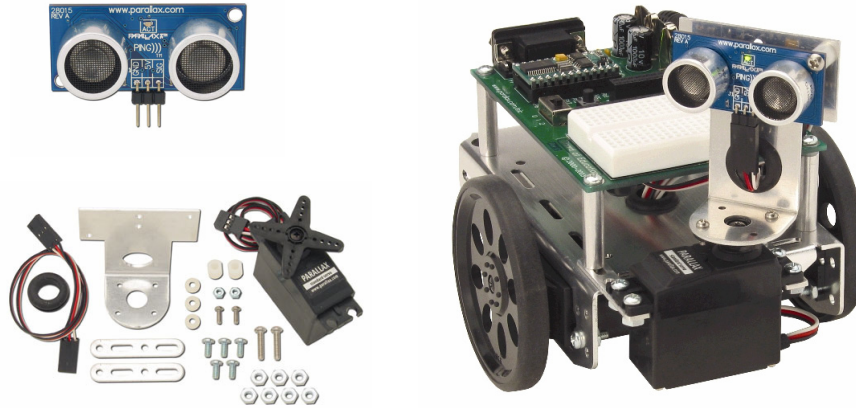


(The activities in this document are excerpts from *Smart Sensors and Applications*, a forthcoming Stamps in Class book by Andy Lindsay. © 2006 Parallax, Inc.)

#### ACTIVITY #4: PING)))DAR - A RADAR STYLE DISPLAY

Figure 8-9 shows a Boe-Bot assembled with the Ping))) Ultrasonic Rangefinder and Mounting Bracket kits. The mounting bracket kit makes it possible for the Boe-Bot to swivel the Ping))) rangefinder and measure object distances across a 180° field in front of it.

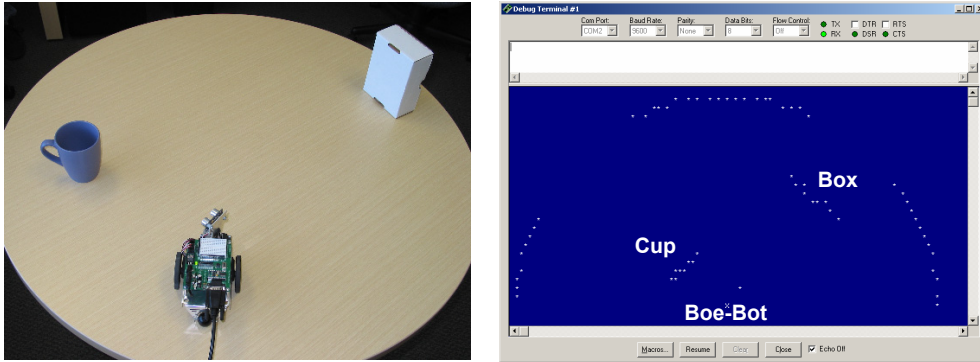
**Figure 8-9:** Boe-Bot with Ping))) Ultrasonic Rangefinder and Mounting Bracket Kit



8

This activity features a program that displays what the Boe-Bot detects in the Debug Terminal as it sweeps the Ping))) rangefinder back and forth. Figure 8-10 shows an example with a cup and box set in the Boe-Bot's 180° field of detection along with their signatures displayed in the Debug Terminal. Experimenting with this program will help you better understand what the Boe-Bot can and cannot detect with the Ping))) Ultrasonic Rangefinder and Mounting Bracket, which is important for writing programs that navigate with this object detection system.

Figure 8-10: Seeing what the Boe-Bot Detects with Ping)))Dar.bs2



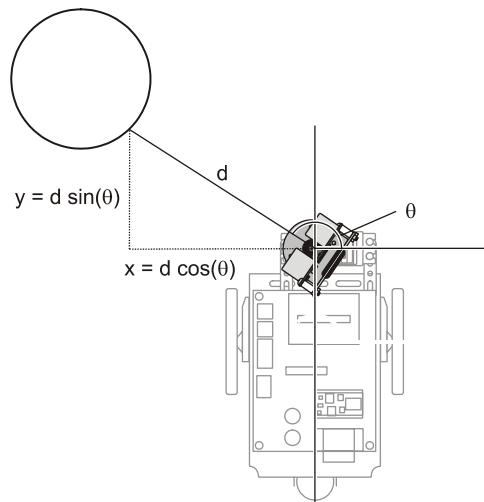
### Recommended Reading

This activity assumes that you have already completed *What's a Microcontroller* and *Robotics with the Boe-Bot*. Both are available for PDF download from [www.parallax.com](http://www.parallax.com). For this particular activity, the background material in *What's a Microcontroller*, Chapter 4 is especially important.

### Polar to Cartesian Coordinate Conversion

Ping)))Dar.bs2 will scan a 180° field in front of the Boe-Bot by incrementally rotating the Ping))) Mounting Bracket and then taking a distance measurement. It will do this rapidly enough that the mounting bracket servo will appear to just be rotating from right to left and back again as it updates the Debug Terminal display.

For each measurement in a given sweep, the information Ping)))Dar.bs2 will have to work with is a distance measurement ( $d$ ) and an angle measurement ( $\theta$ ) as shown in Figure 8-11. When coordinates are given in terms of distance and angle, they are called polar coordinates. These coordinates are typically expressed in parentheses like this: ( $d \angle \theta$ ). When telling those coordinates to someone, you would normally say, "d at an angle of theta." In order to display these measurements graphically in the Debug Terminal, Ping)))Dar.bs2 will have to convert these polar coordinates to Cartesian ( $x, y$ ) coordinates. That way, the `DEBUG CRSRXY, x, y, "*"`  command can be used to graphically display the measurement by positioning the cursor and then placing an asterisk.

**Figure 8-11:** Object's Polar and Cartesian Coordinates

Calculating the x and y axis components given polar coordinates is not difficult. The equations for x and y are shown below. The x-axis component involves multiplying the distance by the cosine of the angle, and the y component is the distance multiplied by the sine of the angle.

$$x = d \cos(\theta) \quad \text{and} \quad y = d \sin(\theta)$$

Since the BASIC Stamp is an integer math processor, the programming for making these conversions is a little different from what you might expect with a PC programming language. Ping)))Dar.bs2's **Polar\_To\_Cartesian** subroutine is explained in *Advanced Topic – Inside the Polar\_To\_Cartesian Subroutine*, which starts on page 337.

### **Parts Required**

- (1) Fully assembled and tested Boe-Bot robot
- (1) Ping))) Ultrasonic Rangefinder
- (1) Ping))) Mounting Bracket Kit

### **Assembly and Electrical Connections**

- √ Follow the instructions in the Ping))) Mounting Bracket Documentation for making the mechanical and electrical connections to the servo and Ping))) Ultrasonic rangefinder. When you are done, the servo will be connected to the Board of Education's servo port 14, and the Ping))) rangefinder will be connected to port 15.

### **Mounting Bracket Adjustments**

Keep in mind that Ping)))Dar.bs2 will measure distances and angles. Recall from *What's a Microcontroller* that a **PULSOUT** command's *Duration* argument sends a message to a standard servo telling it what rotational position to hold. In other words, the **PULSOUT** command's *Duration* argument tells the Ping))) mounting bracket's standard servo what angle to turn to.

Ping)))Dar.bs2 will need to convert the **PULSOUT** command's *Duration* argument to a binary radian measurement before the program can do polar to Cartesian coordinate conversion. Since the object's position is dependent on good distance and angle measurements, it will be important mechanically adjust and calibrate your Ping))) Mounting Bracket system so that the program's **PULSOUT** commands can sweep it from 0° (to the right) to 180° (to the left).

The first adjustment step toward good Ping))) Mounting Bracket servo angular control is making sure that the Ping))) rangefinder is mounted so that the servo points the Ping))) rangefinder straight ahead when the servo receives 1.5 ms center pulses.

- √ Run PingServoCenter.bs2.
- √ Make sure the Board of Education's 3-position power switch is set to position-2.
- √ Check to see whether the mounting bracket servo points the Ping))) rangefinder straight ahead on the Boe-Bot.
- √ If it does, then continue to the *Software Calibration for 180° Sweep* section. Otherwise, keep following the checklist instructions here.
- √ Make sure CenterPing.bs2 is running for the remaining checklist instructions in this section.
- √ Start by removing the screw that connects the servo horn to the output shaft shown in Figure 8-12 (a).

- √ Pull upward on the Ping))) rangefinder. The horn, which is attached to the Ping))) rangefinder with screws should slide up and off the servo's output shaft, also shown in Figure 8-12 (a).
- √ Orient the Ping))) rangefinder so that it is pointing straight ahead as shown in Figure 8-12 (b).
- √ Slide the servo horn back onto the output shaft.

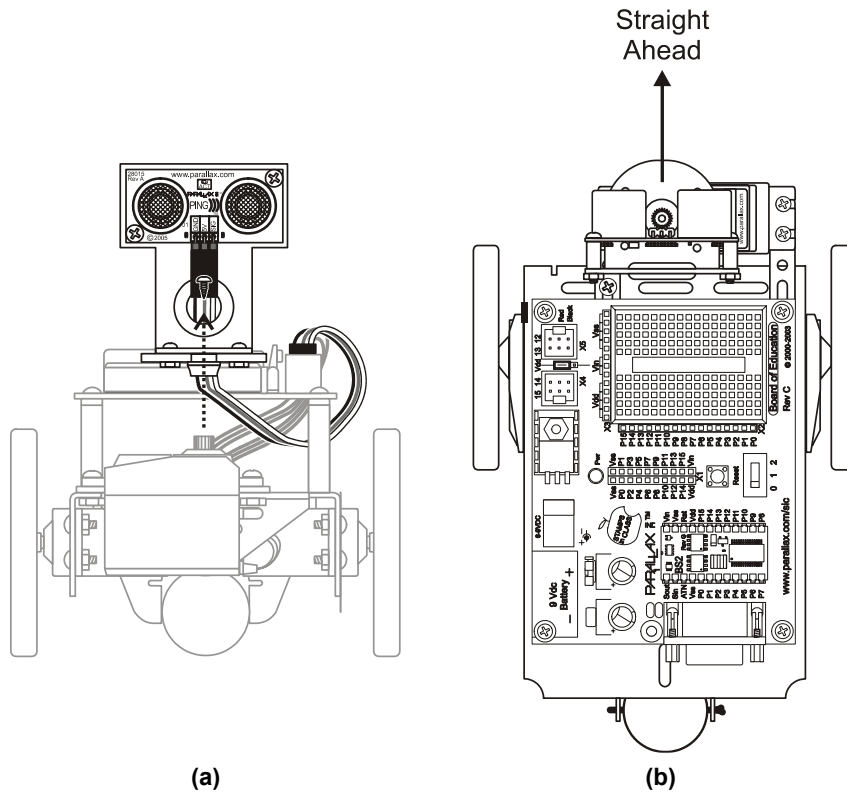


**Perfect alignment may not be possible, so choose the closest to straight ahead.**

If the teeth on the output shaft do not line up with the servo horn when it is pointing exactly straight ahead, there will be two ways that the servo horn lines up: with the Ping))) pointing slightly to the left, or slightly to the right. Choose position that is as close to straight ahead. You will then be able to fine tune straight-ahead by adjusting the screws that attach the servo to the Boe-Bot chassis.

- √ Screw the screw back in that holds the servo horn to the Ping))) mounting bracket.

Figure 8-12: Mounting the Ping))) Rangefinder so that It Points Straight Ahead



- ✓ To adjust for any slight error caused by the output shaft to servo horn gear teeth alignment, start by loosening the screws that attach the servo to the Boe-Bot chassis.
- ✓ Now, you will have some wiggle room to rotate the servo's housing slightly to make up for any offset resulting from the alignment of the servo horn and output shaft's teeth.
- ✓ Make sure to re-tighten all the screws when you are done.

### Example Program – PingServoCenter.bs2

```
' Smart Sensors and Applications - PingServoCenter.bs2  
' Send 1.5 ms "center" pulses to the servo the Ping))) rangefinder is
```

```

' attached to.

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

PingServo      PIN      14          ' Servo directs Ping)))
Center         CON      750          ' Center/0-degree pulse duration
BtwnPulses     CON      20          ' ms between servo pulses

DO
' Main loop

  PULSOUT PingServo, Center          ' Center signal -> Ping))) servo
  PAUSE BtwnPulses                  ' 20 ms delay between pulses

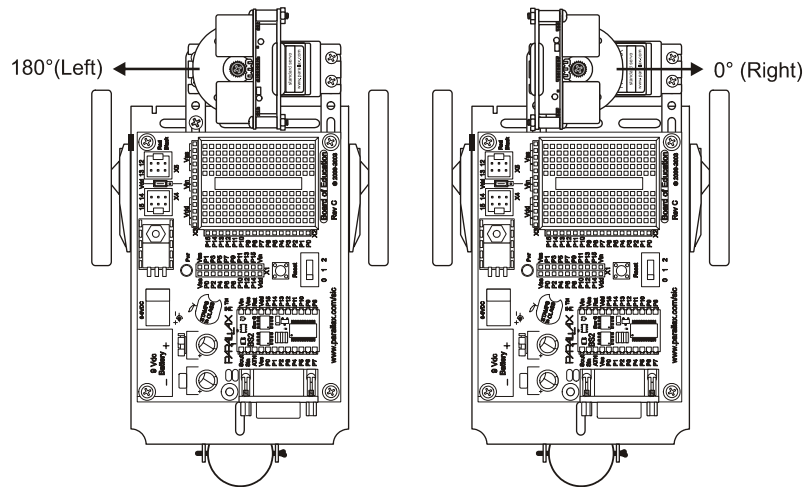
LOOP                                  ' Repeat main loop

```

### **Software Calibration for 180° Sweep**

The Boe-Bot needs to be able to point the Ping))) rangefinder from 0° (to the right) to 180° (to the left) as shown in Figure 8-13. The Ping))) mounting bracket's servo should direct the ping to 0° when it receives 0.5 ms pulses and to 180° when it receives 2.5 ms pulses. Since the `PULSOUT` command times pulse durations in terms of 2 μs units, the command `PULSOUT 14, 250` will cause the servo to rotate the Ping))) rangefinder to 0°, and `PULSOUT 14, 1250` will direct it to 180°.

Figure 8-13: Angle and Ping))) Rangefinder Direction



The `TestPingDirection.bs2` program declares `LimitRight` and `LimitLeft` as constants equal to 250 and 1250. The program also declares `PingServo` to be `PIN 14` and `Center` to be 750. So the command `PULSOUT PingServo, LimitRight` should direct the rangefinder 0° to the right, and `PULSOUT PingServo, LimitLeft` direct the rangefinder 180°. Also, `PULSOUT PingServo, Center` should point the rangefinder straight ahead.

If the `LimitLeft` and `LimitRight` constants do not make the servo point the Ping))) rangefinder to 0° and 180°, the program's `CON` directive values should be adjusted. For `LimitRight`, values smaller than 250 will cause the servo to rotate further in the clockwise direction (to the right) while values larger than 250 will cause it to rotate to a position closer to center. Likewise, values greater than 1250 will cause the servo to rotate further in the counterclockwise direction (to the left) while values smaller than 1250 will cause it to rotate to a position closer to center.

- √ Run `TestPingDirection.bs2` and make sure the servo points the Ping))) rangefinder to 0°, then to 180°, then to 90° (straight ahead).
- √ If the servo doesn't point to 0°, adjust the `LimitRight` constant accordingly. Values less than 250 will result in more clockwise rotation and values greater than 250 will result in less clockwise rotation.

- √ If the servo doesn't point to 180° to the left, adjust the `LimitLeft` constant accordingly. Values greater than 1250 will result in more counterclockwise rotation, and values less than 1250 will result in less counterclockwise rotation.
- √ Update these `CON` directives, and make notes of the values you used. These same `CON` directives will have to be updated in this activity's `Ping)))Dar.bs2` program and also in the next activity's `GotoClosestObject.bs2` program.

### Example Program – TestPingDirection.bs2

```
' Smart Sensors and Applications - TestPingDirection.bs2
' Point the Ping))) 0-degrees (to the right), then 180-degrees (to the right),
' then straight ahead.

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

PingServo      PIN      14                ' Servo directs Ping)))
LimitLeft      CON      1250              ' Ping servo 90-degrees left
LimitRight     CON      250               ' Ping servo 90-degrees right

Center         CON      750               ' Center/0-degree pulse duration
BtwnPulses     CON      20               ' ms between servo pulses

counter        VAR      Byte              ' Loop index

DEBUG "Look to 0-degrees (right).", CR
FOR counter = 1 TO 100                    ' 100 pulses 0-degrees (right)
  PULSOUT 14, LimitRight
  PAUSE 20
NEXT

DEBUG "Look to 180-degrees (left).", CR
FOR counter = 1 TO 100                    ' 100 pulses 180-degrees (left)
  PULSOUT 14, LimitLeft
  PAUSE 20
NEXT

DEBUG "Look to 90-degrees (straight ahead).", CR
FOR counter = 1 TO 100                    ' 100 pulses 90-degrees (ahead)
  PULSOUT 14, Center
  PAUSE 20
NEXT

END
```

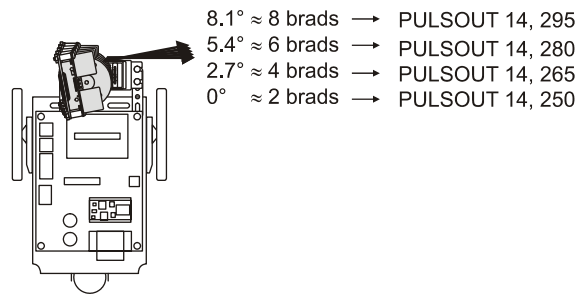
### Testing Ping)))Dar

One of the most important things about Ping)))Dar.bs2 is that it will help give you a better idea of what the Boe-Bot "sees" as it sweeps the Ping))) rangefinder from left to right. Make sure to test a variety of objects at a variety of distances. Keep in mind also that the program is imposing a 100 cm distance limit on measurements to make the display more convenient.

Before running the program, the `LimitRight` and `LimitLeft` `CON` directives should be updated so that the servo that directs the Ping))) rangefinder to sweep from 0° to 180° using the values you determined with `TestPingDirection.bs2`.

Ping)))Dar converts the `PULSOUT` command's *Duration* arguments into binary radian servo rotation angle measurements with the help of a constant and the `**` operator. Let's assume that `LimitRight` is 250 and `LimitLeft` is 1250. The main routine in `Ping)))Dar.bs2` has code that sweeps the `PULSOUT` command's *Duration* argument from 250 to 1250 in increments of 15. As shown in Figure 8-14, that's about 2.7° for each increment of 15, which is pretty close to 2 brads (2.8125°).

**Figure 8-14:** Rotation Angle vs. PULSOUT Duration



Mapping the `PULSOUT` command's duration argument to the number of brads the servo has rotated involves the `**` operator. In the next example program, the command that does the job is `angle = pingDir - LimitRight ** PingDirToAngle`. The `pingDir` variable is used in the command `PULSOUT PingServo, PingDir` to point the Ping))) rangefinder to 0 brads (0°), then 2 brads, then 4 brads and so on all the way to 128 brads (180°). As it does this, the `pingDir` variable starts at `LimitRight` (250), then 265, then 280, and so on up through `LimitLeft` (1250).

Since the expression `angle = pingDir - LimitRight ** PingDirToAngle` subtracts `LimitRight` from `pingDir` before the `**` operation, the value `**` scales will range from 0 through 1000 instead of 250 to 1250. The `PingDirToAngle` constant has to be the right number of 65535ths to scale this range of 0 to 1000 into 0 through 128 brads. This is another job for the `**` scale constant equation introduced in Chapter 3, Activity #3.

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

Here is an example of calculating `PingDirToAngle` constant when `LimitRight` is 250 and `LimitLeft` is 1250. Make sure to follow along because you will need to recalculate this constant based on the `LimitLeft` and `LimitRight` constants you determined for your mounting bracket system with `TestPingDirection.bs2`.

The number of output scale elements is 0 through 128, which is 129 elements since the range includes zero. Next, `LimitLeft - LimitRight = 1250 - 250 = 1000`. Again, we are talking about a range that's inclusive of 0, so there are really 1001 elements in the input scale.

$$\text{output scale elements} = 129$$

$$\begin{aligned} \text{input scale elements} &= \text{RightLimit} - \text{LeftLimit} + 1 \\ &= 1250 - 250 + 1 \\ &= 1001 \end{aligned}$$

Substituting 129 and 1001 into the `**` scale constant equation yields:

$$\begin{aligned} \text{PingDirToAngle} &= \text{Int} \left[ 65536 \left( \frac{\text{output scale elements}}{\text{input scale elements} - 1} \right) \right] \\ \text{PingDirToAngle} &= \text{Int} \left[ 65536 \left( \frac{129}{1001 - 1} \right) \right] \\ &= \text{Int}[8454.14] \\ &= 8454 \end{aligned}$$



#### Why is 1 subtracted from the denominator?

The `*/` operators and `**` operator always round down to the next integer. Subtracting 1 from the denominator, corrects these rounding errors and, on the average, makes `value ** ScaleConstant` sort the input scale elements to their most correct output scale elements. To learn more about how this works, see Chapter 3, Activity #3.

### Example Program – Ping)))Dar.bs2

Some of the checklist instructions below involve updating `CON` directives that you determined with `TestPingDirection.bs2`. While they are not necessarily crucial to this program, they will be crucial for `GotoClosestObject.bs2`, the Boe-Bot Ping))) Mounting Bracket navigation program in the next activity.

- √ Open `Ping)))Dar.bs2` with your BASIC Stamp Editor.
- √ Update the `LimitRight` and `LimitLeft` `CON` directives in the program to tune the 0° right and 180° left pulse widths.
- √ If you changed either `LimitLeft` or `LimitRight`, calculate the value needed for the `PingDirToAngle` `CON` directive with the `PingDirToAngle` equation just discussed. Make sure to update this value in your program too.
- √ Make notes of all these `CON` directives because you will need to substitute them into the next activity's `GotoClosestObject.bs2` example program.
- √ Run the program and maximize your Debug Terminal so that it takes up the entire computer screen.
- √ observe the Debug Terminal as you try placing a variety of objects at various distances from the Boe-Bot at various locations in its 180° field of detection.
- √ Make sure to try the suggestions in the section after the example program entitled *Your Turn - Understanding what the Boe-Bot Does and Does not "See"*. It brings to light some of this setup's capabilities and drawbacks, and they turn out to be important for writing navigation programs.

```
' -----[ Title ]-----
' Smart Sensors and Applications - Ping)))Dar.bs2
' Display radar style object distance measurements in the Debug Terminal.
' as the Boe-Bot sweeps the Ping))) Mounting Bracket servo from right
' (0-degrees) to left (180-degrees).

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

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

```

PingServo    PIN    14                ' Servo that directs Ping)))
Ping         PIN    15                ' Ping))) sensor signal pin

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

LimitLeft    CON    1250              ' Bracket 90-degrees LimitLeft
LimitRight   CON    250               ' Bracket 90-degrees LimitRight
PingDirToAngle CON    8454           ' Servo pulse -> angle with **

CmConstant   CON    2260              ' Echo time -> cm with **
SinCosTo256  CON    517               ' For */ -127..127 -> -256..256
Increment    CON    15                ' Servo PULSOUT increment value
Negative     CON    1                 ' Negative sign
Positive     CON    0                 ' Positive sign

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

pingDir      VAR    Word              ' Pulse duration -> direction
time        VAR    Word              ' Ping))) echo time
distance     VAR    Time              ' Object distance
x           VAR    Word              ' x Debug cursor coordinate
y           VAR    Word              ' y Debug cursor coordinate
angle       VAR    Byte              ' Angle from LimitRight in brads
counter     VAR    angle             ' Loop counter
sweepInc    VAR    Nib               ' Sweep increment
sweepDir    VAR    Bit               ' Increment/decrement pingDir
xSign       VAR    Bit               ' x variable sign
ySign       VAR    xSign             ' Stores sign of y variable
nsign       VAR    Bit               ' Numerator sign
dsign       VAR    Bit               ' Denominator sign

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

pingDir = LimitRight                ' Start servo at 0-degrees

FOR counter = 1 TO 40                ' Initialize servo position
  PULSOUT PingServo, pingDir
  PAUSE 20
NEXT

sweepInc = Increment                ' Set the sweep increment

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

DO

  IF pingDir <= LimitRight THEN      ' Refresh display if far right
    DEBUG CLS, CRSRXY, 50, 25, "X"
  ENDIF

```

```

GOSUB Sweep_Increment          ' Move servo by sweepInc

' Calculate angle from far right in brads.
angle = pingDir - LimitRight ** PingDirToAngle

GOSUB Get_Ping_Cm              ' Get cm measurement
distance = distance MAX 100 / 4 ' Scale for Debug Terminal

GOSUB Polar_To_Cartesian       ' distnace @ angle -> (x, y)

x = x * 2                      ' 2 spaces for every 1 CR

DEBUG CRSRXY, 50 + x, 25 - y, "" ' Display (x, y) coordinate

LOOP

' -----[ Subroutine - Get Ping Cm ]-----
' Gets Ping))) rangefinder measurement and converts time to centimeters.
' Distance may be declared as time to save variable space.

Get Ping Cm:

PULSOUT Ping, 5
PULSIN Ping, 1, time
distance = time ** CmConstant

RETURN

' -----[ Subroutine - Polar_To_Cartesian ]-----
' Calculates x and y (Cartesian coordinates) given distance and angle
' (polar coordinates).

Polar To Cartesian:

' Calculate x coordinate.
x = COS angle                    ' Polar to Cartesian
xSign = x.BIT15                 ' Store sign bit
x = ABS(x) */ SinCOsTo256      ' Polar to Cartesian continued
x = distance */ x
IF xSign = negative THEN x = -x ' Correct sign with sign bit

' Calculate y coordinate.
y = SIN angle                    ' Polar to Cartesian
ySign = y.BIT15                 ' Store sign bit
y = ABS(y) */ SinCOsTo256      ' Polar to Cartesian continued
y = distance */ y
IF ySign = negative THEN y = -y ' Correct sign with sign bit

RETURN

```

```

' -----[ Subroutine - Sweep Increment ]-----
' Increment/decrement the position of the servo that directs the Ping)))
' rangefinder. When pingDir goes outside either LimitRight or LimitLeft,
' the sweep direction toggles.

Sweep Increment:

' Change sweepDir for adding/subtracting increment if at rotation limit.
IF pingDir <= LimitRight THEN
  sweepDir = Positive
ELSEIF pingDir >= LimitLeft THEN
  sweepDir = Negative
ENDIF

' Add/subtract increment to/from pingDir.
IF sweepDir = negative THEN
  pingDir = pingDir - sweepInc
ELSEIF sweepDir = Positive THEN
  pingDir = pingDir + sweepInc
ENDIF

' Send positioning pulse to Ping))) Mounting Bracket servo.
PULSOUT PingServo, pingDir

RETURN

```

### Your Turn - Understanding what the Boe-Bot does and does not "See"

To understand what your Boe-Bot does and does not "see" with the Ping))) rangefinder, a few experiments are in order. Here are some questions that can be answered with Ping)))Dar.bs2 and various objects placed in the Boe-Bot's field of detection.

- √ If you place one object behind another object, can it see the object in back?
- √ How far do you have to rotate a flat object before it is no longer visible to Ping)))?
- √ Start with one or two cylindrical objects about 3 ft (91 cm) apart and 2 ft (61 cm) from the front of the Boe-Bot.



**For best results,** use tall cylindrical objects, such as soda cans, water bottles, etc. Sheets of paper can also be conveniently rolled into 8.5 inch (21.5 cm) tall by 2 to 3 inch (5 to 7.5 cm) diameter cylinders with a couple pieces of tape.

- √ As you move the objects closer to each other, how close can they be to each other before they appear to be one object in the Debug Terminal?
- √ If you keep the objects the same distance from each other but move them closer to the front of the Boe-Bot, is the gap between them detected again at some point?
- √ To what extent does setting the Increment constant is set to a smaller value help the Boe-Bot detect the gap between objects?

### How Ping)))Dar.bs2 Works

The initialization routine applies forty pulses to make sure the servo turns and starts 0° (far-right). In this program, the **FOR...NEXT** loop is only used once. In `GoToClosestObject.bs2` in the next activity, the loop is used frequently to point the Ping))) rangefinder in various directions, so in that program, the loop is placed in a subroutine called `Point_At_PingDir`. One other initialization detail is setting the `sweepInc` variable equal to the `Increment` constant (15). This value is used by the `Sweep_Increment` subroutine to rotate the Ping))) Mounting Bracket servo slightly between each distance measurement.

```
pingDir = LimitRight           ' Start servo at 0-degrees
FOR counter = 1 TO 40         ' Initialize servo position
  PULSOUT PingServo, pingDir
  PAUSE 20
NEXT
sweepInc = Increment         ' Set the sweep increment
```

The main routine's **DO...LOOP** refreshes the Debug Terminal display after each right-left sweep. As the main routine's **DO...LOOP** repeats, the first thing the program does is clear the Debug Terminal and place the "x" character at 50 spaces over and 25 carriage returns down. This "x" indicates the Boe-Bot's position.

```
IF pingDir <= LimitRight THEN ' Refresh display if far right
  DEBUG CLS, CRSRXY, 50, 25, "x"
ENDIF
```

Next, the program calls the `Sweep_Increment` subroutine, which adjusts the Ping))) Mounting Bracket's servo slightly each time it gets called. Calling this subroutine repeatedly each time through the main routine's **DO...LOOP** results in the back and forth sweeping motion. This subroutine uses the `pingDir` variable to direct the servo.

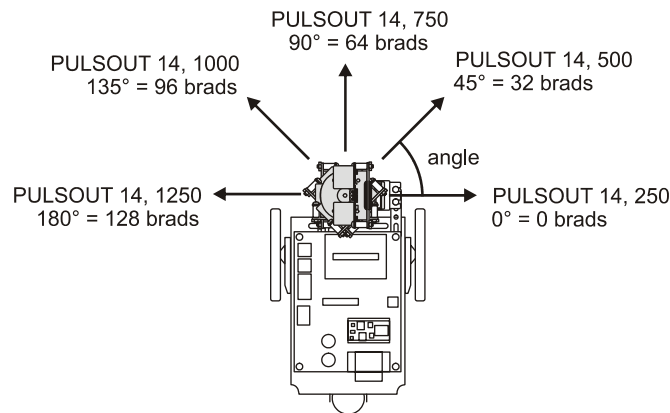
```
GOSUB Sweep_Increment
```

```
' Move servo by sweepInc
```

The `Sweep_Increment` subroutine modified the value of `pingDir`, and then used it in a `PULSOUT` command's `Duration` argument to point the Ping))) Mounting Bracket servo in a new direction. That means we can use the value of `pingDir` to determine the angle to which the Ping))) rangefinder is pointing in terms of binary radians.

Figure 8-15 shows a few examples of how the `PULSOUT` command's `Duration` argument relates to the Ping))) Mounting Bracket servo's rotational position. For example, when `pingDir` is `LimitRight`, (250), the angle is  $0^\circ$ , which is 0 binary radians (brads). When `pingDir` is 500, the angle is about  $45^\circ$ , or 32 brads. Likewise, `pingDir` can be 750 or 1000, and the respective servo angles will be  $90^\circ$  (64 brads) and  $135^\circ$  (96 brads). By the time `pingDir` gets to `LimitLeft` (1250), the angle will be  $180^\circ = 128$  brads.

**Figure 8-15:** PULSOUT Duration Argument vs. Rotation Angle



Here is the command that converts `pingDir`, which ranges from `LimitRight` to `LimitLeft` (250 to 1250) to angle, which ranges from 0 to 128 brads. Note that this is the same kind of equation we used in Chapter 3, Activity #3 and #4.

```
' Calculate angle from far right in brads.
angle = pingDir - LimitRight ** PingDirToAngle
```

Without parentheses, PBASIC executes operators from left to right. So the command `angle = pingDir - LimitRight ** PingDirToAngle` first subtracts `LimitRight` from `pingDir`. This resulting value, which could be anywhere from 0 through 1000, has

the **\*\* PingDirToAngle** operation performed on it. Since **PingDirToAngle** is 8454 in the example program, it means a value from 0 through 1000 gets multiplied by  $8454/65536$ . If **pingDir** stores 1000, the result is 128.9978, and the **\*\*** operator rounds it down to 128, which is the correct number of brads assuming the servo is in fact pointing left (180° from far right). Likewise,  $(750 - 250) \times 8454/65536$  results is 64.4989, which rounds down to 64 brads, which is 90°, and the servo should be pointing straight ahead.

Next, the distance of the object at the angle the Ping))) Mounting Bracket servo is pointing to gets measured and scaled to a centimeter measurement. This centimeter measurement is further scaled down by 1/4 of its actual value to make it fit more conveniently in a Debug Terminal maximized in a typical 1024 by 768 pixel monitor.

```
GOSUB Get_Ping_Cm           ' Get cm measurement
distance = distance MAX 100 / 4   ' Scale for Debug Terminal
```

Now, we know the object's distance and angle relative to the Ping))) rangefinder on the front of the Boe-Bot. The problem is that these values are polar coordinates (distance  $\angle$  angle), but the program needs to express them in terms of Cartesian coordinates (x, y) for Debug Terminal display. The Cartesian coordinates, are necessary for displaying the asterisks that denote the object's position in the Debug Terminal with the **DEBUG** command's **CRSRXY** formatter. So, the program calls the **Polar\_To\_Cartesian** subroutine. Given distance and angle, the subroutine calculates the corresponding x and y coordinates.

```
GOSUB Polar_To_Cartesian      ' distnace @ angle -> (x, y)
```

Before repeating the main routine's **DO...LOOP**, the last step is to plot an asterisk at the (x, y) coordinate. **x** will be a value that ranges from about -50 to 50. So, the x-coordinate in **DEBUG CSRXY** is **50 + x**. The y-coordinate will range from 0 to 25. However, **CRSRXY** will plot the value in terms of carriage returns down from the top line of the Debug Terminal. What we really want is carriage returns upward from the "**x**" that was plotted at 50 spaces over and 25 carriage returns down. That's why the **CRSRXY** formatter's y-coordinate is **25 - y**. As **y** gets larger, the asterisk is plotted closer to the X at (50, 25). As **y** gets smaller, the asterisk is plotted closer to the top of the Debug Terminal.

```
' Display asterisk at x,y coordinate.
DEBUG CSRXY, 50 + x, 25 - y, "**"
```

### Advanced Topic – Inside the Polar\_To\_Cartesian Subroutine

Given a distance at an angle ( $d \angle \theta$ ), you can calculate the x and y coordinates (x, y) with the help of a calculator and these two formulas:

$$x = d \cos(\theta) \quad \text{and} \quad y = d \sin(\theta)$$

Calculating x and y with the BASIC Stamp involves the **SIN**, **COS**, and **\*/** operators.

- √ Look up and review the **\*/**, **SIN** and **COS** operators in either the BASIC Stamp Manual or the BASIC Stamp Editor's Help feature.

Given an angle in brads, PBASIC's **COS** operator returns 127 to 0 to -127 to 0 as the brad angle goes from 0 to 64 to 128 to 192. Calculating the cosine from 0° to 90° to 180° to 270° with a calculator will yield results that range from 1 to 0 to -1 and back to 0 again. The **SIN** operator behaves similarly, returning values that range from 0 to 127 to 0 to -127 as the angle in brads goes from 0 to 64 to 128 to 192. Again, if you calculate the sine of angles from 0° to 90° to 180° to 270° with a calculator, you will see that the actual sine values range from 0 to 1 to 0 to -1. The problem here is that we want to multiply distance results for **SIN** and **COS** values that range from -1 to 1, not -127 to 127.

One way around this is creative use of the **\*/** operator. This operator multiplies a value by a number of 256ths. By using the **\*/** operator twice, we can first scale a value in the range of -127 through 127 up to a range from -256 through 256. Then, we can use **\*/** again to multiply that result by the distance. The second time, the **\*/** operator will be multiplying the distance variable by sine and cosine values that are in terms of a number of 256ths. The result will be very close to what you would get from a calculator, multiplying the distance sine and cosine value that range from -1 through 1.

To scale a value that will fall in the range of -127 through 127 to its equivalent in the range from -256 to 256, we'll use the **\*/** scale constant equation introduced in Chapter 3, Activity #5.

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

Since the input range is -127 through 127, that's 255 possible values. The output range is going to be -256 to 256, which is 513 possible values.

**output scale elements = 513**

**input scale elements = 255**

To calculate a constant we'll call `sinCosTo256`, we'll substitute 255 and 513 into the \*/ scale constant equation:

$$\begin{aligned} \text{SinCosTo256} &= \text{Int}\left[256\left(\frac{513}{255-1}\right)\right] \\ &= \text{Int}[517.04] \\ &= 517 \end{aligned}$$

Now that we know `sinCosTo256` has to be 517, here is the routine for polar to Cartesian coordinate conversion. Notice that the sign of both the cosine and sine calculations are stored in the `xSign` and `ySign` bits, and the rest of the calculations are done with the absolute values of `x` and `y`. That's because `/`, `*/`, `//`, and `**` were designed to be used with positive integers only. So, the absolute value of each measurement is taken first. Then the rest of the operations are done on the positive integer values, and the sign of the `x` and `y` results are restored at the end by `IF . . . THEN` statements.

```
' Calculate x coordinate.
x = COS angle
xSign = x.BIT15
x = ABS(x) */ SinCosTo256
x = distance */ x
IF xSign = negative THEN x = -x

' Calculate y coordinate.
y = SIN angle
ySign = y.BIT15
y = ABS(y) */ SinCosTo256
y = distance */ y
IF ySign = negative THEN y = -y
```

' Polar to Cartesian  
' Store sign bit  
' Polar to Cartesian continued  
' Correct sign with sign bit

' Polar to Cartesian  
' Store sign bit  
' Polar to Cartesian continued  
' Correct sign with sign bit