Chapter 6: Light-Sensitive Navigation with Phototransistors (Supplementary Activity #7..9)

The example programs from Robotics with the Boe-Bot v3.0 in Chapter 6, Activity #4 through #6 depended on calls to the **`Light_Shade_Info`** subroutine for information about ambient light level and differential light/shade measurement. Example programs from the chapter that utilize **`Light_Shade_Info`** and its nested subroutines include:

- LightSensorValues.bs2
- LightSensorDisplay.bs2
- LightSeekingDisplay.bs2
- LightSeekingDisplayBetterRAM.bs2
- LightSeekingBoeBot.bs2
- Some of the project solution example files.

LightSensorValues.bs2 is printed in Robotics with the Boe-Bot v3.0, Chapter 6, Activity #4. Robotics with the Boe-Bot is available for free download from www.parallax.com/go/BoeBot. The rest of the programs referenced in Activity #4 through #6, are in LightSensorExamples.zip, which is also available from www.parallax.com/go/BoeBot.

- ✓ Before continuing here, make sure to complete all of Chapter 6 in Robotics with the Boe-Bot v3.0.

In review, a call to Light_Shade_Info:

- Loads a value into the **`light`** variable that indicates the total amount of light both sensors detect so that a program can decide if one area is darker than another as it roams.
- Loads a value into the **`ndShade`** variable that supplies a normalized, differential light/shade measurement on a scale of –500 to 500. A program can use this value to determine if the Boe-Bot detects brighter or dimmer light on one side. The Boe-Bot's program can in turn use this information for roaming toward or away from light sources.

- Makes incremental adjustment to the **duty** variable for automatic light sensitivity adjustment. By making calls between each servo pulse, the Boe-Bot's program can rapidly adjust to lighter or darker rooms.

As you may have noticed a single call to **Light_Shade_Info** results in a lot of code execution. All tolled, there are 41 lines of executable code in **Light_Shade_Info** its three nested subroutines. The four jobs **Light_Shade_Info** and its helpers perform are:

1) Measure decay times of both photoresistor QT circuits using variable **PWM *Duty*** to charge the circuits' capacitors to starting voltages that can be used to help keep the sum of the decay times in a certain range.
2) Use the **duty** variable that controls **PWM** in step 1 to calculate a value that corresponds to the ambient light levels in the room. If **duty** is at one of the ends of its range, use the sum of the two sensor decay time measurements for extended values.
3) Check to see if the sum of the left and right QT decay measurements are in the desired range. If not, adjust the **duty** variable for the next iteration.
4) Normalize the differential light measurement so that it fits in a scale of -500 to 500.

Optional Activity #7, #8, and #9 chronicle some of the development and tests that went into **Light_Shade_Info** and its nested subroutines. These optional activities also examine how each subroutine contributes to completing the four jobs listed above. Activity #7 takes a closer look at normalized, differential measurements from the standpoint of why it's useful and how the code accomplishes it. Activity #8 examines how automatic light sensitivity works when **Light_Shade_Info** is called repeatedly, and Activity #9 explains how **Light_Shade_Info** extracts a number that indicates overall light level.

Light_Shade_Info Subroutine Overview

The **Light_Shade_Info** subroutine and the subroutines it calls are listed below. They are a modified excerpt from LightSensorValues.bs2. Comments that explain the 5 major steps the code completes have been added. Below each `' Step...` comment is the code that completes the task.

- ✓ Read the comments that start with "Step 1", "Step 2", up through "Step 5" in the **Light_Shade_Info** subroutine below.

- ✓ Optional, follow each subroutine call, examine the code, and try to understand what it's doing. Make a list of questions about portions that do not immediately make sense. Hopefully, they will all be answered by the time you make it through this document. Otherwise, contact the author through forums.parallax.com. His username is Andy Lindsay (Parallax).

```
'-----[ Subroutine - Light_Shade_Info ]-------------------------------------
' Uses tLeft and tRight (RCTIME measurements) and pwm var to calculate:
'   o light   - Ambient light level on a scale of 0 to 324
'   o ndShade - Normalized differential shade on a scale of -500 to + 500
'               (-500 -> dark shade over left, 0 -> equal shade,
'                +500 -> dark shade over right)

Light_Shade_Info:                           ' Subroutine label

  ' Step 1:
  ' Call the Light_Sensors Subroutine.  This subroutine uses a variable
  ' named duty to set the initial voltage it applies to the phototransistor
  ' QT circuit capacitors before measuring the decay.  The decay is controlled
  ' by incident light.
  GOSUB Light_Sensors                       ' Get raw RC light measurements

  ' Step 2:
  ' Calculate light variable (ambient light level) using duty variable.
  ' If duty is below its minimum or above its maximum, use tLeft and tRight
  ' for more info.
  sumdiff = (tLeft + tRight) MAX 65535      ' Start light level with sum
  IF duty <= 70 THEN                        ' If duty at min
    light=duty-(sumdiff/905) MIN 1          ' Find how much darker
    IF sumdiff = 0 THEN light = 0           ' If timeout, max darkness
  ELSEIF duty = 255 THEN                    ' If duty at max
    light=duty+((1800-(sumdiff))/26)        ' Find how much brighter
  ELSE                                      ' If duty in range
    light = duty                            ' light = duty
  ENDIF                                     ' Done with light level

  ' Step 3:
  ' Call the Duty_Auto_Adjust subroutine.  This subroutine checks to find out
  ' if tLeft + tRight add up to something in the 1800 to 2200 range.  If it's
  ' in range, the subroutine takes take no action.  If not, the subroutine
  ' increases or decrease the duty variable to make the Light_Sensors
  ' subroutine return larger or smaller light measurements the next time it
  ' gets called (next time through the Main Routine's DO...LOOP).
  GOSUB Duty_Auto_Adjust                    ' Adjust PWM duty for next loop
```

```
  ' Step 4:
  ' Divide tLeft into tLeft + tRight.  Start by copying tLeft to a variable
  ' named n (for numerator) and tLeft + tRight into a variable named d (for
  ' denominator).  Then, call Fraction_Thousandths, a subroutine that performs
  ' long division with a result of q = n ÷ d.  The subroutine stores the four
  ' digit result in a variable named q.  This result can range from 0 to 1000
  ' and is the fractional portion of the actual quotient in terms of
  ' thousandths.
  n = tLeft                                 ' Set up tLeft/(tLeft+tRight)
  d = tLeft + tRight
  GOSUB Fraction_Thousandths                ' Divide (returns thousandths)

  ' Step 5:
  ' Copy 500 – q to the ndShade variable.  This makes the result range from
  ' -500 to 500 with 0 indicating equal light levels measured by both
  ' QT sensors.
  ndShade = 500-q                           ' Normalized differential shade

  RETURN                                    ' Return from subroutine

'-----[ Subroutine - Light_Sensors ]-----------------------------------------
' Measure P6 and P3 light sensor circuits.  Duty variable must be in 70...255.
' Stores results in tLeft and tRight.

Light_Sensors:                              ' Subroutine label
  PWM 6, duty, 1                            ' Charge cap in P6 circuit
  RCTIME 6, 1, tLeft                        ' Measure P6 decay
  PWM 3, duty, 1                            ' Charge cap in P3 circuit
  RCTIME 3, 1, tRight                       ' Measure decay
  RETURN                                    ' Return from subroutine

'-----[ Subroutine - Duty_Auto_Adjust ]--------------------------------------
' Adjust duty variable to keep tLeft + tRight in the 1800 to 2200 range.
' Requires sumdiff word variable for calculations.

Duty_Auto_Adjust:                           ' Subroutine label
  sumDiff = (tLeft + tRight) MAX 4000       ' Limit max ambient value
  IF sumDiff = 0 THEN sumDiff = 4000        ' If 0 (timeout) then 4000
  IF (sumDiff<=1800) OR (sumDiff>=2200) THEN ' If outside 1800 to 2200
    sumDiff = 2000 - sumDiff                ' Find excursion from target val
    sign = sumDiff.BIT15                    ' Pos/neg if .BIT15 = 0/1
    sumDiff = ABS(sumDiff) / 200            ' Max sumDiff will be +/- 10
    sumDiff = sumDiff MAX ((duty-68)/2)     ' Reduce adjustment increments
    sumDiff = sumDiff MAX ((257-duty)/2)    '  near ends of the range
    IF sign=NEGATIVE THEN sumDiff=-sumDiff  ' Restore sign
    duty = duty + sumDiff MIN 70 MAX 255    ' Limit duty to 70 to 255
  ENDIF                                     ' End of if outside 1800 to 2200
  RETURN                                    ' Return from subroutine
```

```
'-----[ Subroutine - Fraction_Thousandths ]-------------------------------------
' Calculate q = n/d as a number of thousandths.
' n and d should be unsigned and n < d.  Requires Nib size temp & i variables.

Fraction_Thousandths:                          ' Subroutine label
  q = 0                                        ' Clear quotient
  IF n > 6500 THEN                             ' If n > 6500
    temp = n / 6500                            '   scale n into 0..6500
    n = n / temp
    d = d / temp                               '   scale d with n
  ENDIF
  FOR i = 0 TO 3                               ' Long division ten thousandths
    n = n // d * 10                            ' Multiply remainder by 10
    q = q * 10 + (n/d)                         ' Add next digit to quotient
  NEXT
  IF q//10>=5 THEN q=q/10+1 ELSE q=q/10        ' Round q to nearest thousandth
  RETURN                                       ' Return from subroutine
```


ACTIVITY #7: NORMALIZED LIGHT/SHADE MEASUREMENTS

The overall brightness or darkness in a room is called the ambient light level. Ambient light can have a huge effect on phototransistor light measurements, which can in turn make programming the Boe-Bot to recognize shadows and light sources a tricky job. For example, if you tried to test for some shade and light values in one room, they might not work in another room that's brighter or darker. That's because a brighter room will cause the light measurements to be much smaller, and a darker room will cause the light measurements to all be much larger. So without some extra processing, the Boe-Bot's program might not detect any shadows at all in the brighter room, and it might think all its measurements in the darker room are shadows.

As you've already seen in Activity #3, darker rooms pose a another problem because they can make the light measurements take so long that they cause delays in servo pulses. This can in turn cause jittery or halting servo control. Activity #8 examines the **`Duty_Auto_Adjust`** subroutine that automatically adjusts the **`PWM`** command's ***`Duty`*** argument for better performance in the ambient light levels it senses.

Fortunately, the solution to the "tricky" problem of quantifying differences between left and right light levels is fairly simple, even in different ambient light levels. All your program has to do is calculate the percentage of the total lighting that each light sensor circuit detects. Then, it won't matter if ambient lighting causes the measurements to be large or small because the same shadow condition will still result in roughly the same difference in left/right percent measurements.

When the left and right sensors are expressed as percents of the total light measurement (left + right), it is called a normalized measurement. A normalized measurement is one that has been fit to a more useful scale. For example, the Boe-Bot's right light sensor might be under pretty dark shade compared to its left. With all the lights in the room on, the **tLeft** might be 100 and **tRight** might be 400. With some of the lights turned off, the same shade condition might be measured as **tLeft** = 500 and **tRight** = 2000. In both cases, the left sensor is 20% of the total measurement and the right sensor is 80%. By going through the calculations to express what each sensor sees as a percent of the total, the measurements are being normalized.

Example: percent of total when **tLeft** = 100 and **tRight** = 400:

$$\%left = \frac{tLeft}{tLeft + tRight} \times 100\%$$
$$= \frac{100}{100 + 400} \times 100\%$$
$$= \frac{100}{500} \times 100\%$$
$$= 20\%$$

Example: percent of total when **tLeft** = 500 and **tRight** = 2000:

$$\%left = \frac{tLeft}{tLeft + tRight} \times 100\%$$
$$= \frac{500}{500 + 2000} \times 100\%$$
$$= \frac{500}{2500} \times 100\%$$
$$= 20\%$$

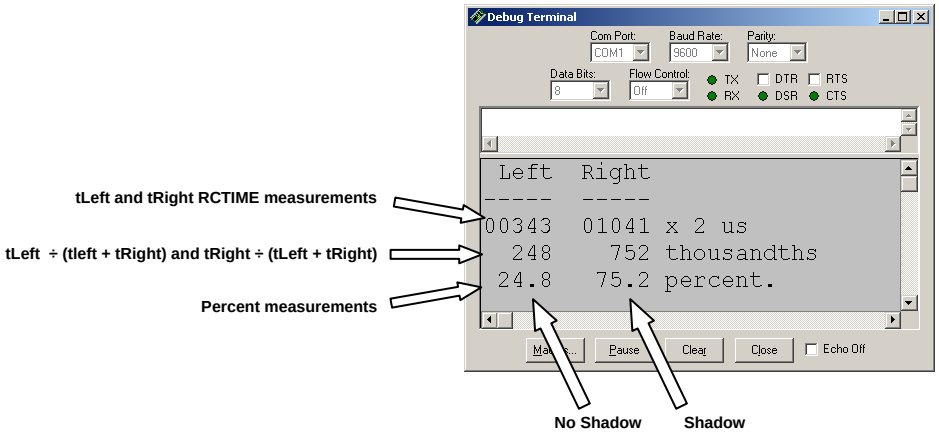
- ✓ Repeat the calculations for **tRight**.
- ✓ Can you just calculate **tLeft** and then subtract from 100% to get **tRight**? Try it.

Example Program: LightPercentDifference.bs2

LightPercentDifference.bs2 displays each decay measurement as a percentage of the total. It does this by dividing **tLeft** into **tLeft + tRight** using a subroutine called **Fraction_Thousandths**. This subroutine is designed to divide a variable named **n** (numerator) into a variable named **d** (denominator) and store the result in a variable named **q** (quotient). The result the subroutine stores in **q** is a value that could be anywhere from 0 to 1000, and it expresses the amount of light **tLeft** detects in tenths of a percent. The **tRight** variable's percent is calculated by simply subtracting **tLeft** from 1000. It could also be calculated by calling **Fraction_Thousandths** again with **tRight** stored in the **n** variable, but that would be less efficient.

- ✓ Either carefully enter and save LightPercentDifference.bs2, or use the BASIC Stamp Editor to open it from the same zip that contains this PDF. The zipped package is available for download from forums.parallax.com -> Forums -> (Education) Stamps in Class -> Stamps in Class "Mini Projects" -> More Light Seeking Activities. Download RwtBBCH6A7to9.zip.
- ✓ Unzip it to a folder. These activities assume you will name the folder RwtBBCH6A7to9.
- ✓ Run LightPercentDifference.bs2; the display should resemble Figure 6-21.
- ✓ Try casting a shadow over the right sensor. Even though your first row of values may be entirely different from the first row in Figure 6-21, you will be able to find a level of shade that results in about a 75% measurement for the right sensor, like the bottom-right measurement in the Debug Terminal, which is 75.2%.

Figure 6-21: Example with about 75% Shade over Right Sensor




```
'-----[ Title ]-----------------------------------------------------------------
' Robotics with the Boe-Bot - LightPercentDifference.bs2
' Normalize tLeft and tRight to a scale of 0 to 1000 and display as percents
' of the sum of both measurements.

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

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

tLeft          VAR     Word                  ' Stores left sensor decay time
```

```
tRight         VAR     Word                  ' Stores right sensor decay time
                                             ' Fraction_Thousandths variables
n              VAR     Word                  '   Numerator
d              VAR     Word                  '   Denominator
q              VAR     Word                  '   Quotient result
i              VAR     Nib                   '   Index
temp           VAR     Word                  '   Temporary value

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

PAUSE 1000                                   ' Wait 1 s before any DEBUG

'-----[ Main Routine ]-----------------------------------------------------
DO                                           ' Main loop
  GOSUB Light_Sensors                        ' Get tLeft and tRight

  DEBUG HOME, " Left  Right", CR,            ' Display times as 2 us units
              "-----  -----", CR,
        DEC5 tleft, "  ", DEC5 tright,
        " x 2 us"

  n = tLeft                                  ' Set up for n/d calculation
  d = tLeft + tRight
  GOSUB Fraction_Thousandths                 ' Divide n into d
  tLeft = q                                  ' Store normalized results in
  tRight = 1000 – q                          '    tLeft and tRight

  DEBUG CR, "  ",                            ' Display thousandths of total
        DEC3 tLeft, "    ", DEC3 tRight,
        " thousandths"
        DEBUG CR, " ",
        DEC2 tLeft/10, ".", DEC1 tLeft,      ' Display percent
        "   ",
        DEC2 tRight/10, ".", DEC1 tRight,
        " percent"

  PAUSE 250                                  ' Wait 0.25 seconds

LOOP                                         ' Repeat main loop

'-----[ Subroutine - Light_Sensors ]----------------------------------------

Light_Sensors:                               ' Check P6 & P3 light sensors
  PWM 6, 184, 1                              ' Charge cap to 3.59 V
  RCTIME 6, 1, tLeft                         ' P6->input, measure decay time
  PWM 3, 184, 1                              ' Charge cap to 3.59 V
  RCTIME 3, 1, tRight                        ' P3->input, measure decay time
  RETURN                                     ' Return from subroutine

'-----[ Subroutine - Fraction_Thousandths ]---------------------------------
```

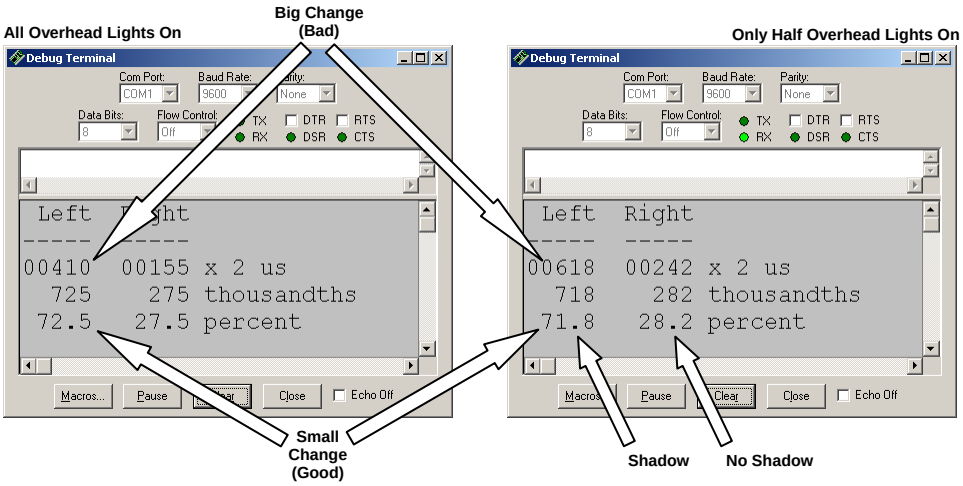
```
' Calculate q = n/d as a number of thousandths.
' n and d should be unsigned and n < d.  Requires a Nib size temp variable.
' All the variables this subroutine uses should be word size, except i and
' temp, which can be nibbles.

Fraction_Thousandths:                          ' Fraction_thousandths subroutine
  q = 0                                        ' Clear quotient
  IF n > 6500 THEN                             ' If n > 6500
    temp = n / 6500                            '   scale n into 0..6500
    n = n / temp
    d = d / temp                               '   scale d with n
  ENDIF
  FOR i = 0 TO 3                               ' Long division ten thousandths
    n = n // d * 10                            ' Multiply remainder by 10
    q = q * 10 + (n/d)                         ' Add next digit to quotient
  NEXT
  IF q//10>=5 THEN q=q/10+1 ELSE q=q/10        ' Round q to nearest thousandth
  RETURN                                       ' Return from subroutine
```


Test the Same Shade in Different Ambient Light Conditions

The Boe-Bot's program needs to determine if each light sensor sees a shadow or light. For this type of sensing, Figure 6-22 shows an example of why it's better to use percents of the total sensor measurements. For both measurements, the same shadow was cast over the Boe-Bot's left light sensor. Then, the ambient light level was changed. These measurements show that for the same shadow, the 2 µs **RCTIME** measurements vary drastically with changes in ambient light, but the percent measurements hardly change at all. Since the shadow condition didn't change, the fact that the percent measurements don't change means that they can give us a much better indicator of how much shade or light one of the sensors detects. Note also that the larger percent measurement means that there's more shade, and the smaller percent measurement indicates less shade (or more light).

Figure 6-22: Two Measurements for the Same Shadow with Different Ambient Light Levels



The test setup for the measurements in Figure 6-22 involved placing the Boe-Bot in the middle of a well lit room and facing it toward a window that let light in from outside (but not direct sunlight). Then, a shade was placed between the left sensor and the window; the right sensor still had a clear view. The measurement on the left side of Figure 6-22, was with all overhead lights in the room on. The measurement on the right side was with half the lights off.

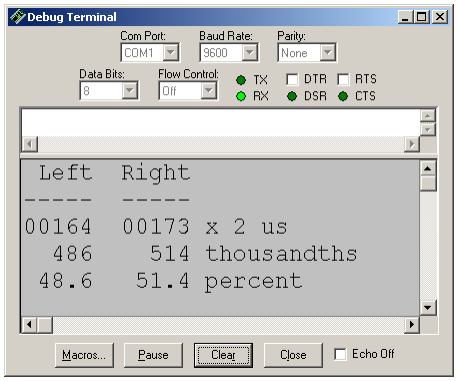
Your Turn: Test Different Shades in Same Ambient Light Conditions

Figure 6-23 shows an example with four different shade conditions, no shade, shade over both sensors, and shade over just the left and just the right sensors. Light following applications in Chapter 6 used the **`tLeft`** value from the thousandths row in Figure 6-23 to calculate the value of the **`ndShade`** variable, which was used to determine which sensor detected darker shade. So it's a good idea to experiment with determining how much shade each sensor detects by watching the thousandths row.

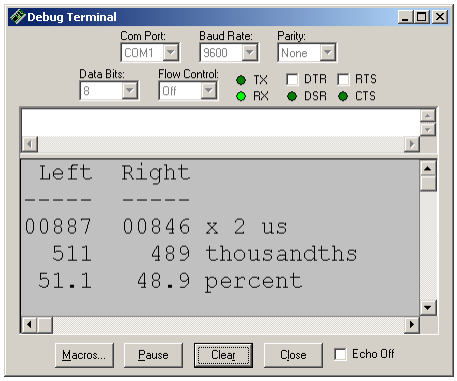
- ✓ Try casting shadows over the right and left sensors, like the example in Figure 6-23.
- ✓ Repeat the same sequence of shadows, but this time make them darker. What changed?
- ✓ Repeat again, this time with shadows that are less dark. Again, what changed.

Figure 6-23: Four Measurements with Different Shade in the Same Ambient Light

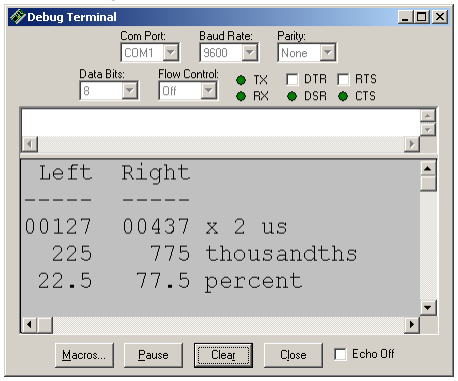
No Shade



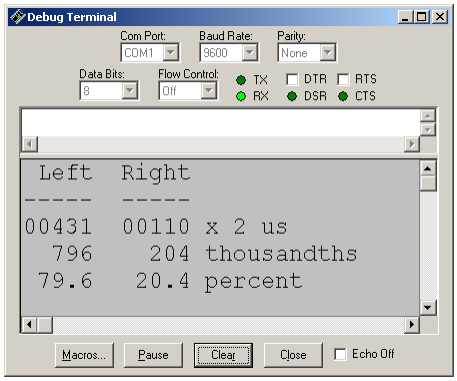
Shade over both



Shade over right



Shade over left



Inside LightPercentDifference.bs2

The Main Routine calls the **`Light_Sensors`** subroutine, and then displays the values of **`tLeft`** and **`tRight`** with a **`DEBUG`** command. Then, it copies **`tLeft`** to the **`n`** variable and **`tLeft + tRight`** to the **`d`** variable. After a call to **`Fraction_Thousandths`**, the result of **`n/d`** is stored in the **`q`** variable. The program copies that value back to **`tLeft`**. Note that **`tRight`** gets calculated with **`1000 - tLeft`**. As mentioned earlier it's a more efficient to subtract **`tLeft`** from 1000 than to repeat the division a second time with **`tRight`** as the numerator. If **`tLeft`** contains a certain number of thousandths of **`(tLeft + tRight)`**, then **`tRight`** has to contain the rest of the thousandths.

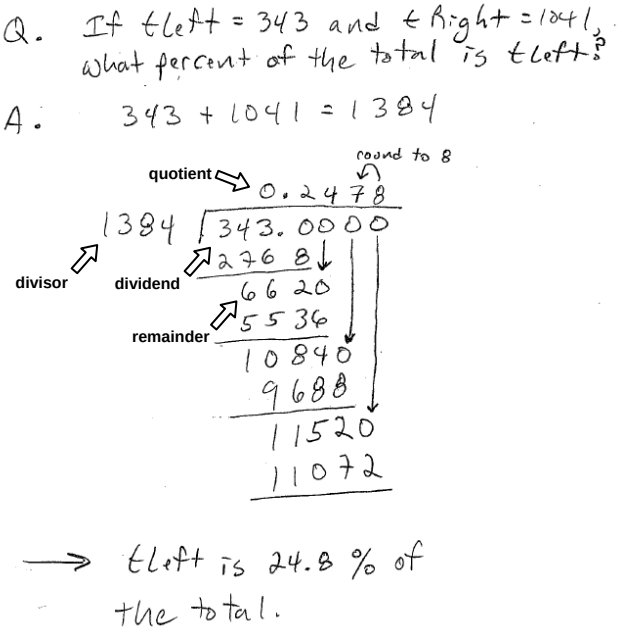
The main routine has a second **DEBUG** command that displays **tLeft** and **tRight** as numbers of thousandths, followed by a display in percents. The **DIG** operator is useful for picking certain digits to print to the left of the decimal point, and another to the right of the decimal point for a display in tenths of a percent.

- ✓ In the BASIC Stamp Editor, click Help and select BASIC Stamp Manual. The document should open into your computer's Adobe Acrobat Reader software.
- ✓ Click Edit and select Search.
- ✓ Type **DIG** into the search field.
- ✓ Since there are a lot of references to digit, refine your search by clicking two checkboxes: Whole words only, and Case sensitive.
- ✓ Click the Search button.
- ✓ The first result of the search should be **DIG** in the table of contents. Use this to find the page with documentation about the **DIG** operator and read it.
- ✓ Make sure to uncheck those PDF search options so that they don't interfere with your next search.
- ✓ Read about **DIG** and use what you learned to decipher how the code by the Display Percent comment in LightPercentDifference.bs2's main routine works.

How the Fraction_Thousandths Subroutine Works

The **Fraction_Thousandths** subroutine divides **n** into **d** and stores a result, which is a number of thousandths, in **q**. This subroutine is designed for calculating the number of thousandths in a fraction where the denominator is larger than the numerator, in other words, **d** > **n**. **Fraction_Thousandths** uses an approach you may have first seen in grade school –calculating the quotient one digit at a time. Figure 6-24 shows an example of a hand calculation for the **tLeft** percentage value from Figure 6-21 on page 7.

Figure 6-24: Example of How the Fraction_Thousandths Subroutine Performs Long Division



In Figure 6-24, a digit in the quotient is calculated by figuring out how many times the divisor fits into the dividend. The result of the divisor multiplied by the quotient digit is then subtracted from the remainder of the previous iteration. This new remainder is then multiplied by 10 and used to find the next quotient digit in the next long division iteration.

Of course, the division problem can also be expressed as a fraction, like in Figure 6-25. This illustrates the names of each of the variables better, because the **`Fraction_Thousandths`** uses a variable named **`n`** to store the numerator, one named **`d`** to store the denominator and **`q`** to store the fractional portion of the quotient result, which again is in terms of thousandths.

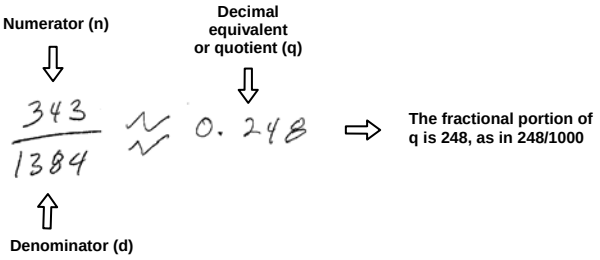


Figure 6-25
Division problem from Figure 6-24 expressed as a fraction

Example Program: LongDivSteps.bs2

LongDivSteps.bs2 uses a slightly expanded version of the **`Fraction_Thousandths`** subroutine that relates the division steps in the **`FOR…NEXT`** loop to the long division steps shown in Figure 6-24. Figure 6-26 shows an example of Debug Terminal output from LongDivsSteps.bs2 with **n** set to 343 and **d** set to 343 + 1041 = 1384.

- ✓ Compare the steps in Figure 6-26 to the steps in Figure 6-24.
- ✓ Find the matching values in the two figures.

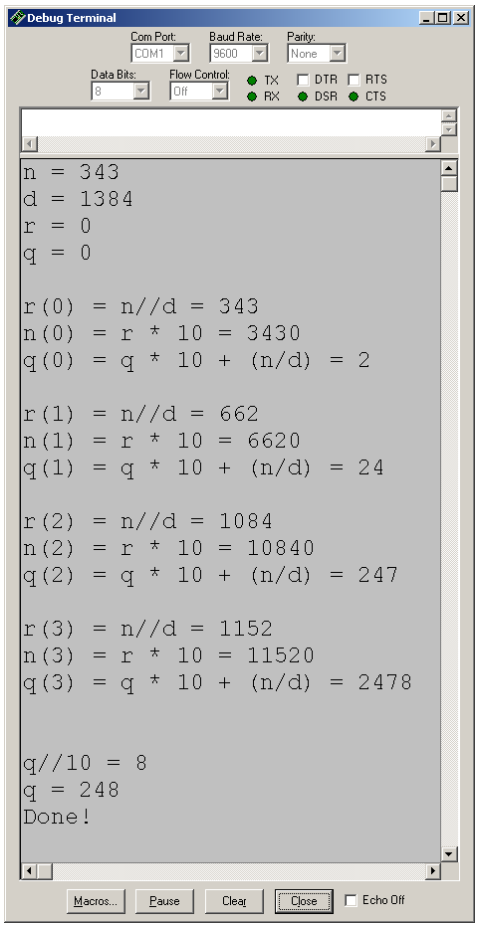


Figure 6-26
LongDivSteps.bs2 Debug Terminal Output

6

- ✓ Examine the code in LongDivSteps.bs2.
- ✓ Run the program as-is and verify the output.
- ✓ Try the long division problem by hand with **`n = 1041`** and **`d = 1041 + 343`**.
- ✓ Change the value of **`n`** from 343 to 1041.
- ✓ Re-run the program and reconcile the output to your long division steps.

```
'-----[ Title ]-------------------------------------------------------------
' Robotics with the Boe-Bot - LongDivSteps.bs2
' Display the main division steps performed by the Fraction_Thousandths
' subroutine.
```

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

n               VAR     Word                  '   Numerator
d               VAR     Word                  '   Denominator
r               VAR     Word                  '   Remainder
q               VAR     Word                  '   Quotient result
temp            VAR     Word                  '   Temporary value storage
i               VAR     Nib                   '   Index

PAUSE 1000                                    ' Wait 1 s before any DEBUG

n = 343                                       ' Set numerator
d = 343 + 1041                                ' Set denominator
r = 0                                         ' Clear remainder
q = 0                                         ' Clear quotient

DEBUG ? n, ? d, ?r, ? q, CR                   ' Display division components

IF n > 6500 THEN                              ' If n > 6500
  temp = n / 6500                             '   scale n into 0..6500
  n = n / temp
  d = d / temp                                '   scale d with n
ENDIF

FOR i = 0 TO 3                                ' Long division ten thousandths
  r = n // d                                  ' Calculate remainder
  DEBUG "r(",DEC i,") = n//d = ",DEC r,CR     ' Display remainder
  n = r * 10                                  ' Multiply remainder by 10
  DEBUG "n(",DEC i,") = r * 10 = ",DEC n,CR   ' Display remainder * 10
  q = q * 10 + (n/d)                          ' Add next digit to quotient
  DEBUG "q(", DEC i, ") = q * 10 + (n/d) = ",' Display next quotient iteration
        DEC q, CR, CR
NEXT                                          ' Repeat long division loop

DEBUG ? q//10                                 ' Display ten-thousandths digit
IF q//10>=5 THEN q=(q/10)+1 ELSE q=q/10       ' Round q to nearest thousandth
DEBUG ? q                                     ' Display result q

DEBUG "Done!"                                 ' Display done
END                                           ' Go into low power mode
```

Notes

Before starting the long division loop, the subroutine makes sure that the numerator **n** is smaller than 6500. If it has to scale **n** down, it also scales **d**. There will be a slight loss of precision if the subroutine scales down **n** and **d** because the BASIC Stamp's integer math

always rounds down. This is not an issue with the example we just tried because the values are well below 6500. It's also not an issue in Chapter 6 because the navigation programs that utilize this subroutine for navigation also limit n to 4000.

Notice that the last step in Figure 6-24 is to check the fourth digit in the result and use it to round the third digit up if the fourth digit is greater than or equal to 5. The code that does this is a simple **`IF…THEN…ELSE`** statement. This code ensures that the result will match values entered into a hand calculator for numerators of 6500 or less. The **`FOR…NEXT`** loop that does the long division calculates the number of ten-thousandths, and uses the fourth digit (number of ten thousandths) to decide whether or not to round the third digit up or down. If the fourth digit is 5 or larger, the third digit gets rounded up. If the fourth digit is instead in the 0 to 4 range, the third digit is left as-is.

ACTIVITY #8: INSIDE AUTOMATIC LIGHT SENSITIVITY ADJUSTMENT

There is still one programming ingredient that has not been covered for making the Boe-Bot seek light effectively. For best results, and especially to prevent halting in darker rooms, the program needs a subroutine that automatically adjusts to different ambient light levels. In other words, the program needs a subroutine that makes the **`PWM`** command charge the capacitors in the light sensing circuits to lower levels in darker rooms and higher levels in brighter rooms.

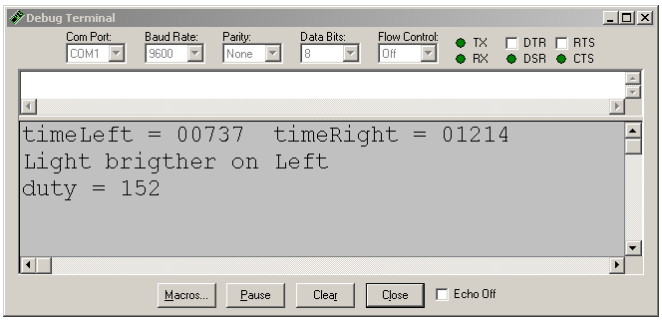
- ✓ Review Chapter 6, Activity #3 before continuing here.

AutoLightSensitivity.bs2 shows an example of how this can be accomplished. Instead of a constant value like 184 or 128, the **`PWM`** command's ***`Duty`*** arguments rely on a variable named **`duty`**. At the beginning of the program, this variable is initialized to 128. After taking and displaying each light measurement, the program calls a subroutine named **`Duty_Auto_Adjust`**. This subroutine checks to find out if the light measurements add up to somewhere in the 1800 to 2200 range. If not, it adjusts the **`duty`** variable so that it will be closer in the next measurement.

Figure 6-27 shows an example of the program's Debug Terminal display after the adjustments have settled down. Notice that the two RC decay measurements (**`tLeft`** and **`tRight`**) add up to 1951, which is in the 1800 to 2200 range. To get the measurements into that range, the program's **`Duty_Auto_Adjust`** subroutine had to adjust the program's **`duty`** variable to 152. The program initialized the **`duty`** variable to 128, and it took several repetitions of the main routine for the program to home in on this value.

Although this is a slow process when the main loop repeats every 0.25 seconds, it will respond to new lighting conditions in less than a second when it's repeating rapidly enough to send servo pulses.

Figure 6-27: `Duty` Variable Automatically Adjusted for Ambient Light



Keep in mind as you test AutoLightSensitivity.bs2 that the ambient light can still be too bright or too dim. If the Debug Terminal shows that the sum of the light measurements are below 1800 and the **`duty`** variable is at 255, it means the light is too bright for the program to compensate enough with **`PWM`**. Likewise, if the measurements are above 2200 and the **`duty`** variable's value is at 70, it means the light is too dark for the program to compensate. In both of those cases, the program compensates as much as it can.

Also keep in mind that the program does not have to keep the values in the 1800 to 2200 range for the sake of determining how much shade or light one sensor sees compared to the other. The **`Fraction_Thousandths`** subroutine from the previous activity takes care of that by normalizing each sensor's measurements to a scale of –500 to 500. The **`Duty_Auto_Adjust`** subroutine in this activity's example program is mainly responsible for reducing the **`duty`** variable for the **`PWM`** command if the room gets darker or increasing it if the light gets brighter. It also improves the servo's behavior it helps keep the delays between control pulses uniform as well.

- ✓ Find AutoLightSensitivity.bs2 in the RwtBBCH6A7to9 folder and open it with the BASIC Stamp editor. (Optionally, you can hand enter it, but be very careful if you do.)
- ✓ Load the program into the BASIC Stamp.
- ✓ Test the program's response to different ambient lighting conditions.

- ✓ Note that it doesn't matter if the light measurements are different because the program is working to keep the sum of the two light measurements in a certain range. So there should still be differences to indicate which light sensor detects brighter light.
- ✓ Also, keep in mind that these are still raw voltage decay time measurements that can be normalized.


```
'-----[ Title ]-------------------------------------------------------------
' Robotics with the Boe-Bot - AutoLightSensitivity.bs2
' Automatically adjusts to keep the sum of tLeft + tright in the
' 1800 to 2200 range.

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

'-----[ Constants/Variables ]-----------------------------------------------

NEGATIVE       CON     1                   ' For use with sign variable

tLeft          VAR     Word                ' Stores left sensor decay time
tRight         VAR     Word                ' Stores right sensor decay time
tDiff          VAR     Word                ' Stores right/left difference
tSum           VAR     Word                ' Sum of left/right times
duty           VAR     Byte                ' Stores duty value
sign           VAR     Bit                 ' 1 -> negative, 0 -> positive

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

PAUSE 1000                                 ' Wait 1 s before any DEBUG
duty = 128                                 ' Initialize duty value

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

DO                                         ' Main loop
  GOSUB Light_Sensors                      ' Call Light_Sensors subroutine
  GOSUB Display                            ' Call Display subroutine
  GOSUB Duty_Auto_Adjust                   ' Call Duty_Auto_Adjust subrtn
LOOP                                       ' Repeat main loop

'-----[ Subroutine - Light_Sensors ]----------------------------------------

Light_Sensors:                             ' Check P6 & P3 light sensors
  PWM 6, duty, 1                           ' Charge cap to duty value
  RCTIME 6, 1, tLeft                       ' P6->input, measure decay time
  PWM 3, duty, 1                           ' Charge cap to duty value
  RCTIME 3, 1, tRight                      ' P3->input, measure decay time
  RETURN                                   ' Return from subroutine

'-----[ Subroutine - Display ]----------------------------------------------
```

```
Display:                                ' Display times, L/R, and duty
  DEBUG HOME, "tLeft = ", DEC5 tLeft,   ' Display results
        "  ", "tRight = ", DEC5 tRight
  DEBUG CR, "Light brigther on "
  IF tLeft < tRight THEN DEBUG "Left "
  IF tleft > tRight THEN DEBUG "Right"
  PAUSE 250                             ' Wait 0.25 seconds
  DEBUG CR, "duty = ", DEC duty, CLREOL
  RETURN                                ' Return from subroutine

'-----[ Subroutine - Duty_Auto_Adjust ]-------------------------------------

' Adjust duty variable to keep tLeft + tRight in the 1800 to 2200 range.
' Requres tSum word variables for calculations.
Duty_Auto_Adjust:                       ' Duty Automatic Adjustment
  tSum = (tLeft + tRight) MAX 4000      ' Limit max tSum value
  IF tSum = 0 THEN tSum = 4000          ' If 0 (timeout) then 4000
  IF (tSum<=1800) OR (tSum>=2200) THEN  ' If outside 1800 to 2200
    tDiff = 2000 - tSum                 ' Find excursion from target val
    sign = tDiff.BIT15                  ' Pos/neg if .BIT15 = 0/1
    tDiff = ABS(tDiff) / 200            ' Max tDiff will be +/- 10
    tDiff = tDiff MAX ((duty-67)/2)     ' Reduce adjustment sizes near
    tDiff = tDiff MAX ((257-duty)/2)    '   ends of the range
    IF sign=NEGATIVE THEN tDiff=-tDiff  ' Restore sign
    duty = duty + tDiff MIN 70 MAX 255  ' Limit duty to 70 to 255
  ENDIF                                 ' End of outside 1800 to 2200
  RETURN                                ' Return from subroutine
```

Optional/Advanced Topic: Inside the Duty_Auto_Adjust Subroutine

In review, the **`Duty_Auto_Adjust`** subroutine tests the value of **`tLeft + tRight`** to see if it's in the 1800 to 2200 range. If it is, the subroutine doesn't have to make any changes. On the other hand, if the sum of those two variables is out of that range, the subroutine adjusts the value of the **`duty`** variable. If it increases the value of the **`duty`** variable, the **`PWM`** commands in the **`Light_Sensors`** subroutine will charge the capacitors to higher voltages, resulting in larger **`RCTIME`** measurements for a given light level. If it decreases the value, the opposite will occur. The capacitors will get charged to lower voltages which result in smaller **`RCTIME`** measurements for that same light level.

This is an earlier revision of Duty_Auto_Adjust. The version in the book uses one variable named **`sumDiff`** in place of two variables: **`tSum`** and **`tDiff`**.

The first thing **Duty_Auto_Adjust** does is add **tLeft** to **tRight** and store the result in a variable named **tSum**. Then, it limits this result to 4000, which is twice the ideal target value of 2000. So the range of **tSum** is 0 to 4000. Lower values of **tSum** indicate brighter light, but 0 is a special case. If both the **RCTIME** commands return 0, it means that both decay measurements took so long that they exceeded 65535 × 2 µs = 0.13107 s. If that's the case, it means it's not actually bright in the room, it's really dark. So, an **IF…THEN** statement checks for this and changes **tSum** to 4000 if it happens to be 0.

```
Duty_Auto_Adjust:
  tSum = (tLeft + tRight) MAX 4000
  IF tSum = 0 THEN tSum = 4000
```


Next the subroutine checks to find out if **tSum** is in the 1800 to 2200 range. If it is, all that's left for the subroutine to do is **RETURN**. If it's not, the subroutine will have to do some work to decide how much to increase or decrease the **duty** variable's value. It starts by subtracting **tSum** from 2000 and storing the result in a variable named **tDiff**. This new result is the difference between **tSum** and the target value of 2000. Since **tSum** can range from 0 to 4000, the **tDiff** result will be in the –2000 to +2000 range. Negative values mean the **duty** variable's value needs to be decreased because it's too dark and the measurements are too large. Positive values mean the **duty** variable's value needs to be increased because it's too bright and the measurements are too small.

```
  IF (tSum<=1800) OR (tSum>=2200) THEN
    tDiff = 2000 - tSum
```

Next the program has to divide **tDiff**, which stores a signed value in the +/– 2000 range, by 200. The result should be in the +/– 10 range, which causes a maximum change in the **duty** variable of +/– 10. PBASIC programs have to remember the sign of the number before performing division. Then, they have to do a division on the absolute value of the number (with an always-positive result), and then restore its sign when done.

Below are the first two of three steps required for dividing 200 into the signed value of the **tDiff** variable. A rule of signed word variables (–32768 to 32767) is that the highest bit in the variable (that's BIT 15) will store a 1 if the variable is negative or a 0 if it's positive. For more information on this, use the PDF search tool to look up references to "two's complement" in the BASIC Stamp Manual. So, by copying the value of **tDiff.BIT15**, into the sign bit variable, the program remembers the **sign** of the variable

before the division started. Then, it divides the absolute value of **tDiff** by 200. The result that gets stored in **tDiff** is positive.

```
sign = tDiff.BIT15
tDiff = ABS(tDiff) / 200
```

Before the sign of the variable can be restored, the value of the adjustment has to be checked to make sure it's not too close to the ends of the **duty** variable's valid range of 70 to 255. When it gets to below 80 or above 245, the two lines of code below prevent the value of **tDiff** from a step size that's larger than 6. In the next iteration of the loop, it is limited to a maximum step size of 3, then a maximum step size of 1. The result for big light changes will be that it skips in increments of 10 with each repetition of the main routine, but when it reaches 80 or 245, it has to take progressively smaller steps to reach the endpoints of the range. This step size limiting near the end points prevents a situation where the program could get stuck skipping back and forth between 70 and 80, or between 245 and 255. If this was allowed to happen, the program would never make the fine adjustment to a value like 72 or 251.

```
tDiff = tDiff MAX ((duty-67)/2)
tDiff = tDiff MAX ((257-duty)/2)
```

As mentioned earlier, the third step in signed division with PBASIC is to restore the sign of the variable. This is done by comparing the **sign** variable to the **NEGATIVE** constant (a value of 1). If the sign variable is equal to 1, it means **tDiff.BIT15** stored a 1 indicating the original value was negative. In that case, the program has to make **tDiff** equal to **–tDiff** to restore its sign.

```
IF sign=NEGATIVE THEN tDiff=-tDiff
```

Finally, the value is limited from stepping outside the 70 to 255 range.

```
  duty = duty + tDiff MIN 70 MAX 255
ENDIF
RETURN
```

The MIN and MAX operators limit a result to above or below certain values. For example the command **`variable = variable MIN 100 MAX 200`** limits variables value to a range of 100 to 200. If it's value before the command was 55, it would get changed to 100. If its value was in the 100 to 200 range, its value would not be changed. If its value was above 200 before the command, it would be changed to 200.

ACTIVITY #9: UNDERSTANDING LIGHT VARIABLE CALCULATIONS

The Boe-Bot can use differential light measurements for deciding whether to turn away from or toward a light source, but tasks like detecting when it's under a bright lamp at the end of a course also requires an ambient light measurements, in other words, a measurement of how bright or dark it is.

Earlier versions of the **`Light_Shade_Info`** subroutine were just for normalized differential measurements and automatic light sensitivity adjustment. When the question of determining overall brightness came up, a few additional calculations had to be added to **`Light_Shade_Info`** for reporting ambient levels. The measurement is not precise, but it's good enough for general Boe-Bot navigation.

The code that was added to **`Light_shade_Info`** for ambient light detection is between calls to **`Light_Sensors`** and **`Duty_Auto_Adjust`**. At that point, the **`Light_Sensors`** subroutine had loaded new measurements into **`tLeft`** and **`tRight`**, and the **`duty`** value used to set light sensitivity was determined during the previous call, which was only a fraction of a second before and still useful.

```
GOSUB Light_Sensors                       ' Get raw RC light measurements
sumdiff = (tLeft + tRight) MAX 65535      ' Start light level with sum
IF duty <= 70 THEN                        ' If duty at min
  light=duty-(sumdiff/905) MIN 1          ' Find how much darker
  IF sumdiff = 0 THEN light = 0           ' If timeout, max darkness
ELSEIF duty = 255 THEN                    ' If duty at max
  light=duty+((1800-(sumdiff))/26)        ' Find how much brighter
ELSE                                      ' If duty in range
  light = duty                            ' light = duty
ENDIF                                     ' Done with light level
GOSUB Duty_Auto_Adjust                    ' Adjust PWM duty for next loop
```

Let's start by looking at the **`ELSE`** condition. That block gets executed if the **`duty`** variable is in the 71…254 range. If its in this range, it typically means that the **`Duty_Auto_Adjust`** subroutine is succeeding at keeping the sum of **`tLeft + tRight`** in

the 1800 to 2000 range. In this case, the program just copies whatever value the **duty** variable stores to the light variable for the indicator of incident light.

```
...

ELSE                                          ' If duty in range
  light = duty                                ' light = duty
...
```

Remember from Activity #3 that the PWM command cannot charge the capacitor high enough for an **RCTIME** measurement if it's below 1.4 V. Charging a capacitor up to this value requires a **PWM** command with its ***Duty*** argument of 72 or higher. In practice, the code still works at slightly lower levels. **Duty_Auto_Adjust** uses 70 as a minimum in Activity #4-#6, and the code for calculating the ambient light level (**light** variable) uses 71.
.

The **Duty_Auto_Adjust** subroutine in LightSensorValues.bs2 prevents the **duty** variable from falling below a value of 70. When the **duty** variable is stuck at 70, it indicates that the light is too dim for the **Duty_Auto_Adjust** subroutine to do anything more to correct. Since the **duty** variable is 70, the **Light_Sensors** subroutine is only charging the capacitors up to 1.4 V, and there is almost no room for decay time. That's fine, because the phototransistors conduct very little current under low light conditions, so the capacitors discharge very slowly.

Because the capacitors discharge so slowly in low light, even though they are being charged to a minimum value, they can still take different amounts of time to decay, which indicates different light level values for calculating the **ndShade** variable. So the Boe-Bot may still be able to navigate, it just depends on how low the light levels are.

When the **duty** variable is stuck at 70, the program needs information from another source to find out what the light level is. After a call to the **Light_Sensors** subroutine, the decay time measurements are stored in the **tLeft** and **tRight** variables. The first thing the **Light_Shade_Info** subroutine does is store the sum of **tLeft** and **tRight** in a variable named **sumdiff**. Under the low light conditions, this variable should store a value in the 2200 to 65535 range, with larger values indicating lower light. In the interest of limiting this larger range to 70 levels (69 to 0), we can simply divide 70 into (65535 – 2200), which turns out to be 904.8 ≈ 905. So, when **duty** is stuck at 70, the expression **light = duty – (sumdiff/905) MIN 1** is roughly equivalent to **70 – (sumdiff /**

905). Keep in mind that **sumdiff** will be somewhere in 2200 to 65535 range. This puts **sumdiff / 905** in the 2 to 72 range. In the end, the light variable will return values from 1 to 68. So, it might actually be better to rewrite the expression: **light=duty-(sumdiff/905) – 2**.

```
...
GOSUB Light_Sensors                         ' Get raw RC light measurements
sumdiff = (tLeft + tRight) MAX 65535        ' Start light level with sum
IF duty <= 70 THEN                          ' If duty at min
  light=duty-(sumdiff/905) MIN 1            ' Find how much darker
  IF sumdiff = 0 THEN light = 0             ' If timeout, max darkness
...
```


In the code above, the last line is **IF sumdiff = 0 THEN light = 0**. This covers the situation where both capacitors still take so long to discharge that they time out, even though the **PWM** command only charges them up to about 1.4 V. This is a really low light situation, and when the measurement times out, instead of a really large value in **tLeft** and **tRight**, the result is 0. So, when both light sensor measurements time out, the **IF…THEN** statement makes the light variable 0 instead of 70.

If it instead turns out that the light is bright, the phototransistors will conduct much more current, causing more rapid capacitor voltage decay. The **Duty_Auto_Adjust** subroutine responds by increasing the **duty** variable. Since the **duty** variable is used by the **Light_Sensors** subroutine to set the initial voltage across the capacitors, it results in larger starting voltages, which will increase decay times. But, at some point, the light gets too bright and the decay times become more rapid, and their sum falls below 1800 even when the **Duty_Auto_Adjust** subroutine has the duty variable maxed out at 255.

When this situation occurs, the **sumdiff** variable, which stores the result of **tLeft + tRight**, will be somewhere between 1800 and 1. The value of **sumdiff** will be smaller as the light gets brighter, so **sumdiff** has to be subtracted from 1800 for an intermediate result that gets larger with brighter light. This intermediate result is divided by 26 to give 69 extra levels above 255, and then added to **duty**, which has to be 255 for this code to get executed. The end result? Values in the 255 to 324 range.

```
ELSEIF duty = 255 THEN                      ' If duty at max
  light=duty+((1800-(sumdiff))/26)          ' Find how much brighter
```