

What's a Microcontroller?

Student Guide

VERSION 3.0



WARRANTY

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

14-DAY MONEY BACK GUARANTEE

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

COPYRIGHTS AND TRADEMARKS

This documentation is Copyright 2003-2009 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, in whole or in part, is permitted subject to the following conditions: the material is to be used solely in conjunction with Parallax products, and the user may recover from the student only the cost of duplication. Check with Parallax for approval prior to duplicating any of our documentation in part or whole for any other use.

BASIC Stamp, Board of Education, Boe-Bot, Stamps in Class, and SumoBot are registered trademarks of Parallax Inc. HomeWork Board, PING))), Parallax, the Parallax logo, Propeller, and Spin are trademarks of Parallax Inc. If you decide to use any of these words on your electronic or printed material, you must state that "(trademark) is a (registered) trademark of Parallax Inc." upon the first use of the trademark name. Other brand and product names herein are trademarks or registered trademarks of their respective holders.

ISBN 9781928982524

3.0.0-09.12.09-HKTP

DISCLAIMER OF LIABILITY

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

ERRATA

While great effort is made to assure the accuracy of our texts, errors may still exist. Occasionally an errata sheet with a list of known errors and corrections for a given text will be posted on the related product page at www.parallax.com. If you find an error, please send an email to editor@parallax.com.

Table of Contents

Preface	7
About Version 3.0	7
Audience	8
Support Forums	8
Resources for Educators	9
Foreign Translations	10
About the Author	10
Special Contributors	10
Chapter 1 : Getting Started	11
How Many Microcontrollers Did You Use Today?	11
The BASIC Stamp 2 – Your New Microcontroller Module	11
Amazing Inventions with BASIC Stamp Microcontrollers	12
Hardware and Software	15
Activity #1 : Getting the Software	15
Activity #2 : Using the Help File for Hardware Setup	21
Summary	23
Chapter 2 : Lights On – Lights Off	27
Indicator Lights	27
Making a Light-Emitting Diode (LED) Emit Light	27
Activity #1 : Building and Testing the LED Circuit	28
Activity #2 : On/Off Control with the BASIC Stamp	37
Activity #3 : Counting and Repeating	43
Activity #4 : Building and Testing a Second LED Circuit	46
Activity #5 : Using Current Direction to Control a Bicolor LED	50
Summary	57
Chapter 3 : Digital Input – Pushbuttons	61
Found on Calculators, Handheld Games, and Appliances	61
Receiving vs. Sending High and Low Signals	61
Activity #1 : Testing a Pushbutton with an LED Circuit	61
Activity #2 : Reading a Pushbutton with the BASIC Stamp	65
Activity #3 : Pushbutton Control of an LED Circuit	70
Activity #4 : Two Pushbuttons Controlling Two LED Circuits	73
Activity #5 : Reaction Timer Test	79
Summary	87
Chapter 4 : Controlling Motion	93
Microcontrolled Motion	93
On/Off Signals and Motor Motion	93
Introducing the Servo	93

Activity #1 : Connecting and Testing the Servo.....	95
Activity #2 : Servo Control Test Program.....	102
Activity #3 : Control Servo Hold Time	112
Activity #4 : Controlling Position with your Computer	118
Activity #5 : Converting Position to Motion	126
Activity #6 : Pushbutton-Controlled Servo	129
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	150
Activity #4 : Controlling a Servo with a Potentiometer	156
Summary	164
Chapter 6 : Digital Display.....	169
The Everyday Digital Display.....	169
What's a 7-Segment Display?	169
Activity #1 : Building and Testing the 7-Segment LED Display	171
Activity #2 : Controlling the 7-Segment LED Display.....	175
Activity #3 : Displaying Digits.....	178
Activity #4 : Displaying the Position of a Dial.....	185
Summary	191
Chapter 7 : Measuring Light.....	195
Devices that Contain Light Sensors	195
Introducing the Phototransistor.....	198
Activity #1 : Building and Testing the Light Meter.....	199
Activity #2 : Tracking Light Events.....	202
Activity #3 : Graphing Light Measurements (Optional)	211
Activity #4 : Simple Light Meter	214
Activity #5 : On/Off Phototransistor Output.....	225
Activity #6 : For Fun—Measure Outdoor Light with an LED	235
Summary	239
Chapter 8 : Frequency and Sound	245
Your Day and Electronic Beeps.....	245
Microcontrollers, Speakers, and On/Off Signals.....	245
Activity #1 : Building and Testing the Speaker	246
Activity #2 : Action Sounds	248
Activity #3 : Musical Notes and Simple Songs.....	253
Activity #4 : Microcontroller Music	258

Activity #5 : Ringtones with RTTTL.....	271
Summary	283
Chapter 9 : Electronic Building Blocks	287
Those Little Black Chips	287
Expand your Projects with Peripheral Integrated Circuits.....	288
Activity #1 : Control Current Flow with a Transistor	289
Activity #2 : Introducing the Digital Potentiometer	292
Summary	302
Chapter 10 : Prototyping Your Own Inventions	307
Apply what You Know to Other Parts and Components	307
Prototyping a Micro Security System	308
Activity #1 : From Idea to Proof of Concept	308
Activity #2 : Build and Test Each Circuit Individually	311
Activity #3 : Organize Coding Tasks Into Small Pieces	313
Activity #4 : Document Your Code!	317
Activity #5 : Give Your App Amazing New Functionality.....	319
Activity #6 : How to Jump Over Design Hurdles	320
Activity #7 : What's Next?	327
Summary	331
Complete Kit Options	334
Bonus Activity: Ohm's Law, Voltage, and Current	336
Index	345

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 explore 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, while using common inexpensive parts.

What’s a Microcontroller? is the gateway text in to the Stamps in Class program. To see the full series, which includes such titles as *Robotics with the Boe-Bot*, *Smart Sensors and Applications*, *Process Control*, and more, visit www.parallax.com/Education.

ABOUT VERSION 3.0

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

- Replacement of the cadmium sulfide photoresistor with an RoHS-compliant light sensor of a type that will be more common in product design going forward. This required rewrites of Chapters 7 and 10, and adjustments in other chapters.
- Improved activities and illustrations of servo control in Chapter 4.
- Moving the “Setup and Testing” portion of Chapter 1 and the Hardware and Troubleshooting appendices to the Help file. This was done to support both serial and USB hardware connections, and other programming connections as our products and technologies continue to expand. This also allows for the dynamic maintenance of the Hardware and Troubleshooting material.
- Removal of references to the Parallax CD, which has been removed from our kits, reducing waste and ensuring that customers download the most recent BASIC Stamp Editor software and USB drivers available for their operating systems.

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

AUDIENCE

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

SUPPORT FORUMS

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

- Propeller Chip: for all discussions related to the multicore Propeller microcontroller and development tools product line.
- BASIC Stamp: Project ideas, support, and related topics for all of the Parallax BASIC Stamp models.
- SX Microcontrollers: Technical assistance for all SX chip products, including the SX/B Compiler, and SX-Key Tool.
- Sensors: Discussion relating to Parallax's wide array of sensors, and interfacing sensors with Parallax microcontrollers.
- Stamps in Class: Students, teachers, and customers discuss Parallax's education materials and school projects here.
- Robotics: For all Parallax robots and custom robots built with Parallax processors and sensors.
- The Sandbox: Topics related to the use of Parallax products but not specific to the other forums.
- Completed Projects: Post your completed projects here, made from Parallax products.
- HYDRA System and Propeller Game Development: Discussion and technical assistance for the HYDRA Game Development Kit and related Propeller microcontroller programming.

RESOURCES FOR EDUCATORS

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

Stamps in Class “Mini Projects”

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

Educators Courses

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

Parallax Educator’s Forum

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

Supplemental Educational Materials

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

Copyright Permissions for Educational Use

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

FOREIGN TRANSLATIONS

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

ABOUT THE AUTHOR

Andy Lindsay joined Parallax Inc. in 1999, and has since authored eight books and numerous articles and product documents for the company. The last three versions of *What's a Microcontroller?* were designed and updated based on observations and educator feedback that Andy collected while traveling the nation and abroad teaching Parallax Educator Courses and events. Andy studied Electrical and Electronic Engineering at California State University, Sacramento, and is a contributing author to several papers that address the topic of microcontrollers in pre-engineering curricula. When he's not writing educational material, Andy does product and application engineering for Parallax.

SPECIAL CONTRIBUTORS

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

Many people contributed to the development of *What's a Microcontroller?* and assisted with previous editions, to whom we are still grateful. Parallax wishes to again thank Robert Ang for his thorough review and detailed input, and the late 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 (*Process 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.

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 devices. 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
Everyday 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 interacting with microcontrollers. That's just the beginning. Here are a few more examples: turning on the television with a handheld remote, playing a handheld game, and using a calculator. All those devices have microcontrollers inside them that interact with you.

THE BASIC STAMP 2 – YOUR NEW MICROCONTROLLER MODULE

Parallax Inc.'s BASIC Stamp[®] 2 module shown in Figure 1-2 has a microcontroller built onto it; it is the largest black chip. 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 gadgets. 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
BASIC Stamp 2
Microcontroller
Module



In this text, “BASIC Stamp” refers to the BASIC Stamp® 2 microcontroller module. Designed and manufactured by Parallax Incorporated, this module’s name is commonly abbreviated BS2, and it’s the first in the series of modules shown in Figure 1-3. Each of the other modules is slightly different, featuring higher speed, more memory, additional functionality, or some combination of these extra features. To learn more, follow the “Compare BASIC Stamp Modules” link at www.parallax.com/basicstamp.

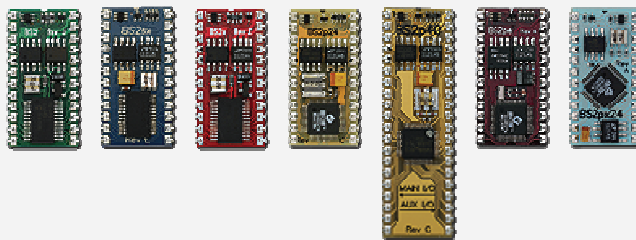


Figure 1-3
BASIC Stamp 2
Models, left to
right: BS2, BS2e,
BS2sx, BS2p24,
BS2p40, BS2pe,
BS2px

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® robot. 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 text. The one on the right is called an underwater ROV (remotely operated vehicle) and it was constructed and tested at a MATE (Marine Advanced Technology Education)

Summer Teachers Institute. Operators view a TV displaying what the ROV sees through a video camera and control it with a combination of hand controls and a laptop. Its BASIC Stamp measures depth and temperature, controls the vertical thrust motor, and exchanges information with the laptop. MATE coordinates regional and international ROV competitions for students at levels ranging from middle school to university.

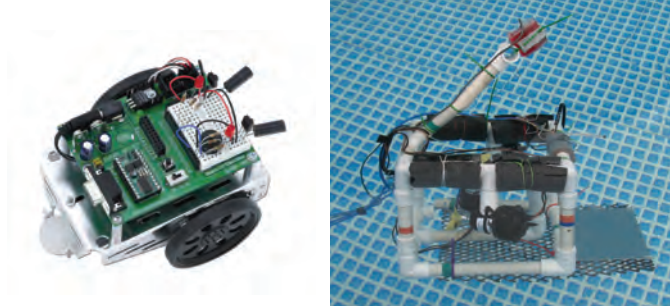


Figure 1-4

Educational Robots

*Boe-Bot robot (left)
ROV at MATE Summer
Teachers Institute (right,
www.marinetech.org)*

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 of Figure 1-5 was developed by a professor at Nanyang Technical University, Singapore. It has more than 50 BASIC Stamp modules on board, and they all communicate with each other in an elaborate network that help 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

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 can 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
BASIC Stamp 2
Microcontroller Module

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 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
Environmental 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 device 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 is take it out of the box, plug it in, install and test some software, and maybe even write some software of their own using a programming language. If this is your first time using a BASIC Stamp module, you will be doing all these same activities. If you are in a class, your hardware may already be all set up for you. If this is the case, your teacher may have other instructions. If not, this chapter will take you through all the steps of getting your new BASIC Stamp microcontroller up and running.

ACTIVITY #1: GETTING THE SOFTWARE

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

Computer System Requirements

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

- Microsoft Windows 2000 or newer operating system
- An available serial or USB port
- Internet access and an Internet browser program

Downloading the Software from the Internet

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

- ✓ Using a web browser, go to www.parallax.com/basicstampsoftware (Figure 1-8).



Figure 1-8

The BASIC Stamp Editor software page at www.parallax.com/basicstampsoftware

This is the place to download the latest version of the software.

- ✓ Click on the [Click Here to Download](#) button to download the latest version of the BASIC Stamp Windows Editor software (Figure 1-9).



Figure 1-9

The Download button on the BASIC Stamp Editor Software page.

Click on the button to start the download.

- ✓ A File Download window will open, asking you if you want to run or to save this file (Figure 1-10). Click on the Save button.

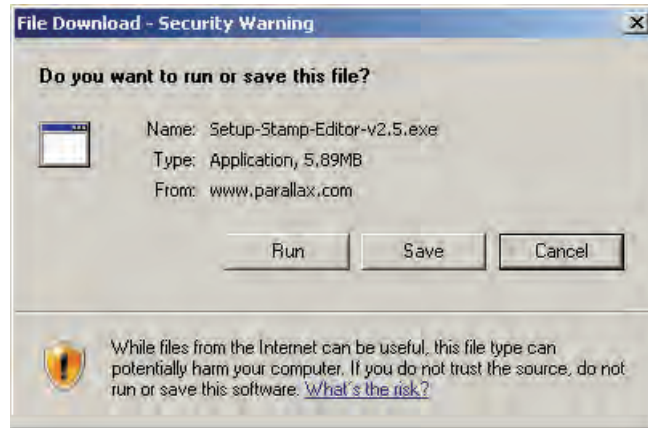


Figure 1-10
File Download Window

Click Save, then save the file to your computer.

- ✓ Use the Save in field to choose a place on your computer to save the installer file, then click the Save button (Figure 1-11).

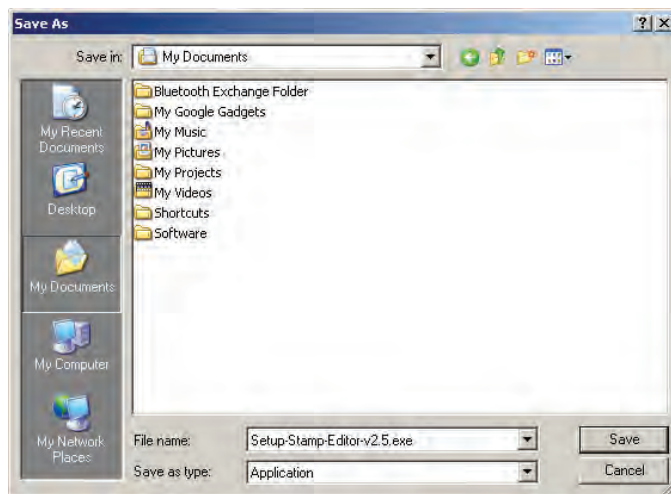


Figure 1-11
Save As Window

Choose a place to save the software installer on your computer, then click Save.

- ✓ When you see “Download Complete,” click the Run button (Figure 1-12.)
- ✓ Follow the prompts that appear. You may see messages from your operating system asking you to verify that you wish to continue with installation. Always agree that you want to continue.

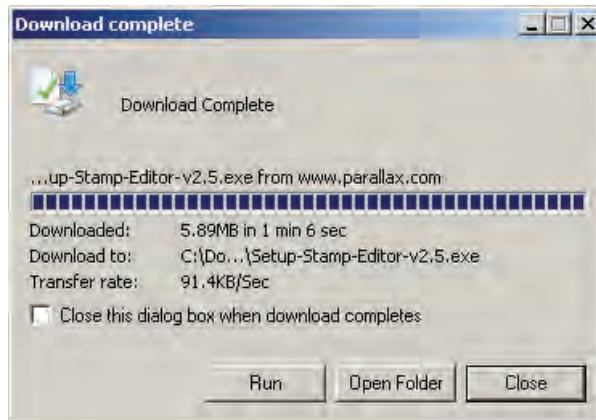


Figure 1-12
Download Complete
Message

Click Run.

*If prompted, always
confirm you want to
continue.*

- ✓ The BASIC Stamp Editor Installer window will open (Figure 1-13). Click Next and follow the prompts, accepting all defaults.



Figure 1-13
BASIC Stamp Editor
Installer Window

Click Next.

- ✓ **IMPORTANT:** When the “Install USB Driver” message appears (Figure 1-14), leave the checkmark in place for the Automatically install/update driver (recommended) box, and then click Next.

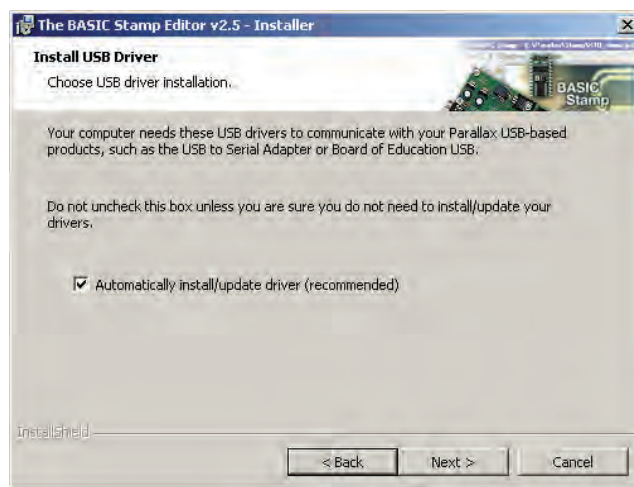


Figure 1-14
Install USB Driver
Message

*Leave the box
checked, and click
Next.*

- ✓ When the “Ready to Install the Program” message appears (Figure 1-15), click the Install button. A progress bar may appear, and this could take a few minutes.

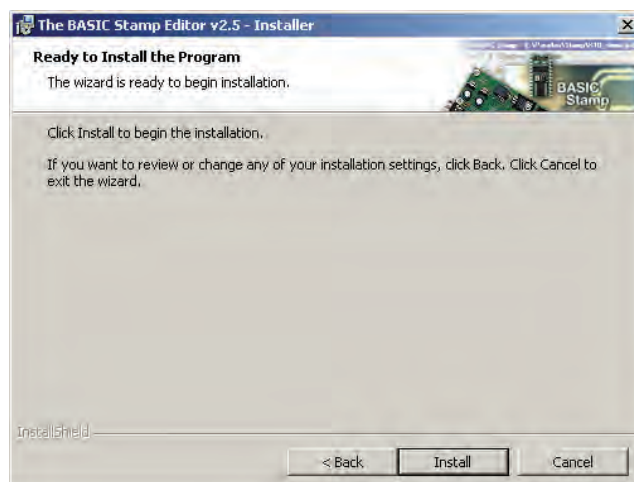


Figure 1-15
Ready to Install the
Program

*Click Install to
continue.*

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



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

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

- ✓ When the window tells you that installation has been successfully completed, click Finish (Figure 1-16).



Figure 1-16
BASIC Stamp
Editor Installation
Completed

Click Finish.

ACTIVITY #2: USING THE HELP FILE FOR HARDWARE SETUP

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

Running the BASIC Stamp Editor for the first time

- ✓ If you see the BASIC Stamp Editor icon on your computer desktop, double-click it (Figure 1-17).
- ✓ Or, click on your computer's Start menu, then choose All Programs ▶ Parallax Inc ▶ BASIC Stamp Editor 2.5 ▶ BASIC Stamp Editor 2.5.



Figure 1-17
BASIC Stamp Editor
Desktop Icon

*Double-click to launch
the program.*

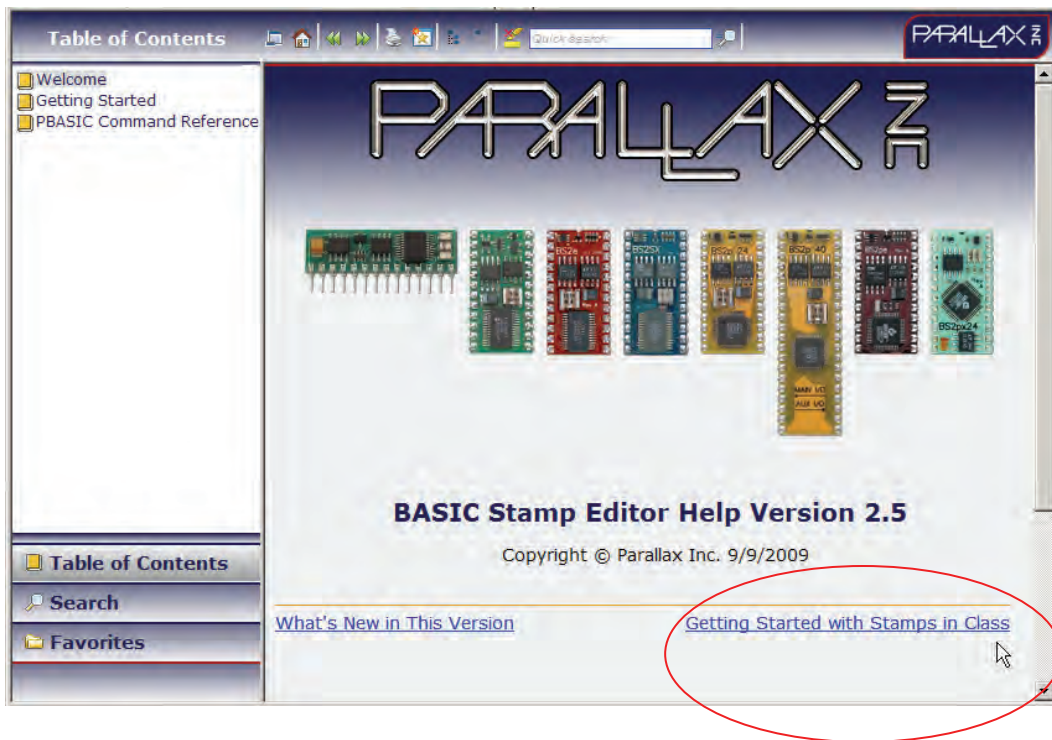
- ✓ On the BASIC Stamp Editor's toolbar, click Help on the toolbar (Figure 1-18) and then select BASIC Stamp Help... from the drop-down menu.



Figure 1-18
Opening the Help Menu

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

Figure 1-19: BASIC Stamp Editor Help



- ✓ Click on the Getting Started with Stamps in Class link on the bottom of the Welcome page, as shown in the lower right corner of Figure 1-19.

Following the Directions in the Help File

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

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

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

**What do I do if I get stuck?**

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

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

SUMMARY

This chapter guided you through the following:

- An introduction to 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
- How to use the BASIC Stamp Editor's Help and the BASIC Stamp Manual
- An introduction to the BASIC Stamp module, Board of Education, and HomeWork Board
- How to set up your BASIC Stamp hardware
- How to test your software and hardware

- How to write and run a PBASIC program
- Using the **DEBUG** and **END** commands
- Using the **CR** control character and **DEC** formatter
- A brief introduction to ASCII code
- How to disconnect the power to your Board of Education or HomeWork Board when you're done

Questions

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

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

Solutions

- Q1. A microcontroller is a kind of miniature computer found in electronic products.
- Q2. The BASIC Stamp module contains a microcontroller chip.
- 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 using the **CR** control character and a comma.

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

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

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

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

END
```

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

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

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

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

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

END
```

The output from the Debug Terminal is:

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

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

Chapter 2: 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, DVD players, disk drives, printers, and alarm system control panels.

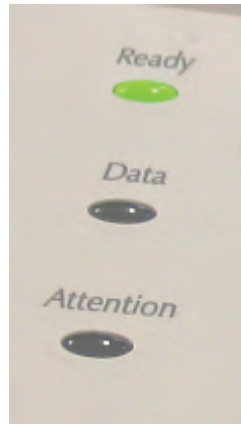


Figure 2-1
Indicator Lights

Indicator lights are common on many everyday devices.

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 bicolor (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 . Later in this book you will see the symbol $k\Omega$, meaning kilo-ohm, or one thousand ohms. The resistor you will be working with in this activity is the $470\ \Omega$ 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 $470\ \Omega$ 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, and where to place it when you build the circuit.

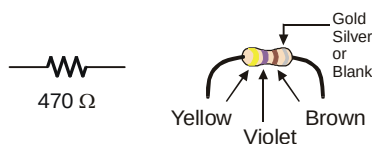


Figure 2-2
470 Ω Resistor Part Drawing

Schematic symbol (left) and Part Drawing (right)

2

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

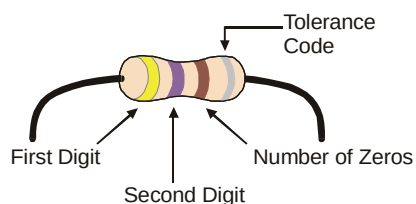


Figure 2-3
Resistor Color
Codes

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 = 470 Ω .

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 it is a one-way current valve, make sure to connect it the right way in your circuit 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.

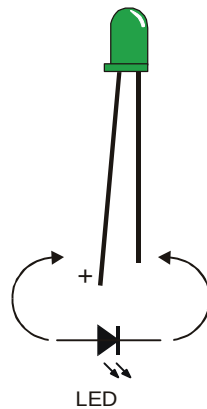


Figure 2-4
LED Part Drawing and Schematic
Symbol

*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.

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 (+5 V), Vin (the unregulated voltage straight from your battery or power supply), and Vss (0 V, also called *ground*). 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.

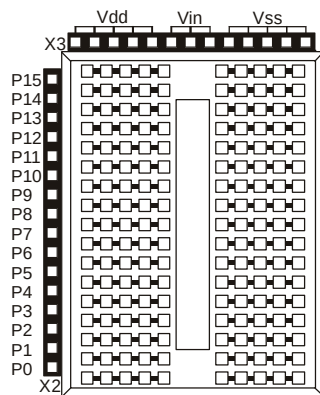


Figure 2-5
Prototyping Area

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.

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. This breadboard has 17 rows of sockets. In each row, there are two five-socket groups separated by a trench in the middle. All the sockets in a 5-socket group are connected together. So, if you plug two wires into the same 5-socket group, they will make electrical contact. Two wires in the same row but on opposite sides of the center trench will not be connected. Many devices are designed to be plugged in over this trench, such as the pushbutton we will use in Chapter 3.



More about breadboarding: To learn about the history of breadboards, how modern breadboards are constructed, and how to use them, see the video resources at www.parallax.com/go/WAM.

Figure 2-6 shows a circuit schematic, and a picture of how that circuit will look when it is built on the prototyping area. Each 5-socket group 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 group. Note that one lead of the resistor is plugged into Vdd (+5 V) 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 (0 V, 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 5-socket group 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.

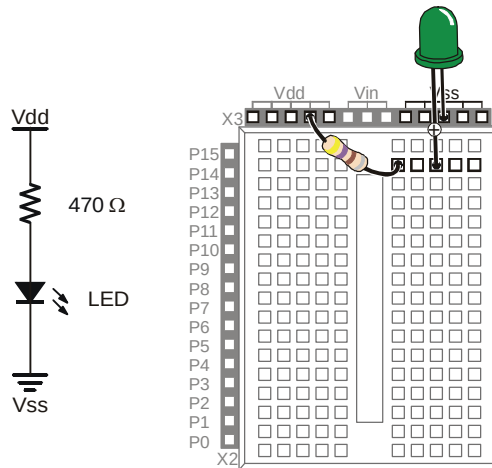


Figure 2-6

LED On, wired directly to power

Schematic (left) and Wiring Diagram (right).

Note that one resistor lead and the green LED's anode lead are plugged into the same 5-socket group. This electrically connects the two components.

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 the LED's 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? Try these free online resources:

Visit the Stamps In Class moderated forums: If you don't have an instructor or friend who can help, you can always check with the Stamps in Class forum at <http://forums.parallax.com>. If you don't get your questions answered there, you can contact 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.

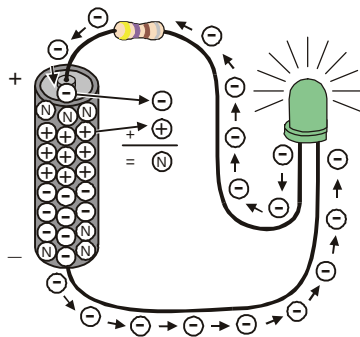


Figure 2-7
LED On Circuit Electron Flow

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 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 electrical 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.

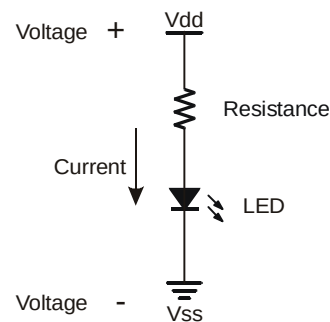


Figure 2-8
LED-On Circuit Schematic Showing
Conventional Voltage and Current
Flow

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 B: More about Electricity 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.



Why does the LED not glow? Since both ends of the circuit are connected to the same voltage (Vss), there isn't any electrical pressure across the circuit. So, no current flows through the circuit, and the LED stays off.

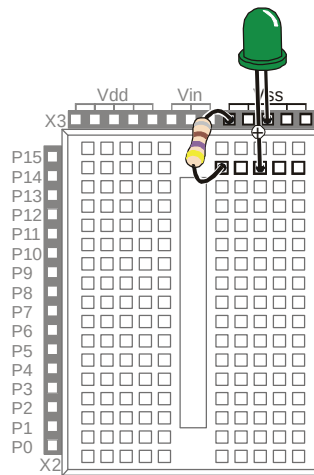
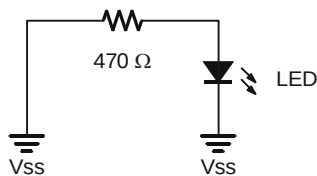


Figure 2-9
LED Off Circuit

*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.

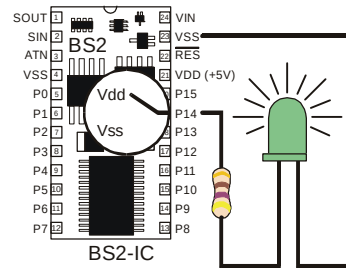
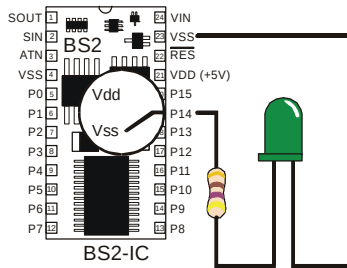


Figure 2-10
BASIC Stamp
Switching

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 to the development board 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 connected to 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.

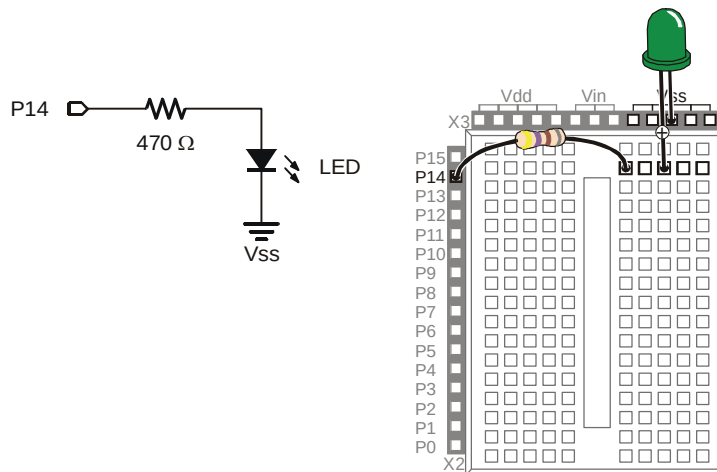


Figure 2-11
BASIC Stamp
Controlled LED Circuit

*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 P14 is blinking!"

DO

    HIGH 14
    PAUSE 500
    LOW 14
    PAUSE 500

LOOP
```

How LedOnOff.bs2 Works

The command **DEBUG "The LED connected to P14 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*. Arguments give PBASIC commands the information that they need to execute. 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.

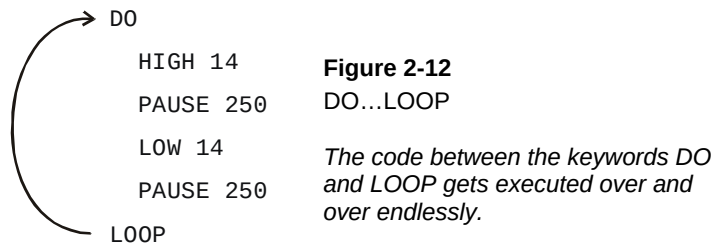


What's a Millisecond? A *millisecond* is 1/1000 of a second. It is abbreviated as ms. It takes 1000 ms to equal one second.

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. Code that repeats a set of commands indefinitely is called an *infinite loop*.



A Diagnostic Test for your Computer

Although it's not common, there are some computer systems, such as certain laptops and docking stations, that will halt the PBASIC program after the first time through a **DO . . . LOOP**. These computers have a non-standard serial port design. By placing a **DEBUG** command in 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** command.
- ✓ 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 will always need to add a single **DEBUG** command, such as:

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


...right after the compiler directives. It 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, but in that case be sure to add a short **DEBUG** command when you start writing your own programs.

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 and save a copy of it as LedOnOffYourTurn.bs2.
- ✓ Change both of the **PAUSE** commands' *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 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 38.

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. This is sometimes called a *counted* or *finite loop*.

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 chooses 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**. You can see the full Reserved Word List in the BASIC Stamp Manual.
2. The name cannot contain a space.
3. Even though the name can contain letters, numbers, or underscores, it must begin with a letter.
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:



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

A **DEBUG** instruction can include *formatters* that determine how information should be displayed in the Debug Terminal. Placing 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
```

```
  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

- ✓ In the program LedOnOffTenTimes.bs2, replace the statement:

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

- ✓ 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 Required

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.

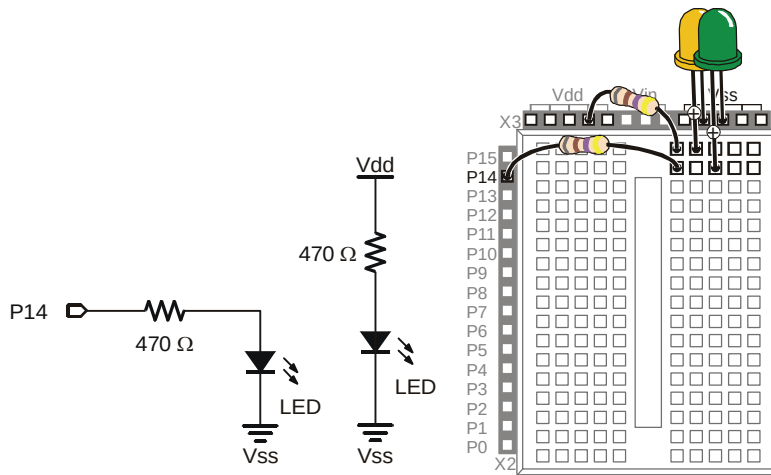


Figure 2-13
Manual Test
Circuit for
Second LED

- ✓ 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.

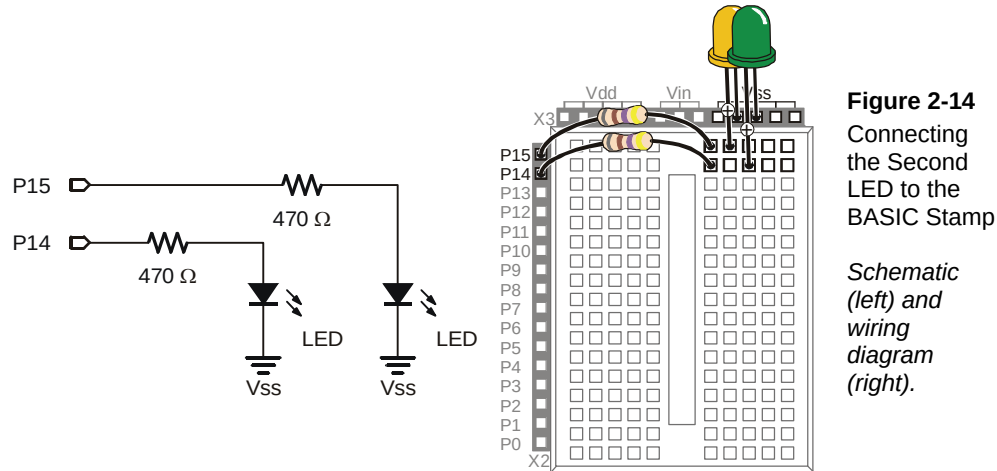


Figure 2-14
Connecting
the Second
LED to the
BASIC Stamp

*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 BICOLOR 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 *bicolor* 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

Bicolor LED in a Security Device

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

Introducing the Bicolor LED

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

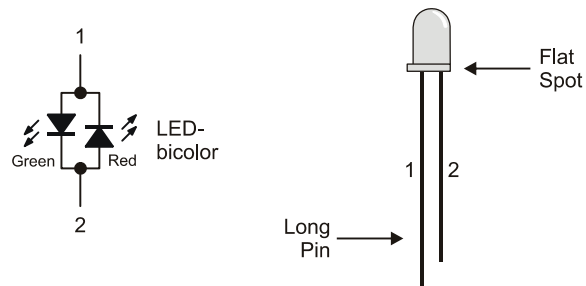


Figure 2-16

Bicolor LED

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

The bicolor 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 green. By disconnecting the LED and plugging it back in reversed, the LED will then glow red. As with the other LEDs, if you connect both terminals of the circuit to Vss, the LED will not emit light.

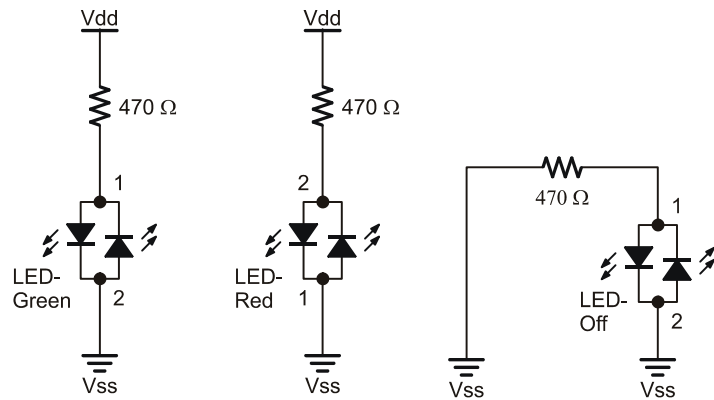


Figure 2-17
Bicolor LED and
Applied Voltage

*Green (left), red
(center) and no
light (right)*

Bicolor LED Circuit Parts

- (1) LED – bicolor
- (1) Resistor – $470\ \Omega$ (yellow-violet-brown)
- (1) Jumper wire

Building and Testing the Bicolor LED Circuit

Figure 2-18 shows the manual test for the bicolor 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 bicolor LED is emitting green light.
- ✓ Disconnect power again.
- ✓ Modify your circuit so that it matches the right side of Figure 2-18.
- ✓ Reconnect power.
- ✓ Verify that the bicolor LED is now emitting red light.
- ✓ Disconnect power.



What if my bicolor LED's colors are reversed? Bicolor LEDs are manufactured like the one in Figure 2-16 as well as with the colors reversed. If your bicolor LED glows red when it's connected in the circuit that should make it glow green and vice-versa, your LED's colors are reversed. If that's the case, always plug pin 1 in where the diagrams show pin 2, and pin 2 where the diagrams show pin 1.

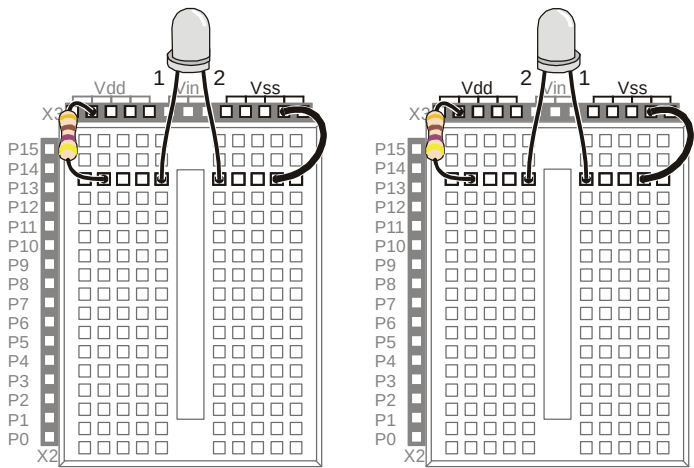


Figure 2-18
Manual bicolor LED Test
*Bicolor LED green (left)
and red (right).*

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

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

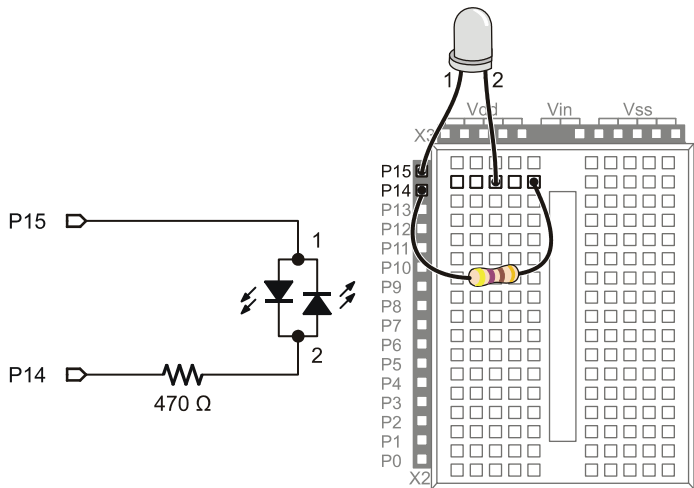


Figure 2-19
Bicolor LED Connected
to BASIC Stamp
*Schematic (left) and
wiring diagram (right).*

BASIC Stamp Bicolor LED Control

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

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 green LED shuts off and does not allow current through. Meanwhile, the red LED turns on, and current passes through the circuit in the opposite direction.

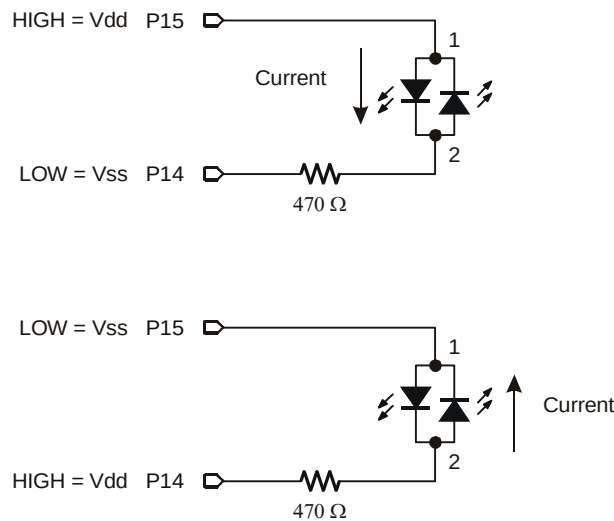


Figure 2-20
BASIC Stamp bicolor
LED Test

*Current through green
LED (above) and red
LED (below).*

Figure 2-20 also shows the key to programming the BASIC Stamp to make the bicolor LED glow two different colors. The upper schematic shows how to make the bicolor LED green using **HIGH 15** and **LOW 14**. The lower schematic shows how to make the bicolor LED glow red 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 bicolor LED will also turn off if you send high signals to both P14 and P15. Why? Because the electrical pressure (voltage) 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 and run TestBiColorLed.bs2 code in the BASIC Stamp Editor.
- ✓ Verify that the LED cycles through the red, green, and off states.

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

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

PAUSE 1000
DEBUG "Program Running!", CR

DO

  DEBUG "Green..."
  HIGH 15
  LOW 14
  PAUSE 1500

  DEBUG "Red..."
  LOW 15
  HIGH 14
  PAUSE 1500

  DEBUG "Off...", CR
  LOW 15
  LOW 14
  PAUSE 1500

LOOP
```

Your Turn – Lights Display

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

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

```
counter VAR BYTE
```

- ✓ Replace the test code in the **DO . . . LOOP** with this **FOR . . . NEXT** 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 for indicating 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. The amount of time is determined by the number used in the **PAUSE** command's *Duration* argument.

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. Counted loops can be made with **FOR . . . NEXT**, a variable to keep track of how many repetitions the loop has made, and numbers to specify where to start and stop counting.

Current direction and voltage polarity were introduced using a bicolor 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 38 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.

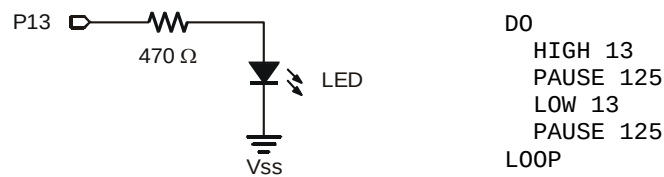
Project

1. Make a 10-second countdown using one yellow LED and one bicolor LED. Make the bicolor LED start out red for 3 seconds. After 3 seconds, change the bicolor LED to green. When the bicolor 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 bicolor 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 5-socket group. You can connect 4 wires by plugging all 4 wires into the same 5-socket group.
- 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**.

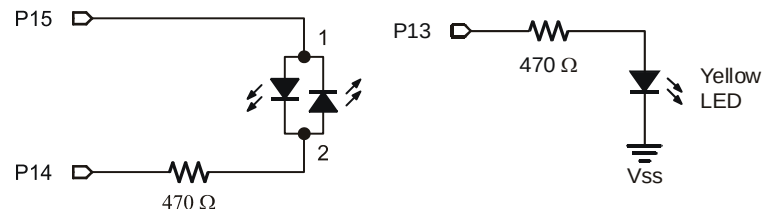


- E2. The **counter** variable has to be changed to **Word** size, and the **FOR** statement has to be modified to count from 1 to 5000.

```

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

- P1. The bicolor LED schematic, on the left, is unchanged from Figure 2-19 on page 53. The yellow LED schematic is based on Figure 2-11 on page 38. For this project P14 was changed to P13, and a yellow LED was used instead of green. NOTE: When the BASIC Stamp runs out of commands, it goes into a low power mode that causes the bicolor LEDs to flicker briefly every 2.3 seconds. The same applies after the program executes an **END** command. There's another command called **STOP** that you can add to the end of the program to make it hold any high/low signals without going into low power mode, which in turn prevents the flicker.



```
' 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
LOW 15
HIGH 14
PAUSE 3000
' Bicolor LED Red

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

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

' Red stays on
LOW 15
HIGH 14
' Bi Color LED Red
```

Chapter 3: Digital Input – Pushbuttons

3

FOUND ON CALCULATORS, HANDHELD 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, TV remote, handheld game, and cell phone. 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.

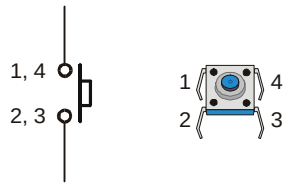


Figure 3-1

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 a *closed circuit*, and current can flow through the pushbutton.

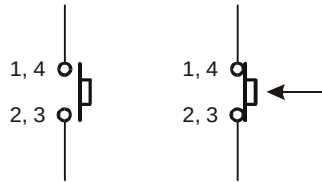


Figure 3-2

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.

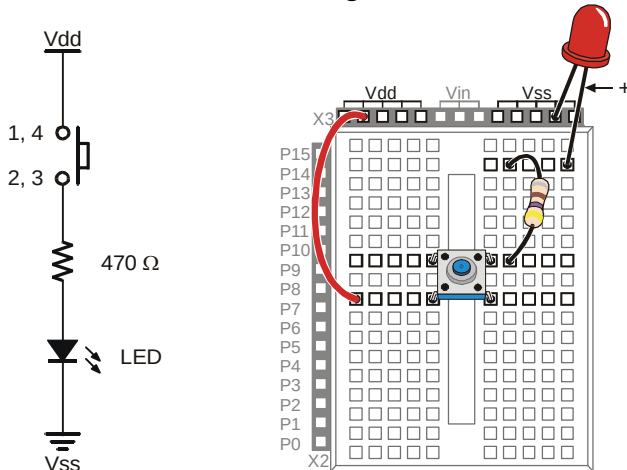


Figure 3-3
Pushbutton Test Circuit

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 reconnect power, it may mean that there is a short circuit from Vdd to Vss or from Vin to Vss. If this happens, disconnect power immediately and find and correct the mistake in your circuit.

The LED built into the HomeWork Board is different. It may either be labeled “Power” or “Running” and it only glows while a program is running. If a program ends, either because it executes an **END** command or because it runs out of commands to execute, the LED will 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, as shown on the right side of the figure, you close the connection between the terminals with conductive metal. This makes a pathway for electrons to flow through the circuit and so the LED emits light as a result.

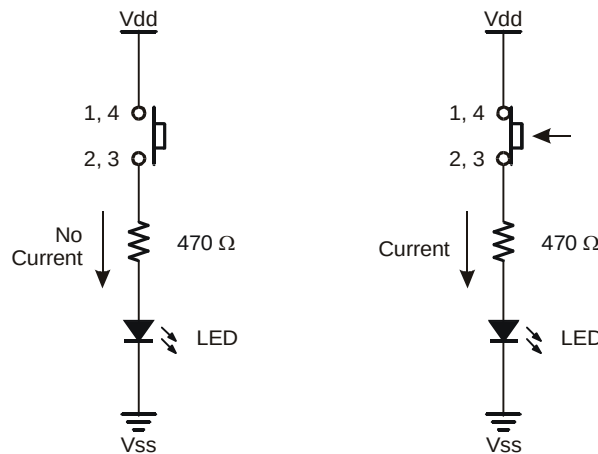


Figure 3-4
Pushbutton Not Pressed,
and Pressed

*Pushbutton not pressed:
circuit open and light off
(left)*

*Pushbutton pressed:
circuit closed and light on
(right)*

Your Turn – Turn the LED off with a Pushbutton

Figure 3-5 shows a 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. Since this pushbutton connects a conductor across terminals 1,4 and 2,3 when pressed, it means that electricity can take the path of least resistance through the pushbutton instead of through the LED. Unlike the potential short circuits discussed in the Warning Signs box, the short circuit the pressed pushbutton creates across the LED's terminals does not damage any circuits and serves a useful purpose.

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

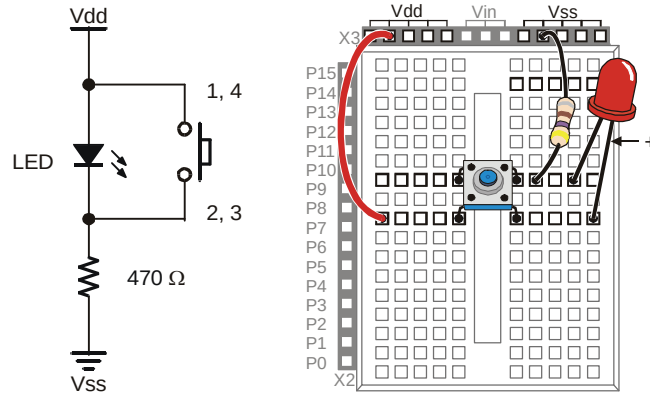


Figure 3-5
*LED that Gets Shorted
by Pushbutton*

3



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 red LED will always use about 1.7 volts, and the resistor will use the remaining 3.3 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.

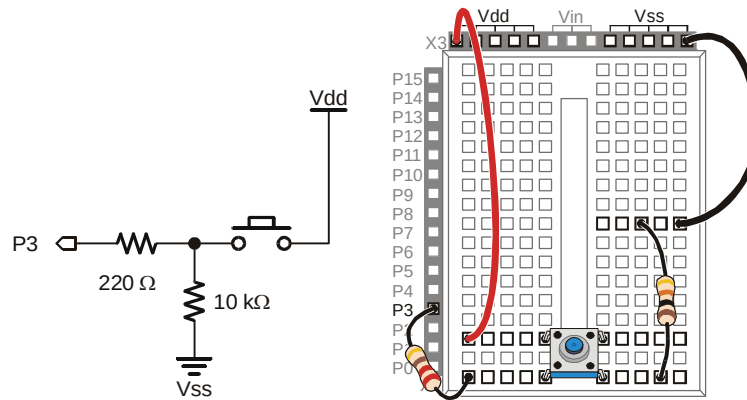


Figure 3-6
Pushbutton Circuit
Connected to I/O
Pin P3

On the wiring diagram, the 220 Ω resistor is on the left side connecting the pushbutton to P3 while the 10 k Ω resistor is on the right, connecting the pushbutton circuit to Vss.

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.

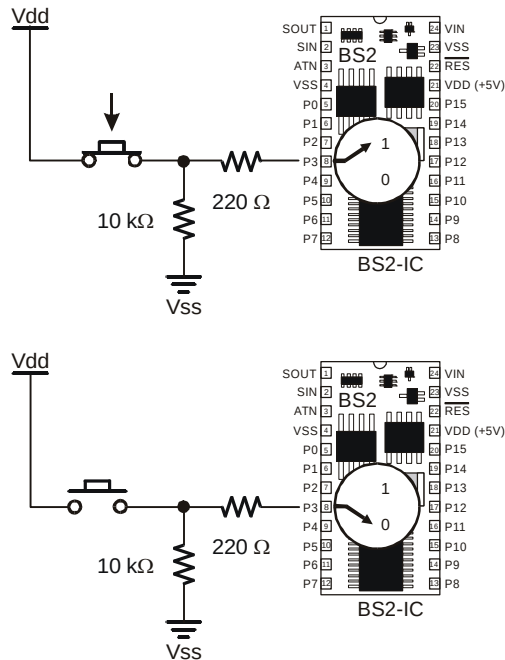


Figure 3-7
BASIC Stamp Reading a Pushbutton

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 next program makes the BASIC Stamp check the pushbutton every ¼ 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.

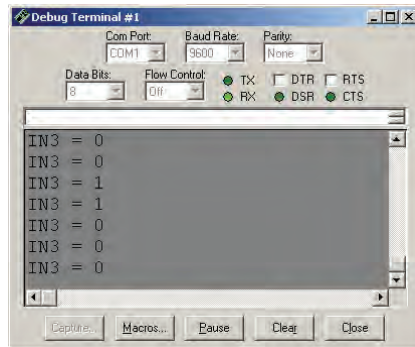


Figure 3-8

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 ¼ 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.

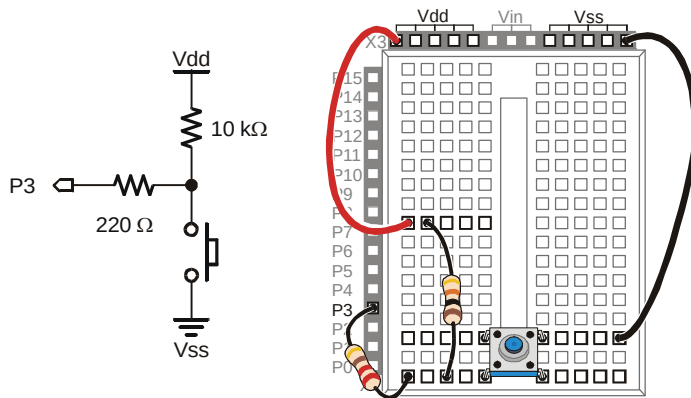


Figure 3-9
Modified Pushbutton
Circuit



Active-low vs. Active-high: The pushbutton circuit in 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 is called *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.

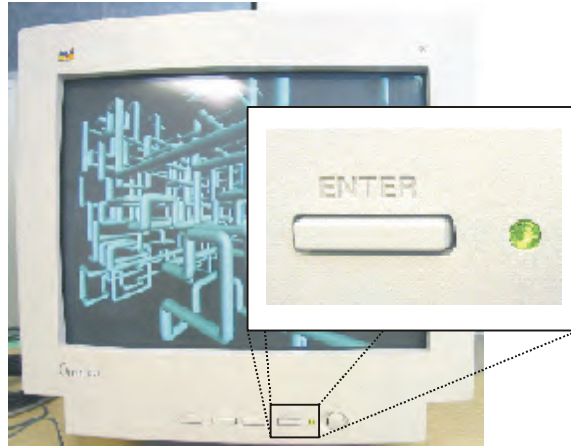


Figure 3-10
Button and LED on
a Computer Monitor

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 from Chapter 2, Activity #2.

- ✓ Build the circuit shown in Figure 3-11.

3

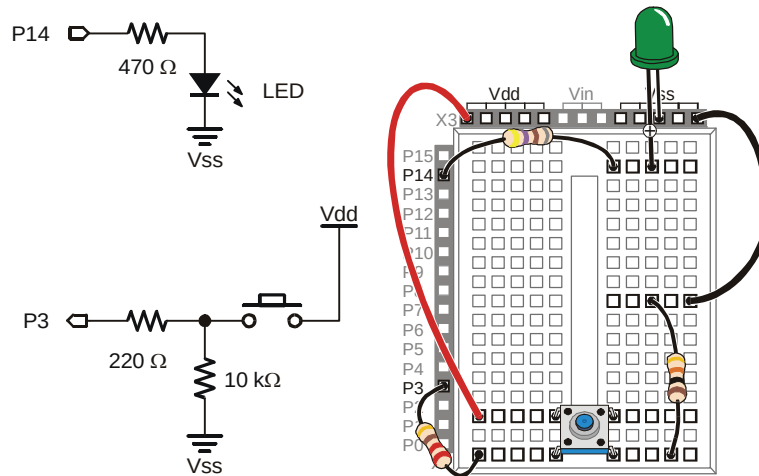


Figure 3-11
Pushbutton and
LED Circuit

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. 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.

ACTIVITY #4: TWO PUSHBUTTONS CONTROLLING TWO LED CIRCUITS

Let's add a second pushbutton to 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
- (3) Jumper wires

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.

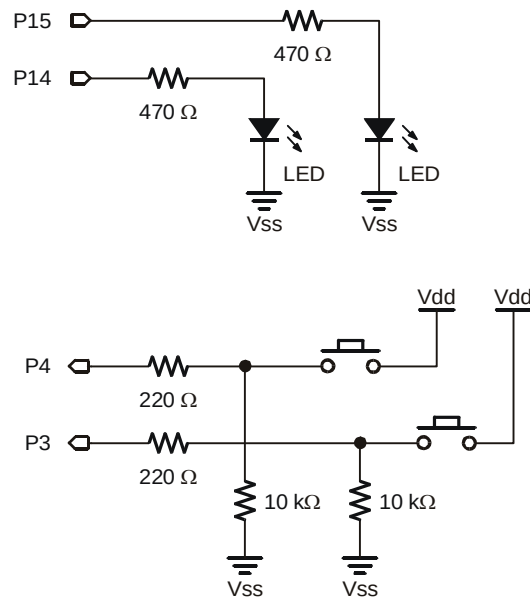


Figure 3-12
Schematic for Two
Pushbuttons and LEDs



Dots indicate connections: There are three places where lines intersect in Figure 3-12, but only two of those intersections have dots. When two lines intersect with a dot, it means they are electrically connected. For example, the 10 kΩ resistor on the lower-right side of Figure 3-12 has one of its terminals connected to one of the P3 circuit's pushbutton terminals and to one of its 220 Ω resistor terminals. When one line crosses another, but there is no dot, it means the two wires DO NOT electrically connect. For example, the line that connects the P4 pushbutton to the 10 kΩ resistor does not connect to the P3 pushbutton circuit because there is no dot at that intersection.

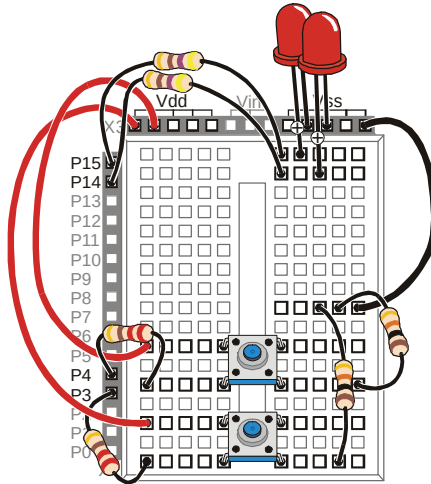


Figure 3-13
Wiring Diagram for Two Pushbuttons
and LEDs

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 and run PushbuttonControlOfTwoLeds.bs2 in the BASIC Stamp Editor.
- ✓ 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}

PAUSE 1000
```

```

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** control character:

```
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.

3

```
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.

Reaction Timer Game Parts

- (1) LED – bicolor
- (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 on page 80.
- ✓ Run `TestBiColorLED.bs2` from Chapter 2, Activity #5 to test the bicolor LED circuit and make sure your wiring is correct.
- ✓ If you just 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.

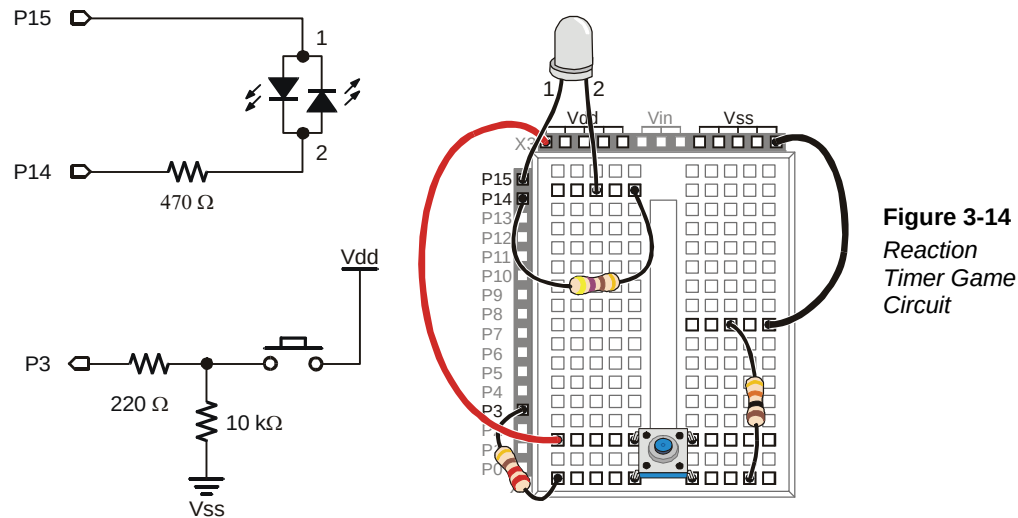


Figure 3-14
Reaction
Timer Game
Circuit

Programming the Reaction Timer

This next example program will leave the bicolor 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 program then measures time it takes the player to release the pushbutton in reaction to the light turning green.

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 bicolor 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).

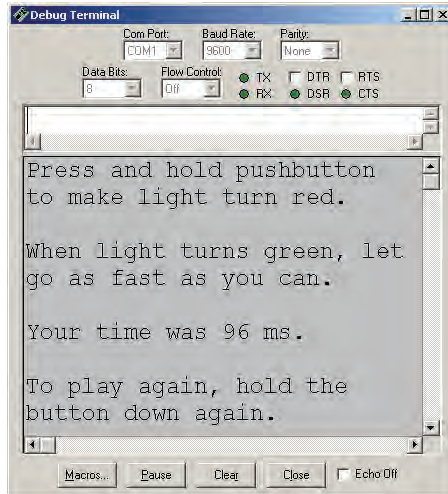


Figure 3-15
Reaction Timer Game Instructions in the Debug Terminal

```
' What's a Microcontroller - ReactionTimer.bs2
' Test reaction time with a pushbutton and a bicolor LED.

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

PAUSE 1000                                ' Wait 1 sec before 1st message.

timeCounter    VAR    Word                ' Declare variable to store time.

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

DO                                           ' Begin main loop.

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

  HIGH 14                                    ' Bicolor LED red.
  LOW 15

  PAUSE 1000                                ' Delay 1 second.

  LOW 14                                    ' Bicolor LED green.
  HIGH 15

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


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 bicolor LED red, delay for one second, then turn it green.

```
HIGH 14                          ' Bicolor LED red.
LOW 15

PAUSE 1000                        ' Delay 1 second.

LOW 14                            ' Bicolor LED green.
HIGH 15
```

As soon as the bicolor 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.
```

After the pushbutton is released, the bicolor LED is turned off.

```
LOW 15
```

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 (Advanced Topics)

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 around 14,000 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 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. 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 ' multiply by 3.69`

3

Multiplying by 2 will scale a result of 14,000 to 28,000, which isn't quite 30,000. $30,000 \div 14,000 \approx 2.14$. To multiply by 2.14 with the ***/** operator for increased precision, we need to figure out how many 256^{ths} are in 2.14. So, $2.14 \times 256 = 547.84 \approx 548$. You can use this value and the ***/** operator to replace **timecounter = timeCounter * 2**.

- ✓ Replace **timecounter = timeCounter * 2** with **timecounter = timeCounter */ 548** and retest your program.

Your 30-second test with the original, unmodified program may yield a value that's slightly different from 14,000. If so, you can use the same procedure with your test results to calculate a value for the ***/** operator to make your results even more precise.

- ✓ Try it!

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 1000** command inside the **DO...LOOP**, 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 that **PAUSE 1000** command so that the “random” **value** is added to its **Duration** argument.

```
PAUSE 1000 + value        ' <- Modify this

LOW 14
HIGH 15
```

- ✓ Since the largest value a byte can store is 255, the **PAUSE** command only varies by $\frac{1}{4}$ second. You can multiply the **value** variable by 4 to make the red light delay vary from 1 to just over 2 seconds.

```
DEBUG "Delay time ", ? 1000 + (value*4), CR ' <- Modify
PAUSE 1000 + (value * 4)                    ' <- Modify this again
```



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 for the seed (change **value** from 23 to some other number), it will result in a different pseudo random sequence.

3

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. Assuming you have made the other changes in items 1 and 2, **timeCounter** will now be 2 instead of 1 if the player releases the button before the light turns green. The changes below will work if **timeCounter** is either 1 or 2.

- If the value of *timeCounter* is less than or equal to 2 (*timeCounter* <= 2)
 - 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.
 - End If
 - Display a “To play again...” message.
- ✓ Modify your program by implementing this pseudo code in PBASIC to fix the cheating player problem.

SUMMARY

This chapter introduced the pushbutton and some 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.

Several programming concepts were introduced, including counting, pseudo code for planning program flow, code overhead in timing-sensitive applications, and seed values for pseudo random events.

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. The **RANDOM** command was introduced to add an element of unpredictability to an application, the Reaction Timer game.

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** control character do in the statement **DEBUG HOME**?

Exercises

1. Explain how to modify ReadPushbuttonState.bs2 on page 68 so that it reads the pushbutton every second instead of every ¼ 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

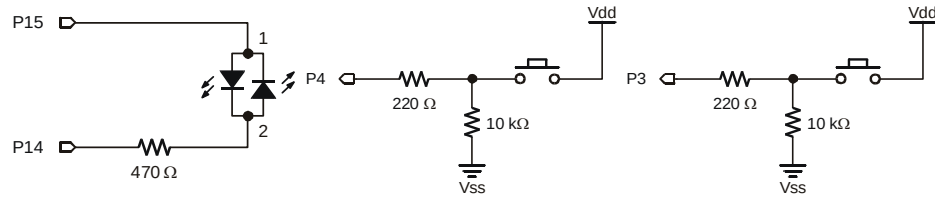
- Q1. Sending uses the BASIC Stamp I/O pin as an output, whereas receiving uses the I/O pin as an input.
- Q2. Normally open means the pushbutton's normal state (not pressed) forms an open circuit.
- Q3. When pressed, the gap between the terminals is bridged by a conductive metal. Current can then flow through the pushbutton.
- Q4. **IN3 = 1** when pushbutton connects it to Vdd. **IN3 = 0** when pushbutton connects it to Vss.
- Q5. **DEBUG ? IN3** displays the text “IN3 = ” followed by the value stored in **IN3** (either a 0 or 1 depending on the state of I/O pin P3), followed by a carriage return.
- Q6. **IF . . . THEN . . . ELSE** and **IF . . . ELSEIF . . . ELSE**.
- Q7. The **HOME** control character sends the cursor to the top left position in the Debug Terminal.
- E1. The **DO . . . LOOP** in the program repeats every ¼ second because of the command **PAUSE 250**. To repeat every second, change the **PAUSE 250** (250 ms = 0.25 s = ¼ s), to **PAUSE 1000** (1000ms = 1 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** command 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 BASIC Stamp I/O pin P4. The schematic is based on Figure 3-14 on page 80.



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 - Ch03Prj01_TwoPlayerReactionTimer.bs2
' Test reaction time with a pushbutton and a bicolor 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
PAUSE 1000              ' 1 s before 1st message

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)

  HIGH 14                ' Bicolor LED red.
  LOW 15

  PAUSE 1000             ' Delay 1 second.

  LOW 14                 ' Bicolor LED green.
  HIGH 15
```

```
timeCounterA = 0
timeCounterB = 0

DO

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

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

LOW 15

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
    DEBUG "It's a tie!", CR
ENDIF
DEBUG CR

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

LOOP
```

Chapter 4: Controlling Motion

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 DVD player is also controlled by a microcontroller.

4

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 much faster than the blinking LED examples from Chapter 2. If you were to use an LED circuit to monitor control signals, some would make the LED flicker on/off so rapidly that the human eye could not detect the switching. The LED would only appear to glow faintly. Others would appear as a rapid flicker, and others would be more easily discernible.

Some 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.

INTRODUCING THE SERVO

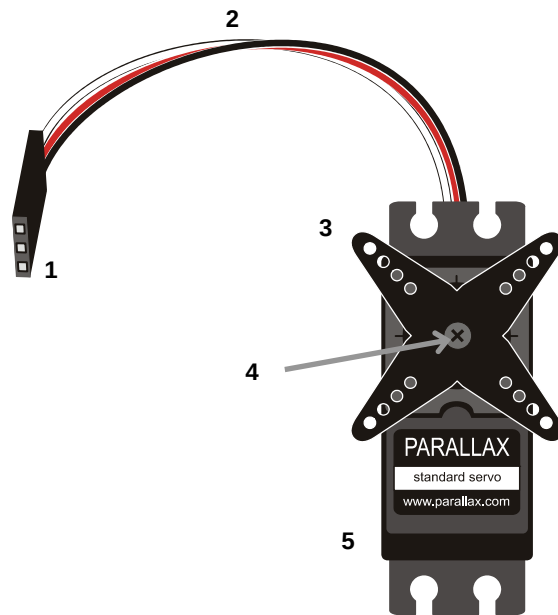
A hobby servo is a device that controls position, and you can find them in just about any radio controlled (RC) car, boat or plane. In RC cars, the servo holds the steering to control how sharply the car turns. In an RC boat, it holds the rudder in position for turns. RC planes typically have several servos that position the different flaps to control the plane's motion. In RC vehicles with gas powered engines, another servo moves the engine's throttle lever to control how fast the engine runs. An example of an RC airplane and its radio controller are shown in Figure 4-1. The hobbyist "flies" the airplane by manipulating thumb joysticks on the radio controller, which causes the servos on the plane to control the positions of the RC plane's elevator flaps and rudder.



Figure 4-1
Model Airplane and
Radio Controller

So, how does holding the radio controller's joystick in a certain position cause a flap on the RC plane to hold a certain position? The radio controller converts the position of the joysticks into pulses of radio activity that last certain amounts of time. The time each pulse lasts indicates the position of one of the joysticks. On the RC plane, a radio receiver converts these radio activity pulses to digital pulses (high/low signals) and sends them to the plane's servos. Each servo has circuitry inside it that converts these digital pulses to a position that the servo maintains. The amount of time each pulse lasts is what tells the servo what position to maintain. These control pulses only last a few thousandths of a second, and repeat around 40 to 50 times per second to make the servo maintain the position it holds.

Figure 4-2 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) has three wires, and it 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 servo holds in different positions. The Phillips screw (4) holds the horn to the servo's output shaft. The case (5) contains the servo's position sensing and control circuits, a DC motor, and gears. These parts work together to take high/low signals from the BASIC Stamp and translate them into positions held by the servo horn.

**Figure 4-2**

The Parallax Standard Servo

- (1) Plug
- (2) Cable
- (3) Horn
- (4) Screw that attaches the horn to the servo's output shaft
- (5) Case

4

In this chapter, you will program the BASIC Stamp to send signals to a servo that control the servo horn's position. By making the BASIC Stamp send signals that tell the servo to hold different positions, your programs can also orchestrate the servo's motion. Your programs can even monitor pushbuttons and use information about whether the buttons are pressed to adjust the position a servo holds (pushbutton servo position control). The BASIC Stamp can also be programmed to receive messages that you type into the Debug Terminal, and use those messages to control the servo's position (terminal servo position control).

ACTIVITY #1: CONNECTING AND TESTING THE SERVO

In this activity, you will follow instructions for connecting a servo to your particular board's power supply and BASIC Stamp.

Servo and LED Circuit Parts

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

The LED circuit will 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” the control signals.



CAUTION: use only a Parallax Standard Servo for the activities in this text! Other servos may be designed to different specifications that might not be compatible with these activities.

Building the Servo and LED Circuits

In Chapter 1, you identified your board and revision using the BASIC Stamp Editor Help. You will need to know which board and revision you have here so that you can find the servo circuit building instructions for your board.

- ✓ If you do not already know which board and revision you have, open the BASIC Stamp Editor Help and click on the Getting Started with Stamps in Class link on the home page. Then, follow the directions to determine which board you have.
- ✓ If you have a Board of Education USB (any Rev) or Serial (Rev C or newer), go to the Board of Education Servo Circuit section below.
- ✓ If you have a BASIC Stamp HomeWork Board (Rev C or newer), go the BASIC Stamp HomeWork Board Servo Circuit section on page 99.
- ✓ If your board is not listed above, go to www.parallax.com/Go/WAM → Servo Circuit Connections to find circuit instructions for your board. When you are done with the servo circuit instructions for your board, go on to Activity #2: Servo Control Test Program on page 102.

Board of Education Servo Circuit

These instructions are for all USB Board of Education Revisions as well as for the Serial Board of Education Rev C or newer.

- ✓ Turn off the power as shown in Figure 4-3.

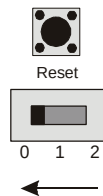


Figure 4-3
Disconnect Power

Set 3-position switch to 0

4

Figure 4-4 shows the servo header on the Board of Education. 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 Vin or Vdd. The jumper is the removable black rectangular piece indicated by the arrow between the two servo headers.

- ✓ Verify that the jumper is set to Vdd as shown in Figure 4-4. If it is instead set to Vin, lift the rectangular jumper up off of the pins it is currently on, and then press it on the two pins closest to the Vdd label.

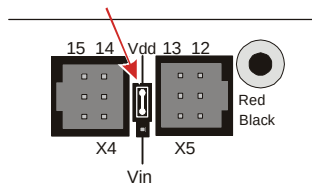


Figure 4-4
Servo Header Jumper Set to Vdd



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, set it to Vdd. DO NOT USE Wall-mount 9 V Battery “replacers.”
- If you are using a 4 AA cell, 6 V battery pack, either setting will work.
- If you are using a wall-mount DC power supply, use only Vdd. Before connecting a wall-mount DC supply to the Board of Education, make sure to check the specifications for acceptable DC supplies listed in the BASIC Stamp Editor Help.

Figure 4-5 shows the schematic of the circuit you will build on your Board of Education.

- ✓ Build the circuit shown in Figure 4-5 and 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 Figure 4-6.

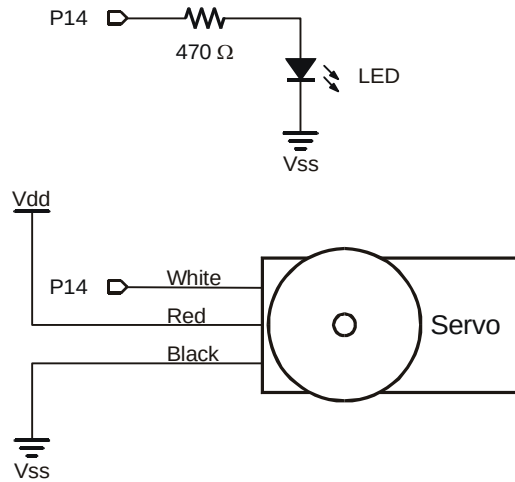


Figure 4-5
Servo and LED
Indicator Schematic
for Board of
Education

*For Serial Board of
Education Rev C or
newer, or any USB
Board of Education*

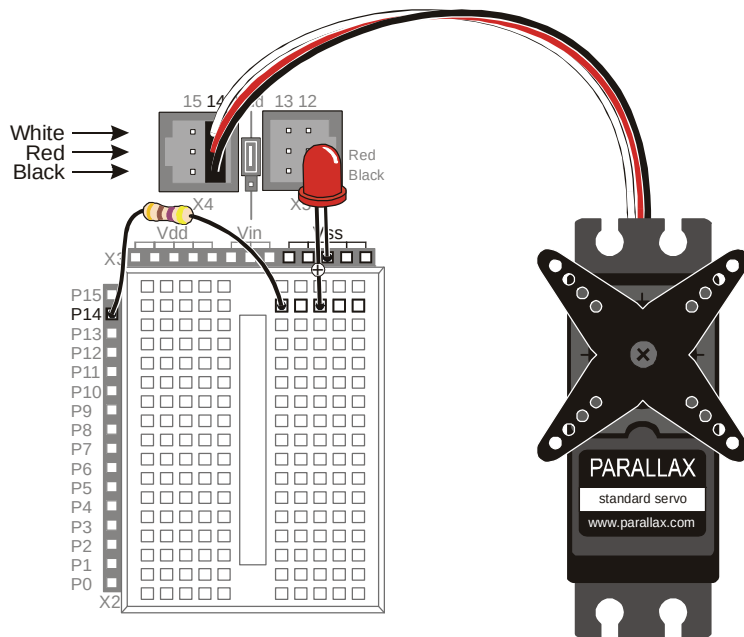


Figure 4-6
Servo and LED
Indicator on Board
of Education

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

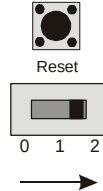


Figure 4-7

Power turned on to Board of Education and Servo Header

4

If you see instructions in this chapter that read “Connect power to your board” move the 3-position switch to position-2. Likewise, if you see instructions in this chapter that read “Disconnect power from your board” move the 3-position switch to position-0.

- ✓ Disconnect power from your board.
- ✓ Go on to Activity #2 on page 102.

BASIC Stamp HomeWork Board Servo Circuit

If you are connecting your servo to a BASIC Stamp HomeWork Board (Rev C or newer), you will need these extra parts from your kit:

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



Figure 4-8

Extra Part for BASIC Stamp HomeWork Board Servo Circuit

3-pin male/male header

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

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

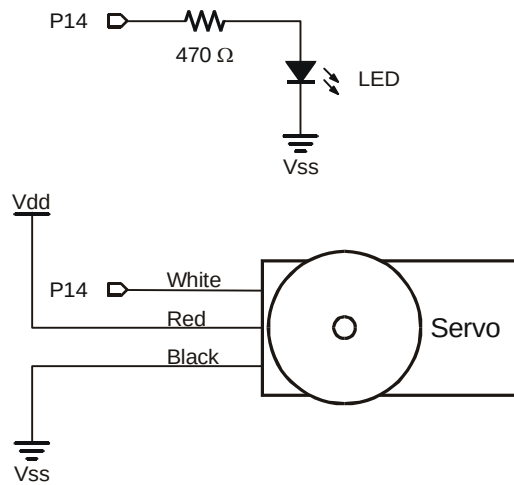


Figure 4-9
Schematic for Servo and
LED Indicator on
HomeWork Board

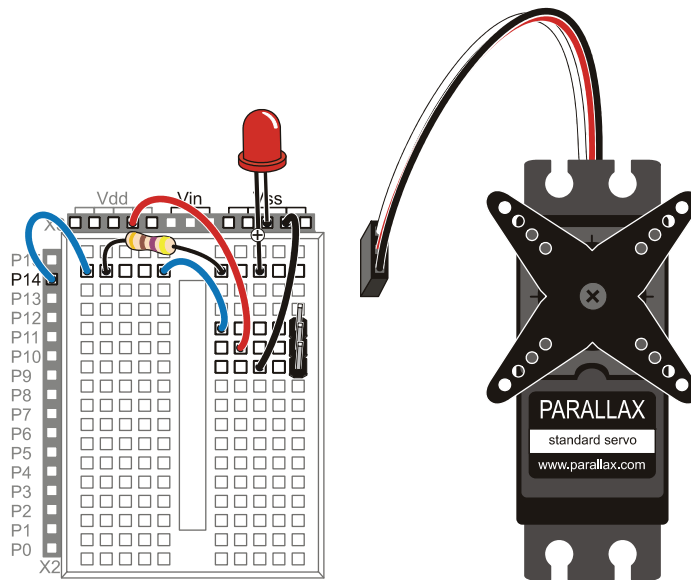


Figure 4-10
LED Indicator and Servo
Header Circuits on
HomeWork Board

- ✓ 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.

**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.

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.

4

- ✓ Reconnect your 9 V battery to your HomeWork Board. The servo may twitch slightly when you make the connection.

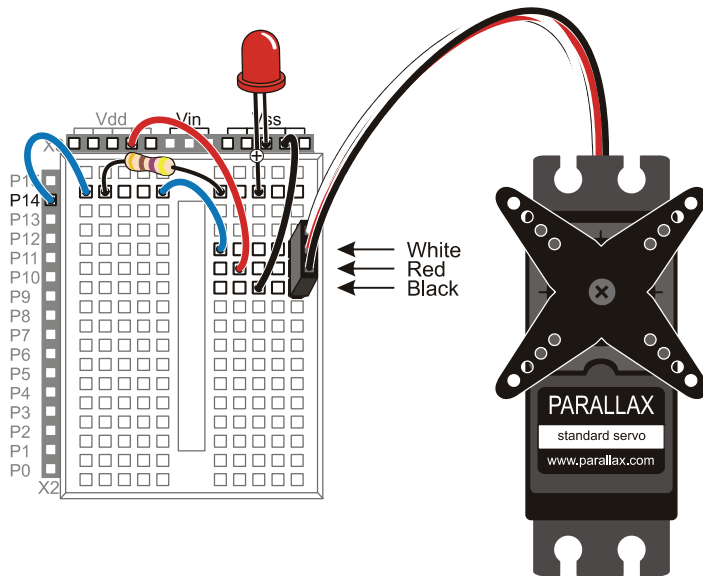


Figure 4-11
Servo Connected to
HomeWork Board

ACTIVITY #2: SERVO CONTROL TEST PROGRAM

A *degree* is an angle measurement denoted by the $^{\circ}$ symbol. Example degree angle measurements are shown in Figure 4-12, including 30° , 45° , 90° , 135° , and 180° . Each degree of angle measurement represents $1/360^{\text{th}}$ of a circle, so the 90° measurement is $\frac{1}{4}$ of a circle since $90 \div 360 = \frac{1}{4}$. Likewise, 180° is $\frac{1}{2}$ of a circle since $180 \div 360 = \frac{1}{2}$, and you can calculate similar fractions for the other degree measurements in the figure.

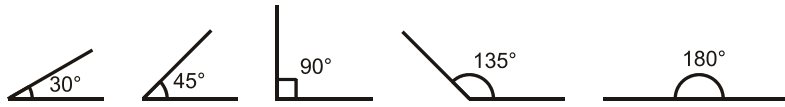
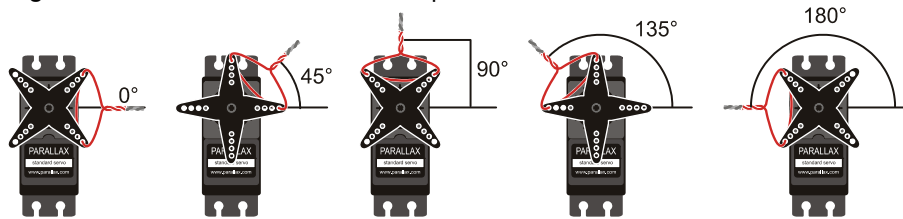


Figure 4-12
Examples of
Degree Angle
Measurements

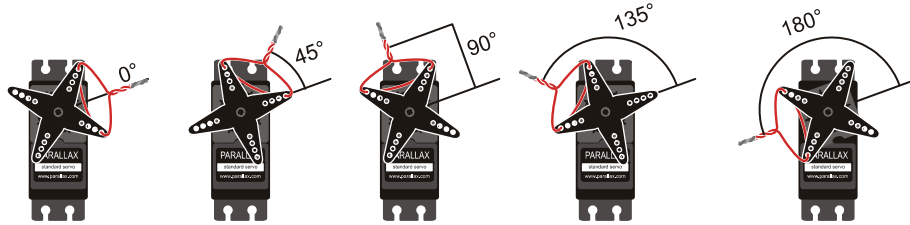
The Parallax Standard Servo can make its horn hold positions anywhere within a 180° range, so degree measurements can be useful for describing the positions the servo holds. Figure 4-13 shows examples of a servo with a loop of wire that has been threaded through two of the holes in its horn and then twist-tied. The direction the twist tie points indicates the angle of the servo's horn, and the figure shows examples of 0° , 45° , 90° , 135° , and 180° .

Figure 4-13: Servo Horn Position Examples



Your servo horn's range of motion and mechanical limits will probably be different from what's shown here. Instructions on how to adjust it to match this figure come after the first example program.

Factory servo horn mounting is random, so your servo horn positions will probably be different from the ones in Figure 4-13. In fact, compared to Figure 4-13, your servo's horn could be mounted anywhere in a $\pm 45^{\circ}$ range. The servo in Figure 4-14 shows an example of a servo whose horn was mounted 20° clockwise from the one in Figure 4-13. After you find the center of the servo horn's range of motion, you can either use it as a 90° reference or mechanically adjust the servo's horn so that it matches Figure 4-13 by following instructions later in this activity.

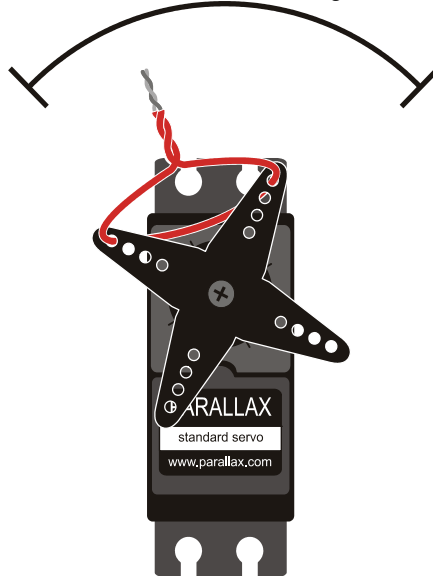
Figure 4-14: Servo Horn Position Examples before Mechanical Adjustment

4

This is an example of a horn that's mounted on the servo's output shaft about 20° counterclockwise of how it was set in Figure 4-13.

You can find the center of the servo's range of motion by gently rotating the horn to find its clockwise and counterclockwise mechanical limits. The half way position between these two limits is the center or 90° position. The servo's center position could fall anywhere in the region shown in Figure 4-15.

The center of your servo horn's range of motion should fall somewhere in this region

**Figure 4-15**
Range of Possible
Center Positions



In these next steps, twist the servo horn slowly and do not force it! The servo has built-in mechanical limits to prevent the horn from rotating outside its 180° range of motion. Twist the horn gently, and you'll be able to feel the when it reaches one of its mechanical limits. Don't try to force it beyond those limits because it could strip the gears inside the servo.

- ✓ Verify that the power to your board is still disconnected.
- ✓ Gently rotate the servo horn to find the servo's clockwise and counterclockwise mechanical limits. The servos horn will turn with very little twisting force until you reach these limits. **DO NOT TRY TO TWIST THE HORN PAST THESE LIMITS;** only twist it far enough to find them.
- ✓ Rotate the servo's horn so that it is half way between the two limits. This is approximately the servo's "center" position.
- ✓ With the servo horn in its center position, thread a jumper wire through the horn and twist tie it so that it points upward into the region shown in Figure 4-15.

Keep in mind the direction the twist tie is pointing in the figure is just an example; your twist tie might point anywhere in the region. Wherever it points when it's in the center of its range of motion should be pretty close to the servo's 90° position. Again, this position can vary from one servo to the next because of the way the horn gets attached to the servo.

Programming Servo Positions

The graph in Figure 4-16 is called a *timing diagram*, and it shows examples of the high/low signals the BASIC Stamp has to send a servo to make it hold its 90° position.

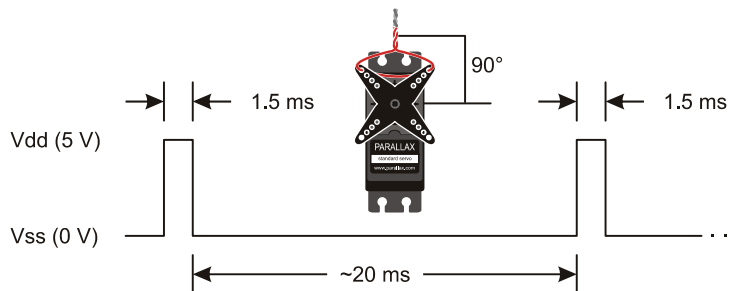


Figure 4-16
Servo Signal Timing
Diagram

*1.5 ms pulses make
the servo hold a 90°
"center" position.*

The timing diagram shows high signals that last for 1.5 ms, separated by low signals that last 20 ms. The ... to the right of the signal is a way of indicating that the 1.5 ms high and 20 ms low signal has to be repeated over and over again to make the servo hold the

position. The “~” symbol in “~20 ms” indicates that the low time can be approximate, and it can actually vary a few milliseconds above or below 20 ms with next to no effect on the where the servo positions its horn. That’s because amount of time the high signal lasts is what tells the servo what position to hold, so it has to be precise.

There’s a special command called **PULSOUT** that gives your program precise control over the durations of those very brief high signals, which are commonly referred to as pulses. Here is the command syntax for **PULSOUT**:

PULSOUT *Pin, Duration*

With the **PULSOUT** command, you can write PBASIC code to make the BASIC Stamp set the servo’s position to 90° using the Figure 4-16 timing diagram as a guide. The **PULSOUT** command’s *Pin* argument has to be a number that tells the BASIC Stamp which I/O pin should transmit the pulse. The **PULSOUT** command’s *Duration* argument is the number of 2-millionths-of-a-second time increments the pulse should last. 2 millionths of a second is equal to 2 microseconds, which is abbreviated 2 μ s.



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 μ s.

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

Fact: 1 ms = 1000 μ s. In other words, you can fit one thousand *millionths* of a second into one *thousandth* of a second.

Now that we know how to use the **PULSOUT** command, ServoCenter.bs2 sends control pulses repeatedly to make the servo hold its 90° position. The command **PULSOUT 14, 750** will send a 1.5 ms pulse to the servo. That’s because the **PULSOUT** command’s *Duration* argument specifies the number of 2 μ s units the pulse should last. Since the *Duration* argument is 750, the **PULSOUT** command will make the pulse last for $750 \times 2 \mu\text{s} = 1500 \mu\text{s}$, which is 1.5 ms since there are 1000 μ s in 1 ms. After the high pulse is done, the **PULSOUT** command leaves the I/O pin sending a low signal. So, a **PAUSE 20** command after **PULSOUT** makes the BASIC Stamp send a low signal for 20 ms. With both of those commands inside a **DO . . . LOOP**, the 1.5 ms high followed by the 20 ms low will repeat over and over again to make the servo hold its position.

Example Program: ServoCenter.bs2

```
' What's a Microcontroller - ServoCenter.bs2
' Hold the servo in its 90 degree center position.

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

DEBUG "Program Running!", CR

DO
  PULSOUT 14, 750
  PAUSE 20
LOOP
```

Test the Servo's 90° "Center" Position

The servo's 90° position is called its *center position* because the 90° point is in the "center" of the servo's 180° range of motion. The 1.5 ms pulses make the servo hold its horn in this center position, which should be close to the half way point you determined by finding the servo's mechanical limits. You can either use whatever center position the servo holds as your reference for 90°, or use a screwdriver to remove and reposition the horn so that 90° makes the jumper wire twist tie point straight up. Instructions for this are coming up in the section titled: Optional – Adjust Servo Horn to 90° Center. If you use the center position as a reference without adjusting it, any other position the servo holds will be relative that 90° position. For instance, the 45° position would be 1/8 of a turn clockwise from it, and the 135° position would be 1/8 of a turn counterclockwise. Examples of this were shown in Figure 4-14 on page 103.

Let's first find what your servo's actual center position is:

- ✓ Gently turn the servo's horn to one of its mechanical limits.
- ✓ Reconnect power to your board. If you have a Board of Education, make sure to slide the 3-position all the way to the right (to position-2).
- ✓ Run ServoCenter.bs2.

As soon as the program loads, the servo's horn should rotate to its center position and stay there. The servo "holds" this position, because standard servos are designed to resist external forces that push against it. That's how the servo holds the RC car steering, boat rudder, or airplane control flap in place.

- ✓ Make a note of your servo's center position.

- ✓ Apply gentle twisting pressure to the horn like you did while rotating the servo to find its mechanical limits. The servo should resist and hold its horn in its center position.

If you disconnect power, you can rotate the servo away from its center position. When you reconnect power, the program will restart, and servo will immediately move the horn back to its center position and hold it there.

- ✓ Try it!

4

Optional – Adjust Servo Horn to 90° Center

You can optionally adjust your servo's horn so that it makes the jumper wire twist tie point straight up when ServoCenter.bs2 is running, like it does in the right side of Figure 4-17. If you make this mechanical adjustment, it'll simplify tracking the servo's angles because each angle will resemble the ones in Figure 4-13 on page 102.



You will need a #2 Phillips screwdriver for this optional adjustment.

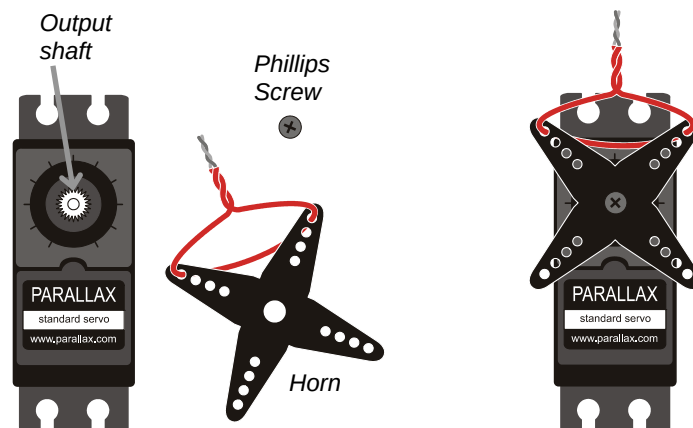


Figure 4-17
Mechanical Servo
Centering

*You can remove
and reposition the
servo horn on the
output shaft with a
small screwdriver.*

- ✓ Disconnect power from your board.

- ✓ Remove the screw that attaches the servo's horn to its output shaft, and then gently pull the horn away from case to free it. Your parts should resemble the left side of Figure 4-17.
- ✓ Reconnect power to your board. The program should make the servo hold its output shaft in the center position.
- ✓ Slip the horn back onto the servo's output shaft so that it makes the twist tied wire point straight up like it does on the right side of Figure 4-17.



Alignment Offset: It might not be possible to get it to line up perfectly because of the way the horn fits onto the output shaft, but it should be close. You can then adjust the wire loop to compensate for this small offset and make the twist tie point straight up.

- ✓ Disconnect power from your board.
- ✓ Retighten the Phillips screw.
- ✓ Reconnect power so that the program makes the servo hold its center position again. The twist tie should now point straight up (or almost straight up) indicating the 90° position.

Your Turn – Programs to Point the Servo in Different Directions

Figure 4-18 shows a few **PULSOUT** commands that tell the servo to hold certain major positions, like 0°, 45°, 90°, 135°, and 180°. These **PULSOUT** commands are approximate, and you may have to adjust the values slightly to get more precise angular positions. You can modify the **PULSOUT** command's **Duration** argument to hold any position in this range. For example, if you want the servo to hold the 30° position, your **PULSOUT** command's **Duration** argument would have to be 417, which is 2/3 of the way between **Duration** arguments of 250 (0°) to 500 (45°).



The pulse durations in Figure 4-18 will get the servo horn close to the angles shown, but they are not necessarily exact. You can experiment with different **PULSOUT Duration** values for more precise positioning.

- ✓ Save a copy of ServoCenter.bs2 as TestServoPositions.bs2
- ✓ Change the program's **PULSOUT Duration** argument from 750 to 500, and run the modified program to verify that it makes the servo hold its 45° position.
- ✓ Repeat this test of **PULSOUT Duration** arguments with 1000 (135°), and 417 (30°).

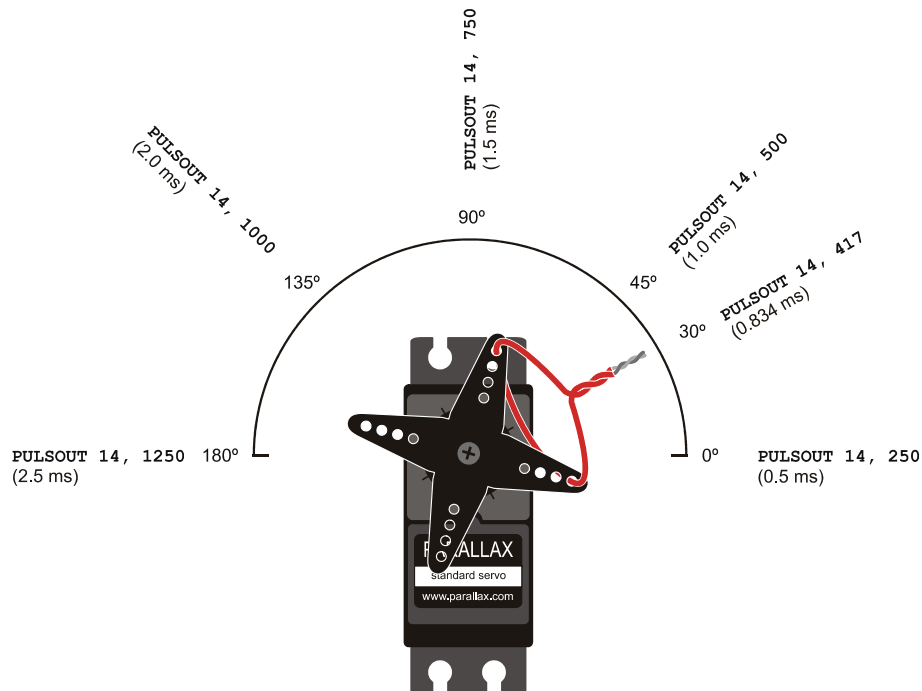
- ✓ Try predicting a **PULSOUT Duration** you would need for a position that's not listed in Figure 4-18, and test to make sure the servo turns the horn to and holds the position you want. Example positions could include 60°, 120°, etc.



Keep your program's PULSOUT Duration arguments in the 350 to 1150 range. The 250 to 1250 range is "in theory" but in practice the servo might try to push against its mechanical limits. This can reduce the servo's useful life. If you want to maximize your servo's range of motion, carefully test values that get gradually closer to the mechanical limits. So long as you use **PULSOUT Duration** values that cause the servo to position its horn just inside its mechanical limits, wear and tear will be normal instead of excessive.

4

Figure 4-18: Servo Horn Positions, PULSOUT Commands, and ms Pulse Durations



Do the Math

Along with each **PULSOUT** command in Figure 4-18, there's a corresponding number of milliseconds that each pulse lasts. For example, the pulse that **PULSOUT 14, 417** sends lasts 0.834 ms, and the pulse that **PULSOUT 14, 500** sends lasts 1.0 ms. If you have a BASIC Stamp 2 and want to convert time from milliseconds to a **Duration** for your **PULSOUT** command, use this equation:

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

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

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

The reason we have to multiply the number of milliseconds in a pulse by 500 to get a **PULSOUT Duration** argument is because **Duration** is in terms of 2 μs units for a BS2. How many 2 μs units are in 1 ms? Just divide 2-one-millionths into 1-one-thousandth to find out.

$$\frac{1}{1,000} \div \frac{2}{1,000,000} = 500$$

If your command is **PULSOUT 14, 500**, the pulse will last for $500 \times 2 \mu\text{s} = 1000 \mu\text{s} = 1.0 \text{ ms}$. (Remember, $1000 \mu\text{s} = 1 \text{ ms}$.)

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?

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

Write Code from Timing Diagrams

Figure 4-19 shows a timing diagram of the signal the BASIC Stamp can send to a servo so its horn will hold a 135° position. Since this timing diagram features repeated pulses separated by 20 ms low signals, the **DO . . . LOOP** from ServoCenter.bs2 provides a good starting point, and all that needs to be adjusted is the high pulse duration. To calculate the **PULSOUT** command's *Duration* argument for the 2 ms pulses in the timing diagram, you can use the *Duration* equation in the Do the Math section:

$$\begin{aligned} \text{Duration} &= \text{number of ms} \times 500 \\ &= 2.0 \times 500 \\ &= 1000 \end{aligned}$$

When 1000 gets substituted into the **PULSOUT** command's *Duration* argument, the servo control loop should look like this:

```
DO
  PULSOUT 14, 1000
  PAUSE 20
LOOP
```

- ✓ Test this **DO . . . LOOP** in a copy of ServoCenter.bs2 and verify that it positions the servo's horn at approximately 135°.
- ✓ Repeat this exercise for the timing diagram in Figure 4-20.

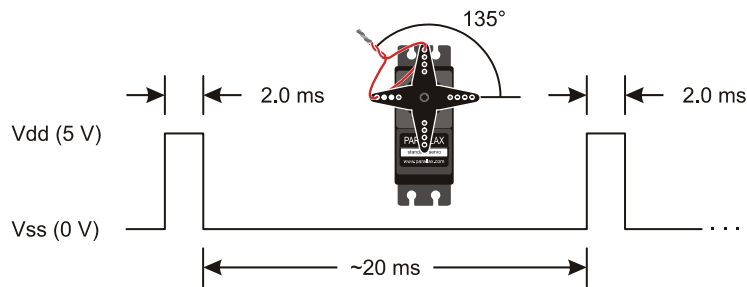


Figure 4-19
Timing Diagram for
135° Position

*2 ms pulses
separated by 20
ms*

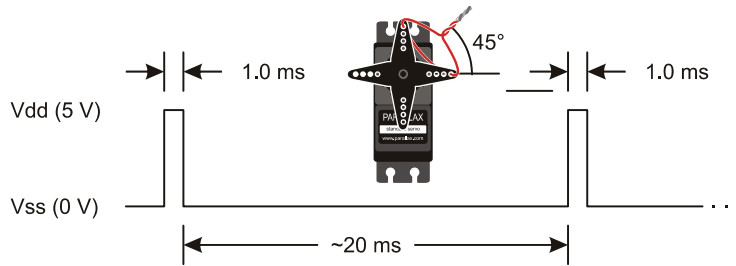


Figure 4-20
Timing Diagram for
45° Position

*1 ms pulses
separated by 20 ms*

ACTIVITY #3: CONTROL SERVO HOLD TIME

Animatronics uses electronics to animate props and special effects, and servos are a common tool in this field. Figure 4-21 shows an example of a robotic hand animatronics project, with servos controlling each finger. The PBASIC program that controls the hand gestures has to make the servos hold positions for certain amounts of time for each gesture. In the previous activity, our programs made the servo hold certain positions indefinitely. This activity introduces how write code that makes the servo hold certain positions for certain amounts of time.



Figure 4-21
Animatronic Hand

There are five servos in the lower right of the figure that pull bicycle break cables that are threaded through the fingers and thumb to make them flex. This gives the BASIC Stamp control over each finger.

FOR . . . NEXT Loops to Control the Time a Servo Holds a Position

If you write code to make an LED blink once every second, you can nest the code in a **FOR . . . NEXT** loop that repeats three times to make the light blink for three seconds. If your LED blinks five times per second, you'd have to make the **FOR . . . NEXT** loop repeat fifteen times to get the LED to blink for three seconds. Since the **PULSOUT** and **PAUSE** commands that control your servo are responsible for sending high/low signals, they also make the LED blink. The signals we sent to the servo in the previous activity made the LED glow faintly, maybe with some apparent flicker, because the on/off signals are so rapid, and the high times are so brief. Let's slow the signals down to 1/10th speed for visible LED indicator light blinking.

Example Program: SlowServoSignalForLed.bs2

Compared to the servo center signal, this example program increases the **PULSOUT** and **PAUSE** durations by a factor of ten so that we can see them as LED indicator light blinking. The program's **FOR . . . NEXT** loop repeats at almost 5 times per second, so 15 repetitions results in making the light blink for three seconds.

- ✓ Disconnect Power to your servo:
 - If you have a Board of Education, set the 3-position switch to position-1 to disconnect power from the servo. Position-1 will still supply power to the rest of the system.
 - If you have a BASIC Stamp Homework Board, temporarily unplug the end of the wire that's plugged into Vdd and leave it floating. This will disconnect power from your servo.
- ✓ Enter and run SlowServoSignalsForLed.bs2.
- ✓ Verify that the LED blinks rapidly for about three seconds.
- ✓ Change the **FOR . . . NEXT** loop's **EndValue** from 15 to 30 and re-run the program. Since the loop repeated twice as many times, the light should blink for twice as long – six seconds.
- ✓ Reconnect power to your servo:
 - If you have a Board of Education, set the 3-position switch back to position-2 to reconnect power to the servo.
 - If you have a BASIC Stamp Homework Board, plug the end of the wire you disconnected back into the Vdd socket.

```
' What's a Microcontroller - SlowServoSignalsForLed.bs2
' Slow down the servo signals to 1/10 speed so that they are we can
' see the LED indicator blink on/off.
```

```

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

DEBUG "Program Running!", CR

counter      VAR      Word

FOR counter = 1 to 15
  PULSOUT 14, 7500
  PAUSE 200
NEXT

```

Example Program: ThreeServoPositions.bs2

If you change **PULSOUT 14, 7500** to **PULSOUT 14, 750** and **PAUSE 200** to **PAUSE 20**, you will have a **FOR...NEXT** loop that briefly sends the center position signal to the servo. Since the signals now last $1/10^{\text{th}}$ of their durations in *SlowServoSignalsForLed.bs2*, the entire **FOR...NEXT** loop will take $1/10^{\text{th}}$ the time to execute. If the goal is to make the servo hold a particular position for three seconds, simply deliver ten times as many pulses by increasing the **FOR...NEXT** loop's *EndValue* argument from 15 to 150.

```

FOR counter = 1 to 150          ' Center for about 3 sec.
  PULSOUT 14, 750
  PAUSE 20
LOOP

```

The *ThreeServoPositions.bs2* example program makes the servo hold the three different positions shown in Figure 4-22, each for about 3 seconds.

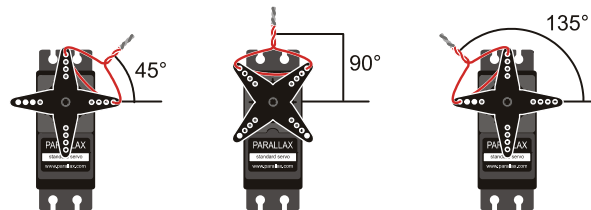


Figure 4-22
ThreeServoPositions.bs2

The program makes the servo hold each position for about three seconds.

- ✓ Enter and run *ThreeServoPositions.bs2*.
- ✓ Verify that the servo holds each position in the Figure 4-22 sequence for about three seconds.

The last position the servo will hold for 3 seconds is 135° and then the program stops. The servo horn will stay in the same position even though the BASIC Stamp has stopped sending control pulses. The difference is that during the three seconds that the BASIC Stamp holds the 135° position, the servo resists any forces that try to push the horn away from that position. After the 3 seconds is up, the servo's horn can be turned by hand.

One way you can tell if the servo is receiving control signals is by watching the indicator LED that is connected to P14. While the indicator LED glows, it means the servo is receiving control signals and is holding its position. When the signal stops you'll see the glow in the indicator LED stop.

- ✓ Re-run the program (or just press and release your board's Reset Button).
- ✓ As soon as the servo gets to the 135° position, keep an eye on the signal indicator LED as you apply light twisting force to the horn.

You should be able to feel the servo resisting while the LED glows faintly indicating the servo is still receiving a control signal. As soon as the LED turns off indicating that the control signal has stopped, the servo will stop holding its position, and you will be able to rotate the horn.

- ✓ When the 135° signal stops, verify that the LED indicates the signal has stopped and that the servo allows you to twist the horn away from the 135° position.

```
' What's a Microcontroller - ThreeServoPositions.bs2
' Servo holds the 45, 90, and 135 degree positions for about 3 seconds each.

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

counter      VAR      Word

PAUSE 1000

DEBUG "Position = 45 degrees...", CR

FOR counter = 1 TO 150                      ' 45 degrees for about 3 sec.
  PULSOUT 14, 500
  PAUSE 20
NEXT

DEBUG "Position = 90 degrees...", CR
```

```

FOR counter = 1 TO 150                                ' 90 degrees for about 3 sec.
  PULSOUT 14, 750
  PAUSE 20
NEXT

DEBUG "Position = 135 degrees...", CR

FOR counter = 1 TO 150                                ' 135 degrees for about 3 sec.
  PULSOUT 14, 1000
  PAUSE 20
NEXT

DEBUG "All done.", CR, CR

END

```

Your Turn – Adjusting Position vs. Adjusting Hold Time

ThreeServoPositions.bs2 assumes that executing 50 servo pulses in a **FOR...NEXT** loop takes about 1 second. You can also use this to adjust a hold time by adjusting the **FOR...NEXT** loop's **EndValue** argument. For example, if you want the servo to only hold its position for two about seconds, change the **EndValue** argument from 150 to 100. For five seconds, change it from 150 to 250, and so on...

- ✓ Save a copy of ThreeServoPositions.bs2.
- ✓ Modify each **FOR...NEXT** loop's **EndValue** argument and experiment with different values for different hold times.
- ✓ Optional: Customize the hold positions by adjusting each **PULSOUT** command's **Duration** argument.

FOR...NEXT Loop Repetition Time – It's really 1/44th of a Second, not 1/50th

1/50th of a second is a rough approximation of the loop repetition. 1/44th of a second is a much closer approximation. Consider how much time each element of the **FOR...NEXT** loop takes to execute. The command **PULSOUT 14, 750** is in the middle of the range of possible pulse durations, so it can be the benchmark for average pulse duration. It sends a pulse that lasts $750 \times 2 \mu\text{s} = 1500 \mu\text{s} = 1.5 \text{ ms}$. The **PAUSE 20** command makes the program delay for 20 ms. A **FOR...NEXT** loop with a **PULSOUT** and **PAUSE** command takes about 1.3 ms to process all the numbers and commands. Although this means that the low signal between pulses really lasts for 21.3 ms instead of 20 ms, this does not affect the servo's performance. The low times can be a few ms off, it's just the high pulse durations that have to be precise, and the **PULSOUT** command is very precise.

So, the total time the **FOR...NEXT** loop takes to repeat is 1.5 ms + 20 ms + 1.3 ms = 22.8 ms, which is 22.8 thousandths of a second. So, how many 22.8-thousandths-of-a-second fit into 1-second? Let's divide 0.0228 into 1 and find out:

$$1 \text{ second} \div 0.0228 \text{ seconds/repetition} \approx 43.86 \text{ repetitions} \\ \approx 44 \text{ repetitions}$$

So that's why the loop repeats at a rate of about 44 repetitions per second. The number of repetitions in 1 second is called a *hertz*, abbreviated Hz. So, we can say that the servo signal repeats or cycles at about 44 Hz.

4



Cycles and hertz (Hz): When a signal repeats itself a certain number of times, each repetition is called a *cycle*. The number of cycles in a second is measured in hertz. Hertz is abbreviated Hz.

Longer or shorter **PULSOUT Duration** values cause the **FOR...NEXT** loop to take a little more or less time to repeat. The **PULSOUT Duration** of 750 is right in the middle of the range of servo control pulse durations shown back in Figure 4-18 on page 109. So, you can use 44 Hz as a benchmark for the number of servo pulses in a second for your code. If you need to be more precise, just repeat the math for the **PULSOUT** command you are using. For example, if the loop has a **PULSOUT** command with a **Duration** of 1000 instead of 750, it takes 2 ms for the pulse instead of 1.5 ms. The loop still has a pause of 20 ms and 1.3 ms of processing time. So that adds up to 2 + 20 + 1.3 ms = 23.3 ms. Divide that in to 1 second to find out the **FOR...NEXT** loop's rate, and we get $1 \div 0.0233 \approx 42.9 \approx 43$ Hz.

FOR...NEXT Loop Servo Control Summary

Figure 4-23 shows the part each number in a **FOR...NEXT** loop plays in servo control. The **FOR...NEXT** loop's **EndValue** determines the number of 44ths of a second the servo holds a position. The **PULSOUT** command's **Duration** argument tells the servo what position to hold. The value 750 sends a 1.5 ms pulse, which instructs the servo to hold a 90° position according to Figure 4-18 on page 108. The **PULSOUT** command's **Pin** argument chooses the I/O pin for sending servo control signals. So, 14 makes the **PULSOUT** command send its brief high signal (pulse) to the servo connected to I/O pin P14. When the pulse ends, it leaves the I/O pin sending a low signal. Then, the **PAUSE 20** command ensures that the low signal lasts for approximately 20 ms before the next pulse.

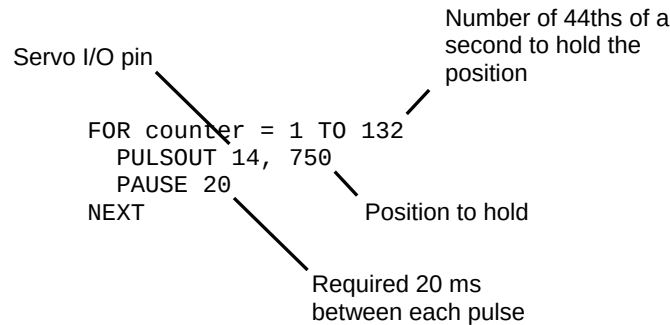


Figure 4-23
Servo Control
For...Next Loop

On the average, a **FOR...NEXT** loop that sends a single **PULSOUT** command to a servo, followed by **PAUSE 20**, repeats at about 44 times per second. Since this loop repeats 132 times, it makes the servo hold the 135° position for about 3 seconds. That's because:

$$132 \text{ repetitions} \div 44 \text{ repetitions/second} = 3 \text{ seconds}$$

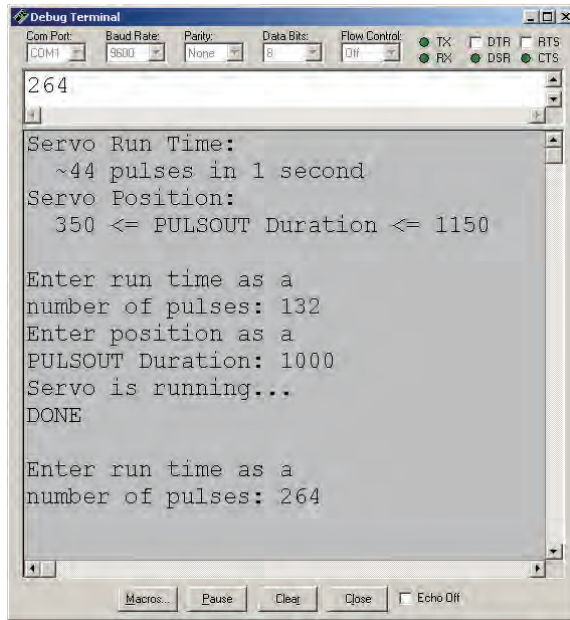
If your application or project needs to make the BASIC Stamp send a servo signal for a certain number of seconds, just multiply the number of seconds by 44, and use the result in your **FOR...NEXT** loop's **EndValue** argument. For example, if your signal needs to last five seconds:

$$5 \text{ seconds} \times 44 \text{ repetitions/second} = 220 \text{ repetitions}$$

ACTIVITY #4: 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, and then sends position information back to the microcontroller. The microcontroller might then update a conveyer 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 4-24. 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 4-24**

Sending Messages to the BASIC Stamp

4

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, and then use these values to control the servo:

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 #2

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 **DEBUGIN**, 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 variable 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 Enter key, 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 4-25 shows the locations of the Debug Terminal's Transmit windowpane along with its Receive windowpane. The Receive windowpane is the one we've been using all along to display messages that the Debug Terminal "receives" from the BASIC Stamp. The Transmit windowpane allows you to type in characters and numbers and "transmit" them to the BASIC Stamp.

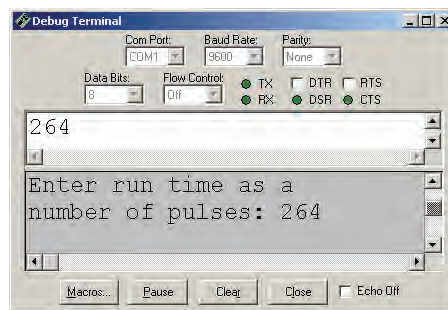


Figure 4-25
Debug Terminal's Windowpanes

← Transmit windowpane

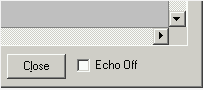
← Receive windowpane

In Figure 4-25, the number 264 is typed into the Debug Terminal's Transmit windowpane. Below, in the Receive windowpane, a copy of the 264 value is shown next

to the “Enter Run time...” message. This copy is called an *echo*, and it only displays in the Receive windowpane if the Echo Off checkbox is left unchecked.



Echo is when you send a message through the Debug Terminal's Transmit windowpane, and a copy of that message appears in the Debug Terminal's Receive windowpane. There is an Echo Off checkbox in the lower right corner of the Debug Terminal, and you can click it to toggle the checkmark. For this activity, we want to display the echoes in the Receive windowpane, so the Echo Off checkbox should be **unchecked**.



4

- ✓ 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 run time as a” in Figure 4-25.
- ✓ Make sure the Echo Off checkbox in the lower-right corner is unchecked.
- ✓ Click the upper, Transmit windowpane to place your cursor there for typing messages.
- ✓ When the Debug Terminal prompts you to, “Enter run time as a number of pulses:” type the number 132, then press your computer keyboard’s Enter key.
- ✓ When the Debug Terminal prompts you to “Enter position as a **PULSOUT** duration:” type the number 1000, then press Enter.



The PULSOUT Duration should be a number between 350 and 1150. If you enter numbers outside that range, the program will change it to the closest number within that range, either 350 or 1150. If the program did not have this safety feature, certain numbers could be entered that would make the servo 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 350 and 1150 for your **PULSOUT** value.

- ✓ Experiment with entering other values between 350 and 1150 for the **PULSOUT Duration** and values between 1 and 65534 for the number of pulses.



It takes about 44 pulses to make the servo hold a position for 1 second. So, to make the servo hold a position for about 5 minutes, you could enter 13200 at the "number of pulses" prompt. That's $44 \text{ pulses/second} \times 60 \text{ seconds/minute} \times 5 \text{ minutes} = 13,200 \text{ pulses}$.

Why use values from 1 to 64434? If you really want to know, read all the way through the **FOR...NEXT** section in the BASIC Stamp Manual to learn about the *16-bit rollover*, or *variable range*, error. It can cause a bug when you are making your own programs!

```
' 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

PAUSE 1000
DEBUG CLS, "Servo Run Time:", CR,
" ~44 pulses in 1 second", CR,
"Servo Position:", CR,
" 350 <= PULSOUT Duration <= 1150", CR, CR

DO

  DEBUG "Enter run time as a ", CR,
    "number of pulses: "
  DEBUGIN DEC pulses

  DEBUG "Enter position as a", CR,
    "PULSOUT Duration: "
  DEBUGIN DEC duration

  duration = duration MIN 350 MAX 1150

  DEBUG "Servo is running...", CR

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

  DEBUG "DONE", CR, CR
  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 of each pulse for the **PULSOUT** command.

A **DEBUG** command provides a reminder that there are about 44 pulses in 1 second in the **FOR...NEXT** loop, and that the **PULSOUT Duration** argument that controls servo position can be a value between 350 and 1150.

```
DEBUG CLS, "Servo Run Time:", CR,
        " ~44 pulses in 1 second", CR,
        "Servo Position:", CR,
        " 350 <= PULSOUT Duration <= 1150", CR, CR
```

The rest of the program is nested inside a **DO...LOOP** without a **WHILE** or **UNTIL Condition** 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 another value you enter into the **duration** variable too.

```
DEBUG "Enter run time as a ", CR,
      "number of pulses: "
DEBUGIN DEC pulses
```

```

DEBUG "Enter position as a", CR,
      "PULSOUT Duration: "
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 during that time:

```

DEBUG "Servo is running...", CR

```

While the servo is running, you can try to gently 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 was used to keep track of the **FOR . . .NEXT** loop's repetitions. The **counter** variable aside, this **FOR . . .NEXT** loop introduces some new techniques for using variables to define how the program (and the servo) behaves. Up until this example, the **FOR . . .NEXT** loops have used constants such as 10 or 132 in the loop's **EndValue** argument. In this **FOR . . .NEXT** loop, the value of the **pulses** variable is used to control the **FOR . . .NEXT** loop's **EndValue**. So, you set the value of **pulses** by entering a number into the Debug Terminal, and it controls the number of repetitions the **FOR . . .NEXT** loop makes, which in turn controls the time the servo holds a given position.

```

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

```

Also, in previous examples, constant values such as 500, 750, and 1000 were used for the **PULSOUT** command's **Duration** argument. In this loop, a variable named **duration**, which you set by entering values into the Debug Terminal's Transmit windowpane, now defines the **PULSOUT** command's pulse duration, which in turn controls the position the servo holds.



Take some time to understand the FOR...NEXT loop in ServoControlWithDebug.bs2.

It is one of the first examples of the amazing things you can do with variables in PBASIC command arguments and loops, and it also highlights how useful a programmable microcontroller module like the BASIC Stamp can be.

Your Turn – Setting Limits in Software

4

Let's imagine that this computer servo control system is one that has been developed for remote-control. 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 that could damage the equipment.

While running `ServoControlWithDebug.bs2`, it is possible to make a mistake while typing the **duration** value into the Debug Terminal. Let's say you accidentally typed 100 instead of 1000 and pressed Enter. The value 100 would cause the servo to try to turn to a position beyond its mechanical limits. Although it won't instantly break the servo, it's certainly not good for the servo or its useful lifespan. So the program has a line that prevents this mistake from doing any damage:

```
duration = duration MIN 350 MAX 1150
```

This command would correct the 100 accident by changing the **duration** variable to 350. Likewise, if you accidentally typed 10000, it would reduce the **duration** variable to 1150. You could do something equivalent with a couple of **IF...THEN** statements:

```
IF duration < 350 THEN duration = 350
IF duration > 1150 THEN duration = 1150
```

There are some machines where even automatically correcting to the nearest value could have undesirable results. For example, if you are a computer controlling a machine that cuts some sort of expensive material, you wouldn't necessarily want the machine to just assume you meant 350 when you tried to type 1000, but accidentally typed 100. If it just cut the material at the 350 setting, it could turn out to be an expensive mistake. So, another approach your program can take is to simply tell you that your value was out of range, and to try again. Here is an example of how you can modify the code to do this:

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

```
DEBUG "Enter position as a", CR,
      "PULSOUT Duration: "
DEBUGIN DEC duration
```

...with this code block:

```
DO
  DEBUG "Enter position as a", CR,
        "PULSOUT Duration: "
  DEBUGIN DEC duration
  IF duration < 350 THEN
    DEBUG "Value of duration must be at least 350", CR
    PAUSE 1000
  ENDIF
  IF duration > 1150 THEN
    DEBUG "Value of duration cannot be more than 1150", CR
    PAUSE 1000
  ENDIF
LOOP UNTIL duration >= 350 AND duration <= 1150
```

- ✓ Save the program.
- ✓ Run the program and verify that it repeats until you enter a value in the correct 350 to 1150 range.

ACTIVITY #5: 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 a 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 45° and then rotate slowly counterclockwise until it gets to 135°. 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 **counter** variable changes, so does the position of the servo's horn.

FOR...NEXT loops have an optional **STEP StepValue** argument. The *StepValue* argument can be used to make the servo rotate faster. For example, you can use the *StepValue* 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 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 down to 500:

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

You can combine counting down with a *StepValue* 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 ServoVelocities.bs2.
- ✓ 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 horn'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

PAUSE 1000

DO
  DEBUG "Pulse width increment by 8", CR

  FOR counter = 500 TO 1000 STEP 8
    PULSOUT 14, counter
    PAUSE 7
    DEBUG DEC5 counter, CR, CRSRUP
  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 eight 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. Since the 20 ms low time between servo pulses doesn't need to be precise, it's okay to tune and adjust it.



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 that, the control character **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.

4

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 **StepValue** 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 **StepValue** value.

ACTIVITY #6: 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 servo control.
- Program the BASIC Stamp to control the servo based on the pushbutton inputs.

When you are done, you will be able to press and hold one button to get the BASIC Stamp to rotate the servo in one direction, and press and hold the other 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 last.

Extra Parts for Pushbutton Servo Control

The same parts from the previous activities in this chapter are still used. In addition, 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 4-26 shows the pushbutton circuits that you will use to control the servo.

- ✓ 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:
 - o Figure 4-27 if you are using a Board of Education USB (any Rev) or Serial (Rev C or newer).
 - o Figure 4-28 if you are using a BASIC Stamp HomeWork Board (Rev C or newer).
- ✓ If your board is not listed above, refer to the Servo Circuit Connections download at go to www.parallax.com/Go/WAM to find circuit instructions for your board.

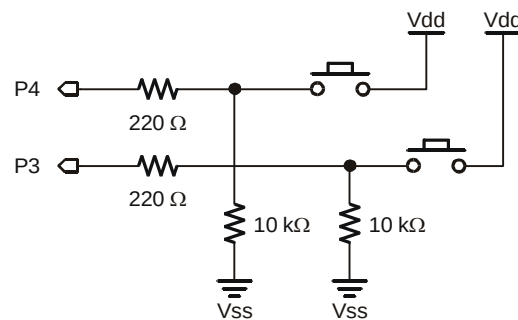
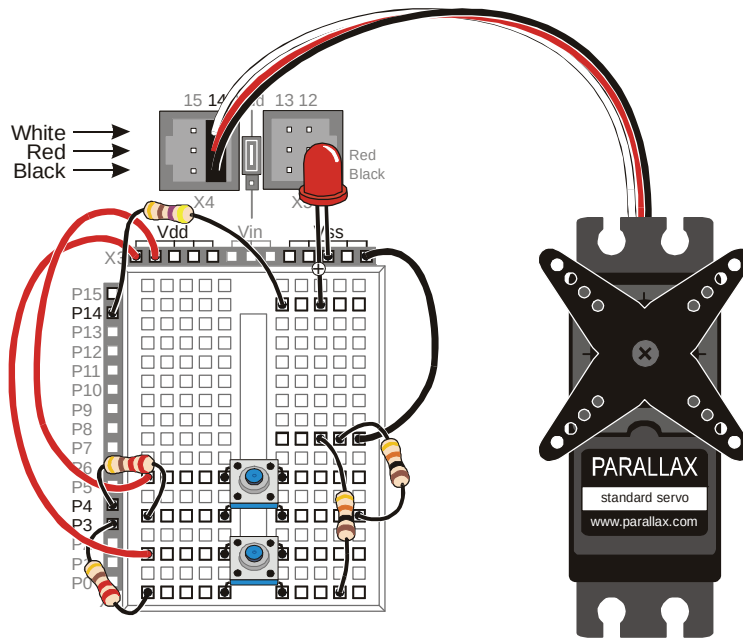


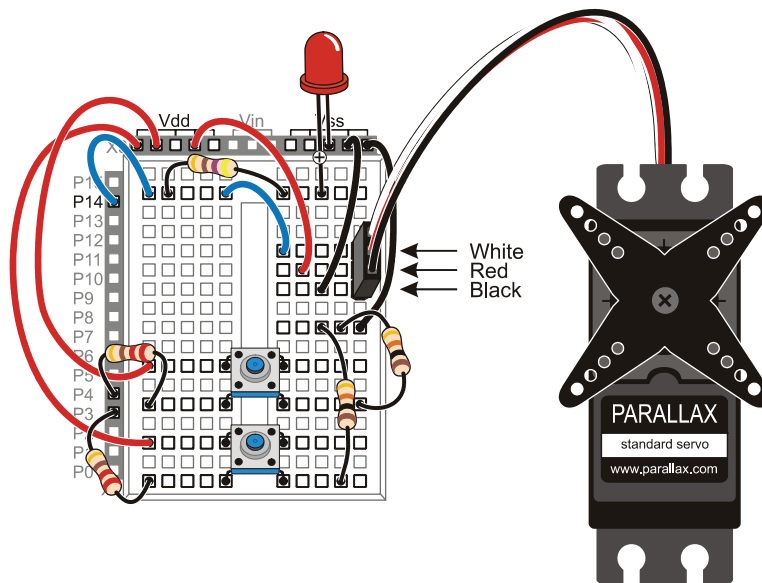
Figure 4-26
Pushbutton
Circuits for Servo
Control

**Figure 4-27**

Board of
Education Servo
Circuit with
Pushbutton
Circuits Added

*For the Board of
Education Serial
Rev C or higher,
or USB of any
revision*

4

**Figure 4-28**

HomeWork
Board Servo
Circuit with
Pushbutton
Circuits Added

*For the
HomeWork
Board Rev C or
higher*

- ✓ 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 67.
- ✓ Modify the program so that it reads P4.
- ✓ Run the modified program to test the pushbutton connected to P4.

Programming Pushbutton Servo Control

IF...THEN code blocks can be used to check pushbutton states and either add to or subtract from a variable named **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
PAUSE 1000

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 – Mechanical Limits vs. Software Limits

The servo's mechanical stoppers prevent the servo from turning beyond about 0° and 180°, which corresponds to **PULSOUT Duration** arguments in the 250 and 1250 neighborhoods. ServoControlWithPushbuttons.bs2 also has software limits, imposed by **IF...THEN** statements that prevent you from using a pushbutton to turn the servo beyond a certain point. In contrast to the mechanical limits, the software limits are very easy to adjust. For example, you can give your pushbutton controlled servo a wider range of motion by simply replacing every instance of 500 with 350, and every instance of 1000 with 1150. Or, you could give your servo a narrower range of motion by replacing instances of 500 with 650 and instances of 1000 with 850. The software limits don't even need to be symmetrical. For example, you could change the software limits from the 500–1000 range to the 350–750 range.

- ✓ Experiment with different software servo limits, including 350 to 1150, 650 to 850, and 350 to 750.

- ✓ Test each set of software limits to make sure they perform as expected.

You can also change how quickly the servo turns as you hold a button down. For example, if you change the two 25 values in the program to 50, the servo will respond twice as quickly. Alternately, you could change them to 30 to make the servo respond just a little faster, or to 20 to make them respond a little slower, or to 10 to make it respond a lot slower.

- ✓ Try it!

SUMMARY

This chapter introduced microcontrolled motion using a Parallax Standard 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 0.5 and 2.5 ms, and they have to be delivered roughly 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 or constants can be used to determine both the number of loop repetitions and the **PULSOUT** command's *Duration* argument.

In this chapter, several ways to get 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 it is used as the *Counter* argument of 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 when 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 five 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?

4

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, the Debug Terminal should not accept any commands, and 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) Screw – attaches servo's horn to the output shaft; 5) 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. Type messages into the Debug Terminal's Transmit windowpane. Use the **DEBUGIN** command and a variable to make the BASIC Stamp receive the characters.

Q6. Either a nested **IF...THEN** statement or a command that uses the **MAX** and **MIN** operators to keep the variable in certain ranges.

E1.

a) Increments of 1

```
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...NEXT** 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. Assume a variable named **pulses** has been declared:

```
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; two are given here.

```
' What's a Microcontroller - Ch04Prj01Soln1__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

```

Below is a version that can even detect button presses while it's sending a signal to the servo. This is important for machinery that needs to STOP IMMEDIATELY when the kill switch is pressed. It utilizes the waiting technique that was introduced in the Reaction Timer game in Chapter 3, Activity #5 in three different places in the program. You can verify that the program stops sending a control signal to the servo by monitoring the LED signal indicator light connected to P14.

```

' 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.

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

counter      VAR      Word
pulses       VAR      Word
duration     VAR      Word

```

```

PAUSE 1000

DEBUG "Press Start switch (P4) to start machinery.", CR

DO:LOOP UNTIL IN4 = 1
DEBUG "Press Kill switch (P3) to stop machinery.", CR

DEBUG CR, CR, "Servo Run Time:", CR,
" ~44 pulses in 1 second", CR,
"Servo Position:", CR,
" 350 <= PULSOUT Duration <= 1150", CR, CR

DO

  IF IN3 = 1 THEN
    DEBUG "Press Start switch (P4) to start machinery.", CR
    DO:LOOP UNTIL IN4 = 1
    DEBUG "Press Kill switch (P3) to stop machinery.", CR
  ENDIF
  DEBUG "Enter run time as a ", CR,
    "number of pulses: "
  DEBUGIN DEC pulses

  DEBUG "Enter position as a", CR,
    "PULSOUT Duration: "
  DEBUGIN DEC duration

  duration = duration MIN 350 MAX 1150

  DEBUG "Servo is running...", CR

  FOR counter = 1 TO pulses
    PULSOUT 14, duration
    PAUSE 20
    IF IN3 = 1 THEN
      DEBUG "Press Start switch (P4) to start machinery.", CR
      DO:LOOP UNTIL IN4 = 1
      DEBUG "Press Kill switch (P3) to stop machinery.", CR
    ENDIF
  NEXT

  DEBUG "DONE", CR, CR
  PAUSE 1000

LOOP

```

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 lights get brighter; twist the dial in the other direction, and the lights get 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 many 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.

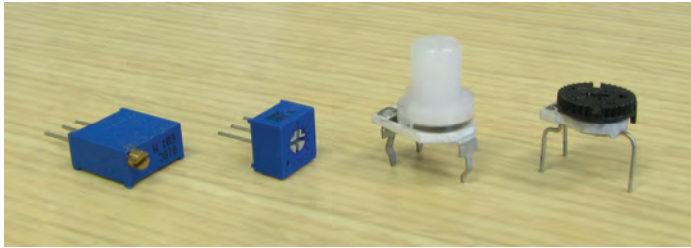


Figure 5-2
A Few Potentiometer
Examples

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.

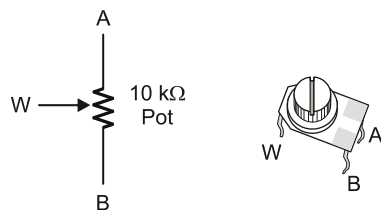


Figure 5-3
Potentiometer Schematic Symbol
and Part Drawing

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.

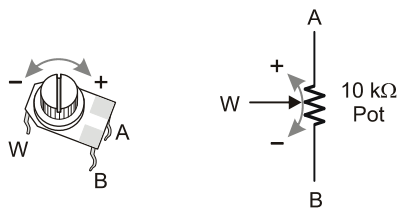


Figure 5-4
Adjusting the Potentiometer's Wiper
Terminal

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 – red
- (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: If you have trouble keeping the potentiometer seated in the breadboard sockets, check its legs. If each one has a small bend, use a needle-nose pliers to straighten them out and then try plugging the pot into the breadboard again. When the pot's legs are straight, they may maintain better contact with the breadboard sockets.

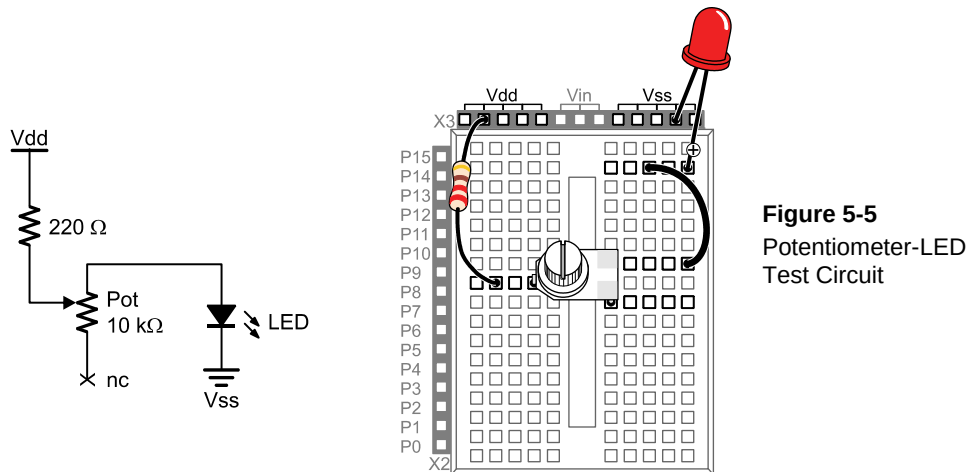


Figure 5-5
Potentiometer-LED
Test Circuit

Testing the Potentiometer Circuit

- ✓ Turn the potentiometer clockwise until it reaches its mechanical limit shown in Figure 5-6 (a).



Press the pot against the breadboard a little as you turn its knob. For these activities, the potentiometer needs to be firmly seated in the breadboard sockets. If you're not careful when you turn the knob, the pot can become disconnected from the breadboard sockets, and that can lead to incorrect measurements. So, apply a little downward pressure as you turn the potentiometer's knob to keep it seated in the breadboard.

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.

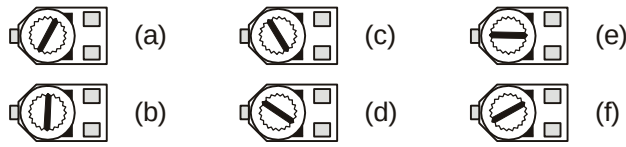


Figure 5-6
Potentiometer Knob

(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. The resistance between the A and W terminals increases as the knob is adjusted further clockwise, which in turn reduces the current through the LED, making it dimmer.

5

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.

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 this particular capacitor is called a *canister*. This type of capacitor, called an *electrolytic capacitor*, must be handled carefully.

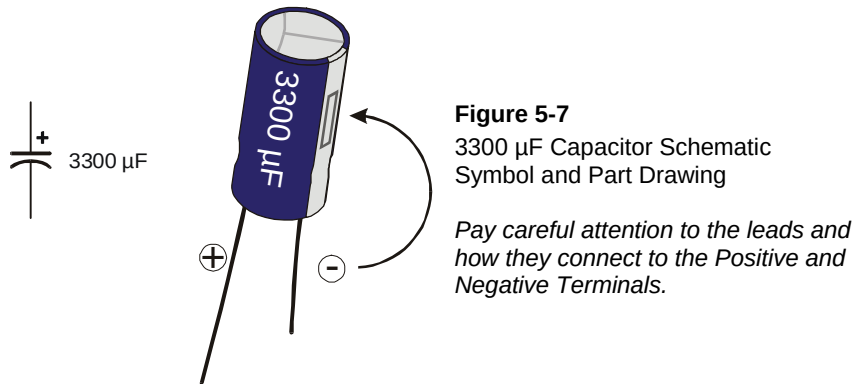
- ✓ Read the CAUTION box on the next page.



CAUTION: 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.

CAUTION: Do not apply more voltage to an electrolytic capacitor than it is rated to handle. The voltage rating is printed on the side of the canister.

CAUTION: Safety goggles or safety glasses are recommended.



Resistance and Time Circuit Parts

- (1) Capacitor – 3300 μF
- (1) Capacitor – 1000 μF
- (1) Resistor – 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)

Building and Testing the Resistor-Capacitor (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 .

- ✓ Read the SAFETY box carefully.

SAFETY

Always observe polarity when connecting the 3300 or 1000 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. Vss is the lowest voltage (0 V) on the Board of Education and BASIC Stamp HomeWork Board. By connecting the capacitor's negative terminal to Vss, you ensure that the polarity across the capacitor's terminals will always be correct.

Never apply voltage to the capacitor that exceeds the voltage rating on the canister.

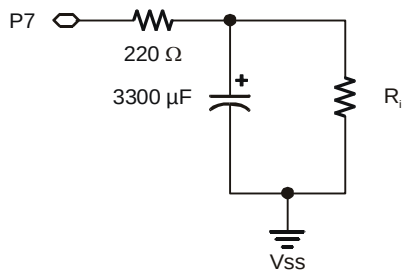
Wear safety goggles or safety glasses during this activity.

Always disconnect power before you build or modify circuits.

Keep your hands and face away from this capacitor when power is connected.

5

- ✓ With power disconnected, build the circuit as shown starting with a 470 Ω resistor in place of the resistor labeled R_i .



$R_1 = 470 \Omega$
 $R_2 = 1 \text{ k}\Omega$
 $R_3 = 2 \text{ k}\Omega$
 $R_4 = 10 \text{ k}\Omega$

Figure 5-8

Schematic for Testing
RC-time Voltage Decay

*The four different
resistors will be used one
at a time as R_i in the
schematic.*

Four different resistors will be used as R_i shown in the schematic. First, the schematic will be built and tested with $R_i = 470 \Omega$, and then $R_i = 1 \text{ k}\Omega$, etc. will be used later.

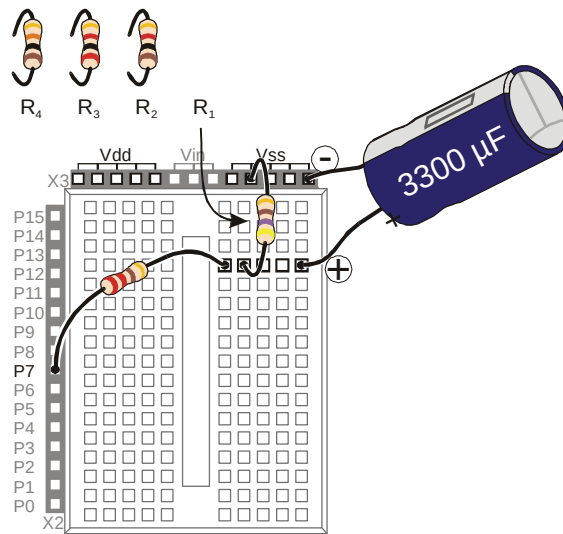


Figure 5-9
Wiring Diagram for
Viewing RC-time Voltage
Decay

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.

- ✓ 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 consistent 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 146, 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 μ F	
Resistance (Ω)	Measured Time (s)
470	
1 k	
2 k	
10 k	

5

```
' 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
PAUSE 1000

DEBUG CLS

HIGH 7
DEBUG "Capacitor Charging...", CR

FOR counter = 5 TO 0
    PAUSE 1000
    DEBUG DEC2 counter, CR, CRSRUP
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 3.4 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, CR$RUP
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 V_{dd}, and a low signal was considered V_{ss}. To the BASIC Stamp, actually a high signal is any voltage above about 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 to 1.4 V.

```
DO

  PAUSE 100
  timeCounter = timeCounter + 1

  DEBUG ? IN7
  DEBUG DEC5 timeCounter, CR, CRSRUP, CRSRUP

LOOP UNTIL IN7 = 0
```

5

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. Later on in the next activity, you will use a capacitor that has 1/33,000 the capacity! 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 μ F	
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 be in the microseconds range. 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 which in turn indicates the knob's position.

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 μ F
- (1) Capacitor – 0.01 μ F
- (2) Jumper wires



These capacitors do not have + and – terminals. They are non-polar. So, you can safely connect these capacitors to a circuit without worrying about positive and negative terminals.



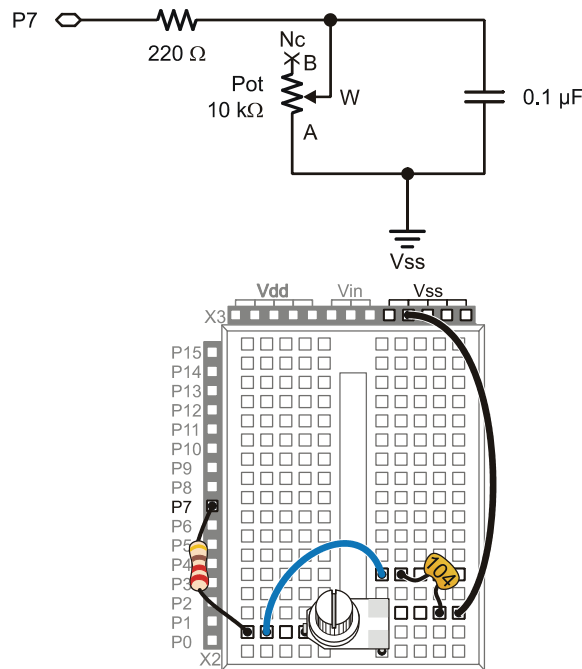
Figure 5-10
Ceramic Capacitors

0.1 μ F capacitor (left)
0.01 μ F capacitor (right)

Building an RC Time Circuit for the BASIC Stamp

Figure 5-11 shows a schematic and wiring diagram for the fast RC-time circuit. This is the circuit that you will use to monitor the position of the potentiometer's knob with the help of the BASIC Stamp and a PBASIC program.

- ✓ Build the circuit shown in Figure 5-11.



5

Figure 5-11
Schematic and wiring
diagram for BASIC
Stamp RCTIME Circuit
with Potentiometer

Programming RC-Time Measurements

The example program in Activity #2 measured 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 to 1.4 V. The result when the program was done polling was that the **timeCounter** variable stored the number of tenths of a second it took for the capacitor's voltage to decay to 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 μ 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 units 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 μF to 0.1 or even 0.01 μF , and still get time measurements that indicate 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 corresponds to the position of the potentiometer's knob.

Example Program: ReadPotWithRcTime.bs2

- ✓ Enter and run ReadPotWithRcTime.bs2
- ✓ Try rotating the potentiometer's knob while monitoring the value of the **time** variable using the Debug Terminal.



Remember to apply a little downward pressure to keep the potentiometer seated on the breadboard as you twist its knob. If your servo starts twitching back and forth unexpectedly instead of holding its position, an un-seated pot may be the culprit.

```
' What's a Microcontroller - ReadPotWithRcTime.bs2
' Read potentiometer in RC-time circuit using RCTIME command.

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

time VAR Word

PAUSE 1000

DO

  HIGH 7
  PAUSE 100
  RCTIME 7, 1, time
  DEBUG HOME, "time = ", DEC5 time

LOOP
```

Your Turn – Changing Time by Changing the Capacitor

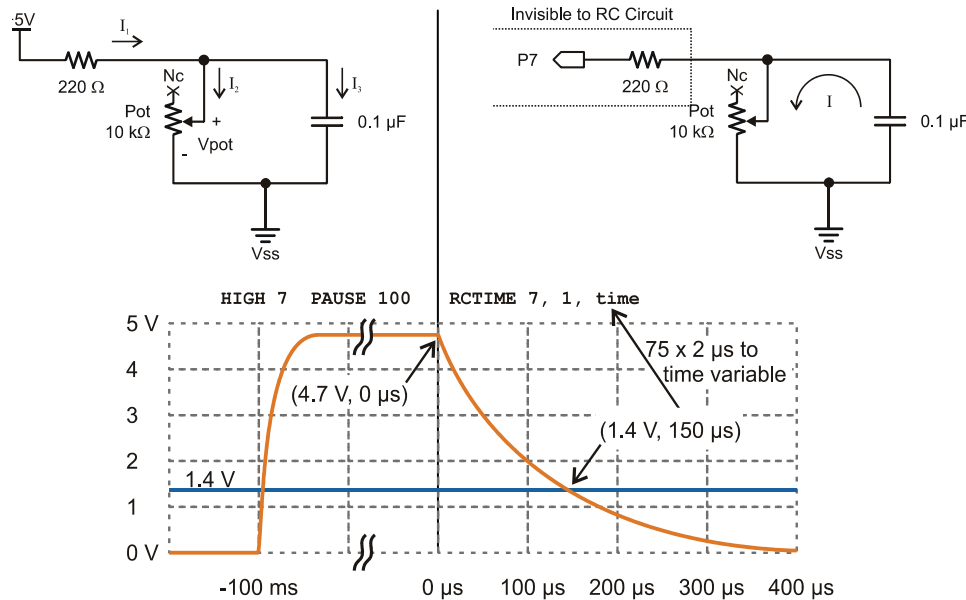
- ✓ Replace the 0.1 μF capacitor with a 0.01 μF capacitor.
- ✓ Try the same positions on the potentiometer that you did in the main activity and compare the value displayed in the Debug Terminal with the values obtained for the 0.1 μF capacitor. Are the **RCTIME** measurements about one tenth the value for a given potentiometer position?

- ✓ Go back to the 0.1 μF capacitor.
- ✓ With the 0.1 μF capacitor back in the circuit and the 0.01 μF capacitor removed, turn the pot's knob to its limit in both directions and make notes of the highest and lowest values for the next activity. Highest:_____ Lowest:_____

How ReadPotWithRcTime.bs2 Works

Figure 5-12 shows how the ReadPotWithRcTime.bs2's **HIGH**, **PAUSE** and **RCTIME** commands interact with the circuit in Figure 5-11.

Figure 5-12: Voltage at P7 through HIGH, PAUSE, and RCTIME



On the left, the **HIGH 7** command causes the BASIC Stamp to internally connect its I/O pin P7 to the 5 V supply (Vdd). Current from the supply flows through the potentiometer's resistor and also charges the capacitor. The closer the capacitor gets to its final charge (almost 5 V), the less current flows into it. The **PAUSE 100** command is primarily to make the Debug Terminal display update at about 10 times per second; **PAUSE 1** is usually sufficient to charge the capacitor. On the right, the **RCTIME 7, 1, time** command changes the I/O pin direction from output to input and starts counting time in 2 μs increments. As an input the I/O pin no longer supplies the circuit with 5 V.

In fact, as an input, it's pretty much invisible to the RC circuit. So, the capacitor starts losing its charge through the potentiometer. As the capacitor loses its charge, its voltage decays. The **RCTIME** command keeps counting time until P7 senses a low signal, meaning the voltage across the capacitor has decayed to 1.4 V, at which point it stores its measurement in the **time** variable.

Figure 5-12 also shows a graph of the voltage across the capacitor during the **HIGH**, **PAUSE**, and **RCTIME** commands. In response to the **HIGH 7** command, which connects the circuit to 5 V, the capacitor quickly charges. Then, it remains level at its final voltage during most of the **PAUSE 100** command. When the program gets to the **RCTIME 7, 1, time** command, it changes the I/O pin direction to input, so the capacitor starts to discharge through the potentiometer. As the capacitor discharges, the voltage at P7 decays. When the voltage decays to 1.4 V (at the 150 μ s mark in this example), the **RCTIME** command stops counting time and stores the measurement result in the **time** variable. Since the **RCTIME** command counts time in 2 μ s units, the result for 150 μ s that gets stored in the **time** variable is 75.



I/O Pin Logic Threshold: 1.4 V is a BASIC Stamp 2 I/O pin's logic threshold. When the I/O pin is set to input, it stores a 1 in its input register if the voltage applied is above 1.4 V or a 0 if the input voltage is 1.4 V or below. The first pushbutton example back in Chapter 3, Activity #2 applied either 5 V or 0 V to P3. Since 5 V is above 1.4 V, **IN3** stored a 1, and since 0 V is below 1.4 V, **IN3** stored a 0.

RCTIME State Argument: In ReadPotWithRcTime.bs2, the voltage across the capacitor decays from almost 5 V, and when it gets to 1.4 V, the value in the **IN7** register changes from 1 to 0. At that point, the **RCTIME** command stores its measurement in its **Duration**, which is the **time** variable in the example program. The **RCTIME** command's **State** argument is 1 in **RCTIME 7, 1, time**, which tells the **RCTIME** command that the **IN7** register will store a 1 when the measurement starts. The **RCTIME** command measures how long it takes for the **IN7** register to change to the opposite state, which happens when the voltage decays below the I/O pin's 1.4 V logic threshold.

For more information: Look up the **RCTIME** command in either the BASIC Stamp Manual or the BASIC Stamp Editor's Help.

Figure 5-13 shows how the decay time changes with the potentiometer's resistance for the circuit in Figure 5-11. Each position of the potentiometer's knob sets it at a certain resistance. Turn it further one direction, and the resistance increases, and in the other direction, the resistance decreases. When the resistance is larger, the decay takes a longer time, and the **RCTIME** command stores a larger value in the **time** variable. When the resistance is smaller, the decay takes a shorter time, and the **RCTIME** command stores a

smaller value in the **time** variable. The **DEBUG** command in ReadPotWithRcTime.bs2 displays this time measurement in the Debug Terminal, and since the decay time changes with the potentiometer's resistance, which in turn changes with the potentiometer knob's position, the number in the Debug Terminal indicates the knob's position.

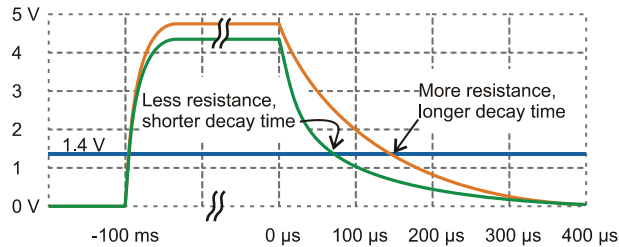


Figure 5-13
How Potentiometer Resistance
Affects Decay Time

5

Why does the capacitor charge to a lower voltage when the potentiometer has less resistance?

Take a look at the schematic in the upper-left corner of Figure 5-12 on page 153. Without the 220 Ω resistor, the I/O pin would be able to charge the capacitor to 5 V, but the 220 Ω resistor is necessary to prevent possible I/O pin damage from a current inrush when it starts charging the capacitor. It also prevents the potentiometer from drawing too much current if it is turned to 0 Ω while the I/O pin sends its 5 V high signal.

With 5 V applied across the 220 Ω resistor in series with the potentiometer, the voltage between them has to be some fraction of 5 V. When two resistors conducting current are placed in series, which results in an intermediate voltage, the circuit is called a *voltage divider*. So the 220 Ω resistor and potentiometer form a voltage divider circuit, and for any given potentiometer resistance (R_{pot}), you can use this equation to calculate the voltage across the potentiometer (V_{pot}):

$$V_{pot} = 5 \text{ V} \times R_{pot} \div (R_{pot} + 220 \text{ } \Omega)$$

The value of V_{pot} sets the ceiling on the capacitor's voltage. In other words, whatever the voltage across the potentiometer would be if the capacitor wasn't connected, that's the voltage the capacitor can charge to, and no higher. For most of the potentiometer knob's range, the resistance values are in the k Ω , and when you calculate V_{pot} for k Ω R_{pot} values, the results are pretty close to 5 V. The 220 Ω resistor doesn't prevent V_{pot} from charging above 1.4 V until the potentiometer's value is down at 85.6 Ω , which is less than 1% of the potentiometer's range of motion. This 1% would have resulted in the lowest measurements anyhow, so it's difficult to tell that measurements of 1 in this range are anything out of the ordinary. Even with the additional 220 Ω resistors built into BASIC Stamp HomeWork board I/O pin connections, only the lowest 1.7% of the potentiometer's range is affected, so it's still virtually unnoticeable.

So the 220 Ω resistor protects the I/O pin, with minimal impact on the RC decay measurement's ability to tell you where you positioned the potentiometer's knob.

ACTIVITY #4: CONTROLLING A SERVO WITH A POTENTIOMETER

Thumb joysticks like the one in Figure 5-14 are commonly found in video game controllers. Each joystick typically has two potentiometers that allow the electronics inside the game controller to report the joystick's position to the video game console. One potentiometer rotates with the joystick's horizontal motion (left/right), and the other rotates with the joystick's vertical motion (forward/backward).

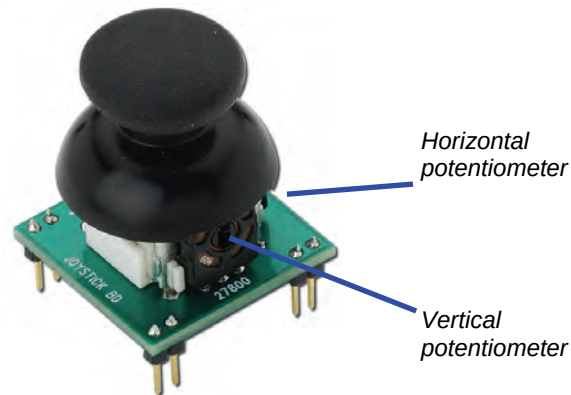


Figure 5-14
Potentiometers Inside
the Parallax Thumb
Joystick Module

Another thumb joystick application that uses potentiometers is the RC radio controller and model airplane in Figure 4-1 on page 94. The controller has two joysticks, and each has two potentiometers. Each potentiometer's position is responsible for controlling a different servo on the RC plane.

In this activity, you will use a potentiometer similar to the ones found in thumb joysticks to control a servo's position. As you turn the potentiometer's knob, the servo's horn will mirror this motion. This activity utilizes two circuits, the potentiometer circuit from Activity #3 in this chapter, and the servo circuit from Chapter 4, Activity #1. The PBASIC program featured in this chapter repeatedly measures the potentiometer's position with an **RCTIME** command, and then uses the measurement and some math to control the servo's position with a **PULSOUT** command.



The **BASIC Stamp** can measure the joystick's position. Since there are two potentiometers in each thumb joystick, each of them can replace the stand alone potentiometer in the circuits in Figure 5-11 on page 151. One **RCTIME** command can then measure the vertical potentiometer's position, and another can measure the horizontal potentiometer.

Potentiometer Controlled Servo Parts

- (1) Potentiometer – 10 k Ω
- (1) Resistor – 220 Ω (red-red-brown)
- (1) Resistor – 470 Ω (yellow-violet-brown)
- (1) Capacitor – 0.1 μ F
- (1) Parallax Standard Servo
- (1) LED – any color
- (2) Jumper wires

5

HomeWork Board users will also need:

- (1) 3-pin male-male header
- (4) Jumper wires

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 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:
 - o Page 96: Board of Education Servo Circuit
 - o Page 99: BASIC Stamp HomeWork Board Servo Circuit

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 500–1000 range for controlling 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 85 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 knob 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}

PAUSE 1000
DEBUG "Program Running!"

time          VAR      Word
```

```

DO
HIGH 7
PAUSE 10
RCTIME 7, 1, time
time = time */ 185
time = time + 500
PULSOUT 14, time
' Scale by 0.724 (X 256 for */).
' Offset by 500.
' 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 158 using your maximum and minimum values.
- ✓ Substitute your scale and offset values in ControlServoWithPot.bs2.
- ✓ Comment out **DEBUG "Program Running!"** with an apostrophe at the beginning of that line.
- ✓ 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.

Declaring Constants and Pin Directives

In larger programs, you may end up using the value of the scale factor (which was 185) and the offset (which was 500) many times in the program. Numbers like 185 and 500 in your program are called *constants* because unlike variables, their values cannot be changed while the program is running. In other words, the value remains “constant.” You can create names for these constants with **CON** directives:

```

ScaleFactor  CON  185
Offset       CON  500
delay        CON  10

```




These **CON** directives are just about always declared near the beginning of the program so that they are easy to find.

Once your constant values have been given names with **CON** directives, you can use **ScaleFactor** in place of 185 in your program and **Offset** in place of 500. For example:

```
time = time */ scaleFactor          ' Scale by 0.724.
time = time + offset                ' Offset by 500.
```

With the values we assigned to the constant names with **CON** directives, the commands are really:

```
time = time */ 185                  ' Scale by 0.724.
time = time + 500                   ' Offset by 500.
```

One important advantage to using constants is that you can change one **CON** directive, and it updates every instance of that constant name in your program. For example, if you write a large program that uses the **ScaleFactor** constant in 11 different places, one change to **Scale Factor CON...**, and all the instances of **ScaleFactor** in your program will use that updated value for the next program download. So, if you changed **ScaleFactor CON 500** to **ScaleFactor CON 510**, every command with **ScaleFactor** will use 510 instead of 500.

You can also give I/O pins names using **PIN** directives. For example, you can declare a **PIN** directive for I/O pin P7 like this:

```
RcPin    PIN 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 later change your circuit to use different I/O pins, all you have to do is change the value in your **PIN** directive, 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.



The PIN directive has an additional feature: The PBASIC compiler can detect whether the pin name is used as an input or output, and it substitutes either the I/O pin number for output, or the corresponding input register bit variable for input. For example, you could declare two pin directives, like **LedPin PIN 14** and **ButtonPin PIN 3**. Then, your code can make a statement like **IF ButtonPin = 1 THEN HIGH LedPin**. The PBASIC compiler converts this to **IF IN3 = 1 THEN HIGH 14**. The **IF ButtonPin = 1...** made a comparison, and the PBASIC compiler knows that you are using **ButtonPin** as an input. So it uses the input register bit **IN3** instead of the number 3. Likewise, the PBASIC compiler knows that **HIGH LedPin** uses the **LedPin** pin name as the constant value 14 for an output operation, so it substitutes **HIGH 14**.

Example Program: ControlServoWithPotUsingDirectives.bs2

This program works just like `ControlServoWithPot.bs2` but makes use of named constants and I/O pins.

- ✓ Enter and run `ControlServoWithPotUsingDirectives.bs2`.
- ✓ Observe how the servo responds to the potentiometer and verify that it behaves the same as `ControlServoWithPot.bs2`.

```
' What's a Microcontroller - ControlServoWithPotUsingDirectives.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}
```

```
rcPin      PIN    7           ' I/O Pin Definitions
servoPin    PIN    14
```

```
scaleFactor CON    185       ' Constant Declarations
offset       CON    500
delay        CON    10
```

```
time        VAR    Word      ' Variable Declaration
```

```
PAUSE 1000           ' Initialization
```

```

DO
    HIGH rcPin
    PAUSE delay
    RCTIME rcPin, 1, time
    time = time * scaleFactor
    time = time + offset
    PULSOUT servoPin, time
    DEBUG HOME, DEC5 time
LOOP

```

' Main Routine
' RC decay measurement

' Scale scaleFactor.
' Offset by offset.
' Send pulse to servo.
' Display adjusted time value.

Your Turn – Updating a PIN Directive

5

As mentioned earlier, if you connect the RC circuit to a different I/O pin, you can simply change the value of the **RcPin PIN** directive, and this change automatically reflects in the **HIGH RcPin** and **RCTIME RcPin, 1, time** commands.

- ✓ Save the example program under a new name.
- ✓ Change **scaleFactor** and **offset** 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 connection from I/O pin P7 to I/O pin P8.
- ✓ Modify the **rcPin** declaration so that it reads:

```
rcPin    PIN 8
```

- ✓ 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 PIN** 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 symbol 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 constant value (a number). After a constant is named, 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. **PIN** directives allow you to name I/O pins. The I/O pin name is context sensitive, so the PBASIC compiler substitutes the corresponding I/O pin number for a pin name in commands like **HIGH**, **LOW**, and **RCTIME**. If the pin name gets used in a conditional statement, it instead substitutes the corresponding input register, like **IN2**, **IN3**, etc.

5

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

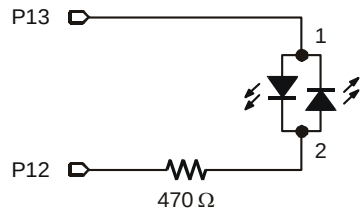
1. Let's say that you have a 0.5 μF capacitor in an RC timer circuit, and you want the measurement to take 10 times as long. Calculate the value of the new capacitor.

Projects

1. Add a bicolor LED circuit to Activity #4. Modify the example program so that the bicolor 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 first 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 x 0.5µF) = 5 µF
 P1. Activity #4 with bicolor LED added.



Potentiometer schematic from Figure 5-11 on page 151, servo from Chapter 4, Activity #1, and bicolor LED from Figure 2-19 on page 53 with P15 and P14 changed to P13 and P12 as shown.

```
' What's a Microcontroller - Ch5Prj01_ControlServoWithPot.bs2
' Read potentiometer in RC-time circuit using RCTIME command.
' The time var ranges from 126 to 713, and an offset of 330 is needed.
' Bicolor 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}

PAUSE 1000
DEBUG "Program Running!"

time          VAR    Word          ' time reading from pot
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 + 350
    IF ( time > prevTime + 2) THEN ' Scale pot, match servo range
        HIGH 13                    ' increased, pot turned CCW
        LOW 12                     ' Bicolor LED red
    ELSEIF ( time < prevTime - 2) THEN ' value decreased, pot turned CW
        LOW 13                     ' Bicolor LED green
        HIGH 12
    ELSE
        LOW 13                    ' Servo holding position
        LOW 12                    ' LED off
    ENDIF

    PULSOUT 14, time

LOOP

```

5

P2. The key is to add **IF...THEN** blocks; an example is shown below. **CLREOL** is a handy **DEBUG** control character meaning “clear to end of line.”

```

' What's a Microcontroller - Ch5Prj02_ControlServoWithPot.bs2
' Read potentiometer in RC-time circuit using RCTIME command.
' Modify with IF...THEN so the servo only rotates from 650 to 850.
' The time variable ranges from 1 to 691, so an offset of at least
' 649 is needed.

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

PAUSE 1000

DEBUG "Program Running!"

time VAR Word

DO

    HIGH 7                        ' Read pot with RCTIME
    PAUSE 10

    RCTIME 7, 1, time
    time = time + 649             ' 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
DEBUG HOME, "time = ", DEC4 time, CLREOL

LOOP
```


Chapter 6: Digital Display

THE EVERYDAY 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 with LEDs to display digits and some letters. Figure 6-2 shows a part drawing of the 7-segment LED display you will use in this chapter's activities. It also has a dot that can be used as a decimal point. Each of the segments (A through G) and the dot contain a separate LED, which can be controlled individually. Most of the pins have a number along with a label that corresponds to 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.

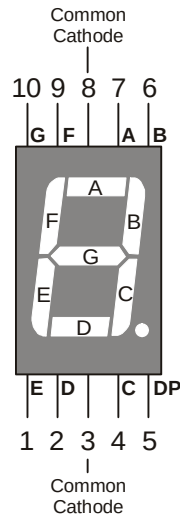


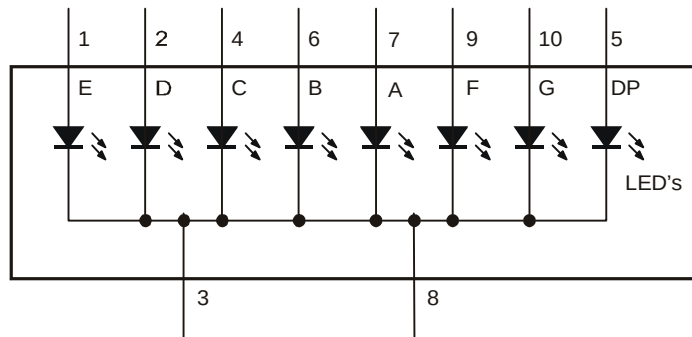
Figure 6-2
7-Segment LED Display Part
Drawing and Pin Map



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 the decimal point. Orient the part so that the decimal point is at the bottom-right. Then you can see from the pin map that Pin 1 is at the bottom-left, and the pin numbers increase counterclockwise around the 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 wires 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.



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)
(1) Jumper wire

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.

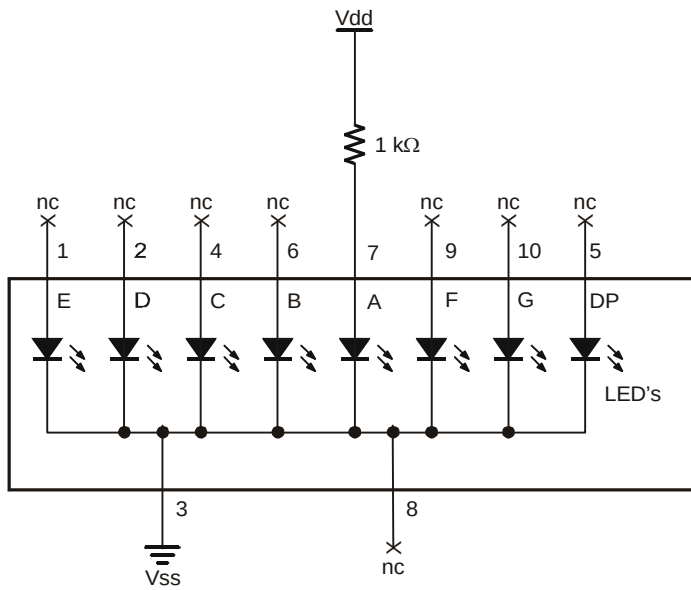


Figure 6-4
Test Circuit Schematic
for the "A" Segment
LED Display

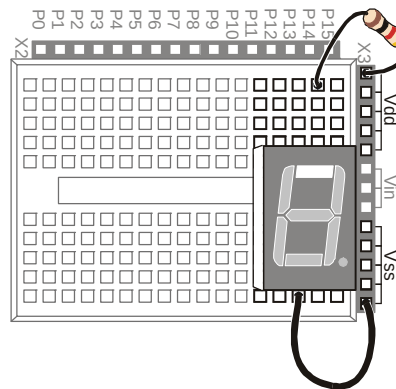


Figure 6-5
Test Circuit Wiring
Diagram for the "A"
Segment LED Display

- ✓ 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.

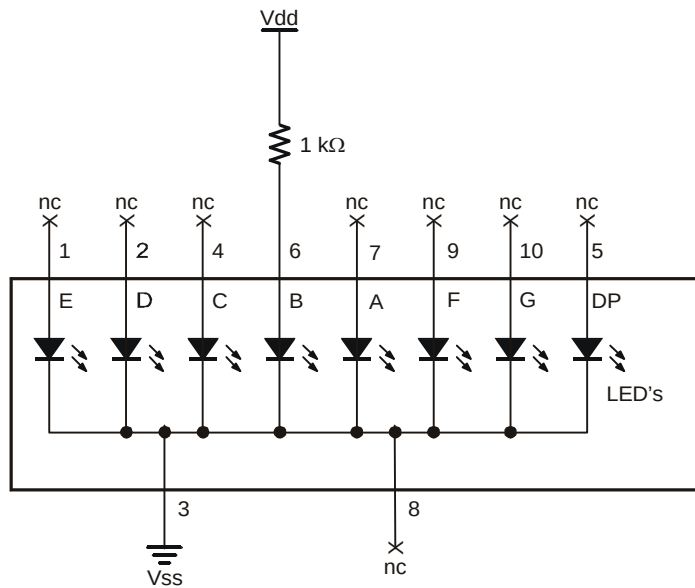


Figure 6-6
Test Circuit Schematic
for the "B" Segment
LED Display

6

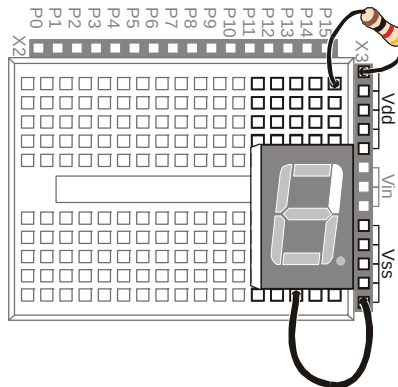


Figure 6-7
Test Circuit Wiring
Diagram for the "B"
Segment LED Display

- ✓ 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 Number 3 and the Letter H

Figure 6-8 and Figure 6-9 show the digit “3” hardwired into the 7-segment LED display.

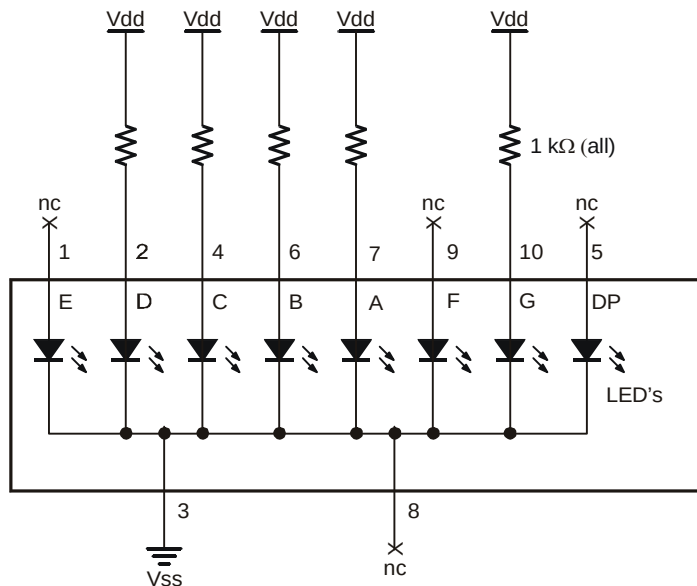


Figure 6-8
Hardwired Digit “3”

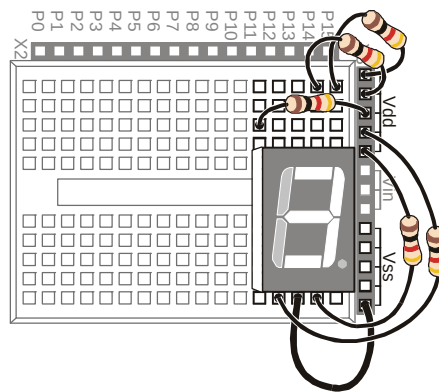


Figure 6-9
Wiring Diagram for
Figure 6-8

- ✓ 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 2 on the 7-segment LED.
- ✓ Build and test the circuit to make sure it works. Trouble-shoot if necessary.
- ✓ Determine the circuit needed for the letter “H” and then build and test it.

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 6-11 shows the schematic and Figure 6-12 shows the wiring diagram for this BASIC Stamp controlled 7-segment LED display example.

6

- ✓ Build the circuit shown in Figure 6-11 and Figure 6-12.



Schematic and pin map: If you are trying to build the circuit from the schematic in Figure 6-11 without relying on Figure 6-12, make sure to consult the 7-segment LED display's pin map, shown here again in Figure 6-10 for convenience.

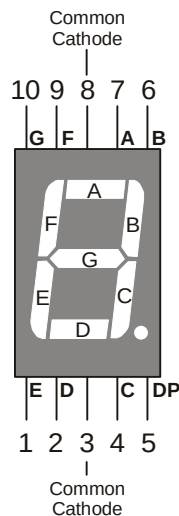


Figure 6-10
7-Segment LED Display Part
Drawing and Pin Map

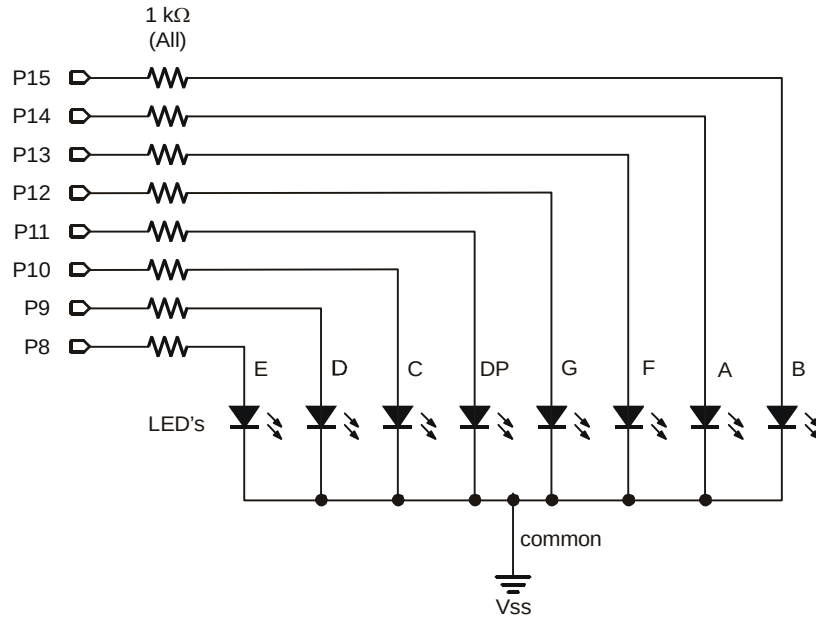


Figure 6-11
BASIC Stamp
Controlled 7-
Segment
LED Display
Schematic



Be careful with the resistors connected to P13 and P14. Look closely at the resistors connected to P13 and P14 in Figure 6-12. 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.

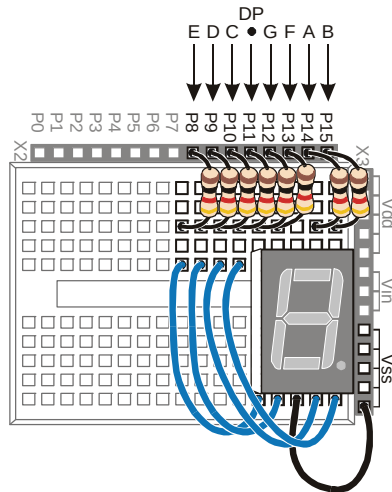


Figure 6-12
Wiring Diagram for
Figure 6-11

Use the segment letters
above this diagram as a
reference.

6



Parallel Device: The 7-segment LED display is called a *parallel device* because the BASIC Stamp has to use a group of I/O lines 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 display.

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 6-11. 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

PAUSE 1000
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 with 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

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 of these I/O pin sends. As you will soon see, **OUTH** is especially useful because you can use it to set the high/low signals for 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:

6

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.
                        ' Digit:

'      BAFG.CDE
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 6-13 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.

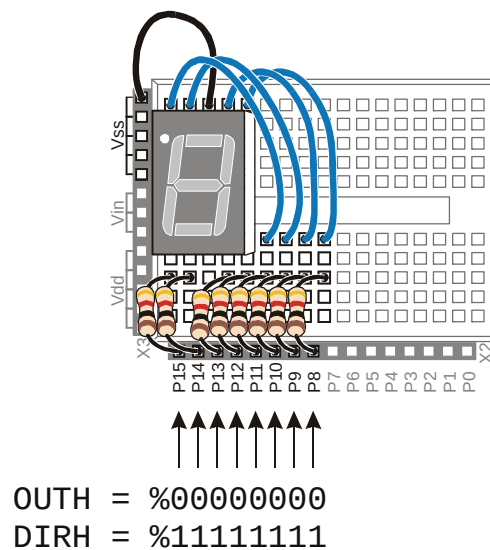


Figure 6-13
Using DIRH and OUTH
to set all I/O Pins to
Output-Low

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 a high signal, you could use **OUTH = %11111111** instead.



What does % do? The % binary formatter 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 I/O pins P8 through P15 to output. As soon as this command is executed, P8 through P15 all start sending low signals. 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 Digital Input – Pushbuttons.

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.

Always set values in a given OUT register before making them outputs with values in the corresponding DIR register. This prevents briefly sending unintended signals. For example, if **DIR5 = 1** is followed by **OUT5 = 1** at the beginning of a program, it will briefly send an unintended low signal before switching to high because **OUT5** stores 0 when the program starts. (All PBASIC variables/registers initialize to 0.) If **OUT5 = 1** is instead followed by **DIR5 = 1**, the I/O pin will send a high signal as soon as it becomes an output.

Since the values stored by all variables default to 0 when the program starts, the command **OUTH = %00000000** is actually redundant.

Figure 6-14 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
```

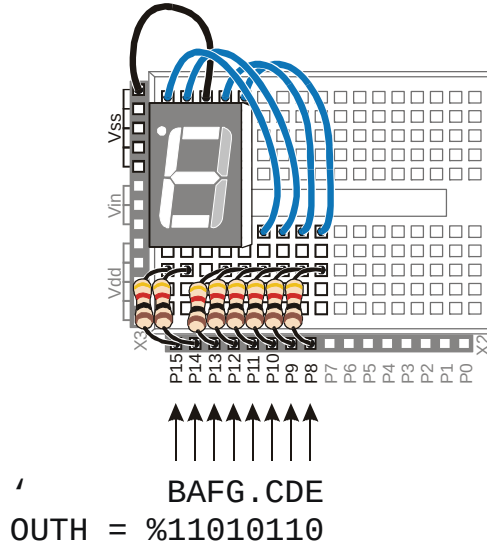


Figure 6-14
Using OUTH to Control
the High/Low Signals of
P8 – P15

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.

	Inside the HIGH and LOW commands:		
	HIGH 15	...is really the same as:	OUT15 = 1 DIR15 = 1
	Likewise, the command :		
	LOW 15	...is the same as:	OUT15 = 0 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 DisplayDigits.bs2 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, 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.

6

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 7. If **index** is 1, **value** stores 85. In the next example program, **index** is set to 2, so the **LOOKUP** command places 19 into **value**, and that’s what the Debug Terminal displays.

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
PAUSE 1000

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** variable 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
PAUSE 1000

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
```

6

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 *EndValue* argument.

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 6-15 shows a schematic of the potentiometer circuit that should be added to the project. Figure 6-16 shows a wiring diagram of the circuit from Figure 6-15 combined with the circuit from Figure 6-11 on page 176.

- ✓ Add the potentiometer circuit to the 7-segment LED display circuit as shown in Figure 6-16.

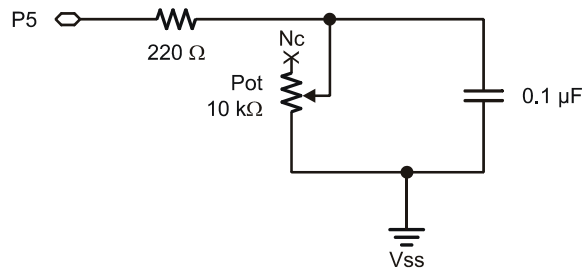


Figure 6-15
Schematic of
Potentiometer Circuit
Added to the Project

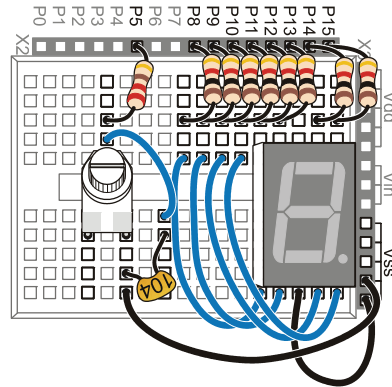


Figure 6-16
Wiring Diagram for
Figure 6-15

Programming the Dial and Display

There is a useful command called **LOOKDOWN**, and yes, it is the reverse of the **LOOKUP** command. While the **LOOKUP** command gives you a number based on an index, the **LOOKDOWN** command gives you an index based on a number.

Example Program: SimpleLookdown.bs2

This example program demonstrates how the **LOOKDOWN** command works.

- ✓ Enter and run SimpleLookdown.bs2.
- ✓ Run the program as-is, with the **value** variable set equal to 167, and use the Debug Terminal to observe the value of **index**.
- ✓ Try setting the **value** variable equal to each of the other numbers listed by the **LOOKDOWN** command: 7, 85, 19, 28.
- ✓ Re-run the program after each change to the **value** variable and note which value from the list gets placed in the **index** variable.



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.

6

```
' 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
PAUSE 1000
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
```

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, the operator returns the index of the first value in the list that makes the statement in the instruction true.

- ✓ Modify SimpleLookdown.bs2 by substituting this **value** and **LOOKDOWN** statement in place of the existing ones:

```
value = 35
LOOKDOWN value, <= [ 7, 19, 28, 85, 167 ], index
```

- ✓ Modify the **DEBUG** command so that it reads:

```
DEBUG "Change the value variable to a ", CR,
      "different number in this range:", CR,
      "0 to 170.", CR, CR,

      "Run the modified program and ", CR,
      "check to see what number the ", CR,
      "LOOKDOWN command places in the ", CR,
      "index variable."
```

- ✓ Experiment with different values and see if the **index** variable displays what you would expect.

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 6-17.

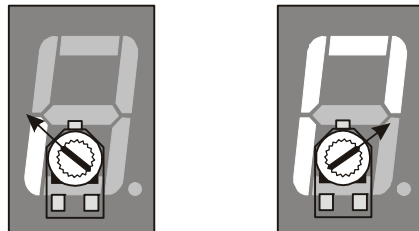


Figure 6-17
Displaying the Potentiometer's
Position with the 7-Segment LED
Display

- ✓ Enter and run DialDisplay.bs2.
- ✓ Twist the potentiometer's knob and make sure it works. Remember to press down to keep the pot seated in the breadboard.
- ✓ When you run the example program, it may not be as precise as shown in Figure 6-17. Adjust the values in the **LOOKDOWN** table so 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}

PAUSE 1000
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
```

6

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, and then loads the entry number (0 to 5 in this case) into the **index** variable.

```
LOOKDOWN time, <= [40, 150, 275, 400, 550, 800], index
```

Next, 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.



Tip: Leave your 7-segment LED circuit on your board. We'll be using the 7-segment LED again along with other circuits in Chapter 7, Activity #4.

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?

6

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 P8 through P13 to send high/low signals while leaving P14 and 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 lists 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

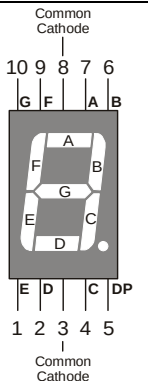
```

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 **OUTH** value. The BAFG.CDE segment list for bits in **OUTH** came from Figure 6-14 on page 182.

Letter	LED Segments	B A F G.C D E	OUTH 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 c and d	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 on page 170.			



P1. Use the schematic from Figure 6-11 on page 176. To solve this problem, modify DisplayDigitsWithLookup.bs2, using the letter patterns worked out in Exercise 2. 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

index      VAR      Byte          ' 19 chars in message

OUTH = %00000000          ' All off to start
DIRH = %11111111          ' All LEDs must be outputs

PAUSE 1000                ' 1 sec. before 1st 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

```


Chapter 7: Measuring Light

DEVICES THAT CONTAIN LIGHT SENSORS

Earlier chapters introduced pushbuttons as contact/pressure sensors and potentiometers as rotation/position sensors. Both of these sensors are common in electronic products—just think of how many devices with buttons and dials you use on a daily basis. Another common sensor found in many products is the light sensor. Here are a few examples of devices that need light sensors to function properly:

- Car headlights that automatically turn on when it's dark
- Streetlights that automatically turn on when it gets dark
- Outdoor security lights that turn on when someone walks by (but only at night)
- Laptop displays that get brighter in well lit areas and dimmer in poorly lit areas
- Cameras with automatic exposure settings
- The sensor inside TVs, DVD players and other entertainment system components that detects the infrared light from a handheld remote

7

The first three examples in the list are automatic lighting, and they depend on ambient light sensors to distinguish day from night. The electronics inside those devices only needs to know whether it's light or dark, so they can treat their light sensors as binary outputs like pushbuttons. Laptop displays and camera auto exposures adjust to area lighting conditions by getting information from their light sensors about how bright or dark it is. They have to treat their light sensors as analog outputs that supply a number that indicates how bright or dark it is, much like the Chapter 5 potentiometer examples where numbers indicated the knob's position.

The light sensors inside TVs and other entertainment system components detect infrared (IR), which is a light that is not visible to the human eye, but can be detected by many electronic devices. For example, if you look at the end of the remote that you point at a TV or other entertainment devices, you will find a clear IR LED. When you press a button on the remote, it sends coded signals to the entertainment system component by flashing the IR LED on/off. Since we can't see infrared light, it doesn't look like the remote's LED does anything when you press a button. However, if you try this while looking at the LED through the lens of a digital camera, the LED will look almost white. White light contains all the colors in the spectrum. The red green and blue sensors inside a camera chip all report that they detect light in response to white light. It so happens

that the red/green/blue sensors all detect the infrared light from the remote's IR LED as well. So the camera also interprets light from an infrared LED as white.



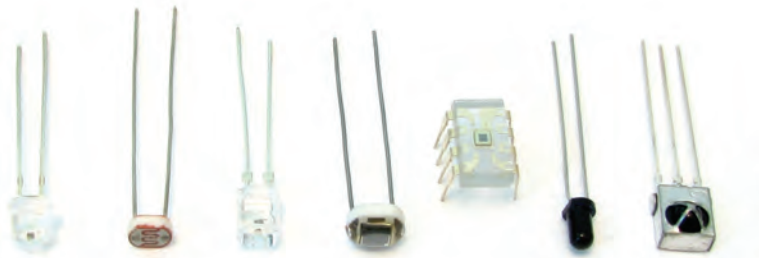
More about infrared LEDs and detectors:

Robotics with the Boe-Bot has examples where the BASIC Stamp controlled Boe-Bot robot uses the IR LED found inside TV remotes and the IR receiver found inside TV sets for detecting objects in front of it. The Boe-Bot uses the IR LED as a tiny flashlight and the IR receiver found inside TVs to detect the IR flashlights' reflections off objects in front of it. *IR Remote for the Boe-Bot* explains how TV remotes code the messages they send to TVs and has examples of how to program the BASIC Stamp microcontroller to decode messages from the remote so that you can send messages to your Boe-Bot robot, and even drive it around, all with a TV remote.

The type of light a given device senses depends on what it's designed to do. For example, light sensors for devices that adjust to ambient lighting conditions need to sense visible light. The red, green and blue pixel sensors inside digital cameras are each sensing the levels of specific colors for a digital image. The IR sensor inside a TV is looking for infrared light that's flashing on/off in the 40 kHz neighborhood. These are just a few examples, and each application requires a different kind of light sensor.

Figure 7-1 shows a few examples of the many light sensors available for various light-sensing requirements. From left to right, it shows a phototransistor, cadmium sulfide photoresistor, linear light sensor, blue enhanced photodiode, light to frequency converter, infrared phototransistor, and an infrared remote receiver from a TV.

Figure 7-1 Examples of Light Sensors





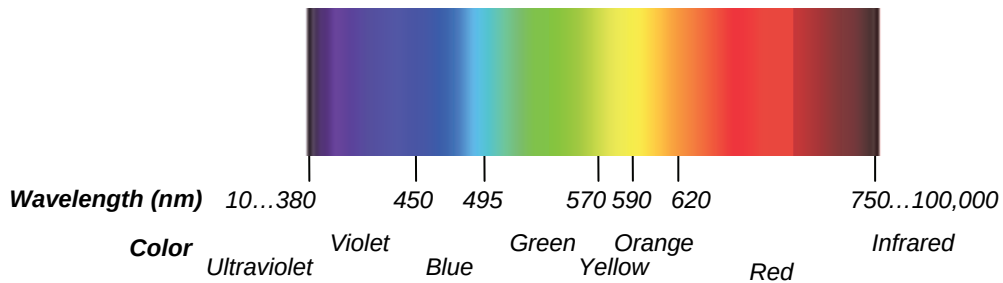
About the Cadmium Sulfide (CdS) Cell or Photoresistor

The Cadmium Sulfide (CdS) cell or photoresistor was one of the most common ambient light sensors in automatic lighting. With the advent of the European Union's Restriction of use of certain Hazardous Substances (RoHS) directive, cadmium sulfide photoresistors can no longer be built into most devices imported into or manufactured in Europe. This has increased the use of a number of photoresistor replacement products, including the phototransistor and linear light sensor. As a result of these changes, this edition now features a phototransistor for detecting light levels instead of a cadmium sulfide photoresistor.

Documentation for each light sensor describes the type of light it detects in terms of wavelength. *Wavelength* is the measure of distance between repeating shapes or cycles. For example, picture a wave traveling through the ocean, bobbing up and down. The wavelength of that wave would be the distance between each peak (or whitecap) of the wave's cycle. The wavelength of light can be measured in a similar way, instead we're measuring the distance between two peaks in the electromagnetic oscillations of light. Each color of light has its own wavelength and is considered to be *visible light*, meaning it can be detected by the human eye. Figure 7-2 shows wavelengths for visible light as well as for some types of light the human eye cannot detect, including ultraviolet and infrared. These wavelengths are measured in nanometers, abbreviated nm. One *nanometer* is one billionth of a meter.

7

Figure 7-2 Wavelengths and their Corresponding Colors



NOTE: If you are viewing this in the grayscale printed book, you may download a full-color PDF from www.parallax.com/go/WAM.

INTRODUCING THE PHOTOTRANSISTOR

A *transistor* is like a valve that allows a certain amount of electric current to pass through two of its terminals. The third terminal of a transistor controls just how much current passes through the other two. Depending on the type of transistor, the current flow can be controlled by voltage, current, or in the case of the phototransistor, by light. Figure 7-3 shows the schematic and part drawing for the phototransistor in your What's a Microcontroller kit. The more light that strikes the phototransistor's base (B) terminal, the more current it will conduct into its collector (C) terminal, which flows out of its emitter (E) terminal. Conversely, less light shining on the base terminal results in less current conducted.

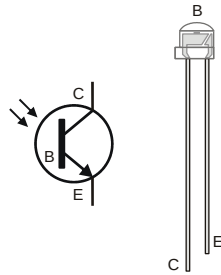


Figure 7-3
Phototransistor Schematic Symbol
and Part Drawing

This phototransistor's peak sensitivity is at 850 nm, which according to Figure 7-2, is in the infrared range. It also responds to visible light, though it's somewhat less sensitive, especially to wavelengths below 450 nm, which are left of blue in Figure 7-2. Light from halogen and incandescent lamps, and especially sunlight, are much stronger sources of infrared than fluorescent lamps. The infrared transistor responds well to all these sources of light, but it is most sensitive to sunlight, then to halogen and incandescent lamps, and somewhat less sensitive to fluorescent lamps.

Circuit designs using the transistor can be adjusted to work best in certain types of lighting conditions, and the phototransistor circuits in this chapter are designed for indoor use. There is one outdoor light sensing application, but it will use a different device that might at first seem like an unlikely candidate for a light sensor: a light emitting diode.

ACTIVITY #1: BUILDING AND TESTING THE LIGHT METER

Chapter 5 introduced RC decay measurements with the **RCTIME** command for measuring the time it took a capacitor to lose its charge through the variable resistor inside a potentiometer. With larger resistance (to the flow of electric current), the potentiometer slowed down the rate the capacitor lost its charge, and smaller resistances sped up that rate. The decay time measurement gave an indication of the potentiometer's resistance, which in turn made it possible for the BASIC Stamp to know the position of the potentiometer's dial.

When placed in an RC decay circuit, a phototransistor, which conducts more or less current when more or less light shines on it, behaves a lot like the potentiometer. When more light shines on the phototransistor, it conducts more current, and the capacitor loses its charge more quickly. With less light, the phototransistor conducts less current, and the capacitor loses its charge less quickly. So the same **RCTIME** measurement that gave us an indication of the dial's position with a potentiometer in Chapter 5 can now be used to measure light levels with a phototransistor.

7

In this activity, you will build and test an RC decay circuit that measures the time it takes a capacitor's charge to decay through a phototransistor. The RC decay measurement will give you an idea of the light levels sensed by the phototransistor's light-collecting surface. As with the potentiometer tests, the time values measured by the **RCTIME** command will be displayed in the Debug Terminal.

Light Detector Test Parts

- (1) Phototransistor
- (1) Resistor – 220 Ω (red-red-brown)
- (2) Capacitors – 0.01 μF (labeled 103)
- (1) Capacitor – 0.1 μF (labeled 104)
- (1) Jumper wire

Building the RC Time Circuit with a Phototransistor

Figure 7-4 shows a schematic and wiring diagram of the RC-time circuit you will use in this chapter. 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 potentiometer has been replaced with a phototransistor.



Tip: Leave your 7-segment LED connected, and add the phototransistor circuit to your board. We'll use the 7-segment LED with the phototransistor in Activity #4.

- ✓ Build the circuit shown in Figure 7-4.
- ✓ Make sure the phototransistor's collector and emitter (C and E) terminals are connected as shown in the wiring diagram.

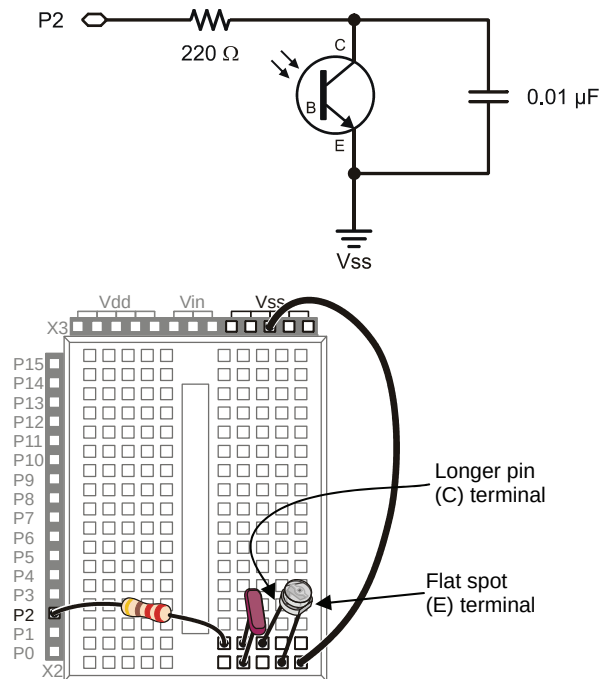


Figure 7-4
Phototransistor RC-time
Circuit Schematic and
Wiring Diagram

*Start with the 0.01 μ F
capacitor, labeled 103.*

Programming the Phototransistor Test

The first example program (TestPhototransistor.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 **HIGH 7** command from the previous example program is now **HIGH 2** since the phototransistor circuit is connected to P2 and not P7.

For the same reason, the command that was **RCTIME 7, 1, time** has been changed to **RCTIME 2, 1, time**.

Example Program: TestPhototransistor.bs2

The phototransistor's light collecting surface is at the top of its clear plastic dome, which is the base (B) terminal shown in Figure 7-3. A small black square should be visible through that dome. That black square is the actual phototransistor, a tiny piece of silicon. The rest of the device is packaging, including plastic case, lead frame, and leads.

Instead of twisting the potentiometer's knob like we did in Chapter 5, this circuit is tested by exposing the phototransistor's light collecting surface 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.



Avoid direct sunlight! The circuit and program you are using is designed to detect variations in indoor lighting and do not work in direct sunlight. Leave the indoor lights on, but close the blinds if sun is streaming in through nearby windows.

7

- ✓ Enter and run TestPhototransistor.bs2.
- ✓ Make a note of the **time** variable on the Debug Terminal under normal lighting conditions.
- ✓ Cast a shadow over the circuit with your hand and check the **time** variable again. It should be a larger number.
- ✓ Cup your hand over the circuit to cast a darker shadow; the Debug Terminal should display a significantly larger value for **time**.

```
' What's a Microcontroller - TestPhototransistor.bs2
' Read phototransistor in RC-time circuit using RCTIME command.

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

time          VAR      Word
PAUSE 1000

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

The time measurements with a 0.1 μF capacitor will take ten times as long as those with the 0.01 μF capacitor, which means the value of the **time** variable displayed by the Debug Terminal should be ten times as large. Replacing the 0.01 μF capacitor with a 0.1 μF capacitor can be useful for more brightly lit rooms where you will typically see smaller measurements with the 0.01 μF capacitor. For example, let's say the lighting conditions are very bright, and the measurements are only ranging from 1 to 13 with 0.01 μF capacitor. If you replace it with a 0.1 μF capacitor, your measurements will instead range from 1 to 130, and your application will be more sensitive to light variations within the room.

- ✓ Modify the circuit by replacing the 0.01 μF capacitor with a 0.1 μF capacitor (labeled 104).
- ✓ Re-run TestPhototransistor.bs2 and verify that the RC-time measurements are roughly ten times their former values.

The longest time interval the **RCTIME** command can measure is 65535 units of 2 μs each, which corresponds to a decay time of: $65535 \times 2 \mu\text{s} = 131 \text{ ms} = 0.131 \text{ s}$. If the decay time exceeds 0.131 seconds, the **RCTIME** command returns 0 to indicate that the maximum measurement time was exceeded.

- ✓ Can you cast a dark enough shadow over the phototransistor to exceed the maximum 65535 measurement and make the **RCTIME** command return 0?

The next activity will rely on the smaller of the two capacitors.

- ✓ Before you move on to the next activity, return the circuit to the original one shown in Figure 7-4 by removing the 0.1 μF capacitor and replacing it with the 0.01 μF capacitor.

ACTIVITY #2: TRACKING LIGHT EVENTS

One of the more useful features of the BASIC Stamp module's program memory is that you can disconnect power to the board without losing your program. As soon as power is reconnected, the program will start running again from the beginning. Since the code for your application typically doesn't fill up the BASIC Stamp module's memory, any portion 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. While

the values stored by variables get erased when the power gets disconnected, the BASIC Stamp will still remember all the values stored in its program memory when the power gets reconnected.



What is datalogging? *Datalogging* is what a microcontroller does when it records and stores periodic sensor measurements for a certain amount of time. Datalogging devices, or dataloggers, are especially useful in scientific research. For example, instead of posting a person in a remote location to take weather measurements, a datalogging weather station can be deployed. It records periodic measurements, and scientists visit the station every so often to collect the data, or in some cases, it uploads its measurements to a computer by cell phone, radio, or satellite.

The chip on the BASIC Stamp that stores program memory and data is shown in Figure 7-5. This chip is called an EEPROM, which stands for Electrically Erasable Programmable Read-Only Memory. That's quite a mouthful, and pronouncing each of the first letters in EEPROM is still a lot of work. So, when people talk about an EEPROM, they usually pronounce it: "E-E-Prom".

7

2 KB EEPROM stores your PBASIC source code.

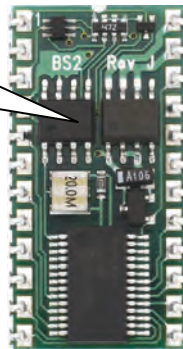


Figure 7-5
EEPROM Chip on BASIC Stamp Module

This EEPROM stores your program code and any other data your program places there, even when power is disconnected.

Figure 7-6 shows the BASIC Stamp Editor's Memory Map window. You can view this window by clicking the BASIC Stamp Editor's Run menu and selecting Memory Map.

The Memory Map uses different colors to show how both the BASIC Stamp module's RAM (variables in random access memory) and EEPROM (program memory) are being used. The red square in the scroll bar at the far left indicates what portion of the EEPROM is visible in the EEPROM Map. You can click and drag this square up and down to view various portions of the EEPROM memory. By dragging that red square down to the bottom, you can see how much EEPROM memory space is used by TestPhototransistor.bs2 from Activity #1. The bytes that contain program tokens are

highlighted in blue, and only 35 bytes out of the 2048 byte EEPROM are used for the program. The remaining 2013 bytes are free to store data.

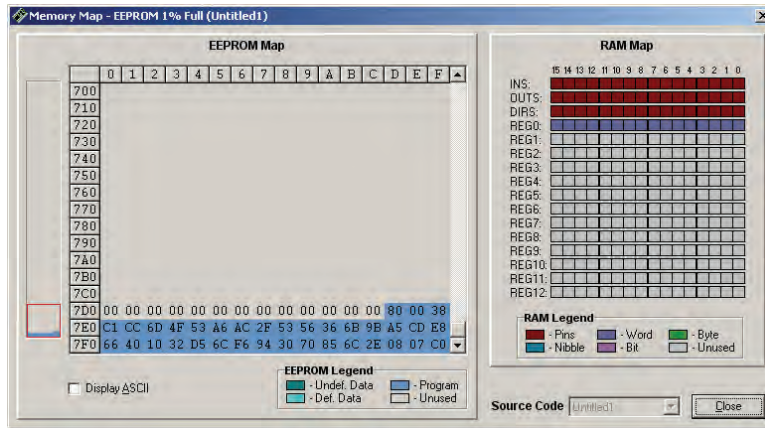


Figure 7-6
Memory Map

To view this window, click Run, and select Memory Map.

The EEPROM Map shows the addresses as hexadecimal values, which were discussed briefly in the Decimal vs. Hexadecimal box on page 183. The values along the left side show the starting address of each row of bytes. The values along the top show the byte number within that row, from 0 to F in hexadecimal, which is 0 to 15 in decimal. For example, in Figure 7-6, the hexadecimal value C1 is stored at address 7E0. CC is stored at address 7E1, 6D is stored at address 7E2, and so on, up through E8, which is stored at address 7EF. If you scroll up and down with the scroll bar, you'll see that the largest memory addresses are at the bottom of the EEPROM Map, and the smallest addresses are at the top, with the very top row starting at 000.

PBASIC programs are always stored at the largest addresses in EEPROM, which are shown at the bottom of the EEPROM Map. So, if your program is going to store data in EEPROM, it should start with the smallest addresses, starting with address 0. This helps ensure that your stored data won't overwrite your PBASIC program, which will usually result in a program crash. In the case of the EEPROM Map shown in Figure 7-6, the PBASIC program resides in addresses 7FF through 7DD, starting at the largest address and building to smaller addresses. So your application can store data from address 000 through 7DC, building from the smallest to the largest. In decimal, that's addresses 0 through 2012.

If you plan on storing data to EEPROM, it is important to be able convert from hexadecimal to decimal in order to calculate the largest writable address. Below is the

math for converting the number 7DC from hexadecimal to decimal. Hexadecimal is a numerical system with a base of 16, meaning it uses 16 different digits to represent its values. The digits 0-9 represent the first 10 values, and the letters A-F represent values 10-15. When converting from hexadecimal to decimal, each digit from the right represents a larger power of sixteen. The rightmost digit is the number of ones, which is the number of 16^0 s. The next digit from the right is the number of 16s, which is the number of 16^1 s. The third digit is the number of 256s, which is the number of 16^2 s.

$$\begin{aligned}
 \text{Hexadecimal 7DC} &= (7 \times 16^2) + (D \times 16^1) + (C \times 16^0) \\
 &= (7 \times 16^2) + (13 \times 16^1) + (12 \times 16^0) \\
 &= (7 \times 256) + (13 \times 16) + (12 \times 1) \\
 &= 1792 + 208 + 12 \\
 &= 2012 \text{ (decimal value)}
 \end{aligned}$$

This conversion approach works the same in other bases, including base 10 decimal values. For example:

$$\begin{aligned}
 2102 &= (2 \times 10^3) + (1 \times 10^2) + (0 \times 10^1) + (2 \times 10^0) \\
 &= (2 \times 1000) + (1 \times 100) + (0 \times 10) + (2 \times 1)
 \end{aligned}$$



2048 bytes = 2 KB.

Although both the upper case “K” and the lower-case “k” are called “kilo,” they are slightly different. In electronics and computing, the upper-case K is used to indicate a binary kilobyte, which is $1 \times 2^{10} = 1024$. When referring to exactly 1000 bytes, use the lower-case k, which stands for kilo and is $1 \times 10^3 = 1000$ in the metric system.

Also, the upper-case “B” stands for bytes, while the lower-case “b” stands for bits. This can make a big difference because 2 Kb means 2048 bits, which is 2048 different numbers, but each number is limited to a value of either 0 or 1. In contrast, 2 KB, is 2048 bytes, each of which can store a value in the 0 to 255 range.

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. It could track the temperature during the entire trip to see if it always stayed cool enough to make sure none of the shipment thawed. 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. This can give an indication of cloud cover at times of day, and some studies use it to monitor the effects of pollution and airplane condensation trails (con trails) on light levels that reach the Earth’s surface.

With light level tracking in mind, this activity introduces a technique for storing measured light levels to the EEPROM and then retrieving them again. 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} Value

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 from 0 to 65565 while byte values can only contain numbers from 0 to 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. If you want to store the first value at address 8, 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 your 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 largest EEPROM addresses, it's best to use the smallest **Location** values for storing numbers with the **WRITE** command.

How do I know if the *Location* I'm using is too large? You can use the Memory Map to figure out the largest value not used by your PBASIC program. The explanation after Figure 7-6 on page 204 describes how to calculate how many memory addresses are available. As a shortcut for converting hexadecimal to decimal, you can use the **DEBUG** command's **DEC** formatter and the **\$** hexadecimal formatter like this:

DEBUG DEC \$7DC

Your program will display the decimal value of hexadecimal 7DC because the **\$** hexadecimal formatter tells the **DEBUG** command that 7DC is a hexadecimal number. Then, the **DEC** formatter makes the **DEBUG** command display that value in decimal format.

Example Program: StoreLightMeasurementsInEeprom.bs2

7

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. Like the previous activity's example program, it displays the measurements in the Debug Terminal, but it also stores each measurement in EEPROM for later retrieval by a different program that uses the **READ** command.

- ✓ Enter and run StoreLightMeasurementsInEeprom.bs2.
- ✓ Record the measurements displayed by the Debug Terminal so that you can verify the measurements read back from the EEPROM.
- ✓ Gradually increase the shade over the phototransistor during the 2 ½ minute test period for meaningful data.

Especially if you have a USB board, reconnecting it to the computer could reset the BASIC Stamp and restart the program, in which case, it would start taking a new set of measurements.

- ✓ After StoreLightMeasurementsInEeprom.bs2 has completed, disconnect power from your board immediately and leave it disconnected until you are ready to run the next example program: ReadLightMeasurementsFromEeprom.bs2.



You can change the pauses in the FOR...NEXT loop. This example program has 5 second pauses, which emphasize the periodic measurements that datalogging devices take. They might seem kind of long, so, you can reduce the **PAUSE 5000** to **PAUSE 500** to make the program execute ten times more quickly for testing.

```
' What's a Microcontroller - StoreLightMeasurementsInEeprom.bs2
' Write light measurements to EEPROM.

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

time          VAR      Word
eepromAddress VAR      Byte

PAUSE 1000
DEBUG "Starting measurements...", CR, CR
      "Measurement   Value", CR,
      "-----", CR
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
```

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 decay time measurement into the word size **time** variable.

```
RCTIME 2, 1, time
```

The value the **time** variable stores is copied to the EEPROM address given by the current value of the **eepromAddress** variable each time through the loop. Remember, that the address for a **WRITE** command is always in terms of bytes. So, the **eepromAddress**

variable is incremented by two each time through the loop because a **word** variable takes up two bytes.

```
WRITE eepromAddress, Word time
NEXT
```

Programming Data Retrieval

To retrieve the values you just recorded from EEPROM, you can use the **READ** command. The syntax for the **READ** command is:

READ *Location*, {**WORD**} *Variable*

While the **WRITE** command can copy a value from either a constant or a variable to EEPROM, the **READ** command has to copy the value stored at an address in EEPROM to a variable, so as its name suggests, the **Variable** argument has to be a variable.



Keep in mind that variables are stored in the BASIC Stamp module's RAM. Unlike EEPROM, RAM values get erased whenever the power disconnected and also whenever the Reset button on your board gets pressed.

The BASIC Stamp 2 has 26 bytes of RAM, shown on the right side of the Memory Map in Figure 7-6 on page 204. If you declare a word variable, you are using up two bytes. A byte variable declaration uses one byte, a nibble uses half a byte, and one bit uses 1/8 of a byte.

Let's say that **eepromValueA** and **eepromValueB** are **word** variables, and **littleEE** is a **Byte** variable. These variables would have to be defined at the beginning of a program with **VAR** variable declarations. Here are some commands to retrieve the values that were stored at certain EEPROM addresses earlier with **WRITE** commands, maybe even in a different program.

```
READ 7, littleEE
READ 8, Word eepromValueA
READ 10, Word eepromValueB
```

The first command retrieves a byte value from EEPROM address 7 and copies it to the variable named **littleEE**. The next command copies the word occupying EEPROM byte addresses 8 and 9 and stores it in the **eepromValueA** word variable. The last of the three commands copies the word occupying EEPROM byte addresses 10 and 11 and stores it in the **eepromValueB** variable.

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.

- ✓ Reconnect power to your board.
- ✓ Enter ReadLightMeasurementsFromEeprom.bs2 into the BASIC Stamp Editor.
- ✓ If you have disconnected power from your board, when you reconnect, immediately click the BASIC Stamp Editor's Run button to download the program into the BASIC Stamp.



Don't wait longer than 6 seconds between reconnecting power and loading ReadLightMeasurementsFromEeprom.bs2 into the BASIC Stamp or the program that's still in the program memory (StoreLightMeasurementsInEeprom.bs2) will start recording over previous measurements. **Also, if you reduced the PAUSE command's *Duration*** from 5000 to 500, you will only have 1.5 seconds!

- ✓ Compare the Debug Terminal 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

PAUSE 1000

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 relies on byte addresses. Since we want to read word values from 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**, and the value gets copied to the **time** variable.

```
READ eepromAddress, Word time
```

The values of the **time** and **eepromAddress** variables are displayed in adjacent columns as a table in the Debug Terminal.

```
DEBUG DEC2 eepromAddress, "          ", DEC time, CR
NEXT
```

7

Your Turn – More Measurements

- ✓ Modify StoreLightMeasurementsInEeprom.bs2 so that it takes and records twice as many measurements in the same amount of time.
- ✓ Modify ReadLightMeasurementsFromEeprom.bs2 so that it displays all of the measurements from the just-modified StoreLightMeasurementsInEeprom.bs2.

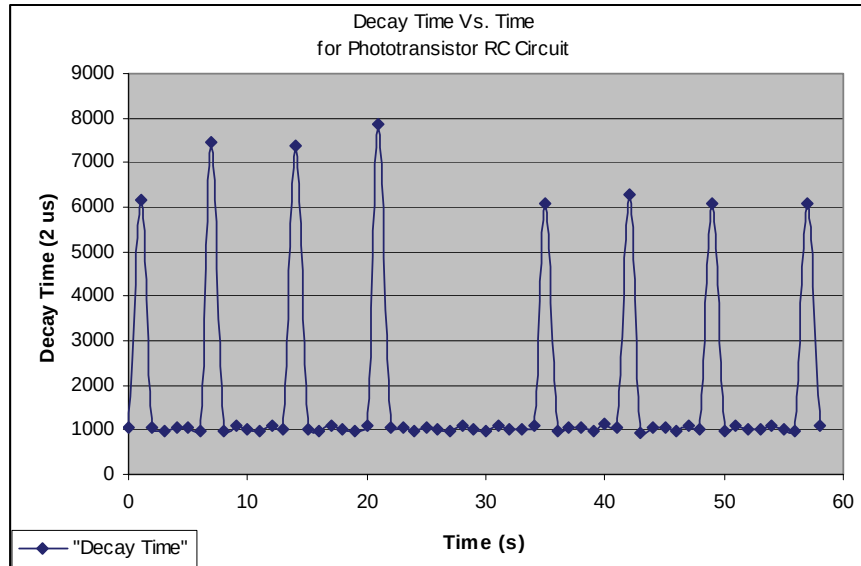
ACTIVITY #3: GRAPHING LIGHT MEASUREMENTS (OPTIONAL)

Lists of measurements like the ones in Activity #2 can be tedious to analyze. Imagine reading through hundreds of those numbers looking for when the sun set. Or maybe you're looking for a particular event, like when the light sensor was briefly covered. You might even be looking for a pattern in how frequently the light sensor was covered. This information could be useful if the light sensor is placed in an area where a person or animal walks over it, or an object passing over it on a conveyer belt needs to be recorded and analyzed. Regardless of the application, if all you have to work with is a long list of numbers, finding those events and patterns can be a difficult and time-consuming task.

If you instead make a graph of the list of measurements, it makes finding events and patterns a lot easier. The person, animal or object passing over the light sensor will show up as a high point or spike in the measurements. Figure 7-7 shows an example of a graph that might indicate that rate at which objects on a conveyer belt are passing over the sensor. The spikes in the graph occur when the measurements get large. In the case of a

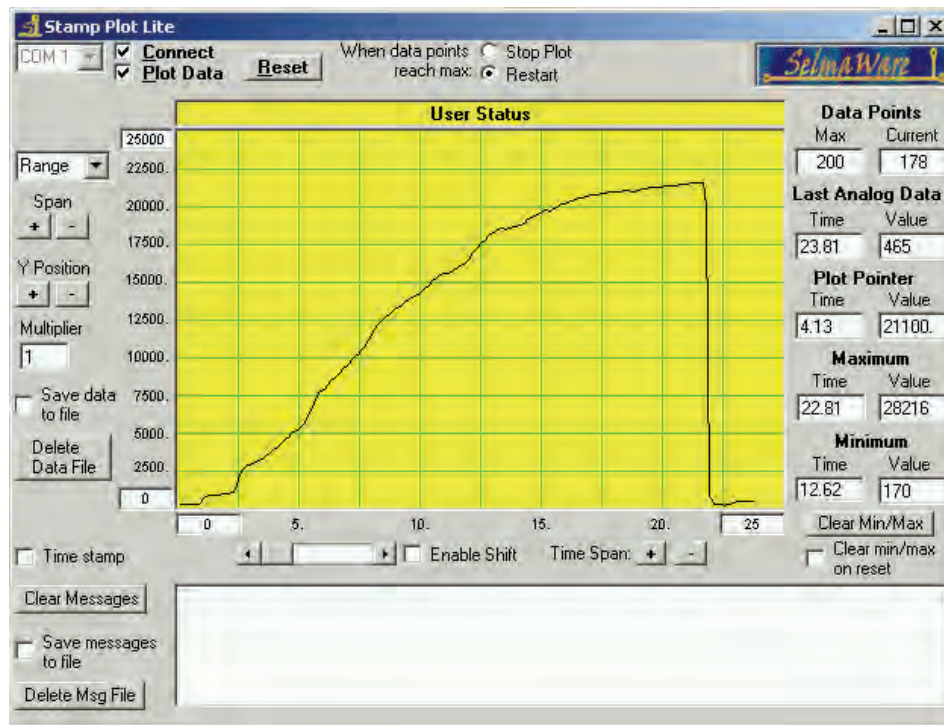
conveyor belt, it would indicate that an object passed over the sensor casting a shadow. This graph makes it easy to see at a glance that an object passes over the sensor roughly every 7 seconds, but that the object we were expecting at 28 seconds was wasn't there.

Figure 7-7 Graph of Phototransistor Light Measurements



The graph in Figure 7-7 was generated by copying and pasting values in the Debug Terminal to a text file which was then imported into a Microsoft Excel spreadsheet. Some graphing utilities can take the place of the Debug Terminal and plot the values directly instead of displaying them as lists of numbers. Figure 7-8 shows an example of one of these utilities, called StampPlot LITE.

Figure 7-8 StampPlot LITE



7

In this optional activity, you can go to www.parallax.com/go/WAM and then follow the Data Plotting link to try a number of activities that demonstrate how to plot values using various spreadsheets and graphing utility software packages.

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 activity changes the rate that the display flickers depending on the light intensity it detects.

Light Meter Parts

- (1) Phototransistor
- (1) Resistor – 220 Ω (red-red-brown)
- (2) Capacitors – 0.01 μF (labeled 103)
- (1) Capacitor – 0.1 μF (labeled 104)
- (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 phototransistor circuit schematics that will be used to make the light meter, and Figure 7-10 shows a wiring diagram of the circuits. The phototransistor circuit is the same one you have been using in the last two activities, and the 7-segment LED display circuit is the same one from Figure 6-11 on page 176.

- ✓ 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 the program `SegmentTestWithHighLow.bs2` from Chapter 6, Activity #2, which starts on page 175.

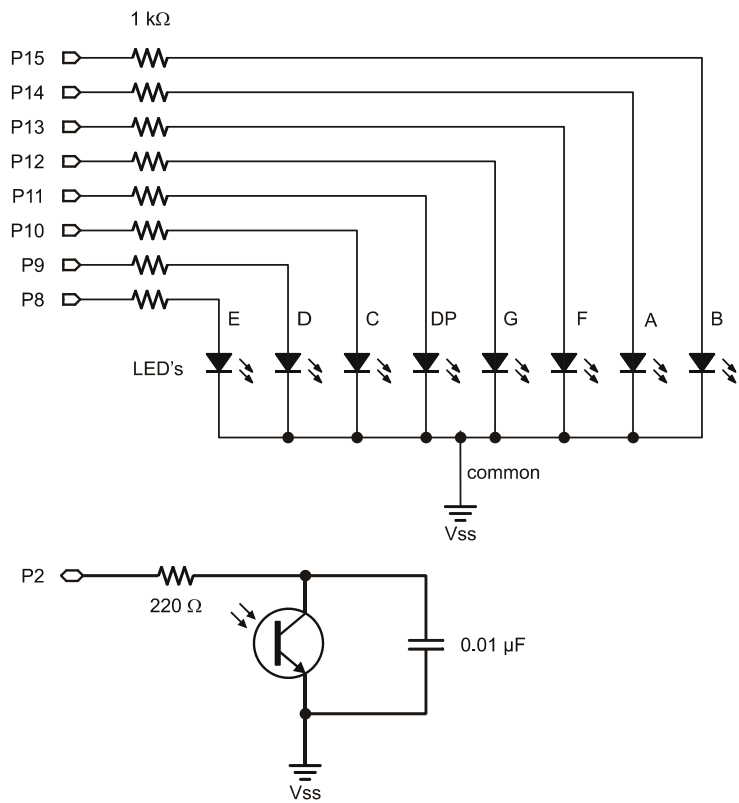


Figure 7-9
Light Meter Circuit
Schematic

7

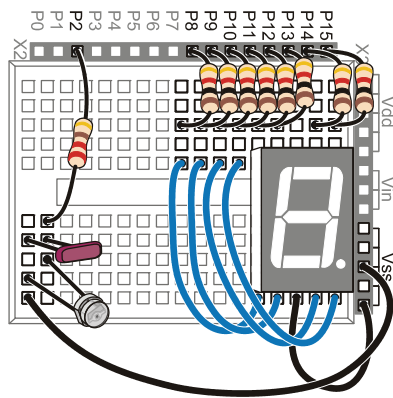


Figure 7-10
Wiring Diagram for
Figure 7-9

Using Subroutines

Most of the programs you have written so far operate inside a **DO...LOOP**. Since the entire program's 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 instead 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. The jobs are called *subroutines* and this process is *calling* 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** command. 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"**.

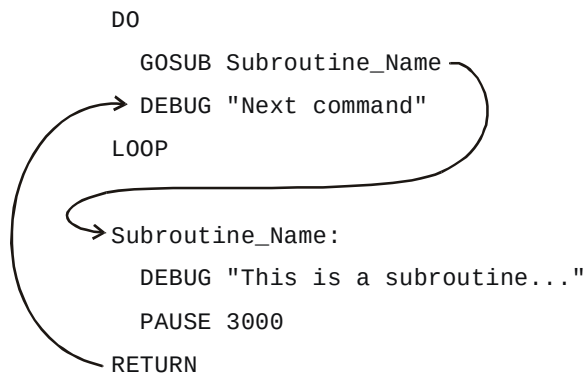


Figure 7-11
How Subroutines Work



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**. A label must end with a colon, and for the sake of style, separate words with the underscore character so they are easy to recognize. When picking a name for a label, make sure not to use a reserved word or a name that is already used in a variable or constant. The rest of the rules for a label name are the same as the ones for naming variables, which are listed in the information box on page 43.

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.

7

```
' What's a Microcontroller - SimpleSubroutines.bs2
' Demonstrate how subroutines work.

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

PAUSE 1000

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. Then, the three commands inside that subroutine are executed. When the program gets to the **RETURN** command, it jumps back to the command that comes right after **GOSUB First_Subroutine**, 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*.

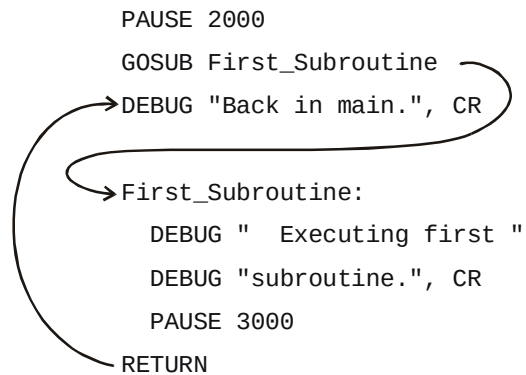


Figure 7-12
First Subroutine Call

Figure 7-13 shows a second example of the same process with the second subroutine call (**GOSUB Second_Subroutine**).

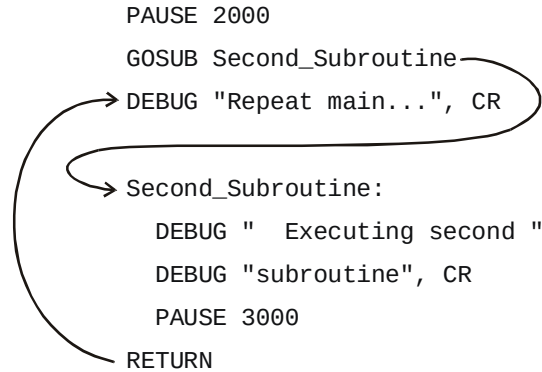


Figure 7-13
Second Subroutine Call

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 shown in Figure 7-11 on page 216 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** command 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 or the BASIC Stamp Editor's Help for more information. Look up **GOSUB** and **RETURN**.

Light Meter Using Subroutines

The next program, LightMeter.bs2 uses subroutines to control the display of the 7-Segment LED depending on the level of light detected by the phototransistor. The display's segments cycle on and off in a circular pattern that gets faster when the light on the phototransistor gets brighter. When the light gets dimmer, the circular pattern cycling goes slower. The LightMeter.bs2 example program uses a subroutine named **Update_Display** to control the order in which the light meter segments advance.

The program that runs the light meter will deal with three different operations:

1. Read the phototransistor.
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 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 little or 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 cycling speed of the circular pattern displayed by the 7-segment LED is controlled by the lighting conditions the phototransistor is sensing. Do this by casting a shadow over it with your hand or a piece of paper and verify that the rate of the circular display pattern slows down.

```
' What's a Microcontroller - LightMeter.bs2
' Indicate light level using 7-segment display.

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

PAUSE 1000

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

Get_Rc_Time:                         ' RC-time subroutine

    HIGH 2
    PAUSE 3
    RCTIME 2, 1, time

RETURN

Delay:                               ' Delay subroutine.

    PAUSE time / 3

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

```

7

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
    time  VAR Word

```

' Variable declarations.

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
```

The 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 phototransistor 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.

```
Get_Rc_Time:           ' Subroutines
                        ' RC-time subroutine
  HIGH 2
  PAUSE 3
  RCTIME 2, 1, time
RETURN
```

The second subroutine's name is **Delay**, and all it contains is **PAUSE time / 3**. The **PAUSE** command allows the measured decay time (how bright the light is) to control the delay between turning on each light segment in the 7-segment LED's revolving circular display. The value to the right of the divide / operator can be made larger for faster rotation in lower light conditions or smaller to slow the display for brighter light conditions. You could also use * to multiply the **time** variable by a value instead of dividing to make the display go a lot slower, and for more precise control over the rate, don't forget about the */ operator. More on this operator in the Your Turn section.

```

Delay:
  PAUSE time / 3
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 six entries in the **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.

7

```

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,” which is the PBASIC program, can be changed to adjust the speed. As mentioned earlier, dividing the **time** variable in the **Delay** subroutine's **PAUSE time / 3** command by numbers larger than 3 will speed up the display, and smaller numbers will slow it down. To really slow it down, time can also be multiplied by values with the multiply ***** operator, and for fine tuning, there's the ***/** operator.

When you connect capacitors in parallel, their values add up. So, if you plug in a second 0.01 μF capacitor right next to the first one as shown in Figure 7-14 and Figure 7-15, the

capacitance will be $0.02 \mu\text{F}$. With twice the capacitance, the decay measurement for the same light level will take twice as long.

- ✓ Connect the second $0.01 \mu\text{F}$ capacitor right next to the first one in the light sensor portion of the light meters circuit in Figure 7-14 and Figure 7-15.
- ✓ Run LightMeter.bs2 and observe the result.

Since the time measurements will be twice as large, the 7-segment LED's circular pattern should rotate half as fast.

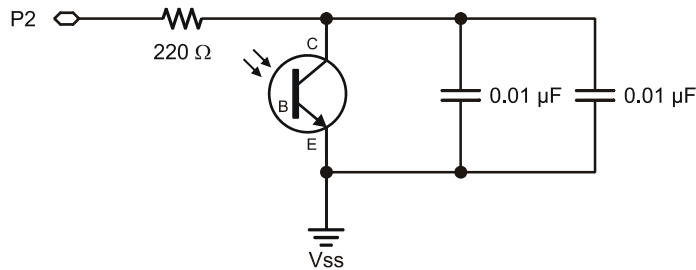


Figure 7-14
Two $0.01 \mu\text{F}$ Capacitors
in Parallel = $0.02 \mu\text{F}$

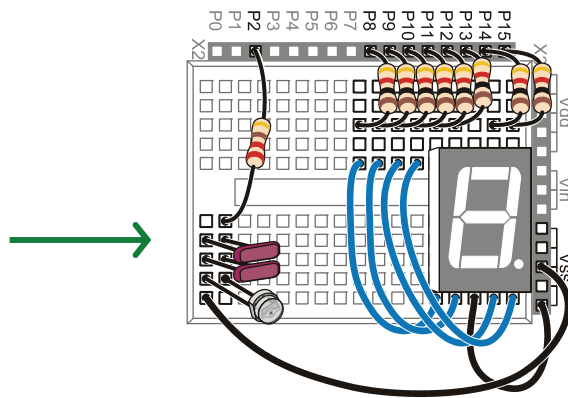


Figure 7-15
Light Meter Circuits with
Two $0.01 \mu\text{F}$ Capacitors
in Parallel

Instead of half the speed of a $0.01 \mu\text{F}$ capacitor, how about one tenth the speed? You can do this by replacing the two $0.01 \mu\text{F}$ capacitors with a $0.1 \mu\text{F}$ capacitor. It will work okay in brightly lit rooms, but will likely be a little slow for normal lighting. Remember that when you use a capacitor that is ten times as large, the RC-time measurement will take ten times as long.

- ✓ Replace the 0.01 μF capacitors with a 0.1 μF capacitor.
- ✓ Run the program and see if the predicted effect occurred.
- ✓ Before continuing, restore the circuit to one 0.01 μF capacitor in parallel with the phototransistor as shown in Figure 7-9 and Figure 7-10, starting on page 215.
- ✓ Test your restored circuit to verify that it works before continuing.



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 under the widest range of conditions. This takes a considerable amount of testing and refinement, but that's all part of the product design process.

ACTIVITY #5: ON/OFF PHOTOTRANSISTOR OUTPUT

7

Before microcontrollers were common in products, photoresistors were used in circuits that varied in their voltage output. When the voltage passed below a threshold value indicating nighttime, other circuits in the device turned the lights on. When the voltage passed above the threshold, indicating daytime, the other circuits in the device turned the lights off. This binary light switch behavior can be emulated with the same BASIC Stamp and the RC decay circuit by simply modifying the PBASIC program. Alternatively, the circuit can be modified so that it sends a 1 or 0 to an I/O pin depending on the amount of voltage supplied to the pin, similar to the way a pushbutton does. In this activity, you will try both these approaches.

Adjusting the Program for On/Off States

PhototransistorAnalogToBinary.bs2 takes the range of phototransistor measurements and compares it to the half way point between the largest and smallest measurements. If the measurement is above the half way point, it displays “Turn light on”; otherwise, it displays “Turn light off.” The program uses constant directives to define the largest and smallest measurements the program should expect from the phototransistor circuit.

```
valMax      CON      4000
valMin      CON      100
```

The program also uses **MIN** and **MAX** operators to ensure that values stay within these limits before using them to make any decisions. If **time** is greater than **valMax** (4000 in the example program), the statement sets **time** to **valMax = 4000**. Likewise if **time** is less than **valMin** (100 in the example program), the statement sets **time** to **valMin = 100**.

```
time = time MAX valMax MIN valMin
```

An **IF...THEN...ELSE** statement converts the range of digitized analog values into a binary output that takes the form of light-on or light-off messages.

```
IF time > (valMax - valMin) / 2 THEN
  DEBUG CR, "Turn light on "
ELSE
  DEBUG CR, "Turn light off"
ENDIF
```

Before this program will work properly, you have to calibrate your lighting conditions as follows:

- ✓ Check your phototransistor circuit to make sure it has just one 0.01 μ F capacitor (labeled 103).
- ✓ Enter PhototransistorAnalogToBinary.bs2 into the BASIC Stamp Editor. Make sure to add an extra space after the "n" in the "Turn light on " message. Otherwise, you'll get a phantom "f" from the "Turn light off" message, which has one more character in it.
- ✓ Load the program into the BASIC Stamp.
- ✓ Watch the Debug Terminal as you apply the darkest and brightest lighting conditions that you want to test, and make notes of the resulting maximum and minimum time values.
- ✓ Enter those values into the program's **valMax** and **valMin CON** directives.

Now, your program is ready to run and test.

- ✓ Load the modified program into the BASIC Stamp.
- ✓ Test to verify that dim lighting conditions result in the "Turn Light on" message and bright lighting conditions result in the "Turn light off" message.

```
' What's a Microcontroller - PhototransistorAnalogToBinary.bs2
' Change digitized analog phototransistor measurement to a binary result.

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

valMax      CON      4000
valMin      CON      100

time        VAR      Word

PAUSE 1000

DO

  HIGH 2
  PAUSE 100
  RCTIME 2, 1, time

  time = time MAX valMax MIN valMin

  DEBUG HOME, "time = ", DEC5 time
  IF time > (valMax - valMin) / 2 THEN
    DEBUG CR, "Turn light on "
  ELSE
    DEBUG CR, "Turn light off"
  ENDIF

LOOP
```

7

Your Turn – Different Thresholds for Light and Dark

If you try to incorporate PhototransistorAnalogToBinary.bs2 into an automatic lighting system, it has a potential defect. Let's say it's dark enough outside to cause the **time** measurement to pass above **(valMax - valMin) / 2**, so the light turns on. But if the sensor detects that light, it would cause the measurement to pass back below **(valMax - valMin) / 2**, so the light would turn off again. This lights-on/lights-off cycle could repeat rapidly all night!

Figure 7-16 shows how this could work in a graph. As the light level drops, the value of the **time** variable increases, and when it crosses the threshold, the automatic lights turn on. Then, since the phototransistor senses the light that just turned on, the **time** variable's measurement drops back below the threshold, so the lights turn off. Then, the **time** variable's value increases again, and it passes above the threshold, so the lights turn on, and the **time** variable drops below the threshold again, and so on...

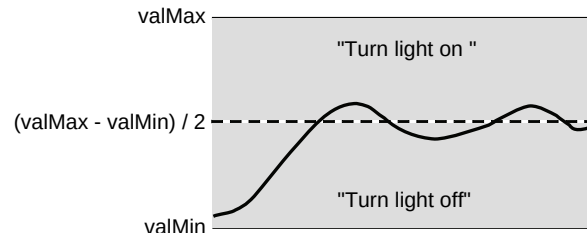


Figure 7-16
Oscillations
Above/Below a
Threshold

One remedy for this problem is to add a second threshold, as illustrated in Figure 7-17. The “Turn light on” threshold only turns the light on after it’s gotten pretty dark, and the “Turn light off” threshold only turns the light back off after it’s gotten pretty bright. With this system, the light comes on after **time** passed into the “Turn light on” range. The light turning on made it brighter, so **time** dropped slightly, but since it didn’t fall clear down past the “Turn light off” threshold, nothing changed, and the light stays on as it should. The term *hysteresis* is used to describe this type of system, which has two different input thresholds that affect its output along with a no-transition zone between them.

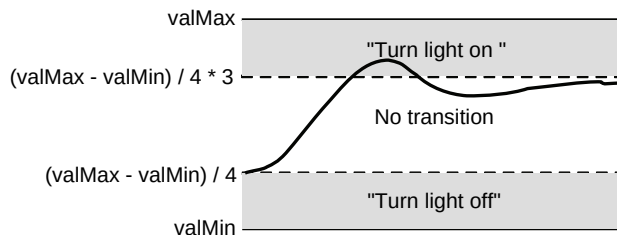


Figure 7-17
Using Different High
and Low Thresholds
to Prevent
Oscillations

You can implement this two-threshold system in your PBASIC code by modifying PhototransistorAnalogToBinary.bs2's **IF...THEN...ELSEIF** statement. Here is an example:

```
IF time > (valMax - valMin) / 4 * 3 THEN
  DEBUG CR, "Turn light on "
ELSEIF time < (valMax - valMin) / 4 THEN
  DEBUG CR, "Turn light off"
ENDIF
```

The first **IF...THEN** code block displays "Turn lights on " when the **time** variable stores a value that's more than $\frac{3}{4}$ of the way to the highest time (lowest light) value. The **ELSEIF** code block only displays "Turn lights off" when the **time** variable stores a value that's less than $\frac{1}{4}$ of the way above the smallest time (brightest) value.

- ✓ Save PhototransistorAnalogToBinary.bs2 as PhototransistorHysteresis.bs2.
- ✓ Before modifying PhototransistorHysteresis.bs2, test it to make sure the existing threshold works. If the lighting has changed, repeat **valMin** and **valMax** calibration steps (before the PhototransistorAnalogToBinary.bs2 example code).
- ✓ Replace PhototransistorHysteresis.bs2's **IF...ELSE...ENDIF** statement with the **IF...ELSEIF...ENDIF** just discussed.
- ✓ Load the PhototransistorHysteresis.bs2 into the BASIC Stamp.
- ✓ Test and verify that the "Turn light on" threshold is darker, and the "Turn light off" threshold is lighter.

7

If you add an LED circuit and modify the code so that it turns the LED on and off, some interesting things might happen. Especially if you put the LED right next to the phototransistor, you might still see that on/off behavior when it gets dark, even with the hysteresis programmed. How far away from the phototransistor does the LED have to be to get the two thresholds to prevent the on/off behavior? Assuming that **valMin** and **valMax** are the same in both programs, how much farther away does the LED have to be for the unmodified PhototransistorAnalogToBinary.bs2 to work properly?

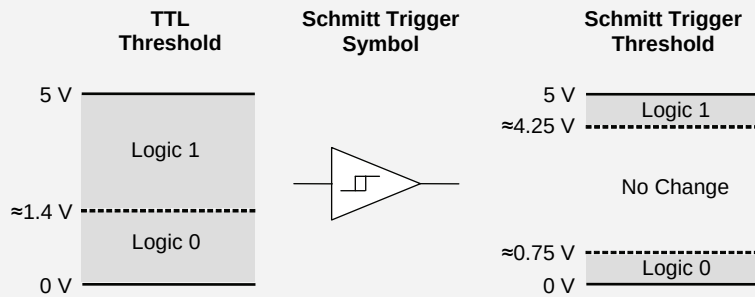


TTL Vs. Schmitt Trigger

Your BASIC Stamp I/O pin sends and receives signals using *transistor-transistor logic* (TTL). As an output, the I/O pin sends a 5 V high signal or a 0 V low signal. The left side of Figure 7-18 shows how the I/O pin behaves as an input. The I/O pin's **IN** register (**IN0**, **IN1**, **IN2**, etc.) stores a 1 if the voltage applied is above 1.4 V, or a 0 if it's below 1.4 V. These are shown as Logic 1 and Logic 0 in the figure.

A *Schmitt trigger* is a circuit represented by the symbol in the center of Figure 7-18. The right side of Figure 7-18 shows how an I/O pin set to input would behave if it had a Schmitt trigger circuit built-in. Like the PBASIC code with two thresholds, the Schmitt trigger has hysteresis. The input value stored by the I/O pin's **INx** register doesn't change from 0 to 1 until the input voltage goes above 4.25 V. Likewise, it doesn't change from 1 to 0 until the input voltage passes below 0.75 V. The BASIC Stamp 2px has a PBASIC command that allows you to configure its input pins to Schmitt trigger.

Figure 7-18 TTL Vs. Schmitt Trigger Thresholds and Symbol



Adjusting the Circuit for On/Off States

As mentioned in Chapter 5, Activity #2, the voltage threshold for a BASIC Stamp I/O pin is 1.4 V. When an I/O pin is set to input, voltages above 1.4 V applied to the I/O pin result in a binary 1 and voltages below 1.4 V result in a binary 0. The V_o node in the circuit shown in Figure 7-19 varies in voltage with light. This circuit can be connected to a BASIC Stamp I/O pin, and with low light, the voltage will pass below the BASIC Stamp's 1.4 V threshold, and the I/O pin's input register will store a 0. In bright light conditions, V_o rises above 1.4 V, and the I/O pin's input register will store a 1.

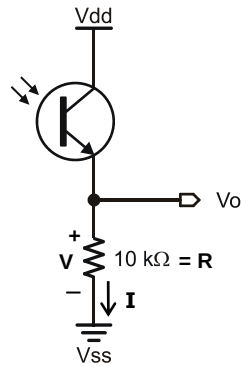


Figure 7-19
Voltage Output Light
Circuit

The reason the voltage at V_o changes with light levels is because of Ohm's Law, which states that the voltage across a resistor (V in Figure 7-19) is equal to the current passing through that resistor (I), multiplied by the resistor's resistance (R).

7

$$V = I \times R$$

Remember that a phototransistor lets more current pass through when exposed to more light, and less current when exposed to less light. Let's take a closer look at the example circuit in Figure 7-19 and calculate how much current would have to pass through the resistor to create a 1.4 V drop across the resistor. First, we know the value of the resistor is 10 kΩ, or 10,000 Ω. We also know that we want the voltage to be equal to 1.4 V, so we need to modify Ohm's Law to solve for I . To do this, divide both sides of the $V = I \times R$ equation by R , which results in $I = V \div R$. Then, substitute the values you know ($V = 1.4$ V and $R = 10$ kΩ), and solve for I .

$$\begin{aligned}
 I &= V \div R \\
 &= 1.4 \text{ V} \div 10 \text{ k}\Omega \\
 &= 1.4 \text{ V} \div 10,000 \Omega \\
 &= 0.00014 \text{ V}/\Omega \\
 &= 0.00014 \text{ A} \\
 &= 0.14 \text{ mA}
 \end{aligned}$$

Now, what if the transistor allows twice that much current through because it's bright, and what would the voltage be across the resistor? For twice the current, $I = 0.28 \text{ mA}$, and the resistance is still $10 \text{ k}\Omega$, so now we are back to solving V from the original $V = I \times R$ equation with $I = 0.28 \text{ mA}$ and $R = 10 \text{ k}\Omega$:

$$\begin{aligned}
 V &= I \times R \\
 &= 0.28 \text{ mA} \times 10 \text{ k}\Omega \\
 &= 0.00028 \text{ A} \times 10,000 \Omega \\
 &= 2.8 \text{ A}\Omega \\
 &= 2.8 \text{ V}
 \end{aligned}$$

With 2.8 V applied to an I/O pin, its input register would store a 1 since 2.8 V is above the I/O pin's 1.4 V threshold voltage.

Your Turn – More calculations

What if the phototransistor allowed half the threshold voltage current (0.07 mA) through the circuit, what would the voltage across the resistor be? Also, what would the I/O pin's input register store?

Test the Binary Light Sensor

Testing the binary light sensor circuit is a lot like testing the pushbutton circuit from Chapter 3. When the circuit is connected to an I/O pin, the voltage will either be above or below the BASIC Stamp I/O pin's 1.4 V threshold, which will result in a 1 or a 0, which can then be displayed with the Debug Terminal.

Analog to Binary Light Sensor Parts

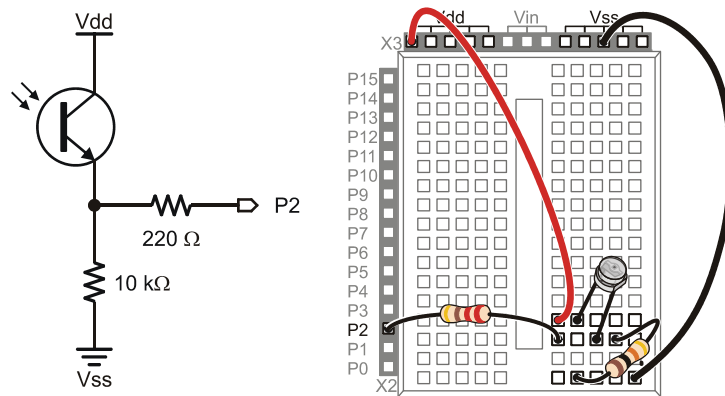
- (1) Phototransistor
- (1) Resistor – 220Ω (red-red-brown)
- (1) Resistor – $10 \text{ k}\Omega$ (brown-black-orange)
- (1) Resistor – $2 \text{ k}\Omega$ (red-black-red)
- (1) Resistor – $4.7 \text{ k}\Omega$ (yellow-violet-red)
- (1) Resistor – $100 \text{ k}\Omega$ (brown-black-yellow)
- (2) Jumper wires

Analog to Binary Light Sensor Circuit

With the circuit shown in Figure 7-20, the circuit behaves like a shadow controlled pushbutton. Shade results in **in2 = 0**, bright light results in **in2 = 1**. Keep in mind that an I/O pin set to input does not affect the circuit it monitors because it doesn't source or sink any current. This makes both the I/O pin and 220 Ω resistor essentially invisible to the circuit. So, the voltage results of our circuit calculations from the previous section will be the same with or without the 220 Ω resistor and I/O pin connected.

- ✓ Build the circuit shown in Figure 7-20.

Figure 7-20: Schematic and Wiring Diagram for Analog to Binary Light Sensor Circuit connected to an I/O pin



Analog to Binary Light Sensor Test Code

TestBinaryPhototransistor.bs2 is a modified version of ReadPushbuttonState.bs2 from Chapter 3, Activity #2. Aside from adjusting the comments, the one change to the actual program is the **DEBUG ? IN2** line, which was **DEBUG ? IN3** in the pushbutton example program because the pushbutton was connected to P3 instead of P2.

- ✓ Review Chapter 3, Activity #2 (page 65).
- ✓ Use TestBinaryPhototransistor.bs2 below to verify that bright light on the phototransistor results in a 1 while darkness results in 0. You might need pretty bright light. If your indoor lighting still results in a 0, try sunlight or a flashlight up close. An alternate remedy for low lighting is to replace the 10 kΩ resistor with a 100 kΩ resistor.

```
' What's a Microcontroller - TestBinaryPhototransistor.bs2
' Check the phototransistor circuit's binary output state every 1/4 second.

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

DO
  DEBUG ? IN2
  PAUSE 250
LOOP
```

Testing Series Resistance

Take a look at the $V = I \times R$ calculations earlier in this activity. If the series resistor is $1/5$ the value, the voltage across the resistor will be $1/5^{\text{th}}$ the value for the same light conditions. Likewise, a resistor that is 10 times as large will cause the voltage to be ten times as large.

What does this do for your circuit? A 100 k Ω resistor in place of a 10 k Ω resistor means the phototransistor only has to conduct $1/10^{\text{th}}$ the current to cross the BASIC Stamp I/O pin's 1.4 V threshold, which in turn means it takes less light to get trigger a binary 1 in the I/O pin's input register. This might work as a sensor in an environment that is supposed to stay dark since it will be sensitive to small amounts of light. In contrast, $1/5$ the resistance value means that the phototransistor has to conduct 5 times as much current to get the voltage across the resistor to cross the 1.4 V threshold, which in turn means that it takes more light to trigger the binary 1 in the I/O pin's input register. So, this circuit would be better for detecting brighter light.

- ✓ Experiment with 2 k Ω , 4.7 k Ω , 10 k Ω , and 100 k Ω resistors and compare the changes in sensitivity to light with each resistor.

Your Turn – Low Light Level Indicator

- ✓ Choose a resistor with the best 1/0 response to low light levels in your work area.
- ✓ Add the LED featured in Chapter 3, Activity #3 to your phototransistor threshold circuit.
- ✓ Put something between the LED and phototransistor so that the phototransistor cannot “see” the LED. This eliminates potential crosstalk between the two devices.
- ✓ Modify the program so that it makes the light blink when a shadow is cast over the phototransistor.

ACTIVITY #6: FOR FUN—MEASURE OUTDOOR LIGHT WITH AN LED

As mentioned earlier, the circuit introduced in Activity #1 is designed for indoor light measurements. What if your application needs to take light measurements outdoors? One option would be to find a phototransistor that generates less current for the same amount of light. Another option would be to use one of the other light sensors in the What's a Microcontroller kit. They are disguised as LEDs, and they perform particularly well for bright light measurements.

When electric current passes through the LED, it emits light, so what do you think happens when light shines on an LED? Yes indeed, it can cause electric current to flow through a circuit. Figure 7-21 shows an LED circuit for detecting light levels outdoors, and in other very brightly light areas. While the phototransistor allows current to pass through provided electrical pressure (voltage) is applied, the LED is more like a tiny solar panel and it creates its own voltage to supply the current. As far as the RC decay circuit is concerned, the result with an LED is about the same. The LED conducts more current and drains the capacitor of its charge more quickly with more light, and it conducts less current and drains the capacitor less quickly with less light.

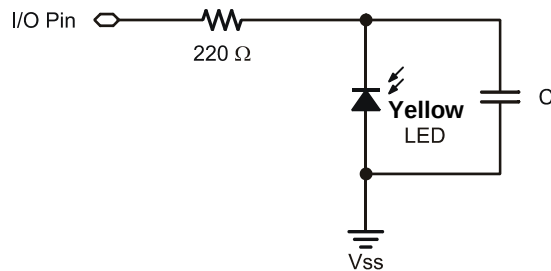


Figure 7-21
Schematic for LED in
Light-Sensing RC-Time
Circuit



Why is the LED plugged in backwards? In Chapter 2, the LED's anode was connected to the 220 Ω resistor, and the cathode was connected to ground. That circuit made the LED emit light as a result of electric current passing through the LED when voltage was applied to the circuit. When light is shining on the LED, it will create a small voltage that generates a small current in the opposite direction. So, the LED has to be plugged in backwards so that the current it conducts allows the capacitor to drain through it for RC decay measurements.

LED Light Sensor Parts

- (1) LED – yellow
- (1) LED – green
- (1) LED – red
- (1) Resistor – $220\ \Omega$ (red-red-brown)
- (1) Jumper wire

LED Light Sensor Circuit

One major difference between the LED and phototransistor is that the LED conducts much less current for the same amount of light, so it takes very bright light for the LED to conduct enough current to discharge the capacitor quickly enough for the **RCTIME** measurement. Remember that the maximum time measurement the **RCTIME** can measure is $65535 \times 2\ \mu\text{s} \approx 131\ \text{ms}$. So for good RC decay measurements with the BASIC Stamp, a much smaller capacitor is needed. In fact, the circuit works better without any external capacitor. The LED has a small amount of capacitance inside it, called *junction capacitance*, and the metal clips that hold wires you plug into the breadboard also have capacitance. Reason being, a capacitor is two metal plates separated by an insulator called a dielectric. So two metal clips inside the breadboard separated by plastic and air forms a capacitor. The combination of the LED's junction capacitance and the breadboard's clip capacitance makes it so that you can use the LED without any external capacitor, as shown in Figure 7-22.

- ✓ Build the circuit shown in Figure 7-22 and Figure 7-23, using the yellow LED. Make sure to observe the polarity shown in the figures!

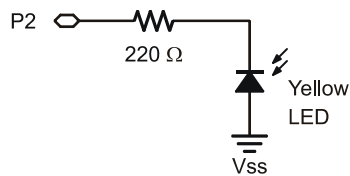


Figure 7-22
LED RCTIME Circuit
Schematic

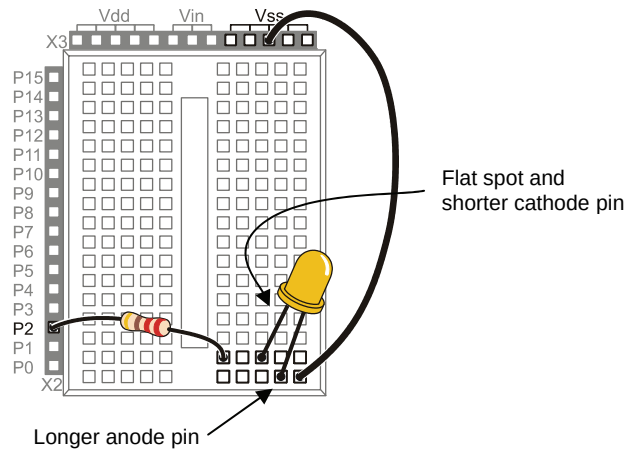


Figure 7-23
LED RCTIME Circuit
Wiring Diagram

7

Testing LED Light Sensor with Code

The LED light sensing circuit can be tested in a brightly lit room or outdoors during the day. In a dimly lit room the measurement times are likely to exceed 65535, in which case **RCTIME** will store zero in the **result** variable. For most situations, the code is the same code from Activity #1, TestPhototransistor.bs2.

If you are in a brightly lit room try this:

- ✓ Run TestPhototransistor.bs2 from Activity #1.
- ✓ Point the LED at the brightest light source by facing your board toward it.
- ✓ Gradually rotate the board away from the brightest light source in the room; the values displayed by the Debug Terminal should get larger as the light gets dimmer.

If you have a bright flashlight, try this:

- ✓ Run TestPhototransistor.bs2 from Activity #1.
- ✓ Eliminate most bright light sources such as sunlight streaming into the windows.
- ✓ Turn on the flashlight and point it into the top of the LED at a distance of about 4 inches (about 10 cm). If possible, turn off some of the fluorescent lights so that the ambient light levels are low.
- ✓ Watch the measurements the Debug Terminal displays as you gradually increase the distance of the flashlight from the top of the LED. It will allow you to determine the flashlight's distance from the LED.

If you are in a room with only fluorescent lights and no bright light sources:

- ✓ Run `TestPhototransistor.bs2` from Activity #1.
- ✓ Eliminate most bright light sources such as sunlight streaming into the windows. If possible, turn off some of the fluorescent lights so that the light levels are low.
- ✓ Point the LED into your computer monitor so that it is almost touching the monitor, and see if the measurements make it possible to distinguish between various colors on the display.

Outdoor tests:

- ✓ Run `StoreLightMeasurementsInEeprom.bs2` from Activity #2.
- ✓ Disconnect the programming cable and take your board outside.
- ✓ Face your board so that the LED is pointing directly at the sun.
- ✓ Press and release your board's Reset button to restart the datalogging program.
- ✓ Gradually rotate your board away from the sun over 2 ½ minutes.
- ✓ Take your board back inside and reconnect to the PC.
- ✓ Run `ReadLightMeasurementsFromEeprom.bs2`, and examine the light measurements. Since you gradually turned the LED away from the sun, successive measurements should get larger.

Your Turn – Can your BASIC Stamp Tell Red from Green?

In Figure 7-2, green is in the middle of the spectrum, and red is to the right. If you download the color PDF version of this textbook from www.parallax.com, you can place the green and then the red LED against the screen and record light measurements across the color spectrum. Then, by comparing the lowest measurements with each LED, you can detect whether the LED is placed against green or red on the screen.

- ✓ Start with a green LED in the Figure 7-22 and Figure 7-23 light detection circuit.
- ✓ Download the PDF version of *What's a Microcontroller?* from www.parallax.com/go/WAM.
- ✓ Display the color spectrum shown in Figure 7-2 (page 197) on your monitor, and zoom in on the image.
- ✓ With the `TestPhototransistor.bs2` program displaying measurements in the Debug Terminal, hold your board so that the green LED's dome is pointing directly into the monitor over the color spectrum. For best results, the dome of the LED should be just barely touching the monitor, and the light levels in the room should be fairly low.

- ✓ Slide the green LED slowly along the spectrum bar displayed on the monitor, and note which color resulted in the lowest measurement.
- ✓ Repeat with the red LED. Did the red LED report its lowest measurements over red while the green LED reported its lowest measurements over green?

The lowest red LED measurements should occur over the red color on the display, and the lowest measurements for the green LED should occur over green.

SUMMARY

This chapter introduced light sensors and described how they are used in a variety of products. Different light sensors detect different kinds of light, and their datasheets describe their sensitivities in terms of light's wavelength. This chapter focused on the phototransistor, a device that controls the current through its collector and emitter terminals by the amount of light shining on its base terminal. Because light can control the amount of current a phototransistor conducts, the technique for measuring the position of a potentiometer's knob in the Chapter 5 RC circuit also works for measuring the light shining on a phototransistor. The time it takes for a capacitor to lose its charge through the phototransistor results in an **RCTIME** measurement that provides a number that corresponds to the brightness of the light shining on the phototransistor.

7

Datalogging by storing light measurements in the unused portion of the BASIC Stamp module's EEPROM program memory was also introduced. **WRITE** and **READ** commands were used to store values to and retrieve values from the BASIC Stamp module's EEPROM. The volume of numbers involved in datalogging can be difficult to analyze, but graphing the data makes it a lot easier to see patterns, trends and events. Logged data can be transferred to conventional spreadsheets and graphed, and certain graphing utilities can even stand in for the Debug Terminal, and plot the values the BASIC Stamp sends instead of displaying them as text. A light meter example application was also developed, which demonstrated how light measurements can be used to control another process, in this case, the rate of a circular pattern displayed by a 7-segment LED. This application also used subroutines to perform three different jobs for the light meter application.

The BASIC Stamp can be programmed to convert an RC decay time measurement to a binary value with **IF...THEN** statements. Additionally, the program can take a range of RC decay measurements and apply hysteresis with a "light on" threshold that's in the darker range of measurements and a "light off" threshold that's in the lighter range. This can help prevent on/off oscillations that might otherwise occur when the sensor reports

darkness and the device turns on an area light. Without hysteresis, the device might sense this light, turn back off, and repeat this cycle indefinitely.

A hardware approach to sensing on/off light states is applying power to the phototransistor in series with a resistor. In keeping with Ohm's Law, the amount of current the phototransistor conducts affects the voltage across the resistor. This variable voltage can be connected to an I/O pin, and will result in a binary 1 if the voltage is above the I/O pin's 1.4 V threshold, or a binary 0 if it's below the threshold.

The LED (light emitting diode) that emits light when current passes through it also behaves like a tiny solar panel when light strikes it, generating a small voltages which in turn can cause electric current in circuits. The currents the LED generates are small enough that a combination of the LED's own junction capacitance and the capacitance inherent to the clips inside the breadboard provides enough capacitance for an RC decay circuit with no external capacitor. While the phototransistor in the What's a Microcontroller kit performs better indoors, the LED is great for outdoor and bright light measurements.

Questions

1. What are some examples of automatic lighting applications that depend on ambient light sensors?
2. What are some examples of products that respond to changes in the brightness of the ambient light?
3. What wavelength range does the visible light spectrum fall into?
4. What are the names of the phototransistor's terminals, and which one controls how much current the device allows through?
5. What does EEPROM stand for?
6. How many bytes can the BASIC Stamp module's EEPROM store? How many bits can it store?
7. 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?
8. What is a label?
9. What is a subroutine?
10. What command is used to call a subroutine? What command is used to end a subroutine?

Exercises

1. Draw the schematic of a phototransistor RC-time circuit connected to P5.
2. Modify TestPhototransistor.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.

Projects

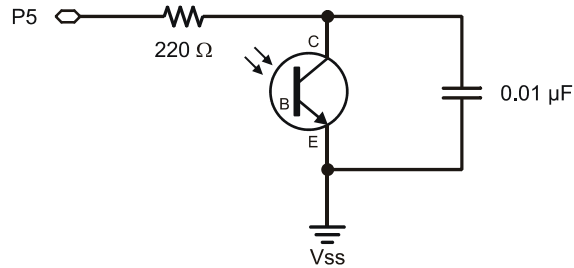
1. Make a small prototype of a system that automatically closes the blinds when it gets too bright and opens them again when it gets less bright. Use the servo for a mechanical actuator. Hint: For code, you can add two servo control commands to PhototransistorAnalogToBinary.bs2, and change the **PAUSE 100** command to **PAUSE 1**. Make sure to follow the instructions in the text for calibrating for area light conditions before you test.
2. For extra credit, enhance your solution to Project 1 by incorporating the hysteresis modifications discussed in Activity #5.

7

Solutions

- Q1. Car headlights, streetlights, and outdoor security lights that automatically turn on when it's dark.
- Q2. Laptop displays and cameras with auto exposure.
- Q3. 380 nm to 750 nm according to Figure 7-2 on page 197.
- Q4. Collector, base, and emitter. The base controls how much current passes into the collector and back out of the emitter.
- Q5. Electrically Erasable Programmable Read-Only Memory.
- Q6. 2048 bytes. $2048 \times 8 = 16,384$ bits.
- Q7. To store a value – **WRITE** ; To retrieve a value – **READ**; The **READ** command requires a variable.
- Q8. A label is a name that can be used as a placeholder in a PBASIC program.
- Q9. A subroutine is a small segment of code that does a certain job.
- Q10. Calling: **GOSUB**; ending: **RETURN**

E1. Schematic based on Figure 7-4 on page 200, with P2 changed to P5.



E2. The required changes are very similar to those explained on page 200.

```

D0
  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. Here are two solutions made with alternative **Update_Display** subroutines.

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

```

Index = 5 '<<Add after Index variable
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. Phototransistor from Figure 7-4 on page 200, servo schematic for your board from Chapter 4, Activity #1.

```
' What's a Microcontroller - Ch07Prj01_Blinds_Control.bs2
' Control servo position with light.

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

valMax      CON      4000
valMin      CON      100
time        VAR      Word

PAUSE 1000

DO

  HIGH 2
  PAUSE 1                                ' PAUSE 100 -> PAUSE 1
  RCTIME 2, 1, time
  DEBUG HOME, "time = ", DEC5 time

  time = time MAX valMax MIN valMin

  IF time > (valMax - valMin) / 2 THEN
    DEBUG CR, "Open blinds "
    PULSOUT 14, 500                      ' Modify
                                         ' Add
  ELSE
    DEBUG CR, "Close blinds"
    PULSOUT 14, 1000                     ' Modify
                                         ' Add
  ENDIF

LOOP
```

7

P2. Hysteresis functionality added for extra credit:

```
' What's a Microcontroller - Ch07Prj02_Blinds_Control_Extra.bs2
' Control servo position with light including hysteresis.

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

valMax      CON      4000
valMin      CON      100

time        VAR      Word

PAUSE 1000

DO
```

```
HIGH 2
PAUSE 1                                ' PAUSE 100 -> PAUSE 1
RCTIME 2, 1, time
DEBUG HOME, "time = ", DEC5 time

time = time MAX valMax MIN valMin

IF time > (valMax - valMin) / 4 * 3 THEN
    DEBUG CR, "Open blinds "          ' Modify
    PULSOUT 14, 500                   ' Add
ELSEIF time < (valMax - valMin) / 4 THEN
    DEBUG CR, "Close blinds"          ' Modify
    PULSOUT 14, 1000                  ' Add
ENDIF

LOOP
```

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 resembling 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. You may have started your day with a beeping alarm clock.

MICROCONTROLLERS, SPEAKERS, 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. A piezoelectric speaker is commonly referred to as a piezo speaker or piezo buzzer, and piezo is pronounced “pE-A-zO.” Its schematic symbol and part drawing are shown in Figure 8-1.

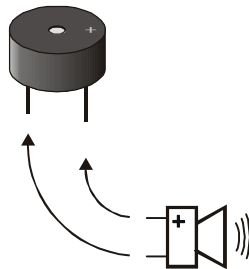


Figure 8-1
Piezoelectric Speaker Part Drawing
and Schematic Symbol

ACTIVITY #1: BUILDING AND TESTING THE SPEAKER

In this activity, you will build and test the piezoelectric speaker circuit.

Speaker Circuit 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.

- ✓ If your piezo speaker has a sticker on it, just remove it (no washing needed).
- ✓ Build the circuit shown in Figure 8-2.

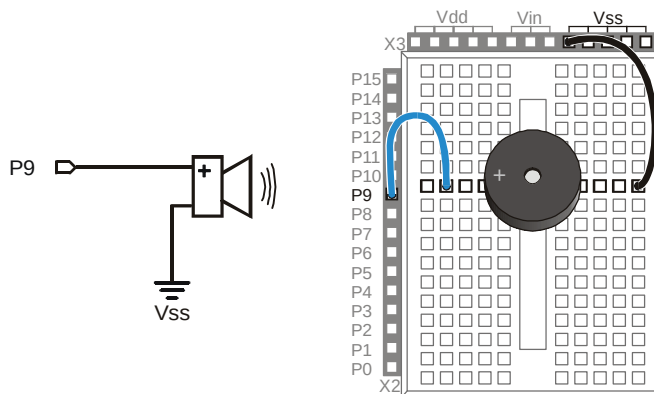


Figure 8-2
Piezoelectric Speaker
Circuit Schematic and
Wiring Diagram

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 play two different tones at the same time.

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
```

Example Program: TestPiezoWithFreqout.bs2

8

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}

PAUSE 1000

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** arguments.
- ✓ 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 playing 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 it. 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

PAUSE 1000

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 duration of 0.5 s with fixed delays 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** variables, 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 **FOR 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 **FOR frequency...** loop is finished, the **FOR duration...** loop has only repeated one of its 15 passes. So it subtracts 1 from the value of **duration** and repeats the **FOR 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** variables change each time through the loop.

```

' What's a Microcontroller - NestedLoops.bs2
' Demonstrate how the nested loop in ActionTones.bs2 works.

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

duration      VAR      Word
frequency     VAR      Word

PAUSE 1000

DEBUG "Duration    Frequency", CR,
      "-----", CR

FOR duration = 4000 TO 1000 STEP 1000
  FOR frequency = 1000 TO 3000 STEP 500
    DEBUG "    ", DEC5 duration,
          "    ", DEC5 frequency, CR
    FREQOUT 9, duration, frequency
  NEXT
  DEBUG CR
NEXT
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:

```
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 creation.

8

Two Frequencies at Once

You can play two frequencies at the same time. Remember the **FREQOUT** command’s syntax from Activity #1:

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

You can use the optional **Freq2** argument to play two frequencies with a single **FREQOUT** command. For example, you can play 2 kHz and 3 kHz like this:

```
FREQOUT 9, 1000, 2000, 3000
```



Each touchtone keypad tone is also an example of two frequencies combined 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. Look up this command in the BASIC Stamp Manual or Help for examples.

Example Program: PairsOfTones.bs2

This example program demonstrates the difference in tone that you get when you play 2 and 3 kHz together. It also demonstrates an interesting phenomenon that occurs when you add two sound waves that are very close in frequency. When you play 2000 Hz and 2001 Hz at the same time, the tone will fade in and out once every second (at a frequency

of 1 Hz). If you play 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 PairsOfTones.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 - PairsOfTones.bs2
' Demonstrate some of the things that happen when you mix two tones.

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

PAUSE 1000

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
```

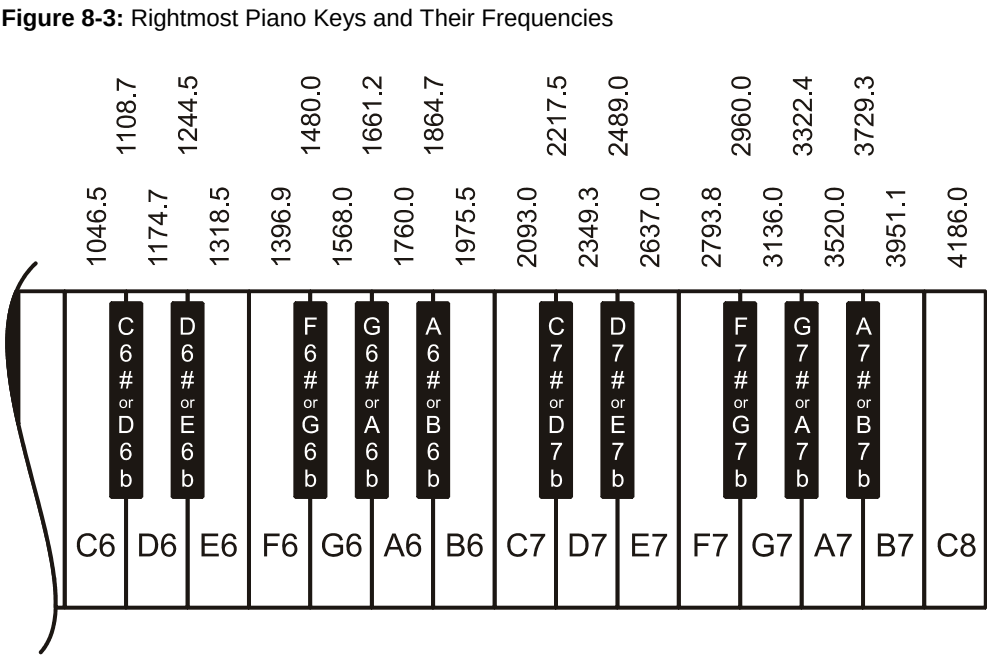
Your Turn – Condensing the Code

PairsOfTones.bs2 was written to demonstrate some interesting things that can happen when you play two different frequencies at once using the **FREQOUT** command’s optional **Freq2** argument. However, it is extremely inefficient.

- ✓ Modify PairsOfTones.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, made up of 7 white keys and 5 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.

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*, and the name "octave" relates to the frequency doubling with every eighth natural key. 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.



Internet search for "musical scale": By using the words "musical scale" 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.

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 notes using a piezoelectric speaker.
'{$STAMP BS2}
'{$PBASIC 2.5}
PAUSE 1000

'Solfege          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
```

8

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 *** 2** operation after each **Freq1** argument. For example, **FREQOUT 9, 500, 1175 * 2** will multiply D6 by 2 to 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} DataItem {, {Word} DataItem, ... }

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
```




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.

```
' 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
```

```

PAUSE 1000

DEBUG    "Note  Duration  Frequency", CR,
        "-----  -----  -----", 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.



More about microcontroller music: After completing this activity, you will be ready to learn how to write PBASIC musical code from sheet music. See the *Playing Sheet Music with the Piezospeaker* tutorial and its accompanying video primer at www.parallax.com/go/WAM.

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 durations. 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 one 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.

8

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 UNTIL...LOOP** checks for "Q" each time through the loop and will repeat until it is found.

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,  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** argument for 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 **noteDuration** starts out as 4 from a **READ** command, it becomes $2000 \div 4 = 500$. If **noteDuration** is 8, it becomes $2000 \div 8 = 250$.

```
noteDuration = WholeNote / noteDuration
```

Now that **noteDuration** and **noteFreq** are determined, the **FREQOUT** command plays the note.

```
FREQOUT 9, noteDuration, noteFreq
```

8

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
```

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 decrease 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 as 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. Here are three more features we can add to a music-playing program:

- Play "dotted" notes
- Determine the whole note duration from the tempo
- 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
```

Music-playing programs typically express the tempo for a song in beats per minute. This is the same as saying quarter notes per minute.

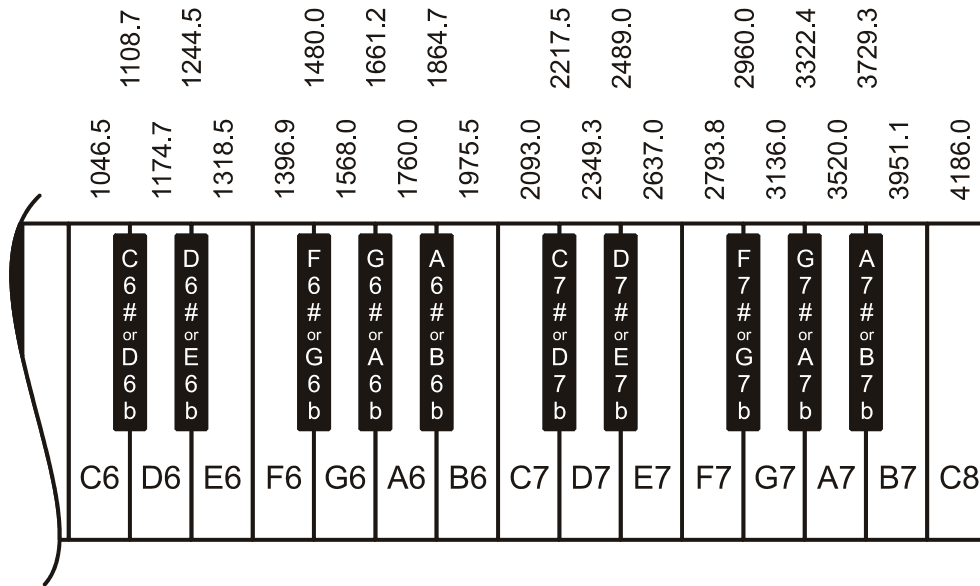
```
BeatsPerMin CON    200
```

Figure 8-4 is a repeat of Figure 8-3 from page 253. 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,  6,  7,  6,
                        6,  6,  6
```

Figure 8-4: Rightmost Piano Keys and Their Frequencies



8

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
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 **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)
```

8



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.

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 a copy of the program as 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, 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: RINGTONES WITH RTTTL

Older cell phones used to play ringtones with a piezospeaker. Ringtones were downloaded from the web to a computer, and then uploaded from the computer to the cell phone's microcontroller. At the time, one of the most widely used ways of composing, recording and posting ringtones was one that featured strings of text with characters that describe each note in the song. Here is an example of how the first few notes from “Beethoven's 5th” look in one of these strings:

Beethoven5:d=8,o=7,b=125:g,g,g,2d#,p,f,f,f,2d

8

This format for storing musical data is called RTTTL, which stands for Ringing Tone Text Transfer Language. The great thing about RTTTL files at the time was that they were widely shared via the World Wide Web. Many sites had RTTTL files available for free download. There were also free software programs that could be used to compose and emulate these files as well as upload them to your cell phone. The RTTTL specification is still published online. Appendix C 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 an 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)
{ CASE ELSE
    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 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 Transmit windowpane in the Debug Terminal to send the characters you type to the BASIC Stamp. The Transmit and Receive windowpanes are shown in Figure 8-5.

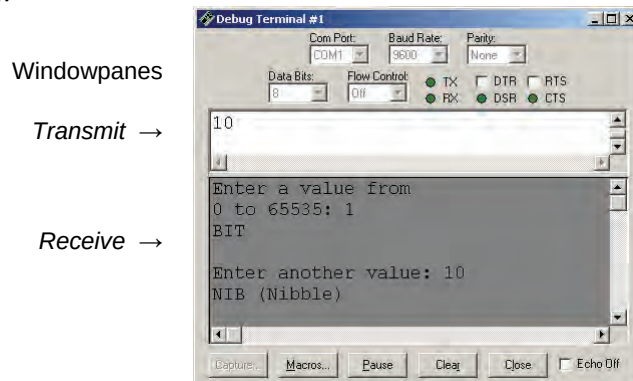


Figure 8-5
Sending Messages
to the BASIC Stamp

*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.
- ✓ In the Debug Terminal, make sure that the Echo Off checkbox is clear (no checkmark).

- ✓ 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 8-1 to verify that the example program makes the right decisions about the size of the numbers you enter into the Debug Terminal.

Table 8-1: 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

```
' 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
PAUSE 1000

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.
- ✓ Make sure the Echo Off checkbox is clear (no checkmark).
- ✓ Click the Debug Terminal's Transmit windowpane to place the cursor there.
- ✓ Enter characters into the Transmit windowpane 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
PAUSE 1000

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. We can treat the following groups of characters and punctuation marks the same way as a range of numbers, since the BASIC Stamp recognizes them by their ASCII numeric equivalents—see the ASCII chart in the BASIC Stamp Editor Help.

```
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 Special Characters

- ✓ 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/go/WAM page. 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.

8

- ✓ With the BASIC Stamp Editor, 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).

```
' 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    PIN    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:
DO
    READ RTTTL_File + counter, char
    counter = counter + 1
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:
default_b = 0
DO
    READ RTTTL_File + counter, char
    IF char = ":" THEN
        default_b = default_b / 10
        counter = counter + 1
        EXIT
    ENDIF
    default_b = default_b + char - 48
    counter = counter + 1
    default_b = default_b * 10
LOOP UNTIL char = ":"

RETURN

' -----[ Subroutine - Look up Octave ]-----

ProcessOctave:
    READ RTTTL_File + counter, char
    SELECT char
        CASE "5" TO "8"
            noteOctave = char - "0"
            counter = counter + 1
        CASE ELSE
            noteOctave = default_o
    ENDSELECT
    IF default_o = 0 THEN
        default_o = noteOctave
    ENDIF

```

' Octave may or may not be included in a given note because any note that is played in the default octave does not specify the octave. If a char from '5' to '8' then use it, else use default_o. Characters are converted to digits by subtracting '0', which is the same as subtracting 48. The first time this subroutine is 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
                                      ' note character. If 'p',
                                      ' index is 0. If 'a' to 'g',
                                      ' read character values in
                                      ' DATA table and find match.
                                      ' Record index value when
                                      ' match is found. If next
                                      ' char is a sharp (#), add
                                      ' 1 to the index value to
                                      ' increase the index (and
                                      ' frequency) by 1 notch.
                                      ' As with other subroutines,
                                      ' increment counter for each
                                      ' character that is processed.
READ RTTTL_File + counter, char
SELECT char
CASE "p"
    index = 0
    counter = counter + 1
CASE "a" TO "g"
    FOR index = 1 TO 12
        READ Notes + index, noteLetter
        IF noteLetter = char THEN EXIT
    NEXT
    counter = counter + 1
READ RTTTL_File + counter, char
SELECT char
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.
                                      ' If yes, then convert from
                                      ' ASCII character to a value
                                      ' by subtracting 48. In the
                                      ' case of 16 or 32, multiply
                                      ' by 10 and add the next
                                      ' digit to the ones column.
READ RTTTL_File + counter, char
SELECT char
CASE "1", "2", "3", "4", "8"
    duration = char - 48
    counter = counter + 1
READ RTTTL_File + counter, char
SELECT char
CASE "6", "2"
    duration = duration * 10 + char - 48
    counter = counter + 1
ENDSELECT
CASE ELSE
    duration = default_d
ENDSELECT
                                      ' If no duration, use
                                      ' 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, except one!

Here is a list of the elements in this program 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
- The **SELECT . . . CASE** command
- **EXIT** is new, but it simply allows the program to “exit” a loop before it is finished, and is often used in **IF . . . THEN** statements.

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,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,b6,c,"
                  "d,b6,a6,g6"
```



Downloading RTTTL Files: There are 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 with 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, and the frequency of the tone (*Freq1*). The optional *Freq2* argument can be used to play two tones at once.

8

Sound effects can be made by adjusting the frequency and duration of tones and the pauses between them. The value of the frequency can also be swept across a range of values 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. In this chapter's examples, the **READ** command's *Location* 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 frequencies 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 older 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. 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 a 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 P15 for one second (1000 ms). The effect of each number: 15 – I/O pin P15; 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 play 3000 Hz and say, 2000 Hz, we simply add the second frequency to the command, after a comma:

```
FREQOUT 15, 1000, 3000, 2000
```

8

- Q4. 1975.5 Hz, see Figure 8-3 on page 253.
- Q5. Use the optional **Word** modifier before each data item.
- Q6. Yes. Each **DATA** directive can have a different optional **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. 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 on page 246; pushbutton circuits from Figure 4-26 on page 130.

```
' 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
```

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 is a microcontroller pre-programmed with the PBASIC Interpreter that 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*.

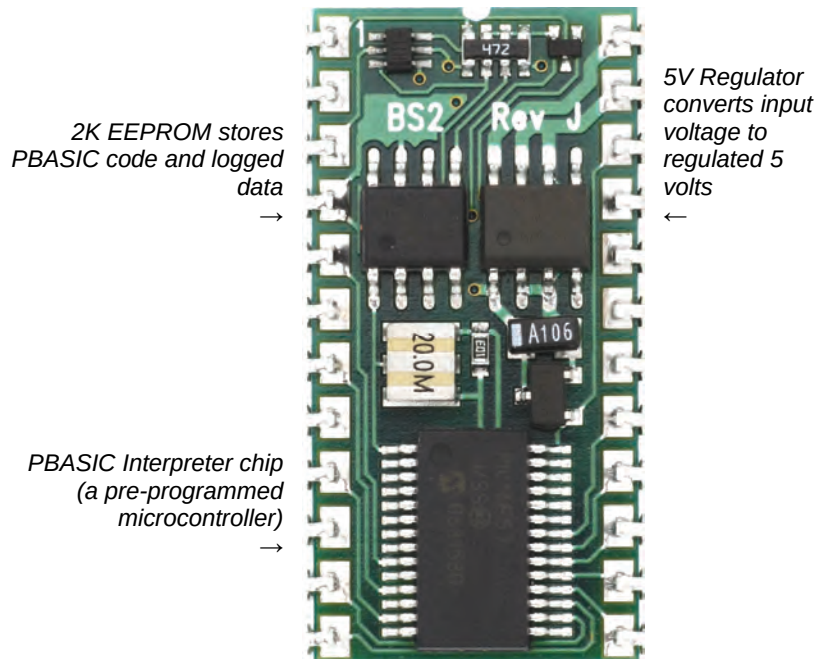


Figure 9-1
Integrated
Circuits on
the BASIC
Stamp 2

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 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 almost 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* and make them available on their web sites. 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.

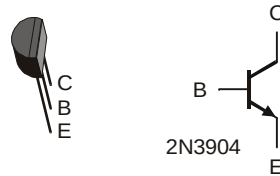


Figure 9-2
2N3904 Transistor

9



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 information. Follow it and look for a 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.
 - Make sure that the LED's anode (longer) pin is connected to Vdd.
 - Double-check your transistor circuit. Note that the transistor's flat side is facing to the right in the wiring diagram.
- ✓ 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.

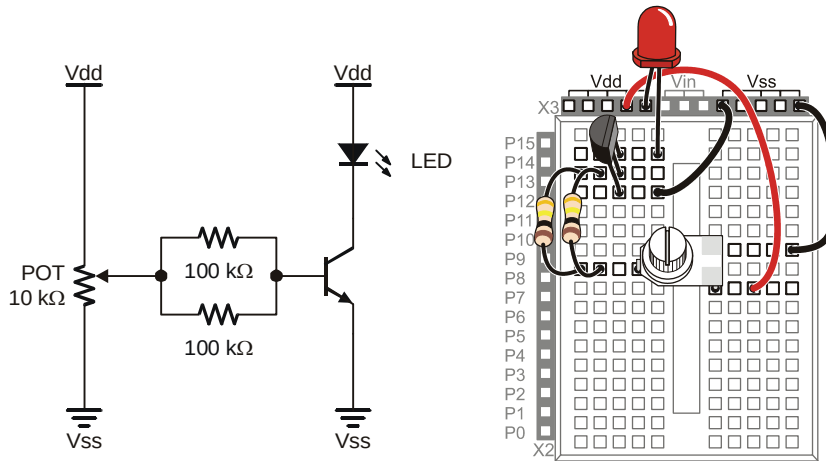


Figure 9-3
Manual
Potentiometer-
Controlled
Transistor
Circuit

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 *Process 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 P14. 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.

9

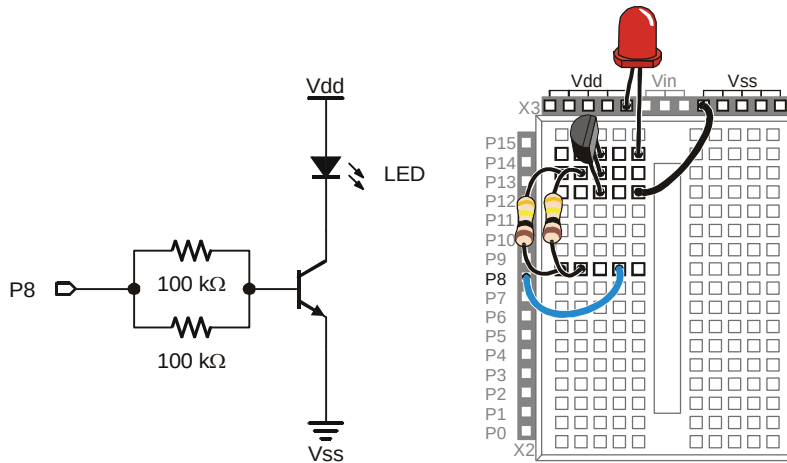


Figure 9-4
Circuit giving
BASIC Stamp
On/Off Control
of Current to
LED with a
Transistor

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.

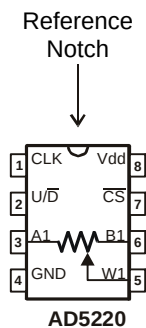


Figure 9-5
AD5220 Pin Map

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
- (10) Jumper wires

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.

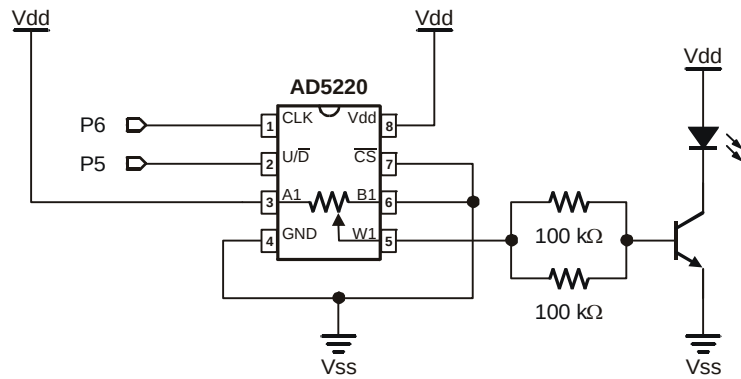


Figure 9-6
Digital Potentiometer
Controlled Transistor
Circuit Schematic

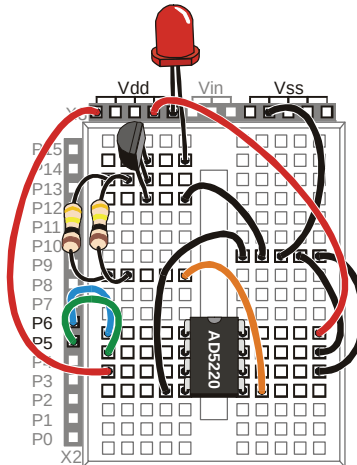


Figure 9-7
Wiring Diagram for
Figure 9-6

Programming Digital Potentiometer Control

Imagine that the knob on the manual potentiometer from the previous exercise has 128 positions. Imagine also that the potentiometer is in the middle of its range of motion. That means you could rotate the knob one direction by 63 steps and the other direction by 64 steps.

Let's say you turn the potentiometer's knob one step clockwise. The LED will get only slightly brighter. This would be the same as sending a high signal to the AD5220's U/D pin and sending one pulse (high-low-high) to the CLK pin.

```
HIGH 5
PULSOUT 6, 1
```

Imagine next that you turn your manual potentiometer 3 steps counterclockwise. The LED will get a little bit dimmer. This would be the same as sending a low signal to the U/D pin on the AD5220 and sending three pulses to the CLK pin.

```
LOW 5
FOR counter = 1 TO 3
  PULSOUT 6, 1
  PAUSE 1
NEXT
```

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
```

9

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.

- ✓ 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 by that name shown below.
- ✓ Run the program and verify that it functions the same way 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 some 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
```

9

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 $10\ \text{k}\Omega$.

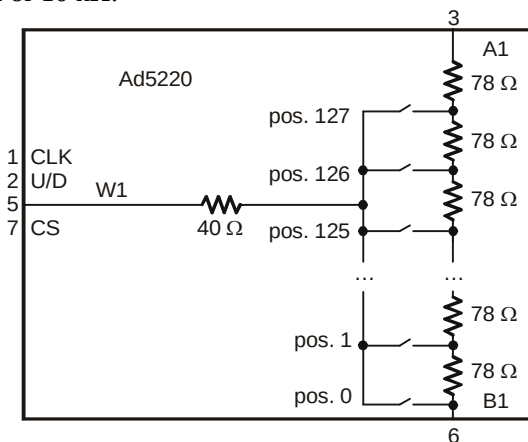


Figure 9-8
Inside the AD5220



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 Ω , with a tolerance of 30% (23.438 Ω) 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.

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 – **DATA** declarations that store predefined lists of values in portions of EEPROM memory that are not needed for program storage
- I/O Definitions – **PIN** directives that name I/O pins
- Constants – **CON** declarations that name values in the program
- Variables – **VAR** declarations that assign names to portions of BASIC Stamp's RAM memory for storing values
- 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.

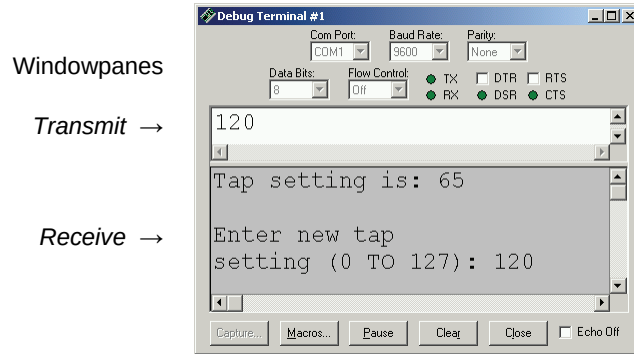


Figure 9-9
Sending Messages
to the BASIC Stamp

*Click the Transmit
(upper) windowpane
and enter the
numbers for the new
tap setting.*

9

- ✓ Enter and run TerminalControlledDigitalPot.bs2.
- ✓ Make sure the Echo Off box is clear (no checkmark).
- ✓ Click the Debug Terminal's Transmit windowpane to place the cursor there.
- ✓ 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      PIN    5          ' Set values of I/O pins
ClkPin     PIN    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
PAUSE 1000                                           ' Wait 1 s before 1st message

' -----[ 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 - Advanced Challenge

1. Add a phototransistor to your project and cause the brightness of the LED to adjust with the brightness seen by the phototransistor. Note: the solution given is worth reading, as it demonstrates a useful approach to scaling an input to another output.

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      PIN      5          ' Set values of I/O pins
ClkPin     PIN      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
```

9

- P1. Use the digital potentiometer circuit from Figure 9-6 on page 294 and the phototransistor circuit from Figure 7-4 on page 200.

This solution builds on TerminalControlledDigitalPot.bs2, and incorporates elements from PhototransistorAnalogToBinary.bs2 from Chapter 7, Activity #5. It also applies some algebra to solve a scaling problem that makes the range of values you could get from the phototransistor **RCTIME** measurement fit into a range of 0 to 128 for the digital potentiometer. Keep in mind that this is one example solution, and by no means the only solution or approach.

The **GOSUB Get_New_Tap_Setting** subroutine call from the program TerminalControlledDigitalPot.bs2 is replaced by two other subroutine calls: **GOSUB Read_Phototransistor** and **GOSUB Scale_Phototransistor**. Likewise, the **Get_New_Tap_Setting** subroutine is replaced by **Read_Phototransistor** and **Scale_Phototransistor** subroutines. **Read_Phototransistor** is a subroutine version of the commands that take the phototransistor **RCIME** measurement and limit its input range in

PhototransistorAnalogToBinary.bs2. The pin, constant and variable names have been adjusted, and the **PAUSE 100** for a 10-times-per-second display was changed to **PAUSE 1**, which is all that's needed to charge the capacitor before taking the **RCTIME** measurement. After this subroutine stores a value in the **lightReading** variable, it will be somewhere between **valMin** (100) and **valMax** (4000). Make sure to test and adjust these values for your own lighting conditions.

The problem we now have is that there are only 128 tap settings, and 3900 possible phototransistor **RCTIME** measurements. To fix this, we need to divide the phototransistor **RCTIME** measurement by some value to make it fit into the 0 to 127 range. So, we know we need to divide the range of input values by some value to make it fit into 128 values. It looks like this in an equation:

$$\frac{\text{Range of Possible Phototransistor Measurements}}{\text{Scale Divisor}} = 128 \text{ Possible Tap Settings}$$

To solve this, multiply both sides of the equation by Scale Divisor, and then divide both sides by 128 Possible Tap Settings.

$$\text{Scale Divisor} = \frac{\text{Range of Possible Phototransistor Measurements}}{128 \text{ Possible Tap Settings}}$$

In the code, the range of possible phototransistor measurements is **valMax - valMin**, **scaleDivisor** is a variable, and 128 is a constant. So, this code from the Declarations and Initialization section figures out the value of **scaleDivisor** like this:

```
scaleDivisor = (valMax - valMin) / 128
```

After every phototransistor **RCTIME** measurement, the **Scale_Phototransistor** subroutine subtracts **valMin** from **lightReading** and then divides the measurement by **scaleDivisor**. The result maps the 100 to 4000 input measurement range to a 0 to 127 output tap setting range.

```
Scale_Phototransistor:
    lightReading = (lightReading - valMin) / scaleDivisor
    RETURN
```


Assuming **ValMin** is 100 and **ValMax** is 4000, the **lightReading** variable could store 3900 possible values. What if the input range was **ValMin = 10,000** to **ValMax = 13,900**? When you subtract **ValMin = 10,000**, there are still 3900 possible values, and dividing **scaleDivisor** into it will correctly map the measurement to the corresponding digital pot tap setting. If your code didn't first subtract **ValMin**, the resulting scaled value would be completely out of the 0 to 128 range for the digital pot.

```
' What's a Microcontroller - Ch9Prj01_PhotoControlledDigitalPot.bs2
' Update digital pot's tap based on phototransistor reading
' {$STAMP BS2}
' {$PBASIC 2.5}

' -----[ Declarations and Initialization]-----
UdPin      PIN    5      ' Set values of I/O pins
ClkPin     PIN    6      ' connected to CLK and U/D.
PhotoPin   PIN    2      ' Phototransistor on pin P2
DelayPulses CON    10    ' Delay to observe LED fade.
DelayReader CON  2000
valMax     CON    4000    ' Max phototransistor val
valMin     CON    100     ' Min phototransistor val

counter     VAR    Byte   ' Counter for FOR...NEXT.
oldTapSetting VAR    Byte ' Previous tap setting.
newTapSetting VAR    Byte ' New tap setting.
lightReading VAR    Word  ' reading from phototransistor
scaleDivisor VAR    Word  ' For scaling values

' Set up a value that can be divided into the phototransistor RCTIME
' measurement to scale it to a range of 0 to 128

scaleDivisor = (valMax - valMin) / 128

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 ClkPin,5
  PAUSE 1
NEXT

PAUSE 1000             ' 1 sec. before 1st message

' -----[ Main Routine ]-----
DO
  GOSUB Read_Phototransistor
  GOSUB Scale_Phototransistor
```

```

    newTapSetting = lightReading MIN 1 MAX 127
    DEBUG HOME, DEC5 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
        HIGH UdPin                        ' tap values to decide
    ELSEIF newTapSetting < oldTapSetting THEN ' value of UdPin. Notify
        LOW UdPin                         ' user if values are
    ENDIF                                  ' equal.
    RETURN

Pulse_Clk_pin:
    FOR counter = oldTapSetting TO newTapSetting ' Deliver pulses
        PULSOUT ClkPin, 1                    ' from old to new
        PAUSE DelayPulses                    ' values.
    NEXT
    oldTapSetting = newTapSetting             ' Keep track of new and old
    RETURN                                  ' tapSetting values.

Read_Phototransistor:
    HIGH PhotoPin
    PAUSE 1
    RCTIME PhotoPin, 1, lightReading
    lightReading = lightReading MAX valMax MIN valMin
    RETURN

Scale_Phototransistor:
    lightReading = (lightReading - valMin) / scaleDivisor
    RETURN

```

Chapter 10: Prototyping Your Own Inventions

This text introduced the basics of integrating an onboard computer into projects and inventions. Common circuit ingredients in everyday products that you now have some experience with include: indicator lights, buttons, servos, dials, digital displays, light sensors, speakers, transistors, and other integrated circuits. You also now have experience connecting these circuits to the BASIC Stamp microcontroller and writing code to test each of them as well as integrate them into small applications.

At this point, you may be interested in using your new skills to invent something, or to learn more, or maybe both. What you have learned in this book can get you well down the road to making prototypes for a wide variety of inventions. In this chapter, we'll use a micro alarm system as an example prototype of a familiar device. Along the way, we'll cover some important prototyping techniques and habits, including:

- Suggestions for early development of your design ideas and inventions
- An example of how to build and test each sub-system in the prototype
- Examples of how to incorporate test code into the project code
- Good practices for code commenting and file versioning
- Examples of using familiar parts as stand-ins for devices with similar interfaces
- Tips and tricks for getting past design hurdles
- Where to go next to find more Stamps in Class projects and interesting devices

10

APPLY WHAT YOU KNOW TO OTHER PARTS AND COMPONENTS

The pushbutton circuit from Chapter 3 is an example of a very simple input device that converts a physical condition (whether or not someone has pressed a button) to a high or low signal the BASIC Stamp can detect and process. You have also used pushbuttons in applications that controlled light blinking, servo positions and speaker tones. There are many sensors that detect a physical condition other than “contact” that also send high or low signals a BASIC Stamp I/O pin can monitor. A few examples include gas, motion, and sound sensors, and there are many, many more. Since you now have experience making the BASIC Stamp monitor a pushbutton circuit, monitoring a sound or motion sensor is very similar, and certainly a reasonable next step.

Another technique from this book is measuring RC decay with the **RCTIME** command to sense potentiometer knob position and light levels with both a phototransistor and an LED. These examples are just the tip of the iceberg in terms of sensors you can use with

an RC decay circuit. Other examples include humidity, temperature, and pressure, and that's still just the beginning. The LED indicator light provides still another example circuit that's representative of a variety of circuits with different functions. The LED circuit is controlled by high/low BASIC Stamp I/O pin output. With additional support circuits, you can use high/low signals to run electric motors forwards and backwards, turn lights on/off, turn heating elements on/off, and more.

Now, think about all the other devices you have experimented with in this book. Each of them is just one example in a list of devices with similar interfaces that you can use to prototype all manner of inventions.

PROTOTYPING A MICRO SECURITY SYSTEM

In this chapter, we'll use parts from the What's a Microcontroller kit to make a very small security system prototype you could use in a desk, dresser, tool chest, or closet. It could come in handy for those of you who suspect siblings or coworkers of borrowing your stuff without asking. With this prototype, we'll also investigate other parts and components you could substitute in your security system that operate on the same principles as the familiar kit parts, but could give your system greatly enhanced functionality. From there, we'll look at how to find, understand, test and incorporate other parts that you may never have worked with before.

ACTIVITY #1: FROM IDEA TO PROOF OF CONCEPT

Many products start out as an idea, in some cases an invention that could be “really cool,” and in other cases it's something that solves a problem. This idea can be developed into a concept with drawings and specifications, and some early design work. The next step is typically to develop a working prototype. It might not be pretty, but it should reliably demonstrate that a device can be made that works according to the concept and specification. In companies that develop products, this *proof of concept* is typically required to get management approval and funding to continue developing the product.

Idea, Concept, and Functional Description

Let's say you have a cabinet with a door on a hinge and a drawer, and it needs a very small alarm system. Or maybe you want to design a special cabinet with built-in security. Figure 10-1 shows a sketch of how a potentiometer and electrical contact similar to a pushbutton could be used to detect when either the door or drawer is open. This sketch is similar to a concept diagram, which focuses only on conveying what the product or invention does.

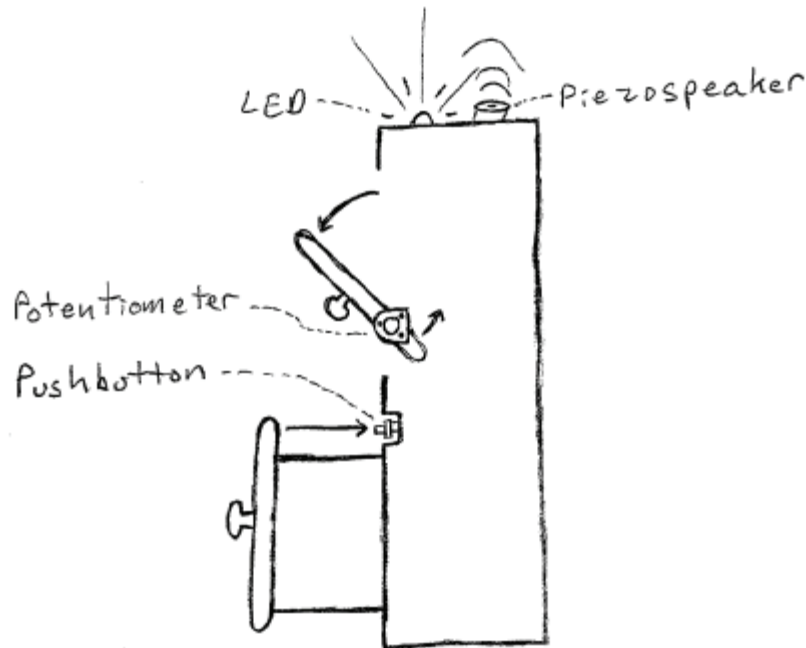


Figure 10-1
Concept
Sketch of a
Cabinet Micro
Security
System

10

The functional description is important. When you have a better idea of what your device is supposed to do at the beginning, it prevents problems that can happen if you have to redesign the device to accommodate something you didn't think about. Designers and companies that create custom devices for customers have to be very careful to cross-examine their customers to understand what they expect. Especially for custom-engineered devices, redesigns can be hugely expensive and time consuming.

Here is an example of a very brief functional description we can use for our simple system: Develop a circuit and program prototype for a micro alarm system that can monitor one small door that's on a hinge and a drawer. If armed, an alarm should sound if either door or drawer is opened. A status LED should glow green when the alarm is not armed, and red when it is armed. A prototype may be armed and disarmed by computer control. A time delay should be incorporated after the device has been armed to allow the user to close the cabinet.

Specification

Beyond the functional description, a *specification* typically accounts for as many aspects of the proposed device as possible, including: cost, power consumption, voltage supply, dimensions, weight, speaker volume, and many other details.

Initial Design

Often, the initial design involves brainstorming for approaches that “might” solve the design problem, and many of these ideas have to be tested to find out if they really are feasible. Other portions of the design might involve fairly standard or common parts and design practices. Our micro alarm fits in this category, at least for the prototype. A pushbutton could be mounted in the cabinet so that when the drawer is closed, it presses the pushbutton. For the door on a hinge, a potentiometer could be attached so that it twists with the door and can sense the door’s position. The bicolor LED is a familiar indicator, and the piezospeaker is certainly a well-known alarm noise maker.

So, now we know the circuits we need for our micro security cabinet prototype: bicolor LED, pushbutton, potentiometer, and piezospeaker. Here is a list of chapters and activities where each of these circuits was introduced:

- Bicolor LED: Chapter 2, Activity #5
- Pushbutton: Chapter 3, Activity #2
- Potentiometer: Chapter 5, Activity #3
- Piezospeaker: Chapter 8, Activity #1

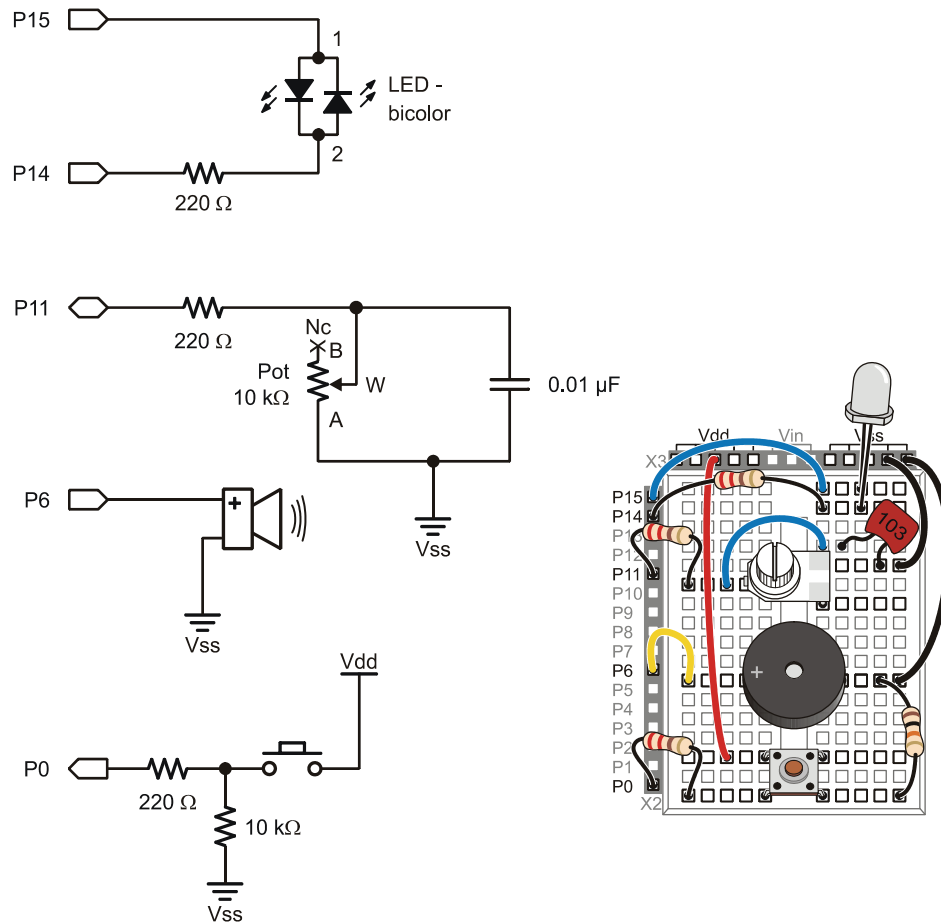
Cabinet Alarm Parts List:

Going back into each chapter and putting all the parts together results in this parts list:

- | | |
|--|-----------------------------------|
| (3) Resistors – 220 Ω (red-red-brown) | (1) Piezospeaker |
| (1) Resistors – 10 k Ω (brown-black-orange) | (1) Capacitor – 0.01 μ F |
| (1) LED – bicolor | (1) Potentiometer – 10 k Ω |
| (1) Pushbutton – normally open | (4) Jumper wires |

Cabinet Alarm Schematic

The schematic in Figure 10-2 is arranged to give all the components plenty of space on the breadboard, so not all the I/O pin connections are the same as they were in earlier chapters. You’ll have to keep this in mind when you harvest code examples from the earlier chapters to test each of the circuits.

Figure 10-2: Alarm System Prototype Schematic

10

ACTIVITY #2: BUILD AND TEST EACH CIRCUIT INDIVIDUALLY

Whenever possible, test each subsystem individually before trying to make them work together. If you follow this rule, your projects will go more smoothly, and it'll save a lot of troubleshooting time. For example, if all the circuits are built but not tested, people have a natural tendency to spend too much time examining code and forget to check each circuit. So, the most important time savings in this procedure is in making sure that there are no circuit mistakes trying to trick you into thinking they are coding errors.

Building and Testing Each Circuit

This activity demonstrates focusing on individual subsystems by building and testing each circuit. Once the pushbutton circuit is built and tested, we'll build and test the speaker circuit. After repeating this process with the potentiometer and bicolor LED, the circuits will all be "known good" and ready for some application code.

- ✓ Find test code in Chapter 3, Activity #2 that you can adapt to testing the Figure 10-2 pushbutton circuit.
- ✓ Change the I/O pin references so that it works with the circuit in Figure 10-2.
- ✓ Test the code and correct any bugs or wiring errors before continuing.
- ✓ Repeat this same process for:
 - Piezospeaker circuit from Chapter 8, Activity #1
 - Potentiometer circuit from Chapter 5, Activity #3
 - Bicolor LED circuit from Chapter 2, Activity #5
- ✓ Make sure to save each modified program under a new name, preferably in a separate folder, maybe named "WAM Chapter 10."

Your Turn – System Test

Now that all the circuits are tested and all the test programs saved on your PC, it's time to build up a system test that displays debug messages indicating which circuit is being tested as it executes the test code. This is a useful exercise because typical alarm systems have self-test and diagnostic modes that utilize all the features in one routine.

- ✓ Combine elements in your test programs into a single program that it:
 - Starts by displaying the color of the bicolor LED in the Debug Terminal as it updates the color...
 - Then displays a message that the piezospeaker is making sound while it beeps...
 - Finally enters a loop that repeatedly reports the pushbutton drawer sensor and potentiometer hinged door sensor status in the Debug Terminal.
- ✓ Test for and fix any bugs before continuing.

ACTIVITY #3: ORGANIZE CODING TASKS INTO SMALL PIECES

Just as each circuit should be built and tested before making them work together, each feature of the code should also be developed and tested individually before incorporating it into the larger application. `MicroAlarmProto(Dev-009).bs2` is an example of a program that's on its way to a proof of concept. Its Debug Terminal user interface is mostly in place, and the alarm system correctly cycles through its various modes or states, including not armed, arming, armed, and triggered.

At this point, the **Alarm_Arming** subroutine at the end of the program is still under construction. It has code in place that triggers the alarm if the pushbutton is released, which indicates that the drawer has been opened, but it does not yet monitor the hinged door. Potentiometer code needs to be added to the **Check_Sensors** subroutine that measures its position. If its position is beyond a certain threshold, 15 for example, the **state** variable should be changed to **Triggered**. Two additional tasks that remain are to turn the bicolor LED green when the alarm is not armed, and red when it is armed. These remaining tasks are indicated by comments in the code that look like this:

```
' To-do: bicolor LED green
...
' To-do: bicolor LED red
...
' To-do: Check if Potentiometer is over threshold
value. If yes, then, trigger alarm
```

10

- ✓ Hand-enter `MicroAlarmProto(Dev-009).bs2` into the BASIC Stamp Editor (recommended), or download it from www.parallax.com/go/WAM and open it with the BASIC Stamp Editor.
- ✓ Examine the program and note how each subroutine is modular, and does a specific job. This is part of organizing coding tasks into small pieces.
- ✓ If you do not remember how to use the Debug Terminal's Transmit and Receive windowpanes, review Figure 9-9 on page 299.
- ✓ Load `MicroAlarmProto(Dev-009).bs2` into the BASIC Stamp and use the Debug Terminal's Transmit windowpane to type the character A to arm the alarm, and D to disarm the alarm. The system does a brief countdown before arming the alarm. Make sure to press and hold the pushbutton before the alarm arms.
- ✓ While the alarm is armed, release the button. You will have a chance to disarm the alarm after a few seconds of alarm tone.
- ✓ Arm the alarm again. This time, type "D" to disarm the alarm before releasing the button.

[illegible]

```

' -----[ Subroutine - Prompt_to_Disarm ]-----
Prompt_to_Disarm:
  DEBUG CLS, "Type D to disarm", CR, ">"      ' Display message
  GOSUB Get_User_Input                        ' Call Get_User_Input
  RETURN                                      ' Return from Prompt_to_Disarm

' -----[ Subroutine - Alarm_Arming ]-----
Alarm_Arming:
  DEBUG CLS, "Close the cabinet.",            ' Warn user to secure cabinet
  CR, "You have"
  FOR seconds = 8 TO 0                        ' Count down seconds left
    DEBUG CRSRX, 9, DEC seconds, CLREOL,      ' Display time remaining
    " seconds left..."
    PAUSE 1000                                ' Wait 1 second
  NEXT                                        ' Repeat count down
  state = Armed                              ' Set state variable to Armed
  RETURN                                      ' Return from Alarm_Arming

' -----[ Subroutine - Alarm_Armed ]-----
Alarm_Armed:
  DO                                          ' Armed loop
    GOSUB Prompt_To_disarm                  ' Check for user input
    GOSUB Check_Sensors                    ' Check sensors
  LOOP UNTIL state <> Armed                  ' Repeat until state not armed
  RETURN                                    ' Return from Alarm_Armed

' -----[ Subroutine - Alarm_Triggered ]-----
Alarm_Triggered:
  DO                                          ' Alarm triggered loop
    DEBUG CLS, "Alarm triggered!!!"          ' Display warning
    FOR counter = 1 TO 15                    ' Sound 15 alarm tones
      FREQOUT 6, 100, 4500
      PAUSE 100
    NEXT
    FOR seconds = 1 TO 6                      ' 3 sec. for user to disarm
      IF state <> triggered THEN EXIT
      GOSUB Prompt_to_Disarm
    NEXT
  LOOP UNTIL state <> triggered              ' Repeat until disarmed

' -----[ Subroutine - Get_User_Input ]-----
Get_User_Input:
  char = 0                                  ' Clear char variable
  SERIN 16, 84, 500, Timeout_Label, [char]  ' Wait 0.5 sec. for key press
  GOSUB Process_Char                        ' If key, call Process_Char
  Timeout_Label:                            ' If no key, skip call
  RETURN                                    ' Return from Get_User_Input

' -----[ Subroutine - Process_Char ]-----
Process_Char:
  SELECT char                                ' Evaluate char case by case
  CASE "A", "a"                             ' If "A" or "a"

```

```

    state = Arming
CASE "D", "d"
    state = NotArmed
CASE ELSE
    DEBUG "Wrong character, try again"
    PAUSE 2000
ENDSELECT
RETURN
' -----[ Subroutine - Check_Sensors ]-----
Check_Sensors:
' To-do: Check if Potentiometer is over threshold value.
' If yes, then, trigger alarm
IF IN0 = 0 THEN state = Triggered
RETURN

```

New Coding Techniques in the Example Code

Take a look at the second **FOR...NEXT** loop in the **Alarm_Triggered** subroutine:

```

FOR seconds = 1 TO 6
    IF state <> triggered THEN EXIT
    GOSUB Prompt_to_Disarm
NEXT

```

If a call to the **Prompt_to_Disarm** subroutine results in a change in the **state** variable, the **IF...THEN** statement uses **EXIT** to get out of the **FOR...NEXT** loop before the 6 repetitions are done.

Another new command called **SERIN** appears in the **Get_User_Input** subroutine. **DEBUG** and **DEBUGIN** are special versions of the more general **SEROUT** and **SERIN** commands. To see how this works, try replacing the command **DEBUG "Program running..."** with **SEROUT 16, 84, ["Program running..."]**. Unlike the **DEBUG** and **DEBUGIN** commands, **SEROUT** and **SERIN** can communicate on any I/O pin, or pin 16 for communication with the **DEBUG** terminal. They also have special codes you can use to select the baud rate that are described in the **SERIN** and **SEROUT** command's Baud Rate tables in the BASIC Stamp Manual.

```

Get_User_Input:
    char = 0
    SERIN 16, 84, 500, Timeout_Label, [char]
    GOSUB Process_Char
    Timeout_Label:
    RETURN

```

The **Get_User_Input** subroutine starts by setting the **char** variable to 0 to clear any old values **char** might be storing. Then, it executes the **SERIN** command, with its optional **Timeout** value set to 500 ms (half a second), and its optional **timeout** label set to **Timeout_Label**, which is two lines below. If the **SERIN** command does receive a character within 500 ms, it stores the result in the **char** variable and moves on to the next line, which calls the **Process_Char** subroutine. If it doesn't get a character in 500 ms, it instead jumps to **Timeout_Label**, which causes it to skip over the subroutine call.

Your Turn – Next Steps Toward the Proof of Concept

It's time to get this program functioning as a proof of concept.

- ✓ Save a copy of `MicroAlarmProto(Dev-009).bs2` as `MicroAlarmProto(Dev-010).bs2`
- ✓ Use segments of your tested code from Activity #2 to complete the three “To-do” items.
- ✓ Test your modified code, and when you get it working right, save a copy of the code as `MicroAlarmProto(Dev-011).bs2`

ACTIVITY #4: DOCUMENT YOUR CODE!

10

`MicroAlarmProto(Dev-011).bs2` is not quite finished because it still needs some documentation and other changes that make the program easier to modify and maintain. For example, in the **Alarm_Triggered** subroutine, the command **FREQOUT 6, 100, 4500** has what some coders call “mystery numbers.” *Mystery numbers* are values that are used in a way the casual observer might not be able to easily discern. You could rewrite this command as **FREQOUT SpeakerPin, BeepTime, AlarmTone**. Then, you can add a Pin Directives section above the Constants section, and declare **SpeakerPin PIN 6**. Also, in the Constants section, declare **BeepTime CON 100**, and **AlarmTone CON 4500**.

Not every constant in a given program has to be named. Keep in mind that mystery numbers are values that are used in a way the casual observer might not be able to easily discern. Another example from the **Alarm_Triggered** subroutine is:

```
FOR seconds = 1 TO 6 ' 3 sec. for user to disarm.
```

The numbers 1 and 6 are not mystery numbers because it's clear that they make the **FOR...NEXT** loop repeat six times, and the comment to its right indicates that six repetitions lasts for three seconds. Not all supervisors may agree with this interpretation,

and some might heatedly proclaim that the 1 and the 6 really are mystery numbers. If you end up coding at work and your boss is a stickler for naming all constants, it's probably a good idea to just adhere to whatever coding style is required.

- ✓ Go through `MicroAlarmProto(Dev-011).bs2` and document mystery numbers by declaring pin directives and constants, and substituting their names for numbers in the program.
- ✓ One exception to **PIN** directives is the **SERIN** command's *Pin* argument, which should be declared as a constant and not a pin. *Pin* arguments are for I/O pins and range from P0 to P15. The *Pin* argument **16** causes the **SERIN** command to listen to the BASIC Stamp module's SIN pin, which is connected to your board's programming port.

Another area where `MicroAlarmProto(Dev-011).bs2`'s documentation is still weak is in the comments that explain each routine and subroutine. Each subroutine should have comments that explain what it does, any variables it depends on to do its job, and any variables that the subroutine uses to store results before its **RETURN**. Here is an example of good documentation added to the beginning of the **Process_Char** subroutine.

```
' -----[ Subroutine - Process_Char ]-----
'
' Updates the state variable based on the contents of the
' char variable. If char contains "A" or "a", the Armed
' constant gets stored in state. If char contains "D" or "d",
' the NotArmed constant gets stored in state.
'
Process_Char:
    '... code omitted here
    RETURN                                ' Return from...
```

- ✓ Update descriptions between the subroutine titles and their labels, and repeat for the main routine as well.
- ✓ When you are done, save a copy of your code with the name `MicroAlarmProofOfConcept(v1.0).bs2`.

Save Copies and Increment Version Numbers after Each Small Change

Make sure to continue saving copies of your code with each small adjustment. This makes it easy to take small steps backward to working code if your change(s) cause bugs. For example, before your next modification, save the file as

MicroAlarmProofOfConcept(v1.01).bs2, or maybe even v1.01a. When your next feature is fully implemented, chose a reasonable revision step. If it's a smaller revision, try v1.1; if it's a big revision, up it to v2.0.

ACTIVITY #5: GIVE YOUR APP AMAZING NEW FUNCTIONALITY

As mentioned earlier, each circuit you have worked with in this text is really an example from a group of components and modules that the BASIC Stamp can interact with in the same way. Figure 10-3 shows some part substitutions you could make to convert your current mini-enclosure security system into one that will protect an object sitting out in the open. This modified system can instead detect motions in the room, and also detect if someone lifts up the object you want to protect:

- Pushbutton: high-low output → replace with PIR Motion Sensor
- Potentiometer: variable resistor → replace with FlexiForce Sensor

The PIR sensor detects changing patterns of passive infrared light in the surrounding area, and sends a high signal to indicate that motion is detected, or a low signal to indicate no motion. The FlexiForce sensor's resistance varies with force applied to the round dot on the end (such as an object sitting on it), so it can be measured in an RC circuit with the **RCTIME** command.

10

Figure 10-3: Sensors to Upgrade our Mini Alarm System



*PIR Motion
Sensor*



FlexiForce Sensor

- ✓ Go to www.parallax.com and type “motion detection” into the Search field, then click the Go button.
- ✓ Find the PIR Sensor in the search results and go to its product page.
- ✓ Download PIR Sensor Documentation (.pdf), and optionally watch the PIR Sensor video clip. The PDF will be in the page's Downloads section.

- ✓ Read the documentation's explanations, schematic, and PIR_Simple.bs2 example code. Could you substitute this sensor for a pushbutton?
- ✓ Go back to your search results (or back to the Parallax home page) and type pressure into the Search field. Then, follow the FlexiForce sensor link.
- ✓ Find and un-zip the FlexiForce Documentation and Source Code (.zip).
- ✓ In the un-zipped folder, open and read the documentation, schematic, and FlexiForce Simple.bs2 source code. Could you substitute this sensor for a potentiometer?



For a step-by-step example that demonstrates how the enhancements in both this and the next activity can be incorporated into your Micro Alarm application, follow the Stamps in Class "Mini Projects" link at: www.parallax.com/Education.

ACTIVITY #6 : HOW TO JUMP OVER DESIGN HURDLES

Now that you're just about done with *What's a Microcontroller?* one of the most important next steps you can take is finding answers for tasks you don't already know how to solve with your microcontroller. Here are the general steps:

- Step 1:** Look for components or circuits that could solve your problem.
- Step 2:** Read about the component/circuit, and find out how it works. Pay special attention to how the BASIC Stamp would need to interact with the component/circuit.
- Step 3:** Check to find out if example code is available for the circuit or component. That'll make it a lot easier to incorporate into your application.

Let's say that the next step in your project is to display the system's status without the computer connection. Here's an example of how you can find and evaluate a component for your application.

- ✓ (Step 1) Go to www.parallax.com and try the term "display" in the Search field. From the home page, you may need to click the Go button instead of just pressing Enter. Go to the product pages of the various result items in the search and see if you can find one that's relatively inexpensive and capable of displaying a couple lines of text.

If you decided the Parallax Serial 2x16 LCD in Figure 10-4 is a good candidate, you're on the right track. However, just about any of the displays are fair game.

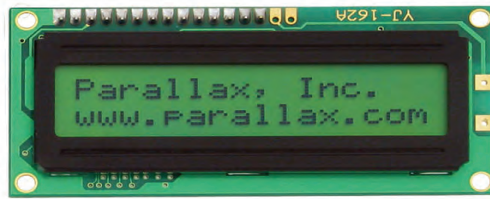


Figure 10-4
Parallax 2x16 Serial LCD

- ✓ (Step 2) Go to the Parallax Serial 2x16 LCD product page. If you haven't already done so, read the product description. Then, find the link to the Parallax Serial 2x16 LCD's PDF Documentation. It'll be in the page's Downloads & Resources section, probably labeled "Parallax Serial 2x16 LCD Documentation v2.0 (pdf)." The 2.0 version number might be higher by the time you try this.
- ✓ (Step 3) Check for example code in the Parallax Serial 2x16 LCD's PDF documentation as well as links to code in the product web page's Downloads & Resources section. Look for a nice, short, simple example program that displays a test message because it usually provides a good starting point.

After the brief introduction to **SERIN** and **SEROUT** that followed this chapter's example program, example code for the Parallax Serial LCD, which relies on **SEROUT**, might look rather familiar.

10



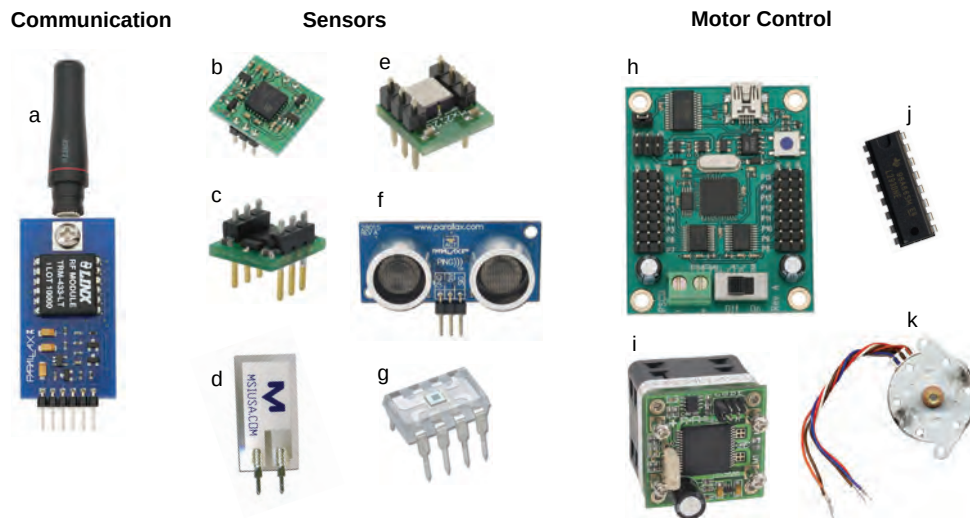
If you follow the **Smart Sensors and Applications** link, you can download the *Smart Sensors and Applications* textbook, which has an entire chapter about controlling this display with the BASIC Stamp 2.

Three Examples out of How Many?

The PIR and FlexiForce Sensors and the Parallax Serial LCD are three examples of modules and components you can use to greatly increase your prototype's functionality. These three are just a drop in the bucket compared to what's available.

Figure 10-5 shows a few more modules and components, and it still represents just a small sample. The examples in the figure are: (a) RF module for radio communication, (b) gyro for detecting rotation speed, (c) compass for finding direction, (d) vibration sensor, (e) accelerometer for detecting tilt and speed changes, (f) ultrasonic sensor for detecting distance, (g) light intensity sensor, (h) servo controller, (i) DC motor controller, (j) Darlington array for driving stepper motor coils, and (k) stepper motor. You can find any of these devices at www.parallax.com with a keyword search. For example, to find out more about (f), enter “ultrasonic sensor” into the Parallax home page’s Search field and then click the Go button.

Figure 10-5: More Module and Accessory Examples



Your Turn – Investigating More Resources

If you have a project in mind and need to find a circuit and code to support one of your project’s features, the search procedure just discussed provides a good starting point, but it only finds product pages on www.parallax.com, and there are a number of design questions that product pages won’t necessarily answer. Fortunately, there are lots more resources, including:

- Stamps in Class PDF textbooks
- Parallax PDF product documentation
- Nuts and Volts of BASIC Stamps columns
- Answers to questions and articles at forums.parallax.com
- BASIC Stamp articles published on the Internet

When you are looking for components and information about how to use them with the BASIC Stamp, it falls in the category of “application information.” When searching for application information, it’s best to start with the manufacturer’s web site, then expand the search to include forums, and if you still haven’t found a good solution, expand it further to include the World Wide Web at large. Figure 10-6 shows an example of a Google keywords search that will search for the terms “infrared” and “remote” in PDF documents and product pages at www.parallax.com. The important part here is that the Google searches PDF documents instead of just product pages. Make sure there are no spaces in `site:www.parallax.com`.



Figure 10-6
Google Search of the site
www.parallax.com

10

You can modify the search to include questions and answers on the Parallax support forums by changing the “www” to “forums” like this:

`infrared remote site:forums.parallax.com`

This searches for all questions, answers and short articles that contain the words “infrared” and “remote” at forums.parallax.com. To find an application specific to the BASIC Stamp, change your search to the terms below. Make sure the words BASIC Stamp are in quotes because it will filter out postage stamp collecting results.

Here is a summary of the Google search sequences for “BASIC Stamp” infrared remote

- ✓ `infrared remote site:www.parallax.com`
 - Searches for the terms “infrared” and “remote” in PDF and product pages at www.parallax.com

- ✓ infrared remote site:forums.parallax.com
 - Searches for the terms “infrared” and “remote” in discussions at forums.parallax.com
- ✓ “BASIC Stamp” infrared remote
 - Searches the web at large for the words “infrared” and “remote” in the same page or PDF with the phrase “BASIC Stamp.”

Let's say that the next step for your Micro Alarm project is a keypad, but the documentation and examples you found with a simple product page search at parallax.com turned out to be sparse and devoid of example circuits and code. Since some more searching would be in order, let's try a Google search of the Parallax site for all references to keypad. Remember, the Google search includes PDF documents.

- ✓ Go to www.google.com.
- ✓ Type “keypad site:www.parallax.com” into the Search field and then press Enter.

The results may take some patience and persistence to sift through, and there may be many pages of results. There's usually enough of an excerpt from each search result to get some context for each link. This will give some idea of which ones to skip and which ones to look at more closely. After a few pages, you might find and follow a link to an IR Remote Parts Kit, shown in Figure 10-7. This might not be a solution you were expecting, but after examining the price, documentation, and example code, it might have a lot of potential for your enhanced micro security system keypad.

Figure 10-7: IR Remote Parts Kit



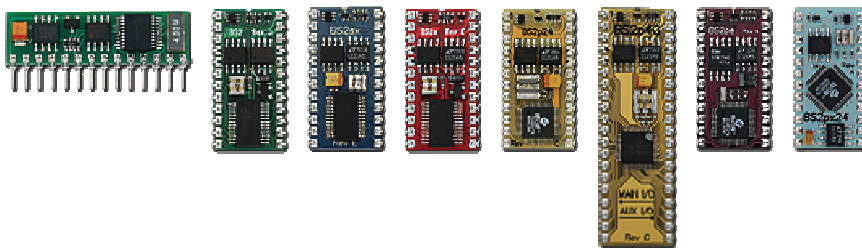
If after all that, you still haven't found the information you need, it's time to ask at forums.parallax.com. When you post a question there, it will be seen by experts in a variety of fields as well as by teachers, hobbyists, and students. The collective expertise of the Parallax Forums should be able to help get you past just about any design hurdle!

Processor Memory and Speed Design Hurdles

In some cases, programs for larger projects can grow long enough to exceed the BASIC Stamp 2's program memory. This design hurdle can sometimes be jumped by rewriting code that does more work with fewer commands. Another option is to upgrade to a BASIC Stamp model with a larger program memory. In other cases, the project might involve storing more variable values than the BASIC Stamp 2 can accommodate. There are also BASIC Stamp 2 models that feature scratchpad RAM for variable values. Other projects might need to do more tasks in less time than the BASIC Stamp 2 is designed to take, so some models of BASIC Stamp 2 are designed with faster processing speeds.

Figure 10-8 shows all of the different BASIC Stamp models. For details about one, follow the “Compare BASIC Stamp Modules” link at www.parallax.com/basicstamp.

Figure 10-8: The Complete Lineup of BASIC Stamp Models



From left: BS1, BS2, BS2e, BS2sx, BS2p24, BS2p40, BS2pe, BS2px

10

- BS1:** Affordable yet capable, perfect for small projects or tight spaces.
- BS2:** Ideal for beginners with a vast resource base of sample code; the core of the Stamps in Class program.
- BS2e:** Perfect for BS2 users who need more program and variable space.
- BS2sx:** Supports the BS2 command set with more variable and program space at more than twice the execution speed.
- BS2p24:** In addition to more speed and variable space, special commands support I/O polling, character LCDs and I²C and 1-wire protocols.
- BS2p40:** All the features of the BS2p24 with a bank of 16 additional I/O pins.
- BS2pe:** Supports the BS2p24 command set paired with lower power consumption and more memory for battery-powered datalogging applications.
- BS2px:** The fastest BASIC Stamp model supports all BS2p24 commands, plus special I/O configuration features.

One thing to keep in mind if you upgrade to a faster model of BASIC Stamp is differences in units for time-sensitive commands like **RCTIME** and **FREQOUT**. Since different models' processors run at different speeds, units for *Duration* and *Frequency* and other arguments might be different. For example, when the BS2 executes **FREQOUT 6, 100, 4500**, it sends a high pitched alarm tone to P6 for 100 ms ($1/10^{\text{th}}$ of a second) at a frequency of 4500 Hz. The same command executed by the BS2px sends a tone that only last 16.6 ms at a frequency of 27,135 Hz, which is so high-pitched that it's not even audible to human ears! For the complete descriptions of how each command works on each model, and for tips on converting BS2 programs to perform correctly on other models, see the BASIC Stamp Editor Help.

High-performance Parallel Processing

Some complex applications require processing agility and memory that's well beyond the BASIC Stamp 2 line's capabilities. These are the kind of projects that the Propeller microcontroller is designed for. This uniquely capable microcontroller has eight much higher speed processors in one chip, along with 32 I/O pins and ample program memory and RAM. The processors can all operate at the same time, both independently and co-operatively, sharing access to global memory and a system clock. Each processor also has its own memory, and additional hardware to perform complex tasks like high-speed I/O pin state monitoring or generating signals for a television or computer display.

The Propeller Education Kit shown in Figure 10-9 is a good way to get started with the Propeller microcontroller. This kit is not necessarily the best next step after *What's a Microcontroller?* The next activity has some good recommendations for next book/kit steps. However, when you notice that your projects are getting more ambitious and challenging, remember the Propeller microcontroller and Propeller Education Kit.



Figure 10-9
Propeller Education Kit
(left)
PE Platform (right)

ACTIVITY #7: WHAT'S NEXT?

Now that you are just about finished with *What's a Microcontroller?* it's time to think about what to learn next. Before continuing, take a moment to consider what you're most interested in. Some general categories you could delve further into include:

- Robotics
- Electronics
- Sensors
- Automation
- Hobby projects
- Earth sciences and climate measurement

This activity inventories resources you can use to move forward with each these categories.



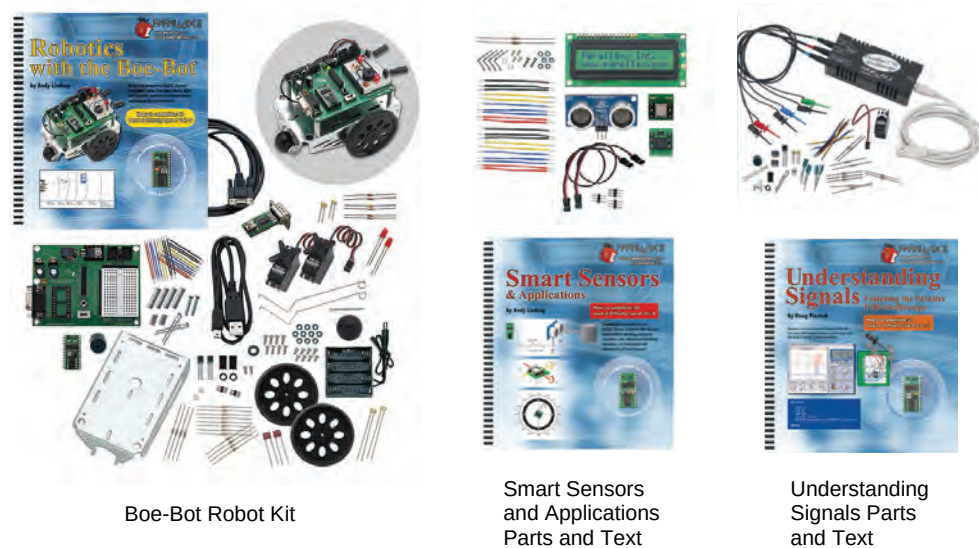
The resources, kits, and components discussed in this activity are current as of when this chapter was written (Fall 2009). Newer and better versions of resources, kits, and components may become available that replace the ones presented here. Make sure to check www.parallax.com for the latest information.

10

What's a Microcontroller Sequels

Figure 10-10 shows the books and kits that make the best sequels to this book. *Robotics with the Boe-Bot* is a lot of fun and a great learning experience because you get to apply many of the techniques from this book to robotics applications with the rolling Boe-Bot[®] robot. *Smart Sensors and Applications* was written to be “What's a Microcontroller, Part 2.” It was renamed because all the nifty sensors and the liquid crystal display shown in the center of Figure 10-10 have coprocessors that communicate with the BASIC Stamp. The coprocessors make them “smart” sensors. *Understanding Signals* is great because it allows you to “see” interactions between the BASIC Stamp and circuits with a Parallax oscilloscope that you plug into your computer's USB port.

Figure 10-10: Great Next Steps after *What's a Microcontroller?*



More Stamps in Class Kits and Textbooks

Figure 10-11 shows a flowchart that outlines all the Stamps in Class kits and textbooks available at the time of this writing. It's accessible through the Stamps in Class Program Overviews and Flowchart link at www.parallax.com/Education, and you can click each picture to visit the product page for the book and its accompanying kit. *What's a Microcontroller?* is at the top-left of the figure. From there, the flowchart indicates that you can either jump to *Robotics with the Boe-Bot* or any text/kit in the Sensors or Signals series.



Full PDF Textbook Downloads: You can download the entire full-color PDF of each Stamps in Class book at www.parallax.com. Click on any of the chart's pictures to navigate to the Text + Kit page, and you will find the PDF link in the page's Downloads section.

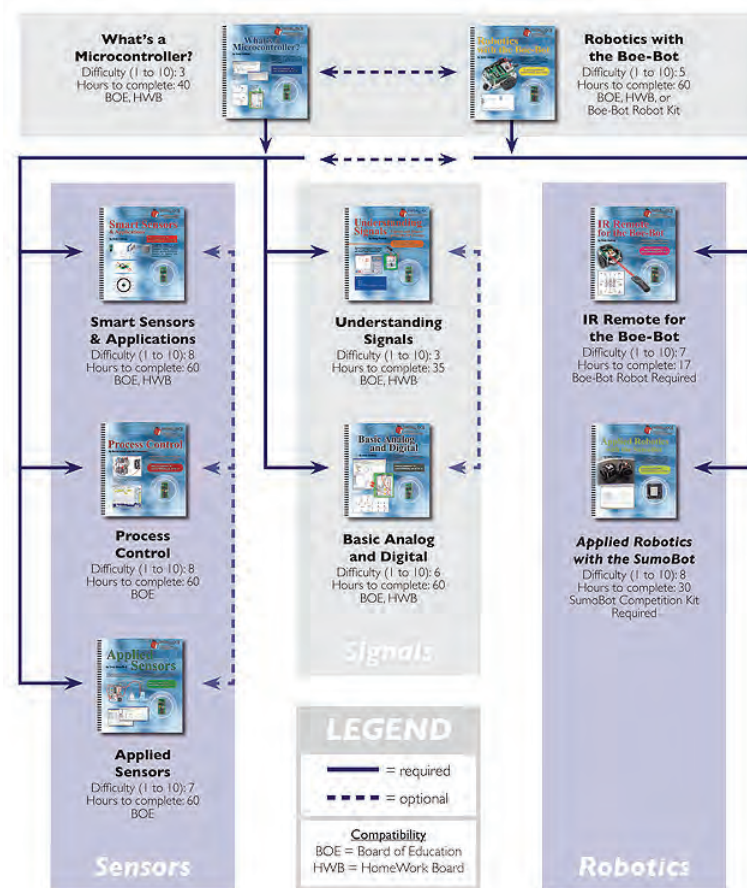


Figure 10-11
Stamps in Class
Flowchart

10

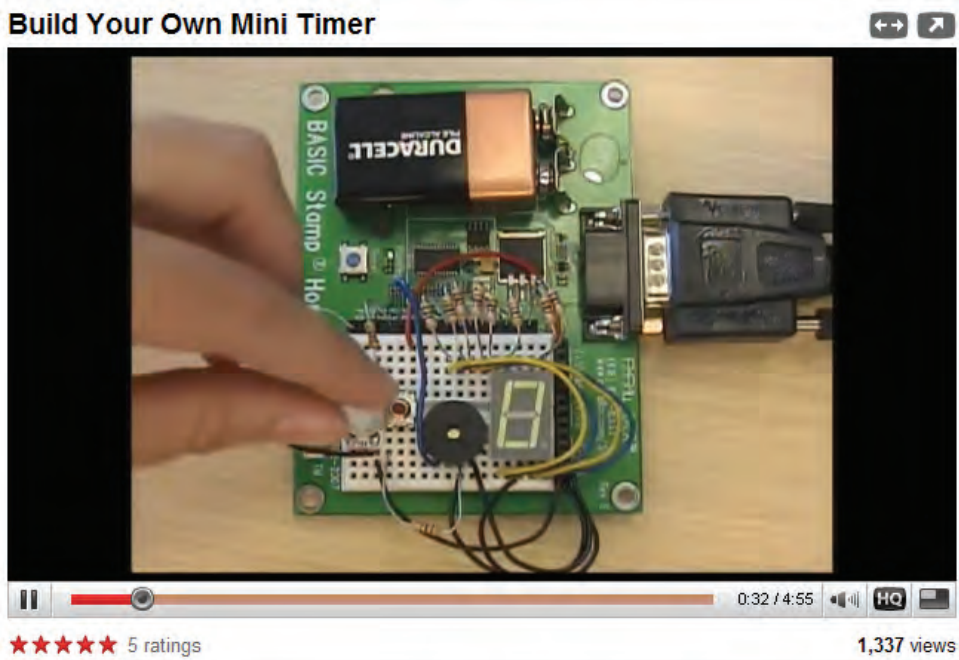
If the category you are interested in is:

- Robotics, then the next step is definitely *Robotics with the Boe-Bot*.
- Sensors, inventing, or hobby projects, then your next step would be *Smart Sensors and Applications*.
- Electronics (signals), then your next step would be *Understanding Signals*.
- Automation, then your next step would be *Process Control*.
- Earth science and climate measurement, then your next step would be *Applied Sensors* (originally named *Earth Measurements*).

Additional Stamps In Class Resources

Above and beyond what's in the Stamps in Class Textbooks, there are Stamps in Class “Mini Projects” linked at www.parallax.com/Education. Some projects utilize just the stock parts from a given kit but demonstrate new ways to use them along with new concepts. Many of these projects are like complete Stamps in Class textbook chapters with activities, schematics, wiring diagrams, and complete code listings that can be downloaded. Some even have accompanying video tutorials. Figure 10-12 is taken from the video for the “Build Your Own Mini Timer” project, which can be done with just the parts you have been using in this book. Whether you are looking for more information or creative inspiration, you might find it here.

Figure 10-12: Example Stamps in Class “Mini Project”



SUMMARY

This book introduced a variety of circuits and techniques, all of which are building blocks in common products as well as in inventions. This book also introduced techniques for orchestrating the various building blocks with the BASIC Stamp microcontroller. This chapter demonstrated how to incorporate these techniques and building blocks into a prototype, and it also recommended some next steps for learning more in your area of interest.

The approach for making the BASIC Stamp interact with a given circuit can be applied to a variety of other circuits and modules to accomplish an even wider range of tasks. Two examples applied to the micro alarm prototype were: (1) a motion sensor with an interface similar to the pushbutton and (2) a pressure sensor with an interface similar to the potentiometer.

While developing code for your application, make sure to save your work frequently under incremented revision names. Also, make sure to use meaningful names for I/O pins and numbers with **PIN** and **CON** directives. Finally, add lots of comments to your code explaining what it does and how it does it. Subroutines should include comments that explain what the subroutine does along with any variables with values it uses to do its job as well as variables that results are stored in when the subroutine is done.

This chapter also introduced a variety of research techniques for implementing features in your prototype. Even if you start with no clue about how to make a particular feature work, you can use search terms to find useful component, circuit, and code examples. Stamps in Class textbooks and kits also feature a wealth of circuits and useful design techniques, and they are a great place to learn more in the fields of robotics, sensors, electronics, automation, earth science, and more. All the textbooks that come with Stamps in Class kits are free downloads.

Now that you have reached the end of this book, take a moment now to think about four things: (1) the techniques you have learned, (2) your next invention, project or prototype, (3) how what you have learned here can be applied to it, and (4) what you want to learn next.

- ✓ Now, it's time to get started on your next project or prototype.
- ✓ Make sure to keep studying and learning new techniques as you go.
- ✓ Have fun, and good luck!

Appendix A: Parts List and Kit Options

What's a Microcontroller Parts & Text Kit #28152, Parts Only #28122 Parts and quantities subject to change without notice		
Parallax Part #	Description	Quantity
150-01020	Resistor, 5%, 1/4W, 1 k Ω	10
150-01030	Resistor, 5%, 1/4W, 10 k Ω	4
150-01040	Resistor, 5%, 1/4W, 100 k Ω	2
150-02020	Resistor, 5%, 1/4W, 2 k Ω	2
150-02210	Resistor, 5%, 1/4W, 220 Ω	6
150-04710	Resistor, 5%, 1/4W, 470 Ω	6
152-01031	Potentiometer - 10 k Ω	1
200-01031	Capacitor, 0.01 μ F	2
200-01040	Capacitor, 0.1 μ F	2
201-01080	Capacitor, 1000 μ F	1
201-03080	Capacitor 3300 μ F	1
28123	What's a Microcontroller? text (in #28152 only)	1
350-00001	LED - Green - T1 3/4	2
350-00005	LED - Bicolor - T1 3/4	1
350-00006	LED - Red - T1 3/4	2
350-00007	LED - Yellow - T1 3/4	2
350-00027	7-segment LED Display	1
350-00029	Phototransistor, 850 nm, T1 3/4	1
400-00002	Pushbutton – Normally Open	2
451-00303	3 Pin Header – Male/Male	1
500-00001	Transistor – 2N3904	1
604-00010	10 k Ω digital potentiometer	1
800-00016	3" Jumper Wires – Bag of 10	2
900-00001	Piezo Speaker	1
900-00005	Parallax Standard Servo	1

COMPLETE KIT OPTIONS

There are several kit options available that include a BASIC Stamp 2 microcontroller development board and all of the electronic components to complete the activities in this text:

- BASIC Stamp Activity Kit (#90005) includes:
 - o BASIC Stamp HomeWork Board with surface-mount BS2
 - o USB to Serial Adapter with USB A to Mini-B Cable (#28031)
 - o What's a Microcontroller? Parts and Text (#28152)
- BASIC Stamp Discovery Kit (Serial #27207 or USB #27807) includes:
 - o Board of Education (Serial #28150 or USB #28850)
 - o BASIC Stamp 2 microcontroller module (#BS2-IC)
 - o Programming Cable (Serial #800-00003 or USB A to Mini-B #805-00006)
 - o What's a Microcontroller? Parts and Text (#28152)
 - o BASIC Stamp Manual (#27218)
- What's a Microcontroller Parts & Text Kit (#28152). **PLUS**
- Board of Education Full Kit (Serial #28103 or USB #28803) includes:
 - o Board of Education (Serial #28150 or USB #28850)
 - o BASIC Stamp 2 microcontroller module (#BS2-IC)
 - o Programming cable (Serial #800-00003 or USB A to Mini-B #805-00006)
 - o Jumper Wires (1 pack of 10)

A note to Educators: Quantity discounts are available for all of the kits listed above; see each kit's product page at www.parallax.com for details. In addition, the BASIC Stamp HomeWork Board is available separately in packs of 10 as an economical solution for classroom use, costing significantly less than the Board of Education + BASIC Stamp 2 module (#28158). Please contact the Parallax Sales Team toll free at (888) 512-1024 for higher quantity pricing.

Appendix B: More about Electricity

B



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 of 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 electrical 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* (short for *ampere*) 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. Milliamp (mA) and microamp (μA) measurements are more convenient for discussing the BASIC Stamp module's supply current as well as currents between I/O pins and circuits. 1 mA = 1/1,000 A, and 1 μA = 1/1,000,000 A.

What's resistance? *Resistance* is the tendency of an element in a circuit to resist 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 (Ω).

What's a conductor? Copper wire has almost no resistance, and it's called a *conductor*.

BONUS ACTIVITY: OHM'S LAW, VOLTAGE, AND CURRENT

This activity applies some of the definitions just discussed.

Ohm's Law Parts

- (1) Resistor – 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) LED – any color

Test Circuit

The resistance value of R_i in Figure B-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 from your Board of Education or HomeWork Board whenever you modify the circuit.
- ✓ Build the circuit shown in Figure B-1 starting with a 220 Ω resistor.
- ✓ Modify the circuit by replacing the 220 Ω resistor with a 470 Ω resistor. Was the LED less bright?
- ✓ Repeat using the 1 k Ω resistor, then the 2 k Ω resistor, checking the change in brightness each time.

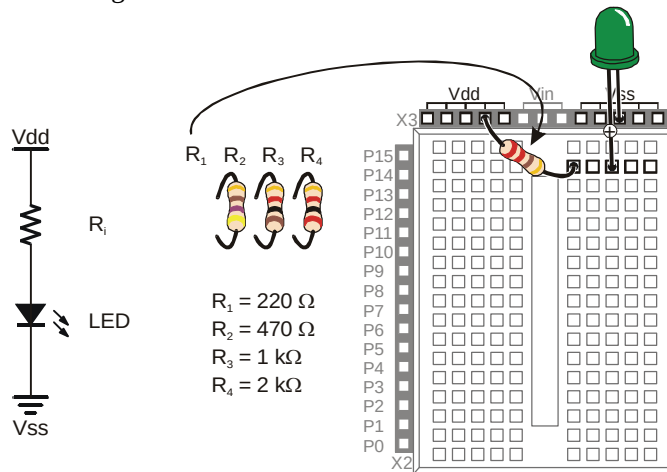


Figure B-1
LED Current Monitor

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. On our system, V_{dd} is 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 B-1, but use a 1 k Ω 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 Ω 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 I/O pins 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.



Current Rules for BASIC Stamp I/O Pins

- An I/O pin can "source" up to 20 mA. In other words, if you send a **HIGH** signal to an I/O pin, it should not supply the LED circuit with more than 20 mA.
- If you rewire the LED circuit so that the BASIC Stamp makes the LED turn on when you send the **LOW** command, an I/O pin can "sink" up to 25 mA.
- P0 through P7 can only source up to 40 mA. Likewise with P8 through P15. 40 mA is also the I/O supply current budget for the BASIC Stamp 2 module's 5 V regulator, so the total current draw for all I/O pins should never exceed 40 mA. If you have lots of LED circuits, you will need larger resistors so that the circuits don't draw too much current.
- For more information, consult the BASIC Stamp 2 Pin Descriptions table in the BASIC Stamp Manual.

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 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. There are also rules for how current and voltage add up in circuits. These are called *Kirchhoff'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 supplies exactly 5 V. (Batteries are not typically 5 V, though four 1.2 V rechargeable nickel-cadmium batteries in series can add up to 4.8 V.) The Board of Education and BASIC Stamp HomeWork Board both have 5 V regulators that convert the 6 to 9 V battery supply voltage to regulated 5 V for the Vdd sockets above the breadboard. The BASIC Stamp also has a built-in voltage regulator that converts the 6 to 9 V input to 5 V for its components.

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: The voltage across a diode's anode and cathode as current passes through it from anode to cathode. For the green LED in the Figure 2-6 circuit on page 33, you can assume that the forward voltage across the LED is approximately 2.1 V for the sake of making circuit calculations. If the LED is yellow, assume 2.0 V, and if it's red, assume 1.7 V. These voltages will vary slightly with the amount of current passing through the circuit. Smaller series resistance and/or higher voltage applied to the circuit results in higher current flow. Larger series resistance and/or smaller applied voltage results in lower current flow.

Kirchhoff'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.

Kirchhoff'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 a red 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 B-2 shows how to calculate the voltage across the resistor. The voltage supplied is on the left; it's 5 V. The voltages used by each component are to the right of the circuit. 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 going to be about 1.7 V (the red light emitting diode's forward voltage). We also know that the voltage across the parts has to add up to 5 V because of Kirchhoff's Voltage Law. The difference between 5 V and 1.7 V is 3.3 V, so that must be the voltage across the resistor V_R .

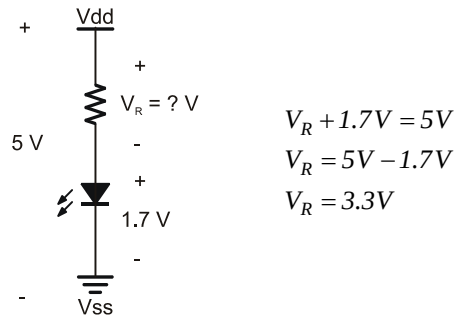


Figure B-2
Voltage Across
the Circuit,
Resistor, and
LED

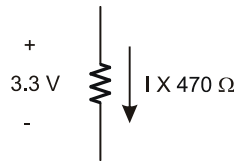


Kilo is metric for 1000. The metric way of saying 1000 is kilo, and it's abbreviated with the lower-case k. Instead of writing 1000 Ω , you can write 1 k Ω . Either way, it's pronounced one-kilo-ohm. Likewise, 2000 Ω is written 2 k Ω .

Milli is metric for 1/1000, and it is abbreviated with a lower-case m. If the BASIC Stamp supplies an LED circuit with 3.3 thousandths of an amp, that's 3.3 milliamps, or 3.3 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.

Now that we have calculated the voltage across the resistor, Figure B-3 shows an example of how to use that value to calculate the current passing through the resistor. Start with Ohm's Law: $V = I \times R$. You know the answers to V (3.3 V) and R (470 Ω). Now, all you have to do is solve for I (the current).



$$V = I \times R$$

$$3.3V = I \times 470\ \Omega$$

$$I = \frac{3.3V}{470\ \Omega}$$

$$I \approx 0.00702\ V/\Omega$$

$$I = 0.00702\ A$$

$$I = \frac{7.02}{1000}\ A$$

$$I = 7.02\ mA$$

Figure B-3

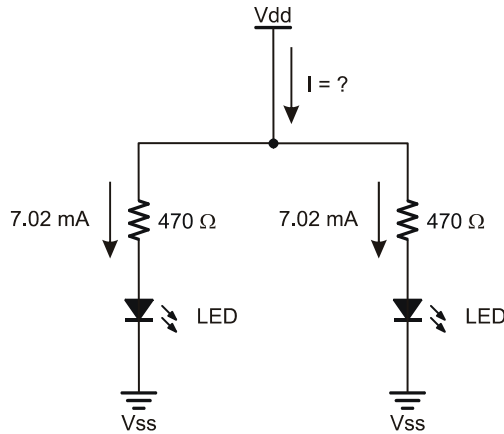
Calculating
Current through
the Resistor



Yes, it's true ! 1 A = 1 V/Ω (One amp is one volt per ohm).

How much current is 7.02 mA? It's the amount of current the LED circuit in Figure B-2 conducts. You can replace the 470 Ω resistor with a 220 Ω resistor, and the circuit will conduct about 15.0 mA, and the LED will glow more brightly. If you use a 1000 Ω resistor, the circuit will conduct 3.3 mA, and the LED will glow less brightly. A 2000 Ω resistor will cause the LED to glow less brightly still, and the current will be 1.65 mA.

Let's say you want to make an I/O pin turn two LEDs on at the same time. That means that inside the BASIC Stamp, it would supply the circuits as shown in Figure B-4. Would the circuit's current draw exceed the I/O pin's 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 B-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 14.04 mA, which you can round to 14.0 mA. Since this current draw is still below the I/O pin's 20 mA limit, it can safely be connected to an I/O pin and switched on/off with the BASIC Stamp.



$$I = I_1 + I_2 + \dots I_i$$

$$I = 7.02\text{ mA} + 7.02\text{ mA}$$

$$I = 14.04\text{ mA} \approx 14.0\text{ mA}$$

Figure B-4
Total
Current
Supplied to
Two LED
Circuits

Your Turn – Modifying the Circuit

- ✓ Repeat the exercise in Figure B-2, but use $V_{in} - V_{ss} = 9\text{ V}$ instead of $V_{dd} - V_{ss} = 5\text{ V}$.

Assuming the forward voltage does not change, the answer is $V_R = 7.3\text{ V}$. The measured resistor voltage will probably be slightly less because of a larger LED forward voltage from more current passing through the circuit.

- ✓ Repeat the exercise in Figure B-3, but use a $1\text{ k}\Omega$ resistor.

Answer: $I = 3.3\text{ mA}$.

- ✓ Use $V_R = 7.3\text{ V}$ to do the exercise in Figure B-3 with a $1\text{ k}\Omega$ resistor.

Answer: $I = 7.3\text{ mA}$.

- ✓ Repeat the exercise shown in Figure B-4 with one of the resistors at $470\ \Omega$ and the other at $1\text{ k}\Omega$.

Answer: $I = 7.02\text{ mA} + 3.3\text{ mA} = 10.32\text{ mA}$.

Appendix C: 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. With 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 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 ½ 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

- \$ -
- \$ (Hexadecimal formatter), 207
- % -
- % (Binary formatter), 181
- * -
- ** (Multiply High operator), 270
- */ (Multiply Middle operator, 85, 270
- ? -
- ? (symbol = x formatter), 45
- μ -
- μF (microfarad), 143
- 1 -
- 16-bit rollover bug, 122
- 7 -
- 7-segment display, 169, 170, 169–71
- A -
- Action sounds, 248
- Active-high vs. active low, 69
- AD5220 digital potentiometer, 292
- Algorithm, 87
- Alphabet Song, 257
- Amp, 335
- AND, 78
- Anode, 30
 - 7-segment display, 170
 - LED, 30
- Apostrophe, 42
- Applied Sensors, 329
- Arguments, 39
- ASCII, 276
- Automation, 329
- B -
- Base
 - Phototransistor, 198
 - Transistor, 289
- Base-10 numbers, 183
- Base-16 numbers, 183
- Base-2 numbers, 67
- BASIC Stamp, 11, 325
- BASIC Stamp Editor, 15
- BASIC Stamp model comparison, 325
- Battery, 35
- Beat, 252
- Benjamin Franklin, 35
- Bicolor LED, 50
- Binary, 61
- Binary numbers, 67, 179, 181
 - % (Binary formatter), 181
- Bit, 45, 179
 - Variable size, 45
- Boolean, 61
- Breadboard, 31, 32, 259
- BS1, 325
- BS2, 325
- BS2e, 325
- BS2p24, 325
- BS2p40, 325
- BS2pe, 325

- BS2px, 230, 325
- BS2sx, 325
- Bug, 16-bit rollover, 122
- Build Your Own Mini Timer project
 - video, 330
- Bus, parallel, 177
- Byte, 45, 179
 - Variable size, 45
- C -
- Cabinet alarm project, 310
- Cadmium sulfide, 197
- Capacitor, 143
 - Ceramic Capacitor Schematic Symbol and Parts Drawing, 150
 - Electrolytic, 143
 - Electrolytic Capacitor Schematic Symbol and Part Drawing, 144
 - Junction capacitance, 236
 - Polar – identifying terminals, 144
 - Used in parallel, 224
- Cathode, 30
 - Common cathode in 8-segment display, 170
 - LED, 30
- Charge, 335
- Closed circuit, 62
- CLREOL, 167
- CMOS, 61
- Code block, 78
- Code overhead, 84
- Collector
 - Phototransistor, 198
 - Transistor, 289

- Color spectrum, 197
- COM port, 41
- Commenting code, 42
- Common cathode, 170
- Communication products, 322
- Concept diagram, 308
- Conductor, 335
- Constants, 160
- Control characters. See DEBUG
 - Control Characters
- Controlling, 61
- Counting, 80
- CR, 25
- CRSRUP, 129
- Current, 28, 35, 335
 - Milliamp, 339
- Cycle, 117, 245
- D -
- DATA, 255
- Datalogging, 203
- DCD, 269
- DEBUG, 39
- DEBUG Control Characters, 129
 - CLREOL, 167
 - CR, 25
 - CRSRUP, 129
 - HOME, 76
- DEBUG Formatters, 129
 - \$ (Hexadecimal Formatter), 207
 - % (Binary formatter), 181
 - ? (symbol = x formatter), 45
 - DEC (Decimal formatter), 120, 207
- Debug Terminal

Transmit and Receive Windowpanes, 120

DEBUGIN, 119

DEC, 120, 207

Decimal formatter DEC, 207

Decimal numbers, 183, 204

Degree, 102

Device, parallel, 177

Diode, 30

Diode Forward Voltage, 338

DIRH, 178

DO...LOOP, 39, 83, 123

Dot, in music, 264

DTMFOUT, 251

Dual Tone Multi Frequency, 251

- E -

Earth Measurements, 329

Earth science, 329

Echo, 121

EEPROM, 203

Electrolytic capacitor, 143

Electron, 34, 35, 335

Embedded system, 11

Emitter

Phototransistor, 198

Transistor, 289

END, 63

EXIT, 282, 316

- F -

Farads, 164

Fetch and execute, 287

Flat notes, 254

FlexiForce Sensor, 319

FOR...NEXT, 43, 124

Formatters, DEBUG. *See* DEBUG

Formatters

Fractions, 85

FREQOUT, 247, 251

Frequency, 245

Functional description, 309

- G -

Google, 323

GOSUB, 216

GOTO, 217

Graphing software, 213

Ground, 31, 338

- H -

hertz, 245, 247

Hexadecimal formatter \$, 207

Hexadecimal numbers, 183

Hexadecimal to decimal conversion,
205

HIGH, 39, 182

HOME, 76

HomeWork board

and the RCTIME circuit voltage divider,
155

Hysteresis, 228

- I -

I/O pin protection, 69

I/O pins

Default direction, 181

DIRH and OUTH registers, 178

I/O Pins. *See* Input/Output Pins

IF...ELSEIF...ELSE, 75

IF...THEN, 78

IF...THEN...ELSE, 71

IN, 67

Input/output pins. *See* I/O pins

Integer, 270

Interference, 252
Interpreter chip, 287
IR Remote Parts Kit, 324

- J -

Junction capacitance, 236

- K -

KCL, 338
kHz, 245
Kilo, 339
Kirchhoff's Laws (Simplified)

Current, 338

Voltage, 338

Kirchhoff's Voltage and Current Laws,
338

KVL, 338

- L -

Label, 217
LCD Display, 320
LED, 27

as a light sensor, 235

Bi-color, 50

Part Drawing and Schematic Symbol, 30

Light Emitting Diode. *See* LED

Light emitting diodes, 27

light meter, 214

Light meter, 214

LOOKDOWN, 186, 187

LOOKUP, 183

LOW, 39, 182

- M -

mA, 339

Main routine, 222

Math Operations, 268

Memory

Memory Map, 204

Overwriting the program, 207

Memory Map, 203, 207

Metric units of measure, 339

Microcontroller, 11

Microfarads, 143

Microsecond, 105

Milli, 339

Millipede Project, 13

Millisecond, 39, 105

Motor Control product, 322

Music

Dot, 264

Rests, 259

Tempo, 260

Mystery numbers, 317

- N -

Nanometer, 197

Natural keys, 254

nc, 171

Negative charge, 335

Nested loop, 249

Nesting subroutines, 219

Neutral, 35

Neutron, 335

Nib, 45

Variable size, 45

No-connect, 171

Nominal value, 298

NPN transistor, 289

Nucleus, 335

Numbers

- Binary, 67, 179
- Decimal, 183
- Hexadecimal, 183
- Nuts and Volts of BASIC Stamps
 - columns, 323
- O -
- Octave, 254
- Offset, 158
- Ohm, 335
- Ohm's Law, 231, 338
- Omega Ω , 28
- ON GOSUB, 217
- ON GOTO, 217
- Open circuit, 62
- OR, 78
- OUTH, 178
- Overflow, 273
- Overwriting the program, 207
- P -
- Parallax Standard Servo
 - Caution, 96
 - Parts diagram, 95
- Parallel
 - bus, 177
 - device, 177
- Parallel capacitors, 224
- Parallel processing, 326
- PAUSE, 39
- PBASIC Language
 - AND, 78
 - Arguments, 39
 - Bit, 45
 - Byte, 45
 - CLREOL, 167
 - CR, 25
 - CRSRUP, 129
 - DATA, 255
 - DCD, 269
 - DEBUG, 39
 - DEBUGIN, 119
 - DEC, 207
 - DEC, 120
 - DIRH, 178
 - DO...LOOP, 39, 83, 123
 - DTMFOUT, 251
 - END, 63
 - EXIT, 282, 316
 - FOR...NEXT, 43, 124
 - FREQOUT, 247, 251
 - GOSUB, 216
 - GOTO, 217
 - HIGH, 39, 182
 - HOME, 76
 - IF...ELSEIF...ELSE, 75
 - IF...THEN, 78
 - IF...THEN...ELSE, 71
 - IN, 67
 - LOOKDOWN, 186, 187
 - LOOKUP, 183
 - LOW, 39, 182
 - Nib, 45

ON GOSUB, 217
OR, 78
OUTH, 178
PAUSE, 39
PIN, 162
PULSOUT, 105
RANDOM, 86
RCTIME, 149, 199
READ, 206
RETURN, 216
SELECT...CASE, 272
SERIN, 316
SEROUT, 316
STEP, 124
TOGGLE, 296
UNTIL, 83, 123
WHILE, 123
Word, 206
WRITE, 206, 207
PBASIC Operators
 ** (Multiply High), 270
 */ (Multiply Middle, 85, 270
 DCD, 269
 Order of execution, 268
 Parentheses, 268
Photoresistor, 197
Phototransistor, 198
Piezoelectric Speaker, 245
PIN, 162
Pin map, 170, 292

PIR Motion Sensor, 319
polling, 83
Polling, 80
Positive charge, 335
Potentiometer, 139
 AD5220 (digital), 292
Process Control, 329
Program
 Loops, nested, 249
 Overwriting, 207
Program Listings
 ActionTones.bs2, 248
 Ch01Prj01_Add1234.bs2, 25
 Ch01Prj02_FirstProgramYourTurn.bs2,
 26
 Ch02Prj01_Countdown.bs2, 60
 Ch03Prj01_TwoPlayerReactionTimer.bs2,
 91
 Ch04Prj01Soln1_KillSwitch.bs2, 136
 Ch07Prj01_Blinds_Control.bs2, 243
 Ch07Prj02_Blinds_Control_Extra.bs2,
 243
 Ch5Prj01_ControlServoWithPot.bs2, 166
 Ch6Prj01_FishAndChips.bs2, 193
 Ch8Prj01_PushButtonToneGenerator.bs2,
 286
 Ch9Ex01_SetTapToZero.bs2, 303
 Ch9Prj01_PhotoControlledDigitalPot.bs2,
 305
 ControlServoWithPot.bs2, 159
 DialDisplay.bs2, 189
 DigitalPotUpDown.bs2, 295

- DigitalPotUpDownWithToggle.bs2, 297
- DisplayDigits.bs2, 179
- DisplayDigitsWithLookup.bs2, 184
- DoReMiFaSolLaTiDo.bs2, 255
- FlashBothLeds.bs2, 49
- LedOnOff.bs2, 38
- LedOnOffTenTimes.bs2, 44
- LightMeter.bs2, 220
- MicroAlarmProto(Dev-009).bs2, 314
- MicroMusicWithRtttl.bs2, 277
- MusicWithMoreFeatures.bs2, 265
- NestedLoops.bs2, 250
- NotesAndDurations.bs2, 260
- PairsOfTones.bs2, 252
- PhototransistorAnalogToBinary.bs2, 227
- PolledRcTimer.bs2, 147
- PushbuttonControlledLed.bs2, 71
- PushbuttonControlOfTwoLeds.bs2, 75
- ReactionTimer.bs2, 81
- ReadLightMeasurementsFromEeprom.bs2, 210
- ReadPotWithRcTime.bs2, 152
- ReadPushbuttonState.bs2, 68
- SegmentTestWithHighLow.bs2, 178
- SelectCaseWithCharacters.bs2, 275
- SelectCaseWithValues.bs2, 273
- ServoCenter.bs2, 106
- ServoControlWithDebug.bs2, 122
- ServoVelocities.bs2, 127
- SimpleLookdown.bs2, 187
- SimpleLookup.bs2, 183
- SimpleSubroutines.bs2, 217
- SlowServoSignalsForLed.bs2, 113
- StoreLightMeasurementsInEeprom.bs2, 208
- TerminalControlledDigitalPot.bs2, 299
- TestBiColorLed.bs2, 55
- TestBinaryPhototransistor.bs2, 234
- TestPhototransistor.bs2, 201
- TestPiezoWithFreqout.bs2, 247
- TestSecondLed.bs2, 48
- ThreeServoPositions.bs2, 115
- TwinkleTwinkle.bs2, 257
- Proof of concept, 317
- Propeller microcontroller, 326
- Proton, 335
- Prototyping, 307
- Prototyping area, 31
- Pseudo code, 72
- Pseudo random, 87
- Pull-up and Pull-down resistors, 69
- PULSOUT, 105
- Pushbutton, 62
 - Active-high, 69
- R -
- RANDOM, 86
- RCTIME, 149, 199
- READ, 206
- Receive Windowpane, 120
- Receiving, 61
- Reference, 338
- Reference notch, 292

Remote, IR Remote Parts Kit, 324

Resistance, 335

Resistor, 28, 38

as I/O pin protection, 38

Color Code Values, 29

I/O pin protection, 69

Part drawing and schematic symbol, 29

Pull-up and Pull-down, 69

Variable, digital potentiometer, 292

Variable, Flexiforce, 319

Variable, potentiometer, 139

Rests, in music, 259

RETURN, 216

Reveille, 277

Ringtone Text Transfer Language,
271

Robotics with the Boe-Bot, 329

Rollover bug, 122

RTTTL, 271

- S -

Scaling, 158

Schematic, 35

Schematic symbol, 28

Schmitt trigger, 230

Seed value for pseudo random numbers,
86, 87

SELECT...CASE, 272

Sending, 61

Sensing, 61

Sensors products, 322

Serial 2x16 LCD, 320

SERIN, 316

SEROUT, 316

Servo

Caution Statement, 96

Power supply warning, 101

Timing diagram, 104

Servo Header Jumper, 97

Sharp notes, 254

Smart Sensors and Applications, 329

Smart Sensors and Applications
textbook, 321

Sockets, 31

Sound waves. *See*

Specification, 310

StampPlot LITE, 213

Stamps in Class Flowchart, 329

Stamps in Class Mini Projects, 330

STEP, 124

Subroutine, 216

Call, 218

Label, 217

Nesting limit, 219

Sunlight, 201

Superposition, 252

Switching, 61

- T -

Take Me Out To The Ball Game, 343

Tap, potentiometer, 298

Tempo, 260

Timing diagram, 104

TOGGLE, 296

Tokens, 203, 207, 287

Tolerance, 29, 298

Transistor, 198, 289

Schematic Symbol and Part Drawing, 289

Transistor-transistor logic (TTL), 230

Transmit Windowpane, 120

Transmitting, 61
TTL, 61
Twinkle Twinkle Little Star., 257

- U -

UNTIL, 83
USB drivers, 20

- V -

Variable range error, 122
Variable resistor, 319
 potentiometer (digital), 292
 potentiometer (single-turn), 139

Variables, 45

 Bit, 45
 Byte, 45
 DIRH, 178
 Initialization, 82
 Naming rules, 43
 Nib, 45
 OUTH, 178
 Overflow, 273

RAM storage, 209

Word, 45

Vdd, 338
Video tutorials, 330
Vin, 338
Virtual COM Port, 20
Visible light, 197
Volt, 335
Voltage, 35, 335
Voltage decay circuit, 145
Voltage divider, 155
Vss, 338

- W -

Wavelength, 197
WHILE, 123
Word, 45

 Variable size, 45

WORD modifier, 206
WRITE, 206, 207

- Ω -

Ω omega, 28

Parts and quantities are subject to change without notice. Parts may differ from what is shown in this picture. If you have any questions about your kit, please email stampsinclass@parallax.com.

