AN0093
入门指南

AT32F435/437 ADC使用指南

前言

AT32 的 ADC 是一个将模拟输入信号转换为设定分辨率数位数字信号的外设。采样率最高可达 5.33MSPS。多达 19 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	AT32F435 系列
	AT32F437 系列

目录

表目录

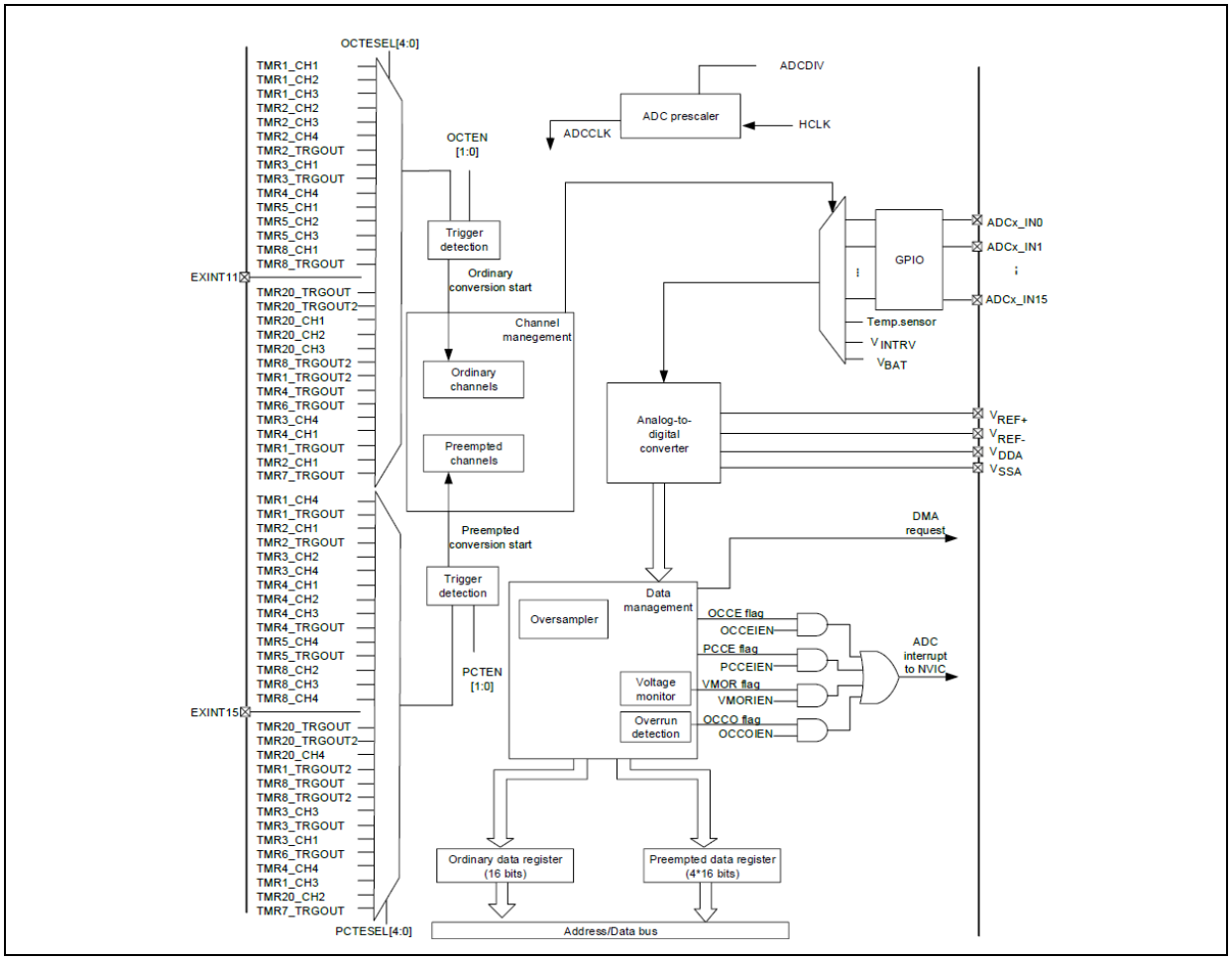
图目录

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
- 分辨率及采样转换，可配置分辨率为 12/10/8/6 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
- 转换中止，可软件控制在 ADC 不掉电状态下实现转换中止
- 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
- 联动多 ADC 的主从模式，可设定同时、交错、位移等多种组合模式，且支持单及双从机选择

图 1. ADC1 框图



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 ADC 公共部分结构体配置完成，其软件实例如下：

```
adc_common_struct.div = ADC_HCLK_DIV_4;
adc_common_config(&adc_common_struct);
```

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

注意：

1）模拟部分的 ADCCLK 由 HCLK 分频而来，其不可大于 80MHz；

2）ADC 数字部分挂在 PCLK2 上，为避免同步问题，ADCCLK 频率不可高于 PCLK2；

3）ADC1、ADC2、ADC3 都有自己独立的时钟使能位。ADC 公共部分无独立的时钟使能位，其会跟随任意 ADCx 时钟使能而自动打开；

4）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；

5）ADC 上电有一段等待时间，应用应该在判定到 ADC 的 RDY flag 置位后再执行后续触发等操作。

2.2 分辨率及采样转换

2.2.1 功能介绍

ADC 可随意设定 12、10、8、6 位分辨率使用。

ADC 可设定 2.5、6.5、12.5、24.5、47.5、92.5、247.5、640.5 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 + 分辨率位数 +0.5
```

示例：

CSPTx 选择 6.5 周期，CRSEL 选择 10 位，一次转换需要 6.5+10 + 0.5 = 17 个 ADCCLK 周期。

2.2.2 软件接口

ADC 分辨率设定，软件由单独的函数接口实现，其软件实例如下：

```
adc_resolution_set(ADC1,ADC_RESOLUTION_6B);
```

注意：ADC 的自校准只能在 12 位分辨率下进行，切分辨需安排在校准完成后执行。

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_12_5);
adc_preempt_channel_set(ADC1, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_47_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，为避免溢出，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。

校准值有两种获取方式：

- 软件下自校准命令，由硬件自动计算，产生的校准值保存在 ADC->CALVAL 寄存器内
- 软件直接根据经验值，手动设定校准值，该值同样被保存在 ADC->CALVAL 寄存器内

自校准的软件流程如下

- 在 12 位分辨率状态下使能 ADC
- 等待 ADC 的 RDY 标志置位
- 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
- 根据应用需求切换到期望配置的分辨率
- 等待 ADC 的 RDY 标志置位
- 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

完整的校准及设定分辨率需由组合命令实现，依据校准值设定方式可区分如下两种

自校准方式，其软件实例如下：

```
adc_resolution_set(ADC1,ADC_RESOLUTION_12B);
```

```
/* adc enable */
adc_enable(ADC1, TRUE);
while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);

/* adc calibration */
adc_calibration_init(ADC1);
while(adc_calibration_init_status_get(ADC1));
adc_calibration_start(ADC1);
while(adc_calibration_status_get(ADC1));

/*set resolution to 6bit.this because calibration must perform at 12 bit resolution */
adc_resolution_set(ADC1,ADC_RESOLUTION_6B);
while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);
```

写经验值校准方式，其软件实例如下：

```
adc_resolution_set(ADC1,ADC_RESOLUTION_12B);


/* adc enable */
adc_enable(ADC1, TRUE);
while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);

/* adc calibration */
adc_calibration_value_set(ADC1, 0x3F);

/*set resolution to 6bit.this because calibration must perform at 12 bit resolution */
adc_resolution_set(ADC1,ADC_RESOLUTION_6B);
while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);
```

注意：

校准值的存放不会置位 OCCE 标志，不会产生中断或 DMA 请求；

ADC 的自校准只能在 12 位分辨率下进行，切分辨需安排在校准完成后执行。

2.4 基本模式

2.4.1 功能介绍

序列模式

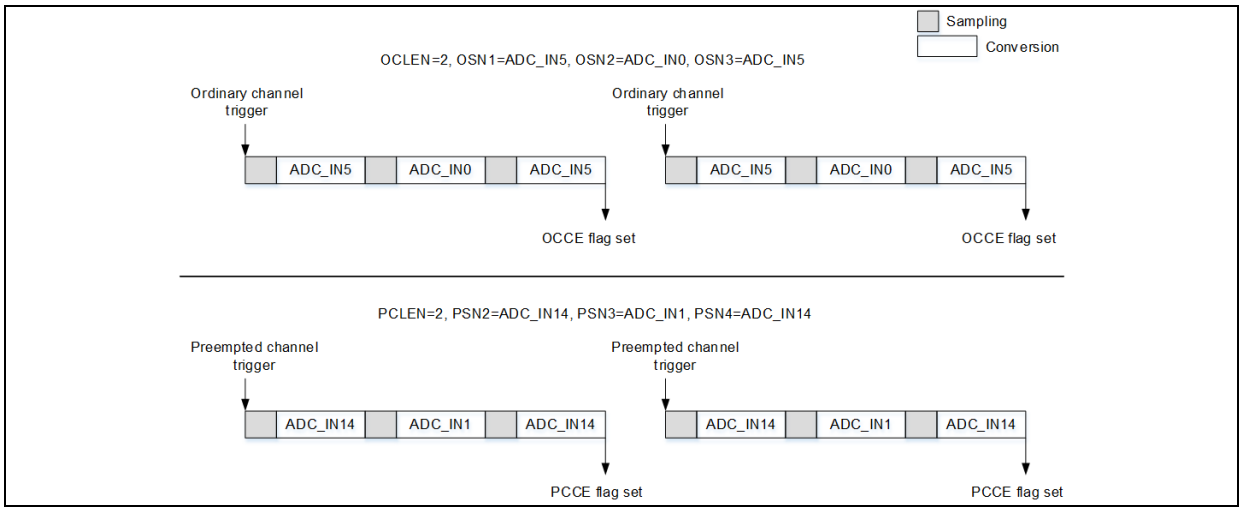
ADC 支持序列模式设定，开启序列模式后，每次触发将序列中的通道依序转换一次。

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

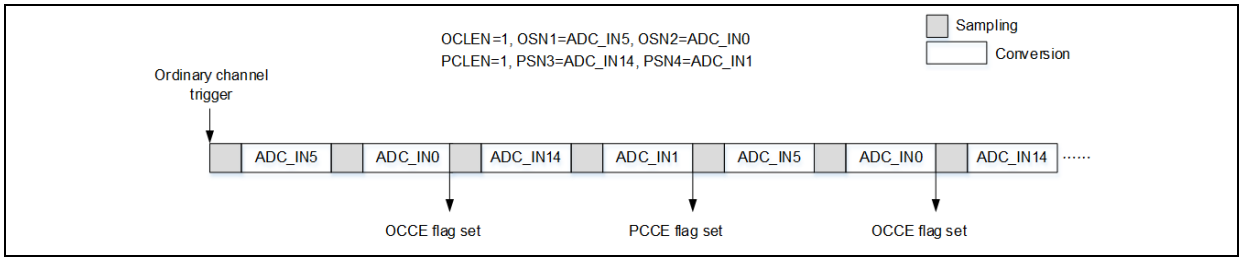
图 2. 序列模式



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



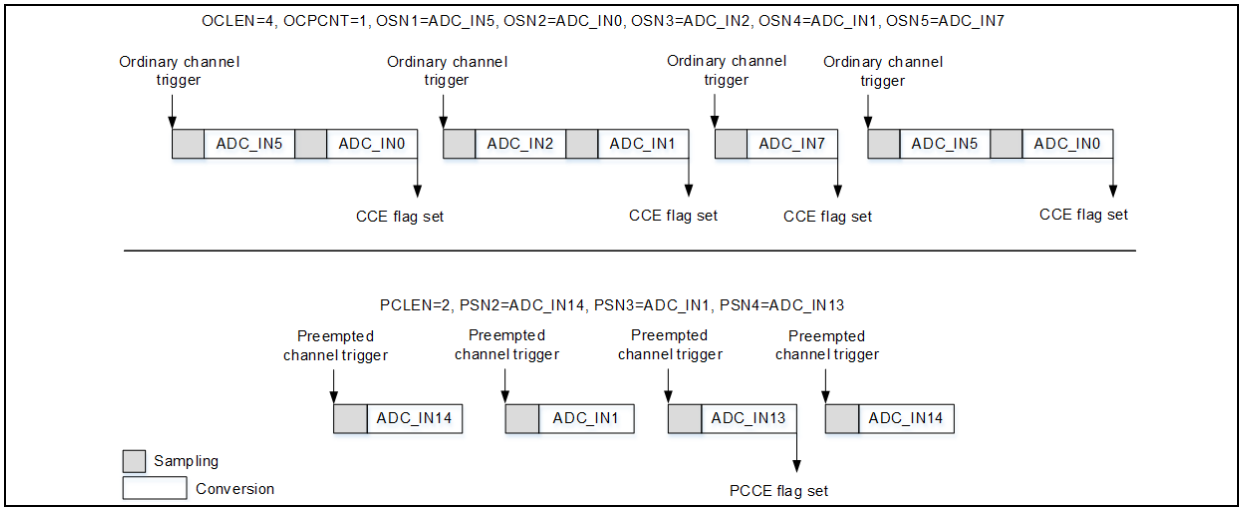
分割模式

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。

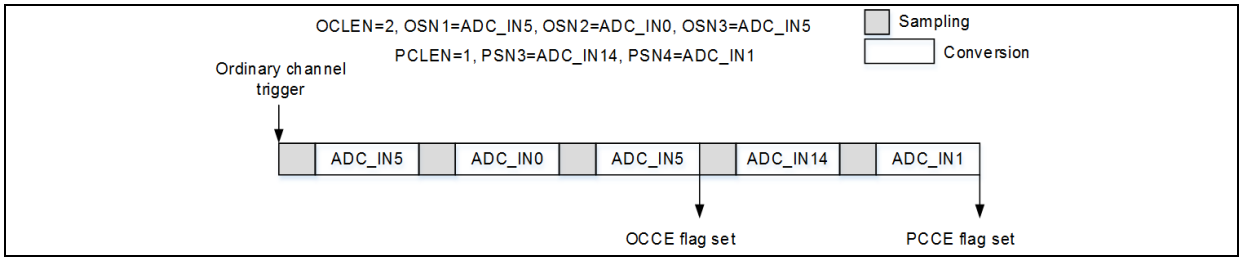
图 4. 分割模式



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，普通通道与抢占通道的分割模式不可共用。

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_47_5);
adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_47_5);
adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_47_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_47_5);
adc_preempt_channel_set(ADC1, ADC_CHANNEL_8, 2, ADC_SAMPLETIME_47_5);
adc_preempt_channel_set(ADC1, ADC_CHANNEL_9, 3, ADC_SAMPLETIME_47_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，外部触发可设定触发极性（触发极性可选择禁止边沿触发、上升沿触发、下降沿触发或任意边沿触发）。

表 1. 普通通道触发源

OCTESEL	触发来源	OCTESEL	触发来源
00000	TMR1_CH1 event	10000	TMR20_TRGOUT event
00001	TMR1_CH2 event	10001	TMR20_TRGOUT2 event
00010	TMR1_CH3 event	10010	TMR20_CH1 event
00011	TMR2_CH2 event	10011	TMR20_CH2 event
00100	TMR2_CH3 event	10100	TMR20_CH3 event
00101	TMR2_CH4 event	10101	TMR8_TRGOUT2 event
00110	TMR2_TRGOUT event	10110	TMR1_TRGOUT2 event
00111	TMR3_CH1 event	10111	TMR4_TRGOUT event
01000	TMR3_TRGOUT event	11000	TMR6_TRGOUT event
01001	TMR4_CH4 event	11001	TMR3_CH4 event
01010	TMR5_CH1 event	11010	TMR4_CH1 event
01011	TMR5_CH2 event	11011	TMR1_TRGOUT event
01100	TMR5_CH3 event	11100	TMR2_CH1 event
01101	TMR8_CH1 event	11101	保留
01110	TMR8_TRGOUT event	11110	TMR7_TRGOUT event
01111	EXINT line11 External pin	11111	保留

表 2. 抢占通道触发源

PCTESEL	触发来源	PCTESEL	触发来源
00000	TMR1_CH4 event	10000	TMR20_TRGOUT event
00001	TMR1_TRGOUT event	10001	TMR20_TRGOUT2 event
00010	TMR2_CH1 event	10010	TMR20_CH4 event
00011	TMR2_TRGOUT event	10011	TMR1_TRGOUT2 event
00100	TMR3_CH2 event	10100	TMR8_TRGOUT event
00101	TMR3_CH4 event	10101	TMR8_TRGOUT2 event
00110	TMR4_CH1 event	10110	TMR3_CH3 event
00111	TMR4_CH2 event	10111	TMR3_TRGOUT event
01000	TMR4_CH3 event	11000	TMR3_CH1 event
01001	TMR4_TRGOUT event	11001	TMR6_TRGOUT event
01010	TMR5_CH4 event	11010	TMR4_CH4 event
01011	TMR5_TRGOUT event	11011	TMR1_CH3 event
01100	TMR8_CH2 event	11100	TMR20_CH2 event
01101	TMR8_CH3 event	11101	保留
01110	TMR8_CH4 event	11110	TMR7_TRGOUT event
01111	EXINT line15 External pin	11111	保留

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 and trigger edge */
adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1CH1,
ADC_ORDINARY_TRIG_EDGE_NONE);
```

```
  /* config preempt trigger source and trigger edge */
  adc_preempt_conversion_trigger_set(ADC1, ADC_PREEMPT_TRIG_TMR1CH4,
ADC_PREEMPT_TRIG_EDGE_NONE);
```

在 ADC Ready 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 and trigger edge */
  adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1CH1,
ADC_ORDINARY_TRIG_EDGE_RISING);

  /* config preempt trigger source and trigger edge */
  adc_preempt_conversion_trigger_set(ADC1, ADC_PREEMPT_TRIG_TMR3CH4,
ADC_PREEMPT_TRIG_EDGE_RISING);
```

在 ADC Ready 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；

使用软件写寄存器触发时，对应触发极性必须选择禁止边沿触发；

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；

多 ADC 的主从模式下，需要将从 ADC 的触发极性选择为禁止边沿触发。

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。

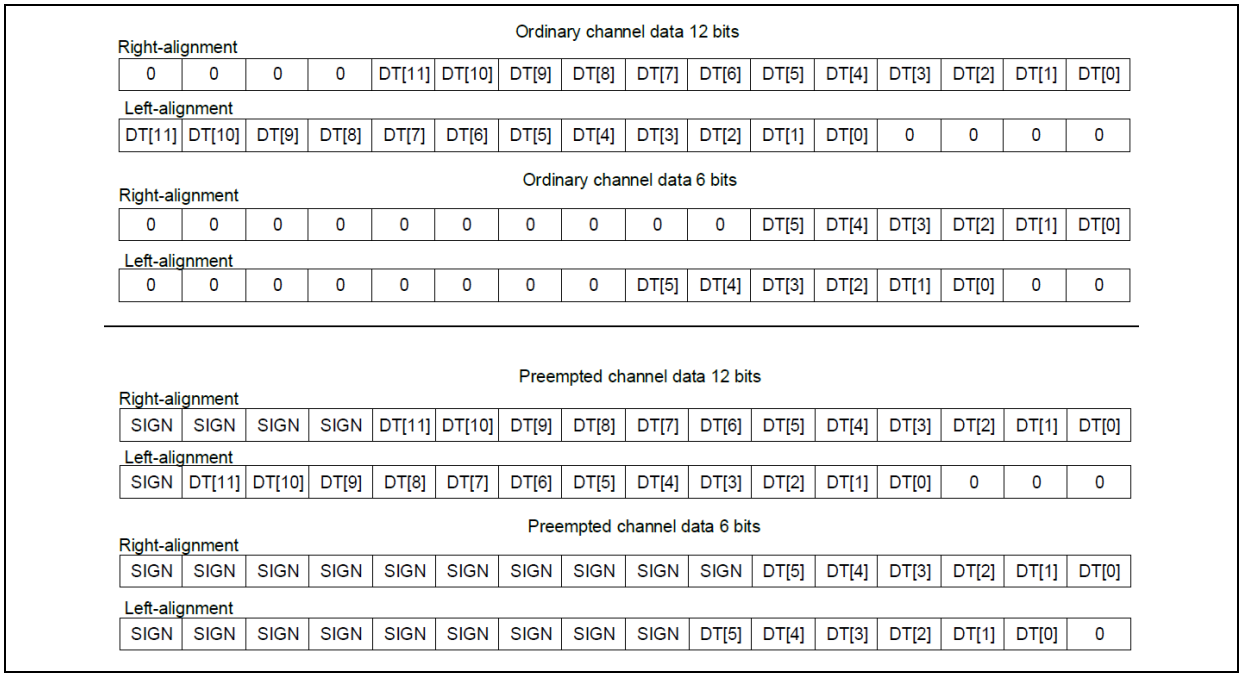
数据对齐

分左对齐和右对齐。分辨率 CRSEL 为 6 位时，数据存储方式以字节为基准摆放，其余皆以半字为基准摆放。

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：

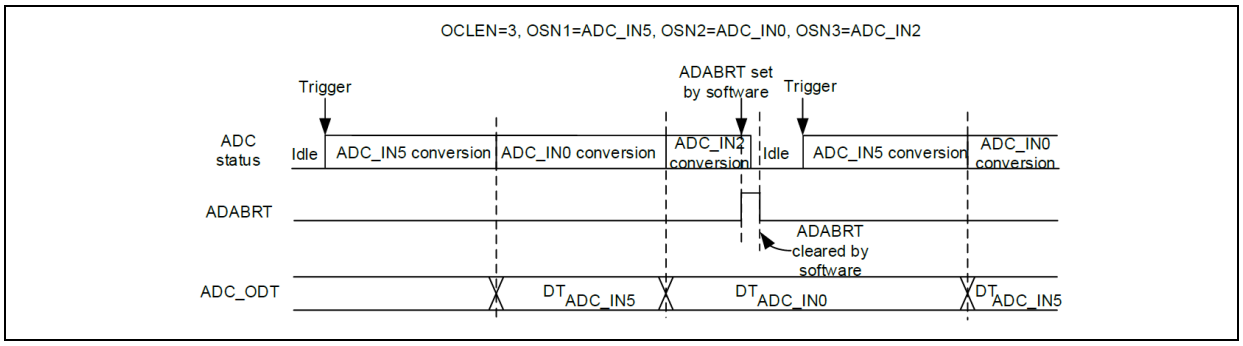
```
/* set preempt channel's conversion value offset */
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_1, 0x000);
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_2, 0x111);
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_3, 0x202);
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_4,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

2.8 转换中止

2.8.1 功能介绍

ADC 具备转换中止功能。在 ADC 转换过程中，用户可利用 ADC_CTRL2 的 ADABRT 使 ADC 停止转换。停止转换后，转换顺序回归第一个通道，用户即可重新编排通道顺序，再次触发 ADC 将从头开始执行新序列的转换。

图 7. 转换中止时序



2.8.2 软件接口

转换中止设定，软件由单独的函数接口实现，其软件实例如下：

```
adc_conversion_stop(ADC1);
while(adc_conversion_stop_status_get(ADC1));
```

注意：

转换中止命令对普通及抢占通道转换都有效；

转换中止功能类似 ADC 重上电，但是转换中止不会使 ADC 掉电，不存在 ADC 上电唤醒时间；

转换中止的执行需要时间，在执行转换中止命令后一定要等待 ADABRT 位被硬件清除后才可进行后续的触发转换。

2.9 过采样器

2.9.1 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 3. 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
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

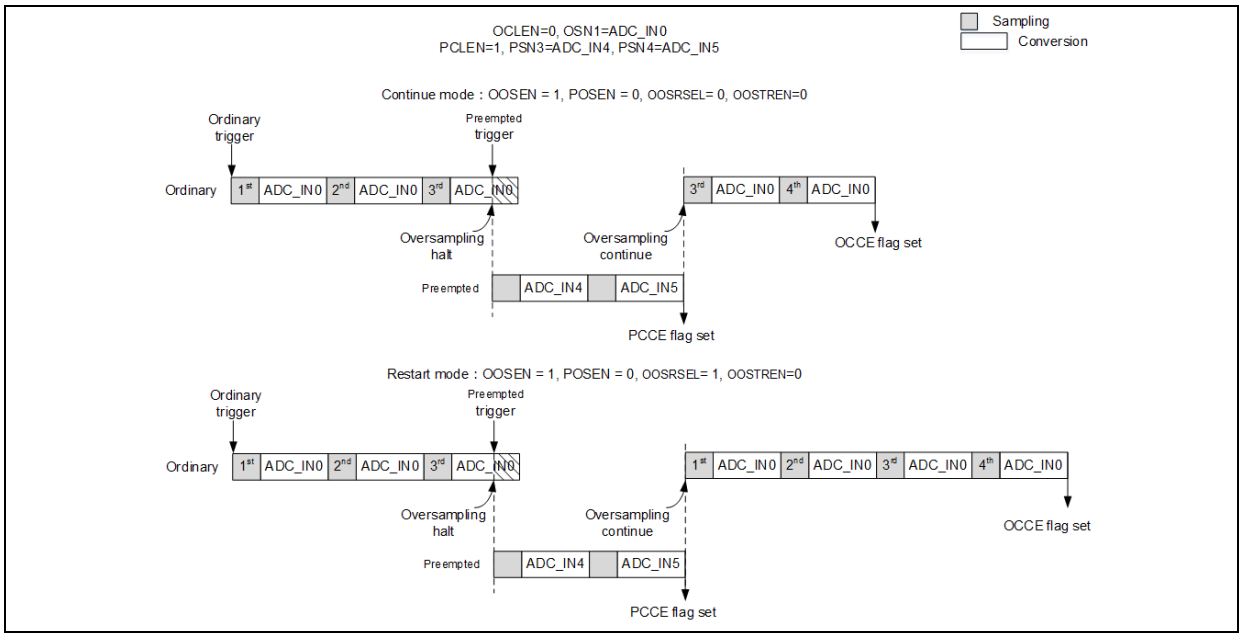
过采样率	2x	4x	8x	16x	32x	64x	128x	256x
移 7 位	0x0040	0x0080	0x0100	0x0200	0x0400	0x0800	0x0FFF	0x1FFE
移 8 位	0x020	0x0040	0x0080	0x0100	0x0200	0x0400	0x0800	0x0FFF

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 8. 普通过采样被打断后的恢复方式



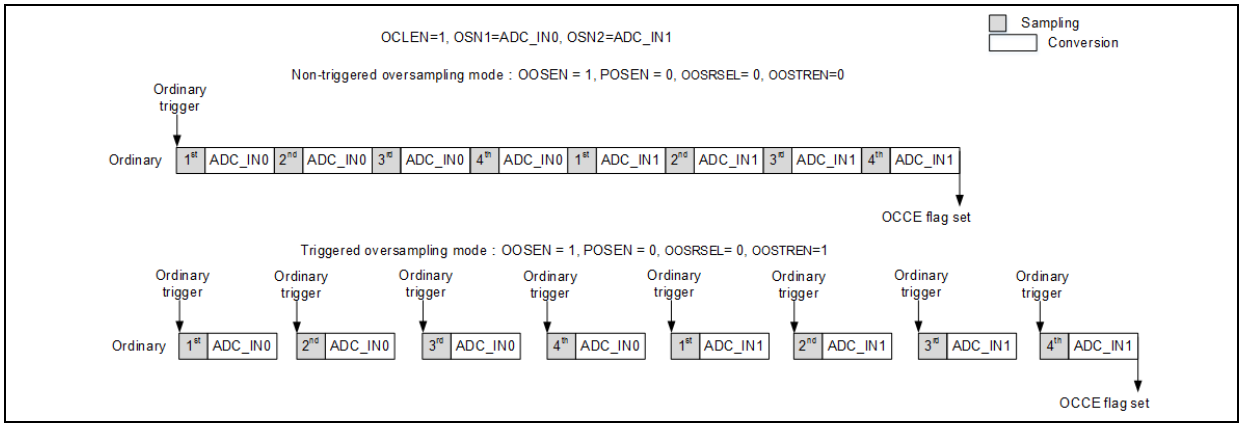
普通通道过采样触发模式

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。

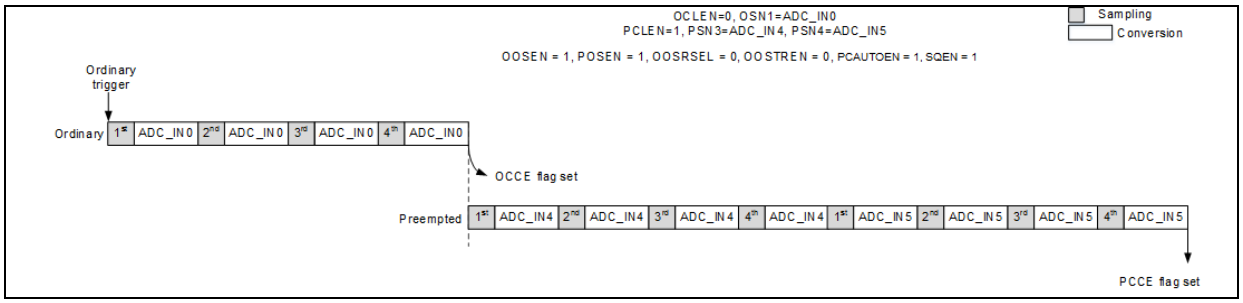
图 9. 普通过采样触发模式



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 10. 抢占自动转换下的过采样模式



2.9.2 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);
```

2.10 电压监测

2.10.1 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。

2.10.2 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：

电压监测一律以转换的原始数据与 12 位边界寄存器做比较，无视分辨率、抢占偏移量与数据对齐的设定；

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。

2.11 中断及状态事件

2.11.1 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。

- **ADC 准备就绪标志**(RDY)

  指示 ADC 状态，只读位，软件不可清除，无产生中断能力。

  ADC 上电完毕后会置位，只有 RDY 标志置位后，才可进行校准及触发转换。

- **普通通道转换溢出标志**(OCCO)

  指示 ADC 转换数据溢出，标志由软件对其自身写零清除，有产生中断能力。

  无独立的溢出检测使能位，在使能 DMA 传输或者 EOCSFEN =1 时有效，当上一笔转换数据未被读走，下一笔转换数据已产生时就会置位此标志。标志清除后的转换恢复分如下两种情况：

  1）非组合模式下，此标志置位后，当前转换停止，转换序列不被清零，因此可不用复位 ADC，直接清除标志再次触发转换即可；

  2）组合模式下，此标志置位后，当前转换停止，转换序列同样保持，但由于此时可能已丢失同步规则，因此需要复位各 ADC，然后重新进行触发转换。

- **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。

- **普通通道转换结束标志**(OCCE)

  指示普通通道转换完成，由软件对其自身写零或读 ODT 寄存器清除，有产生中断能力。

  在普通通道转换完成后置位，应用可使用此标志来读取普通通道的转换数据(EOCSFEN = 1)。

注意：DMA 读取转换数据会同步清除 OCCE 标志，因此在使用 DMA 时禁止再使用 OCCE 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。

2.11.2 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 overflow interrupt */
adc_interrupt_enable(ADC1, ADC_OCCO_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);
```

注意：

435 的三个 ADC 共用一个中断向量；

通用状态寄存器内有每个 ADC 的状态标志的映像，此映像为只读，不可通过映像来清除标志。

2.12 多种转换数据的获取方式

2.12.1 功能介绍

ADC 具备多种转换数据的获取方式。不同通道类型、不同组合模式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**(非组合模式)

只适用于非组合模式，软件设置 ADC_CTRL2 的 EOCSFEN 位让每次普通数据寄存器更新时置位 OCCE 标志，软件通过 OCCE 标志读取转换数据。

- **DMA 读取普通通道数据**(非组合模式)

非组合模式下，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 及 OCDRCEN 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。

- **DMA 读取普通通道数据**(组合模式)

组合模式下，普通通道数据会共同存储于通用普通数据寄存器中。存储方式透过 ADC_CCTRL 的 MSDMASEL 位配置，提供 5 种模式。只要 MSDMASEL 不为 0，就会在每次数据备齐时使用 ADC1 的 DMA 通道请求 DMA 搬运数据。

表 4. 主从组合模式的 DMA 模式

MSDMASEL	主从模式	DMA 请求	ADC_CODT[31:0]
001	单从机	第一次	16 位 0 , ADC1_ODT[15:0]
		第二次	16 位 0 , ADC2_ODT [15:0]
		第三次	16 位 0 , ADC1_ODT [15:0]
	双从机	第一次	16 位 0 , ADC1_ODT[15:0]
		第二次	16 位 0 , ADC2_ODT [15:0]
		第三次	16 位 0 , ADC3_ODT[15:0]
		第四次	16 位 0 , ADC1_ODT [15:0]
010	单从机	第一次	ADC2_ODT[15:0], ADC1_ODT[15:0]

MSDMASEL	主从模式	DMA 请求	ADC_CODT[31:0]
		第二次	ADC2_ODT[15:0], ADC1_ODT[15:0]
	双从机	第一次	ADC2_ODT[15:0], ADC1_ODT[15:0]
		第二次	ADC1_ODT[15:0], ADC3_ODT[15:0]
		第三次	ADC3_ODT[15:0], ADC2_ODT[15:0]
		第四次	ADC2_ODT[15:0], ADC1_ODT[15:0]
011	单从机	第一次	16 位 0 , ADC2_ODT[7:0], ADC1_ODT[7:0]
		第二次	16 位 0 , ADC2_ODT[7:0], ADC1_ODT[7:0]
	双从机	第一次	16 位 0 , ADC2_ODT[7:0], ADC1_ODT[7:0]
		第二次	16 位 0 , ADC1_ODT[7:0], ADC3_ODT[7:0]
		第三次	16 位 0 , ADC3_ODT[7:0], ADC2_ODT[7:0]
		第四次	16 位 0 , ADC2_ODT[7:0], ADC1_ODT[7:0]
100	双从机	第一次	8 位 0 ,ADC3_ODT[7:0], ADC2_ODT[7:0], ADC1_ODT[7:0]
		第二次	8 位 0 ,ADC3_ODT[7:0], ADC2_ODT[7:0], ADC1_ODT[7:0]
101	双从机	第一次	ADC2_ODT[15:0], ADC1_ODT[15:0]
		第二次	16 位 0 , ADC3_ODT[15:0]
		第三次	ADC2_ODT[15:0], ADC1_ODT[15:0]

图 11. DMA 模式与 ADC 模式对应关系

DMA_Mode	适用ADC模式	
	单从机	双从机
Mode1	普通同时模式、普通位移模式	普通同时模式、普通位移模式
Mode2	普通同时模式、普通位移模式	普通位移模式
Mode3	6/8位分辨率的普通同时模式、6/8位分辨率的普通位移模式	6/8位分辨率的普通位移模式
Mode4	无	6/8位分辨率的普通同时模式、6/8位分辨率的普通位移模式
Mode5	无	普通同时模式、普通位移模式

2.12.2 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(非组合模式) ，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(非组合模式) ，软件由单独的函数接口实现，其软件实例如下：

```
/* config dma mode,it's not useful when common dma mode is use */
adc_dma_mode_enable(ADC1, TRUE);

/* config dma request repeat,it's not useful when common dma mode is use */
adc_dma_request_repeat_enable(ADC1, FALSE);

dmamux_enable(DMA1, TRUE);
dmamux_init(DMA1MUX_CHANNEL1, DMAMUX_DMAREQ_ID_ADC1);
```

DMA 读取普通通道数据(组合模式) ，软件由单独的函数接口实现，其软件实例如下：

```
/* config common dma mode,it's useful for ordinary group in combine mode */
adc_common_struct.common_dma_mode = ADC_COMMON_DMAMODE_3;

/* config common dma request repeat */
adc_common_struct.common_dma_request_repeat_state = FALSE;

adc_common_config(&adc_common_struct);

dmamux_enable(DMA1, TRUE);
dmamux_init(DMA1MUX_CHANNEL1, DMAMUX_DMAREQ_ID_ADC1);
```

注意：

使用 CPU 读取普通转换数据时，为避免溢出，通道采样周期需要足够大；

三个 ADC 共用一个中断向量，故中断服务函数一定要简洁，以避免因中断无法及时响应而导致数据溢出；

组合模式下，不可使用 CPU 读取普通通道转换数据；

组合模式下，一定要按照可支持的对应关系选用 DMA 模式。

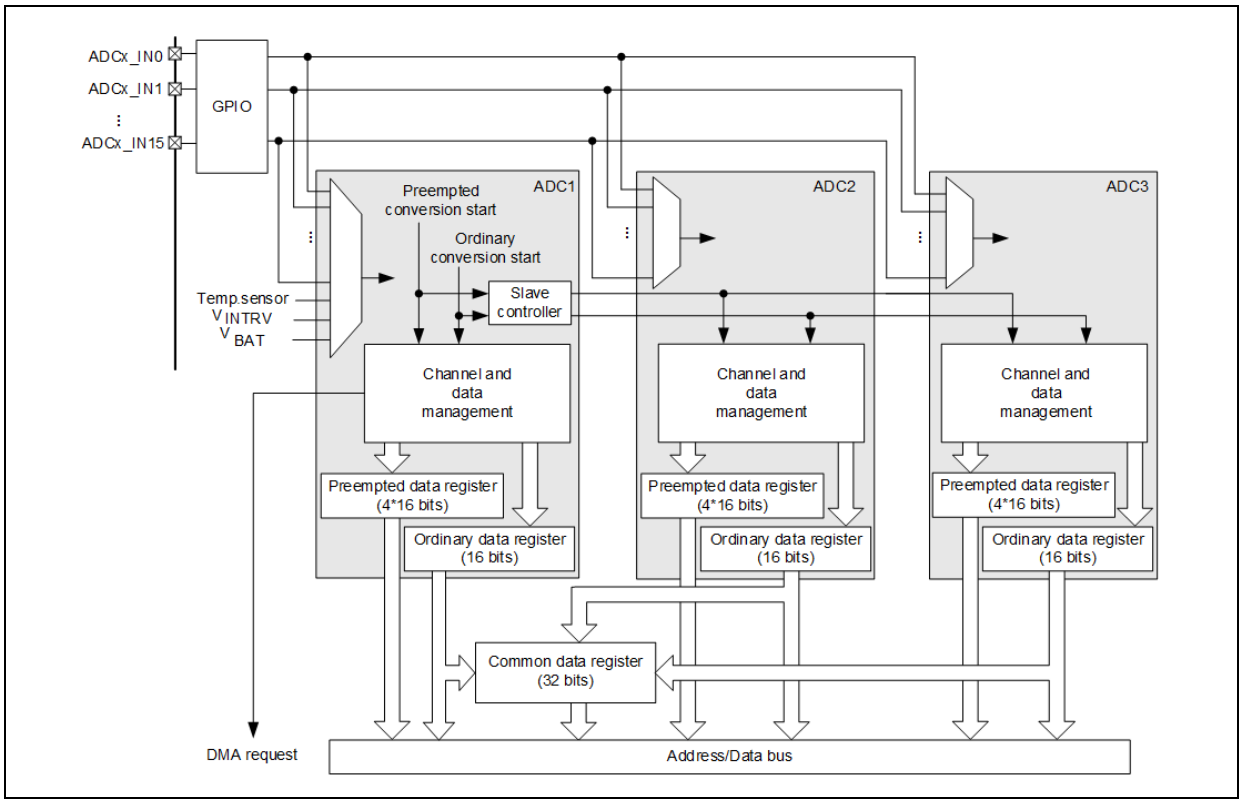
2.13 联动多 ADC 的主从模式

2.13.1 功能介绍

ADC 主从模式即通过触发主机来联动从机进行通道转换，并且将通用普通数据寄存器作为获取主从 ADC 普通通道数据的单一接口。

单从机主从模式以 ADC1 作为主机，ADC2 作为从机，ADC3 独立动作。双从机主从模式以 ADC1 作为主机，ADC2 与 ADC3 作为从机。

图 12. 主从模式的 ADC 框图



■ 同时模式

同时模式可用于普通/抢占/普通抢占组合。配置同时模式后，可触发主机，使主机与从机同时转换各自的通道。在此模式下，必须使用相同的采样时间以及相同的序列长度，以避免主从之间失去同步，遗失数据。

图 13. 普通同时模式

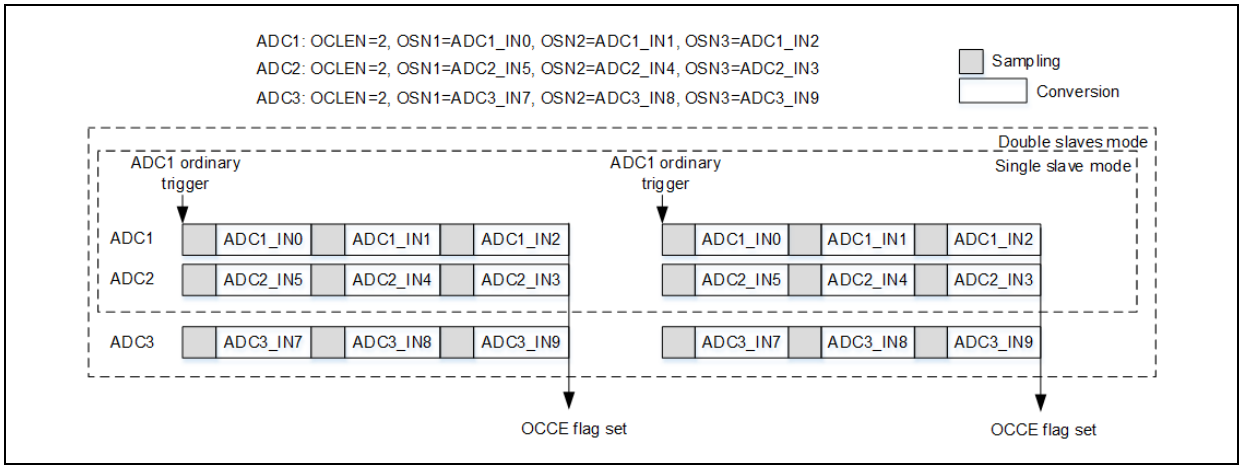
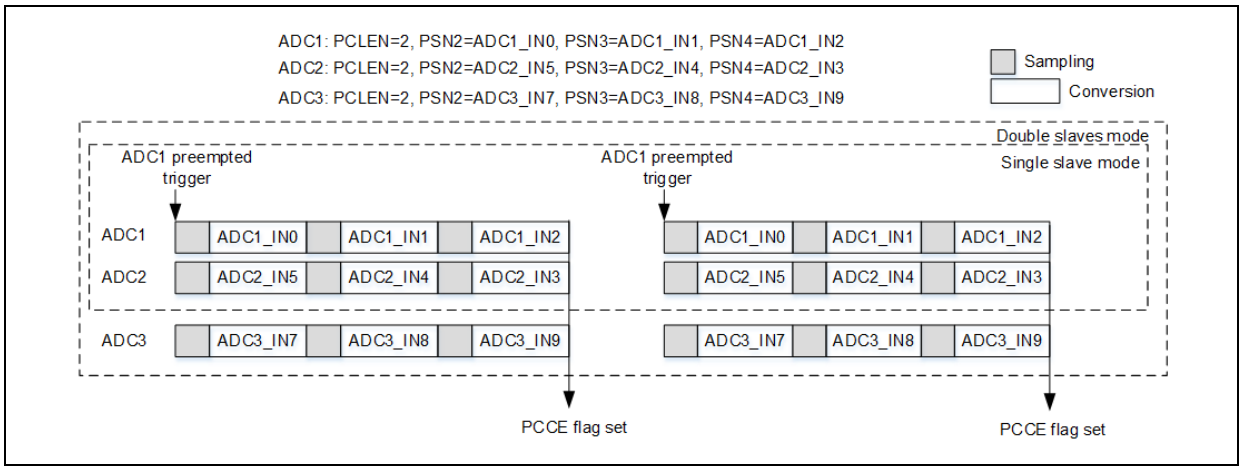


图 14. 抢占同时模式



注意：

同时模式下，需要禁止来自从机的触发；

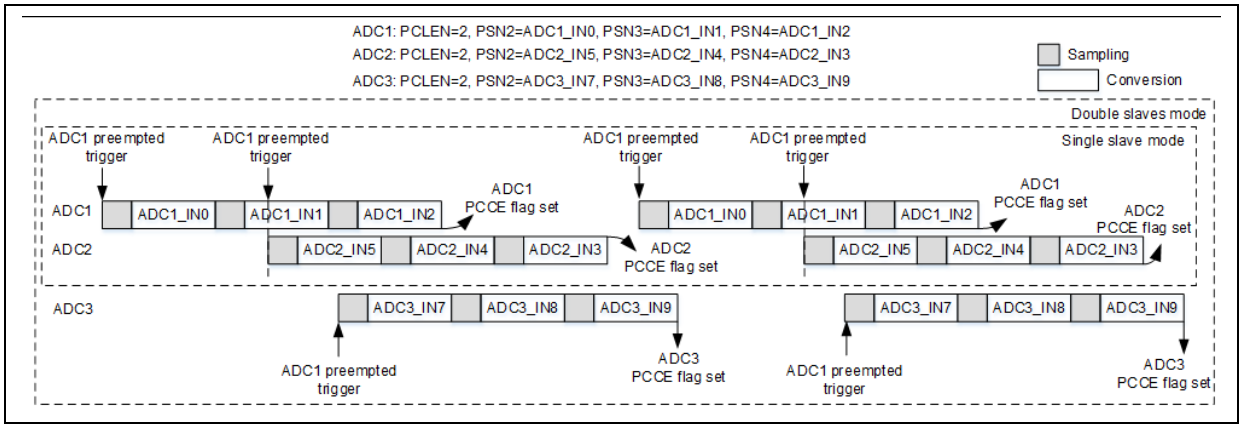
同时模式下，触发间隔需要大于任意通道组的转换周期；

同样的通道不可同时被多个 ADC 采样，因此禁止将相同通道安排在不同 ADC 的同样序列位置。

■ **交错触发模式**

交错触发模式适用于抢占通道组，可单独使用也可与普通同时模式组合使用。配置抢占交错触发模式后，可多次触发主机的抢占通道，促使主从 ADC 轮流转换抢占通道组。

图 15. 抢占交错触发模式



■ **位移模式**

位移模式适用于普通通道组，此模式只可单独使用，不能与抢占通道组合使用。配置普通位移模式后，可触发主机普通通道，使各 ADC 之间自动在普通通道的转换上时序位移。

位移长度可由软件经 ADC_CCTRL 寄存器中的 ASISEL 位进行设定。

此模式下，硬件会保证不同 ADC 间的采样间隔至少 2.5 个 ADCCLK。当软件设定的位移长度无法满足这个条件时，软件设定将会变得无效。因此利用这个特性，可以将相同通道安排在不同 ADC 的同样序列位置。

图 16. 普通位移模式

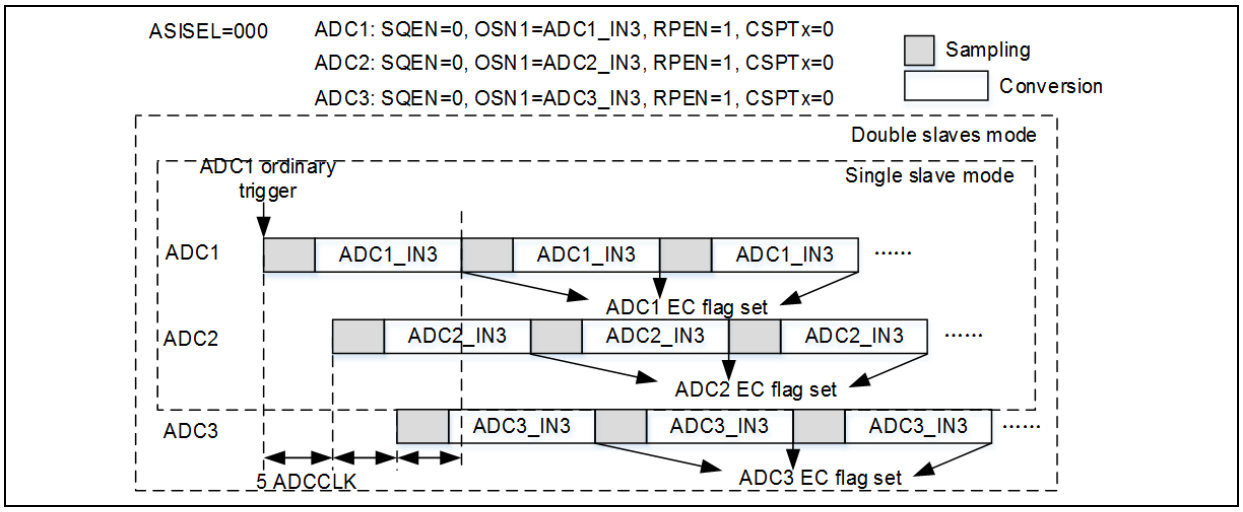
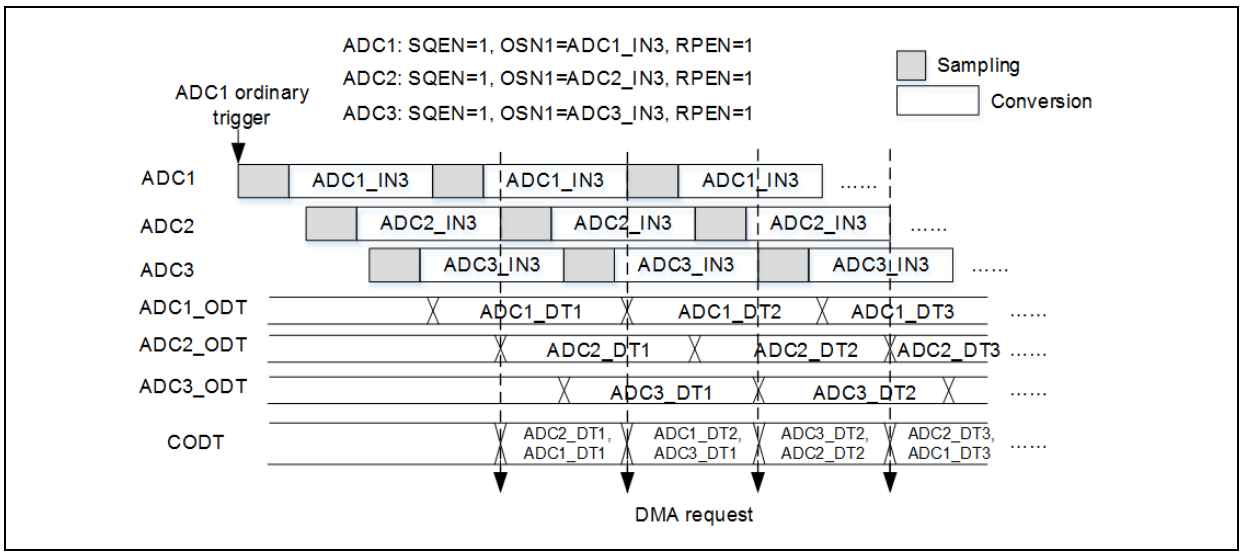


图 17. 使用 DMA 模式 2 时的位移转换



注意：

位移模式下禁止抢占通道触发、禁止从机的普通通道触发

2.13.2 软件接口

联动多 ADC 的主从模式设定，软件由公共部分结构体配置完成，其软件实例如下：

```
/* config combine mode */
adc_common_struct.combine_mode = ADC_ORDINARY_SHIFT_ONLY_TWOSLAVE_MODE;
adc_common_config(&adc_common_struct);
```

注意：

为避免主从间失去同步，主从机必须配置相同分辨率；

组合模式下，一定要按照可支持的对应关系选用 DMA 模式；

若有同时使用多个 ADC 低解析度转换的需求，建议使用主从模式搭配 DMA1 或 DMA2。

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 或 EXINT，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 公共部分结构体配置**

  包括主从模式、ADC 分频、主从模式的 DMA 模式、DMA 请求接续使能、位移模式位移长度、内部温度传感器及 Vintrv、Vbat 相关配置。

  - **主从模式**

    依据从机数量、通道类型、组合类型进行选择，支持多达 13 种模式。

  - **ADC 分频**

    设定 ADC 模拟部分的时钟，其由 HCLK 分频而来，可设定 2~17 中的任意一种分频。

  - **主从模式的 DMA 模式**

    需要结合主从模式进行选择，支持多达 5 种模式，抢占组合模式下需禁止 DMA 模式。

  - **DMA 请求接续使能**

    设定在传输完 DMA 设定个数数据后，普通通道转换完毕是否再产生 DMA 请求。

  - **位移模式位移长度**

    设定位移模式下相邻 ADC 间的采样间隔时间，可设定 5~20 个 ADCCLK 中的任意一种。

  - **内部温度传感器及 Vintrv**

    使能内部温度传感器及内部参考电压，其分别连接到 ADC1 的 CH16 和 CH17。

  - **Vbat**

    使能电池电压，电池电压经除 4 电路后连接到了 ADC1 的 CH18。

    注意：*用 ADC 采集 VBAT 电压，当使能 VBATEN 后，VBAT 会一直耗电，如果应用对功耗敏感，建议只在 VBAT 采样时才打开，采样结束后关闭。*

- **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**

  由触发源和触发边沿检测的设定组成。若使用软件触发时，需要禁止触发边沿检测。

- **数据传输方式**

  可设定 CPU 或 DMA 传输转换数据。在主从组合模式下，需禁止此项配置。

- **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**

  由触发源和触发边沿检测的设定组成。若使用软件触发时，需要禁止触发边沿检测。

- **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。

- **过采样**

  包括过采样率、过采样移位、普通过采样触发模式使能、普通过采样重转模式选择、普通及抢占过采样使能的设定。

- **中断配置**

使能对应中断，包括溢出中断、普通通道转换结束中断、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 ADC 的 RDY 标志处于置位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。

- **ADC 分辨率调整**

  在 ADC 校准完毕后，即可进行分辨率切换。切换分辨率后，软件需要等待 ADC 的 RDY 标志处于置位后才可进行后续动作。

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**

  435 额外支持通过 CPU 读取普通通道数据。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) 软件设置 ADC_CTRL2 的 EOCSFEN 位让每次数据寄存器更新时置位 OCCE 标志；
  2) 普通通道组转换结束中断使能；
  3) 普通通道组转换结束中断函数内将转换数据缓存进数组内；
  4) 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（单 buffer 模式）**

  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。

- **DMA 读取普通通道数据（双 buffer 模式）**

  435 的 EDMA 支持双 buffer 模式。为避免数据处理不及时，可尝试使用双 buffer 缓存数据，方法如下

  1) 初始化 EDMA 成双 buffer 模式并使能 EDMA；
  2) 使能 ADC 的 DMA 模式，使能普通通道数据的 DMA 请求接续；
  3) 在 EDMA 传输完成中断函数内交叉获取 EDMA 的 buffer 数据；
  4) 其他应用逻辑内透过获取到的 buffer 数据进行数据的后续算法处理。

4 案例 ADC 过采样

4.1 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。设定 8 倍过采样，保持 12 位分辨率。

4.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

PA4 and PA7——3.3V

PA5 and PB0——GND

PA6 and PB1——1.5V 左右

2) 软件环境

project\at_start_f4xx\examples\adc\ordinary_preempt_oversampling

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);

  /* config adc pin as analog input mode */
  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_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);

  dmamux_enable(DMA1, TRUE);
  dmamux_init(DMA1MUX_CHANNEL1, DMAMUX_DMAREQ_ID_ADC1);

  /* enable dma transfer complete interrupt */
  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_common_config_type adc_common_struct;
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(ADC1_2_3_IRQn, 0, 0);

  adc_common_default_para_init(&adc_common_struct);

  /* config combine mode */
  adc_common_struct.combine_mode = ADC_INDEPENDENT_MODE;

  /* config division,adcclk is division by hclk */
  adc_common_struct.div = ADC_HCLK_DIV_4;

  /* config common dma mode,it's not useful in independent mode */
  adc_common_struct.common_dma_mode = ADC_COMMON_DMAMODE_DISABLE;
```

```
  /* config common dma request repeat */
  adc_common_struct.common_dma_request_repeat_state = FALSE;

  /* config adjacent adc sampling interval,it's useful for ordinary shifting mode */
  adc_common_struct.sampling_interval = ADC_SAMPLING_INTERVAL_5CYCLES;

  /* config inner temperature sensor and vintrv */
  adc_common_struct.tempervintrv_state = FALSE;

  /* config voltage battery */
  adc_common_struct.vbat_state = FALSE;
  adc_common_config(&adc_common_struct);

  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_resolution_set(ADC1, ADC_RESOLUTION_12B);

  /* config ordinary channel */
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_6_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_6_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_6_5);

  /* config ordinary trigger source and trigger edge */
  adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1CH1,
ADC_ORDINARY_TRIG_EDGE_NONE);

  /* config dma mode,it's not useful when common dma mode is use */
  adc_dma_mode_enable(ADC1, TRUE);

  /* config dma request repeat,it's not useful when common dma mode is use */
  adc_dma_request_repeat_enable(ADC1, FALSE);

  /* config preempt channel */
  adc_preempt_channel_length_set(ADC1, 3);
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_6_5);
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_8, 2, ADC_SAMPLETIME_6_5);
  adc_preempt_channel_set(ADC1, ADC_CHANNEL_9, 3, ADC_SAMPLETIME_6_5);

  /* config preempt trigger source and trigger edge */
  adc_preempt_conversion_trigger_set(ADC1, ADC_PREEMPT_TRIG_TMR1CH4,
ADC_PREEMPT_TRIG_EDGE_NONE);
```

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

  /* enable adc overflow interrupt */
  adc_interrupt_enable(ADC1, ADC_OCCO_INT, TRUE);

  /* enable adc preempt channels conversion end interrupt */
  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 */
  adc_enable(ADC1, TRUE);
  while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);

  /* adc calibration */
  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);
    dma1_trans_complete_flag = 1;
  }
```

```
}
/* 获取 ADC 的溢出状态信息及抢占通道转换数据 */
void ADC1_2_3_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_OCCO_FLAG);
    adc1_overflow_flag++;
  }
  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++;
    }
  }
}
```

■ main 函数代码

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

  /* config the system clock */
  system_clock_config();

  /* init at start board */
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  usart1_config(115200);
  gpio_config();
  dma_config();
  adc_config();
  printf("ordinary_preempt_oversampling \r\n");

  /* adc1 software trigger start conversion */
```

```
  for(index = 0; index < 5; index++)
  {
    adc_ordinary_software_trigger_enable(ADC1, TRUE);
    adc_preempt_software_trigger_enable(ADC1, TRUE);
    delay_sec(1);
  }
  if((dma_trans_complete_flag == 0) || (adc1_overflow_flag != 0))
  {
    /* printf flag when error occur */
    at32_led_on(LED3);
    at32_led_on(LED4);
    printf("error occur\r\n");
    printf("adc1_overflow_flag = %d\r\n",adc1_overflow_flag);
    printf("dma_trans_complete_flag = %d\r\n",dma_trans_complete_flag);
  }
  else
  {
    /* printf data when conversion end without error */
    printf("conversion end without error\r\n");
    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]);
      printf("\r\n");
    }
  }
  at32_led_on(LED2);
  while(1)
  {
  }
}
```

4.4 实验效果

可通过串口打印查看实现效果，最终串口打印的转换数据如下。

串口配置

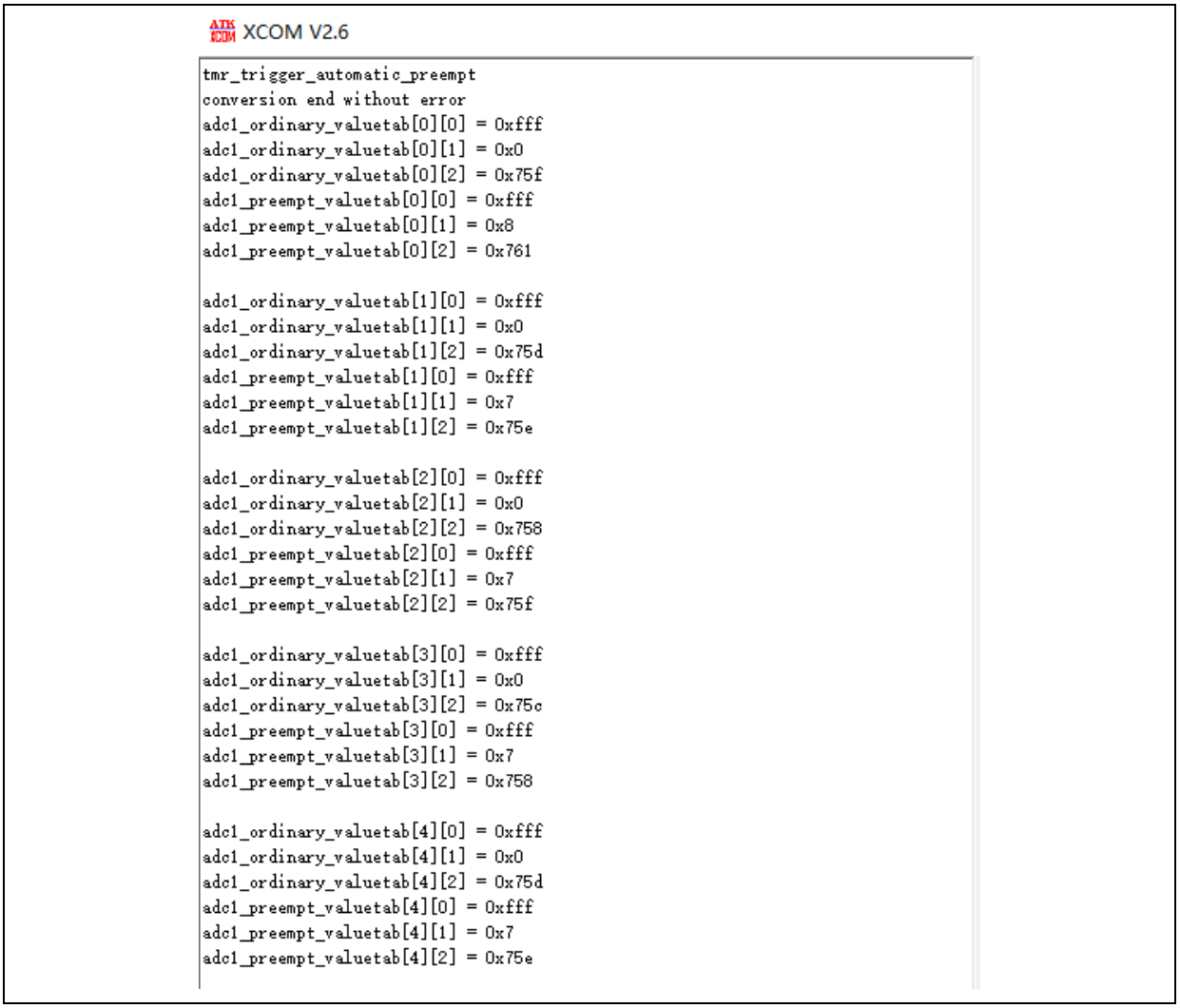
Baud rate：115200

Stop bits：1

Data bits：8

Parity：None

图 18. ADC 过采样实验结果



5 案例 ADC 电压监测

5.1 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。

5.2 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

5.3 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);

  /* config adc pin as analog input mode */
  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_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);

  dmamux_enable(DMA1, TRUE);
  dmamux_init(DMA1MUX_CHANNEL1, DMAMUX_DMAREQ_ID_ADC1);

  /* enable dma transfer complete interrupt */
  dma_interrupt_enable(DMA1_CHANNEL1, DMA_FDT_INT, FALSE);
  dma_channel_enable(DMA1_CHANNEL1, TRUE);
}
```

■ ADC 配置函数代码

```
static void adc_config(void)
{
  adc_common_config_type adc_common_struct;
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(ADC1_2_3_IRQn, 0, 0);

  adc_common_default_para_init(&adc_common_struct);

  /* config combine mode */
  adc_common_struct.combine_mode = ADC_INDEPENDENT_MODE;

  /* config division,adcclk is division by hclk */
  adc_common_struct.div = ADC_HCLK_DIV_4;

  /* config common dma mode,it's not useful in independent mode */
  adc_common_struct.common_dma_mode = ADC_COMMON_DMAMODE_DISABLE;

  /* config common dma request repeat */
  adc_common_struct.common_dma_request_repeat_state = FALSE;

  /* config adjacent adc sampling interval,it's useful for ordinary shifting mode */
```

```
  adc_common_struct.sampling_interval = ADC_SAMPLING_INTERVAL_5CYCLES;

  /* config inner temperature sensor and vintrv */
  adc_common_struct.tempervintrv_state = FALSE;

  /* config voltage battery */
  adc_common_struct.vbat_state = FALSE;
  adc_common_config(&adc_common_struct);

  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_resolution_set(ADC1, ADC_RESOLUTION_12B);

  /* config ordinary channel */
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_47_5);

  /* config ordinary trigger source and trigger edge */
  adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1CH1,
ADC_ORDINARY_TRIG_EDGE_NONE);

  /* config dma mode,it's not useful when common dma mode is use */
  adc_dma_mode_enable(ADC1, TRUE);

  /* config dma request repeat,it's not useful when common dma mode is use */
  adc_dma_request_repeat_enable(ADC1, TRUE);

  /* 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);

  /* enable adc overflow interrupt */
  adc_interrupt_enable(ADC1, ADC_OCCO_INT, TRUE);

  /* enable voltage monitoring out of range interrupt */
  adc_interrupt_enable(ADC1, ADC_VMOR_INT, TRUE);

  /* adc enable */
  adc_enable(ADC1, TRUE);
```

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

  /* adc calibration */
  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

```
/* 溢出及电压超出范围监测 */
void ADC1_2_3_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_OCCO_FLAG);
    adc1_overflow_flag++;
  }
  if(adc_flag_get(ADC1, ADC_VMOR_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_VMOR_FLAG);
    vmor_flag_index++;
  }
}
```

■ main 函数代码

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

  /* config the system clock */
  system_clock_config();

  /* init at start board */
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  usart1_config(115200);
  gpio_config();
  dma_config();
  adc_config();
  printf("voltage_monitoring \r\n");
  while(1)
  {
    at32_led_toggle(LED2);
```

```
    delay_sec(1);
    if((adc1_overflow_flag != 0) || (vmor_flag_index != 0))
    {
      /* printf flag when error occur */
      at32_led_on(LED3);
      at32_led_on(LED4);
      printf("error occur\r\n");
      printf("vmor_flag_index = %d\r\n",vmor_flag_index);
      printf("adc1_overflow_flag = %d\r\n",adc1_overflow_flag);
      printf("out of range:adc1_channel_5 value is = 0x%x!\r\n", adc1_ordinary_valuetab[1]);
    }
    adc_ordinary_software_trigger_enable(ADC1, TRUE);
  }
}
```

5.4 实验效果

可通过串口打印查看实现效果，测试过程中随机将 Channel5 外接 1.5V，最终串口打印信息如下。通过打印信息可以看到：有实际监测到 Channel5 超过监控阈值，且记录到的实际监控值为 1.5V。

串口配置

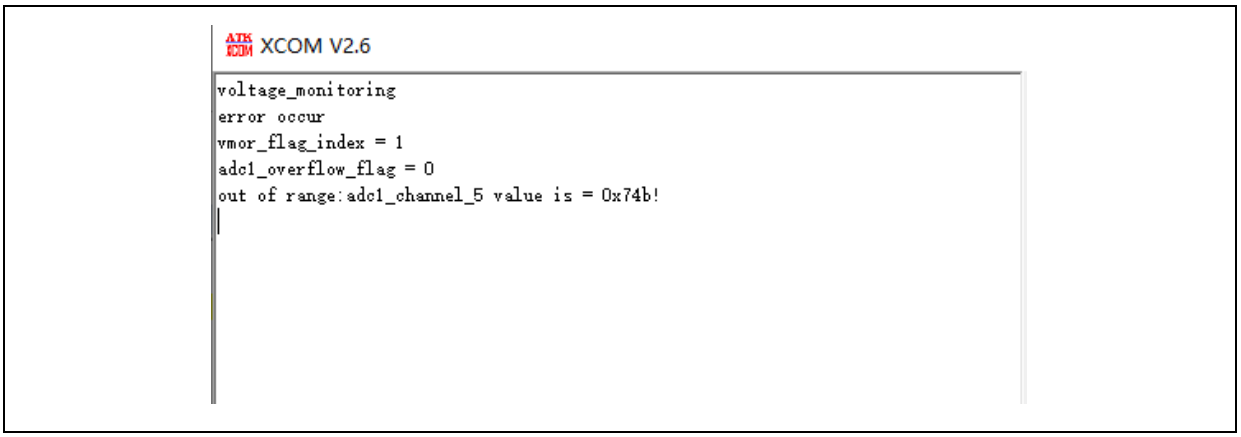
Baud rate：115200

Stop bits：1

Data bits：8

Parity：None

图 19. ADC 电压监测实验结果



6 案例 ADC 双 Buffer

6.1 功能简介

EDMA 具备双缓冲模式功能，ADC 配合这个功能使用，可有效提高数据转换效率。

本案例将以该 EDMA 的双缓冲功能为基础，示范如何配置使用 ADC 的快速转换。

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\edma_double_buffer

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);

  /* config adc pin as analog input mode */
  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 edma_config(void)
{
  edma_init_type edma_init_struct;
  crm_periph_clock_enable(CRM_EDMA_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(EDMA_Stream1_IRQn, 0, 0);
```

```
  edma_reset(EDMA_STREAM1);
  edma_default_para_init(&edma_init_struct);
  edma_init_struct.buffer_size = 3;
  edma_init_struct.direction = EDMA_DIR_PERIPHERAL_TO_MEMORY;
  edma_init_struct.memory0_base_addr = (uint32_t)adc1_ordinary_valuetab;
  edma_init_struct.memory_burst_mode = EDMA_MEMORY_SINGLE;
  edma_init_struct.memory_data_width = EDMA_MEMORY_DATA_WIDTH_HALFWORD;
  edma_init_struct.memory_inc_enable = TRUE;
  edma_init_struct.peripheral_base_addr = (uint32_t)&(ADC1->odt);
  edma_init_struct.peripheral_burst_mode = EDMA_PERIPHERAL_SINGLE;
  edma_init_struct.peripheral_data_width = EDMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
  edma_init_struct.peripheral_inc_enable = FALSE;
  edma_init_struct.priority = EDMA_PRIORITY_VERY_HIGH;
  edma_init_struct.loop_mode_enable = FALSE;
  edma_init_struct.fifo_threshold = EDMA_FIFO_THRESHOLD_1QUARTER;
  edma_init_struct.fifo_mode_enable = FALSE;
  edma_init(EDMA_STREAM1, &edma_init_struct);

  /* edmamux init and enable */
  edmamux_enable(TRUE);
  edmamux_init(EDMAMUX_CHANNEL1, EDMAMUX_DMAREQ_ID_ADC1);

  /* config double buffer mode */
  edma_double_buffer_mode_init(EDMA_STREAM1, (uint32_t)double_adc1_ordinary_valuetab,
EDMA_MEMORY_0);

  /* enable the double memory mode */
  edma_double_buffer_mode_enable(EDMA_STREAM1, TRUE);

  /* enable edma full data transfer intterrupt */
  edma_interrupt_enable(EDMA_STREAM1, EDMA_FDT_INT, TRUE);
  edma_stream_enable(EDMA_STREAM1, TRUE);
}
```

■ ADC 配置函数代码

```
static void adc_config(void)
{
  adc_common_config_type adc_common_struct;
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(ADC1_2_3_IRQn, 0, 0);

  adc_common_default_para_init(&adc_common_struct);

  /* config combine mode */
```

```
  adc_common_struct.combine_mode = ADC_INDEPENDENT_MODE;

  /* config division,adcclk is division by hclk */
  adc_common_struct.div = ADC_HCLK_DIV_17;

  /* config common dma mode,it's not useful in independent mode */
  adc_common_struct.common_dma_mode = ADC_COMMON_DMAMODE_DISABLE;

  /* config common dma request repeat */
  adc_common_struct.common_dma_request_repeat_state = FALSE;

  /* config adjacent adc sampling interval,it's useful for ordinary shifting mode */
  adc_common_struct.sampling_interval = ADC_SAMPLING_INTERVAL_5CYCLES;

  /* config inner temperature sensor and vintrv */
  adc_common_struct.tempervintrv_state = FALSE;

  /* config voltage battery */
  adc_common_struct.vbat_state = FALSE;
  adc_common_config(&adc_common_struct);

  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_resolution_set(ADC1, ADC_RESOLUTION_12B);

  /* config ordinary channel */
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_640_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_640_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_640_5);

  /* config ordinary trigger source and trigger edge */
  adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1CH1,
ADC_ORDINARY_TRIG_EDGE_NONE);

  /* config dma mode,it's not useful when common dma mode is use */
  adc_dma_mode_enable(ADC1, TRUE);

  /* config dma request repeat,it's not useful when common dma mode is use */
  adc_dma_request_repeat_enable(ADC1, TRUE);
```

```
  /* enable adc overflow interrupt */
  adc_interrupt_enable(ADC1, ADC_OCCO_INT, TRUE);

  /* adc enable */
  adc_enable(ADC1, TRUE);
  while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);

  /* adc calibration */
  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 EDMA_Stream1_IRQHandler(void)
{
  if(edma_flag_get(EDMA_FDT1_FLAG) != RESET)
  {
    if(edma_memory_target_get(EDMA_STREAM1))
    {
      double_buffer_is_useful = 0;
    }
    else
    {
      double_buffer_is_useful = 1;
    }
    edma_flag_clear(EDMA_FDT1_FLAG);
    edma_trans_complete_flag = 1;
  }
}
/* 获取 ADC 的溢出状态信息 */
void ADC1_2_3_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_OCCO_FLAG);
    adc1_overflow_flag++;
  }
}
```

■ main 函数代码

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

```
/* config the system clock */
system_clock_config();

/* init at start board */
at32_board_init();
at32_led_off(LED2);
at32_led_off(LED3);
at32_led_off(LED4);
usart1_config(115200);
gpio_config();
edma_config();
adc_config();
printf("edma_double_buffer \r\n");
while(1)
{
  adc_ordinary_software_trigger_enable(ADC1, TRUE);
  while(edma_trans_complete_flag == 0);
  edma_trans_complete_flag = 0;
  if(adc1_overflow_flag != 0)
  {
    /* printf flag when error occur */
    at32_led_on(LED4);
    printf("error occur\r\n");
    printf("adc1_overflow_flag = %d\r\n",adc1_overflow_flag);
  }
  else
  {
    printf("conversion end without error\r\n");
    /* printf data when conversion end without error */
    if(double_buffer_is_useful == 1)
    {
      printf("double_adc1_ordinary_valuetab[0] = 0x%x\r\n", double_adc1_ordinary_valuetab[0]);
      printf("double_adc1_ordinary_valuetab[1] = 0x%x\r\n", double_adc1_ordinary_valuetab[1]);
      printf("double_adc1_ordinary_valuetab[2] = 0x%x\r\n", double_adc1_ordinary_valuetab[2]);
    }
    else
    {
      printf("adc1_ordinary_valuetab[0] = 0x%x\r\n", adc1_ordinary_valuetab[0]);
      printf("adc1_ordinary_valuetab[1] = 0x%x\r\n", adc1_ordinary_valuetab[1]);
      printf("adc1_ordinary_valuetab[2] = 0x%x\r\n", adc1_ordinary_valuetab[2]);
    }
  }
  printf("\r\n");
  delay_sec(1);
}
```

```
}
```

6.4 实验效果

可通过串口打印查看实现效果，如下图所示，ADC 转换数据被硬件循环的存储到 EDMA 设定的两个 buffer 内了。

串口配置

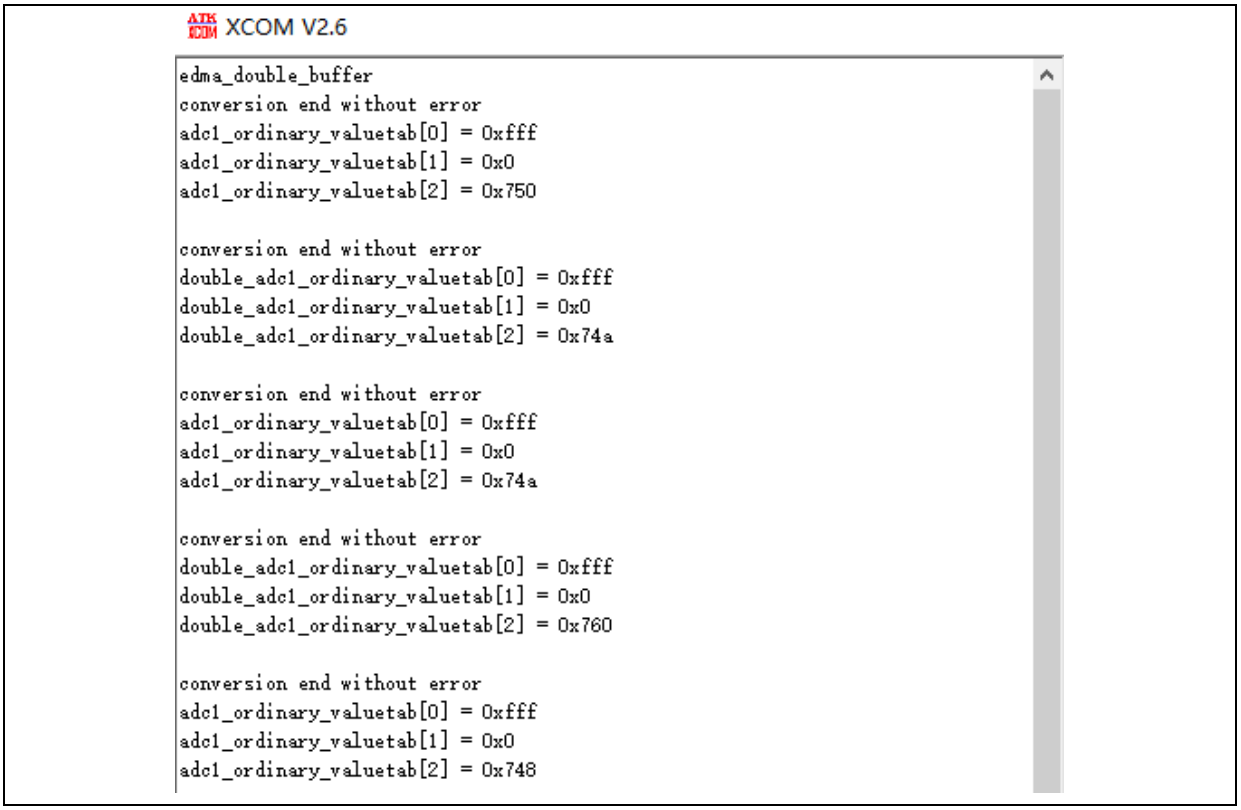
Baud rate：115200

Stop bits：1

Data bits：8

Parity：None

图 20. ADC 双 Buffer 实验结果



7 案例 ADC DMA 模式 1

7.1 功能简介

ADC 主从组合模式下的普通通道转换数据需经 DMA 传输。DMA 支持多种模式，不同模式数据的传输规则存在明显差异。

当选择不适用的 DMA 模式传输转换数据时，会因传输不及时导致数据溢出等问题。因此应用中 ADC 主从组合模式与 DMA 模式需要按照 RM 要求搭配使用。

DMA 模式 1 适用 ADC 组合模式如下：

普通同时模式(单从机)、普通位移模式(单从机)、普通同时模式(双从机)、普通位移模式(双从机)

本案例将以 “DMA 模式 1 +普通同时模式(双从机)”为基础进行使用示范。

7.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

PA4 and PA7 and PC0——3.3V

PA5 and PB0 and PC2——GND

PA6 and PB1 and PC3——1.5V 左右

2) 软件环境

project\at_start_f4xx\examples\ adc\combine_mode_ordinary_smlt_twoslave_dma1

7.3 软件设计

1) 配置流程

- 配置 ADC 使用的 GPIO
- 配置用于触发的 TMR(TMR1_TRGOUT2 event)
- 配置用于普通通道数据传输的 DMA
- ADC 相关配置及设定
- 等待触发完毕后关闭触发 TMR
- 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);
  crm_periph_clock_enable(CRM_GPIOC_PERIPH_CLOCK, TRUE);

  gpio_default_para_init(&gpio_initstructure);

  /* config adc pin as analog input mode */
  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);

  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_0 | GPIO_PINS_2 | GPIO_PINS_3;
  gpio_init(GPIOC, &gpio_initstructure);
}
```

- TMR 配置函数代码

```
static void tmr1_config(void)
{
  gpio_init_type gpio_initstructure;
  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);

  gpio_default_para_init(&gpio_initstructure);
  gpio_initstructure.gpio_mode = GPIO_MODE_MUX;
  gpio_initstructure.gpio_pins = GPIO_PINS_8;
  gpio_initstructure.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
  gpio_initstructure.gpio_pull = GPIO_PULL_NONE;
  gpio_initstructure.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
  gpio_init(GPIOA, &gpio_initstructure);
  gpio_pin_mux_config(GPIOA, GPIO_PINS_SOURCE8, GPIO_MUX_1);

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

  crm_periph_clock_enable(CRM_TMR1_PERIPH_CLOCK, TRUE);

  /* (systermclock/24000)/1000 = 10Hz(100ms) */
  tmr_base_init(TMR1, 1000, 24000);
  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_4, &tmr_oc_init_structure);
  tmr_channel_value_set(TMR1, TMR_SELECT_CHANNEL_4, 500);
```

```
  tmr_trgout2_enable(TMR1, TRUE);
  tmr_channel_enable(TMR1, TMR_SELECT_CHANNEL_4, TRUE);
  tmr_output_enable(TMR1, TRUE);
  tmr_primary_mode_select(TMR1, TMR_PRIMARY_SEL_C4ORAW);
}
```

■ 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_default_para_init(&dma_init_struct);
  dma_init_struct.buffer_size = 27;
  dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
  dma_init_struct.memory_base_addr = (uint32_t)adccom_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)&(ADCCOM->codt);
  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);

  dmamux_enable(DMA1, TRUE);
  dmamux_init(DMA1MUX_CHANNEL1, DMAMUX_DMAREQ_ID_ADC1);

  /* enable dma transfer complete interrupt */
  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_common_config_type adc_common_struct;
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_ADC2_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_ADC3_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(ADC1_2_3_IRQn, 0, 0);

  adc_common_default_para_init(&adc_common_struct);
```

```
  /* config combine mode */
  adc_common_struct.combine_mode = ADC_ORDINARY_SMLT_ONLY_TWOSLAVE_MODE;

  /* config division,adcclk is division by hclk */
  adc_common_struct.div = ADC_HCLK_DIV_4;

  /* config common dma mode,it's useful for ordinary group in combine mode */
  adc_common_struct.common_dma_mode = ADC_COMMON_DMAMODE_1;

  /* config common dma request repeat */
  adc_common_struct.common_dma_request_repeat_state = FALSE;

  /* config adjacent adc sampling interval,it's useful for ordinary shifting mode */
  adc_common_struct.sampling_interval = ADC_SAMPLING_INTERVAL_5CYCLES;

  /* config inner temperature sensor and vintrv */
  adc_common_struct.tempervintrv_state = FALSE;

  /* config voltage battery */
  adc_common_struct.vbat_state = FALSE;
  adc_common_config(&adc_common_struct);

  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_resolution_set(ADC1, ADC_RESOLUTION_12B);

  /* config ordinary channel */
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_47_5);

  /* config ordinary trigger source and trigger edge */
  adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1TRGOUT2,
ADC_ORDINARY_TRIG_EDGE_RISING);

  /* config dma mode,it's not useful when common dma mode is use */
  adc_dma_mode_enable(ADC1, FALSE);

  /* config dma request repeat,it's not useful when common dma mode is use */
  adc_dma_request_repeat_enable(ADC1, FALSE);

  /* enable adc overflow interrupt */
```

```
  adc_interrupt_enable(ADC1, ADC_OCCO_INT, TRUE);

  adc_base_config(ADC2, &adc_base_struct);
  adc_resolution_set(ADC2, ADC_RESOLUTION_12B);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_8, 2, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_9, 3, ADC_SAMPLETIME_47_5);
  adc_ordinary_conversion_trigger_set(ADC2, ADC_ORDINARY_TRIG_TMR1TRGOUT2,
ADC_ORDINARY_TRIG_EDGE_NONE);
  adc_dma_mode_enable(ADC2, FALSE);
  adc_dma_request_repeat_enable(ADC2, FALSE);
  adc_interrupt_enable(ADC2, ADC_OCCO_INT, TRUE);

  adc_base_config(ADC3, &adc_base_struct);
  adc_resolution_set(ADC3, ADC_RESOLUTION_12B);
  adc_ordinary_channel_set(ADC3, ADC_CHANNEL_10, 1, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC3, ADC_CHANNEL_12, 2, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC3, ADC_CHANNEL_13, 3, ADC_SAMPLETIME_47_5);
  adc_ordinary_conversion_trigger_set(ADC3, ADC_ORDINARY_TRIG_TMR1TRGOUT2,
ADC_ORDINARY_TRIG_EDGE_NONE);
  adc_dma_mode_enable(ADC3, FALSE);
  adc_dma_request_repeat_enable(ADC3, FALSE);
  adc_interrupt_enable(ADC3, ADC_OCCO_INT, TRUE);

  /* adc enable */
  adc_enable(ADC1, TRUE);
  adc_enable(ADC2, TRUE);
  adc_enable(ADC3, TRUE);
  while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC2, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC3, ADC_RDY_FLAG) == RESET);

  /* adc calibration */
  adc_calibration_init(ADC1);
  while(adc_calibration_init_status_get(ADC1));
  adc_calibration_start(ADC1);
  while(adc_calibration_status_get(ADC1));
  adc_calibration_init(ADC2);
  while(adc_calibration_init_status_get(ADC2));
  adc_calibration_start(ADC2);
  while(adc_calibration_status_get(ADC2));
  adc_calibration_init(ADC3);
  while(adc_calibration_init_status_get(ADC3));
  adc_calibration_start(ADC3);
  while(adc_calibration_status_get(ADC3));
}
```

■ 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);
    dma1_trans_complete_flag = 1;
  }
}
/* 获取各个 ADC 的溢出状态信息 */
void ADC1_2_3_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_OCCO_FLAG);
    adc1_overflow_flag++;
  }
  if(adc_flag_get(ADC2, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC2, ADC_OCCO_FLAG);
    adc2_overflow_flag++;
  }
  if(adc_flag_get(ADC3, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC3, ADC_OCCO_FLAG);
    adc3_overflow_flag++;
  }
}
```

■ main 函数代码

```
int main(void)
{
  __IO uint32_t index1 = 0;
  __IO uint32_t index2 = 0;
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* config the system clock */
  system_clock_config();

  /* init at start board */
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  usart1_config(115200);
```

```
  gpio_config();
  tmr1_config();
  dma_config();
  adc_config();
  printf("combine_mode_ordinary_smlt_twoslave_dma1 \r\n");
  tmr_counter_enable(TMR1, TRUE);
  while(dma1_trans_complete_flag == 0);
  tmr_counter_enable(TMR1, FALSE);
  if((adc1_overflow_flag != 0) || (adc2_overflow_flag != 0) || (adc3_overflow_flag != 0))
  {
    /* printf flag when error occur */
    at32_led_on(LED3);
    at32_led_on(LED4);
    printf("error occur\r\n");
    printf("adc1_overflow_flag = %d\r\n",adc1_overflow_flag);
    printf("adc2_overflow_flag = %d\r\n",adc2_overflow_flag);
    printf("adc3_overflow_flag = %d\r\n",adc3_overflow_flag);
  }
  else
  {
    /* printf data when conversion end without error */
    printf("conversion end without error\r\n");
    for(index1 = 0; index1 < 3; index1++)
    {
      for(index2 = 0; index2 < 3; index2++)
      {
        printf("adccom_ordinary_valuetab[%d][%d][0] = 0x%x\r\n",index1, index2,
adccom_ordinary_valuetab[index1][index2][0]);
        printf("adccom_ordinary_valuetab[%d][%d][1] = 0x%x\r\n",index1, index2,
adccom_ordinary_valuetab[index1][index2][1]);
        printf("adccom_ordinary_valuetab[%d][%d][2] = 0x%x\r\n",index1, index2,
adccom_ordinary_valuetab[index1][index2][2]);
      }
      printf("\r\n");
    }
  }
  at32_led_on(LED2);
  while(1)
  {
  }
}
```

7.4 实验效果

可通过串口打印查看实现效果，最终串口打印的转换数据如下。

ADC 转换数据以半字为单位进行传输，传输顺序为：

ADC1 第一个转换通道；

ADC2 第一个转换通道；

ADC3 第一个转换通道；

ADC1 第二个转换通道；

ADC2 第二个转换通道；

ADC3 第二个转换通道；

ADC1 第三个转换通道；

ADC2 第三个转换通道；

ADC3 第三个转换通道；

ADC1 第一个转换通道……

串口配置

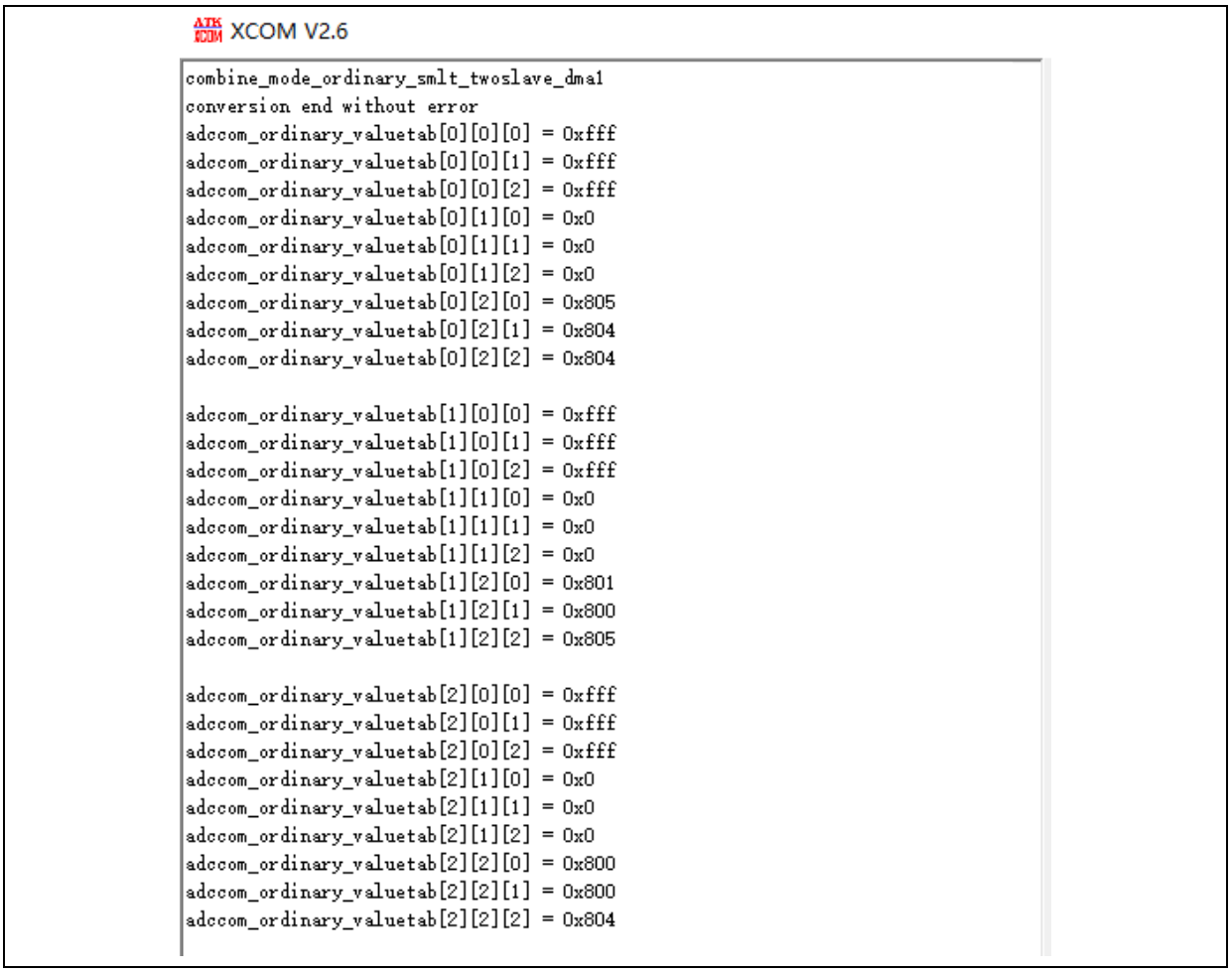
Baud rate：115200

Stop bits：1

Data bits：8

Parity：None

图 21. ADC DMA 模式 1 实验结果



8 案例 ADC DMA 模式 2

8.1 功能简介

ADC 主从组合模式下的普通通道转换数据需经 DMA 传输。DMA 支持多种模式，不同模式数据的传输规则存在明显差异。

当选择不适用的 DMA 模式传输转换数据时，会因传输不及时导致数据溢出等问题。因此应用中 ADC 主从组合模式与 DMA 模式需要按照 RM 要求搭配使用。

DMA 模式 2 适用 ADC 组合模式如下：

普通同时模式(单从机)、普通位移模式(单从机)、普通位移模式(双从机)

本案例将以 “DMA 模式 2 +普通同时模式(单从机)”为基础进行使用示范。

8.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

PA4 and PA7——3.3V

PA5 and PB0——GND

PA6 and PB1——1.5V 左右

2) 软件环境

project\at_start_f4xx\examples\ adc\combine_mode_ordinary_smlt_oneslave_dma2

8.3 软件设计

1) 配置流程

- 开 ADC 使用的 GPIO 配置
- 配置用于普通通道数据传输的 DMA
- ADC 相关配及设定
- 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);
  crm_periph_clock_enable(CRM_GPIOB_PERIPH_CLOCK, TRUE);

  gpio_default_para_init(&gpio_initstructure);

  /* config adc pin as analog input mode */
  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_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)adccom_ordinary_valuetab;
  dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_WORD;
  dma_init_struct.memory_inc_enable = TRUE;
  dma_init_struct.peripheral_base_addr = (uint32_t)&(ADCCOM->codt);
  dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_WORD;
  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);

  dmamux_enable(DMA1, TRUE);
  dmamux_init(DMA1MUX_CHANNEL1, DMAMUX_DMAREQ_ID_ADC1);

  /* enable dma transfer complete interrupt */
  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_common_config_type adc_common_struct;
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_ADC2_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(ADC1_2_3_IRQn, 0, 0);

  adc_common_default_para_init(&adc_common_struct);

  /* config combine mode */
```

```
  adc_common_struct.combine_mode = ADC_ORDINARY_SMLT_ONLY_ONESLAVE_MODE;

  /* config division,adcclk is division by hclk */
  adc_common_struct.div = ADC_HCLK_DIV_4;

  /* config common dma mode,it's useful for ordinary group in combine mode */
  adc_common_struct.common_dma_mode = ADC_COMMON_DMAMODE_2;

  /* config common dma request repeat */
  adc_common_struct.common_dma_request_repeat_state = FALSE;

  /* config adjacent adc sampling interval,it's useful for ordinary shifting mode */
  adc_common_struct.sampling_interval = ADC_SAMPLING_INTERVAL_5CYCLES;

  /* config inner temperature sensor and vintrv */
  adc_common_struct.tempervintrv_state = FALSE;

  /* config voltage battery */
  adc_common_struct.vbat_state = FALSE;
  adc_common_config(&adc_common_struct);

  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_resolution_set(ADC1, ADC_RESOLUTION_12B);

  /* config ordinary channel */
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_640_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_640_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_640_5);

  /* config ordinary trigger source and trigger edge */
  adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1CH1,
ADC_ORDINARY_TRIG_EDGE_NONE);

  /* config dma mode,it's not useful when common dma mode is use */
  adc_dma_mode_enable(ADC1, FALSE);

  /* config dma request repeat,it's not useful when common dma mode is use */
  adc_dma_request_repeat_enable(ADC1, FALSE);

  /* enable adc overflow interrupt */
```

```
  adc_interrupt_enable(ADC1, ADC_OCCO_INT, TRUE);

  adc_base_config(ADC2, &adc_base_struct);
  adc_resolution_set(ADC2, ADC_RESOLUTION_12B);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_640_5);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_8, 2, ADC_SAMPLETIME_640_5);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_9, 3, ADC_SAMPLETIME_640_5);
  adc_ordinary_conversion_trigger_set(ADC2, ADC_ORDINARY_TRIG_TMR1CH1,
ADC_ORDINARY_TRIG_EDGE_NONE);
  adc_dma_mode_enable(ADC2, FALSE);
  adc_dma_request_repeat_enable(ADC2, FALSE);
  adc_interrupt_enable(ADC2, ADC_OCCO_INT, TRUE);

  /* adc enable */
  adc_enable(ADC1, TRUE);
  adc_enable(ADC2, TRUE);
  while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC2, ADC_RDY_FLAG) == RESET);

  /* adc calibration */
  adc_calibration_init(ADC1);
  while(adc_calibration_init_status_get(ADC1));
  adc_calibration_start(ADC1);
  while(adc_calibration_status_get(ADC1));
  adc_calibration_init(ADC2);
  while(adc_calibration_init_status_get(ADC2));
  adc_calibration_start(ADC2);
  while(adc_calibration_status_get(ADC2));
}
```

■ 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);
    dma1_trans_complete_flag = 1;
  }
}
/* 获取各个 ADC 的溢出状态信息 */
void ADC1_2_3_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_OCCO_FLAG);
```

```
    adc1_overflow_flag++;
  }
  if(adc_flag_get(ADC2, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC2, ADC_OCCO_FLAG);
    adc2_overflow_flag++;
  }
}
```

■ main 函数代码

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

  /* config the system clock */
  system_clock_config();

  /* init at start board */
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  usart1_config(115200);
  gpio_config();
  dma_config();
  adc_config();
  printf("combine_mode_ordinary_smlt_oneslave_dma2 \r\n");

  /* adc1 software trigger start conversion */
  for(index = 0; index < 5; index++)
  {
    adc_ordinary_software_trigger_enable(ADC1, TRUE);
    delay_ms(100);
  }

  if((dma1_trans_complete_flag == 0) || (adc1_overflow_flag != 0) || (adc2_overflow_flag != 0))
  {
    /* printf flag when error occur */
    at32_led_on(LED3);
    at32_led_on(LED4);
    printf("error occur\r\n");
    printf("dma1_trans_complete_flag = %d\r\n",dma1_trans_complete_flag);
    printf("adc1_overflow_flag = %d\r\n",adc1_overflow_flag);
    printf("adc2_overflow_flag = %d\r\n",adc2_overflow_flag);
  }
```

```
  else
  {
    /* printf data when conversion end without error */
    printf("conversion end without error\r\n");
    for(index = 0; index < 5; index++)
    {
      printf("adc1_ordinary_channel4[%d] = 0x%x\r\n",index, adccom_ordinary_valuetab[index][0] &
0xFFFF);
      printf("adc1_ordinary_channel5[%d] = 0x%x\r\n",index, adccom_ordinary_valuetab[index][1] &
0xFFFF);
      printf("adc1_ordinary_channel6[%d] = 0x%x\r\n",index, adccom_ordinary_valuetab[index][2] &
0xFFFF);
      printf("adc2_ordinary_channel7[%d] = 0x%x\r\n",index, (adccom_ordinary_valuetab[index][0] >> 16)
& 0xFFFF);
      printf("adc2_ordinary_channel8[%d] = 0x%x\r\n",index, (adccom_ordinary_valuetab[index][1] >> 16)
& 0xFFFF);
      printf("adc2_ordinary_channel9[%d] = 0x%x\r\n",index, (adccom_ordinary_valuetab[index][2] >> 16)
& 0xFFFF);
      printf("\r\n");
    }
  }
  at32_led_on(LED2);
  while(1)
  {
  }
}
```

8.4 实验效果

可通过串口打印查看实现效果，最终串口打印的转换数据如下。

ADC 转换数据以字为单位进行传输，传输顺序为：

(ADC2 第一个转换通道<<16) | ADC1 第一个转换通道；

(ADC2 第二个转换通道<<16) | ADC1 第二个转换通道；

(ADC2 第三个转换通道<<16) | ADC1 第三个转换通道；

(ADC2 第一个转换通道<<16) | ADC1 第一个转换通道……

串口配置

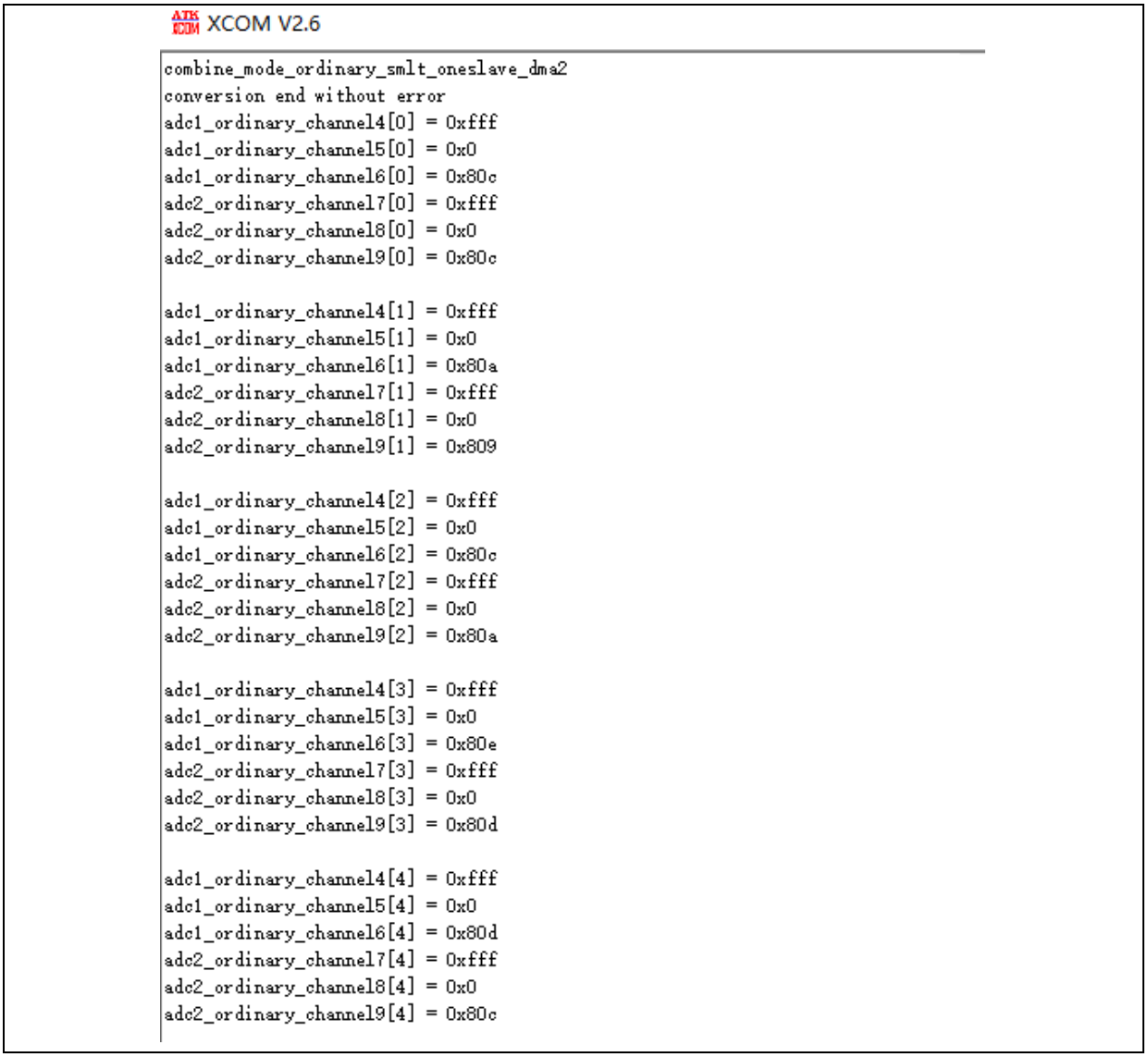
Baud rate：115200

Stop bits：1

Data bits：8

Parity：None

图 22. ADC DMA 模式 2 实验结果



9 案例 ADC DMA 模式 3

9.1 功能简介

ADC 主从组合模式下的普通通道转换数据需经 DMA 传输。DMA 支持多种模式，不同模式数据的传输规则存在明显差异。

当选择不适用的 DMA 模式传输转换数据时，会因传输不及时导致数据溢出等问题。因此应用中 ADC 主从组合模式与 DMA 模式需要按照 RM 要求搭配使用。

DMA 模式 3 适用 ADC 组合模式如下：

6/8 位分辨率的普通同时模式(单从机)、6/8 位分辨率的普通位移模式(单从机)、6/8 位分辨率的普通位移模式(双从机)

本案例将以“DMA 模式 3 + 8 位分辨率的普通位移模式(双从机)”为基础进行使用示范。

9.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

PA4——3.3V

PA7——GND

PC0——1.5V 左右

2) 软件环境

project\at_start_f4xx\examples\ adc\combine_mode_ordinary_shift_twoslave_dma3

9.3 软件设计

1) 配置流程

- 配置 ADC 使用的 GPIO
- 配置用于触发的 TMR(TMR1_TRGOUT event)
- 配置用于普通通道数据传输的 DMA
- ADC 相关配置及设定
- 等待触发完毕后关闭触发 TMR
- 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);
  crm_periph_clock_enable(CRM_GPIOC_PERIPH_CLOCK, TRUE);

  gpio_default_para_init(&gpio_initstructure);

  /* config adc pin as analog input mode */
```

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

  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_0;
  gpio_init(GPIOC, &gpio_initstructure); }
```

- TMR 配置函数代码

```
static void tmr1_config(void)
{
  crm_clocks_freq_type crm_clocks_freq_struct = {0};

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

  crm_periph_clock_enable(CRM_TMR1_PERIPH_CLOCK, TRUE);

  /* (systemclock / 24000) / 1000 = 10Hz(100ms) */
  tmr_base_init(TMR1, 1000, 24000);
  tmr_cnt_dir_set(TMR1, TMR_COUNT_UP);
  tmr_clock_source_div_set(TMR1, TMR_CLOCK_DIV1);
  tmr_primary_mode_select(TMR1, TMR_PRIMARY_SEL_OVERFLOW);
}
```

- 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_default_para_init(&dma_init_struct);
  dma_init_struct.buffer_size = 6;
  dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
  dma_init_struct.memory_base_addr = (uint32_t)adccom_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)&(ADCCOM->codt);
  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);

  dmamux_enable(DMA1, TRUE);
```

```
  dmamux_init(DMA1MUX_CHANNEL1, DMAMUX_DMAREQ_ID_ADC1);

  /* enable dma transfer complete interrupt */
  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_common_config_type adc_common_struct;
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_ADC2_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_ADC3_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(ADC1_2_3_IRQn, 0, 0);

  adc_common_default_para_init(&adc_common_struct);

  /* config combine mode */
  adc_common_struct.combine_mode = ADC_ORDINARY_SHIFT_ONLY_TWOSLAVE_MODE;

  /* config division,adcclk is division by hclk */
  adc_common_struct.div = ADC_HCLK_DIV_4;

  /* config common dma mode,it's useful for ordinary group in combine mode */
  adc_common_struct.common_dma_mode = ADC_COMMON_DMAMODE_3;

  /* config common dma request repeat */
  adc_common_struct.common_dma_request_repeat_state = FALSE;

  /* config adjacent adc sampling interval,it's useful for ordinary shifting mode */
  adc_common_struct.sampling_interval = ADC_SAMPLING_INTERVAL_20CYCLES;

  /* config inner temperature sensor and vintrv */
  adc_common_struct.tempervintrv_state = FALSE;

  /* config voltage battery */
  adc_common_struct.vbat_state = FALSE;
  adc_common_config(&adc_common_struct);

  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_resolution_set(ADC1, ADC_RESOLUTION_12B);

  /* config ordinary channel */
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_2_5);

  /* config ordinary trigger source and trigger edge */
  adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1TRGOUT,
ADC_ORDINARY_TRIG_EDGE_RISING);

  /* config dma mode,it's not useful when common dma mode is use */
  adc_dma_mode_enable(ADC1, FALSE);

  /* config dma request repeat,it's not useful when common dma mode is use */
  adc_dma_request_repeat_enable(ADC1, FALSE);

  /* enable adc overflow interrupt */
  adc_interrupt_enable(ADC1, ADC_OCCO_INT, TRUE);

  adc_base_config(ADC2, &adc_base_struct);
  adc_resolution_set(ADC2, ADC_RESOLUTION_12B);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_2_5);
  adc_ordinary_conversion_trigger_set(ADC2, ADC_ORDINARY_TRIG_TMR1TRGOUT,
ADC_ORDINARY_TRIG_EDGE_NONE);
  adc_dma_mode_enable(ADC2, FALSE);
  adc_dma_request_repeat_enable(ADC2, FALSE);
  adc_interrupt_enable(ADC2, ADC_OCCO_INT, TRUE);

  adc_base_config(ADC3, &adc_base_struct);
  adc_resolution_set(ADC3, ADC_RESOLUTION_12B);
  adc_ordinary_channel_set(ADC3, ADC_CHANNEL_10, 1, ADC_SAMPLETIME_2_5);
  adc_ordinary_conversion_trigger_set(ADC3, ADC_ORDINARY_TRIG_TMR1TRGOUT,
ADC_ORDINARY_TRIG_EDGE_NONE);
  adc_dma_mode_enable(ADC3, FALSE);
  adc_dma_request_repeat_enable(ADC3, FALSE);
  adc_interrupt_enable(ADC3, ADC_OCCO_INT, TRUE);

  /* adc enable */
  adc_enable(ADC1, TRUE);
  adc_enable(ADC2, TRUE);
  adc_enable(ADC3, TRUE);
  while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC2, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC3, ADC_RDY_FLAG) == RESET);
```

```
  /* adc calibration */
  adc_calibration_init(ADC1);
  while(adc_calibration_init_status_get(ADC1));
  adc_calibration_start(ADC1);
  while(adc_calibration_status_get(ADC1));
  adc_calibration_init(ADC2);
  while(adc_calibration_init_status_get(ADC2));
  adc_calibration_start(ADC2);
  while(adc_calibration_status_get(ADC2));
  adc_calibration_init(ADC3);
  while(adc_calibration_init_status_get(ADC3));
  adc_calibration_start(ADC3);
  while(adc_calibration_status_get(ADC3));

  /*set resolution to 8bit.this because calibration must perform at 12 bit resolution */
  adc_resolution_set(ADC1,ADC_RESOLUTION_8B);
  adc_resolution_set(ADC2,ADC_RESOLUTION_8B);
  adc_resolution_set(ADC3,ADC_RESOLUTION_8B);
  while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC2, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC3, ADC_RDY_FLAG) == RESET);
}
```

■ 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);
    dma1_trans_complete_flag = 1;
  }
}
/* 获取各个 ADC 的溢出状态信息 */
void ADC1_2_3_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_OCCO_FLAG);
    adc1_overflow_flag++;
  }
  if(adc_flag_get(ADC2, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC2, ADC_OCCO_FLAG);
    adc2_overflow_flag++;
```

```
  }
  if(adc_flag_get(ADC3, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC3, ADC_OCCO_FLAG);
    adc3_overflow_flag++;
  }
}
```

■ main 函数代码

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

  /* config the system clock */
  system_clock_config();

  /* init at start board */
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  usart1_config(115200);
  gpio_config();
  tmr1_config();
  dma_config();
  adc_config();
  printf("combine_mode_ordinary_shift_twoslave_dma3 \r\n");
  tmr_counter_enable(TMR1, TRUE);
  while(dma1_trans_complete_flag == 0);
  tmr_counter_enable(TMR1, FALSE);
  if((adc1_overflow_flag != 0) || (adc2_overflow_flag != 0) || (adc3_overflow_flag != 0))
  {
    /* printf flag when error occur */
    at32_led_on(LED3);
    at32_led_on(LED4);
    printf("error occur\r\n");
    printf("adc1_overflow_flag = %d\r\n",adc1_overflow_flag);
    printf("adc2_overflow_flag = %d\r\n",adc2_overflow_flag);
    printf("adc3_overflow_flag = %d\r\n",adc3_overflow_flag);
  }
  else
  {
    /* printf data when conversion end without error */
    printf("conversion end without error\r\n");
    for(index = 0; index < 6; index++)
```

```
    {
      printf("adccom_ordinary_valuetab[%d] = 0x%x\r\n",index, adccom_ordinary_valuetab[index]);
    }
    printf("\r\n");
  }
  at32_led_on(LED2);
  while(1)
  {
  }
}
```

9.4 实验效果

可通过串口打印查看实现效果，最终串口打印的转换数据如下。

ADC 转换数据以半字为单位进行传输，传输顺序为：

(ADC2 的转换通道<<8) | ADC1 的转换通道；

(ADC1 的转换通道<<8) | ADC3 的转换通道；

(ADC3 的转换通道<<8) | ADC2 的转换通道；

(ADC2 的转换通道<<8) | ADC1 的转换通道……

串口配置

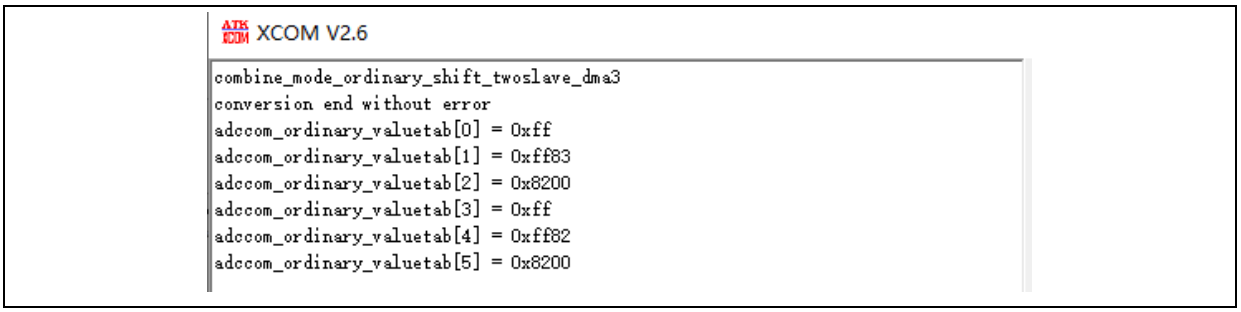
Baud rate：115200

Stop bits：1

Data bits：8

Parity：None

图 23. ADC DMA 模式 3 实验结果



10 案例 ADC DMA 模式 4

10.1 功能简介

ADC 主从组合模式下的普通通道转换数据需经 DMA 传输。DMA 支持多种模式，不同模式数据的传输规则存在明显差异。

当选择不适用的 DMA 模式传输转换数据时，会因传输不及时导致数据溢出等问题。因此应用中 ADC 主从组合模式与 DMA 模式需要按照 RM 要求搭配使用。

DMA 模式 4 适用 ADC 组合模式如下：

6/8 位分辨率的普通同时模式(双从机)、6/8 位分辨率的普通位移模式(双从机)

本案例将以“DMA 模式 4 + 8 位分辨率的普通位移模式(双从机)”为基础进行使用示范。

10.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

PA4——3.3V

PA7——GND

PC0——1.5V 左右

2) 软件环境

project\at_start_f4xx\examples\ adc\combine_mode_ordinary_shift_twoslave_dma4

10.3 软件设计

1) 配置流程

- 配置 ADC 使用的 GPIO
- 配置用于触发的 TMR(TMR1_TRGOUT event)
- 配置用于普通通道数据传输的 DMA
- ADC 相关配置及设定
- 等待触发完毕后关闭触发 TMR
- 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);
  crm_periph_clock_enable(CRM_GPIOC_PERIPH_CLOCK, TRUE);

  gpio_default_para_init(&gpio_initstructure);

  /* config adc pin as analog input mode */
  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
```

```
  gpio_initstructure.gpio_pins = GPIO_PINS_4 | GPIO_PINS_7;
  gpio_init(GPIOA, &gpio_initstructure);

  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_0;
  gpio_init(GPIOC, &gpio_initstructure);
 }
```

■ TMR 配置函数代码

```
static void tmr1_config(void)
{
  crm_clocks_freq_type crm_clocks_freq_struct = {0};

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

  crm_periph_clock_enable(CRM_TMR1_PERIPH_CLOCK, TRUE);

  /* (systemclock / 24000) / 1000 = 10Hz(100ms) */
  tmr_base_init(TMR1, 1000, 24000);
  tmr_cnt_dir_set(TMR1, TMR_COUNT_UP);
  tmr_clock_source_div_set(TMR1, TMR_CLOCK_DIV1);
  tmr_primary_mode_select(TMR1, TMR_PRIMARY_SEL_OVERFLOW);
}
```

■ 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_default_para_init(&dma_init_struct);
  dma_init_struct.buffer_size = 5;
  dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
  dma_init_struct.memory_base_addr = (uint32_t)adccom_ordinary_valuetab;
  dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_WORD;
  dma_init_struct.memory_inc_enable = TRUE;
  dma_init_struct.peripheral_base_addr = (uint32_t)&(ADCCOM->codt);
  dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_WORD;
  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);

  dmamux_enable(DMA1, TRUE);
```

```
  dmamux_init(DMA1MUX_CHANNEL1, DMAMUX_DMAREQ_ID_ADC1);

  /* enable dma transfer complete interrupt */
  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_common_config_type adc_common_struct;
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_ADC2_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_ADC3_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(ADC1_2_3_IRQn, 0, 0);

  adc_common_default_para_init(&adc_common_struct);

  /* config combine mode */
  adc_common_struct.combine_mode = ADC_ORDINARY_SHIFT_ONLY_TWOSLAVE_MODE;

  /* config division,adcclk is division by hclk */
  adc_common_struct.div = ADC_HCLK_DIV_4;

  /* config common dma mode,it's useful for ordinary group in combine mode */
  adc_common_struct.common_dma_mode = ADC_COMMON_DMAMODE_4;

  /* config common dma request repeat */
  adc_common_struct.common_dma_request_repeat_state = FALSE;

  /* config adjacent adc sampling interval,it's useful for ordinary shifting mode */
  adc_common_struct.sampling_interval = ADC_SAMPLING_INTERVAL_20CYCLES;

  /* config inner temperature sensor and vintrv */
  adc_common_struct.tempervintrv_state = FALSE;

  /* config voltage battery */
  adc_common_struct.vbat_state = FALSE;
  adc_common_config(&adc_common_struct);

  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_resolution_set(ADC1, ADC_RESOLUTION_12B);

  /* config ordinary channel */
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_2_5);

  /* config ordinary trigger source and trigger edge */
  adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1TRGOUT,
ADC_ORDINARY_TRIG_EDGE_RISING);

  /* config dma mode,it's not useful when common dma mode is use */
  adc_dma_mode_enable(ADC1, FALSE);

  /* config dma request repeat,it's not useful when common dma mode is use */
  adc_dma_request_repeat_enable(ADC1, FALSE);

  /* enable adc overflow interrupt */
  adc_interrupt_enable(ADC1, ADC_OCCO_INT, TRUE);

  adc_base_config(ADC2, &adc_base_struct);
  adc_resolution_set(ADC2, ADC_RESOLUTION_12B);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_2_5);
  adc_ordinary_conversion_trigger_set(ADC2, ADC_ORDINARY_TRIG_TMR1TRGOUT,
ADC_ORDINARY_TRIG_EDGE_NONE);
  adc_dma_mode_enable(ADC2, FALSE);
  adc_dma_request_repeat_enable(ADC2, FALSE);
  adc_interrupt_enable(ADC2, ADC_OCCO_INT, TRUE);

  adc_base_config(ADC3, &adc_base_struct);
  adc_resolution_set(ADC3, ADC_RESOLUTION_12B);
  adc_ordinary_channel_set(ADC3, ADC_CHANNEL_10, 1, ADC_SAMPLETIME_2_5);
  adc_ordinary_conversion_trigger_set(ADC3, ADC_ORDINARY_TRIG_TMR1TRGOUT,
ADC_ORDINARY_TRIG_EDGE_NONE);
  adc_dma_mode_enable(ADC3, FALSE);
  adc_dma_request_repeat_enable(ADC3, FALSE);
  adc_interrupt_enable(ADC3, ADC_OCCO_INT, TRUE);

  /* adc enable */
  adc_enable(ADC1, TRUE);
  adc_enable(ADC2, TRUE);
  adc_enable(ADC3, TRUE);
  while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC2, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC3, ADC_RDY_FLAG) == RESET);
```

```
  /* adc calibration */
  adc_calibration_init(ADC1);
  while(adc_calibration_init_status_get(ADC1));
  adc_calibration_start(ADC1);
  while(adc_calibration_status_get(ADC1));
  adc_calibration_init(ADC2);
  while(adc_calibration_init_status_get(ADC2));
  adc_calibration_start(ADC2);
  while(adc_calibration_status_get(ADC2));
  adc_calibration_init(ADC3);
  while(adc_calibration_init_status_get(ADC3));
  adc_calibration_start(ADC3);
  while(adc_calibration_status_get(ADC3));

  /*set resolution to 8bit.this because calibration must perform at 12 bit resolution */
  adc_resolution_set(ADC1,ADC_RESOLUTION_8B);
  adc_resolution_set(ADC2,ADC_RESOLUTION_8B);
  adc_resolution_set(ADC3,ADC_RESOLUTION_8B);
  while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC2, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC3, ADC_RDY_FLAG) == RESET);
}
```

■ 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);
    dma1_trans_complete_flag = 1;
  }
}
/* 获取各个 ADC 的溢出状态信息 */
void ADC1_2_3_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_OCCO_FLAG);
    adc1_overflow_flag++;
  }
  if(adc_flag_get(ADC2, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC2, ADC_OCCO_FLAG);
    adc2_overflow_flag++;
  }
```

```
  if(adc_flag_get(ADC3, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC3, ADC_OCCO_FLAG);
    adc3_overflow_flag++;
  }
}
```

■ main 函数代码

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

  /* config the system clock */
  system_clock_config();

  /* init at start board */
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  usart1_config(115200);
  gpio_config();
  tmr1_config();
  dma_config();
  adc_config();
  printf("combine_mode_ordinary_shift_twoslave_dma4 \r\n");
  tmr_counter_enable(TMR1, TRUE);
  while(dma1_trans_complete_flag == 0);
  tmr_counter_enable(TMR1, FALSE);
  if((adc1_overflow_flag != 0) || (adc2_overflow_flag != 0) || (adc3_overflow_flag != 0))
  {
    /* printf flag when error occur */
    at32_led_on(LED3);
    at32_led_on(LED4);
    printf("error occur\r\n");
    printf("adc1_overflow_flag = %d\r\n",adc1_overflow_flag);
    printf("adc2_overflow_flag = %d\r\n",adc2_overflow_flag);
    printf("adc3_overflow_flag = %d\r\n",adc3_overflow_flag);
  }
  else
  {
    /* printf data when conversion end without error */
    printf("conversion end without error\r\n");
    for(index = 0; index < 5; index++)
    {
```

```
        printf("adccom_ordinary_valuetab[%d] = 0x%x\r\n",index, adccom_ordinary_valuetab[index]);
      }
      printf("\r\n");
    }
    at32_led_on(LED2);
    while(1)
    {
    }
}
```

10.4 实验效果

可通过串口打印查看实现效果，最终串口打印的转换数据如下。

ADC 转换数据以字为单位进行传输，传输顺序为：

(ADC3 的转换通道<<16) | (ADC2 的转换通道<<8) | ADC1 的转换通道；

(ADC3 的转换通道<<16) | (ADC2 的转换通道<<8) | ADC1 的转换通道……

串口配置

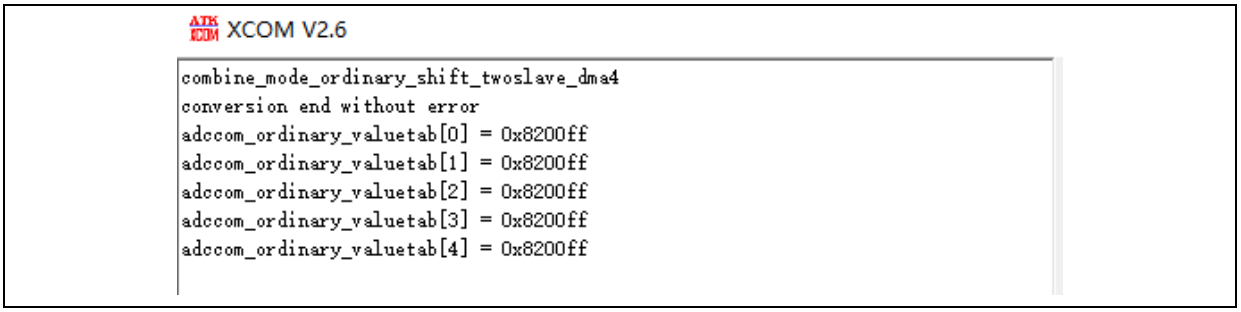
Baud rate：115200

Stop bits：1

Data bits：8

Parity：None

图 24. ADC DMA 模式 4 实验结果



11 案例 ADC DMA 模式 5

11.1 功能简介

ADC 主从组合模式下的普通通道转换数据需经 DMA 传输。DMA 支持多种模式，不同模式数据的传输规则存在明显差异。

当选择不适用的 DMA 模式传输转换数据时，会因传输不及时导致数据溢出等问题。因此应用中 ADC 主从组合模式与 DMA 模式需要按照 RM 要求搭配使用。

DMA 模式 5 适用 ADC 组合模式如下：

普通同时模式(双从机)、普通位移模式(双从机)

本案例将以 “DMA 模式 5 +普通同时模式(双从机)”为基础进行使用示范。

11.2 资源准备

1) 硬件环境:

对应产品型号的 AT-START BOARD

PA4 and PA7 and PC0——3.3V

PA5 and PB0 and PC2——GND

PA6 and PB1 and PC3——1.5V 左右

2) 软件环境

project\at_start_f4xx\examples\ adc\ combine_mode_ordinary_smlt_twoslave_dma5

11.3 软件设计

1) 配置流程

- 配置 ADC 使用的 GPIO
- 配置用于触发的 TMR(TMR1_CH1 event)
- 配置用于普通通道数据传输的 DMA
- ADC 相关配置及设定
- 等待触发完毕后关闭触发 TMR
- 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);
  crm_periph_clock_enable(CRM_GPIOC_PERIPH_CLOCK, TRUE);

  gpio_default_para_init(&gpio_initstructure);

  /* config adc pin as analog input mode */
  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);

  gpio_initstructure.gpio_mode = GPIO_MODE_ANALOG;
  gpio_initstructure.gpio_pins = GPIO_PINS_0 | GPIO_PINS_2 | GPIO_PINS_3;
  gpio_init(GPIOC, &gpio_initstructure);
}
```

■ TMR 配置函数代码

```
static void tmr1_config(void)
{
  gpio_init_type gpio_initstructure;
  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);

  gpio_default_para_init(&gpio_initstructure);
  gpio_initstructure.gpio_mode = GPIO_MODE_MUX;
  gpio_initstructure.gpio_pins = GPIO_PINS_8;
  gpio_initstructure.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
  gpio_initstructure.gpio_pull = GPIO_PULL_NONE;
  gpio_initstructure.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
  gpio_init(GPIOA, &gpio_initstructure);
  gpio_pin_mux_config(GPIOA, GPIO_PINS_SOURCE8, GPIO_MUX_1);

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

  crm_periph_clock_enable(CRM_TMR1_PERIPH_CLOCK, TRUE);

  /* (systermclock/24000)/1000 = 10Hz(100ms) */
  tmr_base_init(TMR1, 1000, 24000);
  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_1, &tmr_oc_init_structure);
  tmr_channel_value_set(TMR1, TMR_SELECT_CHANNEL_1, 500);
```

```
  tmr_channel_enable(TMR1, TMR_SELECT_CHANNEL_1, TRUE);
  tmr_output_enable(TMR1, TRUE);
}
```

■ 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_default_para_init(&dma_init_struct);
  dma_init_struct.buffer_size = 18;
  dma_init_struct.direction = DMA_DIR_PERIPHERAL_TO_MEMORY;
  dma_init_struct.memory_base_addr = (uint32_t)adccom_ordinary_valuetab;
  dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_WORD;
  dma_init_struct.memory_inc_enable = TRUE;
  dma_init_struct.peripheral_base_addr = (uint32_t)&(ADCCOM->codt);
  dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_WORD;
  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);

  dmamux_enable(DMA1, TRUE);
  dmamux_init(DMA1MUX_CHANNEL1, DMAMUX_DMAREQ_ID_ADC1);

  /* enable dma transfer complete interrupt */
  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_common_config_type adc_common_struct;
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_ADC2_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_ADC3_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(ADC1_2_3_IRQn, 0, 0);

  adc_common_default_para_init(&adc_common_struct);

  /* config combine mode */
  adc_common_struct.combine_mode = ADC_ORDINARY_SMLT_ONLY_TWOSLAVE_MODE;
```

```
  /* config division,adcclk is division by hclk */
  adc_common_struct.div = ADC_HCLK_DIV_4;

  /* config common dma mode,it's useful for ordinary group in combine mode */
  adc_common_struct.common_dma_mode = ADC_COMMON_DMAMODE_5;

  /* config common dma request repeat */
  adc_common_struct.common_dma_request_repeat_state = FALSE;

  /* config adjacent adc sampling interval,it's useful for ordinary shifting mode */
  adc_common_struct.sampling_interval = ADC_SAMPLING_INTERVAL_5CYCLES;

  /* config inner temperature sensor and vintrv */
  adc_common_struct.tempervintrv_state = FALSE;

  /* config voltage battery */
  adc_common_struct.vbat_state = FALSE;
  adc_common_config(&adc_common_struct);

  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_resolution_set(ADC1, ADC_RESOLUTION_12B);

  /* config ordinary channel */
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_6, 3, ADC_SAMPLETIME_47_5);

  /* config ordinary trigger source and trigger edge */
  adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1CH1,
ADC_ORDINARY_TRIG_EDGE_RISING);

  /* config dma mode,it's not useful when common dma mode is use */
  adc_dma_mode_enable(ADC1, FALSE);

  /* config dma request repeat,it's not useful when common dma mode is use */
  adc_dma_request_repeat_enable(ADC1, FALSE);

  /* enable adc overflow interrupt */
  adc_interrupt_enable(ADC1, ADC_OCCO_INT, TRUE);
```

```
  adc_base_config(ADC2, &adc_base_struct);
  adc_resolution_set(ADC2, ADC_RESOLUTION_12B);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_8, 2, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_9, 3, ADC_SAMPLETIME_47_5);
  adc_ordinary_conversion_trigger_set(ADC2, ADC_ORDINARY_TRIG_TMR1CH1,
ADC_ORDINARY_TRIG_EDGE_NONE);
  adc_dma_mode_enable(ADC2, FALSE);
  adc_dma_request_repeat_enable(ADC2, FALSE);
  adc_interrupt_enable(ADC2, ADC_OCCO_INT, TRUE);

  adc_base_config(ADC3, &adc_base_struct);
  adc_resolution_set(ADC3, ADC_RESOLUTION_12B);
  adc_ordinary_channel_set(ADC3, ADC_CHANNEL_10, 1, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC3, ADC_CHANNEL_12, 2, ADC_SAMPLETIME_47_5);
  adc_ordinary_channel_set(ADC3, ADC_CHANNEL_13, 3, ADC_SAMPLETIME_47_5);
  adc_ordinary_conversion_trigger_set(ADC3, ADC_ORDINARY_TRIG_TMR1CH1,
ADC_ORDINARY_TRIG_EDGE_NONE);
  adc_dma_mode_enable(ADC3, FALSE);
  adc_dma_request_repeat_enable(ADC3, FALSE);
  adc_interrupt_enable(ADC3, ADC_OCCO_INT, TRUE);

  /* adc enable */
  adc_enable(ADC1, TRUE);
  adc_enable(ADC2, TRUE);
  adc_enable(ADC3, TRUE);
  while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC2, ADC_RDY_FLAG) == RESET);
  while(adc_flag_get(ADC3, ADC_RDY_FLAG) == RESET);

  /* adc calibration */
  adc_calibration_init(ADC1);
  while(adc_calibration_init_status_get(ADC1));
  adc_calibration_start(ADC1);
  while(adc_calibration_status_get(ADC1));
  adc_calibration_init(ADC2);
  while(adc_calibration_init_status_get(ADC2));
  adc_calibration_start(ADC2);
  while(adc_calibration_status_get(ADC2));
  adc_calibration_init(ADC3);
  while(adc_calibration_init_status_get(ADC3));
  adc_calibration_start(ADC3);
  while(adc_calibration_status_get(ADC3));}
```

■ 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);
    dma1_trans_complete_flag = 1;
  }
}
/* 获取各个 ADC 的溢出状态信息 */
void ADC1_2_3_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_OCCO_FLAG);
    adc1_overflow_flag++;
  }
  if(adc_flag_get(ADC2, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC2, ADC_OCCO_FLAG);
    adc2_overflow_flag++;
  }
  if(adc_flag_get(ADC3, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC3, ADC_OCCO_FLAG);
    adc3_overflow_flag++;
  }
}
```

■ main 函数代码

```
int main(void)
{
  __IO uint32_t index1 = 0;
  __IO uint32_t index2 = 0;
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* config the system clock */
  system_clock_config();

  /* init at start board */
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  usart1_config(115200);
  gpio_config();
```

```
  tmr1_config();
  dma_config();
  adc_config();
  printf("combine_mode_ordinary_smlt_twoslave_dma5 \r\n");
  tmr_counter_enable(TMR1, TRUE);
  while(dma1_trans_complete_flag == 0);
  tmr_counter_enable(TMR1, FALSE);
  if((adc1_overflow_flag != 0) || (adc2_overflow_flag != 0) || (adc3_overflow_flag != 0))
  {
    /* printf flag when error occur */
    at32_led_on(LED3);
    at32_led_on(LED4);
    printf("error occur\r\n");
    printf("adc1_overflow_flag = %d\r\n",adc1_overflow_flag);
    printf("adc2_overflow_flag = %d\r\n",adc2_overflow_flag);
    printf("adc3_overflow_flag = %d\r\n",adc3_overflow_flag);
  }
  else
  {
    /* printf data when conversion end without error */
    printf("conversion end without error\r\n");
    for(index1 = 0; index1 < 3; index1++)
    {
      for(index2 = 0; index2 < 3; index2++)
      {
        printf("adccom_ordinary_valuetab[%d][%d][0] = 0x%x\r\n",index1, index2,
adccom_ordinary_valuetab[index1][index2][0]);
        printf("adccom_ordinary_valuetab[%d][%d][1] = 0x%x\r\n",index1, index2,
adccom_ordinary_valuetab[index1][index2][1]);
      }
      printf("\r\n");
    }
  }
  at32_led_on(LED2);
  while(1)
  {
  }
}
```

11.4 实验效果

可通过串口打印查看实现效果，最终串口打印的转换数据如下。

ADC 转换数据以字为单位进行传输，传输顺序为：

(ADC2 第一个转换通道<<16) | ADC1 第一个转换通道；

ADC3 第一个转换通道；

(ADC2 第二个转换通道<<16) | ADC1 第二个转换通道；

ADC3 第二个转换通道；

(ADC2 第三个转换通道<<16) | ADC1 第三个转换通道；

ADC3 第三个转换通道

(ADC2 第一个转换通道<<16) | ADC1 第一个转换通道……

串口配置

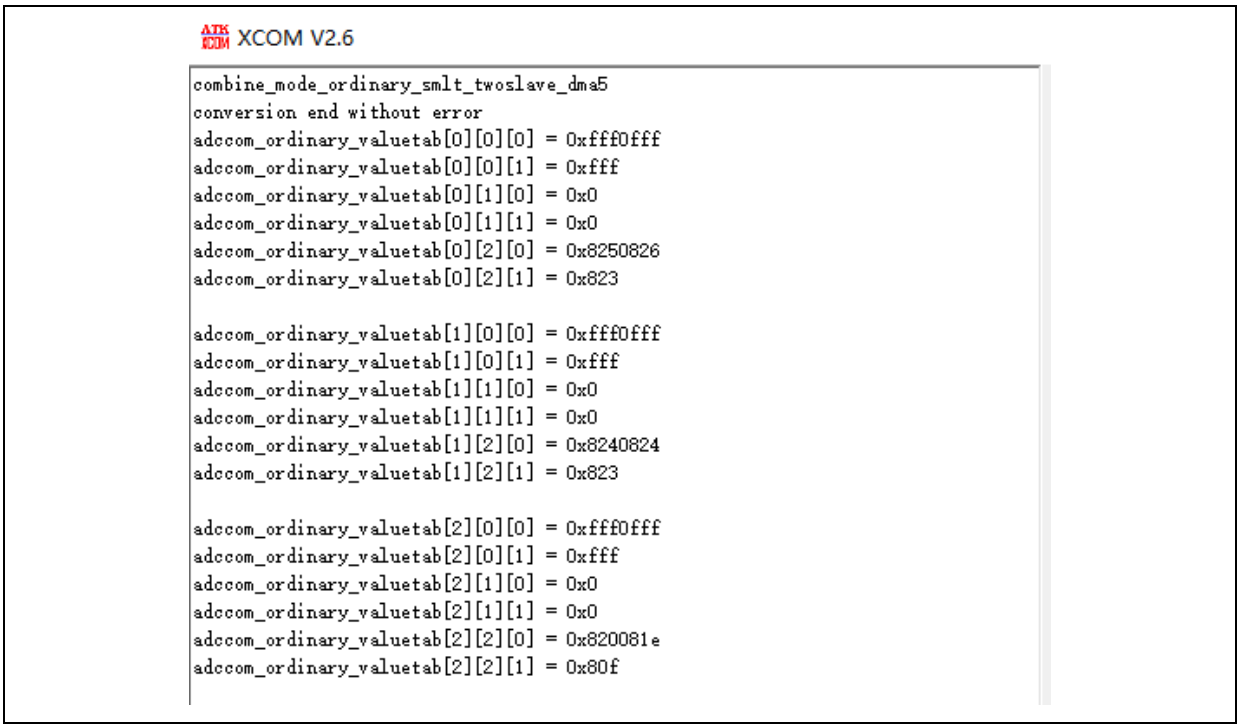
Baud rate：115200

Stop bits：1

Data bits：8

Parity：None

图 25. ADC DMA 模式 5 实验结果



12 案例 ADC 侦测 Vref 电压

12.1 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。

12.2 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

12.3 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);
  nvic_irq_enable(DMA1_Channel1_IRQn, 0, 0);

  dma_reset(DMA1_CHANNEL1);
  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);

  dmamux_enable(DMA1, TRUE);
  dmamux_init(DMA1MUX_CHANNEL1, DMAMUX_DMAREQ_ID_ADC1);

  /* disable dma transfer complete interrupt */
  dma_interrupt_enable(DMA1_CHANNEL1, DMA_FDT_INT, FALSE);
  dma_channel_enable(DMA1_CHANNEL1, TRUE);
}
```

■ ADC 配置函数代码

```
static void adc_config(void)
{
  adc_common_config_type adc_common_struct;
  adc_base_config_type adc_base_struct;
  crm_periph_clock_enable(CRM_ADC1_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(ADC1_2_3_IRQn, 0, 0);

  adc_common_default_para_init(&adc_common_struct);

  /* config combine mode */
  adc_common_struct.combine_mode = ADC_INDEPENDENT_MODE;

  /* config division,adcclk is division by hclk */
  adc_common_struct.div = ADC_HCLK_DIV_4;

  /* config common dma mode,it's not useful in independent mode */
  adc_common_struct.common_dma_mode = ADC_COMMON_DMAMODE_DISABLE;

  /* config common dma request repeat */
  adc_common_struct.common_dma_request_repeat_state = FALSE;

  /* config adjacent adc sampling interval,it's useful for ordinary shifting mode */
  adc_common_struct.sampling_interval = ADC_SAMPLING_INTERVAL_5CYCLES;

  /* config inner temperature sensor and vintrv */
  adc_common_struct.tempervintrv_state = TRUE;

  /* config voltage battery */
  adc_common_struct.vbat_state = FALSE;
  adc_common_config(&adc_common_struct);

  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_resolution_set(ADC1, ADC_RESOLUTION_12B);

  /* config ordinary channel */
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_17, 1, ADC_SAMPLETIME_92_5);

  /* config ordinary trigger source and trigger edge */
  adc_ordinary_conversion_trigger_set(ADC1, ADC_ORDINARY_TRIG_TMR1CH1,
ADC_ORDINARY_TRIG_EDGE_NONE);

  /* config dma mode,it's not useful when common dma mode is use */
  adc_dma_mode_enable(ADC1, TRUE);

  /* config dma request repeat,it's not useful when common dma mode is use */
  adc_dma_request_repeat_enable(ADC1, TRUE);

  /* enable adc overflow interrupt */
  adc_interrupt_enable(ADC1, ADC_OCCO_INT, TRUE);

  /* adc enable */
  adc_enable(ADC1, TRUE);
  while(adc_flag_get(ADC1, ADC_RDY_FLAG) == RESET);

  /* adc calibration */
  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 ADC1_2_3_IRQHandler(void)
{
  if(adc_flag_get(ADC1, ADC_OCCO_FLAG) != RESET)
  {
    adc_flag_clear(ADC1, ADC_OCCO_FLAG);
    adc1_overflow_flag++;
  }
}
```

■ main 函数代码

```
int main(void)
{
```

```
  __IO uint32_t index = 0;
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

  /* config the system clock */
  system_clock_config();

  /* init at start board */
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  usart1_config(115200);
  dma_config();
  adc_config();
  printf("adc1_vref_check \r\n");

  while(1)
  {
    at32_led_toggle(LED2);
    delay_sec(1);
    /* adc1 software trigger start conversion */
    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);
    if(adc1_overflow_flag != 0)
    {
      /* printf flag when error occur */
      at32_led_on(LED3);
      at32_led_on(LED4);
      printf("error occur\r\n");
      printf("adc1_overflow_flag = %d\r\n",adc1_overflow_flag);
    }
  }
}
```

12.4 实验效果

可通过串口打印查看实现效果，测试未独立外接 Vref，并使用 AT-Link 供电，可以看到实际 Vref 电压固定保持 3.32V 左右。

串口配置

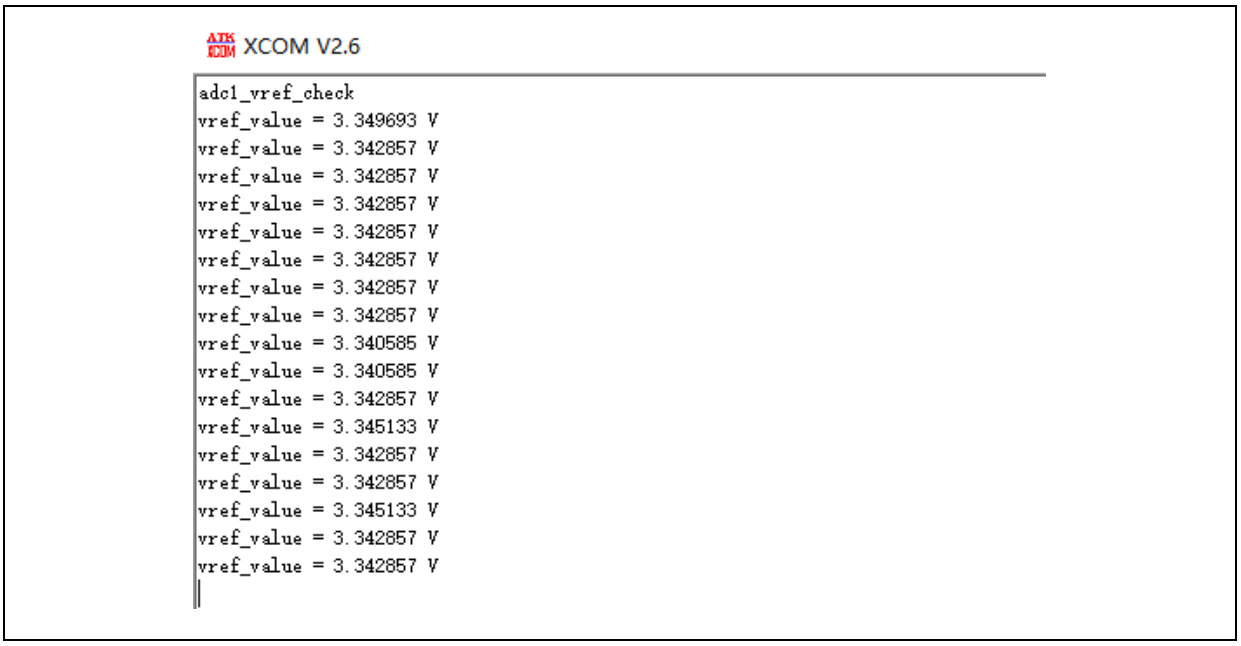
Baud rate：115200

Stop bits：1

Data bits：8

Parity：None

图 26. ADC 侦测 Vref 电压实验结果



13 文档版本历史

表 5. 文档版本历史

日期	版本	变更
2021.9.29	2.0.0	最初版本
2023.3.16	2.0.1	实验效果图优化

重要通知 - 请仔细阅读

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途（及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力产品并非设计或专门用于下列用途的产品：（A）对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）航天应用或航天环境；（D）武器，且/或（E）其他可能导致人身伤害、死亡及财产损害的应用。如果采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，风险及法律责任仍将由采购商单独承担，且采购商应独力负责在前述应用中满足所有法律和法规要求。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。