AN-1023
APPLICATION NOTE

One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106, U.S.A. • Tel: 781.329.4700 • Fax: 781.461.3113 • www.analog.com

Fall Detection Application by Using 3-Axis Accelerometer ADXL345

by Ning Jia

INTRODUCTION

Senior citizens often suffer accidental falls due to their diminished self-care and self-protection ability. These accidents may possibly have serious consequences if no aid is given in time. Statistics show that the majority of serious consequences are not the direct result of the falls, but rather are due to a delay in assistance and treatment after a fall. In the event of a fall, the danger of post-fall consequences can be greatly reduced if relief personnel can be alerted in time. In light of this, there has been increased development of devices for detection and prediction of fall situations.

In recent years, technological advancements in MEMS accelerometer sensors have made it possible to design a fall detector based on a 3-axis accelerometer sensor. These fall detectors operate on the principle of detecting changes in body position when moving by tracking acceleration changes in three orthogonal directions of an individual wearing a sensor. The data is then analyzed algorithmically to determine whether the individual's body is falling. If an individual falls, the device works with a GPS module and a wireless transmitter module to determine the position and issues an alert for assistance. The core part of the fall detector is therefore the detection principle and the algorithm to judge the existence of an emergency fall situation.

The ADXL345 is the latest 3-axis, digital output accelerometer from Analog Devices, Inc., and is well-suited for fall detector applications. This application note, based on the principle research of fall detection for an individual body, proposes a new solution for detection of such fall situations using the ADXL345.

ADXL345 MEMS ACCELEROMETER

Micro Electronic Mechanical Systems (MEMS) is a semiconductor technology that builds micromechanical structures and electrical circuits into a single silicon chip. The MEMS accelerometer is a sensor based on this technology to achieve acceleration sensing on single-axis, dual-axis, or tri-axis conditions. Depending on the application, the accelerometer may offer different ranges of detection from several *g* to tens of *g* of digital or analog output, and may have multiple interrupt modes. These features offer the user more convenient and flexible solutions.

The ADXL345 is the latest MEMS 3-axis accelerometer with digital output from Analog Devices. It features a selectable ±2 *g*, ±4 *g*, ±8 *g*, or ±16 *g* measurement range; up to 13-bit resolution; fixed 4 m*g*/LSB sensitivity; 3 mm × 5 mm × 1 mm ultrasmall package; 40 μA to 145 μA ultralow power consumption; standard I^2C and SPI digital interface; 32-level FIFO storage; various built-in motion status detection options; and a flexible interrupt system. These features greatly simplify the algorithm for fall detection, and thus make ADXL345 an ideal accelerometer for fall detector applications. The fall detection solution, as proposed in this application note, is fully based on the ADXL345's internal functions of motion status detection and interrupt system, and the complexity of the algorithm can be minimized with little requirement to access the actual acceleration values or to perform any other computations.

The interrupt system of the ADXL345 is described in the Interrupts section. For more detailed specifications of the ADXL345, refer to the data sheet or visit www.analog.com. Figure 1 shows the system block diagram and Figure 2 shows the pin definitions of the ADXL345.

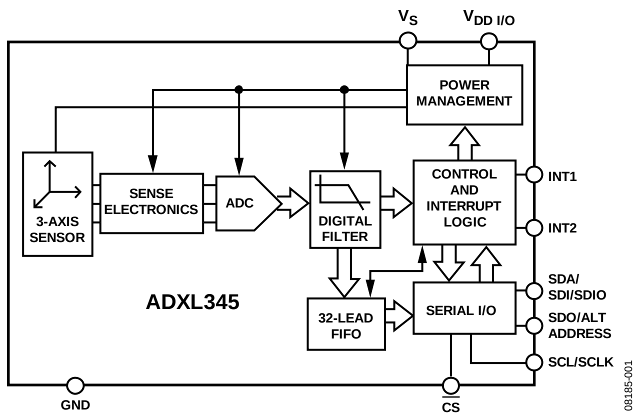

Figure 1. ADXL345 System Block Diagram

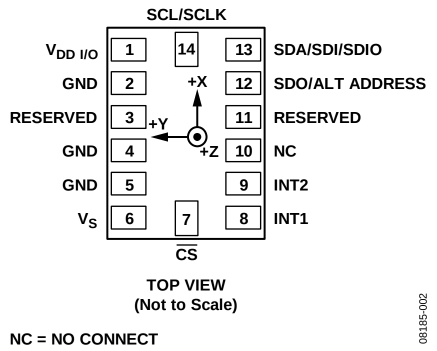

Figure 2. ADXL345 Pin Configuration

TABLE OF CONTENTS

INTERRUPTS

The ADXL345 features two programmable interrupt pins, INT1 and INT2, with a total of eight interrupts: DATA_READY, SINGLE_TAP, DOUBLE_TAP, activity, inactivity, FREE_FALL, watermark, and overrun. Each interrupt can be enabled or disabled independently by setting the appropriate bit in the INT_ENABLE register, with the option to map to either the INT1 or the INT2 pin.

DATA_READY

The DATA_READY bit is set when new data is available and cleared when no new data is available.

SINGLE_TAP

The SINGLE_TAP bit is set when a single acceleration event that is greater than the value in the THRESH_TAP register occurs for a shorter length of time than is specified in the DUR register.

DOUBLE_TAP

The DOUBLE_TAP bit is set when two acceleration events that are greater than the value in the THRESH_TAP register occur for a shorter length of time than is specified in the DUR register, with the second tap starting after the time specified by the latent register and within the time specified in the window register. Figure 3 illustrates the valid SINGLE_TAP and DOUBLE_TAP interrupts.

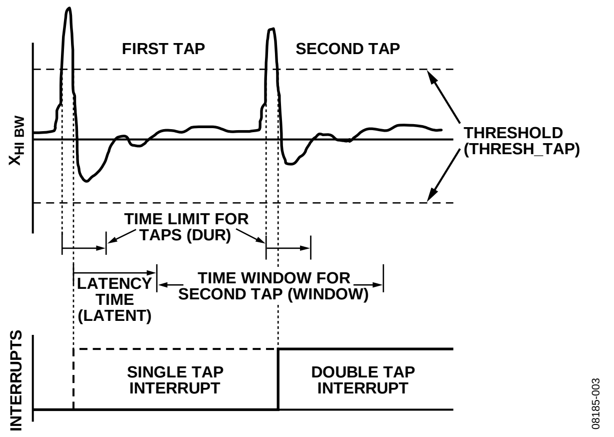

Figure 3. SINGLE_TAP and DOUBLE_TAP Interrupts

Activity

The activity bit is set when acceleration greater than the value stored in the THRESH_ACT register is experienced.

Inactivity

The inactivity bit is set when acceleration of less than the value stored in the THRESH_INACT register is experienced for a longer length of time than is specified in the TIME_INACT register. The maximum value for TIME_INACT is 255 sec. For the activity and inactivity interrupts, the user can individually enable or disable each x-, y-, or z-axis. For example, the activity interrupt for the x-axis can be enabled while disabling the interrupt for the y-axis and z-axis. Furthermore, the user can select between dc-coupled or ac-coupled operation mode for the activity and inactivity interrupts. In dc-coupled operation, the current acceleration is compared with THRESH_ACT and THRESH_INACT directly to determine whether activity or inactivity is detected. In ac-coupled operation for activity detection, the acceleration value at the start of activity detection is taken as a reference value. New samples of acceleration are then compared to this reference value, and if the magnitude of the difference exceeds THRESH_ACT, the device triggers an activity interrupt. In ac-coupled operation for inactivity detection, a reference value is used again for comparison and is updated whenever the device exceeds the inactivity threshold. Once the reference value is selected, the device compares the magnitude of the difference between the reference value and the current acceleration with THRESH_INACT. If the difference is below THRESH_INACT for a total of TIME_INACT, the device is considered inactive and the inactivity interrupt is triggered.

FREE_FALL

The FREE_FALL bit is set when acceleration of less than the value stored in the THRESH_FF register is experienced for a longer length of time than is specified in the TIME_FF register. FREE_FALL interrupt is mainly used in detection of free-falling motion. As a result, the FREE_FALL interrupt differs from the inactivity interrupt in that all axes always participate, the timer period is much smaller (1.28 sec maximum), and it is always dc-coupled.

Watermark

The watermark bit is set when the FIFO has filled up to the value stored in the samples register. It is cleared automatically when the FIFO is read and its content emptied below the value stored in the samples register. The FIFO in the ADXL345 has four operation modes: bypass mode, FIFO mode, stream mode, and trigger mode; and can store up to 32 samples (x-, y-, and z-axis). The FIFO function is an important and very useful feature for the ADXL345; however, the proposed solution for fall detection does not use the FIFO function and thus is not further discussed in this application note. For further details on the FIFO function, see the ADXL345 data sheet.

Overrun

The overrun bit is set when new data has replaced unread data. The precise operation of the overrun function depends on the FIFO mode. In bypass mode, the overrun bit is set when new data replaces unread data in the DATAX, DATAY, and DATAZ registers. In all other modes, the overrun bit is set when the FIFO is filled with 32 samples. The overrun bit is cleared by reading the FIFO contents, and is automatically cleared when the data is read.

ACCELERATION CHANGE CHARACTERISTICS DURING THE FALL PROCESS

The main research on the principles of fall detection focuses on the acceleration change characteristics during the process of a human body falling. Figure 4 to Figure 7 present the acceleration change curves during the motions of walking downstairs, walking upstairs, sitting down, and standing up from a chair. (The fall detector is belt-wired on the individual's body.)

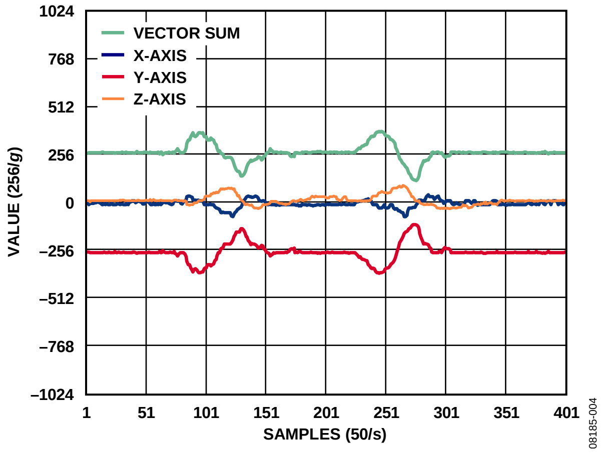

Figure 4. Acceleration Change Curves During Process of Walking Downstairs

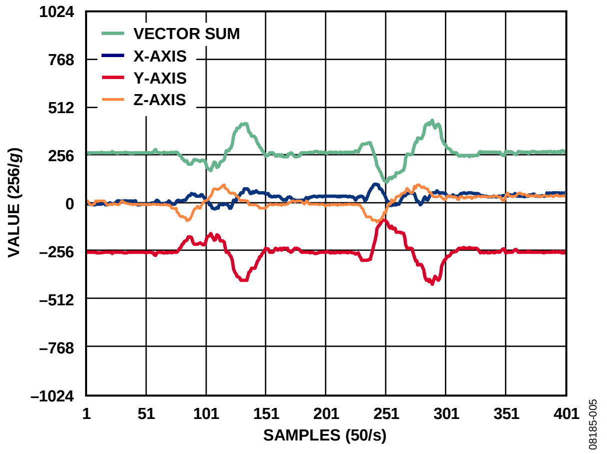

Figure 5. Acceleration Change Curves During Process of Walking Upstairs

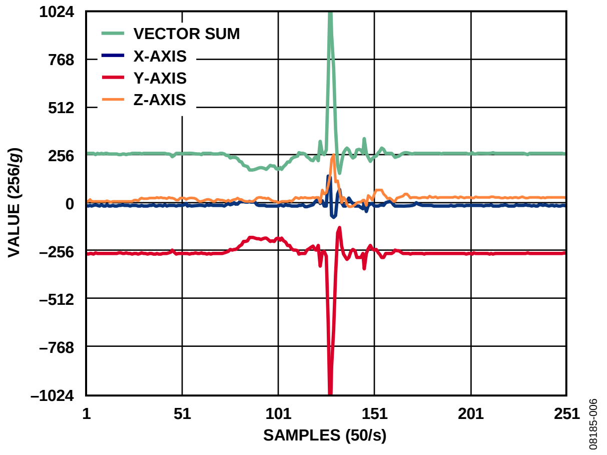

Figure 6. Acceleration Change Curves During Process of Sitting Down

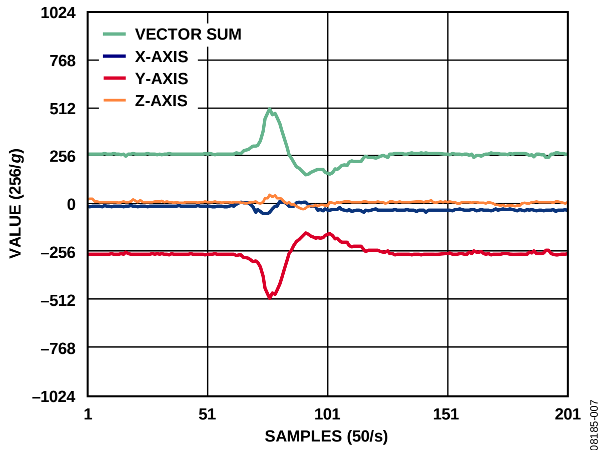

Figure 7. Acceleration Change Curves During Process of Standing Up

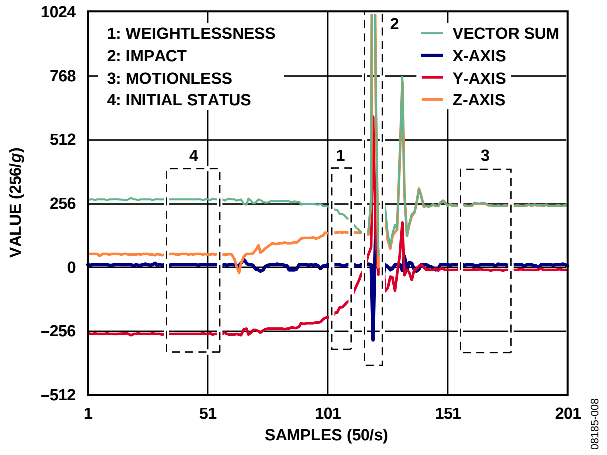

Figure 8. Acceleration Change Curves During the Process of Falling

Because the movement of senior citizens is comparatively slow, the acceleration change is not very conspicuous during the walking motions in Figure 4 and Figure 5. Figure 8 presents the acceleration change curves during the process of falling. By comparing Figure 8 with Figure 4 to Figure 7, it can be seen that there are four critical characteristics of a falling event. These four characteristics can be used as the criterion of the fall detection. They are marked by the boxes in Figure 8 and explained in detail as follows.

Weightlessness

The phenomenon of weightlessness always occur at the start of a fall. This phenomenon becomes more significant during free-fall, and the vector sum of acceleration reduces to near 0 *g*, The duration depends on the height of the free-fall. Even though weightlessness during an ordinary fall is not as significant as that during a free-fall, the vector sum of acceleration is also less than 1 *g* (generally greater than 1 *g* under normal conditions). Therefore, this is the first basis for determining the fall status that can be detected by the FREE_FALL interrupt of the ADXL345.

Impact

After experiencing weightlessness, the human body makes impact with the ground; the acceleration curve shows this as a large shock in Figure 8. This shock is detected by the activity interrupt of the ADXL345. Therefore, the second basis for determining a fall is the activity interrupt immediately after the FREE_FALL interrupt.

Motionless

Generally, the human body, after falling and making impact, cannot rise immediately. Instead, it remains in a motionless position for a short period. This is shown on the acceleration curve as a segment of a flat line in Figure 8, and is detected by the inactivity interrupt of the ADXL345. Therefore, the third basis for determining a fall situation is the inactivity interrupt after the activity interrupt.

Initial Status

After a fall, the human body turns over, so the acceleration in three axes is different from the initial status before the fall. If the fall detector is belt-wired on the human body to obtain the initial status of the acceleration, the acceleration data in three axes can be read after the inactivity interrupt, and the sampling data can then be compared with the initial status. Therefore, it is the fourth basis for determining a fall if the difference between sampling data and initial status exceeds a certain threshold, for example, 0.7 *g*.

The combination of these four bases of determination form the entire fall detection algorithm, and then the system can raise an alert accordingly for the fall status. The time interval between interrupts must be within a reasonable range. In normal cases, the time interval between the FREE_FALL interrupt (weightlessness) and the activity interrupt (impact) should not be very long unless falling from a very tall distance. Similarly, the time interval between the activity interrupt (impact) and the inactivity interrupt (motionless) should not be very long. A practical example is given in the Using the ADXL345 to Simplify Fall Detection Algorithms section with a set of reasonable values. The related interrupt detection threshold and time parameters can be flexibly set as needed. Furthermore, if a fall results in serious consequences such as a coma, the human body remains motionless for an even longer period of time. This status can still be detected by the inactivity interrupt. Therefore, a critical alert can be sent out again if the inactive state was detected to continue for a certain long period of time after a fall.

TYPICAL CIRCUIT CONNECTION OF THE SYSTEM

The circuit connection between the ADXL345 and an MCU is very simple. For this application note, a test platform was created using the ADXL345 and the ADuC7026 microcontroller. Figure 9 shows the typical connection between the ADXL345 and the ADuC7026. With the $\overline{CS}$ pin of the ADXL345 tied high, the ADXL345 works in I^{2}C mode. The SDA and SCL are the data and the clock of the I^{2}C bus, which are connected to the corresponding pins of the ADuC7026. A GPIO of the ADuC7026 is connected to the SDO/ALT ADDRESS pin of the ADXL345 to select the I^{2}C address of the ADXL345. The INT1 pin of the ADXL345 is connected to an IRQ input of the ADuC7026 to generate the interrupt signal. Almost any MCU or processor can be used to access the ADXL345 with a circuit connection similar to the one shown in Figure 9. The ADXL345 can also work in SPI mode to achieve a higher data rate. For an example circuit for SPI connection, refer to the ADXL345 data sheet.

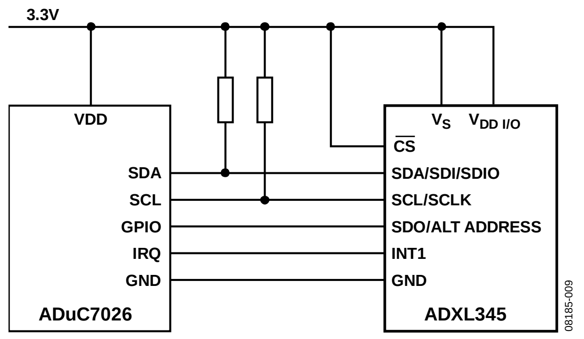

Figure 9. Typical Circuit Connection Between ADXL345 and MCU

USING THE ADXL345 TO SIMPLIFY FALL DETECTION ALGORITHMS

This section presents the realization of the algorithm from the solution mentioned previously.

Table 1 presents the function of each register and the values used in the present algorithm. Refer to the ADXL345 data sheet for detailed definitions of each register bit.

Note that some of the registers presented in Table 1 have two algorithm setting values. This indicates that the algorithm switches between these two values to achieve different detection purposes. The algorithm flow chart is shown in Figure 10.

Table 1. ADXL345 Registers Function Descriptions

Hex Address	Dec Address	Register Name	Type	Reset Value	Description	Settings in Algorithm	Function of the Settings in Algorithm
0x00	0	DEVID	Read-only	0xE5	Device ID	Read-only	
0x01 to 0x1C	1 to 28	Reserved	Reserved		Reserved, do not access	Reserved	
0x1D	29	THRESH_TAP	Read/write	0x00	Tap threshold	Not used	
0x1E	30	OFSX	Read/write	0x00	X-axis offset	0x06	X-axis offset compensation, obtain from initialization calibration
0x1F	31	OFSY	Read/write	0x00	Y-axis offset	0xF9	Y-axis offset compensation, obtain from initialization calibration
0x20	32	OFSZ	Read/write	0x00	Z-axis offset	0xFC	Z-axis offset compensation, obtain from initialization calibration
0x21	33	DUR	Read/write	0x00	Tap duration	Not used	
0x22	34	Latent	Read/write	0x00	Tap latency	Not used	
0x23	35	Window	Read/write	0x00	Tap window	Not used	
0x24	36	THRESH_ACT	Read/write	0x00	Activity threshold	0x20/0x08	Set activity threshold as 2 *g*/0.5 *g*
0x25	37	THRESH_INACT	Read/write	0x00	Inactivity threshold	0x03	Set inactivity threshold as 0.1875 *g*
0x26	38	TIME_INACT	Read/write	0x00	Inactivity time	0x02/0x0A	Set inactivity time as 2 sec or 10 sec
0x27	39	ACT_INACT_CTL	Read/write	0x00	Axis enable control for activity/inactivity	0x7F/0xFF	Enable activity and inactivity of x-, y-, z-axis, wherein inactivity is ac-coupled mode, activity is dc-coupled/ ac-coupled mode
0x28	40	THRESH_FF	Read/write	0x00	Free-fall threshold	0x0C	Set free-fall threshold as 0.75 *g*
0x29	41	TIME_FF	Read/write	0x00	Free-fall time	0x06	Set free-fall time as 30 ms
0x2A	42	TAP_AXES	Read/write	0x00	Axis control for tap/double tap	Not used	
0x2B	43	ACT_TAP_STATUS	Read-only	0x00	Source of activity/tap	Read-only	
0x2C	44	BW_RATE	Read/write	0x0A	Data rate and power mode control	0x0A	Set sample rate as 100 Hz
0x2D	45	POWER_CTL	Read/write	0x00	Power save features control	0x00	Set as normal working mode
0x2E	46	INT_ENABLE	Read/write	0x00	Interrupt enable control	0x1C	Enable activity, inactivity, free-fall interrupts
0x2F	47	INT_MAP	Read/write	0x00	Interrupt mapping control	0x00	Map all interrupts to INT1 pin
0x30	48	INT_SOURCE	Read-only	0x00	Source of interrupts	Read-only	
0x31	49	DATA_FORMAT	Read/write	0x00	Data format control	0x0B	Set as ±16 *g* measurement range, 13-bit right alignment, high level interrupt trigger, I^{2}C interface
0x32	50	DATAX0	Read-only	0x00	X-Axis Data 0	Read-only	
0x33	51	DATAX1	Read-only	0x00	X-Axis Data 1	Read-only	
0x34	52	DATAY0	Read-only	0x00	Y-Axis Data 0	Read-only	
0x35	53	DATAY1	Read-only	0x00	Y-Axis Data 1	Read-only	
0x36	54	DATAZ0	Read-only	0x00	Z-Axis Data 0	Read-only	
0x37	55	DATAZ1	Read-only	0x00	Z-Axis Data 1	Read-only	
0x38	56	FIFO_CTL	Read/write	0x00	FIFO control	Not used	
0x39	57	FIFO_STATUS	Read-only	0x00	FIFO status	Not used	

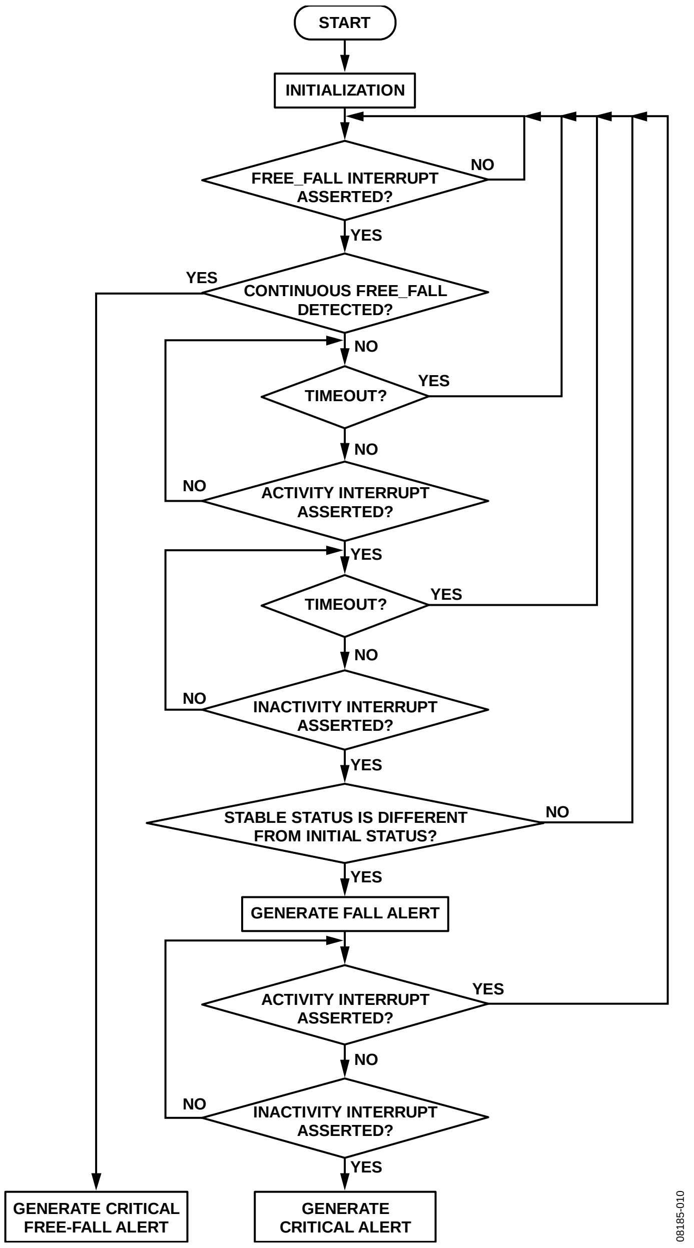
START
INITIALIZATION
FREE_FALL INTERRUPT ASSERTED?
NO
YES
CONTINUOUS FREE_FALL DETECTED?
YES
NO
TIMEOUT?
YES
NO
ACTIVITY INTERRUPT ASSERTED?
NO
YES
TIMEOUT?
YES
NO
INACTIVITY INTERRUPT ASSERTED?
NO
YES
STABLE STATUS IS DIFFERENT FROM INITIAL STATUS?
NO
YES
GENERATE FALL ALERT
ACTIVITY INTERRUPT ASSERTED?
YES
NO
INACTIVITY INTERRUPT ASSERTED?
NO
YES
GENERATE CRITICAL FREE-FALL ALERT
GENERATE CRITICAL ALERT
08185-010

Figure 10. Algorithm Flow Chart

Each interrupt threshold and related time parameter in the algorithm is as follows:

1. After initialization, the system waits for the FREE_FALL interrupt (weightlessness). THRESH_FF is set to 0.75 *g* and TIME_FF is set to 30 ms.
2. After the FREE_FALL interrupt is asserted, the system begins waiting for the activity interrupt (impact). THRESH_ACT is set to 2 *g* and the activity interrupt is operating in dc-coupled mode.
3. The time interval between the FREE_FALL interrupt (weightlessness) and the activity interrupt (impact) is set to 200 ms. If time between these two interrupts is greater than 200 ms, then the status is not valid. The 200 ms counter is realized through the MCU timer.
4. After the activity interrupt is asserted, the system begins waiting for the inactivity interrupt (motionless after impact). THRESH_INACT is set to 0.1875 *g* and TIME_INACT is set to 2 sec. The inactivity interrupt is operating in ac-coupled mode.
5. The inactivity interrupt (motionless after impact) should be asserted within 3.5 sec after the activity interrupt (impact). Otherwise, the result is invalid. The 3.5 sec counter is realized through the MCU timer.
6. If the acceleration difference between stable status and initial status exceeds the 0.7 *g* threshold, a valid fall is detected and system raises a fall alert.
7. After detecting a fall, the activity interrupt and inactivity interrupt must be continuously monitored to determine if there is a long period of motionlessness after the fall. The THRESH_ACT is set to 0.5 *g* and the activity interrupt is operating in ac-coupled mode. THRESH_INACT is set to 0.1875 *g*, TIME_INACT is set to 10 sec and the inactivity interrupt is operating in ac-coupled mode; that is, if the subject's body remains motionless for 10 sec, the inactivity interrupt is asserted and the system raises a critical alert. When the individual's body moves, the activity interrupt is generated and completes the entire sequence.
8. The algorithm can also detect that if the human body free falls from a tall distance. The two FREE_FALL interrupts are considered continuous if the interval between them is shorter than 100 ms. A critical free-fall alert is raised if the FREE_FALL interrupt (weightlessness) is continuously asserted for 300 ms

$$S = \frac{1}{2}gt^2 = \frac{1}{2}\times 10\times 0.3^2 = 0.45\text{ m}$$

This algorithm is developed in C language to be executed on the ADuC7026 microcontroller. A test case is also presented with the proposed solution to verify the algorithm. Each position, including falling forward, falling backward, falling to the left, and falling to the right, is tested 20 times. The first 10 trials are the typical falls without prolonged motionless period after a fall and the second 10 trials are the typical falls with prolonged motionless period after fall. Table 2 presents the test results.

From this experiment, the falling status can be effectively detected with the ADXL345-based proposed solution. Note that this is only a simple experiment and a more comprehensive, effective, and long-term experimentation is required to verify the reliability of this proposed solution.

Table 2. Test Results

	Test Condition		Test Result	
Trial No.	**Falling Position**	**With Prolonged Motionless Period After Fall**	**Fall Detected (No. of Times)**	**Prolonged Motionless Detected (No. of Times)**
1 to 10	Falling forward	No	10	0
11 to 20	Falling forward	Yes	10	10
21 to 30	Falling backward	No	10	0
31 to 40	Falling backward	Yes	10	10
41 to 50	Falling to the left	No	10	0
51 to 60	Falling to the left	Yes	10	10
61 to 70	Falling to the right	No	10	0
71 to 80	Falling to the right	Yes	10	10

EXAMPLE CODE

This section presents the example C code of the proposed solution-based ADXL345 and ADuC7026 platform. There are four .h files and one .c file in the project, compiled by Keil UV3. The FallDetection.c file includes the fall detection algorithm. FallDetection.h details the definitions and variables used for the fall detection algorithm, implementation of the ADXL345 read/write functions, and ADXL345 initialization.

ADuC7026Driver.h includes ADuC7026 GPIO control functions, I^2C master read and write functions, and ADuC7026 initialization. The xl345.h file includes ADXL345 registers and bit definitions. The xl345_io.h file includes wrapper functions for writing and reading bursts from/to the ADXL345 for both I^2C and SPI.

FallDetection.c

```
#include "FallDetection.h" // Include header files

void IRQ_Handler() __irq   // IRQ interrupt
{
      unsigned char i;
      if((IRQSTA & GP_TIMER_BIT)==GP_TIMER_BIT)   // TIMER1 interrupt, interval 20ms
      {
            T1CLRI = 0;        // Clear TIMER1 interrupt
            if(DetectionStatus==0xF2)  // Strike after weightlessness is detected, waiting for stable
            {
                  TimerWaitForStable++;
                  if(TimerWaitForStable>=STABLE_WINDOW)  // Time out, restart
                  {
                        IRQCLR  = GP_TIMER_BIT;  // Disable ADuC7026's Timer1 interrupt
                        DetectionStatus=0xF0;
                        putchar(DetectionStatus);
                        ADXL345Registers[XL345_THRESH_ACT]=STRIKE_THRESHOLD;
                        ADXL345Registers[XL345_THRESH_INACT]=NOMOVEMENT_THRESHOLD;
                        ADXL345Registers[XL345_TIME_INACT]=STABLE_TIME;
                        ADXL345Registers[XL345_ACT_INACT_CTL]=XL345_INACT_Z_ENABLE | XL345_INACT_Y_ENABLE
| XL345_INACT_X_ENABLE | XL345_INACT_AC | XL345_ACT_Z_ENABLE | XL345_ACT_Y_ENABLE | XL345_ACT_X_ENABLE |
XL345_ACT_DC;
                        xl345Write(4, XL345_THRESH_ACT, &ADXL345Registers[XL345_THRESH_ACT]);
                  }
            }
            else if(DetectionStatus==0xF1)  // Weightlessness is detected, waiting for strike
            {
                  TimerWaitForStrike++;
                  if(TimerWaitForStrike>=STRIKE_WINDOW)  // Time out, restart
                  {
                        IRQCLR  = GP_TIMER_BIT;  // Disable ADuC7026's Timer1 interrupt
                        DetectionStatus=0xF0;
                        putchar(DetectionStatus);
                        ADXL345Registers[XL345_THRESH_ACT]=STRIKE_THRESHOLD;
                        ADXL345Registers[XL345_THRESH_INACT]=NOMOVEMENT_THRESHOLD;
                        ADXL345Registers[XL345_TIME_INACT]=STABLE_TIME;
                        ADXL345Registers[XL345_ACT_INACT_CTL]=XL345_INACT_Z_ENABLE | XL345_INACT_Y_ENABLE
| XL345_INACT_X_ENABLE | XL345_INACT_AC | XL345_ACT_Z_ENABLE | XL345_ACT_Y_ENABLE | XL345_ACT_X_ENABLE |
XL345_ACT_DC;
                        xl345Write(4, XL345_THRESH_ACT, &ADXL345Registers[XL345_THRESH_ACT]);
                  }
            }
        }
        if((IRQSTA&SPM4_IO_BIT)==SPM4_IO_BIT)  // External interrupt form ADXL345 INT0
        {
            IRQCLR = SPM4_IO_BIT;  // Disable ADuC7026's external interrupt
            xl345Read(1, XL345_INT_SOURCE, &ADXL345Registers[XL345_INT_SOURCE]);
            if((ADXL345Registers[XL345_INT_SOURCE]&XL345_ACTIVITY)==XL345_ACTIVITY)  // Activity interrupt
asserted
            {
                if(DetectionStatus==0xF1)  // Waiting for strike, and now strike is detected
                {
                    DetectionStatus=0xF2;  // Go to Status "F2"
                    putchar(DetectionStatus);
                    ADXL345Registers[XL345_THRESH_ACT]=STABLE_THRESHOLD;
                    ADXL345Registers[XL345_THRESH_INACT]=NOMOVEMENT_THRESHOLD;
                    ADXL345Registers[XL345_TIME_INACT]=STABLE_TIME;
                    ADXL345Registers[XL345_ACT_INACT_CTL]=XL345_INACT_Z_ENABLE | XL345_INACT_Y_ENABLE
| XL345_INACT_X_ENABLE | XL345_INACT_AC | XL345_ACT_Z_ENABLE | XL345_ACT_Y_ENABLE | XL345_ACT_X_ENABLE |
XL345_ACT_AC;
                    xl345Write(4, XL345_THRESH_ACT, &ADXL345Registers[XL345_THRESH_ACT]);
                    IRQEN|=GP_TIMER_BIT;  // Enable ADuC7026's Timer1 interrupt
                    TimerWaitForStable=0;
                }
                else if(DetectionStatus==0xF4) // Waiting for long time motionless, but a movement is
detected
                {
                    DetectionStatus=0xF0;  // Go to Status "F0", restart
                    putchar(DetectionStatus);
                    ADXL345Registers[XL345_THRESH_ACT]=STRIKE_THRESHOLD;
                    ADXL345Registers[XL345_THRESH_INACT]=NOMOVEMENT_THRESHOLD;
                    ADXL345Registers[XL345_TIME_INACT]=STABLE_TIME;
                    ADXL345Registers[XL345_ACT_INACT_CTL]=XL345_INACT_Z_ENABLE | XL345_INACT_Y_ENABLE
| XL345_INACT_X_ENABLE | XL345_INACT_AC | XL345_ACT_Z_ENABLE | XL345_ACT_Y_ENABLE | XL345_ACT_X_ENABLE |
XL345_ACT_DC;
                    xl345Write(4, XL345_THRESH_ACT, &ADXL345Registers[XL345_THRESH_ACT]);
                }
            }
            else if((ADXL345Registers[XL345_INT_SOURCE]&XL345_INACTIVITY)==XL345_INACTIVITY)  // Inactivity
interrupt asserted
            {
                if(DetectionStatus==0xF2)  // Waiting for stable, and now stable is detected
                {
                    DetectionStatus=0xF3;  // Go to Status "F3"
                    IRQCLR  = GP_TIMER_BIT;
                    putchar(DetectionStatus);
                    xl345Read(6, XL345_DATAX0, &ADXL345Registers[XL345_DATAX0]);
                    DeltaVectorSum=0;
                    for(i=0;i<3; i++)
                    {
                        Acceleration[i]=ADXL345Registers[XL345_DATAX1+i*2]&0x1F;
                        Acceleration[i]=(Acceleration[i]<<8)|ADXL345Registers[XL345_DATAX0+i*2];
                        if(Acceleration[i]<0x1000)
                        {
```

```
                                    Acceleration[i]=Acceleration[i]+0x1000;
                              }
                              else  //if(Acceleration[i]>= 0x1000)
                              {
                                    Acceleration[i]=Acceleration[i]-0x1000;
                              }
                              if(Acceleration[i]>InitialStatus[i])
                              {
                                    DeltaAcceleration[i]=Acceleration[i]-InitialStatus[i];
                              }
                              else
                              {
                                    DeltaAcceleration[i]=InitialStatus[i]-Acceleration[i];
                              }
                              DeltaVectorSum=DeltaVectorSum+DeltaAcceleration[i]*DeltaAcceleration[i];
                        }
                        if(DeltaVectorSum>DELTA_VECTOR_SUM_THRESHOLD)  // The stable status is different 
from the initial status
                        {
                              DetectionStatus=0xF4;  // Valid  fall detection
                              putchar(DetectionStatus);
                              ADXL345Registers[XL345_THRESH_ACT]=STABLE_THRESHOLD;
                              ADXL345Registers[XL345_THRESH_INACT]=NOMOVEMENT_THRESHOLD;
                              ADXL345Registers[XL345_TIME_INACT]=NOMOVEMENT_TIME;
                              ADXL345Registers[XL345_ACT_INACT_CTL]=XL345_INACT_Z_ENABLE | 
XL345_INACT_Y_ENABLE | XL345_INACT_X_ENABLE | XL345_INACT_AC | XL345_ACT_Z_ENABLE | XL345_ACT_Y_ENABLE | 
XL345_ACT_X_ENABLE | XL345_ACT_AC;
                              xl345Write(4, XL345_THRESH_ACT, &ADXL345Registers[XL345_THRESH_ACT]);
                        }
                        else  // Delta vector sum does not exceed the threshold
                        {
                              DetectionStatus=0xF0;  // Go to Status "F0", restar
                              putchar(DetectionStatus);
                              ADXL345Registers[XL345_THRESH_ACT]=STRIKE_THRESHOLD;
                              ADXL345Registers[XL345_THRESH_INACT]=NOMOVEMENT_THRESHOLD;
                              ADXL345Registers[XL345_TIME_INACT]=STABLE_TIME;
                              ADXL345Registers[XL345_ACT_INACT_CTL]=XL345_INACT_Z_ENABLE | 
XL345_INACT_Y_ENABLE | XL345_INACT_X_ENABLE | XL345_INACT_AC | XL345_ACT_Z_ENABLE | XL345_ACT_Y_ENABLE | 
XL345_ACT_X_ENABLE | XL345_ACT_DC;
                              xl345Write(4, XL345_THRESH_ACT, &ADXL345Registers[XL345_THRESH_ACT]);
                        }
                  }
                  else if(DetectionStatus==0xF4)  // Wait for long time motionless, and now it is detected
                  {
                        DetectionStatus=0xF5;  // Valid critical fall detection
                        putchar(DetectionStatus);
                        ADXL345Registers[XL345_THRESH_ACT]=STRIKE_THRESHOLD;
                        ADXL345Registers[XL345_THRESH_INACT]=NOMOVEMENT_THRESHOLD;
                        ADXL345Registers[XL345_TIME_INACT]=STABLE_TIME;
                        ADXL345Registers[XL345_ACT_INACT_CTL]=XL345_INACT_Z_ENABLE | XL345_INACT_Y_ENABLE 
| XL345_INACT_X_ENABLE | XL345_INACT_AC | XL345_ACT_Z_ENABLE | XL345_ACT_Y_ENABLE | XL345_ACT_X_ENABLE | 
XL345_ACT_DC;
                        xl345Write(4, XL345_THRESH_ACT, &ADXL345Registers[XL345_THRESH_ACT]);
                        DetectionStatus=0xF0;  // Go to Status "F0", restart
                        putchar(DetectionStatus);
                  }
            }
            else if((ADXL345Registers[XL345_INT_SOURCE]&XL345_FREEFALL)==XL345_FREEFALL)  // Free fall 
interrupt asserted
            {
                  if(DetectionStatus==0xF0)  // Waiting for weightless, and now it is detected
                  {
                        DetectionStatus=0xF1;  // Go to Status "F1"
                        putchar(DetectionStatus);
                        ADXL345Registers[XL345_THRESH_ACT]=STRIKE_THRESHOLD;
                        ADXL345Registers[XL345_THRESH_INACT]=NOMOVEMENT_THRESHOLD;
                        ADXL345Registers[XL345_TIME_INACT]=STABLE_TIME;
                        ADXL345Registers[XL345_ACT_INACT_CTL]=XL345_INACT_Z_ENABLE | XL345_INACT_Y_ENABLE 
| XL345_INACT_X_ENABLE | XL345_INACT_AC | XL345_ACT_Z_ENABLE | XL345_ACT_Y_ENABLE | XL345_ACT_X_ENABLE | 
XL345_ACT_DC;
                        xl345Write(4, XL345_THRESH_ACT, &ADXL345Registers[XL345_THRESH_ACT]);
                        IRQEN|=GP_TIMER_BIT;  // Enable ADuC7026's Timer1 interrupt
                        TimerWaitForStrike=0;
                        TimerFreeFall=0;
                  }
                  else if(DetectionStatus==0xF1)  // Waiting for strike after weightless, and now a new 
free fall is detected
                  {
                        if(TimerWaitForStrike<FREE_FALL_INTERVAL)  // If the free fall interrupt is 
continuously assert within the time of "FREE_FALL_INTERVAL", 
                  {      // then it is considered a continuous free fall
                        TimerFreeFall=TimerFreeFall+TimerWaitForStrike;
                  }
                  else  // Not a continuous free fall
                  {
                        TimerFreeFall=0;
                  }
                  TimerWaitForStrike=0;
                  if(TimerFreeFall>=FREE_FALL_OVERTIME)  // If the continuous time of free fall is longer 
than "FREE_FALL_OVERTIME"
                  {      // Consider that a free fall from high place is detected
                        DetectionStatus=0xFF;
                        putchar(DetectionStatus);
                        ADXL345Registers[XL345_THRESH_ACT]=STRIKE_THRESHOLD;
                        ADXL345Registers[XL345_THRESH_INACT]=NOMOVEMENT_THRESHOLD;
                        ADXL345Registers[XL345_TIME_INACT]=STABLE_TIME;
                        ADXL345Registers[XL345_ACT_INACT_CTL]=XL345_INACT_Z_ENABLE | XL345_INACT_Y_ENABLE 
| XL345_INACT_X_ENABLE | XL345_INACT_AC | XL345_ACT_Z_ENABLE | XL345_ACT_Y_ENABLE | XL345_ACT_X_ENABLE | 
XL345_ACT_DC;
                        xl345Write(4, XL345_THRESH_ACT, &ADXL345Registers[XL345_THRESH_ACT]);
                        DetectionStatus=0xF0;
                        putchar(DetectionStatus);
                  }
            }
            else
            {
                    TimerFreeFall=0;
             }
      }
      IRQEN  |=SPM4_IO_BIT;  // Enable ADuC7026's external interrupt
      }
}

void  main(void)
{
      ADuC7026_Initiate();        // ADuC7026 initialization
      ADXL345_Initiate();         // ADXL345 initialization
      DetectionStatus=0xF0;       // Clear detection status, start
      InitialStatus[0]=0x1000;    // X axis=0g, unsigned short int, 13 bit resolution, 0x1000 = 4096 = 0g, 
+/-0xFF = +/-256 = +/-1g
      InitialStatus[1]=0x0F00;    // Y axis=-1g
      InitialStatus[2]=0x1000;    // Z axis=0g
      IRQEN =SPM4_IO_BIT;         // Enable ADuC7026's external interrupt, to receive the interrupt from 
ADXL345 INT0
      while(1)                    // Endless loop, wait for interrupts
      {
             ;
      }
}
```

FallDetection.h

```
#include "ADuC7026Driver.h"
#include "xl345.h"
#include "xl345_io.h"

// Definitions used for Fall Detection Algorithm
#define        STRIKE_THRESHOLD                     0x20    //62.5mg/LSB, 0x20=2g
#define        STRIKE_WINDOW                        0x0A    //20ms/LSB, 0x0A=10=200ms
#define        STABLE_THRESHOLD                     0x08    //62.5mg/LSB, 0x10=0.5g
#define        STABLE_TIME                          0x02    //1s/LSB, 0x02=2s
#define        STABLE_WINDOW                        0xAF    //20ms/LSB, 0xAF=175=3.5s
#define        NOMOVEMENT_THRESHOLD                 0x03    //62.5mg/LSB, 0x03=0.1875g
#define        NOMOVEMENT_TIME                      0x0A    //1s/LSB, 0x0A=10s
#define        FREE_FALL_THRESHOLD                  0x0C    //62.5mg/LSB, 0x0C=0.75g
#define        FREE_FALL_TIME                       0x06    //5ms/LSB, 0x06=30ms
#define        FREE_FALL_OVERTIME                   0x0F    //20ms/LSB, 0x0F=15=300ms
#define        FREE_FALL_INTERVAL                   0x05    //20ms/LSB, 0x05=100ms
#define        DELTA_VECTOR_SUM_THRESHOLD           0x7D70  //1g=0xFF, 0x7D70=0.7g^2

// Variables used for Fall Detection Algorithm
unsigned char    DetectionStatus;    // Detection status:
                                     //    0xF0:    Start
                                     //    0xF1:    Weightlessness
                                     //    0xF2:    Strike after weightlessness
                                     //    0xF3:    Stable after strike, valid fall detection
                                     //    0xF4:    Long time motionless, valid critical fall detection
                                     //    0xFF:    Continuous free fall, free fall from a high place
unsigned char    TimerWaitForStable;  // Counter of time that wait for stable after strike
unsigned char    TimerWaitForStrike;  // Counter of time that wait for strike after weightless
unsigned char    TimerFreeFall;       // Counter of continuous time for free fall

unsigned short int    InitialStatus[3];     // Initial status for X-, Y-, Z- axis
unsigned short int    Acceleration[3];      // Acceleration for X-, Y-, Z- axis
unsigned long int     DeltaAcceleration[3]; // Acceleration[] - Initial_Status[]
unsigned long int     DeltaVectorSum;       // Vector sum of the DeltaAcceleration[]

BYTE ADXL345Registers[57];    // ADXL345 registers array, total 57 registers in ADXL345

// Implementation of the read function based ADuC7026
void xl345Read(unsigned char count, unsigned char regaddr, unsigned char *buf)
{
       BYTE r;
       WORD RegisterAddress;
       for (r=0;r<count;r++)  // Read the register
       {
              RegisterAddress = regaddr+r;
              WriteData[0] = RegisterAddress;
              ReadViaI2C(XL345_ALT_ADDR, 0, 1);
              buf[r] = ReadData[0];
       }
}

// Implementation of the write function based ADuC7026
void xl345Write(unsigned char count, unsigned char regaddr, unsigned char *buf)
{
       BYTE r;
       WORD RegisterAddress;
       for (r=0;r<count;r++)  // Write the register
       {
              RegisterAddress = regaddr+r;
              WriteData[0] = RegisterAddress;
              WriteData[1] = buf[r];
              WriteViaI2C(XL345_ALT_ADDR, 0, 1);
       }
}

void ADXL345_Initiate()    // ADXL345 initialization, refer to ADXL345 data sheet
{
       xl345Read(1, XL345_DEVID, &ADXL345Registers[XL345_DEVID]);
       //putchar(ADXL345Registers[XL345_DEVID]);    //byte
       ADXL345Registers[XL345_OFSX]=0xFF;
       ADXL345Registers[XL345_OFSY]=0x05;
       ADXL345Registers[XL345_OFSZ]=0xFF;
       xl345Write(3, XL345_OFSX, &ADXL345Registers[XL345_OFSX]);
       ADXL345Registers[XL345_THRESH_ACT]=STRIKE_THRESHOLD;
       ADXL345Registers[XL345_THRESH_INACT]=NOMOVEMENT_THRESHOLD;
       ADXL345Registers[XL345_TIME_INACT]=STABLE_TIME;
       ADXL345Registers[XL345_ACT_INACT_CTL]=XL345_INACT_Z_ENABLE|XL345_INACT_Y_ENABLE | XL345_INACT_X_ENABLE
| XL345_INACT_AC | XL345_ACT_Z_ENABLE|XL345_ACT_Y_ENABLE | XL345_ACT_X_ENABLE | XL345_ACT_DC;
       ADXL345Registers[XL345_THRESH_FF]=FREE_FALL_THRESHOLD;
       ADXL345Registers[XL345_TIME_FF]=FREE_FALL_TIME;
       xl345Write(6, XL345_THRESH_ACT, &ADXL345Registers[XL345_THRESH_ACT]);
       ADXL345Registers[XL345_BW_RATE]=XL345_RATE_100;
       ADXL345Registers[XL345_POWER_CTL]=XL345_STANDBY;
       ADXL345Registers[XL345_INT_ENABLE]=XL345_ACTIVITY | XL345_INACTIVITY | XL345_FREEFALL;
       ADXL345Registers[XL345_INT_MAP]=0x00;
       xl345Write(4, XL345_BW_RATE, &ADXL345Registers[XL345_BW_RATE]);
       ADXL345Registers[XL345_DATA_FORMAT]=XL345_FULL_RESOLUTION | XL345_DATA_JUST_RIGHT | XL345_RANGE_16G;
       xl345Write(1, XL345_DATA_FORMAT, &ADXL345Registers[XL345_DATA_FORMAT]);
       ADXL345Registers[XL345_POWER_CTL]=XL345_MEASURE;
       xl345Write(1, XL345_POWER_CTL, &ADXL345Registers[XL345_POWER_CTL]);
       xl345Read(1, XL345_INT_SOURCE, &ADXL345Registers[XL345_INT_SOURCE]);
}
```

ADuC7026Driver.h

```
#include <ADuC7026.h>

// Definitions of data type
#define BYTE    unsigned char        // 8_bits
#define WORD    unsigned short int   // 16_bits
#define DWORD   unsigned long int    // 32_bits

#define ADXL345_I2C_ADDRESS_SELECT    0x40    // GPIO:P4.0, to select the ADXL345's I2C address

// Variables for I²C operation, to implement burst read/write based ADuC7026, maximum number to burst
read/write is 8 bytes
BYTE Steps, Status;
BYTE ReadData[8], WriteData[9];

// Rewrite the putchar() function, send one byte data via UART
int putchar(int ch)
{
       COMTX=ch;
       while(!(0x020==(COMSTA0 & 0x020)))
       {;}
       return ch;
}

//GPIO Control functions
void OutputBit(BYTE GPIONum, BYTE Data)    // Write the pin of "GPIONum" with "Data" (0 or 1)
{
       DWORD Temp;
       Temp=1<<(GPIONum&0x0F);
       switch(GPIONum>>4)
       {
              case 0:
                     GP0DAT|=(Temp<<24);
                     if(Data==0)
                     {
                            GP0CLR=(Temp<<16);
                     }
                     else
                     {
                            GP0SET=(Temp<<16);
                     }
                     break;
              case 1:
                     GP1DAT|=(Temp<<24);
                     if(Data==0)
                     {
                            GP1CLR=(Temp<<16);
                     }
                     else
                     {
                        GP1SET=(Temp<<16);
                  }
                  break;
            case 2:
                  GP2DAT|=(Temp<<24);
                  if(Data==0)
                  {
                        GP2CLR=(Temp<<16);
                  }
                  else
                  {
                        GP2SET=(Temp<<16);
                  }
                  break;
            case 3:
                  GP3DAT|=(Temp<<24);
                  if(Data==0)
                  {
                        GP3CLR=(Temp<<16);
                  }
                  else
                  {
                        GP3SET=(Temp<<16);
                  }
                  break;
            case 4:
                  GP4DAT|=(Temp<<24);
                  if(Data==0)
                  {
                        GP4CLR=(Temp<<16);
                  }
                  else
                  {
                        GP4SET=(Temp<<16);
                  }
                  break;
      }
}

// ADuC7026 initialization
void UART_Initiate()  // ADuC7026 UART initialization, initiate the UART Port to 115200bps
{
      POWKEY1 = 0x01;         // Start PLL setting,changeless
      POWCON=0x00;
      POWKEY2 = 0xF4;         // Finish PLL setting,changeless
      GP1CON = 0x2211;        // I2C on P1.2 and P1.3. Setup tx & rx pins on P1.0 and P1.1 for UART
      COMCON0 = 0x80;         // Setting DLAB
      COMDIV0 = 0x0B;         // Setting DIV0 and DIV1 to DL calculated
      COMDIV1 = 0x00;
      COMCON0 = 0x07;         // Clearing DLAB
      COMDIV2 = 0x883E;       // Fractional divider
```

```
                                     // M=1
                                     // N=01101010101=853
                                     // M+N/2048=1.4165
                                     // 41.78MHz/(16*2*2^CD*DL*(M+N/2048))  //CD=0  DL=0B=11
                                     // 115.2Kbps  M+N/2048 =1.0303  M=1, N=62=0x3EH=000 0011 1110
                                     //comdiv2=0x883E
}

void I2C1_Initiate()  // ADuC7026 I2C1 initialization, initiate the I2C1 Port to 100kbps
{
      GP1CON = 0x2211;        // I2C on P1.2 and P1.3. Setup tx & rx pins on P1.0 and P1.1 for UART
      I2C1CFG = 0x82;         // Master Enable & Enable Generation of Master Clock
      I2C1DIV = 0x3232;       // 0x3232 = 400kHz
                              // 0xCFCF = 100kHz
      FIQEN |= SM_MASTER1_BIT;    //Enable I2C1 Master Interupt
}

void Timer1_Initiate()  // ADuC7026 Timer1 initialization, Interval = 20ms
{
      T1LD = 0xCC010;
      T1CON = 0xC0;
}

void ADuC7026_Initiate(void)  // ADuC7026 initialization, initiate the UART, I2C1, Timer1, and GPIOs
{
      UART_Initiate();
      I2C1_Initiate() ;
      Timer1_Initiate();
      OutputBit(ADXL345_I2C_ADDRESS_SELECT,0); //Grounding the SDO (p4.0), I2C address for writing and
reading is 0xA6 and 0xA7
}

// ADuC7026 I2C1 Master, implement burst read/write based ADuC7026, maximum number to burst read/write is 8
bytes
// support 1 byte address and dual byte address
// enable I2C1 interrupt as FIQ interrupt, burst read/write is realized in the FIQ interrupt

void WriteViaI2C(BYTE DeviceAddr, BYTE AddrType, BYTE NumberOfWriteBytes)
// Write "NumberOfWriteBytes" data to "DeviceAddr" address
// AddrType=0, single-byte address;  AddrType=1, dual byte address
// Data to write is saved in "WriteData[]"
{
      Status=0;
      Steps=NumberOfWriteBytes+AddrType+1;
      I2C1ADR = DeviceAddr<<1;
      I2C1CNT=NumberOfWriteBytes+AddrType-1;
      I2C1MTX = WriteData[Status];
      while(Steps != Status)
      {
            ;
      }
}
```

```
void ReadViaI2C(BYTE DeviceAddr, BYTE AddrType, BYTE NumberOfReadBytes)
// Read "NumberOfWriteBytes" data from "DeviceAddr" address
// AddrType=0, single byte address;  AddrType=1, dual byte address
// Readback data is saved in "ReadData[]"
{
       Status=0;
       Steps=AddrType+1;
       I2C1ADR = DeviceAddr<<1;
       I2C1MTX = WriteData[Status];
       while(Steps != Status)
       {
              ;
       }
       Status=0;
       Steps=NumberOfReadBytes;
       I2C1CNT=NumberOfReadBytes-1;
       I2C1ADR = (DeviceAddr<<1)+1;
       while(Steps != Status)
       {
              ;
       }
}

void FIQ_Handler() __fiq    // FIQ interrupt
{
       // ADuC7026 Transmit
       if(((I2C1MSTA & 0x4) == 0x4) && (Status < (Steps-1)) )
       {
              Status++;
              I2C1MTX = WriteData[Status];
       }
       else if(((I2C1MSTA & 0x4) == 0x4) && (Status == (Steps-1)))
       {
              Status ++;
       }
       // ADuC7026 Receive
       else if (((I2C1MSTA & 0x8) == 0x8) && (Status <= (Steps-1)))
       {
              ReadData[Status] =  I2C1MRX;
              Status ++;
       }
}
```

xl345.h

```
/*----------------------------------------------------------------------
  The present firmware, which is for guidance only, aims at providing
  customers with coding information regarding their products in order
  for them to save time.  As a result, Analog Devices shall not be
  held liable for any direct, indirect, or consequential damages with
  respect to any claims arising from the content of such firmware and/or
  the use made by customers of the coding information contained herein
  in connection with their products.
----------------------------------------------------------------------*/

#ifndef __XL345_H
#define __XL345_H

/* --- I2C addresses --- */
/* The primary slave address is used when the SDO pin is tied or pulled
   high.  The alternate address is selected when the SDO pin is tied or
   pulled low.  When building the hardware, if you intend to use I2C,
   the state of the SDO pin must be set.  The SDO pin is also used for
   SPI communication.  To save system power, there is no internal pull-up
   or pull-down.                                                        */
#define XL345_SLAVE_ADDR    0x1d
#define XL345_ALT_ADDR      0x53
/* additional I2C defines for communications functions that need the
   address shifted with the read/write bit appended                   */
#define XL345_SLAVE_READ    XL345_SLAVE_ADDR << 1 | 0x01
#define XL345_SLAVE_WRITE   XL345_SLAVE_ADDR << 1 | 0x00
#define XL345_ALT_READ      XL345_ALT_ADDR << 1 | 0x01
#define XL345_ALT_WRITE     XL345_ALT_ADDR << 1 | 0x00

/* ------- Register names ------- */
#define XL345_DEVID           0x00
#define XL345_RESERVED1       0x01
#define XL345_THRESH_TAP      0x1d
#define XL345_OFSX            0x1e
#define XL345_OFSY            0x1f
#define XL345_OFSZ            0x20
#define XL345_DUR             0x21
#define XL345_LATENT          0x22
#define XL345_WINDOW          0x23
#define XL345_THRESH_ACT      0x24
#define XL345_THRESH_INACT    0x25
#define XL345_TIME_INACT      0x26
#define XL345_ACT_INACT_CTL   0x27
#define XL345_THRESH_FF       0x28
#define XL345_TIME_FF         0x29
#define XL345_TAP_AXES        0x2a
#define XL345_ACT_TAP_STATUS  0x2b
#define XL345_BW_RATE         0x2c
#define XL345_POWER_CTL       0x2d
#define XL345_INT_ENABLE      0x2e
```

```
#define XL345_INT_MAP         0x2f
#define XL345_INT_SOURCE      0x30
#define XL345_DATA_FORMAT     0x31
#define XL345_DATAX0          0x32
#define XL345_DATAX1          0x33
#define XL345_DATAY0          0x34
#define XL345_DATAY1          0x35
#define XL345_DATAZ0          0x36
#define XL345_DATAZ1          0x37
#define XL345_FIFO_CTL        0x38
#define XL345_FIFO_STATUS     0x39

/*----------------------------------------------------------------------
  Bit field definitions and register values
  ----------------------------------------------------------------------*/
//#define XL345_
/* register values for DEVID                                            */
/* The device ID should always read this value, The customer does not
  need to use this value but it can be read to check that the
  device can communicate                                                */

#define XL345_ID              0xe5

/* Reserved soft reset value                                            */
#define XL345_SOFT_RESET      0x52

/* Registers THRESH_TAP through TIME_INACT take only 8-bit values
   There are no specific bit fields in these registers                  */

/* Bit values in ACT_INACT_CTL                                          */
#define XL345_INACT_Z_ENABLE  0x01
#define XL345_INACT_Z_DISABLE 0x00
#define XL345_INACT_Y_ENABLE  0x02
#define XL345_INACT_Y_DISABLE 0x00
#define XL345_INACT_X_ENABLE  0x04
#define XL345_INACT_X_DISABLE 0x00
#define XL345_INACT_AC        0x08
#define XL345_INACT_DC        0x00
#define XL345_ACT_Z_ENABLE    0x10
#define XL345_ACT_Z_DISABLE   0x00
#define XL345_ACT_Y_ENABLE    0x20
#define XL345_ACT_Y_DISABLE   0x00
#define XL345_ACT_X_ENABLE    0x40
#define XL345_ACT_X_DISABLE   0x00
#define XL345_ACT_AC          0x80
#define XL345_ACT_DC          0x00

/* Registers THRESH_FF and TIME_FF take only 8-bit values
   There are no specific bit fields in these registers                  */

/* Bit values in TAP_AXES                                               */
```

```
#define XL345_TAP_Z_ENABLE    0x01
#define XL345_TAP_Z_DISABLE   0x00
#define XL345_TAP_Y_ENABLE    0x02
#define XL345_TAP_Y_DISABLE   0x00
#define XL345_TAP_X_ENABLE    0x04
#define XL345_TAP_X_DISABLE   0x00
#define XL345_TAP_SUPPRESS    0x08

/* Bit values in ACT_TAP_STATUS                                        */
#define XL345_TAP_Z_SOURCE    0x01
#define XL345_TAP_Y_SOURCE    0x02
#define XL345_TAP_X_SOURCE    0x04
#define XL345_STAT_ASLEEP     0x08
#define XL345_ACT_Z_SOURCE    0x10
#define XL345_ACT_Y_SOURCE    0x20
#define XL345_ACT_X_SOURCE    0x40

/* Bit values in BW_RATE                                               */
/* Expresed as output data rate */
#define XL345_RATE_3200       0x0f
#define XL345_RATE_1600       0x0e
#define XL345_RATE_800        0x0d
#define XL345_RATE_400        0x0c
#define XL345_RATE_200        0x0b
#define XL345_RATE_100        0x0a
#define XL345_RATE_50         0x09
#define XL345_RATE_25         0x08
#define XL345_RATE_12_5       0x07
#define XL345_RATE_6_25       0x06
#define XL345_RATE_3_125      0x05
#define XL345_RATE_1_563      0x04
#define XL345_RATE__782       0x03
#define XL345_RATE__39        0x02
#define XL345_RATE__195       0x01
#define XL345_RATE__098       0x00

/* Expressed as output bandwidth */
/* Use either the bandwidth or rate code,
   whichever is more appropriate for your application */
#define XL345_BW_1600         0x0f
#define XL345_BW_800          0x0e
#define XL345_BW_400          0x0d
#define XL345_BW_200          0x0c
#define XL345_BW_100          0x0b
#define XL345_BW_50           0x0a
#define XL345_BW_25           0x09
#define XL345_BW_12_5         0x08
#define XL345_BW_6_25         0x07
#define XL345_BW_3_125        0x06
#define XL345_BW_1_563        0x05
#define XL345_BW__782         0x04
```

```
#define XL345_BW__39          0x03
#define XL345_BW__195         0x02
#define XL345_BW__098         0x01
#define XL345_BW__048         0x00

#define XL345_LOW_POWER       0x08
#define XL345_LOW_NOISE       0x00
/* Bit values in POWER_CTL                                               */
#define XL345_WAKEUP_8HZ           0x00
#define XL345_WAKEUP_4HZ           0x01
#define XL345_WAKEUP_2HZ           0x02
#define XL345_WAKEUP_1HZ           0x03
#define XL345_SLEEP                0x04
#define XL345_MEASURE              0x08
#define XL345_STANDBY              0x00
#define XL345_AUTO_SLEEP           0x10
#define XL345_ACT_INACT_SERIAL     0x20
#define XL345_ACT_INACT_CONCURRENT 0x00

/* Bit values in INT_ENABLE, INT_MAP, and INT_SOURCE are identical.
   Use these bit values to read or write any of these registers.        */
#define XL345_OVERRUN              0x01
#define XL345_WATERMARK            0x02
#define XL345_FREEFALL             0x04
#define XL345_INACTIVITY           0x08
#define XL345_ACTIVITY             0x10
#define XL345_DOUBLETAP            0x20
#define XL345_SINGLETAP            0x40
#define XL345_DATAREADY            0x80

/* Bit values in DATA_FORMAT                                             */

/* Register values read in DATAX0 through DATAZ1 are dependent on the
   value specified in data format.  Customer code will need to interpret
   the data as desired.                                                  */
#define XL345_RANGE_2G             0x00
#define XL345_RANGE_4G             0x01
#define XL345_RANGE_8G             0x02
#define XL345_RANGE_16G            0x03
#define XL345_DATA_JUST_RIGHT      0x00
#define XL345_DATA_JUST_LEFT       0x04
#define XL345_10BIT                0x00
#define XL345_FULL_RESOLUTION      0x08
#define XL345_INT_LOW              0x20
#define XL345_INT_HIGH             0x00
#define XL345_SPI3WIRE             0x40
#define XL345_SPI4WIRE             0x00
#define XL345_SELFTEST             0x80

/* Bit values in FIFO_CTL                                                */
/* The low bits are a value 0 to 31 used for the watermark or the number
```

```
   of pre-trigger samples when in triggered mode                        */
#define XL345_TRIGGER_INT1         0x00
#define XL345_TRIGGER_INT2         0x20
#define XL345_FIFO_MODE_BYPASS     0x00
#define XL345_FIFO_RESET           0x00
#define XL345_FIFO_MODE_FIFO       0x40
#define XL345_FIFO_MODE_STREAM     0x80
#define XL345_FIFO_MODE_TRIGGER    0xc0

/* Bit values in FIFO_STATUS                                            */
/* The low bits are a value 0 to 32 showing the number of entries
   currently available in the FIFO buffer                               */

#define XL345_FIFO_TRIGGERED       0x80


#endif /* __XL345_H */
```

xl345_io.h

```
/*---------------------------------------------------------------------
  The present firmware, which is for guidance only, aims at providing
  customers with coding information regarding their products in order
  for them to save time.  As a result, Analog Devices shall not be
  held liable for any direct, indirect, or consequential damages with
  respect to any claims arising from the content of such firmware and/or
  the use made by customers of the coding information contained herein
  in connection with their products.
---------------------------------------------------------------------*/
#ifndef __XL345_IO_H
#define __XL345_IO_H
#include "XL345.h"
/* Wrapper functions for reading and writing bursts to / from the ADXL345
   These can use I2C or SPI.  Will need to be modified for your hardware
*/

/*
  The read function takes a byte count, a register address, and a
  pointer to the buffer where to return the data.  When the read
  function runs in I2C as an example, it goes through the following
  sequence:
    1) I2C start
    2) Send the correct I2C slave address + write
    3) Send the register address
    4) I2C stop
    6) I2C start
    7) Send the correct I2C slave address + read
    8) I2C read for each byte but the last one + ACK
    9) I2C read for the last byte + NACK
   10) I2C stop
*/
void xl345Read(unsigned char count, unsigned char regaddr, unsigned char *buf);
/*
  The write function takes a byte count and a pointer to the buffer
  with the data.  The first byte of the data should be the start
  register address, the remaining bytes will be written starting at
  that register.  The minimum byte count that should be passed is 2,
  one byte of address, followed by a byte of data.  Multiple
  sequential registers can be written with longer byte counts. When
  the write function runs in I2C as an example, it goes through the
  following sequence:
    1) I2C start
    2) Send the correct I2C slave address + write
    3) Send the number of bytes requested form the buffer
    4) I2C stop
*/
void xl345Write(unsigned char count, unsigned char regaddr, unsigned char *buf);
#endif
```

CONCLUSION

The ADXL345 is a powerful and full-featured accelerometer from Analog Devices. This application note takes advantage of the various built-in motion status detection features and flexible interrupts to propose a new solution for fall detection. This solution is realized through full use of the ADXL345 hardware interrupts and has been tested to feature low algorithm complexity with high detection accuracy.

REFERENCES

ADXL345 Data Sheet. Analog Devices, Inc. 2009.

NOTES

NOTES