μC/OS-II 在 STM32F207VG 上的移植及使用手册

2013 年 3 月 25 日

修改记录

制定日期	生效日期	制定 / 修订 内容摘要	页数	版本	拟稿	审查	批准
2013.03.25		初始版本	18	0.01	朱正晶		

目 录

1 简介

本节将简要说明手册的目的、范围。

1.1 手册目的

本手册的目的在于说明 uC/OS-II 在 STM32F207VG 芯片上移植步骤及 uC/OS-II 的初步简介。

1.2 手册范围

本手册首先简要地介绍 STM32F207VG 芯片的功能，然后说明 uC/OS-II 整体的框架及移植详细步骤和原理，最后简要说明 uC/OS-II 的使用方法。

本手册的使用者包括：

程序编写、维护者

…

2 STM32F207VG 芯片及 uC/OS-II 简述

2.1 STM32F207VG 片上资源

内　　核：Cortex-M3 32-bit RISC；
工作频率：120MHz，150 DMIPS；1.25 DMIPS/MHz
工作电压：1.8V-3.6V；
封　　装：LQFP100；
存储资源：1024KB Flash，128+4kB SRAM；
接口资源：3 x SPI，3 x USART，2 x UART,，2 x I2S，3 x I2C；1 x FSMC，1 x SDIO，2 x CAN；1 x USB 2.FS/HS device/host/OTG 片内硬件控制器；1 x 10/100 Ethernet MAC;1 x 8 to 12-bit parallel camera interface;
模数转换：3 x AD（12 位，1us，分时 24 道），2 x DA（12 位）；
调试下载：支持 JTAG/SWD 接口的调试下载，支持 IAP。
应用领域：医疗产品、工业自动化、智能仪表、消费电子、楼宇安防 等等。

2.2 uC/OS-II

uC/OS-II 是一种可移植的，可裁剪的，抢占式的，实时多任务操作系统内核。它被广泛应用于微处理器、微控制器和数字信号处理器（DSP）。

经过适当的裁剪，uC/OS-II 占用 ROM 不超过 5kB，RAM 不超过 2kB。正常使用 ROM 占用 10kB 左右，RAM 占用 5kB 左右。由此可以看出 STM32F207VG 跑 uC/OS-II 非常适合，还有很大空间的

余地，为以后的程序升级提供了保障。

3 移植的必要性

不复杂的嵌入式系统一般设计成图 3-1 所示的样子。这种系统可称为前后台系统或超循环系统(Super-Loops)。

应用程序是一个无限的循环，循环中调用相应的函数完成相应的操作，这部分可以看成后台行为(background)。中断服务程序处理异步事件，这部分可以看成前台行为（foreground）。后台也可以叫做任务级。前台也叫中断级。时间相关性很强的关键操作(Critical operation)一定是靠中断服务来保证的。

使用前后台系统有限制，比如：

首先，因为中断服务提供的信息一直要等到后台程序走到该处理这个信息这一步时才能得到处理，这种系统在处理信息的及时性上，比实际可以做到的要差。这个指标称作任务级响应时间。最坏情况下的任务级响应时间取决于整个循环的执行时间。因为循环的执行时间不是常数，程序经过某一特定部分的准确时间也是不能确定的。进而，如果程序修改了，循环的时序也会受到影响。

第二，任务间进行通信一般采用全局变量，过多的使用全局变量对程序的可读性造成影响，使后续的维护变得异常复杂。

第三，程序规模变大时，各个模块之间的通信会变得错综复杂，最后牵一发而动全身，bug 多也不奇怪了。

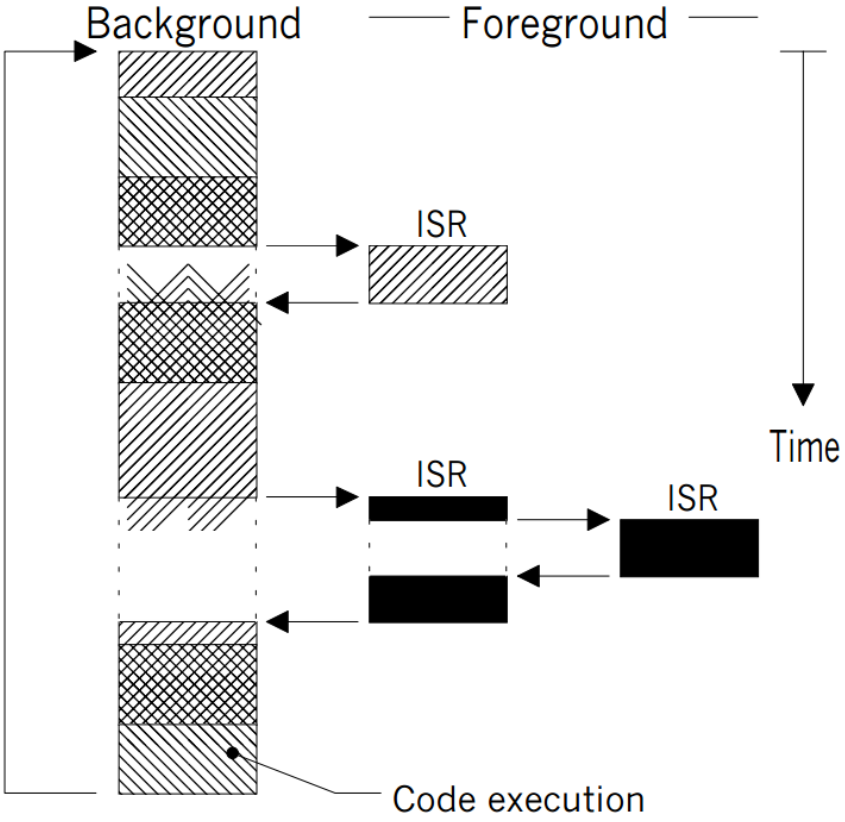

图 3-1 前后台系统

4 uC/OS-II 移植详细步骤

4.1 STM32F2xx standard peripherals library

ST 公司为 STM32 全系列 MCU 提供了非常强大的库代码，学会使用标准库可以让我们少走很多弯路。我们从 ST 官网下载 STM32F207VG 库代码 STM32F2xx standard peripherals library。
下载地址：http://www.st.com/web/en/catalog/tools/PF257898
目前最新的版本为 1.1.0。

4.2 uC/OS-II 实时内核

去 Micrium 官网下载最新的 uC/OS-II 实时内核。目前最新的 uC/OS 版本为 uC/OS-III，我们下载版本 II 的稳定版本（2.91）。
下载地址（需要注册）：
http://micrium.com/downloadcenter/download-results/?searchterm=mp-uc-os-ii&supported=true
除了 kernel，Micrium 还提供了 μC/TCP-IP, μC/FS, μC/GUI，这个等我们需要时再去下载。

4.3 操作下载得到的源代码

解压 STM32F207VGT6 的库，根目录下一共有三个文件夹 Libraries，Project 和 Utilities。

Project 下有非常多的例子，我们可以参照它进行编程，里面包含了 IAR，MDK 等各种开发工具的工程文件。

Utilities 目录下有 STM32 官方评估板的源代码，也可以用来参考。

Libraries 目录下就是真正的库代码了，在 Libraries 目录下有两个文件夹：CMSIS 和 STM32F2xx_StdPeriph_Driver。CMSIS 是 ARM 定义的一套标准，STM32F2xx_StdPeriph_Driver 是 ST 公司对应此款 MCU 的库文件。他们之间的关系如图 4-1：

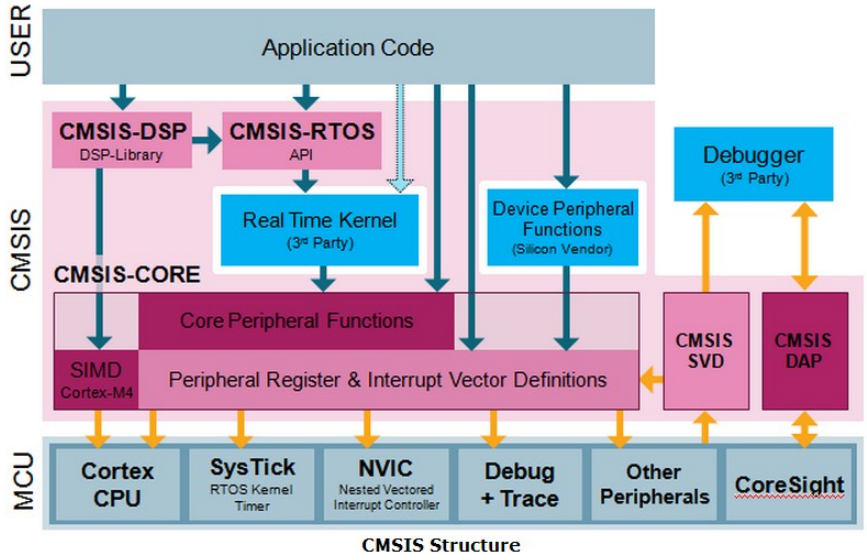

图 4-1 CMSIS 结构框图

4.3.1 CMSIS

ARM 公司于 2008 年 11 月 12 日发布了 ARMCortex 微控制器软件接口标准 CMSIS1.0。CMSIS 是独立于供应商的 Cortex-M 处理器系列硬件抽象层，为芯片厂商和中间件供应商提供了简单的处理器软件接口，简化了软件复用工作，降低了 Cortex-M 上操作系统的移植难度，并减少了新入门的微控制器开发者的学习曲线和新产品的上市时间。根据近期的调查研究，软件开发已经被嵌入式行业公认为最主要的开发成本，图 4-1 为近年来软件开发与硬件开发花费对比图。因此，ARM 与 Atmel、IAR、KEIL、LuminaryMicro、Micrium、NXP、SEGGER 和 ST 等诸多芯片和软件工具厂商合作，将所有 Cortex 芯片厂商的产品的软件接口标准化，制定了 CMSIS 标准。此举意在降低软件开发成本，尤其针对进行新设备项目开发或将已有的软件移植到其他芯片厂商提供的基于 Cortex 处理器的微控制器的情况。有了该标准，芯片厂商就能够将他们的资源专注于对其产品的外设特性进行差异化，并且能够消除对微控制器进行编程时需要维持的不同的、互相不兼容的标准的需求，从而达到降低开发成本的目的。

4.3.2 简要分析在工程中的使用

1. 选择启动文件：根据自己所用的芯片的型号，选择正确的启动文件。我们使用 STM32F207VGT6，就选择 startup_stm32f2xx.s（CMSIS\Device\ST\STM32F2xx\Source\Templates\arm），在这个文件里，首选要定义自己的堆和栈的大小，这个根据自己的需要确定（默认值：栈 0x400，堆 0x200）。文件中已经定义好了中断向量的位置及堆和栈的初始化操作。

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

从上面这段文字中，可以看到，在系统复位后，先执行 SystemInit()，再进入 main()函数。SystemInit()在文件 system_stm32f2xx.c 中定义，下面(3.b)有具体的说明。

2. stm32f2xx.h：这个头文件包含了 STM32F207 的大部分定义：

a.定义芯片的类型，#define STM32F2XX

b.定义是否包含标准库，#define USE_STDPERIPH_DRIVER

c.定义外部振荡器频率，#define HSE_VALUE

上面三个定义，建议在 main.c 文件中刚开始就定义好，或者是在编译器选项中定义好，这样就可以不修改这个文件了

d.定义中断号

e.包含 core_cm3.h，system_stm32f2xx.h（属于 CMSIS）

f.定义数据类型

g.定义外设结构体，地址及用到的数据常量

h.包含 stm32f2xx_conf.h 来配置外设

i.定义位操作的宏

3. system_stm32f2xx.h 和 system_stm32f2xx.c，这两个文件中：

a.定义一个全局变量 uint32_t SystemCoreClock: 系统时钟频率与你选择有关

b.SystemInit():这个函数就是启动文件中调用的函数。主要功能包括进行系统设置，比如初始化片上FLASH，PLL

4. stm32f2xx_conf.h

a.配置需要的标准外设库，需要用到的外设，把相应头文件包含进去就可以。

b.定义 assert_Param 的模式，选择#define USE_FULL_ASSERT 时，断言输出问题所在的位置，在调试时很有用，在正式版本时，把它注释掉即可。

5. stm32f2xx_it.h 和 stm32f2xx_it.c 实现中断处理函数，所以中断必须在这两个文件中编写，而且函数名称必须和 startup_stm32f2xx.s 定义的中断函数名保持一致。

4.3.3 移植

首先来看一下移植 uC/OS-II 所牵涉到的文件，如图 4-2

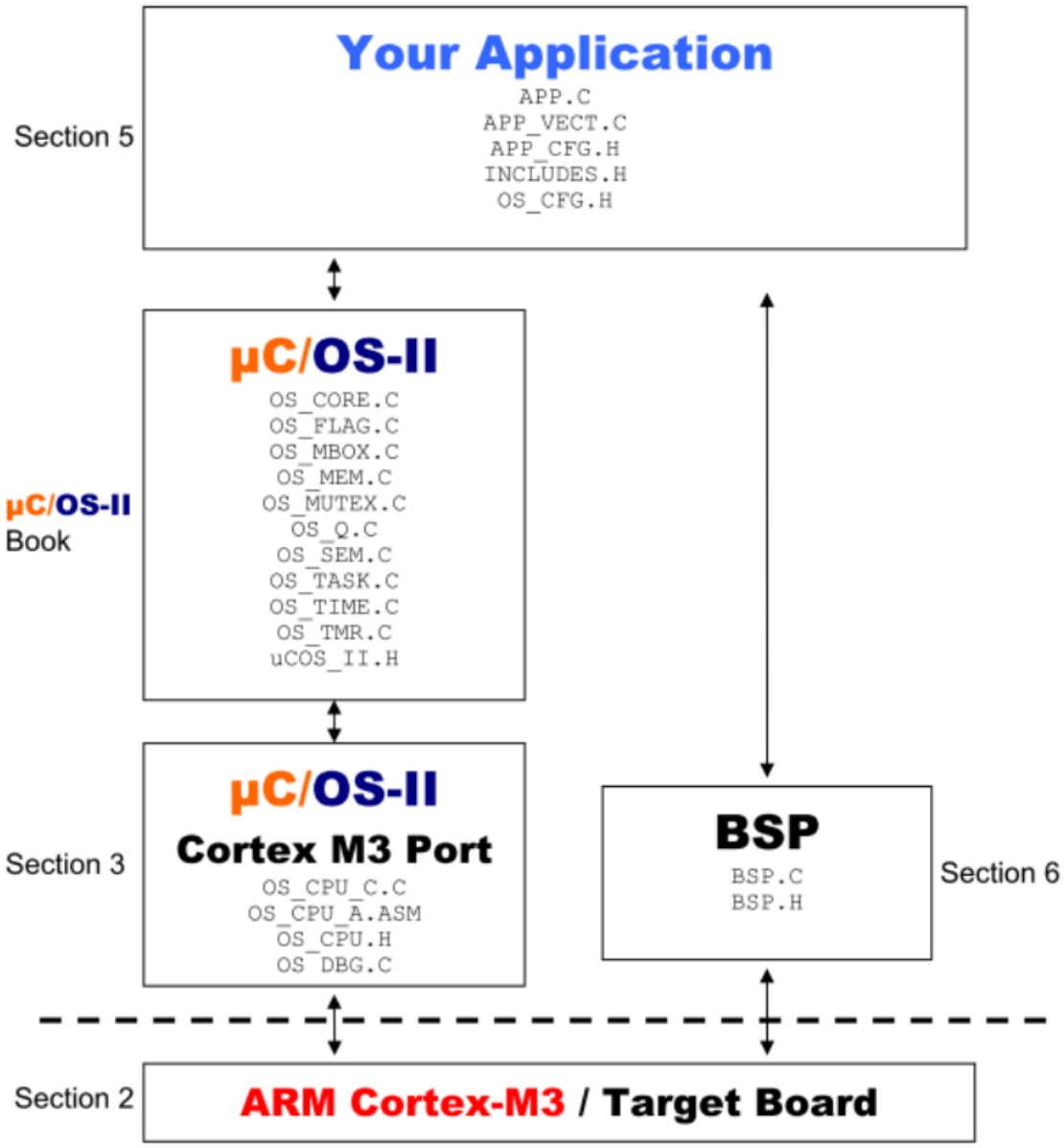

图 4-2

uC/OS-II 已经把 CPU 无关的代码文件放在了 Souce 文件夹下，我们只需要加上四个跟 CPU 相关的文件就可以了。四个文件分别如下：os_cpu_c.c，os_cpu_a.asm，os_cpu.h，os_dgb.c。我们使用 Micrium 已经移植好的 M3 Ports 文件，我们需要读懂这四个文件并加以修改即可在我们的 STM32F207VG 运行。

文件说明如表 4-1

主目录	文件夹	文件名	文件说明
uC/OS-II	Ports	os_cpu.h	定义数据类型、处理器相关代码、声明函数原型
		os_cpu_c.c	定义用户钩子函数，提供扩充软件功能的入口点
		os_cpu_a.asm	与处理器相关汇编函数，主要是任务切换函数
		os_debug.c	内核调试数据和函数
	Source	ucos_ii.h	OS 内部函数参数设置
		os_core.c	内核结构管理，uC/OS 的核心，包含了内核初始化，任务切换，事件块管理、事件标志组管理等功能。
		os_time.c	时间管理，主要是延时
		os_tmr.c	定时器管理，设置定时时间，时间到了就进行一次回调函数处理。
		os_task.c	任务管理
		os_mem.c	内存管理
		os_sem.c	信号量
		os_mutex.c	互斥信号量
		os_mbox.c	消息邮箱
		os_q.c	队列
		os_flag.c	事件标志组

表 4-1 uC/OS-II 主要文件说明

下面我们来详细的了解移植时用的四个文件（Ports 文件夹下）

4.3.3.1 os_cpu.h

定义数据类型、处理器相关代码、声明函数原型。

全局变量

OS_CPU_GLOBALS 和 OS_CPU_EXT 允许我们是否使用全局变量。

```
1.  #ifdef  OS_CPU_GLOBALS
2.  #define OS_CPU_EXT
3.  #else                          //如果没有定义 OS_CPU_GLOBALS
4.  #define OS_CPU_EXT  extern     //则用 OS_CPU_EXT 声明变量已经外部定义了。
5.  #endif
```

数据类型

```
6.  typedef unsigned char  BOOLEAN;
7.  typedef unsigned char  INT8U;
8.  typedef signed   char  INT8S;
9.  typedef unsigned short INT16U;
10.  typedef signed   short INT16S;
11.  typedef unsigned int   INT32U;
12.  typedef signed   int   INT32S;
13.  typedef float         FP32;    //尽管包含了浮点数，但 uC/OS-II 中并没用到
14.  typedef double        FP64;
16.  typedef unsigned int   OS_STK;   //M3 是 32 位，所以堆栈的数据类型 OS_STK 设置 32 位
17.  typedef unsigned int   OS_CPU_SR;  //M3 的状态寄存器（xPSR）是 32 位临界段
```

临界段

临界段，就是不可被中断的代码段，例如常见的入栈出栈等操作就不可被中断。

uC/OS-II 是一个实时内核，需要关闭中断进入和开中断退出临界段。为此，uC/OS-II 定义了两个宏定义来关中断 OS_ENTER_CRITICAL()和开中断 OS_EXIT_CRITICAL()。

```
18.  #define  OS_CRITICAL_METHOD    3  //进入临界段的三种模式，一般选择第 3 种，即这里设置为 3
21.  #define  OS_ENTER_CRITICAL()  {cpu_sr = OS_CPU_SR_Save();} //进入临界段
22.  #define  OS_EXIT_CRITICAL()   {OS_CPU_SR_Restore(cpu_sr);} //退出临界段
```

事实上，有 3 种开关中断的方法，根据不同的处理器选用不同的方法。uC/OS-II 选用第 3 种方法。

OS_CPU_SR_Save() 和 OS_CPU_SR_Restore()为汇编函数，在 os_cpu_a.asm 中定义。

栈生长方向

M3 的栈生长方向是由高地址向低地址增长的，因此 OS_STK_GROWTH 定义为 1。

```
23.  #define  OS_STK_GROWTH        1
```

任务切换宏

定义任务切换宏，关于汇编函数 OSCtxSw()，在 os_cpu_a.asm 文件中定义。

```
24.  #define  OS_TASK_SW()         OSCtxSw()
```

开中断和关中断

如果定义了进入临界段的模式为 3，就声明开中断和关中断函数

```
25.  #if OS_CRITICAL_METHOD == 3
26.  OS_CPU_SR  OS_CPU_SR_Save(void);
27.  void       OS_CPU_SR_Restore(OS_CPU_SR cpu_sr);
28.  #endif
```

任务管理函数

```
/******************任务切换的函数******************/
30.  void OSCtxSw(void);          //用户任务切换
31.  void OSIntCtxSw(void);       //中断任务切换函数
32.  void OSStartHighRdy(void);  //在操作系统第一次启动的时候调用的任务切换
33.  void PendSV_Handler(void);
```

关于任务切换，利用到异常处理知识，可以参考《Cortex-M3 权威指南》（Joseph Yiu 著 宋岩译）中第 3.4 小节。

关于 PendSV，可以看《Cortex-M3 权威指南》中第 7.6 小节 SVC 和 PendSV：SVC（系统服务调用，亦简称系统调用）和 PendSV（可悬起系统调用），它们多用在上了操作系统的软件开发中。

SVC 用于产生系统函数的调用请求，SVC 异常是必须在执行 SVC 指令后立即得到响应的。PendSV（可悬起的系统调用）则不同，它是可以像普通的中断一样被悬起的（不像 SVC 那样会上访）。OS 可以利用它“缓期执行”一个异常——直到其它重要的任务完成后才执行动作。悬起 PendSV 的方法是：手工往 NVIC 的 PendSV 悬起寄存器中写 1。悬起后，如果优先级不够高，则将缓期等待执行。PendSV 的典型使用场合是在上下文切换时（在不同任务之间切换）。例如，一个系统中有两个就绪的任务，上下文切换被触发的场合可以是：

1) 执行一个系统调用

2) 系统滴答定时器（SysTick）中断。（轮转调度中需要）

举个简单的例子来辅助理解。假设有这么一个系统，里面有两个就绪的任务，并且通过 SysTick 异常启动上下文切换。

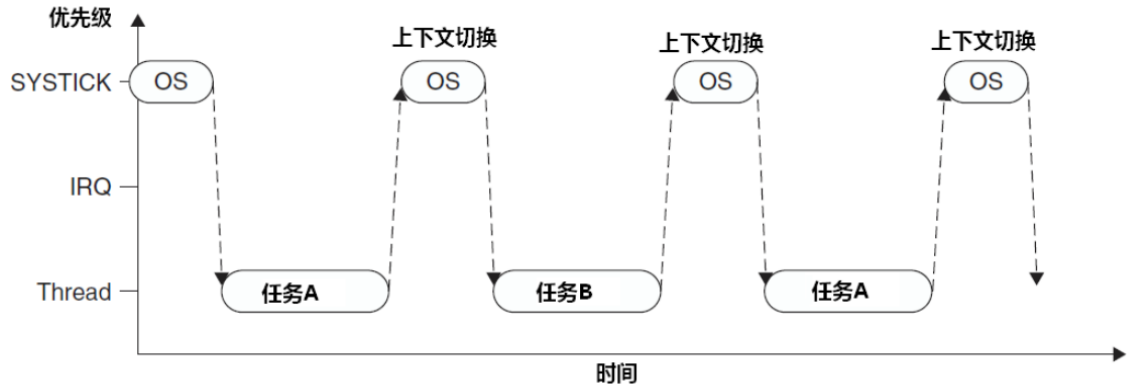

图 4-3 两个任务间通过 SysTick 进行轮转调度的简单模式

图 4-3 是两个任务轮转调度的示意图。但若在产生 SysTick 异常时正在响应一个中断，则 SysTick 异常会抢占其 ISR。在这种情况下，OS 是不能执行上下文切换的，否则将使中断请求被延迟，而且在真实系统中延迟时间还往往不可预知——任何有一丁点实时要求的系统都决不能容忍这种事。因此，在 CM3 中也是严禁没商量——如果 OS 在某中断活跃时尝试切入线程模式，将触犯用法 fault 异常。

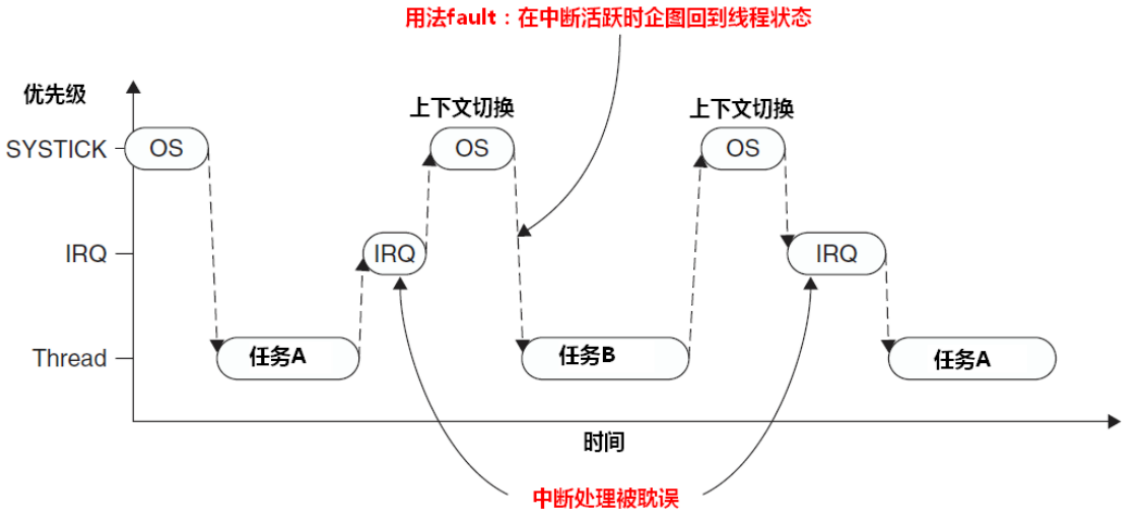

图 4-4 发生 IRQ 时上下文切换的问题

为解决此问题，早期的 OS 大多会检测当前是否有中断在活跃中，只有在无任何中断需要响应时，才执行上下文切换（切换期间无法响应中断）。然而，这种方法的弊端在于，它可以把任务切换动作拖延很久（因为如果抢占了 IRQ，则本次 SysTick 在执行后不得作上下文切换，只能等待下一次 SysTick 异常），尤其是当某中断源的频率和 SysTick 异常的频率比较接近时，会发生“共振”，使上下文切换迟迟不能进行。

现在好了，PendSV 来完美解决这个问题了。PendSV 异常会自动延迟上下文切换的请求，直到其它的 ISR 都完成了处理后才放行。为实现这个机制，需要把 PendSV 编程为最低优先级的异常。如果 OS 检测到某 IRQ 正在活动并且被 SysTick 抢占，它将悬起一个 PendSV 异常，以便缓期执行上下文切换。

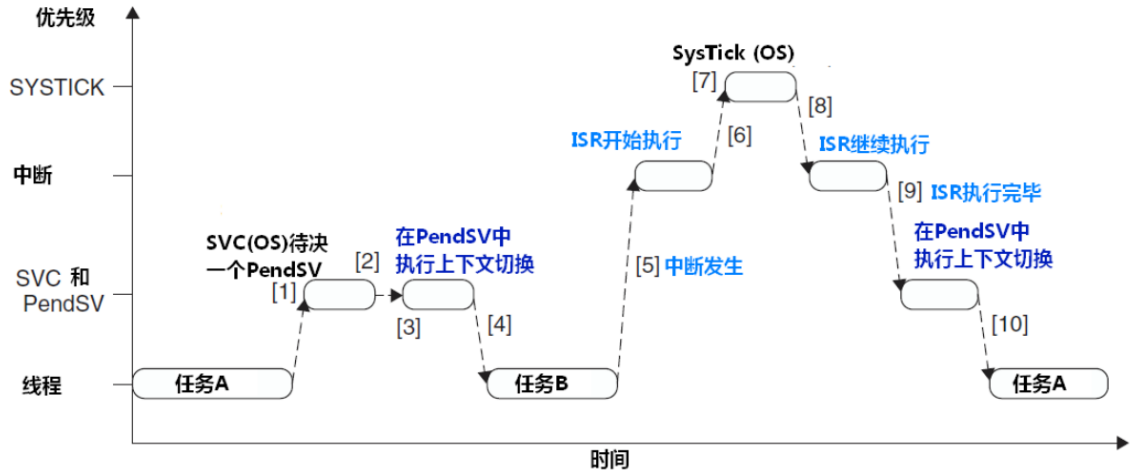

图 4-5 使用 PendSV 控制上下文切换

4.3.3.2 os_cpu_c.c

移植 uC/OS 时，我们需要写 10 个相当简单的 C 函数：9 个钩子函数和 1 个任务堆栈结构初始化函数。

钩子函数

所谓钩子函数，指那些插入到某些函数中为扩展这些函数功能的函数。一般地，钩子函数为第

三方软件开发人员提供扩充软件功能的入口点。为了拓展系统功能，uC/OS-II 中提供有大量的钩子函数，用户不需要修改 uC/OS-II 内核代码程序，而只需要向钩子函数添加代码就可以扩充 uC/OS-II 的功能。在这里我们没有打开钩子函数的功能，在 os_cfg.h 中将 OS_APP_HOOKS_EN 设为 0 即可。

任务堆栈结构初始化函数

OSTaskStkInit() //任务堆栈结构初始化函数

```
OS_STK *OSTaskStkInit (void (*task)(void *pd), void *p_arg,OS_STK *ptos,
INT16U opt)
100.  {
101.      OS_STK *stk;
102.
103.
104.      (void)opt;                    // 'opt' 并没有用到，防止编译器提示警告
105.      stk       = ptos;             // 加载栈指针
106.
107.      /* 中断后 xPSR，PC，LR，R12，R3-R0 被自动保存到栈中*/
108.      *(stk)    = (INT32U)0x01000000L; // xPSR
109.      *(--stk)  = (INT32U)task;       // 任务入口 (PC)
110.      *(--stk)  = (INT32U)0xFFFFFFFEL; // R14 (LR)
111.      *(--stk)  = (INT32U)0x12121212L; // R12
112.      *(--stk)  = (INT32U)0x03030303L; // R3
113.      *(--stk)  = (INT32U)0x02020202L; // R2
114.      *(--stk)  = (INT32U)0x01010101L; // R1
115.      *(--stk)  = (INT32U)p_arg;      // R0 : 变量
116.
117.      /* 剩下的寄存器需要手动保存在堆栈    */
118.      *(--stk)  = (INT32U)0x11111111L; // R11
119.      *(--stk)  = (INT32U)0x10101010L; // R10
120.      *(--stk)  = (INT32U)0x09090909L; // R9
121.      *(--stk)  = (INT32U)0x08080808L; // R8
122.      *(--stk)  = (INT32U)0x07070707L; // R7
123.      *(--stk)  = (INT32U)0x06060606L; // R6
124.      *(--stk)  = (INT32U)0x05050505L; // R5
125.      *(--stk)  = (INT32U)0x04040404L; // R4
126.
127.      return (stk);
128.  }
```

4.3.3.3 os_cpu_a.asm

这个文件包含了需要用汇编编写的代码。

申明外部变量

```
EXTERN  OSRunning       ; External references
EXTERN  OSPrioCur
```

```
EXTERN  OSPrioHighRdy
EXTERN  OSTCBCur
EXTERN  OSTCBHighRdy
EXTERN  OSIntExit
EXTERN  OSTaskSwHook
EXTERN  OS_CPU_ExceptStkBase
```

声明全局变量

```
EXPORT  OS_CPU_SR_Save   ; Functions declared in this file
EXPORT  OS_CPU_SR_Restore
EXPORT  OSStartHighRdy
EXPORT  OSCtxSw
EXPORT  OSIntCtxSw
EXPORT  PendSV_Handler
```

向量中断控制器 NVIC

```
200.  NVIC_INT_CTRL EQU 0xE000ED04  ;中断控制及状态寄存器 ICSR 的地址
201.
202.  NVIC_SYSPRI14 EQU 0xE000ED22  ;系统异常优先级寄存器 PRI_14
203.                                ;即设置 PendSV 的优先级
204.
205.  NVIC_PENDSV_PRI EQU 0xFF  ;定义 PendSV 的可编程优先级为 255，即最低
206.
207.  NVIC_PENDSVSET EQU 0x10000000 ;中断控制及状态寄存器 ICSR 的位 28
208.                ;写 1 以悬起 PendSV 中断。读取它则返回 PendSV 的状态
```

中断

与中断方式 3 相关的有两个汇编函数：

```
209.  ;  OS_ENTER_CRITICAL() 里进入临界段调用，保存现场环境
210.  OS_CPU_SR_Save
211.  MRS     R0, PRIMASK     ; 读取 PRIMASK 到 R0(保存全局中断标记,除了故障中断)
212.  CPSID   I               ; PRIMASK=1，关中断
213.  BX      LR              ; 返回，返回值保存在 R0
214.
215.
216.  ;  OS_EXIT_CRITICAL()  里退出临界段调用，恢复现场环境
217.  OS_CPU_SR_Restore
218.  MSR PRIMASK,R0 ;读取 R0 到 PRIMASK 中(恢复全局中断标记)，通过 R0 传递参数
219.  BX      LR
```

功能：关全局中断前，保存全局中断标志，进入临界段。退出临界段后恢复中断标记。

启动最高优先级任务

OSStartHighRdy()启动最高优先级任务，由 OSStart()里调用，调用前必须先调用OSTaskCreate 创建至少一个用户任务，否则系统会发生崩毁。

```
220.  OSStartHighRdy
221.  LDR  R0, =NVIC_SYSPRI14      ; 装载 系统异常优先级寄存器 PRI_14
222.           ; 即设置 PendSV 中断优先级的寄存器
223.  LDR  R1, =NVIC_PENDSV_PRI    ; 装载 PendSV 的可编程优先级(255)
```

```
224.  STRB R1, [R0]                    ; 无符号字节寄存器存储。R1 是要存储的寄存器
225.                                   ; 存储到内存地址所基于的寄存器
226.                                   ; 即设置 PendSV 中断优先级为 255
227.
228.  MOV  R0, #0           ; 把数值 0 复制到 R0 寄存器
229.  MSR  PSP, R0          ; 将 R0 的内容加载到程序状态寄存器 PSR 的指定字段中
230.
231.  LDR  R0, __OS_Running    ; OSRunning = TRUE
232.  MOV  R1, #1
233.  STRB R1, [R0]
234.
235.  LDR     R0, =NVIC_INT_CTRL      ; 装载 中断控制及状态寄存器 ICSR 的地址
236.  LDR     R1, =NVIC_PENDSVSET     ; 中断控制及状态寄存器 ICSR 的位 28
237.  STR     R1, [R0]                ; 设置 中断控制及状态寄存器 ICSR 位 28 为 1
238.                                  ; 以悬起(允许)PendSV 中断
239.
240.      CPSIE   I                   ;打开中断
```

任务切换

当任务放弃 CPU 的使用权时，就会调用 OS_TASK_SW()

一般情况下，OS_TASK_SW()是做任务切换。但在 M3 中，任务切换的工作都被放到 PendSV 的中断处理服务中去做以加快处理速度，因此 OS_TASK_SW()只需简单的悬起(允许)PendSV 中断即可。当然，这样就只有当再次开中断的时候，PendSV 中断处理函数才能执行。

OS_TASK_SW()是由 OS_Sched() （此函数在 OS_CORE.C）调用。

```
241.  /*******************任务级调度器******************/
242.  void  OS_Sched (void)
243.  {
244.  #if OS_CRITICAL_METHOD == 3
245.      OS_CPU_SR  cpu_sr = 0;
246.  #endif
247.
248.      OS_ENTER_CRITICAL();
249.      if (OSIntNesting == 0) {          //如果没中断服务运行
250.          if (OSLockNesting == 0) {     //调度器没上锁
251.              OS_SchedNew();          //查找最高优先级就绪任务
252.                                      //见 os_core.c ，会修改 OSPrioHighRdy
253.              if (OSPrioHighRdy != OSPrioCur) {  //如果得到的最高优先级就绪
254.                                  //任务不等于当前,注：当前运行的任务也在就绪表里
255.                  OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy]; //得到任务
控制块指针
256.  #if OS_TASK_PROFILE_EN > 0
257.                  OSTCBHighRdy->OSTCBCtxSwCtr++;  //统计任务切换到次任务的
计数器加 1
258.  #endif
259.                  OSCtxSwCtr++;                   //统计任务切换次数的计数器加
```

```
1
260.                OS_TASK_SW();                    //进行任务切换
261.             }
262.          }
263.       }
264.       OS_EXIT_CRITICAL();                        //退出临界段，开中断
265.  }
```

OS_TASK_SW()就是用宏定义包装的 OSCtxSw()(见 OS_CPU.H)：

```
266.  #define  OS_TASK_SW()         OSCtxSw()
267.  OSCtxSw
268.  ; 悬起(允许)PendSV 中断
269.  LDR  R0, =NVIC_INT_CTRL        ; 装载 中断控制及状态寄存器 ICSR 的地址
270.  LDR  R1, =NVIC_PENDSVSET       ; 中断控制及状态寄存器 ICSR 的位 28
271.  STR R1, [R0]                  ;设置 中断控制及状态寄存器 ICSR 位 28 为 1
272.                                 ;以悬起(允许)PendSV 中断
273.  BX   LR                       ; 返回
```

中断退出处理

当中断处理函数退出时，就会调用 OSIntExit()来决定是否有优先级更高的任务需要执行。如果有， OSIntExit()会调用 OSIntCtxSw() 做任务切换。

在 M3 里，与 OSCtxSw 一样，任务切换时，OSIntCtxSw 都只需简单的悬起(允许)PendSV 中断即可，真正的任务切换工作放在 PendSV 中断服务程序里，等待开中断时才正在执行任务切换。

在这里，OSCtxSw 的代码是与 OSCtxSw 完全相同的：

```
274.  OSIntCtxSw
275.        LDR     R0, =NVIC_INT_CTRL      ; trigger the PendSV exception
276.        LDR     R1, =NVIC_PENDSVSET
277.        STR     R1, [R0]
278.        BX      LR
```

PendSV 中断服务

```
297.  OS_CPU_PendSVHandler   ;CPU 会自动保存 xPSR, PC, LR, R12, R0-R3
298.    CPSID   I           ;关中断
299.    MRS     R0, PSP     ;PSP 就是栈指针，R0=PSP
300.    CBZ     R0, OSPendSV_nosave ;当 PSP==0，执行 OSPendSV_nosave 函数
301.
302.    SUB     R0, R0, #0x20;装载 r4-11 到栈 ，共 8 个寄存器，32 位，4 个字节
303.                          ;即 8*4=32=0x20
304.    STM     R0, {R4-R11}          ;
305.
306.    LDR     R1, __OS_TCBCur        ;R1=&OSTCBCur
307.    LDR     R1, [R1]              ;R1=*R1 (R1=OSTCBCur)
308.    STR     R0, [R1]              ;*R1=R0 (*OSTCBCur=SP)
309.
310.  OSPendSV_nosave
311.    PUSH    {R14}                  ;保存 R14
```

```
312.    LDR     R0, __OS_TaskSwHook     ;调用钩子函数 OSTaskSwHook()
313.    BLX     R0
314.    POP     {R14}                   ;恢复 R14
315.
316.    LDR     R0, __OS_PrioCur    ;设置当前优先级为最高优先级就绪任务的优先级
317.
;OSPrioCur = OSPrioHighRdy
318.    LDR     R1, __OS_PrioHighRdy
319.    LDRB    R2, [R1]
320.    STRB    R2, [R0]
321.
322.    LDR     R0, __OS_TCBCur         ;设置当前任务控制块指针
323.    LDR     R1, __OS_TCBHighRdy      ;OSTCBCur  = OSTCBHighRdy
324.    LDR     R2, [R1]
325.    STR     R2, [R0]
326.
327.    LDR     R0, [R2]                ;R0 是新的 SP
328.                                    ;SP = OSTCBHighRdy->OSTCBStkPtr;
329.
330.    LDM     R0, {R4-R11}            ;从新的栈恢复 R4-R11
331.    ADD     R0, R0, #0x20
332.    MSR     PSP, R0                 ;PSP=R0,用新的栈 SP 加载 PSP
333.    ORR     LR, LR, #0x04           ;确保 LR 位 2 为 1，返回到使用进程堆栈
334.    CPSIE   I                       ;开中断
335.    BX      LR                      ;返回
```

当第一次开始任务切换时时，而任务刚创建时 R4-R11 已经保存在堆栈中，此时不用再保存，就会跳到 OS_CPU_PendSVHandler_nosave 执行。

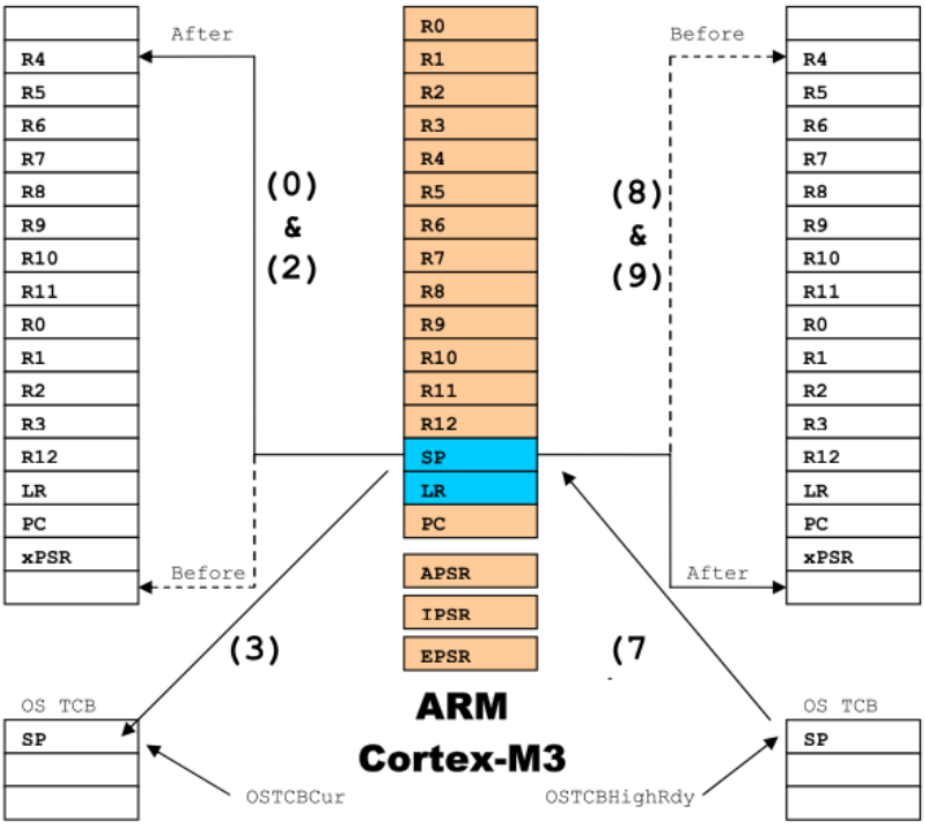

图 4-6 ARM Cortex-M3 Context Switch