Application Note

SPC11X8/SPD11X8 ADC 码值校准使用指南

2019 年 7 月 – 版本 1

目录

表格列表

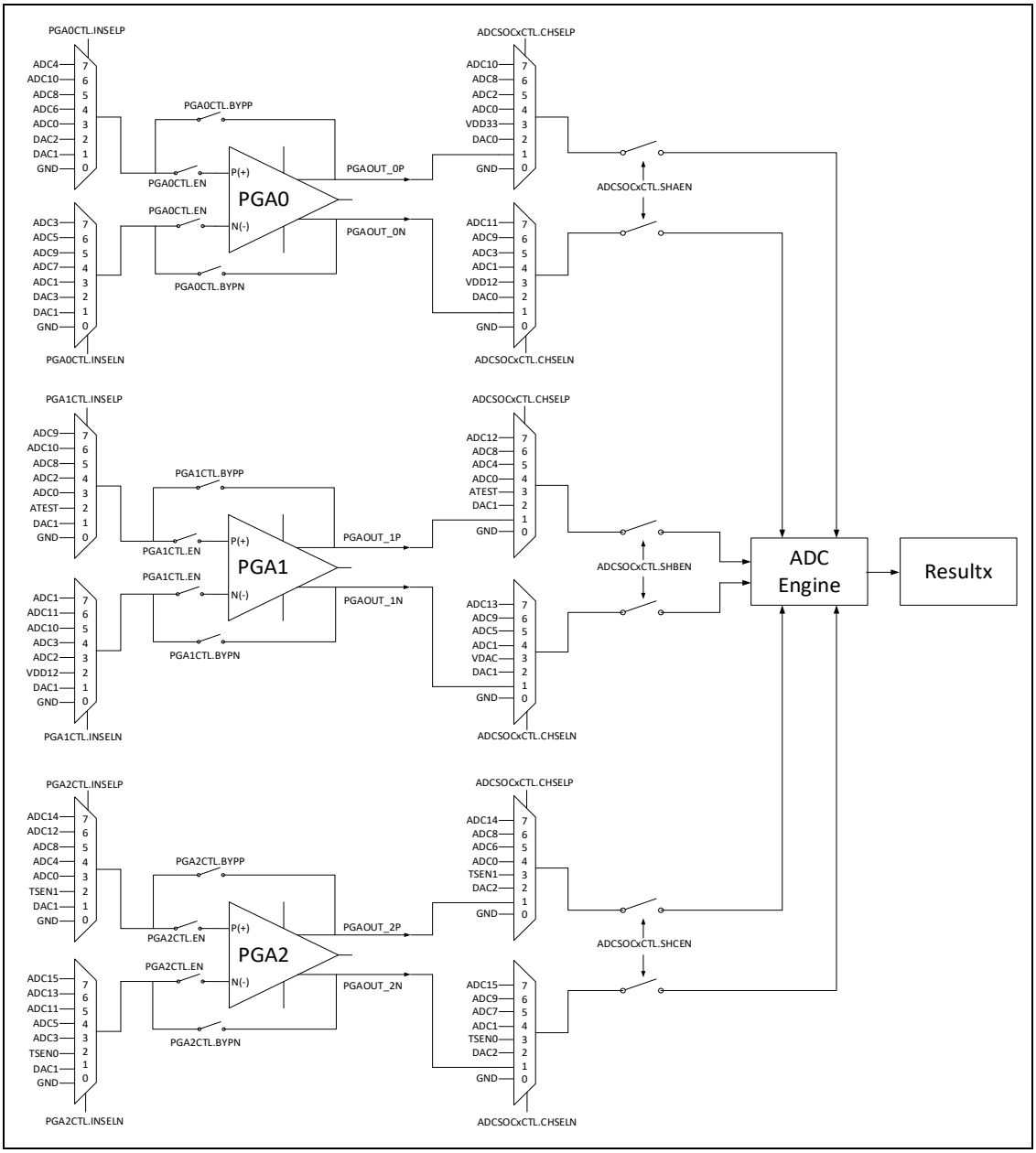
图片列表

1 ADC 概述

SPC11X8/SPD11X8 集成了 3 组增益可调放大器（PGA）和支持三路同步采样的 14 位模式转换器（ADC）。

实际使用时，输入信号先经由 PGA 放大，然后送入 ADC 转化成数字码值。下图集成了 SPC11X8/SPD11X8 中三个重要的仿真器件之讯号流程图（Signal Flow）。PGA 和 ADC 均存在增益误差（Gain Error）和失调误差（Offset Error）。这些误差为线性误差，可以通过校准消除。

图 1-1: ADC 及 PGA 连接框图



实际应用时，随着温度变化，输入信号，PGA 和 ADC 的增益误差和失调误差均会变化，最终体现为 ADC 的输出码值温漂。在这里，我们仅讨论 PGA 和 ADC 的增益误差和失调误差温漂对输出结果的影响。输入信号的温漂与具体应用环境有变化，可以通过本文的分析方法做类似处理。

在阅读完本章后，您将可以了解到以下内容：

- 信号经由 PGA 放大，然后送入 ADC 转化为数字码值。
- PGA 和 ADC 存在失调误差和增益误差，这两个误差存在温漂，因此体现为 ADC 的输出结果会存在温漂。

2 PGA 增益误差和失调

2.1 PGA 单元概述

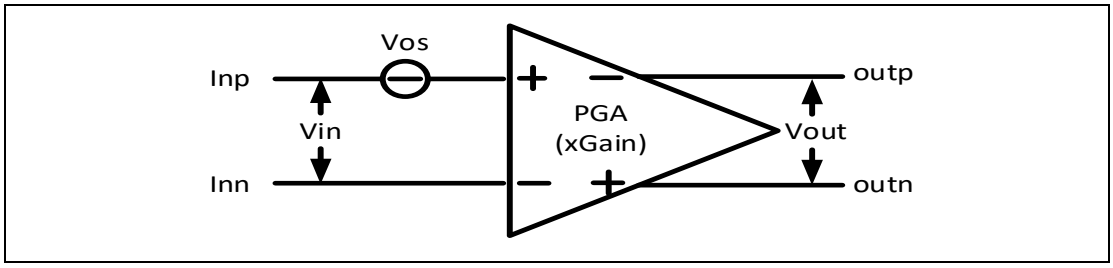
SPC11X8/SPD11X8 集成了 3 组可调增益的差分运放(Truly Differential PGA)，可抑制共模噪声，亦可仿真为单端 PGA 使用。PGA 的基本特点如下：

- 差分模式时放大增益：2X 4X 8X 16X 24X 32X 48X 64X
- 单端模式时放大增益：1X 2X 4X 8X 12X 16X 24X 32X(差分仿真为单端模式时，则与差分模式同)
- 弹性的信道选择
- 输出直接链接 ADC 采样通道
- 输入与输出均可输入比较器 COMP
- Slew Rate > 35V/us
- Band Width > 1MHz
- 建立时间< 880ns (0.3V ~ VDDA-0.3V)
- PGA 的增益温漂±57ppm/oC
- PGA 的失调温漂±0.024mV/oC（在本章节中，将介绍）
- 增益误差和失调误差对 PGA 输出结果的影响
- 当温度变化时，PGA 输出变化

2.2 PGA 的增益误差和失调

下图为 SPC11X8/SPD11X8 仿真电路构架，PGA 为下图套色部分。

图 2-1: PGA Offset and Gain



假设 PGA 在差分模式，增益为 GPGA，增益误差为ΔGPGA,失调为 OSPGA，输入信号为 Vin，输出电压为 Vout。所以存在误差的情况下，PGA 的输出 Vout 为：

$$V_{out} = GPGA * (1 + \Delta GPGA) * (V_{in} + SOPGA) \quad (2\text{-}1)$$

假设差分增益为 4，增益误差1%，失调为1mV。输入差分电压V_{in}为0.5V，则输出电压V_{out}为2.02404V。这个结果相对于理想值2V存在1.2%的偏差。

2.3 PGA 输出电压温漂

实际环境中，PGA 的增益和失调都会随着温度变化。考虑温度变化的情况下，假设 PGA 的增益温漂为 TGPGA 和失调温漂 TOSPGA，为公式（2-1）将变为：

$$V_{out} = GPGA * (1 + \Delta GPGA) * (1 + TGPGA) * (V_{in} + OSPGA + TOSPGA) \quad (2\text{-}2)$$

假设增益温漂为57ppm/℃，失调温漂为0.024mV/℃。输入差分电压V_{in}为0.5V，如果温度变化50℃，则输出电压V_{out}为2.03476V。这个结果相对于常温结果2.02404V存在0.53%的偏差。

3 ADC 的增益误差和失调

3.1 ADC 单元概述

SPC11X8/SPD11X8 集成了 14-bit，4Msps差分输入模数转换器（ADC）。该 ADC 也可以单端输入使用，此时为 13-bit。ADC 的基本特点如下：

- 差分和单端模式的精度分别为 14-bit/13-bit
- 单次转换时间为140ns
- 16 个独立的外部模拟输入信号通道（不包含 PGA 输入）
- ADC 的增益温漂±50ppm/℃
- ADC 的失调温漂±0.03mV/℃（在本章节中，将介绍）
- 增益误差和失调误差对 ADC 输出结果的影响
- 当温度变化时，ADC 输出变化。

3.2 ADC 驱动函数

表 3-1: ADC 相关宏定义

宏名	功能及说明
ADC_EnableInt(eSOC)	使能 ADC 对应 SOC 通道中断
ADC_EnableIntx()	使能 ADC 对应 SOC 通道中断（x=0~15，代表 ADC 通道号）
ADC_DisableInt(eSOC)	关闭 ADC 对应 SOC 通道使能
ADC_DisableIntx()	关闭 ADC 对应 SOC 通道使能（x=0~15，代表 ADC 通道号）
ADC_ClearInt(eSOC)	清除 ADC 对应 SOC 通道中断
ADC_ClearIntx()	清除 ADC 对应 SOC 通道中断（x=0~15，代表 ADC 通道号）
ADC_GetIntFlag(eSOC)	获取 ADC 对应 SOC 通道中断状态
ADC_GetIntxFlag()	获取 ADC 对应 SOC 通道中断状态（x=0~15，代表 ADC 通道号）
ADC_ClearOverflowInt(eSOC)	清除 ADC 对应 SOC 通道溢出中断
ADC_GetOverflowIntFlag(eSOC)	获取 ADC 对应 SOC 通道溢出中断状态
ADC_EnableIntTriggerSOC(eSOC)	使能触发对应 SOC 通道采样的触发源改由 ADCINTSOCSELx 寄存器控制（x=0，1）
ADC_DisableIntTriggerSOC(eSOC)	触发对应 SOC 通道采样的触发源由 ADCSOCCTLx 寄存器控制（x=0~15）
ADC_SetSOCPriority(ePriority)	设置 SOC 通道的优先级

ADC_GetSOCPriority()	获取对应 SOC 通道的优先级
ADC_Enable()	使能 ADC 模组
ADC_Disable()	关闭 ADC 模组
ADC_Reset()	复位 ADC 模组
ADC_SetAverageCnt(eSOC, eAvgCnt)	设置对应 SOC 通道采样次数 eSOC：对应 SOC 通道号 eAvgCnt：采样次数
ADC_SetSOCSH(eSOC, eSH)	设置对应 SOC 通道使用的采样器 eSOC：对应 SOC 通道号 eSH：采样器的编号（SPC11X8/SPD11X8 中有三个采样器 A/B/C）
ADC_SelectTrigger(eSOC, eTriggerSource)	选择触发对应 SOC 采样的采样源 eSOC：对应 SOC 通道号 eTriggerSource：采样源
ADC_SoftwareTrigger(eSOC)	使用软件触发对应 SOC 通道采样
ADC_GetResult(eSOC)	获取 ADC 采样的结果（此接口获取到的数值有可能是负数）
ADC_GetTrimResult1(eSOC)	获取 ADC 采样的结果（此接口获取到的数值有一定为正数）
ADC_GetPPUResult(ePPU)	获取 PPU 子模组的结果
ADC_SetSOCDelayCapture(ePPU, eSOC)	选择进行延迟采样的 SOC 通道 ePPU：PPU 子模组基址 eSOC：对应 SOC 通道
ADC_GetSOCDelay(ePPU)	获取对应 SOC 通道延迟采样的延迟时间
ADC_EnablePPU(ePPU)	使能 PPU 子模组
ADC_DisablePPU(ePPU)	关闭 PPU 子模组
ADC_EnablePPUInt(ePPU, ePPUEvt)	使能 PPU 子模组中对应功能的中断 ePPU：PPU 子模组基址 ePPUEvt：PPU 子模组功能事件
ADC_DisablePPUInt(ePPU, ePPUEvt)	关闭 PPU 子模组中对应功能的中断 ePPU：PPU 子模组基址 ePPUEvt：PPU 子模组功能事件
ADC_EnablePPUTripEvent(ePPU, ePPUEvt)	选择 PPU 子模组对应功能作为 PWM trip-zone 功能的触发事件 ePPU：PPU 子模组基址 ePPUEvt：PPU 子模组功能事件
ADC_DisablePPUTripEvent(	关闭 PPU 子模组对应功能作为 PWM trip-zone 功能的触发事件

ePPU, ePPUEvt)	ePPU：PPU 子模组基址 ePPUEvt：PPU 子模组功能事件
ADC_ClearPPUInt(ePPU,ePPUEvt)	清除 PPU 子模组对应事件的中断
ADC_ClearPPUGlobalInt(ePPU)	清除 PPU 子模组的全局中断
ADC_GetPPUIntFlag(ePPU,ePPUEvt)	获取 PPU 子模组对应事件的中断标记
ADC_GetPPUGlobalIntFlag(ePPU)	获取 PPU 子模组对应事件的全局标记
ADC_SetPPURef(ePPU, i32Val)	设置 PPU 子模组的参考数值 ePPU：PPU 子模组基址 i32Val：参考数值
ADC_SetPPUTooHighThreshold(ePPU, i32Val)	设置 PPU TooHigh 的检测阈值 ePPU：PPU 子模组基址 i32Val：TooHigh 的阈值
ADC_SetPPUTooLowThreshold(ePPU, i32Val)	设置 PPU TooLow 的检测阈值 ePPU：PPU 子模组基址 i32Val：TooLow 的阈值
ADC_WALLOW()	允许操作 ADC 寄存器
ADC_WDIS()	禁止操作 ADC 寄存器

表 3-2: ADC 驱动函数

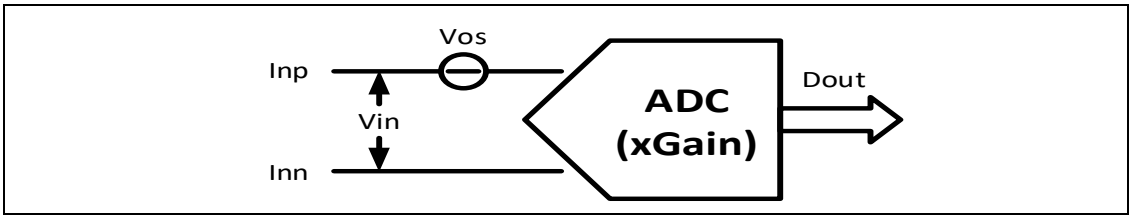
函数名	功能及说明
void ADC_SelectIntTriggerSOC(ADC_SocEnum eSOC, ADC_IntEnum eInt)	当 ADC_EnableIntTriggerSOC(eSOC)被调用后，采用此函数设置触发对应 SOC 通道采样的触发源 eSOC：ADC 模组的 SOC 通道号 eInt：触发 eSOC 采样的中断号（触发源）
void ADC_PowerUp(void)	ADC 上电
void ADC_SetGainAndOffset(ADC_SocEnum eSOC)	设置 SOC 通道的增益及偏差值 eSOC：对应的 SOC 通道号
void ADC_SetSampleAndConvertTime(ADC_SocEnum eSOC, uint32_t u32SampleTimeNs, uint32_t u32ConvTimeNs)	设置采样及转换时间 eSOC：对应的 SOC 通道号 u32SampleTimeNs：采样时间 u32ConvTimeNs：转换时间
void ADC_SelectPinSingleEnded (ADC_SocEnum eSOC, uint8_t u8PinSel, ADC_TriggerSourceEnum eTrigSrc)	配置 ADC 对应 SOC 通道为单端模式 eSOC：对应的 SOC 通道号 u8PinSel：被采样的 ADC 管脚号 eTrigSrc：SOC 采样的触发源
void ADC_SelectPinDifferetial(	配置 ADC 对应 SOC 通道为双端模式

ADC_SocEnum eSOC, ADC_PinSelEnum u8PinSel_1, ADC_PinSelEnum u8PinSel_2, ADC_TriggerSourceEnum eTrigSrc)	eSOC：对应的 SOC 通道号 u8PinSel_1：双端中其中一个 ADC 管脚 u8PinSel_2：双端中另一个 ADC 管脚 eTrigSrc：SOC 采样的触发源
void ADC_EasyInit1(ADC_SocEnum eSOC, uint8_t u8PinSel, ADC_TriggerSourceEnum eTrigSrc)	初始化对应 ADC SOC 通道作为单端，并使能工作 eSOC：对应的 SOC 通道号 u8PinSel：被采样的 ADC 管脚号 eTrigSrc：SOC 采样的触发源
void ADC_EasyInit2(ADC_SocEnum eSOC, uint8_t u8PinSel_1, uint8_t u8PinSel_2, ADC_TriggerSourceEnum eTrigSrc)	初始化对应 ADC SOC 通道作为双端，并使能工作 eSOC：对应的 SOC 通道号 u8PinSel_1：双端中其中一个 ADC 管脚 u8PinSel_2：双端中另一个 ADC 管脚 eTrigSrc：SOC 采样的触发源
ADC_TriggerSourceEnum ADC_TrigFromPWMxSOCA(PWM_REGS* PWMx)	根据 PWM 的模式获取此时 ADC 采样器 SOCA 的触发源应该设置为什么 PWMx：PWM 模组基址
ADC_TriggerSourceEnum ADC_TrigFromPWMxSOCB(PWM_REGS* PWMx)	根据 PWM 的模式获取此时 ADC 采样器 SOCB 的触发源应该设置为什么 PWMx：PWM 模组基址
ADC_TriggerSourceEnum ADC_TrigFromPWMxSOCC(PWM_REGS* PWMx)	根据 PWM 的模式获取此时 ADC 采样器 SOCC 的触发源应该设置为什么 PWMx：PWM 模组基址
void ADC_SetExternalSOC(GPIO_PinEnum ePinNum, uint8_t ePinLevel)	设置外部 GPIO 触发 SOC 采样 ePinNum：外部 GPIO 号 ePinLevel：触发采样的 GPIO 电平
void ADC_PPUInit(ADC_PPUEnum ePPU, uint8_t u8DataSel, int32_t i32Ref, ADC_PPUPolEnum ePol)	初始化 PPU 子模组，并使能工作 ePPU：PPU 子模组基址 u8DataSel：作用于 PPU 的 SOC 通道 i32Ref：PPU 子模组的参考数值 ePol：PPU 子模组计算结果的极性（极性不同会影响结果的正负号）
int32_t ADC_CalculateTemperature(ADC_SocEnum eSOC)	利用 ADC 模组计算 MCU 内部温度传感器感知的温度数值（此函数计算出的结果精确度不高）
int32_t ADC_CalculatePreciseTemperature(ADC_SocEnum eSOC)	利用 ADC 模组计算 MCU 内部温度传感器感知的温度数值（此函数计算出的结果具有较高精确度）

3.3 ADC 的增益误差和失调

下图为 SPC11X8/SPD11X8 仿真电路构架，ADC 为下图套色部分：

图 3-1: ADC Offset and Gain



ADC 增益理想情况为 1，增益误差为ΔGADC,失调为 OSADC，输入信号为V_{in}，输出码值为D_{out}。所以存在误差的情况下，ADC 的输出D_{out}为：

$$D_{out} = (1 + \Delta\text{GADC}) * \frac{V_{in}+\text{OSADC}}{3.657} * 8192 \quad (3\text{-}1)$$

假设 ADC 的增益误差1%，失调为1mV。输入差分电压V_{in}为1V，则输出码值D_{out}为 2264。这个结果相对于理想值 2240 存在1.07%的偏差。

3.4 ADC 输出码值温漂

实际环境中，ADC 的增益和失调都会随着温度变化。考虑温度变化的情况下，假设 ADC 的增益温漂为 TGADC 和失调温漂 TOSADC，为公式（3-1）将变为：

$$D_{out} = (1 + \Delta GADC) * (1 + TGADC) * \frac{V_{in}+\text{OSADC}+\text{TOSADC}}{3.657} * 8192 \quad (3\text{-}2)$$

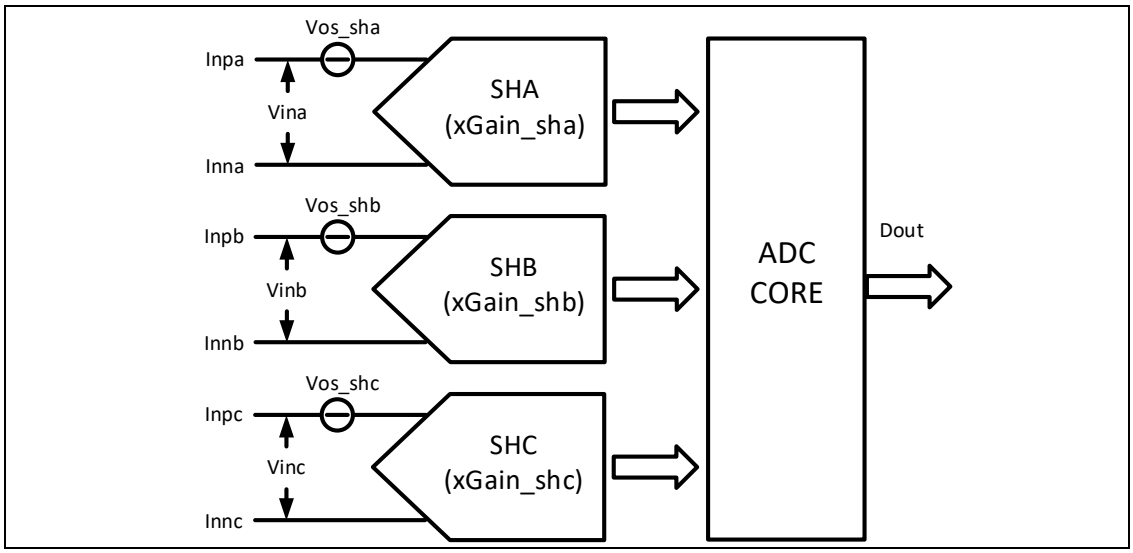
假设增益温漂为50ppm/℃，失调温漂为0.03mV/℃。输入差分电压V_{in}为1V，如果温度变化50℃，则输出码值D_{out}为 2273。这个结果相对于常温结果 2264 存在0.4%的偏差。

4 增益误差和失调校准

4.1 ADC 输出码值校准

由 2.2 和 2.3 章节可知，由于增益误差、失调，以及温度变化，ADC 输出结果存在偏差，导致了信号链的准确度，影响客户应用。下面介绍校准这些误差的方法，客户可以根据需要在软件里面校准这个误差，后续章节将会给出示例代码。

图 4-1: ADC 采样单元示意图



在 SPC11X8/SPD11X8 内部，有三个采样保持器 SHA，SHB 和 SHC，他们的增益误差和失调不同，需要分别校准。我们利用 ADCOFFSETA/ADCGAINA 存储 SHA 的失调和增益校准参数，ADCOFFSETB/ADCGAINB 存储 SHB 的失调和增益校准参数，ADCOFFSETC/ADCGAINC 存储 SHC 的失调和增益校准参数。

为了校准 SHA，SHB 和 SHC，需要分别测量他们的增益误差和失调。测量 SHA 增益误差和失调校准参数的过程如下：

- 第一步，SHA 正端输入和负端输入到 AGND。用 ADC 测到的输出码值为D_{out1}。
- 第二步，SHA 正端输入3.3V，负端输入 AGND。用 ADC 测到的输出码值为D_{out2}。

$$\mathrm{D1(0V)} = 0 = \left(1 + \Delta G_{ADC\_SHA}\right) * \left(D_{out1} + OS_{ADC\_SHA}\right) \quad (4\text{-}1)$$

$$\mathrm{D2(3.3V)} = 7392 = \left(1 + \Delta G_{ADC\_SHA}\right) * \left(D_{out2} + OS_{ADC\_SHA}\right) \quad (4\text{-}2)$$

$$\Delta G_{ADC\_SHA} = \frac{7392}{D_{out2} - D_{out1}} - 1 \quad (4\text{-}3)$$

$$OS_{ADC\_SHA} = \frac{7392}{1 + \Delta G_{ADC_{SHA}}} - D_{out2} \quad (4\text{-}4)$$

将OS_{ADC_SHA}存入 ADCOFFSETA。ΔG_{ADC_SHA}是小数，将其左移 15 位，存入 ADCGAINA。SHB 和 SHC 的测量过程和 SHA 类似，最终将 OSADC_SHB 存入 ADCOFFSETB，ΔGADC_SHB 左移 15 位存入 ADCGAINB，OSADC_SHC 存入 ADCOFFSETC，ΔGADC_SHC 左移 15 位存入 ADCGAINC。得到校准参数以后，在对于任意的D_{out}，只需要按照如下公式计算，即可得到正确结果。对于 SHA 输入的信号有：

$$D_{OUT\_SHA\_CAL} = \left(1 + \Delta G_{ADC\_SHA}\right) * \left(D_{OUT\_SHA} + OS_{ADC\_SHA}\right) \quad (4\text{-}5)$$

对于 SHB 输入的信号有：

$$D_{OUT\_SHB\_CAL} = \left(1 + \Delta G_{ADC\_SHB}\right) * \left(D_{OUT\_SHB} + OS_{ADC\_SHB}\right) \quad (4\text{-}6)$$

对于 SHC 输入的信号有：

$$D_{OUT\_SHC\_CAL} = \left(1 + \Delta G_{ADC\_SHC}\right) * \left(D_{OUT\_SHC} + OS_{ADC\_SHC}\right) \quad (4\text{-}7)$$

4.2 校准 ADC 函数

芯片在出厂前，Spintrol 对每一颗 ADC 进行校准，将校准参数存储在 ADCOFFSETA/ADCGAINA，ADCOFFSETB/ADCGAINB 和 ADCOFFSETC/ADCGAINC 中。同时 Spintrol 也提供了 ADC 校准示例代码，客户只要根据要求配置 ADC 的输入通道，调用校准函数，即可得到校准参数。ADC 的通道配置如下：

ADC 校准示例

```
#include"SPC11x8/SPD11x8.h"
#include<stdio.h>
/* Ideal code for 3.3V input related to +-3.657143V range.
7392(ideal_code)*32768 */
#define    REF3V3_CODE         242221056
/* Ideal code for 1.2V input related to +-3.657143V range.
2688(ideal_code)*32768 */
#define    REF1V2_CODE         88080384
/* Ideal ADC LSB Votage: 0.44643mV * 65536 */
#define    LSB                 29257
/* Unit: time */
#define    NREP                32
/* Unit: us */
#define    TDLY                1

int32_t    voltage;
int16_t    adc_sha_offset   =0;      /* sampler A offset [-80,80] */
int16_t    adc_shb_offset   =0;      /* sampler B offset [-80,80] */
int16_t    adc_shc_offset   =0;      /* sampler C offset [-80,80] */
uint32_t   adc_sha_gain     =0;      /* sampler A Gain [31130,34406] */
uint32_t   adc_shb_gain     =0;      /* sampler A Gain [31130,34406] */
uint32_t   adc_shc_gain     =0;      /* sampler A Gain [31130,34406] */

int main()
{
  FLASH_WALLOW();
  FLASH_SetTiming(200000000);
  /* Disable flash write access after flash operation had done */
  FLASH_WDIS();

  CLOCK_InitWithRCO(CLOCK_HCLK_200MHZ);

  Delay_Init();

  GPIO_SetPinChannel(GPIO_34,GPIO34_UART_TXD);
  GPIO_SetPinChannel(GPIO_35,GPIO35_UART_RXD);
```

```
  CLOCK_EnableModule(UART_MODULE);
  UART_Init(UART,38400);

  printf("\n************** ADC Calibration Test Start
******************\n");

  /************************* ADC configuration *****************/
  /* Power up ADC */
  ADC_PowerUp();

  /* Div ADC clock */
  // ADC Clock divider control : 1 means the clock frequency = systemclk
    / 2
  CLOCK->ADCCLKCTL.bit.DIV = 0x1F;

  // Enable ADC Interrupt Flag
  ADC->ADCIE.all = 0x0000ffff;

  Delay_Us(100);

  // 875ns sample time
  ADC->ADCSOCCTL[0].bit.SAMPCNT = 127;
  // 125ns conversion time
  ADC->ADCSOCCTL[0].bit.CONVCNT = 8;
  // averaging 16 times
  ADC->ADCSOCCTL[0].bit.AVGCNT  = 4;
  // trigger source selection: 0 means software trigger
  ADC->ADCSOCCTL[0].bit.TRIGSEL = 0;
  // positve terminal select AGND
  ADC->ADCSOCCTL[0].bit.CHSELP  = 0;
  // negative terminal select AGND
  ADC->ADCSOCCTL[0].bit.CHSELN  = 0;
  // sampler enable: 1 means sampler A enable
  ADC->ADCSOCCTL[0].bit.SHEN    = 1;
  // clear INT0
  ADC->ADCIC.bit.INT0 = 1;
  // Software trigger SOC0
  ADC->ADCSOCFRC.bit.SOC0 = 1;
  // wait for ADC convertion done
  while(ADC->ADCIF.bit.INT0 != 1) {};

  // 875ns sample time
  ADC->ADCSOCCTL[1].bit.SAMPCNT = 127;
 // 125ns conversion time
```

```
 ADC->ADCSOCCTL[1].bit.CONVCNT = 8;
// averaging 16 times
 ADC->ADCSOCCTL[1].bit.AVGCNT  = 4;
// trigger source selection: 0 means software trigger
 ADC->ADCSOCCTL[1].bit.TRIGSEL = 0;
// positve terminal select ADC0  (3.3V)
 ADC->ADCSOCCTL[1].bit.CHSELP  = 4;
// negative terminal select ADC1 (0V)
 ADC->ADCSOCCTL[1].bit.CHSELN  = 4;
// sampler enable: 1 means sampler A enable
 ADC->ADCSOCCTL[1].bit.SHEN    = 1;
 ADC->ADCIC.bit.INT1 = 1;
 // Software trigger SOC1
 ADC->ADCSOCFRC.bit.SOC1 = 1;
 while(ADC->ADCIF.bit.INT1 != 1) {};

 // 875ns sample time
 ADC->ADCSOCCTL[2].bit.SAMPCNT = 127;
 // 125ns conversion time
 ADC->ADCSOCCTL[2].bit.CONVCNT = 8;
 // averaging 16 times
 ADC->ADCSOCCTL[2].bit.AVGCNT  = 4;
 // trigger source selection: 0 means software trigger
 ADC->ADCSOCCTL[2].bit.TRIGSEL = 0;
 // positve terminal select AGND
 ADC->ADCSOCCTL[2].bit.CHSELP  = 0;
 // negative terminal select AGND
 ADC->ADCSOCCTL[2].bit.CHSELN  = 0;
 // sampler enable: 2 means sampler B enable
 ADC->ADCSOCCTL[2].bit.SHEN    = 2;
 ADC->ADCIC.bit.INT2 = 1;
 // Software trigger SOC2
 ADC->ADCSOCFRC.bit.SOC2 = 1;
 while(ADC->ADCIF.bit.INT2 != 1) {};

 // 875ns sample time
 ADC->ADCSOCCTL[3].bit.SAMPCNT = 127;
 // 125ns conversion time
 ADC->ADCSOCCTL[3].bit.CONVCNT = 8;
 // averaging 16 times
 ADC->ADCSOCCTL[3].bit.AVGCNT  = 4;
 // trigger source selection: 0 means software trigger
 ADC->ADCSOCCTL[3].bit.TRIGSEL = 0;
 // positve terminal select ADC0  (3.3V)
```

```
ADC->ADCSOCCTL[3].bit.CHSELP  = 4;
// negative terminal select ADC1 (0V)
ADC->ADCSOCCTL[3].bit.CHSELN  = 4;
// sampler enable: 2 means sampler B enable
ADC->ADCSOCCTL[3].bit.SHEN    = 2;
ADC->ADCIC.bit.INT3 = 1;
// Software trigger SOC3
ADC->ADCSOCFRC.bit.SOC3 = 1;
while(ADC->ADCIF.bit.INT3 != 1) {};

// 875ns sample time
ADC->ADCSOCCTL[4].bit.SAMPCNT = 127;
// 125ns conversion time
ADC->ADCSOCCTL[4].bit.CONVCNT = 8;
// averaging 16 times
ADC->ADCSOCCTL[4].bit.AVGCNT  = 4;
// trigger source selection: 0 means software trigger
ADC->ADCSOCCTL[4].bit.TRIGSEL = 0;
// positve terminal select AGND
ADC->ADCSOCCTL[4].bit.CHSELP  = 0;
// negative terminal select AGND
ADC->ADCSOCCTL[4].bit.CHSELN  = 0;
// sampler enable: 3 means sampler C enable
ADC->ADCSOCCTL[4].bit.SHEN    = 3;
ADC->ADCIC.bit.INT4 = 1;
// Software trigger SOC4
ADC->ADCSOCFRC.bit.SOC4 = 1;
while(ADC->ADCIF.bit.INT4 != 1) {};

// 875ns sample time
ADC->ADCSOCCTL[5].bit.SAMPCNT = 127;
// 125ns conversion time
ADC->ADCSOCCTL[5].bit.CONVCNT = 8;
// averaging 16 times
ADC->ADCSOCCTL[5].bit.AVGCNT  = 4;
// trigger source selection: 0 means software trigger
ADC->ADCSOCCTL[5].bit.TRIGSEL = 0;
// positve terminal select ADC0 (3.3V)
ADC->ADCSOCCTL[5].bit.CHSELP  = 4;
// negative terminal select ADC1 (0V)
ADC->ADCSOCCTL[5].bit.CHSELN  = 4;
// sampler enable: 3 means sampler C enable
ADC->ADCSOCCTL[5].bit.SHEN    = 3;
ADC->ADCIC.bit.INT5 = 1;
```

```
  // Software trigger SOC5
  ADC->ADCSOCFRC.bit.SOC5 = 1;
  while(ADC->ADCIF.bit.INT5 != 1) {};

  printf("SHA Measure AGND  Code is %d \n", ADC->ADCRESULT[0].all);
  printf("SHA Measure VDD33 Code is %d \n", ADC->ADCRESULT[1].all);
  printf("SHB Measure AGND  Code is %d \n", ADC->ADCRESULT[2].all);
  printf("SHB Measure VDD33 Code is %d \n", ADC->ADCRESULT[3].all);
  printf("SHC Measure AGND  Code is %d \n", ADC->ADCRESULT[4].all);
  printf("SHC Measure VDD33 Code is %d \n", ADC->ADCRESULT[5].all);

  adc_sha_offset = ADC->ADCRESULT[0].all;
  adc_sha_gain = REF3V3_CODE / ( ADC->ADCRESULT[1].all -
                                 ADC->ADCRESULT[0].all );

  adc_shb_offset = ADC->ADCRESULT[2].all;
  adc_shb_gain = REF3V3_CODE / ( ADC->ADCRESULT[3].all -
                                 ADC->ADCRESULT[2].all );

  adc_shc_offset = ADC->ADCRESULT[4].all;
  adc_shc_gain = REF3V3_CODE / ( ADC->ADCRESULT[5].all -
                                ADC->ADCRESULT[4].all );

  ADC->ADCOFFSETA.all = adc_sha_offset;
  ADC->ADCOFFSETB.all = adc_shb_offset;
  ADC->ADCOFFSETC.all = adc_shc_offset;

  ADC->ADCGAINA.all = adc_sha_gain;
  ADC->ADCGAINB.all = adc_shb_gain;
  ADC->ADCGAINC.all = adc_shc_gain;

  printf("ADC SHA Offset: %d \n", adc_sha_offset);
  printf("ADC SHA Gain:   %d \n", adc_sha_gain);
  printf("ADC SHB Offset: %d \n", adc_shb_offset);
  printf("ADC SHB Gain:   %d \n", adc_shb_gain);
  printf("ADC SHC Offset: %d \n", adc_shc_offset);
  printf("ADC SHC Gain:   %d \n", adc_shc_gain);

  printf("\n*************** ADC Calibration Test End
******************\n");

  if(adc_sha_offset > 80 || adc_sha_offset < -80 || adc_sha_gain > 34406
|| adc_sha_gain < 31130 ||
     adc_shb_offset > 80 || adc_shb_offset < -80 || adc_shb_gain > 34406
```

```
|| adc_shb_gain < 31130 ||
     adc_shc_offset > 80 || adc_shc_offset < -80 || adc_shc_gain > 34406
|| adc_shc_gain < 31130)
    return ERROR;
  else
    return SUCCESS;
}
```

4.3 PGA 输出电压校准

由章节 2.2 和 2.3 可知，由于增益误差、失调，以及温度变化，PGA 输出结果存在偏差，导致了信号链的准确度，影响客户应用。下面介绍校准这些误差的方法，客户可以根据需要在软件里面校准这个误差。

由于 PGA 需要使用 ADC 测量，所以首先要对 ADC 进行校准，得到 ADC 的校准参数。每颗芯片在出厂前会进行 ADC 的校准，所以下面的校准过程假设 ADC 的输出是已经校准后的结果。PGA 在差分模式，共模输入端设为负输入端。增益为 GPGA，增益误差为ΔGPGA,失调为 OSPGA，正端输入信号为V_{inp}，负端输入为V_{inn}，正端输出电压为V_{outp}，负端输出电压为V_{outn}。测量 PGA 增益误差和失调校准参数的过程如下：

- 第一步，PGA 设为 bypass 模式，在 PGA 正端输入$\frac{AVDD}{2}+V_{in1}$（V_{in1}是一个较小值）和负端输入$\frac{AVDD}{2}$，用 ADC 测到结果为D_{out_byp1}。
- 第二步，PGA 设为放大模式，在 PGA 正端输入$\frac{AVDD}{2}+V_{in1}$和负端输入$\frac{AVDD}{2}$。用 ADC 测到的输出码值为D_{out_amp1}。
- 第三步，PGA 设为 bypass 模式，在 PGA 正端输入$\frac{AVDD}{2}-V_{in2}$（V_{in2}是一个较小值），负端输入$\frac{AVDD}{2}$。用 ADC 测到的输出码值为D_{out_byp2}。
- 第四步，PGA 设为放大模式，在 PGA 正端输入$\frac{AVDD}{2}-V_{in2}$，负端输入$\frac{AVDD}{2}$。用 ADC 测到的输出码值为D_{out_amp2}。

得到D_{out_byp1}, D_{out_amp1}, D_{out_byp2}和D_{out_amp2}以后按照如下公式计算校准参数：

$$D_{out\_amp1}=\text{Gain}*\left(1+\Delta G_{PGA\_SHA}\right)*\left(D_{out\_byp1}+OS_{PGA\_SHA}\right) \quad (4\text{-}8)$$

$$D_{out\_amp2}=\text{Gain}*\left(1+\Delta G_{PGA\_SHA}\right)*\left(D_{out\_byp2}+OS_{PGA\_SHA}\right) \quad (4\text{-}9)$$

$$\Delta G_{PGA\_SHA}=\frac{\frac{D_{out\_amp2}-D_{out\_amp1}}{D_{out\_byp2}-D_{out\_byp1}}}{Gain}-1 \quad (4\text{-}10)$$

$$OS_{PGA\_SHA}=\frac{\frac{D_{out\_amp2}}{1+\Delta G_{PGA\_SHA}}}{Gain}-D_{out\_byp2} \quad (4\text{-}11)$$

将OS_{PGA_SHA}和ΔG_{PGA_SHA}存为常量，在后续 ADC 输出结果中校准 PGA。

$$D_{OUT\_PGA\_CAL}=\left(1+\Delta G_{PGA\_SHA}\right)*\left(OS_{PGA\_SHA}\right) \quad (4\text{-}12)$$

4.4 校准 PGA 函数

客户可以根据 4.3 章节提供的方法得到 PGA 的校准参数，Spintrol 也提供了 PGA 校准代码示例，客户只要根据自己的需求配置该函数，即可得到相应的校准参数。该函数配置示例如下：

ADC 校准示例

```
#include"SPC11x8/SPD11x8.h"
#include<stdio.h>
/* Ideal ADC LSB Votage: 0.44643mV * 65536 */
#define    LSB                29257
/* the gain value selected */
#define    PGAGAINSEL         PGA_SCALE_4X
 // ( for single-ended mode : 0:1x 1:2x 2:4x 3:8x 4:12x 5:16x 6:24x 7:32x
 //   for differential mode : 0:2x 1:4x 2:8x 3:16x 4:24x 5:32x 6:48x 7:64x)
/* the PGA channel to be calibrated */
#define    PGACHANNELSEL      PGA0
/* 1.65v code for DAC */
#define    VOLTAGE_1_6_5_v    512
volatile   uint32_t CODE1[8] = {913,866,800,759,706,663,614,573};
volatile   uint32_t CODE2[8] = {483,433,389,336,286,236,196,172};

int32_t    i32result_in1;
int32_t    i32result_out1;
int32_t    i32result_in2;
int32_t    i32result_out2;
float      fgain_measure;
float      foffset_measure;

int32_t ADC_Measure_ADCCHANNEL_CODE(uint32_t nAvg)
{
  int32_t result_code=0;
  uint32_t i;

// 8us sample time (set as the max)
  ADC->ADCSOCCTL[0].bit.SAMPCNT = 255;

  for(i=0; i<nAvg; i++)
  {
    ADC_SoftwareTrigger(ADC_SOC_0);
    while(ADC->ADCIF.bit.INT0 != 1) {};
    result_code = result_code + ADC_GetResult(ADC_SOC_0);
  }
  result_code = (int32_t)(result_code)/(int32_t)(nAvg);

  return result_code;
```

```
 }

int main()
{
  FLASH_WALLOW();
  FLASH_SetTiming(200000000);
  FLASH_WDIS();

  CLOCK_InitWithRCO(CLOCK_HCLK_200MHZ);

  Delay_Init();

  GPIO_SetPinChannel(GPIO_34,GPIO34_UART_TXD);
  GPIO_SetPinChannel(GPIO_35,GPIO35_UART_RXD);
  CLOCK_EnableModule(UART_MODULE);
  UART_Init(UART,38400);

  printf("\n\n************** PGA Calibration Test Start
******************\n");

  /* the PGA channle to be calibrated */
  /* Enbale COMP model */
  CLOCK_EnableModule(COMP_MODULE);

  /* Direct mode, DAC code immediately update */
  COMP_SetDACCodeLoadTiming(DAC1, DAC0CTL_BIT_CODELOAD_DIRECT_MODE);

  /* Enable DAC1 */
  COMP_EnableDAC(DAC1);

  /* Init PGA mode (DAC1 as the positive input and ADC3 as the negtive
input) */
  switch(PGACHANNELSEL)
  {
    case 0:
         PGA_DifferentialInit(PGACHANNELSEL, PGA0_CH_P_DAC1,
                              PGA0_CH_N_ADC3, PGAGAINSEL);
         /*ADC Init*/
         ADC_EasyInit2(ADC_SOC_0,ADCx_PGA0P,ADCx_PGA0N,
                       ADCTRIG_Software);
         break;
    case 1:
         PGA_DifferentialInit(PGACHANNELSEL, PGA1_CH_P_DAC1,
                              PGA1_CH_N_ADC3, PGAGAINSEL);
```

```
        /*ADC Init*/
        ADC_EasyInit2(ADC_SOC_0,ADCx_PGA1P,ADCx_PGA1N,
                      ADCTRIG_Software);
        break;
    case 2:
        PGA_DifferentialInit(PGACHANNELSEL, PGA2_CH_P_DAC1,
                             PGA2_CH_N_ADC3, PGAGAINSEL);
        /*ADC Init*/
        ADC_EasyInit2(ADC_SOC_0,ADCx_PGA2P,ADCx_PGA2N,
                      ADCTRIG_Software);
        break;
}

PGA->PGA0CTL.bit.BYPP = 1;  /* set positive path as bypass mode */
PGA->PGA0CTL.bit.BYPN = 1;  /* set negtive path as bypass mode */

COMP_SetDACValue10Bit(DAC1,CODE1[PGAGAINSEL]);
Delay_Ms(10);

/* measure input value first time */
i32result_in1 = ADC_Measure_ADCCHANNEL_CODE(1000);

COMP_SetDACValue10Bit(DAC1,CODE2[PGAGAINSEL]);
Delay_Ms(10);

/* measure input value second time */
i32result_in2 = ADC_Measure_ADCCHANNEL_CODE(1000);

/* set positive path as no bypass mode */
PGA->PGA0CTL.bit.BYPP = 0;
/* set negtive path as no bypass mode */
PGA->PGA0CTL.bit.BYPN = 0;
COMP_SetDACValue10Bit(DAC1,CODE1[PGAGAINSEL]);
Delay_Ms(10);

/* measure output value first time */
i32result_out1 = ADC_Measure_ADCCHANNEL_CODE(1000);
COMP_SetDACValue10Bit(DAC1,CODE2[PGAGAINSEL]);
Delay_Ms(10);

/* measure output value second time */
i32result_out2 = ADC_Measure_ADCCHANNEL_CODE(1000);
fgain_measure = (float)
/* the gain value after calibration */
```

```
(i32result_out2-i32result_out1)/(i32result_in2-i32result_in1);
/* the offset value after calibration */
  foffset_measure =
(int32_t)(i32result_out1-i32result_in1*fgain_measure)*LSB/65536;

  printf("\n**************(in1:%d in2:%d out1:%d out2:%d) GAIN : %0.3f,
            OFFSET : %0.3f **************\n",

i32result_in1,i32result_in2,i32result_out1,i32result_out2,fgain_meas
ure,foffset_measure);

  printf("\n************** PGA Calibration Test End
             ******************\n");
}
```

5 修订记录

表 5-1: 文档修订历史

日期	版本	修改内容
2019-07-25	1	初始版本