第 5 章

CHAPTER 5

ADC 开发

ADC(Analog To Digital Converter,模数转换器)指将连续变化的模拟信号转换为离散的数字信号的器件。STM32F4 系列一般有 3 个 ADC,每个 ADC 有 12 位、10 位、8 位和 6 位精度可选,每个 ADC 有 16 个外部通道、19 个复用通道、两个用于内部源和 VBAT 通道的信号。这些通道的 AD 转换可在单次、连续、扫描或不连续采样模式下进行。ADC 的结果存储在一个左对齐或右对齐的 16 位数据寄存器中。ADC 具有独立模式、双重模式和三重模式,对于不同的 AD 转换要求几乎都有合适的模式可选。

本章从采集光照强度、单 ADC 扫描、ADC 的 DMA 模式、双 ADC 交叉、ADC 定时器触发,逐一展开,介绍相关原理并通过实例帮助读者掌握 ADC 开发能力。

视频 18

5.1 ADC:采集光照强度

学习目标

掌握 ARM Cortex-M 系列芯片外设 ADC 的工作原理,通过配置 STM32F407 的 ADC 来完成光敏传感器电压采集。

5.1.1 开发原理

1. ADC 介绍

STM32F407ZGT6 有 3 个 ADC,每个 ADC 有 12 位、10 位、8 位和 6 位精度可选,每个 ADC 有 16 个外部通道。另外还有用于内部 ADC 源和 VBAT 的通道。ADC 具有独立模式、双重模式和三重模式。ADC 功能非常强大。

ADC 功能框图如图 5-1 所示(详情请参考 STM32F4 参考手册)。

1) 编号①电压输入范围

ADC 输入范围为 VREF−≤VIN≤VREF+。由 VREF−、VREF+、VDDA、VSSA 这 4 个外部引脚决定。我们在设计原理图的时候一般把 VSSA 和 VREF− 接地,把 VREF+ 和 VDDA 接 3V3,得到 ADC 的输入电压范围为 0～3.3V。

如果想让输入的电压范围变宽,可以测试负电压或者更高的正电压,那么可以在外部加一个电压调理电路,把需要转换的电压抬升或者降压到 0～3.3V,这样 ADC 就可以测量了。

2) 编号②输入通道

在确定好 ADC 输入电压之后,电压怎么输入到 ADC? 这里引入通道的概念。STM32 的 ADC 有多达 19 个通道,其中外部的 16 个通道就是图 5-1 中的 ADCx_IN0、ADCx_IN1、……、ADCx_IN5。这 16 个通道对应着不同的 I/O 口,具体是哪一个 I/O 口可以从手册查询到。其

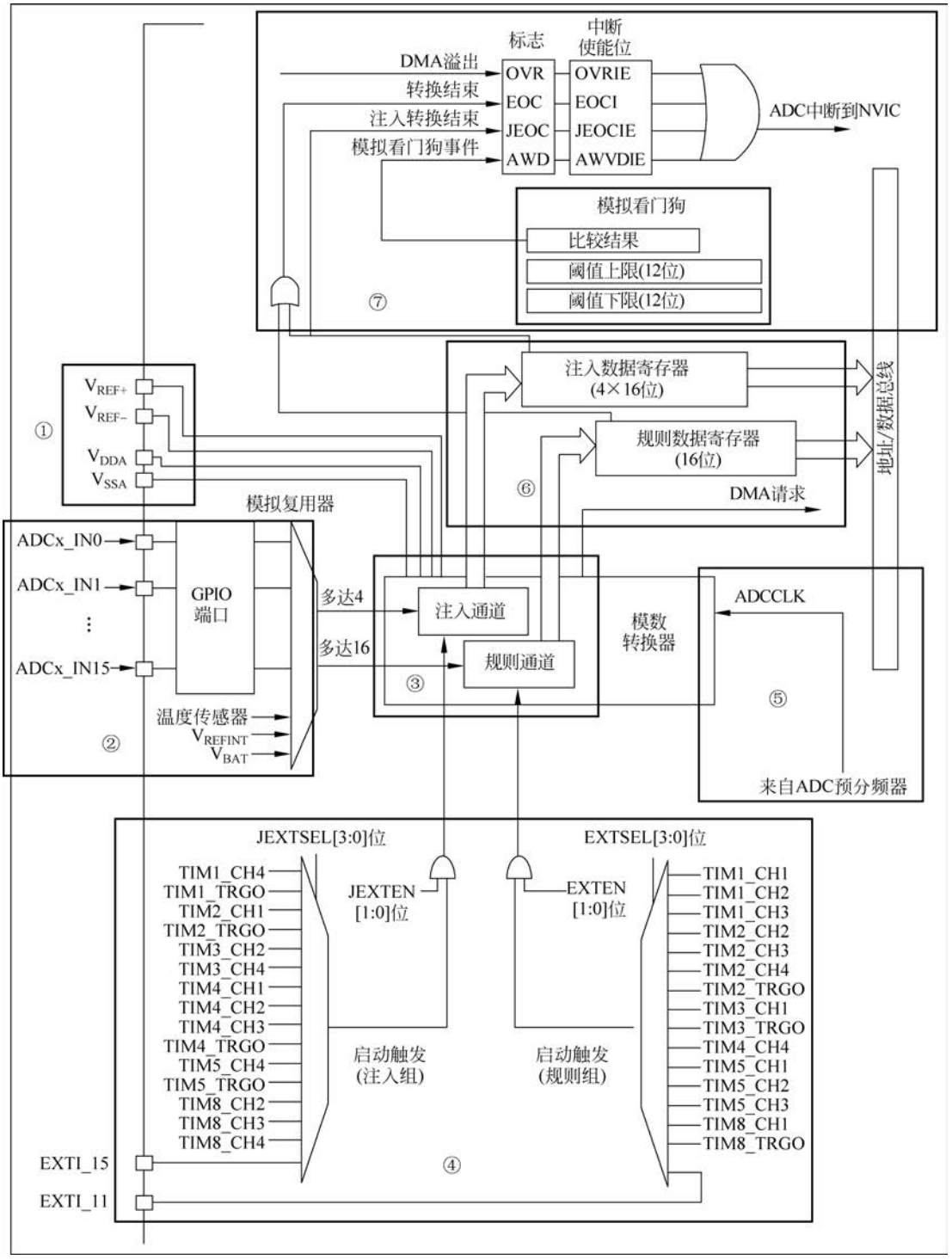

标志
中断使能位
DMA溢出
转换结束
注入转换结束
模拟看门狗事件
OVR
EOC
JEOC
AWD
OVRIE
EOCI
JEOCIE
AWVDIE
ADC中断到NVIC
模拟看门狗
比较结果
阈值上限(12位)
阈值下限(12位)
⑦
注入数据寄存器
(4×16位)
规则数据寄存器
(16位)
⑥
DMA请求
地址/数据总线
VREF+
VREF-
VDDA
VSSA
①
模拟复用器
ADCx_IN0
ADCx_IN1
ADCx_IN15
GPIO端口
温度传感器
VREFINT
VBAT
②
多达4
多达16
注入通道
规则通道
③
模数转换器
ADCCLK
⑤
来自ADC预分频器
JEXTSEL[3:0]位
EXTSEL[3:0]位
TIM1_CH4
TIM1_TRGO
TIM2_CH1
TIM2_TRGO
TIM3_CH2
TIM3_CH4
TIM4_CH1
TIM4_CH2
TIM4_CH3
TIM4_TRGO
TIM5_CH4
TIM5_TRGO
TIM8_CH2
TIM8_CH3
TIM8_CH4
JEXTEN[1:0]位
EXTEN[1:0]位
TIM1_CH1
TIM1_CH2
TIM1_CH3
TIM2_CH2
TIM2_CH3
TIM2_CH4
TIM2_TRGO
TIM3_CH1
TIM3_TRGO
TIM4_CH4
TIM5_CH1
TIM5_CH2
TIM5_CH3
TIM8_CH1
TIM8_TRGO
启动触发(注入组)
启动触发(规则组)
EXTI_15
EXTI_11
④

图 5-1 ADC 功能框图

中 ADC1/ADC2/ADC3 还有内部通道：ADC1 的通道 ADC1_IN16 连接到内部的 VSS，通道 ADC1_IN17 连接到了内部参考电压 VREFINT，通道 ADC1_IN18 连接到了芯片内部的温度传感器或者备用电源 VBAT。ADC2 和 ADC3 的通道 16、17、18 全部连接到了内部的 VSS，如图 5-2 所示。

STM32F407ZGT6 ADC I/O 分配					
ADC1	IO	ADC2	IO	ADC3	IO
通道0	PA0	通道0	PA0	通道0	PA0
通道1	PA1	通道1	PA1	通道1	PA1
通道2	PA2	通道2	PA2	通道2	PA2
通道3	PA3	通道3	PA3	通道3	PA3
通道4	PA4	通道4	PA4	通道4	PF6
通道5	PA5	通道5	PA5	通道5	PF7
通道6	PA6	通道6	PA6	通道6	PF8
通道7	PA7	通道7	PA7	通道7	PF9
通道8	PB0	通道8	PB0	通道8	PF10
通道9	PB1	通道9	PB1	通道9	PF3
通道10	PC0	通道10	PC0	通道10	PC0
通道11	PC1	通道11	PC1	通道11	PC1
通道12	PC2	通道12	PC2	通道12	PC2
通道13	PC3	通道13	PC3	通道13	PC3
通道14	PC4	通道14	PC4	通道14	PF4
通道15	PC5	通道15	PC5	通道15	PF5
通道16	连接内部温度传感器	通道16	连接内部VSS	通道16	连接内部VSS
通道17	连接内部VREFINT	通道17	连接内部VSS	通道17	连接内部VSS

图 5-2 ADC 输入通道

外部的 16 个通道在转换的时候又分为规则通道和注入通道，其中规则通道最多有 16 路，注入通道最多有 4 路。这两个通道的区别如下：

(1) 规则通道。

顾名思义，规则通道就是遵守规则的通道，我们平时一般使用的就是这个通道。

(2) 注入通道。

注入，可以理解为插入、插队的意思，这是一种不安分的通道。它是一种在规则通道转换的时候可强行插入转换的一种通道。如果在规则通道转换过程中，有注入通道插队，那么要先转换完注入通道，等注入通道转换完成后，再回到规则通道的转换流程。这一点与中断程序处理很像。所以，注入通道只有在规则通道存在时才会出现。

3) 编号③转换顺序

(1) 规则序列。

规则序列寄存器有 3 个，分别为 SQR3、SQR2、SQR1。SQR3 控制着规则序列中的第 1 个到第 6 个转换，对应的位为 SQ1[4:0]～SQ6[4:0]，第一次转换的是位 SQ1[4:0]，如果通道 16 想第一次转换，那么在 SQ1[4:0]写 16 即可。SQR2 控制着规则序列中的第 7 到第 12 个转换，对应的位为 SQ7[4:0]～SQ12[4:0]，如果通道 1 向第 8 个转换，则 SQ8[4:0]写 1 即可。SQR1 控制着规则序列中的第 13 到第 16 个转换，对应位为 SQ13[4:0]～SQ16[4:0]；如果通道 6 向第 10 个转换，则 SQ10[4:0]写 6 即可。具体使用多少个通道，由 SQR1 的位 L[3:0]决定，最多 16 个通道，如图 5-3 所示。

规则序列寄存器SQRx(x为1，2，3)			
寄存器	寄存器位	功能	取值
SQR3	SQ1[4:0]	设置第1个转换的通道	1~16
	SQ2[4:0]	设置第2个转换的通道	1~16
	SQ3[4:0]	设置第3个转换的通道	1~16
	SQ4[4:0]	设置第4个转换的通道	1~16
	SQ5[4:0]	设置第5个转换的通道	1~16
	SQ6[4:0]	设置第6个转换的通道	1~16
SQR2	SQ7[4:0]	设置第7个转换的通道	1~16
	SQ8[4:0]	设置第8个转换的通道	1~16
	SQ9[4:0]	设置第9个转换的通道	1~16
	SQ10[4:0]	设置第10个转换的通道	1~16
	SQ11[4:0]	设置第11个转换的通道	1~16
	SQ12[4:0]	设置第12个转换的通道	1~16
SQR1	SQ13[4:0]	设置第13个转换的通道	1~16
	SQ14[4:0]	设置第14个转换的通道	1~16
	SQ15[4:0]	设置第15个转换的通道	1~16
	SQ16[4:0]	设置第16个转换的通道	1~16
	SQL[3:0]	需要转换多少个通道	1~16

图 5-3 规则序列寄存器

(2) 注入序列。

注入序列寄存器 JSQR 只有一个，最多支持 4 个通道，具体多少个由 JSQR 的 JL[2:0]决定。如果 JL 的值小于 4，则 JSQR 与 SQR 决定转换顺序的设置不一样，第一次转换的不是 JSQR1[4:0]，而是 JSQRx[4:0]，x=(4－JL)，与 SQR 刚好相反。如果 JL=00(1 个转换)，那么转换的顺序是从 JSQR4[4:0]开始，而不是从 JSQR1[4:0]开始，这个要注意，编程的时候不要搞错。当 JL 等于 4 时，与 SQR 一样，如图 5-4 所示。

注入序列寄存器JSQR			
寄存器	寄存器位	功能	取值
JSQR	JSQ1[4:0]	设置第1个转换的通道	1~4
	JSQ2[4:0]	设置第2个转换的通道	1~4
	JSQ3[4:0]	设置第3个转换的通道	1~4
	JSQ4[4:0]	设置第4个转换的通道	1~4
	JL[1:0]	需要转换多少个通道	1~4

图 5-4 注入序列寄存器

4) 触发源

通道选好了，转换的顺序也设置好了，接下来就该开始转换了。ADC 转换可以由 ADC 控制寄存器 ADC_CR2 的 ADON 这个位来控制，写 1 的时候开始转换，写 0 的时候停止转换，这是最简单、最好理解的开启 ADC 转换的控制方式。

除了上面的控制方法，ADC 还支持外部事件触发转换，这个触发包括内部定时器触发和外部 I/O 触发。触发源有很多，具体选择哪一种触发源，由 ADC 控制寄存器 ADC_CR2 的 EXTSEL[2:0]和 JEXTSEL[2:0]位来控制。EXTSEL[2:0]用于选择规则通道的触发源，JEXTSEL[2:0]用于选择注入通道的触发源。选定好触发源之后，触发源是否需要被触发，则由 ADC 控制寄存器 ADC_CR2 的 EXTTRIG 和 JEXTTRIG 这两位来决定。

如果使能了外部触发事件，则可以通过设置 ADC 控制寄存器 ADC_CR2 的 EXTEN[1:0]和 JEXTEN[1:0]来控制触发极性，可以有 4 种状态，分别是禁止触发检测、上升沿检测、下降沿

检测以及上升沿和下降沿均检测。

5) 转换时间

(1) ADC 时钟。

ADC 输入时钟 ADC_CLK 由 PCLK2 经过分频产生,最大值为 36MHz,典型值为 30MHz,分频因子由 ADC 通用控制寄存器 ADC_CCR 的 ADCPRE[1:0]设置,可设置的分频系数有 2、4、6 和 8。注意,这里没有 1 分频。

(2) 采样时间。

ADC 需要若干个 ADC_CLK 周期完成对输入的电压进行采样,采样的周期数可通过 ADC 采样时间寄存器 ADC_SMPR1 和 ADC_SMPR2 中的 SMP[2:0]位设置,ADC_SMPR2 控制的是通道 0～9,ADC_SMPR1 控制的是通道 10～17。每个通道可以分别用不同的时间采样。其中采样周期最小是 3 个,即如果要以最快速度采样,那么应该设置采样周期为 3 个周期,这里说的周期就是 1/ADC_CLK。

ADC 的总转换时间跟 ADC 的输入时钟和采样时间有关,公式为:

$$Tconv = 采样时间 + 12\ 个周期$$

当 ADC_CLK=30MHz,即 PCLK2 为 60MHz,ADC 时钟为 2 分频,采样周期设置为 3 个周期,那么总的转换时为 Tconv=3+12=15 个周期=0.5μs。

一般设置 PCLK2=84MHz,经过 ADC 预分频器能分频到最大的时钟只能是 21MHz,采样周期设置为 3 个周期,算出最短的转换时间为 0.7142μs,这个才是最常用的。

6) 数据寄存器

一切准备就绪后,ADC 转换后的数据根据转换组的不同存放位置不同,规则组的数据放在 ADC_DR 寄存器,注入组的数据放在 JDRx。如果是使用双重或者三重模式,那么规则组的数据是存放在通用规则寄存器 ADC_CDR 内的。

(1) 规则数据寄存器 ADC_DR。

ADC 规则组数据寄存器 ADC_DR 只有一个,是一个 32 位的寄存器,只有低 16 位有效并且只是用于以独立模式存放转换完成的数据。因为 ADC 的最大精度是 12 位,ADC_DR 是 16 位有效,所以允许 ADC 存放数据时候选择左对齐或者右对齐,具体是以哪一种方式存放,由 ADC_CR2 的 11 位 ALIGN 设置。假如设置 ADC 精度为 12 位,如果设置数据为左对齐,那么 AD 转换完成的数据存放在 ADC_DR 寄存器的位[4:15]内; 如果为右对齐,则存放在 ADC_DR 寄存器的位[0:11]内。

规则通道可以有 16 个,可规则数据寄存器只有一个,如果使用多通道转换,那么转换的数据就全部都挤在了 DR 里面,前一个时间点转换的通道数据,就会被下一个时间点的另外一个通道转换的数据覆盖掉,所以通道转换完成后就应该把数据取走,或者开启 DMA 模式,把数据传输到内存里面,否则就会造成数据的覆盖。最常用的做法就是开启 DMA 传输。

如果没有使用 DMA 传输,那么一般需要使用 ADC 状态寄存器 ADC_SR 获取当前 ADC 转换的进度状态,进而进行程序控制。

(2) 注入数据寄存器 ADC_JDRx。

ADC 注入组最多有 4 个通道,刚好注入数据寄存器也有 4 个,每个通道对应着自己的寄存器,不会像规则寄存器那样产生数据覆盖的问题。ADC_JDRx 是 32 位的,低 16 位有效,高 16 位保留,数据同样分为左对齐和右对齐,具体以哪一种方式存放,由 ADC_CR2 的 11 位 ALIGN 设置。

(3) 通用规则数据寄存器 ADC_CDR。

规则数据寄存器 ADC_DR 是仅适用于独立模式的,而通用规则数据寄存器 ADC_CDR 是适用于双重模式和三重模式的。独立模式就是仅使用 3 个 ADC 的其中一个,双重模式就是同时使用 ADC1 和 ADC2,而三重模式就是同时使用 3 个 ADC。在双重或者三重模式下一般需要配合 DMA 数据传输使用。

7) 中断

(1) 转换结束中断。

数据转换结束后,可以产生中断,中断分为 4 种:规则通道转换结束中断、注入转换通道转换结束中断、模拟看门狗中断和溢出中断。其中转换结束中断很好理解,跟我们平时接触的中断一样,有相应的中断标志位和中断使能位,还可以根据中断类型编写与之配套的中断服务程序。

(2) 模拟看门狗中断。

当被 ADC 转换的模拟电压低于低阈值或者高于高阈值时,就会产生中断,前提是开启了模拟看门狗中断,其中低阈值和高阈值由 ADC_LTR 和 ADC_HTR 设置。例如,设置高阈值是 2.5V,那么模拟电压超过 2.5V 的时候,就会产生模拟看门狗中断;低阈值时也一样。

(3) 溢出中断。

如果发生 DMA 传输数据丢失,则会对 ADC 状态寄存器 ADC_SR 的 OVR 位置位;如果同时使能了溢出中断,那么在转换结束后会产生一个溢出中断。

(4) DMA 请求。

规则和注入通道转换结束后,除了产生中断外,还可以产生 DMA 请求,把转换好的数据直接存储在内存里面。对于 3 种模式的 AD 转换使用 DMA 传输非常有必要,程序编程简化了很多。

8) 电压转换

模拟电压经过模数转换后,得到的是一个相对精度的数字值,如果通过串口以十六进制数据输出,可读性比较差,那么有时候就需要把数字电压转换成模拟电压,这样也可以与实际的模拟电压(用万用表测)对比,看看转换是否准确。

我们一般在设计原理图的时候会把 ADC 的输入电压范围设定为 0～3.3V,如果设置 ADC 为 12 位的,那么 12 位满量程对应的就是 3.3V,12 位满量程对应的数字值是 2^{12}。数值 0 对应的就是 0V。如果转换后的数值为 X,X 对应的模拟电压为 Y,那么会有如下等式成立:

$$2^{12}/3.3 = X/Y, => Y = (3.3 \times X)/2^{12}$$

2. 光敏传感器简介

光敏传感器是利用光敏元件将光信号转换为电信号的传感器,它的敏感波长在可见光波长附近,包括红外线波长和紫外线波长。光传感器不只局限于对光的探测,它还可以作为探测元件组成其他传感器,对许多非电量进行检测,只要将这些非电量转换为光信号的变化即可。

STM32F4 开发板板载了一个光敏二极管(光敏电阻),作为光敏传感器,它对光的变化非常敏感。光敏二极管也称为光电二极管。光敏二极管与半导体二极管在结构上是类似的,其管芯是一个具有光敏特征的 PN 结,具有单向导电性,因此工作时需加上反向电压。无光照时,有很小的饱和反向漏电流,即暗电流,此时光敏二极管截止。当受到光照时,饱和反向漏电流大大增加,形成光电流,它随入射光强度的变化而变化。当光线照射 PN 结时,可以使 PN 结中产生电子-空穴对,使少数载流子的密度增加。这些载流子在反向电压下漂移,使反向电

流增加。因此可以利用光照强弱来改变电路中的电流。

利用这个电流变化,我们串接一个电阻,就可以转换成电压的变化,从而通过 ADC 读取电压值,判断外部光线的强弱。

5.1.2 开发步骤

(1) 查找开发板电路原理图可知,光敏二极管连接在 PF7 引脚上,如图 5-5 所示。

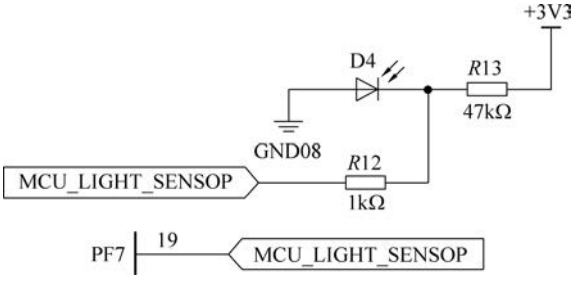


图 5-5 光敏二极管连接图

(2) 查找 STM32F4 数据手册可知,PF7 引脚需使用 ADC3 的通道 5,如图 5-6 所示。

—	—	—	19	K1	25	PF7	I/O	FT	—	TIM11_CH1/FSMC_NREG/EVENTOUT	ADC3_IN5

图 5-6 ADC 引脚映射

(3) 定义 ADC3 初始化结构体句柄 ADC3_Handler,并创建 ADC3_Init()函数,初始化 ADC3。首先调用函数 HAL_ADC_Init()初始化 ADC3,此处配置 ADC 为 4 分频,ADC 时钟为 21MHz,并配置 ADC 为软件触发,且为 12 位数据模式。然后调用 HAL_ADC_ConfigChannel()函数,初始化 ADC3 的通道 5。

```
ADC_HandleTypeDef ADC3_Handler;
//ADC 初始化函数
void ADC3_Init(void)
{
    ADC3_Handler.Instance = ADC3;
    ADC3_Handler.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;  //4 分频,ADCCLK = PCLK2/4
= 84/4 = 21MHz
    ADC3_Handler.Init.Resolution = ADC_RESOLUTION_12B;            //12 位模式
    ADC3_Handler.Init.DataAlign = ADC_DATAALIGN_RIGHT;            //右对齐
    ADC3_Handler.Init.ScanConvMode = DISABLE;                     //扫描模式
    ADC3_Handler.Init.EOCSelection = ADC_EOC_SINGLE_CONV;         //开启 EOC 转换一次中断
    ADC3_Handler.Init.ContinuousConvMode = DISABLE;               //开启连续转换
    ADC3_Handler.Init.NbrOfConversion = 1;                        //1 个转换在规则序列
    ADC3_Handler.Init.ExternalTrigConv = ADC_SOFTWARE_START;      //软件触发
    ADC3_Handler.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE; //关闭外部触发器
    ADC3_Handler.Init.DMAContinuousRequests = DISABLE;            //关闭 DMA 请求
    HAL_ADC_Init(&ADC3_Handler);                                  //初始化

    ADC_ChannelConfTypeDef ADC_ChanneConf;
    //电压采集通道初始化
    ADC_ChanneConf.Channel = ADC_CHANNEL_5;                       //通道 5
    ADC_ChanneConf.Rank = 1;                                      //第一个采样
```

```
    ADC_ChanneConf.SamplingTime = ADC_SAMPLETIME_480CYCLES;       //周期采样时间
    HAL_ADC_ConfigChannel(&ADC3_Handler, &ADC_ChanneConf);
}
```

(4) 重定义 HAL_ADC_MspInit()函数,用来初始化 PF7 引脚,此处设置引脚为模拟模式,然后使能 ADC 时钟。

```
//定义 ADC 底层驱动
void HAL_ADC_MspInit(ADC_HandleTypeDef * hadc)
{
    GPIO_InitTypeDef GPIO_Initure;

    __HAL_RCC_ADC3_CLK_ENABLE();              //使能 ADC3 时钟
    __HAL_RCC_GPIOF_CLK_ENABLE();             //开启 GPIOF 时钟

    GPIO_Initure.Pin = GPIO_PIN_7;            //PF7
    GPIO_Initure.Mode = GPIO_MODE_ANALOG;     //模拟
    GPIO_Initure.Pull = GPIO_NOPULL;          //浮空
    HAL_GPIO_Init(GPIOF, &GPIO_Initure);
}
```

(5) 创建 Light_Value()函数,用来获取传感器测量电压值。函数首先调用 HAL_ADC_Start()开启模数转换,然后调用 HAL_ADC_GetValue()函数,读取 ADC 测量值。

```
//获取 ADC 电压值
uint16_t Light_Value(void)
{
    HAL_ADC_Start(&ADC3_Handler);                       //开始 ADC3

    return (uint16_t)HAL_ADC_GetValue(&ADC3_Handler);  //获取 ADC 电压值
}
```

(6) 主函数 main()程序如下:

第一步,初始化系统时钟和串口。

第二步,初始化 ADC。

第三步,在 while()循环中调用 Light_Value()函数获取光照电压,并使用 printf()将电压通过串口每隔 100ms 打印出来。

```
int main()
{
    CLOCK_Init();                                   //时钟初始化
    UART_Init();                                    //串口初始化
    ADC3_Init();                                    //ADC3 初始化

    while(1)
    {
        printf("Light: %d\r\n", Light_Value()); //获取光照电压并通过串口打印
        HAL_Delay(100);                             //延时 100ms
    }
}
```

5.1.3 运行结果

将程序下载到开发板中,打开串口,可以看到串口调试助手打印出的电压值,光照越强,该值越小;光照越弱,该值越大,如图 5-7 所示。

图 5-7　运行结果

练习

(1) 什么是 ADC?

(2) ADC 具有哪些模式?

(3) 配置其他 ADC 实现 ADC 数值转换。


视频 19

5.2　ADC：单 ADC 扫描转换

学习目标

掌握 ARM Cortex-M 系列芯片外设 ADC 多通道扫描转换的工作原理，通过配置 STM32F407 的 ADC 多通道来分别测量 ADC 内部参考电压、引脚电压以及内部温度传感器电压并计算温度值。

5.2.1　开发原理

STM32F407 的 ADC 内部框图原理请参考 5.1 节，此处不再赘述。

对于 STM32F407，温度传感器在内部和 ADC1_IN16 连接，内部测量电压引脚与 ADC1_IN17 连接，以此来分别转换传感器和内部电压为数字值。在不使用的情况下，温度传感器将处于断电模式。

温度传感器和参考电压通道框图如图 5-8 所示。

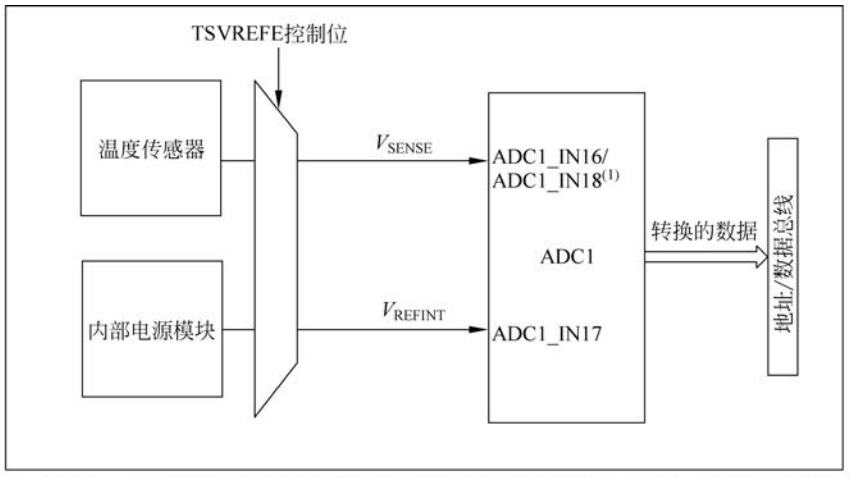
利用以下公式得出温度：

$$\text{温度}(℃) = \frac{V_{\text{SENSE}} - V_{25}}{\text{Avg\_Slope}} + 25$$

其中，

- $V_{25} = V_{\text{SENSE}}$ 在 25℃时的数值。
- Avg_Slope= 温度与 V_{SENSE} 曲线的平均斜率(单位为 mV/℃或 μV/℃)。

STM32F4 数据手册中的温度传感器参数如表 5-1 所示。



注：(1) V_{SENSE}是至ADC1_IN16的输入(对于STM32F40x和STM32F41x器件)，也是ADC1_IN18的输入(对于STM32F42x和STM32F43x器件)。

图 5-8 温度传感器和参考电压通道框图

表 5-1 温度传感器特点

符 号	参 数	最 小 值	典 型 值	最 大 值	单 位
T_L(1)	V_{SENSE} 与温度的线性关系	—	±1	±2	℃
Avg_Slope(1)	平均斜率	—	2.5		mV/℃
V_{25}(1)	25℃的电压	—	0.76		V
t_{START}(2)	启动时间	—	6	10	μs
T_{S_temp}(2)	ADC 读取温度时的采样时间	10	—	—	μs

注：(1) 由特性保证。

(2) 由设计保证。

该温度传感器的输出电压随温度线性变化。这个偏移线性函数取决于每个芯片上工艺变化。内部温度传感器更适合于检测温度的应用变化而不是绝对温度。如果必须有准确的温度读数，那么应该使用外部温度传感器。

内部参考电压校准参数如表 5-2 所示。

表 5-2 内部参考电压校准参数

符 号	参 数	内 存 地 址
V_{REFIN_CAL}	原始数据在 30℃时获得，V_{DDA}=3.3V	0x1FFF 7A2A～0x1FFF 7A2B

地址 0x1FFF 7A2A～0x1FFF 7A2B 中的数据为器件电压 VDDA 为 3.3V，温度为 30℃时的电压校准值。

内部参考电压计算公式如下：

$$VREF = \frac{3.3 \times VREFINT\_CAL}{VREFINT\_DATA}$$

其中，3.3 为 VDDA 的电压，VREFINT_DATA 为 ADC 测出的电压值。

通道电压计算公式如下：

$$Vchannelx = \frac{VREF}{FULL\_SCALE \times ADC\_DATAx}$$

其中，VREF 为 ADC 参考电压，ADC_DATAx 为 ADC 测出的通道电压值，FULL_SCALE 为 ADC 输出的最大数字值。

5.2.2 开发步骤

(1) 本节选取 ADC1_IN4 为电压测量通道。

首先定义 ADC 初始化句柄,创建 ADC 初始化函数 ADC_Init()。函数中调用 HAL_ADC_Init()初始化 ADC,此处初始化 ADC 为扫描模式,并设置规则序列为 3 个通道,使用软件触发。然后调用 HAL_ADC_ConfigChannel()函数分别配置 ADC 的通道 4、通道 17、通道 16,并依次配置采样顺序为 1、2、3。

```
ADC_HandleTypeDef ADC1_Handler; //ADC 句柄

//ADC 初始化函数
void ADC_Init(void)
{
ADC1_Handler.Instance = ADC1;
ADC1_Handler.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4; //4 分频,ADCCLK = PCLK2/4 = 84/
4 = 21MHz
ADC1_Handler.Init.Resolution = ADC_RESOLUTION_12B;                    //12 位模式
ADC1_Handler.Init.DataAlign = ADC_DATAALIGN_RIGHT;                    //右对齐
ADC1_Handler.Init.ScanConvMode = ENABLE;                              //扫描模式
ADC1_Handler.Init.EOCSelection = ADC_EOC_SINGLE_CONV;                 //开启 EOC 转换一次中断
ADC1_Handler.Init.ContinuousConvMode = DISABLE;                       //不开启连续转换
ADC1_Handler.Init.NbrOfConversion = 3;                                //3 个转换在规则序列
ADC1_Handler.Init.ExternalTrigConv = ADC_SOFTWARE_START;              //软件触发
ADC1_Handler.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;     //使用软件触发
ADC1_Handler.Init.DMAContinuousRequests = DISABLE;                    //关闭 DMA 请求
HAL_ADC_Init(&ADC1_Handler);                                          //初始化

ADC_ChannelConfTypeDef ADC_ChanneConf;

//电压采集通道初始化
ADC_ChanneConf.Channel = ADC_CHANNEL_4;                               //通道 4
ADC_ChanneConf.Rank = 1;                                              //第一个采样
ADC_ChanneConf.SamplingTime = ADC_SAMPLETIME_480CYCLES;               //周期采样时间
HAL_ADC_ConfigChannel(&ADC1_Handler, &ADC_ChanneConf);

//参考电压采集通道初始化
ADC_ChanneConf.Channel = ADC_CHANNEL_VREFINT;                         //参考电压通道
ADC_ChanneConf.Rank = 2;                                              //第二个采样
ADC_ChanneConf.SamplingTime = ADC_SAMPLETIME_480CYCLES;               //周期采样时间
HAL_ADC_ConfigChannel(&ADC1_Handler, &ADC_ChanneConf);
//温度传感器采集通道初始化
ADC_ChanneConf.Channel = ADC_CHANNEL_16;                              //读取温度通道
ADC_ChanneConf.Rank = 3;                                              //第三个采样
ADC_ChanneConf.SamplingTime = ADC_SAMPLETIME_480CYCLES;               //周期采样时间
HAL_ADC_ConfigChannel(&ADC1_Handler, &ADC_ChanneConf);
}
```

(2) 由 STM32F4 数据手册可知,ADC1_IN4 通道对应引脚 PA4(通道 16 和 17 为内部通道,不需要外部初始化引脚),如图 5-9 所示。

20	J9	29	40	N4	50	PA4	I/O	TTa	—	SPI1_NSS/SPI3_NSS/ USART2_CK/ DCMI_HSYNC/ OTG_HS_SOF/I2S3_WS/ EVENTOUT	ADC12_IN4/ DAC_OUT1

图 5-9 引脚映射

(3) 重定义 HAL_ADC_MspInit()函数,初始化 PA4 引脚并使能 ADC1。

```
//定义 ADC 底层驱动
void HAL_ADC_MspInit(ADC_HandleTypeDef * hadc)
{
    GPIO_InitTypeDef GPIO_Initure;

    __HAL_RCC_ADC1_CLK_ENABLE();                //使能 ADC1 时钟
    __HAL_RCC_GPIOA_CLK_ENABLE();               //开启 GPIOA 时钟

    GPIO_Initure.Pin = GPIO_PIN_4;              //PA4
    GPIO_Initure.Mode = GPIO_MODE_ANALOG;       //模拟
    GPIO_Initure.Pull = GPIO_NOPULL;            //浮空
    HAL_GPIO_Init(GPIOA, &GPIO_Initure);
}
```

(4) 定义结构体 ADC_DATA,分别存储通道电压、参考电压和温度的值。

```
typedef struct
{
    float Vchannel4;
    float VREF;
    float temperature;
}ADC_DATA;
```

(5) 创建 ADC_GetData()函数,用来处理电压数据。函数中首先调用 HAL_ADC_Start(),然后调用函数 HAL_ADC_GetValue()循环获取 3 个通道电压,最后根据公式计算出参考电压、通道电压和温度的值。

```
ADC_DATA adc_data;
void ADC_GetData(void)
{
    uint16_t adc_value[3];                                          //存储 ADC 测量的电压值

    HAL_ADC_Start(&ADC1_Handler);                                   //开始 ADC

    for(uint8_t i = 0;i < 3;i++)
    {
        while(!__HAL_ADC_GET_FLAG(&ADC1_Handler, ADC_FLAG_EOC));    //等待转换完成标志位
        adc_value[i] = (uint16_t)HAL_ADC_GetValue(&ADC1_Handler);   //获取 ADC 电压值
    }

    adc_data.VREF = 3.3 * VREFINT_CAL /adc_value[1];                //计算参考电压
    adc_data.Vchannel4 = adc_data.VREF /4095 * adc_value[0];        //计算通道电压
    adc_data.temperature = (((adc_data.VREF /4095 * adc_value[2]) - 0.76f) / 2.5f) + 25;
                                                                    //计算内部温度
}
```

(6) 主函数 main()程序如下:

第一步,初始化系统时钟、串口和 ADC。

第二步,调用 ADC_GetData()函数获取 ADC 电压并计算。

第三步,在 while 循环中分别将参考电压、通道电压和温度的值通过串口打印出来。

```
extern ADC_DATA adc_data;

int main()
{
```

```
    CLOCK_Init();                //时钟初始化
    UART_Init();                 //串口初始化
    ADC_Init();                  //ADC 初始化

    while(1)
    {
        ADC_GetData();           //ADC 获取数据
        printf("Vref: %.2f Vch4: %.2f Temp: %.2f\r\n", adc_data.VREF, adc_data.Vchannel4,
adc_data.temperature);
        HAL_Delay(100);
    }
}
```

5.2.3 运行结果

将程序下载到开发板中,将 USART1 引脚通过 USB 转串口模块连接到计算机上,打开串口助手,配置好波特率、数据位、校验位、停止位等,单击打开串口。可以看到串口分别打印出参考电压、通道电压和温度传感器的值,可以取一根杜邦线将 PA4 引脚分别连接到 3.3V 电源和 GND 引脚,再观察一下数值,如图 5-10 所示。

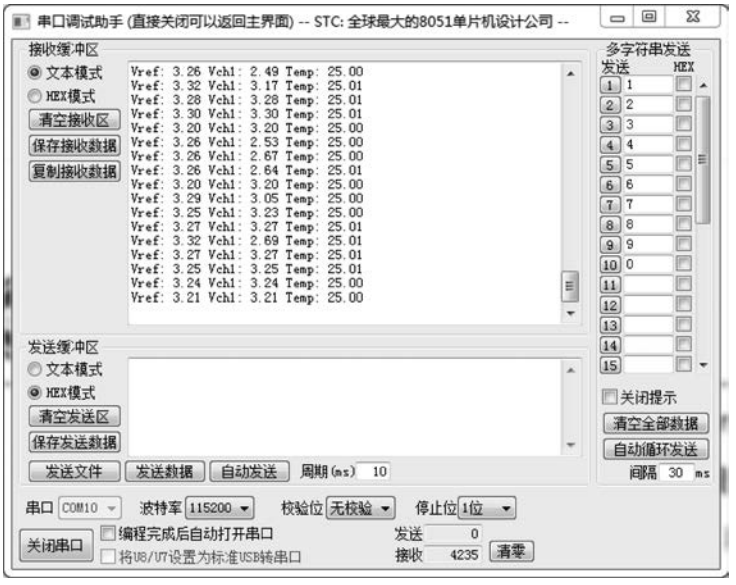


图 5-10 运行结果

练习

(1) 简述单 ADC 实现过程。

(2) 在计算芯片内部温度的公式中各参数代表什么?

(3) 配置其他 ADC 实现 ADC 数据转换。


视频 20

5.3 ADC:ADC 的 DMA 模式

学习目标

掌握 ARM Cortex-M 系列芯片外设 ADC 的 DMA 工作原理,通过配置 STM32F407 的 ADC 和 DMA 来分别测量 ADC 内部参考电压、引脚电压以及内部温度传感器电压并计算温度值。

5.3.1 开发原理

本节同5.2节类似，同样是测量ADC内部参考电压、引脚电压以及内部温度传感器电压值，区别是本节使用DMA传输的方式将ADC测得的3个通道电压直接传输到接收数据缓冲区中，从而直接计算数据缓冲区的数值即可，无须再调用函数读取ADC数据寄存器。

5.3.2 开发步骤

(1) 首先创建ADC初始化结构体句柄ADC1_Handler，然后创建ADC数据接收缓冲区数组adc_value[3]。

(2) 查找DMA通道映射表，可知ADC1需使用DMA2，数据流0、通道0(见STM32F407参考手册)，如图5-11所示。

外设请求	数据流0	数据流1	数据流2	数据流3	数据流4	数据流5	数据流6	数据流7
通道0	ADC1		TIM8_CH1 TIM8_CH2 TIM8_CH3		ADC1		TIM1_CH1 TIM1_CH2 TIM1_CH3	
通道1		DCMI	ADC2	ADC2		SPI6_TX(1)	SPI6_RX(1)	DCMI
通道2	ADC3	ADC3		SPI5_RX(1)	SPI5_TX(1)	CRYP_OUT	CRYP_IN	HASH_IN
通道3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
通道4	SPI4_RX(1)	SPI4_TX(1)	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
通道5		USART6_RX	USART6_RX	SPI4_RX(1)	SPI4_TX(1)		USART6_TX	USART6_TX
通道6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
通道7		TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX(1)	SPI5_TX(1)	TIM8_CH4 TIM8_TRIG TIM8_COM

注：(1) 这些请求在STM32F42xxx和STM32F43xxx上可用。

图5-11 DMA2映射表

(3) 创建ADC_DMA_Init()函数，分别初始化DMA和ADC。

第一步，调用HAL_DMA_Init()函数初始化DMA。

第二步，调用HAL_ADC_Init()函数初始化ADC，此处注意DMA的初始化句柄需要传入到ADC初始化结构体句柄，并使能DMA。

第三步，调用HAL_ADC_ConfigChannel()初始化ADC的3个通道。

第四步，调用HAL_ADC_Start_DMA()函数，传入数据缓冲区adc_value地址，开始DMA、ADC传输。

```
ADC_HandleTypeDef ADC1_Handler;                                    //ADC句柄

uint16_t adc_value[3];                                             //存储ADC测量的电压值
//ADC初始化函数
void ADC_DMA_Init(void)
{
    DMA_HandleTypeDef hdma_adc;

    __HAL_RCC_DMA2_CLK_ENABLE();

    hdma_adc.Instance = DMA2_Stream0;                              //数据流
```

```
    hdma_adc.Init.Channel = DMA_CHANNEL_0;                          //通道 0
    hdma_adc.Init.Direction = DMA_PERIPH_TO_MEMORY;                 //外设到内存
    hdma_adc.Init.PeriphInc = DMA_PINC_DISABLE;                     //外设地址不自增
    hdma_adc.Init.MemInc = DMA_MINC_ENABLE;                         //内存地址自增
    hdma_adc.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;    //一次传输半字
    hdma_adc.Init.MemDataAlignment = DMA_PDATAALIGN_HALFWORD;       //一次传输半字
    hdma_adc.Init.Mode = DMA_CIRCULAR;                              //循环转换模式
    hdma_adc.Init.Priority = DMA_PRIORITY_HIGH;                     //优先级高
    hdma_adc.Init.MemBurst = DMA_MBURST_SINGLE;                     //突发
    hdma_adc.Init.PeriphBurst = DMA_PBURST_SINGLE;

    HAL_DMA_Init(&hdma_adc);

    ADC1_Handler.Instance = ADC1;
    ADC1_Handler.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;  //4 分频,ADCCLK = PCLK2/4 =
                                                                  //84/4 = 21MHz
    ADC1_Handler.Init.Resolution = ADC_RESOLUTION_12B;            //12 位模式
    ADC1_Handler.Init.DataAlign = ADC_DATAALIGN_RIGHT;            //右对齐
    ADC1_Handler.Init.ScanConvMode = ENABLE;                      //扫描模式
    ADC1_Handler.Init.EOCSelection = ADC_EOC_SINGLE_CONV;         //开启 EOC 转换一次中断
    ADC1_Handler.Init.NbrOfConversion = 3;                        //3 个转换在规则序列
    ADC1_Handler.Init.ExternalTrigConv = ADC_SOFTWARE_START;      //软件触发
    ADC1_Handler.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE; //使用软件触发
    ADC1_Handler.Init.ContinuousConvMode = ENABLE;                //开启连续转换
    ADC1_Handler.Init.DMAContinuousRequests = ENABLE;             //开启 DMA 请求
    ADC1_Handler.DMA_Handle = &hdma_adc;                          //传输 DMA 句柄
    HAL_ADC_Init(&ADC1_Handler);                                  //初始化

    ADC_ChannelConfTypeDef ADC_ChanneConf;

    //电压采集通道初始化
    ADC_ChanneConf.Channel = ADC_CHANNEL_4;                       //通道 4
    ADC_ChanneConf.Rank = 1;                                      //第一个采样
    ADC_ChanneConf.SamplingTime = ADC_SAMPLETIME_480CYCLES;       //周期采样时间
    HAL_ADC_ConfigChannel(&ADC1_Handler, &ADC_ChanneConf);

    //参考电压采集通道初始化
    ADC_ChanneConf.Channel = ADC_CHANNEL_VREFINT;                 //参考电压通道
    ADC_ChanneConf.Rank = 2;                                      //第二个采样
    ADC_ChanneConf.SamplingTime = ADC_SAMPLETIME_480CYCLES;       //周期采样时间
    HAL_ADC_ConfigChannel(&ADC1_Handler, &ADC_ChanneConf);

    //温度传感器采集通道初始化
    ADC_ChanneConf.Channel = ADC_CHANNEL_16;                      //读取温度通道
    ADC_ChanneConf.Rank = 3;                                      //第三个采样
    ADC_ChanneConf.SamplingTime = ADC_SAMPLETIME_480CYCLES;       //周期采样时间
    HAL_ADC_ConfigChannel(&ADC1_Handler, &ADC_ChanneConf);

    HAL_ADC_Start_DMA(&ADC1_Handler, (uint32_t*)adc_value, 3);    //开启 DMA ADC 传输
}
```

(4) 重定义 HAL_ADC_MspInit()函数,初始化 PA4 引脚并使能 ADC1。

```
//定义 ADC 底层驱动
void HAL_ADC_MspInit(ADC_HandleTypeDef* hadc)
{
    GPIO_InitTypeDef GPIO_Initure;
```

```
    __HAL_RCC_ADC1_CLK_ENABLE();              //使能 ADC1 时钟
    __HAL_RCC_GPIOA_CLK_ENABLE();             //开启 GPIOA 时钟

    GPIO_Initure.Pin = GPIO_PIN_4;            //PA4
    GPIO_Initure.Mode = GPIO_MODE_ANALOG;     //模拟
    GPIO_Initure.Pull = GPIO_NOPULL;          //浮空
    HAL_GPIO_Init(GPIOA, &GPIO_Initure);
}
```

(5) 定义结构体 ADC_DATA,分别存储通道电压、参考电压和温度的值。

```
typedef struct
{
    float Vchannel4;
    float VREF;
    float temperature;
}ADC_DATA;
```

(6) 创建 DMA_GetData()函数,根据公式和数据缓冲区中的电压值分别计算出参考电压、通道电压和温度的值。

```
ADC_DATA adc_data;
void DMA_GetData(void)
{
    adc_data.VREF = 3.3 * VREFINT_CAL /adc_value[1];          //计算参考电压
    adc_data.Vchannel4 = adc_data.VREF /4095 * adc_value[0]; //计算通道电压
    adc_data.temperature = (((adc_data.VREF /4095 * adc_value[2]) - 0.76f) / 2.5f) + 25;
                                                              //计算内部温度
}
```

(7) 主函数 main()程序如下:

第一步,初始化系统时钟、串口和 ADC-DMA。

第二步,调用 DMA_GetData ()函数分别计算参考电压、通道电压和温度的值。

第三步,在 while 循环中将 3 个值分别通过串口打印出来。

```
extern ADC_DATA adc_data;

int main()
{
    CLOCK_Init();                                             //时钟初始化
    UART_Init();                                              //串口初始化
    ADC_DMA_Init();                                           //ADC-DMA 初始化

    while(1)
    {
        DMA_GetData();                                        //ADC - DMA 获取数据
        printf("Vref: %.2f Vch4: %.2f Temp: %.2f\r\n", adc_data.VREF, adc_data.Vchannel4,
        adc_data.temperature);
        HAL_Delay(100);
    }
}
```

5.3.3 运行结果

将程序下载到开发板中,找到 USART1 引脚通过 USB 转串口模块连接到计算机上,打开串口助手,配置好波特率、数据位、校验位、停止位等,单击打开串口。可以看到串口分别打印出参考电压、通道电压和温度传感器的值,可以取一根杜邦线将 PA4 引脚连接到 3.3V 电源

或 GND 引脚,再观察一下数值,如图 5-12 所示。

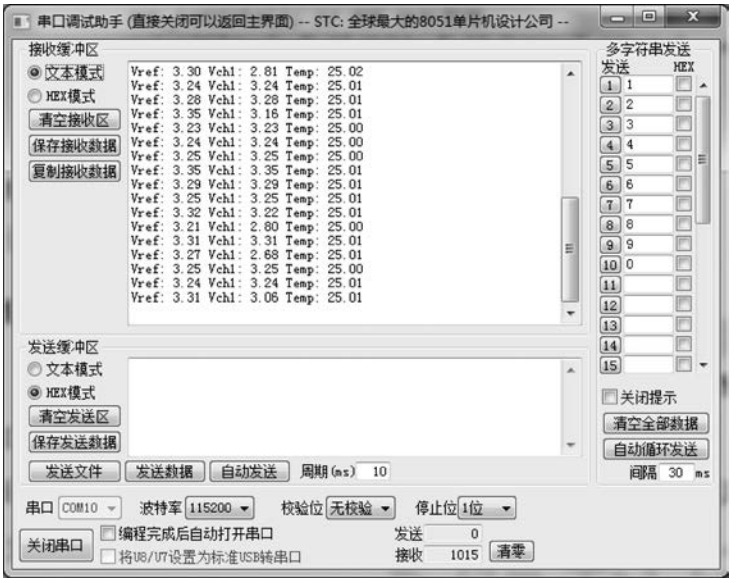


图 5-12　运行结果

练习

(1) STM32F4 系列的 DMA 有哪些传输模式?

(2) 配置其他 ADC 实现 ADC 数据转换。


视频 21

5.4　ADC:双重 ADC 交叉模式

学习目标

掌握 ARM Cortex-M 系列芯片外设多 ADC 工作模式的工作原理,通过配置 STM32F407 的双重 ADC 模式,使用 ADC1 和 ADC2 来同时测量引脚电压。

5.4.1　开发原理

AD 转换包括采样阶段和转换阶段,在采样阶段才对通道数据进行采集;而在转换阶段只是将采集到的数据转换为数字量输出,此刻通道数据变化不会改变转换结果。独立模式的 ADC 采集需要在一个通道采集并且转换完成后才会进行下一个通道的采集。双重或者三重 ADC 的机制使用两个或以上 ADC 同时采样两个或以上不同通道的数据或者交叉采集两个或以上 ADC 同一通道的数据。双重或者三重 ADC 模式较独立模式一个最大的优势就是转换速度快。

在有两个或多个 ADC 模块的产品中,可以使用双重 ADC 模式和三重 ADC 模式。在多 ADC 模式里,根据 ADC_CCR 寄存器中 MULTI[4:0]位所选的模式,转换的启动可以是按照“ADC1 主,ADC2 和 ADC3 从”的模式交替触发或同步触发。

在多重 ADC 模式下,当转换配置成由外部事件触发时,用户必须将其设置为仅触发主 ADC,从 ADC 设置为由软件触发,这样可以防止意外地触发从 ADC 转换。

ADC 有 4 种可能的模式:

- 同步注入模式。
- 同步规则模式。

- 交叉模式。
- 交替触发模式。

还有可以用下列组合方式：

- 同步注入模式＋同步规则模式。
- 同步规则模式＋交替触发模式。

在双重 ADC 模式下，ADC 公共的数据寄存器（ADC_CDR）包含 ADC1、ADC2 和 ADC3 的规则转换数据，32 位寄存器的所有位都将被使用。

在多 DMA 模式下，DMA 支持 3 种模式：

1）DMA 模式 1

每次 DMA 请求（只有一个数据项是有效的），半字代表 ADC 转换数据项被传输。

在双重 ADC 模式下，第一个请求时，ADC1 数据先被传输；第二个请求时，ADC2 数据被传输；以此类推。

在三重 ADC 模式下，第一个请求时，ADC1 数据被传输；第二个请求时，ADC2 数据被传输；第三个请求时，ADC3 被传输；以此类推。

DMA 模式 1 用于规则通道的三重 ADC 模式。

举一个例子：

规则通道的三重 ADC 模式：产生 3 个 DMA 请求（一个请求对应一次转换数据项）。

第 1 个请求 ADC_CDR[31:0] = ADC1_DR[15:0]

第 2 个请求 ADC_CDR[31:0] = ADC2_DR[15:0]

第 3 个请求 ADC_CDR[31:0] = ADC3_DR[15:0]

第 4 个请求 ……

2）DMA 模式 2

每次 DMA 请求（两个数据项有效）代表 2 个 ADC 转换数据项被传输，也就是一个字。

在双重 ADC 模式下，第一个请求时，ADC2 和 ADC1 数据同时被传输（ADC2 占高 16 位，ADC1 占低 16 位），后面以此类推。

在三重 ADC 模式下，第一个请求时 ADC2 和 ADC1 数据被传输，第二个请求时 ADC1 和 ADC3 数据被传输，第三个请求时 ADC3 和 ADC2 数据被传输。

DMA 模式 2 用于交叉模式和规则同步模式（仅双重 ADC 模式）。

举一个例子：

双重 ADC 交叉模式：每次 2 个数据项有效时将产生 DMA 请求。

第 1 个请求 ADC_CDR[31:0] = ADC2_DR[15:0] | ADC1_DR[15:0]

第 2 个请求 ADC_CDR[31:0] = ADC2_DR[15:0] | ADC1_DR[15:0]

……

三重 ADC 交叉模式：每次 2 个数据项有效时将产生 DMA 请求。

第 1 个请求 ADC_CDR[31:0] = ADC2_DR[15:0] | ADC1_DR[15:0]

第 2 个请求 ADC_CDR[31:0] = ADC1_DR[15:0] | ADC3_DR[15:0]

第 3 个请求 ADC_CDR[31:0] = ADC3_DR[15:0] | ADC2_DR[15:0]

第 4 个请求 ADC_CDR[31:0] = ADC2_DR[15:0] | ADC1_DR[15:0]

……

3）DMA 模式 3

这个模式和模式 2 类似，仅有的不同是每次 DMA 请求（两个数据项有效）两个字节代表

两个 ADC 转换的数据项,也就是一个半字,数据传输顺序和 DMA 模式 2 类似。

双重 ADC 交叉模式:每次两个数据项有效时将产生 DMA 请求。

第 1 个请求 ADC_CDR[31:0] = ADC2_DR[7:0] | ADC1_DR[7:0]

第 2 个请求 ADC_CDR[31:0] = ADC2_DR[7:0] | ADC1_DR[7:0]

……

三重 ADC 交叉模式:每次两个数据项有效时将产生 DMA 请求。

第 1 个请求 ADC_CDR[31:0] = ADC2_DR[7:0] | ADC1_DR[7:0]

第 2 个请求 ADC_CDR[31:0] = ADC1_DR[7:0] | ADC3_DR[7:0]

第 3 个请求 ADC_CDR[31:0] = ADC3_DR[7:0] | ADC2_DR[7:0]

第 4 个请求 ADC_CDR[31:0] = ADC2_DR[7:0] | ADC1_DR[7:0]

……

5.4.2 开发步骤

(1) 本节选取 ADC1_IN4 作为数据采集通道,首先分别定义 ADC1 和 ADC2 初始化结构体,以及接收数据存储变量。

```
ADC_HandleTypeDef AdcHandle1;          //ADC1 初始化句柄
ADC_HandleTypeDef AdcHandle2;          //ADC2 初始化句柄

uint16_t uhADCDualConvertedValue;      //获取转换值的变量
```

(2) 创建 MultiADC_Config()函数,用来初始化 ADC1 和 ADC2 以及相应配置。

第一步,调用 HAL_DMA_Init()函数,初始化 ADC 的 DMA 传输。

第二步,调用 HAL_ADC_Init()函数,初始化 ADC1 并初始化 ADC1 的通道 4。

第三步,调用 HAL_ADC_Init()函数,初始化 ADC2 并初始化 ADC2 的通道 4。

第四步,调用 HAL_ADCEx_MultiModeConfigChannel()函数,配置 ADC 为双重交叉模式,且配置 DMA 模式为模式 3,每次 DMA 请求传输两个字节(ADC1 和 ADC2 各一个字节)。最后开启 ADC 的传输。

```
//ADC1 和 ADC2 初始化函数
void MultiADC_Config(void)
{
    //DMA 初始化
    DMA_HandleTypeDef hdma_adc;

    __HAL_RCC_DMA2_CLK_ENABLE();

    hdma_adc.Instance = DMA2_Stream0;                                    //数据流

    hdma_adc.Init.Channel   = DMA_CHANNEL_0;                             //通道 0
    hdma_adc.Init.Direction = DMA_PERIPH_TO_MEMORY;                      //外设到内存
    hdma_adc.Init.PeriphInc = DMA_PINC_DISABLE;                          //外设地址不自增
    hdma_adc.Init.MemInc = DMA_MINC_ENABLE;                              //内存地址自增
    hdma_adc.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;         //一次传输半字
    hdma_adc.Init.MemDataAlignment = DMA_PDATAALIGN_HALFWORD;            //一次传输半字
    hdma_adc.Init.Mode = DMA_CIRCULAR;                                   //循环转换模式
    hdma_adc.Init.Priority = DMA_PRIORITY_HIGH;                          //优先级高
    hdma_adc.Init.MemBurst = DMA_MBURST_SINGLE;                          //突发
    hdma_adc.Init.PeriphBurst = DMA_PBURST_SINGLE;
    HAL_DMA_Init(&hdma_adc);
```

```
    //初始化 ADC1
    AdcHandle1.Instance = ADC1;
    AdcHandle1.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV4;    //4 分频
    AdcHandle1.Init.Resolution = ADC_RESOLUTION_8B;                   //8bit
    AdcHandle1.Init.ScanConvMode = DISABLE;                           //扫描模式不使能
    AdcHandle1.Init.ContinuousConvMode = ENABLE;                      //连续转换模式使能
    AdcHandle1.Init.ExternalTrigConv = ADC_SOFTWARE_START;            //软件触发
    AdcHandle1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;   //使用软件触发
    AdcHandle1.Init.DataAlign = ADC_DATAALIGN_RIGHT;                  //数据右对齐
    AdcHandle1.Init.NbrOfConversion = 1;
    AdcHandle1.Init.DMAContinuousRequests = ENABLE;                   //使用 DMA
    AdcHandle1.DMA_Handle = &hdma_adc;                                //传递 DMA 句柄
    AdcHandle1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;   //开启 EOC 转换一次完成中断
    HAL_ADC_Init(&AdcHandle1);                                        //初始化 ADC1

    //配置 ADC1 通道
    ADC_ChannelConfTypeDef ADC_ChanneConf;                            //ADC 通道配置
    ADC_ChanneConf.Channel = ADC_CHANNEL_4;                           //ADC1 通道
    ADC_ChanneConf.Rank = 1;                                          //转换等级
    ADC_ChanneConf.SamplingTime = ADC_SAMPLETIME_3CYCLES;             //采样周期
    HAL_ADC_ConfigChannel(&AdcHandle1, &ADC_ChanneConf);

    //初始化 ADC2
    AdcHandle2.Instance = ADC2;
    AdcHandle2.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV4;
    AdcHandle2.Init.Resolution = ADC_RESOLUTION_8B;
    AdcHandle2.Init.ScanConvMode = DISABLE;
    AdcHandle2.Init.ContinuousConvMode = ENABLE;
    AdcHandle2.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    AdcHandle2.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    AdcHandle2.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    AdcHandle2.Init.NbrOfConversion = 1;
    AdcHandle2.Init.DMAContinuousRequests = ENABLE;
    AdcHandle2.DMA_Handle = &hdma_adc;                                //传递 DMA 句柄
    AdcHandle2.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    HAL_ADC_Init(&AdcHandle2);                                        //初始化 ADC2

    //配置 ADC2 通道
    ADC_ChanneConf.Channel = ADC_CHANNEL_4;                           //ADC2 通道
    HAL_ADC_ConfigChannel(&AdcHandle2, &ADC_ChanneConf);

    //配置双重交叉模式
    ADC_MultiModeTypeDef   ADC_Multimode;
    ADC_Multimode.Mode = ADC_DUALMODE_INTERL;                         //双重 ADC 交叉模式
    ADC_Multimode.DMAAccessMode = ADC_DMAACCESSMODE_3; //每次 DMA 请求传输两个字节 ADC1/ADC2
                                                       //数据一起传输
    ADC_CDR[15:0] = ADC2_DR[7:0] | ADC1_DR[7:0]
    ADC_Multimode.TwoSamplingDelay = ADC_TWOSAMPLINGDELAY_6CYCLES;   //配置两个采样之间的延迟
    HAL_ADCEx_MultiModeConfigChannel(&AdcHandle1, &ADC_Multimode);

    HAL_ADC_Start(&AdcHandle2);                                       //开始 ADC2

    HAL_ADCEx_MultiModeStart_DMA(&AdcHandle1, (uint32_t * )&uhADCDualConvertedValue, 1);
}
```

(3) 由 STM32F4 数据手册可知,ADC1_IN4 和 ADC2_IN4 通道都在 PA4 引脚,如图 5-13 所示。

20	J9	29	40	N4	50	PA4	I/O	TTa	—	SPI1_NSS/SPI3_NSS/ USART2_CK/ DCMI_HSYNC/ OTG_HS_SOF/I2S3_WS/ EVENTOUT	ADC12_IN4/ DAC_OUT1

图 5-13 引脚映射

(4) 重定义 HAL_ADC_MspInit()函数,初始化 PA4 引脚,并使能 ADC1 和 ADC2 时钟。

```
//初始化 ADC 底层驱动引脚
void HAL_ADC_MspInit(ADC_HandleTypeDef * hadc)
{
    GPIO_InitTypeDef GPIO_Initure;

    __HAL_RCC_ADC1_CLK_ENABLE();                    //使能 ADC1 时钟
    __HAL_RCC_ADC2_CLK_ENABLE();                    //使能 ADC2 时钟
    __HAL_RCC_GPIOA_CLK_ENABLE();                   //开启 GPIOA 时钟

    GPIO_Initure.Pin = GPIO_PIN_4;                  //PA4
    GPIO_Initure.Mode = GPIO_MODE_ANALOG;           //模拟
    GPIO_Initure.Pull = GPIO_NOPULL;                //浮空
    HAL_GPIO_Init(GPIOA, &GPIO_Initure);
}
```

(5) 定义结构体 ADC_DATA,用来存储 ADC1 和 ADC2 测出的数字值和计算后的电压。

```
typedef struct
{
    uint8_t adc1_value;                             //存储 ADC1 测出的值
    uint8_t adc2_value;                             //存储 ADC1 测出的值
    float adc1_voltage;                             //存储计算得到的 ADC1 电压
    float adc2_voltage;                             //存储计算得到的 ADC2 电压
}ADC_DATA;
```

(6) 创建 MultiADC_GetData()函数,用来处理 ADC 测得的数据。函数中首先获取数据变量 uhADCDualConvertedValue 的高 8 位数据和低 8 位数据,其中低 8 位为 ADC1 测出的数据,高 8 位为 ADC2 测出的数据。获取数据后分别计算 ADC1 和 ADC2 的电压。

```
//数据处理函数
void MultiADC_GetData(void)
{
    //获取 ADC1/ADC2 测出数字值
    adc_data.adc1_value = (uhADCDualConvertedValue & 0x00FF);
    adc_data.adc2_value = (uhADCDualConvertedValue >> 8);

    //计算 ADC1/ADC2 得到的电压
    adc_data.adc1_voltage = adc_data.adc1_value * 3.3 / 255;
    adc_data.adc2_voltage = adc_data.adc2_value * 3.3 / 255;
}
```

(7) 主函数 main()程序如下:

第一步,初始化系统时钟和串口。

第二步,调用 MultiADC_Config()函数,初始化双重 ADC 及交叉模式。

第三步，调用 MultiADC_GetData()函数，分别获取两个 ADC 的数据。

第四步，在 while()循环中每隔 100ms 将 ADC1 和 ADC2 通过串口打印出来。

```
extern ADC_DATA adc_data;

int main()
{
    CLOCK_Init();                   //时钟初始化
    UART_Init();                    //串口初始化

    MultiADC_Config();              //双重 ADC 交叉模式初始化

    while(1)
    {
        MultiADC_GetData();         //双重 ADC 交叉模式获取数据
        printf("ADC1 voltage: %.2f ADC2 voltage: %.2f\r\n", adc_data.adc1_voltage, adc_data.
adc2_voltage);
        HAL_Delay(100);
    }
}
```

5.4.3 运行结果

将程序下载到开发板中，将 USART1 引脚通过 USB 转串口模块连接到计算机上，打开串口助手，配置好波特率、数据位、校验位、停止位等，单击打开串口。可以看到串口中打印出 ADC1 和 ADC2 分别测出的电压值，如图 5-14 所示。

图 5-14 运行结果

练习

(1) 简述 AD 转换包括的采样阶段以及转换阶段。

(2) 简述在多重 DMA 模式下，DMA 支持的 3 种模式。

(3) 配置其他 ADC 实现 AD 转换。


视频 22

5.5 ADC：定时器触发模式

学习目标

掌握 ARM Cortex-M 系列芯片外设 ADC 定时器触发的工作原理，通过配置 STM32F407 的定时器和 ADC，使用定时器间隔触发 ADC 来测量电压值。

5.5.1 开发原理

本节使用定时器的方式来触发 ADC 测量电压值。首先选取定时器 1 为触发定时器，并配置定时器为 PWM 输出的方式，配置定时器的分频系数为 16800，因定时器 1 的时钟为 168M，所以定时器计数频率为 168MHz/16800 = 10kHz；设置定时器预装载值为 10000，所以定时器的溢出时间为 1s，设置 PWM 的比较值为 5000，且 ADC 触发方式为上升沿触发，所以 ADC 将每隔 1s 进行一次测量。

5.5.2 开发步骤

(1) 本节同样选取 ADC1 的通道 4 为测量电压通道。

(2) 创建 TIM1_Config()函数，用来初始化定时器及 PWM 输出。

```
//定时器初始化函数
static void TIM1_Config(void)
{
    TIM_HandleTypeDef TIM_Handle;                                    //定时器初始化结构体变量
    TIM_OC_InitTypeDef TIM_OC_Handle;                                //定时器输出初始化结构体变量

    __TIM1_CLK_ENABLE();                                             //使能定时器 1 时钟

    //定时器初始化
    TIM_Handle.Channel = HAL_TIM_ACTIVE_CHANNEL_1;                   //通道 1
    TIM_Handle.Instance = TIM1;                                      //选择定时器 1
    TIM_Handle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;          //时钟 1 分频
    TIM_Handle.Init.CounterMode = TIM_COUNTERMODE_UP;                //向上计数模式
    TIM_Handle.Init.Period = 10000 - 1;                              //自动重装载值
    TIM_Handle.Init.Prescaler = 16800 - 1;                           //预分频系数
    HAL_TIM_PWM_Init(&TIM_Handle);                                   //初始化定时器

    //定时器输出 PWM 初始化
    TIM_OC_Handle.OCMode = TIM_OCMODE_PWM1;                          //模式选择 PWM1
    TIM_OC_Handle.OCPolarity = TIM_OCPOLARITY_LOW;                   //输出比较极性为低
    TIM_OC_Handle.Pulse = 5000;   //设置比较值，此值用来确定占空比，默认比较值为自动重装载值
                                  //的一半，即占空比为 50%
    HAL_TIM_PWM_ConfigChannel(&TIM_Handle, &TIM_OC_Handle, TIM_CHANNEL_1);   //配置 PWM 输出

    HAL_TIM_PWM_Start(&TIM_Handle, TIM_CHANNEL_1);                   //开始 PWM 输出
}
```

(3) 创建 TIM_ADC_Init ()函数，用来初始化 ADC。

第一步，调用 HAL_ADC_Init()函数，初始化 ADC1 和 ADC1 的通道 4，此时设置 ADC 为定时器 1 通道 1 的上升沿触发。

第二步，调用 TIM1_Config()函数，初始化触发定时器，并开启 ADC 中断。

```
uint16_t adc_value; //存储变量
ADC_HandleTypeDef ADC1_Handler;                                    //ADC 句柄

//ADC 初始化函数
void TIM_ADC_Init(void)
{
    //初始化 ADC
    ADC1_Handler.Instance = ADC1;
    ADC1_Handler.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4; //4 分频,ADCCLK = PCLK2/4 =
                                                                 //84/4 = 21MHz
    ADC1_Handler.Init.Resolution = ADC_RESOLUTION_12B;             //12 位模式
    ADC1_Handler.Init.DataAlign = ADC_DATAALIGN_RIGHT;             //右对齐
    ADC1_Handler.Init.ScanConvMode = DISABLE;                      //不扫描模式
    ADC1_Handler.Init.EOCSelection = ADC_EOC_SINGLE_CONV;          //开启 EOC 转换一次中断
    ADC1_Handler.Init.ContinuousConvMode = DISABLE;                //不开启连续转换
    ADC1_Handler.Init.NbrOfConversion = 1;                         //1 个转换在规则序列
    ADC1_Handler.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T1_CC1; //使用定时器 1 通道 1 触发
    ADC1_Handler.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;   //使用上升沿触发
    ADC1_Handler.Init.DMAContinuousRequests = DISABLE;             //不开启 DMA 请求
    HAL_ADC_Init(&ADC1_Handler);                                   //初始化

    ADC_ChannelConfTypeDef ADC_ChanneConf;

    //电压采集通道初始化
    ADC_ChanneConf.Channel = ADC_CHANNEL_4;                        //通道 4
    ADC_ChanneConf.Rank = 1;                                       //第一个采样
    ADC_ChanneConf.SamplingTime = ADC_SAMPLETIME_480CYCLES;        //周期采样时间
    HAL_ADC_ConfigChannel(&ADC1_Handler, &ADC_ChanneConf);

    TIM1_Config();                                                 //初始化定时器

    HAL_ADC_Start_IT(&ADC1_Handler);                               //开启 DMA ADC 传输
}
```

(4) 由 STM32F4 数据手册可知，ADC1_IN4 通道对应 PA4 引脚，如图 5-15 所示。

20	J9	29	40	N4	50	PA4	I/O	TTa	—	SPI1_NSS/SPI3_NSS/ USART2_CK/ DCMI_HSYNC/ OTG_HS_SOF/I2S3_WS/ EVENTOUT	ADC12_IN4 /DAC_OUT1

图 5-15 引脚映射

(5) 重定义 HAL_ADC_MspInit()函数，初始化 PA4 引脚，并使能 ADC1 时钟，同时设置 ADC 的中断优先级并使能中断。

```
//初始化 ADC 底层驱动引脚
void HAL_ADC_MspInit(ADC_HandleTypeDef * hadc)
{
    GPIO_InitTypeDef GPIO_Initure;

    __HAL_RCC_ADC1_CLK_ENABLE();                //使能 ADC1 时钟
```

```
    __HAL_RCC_GPIOA_CLK_ENABLE();                    //开启 GPIOA 时钟

    GPIO_Initure.Pin = GPIO_PIN_4;                   //PA4
    GPIO_Initure.Mode = GPIO_MODE_ANALOG;            //模拟
    GPIO_Initure.Pull = GPIO_NOPULL;                 //浮空
    HAL_GPIO_Init(GPIOA, &GPIO_Initure);

    HAL_NVIC_SetPriority(ADC_IRQn, 0, 0);            //使能 ADC 中断优先级
    HAL_NVIC_EnableIRQ(ADC_IRQn);                    //使能 ADC 中断
}
```

(6) 定义 ADC 的中断服务函数和数据转换完成回调函数,在回调函数中通过函数 HAL_ADC_GetValue()获取 ADC 的测量数字值,然后计算得出电压值,最后通过串口打印出电压值。

```
//ADC 中断服务函数
void ADC_IRQHandler(void)
{
    HAL_ADC_IRQHandler(&ADC1_Handler);
}

//ADC 转换完成回调函数
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef * hadc)
{
    float adc_value1;
    adc_value = HAL_ADC_GetValue(&ADC1_Handler); //获取 ADC 测量值
    adc_value1 = adc_value * 3.3 /4095;          //计算 ADC 测量电压

    printf("%.2f\r\n", adc_value1);                  //定时器触发 ADC 获取数据并通过串口打印
}
```

(7) 主函数 main()程序如下:

第一步,初始化系统时钟和串口。

第二步,初始化定时器和 ADC。

```
int main()
{
    CLOCK_Init();                                    //时钟初始化
    UART_Init();                                     //串口初始化

    TIM_ADC_Init();                                  //定时器触发 ADC 初始化

    while(1)
    {
    }
}
```

5.5.3 运行结果

将程序下载到开发板中,将 USART1 引脚通过 USB 转串口模块连接到计算机上,打开串口助手,配置好波特率、数据位、校验位、停止位等,单击打开串口。可以看到串口中每隔 1s 打印出 ADC1 测出的电压值,如图 5-16 所示。

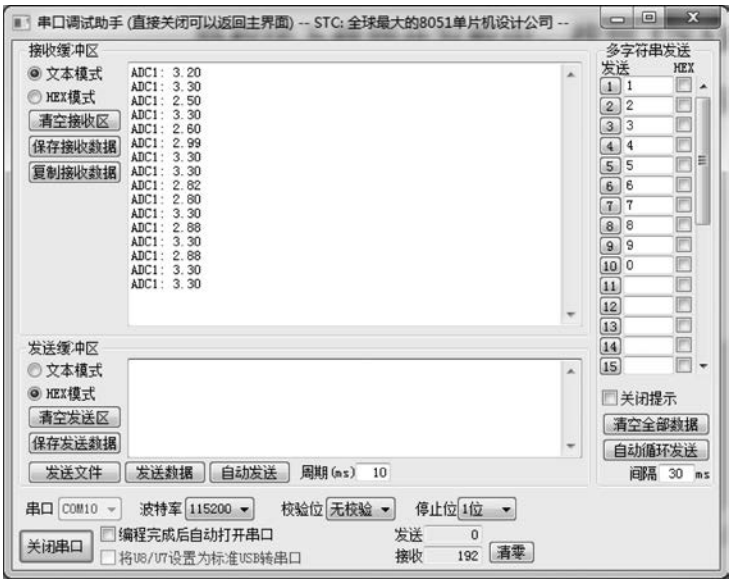


图 5-16 运行结果

练习

(1) 简述 ADC-定时器触发的过程。

(2) 配置其他 ADC 实现 ADC 数据转换。