Accelerometer - Tilt, Graphics and Video Games

The accelerometer is featured in lots of HIDs. HID is short for Human Interface Device, and it includes computer mice, keyboards, and more generally, anything that makes it possible for humans to interact with microprocessors. With limited space on PDAs like the one in Figure 1, tilt control eliminates the need for extra buttons. Tilt control is also a popular feature in certain game controllers.

Figure 1
Tilt Controlled
Game on a PDA

The circuit in products like these is similar to the one introduced in Accelerometer - Getting Started. If you haven't already built and tested the circuit and tried the examples in Activity #1 of Accelerometer - Getting Started, do it first before continuing here.

Where can I find Accelerometer - Getting Stared?

√ Go to the www.parallax.com home page, and enter 28017 into the search field.

√ This will take you to the Memsic 2125 Dual-axis Accelerometer page.

√ Follow the Stamps in Class Memsic Tutorial (.pdf) link.

This chapter has four activities that demonstrate the various facets of using tilt to control a display. Here are summaries of each activity:

- ***Activity #1***: PBASIC Graphic Character Display – introduces some Debug Terminal cursor control and coordinate plotting basics.

The draft material in this Chapter is part of a forthcoming Stamps in Class text by Andy Lindsay.

To post feedback or suggestions, go to http://forums.parallax.com/forums/default.aspx?f=6&m=57588.
Software for BASIC Stamp® Modules and applications are available for free download from www.parallax.com.

- ***Activity #2***: Background Store and Refresh with EEPROM – Each time your game character moves, whatever it was covering up on the screen has to be re-drawn. This activity demonstrates how you can move your character and refresh the background with the help of the BASIC Stamp's EEPROM.
- ***Activity #3***: Tilt the Bubble Graph – With a moving asterisk on a graph, this first application demonstrates how the hot air pocket inside the MX2125 moves when you tilt it. At the same time, it puts the accelerometer fundamentals to work along with the techniques from Activity #2.
- ***Activity #4***: Game Control Example – You are now ready to use tilt to start controlling your game character. The background characters can be used to make decisions about whether your game character is in or out of bounds. Have fun customizing and expanding this tilt controlled video game.

ACTIVITY #1: PBASIC GRAPHIC CHARACTER DISPLAY

This activity introduces some programming techniques you will use to graphically display coordinates with the Debug Terminal. Certain elements of the techniques introduced in this and the next activity are commonly used with liquid crystal and other small displays as well as in certain digital video technologies like MPEG.

The CRSRXY and Other Control Characters

The **`DEBUG`** command's **`CRSRXY`** control character can be used to place the cursor at a location on the Debug Terminal's receive windowpane. For example, **`DEBUG CRSRXY, 7, 3, "*"`** places the asterisk character seven spaces to the right and three characters down. Instead of using constants like 7 and 3, you can use variables to make the placement of the cursor adjustable. Let's say you have two variables, **`x`** and **`y`**, the values these variables store can control the placement of the asterisk in the command **`DEBUG CRSRXY, x, y, "*"`**.

The next example program also makes use of the **`CLRDN`** control character. The command **`DEBUG CLRDN`** causes all the lines below the cursor's current location to be erased.

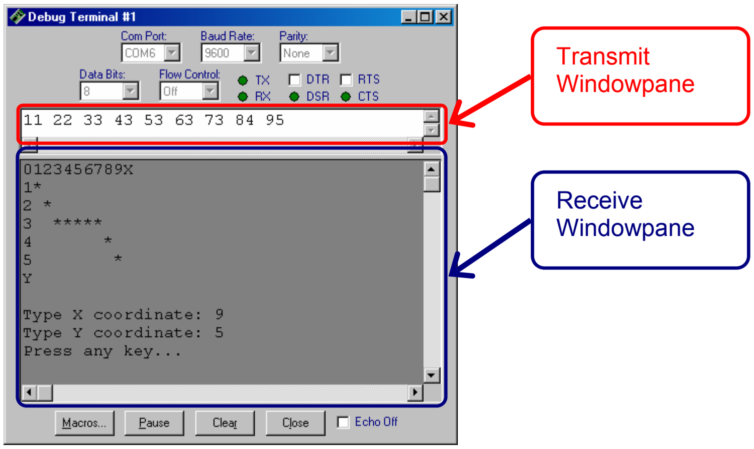
More Control Characters

You can find out more about control characters by looking up the DEBUG command, either in the PBASIC Syntax Guide or the BASIC Stamp Manual. You can get to the PBASIC Syntax guide through your BASIC Stamp Editor (v2.0 or newer). Just click Help and select Index. The BASIC Stamp Manual is available for free download from www.parallax.com → Downloads → Documentation.

Example Program – CrsrxyPlot.bs2

With this program, you can type pairs of digits into the Transmit Windowpane (see Figure 2) to position asterisks on the receive windowpane. Simply click the transmit windowpane and start typing. The first digit you type is the number of spaces to the right to place the cursor, and the second number is the number of carriage returns downward. Before typing a new pair of digits, press the space bar once.

Figure 2 - Debug Terminal Transmit and Receive Windowpanes



Com Port: COM6
Baud Rate: 9600
Parity: None
Data Bits: 8
Flow Control: Off

```
11 22 33 43 53 63 73 84 95
```

```
0123456789X
1*
2 *
3  *****
4       *
5        *
Y

Type X coordinate: 9
Type Y coordinate: 5
Press any key...
```

√ Enter, save, and run CrsrxyPlot.bs2

√ Follow the prompts and type digits into the Debug Terminal's transmit windowpane to place asterisks on the plot.

√ Try the sequence 11, 22, 33, 43, 53, 63, 73, 84, 95. Do the asterisks in your Debug Terminal match the pattern in the example?

√ Try predicting the sequences for various shapes, like a square, triangle, and circle.

√ Enter the sequences to test your predictions.

√ Correct the sequences as needed.

```
' Accelerometer Projects
' CrsrxyPlot.bs2

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

x       VAR    Word
```

```
y      VAR   Word
temp   VAR   Byte

DEBUG CLS,
"0123456789X", CR,
"1          ", CR,
"2          ", CR,
"3          ", CR,
"4          ", CR,
"5          ", CR,
"Y          ", CR, CR

DO

  DEBUG "Type X coordinate: "
  DEBUGIN DEC1 x
  DEBUG CR, "Type Y coordinate: "
  DEBUGIN DEC1 y

  DEBUG CRSRXY, x, y, "*"

  DEBUG CRSRXY, 0, 10, "Press any key..."
  DEBUGIN temp
  DEBUG CRSRXY, 0, 8, CLRDN

LOOP
```

Your Turn – Keeping Characters in the Plot Area

If you type the digit 8 in response to the prompt `"Type Y coordinate: "`, it will overwrite your text. Similar problems occur if you type 0 for either the X or Y coordinates. The asterisk is plotted over the text that shows which row and column **`CRSRXY`** is plotting. One way to fix this is with the **`MAX`** and **`MIN`** operators. Simply add the statement **`y = y MAX 5 MIN 1`**. The **`DEBUGIN`** command's **`DEC1`** operator solves this problem for the maximum X coordinate, since it is limited to a value from 0 to 9. So, all you'll need to clamp the X value is **`x = x MIN 1`**.

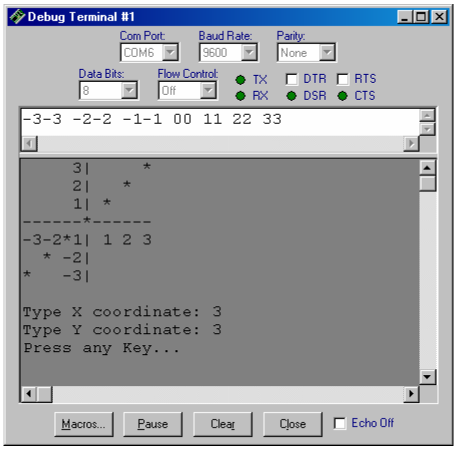
√ Try entering out of bounds values for the Y coordinate (0 and 6 to 9) and 0 for the X coordinate.
√ Observe the effects on the display's background.
√ Modify CrsrxyPlot.bs2 as shown here and try it again

```
  DEBUG CR, "Type Y coordinate: "
  DEBUGIN DEC1 y

  Y = y MAX 5 MIN 1                ' <--- Add
  X = x MIN 1                         ' <--- Add

  DEBUG CRSRXY, x, y, "*"
```

Scale and Offset

Scale and offset were introduced in both *What's a Microcontroller* and *Robotics with the Boe-Bot*. In *What's a Microcontroller*, they were used to adjust servo position based on input, and in *Robotics with the Boe-Bot*, they were used to calibrate light sensors. Here is scale and offset again, this time for positioning characters on a display.

Take a look at the example in Figure 3. When you type in -3-3 into the Debug Terminal's transmit windowpane, it doesn't automatically appear at the (-3, -3) position on the graph. The asterisk actually needs to be placed 0 spaces over and 6 carriage returns down. Here is a second example. When you type-in 2,2, **CRSRXY** actually needs to place the cursor at 10 spaces over and one carriage return down.



Com Port: COM6 Baud Rate: 9600 Parity: None
Data Bits: 8 Flow Control: Off

```
-3-3 -2-2 -1-1 00 11 22 33
```

```
     3|     *
     2|   *
     1| *
------*------
-3-2*1| 1 2 3
  * -2|
*   -3|

Type X coordinate: 3
Type Y coordinate: 3
Press any Key...
```

Figure 3
Entering and
Displaying Coordinates

For values ranging from -3 to 3, the X value has to be multiplied by 2 and added to 6 for **CRSRXY** to place the asterisk the right number of spaces over. That's a scale of 2, and an offset of 6. Here is a PBASIC statement to make the conversion from X coordinate to number of spaces.

```
  x = (x * 2) + 6
```

The Y value has to be multiplied by -1, then added to 3. That's a scale of -1 and an offset of 3. Here is a PBASIC statement to make the conversion from Y coordinate to number of carriage returns.

```
y = 3 - y
```

√ Try substituting X and Y coordinates in the right side of each of these equations, do the math, and verify that each equation yields the right number of spaces and carriage returns.

Example Program – PlotXYGraph.bs2

√ Enter and run PlotXYGraph.bs2.

√ Try entering the sequence of values: -3-3 -2-2 -1-1 00 11 22 33 and verify that it matches the Debug Terminal example.

√ Try some other sequences and/or drawing shapes by their coordinates.

```
' Accelerometer Projects
' PlotXYGraph.bs2

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

x              VAR     Word
y              VAR     Word
temp           VAR     Byte

DEBUG CLS,
"     3|      ", CR,
"     2|      ", CR,
"     1|      ", CR,
"------+------", CR,
"-3-2-1| 1 2 3", CR,
"    -2|      ", CR,
"    -3|      ", CR, CR

DO

  DEBUG "Type X coordinate: "
  DEBUGIN SDEC1 x
  DEBUG CR, "Type Y coordinate: "
  DEBUGIN SDEC1 y

  x = (x * 2) + 6
  y = 3 - y

  DEBUG CRSRXY, x, y, "*"
  DEBUG CRSRXY, 0, 10, "Press any Key..."
  DEBUGIN temp
  DEBUG CRSRXY, 0, 8, CLRDN

LOOP
```

Your Turn – More Keeping Characters in the Plot Area

You can also use **IF…THEN** statements to handle values that are out of bounds. Here is an example of how you can modify PlotXyGraph.bs2 with **IF…THEN**. Instead of clipping the value, the program just waits until a correct value is entered.

√ Modify PlotXYGraph.bs2 as shown here, and then run it. Verify that this program does not allow you to enter characters outside the range of -3 to 3.

```
x = (x * 2) + 6
y = 3 - y

IF (x > 12) OR (y > 6) THEN                ' <--- Add/modify from here...
  DEBUG CRSRXY, 0, 8, CLRDN,               '
       "Enter values from -3 to 3.", CR,   '
       "Try again"                         '
                                           '
ELSE                                       '
                                           '
  DEBUG CRSRXY, x, y, "*"                  '
                                           '
ENDIF                                      ' <--- to here

DEBUG CRSRXY, 0, 10, "Press any Key..."
DEBUGIN temp
```

What negative numbers?

The conditions for the **IF...THEN** statement in your modified version of PlotXYGraph.bs2 are **(x > 12) OR (y > 6)**. This covers positive numbers that are larger than 12 or 6, but it also covers all negative numbers. That's because the BASIC Stamp uses a system called twos complement to store negative numbers. In twos complement, the unsigned version of any negative value is larger than any positive value. For example, -1 is 65535, -2 is 65534, and so on, down to -32768, which is actually 32768. Signed positive values only range from 1 to 32767.

Twos complement is the most common form of negative number storage in both microcontrollers and computers. The reason twos complement is so popular is because its rules are very simple at the binary computing level. If you don't already know the rules for twos complement, try this program, and see if you can figure them out:

```
' Accelerometer Projects
' TwosComplementExample.bs2

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

counter VAR Word

DEBUG "Signed  Unsigned  Binary          ", CR,
      "------  --------  ----------------", CR

FOR counter = - 8 TO -1
  DEBUG SDEC counter, "      ",
        DEC counter,  "     ",
        BIN16 counter, CR
  PAUSE 100
NEXT

FOR counter = 0 TO 8
  DEBUG " ", SDEC counter,
        "      ", DEC counter,
        "         ", BIN16 counter, CR
  PAUSE 100
NEXT

END
```

When writing **IF...THEN** statements that examine negative values for the BASIC Stamp, always keep three things in mind:

1) The BASIC Stamp makes unsigned **IF...THEN** comparisons.
2) Negative values are always larger than positive values.
3) You can always recognize a negative number by testing if its Bit15 is one.
For example, **IF counter.bit15 = 1 THEN**...

Algebra to Determine Scale and Offset

The XY plot displayed in the Debug Terminal in this activity is called the Cartesian coordinate system. Named after 17th century mathematician René Descartes, this system is the basis for graphing techniques used in many mathematical pursuits. Shown in Figure 4, the Cartesian coordinate system's is most commonly displayed with (0, 0) in the center of the graph. Its values get larger going upward (y-axis) and to the right (x-axis). Most displays behave differently, with coordinate 0, 0 starting at the top-left. While the x-axis increases toward the right, the y-axis increases downward.

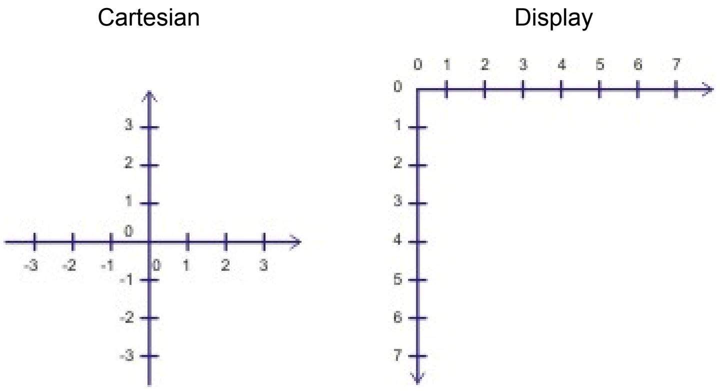


Figure 4
Cartesian vs.
Display
Coordinates

You can use a standard algebra technique, solving two equations in two unknowns, to figure out the statements you will need to transform Cartesian coordinates into debug terminal coordinates. This next example shows how it was done for the statements that converted x and y from Cartesian to display coordinates in PlotXYGraph.bs2.

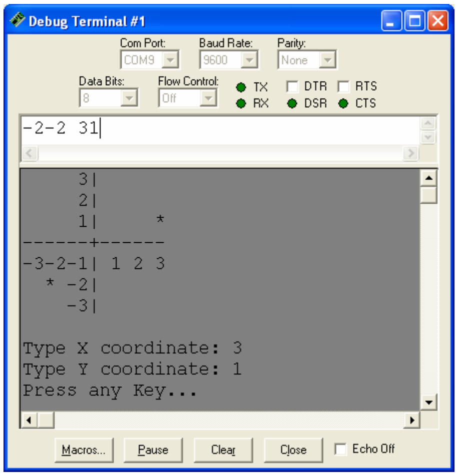
By adding a couple of **DEBUG** commands, you can display the before and after versions of the X-value you entered.

```
DEBUG "Type X coordinate: "
DEBUGIN SDEC1 x
DEBUG CR, "Type Y coordinate: "
DEBUGIN SDEC1 y

DEBUG CRSRXY, 0, 12, "x before: ", SDEC1 x    ' <--- Add

x = (x * 2) + 6
y = 3 - y

DEBUG CRSRXY, 0, 14, "x after:  ", SDEC1 x    ' <--- Add
DEBUG CRSRXY, x, y, "*"
```

- √ Save PlotXyGraph.bs2 under another name, like PlotXyGraphBeforeAfter.bs2.
- √ Add the two **DEBUG** commands that display the "before" and "after" values of x.
- √ Add two more **DEBUG** commands to display the “before” and “after” values of y.
- √ Enter the coordinates (3,1) and (-2,-2) into the Debug Terminal's transmit windowpane. See Figure 5.
- √ Record the after values in the table.

Table: 1 Values Before and After

Coordinate	**before**	**After**
(3, 1)	3	
(-2, 2)	-2	



```
-2-2 31
```

```
     3|
     2|
     1|     *
------+------
-3-2-1| 1 2 3
  * -2|
    -3|


Type X coordinate: 3
Type Y coordinate: 1
Press any Key...
```

Figure 5
Test Coordinates

When designing a display to show Cartesian coordinates, it helps to take a couple of before and after values like the one's in Table 1. You can then use them to solve for scale (K) and offset (C) using two equations with two unknowns.

$$X_{after} = (K \times X_{before}) + C$$

The usual steps for two equations in two unknowns are:

(1) Substitute your two before and after data points into separate copies of the equation.

$$12 = (K \times 3) + C$$
$$2 = (K \times \text{-}2) + C$$

(2) If needed, multiply one of the two equations by a term that causes the number of one of the unknowns in the top and bottom equations to be equal.

Not needed, because the coefficient of C in both equations is 1.

(3) Subtract one equation from the other to make one of the unknowns zero.

$$\begin{array}{r} 12 = (K \times 3) + C \\ -\left[\,2 = (K \times \text{-}2) + C\right] \\ \hline 10 = K \times 5 \end{array}$$

(4) Solve for the unknown that did not subtract to zero.

$$10 = K \times 5$$
$$K = \frac{10}{5}$$
$$K = 2$$

(5) Substitute the value you solved in step 4 into one of the original two equations.

$$12 = (2\times3) + C$$

(6) Solve for the second unknown.

$$12 = (2\times3) + C$$
$$12 = 6 + C$$
$$C = 12\text{-}6$$

$$C = 6$$

(7) Incorporate solved unknowns into your equation.

$$X_{after} = (K \times X_{before}) + C$$
$$K = 2 \text{ and } C = 6$$
$$X_{after} = (2 \times X_{before}) + 6$$

Your Turn – Y-Axis Calculations

√ Modify your program so that it displays the Y-Axis before and after values.
√ Fill in the table for the Y-axis values:

Table: Y Values Before and After

Coordinate	**before**	**After**
(3, 1)	1	
(-2, 2)	2	

√ Repeat steps 1-7 for the Y-Axis equation. The correct answer is $y_{after} = (-1 \times y_{before}) + 3$.

ACTIVITY #2: BACKGROUND STORE AND REFRESH WITH EEPROM

In a video game, when your game character isn’t on the screen, all that’s visible is the background. As soon as your game character enters the screen, it blocks out part of the background. When the character moves, two things have to happen: (1) the game character has to be re-drawn at the new location, and (2) the background that the game character was blocking out has to be re-drawn. If step 2 never happened in your program, your screen would fill up with copies of your game character.

Televisions and CRT computer monitors refresh every pixel many times per second. The refresh rate on televisions is around 30 Hz, and a few of the more common refresh rates on CRTs are 60, 70, and 72 Hz. Other devices like certain LCD and LED displays hold the image automatically, or sometimes with the help of another microcontroller. All the program or microcontroller that controls these devices has to do is tell them what to display or change. This is also how video compression on your computer works. In order to reduce the file size, some compressed video files store the changes to the image instead of all the pixels in a given image frame.

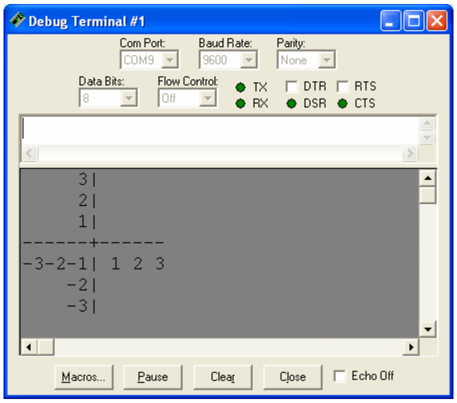
When used with displays that do not need to be refreshed (like the Debug Terminal or an LCD), the BASIC Stamp's can store an image of a game or graph background in its EEPROM. When a game character moves and is redrawn at a different location, the BASIC Stamp can just redraw the background characters at the game characters old location. All you have to do is save the old coordinates of the game character before it moved and then use those coordinates to retrieve the background characters from EEPROM. Depending on how large the display is, this can save a considerable amount of time that the BASIC Stamp might need to perform other tasks.

This activity introduces three elements to game characters and backgrounds:

(1) Storing and displaying the background from EEPROM
(2) Tracking a character's old and new coordinates
(3) Redrawing the old coordinates from EEPROM.

Background Display from EEPROM

This display doesn't have to be made with a single **DEBUG** command, especially if it needs to be maintained as a background with characters traveling over it in the foreground. Instead, it's better to store the characters in EEPROM and then display them individually with a **FOR…NEXT** loop that uses **READ** and **DEBUG** commands to display individual characters. Figure 6 is a display generated with this technique.



Com Port: COM9
Baud Rate: 9600
Parity: None
Data Bits: 8
Flow Control: Off

```
     3|
     2|
     1|
------+------
-3-2-1| 1 2 3
    -2|
    -3|
```

Figure 6
Background
from DATA

You can use the **DATA** directive to store a background in EEPROM. Notice how this **DATA** directive stores 100 characters (0 to 99). Notice also that each row is 14 characters wide when you add the **CR** control character. It makes programming much easier if each row is the same width. Otherwise, finding the character you want become s a more complex problem.

```
  DATA CLS,                     ' 0
  "     3|      ", CR,          ' 14
  "     2|      ", CR,          ' 28
  "     1|      ", CR,          ' 42
  "------+------", CR,          ' 56
  "-3-2-1| 1 2 3", CR,          ' 70
  "    -2|      ", CR,          ' 84
  "    -3|      ", CR, CR       ' 98 + 1 = 99
```

You can then use a **FOR…NEXT** loop to retrieve and display each character stored in EEPROM. The net effect is the same as a long **DEBUG** command.

```
FOR index = 0 TO 99
  READ index, character
  DEBUG character
NEXT
```

Example Program – EepromBackgroundDisplay.bs2

√ Enter, save, and run the program.

√ Verify that the display is the same as PlotXyGraph.bs2.

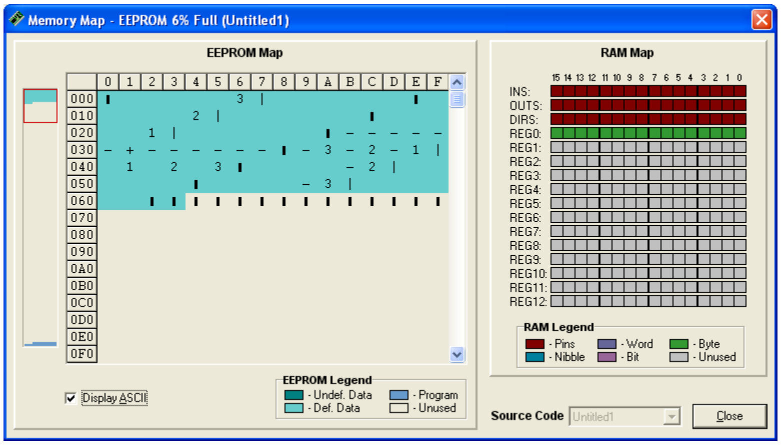
```
' Accelerometer Projects                      ' Program
' EepromBackgroundDisplay.bs2

'{$STAMP BS2}                                ' Stamp & PBASIC Directives
'{$PBASIC 2.5}

index          VAR     Byte                   ' Variables
character      VAR     Byte

DATA CLS,                     ' 0            ' Store background in EEPROM
"     3|      ", CR,          ' 14
"     2|      ", CR,          ' 28
"     1|      ", CR,          ' 42
"------+------", CR,          ' 56
"-3-2-1| 1 2 3", CR,          ' 70
"    -2|      ", CR,          ' 84
"    -3|      ", CR, CR       ' 98 + 1 = 99

FOR index = 0 TO 99                          ' Retrieve and display background
```

```
  READ index, character
  DEBUG character
NEXT

END
```

Your Turn – Viewing the EEPROM Characters

√ In the BASIC Stamp Editor, click Run and select Memory Map.
√ Click the Display Ascii box in the lower left corner of the Memory Map window.
√ The digits, dashes, and vertical bars should appear exactly as shown in Figure 7.
√ Instead of 14 characters per row, the EEPROM map shows 16. Verify that you have a total of 100 (0 to 99) characters stored for display purposes in EEPROM.

Figure 7 - Display Characters Stored in EEPROM



Memory Map - EEPROM 6% Full (Untitled1)

EEPROM Map

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	▌						3	|							▌	
010					2	|							▌			
020			1	|							▌	-	-	-	-	-
030	-	+	-	-	-	-	-	-	▌	-	3	-	2	-	1	|
040		1		2		3	▌					-	2	|		
050					▌					-	3	|				
060			▌	▌	▌	▌	▌	▌	▌	▌	▌	▌	▌	▌	▌	▌
070																
080																
090																
0A0																
0B0																
0C0																
0D0																
0E0																
0F0																

☑ Display ASCII

Tracking a Character's Old and New Coordinates

Let's say you want to track the previous X and Y coordinates in PlotXYGaph.bs2 from Activity #1. It takes two steps:

(1) Declare a couple variables for storing the old values, **`xOld`** and **`yOld`** for example.

```
x               VAR     Word
y               VAR     Word
```

```
xOld           VAR     Nib                   ' <--- Add
yOld           VAR     Nib                   ' <--- Add

temp           VAR     Byte
```

(2) Before loading new values into the **x** and **y** variables, store the current value of **x** into **xOld** and the current value of **y** into **yOld**.

```
DO

  xOld = x                                   ' <--- Add
  yOld = y                                   ' <--- Add

  DEBUG "Type X coordinate: "
```

Why are x and y words while xOld and yOld are nibbles?

When working with signed values, word variables store both the value and the sign.

At the particular place that **xOld** and **yOld** are used in the program, they are only storing values that range from 0 to 12, so all we need are nibble variables.

Here's a third step you can use to test and verify that it works:

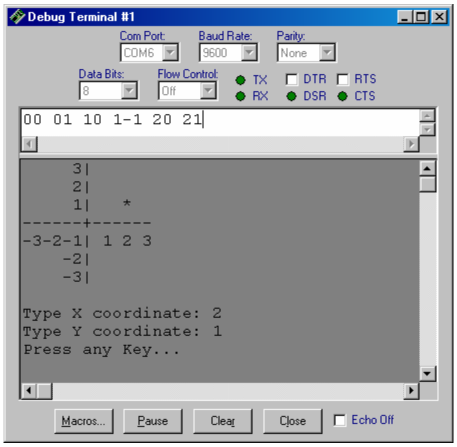
(3) Before loading new values into the x and y variables, store the current value of **x** into **xOld** and the current value of **y** into **yOld**. Keep in mind that both values will be in terms of Debug Terminal coordinates. Also keep in mind that the first time through, the old coordinates will be (0, 0) since all variables initialize to zero in PBASIC.

```
  DEBUG CRSRXY, x, y, "*"

  DEBUG CRSRXY, 0, 10,                       ' <--- Add
        "Current entry:  (",
        DEC x, ",", DEC y, ")"
  DEBUG CRSRXY, 0, 11,                       ' <--- Add
        "Previous entry: (",
        DEC xOld, ",", DEC yOld, ")"
  DEBUG CRSRXY, 0, 12, "Press any Key..."    ' <--- Modify

  DEBUGIN temp
```

√ Start with PlotXYGraph.bs2, save it under a new name, and try the modifications just discussed.

Re-Drawing the Background

The net effect we want for game control is to make the asterisk disappear from its old location and appears in its new location whenever it moves. To make it appear at its new location, simply use a **DEBUG** command to display the asterisk at its current coordinates. To make the asterisk disappear from its old coordinates, the background character that was there has to be looked up in EEPROM and then displayed with **DEBUG**. Notice that six ordered pairs were entered into the Debug Terminal shown in Figure 8, but there is only one asterisk, and it corresponds with the last pair that was entered.



Com Port: COM6
Baud Rate: 9600
Parity: None
Data Bits: 8
Flow Control: Off

```
00 01 10 1-1 20 21
```

```
     3|
     2|
     1|   *
------+------
-3-2-1| 1 2 3
    -2|
    -3|

Type X coordinate: 2
Type Y coordinate: 1
Press any Key...
```

Figure 8
Display with Background Refresh

Here is a routine you can add to PlotXYGraph.bs2 to accomplish this:

```
DEBUG CRSRXY, x, y, "*"

index = (14 * yOld) + xOld + 1              ' <--- Add
READ index, character                       ' <--- Add
DEBUG CRSRXY, xOld, yOld, character         ' <--- Add
```

The **index** variable selects the correct character from EEPROM. The **x** value is the number of spaces over and the **y** value is the number of carriage returns down. To get to the correct address of a character on the third row, your program has to add all the characters in the first two rows. Since each row has 14 characters, **yOld** has to be multiplied by 14 before it can be added to **xOld**. The extra 1 is added to skip the **CLS** at address 0.

Regardless of whether it's a computer display, the liquid crystal display on your cell phone, or your BASIC Stamp application's display, the same technique applies. The processor remembers two different images, the one in the background, and the one in the foreground. As the foreground object moves, it is displayed in a different location and the area that the foreground object used to occupy is re-drawn.

The most important thing to keep in mind about this programming technique is that it saves the processor lots of time. It only has to get one character from EEPROM and send it to the debug terminal. Compared to 99 characters, that's a significant time savings, and the BASIC Stamp can be doing other things with that time, such as monitoring other sensors, controlling servos, etc.

Example Program – EeprogrmBackgroundRefresh.bs2

This is a modified version of PlotXYGraph.bs2 with the background display, coordinate storage, and background redraw techniques introduced in this activity.

√ Enter save and run EepromBackgroundRefresh.bs2.

√ Test and verify that the asterisk disappears form its old location and appears at the new location you entered.

```
' -----[ Title ]-----------------------------------------------------------
' Accelerometer Projects                      ' Program info
' EepromBackgroundRefresh.bs2

'{$STAMP BS2}                                 ' Stamp/PBASIC directives
'{$PBASIC 2.5}

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

x              VAR     Word                   ' Store current position
y              VAR     Word

xOld           VAR     Nib                    ' Store previous position
yOld           VAR     Nib

temp           VAR     Byte                   ' Dummy variable for DEBUGIN

index          VAR     Byte                   ' READ index/character storage
character      VAR     Byte

' -----[ EEPROM Data ]-----------------------------------------------------

DATA CLS,                                     ' Display background
"      3|       ", CR,           ' 14
"     2|      ", CR,          ' 28
"     1|      ", CR,          ' 42
"------+------", CR,          ' 56
"-3-2-1| 1 2 3", CR,          ' 70
"    -2|      ", CR,          ' 84
"    -3|      ", CR, CR       ' 98 + 1 = 99

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

FOR index = 0 TO 99                          ' Display background
  READ index, character
  DEBUG character
NEXT

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

DO

  xOld = x                                   ' Store previous coordinates
  yOld = y

  DEBUG "Type X coordinate: "                ' Get new coordinates
  DEBUGIN SDEC1 x
  DEBUG CR, "Type Y coordinate: "
  DEBUGIN SDEC1 y

  x = (x * 2) + 6                            ' Cartesian to DEBUG values
  y = 3 - y

  DEBUG CRSRXY, x, y, "*"                    ' Display asterisk

  index = (14 * yOld) + xOld + 1             ' Redisplay background
  READ index, character
  DEBUG CRSRXY, xOld, yOld, character

  DEBUG CRSRXY, 0, 10, "Press any Key..."    ' Wait for user
  DEBUGIN temp
  DEBUG CRSRXY, 0, 8, CLRDN                  ' Clear old info

LOOP
```

Your Turn - Redrawing the Background without Extra Variables

Keeping track of the old location of the foreground character isn't always necessary. Think about it this way: in EepromBackgroundRefresh.bs2 the x and y variables store the old values *until you enter new values*. By simply rearranging the order that the **x** and **y** variables are displayed in, you can eliminate the need for **xOld** and **yOld**.

Next is a replacement main routine you can try in EepromBakcgroundRefresh.bs2. As soon as you press the space bar, your old asterisk disappears. The new asterisk reappears when you type the second of the two coordinates. As you will see in the next activity, this technique works really well when the refresh rate is several times per second with tilt control.

- √ Save EepromBakcgroundRefresh.bs2 as EepromBackgroundRefreshYourTurn.bs2.
- √ Comment the xOld and yOld variable declarations.
- √ Replace the Main Routine in EepromBackgroundRefresh.bs2 with this one.
- √ Test it and examine the change in the program's behavior.

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

DO

  index = (14 * y) + x + 1                     ' Redisplay background
  READ index, character
  DEBUG CRSRXY, x, y, character

  DEBUG CRSRXY, 0, 8,                          ' Get new coordinates
        "Type X coordinate: "
  DEBUGIN SDEC1 x
  DEBUG CR, "Type Y coordinate: "
  DEBUGIN SDEC1 y

  x = (x * 2) + 6                              ' Cartesian to DEBUG values
  y = 3 - y

  DEBUG CRSRXY, x, y, "*"                      ' Display asterisk

  DEBUG CRSRXY, 0, 10, "Press any Key..."      ' Wait for user
  DEBUGIN temp
  DEBUG CRSRXY, 0, 8, CLRDN                    ' Clear old info

LOOP
```

Animation and Redrawing the Background

Here is an example of something you can do if you use individual characters, but it won't work if you try to redraw the entire display with a **DEBUG** command.

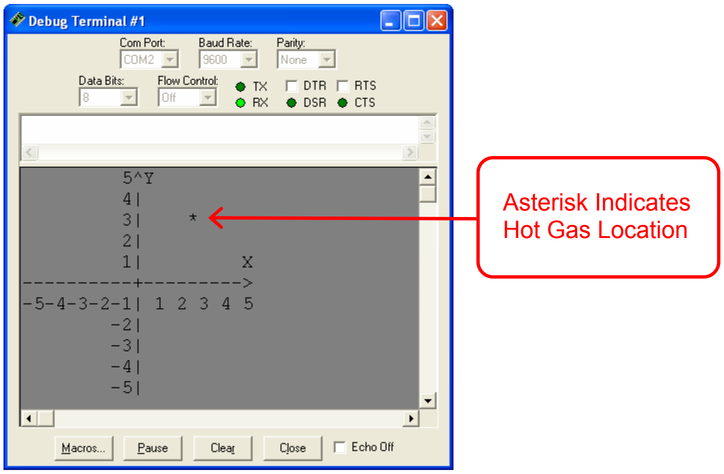
- √ Save EepromBackgroundRefresh.bs2 as ExampleAnimation.bs2
- √ Replace the main routine in the program with the one shown here.
- √ Run it and observe the effect.

```
DO
  FOR y = 0 TO 6
    FOR temp = 1 TO 2
      FOR x = 0 TO 12
        IF (temp.BIT0 = 1) THEN
          DEBUG CRSRXY, x, y, "*"
        ELSE
          index = (14 * yOld) + xOld + 1
          READ index, character
          DEBUG CRSRXY, xOld, yOld, character
          xOld = x
          yOld = y
        ENDIF
        PAUSE 50
      NEXT
    NEXT
  NEXT
LOOP
```

ACTIVITY #3: TILT THE BUBBLE GRAPH

This activity combines the graphics concepts introduced in Activity #1 and #2 with the accelerometer tilt measurement techniques introduced in Chapter 1. The result is an asterisk bubble that demonstrates the movement of the heated gas pocket inside the MX2125's chamber. Figure 9 shows what the Debug Terminal in this activity displays when the accelerometer is tilted up and to the left.

Figure 9 - Accelerometer Hot Gas Location



Debug Terminal #1

Com Port: COM2 Baud Rate: 9600 Parity: None

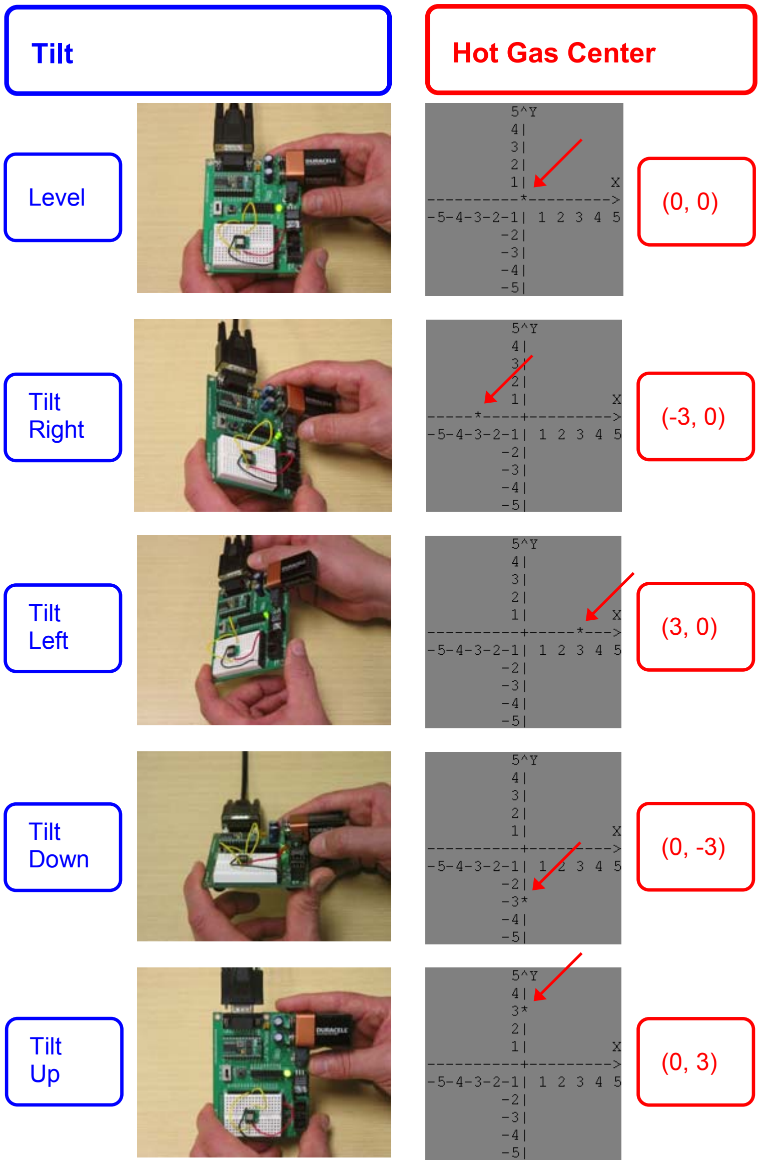
Data Bits: 8 Flow Control: Off

● TX ● RX □ DTR ● DSR □ RTS ● CTS

```
         5^Y
         4|
         3|    *
         2|
         1|         X
----------+--------->
-5-4-3-2-1| 1 2 3 4 5
        -2|
        -3|
        -4|
        -5|
```

Asterisk Indicates Hot Gas Location

Figure 10 shows a legend for the different ways you can tilt the board on its axes along with each tilt's effect on the location of the hot gas pocket.



```
         5^Y
         4|
         3|
         2|
         1|         X
----------*---------->
-5-4-3-2-1| 1 2 3 4 5
        -2|
        -3|
        -4|
        -5|
```

```
         5^Y
         4|
         3|
         2|
         1|         X
-----*----+---------->
-5-4-3-2-1| 1 2 3 4 5
        -2|
        -3|
        -4|
        -5|
```

```
         5^Y
         4|
         3|
         2|
         1|         X
----------+-----*---->
-5-4-3-2-1| 1 2 3 4 5
        -2|
        -3|
        -4|
        -5|
```

```
         5^Y
         4|
         3|
         2|
         1|         X
----------+---------->
-5-4-3-2-1| 1 2 3 4 5
        -2|
        -3*
        -4|
        -5|
```

```
         5^Y
         4|
         3*
         2|
         1|         X
----------+---------->
-5-4-3-2-1| 1 2 3 4 5
        -2|
        -3|
        -4|
        -5|
```

Figure 10

Tilt Control of Asterisk Display

BubbleGraph.bs2 updates the position of the hottest spot inside the accelerometer chamber about 8 times per second (8 Hz). After displaying the (background) XY axes to the debug terminal, it repeats the same steps over and over again.

- Display the background character and pause for the blink-effect.
- Get the X-axis tilt from the accelerometer
- Adjust the value so that it fits on the plot’s X-axis.
- Get the Y-axis tilt from the accelerometer
- Adjust the value so that it fits on the plot’s Y-axis.
- Display the asterisk and pause again for the blink-effect.

Each of these steps is discussed in more detail in the section that follows the example program.

Example Program – BubbleGraph.bs2

√ Enter and run BubbleGraph.bs2.

√ Hold your board as shown in the Tilt Asterisk Display figure.

√ Practice controlling the asterisk by tilting the board.

√ Aside from holding you board horizontally and tilting it, try holding it vertically and rotating it in a circle. The asterisk should travel in a circular arc around the graph as you do so.

```
' -----[ Title ]--------------------------------------------------------------
' Accelerometer Projects                     ' Program info
' BubbleGraph.bs2

'{$STAMP BS2}                                ' Stamp/PBASIC directives
'{$PBASIC 2.5}

' -----[ EEPROM Data ]--------------------------------------------------------
' Store background to EEPROM                 ' Address of last char on row
DATA CLS,                                    '   0
"         5^Y         ", CR,                 '  22
"         4|          ", CR,                 '  44
"         3|          ", CR,                 '  66
"         2|          ", CR,                 '  88
"         1|         X", CR,                 ' 110
"----------+--------->", CR,                 ' 132
"-5-4-3-2-1| 1 2 3 4 5", CR,                 ' 154
"        -2|          ", CR,                 ' 176
"        -3|          ", CR,                 ' 198
"        -4|          ", CR,                 ' 220
"        -5|          ", CR                  ' 242

' -----[ Variables ]----------------------------------------------------------
x       VAR    Word                          ' Store current position
y       VAR    Word

index   VAR     Word                         ' READ index/character storage
char    VAR     Byte

' -----[ Initialization ]-----------------------------------------------------
FOR index = 0 TO 242                         ' Read & display background
  READ index, char
  DEBUG char
NEXT

' -----[ Main Routine ]-------------------------------------------------------
DO                                           ' Begin main routine

  ' Replace asterisk with background character.
  index = (22 * y) + x + 1                   ' Coordinates -> EEPROM address
  READ index, char                           ' Get background character
  DEBUG CRSRXY, x, y, char                   ' Display background character
  PAUSE 50                                   ' Pause for blink effect

  ' Get X-axis tilt & scale to graph.
  PULSIN 6, 1, x                             ' Get X-axis tilt
  x = x MIN 1875 MAX 3125                    ' Keep inside X-axis domain
  x = x – 1875                               ' Offset to zero
  x = x * 2 / 125                            ' Scale

  ' Get Y-axis tilt & scale to graph.
  PULSIN 7, 1, y                             ' Get Y-Axis tilt
  y = y MIN 1875 MAX 3125                    ' Keep in Y-Axis range
  y = y – 1875                               ' Offset to zero
  y = y / 125                                ' Scale
  y = 10 – y                                 ' Offset Cartesian -> Debug

  ' Display asterisk at new cursor position.
  DEBUG CRSRXY, x, y, "*"                    ' Display asterisk
  PAUSE 50                                   ' Pause again for blink effect

LOOP                                         ' Repeat main routine
```

How BubbleGraph.bs2 Works

The first thing the main routine does is displays the background character at the current cursor position. With a 50 ms pause, it completes the “off” portion of a blinking asterisk. While the programs in Activity #2 had 14 characters per row, this larger plot has 22 characters per row. This value has to be multiplied by the **y** display coordinate, then added to the **x** display coordinate, plus one for the **CLS** at EEPROM address zero. The result stored in the **index** variable is the EEPROM address of the correct background character.

```
' Replace asterisk with background character.
index = (22 * y) + x + 1
READ index, char
DEBUG CRSRXY, x, y, char
PAUSE 50
```

The **PULSIN** command measures the X-axis measurement pulse the accelerometer sends to P6 and stores it in the **x** variable. **MIN** and **MAX** values are applied to **x** so that it doesn't cause the program to try to place the asterisk outside the plot area. Then, by subtracting 1875 from **x** causes the variable to range from 0 to 1250. Multiplying by 2 then dividing by 125 results in values ranging from 0 to 20, the number of characters across the X-axis on the plot.

```
' Get X-axis tilt & scale to graph.
PULSIN 6, 1, x
x = x MIN 1875 MAX 3125
x = x – 1875
x = x * 2 / 125
```

The **PULSIN** command measures the Y-axis measurement pulse the accelerometer sends to P7 and stores it in the **y** variable, and the **MIN** and **MAX** values are again applied to prevent the asterisk from wondering off the plot area. While the plot area is 20 spaces wide, it's only 10 spaces tall. This time, a measurement that ranges from 1875 to 3125 has to be mapped to a range of 10 to 0 (not 0 to 10). Dividing y by 125 gives a scale of 10, but we want the largest value to map to 0 carriage returns (Y = + 5) on the Debug terminal while the smallest value maps to 10 carriage returns down (Y = -5). That's what **y = 10 – y** does. When + 10 is substituted for **y** on the right side of the equal sign, the result on the left is 0. When 0 is substituted for **y** on the right side of the equal sign, the result on the left is 0. It works right for 1 through 9 too; give it a try.

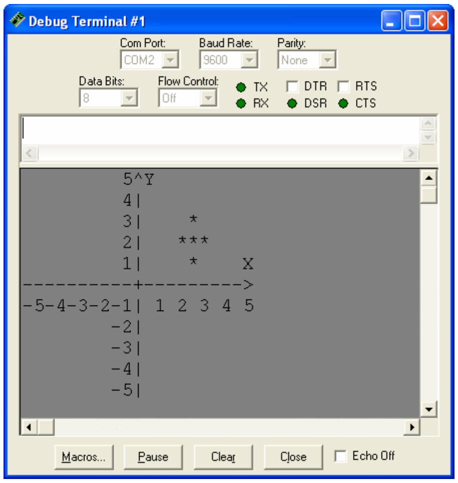
```
' Get Y-axis tilt & scale to graph.
PULSIN 7, 1, y
y = y MIN 1875 MAX 3125
y = y – 1875
y = y / 125
y = 10 – y
```

The last steps before repeating the loop in the main routine is to display the new asterisk at its new **x** and **y** coordinates, then pause for another 50 ms to complete the “on” portion of the blinking asterisk.

```
' Display asterisk at new cursor position.
DEBUG CRSRXY, x, y, "*"
PAUSE 50
```

Your Turn – A Larger Bubble

Displaying and erasing the group of asterisks shown in Figure 11 can be done, but compared to a single character, it’s a little tricky. The program has to ensure that none of the asterisks will be displayed outside the plot area. It also has to ensure that all of the asterisks will be overwritten with the correct characters from EEPROM.



Com Port: COM2
Baud Rate: 9600
Parity: None
Data Bits: 8
Flow Control: Off

```
         5^Y
         4|
         3|    *
         2|   ***
         1|    *    X
----------+--------->
-5-4-3-2-1| 1 2 3 4 5
        -2|
        -3|
        -4|
        -5|
```

Figure 11
Group of Asterisks with Background Refresh

Here is one example of how to modify BubbleGraph.bs2 so that it displays.

√ Save BubbleGraph.bs2 as BubbleGraphYourTurn.bs2.
√ Add this variable declaration to the program’s Variables section:

```
temp    VAR     Byte
```

√ Replace the “Replace asterisk with background character” routine with this:

```
' Replace asterisk with background character (modified).
```

```
FOR temp = (x MIN 1 – 1) TO (x MAX 19 + 1)
  index = (22 * y) + temp + 1
  READ index, char
  DEBUG CRSRXY, temp, y, char
NEXT

FOR temp = (y MIN 1 – 1) TO (y MAX 9 + 1)
  index = (22 * temp) + x + 1
  READ index, char
  DEBUG CRSRXY, x, temp, char
NEXT
PAUSE 50
```

√ Replace the “ Display asterisk at new cursor position” routine with this:

```
' Display asterisk at new cursor position (modified).
DEBUG CRSRXY, x,            y,            "*",
      CRSRXY, x MAX 19 + 1, y,            "*",
      CRSRXY, x,            y MAX 9 + 1,  "*",
      CRSRXY, x MIN 1 - 1,  y,            "*",
      CRSRXY, x,            y MIN 1 - 1,  "*"
PAUSE 50
```

√ Run the program and try it. Test to make sure problems do not occur as one of the outermost asterisks is forced off the plot area.

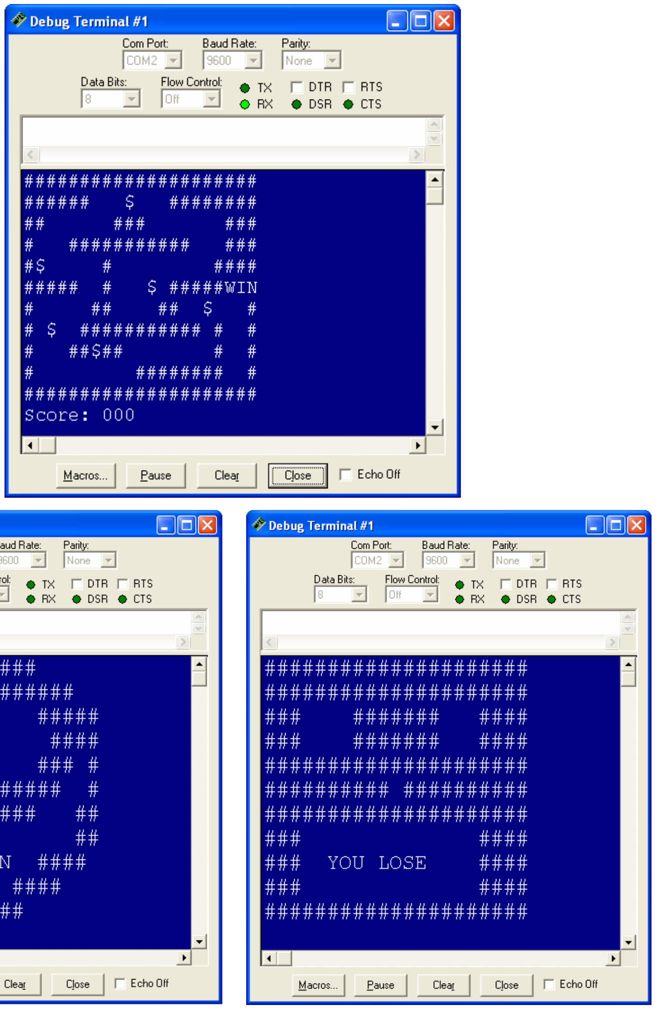
MIN and Negative Numbers

A twos complement "gotcha" to avoid is subtracting 1 from 0 and then setting the **`MIN`** value afterwards. Remember from Activity #1 that twos complement system stores the signed value -1 as 65535. That’s why the **`MIN`** value was set to 1 before subtracting 1. The result is then a correct minimum of 0. The same technique was used for setting the **`MAX`** values even though there really isn’t a problem with **`y + 1 MAX 10`**.

ACTIVITY #4: GAME CONTROL

Here are the rules of this Activity's tilt controlled game example, shown in Figure 12. Tilt your board to control the asterisk. If you get through the maze and place the asterisk on any of the "WIN" characters, the "YOU WIN" screen will display. If you bump into any of the pound signs "#" before you get to the end of the maze, the "YOU LOSE" screen is displayed. As you navigate the maze, try to move your asterisk game character through the dollar signs "$" to get more points.

Figure 12 - Obstacle Course Game



```
#####################
######   $   ########
##      ###       ###
#   ###########   ###
#$     #         ####
#####  #   $ #####WIN
#     ##    ##  $   #
# $  ########### #  #
#   ##$##        #  #
#         ########  #
#####################
Score: 000
```

```
     ###########
  #################
#####   #####   #####
####     ###     ####
# ###   #####   ### #
#  ###############  #
##   ###########   ##
##                 ##
 ####  YOU WIN  ####
   ####       ####
      #########
```

```
#####################
#####################
###    #######   ####
###    #######   ####
#####################
########## ##########
#####################
###              ####
###  YOU LOSE    ####
###              ####
#####################
```

Converting BubbleGraph.bs2 into TiltObstacleGame.bs2

TiltObstacleGame.bs2 is inarguably a hopped-up version of BubbleGraph.bs2. Here is a list of the main changes and additions:

- Change the graph into a maze.
- Add two backgrounds for win and lose to the EEPROM data.
- Give each background a Symbol name.
- Write a game player code block that detects which background character the game character is in front of and uses that information to enforce the rules of the game.

Try the game first, then we'll take a closer look at how it works.

Example Program – TiltObstacleGame.bs2

√ Enter, and save TiltObstacleGame.bs2.

√ Before you run the program, make sure your board is level. Also, make sure you are holding it the same way you did in Activity 3, with the breadboard is closest to you, and the serial cable is furthest away.

√ If you want to refresh the "$" characters, click your BASIC Stamp Editor's Run button. If you want to just practice navigating and not worry about points, press and release the Reset button on your board.

```
' -----[ Title ]--------------------------------------------------------------
' Accelerometer Projects                     ' Program info
' TiltObstacleGame.bs2

'{$STAMP BS2}                                ' Stamp/PBASIC directives
'{$PBASIC 2.5}

' -----[ EEPROM Data ]--------------------------------------------------------
' Store background to EEPROM                 ' 3 backgrounds used in game

Maze DATA @0, HOME,                          ' Maze background
"#####################", CR,
"######   $   ########", CR,
"##      ###       ###", CR,
"#   ###########   ###", CR,
"#$     #         ####", CR,
"#####  #   $ #####WIN", CR,
"#     ##    ##  $   #", CR,
"# $  ########### #  #", CR,
"#   ##$##        #  #", CR,
"#         ########  #", CR,
"#####################", CR

YouLose DATA @243, HOME,                     ' YouLose background
"#####################", CR,
"#####################", CR,
"###    #######   ####", CR,
"###    #######   ####", CR,
"#####################", CR,
"########## ##########", CR,
"#####################", CR,
"###              ####", CR,
"###  YOU LOSE    ####", CR,
"###              ####", CR,
"#####################", CR

YouWin DATA @486, HOME,                      ' YouWin background
"     ###########     ", CR,
"  #################  ", CR,
"#####   #####   #####", CR,
"####     ###     ####", CR,
"# ###   #####   ### #", CR,
"#  ###############  #", CR,
"##   ###########   ##", CR,
"##                 ##", CR,
" ####  YOU WIN  #### ", CR,
"   ####       ####   ", CR,
"      #########      ", CR

' -----[ Variables ]-----------------------------------------------------------
x         VAR     Word                       ' x & y tilts & graph coordinates
y         VAR     Word

index     VAR     Word                       ' EEPROM address and character
char      VAR     Byte

symbol    VAR     Word                       ' Symbol address for EEPROM DATA
points    VAR     Byte                       ' Points during game

' -----[ Initialization ]------------------------------------------------------
x = 10                                       ' Start game character in middle
y = 5

DEBUG CLS                                    ' Clear screen

' Display maze.
symbol = Maze                                ' Set Symbol to Maze EEPROM DATA

FOR index = 0 TO 242                         ' Display maze
  READ index + symbol, char
  DEBUG char
NEXT

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

  ' Display background at cursor position.
  index = (22 * y) + x + 1                   ' Coordinates -> EEPROM address
  READ index + symbol, char                  ' Get background character
  DEBUG CRSRXY, x, y, char                   ' Display background character
  PAUSE 50                                   ' Pause for blink effect

  ' Get X-axis tilt & scale to graph.
  PULSIN 6, 1, x                             ' Get X-axis tilt
  x = x MIN 1875 MAX 3125                    ' Keep inside X-axis domain
  x = x - 1875                               ' Offset to zero
  x = x * 2 / 125                            ' Scale

  ' Get Y-axis tilt & scale to graph.
  PULSIN 7, 1, y                             ' Get Y-Axis tilt
  y = y MIN 1875 MAX 3125                    ' Keep in Y-Axis range
  y = y - 1875                               ' Offset to zero
  y = y / 125                                ' Scale
  y = 10 - y                                 ' Offset Cartesian -> Debug

  ' Display asterisk at new position.
  DEBUG CRSRXY, x,  y, "*"                   ' Display asterisk
  PAUSE 50                                   ' Pause again for blink effect

  ' Display score
  DEBUG CRSRXY, 0, 11,                       ' Display points
        "Score: ", DEC3 points

  ' Did you move the asterisk over a $, W, I, N, or #?
  SELECT char                                ' Check background character
    CASE "$"                                 ' If "$"
      points = points + 10                   ' Add points
      WRITE index, "%"                       ' Write "%" over "$"
    CASE "#"                                 ' If "#", set Symbol to YouLose
      symbol = YouLose
    CASE "W", "I", "N"                       ' If W,I,orN, Symbol -> YouWin
      symbol = YouWin
  ENDSELECT

  ' This routine gets skipped while symbol is still = Maze.  If symbol
  ' was changed to YouWin or YouLose, display new background and end game.
  IF (symbol = YouWin) OR (symbol = YouLose) THEN
    FOR index = 0 TO 242                     ' 242 characters
      READ index + symbol, char              ' Get character
      DEBUG char                             ' Display character
    NEXT                                     ' Next iteration of loop
    END                                      ' End game
  ENDIF                                      ' End symbol-if code block

LOOP                                         ' Repeat main loop
```

How it Works – From BubbleGraph.bs2 to TiltObstacleGame.bs2

Two of the **DATA** directive's optional features were used here. Each of the three backgrounds was given a ***Symbol*** name, **Maze**, **YouWin**, and **YouLose**. These ***Symbol*** names make it easy for the program to select which background to display. The optional ***@Address*** operator was also used to set each directive's beginning EEPROM address. In BubbleGraph.bs2's background, the first character is **CLS** to clear the screen. The problem with **CLS** in these **DATA** directives is that it erases the entire Debug Terminal, including the score, which is displayed below the background. By substituting **HOME** for **CLS**, the entire backgrounds can be drawn and redrawn without erasing the score.

```
Maze DATA @0, HOME,
"#####################", CR,
"######   $   ########", CR,
   .
   .
   .
YouLose DATA @243, HOME,
"#####################", CR,
"#####################", CR,
   .
   .
   .
YouWin DATA @486, HOME,
"     ###########     ", CR,
"  #################  ", CR,
   .
   .
   .
```

Verifying Symbol Values

You can also try commands like **DEBUG DEC YouWin** to verify that YouWin stores the value 486.

Two variables are added, **symbol** to keep track of which background to retrieve characters from, and **points** to keep track of the player's score.

```
symbol    VAR     Word
points    VAR     Byte
```

The initial values of **x** and **y** have to start in the middle of the obstacle course. Since all variables initialize to zero in PBASIC, and that would cause the game character to start in the top-left corner, instead of in the middle.

```
x = 10
y = 5
```

The **symbol** variable is set to **Maze** before executing the **FOR…NEXT** loop that displays the background. Since all variables are initialized to zero in PBASIC, this happens anyhow. However, if you were to insert a **DATA** directive before the **Maze** background, it would be crucial to have this statement.

```
' Display maze.
symbol = Maze
```

The code block that follows the variable initialization is the background display. Look carefully at the **READ** command. It has been changed from **READ index, char** to **READ index + symbol, char**. Since the **symbol** variable was set to store **Maze**, all the characters in the first background will be displayed. If symbol stored **YouLose**, all the characters in the second background would be displayed. If it stored **YouWin**, all the characters in the third background would be displayed. Since either "You Lose" or "You Win" will have to be displayed, this routine will be used again later in the program.

```
FOR index = 0 TO 242
  READ index + symbol, char
  DEBUG char
NEXT
```

Three routines have to be added to the **DO...LOOP** in the main routine. The first simply displays the player's score:

```
  ' Display score
  DEBUG CRSRXY, 0, 11,                    ' Display points
        "Score: ", DEC3 points
```

The second routine is crucial; it's a **SELECT…CASE** statement that enforces the rules of the game. The **SELECT...CASE** statement looks at the character in the background at the asterisk's current location. If the asterisk is over a space **" "**, the **SELECT…CASE** statement doesn't need to change anything, so the main routine's **DO...LOOP** just keeps on repeating itself, checking the accelerometer measurements and updating the asterisk's location. If the asterisk is moved over a **"$"**, the program has to add 10 to the points variable, and write a **"%"** character over the **"$"** in EEPROM. This prevents the program from adding 10 points several times per second while the asterisk is held over the **"$"**. If the asterisk is moved over a **"#"**, the **YouLose** symbol is stored in the **symbol** variable.

If the asterisk moves over any one of the **"W" "I"** or **"N"** letters, **YouWin** is stored in the **symbol** variable.

```
' Did you move the asterisk over a $, W, I, N, or #?
SELECT char                              ' Check background character
  CASE "$"                               ' If "$"
    points = points + 10                 ' Add points
    WRITE index, "%"                     ' Write "%" over "$"
  CASE "#"                               ' If "#", set Symbol to YouLose
    symbol = YouLose
  CASE "W", "I", "N"                     ' If W,I,orN, Symbol -> YouWin
    symbol = YouWin
ENDSELECT
```

As you're navigating your asterisk over **" "**, **"$"**, or **"%"**, this next routine gets skipped because **symbol** still stores **Maze**. The **SELECT…CASE** statement only changes that when the asterisk was moved over **"#"**, **"W"**, **"I"**, or **"N"**. Whenever the **SELECT…CASE** statement changes **symbol** to either **YouWin** or **YouLose**, this routine displays the corresponding background, then ends the game.

```
' This routine gets skipped while symbol is still = Maze.  If symbol
' was changed to YouWin or YouLose, display new background and end game.
IF (symbol = YouWin) OR (symbol = YouLose) THEN
  FOR index = 0 TO 242                   ' 242 characters
    READ index + symbol, char           ' Get character
    DEBUG char                          ' Display character
  NEXT                                  ' Next iteration of loop
  END                                   ' End game
ENDIF                                  ' End symbol-if code block
```

Your Turn – Modifications and Bug Fixes

The game doesn't refresh the **"$"** symbols when you re-run it with the Board of Education's RESET button. It only works when you click the Run button on the BASIC Stamp Editor. That's because the **DATA** directive only writes to the EEPROM when the program is downloaded. If the program is restarted with the RESET button, the BASIC Stamp Editor doesn't get the chance to store the spaces, dollar signs, etc, so the percent signs that were written to EEPROM are still there. To fix the problem, all you have to do is check each character that gets read from EEPROM during the initialization. If that character turns bout to be a **"%"**, use the **WRITE** command to change it back to a **"$"**.

√ Save TiltObstacleGame.bs2 as TiltObstacleGameYourTurn.bs2

√ Modify the **FOR...NEXT** loop in the initialization that displays the maze like this:

```
FOR index = 0 TO 242                            ' Display maze
  READ index + symbol, char
  IF(char = "%") THEN                           ' <--- Add
    char = "$"                                  ' <--- Add
    WRITE index + symbol, char                  ' <--- Add
  ENDIF                                         ' <--- Add
  DEBUG char
NEXT
```

√ Verify that both the BASIC Stamp Editor's Run button and the Board of Education's Reset button both behave the same after this modification.

If the player rapidly changes the board's tilt, it is possible to jump over the **"#"** walls. There are two ways to fix this, one would be to add jumping animation and call it a "feature". Another way to fix it would be to only allow the asterisk to move by 1 character in either the X or Y directions. To fix this, the program will need to keep track of the previous position. This is a job for the **xOld** and **yOld** variables introduced in Activity #2.

√ Add these variable declarations to the Variables section in TiltObstacleGameYourTurn.bs2:

```
x          VAR     Word                         ' x & y tilts & coordinates
y          VAR     Word

xOld       VAR     Word                         ' <--- Add
yOld       VAR     Word                         ' <--- Add
```

√ Add initialization statements for **xOld** and **yOld**.

```
x    = 10                                       ' Start game char in middle
xOld = 10                                       ' <--- Add
y    = 5
yOld = 5                                        ' <--- Add
```

√ Modify the main routine so that it **x** can only be greater than or less than **xOld** by an increment or decrement of 1. Repeat for **y** and **yOld**.

```
y = 10 - y                                      ' Offset Cartesian -> Debug

IF (x > xOld) THEN x = xOld MAX 19 + 1          ' <--- Add
IF (x < xOld) THEN x = xOld MIN 1 - 1           ' <--- Add

IF (y > yOld) THEN y = yOld MAX 9 + 1           ' <--- Add
IF (y < yOld) THEN y = yOld MIN 1 - 1           ' <--- Add
```

```
' Display asterisk at new position.
DEBUG CRSRXY, x,  y, "*"                     ' Display asterisk
PAUSE 50                                     ' Pause again for blink
effect

xOld = x                                     ' <--- Add
yOld = y                                     ' <--- Add

' Display score
```

√ Run and test your modified program and verify that the asterisk can no longer skip **"#"** walls.

SUMMARY

Activity #1 introduced control characters, techniques for keeping characters inside a display's boundaries, and algebra for mapping coordinates to a display. Control character examples included **CRSRXY** and **CLRDN**. Display boundary examples included the **MIN** and **MAX** operators and an **IF…THEN** technique. Mapping techniques included simple PBASIC equations to change the values of X and Y-coordinates from Cartesian to their Debug Terminal equivalents.

Activity #2 introduced a means of storing, displaying and refreshing a background character display image from EEPROM. This is a useful ingredient for many product displays, and will also come in handy for tilt display and games. An entire display background can be printed with a **FOR…NEXT** loop. A **READ** command in the loop depends on the **FOR…NEXT** loop's index variable to address the next character in the sequence. After the **READ** command loads the next character in the variable, the **DEBUG** command can be used to send the character to the Debug Terminal. For erasing the tracks left by a character moving over the background, the character's previous position can be stored in one or more variables. The previous position information is then used along with the **READ** command to look up the character that should replace the moving character after it has moved to its next position.

Activity #3 demonstrated how the accelerometer measurements from Chapter 1 can be combined with cursor positioning and character recall techniques from Activity #1 in this chapter to create a tilt controlled display. Simple **PULSIN** measurements were used to measure the accelerometer's X and Y axis tilt. The tilt values were then scaled, offset, and displayed in the Debug Terminal as an asterisk over a Cartesian plane. The asterisk's position indicated the position of the hottest pocked of gas inside the MX2125's chamber, and as it moved, the background at its previous position was redrawn.

Activity #4 introduced tilt mode game control. The rules of simple games can be implemented with **SELECT...CASE** statements that use the character in the background at the location of the game character to decide what action to take next. Multiple backgrounds can be incorporated into a game by making use of the **DATA** directive's optional **@Address** operator and ***Symbol*** name. Since the ***Symbol*** name is actually the EEPROM address at the beginning of a given **DATA** directive, your program can access elements in different backgrounds by adding the value of ***Symbol*** make to the **READ** command's ***Address*** argument.

Questions

1. What does HID stand for?
2. What two arguments do you need along with **`DEBUG CRSRXY`** to place the cursor at a location in the Debug Terminal?
3. What control character clears the any printed characters that come after the cursor in the Debug Terminal?
4. Where is the Debug Terminal's transmit windowpane in relation to its receive windowpane?
5. What formatter stores a single digit that you type into the Debug Terminal's transmit windowpane in the **`X`** variable?
6. What operator can you use to make sure the value a variable stores does not exceed a maximum value?
7. Are there other coding techniques you can use other operators to prevent the value a variable stores from exceeding a maximum or minimum value?
8. What statements did CrsrXYPlot.bs2 use to convert Cartesian coordinates to Debug Terminal **`CRSRXY`** coordinates?
9. If the BASIC Stamp sends a negative value to the Debug Terminal, what can you say about the unsigned value of that number?
10. How does scale affect mapping Cartesian coordinates to the Debug Terminal?
11. What are the refresh rates of common CRT computer monitors?
12. Name two types of displays that do not need have all their pixels repeatedly refreshed by the BASIC Stamp?
13. What kind of routine do you need to display all the background characters stored in a **`DATA`** directive?
14. Why is it important to know how many background characters are in each row?
15. Why are word variables better for storing signed values?
16. What is the key to redrawing the background with the same variables used to store a character's current position?
17. When you tilt the accelerometer to the left, which way does the asterisk bubble travel?
18. If the coordinates of the asterisk moved from (0, 0) to (0, 3), which direction did you tilt it?
19. If the coordinates of the asterisk started at (-5,0), and ended at (5, 0), what do you think happened to the accelerometer?
20. If the coordinates of the asterisk started at (3, -3) and ended at (-3, 3) what tilt did the accelerometer start in, and what tilt did it end in?
21. Which axis was the fulcrum if the accelerometer started at (2, 2) and ended at (-2, 2)?

22. Here are four unusual coordinates for a single motion: (0, 5), (-5, 0), (0, -5), (5, 0). What motion can you perform on the accelerometer to cause it to report this sequence of coordinates?
23. If the accelerometer's readings travel from (0, 5) to (0, -5), then back again repeatedly, what two motion sequences are likely?
24. What's the beginning address of the **`YouLose`** background?
25. What's the value of **`YouWin`**?
26. In TiltObstacleGame.bs2, why were the control characters at the beginning of each background changed from **`CLS`** to **`HOME`**?
27. What command can you use to check the value of a **`DATA`** directive's **`Symbol`** name?
28. What's the difference between displaying the 23rd character in the **`YouLose`** EEPROM **`DATA`** and the 23rd character in **`YouWin`**?
29. If you change the **`Maze DATA`** directive's **`@Address`** operator from 0 to 10, what will you have to do to the other **`DATA`** directives in the program?
30. If you change the **`YouWin DATA`** directive's optional **`@Address`** operator from 486 to 500, what else in the program will you have to change?
31. In TiltObstacleGame.bs2, what kind of code block enforces the rules of the game?
32. What variable has to change for the game to end?
33. What command changes the **`"%"`** values back to **`"$"`** values in EEPROM?
34. How can you prevent the asterisk from skipping over the **`"#"`** wall?

Exercises

1. Write a **`DEBUG`** command that places the cursor five spaces over, seven space down, and then prints the message ***`"* this is the coordinate (5, 7) in the Debug Terminal"`***.
2. Write a **`DEBUG`** command that displays a Cartesian coordinate system from -2 to 2 on the X and Y axes.
3. Calculate the scale and offset for you will need for a coordinate system that goes from -2 to 2 on both the X and Y axes.
4. Write a **`DEBUG`** command that displays a Cartesian coordinates from -5 to 5 on the X and Y axes.
5. Calculate the scale and offset you will need for a coordinate system that goes from -5 to 5 on both the X and Y axes.
6. Write a routine that draws a line of + characters that extends from (1, 1) to (5, 5) in Cartesian coordinates.

7. Write a routine that draws a rectangle with asterisks. This routine should be 15 asterisks wide and 5 asterisks high.
8. Write a routine that makes a shape such as a rectangle, triangle or circle, then causes it to disappear one asterisk at a time.
9. If your background is 5 characters wide by 3 characters high, predict the minimum size variable you can use to set the address for your read command and explain your choice. Will you have any room for additional characters such as **`CLS`**?
10. Modify the background for a coordinate system from -3 to 3 on both the X and Y axes.
11. Modify the background display initialization for a coordinate system from -3 to 3 on both the X and Y axes.
12. Modify the scale and offset calculations for a -3 to 3 coordinate system.
13. Modify the scale and offset calculations so that the asterisk travels the same direction you tip the board instead of the opposite direction. When you tip the board left, the asterisk should go left, etc.
14. Modify the code block that adds to your score so that it gives you 100 points per **`"$"`**. Explain what else needs to be modified for the program to work properly.
15. Explain how to modify the program so that you can choose between three different mazes.
16. Explain what will happen to the program if you remove the ***`@Address`*** operators from the **`DATA`** directives.
17. Write a segment of code that remembers the highest score.

Projects

1. Modify CrsrXYPlot.bs2 so that it redraws the background before it plots the asterisk. The net effect should be that only one asterisk is visible at any given time. A better way of doing this is introduced in the next activity.
2. Modify PlotXYGraph.bs2 so that it displays the coordinates of the most recently placed asterisk to the right of the plot area.
3. Modify PlotXYGraph.bs2 so that it plots a line of asterisks from (-3, -3) to (3, 3).
4. repeats the line plot.
5. Modify PlotXYGraph.bs2 so that it plots a line of asterisks from from (3,-3) to (-3,3), then erases it, then repeats the line plot.
6. Modify PlotXYGraph.bs2 so that it works on a plot from -4 to 4 on both the X and Y axes.
7. Modify PlotXYGraph.bs2 so that it works on a plot from -2 to 2 on the Y axis in increments of 0.5 and from -4 to 4 on the X axis.

8. Write a program that allows you to move an asterisk around the Cartesian plane with the R, L, U, and D keys. Only one asterisk should appear on the plot at any given time.
9. Write a drawing program that allows you to select characters and draw them over the Cartesian plane. By pressing the enter key twice, the drawing disappears one character at a time.
10. Instead of a coordinate system from -5 to 5 on both axes, modify BubbleGraph.bs2 so that it functions on a coordinate system from -4 to 4.
11. Modify BubbleGraph.bs2 so that it allows you to hold your Board of Education (or BASIC Stamp Homework Board) so that you can read the writing on the board. The way the bubble behaves should be the same as it did in the original program.
12. Modify BubbleGraph.bs2 so that the cursor moves in the direction you tilt the board and test it.
13. Add a pushbutton circuit to the game, and modify the program so that you can use the pushbutton to toggle between different mazes.
14. Modify the program so that the "$" character earns you 10 points, and the "#" characters deduct 10 points. The game should start you with 20 points. If your score becomes negative, display "You Lose".
15. Create a 4 X 16 character version of this game. That's 4 characters high by 16 characters wide.
16. Rearrange the program so that the main routine calls subroutines for everything except executive decision making. That means subroutines have to handle accelerometer, measurements, cursor placement, and display updates.
17. Modify the game so that it displays a character in the direction you are traveling. Use "v", "<", ">", and "^". Add a pushbutton circuit that shoots an asterisk that makes a "#" disappear when it hits it.