应用报告
ZHCA010 – 2004 年 12 月

MSC12xx 的平均值计算

Russell Anderson
数据采集产品

摘要

MSC12xx 系列微控制器产品包含一个 32 位累加器，简化了 24 位模数 (A/D) 转换器的数据平均运算。 本应用手册阐述了数据平均法的优势，并讨论了使用累加器功能计算移动平均值的方法。 还将比较本流程与标准移动平均值（数据块平均）计算，后者首先需要保存所有采样，然后才能计算平均结果。

内容

附图目录

附表目录

1 平均值计算

平均值 (Ave_n) 定义为 *n* 个采样的总和除以 *n*。

$$Ave_n = \frac{\sum_{i=1}^{n} Sample_i}{n} \quad (1)$$

要计算任何一组 *n* 个采样的平均值，必须提供所有的采样数据。 如果 *n* 值很大，则需要很大的存储空间。 但是，可以将此条件限制为仅存储最后 *n* 个采样的数据。 在此种情况下，对于 *k*个数据的平均值，我们可以使用以下公式：

$$Ave_k = \frac{\sum_{i=1+k-n}^{k} Sample_i}{n} \quad (2)$$

但是，在存储和计算要求方面，此种需求的效率仍然很低，因为我们需要完成以下任务：

- 存储 *n* 个数据值
- 执行 *n* 次加法和一次除法

这被称为移动平均法，因为每次 *k*个数据的平均值都是基于最新的 *n* 个值计算出来的。 换句话说，它就象一个每次显示 *n* 个值的移动窗口，用于计算数据序列的平均值（请参阅图 1）。

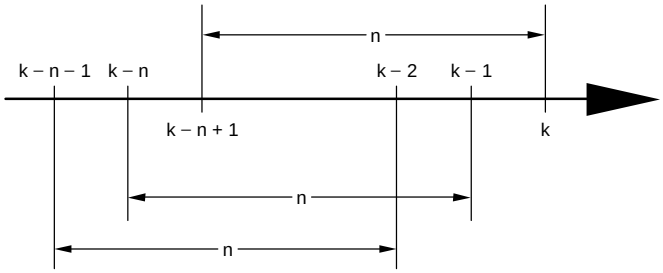

图 1. *n* 个数据点的移动平均法

这会为嵌入式系统带来严重的缺陷。 内存有限，而且除法是一个在 8 位 系统上进行 32 位值计算的复杂处理。 但是有一些方法可以简化此处理过程，同时又不会丧失平均法的优势。

如果可以按递归方式执行计算，则可以改进平均算法处理。 也可以描述为：将前 *n – 1* 个采样的总和加上一个新的采样，然后除以 *n*。

$$Ave_n = \frac{\sum_{i=1}^{n-1} Sample_i + Sample_n}{n} \quad (3)$$

通过，我们可以推导 Ave_{n-1} 的公式。

$$Ave_{n-1} = \frac{\sum_{i=1}^{n-1} Sample_i}{n-1} \quad (4)$$

重新调整后，如 所示 所示：

$$(n-1) \cdot Ave_{n-1} = \sum_{i=1}^{n-1} Sample_i \quad (5)$$

通过 我们可以用 (n 1) x Ave_{n-1} 替换 $\sum_{i=1}^{n} \text{Sample}_i$ 中 的值，从而得出 – 所示：

$$\text{Ave}_n = \frac{(n-1)\cdot \text{Ave}_{n-1} + \text{Sample}_n}{n} = \frac{n\cdot \text{Ave}_{n-1}}{n} - \frac{\text{Ave}_{n-1}}{n} + \frac{\text{Sample}_n}{n} \quad (6)$$

可以简化为 所示：

$$\text{Ave}_n = \text{Ave}_{n-1} + \frac{\text{Sample}_n - \text{Ave}_{n-1}}{n} \quad (7)$$

在与前一平均值合并时，每个新平均值仅需要进行一次减法，一次除法和一次加法处理。 此种方法提供了一种可以持续更新平均值的简便公式，方法就是从当前采样中减去前一平均值，除以 *n*，然后将结果与旧平均值相加。

2 通过除以 2^m 简化移动平均法

除法运算需要在微控制器上耗费大量时间并执行多次循环。 由此看出，对于整数来说，使用 2^m 的幂代替 *n* 时，除法运算只是简单的算术右移 *m* 次。因此可以修改，使用 2^m 代替 *n* 以简化除法。

$$\text{Ave}_n = \text{Ave}_{n-1} + \frac{\text{Sample}_n - \text{Ave}_{n-1}}{2^m} \quad (8)$$

此时就有关于用 2^m 替代 *n* 后，性能将会受到多少损失的疑问。 答案取决于如何选择 *m* 值。 可以选择一个 *m* 值，使其提供的性能与 *n* 值的基本相同。 表 1 显示了两种选择 2^m 中 *m* 的方法。 *简单*列选择不会导致 2^m 超过 *n* 值的最大的 *m* 值。 *最优*列按 所示舍入 *m* 的值 所示：

$$\text{Best fit } m = \text{round}\left(\log_2(n)\right) \quad (9)$$

表 1. 选择 *m* 的值

n	简单	最优
1	1	1
2	2	2
3	2	4
4	4	4
5	4	4
6	4	8
7	4	8
8	8	8
9	8	8
10	8	8
11	8	8
12	8	16
13	8	16
14	8	16
15	8	16
16	16	16
17	16	16

在以下图表中对这两种方法进行了比较。

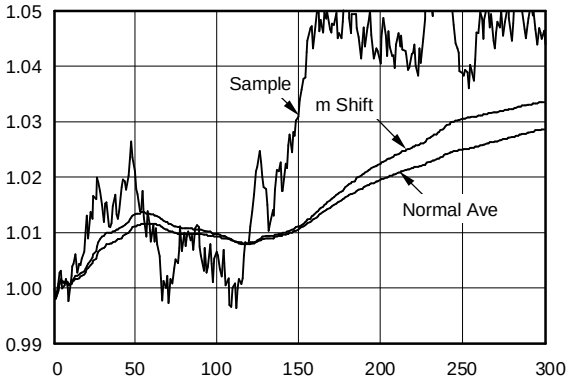

图 2. *m* 的简单选择

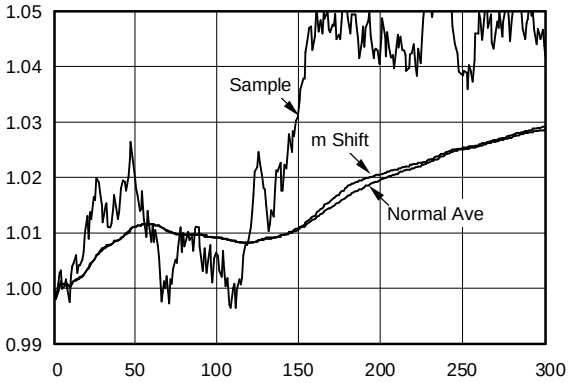

图 3. *m* 的最优选择

图 2 与图 3 显示了使用四舍五入最优法选择的 *m* 得出的结果与直接使用 *n* 值的结果几乎相同。

3 数据块平均响应

此部分讨论平均滤波器，首先讨论一下要求对最后 *n* 个值求和的平均值。 此种类型的滤波器有优点也有缺点。 我们在检查滤波器的步阶响应时，这点变得愈加明显。 我们可以称这类滤波器为***数据块平均滤波器***，因为其获取的是包含 *n* 个值的数据块的平均值。 如同所期望的那样，被平均的数据出现大步阶更改时，数据块平均值不会接近新值，直到该数据块中的所有值都接近当前值为止。 在这种情况下，会在步阶响应中产生一个斜坡，直到旧值不再使用，如图 4 所示。

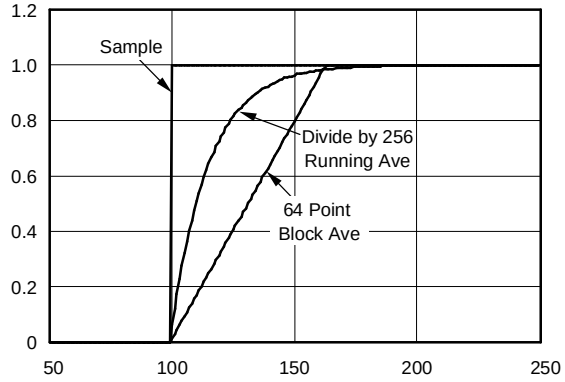

图 4. 数据块平均值和固定除数 256 的稳定时间

图 4 显示了 256 点固定除数法提供的性能与 64 点数据块平均法相似。 但是，如果在数据块平均法稳定后，检查该图的分布（请参阅图 5），可以看出 256 点除法稳定至相同最终值需要花费的时间比 64 点数据块平均法的要长。

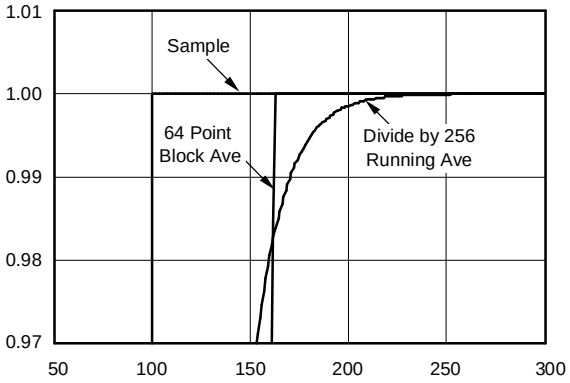

图 5. 平均滤波器的最终稳定时间

数据块平均滤波器的另外一个问题是要使用所有的点，包括有效暂态。 但是，可以使用一种算法使滤波器忽略值上的显著移位。 可以对“固定除数”滤波器使用一种相似的算法，以在发生显著移位时生成一个自动响应并更新该平均值。 图 6 显示了数据中的短时脉冲波形干扰实例。

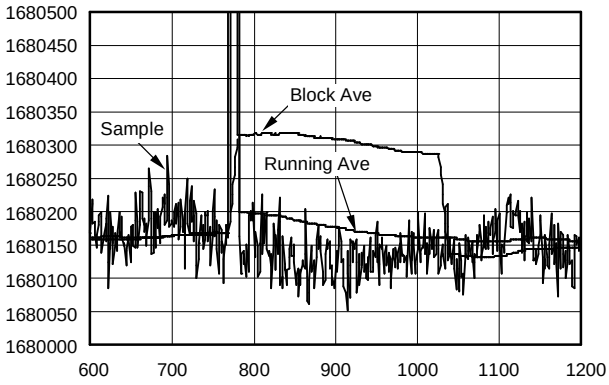

图 6. 数据块平均值和固定除数的比较（使用自动响应配置）

4 使用固定移位量

始终使用相同的除数（或者换句话说，相同的移位量）的简化方法具有一些显著的优势。 例如，移位值为 8，32 位整数的移位就是把 32 位结果的 3 个高位字节用于 3 个低位字节，并根据原始值的符号位设置最高有效字节。

可以看到除以 256 的滤波器看似与 64 点数据块平均法的稳定时间相同，实际上前者花费的时间较长。 如果能够解决此问题，固定除数滤波器就能提供一个简单且强大可靠的滤波器解决方案。

由于稳定时间问题与输入数据中的大步阶相关，因此其中一种解决方案就是仅对步阶更改的幅度作出比较。 如果出现大步阶更改，就使用最新的采样替换该平均值。 这样不但可以使滤波器快速稳定，同时又保留了排除稳定问题的较大 *n* 值的滤波性能。

5 MSC12xx 的平均方法

MSC12xx 产品系列包括 32 位累加器，它可以用于直接平均来自 24 位 A/D 转换器的输入数据。 如果 A/D 转换器信号填满 24 位，则在 32 位累加器溢出之前，最多的平均值数将是 256。 MSC12xx 产品中的此累加器可以设置为自动平均 2、4、8、16、32、64、128 或 256 次，移位以除以相同值，然后生成一个中断，表明进程已结束。 如果全部输入范围减少 4 倍，则仅有低位的 22 个位作为结果，重复“除以 256”四次以达到 1024 的平均值。 然后，32 位累加器仅在求和寄存器的每四次中断后被清零。

另外一种实现较大平均值的方法是使用移动平均法。 对于移动平均法，仅将采样与前一平均值的差值（用 *n* 除）添加至前一平均值中。 然而，MSC12xx 求和寄存器具有内置指令，最高可进行 256 次除法，较大的数字应以迭代的方式实施。 如前所述，移动平均法的缺陷之一是要花费较长时间才能完全稳定。 以下 MSC1210 程序用于监视新采样和当前平均值之间的差值大小。 器件将根据这一信息决定是否使用当前采样重新加载平均值。 这就极大地加快了步阶输入的稳定时间，同时又保持了一个低噪音、持续平均的输出。

6 对移动平均法使用 32 位累加器

请记住移动平均法的公式是：使用新采样减去当前平均值；再将所得的值除以 *n*；然后将结果加至当前平均值，从而创建新平均值，如 Equation 7 所示：

$$Ave_n = Ave_{n-1} + \frac{Sample_n - Ave_{n-1}}{n}$$

MSC1210 中的 32 位求和寄存器没有减法运算。 为了执行减法运算，将当前平均值的负值放入求和寄存器，从而与新采样相加。 尽管 MSC1211 没有减法函数，但这种对平均值进行二进制补码运算的方法比减法运算所需的求和寄存器的设置和清零速度更快。 要执行移动平均法运算，需要进行以下步骤：

1. 保存当前平均值。
2. 清零求和寄存器
3. 对求和寄存器加载当前平均值的 1 的补码（即，反码）。
4. 添加 1 至求和寄存器（以创建平均值的 2 的补码形式）。
5. 添加 A/D 转换器中的新采样以计算（样片 – 平均）。
6. 算术右移 *m* 以用 2^m 除。
7. 添加步骤 1 中的当前平均值。
8. 求和寄存器现在包含新的当前平均值。

输入的大步阶更改可通过更改步骤 6 进行处理（除法）。 不是除以 2^m，而是将（样本 – 平均）的值保存在累加器中，当在步骤 7 添加平均值时，新的平均值将等于最后一个采样。 此种方法提供了一种将平均值重置为当前采样的途径，同时也为步阶响应提供了一个快速恢复时间。

由于移位除法是整数运算，在平均计算过程中会损失一些精度。 补偿方法为保留一个整数及平均值和采样间的差值的总和。 如果差值的总和超过了 2^m，用其减去 2^m，并从（样本 – 平均）的结果中加/减 1。

步阶更改的快速响应和差值跟踪都在除法子例程中完成，该子例程接收 *m* 的值，而将差值存储在 32 位累加器中。 图 7 和图 8 显示了此方法与数据块平均方法的比较

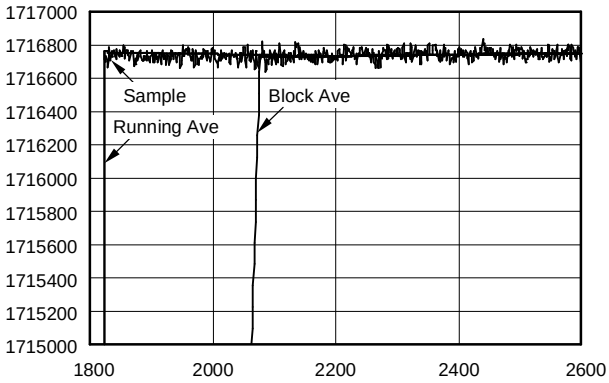

图 7. 固定除数和数据块平均法的步阶响应

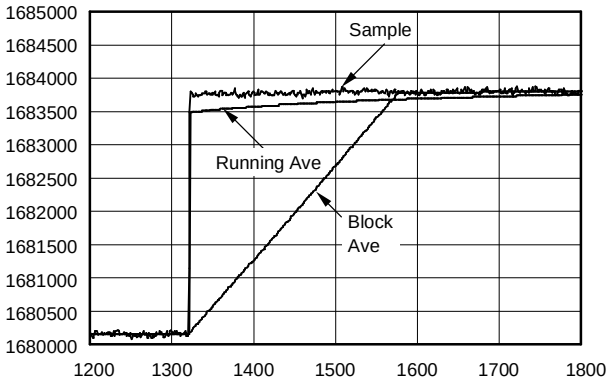

图 8. 不完全的步阶恢复

固定除法的步阶恢复可以通过简单地更改程序以使用剩余部分进行改进，而不是使用上一差值触发除法例程中的移动平均法的替换，如图 9 与图 10 。

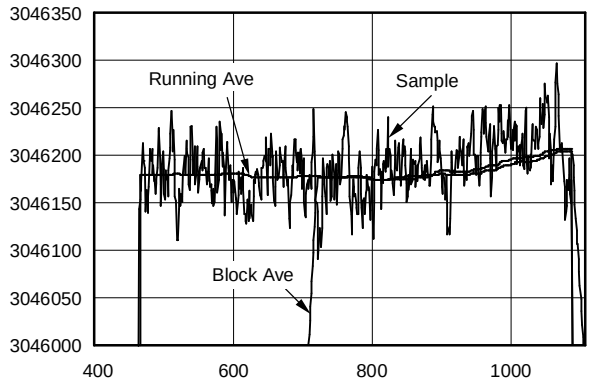

图 9. 正步阶响应

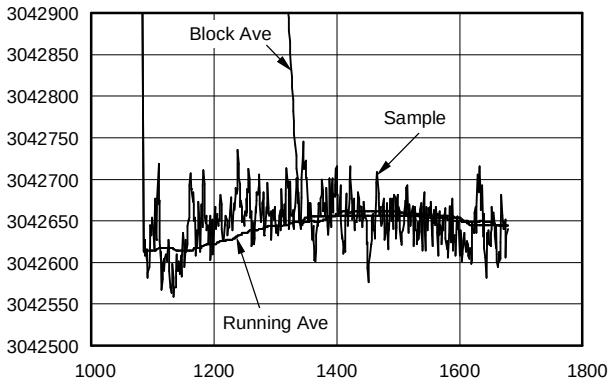

图 10. 负步阶响应

7 程序说明

```
while(1) {       if (done) {                  // ADC interrupt adds result to Summation Register          done =
FALSE;                                      // Running Ave(n) = RA(n-1) + (Sample(n) - RA(n-1))/n
                // 32-bit accumulator contains (Sample - Ave(n-1))          divide(m);                 // Divide by n
~= 2^m (shift by m)          load32acc(runAve);         // Add Previous Average          runAve = read32acc();
// Save New Average          SSCON = 0;                 // Zero accumulator          load32acc(comp(runAve));   //
1's complement of new average          load32acc(1);              // 2's complement "-ave", ready to add sample
   SSCON = 0x40;             // Enable ADC to add with next sample          printf("sample=,%ld, RA=,%ld,
Block=,%ld\n", adresult, runAve,blockave);       }    }
```

主程序循环从加载 (– runAve) 的 32 位累加器（求和寄存器）开始。 初始 (runAve) 的平均值为最新采样。 在 A/D 转换器的结果就绪时，它们自动地被加至 32 位累加器中，这表示累加器当前值为 (sample – runAve)。 此数仅需除以 *n*，并加至 runAve 以创建新版 runAve。

除法例程是此算法的核心部分。 除了将累加器移位 *m* （$n = 2^m$），它还会检查 32 位累加器以查看差值是否足够大，是否需要替换当前平均值。 方法为只需简单地跳过除法并返回 (sample – runAve) 的值。 runAve 的副本被添加到该值中，新的平均值将等于新采样。 在差值很大时，除法例程还可以决定减小除数，从而缩短稳定时间。

保存了新平均值后，32 位累加器需要加载该平均值的负值，以将其与下一采样结果相加。 此进程要求两个步骤，以获得平均值的 2 的补码。 首先，累加器清零，加载平均值的 1 的补码（反位）。 然后，1 被添加至寄存器中以提供平均值的 2 的补码，或者换句话说，平均值的负值。 此时，我们就可以准备下一采样了。

附录 A　　完整程序

```
//*********************************************************************
// File name: runave.c
//
// Copyright 2004 Texas Instruments Inc as an  unpublished work.
//
// Revision History . 。 。 。
//       Version 1.0
//
// Compiler Version: Raisonance Compiler 3.03.34
//
// Module Description:
//    This is a running average (RA) implemented on the MSC1210. The basic
// formula is:  RA(n) = RA(n-1) + (Sample(n) - RA(n-1))/n
// the value for n is converted to 2^m so that the divide is an arithmetic right
// shift by m. The Samples come from the 24-bit ADC.
//
//*********************************************************************
 // Include section #include "legal.c"      //Texas Instruments, Inc. copyright and liability
#include <REG1210.H>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
//  Defines section
#define FALSE 0
#define TRUE !FALSE
//LSB = 2.50/pow(2,24); (1.49e-7) unipolar
//LSB = 5.0/pow(2,24); (2.98e-7) bipolar
#define LSB 2.98e-7
#define comp(x) (0xFFFFFFFF ^ x) // 1's complement operation on long integer
//  Function Prototype Section
extern void autobaud(void);
extern unsigned long unipolar(void);
extern signed long bipolar(void);
extern void load32acc(long int);
extern long read32acc(void);
long int initAve(void);
void Convert(void) interrupt 6;
void divide(char);
// Global Variables for interrupt routine
char done=0;                 // ADC interrupt flag
long int adresult;         // ADC result
```

```
//*********************************************************************
// Function name  : long int initAve(void)
//    returns     : Initial Average Value
// Created by     : Russell Anderson
// Date created   : 11-11-04
// Description
//       * Setup for running average. Initial Ave = sample
//*********************************************************************
long int initAve(void){
   long int ave;
   SSCON = 0;              // Clear 32-bit accumulator, Set to add values
   while (!done){}         // Wait for ADC result
   done = FALSE;
   ave = bipolar();        // Save Current Sample as Average Value in runAve
   load32acc(comp(ave));   // 1's complement of initial average
   load32acc(1);           // 2's complement operation, ready to add sample
   SSCON = 0x40;           // Setup to add ADC result
   return(ave);
 }
//*********************************************************************
// Function name  : void Convert(void) interrupt 6 (Auxillary Interrupt)
//    returns     : return ADC value in adresult
// Created by     : Russell Anderson
// Date created   : 11-11-04
// Description       : Since the ADC is the only interrupt enabled, this
//                   interrupt routine reads the value of the 24-bit ADC
//
//*********************************************************************
void Convert(void) interrupt 6
{
    adresult = bipolar();
    done = TRUE;
    AI=0;                       // Clear Auxiliary Interrupt
 }
```

```
//*********************************************************************
// Function name  : void divide(unsigned char m)
//
//    arg m       : The value to divide (2^m) into 32-bit accumulator
// Created by     : Russell Anderson
// Date created   : 11-11-04
// Description    : This routine divides the Summation register
//                      by 2^m and also compensates for the lack of precision
//                   in the integer division by keeping a residual value.
//                   This routine also skips or adjusts the division for large
//                   differences between the current sample and the average.
// Notes          : maximum m allowed is 8
//*********************************************************************
void divide(unsigned char m)
{
   int n;
   char sign;
   long diff, magnitude;
   static long residue=0;
   #define sigma  44                // Standard Deviation of data to be measured
   diff = read32acc();              // Sample(n) - Ave(n-1)
   residue += diff;                 // Accumulate difference because divide by n can = 0
   magnitude = abs(residue);        // Use the absolute value for step change detection
   sign = (residue>0) ? 1 : 0;      // Determine sign for correction when residue exceeds n
   if (magnitude>sigma*16){         // No divide, return(sample - ave)
      residue=0;
      return;
 }
  if (magnitude>sigma*8){          // Modify filter to reduce settling time
      m--;
 }
 n = 1<<m;                        // n = 2^m
 m--;                             // Summation Register divides by m-1
 SSCON = m | 0x80;                // Using 32-bit Accumulator Divide by 2^m
 if (read32acc() == -1) load32acc(1);    // Add 1 (remove negative bias after divide)
 if (abs(residue) > n){           // Compensate for large residual
    if(sign){
       residue -= n;
       load32acc(1);              // Add +1
   }
 else{
       residue += n;
       load32acc(-1);             // Add -1
    }
  }
}
```

```
void main(void)
 {
     unsigned char i;
     long int runAve;
     autobaud();
     printf("MSC1210 ADC Conversion Test\n");
     //Timer Setup
     USEC= 10;                          // 11.0592 MHz Clock
     /* Setup ADC */
     PDCON &= 0xF7;                     // Enable adc
     ACLK = 8;                          // For 11.0592 MHz, gives modclock of 1.2288 MHz
                                        // Modulation Clock = 19,200 Hz
     ADMUX = 0x58;                      // AIN5-AINCOM
     ADCON0 = 0x30;                     // Vref on 2.50V, Buff on, BOD off
     DECIMATION = 1920;                 // 10 Hz data rate
     AIE = 0x20;                        // Enable A/D interrupt
     EAI=1;                             // Enable Auxillary Interrupts
     ADCON1 = 0X01;                     // bipolar, auto, self calibration, offset, gain
     for (i=0; i<3; i++){               // Wait for Sinc3 filter to settle after calibration
        while (!done) {}
        done = FALSE;
  }
#define m 8                             // n = 256 (m^8)
    runAve = initAve();
    while(1) {
       if (done) {                      // ADC result has been added to Summation Register
           done = FALSE;
                                        // Running Ave(n) = RA(n-1) + (Sample(n) - RA(n-1))/n
                                        // 32-bit accumulator contains (Sample - Ave(n-1))
           divide(m);                   // Divide by n ~= 2^m (shift by m)
           load32acc(runAve);           // Add Previous Average
           runAve = read32acc();        // Save New Average
           SSCON = 0;                   // Zero accumulator
           load32acc(comp(runAve));     // 1's complement of new average
           load32acc(1);                // 2's complement "-ave", ready to add sample
           SSCON = 0x40;                // Enable ADC to add with next sample
           printf("sample=,%ld, RA=,%ld,  Block=,%ld\n", adresult, runAve,blockave);
   }
  }
 }
```

重要声明

德州仪器 (TI) 及其下属子公司有权在不事先通知的情况下，随时对所提供的产品和服务进行更正、修改、增强、改进或其它更改，并有权随时中止提供任何产品和服务。 客户在下订单前应获取最新的相关信息，并验证这些信息是否完整且是最新的。 所有产品的销售都遵循在订单确认时所提供的 TI 销售条款与条件。

TI 保证其所销售的硬件产品的性能符合 TI 标准保修的适用规范。 仅在 TI 保修的范围内，且 TI 认为有必要时才会使用测试或其它质量控制技术。 除非政府做出了硬性规定，否则没有必要对每种产品的所有参数进行测试。

TI 对应用帮助或客户产品设计不承担任何义务。 客户应对其使用 TI 组件的产品和应用自行负责。 为尽量减小与客户产品和应用相关的风险，客户应提供充分的设计与操作安全措施。

TI 不对任何 TI 专利权、版权、屏蔽作品权或其它与使用了 TI 产品或服务的组合设备、机器、流程相关的 TI 知识产权中授予的直接或隐含权限作出任何保证或解释。 TI 所发布的与第三方产品或服务有关的信息，不能构成从 TI 获得使用这些产品或服务的许可、授权、或认可。 使用此类信息可能需要获得第三方的专利权或其它知识产权方面的许可，或是 TI 的专利权或其它知识产权方面的许可。

对于 TI 的数据手册或数据表，仅在没有对内容进行任何篡改且带有相关授权、条件、限制和声明的情况下才允许进行复制。 在复制信息的过程中对内容的篡改属于非法的、欺诈性商业行为。 TI 对此类篡改过的文件不承担任何责任。

在转售 TI 产品或服务时，如果存在对产品或服务参数的虚假陈述，则会失去相关 TI 产品或服务的明示或暗示授权，且这是非法的、欺诈性商业行为。 TI 对此类虚假陈述不承担任何责任。

可访问以下 URL 地址以获取有关其它 TI 产品和应用解决方案的信息：

产品	
放大器	http://www.ti.com.cn/amplifiers
数据转换器	http://www.ti.com.cn/dataconverters
DSP	http://www.ti.com.cn/dsp
接口	http://www.ti.com.cn/interface
逻辑	http://www.ti.com.cn/logic
电源管理	http://www.ti.com.cn/power
微控制器	http://www.ti.com.cn/microcontrollers
应用	
音频	http://www.ti.com.cn/audio
汽车	http://www.ti.com.cn/automotive
宽带	http://www.ti.com.cn/broadband
数字控制	http://www.ti.com.cn/control
光纤网络	http://www.ti.com.cn/opticalnetwork
安全	http://www.ti.com.cn/security
电话	http://www.ti.com.cn/telecom
视频与成像	http://www.ti.com.cn/video
无线	http://www.ti.com.cn/wireless

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265