文档编号：AN2001

上海东软载波微电子有限公司

应用笔记

ES32F027X

修订历史

版本	修订日期	修改概要
V1.0	2019-1-10	初版
V1.1	2020-3-19	1. 添加第 2 章 ES32F027x ADC 应用注意。 2. 变更 Logo。 3. Driver 及应用章节更新
V1.2	2020-10-27	新增 1.9、1.10 UART/I2C/SPI FIFO 相关章节

地址：中国上海市龙漕路 299 号天华信息科技园 2A 楼 5 层
邮编：200235
E-mail：support@essemi.com
电话：+86-21-60910333
传真：+86-21-60914991
网址：http://www.essemi.com/

目录

第1章 ES32F027x应用注意

1.1 开发环境

推荐用户使用 Keil5 、IAR8.11 或者 iDesigner 进行固件开发。由于 Keil4 不支持 PACK 机制，故不推荐用户使用 Keil4。

1.2 寄存器写保护

为避免程序的异常导致运行错误，芯片写保护寄存器用于阻止对被保护的寄存器误操作。

系统控制单元，IWDT、GPIO 等模块支持寄存器写保护，对被保护的寄存器进行写之前需要解除写保护状态（允许写），否则无法对写保护寄存器写入。操作完成后，再使能写保护（禁止写）。库函数中均提供相应宏定义进行解除保护和使能保护。

1.2.1 IWDT写保护

默认条件下，对 IWDT_PR、IWDT_RLR 和 IWDT_WINR 的写访问操作都是受保护的，对 IWDT_KR 寄存器以字写入 0x00005555 解锁码，可以解除写保护，如果写入别的值，将会打破这个顺序，使得对寄存器的访问保护重新生效。这意味着在做重加载操作的时候（向该寄存器写入 0x0000AAAA）就属于这种情况。可以通过一个状态寄存器观察预分频器的更新、看门狗计数器的重加载或窗口值的重加载。

1.2.2 GPIO写保护

对 GPIOx_LCK 寄存器依照特定的写 / 读序列，以字方式写入 0x0001FFFF 会使 GPIOx_MOD、GPIOx_OT、GPIOx_PUD、GPIOx_AFL 和 GPIOx_AFH 进入保护。当正确的锁定序列作用于这个寄存器的位 16 时，LCK[15:0] 的值用来锁定 I/O 口的配置（在写序列期间要保持 LCK[15:0]值不变）。当锁定序列已经作用于一个端口位，该端口位的值再也不能改变直到下一次复位。

1.3 写 1 清零寄存器

有很多中断标志寄存器都是用“写 1 清零”的方式来操作。对于“写 1 清零”的寄存器，不可使用“读-修改-写”的方式来进行“写 1 清零”，否则会引起标志误清，进而产生漏中断的后果。对该类寄存器操作需要以 Word 进行写。

1.4 未使用的GPIO端口处理

系统中未使用的 GPIO 管脚建议设置为输出固定电平并悬空，若设置为输入，须加上拉或下拉电阻接到电源或地。

1.5 系统时钟选择

系统上电默认使用内部 4MHz 高速时钟(HRC)作为系统时钟，几种常用系统时钟配置：

1.5.1 内部高速 4MHz (默认时钟)

使用内部高速 RC 震荡器，频率约为 4MHz，配置如下:

1. 进入到 md_rcu.c 里面的 md_rcu_init 函数更改配置;
2. md_rcu_enable_con_hrcon(RCU);
3. md_rcu_set_cfg_sw(RCU, MD_RCU_SYSCLK_HRC);

1.5.2 外部时钟HOSC (4~32MHz)

外部高速时钟要求为 4MHz 的倍数，如：4MHz、8MHz、12MHz、16MHz、20MHz、24MHz、32MHz。

首先要确认焊接了外部高速时钟，并已知外部高速时钟的频率，假如外部高速时钟为 20MHz，则配置方式如下：

1. 进入到 md_rcu.c 里面的 md_rcu_init 函数更改配置;
2. md_rcu_enable_con_hoscon(RCU);
3. md_rcu_set_cfg_sw(RCU, MD_RCU_SYSCLK_HOSC);

1.5.3 内部锁相环倍频时钟 (4~48MHz，使用HRC倍频)

使用内部锁相环倍频时钟，选择 HRC 做为参考时钟源，需设定 PLL0 倍频系数，如：4MHz、8MHz、12MHz、16MHz、20MHz、24MHz、….、48MHz，配置如下:

1. 进入到 md_rcu.c 里面的 md_rcu_init 函数更改配置;
2. md_rcu_enable_con_hrcon(RCU);
3. md_rcu_enable_con_pll0on (RCU);
4. md_rcu_set_cfg_pllsrc(RCU, MD_RCU_PLL_SRC_HRC);
5. md_rcu_set_cfg_pllmul(RCU, X); (X 为需设定 PLL0 倍频系数);
6. md_rcu_set_cfg_sw(RCU, MD_RCU_SYSCLK_PLL0);

1.5.4 外部锁相环倍频时钟 (4~48MHz，使用HOSC倍频)

使用外部锁相环倍频时钟，选择 HOSC 做为参考时钟源时，首先要确认焊接了外部高速时钟，并已知外部高速时钟的频率，并且确认外部时钟为 4MHz，如果大于 4MHz，则需先除频，再使用 PLL0 配频系数，达到需求频率，如:4MHz、8MHz、12MHz、16MHz、20MHz、24MHz、….、48MHz，则配置方式如下：

1. 进入到 md_rcu.c 里面的 md_rcu_init 函数更改配置;
2. md_rcu_enable_con_hoscon(RCU);
3. md_rcu_enable_con_pll0on (RCU);
4. md_rcu_set_cfg_hoscdiv(RCU, 1); (依需求设定 hoscon 除频系数)
5. md_rcu_set_cfg_pllsrc(RCU, MD_RCU_PLL_SRC_HRC);

6. md_rcu_set_cfg_pllmul(RCU, X);　(X 为需设定 PLL0 倍频系数)
7. md_rcu_set_cfg_sw(RCU, MD_RCU_SYSCLK_PLL0);

1. 5. 5　外部低速时钟(LOSC)

首先要确认焊接了外部低速时钟，配置方式如下：

1. 进入到 md_rcu.c 里面的 md_rcu_init 函数更改配置;
2. md_rcu_enable_con_loscon(RCU);
3. md_rcu_set_cfg_sw(RCU, MD_RCU_SYSCLK_LOSC);

需要注意的是，当系统时钟配置为低速时钟时(低于 1MHz)，SysTick 中断将会被迫关闭。ALD 提供的延迟类函数禁止使用。

1. 6　GPIO模块

问题描述: UART reset 时，打印出来的第一个字符会有漏字或乱码现象。

解决方案: 在任何外设要设置 IO mux 时，需先切换 AF fucntion 再切换 GPIO mode。

1. 7　RTC模块

问题描述: 当 RTC 在 LOSC 系统时钟源切换至其他时钟源时，会造成无法切换的情况。

解决方案: 一开始于 md_rcu.c 里开启 LOSC。

1. 8　库函数选择

ES32 系列芯片提供两种类型库函数 ALD 和 MD：

ALD：提供较为完善的封装，提供更为人性化的 API，适合大部分用户；

MD：基本上只提供寄存器位域级别的“读”、“写”接口，适合对芯片底层较为熟悉的用户。

如果用户对速度不是要求非常严格，一般情况下推荐用户使用 ALD 库。可以大大节约用户产品开发周期。

1. 9　UART/I2C/SPI FIFO 注意事项

1. Tx 中断使用: 如需在中断程序中固定写入 N 笔 Data 到 Tx FIFO，必须在 Data 写入 FIFO 前确认 UART_STAT 中 TFFULL 旗标，确保 Tx FIFO 不是 Full 的状态。或是将 N 笔 Data 写入 FIFO 后才使用 ICR 清除中断。
2. Rx 中断使用: 当 Rx 触发 RFTH 中断后，使用者必须将 FIFO Data 读空或是至少读至小于阀值，否则 RFTH 将不会再触发。

注: 下一章节会详细说明细节

1. 10 UART/I2C/SPI FIFO中断应用范例说明

◆ 配置 TFTH 8 触发中断，在中断程序中固定写入 8 笔 Data 到 FIFO，造成 Data 丢失的可能状况。

正常状态下当 FIFO 从 9→8 时发生中断，中断程序中再写入 8 笔 Data 到 FIFO，FIFO 可正确存储所欲传送的八字节数据，FIFO 里的 Data 不会超过 16 笔，这样的使用状况下，不会造成欲传送数据丢失状况。

如果在发生中断后，在中断程序在填入新的 8 笔 Data 的同时，有其他优先权更高的中断将之打断，就可能造成数据丢失状况。

以下举例说明:
当 FIFO 从 9→8 发生中断 1，准备写入 8 笔 Data，在写入 2 笔后 FIFO 内数据到达 10 笔，此时发生了高优先权中断导致传送暂停，但还有 6 笔数据等待传送，在执行高优先权中断时势必会发生 FIFO 从 10→9→8 又产生一次 TFTH 中断 2。等到高优先权中断执行完毕，会回到中断 1 程序中将未传完的 6 笔 Data 传完，此时 FIFO 个数应为 9~16 之间，接续发生中断 2 后所要传送的 8 笔 data,就会发生 FIFO 数>16 状况，造成 FIFO Overflow，未传入的 Data 即丢失。

以下利用程序仿真中断程序被高优先权中断插队的状况，此范例以 md_tick_waitms(10,70) 模拟高优先权中断插队所需的运行时间:

```
charwords2[8]={'Z','Z','Z','Z','Z','Z','Z','Z'};
charwords3[26]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'};
char*ptr2=words2;
char*ptr3=words3;
externuint8_tT;
/**
  * @brief UART1_IRQHandler.
  * @param none
  * @retval none
  */
voidUART1_IRQHandler(void)/*    IRQ 27 */
{
uint32_ti;
md_uart_clear_flag_tfth(UART1);
if(T)//First enable
{
T=0;
for(i=0;i<8;i++)
{
md_uart_send_txbuf(UART1,*ptr2);
ptr2++;
if(ptr2==words2+26)ptr2=words2;
```

```
}
}
else
{
for(i=0;i<8;i++)
{
if(i==2)
{
md_tick_waitms(10,70);
}
md_uart_send_txbuf(UART1,*ptr3);
ptr3++;
if(ptr3==words3+26)ptr3=words3;
}
}
while(md_uart_is_active_flag_stat_tfoerr(UART1))
{
printf("FIFO OVERFLOW\r\n");
while(1);
}
}
```

要如何避开此问题以下提供两个方式解决：

Case 1:在写入 8 笔 Data 当中，每一笔皆要去确认 FIFO 是否为 Full 的状况，这样就可以避免 FIFO overflow 状态。

```
else
{
for(i=0;i<8;i++)
{
while(md_uart_is_active_flag_stat_tffull(UART1));
if(i==2)
{
md_tick_waitms(10,70);
}
md_uart_send_txbuf(UART1,*ptr3);
ptr3++;
if(ptr3==words3+26)ptr3=words3;
}
}
```

Case 2:一般我们会习惯在一进中断就以 ICR 清除中断标志位，如果移到资料传送完之后才清除中断 ICR，这方法也可以避免 overflow 发生，但是主要是因为在做 ICR 时将上述所说的中断 1 与中断 2 同时清除了，其实中断 2 并无执行，如果在应用上中断 2 是必须的就应避免使用此方法。

```
voidUART1_IRQHandler(void)/*    IRQ 27 */
{
uint32_ti;
      /* md_uart_clear_flag_tfth(UART1); 移到中断程序离开前清除 */
if(T)//First enable
{
T=0;
for(i=0;i<8;i++)
{
md_uart_send_txbuf(UART1,*ptr2);
ptr2++;
if(ptr2==words2+26)ptr2=words2;
}
}
else
{
for(i=0;i<8;i++)
{
while(md_uart_is_active_flag_stat_tffull(UART1));
if(i==2)
{
md_tick_waitms(10,70);
}
md_uart_send_txbuf(UART1,*ptr3);
ptr3++;
if(ptr3==words3+26)ptr3=words3;
}
}
      while(md_uart_is_active_flag_stat_tfoerr(UART1))
{
printf("FIFO OVERFLOW\r\n");
while(1);
}
md_uart_clear_flag_tfth(UART1);
}
```

◆ 使用 RFTH 做为中断，RFTH 中断的触发条件为当 RX FIFO 内数据大于设定的阀值时产生，以 RFTH=4 为例，必须发生 FIFO 数据从 3→4 时才会触发，所以如果在 RX FIFO 触发中断后没有去将 FIFO 里的资料读空或至少拿走直到 FIFO 小于阀值，接下来将不会再触发 RFTH 中断。

以 MD 例程”UART_Rece_Send_By_IT”来看，如果将读 FIFO 这段代码拿掉，也就是当 FIFO 触发中断后并不去读 FIFO 值，接下来就不会再触发此中断，造成所谓的接收死机状态。

```
printf("UART MD Rece_Send_By_IT Example (Rx Threshold = 4)\r\n");

while(1)
{
if(uart1_interrupt==1)
{
uart1_interrupt=0;
//          ReceiveByte = md_uart_recv(UART1);
//          printf("ReceiveByte:%d\r\n",ReceiveByte);
}
}
```

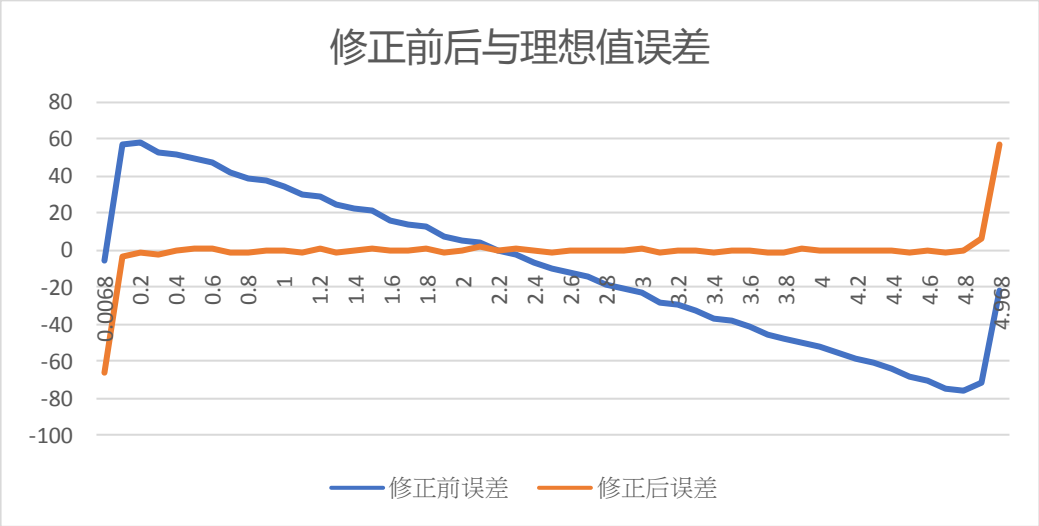
第2章 ES32F027x ADC应用注意

2.1 校正的原因

ADC 的量测结果与理想的 ADC 值有偏差的状况，主要原因有二个：

1. 前端 PGA 放大器，因为不是 rail-to-rail，因此无法量测 Vdd – Gnd 满刻度范围，可量测范围为(Vdd-150mV) – (Gnd+150mv)。
2. ES32F027X SAR ADC 内部电压与分压器件因 IC 制造过程中芯片与芯片间存在一定的差异，因为芯片内部未实现校准机制，因此输入端电压对应 ADC 数值与理想值有所偏差，必须通过软件补偿的方式，将量测所得出的数值做校准，使 ADC 趋近理想曲线。

由下图可看出修正前误差值与经过修正后误差，但从此图中都可看出前后点会有非线性的部分，这是由于我们 ADC 的 PGA 关系，导致输入 0V 并不是真实 0V，所以我们实际可使用的范围为 0.15V~4.85V 之间。



2.2 校准方式

取两个校准方式，目前取 1V 与 2V 所量测出来的值当作校准参考点，将所获得 ADC 值存入 Flash 中 (C/P 生产过程中写入)，通过线性方程式 y=ax+b 的方式求出 a 与 b 的值。

a：斜率 b：offset

1. Reference_V1：参考点 1V
2. Reference_V2：参考点 2V
3. Reference_V1_ADCValue：参考点 1V 量测到的 ADC 值，存入 Flash 中
4. Reference_V2_ADCValue：参考点 2V 量测到的 ADC 值，存入 Flash 中
5. Ideal_V1：参考点 1V 理想 ADC 值
6. Ideal_V2：参考点 2V 理想 ADC 值
7. Offset ：渐进线，方程式求出的 b 值
8. coefficient：系数

根据方程式 y = ax + b 求出 Offset 值(方程式中的 b 值)

offset 简易公式：

$$Offset = \frac{(\text{Reference\_V2} * \text{Reference\_V1\_ADCValue} - \text{Reference\_V1} * \text{Reference\_V2\_ADCValue})}{(\text{Reference\_V2} - \text{Reference\_V1})}$$

参考点 1 理想值(1V 理想值)：

$$\text{Ideal\_V1} = \frac{\text{Reference\_V1}}{5} * 4096$$

系数：

$$coefficient = \frac{Ideal\_V1}{(\text{Reference\_V1\_ADCValue} - \text{Offset})}$$

校正值：

Calibration ADC = (修正前 ADC 值 - Offset) * coefficient

Ex：

1. Reference_V1：1V
2. Reference_V2： 2V
3. Reference_V1_ADCValue： 784.81
4. Reference_V2_ADCValue：1632.81
5. Ideal_V1： 819.2
6. Ideal_V2：1638.4
7. Offset： (2*784.81-1*1632.81)/(2-1) = -63.19
8. coefficient ：819.2/(784.81-(-63.19)) =0.966

Result：(784.81-(-63.19))*0.966 = 819.168

由结果可看出在还没校准前的值为 784.81，校准后的值为 819.168，接近理想值 819。

2. 3 Driver及应用

◆ md_adc.c

Function 说明

1. md_adc_init(ADCx,*ADC_InitStruct,*channel) : 带入 ADC 的初始设定
2. md_adc_calibration_data(vdd,vref,*offset,*coefficient) :带入需要的 VDD 值及 VREF 值，三组设定值提供使用者使用，分别为 case1: vdd_5v/vref_5v、case2: vdd_3.3v/vref_3.3v、case3:vdd_5v/vref_2.5v (存 1V/2V 参考点的值)，function 会根据所选 case 从 Flash 缓存器里将对应的值取出做运算，得到 Offset 与 Coefficient

$$Offset = \frac{(\text{Reference\_V2} * \text{Reference\_V1\_ADCValue} - \text{Reference\_V1} * \text{Reference\_V2\_ADCValue})}{(\text{Reference\_V2} - \text{Reference\_V1})}$$

$$\text{coefficient} = \frac{\text{Ideal\_V1}}{(\text{Reference\_V1\_ADCValue} - \text{Offset})}$$

3. md_adc_get_data(SSx,offset,coefficient,*calibration) : 此 function 需要带入 SSx(采样 Sequence)以及 md_adc_calibration_data(vdd,vref,*offset,*coefficient)所得之 Offset 及 Coefficient，接着将 ADC 所得到的值做运算得到校正后结果。

Calibration value = (adc_value – Offset) * coefficient

◆ md_adc.h

增加 adcinit struct：可在 main.c 里给定 adc 所需的初始设定。

◆ reg_fc.h

缓存器 struct 增加三个 ADCTRIM_VDD5V_VREF5V、ADCTRIM_VDD3V3_VREF3V3、ADCTRIM_VDD5V_VREF2V5，分别用来存 3 个 case ADC 参考点的值

bit[31:24]: 存 0xA5，表示有值
bit[23:12]: 存 2V 参考值
bit[11:0]: 存 1V 参考值

◆ Project 名称：“ADC_Calibration_From_Flash”

设定及使用流程

Step1: md_adc_init(ADCx,*ADC_InitStruct,*channel) 带入 ADC 初始设定及取样 channel
Step2: md_adc_calibration_data(vdd,vref,*offset,*coefficient) 计算 Offset 与 Coefficient 值
vdd: 外部输入电压
vref: 参考电压

EX:
输入外部电压为 5.0V,参考电压为 2.5V 时 function 会选择 FC→ADCTRIM_VDD5V_VREF2V5

所存的值当作参考值,经过 Offset 公式计算出 offset 及 coefficient 的值。

Offset= (2*1587.19-1*3242.12)/(2-1) = -67.44

Coefficient 公式可求出系数 ：1638.4/(1587.19-(-67.44)) = 0.99

Step3: md_adc_get_data(SSx,offset,coefficient,*calibration)

SSx: 准备取样的序列

offset: Step2 所计算出的 offset

coefficient: Step2 所计算出的 coefficient

Ex:

假设 1.5V 所量测到的值为 2417.56,经过修正后

Calibration value = (2417.56-(-67.44))*0.99 = 2460.15