ADC 灵活的通道控制

NuMicro® 32位系列微控制器范例代码介绍

文件信息

应用简述	本范例代码使用 ADC 搭配 PDMA 实现一次触发多个 ADC 通道，并且每个通道可以做多次 A/D 转换。
BSP 版本	M031_Series_BSP_CMSIS_V3.05.000
开发平台	NuMaker-M032KI V1.0

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. 概述

本范例代码使用 M031 系列的 ADC 搭配 PDMA 实现一次触发多个 ADC 通道，并且每个通道可以做多次 A/D 转换的功能。A/D 转换后的所有原始数据会存放在事先定义的变量中，当一轮 ADC 转换完成后，程序可以从变量中取出原始数据做事后分析，例如计算平均值等等。

M031 系列的 ADC 本来就支持多种工作模式。其中，Continuous Scan mode 可以一次触发多个 ADC 通道，但每个通道只能 A/D 转换一次，而我们需要的是每个通道要 A/D 转换多次。举例来说，如果要一次触发转换 1、2、3 三个 ADC 通道，每个通道做两次 A/D 转换，则 Continuous Scan mode 只能做到依序转换通道 1、2、3、1、2、3，这和我们需要的转换顺序 1、1、2、2、3、3 不同。

本范例代码利用 ADC Continuous Scan mode 搭配 PDMA 的自主搬动数据，计算搬动次数，和中断功能，来实现一次触发多个 ADC 通道，并且每个通道可以做多次 A/D 转换的需求。

此外，为了提高范例代码的应用范围，程序架构设计成可以简单的指定 ADC 通道及转换顺序，各 ADC 通道可以有不同的转换次数，甚至可以有不同的 ADC 进阶参数。细节会在后续章节中描述。

1.1 原理

要实现一次触发多个 ADC 通道，并且每个通道可以做多次 A/D 转换的功能，可以参考图 1-1 的流程图，并将整个流程切分成多个小任务，说明如下：

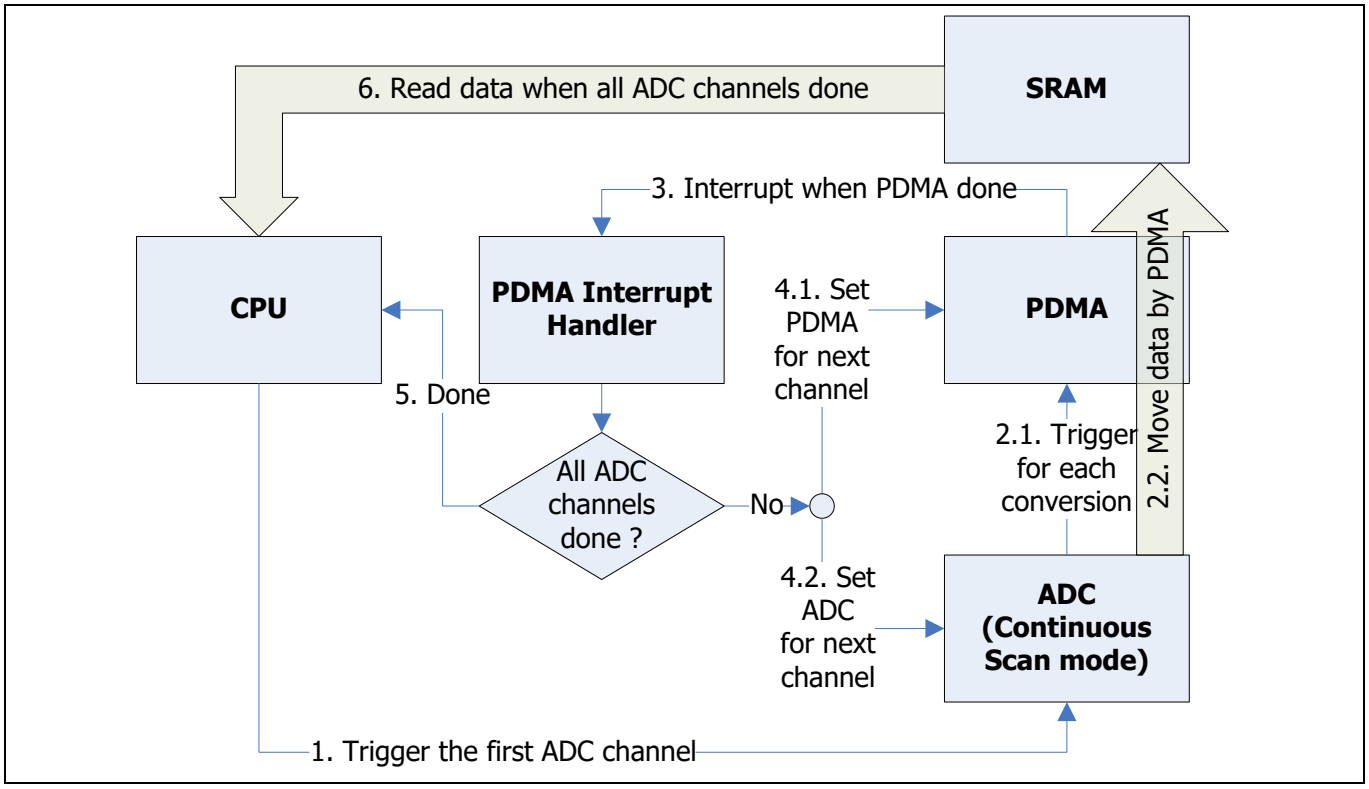

图 1-1 程序流程图

- 一个 ADC 通道可以 A/D 转换多次，并把原始数据存放在指定的 SRAM 地址。

  如流程图步骤 1 和 2，主程序流程只触发 ADC 一次，利用 ADC Continuous Scan mode 多次转换单一 ADC 通道，每完成 A/D 转换一次 PDMA 就把转换结果搬到 SRAM。

- 在 A/D 转换达到指定次数时停止，并触发 PDMA 中断。

  如流程图步骤 3，当 PDMA 搬运数据达到指定次数时，就停止搬运数据并触发 PDMA 中断。

- 切换到下一个 ADC 通道并重新开始 A/D 转换。

  如流程图步骤 4，PDMA 中断处理程序可以修改 PDMA 和 ADC 设定，并开始转换下一个 ADC 通道。

- 当所有 ADC 通道都 A/D 转换完成时，停止转换，完成一个流程。

  如流程图步骤 5 和 6，程序流程回到主程序，CPU 可以从 SRAM 读取 ADC 转换的原始数据做后续处理。

1.2 执行结果

本范例代码的设定条件及实测环境如下：

- ADC 通道 2 转换 6 次，PB2 接地。
- ADC 通道 1 转换 4 次，PB1 输入电压约 1.82 伏特。
- ADC 通道 3 转换 8 次，PB3 输入电压为 Vdd，约 3.08 伏特。

执行结果如图 1-2 所示。

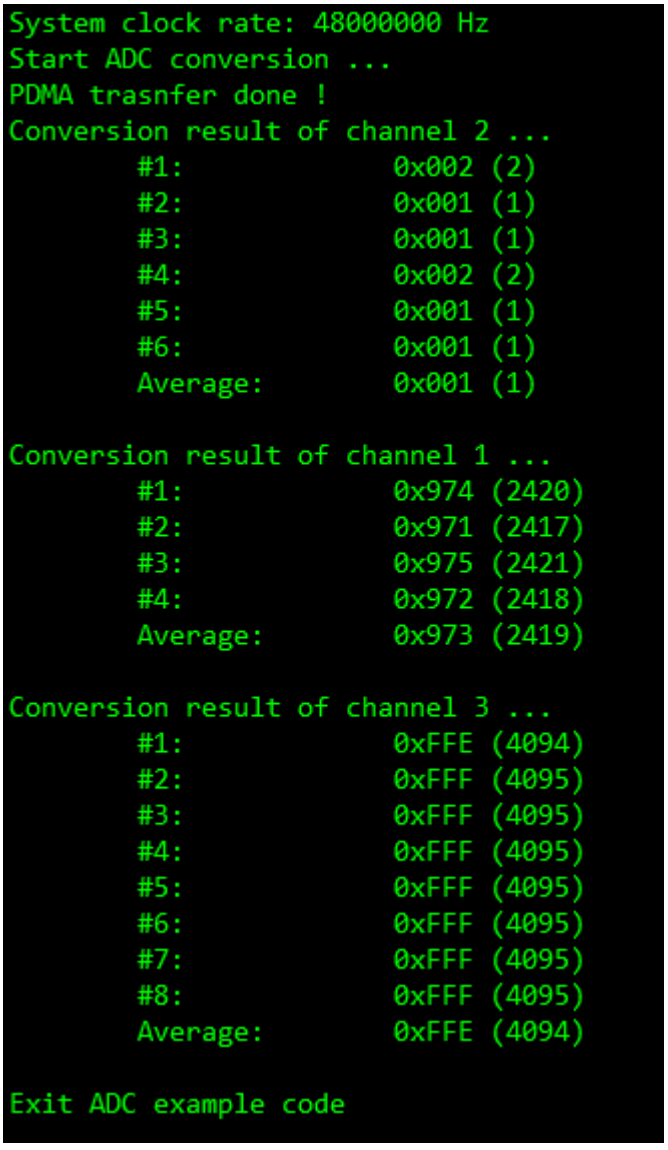

图 1-2 执行结果

2. 代码介绍

本范例代码主程序会设置系统时钟为 HIRC，开启 PDMA 运行时钟，设置 ADC 运行时钟为 HIRC/2，并将 GPIO 针脚 PB1、PB2、PB3 设置成 ADC 通道 1、通道 2、通道 3 模式，设置 GPIO 针脚 PB12、PB13 为 UART0 Rx、UART0 Tx 模式。然后执行 ADC_FunctionTest() 函数实现一次触发多个 ADC 通道，并且每个通道可以做多次 A/D 转换的功能。

```
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);
}
```

在说明 ADC 工作程序代码之前，先介绍一下如何设定 ADC 的转换通道和转换次数。

本范例代码定义一个 struct 数据类型 ADC_CONF_T 来存放相关的 ADC 参数。ADC_CONF_T 的每一个组件可以指定一个 ADC 通道以及这个 ADC 通道的转换次数。另外，ADC_CONF_T 也可以自由增加其他的 ADC 进阶参数，提供扩充程序功能的可能性。例如，本范例代码增加了 ADC 通道的延长采样时间，允许程序可以在不同的 ADC 通道上使用不同的延长采样时间。

基于数据类型 ADC_CONF_F，本范例代码定义一个全局数组变量 g_ADC_Conf。程序会依序执行数组中的每一个 ADC 通道，直到所有 ADC 通道都被转换过为止。所以，根据 g_ADC_Conf 的设定值，本范例代码的执行流程依序是：主程序流程的一次触发就可以转换 ADC 通道 2 六次，转换 ADC 通道 1 四次，转换 ADC 通道 3 八次。只要修改数组中的值，就可以轻松指定 ADC 通道，A/D 转换顺序，以及 A/D 转换次数。

```
#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 */
};
```

为了把 ADC 转换得到的原始数据存放在 SRAM 中，本范例代码也定义另一个全局二维数组变量 gu32_ADC_RawData[ADC_MAX_CH][ADC_MAX_CONV_NUM + 1]。其中，常数 ADC_MAX_CH 应该定义成要转换的 ADC 通道的总数，在本范例代码中就是 3。常数 ADC_MAX_CONV_NUM 应该定义成所有转换次数中的最大值，在本范例代码中就是 8。

需要特别说明的是，数组大小必须是 ADC_MAX_CONV_NUM 加 1，这是因为切换 ADC 通道的期间会残留上一轮 ADC 通道的转换值，所以 PDMA 拿到的第一个原始数据是属于旧的 ADC 通道，不可以拿来使用，所以我们多定义了一个位置来存放这个残留值，在事后分析数据时要忽略这个残留值。

另外也要定义一个全局变量 gu32_ADC_Index 当做 g_ADC_Conf 数组和 gu32_ADC_RawData 数组的索引值，代表目前正在处理哪一个设定值或哪一个原始数据。

```
/* 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;
```

SRAM 中各变量的具体关系可以参考下图。

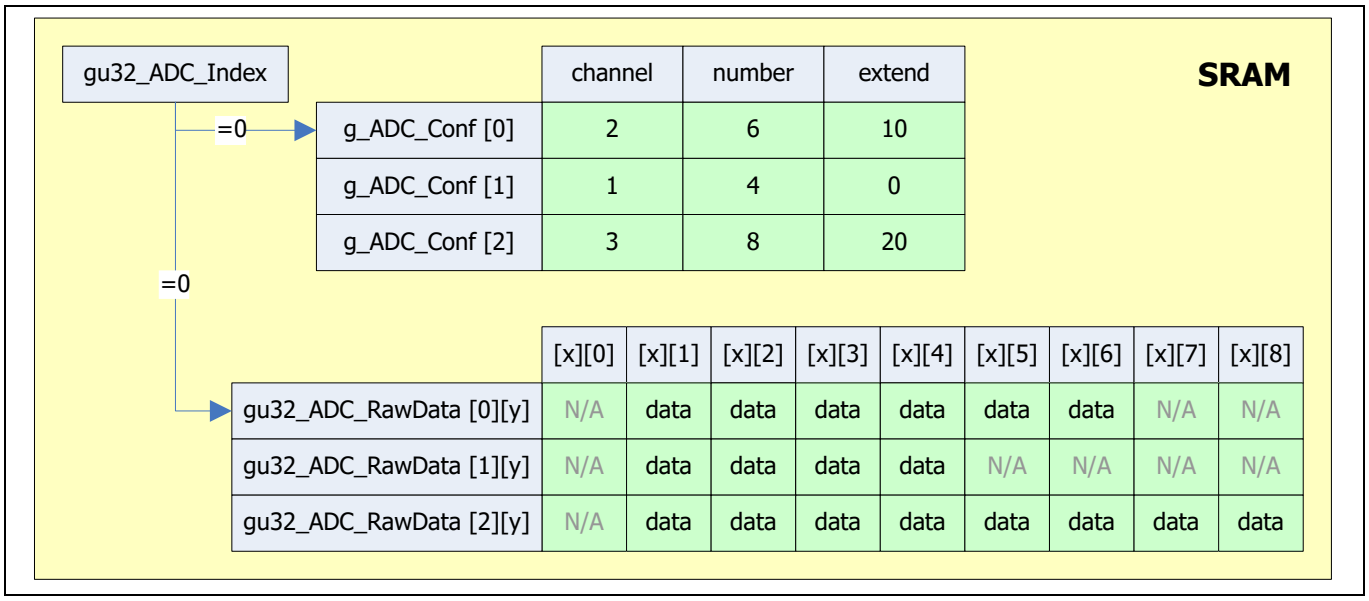

图 2-1 SRAM 中的数据结构

接下来，可以开始说明 ADC 工作程序代码了。

如下方的程序代码，首先要设定数组索引值 gu32_ADC_Index 为 0，让 ADC 和 PDMA 知道接下来要依照第一个 ADC 设定值来做 A/D 转换。再来初始化 PDMA，这会在下个段落仔细说明。再来指定 ADC 使用 Continuous Scan mode 来转换第一个通道。接下来开启 ADC 和 PDMA 之间的连动，触发 ADC 开始做 A/D 转换，并等待 PDMA 完成任务。PDMA 完成任务后，从 gu32_ADC_RawData 数组变量中取得 ADC 原始数据并计算平均值，就算完成一个流程。完整的一个流程只使用了一次 ADC 触发函数 ADC_START_CONV()。

```
/*---------------------------------------------------------------------*/
/* 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);
    }
}
```

PDMA 的初始化会设定使用 PDMA 通道 1，数据从 ADC 搬到 gu32_ADC_RawData 数组变量，搬动的次数 (g_ADC_Conf[gu32_ADC_Index].number + 1)，允许 ADC 触发 PDMA，并开启 PDMA 中断功能。PDMA 搬动次数也要多加 1，原因和前面说的一样，是为了处理上一个 ADC 通道的残留值。

```
/*---------------------------------------------------------------------*/
/* 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);
}
```

当 PDMA 完成一个 ADC 通道的采集后，就要触发中断执行 PDMA 中断处理程序来切换 ADC 通道。PDMA 中断处理程序除了例行的清除中断标志之外，也要判断是否所有的 ADC 通道都已经转换完毕。如果所有的 ADC 通道都已经转换完毕了，就可以结束一轮的采集，并把程控权交回给主程序；如果还有其他的 ADC 通道需要转换，就要重新设定 PDMA 和 ADC 去处理下一个 ADC 通道，并直接触发 ADC 转换，不需要回到主程序中做额外的 ADC 触发。

```
/*---------------------------------------------------------------------*/
/* 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. 软件与硬件需求

3.1 软件需求

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

3.2 硬件需求

- 电路组件:
  - NuMaker-M032KI V1.0.

4. 目录信息

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.

图 4-1 目录信息

5. 范例程序执行

1. 根据目录信息章节进入 ExampleCode 路径中的 KEIL 文件夹，双击 *M031_ADC_Flexible_Channel_Control.uvproj*。
2. 进入编译模式界面。

- 编译。
- 下载代码至内存。
- 进入 / 离开仿真模式。

3. 进入仿真模式界面。

- 执行代码。

6. 修订纪录

Date	Revision	Description
2024.02.16	1.00	初始发布。

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.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.