CW32 电压电流表软件部分教程

一、概述

CW32 电压电流表软件编程开发。

二、实验一：LED 灯原理与驱动

2.1.LED 灯基础知识

2.1.1.LED 灯结构组成

LED 灯，也称发光二极管，是一种能够将电能转化为可见光的固态的半导体器件，它可以直接把电转化为光。LED 的内部是一个半导体的晶片，晶片的一端附在一个支架上，一端是负极，另一端连接电源的正极，整个晶片环-氧树脂封装起来。常见的 LED 灯如图 2-1 所示。

图 2-1 常见 LED 灯

2.1.2.LED 灯发光原理

半导体晶片由两部分组成，一部分是 P 型半导体，另一端是 N 型半导体。这两种半导体连接起来的时候，它们之间就形成了一个 P-N 结。当电流通过导线作用于这个晶片的时候，电子就会被推向 P 区，在 P 区里电子跟空穴复合，然后就会以光子的形式发出能量，这就是 LED 灯发光的原理。

2.1.3.LED 灯驱动原理

LED 驱动指的是通过稳定的电源为 LED 提供合适的电流和电压，使其正常工作点亮。LED 驱动方式主要有恒流和恒压两种。限定电流的恒流驱动是最常见的方式，因为 LED 灯对电流敏感，电流大于其额定值可能导致损坏。恒流驱动保证了稳定的电流，从而确保了 LED 安全。

LED 灯的驱动比较简单，只需要给将对应的正负极接到单片机的正负极即可驱动。

LED 的接法也分有两种，灌入电流和输出电流。

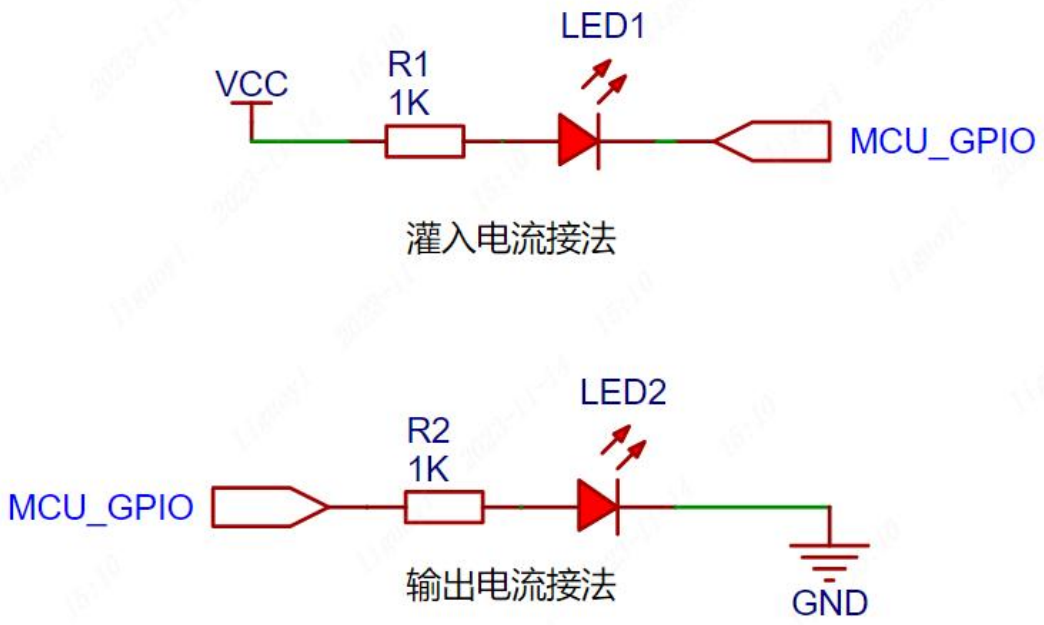

图 2-2 LED 接法示例

- 灌入电流指的是 LED 的供电电流是由外部提供电流，将电流灌入我们的 MCU；风险是当外部电源出现变化时，会导致 MCU 的引脚烧坏。
- 输出电流指的是由 MCU 提供电压电流，将电流输出给 LED；如果使用 MCU 的 GPIO 直接驱动 LED，则驱动能力较弱，可能无法提供足够的电流驱动 LED。

需要注意的 是 LED 灯的颜色不同，对应的电压也不同。**电流不可过大，通常需要接入 220 欧姆到 10K 欧姆左右的限流电阻**，限流电阻的阻值越大，LED 的亮度越暗。

2.2.LED 灯原理图

CW32F003 小蓝板关于 LED 灯的原理图如图 2-3 所示

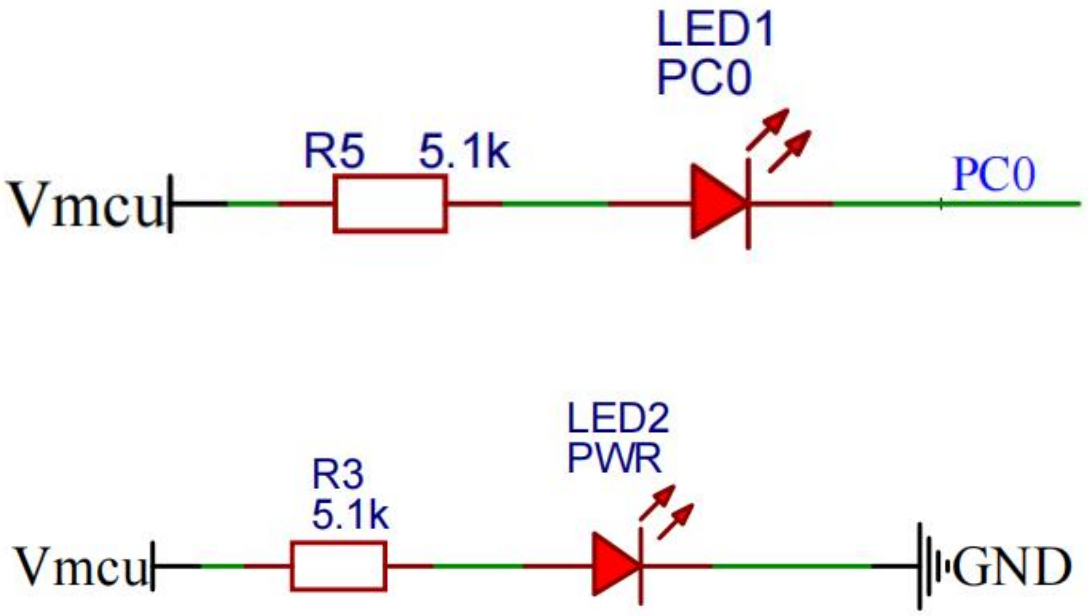

图 2-3 小蓝板 LED 灯原理图

2.3.LED 灯驱动流程（库函数）

通过上面的原理图可以了解到，LED2 中负极接到了电源地，LED2 的正极经限流电阻 R3 连接到电源正。当核心板上电时，使 LED2 导通，这时便有电流流过发光二极管 LED2，使 LED2 发光指示电源接入；

LED1 的正极经限流电阻 R5 接到电源正极，LED1 的负极连接到单片机的 GPIO 口上，通过 LED 灯的驱动原理，只需要将相应 GPIO（PC0）配置为低电平即可点亮 LED1。CW32F003 小蓝板的 LED 部分实物图如图 2-4 所示

图 2-4 小蓝板 LED 实物图

2.3.1.配置流程

一般我们使用 GPIO 的端口，都需要有以下几个步骤。

- 开启 GPIO 的端口时钟
- 配置 GPIO 的模式
- 配置 GPIO 的输出

从开发板原理图了解到 LED2 接的是单片机的 PC0。我们要使能 LED 就需要配置 GPIOC 端口。下面我们就以 LED1 接的 PC0 进行介绍。

2.3.1.1.开启 CW32F003 系统内部时钟

CW32 的系统内部时钟默认并不是我们想要的运行频率,在使用 CW32 之前我们需要先配置内部时钟树。查找 CW32F003 的用户手册可以找到系统内部时钟树如下图：

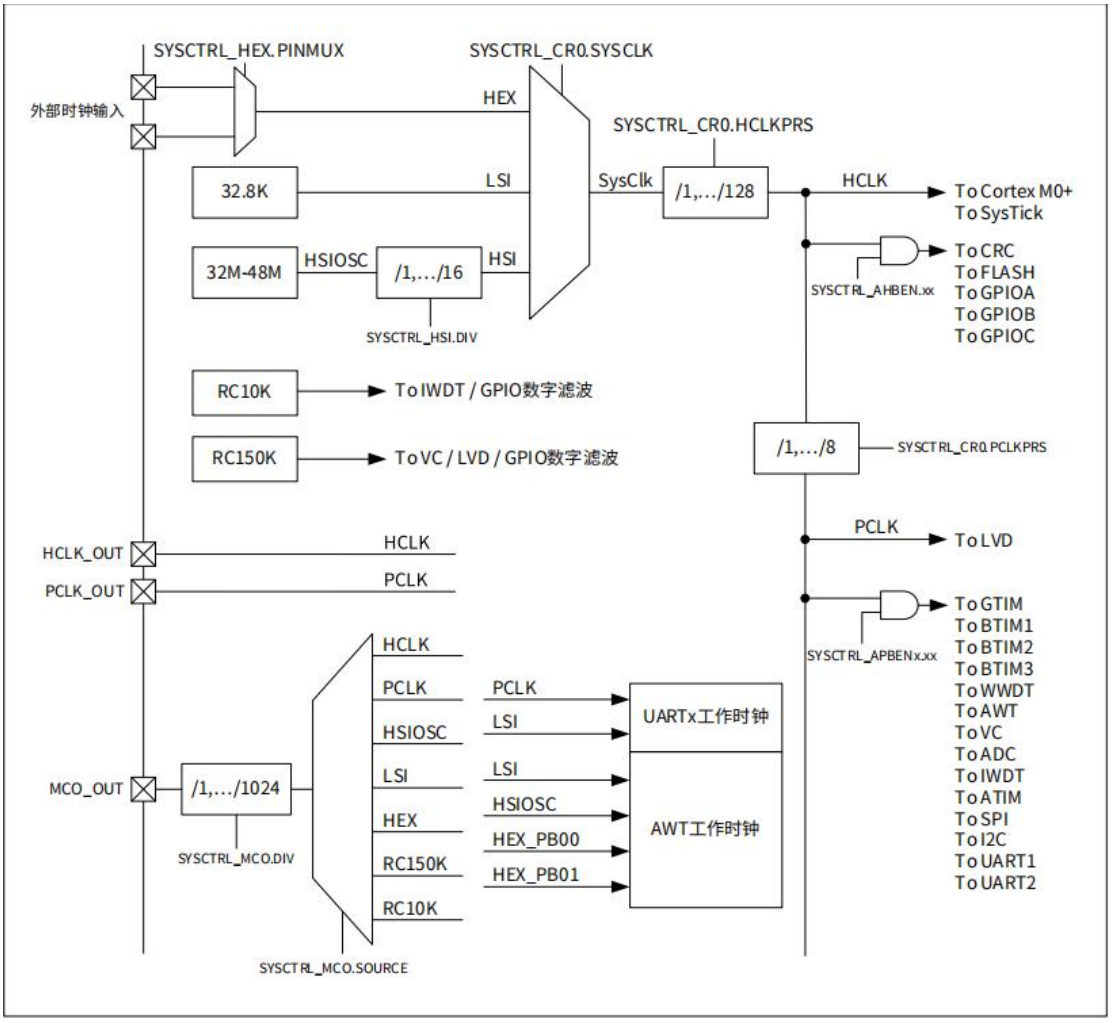

图 2-5 CW32F003 内部时钟树

在 CW32 的库函数中找到 cw32f003_rcc.h，头文件中有时钟树相关配置的函数，如下图：

```
extern void RCC_HCLKPRS_Config(uint32_t HCLKPRS);
extern void RCC_PCLKPRS_Config(uint32_t PCLKPRS);
extern void RCC_SYSCLKSRC_Config(uint32_t SYSCLKSRC);
extern void RCC_LSI_Enable(void);
extern void RCC_LSI_Disable(void);
extern void RCC_HEX_Enable(uint32_t PINMUX);
extern void RCC_HEX_Disable(void);
extern void RCC_HSI_Enable(uint32_t HSIDiv);
extern void RCC_HSI_Disable(void);
extern void RCC_WAKEUPCLK_Config(uint32_t WAKEUPCLK);
extern void RCC_LOCKUP_Config(uint32_t LOCKUP);
extern void RCC_SWDIO_Config(uint32_t SWDIO);
extern void RCC_RSTIO_Config(uint32_t RSTIO);
extern void RCC_DeInit(void);
extern void RCC_SystemCoreClockUpdate(uint32_t NewFreq);
extern uint8_t RCC_SysClk_Switch(uint32_t NewClk);
extern void RCC_ITConfig(uint32_t RCC_IT, FunctionalState NewState);
extern ITStatus RCC_GetITStatus(uint32_t RCC_IT);
extern void RCC_ClearITPendingBit(uint32_t RCC_IT);
extern FlagStatus RCC_GetStableFlag(uint32_t RCC_STABLEFLAG);
extern uint32_t RCC_GetAllStableFlag( void );
extern void RCC_AHBPeriphClk_Enable(uint32_t Periph,FunctionalState NewState);
extern void RCC_APBPeriphClk_Enable1(uint32_t Periph,FunctionalState NewState);
extern void RCC_APBPeriphClk_Enable2(uint32_t Periph,FunctionalState NewState);
extern void RCC_AHBPeriphReset(uint32_t Periph,FunctionalState NewState);
extern void RCC_APBPeriphReset1(uint32_t Periph,FunctionalState NewState);
extern void RCC_APBPeriphReset2(uint32_t Periph,FunctionalState NewState);
extern FlagStatus RCC_GetRstFlag(uint32_t RCC_RSTFLAG);
extern uint32_t RCC_GetAllRstFlag( void );
extern void RCC_ClearRstFlag(uint32_t RCC_RSTFLAG);
extern void RCC_HCLK_OUT(void);
extern void RCC_PCLK_OUT(void);
extern void RCC_MCO_OUT(uint8_t Source, uint8_t Div);
extern void RCC_DEBUG_Config(uint32_t Periph, FunctionalState NewState);

extern uint32_t RCC_Sysctrl_GetHClkFreq(void);
```

图 2-6 时钟树相关配置函数

```
C
void RCC_Configuration(void)
{
  FLASH_SetLatency(FLASH_Latency_2); // 设置主频为 48MHZ 需要注意，
Flah 的访问周期需要更改为 FLASH_Latency_2。
  RCC_HSI_Enable(RCC_HSIOSC_DIV1);   // 设置频率为 48M
  RCC_SYSCLKSRC_Config(RCC_SYSCLKSRC_HSI);  //选择 SYSCLK 时钟源
48MHz
  RCC_HCLKPRS_Config(RCC_HCLK_DIV1);  //配置 SYSTICK 到 HCLK 分频系数
48MHz
  RCC_PCLKPRS_Config(RCC_PCLK_DIV8); //配置 HCLK 到 PCLK 的分频系数
6MHz
}
```

2.3.1.2.开启 GPIO 的端口时钟

CW32 的外时钟默认是全部关闭,使用 GPIO 外设之前我们需要先开启对应的时钟。

在 CW32 提供的库函数中找到 cw32f003_rcc.h，这个头文件包含了所有时钟相关的函数接口。外设时钟的接口如下图所示:

```
189  /* SYSCTRL AHPEN1 ------------------------------------------------------*/
190  #define RCC_APB1_PERIPH_I2C                     bv11
191  #define RCC_APB1_PERIPH_UART2                   bv7
192  #define RCC_APB1_PERIPH_IWDT                    bv5
193  #define RCC_APB1_PERIPH_WWDT                    bv4
194  #define RCC_APB1_PERIPH_GTIM                    bv1
195  #define IS_RCC_APB1_PERIPH(PERIPH)              ((((PERIPH)&0xFFFFF74D)==0x00)&&((PERIPH)!=0x00))
196
197  #define __RCC_I2C_CLK_ENABLE()                  REGBITS_SET(CW_SYSCTRL->APBEN1,RCC_APB1_PERIPH_I2C)
198  #define __RCC_UART2_CLK_ENABLE()                REGBITS_SET(CW_SYSCTRL->APBEN1,RCC_APB1_PERIPH_UART2)
199  #define __RCC_IWDT_CLK_ENABLE()                 REGBITS_SET(CW_SYSCTRL->APBEN1,RCC_APB1_PERIPH_IWDT)
200  #define __RCC_WWDT_CLK_ENABLE()                 REGBITS_SET(CW_SYSCTRL->APBEN1,RCC_APB1_PERIPH_WWDT)
201  #define __RCC_GTIM_CLK_ENABLE()                 REGBITS_SET(CW_SYSCTRL->APBEN1,RCC_APB1_PERIPH_GTIM)
202
203  #define __RCC_I2C_CLK_DISABLE()                 (CW_SYSCTRL->APBEN1&=~(RCC_APB1_PERIPH_I2C))
204  #define __RCC_UART2_CLK_DISABLE()               (CW_SYSCTRL->APBEN1&=~(RCC_APB1_PERIPH_UART2))
205  #define __RCC_IWDT_CLK_DISABLE()                (CW_SYSCTRL->APBEN1&=~(RCC_APB1_PERIPH_IWDT))
206  #define __RCC_WWDT_CLK_DISABLE()                (CW_SYSCTRL->APBEN1&=~(RCC_APB1_PERIPH_WWDT))
207  #define __RCC_GTIM_CLK_DISABLE()                (CW_SYSCTRL->APBEN1&=~(RCC_APB1_PERIPH_GTIM))
208
209  /* SYSCTRL AHPEN1 ------------------------------------------------------*/
210  #define RCC_APB2_PERIPH_AWT                     bv13
211  #define RCC_APB2_PERIPH_BTIM                    bv12
212  #define RCC_APB2_PERIPH_UART1                   bv9
213  #define RCC_APB2_PERIPH_SPI                     bv8
214  #define RCC_APB2_PERIPH_ATIM                    bv7
215  #define RCC_APB2_PERIPH_VC                      bv4
216  #define RCC_APB2_PERIPH_ADC                     bv2
217  #define IS_RCC_APB2_PERIPH(PERIPH)              ((((PERIPH)&0xFFFFCC63)==0x00)&&((PERIPH)!=0x00))
218
219  #define __RCC_AWT_CLK_ENABLE()                  REGBITS_SET(CW_SYSCTRL->APBEN2,RCC_APB2_PERIPH_AWT)
220  #define __RCC_BTIM_CLK_ENABLE()                 REGBITS_SET(CW_SYSCTRL->APBEN2,RCC_APB2_PERIPH_BTIM)
221  #define __RCC_UART1_CLK_ENABLE()                REGBITS_SET(CW_SYSCTRL->APBEN2,RCC_APB2_PERIPH_UART1)
222  #define __RCC_SPI_CLK_ENABLE()                  REGBITS_SET(CW_SYSCTRL->APBEN2,RCC_APB2_PERIPH_SPI)
223  #define __RCC_ATIM_CLK_ENABLE()                 REGBITS_SET(CW_SYSCTRL->APBEN2,RCC_APB2_PERIPH_ATIM)
224  #define __RCC_VC_CLK_ENABLE()                   REGBITS_SET(CW_SYSCTRL->APBEN2,RCC_APB2_PERIPH_VC)
225  #define __RCC_ADC_CLK_ENABLE()                  REGBITS_SET(CW_SYSCTRL->APBEN2,RCC_APB2_PERIPH_ADC)
226
227  #define __RCC_AWT_CLK_DISABLE()                 (CW_SYSCTRL->APBEN2&=~(RCC_APB2_PERIPH_AWT))
228  #define __RCC_BTIM_CLK_DISABLE()                (CW_SYSCTRL->APBEN2&=~(RCC_APB2_PERIPH_BTIM))
229  #define __RCC_UART1_CLK_DISABLE()               (CW_SYSCTRL->APBEN2&=~(RCC_APB2_PERIPH_UART1))
230  #define __RCC_SPI_CLK_DISABLE()                 (CW_SYSCTRL->APBEN2&=~(RCC_APB2_PERIPH_SPI))
231  #define __RCC_ATIM_CLK_DISABLE()                (CW_SYSCTRL->APBEN2&=~(RCC_APB2_PERIPH_ATIM))
232  #define __RCC_VC_CLK_DISABLE()                  (CW_SYSCTRL->APBEN2&=~(RCC_APB2_PERIPH_VC))
233  #define __RCC_ADC_CLK_DISABLE()                 (CW_SYSCTRL->APBEN2&=~(RCC_APB2_PERIPH_ADC))
```

图 2-7 GPIO 配置相关函数

LED1 的控制 IO 是 PC0,因此需要打开 GPIOC 对应的时钟,代码如下:

```
C
__RCC_GPIOC_CLK_ENABLE();//打开 GPIOC 的时钟
```

2.3.1.3.配置 GPIO 初始化

GPIO 初始化包含了模式,中断使能,io 引脚位号。CW32 的 GPIO 初始化是先把所

有初始化项写在一个结构体里面，然后把结构体传入带初始化函数里，完成初始化功能。

与 GPIO 相关的函数接口都在 cw32f003_gpio.h 头文件里面，我们先看 GPIO 模式有哪些，如下图所示：

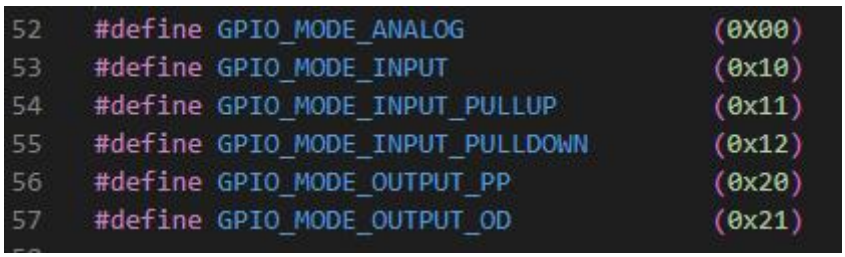

图 2-8 GPIO 模式

GPIO 模式包含了模拟输入，数字输入，数字上拉输入，数字下拉输入，数字推挽输出，数字开漏输出模式。

根据发光二极管控制原理，PC0 接在 LED 的反向端，因此 PC0 为低电平时，LED 点亮。我们需要 PC0 输出高低电平，模式需要选择数字输出模式，由于开漏模式需要外部有上拉或者下拉电阻才可输出高低电平，所以我们选择推挽模式。代码如下：

```c
GPIO_Init_Struct.Mode = GPIO_MODE_OUTPUT_PP;
```

中断使能包含了：无中断，上升沿，下降沿，低电平，高电平中断使能。

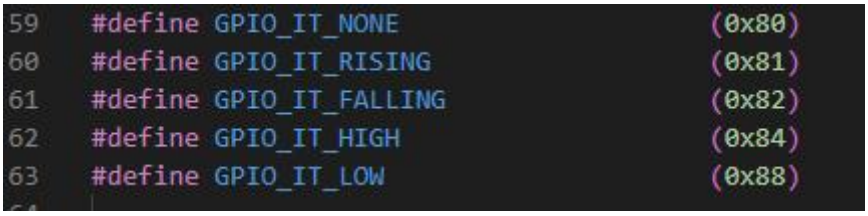

图 2-9 GPIO 中断

因为我们需要的是 GPIO 输出，因此不需要中断使能，代码如下：

```c
GPIO_Init_Struct.IT = GPIO_IT_NONE;
```

GPIO 位号如下图所示：

```
#define GPIO_PIN_0                 ((uint16_t)0x0001)/*Pin0selected*/
#define GPIO_PIN_1                 ((uint16_t)0x0002)/*Pin1selected*/
#define GPIO_PIN_2                 ((uint16_t)0x0004)/*Pin2selected*/
#define GPIO_PIN_3                 ((uint16_t)0x0008)/*Pin3selected*/
#define GPIO_PIN_4                 ((uint16_t)0x0010)/*Pin4selected*/
#define GPIO_PIN_5                 ((uint16_t)0x0020)/*Pin5selected*/
#define GPIO_PIN_6                 ((uint16_t)0x0040)/*Pin6selected*/
#define GPIO_PIN_7                 ((uint16_t)0x0080)/*Pin7selected*/
#define GPIO_PIN_All               ((uint16_t)0x00FF)/*Allpinsselected*/
```

图 2-10 GPIO 位脚

LED 的控制端为 PC0，最后的这个 0 就是位号，配置代码如下：

```c
GPIO_Init_Struct.Pins = GPIO_PIN_0;
```

引脚输出还需要配置 GPIO 的输出速度，有高低速两种速度设置，我们设置高速：

```c
GPIO_Init_Struct.Speed = GPIO_SPEED_HIGH;
```

以上配置就完成了 GPIO 的结构体初始化，然后我们需要调用初始化函数，在 cw32f003_rcc.h 里找到 void GPIO_Init(GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_Init);包含了两个参数，第一个参数是 GPIO 的基地址，第二个参数是初始化配置结构体。调用代码如下：

```c
GPIO_Init(CW_GPIOC, &GPIO_Init_Struct);
```

结合以上配置，完整代码如下：

```c
  GPIO_InitTypeDef GPIO_Init_Struct;
  __RCC_GPIOC_CLK_ENABLE();
  GPIO_Init_Struct.IT   = GPIO_IT_NONE;
  GPIO_Init_Struct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_Init_Struct.Pins = GPIO_PIN_0;
  GPIO_Init(CW_GPIOC, &GPIO_Init_Struct);
```

2.3.1.4.配置 LED 输出

配置好 GPIO 之后，就可以进行点灯了。就是让 PC0 输出高低电平。

在 cw32f003_gpio.h 头文件中可以找到函数

```
C
GPIO_WritePin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pins,
GPIO_PinState PinState);
```

传参有三个，第一个是第一个参数是 GPIO 的基地址，第二个参数是引脚位号，第三个参数是输出状态。

```
C
GPIO_WritePin(CW_GPIOC,GPIO_PIN_0,GPIO_Pin_SET);//PC0 输出高电平
GPIO_WritePin(CW_GPIOC,GPIO_PIN_0,GPIO_Pin_RESET);//PC0 输出低电平
```

我们要点亮 LED1 根据上文的电路图可知要将 PC0 置低。

2.3.1.5.程序下载接线

我们使用 DAP-Link 作为程序下载媒介，DAP-Link 的接线图如下。其中蓝色（3.3V）、绿色（GND）、

紫色（数据线 SWD）、灰色（时钟信号线 SCK），与图 2-12 一一对应。

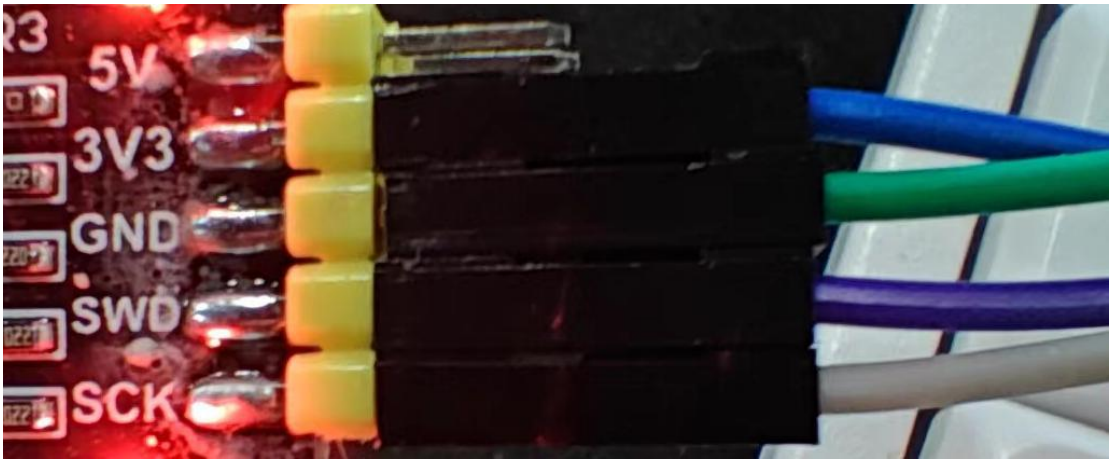

图 2-11 DAP-Link 接线图

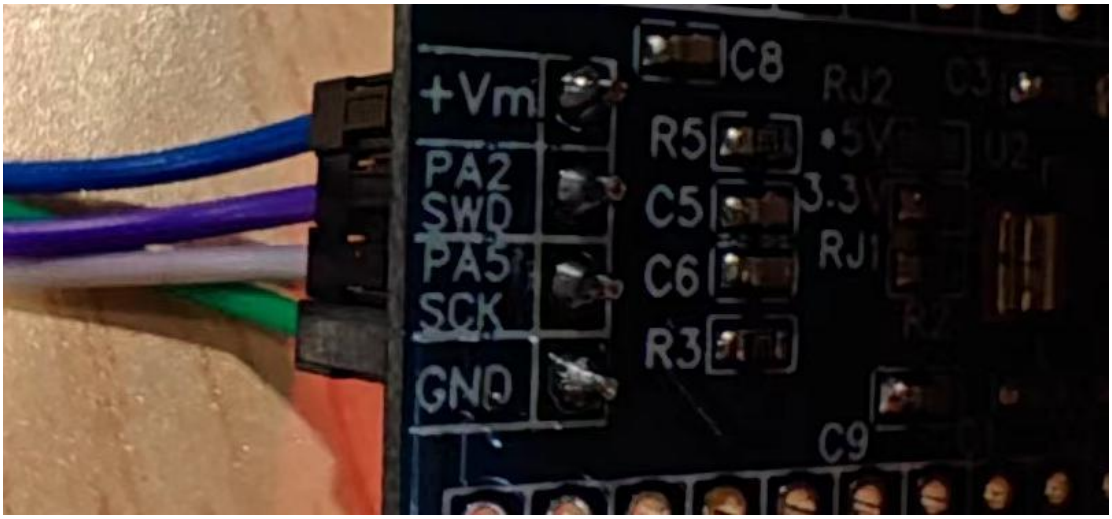

图 2-12 CW32F003 程序下载接线

2.3.2.实验效果

最终的实验效果如下图所示：

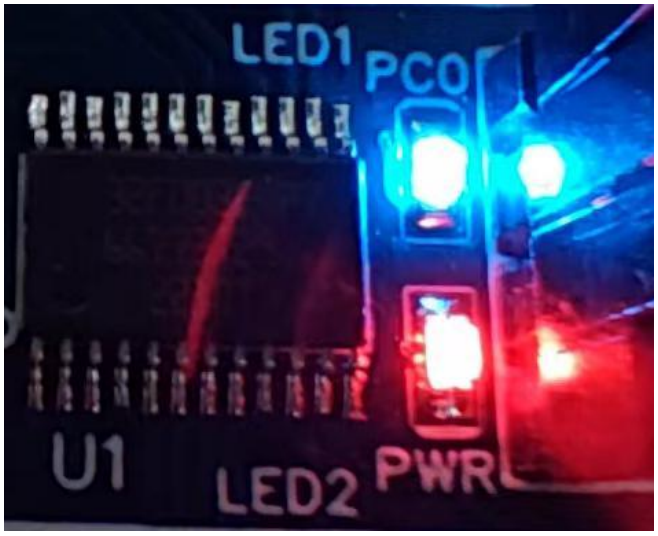

图 2-13 LED 点亮实验现象

三、实验二：按键检测

3.1.独立按键基础知识

3.1.1.独立按键结构组成

独立按键实际上是一个非自锁的轻触开关，有左右两个触点，当按下时左右两个触点闭合，当松开时左右两个触点断开。

3.1.2.独立按键控制原理

想要使用外部的按键控制单片机有两种比较常见的方法：IO 扫描和外部中断。

对于 IO 扫描的方式而言，需要单片机以比较高的频率去不间断地判断 IO 口的输入电平，随后根据 IO 电平来执行后续的逻辑。外部中断的方式会在章节十二中进行详细介绍，本章着重介绍 IO 扫描的控制方式。（单片机通过检测按键按下前后的高低电平变化，来判断按键是否按下。通过程序的控制，就可以实现不同的功能与设置。机械式按键在按下或者释放时，由于机械弹性作用的影响，通常伴随有一定时间的触点机械抖动，然后其触点才稳定下来。抖动时间长短与开关的机械特性有关，一般为 5-10ms。在触点抖动期间检测按键的按下与否，可能会导致判断错误，为了克服机械抖动所产生的影响，必须采取消抖措施，可分为硬件消抖和软件消抖。）

3.1.3.按键亚稳态与按键消抖

对于一个 IO 而言，在将其配置为输入模式之后，该引脚上的电平受外部电路影响，基本可以分为三种状态：高电平、低电平、浮空。高低电平很好理解，这里说明浮空的意义，浮空就是不对该 IO 进行任何电气属性的连接，此时该 IO 上的电平是未知的（虽然从直观感受上来看此时 IO 电压应该是 0，但是空气中会有噪声，电路板上也会有噪声，某些电磁干扰也会充当噪声，所以浮空输入的 IO 电压实际上是未知的）。

典型的浮空输入型 IO 电路如图 3-1 所示：

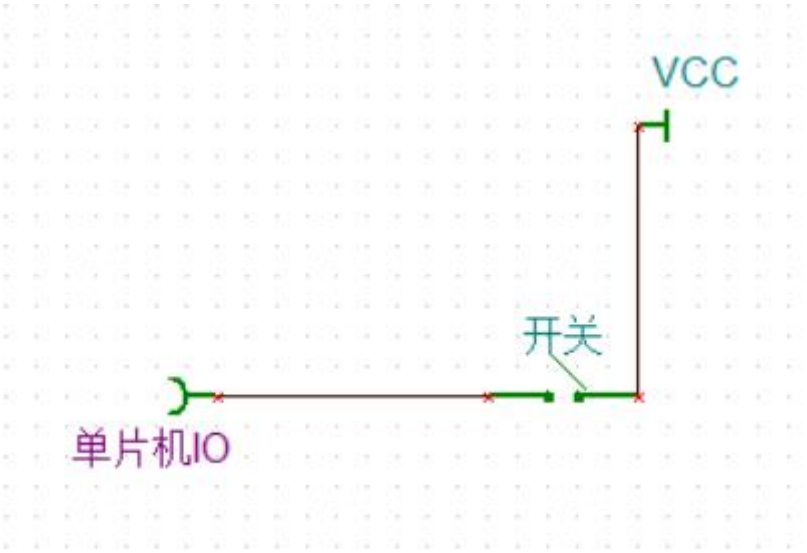

图 3-1 浮空输入 IO 电路

当开关闭合，IO 电压等于 VCC 电压，当开关断开，IO 电压未知，此时 IO 电压可能会受到不明来源的干扰，如果使用该电路作为 IO 扫描的电路方案，抗干扰能力会不好，容易造成误触发，故此应用场景下不考虑使用该电路。

和浮空输入相比，比较好的办法是使用上拉电阻或下拉电阻将 IO 的电压固定下来，带上拉电阻或下拉电阻电路如图 3-2 所示：

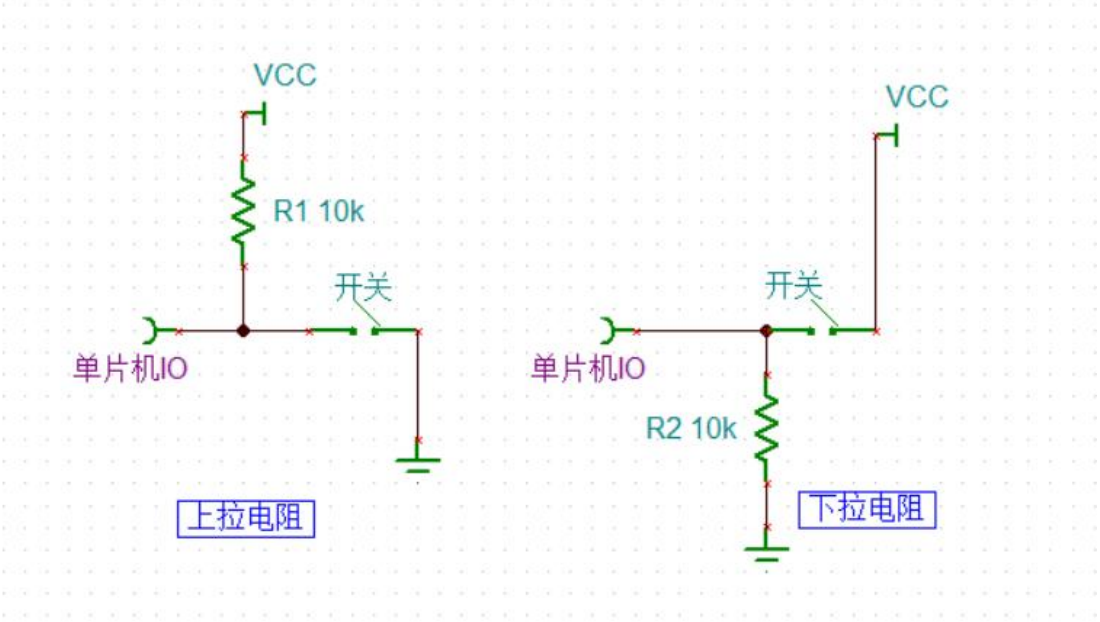

图 3-2 上拉电阻（左） 和 下拉电阻（右）电路

上拉电阻可以将 IO 电压固定在 VCC 电压，当开关闭合时，IO 接地使其电压变为 GND 电压；下拉电阻可以将 IO 电压固定在 GND 电压，当开关闭合时，IO 电压其实就是电阻 R2 的电压，此时 R2 的电压就是 VCC。使用上拉/下拉电阻可以很好地提高 IO 扫描的抗干扰性能，一般情况这两个电路不会有很大区别，挑一个你喜欢的用就好。

按键通过金属导体的相互接触来控制电信号，由于机械特性，这种接触实际上并不可靠，手指按下按键不代表按键真的闭合且保持稳定，这种情况就是按键抖动，抖动过程中按键控制的信号处于亚稳态，亚稳态的信号不可靠，不能将其作为 IO 扫描的最终结果，为了获取正确的按键状态，我们需要对按键进行消抖处理，按键消抖大概可以分为两种方式：

- 硬件消抖：硬件消抖一般会在按键两端并联电容，通过电容的充放电作用将按键按下时的高频振荡吸收掉，当开关处于亚稳态时，IO 电压不规则变化，电容会吸收这些不稳定电压进行充电，这对 IO 电压有平缓的效果，以此达到消抖的目的，硬件消抖电路如图 3-3 所示。

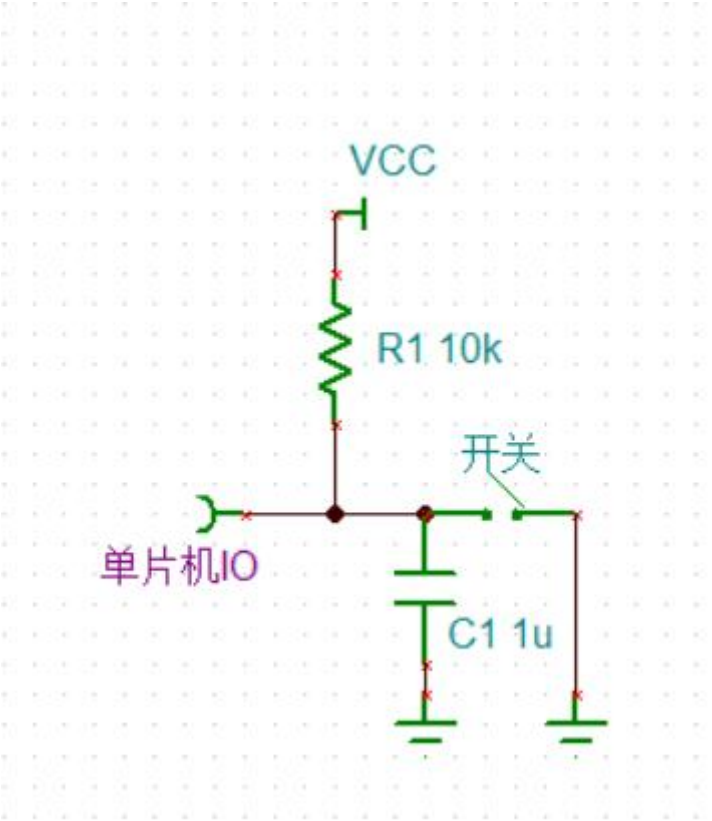

图 3-3 硬件消抖电路

- 简单的软件消抖：极为简单的软件消抖一般是通过延时的办法来跳过亚稳态阶段，当检测到按键按下时，不会立即去检测电平，而是经过短暂的延时之后，再去检测当前引脚的电平，这能在一定程度上消除亚稳态带来的影响，但需要对按键按下和抬起都进行延时判断才能更为有效。
- 更好的软件消抖：在监测到 IO 电平发生变化后的一小段时间内快速采集 IO 的电平状态，如果这一小段时间内 IO 电平全都属于有效电平，则认为按键已按下（这种利用数学进行消抖的方式达到了对数字信号的筛选作用，所以他也是一种简易滤波器）。

在条件允许的情况下，硬件消抖的效果会更好，如果 PCB 没有多余的空间留给这个消抖电容，使用软件消抖同样是一个不错的方案。

3.2.独立按键原理图

CW32F003 核心板上一共有两个按键，一个复位和一个用户按键，复位作为单片机的特殊功能，不可以作为按键使用，故只有用户按键可以作为按键使用。

CW32F003 核心板关于独立按键的原理图如图 3-4 所示。

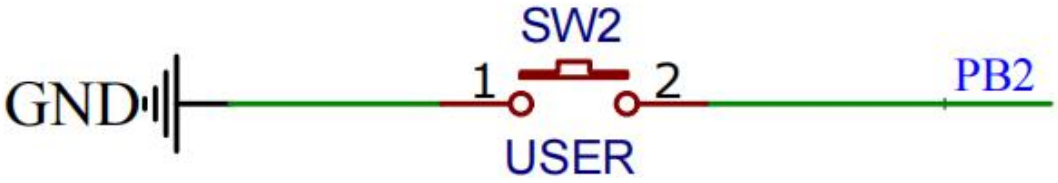

图 3-4 CW32F003 按键电路

3.3.独立按键驱动流程

通过上面的原理图可以了解到，按键的一端接到了地，另一端接到单片机的 PB2 引脚上。通过检测 PB2 引脚的电平状态，判断按键是否按下。当按键松开的时候，PB2 检测到的电平为高电平，当按键按下的时候，PB2 检测到的电平为低电平。

外部电路不含上下拉电阻，对 IO 而言是浮空输入，因此需要使用单片机内部的上下拉电阻；电路不含消抖电容，故编程上需要对按键进行软件消抖。

3.4.按键控制 LED 灯亮灭

3.4.1.配置流程

一般我们使用 GPIO 的输入功能，都需要有以下几个步骤。

- 开启 GPIO 的端口时钟
- 配置 GPIO 的模式
- 配置 GPIO 的输入
- 编写消抖函数

从开发板原理图了解到按键接的是单片机的 PB2。我们要使能按键就需要配置 GPIOB 端口。下面我们就以按键连接的 PB2 进行介绍。

3.4.1.1.开启 GPIO 的端口初始化

由于时钟的配置在之前的章节已有说明，故不再赘述，我们直接对端口进行初始化。初始化的代码与上文 GPIO 输出的配置略有不同，完整代码如下：

```
C
void Gpio_Init(void)
```

```
{
  __RCC_GPIOC_CLK_ENABLE();//打开 GPIOC 的时钟，PC0 控制 LED 亮灭
  GPIO_InitTypeDef GPIO_InitStruct;
  GPIO_InitStruct.Pins = GPIO_PIN_0;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.IT = GPIO_IT_NONE;
  GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
  GPIO_Init(CW_GPIOC, &GPIO_InitStruct);

  __RCC_GPIOB_CLK_ENABLE();//打开 GPIOB 的时钟，PB2 控制按键输入
  GPIO_InitStruct.Pins = GPIO_PIN_2;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT_PULLUP;  //没有输入时 PB2 默
认为高电平
  GPIO_InitStruct.IT = GPIO_IT_NONE;
  GPIO_Init(CW_GPIOB, &GPIO_InitStruct);
}
```

3.4.1.2.编写消抖函数

本次实验采用软件消抖，消抖函数的编写思路为：设置按键检测标志位（Flag_Key），当单片机检测到按键按下，即 PB2 为低电平时，将标志位置 1；在后续的条件判断中如果标志位为 1，则检测按键是否松开，若已松开则完成本次判断，认为按键已经按下过一次。这种方式可以不用延时判断，节约软件资源。

完整的按键检测程序如下：

```
C
uint8_t Flag_Key;         //按键标志位
extern uint8_t Flag_LED;         //LED 显示标志位
void Key_Scan(void)
{
  if(GPIO_ReadPin(CW_GPIOB,GPIO_PIN_2) == GPIO_Pin_RESET)  //检测
PB2 是否为低电平
    {
         Flag_Key = 1;
    }
  if(Flag_Key)     //接着判断标志位
    {
      if(GPIO_ReadPin(CW_GPIOB,GPIO_PIN_2) == GPIO_Pin_SET)    //如
果按键已经松开
       {
         Flag_Key = 0;   //清零标志位，等待下一次按键检测
         if(Flag_LED == 0) Flag_LED = 1; //按键按下该变 LED 显示标志
```

```
位的值，由显示标志位控制 LED
          else Flag_LED = 0;
        }
    }
}
```

3.4.1.3.LED 显示函数

在按键按下更改 LED 显示标志位后，需要根据显示标志位的值来控制 LED 灯的亮灭。

```
C
uint8_t Flag_LED;

void LED_Init(void)
{
  GPIO_WritePin(CW_GPIOC,GPIO_PIN_0,GPIO_Pin_SET);  //初始化让 LED
灯处于熄灭状态
}
void LED_Lighting(void)
{
  if(Flag_LED == 1)
    {
      GPIO_WritePin(CW_GPIOC,GPIO_PIN_0,GPIO_Pin_RESET);        //
亮
    }
  else
    {
      GPIO_WritePin(CW_GPIOC,GPIO_PIN_0,GPIO_Pin_SET);       //灭
    }
}
```

最终主函数里只需要运行相应的初始化函数和上面的函数：

```
C
int main()
{
  RCC_Configuration();
  Gpio_Init();
  LED_Init();

  while(1)
   {
```

```
    Key_Scan();
    LED_Lighting();
  }
}
```

四、实验三：数码管显示数字

4.1.数码管显示原理（来源：CSDN，原文链接：https://blog.csdn.net/qq_42189951/article/details/133347707）

数码管的显示原理是由多个发光的二极管共阴极或者共阳极组成的成“8”字形的显示器件。数码管通过不同的组合可用来显示数字 0~9、字符 A ~ F 及小数点“.”。数码管的工作原理是通过控制外部的 I/O 端口进行驱动数码管的各个段码，使用不同的段码从而形成字符显示出我们要的数字。数码管实际上是由七个发光管组成 8 字形构成的，加上小数点就是 8 个。这些段分别由字母 A、B、C、D、E、F、G、DP 来表示。

当数码管特定的引脚加上高电平后，这些特定的发光二极管就会发亮，以形成我们眼睛看到的字样了。如：在一个共阴极数码管上显示一个“8”字，那么就对 A、B、C、D、E、F、G 对应的引脚置高电平。发光二极管的阳极共同连接至电源的正极称为共阳极数码管，这种类型的数码管点亮需要对引脚置低电平；发光二极管的阴极共同连接到电源的负极称为共阴极数码管，点亮共阴极数码管需要对相应的引脚置高电平。常用 LED 数码管显示的数字和字符是 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F。

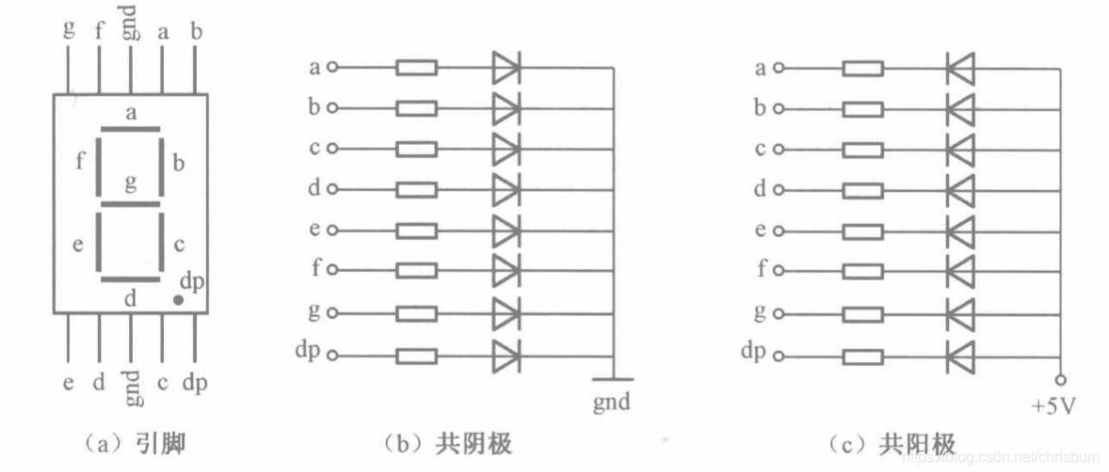

图 4-1 数码管模型图

共阳极数码管的 8 个发光二极管的阳极（二极管正端）连接在一起。通常，公共阳极接高电平（一般接电源），其它管脚接段驱动电路输出端。当某段驱动电路的输出端为低电平时，则该端所连接的字段导通并点亮。根据发光字段的不同组合可显示出各种数字或字符。此时，要求段驱动电路能吸收额定的段导通电流，还需根据外接电源及额定段导通电流来确定相应的限流电阻。

共阴极数码管的 8 个发光二极管的阴极（二极管负端）连接在一起。通常，公共阴极接低电平（一般接地），其它管脚接段驱动电路输出端。当某段驱动电路的输出端为高电平时，则该端所连接的字段导通并点亮，根据发光字段的不同组合可显示出各种数字或字符。此时，要求段驱动电路能提供额定的段导通电流，还需根据外接电源及额定段导通电流来确定相应的限流电阻。

4.2.数码管原理图与实物图

如果数码管可以显示多位数字，如我们的电压电流表所示。那么除了控制段码来选择要显示的内容，还要选择位码来控制某一个数码管的亮灭。

图 4-2 电压电流表三位数码管

数码管的原理图如下，可以看出除了上述的段码引脚之外，还有 COM1、COM2、COM3 的位码引脚，三个位码引脚分别控制三个数码管的亮灭情况，且低电平有效。

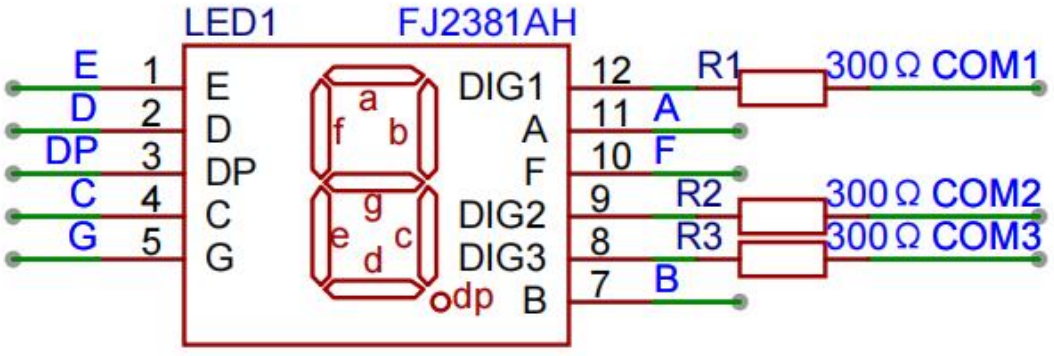

图 4-3 三位数码管原理图

4.3.数码管驱动显示

驱动显示数码管的思路是：先将 A、B、C、D、E、F、G 所代表的引脚从低到高编号，列出数码要显示数字的段码值。比如要显示数字 5，则段码值为 0x6d，二进制表示为 01101101，这说明 G 置 1，F 置 1，E 置 0，D 置 1，C 置 1，B 置 0，A 置 1，最高位则是 DP 的值。将要显示的数字以段码值的方式储存在数组里以供调用，可以简化程序。接着以循环的方式结合 switch 语句对 A、B、C、D、E、F、G 的亮灭情况进行单独计算，先将段码值确定后再进行位码的选择，可以避免因单片机执行程序的时间而造成显示效果的不足。

具体程序如下，将所有与数码管显示相关的函数保存在新建的 Seg_Reg.c 文件中：

```
C
/*  共阴数码管编码表：
 0x3f   0x06   0x5b  0x4f  0x66  0x6d  0x7d  0x07  0x7f  0x6f
  0      1      2     3     4     5     6     7     8     9
 0xbf   0x86   0xdb  0xcf  0xe6  0xed  0xfd  0x87  0xff  0xef
  0.     1.     2.    3.    4.    5.    6.    7.    8.    9. */


uint8_t Seg_Table[20] = {0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d,
0x07, 0x7f, 0x6f,
                         0xbf, 0x86, 0xdb, 0xcf, 0xe6, 0xed, 0xfd,
0x87, 0xff, 0xef};
/*对段码值进行存储*/

void Seg_Init(void)      //查找原理图对数码管相关引脚进行初始化
{
  __RCC_GPIOA_CLK_ENABLE();//打开 GPIOA 的时钟
  __RCC_GPIOB_CLK_ENABLE();//打开 GPIOB 的时钟
  __RCC_GPIOC_CLK_ENABLE();//打开 GPIOC 的时钟

  GPIO_InitTypeDef GPIO_InitStruct;

  GPIO_InitStruct.Pins = GPIO_PIN_0 | GPIO_PIN_4; //PA00,E;PA04,G
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.IT = GPIO_IT_NONE;
  GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
  GPIO_Init(CW_GPIOA, &GPIO_InitStruct);
```

```
  GPIO_InitStruct.Pins = GPIO_PIN_6 | GPIO_PIN_4 | GPIO_PIN_2 |
GPIO_PIN_0 | GPIO_PIN_3 | GPIO_PIN_7;
//PB06,B;PB04,C;PB02,D;PB00,F;PB03,DP  //PB07，COM1
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.IT = GPIO_IT_NONE;
  GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
  GPIO_Init(CW_GPIOB, &GPIO_InitStruct);

  GPIO_InitStruct.Pins = GPIO_PIN_4 | GPIO_PIN_3 | GPIO_PIN_2;
//PC04,A; //PC03,COM2;PC02,COM3
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.IT = GPIO_IT_NONE;
  GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
  GPIO_Init(CW_GPIOC, &GPIO_InitStruct);
}
```

```C
void Seg_Dis(uint8_t Pos,uint8_t Num)   //Pos 表示指定哪一个数码管
亮，Num 表示要显示的数字
{
  int i;
  uint8_t Dis_Value,Location;
  Location = Pos;
  Dis_Value = Seg_Table[Num];

  for(i = 0; i < 8; i++)      //通过循环确定每一个段码引脚的亮灭情况
  {
    switch(i)
      {
/*将 Dis_Value 右移 i 位，再和 0x01（00000001）相与消除其他位的影响，可以
确定该位的写入值，
  学员可以自己用一个实例比如 0x6d（01101101）进行分析*/
       case 0:
         GPIO_WritePin(CW_GPIOC,GPIO_PIN_4,(Dis_Value >> i) &
0x01);    //PC04,A
         break;
       case 1:
         GPIO_WritePin(CW_GPIOB,GPIO_PIN_6,(Dis_Value >> i) &
0x01);    //PB06,B
         break;
       case 2:
         GPIO_WritePin(CW_GPIOB,GPIO_PIN_4,(Dis_Value >> i) &
0x01);    //PB04,C
```

```
        break;
      case 3:
        GPIO_WritePin(CW_GPIOB,GPIO_PIN_2,(Dis_Value >> i) &
0x01);    //PB02,D
        break;
      case 4:
        GPIO_WritePin(CW_GPIOA,GPIO_PIN_0,(Dis_Value >> i) &
0x01);    //PA00,E
        break;
      case 5:
        GPIO_WritePin(CW_GPIOB,GPIO_PIN_0,(Dis_Value >> i) &
0x01);    //PB00,F
        break;
      case 6:
        GPIO_WritePin(CW_GPIOA,GPIO_PIN_4,(Dis_Value >> i) &
0x01);    //PA04,G
        break;
      case 7:
        GPIO_WritePin(CW_GPIOB,GPIO_PIN_3,(Dis_Value >> i) &
0x01);    //PB03,DP
        break;
      default:
        break;
    }
  }

  switch(Location)    //确定段码后再选择位码
  {
    case 0:
      GPIO_WritePin(CW_GPIOB,GPIO_PIN_7,GPIO_Pin_RESET);
//PB07,COM1
      GPIO_WritePin(CW_GPIOC,GPIO_PIN_3,GPIO_Pin_SET);
//PC03,COM2
      GPIO_WritePin(CW_GPIOC,GPIO_PIN_2,GPIO_Pin_SET);
//PC02,COM3
      break;
    case 1:
      GPIO_WritePin(CW_GPIOB,GPIO_PIN_7,GPIO_Pin_SET);
//PB07,COM1
      GPIO_WritePin(CW_GPIOC,GPIO_PIN_3,GPIO_Pin_RESET);
//PC03,COM2
      GPIO_WritePin(CW_GPIOC,GPIO_PIN_2,GPIO_Pin_SET);
//PC02,COM3
      break;
```

```
    case 2:
      GPIO_WritePin(CW_GPIOB,GPIO_PIN_7,GPIO_Pin_SET);
//PB07,COM1
      GPIO_WritePin(CW_GPIOC,GPIO_PIN_3,GPIO_Pin_SET);
//PC03,COM2
      GPIO_WritePin(CW_GPIOC,GPIO_PIN_2,GPIO_Pin_RESET);
//PC02,COM3
      break;
    default:
      break;
  }
}
```

在主函数里调用 Seg_Dis 函数即可在对应位置显示相应数字（别忘了初始化），各位学员熟练之后可以通过 define 定义每个引脚的写入，使代码更加简洁美观。

程序下载接线如图 4-4 所示：

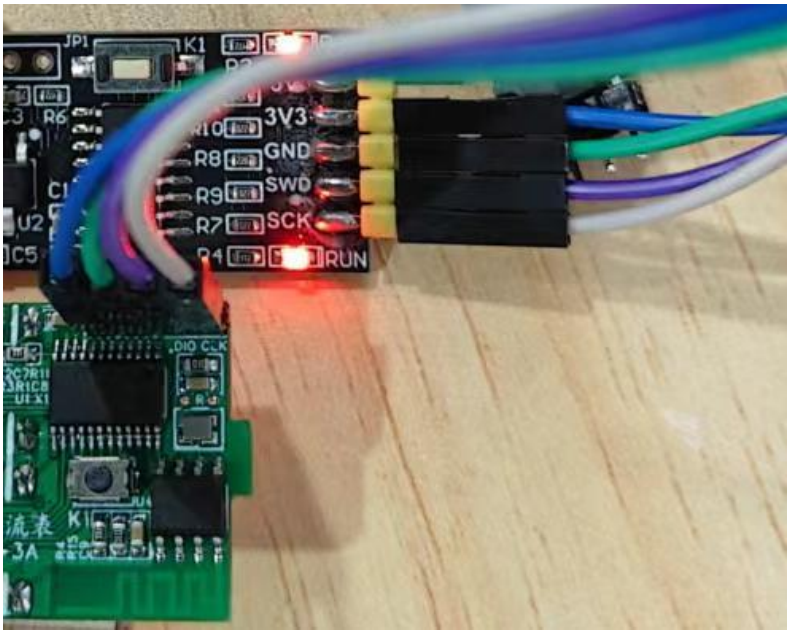

图 4-4 电压电流表程序下载接线

4.4.实验效果

最终的实验效果如下图所示：

图 4-5 数码管显示实验效果

五、实验四：数码管动态显示

5.1.数码管动态显示原理

所谓动态扫描显示即轮流向各位数码管送出段码和位码，利用发光管的余辉和人眼视觉暂留作用，使人眼的感觉好像各位数码管同时都在显示。明确了原理，我们要使电压电流表的三个位同时显示不同的值需要用到 CW32 的定时器功能，在定时器的中断服务程序里面执行显示刷新的动作。有关 CW32 的定时器和中断的相关知识请查看链接：【CW32F003E4 核心板】入门学习教程。本文只讲述如何配置定时器中断并执行数码管刷新函数。

5.2.定时器中断配置

在配置定时器之前需要注意是否将系统时钟树配置完成，定时器利用了高级外设时钟 PCLK，在之前的时钟配置中，我们将高级外设时钟 PCLK 的频率配置成了 6MHz，这将影响我们对定时器的预分频和装填计数值的配置。

在本次实验中，我们使用定时器 BTIM1 进行中断配置，代码如下：

```
C
#include "BTIM1.h"

void Btim1_Init(void)
{
    BTIM_TimeBaseInitTypeDef BTIM_TimeBaseInitStruct;
    __RCC_BTIM_CLK_ENABLE();          //使能 BTIM 的时钟
    NVIC_EnableIRQ(BTIM1_IRQn);       //使能 BTIM1 的中断

    BTIM_TimeBaseInitStruct.BTIM_Mode      = BTIM_Mode_TIMER;//将定
时器配置为计时模式
    BTIM_TimeBaseInitStruct.BTIM_Period    = 6000 - 1;      // 1ms
采集 1 次
    BTIM_TimeBaseInitStruct.BTIM_Prescaler = BTIM_PRS_DIV1; //
6MHZ÷1÷6000 = 1000Hz
    BTIM_TimeBaseInitStruct.BTIM_OPMode    =
BTIM_OPMode_Repetitive;//配置定时器连续运行

    BTIM_TimeBaseInit(CW_BTIM1, &BTIM_TimeBaseInitStruct);  //初始
化 BTIM1
    BTIM_ITConfig(CW_BTIM1, BTIM_IT_OV, ENABLE); //配置 BTIM1 的中
断，定时器溢出产生中断
    BTIM_Cmd(CW_BTIM1, ENABLE);                 //使能 BTIM1
}
```

5.3.数码管动态显示

在上一节的数码管显示数字的数码管显示模块 Seg_Dis.c 文件中，我们还需要添加三个函数和定义一个数组 Seg_Reg 来完成动态显示的功能。

```
C
```

```
uint8_t Seg_Reg[3] = {0,0,0};   //这个数组存放数码管显示三位的数字，0~9

void Close_Com(void)      //关闭所有数码管的显示，防止重影
{
  GPIO_WritePin(CW_GPIOB,GPIO_PIN_7,GPIO_Pin_SET);    //PB07,COM1
  GPIO_WritePin(CW_GPIOC,GPIO_PIN_3,GPIO_Pin_SET);    //PC03,COM2
  GPIO_WritePin(CW_GPIOC,GPIO_PIN_2,GPIO_Pin_SET);    //PC02,COM3
}

void Display(uint32_t value)
{
  uint8_t Hundreds;// 百位数
  uint8_t Tens;    // 十位数
  uint8_t Units;   // 个位数

  Units     = value % 10;      //分别取余获得位数的值
  Tens      = value / 10 % 10;
  Hundreds  = value / 100 % 10;

  Seg_Reg[0] = Hundreds;       //将个十百位的数字分别存放进数组等待调用
  Seg_Reg[1] = Tens;
  Seg_Reg[2] = Units;
}

void Dis_Refresh(void)      //函数将在定时器中断里调用，不断刷新数码管
{
/*静态变量从作用域上分属于局部变量；从生命周期上来看，它与用户程序的生命周期相同。*/
  static uint8_t num = 0;      //这里的静态临时变量 num，轮询 Seg_Reg 数组

  Close_Com();                //先关闭公共端,防止重影
  Seg_Dis(num,Seg_Reg[num]);   //调用显示函数
  num++;
  if(num > 2)
  {
    num = 0;
  }
}
```

最后在定时器中断里不断调用显示刷新函数完成数码管的刷新显示：

```c
C
void BTIM1_IRQHandler(void)        //BTIM1 的中断服务程序
{
  static uint32_t Cnt=0;           //Cnt 作为计数标志位控制刷新函数是否执行
  if (BTIM_GetITStatus(CW_BTIM1, BTIM_IT_OV))
  {
    BTIM_ClearITPendingBit(CW_BTIM1, BTIM_IT_OV);  //清除中断标志位
    Cnt++;
    if (Cnt > 2)                   //3ms 显示一个数码管
    {
      Cnt = 0;
      Dis_Refresh();               //数码管扫描显示
    }
  }
}
```

4.4.实验效果

最终的实验效果如下图所示，人眼已经看不出闪烁，但实际上数码管是依次刷新显示。

图 5-1 数码管动态显示效果图

六、实验五：ADC 采样及显示

6.1. ADC 基础知识

6.1.1. 什么是 ADC

模拟数字转换器即 A/D 转换器，或简称 ADC，通常是指一个将模拟信号转变为数字信号的电子元件。通常的模数转换器是将一个输入电压信号转换为一个输出的数字信号。由于数字信号本身不具有实际意义，仅仅表示一个相对大小。故任何一个模数转换器都需要一个参考模拟量作为转换的标准，比较常见的参考标准为最大的可转换信号大小。而输出的数字量则表示输入信号相对于参考信号的大小。

6.1.2. CW32 的 ADC 介绍

CW32F003 内部集成一个 12 位精度、最高 1M SPS 转换速度的逐次逼近型模数转换器 (SAR ADC)，最多可将 16 路模拟信号转换为数字信号。现实世界中的绝大多数信号都是模拟量，如光、电、声、图像信号等，都要由 ADC 转换成数字信号，才能由 MCU 进行数字化处理。

主要特性

• 12 位精度

• 可编程转换速度，最高达 1M SPS

• 16 路输入转换通道

- 13 路外部引脚输入

- 内置温度传感器

- 内置 BGR 1.2V 基准

- 1/3 VDD 电源电压

• 4 路参考电压源（Vref）

- VDD 电源电压

- ExRef（PB04）引脚电压

- 内置 1.5V 参考电压

- 内置 2.5V 参考电压

• 采样电压输入范围：0 ~ Vref

• 多种转换模式，全部支持转换累加功能

- 单次转换

- 多次转换

- 连续转换

- 序列扫描转换

- 序列断续转换

- 支持单通道、序列通道两种通道选择，最大同时支持 4 个序列
- 支持输入通道电压阈值监测
- 内置信号跟随器，可转换高阻抗输入信号
- 支持片内外设自动触发 ADC 转换

6.1.3. ADC 基本参数

分辨率：表示 ADC 转换器的输出精度，通常以位数（bit）表示，比如 8 位、10 位、12 位等，位数越高，精度越高。

采样率：表示 ADC 对模拟输入信号进行采样的速率，通常以每秒采样次数（samples per second，SPS）表示，也称为转换速率，表示 ADC 能够进行多少次模拟到数字的转换。

采样范围：指 ADC 可以采集到的模拟输入信号的电压范围，范围见下：

$$0 \le ADC \le Vref$$

Vref 为参考电压，CW32F003 有四路电压参考源见上文。

6.1.4. 基本原理

CW32F003 采用的是逐次逼近型的 12 位 ADC，逐次逼近型 ADC 是一种常见的 ADC 工作原理，它的思想是通过比较模拟信号与参考电压之间的大小关系来逐步逼近输入信号的数字表示。在逐次逼近型 ADC 中，输入信号和参考电压被加入一个差分放大器中，产生一个差分电压。然后，这个差分电压被输入到一个逐步逼近的数字量化器中，该量化器以逐步递减的方式将其与一系列参考电压进行比较。具体来说，在每个逼近阶段，量化器将输入信号与一个中间电压点进行比较，将该电压点上方或下方的参考电压作为下一个逼近阶段的参考电压。这个过程一直持续到量化器逼近到最终的数字输出值为止。

我们数字电压电流表的采样电路原理图如下图所示，

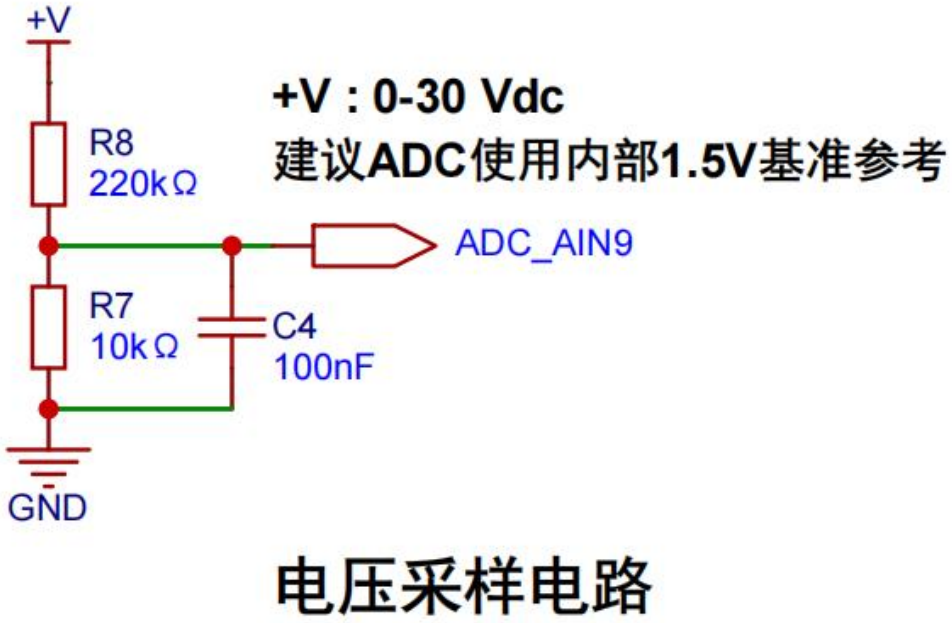

图 6-1 电压采样电路原理图

如果使用 1.5V 作为参考电压，根据 R8 和 R7 的阻值配比可以得到最高采样电压为：

1.5 / 10 * （220 + 10）= 34.5V

电流采样的电路原理图见图 6-2，对电流采样本质上是对检流电阻的电压进行采样。

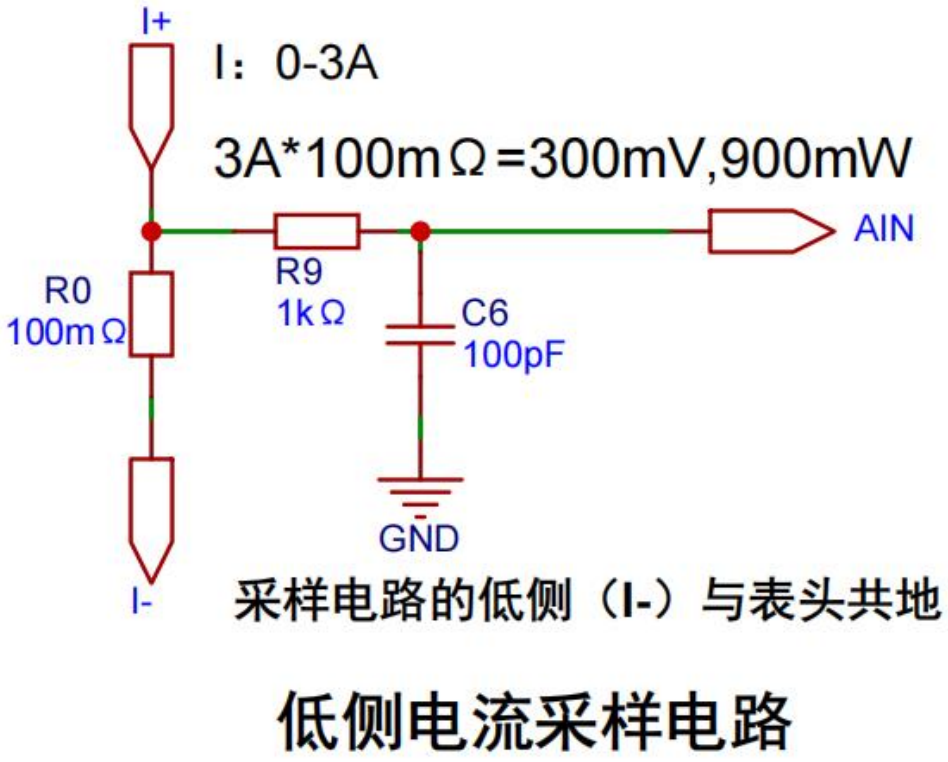

图 6-2 电流采样原理图

6.2. ADC 优点

1. 数字信号具有良好的抗干扰性。数字信号是由一系列离散的数字表示，因此可以抵抗模拟信号受到的各种干扰，如噪声、漂移等。

2. 方便数字信号的存储、处理和传输。由于数字信号是离散的，因此它们可以轻松存储在计算机内存或其他数字设备中，方便进行处理和传输。

3. 具有可编程性。现代的 ADC 出现了很多可编程的功能，例如可编程增益、采样率和滤波器等，可以根据不同的应用场景进行优化。

4. 适用性广泛。ADC 被广泛应用于工业、通信、医疗、电子测量、音频、视频等领域，可转换各种不同类型的模拟信号，包括电压、电流、声音、光信号等。

6.3. ADC 应用

ADC 的应用非常广泛。例如，我们可以用 ADC 将传感器的模拟信号转换为数字信号，然后通过计算机进行处理和分析；ADC 在音频处理中也起着重要的作用，将模拟声音信号转换为数字信号，并接下来进行数字信号处理；无线电通信中的信号调制也需要使用 ADC 等。总的来说，ADC 在现代电子工程中非常重要，是数字信号处理和控制技术的关键部分。

6.4. ADC 采样显示

在下面我们对 CW32F003 的 ADC 通道进行配置，输入 5V 电压给电压表，CW32 将采样得到的值输入数码管显示，对 ADC 通道的配置代码如下：

```
C
#include "ADC.h"

uint16_t Volt_Buffer;        //存放 ADC 采样值

void ADC_init(void)
{
    ADC_InitTypeDef     ADC_InitStructure;         //ADC 配置结构体
    ADC_SerialChTypeDef ADC_SerialChStructure;     //ADC 序列通道结构体
    GPIO_InitTypeDef    GPIO_Init_Struct;

    __RCC_GPIOB_CLK_ENABLE(); //打开 ADC 对应引脚时钟
    __RCC_ADC_CLK_ENABLE();   // 打开 ADC 时钟

    GPIO_Init_Struct.IT   = GPIO_IT_NONE;
    GPIO_Init_Struct.Mode = GPIO_MODE_ANALOG;//将 GPIO 的模式配置成模拟功能
    GPIO_Init_Struct.Pins = GPIO_PIN_1;      // PB01 是电压采集引脚
    GPIO_Init(CW_GPIOB, &GPIO_Init_Struct);
    PB01_ANALOG_ENABLE();                   //使能模拟引脚

    ADC_StructInit(&ADC_InitStructure);      // ADC 默认值初始化
    ADC_InitStructure.ADC_ClkDiv = ADC_Clk_Div4; //ADC 工作时钟配置 PCLK/4 = 6/4 = 1.5Mhz

/*信号电压较低时，可以降低参考电压来提高分辨率。 改变参考电压后，同样二进制表示的电压值就会不一样，
  最大的二进制（全 1）表示的就是你的参考电压，在计算实际电压时，就需要将参考电压考虑进去。*/
    ADC_InitStructure.ADC_VrefSel    = ADC_Vref_BGR1p5;//参考电压设置为 1.5V
    //由于电压信号为慢速信号，ADC 采样时间为十个 ADC 采样周期以确保准确
    ADC_InitStructure.ADC_SampleTime = ADC_SampTime10Clk;
    //Sqr 为序列配置寄存器，这里只用到了序列 0 的通道，所以配置成 0 表示只
```

```
转换 Sqr0 序列
    ADC_SerialChStructure.ADC_SqrEns     = ADC_SqrEns0;
    ADC_SerialChStructure.ADC_Sqr0Chmux  = ADC_SqrCh9;//配置 ADC 序
列，PB01 是 ADC 的第 9 通道
    ADC_SerialChStructure.ADC_InitStruct = ADC_InitStructure;
//ADC 初始化

    ADC_SerialChContinuousModeCfg(&ADC_SerialChStructure);   //ADC
序列连续转换模式配置
    ADC_ClearITPendingAll();           //清除 ADC 所有中断状态
    ADC_Enable();                     // ADC 使能
    ADC_SoftwareStartConvCmd(ENABLE);  //ADC 转换软件启动命令
}

void Get_ADC_Value(void)               //取得 ADC 采样的值传给全局变量
Volt_Buffer
{
    ADC_GetSqr0Result(&Volt_Buffer);
}
```

在主函数中初始化 ADC 后在 BTIM1 的中断服务程序中调用 Get_ADC_Value 得到 ADC 采样的值，再在主函数的 while 循环中调用数码管显示函数 Display 将 ADC 采样值显示到数码管上。下图为数字电压电流表接入 5V 电压时的采样显示图。可以看到接入 5V 时 ADC 采样得到 669，我们可以计算：

$$(669/4096)* [(1.5/10)*(200+10)] = 5.145\ V$$

其中 4096 代表 CW32 的 ADC 采样精度 12 位为 2^12=4096，由于我们的测试样品中 220KΩ的电阻被替换成了 200KΩ，所以计算公式如上，与万用表测量数值相符。（各位学员最终收到的版本是 220KΩ的电阻）

图 6-3 ADC 采样显示

图 6-4 万用表测量 5V

6.5. ADC 采样计算

根据上文，ADC 所采样的值虽然准确地显示在数码管上，但采样值仍需要转换成标准值。计算思路与上述公式类似，只是显示到数码管上需要将数值扩大 100 倍。因此采样计算的思路为：将采样得到的值（比如在 5V 输入的情况下 ADC 采样得到 668）用上述计算公式计算得到的结果后乘以 100：

$$(668/4096)* [(1.5/10)*(200+10)] * 100 = 513.7\ V$$

由于变量为整形，最终输入给显示函数 Display 的值为 513，在 Display 函数里对输入的值进行判定，如果输入值大于 1000，则数码管只能显示 xx.x V，所以我们只取输入值的千百十位；如果输入值小于 1000，比如现在输入值为 513，则数码管可显示 x.xx V,分别将 513 的百十个位存入 Seg_Reg 数组中。

最终需要添加一个 Cal_Buffer 变量来存储 Volt_Buffer 的值、一个电压计算函数，再修改 Display 函数见下文：

```C
uint16_t Cal_Buffer;  //存储 Volt_Buffer 的值

#define ADC_REF_VALUE (1500) //扩大 1000 倍  1.5 * 1000 = 1500
#define R2           (200) //单位：KΩ
#define R1           (10)

void Volt_Cal(void)   //将 ADC 采样值转化为标准值
{
  Cal_Buffer = Volt_Buffer;   //存储中断服务程序中取得的 ADC 采样值
  Cal_Buffer = (Cal_Buffer * ADC_REF_VALUE >> 12) * (R2 +
R1)/R1;//计算得到的值为标准值的 1000 倍

    if(Cal_Buffer % 10 >= 5)     // 四舍五入
    {
        Cal_Buffer = Cal_Buffer / 10 + 1;
    }
    else
    {
        Cal_Buffer = Cal_Buffer / 10;   //此时的值为标准值的 100 倍
    }
}
```

在 while 循环中调用数码管显示函数 Display 之前先调用 Volt_Cal 函数。

```C
int main()
{
  RCC_Configuration();
  Seg_Init();
  Btim1_Init();
  ADC_init();

  while(1)
    {
      Volt_Cal();
      Display(Cal_Buffer);
    }
}
```

Display 函数的更新如下：

```C
```

```
void Display(uint32_t value)
{
  uint8_t Thousands;   //千位
  uint8_t Hundreds;    //百位
  uint8_t Tens;        //十位
  uint8_t Units;       //个位

  Thousands = value / 1000;     //如果输入值大于 1000，只取输入值的千百十位
  if(Thousands > 0)             //大于 0 则说明输入值的千位有值
    {
       Units     = value % 10;
       value     = Units > 5 ? (value + 10) : value; // 根据后一位四舍五入
       Thousands = value / 1000 % 10;               //只取千百十位
       Hundreds  = value / 100 % 10;
       Tens      = value / 10 % 10;

       // 显示 xx.x 伏
       Seg_Reg[0] = Thousands;
       Seg_Reg[1] = Hundreds + 10; // 加 dp 显示
       Seg_Reg[2] = Tens;
    }

  else                          //如果输入值的千位没有值，则取百十个位
    {
        Units     = value % 10;
        Tens      = value / 10 % 10;
        Hundreds  = value / 100 % 10;

        // 显示 x.xx 伏
        Seg_Reg[0] = Hundreds + 10; // 加 dp 显示
        Seg_Reg[1] = Tens;
        Seg_Reg[2] = Units;
    }
}
```

最终显示效果如下图（输入接 5V）：

图 6-5 采样计算后显示值

此时万用表测得电压如下：

图 6-6 万用表测量值（5V）

七、实验六：串口蓝牙发送

7.1.串口基础知识

7.1.1.串口介绍

串行接口简称串口，也称串行通信接口或串行通信接口（通常指 COM 接口），是采用串行通信方式的扩展接口。串行接口 （Serial Interface）是指数据一位一位地顺序传送。其特点是通信线路简单，只要一对传输线就可以实现双向通信（可以直接利用电话线作为传输线），从而大大降低了成本，特别适用于远距离通信，但传送速度较慢。

7.1.2.串口通信参数介绍

- 波特率： 衡量通信速度的参数，它表示每秒钟传送的 bit 的个数。
- 数据位： 衡量通信中实际数据位的参数，表示一个信息包里包含的数据位的个数。
- 停止位： 用于表示单个信息包的最后位，典型值为 1、1.5 和 2 位。由于数据是在传输线上传输的，每个设备都有自己的时钟，很有可能在通信过程中出现不同步，停止位不仅仅表示传输的结束，还能提供校正时钟同步的机会。停止位的位数越多，不同时钟同步的容忍程度越大，但是数据传输率也越慢。
- 奇偶检验位： 表示一种简单的检查错误的方式。

关于更为详细的介绍请搜索百度。

7.1.3.串口工作模式

串口可以工作在单工、半双工和全双工模式下。

- 单工：在通信的任意时刻，信息只能由 A 传到 B。
- 半双工：在通信的任意时刻，信息即可由 A 传到 B，又能由 B 传到 A，但同时只能有一个方向上的传输存在。
- 全双工：在通信的任意时刻，通信线路上存在 A 到 B 和 B 到 A 的双向信号传输。

7.1.4.串口通信协议

串口在进行通信的时候会按照数据包的形式进行发送，帧格式如图 1-4-1 所示。

图 19-2. USART 字符帧（8 数据位和 1 停止位）

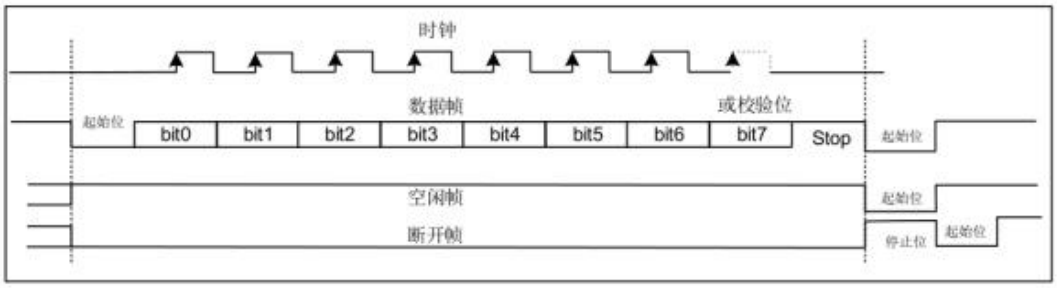

图 9-1 串口通信协议

串口通信是一位一位地传输，每传输一个字节总是以起始位开始，以停止位结束，字符之间没有固定的时间间隔要求。每一个字符的前面都有一位起始位（低电平），后面由 8 位数据位组成，如果开启了校验位，则最后一位数据位是校验位，最后是停止位。停止位后面是不定长的空闲位，停止位和空闲位都规定为高电平。

7.2.串口蓝牙接线原理图

在数字电压电流表上默认使用的串口是串口 2，接口为 CW_UART2_TX 和 CW_UART2_RX。关于串口和蓝牙连接原理图如图 9-2 所示。

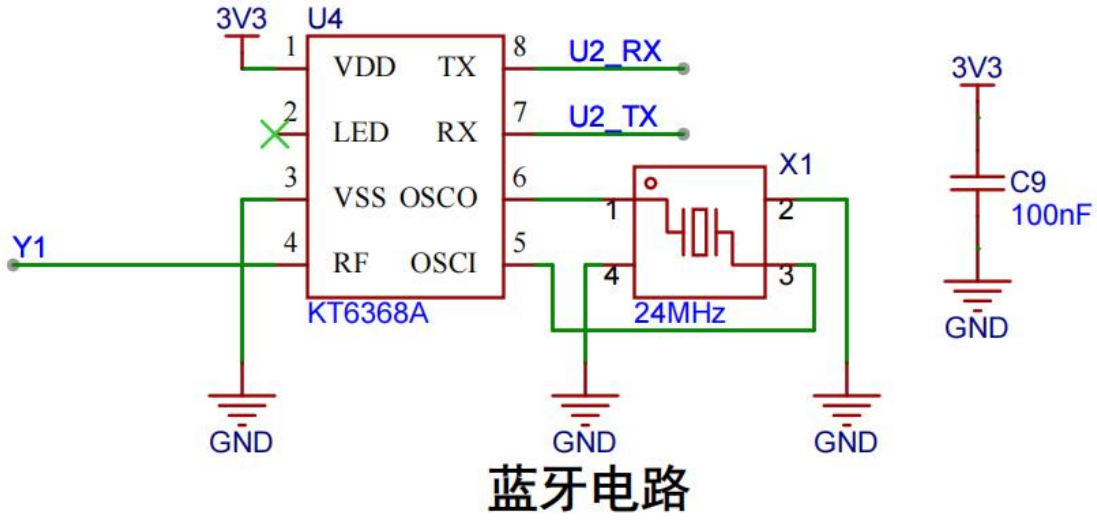

图 9-2 串口蓝牙接线

使用蓝牙模块 KT6368 将信息发送出去，只需要调用 CW32 的 UART_2 将信息传输至蓝牙模块即可。

7.3.串口驱动流程

串口驱动有以下几个流程：配置 CW32 时钟树，配置 UART_2 串口，调用串口打印信息。通过查看上文的 CW32 时钟树可知，串口 2 挂载在高级外设时钟 PCLK 上，而此前的时钟配置已将 PCLK 配置成了 6MHz。具体的配置函数如下：

```
C
void Uart2_Init(void)
{
    //外设时钟使能
    RCC_AHBPeriphClk_Enable(RCC_AHB_PERIPH_GPIOA, ENABLE);      //
使能串口要用到的 GPIO 时钟
    RCC_APBPeriphClk_Enable1(RCC_APB1_PERIPH_UART2, ENABLE);    //
使能串口时钟

    GPIO_InitTypeDef GPIO_InitStructure;
//GPIO 初始化
    GPIO_InitStructure.IT = GPIO_IT_NONE;
    GPIO_InitStructure.Pins =  GPIO_PIN_7;
    GPIO_InitStructure.Mode = GPIO_MODE_INPUT_PULLUP;
    GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
    GPIO_Init(CW_GPIOA, &GPIO_InitStructure);
    PA07_AFx_UART2RXD();

    GPIO_InitStructure.IT = GPIO_IT_NONE;
    GPIO_InitStructure.Pins = GPIO_PIN_6;
    GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
    GPIO_Init(CW_GPIOA, &GPIO_InitStructure);
    PA06_AFx_UART2TXD();

    USART_InitTypeDef USART_InitStructure;                      //串
口初始化
    USART_InitStructure.USART_BaudRate = 115200;                //
设置波特率
    USART_InitStructure.USART_Over = USART_Over_16;             //
串口采样方式
    USART_InitStructure.USART_Source = USART_Source_PCLK;       //
串口传输时钟源采用 PCLK
    USART_InitStructure.USART_UclkFreq =
RCC_Sysctrl_GetPClkFreq();  //频率为 PCLK 的频率
    USART_InitStructure.USART_StartBit = USART_StartBit_FE;     //
数据开始位
```

```
    USART_InitStructure.USART_StopBits = USART_StopBits_1;       //
数据停止位
    USART_InitStructure.USART_Parity = USART_Parity_No ;         //
无校验
    USART_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;//无硬件流控
    USART_InitStructure.USART_Mode = USART_Mode_Rx |
USART_Mode_Tx;  //发送/接收使能
    USART_Init(CW_UART2, &USART_InitStructure);
}
```

在完成串口的配置后，还需要写一个串口发送函数

```
C
void USART_SendString(UART_TypeDef *USARTx, char *String)
{
    while (*String != '\0')     //\0 表示没有数据
    {
        USART_SendData_8bit(USARTx, *String);   //发送一位 8bit 数据
        while (USART_GetFlagStatus(USARTx, USART_FLAG_TXE) ==
RESET);  //发送缓冲器未空则等待
        String++;              //发送一次完成后，准备发送下一位数据
    }
    while (USART_GetFlagStatus(USARTx, USART_FLAG_TXBUSY) == SET);
//发送串口状态忙则等待
}
```

由于定时器中断为 1ms，而串口发送占用的时间较长，所以我们 1000ms 使用蓝牙发送一次信息，这里的写法并未使用定时器。

```
C
char data_reg[24];    //定义数组，用于数据打印
uint32_t Ble_Time=0;  //1000ms 计时变量
while(1)              //main 函数里的 while 循环
    {
      if(GetTick() >= (Ble_Time + 1000))     //如果此时的时间大于上一
次的 1000ms
        {
          Ble_Time = GetTick();              //记录此刻时间
          Volt_Cal();                        //电压计算
          sprintf(data_reg,"volt=%u\r\n",Cal_Buffer);  //打印数据
          USART_SendString(CW_UART2,data_reg);         //调用串口上传
```

```
数据给蓝牙
        }
    }
```

注意 GetTick() 函数定义在头文件 cw32f003_systick.h 中，而此函数要使用到系统时钟中断，所以还需要配置系统时钟中断：

```
C
InitTick(48000000);                    // SYSTICK 的工作频率为 48MHz，每
ms 中断一次
```

最终可以收到蓝牙传递的数据：

图 9-3 蓝牙发送数据

八、实验七：基本数据处理算法（均值滤波）

均值滤波也称为线性滤波，其采用的主要方法为邻域平均法。线性滤波的基本原理是用均值代替原图像中的各个像素值，即对待处理的当前像素点（x，y），选择一个模板，该模板由其近邻的若干像素组成，求模板中所有像素的均值，再把该均值赋予当前像素点（x，y），作为处理后图像在该点上的灰度 g（x，y），即 g（x，y）=∑f（x，y）/m，m 为该模板中包含当前像素在内的像素总个数。

这本是数字图像处理的一种方法，但也可以用在我们数字电压电流表的 ADC 采样数据上。我们选取二十次的 ADC 采样值存储在数组 Volt_Buffer 中，然后去除掉数组中的最大值和最小值后再取平均，得到的值作为结果显示在数码管上，这样可以较大程度获得准确的、不易波动的数据。

程序在实验五的基础上略作修改即可，首先是增加和修改变量：

```
C
#define ADC_SAMPLE_SIZE (20)       //规定采样 20 个数据用来滤波
uint16_t Volt_Buffer[ADC_SAMPLE_SIZE]; //存储 ADC 转换值
uint32_t Led_Dis_Time;             //计数，300ms 改变一次数码管显示值
```

接下来是均值滤波的主体函数：

```
C
uint32_t Mean_Value_Filter(uint16_t *value, uint32_t size)     //均值滤波
{
    uint32_t sum = 0;         //ADC 采样数据和
    uint16_t max = 0;
    uint16_t min = 0xffff;    //min 初值取最大是为了将第一个数据记录
    int      i;

    for(i = 0; i < size; i++)
    {
        sum += value[i];
        if(value[i] > max)
        {
            max = value[i];
        }
        if(value[i] < min)
        {
            min = value[i];
        }
    }
    sum -= max + min;       //去除最大最小值
    sum  = sum / (size - 2);
    return sum;
}
```

对之前的电压计算函数 Volt_Cal() 修改如下：

```
C
void Volt_Cal(void)
{
  Cal_Buffer = Mean_Value_Filter(Volt_Buffer,ADC_SAMPLE_SIZE);
  Cal_Buffer = (Cal_Buffer * ADC_REF_VALUE >> 12) * (R2 + R1)/R1;
        // 四舍五入
```

```
  if(Cal_Buffer % 10 >= 5)
  {
    Cal_Buffer = Cal_Buffer / 10 + 1;
  }
  else
  {
    Cal_Buffer = Cal_Buffer / 10;
  }
}
```

在主函数的 while 循环里每隔 300ms 刷新一次：

```
C
while(1)
  {
    if(GetTick() >= (Led_Dis_Time + 300))
      {
        Led_Dis_Time = GetTick();
        Volt_Cal();
        Display(Cal_Buffer);
      }
  }
```

在之前未加滤波函数时，数码管上显示的电压数据是不稳定、跳变的，而加了滤波函数之后，数码管显示的电压数据可以稳定下来，并且有一定的抗干扰能力。至于电压准确性的问题，在后续章节的数据标定和校准中说明。