第 3 章

嵌入式软件开发介绍

3.1 欢迎进入嵌入式系统编程

如果以前从未进行过微控制器编程，不用担心，做起来也不是很困难。事实上，ARM Cortex-M 处理器使用起来非常容易，尽管本书涉及的处理器架构细节非常多，读者也不必了解全部内容或者成为应用设计专家。只要对 C 编程语言有基本的了解，很快就能在 Cortex-M0 和 Cortex-M0＋处理器上开发简单的应用。

如果使用过其他的微控制器，读者会发现在基于 Cortex-M 的微控制器上编程是非常容易的，由于多数寄存器（如外设）都已经进行了存储器映射，基本上整个程序都可以用 C/C++实现，甚至中断处理也可以全部用 C/C++编写。另外，和其他一些处理器架构不同，对于大多数的一般应用，无须使用编译器相关的语言扩展。

如果只有计算机软件的开发经验，读者会发现微控制器软件的开发有很大的不同。很多嵌入式系统中不存在任何操作系统（这些系统有时被称作裸机）并且没有计算机上的用户接口。

3.2 基本概念

如果是第一次使用微控制器，那么请继续阅读；如果已经进行过微控制器编程，则可以跳过本部分，直接进入 3.3 节。

首先介绍一些基本概念。

3.2.1 复位

程序执行前，微控制器需要被复位到一种已知状态，复位一般由外部的硬件信号产生，例如开发板上的复位按钮（见图 3.1），多数微控制器设备都有一个用于复位的输入引脚。

对于基于 ARM 的微控制器，复位还可由连接到微控制器板的调试器产生，软件开发人员可以通过 IDE（集成开发环境）对微控制器进行复位。有些调试适配器还可以利用调试接

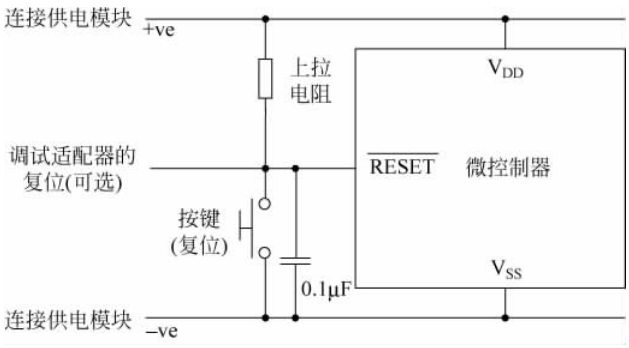

图 3.1　低成本微控制器板的复位连接示例(假定复位引脚为低有效)

头上的特定引脚产生一次复位,对于 ARM Cortex-M 处理器,调试器可以通过调试连接产生复位请求。

复位信号无效后,微控制器硬件内部可能还需要等待一定的时间(如等待时钟振荡器稳定),之后处理器才会开始执行程序。这个延迟通常非常小,用户一般察觉不出来。

3.2.2　时钟

几乎所有的处理器和数字电路都需要时钟信号才能运行,微控制器一般利用外部晶振来产生参考时钟。有些微控制器还具有内部振荡器(但是 R-C 震荡等方式产生的输出频率不是很准确)。

对于许多现代微控制器,可以利用软件控制所使用的时钟源,并且通过编程锁相环(PLL)和时钟分频器来产生所需的各种频率。因此,微控制器电路可能具有 12MHz 的外部晶振,而处理器系统运行的时钟频率可能会高得多(如超过 100MHz),同时有些外设运行的速度可能是分频后的数值。

为了降低功耗,许多微控制器都允许软件开/关每个振荡器和 PLL,并且可以关掉每个外设的时钟信号。

3.2.3　电压

所有的微控制器都需要电源才能运行,因此可以在微控制器上找到供电引脚。多数现代微控制器所需的电压都非常低,例如 3V 等,而有些则需要 1.5V 以下的电压。

如果要设计自己的微控制器开发板或者原型电路,需要确认正在使用的微控制器的数据手册以及微控制器连接部件的电压。例如,中继开关等一些外部接口可能需要 5V 的信号,若是接到 3V 微控制器产生的 3V 输出信号上则会无法工作。

如果是设计自己的开发板,还应该确保供电电压是校准过的,从电网供电到 DC 适配器的电压可能都是不准的,随时都可能会升高或降低,因此在未加稳压器的情况下,是不适合微控制器使用的。

3.2.4 输入和输出

和个人计算机不同，多数嵌入式系统并不具备显示器、键盘和鼠标。可用的输入和输出可能仅限于简单的电子接口，例如数字和模拟输入输出(I/O)、UART、I2C 以及 SPI 等。许多微控制器还提供 USB、以太网、CAN、图形 LCD 以及 SD 卡接口，这些接口由微控制器内的外设控制。

对于基于 ARM 的微控制器，外设由经过了存储器映射的寄存器控制(访问寄存器的实例在本章的 3.3.2 节中介绍)。有些外设要比 8 位和 16 位微控制器中的外设更加复杂，在设置外设时会涉及更多的寄存器。

一般来说，外设的初始化过程由以下步骤组成：

- 编程时钟控制回路以使能连接到外设的时钟信号，如果需要还要设置相应的 I/O 引脚。对于许多低功耗微控制器，为了降低功耗，用于芯片中不同部分的时钟信号可以单独打开或关闭。多数时钟信号默认是关着的，在设置外设前需要对其进行使能。有些情况下，还需要使能总线系统的时钟信号。
- 进行 I/O 配置，多数微控制器会复用 I/O 引脚以实现更多的用途，为使外设接口正常工作，需要对 I/O 引脚进行设置(如复用器的配置寄存器)。另外，有些微控制器 I/O 引脚的电气特性也可以配置，这时需要在 I/O 配置时加入一些步骤。
- 外设配置，多数接口外设中存在多个用于行为控制的可编程寄存器，为使外设正常工作，通常需要遵循一定的编程顺序。
- 中断配置，若外设操作需要中断处理，则需要加入设置中断控制器(如 Cortex-M 处理器中的 NVIC)的步骤。

为便于软件开发，多数微控制器厂商提供了外设/设备驱动库。虽然有设备驱动库，仍然有由应用决定的一定量的底层工作，例如，若需要用户接口，为了实现用户友好以及优秀的嵌入式系统，可能需要开发自己的用户接口函数(注：还有用于创建 GUI 的商业版中间件)。但是，微控制器厂商提供的设备驱动库确实给嵌入式应用开发带来了方便。

对于大多数深度嵌入式系统的开发，丰富的用户接口是没有必要的。但是，LED、DIP 开关以及按键等简单接口虽然只能传递一些基本的信息，为有助于软件调试，一个简单的文字输入/输出控制台程序还是非常有用的，可以通过微控制器上的 UART 接口，利用简单的 RS232 连接连到计算机上的 UART 接口(或者通过 USB 适配器)来实现。根据这种设计，我们可以利用终端应用实现微控制器上文字消息的显示以及用户输入(见图 3.2)。要了解创建这种消息通信的细节，可以参考第 17 章和 18 章的内容。

3.2.5 嵌入式软件程序流程介绍

应用处理流程有多种结构，这里只介绍一些基本概念，请注意，和计算机编程不同，多数嵌入式应用的程序流程都没有结束。

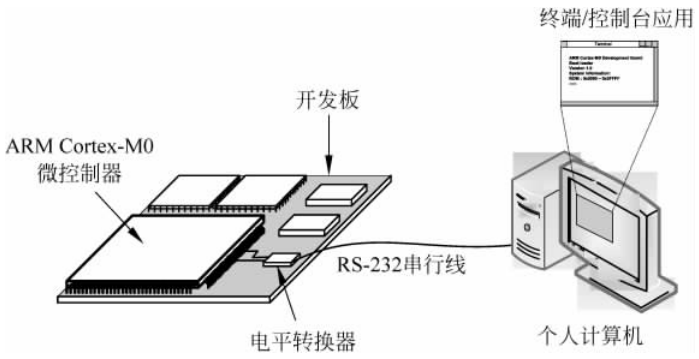

图 3.2　利用 UART 接口实现用户输入和输出

1）轮询

对于简单的应用，轮询（见图 3.3，有时也被称作超级循环）实现起来非常容易而且特别适合简单任务。

但是，当应用变得复杂且需要更高的处理性能时，轮询就不合适了。例如，如果某个进程需要花费较长的时间，则其他的进程可能在一段时间内无法得到服务。使用轮询的另外一个弊端在于，即使没有要处理的任务，处理器也必须一直执行轮询程序，这样降低了能耗效率。

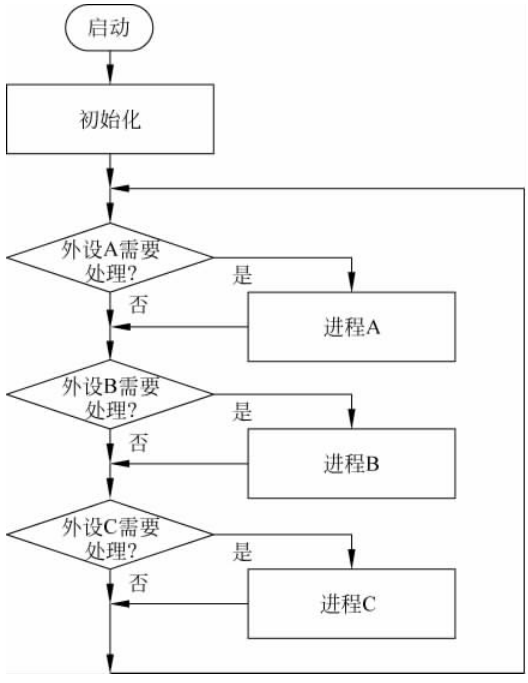

图 3.3　简单应用处理的轮询方式

2）中断驱动

对于有低功耗要求的应用，可以在中断服务程序中执行处理任务，在没有任务执行时，处理器就可以进入休眠模式。中断一般是外部或片上外设产生的，会将处理器唤醒。

在中断驱动的应用中（见图 3.4），来自不同设备的中断可被设置为不同的优先级。即便是在低优先级的中断服务正在执行时，高优先级的中断也可以得到执行，而低优先级的中断则会暂时停止，因此高优先级中断的等待时间就变短了。

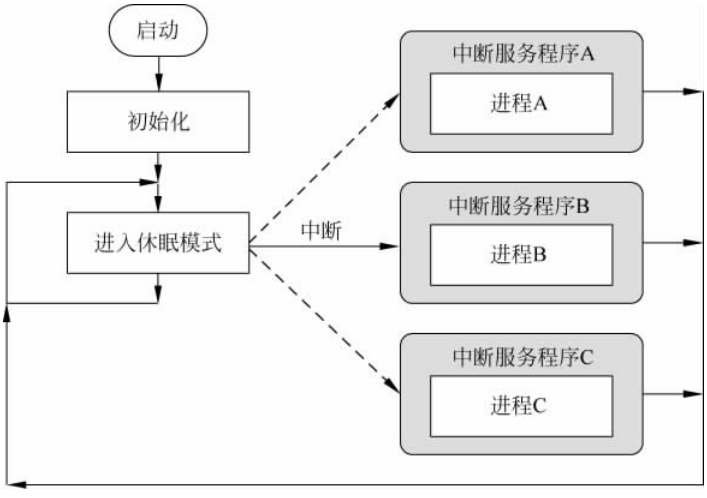

图 3.4　中断驱动应用

许多情况下，应用可以组合使用轮询和中断这两种方式，中断服务程序和应用进程可以利用软件变量进行信息传递（见图 3.5）。

将外设处理任务分成中断服务程序和运行在主程序中的进程后，可以降低中断服务的持续时间，使得更低优先级的中断服务也能得到更多执行的机会。同时，系统仍然可以在没有任务需要执行时进入休眠模式。如图 3.5 所示，应用被分为进程 A、进程 B 和进程 C，但是有些情况下将应用分成单独的部分是不太容易的，可能需要写成一个较大的进程。尽管如此，外设中断也是可以执行的。

3）处理并发进程

有些情况下，应用进程需要花费相当长的时间才能结束，因此无法在如图 3.5 所示的一个大循环中处理。若进程 A 占用太长的时间，进程 B 和进程 C 就无法快速响应外设请求，这样可能会导致系统失败。常见的解决方案如下：

（1）将长的处理任务划分为多个状态，每次进程需要运行时，只执行一个状态。

（2）利用实时操作系统（RTOS）管理多任务。

对于第一种方法（见图 3.6），进程被分为了多个部分和跟踪进程状态的软件变量，进程每次执行时，都会更新状态信息，这样进程再执行时，就可以继续之前的处理了。

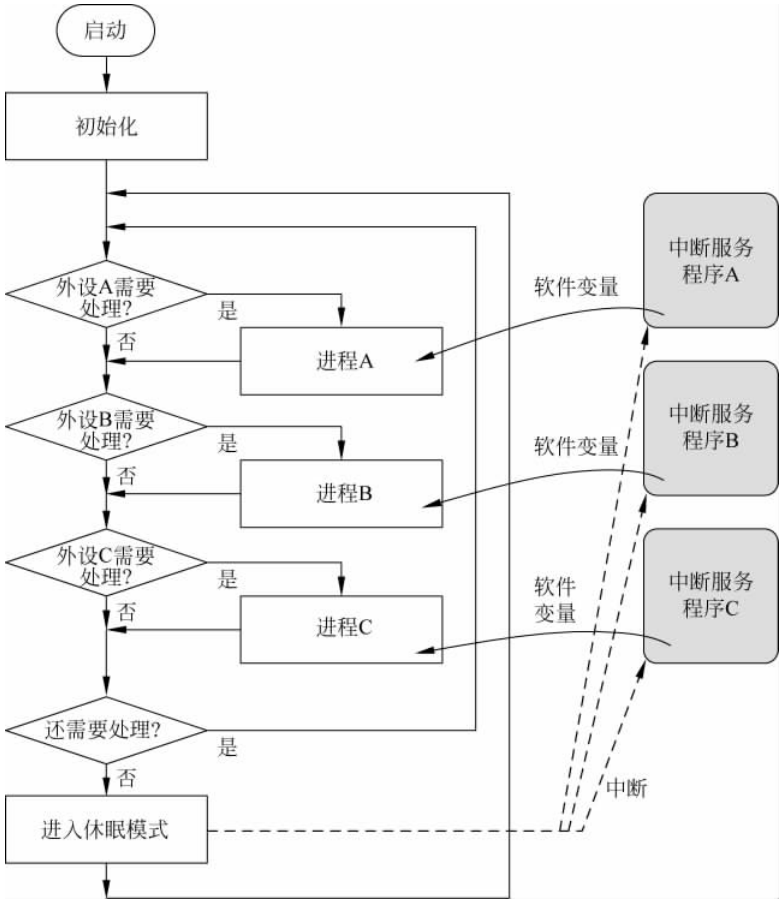

图 3.5　轮询和中断驱动应用的组合

由于进程的执行路径变短，主循环中的其他进程就可以得到更多执行的机会。尽管处理所需的总时间不变(或者由于状态保存和恢复的开销而导致时间稍微增加)，但系统的响应速度却提高了。然而，在应用变得更复杂时，拆分应用任务是不现实的。

对于更加复杂的应用，可能就会用到 RTOS(见图 3.7)，RTOS 在执行多个任务时将处理器执行时间分为多个时间片，并给每个任务分配相应的时间片。在每个时间片结束时，定时器会产生中断并触发确定是否应该执行上下文切换的 RTOS 任务调度器。如果需要执行上下文切换，任务调度器会暂停当前任务的执行，并切换到下一个准备就绪的任务。

因为可以保证所有任务在一定时间内执行，RTOS 的使用提高了系统的响应速度，第 20 章中有使用 RTOS 的示例。

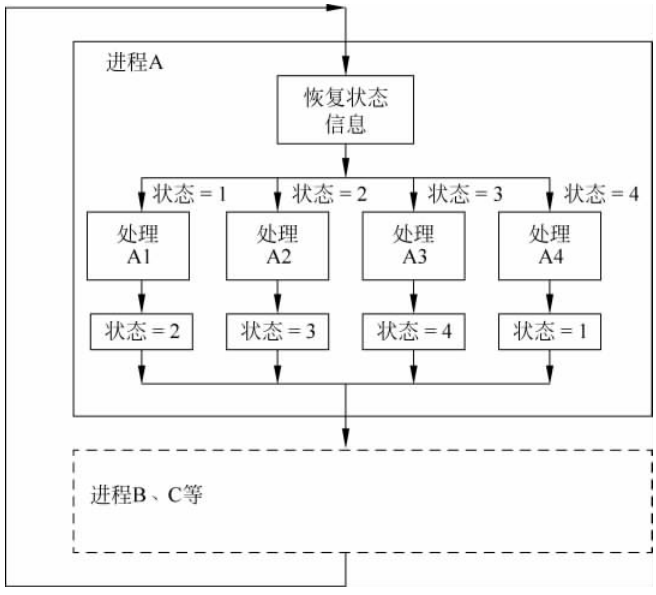

图 3.6　在应用循环内将进程分为多个部分

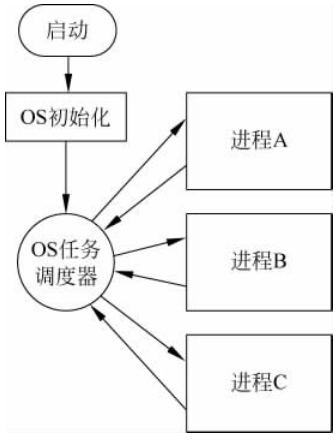

图 3.7　利用实时操作系统处理多个并发应用进程

3.2.6　编程语言选择

对于多数项目，Cortex-M 处理器在编程时可以选择 C/C++语言、汇编语言或者两者的组合。Cortex-M 处理器在设计上是 C 友好的，因此在使用基于 Cortex-M 处理器的微控制器时，无须学习汇编语言。现在还可以使用其他高级语言，例如 Java 和 Matlab/Simulink 等。

对于初学者而言，C/C++语言通常是最佳选择，因其容易学习且多数现代 C 编译器都可以生成 Cortex 微控制器使用的高效代码。表 3.1 对比了 C 语言和汇编语言的差异。

可以在一个工程中混合使用C语言和汇编代码，这样程序中的多数部分可以用C语言编写，而无法用C语言实现的部分则采用汇编。

要了解这方面的详细内容，可以参考第21章的内容。

表3.1　C语言和汇编语言编程对比

语言	优　缺　点
C/C++	**优点**
	易于使用
	可移植
	处理复杂数据结构时简单
	缺点
	对内核寄存器和栈无法直接访问或访问受限
	无法直接控制生成的指令序列
	无法直接控制栈的使用
汇编	**优点**
	可以直接控制每个指令步骤和所有的存储器操作
	可以使用无法用C语言生成的指令
	缺点
	学习时间较长
	处理数据结构困难
	可移植性不高(不同工具链的汇编语法可能不同)

3.3　ARM Cortex-M编程介绍

3.3.1　C编程数据类型

C语言支持多种“标准”数据类型，数据类型的实际情况还要取决于处理器架构和C编译器，对于包括Cortex-M0和Cortex-M0＋在内的ARM处理器，所有的C编译器都支持表3.2所示的数据类型。

表3.2　Cortex-M处理器支持的数据类型

C和C99(stdint.h)数据类型	位数	范围(有符号)	范围(无符号)
char,int8_t,uint8_t	8	－128到127	0到255
short,int16_t,uint16_t	16	－32768到32767	0到65535
int,int32_t,uint32_t	32	－2147483648到－2147483647	0到4294967265
long	32	－2147483648到－2147483647	0到4294967265
long long,int64_t,uint64_t	64	－(2^{63})到(2^{63}－1)	0到(2^{64}－1)
float	32	－3.4028234×10^{38}到3.4028234×10^{38}	

续表

C 和 C99(stdint.h)数据类型	位数	范围(有符号)	范围(无符号)
double	64	$-1.7976931348623157\times10^{308}$到 $1.7976931348623157\times10^{308}$	
long double	64	$-1.7976931348623157\times10^{308}$到 $1.7976931348623157\times10^{308}$	
pointers	32	0x0 到 0xFFFFFFFF	
enum	8/16/32	若没有编译器选项设置，为可能的最小数据类型	
bool(c++),_Bool(C)	8	真或假	
wchar_t	16	0 到 65535	

从其他处理器向 ARM 处理器移植应用时，若数据的大小不同，为保证程序可以正常执行，可能需要修改 C 程序代码。要了解从 8 位和 16 位架构进行软件移植的详细信息，可以参考第 22 章的内容。

在对 Cortex-M0 和 Cortex-M0+进行编程时，位于存储器中的数据变量的地址需要为其大小的倍数，这方面的细节内容将在第 7 章介绍(参见 7.9.1 节“数据对齐”)。

在 ARM 编程中，还可以将数据大小称作字、半字和字节(见表 3.3)。

读者可以在指令集细节等 ARM 文献中找到这些叫法。

表 3.3 ARM 处理器中的数据长度定义

条目	大小
字节	8 位
半字	16 位
字	32 位
双字	64 位

3.3.2 用 C 访问外设

ARM Cortex-M 微控制器的外设是经过存储器映射的，且可以通过数据指针访问。多数情况下，可以使用微控制器厂商提供的设备驱动，这样使得软件开发更加方便，且有利于不同微控制器间的软件移植。若有必要直接访问外设寄存器，则可以使用下面的方法。

若仅要访问几个寄存器，可以将每个外设寄存器都定义为指针。

1) 利用指针定义和访问 UART 外设寄存器的实例

```
#define UART_BASE 0x40003000  //ARM Primecell PL011 基地址
#define UART_DATA (* ((volatile unsigned long *)(UART_BASE + 0x00)))
#define UART_RSR (* ((volatile unsigned long *)(UART_BASE + 0x04)))
#define UART_FLAG (* ((volatile unsigned long *)(UART_BASE + 0x18)))
#define UART_LPR (* ((volatile unsigned long *)(UART_BASE + 0x20)))
```

```
#define UART_IBRD ( * ((volatile unsigned long * )(UART_BASE + 0x24)))
#define UART_FBRD ( * ((volatile unsigned long * )(UART_BASE + 0x28)))
#define UART_LCR_H ( * ((volatile unsigned long * )(UART_BASE + 0x2C)))
#define UART_CR ( * ((volatile unsigned long * )(UART_BASE + 0x30)))
#define UART_IFLS ( * ((volatile unsigned long * )(UART_BASE + 0x34)))
#define UART_MSC ( * ((volatile unsigned long * )(UART_BASE + 0x38)))
#define UART_RIS ( * ((volatile unsigned long * )(UART_BASE + 0x3C)))
#define UART_MIS ( * ((volatile unsigned long * )(UART_BASE + 0x40)))
#define UART_ICR ( * ((volatile unsigned long * )(UART_BASE + 0x44)))
#define UART_DMACR ( * ((volatile unsigned long * )(UART_BASE + 0x48)))
//----- UART 初始化 ----
void uartinit(void)                    //ARM Primecell PL011 的简单初始化
{
  UART_IBRD = 40;                      //ibrd : 25MHz/38400/16 = 40
  UART_FBRD = 11;                      //fbrd : 25MHz/38400 - 16 * ibrd = 11.04
  UART_LCR_H = 0x60;                   //线控 : 8N1
  UART_CR = 0x301;                     //cr : 使能 TX 和 RX, UART 使能
  UART_RSR = 0xA;                      //清除缓冲溢出
}
//----- 发送一个字符 ----
int sendchar(int ch)
{
  while (UART_FLAG & 0x20);            //忙,则等待
  UART_DATA = ch;                      //写字符
  return ch;
}
//----- 收到一个字符 ----
int getkey(void)
{
  while ((UART_FLAG & 0x40) == 0); //无数据,则等待
  return UART_DATA;                    //读取字符
}
```

这种方法非常适合简单应用,若系统中存在同一外设的多个实例,则需要为这些外设分别定义寄存器,这会增加维护的难度。另外,将每个寄存器定义为单独的指针,也可能会增加程序代码,因为每次寄存器访问都需要一个位于 Flash 存储器中的 32 位地址常量。

为简化代码,可将外设寄存器定义成数据结构体,并将外设定义为指向这个数据结构体的存储器指针。

2) 利用结构体指针访问数据结构定义的 UART 寄存器实例

```
typedef struct {                       //ARM Primecell PL011 的基地址
volatile unsigned long DATA;           //0x00
volatile unsigned long RSR;            //0x04
unsigned long RESERVED0[4];            //0x08 - 0x14
volatile unsigned long FLAG;           //0x18
unsigned long RESERVED1;               //0x1C
```

```
volatile unsigned long LPR;           //0x20
volatile unsigned long IBRD;          //0x24
volatile unsigned long FBRD;          //0x28
volatile unsigned long LCR_H;         //0x2C
volatile unsigned long CR;            //0x30
volatile unsigned long IFLS;          //0x34
volatile unsigned long MSC;           //0x38
volatile unsigned long RIS;           //0x3C
volatile unsigned long MIS;           //0x40
volatile unsigned long ICR;           //0x44
volatile unsigned long DMACR;         //0x48
} UART_TypeDef;
#define Uart0 ((UART_TypeDef *) 0x40003000)
#define Uart1 ((UART_TypeDef *) 0x40004000)
#define Uart2 ((UART_TypeDef *) 0x40005000)

/* ----- UART初始化 ---- */
void uartinit(void)                   //Primecell PL011的简单初始化
{
  Uart0->IBRD = 40;                   //ibrd：25MHz/38400/16 = 40
  Uart0->FBRD = 11;                   //fbrd：25MHz/38400 - 16*ibrd = 11.04
  Uart0->LCR_H = 0x60;                //线控：8N1
  Uart0->CR = 0x301;                  //cr：使能TX和RX，UART使能
  Uart0->RSR = 0xA;                   //清除缓冲溢出
}
/* ----- 发送一个字符 ---- */
int sendchar(int ch)
{
while (Uart0->FLAG & 0x20);           //忙，则等待
Uart0->DATA = ch;                     //写字符
return ch;
}
/* ----- 收到一个字符 ---- */
int getkey(void)
{
  while ((Uart0->FLAG & 0x40) == 0);  //无数据，则等待
  return Uart0->DATA;                 //读字符
}
```

在上面的例子中，UART #0 的 IBRD（整数波特率分频器）寄存器通过符号 Uart0->IBRD 访问，而 UART #1 的同一个寄存器则需要通过 Uart1->IBRD 访问。

按照这种处理，外设的一个寄存器数据结构体可被多个实例共用，这样有利于代码维护。另外，由于所需的立即数存储减少，编译后的代码也更小。

经过进一步修改后，可得到多个实例共用的外设函数，访问每个实例则需要传递相应的参数。

3）UART 寄存器定义实例和利用参数传递支持多个 UART 的驱动代码

```
typedef struct {                        //ARM Primecell PL011 的基地址
volatile unsigned long DATA;            //0x00
Volatile unsigned long RSR;             //0x04
unsigned long RESERVED0[4];             //0x08 - 0x14
volatile unsigned long FLAG;            //0x18
unsigned long RESERVED1;                //0x1C
volatile unsigned long LPR;             //0x20
volatile unsigned long IBRD;            //0x24
volatile unsigned long FBRD;            //0x28
volatile unsigned long LCR_H;           //0x2C
volatile unsigned long CR;              //0x30
volatile unsigned long IFLS;            //0x34
volatile unsigned long MSC;             //0x38
volatile unsigned long RIS;             //0x3C
volatile unsigned long MIS;             //0x40
volatile unsigned long ICR;             //0x44
volatile unsigned long DMACR;           //0x48
} UART_TypeDef;
#define Uart0 (( UART_TypeDef * ) 0x40003000)
#define Uart1 (( UART_TypeDef * ) 0x40004000)
#define Uart2 (( UART_TypeDef * ) 0x40005000)


/* ----- UART 初始化 ---- */
void uartinit(UART_Typedef *uartptr)
{
    uartptr->IBRD = 40;                 //ibrd：25MHz/38400/16 = 40
    uartptr->FBRD = 11;                 //fbrd：25MHz/38400 - 16*ibrd = 11.04
    uartptr->LCR_H = 0x60;              //线控：8N1
    uartptr->CR = 0x301;                //cr：使能 TX 和 RX, UART 使能
    uartptr->RSR = 0xA;                 //清除缓冲溢出
}
/* ----- 发送一个字符 ---- */
int sendchar(UART_Typedef *uartptr, int ch)
{
    while (uartptr->FLAG & 0x20);       //忙,则等待
    uartptr->DATA = ch;                 //写字符
    return ch;
}
/* ----- 收到一个字符 ---- */
int getkey(UART_Typedef *uartptr)
{
    while ((uartptr ->FLAG & 0x40) == 0);     //无数据,则等待
    return uartptr ->DATA;                    //读取字符
}
```

多数情况下，外设被定义为32位字，这是因为多数外设都连到了将所有传输作为32位处理的外设总线（使用APB协议，参见2.3节）。有些外设可能还会被连到处理器的系统总线（AHB协议，支持各种传输大小，参见2.3节），此时寄存器可能还会以其他宽度访问。请参考微控制器的用户手册，以确定每个外设所支持的传输大小。

需要注意的是，在定义外设访问的存储器指针时，应该在寄存器定义中使用“volatile”关键字，以确保编译器生成正确的访问。

3.3.3 程序映像内有什么

除了所创建的程序代码，程序映像中还存在多种软件部件：

- 向量表；
- 服务处理/启动代码；
- C启动代码；
- 应用代码；
- C运行时库函数；
- 其他数据。

在本节中，将会简单介绍以下这些部件。

1）向量表

ARM Cortex-M处理器的向量表中包含每个异常和中断的起始地址，而对于Cortex-M0和Cortex-M0＋处理器，复位后，向量表定义在存储器空间的起始位置（地址0x00000000）。向量表的一个字还定义了主栈指针的初始值，下一章将会做进一步的介绍（4.2节“编程模型”），向量表是和设备相关的（取决于所支持的异常），一般位于启动代码中。

2）复位处理/启动代码

复位处理是可选的，若没有复位处理，则会直接执行C启动代码。复位处理中的代码在处理器从复位中退出时会立即执行，有些情况下其中还会存在一些硬件初始化代码。对于使用CMSIS-CORE（Cortex-M处理器用的软件框架，本章稍后将会介绍）的工程，复位处理执行“SystemInit()”函数，其会在跳转到C启动代码前设置时钟和PLL。

启动代码一般由微控制器厂商提供，有时还会出现在工具链软件中，其可能是C代码或汇编代码。

3）C启动代码

若用C/C++或其他许多高级语言编程，处理器需要执行一些程序代码以设置程序执行环境（如设置全局变量以及SRAM中的初始值），对于加载时未初始化的数据存储器中的变量，需要将它们初始化为0。若应用需要使用malloc()等C函数，C启动代码还需要初始化控制堆存储的数据变量，初始化后，C启动代码会跳转到“main”程序的开头。

C启动代码会被工具链自动生成，因此是和工具链相关的，如果是完全用汇编编写的程序，则可能会不存在启动代码。对于ARM编译器，C启动代码的标号为“__main”，而GNU

C 编译器生成的启动代码的标号一般为“_start”。

4）应用代码

应用代码一般是从 main()开始的，其中包括用以执行所需任务的应用程序代码生成的指令，除了指令序列外，还有其他类型的数据如下所示：

- 变量初始值，函数或子例程中的局部变量需要被初始化，在程序执行期间会设置这些初始值。
- 程序代码中的常量，应用代码中的常量数据有多种用法：数据值、地址或外设寄存器以及常量字符串等。这些数据一般被称作文本数据，且在程序映像中会以多个名为文本池的形式分组出现。
- 有些应用中可能还包含查找表、图形映像数据（如位图）等其他常量数据。

5）C 库代码

在使用某些 C/C++函数时，C 库代码会被链接器插入程序映像中。另外，在进行浮点运算和除法等数据处理任务时可能也会包含 C 库代码。Cortex-M0 和 Cortex-M0＋处理器不支持除法指令，除法运算一般由 C 库中的除法函数执行。

为了应对不同用途，有些开发工具会提供各种版本的 C 库。例如，对于 Keil MDK 或 ARM Development Studio 5（DS-5），可以选择使用名为 Microlib 的特殊版本的 C 库。Microlib 面向微控制器应用，并且体积非常小，但是无法提供标准 C 库的所有特性。对于不需要很高的数据处理能力且存储器需求非常紧张的应用，Microlib 是降低代码大小的好方法。

对于不同的应用，C 库代码可能不会出现在简单的 C 应用（无 C 库函数调用）或纯汇编语言工程中。

向量表必须放在存储器映射的开头处，程序映像的其他部分就没有什么限制了。有些情况下，若程序存储器中各部分的布局有特殊的要求，则可以利用链接器脚本控制程序映像的生成。

6）其他数据

程序映像中还包含其他数据，例如全局或静态变量的初始值等。

3.3.4　SRAM 中的数据

处理器系统中的 SRAM 具有以下用途：

（1）数据，存储在 RAM 末端的数据通常包含全局和静态变量。（注：局部变量可以存储在处理器的寄存器中，或者放在栈中以减少 RAM 的使用，未使用函数中的局部变量不会占用存储器空间。）

（2）栈，栈存储的作用包括临时数据存储（一般的栈 PUSH 和 POP 操作）、局部变量的存储器空间、函数调用时的参数传递以及异常流程中的寄存器保存等。Thumb 指令集在处理器数据访问时非常高效，其使用栈指针（SP）相关的寻址模式，并且在很小的指令开销下就可以访问栈存储中的这些数据。

（3）堆，堆存储是可选的，用于C函数中存储器空间的动态分配，如“alloc()”、“malloc”和其他使用这个功能的函数。为保证这些函数能够正确地分配存储空间，C启动代码需要初始化堆存储及其控制变量。

对于ARM处理器，还可以将程序代码复制到内存中并从这里开始执行，但是对于多数微控制器应用，程序一般从Flash等非易失性存储器中开始执行。

将这些数据放到SRAM中的方法有很多种，一般是和工具链相关的。对于不具备OS的简单应用，SRAM中的存储器分布情况如图3.8所示。ARM处理器的栈指针会被初始化为栈存储空间的顶部，在使用栈PUSH操作将数据放入栈中时会减小，而当利用POP操作将数据移出时会增大。

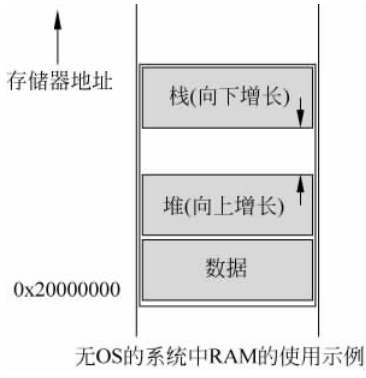

图3.8 单任务系统内存使用实例（无操作系统）

对于具有嵌入式OS（如μClinux）或RTOS（如Keil RTX）的微控制器系统，每个任务的栈都是独立的。许多OS都允许软件开发人员定义每个任务/线程所需的栈大小，有些OS可能会将RAM分割为多个部分，并将每个部分分配给一个任务，其中都包含各自的数据、栈和堆（见图3.9）。

具有RTOS的多数系统会使用如图3.9左侧的数据布局，这里的全局和静态变量以及堆存储都是共用的。

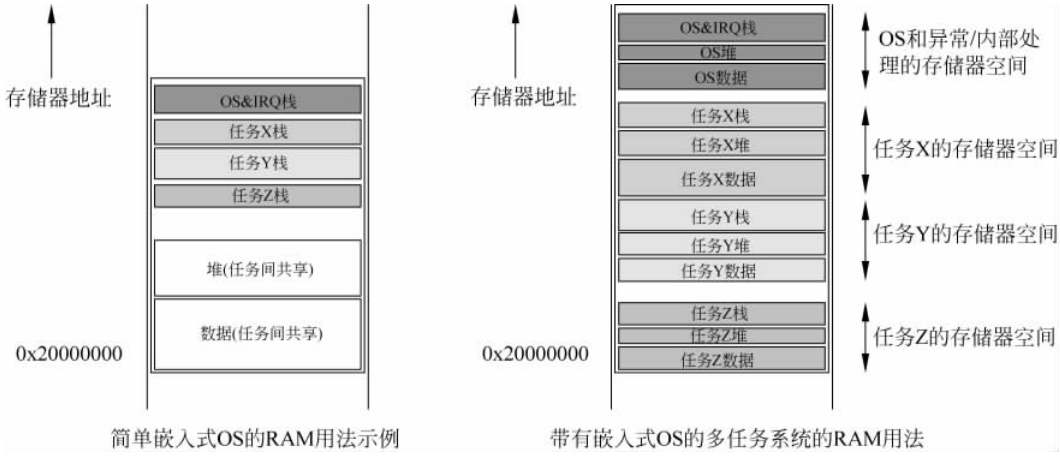

图3.9 多任务系统内存使用示例（有操作系统）

3.3.5 微控制器启动时会发生什么

多数现代微控制器使用片上Flash存储器来存储编译后的程序，Flash存储器的程序是

以二进制的形式存放的，因此用 C 编写的程序在烧入 Flash 存储器前必须进行编译。有些微控制器可能还会包含一个独立的启动 ROM，其中存在一段 Bootloader 程序，该程序会在微控制器执行 Flash 存储器中的用户程序前启动。多数情况下，只有 Flash 存储器中的程序代码才能被修改，而 Bootloader 中的程序代码则已被生产商固化。

编程完 Flash 存储器（或其他类型的程序存储器）后，程序可被处理器访问。处理器复位后会执行复位流程（见图 3.10）。

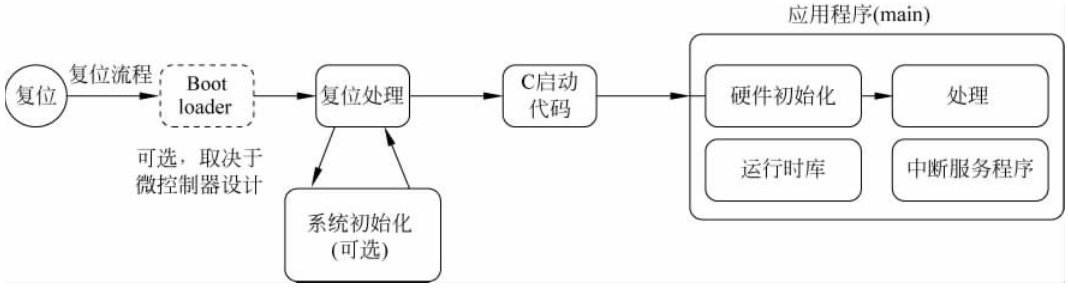

图 3.10　处理器复位之后的工作步骤

在复位流程中，处理器从向量表中取出初始栈指针以及复位向量（异常的起始地址）的数值，然后执行启动代码中的复位处理，复位处理可以选择执行一些硬件的初始化工作。

对于用 C 开发的应用，C 启动代码会在进入主应用代码前执行（见图 3.11），C 启动代码会初始化应用所需的变量和存储器，并被 C 开发组件插入程序映像中。

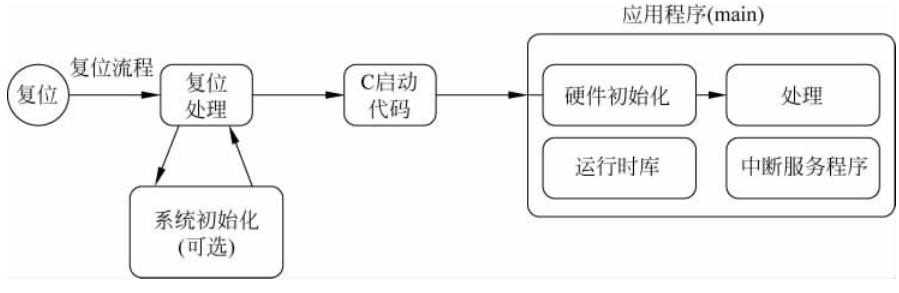

图 3.11　微控制器启动后，C 启动代码所做的工作

在执行了 C 启动代码后，应用就会被启动起来（见图 3.12），应用程序一般由以下部分组成：

- 硬件的初始化；
- 应用的处理部分；
- 中断服务例程。

另外，应用中可能还会存在 C 库函数，此时 C 编译器/链接器会将所需的库函数加入到编译后的程序映像中。

硬件的初始化可能涉及多个外设以及 Cortex-M0/M0＋处理器中的一些系统控制寄存

器和中断控制寄存器，若复位处理未执行系统时钟控制和 PLL 初始化，此处也需要进行处理。在初始化外设后，就可以执行应用处理部分的程序了。

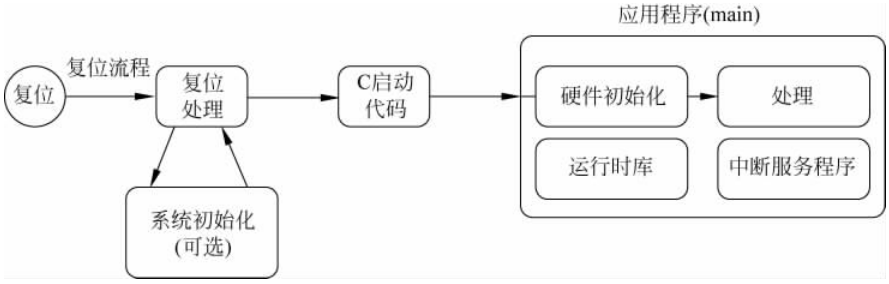

图 3.12　微控制器启动后，应用程序的工作

3.4　软件开发流程

ARM 微控制器可用的开发工具链有很多种，其中多数支持 C/C++以及汇编语言，多数情况下，程序生成流程可以被总结为如图 3.13 所示的形式。

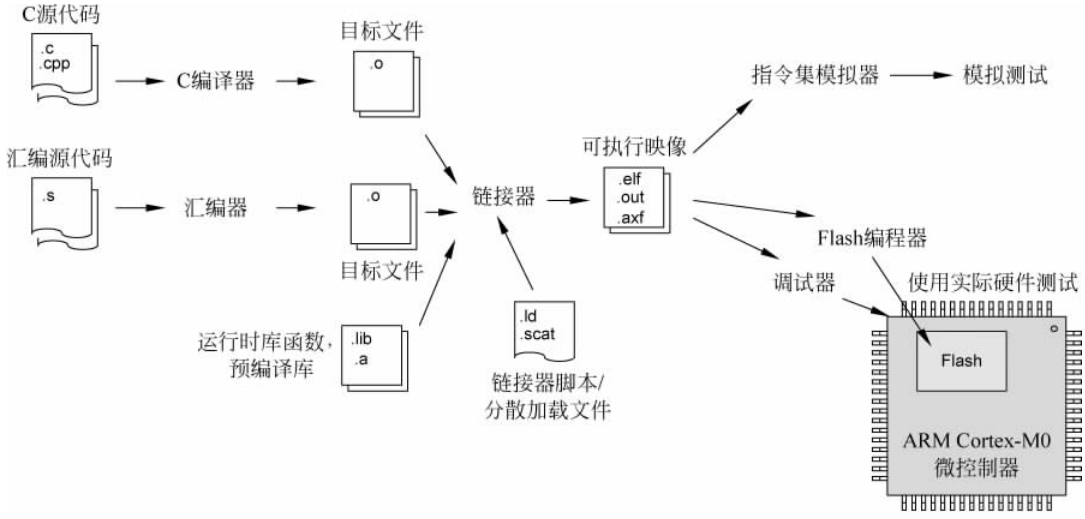

图 3.13　典型的程序生成流程

对于多数简单应用，程序可以全部用 C 语言实现。C 编译器将 C 程序编译为目标文件，然后利用链接器生成可执行程序映像文件。对于 GNU C 编译器，编译和链接阶段一般会被合并为一个步骤。

需要汇编编程的工程利用汇编器将汇编源代码生成目标代码，然后目标文件和其他的目标文件链接在一起以生成可执行映像。

除了程序代码，目标文件和可执行映像中可能还存在各种调试信息。

有的开发工具可以利用命令行选项指定链接器的存储器布局，但是，对于使用 GNU C 编译器的工程，一般需要用链接器脚本来指定存储器布局。当存储器的布局非常复杂时，其他开发工具也需要用链接器脚本。对于 ARM 开发工具，链接器脚本通常被称作分散加载文件。若使用的是 Keil 微控制器开发套件(MDK)，则分散加载文件可从存储器布局窗口自动生成。如果需要，可以使用分散加载文件。

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 RAM 中并进行测试。整个过程相当简单，多数开发组件都有界面友好的 IDE。再加上在线调试器(有时也被称作在线模拟器(ICE)、调试探测或 USB-JTAG 适配器)，经过几个步骤以后，就可以创建一个工程并且在构建自己的应用后将嵌入式应用下载到微控制器中(见图 3.14)。

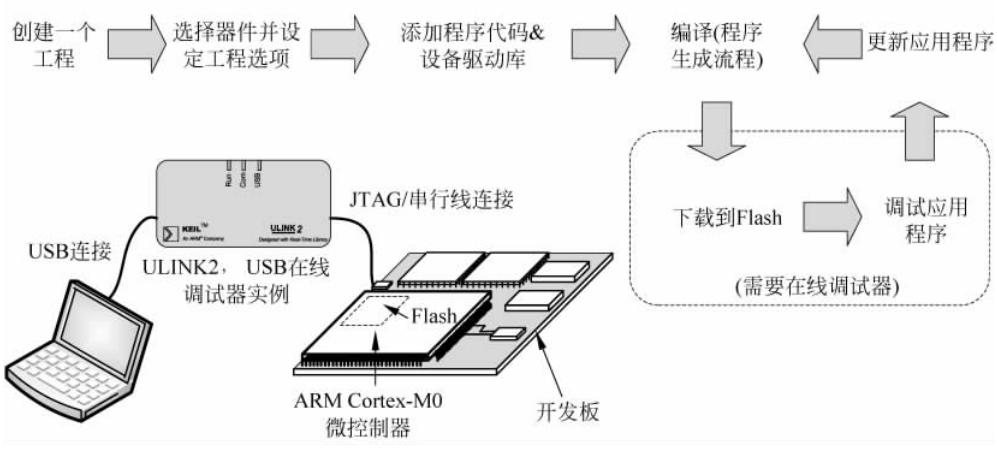

图 3.14　开发流程示例

许多情况下，在线调试器都需要将调试主机(个人计算机)连接到目标板，Keil ULINK2(见图 3.15)就是其中一种产品，且可用于 Keil 微控制器开发套件。

图 3.15　ULINK2 USB-JTAG 适配器

Flash 编程功能可由开发组件中的调试器软件执行，有时可以从微控制器厂商的网站下载至 Flash 编程工具。然后就可以在微控制器上运行程序并执行测试，将调试器连接到微控制器后，程序执行过程就可控了，并且芯片执行的操作也可以被观察到。所有这些功能都可通过 Cortex-M 处理器的调试接口执行(见图 3.16)。

对于简单的程序代码，还可以利用模拟器来测试程序。这样可以观察到程序执行的全部过程，并且测试时也不需要实际硬件。有些开发组件提供的模拟器可以模拟外设的行为，例如，Keil MDK 提供了许多基于 ARM Cortex-M 处理器的微控制器的设备模拟。

除了不同C编译器表现不同的事实外，不同开发组件还提供了不同的C语言扩展特性，并且汇编语言也有不同的语法和伪指令。本书的第5章、第6章和第21章将会介绍ARM开发工具（包括ARM Development Studio 5和Keil MDK）和GNU编译器的汇编语法。另外，不同的开发组件在调试、工具方面的特性会有所差异，并且对调试硬件的支持也不同。

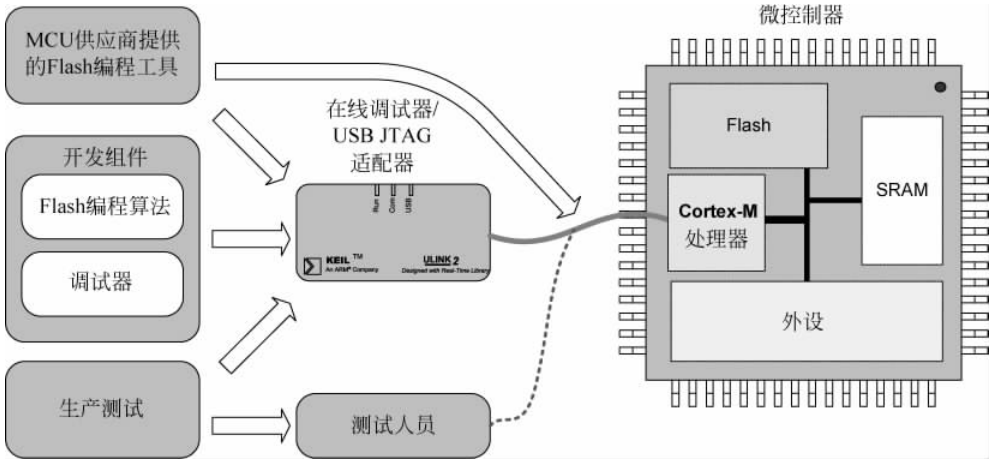

图3.16　Cortex-M处理器调试接口的各种用途

3.5　Cortex微控制器软件接口标准

3.5.1　CMSIS介绍

随着嵌入式系统复杂度的增加，软件代码的兼容性和可重用性也变得愈加重要。具有可重用性的软件开发可以减少后续项目的开发时间，并能让产品更快地推向市场；软件的兼容性则对第三方软件部件的使用有很大的好处。例如，嵌入式系统工程可能会涉及如下几个软件部件：

- 内部开发人员开发的软件；
- 重用的其他工程的软件；
- 微控制器厂商的设备驱动库；
- 嵌入式OS/RTOS；
- 通信协议栈和编解码器等第三方软件产品。

如果在同一个项目中使用了所有这些软件部件，那么对于许多大型软件工程来说，兼容性就是一个非常关键的问题。另外，系统开发人员还想在以后的工程中重用现在已经开发的软件，而且使用的处理器可能还是不同的。

为使这些软件产品具有高度的兼容性，并提高软件的可移植性和可重用性，ARM同各

家微控制器供应商、工具供应商和软件解决方案提供商一同开发了 CMSIS——一个涵盖了大多数 Cortex-M 处理器和 Cortex-M 微控制器产品的软件框架。

CMSIS 文件被集成在微控制器供应商提供的设备驱动库软件包中，并提供了访问中断控制和系统控制功能等处理器特性的标准接口（见图 3.17）。这些处理器特性访问函数中的许多函数都是适用于所有 Cortex-M 处理器的，因此在基于这些处理器的微控制器间移植程序也会非常简单。

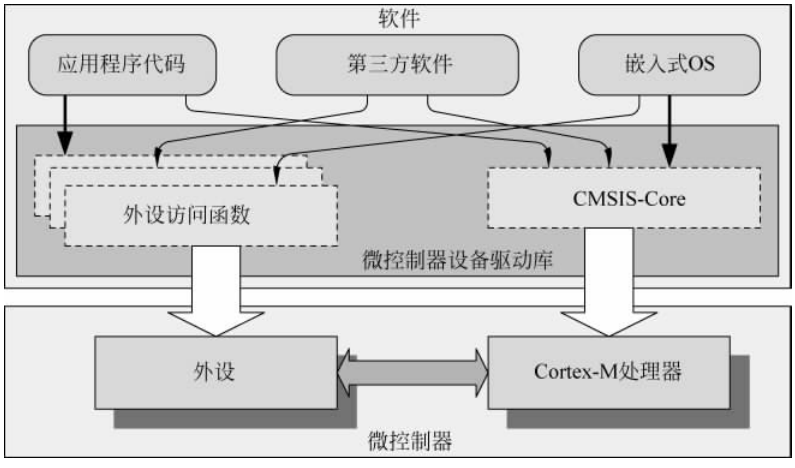

图 3.17　CMSIS-CORE 针对处理器特性提供标准化的操作函数

CMSIS-CORE 对于多家微控制器厂商都是标准的，而且还被多家 C 编译器厂商支持。例如，它可用于 Keil MDK、ARM Development Studio 5（DS-5）、IAR Embedded Workbench TASKING 编译器以及 Atollic TrueStudio 等多个基于 GNU 的编译器组件。

CMSIS-CORE 是 CMSIS 项目的第一部分，并已经逐渐地覆盖了更多的处理器，同时增加了对其他工具链的支持。多年来，CMSIS 已经扩展为多个项目（见表 3.4）。

表 3.4　现有 CMSIS 项目

CMSIS 项目	描　　述
CMSIS-CORE	包括处理器特性应用编程接口（API）、寄存器定义在内的软件框架，与设备驱动库的形式差不多
CMSIS-DSP	适用于所有 Cortex-M 处理器的免费 DSP 软件库
CMSIS-RTOS	应用代码和 RTOS 产品间的 API 接口规范，中间件可以借其同多个 RTOS 配合使用
CMSIS-PACK	软件供应商可以利用它实现软件包，很容易就可以集成到开发组件中
CMSIS-Driver	中间件访问常用设备驱动函数的设备驱动 API
CMSIS-SVD	系统视图描述（SVD）基于 XML 文件标准，对微控制器设备内的外设寄存器进行了描述，CMSIS-SVD 文件由微控制器供应商提供，支持 CMSIS-SVD 的调试器则可以引入这些文件并显示外设寄存器的内容

续表

CMSIS 项目	描　　述
CMSIS-DAP	USB 到调试连接适配器的参考设计，开发组件中的调试器可以通过它和 USB 调试适配器通信，这样微控制器供应商可以设计出适用于多种工具链的低成本调试适配器

多个 CMSIS 项目间的关系如图 3.18 所示。

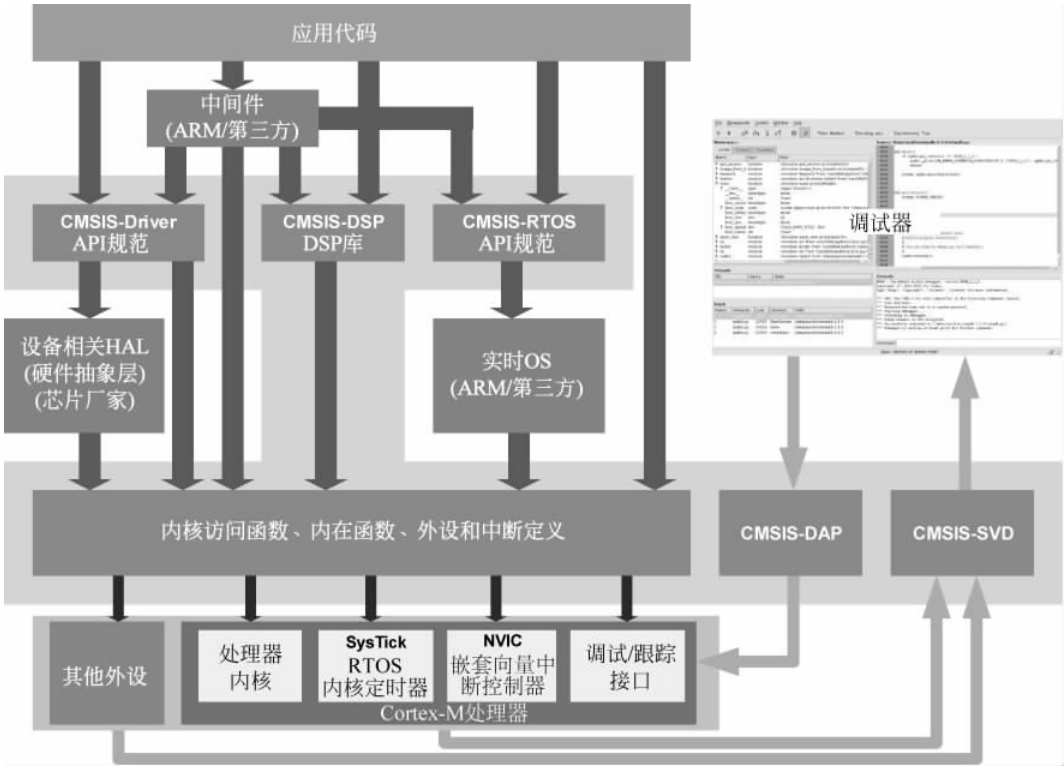

图 3.18　不同 CMSIS 项目间的关系

3.5.2　CMSIS-CORE 所做的标准化

CMSIS 为嵌入式软件提供了以下标准化的内容：

- 标准的访问函数/应用编程接口（API）用以访问内部外设（如 NVIC、系统控制块（SCB）和系统节拍定时器（SysTick）），例如中断控制和 SysTick 的初始化等。本书的多个章节以及附录 C“CMSIS-CORE 快速参考”都会涉及这些函数。
- 处理器内部寄存器的标准定义，为了获得最佳的可移植性，应该使用标准访问函数，但是有些情况下，还需要直接访问这些寄存器。此时，标准的寄存器定义则有助于提高软件的可移植性。

- 访问 Cortex-M 微控制器中特殊指令的标准函数，Cortex-M 处理器中的一些指令无法用普通的 C 代码生成，需要时可以用 CMSIS 提供的这些函数得到，否则，必须用 C 编译器提供的内在函数或者嵌入式/内联汇编语言来实现，但是它们都是和工具链相关的，因此可移植性要差一些。
- 系统异常处理的标准化命名，嵌入式 OS 一般需要系统异常，系统异常有了标准的名称后，嵌入式 OS 在支持不同的设备驱动库时也就更加容易。
- 系统初始化函数的标准化命名，有了系统初始化函数“void SystemInit(void)”，软件开发人员可以很容易地对系统进行设置。
- 名为“SystemCoreClock”的标准软件变量用于确定处理器的时钟频率。
- CMSIS-CORE 还为设备驱动库提供了一个通用平台，每个设备驱动库看起来都是一样的，初学者学习起来更加容易且软件移植也更方便。

CMSIS 是为了保证基本操作的兼容性而开发的，微控制器厂商可以在自己的驱动库中加入其他的函数，以完善他们的软件解决方案，CMSIS 并不会对嵌入式产品的功能加以限制。

3.5.3　CMSIS-CORE 的组织

符合 CMSIS 的设备驱动包括以下内容：

- 内核外设访问层，名称定义、地址定义以及访问内核寄存器和 NVIC 及 SysTick 定时器等内核内部外设的辅助函数。
- 设备外设访问层(MCU 相关)，寄存器名称定义、地址定义和访问外设的设备驱动代码。
- 外设的访问函数(MCU 相关)，外设访问的可选辅助函数，请注意另一个名为 CMSIS-Driver 的项目已经启动，目的是实现一组 API，使得开发的应用代码和中间件能够适合多种微控制器平台。

这些层的作用如图 3.19 所示。

3.5.4　使用 CMSIS-CORE

CMSIS 文件被集成在微控制器供应商提供的设备驱动库软件包中，如果在软件开发中使用了设备驱动库，那么就是已经在使用 CMSIS-CORE 了。如果未使用微控制器提供的设备驱动库，仍然可以使用 CMSIS-CORE：从 ARM 网站(www.arm.com/cmsis)下载 CMSIS 软件包、解压文件并将所需文件添加到自己的工程中。

对于 C 程序代码，一般只需包含微控制器厂商提供的设备驱动库中的一个头文件，这个头文件会把 CMSIS-CORE 特性以及外设驱动等所有需要的头文件包含进来。

还可以包含符合 CMSIS 的启动代码，其可能是 C 也可能是汇编代码。CMSIS-CORE 提供了用于不同工具链的多种启动代码模板。

图 3.20 为使用 CMSIS 设备驱动库软件包的简单工程设置，其中一些文件的命名取决于微控制器设备的名称(图 3.20 中显示为< device >)，在使用设备驱动库中的头文件时，其他所需头文件会被自动包含进来(见表 3.5)。

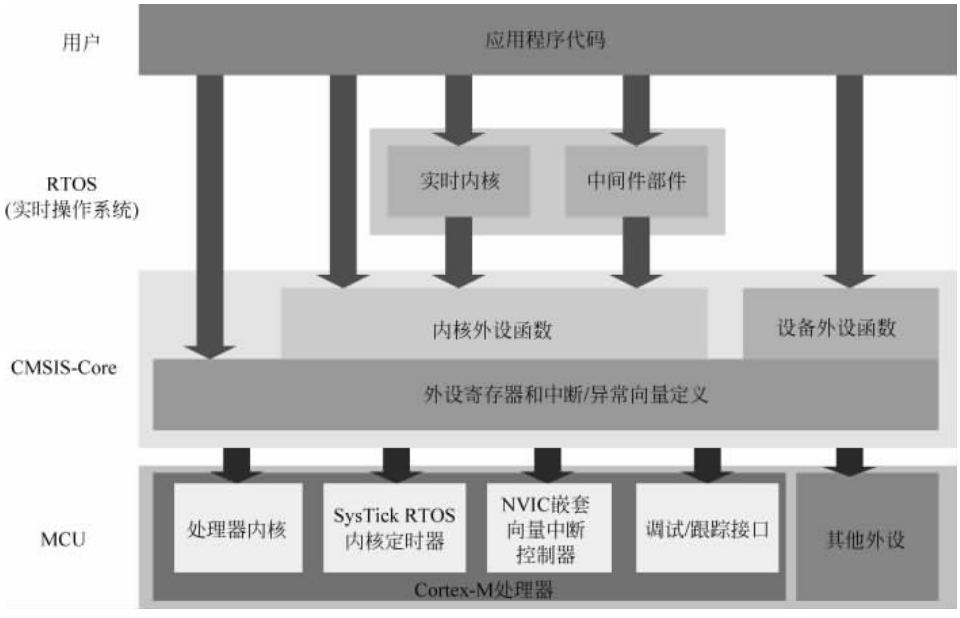

图 3.19　CMSIS 结构

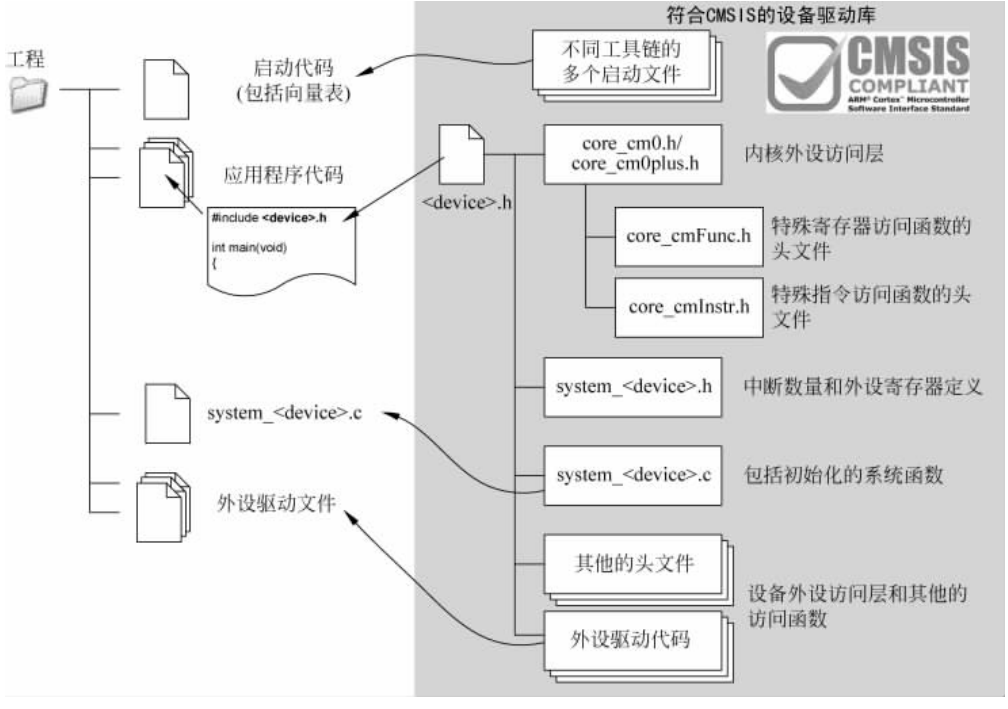

图 3.20　在工程中使用 CMSIS-CORE 设备驱动软件包

表 3.5　CMSIS-CORE 中的文件

文　　件	描　　述
< device >. h	微控制器供应商提供的文件，包括其他头文件以及 CMSIS 需要的多个常量定义、设备特有的异常类型定义、外设寄存器定义以及外设地址定义，实际的文件名同设备相关
core_cm0. h/ core_cm0plus. h	该文件包含处理器外设寄存器的定义，包括 NVIC、SysTick 定时器、系统控制块(SCB)等，还提供中断控制和系统控制等内核操作函数
core_cmFunc. h	提供内核寄存器访问函数
core_cmInstr. h	提供内在函数
启动代码	由于是工具相关的，CMSIS 包含多个版本的启动代码，该代码包含向量表和多个系统异常处理的虚拟定义，从版本 1. 30 开始，在跳转到 C 启动代码之前，复位处理也会执行系统初始化函数“void SystemInit(void)”
system_< device >. h	该文件为 system_< device >. c 中函数的头文件
system_< device >. c	该文件包含系统初始化函数“void SystemInit(void)”、变量“SystemCoreClock(处理器时钟频率)”的定义以及一个名为“void SystemCoreClockUpdate(void)”的函数定义，该函数用于在时钟频率改变后更新“SystemCoreClock”。“SystemCoreClock”和“SystemCoreClockUpdate”从 CMSIS 版本 1. 3 开始使用
其他文件	其他文件包含外设控制代码和其他辅助函数，这些文件提供了 CMSIS 的设备外设访问层

图 3.21 为简单工程中使用符合 CMSIS 驱动的实例。

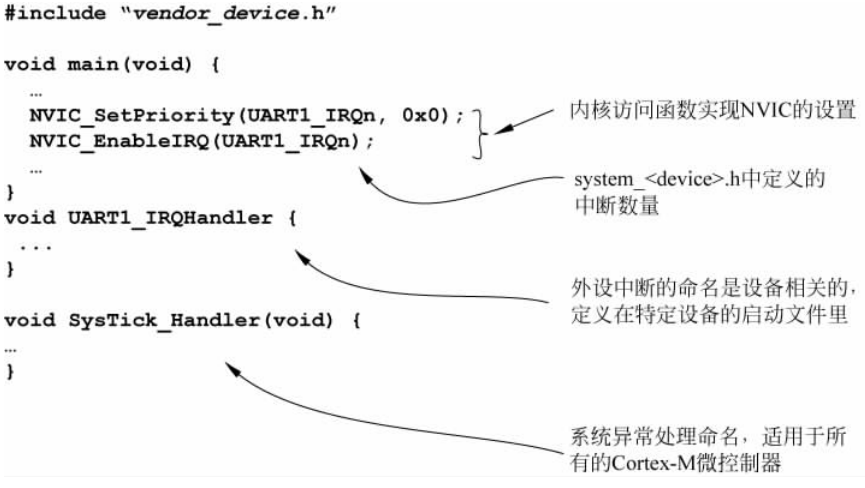

图 3.21　基于 CMSIS-CORE 的应用实例

一般来说，符合 CMSIS 设备驱动库的相关信息和实例位于微控制器厂商提供的代码库包中，ARM 网站(www. arm. com/cmsis)的 CMSIS 包中也有一些使用 CMSIS 的简单实

例，要了解 CMSIS 项目的最新信息，可以访问 http://www.keil.com/CMSIS/。

3.5.5 CMSIS 的优势

对于大多数用户而言，CMSIS 的关键优势在于以下几方面：

1）软件可移植性和可重用性

从基于 Cortex-M 的微控制器移植到另外一个微控制器也更加容易，例如，多数中断控制函数在所有的 Cortex-M 处理器中都是可用的（由于 Cortex-M3/M4 处理器的功能更多，因此 Cortex-M3/M4 的少数几个函数是不能用在 Cortex-M0/M0＋上的，参见 22.5 节），这样就可以很容易地将同一段应用代码用在其他工程中了。可以从 Cortex-M3 工程移植到 Cortex-M0/M0＋以降低功耗，或者从 Cortex-M0/M0＋工程移植到 Cortex-M3 以提高性能。

2）易于学习新设备的编程

学习使用新的 Cortex-M 微控制器也非常容易。由于所有符合 CMSIS 的设备驱动库都具有相同的内核函数和类似的形式，只要使用过一个基于 Cortex-M 的微控制器，就可以很快地开始使用另外一个。

3）软件部件兼容性

在使用第三方软件部件时，CMSIS 还降低了不兼容的风险。由于中间件和嵌入式 RTOS 基于 CMSIS 文件中相同的内核外设寄存器定义以及内核访问函数，这就降低了代码冲突的风险。当多个软件部件都有自己的内核访问函数和寄存器定义时，就可能会产生冲突。如果没有 CMSIS-CORE，不同的第三方软件都有自己的驱动函数，这样就会由于多个函数的命名类似而导致命名的冲突、混乱，且由于重复的函数而带来代码空间的浪费（见图 3.22）。

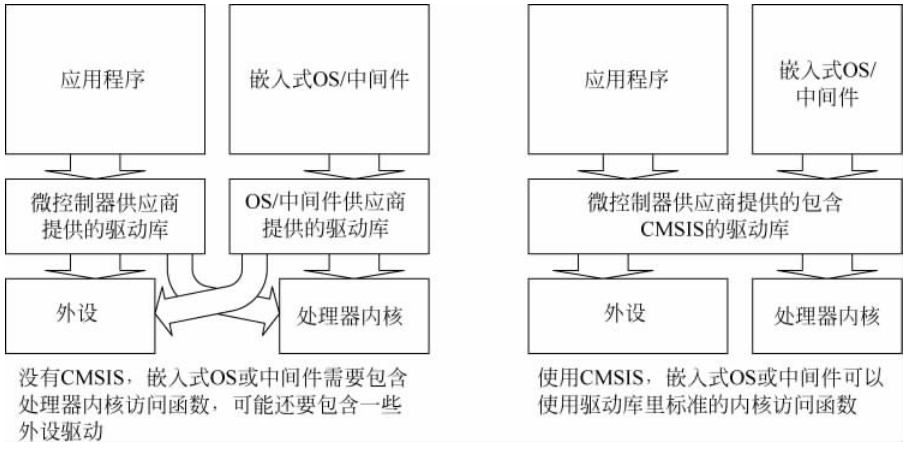

图 3.22 CMSIS 避免驱动代码的重复

4）不会过时

CMSIS 使得软件代码不会过时，未来的 Cortex-M 处理器和基于 Cortex-M 的微控制器

也会支持 CMSIS，因此可以将程序代码重用到未来的产品中。

5）质量

CMSIS 内核访问函数占用的存储器非常少，并且经过多方测试，可减少软件测试时间，CMSIS 符合 MISRA（汽车工业软件可靠性联会）规范。

对于开发嵌入式 OS 或中间件产品的公司，CMSIS 的优势巨大，由于 CMSIS 支持多种编译器组件并得到多家微控制器厂商的支持，因此利用 CMSIS 开发的嵌入式 OS 或中间件可以同多种编译器产品配合工作，且可用于多种微控制器。使用 CMSIS 也意味着这些公司无须开发自己的可移植设备驱动，节省了开发和验证的时间。

3.6　软件开发的其他信息

多数 C 编译器提供在 C 代码中使用汇编代码的方法，例如，ARM 编译器提供嵌入汇编和内联汇编，可以很容易地将汇编函数插入到 C 程序代码中。但是，嵌入汇编和内联汇编的语法是和工具链相关的（不可移植）（注：对于 ARM 编译器，内联汇编对 Thumb 指令的支持始于版本 5.01）。

包括 Development Studio 5（DS-5）和 Keil MDK 中的 ARM C 编译器在内的一些 C 编译器还提供可以插入无法用普通 C 代码生成的特殊指令的内在函数。内在函数一般是和工具相关的，但是利用 CMSIS-CORE 也可以实现 Cortex-M 处理器用的独立于工具的类似函数，21.9 节“访问特殊寄存器”将会介绍这方面的内容。

可以在一个工程中混合使用 C、C++和汇编代码，因此，大部分程序可以用 C/C++编写，而无法用 C 实现的部分则可以使用汇编，此时函数间的接口必须保持一致，以确保输出参数和返回值的正确传递。对于 ARM 软件开发，函数间的接口要符合名为 ARM 架构过程调用标准（AAPCS）的规范文档，AAPCS 属于嵌入式应用二进制接口（EABI）的一部分。在使用嵌入汇编时，应遵循 AAPCS 的规定。AAPCS 和 EABI 文档可从 ARM 网站下载。

要了解这方面的更多细节，可以参考第 21 章的内容。