ADC Flexible Channel Control

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code uses ADC with PDMA to trigger multiple ADC channels only once, and each channel can perform multiple A/D conversions.
BSP Version	M031_Series_BSP_CMSIS_V3.05.000
Hardware	NuMaker-M032KI V1.0

1. Overview

This example code uses ADC with PDMA to trigger multiple ADC channels only once, and each channel can perform multiple A/D conversions. All raw data after A/D conversion will be stored in pre-defined variables. When a round of ADC conversion is completed, the program can take out the raw data from the variables for post-analysis, such as calculating average values.

The ADC of the M031 series essentially supports multiple operating modes. Among them, Continuous Scan mode can trigger multiple ADC channels at one time, but each channel can only A/D convert once. What we need is to perform A/D conversion multiple times for each channel. For example, if you want to trigger the conversion of three ADC channels 1, 2, and 3 at one time, and perform two A/D conversions on each channel, then the Continuous Scan mode can only convert channels 1, 2, 3, 1, 2, 3 in sequence. This is different from the conversion sequence 1, 1, 2, 2, 3, 3 we need.

This example code uses ADC Continuous Scan mode with PDMA's autonomous data movement, calculation of movement times, and interrupt function to achieve the requirement of triggering multiple ADC channels at one time, and each channel can perform multiple A/D conversions.

In addition, to improve the application scope of the example code, the program structure is designed to easily specify the ADC channels and conversion sequence. Each ADC channel can have different conversion times and even different ADC advanced parameters. Details will be described in subsequent chapters.

1.1 Principle

To achieve the function of triggering multiple ADC channels at one time, and each channel can perform multiple A/D conversions, you can refer to the flow chart in Figure 1-1. The entire work is divided into multiple small tasks, as explained below:

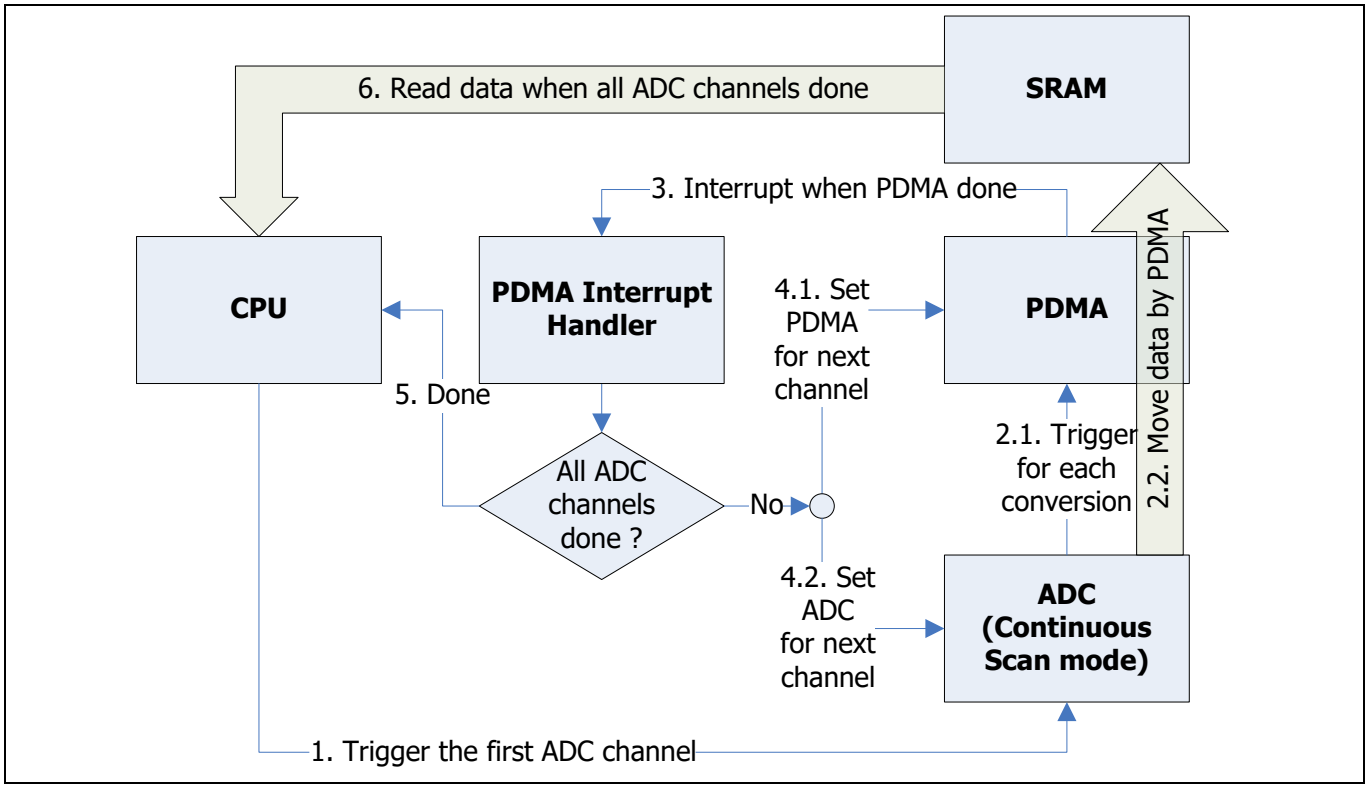

Figure 1-1 Example Code Flow Chart

- An ADC channel can be A/D converted multiple times, and the raw data is stored in the specified SRAM address.

  As shown in step 1 and step 2 of the flow chart, the main program flow only triggers the ADC once and uses the ADC Continuous Scan mode to convert a single ADC channel multiple times. Each time the ADC completes the A/D conversion, PDMA moves the conversion result to SRAM.

- Stop conversion when the A/D conversion reaches the specified number of times and triggers the PDMA interrupt.

  As shown in step 3 of the flow chart, when PDMA transfers data for the specified number of times, it stops transferring data and triggers a PDMA interrupt.

- Switch to the next ADC channel and restart A/D conversion.

  As shown in step 4 of the flow chart, the PDMA interrupt handler can modify both the PDMA and ADC settings and start converting the next ADC channel.

- When all ADC channels have completed A/D conversion, stop the conversion and complete one round of work.

  As shown in step 5 and step 6 of the flow chart, the program flow returns to the main program, and the CPU can read the raw data converted by ADC from SRAM for subsequent processing.

1.2 Demo Result

The setting and actual testing environment of this example code are as follows:

- ADC channel 2 converts 6 times, PB2 is grounded.
- ADC channel 1 converts 4 times, PB1 input voltage is about 1.82 volt.
- ADC channel 3 converts 8 times, PB3 input voltage is V_{DD}, about 3.08 volt.

The execution results are shown in Figure 1-2.

```
System clock rate: 48000000 Hz
Start ADC conversion ...
PDMA trasnfer done !
Conversion result of channel 2 ...
        #1:             0x002 (2)
        #2:             0x001 (1)
        #3:             0x001 (1)
        #4:             0x002 (2)
        #5:             0x001 (1)
        #6:             0x001 (1)
        Average:        0x001 (1)

Conversion result of channel 1 ...
        #1:             0x974 (2420)
        #2:             0x971 (2417)
        #3:             0x975 (2421)
        #4:             0x972 (2418)
        Average:        0x973 (2419)

Conversion result of channel 3 ...
        #1:             0xFFE (4094)
        #2:             0xFFF (4095)
        #3:             0xFFF (4095)
        #4:             0xFFF (4095)
        #5:             0xFFF (4095)
        #6:             0xFFF (4095)
        #7:             0xFFF (4095)
        #8:             0xFFF (4095)
        Average:        0xFFE (4094)

Exit ADC example code
```

Figure 1-2 Execution Results

2. Code Description

The main program of this example code will set the system frequency to HIRC, turn on the PDMA operating frequency, set the ADC operating frequency to HIRC/2, set the GPIO pins PB1, PB2, and PB3 to ADC channel 1, channel 2, and channel 3 modes, and set the GPIO Pins PB12 and PB13 as UART0 Rx and UART0 Tx modes. Then the main program calls the ADC_FunctionTest() function to implement the function of triggering multiple ADC channels at one time and each channel can perform multiple A/D conversions.

```
int32_t main(void)
{
    /* Init System, IP clock and multi-function I/O. */
    SYS_Init();

    /* Init UART0 for printf */
    UART0_Init();

    printf("System clock rate: %d Hz\n", SystemCoreClock);

    /* ADC function test */
    ADC_FunctionTest();

    /* Disable ADC IP clock */
    CLK_DisableModuleClock(ADC_MODULE);

    /* Disable PDMA IP clock */
    CLK_DisableModuleClock(PDMA_MODULE);

    printf("Exit ADC example code\n\n");

    while (1);
}
```

Before explaining the ADC operation code, the following will first introduce how to set the ADC conversion channel and conversion times.

This example code defines a struct data type ADC_CONF_T to store related ADC parameters. Each component of ADC_CONF_T can specify an ADC channel and the number of conversions of this ADC channel. In addition, ADC_CONF_T can also freely add other ADC advanced parameters to provide the possibility of expanding program functions. For example, this example code adds the extended sampling time of the ADC channel to allow the program to use different extended sampling times on different ADC channels.

Based on the data type ADC_CONF_F, this example code defines a global array variable g_ADC_Conf. The program will execute each ADC channel in the array sequentially until all ADC channels have been converted. Therefore, according to the setting value of g_ADC_Conf, the execution flow of this example code is: one trigger in main program flow can convert ADC channel 2 six times, convert ADC channel 1 four times, and convert ADC channel 3 eight times. If you modify the values in the array, you can easily specify the ADC channels, A/D conversion sequence, and the number of A/D conversions.

```
#define ADC_MAX_CH          (3) /* The total number of channels to convert */
#define ADC_MAX_CONV_NUM    (8) /* The max number in ADC_CONF_T.number */

typedef struct
{
    uint32_t channel;   /* The specific ADC channel to convert */
    uint32_t number;    /* The number of A/D conversion for this channel */
    uint32_t extend;    /* The ADC extend sample time for this channel */
} ADC_CONF_T;

volatile ADC_CONF_T g_ADC_Conf[ADC_MAX_CH] =
{
    {2, 6, 10}, /* Convert channel 2 6 times with extend sample time 10 ADC clocks */
    {1, 4,  0}, /* Convert channel 1 4 times with extend sample time 0 ADC clock */
    {3, 8, 20}  /* Convert channel 3 8 times with extend sample time 20 ADC clocks */
};
```

To store the raw data converted by ADC in SRAM, this example code also defines another global two-dimensional array variable gu32_ADC_RawData [ADC_MAX_CH] [ADC_MAX_CONV_NUM + 1]. The constant ADC_MAX_CH should be defined as the total number of ADC channels to be converted, which is 3 in this example code. The constant ADC_MAX_CONV_NUM should be defined as the maximum value of all conversion times, which is 8 in this example code.

It should be noted that the array size must be ADC_MAX_CONV_NUM plus 1. This is because the conversion value of the previous round of ADC channels will remain during the switching of ADC channels. The first raw data obtained by PDMA belongs to the old ADC channel and cannot be used. Thus, an extra location is defined to store this residual value. This residual value should be ignored when analyzing the data later.

In addition, a global variable gu32_ADC_Index must be defined as the index value of the g_ADC_Conf array and gu32_ADC_RawData array. It represents which setting value or raw data is currently being processed.

```
/* The raw data of ADC conversion result */
/*     Because the first conversion result may be the remaining data from
       the previous ADC channel, an extra location is declared to store it,
       but it is not actually used. */
volatile uint16_t gu32_ADC_RawData[ADC_MAX_CH][ADC_MAX_CONV_NUM + 1] = {0};

/* The index of g_ADC_Conf[] and gu32_ADC_RawData[] */
volatile uint32_t gu32_ADC_Index = 0;
```

Please refer to Figure 2-1 for the specific relationship between the variables in SRAM.

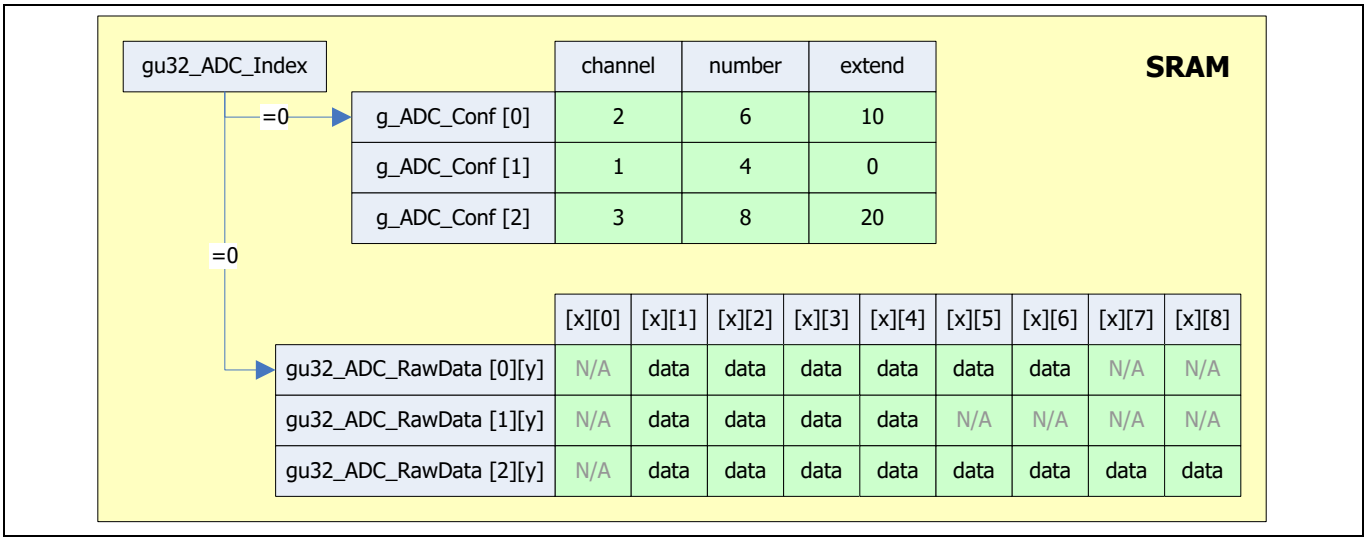

Figure 2-1 Data Structure in SRAM

Next, the ADC operation code will be explained below.

As shown in the code below, the array index value gu32_ADC_Index must first be set to 0 to let the ADC and PDMA know that the A/D conversion must be performed according to the first ADC setting value. Next, initialize PDMA which will be explained in detail in the next paragraph. Then specify the ADC to use Continuous Scan mode to convert the first channel, enable the linkage between ADC and PDMA, trigger ADC to start A/D conversion, and then wait for PDMA to complete the task. After PDMA completes the task, it obtains the ADC raw data from the gu32_ADC_RawData array variable and calculates their average value. This completes one round of work. A complete round of work only uses the ADC trigger function ADC_START_CONV() once.

```
/*---------------------------------------------------------------------*/
/* Main workflow for example code                                      */
/*---------------------------------------------------------------------*/
void ADC_FunctionTest(void)
{
    volatile uint32_t ADC_RawData_Sum[ADC_MAX_CH];
    volatile uint32_t index, number;

    /* Select the ADC configuration for the first element of g_ADC_Conf[] */
    gu32_ADC_Index = 0;

    /* Init PDMA to move data from ADC to SRAM */
    PDMA_Init();

    /* Enable ADC converter */
    ADC_POWER_ON(ADC);

    /* Wait for ADC internal power ready */
    CLK_SysTickDelay(10000);

    /* Set input mode as Single-end, and operation mode as Continuous Scan mode */
```

```
    ADC_Open(ADC, ADC_ADCR_DIFFEN_SINGLE_END, ADC_ADCR_ADMD_CONTINUOUS, (uint32_t)NULL);

    /* Select ADC input channel */
    ADC_SET_INPUT_CHANNEL(ADC, BIT0 << g_ADC_Conf[gu32_ADC_Index].channel);

    /* Set extend sampling time */
    ADC_SetExtendSampleTime(ADC, (uint32_t)NULL, g_ADC_Conf[gu32_ADC_Index].extend);

    /* ADC enable PDMA transfer */
    ADC_ENABLE_PDMA(ADC);

    gu32_PDMA_Status = PDMA_STATUS_IDLE_BUSY;

    /* Start ADC conversion */
    printf("Start ADC conversion ...\n");
    ADC_START_CONV(ADC);

    /* Wait for PDMA to complete its work */
    /* gu32_PDMA_Status will be set at PDMA_IRQHandler function */
    while (gu32_PDMA_Status == PDMA_STATUS_IDLE_BUSY);

    /* Check transfer result */
    if (gu32_PDMA_Status == PDMA_STATUS_DONE)
    {
        printf("PDMA trasnfer done !\n");
    }
    else if (gu32_PDMA_Status == PDMA_STATUS_ABORT)
    {
        printf("PDMA trasnfer abort !!\n");
    }

    /* Clear gu32_PDMA_Status software flag */
    gu32_PDMA_Status = PDMA_STATUS_IDLE_BUSY;

    /* Stop ADC conversion */
    ADC_STOP_CONV(ADC);

    /* Disable PDMA function of ADC */
    ADC_DISABLE_PDMA(ADC);
    NVIC_DisableIRQ(PDMA_IRQn);

    /* Calculate average and print ADC result except first sampling data result that
       belongs to previous channel */
    for (index = 0; index < ADC_MAX_CH; index++)
    {
        printf("Conversion result of channel %d ...\n", g_ADC_Conf[index].channel);
        ADC_RawData_Sum[index] = 0;

        for (number = 1; number < g_ADC_Conf[index].number + 1; number++)
        {
            ADC_RawData_Sum[index] += gu32_ADC_RawData[index][number];
            printf("\t#%d: \t\t0x%03X (%d)\n", number,
                   gu32_ADC_RawData[index][number], gu32_ADC_RawData[index][number]);
        }

        printf("\tAverage: \t0x%03X (%d)\n\n",
```

```
                ADC_RawData_Sum[index] / g_ADC_Conf[index].number,
                ADC_RawData_Sum[index] / g_ADC_Conf[index].number);
    }
}
```

The PDMA is initialized to use PDMA channel 1 to move data from ADC to the gu32_ADC_RawData array variable (g_ADC_Conf[gu32_ADC_Index].number + 1) times, allowing ADC to trigger PDMA, and then enable the PDMA interrupt function. The number moved by PDMA also needs to be increased by 1. The reason is the same as mentioned before, which is to process the residual value of the previous ADC channel.

```
/*---------------------------------------------------------------------*/
/* Configure PDMA for parameters that must be reloaded every round  */
/*---------------------------------------------------------------------*/
void ReloadPDMA(void)
{
    /* Set source address as ADC data register (no increment) and destination address as
       gu32_ADC_RawData[] array (increment) */
    PDMA_SetTransferAddr(PDMA, PDMA_CHANNEL, (uint32_t)&ADC->ADPDMA, PDMA_SAR_FIX,
    (uint32_t)gu32_ADC_RawData[gu32_ADC_Index], PDMA_DAR_INC);

    /* Transfer width is half word (16 bit) and transfer count is
       g_ADC_Conf[gu32_ADC_Index].number+1. */
    /* One more conversion is because the first conversion result belongs to the previous
       channel */
    PDMA_SetTransferCnt(PDMA, PDMA_CHANNEL, PDMA_WIDTH_16,
    g_ADC_Conf[gu32_ADC_Index].number + 1);

    /* Select PDMA request source as ADC RX */
    PDMA_SetTransferMode(PDMA, PDMA_CHANNEL, PDMA_ADC_RX, FALSE, 0);
}

/*---------------------------------------------------------------------*/
/* Configure PDMA peripheral mode from ADC to memory                */
/*---------------------------------------------------------------------*/
void PDMA_Init(void)
{
    /* Open PDMA Channel based on PDMA_CHANNEL setting*/
    PDMA_Open(PDMA, BIT0 << PDMA_CHANNEL);

    /* Configure PDMA for parameters that must be reloaded every round */
    ReloadPDMA();

    /* Set PDMA as single request type for ADC */
    PDMA_SetBurstType(PDMA, PDMA_CHANNEL, PDMA_REQ_SINGLE, 0);

    /* Enable PDMA interrupt */
    PDMA_EnableInt(PDMA, PDMA_CHANNEL, PDMA_INT_TRANS_DONE);
    NVIC_EnableIRQ(PDMA_IRQn);
}
```

When PDMA completes the work of an ADC channel, an interrupt must be triggered to execute the PDMA interrupt handler to switch the ADC channel. In addition to routine clearing the interrupt flag, the PDMA interrupt handler also needs to determine whether all ADC channels have been converted. If all ADC channels have been converted, the PDMA interrupt

handler can end the round of work and return the program control to the main program; if there are other ADC channels that need to be converted, the PDMA interrupt handler needs to reset the PDMA and ADC for the next ADC channel and trigger the ADC conversion directly without going back to the main program to do additional ADC triggering.

```
/*---------------------------------------------------------------------*/
/* PDMA interrupt handler to set ADC and PDMA for next round           */
/*---------------------------------------------------------------------*/
void PDMA_IRQHandler(void)
{
    uint32_t status;

    status = PDMA_GET_INT_STATUS(PDMA);

    if (status & PDMA_INTSTS_ABTIF_Msk)      /* PDMA abort */
    {
        if (PDMA_GET_ABORT_STS(PDMA) & (PDMA_ABTSTS_ABTIF0_Msk << PDMA_CHANNEL))
        {
            gu32_PDMA_Status = PDMA_STATUS_ABORT;
        }

        PDMA_CLR_ABORT_FLAG(PDMA, (PDMA_ABTSTS_ABTIF0_Msk << PDMA_CHANNEL));
    }
    else if (status & PDMA_INTSTS_TDIF_Msk) /* PDMA done */
    {
        if (PDMA_GET_TD_STS(PDMA) & (PDMA_TDSTS_TDIF0_Msk << PDMA_CHANNEL))
        {
            if (gu32_ADC_Index == (ADC_MAX_CH - 1))
            {
                /* All ADC channels done. Stop PDMA and trigger PDMA interrupt. */
                gu32_PDMA_Status = PDMA_STATUS_DONE;
            }
            else
            {
                ADC_STOP_CONV(ADC);
                /* Next ADC channel */
                gu32_ADC_Index++;
                /* Select ADC input channel */
                ADC_SET_INPUT_CHANNEL(ADC, BIT0 << g_ADC_Conf[gu32_ADC_Index].channel);
                /* Set extend sampling time */
                ADC_SetExtendSampleTime(ADC, (uint32_t)NULL,
                g_ADC_Conf[gu32_ADC_Index].extend);
                /* Set PDMA for new ADC channel */
                ReloadPDMA();
                ADC_START_CONV(ADC);
            }
        }

        PDMA_CLR_TD_FLAG(PDMA, (PDMA_TDSTS_TDIF0_Msk << PDMA_CHANNEL));
    }
    else
    {
        printf("Error: Unknown PDMA interrupt !!\n");
    }
}
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version:
  - M031_Series_BSP_CMSIS_V3.05.000.
- IDE version:
  - Keil uVision V4.74.

3.2 Hardware Requirements

- Circuit components:
  - NuMaker-M032KI V1.0.

4. Directory Information

The directory structure is shown below.

EC_M031_ADC_Flexible_Channel_Control_V1.00	
Library	Sample code header and source files.
CMSIS	Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
Device	CMSIS compliant device header file.
StdDriver	All peripheral driver header and source files.
SampleCode	
ExampleCode	
Project	Source file of example code.

Figure 4-1 Directory Structure

5. Example Code Execution

1. Browse the sample code folder as described in the Directory Information section and double-click *M031_ADC_Flexible_Channel_Control.uvproj.*
2. Enter Keil compile mode.
   - Build.
   - Download.
   - Start / Stop debug session.
3. Enter debug mode.
   - Run.

6. Revision History

Date	Revision	Description
2024.02.16	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.