Implementing Positioning Algorithms Using Accelerometers

by: **Kurt Seifert and Oscar Camacho**

OVERVIEW

This document describes and implements a positioning algorithm using the MMA7260QT 3-Axis accelerometer and a 9S08QG8 low cost 8-bit microcontroller unit (MCU).

In today's advanced electronic market, there are many multifunctional products continuously adding features and intelligence. Tracking and gaming are just a few of the markets that can benefit from obtaining positioning information. One option for obtaining this information is through the use of inertial sensors. The signal obtained from these sensors does require processing as there is no direct conversion between acceleration and position.

In order to obtain position a double integral must be applied to the signal. This document describes an easy algorithm to implement a double integration of the signal obtained from the sensor using a low end 8-bit microcontroller. In order to obtain a double integration a simple integration must be made twice. This allows velocity information to be obtained as well.

The algorithm shown in the following pages applies to any sensing axis; therefore, one, two or three dimensional positioning can be determined. When implementing positioning in 3 axes, extra processing is required to null the earth's gravity effect. The implementation below includes an example for a 2-axis system (i.e. mouse).

POTENTIAL APPLICATIONS

Potential applications for this kind of algorithm are personal navigation, car navigation, back-up GPS, anti-theft devices, map tracking, 3-D gaming, PC mouse, plus many others. The products in these categories can benefit from implementing positioning algorithms.

The algorithm described in this document is useful in situations where displacement precision is not extremely critical. Other considerations and implications specific to the application should be considered when adapting this example. With minor modifications and adaptations in the final application this algorithm can achieve higher precision.

BACK UP THEORY AND ALGORITHM

The best approach in understanding this algorithm is with a review of mathematical integration.

The acceleration is the rate of change of the velocity of an object. At the same time, the velocity is the rate of change of the position of that same object. In other words, the velocity is the derivative of the position and the acceleration is the derivative of the velocity, thus:

$$\vec{a} = \frac{d\vec{v}}{dt} \text{ and } \vec{v} = \frac{d\vec{s}}{dt} \therefore \vec{a} = \frac{d(d\vec{s})}{dt^2}$$

The integration is the opposite of the derivative. If the acceleration of an object is known, we can obtain the position data if a double integration is applied (assuming initial conditions are zero):

$$v = \int(\vec{a})dt \text{ and } \vec{s} = \int(\vec{v})dt \therefore \int\left(\int(\overrightarrow{a})dt\right)dt$$

One way to understand this formula is to define the integral as the area below the curve, where the integration is the sum of very small areas whose width is almost zero. In other words, the sum of the integration represents the magnitude of a physical variable.

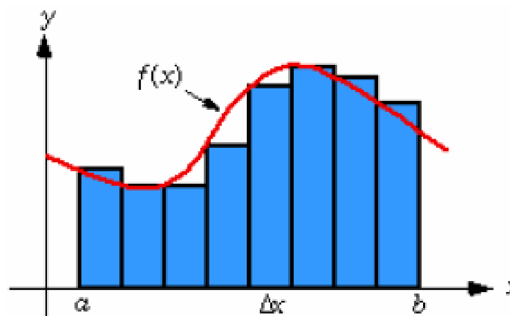

Figure 1. Sampled Accelerometer's Signal

$$\int_a^b f(x)dx = \lim_{n \to \infty} \sum_{i=1}^{n} f(x_i)\Delta x$$

Where:

$$\Delta x = \frac{b-a}{n}$$

© Freescale Semiconductor, Inc., 2007. All rights reserved.

With the previous concept about “areas below the curve” a deduction can be made: Sampling a signal gets us instant values of its magnitude, so small areas can be created between two samples. In order to create a coherent value, sampling time must always be the same. Sampling time represents the base of this area while the sampled value represents its height. In order to eliminate multiplications with fractions (microseconds or milliseconds) involving floating points in the calculation we assume the time is a unit.

Now we know each sample represents an area whose base width is equal to 1. The next deduction could then be that the value of the integral is reduced to be the sum of samples. This assumption is correct if the sampling time tends to be zero. In a real situation an error is generated as shown in Figure 1. This error keeps accumulating for the time the process is active.

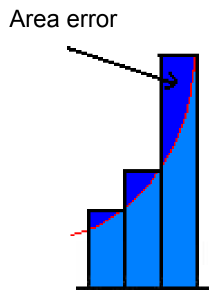

Figure 2. Errors Generated During Integration

These errors are formally known as sampling losses. In order to reduce this error we then make a further assumption. The resulting area can be seen as the combination of two smaller areas:

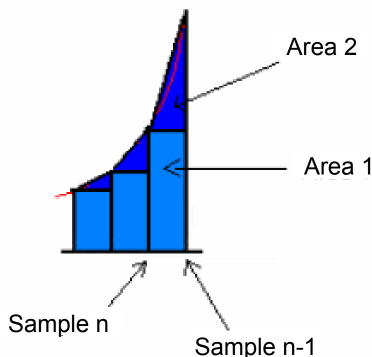

Figure 3. Errors of Integration Are Reduced with a First Order Approximation (Trapezoidal Method)

The first area is the value of the previous sample (a square). The second area is a triangle, formed between the previous sample (sample n-1) and the actual sample (sample n) divided by two.

With this approach, we now have a first order approximation (interpolation) of the signal.

Formula 1

$$Area_n = Sample_n + \frac{|Sample_n - Sample_{n-1}|}{2} \times T$$

Now the error is much lower than in the previous approximation.

Yet data must not be too accurate in order to obtain a “real world” interpretation. Even though acceleration can be positive or negative, samples are always positive (based on the output characteristics of the MMA7260QT); therefore, an offset adjustment must be done. In other words, a reference is needed. This function is defined as the calibration routine.

Calibration is performed on the accelerometer when there is a no movement condition. The output or offset obtained is considered the zero point reference. Values lower than the reference represent negative values (deceleration) while greater values represent positive values (acceleration).

The accelerometer output varies from 0V to Vdd and it is typically interpreted by an analog to digital comparator (A/D). The zero value is near Vdd/2. The calibration value obtained will be affected by the board’s orientation and the static acceleration (earth’s gravity) component in each of the axis. If the module is perfectly parallel to the earth’s surface, the calibration value should be very close to Vdd/2.

The following figure shows the result of the calibration routine:

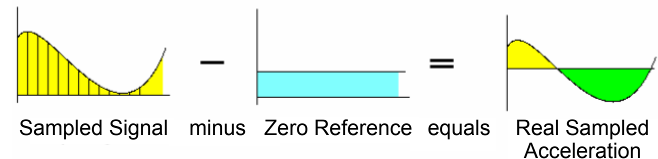

Figure 4. Acceleration Signal After Calibration

From the sampled signal minus the zero reference we obtain true sampled acceleration.

A1 represents a positive acceleration. A2 represents a negative acceleration.

If we considered this data as sampled data, the signal should be similar to the figure below.

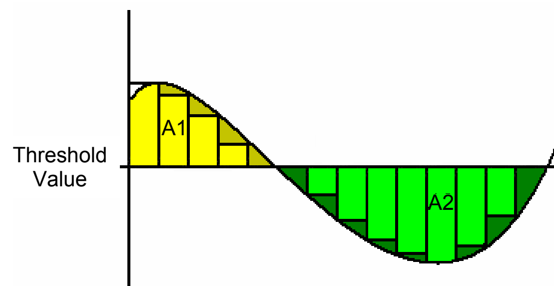

Figure 5. Acceleration Sampled Signal After Calibration

By applying the integration formula, Formula 1, we get a proportional approximation of the velocity. In order to obtain position the integration must be performed again. Applying the same formula and procedure to this obtained velocity data, we now get a proportional approximation of the instantaneous position (see Figure 6).

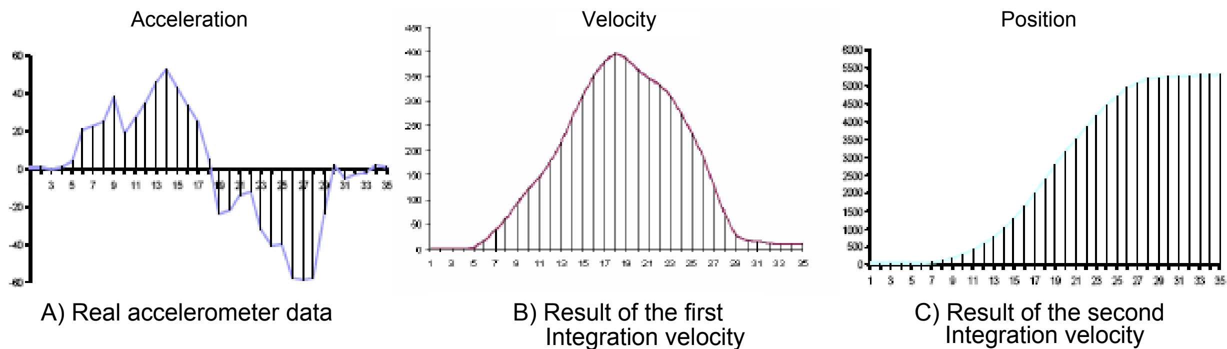

Figure 6. Proportional Approximation of the Instantaneous Position

SOFTWARE DESIGN CONSIDERATIONS

The following steps and recommendations should be considered when implementing this kind of algorithm in a “real world” implementation.

- The signal is not noise free so it must be digitally filtered. The filter used in this algorithm is a moving average; the value to be processed is the result of averaging a certain amount of samples.
- Even with the previous filtering some data can be erroneous due to the “mechanical” noise, so another filter must be implemented. Depending on the number of samples filtered, a window of “real acceleration” can be selected (typically ± 2 sample steps for an average of 16 samples).
- A “no movement” state is critical to obtain correct data. A calibration routine is needed at the beginning of the application. This calibration value must be as accurate as possible.
- The real value of the acceleration is the sample minus the calibration value; it can be either positive or negative. This must never be ignored when declaring variables (signed).
- A faster sampling frequency implies more accurate results due the fact that error is reduced; yet more memory, timing, and hardware considerations are needed.
- The time between samples MUST always be the same. Errors can be generated if this time is not equal.
- A linear approximation between samples (interpolation) is recommended for more accurate results.

CODE EXPLANATION

Calibration Routine:

This calibration routine removes the acceleration offset component in the sensor output due to the earth's gravity (static acceleration).

The calibration routine averages samples when the accelerometer is in a no movement condition. The more samples that are taken, the more accurate the calibration results will be.

```
void Calibrate(void)
{
  unsigned int count1;
  count1 = 0;

  do{
  ADC_GetAllAxis();
  sstatex = sstatex + Sample_X;              // Accumulate Samples
  sstatey = sstatey + Sample_Y;
  count1++;
  }while(count1!=0x0400);                    // 1024 times

  sstatex=sstatex>>10;                       // division between 1024
  sstatey=sstatey>>10;
}
```

Filtering:

Low pass filtering of the signal is a very good way to remove noise (both mechanical and electrical) from the accelerometer. Reducing the noise is critical for a positioning application in order to reduce major errors when integrating the signal.

A simple way for low pass filtering a sampled signal is to perform a rolling average. Filtering is simply then reduced to obtain the average of a set of samples. It is important to obtain the average of a balanced amount of samples. Taking too many samples to do this process can result in a loss of data, yet taking too few can result in an inaccurate value.

```
do{

accelerationx[1]=accelerationx[1] + Sample_X;   //filtering routine for noise attenuation
accelerationy[1]=accelerationy[1] + Sample_Y;   //64 samples are averaged. The resulting
count2++;                                       // average represents the acceleration of
                                           // an instant.
}while (count2!=0x40);                          // 64 sums of the acceleration sample

accelerationx[1]= accelerationx[1]>>6;          // division by 64
accelerationy[1]= accelerationy[1]>>6;
```

Mechanical Filtering Window:

When a no movement condition is present, minor errors in acceleration could be interpreted as a constant velocity due to the fact that samples not equal to zero are being summed; the ideal case for a no movement condition is all the samples to be zero. That constant velocity indicates a continuous movement condition and therefore an unstable position.

Even with the previous filtering some data can be erroneous, so a “window” of discrimination between “valid data” and “invalid data” for the no movement condition must be implemented.

```
if ((accelerationx[1] <=3)&&(accelerationx[1] >= -3)) //Discrimination window applied to
{accelerationx[1] = 0;}                     // the X axis acceleration variable
```

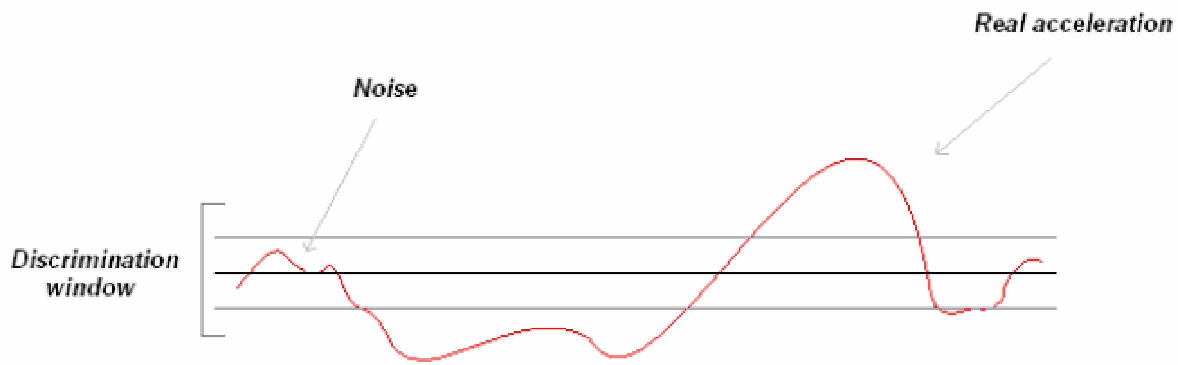

Figure 7. Software Discrimination Window To Reduce Mechanical Noise Effects

Positioning:

Double integration is the process needed to obtain the position using the acceleration data. The integration step must be performed once to obtain velocity and then repeated to obtain position.

As previously shown:

$$\text{Area}_n = \text{Sample}_n + \frac{|\text{Sample}_n - \text{Sample}_{n-1}|}{2} \times T$$

```
//first integration
velocityx[1] = velocityx[0] + accelerationx[0] + ((accelerationx[1] - accelerationx[0])>>1)


//second integration
positionX[1] = positionX[0] + velocityx[0] + ((velocityx[1] - velocityx[0])>>1);
```

Data Transfer:

This function is used for debugging and display purposes; the 32 bits result is split and conditioned in this function.

```
if (positionX[1]>=0) {                //This line compares the sign of the X direction data
  direction= (direction | 0x10);      // if its positive the most significant byte
  posx_seg[0]= positionX[1] & 0x000000FF;       // is set to 1 else it is set to 8
  posx_seg[1]= (positionX[1]>>8) & 0x000000FF; // the data is also managed in the
                                          // subsequent lines in order to be sent.
  posx_seg[2]= (positionX[1]>>16) & 0x000000FF;// The 32 bit variable must be split into
  posx_seg[3]= (positionX[1]>>24) & 0x000000FF;// 4 different 8 bit variables in order to
                                             // be sent via the 8 bit SCI frame
  }
else {
  direction=(direction | 0x80);
  positionXbkp=positionX[1]-1;
  positionXbkp=positionXbkp^0xFFFFFFFF;
  posx_seg[0]= positionXbkp & 0x000000FF;
  posx_seg[1]= (positionXbkp>>8) & 0x000000FF;
  posx_seg[2]= (positionXbkp>>16) & 0x000000FF;
  posx_seg[3]= (positionXbkp>>24) & 0x000000FF;
  }
```

"Movement End" Check

Based on the concept that an integral represents the area below the curve, velocity is the result of the area below the acceleration curve.

If we look at the typical movement of an object from point A to point B in a single axis, a typical acceleration would result as shown in the figure below:

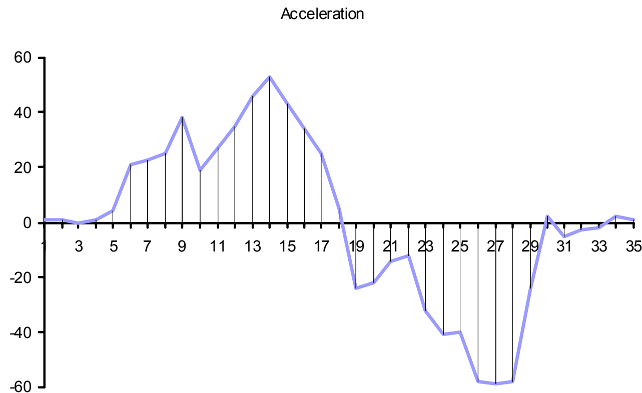

Figure 8. Typical Accelerometer Output Signal As The Result of Dragging an Object in a Single Axis

Looking at graph above, there is an initial acceleration or deceleration until a maximum velocity is reached. Then that acceleration "flips" the opposite way until it reaches zero again. At this point a stable displacement and a new position are reached.

In a real world scenario where the area below the positive side of the curve is not the same as the area above the negative side, the integration result would never reach a zero velocity and therefore would be a sloped positioning (never stable).

Because of this, it is crucial to "force" the velocity down to zero. This is achieved by constantly reading the acceleration and comparing it with zero. If this condition exists during a certain number of samples, velocity is simply returned to zero.

```
if (accelerationx[1]==0)    // we count the number of acceleration samples that equals cero
   { countx++;}
else { countx =0;}
  if (countx>=25)         // if this number exceeds 25, we can assume that velocity is cero
   {
   velocityx[1]=0;
   velocityx[0]=0;
   }
```

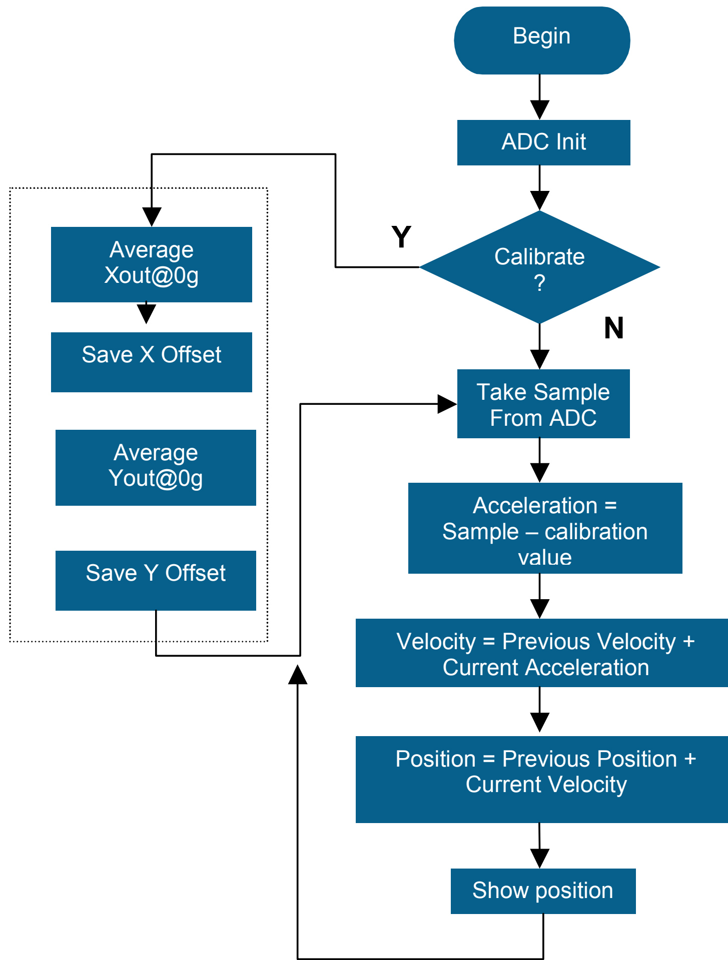

Figure 9. Flow Diagram

SOURCE CODE

```
#include <hidef.h>
#include "derivative.h"
#include "adc.h"
#include "buzzer.h"
#include "SCItx.h"

#pragma DATA_SEG MY_ZEROPAGE
unsigned char near Sample_X;
unsigned char near Sample_Y;
unsigned char near Sample_Z;
unsigned char near Sensor_Data[8];
unsigned char near countx,county ;
signed int near accelerationx[2], accelerationy[2];
signed long near velocityx[2], velocityy[2];
signed long near positionX[2];
signed long near positionY[2];
signed long near positionZ[2];
unsigned char near direction;
unsigned long near sstatex,sstatey;

#pragma DATA_SEG DEFAULT

void init(void);
void Calibrate(void);
void data_transfer(void);
```

```
void concatenate_data(void);
void movement_end_check(void);
void position(void);

void main (void)
{
  init();
  get_threshold();
  do
  {
  position();
  }while(1);
}

/*******************************************************************************
 The purpose of the calibration routine is to obtain the value of the reference threshold.
 It consists on a 1024 samples average in no-movement condition.
********************************************************************************/

void Calibrate(void)
{
  unsigned int count1;
  count1 = 0;

  do{
  ADC_GetAllAxis();
  sstatex = sstatex + Sample_X;              // Accumulate Samples
  sstatey = sstatey + Sample_Y;
  count1++;
  }while(count1!=0x0400);                    // 1024 times

  sstatex=sstatex>>10;                       // division between 1024
  sstatey=sstatey>>10;
}

/*****************************************************************************************/


/*****************************************************************************************
This function obtains magnitude and direction
In  this  particular  protocol  direction  and  magnitude  are  sent  in  separate  variables.
Management can be done in many other different ways.
*****************************************************************************************/

void data_transfer(void)
{

 signed long positionXbkp;
 signed long positionYbkp;
 unsigned int delay;
 unsigned char posx_seg[4], posy_seg[4];


  if (positionX[1]>=0) {              //This line compares the sign of the X direction data
  direction= (direction | 0x10);      //if its positive the most significant byte
  posx_seg[0]= positionX[1] & 0x000000FF;     // is set to 1 else it is set to 8
  posx_seg[1]= (positionX[1]>>8) & 0x000000FF; // the data is also managed in the
  // subsequent lines in order to
  posx_seg[2]= (positionX[1]>>16) & 0x000000FF;  // be sent. The 32 bit variable must be
  posx_seg[3]= (positionX[1]>>24) & 0x000000FF;  // split into 4 different 8 bit
    // variables in order to be sent via
    // the 8 bit SCI frame
```

```
      }


  else {direction=(direction | 0x80);
       positionXbkp=positionX[1]-1;
       positionXbkp=positionXbkp^0xFFFFFFFF;
       posx_seg[0]= positionXbkp & 0x000000FF;
       posx_seg[1]= (positionXbkp>>8) & 0x000000FF;
       posx_seg[2]= (positionXbkp>>16) & 0x000000FF;
       posx_seg[3]= (positionXbkp>>24) & 0x000000FF;
       }


  if (positionY[1]>=0) {                       //  Same management than in the previous case
       direction= (direction | 0x08);         // but with the Y data.
       posy_seg[0]= positionY[1] & 0x000000FF;
       posy_seg[1]= (positionY[1]>>8) & 0x000000FF;
       posy_seg[2]= (positionY[1]>>16) & 0x000000FF;
       posy_seg[3]= (positionY[1]>>24) & 0x000000FF;
       }

  else {direction= (direction | 0x01);
       positionYbkp=positionY[1]-1;
       positionYbkp=positionYbkp^0xFFFFFFFF;
       posy_seg[0]= positionYbkp & 0x000000FF;
       posy_seg[1]= (positionYbkp>>8) & 0x000000FF;
       posy_seg[2]= (positionYbkp>>16) & 0x000000FF;
       posy_seg[3]= (positionYbkp>>24) & 0x000000FF;
       }

  delay = 0x0100;

  Sensor_Data[0] = 0x03;
  Sensor_Data[1] = direction;
  Sensor_Data[2] = posx_seg[3];
  Sensor_Data[3] = posy_seg[3];
  Sensor_Data[4] = 0x01;
  Sensor_Data[5] = 0x01;
  Sensor_Data[6] = END_OF_FRAME;

  while (--delay);

  SCITxMsg(Sensor_Data);         //  Data transferring function
  while (SCIC2 & 0x08);
}

/*******************************************************************************************/

/*******************************************************************************************
This function returns data format to its original state. When obtaining the magnitude and
direction of the position, an inverse two's complement is made. This function makes the two's
complement in order to return the data to it original state.
It is important to notice that the sensibility adjustment is greatly impacted here, the amount
of "ones" inserted in the mask must be equivalent to the "ones" lost in the shifting made in
the previous function upon the sensibility modification.
  *******************************************************************************************/


void data_reintegration(void)
{

  if (direction >=10)
```

```
  {positionX[1]= positionX[1]|0xFFFFC000;}     // 18 "ones" inserted. Same size as the
  //amount of shifts

  direction = direction & 0x01;
  if (direction ==1)

  {positionY[1]= positionY[1]|0xFFFFC000;}
}

/******************************************************************************************
This function allows movement end detection. If a certain number of acceleration samples are
equal to zero we can assume movement has stopped. Accumulated Error generated in the velocity
calculations is eliminated by resetting the velocity variables. This stops position increment
and greatly eliminates position error.
******************************************************************************************/

void movement_end_check(void)
{

  if (accelerationx[1]==0)   //we count the number of acceleration samples that equals cero
       { countx++;}
  else { countx =0;}

  if (countx>=25)          //if this number exceeds 25, we can assume that velocity is cero
   {
   velocityx[1]=0;
   velocityx[0]=0;
   }

   if (accelerationy[1]==0)   //we do the same for the Y axis
   { county++;}
  else { county =0;}

  if (county>=25)
   {
   velocityy[1]=0;
   velocityy[0]=0;
   }
}

/******************************************************************************************/

/******************************************************************************************
This function transforms acceleration to a proportional position by integrating the
acceleration data twice. It also adjusts sensibility by multiplying the "positionX" and
"positionY" variables.
This integration algorithm carries error, which is compensated in the "movenemt_end_check"
subroutine. Faster sampling frequency implies less error but requires more memory. Keep in
mind that the same process is applied to the X and Y axis.
******************************************************************************************/

void position(void)
{

unsigned char count2 ;
count2=0;

    do{

    ADC_GetAllAxis();
    accelerationx[1]=accelerationx[1] + Sample_X; //filtering routine for noise attenuation
    accelerationy[1]=accelerationy[1] + Sample_Y; //64 samples are averaged. The resulting
```

```
     //average represents the acceleration of
     //an instant
    count2++;

    }while (count2!=0x40);                        // 64 sums of the acceleration sample


    accelerationx[1]= accelerationx[1]>>6;          // division by 64
    accelerationy[1]= accelerationy[1]>>6;

    accelerationx[1] = accelerationx[1] - (int)sstatex; //eliminating zero reference
   //offset of the acceleration data
    accelerationy[1] = accelerationy[1] - (int)sstatey; // to obtain positive and negative
   //acceleration


    if ((accelerationx[1] <=3)&&(accelerationx[1] >= -3)) //Discrimination window applied
          {accelerationx[1] = 0;}                        // to the X axis acceleration
     //variable

    if ((accelerationy[1] <=3)&&(accelerationy[1] >= -3))
          {accelerationy[1] = 0;}

  //first X integration:
velocityx[1]= velocityx[0]+ accelerationx[0]+ ((accelerationx[1] -accelerationx[0])>>1);

  //second X integration:
positionX[1]= positionX[0] + velocityx[0] + ((velocityx[1] - velocityx[0])>>1);

  //first Y integration:
velocityy[1] = velocityy[0] + accelerationy[0] + ((accelerationy[1] -accelerationy[0])>>1);

  //second Y integration:
positionY[1] = positionY[0] + velocityy[0] + ((velocityy[1] - velocityy[0])>>1);

    accelerationx[0] = accelerationx[1];  //The current acceleration value must be sent
//to the previous acceleration
    accelerationy[0] = accelerationy[1];     //variable in order to introduce the new
//acceleration value.

    velocityx[0] = velocityx[1];          //Same done for the velocity variable
    velocityy[0] = velocityy[1];


    positionX[1] = positionX[1]<<18;      //The idea behind this shifting (multiplication)
     //is a sensibility adjustment.
    positionY[1] = positionY[1]<<18;      //Some applications require adjustments to a
     //particular situation
                                         //i.e. mouse application
    data_transfer();

    positionX[1] = positionX[1]>>18;      //once the variables are sent them must return to
    positionY[1] = positionY[1]>>18;      //their original state

    movement_end_check();

    positionX[0] = positionX[1];          //actual position data must be sent to the
    positionY[0] = positionY[1];      //previous position

    direction = 0;                        // data variable to direction variable reset
}
/*******************************************************************************************/
```

SCHEMATIC

CONCLUSIONS

This application note provides the basic concepts for implementing a positioning algorithm using accelerometers.

This specific algorithm is useful in situations where displacement precision is not extremely critical. Other considerations and implications specific to the application should be considered when implementing this example application.

This integration algorithm is suitable for low end embedded applications because of its simplicity and small amount of instructions. It also does not involve any floating point calculations.

REFERENCES

Refer to the following documents for product information

MC9S08QG8 Technical Data Sheet

MMA7260Q Technical Data Sheet

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2007. All rights reserved.

AN3397
Rev. 0
02/2007