一. 架构总览出现的问题

1. GPIO 引脚定义

```
#define GPIOA_CRL   (*(volatile uint32_t *)0x40010800)
#define GPIOA_CRH   (*(volatile uint32_t *)0x40010804)
#define GPIOA_ODR   (*(volatile uint32_t *)0x4001080C)
```

但是这时候要是用了 GPIOB

```
#define GPIOB_CRL   (*(volatile uint32_t *)0x40010C00)
#define GPIOB_CRH   (*(volatile uint32_t *)0x40010C04)
#define GPIOB_ODR   (*(volatile uint32_t *)0x40010C0C)
```

每用一个就创建一个，会占用很多的代码空间

由于每一个 GPIO 的内部排序都是同样的，只需要输入每个 GPIO 的起始地址，就可以

偏移地址

```
+0x00   CRL
+0x04   CRH
+0x08   IDR
+0x0C   ODR
+0x10   BSRR
...
```

所以我们可以先描述一次 GPIO 的“内部结构”：

```C
typedef struct
{
    volatile uint32_t CRL;
    volatile uint32_t CRH;
    volatile uint32_t IDR;
    volatile uint32_t ODR;
    volatile uint32_t BSRR;
    volatile uint32_t BRR;
    volatile uint32_t LCKR;
} GPIO_TypeDef;
```

然后只需要告诉它每个 GPIO 的**起始地址**：

```C
#define GPIOA ((GPIO_TypeDef *)0x40010800)
#define GPIOB ((GPIO_TypeDef *)0x40010C00)
#define GPIOC ((GPIO_TypeDef *)0x40011000)
```

于是：

```C
GPIOA->CRL = 0;
GPIOB->CRL = 0;
GPIOC->CRL = 0;
```

就全部解决了。

表示内部所有功能地址

2. 指令集架构 ISA

CPU 和程序之间约定的一套“机器语言规则”

比如 ISA 规定 ADD R0, R1, R2 意思是 R0 = R1 + R2

3. MPU 内存保护单元

给内存划分“权限区域”的硬件模块。防止指针乱访问内存地址

4. 栈指针 SP（MSP PSP）

OS 中有两个栈指针 MSP PSP, MSP 存放操作系统内核 / 中断，PSP 存放普通任务。

这样做的好处就是 PSP 的栈溢出不会影响到总系统的栈，也就仅仅任务失败，不会导致总系统崩溃。

5. 操作状态

- 特权级：权限高，可以访问更多系统资源
- 非特权级：权限低，某些系统寄存器不能随便访问

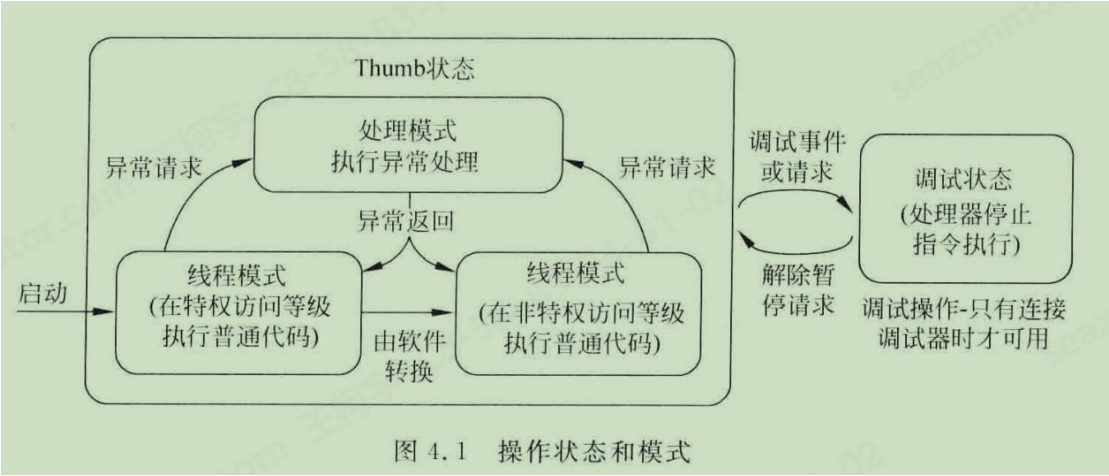


图 4.1　操作状态和模式

①处理模式默认特权级；

②裸机代码通常运行在特权级。份这些通常是防止在 OS 中的普通任务的野指针修改 OS 核心内容，所以要把普通任务操作权限降低，无法修改核心内容；

③特权级代码可以降到非特权级，但是非特权级代码要升到特权级需要进入到中断

6. SVC 服务

SVC 全称 Supervisor Call

他是一个软件主动触发异常的进入入口 SVC_Handler() 进入 SVC 中断，最大的作用是系统调用入口

程序自己执行 SVC 指令

↓

CPU 主动产生 SVCall 异常

↓

进入 SVC_Handler

相较于普通外设中断：

定时器 / 串口 / GPIO

↓

硬件产生中断

↓

进入 IRQ Handler



SVC 的作用：

①进入 SVC 去更改代码的特权级别

②RTOS 系统调用：

如 osDealy（100ms）

③启动第一个 RTOS 任务：

RTOS 第一个任务涉及很多系统级别的处理

对于 SVC_Handler（）；函数，一般是进入了 SVC 模式就会去调用这个函数，执行这个函数的内容，这个函数可以自定义内容，存放于中断向量表内，本身内部是一个弱定

义，自己写的函数将替换掉这个弱定义内容。

7. RTOS 首任务启动

```c
void Task1(void *arg)
{
    while (1)
    {
        // 任务1
    }
}

void Task2(void *arg)
{
    while (1)
    {
        // 任务2
    }
}

int main(void)
{
    xTaskCreate(Task1, "Task1", 128, NULL, 2, NULL);
    xTaskCreate(Task2, "Task2", 128, NULL, 1, NULL);

    vTaskStartScheduler();

    while (1);
}
```

对于 vTaskStartScheduler() 为开启系统调度。

TCB 里面放**任务栈顶地址 pxTop0fStack**、任务优先级、任务当前状态、任务名字、任务所属链表节点、任务通知值、互斥量相关信息等信息。最终要的是这个任务栈顶地址

以后 FreeRTOS 想恢复 Task1：

```
找到 Task1_TCB
↓
读取 pxTopOfStack
↓
PSP = pxTopOfStack
↓
从Task1栈恢复寄存器
↓
Task1继续执行
```

main()

```
  |  xTaskCreate()
  ↓
给每个任务创建：
TCB +  独立任务栈
  ↓
vTaskStartScheduler()       //开启 RTOS 任务调度器
  ↓
选出第一个要运行的任务
  ↓
xPortStartScheduler()       //Cortex-M  移植层：配置好调度相关硬件
  |
  ├── 配置  PendSV / SysTick / SVC
  ├── 启动  SysTick
  ↓
prvPortStartFirstTask()     //正式启动第一个任务
  ↓
SVC #0                      //主动触发异常
  ↓
```

vPortSVCHandler()

↓

找到 pxCurrentTCB

↓

找到第一个任务的栈顶 pxTopOfStack

↓

恢复 R4~R11

↓

PSP = 任务栈

↓

异常返回

↓

CPU 自动恢复 R0~R3、R12、LR、PC、xPSR

↓

PC = 第一个任务入口

↓

第一个任务开始运行



①任务创建

1. xPSR:程序状态寄存器。

存储的是这个任务目前所有的状态。比如任务 A

```
int a = 10;
int b = 20;
```

```
compare (a,b);
```

内核中会计算 a-b 是否大于 0，要是刚计算出 a-b 的值此时来了一个高优先级的任务，

此时所有的状态都会存入 xPSR 寄存器，下次重新启动任务时候会重新载入这个寄存器

的数据。

2. PC：Program Counter　程序计数器

```
xTaskCreate(
    LED_Task,
    "LED",
    128,
    NULL,
    1
    NULL

);
```

在创建这个任务时候，会把 LED_Task 的地址存入 PC 内，这个就是这个任务的地址，

在任务重新触发时候会跳转到这个任务内

3. LR：　link Register 链接寄存器

作用是这个函数结束后返回的地址，假设在 RTOS 中，他们的 LR 指向了返回调用这个

函数的这个行为的地址

Task A

{ fun1（）；

fun2（）；

fun3（）

}

4. RO~R12 寄存器

RO R1 R2 R3 R4 R12 这五个寄存器硬件自动保存压入栈，因为这 5 个被调用的很频繁，

比如 r0 一般是返回值 函数计算时候也都是从 r1 开始调用 r12 放的临时值。

所以在每个任务栈中会发现 R4~R11 先入栈（手动入栈），然后系统自动压入 R0~R4

和 R12 以及 xPSR PC LR。

5. 恢复任务

恢复 R4~R11 --> 自动恢复 R0-R4、R12、LR、PC、xPSR -->运行 PC 上的任务

6. 虚假的栈

对于 OS 系统，他只会遵循上述的恢复路径，但是对于新任务（没有运行过）他并没有

这些栈帧，于是在创建任务时会创建一个虚假的栈，假装他运行过，这样首次调用时候

就按照恢复任务来。

②TCB（Task Control Block，任务控制块）

在任务创建时候会创建 TCB 与任务的栈

TaskA

├—— TCB_A

└—— Stack_A



在任务 B 打断任务 A 后，栈内存的是任务 A 的信息，但是目前还是不知道任务 A 的栈存在哪，TCB 就是用来告诉系统任务 A 栈在哪。其中的 pxTop0fStack 存放的就是栈顶地址

③xPortStartScheduler()：开启 SysTick、PendSV、SVC

SysTick→ 周期性产生系统节拍

PendSV→ 真正干任务上下文切换

SVC→ 第一次启动任务

SVC → 第一次把任务“拉起来”

PendSV → 后面 Task A ↔ Task B 切换

SysTick → 告诉系统“时间到了，看看要不要切任务”

④SVC#0

Thread Mode

↓

产生 SVCall 异常

↓

进入 Handler Mode

↓

查向量表

↓

找到 SVC Handler



而在 SVC Handler 中，在 FreeRTOS 中内部存放的是 vPortSVCHandler()，意为恢复第一个任务。即通过 SVC 异常中断机制来触发运行第一个任务，代码为：

```
ldr r3, =pxCurrentTCB    //把 pxCurrentTCB 地址放入 r3，谁该运行 pxCurrentTCB 地
                                址就是谁的（看任务优先级）
ldr r1, [r3]                    // r1 去 r3 的地址内取数据
ldr r0, [r1]                    // r0 去 r1 的地址内取数据
```

通过 pxCurrentTCB 找到当前任务的 TCB，再从 TCB 里取出任务栈顶指针。

⑤PSP（栈指针）

PSP 指向第一个任务，cpu 接下来开始执行 PSP 所指向的任务

⑥任务切换原理

有两个任务 A、B。当前任务 A 正在运行，此时 PSP 指向任务 A 自己的栈。CPU 内部只有一套真正的 R0~R12、PC、LR 等硬件寄存器，此时这些寄存器中保存的是任务 A 当前运行所需要的数据和状态。当任务 B 需要抢占任务 A 时，CPU 和 FreeRTOS 会把当前 CPU 寄存器中属于任务 A 的现场数据保存到 中。因此，A 栈里所谓的 R0、R1、PC、LR 等，实际上只是这些硬 值的备份。保存完 A 的现场后，FreeRTOS 找到任务 B 的栈，并让 任务栈，然后把 B 栈中之前保存的寄存器值恢复到 CPU 那一套真正 最后恢复 B 的 PC，CPU 就从 B 上一次停止的位置继续执行。

```
Task1 正在运行
PSP → Task1 的任务栈
↓
SysTick 到来
↓
FreeRTOS 判断需要切换任务
↓
PendSV 被挂起
↓
CPU 响应 PendSV 异常
↓
【硬件自动压栈】
把 Task1 的
R0-R3、R12、LR、PC、xPSR
压入 Task1 的 PSP 栈
↓
CPU 进入 Handler Mode
PendSV_Handler 开始执行
此时 Handler 使用 MSP
↓
读取当前 PSP
↓
【软件保存】
把 Task1 的 R4-R11
继续压入 Task1 的 PSP 栈
↓
把当前 PSP
保存到 Task1_TCB->pxTopOfStack
↓
调度器选择 Task2
↓
pxCurrentTCB → Task2_TCB
↓
从 Task2_TCB->pxTopOfStack
取出 Task2 上次保存的栈顶地址
↓
PSP = Task2 的栈顶地址
↓
【软件恢复】
从 Task2 的 PSP 栈
恢复 R4-R11
↓
执行异常返回
↓
【硬件自动恢复】
从 Task2 的 PSP 栈恢复
R0-R3、R12、LR、PC、xPSR
```

⑦ 首次启动与后续启动

第一次：

```
main
 ↓
vTaskStartScheduler
 ↓
SVC
 ↓
恢复Task1
 ↓
Task1运行
```

以后 Task1 → Task2：

```
Task1运行
    ↓
SysTick发现时间到了
    ↓
PendSV挂起
    ↓
进入PendSV_Handler
    ↓
保存Task1现场
    ↓
pxCurrentTCB = Task2
    ↓
恢复Task2现场
    ↓
异常返回
    ↓
Task2运行
```

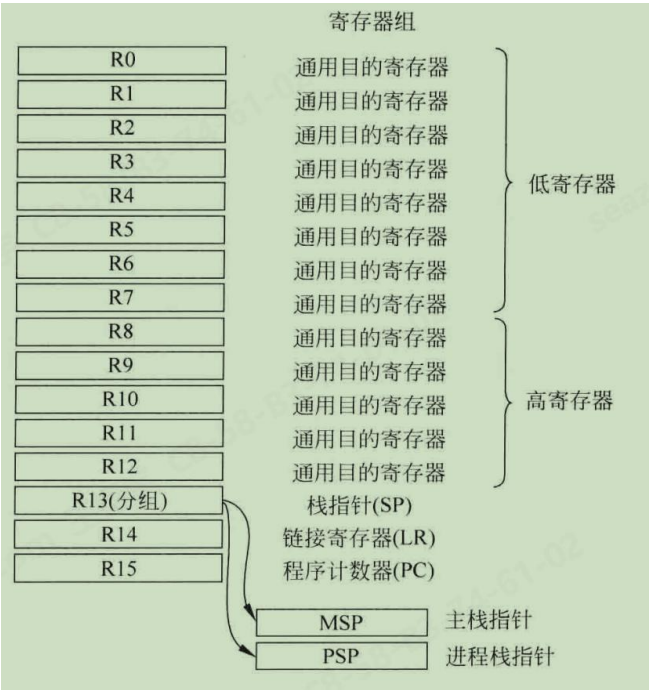
第一次：

SVC 负责“恢复一个任务”

以后：

PendSV 负责“保存旧任务 + 恢复新任务”

8. 寄存器



之前那个是任务的，这个是 CPU 的，之前的那个_xPSR 算是特殊寄存器，不在这 16 之内，要放到 r15 之下。

除了_xPSR 外还有：

CONTROL:控制特权级/非特权级；

PRIMASK：普通中断屏蔽开关；

BASEPRI：按优先级屏蔽中断；

FAULTMASK：更强的异常屏蔽；

MSP：主栈指针

PSP：进程栈指针

①LR（Link Register，链接寄存器）

```
void A(void)
{
    B();
    x = 1;
}
```

运行逻辑为调用 B 函数->LR=B 完成后返回地址->运行 B->PC = LR->回到 A 执行

但是如

```
void A(void)
{
    B();
}
```

```
TaskA 正在运行
Thread Mode
使用 PSP
↓
CAN 中断到来
↓
NVIC 判断中断优先级
↓
CPU 自动把
R0-R3、R12、LR、PC、xPSR
压到 TaskA 的 PSP 栈
↓
CPU 进入 Handler Mode
↓
中断函数开始运行
↓
中断函数使用 MSP
↓
执行 CAN_IRQHandler()
↓
中断结束
↓
异常返回
↓
CPU 从 TaskA 的 PSP
恢复 R0-R3、R12、LR、PC、xPSR
↓
继续运行 TaskA
```

```
void B(void)
{
    C();
}
```

A 调用 B 时候 LR 为回 A 的地址，B 调用 C 时候 LR 为回 B 的地址，所以调用 C 时候会覆盖掉回 A 的地址，于是在进入 B 时候发现 B 需要也调用函数，需要把此时的 LR 也就是返回 A 的地址压入栈，等 B 需要调用的 A 结束返回 B 后再出栈。

对于异常情况时，LR=EXC_RETURN，用来控制异常返回

②PC（Program Counter，程序计数器）

代表了程序下一步需要执行指令的地址。

对于读指令，有提前性：

当前正在运行　　0x00002

已经准备好运行　　0x00004

提前取指令　　0x00006

所以现在要是发一个读取 PC 的指令，读取倒是就是 0x00006 也就是正在运行的+4

对于写指令，写 PC 就等于跳转。于是上述所说的在函数运行结束后会 PC=LR

9. 特殊寄存器

①程序状态寄存器 xPSR

分别为应用 PSR（APSR)、执行 PSR(EPSR)、中断 PSR（IPSR)

	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				GE*							
IPSR												异常编号				
EPSR						ICI/IT	T			ICI/IT						

APSR：主要放运算结果的标志位

N	负数标志
Z	零标志
C	进位/借位相关标志
V	有符号溢出标志
Q	饱和运算标志

IPSR: 记录“现在正在处理哪个异常”,IPSR 主要保存当前异常编号。

I[SR=0 代表没有异常，要是！=0，表示当前正在处理某种异常或者中断

EPSR：记录“CPU 当前怎么执行指令”

T 是 Thumb 状态位，正常=1；

IT 是来自 Thumb 的 IT 指令（If-Then），它的作用是让后面几条指令根据条件决定要不要执行；

ICI：Interrupt-Continuable Instruction，若在运行时发生中断，这给记录的就是程序运行到哪一步了，等中断结束后好恢复运行

②PRIMASK、BASEPRI、FAULTMASK

意义就是保护一些代码不受中断干扰，比如 RTOS 初始化，要是此时中断介入，会死机

PRIMASK 寄存器：普通中断屏蔽开关；

PRIMASK = 0 不屏蔽中断　PRIMASK = 0 屏幕大部分中断

NMI（不可屏蔽中断）和 HardFault（硬故障异常）不受此影响；

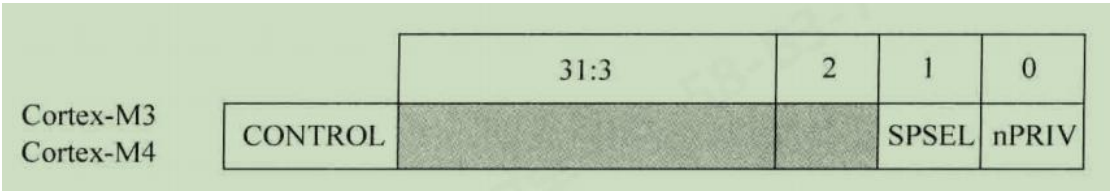
FAULTMASK 寄存器：更强的异常屏蔽；

与上述同理，但是它可以屏蔽 HardFault，不能屏蔽 NMI

BASEPRI 寄存器：按优先级屏蔽中断（数值越小优先级越高）；

BASEPRI = 3，也就是优先级 1、2 可以进，3 以上包括 3 屏蔽掉

③CONTROL 寄存器



nPRIV = 0 特权级 反之

SPSEL = 0 使用 MSP　=1 使用 PSP，hander mode 只能 MSP

这两个可以组合成 4 种情况

组合 00：裸机常用

组合 01：使用自己任务栈的特权任务，RTOS 常用

组合 11：受保护的 RTOS

组合 10:一般不用，没意义

注：M3/M4 一般有 16 个普通寄存器，每一个都是 32 位的（也就是 32bit）

特殊寄存器也是 32 位的

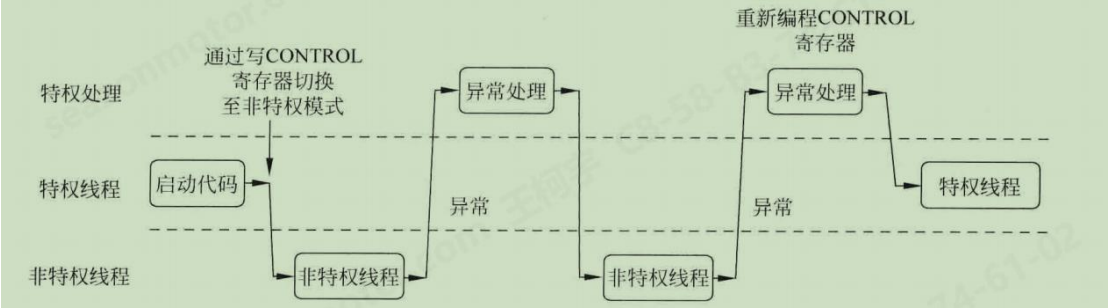
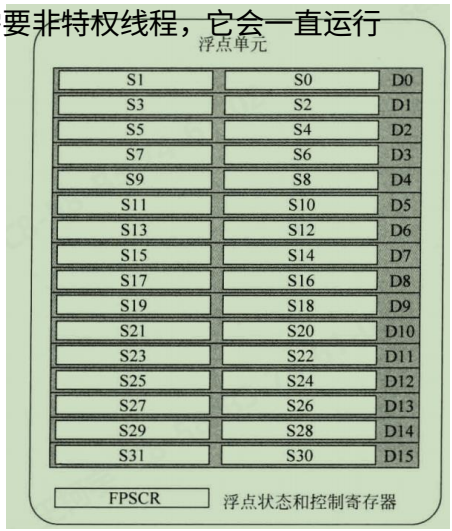


图 4.11　特权线程模式和非特权线程模式间的切换

完善之前的逻辑，特权线程可以转到非特权线程，但是非特权线程需要进入异常模式次啊能转换为特权线程！这是针对 OS 来说，裸机程序不需要非特权线程，它会一直运行在特权线程。



10. 浮点寄存器

浮点状态和寄存器（FPSCR)

其中 s0-s31 为寄存器，一个 D 为两个 s 之和

11. 内存地址映射

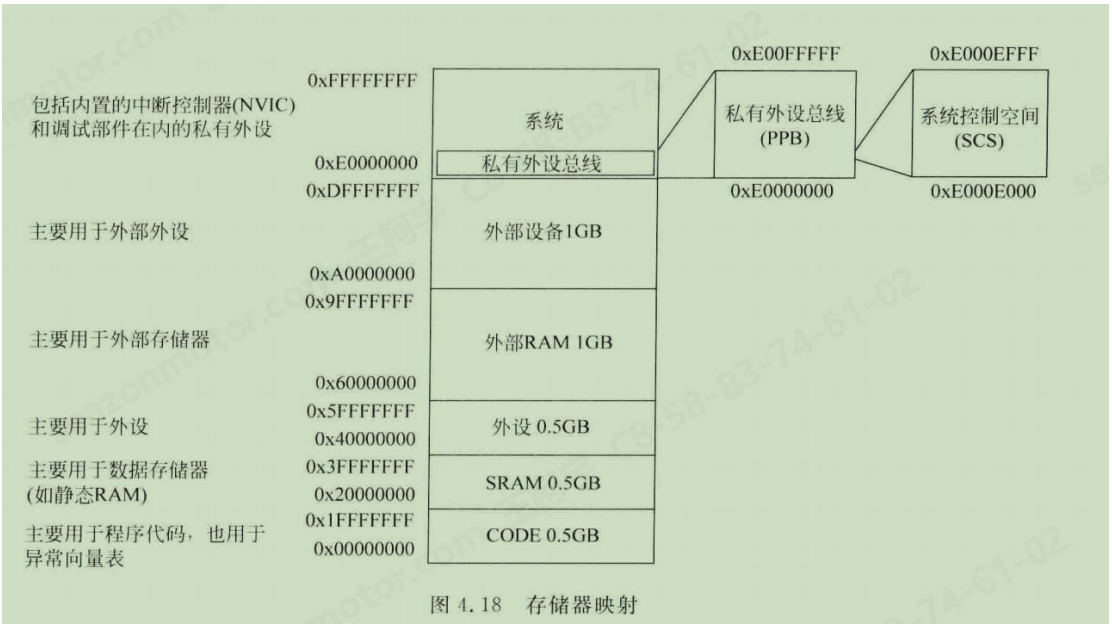


图 4.18　存储器映射

表示各个模块一般在储存里的位置，与他们并不是叠在一起的，而是各自有各自的位置

12. 向量表

档产生异常事件后（中断/错误），内核受理之后都会进行处理，向量表就是对应的异

异常类型	CMSIS 中断编号	地址偏移	向量
18～255	2～239	0x48 - 0x3FF	IRQ #2-#239
17	1	0x44	IRQ#1
16	0	0x40	IRQ#0
15	–1	0x3C	SysTick
14	–2	0x38	PendSV
NA	NA	0x34	保留
12	–4	0x30	调试监控
11	–5	0x2C	SVC
NA	NA	0x28	保留
NA	NA	0x24	保留
NA	NA	0x20	保留
NA	NA	0x1C	保留
6	–10	0x18	使用错误
4	–11	0x14	总线错误
4	–12	0x10	MemManage错误
3	–13	0x0C	HardFault
2	–14	0x08	NMI
1	NA	0x04	复位
NA	NA	0x00	MSP初始值

图 4.26　异常类型(异常向量的最低位应该置 1,以表示 Thumb 状态)

常类型。向量表是可以重定位的，由向量表偏移寄存器控制（VTOR)，复位时默认为0。

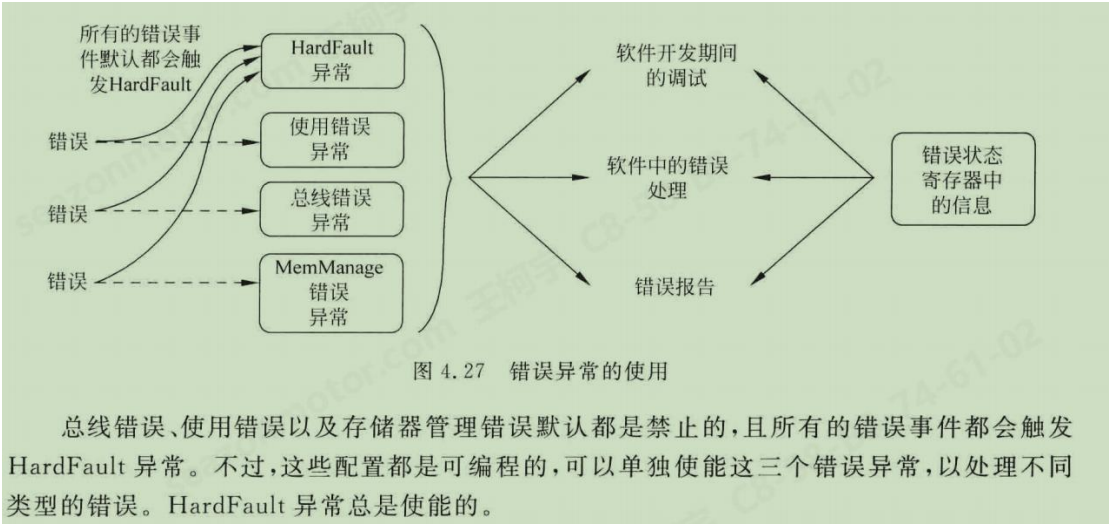


图 4.27　错误异常的使用

总线错误、使用错误以及存储器管理错误默认都是禁止的，且所有的错误事件都会触发 HardFault 异常。不过，这些配置都是可编程的，可以单独使能这三个错误异常，以处理不同类型的错误。HardFault 异常总是使能的。

13. 复位

在系统复位之前要设置 MSP 主栈指针，这样做是防止复位时由中断进入，因为中断进入时需要把正在运行的程序放进栈内保存，中断结束后再出栈，所以要提前设置一下 MSP 主栈指针。

复位

↓

硬件先设置 MSP //先保存在内部存储种

↓

保证 NMI/HardFault 随时来了都有栈

↓

读取 Reset_Handler 地址

↓

启动代码运行

↓

必要时重新调整 MSP　//若外部存储激活就可以二次修改 MSP 保存在外部存储中

↓

main()

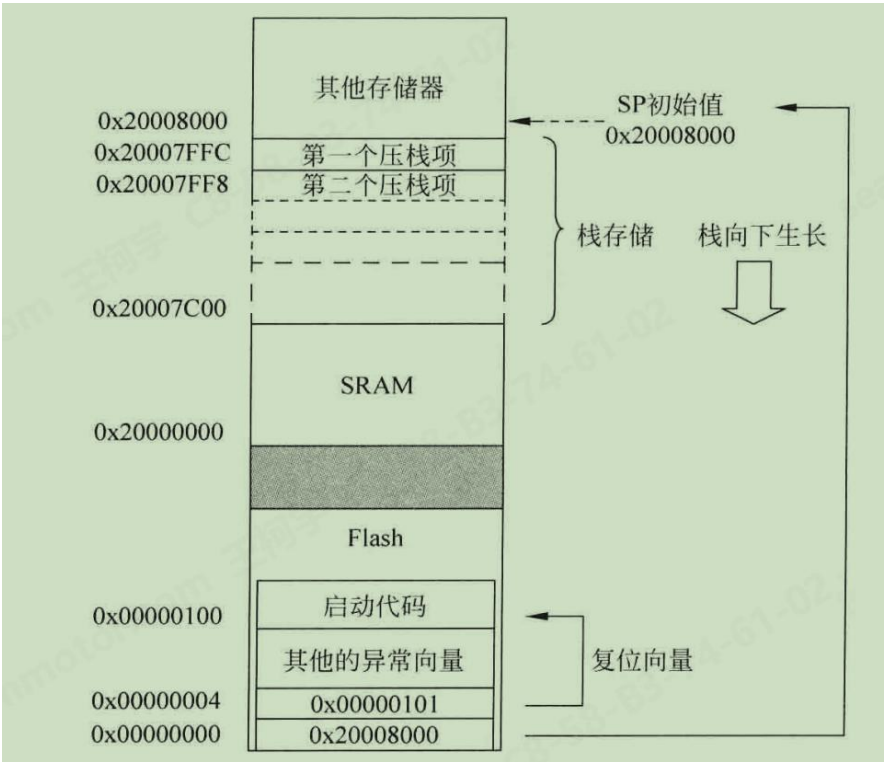


图 4.31　栈指针初始值和程序计数器初始值示例

假设这个存储的空间为 0x20007C00~0x20007FFF(1KB)

复位一开始，系统会自动设置 0x00000000 的 MSP 以及 0x00000004 的 PC

读取到 0x00000000 地址的值为 0x20008000,表示 MSP 主栈一开始从 SRAM 的这个位置开始使用。

既然地址最大为 0x20007FFF，那位啥是 0x20008000?

答：这个 SP 遵循的满栈递减，SP 一开始指向栈区域上边界的“下一个位置”，真正压数据时先减 SP，再写数据。

举个例子，SP 初始值 = 0x20008000，现在要压一个 32 位数据，也就是 4 字节：

SP = SP - 4

= 0x20007FFC

对于 PC 读取到 0X00000101，第一位的 1 为 Thumb 状态，真正的代码是从 0x00000100 开始的

二. 存储器

1. 大端和小端

如 0x12345678　存储地址为 0x2000000

0x12345678　　12 是高位字节　78 是低位字节

0x2000000　低地址

⬇

0x2000001　高地址

小端（低位放低地址）：

0x20000000　　0x78
0x20000001　　0x56
0x20000002　　0x34
0x20000003　　0x12

大端（高位放低地址）

0x20000000　　0x12
0x20000001　　0x34
0x20000002　　0x56
0x20000003　　0x78

2. 数据对齐

注：一个地址可以放一个字节（8bit）

字=4 字节=32bit

半字=2 字节=16bit

字节=8bit

他们在内存里存放时候最好要放在自己字节的整数倍位置

如 32 位的数据可以放在

0x20000000

0x20000004

0x20000008

解释：cotex m3/m4 一般一次性搬运 32bit 也即是 4 个字节搬运一次。如果 32 位在

0x2000002 的地方存放，那么就是前两个字节在第一个搬运块中，后两个字节在第二次

搬运块中，系统后期再自动拼接回造成性能损耗。

3. 内存地址中 bit 的访问

一个地址为一个字节（8bit）

0x00000
0x00004
0x00008

可以通过这个地址访问，但是想要知道其中具体的 bit 是 1 还是 0 需要进行&操作

如：a 的地址为 0x00003 其中 8bit 分别为 10101100 （bit7~bit0）

想要知道 bit3 是什么

```
if (a &  （1<<2) )  //这是个 bool 结果只有 0000100 为 1，000000 为 0
{
```

```
bit3 = 1     //bool 为 1 时
}
```

修改 bit3 为 1

```
a |= (1 << 2);
```

修改 bit3 为 0

```
a&=~（1<<2)
```

4. 位段操作

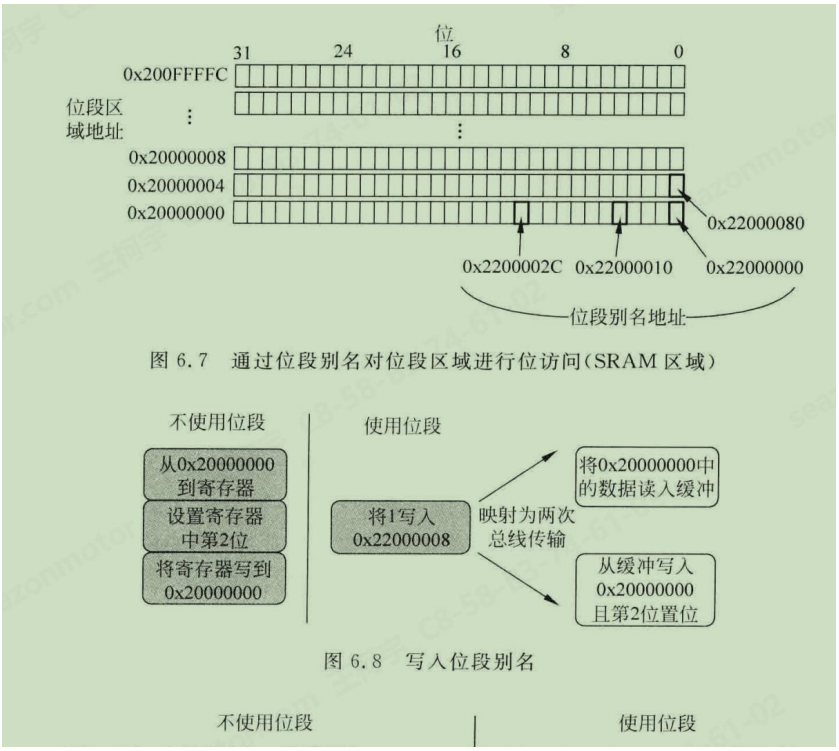


图 6.7 通过位段别名对位段区域进行位访问(SRAM 区域)

图 6.8 写入位段别名

不使用位段 | 使用位段

上述写一个地址上的一个 bit 还需要读取寄存器，然后再&操作，这个位带操作就是用来直接操作 bit，芯片原厂会规定一个 8 位地址作为一个 bit 的映射。如 0x22000010 就代

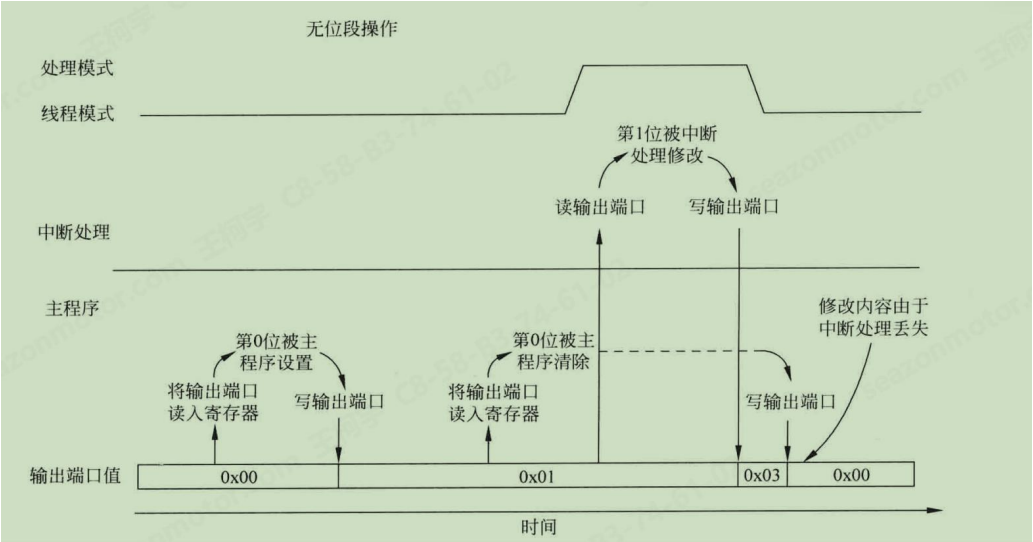
表了 0x20000000 的 bit4。映射规律就是 0x2000000+0x0200000+0x0000004*bit 位

这种用一字节地址去映射 1bit 的奢侈操作自然而然没法通用，它只适用于一些特殊区域

如 SRAM，被映射的地址无法使用，这是单独划分出来为映射而生的地址。

优势：非位段操作流程是 读取寄存器->程序修改位->写入寄存器，可被中断打断

位段操作是针对某一位的 读->改->写，不可被打断



0x00 为 0000 0000

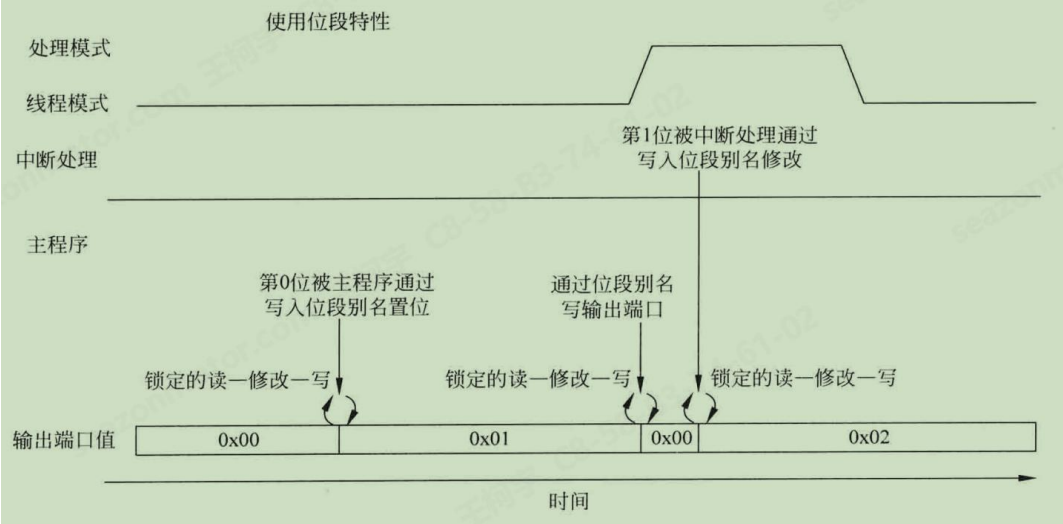
第一步操作 0000 0001 = 0x01

第二步操作应该为 0000 0000 = 0x00

第三步操作为 0000 0010 = 0x02

第三步在第二部中间打断产生，此时第二步只是修改完了还没写入，此时第三步进入为

0000 0011 = 0x03，只后第二步完成后写入寄存器 0000 0000 = 0x00，明显结果错误



对于 C 语言的位带操作

一是使用#define

```
#define DEVICE_REG0          *((volatile unsigned long *) (0x40000000))
#define DEVICE_REG0_BIT0     *((volatile unsigned long *) (0x42000000))
#define DEVICE_REG0_BIT1     *((volatile unsigned long *) (0x42000004))
...
DEVICE_REG0 = 0xAB;                 //使用普通地址访问硬件寄存器
...
DEVICE_REG0 = DEVICE_REG0 j 0x2; //未使用位段特性设置第 1 位
...
DEVICE_REG0_BIT1 = 0x1;             //利用位段特性通过位段别名地址设置第 1 位
```

二是由于对应的位段地址有规律，可以直接定义一个宏

```
//将位段地址和位编号转换为位段别名地址
#define BIT BAND(addr, bitnum) ((addr & 0xF0000000) + 0x2000000 + ((addr &0xFFFFF)<< 5) +
(bitnum << 2))
//将地址转换为指针
#define MEM_ADDR(addr) *((volatile unsigned long *) (addr))
```

基于之前的例子，可以将代码重写如下：

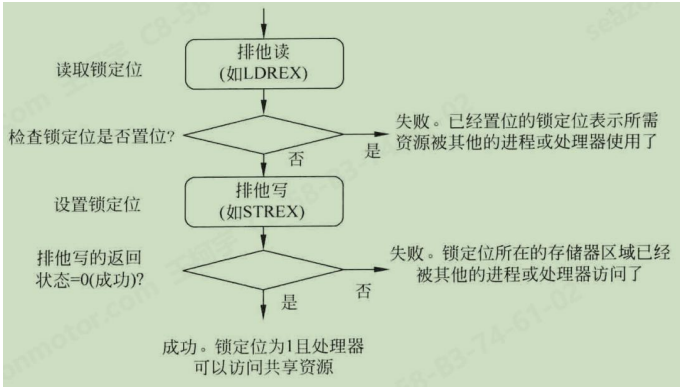
```
#define DEVICE_REG0                0x40000000
#define BIT BAND(addr,bitnum)      ((addr & 0xF0000000) + 0x02000000 + ((addr&0xFFFFF)<< 5)1
(bitnum << 2))
#define MEM_ADDR(addr)             *((volatile unsigned long *) (addr))
...
MEM_ADDR(DEVICE_REG0) = 0xAB;    //利用普通地址访问硬件寄存器
...
//未使用位段特性设置第 1 位
MEM_ADDR(DEVICE_REG0) = MEM_ADDR(DEVICE_REG0) | 0x2;
...
//利用位段特性设置第 1 位
MEM_ADDR(BIT BAND(DEVICE_REG0,1)) = 0x1;
```

5. 访问属性

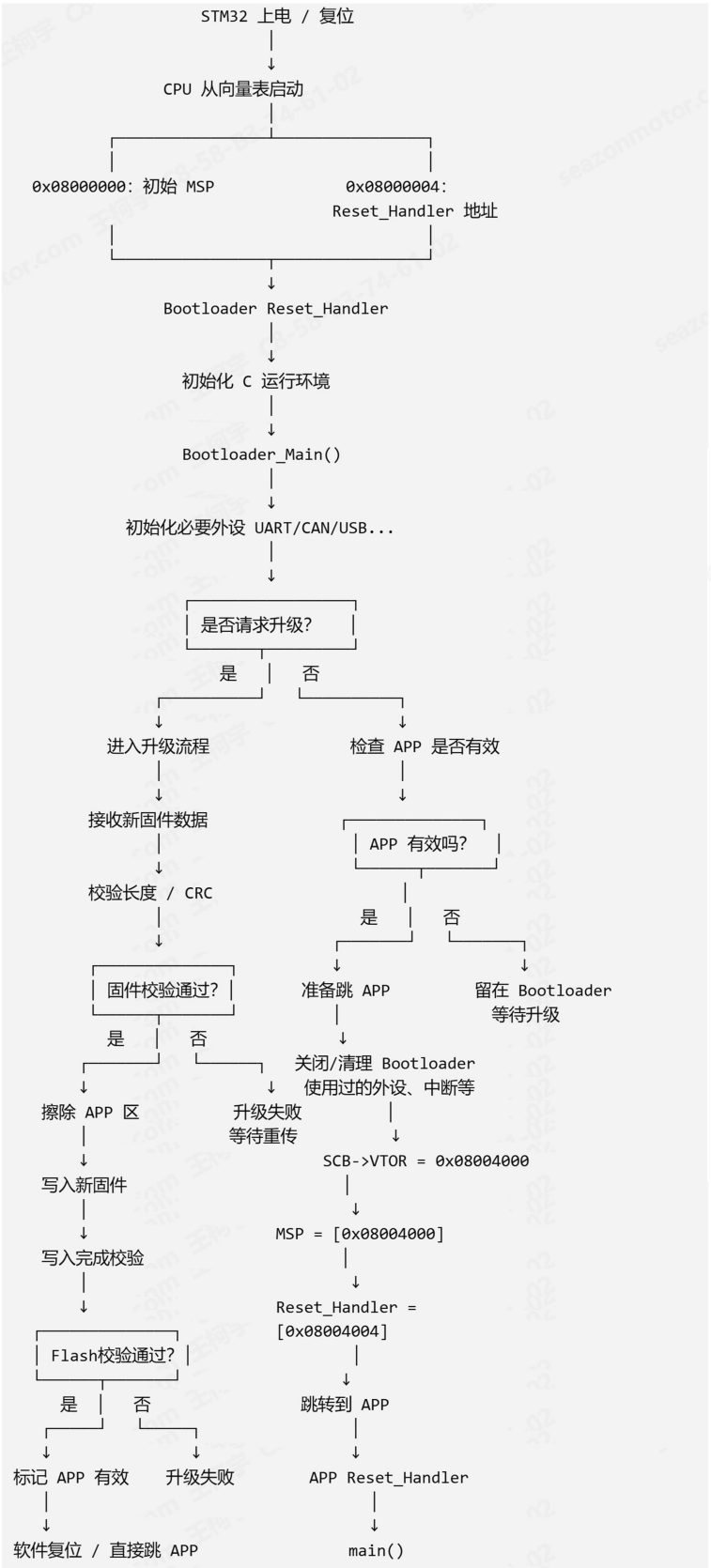
存储器有 4 个特性：可缓冲、可缓存、可执行、可共享

- 可缓冲：cpu 发送写入指令后干别的去了，这个写入指令可以一会存入
- 可缓存：如 cache（缓存），cpu 处理很快，它要数据时候会先看看 cache 里有没有，·没有就去 flash 拿一份，然后顺手存入 cache，下次要数据就可以从 cache 读取。
- 可执行:可以设置为不可执行，执行到会报错
- 可共享：存储不仅仅可以由 CPU 访问，dma 等也会访问，注意数据一致性！

6. 排他访问



7. BootLoader



对应 Flash 布局可以再配一张：

cotex 内核规定：

在 CPU 复位后：

读 0x00000000 → 初始 MSP

读 0x00000004 → Reset_Handler

芯片不同映射区域不一样，比如 stm32 就把

0x08000000 映射到了 0x0000000

对应 0x08000004 就为 Reset_Handler

APP（0x08004000~0x08004FFF)

```
0x08004000  ──────────────
            初始 MSP
0x08004004  ──────────────
            Reset_Handler 地址
0x08004008  ──────────────
            NMI_Handler 地址
0x0800400C  ──────────────
            HardFault_Handler 地址
            ...
            其他中断向量
            ...
0x08004100  ──────────────
            Reset_Handler 真正代码
            ↓
            启动代码
            ↓
            main()
            ↓
            其他函数代码
```

三. 异常与中断

1. NVIC 优先级设定

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
已使用			未使用				

图 7.2 3 位优先级寄存器(8 个可编程优先级)

根据优先级寄存器来配置，开 3 个就有 7 种，开 4 个就有 16 种

注：有三个优先级是最大的-3 的复位 -2 的 NMI（不可屏蔽中断） -1 的硬件错误

对于图 7.2 中的例子，由于第 0～4 位未实现，它们读出总是为 0，对这些位的写操作会被忽略。根据这种设置，可能的优先级为 0x00(高优先级)、0x40、0x60、0x80、0xA0、0xC0 以及 0xE0(最低)，如图 7.4 所示。

一般都是低位无效，高位有效（无效为 0），随着处理器不一样，它高几位有效也不一样，这样先用高位也是为了方便移植到不同芯片。并且一般也不会全开，因为优先级查询会遍历，太多会影响性能。

存储器地址		异常编号
0x0000004C	中断#3向量 1	19
0x00000048	中断#2向量 1	18
0x00000044	中断#1向量 1	17
0x00000040	中断#0向量 1	16
0x0000003C	SysTick向量 1	15
0x00000038	PendSV向量 1	14
0x00000034	未使用	13
0x00000030	调试监控向量 1	12
0x0000002C	SVC向量 1	11
0x00000028	未使用	10
0x00000024	未使用	9
0x00000020	未使用	8
0x0000001C	未使用	7
0x00000018	使用错误向量 1	6
0x00000014	总线错误向量 1	5
0x00000010	MemManage向量 1	4
0x0000000C	HardFault向量 1	3
0x00000008	NMI向量 1	2
0x00000004	复位向量 1	1
0x00000000	MSP初始值	0

备注：向量的LSB必须置1以表示Thumb状态

对于正在发生的中断，若是此时又发生一个同优先级的中断，会比较其子优先级。（在已使用的寄存器里再分位抢占、子优先级）

2. 中断向量表重定位

地址=异常编号 x4

发生中断是，系统会根据中断向量表去查询这中断该跳往何处。

IRQ 是中断请求

ISR 是中断产生后跳转到的函数

可以使用向量表偏移寄存器来重定位中断向量表

在使用 VTOR 时，需要将向量表大小扩展为下一个 2 的整数次方，且新向量表的基地址必须要对齐到这个数值。

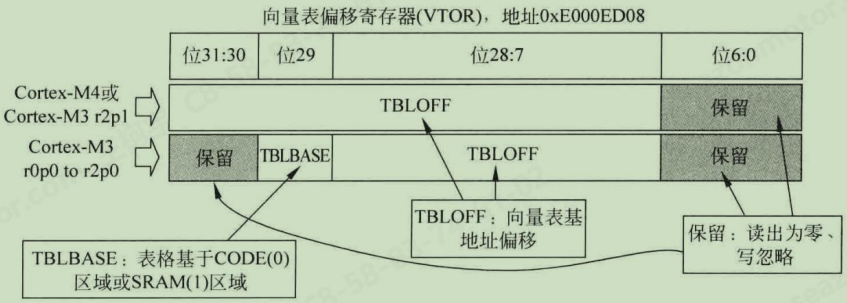


图 7.11　向量表偏移寄存器

例 1：微控制器有 32 个中断源

向量表大小为(32(用于中断)＋16(用于系统异常空间))×4(每个向量的字节数)＝192(0xC0)。将其扩大为下一个 2 的整数次方就得到 256 字节，因此，向量表的地址可被设置为 0x00000000、0x00000100 以及 0x00000200 等。

解释：（只解释 m4）

- 这个 bit0~bit6 是必须为 0；
- TBLOFF + 7 位 0：重定位这个向量表的地址；

• 为什么要重定位：中断向量表是系统编译时候自动生成的，所以裸机程序涉及不到重定位这个事，但是比如 bootloader 需要有两个程序的情况下，默认用的是 bootloader 的向量表，所以需要重定位到 app 的向量表才能正常工作。

• 在例 1 中的 16 是什么：16 就是上面的编号为 0~15 的中断向量，这个是必须的，自定义的中断向量 IRQ 就放在他们之后，因为例子中有 32 个自定义中断 IRQ 所以为（32+16）x4　4 是因为 32bit 为 4 字节。

• 为了中断的跳转速率，字节需要对齐。

何时使用中断向量偏移：

- bootloader+app
- 在 ram 中加载程序

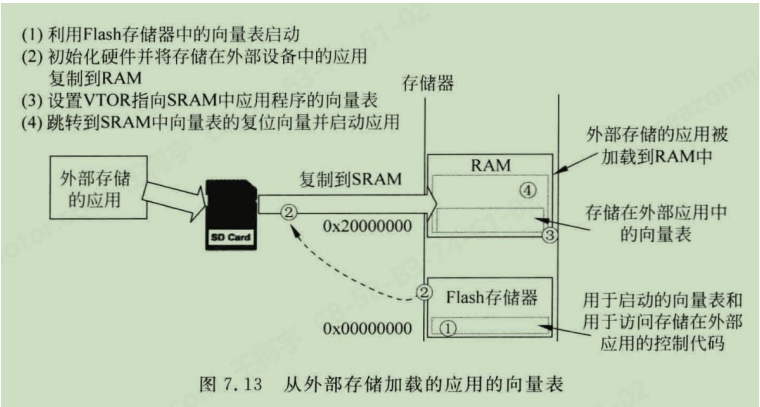


图 7.13　从外部存储加载的应用的向量表

3. 中断的执行逻辑与挂起行为

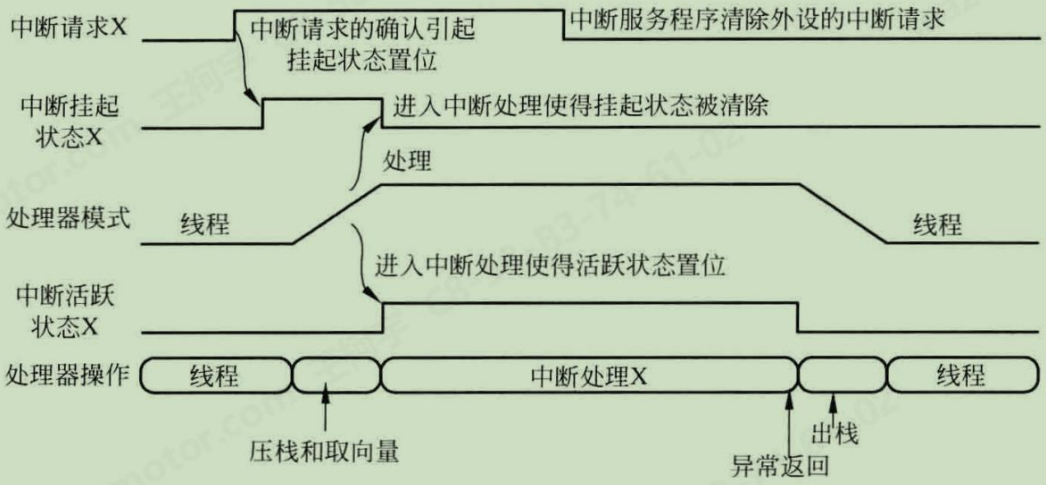


图 7.14　中断挂起和激活行为的简单情况

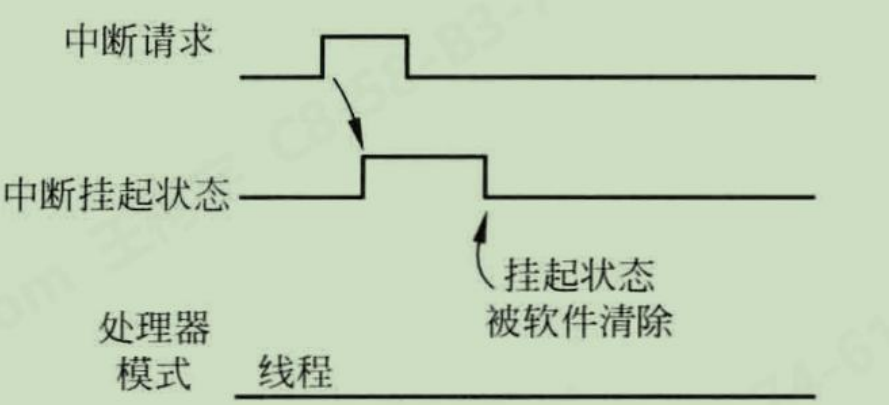


图 7.15　处理器执行操作前中断挂起被清除

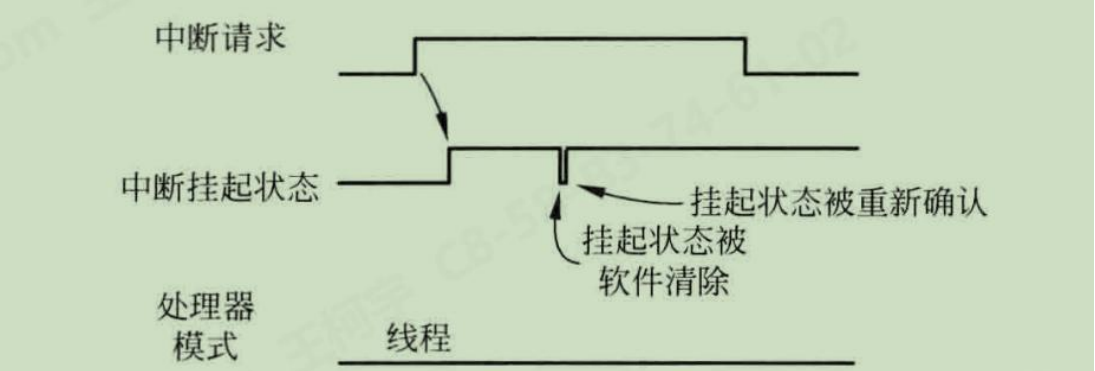


图 7.16　连续的中断请求使得中断挂起状态清除后被重新确认

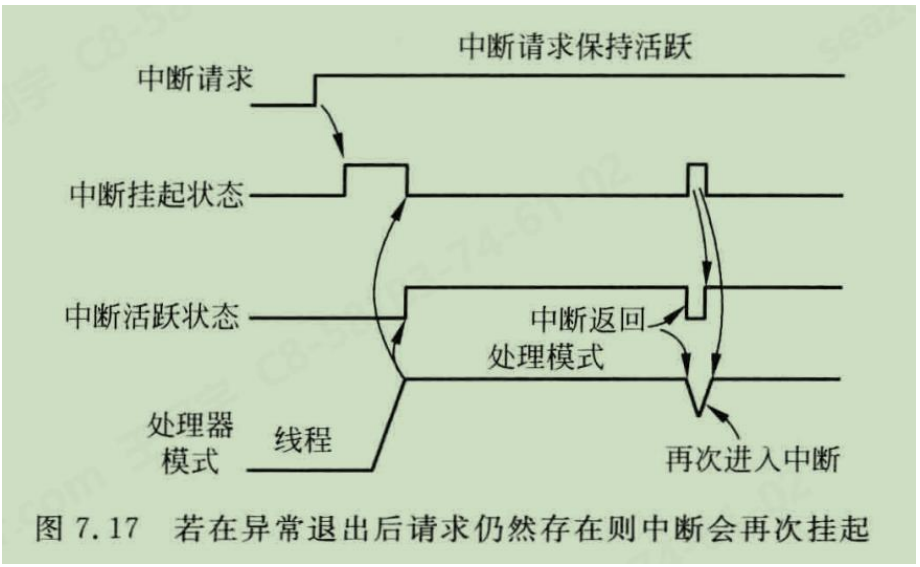


图 7.17　若在异常退出后请求仍然存在则中断会再次挂起

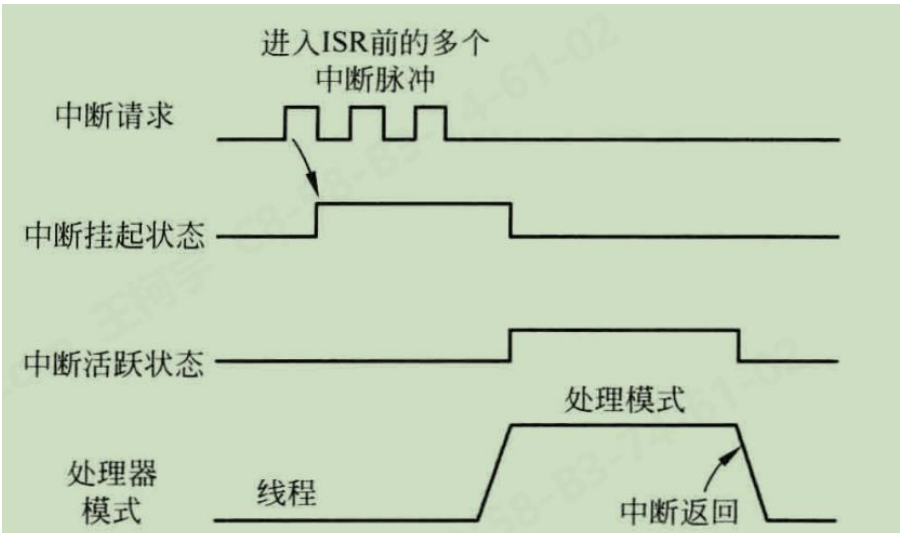


图 7.18　在处理前即使有多个脉冲，中断也只会挂起一次

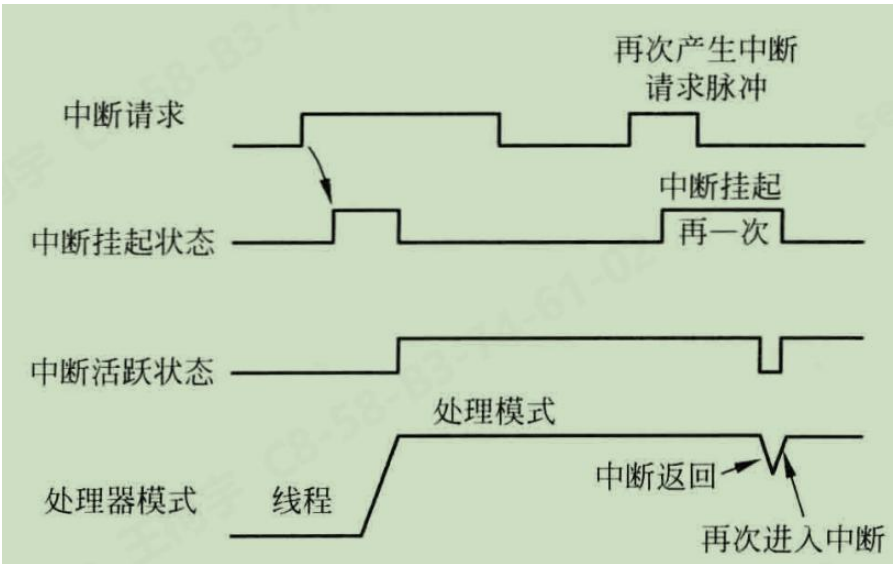


图 7.19　在异常处理执行过程中再次产生中断挂起

4. NVIC 中断寄存器细节：

地址	寄存器	CMSIS-Core 符号	功　能
0xE000E100～ 0xE000E11C	中断设置使能寄存器	NVIC->ISER［0］～ NVIC->ISER［7］	写 1 设置使能
0xE000E180～ 0xE000E19C	中断清除使能寄存器	NVIC->ICER［0］～ NVIC->ICER［7］	写 1 清除使能
0xE000E200～ 0xE000E21C	中断设置挂起寄存器	NVIC->ISPR［0］～ NVIC->ISPR［7］	写 1 设置挂起状态

地址	寄存器	CMSIS-Core 符号	功　能
0xE000E280～ 0xE000E29C	中断清除挂起寄存器	NVIC->ICPR［0］～ NVIC->ICPR［7］	写 1 清除挂起状态
0xE000E300～ 0xE000E31C	中断活跃位寄存器	NVIC->IABR［0］～ NVIC->IABR［7］	活跃状态位，只读
0xE000E400～ 0xE000E4EF	中断优先级寄存器	NVIC->IP［0］～ NVIC->IR［239］	每个中断的中断优先级（8 位宽）
0xE000EF00	软件触发中断寄存器	NVIC->STIR	写中断编号设置相应中断的挂起状态

这些都是描述#0（也就是#16 以后的外部中断 IRQ 的）

每一个【x】都是个 32bit 的寄存器，所以它可以表示 32 个外部中断，写 1 对应的中断使能，写 0 不使能

5.系统控制块 SCB

Cortex-M 内核负责“系统级异常、向量表、复位、睡眠、故障状态”等功能的一组核心寄存器

表 7.16　SCB 中的寄存器一览

地址	寄存器	CMSIS-Core 符号	功　能
0xE000ED00	CPU ID	SCB->CPUID	可用于识别处理器类型和版本的 ID 代码
0xE000ED04	中断控制和状态	SCB->ICSR	系统异常的控制和状态
0xE000ED08	向量表偏移寄存器	SCB->VTOR	使能向量表重定位到其他的地址

地址	寄存器	CMSIS-Core 符号	功　能
0xE000ED0C	应用中断/复位控制寄存器	SCB->AIRCR	优先级分组配置和自复位控制
0xE000ED10	系统控制寄存器	SCB->SCR	休眠模式和低功耗特性的配置
0xE000ED14	配置控制寄存器	SCB->CCR	高级特性的配置
0xE000ED18～0xE000ED23	系统处理优先级寄存器	SCB-> SHP［0］～SCB->SHP[11]	系统异常的优先级设置
0xE000ED24	系统处理控制和状态寄存器	SCB->SHCSR	使能错误异常和系统异常状态的控制
0xE000ED28	可配置错误状态寄存器	SCB->CFSR	引起错误异常的提示信息
0xE000ED2C	硬件错误状态寄存器	SCB->HFSR	引起硬件错误异常的提示信息
0xE000ED30	调试错误状态寄存器	SCB->DFSR	引起调试事件的提示信息
0xE000ED34	存储器管理错误寄存器	SCB->MMFAR	存储器管理错误的地址值
0xE000ED38	总线错误寄存器	SCB->BFAR	总线错误的地址值
0xE000ED3C	辅助错误状态寄存器	SCB->AFSR	设备相关错误状态的信息
0xE000ED40～0xE000ED44	处理器特性寄存器	SCB-> PFR［0］～SCB->PFR[1]	可用处理器特性的只读信息
0xE000ED48	调试特性寄存器	SCB->DFR	可用调试特性的只读信息
0xE000ED4C	辅助特性寄存器	SCB->AFR	可用辅助特性的只读信息
0xE000ED50～0xE000ED5C	存储器模块特性寄存器	SCB-> MMFR［0］～SCB->MMFR［3］	可用存储器模块特性的只读信息
0xE000ED60～0xE000ED70	指令集属性寄存器	SCB-> ISAR［0］～SCB->ISAR［4］	指令集特性的只读信息

上面是控制#16 之前的中断

这个是控制#16 之前一些系统相关的

具体每个寄存器怎么用，看手册

6.中断屏蔽相关寄存器

PRIMASK:屏蔽 NMI 与 HardFault 以外的中断

一般用在一些时许任务上，以防被中断打断

FaultMask：屏蔽 NMI 以外的中断

一般这个用不上，这个屏蔽的太狠了

BASEPRI：禁用优先级 xxx 之下的中断

7.设置中断步骤：

（1）设置优先级分组。优先级分组默认为 0（优先级寄存器中只有第 0 位用于子优先级），这一步是可选的。

（2）设置中断的优先级。中断的优先级默认为 0（最高的可编程优先级），这一步也是可选的。

（3）在 NVIC 或外设中使能中断。

下面的例子为设置中断的步骤：

```
//将优先级分组设置为 5
NVIC_SetPriorityGrouping(5);
//将 Timer0_IRQn 优先级设置为 0xC0（4 位优先级）
NVIC_SetPriority(Timer0_IRQn, 0xC); //CMSIS 函数将其移至 0xC0
//使能 NVIC 中的 Timer 0 中断
NVIC_EnableIRQ(Timer0_IRQn);
```

8.栈帧字节对齐

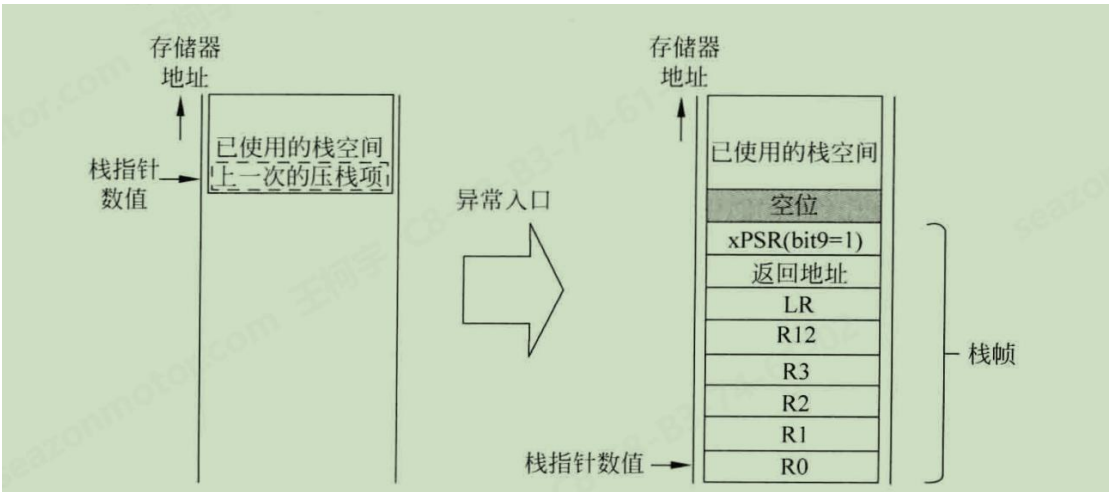


图 8.3　在需要或使能双字栈对齐时，Cortex-M3 或 Cortex-M4（无浮点单元）处理器的异常栈帧

出入栈规定要 8 字节对齐，因为系统一次搬运 32bit 也就是 4 字节，所以有的系统会采用 4 字节对其，但是这个入栈是要 8 字节对其，对于这种情况，系统硬件会自动填入 4 字节来保证 8 字节对其，并在 xPSR 寄存器的 bit9 置 1 来表示插入了 4 节，以后出栈恢复时候就会跳过这 4 字节

9. EXC_RETURN

对于普通模式　LR = 返回地址 A

```
A()
{
```

```
    B();
}
```

但是产生异常后会被硬件塞入 EXC_RETURN，这个的值根据 EXC_RETURN 寄存器来确定，代表恢复后使用什么

表 8.1 EXC_RETURN 的位域

位	描述	数 值
31:28	EXC_RETURN 指示	0xF
27:5	保留(全为 1)	0xEFFFFF(23 位都是 1)
4	栈帧类型	1(8 字)或 0(26 字)。当浮点单元不可用时总是为 1,在进入异常处理时,其会被置为 CONTROL 寄存器的 FPCA 位
3	返回模式	1(返回线程)或 0(返回处理)
2	返回栈	1(返回线程栈)或 0(返回主栈)
1	保留	0
0	保留	1

表 8.2 EXC_RETURN 的合法值

	浮点单元在中断前使用(FPCA=1)	浮点单元未在中断前使用(FPCA=0)
返回处理模式(总是使用主栈)	0xFFFFFFE1	0xFFFFFFF1
返回线程模式并在返回后使用主栈	0xFFFFFFE9	0xFFFFFFF9
返回处理模式并在返回后使用进程栈	0xFFFFFFED	0xFFFFFFFD

10.异常时压栈与入栈（RTOS 也是这种）

异常：就是#16 之前的所有都算异常，以及中断

发生异常时，并不是先压栈再去读取 ISR 去执行，而是并行的。降低中断响应延迟

压栈完成后一般为

但是并不是按顺序去压栈的，先写 PC、xPSR,然后再写别的

```
地址 N      → R0
地址 N+4    → R1
地址 N+8    → R2
地址 N+12   → R3
地址 N+16   → R12
地址 N+20   → LR
地址 N+24   → PC
地址 N+28   → xPSR
```

R0~R3、R12 是硬件自动压栈，并且位置已经提前预留好，并不

需要 PUSH 或者 POP，而是直接写入预留好的格子，

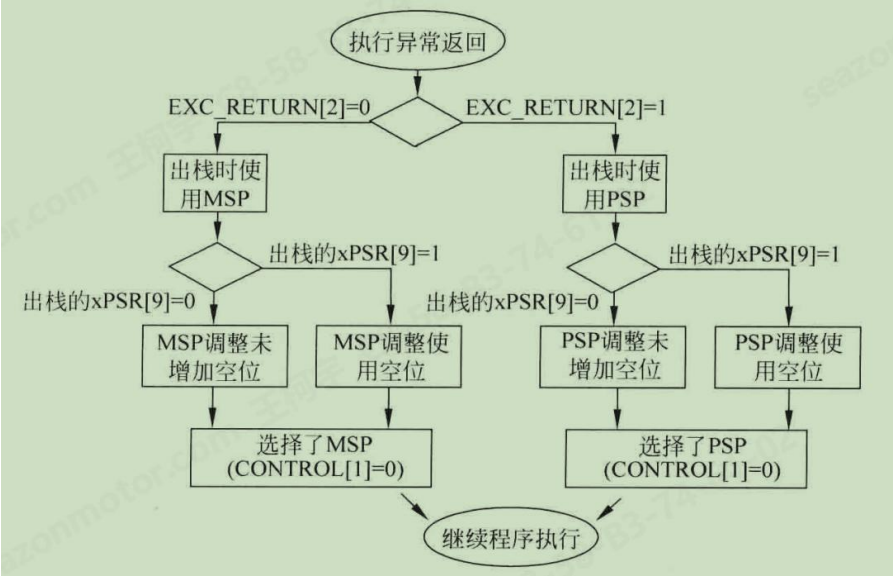
另外 R4~R11 需要的就软件压栈，不需要就不压。

对比普通的入栈，普通的需要什么压什么，没有这种固定形式。

入栈瞬间用的是什么栈指针就压入什么栈里（MSP、PSP)

好处就是可以完全恢复异常之前的工作状态，这个用的 MSP 还是 PSP 信息再

EXC_RETRUN 的 bit2 中，1 是主栈 0 是进程栈



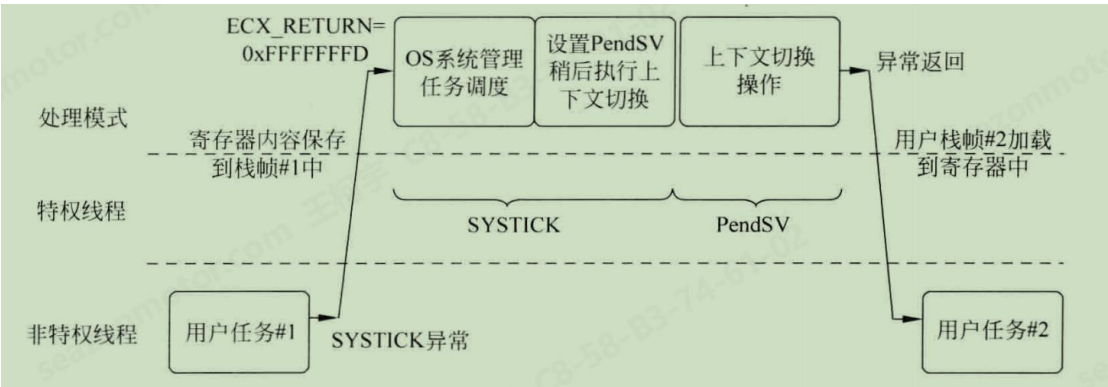
四. OS 特性

1.切换上下文

在 OS 设计中，需要在不同任务间切换，这一般被称作上下文切换，其通常在 PendSV 异常处理中执行，该异常可由 SysTick 异常触发。在上下文切换操作中需要：

- 将寄存器的当前状态保存到当前栈中
- 保存当前 PSP 数值
- 将 PSP 设置为下一个任务的上一次 SP 数值
- 恢复下一个任务的上一次的数值
- 利用异常返回切换任务

在切换时，保存当前 PSP 值（也就是 PSP 现在的地址），并不是把其中的 LR PC XPSR R0~R3 R12 保存了，而是保存了 PSP 现在指向的地址



2.SVC(常用于首任务启动)

利用汇编语言提取 SVC 编号：

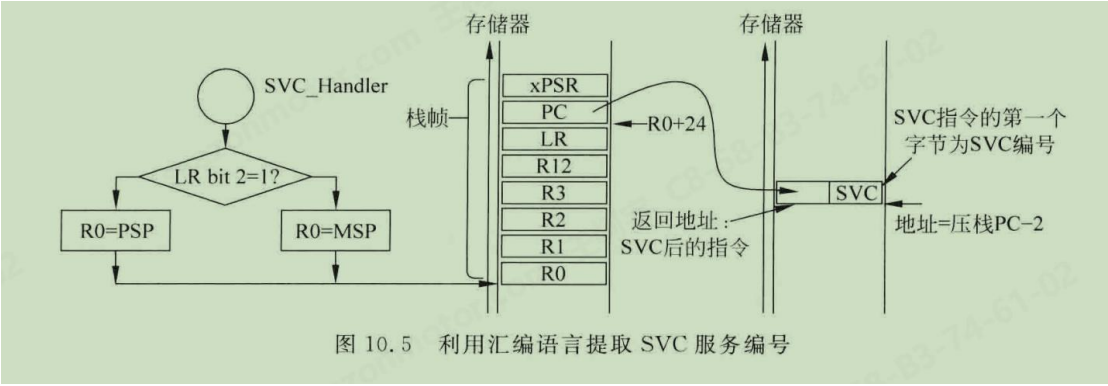


图 10.5　利用汇编语言提取 SVC 服务编号

thumb 是 16 位的（2 字节）

假如触发 SVC 0x0002

执行 SVC　0X0004

这个执行的程序的 PC-2 也就是 SVC 的地址，就能知道是 SVC#几

SVC 的参数和返回值最好通过“异常栈帧”传递，而不是依赖 Handler 执行过程中 CPU 当前的 R0~R3，因为 Handler 还可能被更高优先级中断抢占。

比如：

RO = 10 ;R1 = 20

此时执行 SVC#3

在执行的一瞬间硬件就把 RO-R3 R12 PC LR XPSR 存入栈

SVC#3 会读取栈帧来计算，假如结果 res = 123

假设不通过栈帧传递，结果直接修改当前 R0(返回值），在 SVC 结束后栈帧会返回到 r0 与 r1，会把刚刚修改的值替换成原本的 10

所以要通过栈帧传递，这个结果会直接修改栈内的 r0，在 SVC 结束后返回就好。

3.PendSV(常用于任务切换）

相较于 SVC，PendSV 不是瞬时发生的。SVC 触发后立即进入异常中断，但是 PendSV 他算是提供一个挂起态，当前面有更高优先级的事情在发生，他会等到高优先级结束后再去执行。

特点：可以设置低优先级、可以挂起

所以比较适合作为 RTOS 任务切换的中断

RTOS 任务切换本质就是保存任务 A 的 PSP→找到任务 B 的 PSP→把 PSP 切到任务 B 保存的栈位置→从 B 的栈里恢复 B 的寄存器现场

PendSV 的作用主要有两个：

第一，利用异常进入机制，让硬件自动把当前任务的 R0~R3、R12、LR、PC、xPSR 压入当前任务的 PSP 栈；

第二，进入 PendSV_Handler 后处于特权 Handler Mode，RTOS 可以继续保存 R4~R11、保存/切换 PSP，并恢复下一个任务的上下文。

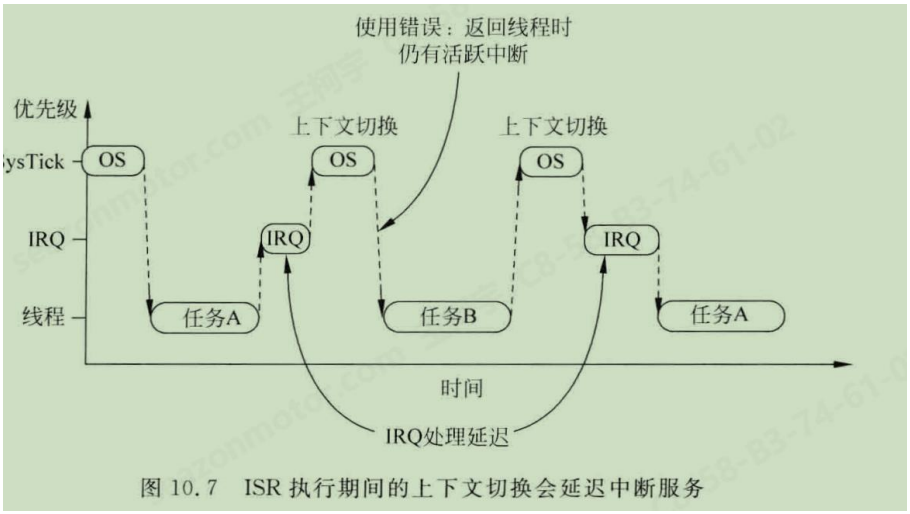


图 10.7 ISR 执行期间的上下文切换会延迟中断服务

上图为错误表示，比如在任务 A 运行时来了外部中断 IRQ，在中断进行中此时要去切换任务，会导致 IRQ 延迟，甚至严重会导致数据错误，所以此时 PendSV 进行挂起，在 IRQ 结束后执行任务切换

PendSV 任务切换图

(1) A 任务调用 SVC 进行任务切换(例如,等待一些工作完成)。
(2) OS 收到请求,准备进行上下文切换,且挂起 PendSV 异常。
(3) 当 CPU 退出 SVC 时,会立即进入 PendSV 且进行上下文切换。
(4) 当 PendSV 完成并返回线程等级时,OS 会执行 B 任务。
(5) 中断产生且进入中断处理。
(6) 在运行中断处理程序时,SYSTICK 异常(用于 OS 节拍)会产生。
(7) OS 执行重要操作,然后挂起 PendSV 异常并准备进行上下文切换。
(8) 当 SYSTICK 异常退出时,会返回到中断服务程序。

(9) 当中断服务程序结束后,PendSV 开始执行实际的上下文切换操作。
(10) 当 PendSV 完成后,程序返回到线程等级,这次它会回到任务 A 并继续执行。

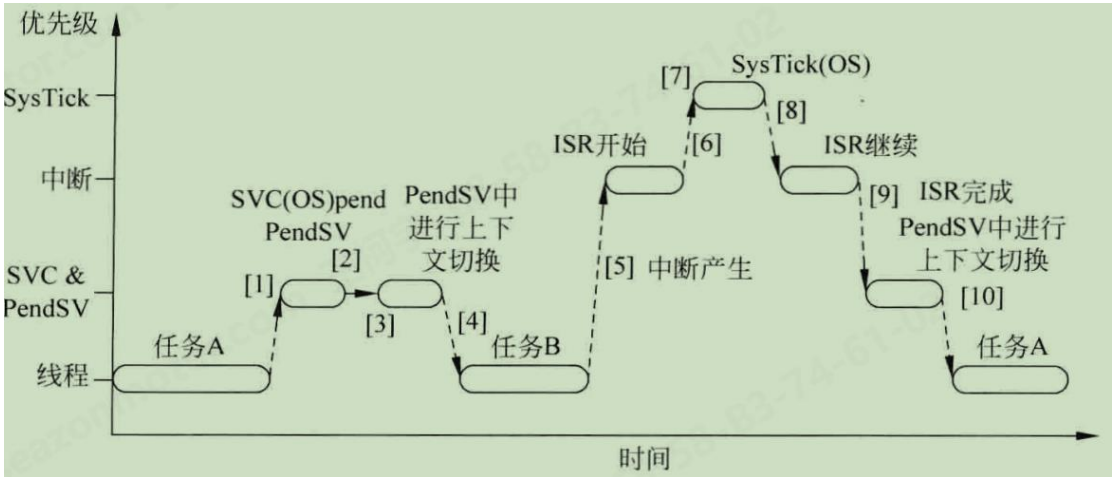


图 10.8 PendSV 上下文切换示例

完整的上下文切换:

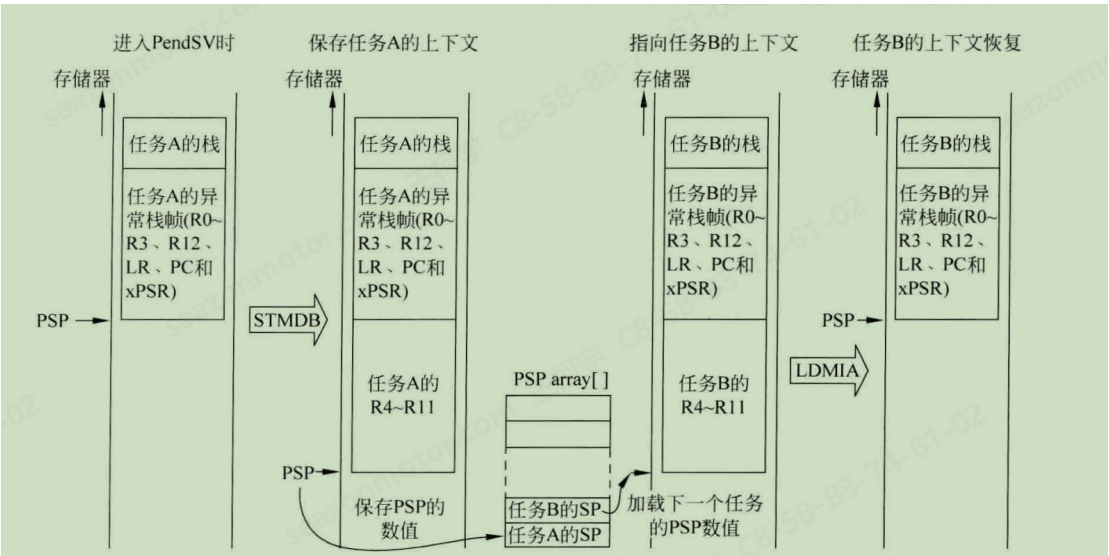


图 10.10　上下文切换

五. 嵌入式 OS

1. 中间件与模拟层

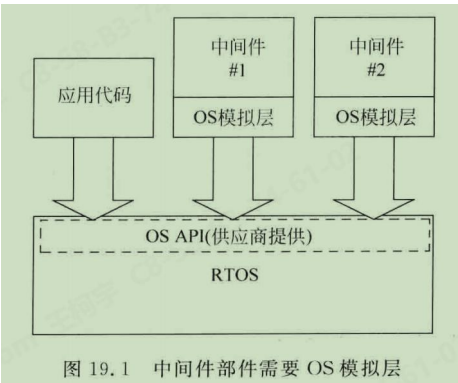


图 19.1　中间件部件需要 OS 模拟层

中间件：各个 os 自己定义的函数

模拟层：把各个中间件的函数统一为 OS 通用的

（CMSIS-RTOS)

2. 线程

在 CMSIS-RTOS 中，每个并行的程序叫线程，一个任务或者进程会有很多线程，每个线程都可以编辑优先级

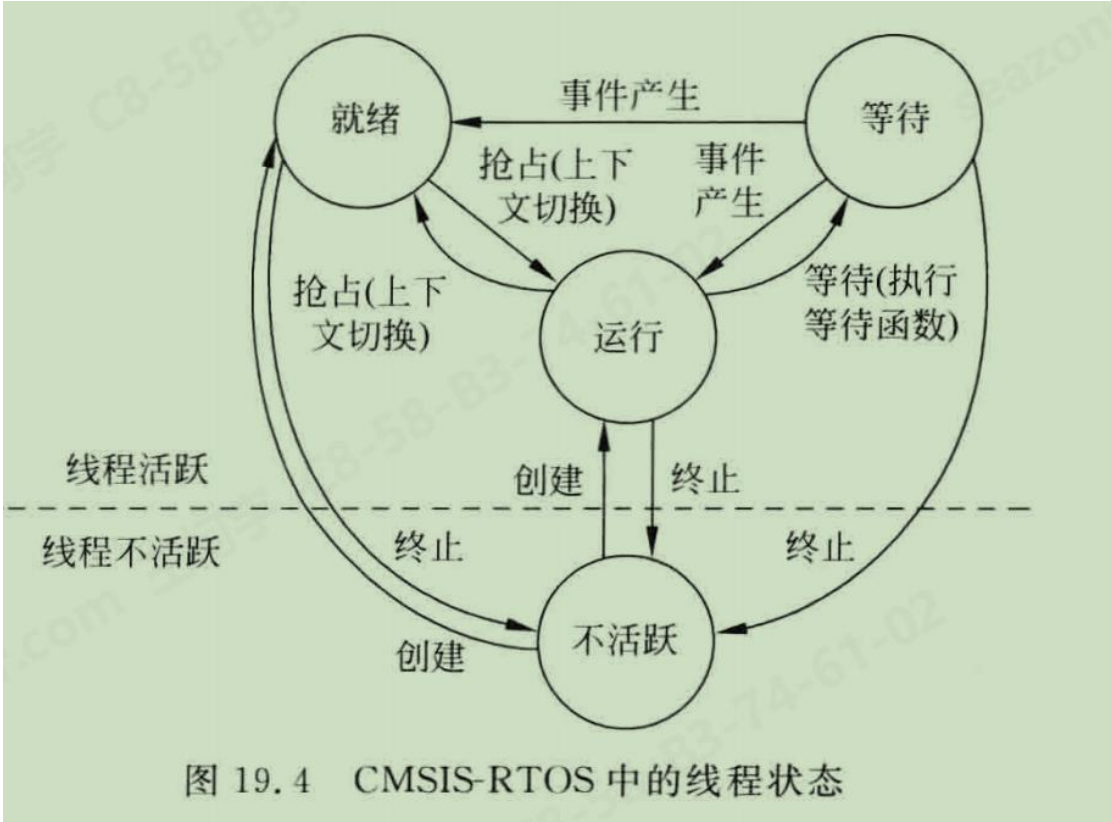


图 19.4　CMSIS-RTOS 中的线程状态

线程之间的状态转换

OS 中任务的切换：

Cortex-M 中 FreeRTOS 通常利用 SysTick 产生系统节拍，并通过 PendSV 完成真

正的上下文切换。任务运行在 Thread Mode，通常使用 PSP。进入异常时，Cortex-M 硬件自动将 R0-R3、R12、LR、PC 和 xPSR 压入当前任务的 PSP 栈。PendSV Handler 再由软件保存 R4-R11，然后将当前 PSP 保存到当前任务 TCB 的 pxTopOfStack 中。调度器切换 pxCurrentTCB 到下一个任务，从新的 TCB 中取出栈顶恢复 PSP，再恢复 R4-R11。最后通过 EXC_RETURN 异常返回，CPU 硬件自动从新任务栈恢复 R0-R3、R12、LR、PC 和 xPSR，其中 PC 决定任务从上一次停止的位置继续执行

先明确 3 个核心定义

1. 栈是什么?

栈是**单片机 SRAM（内存）里划出的一块专用区域**，本质就是一段普通内存，只是约定了 “只能从一端存取、后进先出” 的规则。

- 栈底：栈的起始边界，**固定不动**，地址最高；
- 栈顶：栈里当前最后一个有效数据的位置，**动态移动**，地址最低；
- SP（栈指针寄存器）：CPU 里专门的硬件寄存器，**永远指向栈顶。**

2. 什么叫 “向低地址增长”?

Cortex-M 内核用的是**满递减栈（Full Descending）**：

- 递减：存入新数据（入栈）时，栈顶往**地址数值更小**的方向走（向下延伸）；
- 满：SP 指针永远指向**已经存了数据的栈顶位置**，不是指向空位置。

3. 入栈 / 出栈的本质

- 入栈（PUSH）：**先把 SP 往低地址减，再把寄存器的值复制一份写到 SP 指向的内存里**；
- 出栈（POP）：**先把 SP 指向的内存数据读回寄存器，再把 SP 往高地址加。**

关键：入栈是「复制备份」，不是「剪切移动」。寄存器里的原值还在，只是后面会被 Handler 覆盖，所以先在内存里存个快照。

操作 2：再把 R1=20 压入栈（PUSH {R1}）

1. SP = SP - 4 → 变成 0x20004FF8
2. 把 R1 的值 20，写入 0x20004FF8

```
plaintext
高地址
0x20005000  ──────────  栈底
0x20004FFC  [ 10 ]
0x20004FF8  [ 20 ]    ← SP指向这里
0x20004FF4
低地址
```

你看：数据越多，SP 越往下走，地址数值越小，这就是**向低地址增长。**

操作 3：出栈弹回 R1（POP {R1}）

1. 读取 SP 指向的 0x20004FF8 的值 20，放回 R1 寄存器
2. SP = SP + 4 → 回到 0x20004FFC

```
plaintext
高地址
0x20005000  ──────────  栈底
0x20004FFC  [ 10 ]    ← SP回到这里
0x20004FF8  [ 20 ]    （这个位置的数据还在，但已经是“无效”的了，下次入栈会直接覆盖）
低地址
```

```
0x20004FF4
低地址
```

出栈入栈的详细理解

.data 与.bss：

表格

对比项	.data 段	.bss 段
存放内容	初值非 0 的全局 / 静态变量	未初始化或初值为 0 的全局 / 静态变量
是否占 Flash	是（存初值）	否（只记录大小）
是否占 SRAM	是	是
启动时操作	从 Flash 复制初值到 SRAM	批量清零整片 SRAM 区域
运行时属性	可读可写	可读可写
变量默认值	你写的初值	固定为 0

这个 Bss 批量清零是因为本来存的值就是零 sram 在起始时是随机数，不清零就会随机给

变量赋一个随机值

但是对于局部变量 他是存放在栈中，调用函数时创建 retrun 后释放