PARALLAX INC

Web Site: www.parallax.com
Forums: forums.parallax.com
Sales: sales@parallax.com
Technical: support@parallax.com

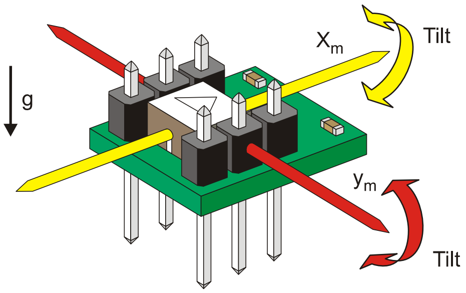
Office: (916) 624-8333
Fax: (916) 624-8003
Sales: (888) 512-1024
Tech Support: (888) 997-8267

Memsic 2125 Dual-Axis Accelerometer (#28017)

The Memsic 2125 is a low-cost thermal accelerometer capable of measuring tilt, collision, static and dynamic acceleration, rotation, and vibration with a range of ±3 g on two axes. Memsic provides the 2125 IC in a surface-mount format. Parallax mounts the circuit on a tiny PCB providing all I/O connections so it can easily be inserted on a breadboard or through-hole prototype area.

Features

- Measures ±3 g on each axis
- Simple pulse output of g-force for each axis
- Convenient 6-pin 0.1” spacing DIP module
- Analog output of temperature (TOut pin)
- Fully temperature compensated over 0 to 70 °C operating temperature range



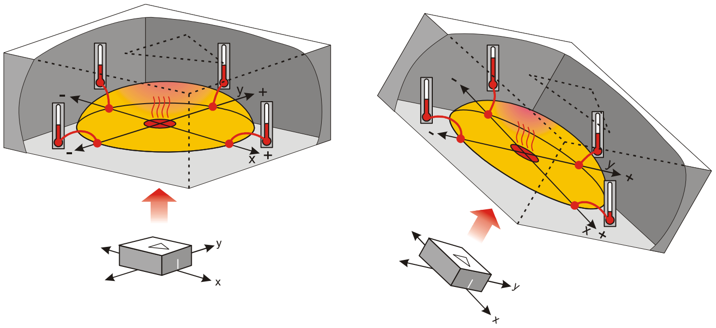
Key Specifications

- Power Requirements: 3.3 to 5 VDC; < 5 mA supply current
- Communication: TTL/CMOS compatible 100 Hz PWM output signal with duty cycle proportional to acceleration
- Dimensions: 0.42 x 0.42 x 0.45 in (10.7 x 10.7 x 11.8 mm)
- Operating temperature: 32 to 158 °F (0 to 70 °C)

Application Ideas

- Dual-axis tilt and acceleration sensing for autonomous robot navigation
- R/C tilt controller or autopilot
- Tilt-sensing Human Interface Device
- Motion/lack-of-motion sensor for alarm system
- Single-axis rotational angle and position sensing

Theory of Operation



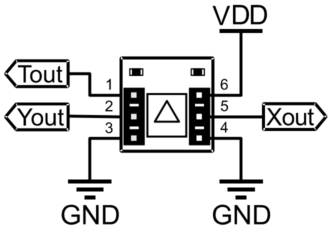
The MX2125 has a chamber of gas with a heating element in the center and four temperature sensors around its edge. When the accelerometer is level, the hot gas pocket rises to the top-center of the chamber, and all the sensors will measure the same temperature.

By tilting the accelerometer, the hot gas will collect closer to some of temperature sensors. By comparing the sensor temperatures, both static acceleration (gravity and tilt) and dynamic acceleration (like taking a ride in a car) can be detected. The MX2125 converts the temperature measurements into signals (pulse durations) that are easy for microcontrollers to measure and decipher.

Pin Definitions

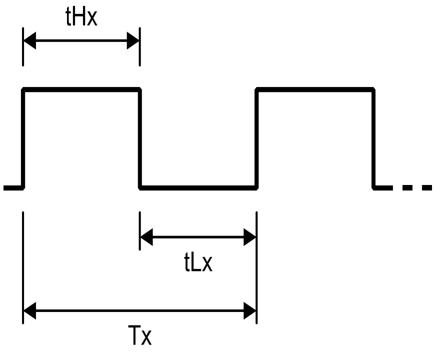
For Memsic MXD2125GL pin ratings, see the manufacturer's datasheet posted on the 28017 product page at www.parallax.com.

Pin	Name	Function
1	Tout	Temperature Out
2	Yout	Y-axis PWM output
3	GND	Ground -> 0 V
4	GND	Ground -> 0 V
5	Xout	X-axis PWM output
6	VDD	Input voltage: +3.3 to +5 VDC



Communication Protocol

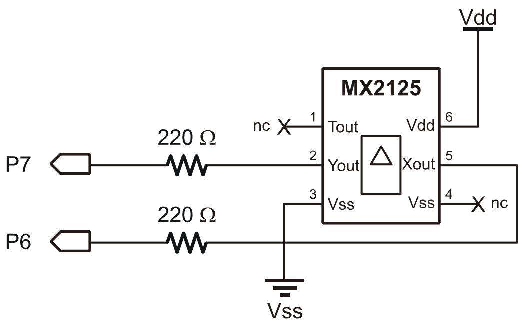
Each axis has a 100 Hz PWM duty cycle output in which acceleration is proportional to the ratio tHx/Tx. In practice, we have found that Tx is consistent so reliable results can be achieved by measuring only the duration of tHx. This is easy to accomplish with the BASIC Stamp PULSIN command or with the Propeller chip's counter modules.

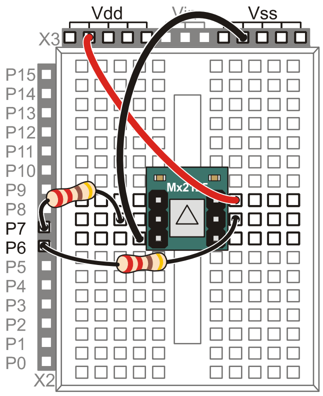


With Vdd = 5V, 50% duty cycle corresponds to 0 g, but this will vary with each individual unit within a range of 48.7% to 51.3%. This zero offset may be different when using Vdd = 3.3 V. See the manufacturer's datasheet for details.

Example Circuit

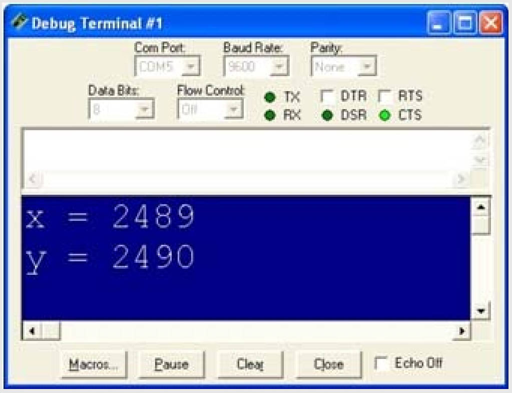
The example schematic and wiring diagram below are for the BASIC Stamp and Board of Education.





The program below, SimpleTilt.bs2, simply measures the pulse width, that is, the duration of tHx, for each axis. The raw values are displayed in the BASIC Stamp Editor's Debug Terminal. If you run the program, then tilt the accelerometer, you should see the values for each axis change.

```
' Smart Sensors and Applications - SimpleTilt.bs2
' Measure room temperature tilt.

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

x               VAR     Word
y               VAR     Word

DEBUG CLS

DO

  PULSIN 6, 1, x
  PULSIN 7, 1, y

  DEBUG HOME, DEC4 ? X, DEC4 ? Y

  PAUSE 100

LOOP
```



```
x = 2489
y = 2490
```

Programming Resources and Downloads

BASIC Stamp

- **Smart Sensors and Applications** — The BASIC Stamp example above is taken from the Stamps in Class text *Smart Sensors and Applications*, which features several chapters specific to the Memsic Dual-Axis Accelerometer. Topics include output scaling and offset, measuring vertical rotation, tilt-controlled video gaming basics, data logging g-force during a skateboard trick, and data logging acceleration on an RC car. The book and sample code can be downloaded from the 28029 product page at http://www.parallax.com

- **Boe-Bot Robot Projects with the Memsic 2125 Accelerometer** — The following projects with source code are posted under the Stamps in Class Mini-Projects sticky-thread in the Stamps in Class Forum at http://forums.parallax.com:
  - o Boe-Bot Robot Navigation with Accelerometer Incline Sensing
  - o A Tilt Radio Controller for Your Boe-Bot
- **The Memsic 2125 Demo Kit BASIC Stamp Source Code** — this source code contains conditional compile directives that allow it to be used with the BS2, BS2e, BS2sx, BS2p, and BS2pe.

Propeller Objects

Several Memsic 2125 Accelerometer code objects and applications for the Propeller chip are available in the Propeller Object Exchange (http://obex.parallax.com).

Below is a photograph of the high-speed Memsic MXD2125 Accelerometer Demo in action. This application “provides a high speed assembly driver, and separate-cog and same-cog Spin versions of the MXD2125 Dual Axis Accelerometer. The high speed version displays the data on a television as a 3D wireframe plane with normal vector.

599 Menlo Drive, Suite 100
Rocklin, California 95765, USA
Office: (916) 624-8333
Fax: (916) 624-8003

General: info@parallax.com
Technical: support@parallax.com
Web Site: www.parallax.com
Educational: www.stampsinclass.com

Memsic 2125 Accelerometer Demo Kit (#28017)

Acceleration, Tilt, and Rotation Measurement

Introduction

The Memsic 2125 is a low cost, dual-axis thermal accelerometer capable of measuring dynamic acceleration (vibration) and static acceleration (gravity) with a range of ±2 g. For integration into existing applications, the Memsic 2125 is electrically compatible with other popular accelerometers.

What kind of things can be done with the Memsic 2125 accelerometer? While there are many possibilities, here's a small list of ideas that can be realized with a Memsic 2125 and the Parallax BASIC Stamp® microcontroller:

- Dual-axis tilt sensing for autonomous robotics applications (BOE-Bot, Toddler, SumoBot)
- Single-axis rotational position sensing
- Movement/Lack-of-movement sensing for alarm systems

Packing List

Verify that your Memsic 2125 Demo Kit is complete in accordance with the list below:

- Parallax Memsic 2125 Demo PCB (uses Memsic MXD2125GL)
- Documentation

Note: Demonstration software files may be downloaded from www.parallax.com.

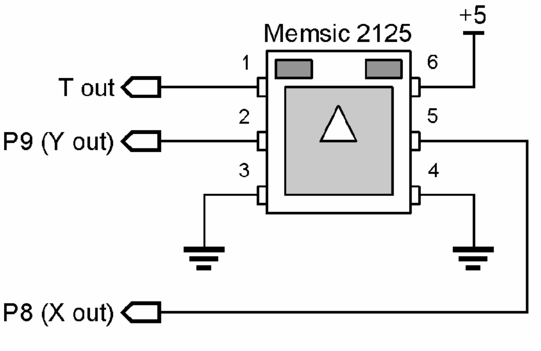
Features

- Measure 0 to ±2 g on either axis; less than 1 mg resolution
- Fully temperature compensated over 0° to 70° C range
- Simple, pulse output of g-force for X and Y axis – direct connection to BASIC Stamp
- Analog output of temperature (TOut pin)
- Low current operation: less than 4 mA at 5 vdc

Connections

Connecting the Memsic 2125 to the BASIC Stamp is a straightforward operation, requiring just two IO pins. If single-axis tilt of less than 60 degrees is your requirement, only one output from the Memsic 2125 need be connected. See Figure 1 for connection details.

Figure 1. Essential Memsic 2125 Connections

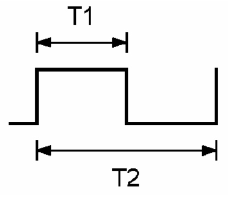


How It Works

Internally, the Memsic 2125 contains a small heater. This heater warms a "bubble" of air within the device. When gravitational forces act on this bubble it moves. This movement is detected by very sensitive thermopiles (temperature sensors) and the onboard electronics convert the bubble position [relative to g-forces] into pulse outputs for the X and Y axis.

The pulse outputs from the Memsic 2125 are set to a 50% duty cycle at 0 g. The duty cycle changes in proportion to acceleration and can be directly measured by the BASIC Stamp. Figure 2 shows the duty cycle output from the Memsic 2125 and the formula for calculating g force.

Figure 2. Memsic 2125 Pulse Output



$$A(g) = ((T1 / T2) - 0.5) / 12.5\%$$

The T2 duration is calibrated to 10 milliseconds at 25° C (room temperature). Knowing this, we can convert the formula to the following BASIC Stamp routine:

```
Read_X_Force:
  PULSIN Xin, HiPulse, xRaw
  xRaw = xRaw */ Scale
  xGForce = ((xRaw / 10) - 500) * 8
  RETURN
```

The T1 duration (Memsic output) is captured by PULSIN in the variable *xRaw*. Since each BASIC Stamp module has its own speed and will return a different raw value for the pulse, the factor called *Scale* (set by the compiler based on the BASIC Stamp module installed) is used to convert the raw output to microseconds. This will allow the program to operate properly with any BASIC Stamp 2-series module. At this point the standard equation provided by Memsic can be applied, adjusting the values to account for the pulse-width in microseconds. Fortunately, one divided by divided by 0.125 (12.5%) is eight, hence the final multiplication. The result is a signed value representing g-force in milli-g's (1/1000th g).

Experiments

Experiment 1: Dual-Axis Tilt Measurement

This experiment reads both axis values and displays the results in the DEBUG window. Calculations for g-force measurement and conversion to tilt were taken directly from Memsic documentation. Since the BASIC Stamp does not have an Arcsine function, it must be derived. Code for Arccosine and Arcsine are provided courtesy Tracy Allen, Ph.D.

```
' =========================================================================
'
'   File...... MEMSIC2125-Dual.BS2
'   Purpose... Memsic 2125 Accelerometer Dual-Axis Demo
'   Author.... (C) 2003-2004 Parallax, Inc -- All Rights Reserved
'   E-mail.... support@parallax.com
'   Started...
'   Updated... 07 SEP 2004
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =========================================================================


' -----[ Program Description ]---------------------------------------------
'
' Read the pulse outputs from a Memsic 2125 accelerometer and converts to
' G-force and tilt angle.
'
' g = ((t1 / 10 ms) - 0.5) / 12.5%
'
' Tilt = ARCSIN(g)
'
' Refer to Memsic documentation (AN-00MX-007.PDF) for details on g-to-tilt
' conversion and considerations.
'
' www.memsic.com


' -----[ Revision History ]------------------------------------------------


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

Xin             PIN     8                       ' X input from Memsic 2125
Yin             PIN     9                       ' Y input from Memsic 2125


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

' Set scale factor for PULSIN

#SELECT $STAMP
  #CASE BS2, BS2E
    Scale       CON     $200                    ' 2.0 us per unit
  #CASE BS2SX
    Scale       CON     $0CC                    ' 0.8 us per unit
  #CASE BS2P
```

```
    Scale           CON     $0C0                    ' 0.75 us per unit
  #CASE BS2PE
    Scale           CON     $1E1                    ' 1.88 us per unit
#ENDSELECT

HiPulse         CON     1                       ' measure high-going pulse
LoPulse         CON     0

DegSym          CON     176                     ' degrees symbol


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

xRaw            VAR     Word                    ' pulse from Memsic 2125
xmG             VAR     Word                    ' g force (1000ths)
xTilt           VAR     Word                    ' tilt angle

yRaw            VAR     Word
ymG             VAR     Word
yTilt           VAR     Word

disp            VAR     Byte                    ' displacement (0.0 - 0.99)
angle           VAR     Byte                    ' tilt angle


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


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

Setup:
  PAUSE 250                                     ' let DEBUG window open
  DEBUG "Memsic 2125 Accelerometer", CR,
        "-------------------------"


' -----[ Program Code ]----------------------------------------------------

Main:
  DO
    GOSUB Read_Tilt                             ' reads G-force and Tilt

    ' display results

    DEBUG CRSRXY, 0, 3
    DEBUG "X Input...  ",
          DEC (xRaw / 1000), ".", DEC3 xRaw, " ms",
          CLREOL, CR,
          "G Force... ", (xmG.BIT15 * 13 + " "),
          DEC (ABS xmG / 1000), ".", DEC3 (ABS xmG), " g",
          CLREOL, CR,
          "X Tilt.... ", (xTilt.BIT15 * 13 + " "),
          DEC ABS xTilt, DegSym, CLREOL

    DEBUG CRSRXY, 0, 7
    DEBUG "Y Input...  ",
          DEC (yRaw / 1000), ".", DEC3 yRaw, " ms",
          CLREOL, CR,
          "G Force... ", (ymG.BIT15 * 13 + " "),
          DEC (ABS ymG / 1000), ".", DEC3 (ABS ymG), " g",
          CLREOL, CR,
          "Y Tilt.... ", (yTilt.BIT15 * 13 + " "),
          DEC ABS yTilt, DegSym, CLREOL

    PAUSE 200                                   ' update about 5x/second
  LOOP
  END


' -----[ Subroutines ]-----------------------------------------------------

Read_G_Force:
  PULSIN Xin, HiPulse, xRaw                     ' read pulse output
  xRaw = xRaw */ Scale                          ' convert to uSecs
  xmG = ((xRaw / 10) - 500) * 8                 ' calc 1/1000 g
  PULSIN Yin, HiPulse, yRaw
  yRaw = yRaw */ Scale
  ymG = ((yRaw / 10) - 500) * 8
  RETURN


Read_Tilt:
  GOSUB Read_G_Force
  disp = ABS xmG / 10 MAX 99                    ' x displacement
  GOSUB Arcsine
  xTilt = angle * (-2 * xmG.BIT15 + 1)          ' fix sign
  disp = ABS ymG / 10 MAX 99                    ' y displacement
  GOSUB Arcsine
  yTilt = angle * (-2 * ymG.BIT15 + 1)          ' fix sign
  RETURN


' Trig routines courtesy Tracy Allen, PhD. (www.emesystems.com)

Arccosine:
  disp = disp */ 983 / 3                        ' normalize input to 127
  angle = 63 - (disp / 2)                       ' approximate angle
  DO                                            ' find angle
    IF (COS angle <= disp) THEN EXIT
    angle = angle + 1
  LOOP
  angle = angle */ 360                          ' convert brads to degrees
  RETURN


Arcsine:
  GOSUB Arccosine
  angle = 90 - angle
  RETURN
```

Experiment 2: Rotational Position Sensing

If the Memsic 2125 is tilted up on its edge (X axis), the X and Y outputs can be combined to measure rotational position. Output from this program is in Brads (binary radians, 0 to 255, the BASIC Stamp's unit of angular measurement) and degrees (0 to 359).

For this code to work, the Memsic 2125 PCB must be positioned such that the sensor is perpendicular to the ground.

```
' ==========================================================================
'
'   File...... MEMSIC2125-Rotation.BS2
'   Purpose... Memsic 2125 Accelerometer Rotational Angle Measurement
'   Author.... (C) 2003-2004 Parallax, Inc -- All Rights Reserved
'   E-mail.... support@parallax.com
'   Started...
'   Updated... 07 SEP 2004
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' ==========================================================================


' -----[ Program Description ]----------------------------------------------
'
' Read the pulse outputs from a Memsic 2125 accelerometer and combine to
' calculation rotational position.
'
' Refer to Memsic documentation (AN-00MX-007.PDF) for details on angle
' conversion and considerations.
'
' www.memsic.com


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

Xin             PIN     8                       ' X input from Memsic 2125
Yin             PIN     9                       ' Y input from Memsic 2125


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

' Set scale factor for PULSIN

#SELECT $STAMP
  #CASE BS2, BS2E
    Scale       CON     $200                    ' 2.0 us per unit
  #CASE BS2SX
    Scale       CON     $0CC                    ' 0.8 us per unit
  #CASE BS2P
    Scale       CON     $0C0                    ' 0.75 us per unit
  #CASE BS2PE
    Scale       CON     $1E1                    ' 1.88 us per unit
#ENDSELECT

HiPulse         CON     1                       ' measure high-going pulse
LoPulse         CON     0
```

```
DegSym          CON     176                     ' degrees symbol


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

pulse           VAR     Word                    ' pulse input
xmG             VAR     Word                    ' g force (1000ths)
ymG             VAR     Word
brads           VAR     Word                    ' binary radians
degrees         VAR     Word


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

Setup:
  DEBUG "Memsic 2125 Rotation", CR,
        "--------------------"


' -----[ Program Code ]----------------------------------------------------

Main:
  DO
    GOSUB Read_G_Force                          ' read X and Y

    brads = (xmG / 8) ATN (ymG / 8)             ' calculate angle
    degrees = brads */ 360                      ' convert to degrees

    DEBUG CRSRXY, 0, 3
    DEBUG "Axis   A(g)", CR,
          "X     ", (xmG.BIT15 * 13 + " "),
          DEC (ABS xmG / 1000), ".", DEC3 (ABS xmG), " g", CR,
          "Y     ", (ymG.BIT15 * 13 + " "),
          DEC (ABS ymG / 1000), ".", DEC3 (ABS ymG), " g", CR, CR,
          "Tilt = ", DEC3 brads, " Brads", CR,
          "       ", DEC3 degrees, " Degrees"

    PAUSE 200                                   ' update about 5x/second
  LOOP
  END


' -----[ Subroutines ]-----------------------------------------------------

Read_G_Force:
  PULSIN Xin, HiPulse, pulse                    ' read pulse output
  pulse = pulse */ Scale                        ' convert to uSecs
  xmG = ((pulse / 10) - 500) * 8                ' calc 1/1000 g
  PULSIN Yin, HiPulse, pulse
  pulse = pulse */ Scale
  ymG = ((pulse / 10) - 500) * 8
  RETURN
```

Experiment 3: Motion Detector

This experiment uses the Memsic 2125 as a movement or vibration detector. The program starts by reading the initial state of the sensor and storing these readings as calibration values. By doing this, the starting position of the sensor is nullified. The main loop of the program reads the sensor and compares the current outputs to the calibration values. If the output from either axis is greater than its calibration value the motion timer is incremented. If both fall below the thresholds motion timer is cleared. If the motion timer exceeds its threshold, the alarm will be turned on and will stay on until the BASIC Stamp is reset.

You can adjust the sensitivity (to motion/vibration) of the program by changing the **XLimit** and **YLimit** constants, as well as the **SampleDelay** constant (should be 100 ms or greater). The **AlarmLevel** constant determines how long motion/vibration must be present before triggering the alarm.

```
' =========================================================================
'
'   File...... MEMSIC2125-Motion.BS2
'   Purpose... Detects continuous motion for given period
'   Author.... Parallax (based on code by A. Chaturvedi of Memsic)
'   E-mail.... support@parallax.com
'   Started...
'   Updated... 15 JAN 2003
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =========================================================================


' -----[ Program Description ]---------------------------------------------
'
' Monitors X and Y inputs from Memsic 2125 and will trigger alarm if
' continuous motion is detected beyond the threshold period.


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

Xin             PIN     8                       ' X pulse input
Yin             PIN     9                       ' Y pulse input
ResetLED        PIN     10                      ' reset LED
AlarmLED        PIN     11                      ' alarm LED


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

HiPulse         CON     1                       ' measure high-going pulse
LoPulse         CON     0

SampleDelay     CON     500                     ' 0.5 sec
AlarmLevel      CON     5                       ' 5 x SampleDelay

XLimit          CON     5                       ' x motion max
YLimit          CON     5                       ' y motion max


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

```
xCal            VAR     Word                    ' x calibration value
yCal            VAR     Word                    ' y calibration value
xMove           VAR     Word                    ' x sample
yMove           VAR     Word                    ' y sample
xDiff           VAR     Word                    ' x axis difference
yDiff           VAR     Word                    ' y axis difference

moTimer         VAR     Word                    ' motion timer


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

Initialize:
  LOW AlarmLED                                  ' alarm off
  moTimer = 0                                   ' clear motion timer

Read_Cal_Values:
  PULSIN Xin, HiPulse, xCal                     ' read calibration values
  PULSIN Yin, HiPulse, yCal
  xCal = xCal / 10                              ' filter for noise & temp
  yCal = yCal / 10

  HIGH ResetLED                                 ' show reset complete
  PAUSE 1000
  LOW ResetLED


' -----[ Program Code ]----------------------------------------------------

Main:
  DO
    GOSUB Get_Data                              ' read inputs
    xDiff = ABS (xMove - xCal)                  ' check for motion
    yDiff = ABS (yMove - yCal)

    IF (xDiff > XLimit) OR (yDiff > YLimit) THEN
      moTimer = moTimer + 1                     ' update motion timer
      IF (moTimer > AlarmLevel) THEN Alarm_On
    ELSE
      moTimer = 0                               ' clear motion timer
    ENDIF
  LOOP
  END


' -----[ Subroutines ]-----------------------------------------------------

' Sample and filter inputs

Get_Data:
  PULSIN Xin, HiPulse, xMove                    ' take first reading
  PULSIN Yin, HiPulse, yMove
  xMove = xMove / 10                            ' filter for noise & temp
  yMove = yMove / 10
  PAUSE SampleDelay
  RETURN


' Blink Alarm LED
' -- will run until BASIC Stamp is reset
```

```
Alarm_On:
  DO
    TOGGLE AlarmLED                             ' blink alarm LED
    PAUSE 250
  LOOP                                          ' loop until reset
```

Application Idea

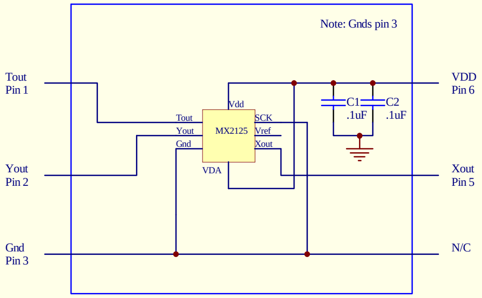
Using the tilt code from Experiment 1, you can create a 3D joystick by mounting the Memsic 2125 and a pushbutton in a small, spherical enclosure (like a tennis ball). With just three pins you can measure tilt of each axis and the status of the switch. This would make an interesting, intelligent "leash" for a Parallax BOE-Bot.

Using TOut

Since the Memsic 2125 is a thermal device, the temperature is available from the TOut pin and can be measured using an external analog to digital converter (i.e., LTC1298).

Details:

- Output calibrated to 1.25 volts @ 25.0° C
- Output change: 5 millivolts per degree C



Title		
Size B	Number	Revision
Date:	27-Sep-2004	Sheet of
File:	P:\Technical Support\Stephens Protel files (read only do not edit files of folders)\Mx2125 module\mx2125 board.ddb	Drawn By:

Improved, Ultra Low Noise ±3 *g* Dual Axis Accelerometer with Digital Outputs

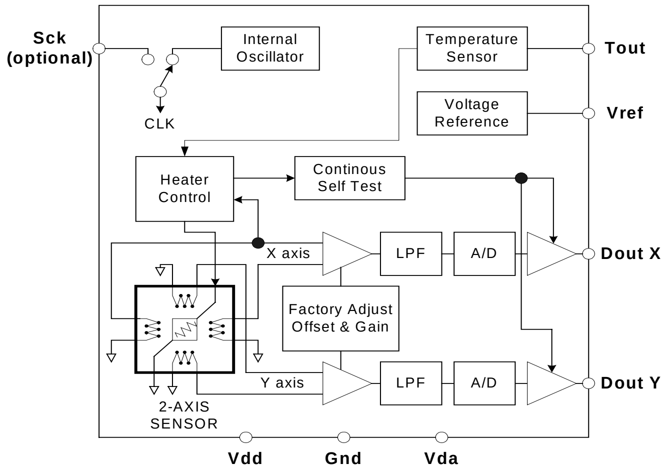
MXD2125G/H MXD2125M/N

FEATURES

Resolution better than 1 milli-***g***
Dual axis accelerometer fabricated on a monolithic CMOS IC
On chip mixed mode signal processing
No moving parts
50,000 ***g*** shock survival rating
17 Hz bandwidth expandable to >160 Hz
3.0V to 5.25V single supply continuous operation
Continuous self test
Independent axis programmability (special order)
Compensated for Sensitivity over temperature
Ultra low initial Zero-g Offset

APPLICATIONS

Automotive – Vehicle Security/Vehicle stability control/
Headlight Angle Control/Tilt Sensing
Security – Gas Line/Elevator/Fatigue Sensing
Information Appliances – Computer Peripherals/PDA's/Mouse
Smart Pens/Cell Phones
Gaming – Joystick/RF Interface/Menu Selection/Tilt Sensing
GPS — electronic Compass tilt Correction
Consumer – LCD projectors, pedometers, blood pressure
Monitor, digital cameras



MXD2125G/H/M/N FUNCTIONAL BLOCK DIAGRAM

GENERAL DESCRIPTION

The MXD2125G/H/M/N is a low cost, dual axis accelerometer fabricated on a standard, submicron CMOS process. It is a complete sensing system with on-chip mixed mode signal processing. The MXD2125G/H/M/N measures acceleration with a full-scale range of ±3 *g* and a sensitivity of 12.5%/g @5V at 25°C. It can measure both dynamic acceleration (e.g. vibration) and static acceleration (e.g. gravity).

The MXD2125G/H/M/N design is based on heat convection and requires no solid proof mass. This eliminates stiction and particle problems associated with competitive devices and provides shock survival of 50,000 *g*, leading to significantly lower failure rate and lower loss due to handling during assembly.

The MXD2125G/H/M/N provides two digital outputs that are set to 50% duty cycle at zero g acceleration. The outputs are digital with duty cycles (ratio of pulse width to period) that are proportional to acceleration. The duty cycle outputs can be directly interfaced to a micro-processor.

The typical noise floor is 0.2 m*g*/$\sqrt{Hz}$ allowing signals below 1 milli-*g* to be resolved at 1 Hz bandwidth. The MXD2125G/H/M/N is packaged in a hermetically sealed LCC surface mount package (5 mm x 5 mm x 2 mm height) and is operational over a -40°C to 105°C(M/N) and 0°C to 70°C(G/H) temperature range.

Information furnished by MEMSIC is believed to be accurate and reliable. However, no responsibility is assumed by MEMSIC for its use, nor for any infringements of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of MEMSIC.

©MEMSIC, Inc.
800 Turnpike St., Suite 202, North Andover, MA 01845
Tel: 978.738.0900 Fax: 978.738.0196
www.memsic.com

MXD2125G/H/M/N SPECIFICATIONS (Measurements @ 25°C, Acceleration = 0 *g* unless otherwise noted; V_{DD}, V_{DA} = 5.0V unless otherwise specified)

		MXD2125G/H			MXD2125M/N			
Parameter	**Conditions**	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**
SENSOR INPUT	Each Axis							
Measurement Range[1]		±3.0			±3.0			*g*
Nonlinearity	Best fit straight line		0.5			0.5		% of FS
Alignment Error[2]	X Sensor to Y Sensor		±1.0			±1.0		degrees
Transverse Sensitivity[3]			±2.0			±2.0		%
SENSITIVITY	Each Axis							
Sensitivity, Digital Outputs at pins		11.8	12.5	13.2	11.8	12.5	13.2	% duty cycle/g
D_{OUTX} and D_{OUTY}[4] Change over Temperature		-10		+8	-25		+8	%
ZERO *g* BIAS LEVEL	Each Axis							
0 *g* Offset[4]		-0.1	0.0	+0.1	-0.1	0.0	+0.1	*g*
0 *g* Duty Cycle[4]		48.7	50	51.3	48.7	50	51.3	% duty cycle
0 *g* Offset over Temperature	Δ from 25°C		±1.5			±1.5		m*g*/°C
	Based on 12.5%/g		±0.02			±0.02		%/°C
NOISE PERFORMANCE								
Noise Density, rms			0.2	0.4		0.2	0.4	$mg/\sqrt{Hz}$
FREQUENCY RESPONSE								
3dB Bandwidth		15	17	9	15	17	19	Hz
TEMPERATURE OUTPUT								
T_{out} Voltage		1.15	1.25	1.35	1.15	1.25	1.35	V
Sensitivity		4.6	5.0	5.4	4.6	5.0	5.4	mV/°K
VOLTAGE REFERENCE								
V_{Ref}	@3.0V-5.25V supply	2.4	2.5	2.65	2.4	2.5	2.65	V
Change over Temperature			0.1			0.1		mV/°C
Current Drive Capability	Source			100			100	µA
SELF TEST								
Continuous Voltage at D_{OUTX}, D_{OUTY} under Failure	@5.0V Supply, output rails to supply voltage		5.0			5.0		V
Continuous Voltage at D_{OUTX}, D_{OUTY} under Failure	@3.0V Supply, output rails to supply voltage		3.0			3.0		V
D_{OUTX} and D_{OUTY} OUTPUTS								
Normal Output Range	Output High	4.8			4.8			V
	Output Low			0.2			0.2	V
Output Frequency	MXD2125G/M	95	100	105	95	100	105	Hz
	MXD2125H/N	380	400	420	380	400	420	Hz
Current	Source or sink, @ 3.0V-5.25V supply			100			100	µA
Rise/Fall Time	3.0 to 5.25V supply	90	100	110	90	100	110	nS
Turn-On Time[5]	@5.0V Supply		160			160		mS
	@3.0V Supply		300			300		mS
POWER SUPPLY								
Operating Voltage Range		3.0		5.25	3.0		5.25	V
Supply Current	@ 5.0V	2.5	3.1	3.9	2.5	3.1	3.9	mA
Supply Current	@ 3.0V	3.0	3.8	4.6	3.0	3.8	4.6	mA
TEMPERATURE RANGE								
Operating Range		0		+70	-40		+105	°C

NOTES

[1] Guaranteed by measurement of initial offset and sensitivity.

[2] Alignment error is specified as the angle between the true and indicated axis of sensitivity.

[3] Transverse sensitivity is the algebraic sum of the alignment and the inherent sensitivity errors.

[4] The device operates over a 3.0V to 5.25V supply range. Please note that sensitivity and zero *g* bias level will be slightly different at 3.0V operation. For devices to be operated at 3.0V in production, they can be trimmed at the factory specifically for this lower supply voltage operation, in which case the sensitivity and zero *g* bias level specifications on this page will be met. Please contact the factory for specially trimmed devices for low supply voltage operation.

[5] Output settled to within ±17mg.

ABSOLUTE MAXIMUM RATINGS*

Supply Voltage (V_{DD}, V_{DA}) ………………...-0.5 to +7.0V
Storage Temperature ……..…………-65°C to +150°C
Acceleration …………………………………..50,000 *g*

*Stresses above those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only; the functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

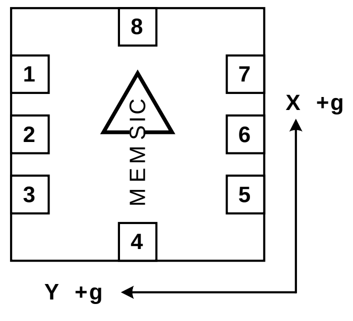
Pin Description: LCC-8 Package

Pin	Name	Description
1	T_{OUT}	Temperature (Analog Voltage)
2	D_{OUTY}	Y-Axis Acceleration Digital Signal
3	Gnd	Ground
4	V_{DA}	Analog Supply Voltage
5	D_{OUTX}	X-Axis Acceleration Digital Signal
6	V_{ref}	2.5V Reference
7	Sck	Optional External Clock
8	V_{DD}	Digital Supply Voltage

Ordering Guide

Model	**Package Style**	**Digital Output**	**Temperature Range**
MXD2125GL	LCC8 RoHS compliant	100 Hz	0 to 70°C
MXD2125GF	LCC8, Pb-free	100 Hz	0 to 70°C
MXD2125HL	LCC8 RoHS compliant	400Hz	0 to 70°C
MXD2125HF	LCC8, Pb-free	400Hz	0 to 70°C
MXD2125ML	LCC8 RoHS compliant	100 Hz	-40 to 105°
MXD2125MF	LCC8, Pb-free	100 Hz	-40 to 105°
MXD2125NL	LCC8 RoHS compliant	400 Hz	-40 to 105°
MXD2125NF	LCC8, Pb-free	400 Hz	-40 to 105°

All parts are shipped in tape and reel packaging.
Caution: ESD (electrostatic discharge) sensitive device.



Top View

Note: The MEMSIC logo's arrow indicates the +X sensing direction of the device. The +Y sensing direction is rotated 90° away from the +X direction following the right-hand rule. Small circle indicates pin one(1).

THEORY OF OPERATION

The MEMSIC device is a complete dual-axis acceleration measurement system fabricated on a monolithic CMOS IC process. The device operation is based on heat transfer by natural convection and operates like other accelerometers having a proof mass. The proof mass in the MEMSIC sensor is a gas.

A single heat source, centered in the silicon chip is suspended across a cavity. Equally spaced aluminum/polysilicon thermopiles (groups of thermocouples) are located equidistantly on all four sides of the heat source (dual axis). Under zero acceleration, a temperature gradient is symmetrical about the heat source, so that the temperature is the same at all four thermopiles, causing them to output the same voltage.

Acceleration in any direction will disturb the temperature profile, due to free convection heat transfer, causing it to be asymmetrical. The temperature, and hence voltage output of the four thermopiles will then be different. The differential voltage at the thermopile outputs is directly proportional to the acceleration. There are two identical acceleration signal paths on the accelerometer, one to measure acceleration in the x-axis and one to measure acceleration in the y-axis. Please visit the MEMSIC website at www.memsic.com for a picture/graphic description of the free convection heat transfer principle.

MXD2125G/H/M/N PIN DESCRIPTIONS

V_{DD} – This is the supply input for the digital circuits and the sensor heater in the accelerometer. The DC voltage should be between 3.0 and 5.25 volts. Refer to the section on PCB layout and fabrication suggestions for guidance on external parts and connections recommended.

V_{DA} – This is the power supply input for the analog amplifiers in the accelerometer. V_{DA} should always be connected to V_{DD}. Refer to the section on PCB layout and fabrication suggestions for guidance on external parts and connections recommended.

Gnd – This is the ground pin for the accelerometer.

D_{OUTX} – This pin is the digital output of the x-axis acceleration sensor. It is factory programmable to 100 Hz or 400 Hz. The user should ensure the load impedance is sufficiently high as to not source/sink >100µA typical. While the sensitivity of this axis has been programmed at the factory to be the same as the sensitivity for the y-axis, the accelerometer can be programmed for non-equal sensitivities on the x- and y-axes. Contact the factory for additional information.

D_{OUTY} – This pin is the digital output of the y-axis acceleration sensor. It is factory programmable to 100 Hz or 400 Hz. The user should ensure the load impedance is sufficiently high as to not source/sink >100µA typical. While the sensitivity of this axis has been programmed at the factory to be the same as the sensitivity for the x-axis, the accelerometer can be programmed for non-equal sensitivities on the x- and y-axes. Contact the factory for additional information.

T_{OUT} – This pin is the buffered output of the temperature sensor. The analog voltage at T_{OUT} is an indication of the die temperature. This voltage is useful as a differential measurement of temperature from ambient and not as an absolute measurement of temperature.

Sck – The standard product is delivered with an internal clock option (800kHz). **This pin should be grounded when operating with the internal clock.** An external clock option can be special ordered from the factory allowing the user to input a clock signal between 400kHz And 1.6MHz

V_{ref} – A reference voltage is available from this pin. It is set at 2.50V typical and has 100µA of drive capability.

DISCUSSION OF TILT APPLICATIONS AND RESOLUTION

Tilt Applications: One of the most popular applications of the MEMSIC accelerometer product line is in tilt/inclination measurement. An accelerometer uses the force of gravity as an input to determine the inclination angle of an object.

A MEMSIC accelerometer is most sensitive to changes in position, or tilt, when the accelerometer's sensitive axis is perpendicular to the force of gravity, or parallel to the Earth's surface. Similarly, when the accelerometer's axis is parallel to the force of gravity (perpendicular to the Earth's surface), it is least sensitive to changes in tilt.

Table 1 and Figure 2 help illustrate the output changes in the X- and Y-axes as the unit is tilted from +90° to 0°. Notice that when one axis has a small change in output per degree of tilt (in m*g*), the second axis has a large change in output per degree of tilt. The complementary nature of these two signals permits low cost accurate tilt sensing to be achieved with the MEMSIC device (reference application note AN-00MX-007).

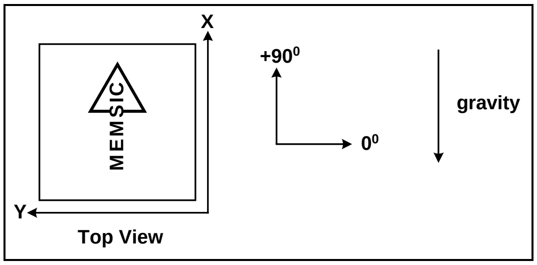


Figure 2: Accelerometer Position Relative to Gravity

	X-Axis		Y-Axis	
X-Axis Orientation To Earth's Surface (deg.)	X Output (*g*)	Change per deg. of tilt (m*g*)	Y Output (*g*)	Change per deg. of tilt (m*g*)
90	1.000	0.15	0.000	17.45
85	0.996	1.37	0.087	17.37
80	0.985	2.88	0.174	17.16
70	0.940	5.86	0.342	16.35
60	0.866	8.59	0.500	15.04
45	0.707	12.23	0.707	12.23
30	0.500	15.04	0.866	8.59
20	0.342	16.35	0.940	5.86
10	0.174	17.16	0.985	2.88
5	0.087	17.37	0.996	1.37
0	0.000	17.45	1.000	0.15

Table 1: Changes in Tilt for X- and Y-Axes

Resolution: The accelerometer resolution is limited by noise. The output noise will vary with the measurement bandwidth. With the reduction of the bandwidth, by applying an external low pass filter, the output noise drops. Reduction of bandwidth will improve the signal to noise ratio and the resolution. The output noise scales directly with the square root of the measurement bandwidth. The maximum amplitude of the noise, its peak- to- peak value, approximately defines the worst case resolution of the measurement. With a simple RC low pass filter, the rms noise is calculated as follows:

Noise (mg rms) = Noise(mg/$\sqrt{Hz}$) * $\sqrt{(Bandwidth(Hz)*1.6)}$

The peak-to-peak noise is approximately equal to 6.6 times the rms value (for an average uncertainty of 0.1%).

DIGITAL INTERFACE

The MXD2125G/H/M/N is easily interfaced with low cost microcontrollers. For the digital output accelerometer, one digital input port is required to read one accelerometer output. For the analog output accelerometer, many low cost microcontrollers are available today that feature integrated A/D (analog to digital converters) with resolutions ranging from 8 to 12 bits.

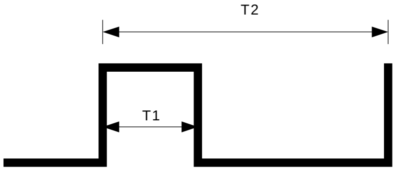
In many applications the microcontroller provides an effective approach for the temperature compensation of the sensitivity and the zero *g* offset. Specific code set, reference designs, and applications notes are available from the factory. The following parameters must be considered in a digital interface:

Resolution: smallest detectable change in input acceleration
Bandwidth: detectable accelerations in a given period of time
Acquisition Time: the duration of the measurement of the acceleration signal

DUTY CYCLE DEFINITION

The MXD2125G/H/M/N has two PWM duty cycle outputs (x,y). The acceleration is proportional to the ratio T1/T2. The zero *g* output is set to 50% duty cycle and the sensitivity scale factor is set to 12.5% duty cycle change per *g*. These nominal values are affected by the initial tolerance of the device including zero *g* offset error and sensitivity error. This device is offered from the factory programmed to either a 10ms period (100 Hz) or a 2.5ms period (400Hz).

T1	Length of the "on" portion of the cycle.
T2 (Period)	Length of the total cycle.
Duty Cycle	Ratio of the "0n" time (T1) of the cycle to the total cycle (T2). Defined as T1/T2.
Pulse width	Time period of the "on" pulse. Defined as T1.



A (g)= (T1/T2 - 0.5)/12.5%
0g = 50% Duty Cycle
T2= 2.5ms or 10ms (factory programmable)

Figure 3: Typical output Duty C ycle

CHOOSING T2 AND COUNTER FREQUENCY DESIGN TRADE-OFFS

The noise level is one determinant of accelerometer resolution. The second relates to the measurement resolution of the counter when decoding the duty cycle output. The actual resolution of the acceleration signal is limited by the time resolution of the counting devices used to decode the duty cycle. The faster the counter clock, the higher the resolution of the duty cycle and the shorter the T2 period can be for a given resolution. Table 2 shows some of the trade-offs. It is important to note that this is the resolution due to the microprocessors' counter. It is probable that the accelerometer's noise floor may set the lower limit on the resolution.

T2 (ms)	MEMSIC Sample Rate	Counter-Clock Rate (MHz)	Counts Per T2 Cycle	Counts per *g*	Reso-lution (m*g*)
2.5	400	2.0	5000	625	1.6
2.5	400	1.0	2500	312.5	3.2
2.5	400	0.5	1250	156.3	6.4
10.0	100	2.0	20000	2500	0.4
10.0	100	1.0	10000	1250	0.8
10.0	100	0.5	5000	625	1.6

Table 2: Trade-Offs Between Microcontroller Counter Rate and T2 Period.

CONVERTING THE DIGITAL OUTPUT TO AN ANALOG OUTPUT

The PWM output can be easily converted into an analog output by integration. A simple RC filter can do the conversion. Note that that the impedance of the circuit following the integrator must be much higher than the impedance of the RC filter. Reference figure 4 for an example.

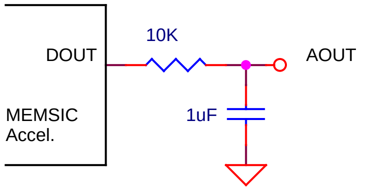


Figure 4: Converting the digital output to an analog voltage

POWER SUPPLY NOISE REJECTION

Two capacitors and a resistor are recommended for best rejection of power supply noise (reference Figure 5 below). The capacitors should be located as close as possible to the device supply pins (V_{DA}, V_{DD}). The capacitor lead length should be as short as possible, and surface mount capacitors are preferred. For typical applications, capacitors C1 and C2 can be ceramic 0.1 µF, and the resistor R can be 10 Ω.

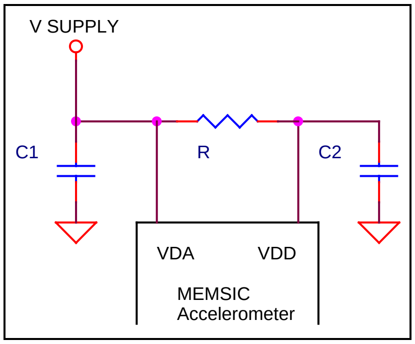
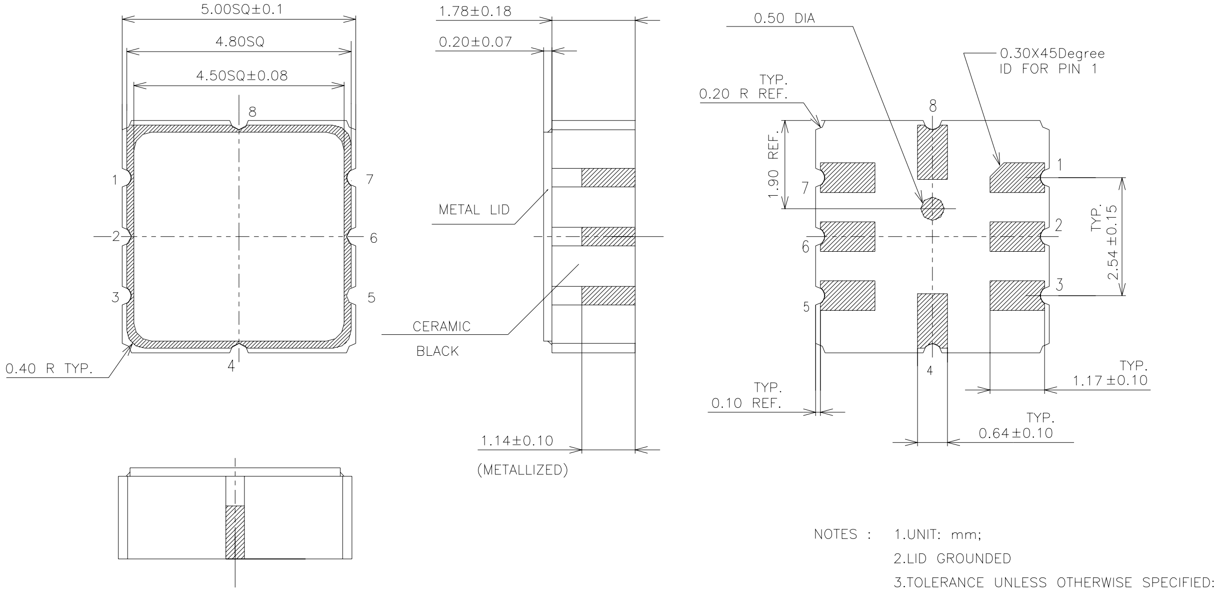


Figure 5: Power Supply Noise Rejection

PCB LAYOUT AND FABRICATION SUGGESTIONS

1. The Sck pin should be grounded to minimize noise.
2. Liberal use of ceramic bypass capacitors is recommended.
3. Robust low inductance ground wiring should be used.
4. Care should be taken to ensure there is “thermal symmetry” on the PCB immediately surrounding the MEMSIC device and that there is no significant heat source nearby.
5. A metal ground plane should be added directly beneath the MEMSIC device. The size of the plane should be similar to the MEMSIC device’s footprint and be as thick as possible.
6. Vias can be added symmetrically around the ground plane. Vias increase thermal isolation of the device from the rest of the PCB.

LCC-8 PACKAGE DRAWING



NOTES : 1.UNIT: mm;
2.LID GROUNDED
3.TOLERANCE UNLESS OTHERWISE SPECIFIED:
WITHIN +/−0.20

Fig 6: Hermetically Sealed Package Outline

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.
Last revised on 4/24/05.
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.

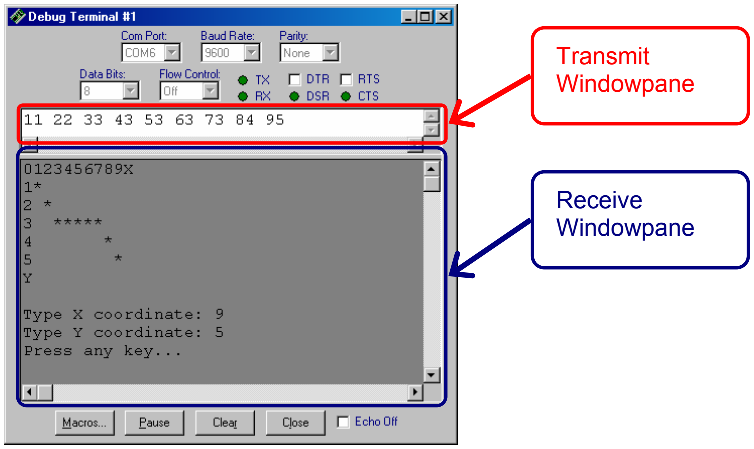
More Control Characters

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



```
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`**.

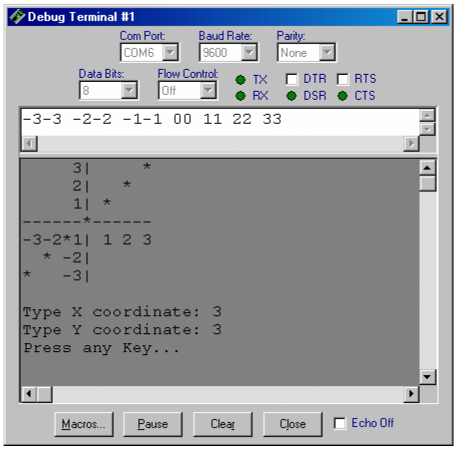
√ Try entering out of bounds values for the Y coordinate (0 and 6 to 9) and 0 for the X coordinate.
√ Observe the effects on the display's background.
√ Modify CrsrxyPlot.bs2 as shown here and try it again

```
  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.



```
-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:

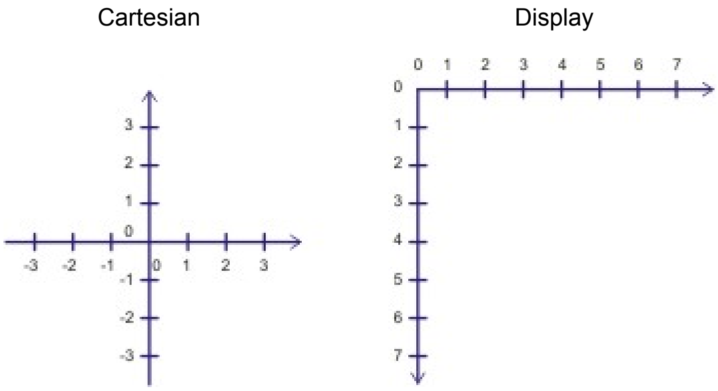
```
' Accelerometer Projects
' TwosComplementExample.bs2

' {$STAMP BS2}
' {$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.


Cartesian

Display

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.

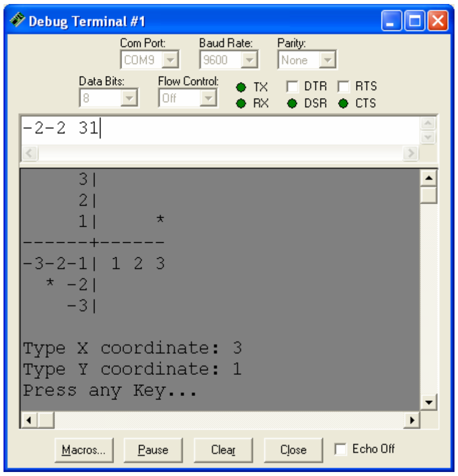
```
DEBUG "Type X coordinate: "
DEBUGIN SDEC1 x
DEBUG CR, "Type Y coordinate: "
DEBUGIN SDEC1 y

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.

$$12 = (2 \times 3) + C$$

(6) Solve for the second unknown.

$$12 = (2 \times 3) + C$$
$$12 = 6 + C$$
$$C = 12\text{-}6$$

$$C = 6$$

(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.

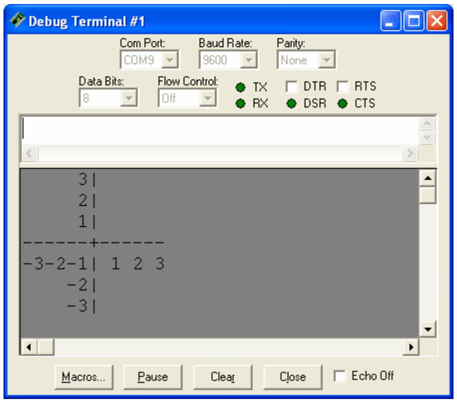
When used with displays that do not need to be refreshed (like the Debug Terminal or an LCD), the BASIC Stamp's can store an image of a game or graph background in its EEPROM. When a game character moves and is redrawn at a different location, the BASIC Stamp can just redraw the background characters 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.

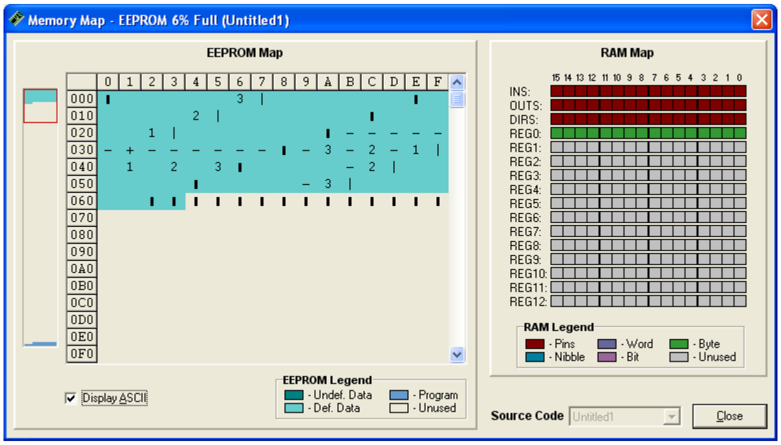
```
' Accelerometer Projects                     ' Program
' EepromBackgroundDisplay.bs2

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



```
     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
```

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:

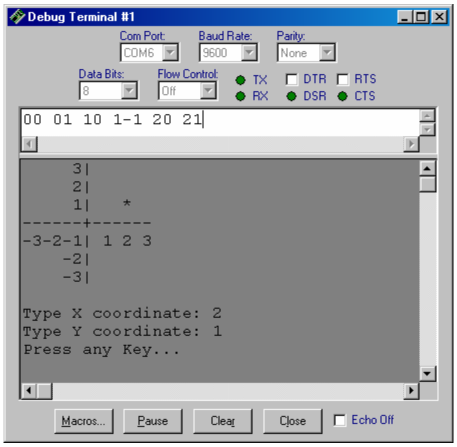
(3) Before loading new values into the x and y variables, store the current value of **x** into **xOld** and the current value of **y** into **yOld**. Keep in mind that both values will be in terms of Debug Terminal coordinates. Also keep in mind that the first time through, the old coordinates will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

● TX □ DTR □ RTS

● RX ● DSR ● CTS

□ Echo 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.

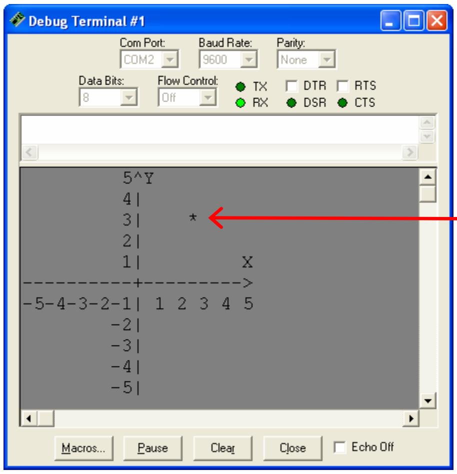
√ Save EepromBackgroundRefresh.bs2 as ExampleAnimation.bs2
√ Replace the main routine in the program with the one shown here.
√ Run it and observe the effect.

```
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

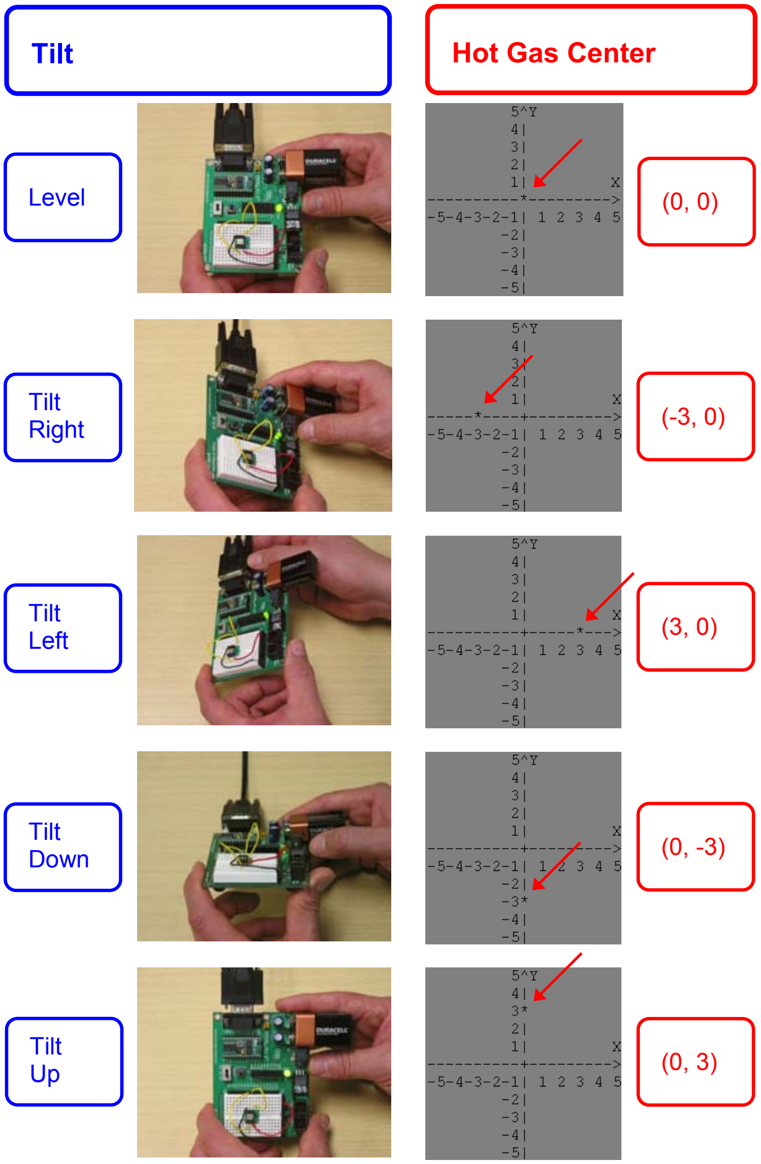


Com Port: COM2 Baud Rate: 9600 Parity: None
Data Bits: 8 Flow Control: Off

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

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.

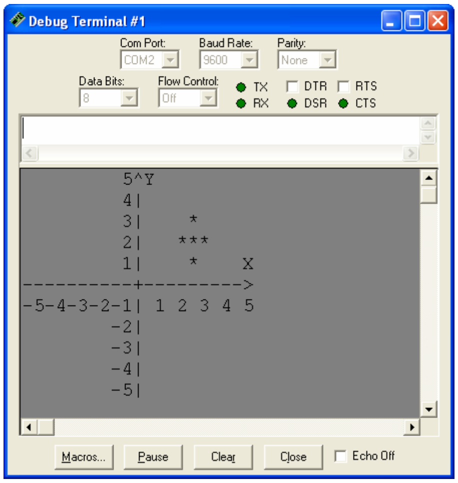
```
' Get Y-axis tilt & scale to graph.
PULSIN 7, 1, y
y = y MIN 1875 MAX 3125
y = y – 1875
y = y / 125
y = 10 – y
```

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
□ DTR □ RTS
● DSR ● CTS

```
         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.

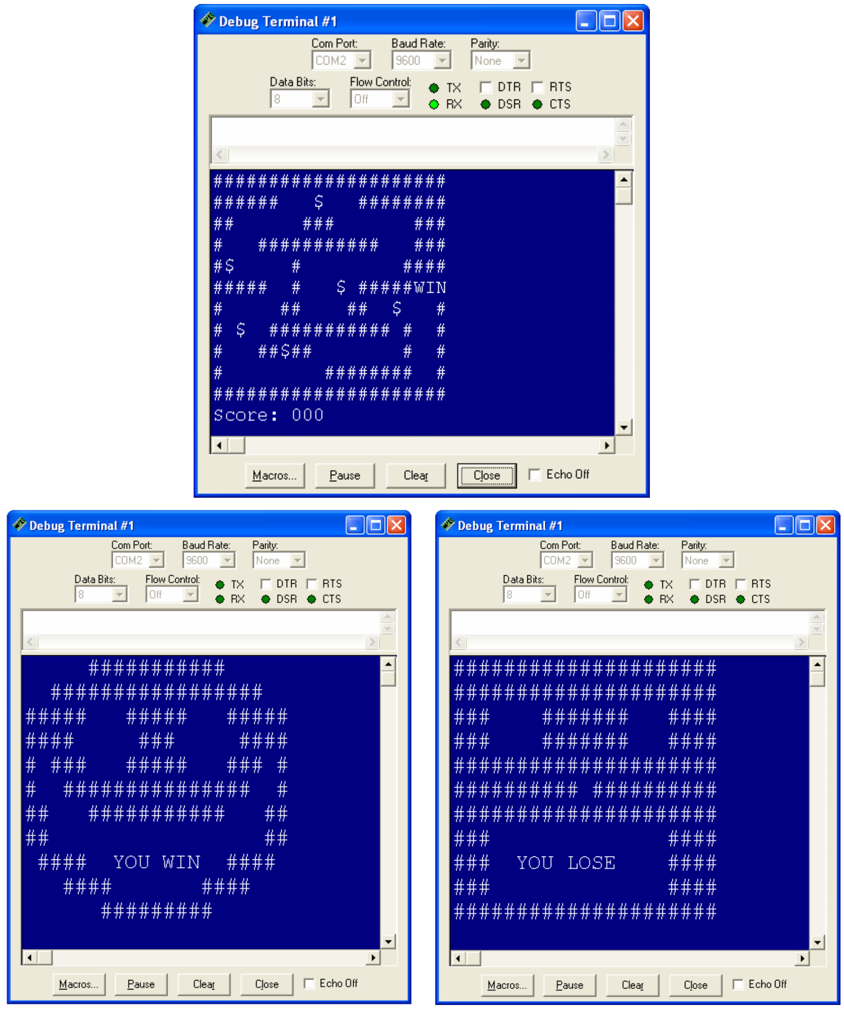
MIN and Negative Numbers

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



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

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

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

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

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

```
#####################
#####################
###    #######   ####
###    #######   ####
#####################
########## ##########
#####################
###              ####
###  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.

Accelerometer - Getting Started

Acceleration is a measure of how quickly speed changes. Just as a speedometer is a meter that measures speed, an accelerometer is a meter that measures acceleration. You can use an accelerometer's ability to sense acceleration to measure a variety of things that are very useful to electronic and robotic projects and designs:

- Acceleration
- Tilt and tilt angle
- Incline
- Rotation
- Vibration
- Collision
- Gravity

Accelerometers are already used in a wide variety of machines, specialized equipment and personal electronics. Here are just a few examples:

- Self balancing robots
- Tilt-mode game controllers
- Model airplane auto pilot
- Car alarm systems
- Crash detection/airbag deployment
- Human motion monitoring
- Leveling tool

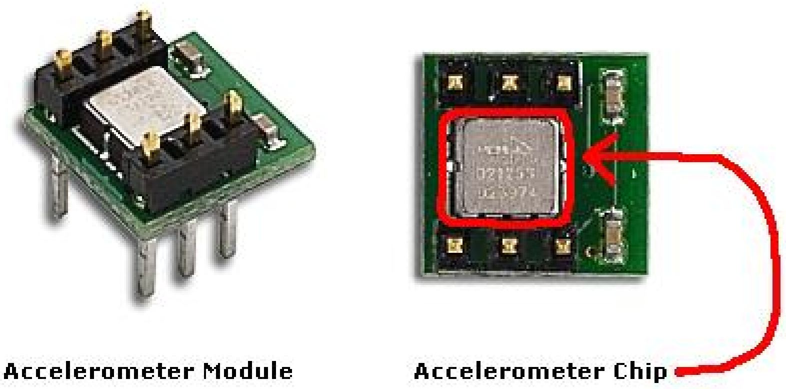
Once upon a time, accelerometers were large, clunky and expensive instruments that did not lend themselves to electronic and robotic projects. This all changed thanks to the advent of MEMS, micro-electro-mechanical-systems. MEMS technology is responsible for an ever increasing number of formerly mechanical devices designed right onto silicon chips.

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=55816.
Software for BASIC Stamp® Modules and applications are available for free download from www.parallax.com.

The accelerometer you will be working with in the forthcoming activities is the Parallax Memsic 2125 Dual Axis Accelerometer module shown in Figure 1. This module measures less than $^1/_2$" X $^1/_2$" X $^1/_2$", and the accelerometer chip itself is less than $^1/_4$" X $^1/_4$" X $^1/_8$".

Figure 1 - Accelerometer Module and MX2125 Chip



People naturally sense acceleration on three axes, forward/backward, left/right and up/down. Just think about the last time you were in the passenger seat of a car on a hilly and curvy road. Forward/backward acceleration is the sensation of speeding up and slowing down. Left/right acceleration involved making turns, and up down acceleration is what you felt going over hills.

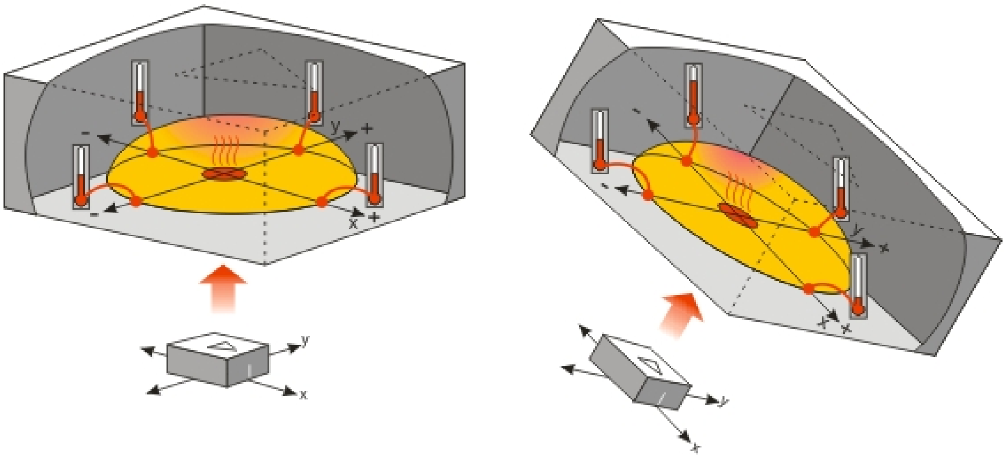
Up/down acceleration is also the way we sense gravity. When on the ground, people tend to sense gravity as their own weight. In free-fall, they sense gravity as weightlessness. In physics terms, gravity is a form of acceleration. When an object is on the ground, gravity is sometimes called static acceleration. When an object is rolling down hill or falling, gravity becomes dynamic acceleration.

Instead of the three axes people sense, the MX2125 accelerometer senses acceleration on two axes. The acceleration it senses depends on how it's positioned. By holding it one way, it can sense forward/backward and left/right. If you hold it a different way, it can sense up/down and forward/backward. Two axes of acceleration is enough for many of the applications listed earlier. However, you can always mount and monitor a second accelerometer to capture that third axis.

THE MX2125 ACCELEROMETER – HOW IT WORKS

The MX2125's design is amazingly simple. It has a chamber of gas with a heating element in the center and four temperature sensors around its edge. Just as hot air rises and cooler air sinks, the same applies to hot and cool gasses. If you hold the accelerometer still, all it senses is gravity, and tilting it gives us an example of how it senses static acceleration. When you hold the accelerometer level, the hot gas pocket is rises to the top-center of the accelerometer's chamber, and all the temperature sensors measure the same temperature. Depending on how you tilt the accelerometer, the hot gas will collect closer to one or maybe two of the temperature sensors.

Figure 2 - Accelerometer Heated Gas Pocket



Both static acceleration (gravity and tilt) and dynamic acceleration (like taking a ride in a car) are detected by the temperature sensors. If you take the accelerometer for a car ride, the hotter and cooler gasses slosh around in the chamber in a manner similar to a container that is partially filled with water.

In most situations, making sense out of these measurements is a simple task thanks to the electronics inside the MX2125. The MX2125 converts the temperature measurements into signals (pulse durations) that are easy for the BASIC Stamp module to measure and decipher.

ACTIVITY #1: CONNECTING AND TILT-TESTING THE MX2125

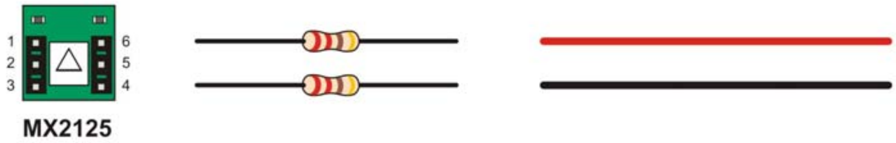
In this activity, you will connect the accelerometer module to the BASIC Stamp, run a test program, and verify that it can be used to sense tilt.

Accelerometer Parts

The parts you will need for this activity are listed here, and Figure 3 shows their drawings.

Parallax		
Part Number	**Quantity**	**Description**
800-00016	(2)	3-inch Jumper wires
150-02210	(2)	Resistor – 220 Ω
28017	(1)	Memsic MX2125 Dual-Axis Accelerometer

Figure 3 - Accelerometer Part Drawings

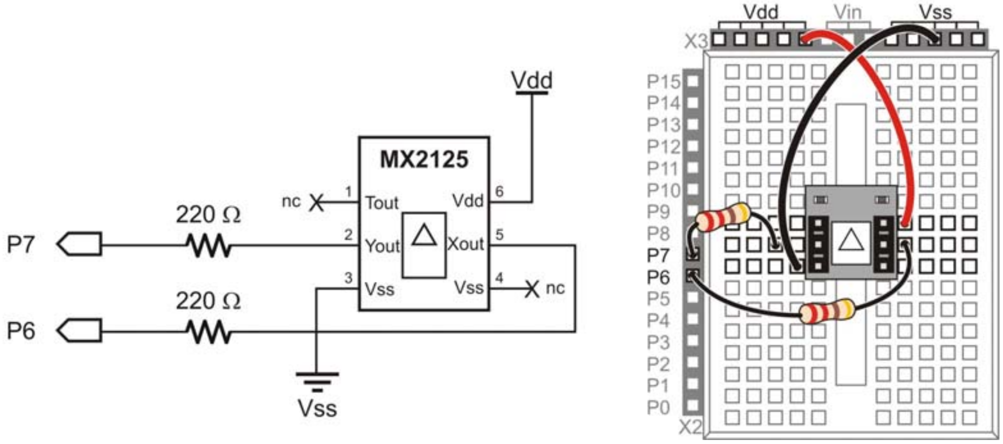


Accelerometer Electrical and Signal Connections

Figure 4 shows how to connect the accelerometer module to the Board of Education's power supply along with the BASIC Stamp I/O pin connections you will need to make to run the example program.

√ Connect the accelerometer module using the schematic and wiring diagram as your guides.

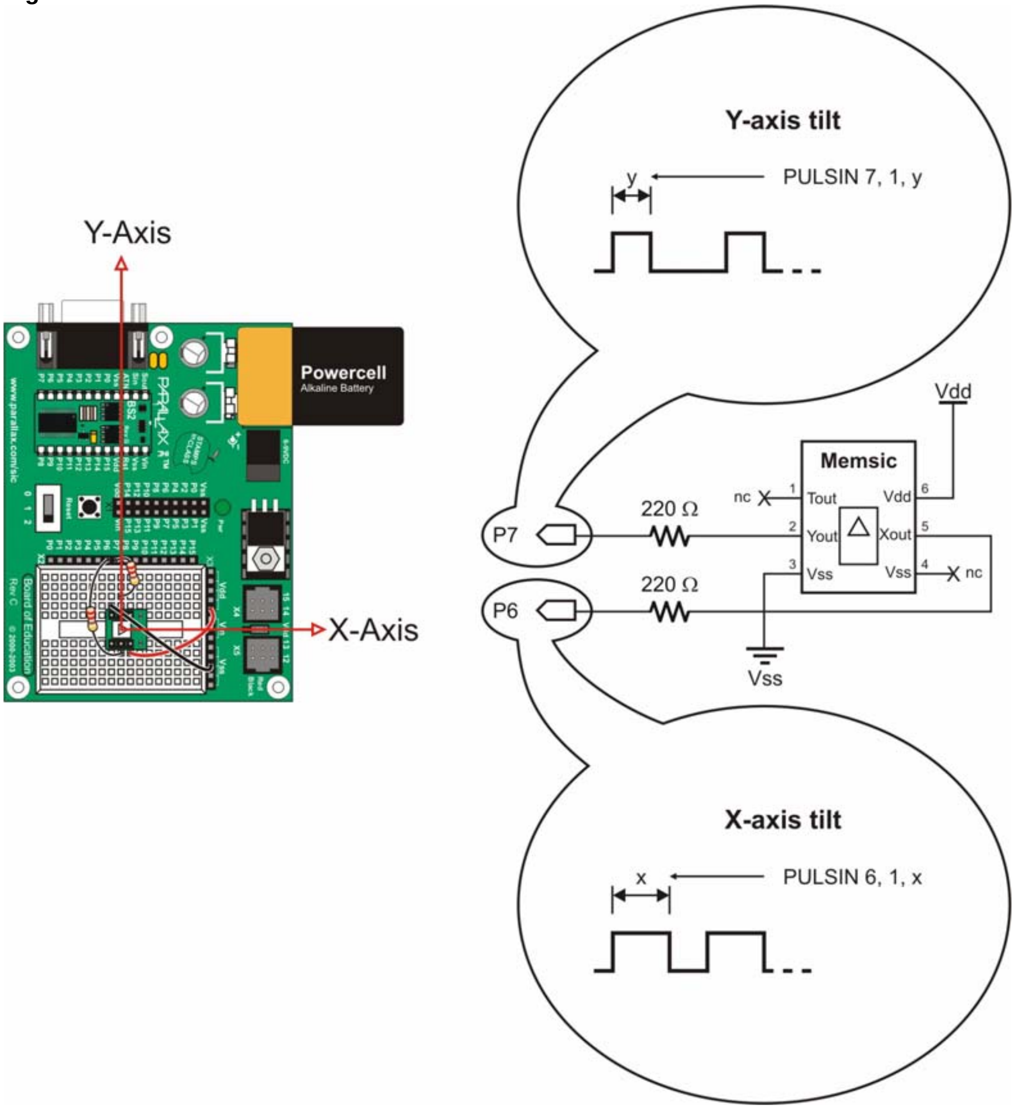
Figure 4 - Accelerometer Schematic and Wiring Diagram



Listening to the Accelerometer's Signals with the BASIC Stamp

The two axes the MX2125 uses to sense gravity and acceleration are labeled X and Y in Figure 5. It will help if you set your board flat on the table in front of you as shown in the figure. That way, the X and Y axes point the same directions they do on most XY plots. For room temperature testing, you can get a pretty good indication of tilt by just measuring the high times of the pulses sent by the MX2125's Xout and Yout pins with the **`PULSIN`** command. Depending on how far you tilt the board and in which direction, the **`PULSIN`** time measurements should range from 1775 to 3125. When the board is level, the **`PULSIN`** command should store values in the neighborhood of 2500.

Figure 5 - Accelerometer Axis Pulse Measurements



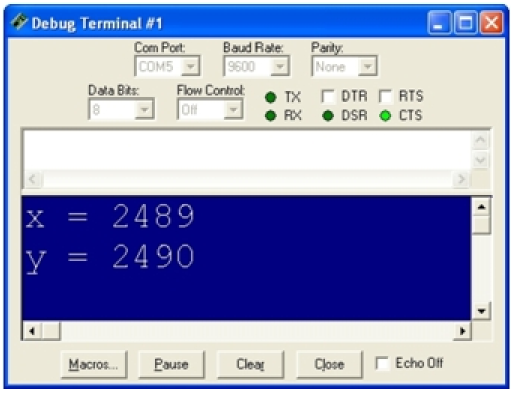
√ Make sure your board is sitting flat on the table, oriented with its X and Y axes as shown in the Figure 5.

√ Enter and run SimpleTilt.bs2.

```
' SimpleTilt.bs2
' Measure room temperature tilt.

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

x               VAR     Word
y               VAR     Word

DO

  PULSIN 6, 1, x
  PULSIN 7, 1, y

  DEBUG CLS, ? X, ? Y

  PAUSE 100

LOOP
```

√ Check to make sure the Debug Terminal reports that the x and y variables are both storing values around of 2500.

Figure 6 - Debug Terminal Output



Com Port: COM5
Baud Rate: 9600
Parity: None
Data Bits: 8
Flow Control: Off

```
x = 2489
y = 2490
```

√ Grab the edge of the board with the Y-Axis label and gradually lift it toward you. The y value should increase as you increase the tilt.

√ Keep tilting the board toward you until it's straight up and down. The Debug Terminal should report that the y variable stores a value near 3125.

√ Lay the board flat again.

√ Next, instead of tilting the board toward you, gradually tilt it away from you. The y axis value should drop below 2500 and gradually decrease to 1875 as you tilt the board until it's straight up and down.

√ Lay the board flat again.

√ Repeat this test with the X-axis. As you tilt the board up with your right hand, the x value should increase and reach a value near 3125 when the board is vertical. As you tilt the board upward with your left hand, the x value should approach 1875.

√ Finally, hold your board in front of you, straight up and down like a steering wheel.

√ As you slowly rotate your board, the x and y values should change. These values will be used in another activity to determine the rotation angle in degrees.

COMING SOON...

Activity #2: Measure 360 Degree Rotation with Arctangent
Activity #3: Measure Tilt Angle with Arcsine
Activity #4: Use Duty Cycle to Improve Your Accelerometer Measurements
Summary