IR Remote for the Boe-Bot

By Andy Lindsay

VERSION 1.1

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

DISCLAIMER OF LIABILITY

INTERNET DISCUSSION LISTS

We maintain active web-based discussion forums for people interested in Parallax products. These lists are accessible from www.parallax.com via the Support → Discussion Forums menu. These are the forums that we operate from our web site:

- BASIC 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 educational products in their courses. The list provides an opportunity for both students and educators to ask questions and get answers.
- Parallax Educators –Exclusively for educators and those who contribute to the development of Stamps in Class. Parallax created this group to obtain feedback on our educational products and to provide a forum for educators to develop and obtain Teacher's Guides and other resources.
- Translators – The purpose of this private list is to provide a conduit between Parallax and those who translate our documentation to languages other than English. Parallax provides editable Word documents to our translating partners and attempts to time the translations to coordinate with our publications. To join, email aalvarez@parallax.com.
- 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.
- ParallaxEFX – For animators, theatre prop builders, and those who create Halloween and other holiday displays using the ParallaxEFX product line.
- Propeller Chip – Forum for those using the Parallax Propeller chip.

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

AN AUTONOMOUS ROBOT AND A HANDHELD REMOTE

The handheld remote may be the tool of choice for channel surfing couch potatoes worldwide, but this device can also be used to send messages to your Boe-Bot® robot. The press of a button on the remote's keypad opens up an array of new Boe-Bot possibilities. Once you understand how a remote uses infrared to transmit messages to a TV, programming the BASIC Stamp® 2 microcontroller to detect and process these messages is pretty easy. Once these tasks are reduced to subroutines, programming the Boe-Bot to take action based on these messages is a snap.

Here are a few Boe-Bot applications that you will have the opportunity to try in the next three chapters:

- Press and hold keys on the remote's keypad to control your Boe-Bot like a remote-controlled car.
- Send messages to your Boe-Bot while it autonomously roams to change the way it behaves.
- Remotely enable/disable the Boe-Bot program with the power on/off key.
- Tell the Boe-Bot which program to run.
- Remotely program the Boe-Bot with motion sequences.

Along the way, you will learn about pulse width modulation (PWM) for sending electronic messages, the binary number system, the PBASIC **PULSIN** command, and new uses for the **RCTIME** and **SELECT…CASE** commands.

AUDIENCE

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

EDUCATOR RESOURCES

IR Remote for the Boe-Bot has a supplemental set of exercises and solutions in an editable Word document that are made available only to teachers. These materials and other Stamps in Class resources can be obtained by joining the free, private Parallax Educators forum.

Both students and teachers are invited to join the public Parallax Stamps in Class forum, where they can discuss their experiences using *IR Remote for the Boe-Bot* or any other Stamps in Class text in the classroom. Students are encouraged to come here for assistance with working through the projects in the text, and teachers are encouraged to offer support. Parallax staff moderate and participate in this forum.

To join the Stamps in Class forum, go to forums.parallax.com. After joining Stamps in Class, educators may email stampsinclass@parallax.com for instructions to join the Parallax Educators forum. Proof of status as an educator will be required.

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.

> **"*Process Control*", Student Guide, Version 2.0, Parallax Inc., 2005**
> **"*Applied Sensors*", Student Guide, Version 1.3, Parallax Inc., 2003**
> **"*Basic Analog and Digital*", Student Guide, Version 1.3, Parallax Inc., 2004**
> **"*Elements of Digital Logic*", Student Guide, Version 1.0, Parallax Inc., 2003**
> **"*Experiments with Renewable Energy*", Student Guide, Version 1.0, 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 1.0, 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 stampsinclass@parallax.com). If you plan on doing any translations please contact us so we can provide the correctly-formatted MS Word documents, images, etc. We also maintain a 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

Chapter 1: Infrared Remote Communication

GETTING STARTED

The IR Remote AppKit has two documents you can use to get started:

- This book – takes you from beginner to advanced in a step-by-step format.
- IR Remote AppKit Documentation – a quick start guide that comes with the AppKit as a package insert and can be found in this book in Appendix A.

This book is, for the most part, a continuation of *Robotics with the Boe-Bot*. It follows the same format in terms of introducing new hardware, explaining how things work, and demonstrating new PBASIC techniques. By doing the activities, questions, exercises, and projects, you will build your programming, electronics, and robotics skills as you learn about infrared communication and control. The knowledge and skills you will gain will be useful for future robot and/or product designs of your own. NOTE: You will need a fully assembled Parallax Boe-Bot robot to complete the material in this text.

IR Remote AppKit Documentation (Appendix A) summarizes selected activities from this book in a few pages. It's mainly the bare essentials that an intermediate to advanced BASIC Stamp programming enthusiast needs to understand how IR communication works, and how to use the applications developed in this text. You can also find this document as a package insert in the Parallax IR Remote AppKit. NOTE: Unlike this text, a Boe-Bot robot is not required for the activities in the AppKit documentation; you may use your own customized BASIC Stamp project.

> **i**   **This book** and the **IR Remote AppKit Documentation** are both available for free download from www.parallax.com.

KIT CONTENTS

The IR Remote AppKit contains a universal remote similar to the one shown in Figure 1-1. The activities in this chapter will make use of the signals sent by a universal remote after it has been configured to control a SONY® television set. Most universal remotes can be configured to send messages to a SONY TV. If you want to try these activities with a universal remote you already own, or one purchased at a local store, you will need to read the documentation that comes with the remote to find out how to configure it to control a SONY TV.

Infrared Remote Parts List:*

(1) 020-00001   Universal Remote and Universal Remote Manual**
(1) 350-00014   IR detector
(1) 150-02210   Resistor – 220 Ω
(1) 800-00016   Jumper wires – bag of 10

*   Batteries sold separately
**   The remote and instruction sheet/booklet may not be identical to the ones shown Figure 1-1.

Figure 1-1
Contents of IR
Remote Kit

*(batteries not
included)*

In addition to a fully assembled Boe-Bot, there are also some parts that you will need from your original Boe-Bot Robot Kit (see Figure 1-2). The activities in this chapter will make use of the same infrared detection circuit used in Chapters 7 and 8 of *Robotics with the Boe-Bot*. The resistors, LEDs, and piezospeaker should also be familiar from earlier chapters. Only one IR detector is required to capture messages sent by the remote. However, some of the later activities in this chapter will make use of the entire IR detection circuit for autonomous roaming combined with remote control.

Figure 1-2
Parts from Boe-Bot Parts Kit

<u>Parts List:</u>

(2) Infrared detectors

(2) IR LEDs – (clear case)
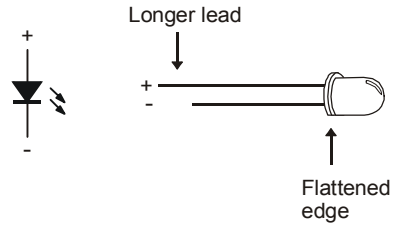
(2) IR LED shield assemblies

(2) Resistors – 220 Ω

(2) Resistors – 1 kΩ

(1) Piezospeaker

(2) LEDs – red

HOW THE REMOTE SENDS MESSAGES

Figure 1-3 shows how the handheld remote can be used to send messages to a Boe-Bot. When a button on the remote is pressed, it flashes its IR LED on/off at 38.5 kHz to broadcast a code for that button. The code is produced by controlling the brief amounts of time the IR LED flashes on and off. The sequence of IR broadcast times shown in the figure corresponds to the beginning of the code that tells a SONY TV the 3 button has been pressed. To send messages to a SONY TV, the remote has to broadcast IR for 2.4 ms signaling that a message is about to start. The message that follows consists of a combination of 1.2 ms (binary-1) and 0.6 ms (binary-0) broadcasts. Notice how the IR detector on the Boe-Bot sends low signals that match the time pattern broadcast by the remote. The BASIC Stamp can then be programmed to detect, measure, record and interpret the durations of low pulses in this pattern to figure out which key on the remote was pressed.

Figure 1-3: Handheld Remote Infrared Messages
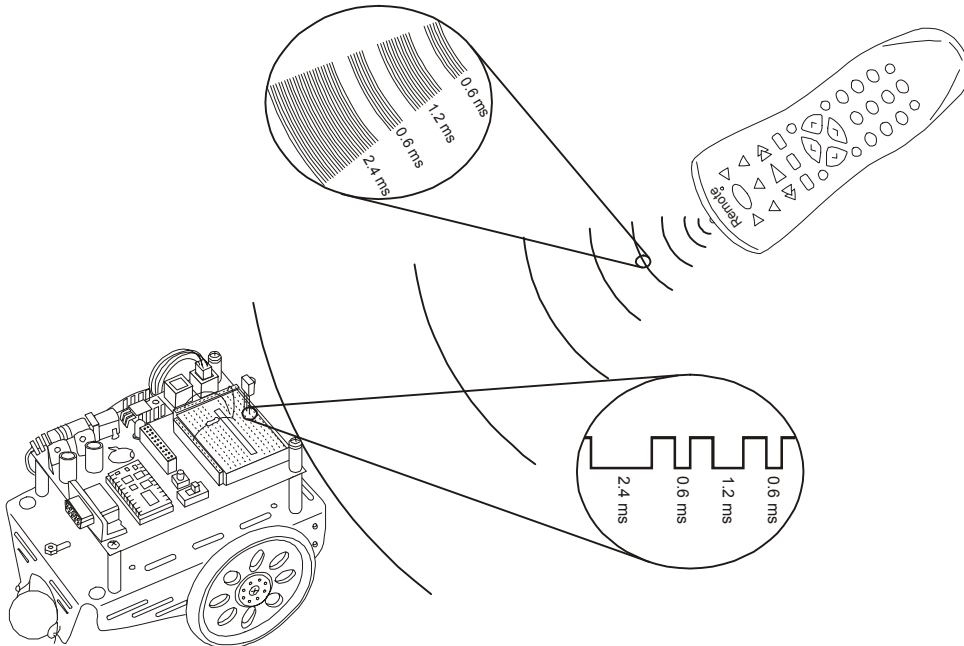
The IR receiver the Boe-Bot used for infrared object detection in *Robotics with the Boe-Bot* is the same detector found in many TVs and VCRs. This detector sends a low signal whenever it detects IR flashing on/off at 38.5 kHz and a high signal the rest of the time. When the IR detector sends low signals, the processor inside a TV or VCR measures how long each of the low signals lasts. Then, it uses these measurements to figure out which key was pressed on the remote. Like the processor inside a TV or VCR, the BASIC Stamp 2 can be programmed to detect, measure, store, and interpret the sequence of low pulses it receives from the same IR detector.

> **Pulse width modulation (PWM):** Pulse durations are used in many applications, a few of which are digital-to-analog conversion, motor control, and communication. Since the IR detector sends low pulses that can be measured to determine what information the IR remote is sending, it's an example of using PWM for communication.
>
> **Carrier signal:** The IR remote uses a 38.5 kHz "carrier signal" to transmit the pulse durations from the remote to the IR detector.
>
> **Communication protocol:** A communication protocol is a set of rules for devices that have to exchange electronic messages. Protocols tend to have rules for voltages, the amount of time signals last, carrier signal frequencies and/or wavelengths, and much more. When two or more devices follow the rules of a given protocol, they should be able to communicate and exchange information.

There are many different communication protocols that a universal remote can use to transmit PWM messages to entertainment system components. This text will focus on the SONY protocol. It is easy to understand, and it works well with the same IR detector that we used for infrared object and distance detection in *Robotics with the Boe-Bot*.

Figure 1-4 shows a timing diagram example for an example signal the IR detector might send to the BASIC Stamp when it receives a SONY TV control message from the IR remote. This message consists of thirteen negative pulses that the BASIC Stamp can easily measure. The first pulse is the start pulse, which lasts for 2.4 ms. The next twelve pulses will either last for 1.2 ms (binary-1) or 0.6 ms (binary-0). The first seven data pulses contain the IR message that indicates which key is pressed. The last five pulses contain a binary value that specifies whether the message is intended being sent to a TV, VCR, CD, DVD player, etc. The pulses are transmitted in LSB-first order. This stands for least significant bit first, meaning the first data pulse is bit-0, the next data pulse is bit-1, and so on. If you press and hold a key on the remote, the same message will be sent over and over again with a 20 to 30 ms rest between messages.

Figure 1-4: IR Message Timing Diagram
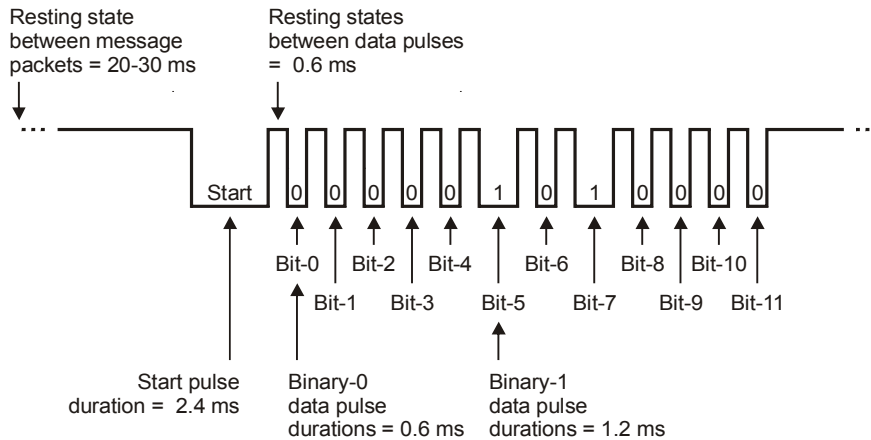
In this chapter, you will start with simple programs to make your BASIC Stamp 2 module interpret and act on the pulse patterns sent by the infrared remote. You will then expand these programs to guide the Boe-Bot with the IR remote.

ACTIVITY #1: CONFIGURING YOUR REMOTE

In this activity, you will program your universal remote so that it sends PWM messages to a television set using the SONY protocol. In this case, the term "programming" means a sequence of key-presses on the remote that tells it to send signals to a SONY TV.

> **If you change the batteries in your remote,** you will probably have to repeat the steps in this activity. Why? Because, when you remove the batteries, the remote will probably forget that it was programmed to be a SONY TV controller.
>
> **Don't throw away the instruction sheet/booklet that comes with your remote!** The batteries for these remotes might last longer than your memory of the code and procedure for entering it. A sticker on the back of the remote with the code and button sequence can also come in really handy.

Infrared Remote Parts

(1) Universal remote
(1) Instruction sheet/booklet for the universal remote
Compatible batteries

How to Configure the Universal Remote to Send SONY TV Protocol Signals

These instructions are for the remote included in the IR Remote AppKit.

- √   Remove the battery compartment cover and determine how many and what kind of batteries to use (AA, AAA, etc).
- √   Load the battery compartment with new (or freshly charged rechargeable) batteries. Do not mix battery types.
- √   Find the TV setup codes section in the instruction sheet/booklet. Here are some examples of titles for that section: "Setup Codes for TV", "Setup Codes for Television", "TV Code List".
- √   Find the code for SONY from the TV code list, and make a note of it.
- √   Find the section that explains how to manually program a TV code into your remote. Here also, are examples of titles for that section: "Programming Your Remote", "To Manually Program Your Remote Control", "Programming for TV".
- √   Follow the instructions in the manual programming section for entering the SONY TV code into your remote.

The instructions might tell you to test it on your TV, but if it's not a SONY, the test probably won't work. In the next activity, you will test to make sure the code was correctly entered by verifying that the signals that the remote transmits have the SONY TV protocol characteristics.

Missing instruction booklets: If you want to try these activities with a remote you already own, you will probably need the instruction booklet for that remote. If it has been misplaced or lost, there may be a copy published on the World Wide Web.

Inexpensive universal remotes can also be purchased from local department stores, and most include instruction booklets. Check the package before you purchase a particular remote to make sure it can be configured to control a SONY TV. If the packaging says something like, "compatible with most/all major brands...", it will most likely work with SONY TVs.

Your Turn – Testing the Remote on Other Devices

√   If you'd like to test it out on your brand of TV/VCR, etc., try following the instructions in the remote manual to see if you can get it to work.

√   **Before moving on to the next activity:** If you followed the instruction in the previous checkmark, make sure to reprogram the remote to control a SONY TV!

ACTIVITY #2: CHARACTERIZING THE IR MESSAGES

This activity focuses on measuring the pulses the Boe-Bot's IR detector sends when it detects messages from the handheld remote. Figure 1-5 shows a timing diagram of the IR detector's output at the beginning of an IR remote message. The IR detector sends a low signal while it detects infrared flashing on/off at 38.5 kHz. While the IR detector does not detect 38.5 kHz infrared, it sends a high signal. Before the message can be interpreted by the BASIC Stamp module, the durations of these low signals have to be measured and stored.
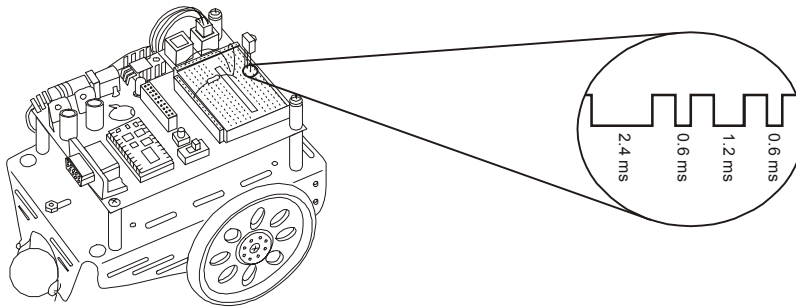
Figure 1-5
IR Detector Output for IR Remote Message

2.4 ms   0.6 ms   1.2 ms   0.6 ms

Infrared Detection Parts

All parts from the previous activity
(1) Infrared detector
(1) Resistor – 220 Ω (red-red-brown)
(2) Jumper wires

The IR Detection Circuit

It only takes one IR detector to capture messages from the IR remote (see Figure 1-6).

√ Build this circuit on your Boe-Bot's prototyping area.

> ℹ **The numbers (1, 2, 3) on the IR detector** are also shown on the pin map in Figure 1-2. You can use this as a guide to make sure the IR detector is correctly wired.
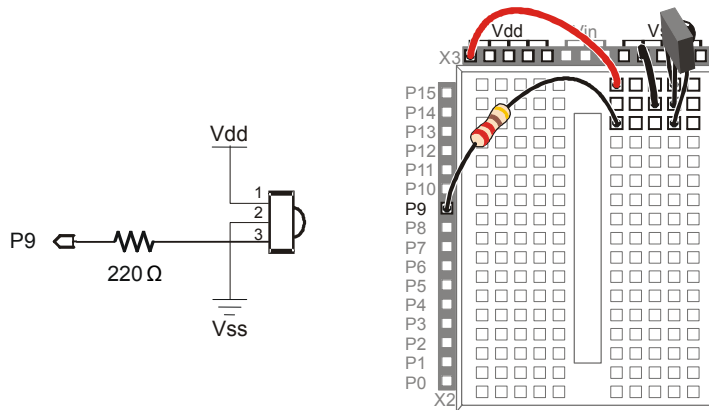
Figure 1-6
IR Detection
Circuit

Measuring Start and Data Pulses

The **PULSOUT** command you've been using to send pulses to the Boe-Bot servos has a complementary command called **PULSIN**. The syntax for the **PULSIN** command is

> PULSIN *Pin, State, Variable*

Pin is, of course, used to select the I/O pin for measuring the pulse. *State* is used to determine whether the pulse is a high pulse (1) or a low pulse (0). *Variable* stores the pulse duration measured by the BASIC Stamp.

---

i

High pulse vs. low pulse: If the voltage a BASIC Stamp I/O pin senses starts low, then goes high for a while before returning to low, that's a **high pulse**. The term **positive pulse** is also commonly used. The **PULSIN** command measures the amount of time the signal is high if the *State* argument is 1.

A **low pulse** is the opposite: the signal will be high, then drop low for a while before returning to the high state. It is also called a **negative pulse**. The **PULSIN** command measures the amount of time the signal is low if the *State* argument is 0.

Let's say your program has a word variable named **time** for storing the measured pulse duration. The 2.4 ms, 1.2 ms, and 0.6 ms pulses shown in Figure 1-7 are negative pulses. To measure them with the IR detector circuit, you will have to use the command:

```
PULSIN 9, 0, time
```
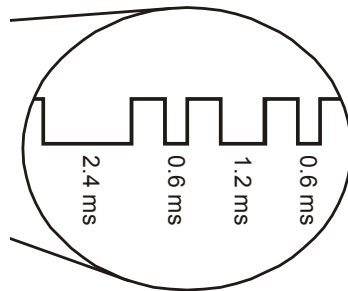

Figure 1-7
A Closer Look at the Pulses

The high time between two low pulses can be measured as a positive pulse. True, it doesn't contain any data, but it can be useful for figuring out how long an entire IR message takes to transmit. By changing the **PULSIN** command's *State* argument from 0 to 1, you can measure the duration of a positive pulse, like this:

```
PULSIN 9, 1, time
```

Example Program: CountStartPulses.bs2

The first pulse that we will examine using the **PULSIN** command is the 2.4 ms start pulse. This pulse won't be exactly 2.4 ms (2400 μs), but it should be fairly close, give or take 250 μs. This is the pulse that signifies that twelve more data pulses are about to be sent by the remote.

CountStartPulses.bs2 counts the number of low pulses that it receives with durations that fall in the 1.95 to 2.85 ms range. Figure 1-8 shows what your Debug Terminal should display after you have pressed a numeric key for a couple of seconds while pointing the remote at the IR detector. The duration shown in the Debug Terminal for a sample start pulse is 2562 μs. Although it's not exactly 2400 μs, it is well within $^+$/- 250 μs.
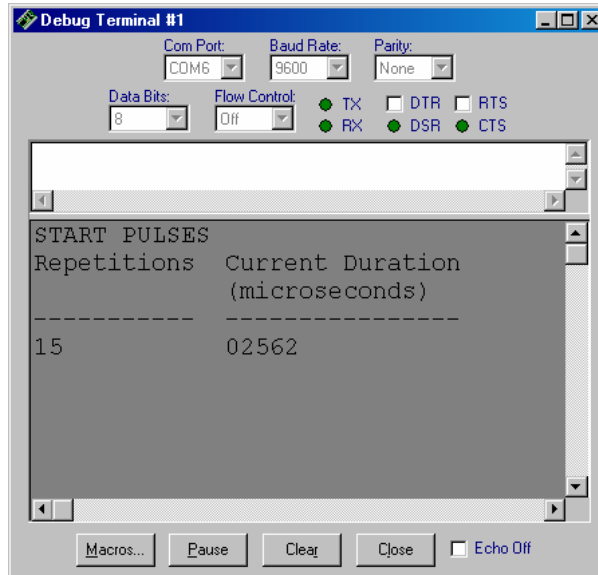
Figure 1-8
Debug Terminal for
CountStartPulses.bs2

√   Enter and run CountStartPulses.bs2.
√   Point the remote at the bubble on the front of the infrared detector.
√   Press and hold one of the numbered keys (0 – 9) on the remote.
√   Make sure pulses in the 2.25 to 2.75 ms range are detected (that's 2250 to 2750 μs displayed in the Debug Terminal).
√   Make a note of the actual value of the start pulse for your remote here: _____. If the value is different each time, make a guess at the average.

> **If the program does not detect the start pulse,** make sure your remote is set to control a SONY TV set. First, press the TV button on the remote, then try beaming your BASIC Stamp another message. If that doesn't work, repeat the steps from Activity #1. Still no luck? Check your IR detector's wiring, then check your program for errors.

> **Do not press CBL, VCR, or TV/VCR.** Your remote may or may not have these buttons. If it does and you press one of those keys, it tells the remote to send messages to a different device, either a VCR or cable box. In either case, the messages the remote sends will use a non-SONY TV protocol. The result will be that the programs in this book will seem to stop working.
>
> **If you accidentally press one of those keys:** just press the TV key to get the remote back to sending (SONY) TV signals.

```
' IR Remote for the Boe-Bot - CountStartPulses.bs2
' Capture and count the number of 2.4 ms (low) start pulses.

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

time               VAR     Word
counter            VAR     Word

DEBUG "START PULSES", CR,
      "Repetitions  Current Duration", CR,
      "             (microseconds)  ", CR,
      "-----------  ----------------"

DO

  PULSIN 9, 0, time

  IF (time > 975) AND (time < 1425) THEN

    counter = counter + 1

    DEBUG CRSRXY, 0, 4,
          DEC counter,
          CRSRXY, 13, 4,
          DEC5 time * 2

  ENDIF

LOOP
```

How CountStartPulses.bs2 Works

This program starts by declaring two word variables, **time** to store the start pulse duration, and **counter** to store the number of start pulses received. Then, a **DEBUG** command adds some column headings for displaying the variables.

```
time                    VAR     Word
counter                 VAR     Word
DEBUG "START PULSES", CR,
      "Repetitions  Current Duration", CR,
      "             (microseconds)  ", CR,
      "-----------  ----------------"
```

Inside the **DO…LOOP**, the program executes the command **PULSIN 9, 0, time**, which measures pulses. Whenever a pulse duration is between 975 (1.95 ms) and 1425 (2.85 ms), the **IF…THEN…ENDIF** code block increments the **counter** variable and displays the **counter** and **time** variables under the **Repetitions** and **Current Duration** headings.

```
DO

  PULSIN 9, 0, time

  IF (time > 975) AND (time < 1425) THEN

    counter = counter + 1

    DEBUG CRSRXY, 0, 4,
          DEC counter,
          CRSRXY, 13, 4,
          DEC5 time * 2

  ENDIF

LOOP
```

> **What's the * 2 in the DEBUG command?** Remember that the **PULSIN** command measures duration in 2 μs units. That's what gets stored in the **time** variable, the number of 2 μs units the **PULSIN** command measured. By multiplying **time** by 2 with the ***** operator, the **DEBUG** command displays the actual number of microseconds for the pulse measurement.

Your Turn – Measuring the Binary-1 and Binary-0 Pulses

By modifying the **DEBUG** and **IF…THEN** statement in the example program, you can capture and display the duration the binary-1 and binary-0 pulses. Here's how to:

√   Save CountStartPulses.bs2 as MeasureBinary1Pulses.bs2.

√   Change

```
DEBUG "START PULSES", CR,
```

to

```
DEBUG "BINARY-1 PULSES", CR,
```

√   Change

```
IF (time > 975) AND (time < 1425) THEN
```

to

```
IF (time > 450) AND (time < 750) THEN
```

√   Save your modified program.
√   Run the program.
√   Press and hold one of the numbered keypad keys until the Debug Terminal displays a pulse duration.
√   Record the pulse duration for binary-1 here _____.
√   Save MeasureBinary1Pulses.bs2 as MeasureBinary0Pulses.bs2.
√   Change

```
DEBUG "BINARY-1 PULSES", CR,
```

to

```
DEBUG "BINARY-0 PULSES", CR,
```

√   Change

```
IF (time > 450) AND (time < 750) THEN
```

to

```
IF (time > 150) AND (time < 450) THEN
```

√   Save your modified program.
√   Run the program and use one of the remote's numbered keypad keys to send messages to your Boe-Bot.
√   Record the pulse duration for binary-0 here _____.

Measuring the Resting State between Messages

When you press and hold a given key on the remote, the remote sends the code for that key, then waits a while and sends it again. CountStartPulses.bs2 also can be modified to search for this resting time between messages. This resting time is shown in Figure 1-9, and it turns out to be an important factor in Boe-Bot navigation.

Resting time between messages

IR message   IR message
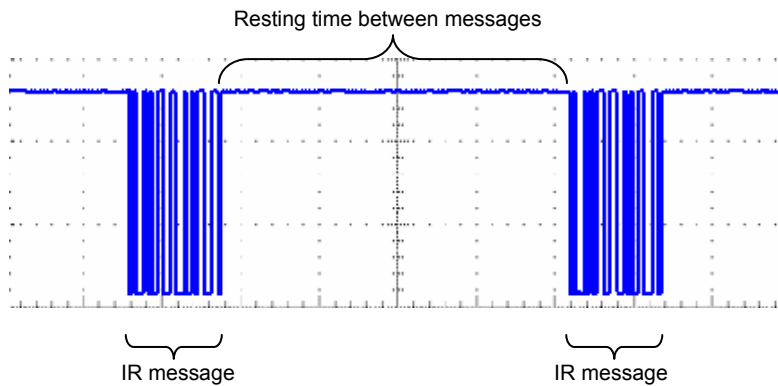
Figure 1-9
Two IR
Messages

*This screen
capture from
the Parallax
USB
Oscilloscope
shows two IR
messages and
the resting
state between
them.*

To measure this resting state with the BASIC Stamp module, all you have to do is think of the high time between messages as a positive pulse with a very long duration. The **PULSIN** command must be modified to search for a positive pulse by changing the *state* argument from 0 to 1. The **IF…THEN** statement also has to be modified so that it takes no action if the measured time is less than 1000 (2 ms). This ensures that the brief high pulses between the start pulse and data bits won't be reported. Instead, only the longer high time between messages will be reported.

Example Program: CountRestingStates.bs2

- √ Enter and run CountRestingStates.bs2.
- √ Point the remote at the IR detector, and press and hold the 5 key.
- √ Make sure high pulses in the 20 to 35 ms range (20,000 to 35,000 µs) are detected.
- √ Make a note of the actual duration of the resting state between messages here: _____.

```
' IR Remote for the Boe-Bot - CountRestingStates.bs2
' Capture and count the number of 20 ms+ (high) resting states.

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

time        VAR     Word
counter     VAR     Word

DEBUG "RESTING STATE", CR,
      "Repetitions  Current Duration", CR,
      "             (microseconds)  ", CR,
      "-----------  ----------------"

DO

  PULSIN 9, 1, time

  IF (time > 1000) THEN

    counter = counter + 1

    DEBUG CRSRXY, 0, 4,
          DEC counter,
          CRSRXY, 13, 4,
          DEC5 time * 2

  ENDIF

LOOP
```

How CountRestingStates.bs2 Works

CountRestingStates.bs2 is just CountStartPulses.bs2 with a few modifications. The first change was just the display heading for the information in the Debug Terminal. The command **DEBUG "START PULSES", CR,** was changed to **DEBUG "RESTING STATE", CR,**. Next, the program has to search for high pulses that last between 20 and 35 ms instead of low pulses that only last 2.4 ms. To accommodate for high pulses instead of low pulses, the command **PULSIN 9, 0, time** was changed to **PULSIN 9, 1, time**. To search for 20-35 ms durations instead of 2.4 ms durations, the condition for the **IF…ENDIF** code block was changed from **IF (time > 975) AND (time < 1425) THEN** to **IF (time > 1000) THEN**.

Your Turn – Measuring the Time between Data Pulses

You can modify the condition for the `IF…ENDIF` code block again, this time to search for the very brief high resting states between data pulses.

√   Save CountRestingStates.bs2 as CountRestingStatesYourTurn.bs2.
√   Change the condition for the `IF…ENDIF` code block from

```
IF (time > 1000) THEN
```

to

```
IF (time > 1) AND (time < 1000) THEN
```

√   Add a `PAUSE 100` command right before the `LOOP` command.
√   Run the program and record the resting state between data pulses here: _____.

IR Message Timing Diagram

Figure 1-10 shows a timing diagram for an IR message from the remote. This diagram shows the timing while the 5 key on its digital keypad is pressed and held. Your task will be to fill in the blanks for each measurement using the numbers you have recorded in this activity. Remember that there are 1000 µs in every 1 ms. Your measurements have been in microseconds, so you will have to divide each by 1000 to enter the millisecond measurements in the timing diagram.

√   Use the measurements you have taken in this activity to fill in the millisecond measurements in Figure 1-10.
√   Add up all the times to calculate the total message time: one resting state between messages, one start pulse, twelve resting states between pulses, twelve data pulses, two of which are binary-1 and ten of which are binary-0.
√   Record your IR message time here: _____.

Resting state between data pulses,
duration = _____ ms
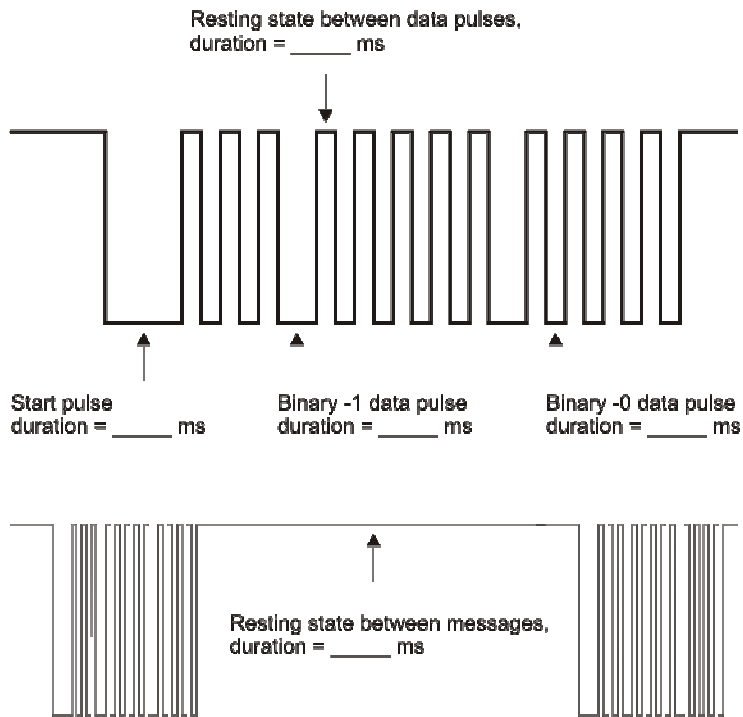
Figure 1-10
IR Message
Timing
Diagram

*Fill in your
measurements
for the times
indicated in
the diagram.*

Start pulse
duration = _____ ms

Binary -1 data pulse
duration = _____ ms

Binary -0 data pulse
duration = _____ ms

Resting state between messages,
duration = _____ ms

Your Turn – Counting the Number of Messages Per Second

By pressing and holding a key on the remote for ten seconds, you can determine how many messages per second the IR remote sends.

- √ Run CountStartPulses.bs2 again; it was the first example program in this activity.
- √ With the aid of a clock or watch, press and hold the 5 key for ten seconds.
- √ Divide the number of messages by 10.
- √ Record the number of messages per second here _____.
- √ You can use this to calculate the time it takes for an IR message and a resting state with this formula:

$$packet\_time = \frac{1}{messages/second}$$

√ Compare this IR message time to the one you calculated by adding up all the IR message's components.

ACTIVITY #3: CAPTURING THE IR MESSAGES

In this activity, you will write programs to measure and capture each pulse in the PWM message sent by the infrared remote. There are a total of twelve data pulses in a given message, and each can be captured and stored. By programming the BASIC Stamp to capture and store these pulses, you will have the key programming ingredient for sending messages to your Boe-Bot with the handheld remote.

Introduction to Array Variables

Twelve pulse measurements have to be stored separately because they are each a unique part of the message. Even so, they are related to each other. They are all pulse measurement values, and they are all going to be examined to determine which key on the remote is being pressed.

The best way to store a group of related values is with a special kind of variable called an array variable. An array variable is a group of variables that all have the same name. Declaring an array variable is similar to declaring most other variables. The only difference is that a number is added in parentheses next to the **Size** argument that tells how many variables to create that will share the same name. Here is the syntax for an array variable declaration:

```
name   VAR   Size(n)
```

Here is an example of an array that can store five bytes:

```
characters VAR Byte(5)
```

Figure 1-11 shows the five **characters** variables created by this declaration: **characters(0)**, **characters(1)**, **characters(2)**, **characters(3)**, and **characters(4)**. Each of these variables, called array elements, can store a byte. Each element has a number called an index (the number inside parentheses) that differentiates it from the other elements. In other words, each array element variable has the same name, but a different index number.

```
characters VAR Byte(5)
```

Five variables named characters, each of which can store a different byte value.

```
characters(0)
characters(1)
characters(2)
characters(3)
characters(4)
```

Figure 1-11
Elements in an Array
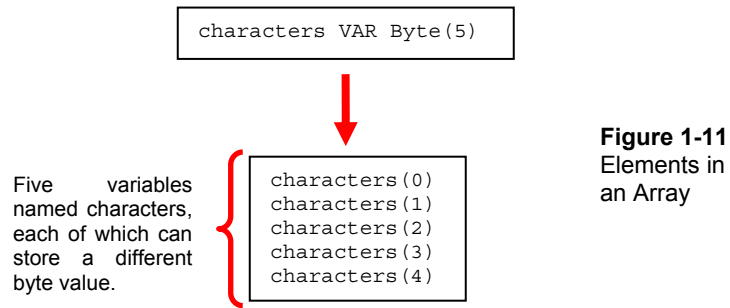
Example Program: ArrayExample.bs2

This example program uses the Debug Terminal for setting array elements and then displaying their contents. Figure 1-12 shows an example of what the Debug Terminal will look like after you run and test the program. Each character you type into the Debug Terminal's Transmit Windowpane is loaded into the next characters array element. After you type the fifth character, the program reads and displays each array element.

Transmit Windowpane

Receive Windowpane
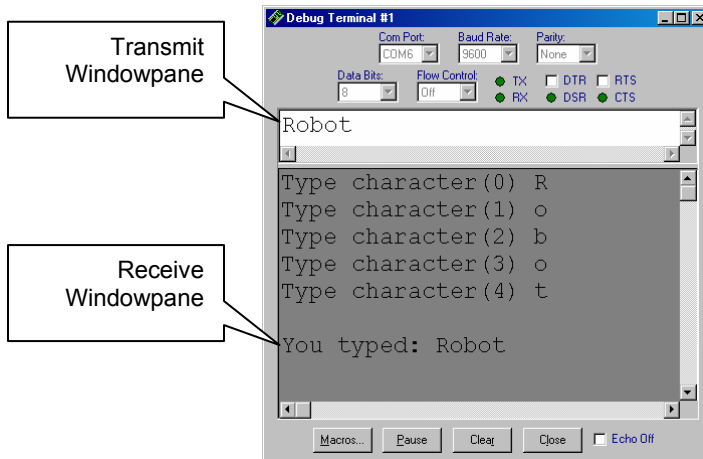
Figure 1-12
Using the Debug Terminal to Type Characters into an Array

Click the Transmit Windowpane, then type five characters.

```
Debug Terminal #1
Com Port:   Baud Rate:   Parity:
COM6        9600         None
Data Bits:  Flow Control:
8           Off          TX   DTR  RTS
                         RX   DSR  CTS

Robot

Type character(0)  R
Type character(1)  o
Type character(2)  b
Type character(3)  o
Type character(4)  t

You typed: Robot

Macros...  Pause  Clear  Close   Echo Off
```

√ Enter and run ArrayExample.bs2.
√ Click the Debug Terminal's Transmit Windowpane.
√ Type five letters.
√ When the Debug Terminal displays "You typed: …", verify that the BASIC Stamp has sent back the same characters you entered.

```
' IR Remote for the Boe-Bot - ArrayExample.bs2
' Set array element values with DEBUGIN and display them with DEBUG.

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

characters      VAR      Byte(5)
index           VAR      Nib

FOR index = 0 TO 4
  DEBUG "Type character(", DEC index, ") "
  DEBUGIN characters(index)
  DEBUG CR
NEXT

DEBUG CR, "You typed: "

FOR index = 0 TO 4
  DEBUG characters(index)
NEXT

END
```

How ArrayExample.bs2 Works

An array of five bytes named `characters` is declared along with a nibble variable named `index`.

```
        characters      VAR      Byte(5)
        index           VAR      Nib
```

A `FOR…NEXT` loop repeats the code block between the `FOR` and `NEXT` statements five times. Each time through, the value of `index` is incremented by 1, so the `DEBUGIN` command stores the character in the next array element.

```
        FOR index = 0 TO 4
          DEBUG "Enter character(", DEC index, ") "
          DEBUGIN characters(index)
          DEBUG CR
        NEXT
```

A second **FOR…NEXT** loop reads and displays the values of each **characters** array element.

```
FOR index = 0 TO 4
  DEBUG characters(index)
NEXT
```

The first time through the **FOR…NEXT** loop, the value of **index** is 0, so **characters**(0) is displayed. The second time through, **index** is 1, so **characters**(1) is displayed, and so on.

Your Turn – Storing Values Instead of Characters

You can use the **DEC** modifier to store values instead of characters. Here's how:

√ Save ArrayExample.bs2 as ArrayExampleYourTurn.bs2.
√ Change

```
DEBUG "Type character(", DEC index, ") "
DEBUGIN characters(index)
DEBUG CR
```

to

```
DEBUG "Type value - character(", DEC index, ") "
DEBUGIN DEC characters(index)
```

√ Change

```
DEBUG characters(index)
```

to

```
DEBUG CR, DEC characters(index)
```

√ Run the modified program and enter values (0 to 255). Make sure to press the Enter key after the last digit in each value you enter.
√ Verify that the program displays the values you stored in each array element.

Capturing the Entire Message

An entire message from the remote has twelve data pulses. Here is how to capture all of their durations:

- Keep executing the **PULSIN** command until the resting state between pulses is detected.
- Use twelve more **PULSIN** commands to capture the next twelve data pulses into an array of word variables.
- Optional – use a **FOR…NEXT** loop to display the elements of the array.

Figure 1-13 shows the array variable declaration **time VAR Word(12)**. The declaration creates twelve different word variables, each of which can store its own value. In the case of the **time** variable array declaration, there's **time(0)**, **time(1)**, **time(2)**, and so on, up through **time(11)**. Each of the elements in the **time** array can store a different value between 0 and 65535.

```
time VAR Word(12)
```

```
time(0)
time(1)
time(2)
time(3)
time(4)
time(5)
time(6)
time(7)
time(8)
time(9)
time(10)
time(11)
```

Twelve variables named time, each of which can store a different word value
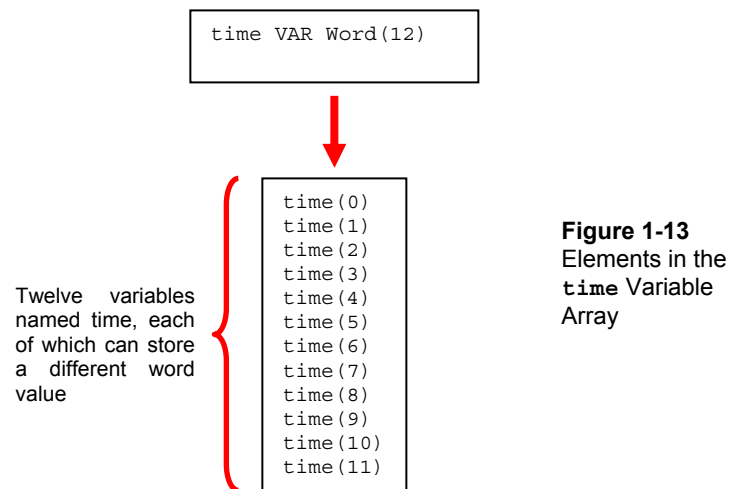
Figure 1-13
Elements in the **time** Variable Array

The next example program will not use a loop to set each of the **time** array element values. Because the time between the pulses sent by the remote is very small, the extra code overhead involved in a **FOR…NEXT** loop could cause the program to miss pulses. So each array element value will be set sequentially with twelve **PULSIN** commands.

```
PULSIN 9, 0, time(0)
PULSIN 9, 0, time(1)
PULSIN 9, 0, time(2)
   •
   •
   •
PULSIN 9, 0, time(11)
```

> **Whenever you see three dots between commands it means there are elements in the sequence that were left out to keep the explanation shorter.** For example, in the sequence of **PULSIN** commands, the
>
> •
> •
> •
>
> indicates that **PULSIN 9, 0, time(3)** through **PULSIN 9, 0, time(10)** were there, but not shown. Three dots **…** are also used to describe commands such as **FOR…NEXT**, **DO…LOOP**, and other statements that have code blocks in between the beginning and ending keywords. The **…** indicates there are one or more commands between the keywords that are not shown.

Example Program: RecordAndDisplayPwm.bs2

This example program measures and displays the durations of all twelve data pulses. Figure 1-14 shows an example of what's displayed when the remote's 5 key is pressed and held. You can use this program to examine the pulse patterns for each key on your remote. Keep in mind that some of the keys are not for TV sets, such as play and pause, which would work with a VCR. These keys do not cause the remote to send messages when it's configured to function as a SONY TV controller.

> **Remember not to press the VCR, TV/VCR, or CBL keys.** If you do by mistake, press the TV key to get back to SONY TV controller mode.

```
time ARRAY
PWM MEASUREMENTS
Element   Duration, 2-us
-------   ---------------
time(0)   370
time(1)   364
time(2)   662
time(3)   364
time(4)   363
time(5)   358
time(6)   351
time(7)   658
time(8)   361
time(9)   362
time(10)  363
time(11)  363
```
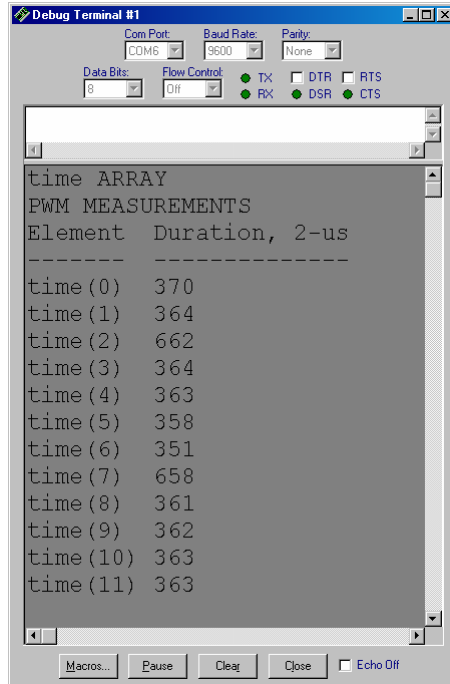
Figure 1-14
Debug Terminal for
RecordAndDisplayPwm.bs2

Pulse pattern for the 5 key.

√ Enter and run RecordAndDisplayPwm.bs2.
√ Resize the Debug Terminal so that it's as tall as your monitor, because otherwise the data might not all fit.
√ Try pressing and holding the following keys and verify that each binary pattern is different for each key: 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, VOL-, VOL+, CH-, CH+, POWER, MUTE, and LAST/PREV. CH. Pay close attention to `time(0)` through `time(6)` and try to detect the pattern of durations for each key.

```
' IR Remote for the Boe-Bot - RecordAndDisplayPwm.bs2
' Measure all data pulses from SONY IR remote set to control a TV.

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

time            VAR     Word(12)                ' SONY TV remote variables.
index           VAR     Nib
                                                ' Display heading.
DEBUG "time ARRAY", CR,
```

```
      "PWM MEASUREMENTS", CR,
      "Element  Duration, 2-us", CR,
      "-------  --------------"

DO                                           ' Beginning of main loop.

  DO                                         ' Wait for rest between messages.
    PULSIN 9, 1, time(0)
  LOOP UNTIL time(0) > 1000

  PULSIN 9, 0, time(0)                       ' Measure/store data pulses.
  PULSIN 9, 0, time(1)
  PULSIN 9, 0, time(2)
  PULSIN 9, 0, time(3)
  PULSIN 9, 0, time(4)
  PULSIN 9, 0, time(5)
  PULSIN 9, 0, time(6)
  PULSIN 9, 0, time(7)
  PULSIN 9, 0, time(8)
  PULSIN 9, 0, time(9)
  PULSIN 9, 0, time(10)
  PULSIN 9, 0, time(11)

  FOR index = 0 TO 11                        ' Display 12 pulse measurements.

    DEBUG CRSRXY, 0, 4 + index, "time(", DEC index, ")",
          CRSRXY, 9, 4 + index, DEC time(index), CLREOL

  NEXT

LOOP                                          ' Repeat main loop.
```

How RecordAndDisplayPwm.bs2 Works

The **time** variable declaration creates an array with twelve word elements. The program also uses a nibble variable named **index**.

```
        time        VAR     Word(12)
        index       VAR     Nib
```

A single **DEBUG** command with lots of text and arguments creates column headings for the information to be displayed.

```
        DEBUG "time ARRAY", CR,
              "PWM MEASUREMENTS", CR,
              "Element  Duration, 2-us", CR,
              "-------  --------------"
```

The code block in the this **DO…LOOP** keeps measuring high pulses until it finds one that's larger than 2 ms, which means it must be the resting state between IR messages.

```
DO
   PULSIN 9, 1, time(0)
LOOP UNTIL time(0) > 1000
```

When the **PULSIN** command finishes measuring the resting pulse and the program is deciding what to do next, the remote is sending the start pulse. This is the perfect time to line up twelve **PULSIN** commands to catch the twelve data pulses. Notice how each is loaded into separate array elements. These twelve **PULSIN** commands also demonstrate how constant values such as 0, 1, 2 through 11 can be used to index elements in an array.

```
PULSIN 9, 0, time(0)
PULSIN 9, 0, time(1)
PULSIN 9, 0, time(2)
   •
   •
   •
PULSIN 9, 0, time(11)
```

Next is an example of another way to index array elements, with a variable. In this case, the variable is **index**, and it is incremented each time through a **FOR…NEXT** loop. The last argument in the **DEBUG** command is **DEC time(index)**. Since the value of **index** increases by one each time through the **FOR…NEXT** loop, the **DEBUG** command displays the value stored by each successive element in the **time** array. The first time through the loop, the **DEBUG** command displays **time(0)**, the second time through, it displays **time(1)**, and so on.

```
FOR index = 0 TO 11

   DEBUG CRSRXY, 0, 4 + index, "time(", DEC index, ")",
         CRSRXY, 9, 4 + index, DEC time(index), CLREOL

NEXT

LOOP
```

The last `LOOP` command sends the program back to the first `DO` command. The code block within this outer `DO…LOOP` repeats over and over again, processing the messages as the remote sends them.

Your Turn – Recording Pulse Patterns for Each Key

The next two activities will feature programs that make navigation decisions based on the pulse durations stored in the `time` array. The first seven `time` array elements contain all the pulse measurements that you'll need to identify the remote's buttons. Table 1-1 has rows for each of the first seven `time` array measurements and columns for each digit key and some of the other buttons. Fill it in so that you can use it as a reference while writing programs in the next two activities.

- √ Press and release each of the keys listed in Table 1-1.
- √ Fill in each key column with the `time` array measurements from the Debug Terminal.

Table 1-1: Time Measurements for Each Key										
Array Element	**Remote Key**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**
time(0)										
time(1)										
time(2)										
time(3)										
time(4)										
time(5)										
time(6)										
Array Element	**Remote Key**									
	VOL-		VOL+		CH-		CH+		ENTER	POWER
time(0)										
time(1)										
time(2)										
time(3)										
time(4)										
time(5)										
time(6)										

ACTIVITY #4: BASIC IR REMOTE BOE-BOT NAVIGATION

One rather fun application is to program the Boe-Bot so that you can control its motion directly with the remote, like a remote-controlled car. In this activity, you will program the Boe-Bot to recognize when you are pressing and holding the 1, 2, 3, or 4 keys, and to perform a different maneuver for each. You can also use the CH+/- and VOL+/- keys.

Infrared Remote Control Parts and Circuit

All parts from Activity #1 and #2
(1) Boe-Bot
(1) Piezospeaker
(misc) Jumper wires

√ Build/rebuild and test the Boe-Bot servo, IR detector, and piezospeaker circuits shown in Figure 1-15 as needed.
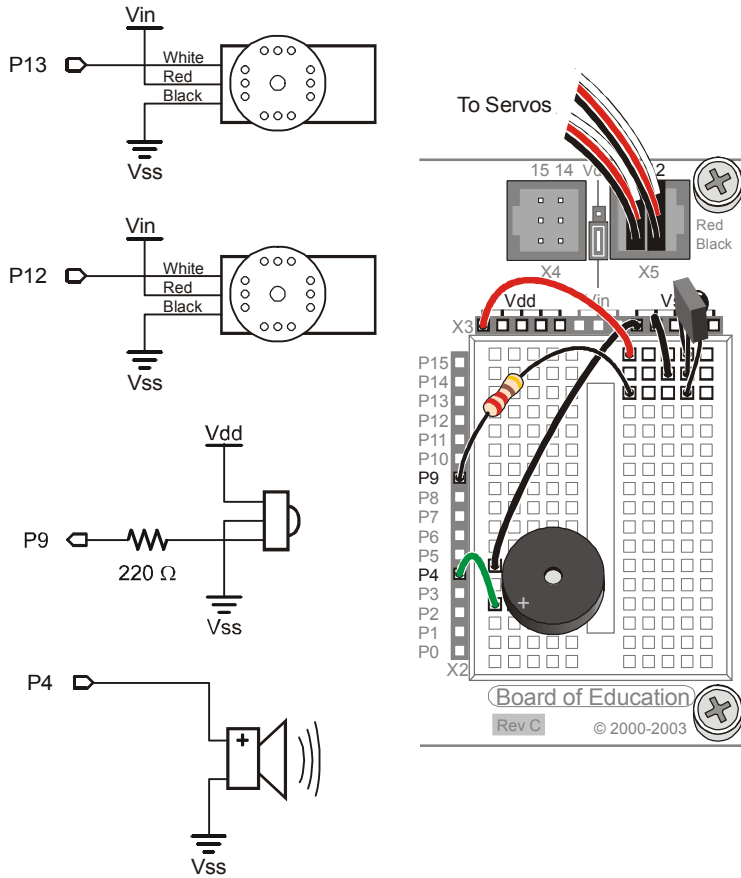
Figure 1-15
Boe-Bot Servo, IR Detector, and Piezospeaker Circuit and Wiring Diagram

The Boe-Bot's left servo should be connected to P13, and its right servo should be connected to P12.

Back to IF…THEN Statements for Navigation

The next example program is a modified version of RecordAndDisplayPwm.bs2. Here's what has to be done to the program to get it to control the Boe-Bot:

√ Remove the **index** variable declaration.
√ Reduce the size of the **time** array to two elements.

√ Add **FREQOUT 4, 2000, 3000** before the first **DO**.
√ Remove all the **DEBUG** commands.
√ Remove all the **PULSIN** commands except the ones that store time measurements into **time(0)** and **time(1)**.
√ Remove the **FOR…NEXT** loop that displays the pulse durations.
√ Use the pulse information in Table 1-1 to make **IF…THEN** statements that choose the **PULSOUT** *Duration* arguments for a maneuver based on four possible combinations of values that **time(0)** and **time(1)** might store.

Go back to Table 1-1 and take a look at the measurements stored in the **time** array for the 1 through 4 keys. Focus on **time(1)** and **time(0)**. If you compare the pulse measurements to 500, there are only four possible combinations:

(1) Both **time(1)** and **time(0)** are less than 500.
(2) **time(1)** is less than 500, but **time(0)** is greater than 500.
(3) **time(1)** is greater than 500, but **time(0)** is less than 500.
(4) Both **time(1)** and **time(0)** are greater than 500.

This is definitely enough information to write an **IF…THEN** statement similar to the ones that guided the Boe-Bot through whisker and IR navigation:

```
IF (time(1) < 500) AND  (time(0) < 500) THEN
  PULSOUT 13, 850
  PULSOUT 12, 650
ELSEIF (time(1) < 500) AND (time(0) > 500) THEN
  PULSOUT 13, 650
  PULSOUT 12, 850
ELSEIF (time(1) > 500) AND (time(0) < 500) THEN
  PULSOUT 13, 850
  PULSOUT 12, 850
ELSEIF (time(1) > 500) AND (time(0) > 500) THEN
  PULSOUT 13, 650
  PULSOUT 12, 650
ENDIF
```

Example Program – 2BitRemoteBoeBot.bs2

You can press and hold the 1, 2, 3, or 4 keys to select from one of four directions: forward, backward, rotate right, rotate left. These directions also work with the CH+, CH-, VOL+, and VOL- keys, which works nicely because those buttons point in the forward, backward, right, and left directions on most remotes.

> ! **These programs are written for Boe-Bots with Parallax Continuous Rotation servos.** If your Boe-Bot is labeled with the letters "PM" highlighted in blue, you will need to use different **PULSOUT *Duration*** arguments in your programs. If you have Parallax PM servos, use 500 in place of 650 and 1000 in place of 850.

√   Enter, save, and run 2BitRemoteBoeBot.bs2.
√   Make sure the power switch on the Board of Education is in position-2.
√   While pointing the remote at the Boe-Bot, press and hold the CH+ button, and verify that the Boe-Bot rolls forward.
√   Repeat the test with the 1 key, it should have the same effect.
√   Test the CH+, 1, CH-, 2, VOL+, 3, VOL-, and 4 keys.
√   Have fun driving the Boe-Bot around.

> ? **For some of the maneuvers, my Boe-Bot seems jittery, why is that?** It has to do with the way the program detects the start of the IR message. It doesn't always get detected on the first try. We'll fix that in the next activity.

```
' IR Remote for the Boe-Bot - 2BitRemoteBoeBot.bs2
' Control your Boe-Bot with an IR remote set to control a SONY TV
' with the 1-4 or CH+/- and VOL+/- keys.

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


time           VAR     Word(2)              ' SONY TV remote variables.
DEBUG "Press and hold a digit key (1-4) or CH+/- and VOL+/-..."

FREQOUT 4, 2000, 3000                       ' Start/reset indicator.

DO                                          ' Beginning of main loop.

  DO                                        ' Wait for rest between messages.
    PULSIN 9, 1, time(0)
  LOOP UNTIL time(0) > 1000

  PULSIN 9, 0, time(0)                      ' Measure/store data pulses.
  PULSIN 9, 0, time(1)

  ' Decide which maneuver to execute depending on the combination
  ' of pulse durations stored in the first two pulse measurements.

  IF (time(1) < 500) AND  (time(0) < 500) THEN
    PULSOUT 13, 850                         ' Forward
    PULSOUT 12, 650
```

```
  ELSEIF (time(1) < 500) AND (time(0) > 500) THEN
    PULSOUT 13, 650                        ' Backward
    PULSOUT 12, 850
  ELSEIF (time(1) > 500) AND (time(0) < 500) THEN
    PULSOUT 13, 850                        ' Right rotate
    PULSOUT 12, 850
  ELSEIF (time(1) > 500) AND (time(0) > 500) THEN
    PULSOUT 13, 650                        ' Left rotate
    PULSOUT 12, 650
  ENDIF

LOOP                                       ' Repeat main loop.
```

Your Turn – Explaining Why CH/VOL and 1-3 Do the Same Thing

√ Go back to Table 1-1 on page 29 and compare the pulse durations in the `time(1)` and `time(0)` rows for 1 and CH+, 2 and CH-, 3 and VOL+, and finally 4 and VOL-.
√ Now, explain why these keys cause the Boe-Bot to perform the same function.
√ Take a look at the `time(4)` column, and explain how a SONY TV can tell the difference between the CH+ and 1 keys.

ACTIVITY #5: ADDING FEATURES TO YOUR SIMPLE IR BOE-BOT

2BitRemoteBoeBot.bs2 can be expanded to perform more maneuvers. Figure 1-16 shows a drawing of the IR remote's keypad with maneuvers assigned to each key. Getting almost all these functions completed is easy, but it takes a little extra work to finish the job. In later activities, you will be introduced to a more universal way to add and remove button/key functions with much less work.
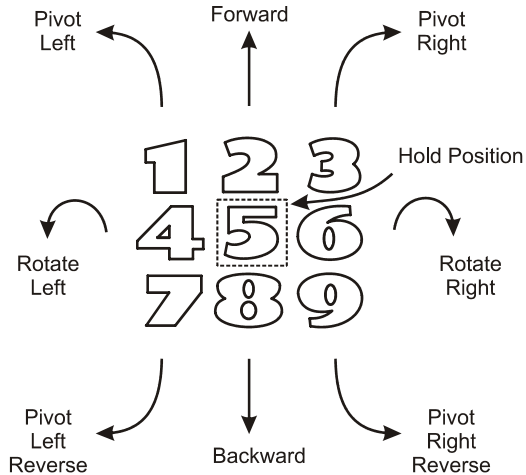
Figure 1-16
Numeric Keypad
Control

One Timing Change and Lots More IF…THEN

There are two changes you will have to make to 2BitRemoteBoeBot.bs2 to get it to accommodate this specification. You may have already guessed the first change; the program will have to make its decisions based on at least three pulses, maybe 4. Take yet another look at Table 1-1 on page 29. If all you are using to make your decisions are **time(1)** and **time(0)**, the program could easily mistake the 5 key for the 1 key. Likewise with the 2 and 6 keys, and so on. To fix this problem, you can use **IF…THEN** reasoning based on three pulses. This will cover keys 1-8. The 9 key is still a problem, but we'll leave that for the Your Turn section.

To measure three data pulses, you will have to make two changes to 2BitRemoteBoeBot.bs2:

√   Modify the **time** variable declaration so that it's a three-word array instead of a two-word array.
√   Add a third **PULSIN** command to capture a third pulse and store it in the **time(2)** variable.
√   Change the Debug Terminal prompt to read "Press and hold a digit key (1-8)…"

Once you've made those changes, here's an **IF…THEN** statement that will get the job done for keys 1-8:

```
IF (time(2) < 500) AND (time(1) < 500) AND (time(0) < 500) THEN
  PULSOUT 13, 750
  PULSOUT 12, 650
ELSEIF (time(2) < 500) AND (time(1) < 500) AND (time(0) > 500) THEN
  PULSOUT 13, 850
  PULSOUT 12, 650
ELSEIF (time(2) < 500) AND (time(1) > 500) AND (time(0) < 500) THEN
  PULSOUT 13, 850
  PULSOUT 12, 750
ELSEIF (time(2) < 500) AND (time(1) > 500) AND (time(0) > 500) THEN
  PULSOUT 13, 650
  PULSOUT 12, 650
ELSEIF (time(2) > 500) AND (time(1) < 500) AND (time(0) < 500) THEN
  PULSOUT 13, 750
  PULSOUT 12, 750
ELSEIF (time(2) > 500) AND (time(1) < 500) AND (time(0) > 500) THEN
  PULSOUT 13, 850
  PULSOUT 12, 850
ELSEIF (time(2) > 500) AND (time(1) > 500) AND (time(0) < 500) THEN
  PULSOUT 13, 750
  PULSOUT 12, 850
ELSEIF (time(2) > 500) AND (time(1) > 500) AND (time(0) > 500) THEN
  PULSOUT 13, 650
  PULSOUT 12, 850
ENDIF
```

Getting familiar with the code block above is useful for getting familiar with counting in binary, which will be introduced in the next chapter. Below is another **IF…THEN** statement you can use that does the same job more efficiently. While the **IF…THEN** statements with three arguments might have to check the value of the **time(2)** variable up to eight times, this one never checks it more than twice.

```
IF (time(2) < 500) THEN
  IF time(1) < 500) AND (time(0) < 500) THEN
    PULSOUT 13, 750
    PULSOUT 12, 650
  ELSEIF (time(1) < 500) AND (time(0) > 500) THEN
      •
      •
      •
  ENDIF
ELSEIF (time(2) > 500) THEN
  IF (time(1) < 500) AND (time(0) < 500) THEN
    PULSOUT 13, 750
    PULSOUT 12, 750
  ELSEIF (time(1) < 500) AND (time(0) > 500) THEN
      •
      •
      •
```

```
        ENDIF
      ENDIF
```

The next change that has to be made is much more subtle, and the reason for the change isn't necessarily all that obvious either. First, here is the change that has to be made:

√   Change the **PULSIN** command in the **DO…LOOP** that scans for the resting state between pulses from this:

```
DO
          PULSIN 9, 1, time(0)
LOOP UNTIL time(0) > 1000
```

to an **RCTIME** command. When you have finished the change, it should look like this:

```
DO
  RCTIME 9, 1, time(0)
LOOP UNTIL time(0) > 1000
```

> ⚠ **Make sure all your programs from this point onward use RCTIME instead of PULSIN to detect the resting time between messages.**

Here's the reason why this change has to be made. The **PULSIN** commands combined with the decision making in the **IF…THEN** statements can end up taking longer than it takes for the IR message to complete. It doesn't right now, but that can change as you add features. The **PULSIN** command needs to detect both the rising and falling edges at the beginning and ending of the resting time shown in Figure 1-17. As soon as the **PULSIN** commands combined with the decision making exceeds the time it takes for the IR message to complete, the **PULSIN** command will miss the rising edge of the resting time between messages.

RCTIME Pin, 1, Variable
Measures from any point on a high
signal until it detects its falling edge.

Falling edge

Rising edge

PULSIN Pin, 1, Variable
Measures from rising to falling edge.

Figure 1-17
PULSIN vs.
RCTIME for
Detecting the
Resting State
Between
Messages
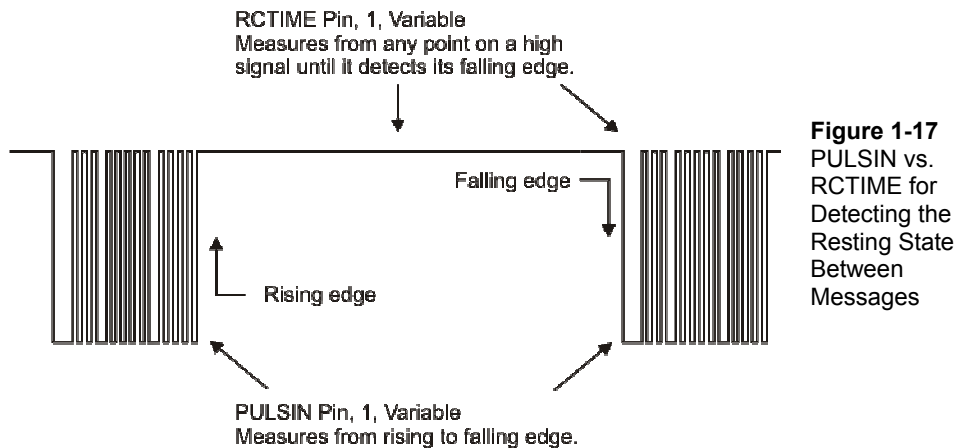
If the **PULSIN** command does not detect the rising edge of the resting state, the loop that searches for a high pulse will keep testing for pulses and waiting for one that's longer than 1000. Eventually, maybe 80 or so ms later, the loop will detect the resting state after the next repeat of the IR message. Finally, at that point, it will move on to the **IF…THEN** and **PULSOUT** commands. This time delay causes a problem for the servos. If the time between servo pulses goes up to 90 ms because the **PULSIN** command can't catch the first rest between messages, the servos won't work well at all.

The solution for this problem is to use **RCTIME** instead. Take a second look at Figure 1-17. While **PULSIN** needs both a rising and falling edge to complete its time measurement, **RCTIME** only needs a falling edge. So, even if the program doesn't start monitoring for the resting state until after it has started, it will still be able to recognize it.

Example Program – 3BitRemoteBoeBot.bs2

This program makes the Boe-Bot respond when you press and hold a given digit key on the remote. The directions for each key are shown in Figure 1-16. The 9 key does not work right, but you will fix that in the Your Turn section. It's better to make sure the 1-8 keys work right before tackling the 9 key.

√   Enter and run 3BitRemoteBoeBot.bs2.
√   Test the keys and have fun driving the Boe-Bot around for a while.

√ Explain the problem with the 9 key using Table 1-1 as a reference.

```
' IR Remote for the Boe-Bot - 3BitRemoteBoeBot.bs2
' Control your Boe-Bot with keys 1-8 on a remote set to control a SONY TV.

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

time            VAR     Word(3)                 ' SONY TV remote variables.

FREQOUT 4, 2000, 3000                           ' Start/reset indicator.
DEBUG "Press and hold a digit key (1-8)..."

DO                                              ' Beginning of main loop.

  DO                                            ' Wait for rest between messages.
    RCTIME 9, 1, time(0)
  LOOP UNTIL time(0) > 1000

  PULSIN 9, 0, time(0)                          ' Measure/store data pulses.
  PULSIN 9, 0, time(1)
  PULSIN 9, 0, time(2)

  ' Decide which maneuver to execute depending on the combination
  ' of pulse durations stored in the first three pulse measurements.

  IF (time(2) < 500) AND (time(1) < 500) AND (time(0) < 500) THEN
    PULSOUT 13, 750
    PULSOUT 12, 650
  ELSEIF (time(2) < 500) AND (time(1) < 500) AND (time(0) > 500) THEN
    PULSOUT 13, 850
    PULSOUT 12, 650
  ELSEIF (time(2) < 500) AND (time(1) > 500) AND (time(0) < 500) THEN
    PULSOUT 13, 850
    PULSOUT 12, 750
  ELSEIF (time(2) < 500) AND (time(1) > 500) AND (time(0) > 500) THEN
    PULSOUT 13, 650
    PULSOUT 12, 650
  ELSEIF (time(2) > 500) AND (time(1) < 500) AND (time(0) < 500) THEN
    PULSOUT 13, 750
    PULSOUT 12, 750
  ELSEIF (time(2) > 500) AND (time(1) < 500) AND (time(0) > 500) THEN
    PULSOUT 13, 850
    PULSOUT 12, 850
  ELSEIF (time(2) > 500) AND (time(1) > 500) AND (time(0) < 500) THEN
    PULSOUT 13, 750
    PULSOUT 12, 850
  ELSEIF (time(2) > 500) AND (time(1) > 500) AND (time(0) > 500) THEN
    PULSOUT 13, 650
    PULSOUT 12, 850
```

```
  ENDIF

LOOP                                       ' Repeat main loop.
```

Your Turn – Fixing the 9 Key

If you refer back to Table 1-1 on page 29, you'll notice that the 9 key has `time(3)` greater than 500 while the rest of the `time` array elements are less than 500. The reason the Boe-Bot pivoted forward and to the left is because the lower three `time` measurements are the same as the 1 key. To fix the problem, you'll have to increase the size of the `time` array by one word and also add another `PULSIN` measurement. Lastly, you'll have to modify the `IF…THEN` statement to catch the 9 key before it misinterprets it as a 1 key.

√   Use Save As from the File menu to save a copy of 3BitRemoteBoeBot.bs2 as 4BitRemoteBoeBot.bs2.
√   Change your `time` array variable declaration so that it sets aside four words named `time` instead of three.
√   Add a `PULSIN` command that measures the fourth data pulse and stores it in `time(3)`.
√   Expand the `IF...THEN` statement by adding a condition to the beginning that checks to see if the fourth data pulse is greater than 500. If it is, then the Boe-Bot should pivot backwards and to the right. You can accomplish this by replacing this line of code:

```
IF (time(2) < 500) AND (time(1) < 500) AND (time(0) < 500) THEN
```

with these four lines:

```
IF (time(3) > 500) THEN
  PULSOUT 13, 650
  PULSOUT 12, 750
ELSEIF (time(2) < 500) AND (time(1) < 500) AND (time(0) < 500) THEN
```

√   Run and test the program, and trouble-shoot as needed.

SUMMARY

This chapter demonstrated how the BASIC Stamp can detect and store messages sent by universal handheld remote controls. The particular signal used to control SONY TV sets was introduced and explained, then measured in detail using an infrared detector circuit Pulse width modulation (PWM) for communication was introduced as the foundation for communication between the handheld remote and entertainment system components. Applications involving direct Boe-Bot remote control demonstrated how to capture and interpret these messages with the help of the **PULSIN** and **RCTIME** commands, and array variables.

Questions

1. What does PWM stand for, and what are its uses?
2. Imagine that the remote's IR LED is flashing on/off at 38.5 kHz. Then, it stops for a while before starting again. What signal does the IR detector send during this rest?
3. What does it mean to "program" a universal remote to send messages to a SONY TV set?
4. What are the **PULSIN** command's three arguments, and what does each do?
5. How do you configure a **PULSIN** command to measure positive pulses? What do you have to do to change a **PULSIN** command that measures positive pulses to make it measure negative pulses?
6. There are several different low pulse durations transmitted by the infrared remote, and each has its own meaning. How can you program the BASIC Stamp to filter for a particular duration and discard the others?
7. Which lasts longer, the IR remote's message or the resting state between messages? Note: Your answer may be correct for SONY TV control, but it might differ for other protocols.
8. What kind of variable works best for storing successive pulse duration measurements? How do you declare this variable?
9. What threshold value was used to distinguish between to make decisions based on data pulse durations?

Exercises

1. Declare an array large enough to hold the string of eight characters.
2. Explain what would be different about Figure 1-14 on page 25 if the 4 key were pressed and held instead of the 5 key.

Projects

1. Write a program that adds up all the IR pulses and displays the duration of an IR message in microseconds. Hints: Start with RecordAndDisplayPwm.bs2. Record and add up all the data pulses first. Then filter for, capture, and add: the resting state between messages, the resting state between data pulses multiplied by twelve, and the start pulse. After you've totaled the data pulses, you can use one `time` array element to store the total, and three more for the other measurements.
2. Modify 4BitRemoteBoeBot.bs2 so that it turns a full-circle every time you press the 5 button.
3. Modify 4BitRemoteBoeBot.bs2 so that the CH+/- and VOL+/- keys work in conjunction with the rest of the keys.

Solutions

Q1. PWM stands for pulse width modulation, and it can be used for digital to analog conversion, motor control, and communication.

Q2. A high signal.

Q3. On page 6, it states: "the term 'programming' means a sequence of key-presses on the remote that tells it to send signals to a SONY TV".

Q4. **Pin** selects the I/O pin that will sense the incoming pulse, **State** configures the **PULSIN** command to measure either positive or negative pulses. **Variable** is a variable that stores the pulse measurement's duration in 2 μs units.

Q5. The **State** argument should be set to 1 to measure positive pulses. It can be changed to 0 if you want to measure negative pulses.

Q6. Use a loop to keep measuring pulses, but only enter a particular code block if the duration falls in a certain range. **IF…THEN** works for this purpose. For example, **IF (time > 975) AND (time < 1425) THEN** filters for pulses that last longer than 1950 μs and shorter than 2850 μs).

Q7. The resting state lasts longer.

Q8. An array variable works best for storing successive, related measurements. Add the number of elements to the variable declaration's **Size** argument in parentheses.

Q9. A value of 500 is convenient for distinguishing between pulses that are typically in the neighborhood of either 300 (0.6 ms) or 600 (1.2 ms).

E1. **characters VAR Byte(8)**.

E2. The **time(2)** array element would probably be 364, while **time(1) and time(0)** would be around 660. This was accomplished with the help of Table 1-1 on page 29.

P1. Start with RecordAndDisplayPwm.bs2 and use various elements in the time array to measure the elements of the message: twelve data pulses, the rest between messages, twelve rests between data pulses, and the start pulse.

```
' IR Remote for the Boe-Bot - Chapter1Project1.bs2
' Add all the pulse durations for total message time.

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

time           VAR     Word(12)         ' SONY TV remote variables
```

```
index          VAR     Nib

DO                                      ' Beginning of main loop

  DO                                    ' Wait for rest between messages
    PULSIN 9, 1, time(0)
  LOOP UNTIL time(0) > 1000

  PULSIN 9, 0, time(0)                  ' Measure/store data pulses
  PULSIN 9, 0, time(1)
  PULSIN 9, 0, time(2)
  PULSIN 9, 0, time(3)
  PULSIN 9, 0, time(4)
  PULSIN 9, 0, time(5)
  PULSIN 9, 0, time(6)
  PULSIN 9, 0, time(7)
  PULSIN 9, 0, time(8)
  PULSIN 9, 0, time(9)
  PULSIN 9, 0, time(10)
  PULSIN 9, 0, time(11)

  FOR index = 1 TO 11                   ' Add up data pulse durations.

    time(0) = time(0) + time(index)

  NEXT

  DO                                    ' Wait for rest between messages
    PULSIN 9, 1, time(1)
  LOOP UNTIL time(1) > 1000

  DO                                    ' Wait for start pulse
    PULSIN 9, 0, time(2)
  LOOP UNTIL (time(2) > 1000)

  DO                                    ' Wait for data pulse rest
    PULSIN 9, 1, time(3)
  LOOP UNTIL (time(3) > 1) AND (time(3) < 1000)

  time(3) = time(3) * 12                ' Twelve data pulse rests

  ' Add time(1) (rest between messages), time(2) (start pulse),
  ' time(3) (twelve rests between messages), and time(0) (the total
  ' data pulses).  This is the message total.

  time(0) = time(0) + time(1) + time(2) + time(3)
```

```
  DEBUG "time(0) = ",
        DEC time(0) * 2,
        " us", CR                       ' Display time in us

LOOP                                     ' Repeat main loop
```

P2. Start with 4BitRemoteBoeBot.bs2.
  √ Add this variable declaration:
```
counter  VAR Byte
```
  √ Delete these two **PULSOUT** commands:
```
PULSOUT 13, 750
PULSOUT 12, 750
```
  √ Replace them with this **FOR…NEXT** loop. You will have to tune the *EndValue*
  of 76.
```
FOR counter = 1 TO 76
  PULSOUT 13, 650
  PULSOUT 12, 650
  PAUSE 20
NEXT
```

P3. Begin with 4BitRemoteBoeBot.bs2.
  √ Increase the time array to 5 elements.
  √ Add a fifth **PULSIN** command to load **time(4).**
  √ Change the **IF** statement to **ELSEIF**.
  √ Before the **ELSEIF** statement (that you just changed from **IF**) add this code:

```
IF (time(4) > 500) THEN
  IF (time(1) < 500) AND  (time(0) < 500) THEN
    PULSOUT 13, 850
    PULSOUT 12, 650
  ELSEIF (time(1) < 500) AND (time(0) > 500) THEN
    PULSOUT 13, 650
    PULSOUT 12, 850
  ELSEIF (time(1) > 500) AND (time(0) < 500) THEN
    PULSOUT 13, 850
    PULSOUT 12, 850
  ELSEIF (time(1) > 500) AND (time(0) > 500) THEN
    PULSOUT 13, 650
    PULSOUT 12, 650
  ENDIF
```

2

Chapter 2: Create and Use Remote Applications

REUSABLE PROGRAMS

In this chapter, you will develop and test two different multi-purpose IR remote application programs. You will test them on their own, and also with new Boe-Bot applications. They will make all kinds of Boe-Bot IR remote programs possible with much less programming. These application programs can also be used with many other projects. Think about all the machines and inventions that have keypads. You can design your own version of these inventions with an infrared detector and a universal remote.

ACTIVITY #1: INTERPRETING THE IR MESSAGES

The key to making the pulse measurements you collected in the previous chapter more useful is decoding. In the case of the messages the universal remote sends, decoding means converting a series of pulse duration measurements into a single value that your PBASIC program can use to make decisions.

> **i**
>
> **Decode**
>
> a: to convert (as a coded message) into intelligible form
> b: to recognize and interpret (an electronic signal)
>
> Source: Merriam-Webster Online Dictionary – www.merriam-webster.com

In this activity, you will write programs that decode the PWM messages sent by the infrared remote. You will also look for relationships between these decoded values and the keys on the remote. If a relationship exists, it will make the remote control programs much simpler. When it's easier to write remote control programs, you will be able to create more powerful Boe-Bot applications with less work.

Counting in Binary vs. Counting in Decimal

Here is how to count to 20 with binary numbers:

Binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011
Decimal	0	1	2	3	4	5	6	7	8	9	10	11

Binary	1100	1101	1110	1111	10000	10001	10010	10011	10100
Decimal	12	13	14	15	16	17	18	19	20

Notice that it takes four or less binary digits to count to 15, but it takes a fifth digit to get to 20.

In binary numbers, digits are usually called bits. Whereas a digit can be 0 through 9 in the decimal number system, a bit can only be 1 or 0 in the binary number system. Figure 2-1 shows a map of the bits in a byte size binary number. The rightmost bit is bit-0, the next one over is bit-1, then bit-2, all the way through bit-7.
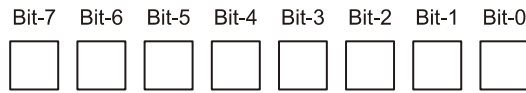
Figure 2-1
The Bits in a
Binary Byte

Bit-7 Bit-6 Bit-5 Bit-4 Bit-3 Bit-2 Bit-1 Bit-0

Each box can hold either a 1 or a 0.

Each bit in a binary number tells you how many of a certain power of 2 the number contains. Bit-0 tells you how many ones are in the binary number, bit-1 tells how many twos, bit-2 tells how many fours, and so on. Even if you have a really large binary number, you can always figure out a given bit value because it's 2^{bit}.

Here is an example of how to take 2 and raise it to the power of its bit position. Bit-0 tells how many ones are in a binary number because $2^0 = 1$. Likewise, bit-1 tells how many twos are in the number because $2^1 = 2$. Bit-2 tells how many fours are in the number because $2^2 = 4$. Bit-6 indicates how many sixty-fours are in a binary number because $2^6 = 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 64$.

Figure 2-2 shows how to convert a binary number with up to 8 bits (a byte) to its corresponding decimal value.

- First, enter the binary number into the upper row of boxes. Make sure to put the rightmost bit into the rightmost box, then continue to the left as you copy numbers.
- Second, multiply each bit by its power of two, and enter the product into the box directly below.
- Third, add up all the product boxes and enter it in the result.
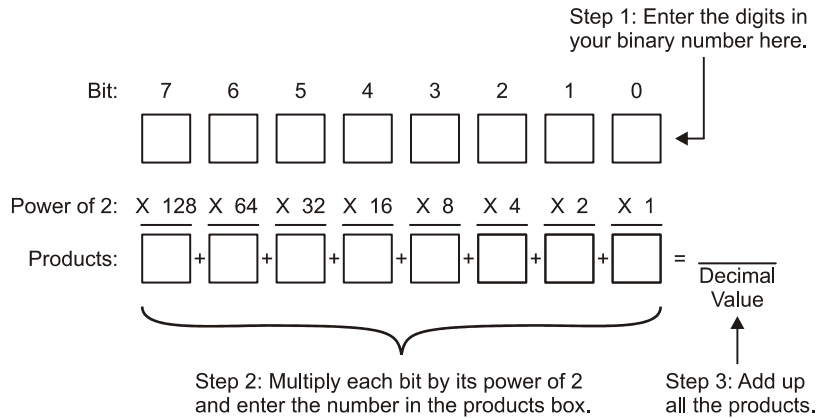
Figure 2-2
Converting Binary to Decimal

Figure 2-3 follows these steps to work out the decimal value of binary 10011.
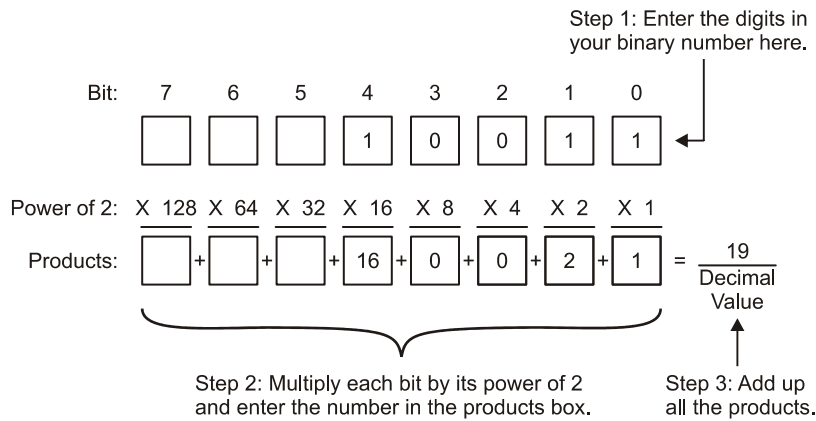
Figure 2-3
Working Out the Decimal Equivalent of Binary-10011

Example Program – BinaryToDecimal.bs2

Although you can write a PBASIC program that follows each of the steps just discussed, the **BIN** and **DEC** modifiers make the job easy.

- √   Enter and run BinaryToDecimal.bs2.
- √   Click the Debug Terminal's Transmit Windowpane.
- √   Type in up to eight binary digits (ones and zeros), then press Enter.
- √   Hand-calculate the value using the steps shown in Figure 2-2.
- √   Compare your hand calculated result to the one in the Debug Terminal result.

```
' IR Remote for the Boe-Bot - BinaryToDecimal.bs2
' Enter a binary value into the Debug Terminal's Transmit Windowpane,
' and get the decimal value in the Receive Windowpane.

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

value           VAR     Byte

DO

  DEBUG "Enter binary value: "
  DEBUGIN BIN value
  DEBUG "Decimal value is: ", DEC value, CR

LOOP
```

Your Turn – Counting in Binary

By entering each of the binary numbers listed beginning on page 46, you will get practice counting in binary. Especially if you delve further into inventions with microcontrollers, the ability to count in binary will be a skill you will rely on over and over again.

- √   While the program is running, enter each of the twenty-one binary values listed beginning on page 46 into the Debug Terminal's Transmit Windowpane.
- √   Make sure that the decimal conversion verifies that you are up-counting correctly.

Setting and Clearing Bits with the .BIT Modifier

Although the **BIN** modifier made it easy to enter binary numbers into the Debug Terminal, it's not very helpful for converting a series of pulses into a binary number. Each pulse corresponds to a different bit in the binary number the remote is transmitting.

2

That means that each pulse has to be translated to a 1 or 0, and then the corresponding bit position in a variable has to be set or cleared.

The **.BIT** variable modifier can be used to set and clear bits in a variable. It does so by making it possible to select individual bits in a given variable.

Let's say that you have a byte variable named **value**, and you want to clear bit-5 and set bit-6. Here's a way to do that with the help of the **.BIT** variable modifier:

```
value.BIT5 = 0
value.BIT6 = 1
```

The next example program demonstrates how to use the **.BIT** modifier to set and clear bits in a byte variable with the help of the Debug Terminal's Transmit and Receive Windowpanes. It uses three variables, a byte named **value**, a nibble named **index,** and a bit named **setClear**. By storing values in **index** and **setClear** with the **DEBUGIN** command, you can pick any bit in the **value** variable and either set it to one or clear it to zero with these commands:

```
DEBUGIN DEC1 index

DEBUGIN BIN1 setClear

IF index = 0 THEN value.BIT0 = setClear
IF index = 1 THEN value.BIT1 = setClear
IF index = 2 THEN value.BIT2 = setClear
IF index = 3 THEN value.BIT3 = setClear
IF index = 4 THEN value.BIT4 = setClear
IF index = 5 THEN value.BIT5 = setClear
IF index = 6 THEN value.BIT6 = setClear
IF index = 7 THEN value.BIT7 = setClear
```

> **Coding tip – the .LOWBIT modifier:** The **.LOWBIT** modifier can be used to treat the bits in a variable as array elements. Here is how you can use the **.LOWBIT** modifier to replace the eight **IF...THEN** statements just discussed.
>
> ```
> value.LOWBIT(index) = setClear
> ```
>
> This technique cannot be used in the upcoming IR remote examples because it would require more calibration with the IR remote pulse measurements. Even so, it's a worthwhile exercise to replace the eight **IF...THEN** statements with this one command in the next example program. Verify that both techniques perform the same operation.

Example Program – SetAndClearWithDotBit.bs2

Figure 2-4 shows an example of what you can do with SetAndClearWithDotBit.bs2. By typing a digit into the Transmit Windowpane when prompted for the "bit index" you can select the bit in the `value` variable that you want to change. Then, by typing a 1 or 0 into the Transmit Windowpane when prompted for "1 to set or 0 to clear", you can control whether the bit in `value` is set to 1 or cleared to 0.
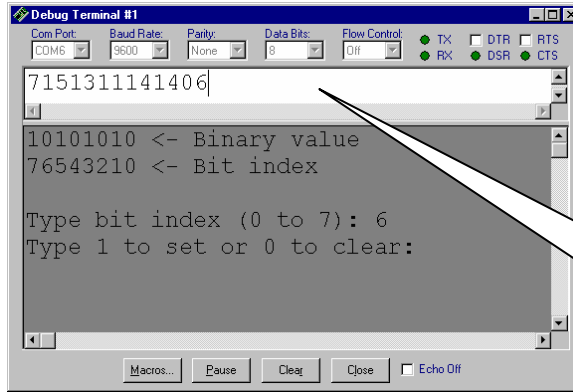
Figure 2-4
Clearing and Setting Bits in a Byte with the Debug Terminal

Use the Transmit Windowpane to reply to the prompts.

√ Enter, save, and run SetAndClearWithDotBit.bs2.
√ When the prompt "Type bit index (0 to 7): " appears, decide which bit you want to change, and type that digit.
√ When the prompt "Type 1 to set or 0 to clear: " appears, type the 1 or 0 key.

The program will pause for half a second, then refresh the display.

√ Check and make sure the change you entered appears in the "Binary value".
√ Experiment with setting and clearing bits in the "Binary value".

```
' IR Remote for the Boe-Bot - SetAndClearWithDotBit.bs2
' Use the Debug Terminal's Transmit Windowpane to choose a bit in
' the value variable and set or clear it.

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


value           VAR     Byte
index           VAR     Nib
setClear        VAR     Bit
```

```
DO

  DEBUG CLS,
        BIN8 value, " <- Binary value", CR,
        "76543210 <- Bit index", CR, CR,
        "Type bit index (0 to 7): "

  DEBUGIN DEC1 index

  DEBUG CR, "Type 1 to set or 0 to clear: "

  DEBUGIN BIN1 setClear

  IF index = 0 THEN value.BIT0 = setClear
  IF index = 1 THEN value.BIT1 = setClear
  IF index = 2 THEN value.BIT2 = setClear
  IF index = 3 THEN value.BIT3 = setClear
  IF index = 4 THEN value.BIT4 = setClear
  IF index = 5 THEN value.BIT5 = setClear
  IF index = 6 THEN value.BIT6 = setClear
  IF index = 7 THEN value.BIT7 = setClear

  PAUSE 500

LOOP
```

How SetAndClearWithDotBit.bs2 Works

The `value` variable is the byte, with 8 bits that you clear and set with the Debug Terminal. The `index` variable stores the value that determines which bit in `value` will be set/cleared, and the `setClear` variable stores a 1 or a 0.

```
        value         VAR     Byte
        index         VAR     Nib
        setClear      VAR     Bit
```

The rest of the commands are nested inside the main DO…LOOP.

A **DEBUG CLS** command clears the Receive Windowpane, displays the 8-bit binary representation of the **value** variable, then displays the index values for each bit, followed by the first prompt for the user (you) to enter a digit.

```
DEBUG CLS,
      BIN8 value, " <- Binary value", CR,
      "76543210 <- Bit index", CR, CR,
      "Type bit index (0 to 7): "
```

The value of **index** is set with a **DEBUGIN** command followed by a second **DEBUG** command, which displays a second prompt. Then a second **DEBUGIN** command gets the value of **setClear** from the Debug Terminal's Transmit Windowpane.

```
DEBUGIN DEC1 index
DEBUG CR, "Type 1 to set or 0 to clear: "
DEBUGIN BIN1 setClear
```

This code block stores the value of **setClear** in the bit you chose in the **value** variable. The series of **IF…THEN** statements examines the **index** variable.   When a match is found, the corresponding bit in **value** is assigned the 1 or 0 you stored in **setClear**.

```
IF index = 0 THEN value.BIT0 = setClear
IF index = 1 THEN value.BIT1 = setClear
IF index = 2 THEN value.BIT2 = setClear
IF index = 3 THEN value.BIT3 = setClear
IF index = 4 THEN value.BIT4 = setClear
IF index = 5 THEN value.BIT5 = setClear
IF index = 6 THEN value.BIT6 = setClear
IF index = 7 THEN value.BIT7 = setClear
```

Next, **PAUSE 500** waits for half a second, then the main **DO…LOOP** repeats. When it does, the screen is cleared, and the new binary representation of the **value** variable is displayed.

Your Turn – Adding Decimal Conversion to the Program

This program can also be a useful tool for examining the relationship between binary and decimal numbers.

   √   Save SetAndClearWithDotBit.bs2 as SetAndClearWithDotBitYourTurn.bs2.

√ Modify the **DEBUG** command so that it looks like this:

```
DEBUG CLS,
      BIN8 value, " <- Binary value", CR,
      "76543210 <- Bit index", CR,
      "Decimal value: ", DEC3 value, CR, CR,
      "Type bit index (0 to 7): ", CR
```

√ Run your modified version of the program. It should now display the decimal equivalent of the binary number whose bits you are setting and clearing.
√ Starting with the rightmost bit (bit-0), set it, note the value, then clear it.
√ Repeat with bit-1, then with bit-2, all the way through bit-7.
√ Explain what you observed.
√ Next, try setting and clearing bits so that you count from 0 to 20 in binary again.

Converting the IR Message Pulse Durations to Decimal Values

The next example program records the first seven (of twelve total) data bit pulses into a byte named **remoteCode**. The program also displays the result stored in **remoteCode** in the Debug Terminal as a decimal value.

While this activity takes a close look at how the example program decodes the IR remote messages, Activity #2 will demonstrate how to incorporate the working code into a subroutine. Once it's in a subroutine, all your program has to do is call the subroutine, and the subroutine will place the result into the **remoteCode** variable.

Figure 2-5 shows a timing diagram of the signal the IR receiver sends to the BASIC Stamp I/O pin as it indicates when it does and does not detect 38 kHz infrared transmitted by IR remote.

Figure 2-5: Command Execution Timing in Relation to Incoming IR Pulses

```
              DO
                RCTIME 9, 1, time
              LOOP UNTIL time > 1000


              RCTIME 9, 0, irPulse
              IF irPulse > 1125 OR irPulse < 675 THEN...


              RCTIME 9, 0, time
              IF time > 300 THEN remoteCode.BIT0 = 1
              RCTIME 9, 0, time
              IF time > 300 THEN remoteCode.BIT1 = 1
              RCTIME 9, 0, time
              IF time > 300 THEN remoteCode.BIT2 = 1
```
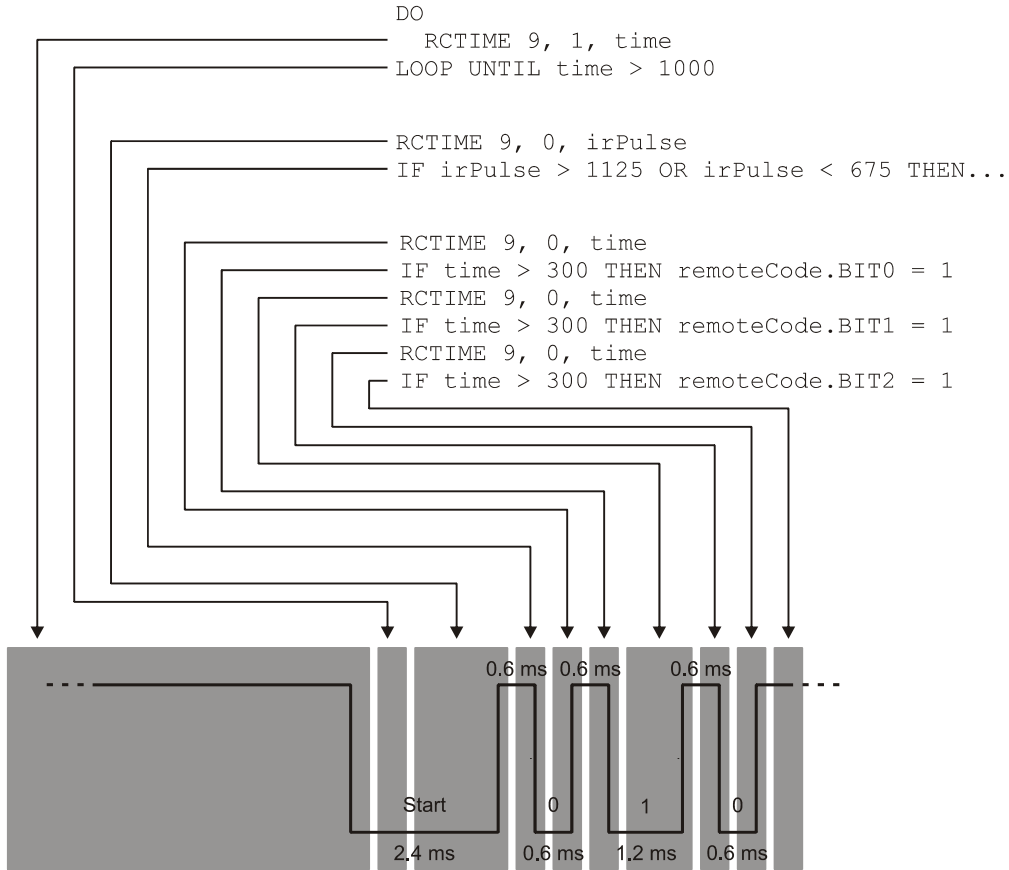
The gray boxes over the timing diagram show the beginning and end of each PBASIC command. Keep in mind that the durations of the low signals are what contain the information. There's a 2.4 ms start pulse, followed by a 0.6 ms data bit 0 pulse. 0.6 ms is a binary-0. The next data bit 1 pulse is low for 1.2 ms, which means it's a binary-1. The bit 2 data pulse is 0.6 ms again, so it is also a binary zero.

2

The first **DO...LOOP** repeats an **RCTIME** command until it detects the end of the 20 to 30 ms high time between messages. Then, another **RCTIME** command measures what remains of the start pulse. An **IF...THEN** statement with two conditions checks to make sure the **RCTIME** measurement is within a range that confirms a start pulse. If it's not in that range, the **IF...THEN** statement sends the program back to the beginning of the routine to try again.

Since none of the **IF...THEN** statements finish until after the next low pulse begins, each one is followed by another **RCTIME** command that measures what's left of the next low pulse. For the data bit pulses, each **RCTIME** command is followed by an **IF...THEN** statement that uses 300 as a threshold to correctly decide whether the time measurement indicates a binary-1 or a binary-0.

Let's try the example program, and then take a closer look at how the **RCTIME** commands and **IF...THEN** statements find and verify the start pulse and decode the data bit pulses.

Example Program: PulsesToByteValue.bs2

This program demonstrates the concepts that have been introduced thus far by decoding the remote's PWM message and displaying it as a decimal value. Any time you need to know what the code for one of the remote keys is, you can use this program. When you know what each code in the remote's keypad is, writing programs that make the Boe-Bot act on the PWM messages it receives using **IF…THEN** and **SELECT…CASE** becomes much easier.

  √ Enter and run PulsesToByteValue.bs2.
  √ Press the various remote keys and find out what each decimal value is.
  √ Test the 1 through 9 keys and explain the relationship between the decimal value displayed in the Debug Terminal and the value of the number on the keypad.
  √ Try the 0 key. Does the value for the zero key present a programming problem?
  √ Compare the binary version of the **remoteCode** variable displayed in the Debug Terminal against your entries in Table 1-1 on page 29. If a given bit in the **remoteCode** variable is 1, the corresponding **time(index)** measurement in Table 1-1 should be above 500. If the bit in **remoteCode** is 0, the corresponding value in the **time(index)** measurement should be below 500.

```
' IR Remote for the Boe-Bot - PulsesToByteValue.bs2
' Display the binary and decimal values of the lower seven bits of
' IR message.

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

time            VAR     Word                ' SONY TV remote variables
remoteCode      VAR     Byte

DEBUG "Binary Value  Decimal Value", CR,    ' Display heading
      "Bit 76543210               ", CR,
      "------------  -------------"
DO                                          ' Beginning of main loop
  Get_Pulses:                               ' Label to restart message check

  remoteCode = 0                            ' Clear all bits in remoteCode

  ' Wait for resting state between messages to end.

  DO
    RCTIME 9, 1, time
  LOOP UNTIL time > 1000

  ' Measure start pulse.  If out of range, then retry at Get_Pulses label.

  RCTIME 9, 0, time
  IF time > 1125 OR time < 675 THEN GOTO Get_Pulses

  ' Get data bit pulses.

  RCTIME 9, 0, time                         ' Measure pulse
  IF time > 300 THEN remoteCode.BIT0 = 1    ' Set (or leave clear) bit-0
  RCTIME 9, 0, time                         ' Measure next pulse
  IF time > 300 THEN remoteCode.BIT1 = 1    ' Set (or leave clear) bit-1
  RCTIME 9, 0, time                         ' etc
  IF time > 300 THEN remoteCode.BIT2 = 1
  RCTIME 9, 0, time
  IF time > 300 THEN remoteCode.BIT3 = 1
  RCTIME 9, 0, time
  IF time > 300 THEN remoteCode.BIT4 = 1
  RCTIME 9, 0, time
  IF time > 300 THEN remoteCode.BIT5 = 1
  RCTIME 9, 0, time
  IF time > 300 THEN remoteCode.BIT6 = 1

  DEBUG CRSRXY, 4, 3, BIN8 remoteCode,      ' Display keypad code
        CRSRXY, 14, 3, DEC2 remoteCode
LOOP                                        ' Repeat main loop
```

2

Advanced Topic – How PulsesToByteValue.bs2 Works

A label named `Get_Pulses` is added at the very beginning of the main `DO...LOOP` in the next example program. The routine in the main `DO...LOOP` captures, records and decodes the data bit pulses from the IR remote. Later in the routine, an `RCTIME` command coupled with an `IF...THEN` statement will check the duration of the start pulse. If the pulse measurement is not the correct duration for a start pulse, it might be interference from a glitch in the IR detector's output, interference that can be generated by some fluorescent lights, or perhaps a nearby remote is being used to control a different brand of TV, VCR, etc. In any of these cases, if the pulse measurement is out of range, a `GOTO` command in the `IF...THEN` statement that checks the pulse measurement will send the program to restart the routine at the `Get_Pulses` label, so that the routine can keep checking for incoming SONY TV protocol IR messages.

```
Get_Pulses:                          ' Label to restart message check
```

PulsesToByteValue.bs2 uses a variable named `remoteCode` to store the result of decoding all the IR message pulses. This simplifies the `IF...THEN` statements that follow each `RCTIME` measurement because they only have to set a bit in the `remoteCode` variable if the pulse duration measurement is greater than 300. If it's less than 300, the bit should be 0, but since all bits in the variable were pre-set to 0, the `IF...THEN` statement doesn't need to take any action. The easiest way to clear all the bits in a variable is by simply setting it equal to zero. If a byte variable stores the value 0, it's really storing the binary number 00000000. In effect, all the bits are set to zero.

```
remoteCode = 0
```

The `DO...LOOP` below was introduced in Chapter 1, Activity #5. It's the first gate keeper in the program that waits for a start pulse from the remote. More specifically, it looks for a signal that stays high for longer than 2 ms before it transitions from high to low. When it finds a signal that stays high for longer than 2 ms, it assumes the 20 to 30 ms resting state between messages has ended, and the start pulse has begun. Either that, or there has been a long period of time between button presses, and finally, after many repetitions of the `DO...LOOP`, it has detected a new message because the person holding the remote has finally decided to press a button. Another function of this loop is to discard the 0.6 ms high times between low data bit pulses and synchronize with the end of the resting state between messages.

```
DO
  RCTIME 9, 1, time
LOOP UNTIL time > 1000
```

The `RCTIME 9, 1, time` command has a *State* argument of 1, and so it measures the time from when it is executed until the signal on P9 transitions from high to low. In contrast, `RCTIME 9, 0, time` has a *State* argument of 0. So, it measures the time from when it is executed until it detects a signal that transitions from low to high. Coupled with the `IF time > 1125 OR time < 675 THEN GOTO Get_Pulses` statement, it's the second gatekeeper in the program, verifying that the end of the resting state between pulses is indeed followed by a start pulse.

```
RCTIME 9, 0, time
IF time > 1125 OR time < 675 THEN GOTO Get_Pulses
```

Since the `LOOP UNTIL time > 1000` condition finishes about 0.6 ms into the 2.4 ms start pulse, the `RCTIME 9, 0, time` command measures the remaining time that the start pulse stays low. Since 1.8 ms of the start pulse should remain, the `RCTIME` command should store a result of about 900 in the time variable. Since pulse durations vary somewhat with brand of universal remote and ambient light levels, the statement `IF time > 1125 OR time < 675 THEN GOTO Get_Pulses` leaves a ± 0.45 ms margin for error. (Remember that one `RCTIME` unit is 2 μs for the BS2.) If the time variable stores a number larger than 1125 or smaller than 675, it certainly can't be the 2.4 ms SONY protocol start pulse, so the `IF...THEN` statement sends the program back to the `Get_Pulses` label at the beginning of the routine to resume searching for the end of the resting state between messages.

By this point in the routine, the program has detected the end of a high signal and verified that it was followed by a low start pulse that lasted somewhere around 2.4 ms. After this, we expect twelve data bit pulses, each lasting 1.2 ms if it's a binary-1, or 0.6 ms if it's a binary zero. The `IF time > 1125 OR time < 675 THEN GOTO Get_Pulses` ends about 0.3 ms into the next data bit pulse. So instead of storing a 600 for a binary-1 or 300 for a binary-0 in the `time` variable, the `RCTIME` command stores 450 for a binary-1 or 150 for a binary-0. The midpoint of 450 and 150 is 300, so the `IF...THEN` statement that follows sets a given bit in the `remoteCode` variable to 1 if it's above 300 (not 450). Otherwise, the bit is left 0 if the time measurement is below 300.

```
RCTIME 9, 0, time                        ' Measure pulse
IF time > 300 THEN remoteCode.BIT0 = 1   ' Set (or leave clear) bit-0
```

It turns out that the processing time difference for an `IF...THEN` statement with two conditions is about the same as the processing time for an `IF...THEN` statement with one condition. So, the `IF...THEN` statements that follow the rest of the `RCTIME` commands can also use 300 as the threshold between a 1 and a 0 pulse.

```
RCTIME 9, 0, time                        ' Measure next pulse
IF time > 300 THEN remoteCode.BIT1 = 1  ' Set (or leave clear) bit-1
RCTIME 9, 0, time                        ' etc.
IF time > 300 THEN remoteCode.BIT2 = 1
RCTIME 9, 0, time
IF time > 300 THEN remoteCode.BIT3 = 1
RCTIME 9, 0, time
IF time > 300 THEN remoteCode.BIT4 = 1
RCTIME 9, 0, time
IF time > 300 THEN remoteCode.BIT5 = 1
RCTIME 9, 0, time
IF time > 300 THEN remoteCode.BIT6 = 1
```

Your Turn – Correcting the Keypad Values

The rules for how numeric values correspond to keys on the keypad can be a little confusing. When **remoteCode** is 0, the key pressed is really 1. When **remoteCode** is 1, the key pressed is really 2, and so on up until **remoteCode** is 8, which means the 9 key was pressed. But wait, when **remoteCode** stores 9, it means the 0 key is pressed!

You can fix this problem with a couple of **IF…THEN** statements. These **IF…THEN** statements can adjust the value stored in the **remoteCode** variable so that it matches the key value on the keypad. In other words, when you press 5, **remoteCode** stores 5. When you press 8, **remoteCode** stores 8. More importantly, when you press 0, **remoteCode** stores 0.

√ Save PulsesToByteValue.bs2 as PulsesToByteValueYourTurn.bs2.
√ Modify the program by inserting these two commands between the last **IF...THEN** statement and the **DEBUG** command.

```
IF (remoteCode < 10) THEN remoteCode = remoteCode + 1
IF (remoteCode = 10) THEN remoteCode = 0
```

√ Run the program and verify that the value stored by **remoteCode** now matches the number of the key pressed on the IR remote's keypad.
√ Write a brief report explaining how these two commands get the job done.
√ Update Table 2-1 below, and mark this page for reference on how the numeric values correspond to the keys pressed on the keypad.

Table 2-1: Remote Key Codes	
Key	**Decimal Value**
0-9	
VOL-	
VOL+	
CH-	
CH+	
ENTER	
POWER	

ACTIVITY #2: DESIGNING A REUSABLE REMOTE PROGRAM

Up to this point, you have completed a program that performs IR remote message capture and decoding for SONY TV signals. Before doing more Boe-Bot applications, it's better to rewrite the program so that all the work is done in subroutines. Along with the subroutines, the program should also have **CON** directives for non-numbered keys and **VAR** declarations for the variables that are used by the subroutines.

Building an Application for Reading the IR Remote

This next program is a more reusable version of PulsesToByteValueYourTurn.bs2. Here are the changes that were made to it:

- Constants – keypad values with meaningful names
  Table 2-1 was used to build a list of **CON** *directives.*
- Variables – variables that have to be used with the subroutines
  The **VAR** *declarations from PulsesToByteValue.bs2 were given their own section.*
- Main routine – call the subroutine and display the data
  A very simple main routine that displays the IR remote code in the Debug Terminal was inserted here along with a comment to add your own code.
- Subroutine – capture message pulses from the infrared remote and decode
  This one should contain everything inside the main the **DO**...**LOOP** *from PulsesToByteValueYourTurn.bs2.*

Example Program: IrRemoteButtons.bs2

√ Enter, save, and run IrRemoteButtons.bs2.

√ Make sure to save this program under the name IrRemoteButtons.bs2.

√ Press and release each digit key on the remote and verify that the correct digit is displayed.

√ Try out the ENTER, CH+, CH-, VOL+, VOL-, and POWER keys. Verify that the displayed value matches the **CON** directives in the program.

```
' -----[ Title ]--------------------------------------------------------
' IR Remote for the Boe-Bot - IrRemoteButtons.bs2
' Capture and store button codes sent by a universal remote configured to
' control a SONY TV.

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

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

' SONY TV IR remote declaration - input received from IR detector

IrDet           PIN     9

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

' SONY TV IR remote constants for non-keypad buttons

Enter           CON     11
ChUp            CON     16
ChDn            CON     17
VolUp           CON     18
VolDn           CON     19
Power           CON     21

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

' SONY TV IR remote variables

irPulse         VAR     Word
remoteCode      VAR     Byte

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

' Replace this DO...LOOP with your own Code.

DO
  GOSUB Get_Ir_Remote_Code
  DEBUG CLS, "Remote code: ", DEC remoteCode
  PAUSE 100
```

```
LOOP


' -----[ Subroutine - Get_Ir_Remote_Code ]---------------------------------

' SONY TV IR remote subroutine loads the remote code into the
' remoteCode variable.

Get_Ir_Remote_Code:

  remoteCode = 0                             ' Clear all bits in remoteCode

  ' Wait for resting state between messages to end.

  DO
    RCTIME IrDet, 1, irPulse
  LOOP UNTIL irPulse > 1000

  ' Measure start pulse.  If out of range, then retry at Get_Ir_Remote_Code.

  RCTIME 9, 0, irPulse
  IF irPulse > 1125 OR irPulse < 675 THEN GOTO Get_Ir_Remote_Code

  ' Get data bit pulses.

  RCTIME IrDet, 0, irPulse                  ' Measure pulse
  IF irPulse > 300 THEN remoteCode.BIT0 = 1  ' Set (or leave clear) bit-0
  RCTIME IrDet, 0, irPulse                  ' Measure next pulse
  IF irPulse > 300 THEN remoteCode.BIT1 = 1  ' Set (or leave clear) bit-1
  RCTIME IrDet, 0, irPulse                   ' etc
  IF irPulse > 300 THEN remoteCode.BIT2 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT3 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT4 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT5 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT6 = 1

  ' Adjust remoteCode so that keypad keys correspond to the value
  ' it stores.

  IF (remoteCode < 10) THEN remoteCode = remoteCode + 1
  IF (remoteCode = 10) THEN remoteCode = 0

  RETURN
```

You will be saving many copies of this file, and then modifying these copies to perform a variety of Boe-Bot functions.

Let's modify a copy of IrRemoteButtons.bs2 to print out descriptions of keys that aren't numeric. To test the remote, the **SELECT…CASE** statement will be used to make sure the program recognizes all the different keys. Remember, **SELECT…CASE** allows you to select a variable and evaluate it on a case by case basis.   You can use single values, a list of values separated by commas, or a range of values.

√   Use the File → Save As menu to save a copy of this program.   Use the name TestIrRemoteButtons.bs2.

√   Replace the existing main routine with this code block:

```
DO

  GOSUB Get_Ir_Remote_Code

  DEBUG CLS, "Remote code: "

  SELECT remoteCode
    CASE 0 TO 9
      DEBUG DEC remoteCode
    CASE Enter
      DEBUG "ENTER"
    CASE ChUp
      DEBUG "CH+"
    CASE ChDn
      DEBUG "CH-"
    CASE VolUp
      DEBUG "VOL+"
    CASE VolDn
      DEBUG "VOL-"
    CASE Power
      DEBUG "POWER"
    CASE ELSE
      DEBUG DEC remoteCode, " (unrecognized)"
  ENDSELECT

  DEBUG CLREOL

LOOP
```

√ Run TestIrRemoteButtons.bs2 and test the remote's keypad keys as well as the other buttons listed in the constant declarations section.

√ Try the MUTE, and LAST/PREV CH keys, what happened? The Your Turn portion of this activity will give you some hints for solving this.

NOTE: Remotes typically use LAST and PREV CH for the same function, to cause the TV to switch to the channel that was previously viewed.

How IrRemoteButtons.bs2 Works

This **PIN** declaration gives a name to the I/O pin that senses the IR detector's output. You can now use the name **IrDet** in place of **IN9**. You can also use it as an argument to a command, so instead of **PULSIN 9, 0, IrPulse**, you can use **PULSIN IrDet, 0, IrPulse**.

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

' SONY TV IR remote declaration - input receives from IR detector

IrDet           PIN     9
```

These constant names can be used in place of the keypad values. This allows you to make decisions on the value from the infrared remote with meaningful names such as **Enter** instead of 11 and **ChUp** instead of 16. For example, if you want to make an **IF…THEN** decision depending on whether or not the ENTER key is pressed, it's much better to use **IF (remoteCode = Enter) THEN** instead of **IF (remoteCode = 11) THEN**.

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

' SONY TV IR remote constants for non-keypad buttons

Enter           CON     11
ChUp            CON     16
ChDn            CON     17
VolUp           CON     18
VolDn           CON     19
Power           CON     21
```

These are the variables you will need for your subroutines. The **time** variable from PulsesToByteValue.bs2 was renamed to **irPulse**.

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

' SONY TV IR remote variables

irPulse          VAR      Word
remoteCode       VAR      Byte
```

Many published PBASIC application examples you will encounter just have a comment, such as **' Insert your code here**, possibly followed by an **END** command. In that case, it will be up to you to look at the comments, subroutines, and other parts of the program and figure out how to make it work. Often there will be a separate program available for download that demonstrates some of the things you can do with the application. This main routine has a comment about inserting code, but it also has a simple **DO…LOOP** that allows you to test it.

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

' Replace this DO...LOOP with your own code.

DO
  GOSUB Get_Ir_Remote_Code
  DEBUG CLS, "Remote code: ", DEC remoteCode
  PAUSE 100
LOOP
```

The code block in the **Get_Ir_Remote_Code** subroutine is mostly the contents of the **DO…LOOP** from PulsesToByteValue.bs2 (from Activity #1). There are two differences. First, it's in a subroutine instead of a **DO…LOOP** in the main routine. Second, this subroutine has two **IF…THEN** statements just before the **RETURN** command that were introduced in the last Your Turn section in Activity #1. They adjust the value the **remoteCode** variable stores so that it matches any keypad digit that gets pressed.

```
' -----[ Subroutine - Get_Ir_Remote_Code ]--------------------------------

' SONY TV IR remote subroutine loads the remote code into the
' remoteCode variable.

Get_Ir_Remote_Code:

  remoteCode = 0                          ' Clear all bits in remoteCode

  ' Wait for resting state between messages to end.

  DO
    RCTIME IrDet, 1, irPulse
  LOOP UNTIL irPulse > 1000

  ' Measure start pulse.  If out of range, then retry at Get_Ir_Remote_Code.

  RCTIME 9, 0, irPulse
  IF irPulse > 1125 OR irPulse < 675 THEN GOTO Get_Ir_Remote_Code

  ' Get data bit pulses.

  RCTIME IrDet, 0, irPulse                ' Measure pulse
  IF irPulse > 300 THEN remoteCode.BIT0 = 1 ' Set (or leave clear) bit-0
  RCTIME IrDet, 0, irPulse                ' Measure next pulse
  IF irPulse > 300 THEN remoteCode.BIT1 = 1 ' Set (or leave clear) bit-1
  RCTIME IrDet, 0, irPulse                ' etc
  IF irPulse > 300 THEN remoteCode.BIT2 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT3 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT4 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT5 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT6 = 1

  ' Adjust remoteCode so that keypad keys correspond to the value
  ' it stores.

  IF (remoteCode < 10) THEN remoteCode = remoteCode + 1
  IF (remoteCode = 10) THEN remoteCode = 0

  RETURN
```

Your Turn – Expanding the List of Known Keys

You can expand your program to include the MUTE, and LAST/PREV CH keys.

√   Re-run TestIrRemoteButtons.bs2 and get the values for MUTE and LAST/PREV CH.

√ Save the program as TestIrRemoteButtonsYourTurn.bs2.

√ Modify the Constants section so that these values are accounted for.

√ Modify the **SELECT…CASE** statement so that it displays "MUTE" when the MUTE key is pressed and "LAST" or "PREV CH" when the LAST (or sometimes PREV CH) key is pressed.

If your remote has VCR control buttons:

VCR control buttons such as >> (FAST FORWARD) and << (REWIND) buttons don't cause the remote to send codes when it's in TV mode. You can use your remote's manual and try programming it with the SONY VCR remote codes. One of them usually works for making the remote speak the same PWM language as the SONY TV controller. Keep in mind that this works with some (but not all) universal remotes.

The VCR button can then be used to enable all the VCR control buttons. Provided you programmed in the right code, most of the TV buttons will still work too. You can use IrRemoteButtons.bs2 to display the values of **remoteCode** for each VCR control button and expand your list of constants (the **CON** directives). The values of the **remoteCode** variable for the VCR control buttons (STOP, PAUSE, PLAY, REWIND, FAST FORWARD, and RECORD) should range between 24 and 29.

ACTIVITY #3: APPLICATION TESTING WITH BOE-BOT NAVIGATION

There are lots of application kits, application notes, and magazine articles published that show how to use BASIC Stamp microcontroller modules with all manner of sensors, actuators, and coprocessors. You can use many of these as resources to add features to your Boe-Bot. Many pre-written example programs in these resources are formatted similarly to the IrRemoteButtons.bs2 application in the activity you just finished. You will often see constant declarations for meaningful numbers, subroutines that do the key jobs, and of course, the variables needed for the application and its subroutines. Other sections that contain **DATA** directives, initialization routines, and revision histories may also appear.

In this activity, you will write a Boe-Bot navigation main routine that uses the features provided by IrRemoteButtons.bs2. By getting familiar with adapting these reusable programs to your Boe-Bot, you can make use of a much wider variety of published resources. Especially when it comes to a new sensor, display, or sound processor, the "hard" work has already been done in the published example program. It will be up to you to adapt it to your specific robotic (or other) application. In many cases, you will take subroutines and their associated constants and variables from several programs and combine them into a master application that controls several subsystems.

A Simple Boe-Bot Main Routine

In the previous activity, you replaced the main routine in IrRemoteButtons.bs2 with a code block that contained a **SELECT…CASE** statement. The best way to remember how **SELECT…CASE** works is that you use it to **SELECT** a (constant, variable or expression) and evaluate it on a **CASE** by **CASE** basis. **SELECT remoteCode** made it possible to evaluate the **remoteCode** variable on a case by case basis, executing different **DEBUG** commands for each different **CASE** (value stored in **remoteCode**).

SELECT…CASE is also really good for Boe-Bot navigation. Instead of **DEBUG** commands, each **CASE** statement can contain code blocks that deliver pulses for different Boe-Bot maneuvers. This can be showcased by repeating the functionality of 4BitRemoteBoeBot.bs2 from Chapter 1, Activity #5. The difference is, this time, it will be really easy to write, and even easier to adjust and expand.

All you have to do with the next example program is open IrRemoteButtons.bs2, save it under a new name, and modify the main routine. The IR remote code reading is taken care of in a subroutine, so you can focus on programming the Boe-Bot, and all you need to know is the value of the **remoteCode** variable after you call the **Get_Ir_Remote_Code** subroutine.

Example Program – 7BitRemoteBoeBot.bs2

This example gives you all the number key features of 4BitRemoteBoeBot.bs2 from Chapter 1, Activity #5, plus the CH+/- and VOL+/- key features from Chapter 1, Activity #4.

- √ Open IrRemoteButtons.bs2 and save it as 7BitRemoteBoeBot.bs2.
- √ Add an initialization section just before the Main Routine:

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

DEBUG "Press and hold a key (1-9 or CH/VOL)..."
FREQOUT 4, 2000, 3000                         ' Start/reset indicator.
```

- √
- √ Replace the **DO…LOOP** in the main routine with this one:

```
DO

  ' Call subroutine that loads the IR message value into the
  ' remoteCode variable.

  GOSUB Get_Ir_Remote_Code

  ' Send PULSOUT durations for the various maneuvers based on
  ' the value of the remoteCode variable.

  SELECT remoteCode
    CASE 2, ChUp                           ' Forward
      PULSOUT 13, 850
      PULSOUT 12, 650
    CASE 4, VolDn                          ' Rotate Right
      PULSOUT 13, 650
      PULSOUT 12, 650
    CASE 6, VolUp                          ' Rotate Left
      PULSOUT 13, 850
      PULSOUT 12, 850
    CASE 8, ChDn                           ' Backward
      PULSOUT 13, 650
      PULSOUT 12, 850
    CASE 1                                 ' Pivot Fwd-left
      PULSOUT 13, 750
      PULSOUT 12, 650
    CASE 3                                 ' Pivot Fwd-right
      PULSOUT 13, 850
      PULSOUT 12, 750
    CASE 7                                 ' Pivot back-left
      PULSOUT 13, 750
      PULSOUT 12, 850
    CASE 9                                 ' Pivot back-right
      PULSOUT 13, 650
      PULSOUT 12, 750
    CASE ELSE                              ' Hold position
      PULSOUT 13, 750
      PULSOUT 12, 750
  ENDSELECT

LOOP
```

A complete copy of the program is below.

√   Run and test the program. Verify that the 1 through 9 keys perform as expected.
√   Try the CH+/- and VOL +/- keys too, and verify that they work correctly.
√   Think about how much easier this was than the approach that was used in Chapter 1, Activities #4 and #5.
√   Save your work, you will also save copies of this program and modify it later.

```
' -----[ Title ]---------------------------------------------------------
' IR Remote for the Boe-Bot - 7BitRemoteBoeBot.bs2

' With an IR remote configured to control a SONY TV, point the remote at
' the Boe-Bot and press and hold the 1-9 keys for different maneuvers.
' You can also use CH+/- and VOL+/-..

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

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

' SONY TV IR remote declaration - input receives from IR detector

IrDet          PIN     9

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

' SONY TV IR remote constants for non-keypad buttons

Enter          CON     11
ChUp           CON     16
ChDn           CON     17
VolUp          CON     18
VolDn          CON     19
Power          CON     21

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

' SONY TV IR remote variables

irPulse        VAR     Word
remoteCode     VAR     Byte

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

DEBUG "Press and hold a key (1-9 or CH/VOL)..."
FREQOUT 4, 2000, 3000                           ' Start/reset indicator.

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

' Boe-Bot button control routine.

DO

  ' Call subroutine that loads the IR message value into the
  ' remoteCode variable.

  GOSUB Get_Ir_Remote_Code

  ' Send PULSOUT durations for the various maneuvers based on
```

2

```
  ' the value of the remoteCode variable.

  SELECT remoteCode
    CASE 2, ChUp                           ' Forward
      PULSOUT 13, 850
      PULSOUT 12, 650
    CASE 4, VolDn                          ' Rotate Left
      PULSOUT 13, 650
      PULSOUT 12, 650
    CASE 6, VolUp                          ' Rotate Right
      PULSOUT 13, 850
      PULSOUT 12, 850
    CASE 8, ChDn                           ' Backward
      PULSOUT 13, 650
      PULSOUT 12, 850
    CASE 1                                 ' Pivot Fwd-left
      PULSOUT 13, 750
      PULSOUT 12, 650
    CASE 3                                 ' Pivot Fwd-right
      PULSOUT 13, 850
      PULSOUT 12, 750
    CASE 7                                 ' Pivot Back-left
      PULSOUT 13, 750
      PULSOUT 12, 850
    CASE 9                                 ' Pivot Back-right
      PULSOUT 13, 650
      PULSOUT 12, 750
    CASE ELSE                              ' Hold Position
      PULSOUT 13, 750
      PULSOUT 12, 750
  ENDSELECT

LOOP

' -----[ Subroutine - Get_Ir_Remote_Code ]----------------------------------

' SONY TV IR remote subroutine loads the remote code into the
' remoteCode variable.

Get_Ir_Remote_Code:

  remoteCode = 0                           ' Clear all bits in remoteCode

  ' Wait for resting state between messages to end.

  DO
    RCTIME IrDet, 1, irPulse
  LOOP UNTIL irPulse > 1000

  ' Measure start pulse.  If out of range, then retry at Get_Ir_Remote_Code.
```

```
RCTIME 9, 0, irPulse
IF irPulse > 1125 OR irPulse < 675 THEN GOTO Get_Ir_Remote_Code

' Get data bit pulses.

RCTIME IrDet, 0, irPulse                      ' Measure pulse
IF irPulse > 300 THEN remoteCode.BIT0 = 1  ' Set (or leave clear) bit-0
RCTIME IrDet, 0, irPulse                      ' Measure next pulse
IF irPulse > 300 THEN remoteCode.BIT1 = 1  ' Set (or leave clear) bit-1
RCTIME IrDet, 0, irPulse                      ' etc
IF irPulse > 300 THEN remoteCode.BIT2 = 1
RCTIME IrDet, 0, irPulse
IF irPulse > 300 THEN remoteCode.BIT3 = 1
RCTIME IrDet, 0, irPulse
IF irPulse > 300 THEN remoteCode.BIT4 = 1
RCTIME IrDet, 0, irPulse
IF irPulse > 300 THEN remoteCode.BIT5 = 1
RCTIME IrDet, 0, irPulse
IF irPulse > 300 THEN remoteCode.BIT6 = 1

' Adjust remoteCode so that keypad keys correspond to the value
' it stores.

IF (remoteCode < 10) THEN remoteCode = remoteCode + 1
IF (remoteCode = 10) THEN remoteCode = 0

RETURN
```

How 7BitRemoteBoeBot.bs2 Works

The **SELECT…CASE** statement inside the main routine's **DO…LOOP** is perfect for delivering **PULSOUT** commands based on the value of the **remoteCode** variable.

```
GOSUB Get_Ir_Remote_Code

SELECT remoteCode
  CASE 2, ChUp                        ' Forward
    PULSOUT 13, 850
    PULSOUT 12, 650
  CASE 4, VolDn                       ' Rotate Left
    PULSOUT 13, 650
    PULSOUT 12, 650
       •
       •
       •
  CASE ELSE                           ' Hold Position
    PULSOUT 13, 750
    PULSOUT 12, 750
ENDSELECT
```

Notice that the **CASE** statements are using constants such as **ChUp** and **VolDn**, from the IrRemoteButtons.bs2 template's constant declarations section:

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

' SONY TV IR remote constants for non-keypad buttons.

Enter          CON     11
ChUp           CON     16
ChDn           CON     17
VolUp          CON     18
VolDn          CON     19
Power          CON     21
```

Your Turn – Adding the POWER Button

You can turn the Power button into a disable switch for the Boe-Bot. In other words, if the Power button is pressed, the Boe-Bot can be programmed to cease to respond to remote commands. In this example, the Reset button on the Board of Education must be pressed and released to restart the program.

- √ Insert a **CASE** statement for POWER with an **END** command into your modified version of 7BitRemoteBoeBot.bs2.
- √ Test it and trouble-shoot as needed.

ACTIVITY #4: ENTERING LARGE NUMBERS WITH THE KEYPAD

Most devices let you enter large numbers by pressing sequences of digit buttons. For example, you might press 3-1-5 on your microwave oven to reheat some food, and the oven cooks the food for three minutes and fifteen seconds. Likewise, you might enter 1-0-0-0 into your calculator as a number (one-thousand) to be multiplied, divided, etc. Many television sets have a menu selection where you can use the keypad on a remote to enter the current time, and VCRs have the same feature so that you can program the VCR to record TV shows when you're not there.

Multi-digit entry is a feature that could be really handy in a second application example that you can add to your library of useful programs. In this activity, you will develop this feature, and then save it for future use and reuse.

The Speaker Circuit

Whether it's a keypad on an alarm system, microwave oven, or a Boe-Bot, a speaker really helps let you know that the device understood when you pressed that digit key. The speaker shown in Figure 2-6 was already added to your Boe-Bot in Chapter 1, Activity #4. Up until now, it was only used to let you know that the program started (or spontaneously restarted due to low batteries).

In this activity, we will also use the piezospeaker circuit to signal that the Boe-Bot detected and understood when you pressed a button on the remote. This is especially useful for typing multi-digit values on the remote's numeric keypad. You will also write commands to send different tones indicating the wrong key was pressed.

√ If you have not already done so, add the speaker circuit shown in Figure 2-6 to your Boe-Bot's prototyping area.
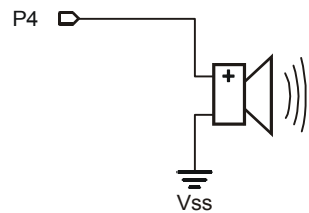
Figure 2-6
Speaker for Keypad
Entry Feedback

Converting Sequences of Digits to Decimal Numbers

Let's say you want to send a larger value to the Boe-Bot, like the decimal value 635. To send this number, you will have to press and release the 6, then the 3, then the 5, then the ENTER key. The BASIC Stamp will have to successively receive each digit, multiply by ten, then add the next most recently received digit. When the ENTER key is pressed, the value 11 is sent, which means the multiply by ten and add routine is finished. Here is a step by step list of how the BASIC Stamp has to process the incoming messages to rebuild the value 635:

- The digit 6 is received, so store 6.
- The digit 3 is received, so multiply 6 by ten, then add 3.
- The digit 5 is received, so multiply 63 by ten, then add 5.
- The value 11 is received, so keep the value 635 and exit the routine.

Example Program: EnterLargeValues.bs2

This example program behaves similarly to a microwave oven with a digital keypad. You can build larger numbers, such as 635 by pressing and releasing the 6, 3, and 5 keys followed by the ENTER key. This program makes the Boe-Bot's piezospeaker send a tone to acknowledge each button press. The speaker also provides debouncing for the pushbutton.

What's "debounce"?

Electronic circuits operate at much higher speeds than human actions and mechanical contacts. Circuit and embedded systems engineers have to take this fact into account when designing circuits. When a switch is closed or pushbutton is pressed, there's a collision between metal contacts. The metal surfaces bounce and scrape against each other before settling down and maintaining contact. This in turn sends a rapid succession of ones and zeros to the processor. A circuit or program routine that makes it impossible for these zeros and ones to confuse the processor is called a **debounced** circuit. Humans also have a tendency to hold down a button or switch for a certain amount of time. Especially when entering the same digit several times, it's important to program the microcontroller to make sure that a person's natural button pressing tendencies are recognized.

√   Open IrRemoteButtons.bs2, then save it as EnterLargeValues.bs2.
√   Add this directive to the I/O Definitions section:

```
Speaker        PIN     4
```

√   Add this variable declaration to the program's Variables section.

```
' Main Routine Variables

value          VAR     Word                ' Stores multi-digit value
```

√   Replace the **DO…LOOP** in the main routine with this one:

```
DO
  DEBUG "Type a value (up to 65535)", CR, "Then press ENTER", CR, CR

  value = 0
  remoteCode = 0
  DEBUG "Digits entered: "

  DO
    value = value * 10 + remoteCode

    GOSUB Get_Ir_Remote_Code

    IF (remoteCode >= 0) AND (remoteCode <= 9) THEN
      DEBUG DEC remoteCode
    ENDIF

    FREQOUT Speaker, 100, 3500
    PAUSE 200
  LOOP UNTIL (remoteCode = Enter)

  DEBUG CR, "The value is: ", DEC value, CR, CR
LOOP
```

√   Follow the Debug Terminal's prompts, and type in a number on your remote as though you are entering a number into a calculator.
√   Press and release the ENTER key.
√   Verify that the value the BASIC Stamp stores is correct.
√   Use the keypad and ENTER key to enter and store numbers with several digits into the BASIC Stamp.

How EnterLargeValues.bs2 Works

This program started as IrRemoteButtons.bs2. Only the I/O Definitions, Variables, and Main Routine sections were modified.

√   In addition to the variables for decoding the IR remote message, a word variable named **value** is declared. This variable stores the multi-digit value entered into the remote. The **Speaker PIN 4** directive, allows you to use **Speaker** in the **FREQOUT** command's *Pin* argument.

```
' SONY TV IR remote variables

irPulse         VAR     Word
remoteCode      VAR     Byte

' Main Routine variables

value           VAR     Word                    ' Stores multi-digit value
```

The first thing the code in the main routine does is clear the **value** and **remoteCode** variables. Then, a **DEBUG** command displays the "Digits entered: " message.

```
value = 0
remoteCode = 0
DEBUG "Digits entered: "
```

The **DO…LOOP** for keypad entry is conditional, and it executes until the remote's ENTER key is pressed. As digits are successively entered, the **value** variable is multiplied by ten, then the most recent digit stored by **remoteCode** is added to value. That's what **value = value * 10 + remoteCode** does. Notice that a **FREQOUT** command and **PAUSE** command follow the **GOSUB Get_Ir_Remote_Code** command. This causes the Boe-Bot to beep after each digit is entered. It beeps and pauses long enough that the user lets go of the key. This prevents one press on the 5 key from being received as the value 555.

```
DO

  value = value * 10 + remoteCode

  GOSUB Get_Ir_Remote_Code

  IF (remoteCode >= 0) AND (remoteCode <= 9) THEN
    DEBUG DEC remoteCode
  ENDIF

  FREQOUT Speaker, 100, 3500
  PAUSE 200

LOOP UNTIL (remoteCode = Enter)
```

After the ENTER key is pressed and released, the **DO…LOOP** terminates, and the multi-digit number stored by the **value** variable is displayed:

```
DEBUG "The value is: ", DEC value, CR, CR
```

Your Turn – Processing Only Digits and Enter

If you were to design a product with a BASIC Stamp and a universal remote, you might get some complaints from customers if you use EnterLargeValues.bs2 without some extra work. The program needs to ignore non-digit keys such as POWER and VOL+. Pressing these keys can lead to some pretty crazy values.

- √ Try entering non-digits with your unmodified version of EnterLargeValues.bs2.
- √ Save the program as EnterLargeValuesYourTurn.bs2.
- √ Replace the **DO…LOOP** in the main routine with this one:

```
DO

  DEBUG "Type a value (up to 65535)", CR, "Then press ENTER", CR, CR

  value = 0
  remoteCode = 0

  DO

    value = value * 10 + remoteCode

    DO
      GOSUB Get_Ir_Remote_Code
      IF (remoteCode < 10) THEN
        DEBUG "You pressed: ", DEC1 remoteCode, CR
        GOSUB Beep_Valid
        EXIT
      ELSEIF (remoteCode = Enter) THEN
        DEBUG "You pressed: ENTER", CR
        GOSUB Beep_Valid
        EXIT
      ELSE
        DEBUG "Press 0-9 or ENTER", CR
        GOSUB Beep_Error
      ENDIF
    LOOP

  LOOP UNTIL (remoteCode = Enter)

  DEBUG CR, "The value is: ", DEC value, CR, CR

LOOP
```

√   Add these two subroutines to the end of your program:

```
' -----[ Subroutine - Beep_Valid ]-----------------------------------

' Call this subroutine to acknowledge a key press.

Beep_Valid:

  FREQOUT Speaker, 100, 3500
  PAUSE 200

  RETURN

' -----[ Subroutine - Beep_Error ]-----------------------------------

' Call this subroutine to reject a key press.

Beep_Error:

  FREQOUT Speaker, 100, 3000
  PAUSE 200

  RETURN
```

√   Test this solution and verify that it works.
√   Save your work.

Adding a Keypad Entry Feature to Your Application

By copying the functional part of EnterLargeValues.bs2 into a subroutine in
IrRemoteButtons.bs2, you will have a new application program that can either read single
remote button presses or receive large values with keypad entry. Which function the
application performs depends on which subroutine is called.

Example Program – IrRemoteKeypad.bs2

√   Open EnterLargeValuesYourTurn.bs2 and save it as IrRemoteKeypad.bs2.
√   Add a subroutine named **Get_Multi_Digit_Value** to IrRemoteKeypad.bs2.

√ Use Edit → Copy and Edit → Paste to move everything shown below from EnterLargeValuesYourTurn.bs2 into your new **Get_Multi_Digit_Value** subroutine in IrRemoteKeypad.bs2.

```
value = 0
remoteCode = 0

DO

  value = value * 10 + remoteCode

  DO
    GOSUB Get_Ir_Remote_Code
    IF (remoteCode < 10) THEN
      DEBUG "You pressed: ", DEC1 remoteCode, CR
      GOSUB Beep_Valid
      EXIT
    ELSEIF (remoteCode = Enter) THEN
      DEBUG "You pressed: ENTER", CR
      GOSUB Beep_Valid
      EXIT
    ELSE
      DEBUG "Press 0-9 or ENTER", CR
      GOSUB Beep_Error
    ENDIF
  LOOP

LOOP UNTIL (remoteCode = Enter)
```

√ Add a **RETURN** command at the end of the subroutine.
√ Modify the main routine so that it looks like this:

```
' Replace this DO...LOOP with your own code.

DO
  DEBUG "Enter a value: " ,CR
  GOSUB Get_Multi_Digit_Value
  DEBUG "The value is: ", DEC value, CR, CR
LOOP
```

√ Add comments to your program explaining how the **DO…LOOP UNTIL** statement works.

After you have made these changes, your program should resemble the one below.

√ Run and test IrRemoteKeypad.bs2.
√ Trouble-shoot as needed.

√   Save the program after you have fully tested it.

```
' -----[ Title ]----------------------------------------------------------
' IR Remote for the Boe-Bot - IrRemoteKeypad.bs2
' Capture and store button codes sent by a universal remote configured to
' control a SONY TV.  This program also supports keypad entry of
' multi-digit values.

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

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

' SONY TV IR remote declaration - input receives from IR detector

IrDet           PIN     9
Speaker         PIN     4

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

' SONY TV IR remote constants for non-keypad buttons

Enter           CON     11
ChUp            CON     16
ChDn            CON     17
VolUp           CON     18
VolDn           CON     19
Power           CON     21

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

' SONY TV IR remote variables

irPulse         VAR     Word                    ' Single-digit remote variables
remoteCode      VAR     Byte
index           VAR     Nib
value           VAR     Word                    ' Stores multi-digit value

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

' Replace this DO...LOOP with your own code.

DO
  DEBUG "Enter a value: ", CR
  GOSUB Get_Multi_Digit_Value
  DEBUG "The value is: ", DEC value, CR, CR
LOOP

' -----[ Subroutine - Get_Ir_Remote_Code ]--------------------------------
```

```
' SONY TV IR remote subroutine loads the remote code into the
' remoteCode variable.

Get_Ir_Remote_Code:

  remoteCode = 0                              ' Clear all bits in remoteCode

  ' Wait for resting state between messages to end.

  DO
    RCTIME IrDet, 1, irPulse
  LOOP UNTIL irPulse > 1000

  ' Measure start pulse.  If out of range, then retry at Get_Ir_Remote_Code.

  RCTIME 9, 0, irPulse
  IF irPulse > 1125 OR irPulse < 675 THEN GOTO Get_Ir_Remote_Code

  ' Get data bit pulses.

  RCTIME IrDet, 0, irPulse                    ' Measure pulse
  IF irPulse > 300 THEN remoteCode.BIT0 = 1   ' Set (or leave clear) bit-0
  RCTIME IrDet, 0, irPulse                    ' Measure next pulse
  IF irPulse > 300 THEN remoteCode.BIT1 = 1   ' Set (or leave clear) bit-1
  RCTIME IrDet, 0, irPulse                    ' etc
  IF irPulse > 300 THEN remoteCode.BIT2 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT3 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT4 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT5 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT6 = 1

  ' Adjust remoteCode so that keypad keys correspond to the value
  ' it stores.

  IF (remoteCode < 10) THEN remoteCode = remoteCode + 1
  IF (remoteCode = 10) THEN remoteCode = 0

  RETURN

' -----[ Subroutine - Get_Multi_Digit_Value ]------------------------------

' Acquire multi-digit value (up to 65535) and store it in
' the value variable.  Speaker beeps each time a key is
' pressed.

Get_Multi_Digit_Value:
```

```
  value = 0
  remoteCode = 0

  DO

    value = value * 10 + remoteCode

    DO
      GOSUB Get_Ir_Remote_Code
      IF (remoteCode < 10) THEN
        DEBUG "You pressed: ", DEC1 remoteCode, CR
        GOSUB Beep_Valid
        EXIT
      ELSEIF (remoteCode = Enter) THEN
        DEBUG "You pressed: ENTER", CR
        GOSUB Beep_Valid
        EXIT
      ELSE
        DEBUG "Press 0-9 or ENTER", CR
        GOSUB Beep_Error
      ENDIF
    LOOP

  LOOP UNTIL (remoteCode = Enter)

  RETURN

' -----[ Subroutine - Beep_Valid ]------------------------------------

' Call this subroutine to acknowledge a key press.

Beep_Valid:

  FREQOUT Speaker, 100, 3500
  PAUSE 200

  RETURN

' -----[ Subroutine - Beep_Error ]------------------------------------
' Call this subroutine to reject a key press.

Beep_Error:

  FREQOUT Speaker, 100, 3000
  PAUSE 200

  RETURN
```

Your Turn – Making Application Backup Copies

You will use and re-use the application programs you have developed in Activity #2 through Activity #4.

√ Create a separate folder for your application programs.
√ Make backup copies of IrRemoteButtons.bs2, 7BitRemoteBoeBot.bs2, and IrRemoteKeypad.bs2.

ACTIVITY #5: KEYPAD BOE-BOT DIRECTION AND DISTANCE

In this activity, you will program the Boe-Bot to receive direction and distance information from the infrared remote. Here is how the program will work:

- Press/release a CH or VOL key to select one of four maneuvers: forward, backward, rotate left, or rotate right.
- Next, use the keypad to enter the number of pulses to deliver.
- When the ENTER button is pressed/released, the Boe-Bot executes the maneuver.

By writing a Boe-Bot main routine for IrRemoteKeypad.bs2, you can make use of its button code and keypad entry subroutine features. This will make writing a main routine that allows you to choose the direction with CH and VOL keys and then enter a number of pulses with the keypad much easier to develop. You will have to call the **Get_Ir_Remote_Code** subroutine to get the CH/VOL key for direction. After that, you can call the **Get_Multi_Digit_Value** subroutine to get the number of pulses.

Example Program – KeypadDirectionDistance.bs2

√ Open IrRemoteKeypad.bs2 and save it as KeypadDirectionDistance.bs2.
√ Add these **PIN** directives:

```
' Boe-Bot Servo Pins

ServoLeft      PIN     13
ServoRight     PIN     12
```

√ Add these declarations to the Variables section.

```
' Boe-Bot navigation variables

direction      VAR     Byte
counter        VAR     Byte
```

√ Add the initialization routine for the Boe-Bot's start/reset indicator along with some operation instructions for the Debug Terminal. This section should be placed just before the Main Routine section.

2

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

DEBUG "Program Starting...", CR, CR          ' Start/reset indicator.

FREQOUT Speaker, 2000, 3000

DEBUG "Use CH/VOL for direction,", CR,        ' Debug instructions.
      "then type distance (up to ", CR,
      "255) then press ENTER.", CR, CR
```

√ Replace the code in the Main Routine section with this:

```
DO

  DEBUG "Select direction (CH/VOL):", CR

  DO

    GOSUB Get_Ir_Remote_Code

    IF (remoteCode < ChUp) OR (remoteCode > VolDn) THEN
      DEBUG "Select direction (CH/VOL):", CR
      GOSUB Beep_Error
    ENDIF

  LOOP UNTIL (remoteCode >= ChUp) AND (remoteCode <= VolDn)

  direction = remoteCode

  GOSUB Beep_Valid

  DEBUG "Enter number of pulses: ", CR

  GOSUB Get_Multi_Digit_Value

  DEBUG "The value is: ", DEC value, CR
  DEBUG "Running...", CR, CR

  FOR counter = 1 TO value

    SELECT direction
      CASE ChUp
        PULSOUT ServoLeft, 850
        PULSOUT ServoRight, 650
      CASE ChDn
        PULSOUT ServoLeft, 650
```

```
            PULSOUT ServoRight, 850
          CASE VolUp
            PULSOUT ServoLeft, 850
            PULSOUT ServoRight, 850
          CASE VolDn
            PULSOUT ServoLeft, 650
            PULSOUT ServoRight, 650
        ENDSELECT

        PAUSE 20

      NEXT

    LOOP
```

A complete program listing is included after these checklist instructions.

- √ Save then run your modified program.
- √ Make sure your Boe-Bot's 3-position switch is set to position-2.
- √ Press/release the CH+ key to select forward.
- √ Press/release the digits 1, 6, 2.
- √ Press/release the ENTER button.

The Boe-Bot should travel forward for about a yard (or meter if you're thinking in metric).

- √ Press/release the VOL+ key to select rotate right.
- √ Press/release the digits 2, 0.
- √ Press/release the ENTER button.

The Boe-Bot should turn roughly 90-degrees clockwise (to the right).

- √ For a review of pulses and distances, see *Robotics with the Boe-Bot* Chapter 4.
- √ Practice entering directions and distances until you are fairly confident with your navigation.

```
' -----[ Title ]------------------------------------------------------
' IR Remote for the Boe-Bot - KeypadDirectionDistance.bs2
' Each Boe-Bot maneuver involves three steps:
' 1) Select a maneuver
'    CH+ = Forward, CH- = Backward, VOL+ = Right, VOL- = Left
' 2) Type in a distance (1 to 255) pulses.
' 3) Press ENTER.
```

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

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

' SONY TV IR remote declaration - input receives from IR detector

IrDet          PIN     9
Speaker        PIN     4

' Boe-Bot Servo Pins

ServoLeft      PIN     13
ServoRight     PIN     12

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

' SONY TV IR remote constants for non-keypad buttons.

Enter          CON     11
ChUp           CON     16
ChDn           CON     17
VolUp          CON     18
VolDn          CON     19
Power          CON     21

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

' SONY TV IR remote variables

irPulse        VAR     Word                ' Single-digit remote variables
remoteCode     VAR     Byte
value          VAR     Word                ' Stores multi-digit value

' Boe-Bot navigation variables

direction      VAR     Byte
counter        VAR     Byte

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

DEBUG "Program Starting...", CR, CR         ' Start/reset indicator.

FREQOUT Speaker, 2000, 3000

DEBUG "Use CH/VOL for direction,", CR,      ' Debug instructions.
      "then type distance (up to ", CR,
      "255) then press ENTER.", CR, CR

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

```
DO

  DEBUG "Select direction (CH/VOL):", CR

  DO

    GOSUB Get_Ir_Remote_Code

    IF (remoteCode < ChUp) OR (remoteCode > VolDn) THEN
      DEBUG "Select direction (CH/VOL):", CR
      GOSUB Beep_Error
    ENDIF

  LOOP UNTIL (remoteCode >= ChUp) AND (remoteCode <= VolDn)

  direction = remoteCode

  GOSUB Beep_Valid

  DEBUG "Enter number of pulses: ", CR

  GOSUB Get_Multi_Digit_Value

  DEBUG "The value is: ", DEC value, CR
  DEBUG "Running...", CR, CR

  FOR counter = 1 TO value

    SELECT direction
      CASE ChUp
        PULSOUT ServoLeft, 850
        PULSOUT ServoRight, 650
      CASE ChDn
        PULSOUT ServoLeft, 650
        PULSOUT ServoRight, 850
      CASE VolUp
        PULSOUT ServoLeft, 850
        PULSOUT ServoRight, 850
      CASE VolDn
        PULSOUT ServoLeft, 650
        PULSOUT ServoRight, 650
    ENDSELECT

    PAUSE 20

  NEXT

LOOP

' -----[ Subroutine - Get_Ir_Remote_Code ]----------------------------------
```

```
' SONY TV IR remote subroutine loads the remote code into the
' remoteCode variable.

Get_Ir_Remote_Code:

  remoteCode = 0                            ' Clear all bits in remoteCode

  ' Wait for resting state between messages to end.

  DO
    RCTIME IrDet, 1, irPulse
  LOOP UNTIL irPulse > 1000

  ' Measure start pulse.  If out of range, then retry at Get_Ir_Remote_Code.

  RCTIME 9, 0, irPulse
  IF irPulse > 1125 OR irPulse < 675 THEN GOTO Get_Ir_Remote_Code

  ' Get data bit pulses.

  RCTIME IrDet, 0, irPulse                  ' Measure pulse
  IF irPulse > 300 THEN remoteCode.BIT0 = 1 ' Set (or leave clear) bit-0
  RCTIME IrDet, 0, irPulse                  ' Measure next pulse
  IF irPulse > 300 THEN remoteCode.BIT1 = 1 ' Set (or leave clear) bit-1
  RCTIME IrDet, 0, irPulse                  ' etc
  IF irPulse > 300 THEN remoteCode.BIT2 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT3 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT4 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT5 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT6 = 1

  ' Adjust remoteCode so that keypad keys correspond to the value
  ' it stores.

  IF (remoteCode < 10) THEN remoteCode = remoteCode + 1
  IF (remoteCode = 10) THEN remoteCode = 0

  RETURN

' -----[ Subroutine - Get_Multi_Digit_Value ]------------------------------

' Acquire multi-digit value (up to 65535) and store it in
' the value variable.  Speaker beeps each time a key is
' pressed.

Get_Multi_Digit_Value:
```

```
  value = 0
  remoteCode = 0

  DO

    value = value * 10 + remoteCode

    DO
      GOSUB Get_Ir_Remote_Code
      IF (remoteCode < 10) THEN
        DEBUG "You pressed: ", DEC1 remoteCode, CR
        GOSUB Beep_Valid
        EXIT
      ELSEIF (remoteCode = Enter) THEN
        DEBUG "You pressed: ENTER", CR
        GOSUB Beep_Valid
        EXIT
      ELSE
        DEBUG "Press 0-9 or ENTER", CR
        GOSUB Beep_Error
      ENDIF
    LOOP

  LOOP UNTIL (remoteCode = Enter)

  RETURN

' -----[ Subroutine - Beep_Valid ]------------------------------------

' Call this subroutine to acknowledge a key press.

Beep_Valid:

  FREQOUT Speaker, 100, 3500
  PAUSE 200

  RETURN

' -----[ Subroutine - Beep_Error ]------------------------------------

' Call this subroutine to reject a key press.

Beep_Error:

  FREQOUT Speaker, 100, 3000
  PAUSE 200

  RETURN
```

How KeypadDirectionDistance.bs2 Works

Two variables have to be added to IrRemoteKeypad.bs2, one for storing the Boe-Bot's direction, and the other for counting the number of pulses in a **FOR…NEXT** loop.

```
' Boe-Bot navigation variables

direction       VAR     Byte
counter         VAR     Byte
```

The first command inside the main **DO…LOOP** is a **DEBUG** command prompting for a CH/VOL direction.

```
DEBUG "Select direction (CH/VOL):", CR
```

The program calls the **Get_Ir_Remote_Code** subroutine over and over again, until a **remoteCode** that falls between **ChUp** (16) and **VolDn** (19) is received from the remote. If some button that's not CH+/- or VOL+/- is pressed, the **IF…THEN** statement delivers the lower pitched error beep along with a Debug Terminal prompt to press one of the CH or VOL buttons.

```
DO

  GOSUB Get_Ir_Remote_Code

  IF (remoteCode < ChUp) OR (remoteCode > VolDn) THEN
    DEBUG "Select direction (CH/VOL):", CR
    GOSUB Beep_Error
  ENDIF

LOOP UNTIL (remoteCode >= ChUp) AND (remoteCode <= VolDn)
```

After the **Get_Ir_Remote_Code** subroutine is called, the direction you sent to the Boe-Bot with the remote is stored in the **remoteCode** variable. This value needs to get stored in a different variable before another IR message is processed; otherwise, the value will be lost. That's why the direction value stored in **remoteCode** has to be copied to another variable, which is conveniently named **direction**.

```
direction = remoteCode
```

The right button had to have been pressed for the program to have exited the **DO…LOOP** that calls the **Get_Ir_Remote_Code**. The program calls the **Beep_Valid** subroutine, which makes the higher pitched acknowledgement beep to let you know the right key was pressed.

```
            GOSUB Beep_Valid
```

Another **DEBUG** command prompts you to enter the number of pulses (using the remote's numeric keypad).

```
        DEBUG "Enter number of pulses: ", CR
```

The **Get_Multi_Digit_Value** subroutine call is the part where you use the numeric keypad to enter the number of pulses to deliver. This is also the part where the value of the **remoteCode** value changes, which is why the direction you entered had to be copied to another variable. Unlike the **remoteCode** variable, the **value** variable will not be overwritten by anything in the program before the pulses are delivered to the servos. So **value** does not need to get copied to another variable; it can store the number of pulses.

```
        GOSUB Get_Multi_Digit_Value
```

A **DEBUG** command also lets you verify the value you entered (if you leave the Boe-Bot connected to the programming cable). A second **DEBUG** command displays the message "Running…" while the servos are being pulsed.

```
        DEBUG "The value is: ", DEC value, CR
        DEBUG "Running...", CR, CR
```

The direction is now stored in the **direction** variable, and the number of pulses is stored in the **value** variable. A **FOR…NEXT** loop uses the **value** variable to determine how many pulses it delivers to the servos. Inside the **FOR…NEXT** loop, a **SELECT…CASE** statement uses the value stored in the **direction** variable to decide which pulse durations to deliver to the servos. After the **SELECT…CASE**, **PAUSE 20** keeps the time between servo pulses constant.

```
        FOR counter = 1 TO value

          SELECT direction
            CASE ChUp
              PULSOUT ServoLeft, 850
              PULSOUT ServoRight, 650
            CASE ChDn
              PULSOUT ServoLeft, 650
              PULSOUT ServoRight, 850
            CASE VolUp
              PULSOUT ServoLeft, 850
              PULSOUT ServoRight, 850
            CASE VolDn
              PULSOUT ServoLeft, 650
```

```
      PULSOUT ServoRight, 650
    ENDSELECT

    PAUSE 20

  NEXT
```

Your Turn – Repeating The "LAST" Action

For TV control, the LAST button (sometimes labeled PREV CH) switches you back to the channel you viewed just before the channel you are currently watching. The LAST/PREV CH button could be wisely employed. Here's one way to modify your program to accomplish this task.

√   Save KeypadDirectionDistance.bs2 as KeypadDirectionDistanceYourTurn.bs2.
√   The Your Turn section of Activity #2 went through expanding the list of **CON** directives for IR remote buttons. Here is a constant you will need to add to your program for the LAST/PREV CH button:

```
Last            CON    59
```

√   Modify this **DO…LOOP**:

```
DO
  GOSUB Get_Ir_Remote_Code

  IF (remoteCode < ChUp) OR (remoteCode > VolDn) THEN
    DEBUG "Select direction (CH/VOL):", CR
    GOSUB Beep_Error
  ENDIF
LOOP UNTIL (remoteCode >= ChUp) AND (remoteCode <= VolDn)
```

by adding a condition to the **IF…THEN** statement that causes the program to jump to a label named **Servo_Pulses** if **remoteCode** stores the **Last** constant value.

```
DO
  GOSUB Get_Ir_Remote_Code

  IF (remoteCode) = Last THEN
    GOSUB Beep_Valid
    GOTO Servo_Pulses
  ELSEIF (remoteCode < ChUp) OR (remoteCode > VolDn) THEN
    DEBUG "Select direction (CH/VOL):", CR
    GOSUB Beep_Error
  ENDIF
LOOP UNTIL (remoteCode >= ChUp) AND (remoteCode <= VolDn)
```

√ Add this `Servo_Pulses:` label between the two `DEBUG` shown here:

```
    DEBUG "The value is: ", DEC value, CR

Servo_Pulses:                               ' <--- Add this label.

    DEBUG "Running...", CR, CR
```

> **Spaghetti Code Alert!**
>
> Using the `GOTO` command to jump to a label elsewhere in a program is frowned upon by many instructors, computer programmers, robot design managers, and others. The reason it is called "spaghetti code" is because of the difficulties you can encounter when trying to find a mistake in a program with too many `GOTO` commands. Understanding how the program works becomes like trying to visually follow a single noodle through a plate of spaghetti.

One non-spaghetti code way to implement the LAST/PREV CH button is by moving the code block that sends pulses to the servos into a subroutine. That way, a `GOSUB` command can be used instead of a `GOTO` command. `GOSUB` tends not to cause spaghetti code because the `RETURN` command sends the program to the command that immediately follows the subroutine call.

√ Save KeypadDirectionDistanceYourTurn.bs2 as
   KeypadDirectionDistanceYourTurn2.bs2.
√ Move the `Servo_Pulses:` label, the `DEBUG` command, and the `FOR…NEXT` loop
   that delivers the servo pulses from the main routine into a subroutine. It should
   look like this when you are done.

```
' -----[ Subroutine - Servo_Pulses ]---------------------------------

' Call this subroutine to deliver pulses to the servos.
' You must store the number of pulses in the value variable
' and the maneuver in the direction variable.  ChUp = forward,
' ChDn = backward, VolUp = rotate right, VolDn = rotate left.

Servo_Pulses:

  DEBUG "Running...", CR, CR
```

```
FOR counter = 1 TO value

  SELECT direction
    CASE ChUp
      PULSOUT ServoLeft, 850
      PULSOUT ServoRight, 650
    CASE ChDn
      PULSOUT ServoLeft, 650
      PULSOUT ServoRight, 850
    CASE VolUp
      PULSOUT ServoLeft, 850
      PULSOUT ServoRight, 850
    CASE VolDn
      PULSOUT ServoLeft, 650
      PULSOUT ServoRight, 650
  ENDSELECT

  PAUSE 20

NEXT

RETURN
```

√ In place of all that code you just removed from the main routine, add this subroutine call just before **LOOP**:

```
GOSUB Servo_Pulses
```

√ Change the line in the main routine that reads:

```
GOTO Servo_Pulses
```

so that it reads:

```
GOSUB Servo_Pulses
```

√ Save, run, and test the program. Verify that it behaves the same as the **GOTO** implementation.

SUMMARY

Decoding is the process of converting an electronic signal into something understandable and useable. In the case of the IR remote message, decoding involved converting pulse duration measurements into binary 1s and 0s in a byte variable. Each time the process completes, the byte variable stores a number (code) that corresponds with a key on the keypad.

In order to understand the decoding process, the concepts of counting in binary and binary to decimal conversion were introduced. The **.BIT** modifier was introduced as a way to set and clear bits in a variable. **IF…THEN** statements were written that examined pulse measurement to determine the value of a bit in a variable that stores the decoded value of the remote's pulse width modulated message. These **IF…THEN** statements employed the **.BIT** modifier to set a bit in a variable if the corresponding pulse measurement was above a certain value, or clear the bit if the pulse measurement was below a certain value.

This chapter developed application programs with constants, variable declarations, and subroutines that serve as building blocks that you can use in larger programs. These application programs reduced programs from Chapter 1 that were somewhat challenging down to a few lines in the application's main routine.

Techniques were introduced for modifying application program examples and using their features to the Boe-Bot's advantage. Examples included using constants with helpful names in decision making, calling subroutines to capture IR messages, and adding variable declarations, **PIN** directives and routines that adapt an application program for use with the Boe-Bot's servos.

Questions

1. What does it mean to "decode" an IR message from the universal remote?
2. What's a binary digit usually called?
3. How can you figure out the value a particular bit position represents?
4. What does it mean to set or clear a bit?
5. How do you use **.BIT** to set and clear bits in a variable?
6. How are constants used in IrRemoteButtons.bs2?
7. What does it mean when a pushbutton is debounced?

8. Why would you use a speaker with a pushbutton or keypad?

Exercises

1. Count from 0 to 7 in binary.
2. What must 24 be in binary?
3. Calculate the multiplier you would use for bit-12 in a binary number.
4. Convert 1111 to a decimal number.
5. Assume you have eight pulse measurements to decode instead of seven. Explain how to modify the `Get_Ir_Remote_Code` subroutine to capture and decode the pulse measurements.
6. Let's say you declared a bit variable named `onOffState`. Write a `SELECT…CASE` command that changes `onOffState` to a 1 if it's a 0 and to 0 if it's a 1. Use the ~ (not) operator.

Project

1. Modify 7BitRemoteBoeBot.bs2 so that the MUTE button on the remote can be used to disable and enable Boe-Bot navigation in response to the other keys.

Solutions

Q1. "To recognize and interpret an electronic signal."

Q2. A bit.

Q3. By raising 2 to the power of the bit position.

Q4. A bit is set when it is "set" to one. A bit is cleared if it is set to zero.

Q5. The variable name has to come before the **.BIT** operator, and the bit address has to follow. Do not use spaces. For example, to use the **.BIT** operator to refer to bit-4 in the **remoteCode** byte variable, use **remoteCode.BIT4**.

Q6. They give meaningful names to values used in the program. Specifically, the non-numeric buttons on the remote can be referred to by their names instead of their numeric values.

Q7. According to the ?-box on page 75, it means that the button can no longer misinterpret the rapid stream of ones and zeros sent by an electrical contact when the surfaces meet. Either part of the pushbutton circuit or the processors program filters out these unpredictable signals.

Q8. The speaker lets the user know the key-press sent the desired message.

E1. 0, 1, 10, 11, 100, 101, 110, 111.

E2. 11000.

E3. $2^{12} = 4096$.

E4. $8 + 4 + 2 + 1 = 15$.

E5. After these two commands:

```
RCTIME IrDet, 0, irPulse
IF irPulse > 300 THEN remoteCode.BIT6 = 1
```

add two more that capture another pulse and then set/clear **remoteCode.BIT7**.

```
RCTIME IrDet, 0, irPulse
IF irPulse > 300 THEN remoteCode.BIT7 = 1
```

E6.
```
SELECT remoteCode
  CASE Power
    onOffState = ~ onOffState
ENDSELECT
```

P1. This solution requires the piezospeaker circuit that was added in Activity #4. The following changes were made to the program:

2

√   Add this declaration to the Constants section:

```
MuteCode CON 20
```

√   Add a bit variable to the Variables section. Name it **mute**.

```
' Boe-Bot control bit
mute      VAR     Bit
```

√   Add this command to the Initialization that sets **mute** to 1.

```
' -----[ Initialization ]------------------------
mute = 1
```

√   Add this command between the **Get_Ir_Remote_Code** subroutine call and the **SELECT** statement:

```
IF (mute = 0) AND (remoteCode <> MuteCode) THEN
  remoteCode = 127
ENDIF
```

√   Add a **CASE** to the **SELECT…CASE** code block that reads:

```
CASE MuteCode
  FREQOUT 4, 50, 4000
  PAUSE 50
  FREQOUT 4, 50, 4000
  PAUSE 300
  mute = ~ mute
```

√   Comment the two **PULSOUT** commands below the **CASE ELSE** statement.

Chapter 3: More IR Remote Applications

EXPANDING APPLICATION PROGRAMS

The whole point of reusable code is that it allows you to more easily build on previous work to make better and more powerful applications. The previous work might be code you wrote, or it might be published code that you add to or adapt. In this chapter, you will see how the applications you have worked with up to this point can be merged with other applications. For example, Boe-Bot code from *Robotics with the Boe-Bot* will be added to the IR remote code application template to perform a variety of tasks.

In Activity #1, you will merge a program for autonomous IR object-avoidance roaming with an IR remote communication template. The result will be a roaming Boe-Bot whose speed you can control with the remote. In Activity #2, you will make the Boe-Bot into a multi-function remote controlled bot. With the press of a button, you will be able to choose from three of the most popular Boe-Bot behaviors: remote controlled Boe-Bot, roaming Boe-Bot, and following Boe-Bot. In Activity #3, you will design an interpreter for executing lists of instructions from the IR remote. In essence, you will be able to "program" your Boe-Bot with the IR remote.

ACTIVITY #1: AUTONOMOUS NAVIGATION WITH REMOTE SPEED CONTROL

This activity will demonstrate an example of autonomous navigation with remote adjustment. For autonomous navigation, the Boe-Bot will roam with infrared using the example program from *Robotics with the Boe-Bot* Chapter 7, Activity #5. This example program will then be modified so that you can set speed control from the Debug Terminal's Transmit Windowpane.

The autonomous IR roaming with speed control program can then be merged with the keypad entry program. After these two programs are combined, issuing speed control commands can be done with the IR remote instead of through the Debug Terminal.

One of the most common mistakes in robotics projects is trying to make all the parts work together in one fell swoop. The end result is usually a programming bug that's too difficult to find. It's best to make sure each part of the project works before integrating it into a larger system. With this point in mind, this activity is separated into four steps:

Step 1 – Rebuild, test, and trouble-shoot the IR detection system by following the steps in *Robotics with the Boe-Bot*, Chapter 7, Activites #1 and #2.

Step 2 – Test the roaming with IR object detection and IR interference programs that were introduced in *Robotics with the Boe-Bot*, Chapter 7, Activity #5.

Step 3 – Modify the program so that you can control roaming speed with a `DEBUGIN` command.

Step 4 – Integrate the roaming with speed control function into the IR remote application program that supports keypad entry of large numbers.

Step 1 - Rebuild, Test, and Trouble-Shoot the IR Detection System

Figure 3-1 shows the circuits for IR detection and user indicators from *Robotics with the Boe-Bot*, Chapter 7, Activity #2, and Figure 3-2 shows a way to build these circuits with wiring diagrams.

√   Build the circuits shown in Figure 3-1 and Figure 3-2.

√   Test the circuits following the instructions from *Robotics with the Boe-Bot*, Chapter 7, Activity #2. You will use both TestIrPairsAndIndicators.bs2 and IrInterferenceSniffer.bs2.

3

Vdd

P9 ◁——ᘓᘓ——
220 Ω

Vss

Left IR Pair

P8 ▷——ᘓᘓ——
1 kΩ

IR
LED

Vss

P2 ▷——ᘓᘓ——
1 kΩ

IR
LED

Vss

Vdd

P0 ◁——ᘓᘓ——
220 Ω

Vss
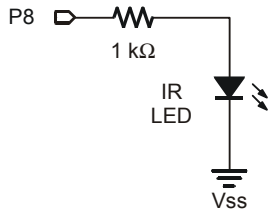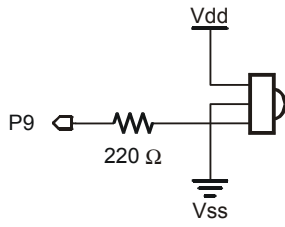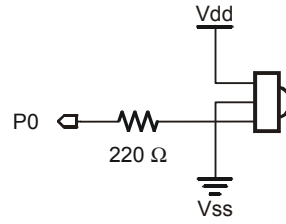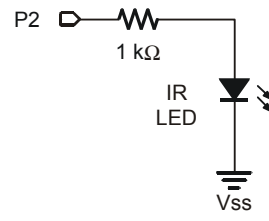
Right IR Pair

Figure 3-1
IR Detection
Testing and
Roaming
Circuit

P10 ▷——ᘓᘓ——
220 Ω

Red
LED
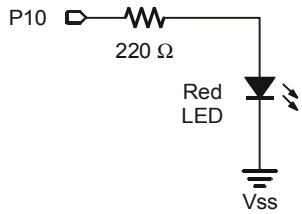
Vss
LED Indicator for Left IR Pair

P1 ▷——ᘓᘓ——
220 Ω

Red
LED
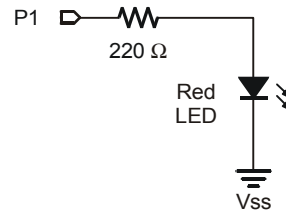
Vss
LED Indicator for Right IR Pair
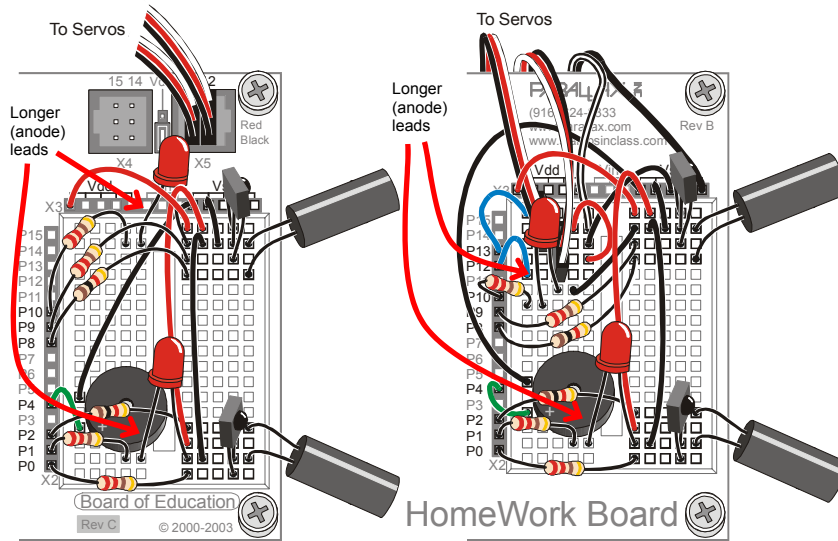
Figure 3-2
IR Detection Testing and Roaming Circuit Wiring Diagrams

Step 2 - Test the Roaming with IR Object Detection Program

Chapter 7, Activity #5 from *Robotics with the Boe-Bot* introduces a high performance version of roaming with IR. This program checks for an object between each servo pulse (that's around 40 times per second), so the Boe-Bot is very responsive when it detects obstacles. You will be modifying this program, so go ahead and do a refresher on how it works. It's also important to make sure the original program works with your circuit before moving on to the other modifications.

- √ Review the program FastIrRoaming.bs2 and the explanation of how it works in *Robotics with the Boe-Bot*, Chapter 7, Activity #5.
- √ Open (or re-enter) and test FastIrRoaming.bs2 by following the instructions in its activity.

Step 3 - Modify the Program so that You Can Control Roaming Speed

Variables can be used to control a variety of Boe-Bot behaviors. Among other things, these variables can tell the Boe-Bot how fast to go (this step) and what task to perform (next step). Just a few examples of conditions that can be used to change these variables are:

- Sensor inputs
- Messages from the Debug Terminal
- Messages from an IR remote

In the next example program, a variable named **speed** is added so that the speed of the servos can be set with a **DEBUGIN** command. By entering a value between 0 and 100, you can make the Boe-Bot roam at anywhere between 0 and 100 % of full speed.

The **IF…THEN** statement from FastIrRoaming.bs2 really needs some restructuring before it lends itself to speed control. Instead of using the values 650 and 850 for full speed, the values of **pulseLeft** and **pulseRight** should be determined by declaring a **speed** variable and then adding to or subtracting it from 750. Table 3-1 shows how the original **IF…THEN** statement from FastIrRoaming.bs2 looks next to the modified **IF…THEN** statement from the next example program, IrRoamingWithSpeedControl.bs2.

Table 3-1: Roaming Code with/without Speed Control	
Without Speed Control	**With Speed Control**

<pre>IF (irDetectLeft = 0) AND ••• THEN
  pulseLeft = 650
  pulseRight = 850
ELSEIF (irDetectLeft = 0) THEN
  pulseLeft = 850
  pulseRight = 850
ELSEIF (irDetectRight = 0) THEN
  pulseLeft = 650
  pulseRight = 650
ELSE
  pulseLeft = 850
  pulseRight = 650
ENDIF</pre>
	<pre>IF (irDetectLeft = 0) AND ••• THEN
	  pulseLeft = 750 - speed
	  pulseRight = 750 + speed
	ELSEIF (irDetectLeft = 0) THEN
	  pulseLeft = 750 + speed
	  pulseRight = 750 + speed
	ELSEIF (irDetectRight = 0) THEN
	  pulseLeft = 750 - speed
	  pulseRight = 750 - speed
	ELSE
	  pulseLeft = 750 + speed
	  pulseRight = 750 - speed
	ENDIF</pre>

On the "With Speed Control" side of the table, the **PULSOUT** commands to the left and right servos use the **pulseLeft** and **pulseRight** variables for their *Duration* arguments. The command **pulseLeft = 650** is replaced with **pulseLeft = 750 – speed**. When **speed** is a small value, **pulseLeft** is close to 750, and the Boe-Bot's left wheel rotates clockwise very slowly. As the **speed** variable gets closer to 100, the Boe-Bot's left wheel gets closer to full speed clockwise. Now, look at how **pulseRight = 850** has been replaced with **pulseRight = 750 + speed**. When **speed** is small, the

right wheel rotates counterclockwise slowly, and when `speed` is close to 100, it turns counterclockwise at full speed.

Example Program: IrRoamingWithSpeedControl.bs2

This example program is a modified version of FastIrRoaming.bs2 from *Robotics with the Boe-Bot*, Chapter 7, Activity #5. The comments in the program listing will show you which lines you will need to add or change to complete the modification.

Before IrRoamingWithSpeedControl.bs2 starts, the Debug Terminal will prompt you to enter a speed between 0 and 100. After you enter the desired speed, the Boe-Bot will roam at that percent of full speed.

- √ Enter and run IrRoamingWithSpeedControl.bs2.
- √ When you run the program, the Debug Terminal will prompt you for the percent of full speed that you want the Boe-Bot to roam at. Enter the desired speed into the Debug Terminal's Transmit Windowpane.

> **i** For a review of how to use the Debug Terminal's Transmit Windowpane, see Figure 1-12 on page 20.

- √ Try re-running the program a few times, each time selecting a different percent of full speed.

```
' IR Remote for the Boe-Bot - IrRoamingWithSpeedControl.bs2
' Higher performance IR object detection assisted navigation.
' The "<-- Add" comments indicate new commands lines of code.
' The "<-- Change" comments indicate lines of code that should be changed.

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

irDetectLeft    VAR     Bit                     ' Variable Declarations
irDetectRight   VAR     Bit
pulseLeft       VAR     Word
pulseRight      VAR     Word
speed           VAR     Byte            ' <-- Add

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

DEBUG CLS, "Enter percent of", CR,      ' <-- Add
          "full speed (0 TO 100): "     ' <-- Add
DEBUGIN DEC speed                       ' <-- Add
```

```
DEBUG "Main routine running..."          ' <-- Add

DO                                       ' Main Routine

  FREQOUT 8, 1, 38500                    ' Check IR Detectors
  irDetectLeft = IN9
  FREQOUT 2, 1, 38500
  irDetectRight = IN0
                                         ' Decide how to navigate.
  IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
    pulseLeft  =  750 - speed            ' <-- Change
    pulseRight =  750 + speed            ' <-- Change
  ELSEIF (irDetectLeft = 0) THEN
    pulseLeft  = 750 + speed             ' <-- Change
    pulseRight = 750 + speed             ' <-- Change
  ELSEIF (irDetectRight = 0) THEN
    pulseLeft = 750 - speed              ' <-- Change
    pulseRight = 750 - speed             ' <-- Change
  ELSE
    pulseLeft = 750 + speed              ' <-- Change
    pulseRight = 750 - speed             ' <-- Change
  ENDIF

  PULSOUT 13,pulseLeft                   ' Apply the pulse.
  PULSOUT 12,pulseRight
  PAUSE 15

LOOP                                     ' Repeat main routine
```

How IrRoamingWithSpeedControl.bs2 Works

A speed control variable is added to the Declarations section. The variable is wisely named **speed**.

```
        speed         VAR     Byte            ' <-- Add
```

A **DEBUG** command prompts you to enter the percent of full speed that you want the Boe-Bot to roam at.

```
        DEBUG CLS, "Enter percent of", CR,     ' <-- Add
                   "full speed (0 TO 100): "    ' <-- Add
```

A **DEBUGIN** command stores the value you enter into the **speed** variable. If the **speed** value is set to 100 by the user, the Boe-Bot will roam at full speed. When a value less than 100 is entered, the Boe-Bot will roam at a percentage of full speed.

```
        DEBUGIN DEC speed                       ' <-- Add
```

> **The percent value you enter determines the percent of the full speed pulse width.** This is not the same as the actual speed. For help predicting the actual speed, consult the transfer curves introduced in *Robotics with the Boe-Bot,* Chapter 3, Activity #4.

A **DEBUG** message indicates that the main routine is running.

```
DEBUG "Main routine running..."          ' <-- Add
```

This **IF…THEN** statement was introduced just before the example program. It sets the value of the **pulseLeft** and **pulseRight** variables. Depending on which direction each servo should turn, the **speed** value is either added to or subtracted from 750 to set the servo's speed.

```
IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
  pulseLeft  =  750 - speed          ' <-- Change
  pulseRight =  750 + speed          ' <-- Change
ELSEIF (irDetectLeft = 0) THEN
  pulseLeft  = 750 + speed           ' <-- Change
  pulseRight = 750 + speed           ' <-- Change
ELSEIF (irDetectRight = 0) THEN
  pulseLeft = 750 - speed            ' <-- Change
  pulseRight = 750 - speed           ' <-- Change
ELSE
  pulseLeft = 750 + speed            ' <-- Change
  pulseRight = 750 - speed           ' <-- Change
ENDIF
```

After the **pulseLeft** and **pulseRight** variable values have been set by the **IF…THEN** statement, these variables are used to set the **PULSOUT** *Duration* arguments. These **PULSOUT** commands send pulses to the left and right servos, and the pulse durations determine the direction and speed the Boe-Bot wheels turn.

```
PULSOUT 13,pulseLeft                       ' Apply the pulse.
PULSOUT 12,pulseRight
PAUSE 15
```

Your Turn – Saving Variable Space

You can save two word-size RAM variables by using **PULSOUT** commands inside the **IF…THEN** statement. Here's how:

- √ Save your working program under another name (like IrRoamingWithSpeedControlYourTurn.bs2).
- √ Click Run and select Memory Map (or CTRL-M or the Memory Map toolbar icon).

√ Check the RAM Map to see how much RAM you are using. The color coding should show that REG 0 and 1 are word storage variables. It should also show that REG 2 has a byte and two bit variables, as shown on the left in Figure 3-3.

√ Comment the `pulseLeft` and `pulseRight` variable declarations.

√ Replace all instances of `pulseLeft =` with `PULSOUT 13`, .
Be careful here, the result of your first substitution should be:

```
PULSOUT 13, 750 - speed
```

√ Replace all 4 instances of `pulseRight =` with `PULSOUT 12`, .
Again, the result of your first of four substitutions should be:

```
PULSOUT 12, 750 + speed
```

√ Comment out the two `PULSOUT` commands that come after the `ENDIF` in the program (by inserting an apostrophe to the left of each of the two commands).

√ Save, run, and test the modified program. Trouble-shoot as needed.

√ Open the Memory Map again, and verify that only a byte and two bit variables are allocated to `REG 0,` as shown on the right in Figure 3-3.
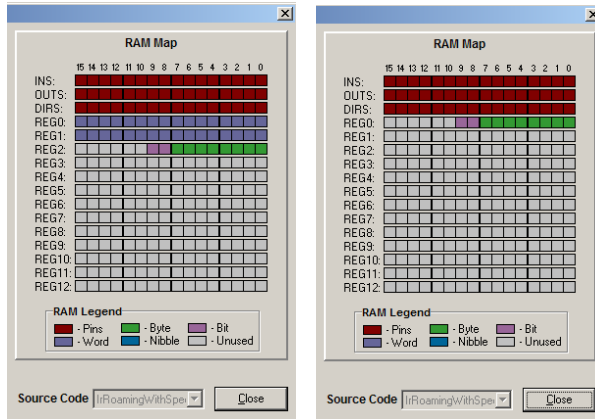
Figure 3-3
The RAM Map portion of the Memory Map

IrRoamingWithSpeedControl.bs2 (left)

Modified Your Turn version (right)

Step 4 - Integrate Roaming with Speed Control into the IR Template

By combining IrRoamingWithSpeedControl.bs2 with IrRemoteKeypad.bs2, you can use the IR remote to adjust the Boe-Bot's speed while it roams.

Here's how it will work:

- By pressing the any button, you will interrupt the Boe-Bot's roaming.
- Then, you will use the numeric keypad to type in the new percent-speed.
- To make the Boe-Bot resume roaming at the new speed, press the ENTER button.

IrRoamingWithSpeedControl.bs2 already uses infrared to check for obstacles between each pair of pulses to the servos. The key to detecting incoming messages from the remote is to check the IR detectors before using the **FREQOUT** command to look for objects. If the program checks to find out if IR is detected before checking for obstacles, it can easily detect an incoming message from the remote. Here are the steps that should be executed inside the main routine's **DO…LOOP**:

- Before testing for objects with the Boe-Bot's IRLED headlights, test the IR detector's output pin to see if a signal is coming from the remote.
  - If the IR detector is sending a low signal it means an infrared message is coming in. Call the **Process_Ir_Message** subroutine.
  - If the IR detector is sending a high, move on to the next task, which is object detection.
- Check the IR detectors.
- Use **IF…THEN** to control the servo directions and speeds.

Example Program: RoamingWithRemoteSpeedControl.bs2

Follow these steps to write the program:

√   Open IrRemoteKeypad.bs2.
√   Save a copy of it as RoamingWithRemoteSpeedControl.bs2.

> ℹ️ You can copy and paste to transfer sections of code from your version of IrRoamingWithSpeedControlYourTurn.bs2 into RoamingWithRemoteSpeedControl.bs2.

√   Add these declarations to the variables section:

```
' Boe-Bot control variables.

irDetectLeft    VAR    Bit                    ' IR detector bit storage
irDetectRight   VAR    Bit
speed           VAR    Byte                   ' Speed control variable
```

√ Add an Initialization section with this startup code for the Boe-Bot just before the main routine section.

3

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

' Boe-Bot initialization.

  DEBUG "Starting...", CR, CR            ' Signal program start/reset.
  FREQOUT 4, 2000, 3000
  speed = 0                             ' Initial speed is zero.
```

√ Replace the **DO…LOOP** in the main routine with this one:

```
DO                                         ' Main Routine.

  IF (IN9 = 0) OR (speed = 0) THEN         ' Check for IR message.
    FREQOUT Speaker, 100, 3500             ' Signal remote message
    PAUSE 100                              ' detected.

    FREQOUT Speaker, 100, 3500
    PAUSE 200

    DEBUG "Speed range: 0 to 100", CR, CR,
          "Type speed on remote", CR,
          "keypad, then press", CR,
          "ENTER", CR, CR

    GOSUB Get_Multi_Digit_Value            ' Get new speed.
    speed = value                          ' Set new speed.

    DEBUG ? speed, CR

    DEBUG "Running...", CR,
          "Press any key to", CR,
          "interrupt roaming", CR, CR

    PAUSE 250                              ' Pause for debounce.
  ENDIF

  FREQOUT 8, 1, 38500                      ' Check IR Detectors.
  irDetectLeft = IN9
  FREQOUT 2, 1, 38500
  irDetectRight = IN0
                                           ' Decide how to navigate.
```

```
        IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
          PULSOUT 13,  750 - speed                ' Backward
          PULSOUT 12,  750 + speed
        ELSEIF (irDetectLeft = 0) THEN
          PULSOUT 13, 750 + speed                 ' Rotate right
          PULSOUT 12, 750 + speed
        ELSEIF (irDetectRight = 0) THEN
          PULSOUT 13, 750 - speed                 ' Rotate left
          PULSOUT 12, 750 - speed
        ELSE
          PULSOUT 13, 750 + speed                 ' Forward
          PULSOUT 12, 750 - speed
        ENDIF

        PAUSE 15                                   ' Between servo pulses.

      LOOP                                         ' Repeat main routine.
```

√   Save and run your modified program.
√   To interrupt the Boe-Bot's roaming, point the remote at it and press any key. The Boe-Bot will beep twice to indicate that it received the signal. Make sure to release the key you're pressing as soon as you hear the beep.
√   Use the remote's numeric keypad to type in the new percent speed value (0 to 100).
√   Press the ENTER key to get the Boe-Bot to resume its roaming at the new speed.

How RoamingWithRemoteSpeedControl.bs2 Works

Normally, during IR detection, a **FREQOUT** command causes one of the IR LEDs to flash IR on/off at 38.5 kHz. If this flashing infrared light reflects off an object in the Boe-Bot's path, it will be detected by the IR detector. Remember that this value has to be stored in a bit variable immediately after the **FREQOUT** command. An instant later, the IR detector's output returns to high indicating that IR is not detected.

During roaming with no IR remote message coming in, the Main Routine's **DO…LOOP** executes over and over again, checking for obstacles with IR and controlling the servos. The key to intercepting an IR remote message is to check the state of the IR detectors output before broadcasting any infrared with the IR LEDs. The perfect time to do this is at the beginning of the Main Routine's **DO…LOOP**, because this comes right after the 15 ms pause at the end of the **DO…LOOP**.

The statement **IF (IN9 = 0) OR (speed = 0) THEN...** does the actual checking for an incoming IR remote message. It also checks whether or not the **speed** variable has

already been set. The first time the program is run, the value of `speed` is initialized to zero. The `IF…THEN` statement detects this and the code block within prompts you to enter a speed. The `IF…THEN` statement also detects when the remote is sending a message. When an infrared message is coming in, `IN9 = 0`. The code block starts by beeping twice to let you know that you can release the button you were pressing on the remote. Then, it calls the `Get_Multi_Digit_Value` subroutine, which is where you press the keys to set the new speed, followed by ENTER to re-start the Boe-Bot. When an infrared message is not coming in, `IN9 = 1`, and the code block between the `IF` and the `ENDIF` does not execute. The result is that the Boe-Bot keeps roaming.

```
IF (IN9 = 0) OR (speed = 0) THEN          ' Check for IR message.
  FREQOUT Speaker, 100, 3500              ' Signal remote message
  PAUSE 100                               ' detected.
     •
     •
     •
  GOSUB Get_Multi_Digit_Value             ' Get new speed.
  speed = value                           ' Set new speed.
     •
     •
     •
  PAUSE 250                               ' Pause for debounce.
ENDIF
```

Your Turn – Filtering for the POWER Key

At present, you can interrupt the Boe-Bot's roaming by pressing any TV control key on the remote. Of course, this won't work if you press keys on the remote that aren't for a TV control. Let's say you only want to use the POWER key to interrupt the Boe-Bot's roaming.

You can modify the code so that the `Get_Multi_Digit_Value` subroutine is only called when the POWER button is pressed by nesting all the code that gets the new speed from the remote inside a second `IF…THEN` statement. Before this `IF…THEN` statement, the `Get_Ir_Remote_Code` subroutine is called. Remember that this subroutine stores the remote button value it received in the `remoteCode` variable. So, this second, inner `IF…THEN` statement can check `remoteCode`. If it's equal to the `Power` constant, then update the `speed` variable; otherwise, play an error code on the piezo speaker.

√   Save RoamingWithRemoteSpeedControl.bs2 as
    RoamingWithRemoteSpeedControlYourTurn.bs2.

√   Replace this **IF…THEN** statement:

```
IF (IN9 = 0) OR (speed = 0) THEN          ' Check for IR message.
  FREQOUT Speaker, 100, 3500              ' Signal remote message
  PAUSE 100                               ' detected.
      •
      •
      •
  GOSUB Get_Multi_Digit_Value             ' Get new speed.
  speed = value                           ' Set new speed.
      •
      •
      •
  PAUSE 250                               ' Pause for debounce.
ENDIF
```

with this one:

```
IF (IN9 = 0) OR (speed = 0) THEN          ' Check for IR message.
  DEBUG "Press POWER to set", CR,
        "speed...", CR, CR
  GOSUB Get_Ir_Remote_Code
  IF (remoteCode = Power) OR (speed = 0) THEN
    FREQOUT Speaker, 100, 3500            ' Signal remote message
    PAUSE 100                             ' detected.

    FREQOUT Speaker, 100, 3500
    PAUSE 200

    DEBUG "Speed range: 0 to 100", CR, CR,
          "Type speed on remote", CR,
          "keypad, then press", CR,
          "ENTER", CR, CR

    GOSUB Get_Multi_Digit_Value           ' Get new speed.
    speed = value                         ' Set new speed.

    DEBUG ? speed, CR

    DEBUG "Running...", CR, CR

    PAUSE 250                             ' Pause for debounce.
  ELSE
    GOSUB Beep_Error
  ENDIF
ENDIF
```

√   Save the changes you made, then run and test the program.

ACTIVITY #2: MULTI-FUNCTION BOE-BOT WITH REMOTE SELECT

With the press of a remote button, you can select between three different Boe-Bot functions:

- Remote controlled Boe-Bot
- Autonomous roaming Boe-Bot
- Following Boe-Bot

Selecting Among Main Routines

The next example program starts with 7BitRemoteBoeBot.bs2 from this text. The main routine from this program and the main routines from FastIrRoaming.bs2 and FollowingBoeBot.bs2 (from *Robotics with the Boe-Bot*) can all be pasted into **CASE** statements. The main routine of this new program can then use one large **SELECT…CASE** statement that uses a variable named **operation** to select which routine to execute. The result is a Boe-Bot with three IR remote selectable functions.

> **i** Keep in mind that **PIN** directives, constant and variable declarations, and subroutines also have to be brought in from FastIrRoaming.bs2 and FollowingBoeBot.bs2.

Getting all the routines to work together involves some adjustments. There has to be some way to interrupt the Boe-Bot's current task, be it roaming, remote control or following, so that you can tell it to perform a different task. Since the number, channel, and volume keys are already being used for one of the tasks, we'll use the POWER button to interrupt the Boe-Bot's current task again.

Here's how the program's main routine will work:

```
      DO
        SELECT operation

          ' If operation = 1, execute a modified version of
          ' 7BitRemoteBoeBot.bs2 that also allows
          ' you to change the operation variable with the POWER key.
          CASE 1

             ' Modified main routine from 7BitRemoteBoeBot.bs2
             ' goes here.
                   •
                   •
                   •

          ' If operation = 2, execute modified FastIrRoaming.bs2.
          CASE 2

             ' Modified main routine from FastIrRoaming.bs2 goes here.
                   •
                   •
                   •

          ' If operation is 3, execute the FollowingBoeBot.bs2.
          CASE 3

             ' Modified main routine from FollowingBoeBot.bs2 goes here.
                   •
                   •
                   •

        ENDSELECT                                ' End SELECT operation

      LOOP
```

Let's first build and test the program, then we'll take a closer look at how it works.

Example Program: IrMultiBot.bs2

This example program starts off as the normal 7BitRemoteBoeBot.bs2. So, you can use the number, channel, and volume keys to drive the Boe-Bot around. You can make the Boe-Bot roam autonomously by pressing the POWER key, then the 2 key. Make the Boe-Bot follow objects by pressing the POWER key, then the 3 key. To return to remote keypad controlled navigation, press the POWER key, then the 1 key.

Here's how to build the program. First each of the three programs you will use to build this larger program have to be run and tested. 7BitRemoteBoeBot.bs2 was developed in this text, Chapter 2, Activity #3. The other two programs were developed in *Robotics*

with the Boe-Bot. FastIrRoaming.bs2 was featured in Chapter 7, Activity #5, and FollowingBoeBot.bs2 was featured in Chapter 8, Activity #2.

√   Load, run, and test 7BitRemoteBoeBot.bs2.
√   Load, run, and test FastIrRoaming.bs2.
√   Load, run, and test FollowingBoeBot.bs2.

When you have run and verified that each of the three programs work properly on their own, you will be ready to start integrating them.

√   Save a copy of 7BitRemoteBoeBot.bs2 as IrMultiBot.bs2.
√   Update the title and description comments in the Title section.

```
' -----[ Title ]-------------------------------------------------
' IR Remote for the Boe-Bot - IrMultiBot.bs2
' Select one of three Boe-Bot behaviors with the IR remote 1-3 keys.

' Press POWER key to interrupt the Boe-Bot's operation.
' Then, press one of these digit keys to select a new mode:

' 1 - Control Boe-Bot with 1-9 keys and/or CH+/- and VOL+/- keys.
' 2 - Roam and avoid objects.
' 3 - Follow objects.

' Note: Startup default is mode 1.

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

You will find the **Speaker PIN** directive to be useful.

√   Add this declaration to the I/O Definitions section:

```
Speaker        PIN    4
```

Next, you will need the constants, variables, and a couple of subroutines from FollowingBoeBot.bs2. Since FollowingBoeBot.bs2 built on FastIrRoaming.bs2, you will not need any extra constants, variables, or subroutines from FastIrRoaming.bs2.

√ Add these constant declarations to the Constants section of the program:

```
' Boe-Bot proportional control constants (from FollowingBoeBot.bs2).

Kpl             CON     -35
Kpr             CON     35
SetPoint        CON     2
CenterPulse     CON     750
```

√ Copy these variable declarations from FollowingBoeBot.bs2, and paste them into IrMultiBot.bs2's Variables section.

```
' Boe-Bot navigation variables (from FollowingBoeBot.bs2).

freqSelect      VAR     Nib
irFrequency     VAR     Word
irDetectLeft    VAR     Bit
irDetectRight   VAR     Bit
distanceLeft    VAR     Nib
distanceRight   VAR     Nib
pulseLeft       VAR     Word
pulseRight      VAR     Word
```

√ Add two more variable declarations for IrMultiBot.bs2 functions.

```
' IrMultiBot.bs2 variables.

counter         VAR     Nib              ' <--- New
operation       VAR     Nib              ' <--- New
```

√ Add these two subroutines to the end of the program:

```
' -----[ Subroutine – Get_IR_Distances ]------------------------------

Get_Ir_Distances:
  distanceLeft = 0
  distanceRight = 0
  FOR freqSelect = 0 TO 4
    LOOKUP freqSelect,[37500,38250,39500,40500,41500], irFrequency

    FREQOUT 8,1,irFrequency
    irDetectLeft = IN9
    distanceLeft = distanceLeft + irDetectLeft

    FREQOUT 2,1,irFrequency
    irDetectRight = IN0
    distanceRight = distanceRight + irDetectRight
  NEXT
  RETURN
```

```
' -----[ Subroutine - Send_Pulse ]------------------------------------

Send_Pulse:
  PULSOUT 13,pulseLeft
  PULSOUT 12,pulseRight
  PAUSE 5
  RETURN
```

√ Replace the initialization routine with the one shown here:

```
' -----[ Initialiazation ]-------------------------------------------

DEBUG "Press POWER to select", CR,
      "mode of operation:", CR, CR,
      "1 – Remote control 1-9 & CH/VOL", CR,
      "2 – Autonomous IR roaming", CR,
      "3 – Object find and follow", CR, CR

FREQOUT Speaker, 2000, 3000

operation = 1                                  ' Initialize to remote.
```

√ Modify the Main Routine section to make it like the one below. Although you can borrow heavily (copy/cut and paste) from the main routines of the programs you loaded and tested, you will still have to make adjustments to each of the **CASE** statements below. Modified or new lines will have comments like: **<---
Modified** or **<--- New**.

```
DO

  SELECT operation                        ' <--- New

    ' If operation = 1, execute a modified version of
    ' 7BitRemoteBoeBot.bs2 that also allows
    ' you to change the operation variable with the POWER key.

    CASE 1                                ' <--- New

      ' Modified main routine from 7BitRemoteBoeBot.bs2
      ' goes here.

      GOSUB Get_Ir_Remote_Code

      ' Check for POWER button.  If yes, get remote code; otherwise,
      ' send PULSOUT durations for the various maneuvers based on
      ' the value of the remoteCode variable.
```

```
            SELECT remoteCode
              CASE Power                               ' <--- New
                FREQOUT Speaker, 100, 3500            ' <--- New
                PAUSE 100                             ' <--- New
                FREQOUT Speaker, 100, 3500            ' <--- New
                PAUSE 200                             ' <--- New
                DEBUG "Select operation mode...", CR  ' <--- New
                GOSUB Get_Ir_Remote_Code              ' <--- New
                operation = remoteCode                ' <--- New
                DEBUG ? operation, CR,                ' <--- New
                      "Running...", CR                ' <--- New
                FREQOUT Speaker, 100, 3500            ' <--- New
                PAUSE 100                             ' <--- New
                FREQOUT Speaker, 100, 3500            ' <--- New
                PAUSE 200                             ' <--- New
              CASE 2, ChUp                             ' Forward
                PULSOUT 13, 850
                PULSOUT 12, 650
              CASE 4, VolDn                            ' Rotate Left
                PULSOUT 13, 650
                PULSOUT 12, 650
              CASE 6, VolUp                            ' Rotate Right
                PULSOUT 13, 850
                PULSOUT 12, 850
              CASE 8, ChDn                             ' Backward
                PULSOUT 13, 650
                PULSOUT 12, 850
              CASE 1                                   ' Pivot Fwd-left
                PULSOUT 13, 750
                PULSOUT 12, 650
              CASE 3                                   ' Pivot Fwd-right
                PULSOUT 13, 850
                PULSOUT 12, 750
              CASE 7                                   ' Pivot Back-left
                PULSOUT 13, 750
                PULSOUT 12, 850
              CASE 9                                   ' Pivot Back-right
                PULSOUT 13, 650
                PULSOUT 12, 750
              CASE ELSE                                ' Hold Position
                PULSOUT 13, 750
                PULSOUT 12, 750
            ENDSELECT

        ' If operation = 2, execute modified FastIrRoaming.bs2.
        CASE 2                                         ' <--- New

          ' Modified main routine from FastIrRoaming.bs2 goes here.
```

```
        IF IN9 = 0 THEN operation = 1              ' <--- New.

        FREQOUT 8, 1, 38500                        ' Check IR Detectors
        irDetectLeft = IN9
        FREQOUT 2, 1, 38500
        irDetectRight = IN0
                                                   ' Decide how to navigate.
        IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
          pulseLeft = 650
          pulseRight = 850
        ELSEIF (irDetectLeft = 0) THEN
          pulseLeft = 850
          pulseRight = 850
        ELSEIF (irDetectRight = 0) THEN
          pulseLeft = 650
          pulseRight = 650
        ELSE
          pulseLeft = 850
          pulseRight = 650
        ENDIF

        GOSUB Send_Pulse                           ' <--- Modified.

      ' If operation is 3, execute the following Boe-Bot routine.
      CASE 3                                        ' <--- New

        IF IN9 = 0 THEN operation = 1              ' <--- New

        GOSUB Get_Ir_Distances

        ' Calculate proportional output.

        pulseLeft =  SetPoint - distanceLeft  * Kpl + CenterPulse
        pulseRight = SetPoint - distanceRight * Kpr + CenterPulse

        GOSUB Send_Pulse

    ENDSELECT                                       ' <--- New
                                                    '(End SELECT operation)
  LOOP                                              ' Repeat Main Routine.
```

The completed program is shown below in case you need to check it for trouble-shooting.

√   Build, save, and run IrMultiBot.bs2.
√   Test the CH+/-, VOL+/-, and numeric keys and verify that it runs properly.
√   Press/release the POWER button. The Boe-Bot should beep twice.
√   Press/release the 2 key.

√ The Boe-Bot should beep twice, then start autonomously roaming and avoiding objects.
√ Press/release the POWER button. The Boe-Bot should beep twice.
√ Press/release the 3 key.
√ The Boe-Bot should now roam in object following mode. Test to make sure it will lock onto and follow an object.
√ Press/release the POWER button. The Boe-Bot should beep twice.
√ Press/release the 1 key.
√ This should return the Boe-Bot to remote control mode (CH/VOL +/- and number keys).

```
' -----[ Title ]------------------------------------------------------
' IR Remote for the Boe-Bot - IrMultiBot.bs2
' Select one of three Boe-Bot behaviors with the IR remote 1-3 keys.

' Press POWER key to interrupt the Boe-Bot's operation.
' Then, press one of these digit keys to select a new mode:

' 1 - Control Boe-Bot with 1-9 keys and/or CH+/- and VOL+/- keys.
' 2 - Roam and avoid objects.
' 3 - Follow objects.

' Note: Startup default is mode 1.

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

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

' SONY TV IR remote declaration - input receives from IR detector

IrDet          PIN    9
Speaker        PIN    4

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

' SONY TV IR remote constants for non-keypad buttons

Enter          CON    11
ChUp           CON    16
ChDn           CON    17
VolUp          CON    18
VolDn          CON    19
Power          CON    21

' Boe-Bot proportional control constants (from FollowingBoeBot.bs2).
```

```
Kpl             CON     -35
Kpr             CON     35
SetPoint        CON     2
CenterPulse     CON     750

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

' SONY TV IR remote variables

irPulse         VAR     Word
remoteCode      VAR     Byte

' Boe-Bot navigation variables (from FollowingBoeBot.bs2).

freqSelect      VAR     Nib
irFrequency     VAR     Word
irDetectLeft    VAR     Bit
irDetectRight   VAR     Bit
distanceLeft    VAR     Nib
distanceRight   VAR     Nib
pulseLeft       VAR     Word
pulseRight      VAR     Word

' IrMultiBot.bs2 variables.

counter         VAR     Nib                     ' <--- New
operation       VAR     Nib                     ' <--- New

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

DEBUG "Press POWER to select", CR,
      "mode of operation:", CR, CR,
      "1 - Remote control 1-9/CH/VOL", CR,
      "2 - Autonomous IR roaming", CR,
      "3 - Object find and follow", CR, CR

FREQOUT Speaker, 2000, 3000

operation = 1                                   ' Initialize to remote.

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

DO

  SELECT operation                              ' <--- New

    ' If operation = 1, execute a modified version of
    ' 7BitRemoteBoeBot.bs2 that also allows
    ' you to change the operation variable with the POWER key.

    CASE 1                                       ' <--- New
```

```
    ' Modified main routine from 7BitRemoteBoeBot.bs2
    ' goes here.

    GOSUB Get_Ir_Remote_Code

    ' Check for POWER button.  If yes, get remote code; otherwise,
    ' send PULSOUT durations for the various maneuvers based on
    ' the value of the remoteCode variable.

    SELECT remoteCode
      CASE Power                                ' <--- New
        FREQOUT Speaker, 100, 3500              ' <--- New
        PAUSE 100                               ' <--- New
        FREQOUT Speaker, 100, 3500              ' <--- New
        PAUSE 200                               ' <--- New
        DEBUG "Select operation mode...", CR    ' <--- New
        GOSUB Get_Ir_Remote_Code                ' <--- New
        operation = remoteCode                  ' <--- New
        DEBUG ? operation, CR,                  ' <--- New
              "Running...", CR                  ' <--- New
        FREQOUT Speaker, 100, 3500              ' <--- New
        PAUSE 100                               ' <--- New
        FREQOUT Speaker, 100, 3500              ' <--- New
        PAUSE 200                               ' <--- New
      CASE 2, ChUp                      ' Forward
        PULSOUT 13, 850
        PULSOUT 12, 650
      CASE 4, VolDn                     ' Rotate Left
        PULSOUT 13, 650
        PULSOUT 12, 650
      CASE 6, VolUp                     ' Rotate Right
        PULSOUT 13, 850
        PULSOUT 12, 850
      CASE 8, ChDn                      ' Backward
        PULSOUT 13, 650
        PULSOUT 12, 850
      CASE 1                            ' Pivot Fwd-left
        PULSOUT 13, 750
        PULSOUT 12, 650
      CASE 3                            ' Pivot Fwd-right
        PULSOUT 13, 850
        PULSOUT 12, 750
      CASE 7                            ' Pivot Back-left
        PULSOUT 13, 750
        PULSOUT 12, 850
      CASE 9                            ' Pivot Back-right
        PULSOUT 13, 650
        PULSOUT 12, 750
      CASE ELSE                         ' Hold Position
        PULSOUT 13, 750
```

```
        PULSOUT 12, 750
      ENDSELECT

    ' If operation = 2, execute modified FastIrRoaming.bs2.
    CASE 2                                     ' <--- New

      ' Modified main routine from FastIrRoaming.bs2 goes here.

      IF IN9 = 0 THEN operation = 1            ' <--- New.

      FREQOUT 8, 1, 38500                      ' Check IR Detectors
      irDetectLeft = IN9
      FREQOUT 2, 1, 38500
      irDetectRight = IN0
                                               ' Decide how to navigate.
      IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
        pulseLeft = 650
        pulseRight = 850
      ELSEIF (irDetectLeft = 0) THEN
        pulseLeft = 850
        pulseRight = 850
      ELSEIF (irDetectRight = 0) THEN
        pulseLeft = 650
        pulseRight = 650
      ELSE
        pulseLeft = 850
        pulseRight = 650
      ENDIF

      GOSUB Send_Pulse                         ' <--- Modified.

    ' If operation is 3, execute the following Boe-Bot routine.
    CASE 3                                     ' <--- New

      IF IN9 = 0 THEN operation = 1            ' <--- New

      GOSUB Get_Ir_Distances

      ' Calculate proportional output.

      pulseLeft =  SetPoint - distanceLeft  * Kpl + CenterPulse
      pulseRight = SetPoint - distanceRight * Kpr + CenterPulse

      GOSUB Send_Pulse

  ENDSELECT                                    ' <--- New
                                               '(End SELECT operation)
LOOP                                           ' Repeat Main Routine.

' -----[ Subroutine - Get_Ir_Remote_Code ]---------------------------------
```

```
' SONY TV IR remote subroutine loads the remote code into the
' remoteCode variable.

Get_Ir_Remote_Code:

  remoteCode = 0                              ' Clear all bits in remoteCode

  ' Wait for resting state between messages to end.

  DO
    RCTIME IrDet, 1, irPulse
  LOOP UNTIL irPulse > 1000

  ' Measure start pulse.  If out of range, then retry at Get_Ir_Remote_Code.

  RCTIME 9, 0, irPulse
  IF irPulse > 1125 OR irPulse < 675 THEN GOTO Get_Ir_Remote_Code

  ' Get data bit pulses.

  RCTIME IrDet, 0, irPulse                    ' Measure pulse
  IF irPulse > 300 THEN remoteCode.BIT0 = 1   ' Set (or leave clear) bit-0
  RCTIME IrDet, 0, irPulse                    ' Measure next pulse
  IF irPulse > 300 THEN remoteCode.BIT1 = 1   ' Set (or leave clear) bit-1
  RCTIME IrDet, 0, irPulse                    ' etc
  IF irPulse > 300 THEN remoteCode.BIT2 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT3 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT4 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT5 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT6 = 1

  ' Adjust remoteCode so that keypad keys correspond to the value
  ' it stores.

  IF (remoteCode < 10) THEN remoteCode = remoteCode + 1
  IF (remoteCode = 10) THEN remoteCode = 0

  RETURN

' -----[ Subroutine - Get IR Distances ]-----------------------------------

Get_Ir_Distances:
  distanceLeft = 0
  distanceRight = 0
  FOR freqSelect = 0 TO 4
    LOOKUP freqSelect,[37500,38250,39500,40500,41500], irFrequency
```

```
    FREQOUT 8,1,irFrequency
    irDetectLeft = IN9
    distanceLeft = distanceLeft + irDetectLeft

    FREQOUT 2,1,irFrequency
    irDetectRight = IN0
    distanceRight = distanceRight + irDetectRight
  NEXT
  RETURN

' -----[ Subroutine - Send_pulse ]-----------------------------------------

Send_Pulse:
  PULSOUT 13,pulseLeft
  PULSOUT 12,pulseRight
  PAUSE 5
  RETURN
```

How IrMultiBot.bs2 Works

As mentioned earlier, the Main Routine selects one of three different routines depending on the value stored in the **operation** variable. Each time through the **DO…LOOP**, if **operation** is 1, the routine inside the **CASE 1** statement is executed. This is the routine for Boe-Bot remote control with the CH/VOL and number keys. If **operation** is 2, the routine inside the **CASE 2** statement is executed. This is the routine for autonomous roaming. If **operation** is 3, the routine inside the **CASE 3** statement is executed. This is the routine for the following Boe-Bot.

```
      DO
        SELECT operation
          CASE 1

              ' Modified 7BitRemoteBoeBot.bs2
            ...

          CASE 2

              ' Modified FastIrRoaming.bs2
            ...

          CASE 3

              ' Modified FollowingBoeBot.bs2
            ...

        ENDSELECT
      LOOP
```

When the program starts, the value of `operation` is initialized to zero by default. This would be a problem because none of the `CASE` statements accept a zero. For this reason, a command setting `operation` to 1 was added to the Initialization section.

```
' -----[ Initialization ]----------------------------------------
   •
   •
   •
operation = 1                                     ' Initialize to remote.
```

The main routine from 7BitRemoteBoeBot.bs2 was modified to let you change the value of the `operation` variable with the remote. To do this, a `CASE` statement had to be added to handle a press/release of the POWER key.

```
SELECT remoteCode
  CASE Power
    FREQOUT Speaker, 100, 3500
    PAUSE 100
    FREQOUT Speaker, 100, 3500
    PAUSE 200
    DEBUG "Select operation mode...", CR
    GOSUB Get_Ir_Remote_Code
    operation = remoteCode
    DEBUG ? operation, CR,
          "Running...", CR
    FREQOUT Speaker, 100, 3500
    PAUSE 100
    FREQOUT Speaker, 100, 3500
    PAUSE 200
```

One problem is, what happens if you press/release the POWER key when `operation` is 2 (roaming) or 3 (following)? In these two modes, there isn't any way to change the value of the `remoteCode` variable with the remote. A simple solution is to insert a line at the beginning of the roaming and following routines that checks for an incoming message from the remote. This one line of code can be added to the other two roaming routines to make them respond to a press of the POWER key.

```
IF (IN9 = 0) THEN operation = 1          ' <--- New
```

With this one line of code, the first thing the roaming and following routines do is check to find out if a message from the remote really is coming in. If it is `(IN9 = 0)`, then, the value of `operation` is changed to 1. The next time through the `DO…LOOP`, the modified main routine from 7BitRemoteBoeBot.bs2 will be executed. This routine has a `CASE` statement for processing the POWER key and storing a new value in the operation

variable. This **CASE** statement contains commands that make it possible for you to select a different routine.

Your Turn – Fixing Bugs and Adding Comments

This program gets lost if you press the POWER key, then a number key other than 1, 2, or 3. Let's say **operation** is set to 4. In this case, there is no **CASE** statement, so the program just keeps repeating the **DO…LOOP** over and over again looking for **CASE 4**, and not finding it. The solution for this is to add a statement that catches all the potential values of **operations** and sends them back to the routine that allows you to select the **operation** variable with the remote. One way to do this is with a **CASE ELSE** statement.

√ Before modifying this program, test it by pressing POWER then any digit on the keypad other than 1, 2, or 3. Verify that there is nothing you can do with the remote to bring the Boe-Bot back to life. (You can press and release the RESET button on your board, but you're out of luck because there's not remote button you can use to wake the Boe-Bot back up.)

√ Locate the last three commands in the Main Routine, they should look like this:

```
    GOSUB Send_Pulse

  ENDSELECT

LOOP
```

√ Add the **CASE ELSE** statement shown below. This will cause the program to listen for commands from the remote, regardless of what the value of **operation** is.

```
    GOSUB Send_Pulse

  CASE ELSE                              ' <--- New

    IF (IN9 = 0) THEN operation = 1        ' <--- New

  ENDSELECT

LOOP
```

√ Test your modified program and make sure the Boe-Bot no longer gets confused when you set the operation variable to a value other than 1, 2, or 3.
√ Save your work!

ACTIVITY #3: REMOTE PROGRAMMED BOE-BOT

You can write a PBASIC program for your Boe-Bot that will allow you to program it with motion patterns using your remote. By pressing a sequence of buttons on the remote, you can load a sequence of maneuvers into the Boe-Bot's BASIC Stamp EEPROM memory. Here's an example of a sequence of key-presses that instructs the Boe-Bot to go forward for 40 pulses, rotate left for 20 pulses, rotate right for 20 pulses, then go backward for 40 pulses:

- POWER to initialize programming.
- CH+, 40, ENTER for forward 40 pulses.
- VOL-, 20, ENTER for rotate left by 20 pulses.
- VOL+, 20, ENTER for rotate right by 20 pulses.
- CH-, 40, ENTER for backward 40 pulses.
- ENTER a second time exits programming mode.
- ENTER a third time makes the Boe-Bot execute the maneuvers.
- ENTER again makes the Boe-Bot repeat the sequence of maneuvers.
- POWER to reprogram a new sequence of maneuvers.

> **What's a user interface?** It starts with the buttons, dials, displays, and menu systems you use to tell machines, appliances, and computer programs what you want them to do. Probably the most important aspect of any user interface is how it behaves in response to your button-press, dial turn, etc.
>
> If a given product's user interface is difficult to learn or doesn't make sense, it will quickly get a bad reputation, and people won't want to buy it
>
> User interface is often abbreviated as UI.

The challenging part about writing code for a UI is making sure that it doesn't confuse the person holding the remote. Here is a list of features the program should have to make the Boe-Bot easier to program with the remote:

- Recognize and discard incorrect key presses.
- Exit programming mode if ENTER is pressed twice in a row.
- Exit programming mode without asking how many pulses.
- Allow the user to replay the motion sequence many times.
- Remember the most recent motion sequence, even when the power has been disconnected and reconnected

This is another design that should be broken into a step-by-step process. The majority of the development work will be done with the Debug Terminal. After the program's functionality has been proven with the Debug Terminal, adapting it to the infrared template will be a relatively simple task.

Step 1 – Use the **READ** command to retrieve and display values that were stored in EEPROM at compile time with a **DATA** statement.

Step 2 – Exit a routine when a terminate character is received without asking for more information.

Step 3 – Store values in EEPROM during runtime with the **WRITE** command, then retrieve and display.

Step 4 – Don't accept characters that have no meaning to the program; wait until the right character is entered.

Step 5 – Nest the store and retrieve routines in a loop with menu options.

Step 6 – Adapt the Debug Terminal prototype to the IR remote template.

Step 7 – Add LED and speaker indicators to help the user.

> **Compile time vs. run time.** The work the editor does on the program before downloading it to the BASIC Stamp is done during compile time. **DATA** statements and **CON** and **VAR** directives are all processed during compile time. Commands that are executed by the BASIC Stamp while the program is running (**DEBUG**, **FREQOUT**, etc) are done during run time.

Step 1 – Read and Display Values Stored in EEPROM by a DATA Statement

In this step, we will write and test a program that stores characters in EEPROM during compile time and fetches and executes them during runtime. The storing will be done with the **DATA** directive, and the fetching and executing will be done with the **READ** and **DEBUG** commands. Let's review the **DATA** directive; here is its syntax from the BASIC Stamp Editor's PBASIC Syntax Guide.
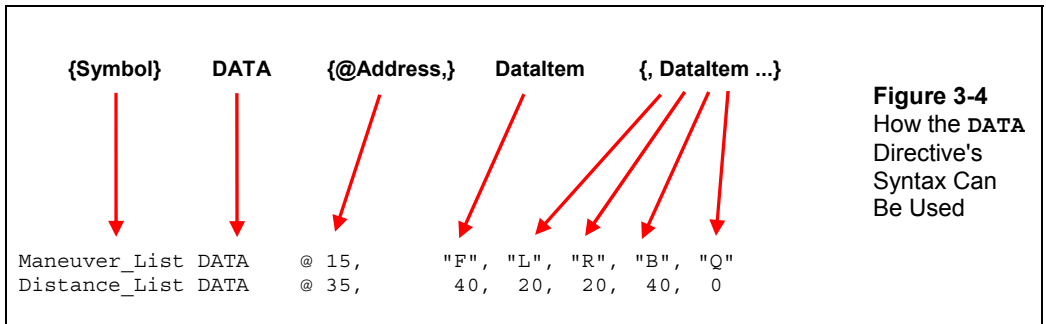
Syntax: {Symbol} DATA {@Address,} {Word} DataItem {, DataItem ...}

> To view the information about **DATA** in the PBASIC Syntax Guide, click Help in the BASIC Stamp Editor, and select Index. Type **DATA** into the keyword field, then double-click the entry when it appears in the list below.

Figure 3-4 shows examples of the two **DATA** directives in the next example program, DebugPlayback.bs2, and how they relate to the command syntax. Most of this syntax was introduced in *Robotics with the Boe-Bot v2.0* in Chapter 4, Activity #6. One element that may be new to you is the optional *{@Address}* argument. It is used in the two example **DATA** directives to place the *DataItems* at specific addresses in the BASIC Stamp's EEPROM.

In the first **DATA** directive, **@ 15** places the first *DataItem* at EEPROM address 15. The "F" is stored at address 15, the "L" at address 16, the "R" at address 17, and etc. The optional **Maneuver_List** symbol will automatically be set to 15 by the BASIC Stamp Editor during compile time. You can use this symbol in your program in place of the number 15, and as you will see, it makes writing programs to access EEPROM data much easier.

Figure 3-4
How the **DATA** Directive's Syntax Can Be Used

{Symbol}	DATA	{@Address,}	DataItem	{, DataItem ...}

```
Maneuver_List DATA    @ 15,      "F", "L", "R", "B", "Q"
Distance_List DATA    @ 35,       40,  20,  20,  40,  0
```

> **The programs in this activity will only use byte values, so the optional Word modifier will not be needed.** For more information and examples of the **Word** modifier, try the **DATA** directive examples with **Word** modifiers in the BASIC Stamp Editor's PBASIC Syntax Guide. You can also find examples that make use of the **Word** modifier in *What's a Microcontroller?* and *Robotics with the Boe-Bot*.

The second **DATA** directive also uses the *@Address* argument, and its *DataItems* begin at address 35. The result is that the number 40 is stored at address 35, the number 20 at address 36, and so on. Both **DATA** directives have the optional *{Symbol}* name. The symbol name for this second **DATA** directive is **Distance_List**, and it will become a constant 35, which you will also use in DebugPlayback.bs2.

Here is the syntax for the **READ** command:

READ *Location*, {*Word*} **Variable {,** {*Word*} **Variable, ...}**

Location is the EEPROM address that stores the value you want to retrieve, and *Variable* is the name of the variable that receives the value fetched from EEPROM. If you look up this command in the BASIC Stamp Editor's PBASIC Syntax Guide, it says this about the *Location* and the *Variable*:

- *Location* is a variable/constant/expression* (0 - 255 on BS1, 0 - 2047 on all other BASIC Stamp modules) that specifies the EEPROM address to read from.
- *Variable* is a variable (usually a byte) where the value is stored.

The great thing about Location is that it can be a "variable/constant/expression". An expression is typically some combination of variables and constants that involve some math.

Here is an excerpt from the next example program that uses expressions in the *Location* arguments of its **READ** commands.

```
eeIndex = 0

DO UNTIL (direction = "Q") OR (eeIndex = 19)

  READ Maneuver_List + eeIndex, direction
  READ Distance_List + eeIndex, distance

  DEBUG direction, "  ", DEC distance, CR

  PAUSE 200
  eeIndex = eeIndex + 1

LOOP
```

The first READ command adds Maneuver_List (a constant equal to 15) to eeIndex (a variable who's value is increased by one each time through the DO…LOOP). The first time through the DO…LOOP, eeIndex will be zero, and Maneuver_List is always 15. The result is that the READ command fetches the byte stored at EEPROM address 15 and stores it in the direction variable. Since the DATA directive discussed earlier stored an "F" at address 15, the READ command stores an "F" in the direction variable the first time through the loop.

The second time through, eeIndex will be 1 while Maneuver_List is still 15. The READ command fetches the "L" stored at address 16 and stores it in the direction variable. The third time through the loop, eeIndex is 2, so the Location expression evaluates to 17, and "R" is stored in the direction variable.

The same principle applies to the second READ command, except that Distance_List is 35, so the first time through, the value 40 is fetched from EEPROM and stored in distance, the second time through, 20, the third time through, another 20, and so on.

Example Program – DebugPlayback.bs2

This example program stores values in the BASIC Stamp's EEPROM with the **DATA** directive, then retrieves and displays these items with the **READ** command.

√   Enter and run DebugPlayback.bs2.

```
' -----[ Title ]-------------------------------------------------------------
' IR Remote for the Boe-Bot - DebugPlayback.bs2
' Fetch and display DataItems from a pair of DATA directives.

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

' -----[ EEPROM Data ]-------------------------------------------------------

 Maneuver_List   DATA    @ 15,   "F", "L", "R", "B", "Q"
 Distance_List   DATA    @ 35,    40,  20,  20,  40,  0

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

 direction      VAR     Byte
 distance       VAR     Byte
 eeIndex        VAR     Byte

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

  ' Playback routine

  eeIndex = 0

  DO UNTIL (direction = "Q") OR (eeIndex = 19)

    READ Maneuver_List + eeIndex, direction
    READ Distance_List + eeIndex, distance

    DEBUG direction, "  ", DEC distance, CR
```

```
    PAUSE 200

  eeIndex = eeIndex + 1

LOOP

END
```

Your Turn – Step 2 – Exit the Routine without Displaying Q or 0

The **DO…LOOP** can be modified so that it skips displaying the "Q" and the 0 at the end of the list. You can do this by removing the **(direction = "Q")** argument from the **DO UNTIL** statement. Then, add an **IF…THEN** statement with an **EXIT** command immediately after the **READ** commands. By executing **EXIT** when **direction = "Q"**, the program will skip out of the **DO…LOOP** before executing the **DEBUG** command:

```
DO UNTIL (eeIndex = 19)

  READ Maneuver_List + eeIndex, direction
  READ Distance_List + eeIndex, distance

  IF (direction = "Q") THEN EXIT

  DEBUG direction, "  ", DEC distance, CR
  PAUSE 200

  eeIndex = eeIndex + 1

LOOP
```

√   Rename and save the program as DebugPlaybackYourTurn.bs2.
√   Try modifying **DO…LOOP** code block in DebugPlaybackYourTurn.bs2 as shown.
√   Run the program and verify that it no longer displays the "Q" and the 0.
√   Save the modified program.

Step 3 – Store Runtime Values with the WRITE Command

When you beam your Boe-Bot directions with the IR remote, you will be storing values in EEPROM during runtime. While the **DATA** directive is for entering EEPROM data at compile time, the **WRITE** command is for storing values in EEPROM during runtime. In this activity, you will expand the program from the previous step so that you can write values to the EEPROM during runtime.

While the **READ** command retrieves a *DataItem* from a *Location* in EEPROM, the **WRITE** command stores a *DataItem* to a *Location*. Here is the **WRITE** command's syntax from the PBASIC Syntax guide:

WRITE *Location, {Word} DataItem {, {Word} DataItem ...}*

Here is the **DO…LOOP** that stores instructions with **WRITE** commands. By adding this to DebugPlaybackYourTurn.bs2, you can build your own list of characters in EEPROM during runtime.

```
' Routine - Record Instructions

  eeIndex = 0

  DO UNTIL (eeIndex = 19)

    DEBUG CR, "F, B, R, L, Q", CR,
          "Enter Direction: "
    DEBUGIN direction
    WRITE Maneuver_List + eeIndex, direction

    IF direction = "Q" THEN
      DEBUG CR, CR
      EXIT
    ENDIF

    DEBUG CR, "Enter distance: "
    DEBUGIN DEC distance
    WRITE Distance_List + eeIndex, distance

    eeIndex = eeIndex + 1

  LOOP
```

This code block will allow you to set the values of the **direction** and **distance** variables with the Debug Terminal's Transmit Windowpane. Each of these values will be stored in EEPROM with the **WRITE** command in the same manner that they were retrieved from EEPROM in DebugPlayback.bs2.

Let's take a closer look at the two **WRITE** commands in the loop:

-
-
-

```
WRITE Maneuver_List + eeIndex, direction
```

-
-
-

```
WRITE Distance_List + eeIndex, distance
```

Since **DATA** directives won't be needed, why do the **WRITE** commands still use **Maneuver_List** and and **Distance_List** in the *Location* expressions? The answer is because the **DATA** directives are still at the beginning of the program to hold those locations for memory space. The only difference is that no *DataItems* is written to EEPROM during compile time.

```
Maneuver_List    DATA    @ 15
Distance_List    DATA    @ 35
```

These two empty **DATA** directives act as anchors. **Maneuver_List** will still point to EEPROM address 15, and **Distance_List** will still point to 35. They can also still be used in **READ** and **WRITE** commands as place holders for the two different lists of EEPROM data. The only difference is that the **WRITE** commands will store values in these EEPROM memory locations during runtime.

The **DO…LOOP** for writing data to **EEPROM** has one other feature similar to the one we added in the previous activity's Your Turn section:

```
IF (direction = "Q") THEN EXIT
```

After the **WRITE** command to the **Direction_List**, the direction variable might still contain the character "Q". We can use this variable to decide whether or not to jump out of the **DO…LOOP** before getting a distance value. Since we really don't want to ask for any distance after "Q" is entered, the **IF…THEN** statement causes the program to jump to the instruction immediately following the **LOOP** command.

Example Program – DebugRecordPlayback.bs2

√   Enter and run DebugRecordPlayback.bs2. (This is a modified version of DebugPlaybackYourTurn.bs2.)

√   For best results, set the Caps Lock key on your keyboard so that the characters you type are capitalized ("F", "B", "L", "R", and "Q").

√   Follow the prompts and use the Debug Terminal's Transmit Windowpane to enter your directions and distances.

√   Verify that the list of directions and distances you entered are the same list that is played back after you enter the character "Q".

```
' -----[ Title ]------------------------------------------------------
' IR Remote for the Boe-Bot - DebugRecordPlayback.bs2
' Use Debug Terminal to Store a list of values to EEPROM, then retrieve
' and display them.

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

' -----[ EEPROM Data ]------------------------------------------------

  Maneuver_List   DATA    @ 15
  Distance_List   DATA    @ 35

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

  direction     VAR     Byte
  distance      VAR     Byte
  eeIndex       VAR     Byte

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

 ' Routine - Record Instructions

  eeIndex = 0

  DO UNTIL (eeIndex = 19)

     DEBUG CR, "F, B, R, L, Q", CR,
           "Enter Direction: "
     DEBUGIN direction
     WRITE Maneuver_List + eeIndex, direction

     IF direction = "Q" THEN EXIT

     DEBUG CR, "Enter distance: "
     DEBUGIN DEC distance
     WRITE Distance_List + eeIndex, distance

     eeIndex = eeIndex + 1

  LOOP
  DEBUG CR, CR

 ' Routine - Play Back Instructions

  eeIndex = 0
  direction = 0

  DO UNTIL (eeIndex = 19)

     READ Maneuver_List + eeIndex, direction
```

```
   READ Distance_List + eeIndex, distance

   IF direction = "Q" THEN EXIT

   DEBUG direction, "  ", DEC distance, CR
   PAUSE 200

   eeIndex = eeIndex + 1

 LOOP

 END
```

Your Turn – Step 4 – Wait until the Right Character is Entered

One problem with the existing example program is that it will accept any character, not just the "F", "B", "L", "R", and "Q" you want for navigation.

√   Run the program again and try entering characters other than those listed when prompted for direction.

A **SELECT…CASE** statement inside a **DO…LOOP** is a tool you can use to filter for only the characters you want before moving on. The **SELECT…CASE** statement can have two cases, one with a list of the characters you want to receive and an **ELSE** case for all the characters you don't want. When the direction variable contains one of the characters you want, the **CASE** with the list of correct characters can **EXIT** from the **DO…LOOP**. The **CASE ELSE** code block only has to contain a message that the wrong character was received. After the **ENDSELECT**, the **LOOP** will cause the **DO…LOOP** to repeat until the correct character is received. Here is how to change the program so that it filters for only the characters you want:

√   Rename and save the program as DebugRecordPlaybackFiltered.bs2
√   Replace these two commands:

```
DEBUG CR, "F, B, R, L, Q", CR,
     "Enter Direction: "
DEBUGIN direction
```

with this code block:

```
DO
  DEBUG CR, "F, B, R, L, Q", CR,
        "Enter Direction: "
  DEBUGIN direction
  SELECT direction
    CASE "F", "B", "R", "L", "Q"
      EXIT
    CASE ELSE
      DEBUG CR, "Invalid character", CR
  ENDSELECT
LOOP
```

√   Re-run the program and verify that it only accepts the characters: "F", "B", "R",
    "L", and "Q".
√   Save the modified program.

Step 5 - Nest the Store and Retrieve Routines in a Loop with Menu Options

The EEPROM storage and retrieval routines are working pretty well now. In this
activity, you will modify the program so that it gives you the choice of "R" for record or
"P" for playback. These characters can be used to choose between the record and
playback routines. Here is an example of the shell that contains your record and playback
routines:

```
DO
  DEBUG CLS,
        "R = Record", CR,
        "P = Play back", CR,
        "Choose operation: "
  DEBUGIN operation
  DEBUG CR

  IF (operation = "R") THEN
    ' Routine - Record Instructions
        •
        •
        •
  ELSEIF (operation = "P") THEN
    ' Routine - Play Back Instructions
        •
        •
        •
  ENDIF
LOOP
```

You will need to declare another byte variable named **operation** to support this main routine. Aside from that, the next example program is how your code should look after the modifications.

Example Program – DebugRecordPlaybackWithMenu.bs2

√   Save DebugRecordPlaybackFiltered.bs2 as
DebugRecordPlaybackWithMenu.bs2.
√   Modify it so that it matches the one below.
√   Run it.
√   Use the Debug Terminal to verify that you can type "R" to record directions and
distances and "P" to play them back.
√   Verify also that any other character will cause the message "Invalid character, try
again" to appear.
√   Save your program.

```
' -----[ Title ]----------------------------------------------------------
' IR Remote for the Boe-Bot - DebugRecordPlaybackWithMenu.bs2
' Use the Debug Terminal to select between record and playback modes.

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

' -----[ EEPROM Data ]----------------------------------------------------

  Maneuver_List   DATA    @ 15
  Distance_List   DATA    @ 35

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

  direction     VAR     Byte
  distance      VAR     Byte
  eeIndex       VAR     Byte
  operation     VAR     Byte

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

DO
  DEBUG CLS,
        "R = Record", CR,
        "P = Play back", CR,
        "Choose operation: "
  DEBUGIN operation
  DEBUG CR

  IF (operation = "R") THEN
```

```
   ' Routine - Record Instructions

   eeIndex = 0
   direction = 0

   DO UNTIL (eeIndex = 19)

     DO

       DEBUG CR, "F, B, R, L, Q", CR,
                  "Enter Direction: "
       DEBUGIN direction

       SELECT direction
         CASE "F", "B", "R", "L", "Q"
           EXIT
         CASE ELSE
           DEBUG CR, "Invalid character", CR
       ENDSELECT

     LOOP

     WRITE Maneuver_List + eeIndex, direction

     IF (direction = "Q") THEN EXIT

     DEBUG CR, "Enter distance: "
     DEBUGIN DEC distance
     WRITE Distance_List + eeIndex, distance

     eeIndex = eeIndex + 1

   LOOP

   DEBUG CR

 ELSEIF (operation = "P") THEN

' Routine - Play Back Instructions

 eeIndex = 0
 direction = 0

 DO UNTIL (eeIndex = 19)

   READ Maneuver_List + eeIndex, direction
   READ Distance_List + eeIndex, distance

   IF direction = "Q" THEN EXIT

   DEBUG direction, "  ", DEC distance, CR
```

```
   PAUSE 200

   eeIndex = eeIndex + 1

 LOOP

 ELSE

   DEBUG "Invalid character", CR,
         "try again."

 ENDIF

 DEBUG CR, "Press any key..."
 DEBUGIN operation
LOOP
```

Your Turn – Step 6 – Adapting maneuvers to the Boe-Bot

You can replace the **PAUSE 200** command with a simple routine to drive the Boe-Bot. Here's how:

- √ Rename and save the program as DebugRecordBoeBotPlayback.bs2.
- √ Add this declaration to the program's Variables section:

```
pulseCount    VAR     Byte
```

- √ Replace the **PAUSE 200** command with this code block:

```
FOR pulseCount = 1 TO distance

  SELECT direction
    CASE "F"
      PULSOUT 13, 850
      PULSOUT 12, 650
    CASE "B"
      PULSOUT 13, 650
      PULSOUT 12, 850
    CASE "L"
      PULSOUT 13, 650
      PULSOUT 12, 650
    CASE "R"
      PULSOUT 13, 850
      PULSOUT 12, 850
  ENDSELECT
  PAUSE 20

NEXT
```

√ Run the program and verify that you can program and reprogram the Boe-Bot's motion sequences with the Debug Terminal.

√ Save your modified program.

Step 7 – Adapting the Program to the Infrared Remote

At this point, you are now very close to a remote programmed Boe-Bot. You can take the main routine from Step 6, DebugRecordBoeBotPlayback.bs2, and, after some adapting, drop it into IrRemoteKeypad.bs2. Here is a list to give you a general idea of the adaptations that need to be made:

- **GOSUB Get_Ir_Remote_Code** is used in place of **DEBUGIN**.
- **GOSUB Get_Multi_Digit_Value** is used in place of **DEBUGIN DEC**.
- **ChUp**, **ChDn**, **VolUp**, and **VolDn** are used in place of "F", "B", "R", and "L".
- **ENTER** is used in place of "Q".
- **POWER** and **ENTER** are used in place of "P" and "R".

Below is a detailed account of how DebugRecordBoeBotPlayback.bs2 is adapted to and integrated into IrRemoteKeypad.bs2:

√ Open IrRemoteKeypad.bs2 and save a copy as RemoteRecordBoeBotPlayback.bs2.

√ Update the Title section so that it includes the correct program name, a brief description, and user instructions:

```
' -----[ Title ]--------------------------------------------------------
' IR Remote for the Boe-Bot - RemoteRecordBoeBotPlayback.bs2
' Press key sequences to program motion routines into the
' Boe-Bot's EEPROM and replay them.

' Press POWER key to program or ENTER key to play program.
' In programming mode, press a CH/VOL key to choose a maneuver.
' Use the keypad to enter the number of pulses, then press ENTER.
' Pressing ENTER again terminates programming.
' Pressing ENTER a third, fourth, etc time replays the program.
' Press POWER to reprogram.

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

√ Insert this section between the Stamp/PBASIC directives and the I/O Definitions.

```
' -----[ EEPROM Data ]-----------------------------------------------

' Set aside data for lists of Maneuver_List and Distance_List.

Maneuver_List DATA @ 15                    ' 20 bytes for Maneuver_List.
Distance_List DATA @ 35                    ' 20 bytes for Distance_List.
```

√ Copy these declarations into the Pin Definitions section:

```
' Boe-Bot Servo Pins

ServoLeft        PIN     13
ServoRight       PIN     12
```

√ Copy these declarations into the Variables section:

```
' Boe-Bot navigation variables

direction        VAR     Byte
counter          VAR     Word
eeIndex          VAR     Byte
```

> **i** If you examine this next **DO…LOOP** closely, you will see that it has the same overall function and structure as the main routine from DebugRecordBoeBotPlayback.bs2.

√ Replace the **DO…LOOP** in the main routine with this one.

```
DO
  DEBUG "Press POWER to record", CR,
        "or ENTER to playback", CR, CR

  GOSUB Get_Ir_Remote_Code

  IF remoteCode = Power THEN

    ' Routine - Record data

    GOSUB Beep_Valid

    eeIndex = 0

    DO UNTIL eeIndex = 19
      DEBUG "Menu: ", CR,
            "  CH+  = Forward", CR,
            "  CH-  = Backward", CR,
            "  VOL+ = Right", CR,
            "  VOL- = Left", CR,
            "  ENTER = Done recording", CR, CR
```

```
      DO

        GOSUB Get_Ir_Remote_Code

        SELECT remoteCode
          CASE ChUp TO VolDn, Enter
            GOSUB Beep_Valid
            EXIT
          CASE ELSE
            DEBUG "Press CH+/-, VOL+/-, or ENTER", CR
            GOSUB Beep_Error
        ENDSELECT

      LOOP

      direction = remoteCode
      WRITE eeIndex + Maneuver_List, direction

      IF (direction = Enter) THEN EXIT

      DEBUG "Enter number of pulses: ", CR
      GOSUB Get_Multi_Digit_VAlue
      DEBUG DEC value, " pulses", CR, CR
      WRITE eeIndex + Distance_List, value
      eeIndex = eeIndex + 1

    LOOP

  ELSEIF remoteCode = Enter THEN

    ' Routine - playback data

    GOSUB Beep_Valid
    DEBUG "Running...", CR, CR

    eeIndex = 0
    direction = 0

    DO UNTIL (eeIndex = 19)

      READ eeIndex + Maneuver_List, direction
      READ eeindex + Distance_List, value

      IF (direction = Enter) THEN EXIT

      FOR counter = 1 TO value
        SELECT direction
          CASE ChUp
            PULSOUT ServoLeft, 850
            PULSOUT ServoRight, 650
```

```
              CASE ChDn
                PULSOUT ServoLeft, 650
                PULSOUT ServoRight, 850
              CASE VolUp
                PULSOUT ServoLeft, 850
                PULSOUT ServoRight, 850
              CASE VolDn
                PULSOUT ServoLeft, 650
                PULSOUT ServoRight, 650
            ENDSELECT
            PAUSE 20
          NEXT

          eeIndex = eeIndex + 1

        LOOP

      ELSE

        GOSUB Beep_Error

      ENDIF

    LOOP
```

Example Program – RemoteRecordBoeBotPlayback.bs2

The completed program is shown below.

√   Enter and run the program and follow the prompts in the Debug Terminal for
     remote programming.
√   Try the following key sequence:

        √   POWER to initialize programming.
        √   CH+, 40, ENTER for forward 40 pulses.
        √   VOL-, 20, ENTER for rotate left 20 pulses.
        √   VOL+, 20, ENTER for rotate right 20 pulses.
        √   CH-, 40, ENTER for backward 40 pulses.
        √   ENTER a second time exits programming mode.
        √   ENTER a third time makes the Boe-Bot execute the maneuvers.
        √   ENTER again makes the Boe-Bot repeat the sequence of maneuvers.
        √   POWER to reprogram a new sequence of maneuvers.

```
' -----[ Title ]--------------------------------------------------
' IR Remote for the Boe-Bot - RemoteRecordBoeBotPlayback.bs2
' Press key sequences to program motion routines into the
```

```
' Boe-Bot's EEPROM and replay them.

' Press POWER key to program or ENTER key to play program.
' In programming mode, press a CH/VOL key to choose a maneuver.
' Use the keypad to enter the number of pulses, then press ENTER.
' Pressing ENTER again terminates programming.
' Pressing ENTER a third, fourth, etc time replays the program.
' Press POWER to reprogram.

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

' -----[ EEPROM Data ]-------------------------------------------------

' Set aside data for lists of Maneuver_List and Distance_List.

Maneuver_List DATA @ 15                 ' 20 bytes for Maneuver_List.
Distance_List DATA @ 35                 ' 20 bytes for Distance_List.

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

' SONY TV IR remote declaration - input receives from IR detector

IrDet           PIN     9
Speaker         PIN     4

' Boe-Bot Servo Pins

ServoLeft       PIN     13
ServoRight      PIN     12

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

' SONY TV IR remote constants for non-keypad buttons.

Enter           CON     11
ChUp            CON     16
ChDn            CON     17
VolUp           CON     18
VolDn           CON     19
Power           CON     21

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

' SONY TV IR remote variables

irPulse         VAR     Word                  ' Single-digit remote variables
remoteCode      VAR     Byte
index           VAR     Nib
value           VAR     Word                  ' Stores multi-digit value
```

3

```
' Boe-Bot navigation variables

direction       VAR     Byte
counter         VAR     Word
eeIndex         VAR     Byte

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

DO

  DEBUG "Press POWER to record", CR,
        "or ENTER to playback", CR, CR

  GOSUB Get_Ir_Remote_Code

  IF remoteCode = Power THEN

    ' Routine - Record data

    GOSUB Beep_Valid

    eeIndex = 0

    DO UNTIL eeIndex = 19

      DEBUG "Menu: ", CR,
            "  CH+  = Forward", CR,
            "  CH-  = Backward", CR,
            "  VOL+ = Right", CR,
            "  VOL- = Left", CR,
            "  ENTER = Done recording", CR, CR

      DO

        GOSUB Get_Ir_Remote_Code

        SELECT remoteCode
          CASE ChUp TO VolDn, Enter
            GOSUB Beep_Valid
            EXIT
          CASE ELSE
            DEBUG "Press CH+/-, VOL+/-, or ENTER", CR
            GOSUB Beep_Error
        ENDSELECT

      LOOP

      direction = remoteCode
      WRITE eeIndex + Maneuver_List, direction

      IF (direction = Enter) THEN EXIT
```

```
      DEBUG "Enter number of pulses: ", CR
      GOSUB Get_Multi_Digit_Value
      DEBUG DEC value, " pulses", CR, CR
      WRITE eeIndex + Distance_List, value
      eeIndex = eeIndex + 1

    LOOP

  ELSEIF remoteCode = Enter THEN

    ' Routine - playback data

    GOSUB Beep_Valid
    DEBUG "Running...", CR, CR

    eeIndex = 0
    direction = 0

    DO UNTIL (eeIndex = 19)

      READ eeIndex + Maneuver_List, direction
      READ eeindex + Distance_List, value

      IF (direction = Enter) THEN EXIT

      FOR counter = 1 TO value
        SELECT direction
          CASE ChUp
            PULSOUT ServoLeft, 850
            PULSOUT ServoRight, 650
          CASE ChDn
            PULSOUT ServoLeft, 650
            PULSOUT ServoRight, 850
          CASE VolUp
            PULSOUT ServoLeft, 850
            PULSOUT ServoRight, 850
          CASE VolDn
            PULSOUT ServoLeft, 650
            PULSOUT ServoRight, 650
        ENDSELECT
        PAUSE 20
      NEXT

      eeIndex = eeIndex + 1

    LOOP

  ELSE

    GOSUB Beep_Error
```

```
   ENDIF

LOOP

' -----[ Subroutine - Get_Ir_Remote_Code ]---------------------------------

' SONY TV IR remote subroutine loads the remote code into the
' remoteCode variable.

Get_Ir_Remote_Code:

  remoteCode = 0                             ' Clear all bits in remoteCode

  ' Wait for resting state between messages to end.

  DO
    RCTIME IrDet, 1, irPulse
  LOOP UNTIL irPulse > 1000

  ' Measure start pulse.  If out of range, then retry at Get_Ir_Remote_Code.

  RCTIME 9, 0, irPulse
  IF irPulse > 1125 OR irPulse < 675 THEN GOTO Get_Ir_Remote_Code

  ' Get data bit pulses.

  RCTIME IrDet, 0, irPulse                   ' Measure pulse
  IF irPulse > 300 THEN remoteCode.BIT0 = 1  ' Set (or leave clear) bit-0
  RCTIME IrDet, 0, irPulse                   ' Measure next pulse
  IF irPulse > 300 THEN remoteCode.BIT1 = 1  ' Set (or leave clear) bit-1
  RCTIME IrDet, 0, irPulse                   ' etc
  IF irPulse > 300 THEN remoteCode.BIT2 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT3 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT4 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT5 = 1
  RCTIME IrDet, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT6 = 1

  ' Adjust remoteCode so that keypad keys correspond to the value
  ' it stores.

  IF (remoteCode < 10) THEN remoteCode = remoteCode + 1
  IF (remoteCode = 10) THEN remoteCode = 0

  RETURN

' -----[ Subroutine - Get_Multi_Digit_Value ]------------------------------
```

```
' Acquire multi-digit value (up to 65535) and store it in
' the value variable.  Speaker beeps each time a key is
' pressed.

Get_Multi_Digit_Value:

  value = 0
  remoteCode = 0

  DO

    value = value * 10 + remoteCode

    DO
      GOSUB Get_Ir_Remote_Code
      IF (remoteCode < 10) THEN
        DEBUG "You pressed: ", DEC1 remoteCode, CR
        GOSUB Beep_Valid
        EXIT
      ELSEIF (remoteCode = Enter) THEN
        DEBUG "You pressed: ENTER", CR
        GOSUB Beep_Valid
        EXIT
      ELSE
        DEBUG "Press 0-9 or ENTER", CR
        GOSUB Beep_Error
      ENDIF
    LOOP

  LOOP UNTIL (remoteCode = Enter)

  RETURN

' -----[ Subroutine - Beep_Valid ]------------------------------------

' Call this subroutine to acknowledge a key press.

Beep_Valid:

  FREQOUT Speaker, 100, 3500
  PAUSE 200

  RETURN

' -----[ Subroutine - Beep_Error ]------------------------------------

' Call this subroutine to reject a key press.

Beep_Error:
```

```
FREQOUT Speaker, 100, 3000
PAUSE 200

RETURN
```

3

Your Turn – LED Prompts and Program Organization

To make this program complete, a pair of LEDs can make it easier to program the Boe-Bot without the Debug Terminal. This could come in handy for certain navigation contests.

- √   Rename your program RemoteRecordBoeBotPlaybackLed.bs2.
- √   Design and implement LED prompts that accompany the Debug Terminal information.
- √   Write an instruction manual for your Boe-Bot describing how to use the remote to program the Boe-Bot relying on just the speaker and LEDs. Use the pamphlet that came with the universal remote as an example.

The Program and Run routines should be moved to subroutines.

- √   Do it, then save your work. You might want to reuse them in other programs.

SUMMARY

This chapter introduced some more applications you can create with the IR remote application. Application programs have to be well organized, with each task accomplished in subroutines. The I/O pins, variables, and constants also have to be defined and documented in the declarations section. When application programs follow these conventions, it makes it possible to combine the two (or more) of these programs to achieve more difficult and complex robotic goals.

The first two activities focused on how to merge and integrate more than one application program into a larger program that does more. In each of these activities, subroutines from the various programs were copied and pasted into a larger program, likewise with the constants, variables, and other declarations. By virtue of the fact that application programs are written to be modular, none of the subroutines had to be modified. Instead, they were utilized and orchestrated in the main routine. By relying on the functionality of the subroutines, the main routine could be written in simpler terms while accomplishing much more difficult tasks.

In the first two activities, testing programs and circuits that were used in *Robotics with the Boe-Bot* was emphasized. Each of the programs and circuits have to be known to work on their own before they can be combined into a larger circuit and/or program. **SELECT…CASE** was introduced as a way of choosing between alternate main routine options that are chosen by a menu system. This scheme is easy to expand by simply adding more **CASE** statements.

The subroutines in a single application template can also be used in new and creative ways to create more powerful robotic behaviors. The third activity demonstrated this by building an IR remote keypad entry scheme for programming the Boe-Bot. EEPROM storage was examined more closely as a tool for storing and retrieving sequences of characters and values. By using the **@Address** operator in a **DATA** directive, you can define blocks of unused program memory for use by the program. Optional *Symbol* names can be placed before the **DATA** to help calculate the location of a given **DataItem**. It makes storing values to **EEPROM** with the **WRITE** command and retrieving them with the **READ** command much simpler. Especially if you have to store and manage more than one list of related items, such as Boe-Bot maneuvers and distances.

Elements of user interface (UI) design have been introduced throughout this text. In earlier chapters, button debouncing and speaker and LED feedback have already been introduced and applied. In the third activity of this chapter, programming techniques were introduced for limiting the key presses that the program will accept. Designing and implementing sensible sequences of key presses for choosing certain robotic functions was introduced by example.

A stepwise process for solving complex tasks was also introduced. The Debug Terminal's Transmit and Receive Windowpanes were relied on heavily as a prototyping tool for UI design. After the interactions with the user were defined with the help of the Debug Terminal, the **DEBUGIN** and **DEBUGIN DEC** commands were ported to an IR remote communication template.

Questions

1. What's the name of the variable used for speed control in IrRoamingWithSpeedControl.bs2?
2. What does the **CLS** in **DEBUG CLS** do?
3. What happens when the variable used in a **SELECT…CASE** statement does not contain any of the values specified in the **CASE** statements? If this becomes a problem, how can you fix it?
4. What does the optional *@Address* argument do for a **DATA** directive? How does this effect the value of the optional *Symbol*?
5. What's the difference between the **READ** command and the **WRITE** command?
6. How can **SELECT…CASE** statement be used inside a **DO…LOOP** to wait for a specific value before continuing to the next step in the program?

Exercises

1. In IrRoamingWithSpeedControl.bs2, what will the pulse durations for the left and right servos be if no object is detected and speed = 50?
2. The Selecting between Main Routines section in Activity #2 shows the shell of a **SELECT…CASE** statement. Expand this shell so that it contains a case for 4 and a catch-all case for any other value.
3. Write **READ** commands to retrieve the values 7 and 13 using **My_List** in the *Location* argument. Assume you have declared a variable = **myValue**.

4. Modify this code block so that it only accepts values between 16 and 19.

```
DO
  DEBUG CR, "F, B, R, L, Q", CR,
        "Enter Direction: "
  DEBUGIN direction
  SELECT direction
    CASE "F", "B", "R", "L", "Q"
      EXIT
    CASE ELSE
      DEBUG CR, "Invalid character", CR
  ENDSELECT
LOOP
```

5. List the remote key press sequence that makes the Boe-Bot draw a 50 pulse by 100 pulse rectangle with RemoteRecordBoeBotPlayback.bs2.

Projects

1. Use 7BitRemoteBoeBot.bs2 from Chapter 2, Activity #3 as your starting point. Add a routine that overrides instructions from the remote and stops the Boe-Bot before it collides with an object. This way, you can try to run the Boe-Bot into an obstacle, but it will not let you.
2. Expand IrMultiBotYourTurn.bs2 so that you can press the following digit keys for the following functions:
   - 1 – Remote button controlled Boe-Bot
   - 2 – Full speed IR roaming
   - 3 – Following Boe-Bot
   - 4 – Servo centering mode
   - 5 – Slow roaming Boe-Bot for lead roaming Boe-Bot
   - 6 – One-shot IR interference sniffer

Solutions

Q1. The variable's name is `speed`.
Q2. According to the PBASIC Syntax Guide, it clears the screen. In other words, it erases everything currently displayed in the Debug Terminal's Transmit Windowpane.
Q3. If none of the `CASE` statements match the `SELECT` statement's variable the program finds the `ENDSELECT` keyword and moves on from there. You can add a catch-all statement `CASE ELSE` with any code that you want the `SELECT...CASE`

statement to execute if the variable that was selected stores a value that does not match any of the other **CASE** statements.

Q4. The **@Address** operator allows you to specify the starting address in EEPROM for the **DATA** directive's first **DataItem**. The **Symbol** for that data directive will be a constant equal to the value that follows the **@Address** operator. This will also be the address of the first **DataItem** in the **DATA** directive.

Q5. The **WRITE** command stores a **DataItem** to a **Location** in EEPROM; the **READ** command fetches a **DataItem** from a **Location** in EEPROM.

Q6. A **DO…LOOP** can have a **SELECT…CASE** statement nested inside it. One of the **CASE** statements can contain a list of possible values that you want to relieve before moving on in the program. The code block for this **CASE** statement should contain the **EXIT** command for breaking out of the **DO…LOOP** since its code block will be executed when one of the desired values is stored by the **SELECT** statement's variable. The other **CASE ELSE** statement should contain a code block that's executed when the wrong value is received. Since it will not contain an **EXIT** command, the **DO…LOOP** will repeat itself until one of the desired values is stored in the **SELECT** statement's variable.

E1. If no object is detected, the **CASE ELSE** statement is executed. This case sets the variables used for the **PULSOUT** commands to the left and right servos. The value of **pulseLeft** will be $750 - 50 = 700$. To figure the pulse duration, multiply the **PULSOUT** command's duration argument by 2 μs. 700×0.000002 s $= 0.0014$ s $= 1.4$ ms. The value of **pulseRight** will be $750 + 50 = 800$, which delivers a 1.6 ms pulse.

E2. Near the main routine's **DO…LOOP**, there is an **ENDSELECT**. Insert this code block just above the **ENDSELECT**.

```
' If operation is 4, execute the YourProgram.bs2.
CASE 4

   ' Modified main routine from YourProgram.bs2 goes here.
           •
           •
           •
' If operation is not 1-4, execute this routine.
CASE ELSE

   ' Catch-all commands go there.
           •
           •
           •
```

E3. Solution: **READ My_List + 4, myValue: READ My_List + 7, myValue**

E4. Here is the modified code. Be careful to make sure to add the **DEC** operator to the **DEBUGIN** command.

```
DO
  DEBUG CR, "16, 17, 18, or 19", CR,
        "Enter Value: "
  DEBUGIN DEC direction
  SELECT direction
    CASE 16 to 19
      EXIT
    CASE ELSE
      DEBUG CR, "Invalid character", CR
  ENDSELECT
LOOP
```

E5. Try this sequence with some adjustments to the 21 pulse turns to get 90-degrees with your servos:

√ POWER to initialize programming.
√ CH+, 100, ENTER for forward 100 pulses.
√ VOL-, 21, ENTER for rotate left 21 pulses.
√ CH+, 50, ENTER for forward 50 pulses.
√ VOL-, 21, ENTER for rotate left 21 pulses.
√ CH+, 100, ENTER for forward 100 pulses.
√ VOL-, 21, ENTER for rotate left 21 pulses.
√ CH+, 50, ENTER for forward 50 pulses.

√ ENTER a second time exits programming mode.
√ ENTER a third time makes the Boe-Bot execute the maneuvers.

3

P1. Add these bit declarations to the Variables section of the program.

```
irDetectLeft    VAR     Bit
irDetectRight   VAR     Bit
```

√ Grab the IR detection functions from TestIrPairsAndIndicators.bs2 and paste them into your project program just before the **SELECT** statement in the main routine.

```
FREQOUT 8, 1, 38500
irDetectLeft = IN9

FREQOUT 2, 1, 38500
irDetectRight = IN0
```

√ Modify this **CASE** statement:

```
CASE 2, ChUp
  PULSOUT 13, 850
  PULSOUT 12, 650
```

so that it reads

```
CASE 2, ChUp
  IF (irDetectLeft = 1) AND (irDetectRight = 1) THEN
    PULSOUT 13, 850
    PULSOUT 12, 650
  ELSE
    FREQOUT Speaker, 3, 4500
  ENDIF
```

P2. Add case statements to the outer **SELECT…CASE** in the Main Routine's **DO…LOOP**
for each function. Remember to update the menu list in the Initialization routine.

```
DO
  SELECT operation

    CASE 1
            •
            •
            •
            •
    CASE 2
            •
            •
            •
    CASE 3
            •
            •
            •
    ' If operation is 4, put Boe-Bot in servo centering mode.
    CASE 4

      IF IN9 = 0 THEN operation = 1

      pulseLeft = 750
      pulseRight = 750

      GOSUB Send_Pulse

    ' If operation is 5, run slow roaming routine for the lead Boe-Bot
    CASE 5

      ' Modified main routine from FastIrRoaming.bs2 goes here.

      IF IN9 = 0 THEN operation = 1

      FREQOUT 8, 1, 38500                    ' Check IR Detectors
      irDetectLeft = IN9
      FREQOUT 2, 1, 38500
      irDetectRight = IN0
                                             ' Decide how to navigate.
      IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
        TOGGLE 1
        TOGGLE 10
        pulseLeft = 725
        pulseRight = 775
      ELSEIF (irDetectLeft = 0) THEN
        TOGGLE 10
        pulseLeft = 775
        pulseRight = 775
      ELSEIF (irDetectRight = 0) THEN
```

```
          TOGGLE 1
          pulseLeft = 725
          pulseRight = 725
        ELSE
          pulseLeft = 775
          pulseRight = 725
        ENDIF

        GOSUB Send_Pulse                        ' <--- Modified.

        LOW 1
        LOW 10

    ' If operation is 6, test for ballast interference.  This versions
    ' of the ballast interference tester only triggers once.  If
    ' triggered, the remote has to be used to reset the Boe-Bot to
    ' function 6 to repeat the 1-shot ballast interference detection.
     CASE 6

        DO

          IF IN9 = 0 THEN

          operation = 1

          FOR counter = 1 TO 10
            FREQOUT 4, 40, 4500
            PAUSE 50
          NEXT

          EXIT

          ENDIF

        LOOP

      CASE ELSE

        IF IN9 = 0 THEN operation = 1          ' <--- New

    ENDSELECT                                   ' <--- New
                                                '(End SELECT operation)
LOOP                                            ' Repeat Main Routine.
```

Appendix A: IR Remote AppKit Documentation

PARALLAX ⋷

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 1.1

INFRARED REMOTE APPKIT (#29122)

A Wireless Keypad for Your BASIC Stamp® Microcontroller Module

With a universal remote and an infrared receiver, you can add a wireless keypad to your BASIC Stamp Applications. The IR receiver is inexpensive, and only takes one I/O pin. Universal remotes are also inexpensive, easy to obtain and replace, and have enough buttons for most applications. The parts in this kit along with the example programs make it possible to enter values and control your projects in the same way you might with a TV, VCR, or other entertainment system component.

IR Remotes can also add zing to your robotics projects. While this package insert provides you with the essential background information, circuits, and example programs to get started, you can learn lots more with *IR Remote for the Boe-Bot*. This text is for the most part, a continuation of *Robotics with the Boe-Bo*, but with an IR remote twist. It follows the same format in terms of introducing new hardware, explaining how things work, and demonstrating new PBASIC techniques. IR remote applications for the Boe-Bot™ robot include remote control, keypad entry control, hybrid autonomous and remote control, and remote motion sequence programming.

Kit Contents*

Infrared Remote Parts List:

(1) 020-00001   Universal Remote and Universal Remote Manual
(1) 350-00014   IR detector
(1) 150-02210   Resistor – 220 Ω
(1) 800-00016   Jumper wires – bag of 10

*Two alkaline AA batteries sold separately
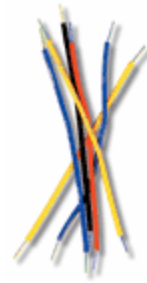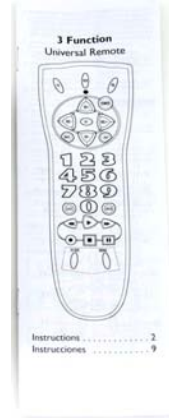
How IR Communication Works

The universal remote sends messages by strobing its IR LED at 38.5 kHz for brief periods of time. The actual data is contained in the amount of time each strobe lasts. There are many different IR protocols, but, in general, the amount of time each 38.5 kHz signal lasts transmits some kind of message. One duration might indicate the start of a message, while another indicates a binary-1, and still another indicates a binary-0.

The IR detector's output pin sends a low signal while it detects the 38.5 kHz IR signal, and a high signal while it does not. So, a low signal of one duration might indicate the start of a message, while another indicates a binary-1, and still another indicates a binary 0. This communication scheme is called pulse width modulation (PWM), because when graphed against time, the IR detector's high/low signals form pulses of different widths that correspond to their durations.
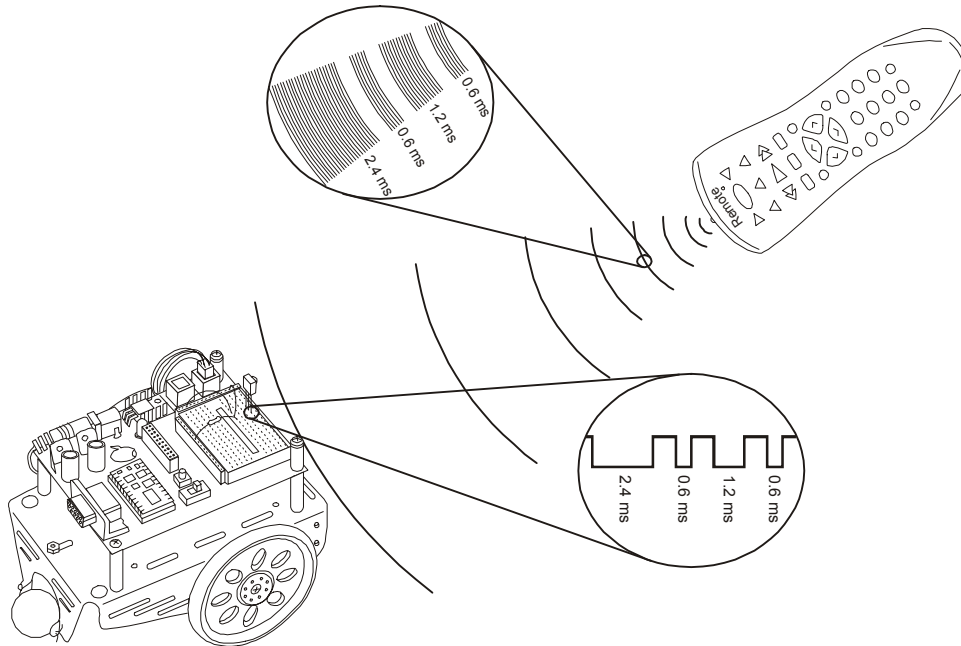
Handheld Remote Infrared Messages

Excerpt from IR Remote for the Boe-Bot text

The examples here will rely on the protocol universal remotes use to control SONY® TV sets. This protocol strobes the IR thirteen times with roughly half a millisecond rest between each pulse. It results in thirteen negative pulses from the IR detector that the BASIC Stamp can easily measure. The first pulse is the start pulse, which lasts for 2.4 ms. The next twelve pulses will either last for 1.2 ms (binary-1) or 0.6 ms (binary-0). The first seven data pulses contain the IR message that indicates which key is pressed. The last five pulses contain a binary value that specifies whether the message is intended being sent to a TV, VCR, CD, DVD player, etc. The pulses are transmitted in LSB-first order, so the first data pulse is bit-0, the next data pulse is bit-1, and so on. If you press and hold a key on the remote, the same message will be re-sent after a 20 to 30 ms rest.

Resting state
between message
packets = 20-30 ms

Resting states
between data pulses
= 0.6 ms

Start

0 0 0 0 0 1 0 1 0 0 0 0

Bit-0   Bit-2   Bit-4   Bit-6   Bit-8   Bit-10
   Bit-1   Bit-3   Bit-5   Bit-7   Bit-9   Bit-11

Start pulse
duration = 2.4 ms

Binary-0
data pulse
durations = 0.6 ms

Binary-1
data pulse
durations = 1.2 ms
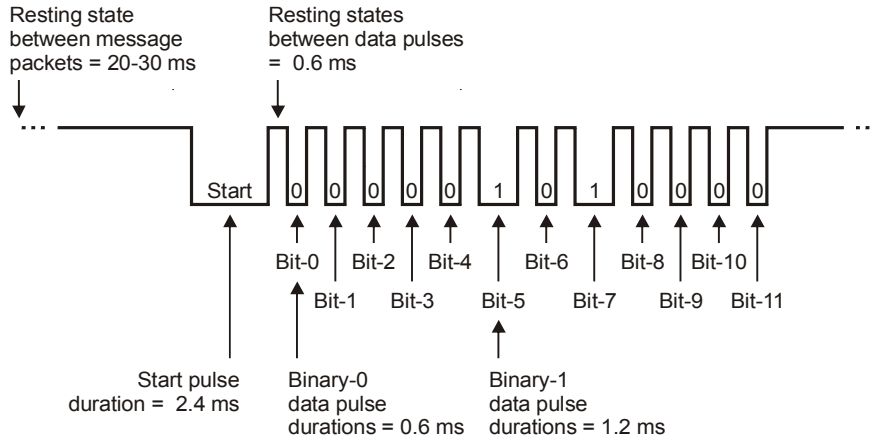
IR Message Timing Diagram

Values are approximate and will vary from one remote to the next.

IR Detection Circuit

For testing purposes, all you need is this IR detector circuit and the Debug Terminal.

Vdd

1
2
3

P9

220 Ω

Vss
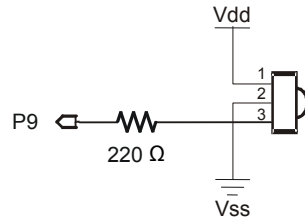
IR Detector Circuit

IR detector viewed from the top. Also see Kit Contents figure for pin map.

BASIC Stamp 2 "Bare-Bones" Example – IrRemoteCodeCapture.bs2

This example program demonstrates how to capture and display a remote code with the BASIC Stamp 2. If you modify the `$STAMP` directive, it can also be used with the BASIC Stamp 2e or 2pe.

√ Make sure to configure your universal remote to control a SONY® TV. Use the documentation that comes with your universal remote.
√ Press the TV button on your remote so that you know it is sending TV signals.
√ Download or hand enter and run IrRemoteCodeCapture.bs2.
√ Point the remote at the IR detector, and press/release the digit keys.
√ Also try POWER, CH+/-, VOL+/-, and ENTER to view the codes for these values.

```
' Ir Remote Application - IrRemoteCodeCapture.bs2
' Process incoming SONY remote messages & display remote code.

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

' SONY TV IR remote variables

irPulse         VAR     Word                  ' Stores pulse widths
remoteCode      VAR     Byte                  ' Stores remote code

DEBUG "Press/release remote buttons..."

DO                                            ' Main DO...LOOP

  Get_Pulses:                                 ' Label to restart message check

  remoteCode = 0                              ' Clear previous remoteCode

  ' Wait for resting state between messages to end.

  DO
    RCTIME 9, 1, irPulse
  LOOP UNTIL irPulse > 1000

  ' Measure start pulse.  If out of range, then retry at Get_Pulses label.

  RCTIME 9, 0, irPulse
  IF irPulse > 1125 OR irPulse < 675 THEN GOTO Get_Pulses

  ' Get Data bit pulses.

  RCTIME 9, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT0 = 1
  RCTIME 9, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT1 = 1
  RCTIME 9, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT2 = 1
  RCTIME 9, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT3 = 1
  RCTIME 9, 0, irPulse
  IF irPulse > 300 THEN remoteCode.BIT4 = 1
```

```
 RCTIME 9, 0, irPulse
 IF irPulse > 300 THEN remoteCode.BIT5 = 1
 RCTIME 9, 0, irPulse
 IF irPulse > 300 THEN remoteCode.BIT6 = 1

 ' Map digit keys to actual values.
 IF (remoteCode < 10) THEN remoteCode = remoteCode + 1
 IF (remoteCode = 10) THEN remoteCode = 0

 DEBUG CLS, ? remoteCode
LOOP                                     ' Repeat main DO...LOOP
```

How IrRemoteCodeCapture.bs2 Works

Each time through the outermost **DO…LOOP**, the value of **remoteCode** is cleared. There's also an inner **DO…LOOP** with an **RCTIME** command to detect the end of a high signal that lasted longer than 2 ms. This indicates that the rest between messages just ended, and the start pulse is beginning. The first **PULSIN** command captures the first data pulse, and the **IF…THEN** statement that follows uses the value of the **irPulse** variable to either set (or leave clear) the corresponding bit in the **remoteCode** variable. Since the next data pulse has already started while the **IF…THEN** statement was executing, the remainder of the next data pulse is measured with an **RCTIME** command. This next value is again used to either set (or leave clear) the next bit in **remoteCode**. This is repeated five more times to get the rest of the useful part of the IR message and set/clear the rest of the bits in **remoteCode**.

The BS2sx and BS2p handle remote codes a little differently. The programs usually search for the actual start pulse with a **PULSIN** command instead of searching for the resting state between messages. They also use **PULSIN** commands to capture all the pulses since the **IF…THEN** statements that sets bits in the **remoteCode** variable complete before the starting edge of the next data pulse. To see a code example that does this, see the **#CASE** statement for the BS2sx, BS2p inside the next example program's **Get_Ir_Remote_Code** subroutine.

BASIC Stamp 2 Series Application Example IrRemoteButtonDisplay.bs2

You can use this application example with BASIC Stamp 2, 2e, 2sx, 2p, or 2pe modules to test your remote and display which key you pressed.

√ As with the previous example program, make sure your remote is configured to control a SONY TV first.
√ Update the **$STAMP** Directive for the BASIC Stamp module you are using.
√ Download or hand enter, then run IrRemoteButtonDisplay.bs2.
√ Point the remote at the IR detector, press and release buttons.
√ Make sure the Debug Terminal reports the correct button. Start with digits, channel, volume, etc.

You can modify or expand the **SELECT…CASE** statement to test for VCR keys defined in the Constants section (Play, Stop, Rewind, etc.). There are usually several different codes for configuring universal remotes to control SONY VCRs, so you may need to try a few before finding the code that makes the remote speak the same PWM language as the TV controller. You can determine if the code worked because number, CH/VOL+/-, and POWER keys will still work after you have pressed the VCR button.

```
' -----[ Title ]-------------------------------------------------------
' Ir Remote Application - IrRemoteButtonDisplay.bs2

' Process incoming SONY remote signals and display the corresponding button
' in the Debug Terminal.

' {$STAMP BS2}                              ' BS2, 2sx, 2e, 2p, or 2pe
' {$PBASIC 2.5}

' -----[ Revision History ]-------------------------------------------

' V1.0 - Supports most SONY TV and VCR control buttons.
'        Supports BASIC Stamp 2, 2SX, 2e, 2p, and 2pe modules.

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

' SONY TV IR remote declaration - input receives from IR detector

IrDet          PIN    9                     ' I/O pin to IR detector output

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

' Pulse duration constants for SONY remote.

#SELECT $STAMP
  #CASE BS2, BS2E, BS2PE                    ' PULSE durations
    ThresholdStart CON 1000                 ' Message rest vs. data rest
    ThresholdEdge  CON 300                  ' Binary 1 vs. 0 for RCTIME
    OverStart      CON 1125                 ' Start pulse max
    UnderStart     CON 675                  ' Start pulse min
  #CASE BS2P, BS2SX
```

```
    ThresholdStart CON 2400                 ' Binary 1 vs. start pulse
    ThresholdPulse CON 500  * 5 / 2         ' Binary 1 vs. 0 for PULSIN
  #CASE #ELSE
    #ERROR This BASIC Stamp NOT supported.
#ENDSELECT

' SONY TV IR remote constants for non-keypad buttons

Enter          CON     11
ChUp           CON     16
ChDn           CON     17
VolUp          CON     18
VolDn          CON     19
Mute           CON     20
Power          CON     21
TvLast         CON     59                   ' AKA PREV CH

' SONY VCR IR remote constants

' IMPORTANT: Before you can make use of these constants, you must
' also follow the universal remote instructions to set your remote
' to control a SONY VCR.  Not all remote codes work,  so you may have to
' test several.

VcrStop        CON     24
VcrPause       CON     25
VcrPlay        CON     26
VcrRewind      CON     27
VcrFastForward CON     28
VcrRecord      CON     29

' Function keys

FnSleep        CON     54
FnMenu         CON     96

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

' SONY TV IR remote variables

irPulse        VAR     Word                 ' Stores pulse widths
remoteCode     VAR     Byte                 ' Stores remote code

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

DEBUG "Press/release remote buttons..."

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

' Replace this button testing DO...LOOP with your own code.
```

```
DO                                          ' Main DO...LOOP

  GOSUB Get_Ir_Remote_Code                  ' Call remote code subroutine

  DEBUG CLS, "Remote button: "              ' Heading

  SELECT remoteCode                         ' Select message to display
    CASE 0 TO 9
      DEBUG DEC remoteCode
    CASE Enter
      DEBUG "ENTER"
    CASE ChUp
      DEBUG "CH+"
    CASE ChDn
      DEBUG "CH-"
    CASE VolUp
      DEBUG "VOL+"
    CASE VolDn
      DEBUG "VOL-"
    CASE Mute
      DEBUG "MUTE"
    CASE Power
      DEBUG "POWER"
    CASE TvLast
      DEBUG "LAST"
    CASE ELSE
      DEBUG DEC remoteCode, " (unrecognized)"
  ENDSELECT

LOOP                                         ' Repeat main DO...LOOP

' -----[ Subroutine - Get_Ir_Remote_Code ]---------------------------------

' SONY TV IR remote subroutine loads the remote code into the
' remoteCode variable.

Get_Ir_Remote_Code:

  remoteCode = 0

  #SELECT $STAMP
    #CASE BS2, BS2E, BS2PE
      ' Wait for resting state between messages to end.
      DO
        RCTIME IrDet, 1, irPulse
      LOOP UNTIL irPulse > ThresholdStart
      ' Measure start pulse.  If out of range, then retry at
      ' Get_Ir_Remote_Code label.
      RCTIME 9, 0, irPulse
      IF irPulse > OverStart OR irPulse < UnderStart THEN Get_Ir_Remote_Code
      ' Get data bit pulses.
```

```
     RCTIME IrDet, 0, irPulse
     IF irPulse > ThresholdEdge THEN remoteCode.BIT0 = 1
     RCTIME IrDet, 0, irPulse
     IF irPulse > ThresholdEdge  THEN remoteCode.BIT1 = 1
     RCTIME IrDet, 0, irPulse
     IF irPulse > ThresholdEdge  THEN remoteCode.BIT2 = 1
     RCTIME IrDet, 0, irPulse
     IF irPulse > ThresholdEdge  THEN remoteCode.BIT3 = 1
     RCTIME IrDet, 0, irPulse
     IF irPulse > ThresholdEdge  THEN remoteCode.BIT4 = 1
     RCTIME IrDet, 0, irPulse
     IF irPulse > ThresholdEdge  THEN remoteCode.BIT5 = 1
     RCTIME IrDet, 0, irPulse
     IF irPulse > ThresholdEdge  THEN remoteCode.BIT6 = 1
  #CASE BS2SX, BS2P
     DO                                   ' Wait for start pulse.
       PULSIN IrDet, 0, irPulse
     LOOP UNTIL irPulse > ThresholdStart
     PULSIN IrDet, 0, irPulse              ' Get data pulses.
     IF irPulse > ThresholdPulse THEN remoteCode.BIT0 = 1
     PULSIN IrDet, 0, irPulse
     IF irPulse > ThresholdPulse THEN remoteCode.BIT1 = 1
     PULSIN IrDet, 0, irPulse
     IF irPulse > ThresholdPulse THEN remoteCode.BIT2 = 1
     PULSIN IrDet, 0, irPulse
     IF irPulse > ThresholdPulse THEN remoteCode.BIT3 = 1
     PULSIN IrDet, 0, irPulse
     IF irPulse > ThresholdPulse THEN remoteCode.BIT4 = 1
     PULSIN IrDet, 0, irPulse
     IF irPulse > ThresholdPulse THEN remoteCode.BIT5 = 1
     PULSIN IrDet, 0, irPulse
     IF irPulse > ThresholdPulse THEN remoteCode.BIT6 = 1
  #CASE #ELSE
     #ERROR "BASIC Stamp version not supported by this program."
 #ENDSELECT

 ' Map digit keys to actual values.
 IF (remoteCode < 10) THEN remoteCode = remoteCode + 1
 IF (remoteCode = 10) THEN remoteCode = 0

 RETURN
```

BASIC Stamp 2 Series Example - Multi-Digit Application

You can use the remote for keypad entry of values by replacing the **DO…LOOP** in
IrRemoteButtonDisplay.bs2's main routine with the one shown below. It works for
values from 0 to 65535; just type in the value on the digital keypad, then press the
remote's ENTER key.

√ Add this declaration to the IrRemoteButtonDisplay.bs2's Variables section:

```
value           VAR     Word                    ' Stores multi-digit value
```

√ Replace the **DO…LOOP** in IrRemoteButtonDisplay.bs2's main routine with the one shown below.

√ Run the program and follow the Debug Terminal's prompts.

```
' Replace the DO...LOOP in the Main Routine with this one for multi-
' digit value acquisition (up to 65535). Value stored in value
' variable.

DEBUG CR, CR, "Type value from", CR, "0 to 65535,", CR,
      "then press ENTER", CR, CR

DO
  value = 0
  remoteCode = 0
  DO
    value = value * 10 + remoteCode
    DO
      GOSUB Get_Ir_Remote_Code
      IF (remoteCode > 9) AND (remoteCode <> Enter) THEN
        DEBUG "Use digit keys or ENTER", CR
        PAUSE 300
      ELSE
        DEBUG "You pressed: "
        IF remoteCode = Enter THEN
          DEBUG "Enter", CR
        ELSE
          DEBUG DEC remoteCode, CR
        ENDIF
        PAUSE 300
      ENDIF
    LOOP UNTIL (remoteCode < 10) OR (remoteCode = Enter)
  LOOP UNTIL (remoteCode = Enter)
  DEBUG ? value, CR, "Ready for next value...", CR
LOOP
```

Boe-Bot Application for the BASIC Stamp 2

This next application requires a Boe-Bot robot with a BASIC Stamp 2 module which you will be able to control by pressing and holding the numeric keys to execute the maneuvers shown in the figure. In addition, you can use CH+ = forward, CH- = backward, VOL+ = rotate right, VOL- = rotate left.

**Numeric Keypad
Direction Control**
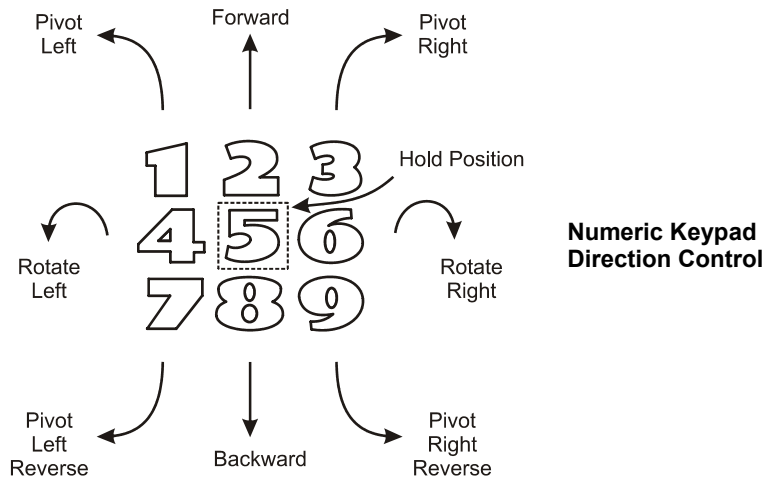
The routine below is for a Boe-Bot robot with Parallax Continuous Rotation servos. Its left servo should be connected to P13, and its right servo connected to P12. If you have Parallax PM servos, use 500 in place of 650 and 1000 in place of 850 for the **PULSOUT** command *Duration* arguments.

√   Replace the **DO…LOOP** in the IrRemoteButtonDisplay.bs2's main routine with this one, run it, and operate the Boe-Bot with your remote. Have fun!

```
DEBUG CR, CR, "Press and hold digit", CR, "or CH+/-, VOL+/- keys", CR,
      "to control the Boe-Bot..."

DO
  GOSUB Get_Ir_Remote_Code
  SELECT remoteCode
    CASE 2, ChUp                            ' Forward
      PULSOUT 13, 850
      PULSOUT 12, 650
    CASE 4, VolDn                           ' Rotate left
      PULSOUT 13, 650
      PULSOUT 12, 650
    CASE 6, VolUp                           ' Rotate right
      PULSOUT 13, 850
      PULSOUT 12, 850
    CASE 8, ChDn                            ' Backward
      PULSOUT 13, 650
      PULSOUT 12, 850
    CASE 1                                  ' Pivot Fwd-left
```

```
      PULSOUT 13, 750
      PULSOUT 12, 650
    CASE 3                                    ' Pivot Fwd-right
      PULSOUT 13, 850
      PULSOUT 12, 750
    CASE 7                                    ' Pivot back-left
      PULSOUT 13, 750
      PULSOUT 12, 850
    CASE 9                                    ' Pivot back-right
      PULSOUT 13, 650
      PULSOUT 12, 750
    CASE ELSE                                 ' Hold position
      PULSOUT 13, 750
      PULSOUT 12, 750
  ENDSELECT
LOOP
```

More Resources

These resources are available from www.parallax.com.

Lindsay, Andy. *IR Remote for the Boe-Bot*, **Student Guide, Version 1.0, California: Parallax, Inc., 2004.**
This book is discussed on the first page of this package insert.

Williams, Jon. *The Nuts and Volts of the BASIC Stamps*, **Volume 3, California: Parallax, Inc., 2003.**
Column #76: *Control from the Couch* introduces capturing and decoding SONY TV IR control signals with the BASIC Stamp 2SX (or 2p).

Appendix B: BS2 to BS2 IR Messages

This appendix/activity demonstrates how to send messages from one BASIC Stamp to another with IR and a custom protocol. This technique lends itself to IR communication between BASIC Stamp modules, Boe-Bot robots, etc.

Parts and Equipment

Since this activity involves beaming information from one BASIC Stamp to another with infrared, you will need at least two BASIC Stamp modules and two carrier boards. The parts listed below are for unidirectional communication. For bidirectional communication, you will need twice the number of 220 Ω resistors, IR receivers, and IR LEDs.

(2) Board of Education platforms with BASIC Stamp modules
(2) 220 Ω resistors (red-red-brown)
(1) IR receiver
(1) IR LED

> **If you have two Boe-Bot robots with their IR object detection circuits built-up, you're ready to go and can skip to the section entitled Developing the IR Communication Protocol.**

Transmit and Receive Circuits

Figure B-1 shows the transmitter board's IR transmit circuit and the receiver board's IR receive circuit.

√ Connect your IR LED and IR detector circuits to separate BASIC Stamp 2 modules as shown in Figure 1.7.

Figure B-1 IR Transmit and Receive Circuits

Receiver board circuit   **Transmitter board circuit**
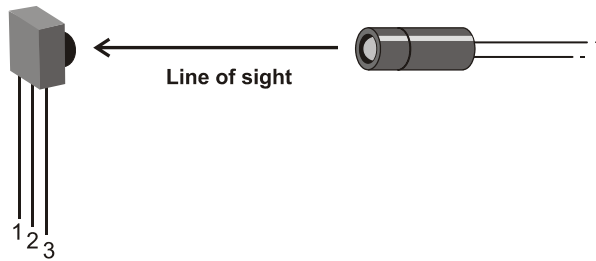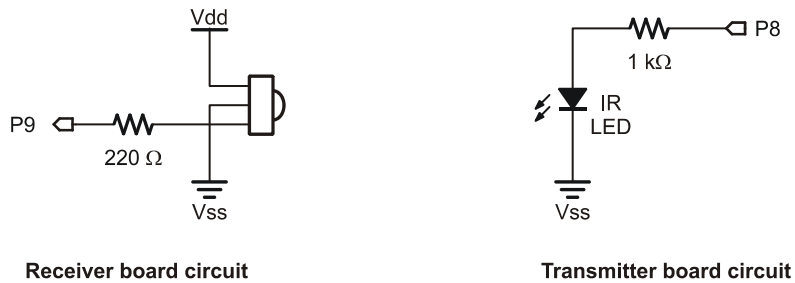
Developing the IR Communion Protocol

All it takes to transmit infrared with the BASIC Stamp 2 is an IR LED, a resistor, and the **FREQOUT** command. The one hitch to the **FREQOUT** command is that the BASIC Stamp will transmit the 38 kHz signal in time increments of 1 ms. With this in mind, **FREQOUT** won't work for the 2.4, 1.2, and 0.6 ms time increments required to transmit the SONY TV protocol signals.

> **i**   **See Appendix C if you want to emulate SONY remote functions with a BASIC Stamp.**
>
> Appendix C introduces a different circuit along with programming techniques you can use to transmit 38 kHz IR signals that last fractions of milliseconds, including 2.4, 1.2, and 0.6 ms.

Since the protocol we are developing in this activity is for communication between BASIC Stamp modules, the sensible solution is to chose pulse durations that last, 3, 2, or 1 ms and assign them as the start, binary-1 and binary-0 pulses. Also, since the BASIC Stamp variable sizes are bit, nibble, byte and word, the number of data bits transmitted

should be either 1, 4, 8, or 16. For this activity, we'll go with 8, but the number of data bits can be increased or decreased, simply by changing a couple lines of code. Here is how the protocol will work:

- Pulses transmitted are least significant bit (LSB) first.
- 8 data bits are transmitted
- A start bit is 3 ms
- A binary-1 is 2 ms
- A binary-0 is 1 ms
- The minimum delay between pulses is at least 1 ms + any loop processing overhead on the transmitter side.
- The maximum delay between pulses is 0.131 s.

TransmitCustomIrMessages.bs2 sends its messages as **FREQOUT** commands to an IR LED circuit. The IR LED flashes on/off at 38 kHz for either 3, 2, or 1 ms. If this 38 kHz IR flashing on/off shines on the IR receiver, it sends low signals while it detects that flashing IR.

As a result of the flashing IR, the IR receiver will send low signals that last either 3, 2, or 1 ms. The BASIC Stamp module on the receiving end, which runs ReceiveCustomIrMessages.bs2, measures these low signals with the **PULSIN** command. It then decodes these durations, storing the result as binary bits in a variable.

Testing IR Communication

TransmitCustomIrMessage.bs2 has some extra code in it that causes it to send a different letter of the alphabet each time the Reset button on the board is pressed and released. The sequence it will transmit is "a", "b", "c", ... "z". This program should be downloaded to the BASIC Stamp on the board with the transmit circuit. This board can then be disconnected from the serial cable. Then, the board with the receiver circuit can be connected (and left connected) to the serial cable while running ReceiveCustomIrMessages.bs2. After pointing the top of the IR LED at the face of the IR receiver (see Figure B-1), each time the transmit board's Reset button is pressed and released, the receiver board should display the character that was received and decoded in the Debug Terminal.

√ Connect the board with the IR transmitting circuit shown in Figure B-1 to the Programming cable.
√ Download TransmitCustomIrMessage.bs2 to the transmitting BS2.

√ Disconnected the board with the transmitting circuit from the programming cable.
√ Connect the board with the receiving circuit shown in Figure B-1 to the programming cable.
√ Download ReceiveCustomIrMessage.bs2 into it.
√ Leave it connected to the programming cable for debugging.
√ Point the top of the IR LED on the transmitter board directly at the face of the IR receiver on the receiver board as shown in Figure B-1.
√ Make sure the distance between the top of the IR LED and the face of the IR receiver is not more than 3 inches to begin with.
√ Press and release the reset button on the transmitter board several times.

Each time you press and release the transmitter board's Reset button, it will transmit the next character in the lower case alphabet. The receiver board should receive these letters and display them in the Debug Terminal as shown in Figure B-2.
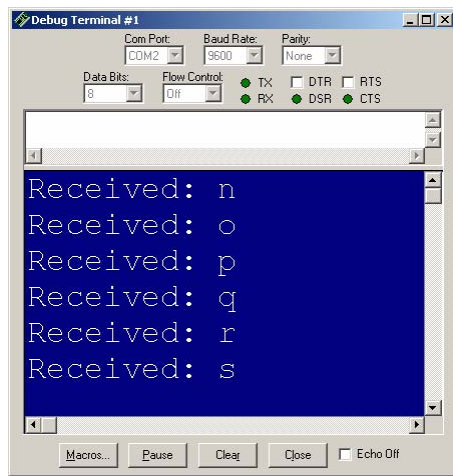
Figure B-2
Receiver Board
Display

Example Program: TransmitCustomIrMessage.bs2

```
' -----[ Title ]-------------------------------------------------------
' IR Remote for the Boe-Bot - TransmitCustomIrMessage.bs2
' Sends an IR message with the custom protocol developed in IR Remote
' with the Boe-Bot, Appendix B.

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

```
' -----[ EEPROM Data ]------------------------------------------------------

eeChars        DATA    "a"

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

IrLedPin      CON     8

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

IrFreq        CON     38000
StartBitTime  CON     3
Binary0Time   CON     2
Binary1Time   CON     1
BtwnPulses    CON     1

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

counter       VAR     Nib
irMessage     VAR     Byte
duration      VAR     Nib
character     VAR     Byte

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

LOW IrLedPin

READ eeChars, character
IF character < "a" OR character > "z" THEN character = "a"
character = character + 1
WRITE eeChars, character

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

irMessage = character - 1
GOSUB Transmit_Ir_Byte

DEBUG "Transmitted: ", irMessage, CR,
      "Press/release Reset button", CR

STOP

' -----[ Subroutine - Transmit_Ir_Byte ]------------------------------------

Transmit_Ir_Byte:

  FREQOUT IrLedPin, StartBitTime, IrFreq

  PAUSE BtwnPulses
```

```
  FOR counter = 0 TO 7

    IF irMessage.LOWBIT(counter) = 1 THEN
        duration = 2
      ELSE
        duration = 1
    ENDIF

    FREQOUT IrLedPin,duration,IrFreq
    PAUSE BtwnPulses

  NEXT
  RETURN
```

How TransmitCustomIrMessage.bs2 Works

Here is a hard-coded example of sending the number 9 with the circuit shown in Figure B-1 and the protocol introduced in this activity. The number 9 in binary is %00001001. Since the protocol sends the bits LSB-first, it means the rightmost digit is sent first, then the second digit from the right, and so on until the leftmost of the eight digits is sent. The Duration argument in the each of the **FREQOUT** commands controls how long each it transmits 38 kHz. Each duration is either 3 (3 ms for the start pulse), 2 (2 ms for a binary-1) or 1 (1 ms for a binary-0). A **PAUSE** 1 between each **FREQOUT** command ensures that the receiving BASIC Stamp will have time to process the bit and move on to the next.

```
        FREQOUT 8, 3, 38000       ' Start pulse
        PAUSE 1
        FREQOUT 8, 2, 38000       ' Bit-0 = binary-1
        PAUSE 1
        FREQOUT 8, 1, 38000       ' Bit-1 = binary-0
        PAUSE 1
        FREQOUT 8, 1, 38000       ' Bit-2 = binary-0
        PAUSE 1
        FREQOUT 8, 2, 38000       ' Bit-3 = binary-1
        PAUSE 1
        FREQOUT 8, 1, 38000       ' Bit-4 = binary-0
        PAUSE 1
        FREQOUT 8, 1, 38000       ' Bit-5 = binary-0
        PAUSE 1
        FREQOUT 8, 1, 38000       ' Bit-6 = binary-0
        PAUSE 1
        FREQOUT 8, 1, 38000       ' Bit-7 = binary-0
```

TransmitCustomIrMessage.bs2 has a subroutine that makes transmitting IR bytes much easier and more versatile than hard coding. All you have to do is set the **irMessage** variable to the value you want to send, and then call the **Transmit_Ir_Byte** subroutine.

B

For example, to send the value 9, simply set the **irMessage** variable to 9, then call **Transmit_Ir_Byte**.

```
irMessage = 9
GOSUB Transmit_Ir_Byte
```

The **Transmit_Ir_Byte** subroutine examines each bit in the **irMessage** variable, and selects the corresponding Duration for the **FREQOUT** command. To transmit the bits LSB-first, the subroutine uses a **FOR...NEXT** loop that increments the counter variable from 0 to 7.

```
Transmit_Ir_Byte:

  FREQOUT IrLedPin, StartBitTime, IrFreq

  PAUSE BtwnPulses

  FOR counter = 0 TO 7

    IF irMessage.LOWBIT(counter) = 1 THEN
       duration = 2
     ELSE
       duration = 1
    ENDIF

    FREQOUT IrLedPin,duration,IrFreq
    PAUSE BtwnPulses

  NEXT

  RETURN
```

Inside the **Transmit_Ir_Byte** subroutine's **FOR...NEXT** loop, an **IF...THEN** statement examines **irMessage.LOWBIT(counter)**. Assuming that the subroutine is transmitting a 9, the first time through the **FOR...NEXT** loop, counter will be 0, so **irMessage.LOWBIT(0)** will be 1. The second time through the loop, counter will be 1, and **irMessage.LOWBIT(1)** is 0. The third time through, **irMessage.LOWBIT(2)** is 0; the fourth time through **irMessage.LOWBIT(3)** is 1, and so on... In each case, an **IF...THEN** statement inside the loop sets the duration variable to 2 if **irMessage.LOWBIT(counter)** stores a 1, or 1 if **irMessage.LOWBIT(counter)** stores a 0.

Before each repeat of the **FOR...NEXT** loop, the command **FREQOUT IrLedPin, duration, IrFreq** sends the 38 kHz signal for the number of ms stored in the duration

variable. Also, **PAUSE BtwnPulses** delays the program for 1 ms. (**BtwnPulses** is defined as 1 in the constant declarations.) This ensures at least a 1 ms delay between data bit pulses.

Example Program: ReceiveCustomIrMessage.bs2

```
' -----[ Title ]----------------------------------------------------------
' IR Remote for the Boe-Bot - ReceiveCustomIrMessage.bs2
' Receives and decodes IR message with the custom protocol developed
' in IR Remote with the Boe-Bot, Appendix B.

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

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

IrDetPin        PIN     9

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

StartMax        CON     1700
StartMin        CON     1300

Bin1Max         CON     1200
Bin1Min         CON     800

Bin0Max         CON     700
Bin0Min         CON     300

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

counter         VAR     Nib
irPulse         VAR     Word
duration        VAR     Nib
irMessage       VAR     Byte

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

DO: LOOP UNTIL IrDetPin = 1

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

DO

  GOSUB Get_IR_Byte_Message
  DEBUG "Received: ", irMessage, CR

LOOP

' -----[ Subroutine - Get_Ir_Byte_Message ]------------------------------
```

B

```
Get_Ir_Byte_Message:

  DO
    PULSIN IrDetPin, 0, irPulse
  LOOP UNTIL irPulse < StartMax AND irPulse > StartMin

  FOR counter = 0 TO 7

    PULSIN IrDetPin, 0, irPulse

    SELECT irPulse
      CASE Bin1Min TO Bin1Max
        irMessage.LOWBIT(counter) = 1
      CASE Bin0Min TO Bin0Max
        irMessage.LOWBIT(counter) = 0
      CASE ELSE
        DEBUG "Pulse out of range, ", CR,
              "trying again...", CR
        GOTO Get_Ir_Byte_Message
    ENDSELECT

  NEXT

  RETURN
```

How ReceiveCustomIrMessage.bs2 Works

The Get_Ir_Byte_Message subroutine does the decoding by measuring each low pulse it gets from the IR receiver and deciding whether it's a start, binary-1, or binary-0 pulse based on its duration.

```
Get_Ir_Byte_Message:

  DO
    PULSIN IrDetPin, 0, irPulse
  LOOP UNTIL irPulse < StartMax AND irPulse > StartMin

  FOR counter = 0 TO 7

    PULSIN IrDetPin, 0, irPulse

    SELECT irPulse
      CASE Bin1Min TO Bin1Max
        irMessage.LOWBIT(counter) = 1
      CASE Bin0Min TO Bin0Max
        irMessage.LOWBIT(counter) = 0
      CASE ELSE
        DEBUG "Pulse out of range, ", CR,
```

```
                    "trying again...", CR
               GOTO Get_Ir_Byte_Message
             ENDSELECT

      NEXT

      RETURN
```

Like the example programs in Chapters 2 and 3, there's a `DO...LOOP` at the beginning of the subroutine waits for the start pulse. Remember that `PULSIN` measures in 2 µs units. So, a 3 ms start pulse should result in a `PULSIN` measurement of 1500. In order to wait for this start pulse, the loop repeats `UNTIL irPulse < StartMax AND irPulse > StartMin`. `StartMax` and `StartMin` are constants defined to be 1700 and 1300, which gives ample room for variations in IR receiver output-low time, probably much more than can occur due to ambient light conditions.

After the start pulse is received, a `FOR...NEXT` loop decodes the eight bit-pulses that follow. Each time through the `FOR...NEXT` loop, a `PULSIN` command measures the next low pulse. A `SELECT...CASE` block decides whether the pulse is 1, 0, or out of range. For example, `CASE Bin1Min TO Bin1Max` stores a 1 in `irMessage.LOWBIT(counter)` if the `PULSIN` command stored a value in the `irPulse` variable that is in the range from `Bin1Min` (a constant defined to be 800) to `Bin1Max` (a constant defined to be 1200). As with the start pulse, this `CASE` statement is filtering for a binary-1 low signal that lasts 2 ms. The second case statement checks to see if `irPulse` is 500, which is a 1 ms pulse. The range it accepts is from `Bin0Min = 300` to `Bin0Max = 700`.

If neither those `CASE` statements are true, the message might be the result of outside interference. Since the message is unreliable, the `CASE ELSE` statement sends the program back to the beginning of the subroutine to try again. On the other hand, if the `FOR...NEXT` loop repeated eight times, the `CASE` statements will either have stored 1 or 0 in the eight bits in the `irMessage` byte variable, and the result will be ready for the program to use.

Your Turn - Characters vs. Numeric Byte and Word Values

Since this protocol sends bytes, you can also transmit values between 0 and 255. For transmitting a sequence of numbers, the TransmitCustomCharacters.bs2 program can be updated as follows:

√   Save TransmitCustomIrMessage.bs2 as TransmitCustomIrValues.bs2.
√   Modify this line so that the **DATA** directive initializes the value stored at the eeChar address to 0. In other words, change

```
eeChars       DATA   "a"
```

to

```
eeChars       DATA   0
```

√   Comment this line by placing an apostrophe to the left of it.

```
' IF character < "a" OR character > "z" THEN character = "a"
```

√   Optionally, replace all instances of character with value. This is most easily done with the BASIC Stamp Editor's Edit -> Find/Replace menu option.
√   Save and download the modified program into the BASIC Stamp on the transmitter board, then disconnect it from the programming cable.

All that has to be changed in ReceiveCustomIrMessage.bs2 is the **DEBUG** command that displays the value it receives.

√   Save ReceiveCustomIrMessage.bs2 as ReceiveCustomIrValues.bs2.
√   Add the **DEC** formatter to this **DEBUG** command

```
DEBUG "Received: ", irMessage, CR
```

so that it looks like this

```
DEBUG "Received: ", DEC irMessage, CR
```

√   Save the program, then downloaded it to the BASIC Stamp on the receiver board, and leave it connected to the programming cable.
√   With line of sight and a close proximity established between the IR LED and the IR receiver, repeatedly press and release the transmitter board's reset button.

The receiver board should display repeated messages in the Debug Terminal with the sequence "Received: 0", "Received: 1", "Received: 2", etc.

The protocol can be updated to transmit and receive word values instead of byte values. The **irMessage** and **character** (or value if you used the BASIC Stamp Editor's search/replace feature) variables need to be declared as Word instead of Byte. The

`FOR...NEXT` loops that transmit and receive and decode the bit pulses need to have their *EndValue* arguments changed from 7 to 15. The transmitting program will have to use the Word formatter before the *DataItem* arguments in the `DATA` directive as well as before the variable values in the `READ` and `WRITE` commands. That should make it possible to transmit and receive values from 0 to 65535.

Appendix C: Transmitting IR Remote Signals with the BASIC Stamp 2

The activities in this text focused on programming the BASIC Stamp to receive and decode IR remote signals. If you are also interested in encoding and transmitting the same signals an IR remote would send, this appendix/activity provides a circuit and example programs that can actually be used in place of the universal remote to send SONY TV control commands to the Boe-Bot.

Other Protocols

Several IR remote protocols can be transmitted by the BASIC Stamp 2. Typically, if the 38 kHz transmit times and the delays between transmit times are greater than or equal to 0.6 ms, it's possible to write a PBASIC program to make the BASIC Stamp 2 do the job. The challenge is typically writing code to control the transmit and delay times. This code has to be executed during the delays, and it contributes to the delays as well. If the code takes longer to do the calculations than the protocol allows for the delays, it may be necessary to use a different microcontroller (see Better Tools and Other Protocols, below).

There is usually more than one way to write code that calculates the 38 kHz transmit times, and it can make a big difference in the delays between transmits. PULSOUT commands to unconnected pins can further tune the delay times when they are critical to the protocol. This kind of tuning involves monitoring the signals the BASIC Stamp sends with an oscilloscope. (See Understanding Signals, which has a section on analyzing the SONY protocol with the Parallax USB Oscilloscope. Both Understanding Signals and the Parallax USB Oscilloscope are available at www.parallax.com)

Better Tools for Other Protocols

Although several different protocols can be mimicked by the BASIC Stamp 2 with the help of the 555 timer circuit introduced in this activity, it really isn't the best tool for the job. It's kind of like using a hammer and a nail to make a hole. It would be better to just use a drill. Microcontrollers like the SX and Propeller are much better suited to precisely timing signals, and they do not need an external 555 timer circuit. They are also reasonable next steps after you have become comfortable with BASIC Stamp programs and circuits. Programs for the Propeller and SX microcontrollers typically monitor an internal timer and turn the 38 kHz signals on/off at the required time intervals.

For more information about the SX and Propeller microcontrollers, go to www.parallax.com.

Pulse Controlled 38 kHz Transmitter Parts

Next to the circuit from the previous appendix, the parts for the circuit in this appendix are arguably the second least expensive form of wireless communication between BASIC Stamps. Some of the parts listed here are not included in the Boe-Bot or IR Remote for

the Boe-Bot kits. However, the missing parts are inexpensive, easy to obtain, and they can also be ordered from Parallax. The Parallax part numbers for the extra parts are included in the parts list below.

(1) Capacitor - 0.01 μF
(1) Resistor - 470 Ω (Yellow, Violet, Brown)
(2) Resistors - 220 Ω (Red, Red, Brown)
(1) IR LED
(6) Jumper wires

(1) Potentiometer - 10 kΩ
    Parallax Part#: 152-01031
(1) NE555N timer IC
    Parallax Part#: 604-00009

Pulse Controlled 38 kHz Transmitter Circuit

Figure C-1 shows the IR transmit circuit. The 10 kΩ potentiometer in this circuit has to be adjusted so that it transmits 38 kHz. The BASIC Stamp can be programmed to monitor and report the frequency transmitted by the 555 timer's Out pin with P7. When the correct transmit frequency is established, the BASIC Stamp can then be programmed to control the durations the 555 timer transmits 38 kHz by sending **PULSOUT** signals to the 555 timer's Res pin with P6.

√   If you are using the 10 kΩ potentiometer with the Parallax Part#: 152-01031, pinch each of its legs with a needle-nose pliers to remove the kinks and straighten them before inserting the part into the breadboard.
√   Build the circuit shown in Figure C-1 on a second board, which you will use to beam IR remote messages to your board with a receiver circuit (or a Boe-Bot with IR object detection circuits).

> **To learn more about designing 555 circuits** for transmitting various frequencies and duty cycles, download the *Basic Analog and Digital* PDF from www.parallax.com, and read the first five pages of Chapter #6.

Figure C-1
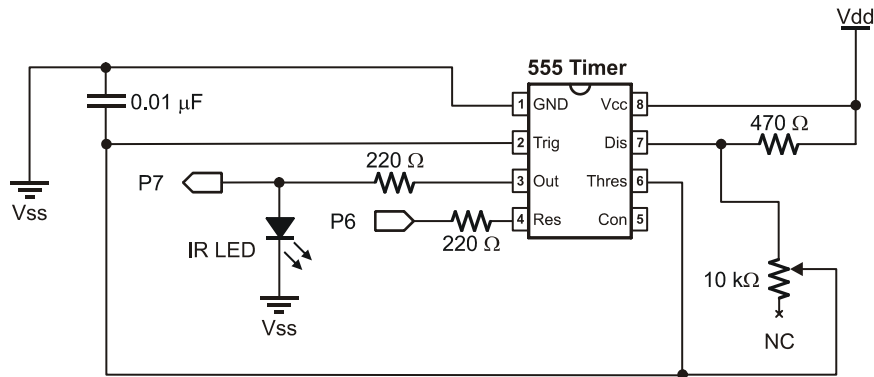
Tuning and Testing the 38 kHz Transmit Circuit

Test555Frquency.bs2 sets P6 high to enable the 555 timer, at which point its Out pin will start sending high/low signals. P7 is connected to the 555 timer's Out pin, and the command **COUNT 7, 100, cycles** stores the number of times it sends the high/low signal in 100 ms. The command **cycles = cycles * 10** gives adjusts the value to the number of cycles per second, which is then displayed with a **DEBUG** command. The potentiometer can then be adjusted for a target 555 timer output frequency of 38 kHz.

√ Enter, save, and run Test555Frequency.bs2
√ Adjust the potentiometer knob with a screwdriver until the Debug Terminal reports a frequency in the 37 to 39 kHz range (your target frequency is 38.0 kHz).

Example Program: Test555Frquency.bs2

```
' IR Remote for the Boe-Bot - Test555Frequency.bs2
' Displays 555 timer frequency for potentiometer adjustment and tuning.

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

cycles VAR Word

DO

  HIGH 6
  COUNT 7, 100, cycles
```

```
  LOW 6
  cycles = cycles * 10
  DEBUG HOME, "Frequency = ", DEC5 cycles, " Hz"

LOOP
```

Transmitting the SONY IR Remote Signal

Since high/low signals to the 555 timer's Res pin turn the 555 timer's signal on/off, you can now use the **PULSOUT** command to send 38 kHz signals for precise amounts of time. For example, to send a 2.4 ms start pulse, you can use the command **PULSOUT 6, 1, 1200**. To send a 1.2 ms binary-1 pulse, you can use the command **PULSOUT 6, 1, 600**, and to send a 0.6 ms binary-0 pulse, the command **PULSOUT 6, 1, 300** does the job.

TransmitIrRemoteButtons.bs2 has a subroutine that automates transmitting IR remote button information. Simply set the remoteCode variable to the button or value you want to transmit, then call the program's **Send_Ir_Remote_Code** subroutine like this.

```
        remoteCode = ChUp
        GOSUB Send_Ir_Remote_Code
```

In practice, it's best to send, between 5 to 10 copies of the same code, since that's what happens when you press and release a button on the remote. So, your code might actually look like this.

```
        FOR counter = 1 to 5
          remoteCode = ChUp
          GOSUB Send_Ir_Remote_Code
        NEXT
```

In Chapter 1, you calculated the number of times per second an IR remote repeats its message. Depending on the remote and the code that was sent, it was probably in the neighborhood of 40 to 50 ms per message. This can be considered the period (T) of the message cycle. The frequency (f) of messages is f = 1/T, and that's the number of messages sent per second the remote sends if you hold your finger on one of the remote's buttons for one second.

$$f_{messages} = 1 \div 0.045 \text{ s} = 22.2.. \text{ Hz}$$

TransmitIrRemoteButtons.bs2 emulates the signal coming from a remote for the sequence of key presses listed below.

- CH+ for 2 seconds
- VOL+ for 1 second
- VOL- for 1 second
- CH- for 2 seconds
- Power for 0.45 seconds
- 3 for 0.45 seconds

Example Program: TransmitRemoteButtons.bs2

The goal with this program is to send commands to a Boe-Bot that is running IrMultiBot.bs2 from Chapter 3, Activity #2. Then, when you run TransmitRemoteButtons.bs2 while pointing the IR LED at the Boe-Bot's IR receiver while maintaining line of sight and close proximity, it will make the Boe-Bot do the following.

- Forward for 2 seconds
- Rotate right for 1 second
- Rotate left for 1 second
- Backward for 2 seconds
- Power for 1/2 a second followed by 3 for 1/2 a second switches the Boe-Bot to mode-3, object following.

> **ⓘ** **If you have just a BASIC Stamp board with the receiver circuit** (but not a Boe-Bot), download IrRemoteButtons.bs2 to your receiver board instead of IrMultiBot.bs2. Then, use the Debug Terminal to display the messages the receiver board receives.

√ Connect your board (or a second Boe-Bot) with the 555 timer circuit to the programming cable.
√ Enter, save, and run TransmitIrRemoteButtons.bs2.
√ Disconnect your transmit board from the programming cable.
√ Connect your receiver board to the serial cable.

If your receiver board is on a Boe-Bot chassis:

√ Open IrMultiBot.bs2 with the BASIC Stamp Editor, download it to your Boe-Bot and disconnect the serial cable.
√ Make sure the 3-position switch on the Boe-Bot that will receive messages is set to 2.

√ Press/release the reset button on the Boe-Bot that will receive messages.
√ Establish line of sight and close proximity between the top of the transmitting IR LED and the face of the IR receiver.
√ Press and release the Reset button on your transmitter board.
√ The Boe-Bot should perform the maneuvers and mode changes discussed earlier. As it turns, you will probably need to adjust the position of your transmitter board to maintain line of sight.

If your receiver board is not on a Boe-Bot chassis:

√ Open IrRemoteButtons.bs2 with the BASIC Stamp Editor, download it to your receiver board, and leave the serial cable connected.
√ Establish line of sight and close proximity between the top of the transmitting IR LED and the face of the IR receiver.
√ Press and release the Reset button on your transmitter board.
√ The Debug Terminal should display the various codes for the amounts of time listed earlier, except for the last code, 3, which will persist until you press/release the transmitter board's reset button again.

```
' -----[ Title ]-----------------------------------------------------
' IR Remote for the Boe-Bot - TransmitIrRemoteButtons.bs2
' Transmit SONY TV protocol IR remote codes to a Boe-Bot using a second
' Board of Education and BASIC Stamp with a 555 timer circuit to supply
' the 38 kHz carrier signal to an IR LED.  The BASIC Stamp's P6 I/O pin
' is connected to the 555 timer's Res pin, which shuts off the signal
' with a low signal and turns it back on with a high signal.

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

' -----[ EEPROM Data ]-----------------------------------------------

' Button press and number of times each message should be re-sent.

Buttons DATA ChUp, VolUp, VolDn, ChDn, Power,   3, 255
Reps    DATA   44,    22,    22,   44,    10,  10, 0

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

tPin          PIN    6                     ' Transmit pin
fPin          PIN    7                     ' Frequency sense pin

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

' SONY TV IR remote constants for non-keypad buttons
```

```
Enter          CON    11                    ' Enter
ChUp           CON    16                    ' Channel +
ChDn           CON    17                    ' Channel -
VolUp          CON    18                    ' Volume +
VolDn          CON    19                    ' Volume -
Power          CON    21                    ' Power on/off

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

' SONY TV IR remote variables

remoteCode     VAR    Word                  ' Stores remote code
cycles         VAR    Word                  ' Stores 555 timer frequency

' EEPROM and message repetition counting variables

index          VAR    Byte                  ' EEPROM index
messageCnt     VAR    Byte                  ' Number of IR message reps
counter        VAR    Byte                  ' General purpose counter

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

  HIGH 6                                    ' Enable 555 timer
  COUNT 7, 1000, cycles                     ' Measure frequency
  LOW 6                                     ' Disable 555 timer
  DEBUG HOME, "Frequency = ", DEC5 cycles,    ' Display frequency
        " Hz", CR,   "Transmitting..."

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

DO                                          ' Main loop
  READ Buttons + index, remoteCode          ' EEPROM button -> remoteCode
  READ Reps + index, messageCnt             ' EEPROM reps -> messageCnt
  IF remoteCode = 255 THEN EXIT             ' remoteCode = 255? Exit loop
  FOR counter = 1 TO messageCnt             ' Repeat messageCnt times
    GOSUB Send_Ir_Remote_Code               ' Send current remoteCode
  NEXT
  index = index + 1                         ' Increment EEPROM index
LOOP                                        ' Repeat main loop

DEBUG CR, "Done!"                           ' Desplay "Done!" message

END                                         ' BASIC Stamp -> Sleep mode

' -----[ Subroutine - Send_Ir_Remote_Code ]---------------------------------

' Sends pulses to 555 timer that causes the IR LED connected to the Out
' pin to send 38 kHz signals of about the same durations as an IR
' remote set to send SONY TV signals.
```

```
Send_Ir_Remote_Code:

  ' Change button values to remote codes.

  IF (remoteCode = 0) THEN remoteCode = 10
  IF (remoteCode <= 10) THEN remoteCode = remoteCode - 1



  remoteCode.BIT7 = 1                               ' Set bit-7 = 1 for TV

  LOW tPin                                          ' 555 signal off
  PAUSE 23                                          ' Rest between messages
  PULSOUT tPin, 1200                                ' Start pulse

  PULSOUT tPin, remoteCode.BIT0 * 300 + 300   ' Bit-0 pulse
  PULSOUT tPin, remoteCode.BIT1 * 300 + 300   ' Bit-1 pulse
  PULSOUT tPin, remoteCode.BIT2 * 300 + 300   ' Bit-2 pulse
  PULSOUT tPin, remoteCode.BIT3 * 300 + 300   ' Etc...
  PULSOUT tPin, remoteCode.BIT4 * 300 + 300
  PULSOUT tPin, remoteCode.BIT5 * 300 + 300
  PULSOUT tPin, remoteCode.BIT6 * 300 + 300
  PULSOUT tPin, remoteCode.BIT7 * 300 + 300
  PULSOUT tPin, remoteCode.BIT8 * 300 + 300
  PULSOUT tPin, remoteCode.BIT9 * 300 + 300
  PULSOUT tPin, remoteCode.BIT10 * 300 + 300
  PULSOUT tPIn, remoteCode.BIT11 * 300 + 300

  RETURN
```

How TransmitIrRemoteButtons.bs2 Works

The **Send_Ir_Remote_Code** subroutine in TransmitIrRemoteButtons.bs2 is shown below. It sets tPin (P6) low to turn off the 555 timer's signal. The command **remoteCode.BIT7 = 1** sets bit-7 in the remote code variable to 1, which is what it always is for SONY TV messages. Then, it pauses for 23 ms. This, plus the processing time for the **READ** commands causes a typical message to repeat every 45 ms. After that, **PULSOUT tPIn, 1200** sends the start pulse. Next, a series of **PULSOUT** commands calculates and sends the pulses of the correct durations for binary-1 and 0. For example, if **remoteCode.BIT3** is 1, the command **PULSOUT tPin, remoteCode.BIT3 * 300 + 300** sends a **PULSOUT** of 600, which is 1.2 ms, which is a binary 1 pulse. However, if **remoteCode.BIT3** is 0, then the **PULSOUT** command only sends a pulse of 300, which lasts 0.6 ms, which transmits a binary-0 pulse.

C

```
Send_Ir_Remote_Code:
  ' Change button values to remote codes.
  IF (remoteCode = 0) THEN remoteCode = 10
  IF (remoteCode <= 10) THEN remoteCode = remoteCode - 1

  remoteCode.BIT7 = 1                        ' Set bit-7 = 1 for TV

  LOW tPin                                   ' 555 signal off
  PAUSE 26                                   ' Rest between messages
  PULSOUT tPin, 1200                         ' Start pulse

  PULSOUT tPin, remoteCode.BIT0 * 300 + 300  ' Bit-0 pulse
  PULSOUT tPin, remoteCode.BIT1 * 300 + 300  ' Bit-1 pulse
  PULSOUT tPin, remoteCode.BIT2 * 300 + 300  ' Bit-2 pulse
  PULSOUT tPin, remoteCode.BIT3 * 300 + 300  ' Etc...
     .
     .
     .
  RETURN
```

Each **PULSOUT** command in the **Send_Ir_Remote_Code** subroutine has an argument that multiplies one of the bits in the remoteCode variable by 300 before adding 300. Without parenthesis, the order of operation is left to right. If **remoteCode.BIT2** = 0 then the result of **remoteCode.BIT2 * 300 + 300** is 0 * 300 + 300 = 300. **If remoteCode.BIT3** is 1, then the result of **remoteCode.BIT3** is 1 * 300 + 300 = 300 + 300 = 600. More generally, the result of these expressions in each **PULSOUT** command's duration argument is 600 if the given bit is 1, or 300 if it's 0, which causes the **PULSOUT** command to turn the 555 timer's 38 kHz signal on, either for a duration of 1.2 ms or 0.6 ms.

Even though the same results can be obtained by the **FOR...NEXT** loop below, it won't work in the program because of signal timing considerations. Unlike the **remoteCode.BITX * 300 + 300** calculation, which takes around 500 μs, the code block below takes well over a millisecond. While a SONY TV set might or might not be forgiving enough to decode the signal, the example programs from this text that receive and decode IR messages are not that forgiving.

```
FOR index = 0 TO 11
  IF remoteCode.LOWBIT(index) = 1 THEN
    duration = 600
  ELSE
    duration = 300
  ENDIF
  PULSOUT tPin, duration
NEXT
```

The reason the way the code is written makes such a big difference in IR remote transmission is because the calculations are performed between **PULSOUT** commands. The **FOR...NEXT** loop shown above, which takes more than a millisecond, throws the timing off so much, that this text's example programs can no longer decode the messages. On the other hand, the calculation **remoteCode.BITX * 300 + 300** takes about 500 μs, which is really close to the 600 μs rests between pulses shown in Figure 1-4 on page 6. It works, both with the Boe-Bot and SONY TVs.

Index

Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

[Parallax](#):
[28139](#) [70016](#)