第 5 章

LED 流水灯与 SysTick 定时器

本章要点

- GPIO 输出库函数；
- LED 流水灯控制；
- SysTick 定时器；
- C 语言中的位运算。

第 4 章介绍了 STM32 的 GPIO 并给出通过操作 GPIO 寄存器的 LED 闪烁程序，使读者对 STM32 程序设计有一定的了解。本章将首先介绍常用的 HAL 库输出函数，随后完成嵌入式系统开发的经典案例——LED 流水灯项目，其中延时实现有两种方法，一种是软件延时，另一种是基于 SysTick 的中断定时。

微课视频

5.1 GPIO 输出库函数

由第 4 章分析可知，要实现 LED 流水灯项目，需要配置 PF 端口的工作模式，并设置 PF0～PF7 的电平状态。现将涉及的 HAL 库函数一一讲解，因为这是本书第一次介绍 HAL 库函数，所以讲解较为详尽。

5.1.1 GPIO 外设时钟使能

要使用 STM32 微控制器控制某一外设，首先就必须打开其外设时钟，HAL 库外设时钟使能和失能是通过一组宏定义完成的，函数前缀为__。

(1) 使能 GPIOF 外设时钟函数：__HAL_RCC_GPIOF_CLK_ENABLE()。

(2) 失能 GPIOF 外设时钟函数：__HAL_RCC_GPIOF_CLK_DISABLE()。

对于 GPIOA、GPIOB、GPIOC 等其他端口外设时钟的使能和失能方式可以类推得到，在此不一一列举。

5.1.2 函数 HAL_GPIO_Init()

表 5-1 描述了函数 HAL_GPIO_Init()。

表 5-1 函数 HAL_GPIO_Init()

函数名	HAL_GPIO_Init()
函数原型	HAL_GPIO_Init(GPIO_TypeDef *GPIO*x*,GPIO_InitTypeDef *GPIO_Init)
功能描述	根据 GPIO_Init 结构体中指定的参数初始化外设 GPIOx 寄存器
输入参数 1	GPIO*x*：*x* 可以是 A～I 中的一个，用来选择 GPIO 外设

续表

输入参数 2	GPIO_Init：指向结构体 GPIO_InitTypeDef 的指针，包含了外设 GPIO 的配置信息。参阅 Section：GPIO_InitTypeDef，查阅更多该参数允许的取值范围
输出参数	无
返回值	无

1. GPIO_InitTypeDef structure

GPIO_InitTypeDef 定义于文件 stm32f4xx_hal_gpio.h，代码如下：

```
typedef struct
{
    uint32_t Pin;
    uint32_t Mode;
    uint32_t Pull;
    uint32_t Speed;
    uint32_t Alternate;
}GPIO_InitTypeDef;
```

2. Pin 参数

Pin 参数选择待设置的 GPIO 引脚，使用操作符|可以一次选中多个引脚。Pin 参数值可以使用表 5-2 中的任意组合。

表 5-2　Pin 参数值

Pin	描　述	Pin	描　述
GPIO_PIN_0	选中引脚 0	GPIO_PIN_9	选中引脚 9
GPIO_PIN_1	选中引脚 1	GPIO_PIN_10	选中引脚 10
GPIO_PIN_2	选中引脚 2	GPIO_PIN_11	选中引脚 11
GPIO_PIN_3	选中引脚 3	GPIO_PIN_12	选中引脚 12
GPIO_PIN_4	选中引脚 4	GPIO_PIN_13	选中引脚 13
GPIO_PIN_5	选中引脚 5	GPIO_PIN_14	选中引脚 14
GPIO_PIN_6	选中引脚 6	GPIO_PIN_15	选中引脚 15
GPIO_PIN_7	选中引脚 7	GPIO_PIN_All	选中全部引脚
GPIO_PIN_8	选中引脚 8		

3. Mode 参数

Mode 参数用于设置选中引脚的工作模式。表 5-3 给出了其可取值。

表 5-3　Mode 参数值

Mode	描　述	Mode	描　述
GPIO_MODE_INPUT	输入浮空模式	GPIO_MODE_IT_RISING	中断上升沿触发
GPIO_MODE_OUTPUT_PP	推挽输出模式	GPIO_MODE_IT_FALLING	中断下降沿触发
GPIO_MODE_OUTPUT_OD	开漏输出模式	GPIO_MODE_IT_RISING_FALLING	中断上、下边沿
GPIO_MODE_AF_PP	复用推挽模式	GPIO_MODE_EVT_RISING	事件上升沿触发
GPIO_MODE_AF_OD	复用开漏模式	GPIO_MODE_EVT_FALLING	事件下降沿触发
GPIO_MODE_ANALOG	模拟信号模式	GPIO_MODE_EVT_RISING_FALLING	事件上、下边沿

4. Pull 参数

Pull 参数用于设置是否使用内部上拉或下拉电阻。表 5-4 给出了其可取值。

表 5-4 Pull 参数值

Pull	描 述
GPIO_NOPULL	无上拉或下拉
GPIO_PULLUP	使用上拉电阻
GPIO_PULLDOWN	使用下拉电阻

5. Speed 参数

Speed 参数用于设置选中引脚的速率。表 5-5 给出了其可取值，引脚实际速率需要参考产品数据手册。

表 5-5 Speed 参数值

Speed	描 述
GPIO_SPEED_FREQ_LOW	2MHz
GPIO_SPEED_FREQ_MEDIUM	12.5～50MHz
GPIO_SPEED_FREQ_HIGH	25～100MHz
GPIO_SPEED_FREQ_VERY_HIGH	50～200MHz

6. Alternate 参数

Alternate 参数定义引脚的复用功能，在文件 stm32f4xx_hal_gpio_ex.h 中定义了该参数的可用宏定义，这些复用功能的宏定义与具体的 MCU 型号有关，表 5-6 是其中的部分参数定义示例。

表 5-6 Alternate 参数值

Speed	描 述
GPIO_AF1_TIM1	TIM1 复用功能映射
GPIO_AF1_TIM2	TIM2 复用功能映射
GPIO_AF5_SPI1	SPI1 复用功能映射
GPIO_AF5_SPI2	SPI2/I2S2 复用功能映射
GPIO_AF7_USART1	USART1 复用功能映射
GPIO_AF7_USART2	USART2 复用功能映射

例：

```
/* Configure all the GPIOA in Input Floating mode */
GPIO_InitTypeDef GPIO_InitStruct = {0};
GPIO_InitStruct.Pin = GPIO_PIN_All;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

5.1.3 函数 HAL_GPIO_DeInit()

表 5-7 描述了函数 HAL_GPIO_DeInit()。

表 5-7 函数 HAL_GPIO_DeInit()

函数名	HAL_GPIO_DeInit()
函数原型	voidHAL_GPIO_DeInit(GPIO_TypeDef *GPIOx,uint32_t GPIO_Pin)
功能描述	反初始化 GPIO 外设寄存器，恢复为复位后的状态
输入参数 1	GPIOx：x 可以是 A～I 中的一个，用来选择 GPIO 外设

续表

输入参数 2	GPIO_Pin：指定反初始化端口引脚，取值参阅表 5-2
输出参数	无
返回值	无

例：

```
/* De-initializes the PA8 peripheral registers to their default reset values. */
HAL_GPIO_DeInit (GPIOA, GPIO_PIN_8);
```

5.1.4　函数 HAL_GPIO_WritePin()

表 5-8 描述了函数 HAL_GPIO_WritePin()。

表 5-8　函数 HAL_GPIO_WritePin()

函数名	HAL_GPIO_WritePin()
函数原型	voidHAL_GPIO_WritePin(GPIO_TypeDef * GPIOx,uint16_t GPIO_Pin,GPIO_PinState PinState)
功能描述	向指定引脚输出高电平或低电平
输入参数 1	GPIOx：x 可以是 A～I 中的一个，用来选择 GPIO 外设
输入参数 2	GPIO_Pin：指定输出端口引脚，取值参阅表 5-2
输入参数 3	PinState：写入电平状态，取值 GPIO_PIN_RESET 或 GPIO_PIN_SET
输出参数	无
返回值	无

例：

```
/* Set the GPIOA port pin 10 and pin 15 */
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10|GPIO_PIN_15, GPIO_PIN_SET);
```

5.1.5　函数 HAL_GPIO_TogglePin()

表 5-9 描述了函数 HAL_GPIO_TogglePin()。

表 5-9　函数 HAL_GPIO_TogglePin()

函数名	HAL_GPIO_TogglePin()
函数原型	voidHAL_GPIO_TogglePin(GPIO_TypeDef * GPIOx,uint16_t GPIO_Pin)
功能描述	翻转指定引脚的电平状态
输入参数 1	GPIOx：x 可以是 A～I 中的一个，用来选择 GPIO 外设
输入参数 2	GPIO_Pin：指定翻转端口引脚，取值参阅表 5-2
输出参数	无
返回值	无

例：

```
/* Toggles the GPIOA port pin 10 and pin 15 */
HAL_GPIO_TogglePin (GPIOA, GPIO_PIN_10|GPIO_PIN_15);
```

5.1.6　输出寄存器访问

HAL 库函数并没有提供访问 GPIO 端口输出数据寄存器的库函数，所以如果程序中需要读取或更新端口数据，可以采用直接访问端口数据寄存器 GPIOx_ODR 的方式来完成，其更加高效和快捷。

例：

```
/* Write data to GPIOA data port */
GPIOA->ODR = 0x1101;
```

5.2 LED流水灯控制

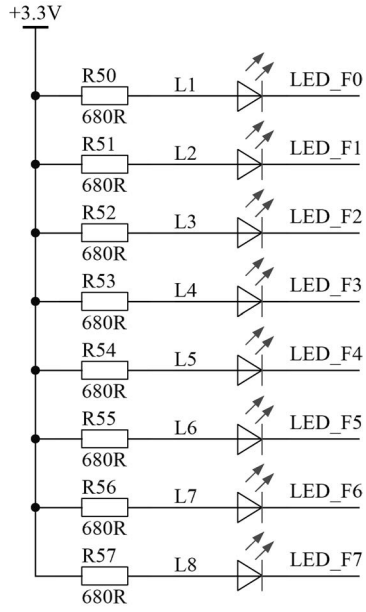

图 5-1 LED流水灯电路原理图

已知开发板LED流水灯原理图如图5-1所示。由图可知，如需实现LED流水灯控制只需要依次点亮L1～L8，即需依次设置PF0～PF7为低电平即可，对应GPIOF端口写入数据分别为0xFE、0xFD、0xFB、0XF7、0xEF、0xDF、0xBF、0x7F。

项目具体实施步骤为：

第1步：复制第3章创建的工程模板文件夹到桌面(也可以复制到其他路径，只是桌面操作更方便)，并将文件夹重命名为0501 LEDWater(其他名称完全可以，只是命名需要遵循一定原则，以便于项目积累)。

第2步：打开工程模板文件夹中的Template.ioc文件，启动STM32CubeMX配置软件，首先在引脚视图下面将PF0～PF7全部设置为GPIO_Output模式，然后选择System Core类别下的GPIO子项，LED控制引脚均设置为推挽、低速、无上拉/下拉、初始输出高电平，添加用户标签LED1～LED8，流水灯项目初始化配置如图5-2所示。时钟配置和工程配置选项无须修改，单击GENERATE CODE按钮生成初始化工程。

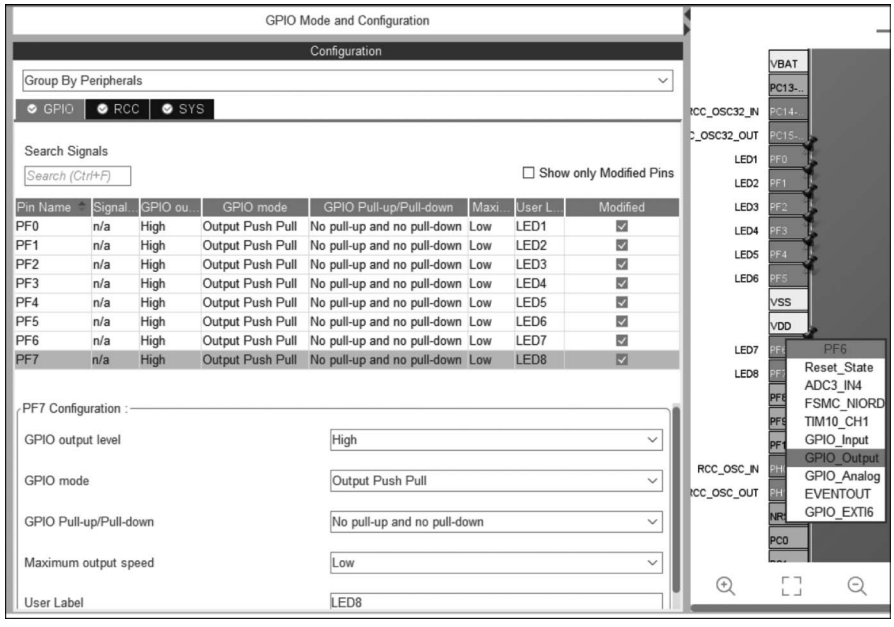

图 5-2 流水灯项目初始化配置

第3步：打开MDK-ARM文件夹下的工程文件Template.uvprojx，将生成工程编译一下，没有错误和警告就可以开始用户程序编写了。此时工程创建了一个gpio.c文件，将其添加到Application/User/Core项目组下面，生成的LED初始化程序就存放在该文件中，部分代码如下：

```
#include "gpio.h"
void MX_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    __HAL_RCC_GPIOF_CLK_ENABLE();
    HAL_GPIO_WritePin(GPIOF, LED1_Pin|LED2_Pin|LED3_Pin|LED4_Pin
        |LED5_Pin|LED6_Pin|LED7_Pin|LED8_Pin, GPIO_PIN_SET);
    GPIO_InitStruct.Pin = LED1_Pin|LED2_Pin|LED3_Pin|LED4_Pin
                        |LED5_Pin|LED6_Pin|LED7_Pin|LED8_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOF, &GPIO_InitStruct);
}
```

上述代码和 STM32CubeMX 配置选项一一对应，因为 LED 引脚既需要输出高电平，又需要输出低电平，所以需要将相应引脚配置为推挽输出模式。而对于输出速度并没有特殊要求，配置成 2MHz 即可。对于 LED 流水灯控制，需要初始化 GPIOF 的 PIN0～PIN7，由于在 STM32CubeMX 初始化配置时使用了标签，所以代码使用宏定义 LED1_Pin 替换 GPIO_PIN_0，其他引脚对应关系以此类推，相应的宏定义存放在 main.h 中，读者可以单击鼠标右键跟踪查看。

第 4 步：打开 main.c 文件，在程序沙箱内编写一个简单的延时程序，其代码很短，但是在嵌入式开发中是经常使用的，所以作者将该函数的声明放到 main.h 中，以便其他文件使用。在 main()函数中，需要完成系统初始化、时钟配置和 GPIO 初始化，上述代码均是由 STM32CubeMX 自动生成。用户只需定位 main()函数的 while 循环程序沙箱，编写流水灯显示程序，即先点亮一个 LED 灯，调用延时函数，等待约 1s 时间，再点亮下一个 LED 灯，如此往复。需要注意的是，用户编写的所有程序均需写在程序沙箱内，否则修改系统配置后再次生成工程时，用户代码将会丢失。main.c 文件的部分程序如下，为便于读者查看用户编写的代码，已将程序沙箱注释语句作加粗显示。

```
#include "LED.h"
#include "main.h"
#include "gpio.h"
void SystemClock_Config(void);

/* USER CODE BEGIN 0 */
void delay(uint32_t i)
{
while(i--);
}
/* USER CODE END 0 */

int main(void)
{
    HAL_Init();                    //系统初始化
    SystemClock_Config();          //系统时钟配置
    MX_GPIO_Init();                //GPIO 初始化
    /* USER CODE BEGIN WHILE */
    while (1)
    {
        GPIOF->ODR = 0xFE;
        delay(24000000);
        GPIOF->ODR = 0xFD;
        delay(24000000);
```

```
            GPIOF->ODR = 0xFB;
            delay(24000000);
            GPIOF->ODR = 0xF7;
            delay(24000000);
            GPIOF->ODR = 0xEF;
            delay(24000000);
            GPIOF->ODR = 0xDF;
            delay(24000000);
            GPIOF->ODR = 0xBF;
            delay(24000000);
            GPIOF->ODR = 0x7F;
            delay(24000000);
        /* USER CODE END WHILE */
        }
    }
```

第 5 步：编译工程，直到没有错误为止，下载程序到开发板，复位运行，检查实验效果。

5.3 SysTick 定时器

5.3.1 SysTick 定时器概述

以前大多操作系统需要一个硬件定时器产生操作系统需要的滴答中断，作为整个系统的时基。例如，为多个任务安排不同数目的时间片，确保没有一个任务能独占系统；或者把每个定时器周期的某个时间范围分配给特定的任务等。操作系统提供的各种定时功能，都与这个滴答定时器有关。因此，需要一个定时器来产生周期性的中断，而且最好是用户程序不能随意访问它的寄存器，以维持操作系统“心跳”的节律。

微课视频

ARM Cortex-M4 处理器内部包含一个简单的定时器。因为所有的 Cortex-M4 芯片都带有这个定时器，软件在不同 Cortex-M4 器件间的移植工作得以化简。该定时器的时钟源可以是内部时钟(FCLK，Cortex-M4 处理器上的自由运行时钟)，也可以是外部时钟(Cortex-M4 处理器上的 STCLK 信号)。不过，STCLK 的具体来源则由芯片设计者决定，因此不同产品之间的时钟频率可能会大不相同，需要查阅芯片的器件手册来决定选择什么作为时钟源。

SysTick 定时器能产生中断，Cortex-M4 专门为它开出一个异常类型，并且在向量表中有其一席之地。SysTick 定时器使操作系统和其他系统软件在 Cortex-M4 器件间的移植变得更简单了，因为在所有 Cortex-M4 产品间对其处理都是相同的。

SysTick 定时器除了能服务于操作系统之外，还能用于其他目的。如作为一个闹铃，用于测量时间等。要注意的是，当处理器在调试期间被喊停(halt)时，SysTick 定时器亦将暂停运作。

5.3.2 SysTick 定时器寄存器

有 4 个寄存器控制 SysTick 定时器，如表 5-10～表 5-13 所示。

表 5-10 SysTick 控制及状态寄存器 STK_CTRL(0xE000_E010)

位 段	名 称	类 型	复 位 值	描 述
16	COUNTFLAG	R	0	如果在上次读取本寄存器后，SysTick 已经数到 0，则该位为 1。如果读取该位，该位将自动清零
2	CLKSOURCE	R/W	0	0=外部时钟(STCLK)，AHB 时钟 8 分频 1=内部时钟(FCLK，HCLK)
1	TICKINT	R/W	0	1=SysTick 倒数到 0 时，产生 SysTick 异常请求 0=SysTick 数到 0 时，无动作
0	ENABLE	R/W	0	SysTick 定时器的使能位

表 5-11 SysTick 重装载数值寄存器 STK_LOAD(0xE000_E014)

位 段	名 称	类 型	复 位 值	描 述
23:0	RELOAD	R/W	0	当 SysTick 倒数至 0 时,将被重装载的值

表 5-12 SysTick 当前数值寄存器 STK_VAL(0xE000_E018)

位 段	名 称	类 型	复 位 值	描 述
23:0	CURRENT	R/Wc	0	读取时,则返回当前倒计数的值,写入时,则清零,同时还会清除在 SysTick 控制及状态寄存器中的 COUNTFLAG 标志

表 5-13 SysTick 校准数值寄存器 STK_CALIB(0xE000_E01C)

位 段	名 称	类 型	复 位 值	描 述
31	NOREF	R	—	1=没有外部参考时钟(STCLK 不可用) 0=外部参考时钟可用
30	SKEW	R	—	1=校准值不是准确的 10ms 0=校准值是准确的 10ms
23:0	TENMS	R/W	0	10ms 的时间内倒计数的格数。芯片设计者应该通过 Cortex-M4 的输入信号提供该数值。若该值读回零,则无法使用校准功能

5.3.3 延时函数 HAL_Delay()

在 HAL 库中提供了一个十分方便的利用 SysTick 定时器实现的延时函数 HAL_Delay(),为更好地理解和应用这一函数,下面对其实现过程进行详细讲解。

HAL_Delay()函数位于 stm32f4xx_hal.c 文件中,其原型为:

```
__weak void HAL_Delay(uint32_t Delay)
{
    uint32_t tickstart = HAL_GetTick();
    uint32_t wait = Delay;
    /* Add a freq to guarantee minimum wait */
    if (wait < HAL_MAX_DELAY) // HAL_MAX_DELAY = 0xFFFF FFFF
    {
        wait += (uint32_t)(uwTickFreq);
    }
    while((HAL_GetTick() - tickstart) < wait)
    {
    }
}
```

延时函数首先调用 HAL_GetTick()函数,并把它的返回值赋给变量 tickstart,同时把函数的形参(Delay,延时的毫秒数)赋给变量 wait,随后使用 if 语句判断 wait 的值是否小于 HAL_MAX_DELAY(宏定义值为 0xFFFF FFFF),如果是,则 wait 变量增加 uwTickFreq(枚举类型,延时为 1ms)。最后进入 while 循环中,while 循环执行的时间即为延时的时间。

继续打开 HAL_GetTick()函数,其代码比较简单,仅返回变量 uwTick 值。

```
__weak uint32_t HAL_GetTick(void)
{
    return uwTick;
}
```

由此可见，变量 uwTick 在延时实现中起关键作用，进一步观察与这一变量数值变化有关的函数，其中一个为 HAL_IncTick()函数，其代码也较为简单，只是将 uwTick 数值加 1。

```
__weak void HAL_IncTick(void)
{
    uwTick += uwTickFreq;
}
```

通过代码跟踪发现 HAL_IncTick()函数被 SysTick 中断服务程序 SysTick_Handler()调用，其位于 stm32f4xx_it.c 文件中，代码如下：

```
void SysTick_Handler(void)
{
    HAL_IncTick();
}
```

综上所述，HAL_Delay()延时函数实现原理为，定义一个无符号整型变量 uwTick，通过中断服务程序每 1ms 使其数值加 1。进入延时函数，首先记录下 uwTick 初始值，然后不断读取当前的 uwTick 变量值，如果二者差值小于延时数值，则一直等待，直至延时完成。

细心的读者可能会发现一个问题，那就是为什么 SysTick 定时器会每 1ms 中断一次，以及定时器的时钟源、中断优先级又是如何设置的？所以本节还要带领大家了解一下 SysTick 定时器的初始化过程。

在 main()主函数中调用的第一个函数是 HAL_Init()，其主要用于复位所有外设，初始化 Flash 接口和 SysTick 定时器，并将中断优先级分组设为分组 4。SysTick 定时器初始化是调用 HAL_InitTick(TICK_INT_PRIORITY)函数完成的，其入口参数 TICK_INT_PRIORITY 数值为 15，该数值为 SysTick 定时器的中断优先级。函数定义如下：

```
__weak HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
    /* Configure the SysTick to have interrupt in 1ms time basis */ //uwTickFreq = 1
    if (HAL_SYSTICK_Config(SystemCoreClock / (1000U / uwTickFreq)) > 0U)
    {  return HAL_ERROR; }
    /* Configure the SysTick IRQ priority */ //__NVIC_PRIO_BITS = 4
    if (TickPriority < (1UL << __NVIC_PRIO_BITS))
    {
        HAL_NVIC_SetPriority(SysTick_IRQn, TickPriority, 0U);
        uwTickPrio = TickPriority;
    }
    else
    {  return HAL_ERROR; }
    /* Return function status */
    return HAL_OK;
}
```

HAL_InitTick()函数主要实现两个功能：一是设置 SysTick 定时器的中断优先级，初始化后中断优先级的数值是 15，即最低优先级(数值越大级别越低)；二是设置定时器的分频系数，以使定时器中断频率为 1kHz，通过调用 HAL_SYSTICK_Config()函数实现，其入口参数为 SystemCoreClock/(1000U/uwTickFreq)，实为定时器相对于内核频率的分频系数，具体实现方法需要进一步查看其函数原型。代码如下：

```
uint32_t HAL_SYSTICK_Config(uint32_t TicksNumb)
{
```

```
    return SysTick_Config(TicksNumb);
}
```

在函数 HAL_SYSTICK_Config()调用了另一个 SysTick_Config(),其入口参数是同值传递的,继续打开 SysTick_Config(),其源代码如下:

```
__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks)
{
    if ((ticks - 1UL) > SysTick_LOAD_RELOAD_Msk)
    {  return (1UL);                                   /* Reload value impossible */  }
    SysTick->LOAD = (uint32_t)(ticks - 1UL);           /* set reload register */
    NVIC_SetPriority (SysTick_IRQn, (1UL << __NVIC_PRIO_BITS) - 1UL);
    SysTick->VAL = 0UL;                                /* Load the SysTick Counter Value */
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
                    SysTick_CTRL_TICKINT_Msk |
                    SysTick_CTRL_ENABLE_Msk;
    return (0UL);                                      /* Function successful */
}
```

为了便于大家更好地理解程序,现将函数参数和相关宏定义说明如下:

(1) ticks:函数入口参数,用于设定 SysTick 定时器中断频率。因为 SysTick 定时器当计数到 0 后再减 1,才将 STK_LOAD 装载到当前值寄存器 STK_VAL,开始下一个周期的计数,所以需要将 ticks 减去 1 之后赋给定时器的重装载寄存器 STK_LOAD。

分频系数由形式参数 ticks 决定,其数值由实际参数 SystemCoreClock/(1000U/uwTickFreq)计算得出。当系统内核频率为 SystemCoreClock=168MHz,uwTickFreq 取默认值 1,则定时器中断频率为:

$$\begin{aligned} f_{\text{Tick}} &= \text{SystemCoreClock}/(\text{SystemCoreClock}/(1000\text{U}/\text{uwTickFreq})) \\ &= 168\text{MHz}/(168\text{MHz}/(1000/1)) \\ &= 1\text{kHz} \end{aligned}$$

由上述计算过程可以看出,内核频率 SystemCoreClock 对定时器的中断频率没有影响,1000U/uwTickFreq 表达式即为最终频率,增加 uwTickFreq 数值将减小输出频率,事实上 HAL_Delay 延时函数还可以配置以 10ms 或 100ms 为单位的延时函数,但是不推荐使用,一般直接使用默认的毫秒延时即可。

(2) __NVIC_PRIO_BITS:中断优先级位数,宏定义,数值为 4,此处再次将 SysTick 定时器中断优先级配置为 15,即最低优先级。

(3) SysTick_LOAD_RELOAD_Msk:定时器最大重载值掩码,宏定义,数值为 0xFF FFFF,向定时器重装载寄存器,写入数据不得大于此数值。

(4) SysTick_CTRL_CLKSOURCE_Msk:定时器时钟来源控制位掩码,宏定义,数值为 0x0000 0004。由此可知 STK_CTRL 的 CLKSOURCE 位为 1,SysTick 定时器时钟源为内部时钟,即 FCLK 或 HCLK,二者数值上是相同的,最高频率 168MHz。

(5) SysTick_CTRL_TICKINT_Msk:判断定时器是否产生异常控制位掩码,宏定义,数值为 0x0000 0002。由此可知,STK_CTRL 的 TICKINT 位为 1,即定时器倒数到 0 时产生异常请求。

(6) SysTick_CTRL_ENABLE_Msk:定时器使能控制位掩码,宏定义,数值为 0x0000 0001,由此可知,STK_CTRL 的 ENABLE 位为 1,使能 SysTick 定时器。

SysTick_Config()函数是 SysTick 定时器初始化的核心函数,进入函数首先将分频系数减 1 后赋给定时器重装载寄存器,随后将 SysTick 定时器中断优先级设为最低,定时器当前值寄存器写入 0。最后配

置定时器控制及状态寄存器，选择内部时钟为时钟源、产生异常中断，并使能定时器。如果用户在编写程序时需要重新配置 SysTick 定时器也可以再次调用 SysTick_Config()函数进行相关设置。

HAL_Delay()延时函数虽然用来全不费工夫，只需要给出需要延时的毫秒数即可，但其实现却是众多配置、层层调用，较为复杂。

由上述分析可知，使用延时函数需要注意以下几点：

①SysTick 定时器使用内部时钟作为时钟源。②SysTick 定时器延时是阻塞运行的，延时过程独占 CPU，无法执行其他任务。③SysTick 定时器要想跳出延时函数，必须保证中断服务程序能够被执行，否则将会导致系统死机。

5.3.4 HAL_Delay()延时实例

前文 LED 流水灯控制程序中延时程序是通过软件延时的方法实现的，这个时间很不精确，只能大概估计。根据上述分析，本节利用 HAL_Delay()函数实现精确的延时，操作较为简单，只需要将原程序中的 delay(24000000)替换为 HAL_Delay(1000)即可。main()函数的参考代码如下：

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    /* USER CODE BEGIN WHILE */
    while (1)
    {
        GPIOF->ODR = 0xFE;          HAL_Delay(1000);
        GPIOF->ODR = 0xFD;          HAL_Delay(1000);
        GPIOF->ODR = 0xFB;          HAL_Delay(1000);
        GPIOF->ODR = 0xF7;          HAL_Delay(1000);
        GPIOF->ODR = 0xEF;          HAL_Delay(1000);
        GPIOF->ODR = 0xDF;          HAL_Delay(1000);
        GPIOF->ODR = 0xBF;          HAL_Delay(1000);
        GPIOF->ODR = 0x7F;          HAL_Delay(1000);
        /* USER CODE END WHILE */
    }
}
```

5.3.5 微秒级延时的实现

HAL_Delay()函数无须用户设计代码，在工程任何位置可以直接调用，十分方便。但它也存在一些缺点：一是使用中断方式，如果在其他中断服务程序中使用，则需重新配置中断优先级，否则将会导致死机；二是无法实现微秒级延时，因为配置的中断服务程序是 1ms 执行一次，只能进行毫秒级延时。本节将使用查询方式实现 SysTick 定时器微秒级的延时，这在某些传感器或数据通信中是经常需要使用的，其参考程序如下：

```
void delay_us(uint32_t nus)
{
    uint32_t ticks;
    uint32_t told,tnow,tcnt = 0;
    uint32_t reload = SysTick->LOAD + 1;                    //计数个数为重装载值加 1
    ticks = nus * (SystemCoreClock/1000000);                //需要的节拍数
    told = SysTick->VAL;                                    //初始计数器值
```

```
    while(1)
    {
        tnow = SysTick -> VAL;
        if(tnow!= told)
        {
            if(tnow < told) tcnt += told - tnow;          //SysTick 递减的计数器
            else tcnt += reload - tnow + told;
            told = tnow;
            if(tcnt >= ticks) break;                      //延时时间已到,退出
        }
    }
}
```

基于查询方式的微秒级延时函数设计思路为：进入函数记录定时器起始值，读取定时器当前值寄存器，如果和起始值不相同，则计算经历过的节拍数，当前值小于起始值，则直接累计，如果当前值大于起始值，则需要加上一个周期的计数个数(重装载值加1)再累计，如此循环，直至累计的节拍数大于或等于需要的节拍数，退出函数。

分析delay_us()函数的实现代码可知，函数使用重装载值参与节拍数的累加，但并没有更改寄存器的数值。根据系统变量SystemCoreClock计算1μs定时所需要的节拍数。上述两点使得延时函数可以应用于任意内核频率系统且和SysTick定时器重装载值无关。

使用delay_us(1000)可以实现1ms的延时，基于查询方式的毫秒延时函数就是延时若干1ms，其参考程序如下：

```
void delay_ms(uint32_t nms)
{
    uint32_t i;
    for(i = nms;i > 0;i -- )
        delay_us(1000) ;
}
```

5.3.6　综合延时程序实例

至此，本章共介绍了4种延时函数的实现方式，还是以LED流水灯为例进行项目实验，采用4种方式分别延时，参考代码如下：

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    /* USER CODE BEGIN WHILE */
    while (1)
    {
        GPIOF -> ODR = 0xFE;
        delay_us(1000000);              //SysTick 查询方式 μs 延时
        GPIOF -> ODR = 0xFD;
        delay_us(1000000);
        GPIOF -> ODR = 0xFB;
        delay_ms(1000);                 // SysTick 查询方式 ms 延时
        GPIOF -> ODR = 0xF7;
        delay_ms(1000);
```

```
            GPIOF->ODR = 0xEF;
            HAL_Delay(1000);                    // SysTick 中断方式 ms 延时
            GPIOF->ODR = 0xDF;
            HAL_Delay(1000);
            GPIOF->ODR = 0xBF;
            delay(24000000);                    //软件延时,空循环等待
            GPIOF->ODR = 0x7F;
            delay(24000000);
            /* USER CODE END WHILE */
        }
    }
```

需要说明的是上述代码中 4 种延时方式混合使用，只为测试方便，这种程序设计风格是不推荐的。测试表明 4 种延时函数可以交替使用，相互之间没有影响，除软件延时时间精度难以保证外，其余三种方式可以实现精确的延时。

这么多的延时方式应该如何选择呢？软件延时函数 delay()简单方便、易于实现，在短延时或不需要精确延时的场合使用更加高效。基于中断方式的 HAL_Delay()延时函数由官方提供，且已经初始化完成，在任何文件中都可以直接使用，在大部分场合，使用该函数实现毫秒级延时。如果要实现微秒级延时，则可以使用作者编写的 delay_us()延时函数，令人欣喜的是，该函数与官方延时函数共用系统初始化，可以交替使用。如果需要采用基于查询方式的毫秒级延时，则可以使用 delay_ms()延时函数。

为了便于读者在后续项目中使用上述延时函数，作者将自定义的 3 个延时函数声明到公共头文件 main.h 中，其他文件若需要使用延时函数仅需要将其包含即可，而这一操作在大部分情况下是系统自动完成的。

5.4 开发经验小结——C 语言中的位运算

C 语言既具有高级语言的特点，又具有低级语言的功能，因而具有广泛的用途和旺盛的生命力，其位运算功能就十分适合编写嵌入式硬件控制程序。

5.4.1 位运算符和位运算

所谓位运算是指进行二进制位的运算。C 语言提供如表 5-14 所列出的位运算符。

表 5-14 C 语言位运算符

运 算 符	含 义	运 算 符	含 义
&	位与	~	位取反
|	位或	≪	位左移
^	位异或	≫	位右移

说明：

(1) 位运算符中除~以外，均为二目运算符，即要求两侧各有一个运算量。

(2) 运算量只能是整型或字符型的数据，不能为实型数据。

下面对各运算符分别介绍如下：

1. 位与运算符(&)

位与运算是指参加运算的两个数据，按二进制位进行“与”运算。如果两个相应的二进制位都为 1，则该位的结果值为 1，否则为 0，即 0&0=0；0&1=0；1&0=0；1&1=1。

例如，两个 8 位无符号数 3 和 5，位与运算计算过程如下：

```
          3=0   0   0   0   0   0   1   1
(&)       5=0   0   0   0   0   1   0   1
-------------------------------------------
          0   0   0   0   0   0   0   1
```

计算时，首先将两个运算量分别转换为二进制数，然后按位相与，最后计算得到 3&5 的运算结果为 1。

2. 位或运算符(|)

位或运算的运算规则是，两个相应的二进制位中只要有一个为 1，该位的结果为 1，即：0|0=0；0|1=1；1|0=1；1|1=1。

例如，两个 8 位十六进制数 0x35 和 0xA8，位或运算过程如下：

```
          0x35 =0   0   1   1   0   1   0   1
(|)       0xA8=1   0   1   0   1   0   0   0
-------------------------------------------
          1   0   1   1   1   1   0   1
```

计算时，首先将两个运算量分别转换为二进制，然后按位相或，最后计算得到 0x35|0xA8 的运算结果为 0xBD。

3. 位异或运算符(^)

位异或运算符^也称为 XOR 运算符，位异或的运算规则是，若参与运算的两个二进制位相同，则结果为 0(假)；两个二进制位不相同，则结果为 1(真)，即 0^0=0；0^1=1；1^0=1；1^1=0。“异或”的意思是判断两个相应位的值是否为“异”，为“异”(值不同)就取真(1)，否则取假(0)。

例如，两个 8 位无符号数 57 和 42，位异或运算过程如下：

```
          57=0   0   1   1   1   0   0   1
(^)       42=0   0   1   0   1   0   1   0
-------------------------------------------
          0   0   0   1   0   0   1   1
```

计算时，首先将两个运算量分别转换为二进制数，然后按位相异或，最后得到 57^42 的运算结果 19。

4. 取反运算符(～)

取反运算符～是一个单目运算符，用来对一个二进制数按位取反，即将 0 变 1，1 变 0。例如，对于 8 位无符号数 0x01 进行取反的运算过程如下：

```
(～)      0x01=0   0   0   0   0   0   0   1
-------------------------------------------
          1   1   1   1   1   1   1   0
```

计算时，只需将待取反运算量转换为二进制，然后按位取反，由此可知～0x01 的结果为 0xFE，也就是十进制数 254。

5. 左移运算符(≪)

左移运算符≪用来将一个数的各二进制位全部左移若干位。例如 a≪2，表示将 a 的二进制数左移 2 位，右边空出的位补 0。若 a=15，即二进制数 00001111，左移 2 位得到 00111100，即十进制数 60(为简单起见，用 8 位二进制数表示十进制数 15，如果用 16 位二进制数表示，结果也是一样的)。高位左移后溢出，舍弃即可。

左移 1 位相当于该数乘以 2，左移 2 位相当于该数乘以 2^2=4。上面举的例子 15≪2=60，即乘以 4。但此结论只适用于该数左移时被溢出舍弃的高位中不包含 1 的情况。

6. 右移运算符(≫)

右移运算符≫用来将一个数的各二进制位全部右移若干位。例如 a≫2，表示将 a 的各二进制位右

移 2 位。移到右端的低位被舍弃，高位补 0。例如，$a=15$，对应的二进制数为 00001111，$a\gg2$ 结果为 00000011，最低两位 11 被移出舍弃。右移一位相当于除以 2，右移 n 位相当于除以 2^n。

实践可知，ARM Cortex-M 平台，使用 MDK 编译器，无论是左移还是右移，移出位均舍弃，移入位均补 0。

7. 位运算符与赋值运算符

位运算符与赋值运算符可以组成复合赋值运算符，分别为：&=，|=，^=，≫=，≪=。例如，a&=b 相当于 a=a&b，a≪=2 相当于 a=a≪2。

5.4.2 嵌入式系统位运算实例

C 语言的位运算可以实现嵌入式系统底层硬件位控制功能，但 C 语言并没有像汇编语言那样，具有 SETB、CLR、CPL 等单个位操作指令，而只能通过对整型数据的位运算实现对单个或多个二进制位的操作。下面给出几个应用实例，以期起到抛砖引玉的效果。

1. 对指定位取反

已知开发板 LED 流水灯电路如图 5-1 所示，现需要将中间 4 个 LED 灯(L3～L6)的状态取反。

分析上面介绍的各位运算操作的特点可以发现，异或运算规则中，与 0 相异或，保持原二进制位状态不变，与 1 相异或其状态取反。所以要实现本例功能，仅需将端口输出寄存器与二进制数 00111100 相异或即可，即中间四位取反，其余位不变，参考代码如下：

```
GPIOF->ODR = GPIOF->ODR^0x3C;
```

2. 流水灯移位实现

5.2 节流水灯控制程序还可以通过移位来实现，具体方法是，将常量 1 依次左移 i 位，$i=0\sim7$，然后将结果取反后送端口输出寄存器，其参考程序如下：

```
GPIOF->ODR = ~(1<<i);
```

3. 实现循环移位功能

在汇编语言中一般会提供循环左移和循环右移功能。在流水灯控制程序中，另外一种实现方法是设置一个初始状态，即点亮第一个 LED 灯，对应端口数据为 0xFE，之后循环移位即可。端口数据循环左移 i 位的参考代码如下：

```
uint8_t LedVal = 0xFE;
LedVal = LedVal<<i|LedVal>>(8-i);
GPIOF->ODR = LedVal;
```

上述代码实现的基本思想为，首先定义一个 8 位无符号变量并赋初值，然后将该变量先左移 i 位，再右移 $8-i$ 位，两者相位或，最后将结果输出到端口寄存器。

本章小结

本章首先介绍 HAL 库 GPIO 输出库函数，包括函数的功能、参数和应用方法，随后介绍了第一个基于 HAL 库的嵌入式开发实例，即 LED 流水灯控制，采用软件延时方式实现流水效果。最后介绍了 SysTick 定时器的功能、原理和控制寄存器，详细讲解了官方延时函数原理及实现方法，编写了基于查询方式的微秒级延时函数和毫秒级延时函数，并将上述延时函数添加到 LED 流水灯项目中，测试表明 4 种延时方法均可实现延时，可交替使用，相互之间无影响。

思考拓展

(1) 函数__HAL_RCC_GPIOA_CLK_ENABLE()的功能是什么？
(2) 函数 HAL_GPIO_Init()的功能是什么？有哪些参数？
(3) 函数 HAL_GPIO_WritePin()的功能是什么？有哪些参数？
(4) 函数 HAL_GPIO_TogglePin()的功能是什么？有哪些参数？
(5) 简要说明 SysTick 定时器的概况以及使用该定时器的好处。
(6) SysTick 定时器相关的控制寄存器有哪些？
(7) SysTick 定时器的时钟源是哪两类？如何设置？
(8) HAL_Delay()函数延时的单位是什么？最大延时时间是多少？
(9) delay_us()延时函数为什么不需要对 SysTick 定时器进行初始化？
(10) 通过位异或运算实现开发板 L1、L3、L5 以秒为周期闪烁程序设计。