AN0093
Application Note

AT32F435/437 ADC Application Note

Introduction

The AT32 MCU ADC is peripheral that converts an analog input signal into a digital signal. Its sampling rate is as high as 5.33 MSPS. It has up to 19 channels for sampling and conversion. This application note mainly introduces ADC main features and examples of ADC application.

Applicable products:

Part number	AT32F435 series
	AT32F437 series

Contents

List of tables

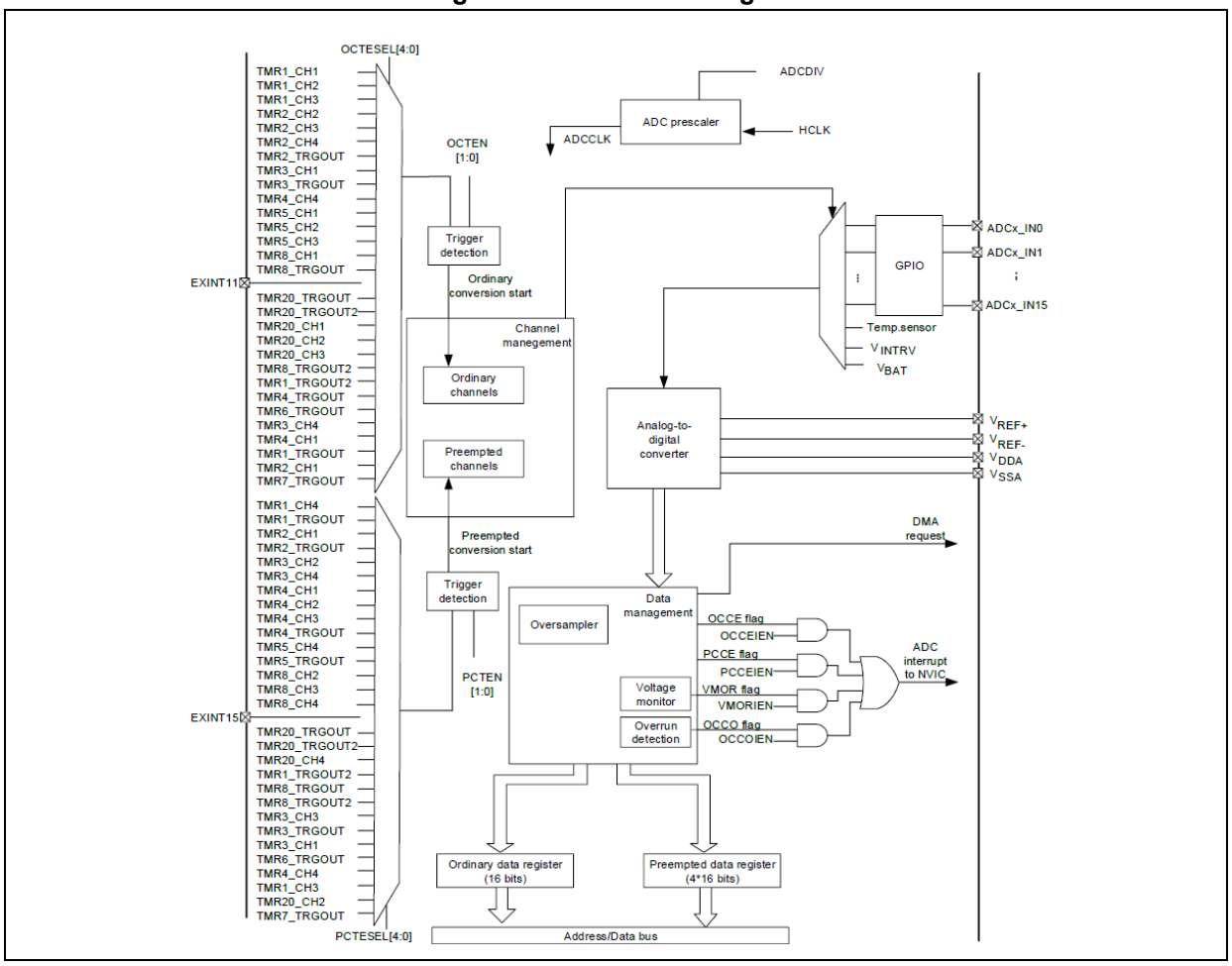
List of figures

1 ADC introduction

The ADC has the following main features:

- Clock and states, which consists of digital and analog clocks
- 12-bit, 10-bit, 8-bit or 6-bit configurable resolution, and configurable sampling period
- Self-calibration for data alignment
- Support multiple modes and flexible combination of modes to meet requirements of various applications
- Ordinary and preempted channels with different priorities
- Multiple independent trigger sources, including TMR, EXINT and software trigger
- Data management, including data alignment and preempted channel offset
- End of conversion by software when the ADC is not powered off
- Oversampling supported by both ordinary and preempted channels
- Voltage monitoring based on the conversion result
- Interrupt and status events: multiple flags of ADC status and some of which has interrupt function
- Read the converted data by using DMA or CPU
- Support master/slave mode, which can be configured as simultaneous mode, alternate preempted trigger mode and regular shift mode, and support single/dual slave

Figure 1. ADC1 block diagram



2 ADC function overview

2.1 Clock and states

2.1.1 Introduction

ADC clocks include a digital clock and an analog clock, which can be enabled by setting the ADCxEN bit in the CRM_APB2EN register.

- Digital clock, which is the PCLK2 derived from HCLK frequency division and used for the digital module.
- Analog clock, which is the ADCCLK derived from ADC frequency division and used for the analog module.

2.1.2 Software design

ADC clock enable is implemented by an independent function, and the code is as follows:

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

After the ADC clock is enabled, the software can start to configure the ADC.

ADC prescaler configuration is implemented by the ADC common structure, and the code is as follows:

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

It is used for the configuration of ADC analog clock (i.e., the ADCCLK derived from divided HCLK, ADCCLK=HCLK/ div).

Notes:

1) The ADCCLK is derived from divided HCLK, and it must be less than 80 MHz.

2) The ADCCLK frequency must be lower than PCLK2.

3) ADC1, ADC2 and ADC3 have an independent clock enable bit respectively, while the ADC common structure does not have an independent clock enable bit, so that it is enabled automatically when any other ADCx clock is enabled.

4) The ADC analog module is powered by setting the ADCEN bit in the ADC_CTRL2 register, and it is not affected by the ADC clock status. Typically, if the ADCEN bit is not cleared when the system enters Deepsleep mode, the ADC analog module still consume power.

5) There is a waiting period for ADC power-on, and the subsequent trigger operation is performed after the RDY flag is set.

2.2 Resolution, sampling and conversion

2.2.1 Introduction

The ADC supports 12-bit, 10-bit, 8-bit and 6-bit configuration resolution.

The ADC sampling time can be selected as 2.5, 6.5, 12.5, 24.5, 47.5, 92.5, 247.5 and 640.5 cycles.

The ADC can read channel data through sampling and conversion.

The sampling is prior to conversion. In the sampling period, select the channel to be converted, and

charge the ADC internal sampling capacitor by external power for the specified sampling period.

The conversion starts automatically after completion of sampling. The ADC uses successive approximation conversion mode to guarantee the accuracy of converted data. A single one conversion time is calculated with the following formula:

A single one conversion time (ADCCLK period) = sampling time + resolution bits +0.5

Example:

If the CSPTx selects 6.5 period and CRSEL selects 10-bit resolution, then one conversion needs 6.5+10 + 0.5 = 17 ADCCLK periods.

2.2.2 Software interface

ADC resolution configuration is implemented by an independent function, and the code is as follows:

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

Note: The ADC self-calibration is only supported in 12-bit resolution, and resolution switching can be performed after completion of calibration.

ADC sampling period is implemented by an independent function, and the code is as follows:

```
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);
```

Notes:

The sampling period is configurable for different channels;

It is recommended to increase the sampling period appropriately to avoid overflow when the ordinary channel data is obtained by interrupt or polling method;

It is recommended to increase the sampling period appropriately under permitted conditions to avoid inaccurate converted data caused by insufficient charging.

2.3 Self-calibration

2.3.1 Introduction

The ADC has self-calibration function. The self-calibration command is executed by software to calculate a calibration value, and then the value is sent back to the ADC automatically without software intervention to compensate the ADC deviation to guarantee accurate converted data.

The calibration value can be obtained by the following means:

- Software sends self-calibration command and then the hardware perform calculation automatically. The calculated calibration value is stored in the ADC->CALVAL register.
- Manually set the calibration value by software, and this value is stored in the ADC->CALVAL register.

Self-calibration process:

- Select 12-bit and enable ADC;
- Wait until the RDY flag is set;

- Execute calibration initialization command and wait until the completion of initialization;
- Execute calibration command and wait until the completion of calibration;
- Switch to the desired resolution;
- Wait until the ADC RDY flag is set;
- Complete the above procedures, and then trigger ADC conversion.

2.3.2 Software design

The complete calibration and resolution configuration is implemented by combined commands, which can be divided into two types as follows.

The code to implement self-calibration is as follows:

```
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);
```

The code to implement self-calibration based on empirical value is as follows:

```
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);
```

Note:

Storing the calibration value does not set the OCCE flag nor generate an interrupt or DMA request;

The ADC self-calibration is only supported in 12-bit resolution, and resolution switching can be performed after completion of calibration.

2.4 Conversion sequence management

2.4.1 Introduction

Sequence mode

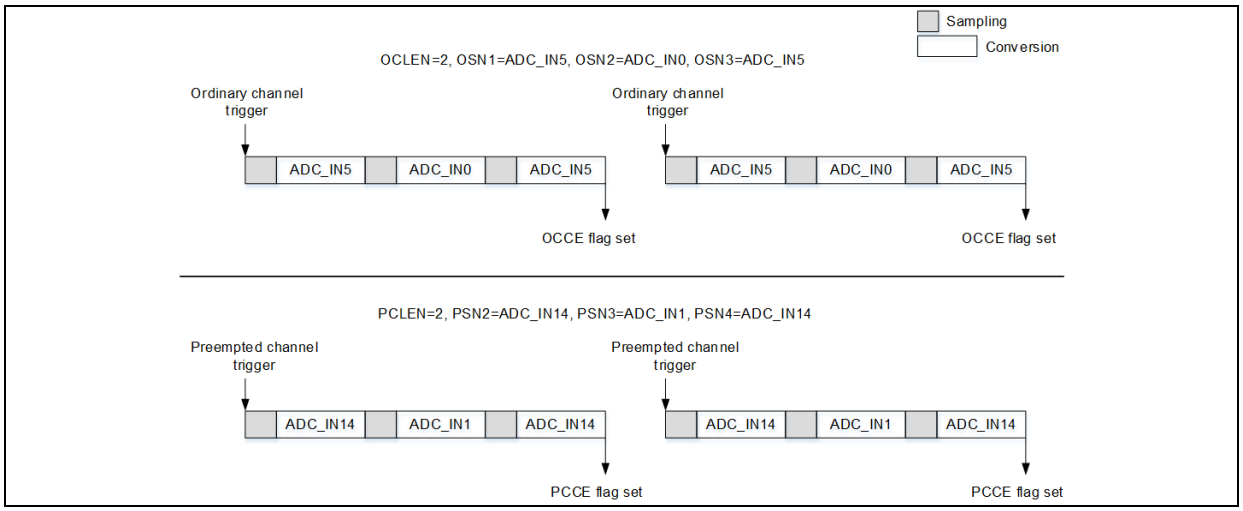
The ADC supports sequence mode. When the sequence mode is enabled, a single trigger event enables the conversion of a group of channels in order

The ADC_OSQx register is used to define the sequence of ordinary channels, and ordinary channels start converting from the QSN1. The ADC_PSQ register is used to define the sequence of preempted channels, and preempted channels start converting from the PSNx (x=4-PCLEN).

Example of preempted channel conversion:

ADC_PSQ[21:0] = 10 00110 00101 00100 00011, and the scan conversion follows the sequence of CH4, CH5, CH6, not CH3, CH4, CH5.

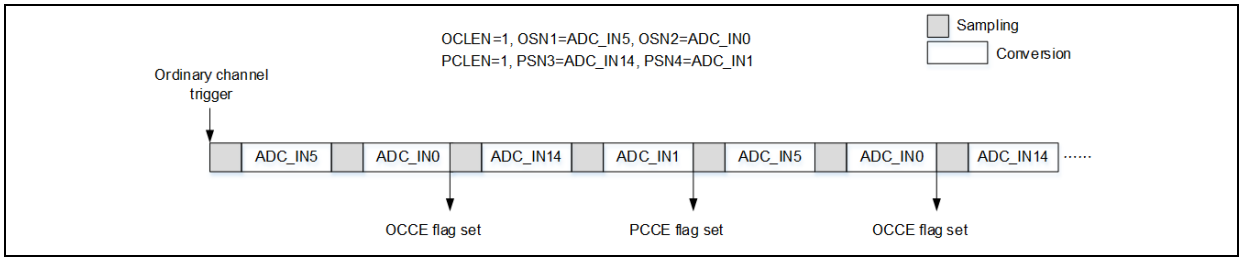
Figure 2. Sequence mode



Repetition mode

The ADC supports repetition mode. In this mode, when a trigger signal is detected, the ordinary channels will be converted repeatedly.

Figure 3. Repetition mode + automatic preempted group conversion mode



Partition mode

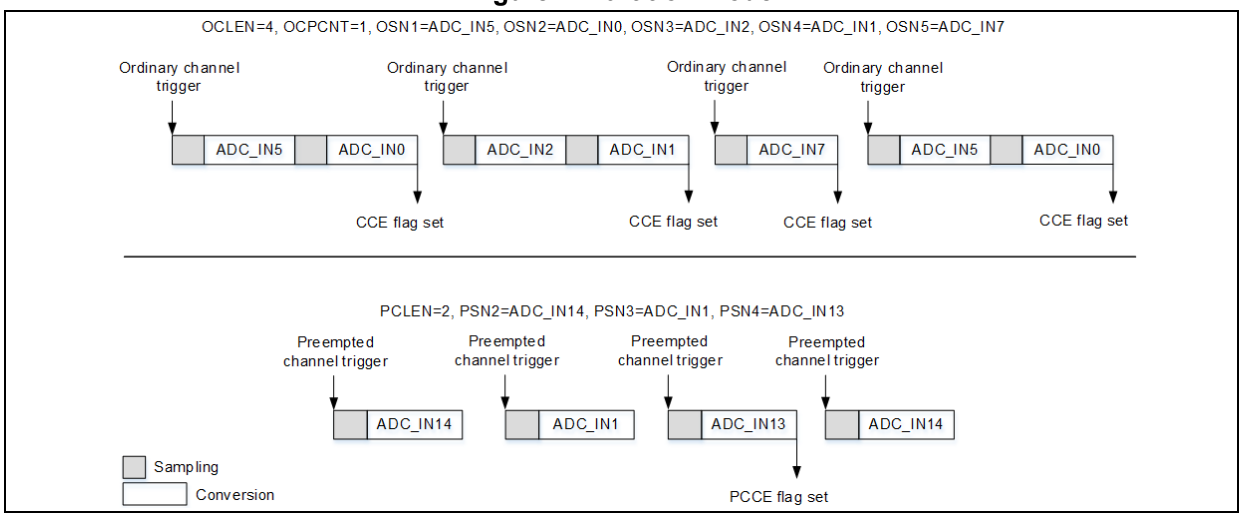
The ADC supports partition mode.

In this mode, the ordinary group conversion sequence length is divided into a smaller sub-group. A single trigger event will enable the conversion of all the channels in the sub-group. Each trigger event selects different sub-group in order.

In this mode, the preempted group conversion sequence length is divided into a sub-group with

only one channel. A single one trigger event will enable the conversion of all the channels in the sub-group. Each trigger event selects different sub-group in order.

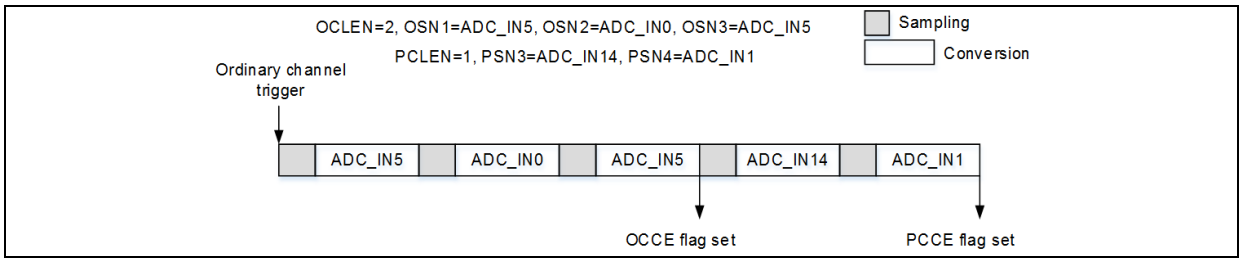
Figure 4. Partition mode



Automatic preempted group conversion mode

The ADC supports automatic preempted group conversion mode. In this mode, once the ordinary channel conversion is over, the preempted group will continue its conversion automatically.

Figure 5. Preempted group auto conversion mode



2.4.2 Software design

The ADC sequence mode and repetition mode are configured by the ADC base structure, and the code is as follows:

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

Notes:

The sequence mode can be configured for both ordinary and preempted channel groups;

The repetition mode is valid for ordinary channel group only;

The repetition mode cannot work with the partition mode;

The repetition mode can work with the automatic preempted group conversion mode to realize repetitive conversion of ordinary and preempted channels in order.

The ADC partition mode is configured by an independent function, and the code is as follows:

```
/* 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);
```

Notes:

The partition mode can be configured for both ordinary and preempted channel groups;

The sub-group length of preempted channel group in partition mode is not configurable, and it is fixed to one single channel;

The partition mode, repetition mode and automatic preempted group conversion mode cannot work together; the partition mode should be configured for the ordinary channels and preempted channels respectively.

The preempted group auto conversion mode is configured by an independent function, and the code is as follows:

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

Notes:

The preempted group auto conversion mode is valid for preempted channel group only;

The preempted group auto conversion mode cannot work with the partition mode.

2.5 Channels with different priorities

2.5.1 Introduction

The ADC conversions are divided into two groups: ordinary channels and preempted channels.

Ordinary channel

It is usually used to perform regular data conversion in the specified order, and supports up to 16 channels. It has no preemptive function.

Preempted channel

It is usually used to perform urgent data conversion in the specified order, and supports up to 4 channels. It has preemptive function, which means that if the preempted channel trigger occurs during the ordinary channel conversion, then the ordinary channel conversion is interrupted, giving the priority to the preempted channel, and the ordinary channel continues its conversion at the end of the preempted channel conversion.

2.5.2 Software design

The ordinary channel configuration (including the number of channels, conversion order and sampling period) is implemented by a function, and the code is as follows:

```
/* 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);
```

The preempted channel configuration (including the number of channels, conversion order and sampling period) is implemented by a function, and the code is as follows:

```
/* 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);
```

Notes:

The sampling period is configurable for different channels;

One channel can be repeatedly arranged for conversion;

In sequence mode, the ordinary channels start converting from OSN1, and preempted channels start converting from PSNx (x=4-PCLEN).

2.6 Multiple independent trigger sources

2.6.1 Introduction

ADC conversion can be triggered by software write operation to the OCSWTRG and PCSWTRG bits of the ADC_CTRL2 register, or by an external event. The external events include timer and pin triggers, with configurable trigger polarity (edge trigger forbidden, rising edge, falling edge or any edge).

Table 1. Trigger sources for ordinary channels

OCTESEL	Trigger source	OCTESEL	Trigger source
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	Reserved
01110	TMR8_TRGOUT event	11110	TMR7_TRGOUT event
01111	EXINT line11 External pin	11111	Reserved

Table 2. Trigger sources for preempted channels

PCTESEL	Trigger source	PCTESEL	Trigger source
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	Reserved
01110	TMR8_CH4 event	11110	TMR7_TRGOUT event
01111	EXINT line15 External pin	11111	Reserved

2.6.2 Software design

The software write operation to the specified register is implemented by an independent function, and the code is as follows:

```
/* 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);
```

Once the ADC is ready, the software can execute “adc_ordinary_software_trigger_enable(ADC1, TRUE);/ adc_preempt_software_trigger_enable (ADC1, TRUE);” to trigger ordinary/preempted channels.

The external trigger event is set by an independent function, and the code is as follows:

```
/* 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);
```

Once the ADC is ready, the TMR1CH1 rising edge triggers ordinary channel conversion, and the TMR3CH4 rising edge triggers preempted channel conversion.

Notes:

The trigger interval should be longer than the channel conversion time, and the trigger for the same channel during conversion is ignored.

For the software write operation to the specified register, the corresponding trigger polarity must be configured as "edge trigger forbidden".

The preempted channel has the highest priority, and it starts to convert immediately after triggering, regardless of the ordinary channel conversion.

Regular trigger has a memory function, which means that the ordinary channel conversion interrupted by the preempted ones will be recorded and restart as soon as the preempted conversion is completed.

In the master/slave mode (several ADCs), the corresponding trigger polarity must be configured as "edge trigger forbidden".

2.7 Data management

2.7.1 Introduction

ADC has dedicated data registers. At the end of the conversion of the ordinary group, the converted value is stored in the ADC_ODT register. Once the preempted group conversion ends, the converted data of the preempted group is stored in the ADC_PDTx register. Data management includes data alignment and preempted channel data offset.

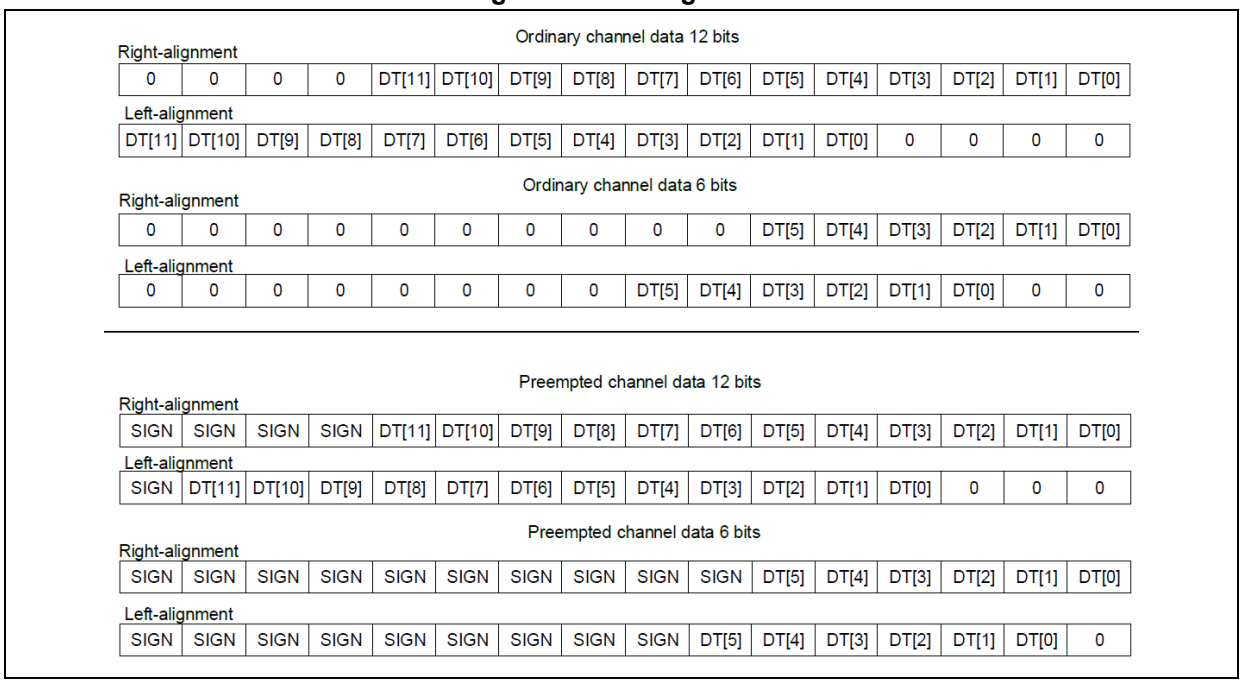
Data alignment

Data can be configured as right-aligned or left-aligned. The data is aligned on a half-word basis expect when the CRSEL is set to 6-bit (in this case, the data are aligned on a byte basis).

Preempted channel data offset

The converted data of the preempted group is decreased by the offset written in the ADC_PCDTOx register. Thus, the result may be a negative value, marked by SIGN.

Figure 6. Data alignment



2.7.2 Software design

The data alignment is implemented by the ADC base structure, and the code is as follows:

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

The preempted channel data offset configuration is implemented by an independent function, and the code is as follows:

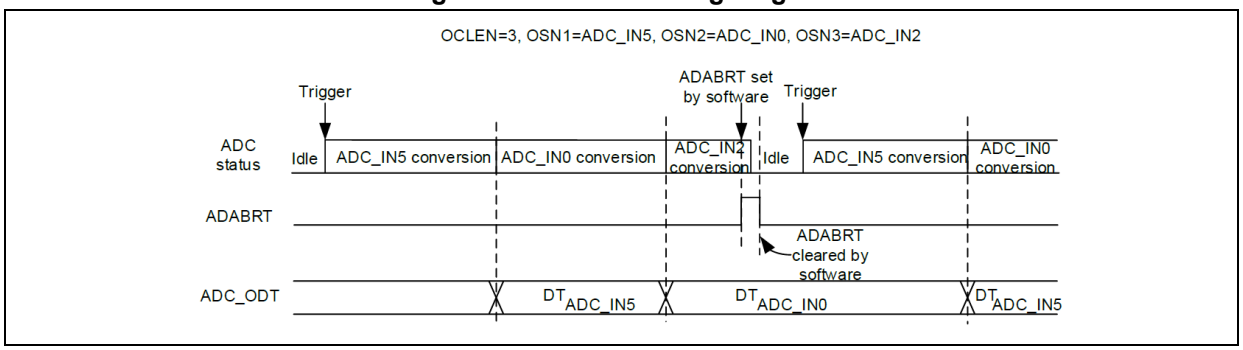
```
/* set preempt channel's conversion value offset */
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_1, 0x000);
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_2, 0x111);
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_3, 0x202);
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_4,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

2.8 End of conversion

2.8.1 Introduction

The ADABRT bit in the ADC_CTRL2 register can be used to stop ADC conversions. At the end of the conversions, the conversion sequence returns to the first channel. This allows the user to configure a new channel sequence, and the ADC starts conversions from the beginning based on the new order when a trigger event occurs.

Figure 7. ADABRT timing diagram



2.8.2 Software design

The conversion stop is implemented by an independent function, and the code is as follows:

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

Notes:

The ADC conversion abort command is valid for both ordinary and preempted channel conversions.

The ADC conversion abort has similar functions to ADC re-power, but it does not power off ADC, therefore the ADC wakeup time is not required.

After the ADC conversion abort command is executed, wait for the ADABRT bit to be cleared by hardware before the subsequent conversion by trigger events.

2.9 Oversampling

2.9.1 Introduction

The ADC has oversampling function. A single oversampling converted data can be done through multiple conversions of the same channel in which the cumulative converted data is averaged.

- Oversampling ratio is selected through the OSRSEL bit in the ADC_OVSP register. This bit is used to specify the oversampling multiple;
- Oversampling shift is selected through the OSSSEL bit in the ADC_OVSP register, which is performed by right shift

If the averaged data is greater than 16 bits, then only pick up the right-aligned 16-bit data and put them into a 16-bit data register.

When using oversampling conversion mode, the data alignment and preempted data shift are ignored, and data must be right aligned.

Table 3. Correlation between maximum cumulative data, oversampling multiple and shift digits

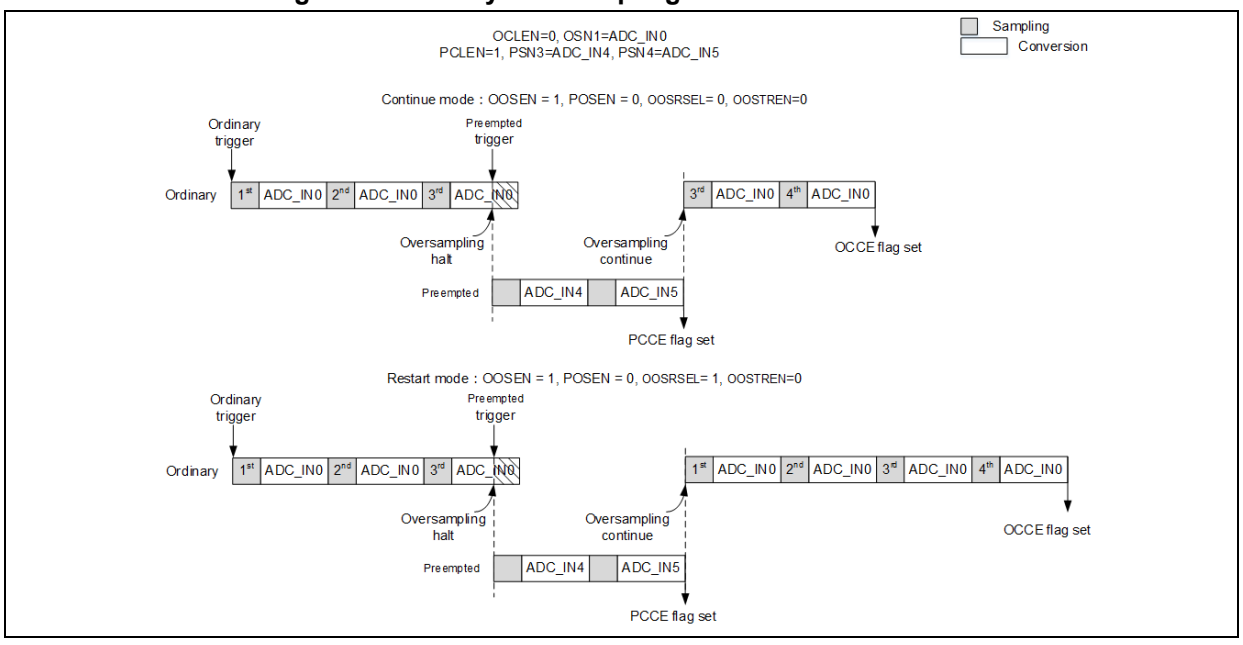
Oversampling multiple	2x	4x	8x	16x	32x	64x	128x	256x
Max. cumulative data	0x1FFE	0x3FFC	0x7FF8	0xFFF0	0x1FFE0	0x3FFC0	0x7FF80	0xFFF00
No shift	0x1FFE	0x3FFC	0x7FF8	0xFFF0	0xFFE0	0xFFC0	0xFF80	0xFF00
Shift 1 digit	0x0FFF	0x1FFE	0x3FFC	0x7FF8	0xFFF0	0xFFE0	0xFFC0	0xFF00
Shift 2 digits	0x0800	0x0FFF	0x1FFE	0x3FFC	0x7FF8	0xFFF0	0xFFE0	0xFFC0
Shift 3 digits	0x0400	0x0800	0x0FFF	0x1FFE	0x3FFC	0x7FF8	0xFFF0	0xFFE0
Shift 4 digits	0x0200	0x0400	0x0800	0x0FFF	0x1FFE	0x3FFC	0x7FF8	0xFFF0
Shift 5 digits	0x0100	0x0200	0x0400	0x0800	0x0FFF	0x1FFE	0x3FFC	0x7FF8
Shift 6 digits	0x0080	0x0100	0x0200	0x0400	0x0800	0x0FFF	0x1FFE	0x3FFC
Shift 7 digits	0x0040	0x0080	0x0100	0x0200	0x0400	0x0800	0x0FFF	0x1FFE
Shift 8 digits	0x020	0x0040	0x0080	0x0100	0x0200	0x0400	0x0800	0x0FFF

Ordinary oversampling restart mode

The OOSRSEL bit is used to resume ordinary oversampling mode.

- OOSRSEL=0: continuous conversion mode. Ordinary group of channels, after being interrupted by preempted group of channels during oversampling, will retain the converted data and resume from the last interrupted ordinary conversion.
- OOSRSEL=1: restart mode. Ordinary group of channels, after being interrupted by preempted group of channels during oversampling, will be reset and restart the ordinary conversion.

Figure 8. Ordinary oversampling restart mode selection



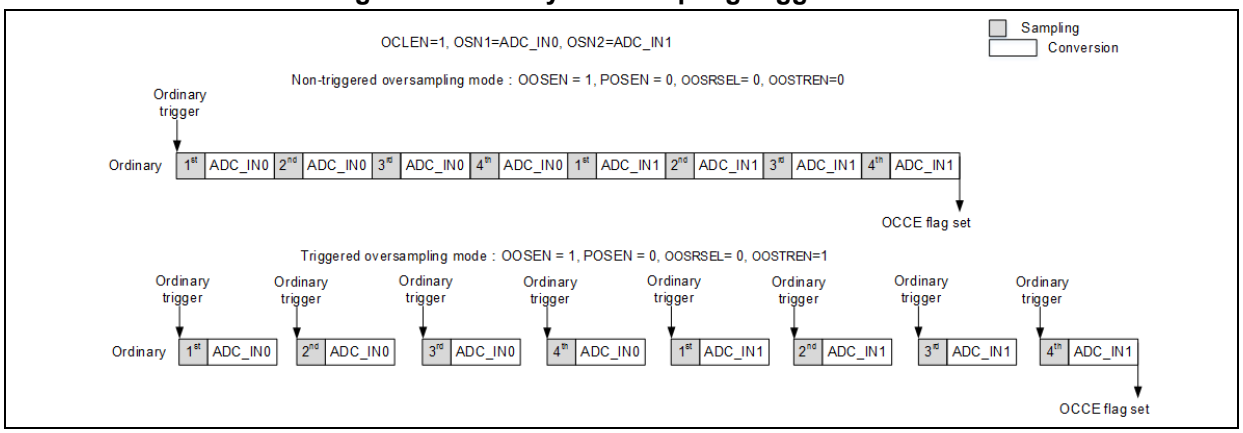
Ordinary oversampling trigger mode

The ordinary oversampling trigger mode can be enabled by setting the OOSTREN bit.

- OOSTREN=0: Disabled; only one trigger is needed for all oversampling conversions.
- OOSTREN=1: Enabled; each oversampling conversion needs a trigger.

In this mode, once the ordinary conversion is interrupted by preempted group of channels, it is necessary to re-trigger ordinary group of channels before resuming the ordinary oversampling.

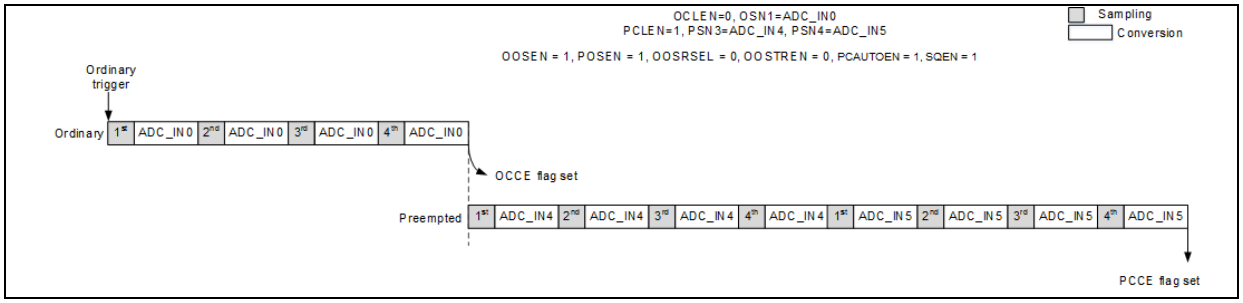
Figure 9. Ordinary oversampling trigger mode



Oversampling of preempted group of channels

It is possible to use both the preempted oversampling and ordinary oversampling simultaneously or individually. The oversampling of the preempted group of channels does not affect the ordinary oversampling modes.

Figure 10. Oversampling of preempted group of channels



2.9.2 Software design

The oversampling ratio, oversampling shift and oversampling enable are implemented by independent functions, and the code is as follows:

```
/* 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);
```

The ordinary oversampling restart mode is implemented by an independent function, and the code is as follows:

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

The ordinary oversampling trigger mode is implemented by an independent function, and the code is as follows:

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

2.10 Voltage monitoring

2.10.1 Introduction

The ADC has monitoring function for the correlation between input voltage and threshold.

The VMOR bit will be set if the converted result is beyond the high threshold (ADC_VMHB[11:0]) or less than the low threshold (ADC_VMLB[11:0]).

The VMSGEN bit is used to enable voltage monitoring on either a single channel or all the channels. The VMCSEL bit is used to select a specific channel that requires voltage monitoring.

2.10.2 Software design

The voltage monitoring on a single channel is implemented by an independent function, and the code is as follows:

```
/* 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);
```

The voltage monitoring on all channels is implemented by an independent function, and the code is as follows:

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

Note:

Voltage monitoring is based on the comparison result between the original converted data and the 12-bit voltage monitor boundary register, irrespective of the resolution, preempted channel data offset and data alignment.

When using an oversampler, voltage monitoring is based on the comparison result between the 16-bit registers (ADC_VMHB[15:0] and ADC_VMLB[15:0]) and the oversampled data.

2.11 Interrupts and status flag

2.11.1 Introduction

ADC has multiple interrupts and status flags, which should be considered for program design.

- **ADC ready to conversion flag** (RDY)

  It indicates the ADC status. It can be read only and cannot be cleared by software nor generate an interrupt.

  It is set by hardware when the ADC is powered. It is possible to perform calibration and trigger conversion after the RDY flag is set.

- **Ordinary channel conversion overflow** (OCCO)

  It indicates the ADC conversion data overflow. It is cleared by software and can generate an interrupt.
  There is no independent bit to enable overflow detection. The overflow detection is applicable to the case of DMA transfer enable or EOCSFEN =1. This bit is set when the previous converted data is not read while the next converted data is generated. After this flag is cleared,

the conversion can be resumed by the following means:

1) In non-combined mode: when this flag is set, the current ongoing conversion stops and the conversion sequence is not cleared; therefore, clear the flag and then trigger conversion, without the need of ADC reset.

2) In combined mode: when this flag is set, the current ongoing conversion stops and the conversion sequence remains unchanged, but the synchronization may lost; therefore, reset each ADC and then trigger conversion.

- **Ordinary channel conversion start fag** (OCCS)

  It indicates the start of ordinary channel conversion. It is cleared by software and cannot generate an interrupt.

- **Preempted channel conversion start flag** (PCCS)

  It indicates the start of preempted channel conversion. It is cleared by software and cannot generate an interrupt.

- **Preempted channel conversion end flag** (PCCE)

  It indicates the end of preempted channel conversion. It is cleared by software and can generate an interrupt. It is set after the preempted channel conversion is completed. This flag is usually used to read the converted data of the preempted group of channels.

- **Ordinary channel conversion end flag** (OCCE)

  It indicates the end of ordinary channel conversion. It is cleared by software or by reading the ADC_ODT register, and it can generate an interrupt.

  After the ordinary channel conversion is completed, this flag can be used to read the converted data of the ordinary group of channels (EOCSFEN = 1).

Note: DMA reading converted data clears the OCCE flag simultaneously. Therefore, the OCCE flag should be disabled when using DMA.

- **Voltage monitoring out-of-range flag** (VMOR)

  It indicates whether the voltage is beyond the programmed threshold. It is cleared by software and can generate an interrupt.

  This flag is set when the ADC channel converted data is beyond the programmed threshold, and it is usually used to monitor the channel voltage.

2.11.2 Software design

The interrupt enable is implemented by an independent function, and the code is as follows:

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

The status flag get is implemented by an independent function, and the code is as follows:

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

The status flag clear is implemented by an independent function, and the code is as follows:

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

Note:

The three ADCs of AT32F435 MCU share one interrupt vector.

The ADC_CSTS register contains the ADC status flag mapping that is read-only and cannot be used to clear any flag.

2.12 Data read

2.12.1 Introduction

The ADC has multiple methods to read the converted data according to the channel type and combined mode.

- **Preempted channel data read access by CPU**

The preempted channel does not support DMA capacities; therefore, in all combined modes, the preempted channel data is obtained by CPU reading the ADC_PDTx register.

- **Ordinary channel data read access by CPU** (in non-combined mode)

It is applicable in non-combined mode. Set the EOCSFEN bit in the ADC_CTRL2 register by software to set the OCCE flag every time the ADC_ODT register is updated. The converted data is read by software through the OCCE flag.

- **Ordinary channel data read access by DMA** (in non-combined mode)

In non-combined mode, the ordinary channel data is stored in an independent ADC data register. Set the OCDMAEN and OCDRCEN bits to generate a DMA request each time the ADC_ODT register is updated. The converted data is read by DMA each time the DMA request is received.

- **Ordinary channel data read access by DMA** (in combined mode)

In combined mode, the ordinary channel data is stored in the ADC_CODT register. The MSDMASEL bit of the ADC_CCTRL register can be used to select from five DMA transfer modes. As long as the MSDMASEL bit is set (≠0), the ADC1 DMA channel is used to generate a DMA request each time the data is ready.

Table 4. Master/slave DMA mode

MSDMASEL	Master/slave mode	DMA request	ADC_CODT[31:0]
001	Single slave	1st	16 bit 0, ADC1_ODT[15:0]
		2nd	16 bit 0, ADC2_ODT [15:0]
		3rd	16 bit 0, ADC1_ODT [15:0]
	Dual slave	1st	16 bit 0, ADC1_ODT[15:0]
		2nd	16 bit 0, ADC2_ODT [15:0]
		3rd	16 bit 0, ADC3_ODT[15:0]
		4th	16 bit 0, ADC1_ODT [15:0]
010	Single slave	1st	ADC2_ODT[15:0], ADC1_ODT[15:0]
		2nd	ADC2_ODT[15:0], ADC1_ODT[15:0]
	Dual slave	1st	ADC2_ODT[15:0], ADC1_ODT[15:0]
		2nd	ADC1_ODT[15:0], ADC3_ODT[15:0]
		3rd	ADC3_ODT[15:0], ADC2_ODT[15:0]
		4th	ADC2_ODT[15:0], ADC1_ODT[15:0]
011	Single slave	1st	16 bit 0, ADC2_ODT[7:0], ADC1_ODT[7:0]
		2nd	16 bit 0, ADC2_ODT[7:0], ADC1_ODT[7:0]
	Dual slave	1st	16 bit 0, ADC2_ODT[7:0], ADC1_ODT[7:0]
		2nd	16 bit 0, ADC1_ODT[7:0], ADC3_ODT[7:0]
		3rd	16 bit 0, ADC3_ODT[7:0], ADC2_ODT[7:0]
		4th	16 bit 0, ADC2_ODT[7:0], ADC1_ODT[7:0]

MSDMASEL	Master/slave mode	DMA request	ADC_CODT[31:0]
100	Dual slave	1st	8 bit 0, ADC3_ODT[7:0], ADC2_ODT[7:0], ADC1_ODT[7:0]
		2nd	8 bit 0, ADC3_ODT[7:0], ADC2_ODT[7:0], ADC1_ODT[7:0]
101	Dual slave	1st	ADC2_ODT[15:0], ADC1_ODT[15:0]
		2nd	16 bit 0, ADC3_ODT[15:0]
		3rd	ADC2_ODT[15:0], ADC1_ODT[15:0]

Figure 11. Correlation between DMA mode and ADC mode

DMA_Mode	Applicable ADC modes	
	Single slave	Dual slave
Mode1	Ordinary simultaneous mode, ordinary shift mode	Ordinary simultaneous mode, ordinary shift mode
Mode2	Ordinary simultaneous mode, ordinary shift mode	Ordinary shift mode
Mode3	6-bit/8-bit resolution ordinary simultaneous mode 6-bit/8-bit resolution ordinary shift mode	6-bit/8-bit resolution ordinary shift mode
Mode4	None	6-bit/8-bit resolution ordinary simultaneous mode 6-bit/8-bit resolution ordinary shift mode
Mode5	None	Ordinary simultaneous mode, ordinary shift mode

2.12.2 Software design

CPU reading the preempted channel data is implemented by an independent function, and the code is as follows:

```
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 reading the ordinary channel data (in non-combined mode) is implemented by an independent function, and the code is as follows:

```
while(adc_flag_get(ADC1, ADC_OCCE_FLAG) == RESET);
*(p_adc1_ordinary++) = adc_ordinary_conversion_data_get(ADC1);
```

DMA reading the ordinary channel data (in non-combined mode) is implemented by an independent function, and the code is as follows:

```
/* 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 reading the ordinary channel data (in combined mode) is implemented by an independent function, and the code is as follows:

```
/* 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);
```

Note:

For CPU reading the ordinary channel data, the sampling period should be long enough to avoid overflow.

Since three ADCs share one interrupt vector, the interrupt service function should be simple so that the interrupt can be responded in time to avoid data overflow.

In combined mode, CPU reading the converted data of ordinary channels is not allowed.

In combined mode, select a DMA mode appropriately.

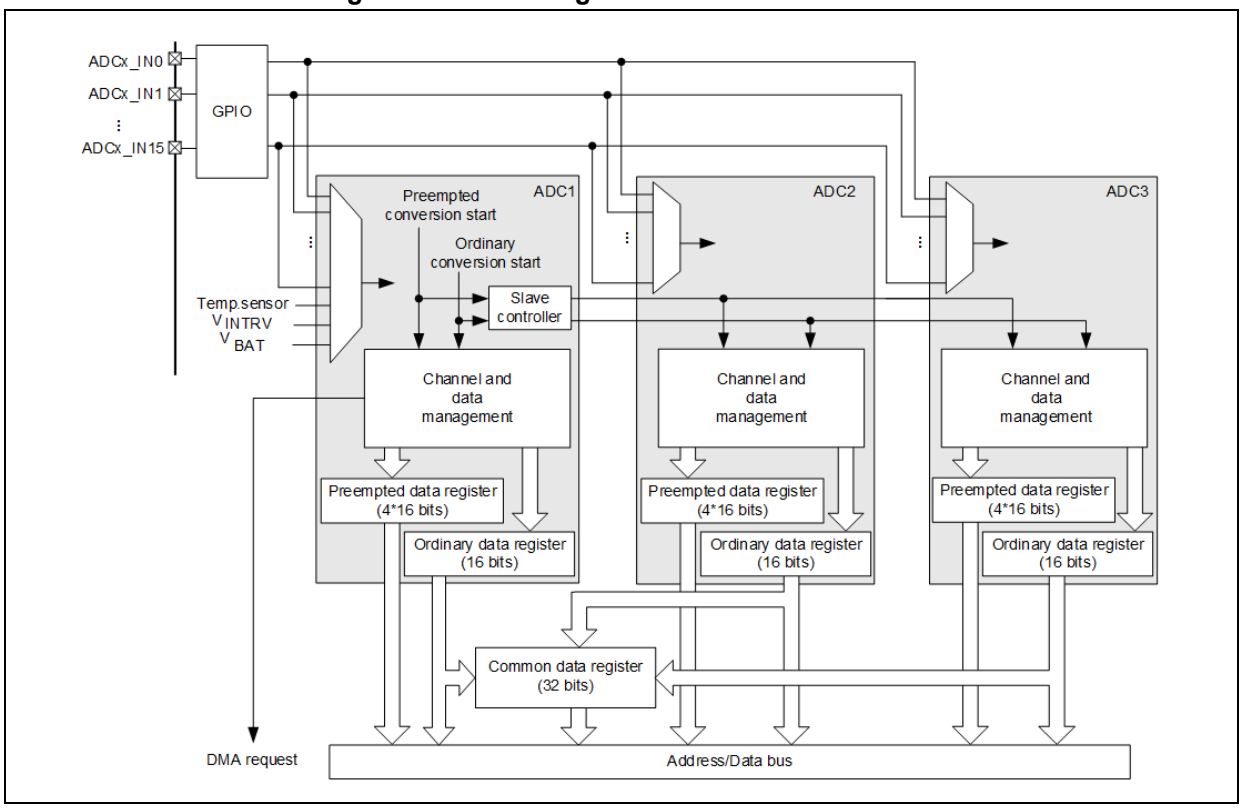
2.13 Master/slave mode

2.13.1 Introduction

If Master/Slave mode is enabled, the master is triggered to work with the slave to do the channel conversion. The ADC_ODT register is used as a single interface obtaining the ordinary channel converted data of master/salve ADC.

In a single master/slave mode, ADC1 acts as a master while ADC2 as a slave, and ADC3 behaves independently. In dual master/slave mode, ADC1 acts as a master, while both ADC2 and ADC3 act as slaves.

Figure 12. Block diagram of master/slave mode



■ Simultaneous mode

The combined ordinary/preempted/ordinary+preempted mode can be selected. If this mode is enabled, the master is triggered so that both the master and the slave convert the corresponding channels simultaneously. In this mode, it is required to configure the same sampling time and the same sequence length for the master and slave to avoid the loss of data due to the lack of synchronization.

Figure 13. Regular simultaneous mode

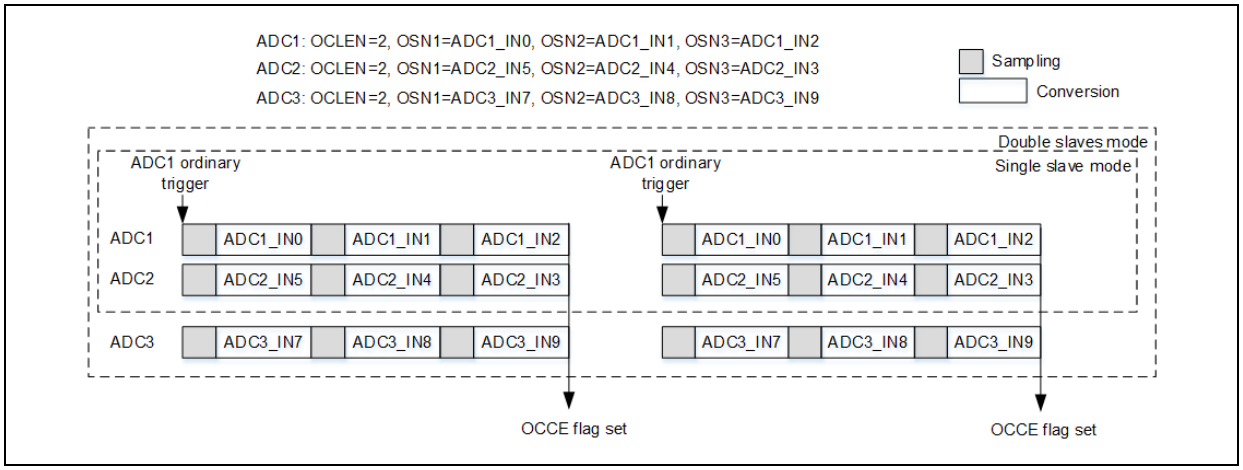
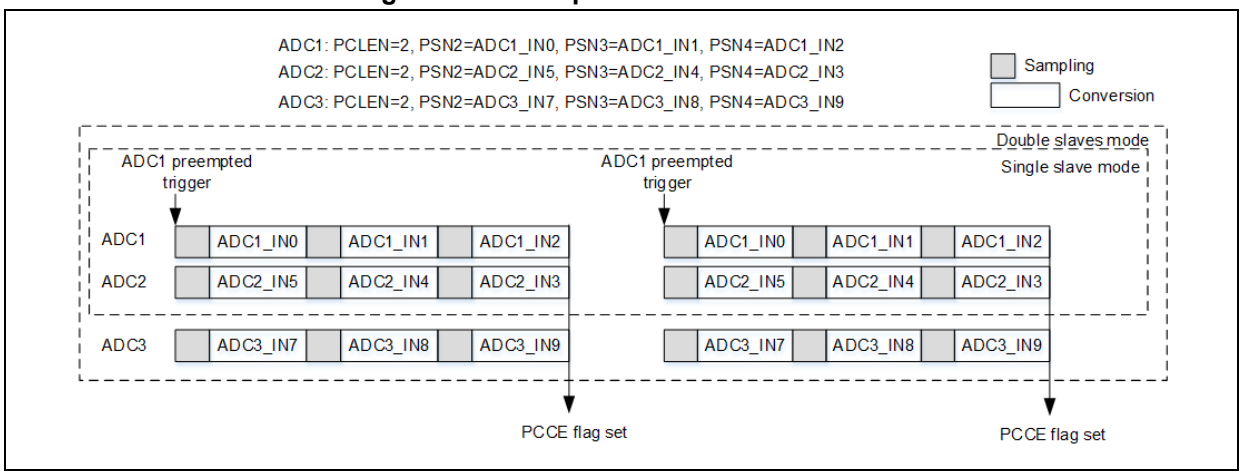


Figure 14. Preempted simultaneous mode



Note:

The slave trigger is not allowed in the simultaneous mode;

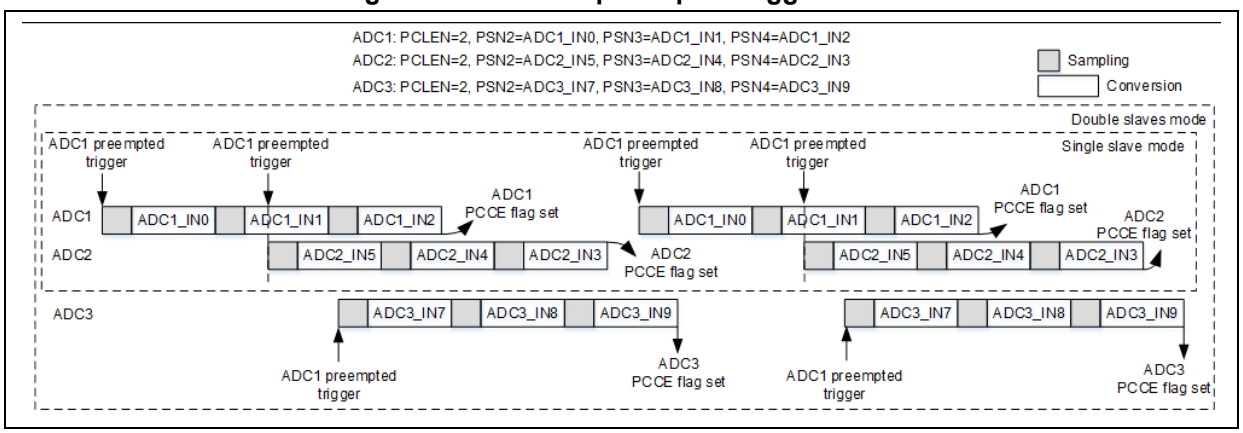
In the simultaneous mode, the trigger interval is greater than the conversion period of any group of channels.

The same channel is not allowed to be sampled by several ADCs simultaneously. Do not put the same channel in the same sequence location of different ADCs.

■ Alternate trigger mode

The alternate trigger mode is applicable to the preempted group of channels or independently used in combination with regular simultaneous mode. Once the alternate preempted trigger mode is enabled, the preempted channels of the master can be triggered continuously so that the master/slave ADCs convert the preempted channels alternately.

Figure 15. Alternate preempted trigger mode



■ Shift mode

The shift mode is applicable to ordinary group of channels and can be used independently only. After a master trigger occurs, the conversion interval between ADCs is based on the programmed shift length.

The shift length is set by the ASISEL bit in the ADC_CCTRL register.

In this mode, the sampling interval between ADCs is configured to at least 2.5 ADCCLK cycles by hardware. Thus, the ASISEL bit becomes invalid when it cannot meet such sampling interval. Because of this feature, it is possible to put the same channel in the same ADC location without causing the overlapped sampling time.

Figure 16. Regular shift mode

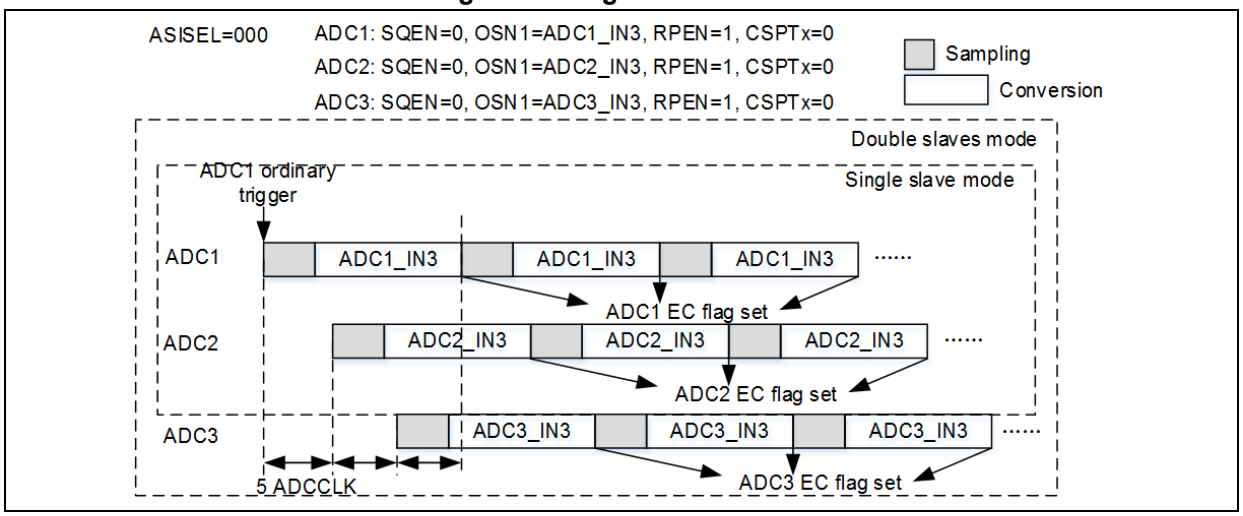
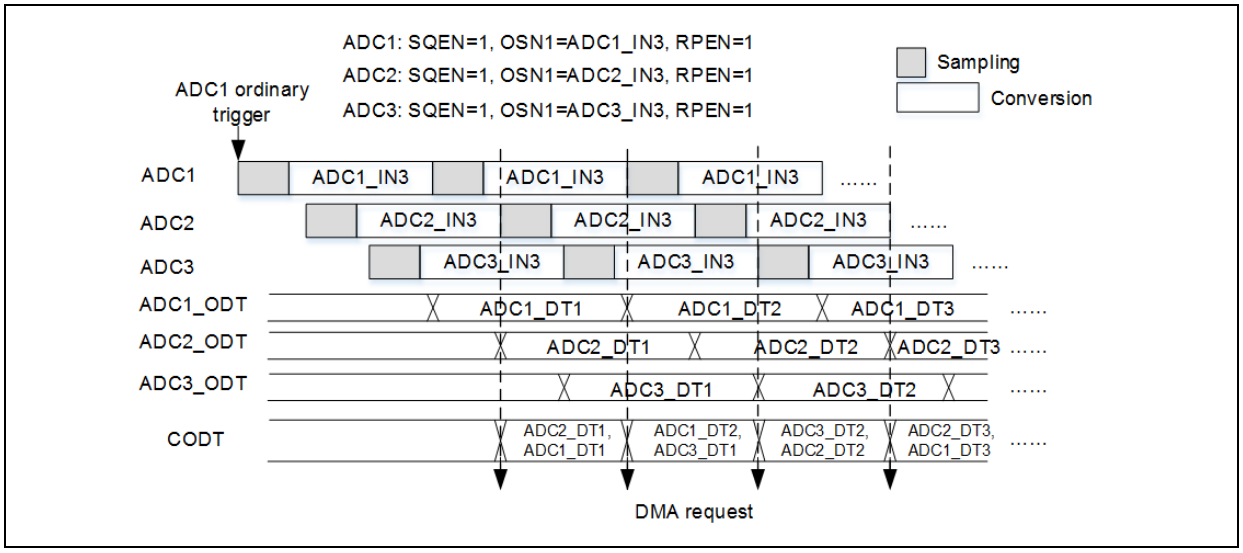


Figure 17. Regular shift mode and DMA mode 2



Note:

The preempted trigger or slave regular trigger is not allowed in this mode.

2.13.2 Software design

The master/slave mode configuration is implemented by the ADC common structure, and the code is as follows:

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

Note:

Both the master and slave must have the same resolution in order to avoid losing synchronization between master and slave.

In combined mode, select a DMA mode appropriately.

To enable several ADC conversions with low resolution simultaneously, it is recommended to use the master/slave mode together with DMA1 or DMA2

3 ADC configuration

This section introduces how to configure ADC and get the relevant data.

3.1 ADC configuration procedure

Generally, the ADC configuration includes:

- **External trigger source**

  The external events include TMR and EXINT triggers. Refer to the configuration of TMR or EXINT.

  Note: It is the configuration of trigger source; the trigger source can be enabled after ADC configuration is completed.

- **DMA configuration enable**

  The converted data of ADC ordinary group of channels can be transferred by DMA. If DMA transfer is required, initialize the DMA in advance (refer to the configuration of DMA).

- **Enable ADC digital clock**

  Enable the ADC digital clock to configure relevant functions.

- **ADC common structure configuration**

  It includes configuration of the master/slave mode, ADC division, master/slave DMA mode, DMA request continuation enable, shift length, internal temperature sensor/Vintrv and Vbat.

  - **Master/slave mode**

    Select from 13 modes according to the number of slave, channel type and combination type.

  - **ADC division**

    Set the ADC analog module clock. It is clocked by a divided HICK, with configurable division factor (2-17).

  - **Master/slave DMA mode**

    It is selected from five modes according to the master/slave mode. The DMA mode should be disabled in the combined preempted mode.

  - **DMA request continuation enable**

    It is used to set whether a DMA request is generated at the end of ordinary conversion after the completion of the programmed number of DMA transfers.

  - **Shift length in shift mode**

    It is used to set the adjacent ADC sampling interval in ordinary shift mode (configurable 5-20 ADCCLK cycles).

  - **Internal temperature sensor and Vintrv**

    Enable the internal temperature sensor and internal reference voltage, and connect to the ADC1 CH16 and CH17 respectively.

  - **Vbat**

    Enable the battery voltage; then it is divided by 4 before being connected to the ADC1 CH18.

  Note: When the ADC is used for sampling VBAT voltage, the VBAT keeps consuming as long as the VBATEN is set. Thus, it is recommended to enable VBAT only during the sampling

period, and disable it at the end of the sampling in order to lower power consumption.

- **ADC base structure configuration**

  It includes the configuration of sequence mode, repetition mode, data alignment and ordinary conversion sequence length.

  - **Sequence mode**

    It is enabled when multiple channels are configured, regardless of ordinary or preempted groups of channels.

  - **Repetition mode**

    The repetition mode is disabled for valid periodic triggers for conversion.

    If periodic triggers for conversion is not required, enable the repetition mode to allow the specified channels to be converted repeatedly through a single trigger.

  - **Data alignment**

    It is used to select the alignment of converted data (right-aligned or left-aligned).

  - **Ordinary conversion sequence length**

    It can be set to any length from 1~16 conversions. It indicates the number of channels in a single ordinary sequence, which should be consistent with that of the actual ordinary sequence.

- **Ordinary channel configuration**

  It includes the configuration of channels, trigger and data transfer.

  - **Configuration of channels**

    It includes the conversion sequence, value of channel and sampling period, in which the same value of channel can be set for different conversion sequences.

  - **Configuration of trigger**

    It includes the trigger source and trigger edge detection. The trigger edge detection is disabled when software trigger is used.

  - **Data transfer**

    It includes the configuration of CPU or DMA transfer mode. It is disabled in the master/slave mode.

- **Preempted channel configuration**

  It includes the configuration of number of channels, channels and triggers.

  - **Number of channels**

    It is configurable from 1~4. It indicates the number of channels in a single preempted sequence, which should be consistent with that of the actual preempted sequence.

  - **Configuration of channels**

    It includes the conversion sequence, value of channel and sampling period, in which the same value of channel can be set for different conversion sequences.

  - **Configuration of trigger**

    It includes the trigger source and trigger edge detection. The trigger edge detection is disabled when software trigger is used.

- **Special mode configuration (optional)**

- **Partition mode**

  It includes the number of ordinary channels of each trigger conversion, ordinary channel partition mode enable, and preempted channel partition mode enable.

- **Automatic preempted group conversion mode**

  It is used to enable the preempted group automatic conversion after the end of ordinary group conversion.

- **Oversampling**

  It includes the oversampling rate, oversampling shift, ordinary oversampling trigger mode enable, ordinary oversampling restart mode selection, and ordinary/preempted oversampling enable.

- **Interrupt configuration**

  Enable the corresponding interrupt, including one or more of the OCCO interrupt, OCCE interrupt, PCCE interrupt and VMOR interrupt.

- **ADC power-on**

  Enable and power on the ADC, and wait until the ADC RDY flag is set.

- **ADC calibration**

  To guarantee accurate converted data, the ADC requires calibration after power-on, which includes:

  A/D initialization calibration and wait for completion of initialization calibration, A/D calibration and wait for completion of calibration.

- **ADC resolution adjustment**

  Adjust the resolution after completion of ADC calibration, and wait until the ADC RDY flag is set.

  Perform the above procedures to complete ADC initialization; then, trigger conversions by software or by enabling trigger source by hardware.

3.2 Data read

The ADC supports multiple methods to read the converted data, including:

- **Preempted channel data read access by CPU**

  The preempted channel does not have DMA capacities; therefore, the preempted channel data is read by CPU only. It is recommended to read the data by the following means:

  1) Enable PCCE interrupt;
  2) Buffer the converted data into an array through the PCCE interrupt function;
  3) Perform subsequent algorithm processing through the converted data by other application logic.

- **Ordinary channel data read access by CPU**

  AT32F435 series supports ordinary channel data read access by CPU. It is recommended to read the data by the following means to guarantee the real-time data reading:

  1) Set the EOCSFEN bit in the ADC_CTRL2 register by software, allowing the OCCE flag to be set each time the data register is updated;

2) Enable OCCE interrupt;
3) Buffer the converted data into an array through the OCCE interrupt function;
4) Perform subsequent algorithm processing through the converted data by other application logic.

- **Ordinary channel data read access by DMA (single buffer mode)**

The ordinary channel data supports DMA capabilities. It is recommended to read the converted data by DMA directly as follows:

1) Initialize and enable DMA;
2) Enable ADC DMA mode;
3) Get the DMA buffer data through the DMA data transfer complete interrupt function;
4) Perform subsequent algorithm processing through the buffer data by other application logic.

- **Ordinary channel data read access by DMA (dual buffer mode)**

The AT32F435 series MCU EDMA support dual buffer mode.

1) Initialize EDMA dual buffer mode and enable the EDMA;
2) Enable the ADC DMA mode and enable the DMA request for ordinary continuous mode;
3) Get the EDMA buffer data through the EDMA transfer complete interrupt function;
4) Perform subsequent algorithm processing through the buffer data by other application logic.

4 Application: ADC oversampling

4.1 Function overview

The ADC supports oversampling to guarantee the accuracy of converted data.

This example introduces oversampling for both ordinary and preempted groups of channels, with 8x oversampling and 12-bit resolution.

4.2 Resources

1) Hardware

   One AT-START BOARD of the corresponding series

   PA4 and PA7——3.3 V

   PA5 and PB0——GND

   PA6 and PB1——about 1.5 V

2) Software

   project\at_start_f4xx\examples\adc\ordinary_preempt_oversampling

4.3 Software design

1) Configuration process
   - Configure the GPIO for ADC
   - Configure the DMA for ordinary channel data transfer
   - Complete ADC relevant configurations
   - Trigger conversion by software
   - Get the converted data

2) Code
   - GPIO configuration function code

```
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 configuration function code

```
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 configuration function code

```
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));
}
```

■ Interrupt service function code

```
/* Get the ordinary channel data transfer complete flag */
void DMA1_Channel1_IRQHandler(void)
{
  if(dma_flag_get(DMA1_FDT1_FLAG) != RESET)
  {
    dma_flag_clear(DMA1_FDT1_FLAG);
    dma1_trans_complete_flag = 1;
  }
```

```
}
/* Get the ADC overflow status and converted data of preempted channel */
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 function code

```
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 Test result

Print out the test result in the serial port assistant, and the converted data is shown in the figure below.

Serial port configuration

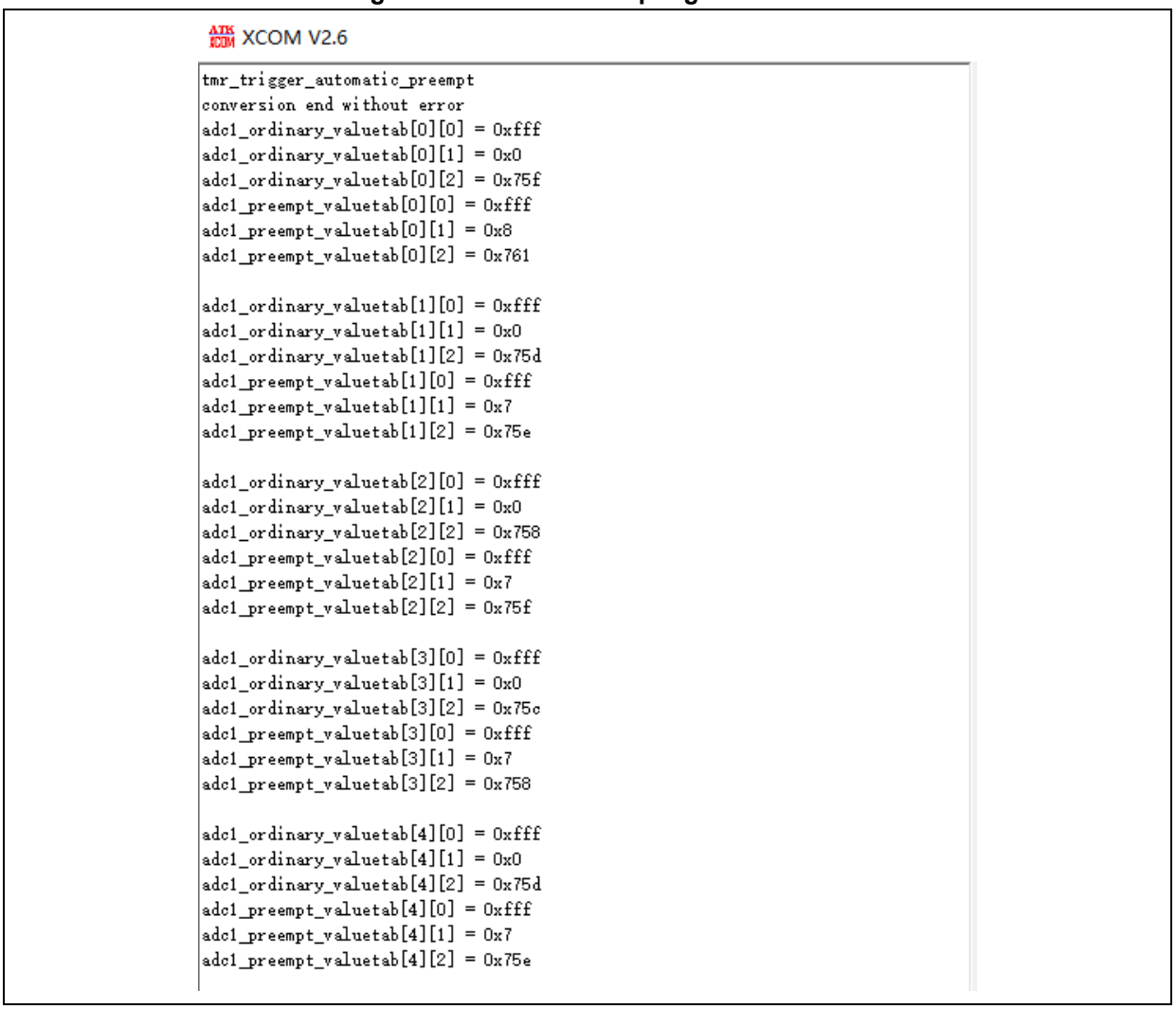
Baud rate: 115200

Stop bits: 1

Data bits: 8

Parity: None

Figure 18. ADC oversampling test result



5 Application: ADC voltage monitoring

5.1 Function overview

The ADC supports voltage monitoring.

This example introduce the voltage monitoring for the ordinary channel 5, and the programmed threshold is 0~Vref+/3.

5.2 Resources

1) Hardware

   One AT-START BOARD of the corresponding series

   PA4——3.3 V

   PA5——0 V

   PA6——about 1.5 V

2) Software

   project\at_start_f4xx\examples\ adc\voltage_monitoring

5.3 Software design

1) Configuration process

   - Configure the GPIO for ADC
   - Configure the DMA for ordinary channel data transfer
   - Complete configurations relevant to ADC and voltage monitoring
   - Trigger conversion by software
   - Get the converted data

2) Code

   - GPIO configuration function code

```
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 configuration function code

```
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 configuration function code

```
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));
}
```

■ Interrupt service function code

```
/* overflow and voltage out-of-range monitoring */
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 function code

```
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 Test result

Print out the test result in the serial port assistant (Channel5 is externally connected to 1.5 V randomly during test), as shown in the figure below.

It can be found that the Channel5 voltage out-of-range is monitored and the recorded actual monitored value is 1.5 V.

Serial port configuration

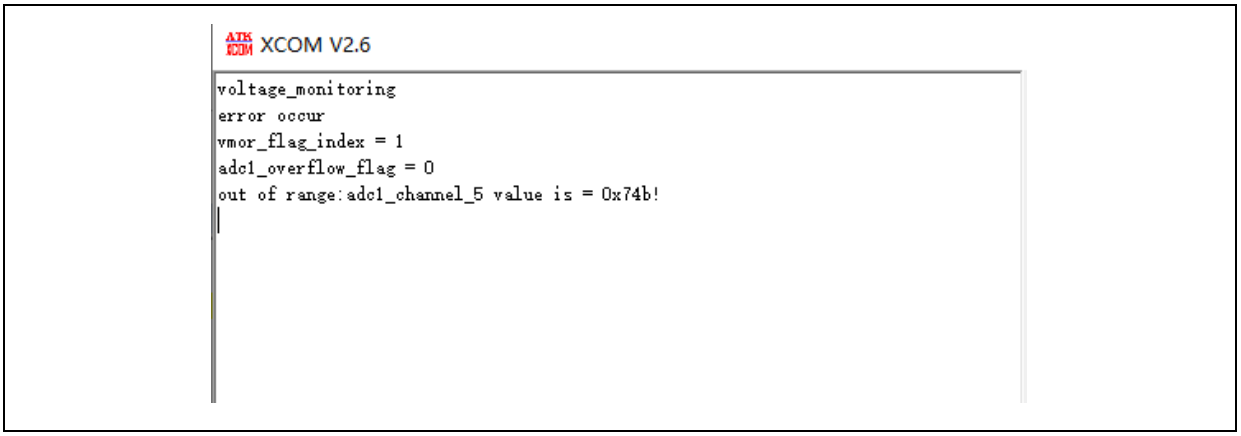
Baud rate: 115200

Stop bits: 1

Data bits: 8

Parity: None

Figure 19. ADC voltage monitoring test result



6 Application: ADC dual buffer mode

6.1 Function overview

The EDMA supports dual buffer mode to improve the data conversion efficiency.

This example introduces how to configure and use ADC fast conversion in the EDMA dual buffer mode.

6.2 Resources

1) Hardware

   One AT-START BOARD of the corresponding series

   PA4——3.3 V

   PA5——0 V

   PA6——about 1.5 V

2) Software

   project\at_start_f4xx\examples\ adc\edma_double_buffer

6.3 Software design

1) Configuration process
   - Configure the GPIO for ADC
   - Configure the DMA for ordinary channel data transfer
   - Complete ADC relevant configurations
   - Trigger conversion by software
   - Get the converted data

2) Code
   - GPIO configuration function code

```
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 configuration function code

```
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 interrupt */
  edma_interrupt_enable(EDMA_STREAM1, EDMA_FDT_INT, TRUE);
  edma_stream_enable(EDMA_STREAM1, TRUE);
}
```

■ ADC configuration function code

```
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));
}
```

■ Interrupt service function code

```
/* Get ordinary channel data transfer complete flag */
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;
  }
}
/* Get ADC overflow flag */
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 function code

```
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 Test result

Print out the test result in the serial port assistant. As shown below, the ADC conversion data is stored into two buffers of EDMA.

Serial port configuration

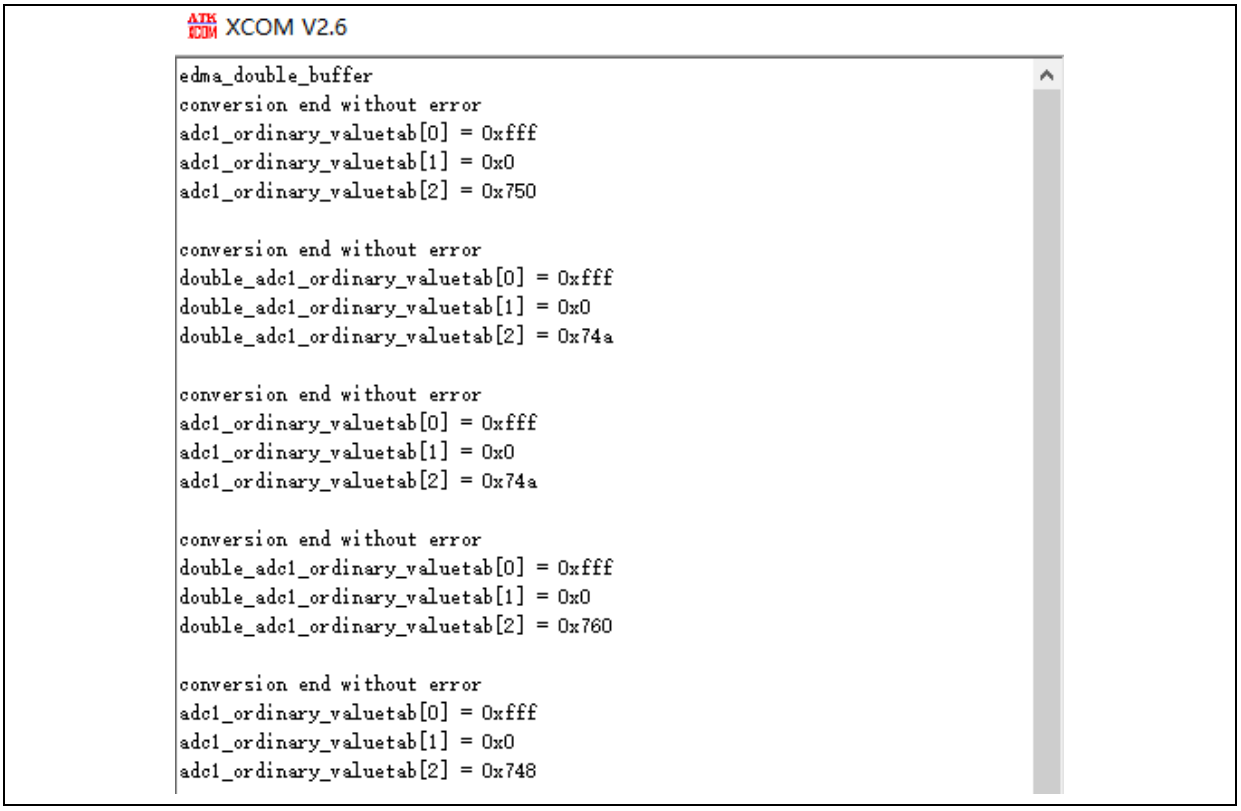
Baud rate: 115200

Stop bits: 1

Data bits: 8

Parity: None

Figure 20. Dual buffer mode test result



7 Application: ADC DMA mode 1

7.1 Function overview

The converted data of ordinary channels in ADC master/slave mode requires DMA transfer. The DMA supports multiple modes, each with different data transfer rules.

Using an inapplicable DMA mode for converted data transfer may cause data overflow due to delay in transmission. Therefore, the ADC master/slave mode and DMA mode should work together according to the Reference Manual.

DMA mode 1 is applicable to:

Ordinary simultaneous mode (single slave), ordinary shift mode (single slave), ordinary simultaneous mode (dual slave) and ordinary shift mode (dual slave)

This example selects “DMA mode 1 + ordinary simultaneous mode (dual slave)”.

7.2 Resources

1) Hardware

   One AT-START BOARD of the corresponding series

   PA4 and PA7 and PC0——3.3 V

   PA5 and PB0 and PC2——GND

   PA6 and PB1 and PC3——about 1.5 V

2) Software

   project\at_start_f4xx\examples\ adc\combine_mode_ordinary_smlt_twoslave_dma1

7.3 Software design

1) Configuration process
   - Configure the GPIO for ADC
   - Configure the TMR for trigger (TMR1_TRGOUT2 event)
   - Configure the DMA for ordinary channel data transfer
   - Complete ADC relevant configurations
   - Wait for the end of trigger and then disable the trigger TMR
   - Get the converted data

2) Code
   - GPIO configuration function code

```
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 configuration function code

```
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 configuration function code

```
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 configuration function code

```
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));
}
```

■ Interrupt service function code

```
/* Get ordinary channel data transfer complete flag */
void DMA1_Channel1_IRQHandler(void)
{
  if(dma_flag_get(DMA1_FDT1_FLAG) != RESET)
  {
    dma_flag_clear(DMA1_FDT1_FLAG);
    dma1_trans_complete_flag = 1;
  }
}
/* Get the overflow flag of each ADC */
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 function code

```
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 Test result

Print out the test result in the serial port assistant, and the converted data is shown in the figure below.

The transfer of converted data is on a half-word basis, and the transfer sequence is as follows:

ADC1 1st conversion in sequence

ADC2 1st conversion in sequence

ADC3 1st conversion in sequence

ADC1 2nd conversion in sequence

ADC2 2nd conversion in sequence

ADC3 2nd conversion in sequence

ADC1 3rd conversion in sequence

ADC2 3rd conversion in sequence

ADC33rd conversion in sequence

ADC1 1st conversion in sequence

……

Serial port configuration

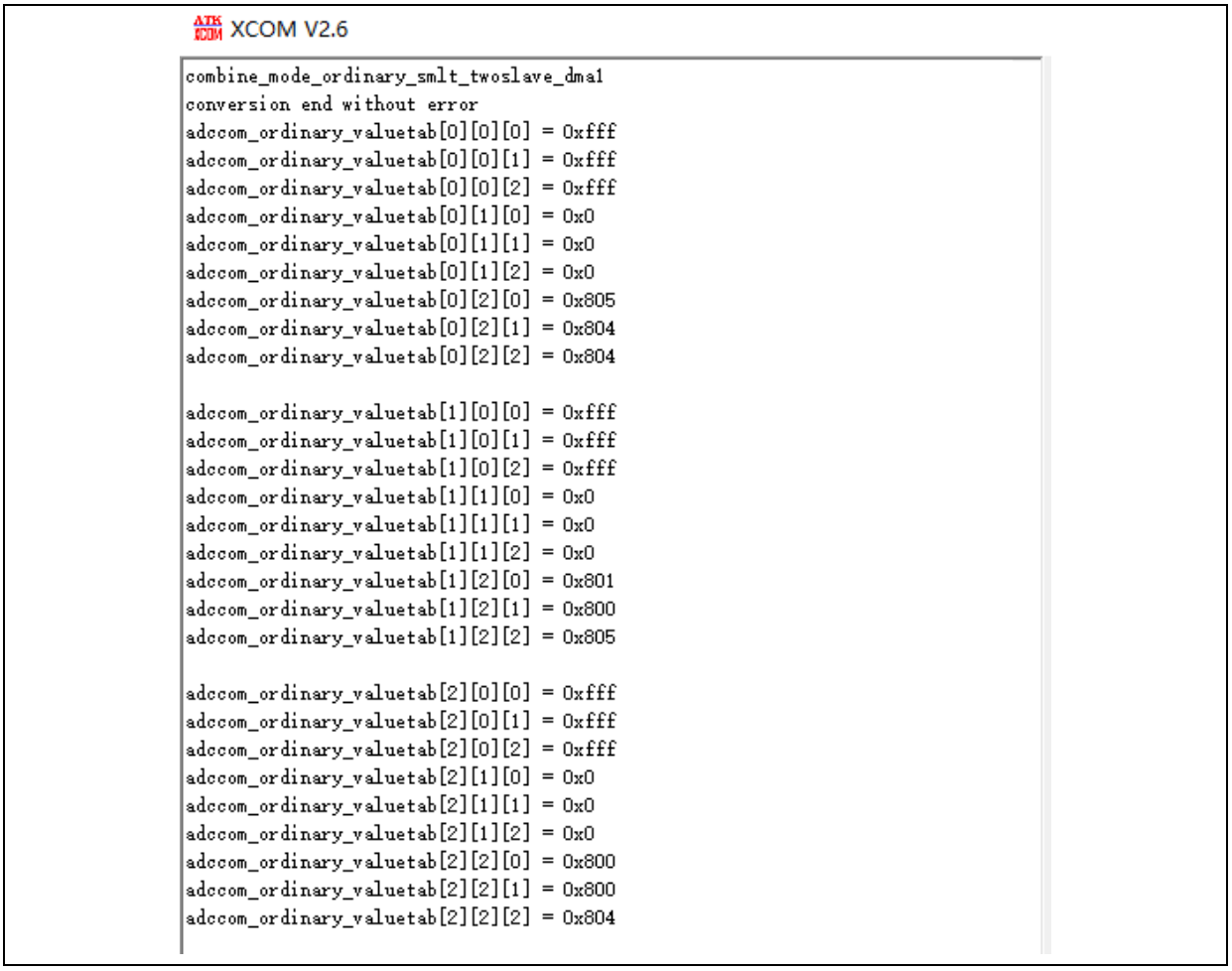
Baud rate: 115200

Stop bits: 1

Data bits: 8

Parity: None

Figure 21. ADC DMA mode 1 test result



8 Application: ADC DMA mode 2

8.1 Function overview

The converted data of ordinary channels in ADC master/slave mode requires DMA transfer. The DMA supports multiple modes, each with different data transfer rules.

Using an inapplicable DMA mode for converted data transfer may cause data overflow due to delay in transmission. Therefore, the ADC master/slave mode and DMA mode should work together according to the Reference Manual.

DMA mode 2 is applicable to:

Ordinary simultaneous mode (single slave), ordinary shift mode (single slave) and ordinary shift mode (dual slave)

This example selects “DMA mode 2 + ordinary simultaneous mode (single slave)”.

8.2 Resources

1) Hardware

   One AT-START BOARD of the corresponding series

   PA4 and PA7——3.3 V

   PA5 and PB0——GND

   PA6 and PB1——about 1.5 V

2) Software

   project\at_start_f4xx\examples\ adc\combine_mode_ordinary_smlt_oneslave_dma2

8.3 Software design

1) Configuration process
   - Configure the GPIO for ADC
   - Configure the DMA for ordinary channel data transfer
   - Complete the ADC relevant configurations
   - Trigger conversion by software
   - Get the converted data

2) Code
   - GPIO configuration function code

```
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 configuration function code

```
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 configuration function code

```
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));
}
```

■ Interrupt service function code

```
/* Get the ordinary channel data transfer complete flag */
void DMA1_Channel1_IRQHandler(void)
{
  if(dma_flag_get(DMA1_FDT1_FLAG) != RESET)
  {
    dma_flag_clear(DMA1_FDT1_FLAG);
    dma1_trans_complete_flag = 1;
  }
}
/* Get the overflow flag of each ADC */
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 function code

```
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 Test result

Print out the test result in the serial port assistant, and the converted data is shown in the figure below.

The transfer of converted data is on a word basis, and the transfer sequence is as follows:

(ADC2 1st conversion in sequence <<16) | ADC1 1st conversion in sequence

(ADC2 2nd conversion in sequence <<16) | ADC1 2nd conversion in sequence

(ADC2 3rd conversion in sequence <<16) | ADC1 3rd conversion in sequence

(ADC2 1st conversion in sequence <<16) | ADC1 1st conversion in sequence

……

Serial port configuration

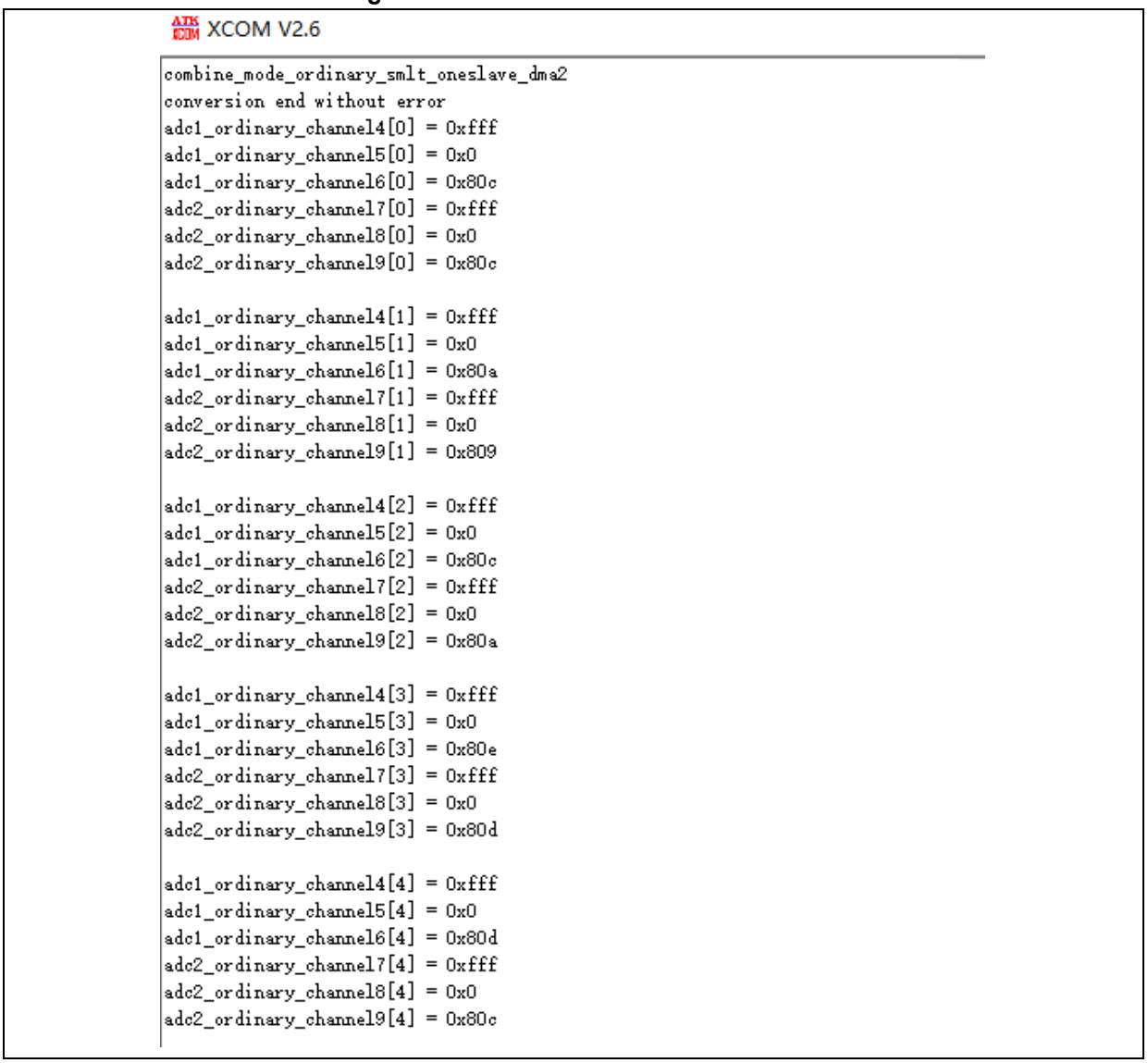
Baud rate: 115200

Stop bits: 1

Data bits: 8

Parity: None

Figure 22. ADC DMA mode 2 test result



9 Application: ADC DMA mode 3

9.1 Function overview

The converted data of ordinary channels in ADC master/slave mode requires DMA transfer. The DMA supports multiple modes, each with different data transfer rules.

Using an inapplicable DMA mode for converted data transfer may cause data overflow due to delay in transmission. Therefore, the ADC master/slave mode and DMA mode should work together according to the Reference Manual.

DMA mode 3 is applicable to:

6-bit/8-bit resolution ordinary simultaneous mode (single slave), 6-bit/8-bit resolution ordinary shift mode (single slave) and 6-bit/8-bit resolution ordinary shift mode (dual slave)

This example selects “DMA mode 3 + 8-bit resolution ordinary simultaneous mode (dual slave)”.

9.2 Resources

1) Hardware

   One AT-START BOARD of the corresponding series

   PA4——3.3 V

   PA7——GND

   PC0——about 1.5 V

2) Software

   project\at_start_f4xx\examples\ adc\combine_mode_ordinary_shift_twoslave_dma3

9.3 Software design

1) Configuration process
   - Configure the GPIO for ADC
   - Configure the TMR for trigger (TMR1_TRGOUT event)
   - Configure the DMA for ordinary channel data transfer
   - Complete ADC relevant configurations
   - Wait for the end of trigger and then disable the trigger TMR
   - Get the converted data

2) Code
   - GPIO configuration function code

```
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 configuration function code

```
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 configuration function code

```
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 configuration function code

```
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);
}
```

■ Interrupt service function code

```
/* Get the ordinary channel data transfer complete flag */
void DMA1_Channel1_IRQHandler(void)
{
  if(dma_flag_get(DMA1_FDT1_FLAG) != RESET)
  {
    dma_flag_clear(DMA1_FDT1_FLAG);
    dma1_trans_complete_flag = 1;
  }
}
/* Get the overflow flag of each ADC */
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 function code

```
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 Test result

Print out the test result in the serial port assistant, and the converted data is shown in the figure below.

The transfer of converted data is on a half-word basis, and the transfer sequence is as follows:

(ADC2 conversion <<8) | ADC1 conversion

(ADC1 conversion <<8) | ADC3 conversion

(ADC3 conversion <<8) | ADC2 conversion

(ADC2 conversion <<8) | ADC1 conversion

……

Serial port configuration

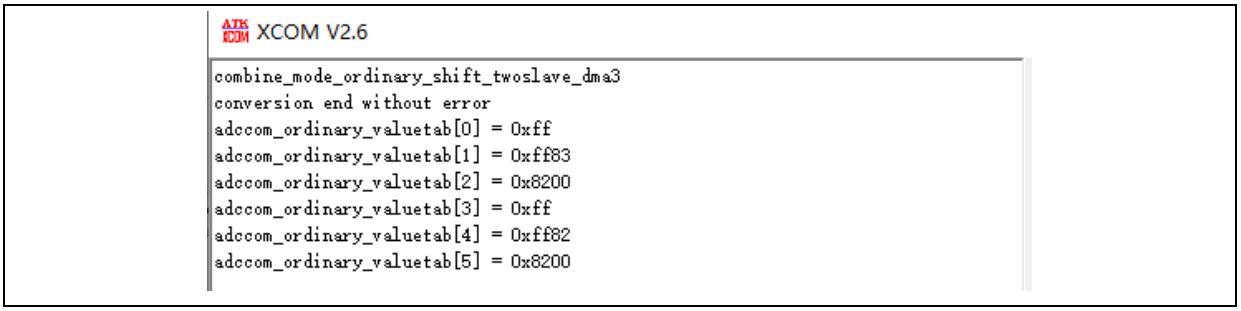
Baud rate: 115200

Stop bits: 1

Data bits: 8

Parity: None

Figure 23. ADC DMA mode 3 test result



10 Application: ADC DMA mode 4

10.1 Function overview

The converted data of ordinary channels in ADC master/slave mode requires DMA transfer. The DMA supports multiple modes, each with different data transfer rules.

Using an inapplicable DMA mode for converted data transfer may cause data overflow due to delay in transmission. Therefore, the ADC master/slave mode and DMA mode should work together according to the Reference Manual.

DMA mode 4 is applicable to:

6-bit/8-bit resolution ordinary simultaneous mode (dual slave), 6-bit/8-bit resolution ordinary shift mode (dual slave)

This example selects “DMA mode 4 + 8-bit resolution ordinary shift mode (dual slave)”.

10.2 Resources

1) Hardware

   One AT-START BOARD of the corresponding series

   PA4——3.3 V

   PA7——GND

   PC0——about 1.5 V

2) Software

   project\at_start_f4xx\examples\ adc\combine_mode_ordinary_shift_twoslave_dma4

10.3 Software design

1) Configuration process
   - Configure the GPIO for ADC
   - Configure the TMR for trigger (TMR1_TRGOUT event)
   - Configure the DMA for ordinary channel data transfer
   - Complete ADC relevant configurations
   - Wait for the end of trigger and then disable the trigger TMR
   - Get the converted data

2) Code
   - GPIO configuration function code

```
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 configuration function code

```
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 configuration function code

```
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 configuration function code

```
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);
}
```

■ Interrupt service function code

```
/* Get the ordinary channel data transfer complete flag */
void DMA1_Channel1_IRQHandler(void)
{
  if(dma_flag_get(DMA1_FDT1_FLAG) != RESET)
  {
    dma_flag_clear(DMA1_FDT1_FLAG);
    dma1_trans_complete_flag = 1;
  }
}
/* Get the overflow flag of each ADC */
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 function code

```
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 Test result

Print out the test result in the serial port assistant, and the converted data is shown in the figure below.

The transfer of converted data is on a word basis, and the transfer sequence is as follows:

(ADC3 conversion <<16) | (ADC2 conversion <<8) | ADC1 conversion

(ADC3 conversion <<16) | (ADC2 conversion <<8) | ADC1 conversion

……

Serial port configuration

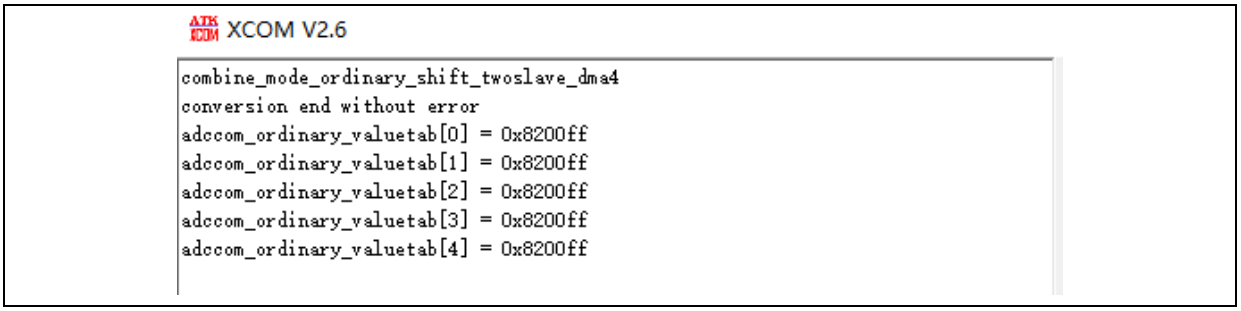
Baud rate: 115200

Stop bits: 1

Data bits: 8

Parity: None

Figure 24. ADC DMA mode 4 test result



11 Application: ADC DMA mode 5

11.1 Function overview

The converted data of ordinary channels in ADC master/slave mode requires DMA transfer. The DMA supports multiple modes, each with different data transfer rules.

Using an inapplicable DMA mode for converted data transfer may cause data overflow due to delay in transmission. Therefore, the ADC master/slave mode and DMA mode should work together according to the Reference Manual.

DMA mode 5 is applicable to:

Ordinary simultaneous mode (dual slave), ordinary shift mode (dual slave)

This example selects “DMA mode 5 + ordinary simultaneous mode (dual slave)”.

11.2 Resources

1) Hardware

   One AT-START BOARD of the corresponding series

   PA4 and PA7 and PC0——3.3 V

   PA5 and PB0 and PC2——GND

   PA6 and PB1 and PC3——about 1.5 V

2) Software

   project\at_start_f4xx\examples\ adc\ combine_mode_ordinary_smlt_twoslave_dma5

11.3 Software design

1) Configuration process

   - Configure the GPIO for ADC
   - Configure the TMR for trigger (TMR1_CH1 event)
   - Configure the DMA for ordinary channel data transfer
   - Complete ADC relevant configurations
   - Wait for the end of trigger and then disable the trigger TMR
   - Get the converted data

2) Code

   - GPIO configuration function code

```
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 configuration function code

```
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 configuration function code

```
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 configuration function code

```
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));}
```

■ Interrupt service function code

```
/* Get the ordinary channel data transfer complete flag */
void DMA1_Channel1_IRQHandler(void)
```

```
{
  if(dma_flag_get(DMA1_FDT1_FLAG) != RESET)
  {
    dma_flag_clear(DMA1_FDT1_FLAG);
    dma1_trans_complete_flag = 1;
  }
}
/* Get the overflow flag of each ADC */
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 function code

```
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 Test result

Print out the test result in the serial port assistant, and the converted data is shown in the figure below.

The transfer of converted data is on a word basis, and the transfer sequence is as follows:

(ADC2 1st conversion in sequence <<16) | ADC1 1st conversion in sequence

ADC3 1st conversion in sequence;

(ADC2 2nd conversion in sequence <<16) | ADC1 2nd conversion in sequence

ADC3 2nd conversion in sequence;

(ADC2 3rd conversion in sequence <<16) | ADC1 3rd conversion in sequence

ADC3 3rd conversion in sequence

(ADC2 1st conversion in sequence <<16) | ADC1 1st conversion in sequence

……

Serial port configuration

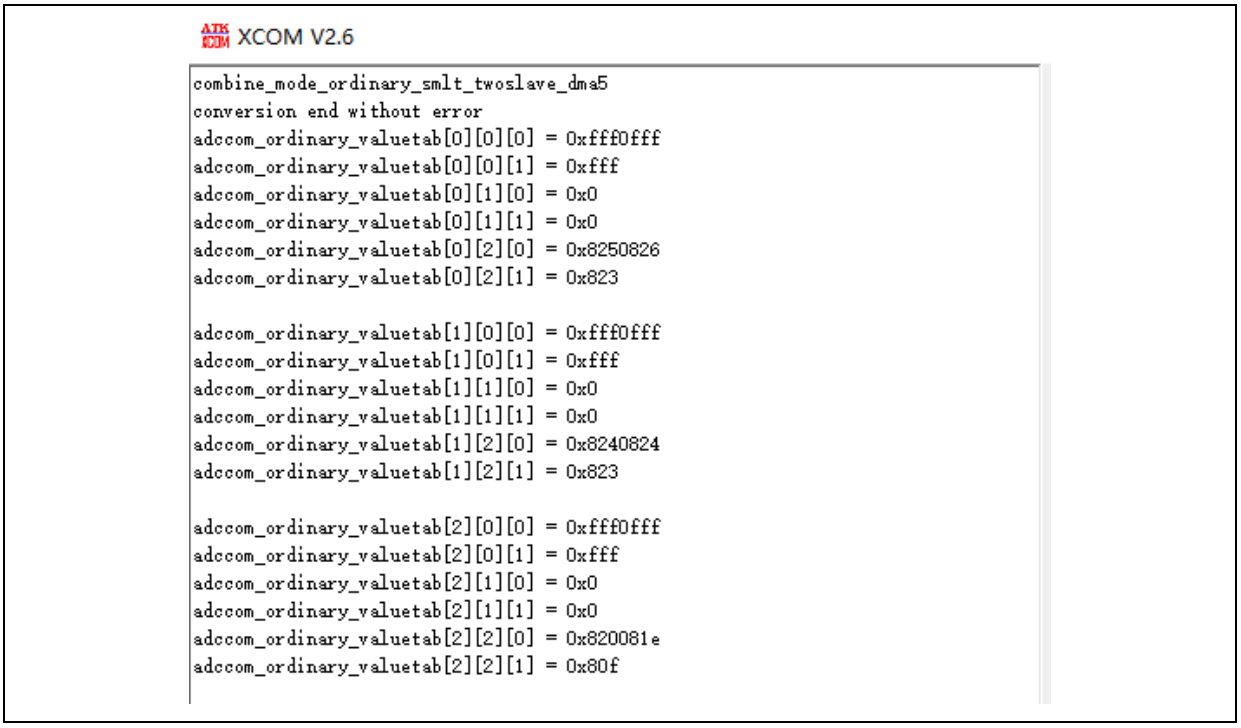
Baud rate: 115200

Stop bits: 1

Data bits: 8

Parity: None

Figure 25. ADC DMA mode 5 test result



12 Application: Vref voltage detection

12.1 Function overview

In actual applications, the ADC analog reference voltage may fluctuate due to factors such as external circuits, resulting in distortion of ADC conversion results. The following solution is recommended if the hardware cannot be optimized.

The ADC1_IN17 is connected to MCU internal reference voltage (1.2 V output from LDO); therefore, the current actual analog reference voltage value can be reversed according to the converted data of ADC1_IN17, and then calculate the ADC conversion channel data according to this analog reference voltage value.

This example introduces how to reverse the current actual analog reference voltage value according to the converted data of ADC1_IN17.

12.2 Resources

1) Hardware

   One AT-START BOARD of the corresponding series

2) Software

   project\at_start_f4xx\examples\ adc\current_vref_value_check

12.3 Software design

1) Configuration process
   - Configure the GPIO for ADC
   - Configure the DMA for ordinary channel data transfer
   - Complete ADC relevant configurations
   - Trigger conversion by software
   - Get the converted data to reverse the current Vref value
2) Code
   - DMA configuration function code

```
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 configuration function code

```
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));
}
```

■ Interrupt service function code

```
/* Get the ordinary channel data transfer complete flag */
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 function code

```
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 Test result

Print out the test result in the serial port assistant. No Vref is independently connected, and AT-Link is used for power supply. The test result shows that the actual Vref value is fixed at about 3.32 V.

Serial port configuration

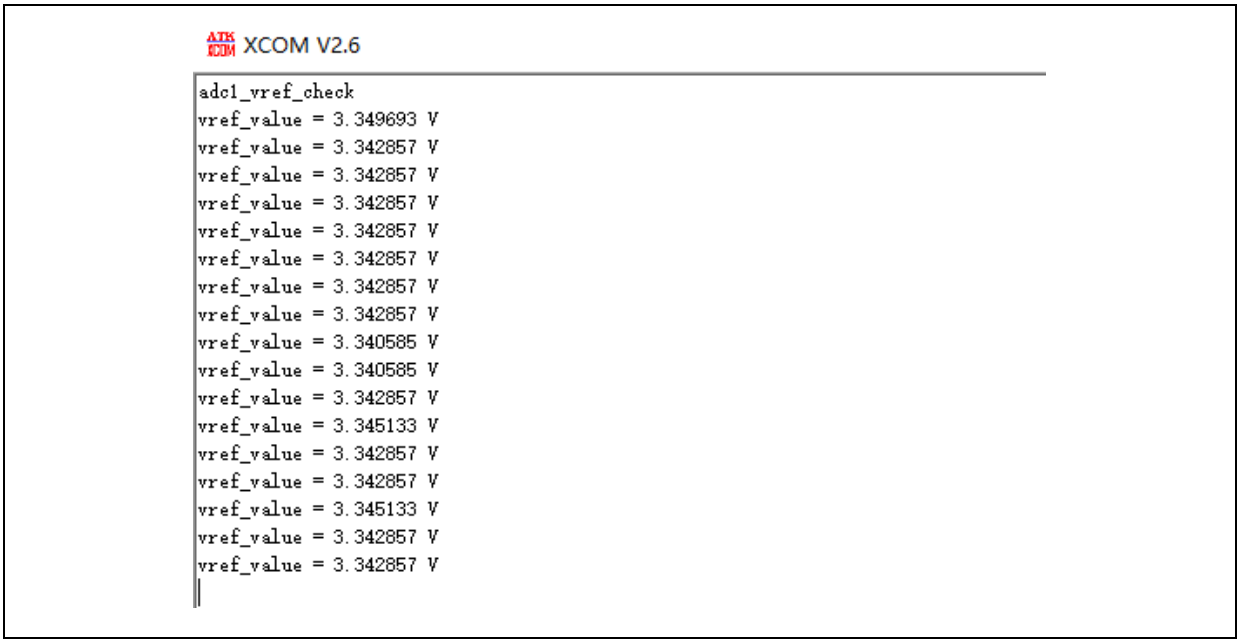
Baud rate: 115200

Stop bits: 1

Data bits: 8

Parity: None

Figure 26. ADC Vref voltage detection test result



13 Revision history

Table 5. Document revision history

Date	Version	Revision note
2021.9.29	2.0.0	Initial release.
2023.3.16	2.0.1	Optimized figures of each test result.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers are solely responsible for the selection and use of ARTERY's products and services; ARTERY assumes no liability for purchasers' selection or use of the products and the relevant services.

No license, express or implied, to any intellectual property right is granted by ARTERY herein regardless of the existence of any previous representation in any forms. If any part of this document involves third party's products or services, it does NOT imply that ARTERY authorizes the use of the third party's products or services, or permits any of the intellectual property, or guarantees any uses of the third party's products or services or intellectual property in any way.

Except as provided in ARTERY's terms and conditions of sale for such products, ARTERY disclaims any express or implied warranty, relating to use and/or sale of the products, including but not restricted to liability or warranties relating to merchantability, fitness for a particular purpose (based on the corresponding legal situation in any unjudicial districts), or infringement of any patent, copyright, or other intellectual property right.

ARTERY's products are not designed for the following purposes, and thus not intended for the following uses: (A) Applications that have specific requirements on safety, for example: life-support applications, active implant devices, or systems that have specific requirements on product function safety; (B) Aviation applications; (C) Aerospace applications or environment; (D) Weapons, and/or (E) Other applications that may cause injuries, deaths or property damages. Since ARTERY products are not intended for the above-mentioned purposes, if purchasers apply ARTERY products to these purposes, purchasers are solely responsible for any consequences or risks caused, even if any written notice is sent to ARTERY by purchasers; in addition, purchasers are solely responsible for the compliance with all statutory and regulatory requirements regarding these uses.

Any inconsistency of the sold ARTERY products with the statement and/or technical features specification described in this document will immediately cause the invalidity of any warranty granted by ARTERY products or services stated in this document by ARTERY, and ARTERY disclaims any responsibility in any form.