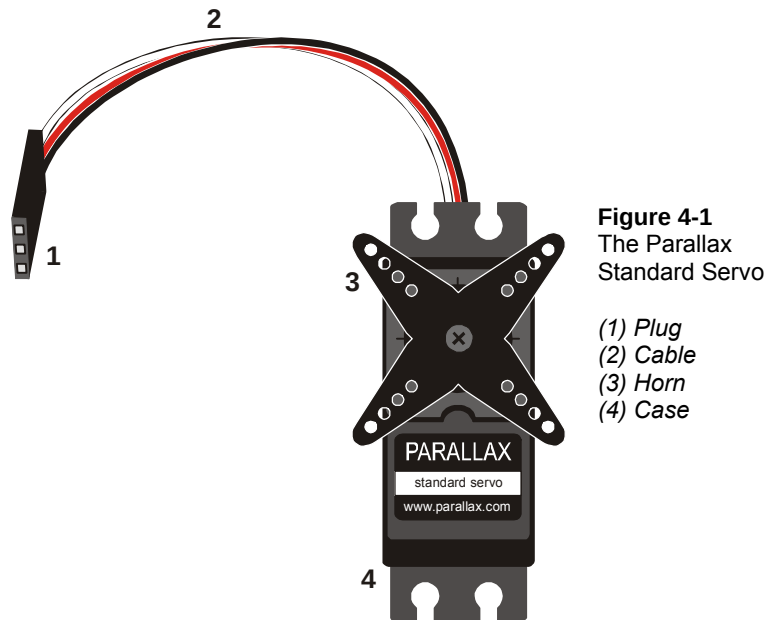
ACTIVITY #1: CONNECTING AND TESTING THE SERVO

In this activity, you will connect a servo to a power supply and the BASIC Stamp. You will then verify that the servo is functioning properly by programming the BASIC Stamp to send signals to the servo that will control the servo's position.

Introducing the Servo

Figure 4-1 shows a drawing of a Parallax Standard Servo. The plug (1) is used to connect the servo to a power source (Vdd and Vss) and a signal source (a BASIC Stamp I/O pin). The cable (2) conducts Vdd, Vss and the signal line from the plug into the servo. The horn (3) is the part of the servo that looks like a four-pointed star. When the servo is running, the horn is the moving part that the BASIC Stamp controls. The case (4)

contains the servo's control circuits, a DC motor, and gears. These parts work together to take high/low signals from the BASIC Stamp and convert them into positions held by the servo horn.



CAUTION: use only a Parallax Standard Servo for the activities in this text!

Do not substitute a Parallax Continuous Rotation Servo, as it may be quickly damaged by some of the circuits shown below. Likewise, we do not recommend using other brands or models of hobby servos, which may not be rated for use with the voltage in these circuits.

Servo and LED Circuit Parts

An LED circuit can be used to monitor the control signal the BASIC Stamp sends to the servo. Keep in mind that the LED circuit is not required to help the servo operate. It is just there to help see what's going on with the control signals.

(1) Parallax Standard Servo

- (1) Resistor – 470 Ω (yellow-violet-brown)
- (1) LED – any color

Building the Servo and LED Circuits

It's really important to be careful when connecting a servo to your BASIC Stamp. How you connect your servo depends on whether you are using the Board of Education Rev B, Rev C, or the HomeWork Board. If you are using the Board of Education, but you're not sure which Rev it is, Figure 4-2 shows an example of the Rev label on the Board of Education Rev B.

4

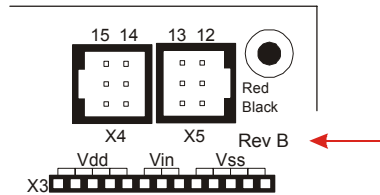


Figure 4-2
Board of
Education
Rev Label

- ✓ Examine the labeling on your carrier board and figure out whether you have a HomeWork Board or a Board of Education Rev B, or Rev C.
- ✓ Skip to instructions for connecting the servo to the BASIC Stamp on your carrier board:
 - Page 105 – Board of Education Rev C
 - Page 108 – BASIC Stamp HomeWork Board
 - Page 111 – Board of Education Rev B

Board of Education Rev C

Figure 4-3 shows the schematic of the circuit you will build on the Board of Education Rev C.

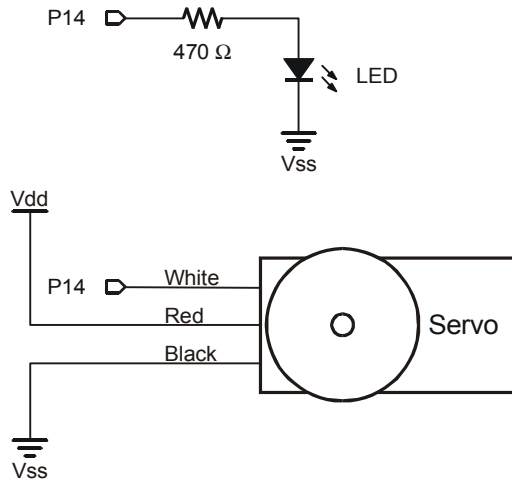


Figure 4-3
Servo and LED
Indicator
Schematic for
Board of Education
Rev C

- ✓ Turn off the power as shown in Figure 4-4.

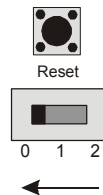


Figure 4-4
Disconnect
Power

Figure 4-5 shows the servo header on the Board of Education Rev C. This is where you will plug in your servo. This board features a jumper that you can use to connect the servo's power supply to either V_{in} or V_{dd} . The jumper is the removable black rectangular piece right between the two servo headers.

- ✓ Set the jumper to V_{dd} as shown in Figure 4-5. This involves lifting the rectangular jumper up off of the pins it is currently on, and replacing it on the two pins closest to the V_{dd} label.

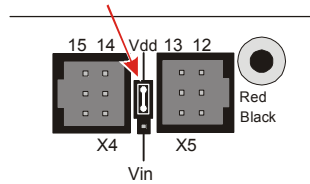


Figure 4-5
Servo
Header
Jumper
Set to Vdd

4



The jumper allows you to choose the power supply (Vin or Vdd) for the Parallax Standard Servo. If you are using a 9 V battery, it is best to set it to Vdd. Either setting will work if you are using a 6 V battery pack. **Use only Vdd** if you are using a DC supply that plugs into an AC outlet. Before using a DC supply, make sure to check the specifications for acceptable DC supplies listed in Appendix D: Batteries and Power Supplies.

- ✓ Build the circuit shown in Figure 4-6.
- ✓ Make sure you did not plug the servo in upside-down. The white, red and black wires should line up as shown in the figure.

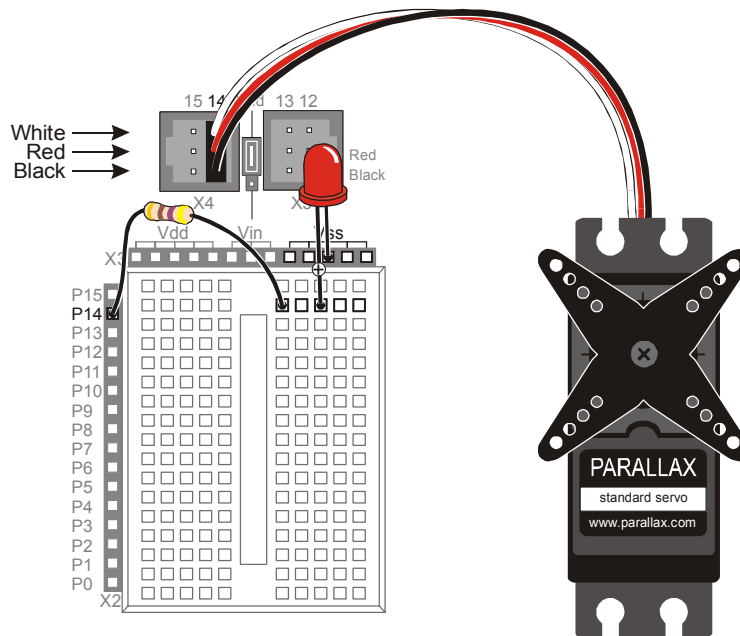


Figure 4-6
Servo and LED
Indicator on Board
of Education Rev C

Up until now, you have been using the 3-position switch in the 1 position. Now, you will move it to the 2 position to turn on the power to the servo header.

- ✓ Supply power to the servo header by adjusting the 3-position switch as shown in Figure 4-7. Your servo may move a bit when you connect the power.

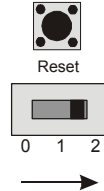


Figure 4-7
Power turned on to Board of Education and Servo Header

- ✓ Move on to Programming Servo Control on page 113.

BASIC Stamp HomeWork Board

If you are connecting your servo to a HomeWork Board, you will need these extra parts:

- (1) 3-pin male/male header (shown in Figure 4-8).
- (4) Jumper wires



Figure 4-8
HomeWork Board or Board of Education – Extra Parts

(1) 3-pin male/male header (top)

Figure 4-9 shows the schematic of the servo and LED circuits on the HomeWork Board. The instructions that come after this figure will show you how to safely build this circuit.



WARNING

Use only a 9 V battery when your Parallax Standard Servo is connected to the BASIC Stamp HomeWork Board. Do not use any kind of DC supply or “battery replacer” that plugs into an AC outlet. Improper use of these devices can cause the activity not to work, or even permanently damage the servo.

For best results, make sure your battery is new. If you are using a rechargeable battery, make sure it is freshly recharged. It should also be rated for 100 mAh (milliamp hours) or more. See Appendix D: Batteries and Power Supplies for more information.

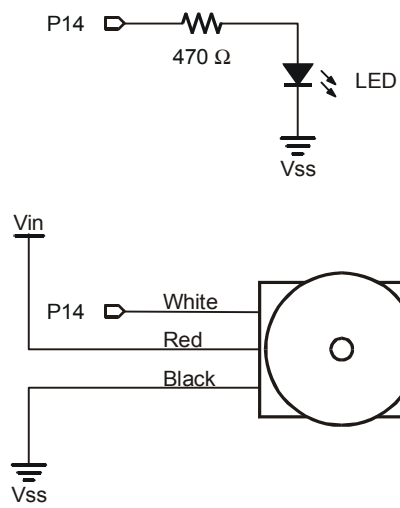


Figure 4-9
Schematic for Servo and LED Indicator on
HomeWork Board

4

- ✓ Disconnect your 9 V battery from your HomeWork Board.
- ✓ Build the LED indicator and servo header circuit shown in Figure 4-10.

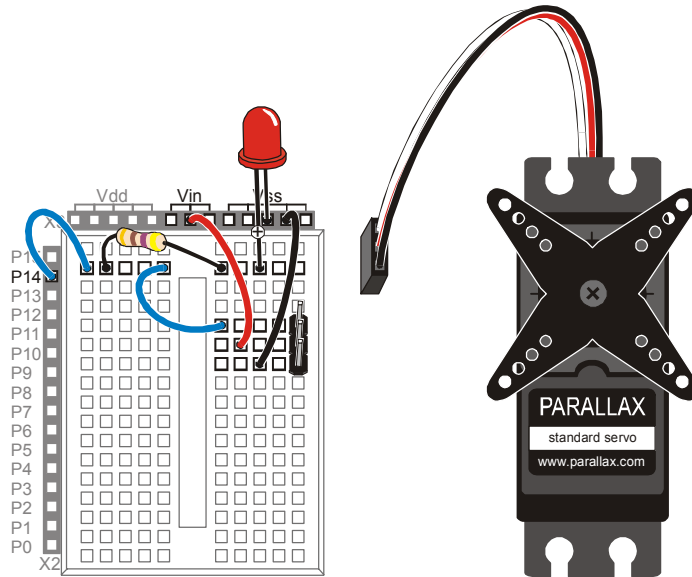


Figure 4-10
LED Indicator
and Servo
Header Circuits
on HomeWork
Board

- ✓ Connect the servo to the servo header as shown in Figure 4-11.
- ✓ Make sure that the colors on the servo's cable align properly with the colors labeled in the picture.
- ✓ Double check your wiring.
- ✓ Reconnect your 9 V battery to your HomeWork Board. The servo may move a bit when you make the connection.

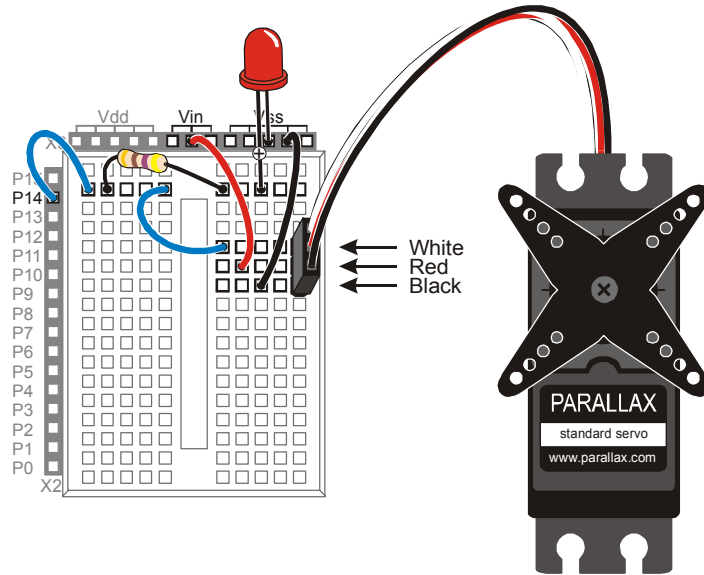


Figure 4-11
Connecting Servo to
Servo Header on a
HomeWork Board

4

✓ Move on to **Programming Servo Control** on page 113.

Board of Education Rev B

Figure 4-12 shows the schematic for the servo and LED circuits on the Board of Education Rev B. The instructions that come after this figure will show you how to safely build this circuit on your Board of Education Rev B.



WARNING

Use only a 9 V battery when your Parallax Standard Servo is connected to the Board of Education Rev B. Do not use any kind of DC supply or “battery replacer” that plugs into an AC outlet. Improper use of these devices can cause the activity not to work, or even permanently damage the servo.

For best results, make sure your battery is new. If you are using a rechargeable battery, make sure it is freshly recharged. It should also be rated for 100 mAh (milliamp hours) or more. See Appendix D: Batteries and Power Supplies for more information.

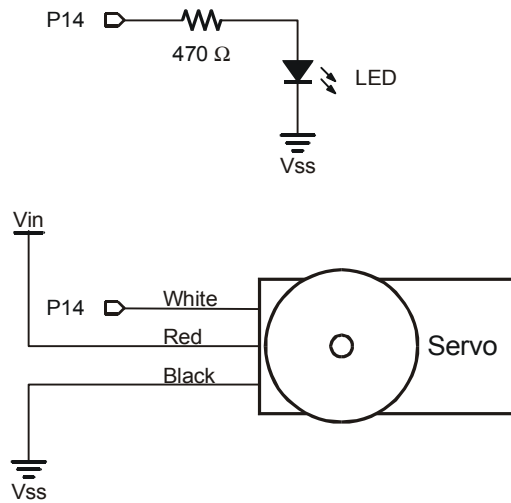


Figure 4-12
Schematic for Servo
and LED Indicator on
Board of Education
Rev B

- ✓ Disconnect your battery or any other power supply from your board.
- ✓ Build the LED circuit shown in Figure 4-12.
- ✓ Connect the servo to the servo header as shown in Figure 4-13.

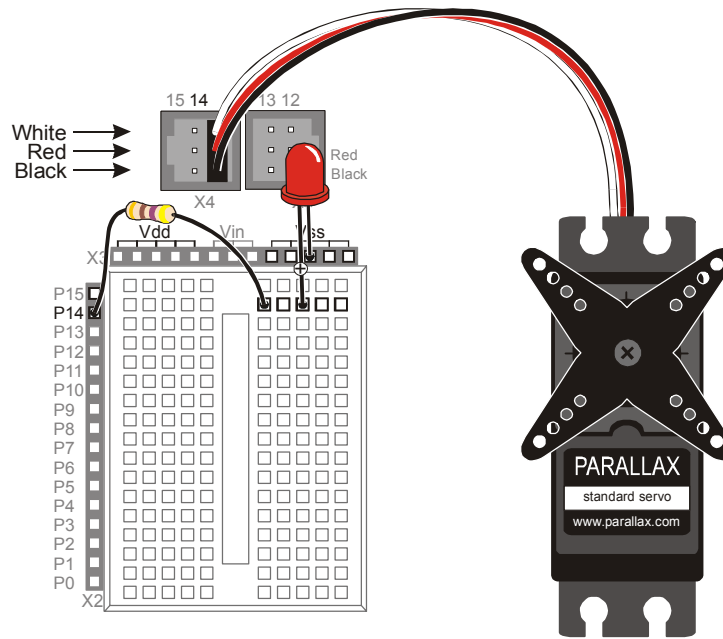


Figure 4-13
Connecting Servo to
Servo Header on the
Board of Education
Rev B.

4

- ✓ Make sure that the colors on the servo's cable align properly with the colors labeled in the picture.
- ✓ Connect a 9 V battery to Board of Education Rev B. The servo may move a little bit when you make the connection.

Programming Servo Control

A servo is controlled by very brief high signals. These high signals are sent over and over again every 20 ms. The high signals last anywhere between 1 and 2 ms. The **PULSOUT** command can be used to send a pulse (a very brief high signal) using a BASIC Stamp I/O pin. Here is the command syntax for the **PULSOUT** command:

PULSOUT *Pin, Duration*

As with **HIGH** and **LOW**, the *Pin* argument is a number that tells the BASIC Stamp which I/O pin the signal is sent on. The *Duration* argument is not milliseconds, like it is in the **PAUSE** command. For the BASIC Stamp 2, the *Duration* is the number of 2-millionth-of-a-second (μ s) time durations that you want the high signal to last.



A **millionth of a second** is called a microsecond. The Greek letter μ is used in place of the word micro and the letter s is used in place of second. This is handy for writing and taking notes, because instead of writing 2 microseconds, you can write $2\ \mu\text{s}$.

Reminder: one thousandth of a second is called a millisecond, and is abbreviated ms.

Fact: $1\ \text{ms} = 1000\ \mu\text{s}$. In other words, you can fit one thousand millionths of a second into one thousandth of a second.

The next example program will use the **PULSOUT** command to deliver pulses that instruct the servo on where to position its horn. **FOR...NEXT** loops are used to deliver a certain number of pulses, which cause the servo to hold a position for a certain amount of time.

Example Program: ServoTest.bs2

ServoTest.bs2 will make the servo's horn start at somewhere in the neighborhood of 10 o'clock and hold that position for about three seconds. Then, the program will move the servo horn clockwise, to about the 2 o'clock position for about three seconds. After that, the servo will hold its "center position", which is about 12 o'clock, for about three seconds.

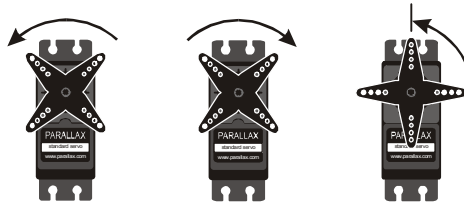


Figure 4-14
Servo Horn Motion

10 o'clock (left)
2 o'clock (middle)
12 o'clock (right)



What if my servo is different? We recommend that you use a Parallax Standard Servo for these activities. There are lots of different servos, and many will respond differently to the signals that TestServo.bs2 sends. Another brand of servo might only rotate to 11 o'clock then 1 o'clock, or maybe to 9 o'clock and then 3 o'clock. It might even rotate the opposite direction and start in the clockwise direction before it goes counterclockwise. But if your servo is rated for use with a 9 V battery, and the motion is easy to observe and consistent, it will work for these activities. You can modify the example programs to get the servo to behave the way you want.

- ✓ Enter ServoTest.bs2 into the BASIC Stamp Editor.
- ✓ Connect power to your Board of Education or HomeWork Board.
- ✓ Run the program.

- ✓ Observe the servo turns at each of the three steps in the program, and record where the horn is really pointing.
- ✓ Re-run the program and verify that the LED flickers dimly. It should be brightest when the BASIC Stamp sends the 10 o'clock signal and dimmest when the BASIC Stamp sends the 2 o'clock signal. This is because the LED circuit is only on for half as long (1 out of 21 ms instead of 2 out of 22 ms).

4

```
' What's a Microcontroller - ServoTest.bs2
' Test the servo at three different position signals.

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

counter          VAR      Word

DEBUG "Counterclockwise 10 o'clock", CR

FOR counter = 1 TO 150
  PULSOUT 14, 1000
  PAUSE 20
NEXT

DEBUG "Clockwise 2 o'clock", CR

FOR counter = 1 TO 150
  PULSOUT 14, 500
  PAUSE 20
NEXT

DEBUG "Center 12 o'clock", CR

FOR counter = 1 TO 150
  PULSOUT 14, 750
  PAUSE 20
NEXT

DEBUG "All done."

END
```

How ServoTest.bs2 Works

The first **FOR...NEXT** loop delivers 150 pulses, each of which lasts 2.0 ms. These pulses instruct the servo to go to a position that is roughly 10 o'clock if you think about it in terms of a clock face.

```
FOR counter = 1 TO 150
```

```

PULSOUT 14, 1000
PAUSE 20
NEXT

```



PULSOUT 14, 1000 sends a pulse that lasts $1000 \times 2 \mu\text{s}$. That's $2000 \mu\text{s}$ or 2 ms.

Figure 4-15 is called a timing diagram. It shows a picture of the high and low signals and how long they last. The timing diagram does not show how many pulses are delivered, but it does give you information about how long the high and low signals last. Each pulse (high signal) lasts for 2.0 ms. Each pulse is separated by a 20 ms delay while the signal is low.

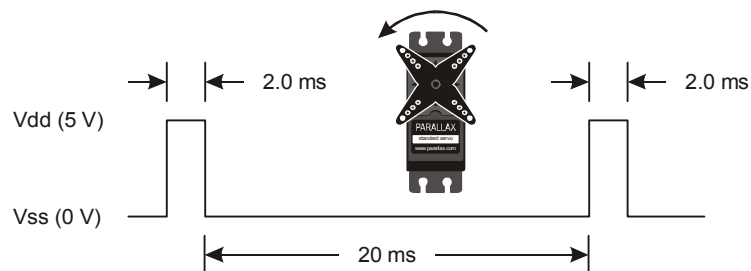


Figure 4-15
Timing Diagram for
2.0 ms Pulses Every
20 ms

*Servo horn in 10
o'clock position.*

The second **FOR...NEXT** loop delivers 150 pulses, but this time, each pulse only lasts 1.0 ms. This instructs the servo to turn to the 2 o'clock position for about 3.15 seconds.

```

FOR COUNTER = 1 TO 150
  PULSOUT 14, 500
  PAUSE 20
NEXT

```



PULSOUT 14, 500 sends a pulse that lasts $500 \times 2 \mu\text{s}$. That's $1000 \mu\text{s}$ or 1 ms.

Figure 4-15 shows the timing diagram for this pulse train. The pauses between pulses still last 20 ms.

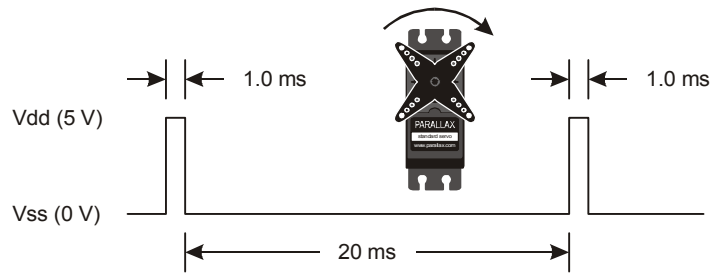


Figure 4-16
Timing Diagram for 1.0
ms Pulses Every 20 ms

*Servo horn in 2 o'clock
position.*

4

The last **FOR...NEXT** loop delivers 150 pulses, each of which lasts 1.5 ms. This instructs the servo to go to its center position (12 o'clock) for about 3.23 seconds.

```
FOR counter = 1 TO 150
  PULSOUT 14, 750
  PAUSE 20
NEXT
```



PULSOUT 14, 750 sends a pulse that lasts $750 \times 2 \mu\text{s}$. That's $1500 \mu\text{s}$ or 1.5 ms.

Figure 4-17 shows the timing diagram for these pulses. While the low time is still 20 ms, the pulse now lasts for 1.5 ms.

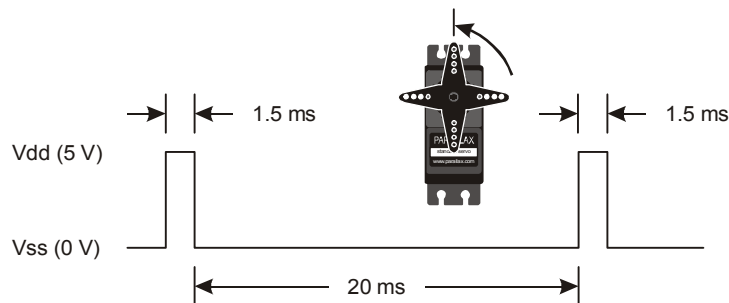


Figure 4-17
Timing Diagram for
1.5 ms Pulses Every
20 ms

*Servo horn is in 12
o'clock position.*

Do the Math

If you want to convert time from milliseconds to a *Duration* you can use for **PULSOUT**, use this equation.

$$\text{Duration} = \text{number of ms} \times 500$$

For example, if you didn't already know that the **PULSOUT** argument for 1.5 ms is 750, here is how you could calculate it.

$$\begin{aligned} \text{Duration} &= 1.5 \times 500 \\ &= 750 \end{aligned}$$

You can also figure out the *Duration* of a mystery **PULSOUT** command using this equation.

$$\text{number of ms} = \frac{\text{Duration}}{500} \text{ ms}$$

For example, if you see the command **PULSOUT 14, 850**, how long does that pulse really last?

$$\begin{aligned} \text{number of ms} &= \frac{850}{500} \text{ ms} \\ &= 1.7 \text{ ms} \end{aligned}$$

Your Turn – Adjusting Position and Hold Time

The number of times each **FOR...NEXT** loop repeats is what controls how long the servo stays in a given position. The value of the **PULSOUT** command's *Duration* argument controls where the servo turns to. It's important to experiment with changing these values to be sure how they work before moving on to the next experiment.

- ✓ Save ServoTest.bs2 as ServoTestYourTurn.bs2.
- ✓ Modify all the **FOR...NEXT** loops so that they execute half as many times as the original program:

```
FOR counter = 1 to 75
```

- ✓ Run the modified program and verify that the servo holds each position for half as long.
- ✓ Modify all the **FOR...NEXT** loops so that they execute twice as many times as the original program:

```
FOR counter = 1 to 300
```

- ✓ Run the modified program and verify that the servo holds each position for twice as long.
- ✓ Modify the **PULSOUT** command in the first loop so that it reads:

```
PULSOUT 14,850
```

- ✓ Modify the **PULSOUT** command in the second loop so that it reads:

```
PULSOUT 14,650
```

- ✓ Run the modified program, and explain the differences in the positions the servo turns to and holds.

4

ACTIVITY #2: CONTROLLING POSITION WITH YOUR COMPUTER

Factory automation often involves microcontrollers communicating with larger computers. The microcontrollers read sensors and transmit that data to the main computer. The main computer interprets and analyzes the sensor data, then sends position information back to the microcontroller. The microcontroller might then update a conveyor belt's speed, or a sorter's position, or some other mechanical, motor controlled task.

You can use the Debug Terminal to send messages from your computer to the BASIC Stamp as shown in Figure 4-18. The BASIC Stamp has to be programmed to listen for the messages you send using the Debug Terminal, and it also has to store the data you send in one or more variables.

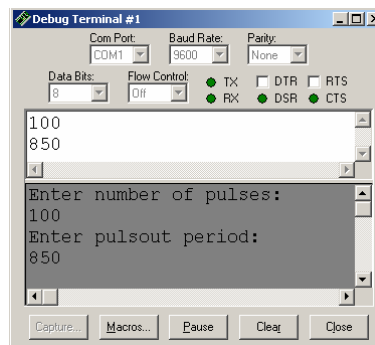


Figure 4-18
Sending Messages to
the BASIC Stamp

Click the white field above the message display pane and type your message. A copy of the message you entered appears in the lower windowpane. This copy is called an echo.

In this activity, you will program the BASIC Stamp to receive two values from the Debug Terminal:

1. The number of pulses to send to the servo
2. The **duration** value used by the **PULSOUT** command

You will also program the BASIC Stamp to use these values to control the servo.

Parts and Circuit

Same as Activity #1

Programming the BASIC Stamp to Receive Messages from Debug

Programming the BASIC Stamp to send messages to the Debug Terminal is done using the **DEBUG** command. Programming the BASIC Stamp to receive messages from the Debug Terminal is done using the **DEBUGIN** command. When using this command, you also have to declare one or more variables for the BASIC Stamp to store the information it receives. Here is an example of a variable you can declare for the BASIC Stamp to store a value:

```
pulses VAR Word
```

Later in the program, you can use this value to store a number received by the **DEBUGIN** command:

```
DEBUGIN DEC pulses
```

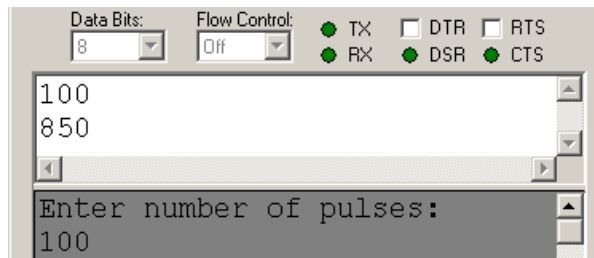
When the BASIC Stamp receives a numeric value from the Debug Terminal, it will store it in the **pulses** variable. The **DEC** formatter tells the **DEBUGIN** command that the characters you are sending will be digits that form a decimal number. As soon as you hit the carriage return, the BASIC Stamp will store the digits it received in the **pulses** variable as a decimal number, then move on.

Although it is not included in the example program, you can add a line to verify that the message was processed by the BASIC Stamp.

```
DEBUG CR, "You sent the value: ", DEC pulses
```

Example Program: ServoControlWithDebug.bs2

Figure 4-19 shows a close-up view of the Transmit Windowpane in the Debug Terminal. Your Transmit Windowpane may be smaller. You can click and drag the separator between the two panes downward to make the Transmit Windowpane larger. You can type characters into the Transmit Windowpane to send them to the BASIC Stamp. In this example, somebody typed in 100, then hit the carriage return, then typed in 850. The Receive Windowpane displays the “Enter number of pulses:” message sent by the BASIC Stamp. It also displays an echo of the characters 100 that were typed into the Transmit Windowpane.

**Figure 4-19**

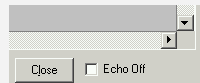
Debug Terminal's Windowpanes:

← *Transmit Windowpane*← *Receive Windowpane*

4



Echo is when you send a message, and a copy of that message appears in your Receive Windowpane. You can click the Echo Off checkbox (shown below) to make a checkmark appear in it. This will make the Debug Terminal stop displaying these echoes.



- ✓ Enter ServoControlWithDebug.bs2 into the BASIC Stamp Editor and run it.
- ✓ If the Transmit Windowpane is too small, resize it using your mouse to click, hold, and drag the separator downward. The separator is shown just above the text message: “Enter number of pulses:” in Figure 4-19.
- ✓ Click the upper, Transmit Windowpane to place your cursor there for typing messages.
- ✓ When the Debug Terminal prompts you to, “Enter number of pulses:”, type the number 100, then press enter.
- ✓ When the Debug Terminal prompts you to “Enter PULSOUT duration:” type the number 850, then press enter.



The PULSOUT Duration should be a number between 500 and 1000. If you enter numbers outside that range, the servo may try to rotate to a position beyond its own mechanical limits. Although it will not break the servo, it could shorten the device's lifespan.

The BASIC Stamp will display the message “Servo is running...” while it is sending pulses to the servo. When it is done sending pulses to the servo, it will display the message “Done” for one second. Then, it will prompt you to enter the number of pulses again. Have fun with it, but make sure to follow the directive in the caution box about staying between 500 and 1000 for your **PULSOUT** value.

- ✓ Experiment with entering other values between 500 and 1000 for the **PULSOUT** *Duration* and values between 1 and 65534 for the number of pulses.



It takes between 40 and 45 pulses to make the servo hold a position for 1 second.

```
' What's a Microcontroller - ServoControlWithDebug.bs2
' Send messages to the BASIC Stamp to control a servo using
' the Debug Terminal.

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

counter      Var      Word
pulses       Var      Word
duration     Var      Word

DO

  DEBUG CLS, "Enter number of pulses:", CR
  DEBUGIN DEC pulses

  DEBUG "Enter PULSOUT duration:", CR
  DEBUGIN DEC duration

  DEBUG "Servo is running...", CR

  FOR counter = 1 TO pulses
    PULSOUT 14, duration
    PAUSE 20
  NEXT

  DEBUG "DONE"
  PAUSE 1000

LOOP
```

How ServoControlWithDebug.bs2 Works

Three **word** variables are declared in this program:

```
counter      Var      WORD
pulses       Var      WORD
duration     Var      WORD
```

The **counter** variable is declared for use by a **FOR...NEXT** loop. See Chapter #2, Activity #3 for details. The **pulses** and **duration** variables are used a couple of different ways. They are both used to receive and store values sent from the Debug Terminal. The **pulses** variable is also used to set the number of repetitions in the **FOR...NEXT** loop that delivers pulses to the servo, and the **duration** variable is used to set the duration for the **PULSOUT** command.

The rest of the program is nested inside a **DO...LOOP** without a **WHILE** or **UNTIL** argument so that the commands execute over and over again.

```
DO
  ' Rest of program not shown.
LOOP
```

The **DEBUG** command is used to send you (the “user” of the software) a message to enter the number of pulses. Then, the **DEBUGIN** command waits for you to enter digits that make up the number and press the Enter key on your keyboard. The digits that you enter are converted to a value that is stored in the **pulses** variable. This process is repeated with a second **DEBUG** and **DEBUGIN** command that loads a value into the **duration** variable too.

```
DEBUG CLS, "Enter number of pulses:", CR
DEBUGIN DEC pulses

DEBUG "Enter PULSOUT duration:", CR
DEBUGIN DEC duration
```

After you enter the second value, it’s useful to display a message while the servo is running so that you don’t try to enter a second value:

```
DEBUG "Servo is running...", CR
```

While the servo is running, you can gently try to move the servo horn away from the position it is holding. The servo resists light pressure applied to the horn.



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

This is the **FOR...NEXT** loop syntax from the BASIC Stamp Manual. It shows that you need a **Counter**, **StartValue** and **EndValue** to control how many times the loop repeats itself. There is also an optional **StepValue** if you want to add a number other than 1 to the value of **Counter** each time through the loop.

As in previous examples, the `counter` variable is used as an index for the `FOR...NEXT` loop. Up until this example, all the `FOR...NEXT` loops have used constants such as 10 or 150 for `EndValue`. In this `FOR...NEXT` loop, the value of the `pulses` variable is used to control the `EndValue` of the `FOR...NEXT` loop. This, in turn, controls how many pulses are delivered to the servo. The end result is that the `pulses` variable controls how long the servo holds a given position. Up until now, constant values such as 500, 750, and 1000 were also used for the `PULSOUT` command's *Duration* argument. Look carefully at this `FOR...NEXT` loop to see where and how these variables are used:

```
FOR counter = 1 to pulses
  PULSOUT 14, duration
  PAUSE 20
NEXT
```



Take some time to understand this `FOR...NEXT` loop. It is one of the first examples of the amazing things you can do with variables in PBASIC command arguments, and it also highlights how useful a programmable microcontroller like the BASIC Stamp can be.

Your Turn – Setting Limits in Software

The example program doesn't stop you or anybody else from entering a `PULSOUT` *duration* such as 1500, which is not good for the servo. This is a problem that needs to be fixed if you are designing this system into a product.

Let's imagine that this computer servo control system is one that has been developed for remote-controlling a door. Perhaps a security guard will use this to open a shipping door that he or she watches on a remote camera. Maybe a college student will use it to control doors in a maze that mice navigate in search of food. Maybe a military gunner will use it to point the cannon at a particular target. If you are designing the product for somebody else to use, the last thing you want is to give the user (security guard, college student, military gunner) the ability to enter the wrong number and damage the equipment.

To fix this problem, try this:

- ✓ Save the example program `ServoControlWithDebug.bs2` under the new name `ServoControlWithDebugYourTurn.bs2`.
- ✓ Replace these two commands:

```
DEBUG "Enter pulsout duration:", CR
DEBUGIN DEC duration
```

with this code block:

```
DO
  DEBUG "Enter pulsout duration:", CR
  DEBUGIN DEC duration
  IF duration < 500 THEN
    DEBUG "Value of duration must be above 499", CR
    PAUSE 1000
  ENDIF
  IF duration > 1000 THEN
    DEBUG "Value of duration must be below 1001", CR
    PAUSE 1000
  ENDIF
LOOP UNTIL duration > 499 AND duration < 1001
```

- ✓ Save the program.
- ✓ Run the program and verify that it rejects values outside the appropriate range for the servo.

ACTIVITY #3: CONVERTING POSITION TO MOTION

In this activity, you will program the servo to change position at different rates. By changing position at different rates, you will cause your servo horn to rotate at different speeds. You can use this technique to make the servo control motion instead of position.

Programming a Rate of Change for Position

You can use a **FOR...NEXT** loop to make a servo sweep through its range of motion like this:

```
FOR counter = 500 TO 1000
  PULSOUT 14, counter
  PAUSE 20
NEXT
```

The **FOR...NEXT** loop causes the servo's horn to start at around 2 o'clock and then rotate slowly counterclockwise until it gets to 10 o'clock. Because **counter** is the index of the **FOR...NEXT** loop, it increases by one each time through. The value of **counter** is also used in the **PULSOUT** command's *Duration* argument, which means the *duration* of each pulse gets a little longer each time through the loop. Since the *duration* changes, so does the position of the servo's horn.

FOR...NEXT loops have an optional **STEP** argument. The **STEP** argument can be used to make the servo rotate faster. For example, you can use the **STEP** argument to add 8 to

counter each time through the loop (instead of 1) by modifying the **FOR** statement like this:

```
FOR counter = 500 TO 1000 STEP 8
```

You can also make the servo turn the opposite direction by counting down instead of counting up. In PBASIC, **FOR...NEXT** loops will also count backwards if the *StartValue* argument is larger than the *EndValue* argument. Here is an example of how to make a **FOR...NEXT** loop count from 1000 to 500:

```
FOR counter = 1000 TO 500
```

You can combine counting down with a **STEP** argument to get the servo to rotate more quickly in the clockwise direction like this:

```
FOR counter = 1000 TO 500 STEP 20
```

The trick to getting the servo to turn at different rates is to use these **FOR...NEXT** loops to count up and down with different step sizes. The next example program uses these techniques to make the servo's horn rotate back and forth at different rates.

Example Program: ServoVelocities.bs2

- ✓ Enter and run the program.
- ✓ As the program runs, watch how the value of **counter** changes in the Debug Terminal.
- ✓ Also, watch how the servo behaves differently through the two different **FOR...NEXT** loops. Both the servo's direction and speed change.

```
' What's a Microcontroller - ServoVelocities.bs2
' Rotate the servo counterclockwise slowly, then clockwise rapidly.

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

counter      VAR      Word

DO

DEBUG "Pulse width increment by 8", CR

    FOR counter = 500 TO 1000 STEP 8
        PULSOUT 14, counter
        PAUSE 7
        DEBUG DEC5 counter, CR, CR$CR$
    NEXT
```

```

DEBUG CR, "Pulse width decrement by 20", CR

FOR counter = 1000 TO 500 STEP 20
  PULSOUT 14, counter
  PAUSE 7
  DEBUG DEC5 counter, CR, CRSRUP
NEXT

DEBUG CR, "Repeat", CR

LOOP

```

How ServoVelocities.bs2 Works

The first **FOR...NEXT** loop counts upwards from 500 to 1000 in steps of 8. Since the **counter** variable is used as the **PULSOUT** command's **Duration** argument, the servo horn's position rotates counterclockwise by steps that are four times the smallest possible step.

```

FOR counter = 500 TO 1000 STEP 8
  PULSOUT 14, counter
  PAUSE 7
  DEBUG DEC5 counter, CR, CRSRUP
NEXT

```



Why PAUSE 7 instead of PAUSE 20? The command **DEBUG DEC5 counter, CR, CRSRUP** takes about 8 ms to execute. This means that **PAUSE 12** would maintain the 20 ms delay between pulses. A few trial and error experiments showed that **PAUSE 7** gave the servo the smoothest motion. Your servo may be different.



More DEBUG formatters and control characters are featured in the **DEBUG** command that displays the value of the **counter** variable. This value is printed using the 5-digit decimal format (**DEC5**). After the value is printed, there is a carriage return (**CR**). After the carriage return, the formatter **CRSRUP** (cursor up) sends the cursor back up to the previous line. This causes the new value of counter to be printed over the old value each time through the loop.

The second **FOR...NEXT** loop counts downwards from 1000 back to 500 in steps of 20. The **counter** variable is also used as an argument for the **PULSOUT** command in this example, so the servo horn rotates clockwise.

```

FOR counter = 1000 TO 500 STEP 20
  PULSOUT 14, counter

```

```
PAUSE 7
DEBUG DEC5 counter, CR, CRSRUP
NEXT
```

Your Turn – Adjusting the Velocities

- ✓ Try different **STEP** values to make the servo turn at different rates.
- ✓ Re-run the program after each modification.
- ✓ Observe the effect of each new **STEP** value on how fast the servo horn turns.
- ✓ Experiment with different **PAUSE** command *Duration* values (between 3 and 12) to find the value that gives the servo the smoothest motion for each new **STEP** value.

ACTIVITY #4: PUSHBUTTON CONTROLLED SERVO

In this chapter, you have written programs that make the servo go through a pre-recorded set of motions, and you have controlled the servo using the Debug Terminal. You can also program the BASIC Stamp to control the servo based on pushbutton inputs. In this activity you will:

- Build a circuit for a pushbutton controlled servo.
- Program the BASIC Stamp to control the servo based on the pushbutton inputs.

When you are done, you will be able to push one button to get the BASIC Stamp to rotate the servo in one direction, and another button to get the servo to rotate in the other direction. When no buttons are pressed, the servo will hold whatever position it moved to.

Extra Parts for Pushbutton Servo Control

The same parts from the previous activities in this chapter are still used. You will need to gather the following parts for the pushbutton circuits:

- (2) Pushbuttons – normally open
- (2) Resistors – 10 k Ω (brown-black-orange)
- (2) Resistors – 220 Ω (red-red-brown)
- (3) Jumper wires

Adding the Pushbutton Control Circuit

Figure 4-20 shows the pushbutton circuits that you will use to control the servo.

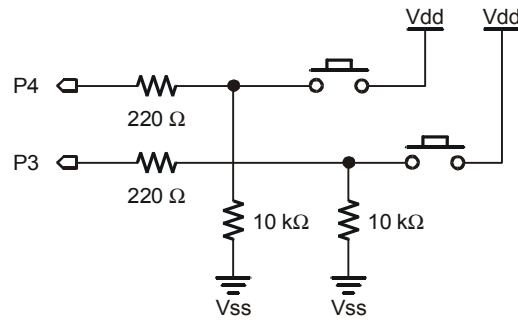


Figure 4-20
Pushbutton
Circuits for
Servo Control

4

✓ Add this circuit to the servo+LED circuit that you have been using up to this point. When you are done your circuit should resemble:

- Figure 4-21 if you are using the Board of Education Rev C
- Figure 4-22 if you are using the HomeWork Board
- Figure 4-23 if you are using the Board of Education Rev B

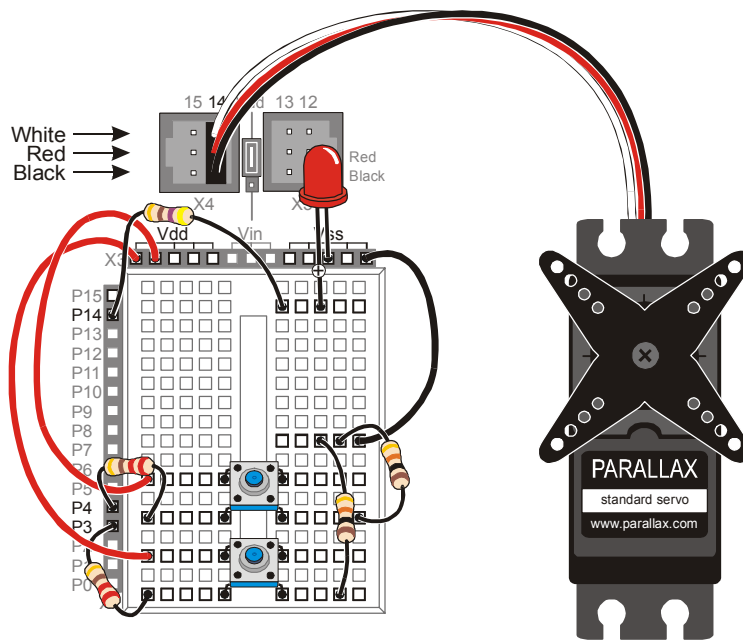


Figure 4-21
Board of Education
Rev C Servo
Circuit with
Pushbutton Circuits
Added

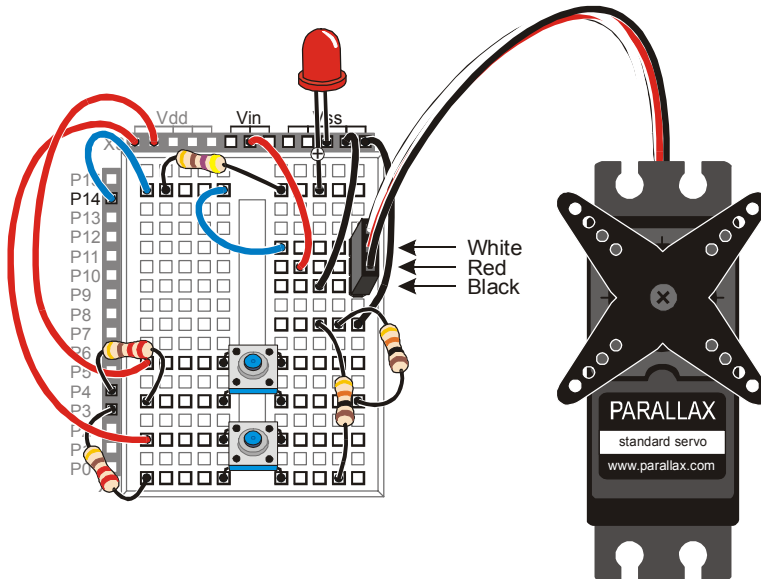


Figure 4-22
HomeWork Board
Servo Circuit with
Pushbutton Circuits
Added

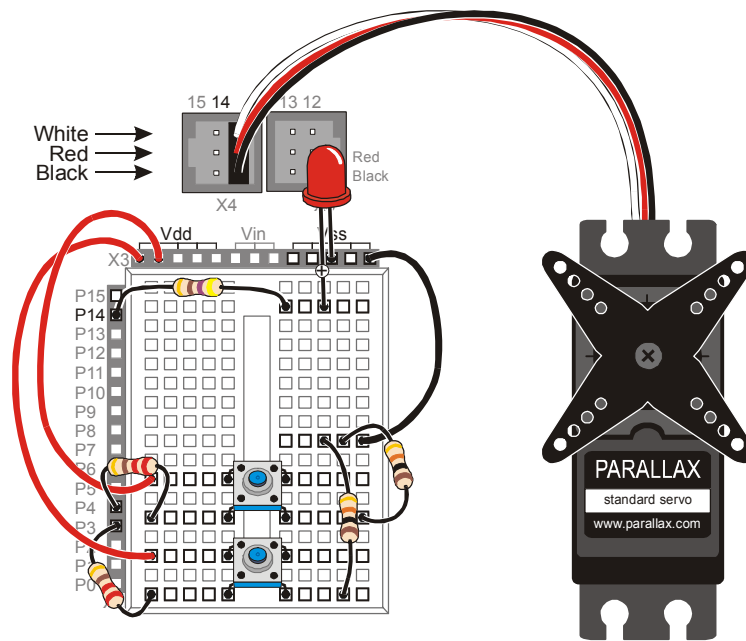


Figure 4-23
Board of Education
Rev B Servo
Circuit with
Pushbutton Circuits
Added

4

- ✓ Test the pushbutton connected to P3 using the original version of ReadPushbuttonState.bs2. The section that has this program and the instructions on how to use it begins on page 77.
- ✓ Modify the program so that it reads P4.
- ✓ Run the modified program to test the pushbutton connected to P4.

Programming Pushbutton Servo Control

Pushbutton servo control is not much different from pushbutton LED control. **IF...THEN** code blocks are used to check the pushbutton states and either add or subtract from a variable called **duration**. This variable is used in the **PULSOUT** command's **Duration** argument. If one of the pushbuttons is pressed, the value of **duration** increases. If the other pushbutton is pressed, the value of **duration** decreases. A nested **IF...THEN** statement is used to decide if the **duration** variable is too large (greater than 1000) or too small (smaller than 500).

Example Program: ServoControlWithPushbuttons.bs2

This example program makes the servo's horn rotate counterclockwise when the pushbutton connected to P4 is pressed. The servo's horn will keep rotating so long as the pushbutton is held down and the value of `duration` is smaller than 1000. When the pushbutton connected to P3 is pressed, the servo horn rotates clockwise. The servo also is limited in its clockwise motion because the `duration` variable is not allowed to go below 500. The Debug Terminal displays the value of `duration` while the program is running.

- ✓ Enter the ServoControlWithPushbuttons.bs2 program into the BASIC Stamp Editor and run it.
- ✓ Verify that the servo turns counterclockwise when you press and hold the pushbutton connected to P4.
- ✓ Verify that as soon as the limit of `duration > 1000` is reached or exceeded that the servo stops turning any further in the counterclockwise direction.
- ✓ Verify that the servo turns clockwise when you press and hold the pushbutton connected to P3.
- ✓ Verify that as soon as the limit of `duration < 500` is reached or exceeded that the servo stops turning any further in the clockwise direction.

```
' What's a Microcontroller - ServoControlWithPushbuttons.bs2
' Press and hold P4 pushbutton to rotate the servo counterclockwise,
' or press the pushbutton connected to P3 to rotate the servo clockwise.

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

duration      VAR      Word

duration = 750

DO

  IF IN3 = 1 THEN
    IF duration > 500 THEN
      duration = duration - 25
    ENDIF
  ENDIF
```

```

IF IN4 = 1 THEN
  IF duration < 1000 THEN
    duration = duration + 25
  ENDIF
ENDIF

PULSOUT 14, duration
PAUSE 10

DEBUG HOME, DEC4 duration, " = duration"

LOOP

```

Your Turn – Software Stoppers

Servos have a built in mechanical stopper that prevents them from turning too far. If you try to send a command like `PULSOUT 14, 2000`, the servo will not turn to a position that corresponds to a *Duration* argument of 2000. This is because servos have built-in mechanical stoppers that limit the range of motion. By gently turning the horn, you can feel when the servo's internal gears run into this mechanical stopper. You can modify the example program in this activity so that the BASIC Stamp limits the servo's motion to a range that is narrower than the limits imposed by the mechanical stoppers.

- ✓ Save ServoControlWithPushbuttons.bs2 under a new name.
- ✓ Adjust the software limits imposed on the servo's motion so that they are 650 and 850 instead of 500 and 1000.
- ✓ Adjust the software imposed rate so that the *duration* variable is incremented or decremented by 10 instead of 25.
- ✓ Decide what differences you expect to see in the way the servo behaves when you press the pushbutton.
- ✓ Run the program and compare the actual results with your expected results.

SUMMARY

This chapter introduced microcontrolled motion using a servo. A servo is a device that moves to and holds a particular position based on electronic signals it receives. These signals take the form of pulses that last anywhere between 1 and 2 ms, and they have to be delivered every 20 ms for the servo to maintain its position.

A programmer can use the **PULSOUT** command to make the BASIC Stamp send these signals. Since pulses have to be delivered every 20 ms for the servo to hold its position, the **PULSOUT** and **PAUSE** commands are usually placed in some kind of loop. Variables can be used to store the value used in the **PULSOUT** command's *Duration* argument. This causes the servo's horn to rotate in steps.

In this chapter, a variety of ways to get the values into the variables were presented. The variable can receive the value from your Debug Terminal using the **DEBUGIN** command. The value of the variable can pass through a sequence of values if the same variable is used as the index for a **FOR...NEXT** loop. This technique can be used to cause the servo to make sweeping motions. **IF...THEN** statements can be used to monitor pushbuttons and add or subtract from the variable used in the **PULSOUT** command's *Duration* argument based on whether or not a certain button is pressed. This allows both position control and sweeping motions depending on how the program is constructed and how the pushbuttons are operated.

Questions

1. What are the four external parts on a servo? What are they used for?
2. Is an LED circuit required to make a servo work?
3. What command controls the low time in the signal sent to a servo? What command controls the high time?
4. What programming element can you use to control the amount of time that a servo holds a particular position?
5. How do you use the Debug Terminal to send messages to the BASIC Stamp? What programming command is used to make the BASIC Stamp receive messages from the Debug Terminal?
6. What type of code block can you write to limit the servo's range of motion?

Exercises

1. Write a code block that sweeps the value of **PULSOUT** controlling a servo from a *Duration* of 700 to 800, then back to 700, in increments of (a) 1, (b) 4.
2. Add a nested **FOR...NEXT** loop to your answer to exercise 1b so that it delivers ten pulses before incrementing the **PULSOUT Duration** argument by 4.

Project

1. Modify ServoControlWithDebug.bs2 so that it monitors a kill switch. If the kill switch (P3 pushbutton) is pressed, Debug Terminal should not accept any commands. It should display: "Press Start switch to start machinery". When the start switch (P4 pushbutton) is pressed, the program should function normally. If power is disconnected and reconnected, the program should behave as though the kill switch has been pressed.

Solutions

- Q1. 1. Plug – Connects servo to power and signal sources
 2. Cable – Conducts power and signals from plug into the servo.
 3. Horn – The moving part of the servo.
 4. Case – Contains DC motor, gears, and control circuits.
- Q2. No, the LED just helps us see what's going on with the control signals.
- Q3. The low time is controlled with the **PAUSE** command. The high time is controlled with the **PULSOUT** command.
- Q4. A **FOR...NEXT** loop.
- Q5. Click the white field above the message display pane in the Debug Terminal and type the message you'd like to send. Use the **DEBUGIN** command to receive the typed characters in the BASIC Stamp.
- Q6. A nested **IF...THEN** statement.

E1. a)

```
FOR counter = 700 TO 800
  PULSOUT 14, counter
  PAUSE 20
NEXT
FOR counter = 800 TO 700
  PULSOUT 14, counter
  PAUSE 20
NEXT
```

b) Add "STEP 4" to both FOR loops.

```
FOR counter = 700 TO 800 STEP 4
  PULSOUT 14, counter
  PAUSE 20
NEXT
FOR counter = 800 TO 700 STEP 4
  PULSOUT 14, counter
  PAUSE 20
NEXT
```

E2.

```

FOR counter = 700 TO 800 STEP 4
  FOR pulses = 1 TO 10
    PULSOUT 14, counter
    PAUSE 20
  NEXT
NEXT
FOR counter = 800 TO 700 STEP 4
  FOR pulses = 1 TO 10
    PULSOUT 14, counter
    PAUSE 20
  NEXT
NEXT

```

P1. There are many possible solutions; one is given below.

```

' What's a Microcontroller - Ch04Prj01Soln2__KillSwitch.bs2
' Send messages to the BASIC Stamp to control a servo using
' the Debug Terminal as long as kill switch is not being pressed.

' Contributed by: Professor Clark J. Radcliffe, Department
' of Mechanical Engineering, Michigan State University

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

counter VAR Word
pulses VAR Word
duration VAR Word
DO
  PAUSE 2000
  IF (IN3 = 1) AND (IN4 = 0) THEN
    DEBUG "Press Start switch to start machinery.      ", CR ,CRSRUP
  ELSEIF (IN3 = 0) AND (IN4 = 1) THEN
    DEBUG CLS, "Enter number of pulses:", CR
    DEBUGIN DEC pulses

    DEBUG "Enter PULSOUT duration:", CR
    DEBUGIN DEC duration

    DEBUG "Servo is running...", CR

    FOR counter = 1 TO pulses
      PULSOUT 14, duration
      PAUSE 20
    NEXT

    DEBUG "DONE"
    PAUSE 2000
  ENDIF
LOOP

```

Further Investigation

The servo, and using sensors to control servos, can be investigated in detail in a variety of Stamps in Class texts.

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

Advanced Robotics: with the Toddler uses servos to control the motions of the Parallax Toddler robot’s legs. Although we take walking for granted, programming a machine to walk, maneuver, and keep its balance can be very challenging. This walking robot is recommended for advanced students who have already mastered the concepts in *What’s a Microcontroller?* and either *Robotics with the Boe-Bot* or *SumoBot*.

“Robotics with the Boe-Bot”, Student Workbook, Version 2.0 Parallax Inc., 2003

Robotics with the Boe-Bot makes use of the same servo control principles you just learned with a twist; servos can also be modified and used as rolling robot motors. Using the BASIC Stamp, Board of Education, and Robotics with the Boe-Bot kit, this text starts with the basics of robot navigation, then guides you through navigation with sensors. It also introduces some more in-depth topics such as solving problems with artificial intelligence and navigation using elementary control systems.

“SumoBot”, Student Workbook, Version 1.1, Parallax Inc., 2002

Robot Sumo is a very exciting competition that is fun for observers and participants. *SumoBot* is a guided tour through building, testing, and competing with your own autonomous Mini-Sumo class SumoBot® robot. This textbook offers a condensed presentation of the *Robotics with the Boe-Bot* text material applied towards the goal of winning a robotic sumo wrestling contest.

Chapter #5: Measuring Rotation

ADJUSTING DIALS AND MONITORING MACHINES

Many households have dials to control the lighting in a room. Twist the dial one direction, and the light gets brighter; twist the dial in the other direction, and the light gets dimmer. Model trains use dials to control motor speed and direction. Many machines have dials or cranks used to fine tune the position of cutting blades and guiding surfaces.

Dials can also be found in audio equipment, where they are used to adjust how music and voices sound. Figure 5-1 shows a simple example of a dial with a knob that is turned to adjust the speaker's volume. By turning the knob, a circuit inside the speaker changes, and the volume of the music the speaker plays changes. Similar circuits can also be found inside joysticks, and even inside the servo used in Chapter #4: Controlling Motion.

5



Figure 5-1
Volume
Adjustment
on a Speaker

THE VARIABLE RESISTOR UNDER THE DIAL – A POTENTIOMETER

The device inside sound system dials, joysticks and servos is called a potentiometer, often abbreviated as a “pot”. Figure 5-2 shows a picture of some common potentiometers. Notice that they all have three pins.



Figure 5-2
A Few
Potentiometer
Examples

Figure 5-3 shows the schematic symbol and part drawing of the potentiometer you will use in this chapter. Terminals A and B are connected to a 10 k Ω resistive element. Terminal W is called the wiper terminal, and it is connected to a wire that touches the resistive element somewhere between its ends.

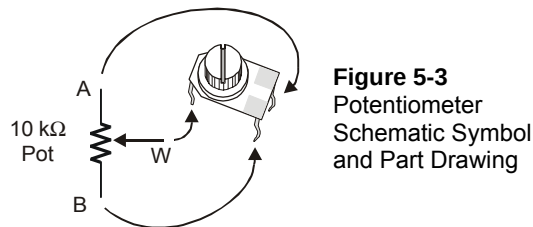


Figure 5-3
Potentiometer
Schematic Symbol
and Part Drawing

Figure 5-4 shows how the wiper on a potentiometer works. As you adjust the knob on top of the potentiometer, the wiper terminal contacts the resistive element at different places. As you turn the knob clockwise, the wiper gets closer to the A terminal, and as you turn the knob counterclockwise, the wiper gets closer to the B terminal.

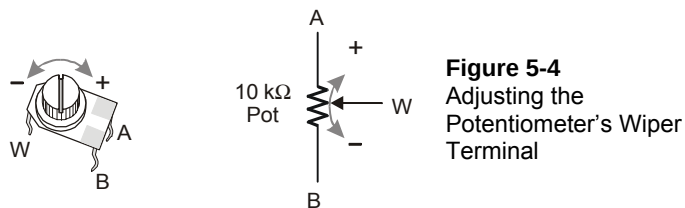


Figure 5-4
Adjusting the
Potentiometer's Wiper
Terminal

ACTIVITY #1: BUILDING AND TESTING THE POTENTIOMETER CIRCUIT

Placing different size resistors in series with an LED causes different amounts of current to flow through the circuit. Large resistance in the LED circuit causes small amounts of current to flow through the circuit, and the LED glows dimly. Small resistances in the LED circuit causes more current to flow through the circuit, and the LED glows more brightly. By connecting the W and A terminals of the potentiometer, in series with an LED circuit, you can use it to adjust the resistance in the circuit. This in turn adjusts the brightness of the LED. In this activity, you will use the potentiometer as a variable resistor and use it to change the brightness of the LED.

5**Dial Circuit Parts**

- (1) Potentiometer – 10 k Ω
- (1) Resistor – 220 Ω (red-red-brown)
- (1) LED – any color
- (1) Jumper wire

Building the Potentiometer Test Circuit

Figure 5-5 shows a circuit that can be used for adjusting the LED's brightness with a potentiometer.

✓ Build the circuit shown in Figure 5-5.



Tip: Use a needle-nose pliers to straighten the kinks out of the potentiometer's legs before plugging the device into the breadboard. When the potentiometer's legs are straight, they maintain better contact with the breadboard sockets.

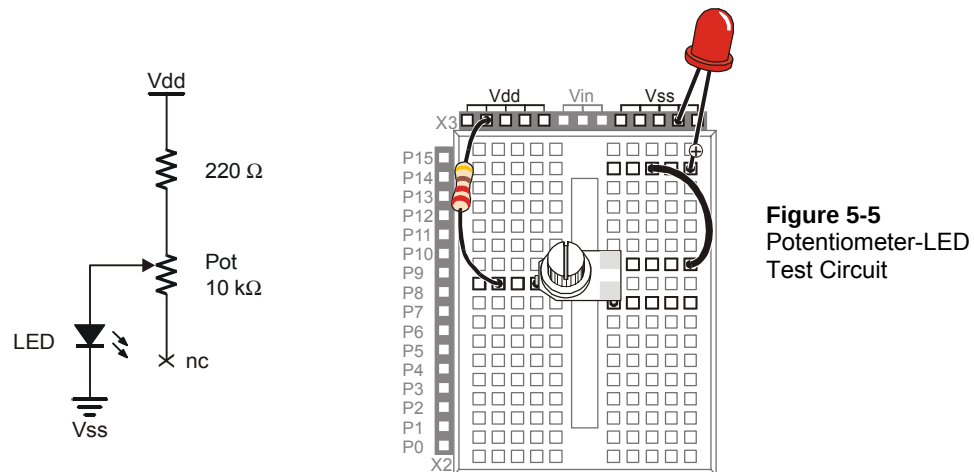


Figure 5-5
Potentiometer-LED
Test Circuit

Testing the Potentiometer Circuit

- ✓ Turn the potentiometer clockwise until it reaches its mechanical limit shown in Figure 5-6.



Handle with care: If your potentiometer will not turn this far, do not try to force it. Just turn it until it reaches its mechanical limit; otherwise, it might break.

- ✓ Gradually rotate the potentiometer counterclockwise to the positions shown in Figure 5-6 (b), (c), (d), (e), and (f) noting the how brightly the LED glows at each position.

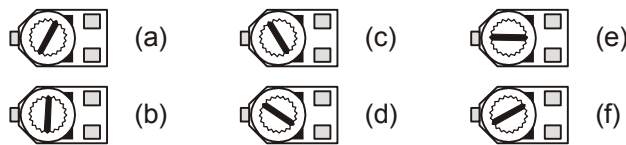


Figure 5-6
Potentiometer Input Shaft

(a) through (f) show the potentiometer's wiper terminal set to different positions.

How the Potentiometer Circuit Works

The total resistance in your test circuit is $220\ \Omega$ plus the resistance between the A and W terminals of the potentiometer. This value could be anywhere from 0 to $10\ \text{k}\Omega$. As you

turn the potentiometer's input shaft, the resistance between the A and W terminals changes. This in turn changes the current flow through the LED circuit.

ACTIVITY #2: MEASURING RESISTANCE BY MEASURING TIME

This activity introduces a new part called a capacitor. A capacitor behaves like a rechargeable battery that only holds its charge for short durations of time. This activity also introduces RC-time, which is an abbreviation for resistor-capacitor time. RC-time is a measurement of how long it takes for a capacitor to lose a certain amount of its stored charge as it supplies current to a resistor. By measuring the time it takes for the capacitor to discharge with different size resistors and capacitors, you will become more familiar with RC-time. In this activity, you will program the BASIC Stamp to charge a capacitor and then measure the time it takes the capacitor to discharge through a resistor.

5

Introducing the Capacitor

Figure 5-7 shows the schematic symbol and part drawing for the type of capacitor used in this activity. Capacitance value is measured in microfarads (μF), and the measurement is typically printed on the capacitors. The cylindrical case of the capacitor is called a canister.



This capacitor has a positive (+) and a negative (-) terminal. The negative terminal is the lead that comes out of the metal canister closest to the stripe with a negative (-) sign. Always make sure to connect these terminals as shown in the circuit diagrams. Connecting one of these capacitors incorrectly can damage it. In some circuits, connecting this type of capacitor incorrectly and then connecting power can cause it to rupture or even explode.

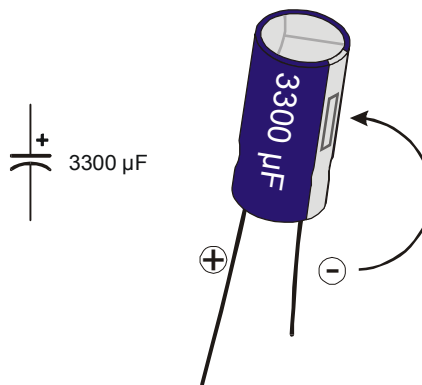


Figure 5-7
3300 μF Capacitor
Schematic Symbol
and Part Drawing

*Pay careful attention
to the leads and how
they connect to the
Positive and
Negative Terminals.*

Resistance and Time Parts

- (1) Capacitor – 3300 μF
- (1) Capacitor – 1000 μF
- (1) Resistors – 220 Ω (red-red-brown)
- (1) Resistor – 470 Ω (yellow-violet-brown)
- (1) Resistor – 1 k Ω (brown-black-red)
- (1) Resistor – 2 k Ω (red-black-red)
- (1) Resistor – 10 k Ω (brown-black-orange)



Recommended Equipment: Safety goggles or safety glasses.

Building and Testing the Resistance Capacitance (RC) Time Circuit

Figure 5-8 shows the circuit schematic and Figure 5-9 shows the wiring diagram for this activity. You will be taking time measurements using different resistor values in place of the resistor labeled R_i .



SAFETY

Always observe polarity when connecting the 3300 μF capacitor. Remember, the negative terminal is the lead that comes out of the metal canister closest to the stripe with a negative (-) sign. Use Figure 5-7 to identify the (+) and (-) terminals.

Your 3300 μF capacitor will work fine in this experiment so long as you make sure that the positive (+) and negative (-) terminals are connected EXACTLY as shown in Figure 5-8 and Figure 5-9.

Never reverse the supply polarity on the 3300 μF or any other polar capacitor. The voltage at the capacitor's (+) terminal must always be higher than the voltage at its (-) terminal. V_{ss} is the lowest voltage (0 V) on the Board of Education and BASIC Stamp HomeWork Board. By connecting the capacitor's negative terminal to V_{ss} , you ensure that the polarity across the capacitor's terminals will always be correct.

Wear safety goggles or safety glasses during this activity.

Always disconnect power before you build or modify circuits.

Keep your hands and face away from this capacitor when power is connected.

- ✓ With power disconnected, build the circuit as shown starting with a 470 Ω resistor in place of the resistor labeled R_i .

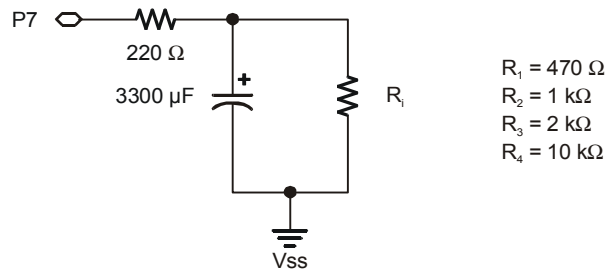


Figure 5-8
Schematic for Viewing
RC-time Voltage Decay

*Four different resistors
will be used as R_i
shown in the schematic.
First, the schematic will
be built and tested with
 $R_i = 470\ \Omega$, then $R_i = 1\ \text{k}\Omega$, etc.*

5

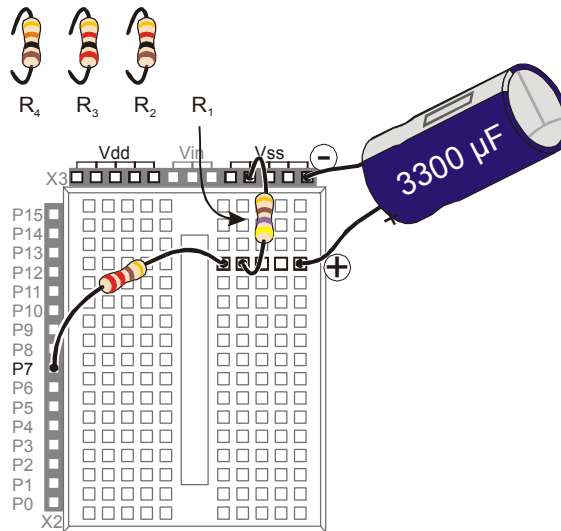


Figure 5-9
Wiring Diagram for
Figure 5-8

*Make sure that the
negative lead of the
capacitor is connected
on your board the same
way it is shown in this
figure, with the negative
lead connected to Vss.*

Polling the RC-Time Circuit with the BASIC Stamp

Although a stopwatch can be used to record how long it takes the capacitor's charge to drop to a certain level, the BASIC Stamp can also be programmed to monitor the circuit and give you a more reliable time measurement.

Example Program: PolledRcTimer.bs2

- ✓ Enter and run PolledRcTimer.bs2.
- ✓ Observe how the BASIC Stamp charges the capacitor and then measures the discharge time.

- ✓ Record the measured time (the capacitor's discharge time) in the 470 Ω row of Table 5-1.
- ✓ Disconnect power from your Board of Education or BASIC Stamp HomeWork Board.
- ✓ Remove the 470 Ω resistor labeled R_i in Figure 5-8 and Figure 5-9 on page 145, and replace it with the 1 k Ω resistor.
- ✓ Reconnect power to your board.
- ✓ Record your next time measurement (for the 1 k Ω resistor).
- ✓ Repeat these steps for each resistor value in Table 5-1.

Table 5-1: Resistance and RC-time for C = 3300 μF	
Resistance (Ω)	Measured Time (s)
470	
1 k	
2 k	
10 k	

```
' What's a Microcontroller - PolledRcTimer.bs2
' Reaction timer program modified to track an RC-time voltage decay.

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

timeCounter    VAR    Word
counter        VAR    Nib

DEBUG CLS

HIGH 7
DEBUG "Capacitor Charging...", CR

FOR counter = 5 TO 0
    PAUSE 1000
    DEBUG DEC2 counter, CR, CR$RUP
NEXT

DEBUG CR, CR, "Measure decay time now!", CR, CR
INPUT 7

DO

    PAUSE 100
    timeCounter = timeCounter + 1
```



```

DEBUG ? IN7
DEBUG DEC5 timeCounter, CR, CRSRUP, CRSRUP

LOOP UNTIL IN7 = 0

DEBUG CR, CR, CR, "The RC decay time was ",
DEC timeCounter, CR,
"tenths of a second.", CR, CR

END

```

How PolledRcTimer.bs2 Works

Two variables are declared. The **timeCounter** variable is used to track how long it takes the capacitor to discharge through R_i . The **counter** variable is used to count down while the capacitor is charging.

```

timeCounter    VAR    Word
counter        VAR    Nib

```

The command **DEBUG CLS** clears the Debug Terminal so that it doesn't get cluttered with successive measurements. **HIGH 7** sets P7 high and starts charging the capacitor, then a "Capacitor charging..." message is displayed. After that, a **FOR...NEXT** loop counts down while the capacitor is charging. As the capacitor charges, the voltage across its terminals increases toward anywhere between 2.5 and 4.9 V (depending on the value of R_i).

```

DEBUG CLS

HIGH 7
DEBUG "Capacitor Charging...", CR

FOR counter = 5 TO 0
  PAUSE 1000
  DEBUG DEC2 counter, CR, CRSRUP
NEXT

```

A message announces when the decay starts getting polled.

```

DEBUG CR, CR, "Measure decay time now!", CR, CR

```

In order to let the capacitor discharge itself through the R_i resistor, the I/O pin is changed from **HIGH** to **INPUT**. As an input, the I/O pin, has no effect on the circuit, but it can sense high or low signals. As soon as the I/O pin releases the circuit, the capacitor

discharges as it feeds current through the resistor. As the capacitor discharges, the voltage across its terminals gets lower and lower (decays).

```
INPUT 7
```

Back in the pushbutton chapter, you used the BASIC Stamp to detect a high or low signal using the variables **IN3** and **IN4**. At that time, a high signal was considered Vdd, and a low signal was considered Vss. It turns out that a high signal is any voltage above 1.4 V. Of course, it could be up to 5 V. Likewise, a low signal is anything between 1.4 V and 0 V. This **DO...LOOP** checks P7 every 100 ms until the value of **IN7** changes from 1 to 0, which indicates that the capacitor voltage decayed below 1.4 V.

```
DO

    PAUSE 100
    timeCounter = timeCounter + 1

    DEBUG ? IN7
    DEBUG DEC5 timeCounter, CR, CRSRUP, CRSRUP

LOOP UNTIL IN7 = 0
```

The result is then displayed and the program ends.

```
DEBUG CR, CR, CR, "The RC decay time was ",

    DEC timeCounter, CR,
    "tenths of a second.", CR, CR

END
```

Your Turn – A Faster Circuit

By using a capacitor that has roughly 1/3 the capacity to hold charge, the time measurement for each resistor value that is used in the circuit will be reduced by 1/3. In Activity #3, you will use a capacitor that is 10,000 times smaller, and the BASIC Stamp will still take the time measurements for you using a command called **RCTIME**.

- ✓ Disconnect power to your Board of Education or HomeWork Board.
- ✓ Replace the 3300 μF capacitor with a 1000 μF capacitor.
- ✓ **Confirm that the polarity of your capacitor is correct.** The negative terminal should be connected to Vss.
- ✓ Reconnect power.

- ✓ Repeat the steps in the Example Program: PolledRcTimer.bs2 section, and record your time measurements in Table 5-2.
- ✓ Compare your time measurements to the ones you took earlier in Table 5-1. How close are they to 1/3 the value of the measurements taken with the 3300 μF capacitor?

Table 5-2: Resistance and RC-time for C = 1000 μF	
Resistance (Ω)	Measured Time (s)
470	
1 k	
2 k	
10 k	

ACTIVITY #3: READING THE DIAL WITH THE BASIC STAMP

In Activity #1, a potentiometer was used as a variable resistor. The resistance in the circuit varied depending on the position of the potentiometer's adjusting knob. In Activity #2, an RC-time circuit was used to measure different resistances. In this activity, you will build an RC-time circuit to read the potentiometer, and use the BASIC Stamp to take the time measurements. The capacitor you use will be very small, and the time measurements will only take a few milliseconds. Even though the measurements take very short durations of time, the BASIC Stamp will give you an excellent indication of the resistance between the potentiometer's A and W terminals.

Parts for Reading RC-Time with the BASIC Stamp

- (1) Potentiometer – 10 k Ω
- (1) Resistor – 220 Ω (red-red-brown)
- (2) Jumper wires
- (1) Capacitor – 0.1 μF shown in Figure 5-10
- (1) Capacitor – 0.01 μF , also shown in Figure 5-10
- (2) Jumper wires



These capacitors do not have + and – terminals. You can safely connect these capacitors to a circuit without worrying about positive and negative terminals.

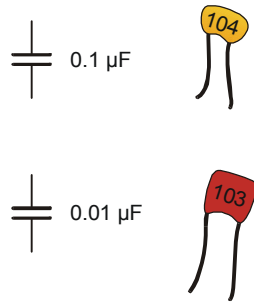


Figure 5-10
Ceramic Capacitors

The 0.1 μ F capacitor (above) and the 0.01 μ F capacitor (below) are both non-polar. You will not have to worry about positive and negative leads with these two parts.

Building an RC Time Circuit for the BASIC Stamp

Figure 5-11 shows a schematic for the fast RC-time circuit, and Figure 5-12 shows the wiring diagram. This is the circuit that you will use to monitor the position of the potentiometer's input shaft with the help of the BASIC Stamp and a PBASIC program.

√ Build the circuit shown in Figure 5-11.

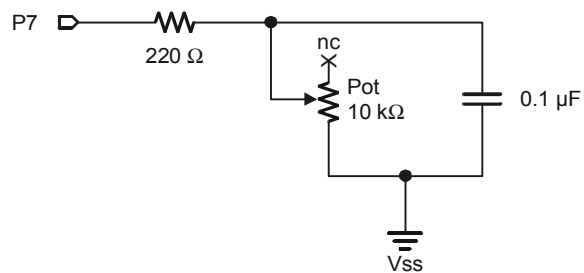


Figure 5-11
BASIC Stamp
RCTIME Circuit
with Potentiometer

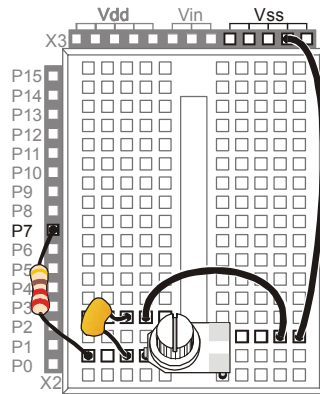


Figure 5-12
Wiring Diagram for
Figure 5-11

5

Programming RC-Time Measurements

The BASIC Stamp program to measure the potentiometer's resistance will do essentially the same thing that you did by hand in Activity #2. The equivalent of pressing and holding the pushbutton is a **HIGH** command followed by a **PAUSE**. The **RCTIME** command is the BASIC Stamp module's way of letting go of the pushbutton and polling until the capacitor's voltage gets low enough to pass the threshold voltage of **IN7** (1.4 V).

Example Program: ReadPotWithRcTime.bs2

- ✓ Enter and run ReadPotWithRcTime.bs2
- ✓ Try rotating the potentiometer's input shaft while monitoring the value of the **time** variable using the Debug Terminal.

```
' What's a Microcontroller - ReadPotWithRcTime.bs2
' Read potentiometer in RC-time circuit using RCTIME command.

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

time VAR Word

DO

  HIGH 7
  PAUSE 100
  RCTIME 7, 1, time
  DEBUG HOME, "time = ", DEC5 time

LOOP
```

How ReadPotWithRcTime.bs2 Works

Here are the pseudo-code steps the program goes through to take the RC-time measurement.

- *Declare the variable `time` to store a time measurement.*
- *Code block within `DO...LOOP`:*
 - *Set I/O pin P7 to `HIGH`.*
 - *Wait for 100 ms (20 ms to make sure the capacitor is charged up and 80 more ms to keep the Debug Terminal display steady).*
 - *Execute the `RCTIME` command.*
 - *Store the time measurement in the `time` variable.*
 - *Display the value `time` in the Debug Terminal.*

Before the `RCTIME` command is executed, the capacitor is fully charged. As soon as the `RCTIME` command executes, the BASIC Stamp changes the I/O pin from an output to an input. As an input, the I/O pin looks about the same to the circuit as when the pushbutton was released (open circuit) in Activity #2. The `RCTIME` command is a high speed version of the polling that was used in Activity #2, and it measures the amount of time it takes for the capacitor to lose its charge and fall below the I/O pin's 1.4 V input threshold. Instead of counting in 100 ms increments, the `RCTIME` command counts in 2 μ s increments.

Your Turn – Changing Time by Changing the Capacitor

- ✓ Replace the 0.1 μ F capacitor with a 0.01 μ F capacitor.
- ✓ Try the same positions on the potentiometer that you did in the main activity and compare the value displayed in the Debug Terminal with the values obtained for the 0.1 μ F capacitor. Are the `RCTIME` measurements one tenth the value?
- ✓ Go back to the 0.1 μ F capacitor.
- ✓ With the 0.1 μ F capacitor back in the circuit and the 0.01 μ F capacitor removed, make a note of the highest and lowest values for the next activity.

ACTIVITY #4: CONTROLLING A SERVO WITH A POTENTIOMETER

Potentiometers together with servos can be used to make lots of fun things. This is the foundation for model airplanes, cars and boats. This activity shows how the BASIC Stamp can be used to monitor a potentiometer circuit and control the position of a servo.

An example of a model airplane and its radio controller are shown in Figure 5-13. The model airplane has servos to control all its flaps and the gas engine's throttle settings. These servos are controlled using the radio control (RC) unit in front of the plane. This RC unit has potentiometers under a pair of joysticks that are used to control the servos that in turn control the plane's elevator and rudder flaps.



5

Figure 5-13
Model Airplane and
Radio Controller



How the RC Unit Controls the Airplane: The potentiometers under the joysticks are monitored by a circuit that converts the position of the joystick into control pulses for the servo. These control pulses are then converted to radio signals and transmitted by the handheld controller to a radio receiver in the model airplane. The radio receiver converts these signals back to control pulses which then position the servos.

Potentiometer Controlled Servo Parts

- (1) Potentiometer – 10 k Ω
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Capacitor – 0.1 μ F
- (1) Parallax Standard Servo
- (1) LED – any color
- (2) Jumper wires

HomeWork Board users will also need:

- (1) 3-pin male-male header

(4) Jumper wires



CAUTION: use only a Parallax Standard Servo for the activities in this text!

Do not substitute a Parallax Continuous Rotation Servo, as it may be quickly damaged by the circuits shown below. Likewise, we do not recommend using other brands of standard hobby servos, which may not be rated for use with the voltage supplied in these circuits.

Building the Dial and Servo Circuits

This activity will use two circuits that you have already built individually: the potentiometer circuit from the activity you just finished and the servo circuit from the previous chapter.

- √ Leave your potentiometer RC-time circuit from Activity #3 on your prototyping area. If you need to rebuild it, use Figure 5-11 on page 150 and Figure 5-12 on page 151. Make sure to use the 0.1 μ F capacitor, not the 0.01 μ F capacitor.
- √ Add your servo circuit from Chapter #4, Activity #1 to the project. Remember that your servo circuit will be different depending on your carrier board. Below are the pages for the sections that you will need to jump to:
 - Page 105 – Board of Education Rev C
 - Page 111 – Board of Education Rev B
 - Page 108 – BASIC Stamp HomeWork Board

Programming Potentiometer Control of the Servo

You will need the smallest and largest value of the `time` variable that you recorded from your RC-time circuit while using a 0.1 μ F capacitor.

- √ If you have not already completed the Your Turn section of the previous activity, go back and complete it now.

For this next example, here are the `time` values that were measured by a Parallax technician; your values will probably be slightly different:

- All the way clockwise: 1
- All the way counterclockwise: 691

So how can these input values be adjusted so that they map to the values of 500 and 1000 that are needed to control the servo with the `PULSOUT` command? The answer is by using multiplication and addition. First, multiply the input values by something to make the difference between the clockwise (minimum) and counterclockwise (maximum) values 500 instead of almost 700. Then, add a constant value to the result so that its range is from 500 to 1000 instead of 1 to 500. In electronics, these operations are called scaling and offset.

Here's how the math works for the multiplication (scaling):

$$time(maximum) = 691 \times \frac{500}{691} = 691 \times 0.724 = 500$$

$$time(minimum) = 1 \times \frac{500}{691} = 0.724$$

After the values are scaled, here is the addition (offset) step.

$$time(maximum) = 500 + 500 = 1000$$

$$time(minimum) = 0.724 + 500 = 500$$

The `*/` operator that was introduced on page 95 is built into PBASIC for scaling by fractional values, like 0.724. Here again are the steps for using `*/` applied to 0.724:

1. Place the value or variable you want to multiply by a fractional value before the `*/` operator.

```
time = time */
```

2. Take the fractional value that you want to use and multiply it by 256.

$$new\ fractional\ value = 0.724 \times 256 = 185.344$$

3. Round off to get rid of anything to the right of the decimal point.

$$new\ fractional\ value = 185$$

4. Place that value after the `*/` operator.

```
time = time */ 185
```

That takes care of the scaling, now all we need to do is add the offset of 500. This can be done with a second command that adds 500 to `time`:

```
time = time */ 185
time = time + 500
```

Now, `time` is ready to be recycled into the `PULSOUT` command's *Duration* argument.

```
time = time */ 185      ' Scale by 0.724.
time = time + 500      ' Offset by 500.
PULSOUT 14, time        ' Send pulse to servo.
```

Example Program: ControlServoWithPot.bs2

- ✓ Enter and run this program, then twist the potentiometer's input shaft and make sure that the servo's movements echo the potentiometer's movements.

```
' What's a Microcontroller - ControlServoWithPot.bs2
' Read potentiometer in RC-time circuit using RCTIME command.
' Scale time by 0.724 and offset by 500 for the servo.

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

DEBUG "Program Running!"

time          VAR      Word

DO
  HIGH 7
  PAUSE 10
  RCTIME 7, 1, time
  time = time */ 185      ' Scale by 0.724 (X 256 for */).
  time = time + 500      ' Offset by 500.
  PULSOUT 14, time        ' Send pulse to servo.
LOOP
```

Your Turn – Scaling the Servo's Relationship to the Dial

Your potentiometer and capacitor will probably give you `time` values that are somewhat different from the ones discussed in this activity. These are the values you gathered in the Your Turn section of the previous activity.

- ✓ Repeat the math discussed in the Programming Potentiometer Control of the Servo section on page 154 using your maximum and minimum values.

- ✓ Substitute your **scale** and **offset** values in `ControlServoWithPot.bs2`.
- ✓ Add this line of code between the **PULSOUT** and **LOOP** commands so that you can view your results.


```
DEBUG HOME, DEC5 time    ' Display adjusted time value.
```
- ✓ Run the modified program and check your work. Because the values were rounded off, the limits may not be exactly 500 and 1000, but they should be pretty close.

Using Constants in Programs

In larger programs, you may end up using the **PULSOUT** command and the value of the scale factor (which was 185) and the offset (which was 500) many times in the program. You can use alias names for these values with the **CON** directive like this:

```
scaleFactor CON 185
offset CON 500
```



These alias names are just about always declared near the beginning of the program so that they are easy to find.

Now, anywhere in your program that you want to use one of these values, you can use the words **offset** or **scaleFactor** instead. For example,

```
time = time */ scaleFactor          ' Scale by 0.724.
time = time + offset                ' Offset by 500.
```

You can also apply the same technique with the I/O pins. For example, you can declare a constant for I/O pin P7.

```
rcPin    CON 7
```

There are two places in the previous example program where the number 7 is used to refer to I/O pin P7. The first can now be written as:

```
HIGH rcPin
```

The second can be written as:

```
RCTIME rcPin, 1, time
```

If you change your circuit later, all you have to do is change the value in your constant declaration, and both the **HIGH** and **RCTIME** commands will be automatically updated.

Likewise, if you have to recalibrate your scale factor or offset, you can also just change the **CON** directives at the beginning of the program.



Assigning an alias is what you do when you give a variable, constant or I/O pin a name using **VAR**, **CON**, or **PIN**.

Example Program: ControlServoWithPotUsingConstants.bs2

This program makes use of aliases in place of almost all numbers.

- ✓ Enter and run ControlServoWithPotUsingConstants.bs2.
- ✓ Observe how the servo responds to the potentiometer and verify that it's the same as the previous example program (ControlServoWithPot.bs2).

```
' What's a Microcontroller - ControlServoWithPotUsingConstants.bs2
' Read potentiometer in RC-time circuit using RCTIME command.
' Apply scale factor and offset, then send value to servo.

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

scaleFactor    CON      185
offset         CON      500
rcPin          CON      7
delay          CON      10
servoPin       CON      14

time           VAR      Word

DO
  HIGH rcPin
  PAUSE delay
  RCTIME rcPin, 1, time
  time = time */ scaleFactor          ' Scale scaleFactor.
  time = time + offset              ' Offset by offset.
  PULSOUT servoPin, time            ' Send pulse to servo.
  DEBUG HOME, DEC5 time             ' Display adjusted time value.
LOOP
```

Your Turn – Using Constants for Calibration and Easy Updating

As mentioned earlier, if you change the I/O pin used by the **HIGH** and **RCTIME** command, you can simply change the value of the **rcPin** constant declaration.

- ✓ Save the example program under a new name.
- ✓ Change the **scaleFactor** and **offset** values to the unique values for your RC circuit that you determined in the previous Your Turn section.
- ✓ Run the modified program and verify that it works correctly.
- ✓ Modify your circuit by moving the RC-time circuit from I/O pin P7 to I/O pin P8.
- ✓ Modify the **rcPin** declaration so that it reads:

```
rcPin    CON 8
```

- ✓ Add this command before the **DO...LOOP** so that you can see that the **rcPin** constant really is just a way of saying the number eight:

```
DEBUG ? rcPin
```

- ✓ Re-run the program and verify that the **HIGH** and **RCTIME** commands are still functioning properly on the different I/O pin with just one change to the **rcPin** **CON** directive.

SUMMARY

This chapter introduced the potentiometer, a part often found under various knobs and dials. The potentiometer has a resistive element that typically connects its outer two terminals and a wiper terminal that contacts a variable point on the resistive element. The potentiometer can be used as a variable resistor if the wiper terminal and one of the two outer terminals is used in a circuit.

The capacitor was also introduced in this chapter. A capacitor can be used to store and release charge. The amount of charge a capacitor can store is related to its value, which is measured in Farads, (F). The μ is engineering notation for micro, and it means one-millionth. The capacitors used in this chapter's activities ranged from 0.01 to 3300 μ F.

A resistor and a capacitor can be connected together in a circuit that takes a certain amount of time to charge and discharge. This circuit is commonly referred to as an RC-time circuit. The R and C in RC-time stand for resistor and capacitor. When one value (C in this chapter's activities) is held constant, the change in the time it takes for the circuit to discharge is related to the value of R. When the value of R changes, the value of the time it takes for the circuit to charge and discharge also changes. The overall time it takes the RC-time circuit to discharge can be scaled by using a capacitor of a different size.

Polling was used to monitor the discharge time of a capacitor in an RC circuit where the value of C was very large. Several different resistors were used to show how the discharge time changes as the value of the resistor in the circuit changes. The **RCTIME** command was then used to monitor a potentiometer (a variable resistor) in an RC-time circuit with smaller value capacitors. Although these capacitors cause the discharge times to range from roughly 2 to 1500 μ s (millionths of a second), the BASIC Stamp has no problem tracking these time measurements with the **RCTIME** command. The I/O pin must be set **HIGH**, and then the capacitor in the RC-time circuit must be allowed to charge by using **PAUSE** before the **RCTIME** command can be used.

PBASIC programming can be used to measure a resistive sensor such as a potentiometer and scale its value so that it is useful to another device, such as a servo. This involves performing mathematical operations on the measured RC discharge time, which the **RCTIME** command stores in a variable. This variable can be adjusted by adding a constant value to it, which comes in handy for controlling a servo. In the Projects section, you

may find yourself using multiplication and division as well. The `CON` directive can be used at the beginning of a program to substitute a name for a number. As with naming variables, naming constants is also called creating an alias. After an alias is created, the name can be used in place of the number throughout the program. This can come in handy, especially if you need to use the same number in 2, 3, or even 100 different places in the program. You can change the number in the `CON` directive, and all 2, 3, or even 100 different instances of that number are automatically updated next time you run the program.

Questions

1. When you turn the dial or knob on a sound system, what component are you most likely adjusting?
2. In a typical potentiometer, is the resistance between the two outer terminals adjustable?
3. How is a capacitor like a rechargeable battery? How is it different?
4. What can you do with an RC-time circuit to give you an indication of the value of a variable resistor?
5. What happens to the RC discharge time as the value of R (the resistor) gets larger or smaller?
6. What does the `CON` directive do? Explain this in terms of a name and a number.

Exercise

1. Let's say that you have a 0.5 μ F capacitor in an RC timer circuit, and you want the measurement to take 10-times as long. Calculate the value of the new capacitor.

Projects

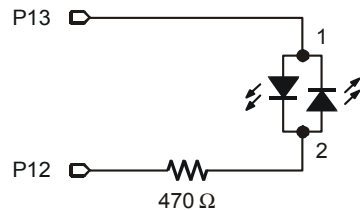
1. Add a bi-color LED circuit to Activity #4. Modify the example program so that the bi-color LED is red when the servo is rotating counterclockwise, green when the servo is rotating clockwise, and off when the servo holding its position.
2. Use `IF...THEN` to modify the example program from Activity #4 so that the servo only rotates between `PULSOUT` values of 650 and 850.

Solutions

- Q1. A potentiometer.

- Q2. No, it's fixed. The variable resistance is between either outer terminal and the wiper (middle) terminal.
- Q3. A capacitor is like a rechargeable battery in that it can be charged up to hold voltage. The difference is that it only holds a charge for a very small amount of time.
- Q4. You can measure the time it takes for the capacitor to discharge (or charge) This time is related to the resistance and capacitance. If the capacitance is known and the resistance is variable, then the discharge time gives an indication of the resistance.
- Q5. As R gets larger, the RC discharge time increases in direct proportion to the increase in R. As R gets smaller, the RC discharge time decreases in direct proportion to the decrease in R.
- Q6. The **CON** directive substitutes a name for a number.
- E1. new cap = 10 x old cap value
 $= 10 \times 0.5\mu\text{F}$
 $= 5\mu\text{F}$

P1. Activity #4 with bi-color LED added.



Potentiometer schematic from Figure 5-11 p. 150, servo from Figure 4-3 p. 106, bi-color LED from Figure 2-19 p. 63, with P15 and P14 changed to P13 and P12.

```
' What's a Microcontroller - Ch5Prj01_ControlServoWithPot.bs2
' Read potentiometer in RC-time circuit using RCTIME command.
' The time variable ranges from 126 to 713, and an offset of 330 is
' needed.
' Bi-color LED on P12, P13 tells direction of servo rotation:
' green for CW, red for CCW, off when servo is holding position.

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

time          VAR      Word      ' time reading from
potentiometer
prevTime      VAR      Word      ' previous reading

DO
```



```

prevTime = time           ' Store previous time reading
HIGH 7                    ' Read pot using RCTIME
PAUSE 10
RCTIME 7, 1, time
time = time + 330         ' Scale pot, match servo range

IF ( time > prevTime + 2) THEN ' increased, pot turned CCW
  HIGH 13                 ' Bi-color LED red
  LOW 12
ELSEIF ( time < prevTime - 2) THEN ' value decreased, pot turned CW
  LOW 13                 ' Bi-color LED green
  HIGH 12
ELSE                       ' Servo holding position
  LOW 13                 ' LED off
  LOW 12
ENDIF

PULSOUT 14, time
LOOP

```

5

P2. The key is to add **IF...THEN** blocks; an example is shown below.

```

' What's a Microcontroller - Ch5Prj02_ControlServoWithPot.bs2
' Read potentiometer in RC-time circuit using RCTIME command.
' The time variable ranges from 126 to 713, and an offset of 330 is
' needed.
' Modify so the servo only rotates from 650 to 850.
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Program Running!"

time VAR Word

DO
  HIGH 7                    ' Read pot with RCTIME
  PAUSE 10
  RCTIME 7, 1, time

  time = time + 330         ' Scale time to servo range

  IF (time < 650) THEN      ' Constrain range from 650 to 850
    time = 650
  ENDIF
  IF (time > 850) THEN
    time = 850
  ENDIF

  PULSOUT 14, time
LOOP

```

Further Investigation

Several different electronic components, concepts and techniques were incorporated in this chapter. Some of the more notable examples are:

- Using a potentiometer as an input device
- Measuring the resistance/capacitance of a device using **RCTIME**
- Performing math on an input value and recycling it into an output
- Controlling a motor based on a measured value

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

“Robotics with the Boe-Bot”, Student Workbook, Version 2.0, Parallax Inc., 2003

“SumoBot”, Student Workbook, Version 1.1, Parallax Inc., 2002

Every Stamp in Class robotics text uses **RCTIME** to measure resistive sensors to detect a variety of conditions. Each condition leads to math and decisions, and the end result is robot movement.

“Basic Analog and Digital”, Student Guide, Version 2.0, Parallax Inc., 2003

Basic Analog and Digital uses the potentiometer to create a variable voltage, called a voltage divider, which is analyzed by an analog to digital converter. A potentiometer is also used as an input device to set the frequency of a 555 timer. This text takes a closer look at the math involved in RC voltage decay.

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

RCTIME is used extensively in this book to collect data from a variety of sensors.

“Industrial Control”, Student Guide, Version 2.0, Parallax Inc., 2002

This book introduces techniques used extensively in industry for controlling machines based on sensor input. The techniques fall under the general category of control systems.

Chapter #6: Digital Display

THE EVERY-DAY DIGITAL DISPLAY

Figure 6-1 shows a display on the front of an oven door. When the oven is not in use, it displays the time. When the oven is in use, it displays the oven's timer, cooking settings, and it flashes on and off at the same time an alarm sounds to let you know the food is done. A microcontroller inside the oven door monitors the pushbuttons and updates the display. It also monitors sensors inside the oven and switches devices that turn the heating elements on and off.



Figure 6-1
Digital Clock 7-Segment
Display on Oven Door

6

Each of the three digits in Figure 6-1 is called a 7-segment display. In this chapter, you will program the BASIC Stamp to display numbers and letters on a 7-segment display.

WHAT'S A 7-SEGMENT DISPLAY?

A 7-segment display is rectangular block of 7 lines of equal length that can be lit selectively to display digits and some letters. A very common form is the 7-segment LED display, a package with a rectangular block of 7 LEDs. Figure 6-2 shows a part drawing of the 7-segment LED display you will use in this chapter's activities. It has one additional LED, a dot that can be used as a decimal point. Each of the segments (A through G) and the dot contains a separate LED, which can be controlled individually. Most of the pins have a number along with a label that corresponds with one of the LED segments. Pin 5 is labeled DP, which stands for decimal point. Pins 3 and 8 are labeled "common cathode", and they will be explained when the schematic for this part is introduced.

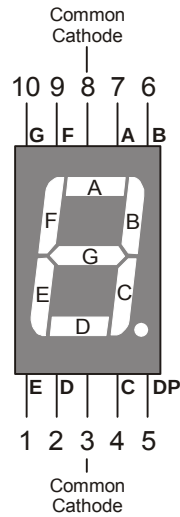


Figure 6-2
7-Segment
LED Display
Part Drawing
and Pin Map



Pin Map: Figure 6-2 is an example of a pin map. A pin map contains useful information that helps you connect a part to other circuits. Pin maps usually show a number for each pin, a name for each pin, and a reference. Take a look at Figure 6-2. Each pin is numbered, and the name for each pin is the segment letter next to the pin. The reference for this part is its overall appearance. You know by looking at the top of the display that pin 1 is closest to the lower-left corner of the display. Other parts have more subtle references, such as the flat spot on a regular LED's case.

Figure 6-3 shows a schematic of the LEDs inside the 7-segment LED display. Each LED anode is connected to an individual pin. All the cathodes are connected together by wire inside the part. Because all the cathodes share a common connection, the 7-segment LED display can be called a “common cathode” display. By connecting either pin 3 or pin 8 of the part to Vss, you will connect all the LED cathodes to Vss.

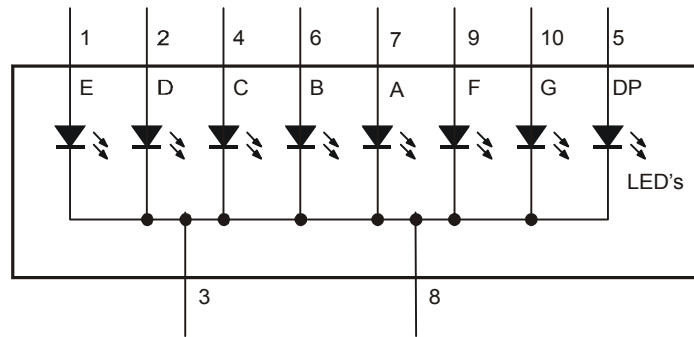


Figure 6-3
7-Segment
Schematic

6

ACTIVITY #1: BUILDING AND TESTING THE 7-SEGMENT LED DISPLAY

In this activity, you will manually build circuits to test each segment in the display.

7-Segment LED Display Test Parts

- (1) 7-segment LED display
- (5) Resistors – 1 k Ω (brown-black-red)
- (5) Jumper wires

7-Segment LED Display Test Circuits

- ✓ With power disconnected from your Board of Education or HomeWork Board, build the circuit shown in Figure 6-4 and Figure 6-5.
- ✓ Reconnect power and verify that the A segment emits light.



What's the x with the nc above it in the schematic? The nc stands for not connected or no-connect. It indicates that a particular pin on the 7-segment LED display is not connected to anything. The x at the end of the pin also means not connected. Schematics sometimes use just the x or just the nc.

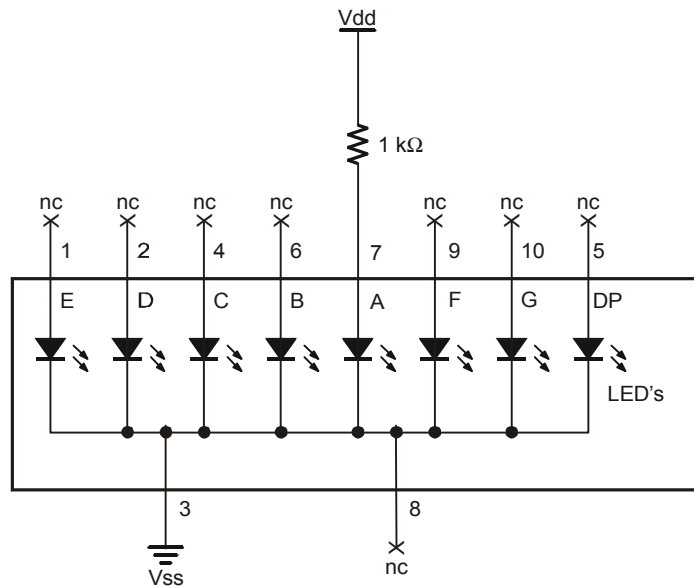


Figure 6-4
Test Circuit
Schematic for
the 'A'
Segment LED
Display.

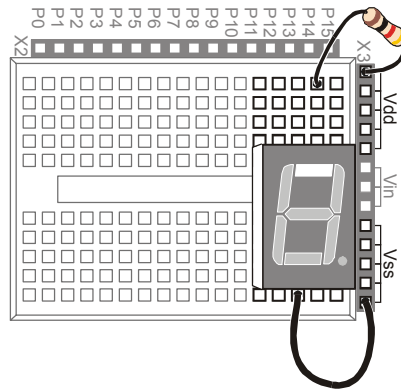


Figure 6-5
Test Circuit
Wiring Diagram
for the 'A'
Segment LED
Display

- √ Disconnect power, and modify the circuit by connecting the resistor to the B LED input as shown in Figure 6-6 and Figure 6-7.

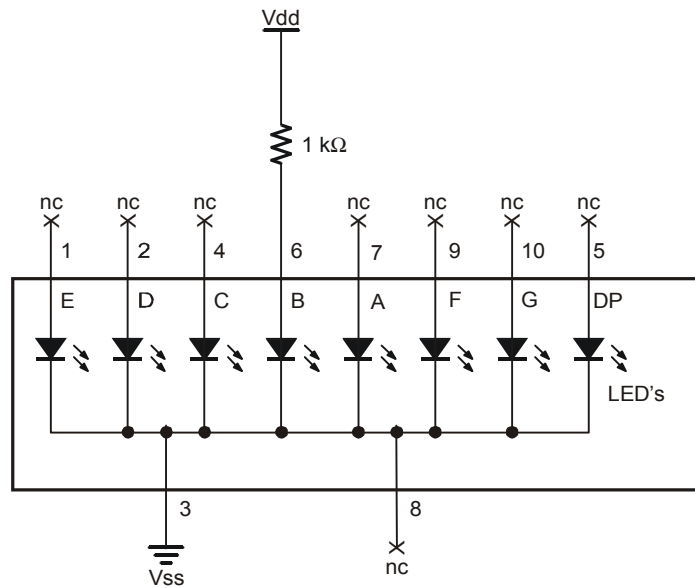


Figure 6-6
Test Circuit
Schematic for
the 'B' Segment
LED Display.

6

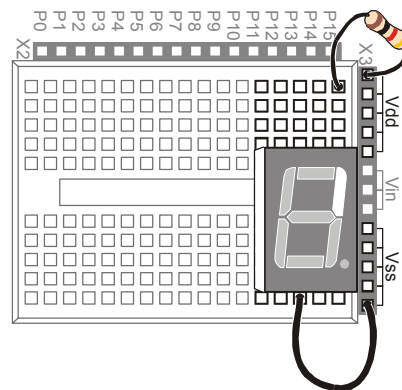


Figure 6-7
Test Circuit
Wiring Diagram
for the 'B'
Segment LED
Display

- ✓ Reconnect power and verify that the B segment emits light.
- ✓ Using the pin map from Figure 6-2 as a guide, repeat these steps for segments C through G.

Your Turn – The Letter A and the Number Two

Figure 6-8 and Figure 6-9 show the digit '3' hardwired into the 7-segment LED display.

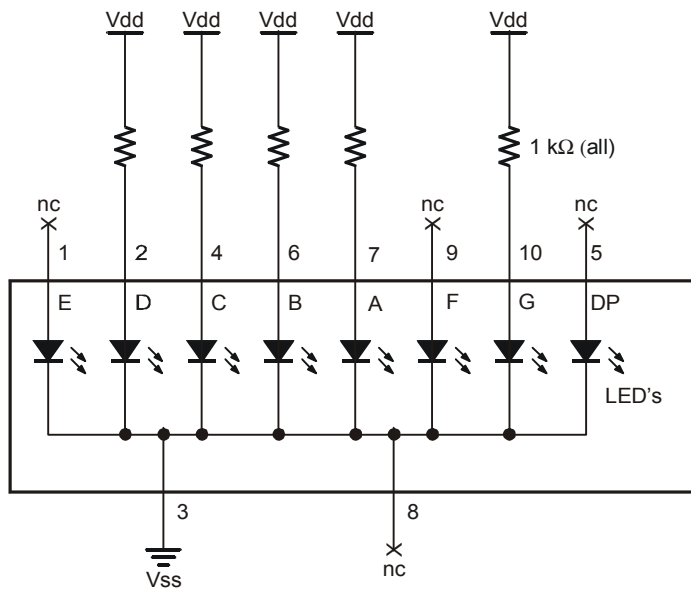


Figure 6-8
Hardwired Digit '3'

The digit "3" is shown on the 7-segment LED display using the circuit shown in this schematic.

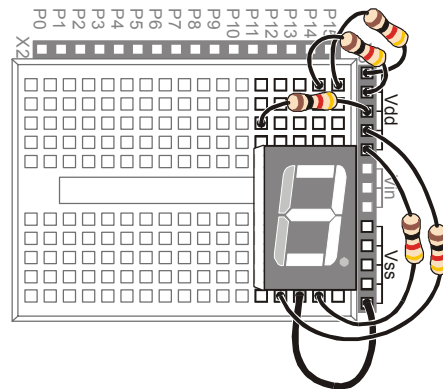


Figure 6-9
Wiring Diagram for
Figure 6-8

- ✓ Build and test the circuit shown in Figure 6-8 and Figure 6-9, and verify that it displays the number three.
- ✓ Draw a schematic that will display the number two on the 7-segment LED.
- ✓ Build and test the circuit to make sure it works. Trouble-shoot if necessary.
- ✓ Repeat for the letter 'A'.

ACTIVITY #2: CONTROLLING THE 7-SEGMENT LED DISPLAY

In this activity, you will connect the 7-segment LED display to the BASIC Stamp, and then run a simple program to test and make sure each LED is properly connected.

7-Segment LED Display Parts

- (1) 7-segment LED display
- (8) Resistors – 1 k Ω (brown-black-red)
- (5) Jumper wires

Connecting the 7-Segment LED Display to the BASIC Stamp

Figure 6-10 shows the schematic and Figure 6-11 shows the wiring diagram for this BASIC Stamp controlled 7-segment LED display example.

6

✓ Build the circuit shown in Figure 6-10 and Figure 6-11.



Schematic and pin map: If you are trying to build the circuit from the schematic in Figure 6-10 without relying on Figure 6-11, make sure to consult the 7-segment LED display's pin map (Figure 6-2, page 166).

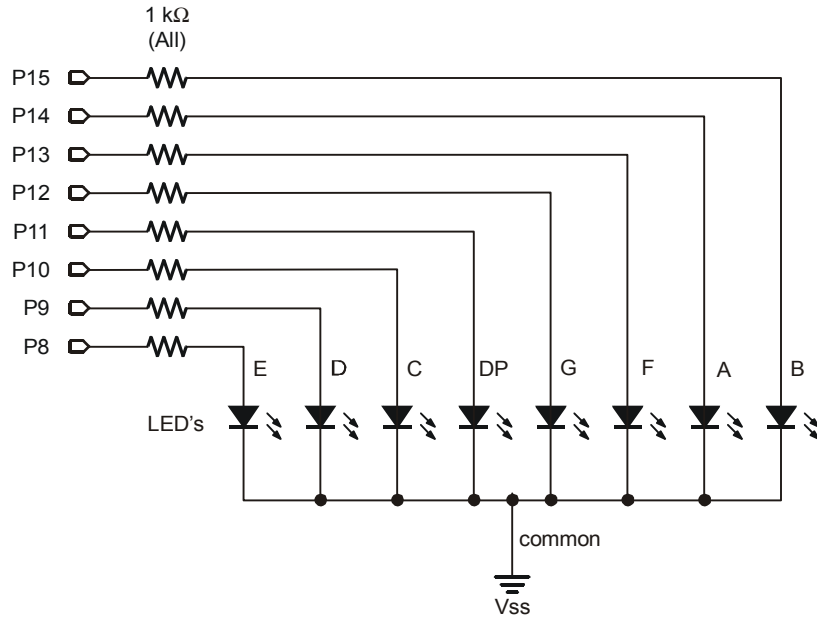


Figure 6-10
BASIC Stamp
Controlled 7-
Segment LED
Display
Schematic



Be careful with the resistors connected to P13 and P14. Look carefully at the resistors connected to P13 and P14 in Figure 6-11. There is gap between these two resistors. The gap is shown because pin 8 on the 7-segment LED display is left unconnected. A resistor connects I/O pin P13 to 7-segment LED display pin 9. Another resistor connects P14 to 7-segment LED display pin 7.

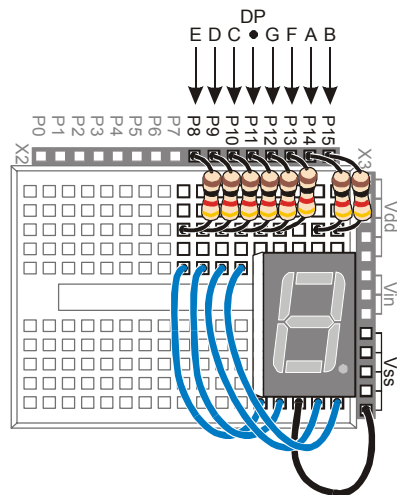


Figure 6-11
Wiring Diagram for
Figure 6-10

Use the segment letters above this diagram as a reference.

6



Parallel Device: The 7-segment LED display is called a parallel device because you have to use more than one I/O line at a time to send data (high and low information) to the device. In the case of this 7-segment LED display, it takes 8 I/O pins to instruct the device what to do.

Parallel Bus: The wires that transmit the **HIGH/LOW** signals from the BASIC Stamp to the 7-segment LED display are called a parallel bus. Note that these wires are drawn as parallel lines in Figure 6-10. The term parallel kind of makes sense given the geometry of the schematic.

Programming the 7-Segment LED Display Test

The **HIGH** and **LOW** commands will accept a variable as a pin argument. To test each segment, one at a time, simply place the **HIGH** and **LOW** commands in a **FOR...NEXT** loop, and use the index to set the I/O pin, high, then low again.

- ✓ Enter and run `SegmentTestWithHighLow.bs2`.
- ✓ Verify that every segment in the 7-segment LED display lights briefly, turning on and then off again.
- ✓ Record a list of which segment each I/O pin controls.

Example Program: SegmentTestWithHighLow.bs2

```
' What's a Microcontroller - SegmentTestWithHighLow.bs2
' Individually test each segment in a 7-Segment LED display.
```

```

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

pinCounter    VAR    Nib

DEBUG "I/O Pin", CR,
      "-----", CR

FOR pinCounter = 8 TO 15

    DEBUG DEC2 pinCounter, CR
    HIGH pinCounter
    PAUSE 1000
    LOW pinCounter

NEXT

```

Your Turn – A Different Pattern

Removing the command **LOW pinCounter** will have an interesting effect:

- ✓ Comment the **LOW pinCounter** command by adding an apostrophe to the left of it.
- ✓ Run the modified program and observe the effect.

ACTIVITY #3: DISPLAYING DIGITS

If you include the decimal point there are eight different BASIC Stamp I/O pins that send high/low signals to the 7-segment LED display. That's eight different **HIGH** or **LOW** commands just to display one number. If you want to count from zero to nine, that would be a huge amount of programming. Fortunately, there are special variables you can use to set the high and low values for groups of I/O pins.

In this activity, you will use 8-digit binary numbers instead of **HIGH** and **LOW** commands to control the high/low signals sent by BASIC Stamp I/O pins. By setting special variables called **DIRH** and **OUTH** equal to the binary numbers, you will be able to control the high/low signals sent by all the I/O pins connected to the 7-segment LED display circuit with a single PBASIC command.



8 bits: A binary number that has 8 digits is said to have 8 bits. Each bit is a slot where you can store either a 1 or a 0.

A byte is a variable that contains 8 bits. There are 256 different combinations of zeros and ones that you can use to count from 0 to 255 using 8 bits. This is why a byte variable can store a number between 0 and 255.

Parts and Circuit for Displaying Digits

Same as previous activity

Programming On/Off Patterns Using Binary Numbers

6

In this activity, you will experiment with the variables **DIRH** and **OUTH**. **DIRH** is a variable that controls the direction (input or output) of I/O pins P8 through P15. **OUTH** controls the high or low signals that each I/O pin sends. As you will soon see, **OUTH** is especially useful because you can use it to set the high/low signals at eight different I/O pins at once with just one command. Here is an example program that shows how these two variables can be used to count from 0 to 9 on the 7-segment LED display without using **HIGH** and **LOW** commands:

Example Program: DisplayDigits.bs2

This example program will cycle the 7-Segment LED display through the digits 0 through 9.

- ✓ Enter and run DisplayDigits.bs2.
- ✓ Verify that the digits 0 through 9 are displayed.

```
' What's a Microcontroller - DisplayDigits.bs2
' Display the digits 0 through 9 on a 7-segment LED display.

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

DEBUG "Program Running!"

OUTH = %00000000      ' OUTH initialized to low.
DIRH = %11111111      ' Set P8-P15 to all output-low.

'          BAFG.CDE    ' Digit:
OUTH = %11100111      ' 0
PAUSE 1000
OUTH = %10000100      ' 1
```

```

PAUSE 1000
OUTH = %11010011          ' 2
PAUSE 1000
OUTH = %11010110          ' 3
PAUSE 1000
OUTH = %10110100          ' 4
PAUSE 1000
OUTH = %01110110          ' 5
PAUSE 1000
OUTH = %01110111          ' 6
PAUSE 1000
OUTH = %11000100          ' 7
PAUSE 1000
OUTH = %11110111          ' 8
PAUSE 1000
OUTH = %11110110          ' 9
PAUSE 1000

DIRH = %00000000          ' I/O pins to input,
                           ' segments off.

END

```

How DisplayDigits.bs2 Works

Figure 6-12 shows how you can use the **DIRH** and **OUTH** variables to control the direction and state (high/low) of I/O pins P8 through P15.

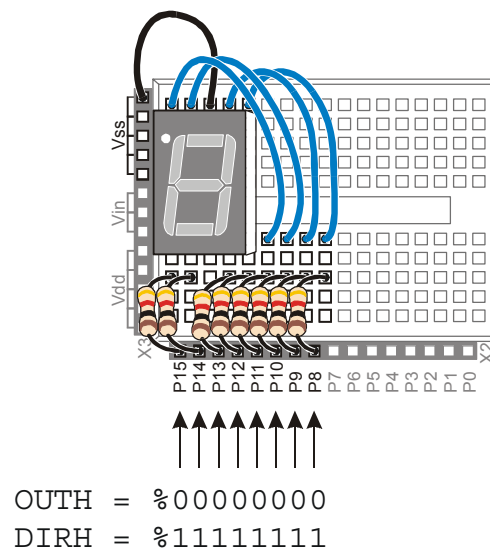


Figure 6-12
Using DIRH
and OUTH to
set all I/O Pins
to Output-Low

The first command:

```
OUTH = %00000000
```

gets all the I/O pins (P8 through P15) ready to send the low signals. If they all send low signals, it will turn all the LEDs in the 7-segment LED display off. If you wanted all the I/O pins to send the high signal, you could use **OUTH = %11111111** instead.



What does % do? The % is used to tell the BASIC Stamp Editor that the number is a binary number. For example, the binary number %00001100 is the same as the decimal number 12. As you will see in this activity, binary numbers can make many programming tasks much easier.

6

The low signals will not actually be sent by the I/O pins until you use the **DIRH** variable to change all the I/O pins from input to output. The command:

```
DIRH = %11111111
```

sets all I/O pins P8 through P15 to output. As soon as this command is executed, P8 through P15 all start sending the low signal. This is because the command **OUTH = %00000000** was executed just before this **DIRH** command. As soon as the **DIRH** command set all the I/O pins to output, they started sending their low signals. You can also use **DIRH = %00000000** to change all the I/O pins back to inputs.



Before I/O pins become outputs: Up until the I/O pins are changed from input to output, they just listen for signals and update the **INH** variable. This is the variable that contains **IN8**, **IN9**, up through **IN15**. These variables can be used the same way that **IN3** and **IN4** were used for reading pushbuttons in Chapter #3.

All BASIC Stamp I/O pins start out as inputs. This is called a “default”. You have to tell a BASIC Stamp I/O pin to become an output before it starts sending a high or low signal. Both the **HIGH** and **LOW** commands automatically change a BASIC Stamp I/O pin’s direction to output. Placing a 1 in the **DIRH** variable also makes one of the I/O pins an output.

Figure 6-13 shows how to use the **OUTH** variable to selectively send high and low signals to P8 through P15. A binary-1 is used to send a high signal, and a binary-0 is used to send a low signal. This example displays the number three on the 7-segment LED display:

```
'          BAFG.CDE
OUTH = %11010110
```

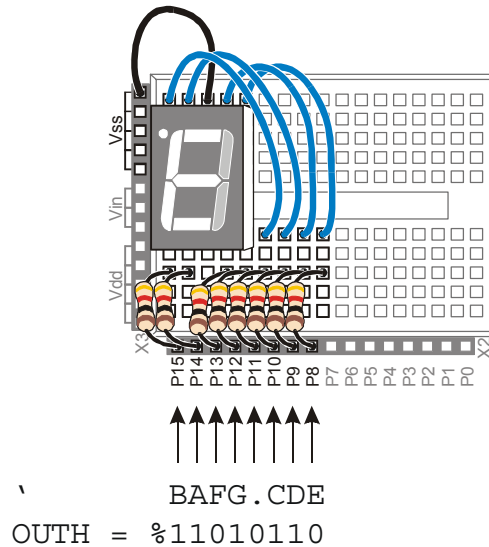


Figure 6-13
Using OUTH to
Control the
High/Low Signals
of P8 – P15.

The display is turned so that the three on the display is upside-down because it more clearly shows how the values in **OUTH** line up with the I/O pins. The command **OUTH = %11010110** uses binary zeros to set I/O pins P8, P11, and P13 low, and it uses binary ones to set P9, P10, P12, P14 and P15 high. The line just before the command is a comment that shows the segment labels line up with the binary value that turns that segment on/off. The next example program shows how to set **OUTH** to binary numbers to make the 7-segment LED display count from zero to nine.



Inside the HIGH and LOW commands: The command **HIGH 15** is really the same as **OUT15 = 1** followed by **DIR15 = 1**. Likewise, the command **LOW 15** is the same as **OUT15 = 1** followed by **DIR15 = 1**. If you want to change P15 back to an input, use **DIR15 = 0**. You can then use **IN15** to detect (instead of send) high/low signals.

Your Turn – Displaying A through F

- ✓ Figure out what bit patterns (combinations of zeros and ones) you will need to display the letters A, b, C, d, E, and F.
- ✓ Modify SevenSegment0to9 so that it displays A, b, C, d, E, F.



Decimal vs. Hexadecimal The basic digits in the decimal (base-10) number system are: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. In the hexadecimal (base-16) number system the basic digits are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, b, C, d, E, F. Base-16 is used extensively in both computer and microcontroller programming. Once you figure out how to display the characters A through F, you can further modify your program to count in hexadecimal from 0 to F.

Keeping Lists of On/Off Patterns

The **LOOKUP** command makes writing code for 7-segment LED display patterns much easier. The **LOOKUP** command lets you “look up” elements in a list. Here is a code example that uses the **LOOKUP** command:

```
LOOKUP index, [7, 85, 19, 167, 28], value
```

There are two variables used in this command, **index** and **value**. If the **index** is 0, **value** stores the 7. If **index** is 1, the **value** stores 85. Since **index** is 2, in this example, the **LOOKUP** command places 19 into **value**, and that's what the Debug Terminal displays.

6

Example Program: SimpleLookup.bs2

- ✓ Enter and run SimpleLookup.bs2.
- ✓ Run the program as-is, with the **index** variable set equal to 2.
- ✓ Try setting the **index** variable equal to numbers between 0 and 4.
- ✓ Re-run the program after each change to the **index** variable and note which value from the list gets placed in the **value** variable.
- ✓ Optional: Modify the program by placing the **LOOKUP** command in a **FOR...NEXT** loop that counts from 0 to 4.

```
' What's a Microcontroller - SimpleLookup.bs2
' Debug a value using an index and a lookup table.

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

value          VAR      Byte
index          VAR      Nib

index = 2

DEBUG ? index

LOOKUP index, [7, 85, 19, 167, 28], value
```

```

DEBUG ? value, CR

DEBUG "Change the index variable to a ", CR,
      "different number(between 0 and 4).", CR, CR,

      "Run the modified program and ", CR,
      "check to see what number the", CR,
      "LOOKUP command places in the", CR,
      "value variable."

END

```

Example Program: DisplayDigitsWithLookup.bs2

This example program shows how the **LOOKUP** command can come in really handy for storing the bit patterns used in the **OUTH** variable. Again, the **index** is used to choose which binary value is placed into the **OUTH** variable. This example program counts from 0 to 9 again. The difference between this program and DisplayDigits.bs2 is that this program is much more versatile. It is much quicker and easier to adjust for different number sequences using lookup tables.

- ✓ Enter and run DisplayDigitsWithLookup.bs2.
- ✓ Verify that it does the same thing as the previous program (with much less work).
- ✓ Take a look at the Debug Terminal while the program runs. It shows how the value of **index** is used by the **LOOKUP** command to load the correct binary value from the list into **OUTH**.

```

' What's a Microcontroller - DisplayDigitsWithLookup.bs2
' Use a lookup table to store and display digits with a 7-segment LED display.

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

index          VAR      Nib

OUTH = %00000000
DIRH = %11111111

DEBUG "index  OUTH   ", CR,
      "-----  -----", CR

FOR index = 0 TO 9

    LOOKUP index, [ %11100111, %10000100, %11010011,
                    %11010110, %10110100, %01110110,

```

```

                                %01110111, %11000100, %11110111, %11110110 ], OUTH
    DEBUG "    ", DEC2 index, "    ", BIN8 OUTH, CR
    PAUSE 1000
NEXT
DIRH = %00000000
END

```

Your Turn – Displaying 0 through F Again

- ✓ Modify `DisplayDigitsWithLookup.bs2` so that it counts from 0 through F in hexadecimal. Don't forget to update the **FOR...NEXT** loop's **start** and **end** values.

6

ACTIVITY #4: DISPLAYING THE POSITION OF A DIAL

In Chapter #5, Activity #4 you used the potentiometer to control the position of a servo. In this activity, you will display the position of the potentiometer using the 7-segment LED display.

Dial and Display Parts

- (1) 7-segment LED display
- (8) Resistors – 1 k Ω (brown-black-red)
- (1) Potentiometer – 10 k Ω
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Capacitor – 0.1 μ F
- (7) Jumper wires

Building the Dial and Display Circuits

Figure 6-14 shows a schematic of the potentiometer circuit that should be added to the project. Figure 6-15 shows a wiring diagram of the circuit from Figure 6-14 combined with the circuit from Figure 6-10 on Page 172.

- ✓ Add the potentiometer circuit to the 7-segment LED display circuit as shown in Figure 6-15.

- ✓ Re-run the program after each change to the **value** variable and note which value from the list gets placed in the **index** variable.

Unless you tell it to make a different kind of comparison, the **LOOKDOWN** command checks to see if a value is equal to an entry in the list. You can also check to see if a value is greater than, less than or equal to, etc. For example, to search for an entry that is less than or equal to the value variable, use the **<=** operator just before the first bracket that starts the list.

- ✓ Modify SimpleLookdown.bs2 by substituting this **value** and **LOOKDOWN** statement:

```
value = 35
LOOKDOWN value, <= [ 7, 19, 28, 85, 167 ], index
```

- ✓ Experiment with different values and see if the index variable displays what you would expect.



Trick question: What happens if your value is greater than 167? This little twist in the **LOOKDOWN** command can cause problems because the **LOOKDOWN** command doesn't make any changes to the index.

```
' What's a Microcontroller - SimpleLookdown.bs2
' Debug an index using a value and a lookup table.

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

value          VAR      Byte
index          VAR      Nib

value = 167

DEBUG ? value

LOOKDOWN value, [7, 85, 19, 167, 28], index

DEBUG ? index, CR

DEBUG "Change the value variable to a ", CR,
      "different number in this list:", CR,
      "7, 85, 19, 167, or 28.", CR, CR,

      "Run the modified program and ", CR,
      "check to see what number the ", CR,
```

```
"LOOKDOWN command places in the ", CR,
"index variable."
```

```
END
```

Example Program: DialDisplay.bs2

This example program mirrors the position of the potentiometer's knob by lighting segments around the outside of the 7-segment LED display as shown in Figure 6-16.

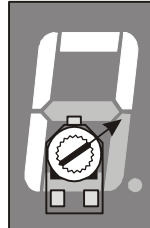
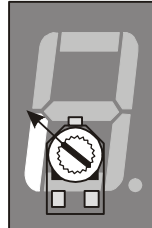


Figure 6-16
Displaying the
Potentiometer's
Position with the
7-Segment LED
Display

- ✓ Enter and run DialDisplay.bs2.
- ✓ Twist the potentiometer's input shaft and make sure it works.
- ✓ When you run the example program, it may not be as precise as shown in Figure 6-16. Adjust the values in the lookdown table so that that the digital display more accurately depicts the position of the potentiometer.

```
' What's a Microcontroller - DialDisplay.bs2
' Display POT position using 7-segment LED display.

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

DEBUG "Program Running!"

index      VAR    Nib
time       VAR    Word

OUTH = %00000000
DIRH = %11111111

DO

  HIGH 5
  PAUSE 100
  RCTIME 5, 1, time

  LOOKDOWN time, <= [40, 150, 275, 400, 550, 800], index
```

```

LOOKUP index, [ %11100101, %11100001, %01100001,
                %00100001, %00000001, %00000000 ], OUTH
LOOP

```

How DialDisplay.bs2 Works

This example program takes an **RCTIME** measurement of the potentiometer and stores it in a variable named **time**.

```

HIGH 5
PAUSE 100
RCTIME 5, 1, time

```

The **time** variable is then used in a **LOOKDOWN** table. The **LOOKDOWN** table decides which number in the list **time** is smaller than, then loads the entry number (0 to 5 in this case) into the **index** variable.

```
LOOKDOWN time, <= [40, 150, 275, 400, 550, 800], index
```

Then, the **index** variable is used in a **LOOKUP** table to choose the binary value to load into the **OUTH** variable.

```

LOOKUP index, [ %11100101, %11100001, %01100001,
                %00100001, %00000001, %00000000 ], OUTH

```

Your Turn – Adding a Segment

DialDisplay.bs2 only makes five of the six segments turn on when you turn the dial. The sequence for turning the LEDs on in DialDisplay.bs2 is E, F, A, B, C, but not D.

- ✓ Save DialDisplay.bs2 under the name DialDisplayYourTurn.bs2.
- ✓ Modify DialDisplayYourTurn.bs2 so that it causes all six outer LEDs to turn on in sequence as the potentiometer is turned. The sequence should be: E, F, A, B, C, and D.

SUMMARY

This chapter introduced the 7-segment LED display, and how to read a pin map. This chapter also introduced some techniques for devices and circuits that have parallel inputs. The `DIRH` and `OUTH` variables were introduced as a means of controlling the values of BASIC Stamp I/O pins P8 through P15. The `LOOKUP` and `LOOKDOWN` commands were introduced as a means for referencing the lists of values used to display letters and numbers.

Questions

1. In a 7-segment LED display, what is the active ingredient that makes the display readable when a microcontroller sends a high or low signal?
2. What does common cathode mean? What do you think common anode means?
3. What is the group of wires that conduct signals to and from a parallel device called?
4. What are the names of the commands in this chapter that are used to handle lists of values?

Exercises

1. Write an `OUTH` command to set P8, P10, P12 high and P9, P11, P13 low. Assuming all your I/O pins started as inputs, write the `DIRH` command that will cause the I/O pins to send high/low signals while leaving P14, P15 configured as inputs.
2. Write the values of `OUTH` required to make the letters: a, C, d, F, H, I, n, P, S.

Project

1. Spell “FISH CHIPS And dIP” over and over again with your 7-segment LED display. Make each letter last for 400 ms.

Solutions

- Q1. The active ingredient is an LED.
- Q2. Common cathode means that all the cathodes are connected together, i.e., they share a common connection point. Common anode would mean that all the anodes are connected together.
- Q3. A parallel bus.
- Q4. `LOOKUP` and `LOOKDOWN` handle list of values.

- E1. The first step for configuring **OUTH** is set to "1" in each bit position specified as **HIGH**. So bits 8, 10, and 12 get set to "1". Then put a "0" for each **LOW**, so bits 9, 11, and 13 get a "0", as shown. To configure **DIRH**, the specified pins, 8, 10, 12, 9, 11, and 13 must be set as outputs by setting those bit to "1". 15 and 14 are configured as inputs by placing zeroes in bits 15 and 14. The second step is to translate this to a PBASIC statement.

```

Bit   15 14 13 12 11 10  9  8      Bit   15 14 13 12 11 10  9  8
OUTH  0  0  0  1  0  1  0  1      DIRH  0  0  1  1  1  1  1  1

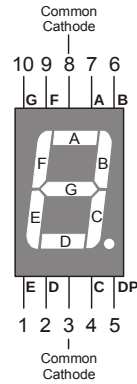
```

OUTH = %00010101

DIRH = %00111111

- E2. The key to solving this problem is to draw out each letter and note which segments must be lit. Place a 1 in every segment that is to be lit. Translate that to the binary **OUTL** value.

Letter	LED Segments	B A F G . C D E	OUTL Value =
a	e, f, a, b, c, g	1 1 1 1 0 1 0 1	%11110101
C	a, f, e, d	0 1 1 0 0 0 1 1	%01100011
d	b, c, d, e, g	1 0 0 1 0 1 1 1	%10010111
F	a, f, e, g	0 1 1 1 0 0 0 1	%01110001
H	f, e, b, c, g	1 0 1 1 0 1 0 1	%10110101
I	f, e	0 0 1 0 0 0 0 1	%00100001
n	e, g, c	0 0 0 1 0 1 0 1	%00010101
P	all but d and e	1 1 1 1 0 0 0 1	%11110001
S	a, f, g, c, d	0 1 1 1 0 1 1 0	%01110110
From Figure 6-2, p. 160			



- P1. Use the Schematic from Figure 6-10 on page 172. To solve this problem, modify `DisplayDigitsWithLookup.bs2`, using the letter patterns worked out in Exercise 1. In the solution, the letters have been set up as constants to make the program more intuitive. Using the binary values is fine too, but more prone to errors.

```

' What's a Microcontroller - Ch6Prj01_FishAndChips.bs2
' Use a lookup table to store and display digits with a 7-segment display.
' Spell out the message:  FISH CHIPS And dIP

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

```

```

' Patterns of 7-Segment Display to create letters
A      CON      %11110101
C      CON      %01100011
d      CON      %10010111
F      CON      %01110001
H      CON      %10110101
I      CON      %00100001
n      CON      %00010101
P      CON      %11110001
S      CON      %01110110
space  CON      %00000000

OUTH = %00000000      ' All off to start
DIRH = %11111111      ' All LEDs must be outputs

index      VAR      Byte      ' 19 chars in message

DO
  DEBUG "index  OUTH  ", CR,
        "-----", CR

  FOR index = 0 TO 18      ' 19 chars in message

    LOOKUP index, [ F, I, S, H, space, C, H, I, P, S, space,
                  A, n, d, space, d, I, P, space ], OUTH

    DEBUG "  ", DEC2 index, " ", BIN8 OUTH, CR

    PAUSE 400      ' 400 ms between letters

  NEXT
LOOP

```

Further Investigation

As with all texts cited in this section, this one is available for free download from www.parallax.com.

“StampWorks”, Workbook, Version 1.2, Parallax Inc., 2001

StampWorks is written by Nuts & Volts author Jon Williams, and features a wide variety of experiments. This text is a collection of 32 experiments including several that use 7-segment LED displays and other types of displays. This text makes use of the INEX 1000 board, which has many of the components you have used in this text built into it.

Chapter #7: Measuring Light

DEVICES THAT CONTAIN LIGHT SENSORS

You have already worked with two different kinds of sensors. The pushbutton can be thought of as a simple pressure sensor, and the potentiometer is a position/rotation sensor. There are many kinds of sensors built into appliances and machines that are not as obvious as a button or a knob. Other common sensors measure things like temperature, smoke, vibration, tilt, and light. Although each of these different kinds of sensors can be found in one or more devices that most people use on a daily basis, light sensors are probably the most common.

One example of an every-day device that contains a light sensor is probably your television. If it can be controlled by a handheld remote, it has a built-in detector for a type of light called infrared that cannot be seen by the human eye. The handheld remote uses infrared light to transmit information about the channel, volume, and other keys that you might press to control the TV. Another common example is a digital camera. A camera's light sensors help it adjust for various lighting conditions so that the picture looks clear regardless of whether it's a sunny or cloudy day.

7

INTRODUCING THE PHOTORESISTOR

Light sensors have many different functions, and they come in different shapes, sizes, and with different price tags. Some sensors are designed to sense a particular color of light, such as blue, green, red, or infrared. Some sensors don't care what color the light is because they react to how bright the light is. Other sensors look for only special kinds of light given off by certain chemical reactions. Light sensors also have a variety of ways to tell a microcontroller what they see. Some sensors send a voltage, some send a sequence of binary values, and others react to different kinds of light or light levels by changing resistance.

Of the light sensors that react to light by changing their resistance, the photoresistor shown in Figure 7-1 is probably the most common, least expensive and easiest to use. Its active ingredient is a chemical compound called cadmium sulfide (CdS). This compound changes resistance depending on how bright the light is that shines on its collecting surface. Bright light causes low resistance values between the two leads while dim light causes higher resistance values.

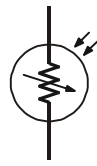


Figure 7-1
Photoresistor Schematic and
Part Drawing.

The photoresistor's cadmium sulfide coated light collecting surface is shown at the top of the part's drawing.

As with a potentiometer, you can use a photoresistor in an RC-time circuit. The program to read the photoresistor is also about the same as the one used to read the potentiometer. Even though the programming is the same, light is very different from rotation or position. The activities in this chapter focus on applications that use light (instead of position) to give the microcontroller information. Along the way, some PBASIC programming techniques will be introduced that will help you with long term data storage, and with making your programs more manageable and readable.

ACTIVITY #1: BUILDING AND TESTING THE LIGHT METER

In this activity, you will build and test an RC-time circuit that reads the value of a photoresistor. The RC-time measurement will give you an idea of the light levels sensed by the photoresistor's light collecting surface. As with the potentiometer test, the time values measured by the `RC TIME` command will be displayed in the Debug Terminal.

Light Detector Test Parts

- (1) Photoresistor
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Capacitor – 0.01 μF
- (1) Capacitor – 0.1 μF
- (1) Jumper wire



Although there are two capacitors in the list of parts, you will only use one capacitor in the circuit at any given time.

Building the RC Time Circuit with a Photoresistor

Figure 7-2 shows a schematic of the RC-time circuit you will use in this chapter, and Figure 7-3 shows the wiring diagram. This circuit is different from the potentiometer circuit from Chapter #5, Activity #3 in two ways. First, the I/O pin used to measure the

decay time is different (P2). Second, the variable resistor is now a photoresistor instead of a potentiometer.

√ Build the circuit shown in Figure 7-2 and Figure 7-3.

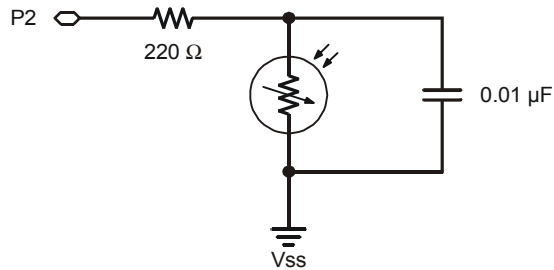


Figure 7-2
Photoresistor RC-time
Circuit Schematic

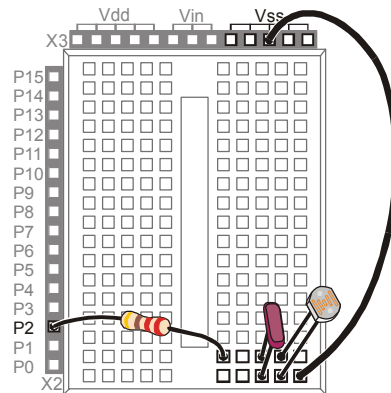


Figure 7-3
Photoresistor RC-time
Wiring Diagram

Programming the Photoresistor Test

The first example program (TestPhotoresistor.bs2) is really just a slightly revised version of ReadPotWithRcTime.bs2 from Chapter #5, Activity #3. The potentiometer circuit from Chapter #5 was connected to I/O pin P7. The circuit in this activity is connected to P2. Because of this difference, the example program has to have two commands updated to make it work. The command that was **HIGH 7**, since the potentiometer circuit was connected to P7, is now **HIGH 2** since the photoresistor circuit is connected to P2. For the same reason, the command that was **RCTIME 7, 1, time** is now **RCTIME 2, 1, time**.

Example Program: TestPhotoresistor.bs2

Instead of twisting the potentiometer's knob, the circuit is tested by exposing the light collecting surface of the photoresistor to different light levels. When the example program is running, the Debug Terminal should display small values for bright light conditions and large values for low light conditions.

- ✓ Enter and run TestPhotoresistor.bs2.
- ✓ While watching the value of the `time` variable on the Debug Terminal note the value under normal lighting conditions.
- ✓ Turn the lights in the room off or cast a shadow over the circuit with your hand and check the `time` variable again. It should be a significantly larger number.
- ✓ If you face the photoresistor's light collecting surface toward direct sunlight, the `time` variable should be fairly small.

```
' What's a Microcontroller - TestPhotoresistor.bs2
' Read photoresistor in RC-time circuit using RCTIME command.

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

time          VAR      Word

DO

    HIGH 2
    PAUSE 100
    RCTIME 2, 1, time
    DEBUG HOME, "time = ", DEC5 time

LOOP
```

Your Turn – Using a Different Capacitor for Different Light Conditions

Replacing the 0.01 μF capacitor with a 0.1 μF capacitor can be useful for more brightly lit rooms or outdoors. The time measurements with the 0.1 μF capacitor will take ten times as long, which means the value of the `time` variable displayed by the Debug Terminal should be ten times as large.

- ✓ Modify the circuit by replacing the 0.01 μF capacitor with a 0.1 μF capacitor.
- ✓ Re-run TestPhotoresistor.bs2 and verify that the RC-time measurements are roughly ten times their former values.

- √ Before you move on to the next activity, return the circuit to the original one shown in Figure 7-2 by removing the 0.1 μF capacitor and replacing it with the 0.01 μF capacitor.

ACTIVITY #2: GRAPHING LIGHT MEASUREMENTS

Factories often have to monitor many sensor inputs to make sure the products they make turn out right. From light levels in a greenhouse to fluid levels in an oil refinery to temperature levels in a nuclear reactor, the people responsible for controlling these levels often rely on graphs of the sensor measurements to get the information they need.

Introducing Stamp Plot Lite

Figure 7-4 shows Stamp Plot Lite software graphing RC-time measurements sent by the BASIC Stamp. The line shown in Figure 7-4 is much easier to understand than the 250 RC-time measurements that were used to plot that line. Viewing the graph from left to right, the RC-time measurements gradually get larger and then suddenly drop off. Since RC-time measurements get larger when the light levels decrease and smaller when they increase, the graph tells a story. It looks like the light level being measured declined gradually, and then it suddenly increased again.

7

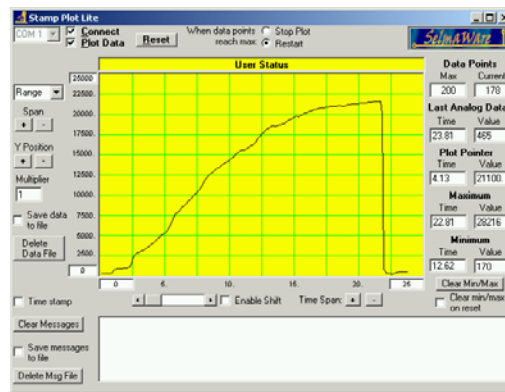


Figure 7-4
Stamp Plot Lite
Graphing
Measured Light
Levels

This graph could depict any number of scenarios to the technician reading it. Perhaps a microcontroller in a greenhouse switched on the artificial lights after the sunlight dropped below a certain level. Perhaps a system of motors and gears that maintains a solar panel's position for maximum sunlight exposure just readjusted the panel's position after

detecting a decrease in light exposure. Regardless of the scenario, if you are familiar with the measurement being graphed, tools like Stamp Plot Lite can really help make sense out of data. It does this by converting lists of measurements into a graph.



Stamp Plot Lite is free for educational use courtesy of SelmaWare Solutions and can be installed from the Parallax CD or downloaded from the Parallax web site or directly from <http://www.selmaware.com/>.

Downloading and Installing Stamp Plot Lite

Before installing Stamp Plot Lite, you must have WinZip installed on your PC or laptop. WinZip can be installed from the Parallax CD or downloaded from www.winzip.com. Below are instructions for downloading and installing Stamp Plot Lite from the Parallax web site:

- ✓ Go to the Downloads area at www.parallax.com.
- ✓ Select BASIC Stamp Software.
- ✓ Download the file labeled “**Stamp Plot Lite** graphing utility...”
- ✓ Save StampPlot.zip to your disk.
- ✓ Open StampPlot.zip by double-clicking it.
- ✓ If you are using the WinZip wizard, follow the screen prompts, and it will run the installation program. If you are using WinZip classic, you will have to double-click Setup.exe to start the installation.
- ✓ Follow the Stamp Plot Lite installation program’s prompts to install the software.

Programming to Send Measurements to Stamp Plot Lite

Sending values you want to graph in Stamp Plot Lite is almost the same as sending numbers to the Debug Terminal, but there are a few rules. First, values are sent using only the **DEC** formatter and the **CR** control character. We want to plot the value of the **time** variable, so all that should be sent to the Debug Terminal is the decimal value followed by a carriage return.

```
DEBUG DEC time, CR
```

You can also send display settings to Stamp Plot Lite by sending special messages in quotes. These messages are called control codes. Control codes are sent to Stamp Plot Lite at the beginning of a PBASIC program. Although you can click and adjust all the settings on the software itself, it is usually easier to program the BASIC Stamp to tell

Stamp Plot Lite those settings for you. Here is an example of some configuration settings from the next example program that will make your RC-time measurements easier to read without any adjustments to Stamp Plot Lite's display settings.

```
DEBUG "!AMAX 1250", CR,
      "!TMAX 25", CR,
      "!TMIN 0", CR,
      "!SHFT ON", CR,
      "!RSET", CR
```



For more information on how to send values and control codes to Stamp Plot Lite, run Stamp Plot Lite's Help file. Click Start, select programs, select Stamp Plot, then click Stamp Plot Help.

Example Program: PlotPhotoresistor.bs2

Follow these steps to plot the photoresistor data:

- ✓ Use the BASIC Stamp Editor to enter and run PlotPhotoresistor.bs2.
- ✓ Verify that there is a single column of values scrolling down the Debug Terminal. Figure 7-5 shows an example.

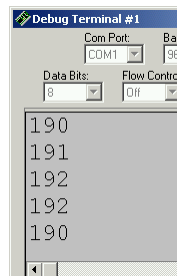


Figure 7-5
Example of Scrolling
Values in the Debug
Terminal.

- ✓ Make a note of the COM number in the COM Port field in the upper left hand corner of the Debug Terminal.
- ✓ Use the Windows Start Menu to run Stamp Plot Lite. Click *Start*, then select *Programs* → *Stamp Plot* → *Stamp Plot Lite*.
- ✓ Set the COM Port field in Stamp Plot Lite to that same value. Figure 7-6 shows an example where the value is COM1 in the Debug Terminal, so Stamp Plot is also set to COM1. Your COM port value may be a different number.

Just check to see what the number is in the Debug Terminal, then set Stamp Plot Lite to that number.

- ✓ Close the Debug Terminal (click the *X* button on the top-right or click the *Close* button near the bottom of the window).
- ✓ In the Stamp Plot Lite window, click *Connect*, then click *Plot Data*. Checkmarks should appear in each box after you click it.

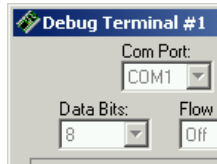


Figure 7-6
COM Port Settings

*Debug Terminal (left)
and Stamp Plot Lite
(right).*

- ✓ Press and release the Reset button on your Board of Education or HomeWork Board. This starts the BASIC Stamp program over from the beginning, which sends the **DEBUG** commands that configure Stamp Plot Lite.
- ✓ The data will start graphing as soon as you click Plot Data. Hold your hand over the photoresistor at different distances to simulate different lighting conditions. Remember, the darker the shadow you cast, the higher the value in the graph; the brighter the light, the smaller the value.



IMPORTANT: Only one program can use a COM port at one time.

Before attempting to run a different program using the BASIC Stamp Editor, you must uncheck the *Connect* and *Plot Data* checkboxes in Stamp Plot Lite.

Before reconnecting Stamp Plot Lite (by clicking the *Connect* and *Plot Data* checkboxes), you must close the Debug Terminal.

```
' What's a Microcontroller - PlotPhotoresistor.bs2
' Graph light levels using Stamp Plot Lite.

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

time          VAR      Word

DEBUG "!AMAX 1250", CR,
      "!TMAX 25", CR,
      "!TMIN 0", CR,
      "!SHFT ON", CR,
      "!RSET", CR
```

```
DO
  HIGH 2
  PAUSE 100
  RCTIME 2, 1, time
  DEBUG DEC time, CR
LOOP
```

Your Turn – Adjusting the Display

Span and Time Span have + and – buttons that you can click to increase or decrease the vertical and horizontal scales. Span is to the left of the area that displays the graph, and you can use it to adjust the maximum and minimum values displayed on the graph. Time Span is below the graph, and you can use it to change how many seconds worth of values plotted in the window.

- √ Experiment with increasing and decreasing these values and note their effects on the how the graph appears.

If you have difficulty finding your plot, you can always press and release the Reset button on your Board of Education or BASIC Stamp HomeWork Board to restore the default settings.

ACTIVITY #3: TRACKING LIGHT EVENTS

One of the more useful features of the BASIC Stamp module's program memory is that you can disconnect the power, but the program is not lost. As soon as you reconnect power, the program starts running again from the beginning. Any portion of the program memory that is not used for the program can be used to store data. This memory is especially good for storing data that you do not want the BASIC Stamp to forget, even if power is disconnected and reconnected.

The chip on the BASIC Stamp that stores program memory and data is shown in Figure 7-7. This chip is called an EEPROM. EEPROM stands for electrically erasable programmable read-only memory. That's quite a mouthful, and pronouncing each of the first letters in EEPROM isn't much better. When people talk about an EEPROM, it is usually pronounced "E-E-Prom".

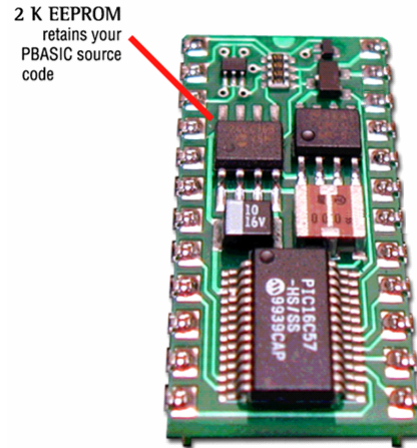


Figure 7-7
EEPROM Chip
on BASIC Stamp

Figure 7-8 shows a window called Memory Map. You can view this window by clicking *Run* and selecting *Memory Map*. The Memory Map uses different colors to show how both the BASIC Stamp module's RAM (variables) and EEPROM (program memory) are being used. The EEPROM Map has two graphs. The bar at the far left shows that only a small fraction of the program memory is used to store the photoresistor program from Activity #2. By scrolling to the bottom of the EEPROM Map's main window and counting the bytes highlighted in blue, you will find that only 101 bytes out of the 2048 byte EEPROM are used for the program. The remaining 1947 bytes are free to store data.

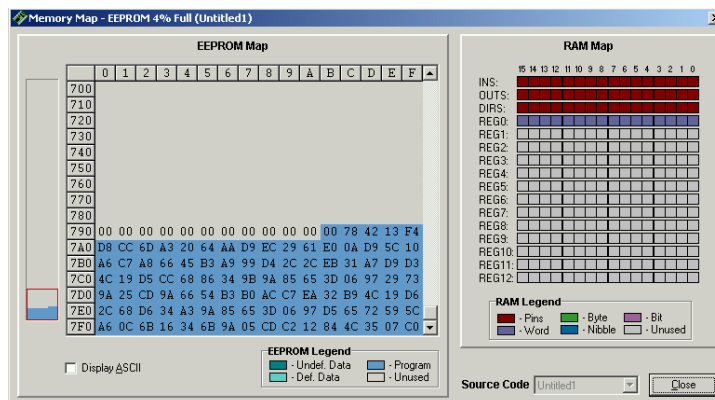


Figure 7-8
Memory Map

*To view this window,
click Run, and select
Memory Map.*



2048 bytes = 2 KB. You can use 2048 bytes to store 2048 different numbers, each of which can store a value between 0 and 255.

The upper-case 'B' stands for bytes. A lower-case 'b' stands for bits. This can make a big difference because 2048 Kb means that 2048 different numbers, but each number is limited to a value of either 0 or 1.

Although both the upper case 'K' and the lower-case 'k' are called kilo, they are slightly different. The upper-case 'K' is used to indicate a binary kilobyte, which is $1 \times 2^{10} = 1024$. When referring to exactly 2000 bytes, you can use the lower-case k, which stands for kilo ($1 \times 10^3 = 1000$) in the metric system.

Using the EEPROM for data storage can be very useful for remote applications. One example of a remote application would be a temperature monitor placed in a truck that hauls frozen food. A second example is a weather monitoring station. One of the pieces of data a weather station might store for later retrieval is light levels.

7

Since we are using a photoresistor to measure light levels, this activity introduces a technique for storing measured light levels to, and retrieving them back from the EEPROM. In this activity, you will run one PBASIC example program that stores a series of light measurements in the BASIC Stamp module's EEPROM. After that program is finished, you will run a second program that retrieves the values from EEPROM and displays them in the Debug Terminal.

Programming Long Term Data Storage

The **WRITE** command is used to store values in the EEPROM, and the **READ** command is used to retrieve those values.

The syntax for the **WRITE** command is:

WRITE Location, {WORD} Data Item

For example, if you want to write the value 195 to address 7 in the EEPROM, you could use the command:

```
WRITE 7, 195
```

Word values can be anywhere between 0 and 65565 while byte values can only contain numbers between 0 and 255. A word value takes the space of two bytes. If you want to write a word size value to EEPROM, you have to use the optional **word** modifier. Be careful though. Since a word takes two bytes, you have to skip one of the byte size addresses in EEPROM before you can write another word. Let's say you need to save

two word values to EEPROM: 659 and 50012. You could start at address 8, but you will have to write the second value to address 10.

```
WRITE 8, Word 659
WRITE 10, Word 50012
```



Is it possible to write over the program? Yes, and if you do, the program is likely to either start behaving strangely or stop running altogether. Since the PBASIC program tokens reside in the highest addresses in the EEPROM, it's best to use the lowest **Address** values for storing numbers with the **WRITE** command.

How do I know if the Address I'm using is too large? You can use the memory map to figure out the highest value not used by your PBASIC program. The numbers on the left side of the main Memory Map graph shown in Figure 7-8 on page 198 are hexadecimal numbers, and 79A is the highest value not occupied by a program token. Hexadecimal means base-16, so 79A is really:

$$\begin{aligned} & 7 \times 16^2 + 9 \times 16^1 + A \times 16^0 \\ &= 7 \times 16^2 + 9 \times 16^1 + 10 \times 16^0 \\ &= 7 \times 256 + 9 \times 16 + 10 \times 1 \\ &= 1946. \end{aligned}$$

Why does this add up to 1946 instead of 1947? There are 1947 total addresses free, but they are numbered Address 0 through Address 1946. You can also program the BASIC Stamp to make this conversion for you using the **DEBUG** command's **DEC** formatter and the **\$** hexadecimal operator like this:

```
DEBUG DEC $79A
```

For a reminder about hexadecimal numbers, see the information box on page 179.

Example Program: StoreLightMeasurementsInEeprom.bs2

This example program demonstrates how to use the **WRITE** command by taking light measurements every 5 seconds for 2 ½ minutes and storing them in EEPROM.

- ✓ Enter and run StoreLightMeasurementsToEeprom.bs2 .
- ✓ Record the measurements displayed on the Debug Terminal so that you can verify the measurements read back from the EEPROM.



Gradually increase the shade over the photoresistor during the 2 ½ minute test period for meaningful data.

```
' Write light measurements to EEPROM.

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

time          VAR      Word
eepromAddress VAR      Byte

DEBUG "Starting measurements...", CR, CR,
      "Measurement  Value", CR,
      "-----", CR

PAUSE 1000

FOR eepromAddress = 0 TO 58 STEP 2
  HIGH 2
  PAUSE 5000
  RCTIME 2, 1, time
  DEBUG DEC2 eepromAddress,
        "          ", DEC time, CR
  WRITE eepromAddress, Word time
NEXT

DEBUG "All done. Now, run:", CR,
      "ReadLightMeasurementsFromEeprom.bs2"

END
```

7

How StoreLightMeasurementsInEeprom.bs2 Works

The **FOR...NEXT** loop that measures the RC-time values and stores them to EEPROM has to count in steps of 2 because word values are written into the EEPROM.

```
FOR eepromAddress = 0 to 58 STEP 2
```

The **RCTIME** command loads the time measurement into the word size **time** variable.

```
RCTIME 2, 1, time
```

The **time** variable is stored at the address given by the current value of the **eepromAddress** variable each time through the loop. Remember, the **eepromAddress** variable is incremented by two each time through the loop because a **Word** variable takes up two bytes. The address for a **WRITE** command is always in terms of bytes.

```
WRITE eepromAddress, Word time
NEXT
```

Programming Data Retrieval

To retrieve these values from EEPROM, you can use the **READ** command. The syntax for the **READ** command is:

READ *Location, {WORD} Data Item*

While the **WRITE** command can use either a constant or a variable, the **READ** command's **DataItem** argument must be a variable, because it has to store the value fetched from the EEPROM by the **READ** command.

Let's say that the variable **eeepromValueA** and **eeepromValueB** are **Word** variables, and **littleEE** is a byte variable. Here are some commands to retrieve the values you stored using the write command.

```
READ 7, littleEE
READ 8, Word eeepromValueA
READ 10, Word eeepromValueB
```

Example Program: ReadLightMeasurementsFromEeprom.bs2

This example program demonstrates how to use the **READ** command to retrieve the light measurements that were stored in EEPROM by **StoreLightMeasurementsInEeprom.bs2**.

- ✓ After **StoreLightMeasurementsToEeprom.bs2** has completed, disconnect and reconnect the power supply to the BASIC Stamp module to prove that the data is not erased from the module's EEPROM when the power is disconnected. Also, close and re-open the BASIC Stamp Editor.
- ✓ Leave the BASIC Stamp module's power disconnected until you are ready to run **ReadLightMeasurementsFromEeprom.bs2**; otherwise, it will start taking measurements again.
- ✓ Enter and run **ReadLightMeasurementsFromEeprom.bs2**.

Compare the table that is displayed by this program with the one displayed by **StoreLightMeasurementsInEeprom.bs2**, and verify that the values are the same.

```
' What's a Microcontroller - ReadLightMeasurementsFromEeprom.bs2
' Read light measurements from EEPROM.

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

time          VAR      Word
```



```

eepromAddress  VAR      Byte

DEBUG "Retrieving measurements", CR, CR,
      "Measurement   Value", CR,
      "-----", CR

FOR eepromAddress = 0 TO 58 STEP 2
  READ eepromAddress, Word time
  DEBUG DEC2 eepromAddress, "          ", DEC time, CR
NEXT

END

```

How ReadLightMeasurementsFromEeprom.bs2 Works

As with the **WRITE** command, the **READ** command uses byte-size addresses. Since words are being read from the EEPROM, the **eepromAddress** variable has to have 2 added to it each time through the **FOR...NEXT** loop.

```
FOR eepromAddress = 0 to 58 STEP 2
```

The **READ** command gets the word size value at **eepromAddress**. This value is loaded into the **time** variable.

```
  READ eepromAddress, Word time
```

The value of the **time** and **eepromAddress** variables are displayed as columns in a table in the Debug Terminal.

```
    DEBUG DEC2 eepromAddress, "          ", DEC time, CR
NEXT
```

Your Turn – Plotting the Stored Data

- ✓ Modify ReadLightMeasurementsFromEeprom.bs2 so that it displays the data to Stamp Plot Lite. Remember, the **DEBUG** statement must only display the value and a carriage return.

ACTIVITY #4: SIMPLE LIGHT METER

Light sensor information can be communicated in a variety of ways. The light meter you will work with in this chapter changes the rate that the display flickers depending on the light intensity it detects.

Light Meter Parts

- (1) Photoresistor
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Capacitor – 0.01 μF
- (1) Capacitor – 0.1 μF
- (1) 7-segment LED display
- (8) Resistors – 1 k Ω (brown-black-red)
- (6) Jumper wires

Building the Light Meter Circuit

Figure 7-9 shows the 7-segment LED display and photoresistor circuit schematics that will be used to make the light meter, and Figure 7-10 shows a wiring diagram of the circuit. The photoresistor circuit is the same one you have been using in the last two activities, and the 7-segment LED display circuit is the one that was controlled by the BASIC Stamp in Chapter #6.

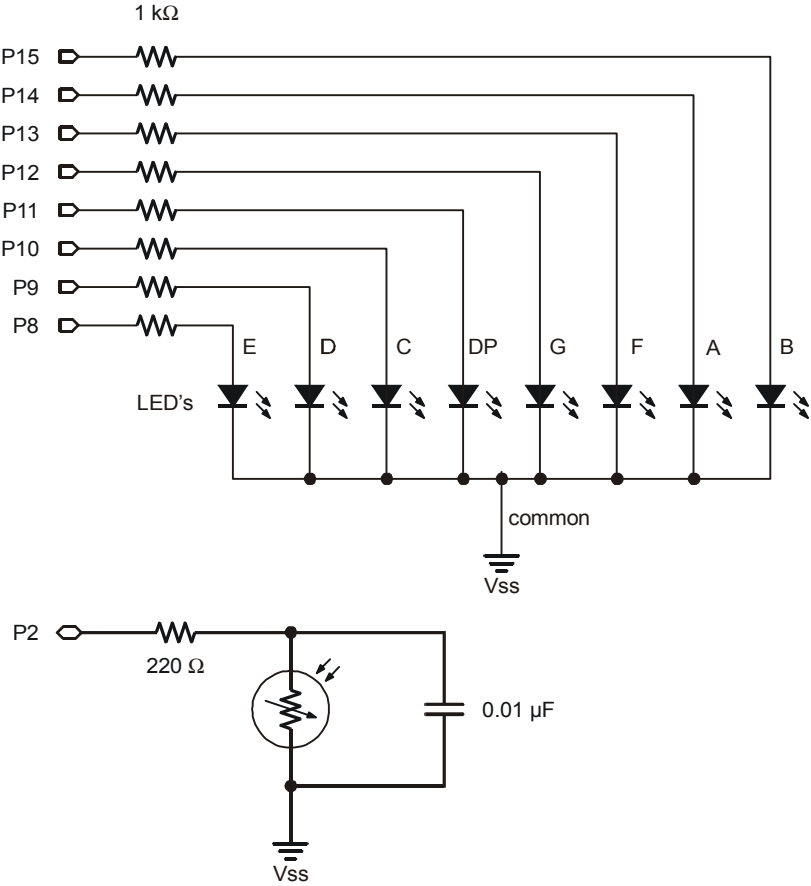


Figure 7-9
Light Meter
Circuit
Schematic.

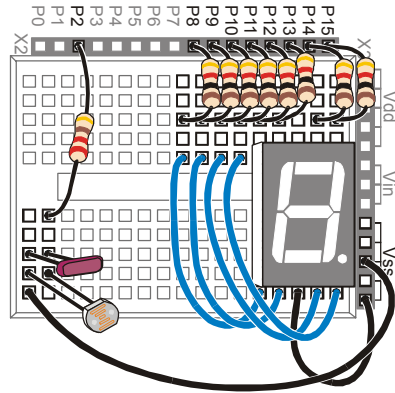


Figure 7-10
Wiring Diagram
for Figure 7-9

- ✓ Build the circuit shown in Figure 7-9 and Figure 7-10.
- ✓ Test the 7-segment LED display to make sure it is connected properly using SegmentTestWithHighLow.bs2 from Chapter #6, Activity #2.

Using Subroutines

Most of the programs you have written so far operate inside a `DO...LOOP`. Since all the main activity happens inside the `DO...LOOP`, it is usually called the main routine. As you add more circuits and more useful functions to your program, it can get kind of difficult to keep track of all the code in the main routine. Your programs will be much easier to work with if you organize them into smaller segments of code that do certain jobs. PBASIC has some commands that you can use to make the program jump out of the main routine, do a job, and then return right back to the same spot in the main routine. This will allow you to keep each segment of code that does a particular job somewhere other than your main routine. Each time you need the program to do one of those jobs, you can write a command inside the main routine that tells the program to jump to that job, do it, and come back when the job is done. This process is called executing a subroutine.

Figure 7-11 shows an example of a subroutine and how it's used. The command `GOSUB Subroutine_Name` causes the program to jump to the `Subroutine_Name:` label. When the program gets to that label, it keeps running and executing commands until it gets to a `RETURN` statement. Then, the program goes back to command that comes after the `GOSUB` command. In the case of the example in Figure 7-11, the next command is: `DEBUG "Next command"`.

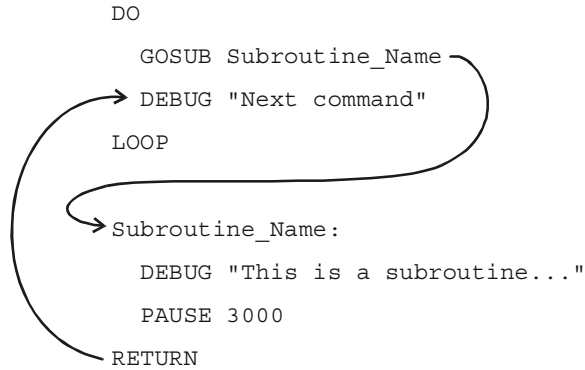


Figure 7-11
How
Subroutines
Work



What's a label? A label is a name that can be used as a placeholder in your program. **GOSUB** is one of the commands you can use to jump to a label. Some others are **GOTO**, **ON GOTO**, and **ON GOSUB**. You can use these commands to jump to labels. A label must end with a colon, and for the sake of style, separate words with the underscore character. When picking a name for a label, make sure not to use a reserved word. The rest of the rules for a label name are the same as the ones for naming variables listed in the information box on page 53.

7

Example Program: SimpleSubroutines.bs2

This example program shows how subroutines work by sending messages to the Debug Terminal.

- ✓ Examine SimpleSubroutines.bs2 and try to guess the order in which the **DEBUG** commands will be executed.
- ✓ Enter and run the program.
- ✓ Compare the program's actual behavior with your predictions.

```

' What's a Microcontroller - SimpleSubroutines.bs2
' Demonstrate how subroutines work.

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

DO

  DEBUG CLS, "Start main routine.", CR
  
```

```

PAUSE 2000
GOSUB First_Subroutine
DEBUG "Back in main.", CR
PAUSE 2000
GOSUB Second_Subroutine
DEBUG "Repeat main...", CR
PAUSE 2000

```

```

LOOP

```

```

First_Subroutine:

```

```

    DEBUG "    Executing first "
    DEBUG "subroutine.", CR
    PAUSE 3000

```

```

RETURN

```

```

Second_Subroutine:

```

```

    DEBUG "    Executing second "
    DEBUG "subroutine.", CR
    PAUSE 3000

```

```

RETURN

```

How SimpleSubroutines.bs2 Works

Figure 7-12 shows how the `First_Subroutine` call in the main routine (the `DO...LOOP`) works. The command `GOSUB First_Subroutine` sends the program to the `First_Subroutine:` label. The three commands inside that subroutine are executed. When the program gets to the `RETURN` statement, it jumps back to the command that comes right after the `GOSUB` command, which is `DEBUG "Back in Main.", CR`.



What's a subroutine call? When you use the `GOSUB` command to make the program jump to a subroutine, it is called a subroutine call.

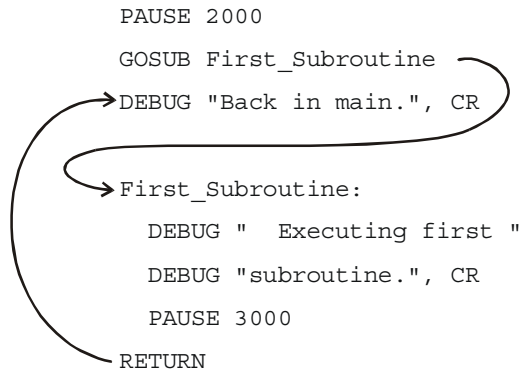


Figure 7-12
First Subroutine
Call

Figure 7-13 shows a second example of the same process with the second subroutine call (`GOSUB Second_Subroutine`).

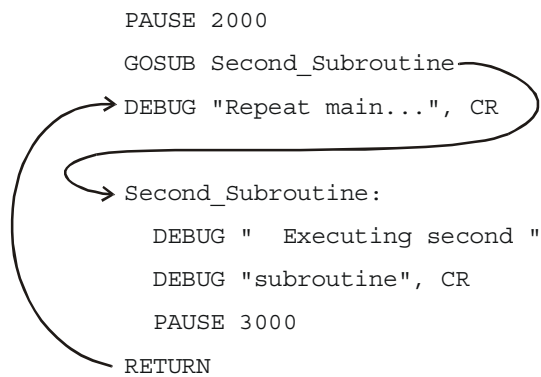


Figure 7-13
Second Subroutine
Call

Your Turn – Adding and Nesting Subroutines

You can add subroutines after the two that are in the program and call them from within the main routine.

- ✓ Add the subroutine example in the Figure 7-11 on page 207 to SimpleSubroutines.bs2.

- ✓ Make any necessary adjustments to the **DEBUG** commands so that the display looks right with all three subroutines.

You can also call one subroutine from within another. This is called nesting subroutines.

- ✓ Try moving the **GOSUB** that calls **Subroutine_Name** into one of the other subroutines, and see how it works.



When nesting subroutines the rule is no more than four deep. See the BASIC Stamp Manual for more details. Look up **GOSUB** and **RETURN**.

Light Meter Using Subroutines

The segments on the display cycle in a circular pattern that gets faster when the light on the photoresistor gets brighter. When the light gets dimmer, the circular pattern displayed by the 7-segment LED display goes slower.

The program that runs the light meter will deal with three different operations:

1. Read the photoresistor.
2. Calculate how long to wait before updating the 7-segment LED display.
3. Update the 7-segment LED display.

Each operation is contained within its own subroutine, and the main **DO...LOOP** routine will cycle and call each one in sequence, over and over again.

Example Program: LightMeter.bs2



Controlled lighting conditions make a big difference. For best results, conduct this test in a room lit by fluorescent lights with no direct sunlight (close the blinds). For information on how to calibrate this meter to other lighting conditions, see the Your Turn section.

- ✓ Enter and run LightMeter.bs2.
- ✓ Verify that the circular pattern displayed by the 7-segment LED display is controlled by the lighting conditions the photoresistor is sensing. Do this by casting a shadow over it with your hand or a piece of paper.


```

' What's a Microcontroller - LightMeter.bs2
' Indicate light level using 7-segment display.

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

DEBUG "Program Running!"

index          VAR      Nib          ' Variable declarations.
time           VAR      Word

OUTH = %00000000          ' Initialize 7-segment display.
DIRH = %11111111

DO                      ' Main routine.

    GOSUB Get_Rc_Time
    GOSUB Delay
    GOSUB Update_Display

LOOP

                                ' Subroutines

Get_Rc_Time:            ' RC-time subroutine

    HIGH 2
    PAUSE 3
    RCTIME 2, 1, time

RETURN

Delay:                  ' Delay subroutine.

    PAUSE time

RETURN

Update_Display:        ' Display updating subroutine.

    IF index = 6 THEN index = 0
    '          BAFG.CDE
    LOOKUP index, [ %01000000,
                    %10000000,
                    %00000100,
                    %00000010,
                    %00000001,
                    %00100000 ], OUTH
    index = index + 1

RETURN

```

How LightMeter.bs2 Works

The first two lines of the program declare variables. It doesn't matter whether these variables are used in subroutines or the main routine, it's always best to declare variables (and constants) at the beginning of your program. Since this is such a common practice, this section of code has a name, variable declarations. This name is shown in the comment to the right of the first variable declaration.

```
index VAR Nib           ' Variable declarations.
time  VAR Word
```

Many programs also have things that need to get done once at the beginning of the program. Setting all the 7-segment I/O pins low and then making them outputs is an example. This section of a PBASIC program also has a name, initialization.

```
OUTH = %00000000      ' Initialize 7-segment display.
DIRH = %11111111
```

This next segment of code is called the main routine. The main routine calls the **Get_Rc_Time** subroutine first. Then, it calls the **Delay** subroutine, and after that, it calls the **Update_Display** subroutine. Keep in mind that the program goes through the three subroutines as fast as it can, over and over again.

```
DO                      ' Main routine.
  GOSUB Get_Rc_Time
  GOSUB Delay
  GOSUB Update_Display
LOOP
```

All subroutines are usually placed after the main routine. The first subroutine's name is **Get_Rc_Time:**, and it takes the RC-time measurement on the photoresistor circuit. This subroutine has a **PAUSE** command that charges up the capacitor. The *Duration* of this command is small because it only needs to pause long enough to make sure the capacitor is charged. Note that the **RCTIME** command sets the value of the **time** variable. This variable will be used by the second subroutine.

```
                        ' Subroutines
Get_Rc_Time:          ' RC-time subroutine
  HIGH 2
  PAUSE 3
  RCTIME 2, 1, time
```

```
RETURN
```

The second subroutine's name is **Delay**, and all it contains is a **PAUSE** command. If you want to do some extra math on the value of the **time** variable before using it in the **PAUSE** command, it would be appropriate to do that in this subroutine.

```
Delay:
  PAUSE time
RETURN
```

The third subroutine is named **Update_Display**. The **LOOKUP** command in this subroutine contains a table with six bit patterns that are used to create the circular pattern around the outside of the 7-segment LED display. By adding 1 to the **index** variable each time the subroutine is called, it causes the next bit pattern in the sequence to get placed in **OUTH**. There are only entries in the **LOOKUP** command's lookup table for **index** values from 0 through 5. What happens when the value of **index** gets to 6? The lookup command doesn't automatically know to go back to the first entry, but you can use an **IF...THEN** statement to fix that problem. The command **IF index = 6 THEN index = 0** resets the value of **index** to 0 each time it gets to 6. It also causes the sequence of bit patterns placed in **OUTH** to repeat itself over and over again. This, in turn, causes the 7-segment LED display to repeat its circular pattern over and over again.

```
Update_Display:
  IF index = 6 THEN index = 0
  '      BAFG.CDE
  LOOKUP index, [ %01000000,
                  %10000000,
                  %00000100,
                  %00000010,
                  %00000001,
                  %00100000 ], OUTH
  index = index + 1
RETURN
```

Your Turn – Adjusting the Meter's Hardware and Software

There are two ways to change the sensitivity of the meter. First the software can be changed. For example, the 7-segment LED display will cycle at one-tenth the speed if you multiply the time variable by 10 in the **Delay** subroutine, and it will cycle twice as fast if you divide the time variable by 2.

- ✓ Modify **LightMeter.bs2** so that the **time** variable is multiplied by 10. The easiest way to do this is to change

```
        PAUSE time  
to  
        PAUSE time * 10  
in the Delay subroutine.
```

- ✓ Run the modified program and test to make sure the cycling of the 7-segment LED display is now one tenth of what it was before.
- ✓ You can also try multiplying the `time` variable by other values such as 5 or 20, or dividing by 2 using `PAUSE time / 2`.

You can also make the display cycle at one tenth the speed by swapping the 0.01 μ F capacitor for the 0.1 μ F capacitor. Remember that when you use a capacitor that is ten times as large, the RC-time measurement will become ten times as long.

- ✓ Replace the 0.01 μ F capacitor with a 0.1 μ F capacitor.
- ✓ Run the program and see if the predicted effect occurred.



Which is better, adjusting the software or the hardware? You should always try to use the best of both worlds. Pick a capacitor that gives you the most accurate measurements over the widest range of light levels. Once your hardware is the best it can be, use the software to automatically adjust the light meter so that it works well for the user, both indoors and outdoors. This takes a considerable amount of testing and refinement, but that's all part of the product design process.

SUMMARY

This chapter introduced a second way to use the **RCTIME** command by using it to measure light levels with a photoresistor. Like the potentiometer, the photoresistor is a variable resistor. Unlike the potentiometer, the photoresistor's resistance changes with light levels instead of with position. Stamp Plot Lite was used to graph successive light measurements, and methods for recording and interpreting graphical data were introduced. The **WRITE** and **READ** commands were used to store and retrieve values to and from the BASIC Stamp module's EEPROM. The EEPROM was then used in an RC-time data logging application. In this chapter's last activity, a light meter application was developed. This application used subroutines to perform the three different jobs required for the light meter to function.

7

Questions

1. What kind of different things can sensors detect?
2. What is the name of the chemical compound that makes a photoresistor sensitive to light?
3. How is a photoresistor similar to a potentiometer? How is it different?
4. What does EEPROM stand for?
5. How many bytes can the BASIC Stamp module's EEPROM store? How many bits can it store?
6. What command do you use to store a value in EEPROM? What command do you use to retrieve a value from EEPROM? Which one requires a variable?
7. What is a label?
8. What is a subroutine?
9. What command is used to call a subroutine? What command is used to end a subroutine?

Exercises

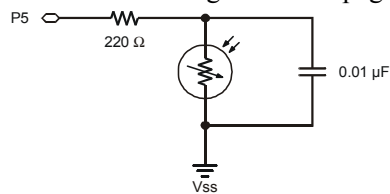
1. Draw the schematic of a photoresistor RC-time circuit connected to P5.
2. Modify TestPhotoresistor.bs2 so that it works on a circuit connected to P5 instead of P2.
3. Explain how you would modify LightMeter.bs2 so that the circular pattern displayed by the 7-segment LED display goes in the opposite direction.

Project

1. In an earlier chapter, you used a pushbutton to make an LED blink. Instead of using a pushbutton, use a photoresistor to make the LED blink when you cast a shadow over it. Hint: You can use an **IF...THEN** statement and the greater than/less than operators to decide if your **time** measurement is above or below a certain value. The operator **>** is used for greater than, and the operator **<** is used for less than.

Solutions

- Q1. Pressure, position, rotation, temperature, smoke, vibration, tilt, light, and almost anything else you may think of: humidity, g-force, flexion, flow rate, the list goes on...
- Q2. Cadmium sulfide (CdS).
- Q3. Both devices have varying resistance values. The photoresistor's resistance varies according to the light level falling upon it, unlike the potentiometer, which only changes when the knob is turned.
- Q4. Electrically Erasable Programmable Read-Only Memory.
- Q5. 2048 bytes. $2048 \times 8 = 16,384$ bits.
- Q6. To store a value – **WRITE**
To retrieve a value – **READ**
The **READ** command requires a variable.
- Q7. A label is a name that can be used as a placeholder in a PBASIC program.
- Q8. A subroutine is a small segment of code that does a certain job.
- Q9. Calling: **GOSUB**; ending: **RETURN**
- E1. Schematic based on Figure 7-2 on page 191, P2 changed to P5.



- E2. The required changes are very similar to those explained on page 191.

```
DO
HIGH 5
PAUSE 100
RCTIME 5, 1, time
```

```

    DEBUG HOME, "time = ", DEC5 time
LOOP

```

- E3. To go in the opposite direction, the patterns must be displayed in the reverse order. This can be done by switching the patterns around inside the `LOOKUP` statement, or by reversing the order they get looked up.

Solution 1

```

Update_Display:
IF index = 6 THEN index = 0
'
    BAFG.CDE
LOOKUP index, [ %01000000,
                %10000000,
                %00000100,
                %00000010,
                %00000001,
                %00100000 ], OUTH

index = index + 1
RETURN

```

Solution 2

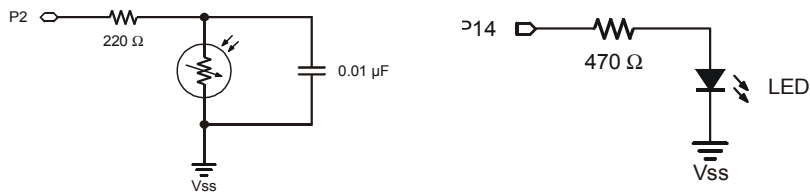
```

Update_Display:
'
    BAFG.CDE
    LOOKUP index, [ %01000000,
                    %10000000,
                    %00000100,
                    %00000010,
                    %00000001,
                    %00100000 ], OUTH

    IF (index = 0) THEN
        index = 5
    ELSE
        index = index - 1
    ENDIF
RETURN

```

- P1. Photoresistor from Figure 7-2, p.191; LED from Figure 2-11, p.48.



The key to solving this problem is to insert an **IF...THEN** statement that tests whether the photoresistor reading is above some threshold value. If it is, flash the LED. The threshold value can be found by running TestPhotoresistor.bs2 and observing the readings. Note the difference between an unshaded and a shaded value. Take a value somewhere in the middle and use that for your threshold. In the solution shown, the threshold value was encoded in a constant named **Dark** to make the program easier to change.

```
' What's a Microcontroller - Ch07Prj01_PhotoresistorFlasher.bs2
' Make LED on P14 flash whenever a shadow is cast over
' the photoresistor. Change "Dark" constant for your conditions.

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

Dark    CON    25
time    VAR    Word

DO

    HIGH 2                      ' Read photoresistor with RCTIME
    PAUSE 100
    RCTIME 2, 1, time

    DEBUG HOME, "time = ", DEC5 time ' Print value to Debug Terminal

    IF (time > Dark) THEN        ' Compare reading to known dark value
        HIGH 14                  ' Blink LED on pin P14
        PAUSE 100
        LOW 14
        PAUSE 100
    ENDIF

LOOP
```

Further Investigation

“Applied Sensors”, Student Guide, Version 2.0, Parallax Inc., 2003

More in-depth coverage of light measurement using a photodiode, scientific units and math are featured in this text along with other sensor applications.

“Industrial Control”, Student Guide, Version 2.0, Parallax Inc., 2002

Stamp Plot Lite was developed in conjunction with this text to demonstrate the fundamentals of techniques used in industrial process control.

Chapter #8: Frequency and Sound

YOUR DAY AND ELECTRONIC BEEPS

Here are a few examples of beeps you might hear during a normal day: The microwave oven beeps when it's done cooking your food. The cell phone plays different tones of beeps that resemble songs to get your attention when a call is coming in. The ATM machine beeps to remind you not to forget your card. A store cash register beeps to let the teller know that the bar code of the grocery item passed over the scanner was read. Many calculators beep when the wrong keys are pressed. Let's not forget that you may have started your day with a beeping alarm clock.

MICROCONTROLLERS, SPEAKERS, BEEPS AND ON/OFF SIGNALS

Just about all of the electronic beeps you hear during your daily routine are made by microcontrollers connected to speakers. The microcontroller creates these beeps by sending rapid high/low signals to various types of speakers. The rate of these high/low signals is called the frequency, and it determines the tone or pitch of the beep. Each time a high/low repeats itself, it is called a cycle. You will often see the number of cycles per second referred to as Hertz, and it is abbreviated Hz. For example, one of the most common frequencies for the beeps that help machines get your attention is 2 kHz. That means that the high/low signals repeat at 2000 times per second.

8

Introducing the Piezoelectric Speaker

In this activity, you will experiment with sending a variety of signals to a common, small, and inexpensive speaker called a piezoelectric speaker. Its schematic symbol and part drawing are shown in Figure 8-1.

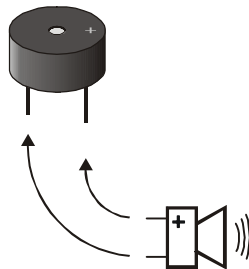


Figure 8-1
Piezoelectric Speaker
Schematic Symbol and
Part Drawing



A **piezoelectric speaker** is commonly referred to as a piezo speaker or piezo buzzer, and piezo is pronounced "pE-A-zO".

ACTIVITY #1: BUILDING AND TESTING THE SPEAKER

In this activity, you will build and test the piezoelectric speaker circuit.

Speaker Parts

- (1) Piezoelectric speaker
- (2) Jumper wires

Building the Piezoelectric Speaker Circuit

The negative terminal of the piezoelectric speaker should be connected to Vss, and the positive terminal should be connected to an I/O pin. The BASIC Stamp will then be programmed to send high/low signals to the piezoelectric speaker's positive terminal.

✓ Build the circuit shown in Figure 8-2.

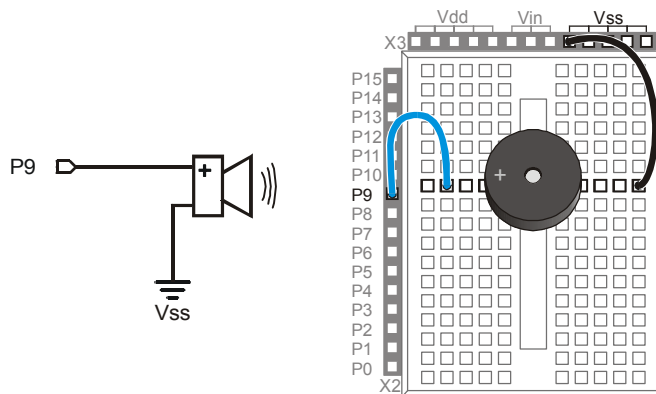


Figure 8-2
Piezoelectric
Speaker Circuit
Schematic and
Wiring Diagram

How the Piezoelectric Speaker Circuit Works

When a guitar string vibrates, it causes changes in air pressure. These changes in air pressure are what your ear detects as a tone. The faster the changes in air pressure, the higher the pitch, and the slower the changes in air pressure, the lower the pitch. The element inside the piezo speaker's plastic case is called a piezoelectric element. When high/low signals are applied to the speaker's positive terminal, the piezoelectric element

vibrates, and it causes changes in air pressure just as a guitar string does. As with the guitar string, your ear detects the changes in air pressure caused by the piezoelectric speaker, and it typically sounds like a beep or a tone.

Programming Speaker Control

The **FREQOUT** command is a convenient way of sending high/low signals to a speaker to make sound. The *BASIC Stamp Manual* shows the command syntax as this:

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

As with most of the other commands used in this book, ***Pin*** is a value you can use to choose which BASIC Stamp I/O pin to use. The ***Duration*** argument is a value that tells the **FREQOUT** command how long the tone should play, in milliseconds. The ***Freq1*** argument is used to set the frequency of the tone, in Hertz. There is an optional ***Freq2*** argument that can be used to mix frequencies.

Here is how to send a tone to I/O pin P9 that lasts for 1.5 seconds and has a frequency of 2 kHz:

```
FREQOUT 9, 1500, 2000
```

8

Example Program: TestPiezoWithFreqout.bs2

This example program sends the 2 kHz tone to the speaker on I/O pin P9 for 1.5 seconds. You can use the Debug terminal to see when the speaker should be beeping and when it should stop.

- ✓ Enter and run TestPiezoWithFreqout.bs2.
- ✓ Verify that the speaker makes a clearly audible tone during the time that the Debug Terminal displays the message “Tone sending...”

```
' What's a Microcontroller - TestPiezoWithFreqout.bs2
' Send a tone to the piezo speaker using the FREQOUT command.

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

DEBUG "Tone sending...", CR

FREQOUT 9, 1500, 2000

DEBUG "Tone done."
```

Your Turn – Adjusting Frequency and Duration

- ✓ Save TestPiezoWithFreqout.bs2 under a different name.
- ✓ Try some different values for the *Duration* and *Freq1* argument.
- ✓ After each change, run the program and make a note of the effect.
- ✓ As the *Freq1* argument gets larger, does the tone's pitch go up or down? Try values of 1500, 2000, 2500 and 3000 to answer this question.

ACTIVITY #2: ACTION SOUNDS

Many toys contain microcontrollers that are used to make action sounds. Action sounds tend to involve rapidly changing the frequency played by the speaker. You can also get some interesting effects from mixing two different tones together using the **FREQOUT** command's optional *Freq2* argument. This activity introduces both techniques.

Programming Action Sounds

Action and appliance sounds have three different components:

1. Pause
2. Duration
3. Frequency

The pause is the time between tones, and you can use the **PAUSE** command to create that pause. The duration is the amount of time a tone lasts, which you can set using the **FREQOUT** command's *Duration* argument. The frequency determines the pitch of the tone. The higher the frequency, the higher the pitch, the lower the frequency, the lower the pitch. This is, of course, determined by the **FREQOUT** command's *Freq1* argument.

Example Program: ActionTones.bs2

ActionTones.bs2 demonstrates a few different combinations of pause, duration, and frequency. The first sequence of tones sounds similar to an electronic alarm clock. The second one sounds similar to something a familiar science fiction movie robot might say. The third is more the kind of sound effect you might hear in an old video game.

- ✓ Enter and run ActionTones.bs2.

```
' What's a Microcontroller - ActionTones.bs2
' Demonstrate how different combinations of pause, duration, and frequency
' can be used to make sound effects.
```

```

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

duration      VAR      Word
frequency     VAR      Word

DEBUG "Alarm...", CR
  PAUSE 100
  FREQOUT 9, 500, 1500
  PAUSE 500
  FREQOUT 9, 500, 1500
  PAUSE 500
  FREQOUT 9, 500, 1500
  PAUSE 500
  FREQOUT 9, 500, 1500
  PAUSE 500

DEBUG "Robot reply...", CR
  PAUSE 100
  FREQOUT 9, 100, 2800
  FREQOUT 9, 200, 2400
  FREQOUT 9, 140, 4200
  FREQOUT 9, 30, 2000
  PAUSE 500

DEBUG "Hyperspace...", CR
  PAUSE 100
  FOR duration = 15 TO 1 STEP 1
    FOR frequency = 2000 TO 2500 STEP 20
      FREQOUT 9, duration, frequency
    NEXT
  NEXT

DEBUG "Done", CR

END

```

How ActionTones.bs2 Works

The “Alarm” routine sounds like an alarm clock. This routine plays tones at a fixed frequency of 1.5 kHz for a duration of 0.5 s with a fixed delay between tones of 0.5 s. The “Robot reply” routine uses various frequencies for brief durations.

The “Hyperspace” routine uses no delay, but it varies both the duration and frequency. By using **FOR...NEXT** loops to rapidly change the frequency and duration, you can get some interesting sound effects. When one **FOR...NEXT** loop executes inside another one, it is called a nested loop. Here is how the nested **FOR...NEXT** loop shown below works. The

duration variable starts at 15, then the **frequency** loop takes over and sends frequencies of 2000, then 2020, then 2040, and so on, up through 2500 to the piezo speaker. When the frequency loop is finished, the **duration** loop has only repeated one of its 15 passes. So it subtracts one from the value of **duration** and repeats the **frequency** loop all over again.

```

FOR duration = 15 TO 1
  FOR frequency = 2000 TO 2500 STEP 15
    FREQOUT 9, duration, frequency
  NEXT
NEXT

```

Example Program: NestedLoops.bs2

To better understand how nested **FOR...NEXT** loops work, NestedLoops.bs2 uses the **DEBUG** command to show the value of a much less complicated version of the nested loop used in ActionTones.bs2.

- ✓ Enter and run NestedLoops.bs2.
- ✓ Examine the Debug Terminal output and verify how the duration and frequency arguments change each time through the loop.

```

' What's a Microcontroller - ActionTones.bs2
' Demonstrate how different combinations of pause, duration, and frequency
' can be used to make sound effects.

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

duration      VAR      Word
frequency     VAR      Word

DEBUG "Alarm...", CR
  PAUSE 100
  FREQOUT 9, 500, 1500
  PAUSE 500
  FREQOUT 9, 500, 1500
  PAUSE 500
  FREQOUT 9, 500, 1500
  PAUSE 500
  FREQOUT 9, 500, 1500
  PAUSE 500

DEBUG "Robot reply...", CR
  PAUSE 100
  FREQOUT 9, 100, 2800

```

```

FREQOUT 9, 200, 2400
FREQOUT 9, 140, 4200
FREQOUT 9, 30, 2000
PAUSE 500

DEBUG "Hyperspace...", CR
PAUSE 100
FOR duration = 15 TO 1 STEP 1
  FOR frequency = 2000 TO 2500 STEP 20
    FREQOUT 9, duration, frequency
  NEXT
NEXT

DEBUG "Done", CR

END

```

Your Turn – More Sound Effects

There is pretty much an endless number of ways to modify `ActionTones.bs2` to get different sound combinations. Here is just one modification to the “Hyperspace” routine:

8

```

DEBUG "Hyperspace jump...", CR
FOR duration = 15 TO 1 STEP 3
  FOR frequency = 2000 TO 2500 STEP 15
    FREQOUT 9, duration, frequency
  NEXT
NEXT
FOR duration = 1 TO 36 STEP 3
  FOR frequency = 2500 TO 2000 STEP 15
    FREQOUT 9, duration, frequency
  NEXT
NEXT

```

- ✓ Save your example program under the name `ActionTonesYourTurn.bs2`.
- ✓ Have fun with this and other modifications of your own invention.

Two Frequencies at Once

You can send two frequencies at the same time. In audio, this is called “mixing”. Remember the **FREQOUT** command’s syntax from Activity #1:

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

You can use the optional **Freq2** argument to mix two frequencies using the **FREQOUT** command. For example, you can mix 2 and 3 kHz together like this:

```
FREQOUT 9, 1000, 2000, 3000
```



Each touchtone keypad tone is also an example of two frequencies mixed together. In telecommunications, that is called DTMF (Dual Tone Multi Frequency). There is also a PBASIC command called **DTMFOUT** that is designed just for sending phone tones. For examples of projects where phone numbers are dialed, see the **DTMFOUT** command in the BASIC Stamp Manual.

Example Program: MixingTones.bs2

This example program demonstrates the difference in tone that you get when you mix 2 and 3 kHz together. It also demonstrates an interesting phenomenon that occurs when you mix two sound waves that are very close in frequency. When you mix 2000 Hz with 2001 Hz, the tone will fade in and out once every second (at a frequency of 1 Hz). If you mix 2000 Hz with 2002 Hz, it will fade in and out twice a second (2 Hz), and so on.



Beat is when two tones very close in frequency are played together causing the tone you hear to fade in and out. The frequency of that fading in and out is the difference between the two frequencies. If the difference is 1 Hz, the tone will fade in and out at 1 Hz. If the difference is 2 Hz, the tone will fade in and out at 2 Hz.

The variations in air pressure made by the piezoelectric speaker are called sound waves. When the tone is loudest, the variations in air pressure caused by the two frequencies are adding to each other (called superposition). When the tone is at its quietest, the variations in air pressure are canceling each other out (called interference).

- ✓ Enter and run MixingTones.bs2.
- ✓ Keep an eye on the Debug Terminal as the tones play, and note the different effects that come from mixing the different tones.

```
' What's a Microcontroller - MixingTones.bs2
' Demonstrate some of the things that happen when you mix two tones.

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

DEBUG "Frequency = 2000", CR
FREQOUT 9, 4000, 2000

DEBUG "Frequency = 3000", CR
FREQOUT 9, 4000, 3000

DEBUG "Frequency = 2000 + 3000", CR
FREQOUT 9, 4000, 2000, 3000

DEBUG "Frequency = 2000 + 2001", CR
```



```

FREQOUT 9, 4000, 2000, 2001

DEBUG "Frequency = 2000 + 2002", CR
FREQOUT 9, 4000, 2000, 2002

DEBUG "Frequency = 2000 + 2003", CR
FREQOUT 9, 4000, 2000, 2003

DEBUG "Frequency = 2000 + 2005", CR
FREQOUT 9, 4000, 2000, 2005

DEBUG "Frequency = 2000 + 2010", CR
FREQOUT 9, 4000, 2000, 2010

DEBUG "Done", CR

END

```

Your Turn – Condensing the Code

MixingTones.bs2 was written to demonstrate some interesting things that can happen when you mix two different frequencies using the **FREQOUT** command's optional **Freq2** argument. However, it is extremely inefficient.

8

- ✓ Modify MixingTones.bs2 so that it cycles through the **Freq2** arguments ranging from 2001 to 2005 using a word variable and a loop.

ACTIVITY #3: MUSICAL NOTES AND SIMPLE SONGS

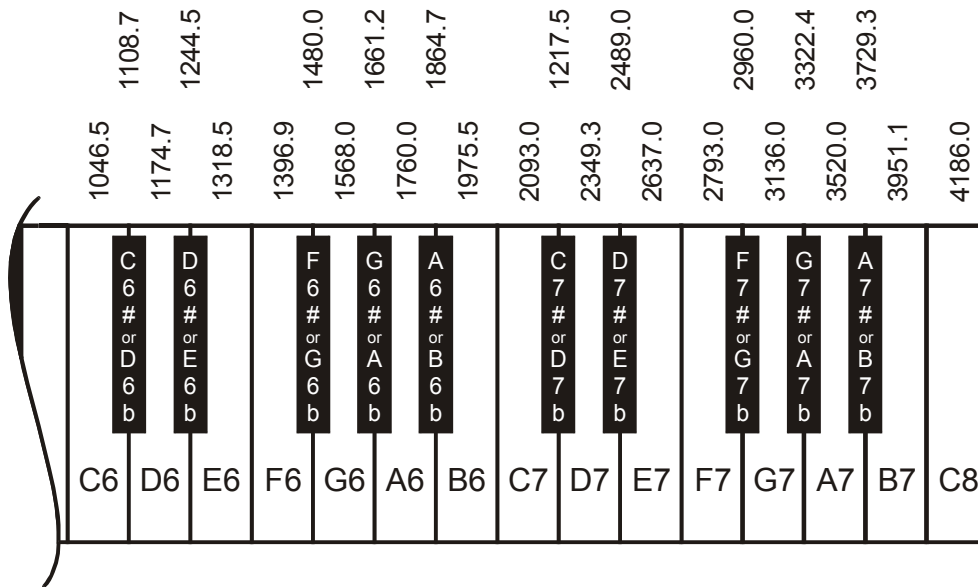
Figure 8-3 shows the rightmost 25 keys of a piano keyboard. It also shows the frequencies at which each wire inside the piano vibrates when that piano key is struck. The keys and their corresponding notes are labeled C6 through C8. These keys are separated into groups of 12. Each group spans one octave, made up of 8 white keys and 4 black keys. The sequence of notes repeats itself every 12 keys. Notes of the same letter are related by frequency, doubling with each higher octave. For example, C7 is twice the frequency of C6, and C8 is twice the frequency of C7. Likewise, if you go one octave down, the frequency will be half the value; for example, A6 is half the frequency of A7.



Internet search for – “musical scale”: By using the words "musical scale" in a search engine like Google or Yahoo, you will find lots of fascinating information about the history, physics and psychology of the subject. The 12 note per octave scale is the main scale of western music. Other cultures use scales that contain 2 to 35 notes per octave.

If you've ever heard a singer practice his/her notes by singing the Solfege, "Do Re Mi Fa Sol La Ti Do", the singer is attempting to match the notes that you get from striking the white keys on a piano keyboard. These white keys are called natural keys. A black key on a piano can either be called sharp or flat. For example, the black key between the C and D keys is either called C-sharp ($C^\sharp$) or D-flat (D^b). Whether a key is called sharp or flat depends on the particular piece being played, and the rules for that are better left to the music classes.

Figure 8-3: Rightmost Piano Keys and Their Frequencies





Tuning Method: The keyboard in Figure 8-3 uses a method of tuning called equal temperament. The frequencies are determined using a reference note, then multiplying it by $2^{(n/12)}$ for values of $n = 1, 2, 3$, etc. For example, you can take the frequency for A6, and multiply by $2^{(1/12)}$ to get the frequency for A6#. Multiply it by $2^{(2/12)}$ to get the frequency for B6, and so on. Here is an example of calculating the frequency for B6 using A6 as a reference frequency:

The frequency of A6 is 1760

$2^{(2/12)} = 1.1224$

$1760 \times 1.1224 = 1975.5$

1975.5 is the frequency of B6

Programming Musical Notes

The **FREQOUT** command is also useful for musical notes. Programming the BASIC Stamp to play music using a piezospeaker involves following a variety of rules used in playing music using any other musical instrument. These rules apply to the same elements that were used to make sound effects, frequency, duration, and pause. This next example program plays some of the musical note frequencies on the piezospeaker, each with a duration of half a second.



Example Program: DoReMiFaSolLaTiDo.bs2

√ Enter and run DoReMiFaSolLaTiDo.bs2

```
' What's a Microcontroller - DoReMiFaSolLaTiDo.bs2
' Send an octave of half second tones using a piezoelectric speaker.

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

'Solfège           Tone           Note
DEBUG "Do...", CR:  FREQOUT 9,500,1047 ' C6
DEBUG "Re...", CR:  FREQOUT 9,500,1175 ' D6
DEBUG "Mi...", CR:  FREQOUT 9,500,1319 ' E6
DEBUG "Fa...", CR:  FREQOUT 9,500,1396 ' F6
DEBUG "Sol...", CR:  FREQOUT 9,500,1568 ' G6
DEBUG "La...", CR:  FREQOUT 9,500,1760 ' A6
```

```

DEBUG "Ti...", CR:  FREQOUT 9,500,1976      ' B6
DEBUG "Do...", CR:  FREQOUT 9,500,2093      ' C7
END

```

Your Turn – Sharp/Flat Notes

- ✓ Use the frequencies shown in Figure 8-3 to add the five sharp/flat notes to DoReMiFaSolLaTiDo.bs2
- ✓ Modify your program so that it plays the next octave up. Hint: Save yourself some typing and just use the $\ast 2$ operation after each **Freq1** argument. For example, **FREQOUT 9, 500, 1175 $\ast$ 2** will give you D7, the D note in the 7th octave.

Storing and Retrieving Sequences of Musical Notes

A good way of saving musical notes is to store them using the BASIC Stamp module's EEPROM. Although you could use many **WRITE** commands to do this, a better way is to use the **DATA** directive. This is the syntax for the **DATA** directive:

{Symbol} DATA {Word} Dataltem {, {Word} Dataltem, ... }

Here is an example of how to use the **DATA** directive to store the characters that correspond to musical notes.

```
Notes DATA "C", "C", "G", "G", "A", "A", "G"
```

You can use the **READ** command to access these characters. The letter 'C' is located at address **Notes + 0**, and a second letter 'C' is located at **Notes + 1**. Then, there's a letter 'G' at **Notes + 2**, and so on. For example, if you want to load the last letter 'G' into a byte variable called **noteLetter**, use the command:

```
READ Notes + 6, noteLetter
```

You can also store lists of numbers using the **DATA** directive. Frequency and duration values that the BASIC Stamp uses for musical notes need to be stored in word variables because they are usually greater than 255. Here is how to do that with a **DATA** directive.

```
Frequencies DATA Word 2093, Word 2093, Word 3136, Word 3136,
                  Word 3520, Word 3520, Word 3136
```

Because each of these values occupies two bytes, accessing them with the read command is different from accessing characters. The first 2093 is at **Frequencies + 0**, but the second 2093 is located at **Frequencies + 2**. The first 3136 is located at **Frequencies + 4**, and the second 3136 is located at **Frequencies + 6**.



The values in the **Frequencies DATA** directive correspond with the musical notes in the **Notes DATA** directive.

Here is a **FOR...NEXT** loop that places the **Notes DATA** into a variable named **noteLetter**, then it places the **Frequencies DATA** into a variable named **noteFreq**.

```
FOR index = 0 to 6
  READ Notes + index, noteLetter
  READ Frequencies + (index * 2), Word noteFreq
  DEBUG noteLetter, " ", DEC noteFreq, CR
NEXT
```

8



What does the (index * 2) do? Each value stored in the **Frequencies DATA** directive takes a word (two bytes), while each character in the **Notes DATA** directive only takes one byte. The value of **index** increases by one each time through the **FOR...NEXT** loop. That's fine for accessing the note characters using the command **READ Notes + index, noteLetter**. The problem is that for every one byte in **Notes**, the **index** variable needs to point twice as far down the **Frequencies** list. The command **READ Frequencies + (index * 2), Word noteFreq**, takes care of this.

The next example program stores notes and durations using **DATA**, and it uses the **FREQOUT** command to play each note frequency for a specific duration. The result is the first few notes from the children's song Twinkle Twinkle Little Star.



The **Alphabet Song** used by children to memorize their "ABCs" uses the same notes as Twinkle Twinkle Little Star.

Example Program: TwinkleTwinkle.bs2

This example program demonstrates how to use the **DATA** directive to store lists and how to use the **READ** command to access the values in the lists.

√ Enter and run TwinkleTwinkle.bs2

- ✓ Verify that the notes sound like the song Twinkle Twinkle Little Star.
- ✓ Use the Debug Terminal to verify that it works as expected by accessing and displaying values from the three **DATA** directives.

```
' What's a Microcontroller - TwinkleTwinkle.bs2
' Play the first seven notes from Twinkle Twinkle Little Star.

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

Notes          DATA  "C","C","G","G","A","A","G"

Frequencies    DATA  Word 2093, Word 2093, Word 3136, Word 3136,
                    Word 3520, Word 3520, Word 3136

Durations      DATA  Word 500, Word 500, Word 500, Word 500,
                    Word 500, Word 500, Word 1000

index          VAR    Nib
noteLetter     VAR    Byte
noteFreq       VAR    Word
noteDuration   VAR    Word

DEBUG  "Note  Duration  Frequency", CR,
      "----  -"

FOR index = 0 TO 6

  READ Notes + index, noteLetter
  DEBUG "  ", noteLetter

  READ Durations + (index * 2), Word noteDuration
  DEBUG "      ", DEC4 noteDuration

  READ Frequencies + (index * 2), Word noteFreq
  DEBUG "          ", DEC4 noteFreq, CR

  FREQOUT 9, noteDuration, noteFreq

NEXT

END
```

Your Turn – Adding and Playing More Notes

This program played the first seven notes from Twinkle Twinkle Little Star. The words go “Twin-kle twin-kle lit-tle star”. The next phrase from the song goes “How I won-der

what you are”, and its notes are F, F, E, E, D, D, C. As with the first phrase the last note is held twice as long as the other notes. To add this phrase to the song from TwinkleTwinkle.bs2, you will need to expand each **DATA** directive appropriately. Don’t forget to change the **FOR...NEXT** loop so that it goes from 0 to 13 instead of from 0 to 6.

- ✓ Modify TwinkleTwinkle.bs2 so that it plays the first two phrases of the song instead of just the first phrase.

ACTIVITY #4: MICROCONTROLLER MUSIC

Note durations are not recorded on sheet music in terms of milliseconds. Instead, they are described as whole, half, quarter, eighth, sixteenth, and thirty-second notes. As the name suggests, a half note lasts half as long as a whole note. A quarter note lasts one fourth the time a whole note lasts, and so on. How long is a whole note? It depends on the piece of music being played. One piece might be played at a tempo that causes a whole note to last for four seconds, another piece might have a two second whole note, and yet another might have some other duration.

Rests are the time between notes when no tones are played. Rest durations are also measured as whole, half, quarter, eighth, sixteenth and thirty-second.

8

A Better System for Storing and Retrieving Music

You can write programs that store twice as much music in your BASIC Stamp by using bytes instead of words in your **DATA** directives. You can also modify your program to make the musical notes easier to read by using some of the more common musical conventions for notes and duration. This activity will start by introducing how to store musical information in a way that relates to the concepts of notes, durations, and rests. Tempo is also introduced, and it will be revisited in the next activity.

Here is an example of the **DATA** directives that stores musical notes and durations for the next example program. When played, it should resemble the song “Frere Jacques”. Only the note characters are stored in the **Notes DATA** directive because **LOOKUP** and **LOOKDOWN** commands will be used to match up letters to their corresponding frequencies.

Notes	DATA	"C", "D", "E", "C", "C", "D", "E", "C", "E", "F", "G", "E", "F", "G", "Q"
Durations	DATA	4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 2, 4, 4, 2

WholeNote

CON

2000

The first number in the **Durations DATA** directive tells the program how long the first note in the **Notes Data** directive should last. The second duration is for the second note, and so on. The durations are no longer in terms of milliseconds. Instead, they are much smaller numbers that can be stored in bytes, so there is no **Word** prefix in the **DATA** directive. Compared to storing values in terms of milliseconds, these numbers are more closely related to sheet music. Here is a list of what each duration means.

- 1 – whole note
- 2 – half note
- 4 – quarter note
- 8 – eighth note
- 16 – sixteenth note
- 32 – thirty-second note

After each value is read from the **Durations DATA** directive, it is divided into the **WholeNote** value to get the **Duration** used in the **FREQOUT** command. The amount of time each note lasts depends on the tempo of the song. A faster tempo means each note lasts for less time, while a slower tempo means each note lasts longer. Since all the note durations are fractions of a whole note, you can use the duration of a whole note to set the tempo.



What does the "Q" in Notes DATA mean? "Q" is for quit, and a **DO WHILE...LOOP** checks for "Q" each time through the loop.

How do I play a rest? You can insert a rest between notes by inserting a **"P"**. The Your Turn section has the first few notes from Beethoven's 5th Symphony, which has a rest in it.

How do I play sharp/flat notes? NotesAndDurations.bs2 has values in its lookup tables for sharp/flat notes. When you use the lower-case version of the note, it will play the flat note. For example, if you want to play B-flat, use "b" instead of "B". Remember that this is the same frequency as A-sharp.

Example Program: NotesAndDurations.bs2

- ✓ Enter and run NotesAndDurations.bs2.
- ✓ How does it sound?

' What's a Microcontroller - NotesAndDurations.bs2


```

' Play the first few notes from Frere Jacques.

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

DEBUG "Program Running!"

Notes          DATA    "C","D","E","C","C","D","E","C","E","F",
                        "G","E","F","G","Q"

Durations      DATA    4,  4,  4,  4,  4,  4,  4,  4,  4,  4,
                        2,  4,  4,  2

WholeNote      CON      2000

index          VAR      Byte
offset         VAR      Nib

noteLetter     VAR      Byte
noteFreq       VAR      Word
noteDuration   VAR      Word

DO UNTIL noteLetter = "Q"

    READ Notes + index, noteLetter

    LOOKDOWN noteLetter, [  "A",  "b",  "B",  "C",  "d",
                           "D",  "e",  "E",  "F",  "g",
                           "G",  "a",  "P",  "Q"    ], offset

    LOOKUP offset,          [ 1760, 1865, 1976, 2093, 2217,
                             2349, 2489, 2637, 2794, 2960,
                             3136, 3322,   0,   0    ], noteFreq

    READ Durations + index, noteDuration

    noteDuration = WholeNote / noteDuration

    FREQOUT 9, noteDuration, noteFreq

    index = index + 1

LOOP

END

```

How NotesAndDurations.bs2 Works

The **Notes** and **Durations DATA** directives were discussed before the program. These directives combined with the **wholeNote** constant are used to store all the musical data used by the program.

The declarations for the five variables used in the program are shown below. Even though a **FOR...NEXT** loop is no longer used to access the data, there still has to be a variable (**index**) that keeps track of which **DATA** entry is being read in **Notes** and **Durations**. The **offset** variable is used in the **LOOKDOWN** and **LOOKUP** commands to select a particular value. The **noteLetter** variable stores a character accessed by the **READ** command. **LOOKUP** and **LOOKDOWN** commands are used to convert this character into a frequency value. This value is stored in the **noteFreq** variable and used as the **FREQOUT** command's **Freq1** argument. The **noteDuration** variable is used in a **READ** command to receive a value from the **Durations DATA**. It is also used to calculate the **Duration** used in the **FREQOUT** command.

```

index          VAR   Byte
offset         VAR   Nib

noteLetter     VAR   Byte
noteFreq       VAR   Word
noteDuration   VAR   Word

```

The main loop keeps executing until the letter 'Q' is read from the **Notes DATA**.

```
DO UNTIL noteLetter = "Q"
```

A **READ** command gets a character from the **Notes DATA**, and stores it in the **noteLetter** variable. The **noteLetter** variable is then used in a **LOOKDOWN** command to set the value of the **offset** variable. Remember that **offset** stores a 1 if "b" is detected, a 2 if "B" is detected, a 3 if "C" is detected, and so on. This **offset** value is then used in a **LOOKUP** command to figure out what the value of the **noteFreq** variable should be. If **offset** is 1, **noteFreq** will be 1865, if **offset** is 2, **noteFreq** will be 1976, if **offset** is 3, **noteFreq** is 2093, and so on.

```

READ Notes + index, noteLetter

LOOKDOWN noteLetter, [  "A",  "b",  "B",  "C",  "d",
                        "D",  "e",  "E",  "F",  "g",
                        "G",  "a",  "P",  "Q" ], offset

LOOKUP offset,          [ 1760, 1865, 1976, 2093, 2217,

```

```
2349, 2489, 2637, 2794, 2960,
3136, 3322, 0, 0 ], noteFreq
```

The note's frequency has been determined, but the duration still has to be figured out. The **READ** command uses the value of **index** to place a value from the **Durations DATA** into **noteDuration**.

```
READ Durations + index, noteDuration
```

Then, **noteDuration** is set equal to the **WholeNote** constant divided by the **noteDuration**. If note duration starts out as 4 from a **READ** command, it becomes $2000 \div 4 = 500$. If **noteDuration** is 8, it becomes $1500 \div 8 = 250$.

```
noteDuration = WholeNote / noteDuration
```

Now that **noteDuration** and **noteFreq** are determined, the **FREQOUT** command plays the note.

```
FREQOUT 9, noteDuration, noteFreq
```

Each time through the main loop, the **index** value must be increased by one. When the main loop gets back to the beginning, the first thing the program does is read the next note, using the **index** variable.

```
index = index + 1
```

```
LOOP
```

8

Your Turn – Experimenting with Tempo and a Different Tune

The length of time that each note lasts is related to the tempo. You can change the tempo by adjusting the **WholeNote** constant. If you increase it to 2250, the tempo will decrease, and the song will play slower. If you increase it to 1750, the tempo will increase and the song will play more quickly.

- ✓ Save NotesAndDurations.bs2 under the name NotesAndDurationsYourTurn.bs2.
- ✓ Modify the tempo of NotesAndDurationsYourTurn.bs2 by adjusting the value of **WholeNote**. Try values of 1500, 1750, 2000, and 2250.
- ✓ Re-run the program after each modification, and decide which one sounds best.

Entering musical data is much easier when all you have to do is record notes and durations. Here are the first eight notes from Beethoven's Fifth Symphony.

```

Notes      DATA  "G", "G", "G", "e", "P", "F", "F", "F", "D", "Q"

Durations  DATA   8,  8,  8,  2,  8,  8,  8,  8,  2

WholeNote  CON    2000

```

- ✓ Save your modified program Beethoven'sFifth.bs2.
- ✓ Replace the **Notes** and **Durations DATA** directives and the **WholeNote** constant declaration with the code above.
- ✓ Run the program. Does it sound familiar?

Adding Musical Features

The example program you just finished introduced notes, durations, and rests. It also used the duration of a whole note to determine tempo. Cell phones that play music to let you know that somebody is calling have three features that were not supported by the previous example program:

- They play “dotted” notes.
- They determine the whole note duration from a value called tempo.
- They play notes from more than one octave.

The term “dotted” refers to a dot used in sheet music to indicate that a note should be played 1 ½ times as long as its normal duration. For example, a dotted quarter note should last for the duration of a quarter note, plus an eighth note. A dotted half note lasts for a half plus a quarter note's duration. You can add a data table that stores whether or not a note is dotted. In this example, a zero means there is no dot while a 1 means there is a dot:

```

Dots          DATA      0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,
                        0,  0,  0,  1,  0

```

Cell phones typically take the tempo for a song in beats per minute. This is the same as saying quarter notes per minute.

```

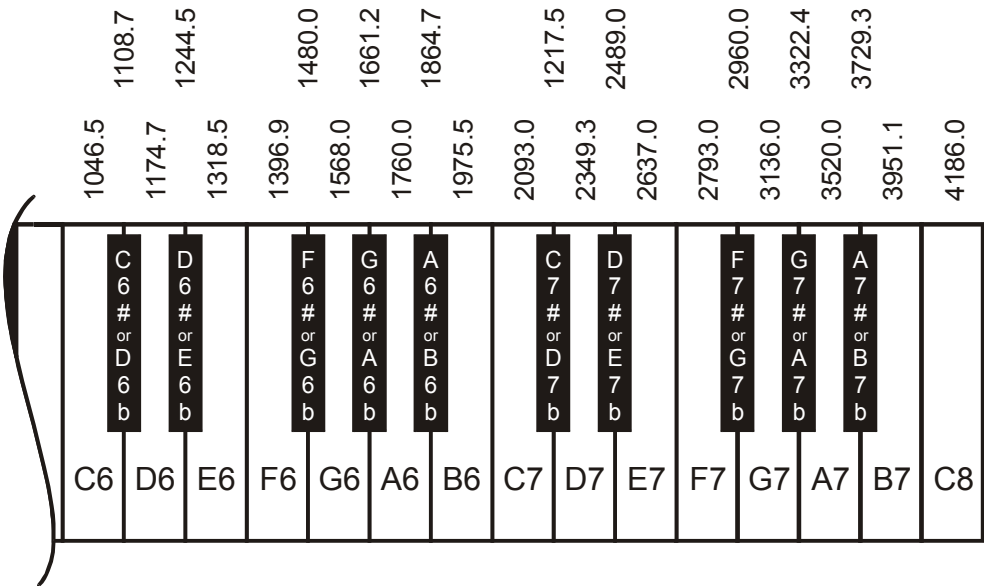
BeatsPerMin   CON    200

```

Figure 8-4 is a repeat of **Figure 8-3** from page 228. It shows the 6th and 7th octaves on the piano keyboard. These are the two octaves that sound the clearest when played by the piezospeaker. Here is an example of a **DATA** directive you will use in the Your Turn section to play notes from more than one octave using the **Notes DATA** directive.

Octaves	DATA	6,	7,	6,	6,	6,	6,	6,	6,	7,	6,
		6,	6,	6							

Figure 8-4
Rightmost Piano Keys and Their Frequencies



Example Program: MusicWithMoreFeatures.bs2

This example program plays the first few notes from For He’s a Jolly Good Fellow. All the notes come from the same (7th) octave, but some of the notes are dotted. In the Your turn section, you will try an example that uses notes from more than one octave, and dotted notes.

- ✓ Enter and run MusicWithMoreFeatures.bs2.
- ✓ Count notes and see if you can hear the dotted (1 ½ duration) notes.
- ✓ Also listen for notes in octave 7. Try changing one of those notes to octave 6. The change in the way the music sounds is pretty drastic.

```
' What's a Microcontroller - MusicWithMoreFeatures.bs2
' Play the beginning of For He's a Jolly Good Fellow.
```

```

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

DEBUG "Program Running!"

Notes          DATA    "C","E","E","E","D","E","F","E","E","D","D",
                        "D","C","D","E","C","Q"
Octaves        DATA    7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
                        7, 7, 7, 7, 7
Durations      DATA    4, 2, 4, 4, 4, 4, 2, 2, 4, 2, 4,
                        4, 4, 4, 2, 2
Dots           DATA    0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                        0, 0, 0, 1, 0

BeatsPerMin    CON      320

index          VAR      Byte
offset         VAR      Nib

noteLetter     VAR      Byte
noteFreq       VAR      Word
noteDuration   VAR      Word
noteOctave     VAR      Nib
noteDot        VAR      Bit

wholeNote      VAR      Word
wholeNote = 60000 / BeatsPerMin * 4

DO UNTIL noteLetter = "Q"

    READ Notes + index, noteLetter

    LOOKDOWN noteLetter, [ "C", "d", "D", "e", "E",
                          "F", "g", "G", "a", "A",
                          "b", "B", "P", "Q" ], offset

    LOOKUP offset, [ 4186, 4435, 4699, 4978, 5274,
                    5588, 5920, 6272, 6645, 7040,
                    7459, 7902, 0, 0 ], noteFreq

    READ Octaves + index, noteOctave
    noteOctave = 8 - noteOctave
    noteFreq = noteFreq / (DCD noteOctave)

    READ Durations + index, noteDuration
    noteDuration = wholeNote / noteDuration

    READ Dots + index, noteDot
    IF noteDot = 1 THEN noteDuration = noteDuration * 3 / 2

```

```
FREQOUT 9, noteDuration, noteFreq

    index = index + 1

LOOP

END
```

How MusicWithMoreFeatures.bs2 Works

Below is the musical data for the entire song. For each note in the **Notes DATA** directive, there is a corresponding entry in the **Octaves**, **Durations**, and **Dots DATA** directives. For example, the first note is a C note in the 7th octave; it's a quarter note and it's not dotted. Here is another example: the second from the last note (not including "Q") is an E note, in the 7th octave. It's a half note, and it is dotted. There is also a **BeatsPerMin** constant that sets the tempo for the song.

Notes	DATA	"C", "E", "E", "E", "D", "E", "F", "E", "E", "D", "D", "D", "C", "D", "E", "C", "Q"
Octaves	DATA	7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7
Durations	DATA	4, 2, 4, 4, 4, 4, 2, 2, 4, 2, 4, 4, 4, 4, 2, 2
Dots	DATA	0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0
BeatsPerMin	CON	320



In the previous example program, **WholeNote** was a constant. This time, it's a variable that will hold the duration of a whole note in ms. After this value is calculated, **WholeNote** will be used to determine all the other note durations, just like in the previous program. The **index**, **offset**, **noteLetter**, and **noteDuration** variables are also used in the same manner they were in the previous program. The **noteFreq** variable is handled a little differently since now it has to be adjusted depending on the octave the note is played in. The **noteOctave** and **noteDot** variables have been added to handle the octave and dot features.

wholeNote	VAR	Word
index	VAR	Byte
offset	VAR	Nib
noteLetter	VAR	Byte
noteFreq	VAR	Word
noteDuration	VAR	Word

```

noteOctave    VAR  Nib
noteDot       VAR  Bit

```

The **wholeNote** variable is calculated using the **BeatsPerMin**. The tempo of the song is defined in beats per minute, and the program has to divide **BeatsPerMin** into 60000 ms, then multiply by 4. The result is the correct value for a whole note.

```
wholeNote = 60000 / BeatsPerMin * 4
```



Math executes from left to right. In the calculation `wholeNote = 60000 / beatsPerMin * 4`, the BASIC Stamp first calculates `60000 / beatsPerMin`. Then, it multiplies that result by 4.

Parentheses can be used to group operations. If you want to divide 4 into `beatsPerMin` first, you can do this: `wholeNote = 60000 / (beatsPerMin * 4)`.

This is all the same as the previous program:

```

DO UNTIL noteLetter = "Q"

  READ Notes + index, noteLetter

  LOOKDOWN noteLetter, [ "C", "d", "D", "e", "E",
                        "F", "g", "G", "a", "A",
                        "b", "B", "P", "Q" ], offset

```

Now that octaves are in the mix, the part of the code that figures out the note frequency has changed. The **LOOKUP** command's table of values contains note frequencies from the 8th octave. These values can be divided by 1 if you want to play notes in the 8th octave, by 2 if you want to play notes in the 7th octave, by 4 if you want to play notes in the 6th octave and by 8 if you want to play notes in the 5th octave. The division happens next. All this **LOOKUP** command does is place a note from the 8th octave into the **noteFreq** variable.

```

LOOKUP offset, [ 4186, 4435, 4699, 4978, 5274,
                 5588, 5920, 6272, 6645, 7040,
                 7459, 7902, 0, 0 ], noteFreq

```

Here is how the **noteFreq** variable is adjusted for the correct octave. First, the **READ** command grabs the octave value stored in the **Octaves DATA**. This could be a value between 5 and 8.

```
READ Octaves + index, noteOctave
```


Depending on the octave, we want to divide `noteFreq` by either 1, 2, 4, or 8. That means that the goal is really to divide by $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, or $2^3 = 8$. The statement below takes the value of `noteOctave`, which could be a value between 5 and 8, and subtracts that value from 8. If `noteOctave` was 8, now it's 0. If `noteOctave` was 7, now it's 1. If `noteOctave` was 6, now it's 2, and if `noteOctave` was 5, now it's 3.

```
noteOctave = 8 - noteOctave
```

Now, `noteOctave` is a value that can be used as an exponent of 2, but how do you raise 2 to a power in PBASIC? One answer is to use the `DCD` operator. `DCD 0` is 1, `DCD 1` is 2, `DCD 2` is 4, and `DCD 3` is 8. Dividing `noteFreq` by `DCD noteOctave` means you are dividing by 1, 2, 4, or 8, which divides `noteFreq` down by the correct value. The end result is that `noteFreq` is set to the correct octave. You will use the Debug Terminal in the Your Turn section to take a closer look at how this works.

```
noteFreq = noteFreq / (DCD noteOctave)
```



How am I supposed to know to use the DCD operator? Keep learning and practicing. Every time you see a new command, operator, or any other keyword used in an example, look it up in the BASIC Stamp manual. Read about it, and try using it in a program of your own design. Get in the habit of periodically reading the BASIC Stamp Manual and trying the short example programs. That's the best way to get familiar with the various commands and operators and how they work. By doing these things, you will develop a habit of always adding to the list of programming tools you can use to solve problems.

8

The first two lines of code for determining the note duration are about the same as the code from the previous example program. Now, however, any note could be dotted, which means the duration might have to be multiplied by 1.5. A `READ` command is used to access values stored in `EEPROM` by the `Dots DATA` directive. An `IF...THEN` statement is used to multiply by 3 and divide by 2 whenever the value of the `noteDot` variable is 1.

```
READ Durations + index, noteDuration
noteDuration = WholeNote / noteDuration

READ Dots + index, noteDot
IF noteDot = 1 THEN noteDuration = noteDuration * 3 / 2
```



Integer math The BASIC Stamp does not automatically process a number like 1.5. When performing math, it only works with integers: ..., -5, -4, -3, -2, -1, 0, 1, 2, 3, ... The best solution for multiplying by 1.5 is to multiply by 3/2. First, multiply by 3, and then divide by 2.

There are many ways to program the BASIC Stamp to handle fractional values. You can program the BASIC Stamp to use integers to figure out the fractional portion of a number. This is introduced in the *Basic Analog and Digital* Student Guide. There are also two operators that make fractional values easier to work with, and they are: ****** and ***/**. These are explained in detail in the *Applied Sensors* Student Guide and in the BASIC Stamp Manual.

The remainder of this example program works the same way that it did in the previous example program:

```
FREQOUT 9, noteDuration, noteFreq
index = index + 1
LOOP
END
```

Your Turn – Playing a Tune with More than One Octave

MusicWithMoreFeatures.bs2 made use of rests, but it stayed in one octave. The tune “Take Me Out to the Ball Game” shown below plays most of its notes in the 6th octave. There are two notes in the 7th octave, and they make a big difference to the way it sounds.

- ✓ Save the example program under the name MusicWithMoreFeaturesYourTurn.bs2.
- ✓ Modify the program by replacing the four data directives and one constant declaration with these:

Notes	DATA	"C", "C", "A", "G", "E", "G", "D", "P", "C", "C", "A", "G", "E", "G", "Q"
Octaves	DATA	6, 7, 6, 6, 6, 6, 6, 6, 7, 6, 6, 6, 6
Durations	DATA	2, 4, 4, 4, 4, 2, 2, 4, 2, 4, 4, 4, 4, 2
Dots	DATA	0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1
BeatsPerMin	CON	240

- ✓ Run the program and verify that it sounds right.

Those two notes in the 7th octave are essential for making the tune sound right. It's interesting to hear what happens if those 7 values are changed to 6.

- ✓ Try changing the two 7 values in the **Octaves DATA** directive so that they are 6. Keep in mind, this will make “Take Me out to the Ball Game” sound weird.
- ✓ Run the program, and listen to the effect of the wrong octaves on the song.
- ✓ Change the **Octaves DATA** back to its original state.
- ✓ Run the program again and listen to see if it sounds correct again.

ACTIVITY #5: CELL PHONE RINGTONES

Many cell phones play music that can be downloaded from web pages. The computer sends the data about the notes to the microcontroller in the phone, then plays the notes whenever a call comes in. These are called ringing tones or often just ringtones.

One of the most widely used way of composing, recording and posting notes is one that features strings of text that describe each note in the song. Here is an example of how the first few notes from Beethoven's 5th look in RTTTL format:

```
Beethoven5:d=8,o=7,b=125:g,g,g,2d#,p,f,f,f,2d
```

This format for storing musical data is called RTTTL, which stand for Ringing Tone Text Transfer Language. The great thing about RTTTL files is that they are widely shared via the World Wide Web. Many sites have RTTTL files available for free download. There are also free software programs that can be used to compose and emulate these files as well as download them to your cell phone. The RTTTL specification is also published on the World Wide Web. Appendix G summarizes how an RTTTL file stores notes, durations, pauses, tempo, and dotted notes.

This activity introduces some PBASIC programming techniques that can be used to recognize different elements of text. The ability to recognize different characters or groups of characters and take action based on what those characters contain is extremely useful. In fact, it's the key to converting RTTTL format ringtone (like Beethoven5 above) into music. At the end of this activity, there is an application program that you can use to play RTTTL format ringtones.

Selecting which Code Block to Execute on a Case by Case Basis

The **SELECT...CASE** statement is probably the best programming tool for recognizing characters or values. Keep in mind that this is one of the tools used to convert an RTTTL ringtone into musical notes. In general, **SELECT...CASE** is used to:

- Select a variable or expression.
- Evaluate that variable or expression on a case by case basis.
- Execute different blocks of code depending on which case that variable's value fits into.

Here is the syntax for **SELECT...CASE**:

```
SELECT expression
CASE condition(s)
  statement(s)
ENDSELECT
```

You can try the next two example programs to see how **SELECT...CASE** works. `SelectCaseWithValues.bs2` takes numeric values you enter into the Debug Terminal and it tells you the minimum variable size you will need to hold that value. `SelectCaseWithCharacters.bs2` tells you whether the character you entered into the Debug Terminal is upper or lower case, a digit, or punctuation.

Remember to use the upper Windowpane in the Debug Terminal to transmit the characters you type to the BASIC Stamp. The Transmit and Receive Windowpanes are shown in Figure 8-5.

Windowpanes

Transmit →

Receive →

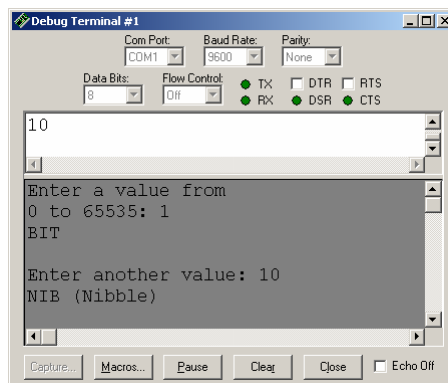


Figure 8-5
Sending Messages
to the BASIC Stamp

Click the Transmit (upper) Windowpane and enter the value or characters you want to transmit to the BASIC Stamp.

Example Program: SelectCaseWithValues.bs2

- ✓ Enter and run SelectCaseWithValues.bs2.
- ✓ Click the Debug Terminal's Transmit Windowpane.
- ✓ Enter a value between 0 and 65535, and press the *Enter* key.



What happens if you enter a number larger than 65535? If you enter the number 65536, the BASIC Stamp will store the number 0. If you enter the number 65537, the BASIC Stamp will store the number 1, and so on. When a number is too large for the variable it fits into, it is called overflow.

- ✓ Use Table 2-2 to verify that the example program makes the right decisions about the size of the numbers you enter into the Debug Terminal.

Table 2-2: Variable Types and Values They Can Store	
Variable type	Range of Values
Bit	0 to 1
Nib	0 to 15
Byte	0 to 255
Word	0 to 65535

8

```
' What's a Microcontroller - SelectCaseWithValues.bs2
' Enter a value and see the minimum variable size required to hold it.

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

value          VAR      Word

DEBUG "Enter a value from", CR,
      "0 to 65535: "

DO

  DEBUGIN DEC value

  SELECT value

    CASE 0, 1
      DEBUG "Bit", CR
      PAUSE 100

    CASE 2 TO 15
      DEBUG "Nib (Nibble)", CR
```

```

    PAUSE 200

    CASE 16 TO 255
        DEBUG "Byte", CR
        PAUSE 300

    CASE 256 TO 65535
        DEBUG "Word", CR
        PAUSE 400

    ENDSELECT

    DEBUG CR, "Enter another value: "

LOOP

```

How SelectCaseWithValues.bs2 Works

A word variable is declared to hold the values entered into the Debug Terminal.

```
value VAR Word
```

The **DEBUGIN** command takes the number you enter and places it into the **value** variable.

```
DEBUGIN DEC value
```

The **SELECT** statement chooses the **value** variable as the one to evaluate cases for.

```
SELECT value
```

The first case is if the **value** variable equals either 0 or 1. If **value** equals either of those numbers, the **DEBUG** and **PAUSE** commands that follow it are executed.

```

    CASE 0, 1
        DEBUG "BIT", CR
        PAUSE 100

```

The second case is if **value** equals any number from 2 to 15. If it does equal any of those numbers, the **DEBUG** and **PAUSE** commands below it are executed.

```

    CASE 2 to 15
        DEBUG "NIB (Nibble)", CR
        PAUSE 200

```

When all the cases are done, the **ENDSELECT** keyword is used to complete the **SELECT..CASE** statement.

```
ENDSELECT
```

Example Program: SelectCaseWithCharacters.bs2

This example program evaluates each character you enter into the Debug Terminal's Transmit Windowpane. It can recognize upper and lower case characters, digits, and some punctuation. If you enter a character the program does not recognize, it will tell you to try again (entering a different character).

- ✓ Enter and run SelectCaseWithCharacters.bs2.
- ✓ Click Debug Terminal's Transmit Windowpane, enter characters and observe the results.

```
' What's a Microcontroller - SelectCaseWithCharacters.bs2
' Program that can identify some characters: case, digit, punctuation.

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

character          VAR      Byte

DEBUG "Enter a character: ", CR

DO

  DEBUGIN character

  SELECT character

    CASE "A" TO "Z"
      DEBUG CR, "Upper case", CR

    CASE "a" TO "z"
      DEBUG CR, "Lower case", CR

    CASE "0" TO "9"
      DEBUG CR, "Digit", CR

    CASE "!", "?", ".", ",", "
      DEBUG CR, "Punctuation", CR

    CASE ELSE
      DEBUG CR, "Character not known.", CR,
        "Try a different one."

  ENDSELECT

  DEBUG CR, "Enter another character", CR

LOOP
```

How SelectCaseWithCharacters.bs2 Works

When compared to SelectCaseWithValues.bs2, this example program has a few differences. First, the name of the **value** variable was changed to **character**, and its size was changed from word to byte. This is because all characters in PBASIC are byte size. The **SELECT** statement chooses the **character** variable for case by case evaluation.

```
SELECT character
```

The quotation marks are used to tell the BASIC Stamp Editor that you are referring to characters.

```
SELECT character

CASE "A" to "Z"
  DEBUG CR, "Upper case", CR

CASE "a" to "z"
  DEBUG CR, "Lower case", CR

CASE "0" to "9"
  DEBUG CR, "Digit", CR

CASE "!", "?", ".", ",", " "
  DEBUG CR, "Punctuation", CR
```

There is also one different **CASE** statement that was not used in the previous example:

```
CASE ELSE
  DEBUG CR, "Character not known.", CR,
    "Try a different one."
```

This **CASE** statement tells the **SELECT** code block what to do if none of the other cases are true. You can get this case to work by entering a character such as % or \$.

Your Turn – Selecting Based on Ranges

- ✓ Modify the **SELECT...CASE** statement in SelectCaseWithCharacters.bs2 so that it displays “Special character” when you enter one of these characters: @, #, \$, %, '^', &, *, (,), _, or +.

RTTTL Ringtone Player Application Program

Below is the RTTTL file that contains the musical information used in the next example program. There are five more **RTTTL_File DATA** directives that you can try in the Your

Turn section. This program plays a tune called Reveille, which is the bugle call played at military camps first thing in the morning. You may have heard it in any number of movies or television shows.

```
RTTTL_File      DATA      "Reveille:d=4,o=7,b=140:8g6,8c,16e,16c,8g6,8e,",
                             "8c,16e,16c,8g6,8e,8c,16e,16c,8a6,8c,e,8c,8g6,",
                             "8c,16e,16c,8g6,8e,8c,16e,16c,8g6,8e,8c,16e,",
                             "16c,8g6,8e,c,p,8e,8e,8e,8e,g,8e,8c,8e,8c,8e,8c,",
                             "e,8c,8e,8e,8e,8e,8e,g,8e,8c,8e,8c,8g6,8g6,c."
```

Example Program: MicroMusicWithRtttl.bs2

This application program is pretty long, and it's a good idea to download the latest version from the www.parallax.com → Downloads → Educational Curriculum page. Look for a link named *Selected Example Programs* near the What's a Microcontroller PDF downloads. Downloading the program and opening it with the BASIC Stamp Editor should save you a significant amount of time. The alternative, of course, is to hand enter and debug four pages of code.

- ✓ Use the BASIC Stamp Editor to open your downloaded MicroMusicWithRtttl.bs2 file, or hand enter the example below very carefully.
- ✓ Run the program, and verify that the piece is recognizable as the Reveille bugle call.
- ✓ Go to the Your Turn section and try some more tunes (RTTTL_File DATA directives).

8

```
' What's a Microcontroller - MicroMusicWithRtttl.bs2
' Play Nokia RTTTL format ringtones using DATA.

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

' -----[ I/O Definitions ]-----

SpeakerPin      CON      9                      ' Piezospeaker connected to P9.

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

counter          VAR      Word                    ' General purpose counter.
char              VAR      Byte                    ' Variable stores characters.
index             VAR      Word                    ' Index for pointing at data.

noteLetter        VAR      Byte                    ' Stores note character.
noteFreq          VAR      Word                    ' Stores note frequency.
```

```

noteOctave      VAR      Word      ' Stores note octave.

duration        VAR      Word      ' Stores note duration.
tempo           VAR      Word      ' Stores tempo.

default_d       VAR      Byte      ' Stores default duration.
default_o       VAR      Byte      ' Stores default octave.
default_b       VAR      Word      ' Stores default beats/min.

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

RTTTL_File      DATA      "Reveille:d=4,o=7,b=140:8g6,8c,16e,16c,8g6,8e,",
                    "8c,16e,16c,8g6,8e,8c,16e,16c,8a6,8c,e,8c,8g6,",
                    "8c,16e,16c,8g6,8e,8c,16e,16c,8g6,8e,8c,16e,",
                    "16c,8g6,8e,c,p,8e,8e,8e,8e,g,8e,8c,8e,8c,8e,8c,",
                    "e,8c,8e,8e,8e,8e,8e,g,8e,8c,8e,8c,8g6,8g6,c."

Done            DATA      ",q,"

Notes           DATA      "p",      "a",      "#",      "b",
                    "c",      "#",      "d",      "#",
                    "e",      "f",      "#",      "g",
                    "#"

Octave8         DATA      Word 0,      Word 3520, Word 3729, Word 3951,
                    Word 4186, Word 4435, Word 4699, Word 4978,
                    Word 5274, Word 5588, Word 5920, Word 6272,
                    Word 6645

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

counter = 0      ' Initialize counter.

GOSUB FindEquals ' Find first '=' in file.
GOSUB ProcessDuration ' Get default duration.
GOSUB FindEquals ' Find next '='.
GOSUB ProcessOctave ' Get default octave.
GOSUB FindEquals ' Find last '='.
GOSUB GetTempo ' Get default tempo.

' ----- [ Program Code ] -----

DO UNTIL char = "q" ' Loop until 'q' in DATA.
    GOSUB ProcessDuration ' Get note duration.
    GOSUB ProcessNote ' Get index value of note.
    GOSUB CheckForDot ' If dot, 3/2 duration.
    GOSUB ProcessOctave ' Get octave.
    GOSUB PlayNote ' Get freq, play note, next.
LOOP ' End of main loop.

END ' End of program.

```

```

' -----[ Subroutine - Find Equals Character ]-----

FindEquals:                                ' Go through characters in
                                           ' RTTTL file looking for
DO                                           ' '='. Increment counter
    READ RTTTL_File + counter, char        ' until '=' is found, then
    counter = counter + 1                  ' return.
LOOP UNTIL char = "="

RETURN

' -----[ Subroutine - Read Tempo from RTTTL Header ]-----
' Each keyboard character has a unique number called an ASCII value.
' The characters 0, 1, 2,...9 have ASCII values of 48, 49, 50,...57.
' You can always convert from the character representing a digit to
' to its value by subtracting 48 from the variable storing the digit.
' You can examine this by comparing DEBUG DEC 49 and DEBUG 49.

GetTempo:                                  ' Parse RTTTL file for Tempo.
                                           ' Convert characters to
default_b = 0                             ' digits by subtracting 48
DO                                           ' from each character's ASCII
    READ RTTTL_File + counter, char        ' value. Iteratively multiply
    IF char = ":" THEN                    ' each digit by 10 if there
        default_b = default_b / 10        ' is another digit, then add
        counter = counter + 1             ' the most recent digit to
    EXIT                                  ' one's column.
ENDIF                                       ' For example, the string
    default_b = default_b + char - 48     ' "120" is (1 X 10 X 10)
    counter = counter + 1                 ' + (2 X 10) + 0. The '1'
    default_b = default_b * 10            ' is converted first, then
LOOP UNTIL char = ":"                     ' multiplied by 10. The '2'
                                           ' is then converted/added.
RETURN                                    ' 0 is converted/added, done.

' -----[ Subroutine - Look up Octave ]-----

ProcessOctave:                             ' Octave may or may not be
                                           ' included in a given note
                                           ' because any note that is
READ RTTTL_File + counter, char           ' played in the default
SELECT char                               ' octave does not specify
    CASE "5" TO "8"                      ' the octave. If a char
        noteOctave = char - "0"           ' from '5' to '8' then use
        counter = counter + 1             ' it, else use default_o.
    CASE ELSE                             ' Characters are converted
        noteOctave = default_o            ' to digits by subtracting
ENDSELECT                                 ' '0', which is the same as
IF default_o = 0 THEN                     ' subtracting 48. The first
    default_o = noteOctave                ' time this subroutine is
ENDIF                                     ' called, default_o is 0.

```

```

RETURN                                ' If 0, then set default_o.

' -----[ Subroutine - Find Index of Note ]-----

ProcessNote:                          ' Set index value for lookup
                                      ' of note frequency based on
READ RTTTL_File + counter, char       ' note character. If 'p',
SELECT char                           ' index is 0. If 'a' to 'g',
CASE "p"                              ' read character values in
    index = 0                          ' DATA table and find match.
    counter = counter + 1              ' Record index value when
CASE "a" TO "g"                        ' match is found. If next
    FOR index = 1 TO 12                ' char is a sharp (#), add
        READ Notes + index, noteLetter ' 1 to the index value to
        IF noteLetter = char THEN EXIT ' increase the index (and
    NEXT                               ' frequency) by 1 notch.
    counter = counter + 1              ' As with other subroutines,
    READ RTTTL_File + counter, char    ' increment counter for each
    SELECT char                        ' character that is processed.
    CASE "#"
        index = index + 1
        counter = counter + 1
    ENDSELECT
ENDSELECT

RETURN

' -----[ Subroutine - Determine Note Duration ]-----

ProcessDuration:                      ' Check to see if characters
                                      ' form 1, 2, 4, 8, 16 or 32.
READ RTTTL_File + counter, char       ' If yes, then convert from
                                      ' ASCII character to a value
SELECT char                           ' by subtracting 48. In the
CASE "1", "2", "3", "4", "8"          ' case of 16 or 32, multiply
    duration = char - 48               ' by 10 and add the next
    counter = counter + 1              ' digit to the ones column.
    READ RTTTL_File + counter, char
    SELECT char
    CASE "6", "2"
        duration = duration * 10 + char - 48
        counter = counter + 1
    ENDSELECT
CASE ELSE
    duration = default_d               ' If no duration, use
ENDSELECT                             ' use default.

IF default_d <> 0 THEN                  ' If default_d not defined
    duration = 60000/default_b/duration*3 ' (if default_d = 0), then
ELSE                                    ' set default_d = to the
    default_d = duration                ' duration from the d=#.

```

```

ENDIF

RETURN

' -----[ Subroutine - Check For '.' Indicating 1.5 Duration ]-----

CheckForDot:                                ' Check for dot indicating
                                           ' multiply duration by 3/2.
READ RTTTL_File + counter, char             ' If dot found, multiply by
SELECT char                                  ' 3/2 and increment counter,
CASE "."                                     ' else, do nothing and
    duration = duration * 3 / 2              ' return.
    counter = counter + 1
ENDSELECT

RETURN

' -----[ Subroutine - Find Comma and Play Note/Duration ]-----

PlayNote:                                    ' Find last comma in the
                                           ' current note entry. Then,
READ RTTTL_File + counter, char              ' fetch the note frequency
SELECT char                                  ' from data, and play it, or
CASE ","                                     ' pause if frequency = 0.
    counter = counter + 1
    READ Octave8 + (index * 2), Word noteFreq
    noteOctave = 8 - noteOctave
    noteFreq = noteFreq / (DCD noteOctave)
    IF noteFreq = 0 THEN
        PAUSE duration
    ELSE
        FREQOUT SpeakerPin, duration, noteFreq
    ENDIF
ENDSELECT

RETURN

```

How MicroMusicWithRtttl.bs2 Works

This example program is fun to use, and it shows the kind of code you will be able to write with some practice. However, it was included in this text more for fun than for the coding concepts it employs.

If you examine the code briefly, you might notice that you have already used all of the commands and operators in the program. Here is a list of the elements in this application example that should, by now, be familiar:

- Comments to help explain your code

- Constant and variable declarations
- **DATA** declarations
- **READ** commands
- **IF...ELSE...ENDIF** blocks
- **DO...LOOP** both with and without **WHILE** and **UNTIL**
- Subroutines with **GOSUB**, labels, and **RETURN**
- **FOR...NEXT** loops
- **LOOKUP** and **LOOKDOWN** commands
- The **FREQOUT** and **PAUSE** commands
- **SELECT...CASE**

Your Turn – Different Tunes

- √ Try replacing the **RTTTL_File DATA** directive in `MicroMusicWithRTTTL.bs2` with each of the five different music files below.



Only one RTTTL_File DATA directive at a time! Make sure to replace, not add, your new **RTTTL_File DATA** directive.

- √ Run `MicroMusicWithRTTTL.bs2` to test each RTTTL file.

RTTTL_File	DATA	"TwinkleTwinkle:d=4,o=7,b=120:c,c,g,g,a,a,2g,f," "f,e,e,d,d,2c,g,g,f,f,e,e,2d,g,g,f,f,e,e,2d,c,c," "g,g,a,a,2g,f,f,e,e,d,d,1c"
RTTTL_File	DATA	"FrereJacques:d=4,o=7,b=125:c,d,e,c,c,d,e,c,e,f," ",2g,e,f,2g,8g,8a,8g,8f,e,c,8g,8a,8g,8f,e,c,c,g6," ",2c,c,g6,2c"
RTTTL_File	DATA	"Beethoven5:d=8,o=7,b=125:g,g,g,2d#,p,f,f,f,2d"
RTTTL_File	DATA	"ForHe'sAJollyGoodFellow:d=4,o=7,b=320:c,2e,e,e," "d,e,2f.,2e,e,2d,d,d,c,d,2e.,2c,d,2e,e,e,d,e,2f," "g,2a,a,g,g,g,2f,d,2c"
RTTTL_File	DATA	"TakeMeOutToTheBallgame:d=4,o=7,b=225:2c6,c,a6," "g6,e6,2g.6,2d6,p,2c6,c,a6,g6,e6,2g.6,g6,p,p,a6," ",g#6,a6,e6,f6,g6,a6,p,f6,2d6,p,2a6,a6,a6,b6,c," "d,b6,a6,g6"



Downloading RTTTL Files: There are lots of RTTTL files available for download from various sites on the World Wide Web. These files are contributed by ring-tone enthusiasts, many of whom are not music experts. Some phone tones are pretty good, others are barely recognizable. If you want to download and play some more RTTTL files, make sure to remove any spaces from between characters, then insert the text file between quotes.

SUMMARY

This chapter introduced techniques for making sounds and musical tones using the BASIC Stamp and a piezoelectric speaker. The **FREQOUT** command can be used to send a piezoelectric speaker high/low signals that cause it to make sound effects and/or musical notes. The **FREQOUT** command has arguments that control the I/O Pin the signal is sent to, the *Duration* of the tone, the frequency of the tone (*Freq1*). The optional *Freq2* argument can be used to mix tones.

Sound effects can be made by adjusting the frequency and duration of tones and the pauses between. The value of the frequency can also be swept across a range of values or mixed to create a variety of effects.

Making musical notes also depends on frequency, duration, and pauses. The value of the **FREQOUT** command's *Duration* argument is determined by the tempo of the song and the duration of the note (whole, half, quarter, etc.). The *Freq1* value of the note is determined by the note's letter and octave. Rests between notes are used to set the duration of the **PAUSE** command.

Playing simple songs using the BASIC Stamp can be done with a sequence of **FREQOUT** commands, but there are better ways to store and retrieve musical data. **DATA** directives along with their optional *Symbol* labels were used to store byte values using no prefix and word values using the **Word** prefix. The **READ** command was used to retrieve values stored by **DATA** directives. The **READ** command's *Address* argument always used the **DATA** directive's optional *Symbol* label to differentiate between different types of data. Some the symbol labels that were used were **Notes**, **Durations**, **Dots**, and **Octaves**.

Musical data can be stored in formats that lend themselves to translation from sheet music. The sheet music style data can then be converted into **Frequency** using the **LOOKUP** and **LOOKDOWN** commands. Mathematic operations can also be performed on variable values to change the octave of a note by dividing its frequency by a power of two. Mathematic operations are also useful for note durations given either the tempo or the duration of a whole note.

SELECT...CASE was introduced as a way of evaluating a variable on a case by case basis. **SELECT...CASE** is particularly useful for examining characters or numbers when there are many choices as to what the variable could be and many different sets of actions that

need to be taken based on the variable's value. A program that converts strings of characters that describe musical tones for cell phones (called RTTTL files) was used to introduce a larger program that makes use of all the programming techniques introduced in this text. **SELECT...CASE** played a prominent role in this program because it is used to examine characters selected in an RTTTL file on a case-by-case basis.

Questions

1. What causes a tone to sound high-pitched? What causes a tone to sound low-pitched?
2. What does **FREQOUT 15, 1000, 3000** do? What effect does each of the numbers have?
3. How can you modify the **FREQOUT** command from Question 2 so that it sends two frequencies at once?
4. If you strike a piano's B6 key, what frequency does it send?
5. How do you modify a **DATA** directive or **READ** command if you want to store and retrieve word values?
6. Can you have more than one **DATA** directive? If so, how would you tell a **READ** command to get data from one or the other **DATA** directive?
7. What's an octave? If you know the frequency of a note in one octave, what do you have to do to that frequency to play it in the next higher octave?
8. What does **SELECT...CASE** do?

Exercises

1. Modify the "Alarm..." tone from ActionTones.bs2 so that the frequency of the tone it plays increases by 500 each time the tone repeats.
2. Explain how to modify MusicWithMoreFeatures.bs2 so that it displays an alert message in the Debug Terminal each time a dotted note is played.

Project

1. Build pushbutton controlled tone generator. If one pushbutton is pressed, the speaker should make a 2 kHz beep for 1/5 of a second. If the other pushbutton is pressed the speaker should make a 3 kHz beep for 1/10 of a second.

Solutions

- Q1. Our ears detect changes in air pressure as tones. A high pitched tone is from faster changes in air pressure, a low pitched tone from slower changes in air pressure.
- Q2. **FREQOUT 15, 1000, 3000** sends a 3000 Hz signal out pin 15 for one second (1000 ms).
The effect of each number:
15 – pin number 15.
1000 – duration of tone equals 1000 ms or one second.
3000 – the frequency of the tone, in Hertz, so this sends a 3000 Hz tone.
- Q3. Use the optional **Freq2** argument. To mix 3000 Hz and say, 2000 Hz, we simply add the second frequency to the command, after a comma:
FREQOUT 15, 1000, 3000, 2000
- Q4. 1975.5 Hz, see Figure 8-3 on page 228.
- Q5. Use the optional **Word** modifier before each data item.
- Q6. Yes. Each **DATA** directive has a different symbol parameter. To specify which **DATA** directive to get the data from, include the symbol parameter after the **READ** keyword. For example: **READ Notes, noteLetter**. In this example, **Notes** is the symbol parameter.
- Q7. An octave is a group of 8 whole and 4 chromatic notes. To get a given note in the next higher octave, multiply the frequency by two.
- Q8. **SELECT...CASE** selects a variable or expression, evaluates it on a case by case basis, and executes different blocks of code depending on which case the variable's value fits into.
- E1. This problem can be solved either by manually increasing each tone by 500, or by utilizing a **FOR...NEXT** loop with a **STEP** value of 500.

Utilizing FOR...NEXT loop:

```
DEBUG "Increasing alarm...", CR
  PAUSE 100
  FOR frequency = 1500 TO 3000 STEP 500
    FREQOUT 9, 500, frequency
    PAUSE 500
  NEXT
```

Manually increasing tone:

```
DEBUG "Increasing Alarm...", CR
  PAUSE 100
  FREQOUT 9, 500, 1500
  PAUSE 500
  FREQOUT 9, 500, 2000
  PAUSE 500
  FREQOUT 9, 500, 2500
  PAUSE 500
  FREQOUT 9, 500, 3000
  PAUSE 500
```

- E2. Modify the lines that check for the dotted note:
READ Dots + index, noteDot

```

    IF noteDot = 1 THEN noteDuration = noteDuration * 3 / 2
Add a DEBUG command to the IF...THEN. Don't forget the ENDIF.
    READ Dots + index, noteDot
    IF noteDot = 1 THEN
        noteDuration = noteDuration * 3 / 2
        DEBUG "Dotted Note!", CR
    ENDIF

```

- P1. Use the piezospeaker circuit from Figure 8-2, p. 220; pushbutton circuits from Figure 4-20, p. 129.

```

' What's a Microcontroller - Ch8Prj01_PushButtonToneGenerator.bs2
' P4 Pressed:  2 kHz beep for 1/5 second. 2 kHz = 2000 Hz.
'             1/5 s = 1000 / 5 ms = 200 ms
' P3 Pressed:  3 kHz beep for 1/10 second. 3 kHz = 3000 Hz.
'             1/10 s = 1000 / 10 ms = 100 ms
'{$STAMP BS2}
'{$PBASIC 2.5}
DEBUG "Program Running!"

DO
    IF (IN4 = 1) THEN
        FREQOUT 9, 200, 2000           ' 2000 Hz for 200 ms
    ELSEIF (IN3 = 1) THEN
        FREQOUT 9, 100, 3000          ' 3000 Hz for 100 ms
    ENDIF
LOOP

```

Further Investigation

“Applied Sensors”, Student Guide, Version 2.0, Parallax Inc., 2003

More sound effects, clicks, crickets are introduced using the piezoelectric speaker. The pentatonic and equal temperament scale are the basis for discussion of math fractions. The speaker is also used as feedback for a variety of sensor measurements.

“Basic Analog and Digital”, Student Guide, Version 2.0, Parallax Inc., 2003

The speaker is used to make a frequency generated by a 555 timer audible. The BASIC Stamp measures the frequency of the tone with a **COUNT** command.

“Understanding Signals”, Student Guide, Version 1.0, Parallax Inc., 2003

You can use this book to view the output of the **FREQOUT** command, both as digital pulses, and as sine waves.

Chapter #9: Electronic Building Blocks

THOSE LITTLE BLACK CHIPS

You need look no further than your BASIC Stamp (see Figure 9-1) to find examples of “those little black chips”. Each of these chips has a special function. The upper-right chip is the voltage regulator. This chip takes the battery voltage and converts it to almost exactly 5.0 V, which is what the rest of the components on the BASIC Stamp need to run properly. The upper-left chip is the BASIC Stamp module’s EEPROM. PBASIC programs are condensed to numbers called tokens that are downloaded to the BASIC Stamp. These tokens are stored in the EEPROM, and you can view them by clicking *Run* and then *Memory Map* in the BASIC Stamp Editor. The largest chip is called the Interpreter chip. It fetches the tokens from the EEPROM and then interprets the PBASIC command that the token represents. Then, it executes the command, fetches the next token, and so on. This process is called “fetch and execute”.

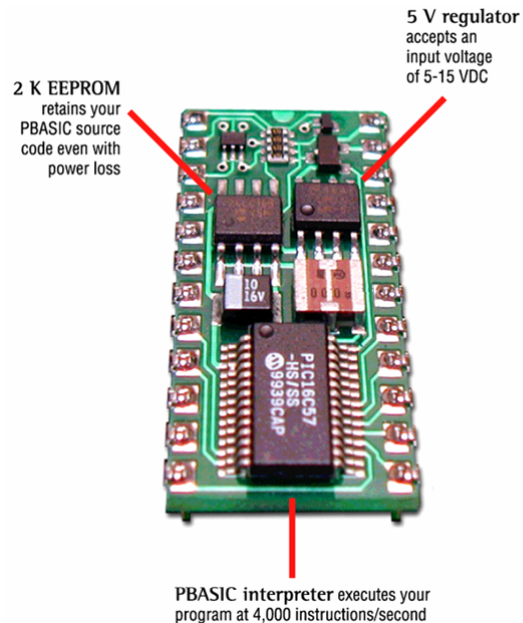


Figure 9-1
Integrated
Circuits on the
BASIC Stamp

People use the term “integrated circuit” (IC) to talk about little black chips. The integrated circuit is actually a tiny silicon chip that’s contained inside the black plastic or ceramic case. Depending on the chip, it may have anywhere between hundreds and millions of transistors. A transistor is the basic building block for integrated circuits, and you will have the opportunity to experiment with a transistor in this chapter. Other familiar components that are designed into silicon chips include diodes, resistors and capacitors.

Take a moment to think about the activities you’ve tried in this book so far. The list includes switching LEDs on and off, reading pushbuttons, controlling servos, reading potentiometers, measuring light, controlling displays, and making sounds. Even though that’s just the beginning, it’s still pretty impressive, especially considering that you can combine these activities to make more complex gizmos and gadgets. The core of the system that made all those activities possible is comprised of just the three integrated circuits shown in Figure 9-1 and a few other parts. It just goes to show how powerful integrated circuits can be when they are designed to work together.

EXPAND YOUR PROJECTS WITH PERIPHERAL INTEGRATED CIRCUITS

There are thousands of integrated circuits designed to be used with microcontrollers. Sometimes different integrated circuit manufacturers make chips that perform the same function. Sometimes each chip’s features differ slightly, and other times, the chips are identical, but one might cost a little less than the other. Each one of the thousands of different integrated circuits can be used as a building block for a variety of designs. Companies publish information on how each of their integrated circuits work in documents called datasheets that are published on the World Wide Web. These manufacturers also publish application notes, which show how to use their integrated circuit in unique or useful ways that make it easier to design products. The integrated circuit manufacturers give away this information in hopes that engineers will use it to build their chip onto the latest must-have toy or appliance. If thousands of toys are sold, it means the company sells thousands of their integrated circuits.

In this chapter, you will experiment with a transistor, and a special-purpose integrated circuit called a digital potentiometer. As mentioned earlier, the transistor is the basic building block for integrated circuits. It’s also a basic building block for lots of other circuits as well. The digital potentiometer also has a variety of uses. Keep in mind that for each activity you have done, there are probably hundreds of different ways that you could use each of these integrated circuits.

ACTIVITY #1: CONTROL CURRENT FLOW WITH A TRANSISTOR

In this activity, you will use a transistor as a way to control the current passing through an LED. You can use the LED to monitor the current since it glows more brightly when more current passes through it and less brightly when less current passes through it.

Introducing the Transistor

Figure 9-2 shows the schematic symbol and part drawing of the 2N3904 transistor. There are many different types of transistors. This one is called NPN, which refers to the type of materials used to manufacture the transistor and the way those materials are layered on the silicon. The best way to get started thinking about a transistor is to imagine it as a valve that is used to control current. Different transistors control how much current passes through by different means. This transistor controls how much current passes into C (collector) and back out of E (emitter). It uses the amount of current allowed into the B (base) terminal to control the current passing from C through E. With a very small amount of current allowed into B, a current flow of about 416 times that amount flows through the transistor into C and out of E.

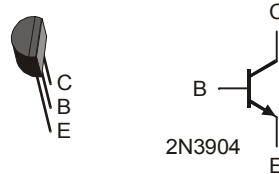


Figure 9-2
2N3904 Transistor



The 2N3904 Part Datasheet: As mentioned earlier, semiconductor manufacturers publish documents called datasheets for the parts they make. These datasheets contain information engineers use to design the part into a product. To see an example of a part datasheet for the 2N3904: Go to www.fairchildsemi.com. Enter 2N3904 into the Search field on Fairchild Semiconductor's home page, and click Go. One of the search results should be a link to the 2N3904 Product Folder. Follow the link to the product folder, then click Download this Datasheet link. Most web browsers display the datasheet by opening it with Adobe Acrobat Reader.

Transistor Example Parts

- (1) Transistor – 2N3904
- (2) Resistors – 100 k Ω (brown-black-yellow)
- (1) LED – any color

- (1) Potentiometer – 10 k Ω
- (3) Jumper wires

Building and Testing the Transistor Circuit

Figure 9-3 shows a circuit that you can use to manually control how much current the transistor allows through the LED. By twisting the knob on the potentiometer, the circuit will deliver different amounts of current to the transistor's base. This will cause a change in the amount of current the transistor allows to pass from its collector to its emitter. The LED will give you a clear indication of the change by glowing more or less brightly.

- ✓ Build the circuit shown in Figure 9-3.
- ✓ Turn the knob on the potentiometer and verify that the LED changes brightness in response to a change in the position of the potentiometer's wiper terminal.

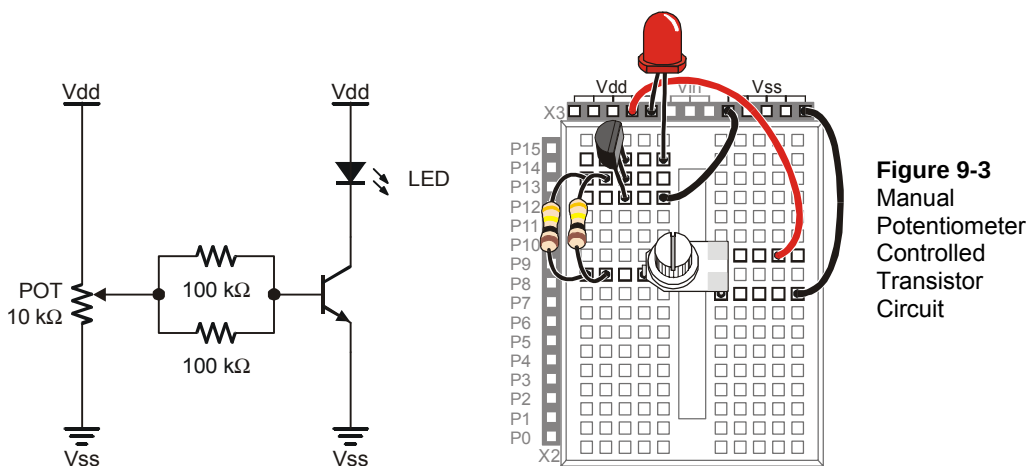


Figure 9-3
Manual
Potentiometer
Controlled
Transistor
Circuit

Your Turn – Switching the Transistor On/Off

If all you want to do is switch a transistor on and off, you can use the circuit shown in Figure 9-4. When the BASIC Stamp sends a high signal to this circuit, it will make it so that the transistor conducts as much current as if you adjusted the potentiometer for maximum brightness. When the BASIC Stamp sends a low signal to this circuit, it will cause the transistor to stop conducting current, and the LED should emit no light.



What's the difference between this and connecting an LED circuit to an I/O pin? BASIC Stamp I/O pins have limitations on how much current they can deliver. Transistors have limitations too, but they are much higher. In the *Industrial Control Student Guide*, a transistor is used to drive a small DC fan. It is also used to supply large amounts of current to a small resistor that is used as a heating element. Either of these two applications would draw so much current that they would quickly damage the BASIC Stamp, but the transistor takes it in stride.

- ✓ Build the circuit shown in Figure 9-4.
- ✓ Write a program that sends high and low signals to P8 twice every second. HINT: LedOnOff.bs2 from Chapter #2 needs only to be modified to send high/low signals to P8 instead of P3. Remember to save it under a new name before making the modifications.
- ✓ Run the program and verify that it gives you on/off control of the LED.

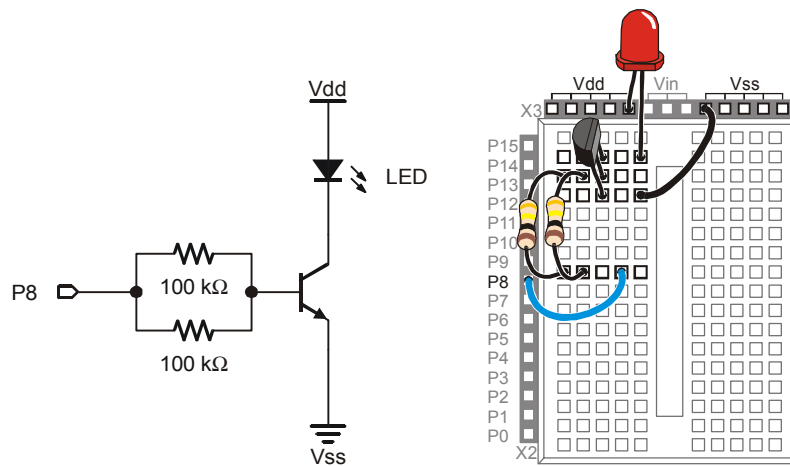


Figure 9-4
Circuit that
Gives BASIC
Stamp
Transistor
On/Off Control

9

ACTIVITY #2: INTRODUCING THE DIGITAL POTENTIOMETER

In this activity, you will replace the manually adjusted potentiometer with an integrated circuit potentiometer that is digitally adjusted. You will then program the BASIC Stamp to adjust the digital potentiometer, which will in turn adjust the LED's brightness in the same way the manual potentiometer did in the previous activity.

Introducing the Digital Potentiometer

Figure 9-5 shows a pin map of the digital potentiometer you will use in this activity. This chip has 8 pins, four on each side that are spaced to make it easy to plug into a breadboard (1/10 inch apart). The manufacturer places a reference notch on the plastic case so that you can tell the difference between pin 1 and pin 5. The reference notch is a small half-circle in the chip's case. You can use this notch as a reference for the pin numbers on the chip. The pin numbers on the chip count upwards, counterclockwise from the reference notch.



Part Substitutions: It is sometimes necessary for Parallax to make a part substitution. The part will function the same, but the label on it may be different. If you find that the digital potentiometer included in your What's a Microcontroller Parts Kit is not labeled AD5220, rest assured that it will still work the same way and perform correctly in this activity.

Reference
Notch

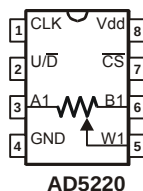


Figure 9-5
AD5220 Pin Map

Use the reference notch to make sure you have the AD5220 right-side-up when building it into your circuit on the breadboard.

Here is a summary of each of the AD5220's pins and functions:

1. CLK – The pin that receives clock pulses (low-high-low signals) to move the wiper terminal.
2. U/D – The pin that receives a high signal to make the wiper (W1) terminal move towards A1, and a low signal to make it move towards B1. This pin just sets the direction, the wiper terminal doesn't actually move until a pulse (a low – high – low signal) is sent to the CLK pin.
3. A1 – The potentiometer's A terminal.
4. GND – The ground connection. The ground on the Board of Education and BASIC Stamp HomeWork Board is the Vss terminal.
5. W1 – The potentiometer's wiper (W) terminal.

6. B1 – The potentiometer's B terminal.
7. CS – The chip select pin. Apply a high signal to this pin, and the chip ignores all control signals sent to CLK and U/D. Apply a low signal to this pin, and it acts on any control signals it receives.
8. Vdd – Connect to +5 V, which is Vdd on the Board of Education and BASIC Stamp HomeWork Board.



The AD5220 Part Datasheet: To see the part datasheet for the AD5220: Go to www.analog.com. Enter AD5220 into the Search field on Analog Devices' home page, and click the Search button. Click the Data Sheets link. Click the link that reads "AD5220: Increment/Decrement Digital Potentiometer Datasheet".

Digital Pot Controlled Transistor Parts

- (1) Transistor – 2N3904
- (2) Resistors – 100 k Ω (brown-black-yellow)
- (1) LED – any color
- (1) Digital potentiometer – AD5220

Building the Digital Potentiometer Circuit

Figure 9-6 shows a circuit schematic with the digital potentiometer used in place of a manual potentiometer, and Figure 9-7 shows a wiring diagram for the circuit. The BASIC Stamp can control the digital potentiometer by issuing control signals to P5 and P6.

✓ Build the circuit shown in Figure 9-6 and Figure 9-7.

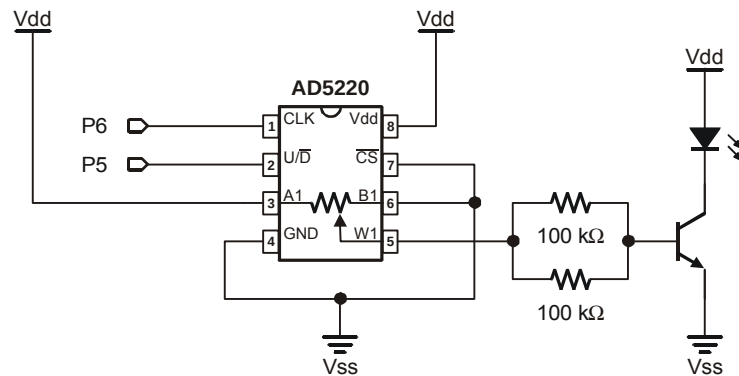


Figure 9-6
Digital Potentiometer
Controlled Transistor
Circuit Schematic

Imagine next that you turn the potentiometer all the way clockwise. That's the same as sending a high signal to the AD5220's U/D pin and sending 65 pulses to the CLK pin. Now the LED should be shining brightly.

```
HIGH 5
FOR counter = 1 TO 65
  PULSOUT 6, 1
  PAUSE 1
NEXT
```

Finally, imagine that you turn your manual potentiometer all the way counterclockwise. The LED should emit no light. That's the same as sending a low signal to the U/D pin, and applying 128 pulses to the CLK pin

```
LOW 5
FOR counter = 0 TO 127
  PULSOUT 6, 1
  PAUSE 1
NEXT
```

Example Program: DigitalPotUpDown.bs2

This example program adjusts the potentiometer up and down, from one end of its range to the other, causing the LED to get gradually brighter, then gradually dimmer.

9

✓ Enter and run DigitalPotUpDown.bs2.

```
' What's a Microcontroller - DigitalPotUpDown.bs2
' Sweep digital pot through values.

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

DEBUG "Program Running!"

counter          VAR      Byte

DO

  LOW 5

  FOR counter = 0 TO 127
    PULSOUT 6, 1
    PAUSE 10
  NEXT

  HIGH 5
```

```

FOR counter = 0 TO 127
  PULSOUT 6, 1
  PAUSE 10
NEXT
LOOP

```

Your Turn – Changing the Rate and Condensing the Code

You can increase or decrease the rate at which the LED gets brighter and dimmer by changing the **PAUSE** command's *Duration* argument.

- ✓ Modify and re-run the program using **PAUSE 20** and note the difference in the rate that the LED gets brighter and dimmer.
- ✓ Repeat for **PAUSE 5**.

You can also use a command called **TOGGLE** to make this program simpler. **TOGGLE** changes the state of a BASIC Stamp I/O pin. If the I/O pin was sending a high signal, **TOGGLE** makes it send a low signal. If the I/O pin was sending a low signal, **TOGGLE** makes it send a high signal.

- ✓ Save DigitalPotUpDown.bs2 as DigitalPotUpDownWithToggle.bs2.
- ✓ Modify the program so that it looks like the one below.
- ✓ Run the program and verify that it functions the same as the DigitalPotUpDown.bs2.
- ✓ Compare the number of lines of code it took to do the same job.



Running out of program memory is a problem many people encounter when their BASIC Stamp projects get large and complicated. Using **TOGGLE** instead of two **FOR...NEXT** loops is just one example of many techniques that can be used to do the same job with half the code.

```

' What's a Microcontroller - DigitalPotUpDownWithToggle.bs2
' Sweep digital pot through values.

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

counter      VAR      Byte

```

```

LOW 5
DO
  FOR counter = 0 TO 127
    PULSOUT 6,5
    PAUSE 10
  NEXT
  TOGGLE 5
LOOP

```

Looking Inside the Digital Potentiometer

Figure 9-8 shows a diagram of the potentiometer inside the AD5220. The AD5220 has 128 resistive elements, each of which is $78.125\ \Omega$ (nominal value). All 128 of these add up to $10,000\ \Omega$ or $10\ \text{k}\Omega$.



A nominal value means a named value. Parts like resistors and capacitors typically have a nominal value and a tolerance. Each of the AD5220's resistive elements has a nominal value of $78.125\ \Omega$, with a tolerance of 30% ($23.438\ \Omega$) above or below the nominal value.

Between each of these resistive elements is a switch, called a tap. Each switch is actually a group of transistors that are switched on or off to let current pass or not pass. Only one of these switches can be closed at one time. If one of the upper switches is closed (like pos. 125, 126, or 127), it's like having the manual potentiometer knob turned most or all the way clockwise. If pos. 0 or 1 is closed, it's like having a manual potentiometer turned most or all the way counterclockwise.

9

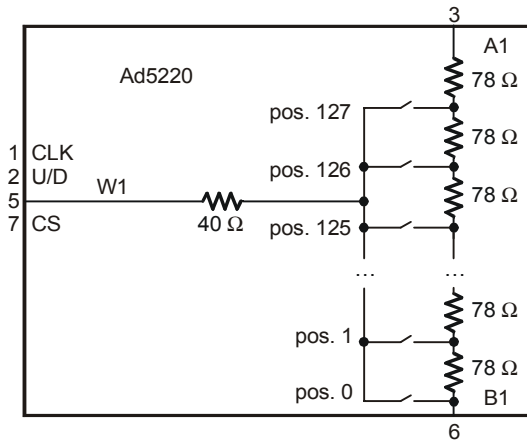


Figure 9-8
Inside the AD5220

Imagine that Pos. 126 is closed. If you want to set the tap to 125, (open pos. 126 and close pos. 125), set U/D low, then apply a pulse to CLK. If you want to set the tap to Pos 127, set U/D high, and apply 2 pulses. If you want to bring the tap down to 1, set U/D low, and apply 126 pulses.

This next example program uses the Debug Terminal to ask you which tap setting you want. Then it decides whether to set the U/D pin high or low, and applies the correct number of pulses to move the tap from its old setting to the new setting.

With the exception of EEPROM Data, the next example program also has all the sections you could normally expect to find in an application program:

- Title – comments that include the filename of a program, its description, and the Stamp and PBASIC directives.
- EEPROM Data – The **DATA** declarations used by the program.
- I/O Definitions – constant declarations that define I/O pin numbers.
- Constants – constant declarations that define other values used in the program.
- Variables – variable declarations.
- Initialization – a routine that gets the program started on the right foot. In this next program, the potentiometer's tap needs to be brought down to zero.
- Main – the routine that handles the primary jobs the program has to do.
- Subroutines – the segments of code that do specific jobs, either for each other, or in this case, for the main routine.

Example Program: TerminalControlledDigitalPot.bs2

You can use this example program and the Debug Terminal to set the digital pot's tap. By changing the tap setting on the digital pot, you change the brightness of the LED connected to the transistor that the digital pot controls. Figure 9-9 shows an example of entering the value 120 into the Debug Terminal's Transmit Windowpane while the program is running. Since the old tap setting was 65, the LED becomes nearly twice as bright when it is adjusted to 120.

Windowpanes

Transmit →

Receive →

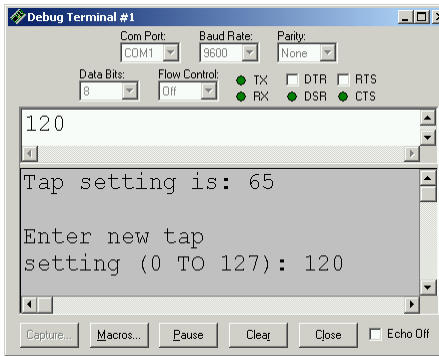


Figure 9-9
Sending Messages
to the BASIC Stamp

*Click the Transmit
(upper) Windowpane
and enter the
numbers for the new
tap setting.*

- ✓ Enter and run TerminalControlledDigitalPot.bs2.
- ✓ Enter values between 0 and 127 into the Debug Terminal. Make sure to press the enter key after you type in the digits.

```
' -----[ Title ]-----
' What's a Microcontroller - TerminalControlledDigitalPot.bs2
' Update digital pot's tap based on Debug Terminal user input.

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

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

' -----[ I/O Definitions ]-----

UdPin      CON      5           ' Set values of I/O pins
ClkPin     CON      6           ' connected to CLK and U/D.

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

DelayPulses CON      10         ' Delay to observe LED fade.
DelayReader CON     2000

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

counter    VAR      Byte        ' Counter for FOR...NEXT.
oldTapSetting VAR    Byte        ' Previous tap setting.
newTapSetting VAR    Byte        ' New tap setting.

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

oldTapSetting = 0           ' Initialize new and old
newTapSetting = 0           ' tap settings to zero.
```



```

LOW UdPin                                ' Set U/D pin for Down.
FOR counter = 0 TO 127                    ' Set tap to lowest position.
    PULSOUT 6,5
    PAUSE 1
NEXT

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

DO

    GOSUB Get_New_Tap_Setting              ' User display and get input.
    GOSUB Set_Ud_Pin                       ' Set U/D pin for up/down.
    GOSUB Pulse_Clk_pin                    ' Deliver pulses.

LOOP

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

Get_New_Tap_Setting:                      ' Display instructions and
                                          ' get user input for new
                                          ' tap setting value.
    DEBUG CLS, "Tap setting is: ",
    DEC newTapSetting, CR, CR
    DEBUG "Enter new tap", CR, "setting (0 TO 127): "
    DEBUGIN DEC newTapSetting

    RETURN

Set_Ud_Pin:                              ' Examine new and old tap values
                                          ' to decide value of U/D pin.
    IF newTapSetting > oldTapSetting THEN ' Notify user if values are
        HIGH UdPin                       ' equal.
        oldTapSetting = oldTapSetting + 1 ' Increment for Pulse_Clk_pin.
    ELSEIF newTapSetting < oldTapSetting THEN
        LOW UdPin
        oldTapSetting = oldTapSetting - 1 ' Decrement for Pulse_Clk_pin.
    ELSE
        DEBUG CR, "New and old settings", CR,
        "are the same, try ", CR,
        "again...", CR
        PAUSE DelayReader                 ' Give reader time to view
    ENDIF                                 ' Message.

    RETURN

Pulse_Clk_pin:

    ' Deliver pulses from old to new values. Keep in mind that Set_Ud_Pin
    ' adjusted the value of oldTapSetting toward newTapSetting by one.
    ' This keeps the FOR...NEXT loop from executing one too many times.

```

```
FOR counter = oldTapSetting TO newTapSetting
  PULSOUT ClkPin, 1
  PAUSE DelayPulses
NEXT

oldTapSetting = newTapSetting      ' Keep track of new and old
                                   ' tapSetting values.
RETURN
```

SUMMARY

This chapter introduced integrated circuits and how they can be used with the BASIC Stamp. A transistor was used as a current valve, and a digital potentiometer was used to control the amount of current passing through the transistor. Examining the digital potentiometer introduced the reference notch and pin map as important elements of electronic chips. The function of each of the digital potentiometer pins was discussed, as well as the device's internal structure. The PBASIC command **TOGGLE** was introduced as a means to save program memory.

Questions

1. What are the names of the terminals on the transistor you used in this chapter?
2. Which terminal controls the current passing through the transistor?
3. What can you do to increase or decrease the current passing through the transistor?

Exercise

1. Write a program that adjusts the tap in the digital pot to position 0 regardless of its current setting.

Project

1. Add a photoresistor to your project and cause the brightness of the LED to adjust with the brightness seen by the photoresistor.

Solutions

- Q1. Emitter, base, and collector.
 Q2. The base controls the current passing through the transistor.
 Q3. Increase or decrease the current allowed into the transistor's base.
 E1. To solve this exercise, look at TerminalControlledDigitalPot.bs2. The first thing it does, in the Initialization section, is to set the tap to the lowest position. This exact code is used in the solution below.

```
' What's a Microcontroller - Ch9Ex01_SetTapToZero.bs2
' Turn tap on digital pot all the way down to zero
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Program Running!"
```

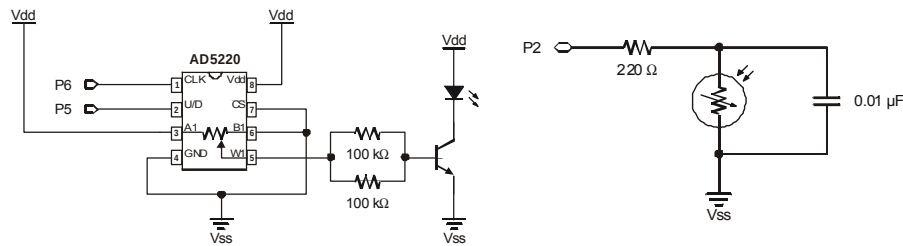
```

UdPin      CON      5           ' Set values of I/O pins
ClkPin     CON      6           ' connected to CLK and U/D.
counter    VAR      Byte       ' Counter for FOR...NEXT.

LOW UdPin                                     ' Set U/D pin for Down.
FOR counter = 0 TO 128                       ' Set tap to lowest position.
  PULSOUT ClkPin,5
  PAUSE 1
NEXT

```

P1. Use the digital potentiometer circuit from Figure 9-6, p. 269 and the photoresistor circuit from Figure 7-2, p. 191.



9

This solution is based on `TerminalControlledDigitalPot.bs2`, except instead of being controlled by typing in the Debug Terminal, it is controlled by reading the photoresistor. A subroutine, `Read_Photoresistor`, reads the photoresistor using `RCTIME` as shown in Chapter 5. The reading from the photoresistor becomes the new tap setting, and the original program does the work to set the tap. The parts of the original program that ask for input from the Debug Terminal were deleted, including the `Get_New_Tap_Setting` subroutine.

```

' What's a Microcontroller - Ch9Prj01_PhotoControlledDigitalPot.bs2
' Update digital pot's tap based on photoresistor reading
' {$STAMP BS2}
' {$PBASIC 2.5}
DEBUG "Program Running!"

' -----[ Declarations and Initialization]-----
UdPin      CON      5           ' Set values of I/O pins
ClkPin     CON      6           ' connected to CLK and U/D.
PhotoPin   CON      2           ' Photoresistor on pin P2
DelayPulses CON      10         ' Delay to observe LED fade.
DelayReader CON      2000

```

```

counter      VAR      Byte      ' Counter for FOR...NEXT.
oldTapSetting VAR      Byte      ' Previous tap setting.
newTapSetting VAR      Byte      ' New tap setting.
lightReading  VAR      Word      ' reading from photoresistor

oldTapSetting = 0      ' Initialize new and old
newTapSetting = 0      ' tap settings to zero.

LOW UdPin      ' Set U/D pin for Down.
FOR counter = 0 TO 128 ' Set tap to lowest position.
  PULSOUT 6,5
  PAUSE 1
NEXT

' -----[ Main Routine ]-----
DO
  GOSUB Read_Photoresistor
  newTapSetting = lightReading
  GOSUB Set_Ud_Pin      ' Set U/D pin for up/down.
  GOSUB Pulse_Clk_pin   ' Deliver pulses.
LOOP

' -----[ Subroutines ]-----
Set_Ud_Pin:
  IF newTapSetting > oldTapSetting THEN ' Examine old and new
    ' tap values to decide
  ELSEIF newTapSetting < oldTapSetting THEN ' value of UdPin. Notify
    ' user if values are
    ' equal.
  LOW UdPin
  ENDIF
  RETURN

Pulse_Clk_pin:
  ' Deliver pulses
  FOR counter = oldTapSetting TO newTapSetting ' from old to new
    PULSOUT ClkPin, 1 ' values.
    PAUSE DelayPulses
  NEXT
  oldTapSetting = newTapSetting ' Keep track of new and old
  RETURN ' tapSetting values.

Read_Photoresistor:
  HIGH PhotoPin
  PAUSE 100
  RCTIME PhotoPin, 1, lightReading
  RETURN

```

Further Investigation

“Industrial Control”, Student Guide, Version 2.0, Parallax Inc., 2002

Industrial Control uses the transistor as an on/off switch for a resistor heating element. It also uses a phototransistor to detect the passage of stripes on a spinning wheel.

Chapter #10: Running the Whole Show

SUBSYSTEM INTEGRATION

Most of the activities in this text introduce how to program the BASIC Stamp to interact with one or two circuits at a time. Many microcontrollers have to handle tens or even hundreds of circuits. This chapter will demonstrate a few of the techniques used to manage a variety of circuits with a single microcontroller. Programming the microcontroller to orchestrate the activities of different circuits performing unique functions is called subsystem integration.

The example in this chapter uses a sensor array consisting of two pushbutton circuits, a potentiometer circuit and a photoresistor circuit. The activities in this chapter will guide you through building and testing each subsystem individually. After each subsystem is built and tested, you will then write a program that combines each subsystem into a working unit. Figure 10-1 shows the circuit schematic for the system you will build, and the master parts list is below.

Always remember: whenever possible, test each subsystem individually before trying to make them work together. If you follow this rule, your projects will go much more smoothly, and your chances of success will be greatly improved. The activities in this chapter will guide you through the process.

10

Sensor Array Parts List

- (4) Resistors – 220 Ω (red-red-brown)
- (2) Resistors – 10 k Ω (brown-black-orange)
- (2) Pushbuttons – normally open
- (2) Capacitors – 0.1 μ F
- (1) Potentiometer – 10 k Ω
- (1) Photoresistor
- (5) Jumper wires

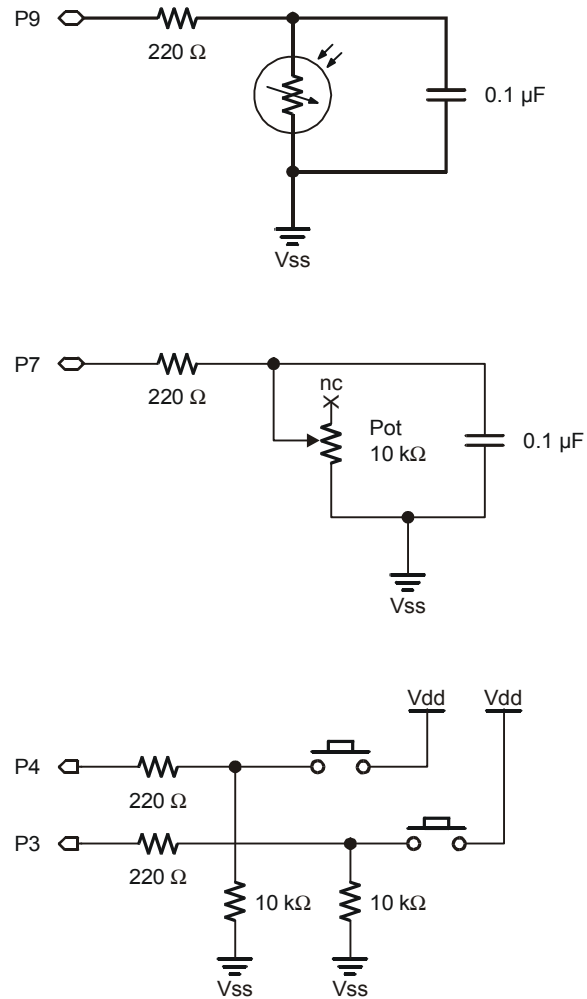


Figure 10-1
Sensor Array
System Schematic

ACTIVITY #1: BUILDING AND TESTING EACH PUSHBUTTON CIRCUIT

This activity begins with building and testing a single pushbutton circuit. Once you have confirmed that the first pushbutton circuit is operating properly, you can then move on to building and testing the second pushbutton circuit.

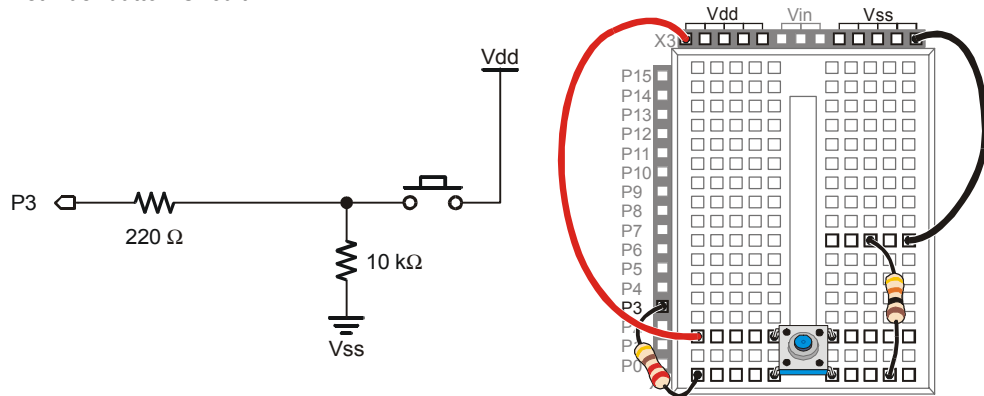
Pushbutton Circuit Parts

- (2) Pushbuttons – normally open
- (2) Resistors – 10 k Ω (brown-black-orange)
- (2) Resistors – 220 Ω (red-red-brown)
- (2) Jumper wires

Building the First Pushbutton Circuit

Build the pushbutton circuit shown in Figure 10-2.

Figure 10-2
First Pushbutton Circuit



10

Testing the First Pushbutton Circuit

Writing a program to test this circuit should be very simple by now, especially since you've already done it before.

Example Program: ReadPushbuttonState.bs2

This is a repeat of the program for testing pushbuttons from Chapter #3:

- ✓ Enter and run the ReadPushbuttonState.bs2.
- ✓ Verify that the BASIC Stamp is able to read the pushbutton.
- ✓ Correct any problems you catch before moving on to the next circuit.


```
' What's a Microcontroller - ReadPushbuttonState.bs2
' Check and send pushbutton state to Debug Terminal every 1/4 second.

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

DO

    DEBUG ? IN3
    PAUSE 250

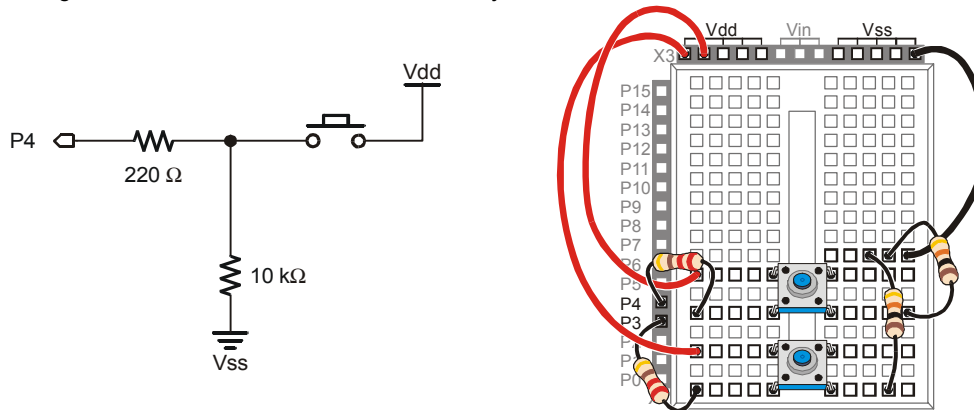
LOOP
```

Your Turn – Building and Testing the Second Pushbutton Circuit

Once the first pushbutton circuit has been built and tested, the process can be repeated for the second pushbutton.

- ✓ Add the pushbutton circuit shown in Figure 10-3.
- ✓ Test the second pushbutton circuit by modifying ReadPushbuttonState.bs2 so that it reads the circuit connected to P4. Run the program and verify that the second pushbutton works.
- ✓ Correct any problems you catch before moving on to the next activity.

Figure 10-3
Adding a Second Pushbutton Circuit to the Project



ACTIVITY #2: BUILDING AND TESTING EACH RC-TIME CIRCUIT

Now that the two pushbutton circuits are built and tested, you can move on to the RC-time circuits.

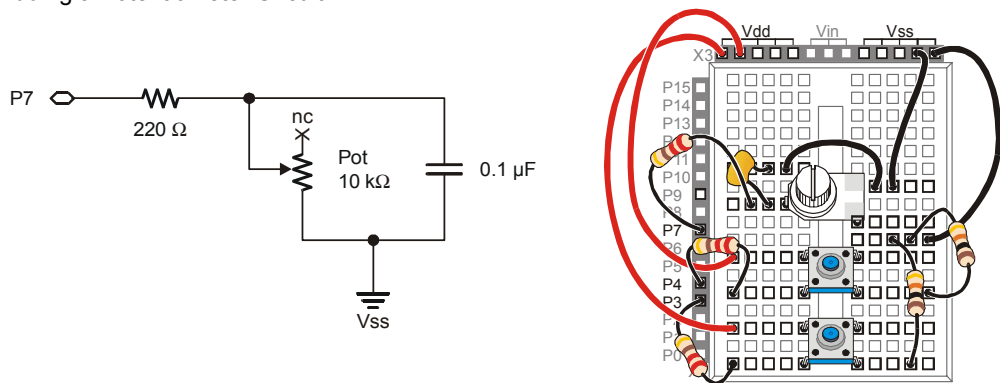
Extra Parts for Potentiometer and Photoresistor Circuits

- (2) Resistors – $220\ \Omega$ (red-red-brown)
- (2) Capacitors – $0.1\ \mu\text{F}$
- (4) Jumper wires
- (1) Photoresistor
- (1) Potentiometer – $10\ \text{k}\Omega$

Building the Potentiometer Circuit

- ✓ Add the potentiometer circuit shown in Figure 10-4 to the project.

Figure 10-4
Adding a Potentiometer Circuit



10

Testing the Potentiometer Circuit

You can test this circuit using ReadPotWithRcTime.bs2. This program was first used in Chapter #5 to read the potentiometer circuit.

Example Program: ReadPotWithRcTime.bs2

- ✓ Enter and run the ReadPotWithRcTime.bs2.

- ✓ Verify that the BASIC Stamp gets reliable measurements from the potentiometer.
- ✓ Correct any problems you catch before moving on to the next circuit.

```
' What's a Microcontroller - ReadPotWithRcTime.bs2
' Read potentiometer in RC-time circuit using RCTIME command.

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

time          VAR      Word

DO

    HIGH 7
    PAUSE 100
    RCTIME 7, 1, time
    DEBUG HOME, " time = ", DEC5 time

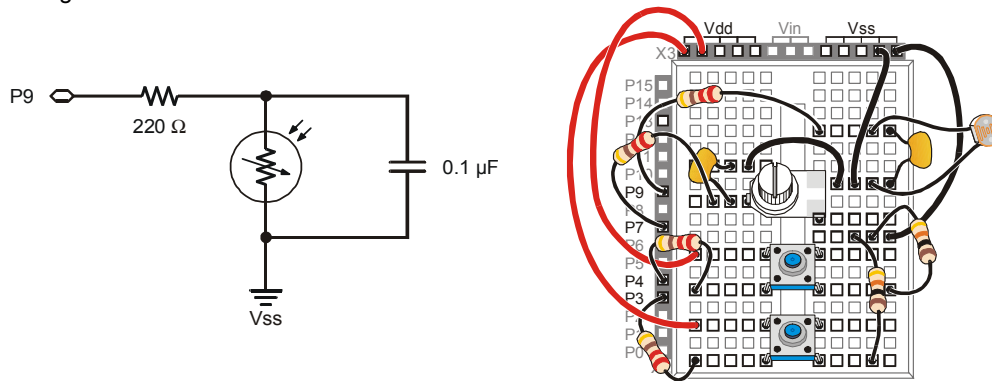
LOOP
```

Your Turn – Building and Testing the Photoresistor Circuit

Once the first RC-time circuit has been built and tested, the process can be repeated for the second RC-time circuit.

- ✓ Add the photoresistor circuit shown in Figure 10-5 to your project on the breadboard.
- ✓ Modify ReadPotWithRcTime.bs2 so the photoresistor is connected to P9.
- ✓ Correct any problems you catch before moving on to the next activity.

Figure 10-5
Adding the Photoresistor Circuit



ACTIVITY #3: SUBSYSTEM INTEGRATION EXAMPLE

Now that all the elements in the sensor array are built and tested, you can now write a program that makes use of all four sensor circuits. This example will demonstrate how to program the BASIC Stamp to display a terminal with a menu that a technician might use to monitor the sensors.

Programming a Menu System and Using the Input/Output PIN Directive

The `ON...GOSUB` command can be very useful for menus.

ON offset , GOSUB Target1, {Target2, Target3,...}

In the next example program, the `ON...GOSUB` command uses the value of a variable named `request` to direct it to one of four subroutines. If the value of `request` is 0, the program executes a `GOSUB Read_Pushbutton_1`. If the Value of `request` is 1, the program executes a `GOSUB Read_Pushbutton_2`, and so on.

```
ON request GOSUB Read_Pushbutton_1, Read_Pushbutton_2,
              Read_Pot, Read_Photoresistor
```

You can use the `PIN` directive to give a name to each I/O pin you are going to use in a PBASIC program. The BASIC Stamp Editor will then figure out whether you are using the I/O pin as an input or an output (or both). The `PIN` directive's syntax is:

PinName PIN PinNumber

The next example program demonstrates how you can declare and then use a *PinName*. For example, the BASIC Stamp Editor assumes that you want to display the input value of an I/O pin (**IN3**) if you are using the **Pb1Pin** *PinName* in a **DEBUG** command. This **DEBUG** command will display a 1 or 0 depending on whether the pushbutton connected to P3 is pressed or not pressed. Why? Because the BASIC Stamp Editor knows to substitute **IN3** for **Pb1Pin**.

```
Pb1Pin      PIN      3
DEBUG CLS, ? Pb1Pin
```

Here is another example where the BASIC Stamp Editor knows to substitute the value of 9 because the **PhotoPin** *PinName* is used in a **HIGH** command and an **RCTIME** command.

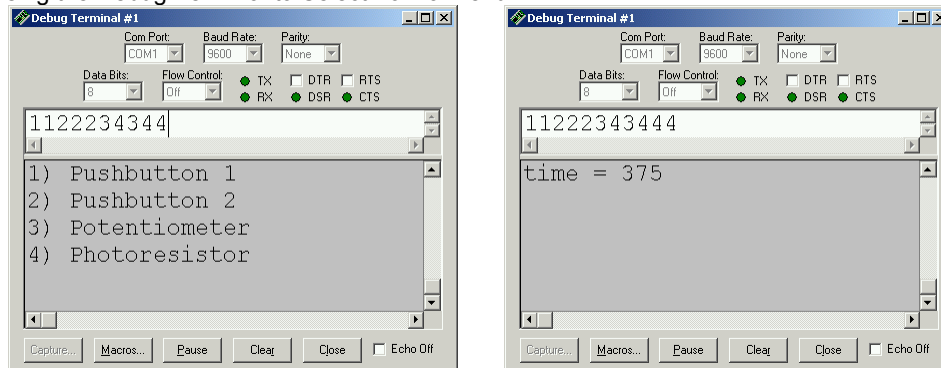
```
PhotoPin    PIN      9
HIGH PhotoPin
RCTIME PhotoPin, 1, time
```

Example Program: TerminalOperatedSensorArray.bs2

Figure 10-6 shows the Debug Terminal displayed by TerminalOperatedSensorArray.bs2. The main menu is shown on the left, and an example of the display when '4' is selected is shown on the right. Remember to use the upper Windowpane to send your menu selections to the BASIC Stamp.

- ✓ Enter and run TerminalOperatedSensorArray.bs2.
- ✓ Click the Debug Terminal's upper Windowpane, and type digits to make your menu selections.
- ✓ The measurement displayed is the one taken at the instant the menu selection is made, and it is displayed for 1.5 seconds. Keep this in mind when you press and hold the pushbuttons, adjust the potentiometer and cast shade onto the photoresistor.

Figure 10-6
Using the Debug Terminal to Select from a Menu



```
' -----[ Title ]-----
' What's a Microcontroller - TerminalOperatedSensorArray.bs2
' Use Debug Terminal to choose to read one of four sensors.

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

' -----[ I/O Definitions ]-----

Pb1Pin      PIN      3
Pb2Pin      PIN      4
PotPin      PIN      7
PhotoPin    PIN      9

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

DelayRc     CON      100
DelayReader CON      1500

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

request     VAR      Nib
time        VAR      Word

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

DO

  GOSUB Display_Menu
  GOSUB Get_Request
  ON request GOSUB Read_Pushbutton_1, Read_Pushbutton_2,
                Read_Pot, Read_Photoresistor
```

```

LOOP

' -----[ Subroutine - Display_Menu ]-----
Display_Menu:

    DEBUG CLS, "MENU: ", CR, CR,
           "1) Pushbutton 1", CR,
           "2) Pushbutton 2", CR,
           "3) Potentiometer", CR,
           "4) Photoresistor", CR, CR

    RETURN

' -----[ Subroutine - Get_Request ]-----
Get_Request:

    DEBUGIN DEC1 request
    request = request - 1

    RETURN

' -----[ Subroutine - Read_Pushbutton_1 ]-----
Read_Pushbutton_1:

    DEBUG CLS, ? Pb1Pin
    PAUSE DelayReader

    RETURN

' -----[ Subroutine - Read_Pushbutton_2 ]-----
Read_Pushbutton_2:

    DEBUG CLS, ? Pb2Pin
    PAUSE DelayReader

    RETURN

' -----[ Subroutine - Read_Pot ]-----
Read_Pot:

    HIGH PotPin
    PAUSE DelayRc
    RCTIME PotPin, 1, time
    DEBUG CLS, ? time, "    "
    PAUSE DelayReader

```

```

RETURN

' -----[ Subroutine - Read_Photorresistor ]-----

Read_Photorresistor:

HIGH PhotoPin
PAUSE DelayRc
RCTYPE PhotoPin, 1, time
DEBUG CLS, ? time, "      "
PAUSE DelayReader

RETURN

```

Your Turn – Modifying Subroutines

Terminals are not the only application for **ON...GOSUB**. For example, if you want to call the four different subroutines one after another, here is a way to do it:

```

DO

    ' GOSUB Display_Menu
    ' GOSUB Get_Request
    FOR request = 0 TO 3
        ON request GOSUB Read_Pushbutton_1, Read_Pushbutton_2,
                        Read_Pot, Read_Photorresistor
    NEXT

LOOP

```

10

Notice that the two subroutines that do the terminal work were commented and the **ON...GOSUB** command was placed in a **FOR...NEXT** loop.

- ✓ Save your example program under the new name: TerminalOperatedSensorArrayYourTurn.bs2.
- ✓ Modify the main routine by making the changes just discussed.
- ✓ Run the program and verify that it cycles through the measurements.

ACTIVITY #4: DEVELOPING AND ADDING A SOFTWARE SUBSYSTEM

Let's say that your sensor array is getting placed in a device that only certain employees are allowed to use, and they have to supply a password. This means you will have to expand your program so that it saves and verifies a password. It's better to write a password program that functions on its own first, and then add it to your larger program.

Programming a Password Checker

You can save a password in a PBASIC program by using a **DATA** directive. For example:

```
Password      DATA      "pass!"
```



The same as: This **DATA** directive is the same as **Password DATA "p", "a", "s", "s", "!"**.

You will need a few variables for storing other values in the program:

```
index          VAR      Nib
temp           VAR      Byte
```

If you are using a five character password, there is a special kind of variable declaration (called an array declaration) that you use to declare five variables with the same name.

```
userEntry      VAR      Byte(5)
```

Now you have five byte variables named `userEntry`: `userEntry(0)`, `userEntry(1)`, `userEntry(2)`, `userEntry(3)`, and `userEntry(4)`.

The **DEBUGIN** command has a formatter called **STR** that automatically loads characters into an array. For example, you can use:

```
DEBUGIN STR userEntry \5
```

If you type five characters in the Debug Terminal's Transmit Windowpane, the first will get placed in `userEntry(0)`, the second will get placed in `userEntry(1)`, etc.

There is a PBASIC keyword called **EXIT** that you can use to break out of a loop. To check a password, you can use an **IF...THEN** statement with an **EXIT** command to cause the loop to end early if not all the characters are identical. When the loop ends early, it means that `index` has not counted all the way up to five, which in turn means the password was not correct:

```
FOR index = 0 TO 4
  READ Password + index, temp
  IF temp <> userEntry(index) THEN EXIT
NEXT

IF index <> 5 THEN
  DEBUG CR, "Password not correct.", CR
ENDIF
```

This next example program places the **DEBUGIN** command, the **FOR...NEXT** loop, and the **IF...THEN** statement inside a **DO...LOOP UNTIL** statement that keeps executing until the value of index gets to 5 (indicating the correct password was entered).

Example Program: PasswordChecker.bs2

Figure 10-7 shows the Debug Terminal displayed by PasswordChecker.bs2. When run, this program will wait for you to type the letters “pass!” in response to the prompt.

- ✓ Enter and run PasswordChecker.bs2.
- ✓ Try entering some letter combinations that are not the password, then enter the letter combination that is the password: “pass!”.

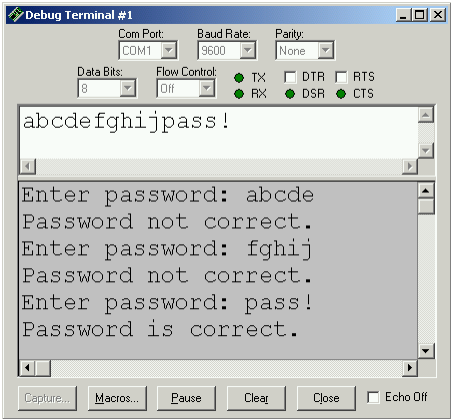


Figure 10-7
Entering Password
into the Transmit
Windowpane

*Click the Transmit
(upper) Windowpane
and enter the
password.*

10

```
' What's a Microcontroller - PasswordChecker.bs2
' Check password entered in Debug Terminal's Transmit Windowpane.

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

Password      DATA    "pass!"          ' Store "secret" password here.

index         VAR      Nib              ' Index variable.
temp          VAR      Byte             ' Stores single char.
userEntry     VAR      Byte(5)          ' Store user entered password.

DO

    DEBUG "Enter password: "            ' User instructions.
```

```

DEBUGIN STR userEntry \5           ' Get user input password.

FOR index = 0 TO 4                 ' Check array against DATA
  READ Password + index, temp      ' Get next password char
  IF temp <> userEntry(index) THEN EXIT ' Compare to user input,
NEXT                               ' exit if not equal.

IF index <> 5 THEN                   ' If exit, then index not equal
  DEBUG CR, "Password not correct.", CR ' to 5 and pass is not correct.
ENDIF

LOOP UNTIL index = 5              ' Only get out of loop when
                                  ' index = 5.

DEBUG CR, "Password is correct;", CR, ' Program can move on when
  "program can continue..."        ' password is correct.
END

```

Your Turn – Modifying Passwords

- ✓ Modify the **Password DATA** directive so that it uses a different five character password.
- ✓ By changing five different values in the program, you can also modify it so that it accepts a four character password instead of a five character password.

Modifying Password Checker for Use in a Larger Program

The goal is to make it easy to fit the PasswordChecker.bs2 program into the example program from Activity #3. Two things will help. First, the password checking program should be modified so that it does most of its work in a subroutine. The different parts of the program should also be labeled with commented headings to help make it clear how to combine the two programs.

Example Program: ReusablePasswordChecker.bs2

ReusablePasswordChecker.bs2 works the same as PasswordChecker.bs2, but it now uses a subroutine to do the work, and it has been reorganized into labeled sections.

- ✓ Verify that the program still works the same as PasswordChecker.bs2.
- ✓ Examine how the **DO...LOOP UNTIL** code block was placed into a subroutine.

```

' -----[ Title ]-----
' What's a Microcontroller - ReusablePasswordChecker.bs2
' Check password entered in Debug Terminal's Transmit Windowpane.

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

' -----[ DATA Directives ]-----

Password      DATA      "pass!"           ' Store "secret" password here.

' -----[ Variable Declarations ]-----

index          VAR      Nib                ' Index variable.
temp           VAR      Byte              ' Stores single char.
userEntry      VAR      Byte(5)           ' Store user entered password.

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

GOSUB Check_Password

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

' There is no main routine in this program.

DEBUG CR, "All Done"

END

' -----[ Subroutine - Check for Correct Password ]-----

Check_Password:

DO

    DEBUG "Enter password: "           ' User instructions.

    DEBUGIN STR userEntry \5          ' Get user input password.

    FOR index = 0 TO 4                ' Check array against DATA
        READ Password + index, temp   ' Get next password char
        IF temp <> userEntry(index) THEN EXIT ' Compare to user input,
    NEXT                               ' exit if not equal.

    IF index <> 5 THEN                  ' If exit, then index not equal
        DEBUG CR,"Password not correct.", CR ' to 5 and pass is not correct.
    ENDIF

    LOOP UNTIL index = 5              ' Only get out of loop when
                                      ' index = 5.

```

```

DEBUG CR, "Password is correct."      ' Program can move on when
                                      ' password is correct.

RETURN                                ' Return when pass is correct.

```

Advanced Topic: Your Turn – Combining the two Programs

Now that both `TerminalOperatedSensorArray.bs2` and `ReusablePasswordChecker.bs2` have both been tested, the task is to combine the two. The final program should check your password before allowing you to choose which sensor to read using the Debug Terminal.

- ✓ You can open both programs (`ReusablePasswordChecker.bs2` and `TerminalOperatedSensorArray.bs2`) in the BASIC Stamp Editor.
- ✓ Save `TerminalOperatedSensorArray.bs2` as `PasswordedSensorTerminal.bs2`

You will need to tab between each program while copying sections from `ReusablePasswordChecker.bs2` and pasting them into `PasswordedSensorTerminal.bs2`.

- ✓ Copy the **Password DATA** directive (including the commented heading with the dashed line) from `ReusablePasswordChecker.bs2` into `PasswordedSensorTerminal.bs2` just before the **I/O Definitions** section.
- ✓ Copy and paste the variable declarations from `ReusablePasswordChecker.bs2` into `PasswordedSensorTerminal.bs2`. These variable declarations are added to the ones in `PasswordedSensorTerminal.bs2`, so don't worry about copying and pasting the commented **Variables** heading.
- ✓ Copy the **Initialization** section (including the commented heading) from `ReusablePasswordChecker.bs2`, and paste it between the **Variables** section and the **Main Routine** in `PasswordedSensorTerminal.bs2`.
- ✓ Copy the entire **Subroutine** section from `ReusablePasswordChecker.bs2` and paste it after the end of the last subroutine in `PasswordedSensorTerminal.bs2`.
- ✓ Test `PasswordedSensorTerminal.bs2`, and see if it works, debug as needed.

SUMMARY

This chapter introduced the technique of individually testing each circuit-subsystem before integrating it into a larger system. This chapter also introduced the `ON...GOSUB` command, which is particularly useful for menu systems. The useful `PIN` directive was demonstrated as a way to name your I/O pins and then let the BASIC Stamp Editor figure out whether to read an input or write to an output. A password program was used to introduce variable arrays, the `EXIT` command, and the `DEBUGIN` command's `STR` formatter. The password program was then integrated into a larger sensor terminal program to give it more functionality.

Questions

1. When should you test subsystems individually before trying to make them work together? Why?
2. How many programs did you use in this chapter that were from other chapters?
3. How does the `PIN` directive differ from the `CON` and `VAR` directives?
4. What's the difference between `EXIT` and `END`?

Exercises

1. Describe the general 3-step process you would use to add a piezospeaker circuit to your project.
2. Modify PasswordChecker.bs2 from page 293 so that the message "you entered: " appears in the Debug Terminal along with the text of the password.

10

Projects

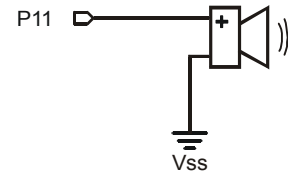
1. Add a piezospeaker circuit to the sensor array you developed in this chapter, and write a program to test it.
2. Modify TerminalOperatedSensorArray.bs2 so that it has a 5th menu item that makes the piezospeaker beep at 2 kHz for 1.5 seconds.
3. While working on one of the activities and projects in this book you may have thought to yourself, "hey, I could use this to build a >> insert your project here: _____ <<!" Use the material you have learned in this book to invent a gadget or gizmo of your own design.

Solutions

- Q1. Whenever possible! It is much easier to find, isolate and prevent problems.
- Q2. Two. `ReadPushbuttonState.bs2` from Chapter #3 and `ReadPotWithRcTime.bs2` from Chapter #5.
- Q3. While the `CON` directive assigns a name to a number, and while the `VAR` directive assigns a name to RAM, the `PIN` directive assigns a name to a BASIC Stamp I/O pin. This name can then be used in place of the I/O pin number throughout the program. PBASIC figures out whether you are using it as an output to send a high/low signal or as an input variable to store the state sensed by the I/O pin (1 or 0).
- Q4. `END` ends the program, it doesn't run anymore. By contrast `EXIT` just breaks out of the loop, and then the program keeps on running.
- E1. a. Build the piezospeaker circuit on the breadboard.
b. Run `TestPiezoWithFreqout.bs2` from Chapter 8 to test the speaker.
c. Integrate the piezospeaker into the program.
- E2. The key to solving this problem is to use the `STR` formatter with `DEBUG`. The text introduced the `STR` formatter with `DEBUGIN`, so it makes sense to guess that the `STR` formatter will also work with `DEBUG`. This can be done by adding this line of code below the `DEBUGIN` statement.
- ```
DEBUG CR, CR, "You entered: ", STR userEntry, CR
```
- P1. Project 1 uses the same schematic as Figure 10-1. p. 282 with the addition of a piezo speaker connected to P11.

```
' What's a Microcontroller -
' TestPiezoWithFreqout.bs2
' Send a tone to the piezo speaker
' using the FREQOUT command.
'{$STAMP BS2}
'{$PBASIC 2.5}

DEBUG "Tone sending...", CR
FREQOUT 11, 1500, 2000
DEBUG "Tone done."
```



- P2. Project 2 uses the same schematic as Project 1.

```
' What's a Microcontroller - Ch10Prj02_AddSpeakerToSensorArray.bs2
' Use Debug Terminal to choose to read one of five sensors.
' Project 2 - Add a 5th menu item that makes a piezo speaker beep
' for 1.5 seconds at a frequency of 2kHz.
' {$STAMP BS2}
' {$PBASIC 2.5}
```

```

' -----[I/O Definitions]-----
Pb1Pin CON 3
Pb2Pin CON 4
PotPin CON 7
PhotoPin CON 9
SpkrPin CON 11

' -----[Constants]-----
DelayRc CON 100
DelayReader CON 1500

' -----[Variables]-----
request VAR Nib
time VAR Word

' -----[Main Routine]-----
DO
 GOSUB Display_Menu
 GOSUB Get_Request
 ON request GOSUB Read_Pushbutton_1, Read_Pushbutton_2,
 Read_Pot, Read_Photoresistor, Beep_Speaker
LOOP

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

Display_Menu:
 DEBUG CLS, "MENU: ", CR,CR,
 "1) Pushbutton 1", CR,
 "2) Pushbutton 2", CR,
 "3) Potentiometer", CR,
 "4) Photoresistor", CR,
 "5) Piezo Speaker", CR, CR
 RETURN

Get_Request:
 DEBUGIN DEC1 request
 request = request - 1
 RETURN

Read_Pushbutton_1:
 DEBUG CLS, ? IN3
 PAUSE DelayReader
 RETURN

Read_Pushbutton_2:

```



```

DEBUG CLS, ? IN4
PAUSE DelayReader
RETURN

Read_Pot:
HIGH 7
PAUSE DelayRc
RCTIME 7, 1, time
DEBUG CLS, DEC time, " "
PAUSE DelayReader
RETURN

Read_Photorresistor:
HIGH 9
PAUSE DelayRc
RCTIME 9, 1, time
DEBUG CLS, DEC time, " "
PAUSE DelayReader
RETURN

Beep_Speaker:
DEBUG CLS, "2000 Hz beep for 1.5 seconds", CR
FREQOUT SpkrPin, 1500, 2000 ' 2000 Hz beep for 1.5 s
RETURN

```

- P3. If you have created an interesting application with your What's a Microcontroller? Kit and you want to share it with other students and teachers, consider joining our Stamps In Class Yahoo Group. See the Web Site and Discussion Lists section right after the title page at the front of this book for details.

### **Further Investigation**

Please go back to the Preface and read the section titles The Stamps in Class Curriculum. All of the books listed there are available for free download from [www.parallax.com](http://www.parallax.com) and are also on the Parallax CD. They contain a wealth of knowledge and instructions for continuing your explorations into electronics, programming, robotics, and engineering. In addition, we highly recommend that you visit [www.parallax.com](http://www.parallax.com) and check out our many free downloads and applications for industry professionals, educators, students, hobbyists, and the naturally curious.

## Appendix A: 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 A-1 and a mini-CD ROM with drivers for use with various operating systems including Microsoft Windows®.



**Figure A-1**  
FTDI's  
US232B/LC USB  
to Serial Adapter

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



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

## Appendix B: Equipment and Parts Lists

### B



**Parts lists are subject to change:** Please note that the part numbers and bills of materials cited in this appendix are subject to change without notice. If you have questions about a particular part or quantity, please contact the Parallax using [www.parallax.com](http://www.parallax.com) → Company → Contact Parallax link, or email [stampsinclass@parallax.com](mailto:stampsinclass@parallax.com).

To complete the exercises in this book, you need to have one of the following Parallax hardware options:

#### Option 1:

- **Board of Education® Full Kit (#28102) - AND-**
- **What's a Microcontroller Parts Kit (#28152 with text, #28122 without text)**

These two kits are also sold separately. The Board of Education Full Kit (contents listed below) is the core equipment of the Stamps in Class curriculum, and it can be used with any of the Stamps in Class texts and kits.

| Board of Education® Full Kit (#28102)                 |                                 |          |
|-------------------------------------------------------|---------------------------------|----------|
| Parts and quantities subject to change without notice |                                 |          |
| Parallax Part #                                       | Description                     | Quantity |
| 550-00022                                             | Board of Education              | 1        |
| 800-00016                                             | Pluggable wires                 | 10       |
| BS2-IC                                                | BASIC Stamp® 2 module           | 1        |
| 800-00003                                             | Serial cable                    | 1        |
| 750-00008                                             | DC power supply – 9 V, 300 mA   | 1        |
| 27000                                                 | Parallax CD – includes software | 1        |
| 700-00037                                             | Rubber feet – set of 4          | 1        |

You may purchase the What's a Microcontroller Parts Kit alone (#28122), or with the parts and the *What's a Microcontroller?* printed text together (#28152). These parts kits are assembled to support the activities and projects in the current printed version of the text. The What's a Microcontroller Parts Kit contents are listed in the table on the following page.

| <b>What's a Microcontroller Parts Kit #28122</b><br><b>What's a Microcontroller Parts &amp; Text #28152</b><br>Parts and quantities subject to change without notice |                                                                     |          |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------|----------|
| Parallax Part #                                                                                                                                                      | Description                                                         | Quantity |
| 150-01020                                                                                                                                                            | Resistor, 5%, 1/4W, 1 k $\Omega$                                    | 10       |
| 150-01030                                                                                                                                                            | Resistor, 5%, 1/4W, 10 k $\Omega$                                   | 4        |
| 150-01040                                                                                                                                                            | Resistor, 5%, 1/4W, 100 k $\Omega$                                  | 2        |
| 150-02020                                                                                                                                                            | Resistor, 5%, 1/4W, 2 k $\Omega$                                    | 2        |
| 150-02210                                                                                                                                                            | Resistor, 5%, 1/4W, 220 $\Omega$                                    | 6        |
| 150-04710                                                                                                                                                            | Resistor, 5%, 1/4W, 470 $\Omega$                                    | 6        |
| 152-01031                                                                                                                                                            | Potentiometer - 10 k $\Omega$                                       | 1        |
| 200-01031                                                                                                                                                            | Capacitor, 0.01 $\mu$ F, 50 V                                       | 1        |
| 200-01040                                                                                                                                                            | Capacitor, 0.1 $\mu$ F, 100 V                                       | 2        |
| 201-01080                                                                                                                                                            | Capacitor, 1000 $\mu$ F, 10 V                                       | 1        |
| 201-03080                                                                                                                                                            | Capacitor 3300 $\mu$ F, 16 V                                        | 1        |
| 28123                                                                                                                                                                | <i>What's a Microcontroller?</i> text<br>(Included in #28152 only.) | 1        |
| 350-00001                                                                                                                                                            | LED - Green - T1 3/4                                                | 2        |
| 350-00005                                                                                                                                                            | LED - Bi-Color - T1 3/4                                             | 1        |
| 350-00006                                                                                                                                                            | LED - Red - T1 3/4                                                  | 2        |
| 350-00007                                                                                                                                                            | LED - Yellow - T1 3/4                                               | 2        |
| 350-00009                                                                                                                                                            | Photoresistor                                                       | 1        |
| 350-00027                                                                                                                                                            | 7-segment LED Display                                               | 1        |
| 400-00002                                                                                                                                                            | Pushbutton – Normally Open                                          | 2        |
| 451-00303                                                                                                                                                            | 3 Pin Header – Male/Male                                            | 1        |
| 500-00001                                                                                                                                                            | Transistor – 2N3904                                                 | 1        |
| 604-00010                                                                                                                                                            | 10 k $\Omega$ digital potentiometer                                 | 1        |
| 800-00016                                                                                                                                                            | 3" Jumper Wires – Bag of 10                                         | 2        |
| 900-00001                                                                                                                                                            | Piezo Speaker                                                       | 1        |
| 900-00005                                                                                                                                                            | Parallax Standard Servo                                             | 1        |

**Option 2:**

- **BASIC Stamp What's a Microcontroller Kit (#90005)**

This kit features the What's a Microcontroller Parts & Text, with a HomeWork Board and accessories that are otherwise sold separately. The BASIC Stamp HomeWork Board can be used with the *What's a Microcontroller?* text in place of the Board of Education and BASIC Stamp 2 module. The HomeWork Board can also be used in the majority of the activities in the Stamps in Class curriculum, though occasional circuit modifications are necessary for certain activities. The BASIC Stamp What's a Microcontroller Kit includes the following items:

| <b>BASIC Stamp® What's a Microcontroller Kit (#90005)</b> |                                    |                 |
|-----------------------------------------------------------|------------------------------------|-----------------|
| Parts and quantities subject to change without notice     |                                    |                 |
| <b>Parallax Part #</b>                                    | <b>Description</b>                 | <b>Quantity</b> |
| 555-28158                                                 | HomeWork Board™ with breadboard    | 1               |
| 28123                                                     | What's a Microcontroller Text      | 1               |
| 27000                                                     | Parallax CD – includes software    | 1               |
| 800-00003                                                 | Serial cable                       | 1               |
| 28122                                                     | What's a Microcontroller Parts Kit | 1               |
| 700-00037                                                 | Rubber feet – set of 4             | 1               |

**A note to educators:** The HomeWork board is available separately in packs of 10 as an economical solution for classroom use, costing significantly less than the Board of Education + BASIC Stamp 2 module. Please contact the Parallax Sales Team toll free at (888) 512-1024 for quantity pricing.

| <b>BASIC Stamp® HomeWork Board™ Ten-Pack (#28158)</b> |                                                                          |                 |
|-------------------------------------------------------|--------------------------------------------------------------------------|-----------------|
| <b>Parallax Part #</b>                                | <b>Description</b>                                                       | <b>Quantity</b> |
| 28158                                                 | BASIC Stamp® HomeWork Board™<br>(BASIC Stamp 2 is built into the board.) | 10              |

## Appendix C: BASIC Stamp and Carrier Board Components and Functions

### The BASIC STAMP® 2

C

Figure C-1 shows a close-up of Parallax Inc.'s BASIC Stamp® 2 microcontroller module. Its major components and their functions are indicated by labels.

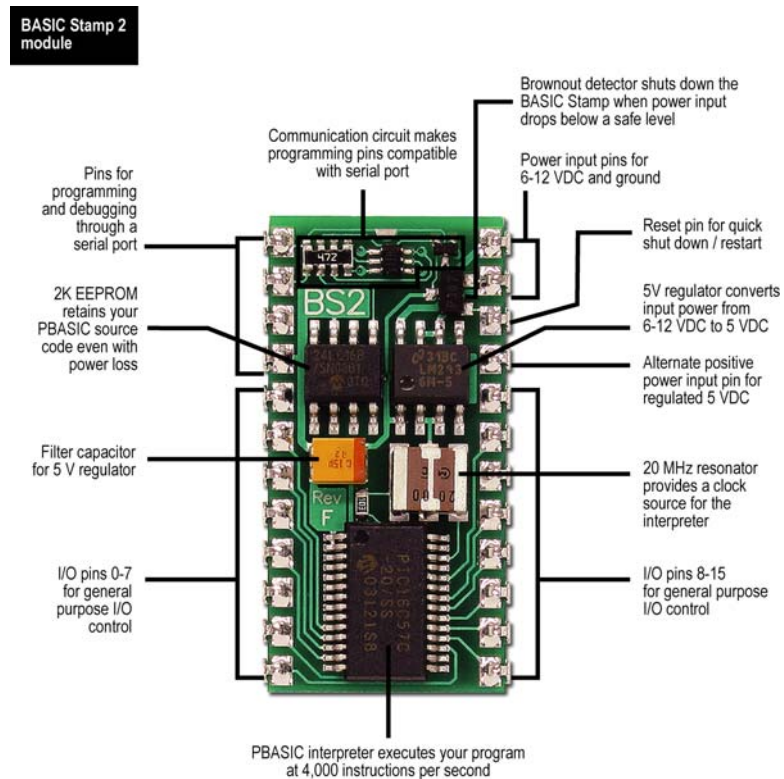


**Figure C-1:** Parallax Inc.'s BASIC Stamp® 2 Microcontroller Module Components and their Functions

## The Board of Education® Rev C

Parallax Inc.'s Board of Education® Rev C carrier board is shown in Figure C-2. Its major components and their functions are indicated by labels.

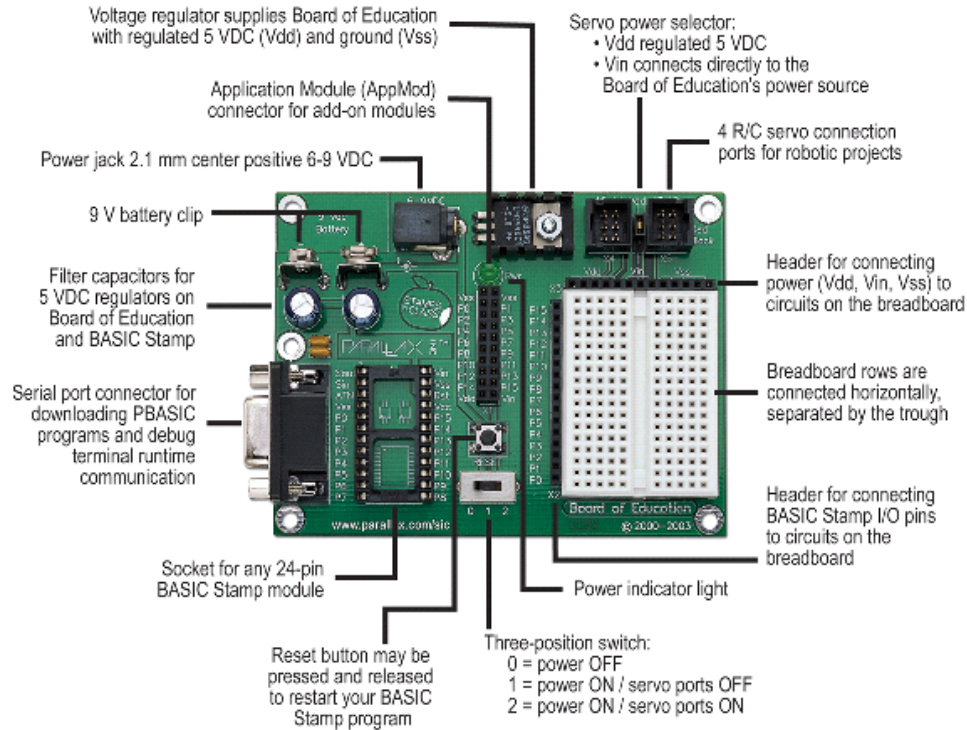


**Figure C-2:** Parallax, Inc.'s Board of Education® Rev C Carrier Board for BASIC Stamp® Microcontroller Modules

### The BASIC Stamp® HomeWork Board™

Parallax Inc.'s BASIC Stamp® HomeWork Board™ project platform is shown in Figure C-3. Its major components and their functions are indicated by labels.

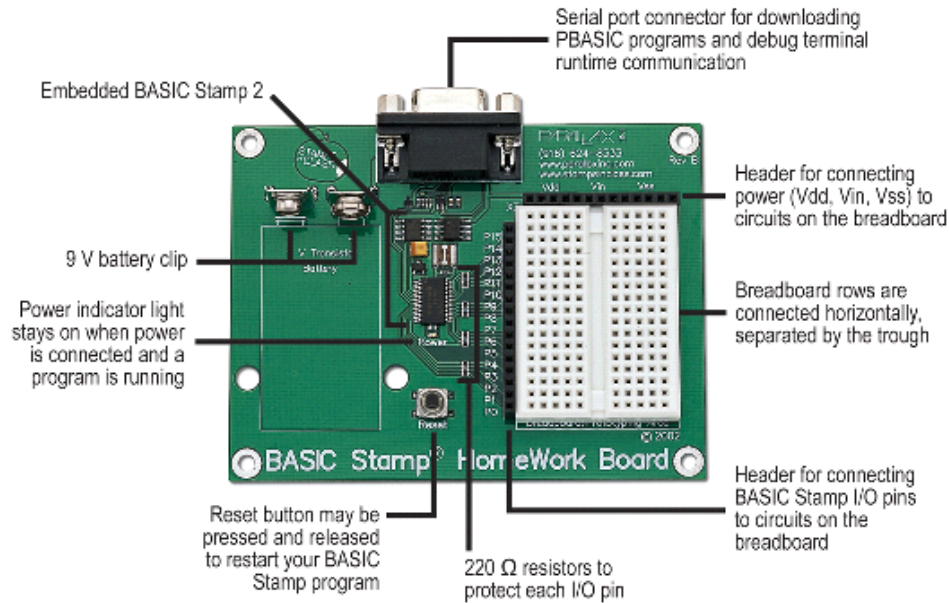


**Figure C-3:** Parallax Inc.'s BASIC Stamp® HomeWork Board™ project platform features a surface-mounted BASIC Stamp® 2 microcontroller module.



## The Board of Education® Rev B

Figure C-4 shows Parallax, Inc.'s Board of Education® Rev B carrier board. Its major components and their functions are indicated by labels.

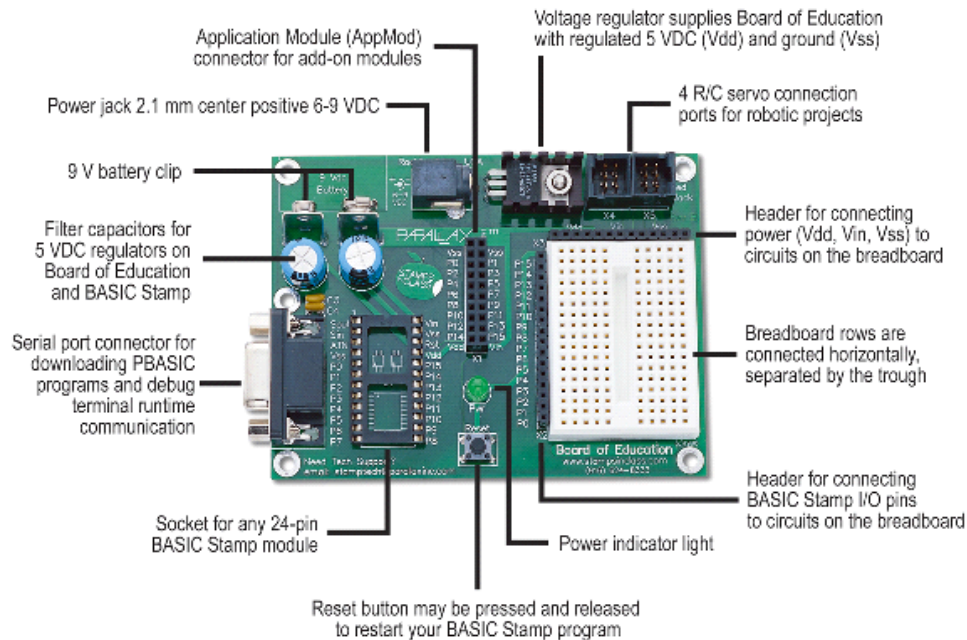


**Figure C-4:** Parallax, Inc.'s Board of Education® Rev B Carrier Board for BASIC Stamp® modules

## Appendix D: Batteries and Power Supplies

### 9 V BATTERIES

For best results, 9 V batteries are recommended.



**9 V battery specifications:** Look for batteries with ratings similar these:

| Not Rechargeable                                             | Rechargeable                                                                                                        |
|--------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• Alkaline</li> </ul> | <ul style="list-style-type: none"> <li>• Ni-Cad (Nickel Cadmium)</li> <li>• Ni-MH (Nickel Metal Hydride)</li> </ul> |

For best results, the battery's milliamp hour (mAh) rating should be 100 or higher.

Not all chargers work for both types of batteries. Make sure that your charger is recommended for the battery you are using (either Ni-Cad or Ni-MH).

Follow all battery and charger instructions and caution statements.

D

### PARALLAX DC SUPPLIES

Parallax carries several power supplies that can be used with the Board of Education Rev C only. For the servo experiments in this text, the jumper between the X4 and X5 servo headers should be set to Vdd. The supplies listed in Table D-1 are designed for AC outlets in the USA, and both have 2.1 mm center-positive barrel plugs that connect to the Board of Education's barrel jack.

| Table D-1: Power Supplies You Can Get from Parallax, Inc. |       |    |        |      |
|-----------------------------------------------------------|-------|----|--------|------|
| Parallax Part #                                           | Input |    | Output |      |
|                                                           | VAC   | Hz | VDC    | mA   |
| 750-00008                                                 | 120   | 60 | 9      | 300  |
| 750-00009                                                 | 120   | 60 | 7.5    | 1000 |

## GENERIC DC SUPPLIES

For best results with the BASIC Stamp HomeWork Board or any of the Board of Education Revisions, use a DC supply (also called an AC adapter) with the following ratings:

### Input

This depends on which country you live in and the amplitude and frequency of the AC power at the wall outlets. For the USA and Canada, the input should be 120 VAC, 60 Hz.

### Output

6 VDC, 800 mA

The mA rating can be higher. A 6 V, 1000 mA supply would be acceptable, for example.

### Plug

The Board of Education has both a barrel jack, which can be connected to a barrel plug, and a 9 V battery connector, which can be connected to a 9 V battery extension. The HomeWork Board has only the 9 V battery connector.

### **Barrel Plug**

Figure D-1 shows DC supply commonly used with the BASIC Stamp and Board of Education. It has a 2.1 mm center-positive barrel plug along with the center positive symbol that is evident on its label.



**Figure D-1**  
DC Supply with Barrel Plug  
and Center Positive Symbol

**D**



### 9 V Battery Extension

Figure D-2 shows an AC adapter connected to a 9 V battery extension that can be used with the BASIC Stamp HomeWork Board. See WARNING discussed next.



**Figure D-2**  
AC Adapter with 9 V  
Battery Extension

**WARNING - Beware of Universal AC Adapters and Reversed Supply Terminals**

Figure D-3 shows a common mistake that should be avoided with universal adapters. Many of these allow you to reverse the terminals on the 9 V battery extension. Although it cannot hurt the BASIC Stamp, Board of Education or Homework Board, it can destroy the Parallax Standard Servo connected to Vin in a matter of seconds. The only system that can protect the servo from this mistake is the Board of Education Rev C (with jumper set to Vdd).

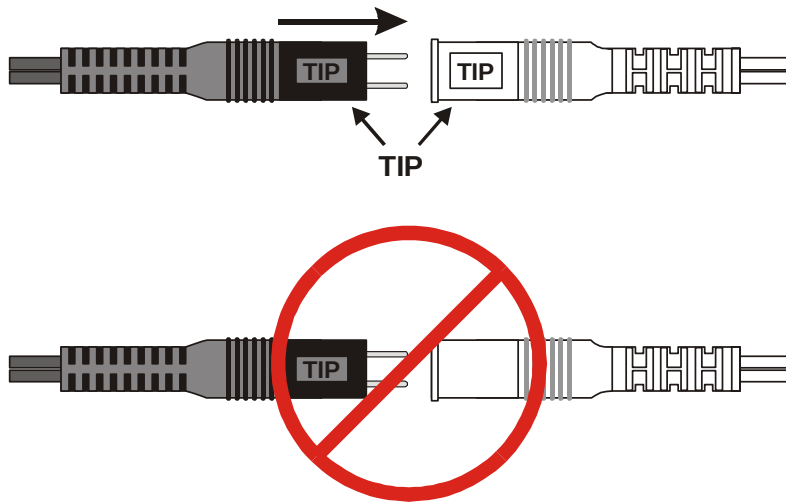


**Figure D-3**  
Polarity on  
Universal  
Adapters

**Beware of “Battery Replacers”**

Many battery replacers are designed to supply appliances with low current draw. With current ratings in the neighborhood of 10 mA, their output capacities are insufficient for many of the activities in this text. For example, two LEDs connected to 220  $\Omega$  resistors draw a total of 14.5 mA, and the BASIC Stamp takes an additional 3 to 7 mA. A servo draws upwards of 100 mA, and that definitely won't work with a “battery replacer”.

NOTE: It's pretty easy to tell when a circuit is drawing more current than the supply can deliver because the Pwr LED on the Board of Education (or HomeWork Board) flickers and/or goes dim.

## Appendix E: Trouble-Shooting

---

Here is a list of things to try to quickly fix any difficulties getting the BASIC Stamp Editor to communicate with the 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.
- ✓ 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 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. If you find that label, do not try to use it to program the BASIC Stamp.
- ✓ 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, aligning the reference notch as shown in Figure 1-28 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 to make sure that none of the pins folded under the module instead of sinking into their sockets on the Board of Education. Disconnect power first, then press down firmly on the module with your thumb.

If your Identification Window looks similar to Figure E-1, it means that the BASIC Stamp Editor cannot find your BASIC Stamp on any COM port. Note that the Loopback and Echo columns show “No.” If you have this problem, try the following:

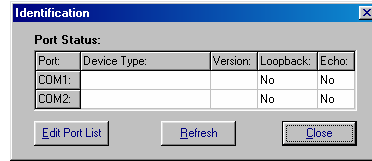


**Figure E-1**  
Identification Window

*Example: BASIC Stamp 2 not found on COM ports.*

- ✓ Close the Identification Window.
- ✓ Make sure the serial cable is properly connected.
- ✓ Try the *Run → Identify* test again.
- ✓ 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, and then try the *Run → Identify* test again.
- ✓ 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.

**E**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 F: More about Electricity

**What's an Electron?** An electron is one of the three fundamental parts of an atom; the other two are the proton and the neutron. One or more protons and neutrons stick together in the center of the molecule in an area called the nucleus. Electrons are very small in comparison to protons and neutrons, and they orbit around the nucleus. Electrons repel each other, and electrons and protons attract to each other.

**What's Charge?** The tendency for an electron to repel from another electron and attract to a nearby proton is called negative charge. The tendency for a proton to repel from another proton and attract an electron is called positive charge. When a molecule has more electrons than protons, it is said to be negatively charged. If a molecule has fewer electrons than protons, it is said to be positively charged. If a molecule has the same number of protons and electrons, it is called neutrally charged.

**What's Voltage?** Voltage is like electrical pressure. When a negatively charged molecule is near a positively charged molecule, the extra electron on the negatively charged molecule tries to get from the negatively charged molecule to the positively charged molecule. Batteries keep a compound with negatively charged molecules separated from a compound with positively charged molecules. Each of these compounds is connected to one of the battery's terminals; the positively charged compound is connected to the positive (+) terminal, and the negative compound is connected to the negative (-) terminal.



**The volt** is a measurement of electric pressure, and it's abbreviated with a capital V. You may already be familiar with the nine volt (9 V) battery used to supply power to the Board of Education or HomeWork Board. Other common batteries include the 12 V batteries found in cars and the 1.5 V AA batteries used in calculators, handheld games, and other devices.

**What's Current?** Current is a measure of the number of electrons per second passing through a circuit. Sometimes the molecules bond in a chemical reaction that creates a compound (that is neutrally charged). Other times, the electron leaves the negatively charged molecule and joins the positively charged molecule by passing through a circuit like the one you just built and tested. The letter most commonly used to refer to current in schematics and books is capital 'I'.

**What's an amp?** An amp is the basic unit of current, and the notation for the amp is the capital 'A'. Compared to the circuits you are using with the BASIC Stamp, an amp is a very large amount of current. It's a convenient value for describing the amount of current that a car battery supplies to headlights, the fan that cool a car's engine, and other high power devices.

**What's Resistance?** Resistance is the element in a circuit that slows down the flow of electrons (the current) from a battery's negative terminal to its positive terminal.

**The ohm** is the basic measurement of resistance. It has already been introduced and it's abbreviated with the Greek letter omega ( $\Omega$ ).

**What's a Conductor?** Copper wire has almost no resistance, and it's called a conductor.

## BONUS ACTIVITY: OHMS LAW, VOLTAGE, AND CURRENT

This activity applies some of the definitions just discussed.

### Ohm's Law Parts

- (1) Resistor – 220  $\Omega$  (red-red-brown)
- (1) Resistor – 470  $\Omega$  (yellow-violet-brown)
- (1) Resistor – 1 k $\Omega$  (brown-black-red)
- (1) Resistor – 2 k $\Omega$  (red-black-red)
- (1) LED – any color

### Test Circuit

The resistance value of  $R_i$  in Figure F-1 can be changed. Lower resistance allows more current through the LED, and it glows more brightly. Higher resistance values will cause the LED to look dim because they do not allow as much current to pass through the circuit.

- ✓ Disconnect power to your Board of Education or HomeWork Board whenever you modify the circuit.
- ✓ Build the circuit shown starting with a 220  $\Omega$  resistor.
- ✓ Modify the circuit by replacing the 220  $\Omega$  resistor with a 470  $\Omega$  resistor. Was the LED less bright?
- ✓ Repeat using the 1 k $\Omega$  resistor, then the 2 k $\Omega$  resistor, checking the change in brightness each time.

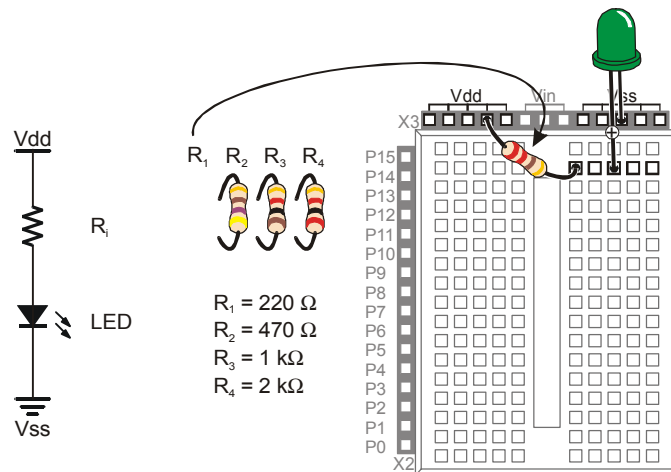


**Figure F-1**  
LED Current Monitor

F

If you are using a 9 V battery, you can also compare the brightness of a different voltage source,  $V_{in}$ .  $V_{in}$  is connected directly to the 9 V battery's + terminal, and  $V_{ss}$  is connected directly to the battery's negative terminal.  $V_{dd}$  is called regulated 5 V. That's about half the voltage of the 9 V battery.

- ✓ If you are not using a 9 V battery, stop here and skip to the Calculating the Current section below. Otherwise, continue.
- ✓ Start with the circuit shown in Figure F-1, but use a 1 k $\Omega$  resistor.
- ✓ Make a note of how bright the LED is.
- ✓ Disconnect power
- ✓ Modify the circuit by disconnecting the resistor lead from  $V_{dd}$  and plugging it into  $V_{in}$ .
- ✓ When you plug the power back in, is the LED brighter? How much brighter?



**DO NOT** try the  $V_{in}$  experiment with a 220 or 470  $\Omega$  resistor, it will supply the LED with more current than it is rated for.

### Calculating the Current

The BASIC Stamp Manual has some rules about how much current it can supply to circuits. If you don't follow these rules, you may end up damaging your BASIC Stamp.

The rules have to do with how much current an I/O pin is allowed to deliver and how much current a group of I/O pins is allowed to deliver.

|                                                                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | <p><b>Current Rules for BASIC Stamp I/O Pins</b></p> <ul style="list-style-type: none"><li>• An I/O pin can “source” 20 mA. In other words, if you send the <b>HIGH</b> signal to an I/O pin, it should not supply the LED circuit with more than 20 mA.</li><li>• If you rewire the LED circuit so that the BASIC Stamp makes the LED turn on when you send the <b>LOW</b> command, an I/O pin can “sink” up to 25 mA.</li><li>• P0 through P7 can only source up to 20 mA. Likewise with P8 through P15. If you have lots of LED circuits, you will need larger resistors so that you don't draw too much current.</li></ul> |
|-----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

If you know how to calculate how much current your circuit will use, then you can decide if it's OK to make your LEDs glow that brightly. Every electronic component has rules for what it does with voltage, resistance, and current. For the light emitting diode, the rule is a value called the diode forward voltage. For the resistor, the rule is called Ohm's Law. You need to figure out how much current your LED circuit is using. There are also rules for how current and voltage add up in circuits. These are called Kirchoff's Voltage and Current Laws.



**Vdd – Vss = 5 V** The voltage (electrical pressure) from Vdd to Vss is 5 V. This is called regulated voltage, and it works about the same as a battery that is exactly 5 V.

**Vin – Vss = 9 V** If you are using 9 V battery, the voltage from Vin to Vss is 9 V. Be careful. If you are using a voltage regulator that plugs into the wall, even if it says 9 V, it could go as high as 18 V.

**Ground and/or reference** refer to the negative terminal of a circuit. When it comes to the BASIC Stamp and Board of Education, Vss is considered the ground reference. It is zero volts, and if you are using a 9 V battery, it is that battery's negative terminal. The battery's positive terminal is 9 V. Vdd is 5 V (above the Vss reference of 0 V), and it is a special voltage made by a voltage regulator chip to supply the BASIC Stamp with power.

**Ohm's Law:  $V = I \times R$**  The voltage measured across a resistor's terminals (V) equals the current passing through the resistor (I) times the resistor's resistance (R).

**Diode Forward Voltage:** When an LED is emitting light, the voltage measured from anode to cathode will be around 1.6 V. Regardless of whether the current passing through it is a large or a small value, the voltage will continue to be approximately 1.6 V.

**Kirchoff's Voltage Law Simplified: voltage used equals voltage supplied.** If you supply a circuit with 5 V, the number of volts all the parts use had better add up to 5 V.

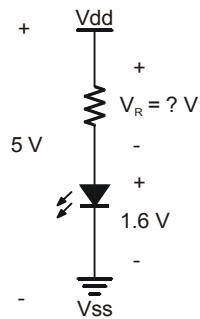
**Kirchoff's Current Law Simplified: current in equals current out.** The current that enters an LED circuit from Vdd is the same amount of current that leaves it through Vss. Also, if you connect three LEDs to the BASIC Stamp, and each LED circuit draws 5 mA, it means the BASIC Stamp has to supply all the circuits with a total of 15 mA.

### Example Calculation: One Circuit, Two Circuits

Calculating how much current an LED circuit draws takes two steps:

1. Figure out the voltage across the resistor
2. Use Ohm's Law to figure out the current through the resistor.

Figure F-2 shows how to figure out the voltage across the resistor. The voltage supplied is on the left; it's 5 V. The voltages used are on the right. The voltage we don't know at the start is  $V_R$ , the voltage across the resistor. But, we do know that the voltage across the LED is 1.6 V (the diode forward voltage). We also know that the voltage across the parts has to add up to 5 V because of Kirchoff's Voltage Law. The difference between 5 V and 1.6 V is 3.4 V, so that must be the voltage across the resistor  $V_R$ .



$$V_R + 1.6V = 5V$$

$$V_R = 5V - 1.6V$$

$$V_R = 3.4V$$

**Figure F-2**  
Voltage Across  
the Circuit,  
Resistor, and  
LED



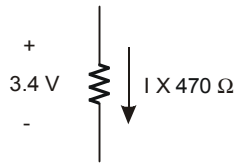
**Kilo is metric for 1000.** The metric way of saying 1000 is kilo, and it's abbreviated with the lower-case k. Instead of writing 1000  $\Omega$ , you can write 1 k $\Omega$ . Either way, it's pronounced one-kilo-ohm. Likewise, 2000  $\Omega$  is written 2 k $\Omega$ .

**Milli is metric for 1/1000**, and it is abbreviated with a lower-case m. If the BASIC Stamp supplies an LED circuit with 3.4 thousandths of an amp, that's 3.4 milliamps, or 3.4 mA.

**What's a mA?** Pronounced milliamp, it's the notation for one-one-thousandth-of-an-amp. The 'm' in mA is the metric notation for milli, which stands for 1/1000. The 'A' in mA stands for amps. Put the two together, and you have milliamps, and it's very useful for describing the amount of current drawn by the BASIC Stamp and the circuits connected to it.

**How much current is 7.23 mA?** It's the amount of current the LED shown on the right side of Figure F-3 conducts. You can replace the 470  $\Omega$  resistor with a 220  $\Omega$  resistor, and the circuit will conduct 15.5 mA, and the LED will glow more brightly. If you use a 1000  $\Omega$  resistor, the circuit will conduct 3.4 mA, and the LED will glow less brightly. A 2000  $\Omega$  resistor will cause the LED to glow less brightly still, and the current will be 1.7 mA.

Figure F-3 shows an example of how to calculate the current the circuit uses if the resistor is 470  $\Omega$ . Start with Ohm's Law. You know the answers to V (3.4 V) and R (470  $\Omega$ ). Now, all you have to do is solve for I (the current).



$$V = I \times R$$

$$3.4V = I \times 470\Omega$$

$$I = \frac{3.4V}{470\Omega}$$

$$I = 0.00723V/\Omega$$

$$I = 0.00723A$$

$$I = \frac{7.23}{1000}A$$

$$I = 7.23mA$$

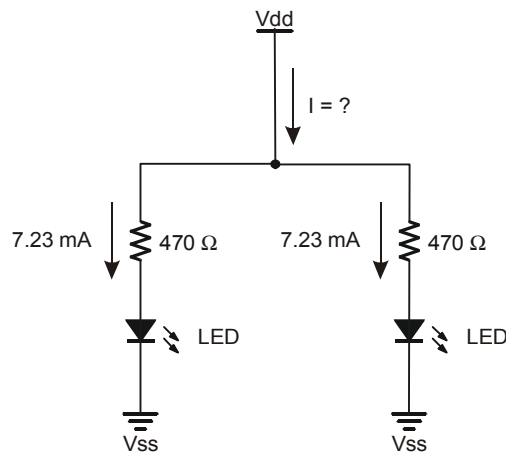
**Figure F-3**  
Current through  
the Resistor



Yes, it's true !  $1A = 1V/\Omega$  (One amp is one volt per ohm).

F

Let's say you turned two LEDs on at the same time. That means that inside the BASIC Stamp, it is supplying the circuits as shown in Figure F-4. Have we exceeded the 20 mA limit? Let's find out. Remember that the simplified version of Kirchhoff's Current Law says that the total current drawn from the supply equals the current supplied to all the circuits. That means that  $I$  in Figure F-4 has to equal the total of the two currents being drawn. Simply add the two current draws, and you'll get an answer of 14.5 mA. You are still under the 20 mA limit, so your circuit is a safe design.



$$I = I_1 + I_2 + \dots I_i$$

$$I = 7.23mA + 7.23mA$$

$$I = 14.5mA$$

**Figure F-4**  
Total Current  
Supplied to  
Two LED  
Circuits

### Your Turn – Modifying the Circuit

- ✓ Repeat the exercise in Figure F-2, but use  $V_{in} - V_{ss} = 9V$ . The answer is  $V_R = 7.4 V$
- ✓ Repeat the exercise in Figure F-3, but use a  $1 k\Omega$  resistor. Answer:  $I = 3.4 mA$ .
- ✓ Use  $V_R = 7.4 V$  to do the exercise in Figure F-3 with a  $1 k\Omega$  resistor. Answer:  $I = 7.4 mA$ .
- ✓ Repeat the exercise shown in Figure F-4 with one of the resistors at  $470 \Omega$  and the other at  $1 k\Omega$ . Answer:  $I = 7.23 mA + 3.4 mA = 10.63 mA$ .



## Appendix G: RTTTL Format Summary

---

This is a summary intended to help make sense out of RTTTL format. The full RTTTL specification can be found published at various web sites. Using any search engine, use the keywords RTTTL specification, to review web pages that include the specification.

Here is an example of an RTTTL format ringtone:

```
TakeMeOutToTheBallgame:d=4,o=7,b=225:2c6,c,a6,g6,e6,2g.6,2d6,p,
2c6,c,a6,g6,e6,2g.6,g6,p,p,a6,g#6,a6,e6,f6,g6,a6,p,f6,2d6,p,2a6
,a6,a6,b6,c, d,b6,a6,g6
```

The text before the first colon is what the cell phone displays as the name of the song. In this case, the ringtone is named:

```
TakeMeOutToTheBallGame:
```

Between the first and second colon, the default settings for the song are entered using d, o, and b. Here is what they mean:

```
d - duration
o - octave
b - beats per minute or tempo.
```

In the TakeMeOutToTheBallGame, the default settings are:

```
d=4,o=7,b=225:
```

The notes in the melody are entered after the second colon, and they are separated by commas. If just the note letter is used, that note will be played for the default duration in the default octave. For example, the second note in TakeMeOutToTheBallGame is:

```
,c,
```

Since it has no other information, it will be played for the default quarter note duration (d=4), in the seventh octave (o=7).

A note could have up to five characters between the commas; here is what each character specifies:

```
,duration note sharp dot octave,
```

For example:

```
,2g#.6,
```



means play the half note G-sharp for  $1\frac{1}{2}$  the duration of a half note, and play it in the sixth octave.

Here are a few examples from TakeMeOutToTheBallGame:

,2g.6, – half note, G, dotted, sixth octave  
,a6, – default quarter note duration, A note played in the sixth octave  
,g#6, – quarter duration, g note, sharp (denoted by #), sixth octave

The character:

,p,

stands for pause, and it is used for rests. With no extra information, the p plays for the default quarter-note duration. You could also play a half note's worth of rest by using:

,2p,

Here is an example of a dotted rest:

,2p.,

In this case the rest would last for a half note plus a quarter note's duration.

# Index

---

- % -
- %, 177
- \* -
- \*, 94
- \*\*, 244
- \*/, 95, 244
- 3 -
- 3-position switch, 16
- 7 -
- 7-Segment Display, 165–70
- A -
- Active-High, 79
- Active-Low, 79
- AD5220 Digital Potentiometer
  - Pin Map, 268
  - Pin Names and Functions, 268
- Algorithm, 96
- American Standard Code for Information Interchange, 30
- Amp, 319
- Anode, 40
- Array, 292
- ASCII, 30, 253
- B -
- Base, 265
- BASIC Stamp
  - BASIC Stamp 2 Module, 1
  - Components, 307
  - HomeWork Board, 305
  - BASIC Stamp Editor
    - Identification Window, 19
    - Identify, 316, 317
    - Memory Map, 263
    - Software, 5
    - Trouble-Shooting, 315
  - BASIC Stamp HomeWork Board, 14
  - BASIC Stamp HomeWork Board
    - Components, 309
  - Battery, 44
  - Beat, 226
  - Bi-Color LED, 61
  - Binary Numbers, 20, 77, 177
  - Bit, 55, 175, 247
  - Board of Education, 13
    - Components, 308
    - Full Kit, 303
    - Revision label, 105
    - Servo Header, 106
  - Board of Education Rev B
    - Components, 310
  - Boe-Bot, 3
  - Breadboard. See Prototyping Area
  - Byte, 55, 247
- C -
- Cadmium Sulfide, 189
- Capacitor, 143–45, 149
  - Ceramic, 150
  - Polar – identifying terminals, 143

- Carriage return, 25
- Cathode, 40
- Charge, 319
- Circuit, 41
- Collector, 265
- COM Port, 15, 196
- Comment, 24
- Comment, 52
- Compiler directives, 21, 24
- Components
  - BASIC Stamp, 307
  - BASIC Stamp HomeWork Board, 309
  - Board of Education, 308
  - Board of Education Rev B, 310
- Conductor, 319
- Counting, 90
- CR, 25
- Current, 38, 44, 45, 319
  - Flow, 45
  - Milliamp, 324
- Cycle, 219
  - D -
- DATA, 230
- DEBUG
  - Command, 25
- DEBUG Formatters
  - CR, 25
  - DEC, 25
- Debug Terminal, 23
  - Receive Windowpane, 120, 121
  - Transmit Windowpane, 120, 121

- DEBUGIN, 120–23, 292
- DEBUGIN Formatters
  - STR, 292
- DEC, 25, 120
- Digital Potentiometer, 267–77
- Diode Forward Voltage, 323
- DIRH, 175–78
- Disconnect power, 32
- Dot Notes, 238
- Duration, 222, 229, 231, 233–38
  - E -
- Echo, 120, 121
- EEPROM, 197–203, 230, 263
- Electron, 45, 319
- Embedded system, 1
- Emitter, 265
- END, 25
- EXIT, 292
  - F -
- Fetch and Execute, 263
- Flat, 228
- FOR...NEXT, 53–56, 114
- FREQOUT, 221
- Frequency
  - Duration, 222
  - Mixing, 222, 225–27
  - Musical Notes, 227–30
  - G -
- GOSUB, 206
- Ground, 323
- Guarantee, 3

## - H -

Hertz, 219  
 HIGH, 49  
 Hz. See Hertz

## - I -

I/O Pins. See Input/Output Pins  
 IC. See Integrated Circuit  
 Identification Window, 19  
 Identify, 316, 317  
 IF...ELSEIF...ELSE, 85  
 IF...THEN...ELSE, 81  
 IN3, 77  
 IN7, 151  
 Indicator light, 37  
 Input/Output Pin  
   Input, 71  
 Input/Output Pins, 42  
   Default Direction, 177  
 Integer, 244  
 Integrated Circuit, 264  
   Pin Map, 268  
   Reference Notch, 268  
 Interference, 226

## - J -

Jumper, 106

## - K -

kB, 199  
 KB, 199  
 KCL, 323  
 Kilo, 324  
 Kilobyte, 199  
 Kirchhoff's Laws (Simplified)

Current, 323

Voltage, 323

KVL, 323

## - L -

Label, 206  
 Light Emitting Diode, 37–38, 40–41  
   Anode, 40  
   Bi-Color, 61  
   Cathode, 40  
   Circuit Calculations, 323  
   Schematic Symbol, 40  
   Terminals, 40  
 LOOKDOWN, 182  
 LOOKUP, 179–81  
 LOW, 49

## - M -

mA, 324  
 Math Operations, 242  
 Memory Map, 198, 263  
 Menu, 287  
 Metric units of measure, 324  
 Microcontroller, 1  
 Microsecond, 114  
 Milli, 324  
 Millisecond, 49, 114  
 Mixing, 222, 225–27  
 Multiply, 94  
 Music  
   Dot, 238  
   Flat, 228  
   FREQOUT and Notes, 229  
   Natural, 228

Note Duration, 229, 231, 233–38

Notes, 227

Piano Keyboard, 227

Rest, 233

Ringtone, 245

RTTTL Format, 327–28

RTTTL Format, 245, 251–57

Sharp, 228

Tempo, 233

- N -

Natural, 228

nc, 167

Nested Loop, 223–25

Neutral, 45

Nib, 55, 247

No-Connect, 167

Notes, 227

NPN, 265

- O -

Offset, 155

Ohm, 38, 319

Ohm's Law, 323

Omega, 38

ON...GOSUB, 287–96

Operator, 95

OUTH \r, 174

Overflow, 247

- P -

Parallax Standard Servo, 103

Caution, 104, 154

Parts, 104

Power Supply Caution, 314

Parallel

Bus, 173

Device, 173

Part Drawing

7-Segment Display, 165

Bi-Color LED, 61

LED, 40

Photoresistor, 190

Piezoelectric Speaker, 219

Potentiometer, 140

Pushbutton – Normally Open, 71

Resistor, 38

Transistor (NPN), 265

PAUSE, 49

PBASIC Commands

DEBUG, 25

DEBUGIN, 120–23, 292

DEC, 120

DO...LOOP, 49

END, 25

EXIT, 292

FOR...NEXT, 53–56, 114

FREQOUT, 221

GOSUB, 206

HIGH, 49

IF...ELSEIF...ELSE, 85

IF...THEN...ELSE, 81

- LOOKDOWN, 182
- LOOKUP, 179–81
- LOW, 49
- ON...GOSUB, 287
- PAUSE, 49
- PULSOUT, 113
- RANDOM, 95
- RCTIME, 151–52, 152
- READ, 202–3, 230
- RETURN, 206
- SELECT...CASE, 246–50
- TOGGLE, 272
- UNTIL, 93
- WRITE, 199–201, 230
- PBASIC Directives
  - DATA, 230–33
  - PBASIC, 24
  - PIN, 287
  - Stamp, 24
- PBASIC I/O Registers
  - DIRH, 175–78
  - IN3, 77
  - OUTH, 175–78
- PBASIC Operators
  - %, 177
  - \*, 94
  - \*\*, 244
  - \*/, 95, 244
- DCD, 243
- Order of Execution, 242
- Parenthesis, 242
- Photoresistor, 189–90
  - RC-Time Circuit, 190
- Piano Keyboard, 227
- Piezoelectric
  - Element, 220
  - Speaker, 219
- Pin Map, 166, 268
- Pitch, 219
- Polling, 90
- Potentiometer, 139–41
  - Digital, 267–77
  - Terminals, 140
- Prototyping Area
  - Input/Output Pins, 41
  - Power Terminals, 41
- Prototyping Areas
  - Socket, 41
- Pseudo Random, 96
- Pull-Up Resistor, 79
- Pulse, 113
- PULSOUT, 113
- Pushbutton, 71–72
  - Circuit, 75–77
  - Normally Open, 72
- R -
- Radio Control, 153
- RANDOM, 95

RC TIME, 151–52, 152  
READ, 199, 230  
Receive Windowpane, 120, 121  
Reference, 323  
Reference Notch, 268  
Resistance, 319  
Resistor, 38–40, 48  
    Color Codes, 39–40  
    Leads, 38  
    Pull-Down, 79  
    Pull-Up, 79  
    Tolerance, 39  
Rest, 233  
RETURN, 206  
Ringtone, 245  
Robot, 2–4  
RTTTL Format, 245, 251–57, 327–28

- S -

Scaling, 155  
Schematic  
    Dots Indicate Connections, 84  
    Drawing, 45  
Schematic Symbol  
    7-Segment Display, 166  
    Bi-Color LED, 61  
    LED, 40  
    Photoresistor, 190  
    Piezoelectric Speaker, 219  
    Potentiometer, 140  
    Pushbutton – Normally Open, 71

Resistor, 38  
    Transistor (NPN), 265  
Seed, 96  
SELECT...CASE, 246  
Sensor, 189  
Sensor Array, 281, 287  
Servo, 103–4  
    Cable, 103  
    Case, 103  
    Caution Statement, 104, 154  
    Connecting to BASIC Stamp, 105–13  
    Header, 106  
    Horn, 103  
    Jack, 103  
    Potentiometer Controlled, 153  
    Power Supply Caution, 314

Sharp, 228  
Speaker, piezoelectric, 220  
Stamp Plot Lite, 193–97  
Status indicator, 37  
STR, 292  
Subroutine, 206–13

    Label, 206

Subsystem Integration, 281  
Superposition, 226  
Syntax highlighting, 24

- T -

Tempo, 233  
Terminal  
    Piezoelectric Speaker, 220



TOGGLE, 272  
Token, 263  
Tolerance, 39  
Transistor, 264, 265–67

    2N3904, 265

    Base, 265

    Base Current, 266

    Collector, 265

    Emitter, 265

    NPN, 265

    Switching, 266

Transmit Windowpane, 120, 121

    - U -

UNTIL, 93

US232B, 301

USB to Serial Adapter, 6, 16, 301

    - V -

Variables, 53, 55, 247

Array, 292

Bit, 55, 247

Byte, 55, 247

Nib, 55, 247

Word, 55, 247

Vdd, 41, 46, 323

Vin, 41, 323

Volt, 319

Voltage, 45, 319

Vss, 41, 46, 323

    - W -

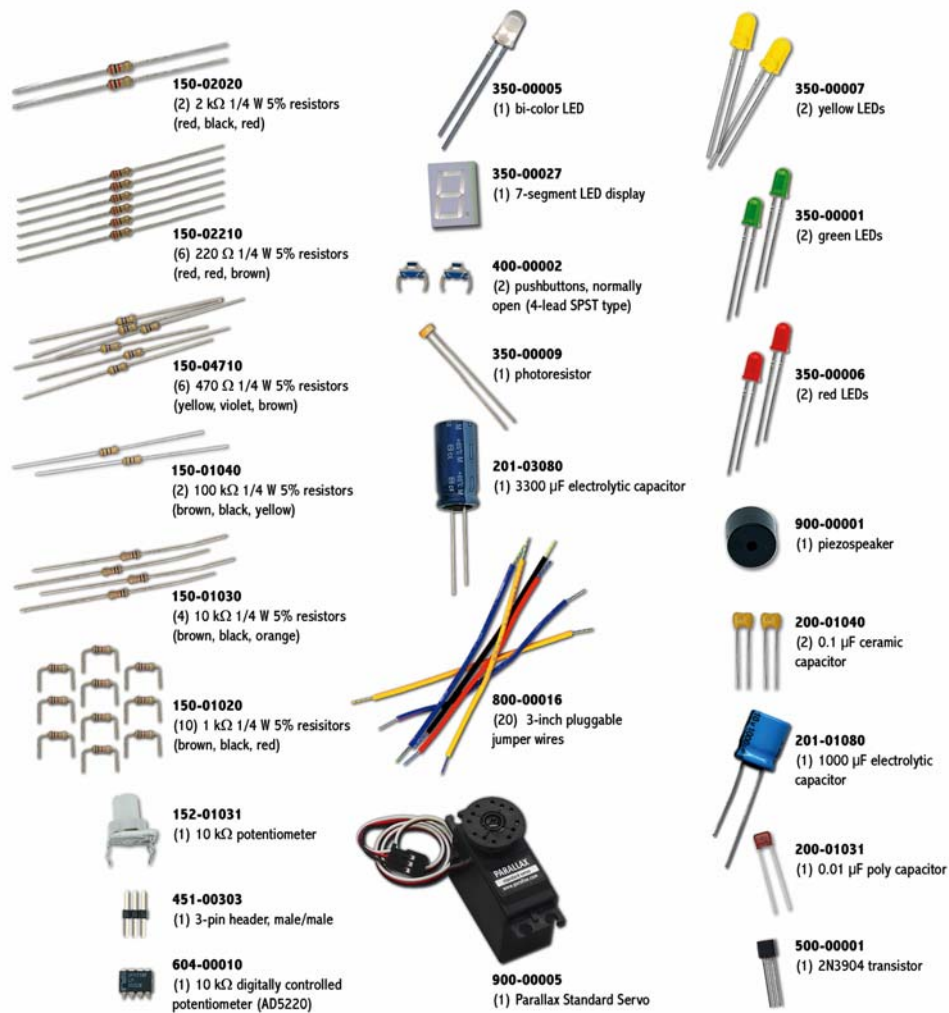
What's a Microcontroller

    Parts Kit, 303

    Text, 303

Word, 55, 247

WRITE, 199–201



Parts and quantities in the various What's a Microcontroller kits are subject to change without notice. Parts may differ from what is shown in this picture. Please contact [stampsinclass@parallax.com](mailto:stampsinclass@parallax.com) if you have any questions about your kit.

## Errata for *What's a Microcontroller?* v2.2 (#28123)

**Page 140:** In *What's a Microcontroller Parts Kits* purchased after January 10, 2007, the potentiometer may look different from the one shown in the text. The two styles are functionally the same, with the same terminals. Here are alternative drawings:

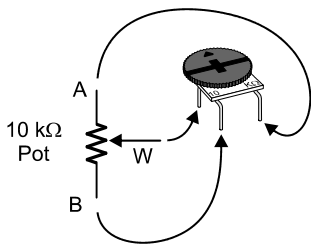
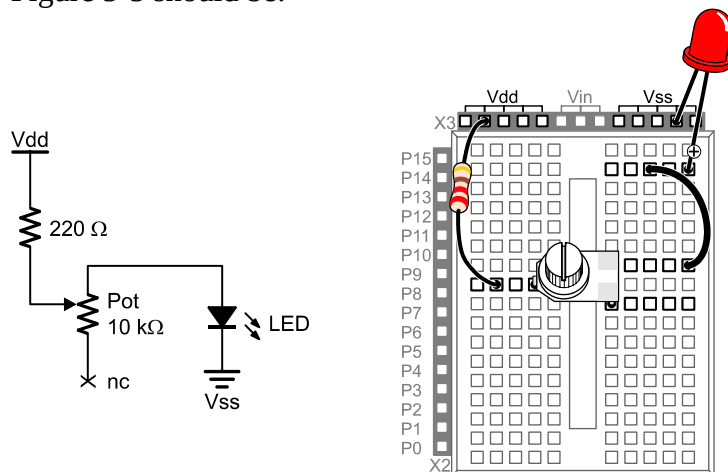


Figure 5-3



Figure 5-4 (left)

**Page 142:** In the text, the circuits represented by the schematic and wiring diagram in Figure 5-5 are not identical though they are functionally equivalent. To be consistent with each other, Figure 5-5 should be:



**Page 151:** Paragraph 1 should read:

The example program in Activity #2 measured the RC decay time by checking whether `IN7 = 0` every 100 ms, and it kept track of how many times it had to check. When `IN7` changed from 1 to 0, it indicated that the capacitor's voltage decayed below 1.4 V. The result when the program was done polling was that `IN7` equaled the number of tenths of a second it took for the capacitor's voltage to decay below 1.4 V.

This next example program uses a PBASIC command called **RCTIME** that makes the BASIC Stamp measure RC decay in terms of 2  $\mu$ s units. So, instead of tenths of a second, the result **RCTIME 7, 1, time** stores in the **time** variable is the number of two-millionths of a second that it takes for the capacitor's voltage to decay below 1.4 V. Since the **RCTIME** command has such fine measurement units, you can reduce the capacitor size from 3300  $\mu$ F to 0.1 or even 0.01  $\mu$ F, and still get time measurements that indicate to the resistor's value. Since the resistance between the potentiometer's A and W terminals changes as you turn the knob, the **RCTIME** measurement will give you a time measurement, which is a number that indicates the position of the potentiometer's knob.

**Page 183:** The first full paragraph should read (changed sentence is bolded):

Unless you tell it to make a different kind of comparison the **LOOKDOWN** command checks to see if a value is equal to an entry in the list. You can also check to see if a value is greater than, less than, or equal to, etc. **For example, to search for an entry that the value variable is less than or equal to, use the <= operator just before the first bracket that starts the list.**

In other words, in this context the operator returns the index of the first value in the list that makes the statement in the instruction true.

**Page 217:** For Solution 2, also add:

Index = 5

...to the Initialization section of your program.

**Page 228:** In the top row of Figure 8-3, the number 1217.5 should be 2217.5.

**Page 265:** In the second checklist instruction, P3 should be P14.

**Page 277:** The program below was printed with several lines omitted. The complete correct listing is included below, with the added lines in **bold**.

```
' -----[Title]-----
' What's a Microcontroller - Ch9Prj01_PhotoControlledDigitalPot.bs2
' Corrected version of program on page 277 of v2.2 text.
' Update digital pot's tap based on photoresistor reading.
' Based on TerminalControlledDigitalPot.bs2

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

' -----[Declarations and Initialization]-----

PhotoPin CON 2 ' Photoresistor on pin P2
UdPin CON 5 ' Set values of I/O pins
ClkPin CON 6 ' connected to CLK and U/D.
DelayPulses CON 10 ' Delay to observe LED fade.
DelayReader CON 2000

counter VAR Byte ' Counter for FOR...NEXT.
oldTapSetting VAR Byte ' Previous tap setting.
newTapSetting VAR Byte ' New tap setting.
lightReading VAR Word ' reading from photoresistor
```

```

oldTapSetting = 0 ' Initialize new and old
newTapSetting = 0 ' tap settings to zero.

LOW Udpin ' Set U/D pin for Down.
FOR counter = 0 TO 128 ' Set tap to lowest position.
 PULSOUT 6,5
 PAUSE 1
NEXT

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

DO:
 GOSUB Read_Photoristor
 lightReading = lightReading MAX 127 ' Constrain to max of 127
 newTapSetting = lightReading
 GOSUB Set_Ud_Pin ' Set U/D pin for up/down.
 IF newTapSetting <> oldTapSetting THEN ' No change if light level steady
 GOSUB Pulse_Clk_pin ' Deliver pulses.
 ENDIF
LOOP

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

Set_Ud_Pin: ' Examine new and old tap values
 ' to decide value of U/D pin.

 IF newTapSetting > oldTapSetting THEN
 HIGH Udpin
 oldTapSetting = oldTapSetting + 1
 ELSEIF newTapSetting < oldTapSetting THEN
 LOW Udpin
 oldTapSetting = oldTapSetting - 1
 ENDIF
 RETURN

Pulse_Clk_pin: ' Deliver pulses from old to
 FOR counter = oldTapSetting TO newTapSetting ' new values.
 PULSOUT ClkPin, 1
 PAUSE DelayPulses
 NEXT
 oldTapSetting = newTapSetting ' Keep track of new and old
 RETURN ' tapSetting values.

Read_Photoristor:
 HIGH PhotoPin
 PAUSE 100
 RCTIME PhotoPin, 1, lightReading
 RETURN

```