在 STM32 上写 RTOS

前言

随着找工作告一段落，如今总算有了一点点空闲时间，趁着在还未开始写大论文，记录一下我学习 RTOS 的过程。这篇文章 RTOS 的入门教程，实现的 OS 功能实在是非常简单和单一，但我希望即使是简单的内容也能见微知著，让大家不仅仅知其然还知其所以然。因此本文也讲述了不少底层的知识，介绍 Cortex-M 的架构。

本文的适用对象是对 STM32、u/COS 有基本了解和使用经历的人，同时需要对汇编有简单的基础。本文是基于 Cortex-M3 的处理器，使用的 STM32F1 系列的芯片，Cortex-M4 的处理器亦可兼容，但显然本文没有考虑 Cortex-M4 的独有特性。

本文用到的参考资料并不多，详见参考文献篇目。依据本文主题，只提取了相关部分，其中第 1、2 章内容主要来源于宋岩（译）《Cortex-M3 权威指南》，这本书实在是太棒了，以至于我几乎未对原文作改动，只进行了节选和少部分细节修改，第 3 章部分内容来源于邵贝贝（译）《嵌入式实时操作系统 uCOS-II》。如果希望能更深层次的理解 RTOS 在 STM32 上的运行，笔者认为有必要对参考数目细细拜读，受篇幅限制，仅靠本文几十页的内容着实是不可能面面俱到，许多方面只能是浅尝辄止，因此最重要的是亲自实践，自己去探索其中的奥妙。也感谢网络上各路大神发布的贴子，未一一列出。

由于时间仓促，及作者本人水平有限，难免有疏漏和不足之处，请多指教！反馈地址为 inhowe@qq.com

INHOWE　2019.10

目录

第一章 Cortex-M 基础

1.1 Cortex-M 简介

Cortex-M（后面简称为 CM）是经典 ARM7 处理器的后继者。与通用处理器不同，CM 面向的是高性价比等中低端应用场合，但是应用场合低端却不代表其架构不先进，设计师们在 ARM7 的基础上大刀阔斧的改革设计架构，突破性的引入了许多崭新的技术和特性，不仅提高了运行能效，也显著的简化的编程难度和移植难度。从架构的角度考虑，CM 内核依然是非常先进的处理器！

CM 处理器是一个 32 位处理器，内部数据路径是 32 位的，寄存器是 32 位的，存储器接口也是 32 位的。CM 采用了哈佛结构，拥有独立的指令总线和数据总线，取值和数据访问可以并行不悖，从而提高了性能。但是，这两个总线共享同样的存储空间（同一块 Flash），所以寻址空间依然是 4GB。

如今 ARM 市场上有 3 种常见的指令系统，一个是具有完善功能但体积略大的 32 位 ARM 指令集，一个是具有更高代码密度但功能有限的精简指令集即 16 位 Thumb 指令集，还有一个是综合两个优点的 Thumb2 指令集，既可以追求高的代码密度也可以追求高的执行效率。CM 根据自身定位果断的放弃了 ARM 指令集，使用了先进的 Thumb2 指令集。图 1.1 是 CM 在两个指令集之间的关系。

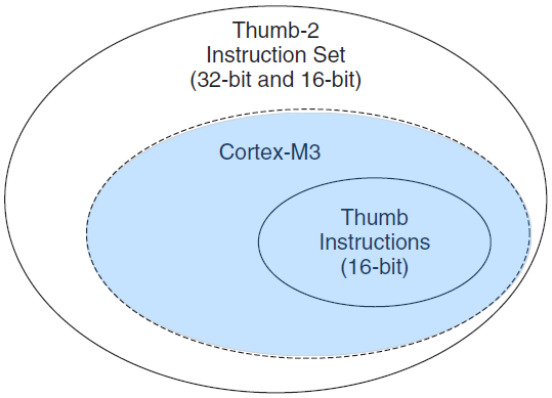


图 1.1 Thumb2 指令集与 Thumb 指令集的关系

1.2 Cortex-M 通用寄存器组

CM（Cortex-M）处理器拥有 16 个通用寄存器，复位后寄存器的值是不可预料的。

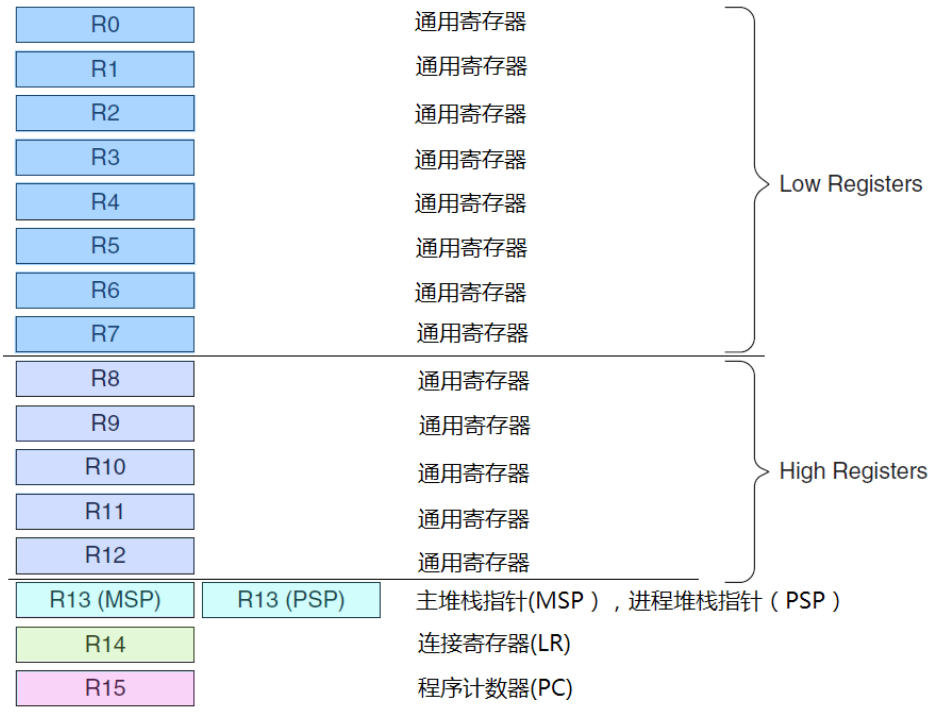


图 1.2 CM 通用寄存器组

R0-R12 通用寄存器

这 13 个寄存器被分为 2 组。低组寄存器 R0-R7 共 8 个可以被所有汇编指令访问。高组寄存器 R8-R12 共 5 个只能被 32 位指令系统访问以及少数的 16 位系统访问。通用寄存器主要用于 CPU 计算过程中的数据交换和记录。

R13（SP 寄存器）

R13 作为堆栈指针，CM 具有 2 个 SP 指针，分别是 MSP 主栈指针，MSP 进程栈指针。同一时间只有一个栈指针在工作，具体是哪个取决于当前工作模式，SP 指针存储当前指针的值。

- 主栈指针（MSP）：复位后缺省指针，也是中断异常处理时用的指针。该指针通常留给操作系统内核使用。
- 进程栈指针（PSP）：该指针留给用户代码使用

由于操作系统和用户代码使用了不同的栈指针，因此当用户代码程序崩溃时，不会破坏系统内核的栈数据，这就意味着内核可以正常的响应，操作系统就是稳定的，此时内核可以帮忙给用户代码擦屁股，比如结束进程并回收内存，而不至于彻底死机只能重启。不过 uC/OS 比较简单，没用到这个特性。

R14（LR 链接寄存器）

当呼叫一个子程序时，由 R14 存储返回地址，这就是链接的含义。返回地址被存储在寄存器里可以减少访问内存（访内）次数，可以更快的找到返回地址，从而提高运行效率。不过，也可以看出，这只对 1 级调用效果好，当发生函数嵌套时，LR 寄存器的数据会被刷新，因此前一级的 LR 寄存器需要压入堆栈保存。

R15（PC 计数寄存器）

PC 寄存器保存的是下一条指令的地址，改变其值就可以改变程序执行流。

1.3 Cortex-M 特殊功能寄存器

CM 还搭载了一些若干特殊功能寄存器，包括程序状态字寄存器（xPSR）、中断屏蔽寄存器组（PRIMASK、FAULTMASK、BASEPRI）、控制寄存器（CONTROL）。他们只能被专用 MRS 和 MSR 指令访问。

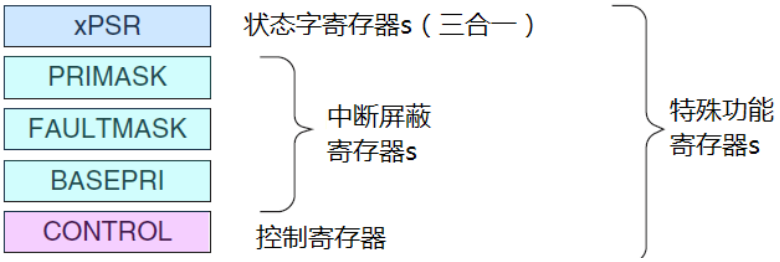


图 1.3 特殊功能寄存器集合

状态字寄存器

状态字寄存器是三合一寄存器，包括应用程序 PSR、中断号 PSR、执行 PSR。

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q											
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

图 1.4 三合一程序状态寄存器

简单介绍一下 APSR 中的算数与逻辑标志，他们是：

- N：负数标志（Negtive）
- Z：零结果标志（Zero）
- C：进位/借位标志（Carry）
- V：溢出标志（Overflow）
- S：饱和标志（Saturation），它不做条件转移的依据

中断屏蔽寄存器

中断屏蔽寄存器的 PRIMASK 是 1 位寄存器，写 1 时关闭所有可屏蔽异常，仅 NMI 和硬 fault 可以响应。该寄存器通常用于开关总中断。

中断屏蔽寄存器的 FAULTMASK 是 1 位寄存器，写 1 时，仅 NMI 可响应，也就是比 PRIMASK 更加严格。

中断屏蔽寄存器的 BASEPRI 最多为 9 位（由表达优先级的位数决定）。它定义了被屏蔽优先级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。

对于时间 - 关键任务而言，PRIMASK 和 BASEPRI 对于暂时关闭中断是非常重要的。而 FAULTMASK 则可以被 OS 用于暂时关闭 fault 处理机能，这种处理在某个任务崩溃时

可能需要。因为在任务崩溃时，常常伴随着一大堆 faults。在系统料理“后事”时，通常不再需要响应这些 fault——人死帐清。总之 FAULTMASK 就是专门留给 OS 用的。

其实，为了快速地开关中断，CM 还专门设置了一条 CPS 指令，有 4 种用法

```
CPSID I    ;PRIMASK=1     ;关中断
CPSIE I    ;PRIMASK=0     ;开中断
CPSID F    ;FAULTMASK=1   ;关异常
CPSIE F    ;FAULTMASK=0   ;开异常
```

控制寄存器

控制寄存器用于定义特权级别，还用于选择当前使用哪个堆栈指针。

表 1.1 控制寄存器位定义

位	功能
CONTROL[1]	堆栈指针选择 0=选择主堆栈指针 MSP（复位后缺省值） 1=选择进程堆栈指针 PSP 在线程或基础级(没有在响应异常——译注),可以使用 PSP。在 handler 模式下， 只允许使用 MSP，所以此时不得往该位写 1。
CONTROL[0]	0=特权级的线程模式 1=用户级的线程模式 Handler 模式永远都是特权级的。

（简注：特权级和用户级类似于 linux 下的 root 用户和普通用户；线程模式和 handler 模式即正常运行和进入异常响应模式，可参考操作模式和特权级别小节详细了解）

1.4 Cortex-M 操作模式和特权级别

CM 处理器支持两种操作模式和两级特权级别。

操作模式分别为：处理者模式（handler mode）和线程模式（thread mode）。引入两个模式的本意，是是用于区别普通应用程序的代码和异常服务例程的代码——包括中断服务例程的代码。

两级特权级别分别为：特权级和用户级。这可以提供一种存储器访问的保护机制，使得普通的用户程序代码不能意外地，甚至是恶意地执行涉及到要害的操作。处理器支持两种特权级，这也是一个基本的安全模型。

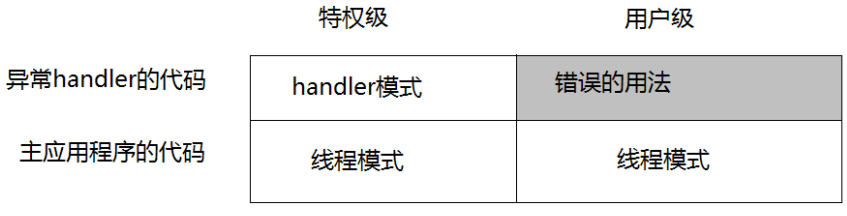


图 1.5 CM 的操作模式和特权级别

CM的主应用程序既可以使用特权级，也可以使用用户级，但异常服务代码只能在特权级下执行。复位后，处理器默认是线程模式，特权级访问。特权级下的代码具有至高无上的权限，可以为所欲为，因此必须谨慎的对待，也通常是设计给操作系统内核所使用的，否则一旦翻车就是车毁人亡，只能重启。用户级的代码则有诸多访问限制，如特殊功能寄

存器是无法访问的，这给代码的稳定运行提供了保障。

不过特权级下的代码也可能会把自己玩进去——切换到用户级。如同linux下想重新回到root权限你得输入密码一样，从用户级切换回特权级也不是随便修改一下CONTROL寄存器的值就可以实现的，需要先申诉，即执行（SVC）指令，请求一个SVC异常，然后由异常服务例程（这部分通常是操作系统的一部分）接管，如果操作系统批准了进入，则在异常服务例程里修改CONTROL寄存器才能重回特权级。

事实上，从用户级到特权级的唯一途径就是异常：如果在程序执行过程中触发了一个异常，处理器总是先切换入特权级，并且在异常服务例程执行完毕退出时，返回先前的状态。

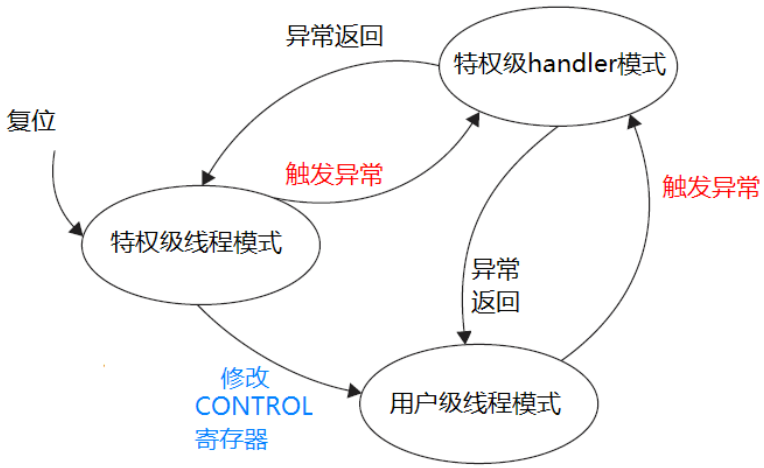


图 1.6 合法的操作模式转换图

通过引入特权级和用户级，就能够在硬件水平上限制某些不受信任的或者还没有调试好的程序，不让它们随便地配置涉及要害的寄存器，因而系统的可靠性得到了提高。进一步地，如果配了MPU，它还可以作为特权机制的补充——保护关键的存储区域不被破坏，这些区域通常是操作系统的区域。

举例来说，操作系统的内核通常都在特权级下执行，所有没有被MPU 禁掉的存储器都可以访问。在操作系统开启了一个用户程序后，通常都会让它在用户级下执行，从而使系统不会因某个程序的崩溃或恶意破坏而受损。

1.5 Cortex-M 的中断与异常

准确来讲，中断和异常有所不同，CM 将中断特指外部中断，这里的外部又是相对 CM 处理器而言的，而非单片机的外部，异常则是特指 CM 处理器内部出现的中断。

CM 支持 16-1-4=11 个系统异常（4+1 个保留）与 240 个外部中断输入，不过具体用多少外部中断输入由厂商自己决定，通常都没用完。

表 1.2 Cortex-M3 异常类型

编号	类型	优先级	简介
0	N/A	N/A	未使用
1	复位	-3（最高）	复位
2	NMI	-2	不可屏蔽中断——外部 NMI 引脚
3	硬 fault	-1	所有被除能的 fault 都将“上访”成硬 fault

4	存储器管理 fault	可编程	MPU 访问犯规以及访问非法位置
5	总线 fault	可编程	总线错误（预取 abort 或数据 abort）
6	用法 fault	可编程	程序错误导致的异常，如无效指令或非法状态切换
7-10	保留	N/A	N/A
11	SVCall	可编程	系统服务调用
12	调试监视器	可编程	断电、数据观察点、外部调试请求
13	保留	N/A	N/A
14	PendSV	可编程	为系统设备而设的“可悬挂请求”
15	Systick	可编程	滴答定时器
16	IRQ#0	可编程	外中断#0
17	IRQ#1	可编程	外中断#1
…	…	…	…
255	IRQ#255	可编程	外中断#255

0 号异常不是什么入口地址，而是给出了复位后的 MSP 指针的初值。3-6 号 Fault 异常用于各种类型 fault 的处理。11 号 SVC 异常用于应用程序向操作系统提出请求的途径。14 号 PendSV 异常由于可悬挂特性，即可以延迟到此时无任何中断发生时才响应该异常，专门用于处理上下文切换。16-255 号异常是 CM 的外部中断，具体什么用取决于芯片厂商。

可以使用 SysTick 或者外中断用于上下文切换吗？可以，但不推荐，每个异常在设计之初都有自己的原因，更换作用与设计目的背道而驰。

向量表

表 1.2 也是中断向量表的顺序，向量表只需存储中断服务函数的地址，根据地址即可跳转到对应的服务代码处，即精简了向量表结构，也固定了向量表的结构（无论服务代码是复杂还是简单，都只用 4 个字节保存函数地址即可，因此可以固定的偏移量对应固定的中断向量）。

当一个发生的异常被 CM 内核接受，对应的异常 handler 就会执行，为了找到对应 handler 的入口地址，CM 会去查询中断向量表，而中断向量表通常放置在地址 0 处（当然也可以重定向到其他地方，这在 bootloader 中或者双系统中常常用到）。根据向量表查询结果得到对应 handler 地址，即函数地址，从而跳转到对应的中断服务函数中。

1.6 Cortex-M 的堆栈

CM 使用的是“向下生子的满栈”模型。SP 指向最后一个被压入堆栈的 32 位数值。下一次压栈时，SP 先减 4，再存入新的数值。

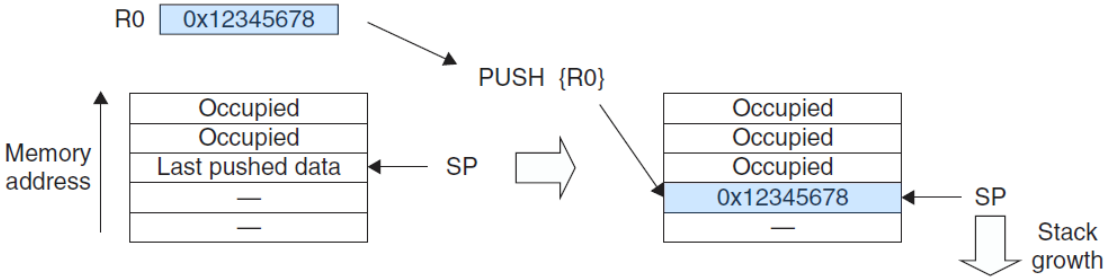


图 1.7 压栈过程

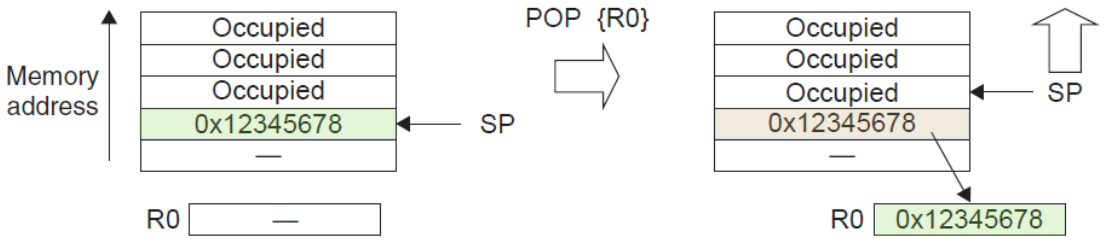


图 1.8 出栈过程

出栈过程中，虽然 POP 后被压入的数值还保存再栈中，但它已经无效了。

CM 的双堆栈机制

前面已经提过 CM 拥有 MSP 主栈指针和 PSP 进程栈指针，可以通过 CONTROL 寄存器来选择。当 CONTROL[1] = 0 时，只使用 MSP，此时用户程序和异常 handler 共享一个堆栈，这也是复位缺省值。

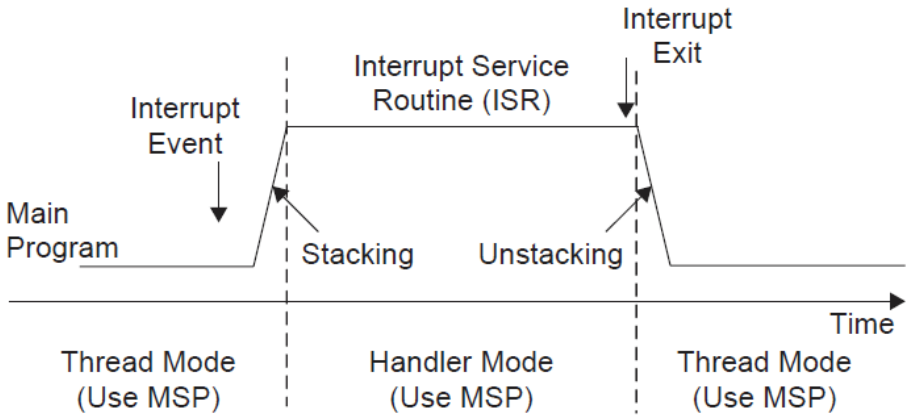


图 1.9 CONTROL[1]=0 时的堆栈使用情况

当 CONTROL[1]=1 时，线程模式将不再使用 PSP，而改用 MSP（handler 模式永远使用 MSP）。此时，进入异常时的自动压栈使用的是进程堆栈，进入异常 handler 后才自动改为 MSP，退出异常时切换回 PSP，并且从进程堆栈上弹出数据。

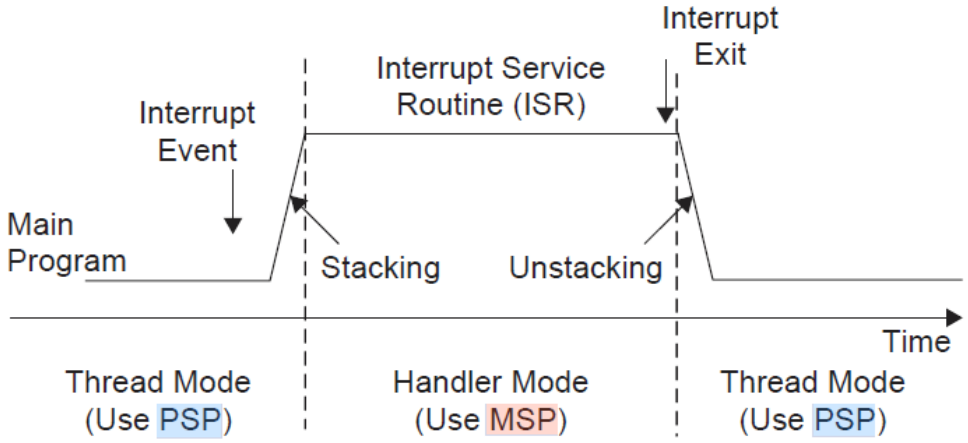


图 1.10 CONTROL[1]=0 时的堆栈切换情况

LR 链接寄存器在 handler 模式下不再存储返回地址，会被重新解释为异常返回值。高 28 位全为 1，只有[3:0]有特殊含义，如表 1.3 所示。

表 1.3 EXC_RETURN 的位段详解

位段	含义

[31:4]	EXC_RETURN的标识：必须全为1
3	0=返回后进入Handler模式 1=返回后进入线程模式
2	0=从主堆栈中做出栈操作，返回后使用MSP， 1=从进程堆栈中做出栈操作，返回后使用PSP
1	保留，必须为0
0	0=返回ARM状态。 1=返回Thumb状态。**在CM3中必须为1**

也就是说，在异常服务代码中，除了 CONTROL 寄存器，也可以通过 LR 寄存器的这些位查询或者控制异常返回后的工作模式和所用指针。位 0 表示了 2 种工作状态，CM 并不支持 ARM 状态，因此强行写 0 会引发 fault 异常。

由于在异常服务程序中，代码处于特权级，因此可以对 SP 指针进行读写或修改，从而控制栈数据，这是多任务中上下文切换的关键。

1.7 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），它们多用于在操作系统之上的软件开发中。SVC 用于产生系统函数的调用请求。例如，操作系统不让用户程序直接访问硬件，而是通过提供一些系统服务函数，用户程序使用 SVC 发出对系统服务函数的呼叫请求，以这种方法调用它们来间接访问硬件。因此，当用户程序想要控制特定的硬件时，它就会产生一个 SVC 异常，然后操作系统提供的 SVC 异常服务例程得到执行，它再调用相关的操作系统函数，后者完成用户程序请求的服务。

这种“提出要求——得到满足”的方式，很好、很强大、很方便、很灵活、很能可持续发展。首先，它使用户程序从控制硬件的繁文缛节中解脱出来，而是由 OS 负责控制具体的硬件。第二，OS 的代码可以经过充分的测试，从而能使系统更加健壮和可靠。第三，它使用户程序无需在特权级下执行，用户程序无需承担因误操作而瘫痪整个系统的风险。第四，通过 SVC 的机制，还让用户程序变得与硬件无关，因此在开发应用程序时无需了解硬件的操作细节，从而简化了开发的难度和繁琐度，并且使应用程序跨硬件平台移植成为可能。开发应用程序唯一需要知道的就是操作系统提供的应用编程接口（API），并且了解各个请求代号和参数表，然后就可以使用 SVC 来提出要求了（事实上，为使用方便，操作系统往往会提供一层封皮，以使系统调用的形式看起来和普通的函数调用一致。各封皮函数会正确使用 SVC 指令来执行系统调用——译者注）。其实，严格地讲，操作硬件的工作是由设备驱动程序完成的，只是对应用程序来说，它们也是操作系统的一部分。如图 1.11 所示。

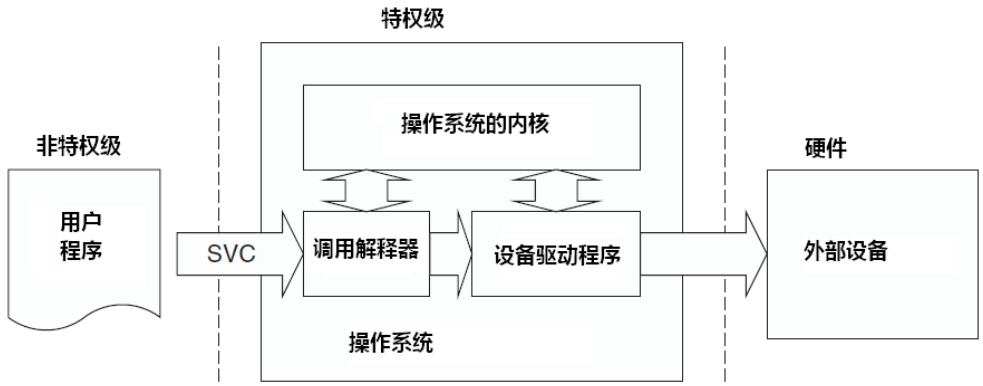


图 1.11 SVC 作为操作系统函数门户示意图

SVC 异常通过执行"SVC"指令来产生。该指令需要一个立即数，充当系统调用代号。SVC 异常服务例程稍后会提取出此代号，从而解释本次调用的具体要求，再调用相应的服务函数。例如，

SVC 0x3；调用 3 号系统服务

在 SVC 服务例程执行后，上次执行的 SVC 指令地址可以根据自动入栈的返回地址计算出。找到了 SVC 指令后，就可以读取该 SVC 指令的机器码，从机器码中萃取出立即数，就获知了请求执行的功能代号。如果用户程序使用的是 PSP，服务例程还需要先执行 MRS Rn, PSP 指令来获取应用程序的堆栈指针。通过分析 LR 的值，可以获知在 SVC 指令执行时，正在使用哪个堆栈（细节参考《Cortex-M3 权威指南》第 8 章）。

由 CM3 的中断优先级模型可知，你不能在 SVC 服务例程中嵌套使用 SVC 指令(事实上这样做也没意义)，因为同优先级的异常不能抢占自身。这种作法会产生一个用法 fault。同理，在 NMI 服务例程中也不得使用 SVC，否则将触发硬 fault。

另一个相关的异常是 PendSV（可悬起的系统调用），它和 SVC 协同使用。一方面，SVC 异常是必须立即得到响应的（若因优先级不比当前正处理的高，或是其它原因使之无法立即响应，将上访成硬 fault——译者注），应用程序执行 SVC 时都是希望所需的请求立即得到响应。另一方面，PendSV 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：

- 执行一个系统调用
- 系统滴答定时器（SYSTICK）中断，（轮转调度中需要）

让我们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，如图1.12所示。

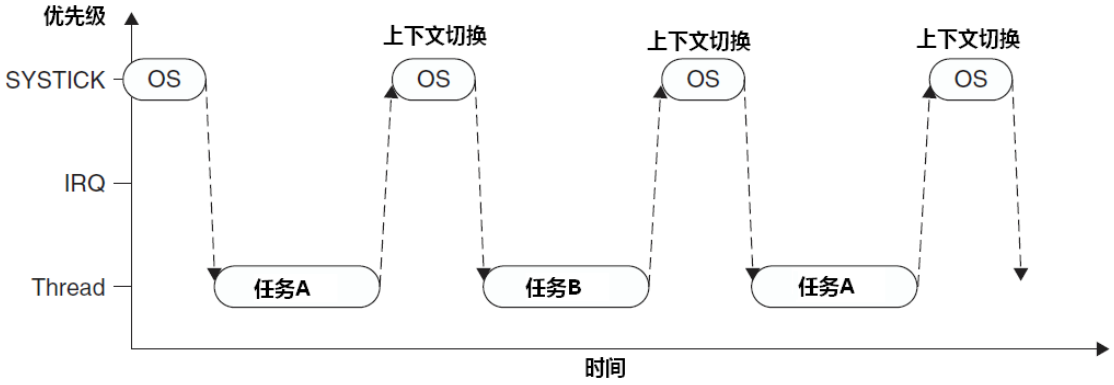


图 1.12 两个任务通过 SysTick 轮转调度的简单模式

上图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不得执行上下文切换，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CM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。

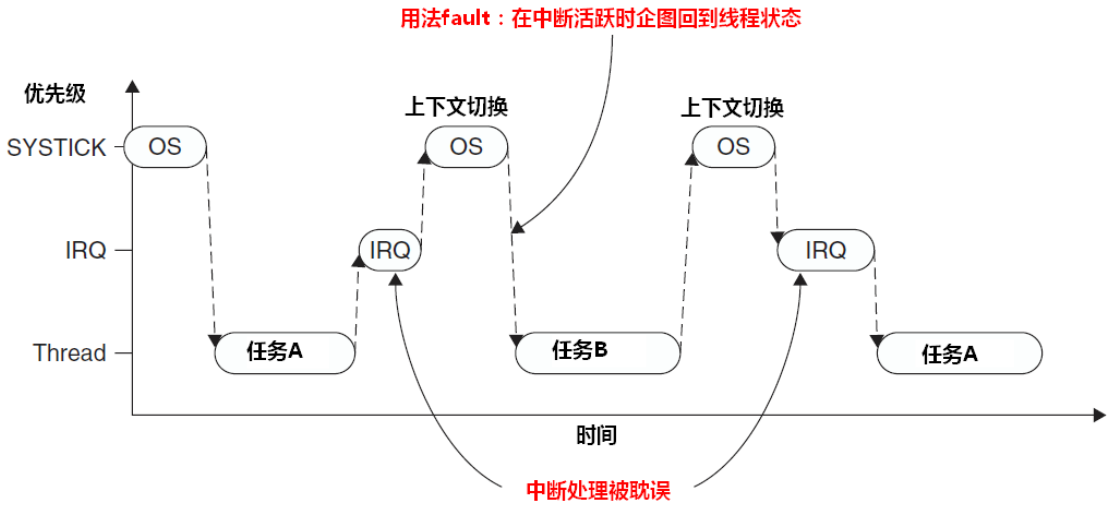


图 1.13 发生 IRQ 时上下文切换的问题

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，只有没有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”。

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。如图1.14所示

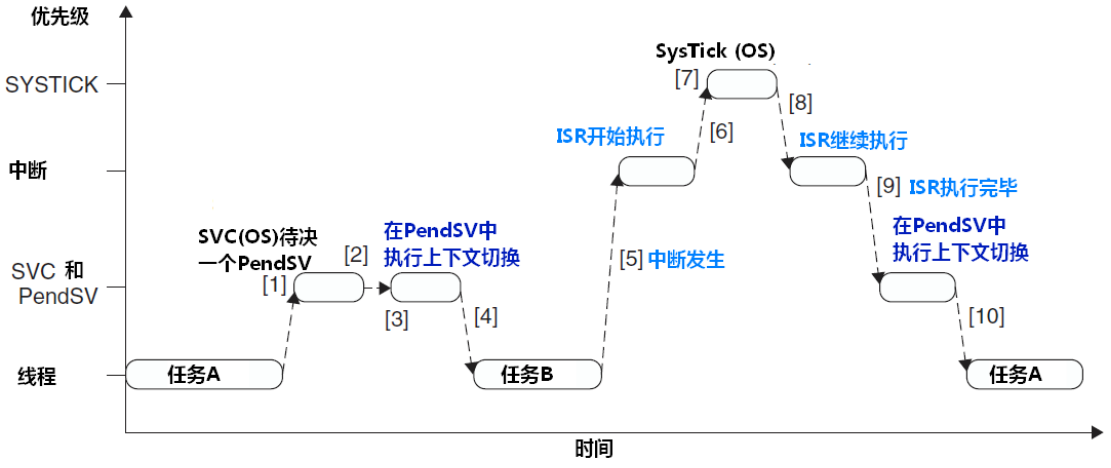


图 1.14 使用 PendSV 控制上下文切换

个中事件的流水账记录如下：

1. 任务 A 呼叫SVC 来请求任务切换（例如，等待某些工作完成）
2. OS 接收到请求，做好上下文切换的准备，并且pend 一个PendSV 异常。
3. 当 CPU 退出SVC 后，它立即进入PendSV，从而执行上下文切换。
4. 当 PendSV 执行完毕后，将返回到任务B，同时进入线程模式。
5. 发生了一个中断，并且中断服务程序开始执行
6. 在 ISR 执行过程中，发生SysTick 异常，并且抢占了该ISR。

7. OS 执行必要的操作，然后pend 起PendSV 异常以作好上下文切换的准备。
8. 当 SysTick 退出后，回到先前被抢占的ISR 中，ISR 继续执行
9. ISR 执行完毕并退出后，PendSV 服务例程开始执行，并在里面执行上下文切换
10. 当 PendSV 执行完毕后，回到任务A，同时系统再次进入线程模式。

1.8 Cortex-M 中断的具体行为

当 CM3 开始响应一个中断时，会在它看不见的体内奔涌起三股暗流：

- 入栈： 把8个寄存器的值压入栈
- 取向量：从向量表中找出对应的服务程序入口地址
- 选择堆栈指针MSP/PSP，更新堆栈指针SP，更新连接寄存器LR，更新程序计数器PC

入栈

响应异常的第一个行动，就是自动保存现场的必要部分：依次把 xPSR, PC, LR, R12 以及 R3 - R0 由硬件自动压入适当的堆栈中：如果当响应异常时，当前的代码正在使用 PSP，则压入 PSP，即使用线程堆栈；否则压入 MSP，使用主堆栈。一旦进入了服务例程，就将一直使用主堆栈。

表 1.4 寄存器压入顺序

地址	寄存器
旧SP (N - 0)	原先已压入的内容
(N - 4)	xPSR
(N - 8)	PC
(N - 12)	LR
(N - 16)	R12
(N - 20)	R3
(N - 24)	R2
(N - 28)	R1
新SP (N - 32)	R0

先把PC与xPSR的值保存，就可以更早地启动服务例程指令的预取——因为这需要修改PC；同时，也做到了在早期就可以更新xPSR中IPSR位段的值。

细心的读者一定在猜测：为啥袒护R0 - R3以及R12呢，R4 - R11就是下等公民？原来，在ARM上，有一套的C函数调用标准约定（《C/C++ Procedure Call Standard for the ARM Architecture》，AAPCS, Ref5）。个中原因就在它上面：它使得中断服务例程能用C语言编写，编译器优先使用被入栈的寄存器来保存中间结果（当然，如果程序过大也可能要用到R4 - R11，此时编译器负责生成代码来push它们。但是，ISR应该短小精悍，不要让系统如此操心——译者注）。

如果读者再仔细看，会发现R0 - R3, R12是最后被压进去的。这里也有一番良苦用心：为的是可以更容易地使用SP基址来索引寻址，（以及为了LDM等多重加载指令，因为LDM必须加载地址连续的一串数据）。参数的传递也是受益者：使之可以方便地通过压入栈的R0 - R3取出（主要为系统软件所利用，多见于SVC与PendSV中的参数传递）。

显然，如果要完成一个完整的上下文调度功能，只需要手动保存/恢复一下R4-R11寄存器即可！其余寄存器在响应异常/返回时会自动的入栈/弹出。

取向量

当数据总线（D-Code总线）正在为入栈操作而忙得团团转时，指令总线（I - Code总线）可不是凉快地坐着看热闹——它正在为响应中断紧张有序地执行另一项重要的任务：从向量表中找出正确的异常向量，然后在服务程序的入口处预取指。由此可以看到各自都有专用总线的好处：入栈与取指这两个工作能同时进行。

更新寄存器

在入栈和取向量的工作都完毕之后，执行服务例程之前，还要更新一系列的寄存器：

- SP：在入栈中会把堆栈指针（PSP或MSP）更新到新的位置。在执行服务例程后，将由MSP负责对堆栈的访问。
- PSR：IPSR位段（地处PSR的最低部分）会被更新为新响应的异常编号。
- PC：在向量取出完毕后，PC将指向服务例程的入口地址，
- LR：LR的用法将被重新解释，其值也被更新成一种特殊的值，称为“EXC_RETURN”，并且在异常返回时使用。EXC_RETURN的二进制值除了最低4位外全为1，而其最低4位则有另外的含义（见表1.3）。

以上是在响应异常时通用寄存器的变化。另一方面，在NVIC中，也伴随着更新了与之相关的若干寄存器。例如，新响应异常的悬起位将被清除，同时其活动位将被置位。

异常返回

当异常服务例程执行完毕后，需要很正式地做一个“异常返回”动作序列，从而恢复先前的系统状态，才能使被中断的程序得以继续执行。从形式上看，有3种途径可以触发异常返回序列，如表1.5所示；不管使用哪一种，都需要用到先前储的LR的值。

表 1.5 触发中断返回的指令

返回指令	工作原理
BX <reg>	当LR存储EXC_RETURN时，使用BX LR即可返回（**最常用**）
POP {PC} 和 **POP {…,PC}**	在服务例程中，LR的值常常会被压栈。此时即可使用POP指令把LR存储的EXC_RETURN往PC里弹，从而激起处理器做中断返回
LDR与LDM	把PC作为目的寄存器，亦可启动中断返回序列

有些处理器使用特殊的返回指令来标示中断返回，例如 8051 就使用 reti。但是在 CM 中，是通过把 EXC_RETURN 往 PC 里写来识别返回动作的。因此，可以使用上述的常规返回指令，从而为使用 C 语言编写服务例程扫清了最后的障碍（无需特殊的编译器命令，如__interrupt）。

在启动了中断返回序列后，下述的处理就将进行：

1. 出栈：先前压入栈中的寄存器在这里恢复。内部的出栈顺序与入栈时的相对应，堆栈指针的值也改回去。
2. 更新NVIC寄存器：伴随着异常的返回，它的活动位也被硬件清除。对于外部中断，倘若中断输入再次被置为有效，悬起位也将再次置位，新一次的中断响应序列也可随之再次开始。

嵌套的中断

CM 为我们自动的解码中断响应顺序，不高于当前正在响应异常的优先级的异常不会抢占当前异常，自己也不能抢占自己。同时自动入栈和出栈的功能可以使我们无需担心中断嵌套时会使数据损毁。

但是，我们必须计算主堆栈容量的最小安全值，由于所有异常服务代码只使用主堆栈，因此当嵌套加深时，主堆栈的压力会不断增加，每嵌套一级，至少需要 8 个字，即 32 个字节，这还没算上 ISR 内部对堆栈的额外需求，一旦堆栈被用穿（溢出），数据被破坏，必将导致程序跑飞/死机！

另外，相同的异常时不允许重入的。在异常处理期间，同级或低级的异常必须阻塞起来，待当前异常执行完毕后，方可继续响应。由此可知，在 SVC 中，不得再次使用 SVC 指令，否则 fault 伺候。

第二章 Cortex-M 指令集

CM 的指令系统内容比较多，本章对一些重要的指令进行介绍。

2.1 汇编基础

书写模式

标号

操作码 操作数 1, 操作数 2, … ；注释。

其中，标号是可选的，如果有，必须顶格。标号的作用是让汇编器来计算程序转移的地址。

操作码是指令的助记符，它的前面必须有至少一个空白符，通常使用一个“Tab”键来产生。操作码后面往往跟随若干个操作数，而第 1 个操作数，通常都给出本指令执行结果的存储地。不同指令需要不同数目的操作数，并且对操作数的语法要求也可以不同。

后缀

在 ARM 处理器中，指令可以带有后缀。

表 2.1 指令后缀介绍

后缀名	含义
S	要求更新 APSR 中的标志 s，例如： ADDS R0, R1；根据加法的结果更新 APSR 中的标志
EQ,NE,LT,GT 等	有条件地执行指令。EQ=Euqal, NE= Not Equal, LT= Less Than, GT= Greater Than。 还有若干个其它的条件。例如： BEQ <Label>；仅当 EQ 满足时转移
.W	显示的指明使用 32 位指令。 如 ADDS.W R0, R1
H	半字操作
B	字节操作
D	双字操作

统一汇编语言

CM 支持 Thumb2 指令，因此，同样的指令功能可以是 16 位的也可以是 32 位的。而统一汇编语言（UAL）允许开发者以同样的语法格式进行书写，由汇编器来决定使用 16 位指令还是 32 位指令。

2.2 指令集

本节节选部分常用指令集。

表 2.2 其他指令

名字	功能
SVC	系统服务调用
NOP	无操作
CPSIE	使能 PRIMASK(CPSIE i)/ FAULTMASK(CPSIE f)——清 0 相应的位
CPSID	除能 PRIMASK(CPSID i)/ FAULTMASK(CPSID f)——置位相应的位
LDREX	加载字到寄存器，并且在内核中标明一段地址进入了互斥访问状态
STREX	检查要写入的地址是否已进入了互斥访问状态，如果是则存储寄存器的字
MRS	加载特殊功能寄存器的值到通用寄存器
MSR	存储通用寄存器的值到特殊功能寄存器
WFE	休眠并且在发生事件时被唤醒
WFI	休眠并且在发生中断时被唤醒

表 2.3 数据操作指令

名字	功能
ADC	带进位加法
ADD	加法
AND	按位与（原文是逻辑与，有误——译注）
ASR	算术右移
BIC	位清零（把一个数按位取反后，与另一个数逻辑与）
CMP	比较两个数并更新标志位
CLZ	计算前导零的数目
EOR	按位异或
LSL	逻辑左移
LSR	逻辑右移
MOV	加载 16 位立即数到寄存器（其实汇编器会产生 MOVW——译注）
MUL	乘法
ORR	按位或（原文为逻辑或，有误——译注）
RBIT	位反转（把一个 32 位整数先用 2 进制表达，再旋转 180 度）
REV	对一个 32 位整数做按字节反转
ROR	圆圈右移
SDIV	带符号除法
SUB	减法
TEQ	测试是否相等（对两个数执行异或，更新标志但不存储结果）
TST	测试（对两个数执行按位与，更新Z 标志但不存储结果）
UDIV	无符号除法

表 2.4 数据传送指令

名字	功能
LDR	加载字到寄存器
LDM	从一片连续的地址空间中加载多个字到若干寄存器
STR	存储寄存器中的字
STM	存储若干寄存器中的字到一片连续的地址空间中
PUSH	把若干寄存器的值压入堆栈中
POP	从堆栈中弹出若干的寄存器的值

表 2.5 转移指令

名字	功能
B	无条件转移
BL	转移并连接（呼叫子程序）
TBB	以字节为单位的查表转移。从一个字节数组中选一个 8 位前向跳转地址并转移
TBH	以半字为单位的查表转移。从一个半字数组中选一个 16 位前向跳转的地址并转移

表 2.6 部分常用汇编用法举例

示例	功能描述
`LDR Rd, [Rn, #offset]`	从地址 Rn+offset 处读取一个字到Rd
`LDR R0, =NVIC_INT_CTRL`	NVIC_INT_CTRL作为地址所指向的数据传给R0
`STR R2, [R0]`	将R2数据存入R0为地址的内存中
`MRS <Rn>, <SReg>`	加载特殊功能寄存器的值到Rn
`MSR <SReg>, <Rn>`	存储Rn 的值到特殊功能寄存器
`ADDS R0, R0, #0x20`	R0=R0+0x20
`LDM R0, {R4-R11}`	从R0为起始地址的内存空间中连续读取数据依次存入R4~R11中
`STM R0, {R4-R11}`	将R4~R11寄存器数据存入以R0为起始地址的内存空间中
`BX LR`	返回子程序
`CBZ R0, label`	判断R0是0的话跳转到label标签
`EXTERN g_OS_Tcb_CurP`	声明g_OS_Tcb_CurP为一个外部变量
`EXPORT OSCtxSw`	导出标号OSCtxSw的地址，即OSCtxSw作为一个对外开放的函数使用
`CPSIE I`	开启全局中断
`CPSID I`	关闭全局中断
`NVIC_INT_CTRL EQU 0xE000ED04`	定义NVIC_INT_CTRL为一个常量

2.3 混合编程

不同的 IDE 可能使用不同的标记来表明使用汇编语言，对于 Keil MDK，若要在 C 语言中混合使用汇编语法，只需要在函数最前面添加“__asm”即可。

```
//一个求 value 后导 0 数量的代码
__asm int __calZero (int value)
{
  CMP      r0, #0x00    ;r0 是 0 的话不必比较直接返回
  BEQ      exit     ;不等于跳转 exit

  RBIT     r0, r0   ;逆序 r0
  CLZ      r0, r0     ;求 r0 前导 0
exit
  BX       lr     ;返回子程序
}
```

根据 C 语言标准调用规范，函数的返回值被存储在 R0 寄存器中，而函数入参列表中，前 4 个参数被分别放置在 R0~R3 寄存器中，对于含有更多的参数的函数，多出来的部分可能会存放在其他通用寄存器中，也可能被压入栈中，具体取决于编译器行为。

第三章 RTOS 设计

3.1 RTOS 基本组成

“实时”就是指系统必须在给定的死线（deadline，亦称作“最后期限”）内做出响应。对于一个合格的 RTOS 应能在确定的时限内完成指定的任务。容易混淆的是，实时并不是指能够快速响应，虽然这也是始终追求的一个很重要的目标，但并不是 OS 是否是 RTOS 的唯一判断依据。

一个最简单的 RTOS 是具备基础的任务调度功能。应该具备的基本元素有：

1. 上下文切换器：实现上下文环境的切换
2. 任务调度器：找出最高优先级的任务，并调用上下文切换器
3. 系统节拍器：对任务遍历，使任务控制块中的延时节拍数减1，若减为0，则将其设为就绪态，最后调用任务调度器
4. 延时函数：取消任务就绪态，并设置延时节拍，最后调用任务调度器。
5. 空闲任务：必不可少的任务，可用于CPU使用率统计，也可进入休眠态降低功耗。
6. 任务元素：包括任务主体（Task函数）、任务控制块、任务栈等
7. 全局元素：主（异常）栈、当前任务指针等

3.2 调度策略

不可调度时机

在执行临界区代码（进程切换、需要原子操作时等的代码）、中断响应期间不可调度。RTOS 必须努力缩小不可调度窗口。

调度/切换任务的时机

只有就绪进程集合发生变动时才有调度/切换的需要，而变动只会在以下情况发生：

1. 当前进程自动放弃 CPU 或运行受阻（Delay、MsgPend）
2. 新建进程、唤醒某进程（TaskCreat、TaskUnsuspend、MsgPost）
3. 进程自杀或被杀（TaskDel）

系统心跳抵达、中断异常返回时会使用调度器，但不一定会进行任务切换！

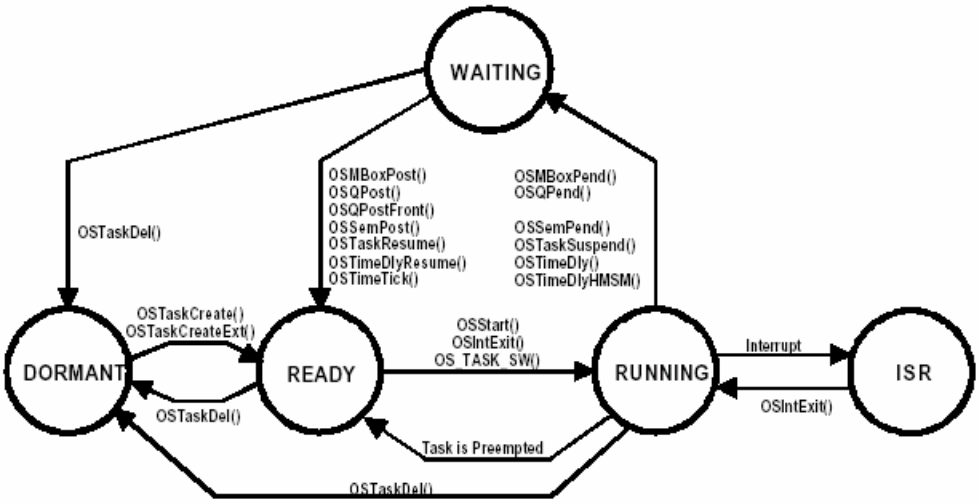


图 3.1 任务状态变化关系

3.3 常用函数基本工作

OSStart()启动函数

1. 寻找最高优先级就绪任务
2. 将当前任务 TCB 更新为刚刚找到的就绪任务
3. 初始化 PendSV、PSP 指针，并触发 PendSV

OS_Sched()调度函数

1. 在无嵌套的情况下（带 PendSV 功能的芯片可能无需考虑嵌套）
   a) 寻找最高优先级就绪任务
   b) 更新当前任务控制块为刚才找到的就绪任务
   c) 触发 PendSV 进行上下文切换

OSIntEnter()进中断函数

1. 中断嵌套数+1

OSIntExit()退中断函数

1. 中断嵌套层-1
2. 无嵌套时，寻找最高优先级任务并触发PendSV中断（任务调度）

OS_TaskIdle()空闲任务

1. 统计执行次数（也可以进入休眠态）

OSTimeDly()延时函数

1. 复位当前任务就绪态
2. 设置当前任务延时节拍数
3. 调度一个新任务

SysTick 系统心跳

1. 所有任务延时节拍-1，并设延时数为 0 的任务为就绪态（未考虑信号量挂起等）
2. 退出中断函数 OSIntExit()
   a) 中断嵌套层-1
   b) 无嵌套时，寻找最高优先级任务并触发 PendSV 中断以切换上下文（任务调度）

OS_TCBInit()任务控制块初始化函数

1. 初始化栈指针、优先级、延时节拍、任务状态、上下 TCB 链、优先级位图索引、栈统计信息、任务间通信 Flag 等

OSTaskCreate()创建一个任务

1. 初始化任务堆栈
2. 初始化 TCB
3. 将 TCB 加入 TCB 列表
4. 进行任务调度

OSTaskDel()删除任务

1. 复位就绪态
2. 将 TCB 从 TCB 列表中删除
3. DeInit TCB
4. 进行任务调度

3.4 任务调度算法

任务调度算法有很多，对于 RTOS，一个很重要的概念是时间的确定性，也就是说寻找最高优先级的过程的耗时必须是可确定的。这里只讲两种，for 循环法和优先级位图法，第一种无法在确定的时间内找到最高优先级任务，因此只做理解，并无实用意义。

for 循环法

for 循环的本质是按优先级从高到低的顺序对任务控制块列表进行遍历，将第一个就绪态的任务进行调度。

```
void OS_Task_Switch(void)
{
  OS_S32 i;
  OS_TCBP tcb_p; //任务控制块指针
  OS_USE_CRITICAL
  for (i = 0; i < OS_TASK_MAX_NUM; i++)
  {
    tcb_p = OS_TCB_TABLE[i]; //从 TCB 列表中获取 TCB
    if (tcb_p == NULL)
      continue;
    if (tcb_p->State == TASK_READY)   //找到就绪态任务
      break;
  }
  OS_ENTER_CRITICAL();
  g_OS_Tcb_HighRdyP = tcb_p;
  g_Prio_HighRdy = i;
  OS_EXIT_CRITICAL();
}
```

C 语言优先级位图法

基于优先级的调度法在嵌入式系统中很常见，uC/OS 就是使用的该种，为减少内存空间的消耗，1 个位表示一个任务是否就绪，对于 64 个任务只需要 8 个字节即可表征。为了使优先级寻找时间为常数，使用查表的方法完成。

u/COS 将 64 个任务每 8 个任务分为一组，一个组只需要用一个 char 型变量表示，8 个组成一个 OSRdyTbl[8]就绪表，在用一个 OSRdyGrp 变量表示就绪表中哪些组存在就绪任务，1 个 bit 表示 1 个组，组中任何一个任务位被置 1，则认为该组存在就绪任务，则。通过一定的技巧，从 OSRdyGrp 找到含有最高优先级就绪所在的组，以此为信息，再从 OSRdyTbl[8]中提取具体的组，再从组内找到对应的优先级。而所用的技巧便是利用 OSUnMapTbl[256]进行查表操作。

表 3.1 优先级位图示例

	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0 组								
1 组								
2 组		1	1					
3 组								
4 组						1		
5 组								
6 组								
7 组								

以表 3.1 为例，表中的 3 个 1 表示 3 个就绪态的任务，最高优先级的任务是第 2 组 Bit5，即优先级为 21。可以看出第 2 组和第 4 组存在就绪任务，因此

```
OSRdyGrp    = 0b00010100 = 0x14;
```

OSRdyTbl[2] = 0b01100000 = 0x60;
OSRdyTbl[4] = 0b00000100 = 0x04

64 个任务用 6 个 bit 即可表示，uC/OS 将高 3bit 作为分组信息，低 3bit 作为组内就绪任务信息，使用代码

```
OSRdyGrp               |= ptcb->OSTCBBitY;
OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
```

将当前任务 ptcb 置为就绪态，OSTCBBitY 与 OSTCBBitX 用于加速计算，在初始化时计算完毕。

```
ptcb->OSTCBY       = (INT8U)(prio >> 3);
ptcb->OSTCBX       = (INT8U)(prio & 0x07);
    ptcb->OSTCBBitY    = (INT8U)(1 << ptcb->OSTCBY);
    ptcb->OSTCBBitX    = (INT8U)(1 << ptcb->OSTCBX);
```

通过

```
y             = OSUnMapTbl[OSRdyGrp];
OSPrioHighRdy  = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
```

在常数时间内找到最高优先级任务。OSUnMapTbl[256]是一个表，就是将 0x00-0xff 每个数据中最低位为 1 的位数一一列举出来，它存的是下标对应最低位的 1 在第几个位置，比如下标 3 存的数应该是 0，因为 3d=11b，又比如下标 4 应该为 2，应为 4d=100b。

```
INT8U  const  OSUnMapTbl[256] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x00 to 0x0F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0  /* 0x10 to 0x1F */
};
```

汇编优先级位图法

汇编语言写的优先级位图法会让你的操作系统有些许特点，同时也更容易理解。这得益于 CM 提供的 2 个汇编指令，分别是 RBIT 位反转 180 度指令和 CLZ 计算前导 0 数量的指令。

对于表 3.1 所示的就绪表，最高优先级 21 的计算结果为 21 = 8*2+5。等号右边的 8、2、5 均可由前导 0 指令计算出来。简易代码如下

```
//一个求 value 后导 0 数量的代码
__asm int __calZero (int value)
{
        CMP     r0, #0x00       ;r0 是 0 的话不必比较直接返回
        BEQ     exit            ;不等于跳转 exit

        RBIT    r0, r0  ;逆序 r0
        CLZ     r0, r0          ;求 r0 前导 0
exit
        BX      lr              ;返回子程序
}
```

上面的函数只计算了传入参数后导 0 的数量，需要调用多次该函数才能计算出优先级，频繁出栈入栈反而可能损失性能，不妨直接写成一个函数。

```
__asm int __ OS_GetHighRdyPrio()
{
  LDR  R0, =OSRdyGrp
  LDR  R0, [R0]
  RBIT R0, R0 ;求 OSRdyGrp
  CLZ  R0, R0
  MOV  R1, R0 ;保存就绪组到 R1

  LDR  R0, =OSRdyTbl ;求就绪位
  ADD  R0, R1
  LDR  R0, [R0]
  RBIT R0, R0
  CLZ  R0, R0

  MOV  R3, #8
  MUL  R1, R3
  ADD  R0, R1
  BX   LR
}
```

3.5 利用 PendSV 实现上下文切换

利用 PendSV 可悬挂中断，可使 OS 自动的延迟上下文切换动作，直到当前无任何其他中断活动。这可以有效的解决在中断活动过程中进行任务切换而导致的中断延迟的问题。另一方面，CM 在响应中断异常的时候，会自动的入栈一部分寄存器数据，因此只需要保存剩余的寄存器即可。

main 函数的主要内容就是初始化硬件、初始化任务、启动 OS，然后进入死循环即可。

```
int main(void)
{
  //初始化 LED 和 SysTick
  LED_GPIO_Config();
  SysTick_Init();

  //栈基址指向栈底（ARM 是向下生长型栈，所以栈底在高地址）
  g_OS_CPU_ExceptStkBase = OS_CPU_ExceptStk + OS_EXCEPT_STK_SIZE;

  //创建 2 个任务
  Task_Create(&TCB_1,task1,&TASK_1_STK[TASK_1_STK_SIZE - 1]);
  Task_Create(&TCB_2,task2,&TASK_2_STK[TASK_2_STK_SIZE - 1]);

  //将任务 1 设为就绪态
  g_OS_Tcb_HighRdyP=&TCB_1;

  //OS 启动
  OSStart_Asm();

  while(1){}
}
```

硬件初始化主要是 SysTick 定时器初始化，用于提供一个周期性的时钟节拍供 OS 使用，本节则是定时调用上下文切换器。

```
vu32 sysCnt = 0;
void SysTick_Handler(void)
{
  sysCnt++;
  Task_Switch();
}
```

Task_Switch 函数是修改当前就绪任务指针，并往中断控制及状态寄存器的 PendSV 位写 1 触发 PendSV 中断。

```
OS_TCBP g_OS_Tcb_CurP;//当前任务控制块指针
OS_TCBP g_OS_Tcb_HighRdyP;//就绪任务控制块指针
#define NVIC_INT_CTRL    0xE000ED04  // 中断控制及状态寄存器
#define NVIC_PENDSVSET   0x10000000  // 第 28bit 置 1 设立 PendSV 中断
__asm int OSCtxSw()
{
    LDR      R0, =NVIC_INT_CTRL
    LDR      R1, =NVIC_PENDSVSET
    STR      R1, [R0]
    BX       LR
}

//任务切换
void Task_Switch(void)
{
  //更改就绪的任务控制块
  if(g_OS_Tcb_CurP == &TCB_1)
    g_OS_Tcb_HighRdyP = &TCB_2;
  else
    g_OS_Tcb_HighRdyP = &TCB_1;
  //上下文切换
  OSCtxSw();
}
```

Task_Create 函数目前只需要初始化堆栈，以及初始化任务控制块，由于寄存器复位后本就是随机值，因此以编号作为通用寄存器的初始值只是方便识别并无其他意义。

```
typedef unsigned int OS_STK;
typedef struct OS_TCB
{
  OS_STK *StkAddr;//存栈顶地址的指针
}OS_TCB,*OS_TCBP;

//创建任务（初始化栈+初始化任务控制块）
void Task_Create(OS_TCB *tcb,OS_TASK task,OS_STK *stk)
{
    OS_STK  *p_stk;
    p_stk      = stk;
    //R13 为 SP 指针，不需要保存
    *(--p_stk) = (OS_STK)0x01000000uL;      //xPSR，初始值
    *(--p_stk) = (OS_STK)task;              //入口地址，即任务地址
    *(--p_stk) = (OS_STK)0xFFFFFFFFuL;      // R14 (LR) 任务不应该返
回。
    *(--p_stk) = (OS_STK)0x12121212uL;      // R12
    *(--p_stk) = (OS_STK)0x03030303uL;      // R3
    *(--p_stk) = (OS_STK)0x02020202uL;      // R2
    *(--p_stk) = (OS_STK)0x01010101uL;      // R1
    *(--p_stk) = (OS_STK)0x00000000uL;      // R0

    *(--p_stk) = (OS_STK)0x11111111uL;      // R11
    *(--p_stk) = (OS_STK)0x10101010uL;      // R10
    *(--p_stk) = (OS_STK)0x09090909uL;      // R9
    *(--p_stk) = (OS_STK)0x08080808uL;      // R8
    *(--p_stk) = (OS_STK)0x07070707uL;      // R7
    *(--p_stk) = (OS_STK)0x06060606uL;      // R6
    *(--p_stk) = (OS_STK)0x05050505uL;      // R5
    *(--p_stk) = (OS_STK)0x04040404uL;      // R4

    tcb->StkAddr=p_stk;//初始化的栈顶地址赋值给任务控制块
}
```

创建的 2 个 Task 分别控制 2 个 LED 以不同频率闪烁。

```
#define TASK_1_STK_SIZE 19
#define TASK_2_STK_SIZE 19

//任务栈与其控制块
static OS_STK TASK_1_STK[TASK_1_STK_SIZE];
static OS_STK TASK_2_STK[TASK_2_STK_SIZE];
static OS_TCB TCB_1;
static OS_TCB TCB_2;

//延时函数
void delay(u32 cnt)
{
  vu32 old = sysCnt;
  while(1){
    if(sysCnt-old > cnt)
      break;
  }
}

//任务 1
void task1(void* para)
{
  while(1){
    LED1_TOGGLE;
    delay(100);
  }
}

//任务 2
void task2(void* para)
{
  while(1){
    LED2_TOGGLE;
    delay(200);
  }
}
```

main 函数在进入死循环的最后一条语句是 OSStart_Asm 函数，主要功能配置和初始化 PendSV 中断、初始化 PSP 指针和 MSP 指针、触发 PendSV 中断完成第一次上下文切换。

```
OSStart_Asm
    LDR     R0, =NVIC_SYSPRI14      ; 配置 PendSV 异常的优先级
    LDR     R1, =NVIC_PENDSV_PRI
    STRB    R1, [R0]

    MOVS    R0, #0                  ; PSP 指针初始化为 0。
    MSR     PSP, R0

    LDR     R0, =g_OS_CPU_ExceptStkBase  ; MSP 指针初始化为
g_OS_CPU_ExceptStkBase 处。
    LDR     R1, [R0]
    MSR     MSP, R1

    LDR     R0, =NVIC_INT_CTRL       ; 触发 PendSV 异常
    LDR     R1, =NVIC_PENDSVSET
    STR     R1, [R0]

    CPSIE   I                        ; 使能中断。

OSStartHang
    B       OSStartHang              ; 一个死循环，不应该一直在这执行。
```

PendSV 函数完成上下文的切换，主要工作是保存上文和切换下文，该函数也由汇编完成。

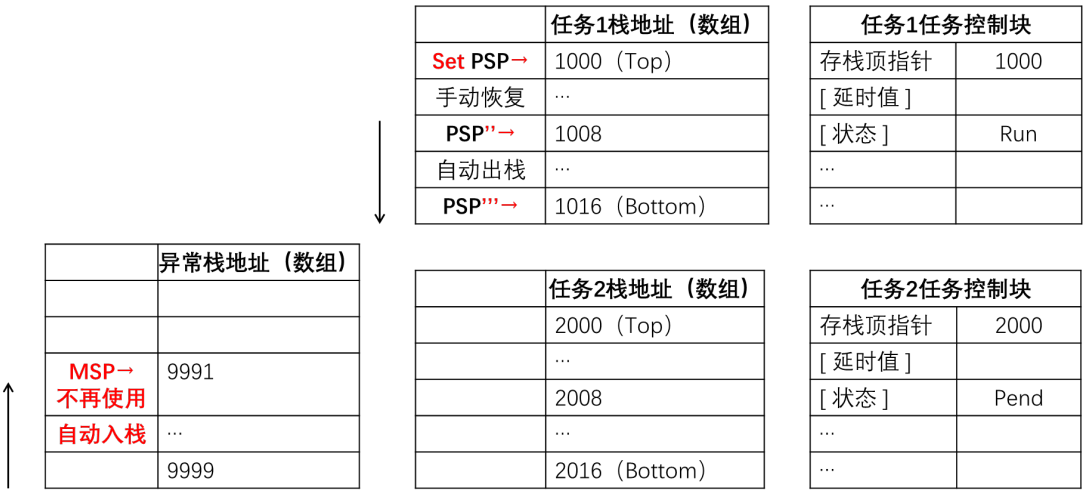
```
PendSV_Handler
    CPSID   I           ; 关闭全局中断。
    MRS     R0, PSP     ; 读取 PSP 到 R0。
    CBZ     R0, OS_CPU_PendSVHandler_nosave      ; 判断 R0 是 0 则是首次执行，无需保存上文，直接切换下文即可
   ;保存上文
    SUBS    R0, R0, #0x20       ; R0 减 0x20=32 往低地址偏移 8 个寄存器*4 字节长度，以保存剩余的 8 个寄存器。
    STM     R0, {R4-R11}        ; 将 R4-R11 的数据存到 R0 指向的连续地址中

    LDR     R1, =g_OS_Tcb_CurP    ; 把 g_OS_Tcb_CurP 地址传给 R1，g_OS_Tcb_CurP 指向的是一个 TCB 结构
    LDR     R1, [R1]        ; 获取 TCB 结构，该结构体的第一个变量就是存栈指针（栈顶地址）的变量）因此其实是传的存栈指针的变量
OSTCBCur->OSTCBStkPtr = SP;
    STR     R0, [R1]        ; 获取存栈指针的变量，并将当前 PSP 指针存进去，上文保存完成。

    ;切换下文
OS_CPU_PendSVHandler_nosave
    LDR     R0, =g_OS_Tcb_CurP                      ; 获取存储当前任务控制块地址的变量，OSTCBCur  = OSTCBHighRdy;
    LDR     R1, =g_OS_Tcb_HighRdyP                  ; 获取存储就绪任务控制块地址的变量
    LDR     R2, [R1]        ; 获取就绪任务控制块地址，存入 R2
    STR     R2, [R0]        ; 将就绪任务控制块地址赋值给当前任务控制块（刷新当前任务）

    LDR     R0, [R2]        ; 获取栈顶地址并加载进 R0。
    LDM     R0, {R4-R11}    ; 将 R0 所指向的连续数据依次读入 R4-R11。
    ADDS    R0, R0, #0x20   ; R0 偏移 0x20=32Byte

    MSR     PSP, R0         ; 新的栈顶地址传给 PSP 指针。
    ORR     LR, LR, #0x04   ; 确保异常返回后使用 PSP 指针

    CPSIE   I               ; 启动中断
    BX      LR              ; 返回子程序
```

本节程序的执行流程大致为在 main 函数中执行完 OSStart_Asm 函数后，发生 PendSV 异常，由于此时 g_OS_Tcb_HighRdyP 指向 TCB1，且首次调度，TCB1 栈数据被恢复并且 g_OS_Tcb_CurP 指向 TCB1，由于 TCB1 的任务栈保存了 LR 信息，而 LR 被初始化为

Task1 地址，因此异常返回后程序将会执行 Task1 函数。等待下一次 SysTick 中断到来后，g_OS_Tcb_HighRdyP 被指向 TCB2，进入异常后保存寄存器数据到 TCB1 任务栈中，并恢复 TCB2 任务栈，并将 g_OS_Tcb_CurP 指向 TCB2，再次进行异常返回。这样，周而复始，两个任务在定时节拍的控制下不断的轮流执行，这实际上是一种时间片调度方法。

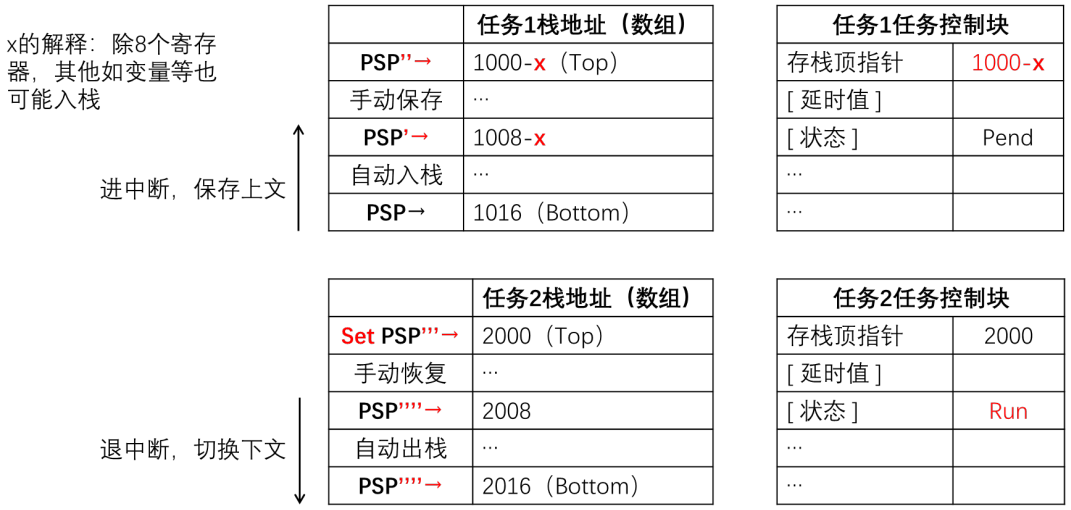
栈指针的变化状态如图 3.2 所示

（a）初始化



（b）首次调度

二次调度（保存上文切换下文）：



	任务1栈地址（数组）
PSP''→	1000-x（Top）
手动保存	…
PSP'→	1008-x
自动入栈	…
PSP→	1016（Bottom）

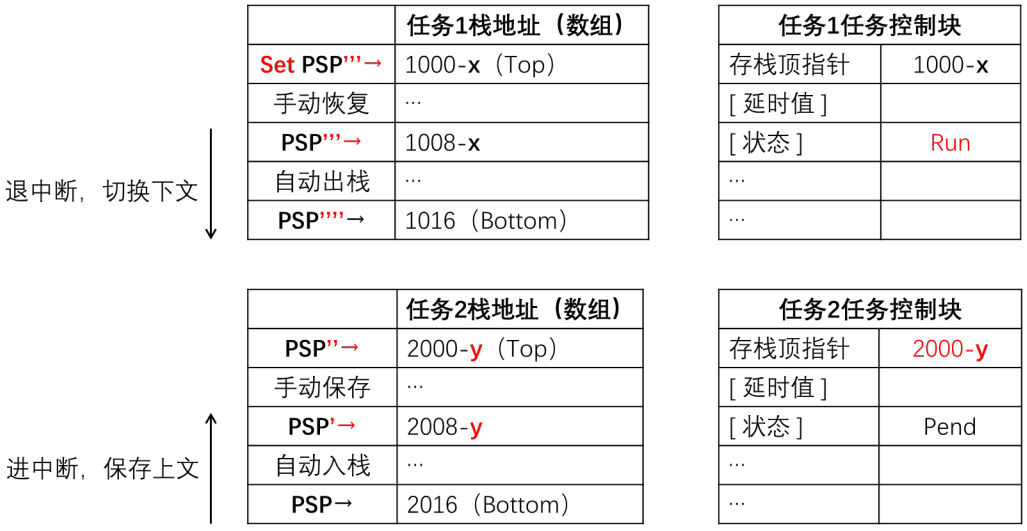
任务1任务控制块	
存栈顶指针	1000-x
[延时值]	
[状态]	Pend
…	
…	

	任务2栈地址（数组）
Set PSP'''→	2000（Top）
手动恢复	…
PSP''''→	2008
自动出栈	…
PSP''''→	2016（Bottom）

任务2任务控制块	
存栈顶指针	2000
[延时值]	
[状态]	Run
…	
…	

（c）第二次调度

三次调度（保存上文切换下文）：



	任务1栈地址（数组）
Set PSP'''→	1000-x（Top）
手动恢复	…
PSP'''→	1008-x
自动出栈	…
PSP''''→	1016（Bottom）

任务1任务控制块	
存栈顶指针	1000-x
[延时值]	
[状态]	Run
…	
…	

	任务2栈地址（数组）
PSP''→	2000-y（Top）
手动保存	…
PSP'→	2008-y
自动入栈	…
PSP→	2016（Bottom）

任务2任务控制块	
存栈顶指针	2000-y
[延时值]	
[状态]	Pend
…	
…	

（d）第三次调度

图 3.2 栈指针的变化过程

本节只实现了最简单的上下文切换，相关代码请查看附件。

3.6 简单的任务调度器

在 3.5 节中，Task_Switch 函数将就绪指针 g_OS_Tcb_HighRdyP 轮休指向 TCB1 和 TCB2，然后使用 PendSV 上下文切换器实现上下文切换（这实际上是一种时间片调度法）。而任务调度器则是在上下文切换器的基础上，添加基于优先级的任务调度算法，在有限的时间内从众多就绪任务中找到就绪任务并完成切换，本节内容请直接阅读源码。

本节内容实际上就是一个最最简单的 OS，可以看出，写一个 OS 并不难，难的是如何写出一个稳定高效的 OS，这就需要对 OS 原理、对 ARM 架构有充分的理解才行。本节的例程可以试一试在屏蔽进出临界区代码、调整延时值、使堆栈溢出等多个方面制造“问题”体现稳定的含义。

3.7 FAQ 集合

本节收集了一些在阅读 uC/OS 源码后可能出现的一些问答。

#上下文切换函数 OSCtxSw 和中断的上下文切换函数 OSIntCtxSw 有什么区别？

对于 STM32，没有区别，他们都是触发 PendSV，依靠 PendSV 的可悬挂特性实现上下文切换。对于无类似 PendSV 可悬挂中断的芯片来说，他们区别**可能**在于，中断中切换上下文时已经自动入栈了一部分数据，因此 OSIntCtxSw 可以少操心一部分数据。

#栈最大使用率统计原理？

在程序初始化前，将栈清 0，随着程序运行，栈数组会填入数据也就成为非 0 值，最后统计连续 0 值占总栈大小的百分比。

#CPU 使用率统计原理？

在所有任务开始运行前，先只开启空闲任务，并运行一个统计周期，计算在该统计周期内空闲任务被运行的总次数。此时启动其他任务，以后只需要统计空闲任务的执行次数占之前总次数的百分比即可得出 CPU 使用率。

#栈大小如何选？

最低栈大小为 64B，因为 CM3 一共有 16 个寄存器要保存，每个寄存器占 4B。一般情况下栈的使用率不应超过 70%。空余的部分一方面是为了防止一些极端的意料不到的情况，另一方面，为添加新功能提供一定方便。

#优先级反转问题？

一个低优先级任务占用一个独享资源时，此时一个高优先级任务也需要该资源，会因为得不到该任务而被挂起，此时一个中等优先级任务就绪，它不需要使用独享资源，所以会将低优先级任务挂起，中等优先级任务开始运行，但是此时高优先级任务还在挂起中，此时中等优先级任务优于高优先级任务执行，优先级发生反转。

解决办法：首选优先级继承法，继承使用该资源的最高优先级。候选优先级天花板法，两个算法的区别是，后者需要找到所有任务中最高优先级的，遍历任务表要花时间。而前者只需要将独占资源的任务优先级提升为希望抢占它的任务的优先级

#抢占式和不可抢占式区别？

不可抢占式的只有在任务主动放弃 CPU 使用权时，其他任务才可以执行。一个优点是可以使用不可重入函数。抢占式的则是一旦有高优先级任务就绪则会发生任务切换。

#何时可以用普通信号量替代互斥型信号量

互斥信号量的不同处就是内建了优先级继承机制，因此如果**没有任务对共享资源的访问有截止时间**，则可以使用普通信号量。（假设低优先级任务 L 占据打印机，此时高优先级任务 H 开始执行，一段时间后也需要占据打印机，H 由于得不到打印机被迫挂起，如果使用互斥信号量，接下来执行的任务必定是 L，如果是普通信号量，则可能是 L 和 H 之间的其他优先级任务被执行。H 得到打印机的时机将无法预估）

#如何避免死锁

死锁：两个任务无限等待对方控制的资源。

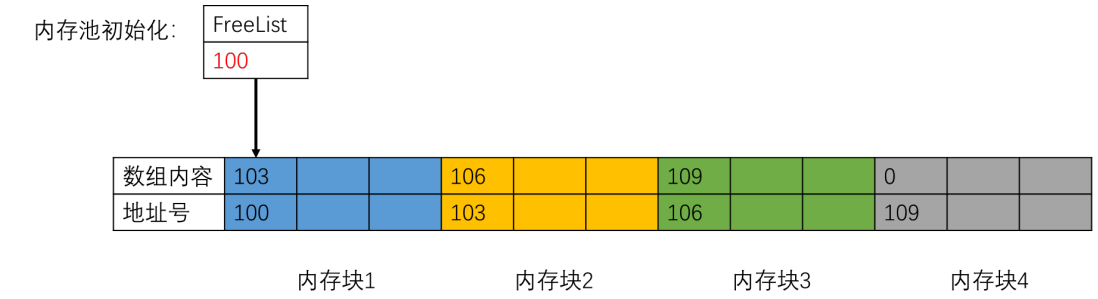
避免：让每个任务**按统一的顺序**获取到**全部资源**后再做下一步工作，同时请求信号量应**设超时时间**

#.s 启动文件基本工作

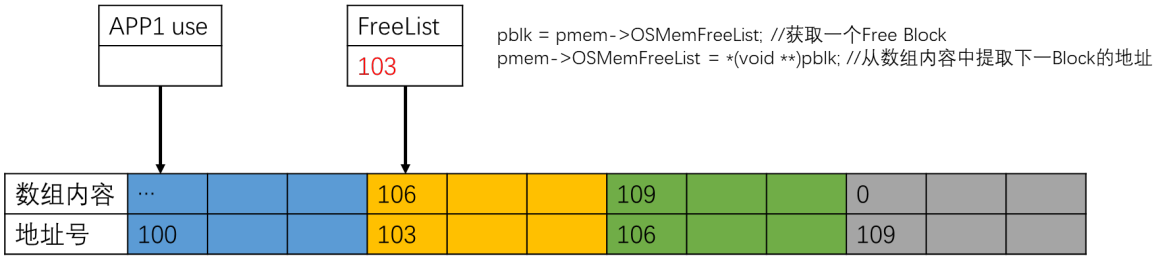
定义堆栈大小、初始化中断向量表（初始化跳转函数地址）、预先写一些弱函数形式的异常，做默认异常处理函数（复位：时钟初始化、调用 main 函数；NMI 异常、Fault 异常）

#内存管理思想

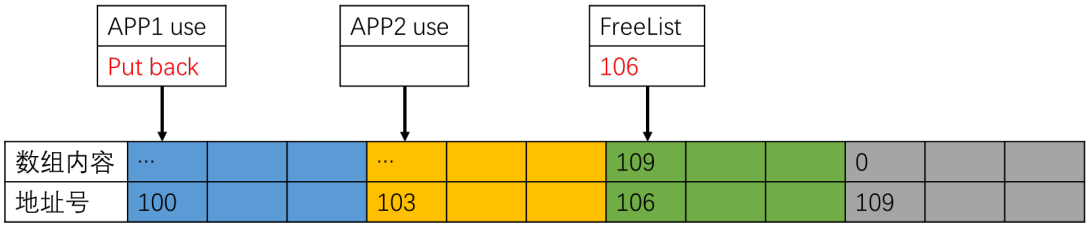
多次执行 malloc 和 free 后，会使堆内存碎片化，导致申请不到内存。为尽可能避免这个问题，uC/OS 对内存进行了简单的分块处理，原理是申请一个大的数组作为内存池并进行分块每个内存块存储下一个内存块的地址，从而链接起来，因此可用一个指针指向下一个可用内存的地址。这样便可避免内存碎片。（无 MMU 的控制器无法阻止内存访问越界，需 APP 自己注意越界问题）



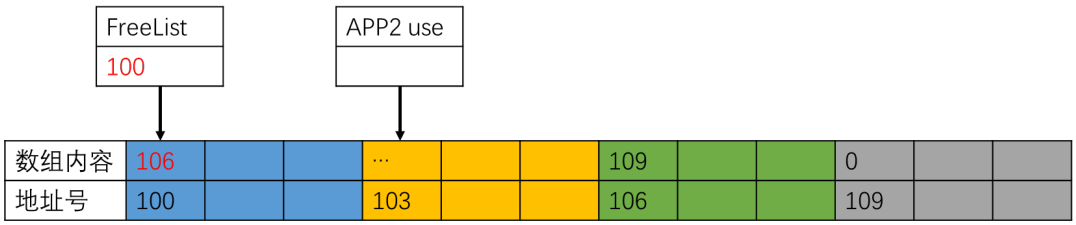
（a）内存池初始化



（b）APP1 申请一个内存块



（c）APP2 申请一个内存块后，APP1 打算释放内存块



（d）APP1 申请的内存被释放回空闲内存块

图 3.3 内存管理过程

在图 3.3 中，内存池（数组）被分成了 4 块，以不同颜色标明，每个块的第一个数组元素存储的是下一块的地址，这样的链式结构可以实现通过访问前一块的内容得到下一空闲内存块的地址。FreeList 指向空闲内存链的第一个内存块。当分配出一个内存块后，FreeList 指向下一个空闲块；当回收一个内存块后，FreeList 指向该回收的内存块，同时将该内存块的首个元素内容修改为 FreeList 原先指向的内存块地址，从而使链表再次连结。

内存块的思想比较好的解决了碎片化的问题，但也引申出了内存浪费的问题，因为分配的内存块大小总是固定的，如果内存块大小是 128Byte，而只需要 2Byte，则会浪费 126Byte 无法使用。Linux 的 buddy 内存算法较好的解决了这个问题，其实就是内存分块思想的加强版，buddy 算法以二分的思想对内存进行分块，比如内存块按 16B、32B、64B 的二次幂规律进行分块，16B 是最小内存块，分配内存时算法分配最接近所需要内存大小的内存块并尽可能切割内存块，如 APP 需要 20B，而此时 16B 和 32B 已经分配出去，buddy 算法会将 64B 切割为 2 个 32B，并将其中一个 32B 分配给 APP，在归还时，如果两个内存块大小一致且相邻，则会考虑将其合并，如该 APP 释放内存后则会将这 2 个 32B 合并回 64B。具体信息请自行查阅。

参考文献

- 宋岩（译）《Cortex-M3 权威指南 》
- 邵贝贝（译）《嵌入式实时操作系统uCOS-II》
- 《STM32参考手册》

附件

PendSV 实现的上下文切换器

easyScheduler - pendSV - f1.zip

实现简单的任务调度器

easyScheduler - os - basic - f1.zip

更加有意义的 RTOS

easyScheduler - os - general - f1.zip

下载链接

https://github.com/inhowe/stm32f1-rtos