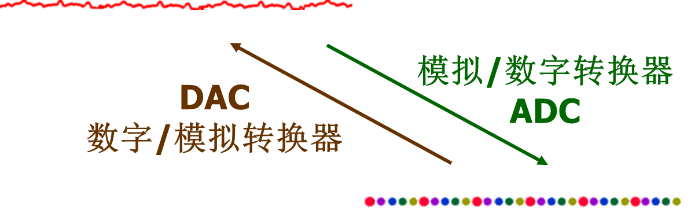
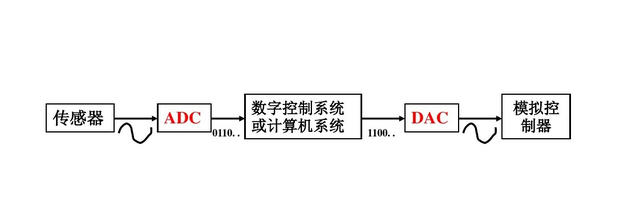
# DAC和ADC

有了异步串行通信，就可以通过虚拟串口，轻松实现嵌入式系统和PC、服务器的数据交互。可以通过微控制器采集传感器的数据，发送给PC进行记录、显示和处理。

我们已经学习过使用IIC总线读取加速度传感器的数据。除了加速度传感器外，还有很多传感器的信号是以模拟信号呈现的。

模拟信号的特点是时间连续、幅值连续。由于现在的计算机几乎已经占领了各个领域，因此这些领域对信号的处理都要转换成计算机能看得懂的信号，然后再交给计算机来处理，这些计算机能看懂的信号就是数字信号。那么将模拟信号转换成数字信号需要一种器件，它叫做模数转换器(ADC)。以此类推，既然ADC是将模拟量转换成数字量，那么也还有一种器件能把数字量转换成模拟量，它叫数模转换器(DAC)。

****



# 数字/模拟转换器DAC

DAC是将数字量转换成模拟量输出的设备。

DAC将输入的二进制数字量转换成模拟量，以电压或电流的形式输出。

DAC实质上是一个译码器（解码器），提供了数字量到模拟量的映射功能。一般常用的线性DAC，其输出模拟电压uO和输入数字量Dn之间成正比关系：

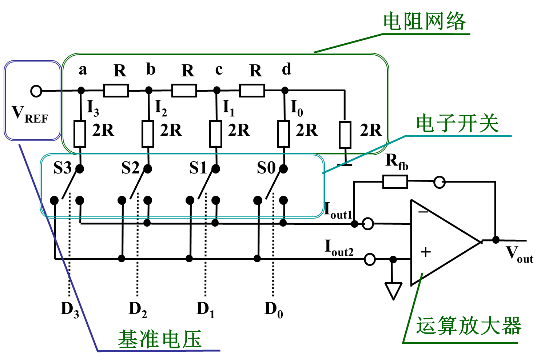
uO＝Dn ◊UREF

UREF为参考电压。



**D/A转换的基本原理**

DAC一般由缓冲寄存器、模拟电子开关、参考电压、解码网络和求和电路等组成。一个四位DAC的原理示意图如图所示。



**缓冲寄存器**

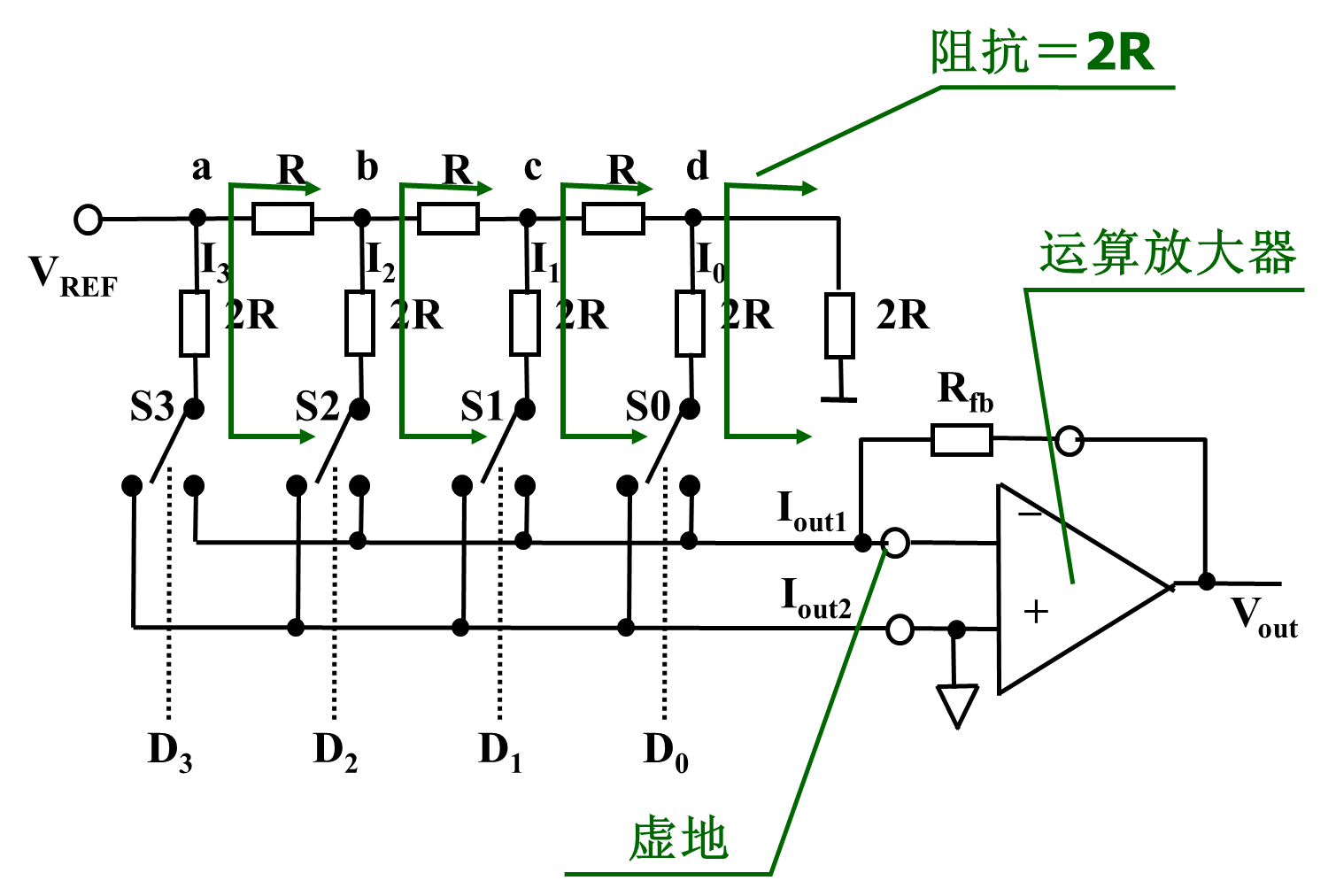
Dn

其中数字输入Dn写入到缓冲寄存器后，由寄存器的每一个位，控制一个模拟电子开关。

本DAC的解码网络由倒T型电阻网络组成，是DAC转换的关键。由运放组成求和电路。

电阻网络设计的十分巧妙，由电阻值为R和2R的电阻组成，因此这种结构也被称作R-2R结构。

在a、b、c、d每一个点，向右看，阻抗都是2R，



而运放的同相输入端接地，根据运放虚短的特点，无论S0、S1、S2、S3四个模拟电子开关向哪个方向拨，开关后的电位都是零。a、b、c、d四个点的电压为：

Va＝VREF

Vb＝VREF/2

Vc＝VREF/4

Vd＝VREF/8

那么，电流 I0、I1、I2、I3可以计算为：

I0＝Vd/2R＝VREF/（8×2R）

I1＝Vc/2R＝VREF/（4×2R）

I2＝Vb/2R＝VREF/（2×2R）

I3＝Va/2R＝VREF/（1×2R）

这时，输入量Dn的每一位，控制一个电子开关。

Dn的第0位Dn(0)控制S0，如果Dn(0)为1，那么S0向右拨，I0流向运放的反向输入端，如果Dn(0)为0，那么S0向左拨，I0流向运放的同向输入端，即流入地。

Dn的第1位Dn(1)控制S1，如果Dn(1)为1，那么S1向右拨，I1流向运放的反向输入端，如果Dn(1)为0，那么S1向左拨，I1流向运放的同向输入端，即流入地。

Dn的第2位Dn(2)控制S2，如果Dn(2)为1，那么S2向右拨，I2流向运放的反向输入端，如果Dn(2)为0，那么S2向左拨，I2流向运放的同向输入端，即流入地。

Dn的第3位Dn(3)控制S3，如果Dn(3)为1，那么S3向右拨，I3流向运放的反向输入端，如果Dn(3)为0，那么S3向左拨，I3流向运放的同向输入端，即流入地。

那么，流向运放反向输入端的电流为：

Iout1＝Dn(0)◊I0＋Dn(1)◊I1＋Dn(2)◊I2＋Dn(3)◊I3

＝VREF/2R×（Dn(0)◊1/8＋Dn(1)◊ 1/4＋Dn(2)◊1/2＋Dn(3)◊1）

那么运放的输出电压为：

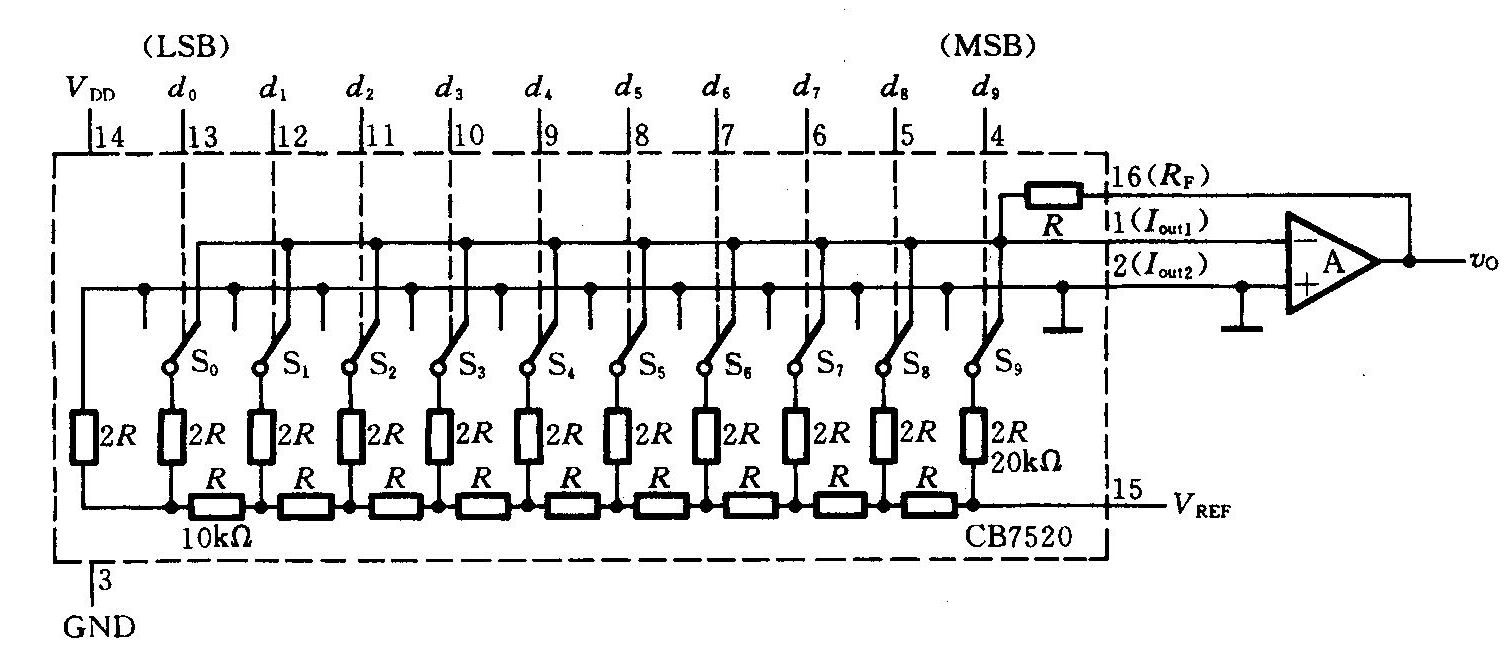
Vout＝－Iout1×Rfb

如果Rfb =R，则

Vout＝－VREF×[（Dn(0)◊20＋Dn(1)◊21＋Dn(2)◊22＋Dn(3)◊23）/24]

可见，DAC将输入的每一位二进制代码，按其权值大小转换成相应的模拟量，然后将代表各位的模拟量相加，则所得的总模拟量就与数字量成正比，这样便实现了从数字量到模拟量的转换。

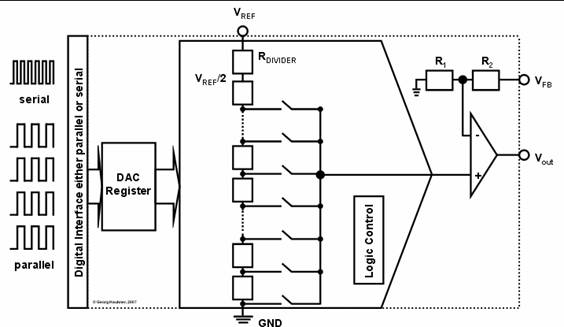
**应用示例(10位DAC CB7520)：**



DAC——CB7520电路原理图

**其他结构的DAC：**

倒T型网络DAC也被称作**R2R 架构**DAC。除此之外，常用的还有电阻串(R-String)结构的DAC。



顾名思义，电阻串架构就是一个以串联形式放置的一串电阻，以构建一个电阻串。从理论上来说，可能会需要 256 个电阻才能构建一款 8 位 DAC (28 = 256)，因此，设计人员推出了其它更小的电路设计方案，如可降低电阻串上所需电阻数量以及接触点的内插式放大器，从而实现了功耗更低且更节省空间的设计。

**DAC的主要技术指标**

1. **转换精度**

输出模拟电压的实际值与理想值之差，也称最大静态转换误差。

1. **分辨率**

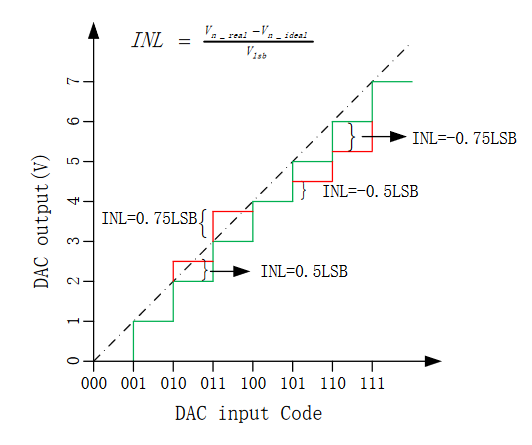
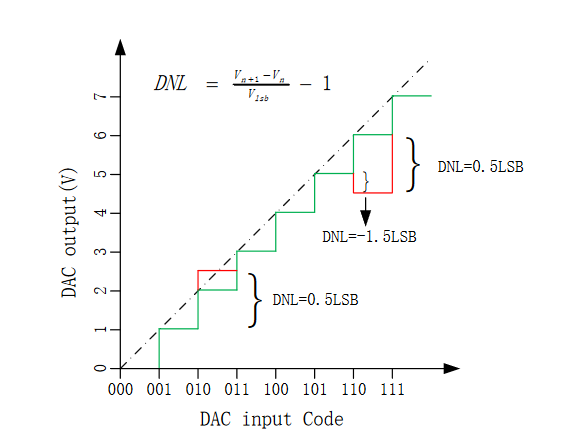
DAC模拟输出电压可能被分离的等级数。

输入数字量位数越多，分辨率越高。所以，在实际应用中，常用字量的位数表示DAC的分辨率，如CB7520为10位DAC

此外，也可用DAC的最小输出电压与最大输出电压之比来表示分辨率，N位DAC的分辨率可表示为 1/（2n-1）。

1. **线性度**

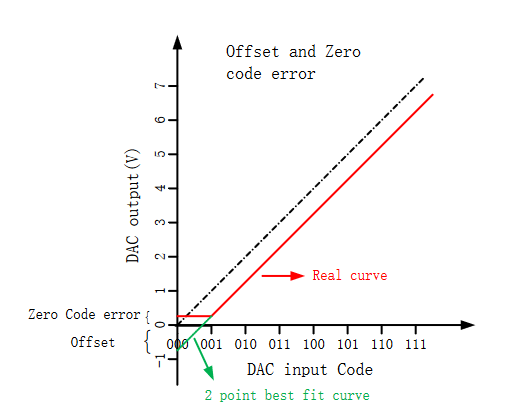
通常用非线性误差的大小表示DAC的线性度。

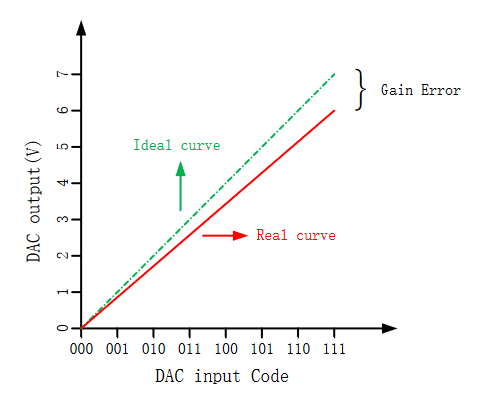


**DNL**  是微分线性度，指相邻两个输出电平的差相对于理想值（1LSB）的偏差。 datasheet 中的 DNL 代表所有台阶中最大的偏差值。 由上图可以看到， 如果出现 DNL<-1LSB 的现象，则 DAC 的输出肯定是非单调的， 也就是说数字编码增加 1，输出不增加反而会下降。 这一点在很多闭环系统应用中是不能接受的，如果 DAC 出现非单调的情况，则控制环路无法收敛。 这时一般会选择 DNL < ±1LSB 的器件。

**INL** 是积分线性度，指实际的输出相对理想 DAC 的输出之间的差异，所以也叫 relative accuracy，用满量程的百分比或者 LSB 来表示。理论上，某个编码对应输出的 INL 就是从第一个编码到这个编码所有的 DNL 的积分， 也印证了“积分”这个名称的含义。Datasheet 中的 INL（或者 relative accuracy）代表所有输出值最大的 INL。 这个指标用来衡量 DAC 输出的准确度如何，应用比较广。 特别是在开环应用中，应当关注 INL 的指标。

1. **偏移**



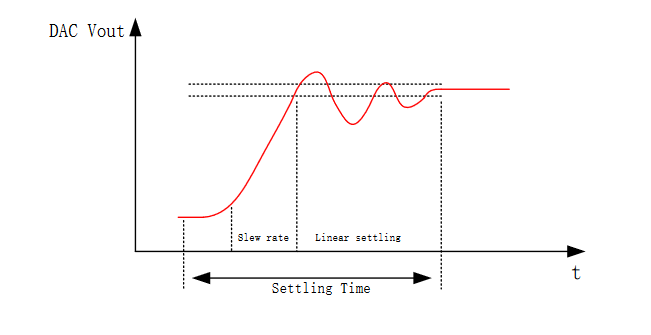


一个无限分辨率的理想 DAC 输出特性应该是通过原点的一条直线，y=x (这里我们把 DAC 增益相对理想值归一化成 1), 但实际的 DAC 输出特性，用靠近首尾两端的两点拟合一条直线， 特性一般是 y=ax+b。其中，a 代表 DAC 实际的输出增益， 即 gain。其相对理想增益的偏差，即 gain error。b 代表这条直线整体相对原点向上或者向下偏移的幅度，即 offset error.

但实际 DAC 在 code 为 0 附近，输出电压也很低时，由于内部电路接近饱和（特别是带输出 buffer 的 DAC），会出现一定的非线性。所以 DAC 会有一个额外的参数来标定 code 为 0 时输出的偏差，叫 zero code error. 另外 gain，offset 在不同温度下也会产生变化，即 gain shift, offset error shift。

1. **转换速度**

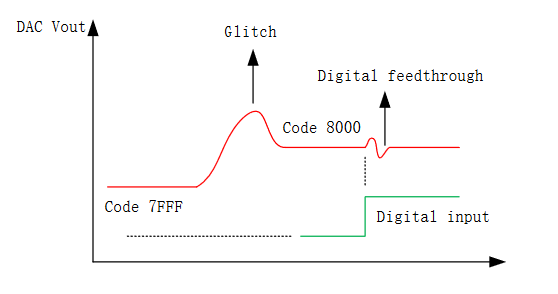
建立时间（Settling time）



输出从 0 到满摆幅变化（或者特定的两个差异较大的值）的总时间，称为 settling time。输出主要经历两个阶段，一是 slew rate， 二是 linear settling。slew rate 反映了输出大摆幅下的极限驱动能力，一般决定了输出 10%~90%变化的时间，。而 linear settling 则主要取决于输出节点的 RC 常数或者输出 buffer 的带宽。

Settling time 是用户考虑精密 DAC 速度的重点参数。

1. **毛刺（Glitch）**

****

**Glitch** 主要与 DAC 核心部分的开关有关。 当内部开关从一个点切换到另一个点时， 会受到寄生电荷以及开关切换不能理想同步的影响，从而造成输出跳动。 跳动的幅度和时间都是我们关注的对象，所以 Glitch 用 nV\*S 这个二者相乘的单位来表示其能量大小。从其产生原理可见， glitch 与具体切换的开关位置有关。Code的高位 MSB 变化时一般会产生较大的 glitch，所以 datasheet 中普遍定义 major carry 处的 glitch。 Glitch也和结构有关， R-string 的 glitch 一般比 R2R 结构的 glitch 小。

Digital feedthrough 则代表了模拟输出与数字输入的隔离程度。 即使 DAC 没有被选中进行通信，总线上的数字 IO 信号或时钟跳动通过内部信号通路或者电源地的耦合也会造成 DAC 输出的跳动，即为 digital feedthrough。 良好的设计可以保证这个值很小。

1. **温度系数**

在输入不变的情况下，输出模拟电压随温度变化产生的变化量。一般用满刻度输出条件下温度每升高1℃，输出电压变化的百分数作为温度系数。

# 模拟/数字转换器ADC

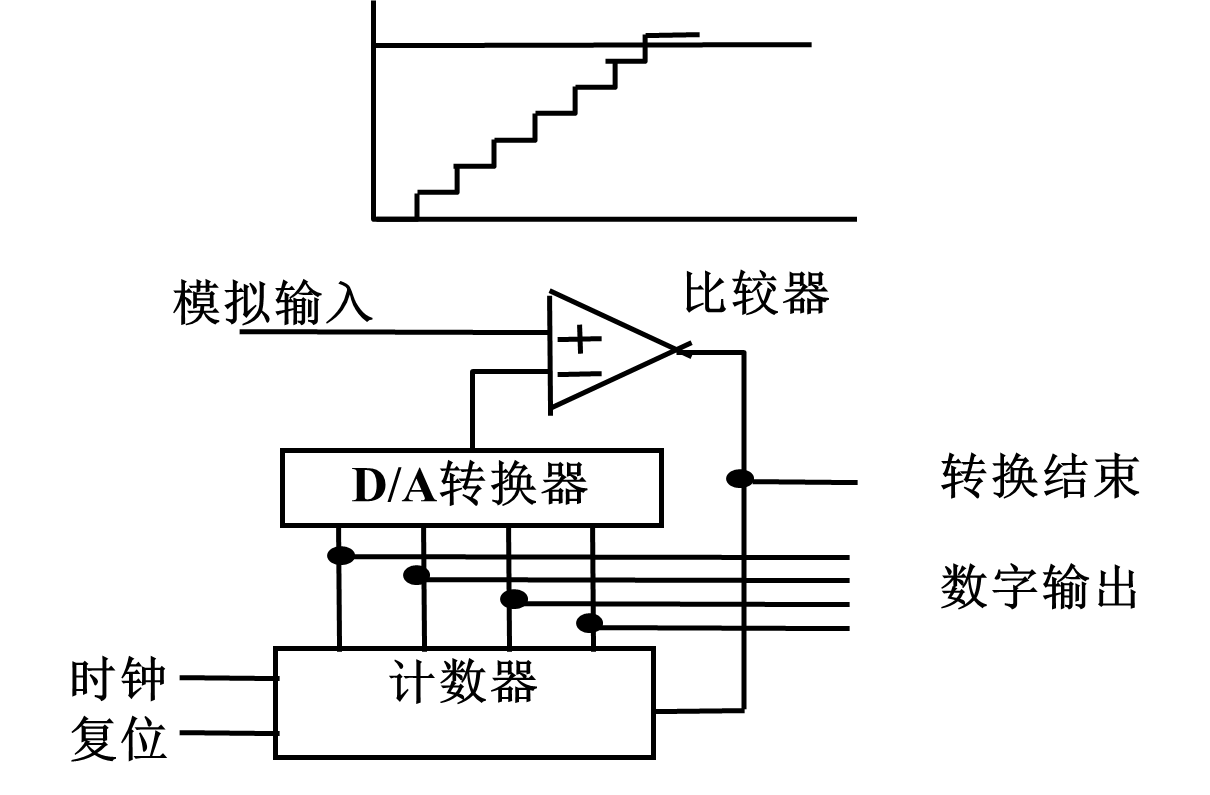
要实现将连续变化的模拟量变为离散的数字量，有多种转换技术，各有特点，分别应用于不同的场合。

以下5种是比较常用的AD转换技术：

* 计数器式
* 逐次逼近式
* 双积分式
* 并行式
* Delta-sigma

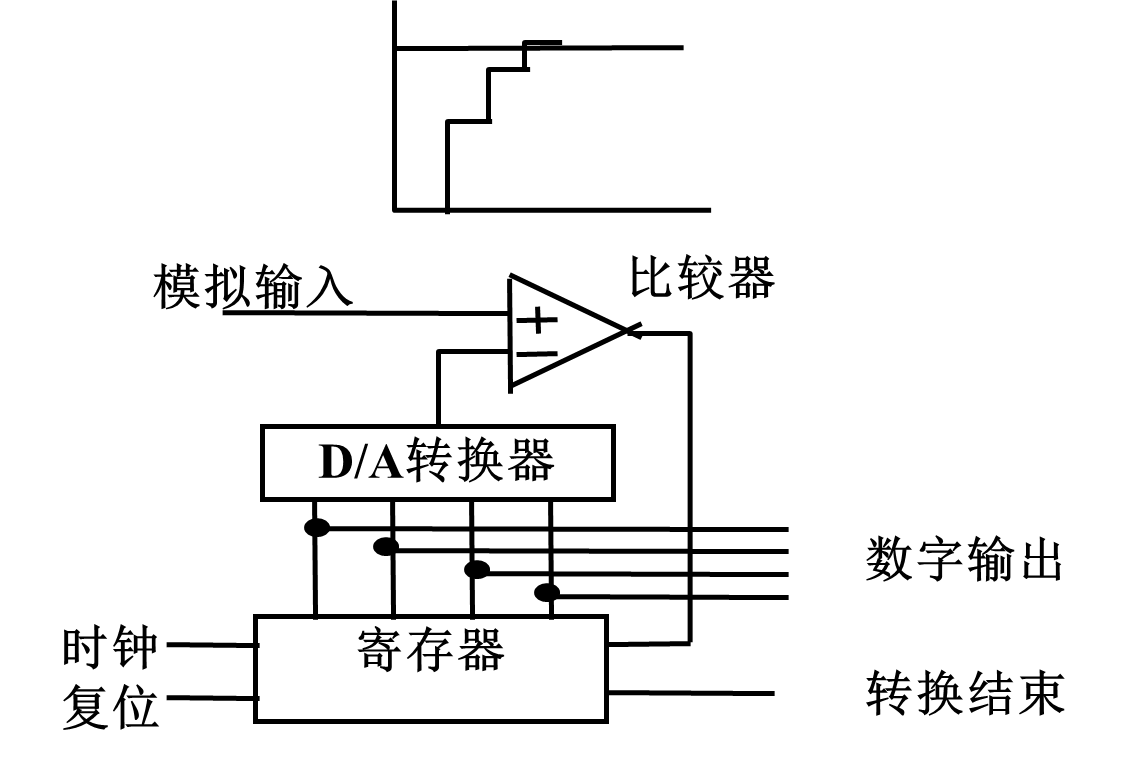
1. **计数器式**

以最低位为增减量单位的逐步计数法

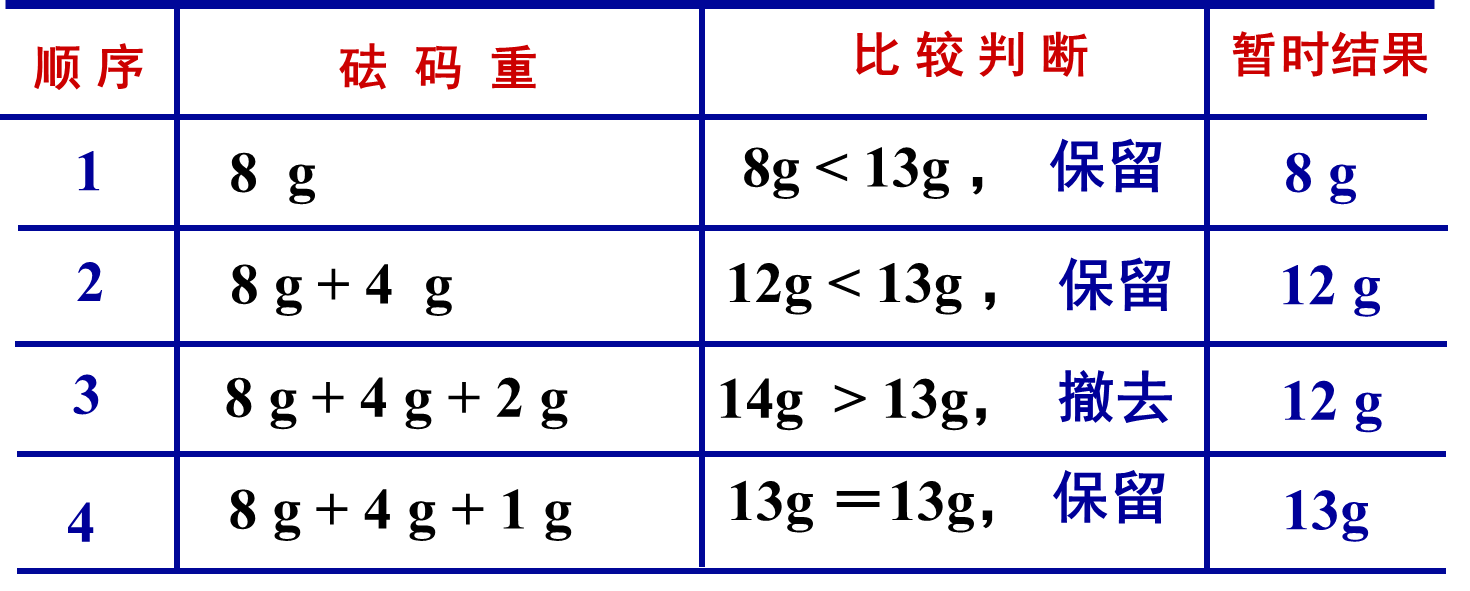


1. **逐次逼近式**

从最高位开始的逐位试探法，类似于查找算法中的二分法。



其工作原理可用天平秤重过程作比喻来说明。若有四个砝码共重15克，每个重量分别为8、4、2、1克。设待秤重量Wx = 13克，可以用下表步骤来秤量：



例：参考电压UREF= 8V，输入电压UI = 5.52V



由于逐次逼近式ADC转换过程中，需要多次比较才能得到最终结果。在比较的过程中，输入电压Ui需要保持不变，因此逐次逼近式ADC的模拟-数字转换需经过四个步骤：采样、保持、量化、编码，一般前两步由采样-保持电路完成，量化和编码由ADC完成。

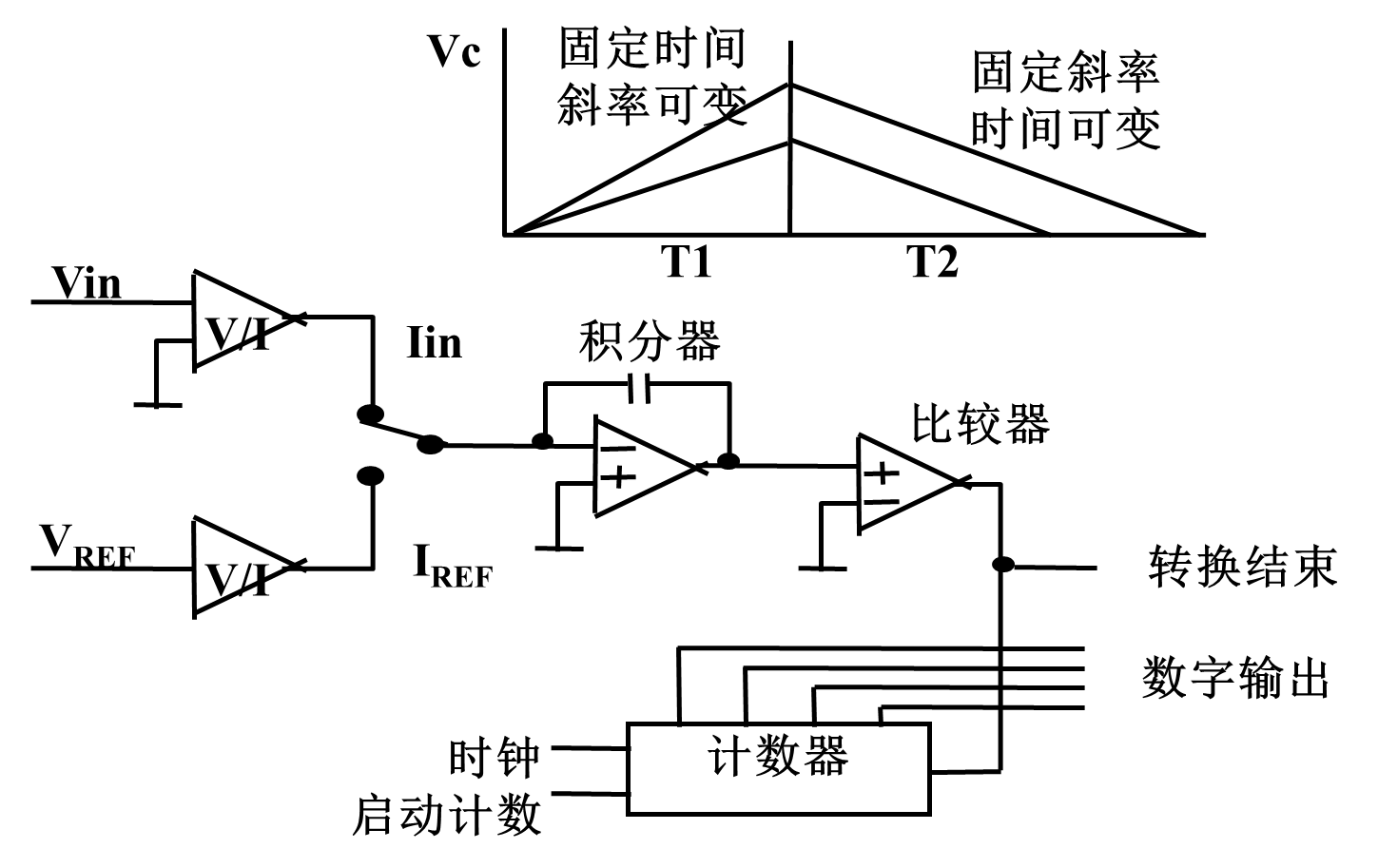
常用的采样保持电路：



1. **双积分式**

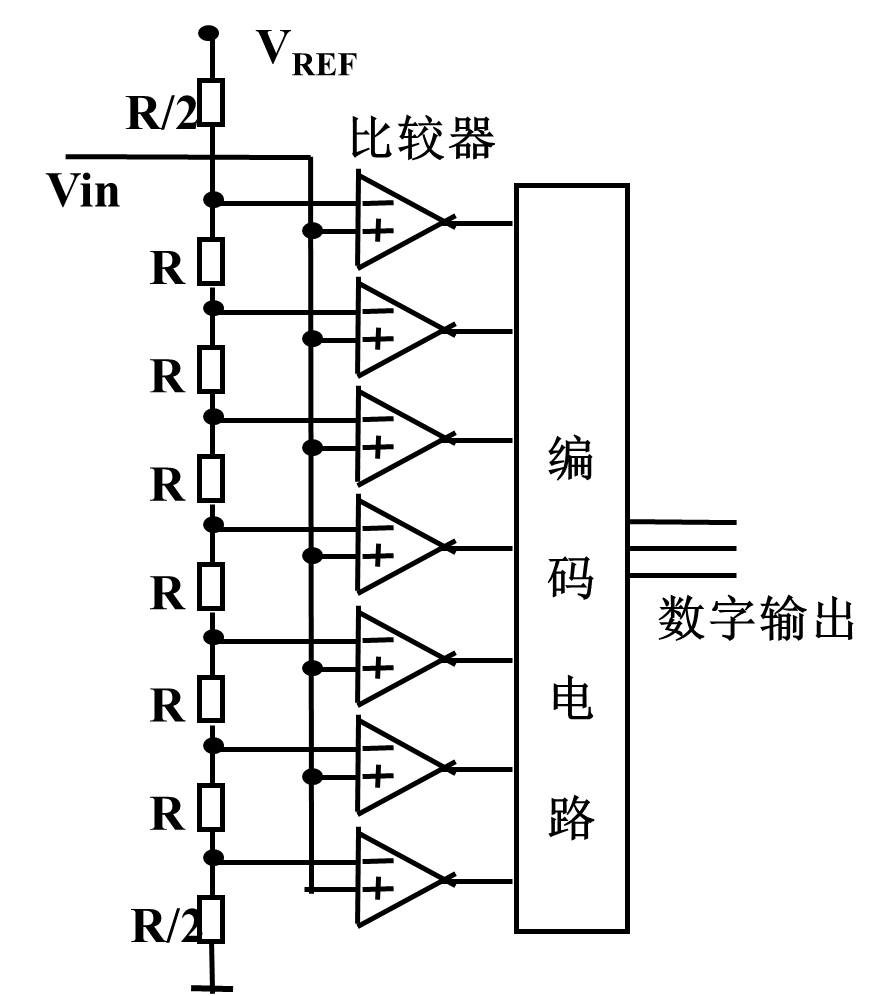
分两个积分阶段，先固定积分时间T1，对输入信号Vin积分。然后对参考信号进行反向积分，计算积分到零的时间T2。就可以根据T2与T1的关系以及VREF的大小，算出Vin。

实质是电压/时间变换



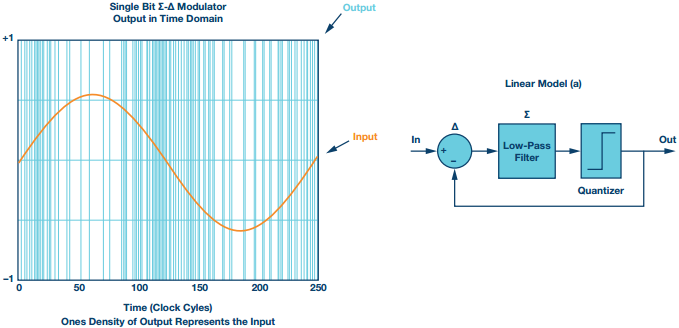
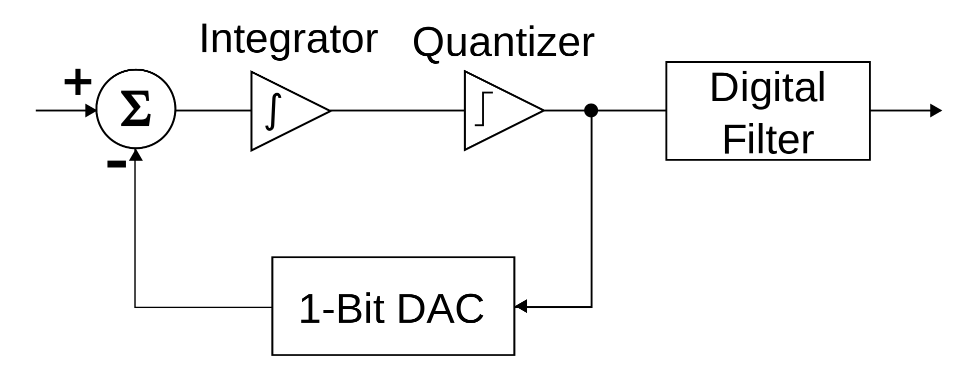
1. **并行式**

用电阻分压实现多个参考电压，然后使输入信号与每个参考电压比较。



1. **Delta-sigma型**

Delta-Sigma（ΔΣ）调制（或称Sigma-Delta（ΣΔ）调制、SDM，中文译作积分-微分调制）是一种数字模拟互相转换的方法，它是把高比特清晰度低频率信号用脉冲密度调制编码为低比特清晰度高频率信号的一种方法。



**A/D 转换器的主要技术指标**

1. **分辨率**

以输出二进制数的位数表示分辨率。位数越多，误差越小，转换精度越高。

1. **转换速度**

完成一次A/D转换所需要的时间，即从它接到转换控制信号起，到输出端得到稳定的数字量输出所需要的时间。

1. **精度**

精度是反映转换器的实际输出接近理想输出的精确程度的物理量。实际转换值和理想特性之间的最大偏差。

1. **量化误差 (Quantizing Error)**

由于AD的有限分辩率而引起的误差，即有限分辩率AD的阶梯状转移特性曲线与无限分辩率AD（理想AD）的转移特性曲线（直线）之间的最大偏差。通常是1 个或半个最小数字量的模拟变化量，表示为1LSB、1/2LSB。

1. **偏移误差(Offset Error)**

输入信号为零时输出信号的值。

1. **满刻度误差(Full Scale Error)**

满度输出时对应的输入信号与理想输入信号值之差。

1. **线性度(Linearity)**

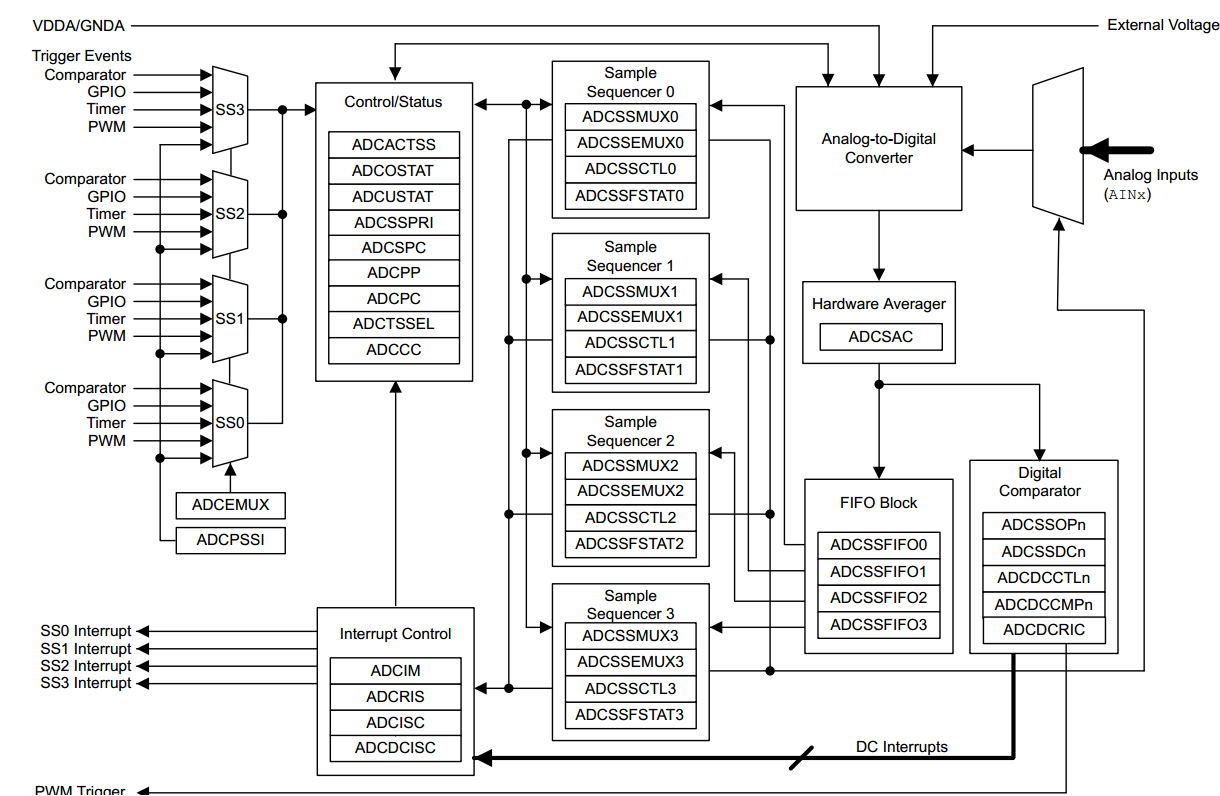
实际转换器的转移函数与理想直线的最大偏移。

# TM4C1294的ADC模块

TM4C1294微控制器有两个12位逐次逼近型ADC，ADC0和ADC1



* 两个ADC共享20路模拟信号和一个片内温度传感器信号
* 每个ADC的最高采样频率为2Msps，采样一次需要至少16个ADC时钟，4个保持周期+12个采样周期
* 以采样序列为单位进行采样，有四个可编程采样序列，每个序列的转换结果都有FIFO保存
* 采样完成后可以产生中断



**TM4C1294的ADC模块的使用方法：**

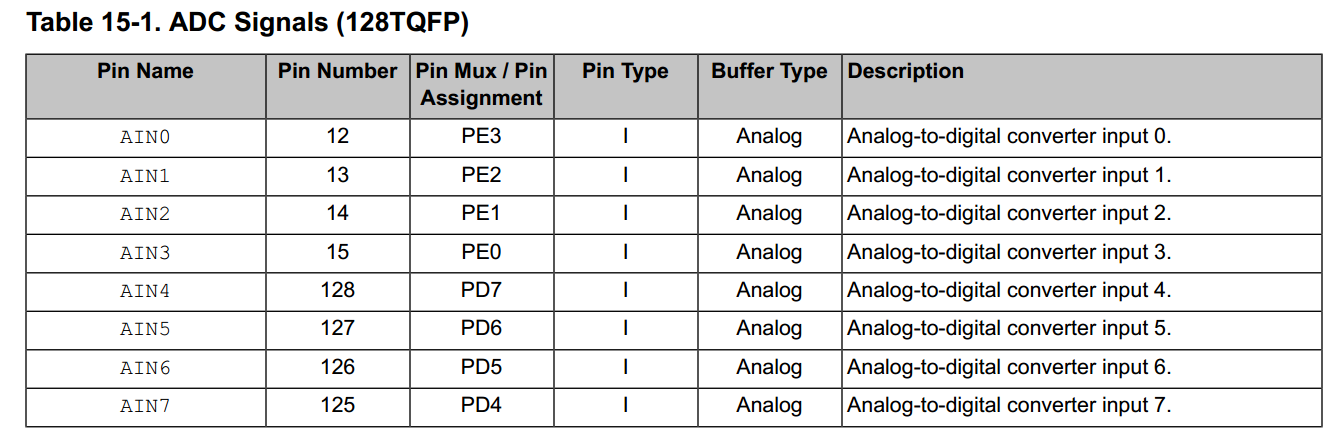
1. **在系统设置模块中，使能ADC模块的时钟：**

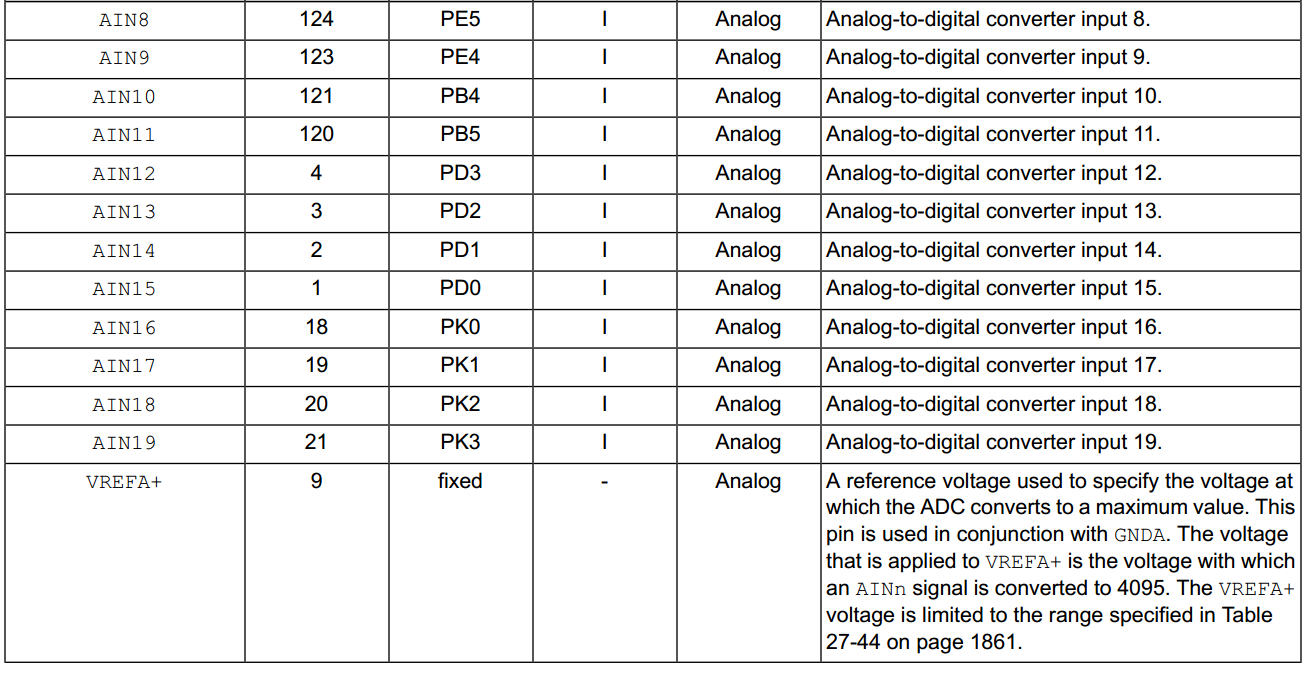
SysCtlPeripheralEnable(SYSCTL\_PERIPH\_ADC0);

1. **使能使用到的ADC引脚所在的GPIO模块的时钟：**

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOE);

ADC输入引脚与GPIO引脚的对应关系为：

****

****

1. **将GPIO的引脚配置为模拟输入**

GPIOPinTypeADC(GPIO\_PORTE\_BASE, GPIO\_PIN\_0);

void

GPIOPinTypeADC(uint32\_t ui32Port, uint8\_t ui8Pins)

{

ASSERT(\_GPIOBaseValid(ui32Port));

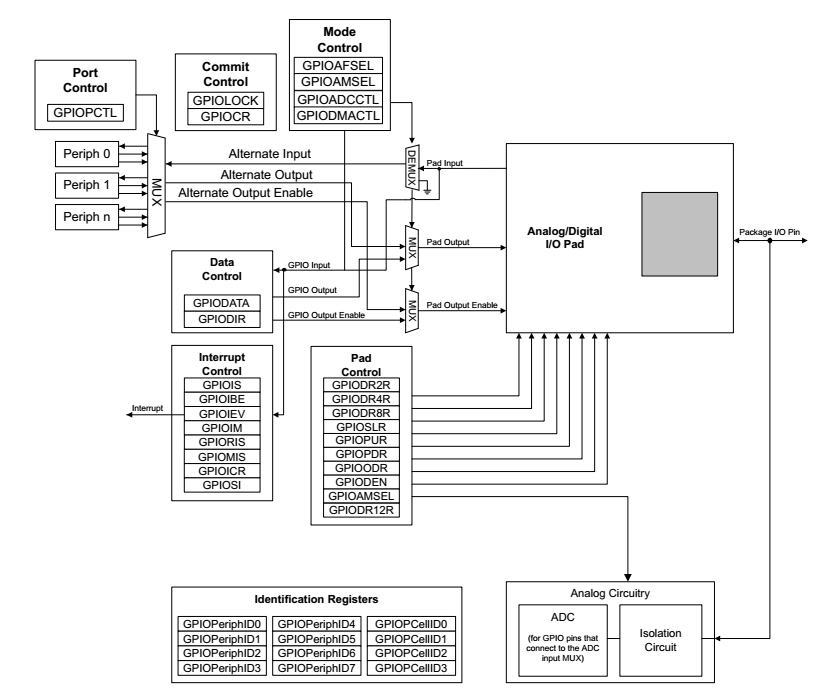
GPIODirModeSet(ui32Port, ui8Pins, GPIO\_DIR\_MODE\_IN);

GPIOPadConfigSet(ui32Port, ui8Pins, GPIO\_STRENGTH\_2MA,

GPIO\_PIN\_TYPE\_ANALOG);

}

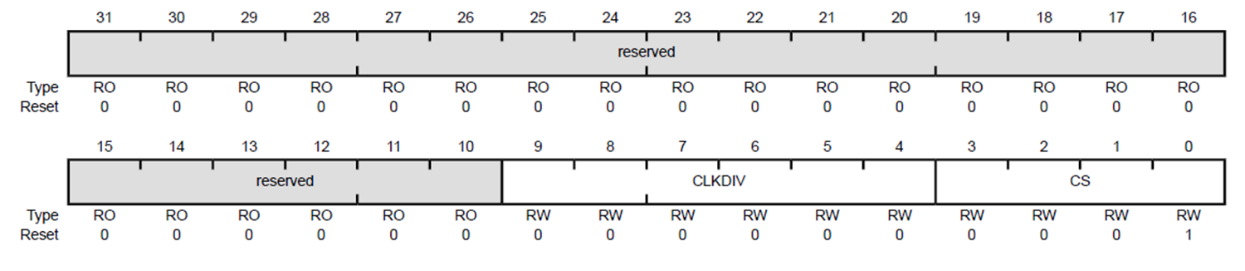
可见，GPIOPinTypeADC函数是通过把GPIOPadConfigSet函数的第三个参数设置为GPIO\_PIN\_TYPE\_ANALOG，把引脚设置为模拟输入。



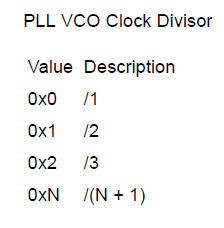
1. **设置ADC模块的采样速率**

有两个寄存器影响ADC的采样速率：

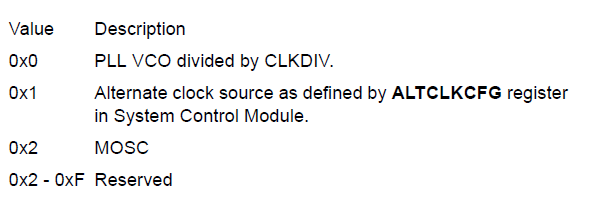
ADC Clock Configuration (**ADCCC**)寄存器用于配置ADC模块的时钟，决定了ADC的采样速率：



CLKDIV域：PLL VCO的分频系数



CS域：时钟源选择



PLL VCO的频率在系统时钟初始化时确定，对于TM4C1294微控制器，VCO只有两个选项：480MHz或者320MHz。

ui32SysClock = SysCtlClockFreqSet((SYSCTL\_XTAL\_25MHZ |

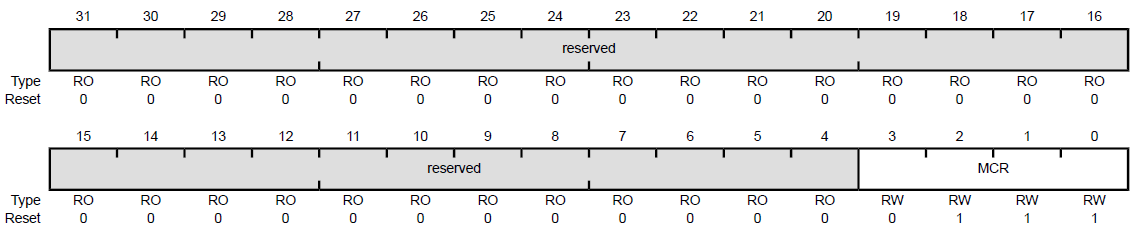
SYSCTL\_OSC\_MAIN |

SYSCTL\_USE\_PLL |

SYSCTL\_CFG\_VCO\_480), 120000000);

无论选择何种时钟源，ADC的时钟频率最高不可超过32MHz。

ADC Peripheral Configuration (**ADCPC**)寄存器，也与采样速率有关。用于在两次采样之间，插入一定的时间，降低采样速率。



MCR域：

0x07：全速模式，两次采样之间不插入什么时间

0x06：半速模式，两次采样之间插入16个ADC时钟周期

0x03: 1/4速模式，两次采样之间插入48个ADC时钟周期

0x01：1/8速模式，两次采样之间插入112个ADC时钟周期

使用TivaWare提供的ADCClockConfigSet函数，设置ADC的采样速率，该函数的定义为：

void

ADCClockConfigSet(uint32\_t ui32Base, uint32\_t ui32Config,

uint32\_t ui32ClockDiv)

{

ASSERT((ui32Base == ADC0\_BASE) || (ui32Base == ADC1\_BASE));

ASSERT((ui32ClockDiv - 1) <= (ADC\_CC\_CLKDIV\_M >> ADC\_CC\_CLKDIV\_S));

ASSERT((ui32Config & ADC\_CLOCK\_RATE\_FULL) != 0);

HWREG(ui32Base + ADC\_O\_PC) = (ui32Config >> 4) & ADC\_PC\_SR\_M;

HWREG(ui32Base + ADC\_O\_CC) = (ui32Config & ADC\_CC\_CS\_M) |

(((ui32ClockDiv - 1) << ADC\_CC\_CLKDIV\_S)) ;

}

ADCClockConfigSet函数的第一个参数为要操作的ADC的基地址，由于两个ADC共享ADC时钟，所以第一个参数只能是ADC0\_BASE。

ADCClockConfigSet函数的第二个参数用于设置ADCCC寄存器的CS域和ADCPC寄存器的MCR域，

ADCCC寄存器的CS域可选的参数为：

**ADC\_CLOCK\_SRC\_PLL** - The main PLL output.

**ADC\_CLOCK\_SRC\_PIOSC -** The internal PIOSC at 16 MHz.

**ADC\_CLOCK\_SRC\_ALTCLK** - The output of the ALTCLK in the system ADC\_CLOCK\_SRC\_MOSC - The external MOSC .

ADCPC寄存器的MCR域可选的参数为：

**ADC\_CLOCK\_RATE\_FULL** - All samples.

**ADC\_CLOCK\_RATE\_HALF** - Every other sample.

**ADC\_CLOCK\_RATE\_QUARTER** - Every fourth sample.

**ADC\_CLOCK\_RATE\_EIGHTH** - Every either sample.

以上两组参数的定义为：

#define ADC\_CLOCK\_RATE\_FULL 0x00000070

#define ADC\_CLOCK\_RATE\_HALF 0x00000050

#define ADC\_CLOCK\_RATE\_FOURTH 0x00000030

#define ADC\_CLOCK\_RATE\_EIGHTH 0x00000010

#define ADC\_CLOCK\_SRC\_PLL 0x00000000

#define ADC\_CLOCK\_SRC\_PIOSC 0x00000001

#define ADC\_CLOCK\_SRC\_ALTCLK 0x00000001

#define ADC\_CLOCK\_SRC\_MOSC 0x00000002

两组参数中各选一个，按位或后，形成ADCClockConfigSet函数的第二个输入参数。

ADCClockConfigSet函数的第三个参数为ADCCC寄存器的CLKDIV域。

如果系统时钟初始化函数为：

ui32SysClock = SysCtlClockFreqSet((SYSCTL\_XTAL\_25MHZ |

SYSCTL\_OSC\_MAIN |

SYSCTL\_USE\_PLL |

SYSCTL\_CFG\_VCO\_480), 120000000);

ADC时钟初始化函数为：

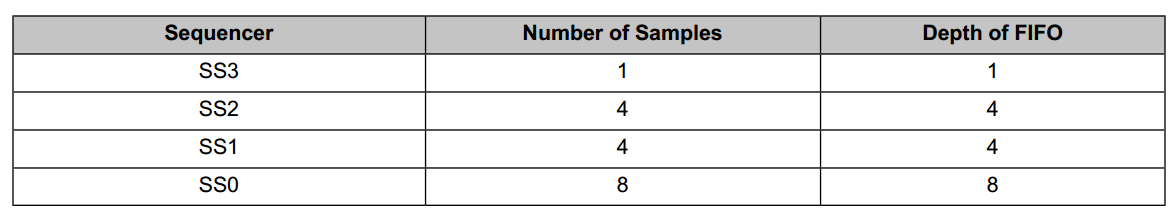
ADCClockConfigSet(ADC0\_BASE, ADC\_CLOCK\_SRC\_PLL | ADC\_CLOCK\_RATE\_FULL, 24);

那么ADC时钟为：480MHz/24=20MHz。最高采样速率为20MHz/16=1.25Msps。（ADC转换过程默认为4个保持周期+12个转换周期）

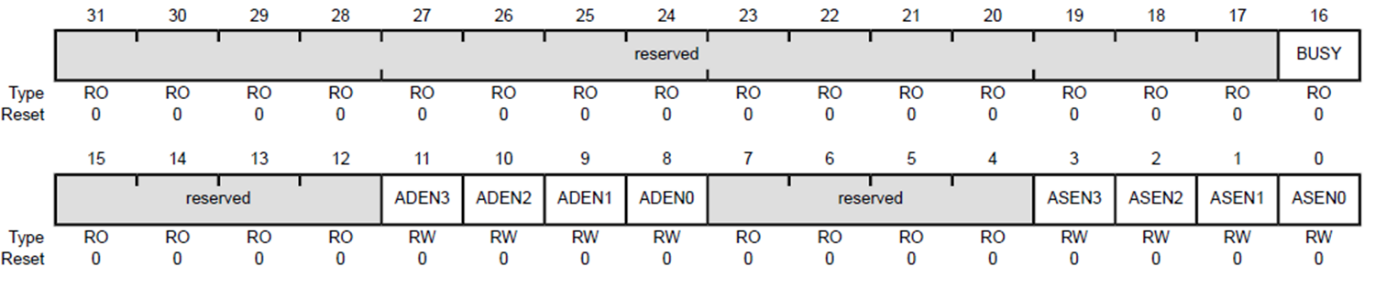
1. **配置采样序列**

5.1 选择并设置ADC的采样序列。

ADC模块有四个采样序列，每个采样序列中，能转换的最大采样数是不同的，如下图所示。



ADC Active Sample Sequencer (**ADCACTSS**)寄存器用于控制四个采样序列的使能：



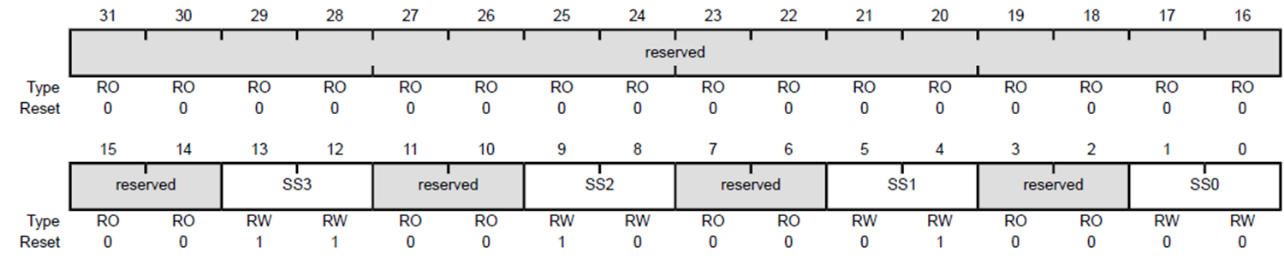
ASEN0：SS0的使能位：

ASEN1：SS1的使能位：

ASEN2：SS2的使能位：

ASEN3：SS3的使能位：

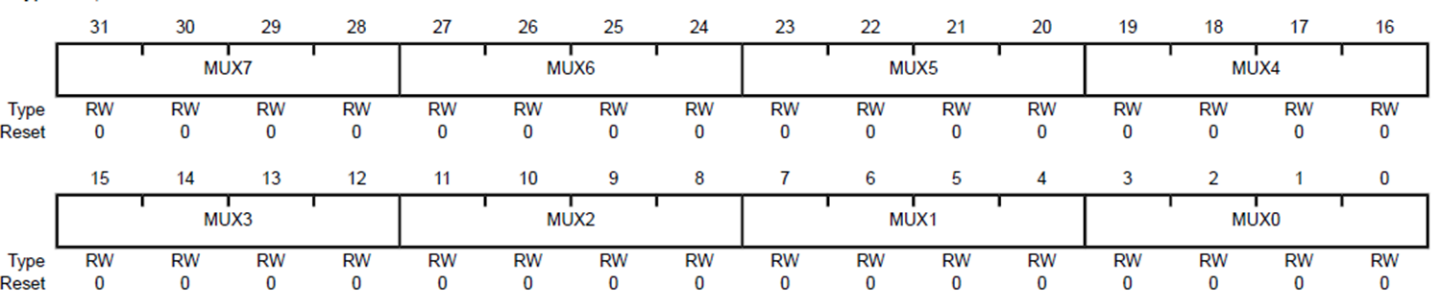
每个ADC模块有四个采样序列，他们的优先级由ADC Sample Sequencer Priority (**ADCSSPRI**) ADC采样序列优先级寄存器决定：



域SS0到SS3分别配置采样序列SS0到SS3的优先级，数值越低，优先级越高。默认情况下，SS0的优先级最高。

**5.2 选择采样序列中需要转换的通道，并设置采样属性。**

如果选用采样序列SS0，那么在这个序列中，需要转换哪些通道，由ADC Sample Sequence Input Multiplexer Select 0（**ADCSSMUX0**）ADC采样序列复选寄存器决定。

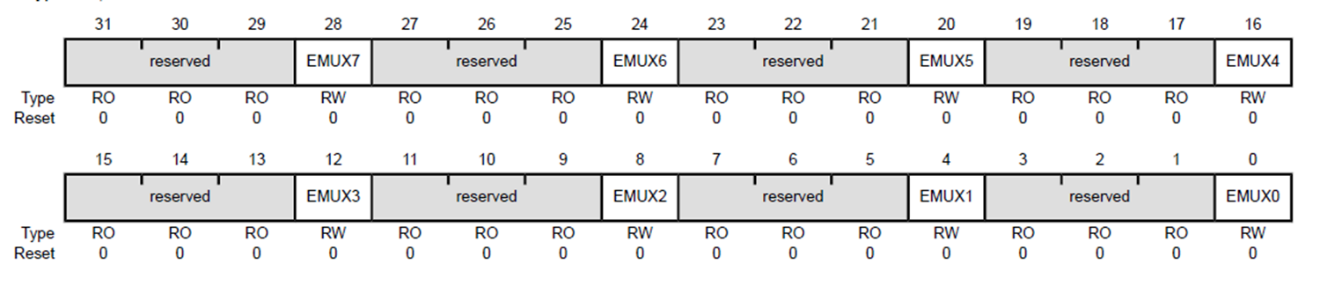


这个寄存器，每四位控制一个AD转换的采样通道，值可以使0-15。假设MUXn的值为n

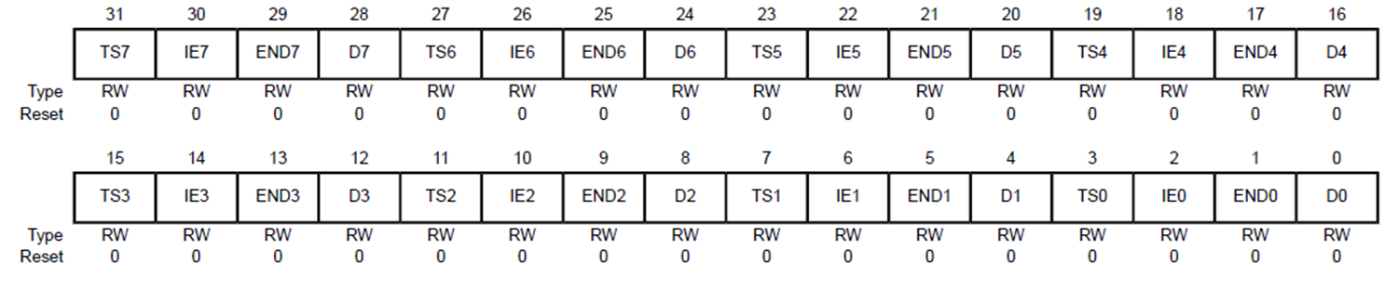
如果**ADCSSEMUX0**.EMUXn为0：MUXn表示相应的AINn通道

如果**ADCSSEMUX0**.EMUXn为1：MUXn表示相应的AIN(n+16)通道

**ADCSSEMUX0**寄存器的结构如图所示：



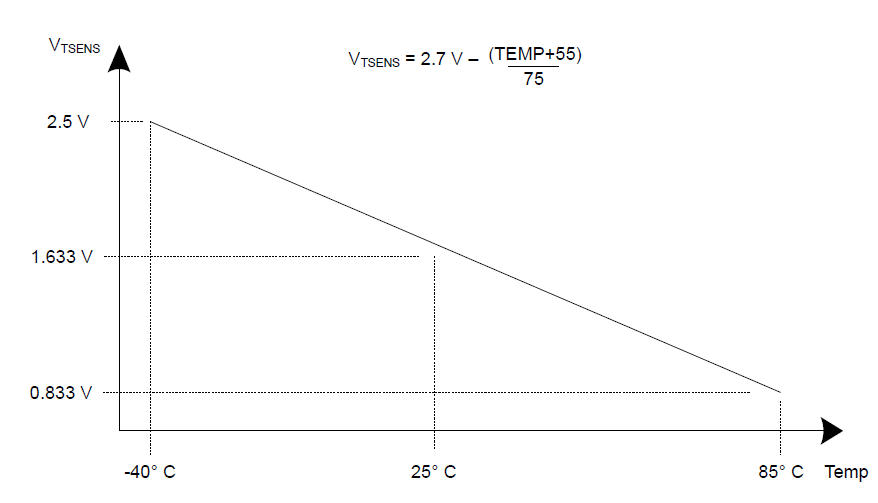
在一个采样序列中，一共要进行多少次采样，由ADC Sample Sequence Control 0 (ADCSSCTL0) ADC采样序列控制寄存器决定。



ADCSSCTL0寄存器中，每4位为1组，决定了一个采样通道的属性。一组中的4个位有不用的含义：

* TSn:表示第n次采样为片内**温度采样**
* IEn: 表示此次采样完成后**触发序列中断**
* ENDn: 本次采样为序列中的**最后一次**采样
* Dn: 本次采样为**差分采样**，采样通道为2i和2i+1

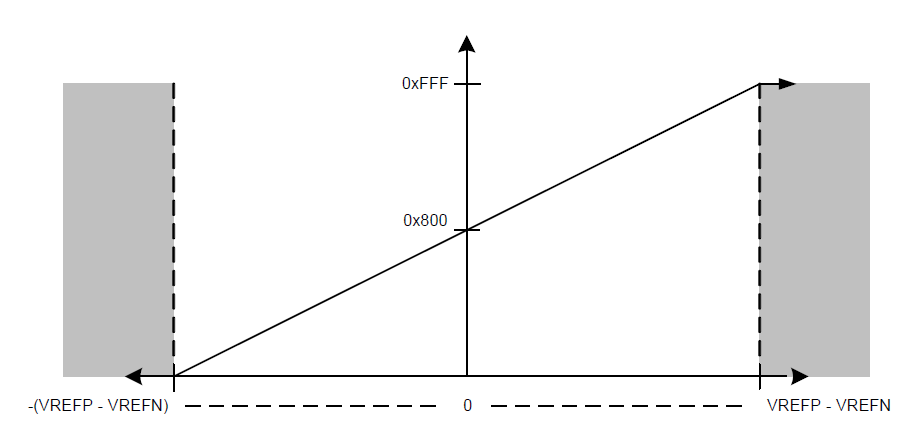
**温度采样：**



TEMP = 147.5 - ((75 \* (VREFP - VREFN) × ADCCODE) / 4096)

VREFP默认3.3V，VREFN默认0V。

**差分采样：**

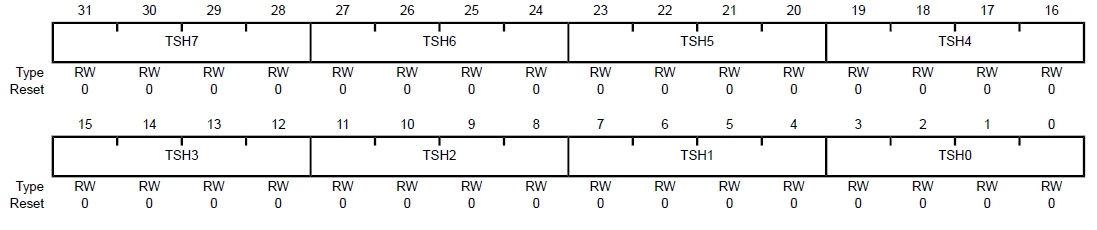


■ If VIND = 0, 转换结果= 0x800

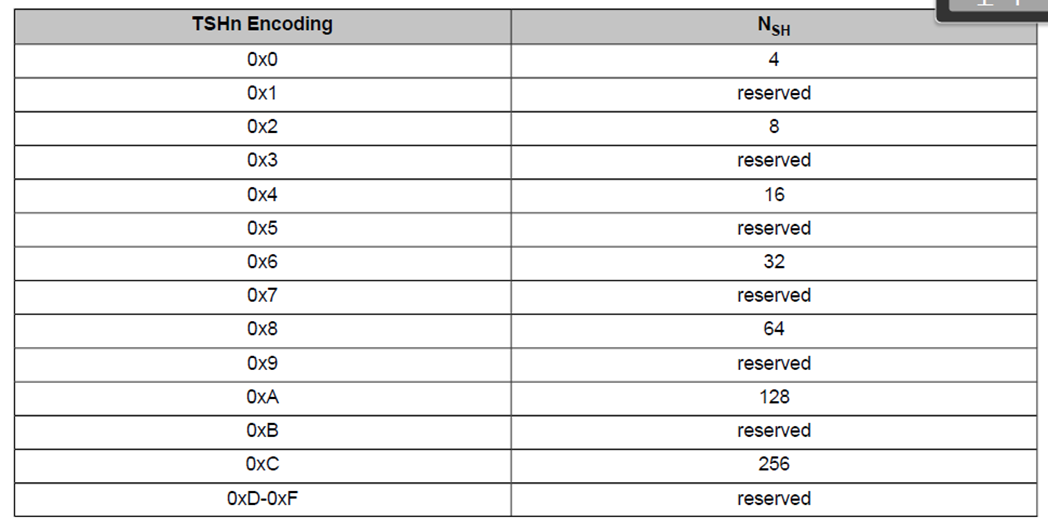
■ If VIND > 0, 转换结果> 0x800 (range is 0x800–0xFFF)

■ If VIND < 0, 转换结果< 0x800 (range is 0–0x800)

ADC Sample Sequence 0 Sample and Hold Time (**ADCSSTSH0**)寄存器，用于设置每次采样的保持时间：

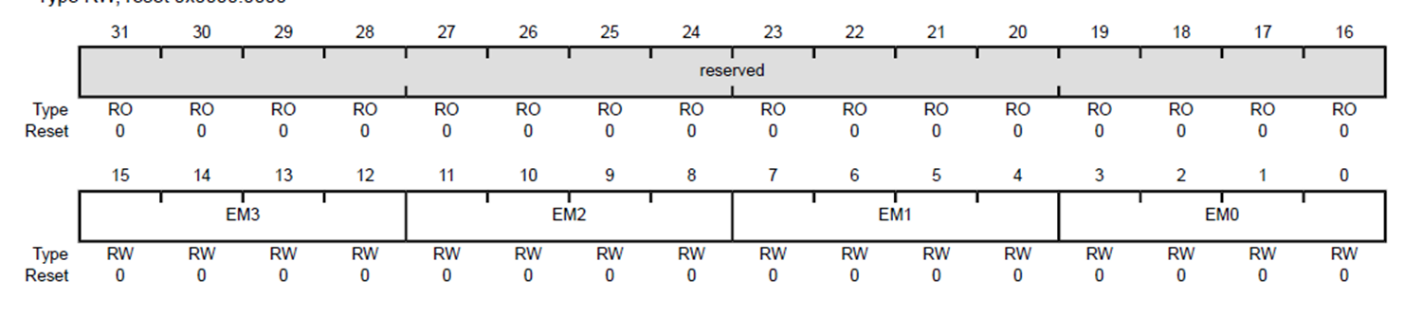


**ADCSSTSH0**寄存器有八个TSHn域，TSH0决定第零次采样的保持时间，以此类推。TSHn的值，与保持时间的对应关系为：



**5.3 选择采样序列开始转换的条件（触发条件）**

ADC Event Multiplexer Select (ADCEMUX) ADC事件选择寄存器用于设置四个采样序列的触发条件：



* EM0设置SS0的触发条件
* EM1设置SS1的触发条件
* EM2设置SS2的触发条件
* EM3设置SS3的触发条件

EMn域由4位组成，不同的值代表不同的触发条件：

**0x00: 由 ADCPSS0I. SSn手动触发**

0x01: 触发条件由ACCTL0配置

0x02: 触发条件由ACCTL1配置

0x03: 触发条件由ACCTL2配置

0x04: 由GPIO引脚触发，由GPIO模块中断的GPIOADCCTL寄存器配置

0x05: 由定时器触发

0x06: 由PWM0中断触发，由PWM0INTEN寄存器配置

0x07: 由PWM1中断触发，由PWM1INTEN寄存器配置

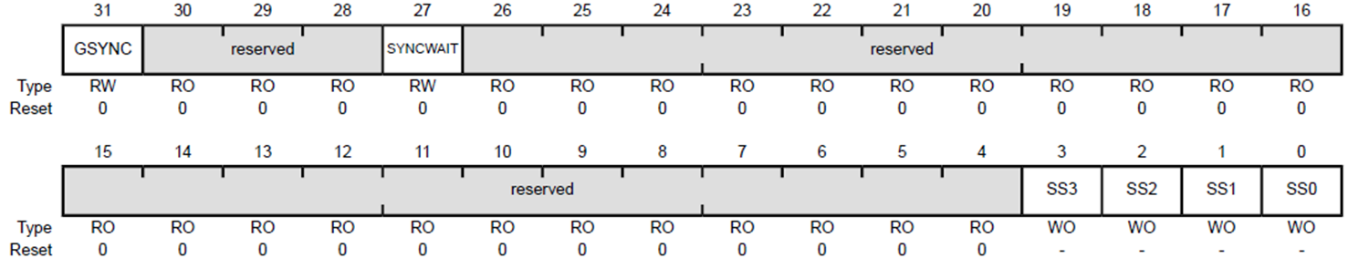
0x08: 由PWM2中断触发，由PWM2INTEN寄存器配置

0x09: 由PWM3中断触发，由PWM3INTEN寄存器配置

0x0E: 不触发

**0x0F: 连续触发**

ADC Processor Sample Sequence Initiate (**ADCPSSI**) ADC采样序列初始化寄存器，使用户可以用软件手动触发ADC的采样序列。



位SS0到SS3分别控制SS0到SS3的初始化，如果SYNCWAIT为零，采样序列初始化后立刻开始采样；如果SYNCWAIT为1，则等到GSYNC置1后开始采样。

* 1. **使用TivaWare库提供的函数，配置采样序列**

**5.4.1** TivaWare提供了ADCSequenceConfigure函数，设置采样序列的触发条件和优先级，其定义为：

void ADCSequenceConfigure (uint32\_t ui32Base, uint32\_t ui32SequenceNum, uint32\_t ui32Trigger, uint32\_t ui32Priority){

uint32\_t ui32Gen;

ASSERT((ui32Base == ADC0\_BASE) || (ui32Base == ADC1\_BASE));

ASSERT(ui32SequenceNum < 4);

ASSERT(((ui32Trigger & 0xF) == ADC\_TRIGGER\_PROCESSOR) ||

((ui32Trigger & 0xF) == ADC\_TRIGGER\_COMP0) ||

((ui32Trigger & 0xF) == ADC\_TRIGGER\_COMP1) ||

((ui32Trigger & 0xF) == ADC\_TRIGGER\_COMP2) ||

((ui32Trigger & 0xF) == ADC\_TRIGGER\_EXTERNAL) ||

((ui32Trigger & 0xF) == ADC\_TRIGGER\_TIMER) ||

((ui32Trigger & 0xF) == ADC\_TRIGGER\_PWM0) ||

((ui32Trigger & 0xF) == ADC\_TRIGGER\_PWM1) ||

((ui32Trigger & 0xF) == ADC\_TRIGGER\_PWM2) ||

((ui32Trigger & 0xF) == ADC\_TRIGGER\_PWM3) ||

((ui32Trigger & 0xF) == ADC\_TRIGGER\_ALWAYS) ||

((ui32Trigger & 0x30) == ADC\_TRIGGER\_PWM\_MOD0) ||

((ui32Trigger & 0x30) == ADC\_TRIGGER\_PWM\_MOD1));

ASSERT(ui32Priority < 4);

ui32SequenceNum \*= 4;

HWREG(ui32Base + ADC\_O\_EMUX) = ((HWREG(ui32Base + ADC\_O\_EMUX) &

~(0xf << ui32SequenceNum)) |

((ui32Trigger & 0xf) << ui32SequenceNum));

//

HWREG(ui32Base + ADC\_O\_SSPRI) = ((HWREG(ui32Base + ADC\_O\_SSPRI) &

~(0xf << ui32SequenceNum)) |

((ui32Priority & 0x3) <<

ui32SequenceNum));

//

// Set the source PWM module for this sequence's PWM triggers.

//

ui32Gen = ui32Trigger & 0x0f;

if(ui32Gen >= ADC\_TRIGGER\_PWM0 && ui32Gen <= ADC\_TRIGGER\_PWM3)

{

//

// Set the shift for the module and generator

//

ui32Gen = (ui32Gen - ADC\_TRIGGER\_PWM0) \* 8;

HWREG(ui32Base + ADC\_O\_TSSEL) = ((HWREG(ui32Base + ADC\_O\_TSSEL) &

~(0x30 << ui32Gen)) |

((ui32Trigger & 0x30) << ui32Gen));

}

}

ui32Base为ADC模块的基地址；

ui32SequenceNum为采样序列号，值可以为0, 1, 2, 3，用于计算设置ADCEMUX寄存器时，需要左移的位数。

ui32Trigger为采样序列的触发方式，设置用于ADCEMUX寄存器，值可以为：

**#define ADC\_TRIGGER\_PROCESSOR 0x00000000 // Processor event**

**#define ADC\_TRIGGER\_COMP0 0x00000001 // Analog comparator 0 event**

**#define ADC\_TRIGGER\_COMP1 0x00000002 // Analog comparator 1 event**

**#define ADC\_TRIGGER\_COMP2 0x00000003 // Analog comparator 2 event**

**#define ADC\_TRIGGER\_EXTERNAL 0x00000004 // External event**

**#define ADC\_TRIGGER\_TIMER 0x00000005 // Timer event**

**#define ADC\_TRIGGER\_PWM0 0x00000006 // PWM0 event**

**#define ADC\_TRIGGER\_PWM1 0x00000007 // PWM1 event**

**#define ADC\_TRIGGER\_PWM2 0x00000008 // PWM2 event**

**#define ADC\_TRIGGER\_PWM3 0x00000009 // PWM3 event**

**#define ADC\_TRIGGER\_NEVER 0x0000000E // Never Trigger**

**#define ADC\_TRIGGER\_ALWAYS 0x0000000F // Always event**

**#define ADC\_TRIGGER\_PWM\_MOD0 0x00000000 // PWM triggers from PWM0**

**#define ADC\_TRIGGER\_PWM\_MOD1 0x00000010 // PWM triggers from PWM1**

ui32Priority 为优先级，用于设置ADCSSPRI寄存器，值可以为0, 1, 2, 3，值越小，优先级越高

应用示例：

ADCSequenceConfigure(ADC0\_BASE, 0, ADC\_TRIGGER\_PROCESSOR, 0);

// 设置ADC0模块的SS0采样序列为软件手动触发，优先级为最高。

**5.4.2** TivaWare提供了ADCSequenceStepConfigure函数，用于设置采样序列中的采样通道的属性，其定义为：

void ADCSequenceStepConfigure (uint32\_t ui32Base, uint32\_t ui32SequenceNum, uint32\_t ui32Step, uint32\_t ui32Config)｛

uint32\_t ui32Temp;

ASSERT((ui32Base == ADC0\_BASE) || (ui32Base == ADC1\_BASE));

ASSERT(ui32SequenceNum < 4);

ASSERT(((ui32SequenceNum == 0) && (ui32Step < 8)) ||

((ui32SequenceNum == 1) && (ui32Step < 4)) ||

((ui32SequenceNum == 2) && (ui32Step < 4)) ||

((ui32SequenceNum == 3) && (ui32Step < 1)));

ui32Base += ADC\_SEQ + (ADC\_SEQ\_STEP \* ui32SequenceNum);

//

// Compute the shift for the bits that control this step.

//

ui32Step \*= 4;

//

// Set the analog mux value for this step.

//

HWREG(ui32Base + ADC\_SSMUX) = ((HWREG(ui32Base + ADC\_SSMUX) &

~(0x0000000f << ui32Step)) |

((ui32Config & 0x0f) << ui32Step));

//

// Set the upper bits of the analog mux value for this step.

//

HWREG(ui32Base + ADC\_SSEMUX) = ((HWREG(ui32Base + ADC\_SSEMUX) &

~(0x0000000f << ui32Step)) |

(((ui32Config & 0xf00) >> 8) << ui32Step));

//

// Set the control value for this step.

//

HWREG(ui32Base + ADC\_SSCTL) = ((HWREG(ui32Base + ADC\_SSCTL) &

~(0x0000000f << ui32Step)) |

(((ui32Config & 0xf0) >> 4) << ui32Step));

//

// Set the sample and hold time for this step. This is not available on

// all devices, however on devices that do not support this feature these

// reserved bits are ignored on write access.

//

HWREG(ui32Base + ADC\_SSTSH) = ((HWREG(ui32Base + ADC\_SSTSH) &

~(0x0000000f << ui32Step)) |

(((ui32Config & 0xf00000) >> 20) << ui32Step));

//

// Enable digital comparator if specified in the ui32Config bit-fields.

//

if(ui32Config & 0x000F0000)

{

//

// Program the comparator for the specified step.

//

ui32Temp = HWREG(ui32Base + ADC\_SSDC);

ui32Temp &= ~(0xF << ui32Step);

ui32Temp |= (((ui32Config & 0x00070000) >> 16) << ui32Step);

HWREG(ui32Base + ADC\_SSDC) = ui32Temp;

//

// Enable the comparator.

//

HWREG(ui32Base + ADC\_SSOP) |= (1 << ui32Step);

}

//

// Disable digital comparator if not specified.

//

else

{

HWREG(ui32Base + ADC\_SSOP) &= ~(1 << ui32Step);

}

}

ui32Base为ADC模块的基地址；

ui32SequenceNum为采样序列号，值可以为0, 1, 2, 3，用于计算该采样序列对应的ADC\_SSMUX、ADC\_SSEMUX、ADC\_SSCTL和ADC\_SSTSH等寄存器的地址。

ui32Step：采样序列中的第几次采样，对于SS0，这个参数的值可以使0-7。用于计算设置ADC\_SSMUX、ADC\_SSEMUX、ADC\_SSCTL和ADC\_SSTSH等寄存器时，需要左移多少位。

ui32Config：本次采样的采样通道号，及采样属性、保持时间。

采样通道号可选的值为ADC\_CTL\_CH0 到ADC\_CTL\_CH23，用于设置ADC\_SSMUX和ADC\_SSEMUX寄存器

采样属性的选项有：ADC\_CTL\_TS, ADC\_CTL\_IE, ADC\_CTL\_END, ADC\_CTL\_D，用于设置ADC\_SSCTL寄存器

#define ADC\_CTL\_TS 0x00000080 // Temperature sensor select

#define ADC\_CTL\_IE 0x00000040 // Interrupt enable

#define ADC\_CTL\_END 0x00000020 // Sequence end select

#define ADC\_CTL\_D 0x00000010 // Differential select

保持时间用于设置ADC\_SSTSH，其选项有：

#define ADC\_CTL\_SHOLD\_4 0x00000000 // Sample and hold 4 ADC clocks

#define ADC\_CTL\_SHOLD\_8 0x00200000 // Sample and hold 8 ADC clocks

#define ADC\_CTL\_SHOLD\_16 0x00400000 // Sample and hold 16 ADC clocks

#define ADC\_CTL\_SHOLD\_32 0x00600000 // Sample and hold 32 ADC clocks

#define ADC\_CTL\_SHOLD\_64 0x00800000 // Sample and hold 64 ADC clocks

#define ADC\_CTL\_SHOLD\_128 0x00A00000 // Sample and hold 128 ADC clocks

#define ADC\_CTL\_SHOLD\_256 0x00C00000 // Sample and hold 256 ADC clocks

将这些选项按位或后，赋值给ui32Config。

应用示例：

ADCSequenceStepConfigure(ADC0\_BASE, 0, 0, ADC\_CTL\_TS); //SS0序列第0个转换的通道为内部温度传感器的输出电压

ADCSequenceStepConfigure(ADC0\_BASE, 0, 5, ADC\_CTL\_CH3|ADC\_CTL\_IE|ADC\_CTL\_END);

//SS0序列第5个转换的通道为AIN3引脚的电压，本次转换完成后，触发SS0序列的中断，本次转换为SS0采样序列的最后一次转换。

**5.4.3** TivaWare提供了ADCSequenceEnable函数，来设置ADCACTSS寄存器，使能采样序列，其定义为

void ADCSequenceEnable (uint32\_t ui32Base, uint32\_t ui32SequenceNum)

{

//

// Check the arguments.

//

ASSERT((ui32Base == ADC0\_BASE) || (ui32Base == ADC1\_BASE));

ASSERT(ui32SequenceNum < 4);

//

// Enable the specified sequence.

//

HWREG(ui32Base + ADC\_O\_ACTSS) |= 1 << ui32SequenceNum;

}

ui32Base为ADC模块的基地址；

ui32SequenceNum为采样序列号，值可以为0, 1, 2, 3

应用示例：

ADCSequenceEnable(ADC0\_BASE, 0); //使能ADC0模块的SS0采样序列

1. **使能并配置中断**

**6.1在PIE中使能ADC0模块的SS0序列的中断**

ADC0模块的SS0到SS3序列，都可以单独向中断管理器发出中断请求，因此需要在中断管理器模块中，使能ADC0模块的SS0序列的中断：

IntEnable(INT\_ADC0SS0);

**6.2注册中断服务函数**

TivaWare提供了ADCIntRegister注册ADC模块指定采样序列的中断服务函数，其定义为：

void ADCIntRegister (uint32\_t ui32Base, uint32\_t ui32SequenceNum, void (pfnHandler)(void))

其中

 ui32Base为ADC模块的基地址；

 ui32SequenceNum为采样序列号，值可以为0, 1, 2, 3

pfnHandler 为中断服务函数地址

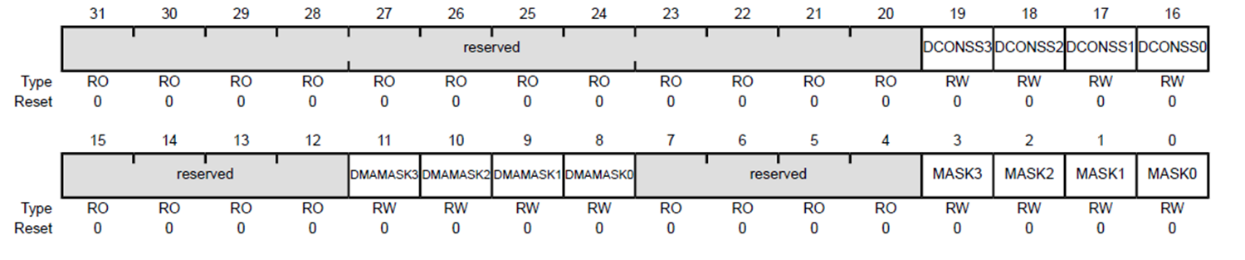
例如：

ADCIntRegister(ADC0\_BASE,0 , adc0\_ss0\_isr )

//把函数adc0\_ss0\_isr注册成为ADC0模块SS0采样序列的中断服务函数。

**6.3 使能ADC模块中用到的采样序列(SS0)的中断**

ADC模块的ADC Interrupt Mask (ADCIM)寄存器的低四位，控制四个采样序列的中断使能：



ADCIM的MASK0位置1，就可以使能SS0的中断。

TivaWare提供了ADCIntEnable函数，童工操作ADCIM寄存器，开启采样序列的中断，其定义为：

void

ADCIntEnableEx(uint32\_t ui32Base, uint32\_t ui32IntFlags)

{

//

// Check the arguments.

//

ASSERT((ui32Base == ADC0\_BASE) || (ui32Base == ADC1\_BASE));

//

// Enable the requested interrupts.

//

HWREG(ui32Base + ADC\_O\_IM) |= ui32IntFlags;

}

其用法为：

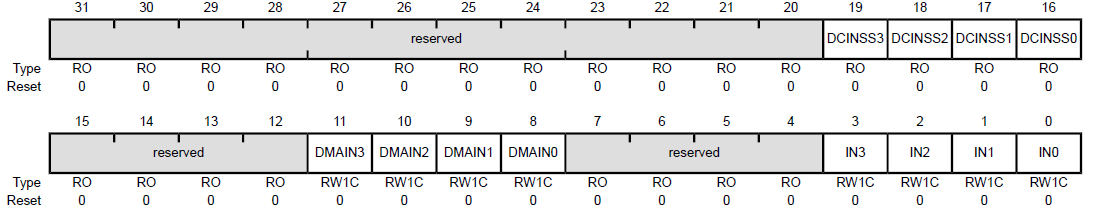
ADCIntEnable(ADC0\_BASE,0);

其中第一个参数为ADC0模块的基地址，第二个参数为采样序列号，可以为0, 1, 2, 3。

**6.4 清除ADC模块中用到的采样序列(SS0)的中断**

ADC模块的中断功能，跟其他外设模块一样，除了使能中断之外，还需要查看中断的状态，并清除中断。

ADC Interrupt Status and Clear (ADCISC) ADC中断状态及清除寄存器，可以查看ADC模块中断的状态，并清除中断：



IN0到IN3分别代表SS0到SS3的中断状态，对IN0写1，清除SS0的中断。

TivaWare提供了ADCIntClear函数，通过操作ADCISC寄存器，清除ADC模块中指定采样序列的中断标志，其定义为：

void

ADCIntClear(uint32\_t ui32Base, uint32\_t ui32SequenceNum)

{

//

// Check the arguments.

//

ASSERT((ui32Base == ADC0\_BASE) || (ui32Base == ADC1\_BASE));

ASSERT(ui32SequenceNum < 4);

//

// Clear the interrupt.

//

HWREG(ui32Base + ADC\_O\_ISC) = 1 << ui32SequenceNum;

}

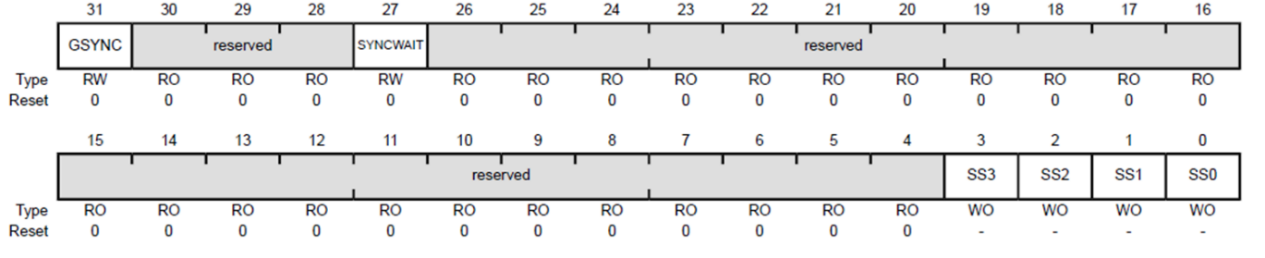
用法为：

**ADCIntClear(ADC0\_BASE, 0);**

以上代码清除ADC0模块的SS0采样序列的中断状态。

1. **开始采样**

ADC Processor Sample Sequence Initiate (ADCPSSI) ADC采样序列初始化寄存器，可以让用户手动触发ADC模块中指定的采样序列。



如果ADCEMUX寄存器被配置为软件手动触发，TivaWare提供了ADCProcessorTrigger函数，操作ADCPSSI寄存器，手动触发ADC采样，其定义为：

void

ADCProcessorTrigger(uint32\_t ui32Base, uint32\_t ui32SequenceNum)

{

//

// Check the arguments.

//

ASSERT((ui32Base == ADC0\_BASE) || (ui32Base == ADC1\_BASE));

ASSERT(ui32SequenceNum < 4);

//

// Generate a processor trigger for this sample sequence.

//

HWREG(ui32Base + ADC\_O\_PSSI) |= ((ui32SequenceNum & 0xffff0000) |

(1 << (ui32SequenceNum & 0xf)));

}

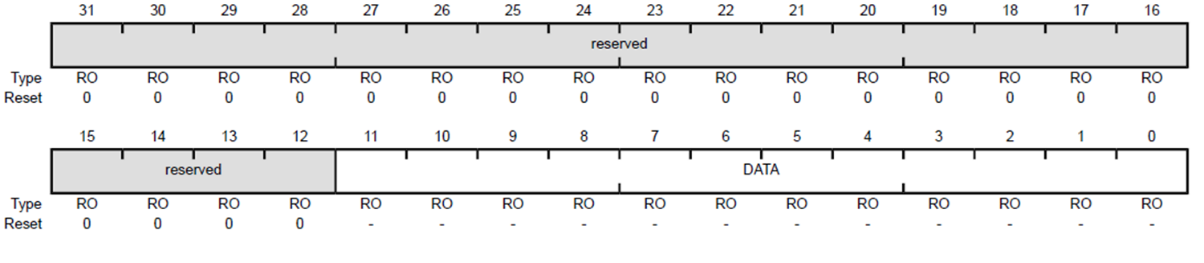
用法为

**ADCProcessorTrigger(ADC0\_BASE, 0);**

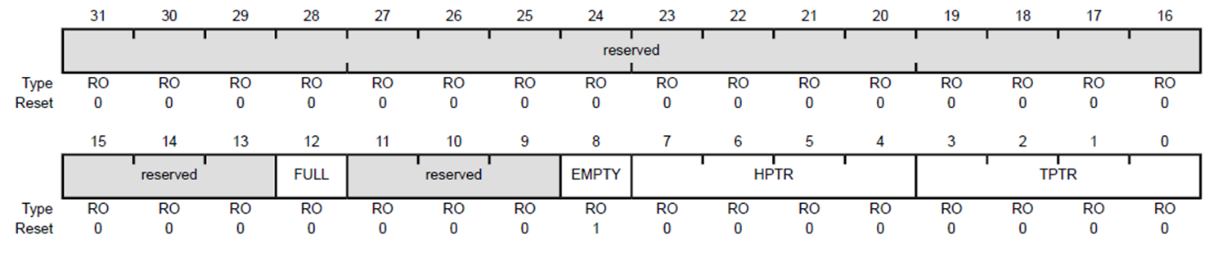
第一个参数为ADC模块的基地址，第二个参数为采样序列的编号，可以是0，1，2，3。

1. **编写中断服务函数，读取数据**

ADC模块通过ADC Sample Sequence Result FIFO 0/1/2/3读取采样序列SS0/1/2/3的转换结果，通过读该寄存器，把转换结果从FIFIO中读出来。



ADC模块提供了FIFO状态寄存器 ADC Sample Sequence FIFO 0/1/2/3 Status 查看结果FIFO的状态



EMPTY域表示FIFO为空，FULL域表示FIFO满。

转换结束后，需要连续读取FIFO，把所有转换结果都取出来，直到FIFO为空为止。

TivaWare提供了ADCSequenceDataGet函数，把FIFO中的转换结果读出来，其定义为：

int32\_t

ADCSequenceDataGet(uint32\_t ui32Base, uint32\_t ui32SequenceNum,

uint32\_t \*pui32Buffer)

{

uint32\_t ui32Count;

ASSERT((ui32Base == ADC0\_BASE) || (ui32Base == ADC1\_BASE));

ASSERT(ui32SequenceNum < 4);

ui32Base += ADC\_SEQ + (ADC\_SEQ\_STEP \* ui32SequenceNum);

ui32Count = 0;

while(!(HWREG(ui32Base + ADC\_SSFSTAT) & ADC\_SSFSTAT0\_EMPTY) &&

(ui32Count < 8))

{

//

// Read the FIFO and copy it to the destination.

//

\*pui32Buffer++ = HWREG(ui32Base + ADC\_SSFIFO);

//

// Increment the count of samples read.

//

ui32Count++;

}

//

// Return the number of samples read.

//

return(ui32Count);

}

可见，该函数连续读取FIFO，把所有转换结果都取出来，直到FIFO为空为止。

用法为：

ADCSequenceDataGet(ADC0\_BASE, 0, ui32ACCValues);

第一个参数为ADC0的基地址，第二个参数为SS0的序列编号，第三个参数为保持转换结果的数组，用uint32\_t ui32ACCValues[8];定义。

以下给出了一个ADC0模块，SS0采样序列的中断服务函数：

void adc0\_ss0\_isr(void){

ADCIntClear(ADC0\_BASE, 0);

ADCSequenceDataGet(ADC0\_BASE, 0, ui32ACCValues);

}