第5章 深入了解项目模板

5.1 启动文件的作用


教学视频

在项目中配置 Run-Time Environment 时，必须选中 CMSIS 下的 CORE 组件，以及 Device 下的 Startup 组件。Startup 组件与 CORE 组件之间存在依存关系，两者必须同时选中。选中了 Startup 组件后，Keil 软件会根据项目所选择单片机芯片的类型，添加相应的启动文件。

本书所用的 STM32F103VET6 单片机为高密度单片机，项目中添加的启动文件为 startup_stm32f10x_hd.s。文件名中的 hd 是 high density 的首字母缩写，高密度指片上 Flash 存储器容量为 256～512KB 的大容量单片机，根据存储器容量，单片机分为小容量、中容量和大容量 3 个系列。

启动文件用汇编语言编写，文件扩展名为.s。启动文件实现了以下 3 个基本功能：

(1) 定义了栈区(stack)和堆区(heap)；

(2) 定义了中断向量表；

(3) 定义了中断处理函数。

5.1.1 定义栈和堆

启动文件中定义了栈和堆的大小。栈区也常常被称作“堆栈”，不要将堆栈与堆混淆。

堆栈(stack)按“先进后出”原则进行操作。将数据存入堆栈称为“压栈”。从堆栈中将数据弹出称为“弹栈”。CPU 中有一个专门的寄存器，堆栈指针寄存器 SP，指示堆栈栈顶位置，压栈、弹栈操作始终在 SP 所指的堆栈栈顶位置进行。对堆栈的操作犹如一个单端口的“弹夹”，我们将数据“子弹”连续压入堆栈“弹夹”中，最先压入的数据“子弹”在最下方。射击时最上方的数据“子弹”最先弹出“弹夹”。

堆栈是软件运行的基础，函数的调用与返回、中断的响应与返回都需要利用堆栈。调用函数时将返回地址压入堆栈，然后跳转到函数体去执行。函数结束返回时从堆栈中弹出返回地址，回到函数调用的位置继续向下执行。此外，函数内部的局部变量也在堆栈中分配存储空间，所以函数嵌套调用或中断嵌套时，嵌套层数越多，对堆栈的消耗越大。如果堆栈“溢出”(overflow)，那么程序也就“飞死”了。

堆(heap)用于动态内存分配。在程序运行过程中调用 malloc()函数动态分配内存时，就是从堆中分配存储空间。动态分配的内存的生存周期由开发人员决定，使用完毕时必须调用 free()函数主动释放，否则就会出现内存泄漏现象。

在单片机程序中，栈和堆都是在片上 SRAM 存储器中分配存储空间，编译时按启动文件中定义的大小划分堆和栈的存储空间。STM32F103VET6 单片机内部集成有 64KB 的 SRAM，栈和堆都是从这 64KB 的 SRAM 中分配的。启动文件中定义栈和堆的相关代码截图见图 5.1。

```
startup_stm32f10x_hd.s | readme.txt | main.c

34
35  Stack_Size      EQU     0x00000400
36
37                  AREA    STACK, NOINIT, READWRITE, ALIGN=3
38  Stack_Mem       SPACE   Stack_Size
39  __initial_sp
40
41  ; <h> Heap Configuration
42  ;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
43  ; </h>
44
45  Heap_Size       EQU     0x00000200
46
47                  AREA    HEAP, NOINIT, READWRITE, ALIGN=3
48  __heap_base
49  Heap_Mem        SPACE   Heap_Size
50  __heap_limit
```

图 5.1　启动文件中关于栈和堆的定义

汇编语言中用分号“；”添加注释信息。Keil 软件用不同的颜色标识各类信息，汇编语言的关键字用蓝色，立即数为红色，而自己定义的符号为黑色，注释信息为灰色。虽然启动文件中所有关键字都是全大写，但是汇编语言是不区分字母大小写的编程语言。

启动文件中首先用伪指令 EQU 定义了 Stack_Size，指定堆栈大小。然后用汇编语言伪指令 AREA 定义了 STACK 段。伪指令 SPACE 用于保留指定字节数的存储空间，堆栈大小设定为 Stack_Size，即 0x00000400。关键字 NOINIT 说明不初始化这个段的存储单元。汇编语句 ALIGN＝3 说明地址按 2^3，即 8 字节对齐。定义了堆栈之后，接着用类似的方式定义了堆。

用 SPACE 伪指令定义了堆栈区大小后，立即定义了符号__initial_sp，这个符号代表了堆栈区尾部的地址。上电复位时用__initial_sp 初始化堆栈指针寄存器 SP，复位后寄存器 SP 指向堆栈区的末尾。压栈时堆栈指针 SP 减小，数据存入堆栈，而弹栈时取出数据，SP 增加。随着压栈弹栈操作，SP 寄存器始终指向堆栈栈顶。

通过仿真器调试程序时，进入 Debug 环境后，打开 Registers 窗口，可以查看内核寄存器，观察 SP 寄存器的变化情况，了解对堆栈存储空间的使用情况，如图 5.2 所示。

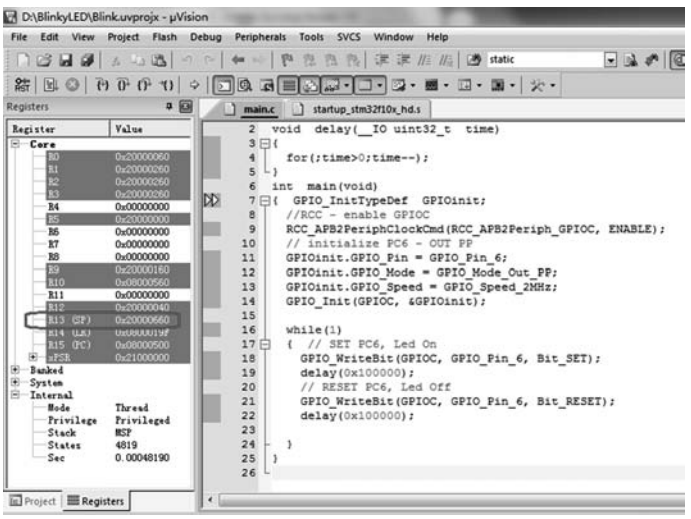


图 5.2　堆栈指针寄存器 SP

5.1.2 定义中断向量表

中断向量指中断处理函数的入口地址。系统中每个中断源有一个对应的中断处理函数，这个中断处理函数的入口地址就是对应的中断向量，所有的中断向量合在一起形成了中断向量表。对于单片机来说，整个中断系统是预先定义好的。

启动文件中用伪指令 DCD 定义了中断向量表，其中包含内核异常(Exception)处理函数以及片上硬件中断(Interrupt)处理函数的名称，如图 5.3 所示。DCD 说明定义的数据项占 4 字节，后面的函数名就代表了函数入口地址。由于整个地址空间为 4GB，地址位宽为 32 位，所以需要 4 字节存放函数入口地址。

main.c | startup_stm32f10x_hd.s

```
51
52                PRESERVE8
53                THUMB
54
55
56  ; Vector Table Mapped to Address 0 at Reset
57                AREA    RESET, DATA, READONLY
58                EXPORT  __Vectors
59                EXPORT  __Vectors_End
60                EXPORT  __Vectors_Size
61
62  __Vectors     DCD     __initial_sp              ; Top of Stack
63                DCD     Reset_Handler             ; Reset Handler
64                DCD     NMI_Handler               ; NMI Handler
65                DCD     HardFault_Handler         ; Hard Fault Handler
66                DCD     MemManage_Handler         ; MPU Fault Handler
67                DCD     BusFault_Handler          ; Bus Fault Handler
68                DCD     UsageFault_Handler        ; Usage Fault Handler
69                DCD     0                         ; Reserved
70                DCD     0                         ; Reserved
71                DCD     0                         ; Reserved
72                DCD     0                         ; Reserved
73                DCD     SVC_Handler               ; SVCall Handler
74                DCD     DebugMon_Handler          ; Debug Monitor Handler
75                DCD     0                         ; Reserved
76                DCD     PendSV_Handler            ; PendSV Handler
77                DCD     SysTick_Handler           ; SysTick Handler
```

图 5.3 中断向量表

定义了中断向量表后，接着定义所有的异常以及中断处理函数。汇编语言中将函数称为“子程序”。伪指令 PROC 和 ENDP 分别定义子程序的开始和结束。除了复位处理子程序 Reset_Handler 以外，其他所有的中断处理子程序都没有实现任何具体功能，只是一个“死循环”，发生异常时就会陷入对应的异常处理子程序中。相关代码截图见图 5.4。

从图 5.4 中可以看到，NMI_Handler 子程序只有一条可执行的指令，汇编指令“B .”的功能是跳转到自身。一旦发生 NMI 中断请求，系统自动响应中断，调用 NMI 处理函数时就会一直执行这条“B .”指令，陷入死循环(infinite loop)。

所有异常和中断处理函数的函数名都用伪指令 EXPORT 输出，并声明为[WEAK]。用[WEAK]说明的符号可以在其他源文件中重新定义，而不会产生错误。如果没有在其他源文件中实现，则以启动文件中的定义作为默认的函数实现。

总的来说，启动文件中定义了中断向量表，规定了每个中断处理函数的函数名，并给出了一个不完成任何功能，仅仅是死循环的函数定义。这个默认的函数定义是可以被改变的，开发人员可以自行在其他源文件中重新实现中断处理函数，但是应注意函数名称必须与中断向量表中的函数名称一致。

main.c | startup_stm32f10x_hd.s

```
143
144                 AREA    |.text|, CODE, READONLY
145
146 ; Reset handler
147 Reset_Handler   PROC
148                 EXPORT  Reset_Handler             [WEAK]
149                 IMPORT  __main
150                 IMPORT  SystemInit
151                 LDR     R0, =SystemInit
152                 BLX     R0
153                 LDR     R0, =__main
154                 BX      R0
155                 ENDP
156
157 ; Dummy Exception Handlers (infinite loops which can be modified)
158
159 NMI_Handler     PROC
160                 EXPORT  NMI_Handler               [WEAK]
161                 B       .
162                 ENDP
163 HardFault_Handler\
164                 PROC
165                 EXPORT  HardFault_Handler         [WEAK]
166                 B       .
167                 ENDP
```

图 5.4　异常处理子程序

5.1.3　定义复位中断子程序

复位时会触发复位中断，内核响应复位中断请求，查中断向量表，跳转到复位中断处理子程序，即 Reset_Handler 去执行。启动文件中复位中断子程序 Reset_Handler 的相关定义见图 5.5。

```
; Reset handler
Reset_Handler   PROC
                EXPORT  Reset_Handler       [WEAK]
                IMPORT  __main
                IMPORT  SystemInit
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

图 5.5　复位处理子程序

Reset_Handler 子程序中首先用伪指令 EXPORT 和 IMPORT 完成了符号的输出和输入。输入的符号__main 和 SystemInit 实际上都是函数名，__main 是系统定义的 C 语言的主函数，而 SystemInit 是在 Startup 组件的库函数文件 system_stm32f10x.c 中定义的系统初始化函数，该函数完成系统时钟 SYSCLK，以及 AHB、APB1、APB2 总线时钟的初始化。

复位子程序的可执行代码只有 4 条，前两条指令调用了 SystemInit()函数，完成系统时钟初始化。后两条指令跳转到__main()函数去执行。系统提供的__main()函数完成库函数的初始化，并且初始化应用程序的执行环境，最后跳转到 main()函数，就进入了用户所编写的程序。初始化步骤如图 5.6 所示，https://www.keil.com/support/man/docs/armclang_intro/armclang_intro_asa1505906246660.htm 讲解了进入用户编写的 main()函数之前系统完成的初始化工作。

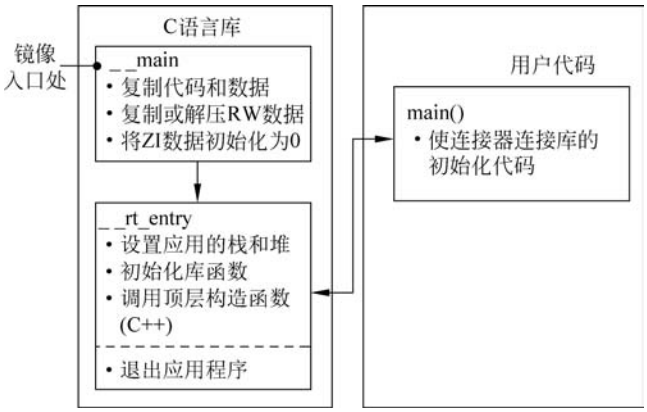


图 5.6　系统初始化

设置仿真配置时，如果在 Options for Target 'BlinkyLED'窗口的 Debug 标签页中，取消选中 Run to main()复选框，那么进入 Debug 环境时，会停留在 Reset_Handler 子程序中。如果选中了 Run to main()复选框，那么进入 Debug 环境时，会直接运行到 main()函数才停止，此时系统初始化工作已经完成，进入到用户编写的代码了，如图 5.7 所示。

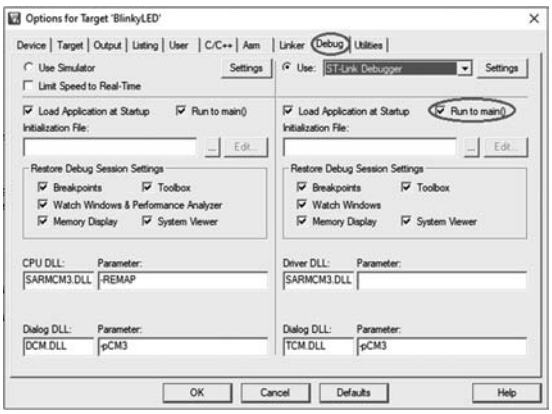


图 5.7　项目中的仿真器配置

5.2　单片机的时钟初始化

单片机内核的工作速度受系统时钟频率控制。Cortex-M3 内核时钟频率最高可达 72MHz，STMicroelectronics 公司生产的 STM32F10x 系列单片机都采用了 Cortex-M3 内核，但是 STM32F101 系列单片机的主频限定为 36MHz，STM32F102 系列单片机的主频最高只有 48MHz，只有 STM32F103 系列单片机的工作速度才真正达到最高频率 72MHz。

STM32F10x 系列单片机的系统时钟 SYSCLK 有 3 个来源：内部高速晶振(High-Speed Internal Oscillator，HSI)、外部高速晶振(High-Speed External Oscillator，HSE)和锁相环时钟 PLLCLK。

内部高速晶振的频率为 8MHz，但内部晶振的精度较低，无法满足对时间精度要求高的硬件模块的需求，实际应用中很少使用它。单片机开发板上大多配置有高精度的外部晶振，但是 STM32F10x 单片机限定外部晶振频率只能为 4～16MHz，远远低于 72MHz，不适宜直接将外部高速晶振作为系统时钟。

单片机内部集成有锁相环 PLL 模块。锁相环 PLL 模块能够将输入的时钟信号倍频，输出更高频率的高品质时钟信号，因此可以将高速外部晶振作为锁相环 PLL 的输入时钟信号，经过倍频后，使锁相环的输出时钟信号 PLLCLK 频率达到 72MHz。将 72MHz 频率的锁相环时钟 PLLCLK 设置为系统时钟 SYSCLK。

目前比较常见的一种时钟配置方案是采用 8MHz 的外部晶振，通过单片机内部的锁相环 PLL，实现 9 倍倍频，产生 72MHz 的锁相环时钟 PLLCLK，选择 PLLCLK 作为系统时钟 SYSCLK。单片机参考手册中的“复位与时钟控制 RCC”相关章节详细说明了如何配置时钟，给出了时钟树示意图，如图 5.8 所示。

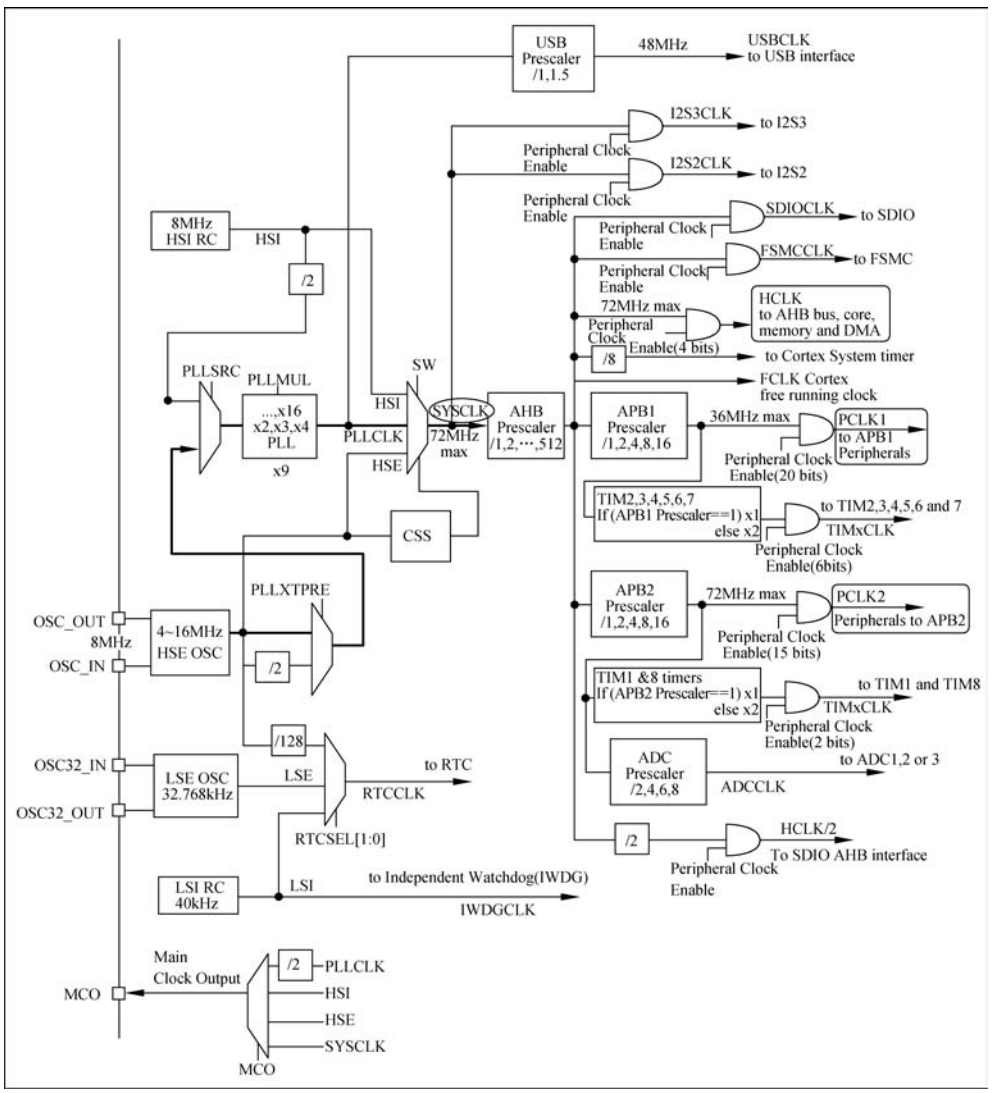


图 5.8　时钟树示意图

图 5.8 中的 OSC_IN 和 OSC_OUT 为单片机芯片的两个引脚，用于连接外部高速晶振。外部晶振为 8MHz，通过配置 PLLXTPRE 和 PLLSRC 多路开关，可以选择直接将外部时钟 HSE 作为锁相环的输入时钟；然后将锁相环的倍频系数 PLLMUL 设置为 9，使锁相环 PLL 实现 9 倍倍频，产生 72MHz 的 PLLCLK；最后通过配置 SW 多路开关，选择 PLLCLK 作为系统时钟 SYSCLK。图 5.8 中的多路开关 PLLXTPRE、PLLSRC、SW，以及倍频系数 PLLMUL 都是 RCC 模块中时钟控制寄存器 CR 和时钟配置寄存器 CFGR 中的控制位。初始化系统时钟时需要正确配置这些寄存器，才能完成系统时钟的初始化。

系统时钟 SYSCLK 经过 AHB 预分频后产生 HCLK 时钟信号。HCLK 频率最高为 72MHz，HCLK 时钟提供给 AHB 总线、内核、存储器以及 DMA。AHB 预分频后的时钟信号再分别经过 APB1 预分频和 APB2 预分频，得到 PCLK1 和 PCLK2 时钟信号。PCLK1 时钟信号即为 APB1 总线时钟，APB1 为低速总线，它的时钟频率最大只能为 36MHz。PCLK2 时钟信号即为 APB2 总线时钟，APB2 总线时钟频率最高可以与 AHB 总线一致。

若系统时钟 SYSCLK 设置为 72MHz，将 AHB 预分频系数设置为 1，那么分频后 HCLK 时钟频率与 SYSCLK 一致，都为 72MHz。将 APB1 预分频系数设置为 2，而 APB2 预分频系数设置为 1，PCLK1 频率为 36MHz，而 PCLK2 频率为 72MHz，使 APB1 和 APB2 总线都达到各自的最高频率。

复位时 Cortex-M3 内核默认直接将 8MHz 的内部高速时钟 HSI 作为系统时钟，时钟频率远远低于系统的最大工作频率，因此复位后的首要工作就是完成时钟初始化，使单片机工作在最高频率下。

系统提供的启动文件中实现了复位中断处理函数 Reset_Handler。复位时硬件触发复位中断请求，内核响应中断请求，自动转去执行 Reset_Handler 中断处理函数，完成时钟初始化以及系统初始化工作后，跳转到用户编写的 main()函数。

复位中断处理函数中首先调用 SystemInit()函数完成时钟初始化工作，该函数改写 RCC 模块的相关寄存器，完成锁相环配置，使锁相环时钟频率达到 72MHz，并选择锁相环时钟 PLLCLK 作为系统时钟 SYSCLK，然后设置 AHB、APB1 和 APB2 的分频系数，完成 3 个总线的时钟初始化。

运行 Keil 开发软件，打开本书配套资料中的项目 BlinkyLED。在 Options for Target 窗口的 C/C++标签页中预定义了 STM32F10X_HD 符号，说明单片机类型为大容量。根据项目所用单片机的类型，stm32f10x.h 头文件中在条件编译语句里定义了宏 HSE_VALUE，声明外部晶振频率为 8MHz。然后在 system_stm32f10x.c 文件中在条件编译语句里定义了宏 SYSCLK_FREQ_72MHz，说明系统时钟频率为 72MHz，相关代码见图 5.9。SystemInit()函数调用 SetSysClock()函数，根据这些宏定义，完成时钟初始化。

SystemInit()函数定义如下。为适应不同单片机类型，函数中用条件编译语句针对单片机类型，编译不同的代码段。本书所用单片机为 STM32F10X_HD 类型，为节省篇幅，将函数中针对其他类型单片机的代码用省略号代替。

stm32f10x.h

```
#if !defined  HSE_VALUE
 #ifdef STM32F10X_CL
  #define HSE_VALUE    ((uint32_t)25000000)
 #else
  #define HSE_VALUE    ((uint32_t)8000000) ,
 #endif /* STM32F10X_CL */
#endif /* HSE_VALUE */
```

system_stm32f10x.c

```
#if defined (STM32F10X_LD_VL) || (defined STM32F10X_MD_VL) || (defined STM32F10X_HD_VL)
/* #define SYSCLK_FREQ_HSE    HSE_VALUE */
 #define SYSCLK_FREQ_24MHz  24000000
#else
/* #define SYSCLK_FREQ_HSE    HSE_VALUE */
/* #define SYSCLK_FREQ_24MHz  24000000 */
/* #define SYSCLK_FREQ_36MHz  36000000 */
/* #define SYSCLK_FREQ_48MHz  48000000 */
/* #define SYSCLK_FREQ_56MHz  56000000 */
#define SYSCLK_FREQ_72MHz  72000000
#endif
```

图 5.9　时钟初始化的相关宏定义

```
void SystemInit (void)
{
  RCC->CR |= (uint32_t)0x00000001;    //按位相或,将 CR 寄存器 d0 位置 1,使能内部高速
                                      //时钟 HSI
#ifndef STM32F10X_CL
  RCC->CFGR &= (uint32_t)0xF8FF0000; //按位相与,将指定位清 0,设置 CFGR 寄存器
#else
  …
#endif /* STM32F10X_CL */
  RCC->CR &= (uint32_t)0xFEF6FFFF;    //将 CR 寄存器中的 HSEON, CSSON 和 PLLON 位清 0
  RCC->CR &= (uint32_t)0xFFFBFFFF;    //将 CR 寄存器中的 HSEBYP 位清 0
  //将 CFGR 寄存器中的 PLLSRC, PLLXTPRE, PLLMUL 和 USBPRE 位清 0
  RCC->CFGR &= (uint32_t)0xFF80FFFF;
#ifdef STM32F10X_CL
  …
#elif defined (STM32F10X_LD_VL)||defined (STM32F10X_MD_VL)||(defined STM32F10X_HD_VL)
  …
#else
  RCC->CIR = 0x009F0000;              //清除悬挂的中断标志位,即将中断标志位清 0
#endif /* STM32F10X_CL */
#if defined (STM32F10X_HD) || (defined STM32F10X_XL) || (defined STM32F10X_HD_VL)
  #ifdef DATA_IN_ExtSRAM
    SystemInit_ExtMemCtl();      //如果扩展了外部 SRAM,才调用该函数进行初始化
  #endif /* DATA_IN_ExtSRAM */
#endif
SetSysClock();    //设置系统时钟频率、HCLK、PCLK2 和 PCLK1 预分频系数,配置 Flash 接口参数
#ifdef VECT_TAB_SRAM
  SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; /* 将向量表重定位到内部 SRAM 中 */
#else
  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* 将向量表重定位到内部 FLASH 中 */
#endif
}
```

SystemInit()函数配置 RCC 中的相关寄存器，初始化 Flash 接口和锁相环 PLL，设置内核时钟、系统时钟 SYSCLK，以及 HCLK、PCLK1 和 PCLK2 时钟。这个函数只在复位中断处理函数 Reset_Handler 中调用一次，完成时钟初始化，此后单片机内部总线时钟就已经设定好了。

单片机内部总线时钟初始化完毕后，挂接在总线上的片上外设才能正常工作。出于降低功耗的目的，复位后在默认情况下，只有 Flash 接口和 SRAM 的时钟处于使能状态，其他片上外设时钟均为禁止状态。若不使能片上外设的时钟，则片上外设不工作，也就不产生功耗。

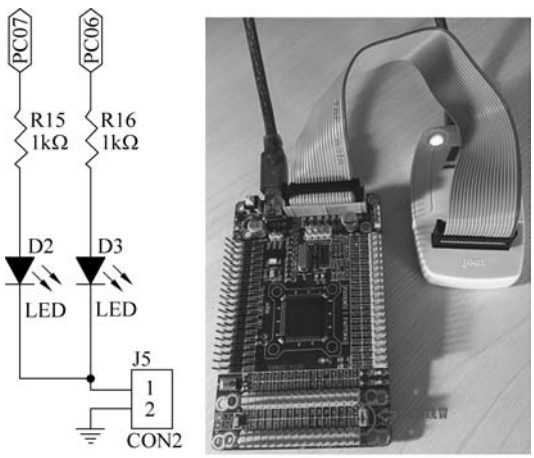
RCC 模块为 AHB、APB1 和 APB2 总线各提供了一个时钟使能寄存器，即 AHBENR、APB1ENR 和 APB2ENR 寄存器。寄存器中的每个二进制位控制总线上一个片上外设的时钟，将对应位置 1 使能时钟，清 0 则禁止时钟。

开发具体的嵌入式系统时，只会用到单片机内部的部分硬件资源。这时应根据项目需求，使能所用到的片上外设的时钟。需要使用某个片上外设时，首先应该调用 RCC 模块的库函数，使能该外设的时钟；然后才能调用硬件模块的库函数，完成片上外设的初始化。

例 5.1：参考例程 BlinkyLED 的分析与调试。

硬件连接：PC6 引脚通过 1kΩ 电阻接 LED 小灯的阳极，小灯阴极接 GND。

单片机最小板上将 PC6 和 PC7 引脚分别通过 1kΩ 电阻接到 2 个贴片发光二极管的阳极，只要将 J5 插针的跳帽接上，发光二极管 D1、D2 的阴极就连接了 GND，如图 5.10 所示。



(a) 最小板电路图　　(b) 硬件实物图

图 5.10　硬件连接示意图

软件分析：通过 GPIOC 模块的 PC6 引脚控制一个 LED 小灯，实现亮灭闪烁的效果。

项目中除了 Keil 软件添加的 CMSIS 库文件外，只编写了 main.c 源文件，main()函

数中首先完成了 GPIOC 模块中的 IO 引脚 PC6 的初始化，然后在 while(1)循环体中控制 PC6 引脚输出高、低电平，控制 LED 小灯亮灭，实现小灯闪烁效果，相关代码如下。

BlinkyLED 项目 main.c 源文件。

```
#include "stm32f10x.h"
void delay(__IO uint32_t time)
{   for(;time>0;time--);  }
int main(void)
{   GPIO_InitTypeDef GPIOinit;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);   //RCC 函数, 使能 GPIOC 时钟
    //调用 GPIO_Init()函数, 初始化 PC6 引脚, 模式为 Out_PP
    GPIOinit.GPIO_Pin = GPIO_Pin_6;
    GPIOinit.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIOinit.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_Init(GPIOC, &GPIOinit);
    while(1)
    {  GPIO_WriteBit(GPIOC, GPIO_Pin_6, Bit_SET);           //输出高电平, 灯亮
       delay(0x200000);
       GPIO_WriteBit(GPIOC, GPIO_Pin_6, Bit_RESET);         //输出低电平, 灯灭
       delay(0x200000);
    }
}
```

复位时触发复位中断，CPU 响应复位中断，自动执行 Reset_Handler 中断处理子程序，完成系统时钟初始化后，进入用户编写的 main()函数。

main()函数中首先应该完成各个硬件模块的初始化工作，然后进入 while(1)死循环，在循环体中编写代码实现具体功能。初始化硬件模块时必须先使能片上外设的时钟，然后才能初始化片上外设的工作模式。片上外设初始化完毕之后，才能调用硬件模块的接口函数，控制片上外设的工作。

调试程序时，可以在合适的指令旁设置断点。设置好断点后，按 F5 键连续运行程序，遇到断点暂停执行程序，这时就可以在 Watch 窗口中观察变量或片上外设寄存器的值。

BlinkyLED 项目只用到一个片上外设，即 GPIOC 模块，GPIO 模块都在 APB2 总线上，所以 main()函数中首先调用 RCC 模块的库函数 RCC_APB2PeriphClockCmd()使能 GPIOC 模块的时钟；然后再调用 GPIO 模块的库函数 GPIO_Init()初始化 GPIOC 模块 PC6 引脚的工作模式，完成引脚初始化。

在 Debug 调试环境下可以看到，调用 GPIO_Init()函数初始化 PC6 引脚后，GPIOC 模块的 CRL 寄存器从复位后默认值 0x44444444 改变成 0x42444444，说明 PC6 引脚的工作模式已经修改成功，如图 5.11 所示。如果没有先开启 GPIOC 模块时钟，那么由于模块不工作，GPIO_Init()函数调用不起作用，所以初始化后 CRL 寄存器依然为 0x44444444。

初始化 PC6 引脚的工作模式和速度后，在 while(1)循环体中，调用 GPIO 模块的库

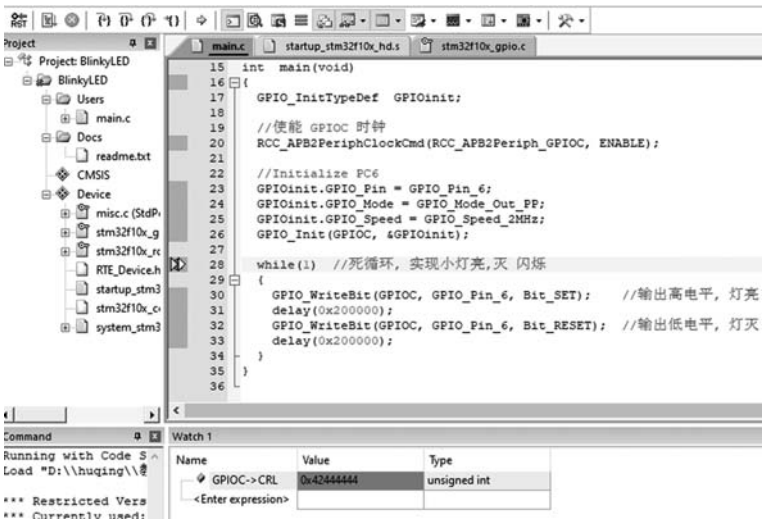


图 5.11 GPIOC 模块 CRL 寄存器情况

函数 GPIO_WriteBit()，先将 PC6 引脚置位，控制 LED 小灯亮，延时一段时间后，再次调用 GPIO_WriteBit()函数将 PC6 引脚复位，控制 LED 小灯灭。

由于单片机执行指令速度非常快，将 PC6 引脚置位或复位后，必须调用延时函数，使小灯亮、灭状态维持一段时间，才能看到小灯亮、灭闪烁的效果。如果不调用延时函数或者延时时间过短，那么单步调试程序时能够观察到小灯亮、灭状态的变换，但是连续运行时就会看到小灯一直点亮，而看不到小灯熄灭的状态了。

5.3 stm32f10x.h 头文件的作用

CMSIS 为片上硬件模块提供了相应的库函数，开发者只需要调用这些库函数，就能完成硬件模块的初始化，控制硬件模块的工作。在 CMSIS 基础上进行开发，能够大大降低开发难度，缩短项目开发的周期。嵌入式系统的开发人员有必要了解 CMSIS 固件库的基本架构，熟悉 CMSIS 库函数。

打开任意一个库函数的 C 语言源文件，会发现它们全都包含 stm32f10x.h 头文件。简单地说，想要在 CMSIS 固件库基础上完成嵌入式应用开发，就必须包含这个头文件。作为库函数的使用者，必须熟悉 stm32f10x.h 头文件，了解该文件的作用。

stm32f10x.h 头文件中主要包含以下信息。

(1) 定义枚举数据类型 IRQn，声明了所有中断源的中断类型号。

Cortex-M3 内核中的 NVIC 模块管理单片机中所有的中断源。每个中断源有一个唯一的编号，称为“中断类型号”。NVIC 模块最多可以管理 240 个中断源，包括内核产生的异常(Exception)和片上外设产生的中断(Interrupt)。内核异常的类型号为负数，片上外设中断的类型号从 0 开始，内核异常的优先级高于片上外设中断。

基于 Cortex-M3 内核，生产商设计具体的单片机型号时，根据单片机集成的片上硬件资源，规定了片上外设中断源的个数以及对应的类型号。所以虽然 NVIC 最多可以管理 240 个中断源，但具体某个型号单片机的中断源个数可能远远低于 240。例如，STM32F10x 系列单片机最多只有 60 个片上外设中断源，中断类型号为 0～59。单片机参考手册的“中断和异常向量”节中给出了中断向量表。

stm32f10x.h 头文件中定义了 IRQn 枚举数据类型，包含了所有内核异常以及片上外设中断的类型号。由于不同类型单片机集成的片上资源不同，定义 IRQn 枚举数据类型时，使用了条件编译指令，根据项目中声明的单片机类型，编译不同的代码片段。例如，STM32F10X_HD 类型的大容量单片机片上资源丰富，IRQn 中定义的中断类型号包含了全部 60 个片上外设中断源，而 STM32F10X_LD 类型的小容量单片机片上资源较少，条件编译语句中定义的中断类型号就比较少。定义枚举数据类型 IRQn 的部分代码如图 5.12 所示。

```
typedef enum IRQn
{
/******  Cortex-M3 Processor Exceptions Numbers ***********************************/
  NonMaskableInt_IRQn        = -14,    /*!< 2 Non Maskable Interrupt                */
  MemoryManagement_IRQn      = -12,    /*!< 4 Cortex-M3 Memory Management Interrupt */
  BusFault_IRQn              = -11,    /*!< 5 Cortex-M3 Bus Fault Interrupt         */
  UsageFault_IRQn            = -10,    /*!< 6 Cortex-M3 Usage Fault Interrupt       */
  SVCall_IRQn                = -5,     /*!< 11 Cortex-M3 SV Call Interrupt          */
  DebugMonitor_IRQn          = -4,     /*!< 12 Cortex-M3 Debug Monitor Interrupt    */
  PendSV_IRQn                = -2,     /*!< 14 Cortex-M3 Pend SV Interrupt          */
  SysTick_IRQn               = -1,     /*!< 15 Cortex-M3 System Tick Interrupt      */

/******  STM32 specific Interrupt Numbers *****************************************/
  WWDG_IRQn                  = 0,      /*!< Window WatchDog Interrupt               */
  PVD_IRQn                   = 1,      /*!< PVD through EXTI Line detection Interrupt */
  TAMPER_IRQn                = 2,      /*!< Tamper Interrupt                        */
  RTC_IRQn                   = 3,      /*!< RTC global Interrupt                    */
  FLASH_IRQn                 = 4,      /*!< FLASH global Interrupt                  */
  RCC_IRQn                   = 5,      /*!< RCC global Interrupt                    */
  EXTI0_IRQn                 = 6,      /*!< EXTI Line0 Interrupt                    */
  EXTI1_IRQn                 = 7,      /*!< EXTI Line1 Interrupt                    */
  EXTI2_IRQn                 = 8,      /*!< EXTI Line2 Interrupt                    */
  EXTI3_IRQn                 = 9,      /*!< EXTI Line3 Interrupt                    */
  EXTI4_IRQn                 = 10,     /*!< EXTI Line4 Interrupt                    */
```

图 5.12　stm32f10x.h 头文件中的 IRQn 枚举数据类型定义

(2) 包含系统头文件。

打开 CMSIS 的库函数文件，常常会看到一些 ANSI C 标准中没有定义的数据类型，例如 uint8_t 或 uint32_t 等。这些数据类型是重命名的 ANSI C 基本数据类型，例如，uint8_t 实际上就是 unsigned char 类型，而 uint32_t 实际就是 unsigned int 类型。

系统头文件 stdint.h 中用 typedef 关键字对无符号和有符号整型数据类型做了一系列的重命名，相关定义代码片段见图 5.13。要使用这些重命名后的数据类型，就必须包含相应的头文件。

```
    /* exact-width signed integer types */
typedef   signed          char int8_t;
typedef   signed short     int int16_t;
typedef   signed           int int32_t;
typedef   signed       __INT64 int64_t;

    /* exact-width unsigned integer types */
typedef unsigned          char uint8_t;
typedef unsigned short     int uint16_t;
typedef unsigned           int uint32_t;
typedef unsigned       __INT64 uint64_t;
```

图 5.13　stdint.h 中重定义整型的部分代码

在嵌入式系统开发中，常常需要定义整型变量来访问片上外设的寄存器，这些寄存器可能是16位的，可能是32位的，所定义的整型变量位宽必须与要访问的寄存器一致。ANSI C标准中的char、short、int等数据类型，从名称上看不出所定义整型变量的位宽，而重命名后的数据类型，名称中就明确说明了整型的位数，以及有符号或无符号的特性，例如，int8_t为有符号8位整型，而uint16_t是无符号16位整型。

除了stdint.h头文件以外，作为嵌入式系统的开发人员，有时需要访问内核的相关寄存器，这时就需要包含core_cm3.h头文件。core_cm3.h是为内核定义的头文件，其中包含内核中硬件模块的结构体数据类型定义，如NVIC模块和SCB模块等。要访问内核硬件模块，就需要包含core_cm3.h头文件。

如果开发人员要自己了解所有系统头文件的作用，并在代码中一一包含这些头文件，会是一件非常头疼的事情。stm32f10x.h头文件中包含了stdint.h、core_cm3.h以及其他的系统头文件，开发人员只需要包含stm32f10x.h这一个头文件，就可以顺利在代码中调用固件库接口函数了。

(3) 包含标准外设驱动的头文件。

在CMSIS固件库基础上完成项目开发时，新建项目后，需要在Options for Target 'BlinkyLED'窗口的C/C++标签页中预定义两个符号，STM32F10X_HD和USE_STDPERIPH_DRIVER，如图5.14所示。符号STM32F10X_HD说明单片机的类型，而符号USE_STDPERIPH_DRIVER说明项目开发使用标准外设驱动。这两个符号的定义会改变stm32f10x.h头文件中条件编译指令的编译结果。

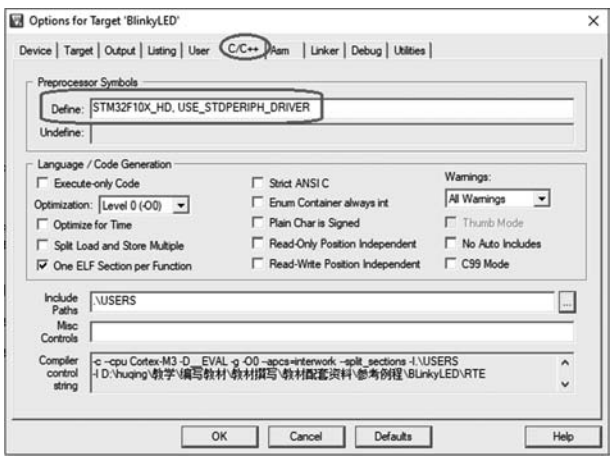


图5.14 C/C++标签页

如果定义了符号USE_STDPERIPH_DRIVER，那么stm32f10x.h头文件中的条件编译指令满足条件，编译时会包含stm32f10x_conf.h头文件。stm32f10x_conf.h头文件被称作“头文件的头文件”，其中包含了项目选中的CMSIS组件的相关头文件，这样开发人员就可以直接调用库函数控制片上外设了，如图5.15所示。

在Manage Run-Time Environment窗口中选中项目所需的所有组件后，Keil 5开发

软件自动生成 RTE_Components.h 头文件，根据配置情况自动生成对应的宏定义指令。stm32f10x_conf.h 头文件中用条件编译语句包含 CMSIS 每个组件的头文件，如果定义了对应的宏，则包含了该组件的头文件。所以只要在 Manage Run-Time Environment 窗口中选中了对应的组件，那么编译后 stm32f10x_conf.h 头文件就会包含该组件的头文件。而 stm32f10x.h 头文件中包含了 stm32f10x_conf.h 头文件，因此，对于开发人员来说，只要包含 stm32f10x.h 头文件，就能调用 CMSIS 固件库的函数了。

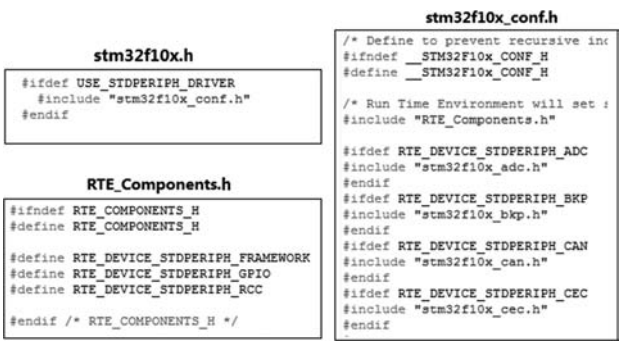


图 5.15　相关头文件

(4) 为访问片上外设寄存器定义结构体数据类型和指针。

单片机内部集成有多种硬件模块，如模数转换模块 ADC、通用目的输入/输出模块 GPIO、定时器模块 TIM。某些硬件模块可能有多个，例如，STM32F103VET6 单片机中集成了 GPIOA～GPIOE 共 5 个 GPIO 模块。每个硬件模块内有多个寄存器，开发人员需要读写这些寄存器，才能了解硬件模块的工作状态，控制硬件模块的工作模式。

单片机集成的片上硬件模块个数以及所有片上外设的地址都是固定的，参考手册“存储器映像”中列出了所有片上外设寄存器组的起始地址。并且参考手册中为每一种硬件模块单独编写了一章，详细介绍硬件模块的功能，模块中每个寄存器的作用，并在最后一节中列表给出模块中寄存器的相对偏移地址。

所有的片上外设都挂接在 AHB、APB1 或 APB2 总线上，总线的基地址以及各个片上外设的起始地址都是确定的。相应地，stm32f10x.h 头文件中定义了一系列的宏，首先定义了各个存储区的基地址，接着定义了 3 总线的基地址，最后定义了总线上片上外设的基地址，如图 5.16 所示。

```
#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 */
#define PERIPH_BASE           ((uint32_t)0x40000000) /*!< Peripheral base address in the alias region */

#define SRAM_BB_BASE          ((uint32_t)0x22000000) /*!< SRAM base address in the bit-band region */
#define PERIPH_BB_BASE        ((uint32_t)0x42000000) /*!< Peripheral base address in the bit-band region */

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

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

#define TIM2_BASE             (APB1PERIPH_BASE + 0x0000)
#define TIM3_BASE             (APB1PERIPH_BASE + 0x0400)
#define TIM4_BASE             (APB1PERIPH_BASE + 0x0800)
```

图 5.16　片上外设基地址的相关宏定义

stm32f10x.h 头文件中为每种硬件模块定义了一个结构体数据类型，结构体中的每个数据成员对应模块中的一个寄存器，而结构体数据成员定义的先后顺序就反映了模块内寄存器的相对偏移地址。

定义了结构体数据类型，通过宏定义指定了片上外设基地址之后，stm32f10x.h 头文件为每个片上外设定义了一个宏，通过强制类型转换，将片上外设的基地址转换成相应的结构体指针。通过结构体指针就能够访问片上外设的寄存器了。

例如 GPIO 模块，参考手册“GPIO 和 AFIO 寄存器地址映像”节给出了 GPIO 模块寄存器的相对地址。而单片机芯片中包含多个 GPIO 模块，STM32F103 系列单片机中最多包含 GPIOA～GPIOG 共 7 个 GPIO 模块，参考手册“存储器映像”节中给出了所有片上外设的地址空间。

stm32f10x.h 头文件中首先为 GPIO 模块定义了结构体数据类型 GPIO_TypeDef，然后定义宏，将每个 GPIO 模块地址强制转换成 GPIO_TypeDef 类型的指针，如图 5.17 所示。

偏移	寄存器	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7～6	5	4	3	2	1	0
000h	GPIOx_CRL	CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[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]	
	复位值	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
004h	GPIOx_CRH	CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[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]	
	复位值	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
008h	GPIOx_IDR	保留																IDR[15:0]															
	复位值																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00Ch	GPIOx_ODR	保留																ODR[15:0]															
	复位值																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
010h	GPIOx_BSRR	BR[15:0]																BSR[15:0]															
	复位值	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
014h	GPIOx_BRR	保留																BR[15:0]															
	复位值																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
018h	GPIOx_LCKR	保留															LCK	LCK[15:0]															
	复位值																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a) GPIO模块寄存器地址映像

0x4001 2000 - 0x4001 23FF	GPIO端口G	APB2
0x4001 2000 - 0x4001 23FF	GPIO端口F	
0x4001 1800 - 0x4001 1BFF	GPIO端口E	
0x4001 1400 - 0x4001 17FF	GPIO端口D	
0x4001 1000 - 0x4001 13FF	GPIO端口C	
0X4001 0C00 - 0x4001 0FFF	GPIO端口B	
0x4001 0800 - 0x4001 0BFF	GPIO端口A	

(b) 寄存器组起始地址

```
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;

#define GPIOA      ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB      ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC      ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD      ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE      ((GPIO_TypeDef *) GPIOE_BASE)
#define GPIOF      ((GPIO_TypeDef *) GPIOF_BASE)
#define GPIOG      ((GPIO_TypeDef *) GPIOG_BASE)
```

(c) stm32f10x.h中相关定义

图 5.17　如何访问片上的 GPIO 模块

stm32f10x.h 头文件中为所有的片上外设都定义了结构体数据类型，定义宏指明模块寄存器基地址，并且通过宏定义，将模块基地址强制转换成对应的结构体指针，因此只

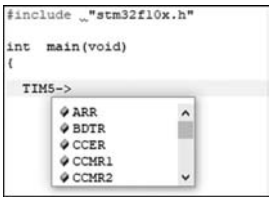


图 5.18　访问 TIM5 寄存器

需要包含 stm32f10x.h 头文件，就可以通过结构体指针访问所有片上外设寄存器了。

例 5.2：编程操作基本定时器 TIM5 的寄存器。

源程序中需要先包含 stm32f10x.h 头文件，编译后，编写代码时 Keil 5 开发软件会自动提示结构体成员，从中选择要访问的寄存器，对其进行读写操作就可以了，如图 5.18 所示。

5.4　项目中的文件管理

5.4.1　CMSIS 固件库文件

安装 Keil 5 开发软件后，需要根据所选用的单片机型号安装相应的 Pack 包。本书选用 STMicroelectronics 公司的 STM32F103 系列单片机，所以需要安装 STM32F1xx_DFP 的 Pack 包。安装了 Pack 包后，Keil 5 开发软件中就包含了 CMSIS 固件库，无须用户自己单独下载或复制 CMSIS 固件库文件到项目中。

建议安装 Keil 5 软件时不要修改安装路径，默认的安装路径为 c:\Keil_v5。软件安装后会新建一系列文件夹，其中 c:\Keil_v5\ARM\PACK 文件夹下包含所有安装的 Pack 包文件，Pack 文件夹下包含 ARM 和 Keil 子文件夹，其中 ARM 子文件夹下是 ARM 公司提供的与内核相关的 CMSIS 固件库，如图 5.19 所示，例如，core_cm3.h 头文件就存放在这个子文件夹下。

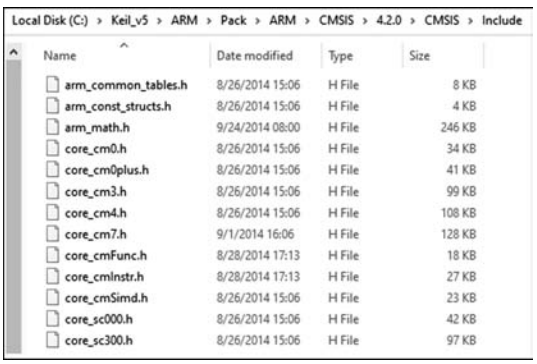
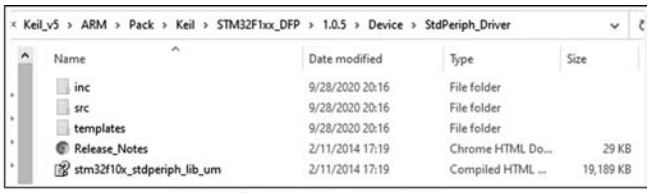
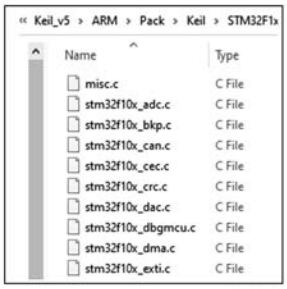


图 5.19　ARM 的 CMSIS 固件库所在路径

c:\Keil_v5\ARM\PACK\Keil 路径下存放各生产商提供的单片机系列的 Pack 包，c:\Keil_v5\ARM\PACK\Keil\STM32F1xx_DFP\1.0.5\Device\StdPeriph_Driver 文件夹下包含标准外设固件库文件，文件路径中的 1.0.5 是所安装 Pack 包的版本号。标准外设固件库的 C 语言源文件都存放在 src 子文件夹下，而对应的头文件都存放在 inc 子文件夹下，如图 5.20 所示。



(a) StdPeriph_Driver子文件夹



(b) StdPeriph_Driver\src子文件夹

(c) StdPeriph_Driver\inc子文件夹

图 5.20　Keil 5 中所安装的 Pack 包文件

5.4.2　项目中的系统文件

运行 Keil 5 开发软件，选择 Project 菜单下的 New uVision Project 菜单项，按照向导的指引新建项目后，Keil 5 软件会在指定的项目文件夹下新建一些子文件夹，并将部分固件库文件复制到项目文件夹下。新建项目后，Keil 5 软件在项目文件夹下新建了 RTE、Objects、Listings 子文件夹，以及扩展名为. uvoprojx 的项目文件和扩展名为. uvoptx 的项目选项配置文件。项目文件的文件名与项目同名，因此项目名称中不要有空格、中文或特殊符号，项目名称可参照 C 语言中标识符的命名规则。图 5.21 显示了 BLinkyLED 项目文件夹的内容，其中 USERS 和 myHardware 子文件夹是开发人员新建的，其他 3 个子文件夹是 Keil 5 新建的。

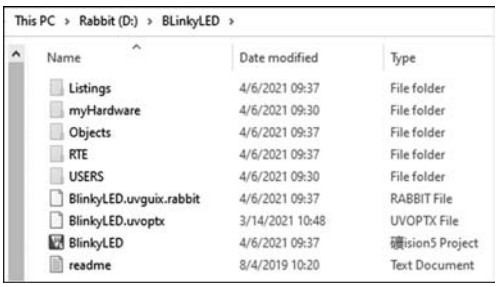


图 5.21　BLinkyLED 项目文件夹内容

(1) RTE 子文件夹

RTE 子文件夹下存放 Run-Time Environment 相关的文件。根据项目 CMSIS 组件

的配置情况，Keil 5 软件自动生成的 RTE_Components.h 头文件以及其他相关文件都在 RTE 子文件夹下，启动文件也复制放在这个文件夹下。只有少量 CMSIS 固件库文件被复制到项目文件夹下，项目使用的 CMSIS 标准外设驱动固件库文件没有复制到项目文件夹下，这些固件库文件在 Keil 5 软件的安装路径下。

（2）Objects 子文件夹

一个项目中包含有多个汇编语言或 C 语言源文件，单击 Build 工具按钮后，首先逐一编译项目中的每一个源文件，编译后输出同名的.d 和.o 文件，其中.o 为目标文件。编译成功后，再连接项目中所有的.o 目标文件，输出一个扩展名为.axf 的可执行文件，编译连接时的提示信息如图 5.22 所示。

图 5.22　编译项目时的提示信息

默认情况下，编译和连接时输出的文件都存放在 Objects 子文件夹，并且输出的可执行文件与项目同名。在项目配置的 Output 标签页中可以修改输出文件的存放位置，以及可执行文件名称，如图 5.23 所示。

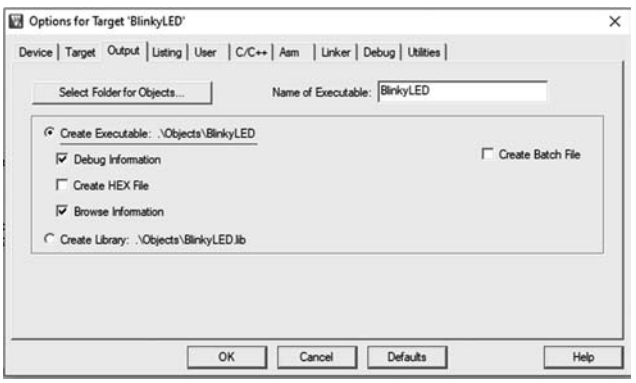
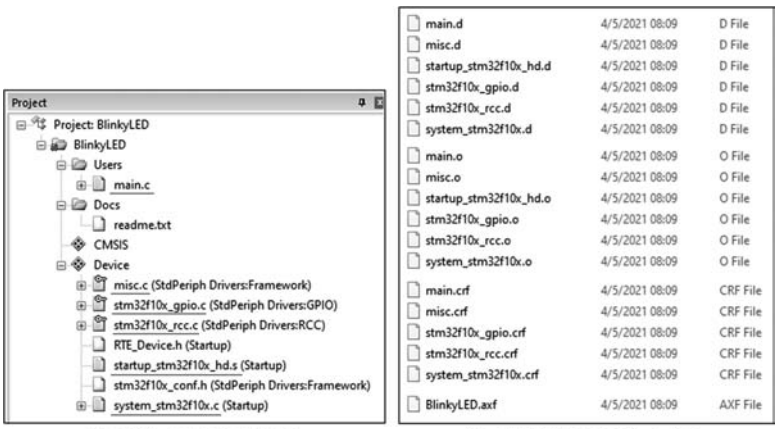


图 5.23　Output 标签页

图 5.23 中“Create Executable:.\Objects\BlinkyLED”中的“.\”说明是相对路径，指代当前目录，即项目文件夹，BlinkyLED 为文件名称。在右上方的输入框中可修改可执行文件的名字，单击上方的 Select Folder for Objects 按钮可重新选择输出文件的存放位置。如果在对话框中选中了 Create HEX File 复选框，那么输出 axf 文件后，转换工具会将 axf 文件转换成 HEX 格式的文件，此时输出文件夹下会产生两个文件名相同，扩展名分别为.axf 和.hex 的可执行文件。

除了目标文件和可执行文件外，编译源文件时还会产生扩展名为.crf的交叉引用文件，这个文件中包含了源代码中的宏定义、变量和函数的定义及声明。有了这个文件，编写程序时才能够通过Go to definitions of快捷菜单跳转到函数或变量的定义。

项目中包含多个源文件，每个源文件都需要经过编译，编译后产生对应的依赖文件.d、目标文件.o以及交叉引用文件.crf。所有源文件都编译成功后，才能连接产生可执行文件.axf。默认情况下，上述所有输出文件都存放在项目文件夹下的Objects子文件夹中，如图5.24所示。



(a) BlinkyLED项目窗口　　(b) 编译连接后的输出文件

图5.24　BlinkyLED项目的输出文件

(3) Listings子文件夹

Listings文件夹下存放由编译器和连接器输出的扩展名为.lst的列表文件以及扩展名为.map的镜像文件，这些对于研究编译过程非常有帮助。在项目配置的Listing标签页中可以修改列表文件的存放位置，以及.map文件包含的内容，如图5.25所示。

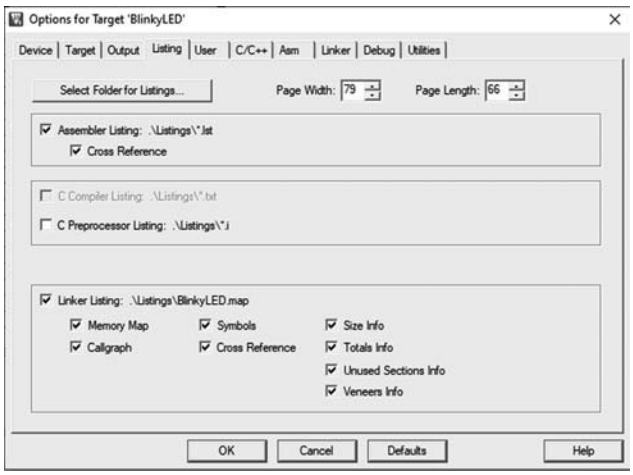


图5.25　Listing标签页

5.4.3 项目中的用户文件

新建项目后，开发人员需要向项目中添加 C 语言源文件，如主程序 main.c 源文件等，在这些源文件中编写代码，实现项目设计的功能。

(1) 用户文件的命名及保存

首先，从原则上说，项目中所有用户新建的源文件以及头文件必须存放在项目文件夹下。这样当需要备份项目，或需要将项目复制到其他 PC 上继续开发时，只需要复制项目文件夹就可以了。

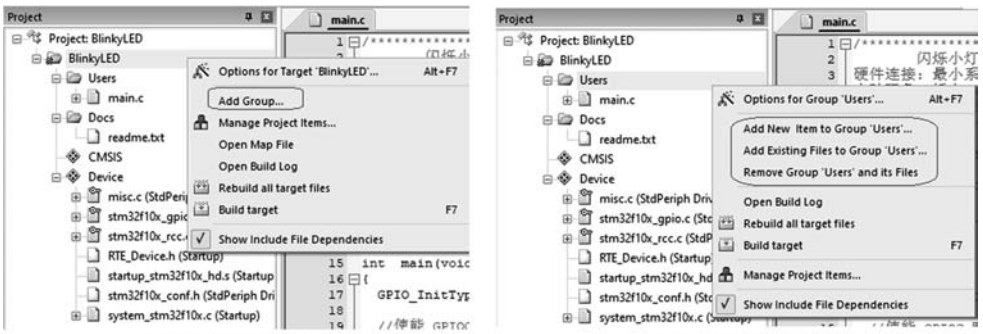
其次，源程序文件的名称应该在一定程度上说明程序的功能，例如，main.c 是主程序源文件，而串口操作的源文件可以命名为 myUart.c，实现复杂算法的源文件可以命名为 myMath.c。

除了 main.c 主程序源文件以外，其他源文件中大多会编程实现一些接口函数，为了方便主程序或其他文件调用这些接口函数，通常会为每个源文件新建一个对应的头文件，头文件的文件名应该与源程序文件名相同。例如，为控制按键和小灯而编写的 C 语言源文件可以命名为 KeyLED.c，而对应的头文件就应该命名为 KeyLED.h。在头文件中做函数声明、宏定义、全局变量的外部声明等，这样其他源程序只要包含头文件，就能够调用对应的接口函数了。

最后，如果项目功能较为复杂，为操作片上外设编写了相应的接口函数，新建了多个 C 语言源文件，就可以考虑在项目文件夹下新建子文件夹，分别存放这些源文件与相应的头文件。

(2) 项目中用户文件的组织

在 Keil 5 中打开项目后，在 Project 窗口中可以看到当前项目文件的组织情况，用户新建的源文件应该添加到项目中。在 Project 窗口的项目名称上右击，或在分组上右击，从弹出的快捷菜单中选择相应的菜单项，就能够为项目新建分组，或将源文件添加到相应分组中，如图 5.26 所示。



(a) 为项目增加分组　　(b) 向项目中添加文件

图 5.26　项目的文件管理

如果项目中有头文件存放在子文件夹下，那么需要在项目配置的C/C++标签页中将子文件夹路径添加到Include Paths中。

BlinkyLED项目在项目文件夹下新建了USERS子文件夹存放main.c源文件，新建了myHardware子文件夹存放GPIOBitBand.h头文件，main.c中包含了GPIOBitBand.h头文件，必须在C/C++标签页中添加myHardware子文件夹的路径，如图5.27所示。

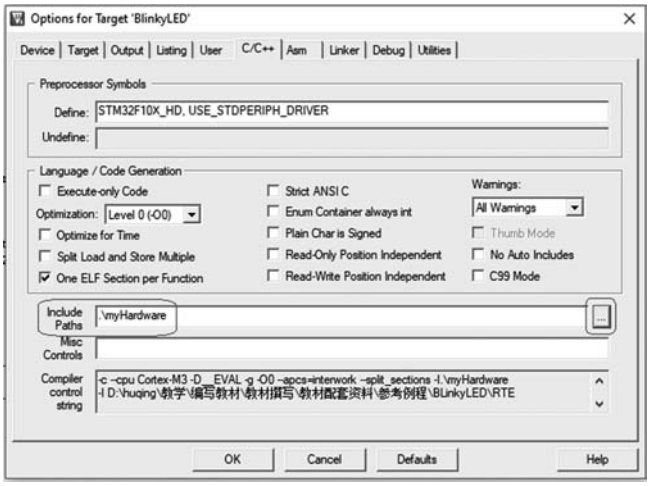


图5.27　C/C++标签页

如果项目文件夹下新建了多个子文件夹，或者一个子文件夹下又新建了子文件夹，那么必须在项目配置的C/C++标签页中逐一添加所有的子文件夹作为搜索路径。单击图5.27右方的“…”按钮，会弹出对话框，在已有搜索路径下方的空白处双击，就能够添加新的搜索路径了。

项目中的源文件需要包含某个头文件时，可以直接用指令“#include"头文件"”，指令中头文件名称用双引号括起来，双引号意味着编译器会先在项目的当前目录中查找，如果找不到，就会去系统配置的库环境变量以及用户配置的路径(即用户在C/C++标签页中添加的路径)中搜索。如果在所有这些路径中都没有找到头文件，编译器就报错，提示找不到头文件。例如，如果删除BlinkyLED项目中配置的包含路径，编译main.c源文件时编译器找不到所包含的GPIOBitBand.h头文件，此时就会产生编译错误，错误提示如图5.28所示。

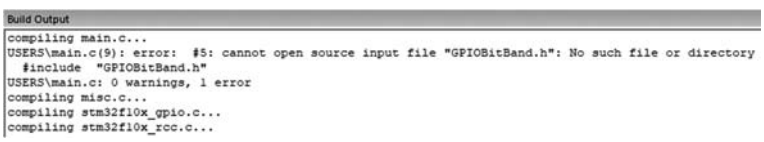


图5.28　编译错误：找不到头文件