零死角玩转STM32

与野火同行 乐意惬无边

1 原创教程，完全开源。

2 通俗易懂，详尽解读。

3 由浅入深，结合实操。

4 配套板子，全面玩转。

5 强强联合，不断更新。

野火团队 Wild Fire Team

0、友情提示

《零死角玩转 STM32》系列教程由初级篇、中级篇、高级篇、系统篇、四个部分组成，根据野火 STM32 开发板旧版教程升级而来，且经过重新深入编写，重新排版，更适合初学者，步步为营，从入门到精通，从裸奔到系统，让您零死角玩转 STM32。M3 的世界，与野火同行，乐意惬无边。

另外，野火团队历时一年精心打造的《STM32 库开发实战指南》将于今年 10 月份由机械工业出版社出版，该书的排版更适于纸质书本阅读以及更有利于查阅资料。内容上会给你带来更多的惊喜。是一本学习 STM32 必备的工具书。敬请期待！

2、ADC（DMA 模式）

2.1 ADC 简介

ADC (Analog to Digital Converter)，模/数转换器。在模拟信号需要以数字形式处理、存储或传输时，模/数转换器几乎必不可少。

STM32 在片上集成的 ADC 外设非常强大。在 STM32F103xC、STM32F103xD 和 STM32F103xE 增强型产品，内嵌 3 个 12 位 的 ADC，每个 ADC 共用多达 21 个外部通道，可以实现单次或多次扫描转换。如野火 STM32 开发板用的是 STM32F103VET6，属于增强型的 CPU，它有 18 个通道，可测量 16 个外部和 2 个内部信号源。各通道的 A/D 转换可以单次、连续、扫描或间断模式执行。ADC 的结果可以左对齐或右对齐方式存储在 16 位数据寄存器中。模拟看门狗特性允许应用程序检测输入电压是否超出用户定义的高/低阀值。

2.2 STM32 的 ADC 主要技术指标

对于 ADC 来说，我们最关注的就是它的分辨率、转换速度、ADC 类型、参考电压范围。

- 分辨率

  12 位分辨率。不能直接测量负电压，所以没有符号位，即其最小量化单位

  $$LSB = V_{ref+} / 2^{12}$$ 。

- 转换时间

  转换时间是可编程的。采样一次至少要用 14 个 ADC 时钟周期，而 ADC 的时钟频率最高为 14MHz，也就是说，它的采样时间*最短为 1us*。足以胜任中、低频数字示波器的采样工作。

- ADC 类型

  ADC 的类型决定了它性能的极限，STM32 的是*逐次比较型 ADC*。

- 参考电压范围

STM32 的 ADC 参考电压输入见图 2-1。

表62　ADC引脚

名称	信号类型	注解
V_{REF+}	输入，模拟参考正极	ADC使用的高端/正极参考电压，2.4V≤ V_{REF+} ≤ V_{DDA}
V_{DDA} (1)	输入，模拟电源	等效于V_{DD}的模拟电源且：2.4V ≤ V_{DDA} ≤ V_{DD}(3.6V)
V_{REF-}	输入，模拟参考负极	ADC使用的低端/负极参考电压，V_{REF-} = V_{SSA}
V_{SSA} (1)	输入，模拟电源地	等效于V_{SS}的模拟电源地
ADCx_IN[15:0]	模拟输入信号	16个模拟输入通道

1．V_{DDA}和V_{SSA}应该分别连接到V_{DD}和V_{SS}。

图 2-1 参考电压

从图中可知，它的参考电压负极是要接地的，即 V_{ref-} = 0V。而参考电压正极的范围为 2.4V≦V_{ref+} ≦3.6V, 所以 STM32 的 ADC 是不能直接测量负电压的，而且其输入的电压信号的范围为：V_{REF-} ≦ V_{IN} ≦ V_{REF+}。*当需要测量负电压或测量的电压信号超出范围时，要先经过运算电路进行平移或利用电阻分压。*

2.3 ADC 工作过程分析

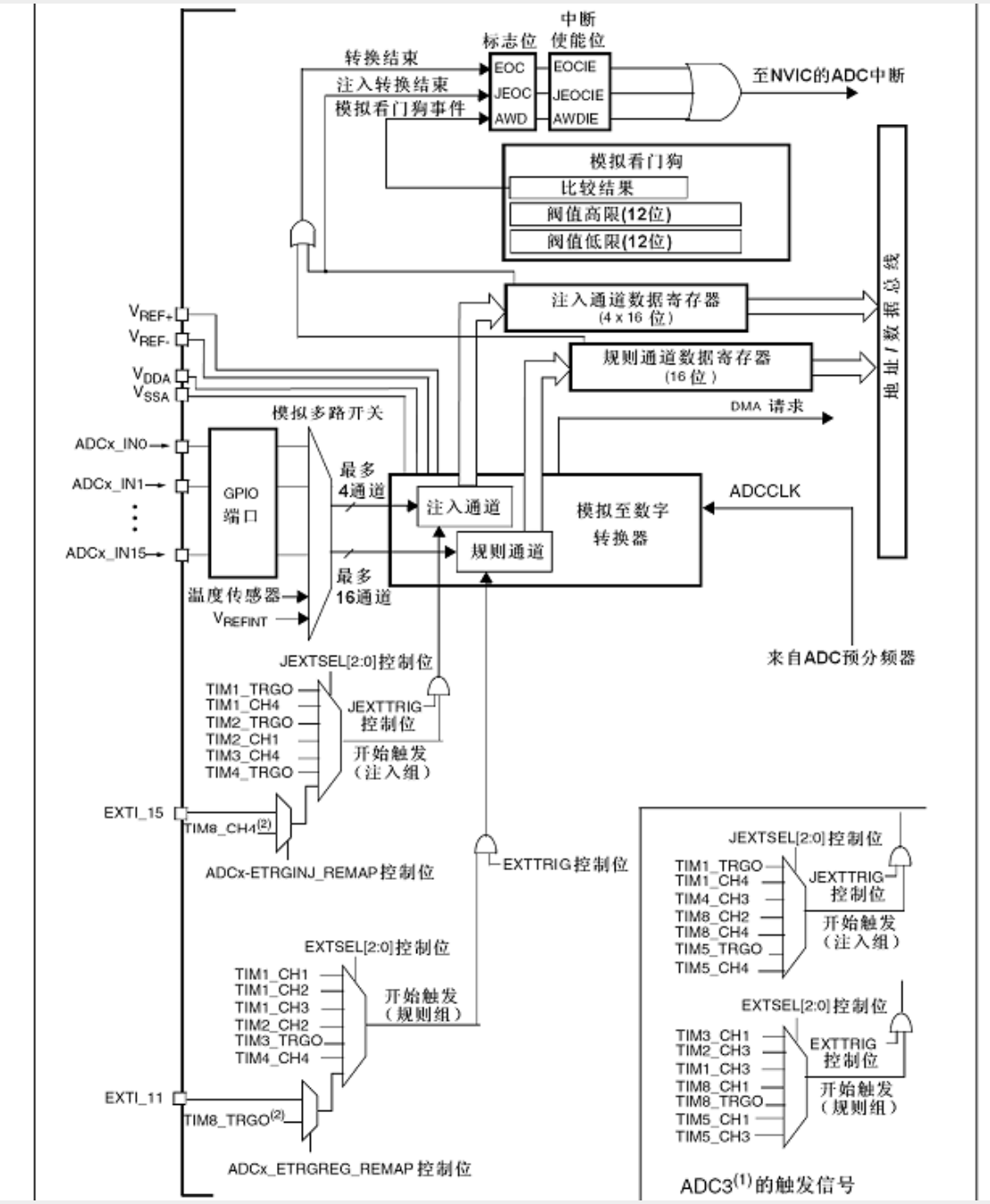

图 2-2 ADC 架构图

我们以 ADC 的*规则通道转换*来进行过程分析。所有的器件都是围绕中间的*模拟至数字转换器部分（下面简称 ADC 部件）*展开的。它的左端为 V_{REF+}、V_{REF-}等 *ADC* 参考电压，*ADCx_IN0~ADCx_IN15* 为 *ADC* 的输入信号通道，即某些 *GPIO* 引脚。输入信号经过这些通道被送到 *ADC* 部件，*ADC* 部件需要受到*触*

发信号才开始进行转换，如 EXTI 外部触发、定时器触发，也可以使用软件触发。ADC 部件接收到触发信号之后，在 ADCCLK 时钟的驱动下对输入通道的信号进行采样，并进行模数转换，其中 ADCCLK 是来自 ADC 预分频器的。

ADC 部件转换后的数值被保存到一个 16 位的规则通道数据寄存器(或注入通道数据寄存器)之中，我们可以通过 CPU 指令或 DMA 把它读取到内存(变量)。模数转换之后，可以触发 DMA 请求，或者触发 ADC 的转换结束事件。如果配置了模拟看门狗，并且采集得的电压大于阈值，会触发看门狗中断。

2.4 ADC 采集实例分析

使用 ADC 时常常需要不间断采集大量的数据，在一般的器件中会使用中断进行处理，但使用中断的效率还是不够高。在 STM32 中，使用 ADC 时往往采用 DMA 传输的方式，由 DMA 把 ADC 外设转换得的数据传输到 SRAM，再进行处理，甚至直接把 ADC 的数据转移到串口发送给上位机。本小节对 ADC 的 DMA 方式采集数据实例进行讲解，在讲解 ADC 的同时让读者进一步熟悉 DMA 的使用。

2.4.1 实验描述及工程文件清单

实验描述	串口 1(USART1)向电脑的超级终端以一定的时间间隔打印当前 ADC1 的转换电压值。
硬件连接	PC1 - ADC1 连接外部电压(通过一个滑动变阻器分压而来)。
用到的库文件	*startup/start_stm32f10x_hd.c* *CMSIS/core_cm3.c* *CMSIS/system_stm32f10x.c* *FWlib/stm32f10x_gpio.c* *FWlib/stm32f10x_rcc.c* *FWlib/stm32f10x_usart.c* *FWlib/stm32f10x_adc.c*

	FWlib/stm32f10x_dma.c
用户编写的文件	*USER/stm32f10x_it.c* *USER/usart1.c* *USER/adc.c*

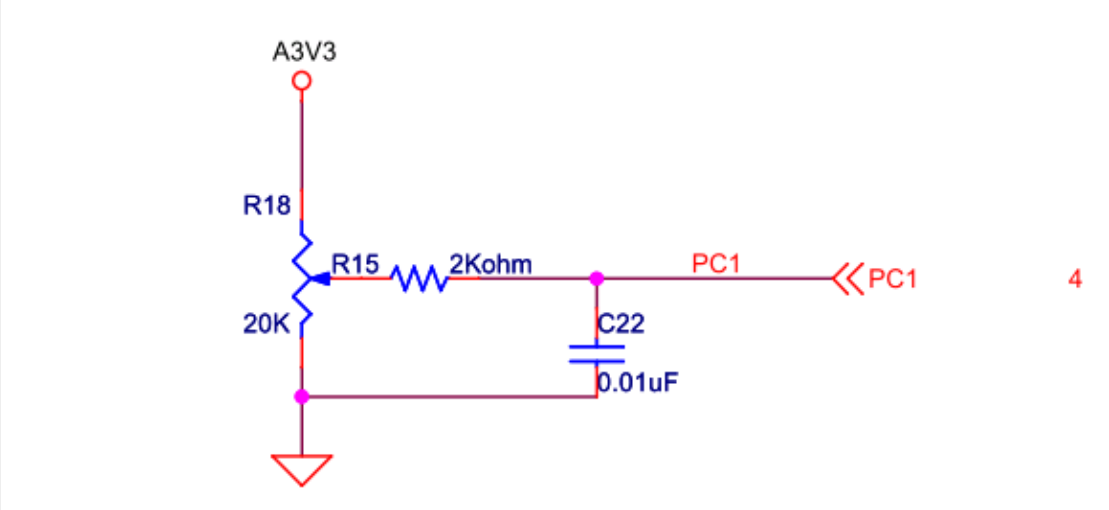

图 2-3 **野火 STM32 开发板 ADC 硬件原理图**

2.4.2 配置工程环境

本 ADC (DMA 方式) 实验中我们用到了 *GPIO*、*RCC*、*USART*、*DMA* 及 *ADC* 外设，所以我们先要把以下库文件添加到工程 *stm32f10x_gpio.c*、*stm32f10x_rcc.c*、*stm32f10x_usart.c*、*stm32f10x_dma.c*、*stm32f10x_adc.c*，添加旧工程中的外设用户文件 *usart1.c*，新建 *adc.c* 及 *adc.h* 文件，并在 *stm32f10x_conf.h* 中把使用到的 ST 库的头文件注释去掉。

```
1.  /**
2.    ******************************************************
3.    * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.    * @author  MCD Application Team
5.    * @version V3.5.0
6.    * @date    08-April-2011
7.    * @brief   Library configuration file.
8.    ******************************************************/
9.  #include "stm32f10x_adc.h"
10. #include "stm32f10x_dma.h"
11. #include "stm32f10x_gpio.h"
12. #include "stm32f10x_rcc.h"
13. #include "stm32f10x_usart.h"
```

2.4.3 main 文件

配置好工程环境之后，我们就从 *main* 文件开始分析：

```
1.  #include "stm32f10x.h"
2.  #include "usart1.h"
3.  #include "adc.h"
4.
5.  // ADC1 转换的电压值通过 MDA 方式传到 SRAM
6.  extern __IO uint16_t ADC_ConvertedValue;
7.
8.  // 局部变量，用于保存转换计算后的电压值
9.
10. float ADC_ConvertedValueLocal;
11.
12. // 软件延时
13. void Delay(__IO uint32_t nCount)
14. {
15.   for(; nCount != 0; nCount--);
16. }
17.
18. /**
19.   * @brief  Main program.
20.   * @param  None
21.   * @retval : None
22.   */
23.
24. int main(void)
25. {
26.     /* USART1 config */
27.     USART1_Config();
28.
29.     /* enable adc1 and config adc1 to dma mode */
30.     ADC1_Init();
31.
32.     printf("\r\n -------这是一个 ADC 实验------\r\n");
33.
34.     while (1)
35.     {
36.         ADC_ConvertedValueLocal =(float) ADC_ConvertedValue/4096*3.3;
    // 读取转换的 AD 值
37.
38.         printf("\r\n The current AD value = 0x%04X \r\n", ADC_Converte
    dValue);
39.         printf("\r\n The current AD value = %f V \r\n",ADC_ConvertedVa
    lueLocal);
40.
41.         Delay(0xffffee);                                    // 延时
42.
43.
44.     }
45. }
```

浏览一遍 main 函数，在调用了用户函数 *USART1_Config()* 及 *ADC1_Init()* 配置好串口和 ADC 之后，就可以直接使用保存了 ADC 转换值的变量 *ADC_ConvertedValue* 了，在 main 函数中并没有对 *ADC_ConvertedValue* 重新赋值，这个变量是在什么时候改变的呢？除了可能在中断服务函数修改了变量

值，就只有 DMA 有这样的能耐了，而且大家知道，在使用 DMA 传输时，由于不是内核执行的指令，所以修改变量值是绝对不会出现赋值语句的。

2.4.4 ADC 初始化

本实验代码中完全没有使用中断，而 ADC 及 DMA 的配置工作都由用户函数 *ADC1_Init()* 完成了。配置完成 ADC 及 DMA 后，ADC 就不停地采集数据，而 DMA 自动地把 ADC 采集得的数据转移至内存中的变量 *ADC_ConvertedValue* 中，所以在 *main* 函数的 while 循环中使用的 *ADC_ConvertedValue* 都是*实时值*。接下来重点分析 *ADC1_Init()*这个函数是如何配置 ADC 的。

ADC1_Init() 函数使能了 *ADC1*，并使 *ADC1* 工作于 *DMA* 方式。*ADC1_Init()* 这个函数是由在用户文件 *adc.c* 中实现的用户函数：

```
1.  /*
2.   * 函数名：ADC1_Init
3.   * 描述   ：无
4.   * 输入   ：无
5.   * 输出   ：无
6.   * 调用   ：外部调用
7.   */
8.  void ADC1_Init(void)
9.  {
10.     ADC1_GPIO_Config();
11.     ADC1_Mode_Config();
12. }
```

*ADC1_Init()*调用了 *ADC1_GPIO_Config()*和 *ADC1_Mode_Config()*。这两个函数的作用分别是配置好 ADC1 所用的 I/O 端口；配置 ADC1 初始化及 DMA 模式。

2.4.4.1 配置 GPIO 端口

*ADC1_GPIO_Config()*代码：

```
1.  /*
2.   * 函数名：ADC1_GPIO_Config
3.   * 描述   ：使能 ADC1 和 DMA1 的时钟，初始化 PC.01
4.   * 输入   ：无
5.   * 输出   ：无
6.   * 调用   ：内部调用
7.   */
8.  static void ADC1_GPIO_Config(void)
9.  {
```

```
10.     GPIO_InitTypeDef GPIO_InitStructure;
11.
12.     /* Enable DMA clock */
13.     RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
14.
15.     /* Enable ADC1 and GPIOC clock */
16.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph_GPIOC,
    ENABLE);
17.
18.     /* Configure PC.01  as analog input */
19.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
20.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
21.     GPIO_Init(GPIOC, &GPIO_InitStructure);    // PC1,输入时不用设置速率
22. }
```

ADC1_GPIO_Config() 代码非常简单，就是使能 DMA 时钟，GPIO 时钟及 ADC1 时钟。然后把 ADC1 的*通道 11* 使用的 GPIO 引脚 *PC1* 配置成*模拟输入模式*，在作为 ADC 的输入时，必须使用模拟输入。

这里涉及到 ADC 通道的知识，每个 ADC 通道都对应着一个 GPIO 引脚端口，GPIO 的引脚在设置为模拟输入模式后可用于模拟电压的输入端。STM32F103VET6 有三个 ADC，这三个 ADC 共用 16 个外部通道，从《STM32 数据手册》的引脚定义可找到 ADC 的通道与 GPIO 引脚的关系。见图 2-4

H1	F1	E8	8	15	26	PC0	I/O		PC0	ADC123_IN10	
H2	F2	F8	9	16	27	PC1	I/O		PC1	ADC123_IN11	
H3	E2	D6	10	17	28	PC2	I/O		PC2	ADC123_IN12	
H4	F3	-	11	18	29	PC3	I/O		PC3	ADC123_IN13	

图 2-4 部分 ADC 通道引脚图

表中的引脚名称标注中出现的 ADC12_INx(x 表示 4~9 或 14~15 之间的整数)，表示这个引脚可以是 ADC1_INx 或 ADC2_INx。例如：ADC12_IN9 表示这个引脚可以配置为 ADC1_IN9，也可以配置为 ADC2_IN9。

本实验中使用的 PC1 对应的默认复用功能为 *ADC123_IN11*，也就是说可以使用 *ADC1 的通道 11、ADC2 的通道 11 或 ADC3 的通道 11* 来采集 PC1 上的模拟电压数据，我们选择 ADC1 的通道 11 来采集。

2.4.4.2 配置 DMA

ADC 模式及其 DMA 传输方式都是在用户函数 *ADC1_Mode_Config()*中实现的。

*ADC1_Mode_Config()*函数代码如下：

```
/* 函数名：ADC1_Mode_Config
 * 描述  ：配置 ADC1 的工作模式为 MDA 模式
 * 输入  ：无
 * 输出  ：无
 * 调用  ：内部调用
 */
static void ADC1_Mode_Config(void)
{
    DMA_InitTypeDef DMA_InitStructure;
    ADC_InitTypeDef ADC_InitStructure;

    /* DMA channel1 configuration */
    DMA_DeInit(DMA1_Channel1);
    DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address;  /*ADC 地址*/
    DMA_InitStructure.DMA_MemoryBaseAddr = (u32)&ADC_ConvertedValue;/*内存地址*/
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC; //外设为数据源
    DMA_InitStructure.DMA_BufferSize = 1;
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;/*外设地址固定*/
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;  /*内存地址固定*/
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord; //半字
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;      //循环传输
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
    DMA_Init(DMA1_Channel1, &DMA_InitStructure);

    /* Enable DMA channel1 */
    DMA_Cmd(DMA1_Channel1, ENABLE);

    /* ADC1 configuration */

    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;  /*独立 ADC 模式*/
    ADC_InitStructure.ADC_ScanConvMode = DISABLE ;    /*禁止扫描模式，扫描模式用于多通道采集*/
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;  /*开启连续转换模式，即不停地进行 ADC 转换*/
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None; /*不使用外部触发转换*/
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;  /*采集数据右对齐*/
    ADC_InitStructure.ADC_NbrOfChannel = 1;      /*要转换的通道数目 1*/
    ADC_Init(ADC1, &ADC_InitStructure);

 /*配置 ADC 时钟，为 PCLK2 的 8 分频，即 9Hz*/
    RCC_ADCCLKConfig(RCC_PCLK2_Div8);
    /*配置 ADC1 的通道 11 为 55.5 个采样周期 */
    ADC_RegularChannelConfig(ADC1, ADC_Channel_11, 1, ADC_SampleTime_55Cycles5);

    /* Enable ADC1 DMA */
    ADC_DMACmd(ADC1, ENABLE);

    /* Enable ADC1 */
    ADC_Cmd(ADC1, ENABLE);

    /*复位校准寄存器 */
    ADC_ResetCalibration(ADC1);
```

```
53.     /*等待校准寄存器复位完成 */
54.     while(ADC_GetResetCalibrationStatus(ADC1));
55.
56.     /* ADC 校准 */
57.     ADC_StartCalibration(ADC1);
58.     /* 等待校准完成*/
59.     while(ADC_GetCalibrationStatus(ADC1));
60.
61.     /* 由于没有采用外部触发，所以使用软件触发 ADC 转换 */
62.     ADC_SoftwareStartConvCmd(ADC1, ENABLE);
63. }
```

ADC 的 DMA 配置部分跟串口 DMA 配置部分很类似，它的 DMA 整体上被配置为：

使用 DMA1 的通道 1，数据从 ADC 外设的数据寄存器(*ADC1_DR_Address*)转移到内存(*ADC_ConvertedValue 变量*)，内存、外设地址都固定，每次传输的数据大小为半字(16 位)，使用 DMA 循环传输模式。

其中 ADC1 外设的 DMA 请求通道为 DMA1 的通道 1，初始化时要注意。

DMA 传输的外设地址 *ADC1_DR_Address* 是一个自定义的宏：

```
1. #define ADC1_DR_Address    ((u32)0x40012400+0x4c)
```

ADC_DR 数据寄存器保存了 ADC 转换后的数值，以它作为 DMA 的传输源地址。它的地址是由 ADC1 外设的基地址(0x4001 2400) 加上 ADC 数据寄存器(ADC_DR)的地址偏移 (0x4c)计算得到的。见截自《SM32 参考手册》的说明图 2-6。

0x4001 2800 - 0x4001 2BFF	ADC2
0x4001 2400 - 0x4001 27FF	ADC1

图 2-5 ADC1 起始地址

11.12.14 ADC规则数据寄存器(ADC_DR)

地址偏移：0x4C

复位值：0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ADC2DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:16	**ADC2DATA[15:0]**：ADC2转换的数据 (ADC2 data) - 在ADC1中：双模式下，这些位包含了ADC2转换的规则通道数据。见11.9：双ADC模式 - 在ADC2和ADC3中：不使用这些位。
位15:0	**DATA[15:0]**：规则转换的数据 (Regular data) 这些位为只读，包含了规则通道的转换结果。数据是左对齐或右对齐，如图29和图30所示。

图 2-6 ADC_DR 寄存器地址偏移

2.4.4.3 配置 ADC 模式

从 *ADC1_Mode_Config()*函数代码的第 32 行开始，为 ADC 模式的配置，主要为对 ADC 的初始化结构体进行赋值。下面对这些结构体成员进行介绍：

1) *.ADC_Mode*

STM32 具有多个 ADC，而不同的 ADC 又是*共用通道*的，当两个 ADC 采集同一个通道的先后顺序、时间间隔不同，就演变出了各种各样的模式，如同步注入模式、同步规则模式等 10 种，根据应用要求选择适合的模式以适应采集数据的要求。

本实验用于测量电阻分压后的电压值，要求不高，只使用一个 ADC 就可以满足要求了，所以本成员被赋值为 *ADC_Mode_Independent (独立模式)*。

2) *.ADC_ScanConvMode*

当有多个通道需要采集信号时，可以把 ADC 配置为按一定的顺序来对各个通道进行扫描转换，即*轮流采集各通道的值*。若采集多个通道，必须开启此模式。

本实验只采集一个通道的信号，所以 *DISABLE(禁止)*使用扫描转换模式。

3) *.ADC_ContinuousConvMode*

连续转换模式，此模式与单次转换模式相反，单次转换模式 ADC 只采集一次数据就停止转换。而连续转换模式则在上一次 ADC 转换完成后，立即开启下一次转换。

本实验需要循环采集电压值，所以 ENABLE(使能)连续转换模式。

4) *.ADC_ExternalTrigConv*

ADC 需要在接收到*触发信号*才开始进行模数转换，这些触发信号可以是*外部中断触发(EXTI 线)、定时器触发*。这两个为外部触发信号，如果不使用外部触发信号可以使用*软件控制触发*。

本实验中使用软件控制触发所以该成员被赋值为 *ADC_ExternalTrigConv_None（不使用外部触发）*。

5) *.ADC_DataAlign*

数据对齐方式。ADC 转换后的数值是被保存到数据寄存器(ADC_DR)的 0~15 位或 16~32 位，数据*宽度为 16 位*，而 ADC 转换精度为 *12 位*。把 12 位的数据保存到 16 位的区域，就涉及*左对齐*和*右对齐*的问题。这里的左、右对齐跟 word 文档中的文本左、右对齐是一样的意思。

左对齐即 ADC 转换的数值最高位 D12 与存储区域的最高位 Bit 15 对齐，存储区域的低 4 位无意义。右对齐则相反，ADC 转换的数值最低位 D0 保存在存储区域的最低位 Bit 0 ，高 4 位无意义。见图 0-8。

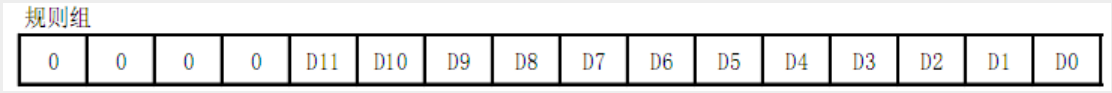

图 0-7 数据右对齐

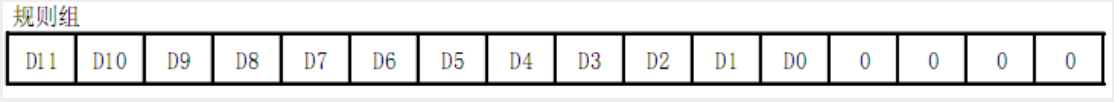

图 0-8 数据左对齐

本实验中 ADC 的转换值最后被保存在一个 16 位的变量之中，选择 *ADC_DataAlign_Right (右对齐)* 会比较方便。

6) *.ADC_NbrOfChannel*

这个成员保存了要进行 ADC 数据转换的通道数，可以为 1~16 个。

本实验中只需要采集 PC1 这个通道，所以把成员*赋值为 1* 就可以了。

填充完结构体，就可以调用外设初始化函数进行初始化了，ADC 的初始化使用 *ADC_Init()*函数，初始化完成后别忘记调用 *ADC_Cmd()* 函数来使能 ADC 外设，用 *ADC_DMACmd()* 函数来使能 ADC 的 DMA 接口。在本实验中初始化 ADC1。

2.4.4.4 ADC 转换时间配置

配置好了 ADC 的模式，还要设置 ADC 的时钟(ADCCLK)，ADC 时钟频率越高，转换速度也就越快，但 *ADC 时钟有上限值，不能超过 14MHz*。

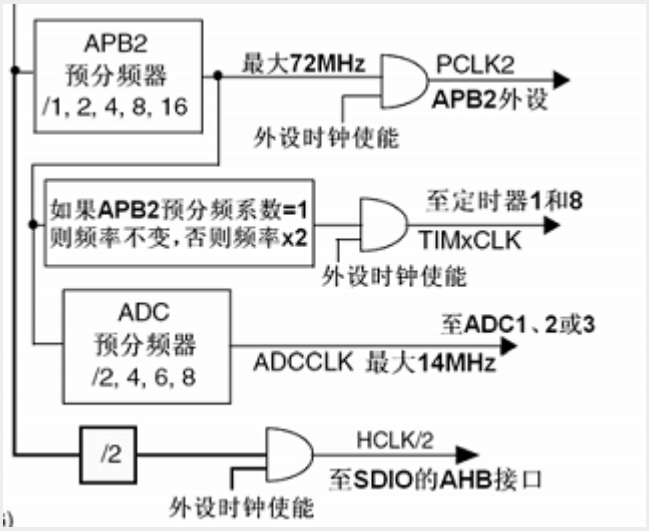

图 0-9 ADC 时钟

配置 ADC 时钟，可使用函数 *RCC_ADCCLKConfig()*配置，见图 2-10

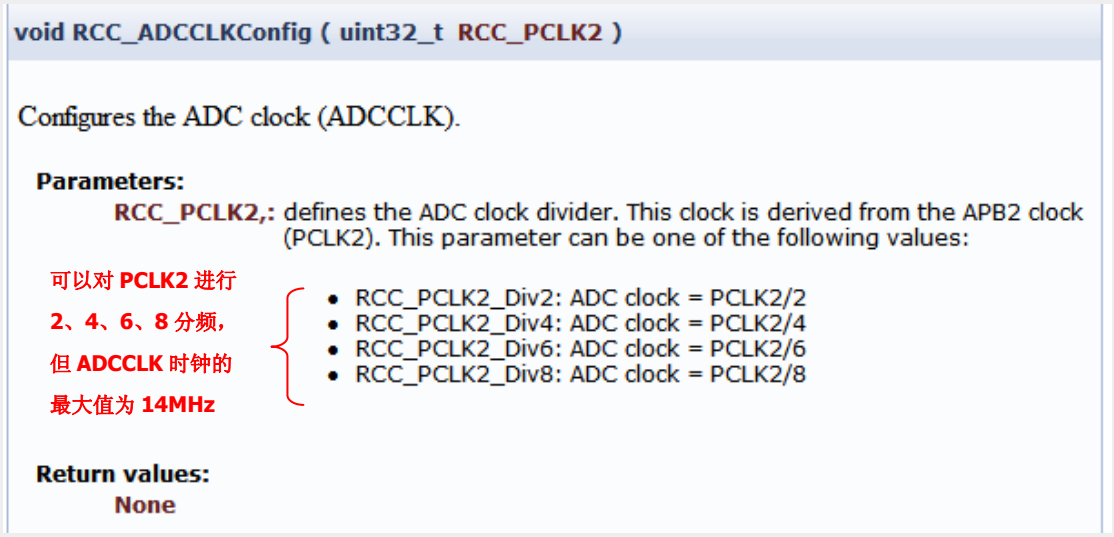

图 2-10 ADCCLK 时钟配置函数

*ADC 的时钟(ADCCLK)为 ADC 预分频器的输出，而 ADC 预分频器的输入则为高速外设时钟(PCLK2)。*使用 *RCC_ADCCLKConfig()*库函数实质就是设置 ADC 预分频器的分频值，可设置为 PCLK2 的 2、4、6、8 分频。

PCLK2 的常用时钟频率为 72MHz，而 ADCCLK 必须低于 14MHz，所以在这个情况下，ADCCLK 最高频率为 PCLK2 的 8 分频，即 ADCCLK=9MHz。若希望使 ADC 以最高的频率 14MHz 运行，可以把 PCLK2 配置为 56MHz，然后再 4 分频得到 ADCCLK。

ADC 的转换时间不仅与 ADC 的时钟有关，还与采样周期相关。

每个不同的 ADC 通道，都可以设置为不同的的采样周期。配置时用库函数 *ADC_RegularChannelConfig()*，它的说明见图 2-11。

```
void ADC_RegularChannelConfig ( ADC_TypeDef * ADCx,
                                uint8_t       ADC_Channel,
                                uint8_t       Rank,
                                uint8_t       ADC_SampleTime
                              )
```

Configures for the selected ADC regular channel its corresponding rank in the sequencer and its sample time.

Parameters:

ADCx,: where x can be 1, 2 or 3 to select the ADC peripheral.

ADC_Channel,: the ADC channel to configure. This parameter can be one of the following values:

选择要配置的 ADC 通道

- ADC_Channel_0: ADC Channel0 selected
- ADC_Channel_1: ADC Channel1 selected
- ADC_Channel_2: ADC Channel2 selected
- ADC_Channel_3: ADC Channel3 selected
- ADC_Channel_4: ADC Channel4 selected
- ADC_Channel_5: ADC Channel5 selected
- ADC_Channel_6: ADC Channel6 selected
- ADC_Channel_7: ADC Channel7 selected
- ADC_Channel_8: ADC Channel8 selected
- ADC_Channel_9: ADC Channel9 selected
- ADC_Channel_10: ADC Channel10 selected
- ADC_Channel_11: ADC Channel11 selected
- ADC_Channel_12: ADC Channel12 selected
- ADC_Channel_13: ADC Channel13 selected
- ADC_Channel_14: ADC Channel14 selected
- ADC_Channel_15: ADC Channel15 selected
- ADC_Channel_16: ADC Channel16 selected
- ADC_Channel_17: ADC Channel17 selected

RANK 参数：配置的数值为多通道扫描时，此通道的采样顺序

Rank,: The rank in the regular group sequencer. This parameter must be between 1 to 16.

ADC_SampleTime,: The sample time value to be set for the selected channel. This parameter can be one of the following values:

可配置的采样周期：1.5、7.5、13.5 等 ADC 时钟周期

- ADC_SampleTime_1Cycles5: Sample time equal to 1.5 cycles
- ADC_SampleTime_7Cycles5: Sample time equal to 7.5 cycles
- ADC_SampleTime_13Cycles5: Sample time equal to 13.5 cycles
- ADC_SampleTime_28Cycles5: Sample time equal to 28.5 cycles
- ADC_SampleTime_41Cycles5: Sample time equal to 41.5 cycles
- ADC_SampleTime_55Cycles5: Sample time equal to 55.5 cycles
- ADC_SampleTime_71Cycles5: Sample time equal to 71.5 cycles
- ADC_SampleTime_239Cycles5: Sample time equal to 239.5 cycles

Return values:

None

图 2-11 规则通道配置函数

*ADC_RegularChannelConfig()*函数中的 *RANK* 值是指在*多通道扫描模式*时，本通道的*扫描顺序*。例如通道 1、4、7 的 RANK 值分别为被配置为 3、2、1 的话，在 ADC 扫描时，扫描的顺序为通道 7、通道 4、最后扫描通道 1。

本实验中只采集一个 ADC 通道，把以把 ADC1 的通道 11RANK 值配置为 1。

*ADC_SampleTime*的参数值则用于配置本通道的*采样周期*，最短可配置为 1.5 个采样周期，这里的周期指 ADCCLK 时钟周期。

本实验中把 ADC1 通道 11 配置为 55.5 个采样周期，而 ADCCLK 在前面已经配置为 9MHz，根据 STM32 的 *ADC 采样时间计算公式*：

$$T_{CONV} = 采样周期 + 12.5 个周期$$

公式中的采样周期就是本函数中配置的 *ADC_SampleTime*，而后边加上的 12.5 个周期为固定的数值。

所以，本实验中 ADC1 通道 11 的转换时间 $T_{CONV} = (55.5+12.5)*1/9 \approx 7.56us$ 。

2.4.4.4.5 ADC 自校准

在开始 ADC 转换之前，需要启动 ADC 的自校准。ADC 有一个内置自校准模式，校准可大幅减小因内部电容器组的变化而造成的准精度误差。在校准期间，在每个电容器上都会计算出一个误差修正码(数字值)，这个码用于消除在随后的转换中每个电容器上产生的误差。

以下为在 *ADC1_Mode_Config()*函数中的 ADC 自校准时调用的库函数和使用步骤。

```
1.  /*复位校准寄存器 */
2.  ADC_ResetCalibration(ADC1);
3.  /*等待校准寄存器复位完成 */
4.  while(ADC_GetResetCalibrationStatus(ADC1));
5.
6.  /* ADC 校准 */
7.  ADC_StartCalibration(ADC1);
8.  /* 等待校准完成*/
9.  while(ADC_GetCalibrationStatus(ADC1));
```

在调用了复位校准函数 *ADC_ResetCalibration()*和开始校准函数 *ADC_StartCalibration()*后，要检查标志位等待校准完成，确保完成后才开始 ADC 转换。建议在每次上电后都进行一次自校准。

在校准完成后，就可以开始进行 ADC 转换了。本实验代码中配置的 ADC 模式为软件触发方式，我们可以调用库函数 *ADC_SoftwareStartConvCmd()*来开启软件触发。其说明见图 2-12。

```
void ADC_SoftwareStartConvCmd ( ADC_TypeDef * ADCx,
                                FunctionalState NewState
                              )
```

Enables or disables the selected ADC software start conversion .

Parameters:

ADCx,: where x can be 1, 2 or 3 to select the ADC peripheral.

NewState,: new state of the selected ADC software start conversion. This parameter can be: ENABLE or DISABLE.

Return values:

None

图 2-12 ADC 软件触发控制函数

调用这个函数使能了 ADC1 的软件触发后，ADC 就开始进行转换了，每次转换完成后，由 DMA 控制器把转换值从 ADC 数据寄存器(ADC_DR)中转移到变量 *ADC_ConvertedValue* 中，当 DMA 传输完成后，在 main 函数中使用的 *ADC_ConvertedValue* 的内容就是 ADC 转换值了。

2.4.4.4.6 volatile 变量

现在我们再来认识 *ADC_ConvertedValue* 这个变量：

```
1.  // ADC1 转换的电压值通过 MDA 方式传到 flash
2.  extern __IO u16 ADC_ConvertedValue;
3.
```

ADC_ConvertedValue 在 文件 *adc.c* 中定义，这个。这里要注意一点的是，这个变量要用 c 语言的 *volatile* 关键字来修饰，为的是让编译器不要去优化这个变量。这样每次用到这两个变量时都要回到相应变量的内存中去取值，而 *volatile* 字面意思就是*“可变的，不确定的”*。

例如：不使用 *volatile* 关键字修饰的变量 a 在被访问的时候可能会直接从 *CPU 的寄存器*中取出（因为之前变量 a 被访问过，也就是说之前就从内存中取出 a 的值保存到某个 CPU 寄存器中），之所以直接从寄存器中取值，而不去内存中取值，是因为编译器优化代码的结果（访问 CPU 寄存器比访问内存快的多）。*这里的 CPU 寄存器指 R0、R1 等 CPU 通用寄存器，用于 CPU 运算是暂存数据的，不是指外设中的寄存器。* ，

用 *volatile* 声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。

因为 *ADC_ConvertedValue* 这个变量值随时都是会 *被 DMA 控制器改变*的，所以我们用 *volatile* 来修饰它，确保每次读取到的都是实时的 ADC 转换值。

2.4.5 计算电压值

回到 main 函数中循环打印电压值部分：

```
1.  while (1)
2.  {
3.      ADC_ConvertedValueLocal =(float) ADC_ConvertedValue/4096*3.3; //
    读取转换的 AD 值
4.
5.      printf("\r\n The current AD value = 0x%04X \r\n", ADC_ConvertedVal
    ue);
6.      printf("\r\n The current AD value = %f V \r\n",ADC_ConvertedValueL
    ocal);
7.
8.      Delay(0xffffee);                                    // 延时
9.
10.
11. }
```

float 型变量 *ADC_ConvertedValueLocal* 保存了由转换值计算出来的电压值，其计算公式是 ADC 通用的：

实际电压值 = ADC 转换值*LSB

STM32 的 ADC 的精度为 12 位，而野火板子中 V_{ref+}接的参考电压值为 3.3v，所以 LSB =3.3/2^{12} 。

2.4.6 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：

当旋转开发板开发板上的滑动变阻器时，ADC1 转换的电压值则会改变。板载的是 20K 的精密电阻，旋转的圈数要多点才能看到 ADC 值的明显变化。

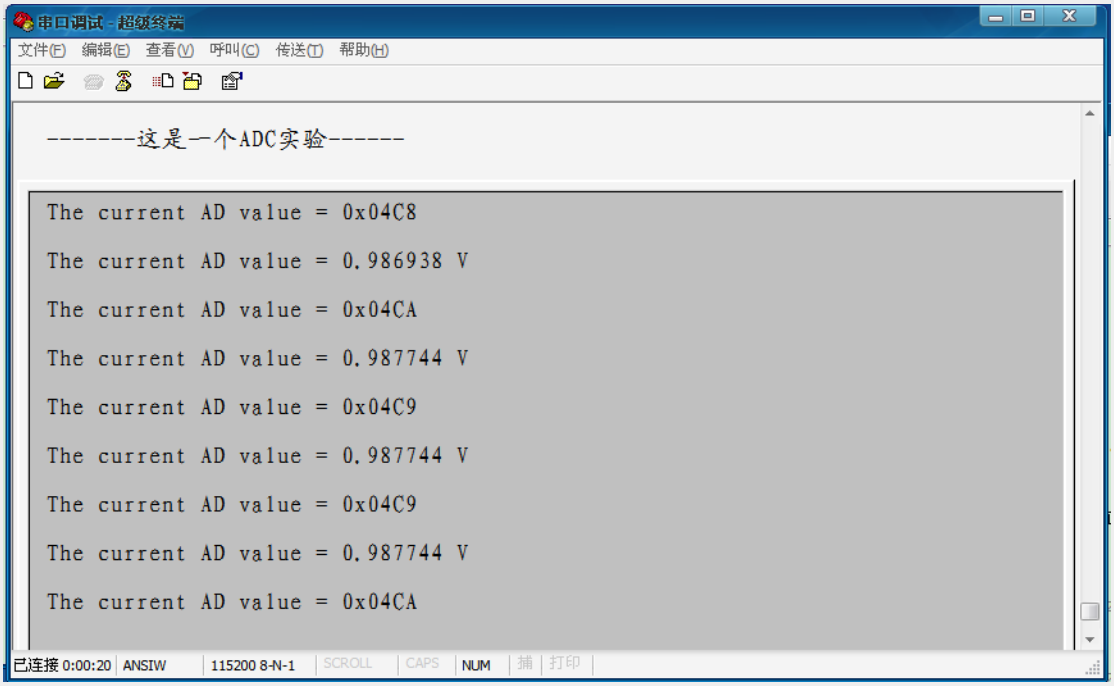

图 2-13 ADC 采集外部电压实验