

Thinking Robotics: Teaching Robots to Make Decisions

Jeffrey R. Peters and Rushabh Patel

Adapted From *Robotics with the Boe-Bot* by Andy Lindsay, Parallax, inc., 2010

Preface

This manual was developed as a result of an accelerated high-school course in robotics. The goal of the course is to provide an introduction to robotics, and more specifically control systems. The material was designed so that it could be taught over 5 days, however, it can be easily extended to a much longer timeline in which students probe each section in more detail.

Since the course was designed for an accelerated pace, there are several concepts in the manual which are assumed to be known either by the student or the individual(s) teaching the course. In the following we outline what we feel the student and teacher should know in order to get the most out of this text. For individual use, it is necessary to be familiar with both the student and teacher portions. Students using this text should have an understanding of basic numerical computations and logic statements, (e.g., does a variable x satisfy either $x < 2$ or $x > 4$?). They should also have a very basic understanding of circuits; more precisely, an awareness of circuit components (batteries, wires, electrical components) and a general idea of how circuits work (currents flow through wires, batteries generate energy, components dissipate energy, etc.). A general familiarity with computers is also helpful. Teachers should have a basic understanding of the physics behind electricity and circuits, namely, the definitions and physical meanings of electrical potential, current, energy, and an understanding of fundamental equations such as Ohm's Law. Teachers should also have had hands-on experience with building circuits and using breadboards; students will need help troubleshooting their circuits and the teachers should be familiar enough with the hardware to help them. Similarly, teachers should have some basic programming experience, namely, knowledge of logic statements, IF/ELSE statements, FOR loops, WHILE loops, and other elementary programming tools. Knowledge of advanced programming is not necessary, but once again, students will need help debugging their codes, and the teacher should be comfortable enough with the material to help. Finally, since the focus of the class is on the control of robots, a knowledge of control systems concepts, such as feedback and simple controllers (proportional gain), is helpful.

The manual was designed to be used with the Boe-Bot kit distributed by Parallax, Inc., a simple programmable robotic platform designed to illustrate basic robotic concepts. Specifically, activities contained herein were based mainly on content from *Robotics with the Boe-Bot* version 3.0 by Andy Lindsay [1], which was written for use with the BASIC Stamp Editor ver. 2.5 and the USB version of the Boe-Bot robot kit. The kit that was available at the time of writing this manual contained the Board of Education Rev. D and the BASIC Stamp 2 micro controller Rev. J. We note that small modifications to the text may be necessary if using a different version of the Boe-Bot robot. We also note that it may be possible to perform the activities herein with other similar robotic platforms, such as the Arduino based Robotic Shield from Parallax, inc., under appropriate modifications.

Using the Boe-Bot robot, we focus mainly on the software, control theory and implementation, however, we also discuss some of the hardware aspects of robotics when relevant. In other words, we mostly concern ourselves with the issue of using the mea-

surements that we get from a robot to accomplish a desired task. We cover how to build simple circuits and write codes to make the robot perform a variety of tasks, including obstacle and light detection, line following, and other motion routines. We break up the tasks and present each as its own activity. Each activity focuses on a different sensor, including physical sensors, phototransistors, and infrared headlights. The final activity is open ended and intended to test how familiar the reader has become with their newly acquired skills in robotics. Most of the topics covered in this manual are also discussed in the text *Robotics with the Boe-Bot*, available at <http://www.parallax.com/>, which is an excellent reference to get further information and clarifications.

We are grateful to Parallax, Inc. for generously allowing us to make this material publicly available. We would like to thank Professor Francesco Bullo, Wendy Ibsen, and Stephanie Lindsay for their support in this endeavor. Finally, we would like to thank Heather Vermilyea, one of our students, for generously volunteering her time in reviewing and improving this manual.

Contents

Introduction	1
Activity 1: Setting Up the Hardware	2
1.1 Testing for communication	2
Activity 2: Your First Program	5
2.1 Sending a Message to the Boe-Bot	5
2.2 How the Code Works	7
Activity 3: Your Boe-Bot's Servo Motors	8
3.1 Connecting the Servo Motors	8
3.2 Centering the Servos	9
3.3 Testing the Servos	12
3.4 Additional Testing	14
3.5 How "BothServosThreeSeconds.bs2" Works	17
Activity 4: Boe-Bot Navigation	18
4.1 Troubleshooting Hardware	18
4.2 Basic Boe-Bot Maneuvers	20
4.3 Tuning the Basic Maneuver	22
4.4 Simplify Navigation with Subroutines	25
Activity 5: Tactile Navigation with Whiskers	29
5.1 Building and Testing the Whiskers	29
5.2 How the Whisker Circuit Works	32
5.3 Navigation with Whiskers	33
Activity 6: Light Sensitive Navigation with Phototransistors	36
6.1 Building and Testing the Phototransistor Circuits	37
6.2 How the Phototransistor circuit works	38
6.3 Roam and Avoid Shadows Like Objects	39
6.4 Getting More Information from Your Phototransistors	41
6.5 How the Code Works	44
6.6 Following the Light	45
Activity 7: Navigating with Infrared Headlights	48
7.1 Building and Testing the IR Pairs	49
7.2 How The Code Works	51
7.3 Infrared Detection Range Adjustments	52
7.4 Object Detection and Avoidance	52
7.5 The Drop-Off Detector	54

Activity 8: Robot Control with Distance Detection	56
8.1 Testing the Frequency Sweep	57
8.2 How “TestFrequencySweep.bs2” Works	60
8.3 Boe-Bot Shadow Vehicle	60
8.4 Following a Stripe	64
Activity 9: Navigating an Obstacle Course	66

Introduction

Robotics is the science of perceiving and manipulating the world around us through the use of electronic and computer-controlled devices [2]. Modern technology is filled with robotic systems that allow us to increase the efficiency, accuracy, and speed at which we are able to perform a variety of useful tasks. Whether it is a car that can parallel park itself, a robotic welding arm, or a controller for a video game, it is apparent that robotics is a field that is abundant in the world around us and will only continue to grow. Therefore, an understanding of the underlying principles of robotics is essential to success in many technology driven fields.

A robotic system consists of two types of components: hardware and software. *Hardware* refers to the physical (tangible) parts of the robot, such as the circuit boards, wires, electrical components, motors, and various types of sensors. *Software* refers to the set of tools that allow us to analyze data, send commands to the hardware, and to define rules that govern how the hardware operates.

How do we use these tools to make the robot perform our desired tasks? This question defines an entire field of study called *control engineering*. Generally speaking, a *control system* is any type of system in which a parameter/value is modified (*controlled*) based on some type input parameter/value. The most common way that this type of control is accomplished is by using hardware to gain information about the state of the system, and then using this information to determine how the system should react to reach the desired state. In other words, it is the job of a control engineer to determine the best way to utilize the information available in order to make the robotic system perform the desired task.

To make this very general definition more concrete, let's take a look at a very common control system, a household heating and cooling system. For a household heating system, the input parameter/value is the desired temperature. In this case, the value that we are trying *control* is the actual temperature, which can be read from a thermostat somewhere in the house; if the current temperature of the house is different than the desired temperature, then the heater/air conditioner is turned on until the desired temperature is reached [3].

The goal of this course is to provide an introduction to robotics, and specifically control systems. Although we will discuss some of the hardware aspects of robotics, we will focus mainly on the software/control theory and implementation. In other words, we will be mostly concerning ourselves with the issue of using the measurements that we get from our robot to accomplish a desired task. Some of the topics that we will cover to achieve our goal include: programming language, basic circuit theory, logic statements, breadboard use, hardware implementation and calibration, and feedback control.

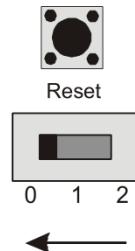
Throughout this course we will be utilizing the Board of Education Robot (Boe-Bot) kit from Parallax, Inc. and the BASIC Stamp editor, both of which are available through <http://www.parallax.com/>. Most of the activities presented in this manual are adapted from the manual "Robotics with the Boe-Bot" by Andy Lindsay [1].

Activity 1: Setting Up the Hardware

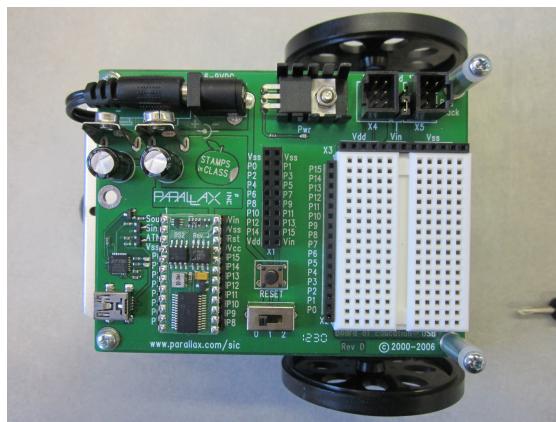
The first thing that we will need to do is test the Boe-Bot. To do this, we will first assemble the main circuit board and use the BASIC Stamp Editor to test for communication between the robot and your computer.

1.1 Testing for communication

1. Locate the main circuit board on your Boe-Bot. The main power switch on the board has 3 positions. Position 0 is for turning the power completely off. For the rest of the lab activities, you **MUST** make sure that the switch is in position 0 before making any wiring changes.



(a) Power Switch



(b) Main Circuit Board

Figure 1: Diagrams of the Main Circuit Boards

2. Load the batteries into the battery pack.
3. Plug the USB cable into the circuit board as shown in Figure 2, and plug the other end into your computer.
4. Move the 3 position switch from position 0 to position 1 to turn the power on. Note that the green light labeled "Pwr" should now be on.
5. Open the BASIC Stamp Editor software.
6. In order to make sure that the BASIC Stamp is communicating with your computer, click the Run menu, then select Identify (see Figure 3). A window similar to the one shown in Figure 4 should appear. Make sure that BASIC STAMP 2 appears in one of the COM Ports. If it does NOT appear, refer to Appendix A of [1] for troubleshooting tips.

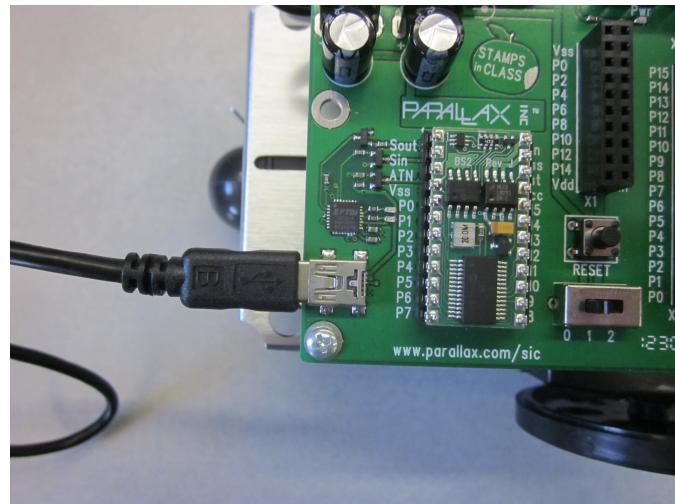


Figure 2: USB Cable

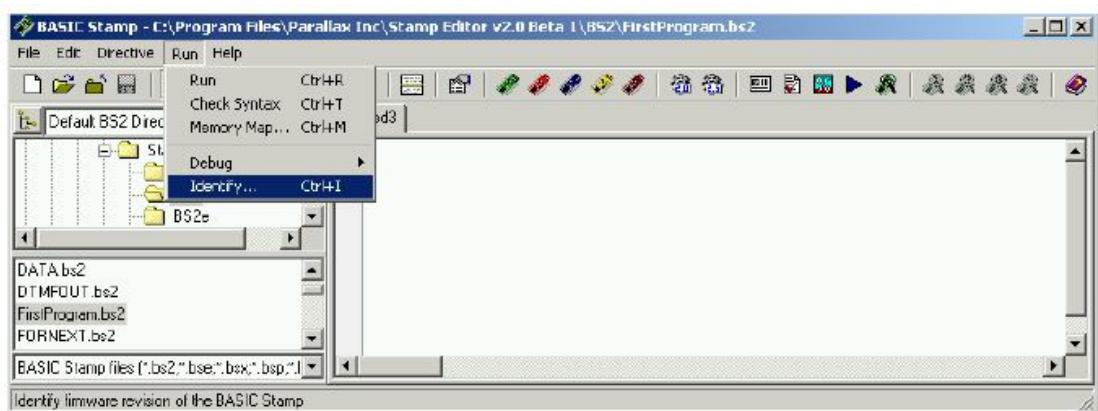


Figure 3: Basic Stamp Editor.

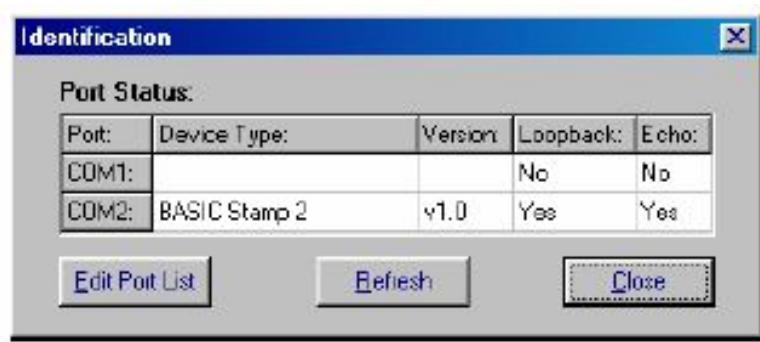


Figure 4: COM Port detection.

Activity 2: Your First Program

Now that we know the computer can successfully communicate with the hardware, we can begin to experiment with sending commands. How exactly do we use the computer to send the Boe-Bot a command to do something? The answer is through the use of a programming language. A *programming language* is an artificial language used to instruct a computer to perform a specific task. Examples of common programming languages are C++, Java, Fortran, and Python. The programming language that is used by the Boe-Bot is a specialized language called PBASIC. Although the syntax and commands in PBASIC are slightly different than the previously mentioned languages, it uses the same basic concepts and is therefore still very useful to study.

Because languages have to be written in a very precise manner, *editors* are often used to aid in writing code. An *editor* is software that aids the user in achieving code that is written correctly. The BASIC Stamp Editor, which we've already seen a bit of in the previous exercise, is the editor we will be using for all our programming.

Sending a robot commands is in many ways similar to giving a human a command. However, the challenge in programming commands is that the robot does not have the capability to make decisions on its own. Therefore, the programmer must use caution to make sure that the commands are very precise and that there are appropriate commands to cover all possible scenarios that the robot may encounter. Before we can code anything too complex, we must first start off with the basics. The following activity will give you practice with writing simple codes that will be used by the Boe-Bot.

2.1 Sending a Message to the Boe-Bot

The first program that you will write will tell the BASIC Stamp to send a message to your PC or laptop. The code for the program is displayed in the gray box below (for the rest of the activities, all programmable codes will be displayed with a gray background).

Code 1: Sample Code

```
' HelloBoeBot.bs2
' BASIC Stamp sends a text message to your PC

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

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

END
```

1. Begin the program by clicking the BS2 icon on the toolbar (the green diagonal chip, see Figure 5). If you hover the mouse above this icon, you will get the description “Stamp Mode: BS2”.
2. Next click on the gear icon labeled “2.5” (see Figure 5). The flyover description is “PBASIC Language: 2.5”. You should see the text ’{\$STAMP BS2} and ’{\$PBASIC 2.5} appear in your code similar to what is shown in Code 1.

NOTE: These two commands that appear in the code are called “compiler directives” and precede every code that we will write in this class. From this point forward, we will refer to them as the *Preamble* of the code. The purpose of the preamble is simply to let the editor know what version of PBASIC and what chip we are using. These commands are very sensitive to syntax errors, therefore, it is important that you make sure to click the icons rather than manually typing them at the beginning of your program.

3. Type the rest of the code EXACTLY how it appears in the box shown in Code 1.
4. Save your work by clicking File⇒Save. Enter the filename “HelloBoeBot.bs2” and click Save. Make sure you are saving to an appropriate directory (e.g., a personal folder or flash drive. If using a shared or network computer, saving work to the desktop or other shared drives may result in lost information).
5. Once your program is saved, make sure that your Boe-Bot is still connected to the computer and click Run⇒Run from the toolbar (Or click the blue arrow icon at the top of your screen). Once the program is downloaded to the Boe-Bot you should see a DEBUG terminal window appear on the screen displaying the message “Hello, this is a message from your Boe-Bot”. To prove that this is a message from the Boe-Bot, you can press and release the RESET button located on the Board of Education (see Figure 1). Each time that you do this, you should see a new message appear in the DEBUG Terminal.



Figure 5: BS2 icon (left) and gear icon (right)

2.2 How the Code Works

Notice that the first two lines in the code in Code 1 are preceded by a ' character. Any lines of code that are preceded by this character are called *comments*, because they will be ignored when the program is executed by the editor (with the exception of the Preamble, discussed next). The purpose of providing comments in any kind of code is to provide clarity for someone who is writing or reading the code. It is proper coding practice to always add comments to your codes. The comments should provide a description of what the code is supposed to be doing so it is easier to read.

The next two lines of the code are the *preamble*, which we already discussed above. Notice that they are also preceded by a ' character, but as we discussed in Section 2.1, they are crucial to the code. The combination of the ' and {\$...} characters tells the program that these lines of code belong to the *preamble*.

Following the preamble is the “DEBUG” command. A *command* is a word that you can use to tell the BASIC Stamp to perform a certain task. The “DEBUG” command outputs data to the computer screen when the Boe-Bot is connected to the computer. In practice, it is used to give the user a better idea of what the code is doing while a program is running. This is handy because it makes the process of fixing small errors in the code much easier. We will utilize this command often throughout this manual.

There are many additional specifications that you can add to the “DEBUG” command called *formatters*, which enable the user to control how the display looks in the debug window. In the interest of time, we will not go over these commands. If you are interested in reading about them, refer to [1].

The last command is the “END” command. This command tells the BASIC Stamp that the program has ended, and puts it into a low power mode. When in this state, the BASIC Stamp waits for either the RESET button to be pressed, or a new program to be loaded. When the RESET button is pressed, the previously loaded program will be executed. If a new program is loaded, the old program will be erased from memory and the new program will be executed.

Activity 3: Your Boe-Bot's Servo Motors

This activity will guide you through connecting, adjusting, and testing the servo motors that are used with the Boe-Bot. By the end of this activity, you will be able to control the speed, duration, and direction of the motion of the Boe-Bot servo motors. Controlling these components requires knowledge of some basic programming and control techniques, which we will explore in this section.

The Boe-Bot uses a continuous rotation servo, which turns electric energy into mechanical energy through the use of an electric motor. The servo that is used by the Boe-Bot is displayed in Figure 6 below.

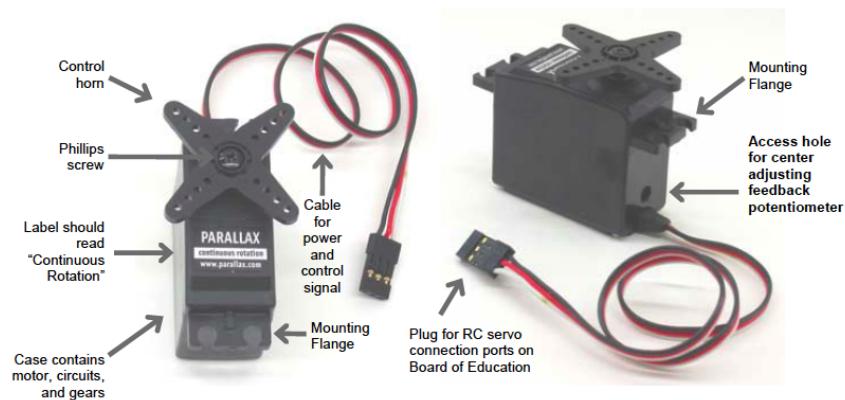


Figure 6: Parallax continuous rotation servo.

The basic idea of controlling a servo is to create a pulse of high and low voltages with a controlled duration. By changing the duration of the pulse, we can control how fast the magnets in the motor change polarity, and therefore how fast the motor turns.

This activity and almost all successive activities will require you to build a variety of circuits using a breadboard. A *breadboard* is a tool for quickly assembling, testing, and modifying electronic circuits. Your lab instructors may have gone over this in lecture, so you might already be familiar with breadboarding techniques. If you have not discussed this or it seems confusing at first, don't worry, the breadboarding concepts will become clearer as you get used to using them.

3.1 Connecting the Servo Motors

First we need to connect the servos to the circuit board. If you are following along in [1], note that we are using Board of Education Rev D.

1. First make sure that the power is off before you start any wiring (set the power switch to 0).

2. Connect the servos to the board as shown in Figure 7. There is a number written on the board right above where you connected the servos. The numbers should be 12 and 13. Take special care to ensure that you have the wires oriented correctly (white wire closest to the top), otherwise the servo will not work.

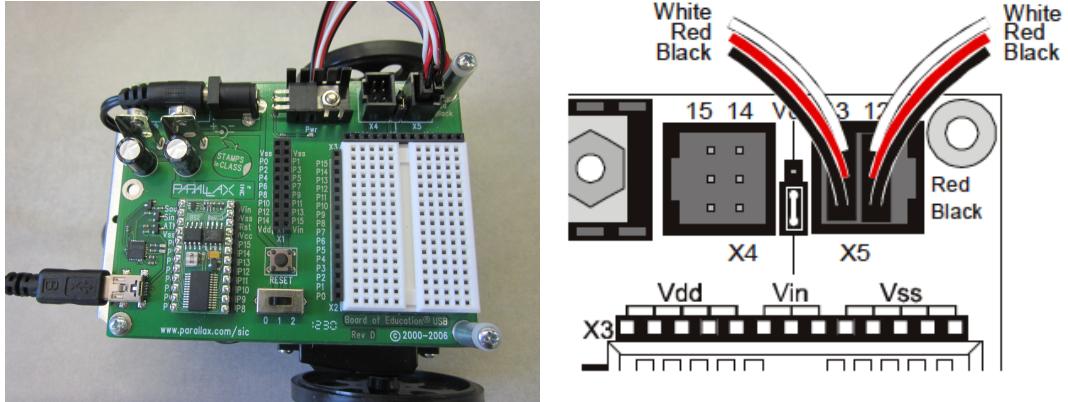


Figure 7: Board of Education with servo and battery pack connected (left), and close up of servo connection (right).

3. Build the circuit shown in Figure 8 on your breadboard.

- Notice that the LED's are connected to the I/O pins labeled P13 and P12. This means they will get power when the servos get power, so we can monitor when signals get sent to the servos.
- The two circuit components with the colored bands are called *resistors*. Their purpose is to “resist” the flow of electricity, and thus they dissipate energy (lower voltage) when current flows through them. Current can flow in either direction through resistors. The colored bands on the resistors indicate how much energy they dissipate (measured in units called Ohms, denoted with an Ω symbol). For more information on the color-coding scheme, see [1].
- The (red) LED's are what we call *diodes*, which are essentially one-way resistors which only allow current flow in one direction. Therefore, it is very important that they are oriented correctly, or the circuit will not function. The line on the LED symbol on the schematic corresponds to the flat side of the LED.

3.2 Centering the Servos

Anytime we use a computer or a controller to send commands to a mechanism, there is always going to be a small amount of error involved. For example, if we send an identical speed command to two different motors, they will most likely go slightly different speeds (This presence of error is what makes the study of control systems a necessity!! We will

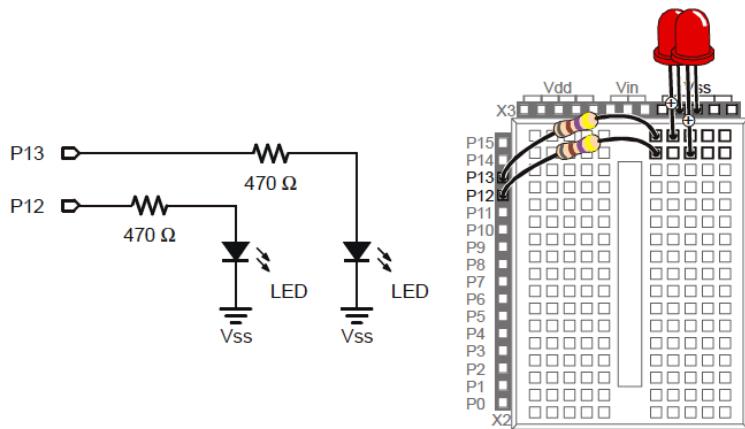


Figure 8: LED circuit to monitor servo signals.

discuss this further in later activities...). The servos on the Boe-Bot are no different than any other mechanism, so they will have a small amount of uncertainty associated with their motion. Therefore, it is necessary to calibrate the motors so they behave the way we expect them to. To do this, we will send a command to each servo instructing them to stay still, and then adjust them so that they actually do so.

The signal that we will send to the servo connected to P12 to calibrate it is shown in Figure 9. This is called the *center signal*. It consists of a series of 1.5 ms pulses with 20 ms pauses in between each pulse. In order to send this command, we will use a PULSOUT command and a PAUSE command nested inside a DO...LOOP. An example of the command written in code is shown in Code 2. For any pin location (the first number next to PULSOUT), the PULSOUT command allows us to specify the duration of the “high” voltage part of the signal in $2\mu s$ increments (the increments are specified by the second number next to PULSOUT). The PAUSE command allows us to specify the length of the “low” voltage part of signal. The purpose of a DO...LOOP is to tell the BASIC Stamp to repeatedly execute the command inside the loop.

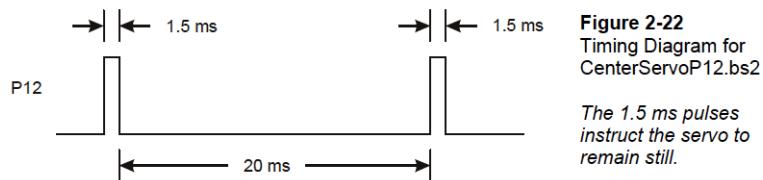


Figure 9: Timing diagram for centering our servo.

Code 2: Centering a Servo

```
'CenterServoP12.bs2
'P12 for manual centering.

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

DEBUG "Program Running!"

DO
    PULSOUT 13, 750
    PAUSE 20
LOOP
```

In order to figure out the necessary value of the PULSOUT's *duration* argument, simply divide the desired duration by $2 \mu\text{s}$ (this is the syntax required by the programming language). For our purposes if we want the pulse *duration* to be 1.5 ms, so we have:

$$\text{Duration Argument} = \frac{0.0015\text{s}}{0.000002\text{s}} = 750.$$

Therefore, a command for a 1.5 ms pulse to the servo connected to port P12 will be PULSOUT 12, 750.

We will center one servo at a time, starting with the servo connected to P12, and then repeating the process for the servo connected to P13.

1. Enter and save the program shown in Code 2. Save the program as "CenterServoP12.bs2."
2. Make sure that the power on the Board of Education is set to position 2 (this is the setting that enables functionality of the servos).
3. Run the program that you wrote in step 1. If the servo has not centered, it should begin to turn slightly.
 - If the servo doesn't turn, it may already be centered. To check if it is working set the *duration* value to 850 instead of 750. If the servo starts rotating quickly then it is working. Do not attempt to *center* the servo when the duration is set to 850, you may damage it. Instead skip step 4 and proceed to step 5.
4. The rate at which the servo turns is controlled by a resistor called a *potentiometer*. Gently use a screwdriver to adjust the potentiometer in the servo as shown in Figure 10. USE CAUTION WHEN YOU DO THIS, THE POTENTIOMETER IS FRAGILE. Adjust the potentiometer until you find a setting that makes the servo stop turning.

NOTE: Although unlikely, a damaged or defective servo could also cause the servo to not turn. If you suspect that your servo is defective, tell your lab instructor.

5. Modify the program to turn the servo attached to P13, and repeat the process.

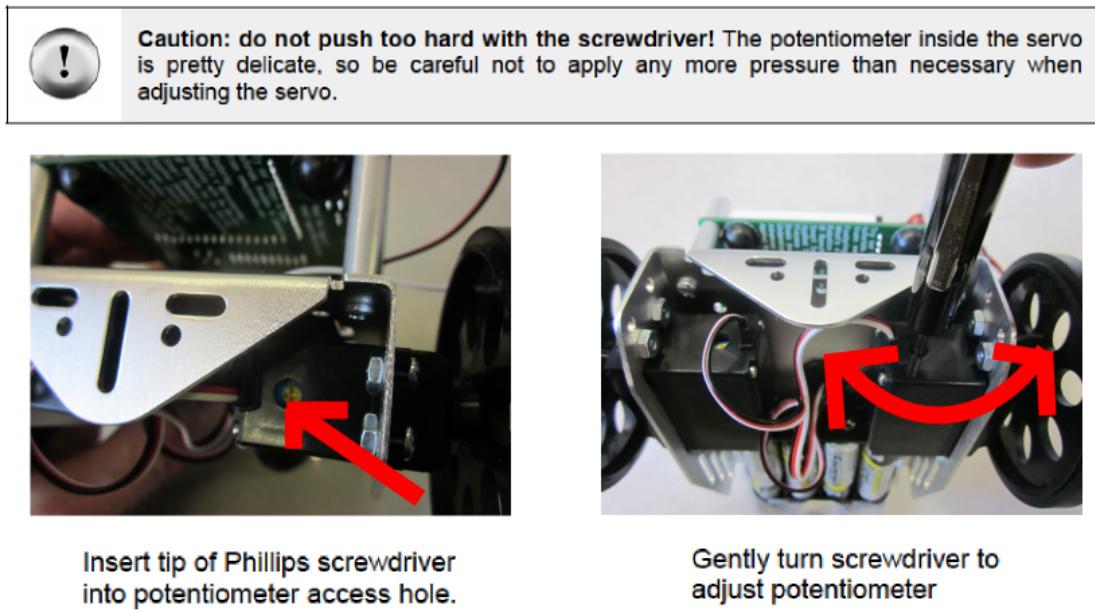


Figure 10: How to center a servo.

3.3 Testing the Servos

The last thing that we need to do before proceeding is to test the servos to make sure that they are operating correctly. The practice of testing the functionality of individual components of a system before attaching them to the overall system is called *subsystem testing*, and is good practice in any mechanical project.

Recall from the last activity that sending a signal with a pulse-width of 1.5 ms to the servos will now cause them to stay still. We did this by using the PULSOUT command with a *duration* argument of 750 and then calibrating the servos. With the particular servo that we are using, shortening the pulse width causes the servo to turn faster. For example, any PULSOUT command that contains a *duration* argument that is less than 750 will cause the servo to turn in a clockwise direction. The shorter the pulse width the faster it will turn until the maximum possible speed is reached. A pulse that has a pulse width of 1.3 ms (a *duration* of 650), will cause rotation at maximum speed in a clockwise direction. Maximum speed with these servos ranges from 50 – 60 RPM (rotations per minute). Conversely, if we send the servo a pulse that has a pulse width that is greater than 1.5 ms, the servo will turn in a counter-clockwise direction. A *duration* argument

of 850 will cause the servo to turn at full speed in a counter-clockwise direction. For brevity, we won't go into detail on how maximum speed is determined, but the concepts demonstrating how maximum speed is determined are illustrated in Figure 11.

In addition to the speed and direction of the servos, we can also control the amount of time that the motor runs. We will do that with the use of the FOR...NEXT loop. You will learn more about FOR...NEXT loops later so we will not go into a great amount of detail here, but the basic idea is that we will write a sequence of commands, and tell the BASIC STAMP to execute them a pre-set number of times. If we know how long it will take to execute the series of commands once, and we know how long we want the motor to run, we can then predict how many iterations to perform in order to achieve our desired run time. To illustrate this, consider the following example

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

Each time through the loop, the PULSOUT command takes 1.7 ms to execute, the PAUSE command takes 20 ms to execute, and it takes about 1.3 ms for the editor to execute the commands themselves. Therefore, each time through the loop takes about

$$1.7 + 20 + 1.3 = 23 \text{ ms.}$$

Since the loop executes 100 times, we can figure out the total length of time as:

$$100 * 0.023s = 2.3 \text{ seconds.}$$

Let's solidify these concepts.

1. Enter, save, and run the program in Code 3 in order to make both of the servos turn forward for 3 seconds, and then backward for 3 seconds (The *duration* command of each servo is different. Why?). Save the program as "BothServosThreeSeconds.bs2". CAUTION- Make sure that you hold the Boe-Bot off of the table when you initially press the "Run" button. Once the code has been loaded onto the chip, you can disconnect the USB cable, set the robot on the ground (or the table, but make sure it doesn't fall), and hit the "Reset Button" to execute the code.
2. Now pick a time (preferably a relatively small time, so that it doesn't take too long) and try to modify the above code to make the servos turn for your desired amount of time and direction. Was your prediction close to what actually happened?

Code 3: Forward and Backward 3 Seconds

```
'BothServosThreeSeconds.bs2
'Run both servos in for three seconds,
'then reverse the direction and run another three seconds.

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

DEBUG "Program Running!"

counter VAR BYTE

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

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

END
```

3.4 Additional Testing

Now that we know the servos are working, we will quickly generate some code to give you a slightly better feel for the motion commands, and to ensure one more time that everything is attached correctly. If the program runs as expected, we will move on to programming motion commands.

Even though we centered the servos in a previous activity, there is still a certain amount of error involved with sending commands to the servos. This is simply due to natural variations in the manufacturing process of the hardware. To illustrate, if we send a PULSOUT 13, 650 the left wheel will begin to rotate at a certain speed. If we send a PULSOUT 12, 650 command, the right wheel will begin to rotate, but it will most likely be at a slightly different speed than the left wheel. Think about these things as you work through the activity, and begin to think about ways to fix this problem.

1. Open up the file “BothServosThreeSeconds.bs2”.

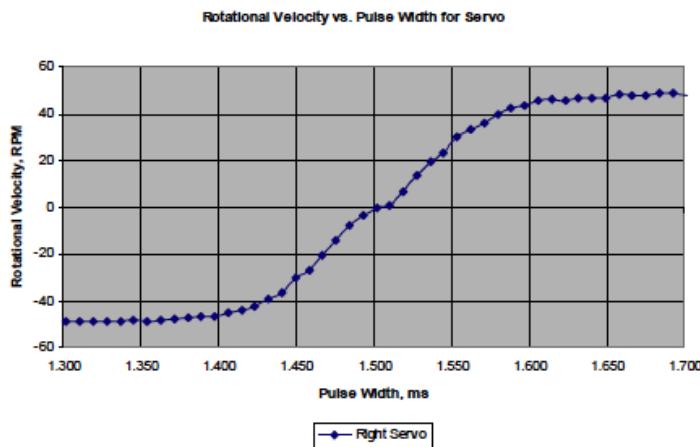


Figure 11: Pulse width versus rotational velocity

2. Connect the Boe-Bot to the computer with the appropriate USB cable.
3. Hold the Boe-Bot up off of the ground, but make sure that the wheels can rotate.
4. Run the code. Make sure that the wheels rotate toward the front of the Boe-Bot for 3 seconds, stop, and then rotate the opposite direction for 3 seconds. The wheels should first rotate in a manner as to move the Boe-Bot forward (The side closest to the breadboard is the front. See Figure 12), and then in reverse. If everything works as expected, move on to the next step.
If your Boe-Bot goes in reverse first, then you have your servos connected backwards. Take the servo wires and switch the ports that they are plugged into. Remember to keep the white wire on top. Run the code again and make sure the problem is fixed. If you need clarification with this, alert your lab instructor.
5. Enter the code shown below in Code 4 and save it as “TestServoSpeed.bs2”. An explanation of how this code works is provided in the following subsection.

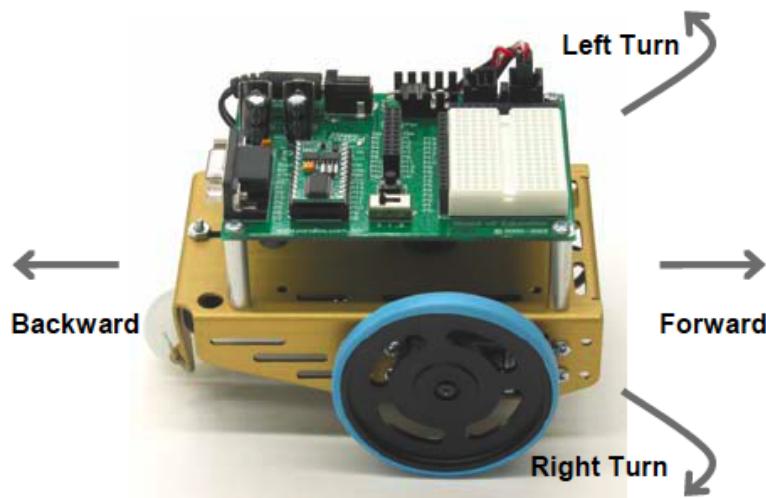


Figure 12: Pulse width versus rotational velocity

Code 4: Code for Calibrating Servos

```
' TestServoSpeed.bs2
'
' {$STAMP BS2}
' {$PBASIC 2.5}

counter      VAR      WORD
pulseWidth   VAR      WORD
pulseWidthComp VAR      WORD

DEBUG "Program Running!"

DO
    DEBUG "Enter pulse width: "
    DEBUGIN DEC pulseWidth
    pulseWidthComp = 1500 - pulseWidth
    FOR counter = 1 TO 224
```

```

PULSOUT 12, pulseWidth
PULSOUT 13, pulseWidthComp
PAUSE 20
NEXT

LOOP

```

6. Run the code. The Debug Terminal's transmit windowpane should appear on the screen. Type in the value 650 into the window and press ENTER.
7. Verify that the servo turns at full speed forward for 6 seconds and then stops.
8. Once the servo has stopped turning, you will be prompted for another value. Type in 850 and hit ENTER. Verify that the servo turns for 6 seconds reverse.

3.5 How “BothServosThreeSeconds.bs2” Works

At the beginning of the code, three variables are declared. *counter* will be used in the FOR...NEXT loop, *pulseWidth* will be used in the DEBUGIN and PULSOUT commands, and *pulseWidthComp* will be used in a second PULSOUT command.

Once the variables have been appropriately declared, we declare a DO...LOOP, so that the code will execute over and over again until the user tells it to terminate. The rest of the code will be nested within this loop. The Debug terminal operator (you) will be prompted to enter a pulse width. The DEBUGIN command is used to store whatever value that you type in to the Transmit windowpane into the *pulseWidth* variable.

The next command after the DEBUGIN command, takes the value of *pulseWidth* that you entered, subtracts it from 1500, and then stores the result as *pulseWidthComp*. These variables are then used in the FOR...NEXT loop that will run for 6 seconds before moving on to the next command.

The final part is a bit tricky. We use two PULSOUT commands, one to each wheel, making one PULSOUT command the same amount above 750 as the other is below 750. As you can probably guess, the sum of the two PULSOUT *duration* arguments is always 1500. By doing this, we ensure that two PULSOUT commands combined will always take the same amount of time (6 seconds). This will make the RPM measurements that you take more accurate. Conveniently, this procedure also is necessary in order to make both of the wheels turn in the same direction at the same speed (Why? Do they turn at exactly the same speed?).

Activity 4: Boe-Bot Navigation

The Boe-Bot can be programmed to perform a variety of maneuvers. Basic tasks such as moving forward, backward, and turning left or right can be written into codes using the same programming techniques that we have discussed thus far. These basic maneuvers are the building blocks for more advanced, environmentally aware codes that we will write in later activities.

In this activity, we will illustrate how to write codes that efficiently execute routines involving the basic Boe-Bot maneuvers. Using the information from the previous activity, as well as a bit of trial and error, we should be able to fine-tune these maneuvers a bit so that they perform at a satisfactory level. In this activity, we are not using any type of *feedback* in our attempt to control the Boe-Bot maneuvers. *Feedback* is when we utilize measurements of some output quantity (such as speed or direction) to improve performance of the system in real-time. Any type of control law that does not use feedback is referred to as *open loop* control.

Before we begin, however, we will need to introduce a technique for troubleshooting that will be valuable throughout the rest of the manual.

4.1 Troubleshooting Hardware

Learning how to troubleshoot hardware is essential for designing any system. When software and hardware are working together, this is especially true. For example, how do you know what's not working? Is it the hardware that's broken, or was the code written incorrectly? It's not hard to imagine how important troubleshooting becomes when your hardware has many parts which is run by thousands of lines of code. Did you know that we already did hardware troubleshooting in our previous activity? When testing the servos, we built the hardware so the LED would blink when the servo was supposed to be turning. If the LED didn't blink, then we knew that no movement command was being sent to the servo. This visual cue indicated when the hardware was communicating with the computer and when it was not.

Even though the LED circuit for detecting servo movement was useful, as we move forward in the activities, we really don't want to add a new LED every time we add or change hardware. What we will do instead, is just add one LED that we can set to turn on at the appropriate time to aid in troubleshooting. It turns out that this will be just as, if not more, useful as adding a new LED for each piece of hardware.

Follow these steps to build your troubleshooting circuit.

1. First make sure that the power is off before you start any wiring (set the power switch to 0).
2. Build the circuit shown in Figure 13 on your breadboard.
3. Enter and save the program shown in Code 5. Save the program as "TroubleShoot.bs2."
4. Make sure that the power on the Board of Education is set to position 1 or 2.

- Run the program that you wrote in step 3.

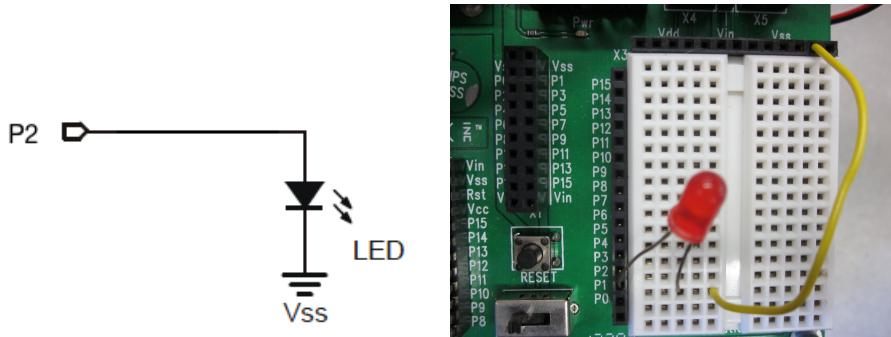


Figure 13: LED Circuit to Troubleshoot.

Code 5: Troubleshooting

```

' TroubleShoot.bs2

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

DEBUG "Program Running!"

For counter =1 to 100           ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

```

As the code runs, you should see the LED blinking and then stop. Since we've already seen code like this before in the last activity, you should have an idea of how it works. Also, from this point forward you will always see this chunk of code at the beginning of all the example codes in the manual. This way, as long as the LED circuit is in tact, when you run any future code you should see the light blink before anything else happens. This will be particularly useful for when your Boe-Bot is not connected to the computer. Now you can tell if the code is actually starting from the beginning when you hit the reset button.

For troubleshooting purposes, write only the portion of the code that makes the light blink (i.e., the FOR...NEXT loop) before and after the command you are trying to troubleshoot. In this way, you can see if the robot ever executes the portion of the code

you think it should be executing. If you run into a problem and are not sure how to troubleshoot, ask your lab instructor for assistance.

4.2 Basic Boe-Bot Maneuvers

Now that we are familiar with how to troubleshoot, we can move forward using what we already know about controlling the Boe-Bot servos from the previous sections to start executing maneuvers. Sending commands to the Boe-Bot to perform specific movement routines is simply a matter of putting the commands we know together in the right sequence. As a reminder, Figure 14 shows the orientation that we will use for the Boe-Bot's front, back, left, and right sides. Moving "forward" refers to the Boe-Bot moving toward the right-hand side of the page, and moving "backward" refers to movement toward the left-hand side of the page. We will use this convention for the remainder of the lab manual.

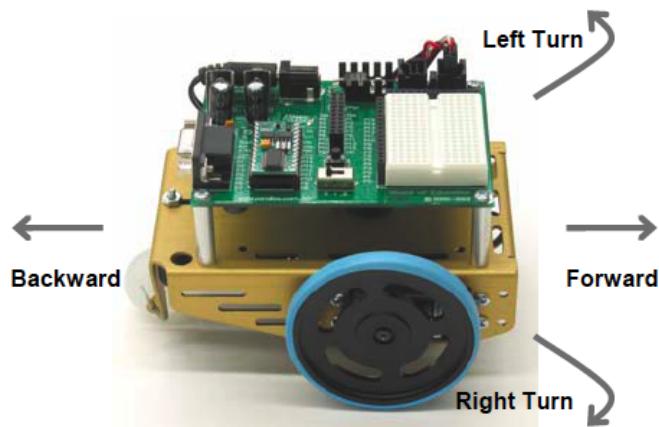


Figure 14: Orientation of Boe-Bot.

The first code that we will write will make the Boe-Bot move forward, then left, then right, and then backward. Keep in mind that in order to make the Boe-Bot move forward or backward, we actually need to command the wheels to move in *opposite* directions (i.e., clockwise and counter-clockwise) because of their opposite orientations. Conversely, to make the Boe-Bot turn right or left, we actually need to command the wheels to move in the *same* direction. This can be a bit confusing, but it should become clear as we proceed through the activity.

1. Make sure that the Boe-Bot power switch is set to position 2, and that the Boe-Bot is connected to the computer via the USB cable.
2. Open the BASIC Stamp Editor. Enter, save, and run the following code in Code 6. Save the code as "ForwardLeftRightBackward.bs2". Note that once the code

runs once and is loaded onto the BASIC Stamp, it no longer needs to be attached to the computer to execute. You can try this by removing the USB cable, setting the Boe-Bot on the ground, and pressing the “reset” button.

Code 6: Boe-Bot Movement

```
' ForwardLeftRightBackward.bs2

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

DEBUG "Program Running!"

counter      VAR      WORD

For counter =1 to 100          ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

FOR counter = 1 TO 64          ' Forward

    PULSOUT 13, 850
    PULSOUT 12, 650
    PAUSE 20

NEXT

PAUSE 200

FOR counter = 1 TO 24          ' Rotate Left

    PULSOUT 13, 650
    PULSOUT 12, 650
    PAUSE 20

NEXT

PAUSE 200

FOR counter = 1 TO 24          ' Rotate Right
```

```

PULSOUT 13, 850
PULSOUT 12, 850
PAUSE 20

NEXT

PAUSE 200

FOR counter = 1 TO 64           ' Backward

PULSOUT 13, 650
PULSOUT 12, 850
PAUSE 20code:act4_3

NEXT

END

```

The code you just wrote doesn't use any new commands, it's all based things that we've previously seen or discussed. See if you can figure out what is happening in the code. Look through the old activities if you need help remembering some of the commands. If it still does not make sense to you, alert one of your lab instructor and ask them to explain it. Understanding these basic codes is crucial for more complicated maneuvers.

4.3 Tuning the Basic Maneuver

The routine implemented in Code 6 is an example of *open loop* control. The problem with open loop control is that the system cannot detect when real world errors begin to accumulate. Therefore, any errors that occur stay in the system. For example, when we tell the Boe-Bot to go forward or backward, there is no guarantee that it will actually do so (well, not precisely anyway). It may veer slightly to the left or slightly to the right. The simplest way to fix this is with *feedback* control, however, in the interest of time and simplicity we will stick to open loop control for now in controlling the basic robot maneuvers. We will use feedback control once we start writing more sophisticated codes. For now, we will do our best to tune the Boe-Bot maneuvers for the best performance. This section will guide you through doing so.

The first thing that we need to do is to figure out if the Boe-Bot moves in a straight line when it is supposed to, or if it curves in one direction or another. We will do this by writing a simple code that tells the Boe-Bot to move forward for ten seconds, and then fine-tuning the commands.

1. Enter, save, and run the code in Code 7. Save it as “ForwardTenSeconds.bs2”.

Once the code is loaded onto the BASIC Stamp, disconnect the Boe-Bot from the computer, set the robot on the ground, and press “reset” to run the code again.

Code 7: Forward Movement

```
'ForwardTenSeconds.bs2
'Make the Boe-Bot roll forward for ten seconds.

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

DEBUG "Program Running!"

counter      VAR      WORD

For counter = 1 to 100          ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

FOR counter = 1 TO 407          ' Number of pulses - run time.
    PULSOUT 13, 850
    PULSOUT 12, 650
    PAUSE 20
NEXT

END
```

2. Observe whether the Boe-Bot moves in a straight line, or if it veers to the side.
3. Modify the code to correct this. If your Boe-Bot veers left, you need to slow the right wheel down. If your Boe-Bot veers right, you need to slow the left wheel down a bit. Think carefully about how to modify the *duration* arguments in order to accomplish this. It’s a bit tricky!
4. Write down the pulse width *duration* values that make the Boe-Bot move as straight as possible. From now on, when you see sample code in the manual that instructs the Boe-Bot to move forward, use the *duration* arguments you just found, rather than the ones that are printed in the example code.
5. Modify the code so the Boe-Bot moves backwards for 10 seconds. Save it as “BackwardTenSeconds.bs2”. Repeat steps 2 through 4 to tune your robot so it moves backward in a straight line.

Now we will follow a similar procedure in order to tune the turns.

1. Enter, save, and run the code in Code 8. Save it as “TuningTurns.bs2”. Once the code is loaded onto the BASIC Stamp, disconnect the Boe-Bot from the computer, set the robot on the ground, and press “reset” to run the code again.

Code 8: Boe-Bot Rotate 90 Degrees

```
' TuningTurns.bs2
' Move servos through cw/ccw rotation combinations.

' {$STAMP BS2}code:act4_5
' {$PBASIC 2.5}

DEBUG "Program Running!"

counter      VAR      WORD

For counter = 1 to 100          ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

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

2. The code should make the Boe-Bot turn 90 degrees to the right. If the robot overshoots the angle (turns more than 90 degrees), modify the counter on the FOR...NEXT loop to make the duration of the loop shorter and try again. Conversely, if the robot undershoots the angle, modify the counter on the loop to make its duration longer. Note that in this step we are NOT modifying the pulse width *duration* argument like we did in the previous step.
3. Once you have found an *endvalue* argument that makes robot perform to satisfactory accuracy, write down the value. From now on, when you see sample code in the manual that instructs the Boe-Bot to perform a 90 degree turn, use the *endvalue* argument on the FOR...NEXT loop that you just found.

4. Modify the code so that the Boe-Bot turns to the left. Repeat steps 2 and 3 with the goal of tuning your Boe-Bot so it now turns 90 degrees to the left.

Make sure to keep a note of all the values you just tuned. You will want to use these values instead of the default ones printed in all subsequent codes. Also, note that these values can change with time due to various factors, so do not be alarmed if your Boe-Bot starts veering suddenly one day when it wasn't before; this simply means a quick re-tuning is in order.

4.4 Simplify Navigation with Subroutines

Writing code is somewhat of an art form, and it takes a lot of practice to become proficient at it no matter what programming language you use. There are usually multiple ways that we can write a program to perform the same task, so the question becomes: How do we write a code that will execute the desired task, and do so in the fastest, most memory efficient way? Computer science is an entire field that is devoted to answering this question, so we will obviously not be able to do it in only a few short classes. However, there are some very common tools that we will introduce in order to simplify the code writing process. One such tool is the use of *subroutines*. Subroutines become useful when there are sections of code that are needed multiple times. For example, we will often be calling on basic maneuvers (move straight, turn left,...etc.) repeatedly, so it would be slightly annoying to re-write the same code over and over again every time we need to perform a maneuver. This is where subroutines are useful.

A *subroutine* is essentially a code within a code. It allows us to embed a chunk of code anywhere in our program without writing out all of it. It starts as a label that serves as the name of the subroutine, and ends with a RETURN command. Within a subroutine, we write a series of commands, and then anytime in the main code that we want to execute those commands we simply type the name of the subroutine (this is referred to as “calling” a subroutine) rather than having to copy the same sequence of commands. Subroutines can also take input arguments, so that even if the commands that we want to execute are not EXACTLY repeated, we can often still accommodate them in a subroutine. We will illustrate some of these concepts here.

1. Open the code “ForwardLeftRightBackwad.bs2” and modify it so that it looks like the code shown below in Code 9.

Code 9: Navigation with Subroutines

```
' MovementsWithSubroutines.bs2

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

DEBUG "Program Running!"
```

```

counter      VAR      WORD

For counter =1 to 100          ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

GOSUB Forward
GOSUB Left
GOSUB Right
GOSUB Backward

END

Forward:
FOR counter = 1 TO 64
    PULSOUT 13, 850
    PULSOUT 12, 650
    PAUSE 20
NEXT
PAUSE 200
RETURN

Left:
FOR counter = 1 TO 24
    PULSOUT 13, 650
    PULSOUT 12, 650
    PAUSE 20
NEXT
PAUSE 200
RETURN

Right:
FOR counter = 1 TO 24
    PULSOUT 13, 850
    PULSOUT 12, 850
    PAUSE 20
NEXT
PAUSE 200
RETURN

Backward:
FOR counter = 1 TO 64

```

```

PULSOUT 13, 650
PULSOUT 12, 850
PAUSE 20
NEXT
RETURN

```

2. Run the code. Did it perform the same maneuvers as the original code?

The GOSUB command tells the BASIC Stamp to execute the appropriate subroutine. Using subroutines, the main portion of the code (everything minus the subroutines) is only about 8 lines long. The above code contains four subroutines: Forward, Left, Right, and Backward. Each one contains the commands necessary to perform the corresponding maneuver. However, you'll notice that the four subroutines are VERY similar... so it seems like we should be able to simplify the code even further. It turns out that we can.

1. Modify the code that you just wrote so that it looks like Code 10.

Code 10: Movement with 1 Subroutine

```

' MovementWithOneSubroutine.bs2
' Make a navigation routine that accepts parameters.

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

DEBUG "Program Running!"

counter      VAR      WORD
pulseLeft     VAR      WORD
pulseRight    VAR      WORD
pulseCount    VAR      BYTE

For counter =1 to 100          ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

'Forward

pulseLeft= 850: pulseRight = 650: pulseCount = 64: GOSUB Navigate

```

```

' Left Turn

pulseLeft= 650: pulseRight = 650: pulseCount = 24: GOSUB Navigate

' Right Turn

pulseLeft = 850: pulseRight = 850: pulseCount = 24: GOSUB Navigate

' Backward

pulseLeft = 650: pulseRight = 850: pulseCount = 64: GOSUB Navigate

END

Navigate:
FOR counter = 1 TO pulseCount
    PULSOUT 13, pulseLeft
    PULSOUT 12, pulseRight
    PAUSE 20
NEXT
PAUSE 200
RETURN

```

2. Save and run the code. Did it execute like you expected?

This code does the same maneuvers as the code in Code 10 and only contains one subroutine. For each maneuver the main code contains a GOSUB command that calls the subroutine. It executes the correct maneuver by assigning 3 new variables: *pulseLeft*, *pulseRight*, and *pulseCount*. We no longer have a set *endvalue* and *duration* argument to the FOR...NEXT loop and the PULSOUT commands, respectively. Instead we replace the set values with a variable and change the value of the variable before calling the routine. Although it may not seem like it here, the practice of writing subroutines can drastically save us time and space. We can actually further save memory by using even more advanced techniques, however, we will omit them here in the interest of time. If you are interested in reading about additional ways to save memory, refer to [1].

Activity 5: Tactile Navigation with Whiskers

Programming a robot to execute predetermined routines is useful in some scenarios. However, applications of such routines are limited, and we can drastically increase a robot's capabilities by introducing environmental awareness. This section will give you the first taste of creating programs that utilize information from the robot's surroundings to determine how it will react. This is the essence of a control system, and we'll be focusing on these types of programs for the remainder of the class.

Generally, the task of environmental awareness is accomplished by using a sensor that can detect changes in the environment. One type of robotic sensor is a called a *tactile switch* which, as you may have guessed, determines information about its surroundings by physical touch. In this activity, we will utilize "whiskers" (a type of tactile switch), which are pieces of wire that protrude out from the Boe-Bot, in order to detect objects in the environment.

From this point forward, you will be doing fairly extensive wiring using a breadboard, which will require you to follow basic breadboarding guidelines. Remember that having an organized breadboard can make troubleshooting much simpler!

5.1 Building and Testing the Whiskers

The first thing that we need to do is assemble the whiskers and test their functionality. This section will guide you through this process. You will be using the parts shown in Figure 15.

Parts List:

- (2) Whisker wires
- (2) $\frac{7}{8}$ " pan head 4-40 Phillips screws
- (2) $\frac{1}{2}$ " round spacer
- (2) Nylon washers – size #4
- (2) 3-pin m/m headers
- (2) Resistors, 220 Ω (red-red-brown)
- (2) Resistors, 10 k Ω (brown-black-orange)



Figure 15: Parts used for whiskers activity.

1. Make sure that the power switch is set to position 0 before making any changes.
2. Remove the front two screws holding the Board of Education to the front standoffs.
 - Use Figure 16 along with steps 3-5 to assemble your whiskers.
3. Thread a nylon washer and then a $\frac{1}{8}$ " round spacer onto each of the $\frac{7}{8}$ " screws. Attach the screws through the holes in your board and into the standoffs below, but do not yet tighten them all the way.

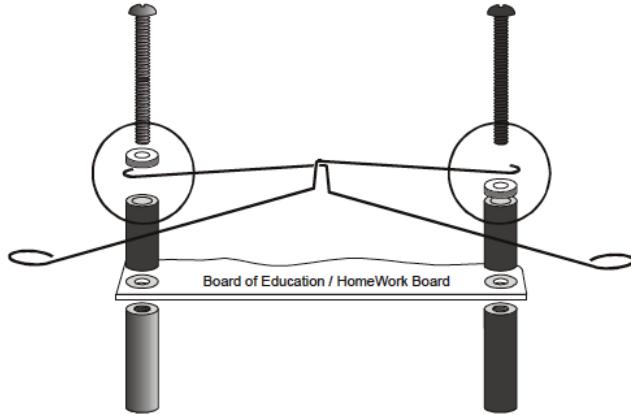


Figure 16: How to assemble the whiskers on the Boe-Bot.

4. Slip the hooked ends of the whisker wires around the screws, one above the washer and the other below the washer, positioning them so that they cross over each other without touching.
5. Tighten the screws into the standoffs.
6. Add the whiskers circuit shown in Figure 17 to the breadboard. You want to make sure that the whisker is close, but not touching the three pin header on the breadboard. A distance of about $\frac{1}{8}$ " should suffice.
7. Once the whiskers are attached, set the power switch on the board to position 2. Enter, save, and run the code in Code 11. Save it as "TestWhiskers.bs2".
8. While the code is running, press the right whisker into the 3 pin header, it should read P5=1 P7=0 in the Debug window. Press the left whisker into the header, it should read P5=0 P7=1. Simultaneously pressing the whiskers should cause the Debug terminal to read P5=1 P7=1. If you get these results, the circuit is functioning properly and you can move onto the next step. Since this code is not crucial to the rest of the lab, we will omit a formal discussion of its functionality, but feel free to ask one of your lab instructors if you're curious!

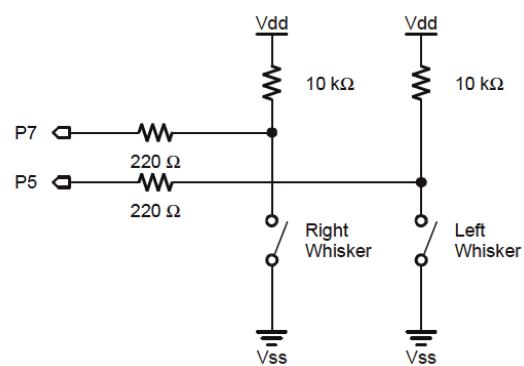
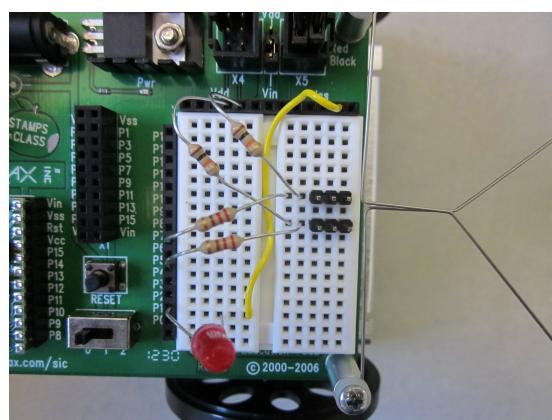


Figure 17: Whiskers wired to bread board (left) and wiring schematic (right).

Code 11: Test Whiskers

```
' TestWhiskers.bs2
' Display what the I/O pins connected to the whiskers sense.

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

counter VAR BYTE

For counter =1 to 100           ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

DEBUG "WHISKER STATES", CR,
      "Left      Right", CR,
      "-----  -----"

DO
DEBUG CRSRXY, 0, 3,
      "P5 = ", BIN1 IN5,
      "P7 = ", BIN1 IN7
PAUSE 50
LOOP
```

5.2 How the Whisker Circuit Works

Refer to the whiskers schematic that is shown in right hand side of Figure 17. Ohm's Law, $V = IR$, tells us that the voltage at a given node of a circuit is directly proportional to the amount of resistance present and the amount of current flowing through the circuit. Each whisker is a mechanical extension of the ground electrical connection of a normally open, single pole, single throw switch. The reason that the whiskers are connected to the ground are because the plated holes at the outer edge of the board are all connected to V_{ss} . Therefore, you can think of the whiskers like a switch that closes when it comes into contact with the 3-pin header.

The way that the circuit functions is that V_{dd} will supply a constant 5 V potential, and the voltage at the I/O pins P7 and P5 will be monitored (the $220\ \Omega$ resistors are to protect hardware in the presence of programming errors. This is very important from a practical standpoint, however, for a properly functioning code, they will not affect the circuit. Therefore, when analyzing this and future similar circuits, you can simply ignore them). When the whisker is not pressed, we have an open circuit so no current flows

and the voltage at the corresponding I/O pin will be 5 V. When the whisker is pressed, the node marked with a black dot in Figure 17 becomes grounded and the voltage at the I/O pin will be very close to 0. The I/O pin will then store a 1 if the input signal is 5 Volts, and it will store a 0 if the input signal is 0 Volts. We can then use this information to determine how the robot should react when the whisker is pressed.

5.3 Navigation with Whiskers

Now with the hardware in place and the theoretical background under control, we can start the fun stuff. In this section, we will write a code to detect when a whisker is pressed, and then take advantage of this information to guide the Boe-Bot. When the whisker is pressed, it means that the Boe-Bot has encountered something, so we need to instruct it to turn around and attempt to move in a different direction.

We already know how to program the Boe-Bot for forward motion, but in order to make a decision on how to react to obstacles, we need to introduce a new programming concept, *conditional statements*. A *conditional statement* is a command that only executes if a given condition is met. PBASIC has a statement that is called an IF...THEN statement that has exactly this functionality. The formal syntax for these statements is:

IF(condition)

...

THEN

...

{ELSEIF(condition)}

...

{ELSE}

...

ENDIF

The parts of the syntax above listed in {} are optional arguments. The ELSEIF part of the syntax allows the user to specify statements that have multiple scenarios (conditions) with different reaction commands. The ELSE part of the syntax tells the system what to do if none of the specified conditions are met. If a set of commands depends on multiple conditions, we can link two conditional statements through the use of logical operators (AND, OR). As expected, an AND statement only executes when both conditional statements are met, and an OR statement executes when either or both of the conditional statements is met. We will need to use an AND statement in our code to tell the robot how to behave when both whiskers are pressed.

1. Reconnect power to your board and servos.
2. Enter, save, and run the code in Code 12. When you first load the code with the Boe-Bot still attached to the computer, lift it off of the ground to avoid damaging equipment. Save the code as “RoamingWithWhiskers.bs2”. Remember that when entering this code, you may need to modify the PULSOUT *duration* arguments

and the FOR...NEXT *endvalue* arguments to be consistent with the movement calibration exercises that we did earlier in Section 4.

3. Read over the code to make sure that you fully understand the purpose of all of the commands. At this point, you should be able to follow everything that is used. If something is unclear, alert one of your lab instructors to clarify.
4. Once this code is loaded onto the BASIC Stamp, disconnect the USB cord, set the Boe-Bot onto the floor, and press the reset button to test out the code.

Code 12: Roaming With Whiskers

```
' RoamingWithWhiskers.bs2
' Boe-Bot uses whiskers to detect objects, and navigates around them.

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

DEBUG "Program Running!"

pulseCount      VAR     BYTE
counter        VAR     BYTE           ' FOR...NEXT loop counter.

For counter =1 to 100                  ' Signal program start/reset.
  PULSOUT 2, 750
  PAUSE 20
NEXT

DO
  IF (IN5 = 0) AND (IN7 = 0) THEN      ' Both whiskers detect obstacle
    GOSUB Back_Up                     ' Back up & U-turn (left twice)
    GOSUB Turn_Left
    GOSUB Turn_Left

  ELSEIF (IN5 = 0) THEN                ' Left whisker contacts
    GOSUB Back_Up                     ' Back up & turn right
    GOSUB Turn_Right

  ELSEIF (IN7 = 0) THEN                ' Right whisker contacts
    GOSUB Back_Up
    GOSUB Turn_Left

  ELSE                                ' Both whiskers 1, no contacts
    GOSUB Forward_Pulse              ' Apply a forward pulse
    ENDIF                             ' and check again
LOOP
```

```

Forward_Pulse:           ' Send a single forward pulse.
    PULSOUT 13, 850
    PULSOUT 12, 650
    PAUSE 20
    RETURN

Turn_Left:               ' Left Turn 90 degrees
    FOR pulseCount = 0 TO 20
        PULSOUT 13, 650
        PULSOUT 12, 650
        PAUSE 20
    NEXT
    RETURN

Turn_Right:              ' Right Turn 90 degrees
    FOR pulseCount = 0 TO 20
        PULSOUT 13, 850
        PULSOUT 12, 850
        PAUSE 20
    NEXT
    RETURN

Back_Up:                 ' Back up.
    FOR pulseCount = 0 TO 40
        PULSOUT 13, 650
        PULSOUT 12, 850
        PAUSE 20
    NEXT
    RETURN

```

This will conclude our activity on “Tactile Navigation with Whiskers.” However, it may be beneficial for you to read through the activity on “artificial intelligence and deciding when you are stuck”, which can be found in [1], as it may be helpful to you in the final activity.

Activity 6: Light Sensitive Navigation with Phototransistors

In this activity, we will explore a different kind of sensor called a phototransistor. A *transistor* is an electrical component that acts like a “valve” that regulates the amount of current that flows through its two terminals. The third terminal is like a switch that determines how to control the current. Depending on the type of transistor, the current flow can be controlled by voltage, current, or, in this case, light. A *phototransistor* is a light-dependent transistor that has spectral sensitivity similar to that of the human eye. A schematic of a phototransistor is shown in Figure 18.

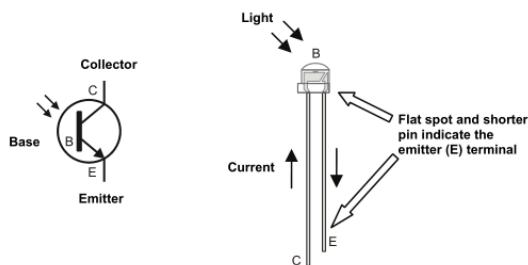


Figure 18: Phototransistor schematic and part drawing.

All three terminals of the phototransistor serve a different purpose, so it is very important when building circuits that the phototransistor is oriented correctly. The phototransistor has two different length pins and a flat spot on its plastic case for identifying its terminals. The longer of the two pins indicates the phototransistor’s *collector* terminal, and the shorter represents the *emitter* terminal. The emitter terminal also connects closer to a flat spot on the clear plastic case, which is useful for identifying the terminals. Note that your phototransistor only has two leads, even though a transistor typically has three. That’s because the third lead is internal, and the input to the lead is the amount of ambient light in the room.

To summarize, the current that is flowing through the phototransistor changes based on the amount of visible light that is present. The idea of this activity is similar to that of the last activity in the sense that we will use a sensor to create a circuit which changes when changes in the environment are detected. We can then use these measurements to create commands so the robot can “react” to its environment. In the previous section, we took advantage of mechanical changes to the whiskers that were induced by the presence of objects, while here we will take advantage of chemical changes to the phototransistors due to incident light.

We can use these phototransistors for a variety of different purposes. Since they detect variations in visible light, we can program the Boe-Bot to stay away from dark areas, report the overall brightness that it sees, or seek out light sources. In this activity, we will only concern ourselves with programming the Boe-Bot to seek out light sources.

6.1 Building and Testing the Phototransistor Circuits

As usual, the first thing that we need to do is build and test the hardware. This section will be very similar to the building and testing phase of the whisker activity.

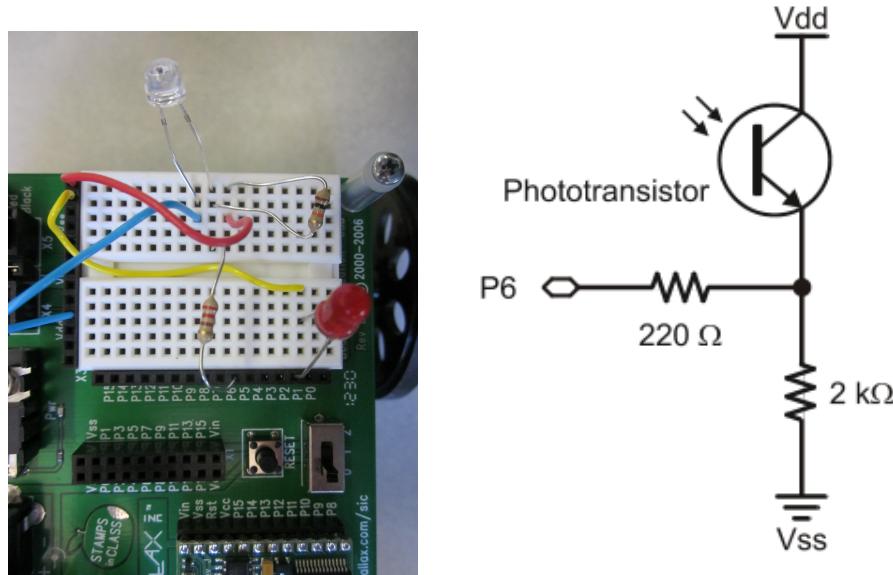


Figure 19: Whiskers wired to bread board (left) and wiring schematic (right).

1. Disconnect all power from your board and servos.
2. Build the circuit shown in Figure 19.
3. Reconnect power to your board.
4. Enter, save, and run the code in Code 13. Notice that this is very similar to the “TestWhiskers.bs2” adapted for use with the phototransistors. Save this code as “TestPhototransistors.bs2”.
5. Verify the following:
 - a) With no shade, both IN6 shows a value 1.
 - b) When you use your hand to cast a shadow over either of the phototransistors, the input register should change from 1 to 0.
 - c) If you are having trouble verifying either of these things, alert your lab instructor to discuss troubleshooting tips.
6. Now that you have verified the circuit functionality, try replacing the 2 kΩ resistors with each of the following resistances: 470 Ω, 1 kΩ, 4.7 kΩ, and 10 kΩ. Remember

to disconnect power before altering the circuit. Determine which combination that you think produces the best results, and use it for the next few activities.

Code 13: Test Phototransistors

```
' TestPhototransistors.bs2
' Display what the I/O pins connected to the phototransistor
' voltage dividers sense.

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

counter VAR BYTE

For counter =1 to 100          ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

DEBUG "PHOTOTRANSISTOR STATE", CR,
      "State", CR,
      "-----"

DO
    DEBUG CRSRXY, 0, 3,
        "P6 = ", BIN1 IN6
    PAUSE 100
LOOP
```

6.2 How the Phototransistor circuit works

The operation of the phototransistor circuit is similar to that of the whisker circuit that was used in the previous activity. V_{dd} will provide a constant potential of 5 V and we will monitor the voltage at P6 through the I/O pin in order to determine the condition of the phototransistor, which in turn will give us insight as to type of ambient light that is present around the Boe-Bot.

In the whisker circuit, it was stated that when the voltage at the I/O pin is equal to 5 V the register will store a 1, and when the voltage at the I/O pin is 0 V, the register will store a 0. In reality, the voltage at the I/O pin does not actually need to be exactly 5 V in order for the input register to store a 1. Actually, any voltage that is higher than about 1.4 V will cause the register to store a 1.

A resistor “resists” the flow of current. Voltage in a circuit with a resistor is similar to water pressure. For a given amount of electric current, more voltage (pressure) is built across a larger resistor than a smaller one. If you instead keep the resistance constant and vary the current, you can measure a larger voltage (increased pressure) across the same resistor with an increase in current and similarly less voltage with less current.

The phototransistor lets more current pass through with more light. Thus, in a very light room, the voltage measured across the $2\text{ k}\Omega$ resistor is very high. This implies that in the presence of a large amount of ambient light, the voltage at P6 will be very high. If the amount of ambient light in the room is small, less current is allowed to flow and there is a smaller voltage across the $2\text{ k}\Omega$ resistor, so the measured voltage at P6 will be low. In summary, if we monitor the voltage at P6, abundant light exposure will increase the voltage at P6 and cause a 1 to be stored. Similarly, a lack of light will cause the voltage at P6 to decrease, and a 0 will be stored.

Given the discussion above, we can conclude that altering the $2\text{ k}\Omega$ resistor value changes the sensitivity of the circuit depending on if it is increased or decreased. The $2\text{ k}\Omega$ resistor was chosen so that the voltage at P6 will lie slightly above the 1.4 V threshold in a “well lit” room. However, since “well lit” is subjective based on your setting you may need to alter the $2\text{ k}\Omega$ resistor so that you get the desired results.

6.3 Roam and Avoid Shadows Like Objects

All that we have to do now in order to make the Boe-Bot react to changes in the ambient light is to add a conditional IF...THEN statement that provides instructions for how the Boe-Bot should react to each possible scenario. Therefore, we just need to modify a few of the statements in “RoamingWithWhiskers.bs2” and we will have a code that tells the Boe-Bot to avoid shadows as if they are objects.

1. Open the program “RoamingWithWhiskers.bs2” and save it as “RoamingWithPhototransistors.bs2”.
2. Make the modifications shown in Code 14.
3. Save the code.
4. Reconnect power to your board and servos.
5. Run and test the program. As before, the first time that you load the code with the Boe-Bot connected to the computer, make sure that the drive wheels are not touching the ground to avoid damaging equipment. Once the code is loaded, disconnect the USB, place the Boe-Bot on the floor, and press the “reset” button.

The code used in the previous procedure is extremely similar to that used in the Roaming with Whiskers activity. Therefore, we omit an explanation of the code. However, you should be able to adequately explain what is going on in the code. If you cannot, ask your lab instructor for an explanation.

Code 14: Avoid Shadows Like Objects

```
' RoamingWithPhototransistors.bs2
' Boe-Bot detects shadows phototransistors voltage divider circuit and
' turns away from them.

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

DEBUG "Program Running!"

pulseCount      VAR     BYTE          ' FOR...NEXT loop counter.
counter        VAR     BYTE

For counter =1 to 100                  ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

DO
    IF IN6 = 0 THEN
        GOSUB Back_Up
        GOSUB Turn_Left
        GOSUB Turn_Left
    ELSE
        GOSUB Forward_Pulse
    ENDIF
LOOP

Forward_Pulse:                         ' Send a single forward pulse
    PULSOUT 13, 850
    PULSOUT 12, 650
    PAUSE 20
    RETURN

Turn_Left:                            ' Left Turn 90 degrees
    FOR pulseCount = 0 TO 20
        PULSOUT 13, 650
        PULSOUT 12, 650
        PAUSE 20
NEXT
```

```

RETURN

Turn_Right:                                ' Right Turn 90 degrees
FOR pulseCount = 0 TO 20
    PULSOUT 13, 850
    PULSOUT 12, 850
    PAUSE 20
NEXT
RETURN

Back_Up:                                    ' Back up.
FOR pulseCount = 0 TO 40
    PULSOUT 13, 650
    PULSOUT 12, 850
    PAUSE 20
NEXT
RETURN

```

6.4 Getting More Information from Your Phototransistors

So far we've only been using binary measurements from the phototransistor, that is, the only information that we have been gathering from the phototransistor is whether or not the ambient light in an area is above or below a given threshold. With a few clever manipulations, we can actually use these same sensors to gather more specific information about the actual level of ambient light that is present in an area. In other words, instead of just registering a binary measurement (0 or 1), our input variable will now be able to take on an entire range of values which will allow us to perform more precise tasks.

In order to do this, it is necessary to introduce a new type of circuit called a *transistor-capacitor circuit*, or TC circuit for short. You should already be familiar with transistors from the previous activities, but we have not yet introduced capacitors. Unfortunately, we will not have time to go into much detail about how we can use capacitors to their full potential, so we will only provide a brief introduction here.

A *capacitor* is an electrical component that is able to store a charge. They are fundamental building blocks of many circuits, and are utilized in virtually every electronic device. The amount of charge that a capacitor is able to store is measured in units called Farads (F). One Farad represents a fairly large amount of charge storage capability, so for the Boe-Bot, we will be using capacitors that are on the order of μF (1 millionth of a Farad). Figure 20 shows a schematic for a 0.1 μF capacitor along with a drawing of the part. Notice that the 104 marking on the capacitor indicates its value.

We will now build the RC circuit that will be used in this activity. Refer to Figure 21 for a diagram of the circuit.

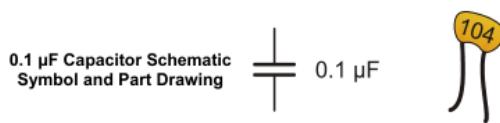


Figure 20: Capacitor.

1. Disconnect power from your board and servos.
2. Build the TC circuit shown in Figure 21.
3. Reconnect power to your board.
4. Enter, save, and run the code that is shown in Code 15. Save it as “TestPhototransistors2.bs2”.
5. Cast a shadow over the phototransistor connected to P6 to verify that the time measurement gets larger as the environment gets darker.
6. Point the phototransistor’s light collecting surface directly at an overhead light source. Verify that the time measurement gets very small.
7. Modify the code to test the phototransistor that is connected to P3. Verify that it is working correctly by repeating the previous step.

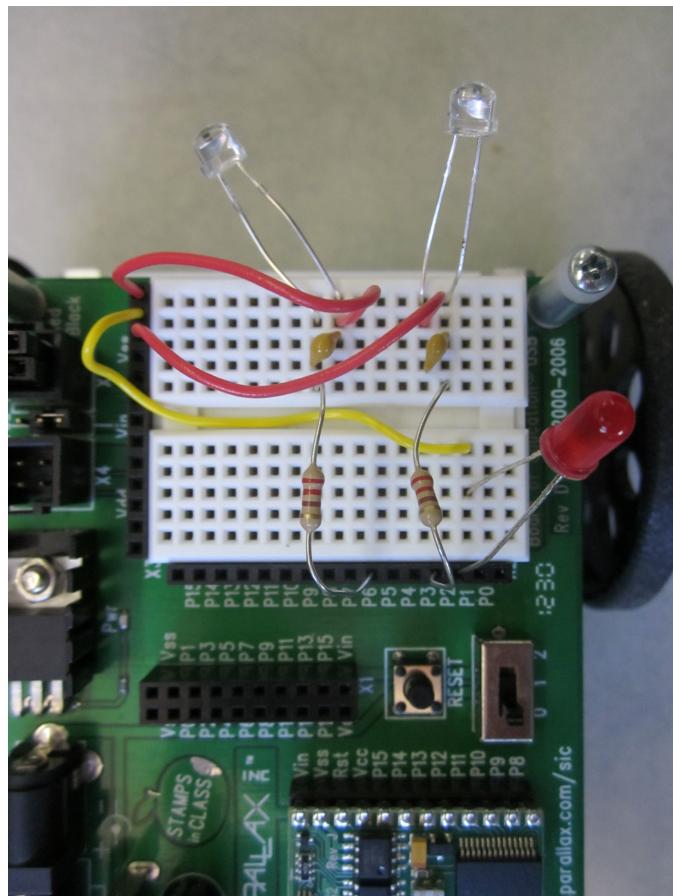
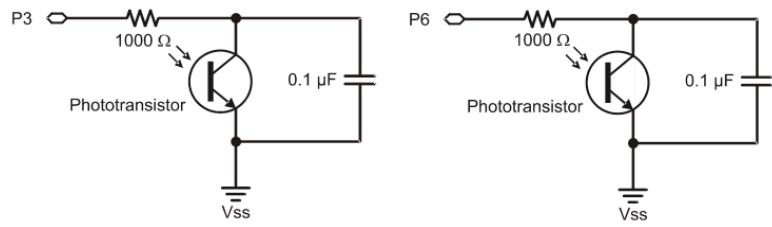


Figure 21: Phototransistor circuit with capacitor.

Code 15: Test Light Measurements

```
' TestPhototransistors2.bs2
' Test Boe-Bot phototransistor circuit connected to P6 and display
' the decay time.

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

timeLeft      VAR      WORD
counter       VAR      BYTE

For counter =1 to 100           ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

PAUSE 1000

DO
    HIGH 6
    PAUSE 1
    RCTIME 6, 1, timeLeft

    DEBUG HOME, "timeLeft = ", DEC5 timeLeft
    PAUSE 100
LOOP
```

6.5 How the Code Works

We can think of the capacitor in the TC circuit as a tiny rechargeable battery. When a 5 V signal is put through the pin P6, the capacitor will essentially charge up to almost 5 V in only a few ms. When the voltage signal at the pin P6 is reduced to 0, the capacitor will then become the power source, and will lose its charge through the phototransistor. Since the phototransistor controls current, it will also controls the voltage drop across it. This means that the voltage across the phototransistor will decrease as the capacitor loses charge. The speed at which the voltage drops depends on the current flowing through the phototransistor, and we already know that the current flowing through the phototransistor depends on the amount of light present. Therefore, the time that it takes for the voltage across the phototransistor to drop below 1.4 V is a measurement of how much light is present in the room. So, in order to get a measurement of how much light is present, we need to write code that will send a voltage signal to the pin to

charge the capacitor, reduce the signal, and measure how fast the voltage decays. If we do this repeatedly, we can have an almost continuous measurement of the light that is present in the room.

In the code, we use a combination of the HIGH and PAUSE commands in order to send 5 V and 0 V signals to the pin. The HIGH command simply outputs a 5 V signal to the appropriate pin for a specified amount of time, and the PAUSE command causes the code execution to stop momentarily. The code also uses a command called RCTIME, which will measure the amount of time that it takes for the voltage across the phototransistor to decay. The syntax for the RCTIME command is

RCTIME *Pin, State, Duration.*

The *Pin* argument tells the code which pin we want to measure voltage from. The *State* argument determines whether we are starting below 1.4 V and measuring the time that it takes for the capacitor to charge (*State* = 0), or if we are starting above 5 V and measuring the amount of time that it takes for the circuit to discharge (*State* = 1). For our purposes, we wish to measure the discharge time so this argument should be set to 1. *Duration* is the variable to which the decay time will be stored, in increments of $2\mu s$.

To measure the decay time, we start by declaring a variable *timeLeft* that will store the decay time of the TC Circuit. We then start a D0...LOOP in order to continuously execute the main part of the code. The next 3 lines of code charge the capacitor, measure the TC decay time, and store it in the *timeLeft* variable. The DEBUG window is then opened, and used to display the measurement.

6.6 Following the Light

We will now expand upon the concepts introduced in the previous subsection in order to write a code that will allow the Boe-Bot to seek out light sources. This activity works best if the phototransistor light-collecting surfaces are pointing upward and outward, as in Figure 22.

The main idea of following the light is to program the Boe-Bot to move straight ahead when the differences between the phototransistor measurements is small, and turning toward the smaller phototransistor measurement when there is a large difference between the two measurements. In other words, this means that the Boe-Bot will turn toward the light. In this program, to take into account sensitivity issues with the phototransistors, we will measure the average of the *timeLeft* and *timeRight* measurements and uses it to set the difference that's needed to justify a turning pulse. For a more detailed discussion on why this is necessary, refer to [1].

1. Enter, save, and run the code shown in Code 16. Save it as “FollowingTheLight.bs2”.
2. Take it to a dark area of the room. Use a flashlight to verify that the Boe-Bot follows the flashlight.
3. Adjust the following threshold values in order to achieve better performance:

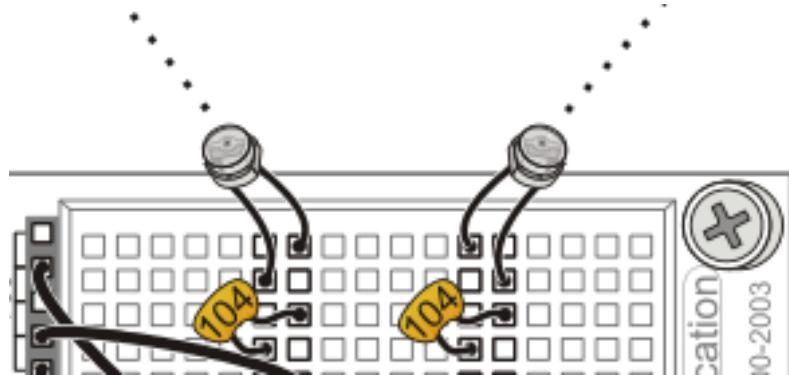


Figure 22: Configuring phototransistors to roam towards the light.

- The Threshold value in the last ELSEIF statement (that is, where it says timeLeft < 00800). Do you need to increase, or decrease this argument?
- The PAUSE values in the “Test_Phototransistors” subroutine. Do you need to increase, or decrease these arguments?

See if you can explain how the code works. If you cannot, refer to [1] or alert one of your lab instructors.

Code 16: Following the Light

```

' FollowingTheLight.bs2
' Boe-Bot roams, and turns away from dark areas in favor of brighter
' areas.

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

DEBUG "Program Running!"

' Declare variables for storing measured RC times of the
' left & right phototransistors.

counter      VAR      WORD
timeLeft     VAR      WORD
timeRight    VAR      WORD
average      VAR      WORD
difference   VAR      WORD

```

```

For counter =1 to 100           ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

DO
    GOSUB Test_Phototransistors      ' For mismatched phototransistors,
    GOSUB Average_and_Difference    ' refer to [1],
    GOSUB Navigate
LOOP

Test_Phototransistors:
    HIGH 6                         ' Left RC time measurement.
    PAUSE 3
    RCTIME 6,1,timeLeft

    HIGH 3                         ' Right RC time measurement.
    PAUSE 3
    RCTIME 3,1,timeRight
RETURN

Average_And_Difference:
    average = timeRight + timeLeft / 2
    difference = average / 6
RETURN

Navigate:
    ' Shadow significantly stronger on left detector, turn right.
    IF (timeLeft > timeRight + difference) THEN
        PULSOUT 13, 850
        PULSOUT 12, 850
    ' Shadow significantly stronger on right detector. turn left.
    ELSEIF (timeRight > timeLeft + difference) THEN
        PULSOUT 13, 650
        PULSOUT 12, 650
    ELSEIF (timeLeft < 00800)
        PULSOUT 13, 850
        PULSOUT 12, 650
    ENDIF
    PAUSE 10
RETURN

```

Activity 7: Navigating with Infrared Headlights

We've already seen that we can detect the presence of objects by attaching whiskers to the front of the Boe-Bot. However, this is a fairly cumbersome way to achieve object detection because it relies on a mechanical device (the whiskers) to physically come into contact with an obstacle. To see why this could be a problem, imagine a manufacturing environment in which a robotic arm is required to pick up a product and move it to a different area. Using whiskers to detect when the arm has come into contact with the object is not desirable because it risks damaging the product with scratches, etc. The robotic arm may also be operating in a limited space, so it would be undesirable to have any kind of large, protruding piece of hardware such as whiskers. This example motivates the use of other sensors that do not require physical contact to function. One such sensor is an infrared headlight. Infrared headlights are inexpensive and relatively simple to use, which makes them a logical choice for many robotic applications.

Infrared headlights function by sending out a low frequency beam of non-visible light and then detecting the light that gets reflected back; Figure 23 gives an illustrative example. Therefore, if an obstacle is in the robot's path, a large amount of infrared light will be reflected back to the sensor. We can then use the same type of conditional statements previously used in order to steer the robot appropriately.

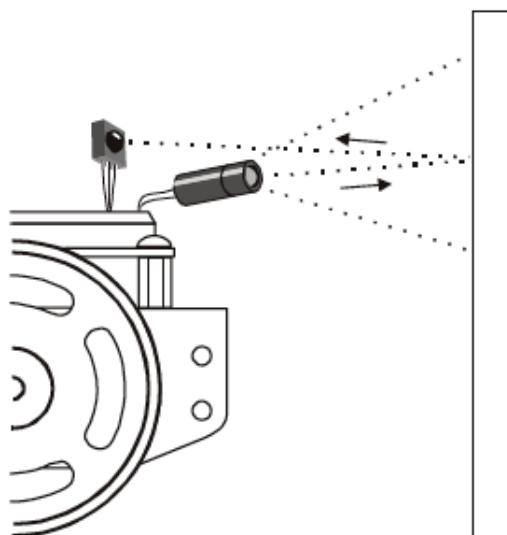


Figure 23: Boe-bot using infrared sensors to detect objects.

In this activity, we will set up and test the infrared headlights. The infrared sensors that we will be using have built-in optical filters that allow very little light except for the 980 nm infrared that we want to detect. It also has an electronic filter that only allows signals around 38.5 kHz to pass through. In other words, the sensor is only looking for infrared that is flashing on and off 38,500 times per second. This way, we can prevent

most IR interference that results from common light sources such as sunlight and indoor lighting. Despite all of the filters, infrared sensors are still somewhat sensitive to ambient light, so you may need to adjust your circuit in order achieve your desired performance. Once the hardware is calibrated, we will then experiment with programming the Boe-Bot for object and edge detection.

7.1 Building and Testing the IR Pairs

As usual, the first step is to build and test the hardware. This section will guide you through doing so. The hardware that we'll be using is shown below in Figure 24.

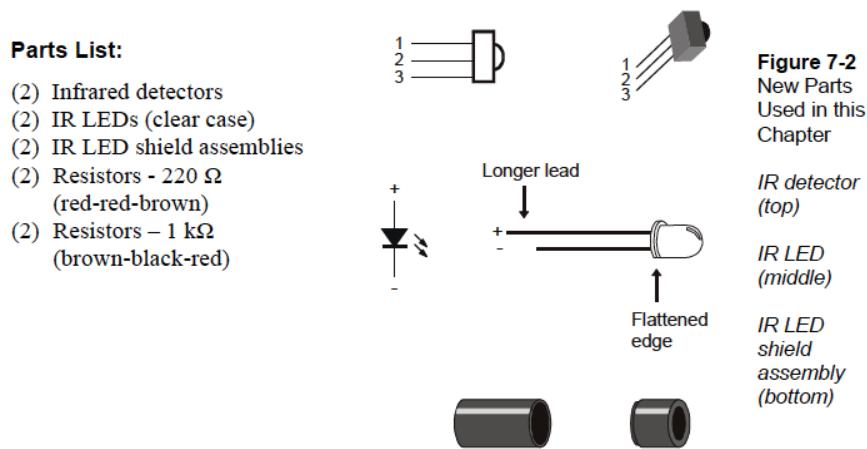


Figure 24: Components needed for infrared circuit.

1. Insert the infrared LED into the shield assembly as shown in Figure 25.

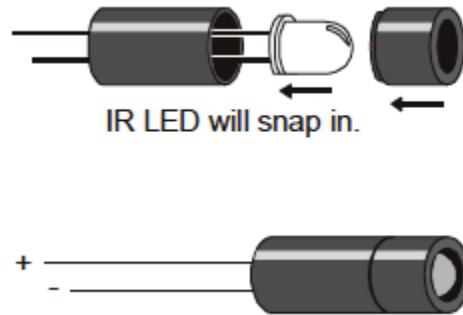


Figure 25: Assembling the infrared detector.

2. Make sure that the LED snaps into the larger part of the housing.
3. Snap the smaller part of the housing over the LED case and onto the larger part.
4. Disconnect power from your board and servos.
5. Build the circuit shown in Figure 26. Note that the small triangle symbol is the symbol for a diode. Recall that a *diode* is essentially a one-way resistor, meaning that it acts like a resistor when current flows in the direction of the point of the triangle, but allows no current flow from the other direction. The infrared headlight is essentially a diode, and therefore it is crucial that it is installed in the correct orientation. The correct orientation is shown in the Figure as well.
6. Enter, save, and run Code 17. Save the code as “TestIRPair.bs2”.
7. Leave the Boe-Bot connected to the computer, as you will be utilizing the DEBUG terminal to test functionality.
8. Place an object (such as your hand) about an inch in front of the left IR pair. Verify that the debug terminal shows 0. Remove your hand and verify that the debug terminal shows a 1. If you do not get the desired results, refer to [1] for some troubleshooting tips.
9. Modify the code as necessary (change the variable name to *irDetectRight*, change the FREQOUT command’s Pin argument from 8 to 3, and change the input register monitored by *irDetectRight* variable from IN9 to IN0), and repeat the tests from the previous two steps to verify that the IR pair on the right is functioning properly.

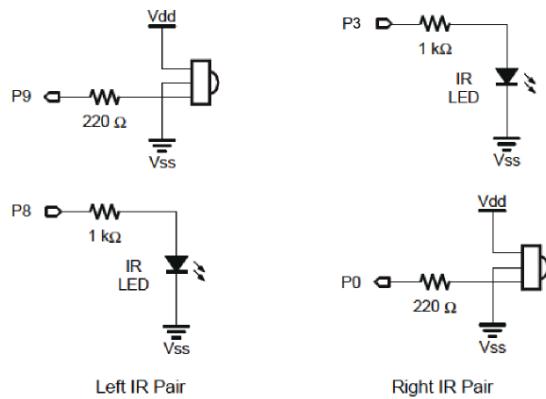
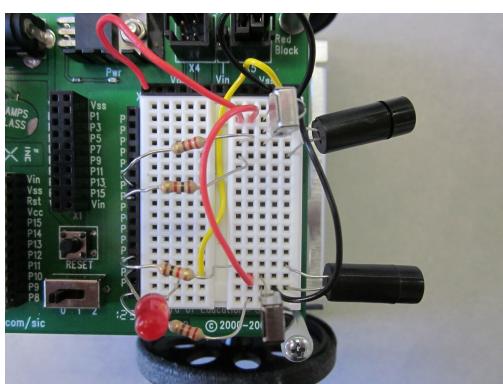


Figure 26: Infrared sensor wired to bread board (left) and wiring schematic (right).

Code 17: Test Left IR Pair

```
' TestIrPair.bs2
' Test IR object detection circuits

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

irDetectLeft    VAR     BIT
counter        VAR     BYTE

For counter =1 to 100          ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

DO
    FREQOUT 8, 1, 38500
    irDetectLeft = IN9

    DEBUG HOME, "irDetectLeft - ", BIN1 irDetectLeft
    PAUSE 100
LOOP
```

7.2 How The Code Works

The circuit itself is very simple. The headlight is a diode connected in series with a resistor and a voltage source, while the detector simply emits a HIGH or LOW signal based on the presence of IR light at 38,500 Hz.

The code that you used to test the IR pairs relies on a function called FREQOUT in order to send an IR signal at the desired frequency. The syntax of the FREQOUT command is as follows:

FREQOUT *Pin*, *Duration* (ms), *Frequency* (Hz)

Therefore, when we give the command FREQOUT 8,1,38500, the IR headlight that is attached to Pin 8 will emit a signal that is 38.5 kHz, and will last 1 ms. The key to making each IR LED/detector pair work is to send a pulse, and then immediately store the IR detector's output as a variable that we can use. It is important that we do this immediately after the FREQOUT command, since the reflected signal that the detector can detect goes away quickly.

By putting these commands into a DO...LOOP, the code will repeatedly be sending IR signals (with a small pause in between iterations). Since this happens very fast, to you it will seem like the IR pair is monitoring instantaneously.

7.3 Infrared Detection Range Adjustments

The IR LEDs function in much the same way as regular LEDs in the sense that we can control how “bright” their light is by adjusting the amount of current that is flowing through them. The term “bright” is really only applicable to visible light, so to extend this notion to IR light we will use the term intensity. *Intensity* is a measure of how much energy is contained in a given light wave. By altering the resistance that is connected in series with the IR LED, we can change the intensity of the light that it emits. With a more intense light source, it would be reasonable to expect that more light will be reflected off objects even if they are far away.

It turns out that this is exactly the case. Ohm’s Law tells us that less resistance equals more current, and vice versa, therefore, if we decrease resistance, the IR LED will emit more intense light and we should be able to detect objects that are far away.

1. Disconnect power from the board and servos.
2. Replace the $1\text{ k}\Omega$ resistors in your circuit with 470Ω resistors and run the code that you wrote in the previous sections to see how it affects the performance of the LED sensors. Were the results as you expected?
3. Repeat this procedure with $220\ \Omega$ resistors, and $4.7\text{ k}\Omega$ resistors. Make sure to disconnect power before making any changes to the circuitry. Choose the resistor combination that gives you the best performance, i.e., gives you a reasonable threshold for object detection. You will use these resistor values for the remainder of the activities that use IR headlights. Note that the resistors used for the two IR pairs do not have to be the same!

7.4 Object Detection and Avoidance

You may have noticed that the IR detectors have outputs that are identical to the whiskers. Because of this, we can modify the code that we used for object detection/avoidance with whiskers to work with IR detectors by making only a few slight changes.

1. Open “RoamingWithWhiskers.bs2”.
2. Modify it so that it matches the program shown in Code 18.
3. Reconnect power to the board and servos.
4. Save and run the program. Save the program as “RoamingWithIr.bs2”. If functioning properly, your Boe-Bot should roam while trying to avoid obstacles in its path.

Code 18: Roaming With Headlights

```
' RoamingWithIr.bs2
' Adapt RoamingWithWhiskers.bs2 for use with IR pairs.

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

DEBUG "Program Running!"

irDetectLeft    VAR     BIT
irDetectRight   VAR     BIT
pulseCount      VAR     BYTE
counter         VAR     BYTE

For counter =1 to 100                      ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

DO
    FREQOUT 8, 1, 38500                  ' Store IR detection values in
    irDetectLeft = IN9                   ' bit variables.

    FREQOUT 3, 1, 38500
    irDetectRight = IN0

    IF (irDetectLeft = 0) AND (irDetectRight = 0) THEN
        GOSUB Back_Up
        GOSUB Turn_Left
        GOSUB Turn_Left
    ELSEIF (irDetectLeft = 0) THEN          ' Left IR pair detects
        GOSUB Back_Up
        GOSUB Turn_Right
    ELSEIF (irDetectRight = 0) THEN         ' Right IR pair detects
        GOSUB Back_Up
        GOSUB Turn_Left
    ELSE
        GOSUB Forward_Pulse
    ENDIF
LOOP

Forward_Pulse:                                ' Send a single forward pulse
    PULSOUT 13, 850
```

```

PULSOUT 12, 650
PAUSE 20
RETURN

Turn_Left:                                ' Left Turn 90 degrees
FOR pulseCount = 0 TO 20
  PULSOUT 13, 650
  PULSOUT 12, 650
  PAUSE 20
NEXT
RETURN

Turn_Right:                               ' Right Turn 90 degrees
FOR pulseCount = 0 TO 20
  PULSOUT 13, 850
  PULSOUT 12, 850
  PAUSE 20
NEXT
RETURN

Back_Up:                                    ' Back up.
FOR pulseCount = 0 TO 40
  PULSOUT 13, 650
  PULSOUT 12, 850
  PAUSE 20
NEXT
RETURN

```

7.5 The Drop-Off Detector

In the previous section, you wrote a code in order to steer the Boe-Bot away from objects. In other words, we programmed the robot to change its state to try and obtain a position such that the IR detector did not detect an obstacle. In this section, we will do the opposite: we want the Boe-Bot to remain in a state such that the IR detector DOES detect an obstacle. With this mindset, we can write a very simple edge detection code that will attempt to keep the Boe-Bot from driving off of the edge of a table.

The way that we do this is fairly simple. We angle the LED headlight and the detectors downward, so that if the table is directly in front of the Boe-Bot, the detector will register a 0. When it approaches the edge, there will be no surface directly in front of the Boe-Bot so the detector will register a 1. We will use almost same code as in the previous activity. Note that your group must use great care when testing your code during this activity. Allowing the Boe-Bot to fall off of a table top will most likely damage or break

the hardware. Therefore, it is EXTREMELY important that you have someone spotting the robot while it is moving to make sure that it does not fall.

1. Disconnect power from your board and servos.
2. Replace the $1\text{ k}\Omega$ resistors with $2\text{ k}\Omega$. For this activity, we want the Boe-Bot to be “nearsighted”.
3. Angle the IR LEDs and the IR detectors toward the ground.
4. Open the program “TestIrPair.bs2”. When you have reached this point, alert your lab instructors, and they will assist you in testing your hardware setup. Run the code to verify that the Boe-Bot detects the edge of the table. If everything works as it should, then move on to the next step.
5. Open the code “RoamingWithIR.bs2”. Modify the code by changing all of the 0’s in the conditional IF and ELSEIF statements to 1’s.
6. Connect the Boe-Bot to the computer, and load the code onto the BASIC STAMP. Once you have done this, disconnect your Boe-Bot, set it on the table, and run the code. IMPORTANT - make sure there is someone “spotting” at all times to make sure that the Boe-Bot does NOT fall off of the table edge.

Activity 8: Robot Control with Distance Detection

In the previous Activity, we only relied on binary measurements, meaning that the measurements that we are using to make decisions only have two possible outcomes. For example, in the edge detection activity of the previous section, each IR detector stored a 0 or a 1 depending on whether an edge was present. These type of binary sensor outputs are great for simple applications that only require rough measurements, but what if we wanted to do something more precise? What if our decisions are based on a measurement that can take an infinite number of values, such as distance or speed? In this case, the code that we write cannot be as simple as a series of conditional IF...THEN statements. To complicate matters, all hardware has a certain amount of inherent error, which can cause a lot of problems in high precision applications.

So how do we deal with these issues? In this section, we will introduce one of the simplest, yet most effective forms of control, the proportional controller.

The basic idea can be illustrated with a simple example. Imagine you are programming a remote control car to maintain a certain speed, say 15 miles per hour. From our system model, we can predict the amount of power that we need to send to the servos in order to make the car go our desired speed. However, because of variations in the terrain on which we are operating, when we provide the car with our predicted signal the actual speed of the car fluctuates between 14 and 16 miles per hour. The way that we can correct this is by continuously adjusting our input signal to compensate for the error between our desired speed and our actual speed. However, since we do not want to manually control the car, we need to come up with a methodical system for making these changes automatically. With a proportional controller, the way that this is done is as follows. At each time instant, we take a measurement of the actual speed of the car and subtract it from our desired speed. We then multiply the result by a constant (called the *proportional gain*) and use this as our new input. For example, suppose a proportional gain of $k_p = 3$, and at some time instant t we record a speed of 16 mph. If we want our speed to move back towards 15 mph then the new input to the system at time t will be

$$u(t) = (15 \text{ mph} - 16 \text{ mph}) * k_p = -1 * 3 = -3.$$

By doing this continuously, we can maintain our desired speed. As a function of time, t , the input will be

$$u(t) = (\text{Desired Speed} - \text{Actual Speed at Time } t) * k_p.$$

If this is not clear at this point, don't worry, it will become more clear as the activity progresses.

In this activity, we will first demonstrate how to use our IR circuit in order to take distance measurements. We will then demonstrate the use of a proportional controller to perform a shadowing task and a route following task.

8.1 Testing the Frequency Sweep

In the previous activity, we stated that the IR detector only detects IR signals that are emitted at 38.5 kHz. In reality, this is not entirely true. The detector will detect signals emitted at other frequencies as well, but it is most sensitive to signals at 38.5 kHz. For example, the detector is only about 20% as sensitive to an IR signal at 42 kHz as it is to a signal at 38.5 kHz. A plot of relative sensitivity vs. carrier frequency is shown in Figure 27. We can actually use this fact to obtain measurements of the distance between our robot and an object. If we emit an IR signal with a less sensitive frequency, an object will need to be much closer in order for our detector to pick it up. Therefore, in order to obtain a distance measurement, all that we need to do is to emit IR signals at different frequencies, and see which frequencies our detector picks up. We will classify distances according to 5 zones, as illustrated by Figure 28.

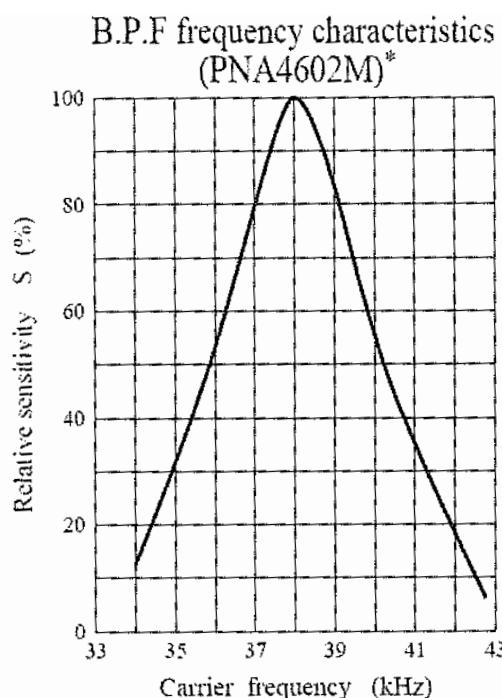


Figure 27: Relative Sensitivity vs. Carrier Frequency

1. Disconnect power from the board and servos.
2. Replace the resistors that are in series with the IR LEDs with resistors that give the best performance, as determined by your experiments in Activity 8.
3. Restore power to your board and servos.

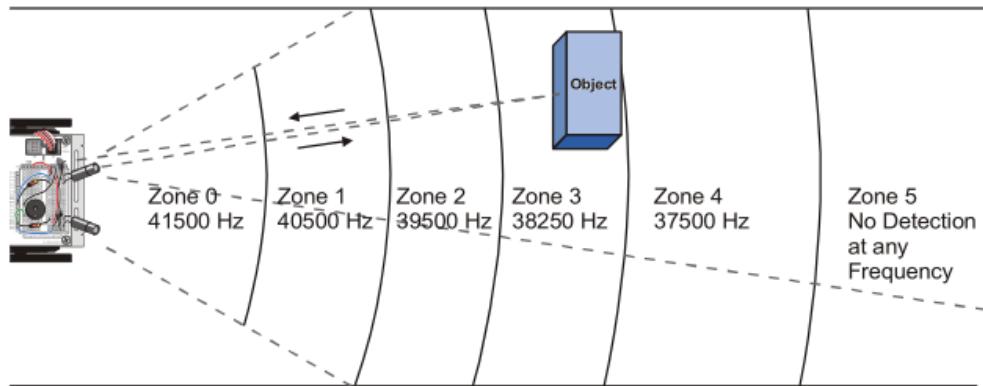


Figure 28: Relative Sensitivity vs. Carrier Frequency

4. Enter, save, and run the program “TestFrequencySweep.bs2”, shown in Code 19.
5. Use a sheet of paper or a notecard facing the IR LED/ detector to test the distance detection.
6. Verify that the “zone” value gets larger as the card is moved further away.
7. Adjust the resistances as needed to give you the best performance. Make sure to disconnect power from the board before making any circuitry changes.

Code 19: Test Frequency Sweep

```

' TestFrequencySweep.bs2
' Tests the Boe-Bot's distance detection

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

freqselect    VAR Nib
irfrequency   VAR Word
irdetectleft  VAR Bit
irdetectright VAR Bit
distanceleft  VAR Nib
distanceright VAR Nib
counter       VAR Byte

For counter =1 to 100          ' Signal program start/reset.
    PULSOUT 2, 750

```

```

PAUSE 20
NEXT

DEBUG CLS,
    "ir object zone",CR,
    "left      right",CR,
    "----      ----"

DO
    GOSUB get_distances
    GOSUB display
LOOP

get_distances:
    distanceleft = 0
    distanceright = 0

    FOR freqselect = 0 TO 4
        LOOKUP freqselect, [37500,38250,39500,40500,41500],irfrequency
        FREQOUT 8,1,irfrequency
        irdetectleft=IN9
        distanceleft = distanceleft+irdetectleft

        FREQOUT 3,1,irfrequency
        irdetectright=INO
        distanceright = distanceright+irdetectright

    PAUSE 100
NEXT
RETURN

display:
    DEBUG CRSRXY,2,3, DEC1 distanceleft,
          CRSRXY,9,3, DEC1 distanceright
RETURN

```

8.2 How “TestFrequencySweep.bs2” Works

The code written above only really uses one command that we haven’t seen before, the LOOKUP command. This command is used when we need to pick out a specific value from a list of values. The syntax of the LOOKUP command is

LOOKUP *Index*, [*Value*₀, ..., *Value*_{*N*}], *Variable*.

When executed, this command will look up the entry in the *Value* array corresponding to *Index*, and store it as *Variable*. In our code, we will utilize this command by putting the frequency values that we want to test into an array, and then executing the LOOKUP command into a FOR...NEXT loop in order to test each one. The code’s main routine is a DO...LOOP. At the beginning of each iteration of this loop, we initialize the variable *distance* to equal 0. Within the loop, we use a nested FOR...NEXT loop and utilize the LOOKUP command to send out frequency signals at 5 different frequencies, one on each iteration of the FOR...NEXT loop. If the IR detector is able to detect an object at a given frequency, the value of the variable *irDetect* will be 0. Conversely, if the IR detector is not able to detect an object at a given frequency, *irDetect* will be equal to 1. After the value of *irDetect* is determined, the current value of the variable *distance* is augmented by the value of *irDetect*. This way, if an object is further away, less frequencies will be able to detect an object and therefore the final value of *distance* will be higher.

The rest of the code that is inside of the FOR...NEXT loop are simply commands that format the appearance of the debug terminal. This whole process is repeated very quickly, so to the naked eye it seems as though the Boe-Bot is detecting distance almost instantaneously!

8.3 Boe-Bot Shadow Vehicle

Using the method for measuring distance that we discussed in the previous section, we can now implement a closed loop controller that will allow us to perform tasks with an accuracy that was previously impossible. One such task is that of following another vehicle or object at a fixed distance. In order to better explain how this controller works, we will introduce the notion of a block diagram. A *block diagram* is a visual map that displays the general framework for how a signal moves through a system. An example block diagram representing one of the systems that we will be building in this exercise is shown in Figure 29. Each item, or *block*, represents a point in the system where some operation is done to the signal. Arrows that go into a block represent *inputs* to a particular operation, and arrows that come out of a block represents the *output* of a particular operation. If an arrow going into a particular block does not represent the output from another block, we refer to it as an *external input*, and if an arrow coming out of a block does not represent the input to another block, it is referred to as an *external output*. The block that is labeled “system” represents the particular mechanical system that you are working with, and is sometimes called the *plant*. The round blocks are *summation* blocks that perform a simple addition or subtraction operations to signals. The rest

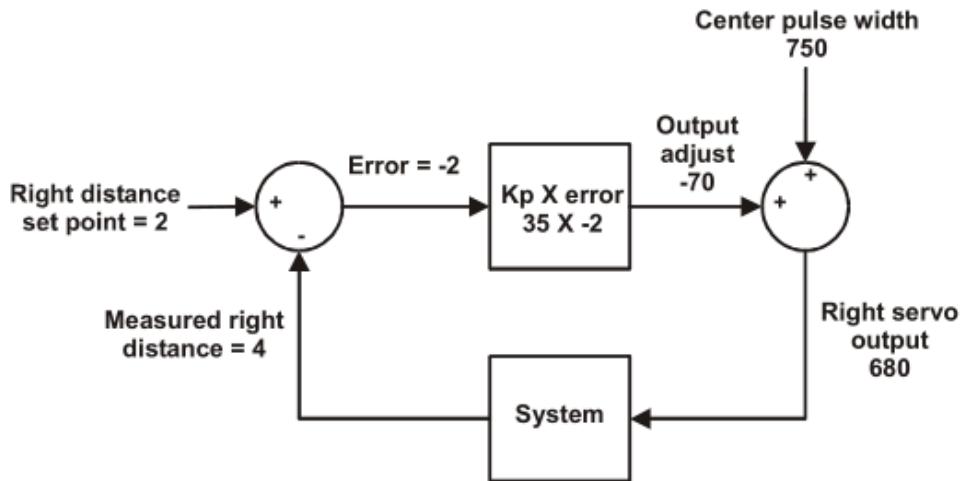


Figure 29: Example Block Diagram

of the blocks represent elements of the controller that we have designed. Although the controller can take many forms, in this particular example of proportional control the controller is simply the block that is labeled “ $k_p \times$ error”. In a typical controller design problem, we are given a plant and a desired external output, and we are charged with the task of designing a controller, external inputs, and the appropriate interconnection between these components that gives us the desired result.

In this activity, you can think of the external output of the system as the actual distance between the Boe-Bot and the vehicle that it is following. Our plant is the Boe-Bot itself, and the external input (which we can control) is our desired distance. In order to demonstrate how the code works, we will simply provide an example.

Imagine we are trying to maintain a distance of 2 behind some leader and we have chosen to use a proportional gain $k_p = 35$. At some time-step, the Boe-Bot measures the distance between itself and its leader as 4. Starting at the top left of the block diagram shown in Figure 29, the actual distance is subtracted from the desired distance in the round summation block and then enters into the controller so we have

$$\text{Error} = \text{Desired Distance} - \text{Actual Distance} = 2 - 4 = -2$$

The signal then enters the controller block, where it is multiplied by the proportional gain k_p

$$\text{Output Adjust} = k_p * \text{Error} = 35 * -2 = -70$$

In order to translate this into a valid movement command, we then add this value to our center pulse width

$$\text{Servo Output} = \text{Center Pulse Width} + \text{Output Adjust} = -70 + 750 = 680$$

So, this means at this time step, the next signal that will enter the plant will be equal to 680. Why is this significant? Careful examination of this block diagram reveals that if the actual distance is equal to the desired distance, a zero signal will enter the plant. If the actual distance is less than the desired distance, the signal entering the plant will be more than 750, and if the actual output is more than the desired output, the input to the plant will be more than 750. Doing this continuously should cause the Boe-Bot to maintain the correct distance (that is, if a certain property called *stability* of the system holds which in this case it does).

1. Enter, save, and run the following code. Save it as “FollowingBoeBot.bs2”.
2. Once the code is entered, point the Boe-Bot at a sheet of paper held in front of it as though it is an obstacle.
3. Try slowly moving the sheet of paper forward and backward. The Boe-Bot should move to maintain the correct distance.
4. Try rotating the paper in front of the Boe-Bot slowly. The Boe-Bot should turn with the paper.
5. Make adjustments to the IR headlights in order to improve performance.

Code 20: Following the Boe-Bot

```

' FollowingBoeBot.bs2
' Uses a proportional controller to follow a Boe-Bot or other
' object.

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

DEBUG "program running"

kpl      CON -35
kpr      CON 35
setpoint CON 3
centerpulse CON 750

freqselect  VAR Nib
irfrequency  VAR Word
irdetectleft  VAR Bit
irdetectright VAR Bit
distanceleft  VAR Nib
distanceright VAR Nib
pulseleft    VAR Word

```

```

pulseright    VAR Word
counter       VAR Word

For counter =1 to 100           ' Signal program start/reset.
    PULSOUT 2, 750
    PAUSE 20
NEXT

DO
    GOSUB get_distances

    pulseleft = setpoint - distanceleft*kpl+centerpulse
    pulseright = setpoint - distanceright*kpr+centerpulse

    GOSUB send_pulse
LOOP

get_distances:
    distanceleft = 0
    distanceright=0
    FOR freqselect = 0 TO 4
        LOOKUP freqselect,[37500,38250,39500,40500,41500],irfrequency

        FREQOUT 8,1,irfrequency
        irdetectleft=IN9
        distanceleft=distanceleft+irdetectleft

        FREQOUT 3,1,irfrequency
        irdetectright=INO
        distanceright=distanceright+irdetectright
    NEXT
    RETURN

send_pulse:
    PULSOUT 13, pulseleft
    PULSOUT 12, pulseright
    PAUSE 5
    RETURN

```

The functionality of the code is fairly simple. It actually runs two controllers at once,

one for each servo, which are implemented in a total of two lines. Can you explain how the code works?

8.4 Following a Stripe

With only minor modifications to the code of the previous section, we can program the Boe-Bot to perform the task of following a route mapped out by a strip of black electrical tape. Since IR light does not reflect well off of a black surface, we can think of the black stripe as the “obstacle free” area. With appropriate values of the proportional gain variables, we can tune our controller to achieve very good performance.

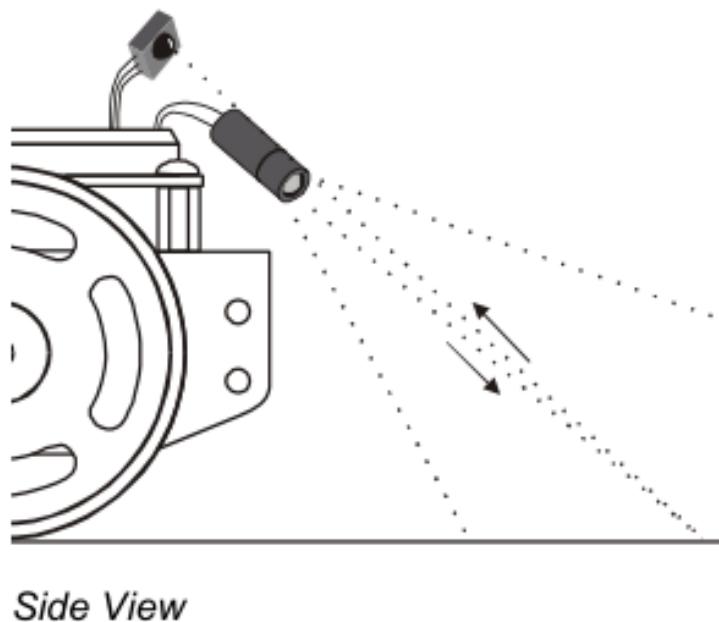


Figure 30: Orientation of Headlights for Stripe Following

1. Disconnect power from your board and servos.
2. Replace the resistors that are in series with the IR LEDs with $2\text{ k}\Omega$ resistors.
3. Angle the IR LED/detector pairs toward the ground as shown in Figure 30.
4. Open “FollowingBoeBot.bs2” and re-save it as “StripeFollowing.bs2”.
5. Modify the code so that $k_{pl} = 35$ and $k_{pr} = -35$. Note that the values of k_{pr} and k_{pl} change sign. This is extremely important because in this activity, we want the Boe-Bot to move *toward* obstacles that are closer and *away* from objects that are further away. (Why?)

6. Once your code is written, load it onto the BASIC Stamp and then bring your Boe-Bot up to the testbed that that your lab instructors have at the front of the room to test your work (or place a strip of black tape on the ground if your lab instructors have not set up the testbed for you).
7. Make adjustments to k_{pr} , k_{pl} and $SetPoint$ in order to improve performance.

Activity 9: Navigating an Obstacle Course

So far in the course we have given you a step by step procedure for programming the Boe-Bot to perform certain tasks. In this activity, you will get the chance to show off what you've learned. Your task is to write a program that will guide your Boe-Bot through the maze that your lab instructors have constructed. You are allowed to use any of the sensors that we have discussed and any of the codes that you have written. However, the robot MUST be able to navigate the maze without any human assistance. You will have an hour and a half to work with your group to come up with a code, and then we will spend the rest of the time testing out your work. Good Luck!!

References

- [1] A. Lindsay. *Robotics with the Boe-Bot*. Parallax, Inc., 2010.
- [2] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005.
- [3] K. Ogata. *Modern Control Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 5th edition, 2010.
- [4] F. Bullo, J. Cortés, and S. Martínez. *Distributed Control of Robotic Networks*. Princeton University Press, 2009.