AN0011
应用笔记

AT32 3ADC simultaneous trigger

前言

本应用笔记介绍了在AT32微控制器上通过同一触发源触发3个ADC转换的方法，实现在任意时刻3个ADC通道的同步动作，以此来满足需要3个ADC同步转换需求的应用。

注：本应用笔记对应的代码是基于雅特力提供的V2.x.x 板级支持包（BSP）而开发，对于其他版本BSP，需要注意使用上的区别。

支持型号列表：

支持型号	AT32F403 系列 AT32F403A 系列 AT32F407 系列

目录

表目录

图目录

1 ADC 框架介绍

ADC 是一个将模拟输入信号转换为12 位数字信号的外设。采样率最高可达2MSPS。多达18 个通道源（16个外部和2个内部）可进行采样及转换。其具备如下功能：

1） 支持包括单次、反复、序列、抢占自动转换或分割等多种转换模式选择；

2） 普通通道和抢占通道均支持软件触发及外部触发，外部触发具备多种触发源选择；

2）转换数据可以设定左或右对齐方式进行存储，且抢占通道支持数据偏移量的设定；

3）电压监测特性允许应用程序监测输入电压是否超出用户定义的高/低阈值；

4）抢占通道组转换结束、通道转换结束、电压监测超出范围均具备相应中断的能力；

5）ADC时钟可调（时钟源来自PCLK2，ACCLK最大不得超过28MHz）；

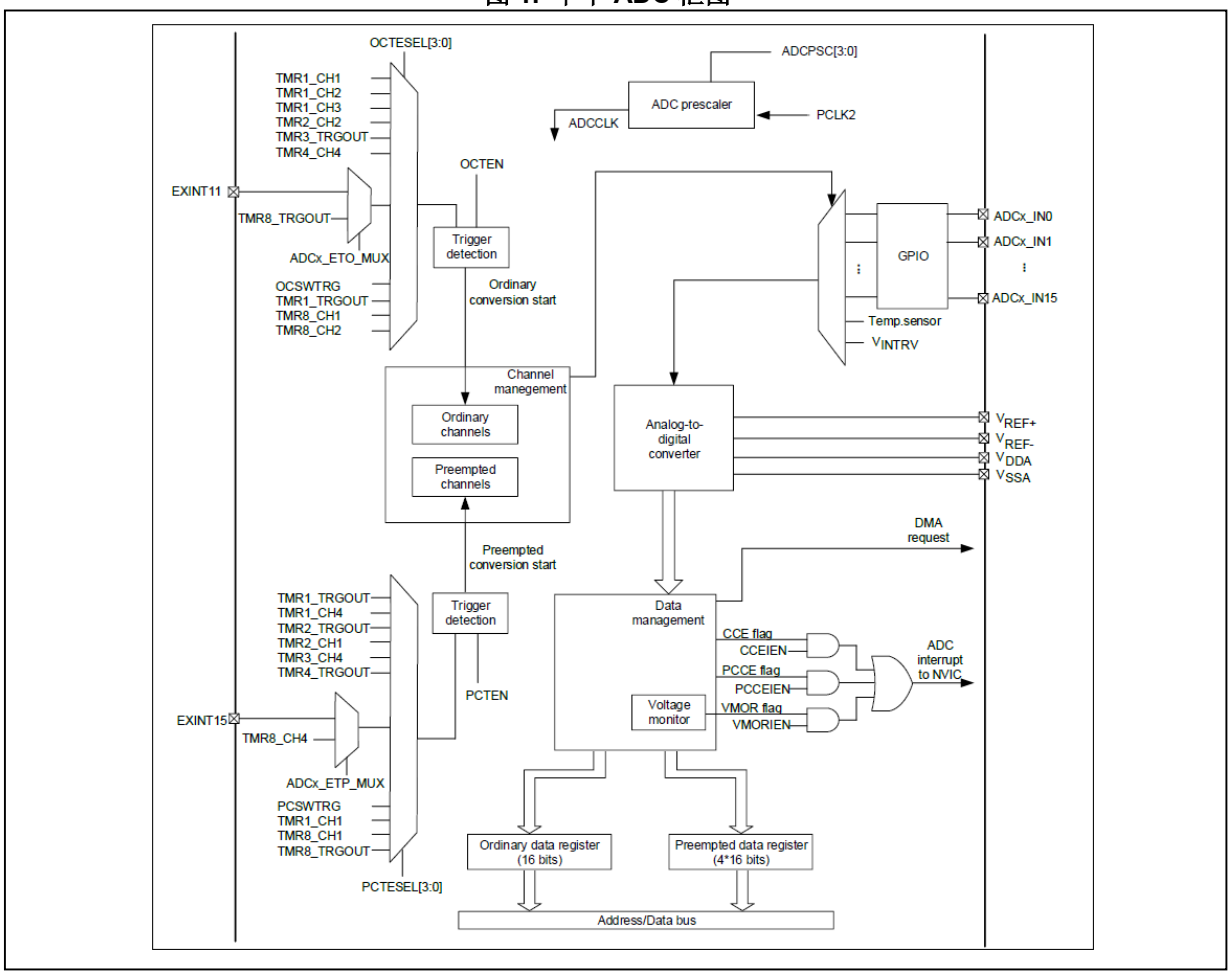
6）支持多达20个通道的采样转换（普通组16，抢占组4）、采样周期可根据需求调整；

7）普通通道转换数据可经DMA传输，当普通通道组设定多通道时，必须使用DMA获取转换数据；

8）支持联动ADC1和ADC2的多种主从组合模式。

单个ADC框图如下：

图 1. 单个 ADC 框图



2 3 个 ADC 同步触发原理说明

ADC主从模式的普通同时模式下，ADC2会完全同步ADC1动作。ADC1与ADC3具备多种相同的触发源。基于此点，应用可将ADC1和ADC2组合成主从模式（普通同时模式），ADC3与ADC1选用相同的触发源，从而就可以达到3个ADC完全同步动作的效果。

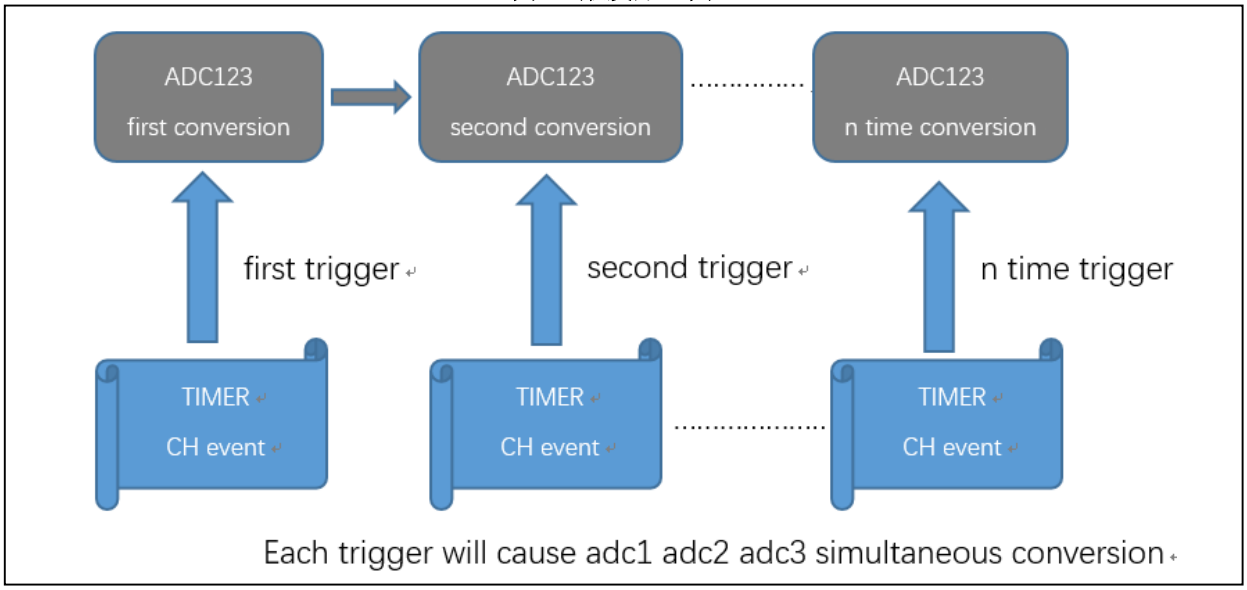
应用实例中，将ADC1、ADC2、ADC3分别各初始化两个通道，ADC1使用通道4、5，ADC2使用通道7、8，ADC3使用通道10、11。故将实现如下的同步触发转换效果：

表 1. 触发情景

触发情景						
触发	第一次		第二次		第三次	
ADC1	通道 4	通道 5	通道 4	通道 5	通道 4	通道 5
ADC2	通道 7	通道 8	通道 7	通道 8	通道 7	通道 8
ADC3	通道 10	通道 11	通道 10	通道 11	通道 10	通道 11

触发原理图如下所示：

图 2. 触发原理图



3 ADC 配置步骤

■ 触发源配置

例程配置代码如下

```
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);

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

  crm_periph_clock_enable(CRM_TMR1_PERIPH_CLOCK, TRUE);

  /* (systemclock/(systemclock/10000))/10000 = 1Hz(1s) */
  tmr_base_init(TMR1, 9999, (crm_clocks_freq_struct.sclk_freq/10000 - 1));
  tmr_cnt_dir_set(TMR1, TMR_COUNT_UP);
  tmr_clock_source_div_set(TMR1, TMR_CLOCK_DIV1);

  tmr_output_default_para_init(&tmr_oc_init_structure);
  tmr_oc_init_structure.oc_mode = TMR_OUTPUT_CONTROL_PWM_MODE_A;
  tmr_oc_init_structure.oc_polarity = TMR_OUTPUT_ACTIVE_LOW;
  tmr_oc_init_structure.oc_output_state = TRUE;
  tmr_oc_init_structure.oc_idle_state = FALSE;
  tmr_output_channel_config(TMR1, TMR_SELECT_CHANNEL_1, &tmr_oc_init_structure);
  tmr_channel_value_set(TMR1, TMR_SELECT_CHANNEL_1, 5000);
  tmr_channel_enable(TMR1, TMR_SELECT_CHANNEL_1, TRUE);
  tmr_output_enable(TMR1, TRUE);
}
```

例程使用timer的ch1事件触发ADC，在常规timer ch事件配置基础上需要注意如下内容：

应用设计需注意如下内容：

1) 为了实现演示效果，配置timer频率为1Hz，应用可根据需求修改此频率，只需保证触发间隔不能小于ADC序列转换时间；
2) 为了能实现ADC的触发，tmr_output_enable(TMR1, TRUE)命令不可省略；
3) 为避免误触发，timer使能需等待DMA及ADC全部配置完毕后才可进行；

4) 除TMR1_CH1外，ADC1与ADC3普通通道触发源还支持如下多种选择
   - TMR1_CH3 event
   - TMR1_TRGOUT event
   - TMR8_CH1 event
   - TMR8_TRGOUT event

■ DMA配置

例程配置代码如下

```
static void dma_config(void)
{
  dma_init_type dma_init_struct;
  crm_periph_clock_enable(CRM_DMA1_PERIPH_CLOCK, TRUE);
  crm_periph_clock_enable(CRM_DMA2_PERIPH_CLOCK, TRUE);
  nvic_irq_enable(DMA1_Channel1_IRQn, 0, 0);
  nvic_irq_enable(DMA2_Channel4_5_IRQn, 0, 0);
  dma_reset(DMA1_CHANNEL1);
  dma_reset(DMA2_CHANNEL5);
  dma_default_para_init(&dma_init_struct);
  dma_init_struct.buffer_size = 2;
  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_WORD;
  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_WORD;
  dma_init_struct.peripheral_inc_enable = FALSE;
  dma_init_struct.priority = DMA_PRIORITY_HIGH;
  dma_init_struct.loop_mode_enable = TRUE;
  dma_init(DMA1_CHANNEL1, &dma_init_struct);

  dma_init_struct.memory_base_addr = (uint32_t)adc3_ordinary_valuetab;
  dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_HALFWORD;
  dma_init_struct.peripheral_base_addr = (uint32_t)&(ADC3->odt);
  dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
  dma_init(DMA2_CHANNEL5, &dma_init_struct);

  dma_interrupt_enable(DMA1_CHANNEL1, DMA_FDT_INT, TRUE);
  dma_interrupt_enable(DMA2_CHANNEL5, DMA_FDT_INT, TRUE);
  dma_channel_enable(DMA1_CHANNEL1, TRUE);
  dma_channel_enable(DMA2_CHANNEL5, TRUE);
}
```

例程中ADC1&ADC2的转换数据通过DMA1_CHANNEL1传输，ADC3的转换数据通过DMA2_CHANNEL5传输。在常规DMA配置基础上需要注意如下内容：

1) 应用若需更换使用DMA_CHANNEL，需注意对通道进行弹性映射配置；
2) ADC1&ADC2转换数据会组合成32位进行传输，因此用于其数据传输的DMA1_CHANNEL1的外设及存储器数据宽度必须配置为32 bit 位宽；

3) ADC3的转换数据保持16位宽度，因此用于其数据传输的DMA2_CHANNEL5的外设及存储器数据宽度需配置为16 bit 位宽；
4) 数据传输是从ADC外设到存储器，因此DMA数据传输方向需设定为外设为源；
5) 为保障数据的稳定传输，DMA的通道传输数量通常匹配ADC的普通通道组个数设定；
6) DMA通道优先级需结合应用设定，当ADC转换足够快时，为避免数据丢失传输，需适当提高DMA通道优先级。

■ ADC配置

例程配置代码如下

```
static void adc_config(void)
{
  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);
  crm_adc_clock_div_set(CRM_ADC_DIV_6);

  /* select combine mode */
  adc_combine_mode_select(ADC_ORDINARY_SMLT_ONLY_MODE);
  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 = 2;
  adc_base_config(ADC1, &adc_base_struct);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_239_5);
  adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_239_5);
  adc_ordinary_conversion_trigger_set(ADC1, ADC12_ORDINARY_TRIG_TMR1CH1, TRUE);
  adc_dma_mode_enable(ADC1, TRUE);

  adc_base_config(ADC2, &adc_base_struct);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_239_5);
  adc_ordinary_channel_set(ADC2, ADC_CHANNEL_8, 2, ADC_SAMPLETIME_239_5);
  adc_ordinary_conversion_trigger_set(ADC2, ADC12_ORDINARY_TRIG_SOFTWARE, TRUE);

  adc_base_config(ADC3, &adc_base_struct);
  adc_ordinary_channel_set(ADC3, ADC_CHANNEL_10, 1, ADC_SAMPLETIME_239_5);
  adc_ordinary_channel_set(ADC3, ADC_CHANNEL_11, 2, ADC_SAMPLETIME_239_5);
  adc_ordinary_conversion_trigger_set(ADC3, ADC3_ORDINARY_TRIG_TMR1CH1, TRUE);
  adc_dma_mode_enable(ADC3, TRUE);

  adc_enable(ADC1, TRUE);
  adc_enable(ADC2, TRUE);
  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_enable(ADC3, TRUE);
  adc_calibration_init(ADC3);
  while(adc_calibration_init_status_get(ADC3));
  adc_calibration_start(ADC3);
  while(adc_calibration_status_get(ADC3));
}
```

例程中有使用ADC1、ADC2和ADC3来实现通道转换。在常规ADC配置基础上需要注意如下内容：

1) ADC2转换数据没有自己独立的DMA传输，并且为了实现同时转换，ADC1与ADC2必须组合成主从模式的普通同时模式。ADC3有自己独立的DMA请求，故ADC3配置为独立模式；
2) 为了实现同时转换，ADC1与ADC3普通通道需选择相同触发源，本例选用TMR1_CH1，其中还可选择如下某一项
   - TMR1_CH3 event
   - TMR1_TRGOUT event
   - TMR8_CH1 event
   - TMR8_TRGOUT event
3) 为避免作为从模式的ADC2丢失同步规则，ADC2的触发源必须选择为软件触发；
4) 为保障触发能被有效相应，触发间隔不能小于ADC序列转换时间；
5) 主从模式的ADC1和ADC2，校准必须在两个ADC使能后进行；
6) 需要确保同一通道不可同时被多个ADC采样转换；
7) 为保障数据的稳定传输，DMA的通道传输数量通常匹配ADC的普通通道组个数设定；
8) 本应用案例仅适用ADC普通通道，抢占通道转换数据不具备DMA传输能力。

■ 整体初始化顺序建议

建议初始化顺序如下

```
gpio_config();
tmr1_config();
dma_config();
adc_config();
tmr_counter_enable(TMR1, TRUE);
```

如此初始化顺序建议，主要是基于如下合理性考虑:

1) 在ADC初始化前配置timer，但是在ADC初始化使能后使能timer

   因为timer使能后就会开始运行，其满足条件时会立即产生触发事件，若此时ADC还没初始化完毕就会丢失触发事件，若此时ADC正在校准或上电等待，可能会一场相应触发事件；
2) 在ADC初始化使能前配置并使能DMA

   因为ADC使能后，只要存在触发条件，ADC就会开始相应转换，每个通道转换完毕会立即产生DMA传输请求，若DMA是能点滞后，会存在数据传输请求不能被及时相应，导致丢失数据最终产生数据错位的现象。

■ 中断服务函数设计

例程代码做如下设计

```
void DMA1_Channel1_IRQHandler(void)
{
  if(dma_flag_get(DMA1_FDT1_FLAG) != RESET)
  {
    dma_flag_clear(DMA1_FDT1_FLAG);
    dma1_trans_complete_flag = 1;
  }
}
void DMA2_Channel4_5_IRQHandler(void)
{
  if(dma_flag_get(DMA2_FDT5_FLAG) != RESET)
  {
    dma_flag_clear(DMA2_FDT5_FLAG);
    dma2_trans_complete_flag = 1;
  }
}
```

例程只使用了两个DMA的传输完成中断，中断服务函数内只记录是否发生了传输完成事件。

中断服务函数设计以精简为主要原则，由于中断函数的响应具备优先原则，为避免因过于繁琐的中断响应而滞后其他较重要的应用代码执行，所以原则上不建议在中断函数内堆太多应用逻辑。

■ main函数设计

例程代码做如下设计

```
int main(void)
{
  __IO uint32_t index = 0;
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
  system_clock_config();
  at32_board_init();
  at32_led_off(LED2);
  at32_led_off(LED3);
  at32_led_off(LED4);
  usart1_config(115200);
  gpio_config();
  tmr1_config();
  dma_config();
  adc_config();

  tmr_counter_enable(TMR1, TRUE);
  while(1)
  {
    if(dma1_trans_complete_flag != 0)
    {
      index++;
```

```
            dma1_trans_complete_flag = 0;
            printf("adc1_channel4_data[0] = 0x%x\r\n", adc1_ordinary_valuetab[0] & 0xFFFF);
            printf("adc1_channel5_data[1] = 0x%x\r\n", adc1_ordinary_valuetab[1] & 0xFFFF);
            printf("adc2_channel7_data[0] = 0x%x\r\n", (adc1_ordinary_valuetab[0] >> 16) & 0xFFFF);
            printf("adc2_channel8_data[1] = 0x%x\r\n", (adc1_ordinary_valuetab[1] >> 16) & 0xFFFF);
            printf("\r\n");
            at32_led_toggle(LED2);
        }
        if(dma2_trans_complete_flag != 0)
        {
            dma2_trans_complete_flag = 0;
            printf("adc3_channel10_data[0] = 0x%x\r\n", adc3_ordinary_valuetab[0]);
            printf("adc3_channel11_data[1] = 0x%x\r\n", adc3_ordinary_valuetab[1]);
            printf("\r\n");
            at32_led_toggle(LED3);
        }
    }
}
```

例程main函数内，除外设初始化外，只进行了转换数据的打印输出，通过查询DMA传输完成事件标志来实现。应用设计需要注意如下内容：

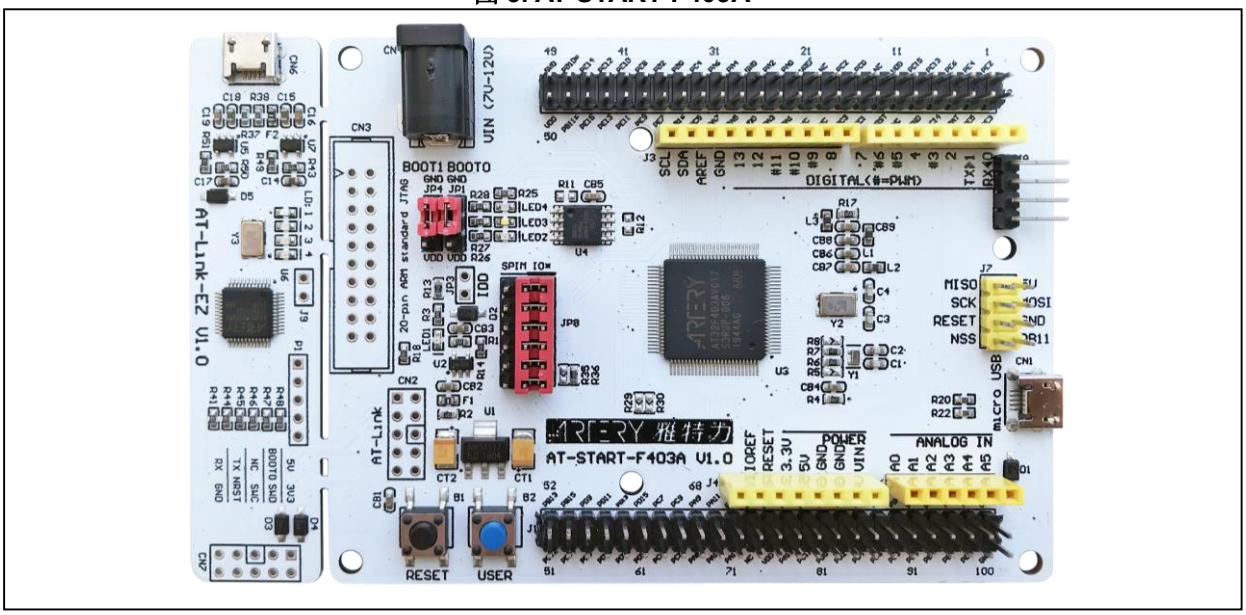
配置为主从普通同时模式的ADC1和ADC2的数据，被硬件组合封装成了32位的数据，所以需要对透过DMA获取到的32位数据进行解析后才可使用，其中，高16位为ADC2的转换数据，低16位为ADC1的转换数据。

4 Demo Code 快速使用方法

■ **硬件资源**

AT-START-F403A V1.0 实验板：

图 3. AT-START-F403A



ADC使用的通道与GPIO口对应如下表所示：

表 2. 通道对应

ADC1	通道 4→ PA4	通道 5→ PA5
ADC2	通道 7→ PA7	通道 8→ PB0
ADC3	通道 10→ PC0	通道 11→ PC1

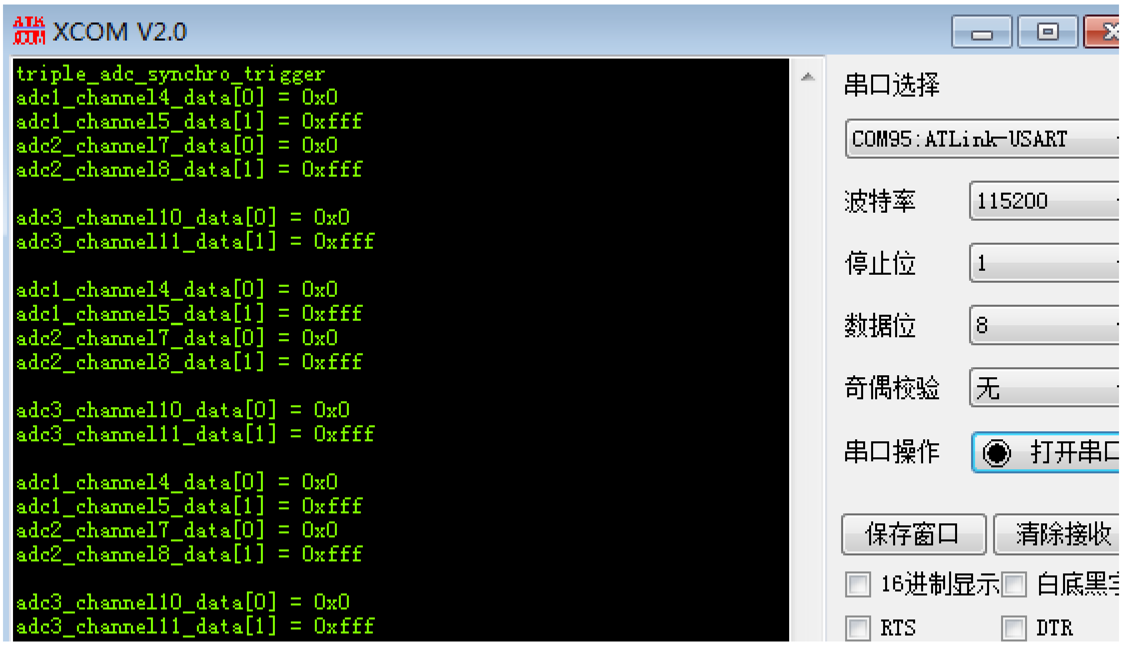
在做实验时，分别对这6个GPIO口灌入电压值即可

■ **测试方法**

1. 打开相应工程，编译并下载到目标板；
2. 对ADC123对应的管脚灌入电压值，通过串口打印或进入调试模式查看转换结果是否符合预期。

结果如下图：

图 4. 测试结果



可见相应通道电压值已经转换并通过DMA传输到指定的数组中。

注：所有project都是基于AT32F403A而建立，若用户需要在其他型号上使用，请参考AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates中各种型号示例工程进行简单修改即可。

注：所有project都是基于keil 5而建立，若用户需要在其他编译环境上使用，请参考AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates中各种编译环境（例如IAR6/7,keil 4/5）进行简单修改即可。

5 版本历史

表 3. 文档版本历史

日期	版本	变更
2021.12.14	2.0.0	最初版本

重要通知 - 请仔细阅读

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

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

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

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

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