第 5 章

CHAPTER 5

ARM 微控制器开发

学习嵌入式系统的根本目的在于应用微控制器进行应用系统设计。为了使得读者尽快进入应用开发的学习状态，本章介绍 Cortex-M3 微控制器应用系统的开发方法和过程。这些方法对于后续的知识点学习具有重要意义。

5.1 开发流程

ARM 微控制器可以使用的开发工具链有很多种，它们大多数都支持 C 和汇编语言。开发嵌入式工程时可以使用 C 语言，也可以用汇编，或者两者混合编程。一般情况下，程序代码生成流程如图 5-1 所示。

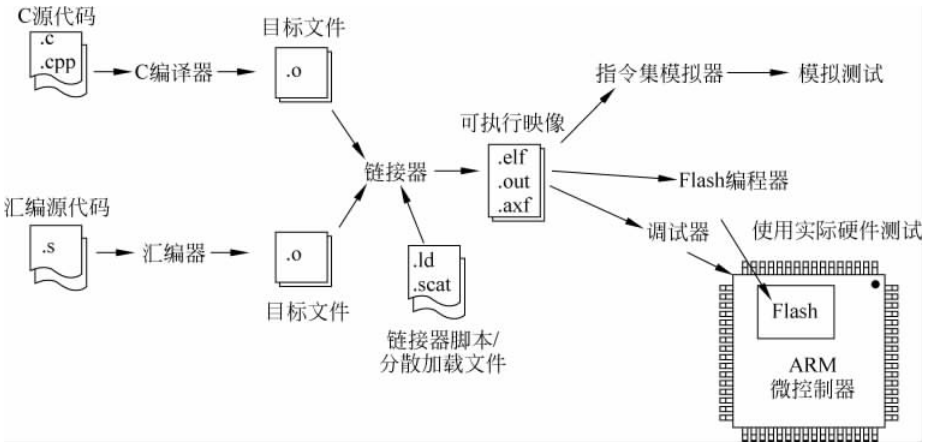

图 5-1 程序代码的生成流程

需要使用汇编实现的工程，则使用汇编器将汇编源代码转换成目标文件，工程中所有的目标代码被链接生成一个可执行映像。除了程序代码，目标文件和可执行映像中也可能含有各种调试信息。

大多数的简单应用程序可以完全用 C 语言编写。C 编译器将 C 程序代码编译成目标代码，然后由链接器生成程序映像文件。

生成可执行映像之后，可以将其下载到微控制器的 Flash 存储器或内存中进行测试。大多数开发工具都包含了一个接口友好的集成开发环境(IDE)，当其与在线调试器(有时也被称作在线仿真器 ICE)配合使用时，可以分步进行以下工作：创建工程，编译应用程序，然

后下载嵌入式应用程序到微控制器中。开发过程如图 5-2 所示。

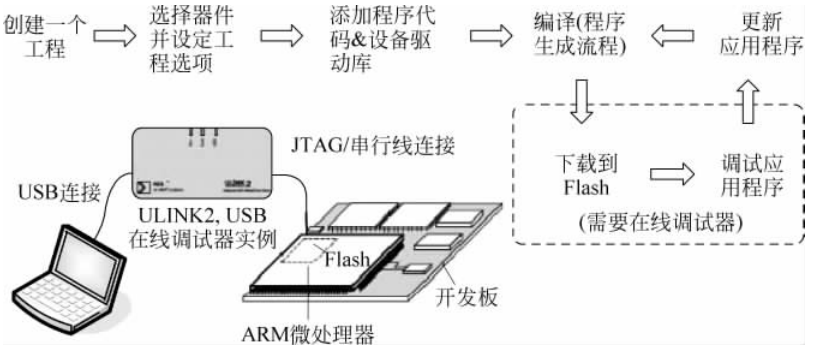

图 5-2　开发过程

5.2　处理器的启动过程

为了保存编译好的程序代码，大多数的现代微控制器都包含片上 Flash 存储器。程序代码以二进制机器码的形式存放在 Flash 存储器中，因此汇编语言程序代码必须经过汇编，而 C 程序代码必须经过编译，才能烧写到 Flash 中。有些微控制器可能还配备了一个独立的启动 ROM，里面有一个小的 Boot Loader 程序。微控制器启动以后，在执行 Flash 中的用户程序前，Boot Loader 会首先运行。大多数情况下，Boot Loader 都是固定的，只有 Flash 存储器中的用户程序才是可变的。

程序代码烧写到 Flash 存储器以后，处理器就可以访问程序了。复位以后，处理器将会运行复位流程。若使用 C 语言编程，则处理器复位以后的工作过程如图 5-3 所示。

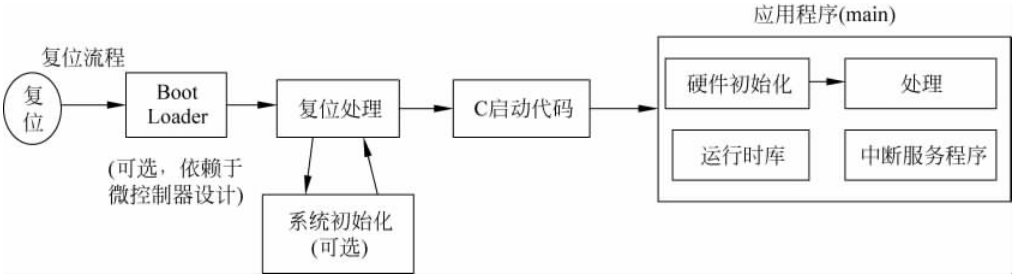

图 5-3　处理器复位以后的工作过程

在复位流程中，处理器会取出 MSP 的初始值和复位向量，然后开始执行复位处理，这些所需信息都放在一个叫作启动代码的程序文件中。启动代码中的复位处理可能还会履行系统初始化的职责(例如，时钟控制电路和锁相环 PLL 的初始化)，有些情况下，系统初始化是在 C 程序的 main()函数中开始的。例如，如果在开发中使用 Keil 微控制器开发套件(MDK)，工程创建向导可以将所选芯片对应的默认启动代码复制到工程中。

对于用 C 语言开发的应用程序，在进入主流程以前，启动代码就已经开始执行，并对应用程序用到的变量和内存等进行了初始化。无须编程者考虑启动代码，因为 C 语言开发工具会将其自动插入程序映像中。而对于使用汇编语言开发的应用程序，许多工作需要开发

者自行完成，例如设置堆栈、设置程序的入口等。第 4 章提供了完整的汇编语言程序框架。

执行完 C 启动代码以后，应用程序就开始执行了。应用程序通常包含以下几个部分：

- 硬件初始化（如时钟、PLL 和外设）；
- 应用程序的处理部分；
- 中断服务程序。

另外，应用程序可能还会用 C 库函数，此时，C 编译器/链接器会将所需的库函数纳入编译好的程序映像中。

硬件初始化可能会涉及一系列的外设、系统控制寄存器及 Cortex-M3 中的中断控制寄存器。如果在复位处理时没有进行处理，系统时钟控制和 PLL 也需要进行初始化。外设初始化完成后，程序就可以继续执行应用程序处理部分了。

5.3 输入和输出接口

在许多嵌入式系统中可用的输入和输出可能有数字和模拟输入/输出（I/O）、UART、I2C 和 SPI 等。许多微控制器还提供 USB、以太网、CAN、LCD 以及 SD 卡等接口，这些接口是由微控制器的外设控制的。

Cortex-M3 的寄存器映射到了系统空间，并且它们还控制着外设。有些外设要比 8 位机和 16 位机上的更加复杂，配置时可能会涉及更多的寄存器。

外设的典型初始化过程一般包括以下步骤：

（1）配置时钟控制回路，使能外设的时钟信号，如果有必要，那么初始化相应的引脚。

在许多低功耗微控制器中，时钟信号被分为了许多路，而且为了降低功耗，它们可以单独开关。大多数时钟信号默认都是关闭的，配置外设前通常需要使能相应的时钟。有些情况下，用户可能还需要使能外设总线系统的时钟。

（2）配置 I/O，大多数微控制器的引脚都是复用的，需要对 I/O 引脚的功能进行配置，以确保外设接口正常工作。另外，有些微控制器的 I/O 引脚的电气特性也是可以配置的，这样也就增加了配置步骤。

（3）配置外设，大多数接口外设都有多个可编程的控制寄存器，因此，为了确保外设工作正常，就需要对寄存器进行一系列的编程操作。

（4）配置中断，如果外设操作需要中断处理，就需要另外配置中断控制器（例如 Cortex-M3 的 NVIC）。

为了方便软件开发，大多数微控制器供应商都会为外设编程提供设备驱动库。

5.4 程序映像

Cortex-M3 的程序映像一般包含以下几个部分：

- 向量表；
- C 启动代码；
- 程序代码（应用程序代码和数据）；
- C 库代码（C 库函数的程序代码，链接时插入）。

1. 向量表

向量表可以用C语言或汇编语言实现。由于向量表的入口需要编译器和链接器生成的内容，所以向量表代码的实现细节同开发工具链相关。例如，栈指针的初始值被链接到链接器生成的栈空间地址，而复位向量则指向了C启动代码的地址，这些都是同编译器相关的。

Keil MDK创建STM32工程时，则将向量表作为汇编启动代码的一部分，并且使用定义常量数据(DCD)指令创建。ST公司提供的启动文件startup_stm32f10x_hd.s使用汇编实现向量表，文件的部分内容如下(在第6章创建C语言工程文件时自动加入工程中，可以直接打开查看完整的内容)：

```
Stack_Size      EQU     0x00000400
                AREA    STACK, NOINIT, READWRITE, ALIGN = 3
Stack_Mem       SPACE   Stack_Size                  ;分配栈空间
__initial_sp
Heap_Size       EQU     0x00000200
                AREA    HEAP, NOINIT, READWRITE, ALIGN = 3
__heap_base
Heap_Mem        SPACE   Heap_Size                   ;分配堆空间
__heap_limit
                PRESERVE8                           ; 表明该文件中的代码预留 8 字节对齐的栈
                THUMB
; 向量表,复位时映射到地址 0
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size
__Vectors       DCD     __initial_sp                ;栈顶
                DCD     Reset_Handler               ;复位处理程序入口地址
                DCD     NMI_Handler                 ;NMI 处理程序入口地址
                DCD     HardFault_Handler           ;硬件错误处理程序入口地址
                DCD     MemManage_Handler           ;MPU Fault Handler
                DCD     BusFault_Handler            ;Bus Fault Handler
                DCD     UsageFault_Handler          ;Usage Fault Handler
                DCD     0                           ;Reserved(保留,占位用)
                DCD     0                           ;Reserved(保留,占位用)
                DCD     0                           ;Reserved(保留,占位用)
                DCD     0                           ;Reserved(保留,占位用)
                DCD     SVC_Handler                 ;SVCall Handler(SVCall 处理)
                DCD     DebugMon_Handler            ;Debug Monitor Handler
                DCD     0                           ;Reserved(保留,占位用)
                DCD     PendSV_Handler              ;PendSV Handler(PendSV 处理)
                DCD     SysTick_Handler             ;SysTick Handler(SysTick 处理)
                ;外部中断
                DCD     WWDG_IRQHandler             ;Window Watchdog
                DCD     PVD_IRQHandler              ;PVD through EXTI Line detect
                DCD     TAMPER_IRQHandler           ;Tamper
                DCD     RTC_IRQHandler              ;RTC
                DCD     FLASH_IRQHandler            ;Flash
```

```
                DCD     RCC_IRQHandler              ;RCC
                DCD     EXTI0_IRQHandler            ;EXTI Line 0
                ......
__Vectors_End

__Vectors_Size EQU __Vectors_End - __Vectors

                AREA    |.text|, CODE, READONLY
; Reset handler
Reset_Handler  PROC
                EXPORT  Reset_Handler               [WEAK]
                IMPORT  __main
                IMPORT  SystemInit
                LDR     R0, = SystemInit
                BLX     R0
                LDR     R0, = __main
                BX      R0
                ENDP

; Dummy Exception Handlers (infinite loops which can be modified)

NMI_Handler     PROC
                EXPORT  NMI_Handler                 [WEAK]
                B       .
                ENDP
HardFault_Handler\
                PROC
                EXPORT  HardFault_Handler           [WEAK]
                B       .
                ENDP
MemManage_Handler\
                PROC
                EXPORT  MemManage_Handler           [WEAK]
                B       .
                ENDP
BusFault_Handler\
                PROC
                EXPORT  BusFault_Handler            [WEAK]
                B       .
                ENDP
UsageFault_Handler\
                PROC
                EXPORT  UsageFault_Handler          [WEAK]
                B       .
                ENDP
SVC_Handler     PROC
                EXPORT  SVC_Handler                 [WEAK]
                B       .
                ENDP
DebugMon_Handler\
                PROC
```

```
                EXPORT   DebugMon_Handler          [WEAK]
                B        .
                ENDP
PendSV_Handler  PROC
                EXPORT   PendSV_Handler            [WEAK]
                B        .
                ENDP
SysTick_Handler PROC
                EXPORT   SysTick_Handler           [WEAK]
                B        .
                ENDP
Default_Handler PROC
                EXPORT   WWDG_IRQHandler           [WEAK]
                EXPORT   PVD_IRQHandler            [WEAK]
                EXPORT   TAMPER_IRQHandler         [WEAK]
                EXPORT   RTC_IRQHandler            [WEAK]
                EXPORT   FLASH_IRQHandler          [WEAK]
                EXPORT   RCC_IRQHandler            [WEAK]
                EXPORT   EXTI0_IRQHandler          [WEAK]
                ......
WWDG_IRQHandler
PVD_IRQHandler
TAMPER_IRQHandler
RTC_IRQHandler
FLASH_IRQHandler
RCC_IRQHandler
EXTI0_IRQHandler
...
                B        .
                ENDP
                ALIGN
;*******************************************************************************
; User Stack and Heap initialization
;*******************************************************************************
                IF       :DEF:__MICROLIB
                EXPORT   __initial_sp
                EXPORT   __heap_base
                EXPORT   __heap_limit
                ELSE
                IMPORT   __use_two_region_memory
                EXPORT   __user_initial_stackheap
__user_initial_stackheap

                LDR      R0, = Heap_Mem
                LDR      R1, = (Stack_Mem + Stack_Size)
                LDR      R2, = (Heap_Mem + Heap_Size)
                LDR      R3, = Stack_Mem
                BX       LR
                ALIGN
                ENDIF
                END
```

在上面的例子中，向量表被赋予了一个段名 RESET。为了将向量表置于系统存储器映射的开头（地址为 0x00000000），链接文件或命令行选项需要知道段的名字，以便链接器能够正确识别向量并对其进行地址映射。复位向量一般指向 C 启动代码的开头。

2. C 启动代码

C 启动代码用于设置像全局变量之类的数据，也会清零加载时未被初始化的内存区域。对于使用 malloc 等 C 函数的应用程序，C 启动代码还需要初始化堆空间的控制变量。初始化完成后，启动代码跳转到 main( )程序执行。

C 启动代码由编译器/链接器自动嵌入到程序中，并且是和开发工具链相关的，而只使用汇编代码编程则可能不存在 C 启动代码。

3. 程序代码

用户指定的任务是由应用程序的指令完成的，除了指令以外，还有以下各类数据：

- 变量的初始值。函数或子程序中的局部变量需要初始化，这些初始值会在程序执行期间被赋给相应的变量。
- 程序代码中的常量。应用程序中的常量数据有多种用法，如数据值、外设寄存器的地址和常量字符串等，这些数据在程序映像中一般作为数据块放在一起。
- 应用程序可能也会包括其他的常量，比如查找表和图像数据，它们也被合并在程序映像中。

4. C 库代码

当使用特定的 C/C++库函数时，它们的库代码就会由链接器嵌入到程序映像中。另外，有些数据处理任务需要浮点数或除法运算，在进行这些运算时，C 库代码也会被包含进来。

5.5 C 语言开发 ARM 应用

1. 数据类型

C 语言支持多个“标准”数据类型，不过，数据类型的使用可能还与处理器的体系结构和 C 编译器相关。对于包括 Cortex-M3 在内的 ARM 处理器，所有的 C 编译器都支持如表 5-1 所示的数据类型。

表 5-1 Cortex-M3 处理器支持的数据类型及长度

C 和 C99 数据类型	位数	范围（有符号）	范围（无符号）
char，int8_t，uint8_t	8	−128～127	0～255
short，int16_t，uint16_t	16	−32 768～32 767	0～65 535
int，int32_t，uint32_t	32	−2 147 483 648～2 147 483 647	0～4 294 967 265
long	32	−2 147 483 648～2 147 483 647	0～4 294 967 265
long long，int64_t，uint64_t	64	$-2^{63}\sim 2^{63}-1$	$0\sim 2^{64}-1$
float	32	$-3.402\,823\,4\times 10^{38}\sim 3.402\,823\,4\times 10^{38}$	
double	64	$-1.797\,693\,134\,862\,315\,7\times 10^{308}\sim 1.797\,693\,134\,862\,315\,7\times 10^{308}$	

续表

C 和 C99 数据类型	位数	范围(有符号)	范围(无符号)
long double	64	$-1.797\ 693\ 134\ 862\ 315\ 7\times10^{308}$～$1.797\ 693\ 134\ 862\ 315\ 7\times10^{308}$	
pointers	32	0x0～0xFFFFFFFF	
enum	8/16/32	可能的最小数据类型	
bool(C++), _Bool(C)	8	真或假	
wchar_t	16	0～65 535	

2. 用 C 语言操作外设

除了变量以外,微控制器的 C 应用程序通常需要操作外设。对于 ARM Cortex-M3 微控制器,外设寄存器被映射到系统存储器空间,可以通过指针访问它们。使用微控制器供应商提供的设备驱动可以简化开发任务,并且增强软件在不同平台间的可移植性。如果需要直接访问外设寄存器可以使用以下方法。

如果只是简单访问几个寄存器可以使用下面的方法将外设寄存器定义为指针:

```
#define PERIPH_BASE       ((uint32_t)0x40000000)  //外设基地址
#define APB1PERIPH_BASE    PERIPH_BASE
#define USART1_BASE       (APB2PERIPH_BASE + 0x3800)
#define USART1             ((USART_TypeDef * ) USART1_BASE)
#define USART2_BASE       (APB1PERIPH_BASE + 0x4400)
#define USART2             ((USART_TypeDef * ) USART2_BASE)
```

结构体 USART_TypeDef 的定义如下:

```
typedef struct
{
    __IO uint16_t SR;
    uint16_t RESERVED0;
    __IO uint16_t DR;
    uint16_t RESERVED1;
    __IO uint16_t BRR;
    uint16_t RESERVED2;
    __IO uint16_t CR1;
    uint16_t RESERVED3;
    __IO uint16_t CR2;
    uint16_t RESERVED4;
    __IO uint16_t CR3;
    uint16_t RESERVED5;
    __IO uint16_t GTPR;
    uint16_t RESERVED6;
} USART_TypeDef;
```

使用 USART1 发送一个字节的数据时,可以使用下面的语句:

```
USART1 -> DR = mydata;
```

用同样的方法可以操作 USART2。

从上述方法可以看出,同一个外设寄存器的结构体可以被多个外设实体共用,这样也使

得程序维护变得容易。另外，由于立即数存储的减少，编译出的程序代码也会变小。

若进一步处理可以将外设操作封装成函数的形式，多个外设实体将其基指针作为参数进行函数调用。例如，利用串口发送一个字节数据，可以封装成如下的函数：

```
void USART_SendData(USART_TypeDef * USARTx, uint16_t Data)
{
    /* Check the parameters */
    assert_param(IS_USART_ALL_PERIPH(USARTx));
    assert_param(IS_USART_DATA(Data));

    /* Transmit Data */
    USARTx->DR = (Data & (uint16_t)0x01FF);
}
```

后面要介绍的 STM32 固件库函数就是基于这种思路设计的。

应该注意的是，在定义 USART_TypeDef 结构时，使用了__IO 变量类型，该类型实质上是 volatile 的宏定义（该宏定义包含在 core_m3.h 文件中）。外设访问定义指针时，需要使用 volatile 关键字。volatile 用于防止相关变量被优化。例如对外部寄存器的读写。对有些外部设备的寄存器来说，读写操作可能都会引发一定硬件操作，但是如果不加 volatile，编译器会把这些寄存器作为普通变量处理，例如连续多次对同一地址写入，会被优化为只有最后一次写入。另一个使用场合是中断。如果一个全局变量，在中断函数和普通函数中都用到过，那么最好对这个变量加 volatile 修饰。否则在普通函数中，可能会仅从寄存器里读取这个变量以便加快速度，而不去实际地址读取该变量。

3. Cortex 微控制器软件接口标准（CMSIS）

1）CMSIS 简介

随着嵌入式系统软件复杂性的增加，软件代码的兼容性和可重用性变得更加重要。可重用的程序能够减少后续项目的开发时间，也就加快了产品推向市场的速度；而软件的兼容性则有助于第三方软件组件的使用。

为了使软件产品具有高度的兼容性和可移植性，ARM 公司与许多微控制器供应商和软件方案供应商共同努力，开发了一个通用的软件框架 CMSIS，该框架适用于大多数的 Cortex-M 处理器以及 Cortex-M 微控制器产品。CMSIS 针对处理器特性提供了标准化的操作函数。

CMSIS 一般是作为微控制器厂商提供的设备驱动库的一部分来使用的。为了使用诸如 NVIC 和系统控制功能等处理器特性，CMSIS 提供了一种标准化的软件接口。CMSIS 对多种微控制器厂商都是统一的，并已被多种 C 编译器和开发工具（包括 Keil MDK）所支持。

2）CMSIS 的标准化

CMSIS 为嵌入式软件提供了以下标准化的内容：

- 标准化的操作函数，用于访问 NVIC、系统控制块（SCB），SysTick 的中断控制和初始化。
- NVIC、SCB 和 SysTick 寄存器的标准化定义，为了达到最佳的可移植性，应该使用这些标准化的操作函数；不过，有些情况下，需要直接操作 NVIC、SCB 和 SysTick

的寄存器。这时，标准化的寄存器定义就能提高软件的可移植性。

- 使用 Cortex-M 微控制器特殊指令的标准化函数。有些指令不能由普通的 C 代码生成，如果需要这些指令，则可以使用 CMSIS 提供的这类函数来实现；否则，用户就不得不使用 C 编译器提供的内在函数或者嵌入汇编代码，这样做就会依赖于开发工具，并且降低代码的可移植性。
- 系统异常处理的标准化命名。嵌入式操作系统往往需要系统异常，当系统异常都有了标准化的命名以后，在一个操作系统中支持不同的设备驱动库也就容易了。
- 系统初始化函数的标准化命名。通用的系统初始化函数被命名为 void SystemInit (void)，这就使得软件开发人员在建立自己工程时花费的力气更小。
- 为时钟频率信息建立标准化的变量。这个变量为 SystemCoreClock(CMSIS v1. 30 及之后)，用于确定处理器的时钟频率。

CMSIS 还提供以下内容：

- 设备驱动库的通用平台，这使得每个设备驱动库看起来都是一样的，也让初学者容易学习，并且方便软件移植。
- 在将来的 CMSIS 的发行版中可能会纳入一套通用的通信访问函数，以便已经开发的中间件(middleware)无须移植就能在不同设备上重复使用。

CMSIS 是为了满足基本操作的兼容性而开发的，微控制器供应商为了加强其软件解决方案可以增加函数接口，以免 CMSIS 限制嵌入式产品的功能和性能。

3) CMSIS 的组织结构

CMSIS 的组织结构如图 5-4 所示。

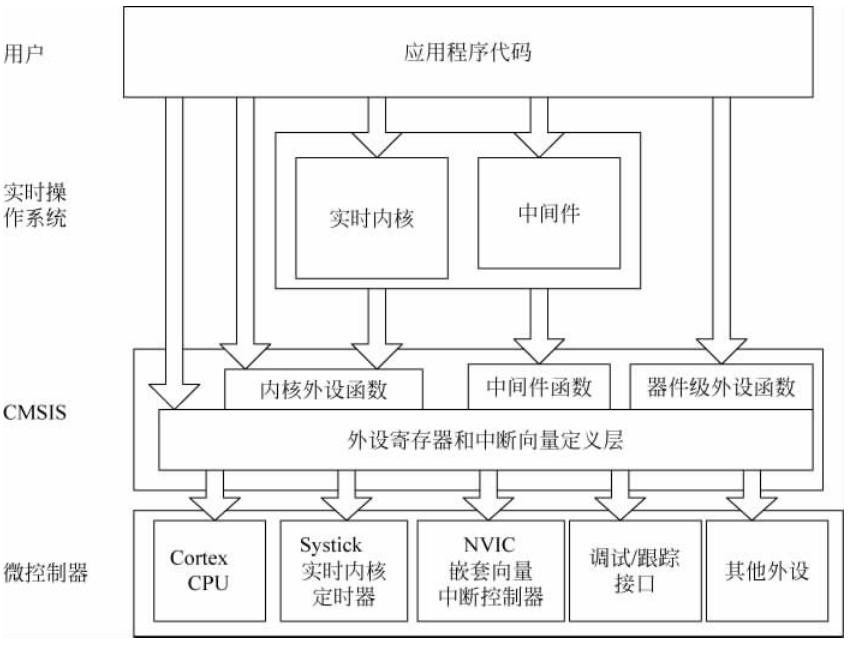

图 5-4　CMSIS 的组织结构

CMSIS 可以分为以下 4 层：

(1) 核心外设访问层。

命名定义、地址定义以及访问核心寄存器和 NVIC、SCB 以及 SysTick 等核心外设的辅助功能。

(2) 中间件访问层。

- 典型嵌入式系统访问外设的通用方法；
- 面向通信接口，包括 UART、Ethernet 和 SPI 等；
- 嵌入式软件能够在任何支持特定通信接口的 Cortex 微控制器上使用。

(3) 设备外设访问层(MCU 相关)。

寄存器名称定义、地址定义以及访问外设的设备驱动代码。

(4) 外设的访问函数(MCU 相关)。

可选的外设辅助函数。

4) CMSIS 的使用

CMSIS 被集成在微控制器供应商提供的设备驱动包中，如果使用设备驱动库进行软件开发，那么就已经在使用 CMSIS 了。

对于 C 程序代码，通常只需要包含微控制器供应商提供的设备驱动库中的头文件。这个头文件又包含了其他所有需要的头文件，包括 CMSIS 特性和外设驱动等。

也可以包含符合 CMSIS 的启动代码，它们可以是 C 代码也可以是汇编代码，CMSIS 为不同开发工具链定制了各种版本的启动代码。

一个使用 CMSIS 包建立的简单工程如图 5-5 所示。

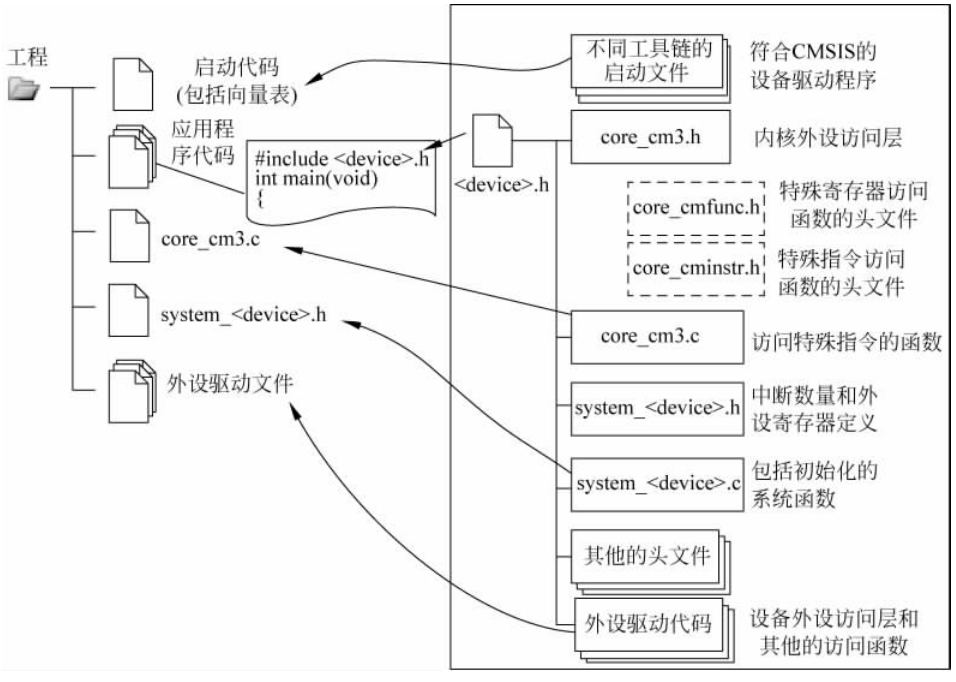

图 5-5　在工程中使用 CMSIS

其中，<device>的名字由实际的微控制器设备决定(例如 STM32F10x 系列微控制器对应的文件名为 system_stm32f10x.c)。当使用设备驱动库提供的头文件时，会自动包含其他所需的头文件(例如 STM32F10x 系列微控制器对应的文件名为 system_ stm32f10x.h)。

一个使用 CMSIS 的简单例子如下：

```
#include "stm32f10x.h"
void delay(u32 delaytime);
int main(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    while(1)
    {
        GPIO_SetBits(GPIOD, GPIO_Pin_8);
        delay(3000000);
        GPIO_ResetBits(GPIOD, GPIO_Pin_8);
        delay(3000000);
    }
}
void delay(u32 delaytime)
{
    while(delaytime--);
}
```

5.6 固件库

意法半导体(STMicroelectronics，以下简称 ST)为了方便用户开发程序，提供了一套丰富的 STM32 固件库。使用固件库开发应用系统可以大大提高用户的开发效率。

5.6.1 基于固件库开发和直接操作寄存器的区别

固件库就是函数的集合，固件库函数的作用是向下负责直接操作寄存器，向上提供用户函数调用的接口(API)。

在 STC15 单片机的学习和开发中，使用 C 语言编程时，通常的做法是直接操作寄存器，比如要控制 P2 口的状态，使用下面的代码直接操作寄存器：

```
P2 = 0x11;
```

而在 STM32 的开发中，同样可以操作寄存器：

```
GPIOx->BRR = 0x0011;
```

这种方法的缺点是，用户只有掌握每个寄存器的用法，才能正确使用 STM32，而 STM32 的

寄存器特别多，记起来很麻烦。为了简化编程，ST 公司推出了官方固件库，该库是一个固件函数包（也称为驱动程序），由程序、数据结构和宏组成，包括对微控制器所有外设的定义和操作。

每个外设驱动都由一组函数组成，这组函数覆盖了该外设的所有操作。每个器件的开发都由一个通用 API（Application Programming Interface，应用编程接口）驱动，API 对该驱动程序的结构、函数和参数名称都进行了标准化。

固件库将寄存器底层操作都装起来，提供一整套 API 供开发者调用。大多数场合下，用户不需要知道操作的是哪个寄存器，只需要知道调用哪些函数即可。通过使用固件函数库，无须深入掌握细节，用户也可以轻松应用每一个外设。因此，使用固态函数库可以大大减少用户的程序编写时间，进而降低开发成本。并且，所有的驱动程序源代码都符合 ANSI-C 标准，不受开发环境影响。

固态函数库通过校验所有库函数的输入值来实现实时错误检测。该动态校验提高了软件的鲁棒性。实时检测适合于用户应用程序的开发和调试，但会增加成本，可以在最终应用程序代码中移去，以优化代码大小和执行速度。由于固件库函数是通用的，并且包含了对所有外设的操作，这必定会影响程序代码的大小和执行速度，因此，对于代码大小和执行速度有严格要求的情况，可以考虑使用直接对寄存器操作来满足要求。

例如，上述控制 BRR 寄存器实现电平控制的功能要求，官方库封装了下面的 API 函数：

```
void GPIO_ResetBits(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)
{
    GPIOx->BRR = GPIO_Pin;
}
```

有了上述函数，就不需要再直接去操作 BRR 寄存器了（操作 BRR 寄存器的工作在 API 函数内部完成），只需要知道 GPIO_ResetBits()函数怎么使用即可。在对外设的工作原理有一定的了解之后，再去看固件库函数，基本上从函数名字就能看出这个函数的功能。

任何处理器，归根结底都是要对处理器的寄存器进行操作。因此，第 6 章给出了汇编语言、C 语言直接操作寄存器和使用固态库函数 3 个版本的程序代码供读者参考。从第 7 章开始，只给出使用固态库函数操作外设的方法，汇编语言和 C 语言直接操作寄存器的方法请读者自行学习。

5.6.2 STM32 固件库

1. STM32 固件库函数

STM32 固件库就是函数的集合。ST 官方库是根据 CMSIS 标准设计的。从图 5-4 可以看出，CMSIS 应用程序的基本结构分为 3 个基本功能层。

（1）核内外设访问层：ARM 公司提供的访问，定义处理器内部寄存器地址以及功能函数。

（2）中间件访问层：定义访问中间件的通用 API，由 ARM 公司提供。

（3）外设访问层：定义硬件寄存器的地址以及外设的访问函数。

从图 5-4 可以看出，CMSIS 层在整个系统中处于中间层，向下负责与内核和各个外设

直接打交道，向上提供实时操作系统用户程序调用的函数接口。芯片生产公司设计的库函数必须按照 CMSIS 规范来设计，这样才能保证系统有良好的可移植性。例如，在使用 STM32 芯片的时候首先要进行系统初始化，CMSIS 规范规定，系统初始化函数名字必须为 SystemInit，所以各个芯片公司写自己的库函数时就必须用 SystemInit 对系统进行初始化。CMSIS 还对各个外设驱动文件的文件名字以及函数名字等规范化。如上述 GPIO_ResetBits 函数名字的定义就是遵循的 CMSIS 规范。

STM32 固件库函数的命名规则如下：

(1) PPP 表示任一外设的缩写，如 GPIO、ADC 等。外设缩写如表 5-2 所示。

表 5-2 外设缩写表

缩 写	外 设 名 称	缩 写	外 设 名 称
ADC	模数转换器	IWDG	独立看门狗
BKP	备份寄存器	NVIC	嵌套中断向量列表控制器
CAN	控制器局域网模块	PWR	电源/功耗控制
CRC	CRC 计算单元	RCC	复位与时钟控制器
DAC	数模转换器	RTC	实时时钟
DMA	直接内存存取控制器	SDIO	SDIO 接口
EXTI	外部中断事件控制器	SPI	串行外设接口
FLASH	闪存存储器	SysTick	系统嘀嗒定时器
FSMC	灵活的静态存储器控制器	TIM	通用定时器
GPIO	通用输入/输出端口	TIM1	高级控制定时器
I2C	I2C 总线接口	USART	通用同步异步接收发射端
I2S	I2S 总线接口	WWDG	窗口看门狗

(2) 系统、源程序文件和头文件命名都以“stm32f10x_”作为开头，例如，stm32f10x_conf.h。

(3) 常量仅被应用于一个文件的，定义于该文件中；被应用于多个文件的，在对应头文件中定义。所有常量都由英文字母大写书写。

(4) 寄存器作为常量处理。它们的名字都以大写英文字母书写。

(5) 外设函数的命名以该外设的缩写加下画线为开头。每个单词的第一个字母都是大写英文字母，例如，SPI_SendData。在函数名中，只允许存在一个下画线，用以分隔外设缩写和函数名的其他部分。

(6) 名为 PPP_Init 的函数，其功能是根据 PPP_InitTypeDef 中指定的参数，初始化外设 PPP，例如，TIM_Init。其中，PPP 表示任一外设的缩写，如表 5-2 所示。

(7) 名为 PPP_DeInit 的函数，其功能为复位外设 PPP 的所有寄存器至默认值，例如，TIM_DeInit。

(8) 名为 PPP_StructInit 的函数，其功能为通过设置 PPP_InitTypeDef 结构中的各种参数来定义外设的功能，例如，USART_StructInit。

(9) 名为 PPP_Cmd 的函数，其功能为使能或者除能外设 PPP，例如，SPI_Cmd。

(10) 名为 PPP_ITConfig 的函数，其功能为使能或者除能来自外设 PPP 某中断源，例如，RCC_ITConfig。

(11) 名为 PPP_DMAConfig 的函数,其功能为使能或者除能外设 PPP 的 DMA 接口,例如,TIM1_DMAConfig。用以配置外设功能的函数,总是以字符串"Config"结尾,例如 GPIO_PinRemapConfig。

(12) 名为 PPP_GetFlagStatus 的函数,其功能为检查外设 PPP 某标志位被设置与否,例如,I2C_GetFlagStatus。

(13) 名为 PPP_ClearFlag 的函数,其功能为清除外设 PPP 标志位,例如,I2C_ClearFlag。

(14) 名为 PPP_GetITStatus 的函数,其功能为判断来自外设 PPP 的中断发生与否,例如,I2C_GetITStatus。

(15) 名为 PPP_ClearITPendingBit 的函数,其功能为清除外设 PPP 中断待处理标志位,例如,I2C_ClearITPendingBit。

2. 变量编码规则

在文件 stm32f10x_type.h 中定义了固件库函数中常用的变量。

1) 普通变量

固态函数库定义了 24 个普通变量类型,它们的类型和大小是固定的。

```
typedef signed long s32;
typedef signed short s16;
typedef signed char s8;
typedef signed long const sc32;                  /* Read Only */
typedef signed short const sc16;                 /* Read Only */
typedef signed char const sc8;                   /* Read Only */
typedef volatile signed long vs32;
typedef volatile signed short vs16;
typedef volatile signed char vs8;
typedef volatile signed long const vsc32;        /* Read Only */
typedef volatile signed short const vsc16;       /* Read Only */
typedef volatile signed char const vsc8;         /* Read Only */
typedef unsigned long u32;
typedef unsigned short u16;
typedef unsigned char u8;
typedef unsigned long const uc32;                /* Read Only */
typedef unsigned short const uc16;               /* Read Only */
typedef unsigned char const uc8;                 /* Read Only */
typedef volatile unsigned long vu32;
typedef volatile unsigned short vu16;
typedef volatile unsigned char vu8;
typedef volatile unsigned long const vuc32;      /* Read Only */
typedef volatile unsigned short const vuc16;     /* Read Only */
typedef volatile unsigned char const vuc8;       /* Read Only */
```

2) 布尔型变量

布尔型变量定义如下:

```
typedef enum
{
    FALSE = 0,
```

```
    TRUE = !FALSE
} bool;
```

3）标志位状态类型(FlagStatus type)

定义标志位类型的两个可能值为“设置”或“重置”(SET 或 RESET)，定义如下：

```
typedef enum
{
    RESET = 0,
    SET = !RESET
} FlagStatus;
```

4）功能状态类型(FunctionalState type)

功能状态类型的两个可能值为“使能”或“除能”(ENABLE 或 DISABLE)，定义如下：

```
typedef enum
{
    DISABLE = 0,
    ENABLE = !DISABLE
} FunctionalState;
```

5）错误状态类型(ErrorStatus type)

错误状态类型类型的两个可能值为“成功”或“出错”(SUCCESS 或 ERROR)，定义如下：

```
typedef enum
{
    ERROR = 0,
    SUCCESS = !ERROR
} ErrorStatus;
```

3. 外设

用户可以通过指向各个外设的指针访问各外设的控制寄存器。这些指针所指向的数据结构与各个外设的控制寄存器布局一一对应。

1）外设控制寄存器结构

文件 stm32f10x_map.h 包含了所有外设控制寄存器的结构。例如，SPI 寄存器结构的声明如下：

```
typedef struct
{
    vu16 CR1;
    u16 RESERVED0;
    vu16 CR2;
    u16 RESERVED1;
    vu16 SR;
    u16 RESERVED2;
    vu16 DR;
    u16 RESERVED3;
    vu16 CRCPR;
    u16 RESERVED4;
```

```
    vu16 RXCRCR;
    u16 RESERVED5;
    vu16 TXCRCR;
    u16 RESERVED6;
} SPI_TypeDef;
```

其中，RESERVEDi(i 为一个整数索引值)表示被保留区域。

2) 外设声明

文件 stm32f10x.h 包含了所有外设的声明。例如，SPI1 外设的声明如下：

```
#define PERIPH_BASE ((u32)0x40000000)
#define APB1PERIPH_BASE PERIPH_BASE
#define APB2PERIPH_BASE (PERIPH_BASE + 0x10000)
…
/* SPI1 Base Address definition */
#define SPI1_BASE (APB2PERIPH_BASE + 0x3000)
…
#define SPI1 ((SPI_TypeDef *) SPI1_BASE)
```

4. 常用的固件函数库文件

常用的固件函数库文件如表 5-3 所示。

表 5-3 常用的固件库函数库文件描述

文 件 名	描 述
stm32f10x_conf.h	该头文件设置了所有使用到的外设，由不同的 Define 语句组成。用户可以在该文件中进行参数设置，可以利用模板使能或除能外设，可以修改外部振荡器的参数
stm32f10x.it.h	该头文件包含了所有的中断处理程序的原型
stm32f10x_it.c	该源文件包含了所有的中断处理程序(如果未使用中断，则所有的函数体都为空)。用户可以加入自己的中断程序代码，对于指向同一个中断向量的多个不同中断请求，可以通过判断外设的中断标志位来确定到底是哪个中断源发生了中断
stm32f10x_lib.h	主头文件，包含了其他头文件
stm32f10x_lib.c	包括所有外设指针的定义、初始化等
stm32f10x.h	该文件包含了外设存储器映像、寄存器数据结构和所有寄存器物理地址的声明
stm32f10x_type.h	该文件包含所有其他文件使用的通用数据类型和枚举
stm32f10x_ppp.h	PPP 外设对应一个头文件，包含了该外设使用的函数原型、数据结构和枚举
stm32f10x_ppp.c	PPP 外设对应的源文件，包含了该外设使用的函数体

5.7 习题

5-1 简述 ARM 微控制器应用系统开发的一般过程。

5-2 固件库函数和 CMSIS 的关系是怎样的？

5-3 对照 STM32F103VET6 的存储器映射，查看 5.4 节给出的程序映像中断向量表各个中断的入口地址。

第 6 章 通用输入/输出接口

CHAPTER 6

从本章开始介绍 STM32F103VET6 微控制器的典型外设。本章介绍 STM32F103VET6 微控制器的通用输入/输出接口(General-Purpose Input/Output,GPIO)的工作模式、结构及使用方法。

6.1 通用输入/输出接口概述

几乎在所有的嵌入式系统应用中,都涉及开关量的输入和输出功能,例如状态指示、报警输出、继电器闭合和断开、按钮状态读入、开关量报警信息的输入等。这些开关量的输入和控制输出都可以通过通用输入/输出接口实现。有些 GPIO 接口具有复用功能,本章仅介绍基本输入/输出功能。

STM32F103VET6 有 80 根多功能双向能承受 5V 电压的快速 I/O 口线。每 16 根口线分为一组,分别为 PA、PB、PC、PD、PE。每个 GPIO 端口有两个 32 位配置寄存器(GPIOx_CRL,GPIOx_CRH),两个 32 位数据寄存器(GPIOx_IDR 和 GPIOx_ODR),一个 32 位置位/复位寄存器(GPIOx_BSRR),一个 16 位复位寄存器(GPIOx_BRR)和一个 32 位锁定寄存器(GPIOx_LCKR)。

GPIO 端口的每个位都可以由软件分别配置成以下模式。

- 输入浮空:浮空(floating)就是逻辑器件的输入引脚既不接高电平,也不接低电平。由于逻辑器件的内部结构,当它输入引脚悬空时,相当于该引脚接了高电平。一般实际运用时,引脚不建议悬空,易受干扰。
- 输入上拉:上拉就是把电压拉高,比如拉到 Vcc。上拉就是将不确定的信号通过一个电阻嵌位在高电平。电阻同时起限流作用。弱强只是上拉电阻的阻值不同,没有什么严格区分。
- 输入下拉:就是把电压拉低,拉到 GND。与上拉原理相似。
- 模拟输入:模拟输入是指传统方式的模拟量输入。数字输入是输入数字信号,即 0 和 1 的二进制数字信号。
- 开漏输出:输出端相当于三极管的集电极。要得到高电平状态需要上拉电阻才行。适合于做电流型的驱动,其吸收电流的能力相对强(一般 20mA 以内)。
- 推挽式输出:可以输出高低电平,连接数字器件;推挽结构一般是指两个三极管分别受两个互补信号的控制,总是在一个三极管导通的时候另一个截止。