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

基于3轴加速度计ADXL345的跌倒检测应用

作者：Ning Jia

简介

老人因自我照顾及保护能力的下降，经常会意外跌倒。如果不及时给予救助，这类事故可能造成严重的后果。统计显示，多数严重后果并不是直接源自跌倒本身，而是跌倒后在救助及处理方面的延迟所致。发生跌倒意外时，如果可以及时通知救援人员，则可大幅降低跌倒后的危险程度。有鉴于此，市场上已开发出大量跌倒检测及预测产品。

近年来，MEMS加速度传感器取得的技术进步为基于3轴加速度传感器设计跌倒检测器创造了可能。这类跌倒检测器基于以下操作原理：通过跟踪传感器佩戴者在三个正交方向的加速度变化，来检测其体位变化。然后以算术方法对数据进行分析，以确定佩戴者的身体是否跌倒。如果佩戴者跌倒，设备将通过GPS模块和无线发射器模块进行定位，并发出警报求助。可见，跌倒检测器的核心部分是判断是否存在紧急跌倒状况的检测原理和算法。

ADXL345是ADI出品的最新3轴数字输出加速度计，完全适用于跌倒检测器应用。本应用笔记以个人身体跌倒检测的原理研究为基础，提出一种基于ADXL345的全新跌倒状况检测解决方案。

ADXL345 MEMS加速度计

微电子机械系统(MEMS)是一种将微机械结构和电路集成在单硅芯片的半导体技术。MEMS加速度计是基于这种技术的一种传感器，旨在实现对单轴、双轴和三轴情况下加速度的感知。根据具体应用，加速度计能够提供不同范围的检测能力，从数*g*到数十*g*，数字或模拟输出，甚至可能具有多种中断模式。这些特点可为用户提供更加方便、更加灵活的解决方案。

ADXL345是来自ADI的数字输出的最新MEMS 3轴加速度计。该器件提供±2 *g*、±4 *g*、±8 *g*和±16 *g*四种可选测量范围；最高达13位的分辨率；固定4 mg/LSB灵敏度；3 mm × 5 mm × 1 mm 超小封装；40 μA至145 μA超低功耗；标准I^2C和SPI数字接口；32-级FIFO存储；多种内置运动状态检测选项；以及灵活的中断系统。这些功能有利于大幅简化跌倒检测算法，从而使ADXL345成为跌倒检测器应用的理想加速度计。本应用笔记提出的跌倒检测解决方案完全基于ADXL345内置的运动状态检测功能和中断系统，几乎无需访问实际加速度值，无需执行任何其他计算，从而最大限度地降低了算法的复杂程度。

ADXL345的中断系统详见“中断”部分。有关ADXL345的更多规格详情，请参阅数据手册，也可访问www.analog.com。图1和图2分别展示了ADXL345的系统框图和引脚定义。

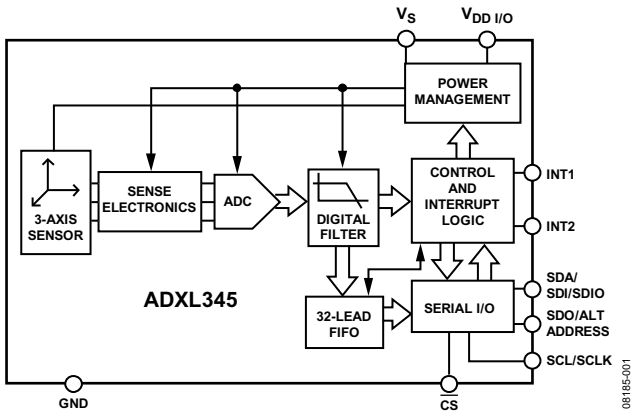

图1. ADXL345系统框图

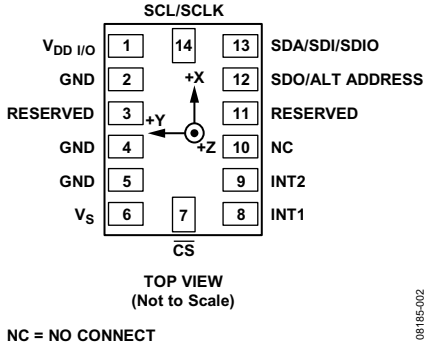

图2. ADXL345的引脚配置

目录

中断

ADXL345 具有两个可编程中断引脚，INT1和INT2，共计八个中断源：DATA_READY、SINGLE_TAP、DOUBLE_TAP、Activity、Inactivity、FREE_FALL、Watermark和Overrun。各个中断均可独立使能或禁用，只需设置INT_ENABLE寄存器中的相应位即可，通过INT_MAP寄存器的相应位来选择映射INT1引脚或INT2引脚。

DATA_READY

当有新的数据产生时，Data_Ready中断置位；当没有新的数据时，Data_Ready中断清除。

SINGLE_TAP

当加速度值超过一定门限(THRESH_TAP)并且持续时间小于一定时间范围(DUR)的时候，Single_Tap中断置位。

DOUBLE_TAP

当第一次Single_Tap事件发生后，在一定时间(LATENT)之后，并在一定时间(WINDOW)之内，又发生第二次Single_Tap事件时，Double _Tap中断置位。

图3所示为有效的SINGLE_TAP及DOUBLE_TAP两种中断。

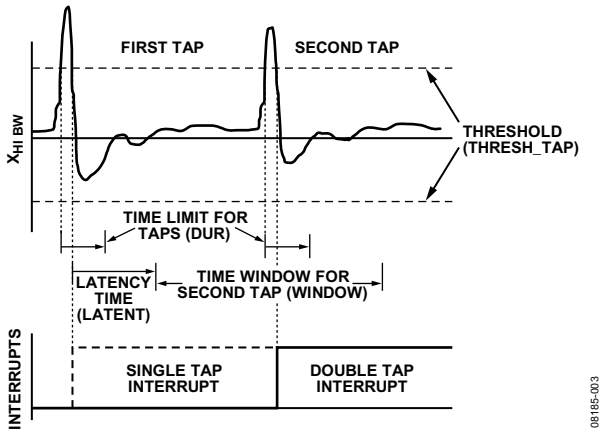

图3. SINGLE_TAP和DOUBLE_TAP中断

Activity

当加速度值超过一定门限(THRESH_ACT)时，Activity中断置位。

Inactivity

当加速度值低于一定门限(THRESH_INACT)并且持续超过一定时间(TIME_INACT)时，Inactivity中断置位。TIME_INACT可以设定的最长时间为255s。

需要指出的是，对于Activity和Inactivity中断，用户可以针对X、Y、Z轴来分别进行使能或禁用。比如，可以只使能X轴的Activity中断，而禁用Y轴和Z轴的Activity中断。

另外，对于Activity和Inactivity中断，用户还可以自由选择DC coupled工作方式或者AC coupled工作方式。其区别在于，DC coupled工作方式下，每个采样点的加速度值将直接与门限(THRESH_ACT或THRESH_INACT)进行比较，来判断是否发生中断；而AC coupled工作方式下，新的采样点将以之前的某个采样点为参考，用两个采样点的差值与门限(THRESH_ACT或THRESH_INACT)进行比较，来判断是否发生中断。AC coupled工作方式下的Activity检测，是选择检测开始时的那一个采样点作为参考，以后每个采样点的加速度值都与参考点进行比较。如果它们的差值超过门限(THRESH_ACT)，则Activity中断置位。AC coupled工作方式下的Inctivity检测，同样要选择一个参考点。如果新采样点与参考点的加速度差值超过门限(THRESH_INACT)，参考点会被该采样点更新。如果新采样点与参考点的加速度差值小于门限(THRESH_INACT)，并且持续超过一定时间(TIME_INACT)，则Inctivity置位。

FREE_FALL

当加速度值低于一定门限(THRESH_FF)并且持续超过一定时间(TIME_FF)时，Free_Fall中断置位。

与Inactivity中断的区别在于，Free_Fall中断主要用于对自由落体运动的检测。因此，X、Y、Z轴总是同时被使能或禁用；其时间设定也比Inactivity中断中要小很多，TIME_FF可以设定的最大值为1.28s；而且Free_Fall中断只能是DC coupled工作方式。

Watermark

当FIFO里所存的采样点超过一定点数(SAMPLES)时，Watermark中断置位。当FIFO里的采样点被读取，使得其中保存的采样点数小于该数值(SAMPLES)时，Watermark中断自动清除。

需要指出的是，ADXL345的FIFO最多可以存储32个采样点(X、Y、Z三轴数值)，且具有Bypass模式、普通FIFO模式、Stream模式和Trigger模式，一共4种工作模式。FIFO功能也是ADXL345的一个重要且十分有用的功能。但是本文后面给出的解决方案中，并没有使用到FIFO功能，所以，在此不做详细介绍。有关FIFO的更多详情，请参阅ADXL345数据手册。

Overrun

当有新采样点更新了未被读取得前次采样点时，Overrun中断置位。

Overrun功能与FIFO的工作模式有关，当FIFO工作在Bypass模式下，如果有新采样点更新了DATAX、DATAY和DATAZ寄存器里的数值，则Overrun中断置位。当FIFO工作在其他三种模式下，只有FIFO被存满32点时，Overrun中断才会置位。FIFO里的采样点被读取后，Overrun中断自动清除。

跌倒过程中的加速度变化特性

关于跌倒检测原理的主要研究集中于人体跌倒过程中的加速度变化特性。图4至图7分别表示下楼、上楼、坐下及从椅子上站起这四种运动中的加速度变化曲线。(跌倒检测器戴在被测人的身上。)

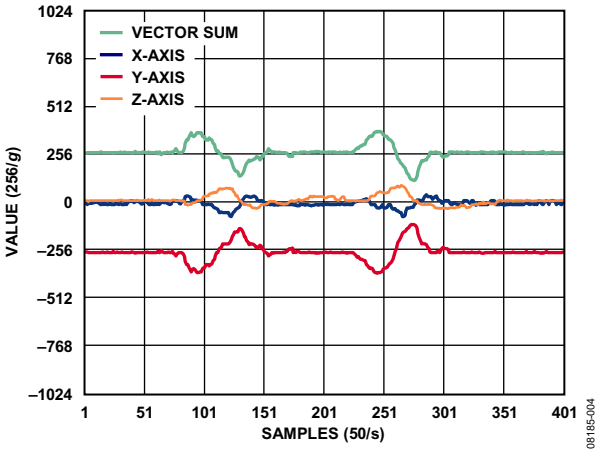

图4. 下楼过程中的加速度变化曲线

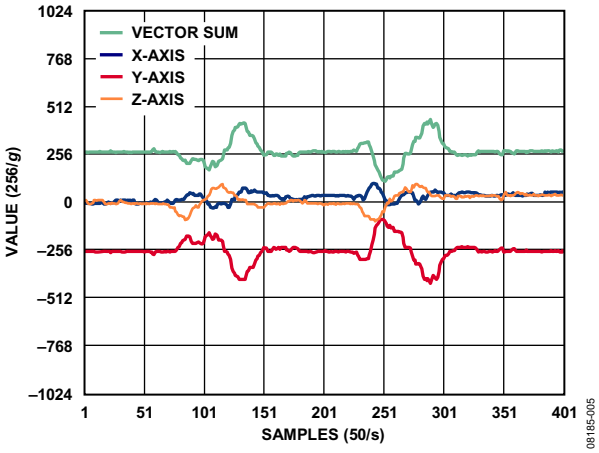

图5. 上楼过程中的加速度变化曲线

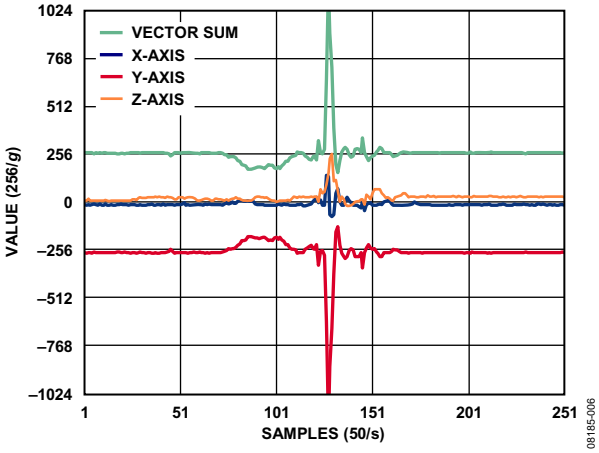

图6. 坐下过程中的加速度变化曲线

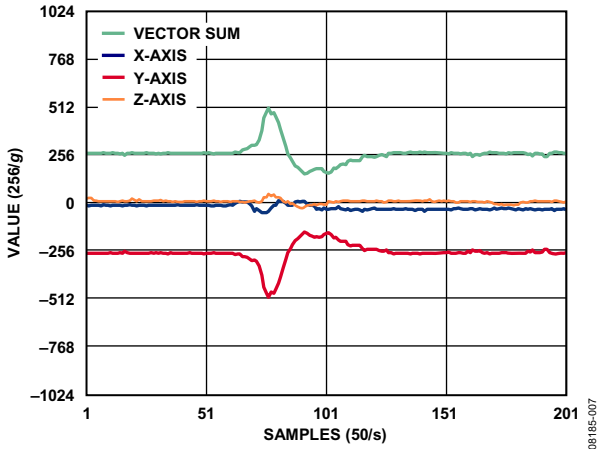

图7. 站起过程中的加速度变化曲线

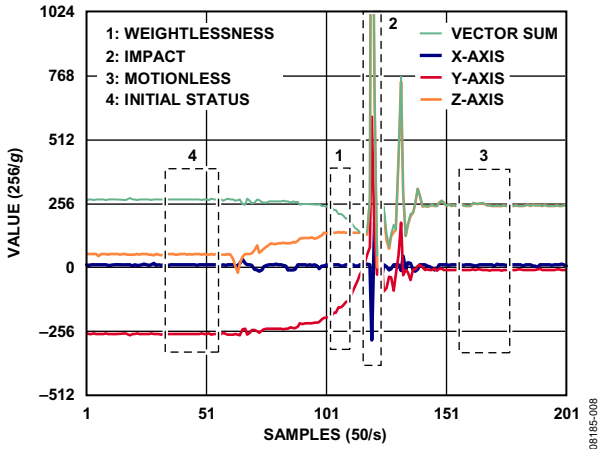

图8. 跌倒过程中的加速度变化曲线

由于老年人的动作相对较慢，因此，在图4和图5所示步行运动中，加速度变化并不是很明显。图8表示跌倒过程中的加速度变化曲线。通过比较图8与图4至图7四幅图，可以发现，跌倒事件有四种关键特性。这四种特性可用作跌倒检测的标准，图8中以方框标示，下面逐一详细解释。

失重

失重现象会发生在跌倒之初。这种现象在自由跌倒过程中尤其显著，加速度的矢量和降至近0 *g*的水平。持续时间取决于自由跌倒的高度。尽管正常跌倒过程中的失重现象不如自由跌倒过程中显著，但其加速度矢量和同样小于1 *g*(正常情况下一般大于1 *g*)。因此，这可以作为跌倒状态的第一个判断依据。可以由ADXL345的Free_Fall中断来检测。

撞击

发生失重现象后，人体与地面相撞击；在图8的加速度曲线中，该现象表现为剧烈冲击。这种冲击由ADXL345的Activity中断检测。因此，确定跌倒的第二个依据是在FREE_FALL中断之后出现的Activity中断。

静止

一般地，跌倒并撞击地面后，人体无法立即站起来。人体会在短时间内保持静止状态。在图8的加速度曲线中，表示为一段平线，由ADXL345的Inactivity中断检测。因此，确定跌倒的第三个依据是在Activity中断之后出现Activity中断。

与初始状态比较

跌倒后，人体会翻转，因此三个轴的加速度与跌倒前的初始状态有所不同。如果将跌倒检测器戴在人体上以取得加速度的初始状态，则可在Inactivity中断之后读取三个轴的加速度数据，然后将采样数据与初始状态进行比较。因此，确定跌倒的第四个依据是采样数据与初始状态之差是否超过一定阈值(如0.7 *g*)。

这四个依据共同构成整个跌倒检测算法，在此基础上，系统可根据跌倒状态报警。中断间的时间间隔必须在一定范围内。正常情况下，FREE_FALL中断(失重)与Activity中断(冲击)之间的时间间隔不应过长，除非从极高处跌落。类似地，Activity中断(冲击)与Inactivity中断(静止)之间的时间间隔也不应过长。“利用ADXL345简化跌倒检测算法”部分给出了一个实例及一组合理的值。相关的中断检测阈值和时间参数可根据需要灵活设置。另外，如果跌倒导致昏迷等严重后果，人体保持静止的时间会更长。这种状态仍然可以通过Inactivity中断检测到。因此，如果跌倒后检测到Inactivity状态持续一定长的时间，则可发出紧急报警。

系统的典型电路连接

ADXL345与MCU之间的电路连接非常简单。在本应用笔记中，我们利用ADXL345和ADuC7026微控制器创建了一个测试平台。图9所示为ADXL345与ADuC7026之间的典型连接。其中，ADXL345的 引脚连接高电平，ADXL345工作于I2C模式。SDA和SCL分别为I^2C总线的数据和时钟，与ADuC7026的对应引脚相连。ADuC7026的一个GPIO连接至ADXL345的SDO/ALT ADDRESS引脚，以选择ADXL345的I2C地址。ADXL345的INT1引脚连接至ADuC7026的一个IRQ输入端，以产生中断信号。其他的单片机或者处理器都可以采用与图9类似的电路与ADXL345进行连接。。为获得更高数据速率，ADXL345也可工作于SPI模式。对于关于SPI连接的示例电路，请参阅ADXL345数据手册。

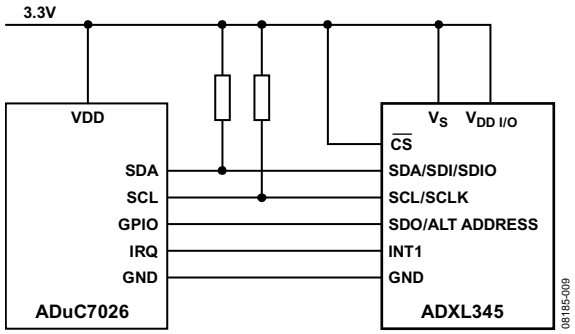

图9. ADXL345与MCU之间的典型电路连接

利用ADXL345简化跌倒检测算法

本节将给出以上解决方案的具体算法实现。

表1列出了各寄存器的功能以及本算法中用到的值。有关各寄存器位的详细定义，请参阅ADXL345数据手册。

请注意，表1中所示的部分寄存器有两个设置值。这表示在算法中会切换使用这两个数值，以实现不同的检测目的。算法流程图如图10所示。

表1. ADXL345寄存器功能描述

十六进制地址	十进制地址	寄存器名称	类型	复位值	描述	算法中的设置	算法中设置的功能
0x00	0	DEVID	只读	0xE5	器件ID	只读	
0x01 至 0x1C	1 至 28	Reserved	保留		保留，不要操作	保留	
0x1D	29	THRESH_TAP	读/写	0x00	Tap的门限	不使用	
0x1E	30	OFSX	读/写	0x00	X-轴失调	0x06	补偿X轴失调，通过初始化校正获得
0x1F	31	OFSY	读/写	0x00	Y-轴失调	0xF9	补偿Y轴失调，通过初始化校正获得
0x20	32	OFSZ	读/写	0x00	Z-轴失调	0xFC	补偿Z轴失调，通过初始化校正获得
0x21	33	DUR	读/写	0x00	Tap的持续时间	不使用	
0x22	34	Latent	读/写	0x00	Tap的延迟时间	不使用	
0x23	35	Window	读/写	0x00	Tap的时间窗	不使用	
0x24	36	THRESH_ACT	读/写	0x00	Activity的门限	0x20/0x08	设置Activity的门限为2g或0.5g
0x25	37	THRESH_INACT	读/写	0x00	Inactivity的门限	0x03	设置Inactivity的门限为0.1875g
0x26	38	TIME_INACT	读/写	0x00	Inactivity的时间	0x02/0x0A	设置Inactivity的时间为2s或10s
0x27	39	ACT_INACT_CTL	读/写	0x00	Activity/Inactivity使能控制	0x7F/0xFF	使能X、Y、Z三轴的Activity和Inactivity功能，其中Inactivity为AC coupled模式，Activity为DC coupled 或 AC coupled模式
0x28	40	THRESH_FF	读/写	0x00	Free-Fall的门限	0x0C	设置Free-Fall的门限为0.75g
0x29	41	TIME_FF	读/写	0x00	Free-Fall的时间	0x06	设置Free-Fall的时间为30ms
0x2A	42	TAP_AXES	读/写	0x00	Tap/Double Tap使能控制	不使用	
0x2B	43	ACT_TAP_STATUS	只读	0x00	Activity/Tap中断轴指示	只读	
0x2C	44	BW_RATE	读/写	0x0A	采样率和功耗模式控制	0x0A	设置采样率为100Hz
0x2D	45	POWER_CTL	读/写	0x00	工作模式控制	0x00	设置为正常工作模式
0x2E	46	INT_ENABLE	读/写	0x00	中断使能控制	0x1C	使能Activity、Inactivity、Free-Fall中断
0x2F	47	INT_MAP	读/写	0x00	中断映射控制	0x00	将所有中断映射到INT1引脚
0x30	48	INT_SOURCE	只读	0x00	中断源指示	只读	
0x31	49	DATA_FORMAT	读/写	0x00	数据格式控制	0x0B	设置为+/-16g测量范围，13bit右对齐模式，中断为高电平触发，使用I2C数据接口
0x32	50	DATAX0	只读	0x00	X-轴数据0	只读	
0x33	51	DATAX1	只读	0x00	X-轴数据1	只读	
0x34	52	DATAY0	只读	0x00	Y-轴数据0	只读	
0x35	53	DATAY1	只读	0x00	Y-轴数据1	只读	
0x36	54	DATAZ0	只读	0x00	Z-轴数据0	只读	
0x37	55	DATAZ1	只读	0x00	Z-轴数据1	只读	
0x38	56	FIFO_CTL	只读	0x00	FIFO控制	不使用	
0x39	57	FIFO_STATUS	只读	0x00	FIFO状态	只读	

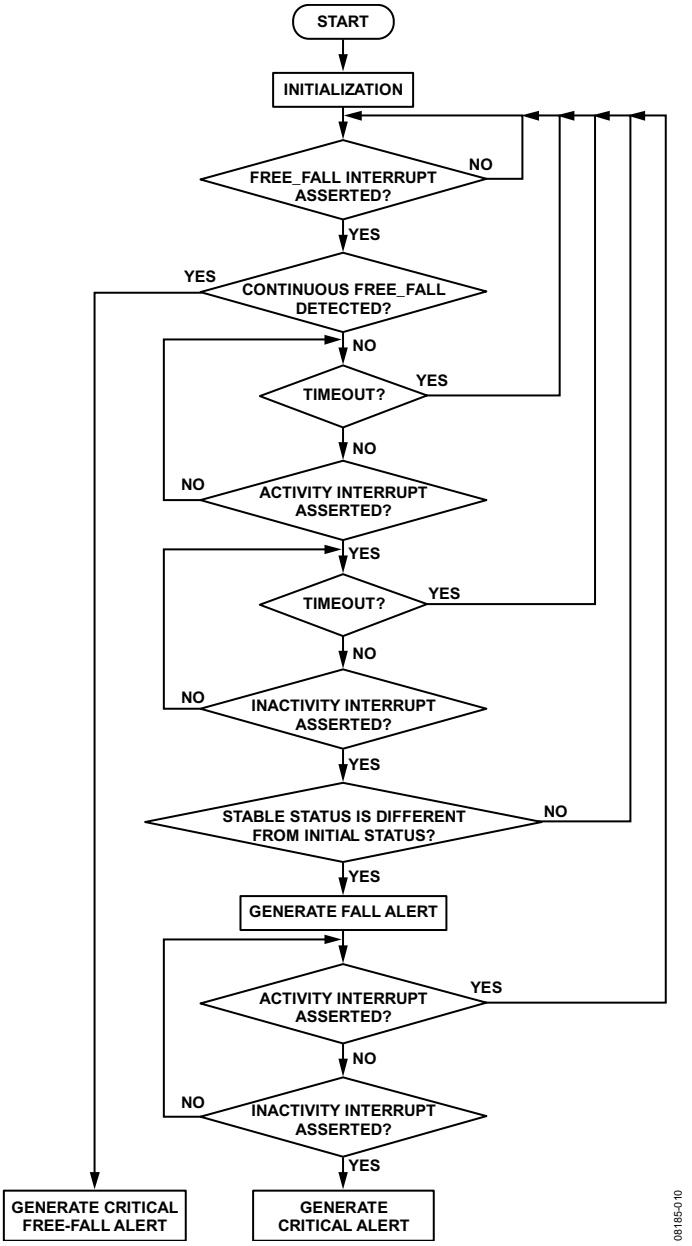
START
INITIALIZATION
FREE_FALL INTERRUPT ASSERTED?
NO
YES
CONTINUOUS FREE_FALL DETECTED?
YES
NO
TIMEOUT?
YES
NO
ACTIVITY INTERRUPT ASSERTED?
NO
YES
TIMEOUT?
YES
NO
INACTIVITY INTERRUPT ASSERTED?
NO
YES
STABLE STATUS IS DIFFERENT FROM INITIAL STATUS?
NO
YES
GENERATE FALL ALERT
ACTIVITY INTERRUPT ASSERTED?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

图10. 算法流程图

算法中的各中断门限和相关时间参数如下：

1. 初始化后，系统等待FREE_FALL中断(失重)。THRESH_FF设为0.75 g，TIME_FF设为30ms。
2. FREE_FALL中断产生之后，系统开始等待Activity中断(冲击)。THRESH_ACT设为2 g，Activity中断为DC coupled工作模式。
3. FREE_FALL中断(失重)与Activity中断(冲击)之间的时间间隔设为200ms。如果该时间间隔大于200ms，则认为无效。200ms计时通过MCU定时器实现。
4. Activity中断产生之后，系统开始等待Inactivity中断(冲击后静止)。THRESH_INACT设为0.1875 g，TIME_INACT设为2s。Inactivity中断为AC coupled工作模式。
5. 在Activity中断产(撞击)生之后的3.5s时间之内，应该有Inactivity中断(撞击后的静止)产生。如果超时，则认为无效。3.5s计时需要通过MCU中的定时器来实现。
6. 如果Inactivity中断之后的加速度值与初始状态(假设已知)下数值的矢量差超过0.7g，则说明检测到一次有效的跌倒，系统会给出一个报警。
7. 当检测到跌倒状态之后，为了判断是否在跌倒之后人体有长时间的静止不动。需要继续检测Activity中断和Inactivity中断。这里把THRESH_ACT设为0.5g，Activity中断为ACcoupled工作模式。把THRESH_INACT设为0.1875g，把TIME_INACT设为10s，Inactivity中断为AC coupled工作模式。也就是说，如果在10s之内，人体一直没有任何动作，则会产生Inactivity中断，使系统给出一个严重报警。而在此期间一旦人体有所动作，则会产生Activity中断，从而结束整个判断过程。
8. 本算法还可以检测出人体从较高的地方跌落。如果Free_Fall中断连续产生且之间的间隔小于100ms，可以认为，人体处于连续的跌落状态。如果Free_Fall中断(失重)连续发生300ms，则说明人体是从超过0.45m的高度跌落，系统会给出一个跌落的报警。

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

本算法已在ADuC7026微控制器中以C语言实现。我们将同时提供一个基于该解决方案的测试案例，以便验证算法。包括向前跌倒、向后跌倒、向左侧跌倒、向右侧跌倒在内的四种体位，每种体位均测试20次。前10次试验为典型跌倒，在跌倒后无长时间静止，后10次试验也为典型跌倒，只是跌倒后有长时间静止。表2列出了测试结果。

本试验表明，本文提出的基于ADXL345的解决方案可以有效地检测到跌倒状态。请注意，这只是一个简单的实验，要验证本解决方案的可靠性，还需要进行更加全面、有效和长期的实验。

表2. 测试结果

	测试条件		测试结果	
实验编号	跌倒体位	跌倒后有无长时间静止	检测到的跌倒(次数)	检测到的长时间静止(次数)
1 至 10	向前跌倒	无	10	0
11 至 20	向前跌倒	有	10	10
21 至 30	向前跌倒	无	10	0
31 至 40	向前跌倒	有	10	10
41 至 50	向左侧跌倒	无	10	0
51 至 60	向左侧跌倒	有	10	10
61 至 70	向左侧跌倒	无	10	0
71 至 80	向左侧跌倒	有	10	10

代码实例

本节将给出以上解决方案基于ADXL345和ADuC7026平台的C语言代码。项目中含有四个.h文件和一个.c文件，采用Keil UV3进行编译。文件“FallDetection.c”含跌倒检测算法。“FallDetection.h”详细列出跌倒检测算法中用到的函数及变量定义、ADXL345读/写函数的应用以及ADXL345初始化。“ADuC7026Driver.h”包括ADuC7026 GPIO控制函数、I^2C主机读写函数以及ADuC7026初始化。“xl345.h”含ADXL345寄存器及位定义。“xl345_io.h”文件含用于通过I^2C及SPI读写ADXL345的函数。

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
```

结论

ADXL345是ADI出品的一种功能强大而全面的加速度计。本应用笔记基于多种内置的运动状态检测功能及灵活的中断，提出了一种新的跌倒检测解决方案。本解决方案通过ADXL345硬件中断实现，具有算法复杂度低、检测准确率高的特点。

参考文献

ADXL345 Data Sheet. Analog Devices, Inc. 2009

注释

注释

AN08185-0-6/09(0)