AN0124
入门指南

AT32F425 ADC使用指南

前言

AT32F425 的 ADC 是一个将模拟输入信号转换为 12bit 分辨率数位数字信号的外设。采样率最高可达 2MSPS。多达 18 个通道源可进行采样及转换。具备多种功能强大的模式，本文主要以 ADC 的特色功能进行讲解和案列解析。

支持型号列表：

支持型号	AT32F425xx

目录

1 ADC 简介 7

2 ADC 功能解析 9

2.1 时钟及状态 9

2.1.1 功能介绍 9

2.1.2 软件接口 9

2.2 采样转换 9

2.2.1 功能介绍 9

2.2.2 软件接口 10

2.3 自校准 10

2.3.1 功能介绍 10

2.3.2 软件接口 10

2.4 基本模式 10

2.4.1 功能介绍 10

2.4.2 软件接口 12

2.5 不同优先权的通道 13

2.5.1 功能介绍 13

2.5.2 软件接口 13

2.6 多种独立的触发源 13

2.6.1 功能介绍 13

2.6.2 软件接口 14

2.7 数据后级处理 14

2.7.1 功能介绍 14

2.7.2 软件接口 15

2.8 过采样器 15

2.8.1 功能介绍 15

2.8.2 软件接口 17

2.9 电压监测 18

2.9.1 功能介绍 18
2.9.2 软件接口 18
2.10 中断及状态事件 18
2.10.1 功能介绍 18
2.10.2 软件接口 19
2.11 多种转换数据的获取方式 19
2.11.1 功能介绍 19
2.11.2 软件接口 19

3 ADC 配置解析 21

3.1 ADC 配置流程 21
3.2 ADC 数据获取方法 22

4 ADC 案例 24

4.1 案例 ADC 侦测 Vref 电压 24
4.1.1 功能简介 24
4.1.2 资源准备 24
4.1.3 软件设计 24
4.2 案例 ADC EXINT 触发+分割模式 26
4.2.1 功能简介 26
4.2.2 资源准备 26
4.2.3 软件设计 26
4.3 案例 ADC 过采样 30
4.3.1 功能简介 30
4.3.2 资源准备 30
4.3.3 软件设计 31
4.4 案例 ADC 软件触发+反复模式 34
4.4.1 功能简介 34
4.4.2 资源准备 34
4.4.3 软件设计 35
4.5 案例 ADC 定时器触发+抢占自动转换模式 37

4.5.1 功能简介 37
4.5.2 资源准备 37
4.5.3 软件设计 37
4.6 案例 ADC 电压监测 41
4.6.1 功能简介 41
4.6.2 资源准备 42
4.6.3 软件设计 42
5 文档版本历史 45

表目录

表 1. 触发源 14
表 2. 最大累加数据与过采样倍数及位移系数关系 15
表 3. 文档版本历史 45

图目录

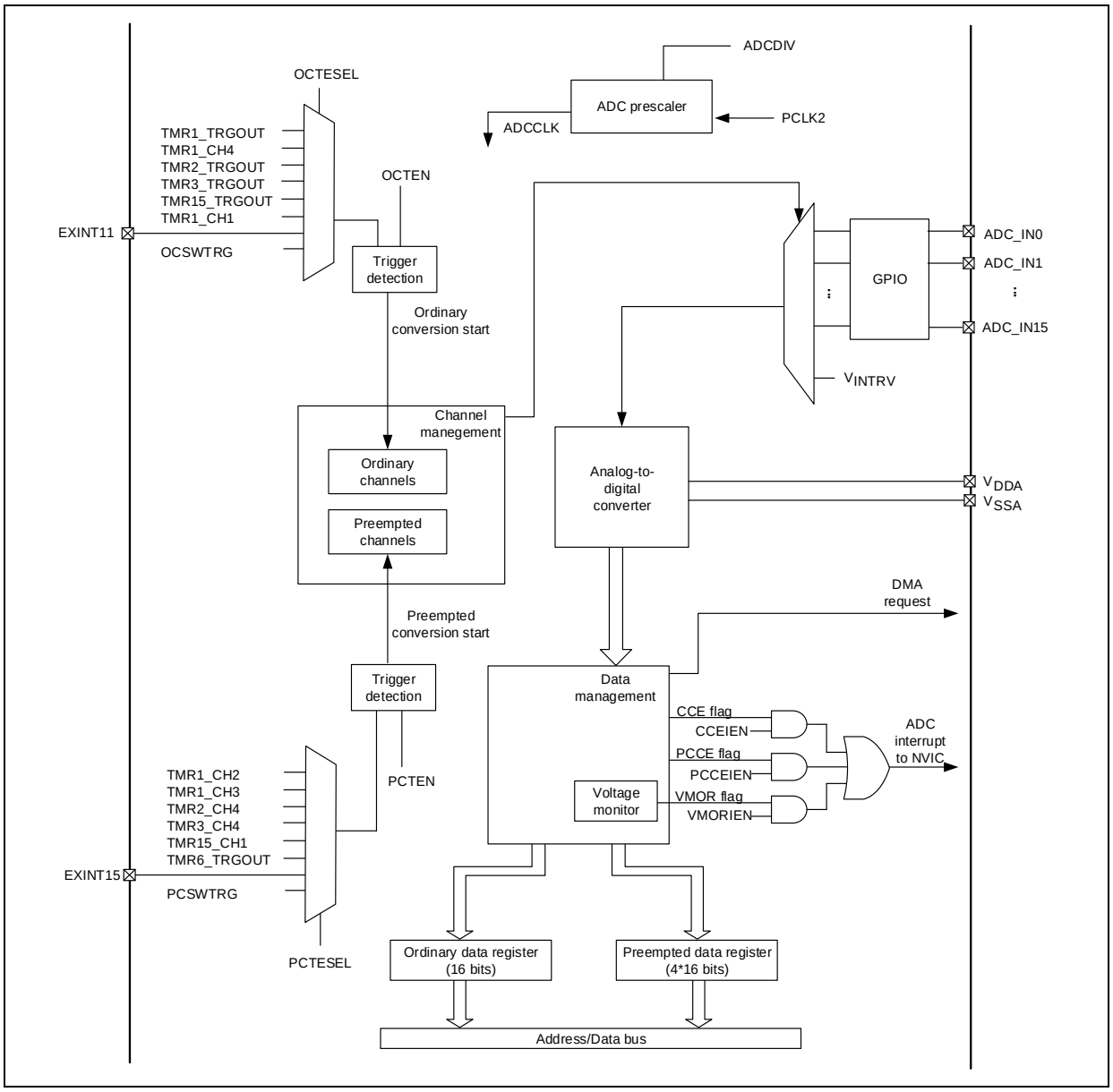
图 1. ADC1 框图 8
图 2. 序列模式 11
图 3. 反复模式+抢占自动转换模式 11
图 4. 分割模式 11
图 5. 抢占自动转换模式 12
图 6. 数据内容处理 15
图 7. 普通过采样被打断后的恢复方式 16
图 8. 普通过采样触发模式 17
图 9. 抢占自动转换下的过采样模式 17

1 ADC 简介

ADC 控制器的功能极其强大。其包含但不限于以下内容

- 时钟及状态，由数字和模拟时钟两个部分组成
- 支持分辨率为 12 位的转换，采样周期支持广范围的配置
- 自校准，自带校准功能以纠正数据偏移
- 基本模式，支持多种模式，不同模式可组合使用满足多种应用
- 不同优先权的通道，普通通道与抢占通道具备不同的优先权
- 多种独立的触发源，包括 TMR、EXINT、软触发等多种触发选择
- 数据后级处理，包括数据的对齐，抢占通道偏移量等多种处理
- 过采样器，普通及抢占通道均支持过采样
- 电压监测，通过对转换结果的判定来实现电压监测
- 中断及状态事件，具备多种标志指示 ADC 状态，且某些标志还具备中断功能
- 多种转换数据的获取方式，包括 DMA 获取、CPU 获取两种方式实现转换数据的读取

图 1. ADC1 框图


ADCDIV
OCTESEL
ADC prescaler
PCLK2
ADCCLK
TMR1_TRGOUT
TMR1_CH4
TMR2_TRGOUT
TMR3_TRGOUT
TMR15_TRGOUT
TMR1_CH1
OCTEN
EXINT11
OCSWTRG
Trigger detection
Ordinary conversion start
GPIO
ADC_IN0
ADC_IN1
ADC_IN15
VINTRV
Channel manegement
Ordinary channels
Preempted channels
Analog-to-digital converter
VDDA
VSSA
DMA request
Preempted conversion start
Trigger detection
Data management
CCE flag
CCEIEN
PCCE flag
PCCEIEN
VMOR flag
VMORIEN
ADC interrupt to NVIC
Voltage monitor
TMR1_CH2
TMR1_CH3
TMR2_CH4
TMR3_CH4
TMR15_CH1
TMR6_TRGOUT
PCTEN
EXINT15
PCSWTRG
PCTESEL
Ordinary data register (16 bits)
Preempted data register (4*16 bits)
Address/Data bus

2 ADC 功能解析

2.1 时钟及状态

2.1.1 功能介绍

ADC 的时钟分为数字时钟与模拟时钟。其统一通过 CRM_APB2EN 的 ADCxEN 位使能。

- 数字时钟：即 PCLK2，经 HCLK 分频而来，提供给数字部分使用。
- 模拟时钟：即 ADCCLK，经 ADC 预分频器分频而来，提供给模拟部分使用。

2.1.2 软件接口

ADC 时钟使能，软件由单独的函数接口实现，其软件实例如下：

```
crm_periph_clock_enable(CRM_ADCx_PERIPH_CLOCK, TRUE);
```

当 ADC 时钟使能后，软件即可开始进行 ADC 的一些相关配置。

ADC 预分频设定，软件由单独的函数接口实现，其软件实例如下：

```
crm_adc_clock_div_set(CRM_ADC_DIV_6);
```

此项实际用于设定 ADC 模拟部分的时钟，其由 PCLK2 分频而来，故 ADCCLK= PCLK2/ div

注意：

1）模拟部分的 ADCCLK 由 PCLK2 分频而来，其不可大于 28MHz；

2）ADC 模拟部分电源由 ADC_CTRL2 的 ADCEN，其不受 ADC 的时钟状态影响。典型的，如果系统需要进入深度睡眠模式，如果不关闭 ADCEN，此时 ADC 模拟器件将还会消耗电流；

3）ADC 上电有一段等待时间 t_{STAB}。

4）为避免充电不充分导致转换数据不准确，应用允许的条件下，建议合理增大采样周期。

2.2 采样转换

2.2.1 功能介绍

ADC 可设定 1.5、7.5、13.5、28.5、41.5、55.5、71.5、239.5 个采样周期。

ADC 对通道数据的获取由采样和转换两个部分组成。

采样先于转换执行，采样期间内选通需要转换的通道，外部电压对 ADC 内部采样电容充电，将持续执行设定的采样周期长度时间的充电。

采样结束后就会自动开始转换，ADC 采用逐次逼近的转换方式，可有效保障转换数据的准确性。此转换方式需要分辨率位数个 ADCCLK 的转换时间来完成单通道的转换，再结合数据处理，因此单个通道的整体转换时间即

```
单通道单次转换时间（ADCCLK 的周期） = 采样周期 + 12.5
```

示例：

CSPTx 选择 7.5 周期，一次转换需要 7.5+12.5 = 20 个 ADCCLK 周期。

2.2.2 软件接口

ADC 采样周期设定，软件由单独的函数接口实现，其软件实例如下：

```
adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_13_5);
adc_preempt_channel_set(ADC1, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_41_5);
```

注意：

不同通道可设定不同的采样周期；

当采用中断或轮询方式获取普通通道数据，为避免数据读取不及时，建议合理增大采样周期；

为避免充电不充分导致转换数据不准确，应用允许的条件下，建议合理增大采样周期。

2.3 自校准

2.3.1 功能介绍

ADC 具备自校准能力，软件可以执行自校准命令，透过自校准可以计算出一个校准值。不需要软件干预，ADC 会自动将该校准值反馈回 ADC 内部补偿 ADC 基础偏差，以保障转换数据的准确性。

自校准的软件流程如下：

- 使能 ADC
- 执行初始化校准命令并等待初始化校准完成
- 执行校准命令并等待校准完成
- 执行完上述流程后，即可开始进行 ADC 的触发转换

2.3.2 软件接口

自校准方式，其软件实例如下：

```
adc_enable(ADC1, TRUE);
adc_calibration_init(ADC1);
while(adc_calibration_init_status_get(ADC1));
adc_calibration_start(ADC1);
while(adc_calibration_status_get(ADC1));
```

注意：校准值的存放不会置位 CCE 标志，不会产生中断或 DMA 请求。

2.4 基本模式

2.4.1 功能介绍

序列模式

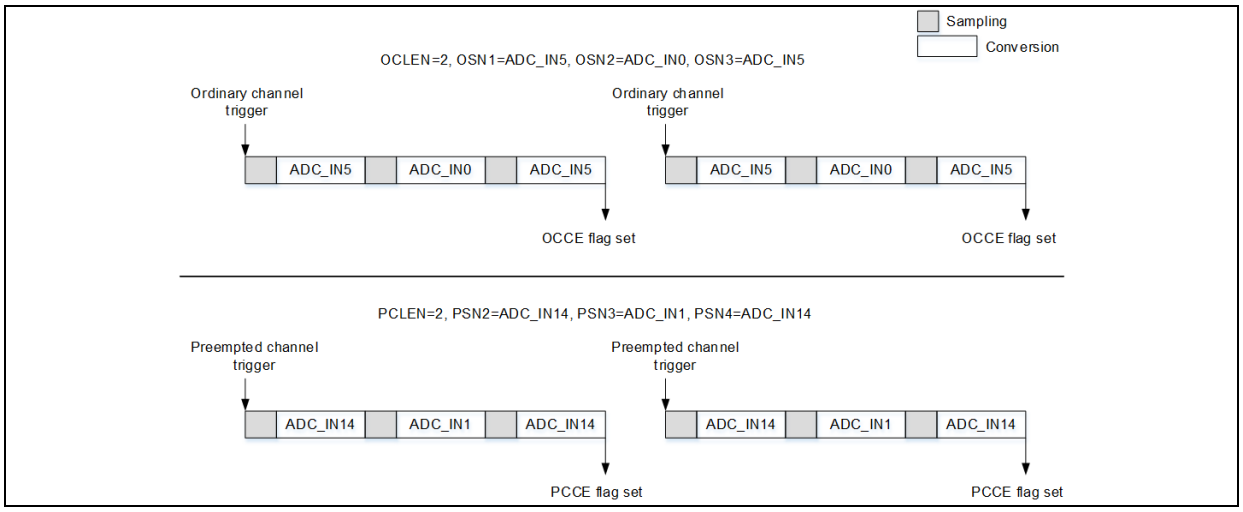
ADC 支持序列模式设定，开启序列模式后，每次触发将序列中的通道依序转换一次。

用户于 ADC_OSQx 配置普通通道序列，普通通道从 OSN1 开始转换；于 ADC_PSQ 配置抢占通道序列，抢占通道是从 PSNx 开始转换(x=4-PCLEN)。

抢占通道转换示例：

ADC_PSQ[21：0] = 10 00110 00101 00100 00011，此时扫描转换顺序为 CH4、CH5、CH6，而不是 CH3、CH4、CH5。

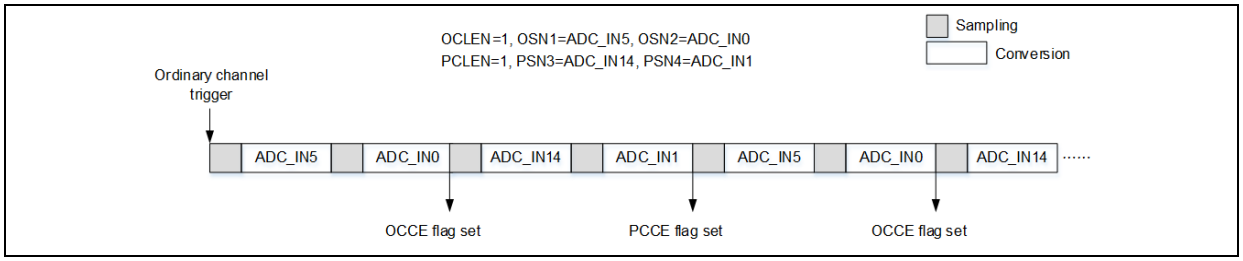
图 2. 序列模式



反复模式

ADC 支持反复模式设定，开启反复模式后，当检测到触发后就即会反复不断地转换普通通道组。

图 3. 反复模式+抢占自动转换模式



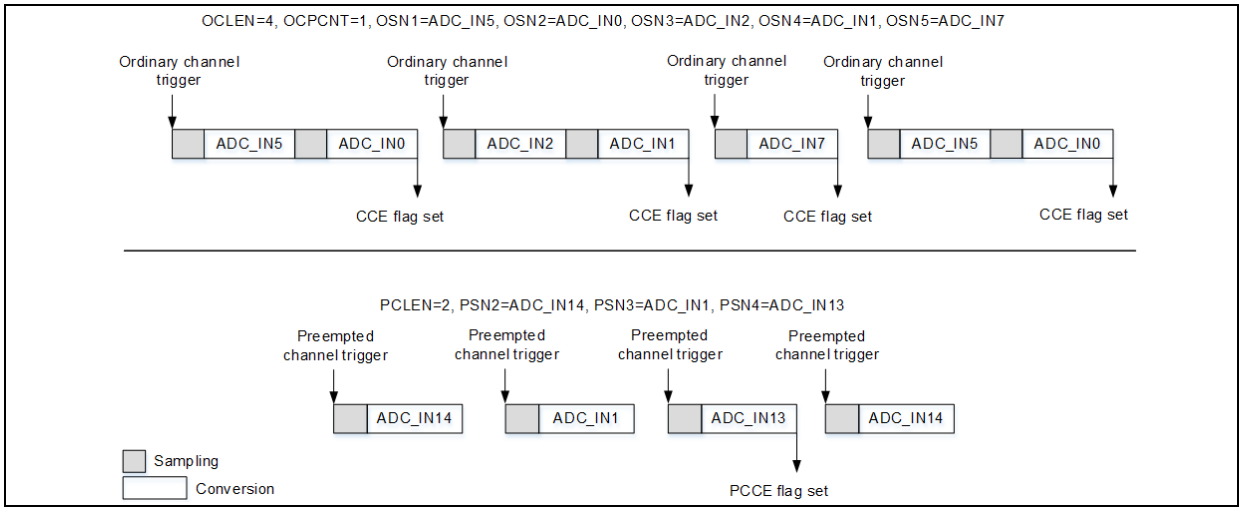
分割模式

ADC 支持分割模式设定。

对于普通通道组，分割模式可依据设定将通道组分割成长度较小的子组别。一次触发将转换子组别中的所有通道。每次触发会依序选择不同的子组别进行转换。

对于抢占通道组，分割模式直接以通道为单位进行分割，一次触发将转换单个通道。每次触发会依序选择不同的通道进行转换。

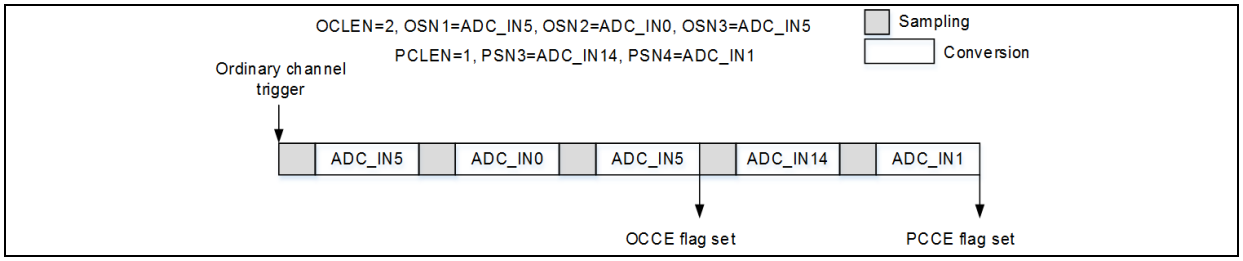
图 4. 分割模式



抢占自动转换模式

ADC 支持抢占自动转换模式设定，开启抢占自动转换模式后，当普通通道转换完成后，抢占通道将自动接续着转换，而不需要进行抢占通道的触发。

图 5. 抢占自动转换模式



2.4.2 软件接口

ADC 序列模式和反复模式设定，由 ADC 基础部分结构体配置完成，其软件实例如下：

```
adc_base_struct.sequence_mode = TRUE;
adc_base_struct.repeat_mode = TRUE;
adc_base_config(ADC1, &adc_base_struct);
```

注意：

序列模式对普通及抢占通道组均有效；

反复模式仅对普通通道组有效,抢占通道组不具备反复模式功能；

反复模式与分割模式不可共用；

反复模式可与抢占自动转换模式共用，将实现依次反复的转换普通通道序列及抢占通道序列。

ADC 分割模式设定，软件由单独的函数接口实现，其软件实例如下：

```
/* set ordinary partitioned mode channel count */
adc_ordinary_part_count_set(ADC1, 1);

/* enable the partitioned mode on ordinary channel */
adc_ordinary_part_mode_enable(ADC1, TRUE);

/* enable the partitioned mode on preempt channel */
adc_preempt_part_mode_enable(ADC1, TRUE);
```

注意：

分割模式对普通及抢占通道组均有效;

抢占通道组分割模式子组别长度不可设定，其固定为单个通道；

分割模式与反复模式、抢占自动转换模式不可共用。

抢占自动转换模式设定，软件由单独的函数接口实现，其软件实例如下：

```
/* preempt group automatic conversion after ordinary group */
adc_preempt_auto_mode_enable(ADC1, TRUE);
```

注意：

抢占自动转换模式仅对抢占通道组有效;

抢占自动转换模式与分割模式不可共用。

2.5 不同优先权的通道

2.5.1 功能介绍

ADC 设计有具备不同优先权的两种通道组：普通通道组与抢占通道组。

普通通道组

通常用于执行常规的数据转换。支持最多配置 16 个通道，转换将按照设定的通道顺序依次进行。其不具备抢占能力。

抢占通道组

通常用于执行相对紧急的数据转换。支持最多配置 4 个通道，转换将按照设定的通道顺序依次进行。其具备抢占能力，即抢占通道组的转换可以打断正在执行的普通通道转换，待抢占通道组转换完毕后再恢复执行被打断的普通通道组转换。

2.5.2 软件接口

普通通道组设定，软件包括通道数量、通道数值、转换顺序、采样周期的设定，其软件实例如下：

```
/* config ordinary channel count */
adc_base_struct.ordinary_channel_length = 3;
adc_base_config(ADC1, &adc_base_struct);

/* config ordinary channel */
adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_41_5);
adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_41_5);
adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_41_5);
```

抢占通道组设定，软件包括通道数量、通道数值、转换顺序、采样周期的设定，其软件实例如下：

```
/* config preempt channel count */
adc_preempt_channel_length_set(ADC1, 3);

/* config preempt channel */
adc_preempt_channel_set(ADC1, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_41_5);
adc_preempt_channel_set(ADC1, ADC_CHANNEL_8, 2, ADC_SAMPLETIME_41_5);
adc_preempt_channel_set(ADC1, ADC_CHANNEL_9, 3, ADC_SAMPLETIME_41_5);
```

注意：

不同通道可以设定不同的采样周期；

同一通道可以被反复编排进转换序列进行转换；

序列模式下，普通通道组从 OSN1 开始转换，抢占通道组是从 PSNx 开始转换(x=4-PCLEN)。

2.6 多种独立的触发源

2.6.1 功能介绍

ADC 支持多种触发源，包含软件写寄存器触发（ADC_CTRL2 的 OCSWTRG 与 PCSWTRG）以及外部触发。外部触发包含定时器触发与引脚触发。

普通通道还有一种特殊的触发来源，即重复使能 ADCEN 触发转换。此种情况下不需要使能 ADC

控制寄存器 2（ADC_CTRL2）的 OCTEN 也可导致普通通道响应转换。

注意：定时器触发、引脚触发、软件触发均需要使能触发模式（ADC_CTRL2 的 OCTEN 与 PCTEN）。

表 1. 触发源

OCTESEL	触发来源	PCTESEL	触发来源
000	TMR1_TRGOUT event	000	TMR1_CH2 event
001	TMR1_CH4 event	001	TMR1_CH3 event
010	TMR2_TRGOUT event	010	TMR2_CH4 event
011	TMR3_TRGOUT event	011	TMR3_CH4 event
100	TMR15_TRGOUT event	100	TMR15_CH1 event
101	TMR1_CH1 event	101	TMR6_TRGOUT event
110	EXINT line11 external pin	110	EXINT line15 external pin
111	OCSWTRG bit	111	PCSWTRG bit

2.6.2 软件接口

软件写寄存器触发设定，软件由单独的函数接口实现，其软件实例如下：

```
/* config ordinary trigger source */
adc_ordinary_conversion_trigger_set(ADC1, ADC12_ORDINARY_TRIG_SOFTWARE, TRUE);

/* config preempt trigger source */
adc_preempt_conversion_trigger_set(ADC1, ADC12_PREEMPT_TRIG_SOFTWARE, TRUE);
```

在 ADC 使能 t_{STAB} 后，软件即可执行 adc_ordinary_software_trigger_enable(ADC1, TRUE);/ adc_preempt_software_trigger_enable (ADC1, TRUE);来进行普通/抢占通道的触发。

外部触发设定，软件由单独的函数接口实现，其软件实例如下：

```
/* config ordinary trigger source */
adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1CH1, TRUE);

/* config preempt trigger source */
adc_preempt_conversion_trigger_set(ADC1, ADC_PREEMPT_TRIG_TMR3CH4, TRUE);
```

在 ADC 使能 t_{STAB} 后，TMR1CH1 的上升沿事件就会触发普通通道组转换，TMR3CH4 的上升沿事件就会触发抢占通道组转换。

注意：

触发间隔需要大于通道组转换的时间，转换期间发生的相同通道组的触发会被忽略；

抢占通道转换优先权最高，不管当前是否有普通通道转换，其触发后就会立即开始响应转换；

普通触发具备记忆功能，在抢占转换时执行普通触发，该触发会被记录并在抢占转换完毕后响应。

2.7 数据后级处理

2.7.1 功能介绍

ADC 具备专有的数据寄存器，普通通道转换完成后数据存储于普通数据寄存器（ADC_ODT），抢占通道转换完成后数据存储于抢占数据寄存器 x（ADC_PDTx）。数据寄存器内存储的是经过处理后的数据。该处理包括数据对齐、抢占数据偏移。

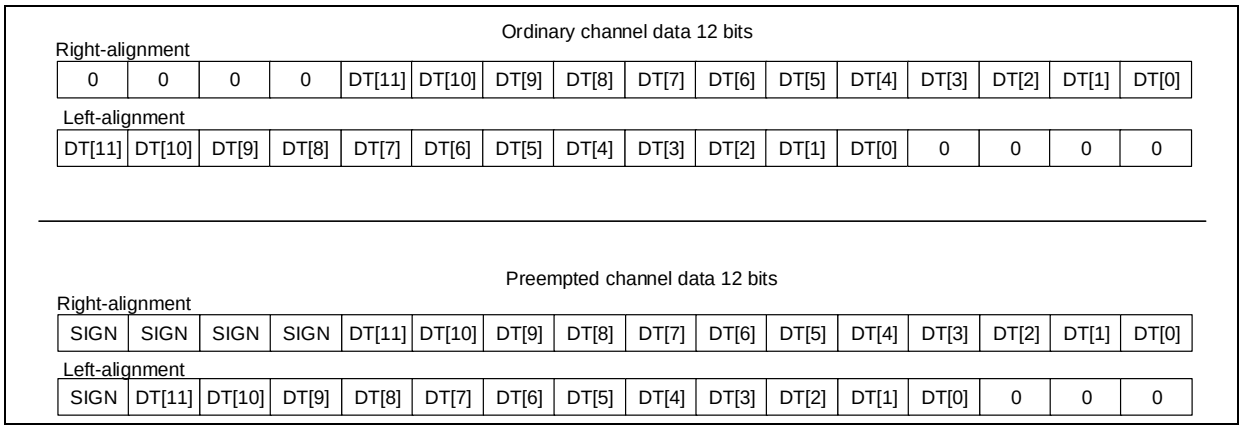
数据对齐

分左对齐和右对齐，以半字为基准摆放。

抢占数据偏移

抢占通道的数据会减去抢占数据偏移寄存器 x（ADC_PCDTOx）内的偏移量，因此抢占通道数据有可能为负值，以 SIGN 作为符号。

图 6. 数据内容处理



2.7.2 软件接口

数据对齐设定，软件由 ADC 基础部分结构体配置完成，其软件实例如下：

```
adc_base_struct.data_align = ADC_RIGHT_ALIGNMENT;
adc_base_config(ADC1, &adc_base_struct);
```

抢占数据偏移设定，软件由单独的函数接口实现，其软件实例如下：

```
/* set preempt channel's conversion value offset */
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_1, 0x000);
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_2, 0x111);
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_3, 0x202);
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_4, 0x123);
```

2.8 过采样器

2.8.1 功能介绍

ADC 具备过采样功能。一次过采样是透过转换多次相同通道，累加转换数据后作平均实现的。

- 由 ADC_OVSP 的 OSRSEL 选择过采样率，此位用来定义过采样倍数；
- 由 ADC_OVSP 的 OSSSEL 选择过采样移位，此位用来定义平均系数。

若平均后数据大于 16 位，只取靠右 16 位数据，放入 16 位数据寄存器。

使用过采样时，忽视数据对齐及抢占数据偏移的设定，数据一律靠右摆放。

表 2. 最大累加数据与过采样倍数及位移系数关系

过采样率	2x	4x	8x	16x	32x	64x	128x	256x
最大累加数据	**0x1FFE**	**0x3FFC**	**0x7FF8**	**0xFFF0**	**0x1FFE0**	**0x3FFC0**	**0x7FF80**	**0xFFF00**
不移位	0x1FFE	0x3FFC	0x7FF8	0xFFF0	0xFFE0	0xFFC0	0xFF80	0xFF00

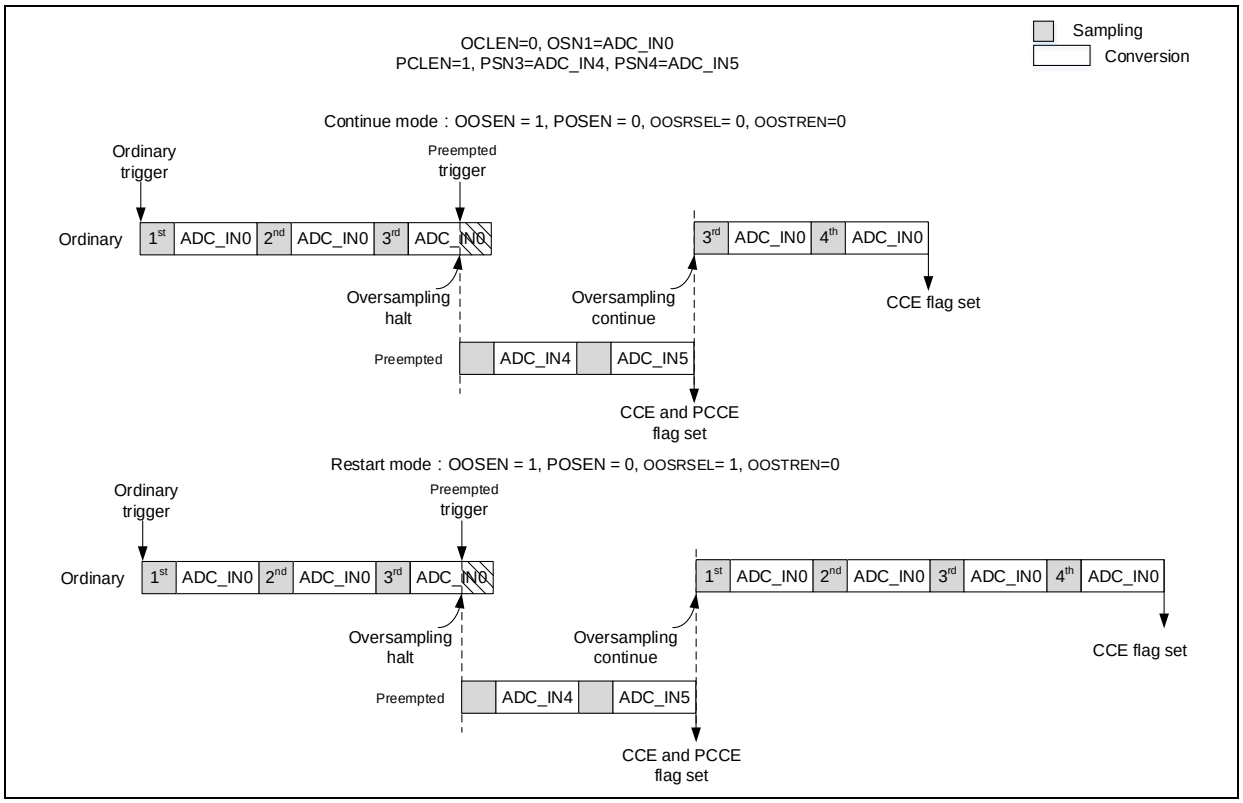
过采样率	2x	4x	8x	16x	32x	64x	128x	256x
移 1 位	0x0FFF	0x1FFE	0x3FFC	0x7FF8	0xFFF0	0xFFE0	0xFFC0	0xFF00
移 2 位	0x0800	0x0FFF	0x1FFE	0x3FFC	0x7FF8	0xFFF0	0xFFE0	0xFFC0
移 3 位	0x0400	0x0800	0x0FFF	0x1FFE	0x3FFC	0x7FF8	0xFFF0	0xFFE0
移 4 位	0x0200	0x0400	0x0800	0x0FFF	0x1FFE	0x3FFC	0x7FF8	0xFFF0
移 5 位	0x0100	0x0200	0x0400	0x0800	0x0FFF	0x1FFE	0x3FFC	0x7FF8
移 6 位	0x0080	0x0100	0x0200	0x0400	0x0800	0x0FFF	0x1FFE	0x3FFC
移 7 位	0x0040	0x0080	0x0100	0x0200	0x0400	0x0800	0x0FFF	0x1FFE
移 8 位	0x020	0x0040	0x0080	0x0100	0x0200	0x0400	0x0800	0x0FFF

普通通道过采样被打断后的恢复方式

普通通道过采样中途被抢占通道转换打断后的恢复方式由 OOSRSEL 设定

- OOSRSEL=0：接续模式。保留已累加的数据，再次开始转换时将从打断处转换；
- OOSRSEL=1：重转模式。累加的数据被清空，再次开始转换时重新开始该通道的过采样转换。

图 7. 普通过采样被打断后的恢复方式



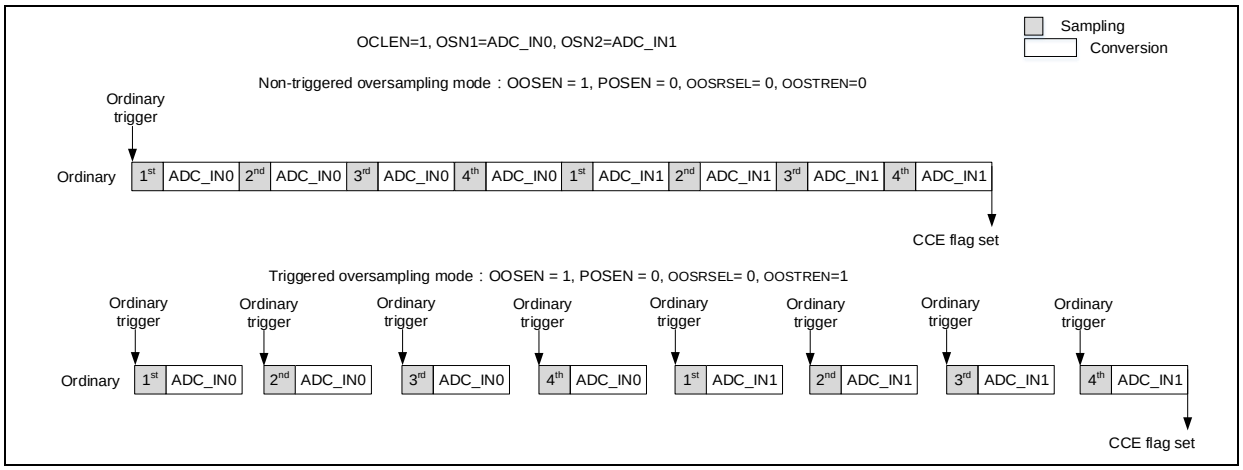
普通通道过采样触发模式

普通通道过采样的触发模式由 OOSTREN 设定

- OOSTREN=0：关闭触发模式。通道的所有过采样转换仅需一次触发；
- OOSTREN=1：开启触发模式。通道的每个过采样转换均需进行触发。

此模式下，中途被抢占通道触发打断后，须重新触发普通通道才会恢复转换普通通道过采样。

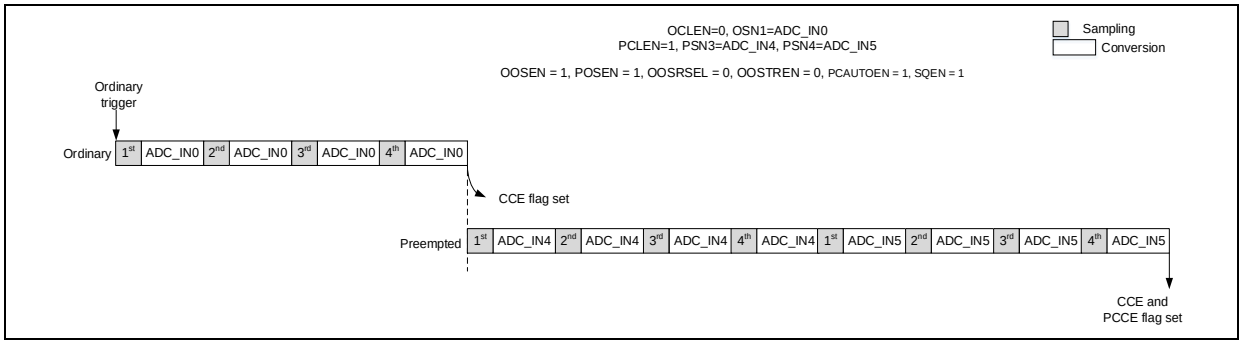
图 8. 普通过采样触发模式



抢占通道过采样

抢占过采样可与普通过采样同时使用，也可分别使用。抢占过采样不影响到普通过采样的各种模式。

图 9. 抢占自动转换下的过采样模式



2.8.2 软件接口

过采样率、过采样移位及过采样使能设定，软件由单独的函数接口实现，其软件实例如下：

```
/* set oversampling ratio and shift */
adc_oversample_ratio_shift_set(ADC1, ADC_OVERSAMPLE_RATIO_8, ADC_OVERSAMPLE_SHIFT_3);

/* enable ordinary oversampling */
adc_ordinary_oversample_enable(ADC1, TRUE);

/* enable preempt oversampling */
adc_preempt_oversample_enable(ADC1, TRUE);
```

普通通道过采样被打断后的恢复方式设定，软件由单独的函数接口实现，其软件实例如下：

```
/* set ordinary oversample restart mode */
adc_ordinary_oversample_restart_set(ADC1, ADC_OVERSAMPLE_CONTINUE);
```

普通通道过采样触发模式设定，软件由单独的函数接口实现，其软件实例如下：

```
/* disable ordinary oversampling trigger mode */
adc_ordinary_oversample_trig_enable(ADC1, FALSE);
```

2.9 电压监测

2.9.1 功能介绍

ADC 具备电压监测功能。用以监控输入电压与设定阈值的关系。

当转换结果大于高边界 ADC_VMHB[11:0]寄存器或是小于低边界 ADC_VMLB[11:0]寄存器时，电压监测超出标志 VMOR 会置起。

透过 VMSGEN 选择对单一通道或是所有通道监测。对单一通道监测的话，由 VMCSEL 配置通道。

2.9.2 软件接口

监测单一通道，软件由单独的函数接口实现，其软件实例如下：

```
/* config voltage_monitoring */
adc_voltage_monitor_threshold_value_set(ADC1, 0x100, 0x000);
adc_voltage_monitor_single_channel_select(ADC1, ADC_CHANNEL_5);
adc_voltage_monitor_enable(ADC1, ADC_VMONITOR_SINGLE_ORDINARY_PREEMPT);
```

监测所有通道，软件由单独的函数接口实现，其软件实例如下：

```
/* config voltage_monitoring */
adc_voltage_monitor_threshold_value_set(ADC1, 0x100, 0x000);
adc_voltage_monitor_enable(ADC1, ADC_VMONITOR_ALL_ORDINARY_PREEMPT);
```

注意：

若使用过采样器，则是以 ADC_VMHB[15:0]与 ADC_VMLB[15:0]完整的 16 位寄存器与过采样数据作比较。

2.10 中断及状态事件

2.10.1 功能介绍

ADC 含有多种中断及状态标志。应用需要结合这些标志进行程序设计。

- **普通通道转换开始标志**(OCCS)

  指示普通通道转换开始，由软件对其自身写零清除，无产生中断能力。

- **抢占通道转换开始标志**(PCCS)

  指示抢占通道转换开始，由软件对其自身写零清除，无产生中断能力。

- **抢占通道组转换结束标志**(PCCE)

  指示抢占通道组转换完成，由软件对其自身写零清除，有产生中断能力。

  在抢占通道组转换完成后置位，通常应用使用此标志来读取抢占通道组的转换数据。

- **通道转换结束标志**(CCE)

  指示通道序列转换完成，由软件对其自身写零或读 ODT 寄存器清除，有产生中断能力。

  在普通/抢占通道序列转换完成后置位，应用可使用此标志来读取普通/抢占通道的转换数据。

注意：

普通通道数据寄存器只有一个，CCE 标志只会在序列转换完毕时置位，在多通道应用中，若通过查询 CCE 状态获取数据的话，每次只能获取序列的最后一个通道数据而造成数据丢失。因此普通通道多通道数据必须使用 DMA 方式获取。DMA 读取转换数据会同步清除 CCE 标志。

- **电压监测超出范围标志**(VMOR)

  指示通道电压超出设定阈值，由软件对其自身写零清除，有产生中断能力。

在 ADC 的通道转换数据超过设定阈值后置位，通常应用使用此标志来监控通道电压。

2.10.2 软件接口

中断使能设定，软件由单独的函数接口实现，其软件实例如下：

```
/* enable adc preempt channels conversion end interrupt */
adc_interrupt_enable(ADC1, ADC_PCCE_INT, TRUE);
```

标志状态获取，软件由单独的函数接口实现，其软件实例如下：

```
if(adc_flag_get(ADC1, ADC_VMOR_FLAG) != RESET)
```

标志状态清除，软件由单独的函数接口实现，其软件实例如下：

```
adc_flag_clear(ADC1, ADC_PCCS_FLAG);
```

2.11 多种转换数据的获取方式

2.11.1 功能介绍

ADC 具备多种转换数据的获取方式。不同通道类型可支持的数据获取方式不同。

- **CPU 读取抢占通道数据**

抢占通道不具备 DMA 能力，因此不管什么组合模式，抢占通道数据均由 CPU 读取抢占数据寄存器 x（ADC_PDTx）获得。

- **CPU 读取普通通道数据**（单通道）

这种方式只适用于普通通道数量为 1 的情况。软件设置 ADC_CTRL1 的 CCEIEN 位使能通道转换结束中断，普通通道数据由 CPU 读取普通通道数据寄存器（ADC_ODT）获得。

- **DMA 读取普通通道数据**

普通通道数据存储于 ADC 自己独立的数据寄存器中。软件设置 OCDMAEN 位让每次普通数据寄存器更新时产生 DMA 请求，DMA 在每次收到 DMA 请求时读取转换数据。

2.11.2 软件接口

CPU 读取抢占通道数据，软件由单独的函数接口实现，其软件实例如下：

```
if(adc_flag_get(ADC1, ADC_PCCE_FLAG) != RESET)
{
  adc_flag_clear(ADC1, ADC_PCCE_FLAG);
  adc1_preempt_valuetab[preempt_conversion_count][0] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_1);
  adc1_preempt_valuetab[preempt_conversion_count][1] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_2);
  adc1_preempt_valuetab[preempt_conversion_count][2] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_3);
  preempt_conversion_count++;
}
```

CPU 读取普通通道数据，软件由单独的函数接口实现，其软件实例如下：

```
while(adc_flag_get(ADC1, ADC_OCCE_FLAG) == RESET);
```

```
*(p_adc1_ordinary++) = adc_ordinary_conversion_data_get(ADC1);
```

DMA 读取普通通道数据，软件由单独的函数接口实现，其软件实例如下：

```
/* enable dma mode */
adc_dma_mode_enable(ADC1, TRUE);

dma_flexible_config(DMA1, FLEX_CHANNEL1, DMA_FLEXIBLE_ADC1);
dma_default_para_init(&dma_init_struct);
dma_init_struct.buffer_size = 3;
dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
…
dma_init(DMA1_CHANNEL1, &dma_init_struct);
dma_channel_enable(DMA1_CHANNEL1, TRUE);
```

注意：使用 CPU 读取普通转换数据时，为避免数据读取不及时，通道采样周期需要足够大。

3 ADC 配置解析

以下对 ADC 的配置流程及数据获取方法进行说明。

3.1 ADC 配置流程

ADC 的配置一般包括如下内容

- **外部触发源配置**

  ADC 外部触发源有 TMR、EXINT 或软件触发，其配置无特殊性，参考普通的 TMR 或 EXINT 配置即可。

注意：此处仅是触发源的配置，触发源的使能需在 ADC 全部配置完毕后才可进行。

- **DMA 配置使能**

  ADC 普通通道转换数据可通过 DMA 传输，若应用需要 DMA 传输时，需提前进行 DMA 的初始化配置，其配置无特殊性，参考普通的 DMA 配置即可。

- **开启 ADC 数字时钟**

  开启 ADC 数字时钟，允许进行相关功能配置。

- **ADC 分频**

  设定 ADC 模拟部分的时钟，其由 PCLK2 分频而来，可设定 2/4/6/8/12/16 中的任意一种分频。

- **Vintrv（非必需）**

  使能内部参考电压，其连接到 ADC1 的 CH17。

- **ADC 基础部分结构体配置**

  包括序列模式、反复模式、数据对齐、普通转换序列长度。

  - **序列模式**

    不论普通还是抢占组，只要配置有多个通道，就需要开启序列模式。

  - **反复模式**

    若应用需要周期性的触发转换时，就需要关闭反复模式，不然周期性的触发将变得无效。

    当应用不想周期性的触发，而期望单次触发后就不停的转换设定通道组时需开启反复模式。

  - **数据对齐**

    设定转换数据靠右或是靠左对齐放置于数据寄存器。

  - **普通转换序列长度**

    可设定 1~16 中的任何一个长度，指示单个普通序列包含的通道个数，需与实际普通通道序列个数一致。

- **普通通道配置**

  包含通道配置、触发配置、数据传输方式。

  - **通道配置**

    由转换顺序、通道值、采样周期的设定组成。其中不同顺序可配置相同通道值。

  - **触发配置**

    选择普通通道的触发源。

  - **数据传输方式**

    可设定 CPU 或 DMA 传输转换数据。

- **抢占通道配置**

包含通道个数、通道配置、触发配置。

- **通道个数**

  可设定 1~4 中的任何一个长度，指示单个抢占序列包含的通道个数，需与实际抢占通道序列个数一致。

- **通道配置**

  由转换顺序、通道值、采样周期的设定组成。其中不同顺序可配置相同通道值。

- **触发配置**

  设置抢占通道的触发源。

■ **特殊模式配置（非必需）**

- **分割模式**

  包括每次触发转换的普通通道个数、普通通道分割模式使能、抢占通道分割模式使能。

- **抢占自动转换模式**

  用于设定普通组转换结束后的抢占通道组自动转换使能。

■ **中断配置**

使能对应中断，包括通道转换结束中断、抢占通道组转换结束中断、电压检测超过范围中断中的一个或多个。

■ **ADC 上电**

使能 ADC 让 ADC 上电，由于上电需要稳定时间，因此 ADC 上电后需等待 t_{STAB} 后才可进行后续动作。

■ **ADC 校准**

为保障 ADC 转换数据准确，在 ADC 上电后需进行校准。其包含：

A/D 初始化校准、等待初始化校准完成、A/D 校准、等待校准完成。

至此，ADC 的初始化配置就算全部完成。随后，可通过软件或使能硬件触发源进行触发转换。

3.2 ADC 数据获取方法

ADC 支持多种数据获取方法，通常可概括为如下几种

■ **CPU 获取抢占通道数据**

抢占通道数据不具备 DMA 能力，只能透过 CPU 获取。推荐使用中断获取，方法如下

1) 抢占通道组转换结束中断使能；
2) 抢占通道组转换结束中断函数内将转换数据缓存进数组内；
3) 其他应用逻辑内透过数组内的数据进行数据的后续算法处理。

■ **CPU 读取普通通道数据**

425 CPU 方式读取普通通道数据仅支持普通通道数为 1 的情形。为保障数据读取的实时性，同样推荐使用中断获取，方法如下：

1) 通道转换结束中断使能；
2) 通道组转换结束中断函数内将转换数据缓存进数组内；
3) 其他应用逻辑内透过数组内的数据进行数据的后续算法处理。

- **DMA 读取普通通道数据**

  普通通道数据具备 DMA 能力。为避免软件耗时，可直接采用 DMA 读取转换数据，方法如下

  1) 初始化并使能 DMA；
  2) 使能 ADC 的 DMA 模式；
  3) 在 DMA 传输完成中断函数内获取 DMA 的 buffer 数据；
  4) 其他应用逻辑内透过 buffer 数据进行数据的后续算法处理。

4 ADC 案例

4.1 案例 ADC 侦测 Vref 电压

4.1.1 功能简介

实际应用场景中，受外部电路等因素影响，ADC 的模拟参考电压可能出现波动，导致 ADC 转换结果出现失真。当无法优化硬件时，可尝试如下解决办法。

ADC 通道 ADC1_IN17 连接到了 MCU 的内部参考电压，该内部参考电压是经 LDO 输出的稳定的 1.2V 电压值。故可通过 ADC1_IN17 的转换数据来反推当前的实际模拟参考电压值。再以获取的模拟参考电压进行 ADC 转换通道数据的计算。

本例将示例如何以 ADC1_IN17 的转换结果来反推当前的模拟参考电压值。

4.1.2 资源准备

1) 硬件环境:

   对应产品型号的 AT-START BOARD

2) 软件环境

   project\at_start_f4xx\examples\ adc\current_vref_value_check

4.1.3 软件设计

1) 配置流程
   - 配置用于普通通道数据传输的 DMA
   - ADC 相关配置及设定
   - 普通通道软触发
   - 获取转换数据反推当前 Vref 值
2) 代码介绍
   - DMA 配置函数代码

```
static void dma_config(void)
{
  dma_init_type dma_init_struct;
  crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);
  dma_reset(DMA1_CHANNEL1);

  dma_flexible_config(DMA1, FLEX_CHANNEL1, DMA_FLEXIBLE_ADC1);

  dma_default_para_init(&dma_init_struct);
  dma_init_struct.buffer_size = 1;
  dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
  dma_init_struct.memory_base_addr = (uint32_t)&adc1_ordinary_value;
  dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_HALFWORD;
  dma_init_struct.memory_inc_enable = FALSE;
  dma_init_struct.peripheral_base_addr = (uint32_t)&(ADC1->odt);
  dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
  dma_init_struct.peripheral_inc_enable = FALSE;
```

```
  dma_init_struct.priority = DMA_PRIORITY_HIGH;
  dma_init_struct.loop_mode_enable = TRUE;
  dma_init(DMA1_CHANNEL1, &dma_init_struct);

  dma_channel_enable(DMA1_CHANNEL1, TRUE);
}
```

■ ADC 配置函数代码

```
static void adc_config(void)
{
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  crm_adc_clock_div_set(CRM_ADC_DIV_6);

  adc_base_default_para_init(&adc_base_struct);
  adc_base_struct.sequence_mode = FALSE;
  adc_base_struct.repeat_mode = FALSE;
  adc_base_struct.data_align = ADC_RIGHT_ALIGNMENT;
  adc_base_struct.ordinary_channel_length = 1;
  adc_base_config(ADC1, &adc_base_struct);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_17, 1, ADC_SAMPLETIME_239_5);
  adc_ordinary_conversion_trigger_set(ADC1, ADC12_ORDINARY_TRIG_SOFTWARE, TRUE);
  adc_dma_mode_enable(ADC1, TRUE);
  adc_tempersensor_vintrv_enable(TRUE);

  adc_enable(ADC1, TRUE);
  adc_calibration_init(ADC1);
  while(adc_calibration_init_status_get(ADC1));
  adc_calibration_start(ADC1);
  while(adc_calibration_status_get(ADC1));
}
```

■ main 函数代码

```
int main(void)
{
  __IO uint32_t index = 0;
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
  system_clock_config();
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  uart_print_init(115200);
  dma_config();
  adc_config();
  printf("adc1_vref_check \r\n");
  while(1)
```

```
  {
    at32_led_on(LED2);
    delay_sec(1);
    adc_ordinary_software_trigger_enable(ADC1, TRUE);
    while(dma_flag_get(DMA1_FDT1_FLAG) == RESET);
    dma_flag_clear(DMA1_FDT1_FLAG);
    printf("vref_value = %f V\r\n", ((double)1.2 * 4095) / adc1_ordinary_value);
  }
}
```

4.2 案例 ADC EXINT 触发+分割模式

4.2.1 功能简介

本案例介绍了使用 EXINT LINE 边沿触发 ADC，并开启普通通道分割模式（抢占通道不开启分割模式）。

4.2.2 资源准备

1) 硬件环境:

   对应产品型号的 AT-START BOARD

2) 软件环境

   project\at_start_f4xx\examples\adc\exint_trigger_partitioned

4.2.3 软件设计

1) 配置流程

   - 配置 ADC 使用的 GPIO
   - EXINT 配置
   - 配置用于普通通道数据传输的 DMA
   - ADC 相关配置及设定
   - 使用外部信号触发 ADC 普通组和抢占组
   - 获取转换数据并打印

2) 代码介绍

   - GPIO 配置函数代码

```
static void gpio_config(void)
{
  gpio_init_type gpio_initstructure;
  crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_GPIOB_PERIPH_CLOCK, TRUE);

  gpio_default_para_init(&gpio_initstructure);
  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_4 | GPIO_PINS_5 | GPIO_PINS_6 | GPIO_PINS_7;
  gpio_init(GPIOA, &gpio_initstructure);
```

```
  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_0 | GPIO_PINS_1;
  gpio_init(GPIOB, &gpio_initstructure);
}
```

■ DMA 配置函数代码

```
static void dma_config(void)
{
  dma_init_type dma_init_struct;
  crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(DMA1_Channel1_IRQn, 0, 0);
  dma_reset(DMA1_CHANNEL1);

  dma_flexible_config(DMA1, FLEX_CHANNEL1, DMA_FLEXIBLE_ADC1);

  dma_default_para_init(&dma_init_struct);
  dma_init_struct.buffer_size = 9;
  dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
  dma_init_struct.memory_base_addr = (uint32_t)adc1_ordinary_valuetab;
  dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_HALFWORD;
  dma_init_struct.memory_inc_enable = TRUE;
  dma_init_struct.peripheral_base_addr = (uint32_t)&(ADC1->odt);
  dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
  dma_init_struct.peripheral_inc_enable = FALSE;
  dma_init_struct.priority = DMA_PRIORITY_HIGH;
  dma_init_struct.loop_mode_enable = FALSE;
  dma_init(DMA1_CHANNEL1, &dma_init_struct);

  dma_interrupt_enable(DMA1_CHANNEL1, DMA_FDT_INT, TRUE);
  dma_channel_enable(DMA1_CHANNEL1, TRUE);
}
```

■ EXINT 配置函数代码

```
static void exint_config(void)
{
  gpio_init_type gpio_initstructure;
  exint_init_type exint_init_struct;

  crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);

  gpio_default_para_init(&gpio_initstructure);
  gpio_initstructure.gpio_pull          = GPIO_PULL_DOWN;
  gpio_initstructure.gpio_mode          = GPIO_MODE_INPUT;
  gpio_initstructure.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
  gpio_initstructure.gpio_pins          = GPIO_PINS_11;
```

```
  gpio_init(GPIOA, &gpio_initstructure);
  gpio_initstructure.gpio_pull              = GPIO_PULL_DOWN;
  gpio_initstructure.gpio_mode              = GPIO_MODE_INPUT;
  gpio_initstructure.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
  gpio_initstructure.gpio_pins              = GPIO_PINS_15;
  gpio_init(GPIOA, &gpio_initstructure);

  scfg_exint_line_config(SCFG_PORT_SOURCE_GPIOA, SCFG_PINS_SOURCE11);
  scfg_exint_line_config(SCFG_PORT_SOURCE_GPIOA, SCFG_PINS_SOURCE15);

  exint_default_para_init(&exint_init_struct);
  exint_init_struct.line_enable = TRUE;
  exint_init_struct.line_mode = EXINT_LINE_EVENT;
  exint_init_struct.line_select = EXINT_LINE_11;
  exint_init_struct.line_polarity = EXINT_TRIGGER_RISING_EDGE;
  exint_init(&exint_init_struct);

  exint_init_struct.line_select = EXINT_LINE_15;
  exint_init(&exint_init_struct);
}
```

■ ADC 配置函数代码

```
static void adc_config(void)
{
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  crm_adc_clock_div_set(CRM_ADC_DIV_6);
  nvic_irq_enable(ADC1_IRQn, 0, 0);

  adc_base_default_para_init(&adc_base_struct);
  adc_base_struct.sequence_mode = TRUE;
  adc_base_struct.repeat_mode = FALSE;
  adc_base_struct.data_align = ADC_RIGHT_ALIGNMENT;
  adc_base_struct.ordinary_channel_length = 3;
  adc_base_config(ADC1, &adc_base_struct);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_239_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_239_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_239_5);
  adc_ordinary_conversion_trigger_set(ADC1, ADC12_ORDINARY_TRIG_EXINT11, TRUE);
  adc_dma_mode_enable(ADC1, TRUE);
  adc_ordinary_part_count_set(ADC1, 1);
  adc_ordinary_part_mode_enable(ADC1, TRUE);

  adc_preempt_channel_length_set(ADC1, 3);
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_239_5);
```

```
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_8, 2, ADC_SAMPLETIME_239_5);
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_9, 3, ADC_SAMPLETIME_239_5);
  adc_preempt_conversion_trigger_set(ADC1, ADC12_PREEMPT_TRIG_EXINT15, TRUE);
  adc_interrupt_enable(ADC1, ADC_PCCE_INT, TRUE);

  adc_enable(ADC1, TRUE);
  adc_calibration_init(ADC1);
  while(adc_calibration_init_status_get(ADC1));
  adc_calibration_start(ADC1);
  while(adc_calibration_status_get(ADC1));
}
```

■ 中断服务函数代码

```
/* 获取抢占通道数据*/
void ADC1_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_PCCE_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_PCCE_FLAG);
    if(preempt_trigger_count < 3)
    {
      adc1_preempt_valuetab[preempt_trigger_count][0] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_1);
      adc1_preempt_valuetab[preempt_trigger_count][1] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_2);
      adc1_preempt_valuetab[preempt_trigger_count][2] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_3);
      preempt_trigger_count++;
    }
  }
}

/* DMA1_Channel1 中断函数--清除 DMA 传输完成中断标志 */
void DMA1_Channel1_IRQHandler(void)
{
  if(dma_flag_get(DMA1_FDT1_FLAG) != RESET)
  {
    dma_flag_clear(DMA1_FDT1_FLAG);
    dma_trans_complete_flag = 1;
  }
}
```

■ main 函数代码

```
int main(void)
{
  __IO uint32_t index = 0;
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
```

```
  system_clock_config();
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  uart_print_init(115200);
  gpio_config();
  exint_config();
  dma_config();
  adc_config();
  printf("exint_trigger_partitioned \r\n");
  /*此处使用外部信号的边沿来触发 ADC。
外部信号连接 PC11 触发普通通道，每次触发转换一个普通通道，3 次触发完成一次普通通道序列的转换；
外部信号连接 PA15 触发抢占通道，一次触发即可完成抢占通道整个序列的转换*/
  while(dma_trans_complete_flag == 0);
  while(preempt_trigger_count != 3);
  for(index = 0; index < 3; index++)
  {
    printf("adc1_ordinary_valuetab[%d][0] = 0x%x\r\n",index, adc1_ordinary_valuetab[index][0]);
    printf("adc1_ordinary_valuetab[%d][1] = 0x%x\r\n",index, adc1_ordinary_valuetab[index][1]);
    printf("adc1_ordinary_valuetab[%d][2] = 0x%x\r\n",index, adc1_ordinary_valuetab[index][2]);
    printf("adc1_preempt_valuetab[%d][0] = 0x%x\r\n",index, adc1_preempt_valuetab[index][0]);
    printf("adc1_preempt_valuetab[%d][1] = 0x%x\r\n",index, adc1_preempt_valuetab[index][1]);
    printf("adc1_preempt_valuetab[%d][2] = 0x%x\r\n",index, adc1_preempt_valuetab[index][2]);
    printf("\r\n");
  }
  at32_led_on(LED2);
  while(1)
  {
  }
}
```

4.3 案例 ADC 过采样

4.3.1 功能简介

ADC 支持过采样功能，在一些要求转换数据准确性的场合，可以使用过采样来实现。

本例将同时使用普通及抢占通道组的过采样，并开启抢占自动转换模式。设定 8 倍过采样，右移 3bit。

4.3.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

普通通道　抢占通道

PA4	PA7
PA5	PB0
PA6	PB1

2) 软件环境

project\at_start_f4xx\examples\adc\ordinary_preempt_oversampling

4.3.3 软件设计

1) 配置流程

- 配置 ADC 使用的 GPIO
- 配置用于普通通道数据传输的 DMA
- ADC 相关配置设定
- 软件触发转换
- 获取转换数据

2) 代码介绍

- GPIO 配置函数代码

```
static void gpio_config(void)
{
  gpio_init_type gpio_initstructure;
  crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_GPIOB_PERIPH_CLOCK, TRUE);

  gpio_default_para_init(&gpio_initstructure);
  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_4 | GPIO_PINS_5 | GPIO_PINS_6 | GPIO_PINS_7;
  gpio_init(GPIOA, &gpio_initstructure);

  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_0 | GPIO_PINS_1;
  gpio_init(GPIOB, &gpio_initstructure);
}
```

- DMA 配置函数代码

```
static void dma_config(void)
{
  dma_init_type dma_init_struct;
  crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(DMA1_Channel1_IRQn, 0, 0);
  dma_reset(DMA1_CHANNEL1);

  dma_flexible_config(DMA1, FLEX_CHANNEL1, DMA_FLEXIBLE_ADC1);

  dma_default_para_init(&dma_init_struct);
  dma_init_struct.buffer_size = 15;
  dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
```

```
  dma_init_struct.memory_base_addr = (uint32_t)adc1_ordinary_valuetab;
  dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_HALFWORD;
  dma_init_struct.memory_inc_enable = TRUE;
  dma_init_struct.peripheral_base_addr = (uint32_t)&(ADC1->odt);
  dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
  dma_init_struct.peripheral_inc_enable = FALSE;
  dma_init_struct.priority = DMA_PRIORITY_HIGH;
  dma_init_struct.loop_mode_enable = FALSE;
  dma_init(DMA1_CHANNEL1, &dma_init_struct);

  dma_interrupt_enable(DMA1_CHANNEL1, DMA_FDT_INT, TRUE);
  dma_channel_enable(DMA1_CHANNEL1, TRUE);
}
```

■ ADC 配置函数代码

```
static void adc_config(void)
{
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  crm_adc_clock_div_set(CRM_ADC_DIV_6);
  nvic_irq_enable(ADC1_IRQn, 0, 0);

  adc_base_default_para_init(&adc_base_struct);
  adc_base_struct.sequence_mode = TRUE;
  adc_base_struct.repeat_mode = TRUE;
  adc_base_struct.data_align = ADC_RIGHT_ALIGNMENT;
  adc_base_struct.ordinary_channel_length = 3;
  adc_base_config(ADC1, &adc_base_struct);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_239_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_239_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_239_5);
  adc_ordinary_conversion_trigger_set(ADC1, ADC12_ORDINARY_TRIG_SOFTWARE, TRUE);
  adc_dma_mode_enable(ADC1, TRUE);

  adc_preempt_channel_length_set(ADC1, 3);
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_239_5);
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_8, 2, ADC_SAMPLETIME_239_5);
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_9, 3, ADC_SAMPLETIME_239_5);
  adc_preempt_conversion_trigger_set(ADC1, ADC12_PREEMPT_TRIG_TMR1CH2, TRUE);
  adc_preempt_auto_mode_enable(ADC1, TRUE);
  adc_interrupt_enable(ADC1, ADC_PCCE_INT, TRUE);

  /* set oversampling ratio and shift */
  adc_oversample_ratio_shift_set(ADC1, ADC_OVERSAMPLE_RATIO_8,
ADC_OVERSAMPLE_SHIFT_3);
```

```
  /* disable ordinary oversampling trigger mode */
  adc_ordinary_oversample_trig_enable(ADC1, FALSE);

  /* set ordinary oversample restart mode */
  adc_ordinary_oversample_restart_set(ADC1, ADC_OVERSAMPLE_CONTINUE);

  /* enable ordinary oversampling */
  adc_ordinary_oversample_enable(ADC1, TRUE);

  /* enable preempt oversampling */
  adc_preempt_oversample_enable(ADC1, TRUE);

  adc_enable(ADC1, TRUE);
  adc_calibration_init(ADC1);
  while(adc_calibration_init_status_get(ADC1));
  adc_calibration_start(ADC1);
  while(adc_calibration_status_get(ADC1));
}
```

■ 中断服务函数代码

```
/* 获取普通通道数据传输完成状态 */
void DMA1_Channel1_IRQHandler(void)
{
  if(dma_flag_get(DMA1_FDT1_FLAG) != RESET)
  {
    dma_flag_clear(DMA1_FDT1_FLAG);
    dma_trans_complete_flag = 1;
  }
}
/* 获取 ADC 的抢占通道转换数据 */
void ADC1_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_PCCE_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_PCCE_FLAG);
    if(preempt_conversion_count < 5)
    {
      adc1_preempt_valuetab[preempt_conversion_count][0] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_1);
      adc1_preempt_valuetab[preempt_conversion_count][1] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_2);
      adc1_preempt_valuetab[preempt_conversion_count][2] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_3);
      preempt_conversion_count++;
    }
    at32_led_toggle(LED3);
```

```
  }
}
```

■ main 函数代码

```
int main(void)
{
  __IO uint32_t index = 0;
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
  system_clock_config();
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  uart_print_init(115200);
  gpio_config();
  dma_config();
  adc_config();
  printf("software_trigger_repeat \r\n");
  adc_ordinary_software_trigger_enable(ADC1, TRUE);
  while(dma_trans_complete_flag == 0);
  for(index = 0; index < 5; index++)
  {
    printf("adc1_ordinary_valuetab[%d][0] = 0x%x\r\n", index, adc1_ordinary_valuetab[index][0]);
    printf("adc1_ordinary_valuetab[%d][1] = 0x%x\r\n", index, adc1_ordinary_valuetab[index][1]);
    printf("adc1_ordinary_valuetab[%d][2] = 0x%x\r\n", index, adc1_ordinary_valuetab[index][2]);
    printf("adc1_preempt_valuetab[%d][0] = 0x%x\r\n", index, adc1_preempt_valuetab[index][0]);
    printf("adc1_preempt_valuetab[%d][1] = 0x%x\r\n", index, adc1_preempt_valuetab[index][1]);
    printf("adc1_preempt_valuetab[%d][2] = 0x%x\r\n", index, adc1_preempt_valuetab[index][2]);
  }
  at32_led_on(LED2);
  while(1)
  {
  }
}
```

4.4 案例 ADC 软件触发+反复模式

4.4.1 功能简介

本案例介绍了使用软件触发方式触发 ADC，并开启反复模式。

4.4.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

2) 软件环境

project\at_start_f4xx\examples\adc\software_trigger_repeat

4.4.3 软件设计

1) 配置流程

- 配置 ADC 使用的 GPIO
- 配置用于普通通道数据传输的 DMA
- ADC 相关配置及设定
- 普通通道软触发
- 获取转换数据并打印

2) 代码介绍

- GPIO 配置函数代码

```
static void gpio_config(void)
{
  gpio_init_type gpio_initstructure;
  crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);

  gpio_default_para_init(&gpio_initstructure);
  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_4 | GPIO_PINS_5 | GPIO_PINS_6;
  gpio_init(GPIOA, &gpio_initstructure);
}
```

- DMA 配置函数代码

```
static void dma_config(void)
{
  dma_init_type dma_init_struct;
  crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(DMA1_Channel1_IRQn, 0, 0);
  dma_reset(DMA1_CHANNEL1);

  dma_flexible_config(DMA1, FLEX_CHANNEL1, DMA_FLEXIBLE_ADC1);

  dma_default_para_init(&dma_init_struct);
  dma_init_struct.buffer_size = 30;
  dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
  dma_init_struct.memory_base_addr = (uint32_t)adc1_ordinary_valuetab;
  dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_HALFWORD;
  dma_init_struct.memory_inc_enable = TRUE;
  dma_init_struct.peripheral_base_addr = (uint32_t)&(ADC1->odt);
  dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
  dma_init_struct.peripheral_inc_enable = FALSE;
  dma_init_struct.priority = DMA_PRIORITY_HIGH;
  dma_init_struct.loop_mode_enable = FALSE;
  dma_init(DMA1_CHANNEL1, &dma_init_struct);

  dma_interrupt_enable(DMA1_CHANNEL1, DMA_FDT_INT, TRUE);
```

```
    dma_channel_enable(DMA1_CHANNEL1, TRUE);
}
```

■ ADC 配置函数代码

```
static void adc_config(void)
{
    adc_base_config_type adc_base_struct;
    crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
    crm_adc_clock_div_set(CRM_ADC_DIV_6);

    adc_base_default_para_init(&adc_base_struct);
    adc_base_struct.sequence_mode = TRUE;
    adc_base_struct.repeat_mode = TRUE;
    adc_base_struct.data_align = ADC_RIGHT_ALIGNMENT;
    adc_base_struct.ordinary_channel_length = 3;
    adc_base_config(ADC1, &adc_base_struct);
    adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_239_5);
    adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_239_5);
    adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_239_5);
    adc_ordinary_conversion_trigger_set(ADC1, ADC12_ORDINARY_TRIG_SOFTWARE, TRUE);
    adc_dma_mode_enable(ADC1, TRUE);

    adc_enable(ADC1, TRUE);
    adc_calibration_init(ADC1);
    while(adc_calibration_init_status_get(ADC1));
    adc_calibration_start(ADC1);
    while(adc_calibration_status_get(ADC1));
}
```

■ 中断服务函数代码

```
/* 获取普通通道数据传输完成状态 */
void DMA1_Channel1_IRQHandler(void)
{
    if(dma_flag_get(DMA1_FDT1_FLAG) != RESET)
    {
        dma_flag_clear(DMA1_FDT1_FLAG);
        dma_trans_complete_flag = 1;
    }
}
```

■ main 函数代码

```
int main(void)
{
    __IO uint32_t index = 0;
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
    system_clock_config();
    at32_board_init();
    at32_led_off(LED2);
```

```
  at32_led_off(LED3);
  at32_led_off(LED4);
  uart_print_init(115200);
  gpio_config();
  dma_config();
  adc_config();
  printf("software_trigger_repeat \r\n");
  adc_ordinary_software_trigger_enable(ADC1, TRUE);
  while(dma_trans_complete_flag == 0);
  for(index = 0; index < 10; index++)
  {
    printf("adc1_ordinary_valuetab[%d][0] = 0x%x\r\n", index, adc1_ordinary_valuetab[index][0]);
    printf("adc1_ordinary_valuetab[%d][1] = 0x%x\r\n", index, adc1_ordinary_valuetab[index][1]);
    printf("adc1_ordinary_valuetab[%d][2] = 0x%x\r\n", index, adc1_ordinary_valuetab[index][2]);
  }
  at32_led_on(LED2);
  while(1)
  {
  }
}
```

4.5 案例 ADC 定时器触发+抢占自动转换模式

4.5.1 功能简介

本案例介绍了使用定时器触发方式触发 ADC，并开启抢占通道自动转换模式。

4.5.2 资源准备

1) 硬件环境:

   对应产品型号的 AT-START BOARD

2) 软件环境

   project\at_start_f4xx\examples\adc\tmr_trigger_automatic_preempted

4.5.3 软件设计

1) 配置流程

   - 配置 ADC 使用的 GPIO
   - 配置用于触发的定时器
   - 配置用于普通通道数据传输的 DMA
   - ADC 相关配置及设定
   - 定时器使能，以触发 ADC
   - 获取转换数据并打印

2) 代码介绍

   - GPIO 配置函数代码

```
static void gpio_config(void)
```

```
{
  gpio_init_type gpio_initstructure;
  crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_GPIOB_PERIPH_CLOCK, TRUE);

  gpio_default_para_init(&gpio_initstructure);
  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_4 | GPIO_PINS_5 | GPIO_PINS_6 | GPIO_PINS_7;
  gpio_init(GPIOA, &gpio_initstructure);

  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_0 | GPIO_PINS_1;
  gpio_init(GPIOB, &gpio_initstructure);
}
```

■ DMA 配置函数代码

```
static void dma_config(void)
{
  dma_init_type dma_init_struct;
  crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(DMA1_Channel1_IRQn, 0, 0);
  dma_reset(DMA1_CHANNEL1);

  dma_flexible_config(DMA1, FLEX_CHANNEL1, DMA_FLEXIBLE_ADC1);

  dma_default_para_init(&dma_init_struct);
  dma_init_struct.buffer_size = 15;
  dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
  dma_init_struct.memory_base_addr = (uint32_t)adc1_ordinary_valuetab;
  dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_HALFWORD;
  dma_init_struct.memory_inc_enable = TRUE;
  dma_init_struct.peripheral_base_addr = (uint32_t)&(ADC1->odt);
  dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
  dma_init_struct.peripheral_inc_enable = FALSE;
  dma_init_struct.priority = DMA_PRIORITY_HIGH;
  dma_init_struct.loop_mode_enable = FALSE;
  dma_init(DMA1_CHANNEL1, &dma_init_struct);

  dma_interrupt_enable(DMA1_CHANNEL1, DMA_FDT_INT, TRUE);
  dma_channel_enable(DMA1_CHANNEL1, TRUE);
}
```

■ TMR 配置函数代码

```
static void tmr1_config(void)
{
  tmr_output_config_type tmr_oc_init_structure;
```

```
  crm_clocks_freq_type crm_clocks_freq_struct = {0};
  crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);

  /* get system clock */
  crm_clocks_freq_get(&crm_clocks_freq_struct);

  crm_periph_clock_enable(CRM_TMR1_PERIPH_CLOCK, TRUE);

  /* (systemclock/(systemclock/10000))/1000 = 10Hz(100ms) */
  tmr_base_init(TMR1, 999, (crm_clocks_freq_struct.sclk_freq/10000 - 1));
  tmr_cnt_dir_set(TMR1, TMR_COUNT_UP);
  tmr_clock_source_div_set(TMR1, TMR_CLOCK_DIV1);

  tmr_output_default_para_init(&tmr_oc_init_structure);
  tmr_oc_init_structure.oc_mode = TMR_OUTPUT_CONTROL_PWM_MODE_A;
  tmr_oc_init_structure.oc_polarity = TMR_OUTPUT_ACTIVE_LOW;
  tmr_oc_init_structure.oc_output_state = TRUE;
  tmr_oc_init_structure.oc_idle_state = FALSE;
  tmr_output_channel_config(TMR1, TMR_SELECT_CHANNEL_2, &tmr_oc_init_structure);
  tmr_output_channel_config(TMR1, TMR_SELECT_CHANNEL_4, &tmr_oc_init_structure);
  tmr_channel_value_set(TMR1, TMR_SELECT_CHANNEL_2, 500);
  tmr_channel_value_set(TMR1, TMR_SELECT_CHANNEL_4, 500);
}
```

- ADC 配置函数代码

```
static void adc_config(void)
{
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  crm_adc_clock_div_set(CRM_ADC_DIV_6);
  nvic_irq_enable(ADC1_IRQn, 0, 0);

  adc_base_default_para_init(&adc_base_struct);
  adc_base_struct.sequence_mode = TRUE;
  adc_base_struct.repeat_mode = FALSE;
  adc_base_struct.data_align = ADC_RIGHT_ALIGNMENT;
  adc_base_struct.ordinary_channel_length = 3;
  adc_base_config(ADC1, &adc_base_struct);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_239_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_239_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_239_5);
  adc_ordinary_conversion_trigger_set(ADC1, ADC12_ORDINARY_TRIG_TMR1CH4, TRUE);
  adc_dma_mode_enable(ADC1, TRUE);

  adc_preempt_channel_length_set(ADC1, 3);
```

```
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_239_5);
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_8, 2, ADC_SAMPLETIME_239_5);
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_9, 3, ADC_SAMPLETIME_239_5);
  adc_preempt_conversion_trigger_set(ADC1, ADC12_PREEMPT_TRIG_TMR1CH2, TRUE);
  adc_preempt_auto_mode_enable(ADC1, TRUE);
  adc_interrupt_enable(ADC1, ADC_PCCE_INT, TRUE);

  adc_enable(ADC1, TRUE);
  adc_calibration_init(ADC1);
  while(adc_calibration_init_status_get(ADC1));
  adc_calibration_start(ADC1);
  while(adc_calibration_status_get(ADC1));
}
```

■ 中断服务函数代码

```
/* 获取抢占通道数据*/
void ADC1_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_PCCE_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_PCCE_FLAG);
    if(preempt_conversion_count < 5)
    {
      adc1_preempt_valuetab[preempt_conversion_count][0] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_1);
      adc1_preempt_valuetab[preempt_conversion_count][1] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_2);
      adc1_preempt_valuetab[preempt_conversion_count][2] = adc_preempt_conversion_data_get(ADC1,
ADC_PREEMPT_CHANNEL_3);
      preempt_conversion_count++;
    }
    at32_led_toggle(LED3);
  }
}

/* DMA1_Channel1 中断函数--清除 DMA 传输完成中断标志 */
void DMA1_Channel1_IRQHandler(void)
{
  if(dma_flag_get(DMA1_FDT1_FLAG) != RESET)
  {
    dma_flag_clear(DMA1_FDT1_FLAG);
    dma_trans_complete_flag = 1;
  }
}
```

■ main 函数代码

```
int main(void)
```

```
{
  __IO uint32_t index = 0;
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
  system_clock_config();
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  uart_print_init(115200);
  gpio_config();
  tmr1_config();
  dma_config();
  adc_config();
  printf("tmr_trigger_automatic_preempted \r\n");
  tmr_counter_enable(TMR1, TRUE);
  tmr_channel_enable(TMR1, TMR_SELECT_CHANNEL_2, TRUE);
  tmr_channel_enable(TMR1, TMR_SELECT_CHANNEL_4, TRUE);
  tmr_output_enable(TMR1, TRUE);
  while(preempt_conversion_count < 5);
  while(dma_trans_complete_flag == 0);
  tmr_counter_enable(TMR1, FALSE);
  for(index = 0; index < 5; index++)
  {
    printf("adc1_ordinary_valuetab[%d][0] = 0x%x\r\n", index, adc1_ordinary_valuetab[index][0]);
    printf("adc1_ordinary_valuetab[%d][1] = 0x%x\r\n", index, adc1_ordinary_valuetab[index][1]);
    printf("adc1_ordinary_valuetab[%d][2] = 0x%x\r\n", index, adc1_ordinary_valuetab[index][2]);
    printf("adc1_preempted_valuetab[%d][0] = 0x%x\r\n", index, adc1_preempt_valuetab[index][0]);
    printf("adc1_preempted_valuetab[%d][1] = 0x%x\r\n", index, adc1_preempt_valuetab[index][1]);
    printf("adc1_preempted_valuetab[%d][2] = 0x%x\r\n", index, adc1_preempt_valuetab[index][2]);
    printf("\r\n");
  }
  at32_led_on(LED2);
  while(1)
  {
  }
}
```

4.6 案例 ADC 电压监测

4.6.1 功能简介

ADC 支持电压监测功能，在需要监测通道电压时，可参考本例进行设计。

本例将固定监控普通通道组的 Channel5，监控阈值为 0~Vref+/3。

4.6.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

PA4——3.3V

PA5——0V

PA6——1.5V 左右

2) 软件环境

project\at_start_f4xx\examples\adc\voltage_monitoring

4.6.3 软件设计

1) 配置流程

- 配置 ADC 使用的 GPIO
- 配置用于普通通道数据传输的 DMA
- ADC 相关配置及电压监测设定
- 普通通道软触发
- 获取转换数据

2) 代码介绍

- GPIO 配置函数代码

```
static void gpio_config(void)
{
  gpio_init_type gpio_initstructure;
  crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);

  gpio_default_para_init(&gpio_initstructure);
  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_4 | GPIO_PINS_5 | GPIO_PINS_6;
  gpio_init(GPIOA, &gpio_initstructure);
}
```

- DMA 配置函数代码

```
static void dma_config(void)
{
  dma_init_type dma_init_struct;
  crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);
  dma_reset(DMA1_CHANNEL1);

  dma_flexible_config(DMA1, FLEX_CHANNEL1, DMA_FLEXIBLE_ADC1);

  dma_default_para_init(&dma_init_struct);
  dma_init_struct.buffer_size = 3;
  dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
  dma_init_struct.memory_base_addr = (uint32_t)adc1_ordinary_valuetab;
  dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_HALFWORD;
```

```
  dma_init_struct.memory_inc_enable = TRUE;
  dma_init_struct.peripheral_base_addr = (uint32_t)&(ADC1->odt);
  dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
  dma_init_struct.peripheral_inc_enable = FALSE;
  dma_init_struct.priority = DMA_PRIORITY_HIGH;
  dma_init_struct.loop_mode_enable = TRUE;
  dma_init(DMA1_CHANNEL1, &dma_init_struct);

  dma_channel_enable(DMA1_CHANNEL1, TRUE);
}
```

■ ADC 配置函数代码

```
static void adc_config(void)
{
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  crm_adc_clock_div_set(CRM_ADC_DIV_6);
  nvic_irq_enable(ADC1_IRQn, 0, 0);

  adc_base_default_para_init(&adc_base_struct);
  adc_base_struct.sequence_mode = TRUE;
  adc_base_struct.repeat_mode = FALSE;
  adc_base_struct.data_align = ADC_RIGHT_ALIGNMENT;
  adc_base_struct.ordinary_channel_length = 3;
  adc_base_config(ADC1, &adc_base_struct);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_239_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_239_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_239_5);
  adc_ordinary_conversion_trigger_set(ADC1, ADC12_ORDINARY_TRIG_SOFTWARE, TRUE);
  adc_dma_mode_enable(ADC1, TRUE);
  adc_voltage_monitor_enable(ADC1, ADC_VMONITOR_SINGLE_ORDINARY);
  adc_voltage_monitor_threshold_value_set(ADC1, 0xBBB, 0xAAA);
  adc_voltage_monitor_single_channel_select(ADC1, ADC_CHANNEL_5);
  adc_interrupt_enable(ADC1, ADC_VMOR_INT, TRUE);

  adc_enable(ADC1, TRUE);
  adc_calibration_init(ADC1);
  while(adc_calibration_init_status_get(ADC1));
  adc_calibration_start(ADC1);
  while(adc_calibration_status_get(ADC1));
}
```

■ 中断服务函数代码

```
/*电压超出范围监测 */
void ADC1_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_VMOR_FLAG) != RESET)
```

```
  {
    at32_led_toggle(LED3);
    adc_flag_clear(ADC1, ADC_VMOR_FLAG);
    vmor_flag_index = 1;
  }
}
```

■ main 函数代码

```
int main(void)
{
  __IO uint32_t index = 0;
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
  system_clock_config();
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  uart_print_init(115200);
  gpio_config();
  dma_config();
  adc_config();
  printf("voltage_monitoring \r\n");
  while(1)
  {
    at32_led_toggle(LED2);
    delay_sec(1);
    if(vmor_flag_index == 1)
    {
      vmor_flag_index = 0;
      printf("out of range:adc1_channel_5 value is = %x!\r\n", adc1_ordinary_valuetab[1]);
    }
    adc_ordinary_software_trigger_enable(ADC1, TRUE);
  }
}
```

5 文档版本历史

表 3. 文档版本历史

日期	版本	变更
2022.7.19	2.0.0	最初版本
2023.06.19	2.0.1	文档格式修订