第5章 点亮LED灯——固件库版

本章通过介绍使用固件库点亮 LED 灯的实验，着重分析 CMSIS 标准及库层次关系和如何构建库函数，从而从感性上清楚完成一个工程需要哪些具体操作和配置。

5.1 使用固件库点亮 LED 灯

5.1.1 新建工程模板——库函数版

建立一个空的工程文件，作为工程模板，以后各个外设工程都可以在此模板的基础上进行开发。

为了使工程目录更加清晰，在本地计算机上新建一个 05_Template 文件夹，并在它下面再新建 6 个文件夹，具体如表 5.1 和图 5.1 所示。

表 5.1 工程目录文件夹说明

文件夹名称	作　用
Doc	用来存放程序说明的文件，由写程序的人添加
Libraries	存放的是库文件
Listing	存放编译器编译时候产生的 C/汇编/链接的列表清单
Output	存放编译产生的调试信息、hex 文件、预览信息、封装库等
Project	用来存放工程
User	用户编写的驱动文件

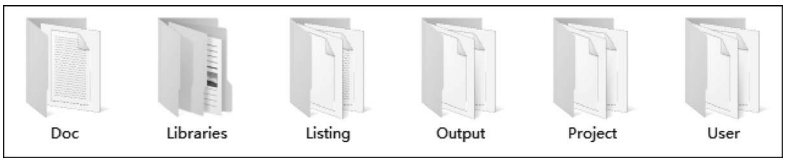


图 5.1 工程文件夹

在本地新建文件夹后，把准备好的库文件添加到相应的文件夹下，具体如下：

(1) 在 Doc 文件夹中新建 readme.txt 文件。

(2) 直接将固件库中的 Libraries 文件夹中的 CMSIS 文件夹和 STM32F10x_StdPeriph_Driver 文件夹复制到建好的 Libraries 文件夹中，其中 CMSIS 中存放 CM3(Cortex-M3)内核有关的库文件，STM32F10x_StdPeriph_Driver 中存放 STM32 外设库文件。

(3) Listing、Output、Project 文件夹均暂时为空。

(4) User 文件夹中存放 stm32f10x_conf.h、stm32f10x_it.h、stm32f10x_it.c 和 main.c 等文件，其中 stm32f10x_conf.h 文件用来配置库的头文件，stm32f10x_it.h 和 stm32f10x_it.c 用来存放中断相关的函数，main.c 就是 main 函数文件。

打开 Keil 5，新建一个工程，工程名为 Template(模板)，保存在 Project 文件夹下，如图 5.2 和图 5.3 所示。

1. 选择 CPU 型号

根据开发板使用的 CPU 具体的型号来选择，开发板选 STM32F103VE 型号，如图 5.4 所示。

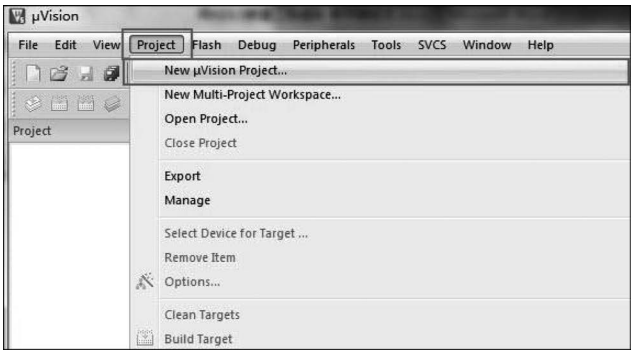


图 5.2　新建工程

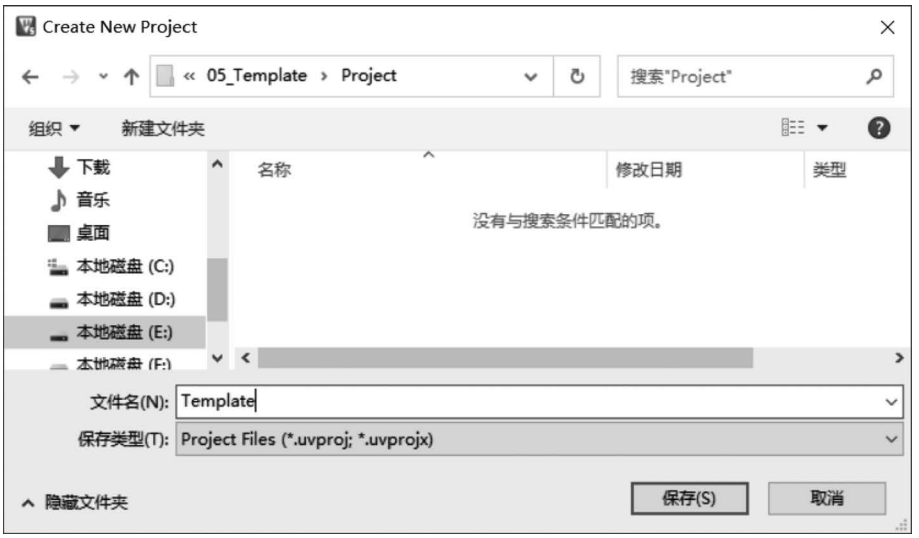


图 5.3　工程命名

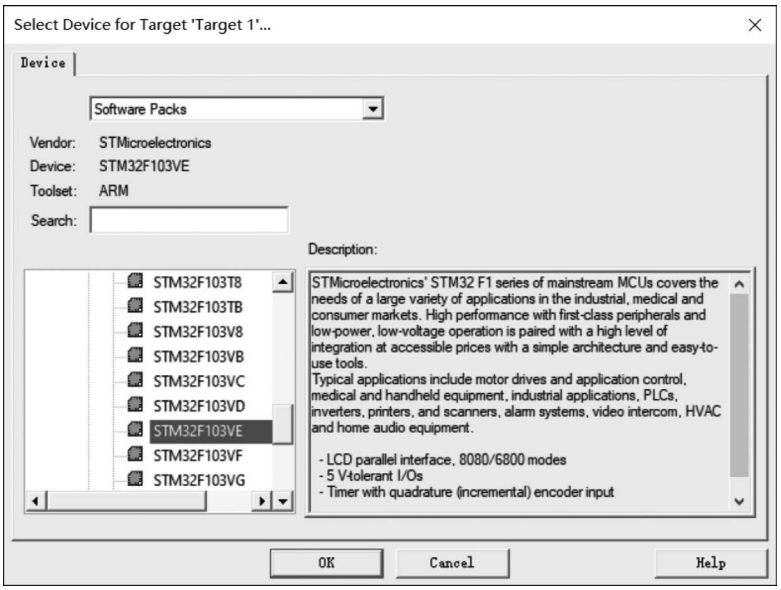


图 5.4　选择 CPU 型号

2. 在线添加库文件

手动添加库文件，单击“关闭”按钮，如图 5.5 所示。

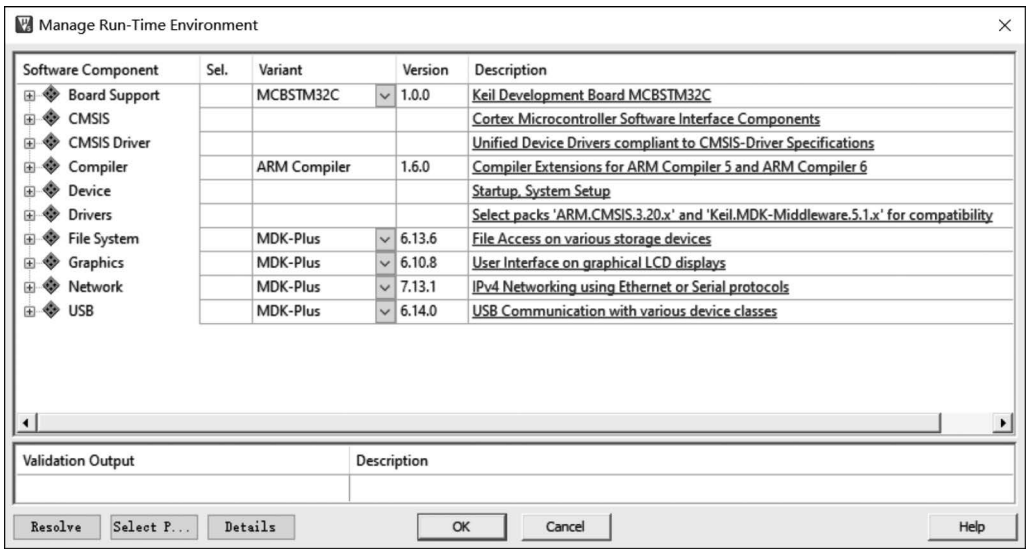


图 5.5　库文件管理

3. 添加组文件夹

在新建的工程中添加 5 个组文件夹，用来存放不同的文件。右击 Target1，然后选择 Add Group，将 New Group 重新命名为相应的文件夹名，如图 5.6 所示；或者右击 Target1，然后选择 Manage Project Items(这里选择此方法)，如图 5.7 所示。弹出图 5.8 所示对话框，单击“1”处，就可以在“2”处输入相应的文件组名。

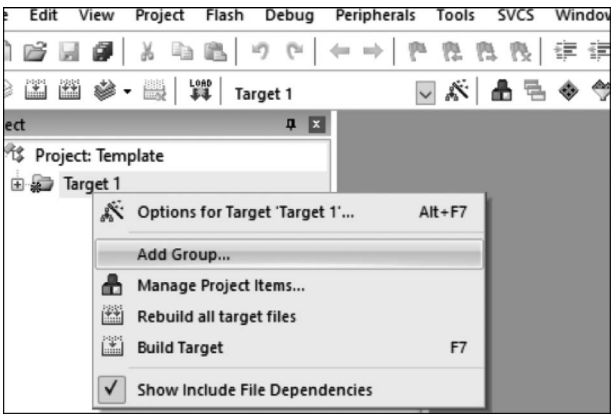


图 5.6　添加新的组文件 1

新建的 5 个组文件夹分别为 STARTUP、CMSIS、FWLB、USER、DOC，如图 5.9 所示。

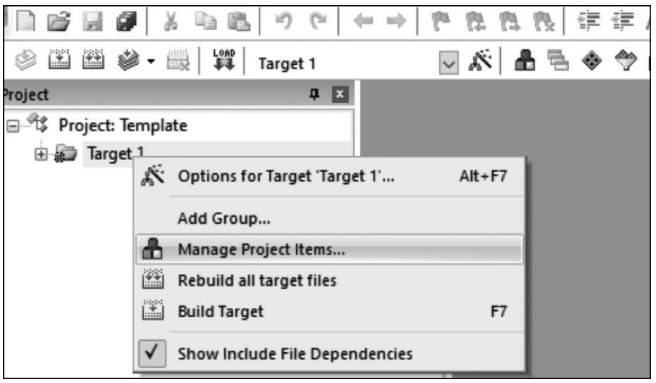


图 5.7　添加新的组文件 2

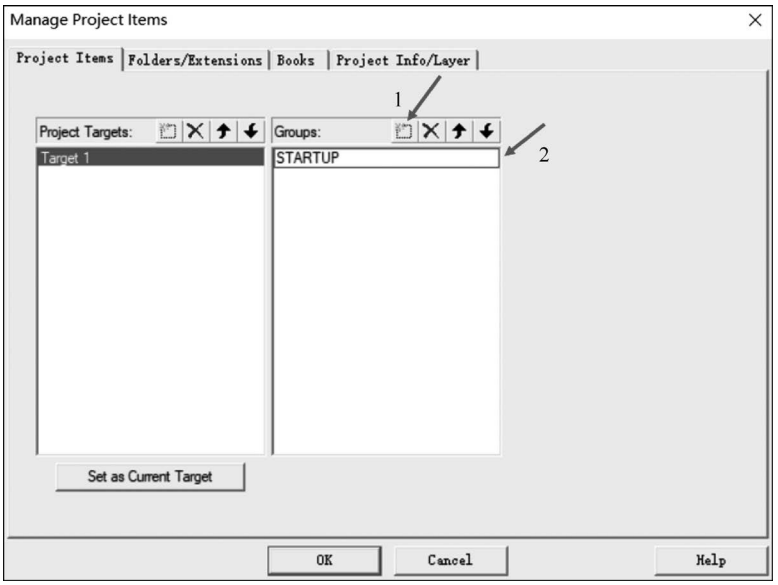


图 5.8　添加新的组文件 3

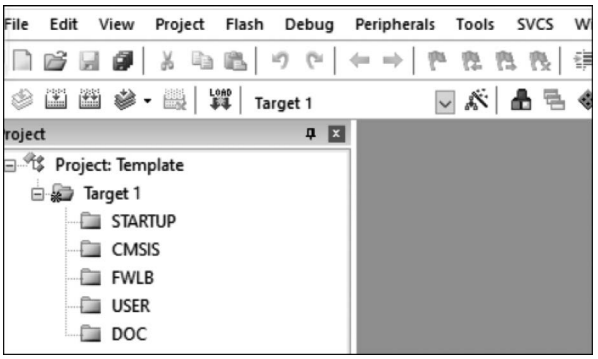


图 5.9　新建 5 个组文件夹

4. 添加文件

文件从本地建好的工程文件夹中获取，双击各组文件夹就会出现添加文件的路径，然后选择文件即可。各组文件夹内添加的文件如下：

(1) STARTUP，存放启动文件 startup_stm32f10x_hd.s。

(2) CMSIS，存放 core_cm.c、system_stm32f10x.c。

(3) FWLB，存放 STM32F10x_StdPeriph_Driver\src 文件夹下的全部 C 文件，即固件库。

(4) USER，存放用户编写的文件：main.c，main 函数文件，暂时为空；stm32f10x_it.c，与中断有关的函数都放这个文件，暂时为空。

(5) DOC，存放 readme.txt 程序说明文件，用于说明程序的功能和注意事项。

STARTUP 添加文件如图 5.10 所示。其他组文件夹添加文件的过程与 STARTUP 添加文件一致。

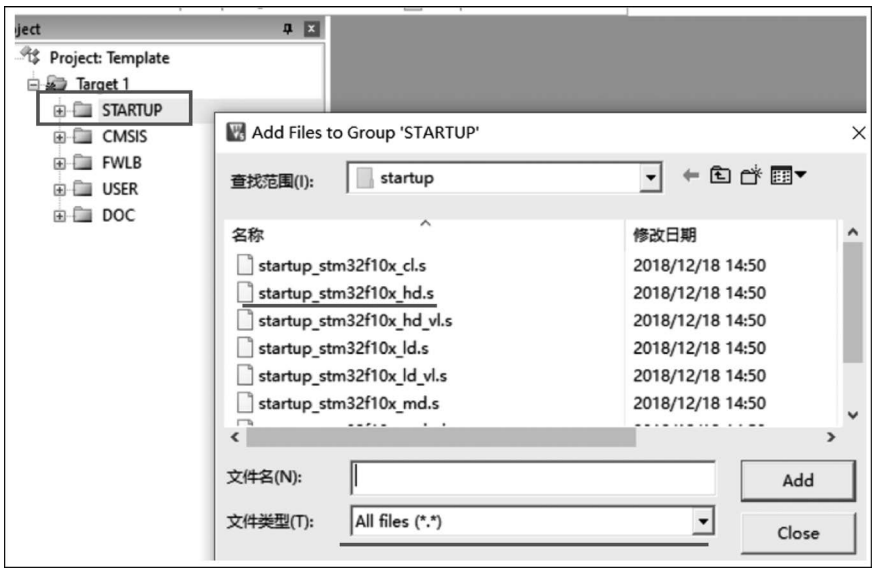


图 5.10 STARTUP 添加文件

5. 工程选项卡配置

单击魔术棒(Options for Target)，进入 Options for Target‘Target 1’对话框，如图 5.11 所示。

(1) 在 Target 选项卡中勾选 Use MicroLIB 复选框，目的是在日后编写串口驱动时可以使用 printf 函数，如图 5.12 所示。这一步的配置工作很重要，很多人串口用不了 printf 函数，编译有问题，下载有问题，都是因为这个步骤的配置有问题。

(2) 在 Output 选项卡中勾选 Create HEX File 复选框，以便在编译的过程中生成 hex 文件，如图 5.13 所示。

(3) 在 Listing 选项卡中把输出文件夹定位到工程目录下的 Listing 文件夹，双击

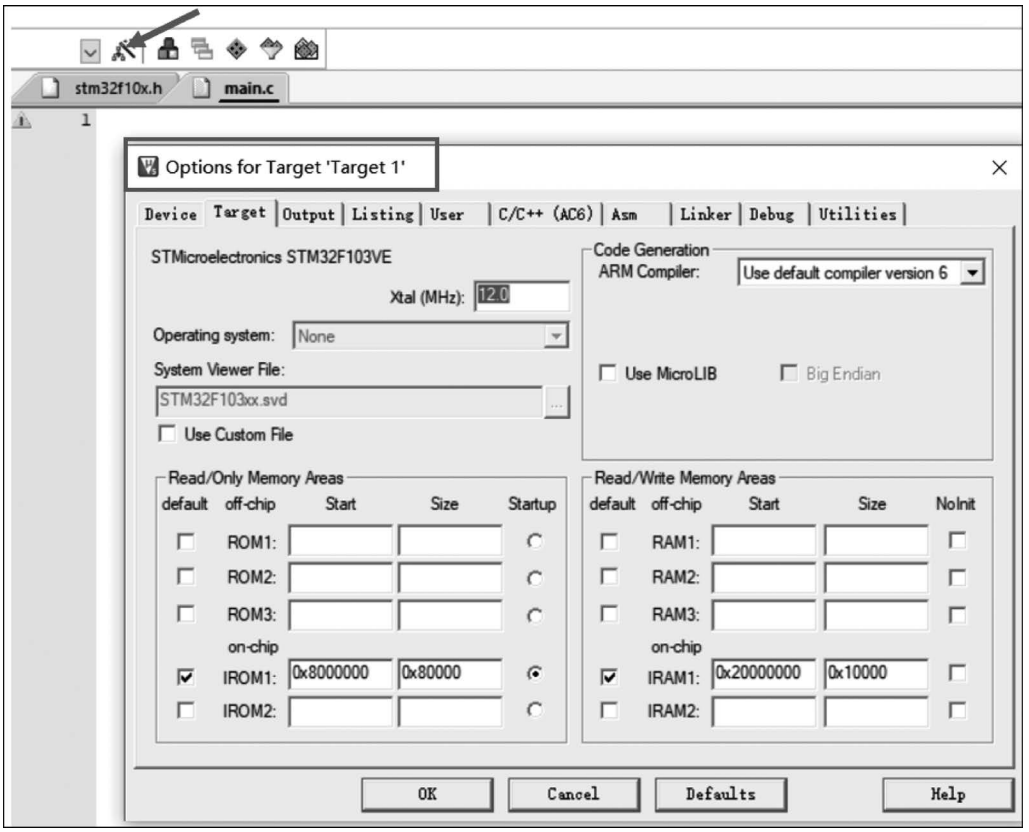


图 5.11　Options for Target‘Target 1’对话框

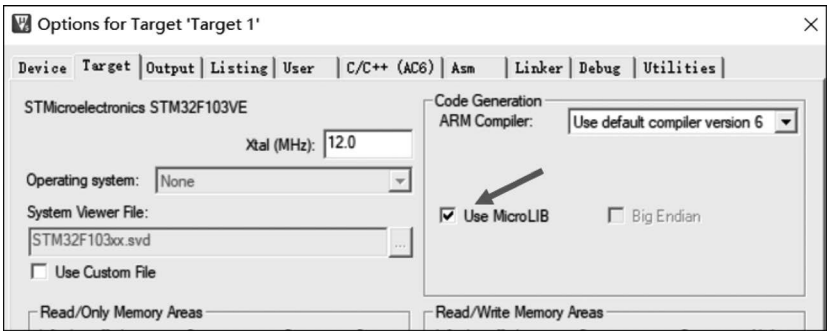


图 5.12　添加微库

Listing 文件夹，打开 Listing 文件夹，然后单击 OK 按钮，如图 5.14 和图 5.15 所示。

（4）在 C/C++选项卡中添加处理宏及编译器编译时查找的头文件路径。若头文件路径添加有误，则编译时会报错找不到头文件。在 Preprocessor Symbols 下的 Define 中输入 STM32F10X_HD 和 USE_STDPERIPH_DRIVER，中间用“,”隔开，如图 5.16 所示。

在这个选项中添加宏，就相当于在文件中使用“#define”语句定义宏。在编译器中

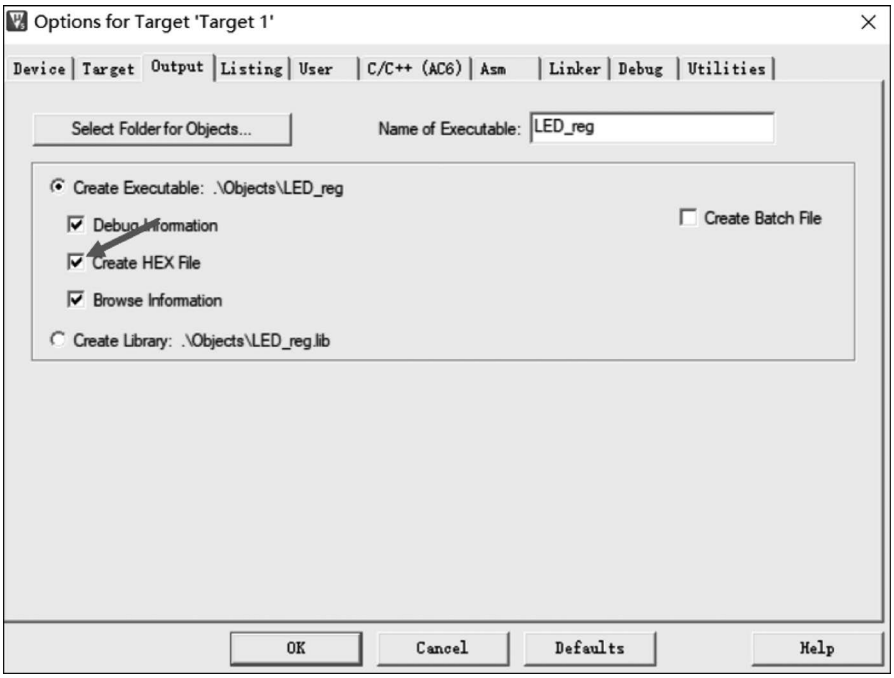


图 5.13　配置 Output 选项卡

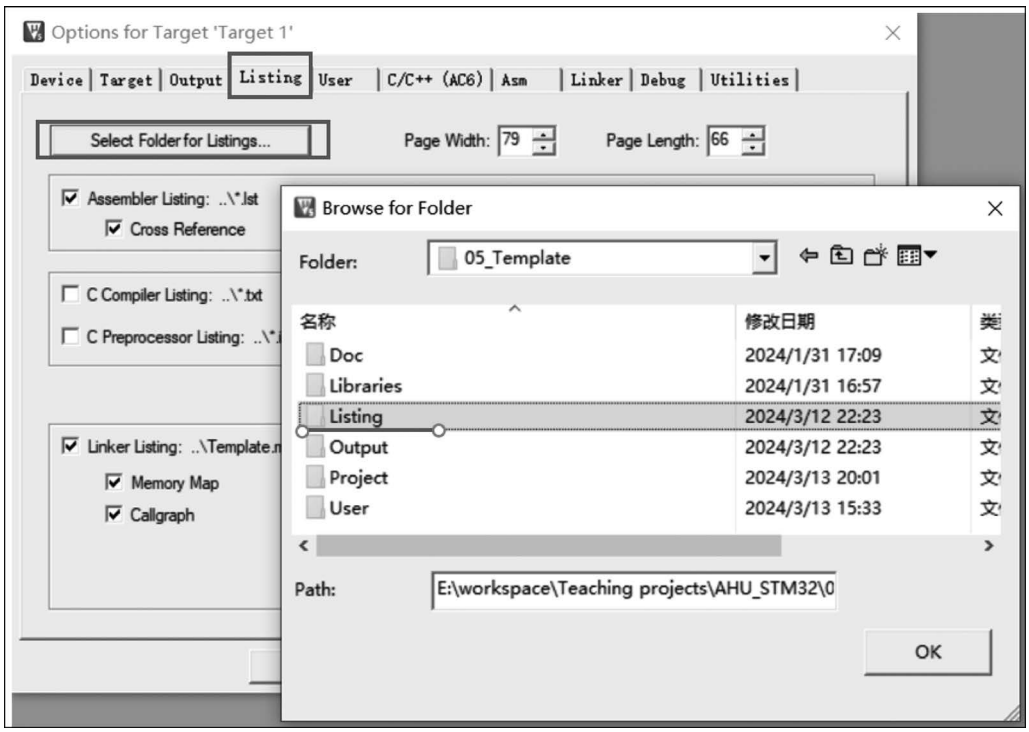


图 5.14　配置 Listing 选项卡 1

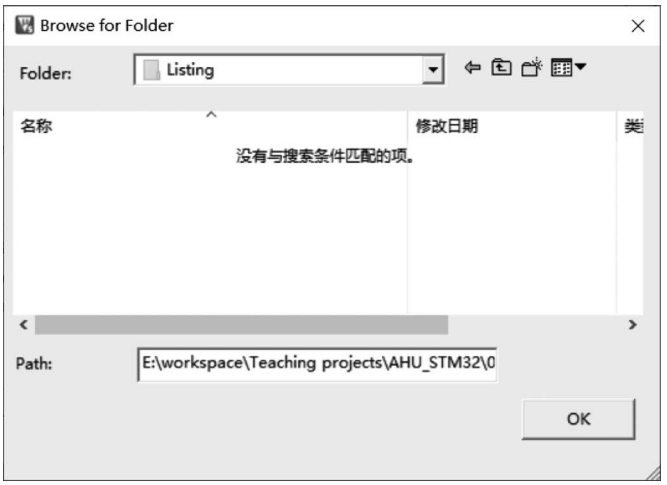


图 5.15　配置 Listing 选项卡 2

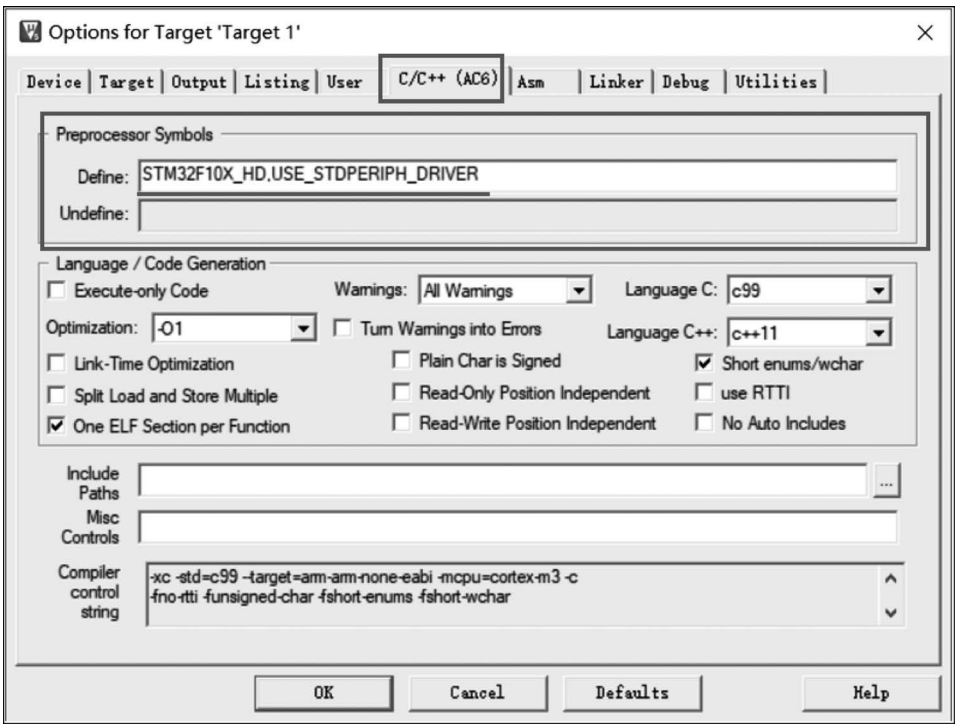


图 5.16　配置 C/C++选项卡 1

添加宏就不用在源文件中修改代码。

STM32F10X_HD 宏：为了告诉 STM32 标准库，项目工程使用的芯片类型是 STM32 大容量的，使 STM32 标准库根据选定的芯片型号来配置。

USE_STDPERIPH_DRIVER 宏：为了让 stm32f10x.h 包含 stm32f10x_conf.h 这个头文件。

C/C++的 Include Paths 中添加的是头文件的路径，如图 5.17 所示。如果编译时提示找不到头文件，一般是这里配置出了问题。头文件放到了哪个文件夹，就把哪个文件夹添加到这里。

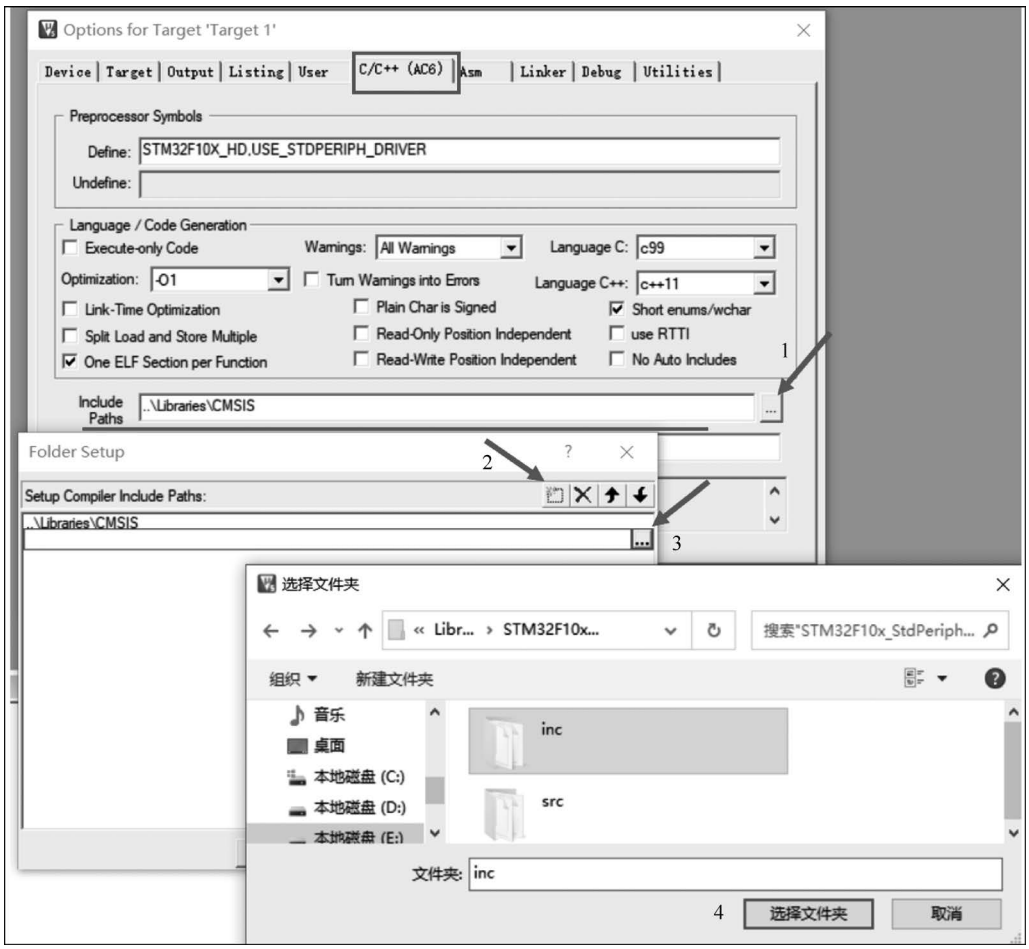


图 5.17　配置 C/C++选项卡 2

(5) 下载器配置。在仿真器连接好计算机和开发板且开发板供电正常的情况下，按照 2.2.2 节 DAP 仿真器配置进行配置，不再赘述。

至此，一个新的工程模板建立完毕。

5.1.2　使用库函数点亮 LED 灯

1. 硬件电路设计

使用库函数点亮 LED 灯的硬件电路设计与 4.2.4 节一样，不再赘述。

2. 软件电路设计

为了对整个项目工程中的各个文件关系有感性认识，先熟悉库函数点亮 LED 灯项目，后面将在此项目基础上详细分析 CMSIS 标准和库层次关系，以及如何构建库函数，

在此只介绍核心代码部分。

新建工程文件夹“05_LED_FLB”，并将工程模板文件夹“05_Template”的 6 个文件夹都复制到“05_LED_FLB”文件夹中，同时将“Project”的工程文件“Template”更名为“LED_FLB”。

为了使工程更加有条理，把 LED 灯控制相关的代码独立分开存储，方便以后移植。在“工程模板”的“USER”中新建文件夹“bsp_led”，并在该文件夹中新建“bsp_led.c”及“bsp_led.h”文件。“bsp”即板级支持包(board support packet)。这两个文件不属于 STM32 标准库的内容，是根据应用需要编写的。

编程要点如下：

(1) 使能 GPIO 端口时钟；

(2) 初始化 GPIO 目标引脚为推挽输出模式；

(3) 根据功能逻辑编写程序，控制 GPIO 引脚输出高、低电平。

3. 代码编写与分析

为了实现三个 LED 灯依次循环点亮，需要对 PA1、PB0、PB1 三个引脚依次循环输出低电平。

为了更好地进行程序移植，缩短开发周期，在编写应用程序的过程中，如果更改硬件环境，技术人员希望程序只需要做最小的修改即可在新的环境正常运行。例如，LED 灯的控制引脚与当前的不一样，一般把硬件相关的部分使用宏来封装，若更改了硬件环境，只修改这些硬件相关的宏即可。这些宏定义一般存储在头文件，如本例中的“bsp_led.h”文件中，见代码清单 5.1。

代码清单 5.1　LED 控制引脚相关的宏

```
1  /* 定义 LED 连接的 GPIO 口，只需要修改下面的代码即可改变控制的 LED 引脚 */
2  // R-绿色
3  #define LED1_GPIO_PORT     GPIOA                    /* GPIO 端口 */
4  #define LED1_GPIO_CLK      RCC_APB2Periph_GPIOA     /* GPIO 端口时钟 */
5  #define LED1_GPIO_PIN      GPIO_Pin_1               /* 连接到 SCL 时钟线的 GPIO */
6
7  // G-红色
8  #define LED2_GPIO_PORT     GPIOB                    /* GPIO 端口 */
9  #define LED2_GPIO_CLK      RCC_APB2Periph_GPIOB     /* GPIO 端口时钟 */
10 #define LED2_GPIO_PIN      GPIO_Pin_0               /* 连接到 SCL 时钟线的 GPIO */
11
12 // B-蓝色
13 #define LED3_GPIO_PORT     GPIOB                    /* GPIO 端口 */
14 #define LED3_GPIO_CLK      RCC_APB2Periph_GPIOB     /* GPIO 端口时钟 */
15 #define LED3_GPIO_PIN      GPIO_Pin_1               /* 连接到 SCL 时钟线的 GPIO */
```

以上代码分别把控制 LED 灯的 GPIO 端口、GPIO 引脚号以及 GPIO 端口时钟封装起来。在实际控制时直接用这些宏，而不需要直接对硬件相关寄存器操作。其中的 GPIO 时钟宏“RCC_APB2Periph_GPIOB”是 STM32 标准库定义的 GPIO 端口时钟相关的宏，它的作用与“GPIO_Pin_x”这类宏类似，是用于指示寄存器位的，方便库函数使用。下面初始化 GPIO 时钟时可以看到它的用法。

1）控制 LED 灯亮灭状态的宏定义

为了方便控制 LED 灯，把 LED 灯常用的亮灭及状态反转的控制也直接定义成宏，见代码清单 5.2。

代码清单 5.2　控制 LED 灯亮灭的宏

```
1  /* 直接操作寄存器的方法控制 IO */
2  #define digitalHi(p,i)     {p->BSRR = i;}          //输出为高电平
3  #define digitalLo(p,i)     {p->BRR = i;}           //输出低电平
4  #define digitalToggle(p,i){p->ODR ^= i;}          //输出反转状态
5
6  /* 定义控制 IO 的宏 */
7  #define LED1_TOGGLE        digitalToggle(LED1_GPIO_PORT,LED1_GPIO_PIN)
8  #define LED1_OFF           digitalHi(LED1_GPIO_PORT,LED1_GPIO_PIN)
9  #define LED1_ON            digitalLo(LED1_GPIO_PORT,LED1_GPIO_PIN)
10
11 #define LED2_TOGGLE        digitalToggle(LED2_GPIO_PORT,LED2_GPIO_PIN)
12 #define LED2_OFF           digitalHi(LED2_GPIO_PORT,LED2_GPIO_PIN)
13 #define LED2_ON            digitalLo(LED2_GPIO_PORT,LED2_GPIO_PIN)
14
15 #define LED3_TOGGLE        digitalToggle(LED3_GPIO_PORT,LED3_GPIO_PIN)
16 #define LED3_OFF           digitalHi(LED3_GPIO_PORT,LED3_GPIO_PIN)
17 #define LED3_ON            digitalLo(LED3_GPIO_PORT,LED3_GPIO_PIN)
18
19 /* 基本混色,后面高级用法使用 PWM 可混出全彩颜色,且效果更好 */
20 //红
21 #define LED_RED \
                    LED1_ON;\
                    LED2_OFF\
                    LED3_OFF
22
23 //绿
24 #define LED_GREEN\
                    LED1_OFF;\
                    LED2_ON\
                    LED3_OFF
25
26 //蓝
27 #define LED_BLUE\
                    LED1_OFF;\
                    LED2_OFF\
                    LED3_ON
28
29 //黄(红 + 绿)
30 #define LED_YELLOW\
                    LED1_ON;\
                    LED2_ON\
                    LED3_OFF

31 //紫(红 + 蓝)
32 #define LED_PURPLE\
                    LED1_ON;\
                    LED2_OFF\
                    LED3_ON
33
34 //青(绿 + 蓝)
```

```
35  #define LED_CYAN \
                        LED1_OFF;\
                        LED2_ON\
                        LED3_ON
36
37  //白(红 + 绿 + 蓝)
38  #define LED_WHITE\
                        LED1_ON;\
                        LED2_ON\
                        LED3_ON
39
40  //黑(全部关闭)
41  #define LED_RGBOFF\
                        LED1_OFF;\
                        LED2_OFF\
                        LED3_OFF
```

这部分宏控制 LED 亮灭的操作是直接向 BSRR、BRR 和 ODR 这三个寄存器写入控制指令来实现的，对 BSRR 写 1 输出高电平，对 BRR 写 1 输出低电平，对 ODR 寄存器某位进行异或操作可反转位的状态。宏定义语句间的关系(图 5.18)如下：

(1) 代码通过宏定义将 GPIOA 与 GPIO_Pin_1 分别定义为 LED1_GPIO_PORT 和 LED1_GPIO_PIN，这部分代码在代码清单 5.1 中；

(2) 宏定义语句“#define LED1_TOGGLE digitalToggle (LED1_GPIO_PORT, LED1_GPIO_PIN)”，将函数 digitalToggle (LED1_GPIO_PORT, LED1_GPIO_PIN)定义为 LED1_TOGGLE，完成输出引脚反转操作；

(3) 宏定义语句“#define digitalToggle(p,i) {p-> ODR^=i;}”，将对 I/O 寄存器操作定义为函数“digitalToggle(p,i)”。

通过几轮宏定义，将程序变得更具可读性，变得更具移植性，但实际还是实现对 I/O 寄存器的操作，即“GPIOA-> ODR ^=0x0002”。宏定义语句“#define GPIO_Pin_1 ((uint16_t)0x0002)”位于固件库中的文件“stm32f10x_gpio.h”中。

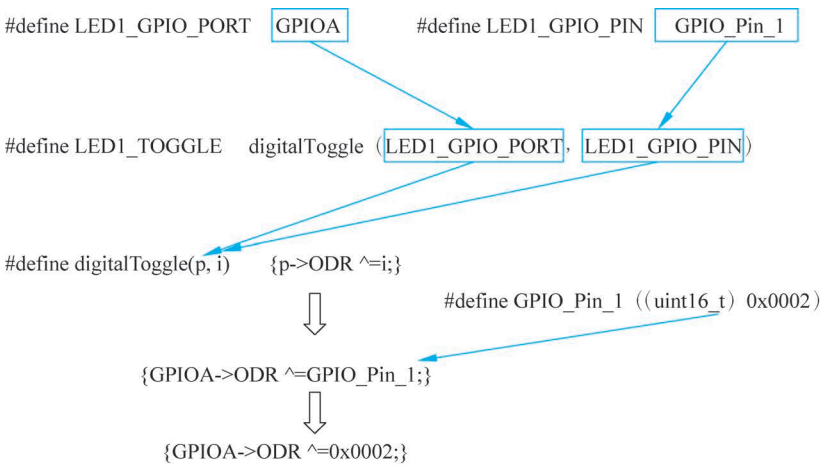


图 5.18 宏定义语句间的关系

RGB彩灯可以实现混色，如代码清单5.2中31行和32行代码，控制红灯和蓝灯亮而绿灯灭，可混出紫色效果。

代码中的“\”是C语言中的续行符语法，表示续行符的下一行与续行符所在的代码是同一行。代码中因为宏定义关键字“#define”只是对当前行有效，所以使用续行符来连接起来，与代码“#define LED_YELLOW LED1_ON; LED2_ON; LED3_OFF”是等效的。应用续行符时要注意，在“\”后面不能有任何字符（包括注释、空格），只能直接按回车键。

2）LED GPIO初始化函数

利用上面的宏，编写LED灯的初始化函数，见代码清单5.3。

代码清单5.3 LED GPIO初始化函数

```
1  void LED_GPIO_Config(void)
2  {
3  /* 定义一个 GPIO_InitTypeDef 类型的结构体 */
4  GPIO_InitTypeDef GPIO_InitStructure;
5
6  /* 开启 LED 相关的 GPIO 外设时钟 */
7  RCC_APB2PeriphClockCmd(   LED1_GPIO_CLK |
8                            LED2_GPIO_CLK |
9                            LED3_GPIO_CLK, ENABLE);
10 /* 选择要控制的 GPIO 引脚 */
11 GPIO_InitStructure.GPIO_Pin = LED1_GPIO_PIN;
12
13 /* 设置引脚模式为通用推挽输出 */
14 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
15
16 /* 设置引脚速率为 50MHz */
17 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
18
19 /* 调用库函数，初始化 GPIO */
20 GPIO_Init(LED1_GPIO_PORT, &GPIO_InitStructure);
21
22 /* 选择要控制的 GPIO 引脚 */
23 GPIO_InitStructure.GPIO_Pin = LED2_GPIO_PIN;
24
25 /* 调用库函数，初始化 GPIO */
26 GPIO_Init(LED2_GPIO_PORT, &GPIO_InitStructure);
27
28 /* 选择要控制的 GPIO 引脚 */
29 GPIO_InitStructure.GPIO_Pin = LED3_GPIO_PIN;
30
31 /* 调用库函数，初始化 GPIOF */
32 GPIO_Init(LED3_GPIO_PORT, &GPIO_InitStructure);
33
34 /*  关闭所有 led 灯 */
35 GPIO_SetBits(LED1_GPIO_PORT, LED1_GPIO_PIN);
36
37 /*  关闭所有 led 灯 */
38 GPIO_SetBits(LED2_GPIO_PORT, LED2_GPIO_PIN);
39
40 /*  关闭所有 led 灯 */
41 GPIO_SetBits(LED3_GPIO_PORT, LED3_GPIO_PIN);
42 }
```

整个函数与硬件相关的部分使用宏来代替。初始化 GPIO 端口时钟时也采用了 STM32 库函数，函数执行流程如下：

（1）使用 GPIO_InitTypeDef 定义 GPIO 初始化结构体变量，以便下面用于存储 GPIO 配置。

（2）调用库函数 RCC_APB2PeriphClockCmd 来使能 LED 灯的 GPIO 端口时钟。该函数有两个输入参数：一个参数用于指示要配置的时钟，如本例中的“RCC_APB2Periph_GPIOB”，应用时使用“|”操作同时配置 3 个 LED 灯的时钟；另一个参数用于设置状态，可输入“Disable”关闭或“Enable”使能时钟。

（3）向 GPIO 初始化结构体赋值，把引脚初始化成推挽输出模式，其中的 GPIO_Pin 使用宏“LEDx_GPIO_PIN”来赋值，使函数的实现方便移植。

（4）使用以上初始化结构体的配置，调用 GPIO_Init 函数向寄存器写入参数，完成 GPIO 的初始化。这里的 GPIO 端口使用“LEDx_GPIO_PORT”宏来赋值，也是为了程序移植方便。

（5）使用同样的初始化结构体，只修改控制的引脚和端口，初始化其他 LED 灯使用的 GPIO 引脚。

（6）使用宏控制 RGB 灯默认关闭。

3）主函数

编写完 LED 灯的控制函数后，就可以在 main 函数中测试，见代码清单 5.4。

代码清单 5.4　控制 LED 灯 main 程序

```
#include "stm32f10x.h"
#include "bsp_led.h"

#define SOFT_DELAY Delay(0x01FFFFF);

void Delay(__IO u32 nCount);
int main(void)
{
  /* LED 端口初始化 */
  LED_GPIO_Config();

  while (1)
  {
    LED1_ON;                          // 亮
    SOFT_DELAY;
    LED1_OFF;                         // 灭

    LED2_ON;                          // 亮
    SOFT_DELAY;
    LED2_OFF;                         // 灭

    LED3_ON;                          // 亮
    SOFT_DELAY;
    LED3_OFF;                         // 灭

    /*轮流显示 红绿蓝黄紫青白 颜色*/
    LED_RED;
```

```
28        SOFT_DELAY;
29
30        LED_GREEN;
31        SOFT_DELAY;
32
33        LED_BLUE;
34        SOFT_DELAY;
35
36        LED_YELLOW;
37        SOFT_DELAY;
38
39        LED_PURPLE;
40        SOFT_DELAY;
41
42        LED_CYAN;
43        SOFT_DELAY;
44
45        LED_WHITE;
46        SOFT_DELAY;
47
48        LED_RGBOFF;
49        SOFT_DELAY;
50      }
51    }
52
53    void Delay(__IO uint32_t nCount)        //简单的延时函数
54    {
55      for(; nCount != 0; nCount--);
56    }
```

在 main 函数中，调用前面定义的 LED_GPIO_Config 初始化好 LED 的控制引脚，然后直接调用各种控制 LED 灯亮灭的宏来实现 LED 灯的控制。

以上就是一个使用 STM32 标准软件库开发应用的流程。

4. 下载验证

把编译好的程序下载到开发板，可看到 RGB 彩灯轮流显示不同的颜色，如图 5.19 所示。

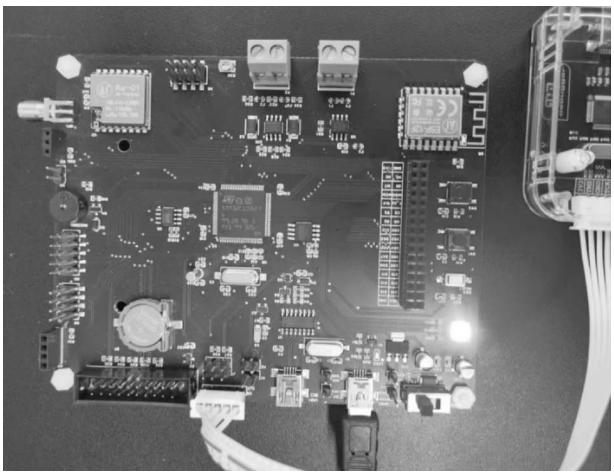


图 5.19　RGB 灯依次闪烁

5.2 CMSIS 标准及 STM32 库层次关系

基于 Cortex-M3（简写为 CM3）内核的芯片制造厂商很多，如 ST、Freescale、SAMSUNG 等。虽然基于 CM3 架构系列芯片采用的内核都是相同的，但它们核外的片上外设的不同导致了在相同内核上运行的软件移植困难。为了解决不同厂商生产的 Cortex 微控制器软件的兼容性问题，ARM 公司于 2008 年 11 月发布了旨在降低 Cortex-M 处理器软件的移植难度，并减少新手使用微控制器学习与开发实践的处理器软件接口 CMSIS，即 ARM Cortex 微控制器软件接口标准（Cortex Micro-controller Software Interface Standard）。STM32 固件库是基于 CMSIS 标准的 BSP（Board Support Package）包。

5.2.1 基于 CMSIS 标准的软件架构

基于 CMSIS 标准的软件架构如图 5.20 所示。

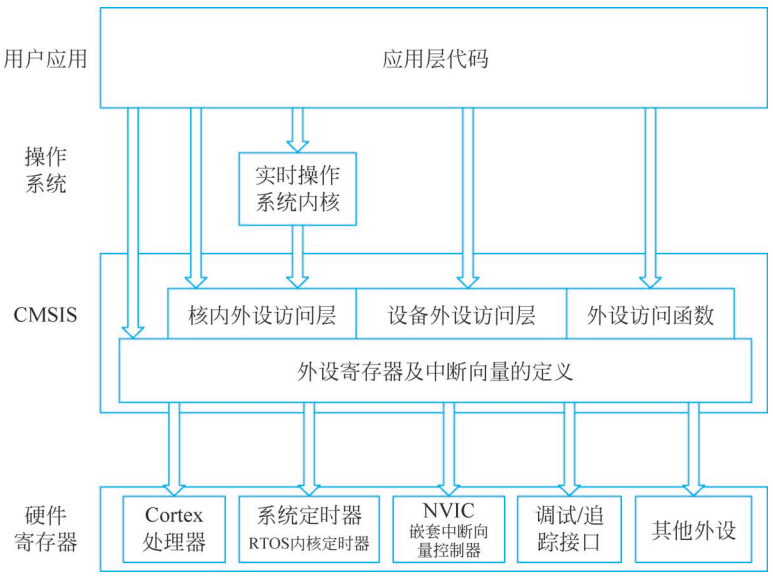


图 5.20 基于 CMSIS 标准的软件架构

从图 5.20 可以看到，基于 CMSIS 标准的软件架构主要分成用户应用层、操作系统层、CMSIS 层以及硬件寄存器层。其中 CMSIS 起着承上启下的作用，一方面对硬件寄存器层进行了统一的实现，屏蔽了不同厂商对 Cortex-M 系列微处理器内核外设寄存器的不同定义；另一方面向上层的操作系统和用户应用层提供接口，简化了应用程序开发的难度，使开发人员能够在完全透明的情况下进行一些应用程序的开发。

CMSIS 层主要分为以下三个层次：

（1）核内外设访问层（Core Peripheral Access Layer，CPAL）：该层由 ARM 负责实现，所定义的接口函数都是可重入的。其实现文件为 core_cm3.h 和 core_cm3.c，内容主要包括：

① 对核内寄存器名称，地址的定义；

② 嵌套向量中断控制器，以及对特殊用途寄存器、调试子系统的访问接口定义；

③ 对不同编译器的差异使用“__INLINE”来进行统一化处理；

④ 定义了一些访问 CM3 核内寄存器的函数，如对 xPSR、MSP、PSP 等寄存器的访问。

(2) 设备外设访问层(Device Peripheral Access Layer，DPAL)：该层由芯片厂商负责实现，负责对外设寄存器地址，及其访问接口进行定义。该层可调用 CPAL 提供的接口函数，同时根据处理器特性对异常向量表进行扩展，以处理相应外设的中断请求。相应的实现文件有 stm32f10x. h、system_stm32f10x. h、system_stm32f10x. c、startup_stm32f10x_hd. s、stm32f10x_it. h、stm32f10x_it. c。

(3) 外设访问函数(Access Functions for Peripherals，AFP)：这一层也由芯片厂商负责实现，主要提供访问片上外设的操作函数。

对一个 Cortex-M 微控制器系统而言，CMSIS 通过以上三个部分实现了定义了访问外设寄存器和异常向量的通用方法，定义了核内外设的寄存器名称和核异常向量的名称，以及为 RTOS 核定义了与设备独立的接口，包括 Debug 通道。

这样芯片厂商就能专注对芯片外设特性进行差异化，并且消除他们对微控制器进行编程时需要维持的不同的、互不兼容的标准需求，以达到低成本开发的目的。

5.2.2 STM32 固件库

本书讲解的 3.5.0 版本的标准固件库，以下内容请参考文件阅读。解压库文件后进入其目录为“STM32F10x_StdPeriph_Lib_V3.5.0\”。该文件夹下包含的内容如图 5.21 所示。

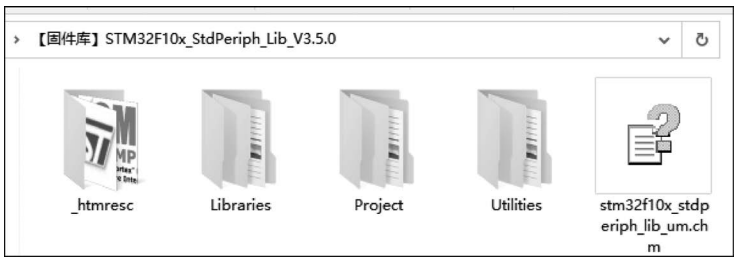


图 5.21 STM32F10x_StdPeriph_Lib_V3.5.0 文件夹中的内容

_htmresc 文件夹：其里面是两张 Logo 图片，一张是 CMSIS 的，另一张是 ST 的。

Libraries 文件夹：其里面是驱动库的源代码及启动文件。这里面的两个文件夹中的文件都非常重要。

Project 文件夹：文件夹下是用驱动库写的例子文件夹 STM32F10x_StdPeriph_Examples 和工程模板文件夹 STM32F10x_StdPeriph_Template，其中那些为每个外设写好的例程可以作程序编写的参考，例程非常全面，包括了外设的所有功能。

Utilities 文件夹：包含了基于 ST 官方实验板/评估板的例程，不需要用到，略过

即可。

stm32f10x_stdperiph_lib_um.chm 文件夹：库帮助文档，可以查询到每个外设库函数说明，是 ST 公司已经写好了的每个外设驱动，非常翔实。

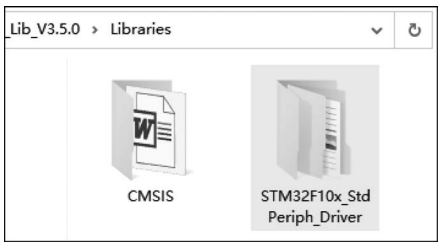


图 5.22　Libraries 文件内的内容

在使用固件库开发时，仅需要把 Libraries 目录下的库函数文件添加到工程的相应文件夹中，并通过查阅库帮助文档来了解 ST 提供的库函数的使用方法。进入 Libraries 文件夹可以看到 CMSIS 和 STM32F10x_StdPeriph_Driver 文件夹，如图 5.22 所示。

1. CMSIS 文件夹

STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\文件夹展开内容如图 5.23 所示。

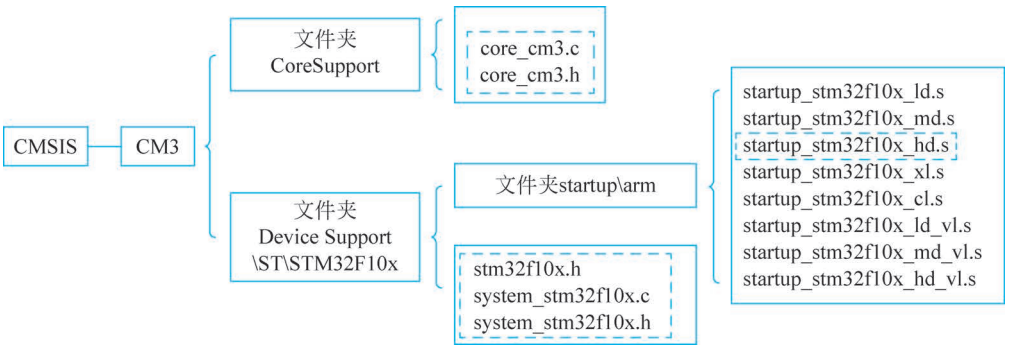


图 5.23　CMSIS 文件夹内容

1）内核文件：core_cm3.h 文件与 core_cm3.c 文件

core_cm3.h 头文件里面实现了内核的寄存器映射，对应外设头文件 stm32f10x.h。它们的区别是 core_cm3.h 针对内核的外设，stm32f10x.h 针对片上（内核之外）的外设。

core_cm3.c 文件实现了一下操作内核外设寄存器的函数。此外，core_cm3.c 文件包含了“stdint.h”头文件。“stdint.h”是一个 ANSI C 文件，类似于 C 语言头文件“stdio.h”，是独立于处理器之外的，位于 Keil 软件的安装目录下，主要作用是提供一些类型定义。stdint.h 文件的部分代码见代码清单 5.5。

代码清单 5.5　stdint.h 文件的部分代码

```
1  /* exact-width signed integer types */
2  typedef signed          char       int8_t;
3  typedef signed short    int        int16_t;
4  typedef signed          int        int32_t;
5  typedef signed          __INT64    int64_t;

6  /* exact-width unsigned integer types */
7  typedef unsigned        char       uint8_t;
```

```
typedef unsigned short  int         uint16_t;
typedef unsigned        int         uint32_t;
typedef unsigned        __INT64     uint64_t;
```

2）启动文件：startup_stm32f10x_hd.s

在 startup/arm 文件夹下面存放了多个启动文件，不同的驱动文件针对不同型号的 stm32f10x 系列的芯片，以 stm32f10x 系列的芯片片内的 Flash 容量来做区分。本教程开发板中用的 STM32F103VET6 的 Flash 是 512K，属于基本型的大容量产品，启动文件统一选择 startup_stm32f10x_hd.s。

3）头文件：stm32f10x.h

stm32f10x.h 为最基础的头文件，它主要包含了以下三方面的内容：

（1）通用数据类型定义，见代码清单 5.6。

代码清单 5.6　stm32f10x.h 头文件里通用数据类型定义

```
/*!< STM32F10x Standard Peripheral Library old types (maintained for legacy
 purpose) */
typedef int32_t s32;
typedef int16_t s16;
typedef int8_t  s8;

typedef const int32_t sc32;   /*!< Read Only */
typedef const int16_t sc16;   /*!< Read Only */
typedef const int8_t sc8;     /*!< Read Only */
……
```

（2）定义所有外设的寄存器组结构，如 5.1 节中使用到的 GPIO 寄存器组，见代码清单 5.7。

代码清单 5.7　外设寄存器组定义

```
typedef struct
{
__IO uint32_t CRL;
__IO uint32_t CRH;
__IO uint32_t IDR;
__IO uint32_t ODR;
__IO uint32_t BSRR;
__IO uint32_t BRR;
__IO uint32_t LCKR;
} GPIO_TypeDef;
```

由于外设的功能是通过其内部的寄存器来实现的，应将这些寄存器视为一个整体。通过 C 语言的结构体类型定义的方式在代码级就可实现这个操作。上面代码片断中的结构体类型 GPIO_TypeDef 代表了外设 GPIO，其内部成员是 GPIO 的 7 个寄存器。

（3）外设变量的声明，其部分代码见代码清单 5.8。

代码清单 5.8　外设变量的声明

```
#define FLASH_BASE      ((uint32_t)0x08000000) /*!< FLASH base address in the alias
region */
#define SRAM_BASE       ((uint32_t)0x20000000) /*!< SRAM base address in the alias
```

```
    region */
3   #define PERIPH_BASE     ((uint32_t)0x40000000) /*!< Peripheral base address in the
    alias region */
4   #define SRAM_BB_BASE    ((uint32_t)0x22000000) /*!< SRAM base address in the bit-
    band region */
5   #define PERIPH_BB_BASE  ((uint32_t)0x42000000) /*!< Peripheral base address in the
    bit-band region */

6   #define FSMC_R_BASE     ((uint32_t)0xA0000000) /*!< FSMC registers base address */

7   /*!< Peripheral memory map */
8   #define APB1PERIPH_BASE PERIPH_BASE
9   #define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
10  #define AHBPERIPH_BASE (PERIPH_BASE + 0x20000)

11  #define TIM2_BASE (APB1PERIPH_BASE + 0x0000)
12  #define TIM3_BASE (APB1PERIPH_BASE + 0x0400)
13  ......
```

所谓的“外设的声明”,其实质主要是一些宏定义,宏名就代表了某种外设,而宏值就是该外设基地址(实际上这种表述并不准确,在没有讲解具体外设之前,姑且这样认为)。如此一来,操作外设名就是在操作外设地址所在的寄存器,如对外设 GPIOA 的操作可以是“GPIOA-> IDR=17;”。

由此可见,文件 stm32f10x.h 在基于 STM32 库开发过程中的地位和作用。由于 stm32f10x.h 中绝大部分代码都是进行外设的声明和寄存器位定义,在用户使用 STM32 库编写外设驱动时必须将其包含在自己的工程文件中。

4) system_stm32f10x.h 和 system_stm32f10x.c

在系统上电复位那一刻,完成系统初始化的两个函数 SystemInit( ) 和 SystemCoreClockUpdate( ),以及全局变量 SystemCoreClock 就在这两个文件中实现,它们的作用分别如下:

(1) SystemInit(): 设置系统时钟源,其中涉及锁相环(PLL)倍频因子、AHB-APB 预分频因子,以及扩展 Flash 的设置等,该函数在启动文件(startup_stm32f10x_xx.s)中被调用。

(2) SystemCoreClock: 此变量代表高性能总线时钟(HCLK)的频率值,系统的“滴答”定时器定时长度的计算也是基于这个变量的。

(3) SystemCoreClockUpdate(): 在系统运行期间,若核心时钟 HCLK 需要改变,则必须调用此函数来调整 SystemCoreClock 的值。注意,只是在 HCLK 有了变化的情况下才使用它。

关于 system_stm32f10x 文件中所涉及的知识点,与系统的启动过程联系紧密,将在第 8 章“RCC 与 STM32 时钟”进行详细讲解。

2. STM32F10x_StdPeriph_Driver 文件夹

进入 Libraries 目录的 STM32F10x_StdPeriph_Driver 文件夹可以看到文件夹 inc 和

src，两个文件夹中的文件属于 CMSIS 之外的、芯片片上外设部分。src 里面是每个设备外设的驱动源程序，inc 则是相对应的外设头文件。src 和 inc 文件夹是 ST 标准库的主要内容。在 src 和 inc 文件夹里的是 ST 公司针对每个 STM32 外设而编写的库函数文件，每个外设对应一个.c 和.h 后缀的文件。这类外设文件统称为 stm32f10x_ppp.c 或 stm32f10x_ppp.h 文件，ppp 表示外设名称。如针对 GPIO 外设，在 src 文件夹下有一个 stm32f10x_gpio.c 源文件，在 inc 文件夹下有一个 stm32f10x_gpio.h 头文件，若开发的工程中用到了 STM32 内部的 ADC，则至少要把这两个文件包含到工程里，如图 5.24 所示。

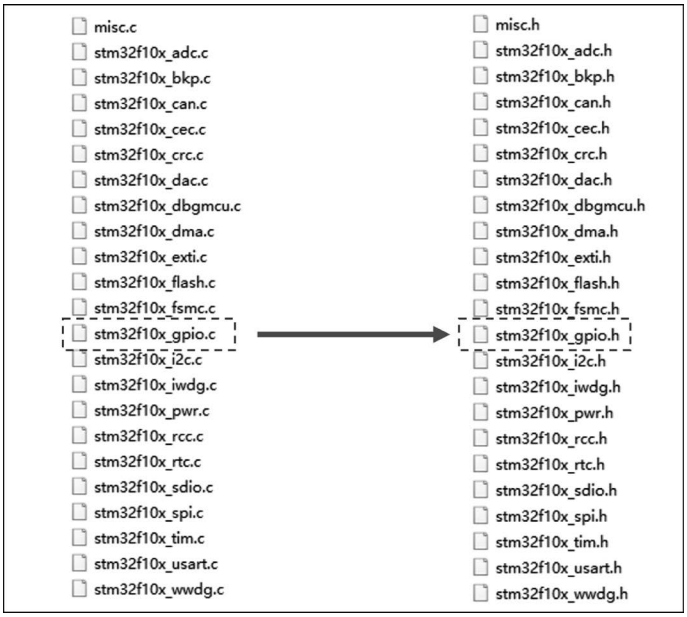


图 5.24　驱动的源文件及头文件

此外，还有相对特别的文件 misc.h 与 misc.c，这个文件提供了外设对内核中的 NVIC 的访问函数，在配置中断时必须把这个文件添加到工程中。

3. STM32F10x_StdPeriph_Template 文件夹

STM32F10x_StdPeriph_Template 文件夹在 Project 文件夹中。在这个文件目录下，存放了官方的一个库工程模板，在用库建立一个完整的工程时，还需要添加这个目录下的 stm32f10x_it.c、stm32f10x_it.h、stm32f10x_conf.h 和 system_stm32f10x.c 四个文件。

stm32f10x_it.c：文件是专门用来编写中断服务函数的，这个文件已经定义了一些系统异常（特殊中断）的接口，其他普通中断服务函数由自己添加。

stm32f10x_conf.h：文件被包含进 stm32f10x.h 文件。当使用固件库编程的时候，如果需要某个外设的驱动库，就应包含该外设的头文件 stm32f10x_ppp.h，包含一个还好，如果使用了多外设，就需要包含多个头文件，这不仅影响代码美观，而且不好管理。用一个头文件 stm32f10x_conf.h 把这些外设的头文件都包含在里面，让这个配置头文件

统一管理这些外设的头文件。这样在应用程序中只需要包含这个配置头文件即可。这个头文件在 stm32f10x.h 的最后被包含，所以最终只需要包含 stm32f10x.h 这个头文件即可，非常方便。stm32f10x_conf.h 见代码清单 5.9。默认情况下所有头文件都被包含，没有注释掉，也可以把不要的注释掉，只留下需要使用的。

代码清单 5.9　stm32f10x_conf.h 内容

```
/* Define to prevent recursive inclusion ------------------------------------- */
#ifndef __STM32F10x_CONF_H
#define __STM32F10x_CONF_H

/* Includes ------------------------------------------------------------------ */
/*  Uncomment/Comment the line below to enable/disable peripheral header file
  inclusion */
#include "stm32f10x_adc.h"
#include "stm32f10x_bkp.h"
#include "stm32f10x_can.h"
#include "stm32f10x_cec.h"
#include "stm32f10x_crc.h"
#include "stm32f10x_dac.h"
#include "stm32f10x_dbgmcu.h"
#include "stm32f10x_dma.h"
#include "stm32f10x_exti.h"
#include "stm32f10x_flash.h"
#include "stm32f10x_fsmc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_i2c.h"
#include "stm32f10x_iwdg.h"
#include "stm32f10x_pwr.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_rtc.h"
#include "stm32f10x_sdio.h"
#include "stm32f10x_spi.h"
#include "stm32f10x_tim.h"
#include "stm32f10x_usart.h"
#include "stm32f10x_wwdg.h"
#include "misc.h" /* High level functions for NVIC and SysTick (add-on to CMSIS
  functions) */
```

5.2.3　STM32 库层次关系

前面简单介绍了各个库文件的作用，库文件直接包含进工程即可，不用修改，而有的文件在使用时根据具体需要进行配置。库工程中各个文件关系如图 5.25 所示。

图 5.25 中描述了 STM32 库各文件之间的调用关系，在实际的使用库开发工程的过程中，要把位于 CMSIS 层的文件包含进工程，除了特殊系统时钟需要修改 system_stm32f10x.c，其他文件不用修改，也不建议修改。对位于用户层的几个文件，就是在使用库的时候针对不同的应用对库文件进行增删（用条件编译的方法增删）和改动的文件。

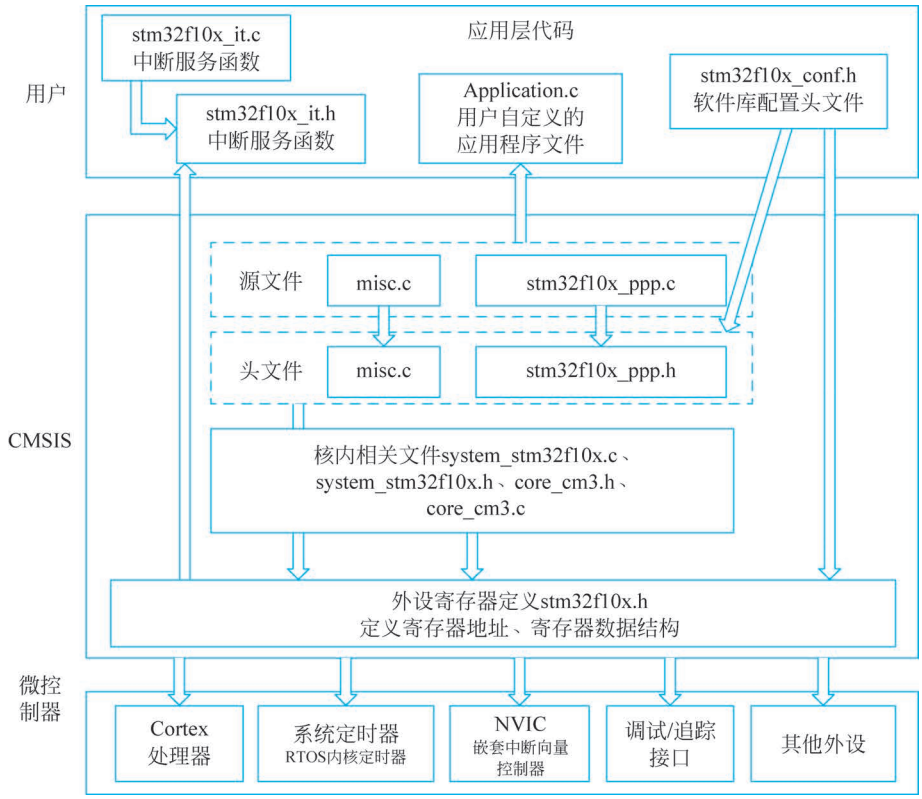


图 5.25　工程中各个文件关系

5.2.4　帮助文档

官方资料是所有关于 STM32 知识的源头，所以本节介绍如何使用官方资料。官方的帮助手册是最好的教程，几乎包含了所有在开发过程中遇到的问题。常用官方资料如下：

《STM32F10xxx 参考手册》：该文档全方位介绍了 STM32 芯片的各种片上外设，将 STM32 的时钟、存储器架构、各种外设、寄存器都描述得很清楚。

《STM32 规格书》：该文档相当于 STM32 的 datasheet，包含了 STM32 芯片所有的引脚功能说明，以及存储器架构、芯片外设架构说明。

《Cortex^{TM}-M3 内核编程手册》：该文档由 ST 公司提供，主要讲解 STM32 内核寄存器相关的说明，如系统定时器、NVIC 等核外设的寄存器。这部分的内容是《STM32F10xxx 参考手册》没涉及的内核部分的补充。

《Cortex-M3 权威指南》：该手册是由 ARM 公司提供的，详细讲解了 Cortex 内核的架构和特性，深入了解 Cortex-M 内核，这是首选。

《stm32f10x_stdperiph_lib_um.chm》：这就是本章提到的库的帮助文档，在使用库函数时，最好通过查阅此文件来了解标准库提供了哪些外设、函数原型或库函数的调用

的方法，如图 5.26 所示。

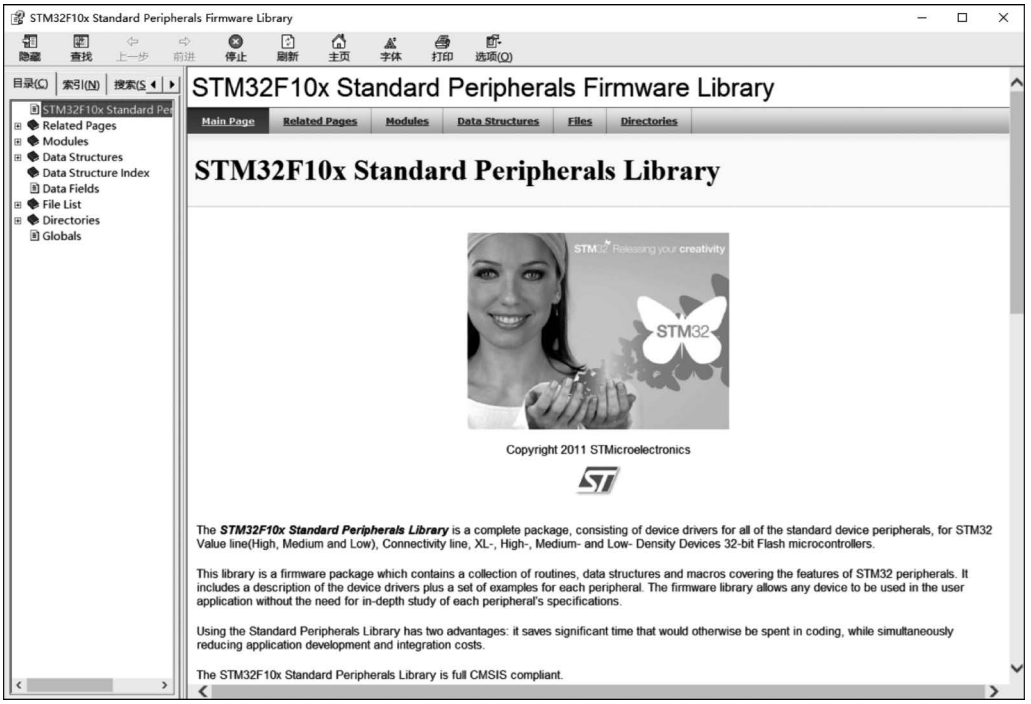


图 5.26　stm32f10x_stdperiph_lib_um.chm 帮助文件

5.3 库函数及其构建

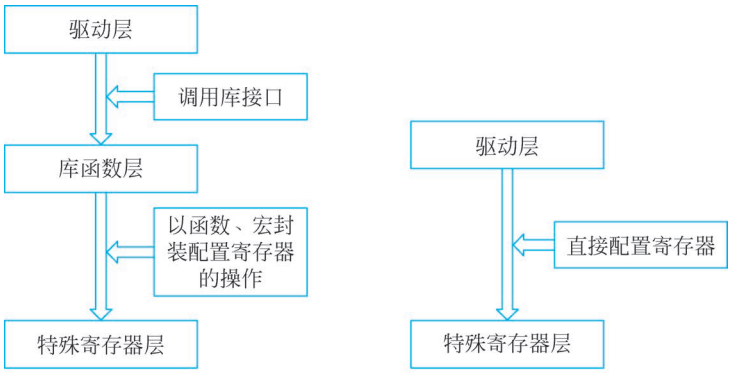
通过利用库函数驱动 LED 灯的实验从感性上认识了 STM32 固件库；通过对基于 CMSIS 标准软件架构与 STM32 固件库的介绍对 STM32 固件库有了更深刻的理解。为了进一步掌握 STM32 固件库的使用，本小节指导读者自己来写固件库。

5.3.1 固件库开发与寄存器开发

固件库是指“STM32 标准函数库”，它是由 ST 公司针对 STM32 提供的应用程序编程接口（Application Program Interface，API）。工程师通过调用这些函数接口来配置 STM32 的寄存器，避免了面对纷繁复杂的最底层的寄存器。基于固件库开发具有开发速度、易于阅读、维护成本低等优点。

库是架设在寄存器与用户驱动层之间的代码，向下处理与寄存器直接相关的配置，向上为用户提供配置寄存器的接口。固件库开发与寄存器开发的对比如图 5.27 所示。

相对固件库开发，寄存器开发生产的代码量少，具体参数直观、程序运行占用资源少。但因为 STM32 外设资源丰富，寄存器的数量众多，且寄存器位数也不再是 8 位，复杂程度更高，带来的不足是开发速度慢、程序可读性差、代码维护工作量大，导致各项成本增加。相对 8 位 CPU，STM32 的 CPU 资源充足，运行速度快。权衡优势与不足，在绝



(a) 固件库开发　　(b) 寄存器开发

图 5.27　固件库开发与寄存器开发的对比

大部分时候可以牺牲一些 CPU 资源，选择固件库开发方式。

5.3.2　构建库函数

尽管库的优点很多，但是一开始用库时面对的代码多、文件多，反不知道从何下手。其原因是不清楚什么是库，也不知道库是怎么实现的。

为此，在对用固件库开发有了一定认识后，接下来将对 5.1.2 节稍做修改，将与底层硬件相关代码都直接使用标准固件库来完成。

下面采用自顶向下的方式来封装库函数，进行编写程序。

1. 主函数

主函数先通过调用子程序“LED_GPIO_Config()”对 LED 端口进行初始化，接着循环执行 LED1、LED2、LED3 点亮、熄灭，见代码清单 5.10。

代码清单 5.10　主函数代码

```
#include "stm32f10x.h"
#include "stm32f10x_gpio.h"
#include "bsp_led.h"

#define SOFT_DELAY Delay(0x01FFFFF);

void Delay(uint32_t nCount);

int main(void)
{
  /* LED 端口初始化 */
  LED_GPIO_Config();

  while (1)
  {
    LED1(ON);                    // LED1 灯点亮
    SOFT_DELAY;                  // 延时
    LED1(OFF);                   // LED1 灯熄灭

    LED2(ON);
```

```
    SOFT_DELAY;
    LED2(OFF);

    LED3(ON);
    SOFT_DELAY;
    LED3(OFF);
  }
}

void Delay(__IO uint32_t nCount)    //简单软件延时
{
  for(; nCount != 0; nCount--);
}
  /*********************************** END OF FILE **********************/
```

从代码清单 5.10 可以看到，需要声明主函数涉及外设的 LED_GPIO_Config()、LED1(ON)、LED1(OFF)、LED2(ON)等子程序。将这些声明在 LED 外设的头文件 bsp_led.h 中完成。“__IO”在文件“stm32f10x.h”中做了声明。子程序 LED_GPIO_Config()在文件“bsp_led.c”中。

2. 外设 LED 程序

文件“bsp_led.c”文件主要编写了 LED_GPIO_Config()，见代码清单 5.11。

代码清单 5.11　bps_led.c 代码

```
#include "stm32f10x_gpio.h"
#include "bsp_led.h"
#include "stm32f10x_rcc.h"

void LED_GPIO_Config(void)
{
  /* 定义一个 GPIO_InitTypeDef 类型的结构体 */
  GPIO_InitTypeDef GPIO_InitStructure;

  /* 开启 LED 相关的 GPIO 外设时钟 */
  RCC_APB2PeriphClockCmd( LED1_GPIO_CLK | LED2_GPIO_CLK | LED3_GPIO_CLK, ENABLE);
  /* 选择要控制的 GPIO 引脚 */
  GPIO_InitStructure.GPIO_Pin = LED1_GPIO_PIN;

  /* 设置引脚模式为通用推挽输出 */
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;

  /* 设置引脚速率为 50MHz */
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;

  /* 调用库函数，初始化 GPIO */
  GPIO_Init(LED1_GPIO_PORT, &GPIO_InitStructure);

  /* 选择要控制的 GPIO 引脚 */
  GPIO_InitStructure.GPIO_Pin = LED2_GPIO_PIN;

  /* 调用库函数，初始化 GPIO */
  GPIO_Init(LED2_GPIO_PORT, &GPIO_InitStructure);

  /* 选择要控制的 GPIO 引脚 */
```

```
31    GPIO_InitStructure.GPIO_Pin = LED3_GPIO_PIN;
32
33    /* 调用库函数,初始化 GPIOF */
34    GPIO_Init(LED3_GPIO_PORT, &GPIO_InitStructure);
35
36    /* 关闭所有 led 灯   */
37    GPIO_SetBits(LED1_GPIO_PORT, LED1_GPIO_PIN);
38
39    /* 关闭所有 led 灯   */
40    GPIO_SetBits(LED2_GPIO_PORT, LED2_GPIO_PIN);
41
42    /* 关闭所有 led 灯   */
43    GPIO_SetBits(LED3_GPIO_PORT, LED3_GPIO_PIN);
44  }
45
46    /****************************** END OF FILE ***************************/
```

LED_GPIO_Config()函数执行流程如下：

(1) 使用 GPIO_InitTypeDef 定义 GPIO 初始化结构体变量，以便下面用于存储 GPIO 配置。GPIO_InitTypeDef 是一个结构体类型，用于配置 GPIO 端口的初始化参数 GPIO_Pin、GPIO_Speed、GPIO_Mode。该结构体类型定义在“stm32f10x. gpio. h”文件中。

(2) 调用库函数 RCC_APB2PeriphClockCmd 来使能 LED 灯的 GPIO 端口时钟。该函数定义在“stm32f10x. rcc. c”文件中，它有两个输入参数；一个参数用于指示要配置时钟的外设端口，如本例中的“RCC_APB2Periph_GPIOB”。在此语句中使用了“|”操作，可以同时配置 3 个 LED 灯的时钟。另一个参数用于设置状态，可输入“Disable”关闭或“Enable”使能时钟，“Disable”与“Enable”将在文件“stm32f10x. h”定义。

(3) 向 GPIO 初始化结构体赋值，即选择要控制的 GPIO 引脚，设置引脚模式为通用推挽输出，设置引脚速率为 50MHz。其中的 GPIO_Pin 使用宏“LEDx_GPIO_PIN”来赋值，使函数方便移植。

(4) 调用库函数 GPIO_Init(LED1_GPIO_PORT，&GPIO_InitStructure)，向寄存器写入参数，完成 GPIO 的初始化。

(5) 使用同样的初始化结构体，只修改控制的引脚和端口，初始化其他 LED 灯使用的 GPIO 引脚。

(6) 使用宏控制 RGB 灯默认关闭。

3. 外设 LED 的头文件

外设 LED 的头文件“bsp_led. h”代码见代码清单 5. 12。

代码清单 5. 12　外设的函数声明与端口定义

```
1  #ifndef __LED_H
2  #define __LED_H
3
4  #include "stm32f10x.h"
5
6  /* 定义 LED 连接的 GPIO 端口,用户只需修改下面代码即可改变控制的 LED 引脚 */
7  // R-红色
```

```
 8  #define LED1_GPIO_PORT    GPIOA                      /* GPIO端口 */
 9  #define LED1_GPIO_CLK     RCC_APB2Periph_GPIOA       /* GPIO端口时钟*/
10  #define LED1_GPIO_PIN     GPIO_Pin_1                 /*连接到GPIO的具体引脚上 */
11
12  // G-绿色
13  #define LED2_GPIO_PORT    GPIOB                      /* GPIO端口*/
14  #define LED2_GPIO_CLK     RCC_APB2Periph_GPIOB       /* GPIO端口时钟 */
15  #define LED2_GPIO_PIN     GPIO_Pin_0                 /*连接到GPIO的具体引脚上*/
16
17  // B-蓝色
18  #define LED3_GPIO_PORT    GPIOB                      /* GPIO端口 */
19  #define LED3_GPIO_CLK     RCC_APB2Periph_GPIOB       /* GPIO端口时钟 */
20  #define LED3_GPIO_PIN     GPIO_Pin_1                 /*连接到GPIO的具体引脚上*/
21
22  /** the macro definition to trigger the led on or off
23  1 - off
24  0 - on
25   */
26  #define ON  0
27  #define OFF 1
28
29  /* 使用标准的固件库控制IO*/
30  #define LED1(a)  if (a)  \
31           GPIO_SetBits(LED1_GPIO_PORT,LED1_GPIO_PIN);\
32           else     \
33           GPIO_ResetBits(LED1_GPIO_PORT,LED1_GPIO_PIN)
34
35  #define LED2(a)  if (a)  \
36           GPIO_SetBits(LED2_GPIO_PORT,LED2_GPIO_PIN);\
37           else     \
38           GPIO_ResetBits(LED2_GPIO_PORT,LED2_GPIO_PIN)
39
40  #define LED3(a)  if (a)  \
41           GPIO_SetBits(LED3_GPIO_PORT,LED3_GPIO_PIN);\
42           else     \
43           GPIO_ResetBits(LED3_GPIO_PORT,LED3_GPIO_PIN)
44
45  void     LED_GPIO_Config(void);
46
47  #endif /* __LED_H */
```

代码的8～20行分别对LED灯的端口号、引脚号和端口使能时钟做了定义。GPIOA/GPIOB外设端口在文件“stm32f10x.h”中做了声明；RCC_APB2Periph_GPIOx是通过RCC外设使能寄存器设置APB2总线对GPIOx外设时钟的使能或禁用状态的寄存器，定义在“stm32f10x_rcc.h”中；GPIO_Pin_x是端口具体引脚的声明，在文件“stm32f10x.gpio.h”中。代码30～40行实现的是根据a值的真假，用宏定义LEDx(a)来选择对应引脚的置位操作或复位操作。若a为真，则选择置位操作；否则选择复位操作。置位函数GPIO_SetBits(LEDx_GPIO_PORT、LEDx_GPIO_PIN)和复位函数GPIO_ResetBits(LEDx_GPIO_PORT、LEDx_GPIO_PIN)实现相应位的置位与复位，在文件“stm32f10x.gpio.c”中。45行是LED灯与GPIO相关的配置函数“LED_GPIO_Config(void);”在“bsp_led.c”文件中。

4. stm32f10x.h 头文件

1）外设存储器映射

外设寄存器结构体定义仅是一个定义，实现给这个结构体赋值就达到操作寄存器的效果。另外，还需要找到该寄存器的地址，从而把寄存器地址与结构体的地址对应起来。再找到外设的地址，把这些外设的地址定义成一个个宏，实现外设存储器的映射。

在“stm32f10x.h”文件中，把片上外设基地址、APB2 总线基地址、AHB 总线基地址、具体外设的基地址和 RCC 外设基地址定义成相应的宏，实现映射，见代码清单 5.13。

代码清单 5.13　外设存储器映射

```
#ifndef __STM32F10X_H
#define __STM32F10X_H

// 片上外设基地址
#define PERIPH_BASE     ((unsigned int)0x40000000)

/* APB1 总线基地址 */
#define APB1PERIPH_BASE PERIPH_BASE
/* APB2 总线基地址 */
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
/* RCC 外设基地址 */
#define AHBPERIPH_BASE  (PERIPH_BASE + 0x20000)

/* RCC 外设基地址 */
#define RCC_BASE        (AHBPERIPH_BASE + 0x1000)
/* GPIO 外设基地址 */
#define GPIOA_BASE      (APB2PERIPH_BASE + 0x0800)
#define GPIOB_BASE      (APB2PERIPH_BASE + 0x0C00)

/* RCC 外设基地址 */
#define RCC_APB2ENR      *(unsigned int *)(RCC_BASE + 0x18)

```

从代码清单 5.13 可以看出，GPIO 端口是挂接在 APB2 总线上，RCC 是挂接在 AHB 总线上，ABP2 与 AHB 都挂接在片上外设上，即 GPIOA_BASE 的地址是 0x40010800。

2）外设寄存器结构体定义

在操作寄存器时，操作的是都寄存器的绝对地址，如果每个外设寄存器都这样操作，将会相当复杂。考虑到外设寄存器的地址都是基于外设基地址的偏移地址，并且是在外设基地址上逐个连续递增的，每个寄存器占 32B。这种方式与结构体里面的成员类似，可定义一种外设结构体，结构体的地址代表外设的基地址，结构体的成员代表寄存器，成员的排列顺序和寄存器的顺序一样。这样，在操作寄存器时就不用每次都找到绝对地址，只知道外设的基地址就可以操作外设的全部寄存器，即操作结构体的成员。

在“stm32f10x.h”文件中，使用结构体封装 GPIO 及 RCC 外设的寄存器，如代码清单 5.14。结构体成员的顺序按照寄存器的偏移地址从低到高排列，成员类型和寄存器类型一样。

代码清单 5.14　封装寄存器列表

```
23  //寄存器的值常常是芯片外设自动更改的,即使 CPU 没有执行程序,也有可能发生变化
24  //编译器有可能会对没有执行程序的变量进行优化
25
26  //volatile 表示易变的变量,防止编译器优化
27  #define __IO volatile
28  typedef unsigned int uint32_t;
29  typedef unsigned short uint16_t;
30  typedef enum {DISABLE = 0, ENABLE = !DISABLE} FunctionalState;
31  // GPIO 寄存器结构体定义
32  typedef struct
33  {
34    __IO uint32_t CRL;   // 端口配置低寄存器,地址偏移 0x00
35    __IO uint32_t CRH;   // 端口配置高寄存器,地址偏移 0x04
36    __IO uint32_t IDR;   // 端口数据输入寄存器,地址偏移 0x08
37    __IO uint32_t ODR;   // 端口数据输出寄存器,地址偏移 0x0C
38    __IO uint32_t BSRR;  // 端口位设置/清除寄存器,地址偏移 0x10
39    __IO uint32_t BRR;   // 端口位清除寄存器,地址偏移 0x14
40    __IO uint32_t LCKR;  // 端口配置锁定寄存器,地址偏移 0x18
41  } GPIO_TypeDef;
42
43  typedef struct          // rcc 寄存器结构体定义
44  {
45      __IO uint32_t CR;
46      __IO uint32_t CFGR;
47      __IO uint32_t CIR;
48      __IO uint32_t APB2RSTR;
49      __IO uint32_t APB1RSTR;
50      __IO uint32_t AHBENR;
51      __IO uint32_t APB2ENR;
52      __IO uint32_t APB1ENR;
53      __IO uint32_t BDCR;
54      __IO uint32_t CSR;
55  }RCC_TypeDef;
56
```

这段代码在每个结构体成员前增加了“__IO”前缀,它的原型在这段代码的第 1 行,代表了 C 语言中的关键字“volatile”,在 C 语言中该关键字用于表示变量是易变的,要求编译器不优化。这些结构体内的成员都代表着寄存器,而寄存器很多时候是由外设或 STM32 芯片状态修改的,也就是说,即使 CPU 不执行代码修改这些变量,变量的值也有可能被外设修改、更新。因此,每次使用这些变量的时候,都要求 CPU 去该变量的地址重新访问。若没有这个关键字修饰,在某些情况下,编译器认为没有代码修改该变量,就直接从 CPU 的某个缓存获取该变量值,这时可以加快执行速度,但该缓存中的是陈旧数据,与要求的寄存器最新状态可能会有出入。

3) 外设声明

定义好外设寄存器结构体,实现完外设存储器映射后,再把外设的基址强制类型转换成相应的外设寄存器结构体指针,然后把该指针声明成外设名。这样,外设名就与外设的地址对应起来,而且该外设名还是一个该外设类型的寄存器结构体指针,通过该指针可以直接操作该外设的全部寄存器,见代码清单 5.15,此代码也在文件“stm32f10x.h”中。

代码清单 5.15　指向外设首地址的结构体指针

```
57  // GPIO 外设声明
58  #define GPIOA    ((GPIO_TypeDef *) GPIOA_BASE)
59  #define GPIOB    ((GPIO_TypeDef *) GPIOB_BASE)
60  // RCC 外设声明
61  #define RCC      ((RCC_TypeDef *) RCC_BASE)
62  /* RCC 的 AHB1 时钟使能寄存器地址,强制转换成指针 */
63  #endif           /* __STM32F10X_H */
```

首先通过强制类型转换把外设的基地址转换成 GPIO_TypeDef 类型的结构体指针，然后通过宏定义把 GPIOA、GPIOB 等定义成外设的结构体指针，通过外设的结构体指针就可以达到访问外设的寄存器的目的。

5. stm32f10x.gpio.c 文件

本工程中的文件 stm32f10x.gpio.c 是我们自己构建的，与标准库中的有所区别。该文件定义了 GPIO 初始化函数和位操作函数。

1）GPIO 初始化函数

对初始化结构体赋值后，把它输入 GPIO 初始化函数，由它来实现寄存器配置。GPIO 初始化函数见代码清单 5.16。

代码清单 5.16　GPIO 初始化函数

```
1   #include "stm32f10x_gpio.h"
2   /**
3     * 函数功能:初始化引脚模式
4     * 参数说明:GPIOx,该参数为 GPIO_TypeDef 类型的指针,指向 GPIO 端口的地址
5     * GPIO_InitTypeDef:GPIO_InitTypeDef 结构体指针,指向初始化变量
6   */
7   void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
8   {
9     uint32_t currentmode = 0x00, currentpin = 0x00, pinpos = 0x00, pos = 0x00;
10    uint32_t tmpreg = 0x00, pinmask = 0x00;
11
12    /* ------------------ GPIO 模式配置 -------------------- */
13    // 把输入参数 GPIO_Mode 的低 4 位暂存在 currentmode
14      currentmode = ((uint32_t)GPIO_InitStruct->GPIO_Mode) & ((uint32_t)0x0F);
15    // bit4 是 1 表示输出,bit4 是 0 则是输入
16    // 判断 bit4 是 1 还是 0,即首选判断是输入还是输出模式
17      if ((((uint32_t)GPIO_InitStruct->GPIO_Mode) & ((uint32_t)0x10)) != 0x00)
18      {
19
20        /* 输出模式则要设置输出速率 */
21        currentmode |= (uint32_t)GPIO_InitStruct->GPIO_Speed;
22      }
23
24  /* -------- GPIO CRL 寄存器配置 CRL 寄存器控制着低 8 位 IO- ---- */
25    /* 配置端口低 8 位,即 Pin0~Pin7 */
26    if (((uint32_t)GPIO_InitStruct->GPIO_Pin & ((uint32_t)0x00FF)) != 0x00)
27    {
28      //先备份 CRL 寄存器的值
29      tmpreg = GPIOx->CRL;
30      //循环,从 Pin0 开始配对,找出具体的 Pin
31      for (pinpos = 0x00; pinpos < 0x08; pinpos++)
```

```
    {
      // pos 的值为 1 左移 pinpos 位
      pos = ((uint32_t)0x01) << pinpos;

      /* 令 pos 与输入参数 GPIO_PIN 作位与运算 */
      currentpin = (GPIO_InitStruct->GPIO_Pin) & pos;

      //若 currentpin = pos,则找到使用的引脚
      if (currentpin == pos)
      {
        // pinpos 的值左移两位(乘以 4),因为寄存器中 4 个位配置一个引脚
        pos = pinpos << 2;
        /* 把控制这个引脚的 4 个寄存器位清零,其他寄存器位不变 */
        pinmask = ((uint32_t)0x0F) << pos;
        tmpreg &= ~pinmask;

        /* 向寄存器写入将要配置的引脚的模式 */
        tmpreg |= (currentmode << pos);

        /* 判断是否为下拉输入模式 */
        if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPD)
        {
          //下拉输入模式,引脚默认置 0,对 BRR 寄存器写 1 对引脚置 0
          GPIOx->BRR = (((uint32_t)0x01) << pinpos);
        }
        else
        {
          /* 判断是否为上拉输入模式 */
          if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPU)
          {
            //上拉输入模式,引脚默认值为 1,对 BSRR 寄存器写 1 对引脚置 1
            GPIOx->BSRR = (((uint32_t)0x01) << pinpos);
          }
        }
      }
    }
    把前面处理后的暂存值写入 CRL 寄存器之中
    GPIOx->CRL = tmpreg;
  }
/* --------GPIO CRH 寄存器配置 CRH 寄存器控制着高 8 位 IO-  ----- */
  /* 配置端口高 8 位,即 Pin8~Pin15 */
  if (GPIO_InitStruct->GPIO_Pin > 0x00FF)
  {
    //先备份 CRH 寄存器的值
    tmpreg = GPIOx->CRH;
    //循环,从 Pin8 开始配对,找出具体的 Pin
    for (pinpos = 0x00; pinpos < 0x08; pinpos++)
    {
      pos = (((uint32_t)0x01) << (pinpos + 0x08));
      /* pos 与输入参数 GPIO_PIN 做位与运算 */
      currentpin = ((GPIO_InitStruct->GPIO_Pin) & pos);
      //若 currentpin = pos,则找到使用的引脚
      if (currentpin == pos)
      {
        //pinpos 的值左移两位(乘以 4),因为寄存器中 4 个位配置一个引脚
        pos = pinpos << 2;
        /* 把控制这个引脚的 4 个寄存器位清零,其他寄存器位不变 */
        pinmask = ((uint32_t)0x0F) << pos;
```

```
90          tmpreg &= ~pinmask;
91          /* 向寄存器写入将要配置的引脚的模式 */
92          tmpreg |= (currentmode << pos);
93          /* 判断是否为下拉输入模式 */
94          if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPD)
95          {
96            //下拉输入模式,引脚默认置 0,对 BRR 寄存器写 1 可对引脚置 0
97            GPIOx->BRR = (((uint32_t)0x01) << (pinpos + 0x08));
98          }
99          /* 判断是否为上拉输入模式 */
100         if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPU)
101         {
102           //上拉输入模式,引脚默认值为 1,对 BSRR 寄存器写 1 可对引脚置 1
103           GPIOx->BSRR = (((uint32_t)0x01) << (pinpos + 0x08));
104         }
105       }
106     }
107     //把前面处理后的暂存值写入 CRH 寄存器之中
108     GPIOx->CRH = tmpreg;
109   }
110 }
```

这个函数有 GPIOx 和 GPIO_InitStruct 两个输入参数,分别是 GPIO 外设指针和 GPIO 初始化结构体指针,分别用来指定要初始化的 GPIO 端口及引脚的工作模式。要充分理解这个 GPIO 初始化函数,需结合 GPIO 引脚工作模式真值表(表 5.2)与注释来分析。

表 5.2 GPIO 引脚工作模式真值表

GPIOMode_Typedef		十六进制	二进制							
			bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
				上拉/下拉		输入/输出	工作模式依寄存器说明			
GPIO _ Mode _AIN	模拟输入	0x00								
GPIO_Mode_IN_FLOATING	浮空输入	0x04								
GPIO _ Mode _IPD	下拉输入	0x28								
GPIO _ Mode _IPU	上拉输入	0x48								
GPIO_Mode_OUT_OD	开漏输出	0x14								
GPIO_Mode_OUT_PP	推挽输出	0x10								
GPIO_Mode_AF_OD	复用开漏输出	0x1c								
GPIO_Mode_AF_PP	复用推挽输出	0x18								
这 8 个宏的高 4bit 可随意设置,只要能在程序上帮助判断出模式即可,真正写到寄存器的值是 bit2 和 bit3										

（1）取得 GPIO_Mode 的值，判断 bit4 是 1 还是 0 来确定是输出还是输入。若是输出，则设置输出速率，即加上 GPIO_Speed 的值；而输入没有速率之说，不用设置。

（2）配置 CRL 寄存器。通过 GPIO_Pin 的值计算出具体需要初始化哪个引脚，然后把需要配置的值写入 CRL 寄存器中，具体分析见代码注释。上拉/下拉输入不是直接通过配置某个寄存器来实现的，而是通过写 BSRR 或者 BRR 寄存器来实现的，如图 5.28 所示。因为《STM32F10xxx 参考手册》的寄存器说明中没有明确地指出如何配置上拉/下拉，只看手册没看固件库底层源码是弄不清楚的。

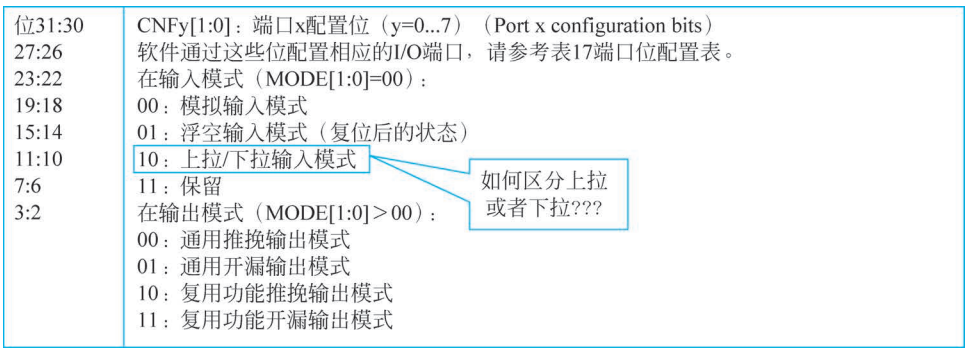


图 5.28　上拉下拉寄存器说明

2）定义位操作函数

使用函数来封装 GPIO 的基本操作，以后应用时不需要查询寄存器，而是直接通过调用这里定义的函数来实现。把针对 GPIO 外设操作的函数及其宏定义分别存放在“stm32f10x_gpio.c”和“stm32f10x_gpio.h”文件中，这两个文件需要自己新建。

在“stm32f10x_gpio.c”文件定义两个位操作函数，分别用于控制引脚输出高电平和低电平，见代码清单 5.17。

代码清单 5.17　GPIO 置位函数与复位函数的定义

```
1  /**
2    * 函数功能:设置引脚为高电平
3    * 参数说明:GPIOx:该参数为 GPIO_TypeDef 类型的指针,指向 GPIO 端口的地址
4    *          GPIO_Pin:选择要设置的 GPIO 端口引脚,可输入宏 GPIO_Pin_0-15,
5    *          表示 GPIOx 端口的 0～15 号引脚.
6   */
7  void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
8  {
9    /*  设置 GPIOx 端口 BSRR 寄存器的第 GPIO_Pin 位,使其输出高电平*/
10   /*  因为 BSRR 寄存器写 0 不影响,
11    *  宏 GPIO_Pin 只是对应位为 1,其他位均为 0,所以可以直接赋值*/
12   GPIOx->BSRR = GPIO_Pin;
13 }
14 /**
15   * 函数功能:设置引脚为低电平
16   * 参数说明:GPIOx:该参数为 GPIO_TypeDef 类型的指针,指向 GPIO 端口的地址
```

```
17      *           GPIO_Pin:选择要设置的 GPIO 端口引脚,可输入宏 GPIO_Pin_0 - 15,
18      *           表示 GPIOx 端口的 0～15 号引脚.
19    */
20    void GPIO_ResetBits(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)
21    {
22      /*  设置 GPIOx 端口 BRR 寄存器的第 GPIO_Pin 位,使其输出低电平 */
23      /*  因为 BRR 寄存器写 0 不影响,
24       *  宏 GPIO_Pin 只是对应位为 1,其他位均为 0,所以可以直接赋值 */
25      GPIOx->BRR = GPIO_Pin;
26    }
```

这两个函数体内都是只有一个语句,对 GPIOx 的 BSRR 或 BRR 寄存器赋值,从而设置引脚为高电平或低电平,操作 BSRR 或者 BRR 可以实现单独地操作某一位,有关这两个的寄存器说明见图 5.29 和图 5.30。其中 GPIOx 是一个指针变量,通过函数的输入参数可以修改它的值,如给它赋予 GPIOA、GPIOB、GPIOH 等结构体指针值,这个函数就可以控制相应的 GPIOA、GPIOB、GPIOH 等端口的输出。

端口位设置/清除寄存器（GPIOx_BSRR）（x=A..E）

偏移地址：0x10
复位值：0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位31:16	BRy：清除端口x的位y（y=0...15）（Port x Reset bit y） 这些位只能写入并只能以字（16位）的形式操作。 0：对对应的ODRy位不产生影响 1：清除对应的ODRy位为0 注：如果同时设置了BSy和BRy的对应位，BSy位起作用。
位15:0	BSy：设置端口x的位y（y=0...15）（Port x Set bit y） 这些位只能写入并只能以字（16位）的形式操作。 0：对对应的ODRy位不产生影响 1：设置对应的ODRy位为1

图 5.29　BSRR 寄存器说明

利用这两个位操作函数可以方便地操作各种 GPIO 的引脚电平,如代码语句“GPIO_SetBits(GPIOB,(uint16_t)(1<<10));”,控制 GPIOB 的引脚 10 输出高电平。使用以上函数输入参数设置引脚号时还是稍感不便,为此把表示 16 个引脚的操作数都定义成宏,这个宏定义在文件“stm32f10x_gpio.h”中。

6. stm32f10x_gpio.h 文件

1）引脚号定义

GPIO 引脚号定义在 stm32f10x_gpio.h 头文件中,见代码清单 5.18。

端口位清除寄存器（GPIOx_BRR）（x=A..E）

偏移地址：0x14
复位值：0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位31:16	保留。
位15:0	BRy：清除端口x的位y（y=0...15）（Port x Reset bit y） 这些位只能写入并只能以字（16位）的形式操作。 0：对对应的ODRy位不产生影响 1：清除对应的ODRy位为0

图 5.30　BRR 寄存器说明

代码清单 5.18　选择引脚参数的宏

```
#ifndef __STM32F10X_GPIO_H
#define __STM32F10X_GPIO_H

#include "stm32f10x.h"

/* GPIO 引脚号定义 */
#define GPIO_Pin_0   (uint16_t)0x0001)   /*!< 选择 Pin0 (1 << 0) */
#define GPIO_Pin_1   ((uint16_t)0x0002)  /*!< 选择 Pin1 (1 << 1) */
#define GPIO_Pin_2   ((uint16_t)0x0004)  /*!< 选择 Pin2 (1 << 2) */
#define GPIO_Pin_3   ((uint16_t)0x0008)  /*!< 选择 Pin3 (1 << 3) */
#define GPIO_Pin_4   ((uint16_t)0x0010)  /*!< 选择 Pin4 */
#define GPIO_Pin_5   ((uint16_t)0x0020)  /*!< 选择 Pin5 */
#define GPIO_Pin_6   ((uint16_t)0x0040)  /*!< 选择 Pin6 */
#define GPIO_Pin_7   ((uint16_t)0x0080)  /*!< 选择 Pin7 */
#define GPIO_Pin_8   ((uint16_t)0x0100)  /*!< 选择 Pin8 */
#define GPIO_Pin_9   ((uint16_t)0x0200)  /*!< 选择 Pin9 */
#define GPIO_Pin_10  ((uint16_t)0x0400)  /*!< 选择 Pin10 */
#define GPIO_Pin_11  ((uint16_t)0x0800)  /*!< 选择 Pin11 */
#define GPIO_Pin_12  ((uint16_t)0x1000)  /*!< 选择 Pin12 */
#define GPIO_Pin_13  ((uint16_t)0x2000)  /*!< 选择 Pin13 */
#define GPIO_Pin_14  ((uint16_t)0x4000)  /*!< 选择 Pin14 */
#define GPIO_Pin_15  ((uint16_t)0x8000)  /*!< 选择 Pin15 */
#define GPIO_Pin_All ((uint16_t)0xFFFF)  /*!< 选择全部引脚 */
```

这些宏代表的参数是某位置“1”、其他位置“0”的数值，其中最后一个“GPIO_Pin_ALL”是所有数据位都为“1”，所以用它可以一次控制设置整个端口的 0～15 所有引脚。利用这些宏对 GPIOB 的引脚 10 输出高电平的控制可以改写成“GPIO_SetBits(GPIOB, GPIO_Pin_10);”。使用以上代码控制 GPIO 不需要再看寄存器，直接从函数名和输入参数就可以直观看出这个语句要实现什么操作。

2）定义初始化结构体 GPIO_InitTypeDef

在控制 GPIO 输出电平前需要初始化 GPIO 引脚的各种模式，这部分代码涉及的寄

存器很多。为此，先根据 GPIO 初始化时涉及的初始化参数以结构体的形式封装起来，声明一个名为 GPIO_InitTypeDef 的结构体类型，见代码清单 5.19。

代码清单 5.19　定义 GPIO 初始化结构体

```
25  typedef struct
26  {
27  uint16_t GPIO_Pin;    /*!< 选择要配置的 GPIO 引脚 */
28
29  uint16_t GPIO_Speed; /*!< 选择 GPIO 引脚的速率 */
30
31  uint16_t GPIO_Mode;   /*!< 选择 GPIO 引脚的工作模式 */
32  } GPIO_InitTypeDef;
```

这个结构体中包含了初始化 GPIO 所需要的信息，如引脚号、工作模式和输出速率。设计这个结构体的思路是初始化 GPIO 前先定义一个这样的结构体变量，根据需要配置的 GPIO 模式对这个结构体的各个成员进行赋值，然后把这个变量作为“GPIO 初始化函数”的输入参数，该函数能根据这个变量值中的内容去配置寄存器，从而实现 GPIO 的初始化。

3）定义引脚模式的枚举

上面定义的结构体很直接，不足之处是在对结构体中各个成员赋值实现某个功能时还需要查询手册的寄存器说明。GPIO_Speed 和 GPIO_Mode 对应的寄存器是端口配置寄存器 CRL 和 CRH，具体见图 5.31 和图 5.32。

偏移地址：0x00
复位值：0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	CNFy[1:0]：端口x配置位（y=0...7）（Port x configuration bits） 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式（MODE[1:0]=00）： 00：模拟输入模式 01：浮空输入模式（复位后的状态） 10：上拉/下拉输入模式 11：保留 在输出模式（MODE[1:0]＞00）： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位29:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0	MODEy[1:0]：端口x的模式位（y=0...7）（Port x mode bits） 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00：输入模式（复位后的状态） 01：输出模式，最大速率10MHz 10：输出模式，最大速率2MHz 11：输出模式，最大速率50MHz

图 5.31　端口配置低寄存器（GPIOx_CRL）（x＝A..E）

偏移地址：0x04
复位值：0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	CNFy[1:0]：端口x配置位（y=8...15）（Port x configuration bits） 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式（MODE[1:0]=00）： 00：模拟输入模式 01：浮空输入模式（复位后的状态） 10：上拉/下拉输入模式 11：保留 在输出模式（MODE[1:0]＞00）： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位29:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0	MODEy[1:0]：端口x的模式位（y=8...15）（Port x mode bits） 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00：输入模式（复位后的状态） 01：输出模式，最大速率为10MHz 10：输出模式，最大速率为2MHz 11：输出模式，最大速率为50MHz

图 5.32　端口配置高寄存器(GPIOx_CRH)（x＝A..E）

人们不希望每次用到时都要去查询手册，可以使用 C 语言中的枚举定义功能，根据手册把每个成员的所有取值都定义好，具体见代码清单 5.20。

代码清单 5.20　GPIO 枚举类型定义

```
34  /*
35   * GPIO 输出速率枚举定义
36   */
37  typedef enum
38  {
39  GPIO_Speed_10MHz = 1,// 10MHZ (01)b
40  GPIO_Speed_2MHz,// 2MHZ (10)b
41  GPIO_Speed_50MHz// 50MHZ (11)b
42  } GPIOSpeed_TypeDef;
43  /**
44     * GPIO 工作模式枚举定义
45   */
46  typedef enum
47  {
48  GPIO_Mode_AIN = 0x0,          // 模拟输入(0000 0000)b
49  GPIO_Mode_IN_FLOATING = 0x04,// 浮空输入(0000 0100)b
50  GPIO_Mode_IPD = 0x28,         // 下拉输入(0010 1000)b
```

```
51  GPIO_Mode_IPU = 0x48,          // 上拉输入(0100 1000)b
52
53  GPIO_Mode_Out_OD = 0x14,       // 开漏输出(0001 0100)b
54  GPIO_Mode_Out_PP = 0x10,       // 推挽输出(0001 0000)b
55  GPIO_Mode_AF_OD = 0x1C,        // 复用开漏输出 (0001 1100)b
56  GPIO_Mode_AF_PP = 0x18         // 复用推挽输出 (0001 1000)b
57  } GPIOMode_TypeDef;
58
59  void GPIO_SetBits(GPIO_TypeDef * GPIOx,uint16_t GPIO_Pin);
60  void GPIO_ResetBits( GPIO_TypeDef * GPIOx,uint16_t GPIO_Pin );
61  void GPIO_Init(GPIO_TypeDef * GPIOx, GPIO_InitTypeDef * GPIO_InitStruct);
62
63  #endif /* __STM32F10X_GPIO_H */
```

下面分析这两个枚举类型的值如何跟端口控制寄存器里面的说明对应起来。有关速率的枚举类型有(01)b10MHz、(10)b2MHz 和(11)b50MHz,这三个值跟寄存器说明对得上,很容易理解。模式的枚举类型的值理解起来有些难。可以通过表 5.2 梳理,帮助理解。

单从这些枚举值的十六进制来看很难发现规律,转化成二进制之后就比较容易发现规律。bit4 用来区分端口是输入还是输出,0 表示输入,1 表示输出,bit2 和 bit3 对应寄存器的 CNFY[1:0]位,是真正要写入到端口控制寄存器 CRL 和 CRH 中的值。bit0 和 bit1 对应寄存器的 MODEY[1:0]位,这里暂不初始化,在 GPIO_Init()初始化函数中用来和 GPIOSpeed 的值相加即可实现速率的配置。有关具体的代码分析见 GPIO_Init()库函数。其中在下拉输入和上拉输入中设置 bit5 和 bit6 的值为 01 和 10 来以示区别。

有了这些枚举定义,GPIO_InitTypeDef 结构体就可以使用枚举类型来限定输入参数。如果不使用枚举类型,仍使用“uint16_t”类型来定义结构体成员,成员值的范围就是 0～255,实际上这些成员只能输入几个数值。因此,使用枚举类型可以对结构体成员起到限定输入的作用,只能输入相应已定义的枚举值。

此外,59 行、60 行、61 行声明了 GPIO_SetBits、GPIO_ResetBits、GPIO_Init 三个函数。

7. stm32f10x_rcc.c 与 stm32f10x_rcc.h 文件

文件“stm32f10x_rcc.c”程序见代码清单 5.21。

代码清单 5.21　文件“stm32f10x_rcc.c”程序代码

```
1   #include "stm32f10x_rcc.h"
2
3   void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState)
4   {
5
6     if (NewState != DISABLE)
7     {
8       RCC->APB2ENR |= RCC_APB2Periph;
9     }
10    else
11    {
12      RCC->APB2ENR &= ~RCC_APB2Periph;
13    }
14  }
```

文件“stm32f10x_rcc.c”定义了函数 RCC_APB2PeriphClockCmd，用于使能或失能 APB2 外设时钟。

文件“stm32f10x_rcc.h”程序见代码清单 5.22。

代码清单 5.22　文件“stm32f10x_rcc.h”程序代码

```
#ifndef __STM32F10x_RCC_H
#define __STM32F10x_RCC_H

#include "stm32f10x.h"

#define RCC_APB2Periph_GPIOA ((uint32_t)0x00000004)
#define RCC_APB2Periph_GPIOB ((uint32_t)0x00000008)

void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState);

#endif
```

文件“stm32f10x_rcc.c”声明了函数 RCC_APB2PeriphClockCmd，并对 RCC_APB2Periph_GPIOA 与 RCC_APB2Periph_GPIOB 做了声明。