Smart Sensors and Applications

Student Guide

VERSION 1.0

PARALLAX

WARRANTY

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

14-DAY MONEY BACK GUARANTEE

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

COPYRIGHTS AND TRADEMARKS

This documentation is copyright 2006 by Parallax Inc. By downloading or obtaining a printed copy of this documentation or software you agree that it is to be used exclusively with Parallax products. Any other uses are not permitted and may represent a violation of Parallax copyrights, legally punishable according to Federal copyright or intellectual property laws. Any duplication of this documentation for commercial uses is expressly prohibited by Parallax Inc. Duplication for educational use is permitted, subject to the following conditions: the text, or any portion thereof, may not be duplicated for commercial use; it may be duplicated only for educational purposes when used solely in conjunction with Parallax products, and the user may recover from the student only the cost of duplication.

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

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

ISBN 1-928982-39-5

DISCLAIMER OF LIABILITY

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

3RD PRINTING

We maintain active web-based discussion forums for people interested in Parallax products. These lists are accessible from www.parallax.com:

- Propeller chip – This list is specifically for our customers using Propeller chips and products.
- BASIC Stamp – This list is widely utilized by engineers, hobbyists and students who share their BASIC Stamp projects and ask questions.
- Stamps in Class® – Created for educators and students, subscribers discuss the use of the Stamps in Class curriculum in their courses. The list provides an opportunity for both students and educators to ask questions and get answers.
- Parallax Educators – A private forum exclusively for educators and those who contribute to the development of Stamps in Class. Parallax created this group to obtain feedback on our curricula and to provide a place for educators to develop and obtain Teacher's Guides.
- Robotics – Designed for Parallax robots, this forum is intended to be an open dialogue for robotics enthusiasts. Topics include assembly, source code, expansion, and manual updates. The Boe-Bot®, Toddler®, SumoBot®, HexCrawler and QuadCrawler robots are discussed here.
- SX Microcontrollers and SX-Key – Discussion of programming the SX microcontroller with Parallax assembly language SX – Key® tools and 3rd party BASIC and C compilers.
- Javelin Stamp – Discussion of application and design using the Javelin Stamp, a Parallax module that is programmed using a subset of Sun Microsystems' Java® programming language.

ERRATA

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

Table of Contents

Preface iii
Introduction and Author's Note iii
Overview v
Before you Start v
The Stamps In Class Educational Series vi
Foreign Translations vii
Special Contributors vii

Chapter 1: The Parallax Serial LCD Display 1
LCDs in Products 2
The Parallax Serial LCD - Your Mobile Debug Terminal 2
Activity #1: Connecting and Testing the LCD 4
Activity #2: Displaying Simple Messages 8
Activity #3: Timer Application 17
Activity #4: Custom Characters and LCD Animation 19
Activity #5: Scrolling Text Across the Display 25
Summary 34

Chapter 2: The Ping))) Ultrasonic Distance Sensor 41
How Does the Ping))) Sensor Work? 41
Activity #1: Measuring Echo Time 42
Activity #2: Centimeter Measurements 46
Activity #3: Inch Measurements 49
Activity #4: Mobile Measurements 51
Activity #5: Temperature's Effect on the Speed of Sound 58
Summary 61

Chapter 3: The Memsic Dual-Axis Accelerometer 65
The MX2125 Accelerometer – How it Works 67
Activity #1: Connecting and Tilt-Testing the MX2125 68
Activity #2: Mobile Measurements 71
Activity #3: Scaling Down and Offsetting Input Values 76
Activity #4: Scaling to 1/100 g 83
Activity #5: Measuring 360° Vertical Rotation 85
Activity #6: Measure Tilt From the Horizontal 98
Summary 112

Chapter 4: The Hitachi HM55B Compass Module 119
Interpreting the Compass Measurements 119
Activity #1: Connecting and Testing the Compass Module 120
Activity #2: Compass Module Calibration 128

Activity #3: Testing the Calibration 138
Activity #4: Improve Compass Measurements by Averaging 143
Activity #5: Mobile Measurements 148
Summary 159

Chapter 5: Accelerometer Gaming Basics 167
Activity #1: PBASIC Graphic Character Display 168
Activity #2: Background Store and Refresh with EEPROM 179
Activity #3: Tilt the Bubble Graph 188
Activity #4: Game Control 196
SUMMARY 205

Chapter 6: More Accelerometer Projects 211
Activity #1: Measure Heights of Buildings, Trees, Etc. 211
Activity #2: Record and Playback 213
Activity #3: Use EEPROM to Toggle Modes 219
Activity #4: Remote Datalogging of Acceleration 223
Activity #5: RC Car Acceleration Study 230
Activity #6: Skateboard Trick Acceleration Study 240
Activity #7: Bicycle Distance 247
Summary 255

Chapter 7: LCD Bar Graphs for Distance and Tilt 261
Activity #1: Custom Character Swapping 261
Activity #2: Horizontal Bar Graphs for Ping))) Distance 271
Activity #3: Two-Axis Bar Graph for Accelerometer Tilt 281
Summary 294

Appendix A: ASCII Chart 303

Appendix B: Parallax Serial LCD Documentation 305

Appendix C: Hexadecimal Character Definitions 317

Appendix D: Parts Listing 321

Index 323

Preface

INTRODUCTION AND AUTHOR'S NOTE

The first time I saw the term "smart sensor" was in Tracy Allen's *Applied Sensors* text (then known as *Earth Measurements*). Tracy aptly applied this term to the DS1620 digital thermometer, which has built-in electronics that simplify microcontroller temperature measurements. In addition, it can remember settings it receives from a microcontroller and even function on its own as a thermostat controller.

In contrast to smart sensors, primitive sensors are devices or materials that have some electrical property that changes with some physical phenomenon. An example of a primitive sensor from *What's a Microcontroller?* is the cadmium sulfide photoresistor. Its resistance changes with light intensity. With the right circuit and program, microcontroller light measurements are possible. Other examples of common primitive sensors are current/voltage output temperature sensors, microphone transducers, and even the potentiometer, which is a rotational position sensor.

Inside every smart sensor is one or more primitive sensors and support circuitry. The thing that makes a smart sensor "smart" is the additional, built-in electronics. The electronics make these sensors able to do one or more of the following:

- Pre-process their measured values into meaningful quantities
- Communicate their measurements with digital signals and communication protocols
- Orchestrate the actions of primitive circuits and sensors to "take" measurements
- Make decisions and initiate action based on sensed conditions, independent of a microcontroller
- Remember calibration or configuration settings

During my first encounter with a smart sensor, I thought to myself, "Wow, an entire kit full of these smart sensors with a book could be REALLY interesting! I sure hope somebody does a kit and book like that soon..." Little did I know that "soon" would end up being almost six years later, and that "somebody" would turn out to be me. And if one of my bosses was to have told me back then that the kit would contain an accelerometer, ultrasonic rangefinder, digital compass, and a serial LCD for mobile measurements, I might just have come completely unglued. Since it was only recently possible for us to

put together such an awesome group of components into a single kit, I'd have to say it was worth the wait.

In keeping with the rest of the Stamps in Class tutorials, this book is a collection of activities, some of which cover basics, some more advanced, and some demonstrate applications or building blocks for various products and/or inventions. The first half of the book introduces each sensor, along with some mobile LCD displayed measurements. Then, the second half of the book has lots of applications for you to try, such as tilt video games, custom measurement tools, and diagnostic devices for hobby and sports pursuits. The page limit to keep these books inside our packaging is 350, and it was kind of difficult to stop there. Additional Smart Sensor activities for the Boe-Bot robot can be found in the Stamps in Class forum at www.parallax.com.

While this book covers the basics and demonstrates some example applications, it really only scratches the surface of what you can do with these devices. The main purpose of this book is to provide some building blocks and ideas for future class projects and inventions. For example, after finishing chapter 3, our book reviewer Kris Magri put her Board of Education with the accelerometer and LCD on her dashboard, and now her car has a flatland acceleration meter along with the speedometer. With a few modifications to the code, it could be made into a rollover warning system for 4-wheel drive. After looking at the mechanical sighting device used to predict avalanche conditions in mountainous areas based on hill incline, Ken Gracey whipped up the digital version in one night with the same parts that went onto Kris’s dashboard.

The dashboard acceleration and avalanche risk meters are just two novel examples of the many applications, projects, and inventions that the Smart Sensors kit and text can inspire. We'd like to see what you do with your kit on the Stamps in Class forum. It doesn't matter whether you think your project turned out to be cool, unique, corny, or whatever. Just take a few minutes to post things you've made with theses smart sensors to http://forums.parallax.com/forums/ → Stamps in Class. Make sure to include a few snapshots, a brief description, and preferably the schematic and PBASIC program.

So, have fun with this kit and book, and we'll look forward to seeing your inventions on the Stamps in Class forum.

OVERVIEW

The smart sensors kit contains four devices that, when used with the BASIC Stamp and Board of Education or HomeWork Board, can be building blocks for a variety of inventions and student projects. Here is a list of the devices:

- Parallax 2x16 Serial LCD
- Ping))) Ultrasonic Rangefinder
- Memsic 2125 2-Axis Accelerometer
- Hitachi HM55B Compass Module

Aside from providing both the equipment and how-to information for student projects, this text has two major emphases that provide theory, examples, and required calculations, which can be used to reinforce a variety of measurement, physics/engineering, and trigonometric concepts. These emphases are:

- Math techniques for scaling raw sensor values into measurements that are meaningful because they are expressed in common unit systems.
- Interpreting the projection of gravity and magnetic vector fields onto Cartesian sensing axes.

BEFORE YOU START

To perform the experiments in this text, you will need to have your Board of Education or HomeWork Board connected to your computer, the BASIC Stamp Editor software installed, and to have verified the communication between your computer and your BASIC Stamp. For detailed instructions, see *What's a Microcontroller?* which is available as a free download from www.parallax.com. You will also need the parts contained in the Smart Sensors Parts Kit. For a full listing of system, software, and hardware requirements, see Appendix D.

THE STAMPS IN CLASS EDUCATIONAL SERIES

The Stamps in Class series of texts and kits provides affordable resources for electronics and engineering education. All of the books listed are available for free download from www.parallax.com. The versions cited below were current at the time of this printing. Please check our web sites www.parallax.com or www.stampsinclass.com for the latest revisions; we continually strive to improve our educational program.

Stamps in Class Student Guides:

What's a Microcontroller? is the recommended entry level text to the Stamps In Class educational series. Some students instead start with *Robotics with the Boe-Bot*, also designed for beginners.

"*What's a Microcontroller?*", Student Guide, Version 2.2, Parallax Inc., 2004
"*Robotics with the Boe-Bot*", Student Guide, Version 2.2, Parallax Inc., 2004

You may continue on with other Educational Project topics, or you may wish to explore our other Robotics Kits.

Educational Project Kits:

The following texts and kits provides a variety of activities that are useful to hobbyists, inventors and product designers interested in trying a wide range of projects.

"*Smart Sensors and Applications*", Student Guide, Version 1.0, Parallax Inc., 2006
"*Process Control*", Student Guide, Version 1.0, Parallax Inc., 2006
"*Applied Sensors*", Student Guide, Version 1.3, Parallax Inc., 2003
"*Basic Analog and Digital*", Student Guide, Version 1.3, Parallax Inc., 2004
"*Understanding Signals*", Student Guide, Version 1.0, Parallax Inc., 2003

Robotics Kits:

To gain experience with robotics, consider continuing with the following Stamps in Class student guides, each of which has a corresponding robot kit:

"*IR Remote for the Boe-Bot*", Student Guide, Version 1.1, Parallax Inc., 2004
"*Applied Robotics with the SumoBot*", Student Guide, Version 1.0, Parallax Inc., 2005
"*Advanced Robotics: with the Toddler*", Student Guide, Version 1.2, Parallax Inc., 2003

Reference

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

"*BASIC Stamp Manual*", Version 2.2, Parallax Inc., 2005

FOREIGN TRANSLATIONS

Parallax educational texts may be translated to other languages with our permission (e-mail translations@parallax.com). If you plan on doing any translations please contact us so we can provide the correctly-formatted MS Word documents, images, etc. We also maintain a private discussion group for Parallax translators which you may join. This will ensure that you are kept current on our frequent text revisions.

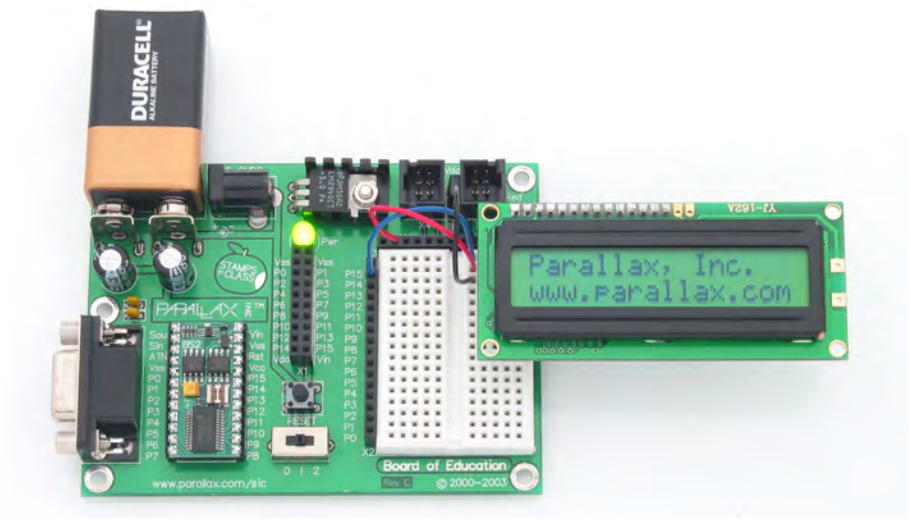
SPECIAL CONTRIBUTORS

Parallax Inc. would like to recognize their Education Team members: Project Manager Aristides Alvarez, Technical Illustrator Rich Allred, Graphic Designer Larissa Crittenden, Technical Reviewer Kris Magri, and Technical Editor Stephanie Lindsay. In addition, thanks go to customer Steve Nicholson for test-driving most of the activities. As always, special thanks go to Ken Gracey, the founder of Parallax Inc.'s Stamps in Class educational program.

Chapter 1: The Parallax Serial LCD Display

Displaying the information a sensor sends in a readable format has many uses, and in some applications, it's all that matters. The digital thermometer is a common example that can be found in many households. Inside each digital thermometer, there is a temperature probe, a microcontroller, and a liquid crystal display (LCD) for displaying the measurements. The BASIC Stamp microcontroller and Parallax Serial LCD shown in Figure 1-1 can provide the microcontroller and display elements for this type of product. This setup is also great for displaying mobile measurements, making it possible to disconnect your board from the PC and Debug Terminal and field-test your smart sensors.

Figure 1-1: BASIC Stamp, Board of Education, and Parallax Serial LCD



The activities in this chapter introduce some Parallax LCD basics, like connecting the LCD to the BASIC Stamp, turning it on and off, placing its cursor, and displaying text and digits. Later chapters will introduce creating and animating custom characters and displaying scrolling messages.

LCDS IN PRODUCTS

The products shown in Figure 1-2 all have liquid crystal displays. They are easy to read, and the smaller ones consume very little power. Think about how many products you own with liquid crystal displays. Think also as you go through these activities about the various BASIC Stamp projects, prototypes and inventions you've got in the works, and how a serial LCD might enhance or help them to completion.

Figure 1-2: Product Examples with LCD Displays

Clockwise from top-left: cell phone, portable GPS unit, calculator, digital multimeter, office clock, laptop computer, oscilloscope, office phone.



THE PARALLAX SERIAL LCD - YOUR MOBILE DEBUG TERMINAL

If you've worked through any of the other Stamps in Class texts, you're probably familiar with what a valuable tool the Debug Terminal can be. The Debug Terminal is a window that you can use to make your computer display messages it receives from the BASIC Stamp. It's especially useful for displaying diagnostic messages and variable values,

making it easier to isolate program bugs. It's also a handy tool for testing circuits, sensors, and more.

The Debug Terminal has one drawback, and that's the serial cable connection. Consider how many times it wasn't convenient to have your board connected to the computer to test a sensor, or find out what the Boe-Bot robot was "seeing" with its infrared object detectors in another room. These are all situations that can be remedied with the Parallax Serial LCD shown in Figure 1-3. Once you've built up a sensor circuit on the Board of Education, you can use a battery and the Parallax Serial LCD to take the setup as far away from your programming station as you want, all the while displaying sensor measurements and other diagnostic information.



Figure 1-3
Parallax (2×16)
Serial LCD

The Parallax 2×16 Serial LCD has two sixteen-character-wide rows for displaying messages. The display is controlled by serial messages from the BASIC Stamp. The BASIC Stamp sends these messages from a single I/O pin that is connected to the LCD's serial input. There are two versions, standard and backlit:

Version	Parallax Part #
Standard	27976
Backlit	27977

Serial Vs Parallel LCDs

The Parallel LCD is probably the most common type of LCD. It typically requires a minimum of 6 I/O pins for the BASIC Stamp to control. Also, unless you are using a BASIC Stamp 2p, 2pe, or 2px, the code for controlling them tends to be more complex than serial LCD code.

The serial LCD is actually just a parallel LCD with an extra microcontroller. The extra microcontroller on the serial LCD converts the serial messages from the BASIC Stamp to the parallel messages that control the parallel LCD.

ACTIVITY #1: CONNECTING AND TESTING THE LCD

Along with the electrical connections and some simple PBASIC test programs for the Parallax Serial LCD, this activity introduces the **SEROUT** command. It also demonstrates how **DEBUG** is just a special case of **SEROUT**. This is especially useful for working with your serial LCD because you can take many of the **DEBUG** command arguments and use them with the **SEROUT** command to control and format the information your LCD displays.

Parts Required

(1) Parallax 2×16 Serial LCD
(3) Jumper wires

In addition to the Parallax Serial LCD and three wires, it will be especially important to have the Parallax Serial LCD Documentation (included as Appendix B in this text). Although it's only a few pages, it has a long list of values you can send to your LCD to make it perform functions similar to those you've used with the Debug Terminal. Cursor control, carriage returns, clear screen and so on, all have their own special codes. In some cases, these codes are identical to the ones for the Debug Terminal; in other cases, they are quite different.

Building the Serial LCD Circuit

Connecting the Parallax Serial LCD to the BASIC Stamp is amazingly simple, as shown in Figure 1-4. You only need to make three connections: one for power, one for ground, and one for signal. The LCD's RX pin is for the signal and should be connected to a BASIC Stamp I/O pin. In this activity, we will use P14. The LCD's GND pin should be connected to Vss on the Board of Education, and the LCD's 5 V pin should be connected to Vdd.



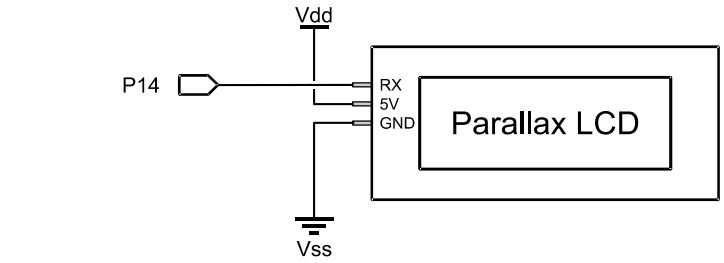
CAUTIONS: Wiring mistakes can damage this LCD.

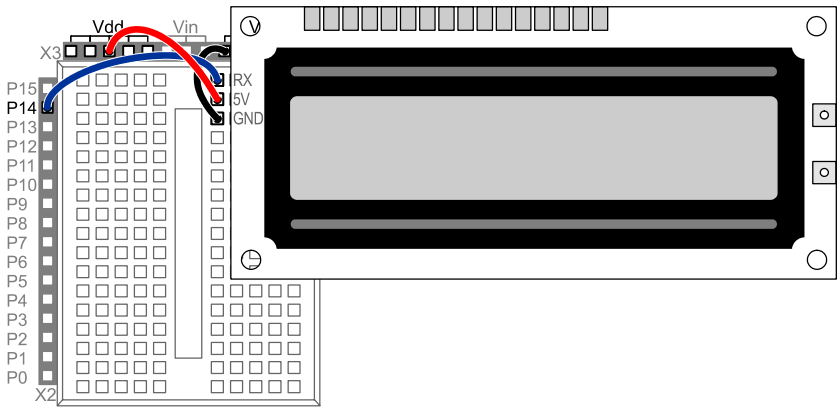
Rev D and earlier models if this LCD had five pins. If you have a 5-pin model, please see Figure B-1 on page 306 to verify the correct pins to use in the circuits in this book.

The five-pin version is NOT pin compatible with Scott Edwards or Matrix Orbital models. If you have used other brands of serial LCDs before, be aware that this LCD's pinout is different. Don't make the mistake of using the same wiring that you used for other models.

- √ Disconnect power from your Board of Education.
- √ Connect the Board of Education's Vss socket to the LCD's GND pin.
- √ Connect the Board of Education's P14 socket to the LCD's RX pin, as shown in Figure 1-4.
- √ Connect the Board of Education's Vdd socket to the LCD's 5V pin
- √ Do not turn the power back on yet.

Figure 1-4: Schematic and Wiring Diagram





Testing the Serial LCD

The Parallax Serial LCD has a self-test mode you can use to make sure it's in working order and that the contrast is properly set. Figure 1-5 shows the back of the LCD module. The switches labeled (SW1 and SW2) are for self-test mode and baud rate adjustment, and there is a contrast adjustment potentiometer labeled "INCREASE CONTRAST."

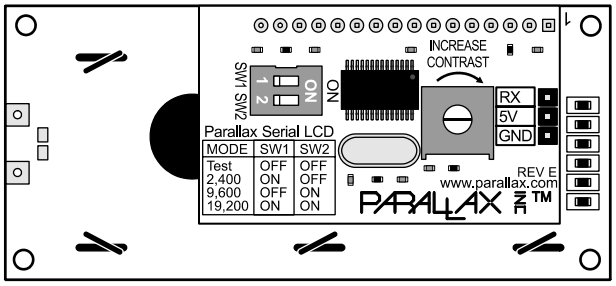


Figure 1-5
LCD Module - Back View

- √ The power to your board should still be off.
- √ Find SW1 and SW2 on the underside of the LCD module shown in Figure 1-6 .
- √ Set SW1 off.
- √ Set SW2 off.
- √ Turn the power back on now.

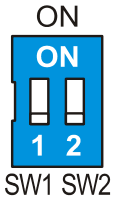


Figure 1-6
Setting Baud Rate Switches to Self-test Mode

- √ When you turn the power back on, the LCD should display the text "Parallax, Inc." on the top line (Line 0) and "www.parallax.com" on the bottom line (Line 1), as you can also see in Figure 1-3. If you leave the LCD in this mode for a while, a custom character reminiscent of 1980's video games will appear and eat all the text.

√ If the display seems dim or looks blank, there is a contrast adjustment potentiometer shown in Figure 1-7 that you can turn with a screwdriver. If the display is already clear and all the characters look good, you probably don't need to adjust it. If the characters are too dim, or they appear in gray boxes, adjusting the potentiometer should help.

√ Adjust the contrast potentiometer if needed.

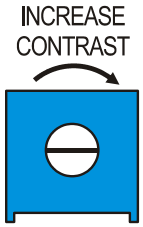


Figure 1-7
Contrast Adjustment Potentiometer

Adjusting the LCD to Receive Messages from the BASIC Stamp

Serial communication involves a baud rate. That's the number of bits per second (bps) the sender transmits, and the receiver has to be ready to receive data at the same baud rate. In this chapter's activities, the BASIC Stamp will be programmed to send messages to the LCD at 9600 bps. You can adjust the same switches you used for the LCD self-test to set this baud rate.

√ Turn the Board of Education power off.
√ Leave SW1 in the OFF position.
√ Set SW2 to ON as shown in Figure 1-8.
√ Turn the power back on now.

The screen will remain blank until you program the BASIC Stamp 2 to control the display.

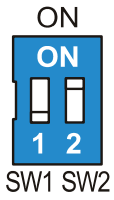


Figure 1-8
Baud Rate Switches to 9600 bps

Figure 1-9 shows the mode table printed on the back of the Parallax Serial LCD. If you want to send messages at other baud rates, (2400 or 19,200 bps), use this table and adjust SW1 and SW2 accordingly.

Parallax Serial LCD

MODE	SW1	SW2
Test	OFF	OFF
2,400	ON	OFF
9,600	OFF	ON
19,200	ON	ON

Figure 1-9
Baud Rate Switch Settings

ACTIVITY #2: DISPLAYING SIMPLE MESSAGES

As mentioned earlier, the commands that send text, numbers, formatters and control codes (control characters) to a serial LCD are related to the **DEBUG** command. In fact, the **DEBUG** command is just a special version of a more general command called **SEROUT**. The **SEROUT** command has many uses, some of which include sending messages to serial LCDs, other BASIC Stamp modules, and computers.

In this activity, you will program the BASIC Stamp to make the LCD display text messages and numeric values. As a first step into animation, you will also modify the programs to flash the text and numbers on and off. The **SEROUT** command will be your tool for accomplishing these tasks. You will use the **SEROUT** command to send text, numbers, control codes, and formatters to the Parallax Serial LCD. As you will soon see, the text, numbers, and formatters are identical to ones you use with the **DEBUG** command. The control codes will be a little different, but with some practice, they'll be just as easy to use as **CR**, **CLS**, **HOME**, and **CRSRXY**. (If you are not familiar with **CRSRXY**, you can learn more about it in Chapter 5, Activity #1.)

The minimal version of the **SEROUT** command's syntax looks like this:

SEROUT ***Pin, BaudMode,*** **[** ***DataItem,*** **{*****DataItem, ...*****}** **]**

In our programs, the ***Pin*** argument has to be 14 since the LCD's RX (receive data) pin is connected to BASIC Stamp I/O pin P14.

The ***BaudMode*** argument is a value that tells the BASIC Stamp how fast to send the serial data, and it determines some of the serial signal characteristics as well. The BASIC Stamp Editor's Help program has tables that give the ***BaudMode*** values for common baud rates and signals. It turns out that 84 is the ***BaudMode*** argument for 9600 bits per second (bps), 8 data bits, no parity, true signal. This is exactly what the Parallax Serial LCD is designed to receive.

DataItem arguments can be the text between quotes like "Hello". They can also be control characters like **CR**, **CLS**, or values, with or without the formatters like **DEC**, **BIN**, and **?**. If they are sent with formatters, they are sent as the characters that represent the value. If they are sent without formatters, they will be sent as values, like 22, 12, and 13. We can send unformatted values like these to the LCD, which will interpret them as control codes.



More about SEROUT

If you want to try using the Debug Terminal with **SEROUT** instead of **DEBUG**, first open it from the toolbar via Run → Debug → New. Next, select Run → Identify to see which port your BASIC Stamp is using. Then, in the Debug Terminal, set the Com Port to match. Note that you can also change the Debug Terminal's Baud Rate and other communication parameters.

There's lots more to learn about the **SEROUT**. Both the BASIC Stamp Manual and the BASIC Stamp Editor's PBASIC Syntax Guide give the **SEROUT** command extensive coverage. The BASIC Stamp Manual is available for free download from www.parallax.com → Downloads → Documentation. If your BASIC Stamp Editor supports PBASIC 2.5, you probably already have the PBASIC Syntax guide. To access it, simply select Index from the BASIC Stamp Editor's Help menu.

Simple Text Messages and Control Codes

Unlike the Debug Terminal, the serial LCD needs to be turned on with a command from the BASIC Stamp. The LCD has to receive the value 22 from the BASIC Stamp to

activate its display. Here's the PBASIC command for sending the serial LCD the value 22:

```
SEROUT 14, 84, [22]
```

Used in this way, **22** is an example of an LCD control code. Here's a list of some more basic control codes:

- **12** clears the display. Note: always follow with **PAUSE 5** to give the LCD time to clear the display.
- **13** is a carriage return; it sends the cursor to the next line.
- **21** turns the LCD off.
- **22** turns the LCD on.



Commands to turn the backlighting on and off (for the backlit LCD only):

Some LCDs have backlighting so that you can read them when it's dark. If you have the backlit version of the Parallax Serial LCD (part number 27977), you can control the backlighting with these values:

- **17** turns the backlighting on.
- **18** turns the backlighting off.



In PBASIC, CR is a predefined constant for the value 13. Whenever you use the constant **CR** in a **DEBUG** command, it sends the value 13 to the Debug Terminal. The Debug Terminal moves the cursor to the beginning of the next line whenever it receives the value 13. In this case, the two commands below are the equivalent:

```
SEROUT 14, 84, ["See this?", CR, "The LCD works!"]

SEROUT 14, 84, ["See this?", 13, "The LCD works!"]
```

While this works with CR, it does not work for other predefined PBASIC constants. For example, **CLS**, which is a predefined constant for the number 0, does not clear the LCD display. The Parallax Serial LCD equivalent of **CLS** is **12**. Likewise, **HOME**, which is a predefined constant for the value 1, does not send the cursor to the top left "home" character in the LCD display. The control code **128** does that for the Parallax Serial LCD.

Example Program - LcdTestMessage.bs2

√ Enter, save, and run LcdTestMessage.bs2. Verify that it displayed the message "See this?" on Line 0 and "The LCD works!" on Line 1, as in Figure 1-10.

```
' Smart Sensors and Applications - LcdTestMessage.bs2
' Display a test message on the Parallax Serial LCD.

' {$STAMP BS2}                              ' Target device = BASIC Stamp 2
' {$PBASIC 2.5}                             ' Language      = PBASIC 2.5

SEROUT 14, 84, [22, 12]                     ' Initialize LCD
PAUSE 5

SEROUT 14, 84, ["See this?", 13,            ' Text message, carriage return
                "The LCD works!"]           ' more text on Line 1.
END                                         ' Program end
```

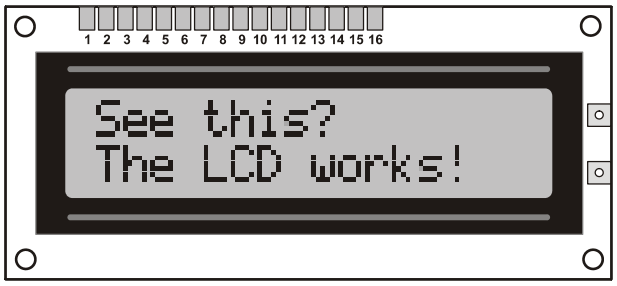


Figure 1-10
Text Display

If the LCD didn't display properly: Double-check your wiring, your program, and the SW settings on the back of the LCD. Also try disconnecting and reconnecting power to your Board of Education. If needed, Go through the check-marked instructions leading up to this program, and verify that each one was completed correctly.

Your Turn - Control Codes to Make the Display Flash On/Off

Remember that **22** turns the display on, and **21** turns it off? You can use these control codes to make the text flash on and off.

√ Replace the **END** command in LcdTestMessage.bs2 with this code block.

```
DO                                  ' Start DO...LOOP code block
  PAUSE 600                         ' 6/10 second delay
  SEROUT 14, 84, [22]               ' Turn display on
  PAUSE 400                         ' 4/10 second delay
  SEROUT 14, 84, [21]               ' Turn display off
LOOP                                ' Repeat DO...LOOP code block
```

√ Run the modified program and note the effect.

Display Numbers with Formatters

Most of the formatters that worked for displaying numbers with the Debug Terminal can also be used with the Parallax Serial LCD. The **DEC** formatter is probably the most useful, but you can also use **DIG**, **REP**, **ASC**, **BIN**, **HEX, SDEC**, and most of the others. For example, if you want to display the decimal value of a variable named **counter**, you can use commands like this:

```
SEROUT 14, 84, [DEC counter]
```

Example Program - LcdTestNumbers.bs2

Aside from demonstrating that you can display variable values on the serial LCD, this program also shows what happens if the program sends more than 16 printable characters to Line 0. It wraps to Line 1. Also, after printing sixteen more characters and filling Line 1, the text will wrap again, to Line 0.

√ Enter, save, and run LcdTestNumbers.bs2

```
' Smart Sensors and Applications - LcdTestNumbers.bs2
' Display number values with the Parallax Serial LCD.

' {$STAMP BS2}                              ' Target device = BASIC Stamp 2
' {$PBASIC 2.5}                             ' Language      = PBASIC 2.5

counter        VAR     Byte                 ' FOR...NEXT loop index

SEROUT 14, 84, [22, 12]                     ' Initialize LCD
PAUSE 5                                     ' 5 ms delay for clearing display

FOR counter = 0 TO 12                       ' Count to 12; increment at 1/2 s

  SEROUT 14, 84, [DEC counter, " "]
  PAUSE 500

NEXT

END                                         ' Program end
```

√ Verify that the display resembles Figure 1-11.

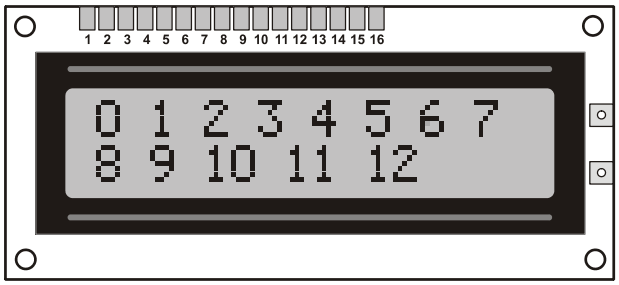


Figure 1-11
Display Numbers

Your Turn - Other Formatters

- √ Try replacing **DEC** with **DEC2** and observe what happens.
- √ Repeat with the **?** formatter.
- √ If necessary, look these commands up in either the BASIC Stamp Manual or the BASIC Stamp Editor's Help. Also try them in the Debug Terminal.
- √ What are the similarities and differences between using these formatters in the Debug Terminal and using them in the Parallax Serial LCD?

Control Codes for Cursor Positioning

The LCD's control codes are different from the **DEBUG** command's control characters. For example, **HOME**, and **CRSRXY** just don't have the same effect they do with the Debug Terminal. However, there are cursor commands for the Parallax Serial LCD that you can use to control the cursor's X and Y coordinates. You can also send the cursor to the top-left "home position". Take a look at the LCD documentation's Command Set section beginning on page 312. It lists all the valid control commands for the LCD; below are a few examples from the list that control cursor position.

- **8** Cursor left
- **9** Cursor right
- **10** Cursor down (bottom line will wrap to top line)
- **128** to **143** Position cursor on Line 0, character 0 to 15
- **148** to **163** Position cursor on Line 1, character 0 to 15

The values from **128** to **143** and **148** to **163** are particularly useful. Figure 1-12 shows where each value places the cursor. You can use values from **128** to **143** to place the cursor at characters 0 to 15 on the top line of the LCD (Line 0). Likewise, you can use values from **148** to **163** to place the cursor on characters 0 to 15 of the bottom line (Line 1).

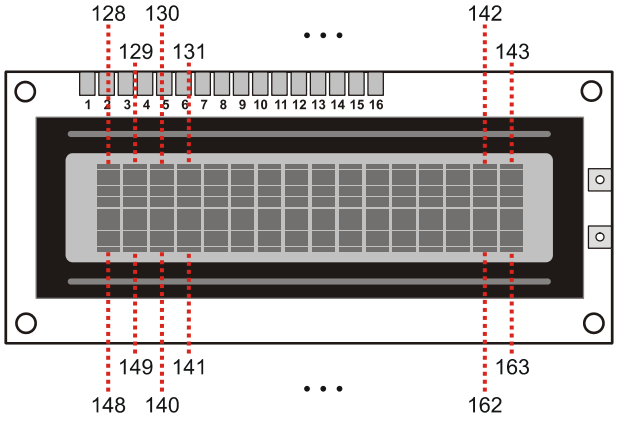


Figure 1-12
Text Display

After placing the cursor, the next character that you send to the LCD will be displayed at that location. For example, here is a **SEROUT** command with an optional ***Pace*** argument value set to 200 ms. This command will display the characters "L", "I", "N", "E", "-", and "0", evenly spaced across the top line, one character every 200 ms.

```
SEROUT 14, 84, 200, [128, "L",
                     131, "I",
                     134, "N",
                     137, "E",
                     140, "-",
                     143, "0"]
```

If displaying multiple characters after the giving an initial position, the LCD will still automatically shift the cursor to the right after each character. For example, you can also place the cursor on character 7 of the top line and then display "ALL", then move the cursor to character 6 of the bottom line and display "DONE!" like this:

```
SEROUT 14, 84, [135, "ALL", 154, "DONE!"]
```

Here's a code block that will make the text "Line 1" slide across the display's bottom line, from right to left.

```
FOR index = 9 TO 0
          ' IMPORTANT: Leave a space after the 1 in "Line 1 "
  SEROUT 14, 84, [148 + index, "Line 1 "]
  PAUSE 100
NEXT
```



Erasing Characters

You can always erase a character by placing the cursor where you want it and then sending the space " " character to overwrite whatever might be there. This is why the text "Line 1 " has a space after the "1" character, to erase the characters to its right as the text moves left.

Example Program - CursorPositions.bs2

This program introduces a few basic cursor placement tricks.

- √ Look over CursorPositions.bs2 and try to predict what the program will make the LCD display do. Also try to predict the sequence and timing.
- √ Enter, save, and run CursorPositions.bs2.
- √ Compare the LCD display's behavior to your predictions.

```
' Smart Sensors and Applications - CursorPositions.bs2
' Display number values with the Parallax Serial LCD.

' {$STAMP BS2}                              ' Target device = BASIC Stamp 2
' {$PBASIC 2.5}                             ' Language      = PBASIC 2.5

index          VAR     Nib                 ' FOR...NEXT loop index
character      VAR     Byte                ' Character storage
offset         VAR     Byte                ' Offset value

SEROUT 14, 84, [22, 12]                    ' Initialize LCD
PAUSE 500                                  ' 1/2 second delay

' Display evenly spaced characters on Line 0 every 200 ms.
SEROUT 14, 84, 200, [128, "L",
                     131, "I",
                     134, "N",
                     137, "E",
                     140, "-",
                     143, "1"]
PAUSE 1000
```

```
' Shift "Line 1" across Line 1 right to left, then left to right.
FOR index = 9 TO 0
  ' IMPORTANT: Make sure there's a space after the 1 in "Line 1 ".
  SEROUT 14, 84, [148 + index, "Line 1 "]
  PAUSE 100
NEXT

FOR index = 0 TO 9
  ' IMPORTANT: Make sure there's a space between the " and the L character.
  SEROUT 14, 84, [148 + index, " Line 1"]
  PAUSE 250
NEXT

PAUSE 1000                                  ' 1 second delay

' Clear LCD, then display then Display "ALL DONE" in center and flash 5 times
SEROUT 14, 84, [12]: PAUSE 5                ' Clear LCD
SEROUT 14, 84, [135, "ALL", 13, 154, "DONE!"]' "ALL" and "DONE" centered

FOR index = 1 TO 4                          ' Flash display 5 times
  SEROUT 14, 84, 500, [21, 22]
NEXT
END                                         ' Program end
```

Your Turn - More Positioning

More elaborate displays can benefit from loops and lookup tables. Here is an example of a "T E S T" display in a loop and with the help of a couple of **`LOOKUP`** commands. Note that you can control the position of each character's placement by adjusting the values for **`offset`** in the second **`LOOKUP`** command's list of values.

```
PAUSE 1000
SEROUT 14, 84, [12]: PAUSE 5       ' Clear display
SEROUT 14, 84, ["This is a", 13]   ' Text & CR

FOR index = 0 TO 3                 ' Character display sequence
  PAUSE 600
  LOOKUP index, ["T", "E", "S", "T"], character
  LOOKUP index, [  1,   5,  9,  13], offset
  SEROUT 14, 84, [(148 + offset), character]
NEXT
```

√ Try it!

ACTIVITY #3: TIMER APPLICATION

This activity applies the techniques introduced in Activity #2 to an hour-minute-second timer.

Displaying Time Elapsed

Here is a code block that starts the LCD, clears the screen, and places some display characters on the LCD that will not change. The rest of the program can then display changing hour, minute, and second number values next to the stationary "h", "m", and "s" characters.

```
SEROUT 14, 84, [22, 12]                 ' Start LCD & clear display
PAUSE 5                                 ' Pause 5 ms for clear display

SEROUT 14, 84, ["Time Elapsed...", 13]  ' Text + carriage return
SEROUT 14, 84, ["  h    m    s"]        ' Text on second line
```

For this application, control codes for cursor placement can be particularly useful. For example, the cursor can be placed on Line 1, character 0 before sending the two-digit decimal value of hours. The cursor can then be moved to Line 1, character 5 to display the minutes, and then to Line 1, character 10 to display the seconds.

Here is a single **SEROUT** command that displays all three variable values, each at the correct location:

```
SEROUT 14, 84, [ 148, DEC2 hours,
                 153, DEC2 minutes,
                 158, DEC2 seconds ]
```

The next example program applies this concept with just the BASIC Stamp module's timing abilities. The accuracy isn't digital wristwatch quality by a long shot; however, it is good enough for showing how the time display can work with character positioning. For higher accuracy, try incorporating the DS1302 timekeeping chip. It's available from www.parallax.com, just enter DS1302 into the search field.

Example Program - LcdTimer.bs2

This example program displays hours, minutes and seconds elapsed with the Parallax Serial LCD. By pressing the RESET button on the Board of Education, you can restart the timer.

√ Enter, save, and run LcdTimer.bs2.

√ Verify that the display works as advertised.

```
' Smart Sensors and Applications - LcdTimer.bs2
' Display elapsed time with BS2 and Parallax Serial LCD.

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

hours          VAR     Byte                  ' Stores hours
minutes        VAR     Byte                  ' Stores minutes
seconds        VAR     Byte                  ' Stores seconds

SEROUT 14, 84, [22, 12]                      ' Start LCD & clear display
PAUSE 5                                      ' Pause 5 ms for clear display

SEROUT 14, 84, ["Time Elapsed...", 13]       ' Text + carriage return
SEROUT 14, 84, ["  h    m    s"]             ' Text on second line

DO                                           ' Main Routine
  ' Calculate hours, minutes, seconds
  IF seconds = 60 THEN seconds = 0: minutes = minutes + 1
  IF minutes = 60 THEN minutes = 0: hours = hours + 1
  IF hours   = 24 THEN hours   = 0

  ' Display digits on LCD on Line 1.  The values 148, 153, 158
  ' place the cursor at character 0, 5, and 10 for the time values.
  SEROUT 14, 84, [148, DEC2 hours,
                  153, DEC2 minutes,
                  158, DEC2 seconds ]

  PAUSE 991                                  ' Pause + overhead ~ 1 second
  seconds = seconds + 1                      ' Increment second counter
LOOP                                         ' Repeat Main Routine
```

Your Turn - Defining Control Codes with Constants

Up to this point, the LCD control codes have been decimal values. However, when you are writing or reading a long program, memorizing all those control code values can be tedious. It's better to declare a constant for each control code at the beginning of the program. Then, use the constant names instead of numbers. You can also do the same with the ***BaudMode*** value, and then add a **`PIN`** directive for I/O pin P14 as well. Here is an example:

```
       LcdPin         PIN     14          ' LCD I/O pin

       T9600          CON     84          ' True, 8-bits, no parity, 9600

       LcdCls         CON     12          ' Form feed -> clear screen
```

```
LcdCr          CON     13          ' Carriage return
LcdOff         CON     21          ' Turns display off
LcdOn          CON     22          ' Turns display on
Line0          CON     128         ' Line 0, character 0
Line1          CON     148         ' Line 1, character 0
```

These declarations will make your code easier to understand, which is especially important if you decide to make changes to your program after not having looked at it for several months. For example, the first **SEROUT** command can be rewritten like this:

```
SEROUT LcdPin, T9600, [LcdOn, LcdCls]
```

The **SEROUT** command in LcdTimer.bs2 that displays the numbers on Line 1 of the LCD can be rewritten like this:

```
SEROUT LcdPin, T9600, [(Line1 + 0), DEC2 hours,
                       (Line1 + 5), DEC2 minutes,
                       (Line1 + 10), DEC2 seconds]
```

√ Save LcdTimer.bs2 under a new name.
√ Add descriptive constants to your program.
√ Replace as many numbers as you can with meaningful constant names.
√ Run your program and troubleshoot as needed.

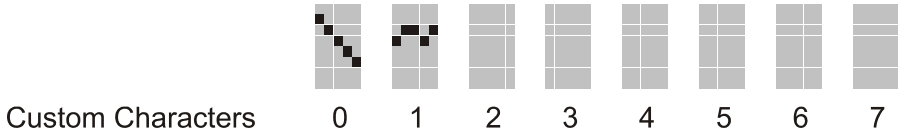
ACTIVITY #4: CUSTOM CHARACTERS AND LCD ANIMATION

While not every picture saves a thousand words, even the ones that only save a sentence or two are useful when you've got just 32 character spaces to work with. One example of a useful picture is that hourglass cursor your computer screen uses to let you know the program is busy. This simple animated icon works much better than a message somewhere on the screen that says, "please wait, the program is busy...". This activity uses an hourglass to introduce techniques for defining, storing, displaying, and animating custom characters.

Custom Characters in the Parallax LCD

The Parallax Serial LCD has room set aside for eight custom characters shown in Figure 1-13. To display Custom Character 0, just send the LCD the value **0** with the **SEROUT** command. Likewise, to display Custom Character 1, just send a **1**, to display Custom Character 2, send a **2**, and so on. Note that Custom Characters 0 and 1 are pre-configured to be the backslash and tilde. Here is an example **SEROUT** command that displays both of them - **SEROUT 14, 84, [0, 1]**.

Figure 1-13: Predefined Custom Characters O (Backslash) and 1 (Tilde)



Example Program: PredfinedCustomCharacters.bs2

This example sends the serial LCD the two commands to make it display Custom Characters 0 and 1, the backslash "\" and tilde "~".

√ Enter and run the program, and verify that it displays the backslash and tilde.

```
' Smart Sensors and Applications - PredefinedCustomCharacters.bs2

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

SEROUT 14, 84, [22, 12]                     ' Initialize LCD
PAUSE 5                                     ' 5 ms delay for clearing display

' Display pre-defined custom characters: "\" (custom-character-0) and
' "~" (custom-character-1).

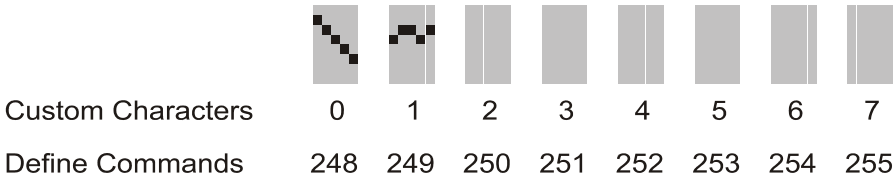
SEROUT 14, 84, [0, 1]
```

Defining (and Redefining) Custom Characters

The Parallax Serial LCD's custom characters are stored in its RAM. To define one of its eight custom characters, your **SEROUT** command has to tell the LCD which of the eight custom characters you are defining and then describe the on/off states of each pixel in the character. Each character has 40 pixels, 8 pixels tall by 5 pixels wide.

Figure 6-14 shows the Define commands you can send the LCD to tell it which custom character you are about to define. You can also think about it like this: to tell the LCD which custom character you are defining, send the value of the custom character plus 248. For example, if you want to define Custom Character 0, send **248**, if you want to define Custom Character 1, send **249**, and so on up to **255** for Custom Character 7.

Figure 1-14: Custom Character Define Commands



After sending the code that tells the LCD which custom character you are about to define, you have to then send eight bytes that describe the character. The LCD uses the lowest five bits of each byte it receives to describe each five-pixel-wide line in the character. Figure 1-15 shows an example of defining Custom Character 0 to be an hourglass that's just been turned upside down.

```
SEROUT 14, 84, [248,
                %00000, →
                %11111, →
                %11111, →
                %01110, →
                %00100, →
                %01010, →
                %10001, →
                %11111] →
```

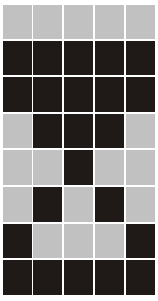


Figure 1-15
Redefining Custom Character 0

Notice how each successive value in the **SEROUT** command corresponds to a row of pixels in the custom character. Notice also how the 1s correspond to black pixels, and the 0s correspond to white.



The SEROUT custom character definitions are not permanent. Each time the LCD's power is turned on and off the custom characters are erased. Since the BASIC Stamp and LCD share the same power supply, the BASIC Stamp program also restarts when the power is reset. It's a good practice to define custom characters you plan to use at the beginning of a program, so that the BASIC Stamp can define the custom characters every time its power is connected.

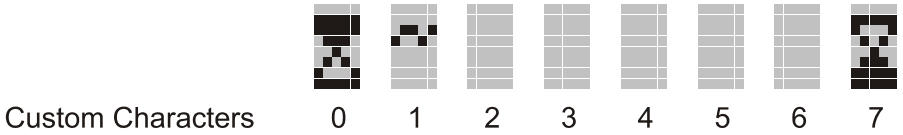
Here is another custom character definition of an hourglass with its four pixels of sand drained into its bottom chamber. This definition uses **255** to tell the LCD to make it

Custom Character 7. It also uses a technique for drawing the characters with asterisks in the comments to the right of the **SEROUT** command. Start with a **SEROUT** command with all the binary values set to %00000, and then draw the character with asterisks in the comment to the right. After it looks right, use the asterisks to dictate which zeros should be changed to ones.

```
SEROUT 14, 84, [255,              ' Define Custom Character 7
                %00000,           '
                %11111,           ' * * * * *
                %10001,           ' *       *
                %01010,           '   *   *
                %00100,           '     *
                %01110,           '   * * *
                %11111,           ' * * * * *
                %11111]           ' * * * * *
```

Figure **1-16** shows how the two **SEROUT** commands just discussed will redefine the LCD's custom characters.

Figure 1-16: After Defining Custom Characters 0 and 7



Custom characters are sometimes defined with hexadecimal values. You will even see this in example programs available for download from the Parallax Serial LCD product pages at www.parallax.com. For information on how hexadecimal character definitions work, try the activity in Appendix B: Hexadecimal Character Definitions.

With these new custom character definitions, you can write a loop to make the hourglass toggle between empty and full, indicating that the user should wait. The **DO**...**LOOP** below does this by first placing the cursor on Line 0, character 5 in the LCD. Then it displays Custom Character 0, the hourglass that was just turned upside down. After a brief **PAUSE**, the program sends the backspace command (8) to get the cursor back to character 5. Then, it sends Custom Character 7, the hourglass with the sand drained into the base. By repeating this sequence, it looks as though the hourglass is turned upside-down, drained, turned again, drained again, and so on.

```
DO

  SEROUT 14, 84, [133]          ' Cursor -> Line 0, char
  SEROUT 14, 84, [0]            ' Display Custom Character 0
  PAUSE  1250                   ' Delay for 1.25 seconds
  SEROUT 14, 84, [8]            ' Backspace
  SEROUT 14, 84, [7]            ' Display Custom Character 7
  PAUSE 1500                    ' Delay for 1.50 seconds

LOOP
```

Example Program: Hourglass.bs2

This program defines and displays the hourglass custom characters just discussed.

- √ Enter, save, and run the program.
- √ Verify that it alternately displays the two hourglass characters at the sixth character in the LCD's top row.

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - Hourglass.bs2
' Define and display custom characters.

' {$STAMP BS2}                              ' Target device = BASIC Stamp 2
' {$PBASIC 2.5}                             ' Language      = PBASIC 2.5

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

PAUSE 250                                   ' Debounce power supply

SEROUT 14, 84, [248,                        ' Define Custom Character 0
                %00000,                     '
                %11111,                     ' * * * * *
                %11111,                     ' * * * * *
                %01110,                     '   * * *
                %00100,                     '     *
                %01010,                     '   *   *
                %10001,                     ' *       *
                %11111]                     ' * * * * *

SEROUT 14, 84, [255,                        ' Define Custom Character 7
                %00000,                     '
                %11111,                     ' * * * * *
                %10001,                     ' *       *
                %01010,                     '   *   *
                %00100,                     '     *
                %01110,                     '   * * *
                %11111,                     ' * * * * *
                %11111]                     ' * * * * *
```

```
SEROUT 14, 84, [22, 12]                       ' Turn on display and clear
PAUSE 5                                       ' 5 ms delay for clearing display

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

DO

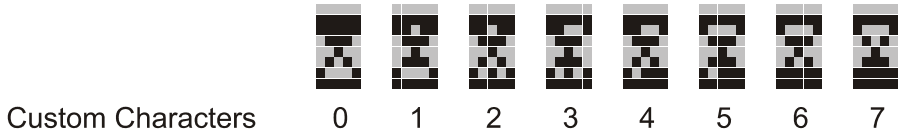
  SEROUT 14, 84, [133]                        ' Cursor -> Line 0, char
  SEROUT 14, 84, [0]                          ' Display Custom Character 0
  PAUSE  1250                                 ' Delay for 1.25 seconds
  SEROUT 14, 84, [8]                          ' Backspace
  SEROUT 14, 84, [7]                          ' Display Custom Character 7
  PAUSE 1500                                  ' Delay for 1.50 seconds

LOOP
```

Your Turn

Figure 1-17 shows the custom characters depicting the grains of sand in the hourglass moving from the top to the bottom.

Figure 1-17: Custom Characters for Animated Hourglass



- √ Save Hourglass.bs2 as HourGlassYourTurn.bs2.
- √ Expand the Initialization routine so that it defines all eight custom characters as shown in Figure 1-17.
- √ Modify the Main Routine so that it gives an animated hourglass effect as the grains of sand fall from top to bottom.

Here is a Main Routine you can also try for animating the eight custom characters once you have updated the Initialization section:

```
DO

  ' Place cursor at character 5, and display Custom Character 0.
  SEROUT 14, 84, 100, [133, 0]
  PAUSE 750                                   ' 0.750 second delay
```

```
  ' Backspace, Custom Character 1, backspace, Custom Character 2, etc.
  ' optional Pace argument of 100 sends each value every 1/10 of a second.
  SEROUT 14, 84, 100, [8, 1, 8, 2, 8, 3, 8, 4, 8, 5, 8, 6, 8, 7, 8]
  PAUSE 750

LOOP
```

√ Try it!



Even though the LCD only stores 8 custom characters at a time, your program can store as many as you need. Remember, your program can redefine any of the custom characters at any time. If your application needs twenty custom characters, your PBASIC program can store 20 custom characters and redefine them for the LCD as needed.

You can display the hourglass with just one custom character. The entire hourglass animation can be done with just one custom character. The trick is to redefine the custom character between each time the display is updated.

ACTIVITY #5: SCROLLING TEXT ACROSS THE DISPLAY

If your message is too wide for the 16 character display, scrolling the text across the display can make it work. Figure 1-18 shows an example. With scrolling, the message begins at the far right of the display. Then, the text shifts across the display one letter at a time.

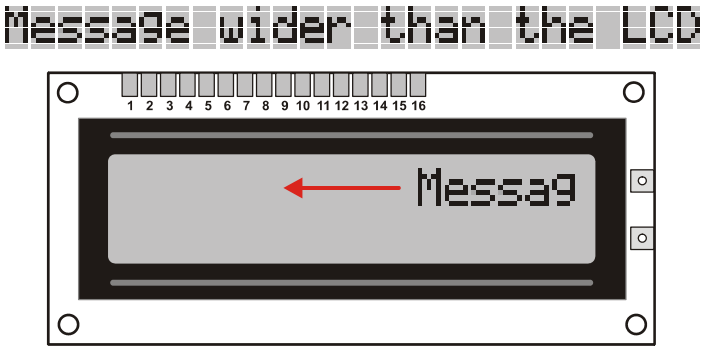


Figure 1-18
Scrolling Text

The scrolling code introduced in this activity is quite different from the example program in Activity #2 that made "Line 1" move across the display. The main reason it's different is because the message in Activity #2 stopped at the leftmost character. When the

message is larger than the display window, stopping at the left edge of the display will keep the rest of the message from becoming visible.

To get text to scroll across just one line, the program has to start with the first character in a message and display it at the rightmost character. After a short delay, the program has to move the cursor to the second character from the right, and print both the first and second characters. It has to continue this process until the cursor gets to the left of the display. Then, the cursor has to be repeatedly repositioned to that same location as sixteen-character portions of the message are displayed, making the message appear to shift from right to left, one character at a time.

The programming technique for this process is called sliding-window. Aside from being useful for the Parallax LCD, sliding window is what you see when you scroll text up and down in programs like the BASIC Stamp Editor and your web browser. It's also used in programs for transmitting and collecting TCP/IP packets. So every time you open your web browser, there's more than one instance of sliding-window code at work in the background.

A Configurable Scrolling Subroutine

This next example program features a subroutine that is convenient for displaying a variety of scrolling messages with minimal work. All it involves is putting the messages in **DATA** directives preceded with *Symbol* names, setting a few variables, and then calling the scrolling subroutine.

Here are some example **DATA** directives.

```
Message1   DATA @ 2, "Message "
Message2   DATA      "again"
Message3   DATA      "Larger message, going faster"
Message4   DATA
```

The first text message begins at an EEPROM address equal to the value of the **Message1** symbol, which has been set to 2 with the **DATA** directive's optional ***@Address*** argument. The address after the end of **Message1** is EEPROM address 11. This is denoted by the **Message2** label, which is also the beginning of the second message. Since you can set variables equal to the values of **Message1** to **Message4**, it's an especially flexible system for a variety of messages.

The next example program also has variables you can set to configure different window locations, widths, and increments. After setting these variable values, you can then call the **Scroll_Message** subroutine, and it does the rest of the work. Here is an example of a code block that makes the subroutine display all the characters between the **Message1** and **Message2** labels in the middle four characters on the LCD's top line.

```
messageStart = Message1:  messageEnd  = message2
windowLeft   = 134:       windowRight = 137
increment    = 1
GOSUB Scroll_Message
```

The beginning and ending EEPROM addresses are stored in the **messageStart** and **messageEnd** variables. The starting and ending LCD character addresses that define the window are stored in **windowLeft** and **windowRight**. Last but not least, the **increment** variable is set to the number of characters the text moves each time it shifts. With all those values set, the **Scroll_Message** subroutine has all it needs to do its job.

There are three more examples in the next program's Main Routine. Not all the examples assign values to all the variables. Some of the examples only set a few values because they are recycling values that were assigned before the previous subroutine call. For example, the value of the **increment** variable was set to 1 before the first subroutine call. Since the **Scroll_Message** subroutine doesn't make any changes to that variable, the value 1 doesn't need to be reassigned to it before calling the **Scroll_Message** subroutine again.

```
' Change the values of various configuration variables
' and demonstrate the effect on the display with each change.
windowLeft   = 131:       windowRight = 140
GOSUB Scroll_Message
```

Here is the last example in the Main Routine. Note that it takes up the better part of the second line and scrolls two characters at a time:

```
messageStart = Message3:  messageEnd  = message4
windowLeft   = 150:       windowRight = 161
increment    = 2
GOSUB Scroll_Message
```

Example Program - TestScrollingSubroutine.bs2

- √ Review the code blocks in the Main Routine of the program and predict how wide the scrolling window will be, what text will be displayed, and how many characters at a time the message will shift.
- √ Enter, save, and run TestScrollingSubroutine.bs2.
- √ Compare your predictions to what actually occurred and reconcile any differences.

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - TestScrollingSubroutine.bs2
' Scroll a text message across a four character wide window in the LCD.

' {$STAMP BS2}                              ' BASIC Stamp Directive
' {$PBASIC 2.5}                             ' PBASIC Directive

' -----[ DATA Directives ]-------------------------------------------------

Message1   DATA @ 2, "Message "
Message2   DATA      "again"
Message3   DATA      "Larger message, going faster..."
Message4   DATA
' -----[ I/O Definitions ]-------------------------------------------------

LcdPin         PIN     14                  ' LCD I/O pin

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

T9600          CON     84                  ' True, 8-bits, no parity, 9600
LcdCls         CON     12                  ' Form feed -> clear screen
LcdCr          CON     13                  ' Carriage return
LcdOff         CON     21                  ' Turns display off
LcdOn          CON     22                  ' Turns display on
Line0          CON     128                 ' Line 0, character 0
Line1          CON     148                 ' Line 1, character 0

TimeOn         CON     250                 ' Character on time
TimeOff        CON     0                   ' Character fade time

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

' Functional variables for Scroll_Message subroutine.

cursorStart    VAR     Byte                ' First character location
head           VAR     Byte                ' Start of displayed text
tail           VAR     Byte                ' End of displayed text
pointer        VAR     Byte                ' EEPROM address pointer
character      VAR     Byte                ' Stores a character
```

```
' Configuration variables for Scroll_Message subroutine.

increment      VAR     Nib                  ' Characters to shift
windowRight    VAR     Byte                 ' Rightmost character address
windowLeft     VAR     Byte                 ' Leftmost character address
messageStart   VAR     Byte                 ' Start EEPROM address
messageEnd     VAR     Byte                 ' End EEPROM address

' -----[ Initialization ]---------------------------------------------------
SEROUT LcdPin, T9600, [LcdOn, LcdCls]      ' Turn on display & clear
PAUSE 5                                    ' Delay 5 ms

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

' Set values of configuration variables, then call Scroll_Message.

messageStart = Message1:  messageEnd  = message2
windowLeft   = 134:       windowRight = 137
increment    = 1
GOSUB Scroll_Message

' Change the values of various configuration variables and demonstrate the
' effect on the display with each change.

windowLeft   = 131:       windowRight = 140
GOSUB Scroll_Message

messageStart = Message1:  messageEnd  = message3
GOSUB Scroll_Message

messageStart = Message3:  messageEnd  = message4
windowLeft   = 150:       windowRight = 161
increment    = 2
GOSUB Scroll_Message

END

' -----[ Subroutine - Scroll_Message ]--------------------------------------

Scroll_Message:

  cursorStart = windowRight - increment + 1    ' Rightmost character in window
  head = 0                                     ' Initialize head and tail
  tail = increment - 1                         ' of message

  ' Scrolling loop
  DO WHILE tail<(MessageEnd-MessageStart)+(windowRight-windowLeft+increment)

    SEROUT LcdPin, T9600, [cursorStart]      ' Rightmost character in window

    FOR pointer = head TO tail               ' Clear old characters.
```

```
    SEROUT LcdPin, T9600, [" "]
  NEXT

  PAUSE timeOff                             ' Let characters fade away

  SEROUT LcdPin, T9600, [cursorStart]      ' Rightmost character in window

  ' This FOR...NEXT loop refreshes the message, shifted increment characters
  ' to the left each time through until the end of the EEPROM message.
  ' Then, it fills the display with space characters as the remainder of the
  '  message shifts out to the window.
  FOR pointer = head TO tail
    IF (pointer <= (MessageEnd - MessageStart - 1)) THEN
      READ pointer + MessageStart, character
    ELSE
      character = " "
    ENDIF
    SEROUT LcdPin, T9600, [character]
  NEXT

  PAUSE timeOn                               ' Give the characters some time

  ' Increment until at window-left
  cursorStart = cursorStart - increment MIN windowLeft
  tail = tail + increment                    ' Increment tail pointer

  ' Increment head pointer if tail pointer > window width.
  IF tail > (windowRight - windowLeft) THEN
     head = head + increment
  ELSE
    head = 0
  ENDIF
LOOP                                        ' Repeat scrolling loop

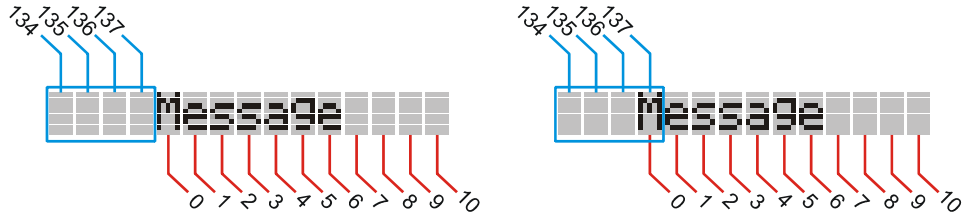
RETURN
```

The Scroll_Message Subroutine's Sliding Window

Let's say the shifting text display window in your LCD is four characters wide on the top row because there are other messages that need to be displayed at all times on the LCD. The task at hand is to slide the text through the smaller window without overprinting any of the characters displayed outside it.

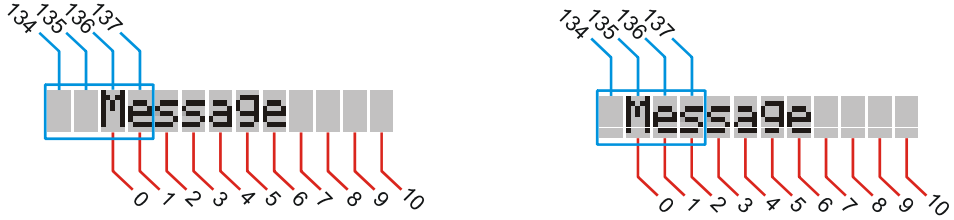
Figure 1-19 shows the setup and Step 0 of a four-character wide window. In the setup step, nothing is displayed in the window. Then, Step 0 places the cursor in position 137, and displays character 0, the "M".

Figure 1-19: Shifting Text Through Window, Setup and Step 0



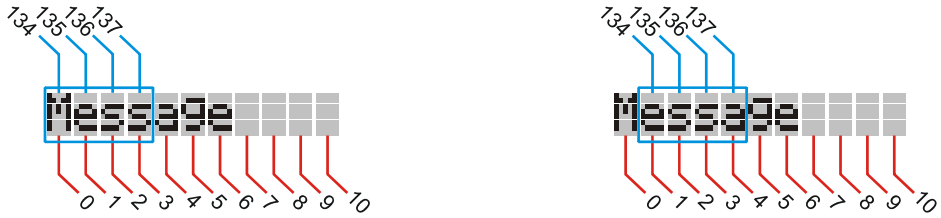
√ Figure 1-20 shows Steps 1 and 2. After waiting a moment for the "M" to be visible, the cursor has to be placed at position 136, and then characters 0 and 1, "Me", can be displayed. Next, move the cursor to 135, and display characters 0 to 2, "Mes".

Figure 1-20: Shifting Text Through Window, Steps 1 and 2



√ Figure 1-21 shows Steps 3 and 4. Moving the cursor to position 134 and displaying characters 0 to 3, "Mess" is still the same sequence, but when the "M" leaves the window, the sequence has to change. The cursor's starting point, or head pointer, can no longer advance to the left; it has to stay at position 134. Also, instead of displaying characters 0 to 3, characters 1 to 4, "essa", have to be displayed.

Figure 1-21: Shifting Text Through Window, Steps 3 and 4



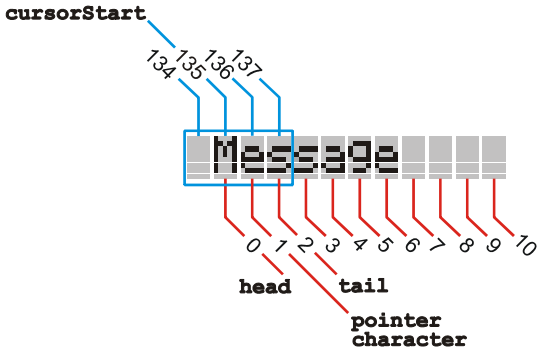
√ The starting cursor position has to remain at 134 while the head and tail characters continue to advance: 2 to 5 - "ssag", 3 to 6 - "sage". The window keeps sliding, and Figure 1-22 shows the second-to-last step of characters 6 to 9 - "e" followed by three spaces, and finally the last step, 7 to 10 - four space characters.

Figure 1-22: Shifting Text Through Window, Steps 9 and 10



The TestScrollingSubroutine.bs2 uses the variables shown in Figure 1-23 for the sliding window. The **cursorStart** variable stores the position that the cursor is placed each time before it starts printing the characters in the message. In the figure, **cursorStart** stores the value 135. The next time the text shifts to the left, it will store 134. Two variables, **head** and **tail**, store the beginning and ending addresses of the text that will fit in the message window. In the figure, **head** stores 0, and **tail** stores 2. The **pointer** variable will be used by the **READ** command to get the right character, and the **character** variable will store the character the **READ** command retrieves from EEPROM.

Figure 1-23: Variables from TestScrollingRoutine.bs2.



In Figure 1-23, **pointer** is pointing at character 1 in the sequence, which is "e". A **FOR**...**NEXT** loop uses the **pointer** variable to read each of the characters in EEPROM, from **head** to **tail** and then display each with the **SEROUT** command. Each time the text shifts to the right, the new text has to overwrite the old text with the same **head** to **tail** loop.

SUMMARY

The liquid crystal display (LCD) is used in a tremendous variety of products. Simple character displays like the Parallax 2X16 serial LCD can substitute for the Debug Terminal's display features, which is especially useful when the test site for your project is not within reach of a serial cable and PC.

The Parallax Serial LCD has a contrast adjustment potentiometer on the back, along with two switches you can use to select from three different baud rates and a self-test mode. There are three pins on the back of the Parallax Serial LCD, as only three connections are needed to operate it: Vdd, RX, and Vss.

The Parallax Serial LCD has an extensive command set, and a full list of these commands is included in the Parallax Serial LCD Product Documentation (Appendix B). This chapter introduced commands for turning the display on and off, clearing it, cursor placement, backlighting control for the backlit model, and character display.

The Parallax Serial LCD depends on serial messages from the BASIC Stamp that are programmed into it by the PBASIC **`SEROUT`** command. Many of the **`DEBUG`** command's features can be used with the **`SEROUT`** command, including text between quotation marks and formatters like **`DEC`**, **`BIN`**, **`DIG`**, and so on. These all have LCD results that are similar to the Debug Terminal. The LCD's control codes are different from and more numerous than the ones used with the Debug Terminal. Instead of trying to use **`CR`**, **`CLS`**, **`CRSRXY`**, etc, use the control code values listed in the LCD's command set. It's also a good idea to make constants for these values, such as **`LcdCls CON 12`**, **`LcdClr CON 13`**, **`LcdOn CON 22`**, **`LcdOff CON 21`**, and so on.

The Parallax Serial LCD has eight custom characters, 0 to 7. You can display any one of them by sending its value to the LCD. For example, **`SEROUT 14, 84, [3]`** makes the LCD print Custom Character 3. The commands to define custom characters range from 248 to 255. Sending **`248`** instructs the LCD to define Custom Character 0, **`249`** defines Custom Character 1, and so on, up to **`255`**, which defines Custom Character 7. After sending a Define Custom Character command, the next eight bytes are binary values, the lower five bits of which define the pixels in a given line of pixels. A 1 makes the pixel black, and a 0 makes it white.

This chapter also introduced a subroutine for scrolling text from right to left inside a window. This subroutine looks for start and stop addresses that correspond to *Symbol* address labels that precede the **DATA** directives that contain the text to be displayed. The way the subroutine's text is displayed is defined by five variables: **messageStart**, **messageEnd**, **windowLeft**, **windowRight**, and **increment**. The **messageStart** and **messageEnd** variables store the starting and ending EEPROM addresses of the text to be displayed. The **windowLeft** and **windowRight** variables store the start and end LCD character addresses that define the window, and the increment variable stores how many characters at a time the message is shifted from right to left.

Questions

1. What are three devices you use every day that display information with LCDs?
2. What do the 2 and 16 indicate in the term 2x16 LCD?
3. What command do you use to send information to the Parallax Serial LCD?
4. How are the **DEBUG** and **SEROUT** commands different?
5. What position do SW1 and SW2 need to be in if you want to write a program that sends messages to the Parallax Serial LCD at a rate of 19,200 bps?
6. What component do you adjust to change the LCD's display contrast?
7. What **SEROUT** command will clear the display?
8. What special considerations come into play when using the **DEBUG** command's **CR**, **CLS**, and **HOME** control codes with the Parallax Serial LCD?
9. What three arguments do you need for a minimal **SEROUT** command?
10. How can you make the text displayed in the LCD flash on and off?
11. What ranges of values can you send to the LCD to place the cursor?
12. What character resides in Custom Character 1 by default?
13. How do you display a custom character after it has been defined?
14. What are some applications of sliding window?

Exercises

1. Make the message "Hello" appear in the Debug Terminal without using the **DEBUG** command.
2. Display the message "Hello" centered on the top line of the LCD.
3. Make the message "Hello" flash on and off once every second.
4. Write a command to make the message "Start" appear at the beginning of Line 0 and the message "Finish" appear on the right side of Line 1.

5. Write a **SEROUT** command to send the LCD messages when SW1 and SW2 are both ON.
6. Write a **SEROUT** command to send the LCD a message when SW1 is ON and SW2 is OFF.

Projects

1. Write a program that displays a six-line message. It starts by displaying lines 0 and 1 with a pause. Then it advances to lines 2 and 3, again with a pause. And finally, it displays lines 4 and 5.
2. Write a program that prints three copies of a custom character. Then, redefine the custom character. What happens to all three copies of the custom character?

Solutions

Q1. Wrist watch, calculator, telephone (answers will vary).
Q2. Two rows of text, each row is 16 characters wide.
Q3. The **SEROUT** command.
Q4. When using the **SEROUT** command you must specify the pin number and the baud rate.
Q5. Both SW1 and SW2 should be in the ON position for 19,200 bps.
Q6. A potentiometer.
Q7. The command **SEROUT 14, 84, [12]** will clear the display.
Q8. The pre-defined PBASIC constants such as **CR**, **CLS**, and **HOME** are not necessarily defined correctly to work with the serial LCD.
Q9. **SEROUT** requires ***Pin***, ***Baudmode***, and ***DataItem*** arguments.
Q10. Print the text, then turn the display on and off using control characters 21 and 22.
Q11. From 128-143 for Line 0, and 148-163 for Line 1.
Q12. The backslash.
Q13. Send the LCD the value of the custom character with the **SEROUT** command. For example, **SEROUT 14, 84, [4]** will display Custom Character 4.
Q14. LCD screens, even those large screens you see at train stations, gates in airports, or at sporting events, as well as scrolling text in Windows applications and TCP/IP packets.
E1. From the BASIC Stamp Editor Help file: "For the built-in serial port set the Tpin argument to 16 in the **SEROUT** command."

```
' Smart Sensors and Applications - Ch1_Ex01.bs2
' {$STAMP BS2}
' {$PBASIC 2.5}

DEBUG "Hello, this is DEBUG", CR
SEROUT 16, 84, ["Hello - This is SEROUT", CR]
```

E2. Example solution:

```
' Smart Sensors and Applications - Ch1_Ex02.bs2
' {$STAMP BS2}
' {$PBASIC 2.5}

SEROUT 14, 84, [22, 12]                     ' Turn on, clear screen
'               1234567890123456
SEROUT 14, 84, ["     Hello      ", CR]      ' Center text on top line
```

E3. Example solution:

```
' Smart Sensors and Applications - Ch1_Ex03.bs2
' Make message flash on and off once per second
' {$STAMP BS2}
' {$PBASIC 2.5}

SEROUT 14, 84, [22, 12]                      ' Turn on, clear screen
'                1234567890123456
SEROUT 14, 84, ["     Hello      ", CR]      ' Center text on top line

DO
  SEROUT 14, 84, [21]                        ' Turn screen off
  PAUSE 500
  SEROUT 14, 84, [22]                        ' Turn screen on
  PAUSE 500
LOOP
```

E4. Example solution:

```
' Smart Sensors and Applications - Ch1_Ex04.bs2
' Print Start beginning of Line1, Finish end of Line2
' {$STAMP BS2}
' {$PBASIC 2.5}

SEROUT 14, 84, [22, 12]                   ' Turn on, clear screen
SEROUT 14, 84, ["Start"]                  ' Print on Line 0
SEROUT 14, 84, [158]                      ' Line2,6th char from rt edge
SEROUT 14, 84, ["Finish"]                 ' Print rt edge of Line 1
```

E5. Example solution:

```
' Smart Sensors and Applications - Ch1_Ex05.bs2
' Print at 19200 baud
' {$STAMP BS2}
' {$PBASIC 2.5}

SEROUT 14, 32, [22, 12]                      ' Turn on, clear screen
SEROUT 14, 32, ["Using 19200 bps"]           ' Print on Line 0
```

E6. Example solution:

```
' Smart Sensors and Applications - Ch1_Ex06.bs2
' Print at 2400 baud
' {$STAMP BS2}
' {$PBASIC 2.5}

SEROUT 14, 396, [22, 12]                      ' Turn on, clear screen
SEROUT 14, 396, ["Using 2400 bps"]            ' Print on Line 0
```

P1. Example solution:

```
' Smart Sensors and Applications - Ch1_Project1.bs2
' Display a 6-line message
' {$STAMP BS2}
' {$PBASIC 2.5}

LcdPin         PIN     14
T9600          CON     84

PAUSE 250
SEROUT 14, 84, [22, 12]               ' Turn on display and clear
PAUSE 5                               ' 5 ms delay for clearing display

SEROUT LcdPin, T9600, ["I have never let"]
SEROUT LcdPin, T9600, ["my schooling    "]
PAUSE 1500
SEROUT LcdPin, T9600, ["interfere with  "]
SEROUT LcdPin, T9600, ["my education.   "]
PAUSE 1500
SEROUT LcdPin, T9600, ["     -Mark Twain"]
SEROUT LcdPin, T9600, ["      1835-1910 "]

END
```

P2. All three copies will change into the newly defined character! It's like magic. A sample program is shown below.

```
' Smart Sensors and Applications - Ch1_Project2.bs2
' Print 3 copies of custom character, then redefine character.
' {$STAMP BS2}                          ' Target device = BASIC Stamp 2
' {$PBASIC 2.5}                         ' Language      = PBASIC 2.5

Line0          CON     128
Line1          CON     148
copies         VAR     Nib

PAUSE 250
SEROUT 14, 84, [22, 12]                 ' Turn on display and clear
PAUSE 5                                 ' 5 ms delay for clearing
display


SEROUT 14, 84, [248,                    ' Define Custom Character 0
                %00110,                 '     * *
                %00101,                 '     *   *
                %00100,                 '     *
                %11111,                 ' * * * * *
                %00100,                 '     *
                %01110,                 '   * * *
                %10101,                 ' *   *   *
                %00100]                 '     *
```

```
FOR copies = 1 TO 3
  SEROUT 14, 84, [0]                      ' Display Custom Character 0
NEXT

PAUSE 1000                                ' Allow time to view
SEROUT 14, 84, [Line1, "now re-defining"]' Display message on Line 1
PAUSE 1000
SEROUT 14, 84, [Line1, "               "]' Clear message

SEROUT 14, 84, [248,                      ' Re-define Custom Character 0
                %00100,                   '     *
                %10011,                   ' *     *  *
                %01001,                   '   *       *
                %00101,                   '     *     *
                %00001,                   '           *
                %00010,                   '       *
                %00100,                   '     *
                %11000]                   ' * *
END
```

Chapter 2: The Ping))) Ultrasonic Distance Sensor

2

The Ping))) sensor interfaced with a BASIC Stamp can measure how far away objects are. With a range of 3 centimeters to 3.3 meters, it's a shoo-in for any number of robotics and automation projects. It's also remarkably accurate, easily detecting an object's distance down to the centimeter.

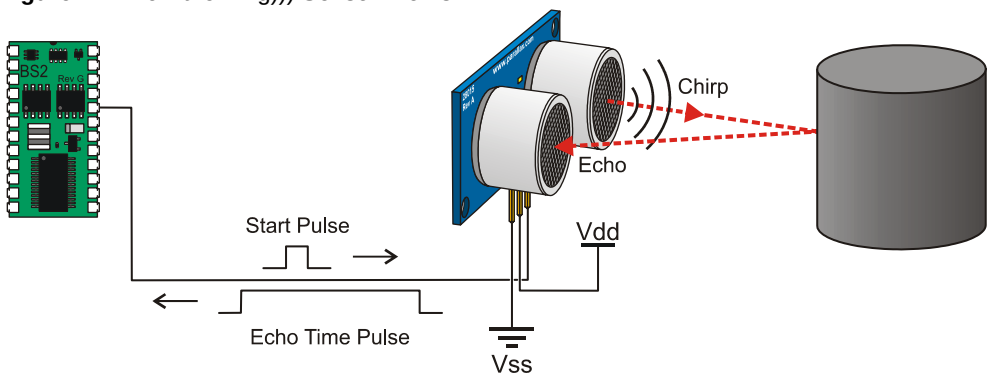


Figure 2-1
The Ping)))™ Ultrasonic Distance Sensor

HOW DOES THE PING))) SENSOR WORK?

Figure 2-2 shows how the Ping))) sensor sends a brief chirp with its ultrasonic speaker and measures the echo's return time to its ultrasonic microphone. The BASIC Stamp starts by sending the Ping))) sensor a pulse to start the measurement. Then, the Ping))) sensor waits long enough for the BASIC Stamp program to start a **`PULSIN`** command. Then, at the same time the Ping))) sensor chirps its 40 kHz tone, it sends a high signal to the BASIC Stamp. When the Ping))) sensor detects the echo with its ultrasonic microphone, it changes that high signal back to low.

Figure 2-2: How the Ping))) Sensor Works



The BASIC Stamp **PULSIN** command uses a variable to store how long the high signal from the Ping))) sensor lasted. This time measurement is how long it took sound to travel to the object and back. Using this measurement and the speed of sound in air, you can make your program calculate the object's distance in centimeters, inches, feet, etc.



The Ping))) sensor's chirps are not audible because 40 kHz is ultrasonic.

What we consider sound is our inner ear's ability to detect the variations in air pressure caused by vibration. The rate of these variations determines the pitch of the tone. Higher frequency tones result in higher pitch sounds and lower frequency tones result in lower pitch tones.

Most people can hear tones that range from 20 Hz, which is very low pitch, to 20 kHz, which is very high pitch. Subsonic is sound with frequencies below 20 Hz, and ultrasonic is sound with frequencies above 20 kHz. Since the Ping))) sensor's chirps are at 40 kHz, they are definitely ultrasonic, and not audible to people.

ACTIVITY #1: MEASURING ECHO TIME

In this activity, you will test the Ping))) sensor and verify that it gives you echo time measurements that correspond to an object's distance.

Parts Required

(1) Ping))) Ultrasonic Distance Sensor
(3) Jumper Wires

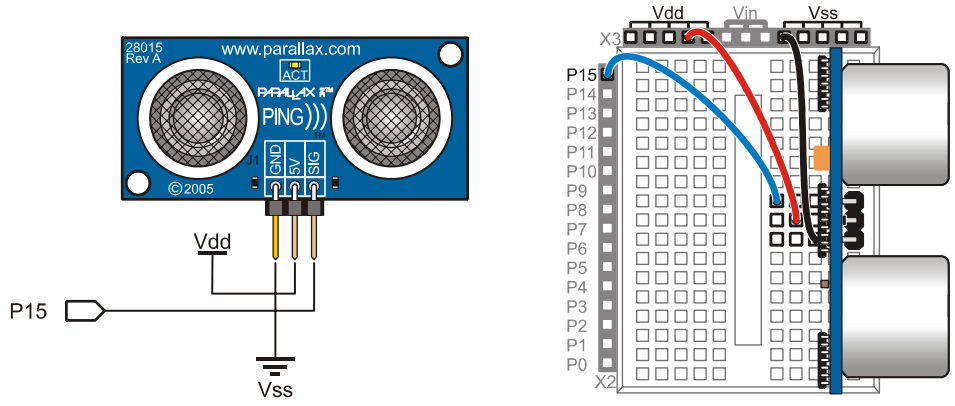
All you'll need is a Ping))) sensor and three jumper wires to make it work. The Ping))) sensor has protection against programming mistakes (and wiring mistakes) built-in, so there's no need to use a 220 Ω resistor between P15 and the Ping))) sensor's SIG terminal.

Ping))) Sensor Circuit

Figure 2-3 shows a schematic and wiring diagram for testing the Ping))) sensor.

√ Build the circuit.

Figure 2-3: Ping))) Sensor Schematic and Wiring Diagram



Testing the Ping))) Sensor

As mentioned earlier, the Ping))) sensor needs a start pulse from the BASIC Stamp to start its measurement. A pulse to P15 that lasts 10 µs (**`PULSOUT 15, 5`**) is easily detected by the Ping))) sensor, and it only takes a small amount of time for the BASIC Stamp to send. A **`PULSIN`** command that stores the duration of the Ping))) sensor's echo pulse (**`PULSIN 15, 1, time`**) has to come immediately after the **`PULSOUT`** command. In this example, the result the **`PULSIN`** command stores in the variable **`time`** is the round trip time for the Ping))) sensor's chirp to get to the object, reflect, and return.

Example Program - PingTest.bs2

You can test this next program by measuring the distances of a few close-up objects. For close-up measurements, the Ping))) sensor only needs to be 3 to 4 inches (roughly 8 to 10 cm) above your working surface. However, if you are measuring objects that are more than a half a meter away, you may need to elevate your Ping))) sensor to prevent echoes from the floor registering as detected objects.

- √ Place your Board of Education with the Ping))) sensor circuit on something to keep it at least 8 cm above the table surface.
- √ Place an object (like a water bottle, box, or paper target) 15 cm from the front of the Ping))) sensor.
- √ Enter, save, and run PingTest.bs2.
- √ The Debug Terminal should start reporting a value in the 400 to 500 range.

- √ Move the target to a distance of 30 cm from the Ping))) sensor and verify that the value of the time variable roughly doubled.
- √ Point your Ping))) sensor at a variety of near and far objects, and observe the time measurements.

```
' Smart Sensors and Applications - PingTest.bs2
' Tests the Ping))) ultrasonic distance sensor

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

time VAR Word

DO
  PULSOUT 15, 5
  PULSIN 15, 1, time

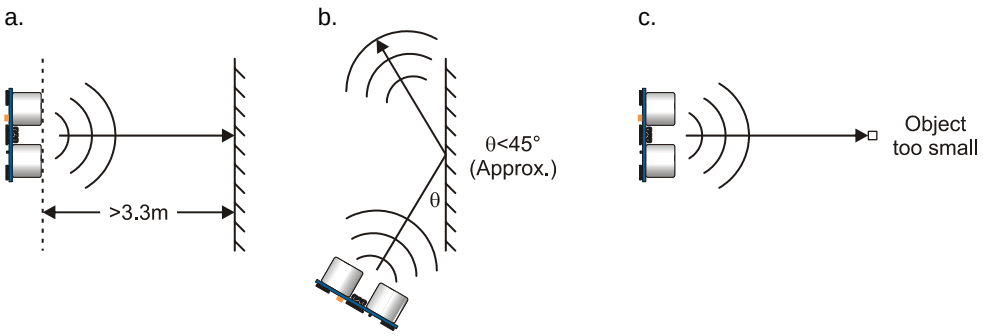
  DEBUG HOME, "time = ", DEC5 time

  PAUSE 100
LOOP
```

Your Turn - Testing Range, Angle and Object Size

In terms of accuracy and overall usefulness, ultrasonic distance detection is really great, especially compared to other low-cost distance detection systems. That doesn't mean that the Ping))) sensor is capable of measuring "everything". Figure 2-4 shows a few situations that the Ping))) is not designed to measure: (a) distances over 3 meters, (b) shallow angles, and (c) objects that are too small.

Figure 2-4: The Ping))) Sensor is Not Designed for these Situations:



In addition, as Ken Gracey of Parallax Inc. discovered during a classroom demonstration at his son's school, some objects with soft, irregular surfaces (such as stuffed animals) will absorb rather than reflect sound and therefore can be difficult for the Ping))) sensor to detect. Objects with smooth surfaces that readily reflect sound are easier for the sensor to detect.

- √ Try pointing the Ping))) sensor at various objects that are different distances away. What's the largest value the Ping))) sensor returns? How close do you have to get to the object before the time measurements start to decrease?
- √ Try standing one meter away from the wall, and point the Ping))) sensor at it, and record the measurement. Next, try pointing the Ping))) sensor at the wall at different angles, as shown in Figure 2-5. Do the values change? At what angle does the Ping))) sensor cease to detect the wall?

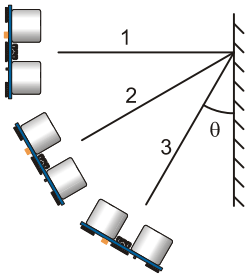


Figure 2-5
Determining the Minimum Angle of Detection

- √ Try hanging various objects from the ceiling at about 1.5 meters from the Ping))) sensor. How small can the object be? Does shape or angle matter? Does the size requirement change at 0.5 meters?
- √ Try detecting objects of similar size but made from different materials, such as a cardboard shoebox and a fuzzy slipper, to see if you have a smaller effective range with sound-absorbing objects. Can you find any objects invisible to the Ping))) sensor? How about a wad of cotton balls, or of tulle netting?

ACTIVITY #2: CENTIMETER MEASUREMENTS

This activity demonstrates how to use the speed of sound and the PBASIC Multiply High operator (`**`) to calculate the distance of an object based on the echo time measurement from the Ping))) sensor.

Calculating Centimeter Distance with PBASIC

The equation for the distance sound travels is $S = C_{air}\ t$, where S is distance, C_{air} is the speed of sound in air, and t is time. Since the Ping))) sensor's time measurement is the time it takes sound to get to the object and bounce back, the actual distance, S_{object}, is half of the total distance the sounds travels.

$$S = C_{air}\,t$$

$$S_{object} = \frac{S}{2} = \frac{C_{air}\,t}{2}$$

The speed of sound in air is most commonly documented in terms of meters per second (m/s). However, centimeter (cm) measurements will be more convenient to calculate with the BASIC Stamp. Since there are 100 centimeters in a meter, let's use $S_{object\text{-}cm}$ which is simply 100 times S_{object}. The **`PULSIN`** ***`Duration`*** measurement units for the BASIC Stamp 2 are 2/1,000,000 of a second (2 µs). So, instead of t, which has to be a measurement of seconds, we'll use $t_{PULSIN\text{-}BS2}$. When multiplied by 2/1,000,000 $t_{PULSIN\text{-}BS2}$ gives us the number of seconds. There is a pair of 2s in the numerator and denominator that cancel, and 100 in the numerator can cancel with two of the zeros in the denominator's 1,000,000. The result of these substitutions and cancellations is $S_{object\text{-}cm} = (C_{air}\ t_{PULSIN\text{-}BS2})/10{,}000$.

$$S_{object\text{-}cm} = \frac{100\,C_{air}\,t}{2}$$

$$S_{object\text{-}cm} = \frac{100\,C_{air}\,t_{PULSIN\text{-}BS2}}{2} \times \frac{2}{1{,}000{,}000}$$

$$S_{object\text{-}cm} = \frac{C_{air}\,t_{PULSIN\text{-}BS2}}{10{,}000}$$

The speed of sound in air at room temperature 72 °F (22.2 °C) is 344.8 m/s. Dividing 10,000 into this leaves us with $S_{object\text{-}cm} = 0.03448\ t_{PULSIN\text{-}BS2}$.

$$S_{object\text{-}cm} = \frac{344.8\ t_{PULSIN\text{-}BS2}}{10{,}000}$$

$$= 0.03448\ t_{PULSIN\text{-}BS2}$$

The BASIC Stamp can use the ****** operator to multiply a variable that stores the **PULSIN** command's ***Duration*** measurement by a fractional value that's less than 1. For example, if the **PULSIN** command stores the echo measurement in the **time** variable, this command will store the centimeter distance result in the **cmDistance** variable:

```
cmDistance = CmConstant ** time
```

With the ****** operator, **CmConstant** will have to be 2260, which is the ****** equivalent of 0.03448. Instead of a decimal denominator, like 10,000 (in the case of 0.03448), the ****** operator needs a value that would go in the numerator of a fraction with a denominator of 65536. To get that numerator, multiply your fractional value by 65536.

$$CmConstant = 0.03448 \times 65536 = 2260$$

Now, we've got the value we need to modify PingTest.bs2 so that it will measure centimeter distance. We'll also add a variable to store distance (**cmDistance**) along with the constant to store the value 2260 (**CmConstant**).

```
CmConstant  CON   2260

cmDistance  VAR   Word
```

Then, the ****** calculation can be added to the PingText.bs2's **DO**...**LOOP** to calculate the centimeter measurement. The **DEBUG** command in the program can then be modified to display the measurement.

```
cmDistance = CmConstant ** time

DEBUG HOME, DEC3 cmDistance, " cm"
```

Example Program: PingMeasureCm.bs2

- √ Enter, save and run PingMeasureCm.bs2.
- √ Move the target object until the measurement displays 20 cm.
- √ Align your ruler with that measurement. The 0 cm mark should align somewhere with the Ping))) sensor, typically somewhere between the printed circuit board and the front-most part of the speaker/microphone.
- √ Now, experiment with other distance measurements.

```
' Smart Sensors and Applications - PingMeasureCm.bs2
' Measure distance with Ping))) sensor and display in centimeters.

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

' Conversion constants for room temperature measurements.
CmConstant  CON   2260

cmDistance  VAR   Word
time        VAR   Word

DO

  PULSOUT 15, 5
  PULSIN 15, 1, time

  cmDistance = CmConstant ** time

  DEBUG HOME, DEC3 cmDistance, " cm"

  PAUSE 100

LOOP
```

Your Turn - Verifying the Calculations

Let's verify that that the program is correctly calculating the distance.

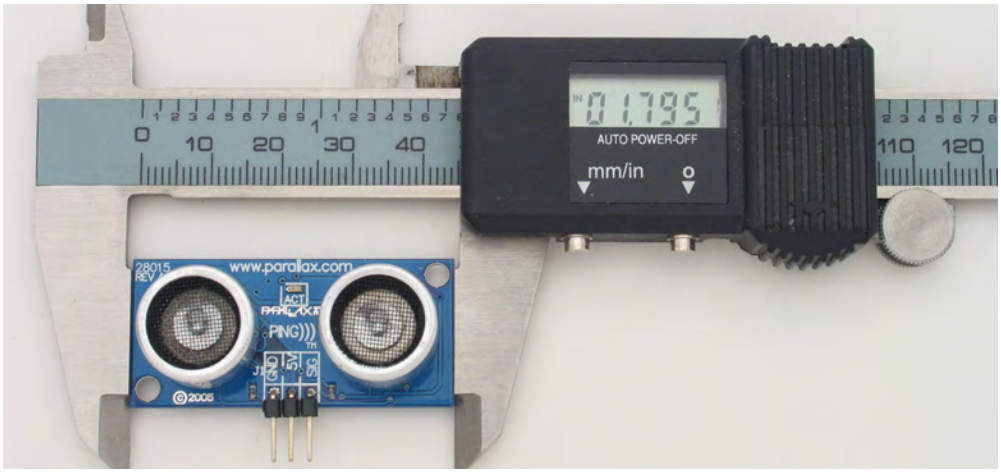
- √ Modify PingMeasureCm.bs2 so that it displays the values of both the time and the distance variables.
- √ Use a calculator to verify that you get the same result from the distance equation that you do from the program.

$$S_{object\text{-}cm} = 0.03448 \times t_{PULSIN\text{-}BS2}$$

ACTIVITY #3: INCH MEASUREMENTS

Most electronic distance measuring devices offer results in either metric or English units. For example, the calipers shown in Figure 2-6 has a button you can press to choose between mm and inches. Other measuring devices offer yards or meters, or inches or centimeters, etc. So that your program can display both centimeters and inches, this activity introduces uses the multiply-high (******) operator a second time to convert from centimeters to inches.

Figure 2-6: Calipers with a mm/in Toggle Button



An Inches ** Constant

The **`CmConstant`** used in **`cmDistance = time ** CmConstant`** is a measure of the speed of sound in centimeters per **`PULSOUT`** time unit. There are 2.54 centimeters in every inch. So, the conversion formula from centimeter to inch distances can be written like this:

$$S_{in} = S_{cm} \div 2.54$$

The easiest way to convert to inches is to simply divide the value of **`CmConstant`** by 2.54, and use the result declared in another constant, like **`InConstant`**. Remember that constants for the **`**`** operator should be integers, so round the result to the nearest integer.

$$\texttt{InConstant} = 2260 \div 2.54 = 889.76 \approx 890$$

Example Program: PingMeasureCmAndIn.bs2

- √ Enter, save, and run PingMeasureCmAndIn.bs2.
- √ Experiment with the distance measurements and verify that they are correct in both systems.

```
' Smart Sensors and Applications - PingMeasureCmAndIn.bs2
' Measure distance with Ping))) sensor and display in both in & cm

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

' Conversion constants for room temperature measurements.
CmConstant  CON   2260
InConstant  CON   890

cmDistance  VAR   Word
inDistance  VAR   Word
time        VAR   Word

DO

  PULSOUT 15, 5
  PULSIN 15, 1, time

  cmDistance = cmConstant ** time
  inDistance = inConstant ** time

  DEBUG HOME, DEC3 cmDistance, " cm"
  DEBUG CR, DEC3 inDistance, " in"

  PAUSE 100

LOOP
```

Your Turn

- √ There are 12 inches in 1 foot. Modify the program so that it displays feet and inches. Hint: After calculating **`inDistance`**, use **`/ 12`** to figure out the number of feet, and **`// 12`** to find the remainder in inches.
- √ There are 10 centimeters in a decimeter. Repeat for decimeters and centimeters.

ACTIVITY #4: MOBILE MEASUREMENTS

This activity demonstrates displaying the Ping))) sensor's centimeter and inch measurements on the Parallax Serial LCD. Provided you're using a battery, you can disconnect from your computer and take the setup to remote locations of your choosing.

Connecting the Ping))) Sensor with an Extension Cable

In order to make room for the Parallax Serial LCD on the Board of Education, we'll connect the Ping))) sensor to the board with an extension cable. You can then hold it and point it various places, or use hardware to mount it next to your Board of Education.

Parts Required

(1) Ping))) Ultrasonic Sensor
(1) Parallax Serial LCD (2×16)
(1) 14-inch LCD Extension Cable
(3) Jumper Wires

If you are working from a BASIC Stamp HomeWork Board or a serial Board of Education Rev A or B, you will also need:

(1) 3-pin header
(3) Additional jumper wires

Ping))) Sensor and LCD Cable Connections

The schematics shown in Figure 2-7 below are identical to the ones that have been used for the Ping))) sensor and the Parallax Serial LCD up to this point. We will now change the way these electrical connections are made by adding a cable, so that both devices can be conveniently connected to your board at the same time. Though the schematics are the same, the actual cable connections will vary depending on which BASIC Stamp educational board you are using.

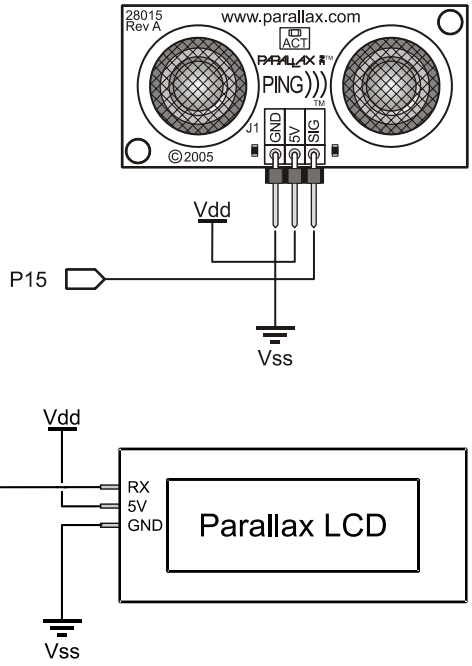


Figure 2-7
Ping))) Sensor and Parallax Serial LCD Schematic

Board of Education Rev C and USB Board of Education Cable Connections

These instructions are for the boards that have servo ports with a Vdd/Vss jumper in between, such as the Board of Education Rev C and USB Board of Education. For all other boards, skip to **All other BASIC Stamp Educational Boards** on page 54.

- √ Disconnect power to your board.
- √ Set the jumper between the X4 and X5 servo to Vdd (+5 V) as shown in Figure 2-8. The jumper should cover the two pins closest to Vdd, and the third pin next to Vin should be visible.

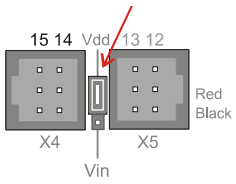


Figure 2-8
The Servo Port Jumper Set to Vdd (+5 V)



Vdd vs. Vin jumper settings determine which power supply is connected to the X4 and X5 ports. When the jumper is set to Vdd, these ports receive regulated 5 V from the Board of Education's voltage regulator. If the jumper is set to Vin, the port receives power directly from the battery or power supply.

- √ Connect the Parallax Serial LCD as shown. It's the same as the previous chapter.
- √ Plug one end of the extension cable into Port 15 of the X4 header, making sure that the "Red" and "Black" labels along the right side of the X5 port line up with the cable's red and black wires.
- √ Verify that your cable is plugged in correctly by checking to make sure the white wire is closest to the 15 label and the black wire is closest to the X4 label.

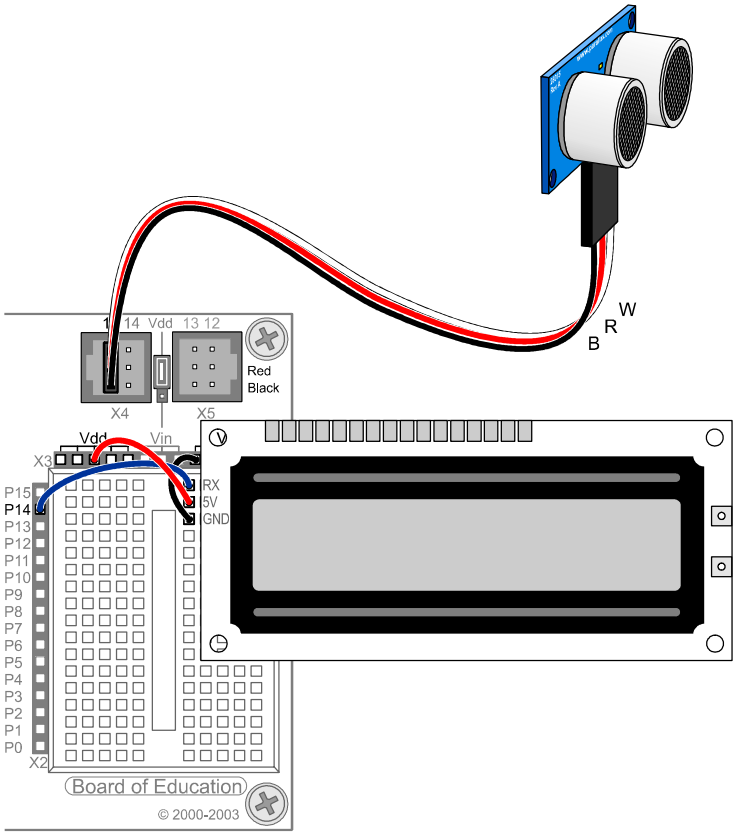


Figure 2-9
Servo Port and Power Jumper Connection for Ping))) Sensor

√ Connect the other end of the cable so that the black wire is connected to the Ping))) module's GND pin, the red wire is connected to the 5 V pin, and the white wire is connected to the RX pin.
√ Double-check all your connections, including your jumper setting, and make sure they are correct.



WARNING! Do not connect power to your board until you are positive the connections are correct. If you make a mistake with the LCD connections, the Parallax Serial LCD could be permanently damaged.

√ Plug the power back into the board.
√ Set the 3-position switch on the Board of Education to 2.
√ If you have a Board of Education Rev C, Skip to **LCD Distance Display** on page 57.



You can also connect the Parallax Serial LCD to Port 14 with a cable. The instructions are about the same as connecting the Ping))). Start by disconnecting power to your board. The jumper for Vdd and Vin between the servo ports has to be set to Vdd. The cable has to be plugged into the X4 header so that the black wire is closest to the X4 label and the white wire is closest to the 14 label. When connecting the other end of the cable to the Parallax Serial LCD, make sure the black wire connects to GND, the red wire to 5V, and the white wire to RX.

All other BASIC Stamp Educational Boards

This section is for connecting the Ping))) sensor and Parallax Serial LCD to one of the following BASIC Stamp educational boards:

- BASIC Stamp HomeWork Board
- Board of Education Rev A (Serial version)
- Board of Education Rev B (Serial version)

√ Disconnect power from your board.
√ Build the breadboard connections as shown in Figure 2-10.

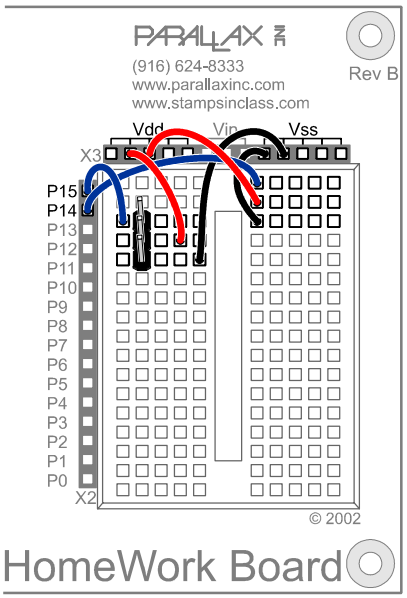


Figure 2-10
Breadboard Wiring for Ping)))
Sensor Cable Connection

√ Plug the Parallax Serial LCD into the breadboard as shown in Figure 2-11 on page 56.

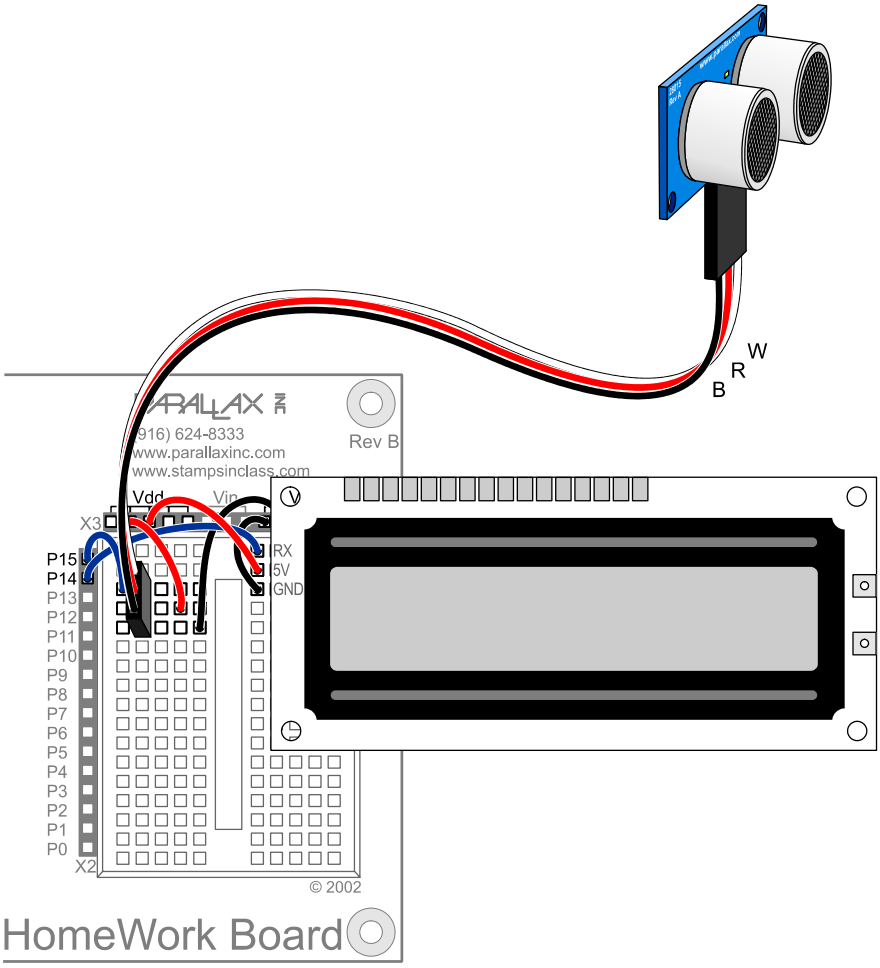
√ Plug one end of the extension cable into the 3-pin header, making sure the white, red, and black wires are oriented as shown. The black wire should be connected to Vss, the Red wire to Vdd, and the white wire to P15.

√ Connect the other end of the cable so that the black wire is connected to the Ping)))'s GND pin, the red wire is connected to the 5 V pin, and the white wire is connected to the RX pin. Double-check all your connections, including your jumper setting, and make sure they are correct.

WARNING! Do not connect power to your board until you are positive the connections are correct. If you make a mistake with the LCD connections, the Parallax Serial LCD could be permanently damaged.

√ Reconnect power to your board.

Figure 2-11: Breadboard Connections for Ping))) Sensor and Parallax Serial LCD



LCD Distance Display

It doesn't take much in the way of changes to modify PingMeasureCmAndIn.bs2 to make it display its measurements on the LCD. First, an Initialization section has to be added so that the program waits for the power supply to stabilize and then turns on and clears the LCD.

```
PAUSE 200
SEROUT 14, 84, [22, 12]
PAUSE 5
```

Next, the **DEBUG** commands need to be changed to **SEROUT** commands. Here are the **DEBUG** commands from PingMeasureCmAndIn.bs2.

```
DEBUG HOME, DEC3 cmDistance, " cm"
DEBUG CR, DEC3 inDistance, " in"
```

The Debug Terminal control characters (**HOME** and **CR**) need to be changed to control codes that will place the cursor in the LCD.

```
SEROUT 14, 84, [128, DEC3 cmDistance, " cm"]
SEROUT 14, 84, [148, DEC3 inDistance, " in"]
```

Example Program: PingLcdCmAndIn.bs2

This program is a modified version of PingMeasureCmAndIn.bs2 from the previous activity. Instead of displaying its measurements in the Debug Terminal, it displays them on the Parallax Serial LCD.

- √ Connect a battery to your board.
- √ Enter, save and run PingLcdCmAndIn.bs2.
- √ Disconnect the serial cable, and take your board with you to wherever you want to test the Ping))) sensor's measurements.

```
' Smart Sensors and Applications - PingLcdCmAndIn.bs2
' Measure Distance with the Ping))) sensor and display on LCD

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

' Conversion constants for room temperature measurements.
CmConstant    CON   2260
InConstant    CON   890

cmDistance  VAR   Word
```

```
inDistance  VAR   Word
time        VAR   Word

PAUSE 200
SEROUT 14, 84, [22, 12]
PAUSE 5
DEBUG CLS, "Program running..."

DO

  PULSOUT 15, 5
  PULSIN 15, 1, time

  cmDistance = cmConstant ** time
  inDistance = inConstant ** time

  SEROUT 14, 84, [128, DEC3 cmDistance, " cm"]
  SEROUT 14, 84, [148, DEC3 inDistance, " in"]

  PAUSE 100

LOOP
```

Your Turn - Customizing the Display

- √ The measurements are currently left-justified. Try centering them.
- √ Try right justifying the measurements and displaying "Distance: " before the cm measurement on the LCD's top line.
- √ Modify the program so that it displays both of the distance measurements on the top line. Then, display the actual echo time on the bottom line. You can display it in millionths of a second (µs) by multiplying the time variable by 2 before displaying it. Make sure your program waits until after it has done its distance conversions before multiplying time by 2.

ACTIVITY #5: TEMPERATURE'S EFFECT ON THE SPEED OF SOUND

This activity investigates changes in the speed of sound caused by changes in air temperature. These changes in the speed of sound can result in visible changes to your distance measurements.

Speed of Sound Vs Temperature and Percent Error Measurements

The speed of sound changes with air temperature, humidity, and even air quality. Neither humidity nor air quality make enough of a difference to figure into Ping))) sensor

distance calculations. Air temperature, on the other hand, can cause measurable distance errors. The speed of sound increases by 0.6 meters per second (m/s) for every degree-Celsius (°C) increase in temperature. Since the speed of sound is about 331.5 m/s at 0 °C, we can use this equation to calculate the speed of sound at a given temperature.

$$C_{air} = 331.5 + (0.6 \times T_C)\ m/s$$



Converting from °F to °C and Visa Versa

To convert a degree-Fahrenheit to Celsius, subtract 32 from T_F (the Fahrenheit measurement), then divide by 1.8. The result will be T_C, the Celsius equivalent. To convert from Celsius to Fahrenheit, multiply T_C by 1.8, then add 32. The result will be T_F.

$$T_C = (T_F - 32) \div 1.8 \qquad T_F = 1.8 \times T_C + 32$$

Below are examples for the speed of sound at two fairly comfortable, but slightly different indoor temperatures.

Example 1: Calculate the speed of sound at 22.2 °C, which is approximately 72 degrees Fahrenheit (°F).

$$C_{air}\ (22.2°C) = 331.5 + (0.6 \times 22.2\ m/s) = 344.8\ m/s$$

Example 2: Calculate the speed of sound at 25 °C, which is 77 degrees Fahrenheit (°F).

$$C_{air}\ (25°C) = 331.5 + (0.6 \times 25)\ m/s = 346.5\ m/s$$

How much of a difference does this make to your distance measurements? We can calculate the percent error this will propagate with the percent error equation.

$$\%\ error = \frac{actual - predicted}{predicted} \times 100\%$$

If the predicted temperature in the room is 72 °F (22.2 °C), and the actual temperature is 77 °F (25 °C), the error is 0.49 percent. Half a percent error would cause you to have to

move the object half a centimeter beyond 100 cm before it would transition from 99 to 100 cm.

$$\%\,\mathbf{error} = \frac{346.5 - 344.8}{344.8} \times 100\%$$

$$= 0.49\%$$

Your Turn - Room Temperature Vs. Freezing

- √ Calculate the percent measurement error that would result from assuming that the ambient temperature is freezing (32 °F, 0 °C), but it's actually room temperature (72 °F, 22.2 °C).
- √ How far off would the measurement be if the object is 1 m away?
- √ Use the procedure introduced in Activity #2 to calculate the speed of sound and **`CmConstant`** for measurements at 0 °C.
- √ Save PingMeasureCm.bs2 as PingMeasureCmYourTurn.bs2
- √ Run the program before modifying it and test the distance measurement of an object at 1 m.
- √ Modify the **`cmConstant CON`** directive with the value for 0 °C.
- √ Re-test the program with an object at 1 m. How close is your predicted error to the actual error?

SUMMARY

The BASIC Stamp requests a measurement from the Ping))) sensor by sending it a brief pulse, which causes it to emit a 40 kHz chirp. Then, the Ping))) listens for an echo of that chirp. It reports the echo by sending a pulse back to the BASIC Stamp that is equal to the time it took for the Ping))) sensor to receive the echo.

To calculate the distance based on the echo time measurement, the speed of sound must be converted into units that are convenient for the BASIC Stamp. This involves converting meters per second to centimeters per **PULSIN** measurement units. The resulting value also has to be converted to a value that can be used with the multiply high (******) operator by multiplying it by 65536.

The speed of sound in air is $c_{air} = 331.5 + (0.6 \times T_C)$ m/s. While the speed of sound changes with temperature, the resulting measurement errors are small, especially at room temperature.

Questions

1. What is the Ping))) sensor's range?
2. What does ultrasonic mean?
3. What signal does the Ping))) sensor send the BASIC Stamp and how does it correspond to a distance measurement?
4. What three sensor-object orientation scenarios could cause the Ping))) sensor to return an incorrect distance measurement?
5. What time increments does the **PULSIN** command return when using a BS2?
6. What's the speed of sound in air at room temperature?
7. How does **CmConstant** relate to the speed of sound in air?
8. What do you have to do to the jumper between the X4 and X5 servo headers on the Board of Education to provide the correct supply voltage to devices like the Ping))) sensor and the Parallax Serial LCD? What might happen if this jumper is not correctly set?
9. What commands have to be modified if you want to make the Parallax LCD display what the Debug Terminal was displaying?
10. What role does air temperature play in the speed of sound in air?

Exercises

1. Calculate how many meters away an object is if the echo time is 15 ms, and the temperature is 22.5 °C.
2. Calculate the °C equivalent of 100 °F.
3. Calculate the foot equivalent of 30.48 cm.
4. Calculate the percent error if **`CmConstant`** is for 37.8 °C but the temperature is 0 °C. Predict what the measured distance would be if the object were placed at 0.5 m.

Projects

1. Add an LED circuit to your board and program the BASIC Stamp to make the LED flash when there is no object in range.
2. Use a piezospeaker to make an alarm that signals when people pass through a doorway. The Ping))) sensor should be mounted next to the doorway, pointing across the path people walk when they enter or leave.

Solutions

Q1. 3 centimeters to 3.3 meters.
Q2. Sound with frequencies above 20 kHz.
Q3. A high pulse, whose duration corresponds to the time it took for the chirp sound to travel to the object and back.
Q4. a) Distance over 3 meters, b) Shallow angles, c) Objects that are too small.
Q5. 2µs increments.
Q6. 344.8 m/s.
Q7. **`CmConstant`** is the **`**`** equivalent of the speed of sound in air divided by 10000, or 0.03448.
Q8. The jumper should be set to the Vdd position, otherwise the LCD could be damaged.
Q9. All **`DEBUG`** commands have to be modified, and control characters have to be modified to use the LCD's control codes.
Q10. A very important role, with the speed increasing 0.6 m/s for every degree C increase in air temperature.
E1. The object is 2.59 m away.
E2. 100 °F = 37.7 °C
E3. 30.48 cm = 1.0 ft.
E4. % error = +/- 6.84%; measured distance = 0.466 m.
P1. This example solution places an active-high LED on P13.

```
' Smart Sensors and Applications - Ch2_Project1.bs2
' Indicate out-of-range with flashing LED.  Adjust MaxDistance to suit.
' {$STAMP BS2}
' {$PBASIC 2.5}

LED             PIN     13              ' Red LED active high
LCD             PIN     14              ' Parallax Serial LCD
Ping            PIN     15              ' Parallax Ping))) sensor

CmConstant      CON     2260            ' Calc roundtrip time of sound
InConstant      CON     890
MaxDistance     CON     361             ' Maximum can measure (empirical)
cmDistance      VAR     Word            ' Distance in centimeters
time            VAR     Word            ' Round trip echo time

PAUSE 200                               ' Initialize LCD
SEROUT LCD, 84, [22, 12]
PAUSE 5

DO
  LOW LED                               ' LED off before each measurement
  PULSOUT 15, 5                         ' Start Ping)))
```

```
  PULSIN 15, 1, time                ' Read echo time
  cmDistance = cmConstant ** time   ' Calculate distance from time
  SEROUT LCD, 84, [128, DEC3 cmDistance, " cm"]  ' Print distance on
                                                 ' LCD scrn
  IF cmDistance >= MaxDistance THEN HIGH LED     ' Toggle LED if out
                                                 ' of range
  PAUSE 100
LOOP
```

P2. Example solution:

```
' Smart Sensors and Applications - Ch2_Project2.bs2
' Make a sound when someone passes through the doorway.

' {$STAMP BS2}
' {$PBASIC 2.5}
' -----[ I/O Definitions ]-------------------------------------------
Ping           PIN     15            ' Parallax Ping))) sensor
Speaker        PIN     9             ' Optional speaker

' -----[ Constants ]-------------------------------------------------
InConstant     CON     890
Doorjamb       CON     35            ' Doorway width is 35 inches

' -----[ Variables ]-------------------------------------------------
inDistance     VAR     Word
time           VAR     Word          ' Round trip echo time
counter        VAR     Nib

' -----[ Main Routine ]----------------------------------------------
DO
  GOSUB Read_Ping
  GOSUB Calc_Distance
  IF (inDistance < Doorjamb) THEN
    GOSUB Sound_Alarm
  ENDIF
LOOP
' -----[ Subroutines ] ----------------------------------------------
Read_Ping:
  PULSOUT 15, 5                     ' Start Ping)))
  PULSIN 15, 1, time                ' Read echo time
  RETURN

Sound_Alarm:
  FREQOUT Speaker, 300, 3300        ' Bing
  PAUSE 50
  FREQOUT Speaker, 450, 2200        ' Bong
  RETURN

Calc_Distance:
  inDistance = inConstant ** time   ' These are the whole measurements
  RETURN
```

Chapter 3: The Memsic Dual-Axis Accelerometer

3

Acceleration is a measure of how quickly speed changes. Just as a speedometer is a meter that measures speed, an accelerometer is a meter that measures acceleration. You can use an accelerometer's ability to sense acceleration to take a variety of measurements that can be very useful to electronic and robotic projects and designs. Here are some examples:

- Acceleration
- Tilt and tilt angle
- Incline
- Rotation
- Vibration
- Collision
- Gravity

Accelerometers are already used in many different devices, including personal electronics, specialized equipment and machines. Here are just a few examples:

- Self-balancing robots
- Tilt-mode game controllers
- Model airplane autopilots
- Car alarm systems
- Crash detection/airbag deployment systems
- Human motion monitoring systems
- Leveling tools

Once upon a time, accelerometers were large, clunky and expensive instruments that did not lend themselves to electronic and robotic projects. This all changed thanks to the advent of MEMS, micro-electro-mechanical-systems. MEMS technology is responsible for an ever-increasing number of formerly mechanical devices being designed right onto silicon chips.

The accelerometer you will be working with in this text is the Parallax Memsic 2125 Dual Axis Accelerometer module shown in Figure 3-1. This module measures less than $^1/_2$" × $^1/_2$" × $^1/_2$", and the accelerometer chip itself is less than $^1/_4$" × $^1/_4$" × $^1/_8$".

Figure 3-1: Accelerometer Module and MX2125 Chip



Accelerometer Module



MX2125 Chip

People naturally sense acceleration on three axes: forward/backward, left/right and up/down. Just think about the last time you were in the passenger seat of a car on a hilly and curvy road. Forward/backward acceleration is the sensation of speeding up and slowing down. Left/right acceleration makes you lean when making turns, and up down acceleration is what you felt going over hills.

Instead of the three axes people sense, the MX2125 accelerometer senses acceleration on two axes. The acceleration it senses depends on how it's positioned. By holding it one way, it can sense forward/backward and left/right. If you hold it a different way, it can sense up/down and forward/backward. Two axes of acceleration is enough for many of the applications listed earlier. While you can always mount and monitor a second accelerometer to capture that third axis, three-axis accelerometers are also common.

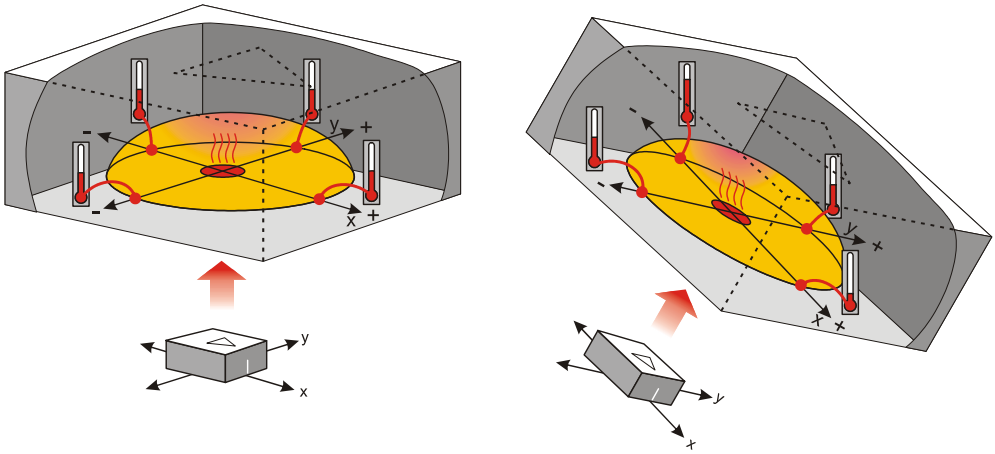


For a 3-axis accelerometer, try our Hitachi H48C Tri-Axis Accelerometer Module, Parallax part #28026.

THE MX2125 ACCELEROMETER – HOW IT WORKS

The MX2125's design is amazingly simple. It has a chamber of gas with a heating element in the center and four temperature sensors around its edge. Just as hot air rises and cooler air sinks, the same applies to hot and cool gasses. If you hold the accelerometer still, all it senses is gravity, and tilting it gives us an example of how it senses static acceleration. When you hold the accelerometer level, the hot gas pocket rises to the top-center of the accelerometer's chamber, and all the sensors will measure the same temperature. Depending on how you tilt the accelerometer, the hot gas will collect closer to one or maybe two of the temperature sensors.

Figure 3-2: The Accelerometer's Heated Gas Pocket



By comparing the sensor temperatures, both static acceleration (gravity and tilt) and dynamic acceleration (like taking a ride in a car) can be detected. If you were to take the accelerometer for a car ride, the hotter and cooler gasses would slosh around in the chamber in a manner similar to a container half-full of water, and the sensors would detect this.

In most situations, making sense out of these measurements is a simple task thanks to the electronics inside the MX2125. The MX2125 converts the temperature measurements into signals (pulse durations) that are easy for the BASIC Stamp microcontroller to measure and decipher.

ACTIVITY #1: CONNECTING AND TILT-TESTING THE MX2125

In this activity, you will connect the accelerometer module to the BASIC Stamp, run a test program, and verify that it can be used to sense tilt.

Parts Required

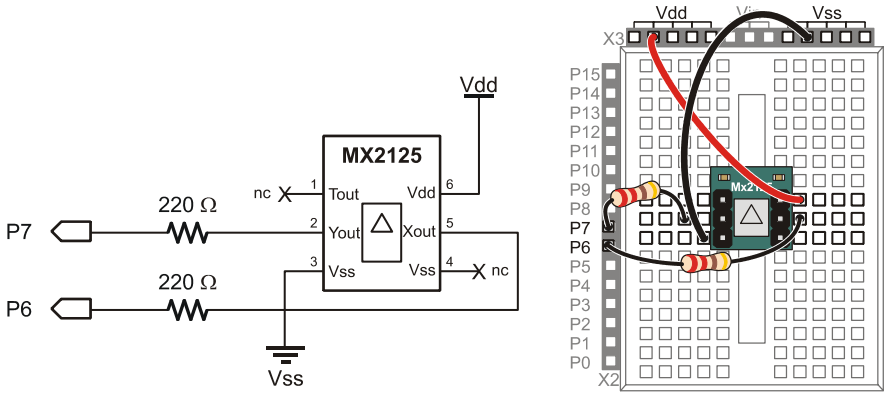
(2) 3-inch Jumper Wires
(2) Resistors – 220 Ω
(1) Memsic MX2125 Dual-Axis Accelerometer

Accelerometer Electrical and Signal Connections

Figure 3-3 shows how to connect the accelerometer module to the Board of Education's power supply, along with the BASIC Stamp I/O pin connections you will need to make to run the example program.

√ Connect the accelerometer module using Figure 3-3 as your guide.

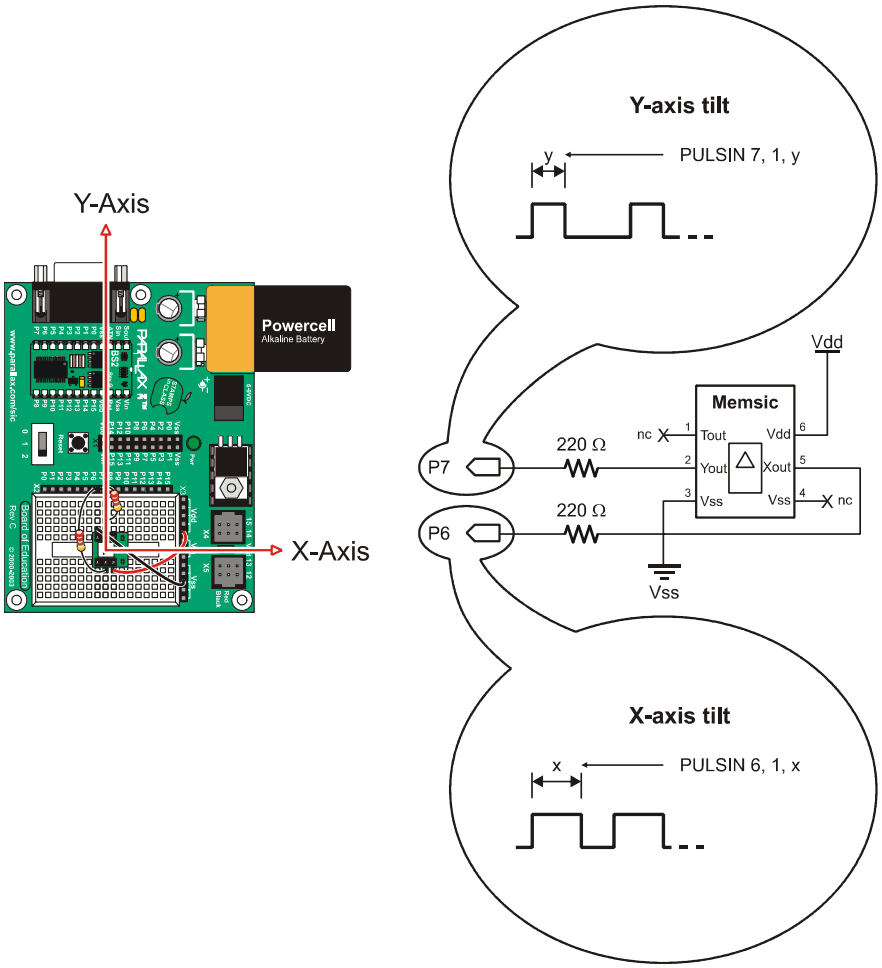
Figure 3-3: Accelerometer Schematic and Wiring Diagram



Listening to the Accelerometer's Signals with the BASIC Stamp

The two axes the MX2125 uses to sense gravity and acceleration are labeled x and y in Figure 3-4. It will help if you set your board flat on the table in front of you as shown in the figure. That way, the x and y axes point the same directions they do on most xy plots.

Figure 3-4: Accelerometer Axis Pulse Measurements



For room temperature testing, you can get a pretty good indication of tilt by just measuring the high times of the pulses sent by the MX2125's Xout and Yout pins with the **`PULSIN`** command. Depending on how far you tilt the board and in which direction, the **`PULSIN`** time measurements should range from 1875 to 3125. When the board is level, the **`PULSIN`** command should store values in the neighborhood of 2500.

- √ Make sure your board is sitting flat on the table, oriented with its x and y axes as shown in Figure 3-4.
- √ Enter and run SimpleTilt.bs2.

```
' Smart Sensors and Applications - SimpleTilt.bs2
' Measure room temperature tilt.

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

x               VAR     Word
y               VAR     Word

DEBUG CLS

DO

  PULSIN 6, 1, x
  PULSIN 7, 1, y

  DEBUG HOME, DEC4 ? X, DEC4 ? Y

  PAUSE 100

LOOP
```

- √ Check to make sure the Debug Terminal reports that the **x** and **y** variables are both storing values around of 2500 as shown in Figure 3-5.

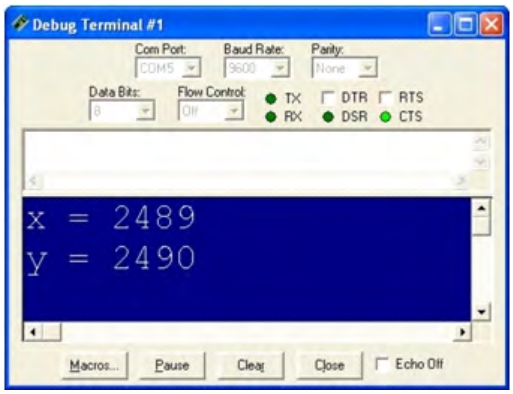


Figure 3-5
Debug Terminal Output

- √ Grab the edge of the board with the Y-Axis label and gradually lift it toward you. The **y** value should increase as you increase the tilt.
- √ Keep tilting the board toward you until it's straight up and down. The Debug Terminal should report that the **y** variable stores a value near 3125.
- √ Lay the board flat again.
- √ Next, instead of tilting the board toward you, gradually tilt it away from you. The **y** value should drop below 2500 and gradually decrease to 1875 as you tilt the board until it's straight up and down.
- √ Lay the board flat again.
- √ Repeat this test with the x-axis. As you tilt the board up with your right hand, the **x** value should increase and reach a value near 3125 when the board is vertical. As you tilt the board upward with your left hand, the **x** value should approach 1875.
- √ Finally, hold your board in front of you, straight up and down like a steering wheel.
- √ As you slowly rotate your board, the **x** and **y** values should change. These values will be used in another activity to determine the rotation angle in degrees.

ACTIVITY #2: MOBILE MEASUREMENTS

This activity will display the Memsic Accelerometer's measurements on the Parallax Serial LCD. Provided you're using a battery, after programming you can disconnect from your computer and take the setup to remote locations of your choosing.

Connecting Both Modules to the BASIC Stamp

Both the Memsic Accelerometer and the Serial LCD can fit on your board at the same time, so there will be no need for extension cables unless you chose to mount the Parallax Serial LCD near the Board of Education or in a project box.

Parts Required

(1) Memsic 2125 Accelerometer
(1) Parallax Serial LCD (2×16)
(5) Jumper Wires
(2) Resistors 220 Ω

Building the Accelerometer and LCD Circuits

The schematics shown in Figure 3-6 are identical to the ones that have been used for the Memsic accelerometer and Parallax Serial LCD in previous activities.

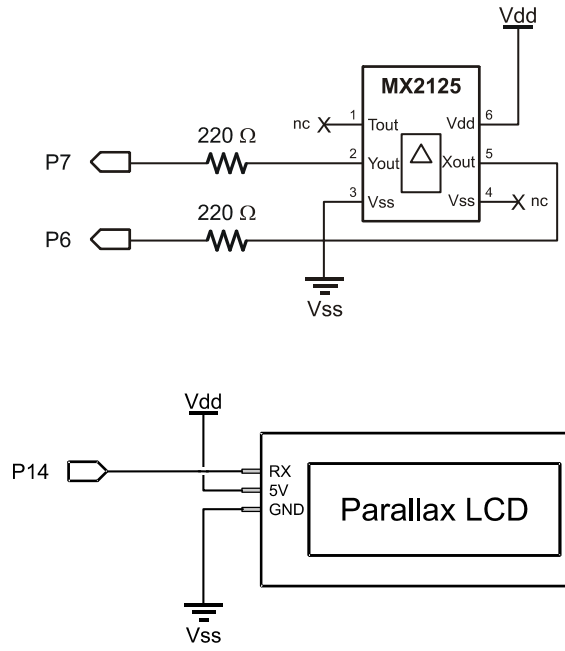


Figure 3-6
Ping))) Sensor and Parallax Serial LCD Schematics

The wiring diagrams for the Memsic Accelerometer and Parallax Serial LCD shown in Figure 3-7 and Figure 3-8 are a combination of the two earlier wiring diagrams for the individual modules.

√ Build the wiring diagram shown in Figure 3-7 first.

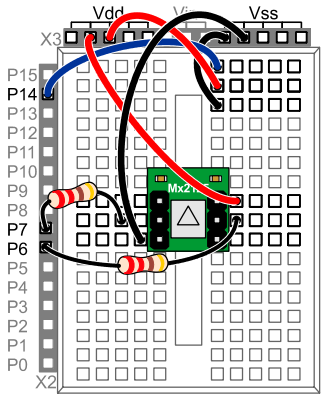
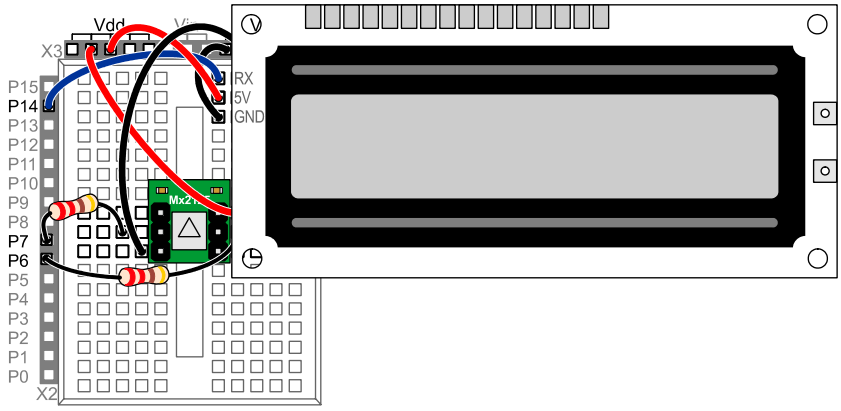


Figure 3-7
Debug Terminal Output

√ Then insert the Parallax Serial LCD as shown Figure 3-8.

Figure 3-8: Debug Terminal Output



LCD Tilt Display

Modifying any of the accelerometer example programs from this chapter to make them display measurements on the LCD is typically a 2-step process. First comes adding the LCD initialization routine, and next comes replacing **DEBUG** commands with **SEROUT** commands that will display the information on the LCD.

Always remember to add this initialization, either before the Main Routine, or in small programs, before the first **DO** keyword. That will keep the initialization from being repeated over and over again in the **DO**...**LOOP** with the rest of the program. Make sure to keep it out of the main **DO**...**LOOP** because it could cause the display to flicker.

```
' Initialize LCD
PAUSE 200
SEROUT 14, 84, [22, 12]
PAUSE 5
```

Next, the **DEBUG** commands need to be changed to **SEROUT** commands. Here is the **DEBUG** command from SimpleTilt.bs2.

```
DEBUG HOME, DEC4 ? X, DEC4 ? Y
```

The **HOME** control should be replaced with 128, which is the LCD's home character. The **?** directive displays the variable name, and then a carriage return (**CR**) character afterwards. Remember from Chapter 1 that **CR** is the one control character that happens to be the same for both the Debug Terminal and the Parallax Serial LCD? Because of this, we can leave the **?** directive in the **SEROUT** commands to the LCD. Here is a **SEROUT** command that does the equivalent display on the Parallax Serial LCD.

```
SEROUT 14, 84, [128, DEC4 ? X, DEC4 ? Y]
```

Example Program: SimpleTiltLcd.bs2

This program is a modified version of SimpleTilt.bs2 from the previous activity. Instead of displaying its measurements in the Debug Terminal, it displays them in the Parallax Serial LCD.

- √ Connect a battery to your board.
- √ Enter, save and run SimpleTiltLcd.bs2.
- √ Disconnect the serial cable, and take your board with you to wherever you want to test the Memsic Accelerometer's measurements.

```
' Smart Sensors and Applications - SimpleTiltLcd.bs2
' Measure room temperature tilt and display them on the Parallax Serial LCD.

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

x              VAR     Word
y              VAR     Word
```

```
' DEBUG CLS

' Initialize LCD
PAUSE 200
SEROUT 14, 84, [22, 12]
PAUSE 5

DO

  PULSIN 6, 1, x
  PULSIN 7, 1, y

'  DEBUG HOME, DEC4 ? X, DEC4 ? Y
  SEROUT 14, 84, [128, DEC4 ? X, DEC4 ? Y]

  PAUSE 100

LOOP
```

Your Turn - Customizing the Display

The carriage return (**CR**) that's built into the **?** operator makes it more difficult to display information after the **x** or **y** variable values. You can rewrite the **DEBUG** and **SEROUT** commands to perform the same operations like this.

```
DEBUG HOME, "x = ", DEC4 x, CR, "y = ", DEC4 y
```

This **SEROUT** command displays the same information on the Parallax Serial LCD. Notice how the control code 128 places the cursor on Line 0, character 0. Instead of a **CR** control character, 148 places the LCD's cursor on Line 1, character 0.

```
SEROUT 14, 84, [128, "x = ", DEC4 x, 148, "y = ", DEC4 y]
```

With this modified **SEROUT** command, it's easier to display characters after each value. For example, here is a **SEROUT** command that multiplies each measurement by 2 and displays "us" afterwards.

```
SEROUT 14, 84, [128, "x = ", DEC4 (2 * x), " us",
                148, "y = ", DEC4 (2 I y), " us"]
```

While "us" isn't really the same as "µs" because we are using u instead of the Greek character mu, most people take its meaning. You can also make a custom character for mu. This will involve adding a **SEROUT** command to the beginning of the program that defines a custom character. Then, you will have to display that custom character where "u" is currently displayed.

ACTIVITY #3: SCALING DOWN AND OFFSETTING INPUT VALUES

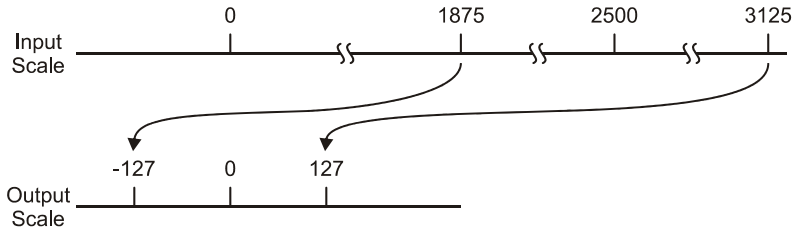
When working with the MX2125 and BASIC Stamp 2, tilt measurements range between 1875 and 3125. This range may have to be scaled and offset any number of ways. For example, Activity #4 scales this to a range of −100 to 100. Activity #5 scales it to −127 and 127.

Introducing an offset into a range of values is easy, and typically involves an addition or subtraction operation. Scaling can be a little trickier, especially with a processor like the BASIC Stamp, which does all its calculations with integer math. This activity introduces the simplest and most accurate way to scale a larger range of values into a smaller range with a PBASIC program. The technique introduced here helps prevent errors from creeping into your sensor measurements with each successive PBASIC calculation, and it will be used and re-used in many of this book's activities.

Scale and Offset Example

In this first example, we'll take an input value that could be anywhere between 1875 and 3125, and scale and offset it to a corresponding output value that falls in a range from −127 to 127. Figure 3-9 shows how this should work. The position of the value in the output scale should be proportional to the position of the value in the input scale. For example, if the input value is 2500, which is halfway between 1875 and 3125, we should expect the output value to be 0, which is half way between −127 and 127.

Figure 3-9: Example Input and Output Scales



To apply scale and offset in PBASIC, remember these three steps:

1) Apply offset to align the input scale to zero.
2) Apply the scale.
3) Apply any additional offset that is needed for our output scale.

Figure 3-10 shows how to apply these steps with a single PBASIC command that performs both scaling and offset. Keep in mind that PBASIC calculations work from left to right unless they are overridden with parentheses. So the first thing this calculation does is subtract 1875 from the input value. The new range is now 0 to 1200 instead of 1875 to 3215. Next, **** 13369** scales value down to 0 to 254. After the range has been scaled, 127 is subtracted from it resulting in −127 to 127.

```
                (1875 to 3125)→(0 to 1200)
                 |
                 |      (0 to 1200)→(0 to 254)
                 |       |
                 |       |      (0 to 254)→(-127 to 127)
                 |       |       |
value = value - 1875 ** 13369 - 127
```

Figure 3-10: Scaling the Value Variable

Choosing the Right ** Constant for Scaling

The value 13369 used with the ****** constant to scale (0 to 1250) to (0 to 254) was determined by substituting the number of elements in the input and output scales into this equation. The number of output scale elements is 255, including 0, and the number of input scale elements is 1251, also including 0. Use this equation whenever you need to fit a larger scale into a smaller one with the ****** operator.

$$\mathbf{ScaleConstant} = \mathbf{Int}\left[\mathbf{65536}\left(\frac{\mathbf{output\ scale\ elements}}{\mathbf{input\ scale\ elements} - \mathbf{1}}\right)\right]$$

$$\mathbf{ScaleConstant} = \mathbf{Int}\left[\mathbf{65536}\left(\frac{\mathbf{255}}{\mathbf{1251} - \mathbf{1}}\right)\right]$$

$$\mathbf{ScaleConstant} = \mathbf{Int}\left[\mathbf{13,369.344}\right]$$

$$\mathbf{ScaleConstant} = \mathbf{13369}$$



Always round your ScaleConstant result down, even if the result is already an integer! Otherwise, the largest value in your input scale might be one value outside the output scale's range.

Clamping the Input Range

The best way to make sure the output values do not exceed the output range is to make sure the input values do not go outside the input range. For example, if you do not want the output of this command to go outside −127 to 127, the most convenient approach is to make sure that the input values do not go below 1875 or above 3125. Here is a modified version of **`value = value - 1875 ** 13369 - 127`** that prevents the problem.

```
value = (value MIN 1875 MAX 3125) - 1875 ** 13369 - 127
```

Before subtracting 1875 from the **`value`** variable, this command uses two operators, **`MIN 1875`** and **`MAX 3125`**, to make sure **`value`** stores something in this range. If the **`value`** variable is storing a number in this range, the **`MIN`** and **`MAX`** operators leave it alone. However, if it's storing something less than 1875, **`MIN 1875`** will change value to 1875. Likewise, if it's storing something above 3125, **`MAX 3125`** changes it to 3125.

Example Program: TestScaleOffset.bs2

Figure 3-11 shows what the Debug Terminal looks like as the next example program is tested. When you enter input values (separated by commas) into the Debug Terminal's Transmit windowpane, the program displays the scaled and offset equivalent in the Debug Terminal's Receive windowpane.

√ Enter, save, and run TestScaleOffset.bs2.

```
' Smart Sensors and Applications - TestScaleOffset.bs2
' Test scaling from an input range of 1875 to 3125 to an output
' range of -127 to + 127.

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

value          VAR     Word

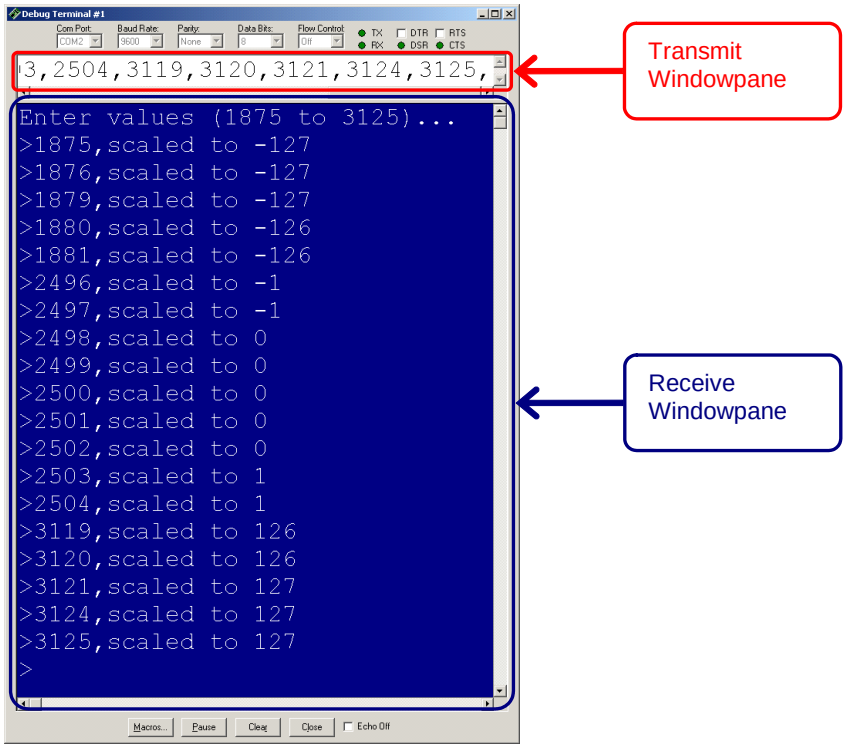
DEBUG CLS, "Enter values (1875 to 3125)...", CR

DO

  DEBUG ">"
```

```
  DEBUGIN DEC value
  value = (value MIN 1875 MAX 3125) - 1875 ** 13369 - 127
  DEBUG "scaled to ", SDEC value, CR
LOOP
```

Figure 3-11: Scaling Test



√ Click in the Debug Terminal's Transmit windowpane and enter this sequence, including commas: 1875, 1876, 1879, 1880, 1881, 2496, 2497, 2498, 2499, 2500, 2501, 2502, 2503, 2504, 3119, 3120, 3121, 3124, 3125.

√ Test various other values that range from 1875 to 3125, and verify with a calculator that the output value's position in the output range is proportional to the input value's position in the input range.

Your Turn - PBASIC and Negative Numbers

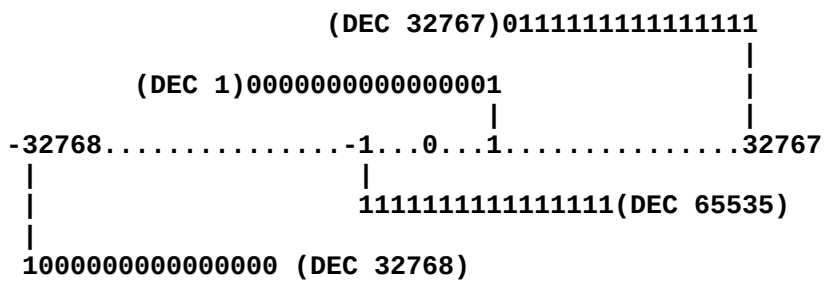
The last example program used the **`DEBUG SDEC`** modifier to display the **`value`** variable as a signed number. Remember that in PBASIC a word-sized variable can hold an unsigned value in the range from 0 to 65535 or a signed value from −32768 to +32767. That's because it uses the two's complement method for signed numbers. In this system, all positive numbers, in binary, begin with a 0 and all negative numbers, in binary, begin with a 1. Using two's complement, the values 0 to 32767 are represented by their normal 16-bit binary equivalents, but −1 to −32768 are not. Instead, those negative numbers are represented by the binary equivalents of 32768 to 65535.

Table 3-1: Two's Complement Signed Decimal and Binary Numbers

Unsigned Decimal	16-Bit Binary ↓Bit 15 Bit 0↓	Signed Decimal
1	0000000000000001	1
32767	0111111111111111	32767
32768	1000000000000000	-32768
65535	1111111111111111	-1

Picture a number line, as in Figure 3-12 . From 0 forward, the values 0 to 32767 are represented by their normal 16-bit binary equivalents: the value 1 is represented by binary 1, and so on, up to 32767. But −1 is represented by the binary equivalent of 65535, the largest word-sized value, which is all 1's. Going backwards along the negative values, the representative binary numbers get smaller until −32768 is represented by binary 32768.

Figure 3-12: Two's Complement Signed Decimal Number Line



The pattern emerges when you can see an unsigned decimal number compared next to its signed decimal and binary equivalents.

- √ Try running SignedNumbers.bs2 with different values for **x** until the pattern becomes clear to you. Try these **x** values: 0, 1, 2, -1, -2.
- √ Then try 65535, 65534, 32767, 32768, and 32769. Do you see how it works?

```
' Smart Sensors and Applications - SignedNumbers.bs2
' {$STAMP BS2}
' {$PBASIC 2.5}

x  VAR  Word
x = 32768      '<<< Enter new values for x here, and re-run the program

DEBUG "you entered decimal: ", DEC x, CR
DEBUG "signed decimal: ", SDEC x, CR
DEBUG "16-bit binary: ", BIN16 x, CR
```

In PBASIC, only word variables can hold signed numbers, so *all* signed numbers have 16 bits. By looking at the leftmost bit, Bit 15, we can know whether a signed number is negative or positive. You can use **value.BIT15** as a variable that tells you whether **value** is a positive or negative number. If **value.BIT15** is equal to 0, the number is positive. If it is equal to 1, the number is negative.

This is an important hint, because some PBASIC operators only work with positive integers, such division "/" and modulus "//". When using these operators, it is handy to save the sign of a number, perform the operation with its absolute value, then reapply the sign afterwards. In fact, we will be doing that later in Chapter 4.

Your Turn - A Closer Look at the ScaleConstant and ** Operator

For small input and output ranges, we can examine them with a calculator, pencil and paper. Let's take 0 to 10 as our input scale, and 0 to 2 as our output scale. The first step is to figure out what the constant for the **`**`** operation should be, by using the scale constant equation.

$$\mathbf{ScaleConstant} = \mathbf{Int}\left[\mathbf{65536}\left(\frac{\mathbf{output\ scale\ elements}}{\mathbf{input\ scale\ elements} - \mathbf{1}}\right)\right]$$

There are three elements in the output scale, 0, 1, and 2. There are 11 elements in the input scale, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10. Remembering to round down to the nearest integer, the result is 19660 which is the constant to use with the **`**`** operator.

$$\mathbf{ScaleConstant} = \mathbf{Int}\left[\mathbf{65536}\left(\frac{\mathbf{3}}{\mathbf{11} - \mathbf{1}}\right)\right]$$

$$\mathbf{ScaleConstant} = \mathbf{Int}[\mathbf{19{,}660.8}\,]$$

$$\mathbf{ScaleConstant} = \mathbf{19660}$$

The term **`value = value ** 19660`** multiplies the value variable by:

$$\mathbf{19660} \div \mathbf{65536} \approx \mathbf{0.29999} \rightarrow \mathbf{value} = \mathbf{value} \times \mathbf{0.29999}$$

Table 3-2 shows some examples of the BASIC Stamp calculations for each of the values in the input range for **`value = value ** 19660`**. Keep in mind that it's about the same as multiplying **`value`** by 0.29999 with a calculator. Since the BASIC Stamp is an integer math processor, it truncates any result to an integer value, effectively rounding down. Notice how the first four input values result in outputs of zero. Then, when the input value is 4, the result is 1.19996, which gets rounded to 1. As you perform the rest of the calculations in the table, notice how the output scale of 2 receives four input elements. If -1 was not used in the denominator, it would only receive one input element.

√ Finish the calculations in Table 3-2 for input values from 5 to 10.

Table 3-2: Measured voltages during charge cycle			
Value	**Scale Constant	Calculates Value	BASIC Stamp Integer Result
0	x 0.2999 =	0	0
1	x 0.2999 =	0.2999	0
2	x 0.2999 =	0.5998	0
3	x 0.2999 =	0.8997	0
4	x 0.2999 =	1.1996	1
5	x 0.2999 =		
6	x 0.2999 =		
7	x 0.2999 =		
8	x 0.2999 =		
9	x 0.2999 =		
10	x 0.2999 =		

- √ Save TestScaleOffset.bs2 as TestScaleOffsetYourTurn.bs2.
- √ Modify the program so that you can test Table 3-2 with the BASIC Stamp and Debug Terminal.
- √ Compare the Debug Terminal results to your table.

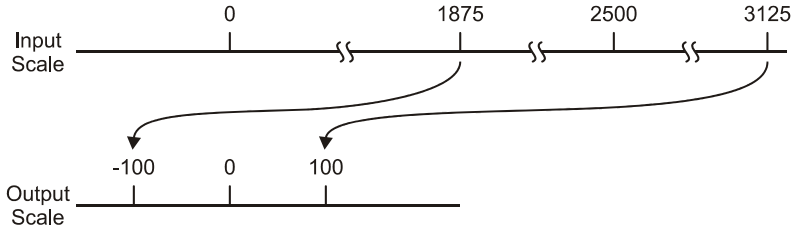
ACTIVITY #4: SCALING TO 1/100 G

The standard measure of gravity on the earth's surface is abbreviated "g." This activity demonstrates how to use the techniques introduced in the previous activity to display the number of hundredths of a g acting on the accelerometer's x and y axes.

From PULSIN to 1/100 g

The goal here is to modify the example program from Activity #1 so that it displays the x- and y-axis measurements in terms of 1/100 g instead of 2 µs units. It's another scaling and offset problem, but this time, we want to fit the 1875 to 3125 input scale into an output scale of −100 to 100 as shown in Figure 3-13.

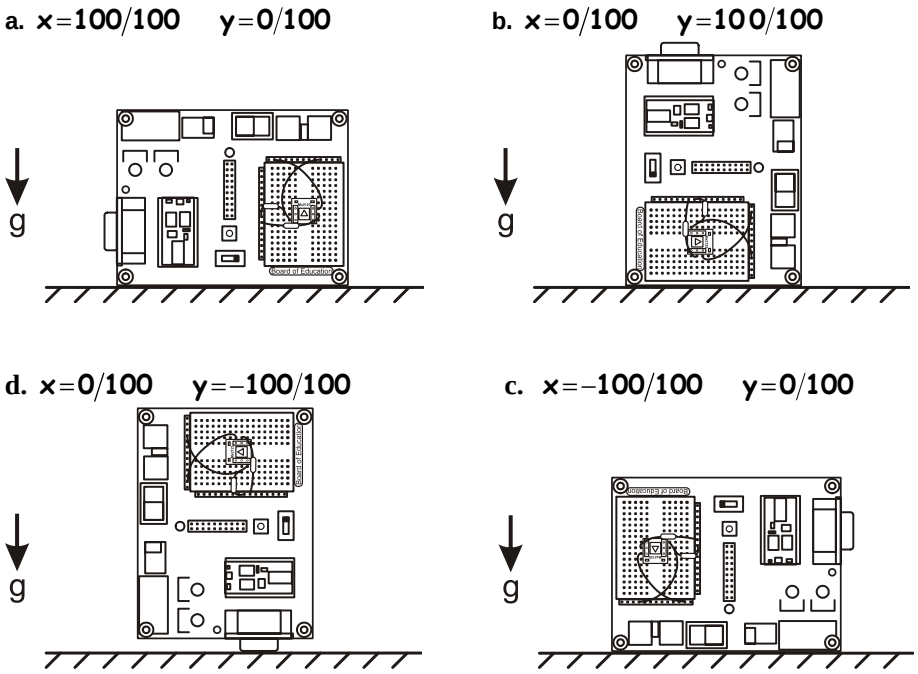
Figure 3-13: Scaling and Offset for 1/100 g.



Your Turn - Developing the Program

The goal here is to use the scaling techniques from Activity #3 to modify the program from Activity #1 so that it displays the x and y-axis measurements in terms of 1/100 g. Figure 3-14 shows the approximate readings you should expect after your modifications.

Figure 3-14: Sample Readings at Various Orientations (start at top left, rotate clockwise)



- √ Open SimpleTilt.bs2 from Activity #1 and save it as CentigravityTilt.bs2
- √ Follow the steps for scaling from Activity #3 and determine the ****** operation scale constants.
- √ Add lines of code to the program that scale the x and y values down to g/100.
- √ Modify the display so that it shows in the Debug Terminal.
- √ Test according to Figure 3-14 and troubleshoot if necessary.

ACTIVITY #5: MEASURING 360° VERTICAL ROTATION

The MX2125 has a built-in feature that allows you to use both the x and y axis tilt measurements to calculate the accelerometer's angle of rotation in the vertical plane, as shown in Figure 3-15. There are lots of applications where vertical tilt is useful, including virtual steering wheels for video games and counting bicycle wheel revolutions. This activity demonstrates how to calculate tilt on the vertical plane with the PBASIC **ATN** operator.

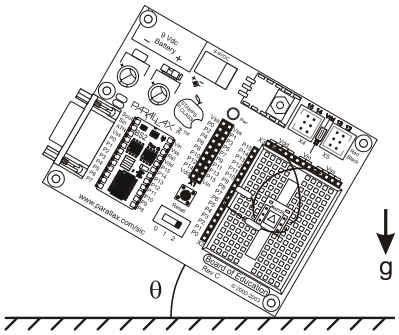
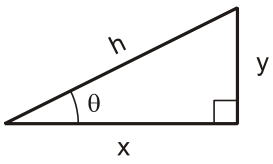


Figure 3-15
Tilt on the Vertical Plane

Calculating Arctangent with PBASIC

The tangent of an angle theta (θ) in a right triangle is the ratio of the opposite side of a right triangle (y) divided by the adjacent side (x). If you know the values of x and y, you can use the inverse tangent or arctangent to figure out the angle θ. The most common notations for arctangent are $\tan^{-1}$ and arctan.



$$\tan\theta = \frac{y}{x}$$

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

Figure 3-16
Tangent and Arctangent

The arctangent function can be used to determine the accelerometer's rotation angle with its x and y measurements. PBASIC has an operator called **`ATN`** you can use for calculating $\tan^{-1}(y/x)$. To calculate the arctangent of y/x and store it in a variable named **`angle`**, use the command **`angle = x ATN y`**.

$$\theta = \tan^{-1}\left(\frac{y}{x}\right) \rightarrow \mathbf{angle = x\ atn\ y}$$

Figure 3-17 is from the BASIC Stamp Editor's Help file, and it shows how the **`ATN`** operator works. Both the **`x`** and **`y`** variables have to be scaled to values between −127 and 127. The result of the **`ATN`** operator is the angle in binary radians, which is abbreviated to "brads". With brads, a circle is split up into 256 segments in the same way that degrees split a circle into 360 segments.

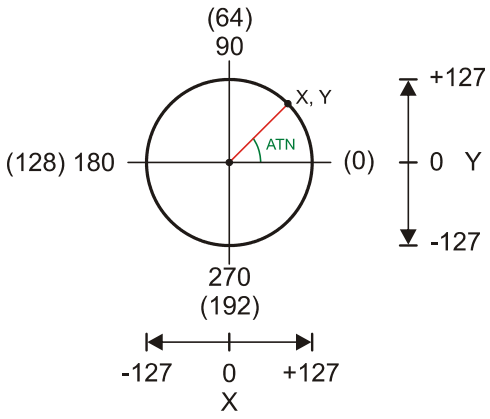


Figure 3-17
Unit Circle in Degrees and Binary Radians

Converting from Brads to Degrees with */

In the previous activity, we used the ****** operator to scale values down from a larger range to a smaller range. Converting from brads to degrees involves scaling a smaller scale of 0 to 255 to a larger scale of 0 to 359. The PBASIC ***/** operator is designed for this job.

When you use a command like **value = ScaleConstant */ value**, the **ScaleConstant** term is the number of 256ths you want to multiply the **value** variable by. For example, let's say you want to multiply **value** by 2.5. Multiply 2.5 by 256 and the result is 640. Now, if **value** starts as 10, the result of **value = 640 */ value** will be 25. If we want **value** to equal 2.5 times **value**:

$$\mathbf{ScaleConstant} = 2.5 \times 256 = 640$$

```
value = 640 */value      'multiply by 2.5
```



Remember

The ****** operator multiplies by a number of 65536ths.

The ***/** operator multiplies by a number of 256ths.

The rules of integer math for scaling from one scale to another still apply, even though we are converting from a smaller scale to a larger one. The only thing that will change is the scale constant, which is a numerator of 256 for ***/**, instead of 65536 for ******.

$$\mathbf{*/\ ScaleConstant} = \mathbf{Int}\left[256\left(\frac{\text{output scale elements}}{\text{input scale elements} - 1}\right)\right]$$

The input scale is 0 to 255, which has 256 elements, and the output is 0 to 359, which has 360 elements. The result after substituting these values into the ***/** scale constant equation is 361.

$$\mathbf{*/\ ScaleConstant} = \mathbf{Int}\left[256\left(\frac{360}{256 - 1}\right)\right]$$

$$\mathbf{*/\ ScaleConstant} = \mathbf{Int}[361.412]$$

$$\mathbf{*/\ ScaleConstant} = 361$$

This demonstrates that if the angle variable stores a measure of brads, and you want to store a measure of degrees instead, use this command:

```
angle = 361 */ angle
```



Most documents recommend angle = 360 */ angle. However, using a */ scale constant of 361 is slightly more accurate over the input/output ranges. Try comparing the results of this operation made with a BASIC Stamp to those made with a spreadsheet.

$$angle_{degrees} = (360/256) \times angle_{brads}$$

Round the result of $angle_{degrees}$ to the nearest integer. If the result has a fractional component of 0.5 or higher, round up. Otherwise, round down. Then compare it to the 256 possible Debug Terminal outputs with 360 */ angle, then repeat with 361 */ angle. A spreadsheet is useful for this comparison. If you try it, you'll se that the rate of integer value matches is much higher with 361 */ angle.

Example Program: TestAtn.bs2

This example program calculates angles based on the y and x values you enter into the Debug Terminal's Transmit windowpane.

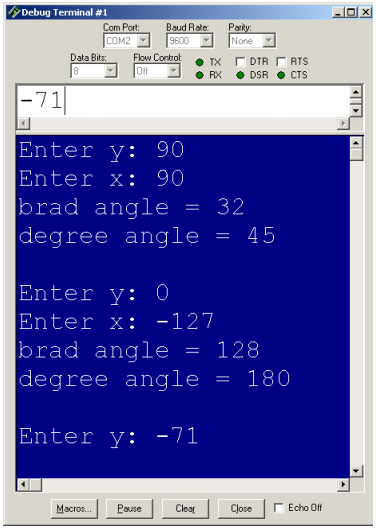
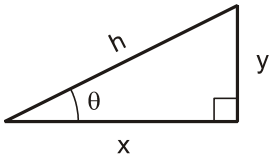


Figure 3-18
Tangent and Arctangent

To calculate x and y values to enter into the Debug Terminal, use these equations:



$y = h \sin\theta$

$x = h \cos\theta$

Figure 3-19
Tangent and Arctangent

For example, let's say that h = 127 and θ = 45°, then the x and y values to be entered into the Debug Terminal are both 90. If h = 100 and θ = 315°, the y value to enter into the Debug Terminal will be −71, and the x value will be 71. If h = 100 and θ = 180°, y will be 0 and x will be −127.

For h=127 and θ=45°	**For h=100 and θ=315°**	**For h=127 and θ=180°**
y=127 sin45°	**y=100 sin 315°**	**y=127 sin 180°**
=89.9	**=−71**	**=0**
≈90		
x=127 cos 45°	**x=100 cos 315°**	**x=127 cos 180°**
=89.9	**=71**	**=−127**
≈90		

- √ Enter, save, and run TestAtn.bs2
- √ Click the Debug Terminal's Transmit windowpane. When prompted for the x value, type 90 and press the carriage return. When prompted for y, type 90 and the carriage return again.
- √ Verify that the result is 32 brads = 45°.
- √ Repeat for the other x and y values just discussed.
- √ Use your calculator to determine the x and y values that correspond with various h and θ values. Compare your calculated results to the Debug Terminal's results.



Some values will be lower than you predict. For example, when h = 100 and θ = 30°, y = 50 and x = 87. The Debug Terminal will display 21 for the brad angle, which is correct, but 29 for the degree angle is not correct. It should be 30. This happens occasionally when scaling from a smaller range to a larger range. The 21 brads measurement corresponds to 29° and 22 brads corresponds to 31°.

```
' Smart Sensors and Applications - TestAtn.bs2
' Test BASIC Stamp arctangent calculations.

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

angle          VAR     Word
x              VAR     Word
y              VAR     Word

DO

  DEBUG "Enter y: "
  DEBUGIN SDEC y
  DEBUG "Enter x: "
  DEBUGIN SDEC x

  angle = x ATN y

  DEBUG "brad ", SDEC ? angle

  angle = angle */ 361

  DEBUG "degree ", SDEC ? angle, CR

LOOP
```

Your Turn - Testing Brad to Degree Conversion

As mentioned earlier, the ideal integer result comes from calculating $angle_{degrees}$ = (360/256) x $angle_{brands}$ and then rounding up if the value to the right of the decimal point is 5 to 9 or down if it is 1 to 4. You can generate a list of all 256 brad to degree conversions with this program.

```
' Smart Sensors and Applications - BradsToDegrees.bs2
' Display brad to degree conversions for */ 360 and */ 361.

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

```
angle          VAR     Word
brads          VAR     Word

DEBUG CLS, "brads  */ 360   */ 361", CR

FOR brads = 0 TO 255

  DEBUG DEC3 brads

  angle = brads */ 360
  DEBUG "      ", DEC3 angle

  angle = brads */ 361
  DEBUG "      ", DEC3 angle, CR

NEXT

END
```

√ Enter, save and run BradsToDegrees.bs2.

√ Use a spreadsheet or calculator to generate a list with this formula.

$$angle_{degrees} = (360/256) \times angle_{brads}$$

Remember to round up if the value to the right of the decimal point is 5 to 9 or down if it's 1 to 4.

√ Compare your results to the Debug Terminal display. How many exact matches occurred for **`*/ 360`**? How many occurred for **`*/ 361`**?

Measuring Tilt Angle On the Vertical Plane

The angle of your board's clockwise rotation in the vertical plane (θ) is the arctangent of the gravity's effect on the MX2125's y-axis (A_y) divided by its effect on its x-axis (A_x), as shown in Figure 3-20.

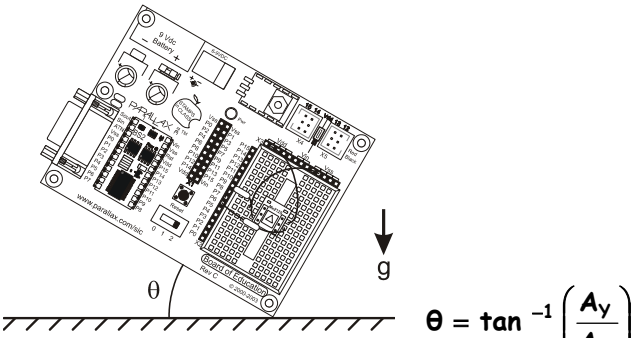
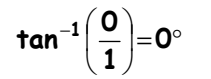
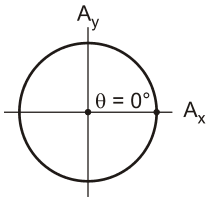


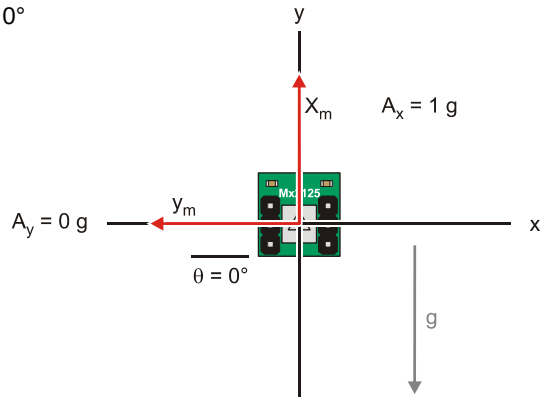
Figure 3-20
Clockwise Vertical Rotation

Here are a few examples of what the accelerometer detects and how it relates to the arctangent of the ratio of A_y to A_x. Figure 3-21 shows what the accelerometer senses at the 0° mark. If θ is 0°, then A_y senses 0-gravity (g), and A_x senses 1 g, the arctangent of 0/1 is 0°.

Figure 3-21: Accelerometer Rotated 0°





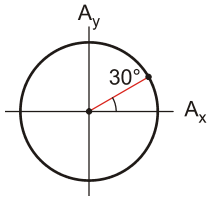


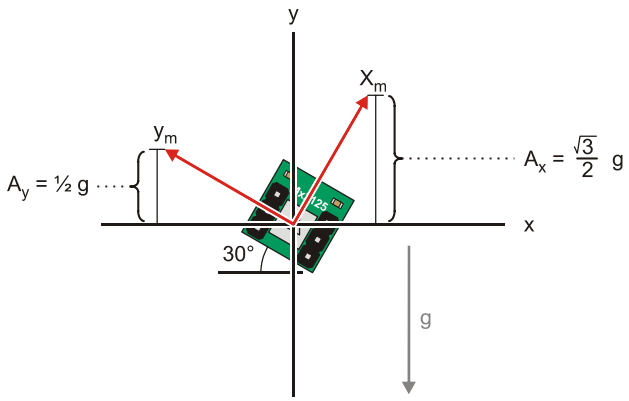
When the accelerometer is rotated 30° clockwise, as shown in Figure 3-22, the component of gravity acting on the accelerometer's x-axis is approximately √3/2 g. The

component of gravity acting on the y-axis is 1/2 g, and the arctangent of √3/2 ÷ 1/2 is 30°.

Figure 3-22: Accelerometer Rotated 30° Clockwise

$$\tan^{-1}\left(\frac{\frac{1}{2}}{\frac{\sqrt{3}}{2}}\right)=30^\circ$$

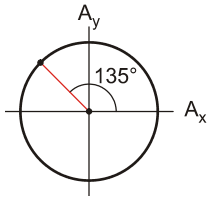


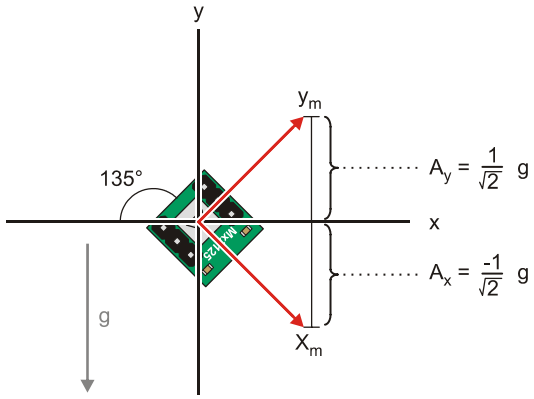


When the accelerometer is rotated to 135° clockwise as in Figure 3-23, the component of gravity acting on the accelerometer's x-axis is A_x = -1/√2, and the component acting on its y-axis is 1/√2. The arctangent of 1/√2 ÷ (−1/√2) is 135°.

Figure 3-23: Accelerometer Rotated 135° Clockwise

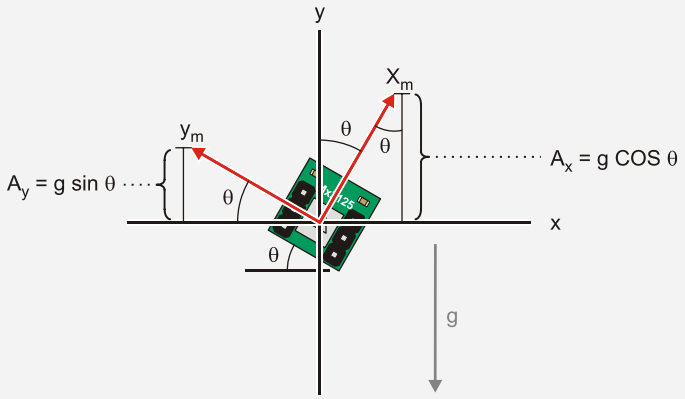
$$\tan^{-1}\left(\frac{\frac{1}{\sqrt{2}}}{\frac{-1}{\sqrt{2}}}\right)=135^\circ$$





The General Case

The angle of rotation (θ) is the inverse tangent or arctangent of the component of gravity acting on the Memsic 2125's Y sensing (A_Y) divided by the component of gravity acting on the X sensing axis (A_X). The figure below shows the MX2125 tilted at an angle θ, which rotates both sensing axes by θ. By applying a couple of geometry identities, θ is also inside the two triangles that show the components of gravity acting on each of the accelerometers sensing axes (x_m and y_m). The component of gravity acting on x_m is $A_X = g \cos\theta$, and the component acting on y_m is $A_Y = g \sin\theta$. After applying the trig identities shown on the right, it demonstrates that the angle of rotation θ is in fact the arctangent of A_Y/A_X.



$$\frac{A_Y}{A_X} = \frac{g \sin\theta}{g \cos\theta} = \tan\theta$$

$$\tan\theta = \frac{A_Y}{A_X}$$

$$\tan^{-1}(\tan\theta) = \tan^{-1}\left(\frac{A_Y}{A_X}\right)$$

$$\theta = \tan^{-1}\left(\frac{A_Y}{A_X}\right)$$

Example Program: VertWheelRotation.bs2

This program displays your board's angle of rotation as shown in Figure 3-20 at the beginning of this activity, page 92.

- √ Enter, save, and run VertWheelRotation.bs2.
- √ Hold the board in vertically in front of you like a steering wheel.
- √ Rotate the board clockwise, and watch the angle measurement grow.
- √ Verify that the display angle ranges from 0 to 359.

```
' Smart Sensors and Applications - VertWheelRotation.bs2
' Mount accelerometer on a vertical wheel and measure
' the rotation angle.

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

angle          VAR     Word
x              VAR     Word
y              VAR     Word

DO

  PULSIN 6, 1, x
  PULSIN 7, 1, y

  x = (x MIN 1875 MAX 3125) - 1875 ** 13369 - 127
  y = (y MIN 1875 MAX 3125) - 1875 ** 13369 - 127

  angle = x ATN y
  angle = angle */ 361

  DEBUG HOME, CLREOL, SDEC ? x,
              CLREOL, SDEC ? y,
              "angle = ", CLREOL,
              DEC angle,
              176                           ' ASCII 176 is degree symbol

  PAUSE 100

LOOP
```

Your Turn - Debug Terminal Behavior

This **DEBUG** command below displays signed values of the **x** and **y** variables followed by **angle** and the degree symbol (which is the ASCII code 176). The reason **CLREOL** comes before each number is to prevent characters that don't disappear on the right of some measurements. For example, if one measurement is −105, and the next measurement is 076, it will display as 0755 if the **CLREOL** doesn't clear the previous value before displaying the new one. Although **CLS** can fix this problem too, the Debug Terminal

flicker that results is not pleasant to examine for any length of time. **CLREOL** erases to the right of the cursor on a given line. While it still causes a little bit of flicker in each value, you'll likely agree that it's much easier to look at than the **CLS** version.

```
DEBUG HOME, CLREOL, SDEC ? x,
            CLREOL, SDEC ? y,
            "angle = ", DEC3 angle,
            176                      ' ASCII 176 is degree symbol
```

- √ Save VertWheelRotation.bs2 as VertWheelDisplayTest.bs2.
- √ Replace **HOME** with **CLS** in the **DEBUG** command and run the program.
- √ Change **CLS** back to **HOME** and run the program again. Do you see an improvement in the display?
- √ Remove the **CLREOL** control characters and note the effect on the display as you rotate the board. Extra digits will appear at the end of non-negative values.
- √ Put the **CLREOL** control characters back in and run the program again. Those pesky extra digits that didn't disappear before should be gone.

Your Turn - LCD Display

The **DEBUG** command has three lines of display, and the degree symbol will need a custom character. Here is an initialization command for the LCD that does three things: (1) start the LCD, (2) display text that doesn't change, and (3) defines Custom Character 7 as the degree symbol " ° ."

```
' Initialize LCD
PAUSE 200
SEROUT 14, 84, [22, 12]
PAUSE 5

SEROUT 14, 84, [ 130, "angle = ", DEC angle, 7,
                 150, "x=", SDEC x,
                 157, "y=", SDEC y ]

SEROUT 14, 84, [255,                  ' Define Custom Character 7
                %01000,               '   *
                %10100,               ' *   *
                %01000,               '   *
                %00000,               '
                %00000,               '
                %00000,               '
                %00000,               '
                %00000]               '
```



Custom Character Definitions Remember, 248 defines Custom Character 0. 249 defines Custom Character 1. 250 defines Custom Character 2, and so on, up to 255, which defines Custom Character 7.

The **DEBUG** command that the **SEROUT** command has to replace uses three lines in the Debug Terminal. The **SEROUT** command below only uses two. To minimize LCD display flicker, only the digits are erased before the new digits are printed. The **SEROUT** places the cursor at 138 (Line 0, character 10), then overprints the previous measurement with five spaces. Then, it places the cursor at 138 again and displays the new degree measurement with **DEC** angle. Finally, it prints the degree sign with Custom Character 7. This is repeated for the x and y measurements, but there only need to be four spaces between quotes following cursor positions 152 and 159.

```
' LCD Display Routine
SEROUT 14, 84, [ 138, "     ", 138, DEC angle, 7,
                 152, "    ", 152, SDEC x,
                 159, "    ", 159, SDEC y ]
```

- √ Save VertWheelRotation.bs2 as VertWheelRotationLcd.bs2.
- √ Insert the initialize LCD routine between the variable declarations and the **DO** keyword.
- √ Replace the **DEBUG** command in the **DO**...**LOOP** with the LCD Display Routine.
- √ Change **PAUSE 100** to **PAUSE 350**.
- √ Run the program and test your LCD rotation display.

Your Turn - Rotation in the Opposite Direction

Diagrams that show the rotation angle increasing as the object rotates counterclockwise, like Figure 3-24, are quite a bit more common than the clockwise diagram we used previously.

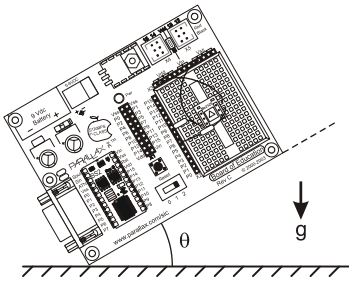


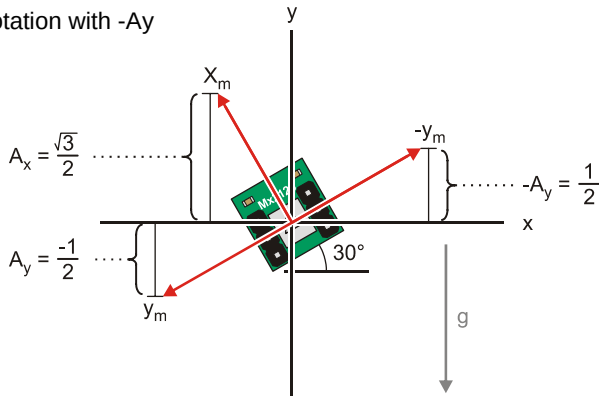
Figure 3-24
Angle Measurement with Counterclockwise Rotation

To reverse the angle of rotation the that program displays, all you have to do is use −Ay instead of A_y. Take a look at Figure 3-25. If you rotate the accelerometer counterclockwise, A_y is −1/2, and the arctangent turns out to be 330°. By taking the arctangent of $-A_y/A_x$, the result is 30°.

Figure 3-25: Reversing Direction of Rotation with -Ay

$$\tan^{-1}\left(\frac{-A_y}{A_x}\right) = \tan^{-1}\left(\frac{\frac{1}{2}}{\frac{\sqrt{3}}{-2}}\right) = 330°$$

$$\tan^{-1}\left(\frac{-Ay}{Ax}\right) = \tan^{-1}\left(\frac{\frac{1}{2}}{\frac{\sqrt{3}}{2}}\right) = 30°$$



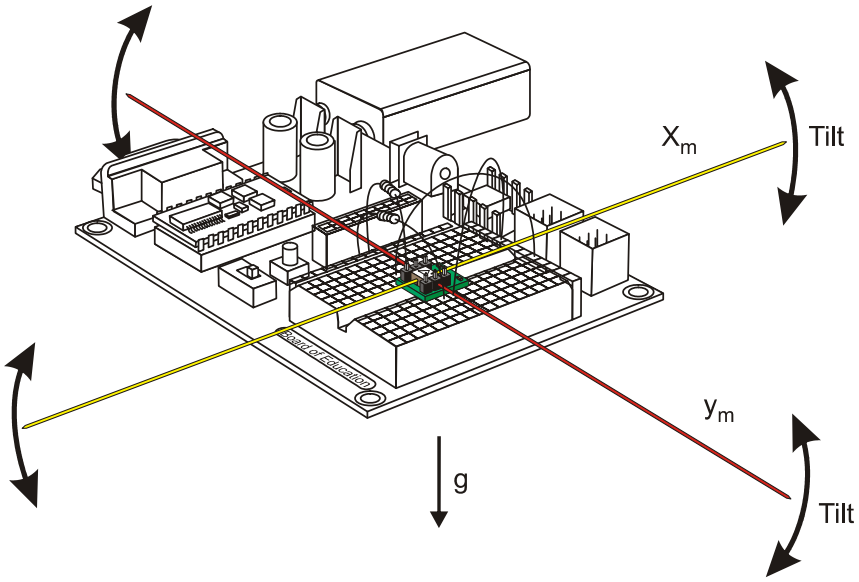
This change is easy to make in the program. Simply insert a negative sign before the y in **`angle = x ATN y`**.

- √ Save VertWheelRotation.bs2 as VertWheelRotationCounterclockwise.bs2
- √ Change **`angle = x ATN y`** to **`angle = x ATN -y`**.
- √ Run the program and verify that the rotation angle now increases as you rotate the board counterclockwise.

ACTIVITY #6: MEASURE TILT FROM THE HORIZONTAL

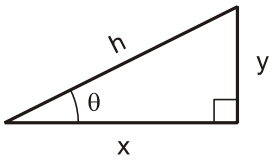
This activity measures how far the Board of Education is tilted from the horizontal. Figure 3-26 shows the Board of Education with the Memsic Accelerometer on the breadboard. The accelerometer's acceleration-sensing axes (x_m and y_m) point toward the top and left of the Board of Education. This activity develops a program that displays the tilt angle for each axis. When the board is held level, the tilt angle is 0° for both the x_m an y_m axes. If you tilt the board so that y_m points up, the program will report a positive tilt angle for the y-axis. If you tilt it so that y_m points down, it will report a negative tilt angle. The same applies for x_m; point it up for a positive tilt angle or down for a negative tilt angle. If you tilt the board towards one of its corners, the program will report tilt for both the x_m and y_m axes.

Figure 3-26: Tilt Axes on the Board of Education



Sine and Cosine

Figure 3-27 shows the relationship between the sides of a right triangle and the sine and cosine functions. The sine of an angle is the opposite side of the triangle (y) divided by the hypotenuse (h). If you know h and y, and want to know the angle (θ), use arcsine ($\sin^{-1}$). The cosine of the angle is the adjacent side (x) divided by h. If you want to know the angle given x and h, use arccosine ($\cos^{-1}$).



$$\sin\theta = \frac{y}{h}$$

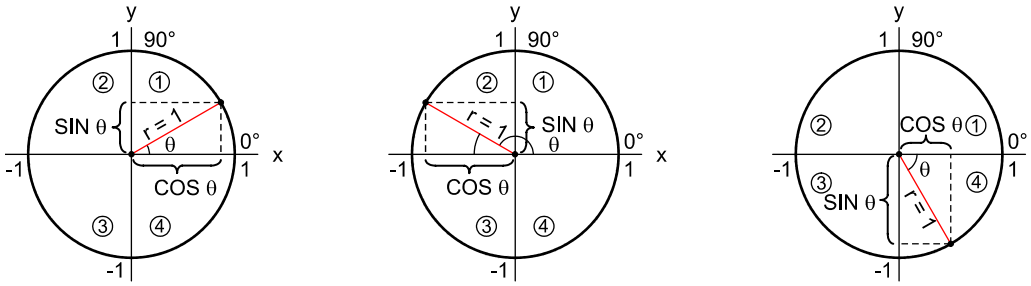
$$\cos\theta = \frac{x}{h}$$

Figure 3-27
Sine and Cosine

Note from the equations for Figure 3-27 that the x value can be at most the same as h when θ = 0°. Likewise, the y value can be at most h when θ = 90°. For angles between 0 and 90°, the ratio of x/h and y/h are both less than 1. It doesn't matter how large the triangle is, the ratio will always be between 1 and 0.

The unit circle is a common device for describing the sine and cosine functions. The triangle's hypotenuse becomes the radius of the circle. The unit circle is so named because the length of the hypotenuse is 1 (one unit). As the hypotenuse is rotated counterclockwise, the angle θ becomes larger, or smaller if it is rotated clockwise. The cosine is determined by drawing a vertical line from the point where the hypotenuse meets the circle down (or up if the hypotenuse is below) to the x-axis. Whatever the x value is, that's the cosine. The sine of the angle is determined by drawing a line from the end of the radius horizontally to the y-axis.

Figure 3-28: Unit Circle Sine and Cosine Examples



The range from 0 to 90° is the unit circle's Quadrant 1. When θ is in Quadrant 1, both the cosine and sine of the angle will be positive numbers. When θ is between 90 and 180° (Quadrant 2), the cosine becomes negative but the sine is still positive. In Quadrant 3, both sine and cosine are negative, and in Quadrant 4, the sine is still negative but cosine is positive again. Notice in Figure 3-28 (c) that a negative value of θ (between 0 and −90) can be in Quadrant 4 just as a value between 270 and 360°. One other thing to keep in mind here is that the minimum value for both sine and cosine is −1, and the maximum value is 1. For example, when θ = 0°, cos θ = 1, and sin θ = 0. If θ = 90°, sin θ = 1 and cos θ = 0. At θ = 180°, cos θ = −1 and sin θ = 0.

Figure 3-29 shows the BASIC Stamp version of a unit circle for its **SIN** and **COS** operators. Instead of results that range from −1 to 1, the results for **SIN** and **COS** range from −127 to 127. Angles for the **SIN** and **COS** operators are in terms of brads. So, instead of 45°, use 32 brads. Instead of 90°, use 64 brads, and so on. To convert from brads to degrees with a calculator, multiply the number of brads by 360/256. To convert from degrees to brads, use 256/360.

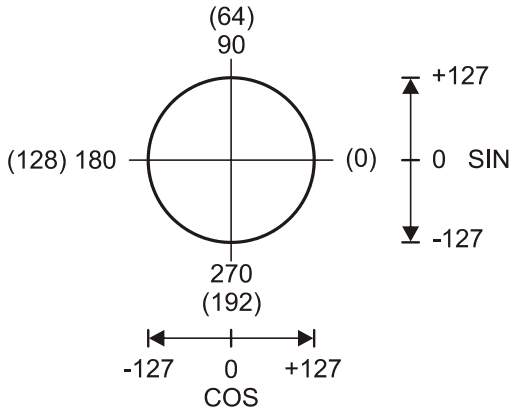


Figure 3-29
BASIC Stamp Unit Circle Sine and Cosine Operators

Example Program: SineCosine.bs2

This example program displays the BASIC Stamp integer calculations for sine and cosine. You can divide these values by 127 to get an approximation of the actual sine or cosine values. It converts degrees to brads with **** 46733**, which was derived using the ScaleConstant equation from Activity #3.

- √ Enter, save and run SineCosine.bs2
- √ Compare the results (divided by 127) to calculated values of sine and cosine.

```
' Smart Sensors and Applications - SineCosine.bs2
' Display BASIC Stamp sine and cosine values.

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

degrees VAR Word
brads VAR Word
```

```
sine  VAR Word
cosine VAR Word

DEBUG "Degrees  Brads  Cosine  Sine", CR

FOR degrees = 0 TO 359 STEP 15

  brads = degrees ** 46733
  sine = SIN brads
  cosine = COS brads
  DEBUG "  ",
        SDEC3 degrees, "     ",
        SDEC3 brads, "    ",
        SDEC3 cosine, "    ",
        SDEC3 sine, CR

NEXT

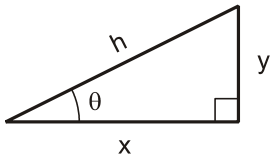
END
```

Your Turn - Program Modifications

- √ Try modifying the **FOR**...**NEXT** loop's **STEP** argument to get different values.
- √ Try modifying the program so that it prompts you for a degree value with the **DEBUGIN** command and then displays the result.

Arcsine and Arccosine Subroutines

While the sine is a ratio of y/h for a given angle, arcsine ($\sin^{-1}$) is the inverse, as you can see in Figure 3-30. Given the ratio y/h, arcsine tells you the angle. Likewise, cosine is the ratio of x/h for a given angle, and arccosine ($\cos^{-1}$) is the angle for a given ratio of x/h.



$$\sin\theta = \frac{y}{h} \quad \theta = \sin^{-1}\frac{y}{h}$$

$$\cos\theta = \frac{x}{h} \quad \theta = \cos^{-1}\frac{x}{h}$$

Figure 3-30
Sine, Arcsine, Cosine, and Arccosine

While the BASIC Stamp does not have ASIN and ACOS operators, Tracy Allen, author of the Stamps in Class text *Applied Sensors*, published some very nice subroutines that

perform these functions on his web site www.emesystems.com. The next example program uses modified versions of these subroutines.

Remember that the **SIN** and **COS** operators return values between −127 and 127. If you divide the result by 127, you'll get a value between −1 and 1 that is an approximation of the actual sine (y/h) or cosine (x/h) ratios. With the **Arcsine** and **Arccosine** subroutines, you can set a variable named **side** to a value between −127 and 127, and the subroutine will store the degree measurement results in the **angle** variable.



If you want the Arcsine and Arccosine subroutines to return brads instead of degrees, just make three changes:

In the **Arccosine** subroutine, comment the line of code that converts from brads to degrees:

```
'  angle = angle */ 361    ' Convert brads to degrees
```

Then, in the **IF...THEN** statement change **180** to **128** because we are now using a 256-division circle:

```
IF sign = Negative THEN angle = 128 - angle
```

Likewise, in the **Arccosine** subroutine change **90** to **64:**

```
angle = 64 - angle
```

Example Program: TestArcsine.bs2

This next program sweeps sine values from −127 to 127, and its **Arcsine** subroutine converts these sine values back to degree angles. Keep in mind that this is the reverse of the calculations in the previous example program. The previous example program displayed sine values for given angles. This one displays angles for given sine values.

- √ Enter, save, and run TestArcsine.bs2
- √ Compare the results to the sine values calculated in the previous example program.

```
' -----[ Title ]-------------------------------------------------------------
' Smart Sensors and Applications - TestArcsine.bs2
' Test arcsine for sine values from -127 to 127.

' {$STAMP BS2}                          ' BASIC Stamp Directive
' {$PBASIC 2.5}                         ' PBASIC Directive
```

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

Negative       CON     1               ' Sign - .bit15 of Word variables
Positive       CON     0

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

sine           VAR     Word            ' sine in circle r = 127
side           VAR     Word            ' trig subroutine variable
angle          VAR     Word            ' result angle - degrees
sign           VAR     Bit             ' Sign bit

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

DEBUG CLS                              ' Clear Debug Terminal
sine = -128                            ' Start y at -128

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

DO UNTIL sine = 127                    ' Sweep from y = -127 to y = 127
  sine = sine + 1                      ' Increment by 1
  side = sine                          ' Set side to y
  DEBUG "sine = ", SDEC sine, "  "     ' Display sine value
  GOSUB Arcsine                        ' Calculate arcsine
  DEBUG SDEC ? angle                   ' Display the result angle
LOOP                                   ' Repeat DO...LOOP

END                                    ' End program

' -----[ Subroutine - Arcsine ]---------------------------------------------

' This subroutine calculates arcsine based on the y coordinate on a circle
' of radius 127.  Set the side variable equal to your y coordinate before
' calling this subroutine.

Arcsine:                               ' Inverse sine subroutine
  GOSUB Arccosine                      ' Get inverse cosine
  angle = 90 - angle                   ' sin(angle) = cos(90 - angle)
  RETURN

' -----[ Subroutine - Arccosine ]-------------------------------------------

' This subroutine calculates arccosine based on the x coordinate on a circle
' of radius 127.  Set the side variable equal to your x coordinate before
' calling this subroutine.

Arccosine:                                 ' Inverse cosine subroutine
  sign = side.BIT15                        ' Save sign of side
  side = ABS(side)                         ' Evaluate positive side
  angle = 63 - (side / 2)                  ' Initial angle approximation
  DO                                       ' Successive approximation loop
```

```
  IF (COS angle <= side) THEN EXIT         ' Done when COS angle <= side
  angle = angle + 1                        ' Keep increasing angle
LOOP
angle = angle */ 361                       ' Convert brads to degrees
IF sign = Negative THEN angle = 180 - angle' Adjust if sign is negative.
RETURN
```


Your Turn - Testing the Arccosine Subroutine

Here are some modifications you can make to TestArcsine.bs2 to make it test the **`Arccosine`** subroutine instead.

- √ Save TestArcsine.bs2 as TestArccosine.bs2.
- √ Update the comments in the title section. Cosine values will be swept from 127 to −127.
- √ Change **`sine VAR Word`** to **`cosine VAR Word`** in the Variables section.
- √ Change **`sine = −128`** to **`cosine = 128`** in the Initialization section
- √ Modify the Main Routine so that it looks like this

```
DO UNTIL cosine = -127
  cosine = cosine - 1
  side = cosine
  DEBUG "cosine = ", SDEC cosine, "  "
  GOSUB Arccosine
  DEBUG SDEC ? angle
LOOP

END
```

- √ Run the modified test program. As **`cosine`** sweeps from 127 to -127, the angle should sweep from 0 to 180°.

Displaying Accelerometer Tilt Angle

Figure 3-31 shows the Board of Education with a Memsic Accelerometer. The figure also shows a close-up of the accelerometer module and its x_m and y_m acceleration sensing axes. These sensing axes detect components of the earth's acceleration due to gravity. As you tilt a given axis toward vertical, larger components of the earth's 1 g act on the axis.

Figure 3-31: Tilting the Board of Education, Tilting the Memsic Accelerometer

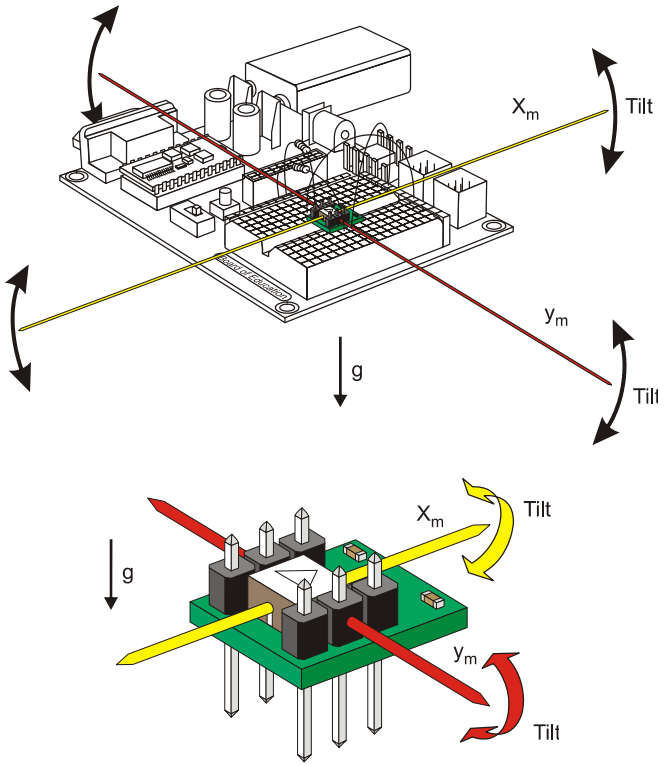


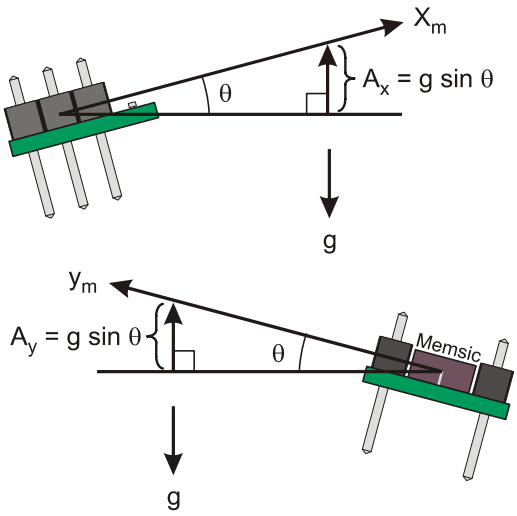
Figure 3-32 shows how arcsine can be used to determine the tilt angle. Looking at the Memsic Accelerometer Module from the side, the component of gravity acting on its x_m is the x-axis acceleration (A_x), which is $g \times \sin \theta$. Since $\sin \theta$ is equal to A_x / g, θ_x can be determined by taking the arcsine of A_x / g. In terms of an equation, that's :

$$\theta_X = \sin^{-1}\left(\frac{A_X}{g}\right)$$

The same principle applies to the accelerometer's y_m axis, and the result is:

$$\theta_X = \sin^{-1}\left(\frac{A_Y}{g}\right)$$

Figure 3-32: Determining Tilt Angle with Arcsine



$$A_X = g \sin\theta$$

$$\frac{A_X}{g} = \sin\theta$$

$$\sin\theta = \frac{A_X}{g}$$

$$\sin^{-1}(\sin\theta) = \sin^{-1}\left(\frac{A_X}{g}\right)$$

$$\theta = \sin^{-1}\left(\frac{A_X}{g}\right)$$

With the MX2125, a measurement of 1875 is −1 g, and a measurement of 3125 is 1 g. In Activity #3, we scaled this to a range of −127 to 127. Remember that −127 is the equivalent of −1 for the **`Arcsine`** subroutine, and 127 is the equivalent of 1. Anything between −127 and 127 is the equivalent of a fraction, and coming from the MX2125, it's actually sin θ. So, once the MX2125's measurement has been scaled to −127 to 127, all you have to do is use the **`Arcsine`** subroutine to determine the tilt angle (the value of θ).

The simplest way to write a tilt program is to start with the previous example program, TestArcsine.bs2. Then, incorporate the accelerometer measurement and scaling and offset commands from TestScaleOffset.bs2 and the accelerometer measurements from VertWheelRotation.bs2. This program's Main Routine boils down to two commands for measuring the x and y axes, two commands for scaling, and two small routines that call the **`Arcsine`** subroutine and display the result.

```
DO

  PULSIN 6, 1, x
  PULSIN 7, 1, y

  x = (x MIN 1875 MAX 3125) - 1875 ** 13369 - 127
  y = (y MIN 1875 MAX 3125) - 1875 ** 13369 - 127
```

```
          side = x
          GOSUB Arcsine
          DEBUG HOME, "x tilt angle = ", CLREOL, SDEC3 angle, CR

          side = y
          GOSUB Arcsine
          DEBUG "y tilt angle = ", CLREOL, SDEC3 angle

          PAUSE 100

        LOOP
```

Example Program: HorizontalTilt.bs2

This example program displays your board's tilt in terms of degrees from horizontal.

- √ Enter, save and run HorizontalTilt.bs2.
- √ Compare various tilt angles to the Debug Terminal's axis display.

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - HorizontalTilt.bs2
' Test arcsine for sine values from -127 to 127.

' {$STAMP BS2}                              ' BASIC Stamp Directive
' {$PBASIC 2.5}                             ' PBASIC Directive

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

Negative       CON     1                   ' Sign - .bit15 of Word variables
Positive       CON     0

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

x              VAR     Word                ' Memsic x-axis measurement
y              VAR     Word                ' Memsic y-axis measurement

side           VAR     Word                ' trig subroutine variable
angle          VAR     Word                ' result angle - degrees
sign           VAR     Bit                 ' Sign bit

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

DEBUG CLS                                  ' Clear Debug Terminal

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

DO

  PULSIN 6, 1, x                           ' x-axis measurement
```

```
  PULSIN 7, 1, y                              ' y-axis measurement

  ' Scale and offset x and y-axis values to -127 to 127.
  x = (x MIN 1875 MAX 3125) - 1875 ** 13369 - 127
  y = (y MIN 1875 MAX 3125) - 1875 ** 13369 - 127

  ' Calculate and display Arcsine of x-axis measurement.
  side = x
  GOSUB Arcsine
  DEBUG HOME, "x tilt angle = ", CLREOL, SDEC3 angle, CR

  ' Calculate and display Arcsine of y-axis measurement.
  side = y
  GOSUB Arcsine
  DEBUG "y tilt angle = ", CLREOL, SDEC3 angle

  PAUSE 100                                   ' Pause 1/10 second

LOOP                                          ' Repeat DO...LOOP

' -----[ Subroutine - Arcsine ]-------------------------------------------

' This subroutine calculates arcsine based on the y coordinate on a circle
' of radius 127.  Set the side variable equal to your y coordinate before
' calling this subroutine.

Arcsine:                                      ' Inverse sine subroutine
  GOSUB Arccosine                             ' Get inverse cosine
  angle = 90 - angle                          ' sin(angle) = cos(90 - angle)
  RETURN

' -----[ Subroutine - Arccosine ]-----------------------------------------

' This subroutine calculates arccosine based on the x coordinate on a circle
' of radius 127.  Set the side variable equal to your x coordinate before
' calling this subroutine.

Arccosine:                                    ' Inverse cosine subroutine
  sign = side.BIT15                           ' Save sign of side
  side = ABS(side)                            ' Evaluate positive side
  angle = 63 - (side / 2)                     ' Initial angle approximation
  DO                                          ' Successive approximation loop
    IF (COS angle <= side) THEN EXIT          ' Done when COS angle <= side
    angle = angle + 1                         ' Keep increasing angle
  LOOP
  angle = angle */ 361                        ' Convert brads to degrees
  IF sign = Negative THEN angle = 180 - angle' Adjust if sign is negative.
  RETURN
```

Your Turn - LCD Display

Modifying the example program to display the tilt measurements on the Parallax Serial LCD is still a matter of adding an Initialization routine and porting **DEBUG** commands to **SEROUT** commands. As with the program from Activity #5, this program displays characters that don't change in the Initialization routine to prevent display flicker.

- √ Save HorizontalTilt.bs2 as HorizontalTiltLcd.bs2
- √ Replace the **DEBUG** command in the Initialization routine with this.

```
' Initialize LCD
PAUSE 200
SEROUT 14, 84, [22, 12]
PAUSE 5

SEROUT 14, 84, [128, "x-tilt=",
                148, "y-tilt="]

SEROUT 14, 84, [255,                  ' Define Custom Character 7
                %01000,               '   *
                %10100,               ' *   *
                %01000,               '   *
                %00000,               '
                %00000,               '
                %00000,               '
                %00000,               '
                %00000]               '
```

- √ Replace the first **DEBUG** command in the Main Routine's **DO**...**LOOP** with the **SEROUT** command below. Make sure there are four spaces between the quotation marks. The four spaces are needed to erase the maximum of four characters that the command might send to the LCD: a negative sign, two digits, and the Custom Character 7 degree symbol.

```
SEROUT 14, 84, [135, "    ", 135, SDEC angle, 7]
```

- √ Replace the second **DEBUG** command in the Main Routine's **DO**...**LOOP** with this. Again, make sure to put four spaces between the quotation marks to erase the previous value.

```
SEROUT 14, 84, [155, "    ", 155, SDEC angle, 7]
```

- √ Change **PAUSE 100** to **PAUSE 350**.
- √ Run the program and test the display.

Your Turn - Adjustments

If your display did not go all the way to 90° when you held your board with a particular axis vertical, you can customize your scaling and offset to get it to fit. This will involve determining your accelerometer's actual output scale. If it's really 1865 to 3100, repeat the steps in Activity #3 to make the scaling and offset corrections.

SUMMARY

This chapter focused on sensing the acceleration due to gravity with the Memsic 2125 Dual Axis Accelerometer. Sensing gravity makes it possible to measure both tilt and rotation. The Memsic Accelerometer transmits pulses that indicate the acceleration acting on its x and y axes. At room temperature, the pulses range from 3750 to 6250 µs, which can be used to measure a range of −1 to 1 g with either one of the accelerometer's two sensing axes. The **PULSIN** command is used to measure these pulses, and since it measures time in 2 µs units, the range which programs have to examine is 1875 to 3125.

Accelerometer measurements can be displayed with the Parallax Serial LCD. If the program has already been tested with the Debug Terminal, displaying measurements with the serial LCD is typically a matter of adding an LCD initialization routine to the beginning of the program and using **SEROUT** commands in place of **DEBUG** commands. Custom characters come in handy for displaying the degree symbol (°), and the Greek letter mu (µ).

The accelerometer can be used to measure rotation in the vertical plane. To do this, the BASIC Stamp must calculate the arctangent of the accelerometer's y-axis measurement, divided by its x-axis measurement. The x and y axis measurements have to be scaled and offset to fit in a range of −127 to 127, which is what the PBASIC **ATN** operator needs to return an angle, measured in binary radians. While degrees separate a circle into 360 segments, binary radians separate it into 256 segments. The PBASIC ***/** operator can be used to convert a given binary radian measurement to degrees.

The accelerometer can also be used to measure tilt angles. Since the component of gravity acting on each of the accelerometer's sensing axes is the sine of the tilt angle, the inverse sine or arcsine can be used on an axis' measurement to determine the tilt angle. An Arcsine subroutine can be used to calculate the angle (in degrees) given a value that ranges from −127 to 127. This range corresponds to sine values of −1 to + 1.

Since both the **ATN** operator and the **Arcsine** subroutine expect a value between −127 and 127, techniques for scaling and offsetting the accelerometer measurements were introduced. The range of measurements the BASIC Stamp collects from the accelerometer are on a scale of 1875 to 3125. The most efficient way to scale these values to a range of −127 to 127 involves subtracting 1875 to zero-align the range, then using the ****** operator to reduce the scale, then subtracting 127. This is the resulting line

of code: **`value = (value MIN 1875 MAX 3125) - 1875 ** 13369 - 127`**. The value 13369 is determined by the **`**`** scale constant equation in Activity #2.

Questions

1. What are seven quantities you can measure with an accelerometer?
2. What does MEMS stand for?
3. What moves inside the MX2125 when you tilt it?
4. Can gravity be considered a form of acceleration?
5. What do you have to do to a program that displays measurements in the Debug Terminal to make it display measurements in a serial LCD instead?
6. How can you restrict a variable to a range of values?
7. How can you orient your board to apply 1 g to the accelerometer's x-axis?
8. How can you orient your board to apply 0 g to both axes?
9. What's the difference between a binary radian and a degree?
10. What range of values do the **`SIN`** and **`COS`** operators accept? What do these values represent?
11. How can you convert from brads to degrees?
12. What range of values does the **`ATN`** operator accept? What do these values represent?
13. Why can you use **`ATN`** to calculate your board's angle of rotation?
14. What range of values is the **`Arccosine`** subroutine designed to accept? What do these values represent?
15. What range of values is the **`Arcsine`** subroutine designed to accept? What do these values represent?
16. Why is it necessary to use the **`Arcsine`** subroutine to determine tilt angle?

Exercises

1. Write a command that receives the acceleration measurement from the accelerometer's y-axis output pin connected to P10.
2. Write a command that receives the acceleration measurement from the accelerometer's x-axis output pin connected to P9.
3. Write a command that converts the x-axis measurement to microseconds.
4. Write a command that converts the x-axis measurement to milliseconds.
5. Write a line of PBASIC code that scales a range from 0 to 100 to a range of 20 to 32.

Projects

1. Design a device that counts the number of times you rotate your board on the vertical plane. Assume you are starting at 0˚.
2. Design a device that displays an alarm message every time it has been tilted beyond 10˚ from the horizontal.

Solutions

Q1. Acceleration, tilt and tilt angle, incline, rotation, vibration, collision, gravity.
Q2. Micro electro-mechanical systems.
Q3. A bubble of heated gas.
Q4. Yes, either static or dynamic.
Q5. Add an initialization routine for the LCD, and convert the **DEBUG** commands to **SEROUT** commands.
Q6. Use the **MAX** and **MIN** operators.
Q7. Tilt it up on its longer edge, with the servo ports up. (As in Figure 3-14a).
Q8. Place it flat on a table.
Q9. The degree splits a circle into 360 units, whereas a binary radian splits a circle into 256 units.
Q10. 0 to 255. They represent the angle, in brads (binary radians).
Q11. Degrees = brads * 360 / 256.
Q12. -127 to +127, which represents the opposite and adjacent sides of the triangle.
Q13. Since the accelerometer will measure the acceleration acting on the Memsic's ym axis, as well as that along it's xm axis, the **ATN** of Ay/Ax can be used to find the angle of rotation from the vertical plane, along which g is acting.
Q14. From -127 to 127, which represents the length of the x side of the triangle.
Q15. From -127 to 127, which represents the length of the y side of the triangle.
Q16. We know from geometry that the component of gravity acting on the accelerometer is g sin θ, so to get the angle we must take the arcsine.

E1.
```
y VAR Word
PULSIN 10, 1, y
```

E2.
```
x VAR Word
PULSIN 9, 1, x
```

E3.
```
x = x * 2
```

E4.
```
x = x * 2 / 1000
```
-OR-
```
x = x /500
```

E5.
```
value = (value MIN 0 MAX 100) ** 8519 + 20
```

P1. Example solution:

```
' Smart Sensors and Applications - Ch3Proj1.bs2
' Based on VertWheelRotation.bs2, this device counts the number
' of times the board has been rotated on the vertical plane.

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

angle          VAR     Word
angleOld       VAR     Word
x              VAR     Word
y              VAR     Word
turnCount      VAR     Word

PAUSE 250                                      ' Initialize LCD
SEROUT 14, 84, [22, 12]
PAUSE 5

SEROUT 14, 84, [128, DEC5 turnCount]

DO

  PULSIN 6, 1, x
  PULSIN 7, 1, y

  x = (x MIN 1875 MAX 3125) - 1875 ** 13369 - 127
  y = (y MIN 1875 MAX 3125) - 1875 ** 13369 - 127

  angle = x ATN y
  angle = angle */ 361

  IF (angle >= 90 AND angle < 180) AND (angleOld < 90 OR angleOld >= 270) THEN
    turnCount = turnCount + 1
    angleOld = angle
  ENDIF

  IF angle >= 270 AND (angleOld >= 90 AND angleOld < 180) THEN
    turnCount = turnCount + 1
    angleOld = angle
  ENDIF

  SEROUT 14, 84, [128, DEC5 (turnCount / 2)]

LOOP
```

P2. Example solution: Below is a modified main routine from HorizontalTilt.bs2

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

DO

  PULSIN 6, 1, x                            ' x-axis measurement
  PULSIN 7, 1, y                            ' y-axis measurement

  ' Scale and offset x and y-axis values to -127 to 127.
  x = (x MIN 1875 MAX 3125) - 1875 ** 13369 - 127
  y = (y MIN 1875 MAX 3125) - 1875 ** 13369 - 127

  ' Calculate and display Arcsine of x-axis measurement.
  side = x
  GOSUB Arcsine
  DEBUG HOME, "x tilt angle = ", CLREOL, SDEC3 angle, CR

  IF ABS(angle) > 10 THEN
    DEBUG CRSRXY, 0, 2, "Warning! Check x-axis!"
  ELSE
    DEBUG CRSRXY, 0, 2, CLREOL
  ENDIF

  ' Calculate and display Arcsine of y-axis measurement.
  side = y
  GOSUB Arcsine
  DEBUG CRSRXY, 0, 1, "y tilt angle = ", CLREOL, SDEC3 angle

  IF ABS(angle) > 10 THEN
    DEBUG CRSRXY, 0, 3, "Warning! Check y-axis!"
  ELSE
    DEBUG CRSRXY, 0, 3, CLREOL
  ENDIF

  PAUSE 100                                 ' Pause 1/10 second

LOOP                                        ' Repeat DO...LOOP
```

Chapter 4: The Hitachi HM55B Compass Module

The Hitachi HM55B Compass module measures direction. You can use it along with your BASIC Stamp, Board of Education, and Parallax Serial LCD to make a digital compass that works as shown in Figure 4-1. The module's Hitachi HM55B chip is an increasingly common feature in automobile electronics, providing a compass heading for the driver. The compass module is also a great tool for mobile robots, giving them a sense of direction which can make a tremendous difference in robot team sports as well as mazes.

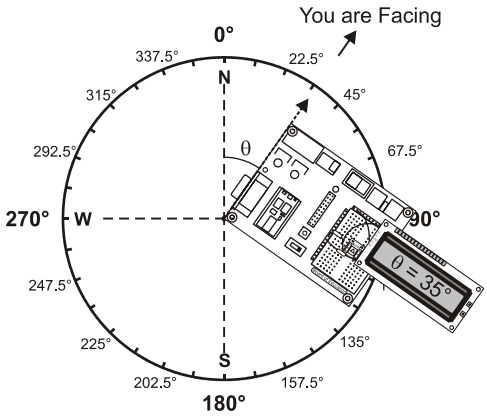


Figure 4-1
Hitachi Compass Module on the Board of Education with an LCD Display

This chapter uses modified versions of the programs from the Hitachi HM55B Compass Module product documentation for testing and calibration. It also introduces averaging as a way to filter measurement noise and demonstrates how to modify the existing example programs to display the compass heading on the Parallax Serial LCD.

INTERPRETING THE COMPASS MEASUREMENTS

The Hitachi HM55B Compass Module product documentation has example programs that all use a subroutine named **`Compass_Get_Axes`** that returns x and y magnetic field strength measurements. The value of x is the component of the earth's magnetic field acting on the sensor's $x_m$axis shown in Figure 4-2. The value of y is the negative of the earth's magnetic field acting on the y_m axis. If N is the value reported by x or y when it is aligned with the earth's magnetic field, then the x measurement at some angle θ will be N cos θ, and the y measurement will be −N sin θ. Using these facts and a couple of

trigonometry identities, it turns out that the angle θ is the arctangent of −y/x. So in addition to accelerometer rotation, the compass module's angle from north is another value that can be determined using the PBASIC **ATN** operator.

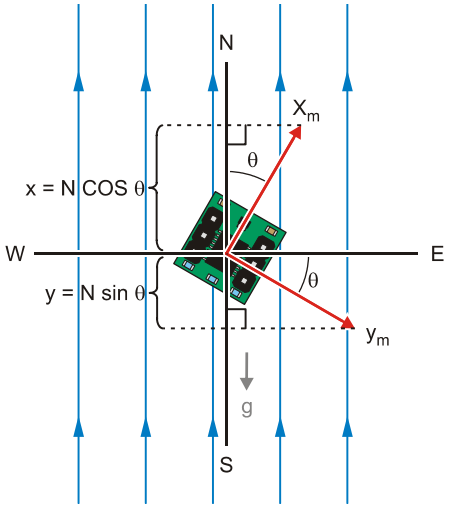


Figure 4-2
Compass Module Sensing Axes

$$\tan\theta = \frac{-N\sin\theta}{N\cos\theta} = \frac{-y}{x}$$

$$\tan^{-1}(\tan\theta) = \tan^{-1}\left(\frac{-y}{x}\right)$$

$$\theta = \tan^{-1}\left(\frac{-y}{x}\right)$$

ACTIVITY #1: CONNECTING AND TESTING THE COMPASS MODULE

In this activity, you will connect the compass module to the BASIC Stamp and run a test program. This will verify that the electrical connections are correct and the module is in working order.

Connecting the Compass Module

The Hitachi HM55B Compass module needs connections to Vdd and Vss (power and ground) and three communication line connections to the BASIC Stamp.

Parts Required

(1) Hitachi HM55B Compass Module
(6) Jumper wires

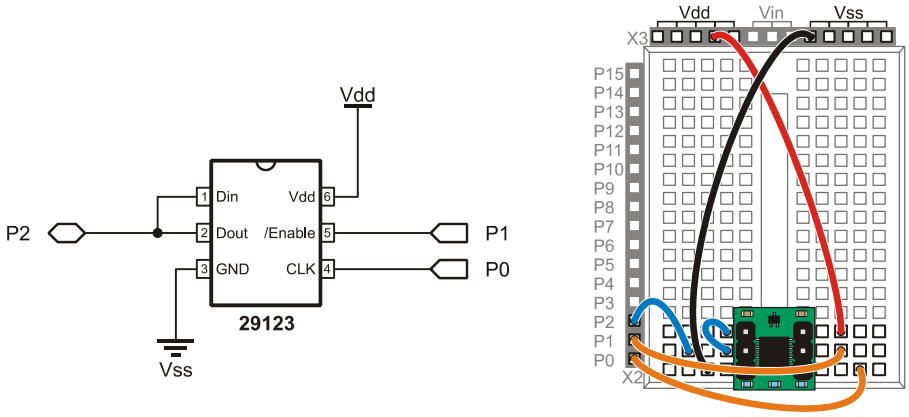
There are no external resistors or capacitors required; they are all built onto the module.

Schematic and Wiring Diagram

The HM55B can be connected with its Dout and Din pins tied together so that they transmit and receive signals to and from the same BASIC Stamp I/O pin. Another BASIC Stamp I/O pin is connected to the device's clock (CLK) pin. The BASIC Stamp will send pulses to this pin as it makes the chip send its status or measurements or receive commands. The BASIC Stamp also sends low signals to the Compass Module's /Enable pin before it exchanges any data, and also to initialize each magnetic field measurement.

√ Build the circuit shown in Figure 4-3.

Figure 4-3: Compass Module Schematic and Wiring Diagram

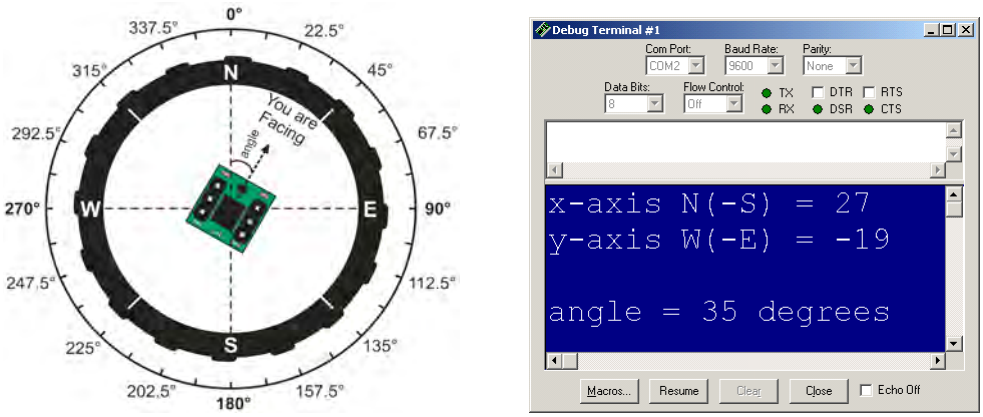


Testing the Compass Module

This example program tests to make sure the compass module is connected properly and in working order. There may be sizeable differences between the magnetic north reported by a mechanical compass and the one reported by the compass module. After the calibration programs in upcoming activities, all apparent measurement errors should disappear.

Figure 4-4 shows what the compass should display when it detects that it is facing 35° clockwise of north. Again, don't worry about exact direction at this point because the program is only testing to make sure the module is working. So long as you can use it to get a general idea of north, south, east and west, it's in working order.

Figure 4-4: Debug Terminal Output with Compass Facing 35° Clockwise of North



Example Program: TestCompass.bs2

Free Download! This program is available as a free .bs2 file download from the Smart Sensors and Applications Product Page at www.parallax.com.

- √ Download and unzip the selected source code from the Smart Sensors and Applications product page at www.parallax.com.
- √ Open the TestCompass.bs2 file with the BASIC Stamp Editor and run the program.
- √ The Debug Terminal should display the compass x and y axis measurements and the angle it is facing, clockwise from north.
- √ If your compass reports measurements with less than a 40° error, it means it's working and ready for the calibration program featured in Activity #2.

```
' -----[ Title ]-------------------------------------------------------------
' Smart Sensors and Applications - TestCompass.bs2
' Test to make sure Hitachi HM55B Compass Module is working.

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

' -----[ I/O Definitions ]-----------------------------------------------------
```

```
DinDout         PIN      2                    ' P2 transceives to/from Din/Dout
Clk             PIN      0                    ' P0 sends pulses to HM55B's Clk
En              PIN      1                    ' P2 controls HM55B's /EN(ABLE)

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

Reset           CON      %0000                ' Reset command for HM55B
Measure         CON      %1000                ' Start measurement command
Report          CON      %1100                ' Get status/axis values command
Ready           CON      %1100                ' 11 -> Done, 00 -> no errors
NegMask         CON      %1111100000000000    ' For 11-bit negative to 16-bits

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

x               VAR      Word                 ' x-axis data
y               VAR      Word                 ' y-axis data
status          VAR      Nib                  ' Status flags
angle           VAR      Word                 ' Store angle measurement

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

DO                                            ' Main loop

  GOSUB Compass_Get_Axes                      ' Get x, and y values

  angle = x ATN -y                            ' Convert x and y to brads
  angle = angle */ 361                        ' Convert brads to degrees

  DEBUG HOME, "x-axis N(-S) = ",SDEC x,       ' Display axes and degrees
        CLREOL, CR, "y-axis W(-E) = ",
        SDEC y, CLREOL, CR, CR, "angle = ",
        DEC angle, " degrees", CLREOL

  PAUSE 150                                   ' Debug delay for slower PCs

LOOP                                          ' Repeat main loop

' -----[ Subroutine - Compass_Get_Axes ]-----------------------------------

Compass_Get_Axes:                             ' Compass module subroutine

  HIGH En: LOW En                             ' Send reset command to HM55B
  SHIFTOUT DinDout,clk,MSBFIRST,[Reset\4]

  HIGH En: LOW En                             ' HM55B start measurement command
  SHIFTOUT DinDout,clk,MSBFIRST,[Measure\4]
  status = 0                                  ' Clear previous status flags

  DO                                          ' Status flag checking loop
    HIGH En: LOW En                           ' Measurement status command
    SHIFTOUT DinDout,clk,MSBFIRST,[Report\4]
```

```
  SHIFTIN  DinDout,clk,MSBPOST,[Status\4]  ' Get Status
LOOP UNTIL status = Ready                  ' Exit loop when status is ready

SHIFTIN  DinDout,clk,MSBPOST,[x\11,y\11]   ' Get x & y axis values
HIGH En                                    ' Disable module

IF (y.BIT10 = 1) THEN y = y | NegMask      ' Store 11-bits as signed word
IF (x.BIT10 = 1) THEN x = x | NegMask      ' Repeat for other axis

RETURN
```

Your Turn - Experiments with Magnetic Fields

There aren't all that many places where the earth's magnetic field is parallel to the ground. It's either pointing into or up from the ground. The angle at which the Earth's magnetic field points into or out of the ground is called inclination.

- √ Hold your board level, and align your compass module's x-axis with magnetic north. When the x-axis is aligned with north, the Debug Terminal should display the largest x value, and the angle should read 0 degrees.
- √ Keep pointing your compass north, but try tilting it up and down. Chances are you'll find an even larger measurement at a certain tilt than you do while holding it level. That's because the magnetic field is either pointing into, or up from, the ground in your locale.
- √ Make a note of the very largest x-axis measurement you were able to achieve.

Declination is the measure of degrees between magnetic north and true north. For the United States, you can find information on the difference at http://nationalatlas.gov. At the time of this writing, an article with information about both inclination and a map of declinations was at this page:

http://nationalatlas.gov/articles/geology/a_geomag.html

One of the maps on this page also shows the total magnetic field intensity in nanotesla measurements. The tesla (T) is a measurement of magnetic field intensity, and nanoteslas (nT) are billionths of teslas. The readings the compass module's x and y axes return are in approximately millionths of teslas (µT). According to the HM55B's chip datasheet, your compass module’s units could be anywhere from 1 to 1.6 µT.

√ Find a total magnetic field intensity map that shows your locale, and then use it to calculate the x-axis magnetic field intensity units for your compass module. If the total magnetic field intensity was listed in nanoteslas, then your result will be in nanoteslas per x-axis unit. To convert to microteslas, divide your result by 1000.

$$\text{x axis units} = \frac{\text{total magnetic field intensity}}{\text{x axis reading}}$$

The compass module can also sense magnetic fields from magnets, but magnets can also damage the sensor! BE CAREFUL!



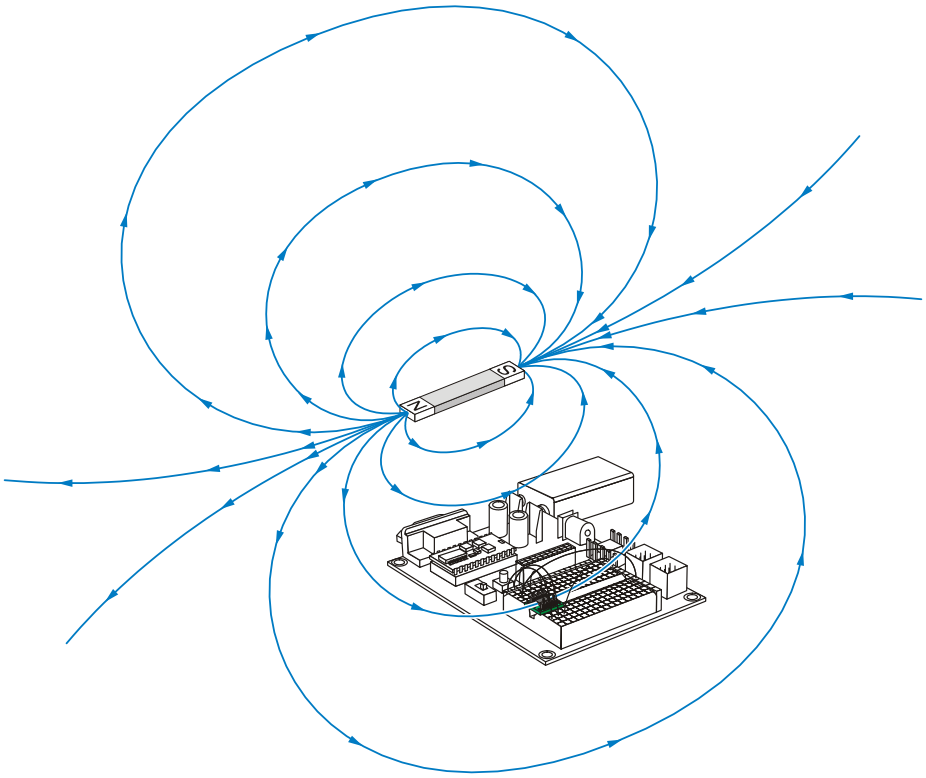
Do not place powerful magnets close to the compass module!

Keep bar, horseshoe, and electro magnets well away from your compass module until you have determined a safe distance using the procedure below. Make sure not to ever place them close enough to cause x or y-axis readings larger than ±300, because it could damage the module.

√ Start by setting your board on a table and lining up its x-axis with magnetic north.

√ Hold a bar magnet above the compass module with its S pole pointing north and its N pole pointing south as shown in Figure 4-5. Start from 1 m above, and lower it until the Debug Terminal reports an x-axis reading of 120.

Figure 4-5: Bar Magnet's Field above the Compass Module



√ Keep the bar magnet horizontal at the same height, and rotate it so that its N and S poles are no longer aligned with the earth's magnetic north and south. As you rotate it, the bar magnet's rotation should be pretty easy to track with the Debug Terminal.

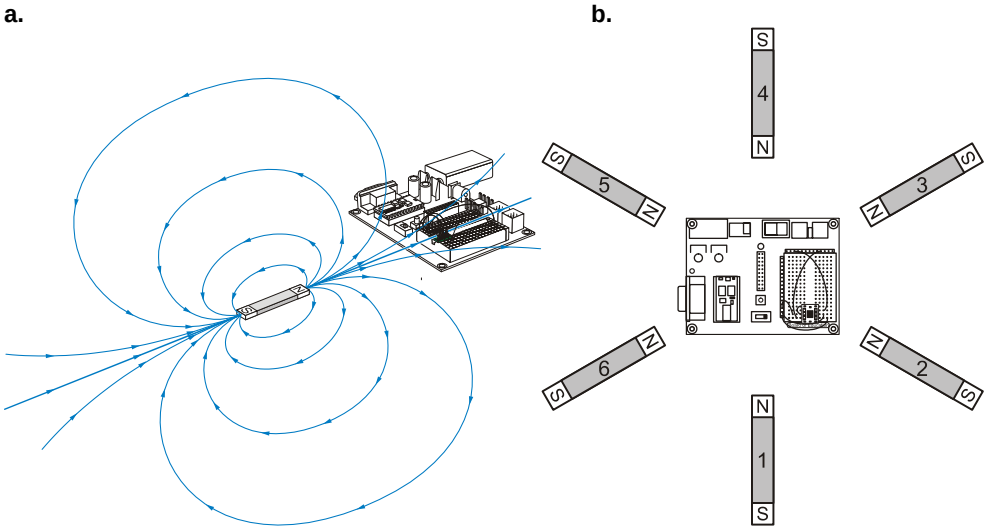
Note how the magnetic field acting on the compass module was the opposite of what the poles on the bar magnet showed. That's because of the way the magnetic field wraps around from the bar magnet's north to south poles. Figure 4-5 illustrates this with magnetic field lines that show the magnetic field's pattern around a bar magnet.

You can also hold the bar magnet at the same level with the compass module, directly in front of it, as shown in Figure 4-6a. This time the poles magnet's poles are lined up with north and south instead of the opposite.

- √ With the bar magnet oriented in Position 1 as shown in Figure 4-6b and your board oriented to north, find a distance that causes an x measurement of 120. Start from 1 m away again.
- √ Next, try placing the bar magnet in positions 2 through 6. Can you use the Debug Terminal to determine where the bar magnet is?

Figure 4-6: Finding the Safe Distance Limits

a. **b.**





Measurements between 127 and 300

Remember that the **ATN** command inputs can range from −127 to 127. If you hold the bar magnet close enough to the compass module so that it causes measurements above 127, you will need to scale the measurements down before using the **ATN** command. The scaling procedure introduced in Chapter 3, Activity #3 will work well for this.

The bar magnet in a mechanical compass will have a similar effect. It's not a very strong magnet, so there probably won't be a problem with getting it too close to the compass

module. With a mechanical compass, its bar magnet automatically lines up with north, so you will instead have to move the compass module around the mechanical compass.

√ Try it, and note how much distortion a nearby mechanical compass causes in the compass module's measurements.

With this lesson in mind, make sure to keep mechanical compasses well away from the compass module while performing and testing the calibrations in the next two activities.

ACTIVITY #2: COMPASS MODULE CALIBRATION

The calibration process involves pointing the compass module to known correct directions as the calibration program is running. The calibration program will record the values reported by the compass module into an unused portion of the BASIC Stamp's EEPROM program memory. When you run the program in the next activity, it will read these values from EEPROM and use them to determine the Compass Module's actual heading. This is called "calibration in software" because the procedure does not make any physical adjustments to the actual compass module.

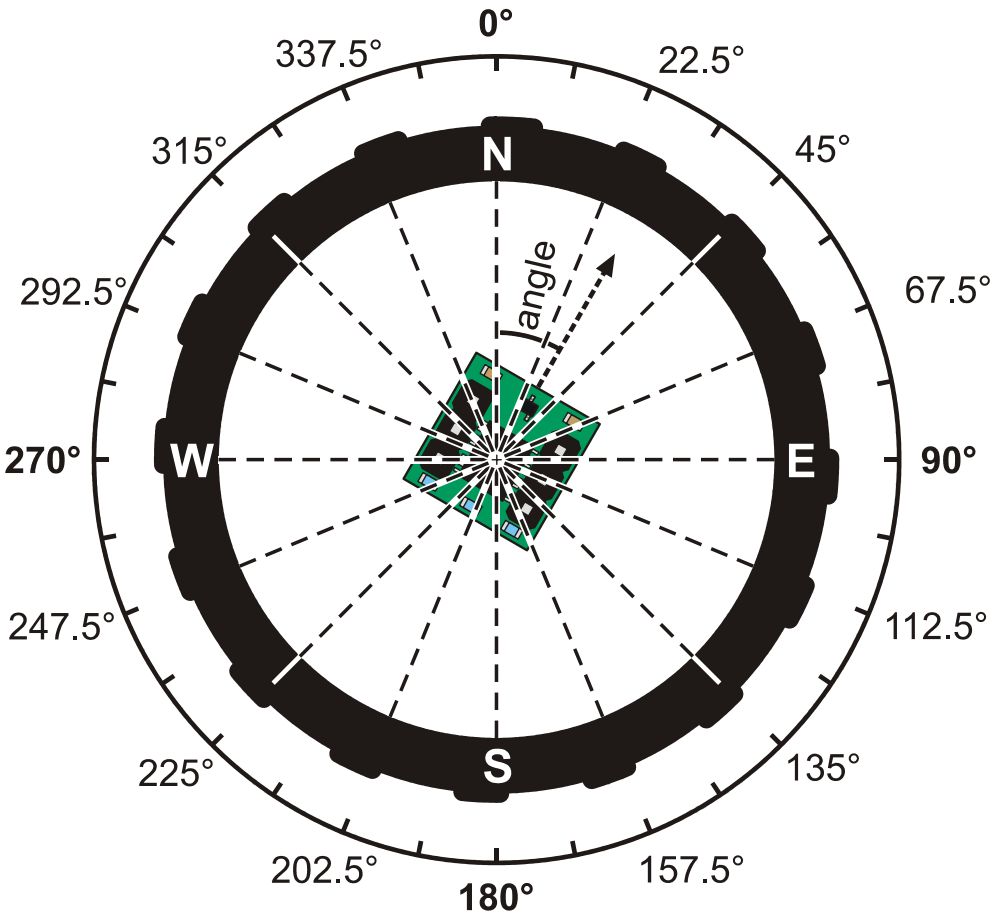
Calibration Setup and Procedure

The setup involves aligning a compass printout and taping it to a flat surface. The procedure involves running this activity's example program and following the prompts as you align the Board of Education to the various spokes in the compass wheel.

Setup

√ Print or make a photocopy of Figure 4-7. If you are working from a printed copy of the book and don't have a photocopier at your disposal, just download the .pdf version of this text from the Smart Sensors and Applications product page at www.parallax.com. Then, make a printout from that.

Figure 4-7: Calibration Compass



√ Place your copy of Figure 4-7 on a flat, level, non metallic surface. Make sure it is as far away from your monitor as your programming cable can reach. The location should also be as far as possible from metal containers, appliances, and any other potential source of magnetic field interference. Also check your table for metal mountings underneath.

√ Before finalizing your location, take your mechanical compass well away from any sources of magnetic interference and note the direction. Then, place the

mechanical compass on your work surface. The direction of north it indicates should not change. If it does, find a different location without magnetic interference.

√ Use the mechanical compass to align the 0° line with magnetic north as shown in Figure 4-8.

√ Tape the printout to the table making sure not to disturb the sheet as you do so.

√ Set the mechanical compass well away from your printout.

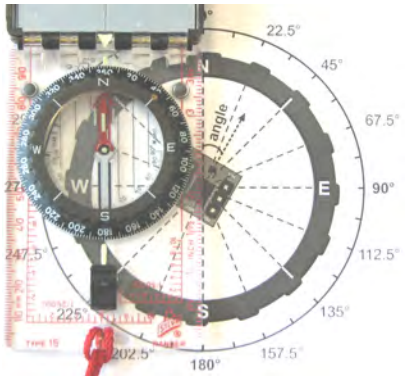


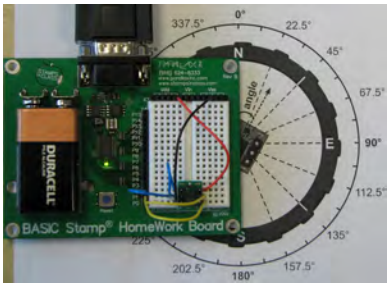
Figure 4-8
Aligning to Magnetic North

Procedure

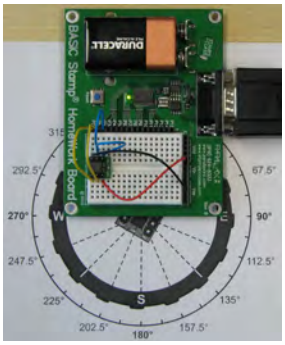
When you run CalibrateCompass.bs2, it will prompt you to align your board to various angles on the compass printout, and to press the enter key after each one. The first two angles (0 and 90°) are shown in Figure 4-9.

Figure 4-9: Compass at 0° and 90°

a.



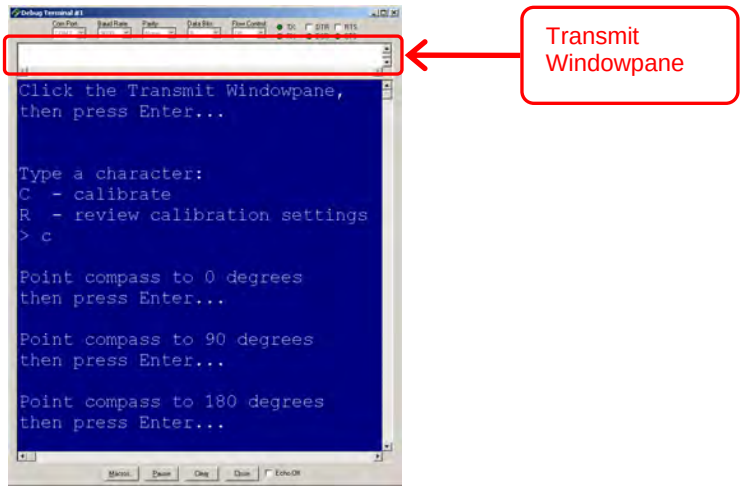
b.



When you run CalibrateCompass.bs2, you will first be prompted to click the Debug Terminal's Transmit windowpane (shown in Figure 4-10), and then to press Enter. After that you will be prompted to type C for calibrate or R for review calibration settings. Typing the letter C will start the calibration, during which you will be prompted to point the compass to 0°, and 90° as shown in Figure 4-9, and then to: 180°, 270°, 0°, 22.5°, 45°, 67.5°, 90°, 112.5°, 135°, 157.5°, 180°, 202.5°, 225°, 247.5°, 250°, 292.5°, 315°, and finally, 337.5°. You will need to press the Enter key before advancing your board to each angle.

- √ Open and run CalibrateCompass.bs2.
- √ Follow the prompts until you get to the "CALIBRATION COMPLETED" message.
- √ If you make a mistake, restart the program and start from the beginning. The calibration process only takes a minute or two, and it's worth it to have the correct settings in your BASIC Stamp 2's EEPROM memory.

Figure 4-10: Transmit Windowpane



Example Program - CalibrateCompass.bs2



Free Download! This program is available as a free .bs2 file download from the Smart Sensors and Applications Product Page at www.parallax.com. If you would like to know how this program works, read through the comments here.

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - CalibrateCompass.bs2
' This program collects and stores Hitachi HM55B Compass Module measurements
' in EEPROM for axis offset and linear interpolation corrections that will be
' performed by TestCalibratedCompass.bs2.
'
' {$STAMP BS2}
' {$PBASIC 2.5}
'
' IMPORTANT: Follow the instructions in Chapter #4, Activity #2 of
'            Smart Sensors and Applications.  It's available for
'            download from the Smart Sensors and Applications page at
'            www.parallax.com.

' -----[ EEPROM Data ]------------------------------------------------------

CompassOffsets DATA @ 0, (4)                ' Stores x and y axis offsets
CompassLowVal  DATA      (1)                ' Stores index of lowest angle
CompassCal     DATA      (16)               ' 16 reference compass angles

' -----[ Pin Definitions ]--------------------------------------------------
DinDout        PIN      2                   ' P2 transceives to/from Din/Dout
Clk            PIN      0                   ' P0 sends pulses to HM55B's Clk
En             PIN      1                   ' P1 controls HM55B's /EN(ABLE)

' -----[ Constants ]-------------------------------------------------------
Reset          CON      %0000               ' Reset command for HM55B
Measure        CON      %1000               ' Start measurement command
Report         CON      %1100               ' Get status/axis values command
Ready          CON      %1100               ' 11 -> Done, 00 -> no errors
NegMask        CON      %1111100000000000   ' For 11-bit negative to 16-bits

Current        CON      0                   ' Index for table array
Previous       CON      1                   ' Index for table array

' -----[ Variables ]-------------------------------------------------------
x              VAR      Word                ' x-axis data
y              VAR      Word                ' y-axis data
status         VAR      Nib                 ' Status flags
angle          VAR      Word                ' Angle measurement
counter        VAR      Byte                ' Loop counter
index          VAR      Nib                 ' EEPROM index
character      VAR      Byte                ' Stores a DEBUGIN character
```

```
integer        VAR      Word                ' Integer values for display
fraction       VAR      Nib                 ' Fractional values for display
brads          VAR      Byte                ' Binary radian measurements
table          VAR      Byte(2)             ' Stores table values
temp           VAR      Word(2)             ' Stores axis measurements
axisOffset     VAR      Word                ' Stores axis offset value

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

DEBUG "Click the Transmit Windowpane, ", CR, ' Wait for user.
      "then press Enter... ", CR, CR

DEBUGIN character

DO                                          ' Main loop

  DEBUG "Type a character: ", CR,           ' Menu
        "C  - calibrate ", CR,
        "R  - review calibration settings", CR,
        "> "
  DEBUGIN Character                         ' Get user selection
  DEBUG CR

  IF character = "c" OR character = "C" THEN ' "c" -> calibrate
    GOSUB Compass_Calibrate                 ' "r" -> review settings
  ELSEIF character = "r" OR character = "R" THEN
    GOSUB Calibration_Review
  ENDIF

  DEBUG CR, "Press any key to",             ' wait for user
        CR, "continue"
  DEBUGIN character
  DEBUG CR, CR

LOOP                                        ' Repeat main loop

' -----[ Subroutine - Compass_Calibrate ]-----------------------------------

  Compass_Calibrate:

    GOSUB Get_And_Store_Axis_Offsets
    GOSUB Get_And_Store_Interpolation
    GOSUB Get_And_Store_Low_Value_Address
    DEBUG CR, "CALIBRATION COMPLETED...", CR,
          "You are now ready to run ", CR,
          "TestCalibratedCompass.bs2.", CR
    RETURN

' -----[ Subroutine - Get_And_Store_Axis_Offsets ]---------------------------

' This subroutine prompts the user to point the compass north, then east, then
```

```
' south, then west.  It then averages the maximum and minimum values for each
' axis and stores that average in the EEPROM area reserved by the
' CompassOffsets DATA directive.

Get_And_Store_Axis_Offsets:

  ' FOR...NEXT loop repeats for four axis measurements.
  FOR counter = 0 TO 3

    ' Instruct user to point compass to a particular direction, then wait
    ' for ENTER character.
    DEBUG CR, "Point compass to "
    LOOKUP counter, [ 0, 90, 180, 270 ], integer
    DEBUG DEC integer
    DEBUG " degrees", CR, "then press Enter..."
    DEBUGIN character

    GOSUB Compass_Get_Axes                  ' Get axis measurements

    ' Calculate offsets based on max and min values for each axis, then store
    ' in EEPROM.
    SELECT counter
      CASE 0                                ' North
        temp(0) = x
      CASE 1                                ' East
        temp(1) = y
      CASE 2                                ' South
        x = x + temp(0)
        IF x.BIT15 = 1 THEN
          x = ABS(x)/2
          x = -x
        ELSE
          x = x / 2
        ENDIF
        WRITE CompassOffsets, Word x
      CASE 3                                ' West
        y = y + temp(1)
        IF Y.BIT15 = 1 THEN
          y = ABS(y)/2
          y = - y
        ELSE
          y = x / 2
        ENDIF
        WRITE CompassOffsets + 2, Word y
    ENDSELECT

  NEXT

  RETURN

' -----[ Subroutine - Get_And_Store_Interpolation ]----------------------
```

```
' This subroutine prompts the user to point the compass to directions
' separated by 22.5 degrees and stores the angle for each of the measurements
' in the EEPROM area reserved by the CompassCal DATA directive.

Get_And_Store_Interpolation:

  FOR counter = 0 TO 15
    DEBUG CR, "Point compass to "
    LOOKUP counter, [0,    22,  45,  67,  90, 112, 135, 157,
                     180, 202, 225, 247, 270, 292, 315, 337], integer
    LOOKUP counter, [ 0,    5,   0,   5,   0,   5,   0,   5,
                      0,    5,   0,   5,   0,   5,   0,   5 ], fraction
    DEBUG DEC integer
    IF fraction = 5 THEN DEBUG ".", DEC fraction
    DEBUG " degrees", CR, "then press Enter..."
    DEBUGIN character                          ' Wait for user
    GOSUB Compass_Get_Axes                     ' Get x, and y values
    GOSUB Compass_Correct_Offsets              ' Correct axis offsets
    angle = x ATN - y                          ' Convert x and y to brads
    WRITE CompassCal + counter, angle          ' Store as brad value
  NEXT

  RETURN

' -----[ Subroutine - Get_And_Store_Low_Value_Address ]-----------------------

' This subroutine finds and stores the address of the lowest value in the
' EEPROM area reserved by the CompassCal DATA directive and stores it in
' a byte reserved by the CompassLowVal DATA directive.  This reduces the
' code overhead in TestCalibratedCompass.bs2.

Get_And_Store_Low_Value_Address:

  index = 8
  table(current) = 0: table(previous) = 0
  DO
    index = index + 1
    READ CompassCal + index, table(current)
    READ CompassCal + (index - 1 & $F), table(previous)
  LOOP UNTIL table(current) < table(previous)
  WRITE CompassLowVal, index

  RETURN

' -----[ Subroutine - Calibration_Review ]-----------------------------------

  ' Display EEPROM values.

  Calibration_Review:
```

```
    DEBUG CR, "Axis Offsets:", CR
    READ CompassOffsets, Word x
    DEBUG CR, "x-Offset = ", SDEC x
    READ CompassOffsets + 2, Word y
    DEBUG CR, "y-Offset = ", SDEC y, CR

    DEBUG CR, "Index of low value in CompassCal:", CR
    READ CompassLowVal, index
    DEBUG CR, "Low value ", ? index

    DEBUG CR, "TestCalibratedCompass.bs2", CR,
              "uses these values to ", CR,
              "correct errors:", CR

    DEBUG CR, "Brad Angle        Degree Angle",
          CR, "Ideal    Actual   Ideal    Actual",
          CR, "------   ------   ------   ------", CR

    FOR counter = 0 TO 15
      brads = counter * 16
      DEBUG CRSRX, 1, DEC3 brads
      READ CompassCal + counter, angle
      DEBUG CRSRX, 10, DEC3 angle
      LOOKUP counter, [0,    22,  45,  67,  90, 112, 135, 157,
                       180, 202, 225, 247, 270, 292, 315, 337], integer
      LOOKUP counter, [ 0,    5,   0,   5,   0,   5,   0,   5,
                        0,    5,   0,   5,   0,   5,   0,   5 ], fraction
      DEBUG CRSRX, 19, DEC3 integer, ".", DEC fraction
      angle = angle */ 361                       ' Convert brads to degrees
      DEBUG CRSRX, 28, DEC3 angle, CR
      PAUSE 50                                   ' Debug delay for slower PCs
    NEXT

    DEBUG CR

    RETURN

' -----[ Subroutine - Compass_Get_Axes ]-------------------------------------

Compass_Get_Axes:                            ' Compass module subroutine

  HIGH En: LOW En                            ' Send reset command to HM55B
  SHIFTOUT DinDout,clk,MSBFIRST,[Reset\4]

  HIGH En: LOW En                            ' HM55B start measurement command
  SHIFTOUT DinDout,clk,MSBFIRST,[Measure\4]
  status = 0                                 ' Clear previous status flags

  DO                                         ' Status flag checking loop
    HIGH En: LOW En                          ' Measurement status command
    SHIFTOUT DinDout,clk,MSBFIRST,[Report\4]
```

```
    SHIFTIN  DinDout,clk,MSBPOST,[Status\4]  ' Get Status
  LOOP UNTIL status = Ready                  ' Exit loop when status is ready

  SHIFTIN  DinDout,clk,MSBPOST,[x\11,y\11]   ' Get x & y axis values
  HIGH En                                    ' Disable module

  IF (y.BIT10 = 1) THEN y = y | NegMask      ' Store 11-bits as signed word
  IF (x.BIT10 = 1) THEN x = x | NegMask      ' Repeat for other axis

  RETURN

' -----[ Subroutine - Compass_Correct_Offsets ]------------------------------

' This subroutine corrects cumulative magnetic field interference that can
' come from sources such as the PCB, jumper wires, a nearby battery, or a
' nearby current source.  This subroutine relies on values stored in
' the EEPROM space that was reserved by the CompassOffsets DATA directive.
' These EEPROM values are written by this program during calibration.

Compass_Correct_Offsets:

  READ CompassOffsets, Word axisOffset           ' Get x-axis offset
  x = x - axisOffset                             ' Correct x-axis
  READ CompassOffsets + 2, Word axisOffset       ' Get y-axis offset
  y = y - axisOffset                             ' Correct y-axis

  RETURN
```

Your Turn - Reviewing the Calibration Settings

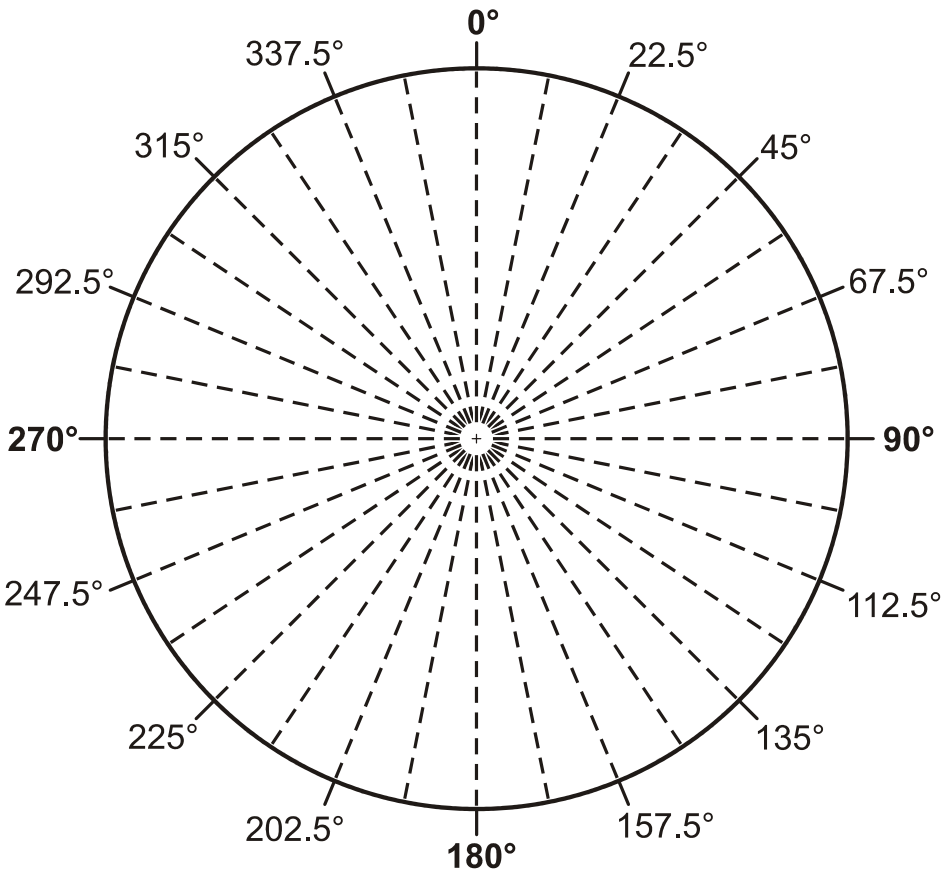
In the main activity, you typed the letter C to store calibration values in the BASIC Stamp's EEPROM. You can also review these calibration values by running the program and typing R instead of C. This will show a comparison of the actual vs. ideal angle measurements in binary radians. These errors are caused in part by the printed circuit board that the sensor is mounted on. Some of the materials in the printed circuit board are magnetic, and are not necessarily aligned with the Earth's magnetic field. Other magnetic field sources that can cause measurement errors come from nearby electrical currents, such as electrons flowing through the Vdd and Vss lines to power your board's power LED.

- √ Run CalibrateCompass.bs2 again.
- √ Click the Debug Terminal's Transmit windowpane and press Enter.
- √ Type R to review the calibration settings.
- √ Examine the errors reported, which the example program in the next activity will use to make corrections.

ACTIVITY #3: TESTING THE CALIBRATION

After Activity #2, the program in this activity should make your compass perform pretty well, well enough to correctly recognize most of the 64 directions in Figure 4-11. In this activity, Figure 4-11 will be used to test the compass module's performance.

Figure 4-11: 64-Direction Scale



Heading

TestCalibratedCompass.bs2 goes and finds the values that CalibrateCompass.bs2 recorded in the BASIC Stamp's EEPROM memory. Then, it uses the values to correct for scale error, and refines the measurements using a technique called linear interpolation.

- √ Print or photocopy the scale shown in Figure 4-11 and following the Setup in Activity #2 for aligning it to north and affixing it to the table.
- √ Calculate the unmarked angles on the scale.
- √ Open and run TestCalibratedCompass.bs2.
- √ Align your board to the various angles, and compare the measured angles reported by the compass module to the actual angles.

If this still isn't enough accuracy for you, the next activity will show you how to improve it even more.

Example Program: TestCalibratedCompass.bs2



Free Download This program is available as a free .bs2 file download from the Smart Sensors and Applications Product Page at www.parallax.com. Read the code comments for an explanation of its function.

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - TestCalibratedCompass.bs2
' Demonstrates Hitachi HM55B Compass Module's accuracy after calibration with
' CalibrateCompass.bs2.

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

' -----[ Program Description ]---------------------------------------------
'
' This program displays the following Hitachi HM55B Compass Sensor
' measurements:
'
'  - Offset corrected x and y-axis magnetic field measurements
'  - Binary radian angle clockwise from north corrected by linear
'    interpolation table
'  - Degree angle clockwise from north corrected by linear interpolation
'    table

' IMPORTANT: This program relies on EEPROM values that are stored by
'            CalibrateCompass.bs2 during the calibration process.
```

```
'            That calibration process must be performed prior to running
'            this test program.
'
'            For instructions on how to perform the calibration process,
'            consult Chapter #4, Activity #2 of
'            of Smart Sensors and Applications.  It's available for
'            download from the Smart Sensors and Applications page at
'            www.parallax.com.

' -----[ EEPROM Data ]-----------------------------------------------------

CompassOffsets DATA @ 0, (4)                ' Stores x and y axis offsets
CompassLowVal  DATA      (1)                ' Stores index of lowest angle
CompassCal     DATA      (16)               ' 16 reference compass angles

' -----[ Pin Definitions ]-------------------------------------------------

DinDout        PIN      2                  ' P6 transceives to/from Din/Dout
Clk            PIN      0                  ' P5 sends pulses to HM55B's Clk
En             PIN      1                  ' P4 controls HM55B's /EN(ABLE)

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

Reset          CON      %0000              ' Reset command for HM55B
Measure        CON      %1000              ' Start measurement command
Report         CON      %1100              ' Get status/axis values command
Ready          CON      %1100              ' 11 -> Done, 00 -> no errors
NegMask        CON      %1111100000000000  ' For 11-bit negative to 16-bits
current        CON      0                  ' Table array index
previous       CON      1                  ' Table array index

' -----[ Variables ]------------------------------------------------------
x              VAR      Word               ' x-axis data
y              VAR      Word               ' y-axis data
status         VAR      Nib                ' Status flags
angle          VAR      Word               ' Angle measurement
axisOffset     VAR      angle              ' Axis offset

index          VAR      Status             ' EEPROM index
table          VAR      Byte(2)            ' Stores EEPROM table values
span           VAR      x                  ' Span between table entries
angleOffset    VAR      y                  ' Offset btwn measured and table

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

DEBUG CLS

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

DO                                          ' Main loop
```

```
  GOSUB Compass_Get_Axes                      ' Get x, and y values
  GOSUB Compass_Correct_Offsets               ' Correct axis offsetes
  angle = x ATN -y                            ' Convert x and y to brads
  DEBUG HOME, "x-axis N(-S) = ",SDEC x,       ' Display corrected axes
        CLREOL, CR, "y-axis W(-E) = ",
        SDEC y, CLREOL
  GOSUB Compass_Interpolate                   ' Linear interpolation
  DEBUG CR, CR, "angle = ",                   ' Display inrerpolated angle
        DEC angle, " brads", CLREOL           ' ... in brads
  angle = angle */ 361                        ' Convert brads to degrees
  DEBUG CR,"angle = ",                        ' Display inrerpolated angle
        DEC angle, " degrees", CLREOL         ' ... in degrees
  PAUSE 150                                   ' Debug delay for slower PCs

LOOP                                          ' Repeat main loop

' -----[ Subroutine - Compass_Get_Axes ]-------------------------------------

' This subroutine handles BASIC Stamp - HM55B communication and stores the
' magnetic field strength measurements returned by the device in the x and
' y axis variables.

Compass_Get_Axes:                             ' Compass module subroutine

  HIGH En: LOW En                             ' Send reset command to HM55B
  SHIFTOUT DinDout,clk,MSBFIRST,[Reset\4]

  HIGH En: LOW En                             ' HM55B start measurement command
  SHIFTOUT DinDout,clk,MSBFIRST,[Measure\4]
  status = 0                                  ' Clear previous status flags

  DO                                          ' Status flag checking loop
    HIGH En: LOW En                           ' Measurement status command
    SHIFTOUT DinDout,clk,MSBFIRST,[Report\4]
    SHIFTIN  DinDout,clk,MSBPOST,[Status\4]   ' Get Status
  LOOP UNTIL status = Ready                   ' Exit loop when status is ready

  SHIFTIN  DinDout,clk,MSBPOST,[x\11,y\11]    ' Get x & y axis values
  HIGH En                                     ' Disable module

  IF (y.BIT10 = 1) THEN y = y | NegMask       ' Store 11-bits as signed word
  IF (x.BIT10 = 1) THEN x = x | NegMask       ' Repeat for other axis

  RETURN

' -----[ Subroutine - Compass_Correct_Offsets ]------------------------------

' This subroutine corrects cumulative magnetic field interference that can
' come from sources such as the PCB, jumper wires, a nearby battery, or a
' nearby current source.  This subroutine relies on values stored in
' the EEPROM space that was reserved by the CompassOffsets DATA directive.
```

```
' These EEPROM values were written by CalibrateCompass.bs2.

Compass_Correct_Offsets:

  READ CompassOffsets, Word axisOffset           ' Get x-axis offset
  x = x - axisOffset                             ' Correct x-axis
  READ CompassOffsets + 2, Word axisOffset       ' Get y-axis offset
  y = y - axisOffset                             ' Correct y-axis

  RETURN

' -----[ Subroutine - Compass_Interpolate ]----------------------------------

' This subroutine applies linear interpolation to the refine the compass
' measurement.  This second level of refinement can be performed after the
' Compass_Correct_Offsets subroutine, and it can correct axis skew and other
' factors inherent to the HM55B chip.
'
' The subroutine relies on 16 actual compass measurements that were stored
' in the sixteen EEPROM locations reserved by the CompassCal DATA directive.
' These measurements were stored by CalibrateCompass.bs2, and they
' represent the actual compass measurements for 0, 22.5, 45, 90,..., 337.5
' degrees.  The subroutine finds the two EEPROM measurements that the current
' angle measurement falls between.  It then updates the angle measurement
' based on where the angle measurement falls between the two known table
' values.

Compass_Interpolate:

  ' Start with the lowest value in the CompassCal table.

  READ CompassLowVal, index

  ' Load current and previous table values.

  READ CompassCal + index, table(current)
  READ (CompassCal + (index - 1 & $F)), table(previous)


  ' The IF...ELSEIF...ELSE...ENDIF code block finds the two EEPROM CompassCal
  ' table values that the current angle measurement falls between and
  ' calculates the difference between the current angle measurement and the
  ' lower of the two table values.  The IF and ELSEIF blocks deal with values
  ' that are greater than the highest or less than the lowest table values.
  ' The ELSE block handles everything in between the highest and lowest table
  ' values.

  IF (angle >= table(previous)) THEN
    span = (255 - table(previous)) + table(current)
    angleOffset = angle - table(previous)
  ELSEIF (angle <= table(current)) THEN
```

```
  span = table(current) + (255 - table(previous))
  angleOffset = angle + (255 - table(previous))
ELSE
  index = index - 1
  READ CompassCal + index, table(current)
  DO
    table(previous) = table(current)
    index = index + 1
    READ CompassCal + index, table(current)
    IF (angle <= table(current)) AND (angle > table(previous)) THEN
      span = table(current) - table(previous)
      angleOffset = angle - table(previous)
      EXIT
    ENDIF
  LOOP
ENDIF

' After the offset between the current angle measurement and the next lower
' table measurement has been determined, this code block uses it along with
' the span between the table entries above and below the angle measurement
' to solve for: angle(corrected) = angle(offset) * 16 / span.
' This code block also rounds up or down by comparing the remainder of
' the angleOffset / span division to the value of (span / 2).

angleOffset = angleOffset * 16
angle = (angleOffset / span) + ((angleOffset // span) / (span / 2))
angle = ((index - 1 & $F) * 16) + angle
angle = angle & $ff

RETURN
```

Your Turn - Displaying "Degrees" as °

Displaying the degree ° symbol in the Debug Terminal was first introduced in Chapter #3, Activity #5.

√ Modify the program to display degrees with ASCII character 176, the ° symbol.

ACTIVITY #4: IMPROVE COMPASS MEASUREMENTS BY AVERAGING

You may have noticed that the x and y measurements in the Debug Terminal tended to alternate between two or even three different values. This is the result of several different types of interference that are collectively called noise. Some common culprits are nearby AC devices and power lines, digital activity in the BASIC Stamp, and even digital activity inside the HM55B chip.

One effective way to eliminate the effects of noise is by taking an average of the compass' x and y axis measurements. That way, if noise causes one measurement to be a little high, the next one a little low, and the one after that is about right, the average of all the measurements will eliminate the highs and lows and settle on the right value.

One of the reasons the calibration and calibration testing activities may not have yielded the best results is because of noise. This activity demonstrates how you can modify any of the example programs in this chapter, including the calibration and calibration test programs, to take averaged measurements and eliminate the effects of noise.

Incorporating Averaging into the Compass Programs

There are three major steps to incorporating averaging into this chapter's example programs. First, add a couple of **CON** directives to the program's Constants section.

```
Negative       CON      1            ' Word.bit15 = 1 -> negative
Positive       CON      0            ' Word.bit15 = 0 -> positive
```

Then, add four variables to the program's Variables section.

```
mCount         VAR      Nib          ' Measurement count
xSum           VAR      Word         ' x-axis measurement accumulator
ySum           VAR      Word         ' y-axis measurement accumulator
sign           VAR      Bit          ' Sign bit
```

Finally, modify the **Compass_Get_Axes** subroutine as shown below. The code from the original **Compass_Get_Axes** subroutine is nested in a **FOR**...**NEXT** loop that keeps a running sum of the x and y-axis measurements with the **xSum** and **ySum** variables. Since the average of a group of measurements is the sum of the measurements divided by the number of measurements taken, there are code blocks after the **FOR**...**NEXT** loop that set x equal to **xSum ÷ 10** and y equal **to ySum ÷ 10**.

```
Compass_Get_Axes:                   ' Modified subroutine

  xSum = 0                          ' Accumulators to zero
  ySum = 0

  FOR mCount = 1 TO 10              ' Take ten measurements

    ' *** Original code from Compass_Get_Axes subroutine goes here ***

    xSum = xSum + x                 ' Keep a running sum of x
    ySum = ySum + y                 ' Keep a running sum of y

  NEXT
```

```
' Divide xSum
sign = xSum.BIT15                   ' Store sign of xSum
xSum = ABS(xSum)                    ' Take absolute value
x = xSum / 10                       ' x = the average measurement
IF xSum // 10 >=5 THEN x = x + 1    ' Fraction > .5? Round up
IF sign = Negative THEN x = - x     ' if xSum negative, negate x

sign = ySum.BIT15                   ' Store sign of ySum
ySum = ABS(ySum)                    ' Take absolute value
y = ySum / 10                       ' y = the average measurement
IF ySum // 10 >=5 THEN y = y + 1    ' Fraction > .5? Round up
IF sign = Negative THEN y = - y     ' if ySum negative, negate y

RETURN
```



PBASIC Division with Negative Numbers

The PBASIC division and modulus (/ and //) operators are for use with positive numbers. If the numerator might be negative, the best approach is to save the numerator's sign before taking its absolute value (**`sign = numerator.BIT15`**). Then, perform the division operation. Optionally, you can also round up or down depending on the remainder of the division. Before you're done, check the sign, and if it's negative, make the result negative (**`result = - result`**).

```
numerator VAR Word
denominator VAR Word
result VAR Word
sign VAR Bit

Negative CON 1
Positive CON 0

' Division routine with a numerator that might be negative.
    sign = numerator.BIT15
    numerator = ABS(numerator)
    result = numerator / denominator
    IF numerator // denominator >= (denominator / 2) THEN
      result = result + 1
    ENDIF
    IF sign = Negative THEN result = - result
```

Example Program: TestCompassAveraged.bs2



Free Download This program is available as a free .bs2 file download from the Smart Sensors and Applications Product Page at www.parallax.com.

The procedure for converting a program to average its x and y-axis measurements was applied to TestCompass.bs2, and then saved as TestCompassAveraged.bs2.

- √ Open and run TestCompass.bs2 from activity #1.
- √ Watch the x and y-axis measurements at a few different headings. They will likely be noisy, flickering between two or three different values.
- √ Open and run TestCompassAveraged.bs2.
- √ The measurements should be much more stable. They should only flicker when you are very close to the transition between two different results.

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - TestCompassAveraged.bs2
' Test to make sure Hitachi HM55B Compass Module is working.

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

' -----[ I/O Definitions ]--------------------------------------------------

DinDout        PIN      2                   ' P2 transceives to/from Din/Dout
Clk            PIN      0                   ' P0 sends pulses to HM55B's Clk
En             PIN      1                   ' P2 controls HM55B's /EN(ABLE)

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

Reset          CON      %0000               ' Reset command for HM55B
Measure        CON      %1000               ' Start measurement command
Report         CON      %1100               ' Get status/axis values command
Ready          CON      %1100               ' 11 -> Done, 00 -> no errors
NegMask        CON      %1111100000000000   ' For 11-bit negative to 16-bits

Negative       CON      1                   ' Word.bit15 = 1 -> negative
Positive       CON      0                   ' Word.bit15 = 0 -> positive

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

x              VAR      Word                ' x-axis data
y              VAR      Word                ' y-axis data
status         VAR      Nib                 ' Status flags
angle          VAR      Word                ' Store angle measurement

mCount         VAR      Nib                 ' Measurement count
xSum           VAR      Word                ' x-axis measurement accumulator
ySum           VAR      Word                ' y-axis measurement accumulator
sign           VAR      Bit                 ' Sign bit

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

```
DO                                          ' Main loop

  GOSUB Compass_Get_Axes                    ' Get x, and y values

  angle = x ATN -y                          ' Convert x and y to brads
  angle = angle */ 361                      ' Convert brads to degrees

  DEBUG HOME, "x-axis N(-S) = ",SDEC x,     ' Display axes and degrees
        CLREOL, CR, "y-axis W(-E) = ",
        SDEC y, CLREOL, CR, CR, "angle = ",
        DEC angle, " degrees", CLREOL

  PAUSE 150                                 ' Debug delay for slower PCs

LOOP                                        ' Repeat main loop

' -----[ Subroutine - Compass_Get_Axes ]-------------------------------------

Compass_Get_Axes:                           ' Compass module subroutine

  xSum = 0                                    ' Accumulators to zero
  ySum = 0

  FOR mCount = 1 TO 10                        ' Take ten measurements

    HIGH En: LOW En                           ' Send reset command to HM55B
    SHIFTOUT DinDout,clk,MSBFIRST,[Reset\4]

    HIGH En: LOW En                           ' HM55B start measurement cmd
    SHIFTOUT DinDout,clk,MSBFIRST,[Measure\4]
    status = 0                                ' Clear previous status flags

    DO                                        ' Status flag checking loop
      HIGH En: LOW En                         ' Measurement status command
      SHIFTOUT DinDout,clk,MSBFIRST,[Report\4]
      SHIFTIN  DinDout,clk,MSBPOST,[Status\4]  ' Get Status
    LOOP UNTIL status = Ready                 ' Status ready? Exit loop

    SHIFTIN  DinDout,clk,MSBPOST,[x\11,y\11]   ' Get x & y axis values
    HIGH En                                   ' Disable module

    IF (y.BIT10 = 1) THEN y = y | NegMask      ' Store 11-bits as signed word
    IF (x.BIT10 = 1) THEN x = x | NegMask      ' Repeat for other axis

    xSum = xSum + x                           ' Keep a running sum of x
    ySum = ySum + y                           ' Keep a running sum of y

  NEXT

  sign = xSum.BIT15                           ' Store sign of xSum
```

```
xSum = ABS(xSum)                              ' Take absolute value
x = xSum / 10                                 ' x = the average measurement
IF xSum // 10 >=5 THEN x = x + 1              ' Fraction > .5? Round up
IF sign = Negative THEN x = - x               ' if xSum negative, negate x

sign = ySum.BIT15                             ' Store sign of ySum
ySum = ABS(ySum)                              ' Take absolute value
y = ySum / 10                                 ' y = the average measurement
IF ySum // 10 >=5 THEN y = y + 1              ' Fraction > .5? Round up
IF sign = Negative THEN y = - y               ' if ySum negative, negate y

RETURN
```

Your Turn - Averaging the Calibration and Calibration Test Programs

The calibration and test calibration programs significantly improve the accuracy of your digital compass. By incorporating averaging into both programs, the accuracy of your digital compass will be further improved.

- √ Follow the steps in this activity to incorporate averaging into a copy of CalibrateCompass.bs2. Instead of modifying the program's **`Compass_Get_Axes`** subroutine just copy the modified subroutine from this program (TestCompassAveraged.bs2) and paste it over the one in your copy of CalibrateCompass.bs2.
- √ Run your modified copy of CalibrateCompass.bs2 and repeat the steps in Activity #2.
- √ Make a copy of TestCalibratedCompass.bs2, and modify it to perform averaging.
- √ Repeat the accuracy tests in Activity #3. Your digital compass should perform really well now.

ACTIVITY #5: MOBILE MEASUREMENTS

This activity demonstrates how to replace the Debug Terminal with the Parallax Serial LCD to make your digital compass mobile.

Connecting the Parallax Serial LCD with an Extension Cable

The Parallax Serial LCD is a definite source of magnetic field disturbance and needs to be operated well away from the compass module. This is easily done with an extension cable.

Parts Required

(1) Hitachi HM55B Compass Module
(1) Parallax Serial LCD (2×16)
(1) 14-inch LCD Extension Cable
(6) Jumper Wires

If you are working from a BASIC Stamp HomeWork Board or a serial Board of Education Rev A or B, you will also need:

(1) 3-pin header
(3) additional jumper wires

LCD Cable Connections

The schematics shown in Figure 4-12 are identical to the ones that have been used for Compass Module and Parallax Serial LCD up to this point. The only thing that will be changed is the way the LCD is connected to your board, with an extension cable. No changes should be made to the compass module's wiring.

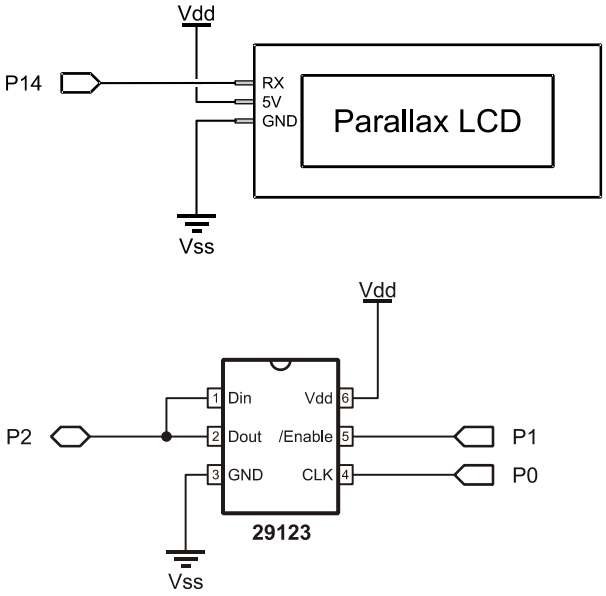


Figure 4-12
Parallax Serial LCD and Compass Module Schematics

Board of Education Rev C and USB Board of Education Cable Connections

These instructions are for the boards that have servo ports with a Vdd/Vss jumper in between, such as the Board of Education Rev C and USB Board of Education. For all other boards, skip to **All other BASIC Stamp Educational Boards** on page 151.

- √ Disconnect power to your board.
- √ Set the jumper between the X4 and X5 servo to Vdd (+5 V) as shown in Figure 4-13. The jumper should cover the two pins closest to Vdd, and the third pin next to Vin should be visible.

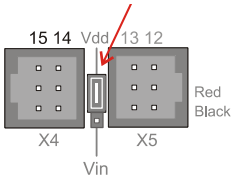


Figure 4-13
Setting the Servo Port Jumper to Vdd

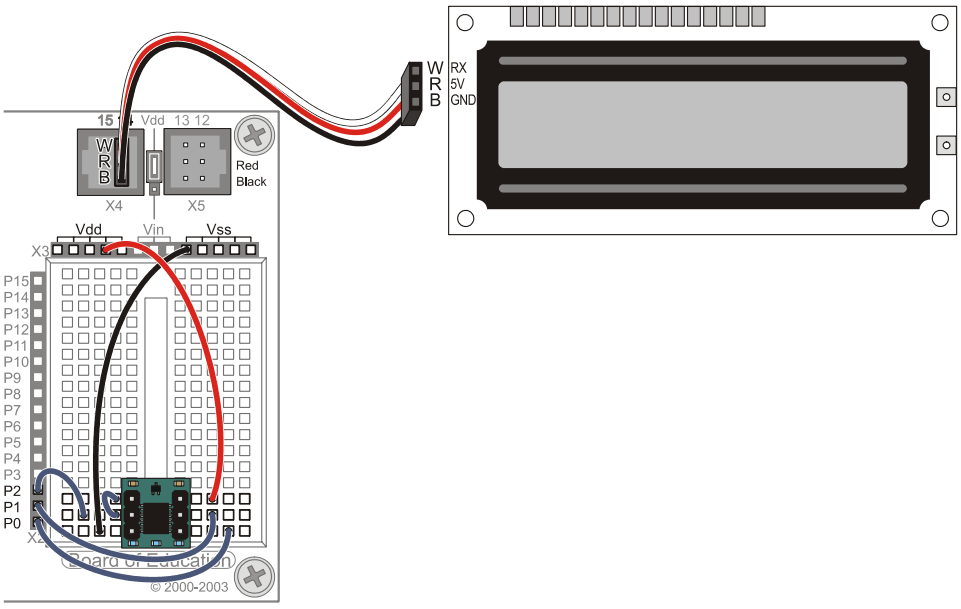
Vdd vs. Vin jumper settings determine which power supply is connected to the X4 and X5 ports. When the jumper is set to Vdd, these ports receive regulated 5 V from the Board of Education's voltage regulator. If the jumper is set to Vin, the port receives power directly from the battery or power supply. **WARNING!! MAKE SURE YOUR JUMPER IS SET CORRECTLY TO Vdd OR YOU WILL PERMANENTLY DAMAGE YOUR LCD!!**

- √ Plug one end of the extension cable into Port 14 of the X4 header, making sure that the "Red" and "Black" labels along the right side of the X5 port line up with the cable's red and black wires.
- √ Verify that your cable is plugged in correctly by checking to make sure the white wire is closest to the 14 label and the black wire is closest to the X4 label.
- √ Connect the other end of the cable so that the black wire is connected to the Parallax Serial LCD's GND pin, the red wire is connected to the 5 V pin, and the white wire is connected to the RX pin.
- √ Double-check all your connections and make sure they are correct.

WARNING!

Do not reconnect power to your board until you are positive the connections are correct. If you make a mistake with the LCD connections, the Parallax Serial LCD will be permanently damaged.

Figure 4-14: Parallax Serial LCD Servo Port Connections



√ Plug the power back into the Board of Education.
√ Set the Board of Education's 3-position switch to 2.
√ Skip to **Optional LCD Mounting Brackets** on page 153.

All other BASIC Stamp Educational Boards

This section is for connecting the Compass Module and Parallax Serial LCD to one of the following BASIC Stamp educational boards:

- BASIC Stamp HomeWork Board
- Board of Education Rev A (Serial version)
- Board of Education Rev B (Serial version)

√ Disconnect power from your board.
√ Build the extension cable port shown in Figure 4-15.

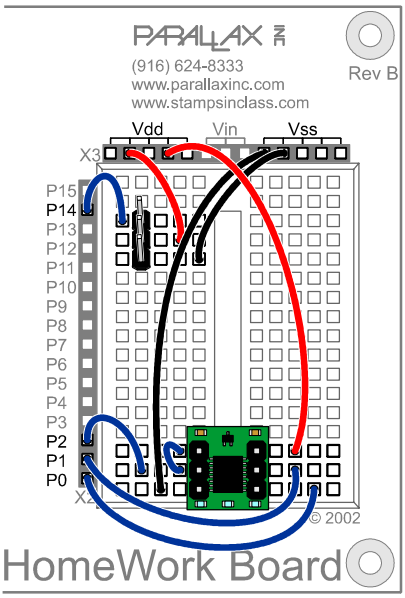
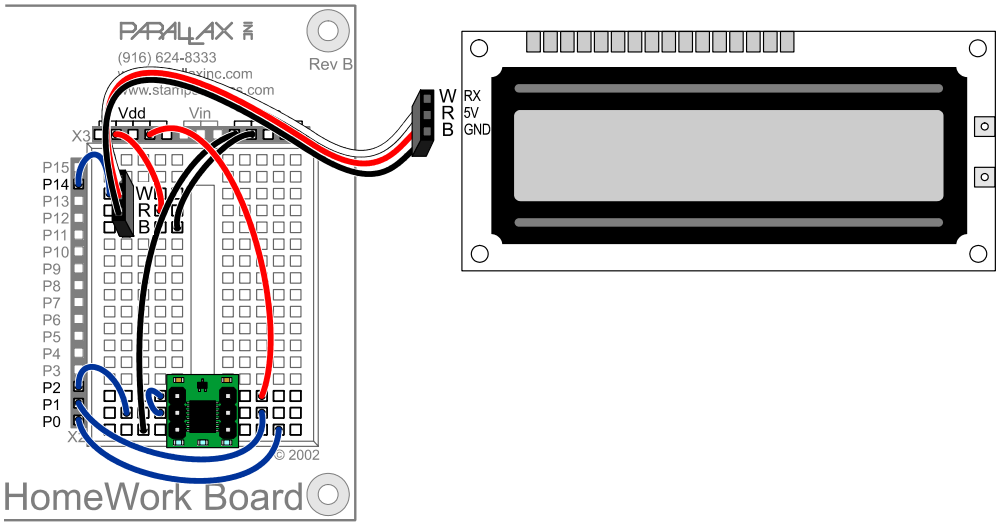


Figure 4-15
Breadboard Wiring for Parallax Serial LCD Cable Connection and the Compass Module

√ Plug one end of the extension cable into the 3-pin header on the board as shown in Figure 4-16. Make sure the white, red, and black wires are oriented as shown. The black wire should be connected to Vss, the red wire to Vdd, and the white wire to P14.

√ Connect the other end of the cable so that the black wire is connected to the Parallax Serial LCD's GND pin, the red wire is connected to the 5 V pin, and the white wire is connected to the RX pin.

Figure 4-16: Compass Module and Parallax Serial LCD Connected with Extension Cable



√ Double-check all your connections and make sure they are correct.

WARNING!

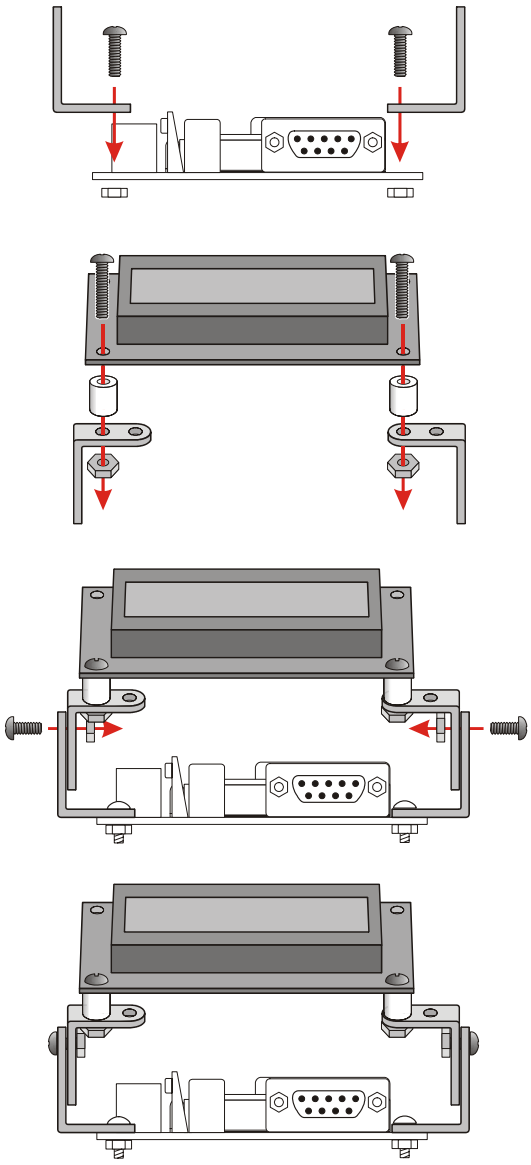
Do not reconnect power to your board until you are positive the connections are correct. If you make a mistake with the LCD connections, the Parallax Serial LCD will be permanently damaged.

√ Reconnect power to your board.

Optional LCD Mounting Brackets

If you wish, you may mount your LCD to your Board of Education or HomeWork Board with the brackets and hardware supplied in your kit. The parts list and directions are given on the next page with Figure 4-17.

Figure 4-17: Assembling the Optional LCD Bracket



Parts Required

(4) 90-degree mounting brackets
(2) 1/4-inch #4 round nylon spacers
(2) 1/2-inch 4-40 pan-head screws
(4) 1/4-inch 4-40 pan-head screws
(6) 4-40 zinc plated nuts

- √ Thread a ¼" screw through the outer hole of a bracket and the top left hole in your board, secure with a nut. Repeat at the bottom left hole in your board (top picture).
- √ Thread a ½" screw through the lower-left hole in the LCD board, a nylon standoff, the inner hole of a bracket, and secure with a nut (2nd picture, left).
- √ Thread a ½" screw through the lower-right hole in the LCD board, a nylon standoff, the outer hole of a bracket, and secure with a nut (2nd picture, right).
- √ Using the remaining ¼"screws and nuts, attach the LCD brackets to the brackets on your board, using two ¼" screws and nuts (3rd picture).

LCD Display Programming

This LCD initialization routine takes care of the LCD initialization basics, defines Custom Character 7 to be the ° symbol, then displays static text (text that won't change while the program is running).

```
' LCD Initialization routine
PAUSE 200                               ' Debounce power supply
SEROUT 14, 84, [22, 12]                 ' Turn on & clear LCD
PAUSE 5                                 ' 5 ms delay for clear cmd

SEROUT 14, 84, [255,                    ' Define Custom Character 7
                %01000,                 '   *
                %10100,                 ' *   *
                %01000,                 '   *
                %00000,                 '
                %00000,                 '
                %00000,                 '
                %00000,                 '
                %00000]                 '

SEROUT 14, 84, [129, "Heading...",      ' Static characters
                149, "x=",
                158, "y="]
```

The **SEROUT** command below places the LCD's cursor, then prints spaces to overwrite the previous value. Then it places the cursor at the same starting location and prints the new value. This prevents ghost characters from appearing when the number of digits in the value changes, but without the annoying side effects of screen flicker that you would otherwise get from clearing the display between each measurement.

```
' LCD Display heading in degrees on top line and x and y
' measurements on the bottom line.
SEROUT 14, 84, [139, "    ", 139, DEC angle, 7,
                151, "   ", 151, SDEC X,
                160, "   ", 160, SDEC y]
```

Example Program: LcdTestCompass.bs2



Free Download! This program is available as a free .bs2 file download from the Smart Sensors and Applications Product Page at www.parallax.com.

This example program is a modified version of TestCompass.bs2 that uses LCD display commands instead of Debug Terminal commands.

√ Open LcdTestCompass.bs2 try running it with the serial cable disconnected.

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - LcdTestCompass.bs2
' Test to make sure Hitachi HM55B Compass Module is working.

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

' -----[ I/O Definitions ]--------------------------------------------------

DinDout        PIN      2                   ' P2 transceives to/from Din/Dout
Clk            PIN      0                   ' P0 sends pulses to HM55B's Clk
En             PIN      1                   ' P2 controls HM55B's /EN(ABLE)

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

Reset          CON      %0000               ' Reset command for HM55B
Measure        CON      %1000               ' Start measurement command
Report         CON      %1100               ' Get status/axis values command
Ready          CON      %1100               ' 11 -> Done, 00 -> no errors
NegMask        CON      %1111100000000000   ' For 11-bit negative to 16-bits

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

x              VAR      Word                ' x-axis data
y              VAR      Word                ' y-axis data
status         VAR      Nib                 ' Status flags
angle          VAR      Word                ' Store angle measurement

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

' LCD Initialization
PAUSE 200                                 ' Debounce power supply
SEROUT 14, 84, [22, 12]                   ' Turn on & clear LCD
PAUSE 5                                   ' 5 ms delay for clear cmd

SEROUT 14, 84, [255,                      ' Define Custom Character 7
                %01000,                   '   *
                %10100,                   ' *   *
                %01000,                   '   *
                %00000,                   '
                %00000,                   '
                %00000,                   '
                %00000,                   '
                %00000]                   '
```

```
SEROUT 14, 84, [129, "Heading...",    ' Static characters
               149, "x=",
               158, "y="]

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

DO                                          ' Main loop

  GOSUB Compass_Get_Axes                    ' Get x, and y values

  angle = x ATN -y                          ' Convert x and y to brads
  angle = angle */ 361                      ' Convert brads to degrees

  ' LCD Display heading in degrees on top line and x and y
  ' measurements on the bottom line.
  SEROUT 14, 84, [139, "    ", 139, DEC angle, 7,
                 151, "   ", 151, SDEC X,
                 160, "   ", 160, SDEC y]

  PAUSE 150                                 ' Debug delay for slower PCs

LOOP                                        ' Repeat main loop

' -----[ Subroutine - Compass_Get_Axes ]-----------------------------------

Compass_Get_Axes:                           ' Compass module subroutine

  HIGH En: LOW En                           ' Send reset command to HM55B
  SHIFTOUT DinDout,clk,MSBFIRST,[Reset\4]

  HIGH En: LOW En                           ' HM55B start measurement command
  SHIFTOUT DinDout,clk,MSBFIRST,[Measure\4]
  status = 0                                ' Clear previous status flags

  DO                                        ' Status flag checking loop
    HIGH En: LOW En                         ' Measurement status command
    SHIFTOUT DinDout,clk,MSBFIRST,[Report\4]
    SHIFTIN  DinDout,clk,MSBPOST,[Status\4]  ' Get Status
  LOOP UNTIL status = Ready                 ' Exit loop when status is ready

  SHIFTIN  DinDout,clk,MSBPOST,[x\11,y\11]   ' Get x & y axis values
  HIGH En                                   ' Disable module

  IF (y.BIT10 = 1) THEN y = y | NegMask      ' Store 11-bits as signed word
  IF (x.BIT10 = 1) THEN x = x | NegMask      ' Repeat for other axis

  RETURN
```

Your Turn

Try extending the Your Turn from Activity #4 with the Parallax Serial LCD. Don't worry about adding LCD functionality to the calibration program, just to the modified calibration test program from Activity #3. You may need to recalibrate to eliminate magnetic interference caused by the close proximity of the LCD. Add a command to the CalibrateCompass.bs2 program's Main Routine that sends a few characters to the LCD each time through the loop before you run it. To free up some code space, try removing a few characters from one of the **`DEBUG`** commands in the **`Calibration_Review`** subroutine.

SUMMARY

The Hitachi HM55B Compass module is a dual-axis magnetic field sensor capable of detecting microtesla variations in the components of the earth's magnetic field acting on its x and y axes. The module's angle from north can be determined by dividing the x axis measurement into the −y axis measurement, and then taking the arctangent of the result. The compass module can also be used to detect magnetic fields from bar magnets as well as the inclination of the earth's magnetic field at your locale.

The BASIC Stamp can store measured directions in the unused portion of its EEPROM program memory with a calibration program. Then, a test program can access these saved values, and use them to perform scale correction and linear interpolation on the measurements. These correction techniques can significantly improve the direction measurement. By averaging the axis measurements, the direction measurement can be further refined. The direction can be displayed in text format on the Parallax Serial LCD by adding a small initialization routine to the program's Initialization section and a **SEROUT** command to the program's main **DO**...**LOOP**.

Questions

1. What's the relationship between the compass module's x-axis measurement at a given angle and the measurement when the x-axis is aligned with magnetic north?
2. What are the names of each of the compass module's pins that have to be connected to BASIC Stamp I/O pins?
3. Which way does the angle from north increase in conventional compasses?
4. What's declination?
5. If you are measuring a magnetic field next to a bar magnet, how does the direction of the magnetic field relate to its N and S markings?
6. Why can a nearby mechanical compass cause errors in the compass module's measurements?
7. How would you average twenty measurements?
8. What variables do you have to set to zero before averaging the x and y measurements? Why do they have to be set to zero?
9. What's the **SEROUT** command for defining the degree symbol? How many bytes does it send?
10. If the number of digits in a displayed measurement might change, how do you prevent ghost characters from appearing at the right?

Exercises

1. Calculate the angle from north if the x axis reading is 34 and the y-axis reading is 0.
2. Calculate the angle from north if the x-axis reading is 16 and the y-axis reading is 31.
3. Calculate the number of nanoteslas in 1.6 microteslas.
4. Write a routine that converts from microteslas to nanoteslas.
5. Write a routine that examines a variable and displays whether or not it is negative.

Projects

1. Display the current heading on the serial LCD. Press and release a pushbutton to remember that desired heading. If the heading is off by more than 5°, beep a warning with a piezospeaker.
2. Design a prototype that can tell you when your Hitachi HM55B Digital Compass is held level with the aid of the Memsic 2125 Accelerometer.

Solutions

Q1. If N is the value reported by x when it is aligned with magnetic north, then for a given angle theta, $x = N\cos\theta$.

Q2. Din, Dout, /Enable, and CLK.

Q3. Clockwise from north.

Q4. Declination is the difference, in degrees, between magnetic north and true north.

Q5. The direction of the field will seem to be opposite of the N and S markings, because of the way the magnetic field wraps around.

Q6. The mechanical compass contains a small magnet which can affect the compass module.

Q7. Keep a running sum of all twenty measurements, then divide the total sum by 20.

Q8. The running sum of both x and y must start at zero. Once a calculation is done, the sums will contain a large number. To do a second calculation, you must reset the sum to zero.

Q9. The command is 255, followed by 8 bytes of character data.

```
SEROUT pinNumber, baudrate, [255, byte0…byte7]
```

Q10. First print spaces to overwrite (blank out) the previous value, then place the cursor back at the starting point and print the new value.

E1. $\theta = 0°$ degrees from North, or due North.

E2. $\theta = 297.3°$ from North.

E3. 1.6 microtesla = 1.6×10^{-6} T = 1600×10^{-9} T = 1600 nanotesla

E4. Example routine:

```
Convert:
  nanoT = 1000 * microT
  RETURN
```

E5. Example routine:

```
value          VAR     Word

IF (value.BIT15 = 1) THEN
  DEBUG "Negative", CR
ELSE
  DEBUG "Positive", CR
ENDIF
```

P1. Example solution:

Assuming an active-low pushbutton is connected to P10 and a piezospeaker is connected to P11 (See *What's a Microcontroller?*, Chapters 3 and 8), LcdTestCompass.bs2 can then be modified as follows:

Add these pin directives to the I/O definitions section:

```
pushbutton     PIN      10
speaker        PIN      11
```

Add these variables to the Variables Section:

```
angleMem          VAR      Word
difference        VAR      Word
alarmArm          VAR      Bit
```

Modify the last **SEROUT** command in the initialization routine:

```
SEROUT 14, 84, [128, "Alarm Angle Set ",    ' Static characters
                148, "OFF             "]
```

Modify the main routine as shown here:

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

DO                                        ' Main loop

  GOSUB Compass_Get_Axes                  ' Get x, and y values

  angle = x ATN -y                        ' Convert x and y to brads
  angle = angle */ 361                    ' Convert brads to degrees

  ' LCD Display heading in degrees on top line and x and y
  ' measurements on the bottom line.
  SEROUT 14, 84, [154, "    ", 154, DEC angle, 7]

  IF PushButton = 1 THEN
    angleMem = angle
    alarmArm = 1
    FREQOUT speaker, 20, 3000
    SEROUT 14, 84, [148,"ON " ]
  ENDIF

  difference = ABS(angle - angleMem)

  IF alarmArm = 1 THEN
    SEROUT 14, 84, [160, "    ", 160, DEC angleMem, 7]
    IF difference > 5 AND difference < 355 THEN
      FREQOUT speaker, 10, 4000
    ENDIF
  ENDIF

  PAUSE 150                               ' Debug delay for slower PCs

LOOP                                      ' Repeat main loop
```

P2. Example solution: Ch4_Project2.bs2
This program is a combination of HorizontalTilt.bs2 and TestCompass.bs2.

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - Ch4_Project2.bs2
' Display digital compass and tilt measurements with one program.

' {$STAMP BS2}                              ' BASIC Stamp Directive
' {$PBASIC 2.5}                             ' PBASIC Directive

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

Negative       CON     1                   ' Sign - .bit15 of Word variables
Positive       CON     0

' -----[ I/O Definitions ]-------------------------------------------------

DinDout        PIN      2                  ' P2 transceives to/from Din/Dout
Clk            PIN      0                  ' P0 sends pulses to HM55B's Clk
En             PIN      1                  ' P2 controls HM55B's /EN(ABLE)

Reset          CON      %0000              ' Reset command for HM55B
Measure        CON      %1000              ' Start measurement command
Report         CON      %1100              ' Get status/axis values command
Ready          CON      %1100              ' 11 -> Done, 00 -> no errors
NegMask        CON      %1111100000000000  ' For 11-bit negative to 16-bits

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

xTilt          VAR     Word                ' Memsic x-axis measurement
yTilt          VAR     Word                ' Memsic y-axis measurement

side           VAR     Word                ' trig subroutine variable
angleTilt      VAR     Word                ' result angle - degrees
sign           VAR     Bit                 ' Sign bit

xCompass       VAR      Word               ' x-axis data
yCompass       VAR      Word               ' y-axis data
status         VAR      Nib                ' Status flags
angleCompass   VAR      Word               ' Store angle measurement

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

DEBUG CLS                                  ' Clear Debug Terminal

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

DO
```

```
  PULSIN 6, 1, xTilt                        ' x-axis measurement
  PULSIN 7, 1, yTilt                        ' y-axis measurement

  ' Scale and offset x and y-axis values to -127 to 127.
  xTilt = (xTilt MIN 1875 MAX 3125) - 1875 ** 13369 - 127
  yTilt = (yTilt MIN 1875 MAX 3125) - 1875 ** 13369 - 127

  ' Calculate and display Arcsine of x-axis measurement.
  side = xTilt
  GOSUB Arcsine
  DEBUG HOME, "x tilt angle = ", CLREOL, SDEC3 angleTilt, CR

  ' Calculate and display Arcsine of y-axis measurement.
  side = yTilt
  GOSUB Arcsine
  DEBUG "y tilt angle = ", CLREOL, SDEC3 angleTilt

  GOSUB Compass_Get_Axes                    ' Get x, and y values

  angleCompass = xCompass ATN -yCompass     ' Convert x and y to brads
  angleCompass = angleCompass */ 361        ' Convert brads to degrees

  DEBUG CR, "Compass angle = ",
        DEC angleCompass, " degrees", CLREOL

  PAUSE 150                                 ' Debug delay for slower PCs

LOOP                                        ' Repeat DO...LOOP

' -----[ Subroutine - Arcsine ]-------------------------------------------------

' This subroutine calculates arcsine based on the y coordinate on a circle
' of radius 127.  Set the side variable equal to your y coordinate before
' calling this subroutine.

Arcsine:                                    ' Inverse sine subroutine
  GOSUB Arccosine                           ' Get inverse cosine
  angleTilt = 90 - angleTilt                ' sin(angle) = cos(90 - angle)
  RETURN

' -----[ Subroutine - Arccosine ]-----------------------------------------------

' This subroutine calculates arccosine based on the x coordinate on a circle
' of radius 127.  Set the side variable equal to your x coordinate before
' calling this subroutine.

Arccosine:                                  ' Inverse cosine subroutine
  sign = side.BIT15                         ' Save sign of side
  side = ABS(side)                          ' Evaluate positive side
  angleTilt = 63 - (side / 2)               ' Initial angle approximation
  DO                                        ' Successive approximation loop
```

```
    IF (COS angleTilt <= side) THEN EXIT     ' Done when COS angle <= side
    angleTilt = angleTilt + 1                ' Keep increasing angle
  LOOP
  angleTilt = angleTilt */ 361               ' Convert brads to degrees
  IF sign = Negative THEN                    ' Adjust if sign is negative.
    angleTilt = 180 - angleTilt
  ENDIF
  RETURN

  ' -----[ Subroutine - Compass_Get_Axes ]-----------------------------------

Compass_Get_Axes:                            ' Compass module subroutine

  HIGH En: LOW En                            ' Send reset command to HM55B
  SHIFTOUT DinDout,clk,MSBFIRST,[Reset\4]

  HIGH En: LOW En                            ' HM55B start measurement command
  SHIFTOUT DinDout,clk,MSBFIRST,[Measure\4]
  status = 0                                 ' Clear previous status flags

  DO                                         ' Status flag checking loop
    HIGH En: LOW En                          ' Measurement status command
    SHIFTOUT DinDout,clk,MSBFIRST,[Report\4]
    SHIFTIN  DinDout,clk,MSBPOST,[Status\4]  ' Get Status
  LOOP UNTIL status = Ready                  ' Exit loop when status is ready

  ' Get x & y axis values
  SHIFTIN  DinDout,clk,MSBPOST,[xCompass\11,yCompass\11]
  HIGH En                                    ' Disable module
  ' Store 11-bits as signed words for both axes
  IF (yCompass.BIT10 = 1) THEN yCompass = yCompass | NegMask
  IF (xCompass.BIT10 = 1) THEN xCompass = xCompass | NegMask

  RETURN
```

Chapter 5: Accelerometer Gaming Basics

Chapter 3 introduced you to the Memsic Dual-Axis Accelerometer. Similar devices can be found in lots of HIDs (Human Interface Devices), a category which includes computer mice, keyboards, and more generally, anything that makes it possible for humans to interact with microprocessors. With limited space on PDAs (Personal Digital Assistants) like the one in Figure 5-1, tilt control eliminates the need for extra buttons. In this example, tilting allows the user to pan around on a map without pushing any buttons. Tilt control is also a popular feature in certain game controllers.

Figure 5-1:Tilt-Controlled PDA



Photos of RotoView® tilt-controlled PDA interface in action courtesy of Innoventions®, www.innoventions.com

This chapter has four activities that demonstrate the various facets of using tilt to control a display. Here are summaries of each activity:

- **Activity #1: PBASIC Graphic Character Display** – introduces some Debug Terminal cursor control and coordinate-plotting basics.
- **Activity #2: Background Store and Refresh with EEPROM** – Each time your game character moves, whatever it was covering up on the screen has to be re-drawn. This activity demonstrates how you can move your character and refresh the background with the help of the BASIC Stamp's EEPROM.
- **Activity #3: Tilt the Bubble Graph** – With a moving asterisk on a graph, this first application illustrates how the hot air pocket inside the MX2125 moves when you tilt it. At the same time, it puts the accelerometer fundamentals to work along with the techniques from Activity #2.

- **Activity #4: Game Control** – You are now ready to use tilt to start controlling your game character. The background characters can be used to make decisions about whether your game character is in or out of bounds. Have fun customizing and expanding this tilt-controlled video game.

ACTIVITY #1: PBASIC GRAPHIC CHARACTER DISPLAY

This activity introduces some programming techniques you will use to graphically display coordinates with the Debug Terminal. Certain elements of the techniques introduced in this activity and the next will be familiar from the previous Accelerometer and LCD chapters.

The CRSRXY and Other Control Characters

The **DEBUG** command's **CRSRXY** control character can be used to place the cursor at a specific location on the Debug Terminal's Receive windowpane. For example, **DEBUG CRSRXY, 7, 3, "*"** places the asterisk character seven spaces to the right and three characters down. Instead of using constants like 7 and 3, you can use variables to make the placement of the cursor adjustable. Let's say you have two variables named **x** and **y**. The values these variables store can control the placement of the asterisk in the command **DEBUG CRSRXY, x, y, "*"**.

The next example program also makes use of the **CLRDN** control character. The command **DEBUG CLRDN** causes all the lines below the cursor's current location to be erased.



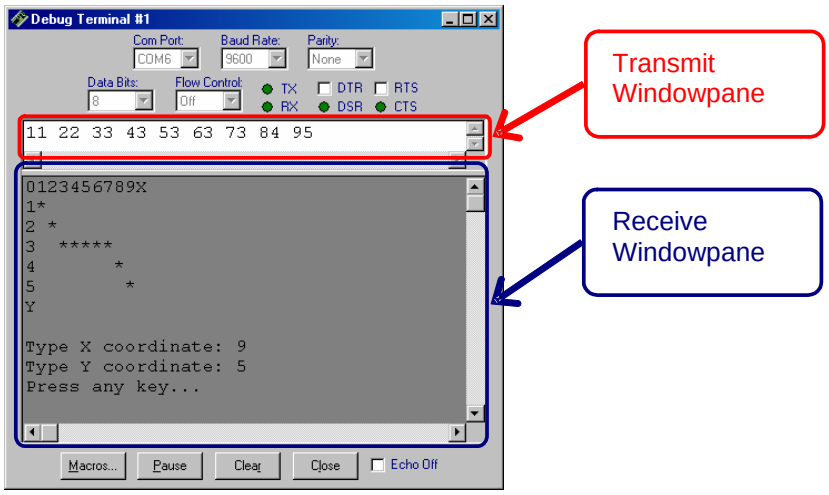
More Control Characters

You can find out more about control characters by looking up the **DEBUG** command, either in the PBASIC Syntax Guide or the BASIC Stamp Manual. You can get to the PBASIC Syntax Guide through your BASIC Stamp Editor (v2.0 or newer). Just click Help and select Index. The BASIC Stamp Manual is available for download from www.parallax.com.

Example Program – CrsrxyPlot.bs2

With this program, you can type pairs of digits into the Transmit windowpane as in Figure 5-2, to position asterisks on the Receive windowpane. Simply click the Transmit windowpane and start typing. The first digit you type is the number of spaces to the right to place the cursor, and the second number is the number of carriage returns downward. Before typing a new pair of digits, press the space bar once.

Figure 5-2: Debug Terminal Transmit and Receive Windowpanes



- √ Enter, save, and run CrsrxyPlot.bs2.
- √ Resize the Debug Terminal so there is ample room to display both the plot area and the prompts.
- √ Click in the Debug Terminal's Transmit windowpane, and follow the prompts to type digits to place asterisks on the plot.
- √ Try the sequence 11, 22, 33, 43, 53, 63, 73, 84, 95. Do the asterisks in your Debug Terminal match the pattern in Figure 5-2?
- √ Try predicting the sequences for various shapes, like a square, triangle, and circle.
- √ Enter the sequences to test your predictions.
- √ Correct the sequences as needed.

```
' Smart Sensors and Applications – CrsrxyPlot.bs2
' Type pairs of digits into the Debug Terminal to position asterisks.

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

x       VAR   Word
y       VAR   Word
temp    VAR   Byte
```

```
DEBUG CLS,
"0123456789X", CR,
"1          ", CR,
"2          ", CR,
"3          ", CR,
"4          ", CR,
"5          ", CR,
"Y          ", CR, CR

DO

  DEBUG "Type X coordinate: "
  DEBUGIN DEC1 x
  DEBUG CR, "Type Y coordinate: "
  DEBUGIN DEC1 y

  DEBUG CRSRXY, x, y, "*"

  DEBUG CRSRXY, 0, 10, "Press any key..."
  DEBUGIN temp
  DEBUG CRSRXY, 0, 8, CLRDN

LOOP
```

Your Turn – Keeping Characters in the Plot Area

If you type the digit 8 in response to the prompt `"Type Y coordinate: "`, it will overwrite your text. Similar problems occur if you type 0 for either the X or Y coordinates. The asterisk is plotted over the text that shows which row and column **`CRSRXY`** is plotting. One way to fix this is with the **`MAX`** and **`MIN`** operators. Simply add the statement **`y = y MAX 5 MIN 1`**. The **`DEBUGIN`** command's **`DEC1`** operator solves this problem for the maximum X coordinate, since it is limited to a value from 0 to 9. So, all you'll need to clamp the X value is **`x = x MIN 1`**.

- √ Try entering out-of-bounds values for the Y coordinate (0 and 6 to 9) and 0 for the X coordinate.
- √ Observe the effects on the display's background.
- √ Modify CrsrxyPlot.bs2 as shown here and try it again

```
DEBUG CR, "Type Y coordinate: "
DEBUGIN DEC1 y

Y = y MAX 5 MIN 1               ' <--- Add
X = x MIN 1                     ' <--- Add

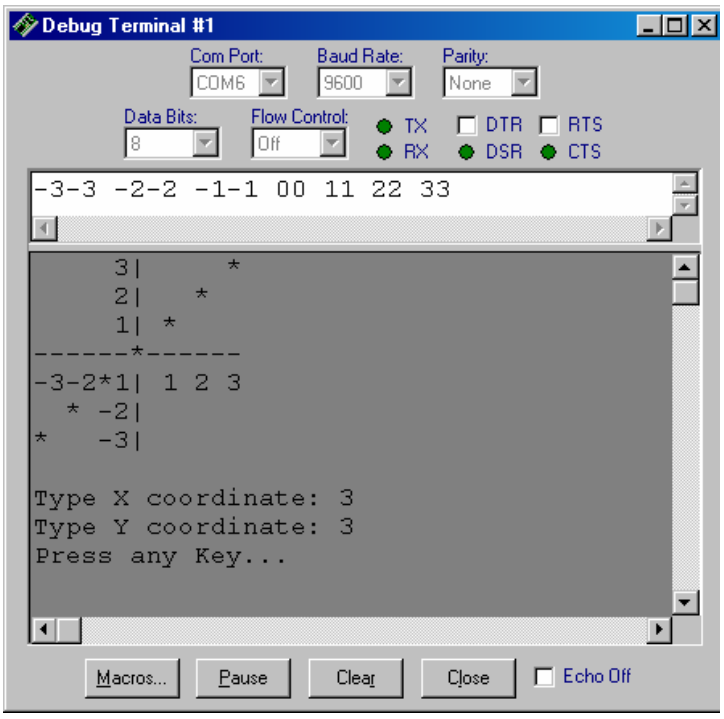
DEBUG CRSRXY, x, y, "*"
```

Scale and Offset

Scale and offset were introduced in Chapter 3 for managing input values from the accelerometer. In Chapter 3, we used the ****** operator for scaling (multiplying) by fractional values. In this example, we'll just use the ***** operator because a variable in the next example program only needs to be multiplied by the integer value 2.

Take a look at Figure 5-3, where a graph with both positive and negative axes has been printed in the Debug Terminal. The horizontal, or x, axis has a space between each numeral, and the vertical, y, axis does not. In order to position the cursor in a particular spot using the **CRSRXY** command, we'll need mapping between the axes printed in the Debug Terminal and the axes used by the **CRSRXY** command.

Figure 5-3: Entering and Displaying Coordinates



For example, where the axes of the graph intersect at coordinates (0, 0) is actually **CRSRXY** position 6,3. (Compare this with Figure 5-2 until you see how this is so.) For this program we would like to be able to type "-3-3" in to the Debug Terminal and have the asterisk appear at graph coordinate (-3, -3), which would be **CRSRXY** position 0,6. As another example, when you type in 2,2, **CRSRXY** actually needs to position the asterisk at 10,1. Now, it's time to figure out how to do this mapping translation using scale and offset.

For values ranging from -3 to 3, the X value has to be multiplied by 2 and added to 6 for **CRSRXY** to place the asterisk the right number of spaces over. That's a scale of 2, and an offset of 6. Here is a PBASIC statement to make the conversion from X coordinate to number of spaces.

```
x = (x * 2) + 6
```

The Y value has to be multiplied by -1, then added to 3. That's a scale of -1 and an offset of 3. Here is a PBASIC statement to make the conversion from Y coordinate to number of carriage returns.

```
y = 3 - y
```

√ Try substituting X and Y coordinates in the right side of each of these equations, do the math, and verify that each equation yields the right number of spaces and carriage returns.

Example Program – PlotXYGraph.bs2

√ Enter and run PlotXYGraph.bs2.
√ Try entering the sequence of values: -3-3, -2-2, -1-1, 00, 11, 22, 33, and verify that it matches the Debug Terminal example in Figure 5-3.
√ Try some other sequences and/or drawing shapes by their coordinates.

```
' Smart Sensors and Applications – PlotXYGraph.bs2
' Position cursor on plot interactively in Debug Terminal

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

x               VAR     Word
y               VAR     Word
temp            VAR     Byte
```

```
DEBUG CLS,
"     3|      ", CR,
"     2|      ", CR,
"     1|      ", CR,
"------+------", CR,
"-3-2-1| 1 2 3", CR,
"    -2|      ", CR,
"    -3|      ", CR, CR

DO

  DEBUG "Type X coordinate: "
  DEBUGIN SDEC1 x
  DEBUG CR, "Type Y coordinate: "
  DEBUGIN SDEC1 y

  x = (x * 2) + 6
  y = 3 - y

  DEBUG CRSRXY, x, y, "*"

  DEBUG CRSRXY, 0, 10, "Press any Key..."
  DEBUGIN temp
  DEBUG CRSRXY, 0, 8, CLRDN

LOOP
```


Your Turn – More Keeping Characters in the Plot Area

The **MAX** and **MIN** operators were introduced earlier to prevent the asterisk from appearing outside the display area. You can also use **IF…THEN** statements to handle values that are out of bounds. Here is an example of how you can modify PlotXYGraph.bs2 with **IF…THEN**. Instead of clipping the values and placing the asterisk within the allowed boundaries, this program just waits until a correct value is entered.

Modify PlotXYGraph.bs2 by replacing the **DEBUG CRSRXY, x, y, "*"** instruction with the **IF**...**THEN**...**ELSE**...**ENDIF** block shown below, and then run it.

```
x = (x * 2) + 6
y = 3 – y

IF (x > 12) OR (y > 6) THEN              ' <--- Add code from here...
  DEBUG CRSRXY, 0, 8, CLRDN,             '
        "Enter values from -3 to 3.", CR, '
        "Try again"                      '
                                         '
ELSE                                     '
                                         '
```

```
DEBUG CRSRXY, x, y, "*"
                                                '
ENDIF                                           ' <--- to here

DEBUG CRSRXY, 0, 10, "Press any Key..."
DEBUGIN temp
```

√ Verify that this program does not allow you to enter characters outside the range of -3 to 3.



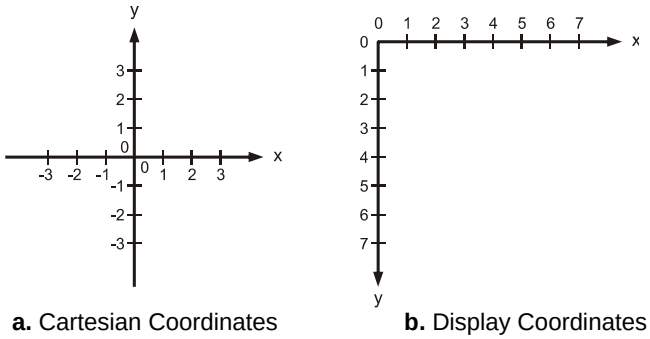
What about negative numbers?

The conditions for the **IF...THEN** statement in your modified version of PlotXYGraph.bs2 are (x > 12) OR (y > 6). This covers positive numbers that are larger than 12 or 6, but it also covers all negative numbers. That's because the BASIC Stamp uses a system called two's complement to store negative numbers. In two's complement, the unsigned version of any negative value is larger than any positive value. For example, -1 is 65535, -2 is 65534, and so on, down to -32768, which is actually 32768. Signed positive values only range from 1 to 32767.

Algebra to Determine Scale and Offset

The XY plot displayed in the Debug Terminal in this activity is called the Cartesian coordinate system. Named after 17th century mathematician René Descartes, this system is the basis for graphing techniques used in many mathematical pursuits. Shown in Figure 5-4a, the Cartesian coordinate system is most commonly displayed with (0, 0) in the center of the graph. Its values get larger going upward (y-axis) and to the right (x-axis). Many displays behave differently, with coordinate 0, 0 starting at the top-left as in Figure 5-4b. While the x-axis increases toward the right, the y-axis increases downward.

Figure 5-4: Cartesian vs. Display Coordinates



a. Cartesian Coordinates **b.** Display Coordinates

You can use a standard algebra technique, solving two equations in two unknowns, to figure out the statements you will need to transform Cartesian coordinates into display coordinates for the Debug Terminal. This next example shows how it was done for the statements that converted x and y from Cartesian to display coordinates in PlotXYGraph.bs2.

By adding a couple of **DEBUG** commands to PlotXYGraph.bs2, you can display the before and after versions of the X-value you entered.


```
DEBUG "Type X coordinate: "
DEBUGIN SDEC1 x
DEBUG CR, "Type Y coordinate: "
DEBUGIN SDEC1 y

DEBUG CRSRXY, 0, 12, "x before: ", SDEC x   ' <--- Add

x = (x * 2) + 6
y = 3 - y

DEBUG CRSRXY, 0, 14, "x after:  ", SDEC x   ' <--- Add

DEBUG CRSRXY, x, y, "*"
```

- √ Save PlotXyGraph.bs2 under another name, like PlotXyGraphBeforeAfter.bs2.
- √ Add the two **DEBUG** commands that display the "before" and "after" values of x.
- √ Add two more **DEBUG** commands to display the “before” and “after” values of y.
- √ Enter the coordinates (3,1) and (-2,-2) into the Debug Terminal's Transmit windowpane. See Figure 5-5.
- √ Record the After values in Table 5-1.

Table 5-1: X Values Before and After

Coordinates	Before	After
(3, 1)	3	
(-2, -2)	-2	

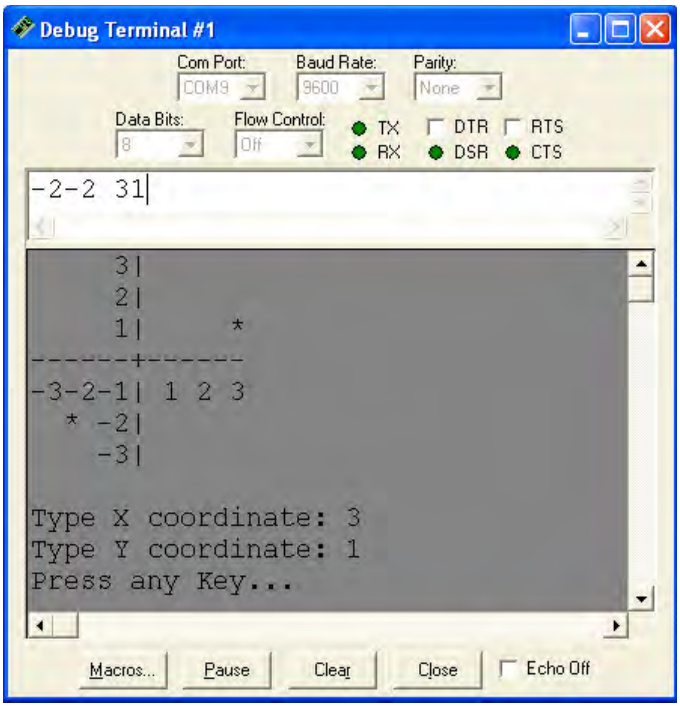


Figure 5-5
Test Coordinates

When designing a display to show Cartesian coordinates, it helps to take a couple of before and after values like the ones in Table 5-1. You can then use them to solve for scale (K) and offset (C) using two equations with two unknowns.

$$X_{after} = (K \times X_{before}) + C$$

The usual steps for two equations in two unknowns are:

(1) Substitute your two before and after data points into separate copies of the equation.

$$12 = (K \times 3) + C$$

$$2 = (K \times -2) + C$$

(2) If needed, multiply one of the two equations by a term that causes the number of one of the unknowns in the top and bottom equations to be equal.

Not needed, because the coefficient of C in both equations is 1.

(3) Subtract one equation from the other to make one of the unknowns zero.

$$\begin{array}{r} 12 = (K \times 3) + C \\ -\left[\, 2 = (K \times -2) + C \,\right] \\ \hline 10 = K \times 5 \end{array}$$

(4) Solve for the unknown that did not subtract to zero.

$$10 = K \times 5$$
$$K = \frac{10}{5}$$
$$K = 2$$

(5) Substitute the value you solved in step 4 into one of the original two equations.

$$12 = (2 \times 3) + C$$

(6) Solve for the second unknown.

$$12 = (2 \times 3) + C$$
$$12 = 6 + C$$
$$C = 12 - 6$$
$$C = 6$$

(7) Incorporate solved unknowns into your equation.

$$X_{after} = (K \times X_{before}) + C$$
$$K = 2 \text{ and } C = 6$$
$$X_{after} = (2 \times X_{before}) + 6$$

Your Turn – Y-Axis Calculations

√ Modify your program so that it displays the Y-Axis before and after values.
√ Fill in Table 5-2 for the Y-axis values:

Table 5-2: Y Values Before and After		
Coordinates	Before	After
(3, 1)	1	
(-2, -2)	-2	

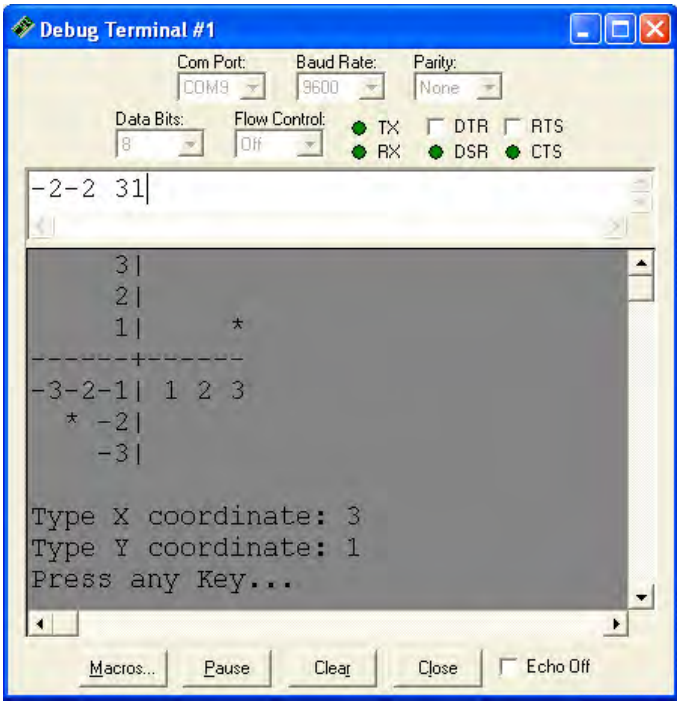


Figure 5-6
Test Coordinates

√ Repeat steps 1 through 7 for the Y-Axis equation. The correct answer is:

$$y_{after} = (-1 \times y_{before}) + 3$$

ACTIVITY #2: BACKGROUND STORE AND REFRESH WITH EEPROM

In a video game, when your game character isn't on the screen, all that's visible is the background. As soon as your game character enters the screen, it blocks out part of the background. When the character moves, two things have to happen: (1) the game character has to be re-drawn at the new location, and (2) the background that the game character was blocking out has to be re-drawn. If step 2 never happened in your program, your screen would fill up with copies of your game character.

Televisions and CRT computer monitors refresh every pixel many times per second. The refresh rate on televisions is around 30 Hz, and a few of the more common refresh rates on CRTs are 60, 70, and 72 Hz. Other devices like certain LCD and LED displays hold the image automatically, or sometimes with the help of another microcontroller. All the program or microcontroller that controls these devices has to do is tell them what to display or change. This is also how video compression on your computer works. In order to reduce the file size, some compressed video files store the changes to the image instead of all the pixels in a given image frame.

When used with displays that do not need to be refreshed (like the Debug Terminal or an LCD), the BASIC Stamp EEPROM can store an image of a game or graph background in its EEPROM. When a game character moves and is redrawn at a different location, the BASIC Stamp can just read its EEPROM and redraw the background characters at the game character's old location. To do this, simply save the old coordinates of the game character before it moved and then use those coordinates to retrieve the background character from EEPROM. Depending on how large the display is, this can save a considerable amount of time that the BASIC Stamp might need to perform other tasks.

This activity introduces three elements to game characters and backgrounds:

1. Storing and displaying the background from EEPROM
2. Tracking a character's old and new coordinates
3. Redrawing the old coordinates from EEPROM

Background Display from EEPROM

This display doesn't have to be made with a single **`DEBUG`** command, especially if it needs to be maintained as a background with characters traveling over it in the foreground. Instead, it's better to store the characters in EEPROM and then display them

individually with a **FOR…NEXT** loop that uses **READ** and **DEBUG** commands to display individual characters. Figure 5-7 is a display generated with this technique.

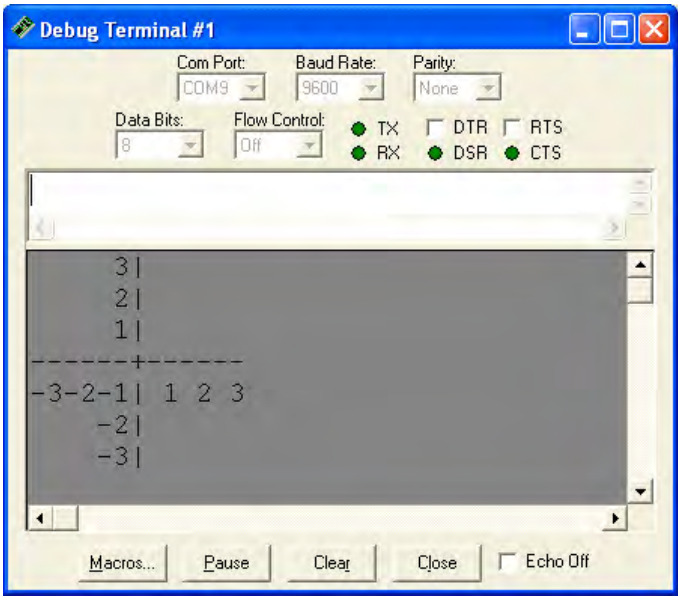


Figure 5-7
Background
from DATA Directive

You can use the **DATA** directive to store a background in EEPROM. Notice how this **DATA** directive stores 100 characters (0 to 99). Notice also that each row is 14 characters wide when you add the **CR** control character. It makes programming much easier if each row is the same width. Otherwise, finding the character you want becomes a more complex problem.

```
DATA CLS,                       ' 0
"     3|      ", CR,            ' 14
"     2|      ", CR,            ' 28
"     1|      ", CR,            ' 42
"------+------", CR,            ' 56
"-3-2-1| 1 2 3", CR,            ' 70
"    -2|      ", CR,            ' 84
"    -3|      ", CR, CR         ' 98 + 1 = 99
```

To display the entire background once at the beginning of the program, you can then use a **FOR…NEXT** loop. It retrieves and displays each character stored in EEPROM. Keep in mind that, while the net effect is the same as a long **DEBUG** command, the EEPROM is

more flexible because you can also fetch and display individual characters as needed to refresh the background.

```
FOR index = 0 TO 99
  READ index, character
  DEBUG character
NEXT
```

Example Program – EepromBackgroundDisplay.bs2

- √ Enter, save, and run the program.
- √ Verify that the display is the same as PlotXyGraph.bs2.

```
' Smart Sensors and Applications - EepromBackgroundDisplay.bs2
'

'{$STAMP BS2}                                ' Stamp & PBASIC Directives
'{$PBASIC 2.5}

index          VAR     Byte                  ' Variables
character      VAR     Byte

DATA CLS,                      ' 0           ' Store background in EEPROM
"     3|      ", CR,           ' 14
"     2|      ", CR,           ' 28
"     1|      ", CR,           ' 42
"------+------", CR,           ' 56
"-3-2-1| 1 2 3", CR,           ' 70
"    -2|      ", CR,           ' 84
"    -3|      ", CR, CR        ' 98 + 1 = 99

FOR index = 0 TO 99                          ' Retrieve and display background
  READ index, character
  DEBUG character
NEXT

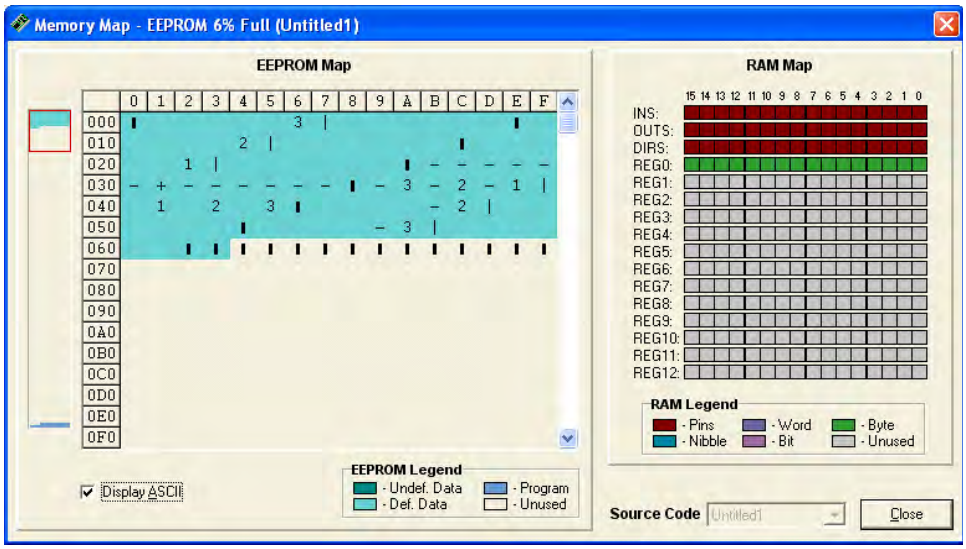
END
```

Your Turn – Viewing the EEPROM Characters

- √ In the BASIC Stamp Editor, click Run and select Memory Map.
- √ Click the Display ASCII box in the lower left corner of the Memory Map window.
- √ The digits, dashes, and vertical bars should appear in the EEPROM Map exactly as shown in Figure 5-8.

√ Instead of 14 characters per row, the EEPROM Map shows 16. Verify that you have a total of 100 (0 to 99) characters stored for display purposes in EEPROM.

Figure 5-8: Display Characters Stored in EEPROM



Tracking a Character's Old and New Coordinates

Let's say you want to track the previous X and Y coordinates in the original unmodified PlotXYGraph.bs2 from Activity #1. It takes two steps:

(1) Declare a couple of variables for storing the old values, **xOld** and **yOld** for example.

```
x               VAR     Word
y               VAR     Word

xOld            VAR     Nib                 ' <--- Add
yOld            VAR     Nib                 ' <--- Add

temp            VAR     Byte
```

(2) Before loading new values into the **x** and **y** variables, store the current value of **x** into **xOld** and the current value of **y** into **yOld**.

```
DO

  xOld = x                                      ' <--- Add
  yOld = y                                      ' <--- Add

  DEBUG "Type X coordinate: "
```



Why are x and y words while xOld and yOld are nibbles?

When working with signed values, word variables store both the value and the sign.

At the particular place that **xOld** and **yOld** are used in the program, they are only storing values that range from 0 to 12, so all we need are nibble variables.

Here's a third step you can use to test and verify that it works:

(3) Add **DEBUG** statements to display the current and previous values of x, y.

```
DEBUG CRSRXY, x, y, "*"

  DEBUG CRSRXY, 0, 10,                          ' <--- Add
        "Current entry:  (",
        DEC x, ",", DEC y, ")"
  DEBUG CRSRXY, 0, 11,                          ' <--- Add
        "Previous entry: (",
        DEC xOld, ",", DEC yOld, ")"
  DEBUG CRSRXY, 0, 12, "Press any Key..."       ' <--- Modify

  DEBUGIN temp
```

√ Start with the original, unmodified version of PlotXYGraph.bs2, save it under the name PlotXYGraphRecall.bs2, and try the modifications just discussed in steps (1) through (3) above. Keep in mind that both values displayed will be in terms of Debug Terminal coordinates. Also keep in mind that the first time through, the old coordinates will be (0, 0) since all variables initialize to zero in PBASIC.

Re-Drawing the Background

Up to this point, all of our plots accumulated asterisks as we entered more values in the Transmit windowpane. The net effect we want for game control is to make the asterisk disappear from its old location and appear in its new location whenever we redefine it, to give the appearance of one asterisk moving.

Take a look at Figure 5-9. Notice that six ordered pairs were entered into the Debug Terminal, but there is only one asterisk, and it corresponds with the last pair that was entered. That's because the program used here makes the old asterisk disappear by taking the old x, y coordinates to look up the background character from EEPROM, and then displaying it with **DEBUG**. To make the asterisk appear at its new location, it simply uses a **DEBUG** command with the current x, y coordinates as did our previous example programs.

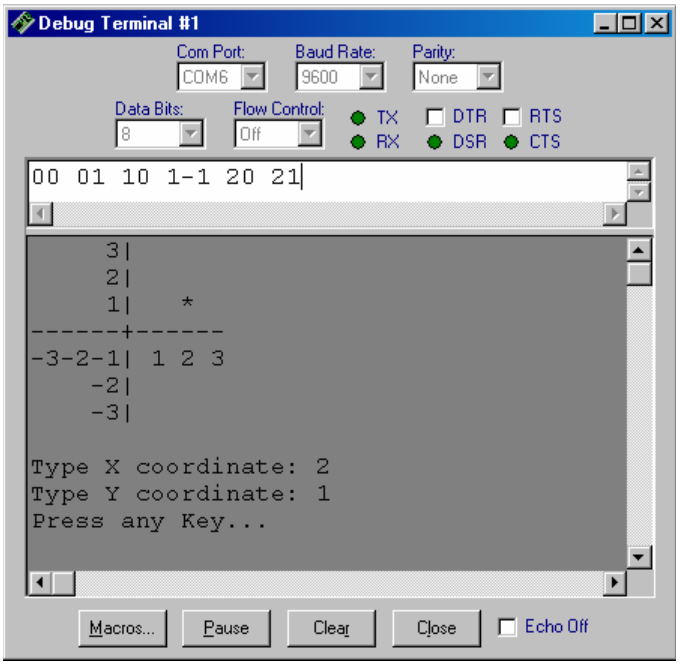


Figure 5-9
Display using EEPROM Background Refresh

The program used to create Figure 5-9 combines the **DATA**-defined background technique from EepromBackgroundDisplay.bs2 with the asterisk plotting and location tracking technique from PlotXYGraphRecall.bs2. This combination allows us to redraw the background character over the old asterisk with this code:

```
IF (x <> xold) AND (y <> yold) THEN      ' Check if asterisk moved
  index = (14 * yOld) + xOld + 1         ' Background character address
  READ index, character                  ' Get background character
  DEBUG CRSRXY, xOld, yOld, character    ' Display background character
ENDIF
```

The **index** variable selects the correct character from EEPROM. The x value is the number of spaces over and the y value is the number of carriage returns down. To get to the correct address of a character on the third row, your program has to add all the characters in the first two rows. Since each row has 14 characters, **yOld** has to be multiplied by 14 before it can be added to **xOld**. The extra 1 is added to skip the **CLS** at address 0.

Regardless of whether it's a computer display, the liquid crystal display on your cell phone, or your BASIC Stamp application's display, the same technique applies. The processor remembers two different images, the one in the background, and the one in the foreground. As the foreground object "moves" it is displayed in a different location and the area that the foreground object vacated is re-drawn.

The most important thing to keep in mind about this programming technique is that it saves the processor lots of time. It only has to get one character from EEPROM and send it to the Debug Terminal. Compared to 99 characters, that's a significant time savings, and the BASIC Stamp can be doing other things with that time, such as monitoring other sensors, controlling servos, etc.

Example Program – EepromBackgroundRefresh.bs2

This program takes PlotXYGraph.bs2 combined with EepromBackgroundDisplay.bs2, using the background display, coordinate storage, and the background re-draw technique just discussed.

- √ Enter save and run EepromBackgroundRefresh.bs2.
- √ Test and verify that the asterisk disappears from its old location and appears at the new location you entered.

```
' -----[ Title ]-------------------------------------------------------------
' Smart Sensors and Applications - EepromBackgroundRefresh.bs2

'{$STAMP BS2}                                ' Stamp/PBASIC directives
'{$PBASIC 2.5}

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

x              VAR     Word                  ' Store current position
y              VAR     Word

xOld           VAR     Nib                   ' Store previous position
yOld           VAR     Nib
```

```
temp           VAR     Byte                  ' Dummy variable for DEBUGIN

index          VAR     Byte                  ' READ index/character storage
character      VAR     Byte

' -----[ EEPROM Data ]-----------------------------------------------------

DATA CLS,                                    ' Background data
"     3|      ", CR,           ' 14
"     2|      ", CR,           ' 28
"     1|      ", CR,           ' 42
"------+------", CR,           ' 56
"-3-2-1| 1 2 3", CR,           ' 70
"    -2|      ", CR,           ' 84
"    -3|      ", CR, CR        ' 98 + 1 = 99

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

FOR index = 0 TO 99                          ' Display background
  READ index, character
  DEBUG character
NEXT

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

DO

  xOld = x                                   ' Store previous coordinates
  yOld = y

  DEBUG "Type X coordinate: "                ' Get new coordinates
  DEBUGIN SDEC1 x
  DEBUG CR, "Type Y coordinate: "
  DEBUGIN SDEC1 y

  x = (x * 2) + 6                            ' Cartesian to DEBUG values
  y = 3 - y

  DEBUG CRSRXY, x, y, "*"                    ' Display asterisk

  IF (x <> xold) AND (y <> yold) THEN        ' Check if asterisk moved
    index = (14 * yOld) + xOld + 1           ' Background character address
    READ index, character                    ' Get background character
    DEBUG CRSRXY, xOld, yOld, character      ' Display background character
  ENDIF

  DEBUG CRSRXY, 0, 10, "Press any Key..."    ' Wait for user
  DEBUGIN temp
  DEBUG CRSRXY, 0, 8, CLRDN                  ' Clear old info

LOOP
```

Your Turn - Redrawing the Background without Extra Variables

Keeping track of the old location of the foreground character isn't always necessary. Think about it this way: in EepromBackgroundRefresh.bs2 the **x** and **y** variables store the old values until you enter new values. By simply rearranging the order in which the **x** and **y** variables are displayed, you can eliminate the need for **xOld** and **yOld**.

Next is a replacement Main Routine you can try in EepromBackgroundRefresh.bs2. As soon as you press the space bar, your old asterisk disappears. The new asterisk reappears when you type the second of the two coordinates. As you will see in the next activity, this technique works really well when the refresh rate is several times per second with tilt control.

- √ Save EepromBackgroundRefresh.bs2 as EepromBackgroundRefreshYourTurn.bs2.
- √ Comment out the **xOld** and **yOld** variable declarations.
- √ Replace the Main Routine in EepromBackgroundRefresh.bs2 with this one.
- √ Test it and examine the change in the program's behavior.

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

DO

  index = (14 * y) + x + 1              ' Redisplay background
  READ index, character
  DEBUG CRSRXY, x, y, character

  DEBUG CRSRXY, 0, 8,                   ' Get new coordinates
        "Type X coordinate: "
  DEBUGIN SDEC1 x
  DEBUG CR, "Type Y coordinate: "
  DEBUGIN SDEC1 y

  x = (x * 2) + 6                       ' Cartesian to DEBUG values
  y = 3 - y

  DEBUG CRSRXY, x, y, "*"               ' Display asterisk

  DEBUG CRSRXY, 0, 10, "Press any Key..."    ' Wait for user
  DEBUGIN temp
  DEBUG CRSRXY, 0, 8, CLRDN             ' Clear old info

LOOP
```

Animation and Redrawing the Background

Here is an example of something you can do if you use individual characters, but it won't work if you try to redraw the entire display with a **DEBUG** command.

- √ Save EepromBackgroundRefresh.bs2 as ExampleAnimation.bs2.
- √ Replace the Main Routine in the program with the one shown here.
- √ Run it and observe the effect.

```
DO
  FOR y = 0 TO 6
    FOR temp = 1 TO 2
      FOR x = 0 TO 12
        IF (temp.BIT0 = 1) THEN
          DEBUG CRSRXY, x, y, "*"
        ELSE
          index = (14 * yOld) + xOld + 1
          READ index, character
          DEBUG CRSRXY, xOld, yOld, character
          xOld = x
          yOld = y
        ENDIF
        PAUSE 50
      NEXT
    NEXT
  NEXT
LOOP
```

ACTIVITY #3: TILT THE BUBBLE GRAPH

This activity combines the graphics concepts introduced in Activities #1 and #2 with the accelerometer tilt measurement techniques introduced in Chapter 3. The result is an asterisk "bubble" that illustrates the movement of the heated gas pocket inside the MX2125's chamber. Figure 5-11 on the next page shows what the Debug Terminal in this activity displays when the accelerometer is tilted up and to the left.

Parts Required

(2) 3-inch Jumper Wires
(2) Resistors – 220 Ω
(1) Memsic MX2125 Dual-Axis Accelerometer

- √ Connect the accelerometer module using Figure 5-10 as your guide.

Figure 5-10: Accelerometer Schematic and Wiring Diagram

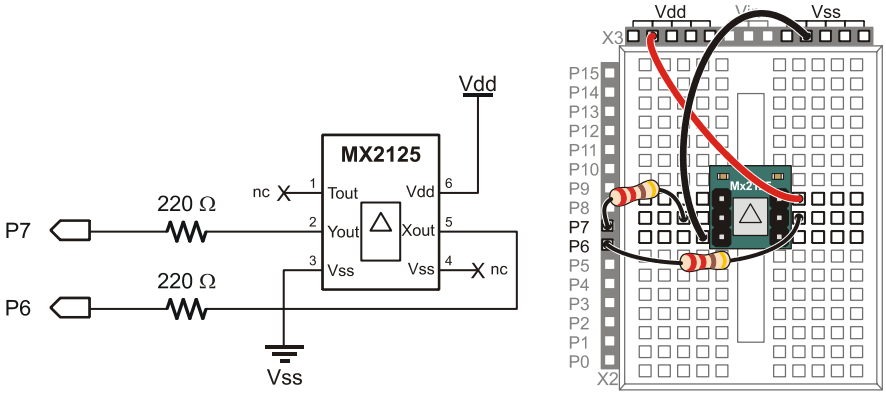


Figure 5-11: Accelerometer Hot Gas Location

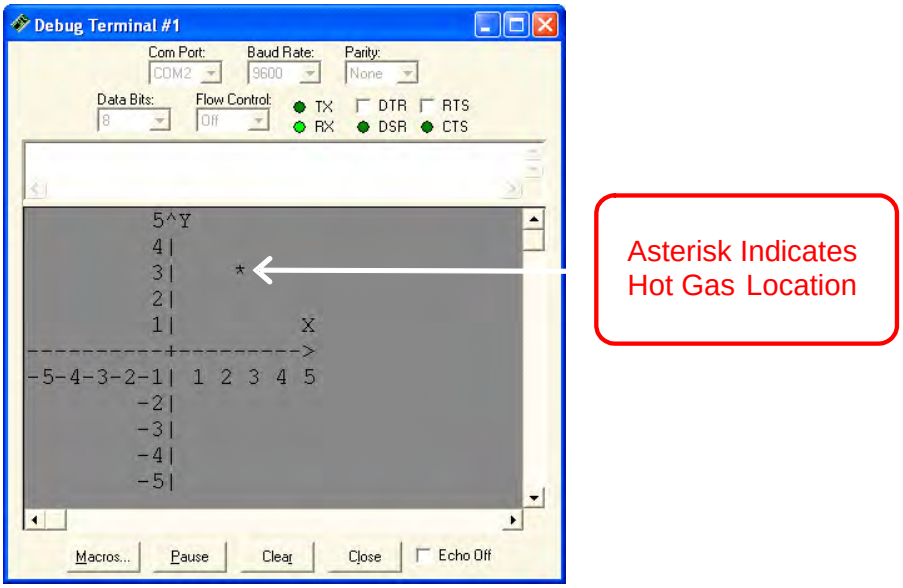


Figure 5-12 shows a legend for the different ways you can tilt the board on its axes along with each tilt's effect on the location of the hot gas pocket.

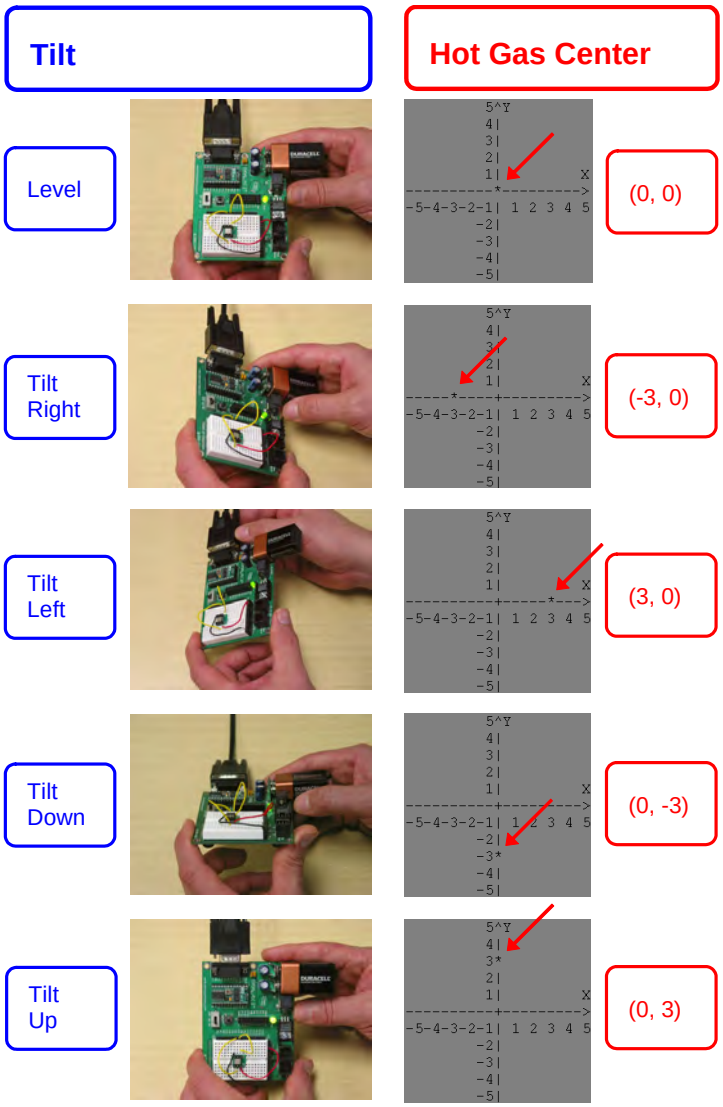


Figure 5-12: Accelerometer Tilt and Cursor Location

Tilt Control of Asterisk Display

BubbleGraph.bs2 updates the position of the hottest spot inside the accelerometer chamber about 8 times per second (8 Hz). After displaying the background (X and Y axes) to the Debug Terminal, it repeats the same steps over and over again.

- Replace the asterisk with the background character and pause for the blink-effect.
- Get the X-axis tilt from the accelerometer.
- Get the Y-axis tilt from the accelerometer.
- Adjust the value so that it fits on the plot's X-axis.
- Adjust the value so that it fits on the plot's Y-axis.
- Display the asterisk and pause again for the blink-effect.

Each of these steps is discussed in more detail in the section that follows the example program.

Example Program – BubbleGraph.bs2

√ Enter and run BubbleGraph.bs2.

```
' -----[ Title ]-----------------------------------------------------------------
' Smart Sensors and Applications - BubbleGraph.bs2
' Position of bubble in tilt sensor displays on graph in Debug Terminal

'{$STAMP BS2}                                  ' Stamp/PBASIC directives
'{$PBASIC 2.5}

' -----[ EEPROM Data ]-----------------------------------------------------------

' Store background to EEPROM                   ' Address of last char on row

DATA CLS,                                      '   0
"         5^Y         ", CR,                   '  22
"         4|          ", CR,                   '  44
"         3|          ", CR,                   '  66
"         2|          ", CR,                   '  88
"         1|         X", CR,                   ' 110
"----------+---------->", CR,                  ' 132
"-5-4-3-2-1| 1 2 3 4 5", CR,                   ' 154
"        -2|          ", CR,                   ' 176
"        -3|          ", CR,                   ' 198
"        -4|          ", CR,                   ' 220
"        -5|          ", CR                    ' 242
```

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

x       VAR    Word                         ' Store current position
y       VAR    Word

index   VAR     Word                        ' READ index/character storage
char    VAR     Byte

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

FOR index = 0 TO 242                        ' Read & display background
  READ index, char
  DEBUG char
NEXT

' -----[ Main Routine ]----------------------------------------------------
DO                                          ' Begin main routine

  ' Replace asterisk with background character.
  index = (22 * y) + x + 1                  ' Coordinates -> EEPROM address
  READ index, char                          ' Get background character
  DEBUG CRSRXY, x, y, char                  ' Display background character
  PAUSE 50                                  ' Pause for blink effect

  ' Measure tilt.
  PULSIN 6, 1, x                            ' Get Ax and Ay
  PULSIN 7, 1, y

  ' Calculate cursor position.
  x = (x MIN 1875 MAX 3125) - 1875 ** 1101   ' Calculate x position
  y = (y MIN 1875 MAX 3125) - 1875 ** 576    ' Calculate y position
  y = 10 - y

  ' Display asterisk at new cursor position.
  DEBUG CRSRXY, x, y, "*"                   ' Display asterisk
  PAUSE 50                                  ' Pause again for blink effect

LOOP                                        ' Repeat main routine
```

- √ Hold your board as shown in the top of Figure 5-12.
- √ Practice controlling the asterisk by tilting the board.
- √ Aside from holding your board horizontally and tilting it, try holding it vertically and rotating it in a circle. The asterisk should travel in a circular arc around the graph as you do so.

How BubbleGraph.bs2 Works

The first thing the Main Routine does is display the background character at the current cursor position. With a 50 ms pause, it completes the "off" portion of a blinking asterisk.

```
' Replace asterisk with background character.
index = (22 * y) + x + 1            ' Coordinates -> EEPROM address
READ index, char                    ' Get background character
DEBUG CRSRXY, x, y, char            ' Display background character
PAUSE 50                            ' Pause for blink effect
```


Next, the program acquires the x and y tilt.

```
' Measure tilt.
PULSIN 6, 1, x                      ' Get Ax and Ay
PULSIN 7, 1, y
```

The program needs to scale and offset the x and y-axis tilt measurements so that the asterisk is correctly placed in the Debug Terminal. The same scaling and offset introduced in Chapter 3, Activity #3 works for this task. For the x-axis, the accelerometer's pulse values of 1875 to 3125 have to be scaled to asterisk placements of 0 to 20. By subtracting 1875 from the accelerometer measurement before scaling, we have an input scale of 0 to 1250 (1251 elements) and an output scale of 0 to 20 (21 elements). The equation for calculating the ****** scale constant is:

ScaleConstant = Int[65536(output scale elements)/(input scale elements - 1)]

Substituting the number of elements in the input and output scales gives us a ****** scale constant of 1101.

ScaleConstant = Int[65536(21/(1251-1))]
ScaleConstant = Int[65536(21/1250)]
ScaleConstant = Int[1101.0048]
ScaleConstant = 1101

A similar process results in a ****** scale constant of 576 for the y-axis, and the resulting code for scale and offset for both axes is:

```
' Calculate cursor position.
x = (x MIN 1875 MAX 3125) - 1875 ** 1101   ' Calculate x position
y = (y MIN 1875 MAX 3125) - 1875 ** 576    ' Calculate y position
y = 10 - y
```

If the accelerometer measurements are slightly outside the 1875 to 3125 scale, it can cause strange display symptoms. The **MIN** and **MAX** operators prevent this problem. Also, 1875 is subtracted from each axis before it is scaled with the ****** operator. The result for the x-axis is 0 to 1250 is scaled to 0 to 20. For the y-axis, 0 to 1250 is scaled to 0 to 10.

As the y-axis tilt measurements decrease, the downward position of the cursor has to increase. So instead of fitting 1875 to 3125 into 0 to 10, the program has to fit it into 10 to 0 instead. The statement **y = 10 - y** solves this. If y is 0 after scaling, it becomes 10. Likewise if y is 10 after scaling it becomes 0. If it's 9 after scaling it becomes 1, if it's 8 after scaling, it becomes 2, and so on.

The last steps before repeating the loop in the Main Routine is to display the new asterisk at its new **x** and **y** coordinates, then pause for another 50 ms to complete the "on" portion of the blinking asterisk.

```
' Display asterisk at new cursor position.
DEBUG CRSRXY, x, y, "*"
PAUSE 50
```

Your Turn – A Larger Bubble

Displaying and erasing a group of asterisks like the one shown in Figure 5-13 can be done, but compared to a single character, it's a little tricky. The program has to ensure that none of the asterisks will be displayed outside the plot area. It also has to ensure that all of the asterisks will be overwritten with the correct characters from EEPROM. Here is one example of how to modify BubbleGraph.bs2 so that it displays a 5-asterisk coordinate indicator:

- √ Save BubbleGraph.bs2 as BubbleGraphYourTurn.bs2.
- √ Add this variable declaration to the program's Variables section:

```
temp    VAR     Byte
```

- √ Replace the "Replace asterisk with background character" routine with this:

```
' Replace asterisk with background character (modified).
FOR temp = (x MIN 1 – 1) TO (x MAX 19 + 1)
  index = (22 * y) + temp + 1
  READ index, char
  DEBUG CRSRXY, temp, y, char
NEXT
```

```
FOR temp = (y MIN 1 – 1) TO (y MAX 9 + 1)
  index = (22 * temp) + x + 1
  READ index, char
  DEBUG CRSRXY, x, temp, char
NEXT
PAUSE 50
```

√ Replace the " Display asterisk at new cursor position" routine with this:

```
' Display asterisk at new cursor position (modified).
DEBUG CRSRXY, x,           y,           "*",
      CRSRXY, x MAX 19 + 1, y,          "*",
      CRSRXY, x,           y MAX 9 + 1, "*",
      CRSRXY, x MIN 1 - 1,  y,          "*",
      CRSRXY, x,           y MIN 1 - 1, "*"
PAUSE 50
```

√ Run the program and try it. Test to make sure problems do not occur as one of the outermost asterisks is forced off the plot area.

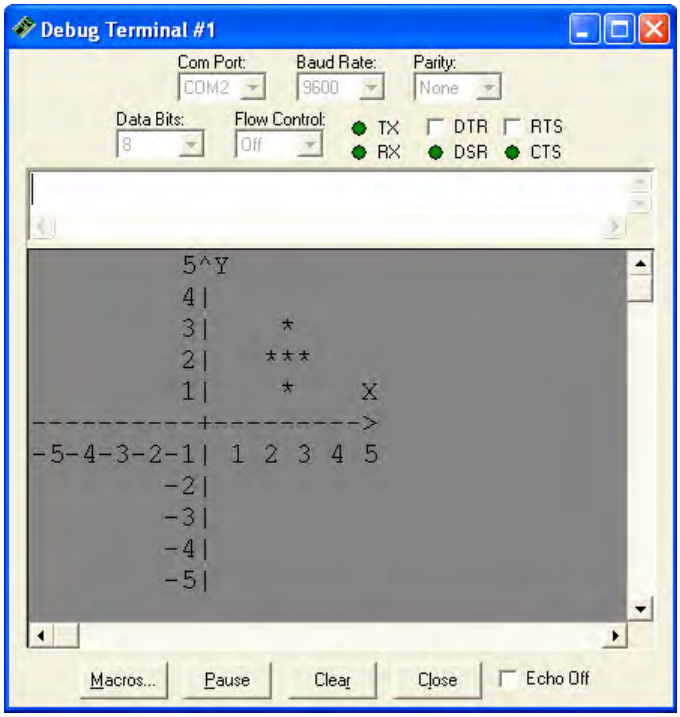


Figure 5-13
Group of Asterisks with Background Refresh



MIN and Negative Numbers

A two's complement "gotcha" to avoid is subtracting 1 from 0 and then setting the **MIN** value afterwards. Remember from Chapter 3 that the two's complement system stores the signed value -1 as 65535. That's why the **MIN** value was set to 1 before subtracting 1. The result is then a correct minimum of 0. The same technique was used for setting the **MAX** values even though there really isn't a problem with `y + 1 MAX 10`.

ACTIVITY #4: GAME CONTROL

Here are the rules of this Activity's tilt-controlled game example, shown in Figure 12. Tilt your board to control the asterisk. If you get through the maze and place the asterisk on any of the "WIN" characters, the "YOU WIN" screen will display. If you bump into any of the pound signs "#" before you get to the end of the maze, the "YOU LOSE" screen is displayed. As you navigate the maze, try to move your asterisk game character through the dollar signs "$" to get more points.

Converting BubbleGraph.bs2 into TiltObstacleGame.bs2

TiltObstacleGame.bs2 is inarguably a hopped-up version of BubbleGraph.bs2. Here is a list of the main changes and additions:

- Change the graph into a maze.
- Add two backgrounds for win and lose to the EEPROM data.
- Give each background a Symbol name.
- Write a game player code block that detects which background character the game character is in front of and uses that information to enforce the rules of the game.

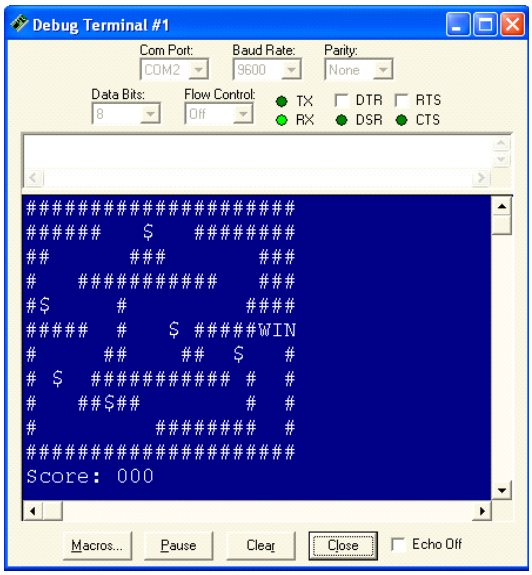
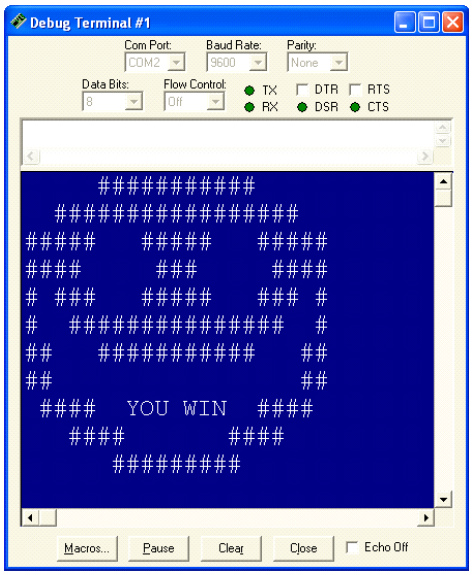
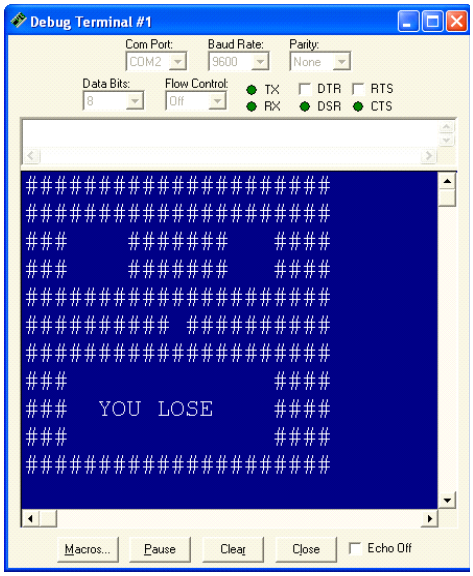


Figure 5-14: Obstacle Course Game

Game Maze background (left)
"You Win" Display (below left)
"You Lose Display (below right)





Try the game first, and then we'll take a closer look at how it works.

Example Program – TiltObstacleGame.bs2



Free Download! This program is available as a free .bs2 file download from the Smart Sensors and Applications Product Page at www.parallax.com.

- √ Open (or enter) and save TiltObstacleGame.bs2.
- √ Before you run the program, make sure your board is level. Also, make sure you are holding it the same way you did in Activity 3, with the breadboard closest to you, and the serial cable furthest away.
- √ If you want to refresh the "$" characters, click your BASIC Stamp Editor's Run button. If you want to just practice navigating and not worry about points, press and release the Reset button on your board.

```
' -----[ Title ]-----------------------------------------------------------------
' Smart Sensors and Applications - TiltObstacleGame.bs2
' Tilt accelerometer to guide cursor through maze, collect $ in Debug Terminal

'{$STAMP BS2}                                  ' Stamp/PBASIC directives
'{$PBASIC 2.5}

' -----[ EEPROM Data ]-----------------------------------------------------------
' Store background to EEPROM                   ' 3 backgrounds used in game

Maze DATA @0, HOME,                            ' Maze background
"######################", CR,
"######   $   ########", CR,
"##      ###       ###", CR,
"#   ############   ###", CR,
"#$     #         ####", CR,
"#####  #   $ #####WIN", CR,
"#     ##    ##  $   #", CR,
"# $  ########### #  #", CR,
"#   ##$##        #  #", CR,
"#         ########  #", CR,
"######################", CR

YouLose DATA @243, HOME,                       ' YouLose background
"######################", CR,
"######################", CR,
"###    #######   ####", CR,
"###    #######   ####", CR,
"######################", CR,
"########## ##########", CR,
```

```
"######################", CR,
"###                ####", CR,
"###  YOU LOSE      ####", CR,
"###                ####", CR,
"######################", CR

YouWin DATA @486, HOME,                         ' YouWin background
"     ############     ", CR,
"  ##################  ", CR,
"#####   #####   #####", CR,
"####     ###     ####", CR,
"# ###   #####   ### #", CR,
"#  ###############  #", CR,
"##   ############   ##", CR,
"##                  ##", CR,
" ####  YOU WIN  #### ", CR,
"   ####       ####   ", CR,
"      #########      ", CR

' -----[ Variables ]-------------------------------------------------------
x         VAR     Word                          ' x & y tilts & graph coordinates
y         VAR     Word

index     VAR     Word                          ' EEPROM address and character
char      VAR     Byte

symbol    VAR     Word                          ' Symbol address for EEPROM DATA
points    VAR     Byte                          ' Points during game

' -----[ Initialization ]---------------------------------------------------
x = 10                                          ' Start game character in middle
y = 5

DEBUG CLS                                       ' Clear screen

' Display maze.
symbol = Maze                                   ' Set Symbol to Maze EEPROM DATA

FOR index = 0 TO 242                            ' Display maze
  READ index + symbol, char
  DEBUG char
NEXT

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

  ' Display background at cursor position.
  index = (22 * y) + x + 1                      ' Coordinates -> EEPROM address
  READ index + symbol, char                     ' Get background character
  DEBUG CRSRXY, x, y, char                      ' Display background character
  PAUSE 50                                      ' Pause for blink effect
```

```
  ' Measure tilt and calculate cursor position.
  PULSIN 6, 1, x                            ' Get Ax and Ay
  PULSIN 7, 1, y

  x = (x MIN 1875 MAX 3125) - 1875 ** 1101   ' Calculate x position
  y = (y MIN 1875 MAX 3125) - 1875 ** 576    ' Calculate y position
  y = 10 - y

  ' Display asterisk at new position.
  DEBUG CRSRXY, x,  y, "*"                  ' Display asterisk
  PAUSE 50                                  ' Pause again for blink effect

  ' Display score
  DEBUG CRSRXY, 0, 11,                      ' Display points
        "Score: ", DEC3 points

  ' Did you move the asterisk over a $, W, I, N, or #?
  SELECT char                               ' Check background character
    CASE "$"                                ' If "$"
      points = points + 10                  ' Add points
      WRITE index, "%"                      ' Write "%" over "$"
    CASE "#"                                ' If "#", set Symbol to YouLose
      symbol = YouLose
    CASE "W", "I", "N"                      ' If W,I,orN, Symbol -> YouWin
      symbol = YouWin
  ENDSELECT

  ' This routine gets skipped while symbol is still = Maze.  If symbol
  ' was changed to YouWin or YouLose, display new background and end game.
  IF (symbol = YouWin) OR (symbol = YouLose) THEN
    FOR index = 0 TO 242                    ' 242 characters
      READ index + symbol, char             ' Get character
      DEBUG char                            ' Display character
    NEXT                                    ' Next iteration of loop
    END                                     ' End game
  ENDIF                                     ' End symbol-if code block

LOOP                                        ' Repeat main loop
```

How it Works – From BubbleGraph.bs2 to TiltObstacleGame.bs2

Two of the **DATA** directive's optional features were used here. Each of the three backgrounds was given a *Symbol* name, **Maze**, **YouWin**, and **YouLose**. These *Symbol* names make it easy for the program to select which background to display. The optional **@Address** operator was also used to set each directive's beginning EEPROM address. In BubbleGraph.bs2's background, the first character is **CLS** to clear the screen. The problem with **CLS** in these **DATA** directives is that it erases the entire Debug Terminal,

including the score, which is displayed below the background. By substituting **HOME** for **CLS**, the entire backgrounds can be drawn and redrawn without erasing the score.

```
Maze DATA @0, HOME,
"####################", CR,
"######   $   ########", CR,
   .
   .
   .
YouLose DATA @243, HOME,
"####################", CR,
"####################", CR,
   .
   .
   .
YouWin DATA @486, HOME,
"     ###########     ", CR,
"  #################  ", CR,
   .
   .
```



Verifying Symbol Values

You can also try commands like **DEBUG DEC YouWin** to verify **YouWin** stores the value 486.

Two variables are added, **symbol** to keep track of which background to retrieve characters from, and **points** to keep track of the player's score.

```
symbol    VAR     Word
points    VAR     Byte
```

The initial values of **x** and **y** have to start in the middle of the obstacle course. Since all variables initialize to zero in PBASIC, failure to initialize them would cause the game character to start in the top-left corner, instead of in the middle.

```
x = 10
y = 5
```

The symbol variable is set to **Maze** before executing the **FOR…NEXT** loop that displays the background. Since all variables are initialized to zero in PBASIC, this happens anyhow. However, if you were to insert a **DATA** directive before the **Maze** background, it would be crucial to have this statement.

```
' Display maze.
symbol = Maze
```

The code block that follows the variable initialization is the background display. Look carefully at the **READ** command. It has been changed from **READ index, char** to **READ index + symbol, char.** Since the **symbol** variable was set to store **Maze**, all the characters in the first background will be displayed. If **symbol** stored **YouLose**, all the characters in the second background would be displayed. If it stored **YouWin**, all the characters in the third background would be displayed. Since either "**You Lose**" or "**You Win**" will have to be displayed, this routine will be used again later in the program.

```
FOR index = 0 TO 242
  READ index + symbol, char
  DEBUG char
NEXT
```

Three routines have to be added to the **DO...LOOP** in the Main Routine. The first simply displays the player's score:

```
' Display score
DEBUG CRSRXY, 0, 11,                    ' Display points
"Score: ", DEC3 points
```

The second routine is crucial; it's a **SELECT…CASE** statement that enforces the rules of the game. The **SELECT…CASE** statement looks at the background character at the asterisk's current location. If the asterisk is over a space **" "**, the **SELECT…CASE** statement doesn't need to change anything, so the Main Routine's **DO…LOOP** just keeps on repeating itself, checking the accelerometer measurements and updating the asterisk's location. If the asterisk is moved over a **"$"**, the program has to add 10 to the **points** variable, and write a **"%"** character over the **"$"** in EEPROM. This prevents the program from adding 10 points several times per second while the asterisk is held over the **"$"**. If the asterisk is moved over a **"#"**, the **YouLose** symbol is stored in the **symbol** variable. If the asterisk moves over any one of the **"W" "I"** or **"N"** letters, **YouWin** is stored in **symbol**.

```
' Did you move the asterisk over a $, W, I, N, or #?
SELECT char                             ' Check background character
  CASE "$"                              ' If "$"
    points = points + 10                ' Add points
    WRITE index, "%"                    ' Write "%" over "$"
  CASE "#"                              ' If "#", set Symbol to YouLose
    symbol = YouLose
  CASE "W", "I", "N"                    ' If W,I,orN, Symbol -> YouWin
    symbol = YouWin
  ENDSELECT
```

As you're navigating your asterisk over **" "**, **"$"**, or **"%"**, this next routine gets skipped because **symbol** still stores **Maze**. The **SELECT…CASE** statement only changes that when the asterisk was moved over **"#"**, **"W"**, **"I"**, or **"N"**. Whenever the **SELECT…CASE** statement changes **symbol** to either **YouWin** or **YouLose**, this routine displays the corresponding background, then ends the game.

```
' This routine gets skipped while symbol is still = Maze.  If symbol
' was changed to YouWin or YouLose, display new background & end game.
  IF (symbol = YouWin) OR (symbol = YouLose) THEN
    FOR index = 0 TO 242                ' 242 characters
      READ index + symbol, char         ' Get character
      DEBUG char                        ' Display character
    NEXT                                ' Next iteration of loop
    END                                 ' End game
  ENDIF                                 ' End symbol-if code block
```


Your Turn – Modifications and Bug Fixes

The game doesn't refresh the "**$**" symbols when you re-run it with the Board of Education's Reset button. It only works when you click the Run button on the BASIC Stamp Editor. That's because the **DATA** directive only writes to the EEPROM when the program is downloaded. If the program is restarted with the Reset button, the BASIC Stamp Editor doesn't get the chance to store the spaces, dollar signs, etc, so the percent signs that were written to EEPROM are still there. To fix the problem, all you have to do is check each character that gets read from EEPROM during the initialization. If that character turns out to be a "**%**", use the **WRITE** command to change it back to a "**$**".

- √ Save TiltObstacleGame.bs2 as TiltObstacleGameYourTurn.bs2
- √ Modify the **FOR...NEXT** loop in the initialization that displays the maze like this:

```
FOR index = 0 TO 242                ' Display maze
  READ index + symbol, char
  IF(char = "%") THEN               ' <--- Add
    char = "$"                      ' <--- Add
    WRITE index + symbol, char      ' <--- Add
  ENDIF                             ' <--- Add
  DEBUG char
NEXT
```

- √ Verify that both the BASIC Stamp Editor's Run button and the Board of Education's Reset button behave the same after this modification.

If the player rapidly changes the board's tilt, it is possible to jump over the "**#**" walls. There are two ways to fix this. One would be to add jumping animation and call it a "feature". Another way to fix it would be to only allow the asterisk to move by 1 character in either the X or Y directions. To fix this, the program will need to keep track of the previous position. This is a job for the **xOld** and **yOld** variables introduced in Activity #2.

√ Add these variable declarations to the Variables section in TiltObstacleGameYourTurn.bs2:

```
x         VAR     Word                  ' x & y tilts & coordinates
y         VAR     Word

xOld      VAR     Word                  ' <--- Add
yOld      VAR     Word                  ' <--- Add
```

√ Add initialization statements for **xOld** and **yOld**.

```
x    = 10                               ' Start game char in middle
xOld = 10                               ' <--- Add
y    = 5
yOld = 5                                ' <--- Add
```

√ Modify the Main Routine so that **x** can only be greater than or less than **xOld** by an increment or decrement of 1. Repeat for **y** and **yOld**.

```
y = 10 - y                              ' Offset Cartesian -> Debug

IF (x > xOld) THEN x = xOld MAX 19 + 1  ' <--- Add
IF (x < xOld) THEN x = xOld MIN 1 - 1   ' <--- Add

IF (y > yOld) THEN y = yOld MAX 9 + 1   ' <--- Add
IF (y < yOld) THEN y = yOld MIN 1 - 1   ' <--- Add

' Display asterisk at new position.
DEBUG CRSRXY, x,  y, "*"                ' Display asterisk
PAUSE 50                                ' Pause again for blink effect

xOld = x                                ' <--- Add
yOld = y                                ' <--- Add

' Display score
```

√ Run and test your modified program and verify that the asterisk can no longer skip "**#**" walls.

SUMMARY

Activity #1 introduced control characters, techniques for keeping characters inside a display's boundaries, and algebra for mapping coordinates to a display. Control character examples included **CRSRXY** and **CLRDN**. Display boundary examples included the **MIN** and **MAX** operators and an **IF…THEN** technique. Mapping techniques included simple PBASIC equations to change the values of X and Y coordinates from Cartesian to their Debug Terminal equivalents.

Activity #2 introduced a means of storing, displaying and refreshing a background character display image from EEPROM. This is a useful ingredient for many product displays, and will also come in handy for tilt display and games. An entire display background can be printed with a **FOR…NEXT** loop. A **READ** command in the loop depends on the **FOR…NEXT** loop's **index** variable to address the next character in the sequence. After the **READ** command loads the next character in the variable, the **DEBUG** command can be used to send the character to the Debug Terminal. For erasing the tracks left by a character moving over the background, the character's previous position can be stored in one or more variables. The previous position information is then used along with the **READ** command to look up the character that should replace the moving character after it has moved to its next position.

Activity #3 demonstrated how the accelerometer measurements from Chapter 3 can be combined with cursor positioning and character recall techniques from Activity #2 to create a tilt-controlled display. Simple **PULSIN** measurements were used to measure the accelerometer's x and y-axis tilt. The tilt values were then scaled and offset using techniques introduced in Chapter 3, Activity #3. The modified **x** and **y** values dictated cursor placement for printing the asterisk in the Debug Terminal. The asterisk's position relative to the Cartesian plane displayed in the background represented the center of the hot air pocket inside the MX2125's chamber. As the asterisk moved, the background of its previous location was redrawn using techniques introduced in Activity #2.

Activity #4 introduced tilt-mode game control. The rules of simple games can be implemented with **SELECT...CASE** statements that use the character in the background at the location of the game character to decide what action to take next. Multiple backgrounds can be incorporated into a game by making use of the **DATA** directive's optional **@Address** operator and *Symbol* name. Since the *Symbol* name is actually the EEPROM address at the beginning of a given **DATA** directive, your program can access

elements in different backgrounds by adding the value of *Symbol* to the **READ** command's ***Address*** argument.

Questions

1. What does HID stand for?
2. What control character causes all the lines below the cursor's current location to be erased?
3. What command and formatter can you use to store a single digit that you type into the Debug Terminal's Transmit windowpane in the **X** variable?
4. Are there other coding techniques you can use with other operators to prevent the value a variable stores from exceeding a maximum or minimum value?
5. What are the refresh rates of common CRT computer monitors?
6. What kind of routine do you need to display all the background characters stored in a **DATA** directive?
7. Why are word variables necessary for storing signed values in PBASIC?
8. When you tilt the accelerometer to the left, which way does the asterisk bubble travel?
9. If the coordinates of the asterisk started at (-5, 0), and ended at (5, 0), what do you think happened to the accelerometer?
10. Which axis was the fulcrum if the accelerometer started at (2, 2) and ended at (-2, 2)?
11. If the accelerometer's readings travel from (0, 5) to (0, -5), then back again repeatedly, what motion sequence is likely?
12. What's the value of **YouWin**?
13. What command can you use to check the value of a **DATA** directive's ***Symbol*** name?
14. If you change the **Maze DATA** directive's **@Address** operator from 0 to 10, what will you have to do to the other **DATA** directives in the program?
15. In TiltObstacleGame.bs2, what kind of code block enforces the rules of the game?
16. What command changes the "**%**" values back to "**$**" values in EEPROM?

Exercises

1. Write a **DEBUG** command that places the cursor five spaces over, seven spaces down, and then prints the message "* this is the coordinate (5, 7) in the Debug Terminal".

2. Write a **DEBUG** command that displays a Cartesian coordinate system from -2 to 2 on the X and Y axes.
3. Calculate the scale and offset you will need to display ordered pairs entered into the Debug Terminal's Transmit windowpane on a Cartesian coordinate system that goes from -5 to 5 on both the X and Y axes.
4. Write a routine that draws a rectangle with asterisks. This routine should be 15 asterisks wide and 5 asterisks high.
5. If your background is 5 characters wide by 3 characters high, predict the minimum size variable you can use to set the address for your **READ** command and explain your choice. Will you have any room for additional characters such as **CLS**?

Projects

1. Modify CrsrXYPlot.bs2 so that it redraws the background before it plots the asterisk.
2. Modify PlotXYGraph.bs2 so that it displays the coordinates of the most recently placed asterisk to the right of the plot area.

Solutions

Q1. Human interface device.
Q2. **CLRDN**.
Q3. **DEBUGIN DEC1 x**.
Q4. Yes, you can also use **IF…THEN** statements to check whether values are out of bounds.
Q5. 60, 70, and 72 Hz.
Q6. You can use a **FOR…NEXT** loop, with statements to retrieve and display each character stored. Word variables store both the value and the sign.
Q7. Word variables store both the value and the sign.
Q8. The bubble moves to the right.
Q9. It went from a tilted right position to tilted left position.
Q10. The Y axis, or the long axis of the Board of Education.
Q11. Titling the board up and down repeatedly.
Q12. EEPROM address 486.
Q13. You can use the **DEBUG** command as you would to display the value of any other numeric quantity, use the **DEC** modifier.

```
DEBUG DEC symbol
READ index + symbol, char
```

Q14. You would need to add 10 to each symbol value, so your program would become:

```
YouLose DATA @253…
YouWin  DATA @496 …
```

Q15. A **SELECT…CASE** statement.
Q16. The **WRITE** command.
E1. Example solution:

```
DEBUG CRSRXY, 5, 7,
       "* this is the coordinate (5,7) in the Debug Terminal"
```

E2. Example solution:

```
DEBUG CLS,
"   2|    ", CR,
"   1|    ", CR,
"----+----", CR,
"-2-1| 1 2", CR,
"  -2|    ", CR, CR
```

E3. X axis scale is 2 and offset is 10; Y axis, scale is -1 and offset is 5.
E4. Example solution:

```
x         VAR     Byte
y         VAR     Byte
```

```
FOR x = 1 TO 15
  FOR y = 1 TO 5
    DEBUG CRSRXY, x, y, "*"
  NEXT
NEXT
```

E5. The number of characters to store equals 5 x 3 = 15. The minimum variable size to use would be a Nib (4 bits), with which the addresses could range from 0 to 15. There would be room for only one (1) additional character.

P1. The key to solving this problem is to move the graph inside the **DO…LOOP**, and change **CLS** to **HOME**. There will be other ways to solve the problem as well. Example solution:

```
DO
  DEBUG HOME,
  "0123456789X", CR,
  "1          ", CR,
  "2          ", CR,
  "3          ", CR,
  "4          ", CR,
  "5          ", CR,
  "Y          ", CR, CR
  DEBUG "Type X coordinate: "
  DEBUGIN DEC1 x
  DEBUG CR, "Type Y coordinate: "
  DEBUGIN DEC1 y
  DEBUG CRSRXY, x, y, "*"
  DEBUG CRSRXY, 0, 10, "Press any key..."
  DEBUGIN temp
  DEBUG CRSRXY, 0, 8, CLRDN
LOOP
```

P2. Modify PlotXYGraph.bs2 so that it displays the coordinates of the most recently placed asterisk to the right of the plot area. To properly display the negative coordinates, use the **SDEC** modifier.

```
DO
  DEBUG "Type X coordinate: "
  DEBUGIN SDEC1 x
  DEBUG CR, "Type Y coordinate: "
  DEBUGIN SDEC1 y
  DEBUG CRSRXY, 15, 3, "(X,Y) = (",
        SDEC x, ",", SDEC y, ")", CLREOL
  x = (x * 2) + 6
  y = 3 - y
  DEBUG CRSRXY, x, y, "*"
  DEBUG CRSRXY, 0, 10, "Press any Key..."
  DEBUGIN temp
  DEBUG CRSRXY, 0, 8, CLRDN
LOOP
```

Chapter 6: More Accelerometer Projects

There are three types of projects in this chapter. The first type is a direct application of hardware and programs that were used in earlier chapters. The second type requires datalogging of the acceleration measurements, and so several activities are devoted to a datalogging program. The third type requires datalogging to figure out what kind of measurements the accelerometer will report. Then, based on the datalogging results, you will have enough information to write a program to make the device work reliably.

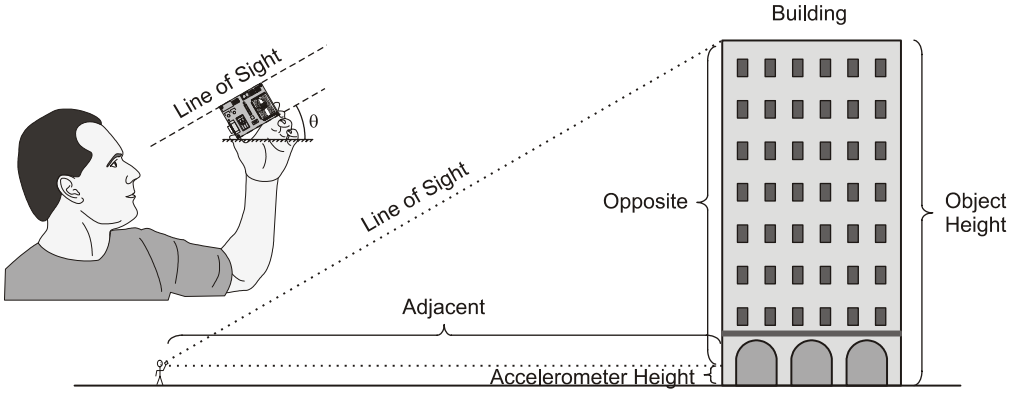
ACTIVITY #1: MEASURE HEIGHTS OF BUILDINGS, TREES, ETC.

Climbing to the top of an object to measure its height is not always convenient, practical, or even safe. This activity introduces a novel way to use some of the accelerometer measurements developed in Chapter 3 to make height measurements from a safe vantage point on the ground.

Sighting the Top and Determining Height

Figure 6-1 shows a scheme for measuring the height of an object, using the accelerometer and LCD display as an angle-sight. First, sight the top of the object with the edge of your board, and record the measured angle. Then, measure the distance between the spot you took your measurement and the object, which is the adjacent side shown in Figure 6-1. The adjacent distance, the angle θ, and the height of the accelerometer above the ground are the three key pieces of information you need to calculate the object height.

Figure 6-1: Determining Height with Line-of-Sight



Parts Required

Use the parts list and circuit from Chapter 3, Activity #2 on page 71 for your angle-sight.

Example Program

Use the example program VertWheelRotation.bs2 from page 95 (Chapter 3, Activity #5) along with the LCD display modifications in the Your Turn - LCD Display section on page 96.

Procedure

- √ Use your board to angle-sight the top of the object, and record the angle.
- √ Measure the distance between the sight point and the object (adjacent side in Figure 6-1).
- √ Measure the height at which the accelerometer was held.
- √ Use the calculations introduced next to determine the object's height.

Calculations

We know from earlier chapters θ is equal to the opposite side divided by the adjacent side of a right triangle. Multiplying both sides by the adjacent distance results in an expression for solving the opposite height. It's the adjacent distance multiplied by the tangent of the angle.

$$\tan\theta = \frac{\text{opposite}}{\text{adjacent}}$$

$$\text{opposite} = \text{adjacent}\ \tan\theta$$

After determining the opposite height (shown in Figure 6-1), all you have to do is add to that the height at which you held the accelerometer when you took the measurement.

$$\text{object height} = \text{opposite} + \text{accelerometer height}$$

$$\text{object height} = \text{adjacent}\ \tan\theta + \text{accelerometer height}$$

Example

Let's say that the adjacent distance to an object is 10 m, and at that distance the accelerometer was held 1.5 m from the ground to get the line of sight of the top of an object. The angle reported by the accelerometer unit was 61°. From this, we can estimate the height of the object to be 19.54 m, as shown below.

$$\text{Adjacent distance} = 10\,\text{m}$$
$$\text{Accelerometer measured } \theta = 61^\circ$$
$$\text{Accelerometer height} = 1.5\ \text{m}$$

$$\begin{aligned}\text{opposite} &= \text{adjacent} \tan\theta \\ &= 10\,\text{m} \tan\theta \\ &= (10\,\text{m})(1.804) \\ &= 18.04\end{aligned}$$

$$\begin{aligned}\text{object height} &= \text{opposite} + \text{accelerometer height} \\ &= 18.04\,\text{m} + 1.5\,\text{m} \\ &= 19.54\,\text{m}\end{aligned}$$

ACTIVITY #2: RECORD AND PLAYBACK

With accelerometer projects, it will often be necessary to record and play back lots of accelerometer measurements. In some cases, recording the value is the desired function, like datalogging how a car handles a turn. In other cases, such as detecting the human walking motion, it will be necessary to understand what the measurements are like before a program can be written that tracks steps. In either case, recording and playing back acceleration measurements is a necessary ingredient. This activity introduces a program with subroutines that demonstrate how to record, play back, and erase values stored in the unused portion of the BASIC Stamp EEPROM program memory.

EEPROM Storage with DATA, WRITE and READ

While not required for recording and playing back measurements, **`DATA`** directives can be used to set aside chunks of unused program memory. The **`DATA`** directive's optional

Symbol name is especially useful for recordkeeping. The **Records DATA** directive does not actually store any values in EEPROM addresses 0 to 9. It just reserves these bytes for your PBASIC code, and gives a name to the address of the first byte: **Records**. The **RecordsEnd DATA** directive reserves a single byte at EEPROM address 10.

```
Records       DATA    (10)
RecordsEnd    DATA
```

The ***Symbol*** names (**Records** and **RecordsEnd**) become constants that store the starting address of the EEPROM **DATA** directives they precede. Table 6-1 shows how it works for the two **DATA** directives. Since **Records** is the first **DATA** directive, it sets aside the first ten bytes (addresses 0 to 9). Since address 0 is the beginning address, **Records** becomes a constant for the value 0 in the program. Likewise, since the **RecordsEnd DATA** directive sets aside a byte at address 10, **RecordsEnd** becomes the constant value 10 in the program.

Table 6-1: DATA directives and EEPROM Addresses

Byte Contents	00	00	00	00	00	00	00	00	00	00	00	00
Addresses	0	1	2	3	4	5	6	7	8	9	10	11
↥ **Records = 0**											**RecordsEnd = 10** ↥	



The EEPROM bytes don't necessarily contain zero. With the command **Records DATA (10)**, whatever values are already there will not be changed. If you want to initialize the EEPROM values to zero, use **Records DATA 0 (10)**. This will store 0 in EEPROM addresses 0 to 9. The BASIC Stamp Editor only does this when it downloads the program. If you press and release your board's Reset button or disconnect and reconnect power, no values are written to those EEPROM addresses. This is a handy feature, as you will see in the next activity.

The **Clear_Data** subroutine in the next example program has a **FOR**...**NEXT** loop that repeats from **Records** to **RecordsEnd** (0 to 10). Each time through the loop, the **eeIndex** variable increases by 1, so **WRITE eeIndex, 100** stores 100 in each of the EEPROM bytes, from address 0 to address 10.

```
Clear_Data:
  FOR eeIndex = Records TO RecordsEnd
    WRITE eeIndex, 100
  NEXT
  DEBUG CR, "Records cleared."
  PAUSE 1000
  RETURN
```

The **Record_Data** subroutine in the next example program collects the values that you enter into the Debug Terminal's Transmit windowpane. In the next activity, this subroutine will be modified to store accelerometer readings instead. The **FOR**...**NEXT** loop again starts at **Records** and repeats until **eeIndex** exceeds **RecordsEnd**. Each time through the loop, the **value** variable receives a signed decimal number from the Debug Terminal's Transmit windowpane and stores it in the EEPROM address selected by **eeIndex** with **WRITE eeIndex, value**.


```
Record_Data:
  DEBUG CR, "Enter values from -100 to 100", CR
  FOR eeIndex = Records TO RecordsEnd
    DEBUG "Record ", DEC eeIndex, " >"
    DEBUGIN SDEC value
    value = value + 100
    WRITE eeIndex, value
  NEXT
  DEBUG CR, "End of records.",
        CR, "Press Enter for menu..."
  DEBUGIN char
  RETURN
```



Saving space with `value = value + 100`

Before each **value** variables' content is copied to EEPROM, 100 is added to it. So instead of a value between −100 and 100, a value between 0 and 200 is stored in the EEPROM. This is because each EEPROM memory cell can store a byte-sized value between 0 and 255. Storing negative numbers would require word-sized storage.

Word-size values can also be stored with **DATA** directives if you place the **Word** modifier before the ***DataItem***. For example **WRITE eeIndex, Word value**. Keep in mind that this command uses two EEPROM bytes to store the word-size value, so **eeIndex** will have to be incremented by 2 before the next value is written. By adding 100, we've saved one byte-sized cell per write.

To retrieve and display the values that were stored, the **Display_Data** subroutine has a **FOR**...**NEXT** loop with **READ eeIndex, value**. Since 100 was added to each value before

it was stored with the **WRITE** command, 100 is subtracted from the **value** variable after the **READ** command to bring **value** back into the −100 to 100 scale.

```
Display_Data:
  DEBUG CR, "Index  Record",
        CR, "-----  ------",
        CR
  FOR eeIndex = Records TO RecordsEnd
    READ eeIndex, value
    value = value - 100
    DEBUG DEC eeIndex, CRSRX, 7, SDEC value, CR
  NEXT
  DEBUG CR, "Press Enter for menu..."
  DEBUGIN char
  RETURN
```

Example Program: EepromDataStorage.bs2

This example program displays a three-choice menu in the Debug Terminal's Receive windowpane shown in Figure 6-2. By typing C into the Debug Terminal's Transmit windowpane, the values in the EEPROM set aside for storage are cleared. If R is typed, the program records values you enter into the Receive windowpane in EEPROM. If D is typed, the values that were stored in EEPROM are displayed.

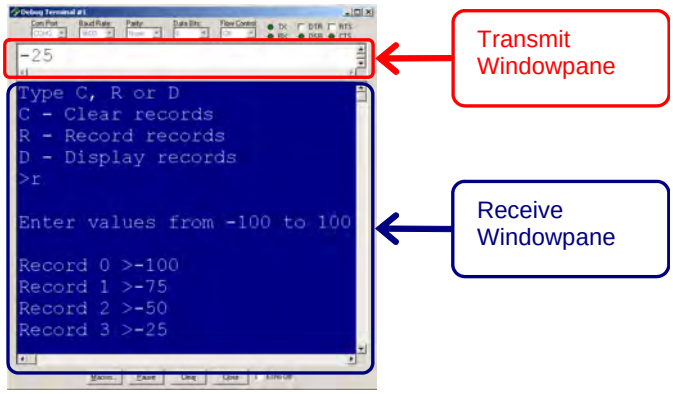


Figure 6-2
Entering Values for EepromDataStorage.bs2

√ Enter, save, and run EepromDataStorage.bs2.
√ Click the Debug Terminal's Transmit windowpane.
√ Type R, and then enter eleven values between −100 and 100. Press the Enter key when prompted after the eleventh value to get back to the menu.

- √ Type D and verify that the values you entered are correctly displayed. Press the enter key to return to the menu.
- √ Type C to clear the memory.
- √ Type D to verify that the memory values have been cleared (set to zero).

```
' -----[ Title ]-------------------------------------------------------------
' Smart Sensors and Applications - EepromDataStorage.bs2
' Demonstrates storing, retrieving and erasing values in EEPROM memory.

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

' -----[ DATA Directives ]-----------------------------------------------------

Records        DATA    (10)
RecordsEnd     DATA

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

char           VAR     Byte
eeIndex        VAR     Word
value          VAR     Word

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

DO

  DEBUG CLS,
        "Type C, R or D", CR,
        "C - Clear records", CR,
        "R - Record records", CR,
        "D - Display records", CR,
        ">"

  DEBUGIN char
  DEBUG CR

  SELECT char
    CASE "C", "c"
      GOSUB Clear_Data
    CASE "R", "r"
      GOSUB Record_Data
    CASE "D", "d"
      GOSUB Display_Data
    CASE ELSE
      DEBUG CR, "Not a valid entry.",
            CR, "Try again."
      PAUSE 1500
  ENDSELECT
```

```
LOOP

' -----[ Subroutine - Clear_Data ]-----------------------------------------

Clear_Data:
  FOR eeIndex = Records TO RecordsEnd
    WRITE eeIndex, 100
  NEXT
  DEBUG CR, "Records cleared."
  PAUSE 1000
  RETURN

' -----[ Subroutine - Record_Data ]-----------------------------------------

Record_Data:
  DEBUG CR, "Enter values from -100 to 100", CR
  FOR eeIndex = Records TO RecordsEnd
    DEBUG "Record ", DEC eeIndex, " >"
    DEBUGIN SDEC value
    value = value + 100
    WRITE eeIndex, value
  NEXT
  DEBUG CR, "End of records.",
        CR, "Press Enter for menu..."
  DEBUGIN char
  RETURN

' -----[ Subroutine - Display_Data ]-----------------------------------------

Display_Data:
  DEBUG CR, "Index  Record",
        CR, "-----  ------",
        CR
  FOR eeIndex = Records TO RecordsEnd
    READ eeIndex, value
    value = value - 100
    DEBUG DEC eeIndex, CRSRX, 7, SDEC value, CR
  NEXT
  DEBUG CR, "Press Enter for menu..."
  DEBUGIN char
  RETURN
```

Your Turn - How Many Bytes do You Want to Store?

EepromDataStorage.bs2 uses the **`Records`** and **`RecordsEnd`** for all loops that perform **`READ`** and **`WRITE`** operations. Because of this, you can change the number of values the

program stores by simply changing the number of elements in the **Records DATA** directive.

- √ Try changing the number of elements the program stores from 11 to 7. All you have to do is **change Records DATA (10)** to **Records DATA (6)**.
- √ Test and verify that it works.

In Activity #4, we'll use this feature to change the number of records the program stores to 1000 with **Records DATA (1000)**. 11

ACTIVITY #3: USE EEPROM TO TOGGLE MODES

This activity introduces an EEPROM trick you can use to turn the Board of Education's Reset button into a switch for selecting different program modes.

Code that Makes the Reset Button a Mode Selector

If you set aside one byte of EEPROM, it can give you the ability to select among as many as 256 different program modes. In the next example program, we'll just use two modes: a menu mode, and a mode that jumps to datalogging after a slight delay. Here is a **DATA** directive that names an EEPROM byte **Reset**, and initializes the value stored by this byte to zero.

```
Reset          DATA    0
```

The simplest form of the initialization is an on/off switch configuration. This is where the value from the **Reset** EEPROM byte is read, 1 is added to it, and then the modified value is written to the **Reset** byte. The modified value is also examined to see if it is odd or even with **IF value // 2 = 0 THEN**... I

```
READ Reset, value
value = value + 1
WRITE Reset, value
IF value // 2 = 0 THEN END
```

In this example, if that condition is true, the program ends right there. The next time you press and release your board's Reset button, **value** will be odd, the condition will be false, and the code block will not halt the program before it has reached the Main Routine. If the Reset button is pressed and released yet again, the code block will halt the

program again. The time after that, it does not halt the program, and so on. So the program utilizes your Board of Education's Reset button as an on/off toggle button.

Below is an example that uses the code block in a different way. Instead of halting or allowing the program to continue, the **IF**...**THEN** code block is skipped the first time the program is run, then executed the second time the program is run (after pressing and releasing the Reset button). It is then skipped the next time and executed again the time after that. The net effect is that the program either counts down and jumps straight to the **Record_Data** subroutine, or moves on to the main menu in the program, depending on whether your board's Reset button has been pressed/released an odd or even number of times.

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

READ Reset, value
value = value + 1
WRITE Reset, value

IF value // 2 = 0 THEN

  FOR char = 15 TO 0
    DEBUG CLS, "Datalogging starts", CR,
               "in ", DEC2 char, " seconds",
               CR, CR,
               "Press/release Reset", CR,
               "for menu..."
    FREQOUT 4, 50, 3750
    PAUSE 950
  NEXT

  GOTO Record_Data

ENDIF
```

Example Program: EepromDataStorageWithReset.bs2

This program demonstrates how to use an address in EEPROM to control the way the program behaves, depending on whether the program has been run or re-run an odd or even number of times. The number of times the program has been run will be controlled by the Reset button after download. If the Reset button has been pressed/released an even number of times, the program starts with the menu from the previous activity. If it has been pressed/released an odd number of times, it performs a countdown, and then calls the **Record_Data** subroutine.

- √ Enter and run EepromDataStorageWithReset.bs2.
- √ Verify that you can toggle the program's start mode by pressing and releasing the Reset button.
- √ Test the program's features, and make sure they all work.

```
' -----[ Title ]-------------------------------------------------------------
' Smart Sensors and Applications - EepromDataStorageWithReset.bs2
' Demonstrates storing, retrieving and erasing values in EEPROM memory.

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

' -----[ DATA Directives ]-----------------------------------------------

Reset           DATA    0
Records         DATA    (10)
RecordsEnd      DATA

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

char            VAR     Byte
eeIndex         VAR     Word
value           VAR     Word

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

READ Reset, value
value = value + 1
WRITE Reset, value

IF value // 2 = 0 THEN

  FOR char = 15 TO 0
    DEBUG CLS, "Datalogging starts", CR,
               "in ", DEC2 char, " seconds",
               CR, CR,
               "Press/release Reset", CR,
               "for menu..."
    FREQOUT 4, 50, 3750
    PAUSE 950
  NEXT

  GOTO Record_Data

ENDIF

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

DO
```

```
  DEBUG CLS,
        "Press/Release Reset", CR,
        "to arm datalogger ", CR, CR,
        " - or - ", CR, CR,
        "Type C, R or D", CR,
        "C - Clear records", CR,
        "R - Record records", CR,
        "D - Display records", CR,
        ">"

  DEBUGIN char
  DEBUG CR

  SELECT char
    CASE "C", "c"
      GOSUB Clear_Data
    CASE "R", "r"
      GOSUB Record_Data
    CASE "D", "d"
      GOSUB Display_Data
    CASE ELSE
      DEBUG CR, "Not a valid entry.",
            CR, "Try again."
      PAUSE 1500
  ENDSELECT

LOOP

' -----[ Subroutine - Clear_Data ]-------------------------------------------

Clear_Data:
  FOR eeIndex = Records TO RecordsEnd
    WRITE eeIndex, 100
  NEXT
  DEBUG CR, "Records cleared."
  PAUSE 1000
  RETURN

' -----[ Subroutine - Record_Data ]-------------------------------------------

Record_Data:
  DEBUG CR, "Enter values from -100 to 100", CR
  FOR eeIndex = Records TO RecordsEnd
    DEBUG "Record ", DEC eeIndex, " >"
    DEBUGIN SDEC value
    value = value + 100
    WRITE eeIndex, value
  NEXT
  DEBUG CR, "End of records.",
        CR, "Press Enter for menu..."
```

```
   DEBUGIN char
  RETURN

' -----[ Subroutine - Display_Data ]-----------------------------------------

Display_Data:
  DEBUG CR, "Index  Record",
        CR, "-----  ------",
        CR
  FOR eeIndex = Records TO RecordsEnd
    READ eeIndex, value
    value = value - 100
    DEBUG DEC eeIndex, CRSRX, 7, SDEC value, CR
  NEXT
  DEBUG CR, "Press Enter for menu..."
  DEBUGIN char
  RETURN
```


Your Turn - The DATA Directive's Automatic EEPROM Addressing

Did you notice that the record numbers changed in this program? Instead of 0 to 10, they were 1 to 11. Try moving the **`Reset DATA`** directive after the other two. Then, run the modified program and examine the result. Make tables similar to Table 6-1 that illustrate the values stored by **`Reset`**, **`Records`**, and **`RecordsEnd`**. Make the first table to illustrate the original program, and the second one to illustrate the modified program in which you changed the order of the **`DATA`** directives.

ACTIVITY #4: REMOTE DATALOGGING OF ACCELERATION

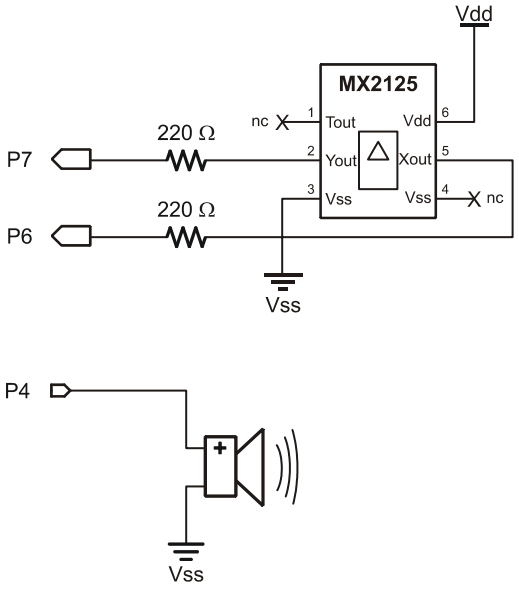
In this activity, you will add a piezospeaker to the existing accelerometer circuit. Then, you will modify the program so that it provides you with a remote datalogging tool that's easy to operate. The piezospeaker will be handy for indicating countdown, start, and stop. The accelerometer circuit will the same one used in Chapter #3, and the piezospeaker will be added below it on the breadboard.

Parts Required

(1) Memsic 2125 Accelerometer
(2) 220 Ω resistors
(1) piezospeaker
(4) jumper wires

Circuit

√ Build the accelerometer and piezospeaker circuits shown in Figure 6-3.



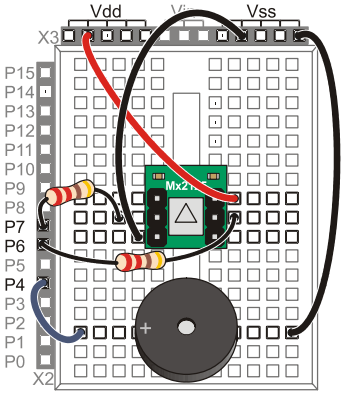


Figure 6-3
Accelerometer and Piezospeaker
Schematic (left) and
Wiring Diagram (below)

Program Modifications

The next example program, DatalogAcceleration.bs2, is an expansion of EepromDataStorageWithReset.bs2. It has been modified so that you can start, stop and restart datalogging with your board's Reset button. You can disconnect the board from your computer to perform the datalogging, and reconnect it to display the measurements in the Debug Terminal. This is a crucial feature for taking field measurements and then displaying them later.

DatalogAcceleration.bs2 has a modified Initialization section that makes the piezospeaker beep every second for ten seconds before it starts datalogging.

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

Init:
  .
  .
```

```
.
FOR char = 10 TO 0
.
.
.
  FREQOUT 4, 50, 3750
  PAUSE 950
NEXT
.
.
.
```


DatalogAcceleration.bs2 also has a modified **Record_Data** subroutine that gets the **x** and **y** values from the accelerometer, scales them to (-100 to 100), and writes both of them to EEPROM. The **FOR**...**NEXT** loop increments in steps of 2 with the **STEP 2** argument since each time through the loop, the routine saves two bytes. The **Display_Data** subroutine has similar modifications so that it displays both the **x** and **y** values in a table.

```
Record_Data:

  FREQOUT 4, 75, 4000
  PAUSE 200
  FREQOUT 4, 75, 4000

  DEBUG CLS, "Recording..."

  FOR eeIndex = Records TO RecordsEnd STEP 2

    PULSIN 6, 1, x
    PULSIN 7, 1, y

    x = (x MIN 1875 MAX 3125) - 1875 ** 10538
    y = (y MIN 1875 MAX 3125) - 1875 ** 10538

    WRITE eeIndex, x
    WRITE eeIndex + 1, y

  NEXT

  FREQOUT 4, 200, 4000

  DEBUG CR, "End of records.",
        CR, "Press Enter for menu..."
  DEBUGIN char
  RETURN
```

The piezospeaker also beeps twice at a higher pitch right at the beginning of the datalogging. One important feature of this ten-second countdown is that you can stop the

datalogging before it starts by simply pressing and releasing your board's Reset button. Then, to restart the countdown, just press and release the Reset button again.

Example Program: DatalogAcceleration.bs2



Free Download! This program is available as a free .bs2 file download from the Smart Sensors and Applications Product Page at www.parallax.com.

This program takes and stores 500 accelerometer x and y-axis measurements in about 15 seconds. This equates to a sampling rate of about 33 measurements per second. This is good for a variety of measurements. To measure longer and slower processes, the **`Record_Data`** subroutine can be slowed down with a **`PAUSE`** command.

- √ Open and run DatalogAcceleration.bs2.
- √ Click the Debug Terminal's Transmit windowpane.
- √ Type R to start recording, and tilt your accelerometer this way and that for fifteen seconds.
- √ When prompted, press Enter to return to the program's menu.
- √ Type D to display the measurements. Review them and verify that they correspond to how you tilted the accelerometer.
- √ Disconnect your board from the serial cable. If it starts beeping as you do so, press and release the reset button to make it stop.

When you are ready to start tilting the accelerometer for fifteen seconds, press and release the Reset button. The datalogger will beep for a ten second countdown, then end with two higher-pitched beeps signaling the start of the datalogging. It will make a single high-pitched beep when it's finished.

- √ Press and release the reset button. Wait the ten seconds, then tilt your accelerometer in a pattern that you can remember, for 15 seconds.
- √ Plug your accelerometer back into your computer. If it starts beeping, press and release the reset button to stop the countdown.
- √ Click the BASIC Stamp Editor's Run button to download the program to the BASIC Stamp and refresh the Debug Terminal's Menu display.
- √ Type D to display the datalogged measurements.
- √ Compare them to the directions you tilted the board and make sure they correspond.

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - DatalogAcceleration.bs2
' Datalogs 500 x and y-axis acceleration measurements.

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

' -----[ DATA Directives ]--------------------------------------------------

Reset          DATA    0
Records        DATA    (1000)
RecordsEnd     DATA

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

char           VAR     Byte
eeIndex        VAR     Word
value          VAR     Word
x              VAR     value
y              VAR     Word

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

Init:

READ Reset, value
value = value + 1
WRITE Reset, value

IF value // 2 = 0 THEN

  FOR char = 10 TO 0
    DEBUG CLS, "Datalogging starts", CR,
               "in ", DEC2 char, " seconds",
               CR, CR,
               "Press/release Reset", CR,
               "for menu..."
    FREQOUT 4, 50, 3750
    PAUSE 950
  NEXT

  GOSUB Record_Data

ENDIF

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

DO

  DEBUG CLS,
        "Press/Release Reset", CR,
```

```
        "to arm datalogger ", CR, CR,
        " - or - ", CR, CR,
        "Type C, R or D", CR,
        "C - Clear records", CR,
        "R - Record records", CR,
        "D - Display records", CR,
        ">"

  DEBUGIN char
  DEBUG CR

  SELECT char
    CASE "C", "c"
      GOSUB Clear_Data
    CASE "R", "r"
      GOSUB Record_Data
    CASE "D", "d"
      GOSUB Display_Data
    CASE ELSE
      DEBUG CR, "Not a valid entry.",
            CR, "Try again."
      PAUSE 1500
  ENDSELECT

LOOP

' -----[ Subroutine - Clear_Data ]-----------------------------------------

Clear_Data:
  DEBUG CR, "Clearing..."
  FOR eeIndex = Records TO RecordsEnd
    WRITE eeIndex, 0
  NEXT
  DEBUG CR, "Records cleared."
  PAUSE 1000
  RETURN

' -----[ Subroutine - Record_Data ]-----------------------------------------

Record_Data:

  FREQOUT 4, 75, 4000
  PAUSE 200
  FREQOUT 4, 75, 4000

  DEBUG CLS, "Recording..."

  FOR eeIndex = Records TO RecordsEnd STEP 2

    PULSIN 6, 1, x
    PULSIN 7, 1, y
```

```
    x = (x MIN 1875 MAX 3125) - 1875 ** 10538
    y = (y MIN 1875 MAX 3125) - 1875 ** 10538

    WRITE eeIndex, x
    WRITE eeIndex + 1, y

  NEXT

  FREQOUT 4, 200, 4000

  DEBUG CR, "End of records.",
        CR, "Press Enter for menu..."
  DEBUGIN char

  RETURN

' -----[ Subroutine - Display_Data ]-----------------------------------------

Display_Data:

  DEBUG CR, "Index  x-axis  y-axis",
        CR, "-----  ------  ------",
        CR
  FOR eeIndex = Records TO RecordsEnd STEP 2
    READ eeIndex, x
    x = x - 100
    READ eeIndex + 1, y
    y = y - 100
    DEBUG DEC eeIndex, CRSRX, 7, SDEC x, CRSRX, 14, SDEC y, CR
  NEXT
  DEBUG CR, "Press Enter for menu..."
  DEBUGIN char
  RETURN
```

Your Turn - Datalogging Rotation Angle

Chapter 3, Activity #5 introduced vertical rotation measurements with the accelerometer. Since binary radians are values from 0 to 255, you can store a single angle measurement in one EEPROM byte. This will double the number of measurements the application will take. It only takes a few modifications to DatalogAcceleration.bs2 to make it store rotation angle instead. Here's how:

- √ Save DatalogAccleration.bs2 as DatalogAngle.bs2.
- √ Update the comments in the Title section.

√ Remove the **STEP 2** argument from the **FOR**...**NEXT** loops in the **Record_Data** and **Display_Data** subroutines.

√ In the **Record_Data** subroutine, replace these two **WRITE** commands:

```
WRITE eeIndex, x
WRITE eeIndex + 1, y
```

...with this **ATN** operation and **WRITE** command:

```
value = x ATN y
WRITE eeIndex, value
```

√ Modify the display heading in the **Display_Data** subroutine so that it looks like this:

```
DEBUG CR, "Index  angle ",
      CR, "-----  ------",
      CR
```

√ Replace these four commands:

```
READ eeIndex, x
x = x - 100
READ eeIndex + 1, y
y = y - 100
```

...with these two:

```
READ eeIndex, value
value = value */ 361
```

√ Save your changes and test the modified program.

ACTIVITY #5: RC CAR ACCELERATION STUDY

This activity demonstrates how to use the program DatalogAcceleration.bs2 from the previous activity to analyze the acceleration forces on a radio-controlled (RC) car during a variety of maneuvers. This activity also demonstrates how these datalogged acceleration forces can be used to track and plot the car's velocity and position. Although the actual equipment and calculations are somewhat more involved, deriving position from successive acceleration measurements is a component of the inertial guidance systems employed in rockets and spaceships. These systems use a combination of the 3-

axis version of the acceleration measurements covered in this activity along with gyroscopes that measure the vehicle's rotation.

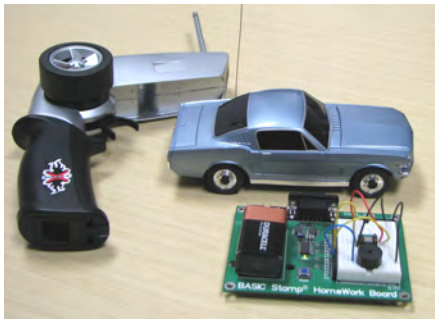
Parts, Equipment and Circuit Diagrams

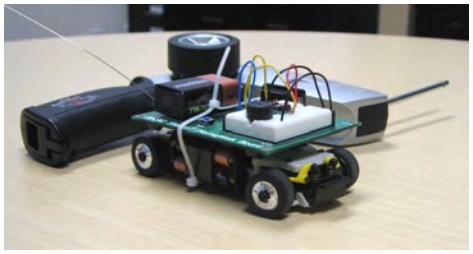
In addition to the parts for Activity #4, you will need an RC car and controller, which are not included in the Smart Sensors and Applications Parts Kit. The circuit diagrams that should be built on your board are at the beginning of Activity #4 in this chapter.

Hardware and Setup

Figure 6-4a shows an inexpensive RC car that can be obtained at many hobby shops and retail electronics outlets. Figure 6-4b shows how the board was mounted. Rubber feet were affixed to the underside of the board in a way that prevented any of its electrical connections from coming in contact with any of the RC car's electrical metal parts. Another option would be to use double-stick tape to affix the board to the roof of the plastic shell. The board was oriented with the breadboard toward the front of the car.

Figure 6-4: RC Car with Acceleration Datalogger

a. 

b. 



Avoid accidental short-circuits. Make sure your board is mounted on the car so that exposed metal underneath the board has no way of coming in contact with any of the RC car's metal parts or electrical connections.

How it Works

Figure 6-5 shows a graph of the accelerometer's y-axis measurements as the car accelerated forward, slowed to a stop, and then accelerated backwards. The

measurements were acquired with DatalogAcceleration.bs2 from Activity #4. After displaying them in the Debug Terminal, they were shaded, copied and pasted into Windows Notepad. From there, they were imported into the Microsoft Excel spreadsheet program and then graphed.

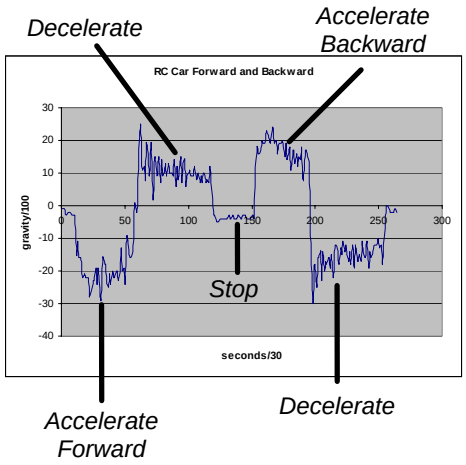


Figure 6-5
RC Car Accelerometer Y-Axis Measurements

The reason the forward acceleration is negative is because the y_m sensing axis is pointing to the back of the RC car as shown in Figure 6-6. So, as the car is accelerating forward, the acceleration is negative. When a car slows down, it is actually accelerating backwards. This is shown in Figure 6-5. First, the car accelerated forward, then it applied the brakes and slowed down (decelerated). The y measurement was positive, so acceleration was negative. After a brief stop, the car accelerated backwards. Notice that the y is again positive. Then, when it slows down (decelerates) from its backwards speed to stop again, the car is, in effect, accelerating forward, and the y measurement is negative again.

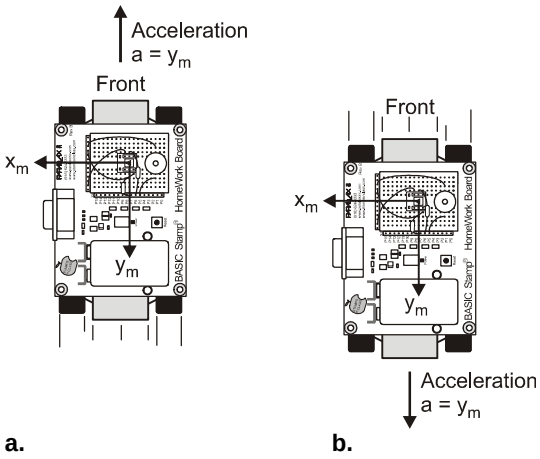


Figure 6-6
RC Car Acceleration vs Accelerometer Sensing Axes

If you're driving a car, when the car accelerates forward, you can feel the seat pushing you forward. Well, if you make a sharp right turn, the left side of the car pushes you to the right. That's because you are accelerating right as you turn. This is shown in Figure 6-7, which illustrates how an object can be traveling forward at a constant velocity, and to make it turn, it always has to be accelerating toward the center of the circle it is traveling in.

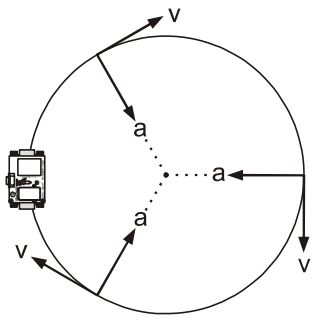


Figure 6-7
Traveling in a Circle

This causes continuous acceleration toward the center.

Figure 6-8 shows a graph of the accelerometer's x-axis measurements as the RC car is first driven in circles turning left, then in circles turning right. Notice how the x-axis measurement shows positive acceleration as the RC car circles to the left, and negative acceleration as the car circles to the right.

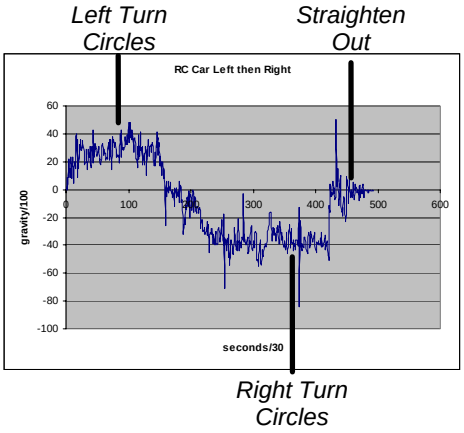
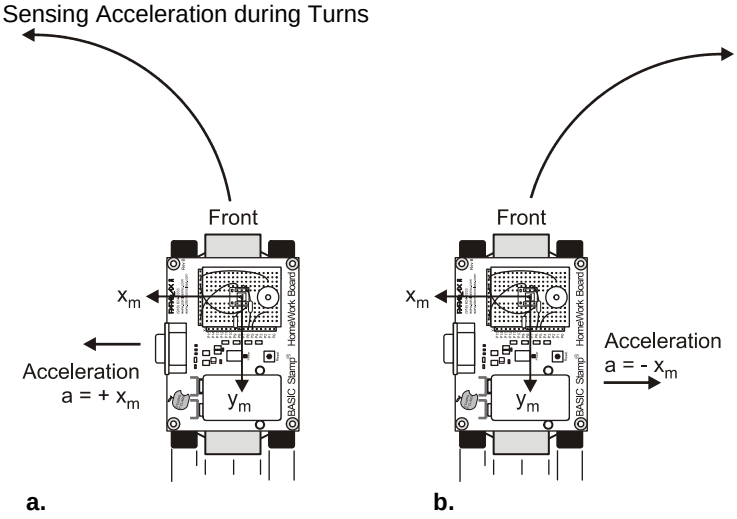


Figure 6-8
RC Car Accelerometer Measurements while Driving in Circles

Figure 6-9 shows how the accelerometer's x-axis is oriented, and the acceleration it measures. For a left turn, the car is accelerating to the left, which for the accelerometer is a positive x-axis acceleration measurement. When it turns right, acceleration is in the opposite direction of the positive x-axis, so the x-axis measurement it negative.

Figure 6-9: Sensing Acceleration during Turns



Procedure

The procedure for measuring and then graphing RC car acceleration is as follows.

- √ Attach your board to the RC car.
- √ Download DatalogAcceleration.bs2 into the BASIC Stamp.
- √ Set the car down in an open area and press/release the board's Reset button.
- √ Wait for the countdown to indicate that datalogging has started.
- √ Drive the car through these maneuvers, in about 15 seconds:
  - o Accelerate the car forward, then come to a stop.
  - o Accelerate the car backward, then come to a stop.
  - o Drive in a figure-eight.
- √ When the board beeps again (after about fifteen seconds) it means the datalogging is over. Connect the board back to your PC.
- √ Run DatalogAcceleration.bs2 again.
- √ Click the Debug Terminal's Transmit windowpane.
- √ Type D to display the data.
- √ Use your mouse to shade the table headings and all the measurements in the Debug Terminal's blue Receive windowpane. (Don't shade the menu.)
- √ Press CTRL + C to copy the records.
- √ Open Notepad.
- √ Click Edit and select Paste.
- √ Save the file.

These next instructions explain how to import the .txt file into Microsoft Excel 2002 and graph it. If you are using a different spreadsheet program, the keywords such as space delimited, XY scatter plot may provide leads on how to accomplish it with your particular spreadsheet software.

- √ In Excel, click File and select Open.
- √ In the files of type field, select All files (*.*).
- √ Find the .txt file you saved with notepad, select it, and click the Open button.
- √ In Text Import Wizard Step 1, click the Delimited radio button, then click Next.
- √ Click the checkbox next to Space to indicate that the file is space delimited.
- √ Make sure the checkbox for "Treat consecutive delimiters as one" box is also checked, then click next.
- √ Make sure the radio button for General column data format is selected, then click finish.

√ Your spreadsheet should be three columns wide and about 503 rows long.

The next step, which is also documented for Microsoft Excel 2002, is to run the chart utility and tell it what to graph and how you want it to look.

√ Place the cursor in a cell somewhere to the right of your three columns of data.
√ Click Insert and select Chart.
√ In the Standard Types tab, select XY (Scatter). Also click the graphic that configures it to "Scatter with data points connected to smoothed Lines without markers". Then, click Next.
√ Assuming your y-axis data begins in C3 and ends in C503, type C3..C503 in the Data range. Click the radio button next to Columns to indicate that the series of data points is in a column. Then, click Next.
√ Fill in the chart title and axis information, then click Finish.
√ Repeat for the x-axis.



Only portions of each graph are relevant. Keep in mind that the data that will make sense for the y-axis is the portion of time the car accelerated forward and backward. Likewise, the part of the graph that will make sense for the x-axis is the portion when the car was turning.

Graphing the Car's Position and Velocity

If you know the initial position and velocity of an object, you can use the acceleration during a period of time to calculate its position. These calculations can be made iteratively in a spreadsheet to plot the velocity and the path of the RC car.

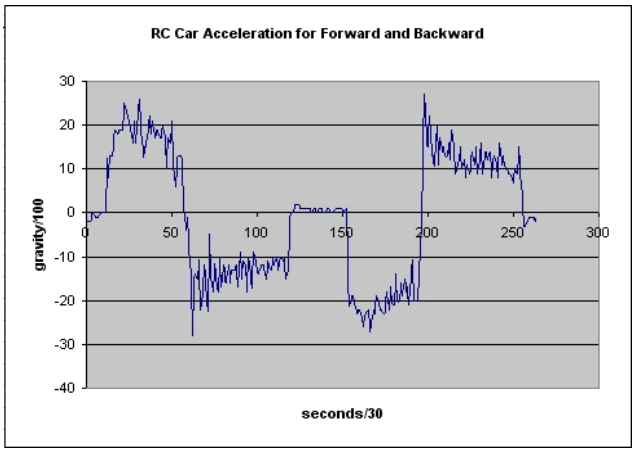


Downloading the Spreadsheet. The MS Excel spreadsheets used to plot these graphs are available for download from the Smart Sensors and Applications pages at www.parallax.com. Download the spreadsheet and examine the equations in the various columns along with the settings for each plot.

For example, the acceleration plot in Figure 6-10 shows a plot of the RC car as it accelerates forward and backward. (The spreadsheet was modified so that positive values indicate forward acceleration and negative values indicate backward acceleration or deceleration.) So, this graph shows that the car accelerated forward at an average of around 0.16 g for a little under 2 seconds. Then, it decelerated at an average of around

1.4 g for a little over 2 seconds. Then it rested for about 1 second. After that, it accelerated backward, and then decelerated (accelerated forward) to a second stop.

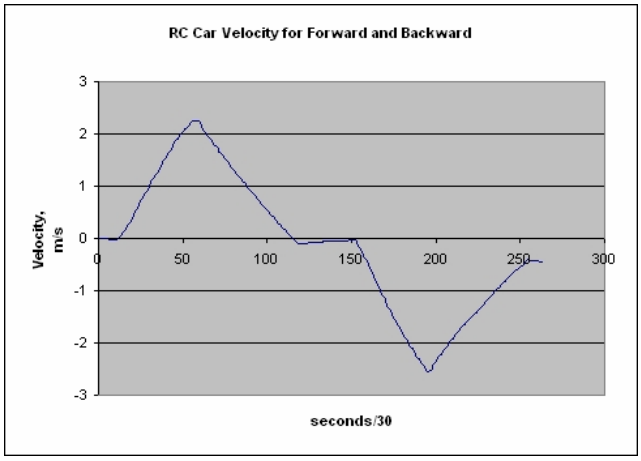
Figure 6-10: Acceleration Graph Modified with Positive Acceleration Indicating Forward



Selecting Data to Graph. Right-click the line in the plot with the title "RC Car Acceleration for Forward and Backward." Then choose source data and click the Series tab. Note that the series being plotted is from F229 to F492. This is the second of two forward/backward tests that were performed during the datalogging session. The same applies to the Velocity and Position graphs.

A column with an equation was added to the spreadsheet that calculates the change in velocity for each acceleration measurement. The equation for velocity in a straight line is **v = v0 + at**. That's the initial velocity (v0) plus the product of the acceleration (a) and the duration of that acceleration (t). Adding a column to the spreadsheet that recalculates velocity between each acceleration measurement makes it possible to graph the velocity as shown in Figure 6-11. As expected, when the car accelerates forward, its velocity increases. Then, when it slows down, its velocity decreases. As it accelerates backward, its velocity decreases further (increases in the negative direction). Then, as it slows its backward motion, its velocity returns to almost zero.

Figure 6-11: Velocity Graph Derived from Initial Position and Acceleration Data



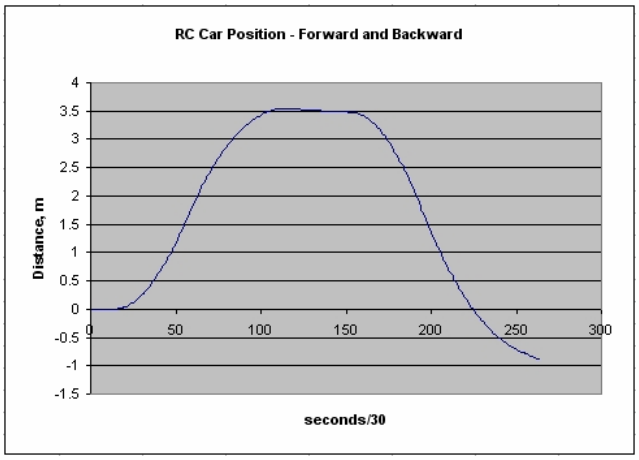
The calculations for this plot are made in the spreadsheet's column-F. If you click cell F-17 in the spreadsheet, this equation should appear in the function field:

```
=F16 + (0.03*9.8*E17/100)
```

In this case, F16 is the cell just above F17, and it has the previous velocity. This previous velocity is used as $\mathbf{V_0}$ for the sample interval. 0.03 is **t**, the time between samples, and 9.8 * E17 / 100 takes the E17 measurement, which is in hundredths of a g and converts it to meters per second (m/s^2). Dividing by 100 takes the value from hundredths of a g down to g and then multiplying by 9.8 converts from g to m/s^2. That's because 1 g is approximately 9.8 m/s^2.

With columns in the spreadsheet for acceleration and velocity, it is now possible to also keep track of the car's position using the equation $\mathbf{s = s_0 + v_0t + {}^1\!/_2\, at^2}$. That is, the position of the car (**s**) is equal to the initial position ($\mathbf{s_0}$) plus the product of initial velocity and time ($\mathbf{v_0t}$), plus half the product of acceleration and the square of time ($\mathbf{{}^1\!/_2\, at^2}$). The resulting graph of position shown in Figure 6-12 is surprisingly accurate. The car did in fact go forward to about the 3.5 meter mark before stopping. Then, it backed up and came to rest almost a meter behind where it started.

Figure 6-12: Position Graph Derived from Initial Position, Initial Velocity and Acceleration Data



The equation that calculates position in the G17 cell is:

```
=G16+(F16*0.03)+((0.5*E17*9.8/100)*(0.03^2))
```

G16 is the position after the previous sample, which is $\mathbf{S_0}$, the initial position. F16*0.03 is $\mathbf{v_0 t}$, initial velocity multiplied by time. (0.5*E17*9.8/100)*(0.03^2) is $^1/_2\ \mathbf{at^2}$, where **t** is again 0.03 seconds.

While this technique is pretty accurate over short periods of time, some error creeps into each measurement from several sources. Rough surfaces and vibration will effect the acceleration measurements. Also, while the equations assume the acceleration between each measurement is constant, in many cases, the acceleration will change during the time between each sample. In addition, each accelerometer measurement will tend to be a few percent off because of the nature of the MX2125. The MX2125's datasheet (available from Memsic's web site - www.memsic.com) explains these errors, the largest of which are called zero offset and sensitivity errors. They will vary from one chip to the next, and they are also influenced by temperature. Taking precision measurements with the MX2125 involves an A/D converter, a floating point coprocessor, and data collected from calibration tests. This calibration procedure is outside the scope of this text. To find out more about this topic, consult *#AN-00MX-002 Thermal Acceleromters Temperature Compensation*, which is available at Memsic's web site.

Your Turn - Logging Your RC Car's Acceleration/Velocity/Position

As mentioned earlier, the MS Excel spreadsheets used to plot these graphs are available for download from the Smart Sensors and Applications pages at www.parallax.com. Download the spreadsheet and examine the equations in the various columns along with the settings for each plot. Then, experiment with plotting data gathered from your own RC vehicle. Whatever data you plot should start from a known position with the car at rest. That way you know the initial position (s_0), and more importantly, the initial velocity, v_0 is 0 m/s.

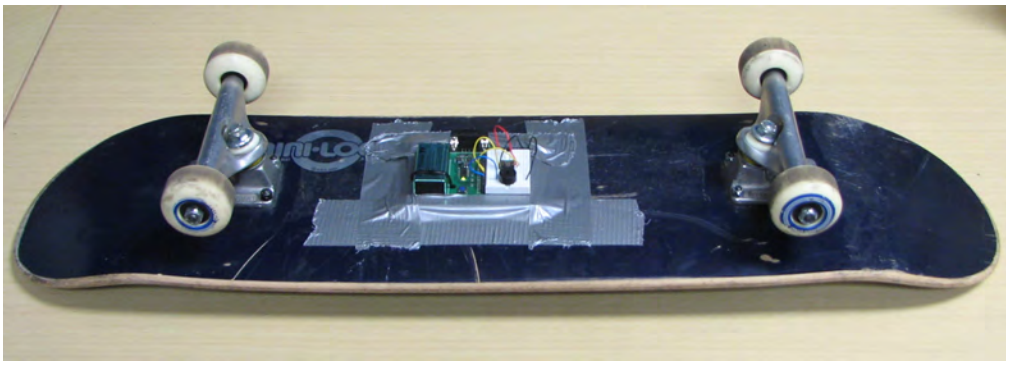
ACTIVITY #6: SKATEBOARD TRICK ACCELERATION STUDY

This activity looks at a second acceleration study example. This one datalogs a skateboard trick called the ollie. The setup for datalogging the ollie shown in Figure 6-13 is a BASIC Stamp HomeWork Board duct taped to the underside of a skateboard.



This Activity is included for illustration purposes only – the reader is not expected to get on a skateboard! This is just an example of how the reader may use the accelerometer to do motion studies with their own favorite hobbies or sports activities; the author happens to be a skateboarder. With all your BASIC Stamp applications, use common sense and appropriate protective gear, and conduct experiments at your own risk (see Disclaimer of Liability on the reverse of the title page).

Figure 6-13: BASIC Stamp HomeWork Board Duct-taped to the Author's Skateboard



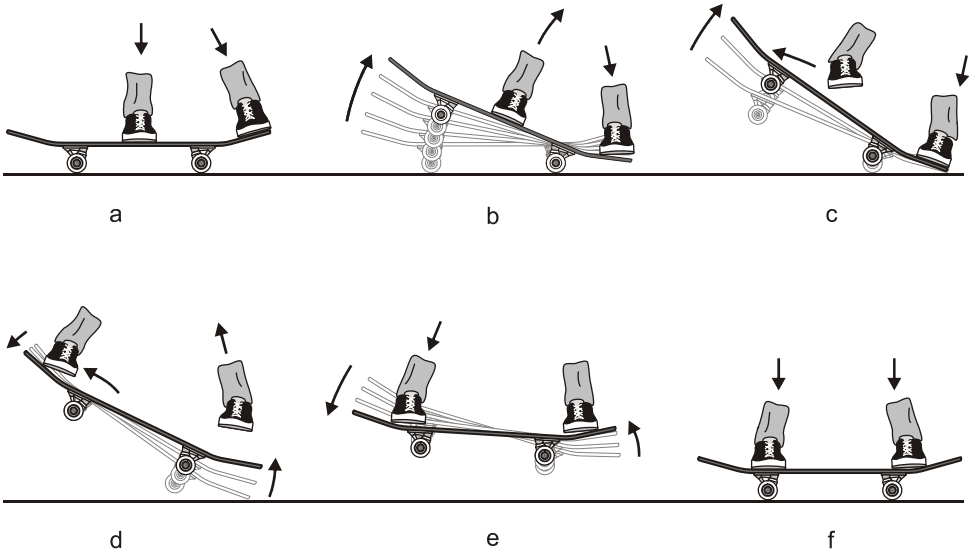
About the Ollie

The first documented ollie was done by Alan (Ollie) Gelfand in the late 1970s. Gelfand pioneered it in ramps and bowls. The flatland version of the ollie evolved in the early 1980s. When a skater does an ollie, he jumps, and it looks like his board is attached to his feet, even though it's not. Regardless of the environment or skating style, most skateboard tricks today are variations of the ollie.

Ollie Mechanics

Figure 6-14 shows the mechanics of an ollie. As the skater jumps, (a) his feet are both pushing the board down. Just before the skater is about to become airborne, (b) he lifts his front foot and at the same time extends his back foot to tiptoe, and the tail of the board smacks the concrete. The momentum of the front of the board keeps it rising (c), and the skater now lifts his back foot, and kicks his front foot forward. This causes the back of the board to rise (d), and move slightly forward. As the deck meets the skater's back foot (e), the skater applies just enough pressure to keep the board against his feet as it falls back to the ground (f). The highest ollie to date, performed by Danny Wainwright, was in excess of five feet high.

Figure 6-14: Ollie Mechanics



Graphing Ollie Acceleration

Figure 6-15 shows a graph of the accelerometer's y-axis for the first of two ollies that were datalogged with the next example program. Each step from Figure 6-14 is marked on the graph.

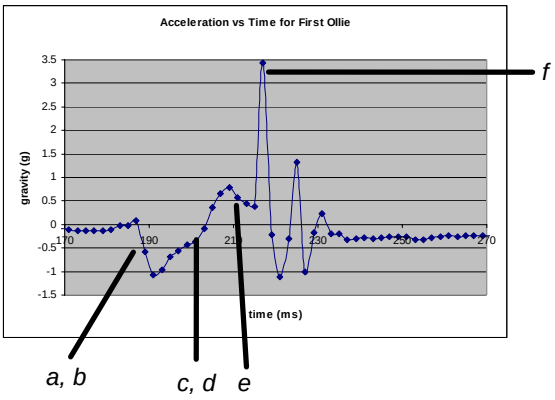


Figure 6-15
Acceleration during an Ollie

This first ollie was a little deficient in Figure 6-14 steps b and c, so the back of the board didn't quite meet the back foot in step e. Note that the impact of the board during step f was 3.5 g. The highs and lows that follow step f resemble the oscillations when a bell is struck. This is partially due to the board's vibration and partially due the turbulence of the gas inside the accelerometer caused by the impact.

Figure 6-16 shows the data for a slightly better ollie. It was a little higher, and it made it to step e gracefully. Notice that step a to b is steeper, and gets to -1.25 g before rising to over 1 g for steps c and d. These values, which are larger than the ones from the previous graph, indicate a higher ollie. Notice also that the impact was below 3 g, because the skater was not trying to catch up with the board on the way down.

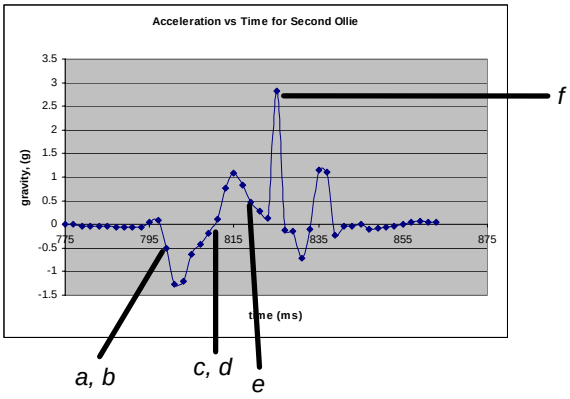


Figure 6-16
Graph of a Slightly Better Ollie

6

Datalogging an Ollie

Figure 6-17 shows how the accelerometer's y sensing axis is aligned to sense the skateboard's various tilts and rotations. This is the only axis we want to log in the next example program.

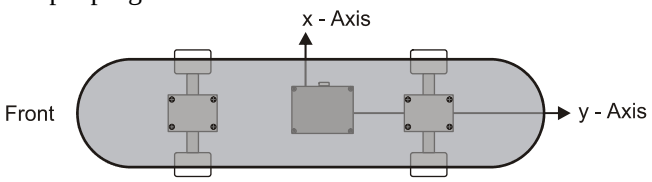


Figure 6-17
Accelerometer's Sensing Axes on the Skateboard

The program from Activity #4 was modified to store just the raw y-axis accelerometer measurements with no scale or offset. The value of **y** would range from 1875 to 3125 (for ± 1 g) if no acceleration is involved. When the acceleration measurement is 3.5 g, that would result in a measurement of 4687. In any event, these are word values, and so the **WRITE** command in the **Record_Data** subroutine has to be modified so that it stores word variables. Since a word takes up two bytes, the **FOR**...**NEXT** loop still has to count in steps of 2.

```
FOR eeIndex = Records TO RecordsEnd STEP 2
  PULSIN 7, 1, y
  WRITE eeIndex, Word y
NEXT
```

Similar modifications are made to the **FOR**...**NEXT** loop in the **Display_Data** subroutine shown here.

```
FOR eeIndex = Records TO RecordsEnd STEP 2

  READ eeIndex, Word y
  DEBUG DEC eeIndex, CRSRX, 7, SDEC y, CR

NEXT
```

Example Program: DatalogYaxisUnscaled.bs2

This next example program was used to log the data graphed in Figure 6-15 and Figure 6-16. It gives you about ten seconds of datalogging, which is enough time for two or three ollies. Moving the data to a spreadsheet and graphing it is based on the procedure in Activity #5. The spreadsheet was modified to generate the graphs shown in this activity by adding a column with a formula that takes the y-axis data, subtracts 2500 from it, and then divides it by 625. This gives a measurement in units of earth-gravity (g).

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - DatalogYaxisUnscaled.bs2
' Datalogs 500 word size y-axis acceleration measurements.

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

' -----[ DATA Directives ]-------------------------------------------------

Reset          DATA    0
Records        DATA    (1000)
RecordsEnd     DATA

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

char           VAR     Byte
eeIndex        VAR     Word
value          VAR     Word
x              VAR     value
y              VAR     Word

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

Init:

READ Reset, value
value = value + 1
WRITE Reset, value
```

```
IF value // 2 = 0 THEN

  FOR char = 10 TO 0
    DEBUG CLS, "Datalogging starts", CR,
               "in ", DEC2 char, " seconds",
               CR, CR,
               "Press/release Reset", CR,
               "for menu..."
    FREQOUT 4, 50, 3750
    PAUSE 950
  NEXT

  GOSUB Record_Data

ENDIF

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

DO

  DEBUG CLS,
        "Press/Release Reset", CR,
        "to arm datalogger ", CR, CR,
        " - or - ", CR, CR,
        "Type C, R or D", CR,
        "C - Clear records", CR,
        "R - Record records", CR,
        "D - Display records", CR,
        ">"

  DEBUGIN char
  DEBUG CR

  SELECT char
    CASE "C", "c"
      GOSUB Clear_Data
    CASE "R", "r"
      GOSUB Record_Data
    CASE "D", "d"
      GOSUB Display_Data
    CASE ELSE
      DEBUG CR, "Not a valid entry.",
            CR, "Try again."
      PAUSE 1500
  ENDSELECT

LOOP

' -----[ Subroutine - Clear_Data ]-------------------------------------------

Clear_Data:
```

```
  DEBUG CR, "Clearing..."

  FOR eeIndex = Records TO RecordsEnd
    WRITE eeIndex, 0
  NEXT

  DEBUG CR, "Records cleared."
  PAUSE 1000

  RETURN

' -----[ Subroutine - Record_Data ]-----------------------------------------

Record_Data:

  FREQOUT 4, 75, 4000
  PAUSE 200
  FREQOUT 4, 75, 4000

  DEBUG CLS, "Recording..."

  FOR eeIndex = Records TO RecordsEnd STEP 2

    PULSIN 7, 1, y

    WRITE eeIndex, Word y

  NEXT

  FREQOUT 4, 200, 4000

  DEBUG CR, "End of records.",
        CR, "Press Enter for menu..."
  DEBUGIN char

  RETURN

' -----[ Subroutine - Display_Data ]----------------------------------------

Display_Data:

  DEBUG CR, "Index  x-axis  y-axis",
        CR, "-----  ------  ------",
        CR

  FOR eeIndex = Records TO RecordsEnd STEP 2

    READ eeIndex, Word y
    DEBUG DEC eeIndex, CRSRX, 7, SDEC y, CR
```

```
NEXT

DEBUG CR, "Press Enter for menu..."
DEBUGIN char

RETURN
```

Your Turn - What Makes a High Ollie?

It would be interesting to datalog and compare different skaters' ollies. The best way to do it would be to take video of each ollie, and then watch the video and examine the graph at the same time. Another thing that can be measured is the time in the air, which is the time between steps a and f in the graphs.

ACTIVITY #7: BICYCLE DISTANCE

Figure 6-18 on the next page shows how the board and accelerometer can be mounted inside a bicycle wheel, in order to measure bicycle distance. As the bicycle is upright, this might at first seem like an angle of rotation problem, like in Chapter 3, Activity #5. However, there is also acceleration toward the center of the wheel that the axes will measure. This is because the accelerometer is traveling in a circular path, just like the RC car from the previous activity. This acceleration toward the center of the wheel will be different at different speeds, and will result in skewed angle measurements. The accelerometer measurements will also be affected when the bike rider applies the brakes, speeds up, and leans into turns. In addition, what criteria should be used to add one to the number of full circles the bike wheel has turned? This activity introduces hysteresis as a way of measuring wheel rotation. It also demonstrates how the datalogging techniques used in earlier activities can be used to examine each of these issues and test for prototype reliability.



Do not let the metal on the underside of your board come into contact with the spokes. Use an insulating material such as plastic, cardboard, or esd foam to insulate the underside of the board from the spokes.

a.  b. 

Figure 6-18
HomeWork Board with Accelerometer Mounted on Bicycle Wheel

Counting Wheel Revolutions with Hysteresis

One problem with counting wheel revolutions is making sure that the program doesn't advance the count if the wheel hasn't turned full circle. The most common mistake that is made when measuring wheel revolutions is setting a single threshold. What if the rider is waiting at a stop light, and is moving his/her bike back and forth by an inch or two? If there is a single threshold, the wheel revolution counter will keep increasing every time the rider rocks back and forth.

The next example program demonstrates a way of solving this problem with hysteresis. Hysteresis is the process of setting two different values that have to be crossed before a change in state occurs. In our case, the change of state is an increase in the wheel revolution count. With hysteresis, the measurement must fall below a low value, and then the program waits until it has risen up above a higher value before acknowledging an upward change. Then, the measurement will have to go below the low threshold again before a change from high to low is acknowledged. Each time the program acknowledges that the measurement went below the low value and then above the high value, it increases the wheel revolution count by 1.

Here is some code that performs hysteresis. In the first of the two nested **`DO`**...**`LOOP`** blocks, the program waits until the y axis rises above 2650. Then, the second of the two nested **`DO`**...**`LOOP`** commands waits until the y axis measurement drops below 2350. Only then will it add 1 to the **`counter`** variable. After that, the program makes the piezospeaker beep, and then repeats the outer **`DO`**...**`LOOP`**. At this point, the program is back to waiting for the y axis measurement, which was below 2350 to rise back above

2650 again. Keep in mind that this is not necessarily the optimum way to measure wheel revolutions. That's for you to determine.

```
DO

  DO UNTIL y > 2650
    PULSIN 7, 1, y
  LOOP

  DO UNTIL y < 2350
    PULSIN 7, 1, y
  LOOP

  counter = counter + 1
  FREQOUT 4, 200, 3750

LOOP
```



Deadband: The range between 2350 and 2650 in the code block above is referred to as deadband.

Example Program: TestWheelCounter.bs2

- √ Mount your board inside a bicycle wheel as shown in Figure 6-18. Make sure to keep a good insulator between the spokes and the underside of the board.
- √ Enter, save, and run TestWheelCounter.bs2.
- √ Spin the wheel, and verify that it beeps once per revolution.

```
' Smart Sensors and Applications - TestWheelCounter.bs2
' Tracks bicycle wheel revolutions.

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

x              VAR     Word
y              VAR     Word
counter        VAR     Word

DEBUG CLS

DO

  DO UNTIL y > 2650
    PULSIN 7, 1, y
  LOOP
```

```
  DO UNTIL y < 2350
    PULSIN 7, 1, y
  LOOP

  counter = counter + 1
  FREQOUT 4, 200, 3750

LOOP
```

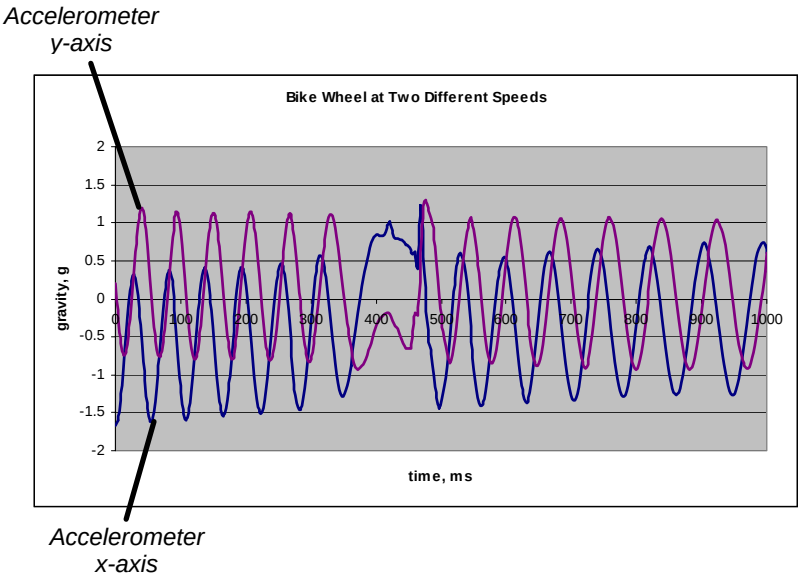
Datalogging Various Operating Conditions

It might seem at this point like the application is ready for some code that converts revolutions to distance, and maybe an LCD display and a couple of buttons for selecting LCD menu items. The problem here is we only examined the wheel turning two speeds. What about when the rider is leaning into sharp turns – does the acceleration change then? What about in cold and hot temperatures – will they cause the measurements to be different? It certainly wouldn't do to have a product on the market that only tracked bicycle distance some of the time. The product would get a bad reputation very quickly.

Figure 6-19 shows a datalogged acceleration study for the bicycle at two slightly different speeds. The area around 400 ms is where the wheel was slowed down. The important thing to note from this graph is the offset of the x and y-axis measurements. At the higher speed, the y-axis signal varied between 1 and −0.5 g while the x-axis measurements varied between −1.5 and 0.25 g. After slowing the wheel down, the y-axis measurement varied between 1 and -1 g, while the x-axis measurement varied between about 0.7 and −1.3 g.

Figure 6-19 is just a graph of two different speeds. True, the hysteresis code from TestWheelCounter.bs2 works under both of these conditions, but does it work under ALL conditions? With this kind of question, engineers might apply a few equations to predict the accelerations under various extreme conditions that they anticipate. Simulation software could also be used to verify the outcomes. Even if this kind of expertise is available, the product still has to be tested in a variety of "real life" conditions, especially to rule out the possibility of incorrect assumptions on the part of the engineers. That's where datalogging comes back into the picture. The actual prototype still has to be taken through the various conditions that it might experience on anybody's bicycle before it is safe to make the investment in the plastic case, a refined electronic design that features low cost parts, mass production and inventory costs.

Figure 6-19: Bicycle Wheel Acceleration Study



With this in mind, we are back to performing acceleration studies, under as many different situations as possible. Here is the program that was used to log the data for the graph in Figure 6-19. Notice that it is logging word-sized values for both the x and y-axis measurements. The spreadsheet is given the job of changing the raw accelerometer **`PULSIN`** measurements into gravity measurements.

Example Program: BikeWheelAcceleration.bs2

As a project, test the bicycle meter in different temperatures and riding conditions, turns, up hill, down hill, slow, fast, etc. Look for a sequence of changes in measurements that can be tracked regardless of the conditions. If there is not a hysteresis range for all conditions, your code may need to periodically update the most recent high and low values, and then look for hysteresis within that range.

```
' -----[ Title ]-------------------------------------------------------------
' Smart Sensors and Applications - BikeWheelAcceleration.bs2
' Datalogs 500 x and y-axis acceleration measurements.

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

```
' -----[ DATA Directives ]-----------------------------------------------------

Reset          DATA    0
Records        DATA    (1000)
RecordsEnd     DATA

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

char           VAR     Byte
eeIndex        VAR     Word
value          VAR     Word
x              VAR     value
y              VAR     Word

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

Init:

READ Reset, value
value = value + 1
WRITE Reset, value

IF value // 2 = 0 THEN

  FOR char = 10 TO 0
    DEBUG CLS, "Datalogging starts", CR,
               "in ", DEC2 char, " seconds",
               CR, CR,
               "Press/release Reset", CR,
               "for menu..."
    FREQOUT 4, 50, 3750
    PAUSE 950
  NEXT

  GOSUB Record_Data

ENDIF

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

DO

  DEBUG CLS,
        "Press/Release Reset", CR,
        "to arm datalogger ", CR, CR,
        " - or - ", CR, CR,
        "Type C, R or D", CR,
        "C - Clear records", CR,
        "R - Record records", CR,
        "D - Display records", CR,
```

```
        ">"

  DEBUGIN char
  DEBUG CR

  SELECT char
    CASE "C", "c"
      GOSUB Clear_Data
    CASE "R", "r"
      GOSUB Record_Data
    CASE "D", "d"
      GOSUB Display_Data
    CASE ELSE
      DEBUG CR, "Not a valid entry.",
            CR, "Try again."
      PAUSE 1500
  ENDSELECT

LOOP

' -----[ Subroutine - Clear_Data ]-------------------------------------------

Clear_Data:
  DEBUG CR, "Clearing..."
  FOR eeIndex = Records TO RecordsEnd
    WRITE eeIndex, 0
  NEXT
  DEBUG CR, "Records cleared."
  PAUSE 1000
  RETURN

' -----[ Subroutine - Record_Data ]-------------------------------------------

Record_Data:

  FREQOUT 4, 75, 4000
  PAUSE 200
  FREQOUT 4, 75, 4000

  DEBUG CLS, "Recording..."

  FOR eeIndex = Records TO RecordsEnd STEP 4

    PULSIN 6, 1, x
    PULSIN 7, 1, y

    WRITE eeIndex, Word x
    WRITE eeIndex + 2, Word y

  NEXT
```

```
  FREQOUT 4, 200, 4000

  DEBUG CR, "End of records.",
        CR, "Press Enter for menu..."
  DEBUGIN char

  RETURN

' -----[ Subroutine - Display_Data ]-----------------------------------------

Display_Data:

  DEBUG CR, "Index  x-axis  y-axis",
        CR, "-----  ------  ------",
        CR
  FOR eeIndex = Records TO RecordsEnd STEP 4
    READ eeIndex, Word x
    READ eeIndex + 2, Word y
    DEBUG DEC eeIndex, CRSRX, 7, SDEC x, CRSRX, 14, SDEC y, CR
  NEXT
  DEBUG CR, "Press Enter for menu..."
  DEBUGIN char
  RETURN
```

Your Turn

Another thing to examine is how vertical plane rotation measurements perform under the various bicycle wheel conditions.

- √ The Your Turn section of Activity #4 datalogs brad measurements. Use it to datalog your bicycle wheel rotation in brads.
- √ Graph the rotation over time under the various riding conditions discussed in this Activity.

Is there an angle measurement behavior for which hysteresis can be applied under all riding conditions?

SUMMARY

This chapter introduced a variety of accelerometer applications and datalogging techniques that can be used to study the accelerometer's measurements in various conditions, and in some cases, to refine your programs. When sighting the top of an object, vertical plane rotation measurements can be used with the distance to the object and some trigonometry to determine the object's height.

DATA directives with optional ***Symbol*** names were introduced as a way to simplify recordkeeping in datalogging programs. They can be used to define ranges of unused EEPROM program memory. Since ***Symbol*** names store the starting address of **DATA** directives, they are handy in **FOR**...**NEXT** loops that perform **READ**/**WRITE** operations over the range of EEPROM bytes defined by the beginning and ending **DATA** directives.

A technique was also introduced for using a **DATA** directive to set aside one byte for setting the program mode. Each time the program starts, an initialization routine reads the byte, adds one to it, and replaces the old value in EEPROM with the modified value. Each time the program is restarted by pressing and releasing the board's Reset button, the program can use the new value in EEPROM to select between different modes. For toggling a feature in the program on and off, an **IF**...**THEN** statement was used that examines whether the remainder of the value divided by two is zero. This makes it possible to start and stop datalogging without being connected to the computer.

Accelerometer applications with datalogging included RC car acceleration, skateboard trick measurements, and bicycle wheel measurements. Each of these employed a program that was a variation of the remote datalogging program introduced in Activity #4. The data displayed in the Debug Terminal was shaded, copied, and pasted into text files. The text files were then imported into a spreadsheet program and graphed. The graphs were analyzed to examine accelerations, tilts, and angles involved RC car, skateboard, and bicycle wheel motions.

Questions

1. What three pieces of information do you need to measure the height of a building from a distance?
2. What's the difference between **DATA (100)** and **DATA 20 (100)**?
3. What's wrong with this command? **WRITE eeIndex, 1000**. How can you fix it?

4. What other directives and commands have to be present for **`IF value // 2 = 0 THEN`**... to make it possible to toggle program modes with your board's Reset button?
5. What does the piezospeaker do in DatalogAcceleration.bs2?
6. How can you modify a **`DATA`** directive to make it set aside more values?
7. How does forward acceleration differ from forward deceleration?
8. When driving in circles at a constant velocity and radius, what direction is the acceleration?
9. How does the datalogging program that measures a skateboarder's ollie differ from the program that measures RC car motions? How are they similar?

Exercises

1. The top of a building was sighted to be 75° from a vantage point 15 m from the building and 1 m from the ground. How tall is the building?
2. Write a pair of **`DATA`** directives that reserve 1501 bytes. Use symbol names.
3. Write a **`FOR`**...**`NEXT`** loop that retrieves 751 words. Assume that your **`DATA`** directive ***`Symbol`*** names are **`StartData`** and **`EndData`**.
4. Modify a block of code in DatalogAcceleration.bs2 so that its countdown is five seconds.

Projects

1. Use Google to find the slope above which snow is likely to avalanche. Prototype a measuring device that warns you if a slope is too steep. This device could be used to replace a mechanical one commonly used in ski resorts.
2. Design a pedometer (step counter) prototype.

Solutions

Q1. (1) The height from which the measurement is taken, (2) the distance from the base of the building, and (3) the angle from horizontal at which the top of the building was sighted.

Q2. The **`DATA (100)`** sets aside 100 bytes in EEPROM; whereas, **`DATA 20 (100)`**stores the value 20 in each of the 100 bytes.

Q3. The **`WRITE`** commands stores byte values. To fix the command, you would have to insert the **`Word`** modifier before the value 1000. Keep in mind that you will have to increment **`eeIndex`** by 2 before storing the next value.

Q4. You would need: **`Reset DATA 0`**; **`value VAR Word`**; **`READ Reset, value, value = value + 1`**; **`WRITE Reset, value`**

Q5. It lets the user know what mode the device is operating in by emitting chirps (tones of certain durations and frequencies). The countdown before datalogging involves eleven 50 ms 3.75 kHz tones followed by two more pronounced higher pitched tones (75 ms 4 kHz tones). After datalogging, the piezospeaker emits a longer chirp (200 ms 4 kHz) to let the user know datalogging is complete.

Q6. Increase the value between parentheses that follows the **`DATA`** keyword.

Q7. If you are traveling forward and decelerating, it's the same as accelerating backwards, which is accelerating in the opposite direction of forward.

Q8. Toward the center of a circle.

Q9. This is a comparison of two example programs, DatalogAcceleration.bs2 against DatalogYaxisUnscaled.bs2. DatalogAcceleration.bs2 stores x and y values that are scaled down to 0 to 200 with 100 as 0 g. DatalogYaxisUnscaled.bs2 not only stores the unscaled version, it doesn't apply any offset either. It's just the raw y-axis measurement, which ranges from 1875 to 3125. In both programs, it takes a word value to store each measurement in DatalogAcceleration.bs2, it's a word (2 bytes) that stores the scaled and offset x and y-axis measurements. In DatalogYaxisUnscaled.bs2, each y-axis value takes an entire word. Even though the time it takes to store bytes is the same in both programs, some time between measurements is saved by DatalogYaxisUnscaled.bs2 because it doesn't take any time for a **`PULSIN`** command that would otherwise read the x-axis.

E1. object height = opposite + accelerometer height
= (adjacent × tan (75˚)) m + 1 m
= (15 × 3.73) m + 1 m
= 56.98 m

E2.

```
StartSymbol DATA (1500)
EndSymbol DATA
```

E3.

```
' Assume StartData and EndData have been correctly defined
FOR counter = StartData TO EndData
  READ counter, dataItem
  DEBUG dataItem ' Format wasn't specified, so nothing like DEC was
                 ' used. If DEC dataItem was used, it would still be
                 ' right because that's what the text examples used.
NEXT
```

E4.

```
  FOR char = 4 TO 0
    DEBUG CLS, "Datalogging starts", CR,
               "in ", DEC2 char, " seconds",
               CR, CR,
               "Press/release Reset", CR,
               "for menu..."
    FREQOUT 4, 50, 3750
    PAUSE 950
  NEXT
```

P1. Keywords used: avalanche prediction slope
Resulting helpful article used: http://en.wikipedia.org/wiki/Avalanche
Key information: Terrain section states that steepness below 25˚ or above 60˚ presents low risk and that peak risk is in the 35 to 45˚ slopes.

The hardware could involve an accelerometer, LCD, pushbuttons to select mode, and a piezospeaker for alarm. The key to the prototype would be to demonstrate that the device can detect certain levels of risk from, say, the base of the slope. Features can be added assuming the prototype is approved.

Note: Ski resorts regularly maintain their slopes by taking these measurements and then launching explosive charges into the hill to create small avalanches, thereby preventing a larger avalanche later. So even though it might seem risky to measure from the bottom of the slope, we are talking about a slope that is regularly maintained to prevent avalanches.

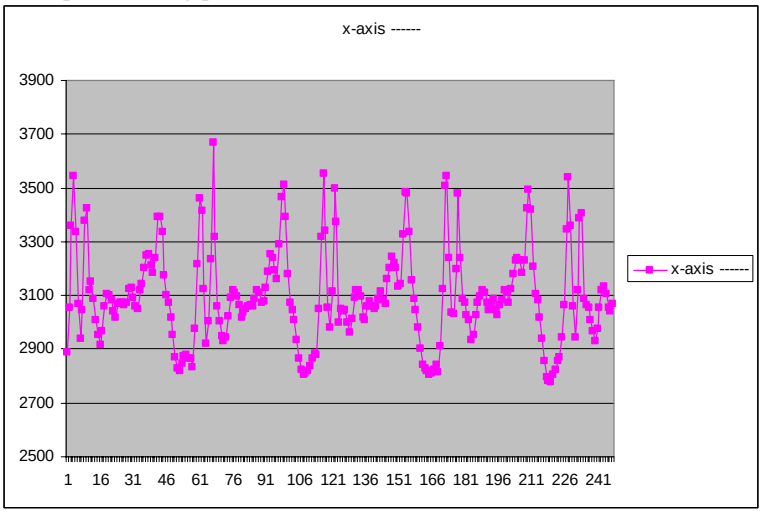
The prototype involves using your board to sight the top of the slope from the base, which turns out to be a simple modification of the application discussed in this Chapter's Activity #1. A **SELECT…CASE** statement can be added to the modified version of VertWheelRotation.bs2 that displays different messages on the LCD based on the measured angle. The **SELECT…CASE** statement might look like this when you're done:

```
SELECT angle
  CASE 0 to 25
    GOSUB Display_Pretty_Safe
  CASE 25 to 35, 45 to 60
    GOSUB Display_Some_Danger
  CASE 35 to 45
    GOSUB Display_Max_Danger
  ENDSELECT
```

Each subroutine would display a risk indication message after the measured angle.

P2. Load and run DatalogYaxisUnscaled.bs2. Clip the board to your belt at the hip so that the y-axis is vertical. Press and release your board's reset button and walk during the datalogging period. Follow the instructions in Activity #5 for plotting your datalogged points. You will probably want to graph 200 of the 1001 measurements. Look for a pattern, the graph should exceed some value and go below some value with each step. Here is an example. Notice that each step's acceleration drops below 2900 then exceeds 3300.

Example walking plot:



Use those values to define the **`StepLow`** and **`StepHigh`** constants in the program below:

```
' Smart Sensors and Applications - Ch6_Project2.bs2

' IMPORTANT, follow the instructions to come up with your
' own values.  You will find the Solutions section of Chapter 6.
' Examine the solution to Project 2 for information on how to
' determine your own StepLow and StepHigh constants.

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

StepLow CON 2900
StepHigh CON 3300

y       VAR Word
yOld    VAR Word
stepCnt VAR Word

yOld = 3300

PAUSE 250                       ' Debounce the power supply
SEROUT 14, 84, [22, 12]         ' Start LCD & clear display
PAUSE 5                         ' Pause 5 ms for clear display

DO

 PULSIN 7, 1, y

 IF (y <= 2900 AND yOld >= 3300) OR (y >= 3300 AND yOld <= 2900) THEN
   yOld = y
   stepCnt = stepCnt + 1
 ENDIF

 SEROUT 14, 84, [128, DEC5 stepCnt / 2]

LOOP
```

Chapter 7: LCD Bar Graphs for Distance and Tilt

Defining and displaying custom characters with the Parallax Serial LCD was introduced in Chapter 1, Activity #4. This chapter introduces some more custom character techniques, and then applies them to bar graph displays. These displays will indicate the distance of an object from the Ping))) ultrasonic sensor and the tilt of the Memsic 2125 Dual Axis Accelerometer.

ACTIVITY #1: CUSTOM CHARACTER SWAPPING

The Parallax Serial LCD can display up to eight custom characters at any given time. However, there can be many more than eight custom characters in your application, because custom characters can be defined and redefined as needed. The only limitation is that only eight can be displayed simultaneously, and eight is ample for most projects.

The place where you can define and store a library of more than eight custom character definitions is in the part of the BASIC Stamp EEPROM memory that is not used for program storage. Since PBASIC programs rarely fill the entire EEPROM memory, there is typically room for all the custom character definitions an application might need.

One powerful technique is to use just one of the LCD's eight custom character slots to display a sequence of custom character definitions that are stored in the BASIC Stamp EEPROM. This is especially useful for animation, but it will also be important for bar graph displays. This activity provides an animation example.

From EEPROM Storage to LCD Character Memory

The next example program will demonstrate a convenient way to store custom character definitions in the BASIC Stamp EEPROM. Two of the program's fifteen custom character definitions are shown below. Each definition gets a unique ***Symbol*** name, like **Char0**, **Char1**, **Char2**, and so on, up through **Char14**. Each of these ***Symbol*** names represents the EEPROM address of the first byte in the **DATA** directive. The subroutine that transfers the definitions from EEPROM to the LCD's custom character memory slots uses these ***Symbol*** names as a reference point for reading the bytes from EEPROM. After reading each byte from EEPROM, the subroutine sends it to the serial LCD.

```
                        .
                        .
                        .
Char4           DATA    %11111,         ' * * * * *
                        %00011,         ' 0 0 0 * *
                        %11011,         ' * * 0 * *
                        %11011,         ' * * 0 * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111          ' * * * * *
                           .
                           .
                           .
Char9           DATA    %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11011,         ' * * 0 * *
                        %11011,         ' * * 0 * *
                        %11000,         ' * * 0 0 0
                        %11111          ' * * * * *
                           .
                           .
                           .
```

The next example program fetches custom character definitions from EEPROM and sends them to the serial LCD using a subroutine named **Def_Cust_Char**. All you have to do before calling the subroutine is set the value of two variables: **custChar** and **charBase**. The **custChar** variable is for selecting which custom character slot to define (0, 1, 2...7). The **charBase** variable is used to tell the **Def_Cust_Char** subroutine where to look in EEPROM for the beginning of the character definition. For example, to transfer the **Char9** definition in the BASIC Stamp EEPROM to Custom Character 5 in the Parallax Serial LCD's character memory, use these three commands:

```
custChar = 5
charBase = Char9
GOSUB Def_Cust_Char
```

If your program has to choose among many different custom characters definitions, you can replace **charBase = Char9** with a **LOOKUP** command. Below is an example that chooses one of the three different custom character definitions in the next example program with a **LOOKUP** command. Depending on the value of **counter**, either **Char0**, **Char1** or **Char2** will be copied to the **charBase** variable.

```
DO
  counter = counter + 1
  counter = counter // 3

  ' Define custom character.
  custChar = 5
  LOOKUP counter,
        [Char0, Char1, Char2],
         charBase
  GOSUB Def_Cust_Char
    .
    .
    .
LOOP
```




How to make `counter` count 0, 1, 2, 0, 1, 2, ... without FOR...NEXT

The code block above is in a **DO**...**LOOP**, so it gets repeated indefinitely. The counter variable increases by 1 each time through a loop. The command **`counter = counter // 3`** uses the PBASIC modulus operator // to calculate the remainder of counter ÷ 3. By setting counter equal to the remainder of counter ÷ 3, it causes counter to only count to 2 before falling back to 0. The resulting sequence of values stored by counter is 0, 1, 2, 0, 1, 2, 0, 1, 2, ...

To display the custom character at a certain character, the next example program has a **`Disp_Cust_Char`** subroutine. This subroutine depends on three variables, **`line`**, **`custChar`**, and **`cursor`**. The **`line`** variable should be set to either **`Line0`** or **`Line1`**, which are defined in the example program's Constants section. Again, the **`custChar`** variable is a value between 0 and 7, which selects a custom character in the LCD's character memory. The **`cursor`** variable can be a value from 0 to 15 depending on how far from the LCD's left you want the character printed. For example, to print Custom Character 5 on the LCD's Line 0 at character 8, use these commands:

```
custChar = 5
line = Line0
cursor = 8
GOSUB Disp_Cust_Char
```

Since the next example program is just animating a character, a simple **`FOR`**...**`NEXT`** loop can be used to access each of the custom character **`DATA`** directives. The starting address of each **`DATA`** directive will be eight bytes after the next. Updating the LCD with each character definition in the sequence of **`DATA`** directives can be done with a **`FOR`**...**`NEXT`** loop that takes steps of 8, and begins at **`Char0`** and ends at **`Char14`**.

```
DO

  FOR charBase = Char0 TO Char14 STEP 8
    GOSUB Def_Cust_Char
    cursor = 7
    GOSUB Disp_Cust_Char
    PAUSE 200 '- charBase
  NEXT

  PAUSE 1000

LOOP
```

Example Program: EepromPixelWorm.bs2



Free Download! This program is available as a free .bs2 file download from the Smart Sensors and Applications Product Page at www.parallax.com.

EepromPixelWorm.bs2 creates a pixel-worm, crawling through a character.

- √ Examine the EEPROM character definitions and predict how the animation will look when you run the program.
- √ Open and run EepromPixelWorm.bs2.
- √ Compare your expected results to the LCD display.

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - EepromPixelWorm.bs2
' Displays an animated pixel worm within a single LCD character

' {$STAMP BS2}                              ' Target device = BASIC Stamp 2
' {$PBASIC 2.5}                             ' Language      = PBASIC 2.5

' -----[ EEPROM Data ]-----------------------------------------------------

Char0           DATA        %11111,         ' * * * * *
                            %01111,         ' 0 * * * *
                            %11111,         ' * * * * *
                            %11111,         ' * * * * *
                            %11111,         ' * * * * *
                            %11111,         ' * * * * *
                            %11111,         ' * * * * *
                            %11111          ' * * * * *

Char1           DATA        %11111,         ' * * * * *
                            %00111,         ' 0 0 * * *
```

```
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111          ' * * * * *

Char2           DATA    %11111,         ' * * * * *
                        %00011,         ' 0 0 0 * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111          ' * * * * *

Char3           DATA    %11111,         ' * * * * *
                        %00011,         ' 0 0 0 * *
                        %11011,         ' * * 0 * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111          ' * * * * *

Char4           DATA    %11111,         ' * * * * *
                        %00011,         ' 0 0 0 * *
                        %11011,         ' * * 0 * *
                        %11011,         ' * * 0 * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111          ' * * * * *

Char5           DATA    %11111,         ' * * * * *
                        %10011,         ' * 0 0 * *
                        %11011,         ' * * 0 * *
                        %11011,         ' * * 0 * *
                        %11011,         ' * * 0 * *
                        %11111,         ' * * * * *
                        %11111,         ' * * * * *
                        %11111          ' * * * * *

Char6           DATA    %11111,         ' * * * * *
                        %11011,         ' * * 0 * *
                        %11011,         ' * * 0 * *
                        %11011,         ' * * 0 * *
                        %11011,         ' * * 0 * *
                        %11011,         ' * * 0 * *
                        %11111,         ' * * * * *
                        %11111          ' * * * * *
```

```
Char7           DATA        %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11011,          ' * * 0 * *
                            %11011,          ' * * 0 * *
                            %11011,          ' * * 0 * *
                            %11011,          ' * * 0 * *
                            %11011,          ' * * 0 * *
                            %11111           ' * * * * *

Char8           DATA        %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11011,          ' * * 0 * *
                            %11011,          ' * * 0 * *
                            %11011,          ' * * 0 * *
                            %11001,          ' * * 0 0 *
                            %11111           ' * * * * *

Char9           DATA        %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11011,          ' * * 0 * *
                            %11011,          ' * * 0 * *
                            %11000,          ' * * 0 0 0
                            %11111           ' * * * * *

Char10          DATA        %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11011,          ' * * 0 * *
                            %11000,          ' * * 0 0 0
                            %11111           ' * * * * *

Char11          DATA        %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11000,          ' * * 0 0 0
                            %11111           ' * * * * *

Char12          DATA        %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
                            %11111,          ' * * * * *
```

```
                            %11111,           ' * * * * *
                            %11100,           ' * * * 0 0
                            %11111            ' * * * * *

Char13          DATA        %11111,           ' * * * * *
                            %11111,           ' * * * * *
                            %11111,           ' * * * * *
                            %11111,           ' * * * * *
                            %11111,           ' * * * * *
                            %11111,           ' * * * * *
                            %11110,           ' * * * * 0
                            %11111            ' * * * * *

Char14          DATA        %11111,           ' * * * * *
                            %11111,           ' * * * * *
                            %11111,           ' * * * * *
                            %11111,           ' * * * * *
                            %11111,           ' * * * * *
                            %11111,           ' * * * * *
                            %11111,           ' * * * * *
                            %11111            ' * * * * *

' -----[ I/O Pins ]------------------------------------------------------------

LcdPin          PIN     14                    ' I/O pin connected to LCD's RX

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

T9600        CON     84                    ' True, 8-bits, no parity, 9600

LcdCls          CON     12                    ' Form feed -> clear screen
LcdCr           CON     13                    ' Carriage return
LcdOff          CON     21                    ' Turns display off
LcdOn           CON     22                    ' Turns display on
Line0           CON     128                   ' Line 0, character 0
Line1           CON     148                   ' Line 1, character 0
Define          CON     248                   ' Address defines cust char 0

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

custChar        VAR     Nib                   ' Custom charcter selector
index           VAR     Nib                   ' Eeprom index variable
charBase        VAR     Byte                  ' Character base for READ
dotLine         VAR     Byte                  ' 5-pixel dotted line
cursor          VAR     Nib                   ' Cursor placement
counter         VAR     Nib                   ' Main loop counting variable
line            VAR     Byte                  ' Line0 or Line1

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

PAUSE 100                                     ' Debounce power supply
```

```
SEROUT LcdPin, T9600, [LcdOn, LcdCls]        ' Initialize LCD
PAUSE 5                                      ' 5 ms delay for clearing display

custChar = 2                                 ' Select Custom Character 2
line = Line0                                 ' BarGraph on Line 0.

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

DO                                           ' Main loop

  FOR charBase = Char0 TO Char14 STEP 8      ' Go through 10 custom characters
    GOSUB Def_Cust_Char                      ' Define the chracter
    cursor = 7                               ' Place the cursor
    GOSUB Disp_Cust_Char                     ' Print the character
    PAUSE 200 '- charBase                    ' charbase bigger - pause smaller
  NEXT                                       ' Repeat FOR charbase...

  PAUSE 1000                                 ' Pause 1 second

LOOP                                         ' Repeat main loop

' -----[ Subroutine - Def_Cust_Char ]----------------------------------------

' This subroutine defines one of the LCD's eight custom characters.  Set the
' charBase variable equal to one of the Symbol name that precedes the
' custom character's DATA directive.  Set  the custChar variable to a value
' between 0 and 7 to select one of the LCD's eight custom characters.

Def_Cust_Char:
  SEROUT LcdPin, T9600,                      ' Define custom character
         [Define + custChar]
  FOR index = 0 TO 7                         ' 7 bytes, define 7 dotted lines
    READ charBase + index, dotLine           ' Get byte for dotted line
    SEROUT LcdPin, T9600, [dotLine]          ' Send it to the LCD
  NEXT

  RETURN

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

' This subroutine displays a custom character.  The line variable can
' be set to either Line0 or Line1, and the cursor variable can be set
' to a value between 0 and 15.  The custChar variable selects one of the
' LCD's custom characters and should be set to a value between 0 and 7.

Disp_Cust_Char:
  SEROUT LcdPin, T9600,                      ' Print custom character
         [line + cursor, custChar]
  RETURN
```

Inside the Subroutines - Def_Cust_Char and Disp_Cust_Char

Let's take a look at the **Def_Cust_Char** subroutine (below). The first command, **SEROUT LcdPin, T9600, [Define + custChar]** sends a value between 248 and 255 to the LCD. That's because **Define** is set to 248 in the Constants section. 248 is the value that tells the LCD to define Custom Character 0. If you want to define Custom Character 1, it's 249, and so on up to Custom Character 7, which is 255. So the term **Define + custChar** can be 248 if **custChar** stores 0, or 249 if **custChar** stores 1, and so on up to 255 if **custChar** stores 7.

```
Def_Cust_Char:

  SEROUT LcdPin, T9600,                ' Define Custom Character 2
         [Define + custChar]
  FOR index = 0 TO 7                   ' 7 bytes, define 7 dotted lines
    READ charBase + index, dotLine     ' Get byte for dotted line
    SEROUT LcdPin, T9600, [dotLine]    ' Send it to the LCD
  NEXT

  RETURN
```


After the LCD receives a value between 248 and 255, it expects to receive eight more bytes, each containing one of the eight horizontal dotted lines that make up a custom character definition. The **FOR**...**NEXT** loop in **Def_Cust_Char** reads each byte in EEPROM, starting at **charBase**. Keep in mind that the program's Main Routine sets **charBase** to the ***Symbol*** name of one of the **DATA** directives, which is a constant equal to the starting address of the data. The command **READ charBase + index, dotLine** reads a byte of the character definition and stores it in the **dotLine** variable. The first time through the **FOR**...**NEXT** loop, **index** is 0, so the first byte in the character definition is fetched from EEPROM and stored in the **dotLine** variable. Then, the value stored by **dotLine** is sent to the LCD with the command **SEROUT LcdPin, T9600, [dotLine]**. The second time through the **FOR**...**NEXT** loop, **index** is now 1, and so the second byte is read from EEPROM and sent to the LCD. The third time through, the third byte in the character definition is fetched and sent, and so on, up to the eighth byte when **index** gets to 7.

The **Disp_Cust_Char** subroutine has just one command, **SEROUT LcdPin, T9600, [line + cursor, custChar]**. It sends two values to the LCD. The first is **line + cursor**, which places the cursor where the custom character is to be printed. The **line** variable contains either **Line0** or **Line1**. **Line0** is a the constant value 128, which points to character 0 on Line 0. **Line1** is the constant value 148, which points to character 0 on

Line 1. By adding **cursor** (a value between 0 and 15) this gives you control over which line and character the cursor is placed on. The **custChar** variable, which contains a value between 0 and 7 will cause the custom character to be printed where the cursor has been placed. The second value the **SEROUT** command sends is **custChar**, which contains a value between 0 and 7. This value makes the LCD print a custom character where the cursor was placed.

```
Disp_Cust_Char:

  SEROUT LcdPin, T9600,
         [line + cursor, custChar]
  RETURN
```

Your Turn - Multiple Copies of Custom Characters?

If you have lots of different custom characters, but you only want to display one at any given time, you only need one of the LCD's custom character definitions. However, if you want more than one different custom character to be displayed by the LCD at the same time, use more than one of the different custom character definitions in the LCD's character memory.

√ Save CustomEepromCharacters.bs2 as CustomEepromCharactersYourTurn.bs2.
√ Add this code block to the program's Initialization section.

```
custChar = 5
line = Line0
cursor = 9
GOSUB Disp_Cust_Char
line = Line1
GOSUB Disp_Cust_Char
cursor = 8
GOSUB Disp_Cust_Char
```

√ Run the program, and observe the result displayed on the LCD.

The main **DO**...**LOOP** is only updating Line 0, character 8, yet the other three instances of Custom Character 5 are also changing! Why? *When the definition of Custom Character 5 changes, all the Custom Character 5s on the display are automatically updated.*

In some cases, this is a desirable. For example, you can get some interesting visual effects from making 32 copies of one custom character and then repeatedly updating the character definition. For situations where you want to display more than one custom

character at the same time, simply use more than one custom character. In other cases, when you want the LCD to display more than one different custom character to display at the same time, use different custom character definitions.

Custom character display summary:

- If you have lots of different custom characters, but you only want to display one at any given time, use a single custom character and update its definition to change the character.
- If you want more than one different custom character to be displayed by the LCD at the same time, use more than one of the different custom character definitions in the LCD's character memory.

For the second rule, about having more than one custom character on the display at one time, the next activity provides a working example.

ACTIVITY #2: HORIZONTAL BAR GRAPHS FOR PING))) DISTANCE

This first bar graph example demonstrates how to graphically display a Ping))) sensor's measurement of an object's centimeter distance.

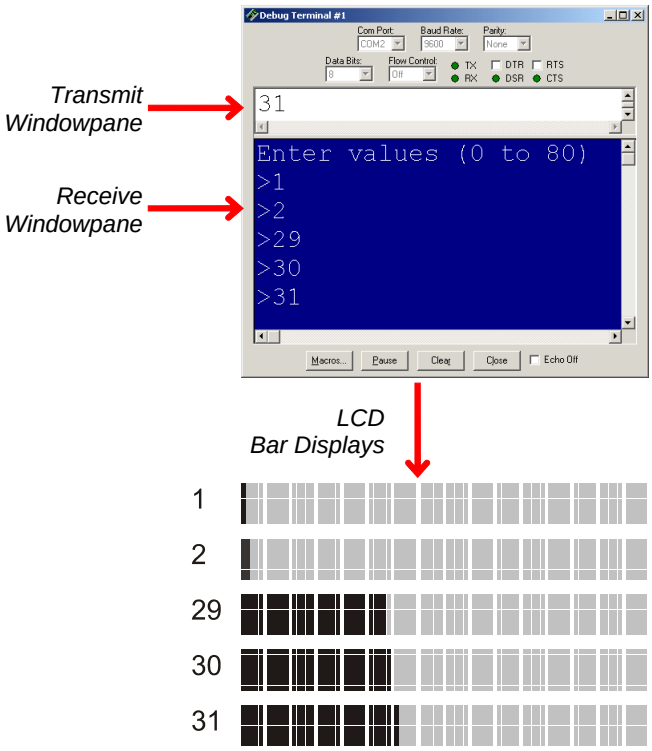
Parts and Equipment

See Chapter #2, Activity #4.

A Horizontal Bar Graph

To test the next example program's horizontal bar graph, you will enter values into the Debug Terminal's Transmit windowpane, and the LCD will display the result in bar graph format as shown in Figure 7-1. Each row on the LCD has eighty columns of vertical dotted lines. If you enter 1, the leftmost column in the leftmost character will go black. Entering 2 will cause two columns to turn black. If you enter 29, it will cause 29 columns to turn black. Since each character is 5 columns wide, the value 29 is actually displayed as five blacked out characters and the sixth character with four black columns.

Figure 7-1: Horizontal Bar Graph Custom Characters



The next example program uses both of the custom character display rules discussed in the previous activity's Your Turn section. Depending on the value that is displayed, the program stores one of five different character definitions in Custom Character 2. However, unlike CustomEepromCharacters.bs2, this next program does not store custom character definitions in the BASIC Stamp EEPROM. Instead, the definitions are derived during program execution, based on the value of the variable being expressed by the bar graph.

Figure 7-2 shows how definitions range from blank (**`Char0`**) to four black vertical columns (**`Char4`**). These definitions will be stored in EEPROM, and used one at a time to redefine character 2, since only one of them is needed at any given time. But the

program may need to display multiple copies of an all-black character for numbers greater than 10; for this task it uses Custom Character 3.

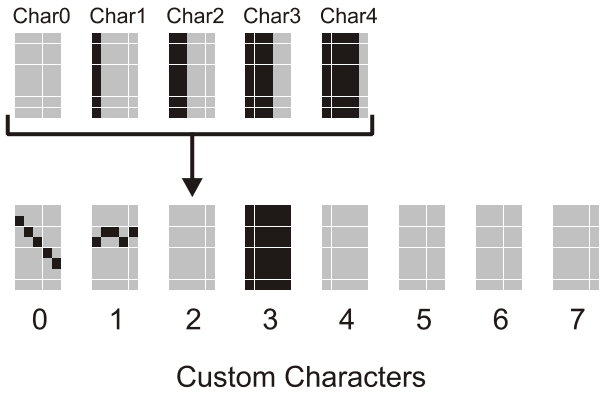


Figure 7-2
Library of Definitions for Custom Character 2

Example Program: HorizBarGraph.bs2

- √ Enter, save, and run HorizBarGraph.bs2.
- √ Click the Debug Terminal's Transmit windowpane.
- √ Type the digit 8, then press the Enter key.
- √ Check the bar graph and make sure the leftmost character is black, and the one next to displays Char3.
- √ Try the values shown in Figure 7-1.
- √ Try 45, 46, 47, 48, 49, 50, 51, 52.
- √ Experiment with limits such as 0, 80.
- √ Try values outside the limits. What happens?

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - HorizBarGraph.bs2
' Display values entered into the Debug Terminal's Receive windowpane
' as horizontal bar graph data in the LCD.

' {$STAMP BS2}                                ' Target device = BASIC Stamp 2
' {$PBASIC 2.5}                               ' Language      = PBASIC 2.5

' -----[ I/O Pins ]--------------------------------------------------------

LcdPin         PIN     14                    ' I/O pin connected to LCD's RX
```

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

T9600          CON     84                   ' True, 8-bits, no parity, 9600

LcdCls         CON     12                   ' Form feed -> clear screen
LcdCr          CON     13                   ' Carriage return
LcdOff         CON     21                   ' Turns display off
LcdOn          CON     22                   ' Turns display on
Line0          CON     128                  ' Line 0, character 0
Line1          CON     148                  ' Line 1, character 0
Define         CON     248                  ' Address defines cust char 0

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

custChar       VAR     Nib                  ' Custom charcter selector
index          VAR     Nib                  ' Eeprom index variable
dotLine        VAR     Byte                 ' 5-pixel dotted line
cursor         VAR     Nib                  ' Cursor placement
value          VAR     Byte                 ' Value to be graphed.
charCnt        VAR     Byte                 ' Character counting variable
line           VAR     Byte                 ' Line0 or Line1

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

PAUSE 100                                   ' Debounce power supply
SEROUT LcdPin, T9600, [LcdOn, LcdCls]       ' Initialize LCD
PAUSE 5                                     ' 5 ms delay for clearing display

custChar = 3                                ' Select Custom Character 3
dotLine = %11111                            ' Black all pixels in each line
GOSUB Def_Horiz_Bar_Char                    ' Character define subroutine

line = Line0                                ' BarGraph on Line 0.

DEBUG "Enter values (0 to 80)", CR

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

DO                                          ' Main loop

  DEBUG ">"
  DEBUGIN DEC value                         ' Value from Transmit windowpane
  GOSUB Bar_Graph                           ' Display as bar graph

LOOP                                        ' Repeat main loop

' -----[ Subroutine - Bar_Graph ]-------------------------------------------

Bar_Graph:

  ' Fill from left with black bars
```

```
  value = value MAX 80                      ' Limit value - 0 to 80
  charCnt = value / 5                       ' Number of black bars
  custChar = 3                              ' Choose black custom character

  IF charCnt > 0 THEN                       ' If black bars to print then
    FOR cursor = 0 TO charCnt - 1           ' Print charCnt - 1 black bars
      GOSUB Disp_Cust_Char                  ' Print the black bar
    NEXT
  ENDIF

  ' Display Custom Character 2 with a certain number of black columns.

  cursor = charCnt                          ' Place cursor
  custChar = value // 5                     ' How many 5ths of a bar?
  ' Choose bit pattern for custom character definition
  LOOKUP custChar,
         [%00000, %10000, %11000, %11100, %11110],
         dotLine
  custChar = 2                              ' Set custom character to 2
  GOSUB Def_Horiz_Bar_Char                  ' Define the custom character
  GOSUB Disp_Cust_Char                      ' Display the custom character

  ' Print over everything to the right with spaces.

  IF (charCnt + 1) < 15 THEN                ' Partial char left of char 15?
    FOR cursor = (charCnt + 1) TO 15        ' Fill to right with " "
      SEROUT LcdPin, T9600,
             [line + cursor, " "]
    NEXT
  ELSEIF value = 80 THEN                    ' Special case: value = 80
    SEROUT LcdPin, T9600,
           [line + cursor, 3]
  ELSEIF charCnt = 14 THEN                  ' Special case: 75 <= value <= 80
    SEROUT LcdPin, T9600, [line + 15, " "]
  ENDIF

  RETURN

' -----[ Subroutine - Def_Horiz_Bar_Char ]-----------------------------------

Def_Horiz_Bar_Char:

  SEROUT LcdPin, T9600,                     ' Define custom character
         [Define + custChar]
  FOR index = 0 TO 7                        ' 7 bytes, define 7 dotted lines
    SEROUT LcdPin, T9600, [dotLine]         ' Send it to the LCD
  NEXT

  RETURN
```

```
' -----[ Subroutines - Disp_Cust_Char ]-------------------------------------

' This subroutine displays a custom character.  The line variable can
' be set to either Line0 or Line1, and the cursor variable can be set
' to a value between 0 and 15.  The custChar variable selects one of the
' LCD's custom characters and should be set to a value between 0 and 7.

Disp_Cust_Char:

  SEROUT LcdPin, T9600,                    ' Print custom character
         [line + cursor, custChar]
  RETURN
```

How HorizBarGraph.bs2 Works

The LCD's Custom Character 2 is reserved for displaying one of five different custom character definitions, but Custom Character 3 will always use the same definition, to have all its pixels black. With this in mind, a code block was added to the Initialization section that defines the LCD's Custom Character 3 using the **`Def_Horiz_Bar_Char`** subroutine. This subroutine makes 8 identical copies of the 5-pixel row stored by the **`dotLine`** variable to build the custom character. After that, the line variable is set to place the cursor on the top row, and the program displays a user prompt to "Enter values (0 to 80)", followed by a carriage return.

```
custChar = 3                          ' Select Custom Character 3
dotLine = %11111                      ' Black all pixels in each line
GOSUB Def_Horiz_Bar_Char              ' Character define subroutine

line = Line0                          ' BarGraph on Line 0.

DEBUG "Enter values (0 to 80)", CR
```

The Main Routine is a **`DO`**...**`LOOP`** that repeatedly displays the **`">"`** prompt, and then gets decimal values entered into the Debug Terminal's Transmit windowpane. Then it calls the **`Bar_Graph`** subroutine.

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

DO                                    ' Main loop

  DEBUG ">"
  DEBUGIN DEC value                   ' Value from Transmit windowpane
  GOSUB Bar_Graph                     ' Display as bar graph

LOOP                                  ' Repeat main loop
```

The **Bar_Graph** subroutine takes whatever is stored in the **value** variable and represents it on the LCD with a bar graph display. This subroutine relies on both the **Def_Cust_Char** and **Disp_Cust_Char** subroutines that were introduced in the previous activity. The **Bar_Graph** subroutine consists of three major steps:

1. Fill any black characters from left to right. For example, if the **value** variable is set to 28, five LCD characters (with 5 vertical black lines each) have to be blackened.
2. Continuing with the example, the sixth LCD character will have three vertical lines. Remember, Custom Character 2 is used to display one of five character definitions shown in Figure 7-2. The number of black columns in the character is the remainder is the remainder of **value // 5**. This result selects a bit pattern from a **LOOKUP** table and copies it to the **dotLine** variable. Then **custChar** is set to 2 and the **Def_Horiz_Bar_Char** subroutine copies this bit pattern to all 8 rows in the character. After the character is redefined, it can then be printed.
3. All characters not needed to represent the value on the bar graph have to be cleared with the space " " character. In this **value = 28** example, that means clearing everything between the right of the sixth character and the 15th character. While this only absolutely necessary the previous **value** is smaller than the current **value**, the program does it every time through the loop.

The first step in the **Bar_Graph** subroutine is to fill black characters. The **value** variable is first clamped to 80 or less. Next, the **charCnt** variable stores the number of black characters that have to be printed, which is 1/5 of the **value** variable. The **custChar** variable has to be set equal to three, since Custom Character 3 stores the black character. If **charCnt** is larger than 0, it means there will be some all-black characters that have to be printed, and a **FOR**...**NEXT** loop repeatedly calls the **Disp_Cust_Char** subroutine. Remember, this subroutine depends on two variables: **cursor** and **custChar**. The value of **charCnt** was set before the **FOR**...**NEXT** loop, and the cursor variable is the **FOR**...**NEXT** loop's **index** variable. Each time through the **FOR**...**NEXT** loop, the **cursor** variable increases by 1, which causes the **Disp_Cust_Char** subroutine to place the cursor one notch to the right each time it is called, thereby filling black characters from left to right.

```
' -----[ Subroutine - Bar_Graph ]----------------------------------

Bar_Graph:

  ' Fill from left with black bars

  value = value MAX 80              ' Limit value - 0 to 80
```

```
charCnt = value / 5              ' Number of black bars
custChar = 3                     ' Choose black custom character

IF charCnt > 0 THEN              ' If black bars to print then
  FOR cursor = 0 TO charCnt - 1  ' Print charCnt - 1 black bars
    GOSUB Disp_Cust_Char         ' Print the black bar
  NEXT
ENDIF
```

The second step is to display the one character that is partially black. The command **`cursor = charCnt`** makes sure that the cursor is now just to the right of the black characters that were printed with a **`FOR`**...**`NEXT`** loop in the previous step. Next, **`custChar = value // 5`** sets the **`custChar`** variable to the remainder of value ÷ 5. For example, if value is 28, the remainder of 28 ÷ 5 is 3. If **`custChar = 3`**, a lookup table stores **`%11100`** in the **`dotLine`** variable. The **`Def_Horiz_Bar_Char`** subroutine needs to know two things to do its job, the **`dotLine`**, and the **`custChar`**. We are using and re-using Custom Character 2 for defining and redefining the partially blackened character. So, before calling **`Def_Horiz_Bar_Char`**, **`custChar`** needs to be changed from 3 to 2 with the command **`custChar = 2`**. Then, **`Def_Horiz_Bar_Char`** can be called to define the custom character, followed by **`Disp_Cust_Char`** to display it.

```
' Display Custom Character 2 with a certain number of black columns.

cursor = charCnt                 ' Place cursor
custChar = value // 5            ' How many 5ths of a bar?
' Choose bit pattern for custom character definition
LOOKUP custChar,
       [%00000, %10000, %11000, %11100, %11110],
       dotLine
custChar = 2                     ' Set custom character to 2
GOSUB Def_Horiz_Bar_Char         ' Define the custom character
GOSUB Disp_Cust_Char             ' Display the custom character
```

The **`Def_Horiz_Bar_Char`** subroutine that gets called after the value of **`dotLine`** and **`custChar`** variables are set is what makes storing custom characters in EEPROM unnecessary. Reason being, *if you want to create a custom character with several columns of black pixels, all you have to do is send the LCD the same binary value, eight times in a row*. The **`dotLine`** variable is the one that stores the binary definition for the rows in the partially filled custom character. If **`dotLine`** is %11000, the left two columns of pixels become black. If **`dotLine`** is %11100, the left three columns of pixels become black, and so on.

```
Def_Horiz_Bar_Char:

  SEROUT LcdPin, T9600,              ' Define custom character
         [Define + custChar]
  FOR index = 0 TO 7                   ' 7 bytes, define 7 dotted lines
    SEROUT LcdPin, T9600, [dotLine] ' Send it to the LCD
  NEXT

  RETURN
```

Overprinting any black characters to the right of the character displayed in step 2 does not involve any custom characters since the space character " " does a good job of erasing things. For most cases, a **FOR**...**NEXT** loop printing from **(charCnt + 1) TO 15** clears everything to the right. However, there are some special circumstances that occur when the 15th character has one or more black columns. If **value = 80**, an earlier part of the program will print a blank character in position-0. Position-0 should be black, so the **ELSEIF value = 80** code block replaces that blank character with a black one. Also, if **charCnt** is 14, a single empty character has to be printed in position 15.

```
' Print over everything to the right with spaces.

IF charCnt + 1 < 15 THEN            ' Partial char left of char 15?
  FOR cursor = (charCnt + 1) TO 15  ' Fill to right with " "
    SEROUT LcdPin, T9600,
           [Line0 + cursor, " "]
  NEXT
ELSEIF value = 80 THEN              ' Special case: value = 80
  SEROUT LcdPin, T9600,
         [Line0 + cursor, 3]
ELSEIF charCnt = 14 THEN            ' Special case: 75 <= value <= 80
  SEROUT LcdPin, T9600, [Line0 + 15, " "]
ENDIF

RETURN
```

Your Turn - Graphically Display Ping))) Sensor Distance

Displaying a horizontal bar graph that indicates up to 80 cm is easy with the Ping))) sensor and the Parallax Serial LCD. The trick is to take components from PingMeasureCm.bs2, and incorporate them into a copy of this activity's HorizBarGraph.bs2.

√ Follow the instructions for connecting the Ping))) sensor and the Serial LCD to your board. They're in Chapter 2, Activity #4, page 51.

- √ Open PingMeasureCm.bs2 from Chapter 2, Activity #2 (page 48) into the BASIC Stamp Editor.
- √ Open HorizBarGraph.bs2 (this activity), and save it as PingBarGraph.bs2.
- √ Shade and copy the **CON** and **VAR** directives from PingDistance.bs2, and paste them into the **CON** and **VAR** sections in PingBarGraph.bs2.
- √ Replace the **DEBUG** and **DEBUGIN** commands in PingBarGraph.bs2's Main Routine **DO**...**LOOP** the commands in PingMeasureCm.bs2's **DO**...**LOOP**.
- √ Add a command just before **GOSUB Bar_Graph** that sets the **value** variable equal to **cmDistance**.
- √ Then move the **PAUSE 100** command so that it comes just before **LOOP**.

Now, the Main Routine in PingBarGraph.bs2 should look like this:

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

DO                                      ' Main loop

  PULSOUT 15, 5
  PULSIN 15, 1, time

  cmDistance = CmConstant ** time

  DEBUG HOME, DEC3 cmDistance, " cm"

  value = cmDistance

  GOSUB Bar_Graph                       ' Display as bar graph

  PAUSE 100

LOOP                                    ' Repeat main loop
```

- √ Save the modified program, test it, and verify that it works.

You can also replace the **DEBUG** command with a **SEROUT** command that displays the measurements on the bottom row. Remember that you will have to send the LCD a control code to place the cursor on Line 1 character-0, instead of using **HOME**.

- √ Try it!

ACTIVITY #3: TWO-AXIS BAR GRAPH FOR ACCELEROMETER TILT

This activity develops a two axis bar graph that is useful for graphically displaying the accelerometer's tilt measurements.

Parts and Equipment

Use the circuit from Chapter 3, Activity #2, page 71.

Vertical Bar Graph Custom Characters

The **`Def_Vert_Bar_Char`** subroutine in the next example program defines any one of 16 different custom vertical bar characters. Figure 7-3 shows how each bar character corresponds to a value.

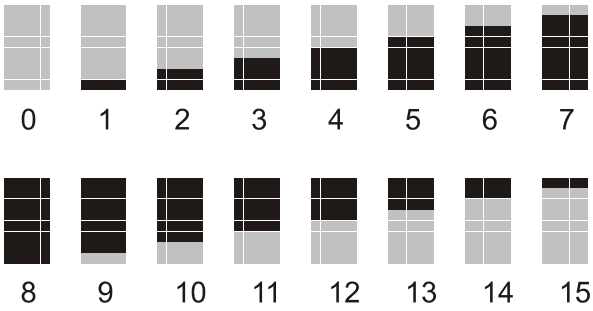


Figure 7-3
Vertical Character Bar Graph Values

The custom character definitions in Figure 7-3 follow a sequence that relates directly to the values they represent, so the definitions can be calculated as needed rather than being stored in EEPROM. Since each custom character definition takes up eight bytes, eliminating 16 definitions will save 128 bytes of program memory.

The **`Def_Vert_Bar_Char`** subroutine defines the characters shown in Figure 7-3. The **`value`** variable defines which one of the 0 to 15 characters will be displayed. Each bit in the **`rowMap`** variable determines whether one of the rows in the custom character is black or white.

```
Def_Vert_Bar_Char:

  SEROUT LcdPin, T9600,
         [Define + custChar]

  rowMap = %1111111100000000 >> (value & %1111)
```

```
FOR index = 0 TO 7
  IF rowMap.LOWBIT(index) = 1 THEN
    SEROUT LcdPin, T9600, [%11111]
  ELSE
    SEROUT LcdPin, T9600, [%00000]
  ENDIF
NEXT

RETURN
```

The command **`rowMap = %1111111100000000 >> (value & %1111)`** shifts the eight 1 bits in %1111111100000000 right by the **`value`** variable. The term **`(value & %1111)`** is called a bit mask, and it makes it possible to use values from 16 to 31 to give you the same results as 0 to 15. If **`value`** is 3, the command **`rowMap = %1111111100000000 >> (value & %1111)`** stores %0001111111100000 in the **`rowMap`** variable. Since **`rowMap`** is a only byte variable, it only stores the lowest eight bits of the term, which is %11100000. Compare that to the custom character for 3 in Figure 7-3. If **`value`** is 4, the low byte result of the **`rowMap`** variable is %11110000. Now, take a look at character 4 in Figure 7-3. Try it for each value (0 to 15) and you'll see the pattern that the bits in the **`rowMap`** byte match the pattern of the rows with black pixels in Figure 7-3.

Defining a custom character involves sending eight bytes. The binary values in each successive byte define each of the eight rows in the character, from top to bottom. The **`FOR`**...**`NEXT`** loop in the **`Def_Vert_Bar_Char`** subroutine has an **`IF`**...**`THEN`** code block that uses the **`.LOWBIT`** operator to check each bit in the **`rowMap`** variable and use it to either define a row of 5 black or white pixels in the custom character. Let's say that value is 3, so **`rowMap`** is %11100000. The first time through the **`FOR`**...**`NEXT`** loop, **`index`** is 0, so the **`IF`**...**`THEN`** statement examines **`rowMap.LOWBIT(0`**), the rightmost bit. Since it's 0, the **`IF`**...**`THEN`** statement sends a byte containing %00000 to the LCD, which makes all the pixels white. By the sixth time through the **`FOR`**...**`NEXT`** loop, **`rowMap.LOWBIT(5)`** = 1, so the **`IF`**...**`THEN`** statement sends a **`SEROUT`** command with a byte containing %11111. So, the sixth, seventh, and eighth rows down will be black, which results in the character shown in Figure 7-3.



`SEROUT 14, 84, [%11111]` actually sends the byte %00011111, and the lower five bits are the ones the LCD uses to define its five-pixel-wide rows.

Example Program: TestVerticalBars.bs2

This example program displays the 16 different vertical bar custom characters on the Parallax Serial LCD's Line 0, character-8. It does so in rapid succession, over and over again. It also displays the number that the **`value`** variable stores in the Debug Terminal.

- √ Open TestVerticalBars.bs2 in the BASIC Stamp Editor.
- √ Run it and verify that all the custom characters in Figure 7-3 are displayed.
- √ Increase the **`PAUSE`** command's ***`Duration`*** argument.
- √ Re-run the program and verify that the LCD bars correctly represent the numbers displayed in the Debug Terminal.


```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - TestVerticalBars.bs2
' Displays sixteen different vertical bar characters on Line 0, character-8
' of the Parallax Serial LCD.

' {$STAMP BS2}                              ' Target device = BASIC Stamp 2
' {$PBASIC 2.5}                             ' Language      = PBASIC 2.5

' -----[ I/O Pins ]--------------------------------------------------------

LcdPin          PIN     14                  ' I/O pin connected to LCD's RX

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

T9600           CON     84                  ' True, 8-bits, no parity, 9600

LcdCls          CON     12                  ' Form feed -> clear screen
LcdOn           CON     22                  ' Turns display on
Line0           CON     128                 ' Line 0, character 0
Line1           CON     148                 ' Line 1, character 0
Define          CON     248                 ' Address defines cust char 0

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

custChar        VAR     Byte                ' Custom character selector
index           VAR     Nib                 ' Eeprom index variable
rowMap          VAR     Byte                ' 5-pixel dotted line
cursor          VAR     Byte                ' Cursor placement
value           VAR     Byte                ' Value to be graphed
line            VAR     Byte                ' Line0 or Line1

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

' LCD initializiation.
PAUSE 100                                   ' Debounce power supply
```

```
SEROUT LcdPin, T9600, [LcdOn, LcdCls]      ' Initialize LCD
PAUSE 5                                     ' 5 ms delay for clearing display

' Custom character subroutine values.
custChar = 3                                ' Use Custom Character 3
line = Line0                                ' Cursor to Line0
cursor = 7                                  ' Cursor to 8th character

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

DO                                          ' Main loop

  FOR value = 0 TO 16                       ' value counts 0 to 16
    DEBUG ? value                           ' Display value in Debug Terminal
    GOSUB Def_Vert_Bar_Char                 ' Define bar graph character
    GOSUB Disp_Cust_Char                    ' Display character on LCD
    PAUSE 50                                ' Slow down the loop
  NEXT

  DEBUG CR, CR                              ' Spaces before next sequence
  PAUSE 500                                 ' Delay before next sequence

LOOP                                        ' Repeat main loop

' -----[ Subroutine - Def_Vert_Bar_Char ]-----------------------------------

' Defines a vertical bar graph character based on the value variable
' (0 to 15) and the custChar variable, which selects the Parallax Serial
' LCD's custom characters between 0 and 7.

Def_Vert_Bar_Char:
  ' Start define custom character
  SEROUT LcdPin, T9600,
         [Define + custChar]

  ' Select a row map for the custom character based on value.
  rowMap = %1111111100000000 >> (value & %1111)

  ' Send 8 bytes, each defining a dotted row in the custom character.  Each
  ' row is determined by examining a bit in the rowMap variable, and then
  ' sending %11111 if the bit is 1, or %00000 if the bit is 0.
  FOR index = 0 TO 7                        ' Repeat 7 times, index counts
    IF rowMap.LOWBIT(index) = 1 THEN        ' Examine next bit in rowMap
      SEROUT LcdPin, T9600, [%11111]        ' If 1, send %11111
    ELSE
      SEROUT LcdPin, T9600, [%00000]        ' Otherwise, send %00000
    ENDIF
  NEXT
                                            ' Return from subroutine.
  RETURN
```

```
' -----[ Subroutines - Disp_Cust_Char ]-------------------------------------

' This subroutine displays a custom character.  The line variable can
' be set to either Line0 or Line1, and the cursor variable can be set
' to a value between 0 and 15.  The custChar variable selects one of the
' LCD's custom characters and should be set to a value between 0 and 7.

Disp_Cust_Char:
  SEROUT LcdPin, T9600,                    ' Print custom character
         [line + cursor, custChar]
  RETURN
```

Your Turn - 31 Levels Covering Two Rows and Adding a Horizontal Axis

The command that sets the bit pattern in the **rowMap** variable was **rowMap = %1111111100000000 >> (value & %1111)**. The calculation **value & %1111** will result in 0 if **value** is 16, in 1 if **value** is 17, and so on up to 15 if **value** is 31. It will continue this pattern regardless of how large **value** becomes. Since the bar graph will behave the same way for values between 16 and 31 as it does for values between 0 and 15, the bar graph can be placed either on the upper or lower line to indicate which range the **value** variable falls between. The actual display range is from 1 to 31, with 16 showing no bars.

Modifying the program to display characters in this fashion involves just a few small changes to the Main Routine. First, the **FOR**...**NEXT** loop ***StartValue*** and ***EndValue*** arguments have to be changed from (0 to 15) to (1 to 31). Also, a **SEROUT** command has to be added to overwrite the previous custom characters with blank spaces. Then, an **IF**...**THEN**...**ELSE** code block can be added that positions the cursor on either Line 0 or Line 1. It must place the cursor on Line 0 when value is greater than 16, or on the lower line when value is less than or equal to 16.

- √ Save TestVerticalBars.bs2 as TestVerticalBarsYourTurn.bs2.
- √ Modify the **DO**...**LOOP** in the Main Routine as shown below.
- √ Run the program and verify that the bar graph now displays 31 different levels, from 1 to 31.

```
DO

  FOR value = 1 TO 31
    SEROUT 14, 84, [Line0 + cursor, " ",
                    Line1 + cursor, " "]
    IF value <= 16 THEN
```

```
        line = Line1
      ELSE
        line = Line0
      ENDIF
      DEBUG ? value
      GOSUB Def_Vert_Bar_Char
      GOSUB Disp_Cust_Char
      PAUSE 50
    NEXT

    DEBUG CR, CR
    PAUSE 500

  LOOP
```

You can also nest the **`FOR value = 1 TO 31`**...**`NEXT`** loop inside a **`FOR cursor = 0 TO 15`**...**`NEXT`** loop, and cause the bar graph to move across the display each time it repeats. By controlling the cursor offset like this, the bar graph can display two axes with vertical values from 1 to 31 and horizontal values from 0 to 15.

√ Try it!

A Two-Axis Display

While the previous Your Turn section demonstrated displaying characters on two axes, a completely blank display doesn't really communicate that the value is in the middle of its range. A better way to get the message across is by causing two custom characters to appear next to each other. Figure 7-4 shows how it works. The first two display examples are not in the middle of either the horizontal or vertical ranges, so single characters are displayed. The third example is in the middle of the horizontal range (8), so the vertical measurement is displayed on two adjacent characters. The fourth example shows that the measurement is in the middle of its vertical range, so rows in both the top and bottom character have black pixels. When the measurement is centered in both the horizontal and vertical the fifth example shows how it should look. These features are especially important for graphically indicating when one or both of the two accelerometer axes are level.

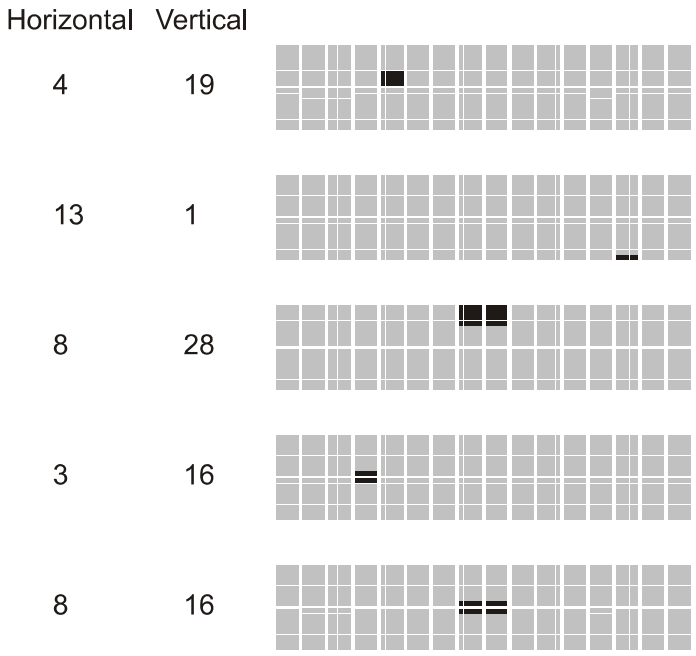


Figure 7-4
Points Plotted on the Two-Axis Bar Graph

While the features shown in Figure 7-4 make the display a little less cryptic, the program has to make a lot more decisions. In the next example program, the **`Bar_Graph_H_V`** and **`Horizontal_Placement`** subroutines take care of all the decisions that support these extra features.

Example Program: TwoAxisBarDisplay.bs2

Free Download! This program is available as a free .bs2 file download from the Smart Sensors and Applications Product Page at www.parallax.com.

You can test this example program by entering horizontal and vertical coordinates into the Debug Terminal's Transmit windowpane.

- √ Open and run TwoAxisBarDisplay.bs2 with the BASIC Stamp Editor.
- √ Enter values in the 0 to 16 range for horizontal and values in the 1 to 31 range fore vertical, and observe the displayed results by the LCD.

- √ Try the values shown in Figure 7-4 and verify that they match the displays.
- √ Make sure to test horizontal values of 0, 8, and 16 for various vertical values.
- √ Also test vertical values of 1, 16, and 31 for various horizontal values.

Another way of testing the display is with the alternate Main Routine shown below. It has nested **`FOR`**...**`NEXT`** loops that test every **`value`** variable across all the possible **`cursor`** variable values.

- √ Save a copy of TwoAxisBarDisplay.bs2 as TwoAxisBarDisplayTest.bs2.
- √ Run the modified program and watch to make sure all the values are displayed correctly.

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

DO
  FOR cursor = 0 TO 16
    FOR value = 1 TO 31
      GOSUB Bar_Graph_H_V
      PAUSE 200
    NEXT
    SEROUT 14, 84, [LcdCls]
    PAUSE 500
  NEXT
LOOP
```

```
' -----[ Title ]-------------------------------------------------------------
' Smart Sensors and Applications - TwoAxisBarDisplay.bs2
' Displays a character that shifts both vertically and horizontally.

' {$STAMP BS2}                              ' Target device = BASIC Stamp 2
' {$PBASIC 2.5}                             ' Language      = PBASIC 2.5

' -----[ I/O Pins ]-----------------------------------------------------------

LcdPin         PIN     14                   ' I/O pin connected to LCD's RX

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

T9600        CON     84                   ' True, 8-bits, no parity, 9600

LcdCls         CON     12                   ' Form feed -> clear screen
LcdCr          CON     13                   ' Carriage return
LcdOff         CON     21                   ' Turns display off
LcdOn          CON     22                   ' Turns display on
Line0          CON     128                  ' Line 0, character 0
Line1          CON     148                  ' Line 1, character 0
Define         CON     248                  ' Address defines cust char 0
```

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

custChar       VAR     Byte                ' Custom charcter selector
index          VAR     Nib                 ' Eeprom index variable
rowMap         VAR     Byte                ' 5-pixel dotted line
cursor         VAR     Byte                ' Cursor placement
value          VAR     Byte                ' Value to be graphed
line           VAR     Byte                ' Line0 or Line1

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

PAUSE 100                                  ' Debounce power supply
SEROUT LcdPin, T9600, [LcdOn, LcdCls]      ' Initialize LCD
PAUSE 5                                    ' 5 ms delay for clearing display

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

DO                                         ' Main loop

  DEBUG "Enter horizontal",                ' Prompt for character offset
        CR, "value (0 TO 16)"
  DEBUGIN DEC cursor                       ' Get character offset
  DEBUG "Enter vertical",
        CR, "value (1 TO 31)"              ' Prompt user for value
  DEBUGIN DEC value                        ' Get value
  GOSUB Bar_Graph_H_V                      ' Call Bar_Graph_H_V

LOOP                                       ' Repeat main loop

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

' Defines and displays two axis bar graph characters based on the value of
' the cursor (0 to 16) and value (1 to 31).  Calls Def_Vert_Bar_Char, and
' Horizontal_Placement.

Bar_Graph_H_V:

  SEROUT 14, 84, [LcdCls]                  ' Clear previous plot
  PAUSE 5                                  ' 5 ms delay for clearing display

  ' Decide whether to display on Line 0 or Line 1.
  IF value >= 16 THEN line = Line0 ELSE Line = Line1

  GOSUB Def_Vert_Bar_Char                  ' Define custom character
  GOSUB Horizontal_Placement

  IF value = 16 THEN                       ' Special case: value = 16

    value = 1                              ' Line 0 display
    custChar = 2
```

```
    GOSUB Def_Vert_Bar_Char
    line = Line0
    GOSUB Horizontal_Placement

    value = 15                                ' Line 1 Display
    custChar = 3
    GOSUB Def_Vert_Bar_Char
    line = Line1
    GOSUB Horizontal_Placement

    value = 16                                ' Restore value

  ENDIF

  RETURN

' -----[ Subroutine - Def_Vert_Bar_Char ]-------------------------------------

' Defines a vertical bar graph character based on the value variable
' (0 to 15) and the custChar variable, which selects the Parallax Serial
' LCD's custom characters between 0 and 7.

Def_Vert_Bar_Char:

  ' Start define custom character
  SEROUT LcdPin, T9600,
         [Define + custChar]

  ' Select a row map for the custom character based on value.
  rowMap = %1111111100000000 >> (value & %1111)

  ' Send 8 bytes, each defining a dotted row in the custom character.  Each
  ' row is determined by examining a bit in the rowMap variable, and then
  ' sending %11111 if the bit is 1, or %00000 if the bit is 0.
  FOR index = 0 TO 7                          ' Repeat 7 times, index counts
    IF rowMap.LOWBIT(index) = 1 THEN          ' Examine next bit in rowMap
      SEROUT 14, 84, [%11111]                 ' If 1, send %11111
    ELSE
      SEROUT 14, 84, [%00000]                 ' Otherwise, send %00000
    ENDIF
  NEXT
                                              ' Return from subroutine.
  RETURN

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

' Places the vertical bar graph at one of the Parallax 2X16 LCD's sixteen
' vertical columns.  The cursor variable can set the horizontal location to
' values between 0 and 16, with 8 the center.  Calls Disp_Custom_Char.

Horizontal_Placement:
```

```
  SELECT cursor                                 ' Cursor 0 to 7, no changes
    CASE 0 TO 7
      GOSUB Disp_Cust_Char
    CASE 8                                      ' Cursor 8, display at 7 & 8
      cursor = 7
      GOSUB Disp_Cust_Char
      cursor = 8
      GOSUB Disp_Cust_Char
    CASE 9 TO 16                                ' Cursor 9 to 16, display 1 left
      cursor = cursor - 1
      GOSUB Disp_Cust_Char
      cursor = cursor + 1
    ENDSELECT

  RETURN

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

' This subroutine displays a custom character.  The line variable can
' be set to either Line0 or Line1, and the cursor variable can be set
' to a value between 0 and 15.  The custChar variable selects one of the
' LCD's custom characters and should be set to a value between 0 and 7.

Disp_Cust_Char:

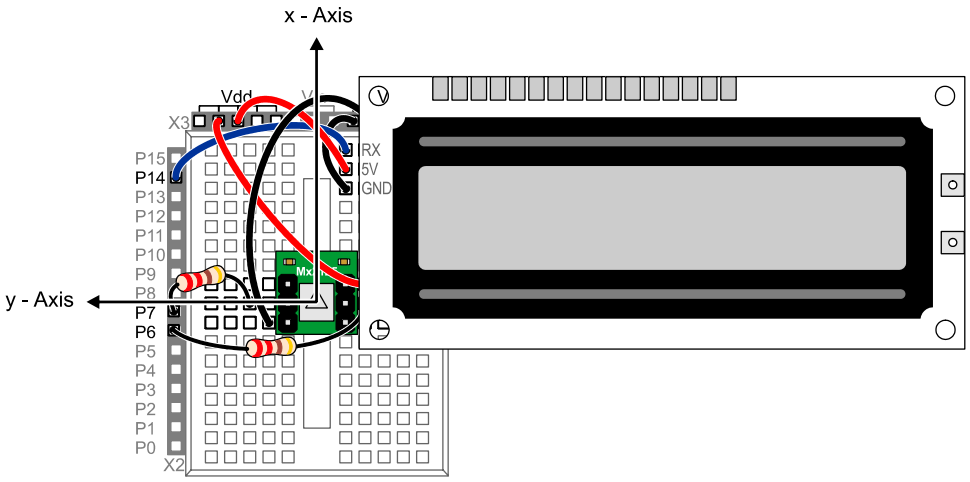
  SEROUT LcdPin, T9600,                     ' Print custom character
         [line + cursor, custChar]
  RETURN
```


Your Turn - Graphical Two-Axis Tilt Display

Indicating tilt with the TwoAxisBarDisplay.bs2 involves **`PULSIN`** commands to acquire the accelerometer's x and y axis measurements. It also requires scaling and offset to fit the accelerometer measurements into a vertical scale of 31 and a horizontal scale of 17.

The horizontal scale also has to be reversed. Figure 7-5 shows how the accelerometer's x and y axes relate to the LCD's horizontal and vertical axes. Note that that the direction of the positive y-axis points away from the direction that values increase on the LCD's horizontal axis. Whenever the scaled y-axis value is 16, the display should show 0, and whenever the scaled y-axis value is 0, the display should show 16. The way to correct this is to use a command that subtracts the scaled y-axis value from 16. Essentially, this boils down to **`cursor`** = 16 − (scaled y value).

Figure 7-5: Accelerometer X and Y Axes



This simplest way to convert TwoAxisBarDisplay.bs2 to a dual-axis accelerometer leveling tool is to borrow code blocks from Chapter 3 and adjust them as needed. The **x** and **y** axis variables, **PULSIN** commands, and the scale and offset commands from HorizontalTilt.bs2 in Chapter 3, Activity #6 provide a starting point. The **PULSIN** and scaling commands can replace the **DEBUG** and **DEBUGIN** commands in TwoAxisBarDisplay.bs2's Main Routine. The **Bar_Graph_H_V** subroutine uses the **value** and **cursor** variables to plot on the LCD. Before it can be called, **value** needs to be set equal to the scaled x-axis value, and **cursor** needs to be set equal to the scaled y-axis value, subtracted from 16.



Deriving the ** scale constants are left as exercises at the end of the chapter.

√ Save TwoAxisBarDisplay.bs2 as TwoAxisBarTiltDisplay.bs2.
√ Add these two declarations to the program's Variables section.

```
X       VAR     Word                    ' Accelerometer x-axis
y       VAR     Word                    ' Accelerometer y-axis
```

√ Replace the outermost **DO**...**LOOP** in the Main Routine with the one below:

```
DO                                              ' Main loop

  PULSIN 6, 1, x
  PULSIN 7, 1, y

  value = (x MIN 1875 MAX 3125) - 1875 ** 1625 + 1
  cursor = 16 - ((y MIN 1875 MAX 3125) - 1875 ** 891)

  GOSUB Bar_Graph_H_V                           ' Call Bar_Graph_H_V

LOOP                                            ' Repeat main loop
```

√ Save and run the program, and test under a variety of tilt circumstances.

SUMMARY

Both distance and tilt measurements lend themselves to bar graph displays. Most bar graphs can be done using only one or two of the Parallax Serial LCD's eight custom character memory slots. The custom character definitions for bar graphs can either be stored in the BASIC Stamp EEPROM, or defined by code that relates the value being graphed to the number of pixel lines or columns that need to be displayed.

Storing custom characters in EEPROM involves **DATA** directives that contain eight bytes each. The lower five bits of each byte store the bit pattern for each of the eight (five-pixel-wide) lines that make up a character. ***Symbol*** names preceding **DATA** directives make it possible to copy the starting address of the bytes in a given **DATA** directive into a variable. A subroutine can use the variable that stores the start address for **READ** operations that sequentially copy bytes to a variable, then send the contents of that variable to the LCD.

Displaying bar graphs is typically more memory-efficient if the characters can be generated based on the value. When the bar graph is horizontal, a **LOOKUP** command is useful for storing binary values that correspond to the number of columns that will have black pixels in a given custom character. For vertical graphs, bits can be shifted into a variable. In either case, subroutines can examine the bits in a variable, then darken pixels in a certain number of rows or columns in a character.

Questions

1. What part of **DATA** directives makes it possible to find the starting address of a character definition?
2. How do programs in this chapter tell the **Def_Cust_Char** subroutine which character to define and where to find the character definition? How do you specify a character location for the **Disp_Cust_Char** subroutine? If you want to display two different custom characters at the same time, can you use the same custom character definition? Why?
3. How does HorizBarGraph.bs2 store information about which columns in a custom character should have black or white pixels?
4. Why can you use just one byte instead of eight to define a custom character with certain columns black?
5. What does the instruction **IF rowMap.LOWBIT(index) = 1 THEN**... do in TestVerticalBars.bs2?

6. Why can you use just one byte instead of eight to define a custom character with certain columns black?
7. What does the instruction **`IF rowMap.LOWBIT(index) = 1 THEN`**... do in the program TestVerticalBars.bs2?

Exercises

1. Write a custom character **`DATA`** directive for a character with all horizontal stripes.
2. Derive the value 1625 in the command **`value = (x MIN 1875 MAX 3125) - 1875 ** 1625 + 1`**.

Projects

1. Modify PingBarGraph.bs2 so that it displays the centimeter distance measurement on the bottom line.
2. Modify TwoAxisBarTiltDisplay.bs2 so that it displays bars that correspond to tilt angle. Remember that tilt angle is a function of the gravity sensed by each of the accelerometer's axes. Hint: Use example code from Chapter 3, Activity #6 for tilt angle calculations.

Solutions

Q1. The ***Symbol*** name.

Q2. The programs have to set two variables before making the **Def_Cust_Char** subroutine call: **custChar** and **charBase**. The **custChar** variable should store the value the program will use to tell the LCD to display the custom character. The LCD's custom character definitions can be a value from 0 to 7, so **custChar** has to fall in that range. The custom character's **DATA** directive ***Symbol*** name has to be stored in the **charBase** variable. Since **variable = SymbolName** stores the EEPROM address of the first byte in a **DATA** directive defined by **SymbolName** into variable, it makes it possible to pass the starting address of a custom character's **DATA** directive to the **Def_Cust_Char** subroutine.

Q3. Set the value of the **cursor** and **line** variables before calling the subroutine. The **Disp_Cust_Char** routine will position the cursor using these two variables before telling the LCD to display the custom character at that location.

Q4. No, because all instances of the same custom character currently displayed by the LCD will be updated as soon as the character definition is updated.

Q5. The **dotLine** variable ends up storing one of five values: %00000, %10000, %11000, %11100, or %11110. Each digit corresponds to one of the five vertical columns of pixels in a custom character. If the value is 1, the pixels in that column will be set to black. If it's 0, they will be left white.

Q6. Each of the eight rows in the custom character definition will have the same pattern, so the program only has to send the same row byte eight times to define each row of the custom character.

Q7. When the **IF…THEN** condition evaluates to 1, the pixels for the row are set black with **SEROUT 14, 84, [%11111]**; otherwise, they are set white with **SEROUT 14, 84, [%00000]**.

E1. Example solution:

```
Stripes         DATA        %11111,          ' * * * * *
                            %00000,          '
                            %11111,          ' * * * * *
                            %00000,          '
                            %11111,          ' * * * * *
                            %00000,          '
                            %11111,          ' * * * * *
                            %00000           '
```

E2. The goal of the expression **`value = (x MIN 1875 MAX 3125) - 1875 ** 1625 + 1`** in Activity #3 is to scale the x-axis accelerometer value, which could be anywhere from 1875 to 3125, to a value from 1 to 31. This will pick one of 31 possible vertical bar graph possibilities for the **`Bar_Graph_H_V`** subroutine. By the time the **`**`** operator is reached, 1875 has already been subtracted, so the range is 0 to 1250 (1251 input scale elements), and we want to scale that to a range of 1 to 31 (31 output scale elements). From Chapter 3, Activity #3, we know that ** ScaleConstant = Int[65536 × (output scale elements ÷ (input scale elements – 1))]. So, it's Int[65536 × (31 ÷ (1251 – 1))] = Int[65536 × (31 ÷ 1250)] = Int[65536 × 0.0248] = Int[1625.2928] = 1625.

P1. Example solution – modified main routine from PingBarGraph.bs2.

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

DO                                      ' Main loop

  PULSOUT 15, 5
  PULSIN 15, 1, time

  cmDistance = CmConstant ** time

  DEBUG HOME, DEC3 cmDistance, " cm"

  value = cmDistance

  GOSUB Bar_Graph                       ' Display as bar graph

  SEROUT 14, 84, [Line1, DEC4 value,    ' Display cm measurement
                  " cm"]                 ' on Line 1

  PAUSE 100

LOOP                                    ' Repeat main loop
```

P2. Example solution – See comments in the title section for how this program was constructed from earlier example programs and calculations explained in the text:

```
' -----[ Title ]-----------------------------------------------------------
' Smart Sensors and Applications - Ch7Proj2.bs2
' Displays a character that shifts both vertically and horizontally based
' on the MX2125's tilt angle in degrees.  This program is a combination of
' HorizontalTilt.bs2 from Chapter 3, Activity #6 and TwoAxisBarTiltDisplay.bs2
' from Chapter 7, Activity #3.  The x-axis measurement had to be changed from
' -90 to +90 degrees to a value from 1 to 31 using the ** scale constant
' equation.  Likewise, the -90 to +90 degree y-axis measurement had to be
' changed to a value from 0 to 16.  The calculations are commented and shown
' in the main routine just above their use with the ** operators.

' {$STAMP BS2}                              ' Target device = BASIC Stamp 2
' {$PBASIC 2.5}                             ' Language      = PBASIC 2.5

' -----[ I/O Pins ]---------------------------------------------------------

LcdPin         PIN     14                   ' I/O pin connected to LCD's RX

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

T9600        CON     84                   ' True, 8-bits, no parity, 9600

LcdCls         CON     12                   ' Form feed -> clear screen
LcdCr          CON     13                   ' Carriage return
LcdOff         CON     21                   ' Turns display off
LcdOn          CON     22                   ' Turns display on
Line0          CON     128                  ' Line 0, character 0
Line1          CON     148                  ' Line 1, character 0
Define         CON     248                  ' Address defines cust char 0
Negative       CON     1                    ' Sign - .bit15 of Word variables
Positive       CON     0

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

custChar       VAR     Byte                 ' Custom character selector
index          VAR     Nib                  ' Eeprom index variable
rowMap         VAR     Byte                 ' 5-pixel dotted line
cursor         VAR     Byte                 ' Cursor placement
value          VAR     Byte                 ' Value to be graphed
line           VAR     Byte                 ' Line0 or Line1
x              VAR     Word
y              VAR     Word
sine           VAR     Word            ' sine in circle r = 127
side           VAR     Word            ' trig subroutine variable
angle          VAR     Word            ' result angle - degrees
sign           VAR     Bit             ' Sign bit
```

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

PAUSE 100                                   ' Debounce power supply
SEROUT LcdPin, T9600, [LcdOn, LcdCls]       ' Initialize LCD
PAUSE 5                                     ' 5 ms delay for clearing display

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

DO                                          ' Main loop

  PULSIN 6, 1, x
  PULSIN 7, 1, y

  x = (x MIN 1875 MAX 3125) - 1875 ** 13369 - 127
  y = (y MIN 1875 MAX 3125) - 1875 ** 13369 - 127

  side = x
  GOSUB Arcsine
  ' Int[65536 * (31 / (181 - 1))] = 11286
  value = angle + 90 ** 11286 + 1

  side = y
  GOSUB Arcsine
  ' Int[65536 * (17 / (181 - 1))] = 6189
  cursor = 16 - (angle + 90 ** 6189)


  GOSUB Bar_Graph_H_V                      ' Call Bar_Graph_H_V

LOOP                                        ' Repeat main loop

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

' Defines and displays two axis bar graph characters based on the value of
' the cursor (0 to 16) and value (1 to 31).  Calls Def_Vert_Bar_Char, and
' Horizontal_Placement.

Bar_Graph_H_V:

  SEROUT 14, 84, [LcdCls]                   ' Clear previous plot
  PAUSE 5                                   ' 5 ms delay for clearing display

  ' Decide whether to display on Line 0 or Line 1.
  IF value >= 16 THEN line = Line0 ELSE Line = Line1

  GOSUB Def_Vert_Bar_Char                   ' Define custom character
  GOSUB Horizontal_Placement

  IF value = 16 THEN                        ' Special case: value = 16
```

```
    value = 1                                ' Line 0 display
    custChar = 2
    GOSUB Def_Vert_Bar_Char
    line = Line0
    GOSUB Horizontal_Placement

    value = 15                               ' Line 1 Display
    custChar = 3
    GOSUB Def_Vert_Bar_Char
    line = Line1
    GOSUB Horizontal_Placement

    value = 16                               ' Restore value

  ENDIF

  RETURN

' -----[ Subroutine - Def_Vert_Bar_Char ]-----------------------------------

' Defines a vertical bar graph character based on the value variable
' (0 to 15) and the custChar variable, which selects the Parallax Serial
' LCD's custom characters between 0 and 7.

Def_Vert_Bar_Char:

  ' Start define custom character
  SEROUT LcdPin, T9600,
         [Define + custChar]

  ' Select a row map for the custom character based on value.
  rowMap = %1111111100000000 >> (value & %1111)

  ' Send 8 bytes, each defining a dotted row in the custom character.  Each
  ' row is determined by examining a bit in the rowMap variable, and then
  ' sending %11111 if the bit is 1, or %00000 if the bit is 0.
  FOR index = 0 TO 7                         ' Repeat 7 times, index counts
    IF rowMap.LOWBIT(index) = 1 THEN         ' Examine next bit in rowMap
      SEROUT 14, 84, [%11111]                ' If 1, send %11111
    ELSE
      SEROUT 14, 84, [%00000]                ' Otherwise, send %00000
    ENDIF
  NEXT
                                             ' Return from subroutine.
  RETURN

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

' Places the vertical bar graph at one of the Parallax 2X16 LCD's sixteen
' vertical columns.  The cursor variable can set the horizontal location to
```

```
' values between 0 and 16, with 8 the center.  Calls Disp_Custom_Char.

Horizontal_Placement:

  SELECT cursor                               ' Cursor 0 to 7, no changes
    CASE 0 TO 7
      GOSUB Disp_Cust_Char
    CASE 8                                    ' Cursor 8, display at 7 & 8
      cursor = 7
      GOSUB Disp_Cust_Char
      cursor = 8
      GOSUB Disp_Cust_Char
    CASE 9 TO 16                              ' Cursor 9 to 16, display 1 left
      cursor = cursor - 1
      GOSUB Disp_Cust_Char
      cursor = cursor + 1
    ENDSELECT

  RETURN

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

' This subroutine displays a custom character.  The line variable can
' be set to either Line0 or Line1, and the cursor variable can be set
' to a value between 0 and 15.  The custChar variable selects one of the
' LCD's custom characters and should be set to a value between 0 and 7.

Disp_Cust_Char:

  SEROUT LcdPin, T9600,                    ' Print custom character
         [line + cursor, custChar]
  RETURN

' -----[ Subroutine - Arcsine ]---------------------------------------------

' This subroutine calculates arcsine based on the y coordinate on a circle
' of radius 127.  Set the side variable equal to your y coordinate before
' calling this subroutine.

Arcsine:                                  ' Inverse sine subroutine
  GOSUB Arccosine                         ' Get inverse cosine
  angle = 90 - angle                      ' sin(angle) = cos(90 - angle)
  RETURN

' -----[ Subroutine - Arccosine ]-------------------------------------------

' This subroutine calculates arccosine based on the x coordinate on a circle
' of radius 127.  Set the side variable equal to your x coordinate before
' calling this subroutine.

Arccosine:                                  ' Inverse cosine subroutine
```

```
  sign = side.BIT15                            ' Save sign of side
  side = ABS(side)                             ' Evaluate positive side
  angle = 63 - (side / 2)                      ' Initial angle approximation
  DO                                           ' Successive approximation loop
    IF (COS angle <= side) THEN EXIT           ' Done when COS angle <= side
    angle = angle + 1                          ' Keep increasing angle
  LOOP
  angle = angle */ 361                         ' Convert brads to degrees
  IF sign = Negative THEN angle = 180 - angle' Adjust if sign is negative.
  RETURN
```

Appendix A: ASCII Chart

ASCII Chart (first 128 characters)

Dec	Hex	Char	Name / Function
0	00	NUL	Null
1	01	SOH	Start Of Heading
2	02	STX	Start Of Text
3	03	ETX	End Of Text
4	04	EOT	End Of Transmit
5	05	ENQ	Enquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal Tab
10	0A	LF	Line Feed
11	0B	VT	Vertical Tab
12	0C	FF	Form Feed
13	0D	CR	Carriage Return
14	0E	SO	Shift Out
15	0F	SI	Shift In
16	10	DLE	Data Line Escape
17	11	DC1	Device Control 1
18	12	DC2	Device Control 2
19	13	DC3	Device Control 3
20	14	DC4	Device Control 4
21	15	NAK	Non Acknowledge
22	16	SYN	Synchronous Idle
23	17	ETB	End Transmit Block
24	18	CAN	Cancel
25	19	EM	End Of Medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File Separator
29	1D	GS	Group Separator
30	1E	RS	Record Separator
31	1F	US	Unit Separator

Dec	Hex	Char
32	20	space
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	delete

Note that the control codes (lowest 32 ASCII characters) have no standardized screen symbols. The characters listed for them are just names used in referring to these codes. For example, to move the cursor to the beginning of the next line of a printer or terminal often requires sending line feed and carriage return codes. This common pair is referred to as "LF/CR."

Appendix B: Parallax Serial LCD Documentation

599 Menlo Drive, Suite 100
Rocklin, California 95765, USA
Office: (916) 624-8333
Fax: (916) 624-8003

General: info@parallax.com
Technical: support@parallax.com
Web Site: www.parallax.com
Educational: www.stampsinclass.com

Version 2.0

Parallax Serial LCD

2 rows x 16 characters Non-backlit (#27976)
2 rows x 16 characters Backlit (#27977)
4 rows x 20 characters Backlit (#27979)

Introduction

The Parallax Serial LCDs are very functional, low-cost LCDs that can be easily controlled by a BASIC Stamp® microcontroller. The LCD displays are either two rows by 16 characters or four rows by 16 characters, and provide basic text wrapping so that your text looks right on the display. In addition, the Serial LCDs also provide you with full control over all of their advanced LCD features, allowing you to move the cursor anywhere on the display with a single instruction and turn the display on and off in any configuration. They support the same visible characters as the BASIC Stamp Editor's Debug Terminal (ASCII Dec 32-127). In addition, you may define up to eight of your own custom characters to display anywhere on the LCD.

Application Ideas

What can you do with a Parallax Serial LCD? While there are many possibilities, here's a small list of ideas that can be realized with a Serial LCD and the Parallax BASIC Stamp:

- A professional-looking text user interface on any microcontroller application
- Easy-to-implement serial debugging without a PC
- Real-time sensor data output on autonomous robotics applications (Boe-Bot®, Toddler®, SumoBot®)

LCD Extension Cables

The Parallax Serial LCDs are compatible with our 14-inch LCD Extension Cables, part #805-00012, sold separately from www.parallax.com. This 3-pin female-female cable comes with a 3-pin header so you may conveniently connect your LCD to your breadboard projects.

Sample Code

Demonstration BASIC Stamp software files may be downloaded from:
http://www.parallax.com/detail.asp?product_id=27976
http://www.parallax.com/detail.asp?product_id=27977
http://www.parallax.com/detail.asp?product_id=27979

Features

- Displays ASCII character set directly to the display
- Wraps to the next line automatically for easy display of text strings
- Works at 2400, 9600, and 19,200 baud
- Moves the cursor anywhere on the display with a single command
- Clears the whole display with a single command
- Allows you to define up to eight custom characters

Connections

Connecting the Serial LCD to the BASIC Stamp is a straightforward operation, requiring just three IO pins. See Figure B-1 and B-2 for electrical connection details. See Figures B-3 and B-4 on the following pages for size and mechanical mounting details.

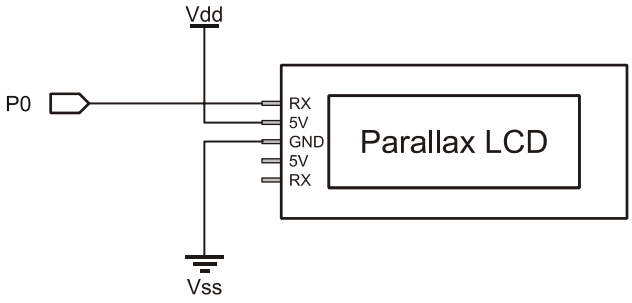


Figure B-1

Serial LCD connections for Rev D and earlier displays.

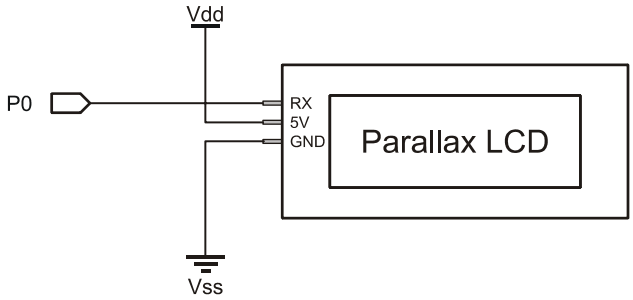


Figure B-2

Serial LCD connections for Rev E and later displays.

The table below lists the voltage and current requirements of the Serial LCD, with the backlight turned on and off. Since the current draw in either case exceeds the capabilities of all of the BASIC Stamp modules, you should power the Serial LCD from an external 5 V power supply. Make sure the power supply has an adequate current rating to power the Serial LCD and the BASIC Stamp.

Serial LCD State	Voltage	Current
All Models Backlight off	5 VDC	20 mA
27977/27979 Backlight on	5 VDC	80 mA

Figure B-3 Size and Mounting Specifications for Models 27976, 27977

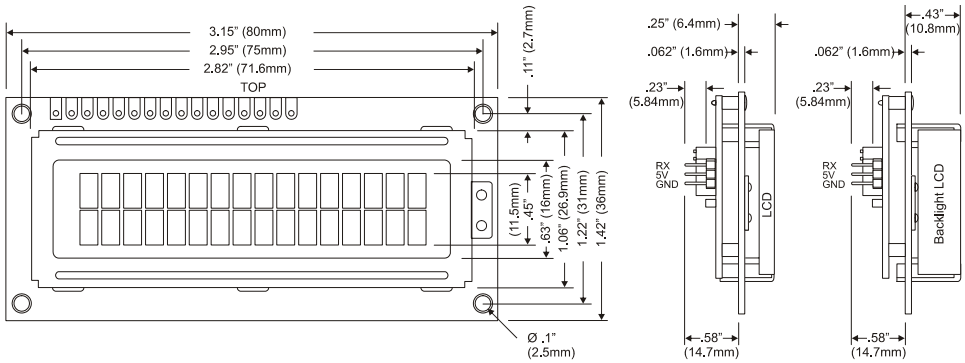
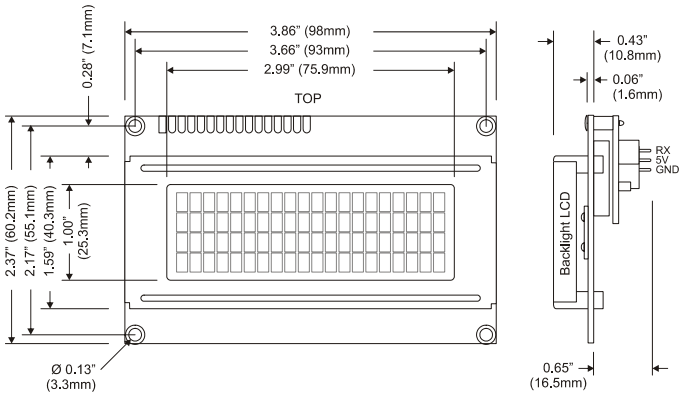


Figure B-4 Size and Mounting Specifications for Model 27979



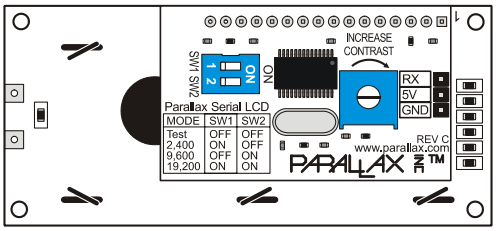
Technical Notes

Characteristic	Description
LCD display type	STN, YG, positive, transflective
Viewing direction	6H
Backlight	YG LED
Operating temperature	-4°F~158°F (-20°C~70°C)
Storage temperature	-22°F~176°F (-30°C~80°C)
Dimension tolerance	±.02" (.5mm)

Baud Rate Setup

After connecting the Serial LCD, you will need to select the baud rate at which you are going to send it data. You have three choices: 2400, 9600, and 19,200 baud. To set the baud rate, move the dip switches on the back of the LCD into the correct positions according to the table next to the switches, which is also repeated below:

MODE	SW1	SW2
Test	OFF	OFF
2,400	ON	OFF
9,600	OFF	ON
19.200	ON	ON



As you can see from the table, there is also a fourth choice called Test. Now that you've connected the power, use this Test mode to confirm that the power and ground to the LCD are hooked up correctly before you send it any data. Move the dip switches to the Test setting and turn on the power. The LCD display should turn on with the backlight on (models 27977, 27979) and display the following text:

Parallax, Inc.
www.parallax.com

If you don't see the text at first, try adjusting the LCD contrast by turning the pot labeled "Increase Contrast" with a screwdriver. Turn it in the direction of the arrow to make the characters show up more clearly. If you still don't see the characters, go back and check your electrical connections and try again. Once you've successfully completed test mode, move the dip switches to the correct positions to select the baud rate you want to use for your application.

Displaying Text

Now that the LCD is set up, it's time to start sending text to the display. To display a character of text on the Serial LCD, simply send the ASCII code of that character to the Serial LCD over the serial port at the correct baud rate.

When a character is received, the Serial LCD displays that character at the current cursor position and then moves the cursor one position to the right. When you first turn on the LCD, the cursor is in the leftmost position on the top line, as you might expect. The short bar on the bottom of the character shows where the cursor is positioned currently.

Once you've sent a full line characters to the LCD, you will notice that the cursor automatically wraps around to the leftmost position of the second line, just like the text in a book. The text will wrap like this at the end of every line, with the end of the bottom line wrapping back around to the top line of the LCD. The text will never "run off" the display; you'll always see all of the characters you send.

Try the following code on your BASIC Stamp 2 to send a text string to the LCD display. First, set the baud rate on your Serial LCD to 19,200. Then, load the code below into your BASIC Stamp 2 and run it. You will see the text string show up and wrap to the second line of the display.

In all of your Serial LCD code, you should pause for 100 ms at start-up to give time for the Serial LCD to initialize. You should also set the serial port pin on the BASIC Stamp to HIGH before the 100 ms start-up delay, as this is the normal state of a serial port when it isn't sending any data.

```
' {$STAMP BS2}

TxPin           CON     0
Baud19200       CON     32

  HIGH TxPin                    ' Set pin high to be a serial port
  PAUSE 100                     ' Pause for Serial LCD to initialize
  SEROUT TxPin, Baud19200, ["Hello, this text will wrap."]
```

Moving the Cursor

When you send a character to the Serial LCD, it always displays at the current cursor position. There are a few different ways to move the cursor on the Serial LCD display. After each character you send, the cursor automatically moves over one position. Along with this, there is a standard set of cursor move commands including Backspace, Carriage Return, and Line Feed.

The Backspace/Left command (Dec 8) moves the cursor one place to the left and the Right command (Dec 9) moves the cursor one place to the right. These can be useful for moving the cursor around to overwrite existing text. These commands wrap to the next line of the display, if necessary. The Line Feed command (Dec 10) moves the cursor to the next line of the display without changing the horizontal position of the cursor. The Carriage Return command (Dec 13) also moves the cursor to the next line, but it moves the cursor to the leftmost position on that line as well. The Form Feed command (Dec 12) clears the entire display and moves the cursor to the leftmost position on Line 0, just like when you first turn on the display. You will need to pause for 5mS in your code after sending the Form Feed command, to give the Serial LCD time to clear the display. Except for Form Feed, none of these move commands affects the characters on the display.

There are also direct move commands that you can use to move the cursor to any position on the display with a single command. The commands in the range Dec 128 to 143 and Dec 148 to 163 move the cursor to the 16 different positions on each of the two lines of

the model 27976 and 27977 LCDs. The commands in the range Dec 128 to 207 move the cursor to the 20 different positions on each of the four lines of the model 27979 LCD.

Controlling the Display

You also have control over the various display modes of the Serial LCD. The display-off command (Dec 21) turns off the display so that all of the characters disappear. The characters aren't erased from the display, though, and you can even keep writing new characters to the display when it is turned off. A trick to make a lot of text show up all at once, even at a slow baud rate, is to turn off the display and then send all of your text. Then, when you turn the display on again, all of the text appears instantly.

The display-on commands (DEC 22 to 25) turn the display back on and also control whether you want to display the cursor and/or make the cursor character blink. The cursor is the short bar that shows up below the character at the current cursor position. The blink option makes that character blink on and off repeatedly. You can turn the cursor and blink options on or off, in any combination, as listed in the command set table. You can change the cursor and blink mode even if the display is already on; you don't need to turn it off and then back on again.

With models 27977 and 27979, you can also control the backlight of the display. The backlight lights up the display so that it is easier to see in the dark. There are commands to turn the backlight on (Dec 17) and off (Dec 18).

Custom Characters

The Serial LCD has the capability to store up to eight user-defined custom characters. The custom characters are stored in RAM and so they need to be redefined if you turn off the power. You can display the custom characters by sending the commands Dec 0 to 7, as shown in the command set table. The custom character will display at the current cursor position.

The custom characters are five pixels wide by eight pixels high. Each of the characters is stored as a series of eight data bytes where the low five bits of each byte represent a row of pixels in the character. The high three bits of each byte are ignored. A bit value of one turns that pixel on (i.e. makes it black). The bottom row of pixels is often left blank (all zeros) to make it easier to see the cursor.

To define a custom character, you will send a total of 9 bytes to the Serial LCD. The first byte needs to be a valid define-custom-character command (Dec 248 to 255) and must be followed by eight data bytes that define the pixels of the character. The Serial LCD will always use the next eight bytes it receives to set the pixels of the character. The data bytes define the character starting at the topmost row of pixels, as shown in the example code.

Define a custom character using the code example below. First, set the baud rate on your Serial LCD to 19,200. Then, load the code below into your BASIC Stamp 2 and run it. You will see a diamond character appear on the screen.

```
' {$STAMP BS2}

TxPin           CON     0
Baud19200       CON     32

  HIGH TxPin                        ' Set pin high to be a serial port
  PAUSE 100                         ' Pause for Serial LCD to initialize

  SEROUT TxPin, Baud19200, [250]  ' Define Custom Character 2
                                    ' Now send the eight data bytes
  SEROUT TxPin, Baud19200, [0]    ' 0  = %00000
  SEROUT TxPin, Baud19200, [4]    ' 4  = %00100      *
  SEROUT TxPin, Baud19200, [14]   ' 14 = %01110    * * *
  SEROUT TxPin, Baud19200, [31]   ' 31 = %11111  * * * * *
  SEROUT TxPin, Baud19200, [14]   ' 14 = %01110    * * *
  SEROUT TxPin, Baud19200, [4]    ' 4  = %00100      *
  SEROUT TxPin, Baud19200, [0]    ' 0  = %00000
  SEROUT TxPin, Baud19200, [0]    ' 0  = %00000
  SEROUT TxPin, Baud19200, [2]    ' Display the new Custom Character 2
```

Command Set

The tables below list all of the valid Serial LCD commands. Commands marked as N/A are invalid and are ignored. The lines of the LCD display are numbered starting from 0, with Line 0 being the top line. The character positions on each line are numbered starting from 0, with position 0 being the leftmost position on the line.

B

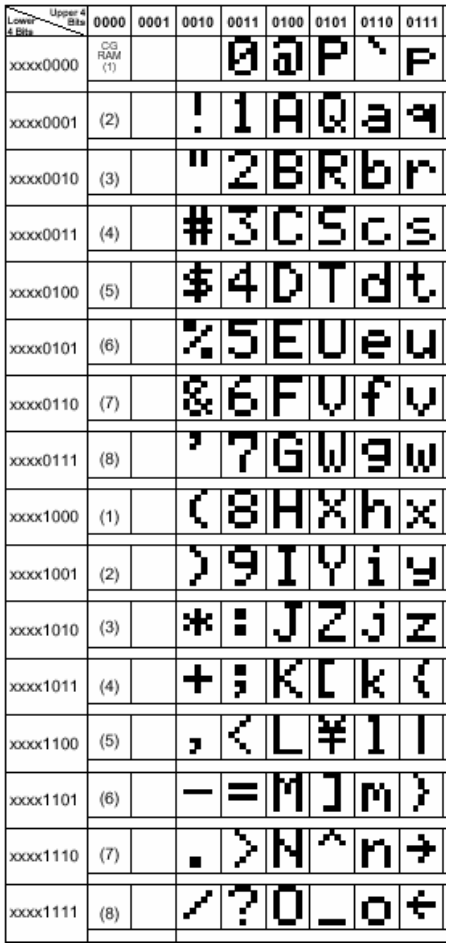
Dec	Hex	Action
0	00	Display Custom Character 0
1	01	Display Custom Character 1
2	02	Display Custom Character 2
3	03	Display Custom Character 3
4	04	Display Custom Character 4
5	05	Display Custom Character 5
6	06	Display Custom Character 6
7	07	Display Custom Character 7
8	08	Backspace / Left - The cursor is moved one position to the left. The command doesn't erase the character.
9	09	Right - The cursor is moved one position to the right. The command doesn't erase the character.
10	0A	Line Feed - The cursor is moved down one line. For the two line LCD model, if on line 0 it goes to line 1. If on line 1, it wraps around to line 0. The horizontal position remains the same.
11	0B	N/A
12	0C	Form Feed - The cursor is moved to position 0 on line 0 and the entire display is cleared. Users must pause 5mS after this command.
13	0D	Carriage Return – For the two-line LCD models, if on line 0 the cursor is moved to position 0 on line 1. If on line 1, it wraps around to position 0 on line 0.
14 - 16	0E - 10	N/A
17	11	Turn backlight on (only on models 27977, 27979)
18	12	Turn backlight off (Default)
19 - 20	13 - 14	N/A
21	15	Turn the display off
22	16	Turn the display on, with cursor off and no blink
23	17	Turn the display on, with cursor off and character blink
24	18	Turn the display on, with cursor on and no blink (Default)
25	19	Turn the display on, with cursor on and character blink
26 - 31	1A - 1F	N/A
32 - 127	20 - 7F	Display ASCII characters. See the ASCII character set table.
128	80	Move cursor to line 0, position 0
129	81	Move cursor to line 0, position 1
130	82	Move cursor to line 0, position 2
131	83	Move cursor to line 0, position 3
132	84	Move cursor to line 0, position 4
133	85	Move cursor to line 0, position 5
134	86	Move cursor to line 0, position 6

Dec	Hex	Action
135	87	Move cursor to line 0, position 7
136	88	Move cursor to line 0, position 8
137	89	Move cursor to line 0, position 9
138	8A	Move cursor to line 0, position 10
139	8B	Move cursor to line 0, position 11
140	8C	Move cursor to line 0, position 12
141	8D	Move cursor to line 0, position 13
142	8E	Move cursor to line 0, position 14
143	8F	Move cursor to line 0, position 15
144	90	Move cursor to line 0, position 16 (only on model 27979)
145	91	Move cursor to line 0, position 17 (only on model 27979)
146	92	Move cursor to line 0, position 18 (only on model 27979)
147	93	Move cursor to line 0, position 19 (only on model 27979)
148	94	Move cursor to line 1, position 0
149	95	Move cursor to line 1, position 1
150	96	Move cursor to line 1, position 2
151	97	Move cursor to line 1, position 3
152	98	Move cursor to line 1, position 4
153	99	Move cursor to line 1, position 5
154	9A	Move cursor to line 1, position 6
155	9B	Move cursor to line 1, position 7
156	9C	Move cursor to line 1, position 8
157	9D	Move cursor to line 1, position 9
158	9E	Move cursor to line 1, position 10
159	9F	Move cursor to line 1, position 11
160	A0	Move cursor to line 1, position 12
161	A1	Move cursor to line 1, position 13
162	A2	Move cursor to line 1, position 14
163	A3	Move cursor to line 1, position 15
164	A4	Move cursor to line 1, position 16 (only on model 27979)
165	A5	Move cursor to line 1, position 17 (only on model 27979)
166	A6	Move cursor to line 1, position 18 (only on model 27979)
167	A7	Move cursor to line 1, position 19 (only on model 27979)
168	A8	Move cursor to line 2, position 0 (only on model 27979)
169	A9	Move cursor to line 2, position 1 (only on model 27979)
170	AA	Move cursor to line 2, position 2 (only on model 27979)
171	AB	Move cursor to line 2, position 3 (only on model 27979)
172	AC	Move cursor to line 2, position 4 (only on model 27979)
173	AD	Move cursor to line 2, position 5 (only on model 27979)
174	AE	Move cursor to line 2, position 6 (only on model 27979)
175	AF	Move cursor to line 2, position 7 (only on model 27979)
176	B0	Move cursor to line 2, position 8 (only on model 27979)
177	B1	Move cursor to line 2, position 9 (only on model 27979)
178	B2	Move cursor to line 2, position 10 (only on model 27979)
179	B3	Move cursor to line 2, position 11 (only on model 27979)
180	B4	Move cursor to line 2, position 12 (only on model 27979)

Dec	Hex	Action
181	B5	Move cursor to line 2, position 13 (only on model 27979)
182	B6	Move cursor to line 2, position 14 (only on model 27979)
183	B7	Move cursor to line 2, position 15 (only on model 27979)
184	B8	Move cursor to line 2, position 16 (only on model 27979)
185	B9	Move cursor to line 2, position 17 (only on model 27979)
186	BA	Move cursor to line 2, position 18 (only on model 27979)
187	BB	Move cursor to line 2, position 19 (only on model 27979)
188	BC	Move cursor to line 3, position 0 (only on model 27979)
189	BD	Move cursor to line 3, position 1 (only on model 27979)
190	BE	Move cursor to line 3, position 2 (only on model 27979)
191	BF	Move cursor to line 3, position 3 (only on model 27979)
192	C0	Move cursor to line 3, position 4 (only on model 27979)
193	C1	Move cursor to line 3, position 5 (only on model 27979)
194	C2	Move cursor to line 3, position 6 (only on model 27979)
195	C3	Move cursor to line 3, position 7 (only on model 27979)
196	C4	Move cursor to line 3, position 8 (only on model 27979)
197	C5	Move cursor to line 3, position 9 (only on model 27979)
198	C6	Move cursor to line 3, position 10 (only on model 27979)
199	C7	Move cursor to line 3, position 11 (only on model 27979)
200	C8	Move cursor to line 3, position 12 (only on model 27979)
201	C9	Move cursor to line 3, position 13 (only on model 27979)
202	CA	Move cursor to line 3, position 14 (only on model 27979)
203	CB	Move cursor to line 3, position 15 (only on model 27979)
204	CC	Move cursor to line 3, position 16 (only on model 27979)
205	CD	Move cursor to line 3, position 17 (only on model 27979)
206	CE	Move cursor to line 3, position 18 (only on model 27979)
207	CF	Move cursor to line 3, position 19 (only on model 27979)
208 - 247	D0 – F7	N/A
248	F8	Define Custom Character 0. Command must be followed by eight data bytes.
249	F9	Define Custom Character 1. Command must be followed by eight data bytes.
250	FA	Define Custom Character 2. Command must be followed by eight data bytes.
251	FB	Define Custom Character 3. Command must be followed by eight data bytes.
252	FC	Define Custom Character 4. Command must be followed by eight data bytes.
253	FD	Define Custom Character 5. Command must be followed by eight data bytes.
254	FE	Define Custom Character 6. Command must be followed by eight data bytes.
255	FF	Define Custom Character 7. Command must be followed by eight data bytes.

ASCII Character Set

The table below shows all of all the ASCII characters as they are displayed on the Serial LCD. All of the ASCII characters (Dec 32 to 127) are standard ASCII characters, except for the '\' back-slash (Dec 92) and '~' tilde (Dec 126) characters. For your convenience, the Serial LCD comes pre-programmed with these characters in the first two custom characters. So, to display a back-slash, use command Dec 0 and to display a tilde, use command Dec 1. Of course, you can always overwrite these characters with your own custom characters.



Appendix C: Hexadecimal Character Definitions

Lots of LCD application notes and documentation use hexadecimal numbers to instead of binary numbers to define commands and characters. In PBASIC, printing an exclamation point is a simple matter of a **SEROUT** command with an exclamation point in quotes.

```
SEROUT 14, 84, ["!"]
```

Not all programming languages for controllers support native use of printable characters like that. In some cases, mostly in assembly language, the ASCII code for the exclamation point is used instead. The ASCII code for the exclamation point is 33, and even in PBASIC, the command **SEROUT 14, 84, [33]** accomplishes the same task. In assembly language, hexadecimal values are sometimes the preferred number base because it makes certain tasks easier. Because of this, lots of LCD documentation lists their LCD commands as hexadecimal values.

The hexadecimal equivalent of decimal-33 is hexadecimal-21. That's (2 × 16) + 1. You can use the $ operator to specify that a value is hexadecimal, which would make the command to display an exclamation point look like this: **SEROUT 14, 84, [$21]**.

Here is an example of a **SEROUT** command that defines a bar in a bar graph. If fills the lower half of the character with black pixels and leaves the upper half white:

```
SEROUT 14, 84, [250,
                %00000,          '
                %00000,          '
                %00000,          '
                %00000,          '
                %11111,          ' * * * * *
                %11111,          ' * * * * *
                %11111,          ' * * * * *
                %11111]          ' * * * * *
```

Here is an equivalent command using hexadecimal values instead. While it saves a lot of space, it's still not as easy to understand as defining a custom character with binary numbers. Be that as it may, you will see PBASIC application programs written this way from time to time, mainly because of the prevalence of hexadecimal values in LCD documentation.

```
SEROUT 14, 84, [250, $00, $00, $00, $00, $1F, $1F, $1F, $1F]
```

Table 7-1 counts to 15 in decimal, hexadecimal, and binary. In terms of converting from hexadecimal to binary and back, this table is all you'll need. Reason being, each single hexadecimal digit corresponds to a group of four binary digits.

Table 7-1: Decimal, Hexadecimal and Binary values

Base 10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base 16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Base 2	00 00	00 01	00 10	00 11	01 00	01 01	01 10	01 11	10 00	10 01	10 10	10 11	11 00	11 01	11 10	11 11

Example: Convert $8FE6 to binary.
Solution: Each hexadecimal digit converts to a group of four binary numbers, so the binary value can be written with the help of Table 7-1 like this:

Hexadecimal	8	F	E	6
Binary	1000	1111	1110	0110

$8FE6 = %1000111111100110

Example: Convert %1100011000001001 to hexadecimal.
Solution: Arrange the binary number into groups of four digits, then use Table 7-1:

Binary	1100	0110	0000	1001
Hexadecimal	C	6	0	9

%1100011000001001 = $C609

Example Program: ConvertBinaryToHexadecimal.bs2

Of course, you can also make the BASIC Stamp do it for you. Simply use the % operator to define the binary number, and then use the **HEX** formatter in a **SEROUT** command to display the value.

√ Try it and see if the LCD agrees with our calculations

```
' Smart Sensors and Applications - ConvertBinaryToHexadecimal.bs2

' {$STAMP BS2}
```

```
' {$PBASIC 2.5}

PAUSE 250                                       ' Debounce power supply
SEROUT 14, 84, [22, 12]                         ' Turn on display and clear
PAUSE 5                                         ' 5 ms delay for clearing display

SEROUT 14, 84, ["Value = ", HEX %1100011000001001]

END
```

Your Turn - Converting from Hexadecimal to Binary

Converting from hexadecimal to binary is a matter of using the **BIN** formatter instead of **HEX**, and using the **$** operator to tell the BASIC Stamp Editor you are giving it a hexadecimal (instead of % for binary). If you are converting four hexadecimal digits to binary, the result will be a 16-digit number. So, the **SEROUT** command should also be modified to display the result starting at the beginning of the second line.

√ Comment the existing line of code that performs the binary to hexadecimal conversion.

```
SEROUT 14, 84, ["Value = ", 148, BIN $8FE6]
```

Appendix D: Parts Listing

Computer System Requirements:

- PC running Windows 2000/XP
- An available serial port or USB port. If you need a USB to Serial Adapter, we recommend Parallax part #800-00030.
- Internet access

Software Requirements:

- BASIC Stamp Editor for Windows v2.0 or higher (Free download from www.parallax.com)
- Selected Example Programs (Free download from www.parallax.com)
- Microsoft Notepad and Microsoft Excel 2002 or higher (for Chapter 6 acceleration studies)

Hardware Requirements

- One of the following kits that includes a BASIC Stamp 2 programming platform, plus the appropriate power supply or batteries:
  - Board of Education Full Kit Serial (#28102) or USB (#28802)
  - Boe-Bot Robot Kit Serial (#28132) or USB (#28832)
  - BASIC Stamp Activity Kit***** (includes HomeWork Board) (#90005)
- Smart Sensors and Applications Parts & Text (#28029) or Parts Only Kit (#130-28029)

Household Items:

- Small bar magnet
- Mechanical Compass (for calibrating the Compass Module, Chapter 4)
- RC Car and Controller, with Batteries (for Chapter 6, Activity #5)
- Bicycle Wheel (for Chapter 6, Activity #6)

*All of the activities in this text are compatible with the BASIC Stamp HomeWork Board as long as you are also using the non-backlit LCD included in the Smart Sensors kit. If you are using a back-lit LCD, also use a Board of Education to protect your LCD.

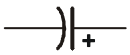
Smart Sensors and Applications Parts & Text #28029 (Without book, #130-28029) Parts and quantities subject to change without notice		
Parallax Part #	Description	Quantity
122-28029	Smart Sensors and Applications book	1
150-02210	220 Ω Resistors, ¼ watt 5	2
27976	Parallax Serial LCD 2 x 16 non-backlit	1
28015	Ping))) Ultrasonic Distance Sensor	1
28017	Memsic Dual-Axis Accelerometer	1
29123	Hitachi Compass Module	1
451-00303	3-pin male/male header	3
700-00003	4-40 zinc-plated nuts	6
700-00028	1/4" 4-40 pan-head screws	4
710-00006	1/2" 4-40 pan-head screws	2
713-00005	1/4" round nylon spacer #4	2
720-00011	90-degree universal mounting brackets	4
800-00016	3-inch Jumper Wires, bag of 10	2
805-00002	Servo/LCD Extension Cable	2
900-00001	Piezospeaker	1



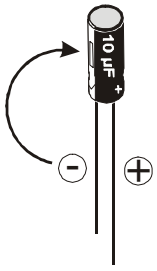
You may find that you have some extra electronic components in your kit that are not used in the *Smart Sensors and Applications* book. If you want to use these parts to build your own circuits, please learn about electrolytic capacitors and their safety requirements, below, before proceeding. **WARNING: Incorrect use of electrolytic capacitors can cause them to explode. Follow the safety guidelines below to avoid possible injury.**

Electrolytic capacitors have a positive (+) and a negative (-) terminal. The voltage at the capacitor's (+) terminal must always be higher than the voltage at its (-) terminal. Use the picture (right) to identify the (+) and (-) terminals. Always make sure to connect these terminals exactly as shown in reliable 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. 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.

Capacitor Symbol



10 µF



SAFETY

Always disconnect power before you build or modify circuits.

Always observe polarity when connecting electrolytic capacitors.

Never reverse the supply polarity on any polar capacitor.

Wear safety goggles or safety glasses when using these capacitors.

Keep your hands and face away from these capacitors when powered.

Index

- A -

acceleration, 65
- dynamic, 67
- on a circular path, 233
- RC car study, 230
- skateboard trick study, 240
- static, 67

accelerometer
- and LCD tilt bubble graph, 188
- measuring height, 211
- MX2125 design (picture), 67
- schematic and wiring diagram, 68
- three-axis, 66
- uses, 65

animation
- flashing text, 11
- hourglass custom characters, 24
- pixel worm custom characters, 264
- scrolling text, 25
- sliding window, 26

arccosine, 99
arcsine, 99
arctangent, 85, 92
ASCII chart, 303
ATN, 85, 127
averaging compass measurements, 144

- B -

bar graph
- horizontal, 271
- two-axis, 291
- vertical, 281

bar magnet, 125
baud rate, 7
baud rate switches for LCD, 6
bicycle distance measurement, 247
binary radians (brads), 86, 101
- converting to degrees, 87

- C -

calipers, 49
capacitor safety, 322
Cartesian coordinates, 174
Celsius to Fahrenheit conversion, 59
clamping input range, 78
CLREOL, 96
collision, 65
compass (drawing), 129
compass module, Hitachi HM55B, 119
- calibration, 128
- interpreting measurements, 119
- magnet cautions, 125
- schematic and wiring diagram, 121
- sensing axes, 120
- testing, 121

computer system requirements, 321

contrast adjustment for LCD, 7
conversion
  binary radians to degrees, 87
  Fahrenheit to Celsius temperature, 59
  hexadecimal to binary numbers, 319
coordinate systems, 174
COS, 101
cosine, 99
counting wheel revolutions, 248
CR, 10
CRSRXY, 8, 168
custom characters
  defining, 20
  hourglass animation, 24
  pixel worm animation, 264
  predefined in Parallax Serial LCD, 19
  swapping, 261

- D -

DATA, 180, 213, 223
datalogging, 223
deadband, 249
declination, 124
degree symbol in ASCII, 143
digital thermometer, 1
display coordinates, 174
dynamic acceleration, 67

- E -

Earth's magnetic field, 124
EEPROM, 214, 262

- F -

Fahrenheit to Celsius conversion, 59

- G -

Gelfand, Alan (Ollie), 241
graphic character display, 168
gravity, 65, 83

- H -

hardware requirements, 321
height measurement with accelerometer, 211
hexadecimal, 22, 317
hexadecimal to binary conversion, 319
HIDs (Human Interface Devices), 167
horizontal bar graph, 271
household items required, 321
hysteresis, 248

- I -

inclination, 124
incline, 65
input range clamping, 78

- L -

LCD
  and accelerometer tilt bubble graph, 188
  baud rate switches, 6
  cautions for older models, 4
  contrast adjustment, 7
  control codes, 9
  creating custom characters, 264
  defining custom characters, 20
  horizontal bar graph, 271
  in commercial products, 2
  mounting brackets, 153
  Parallax Serial LCD Documentation, 305

predefined custom characters, 19
schematic and wiring diagram, 5
scrolling text, 25
scrolling text in window, 30
sliding-window, 26
two-axis bar graph, 291
vertical bar graph, 281
liquid crystal display (LCD), 1

- M -

magnetic field, 124
magnetic field intensity, 124
mechanical compass, 130
MEMS technology, 65
MIN, 196
momentum, 241

- N -

negative numbers and PBASIC, 80, 174
and MIN operator, 196
negative numbers in PBASIC
division, 145

- O -

offsetting input values, 76
ollie, 241
oscillations, 242

- P -

Parts Kit component listing, 322
percent error measurements, 58
piezospeaker circuit, 223
Ping))) sensor, 41
and bar graph LCD display, 271
distance measurements, centimeter, 46
distance measurements, inches, 49
extension cable connections, 51
schematic and wiring diagram, 43
useful limits, 44
Predefined Custom Characters, 20
Programs
BikeWheelAcceleration.bs2, 251
BradsToDegrees.bs2, 90
BubbleGraph.bs2, 191
CalibrateCompass.bs2, 132
ConvertBinaryToHexadecimal.bs2, 318
CrsrxyPlot.bs2, 169
CursorPositions.bs2, 15
DatalogAcceleration.bs2, 226
DatalogYaxisUnscaled.bs2, 244
EepromBackgroundDisplay.bs2, 181
EepromBackgroundRefresh.bs2, 185
EepromDataStorage.bs2, 216
EepromDataStorageWithReset.bs2, 221
EepromPixelWorm.bs2, 264
HorizBarGraph.bs2, 273
HorizontalTilt.bs2, 108
Hourglass.bs2, 23
LcdTestCompass.bs2, 155
LcdTestMessage.bs2, 10
LcdTestNumbers.bs2, 12
LcdTimer.bs2, 17
PingLcdCmAndIn.bs2, 57

PingMeasureCm.bs2, 48
PingMeasureCmAndIn.bs2, 50
PingTest.bs2, 43
PlotXYGraph.bs2, 172
PredfinedCustomCharacters.bs2, 20
SignedNumbers.bs2, 81
SimpleTilt.bs2, 70
SimpleTiltLcd.bs2, 74
SineCosine.bs2, 101
TestArcsine.bs2, 103
TestAtn.bs2, 89
TestCalibratedCompass.bs2, 139
TestCompass.bs2, 122
TestCompassAveraged.bs2, 146
TestScaleOffset.bs2, 78
TestScrollingSubroutine.bs2, 28
TestVerticalBars.bs2, 283
TestWheelCounter.bs2, 249
TiltObstacleGame.bs2, 198
TwoAxisBarDisplay.bs2, 287
VertWheelRotation.bs2, 95

PULSIN, 69

- R -

RC car acceleration, 230
READ, 213
record and play back, 213
refresh rate, 179
Reset button as program switch, 219
rotation, 65, 85, 92, 229

- S -

scaling constant, 77
scaling input values, 76
scrolling text, 25
SEROUT, 9, 317
servo port jumper, 150
signed numbers and PBASIC, 174
and MIN operator, 196
signed numbers in PBASIC, 80
division, 145
SIN, 101
sine, 99
skateboard trick acceleration study, 240
sliding-window, 26
software requirements, 321
sound, speed of, and temperature, 58
static acceleration, 67

- T -

temperature unit conversion, 59
tesla (T), 124
tilt, 65, 98
tilt game controller, 196
tracking character coordinates, 182
two equations with two unknowns, 176
two’s complement, 80, 174, 196
two-axis bar graph, 291

- U -

unit circle, 100

- V -

vertial bar graph, 281
vibration, 65

- W -

Wainwright, Danny, 241
worm, custom character for LCD, 264
WRITE, 213



150-01020
(2) 1kΩ 1/4 W resistor
(brown, black, red)



700-00003
(6) 4-40 zinc-plated nuts



700-00028
(4) 1/4" 4-40 pan head screws



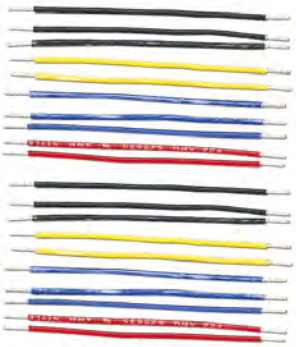
710-00006
(2) 1/2" 4-40 pan head screws



713-00005
(2) 1/4" round nylon spacer (#4)



720-00011
(4) 90-degree universal mounting brackets



800-00016
jumper wires (2 bags of 10)



27976
(1) Parallax Serial LCD 2x16, Non-backlit



28015
(1) PING))) Ultrasonic Distance Sensor



28017
(1) Memsic Dual-Axis Accelerometer



29123
(1) Hitachi HM55B Compass Module



805-00002
(2) Servo/LCD extension cables



451-00303
(3) 3-pin male/male headers

Parts and quantities in the Smart Sensors and Applications Parts Kit (#130-28029) are subject to change without notice.