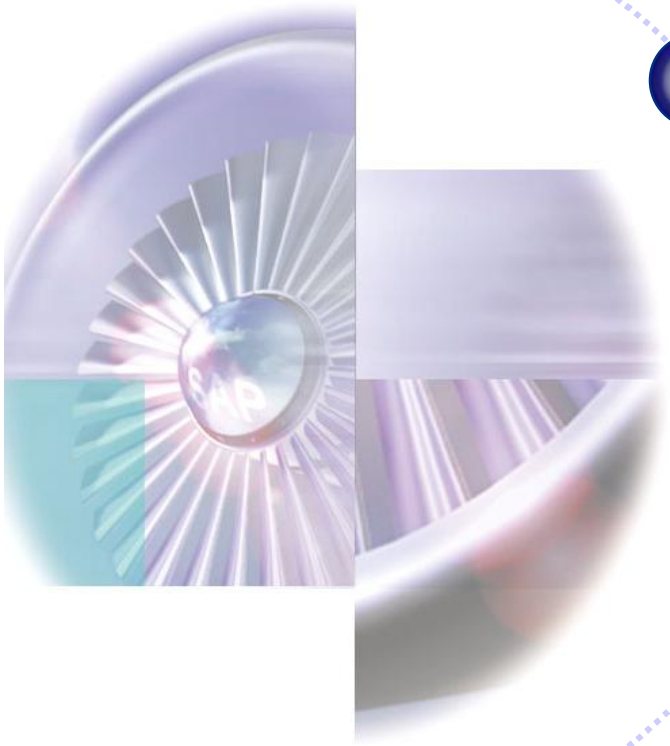
嵌入式系统设计导论

——基于32位微处理器与实时操作系统

第六讲系统初始化分析与mC/OS-Ⅱ移植

北京航空航天大学
机器人研究所
魏洪兴

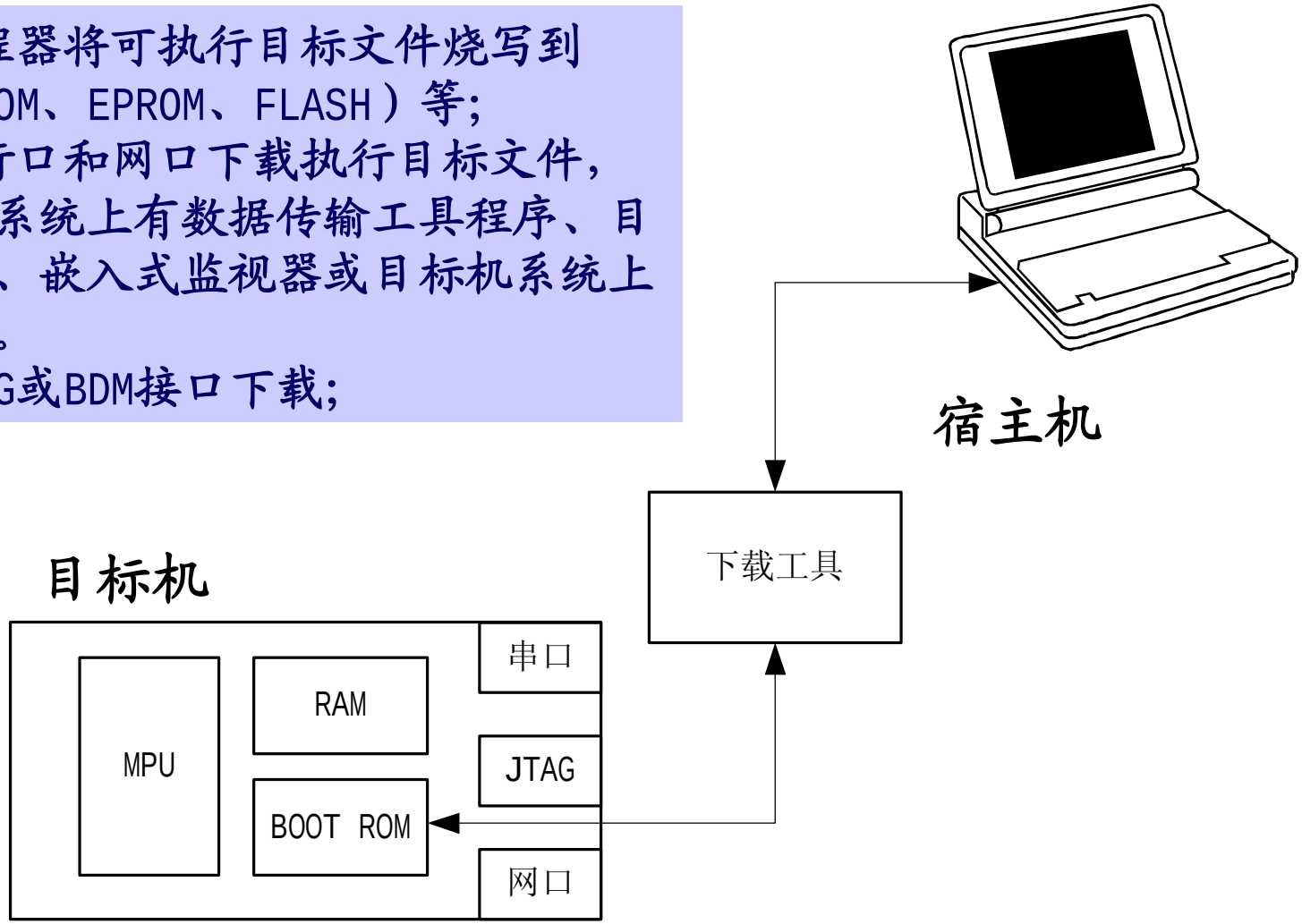
本节提要



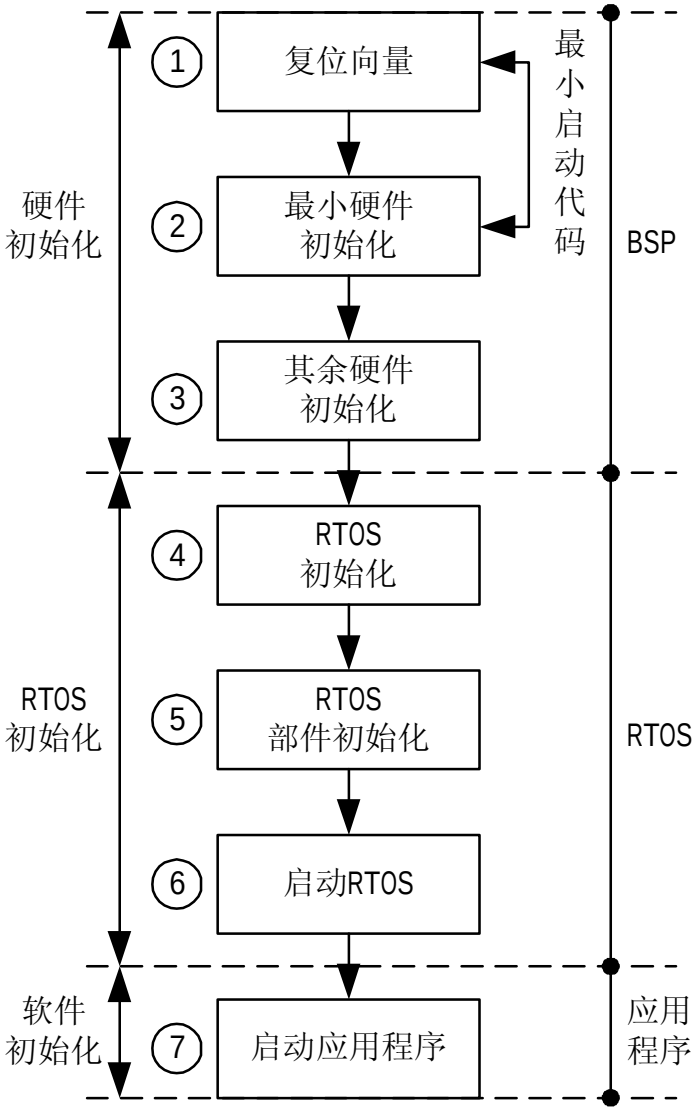
1. 嵌入式系统的初始化
2. mC/OS-Ⅱ的移植

初始化程序的下载执行

1）通过编程器将可执行目标文件烧写到BootROM（ROM、EPROM、FLASH）等;
2）通过串行口和网口下载执行目标文件，要求宿主机系统上有数据传输工具程序、目标机装载器、嵌入式监视器或目标机系统上的调试代理。
3）通过JTAG或BDM接口下载;

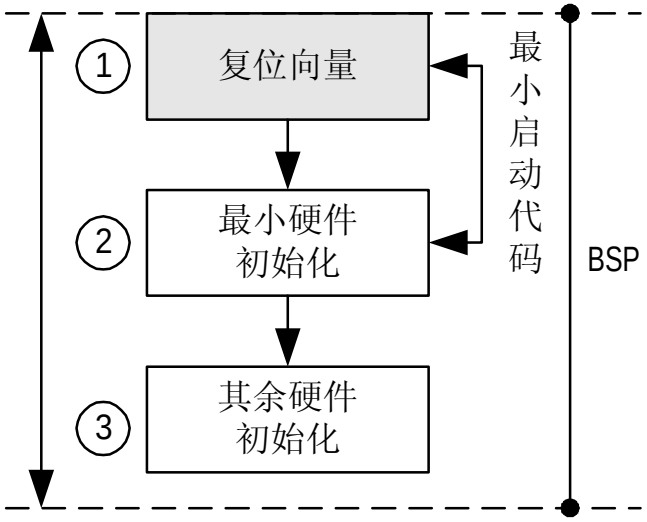


嵌入式系统的初始化过程



嵌入式系统的初始化过程

硬件初始化阶段

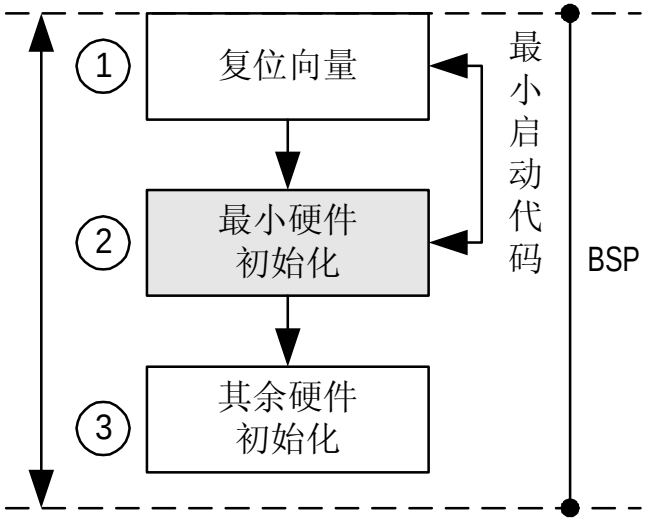


1、复位向量

```
ENTRY
    b ResetHandler  ;for debug
    b HandlerUndef  ;handlerUndef
    b HandlerSWI    ;SWI interrupt handler
    b HandlerPabort ;handlerPAbort
    b HandlerDabort ;handlerDAbort
    b .             ;handlerReserved
    b HandlerIRQ
    b HandlerFIQ
```

嵌入式系统的初始化过程（2）

硬件初始化阶段



2、最小硬件初始化

1）设置适当的寄存器，使嵌入式处理器处于一个已知的状态：

- 获得CPU的类型；
- 获得或设置CPU的时钟频率。

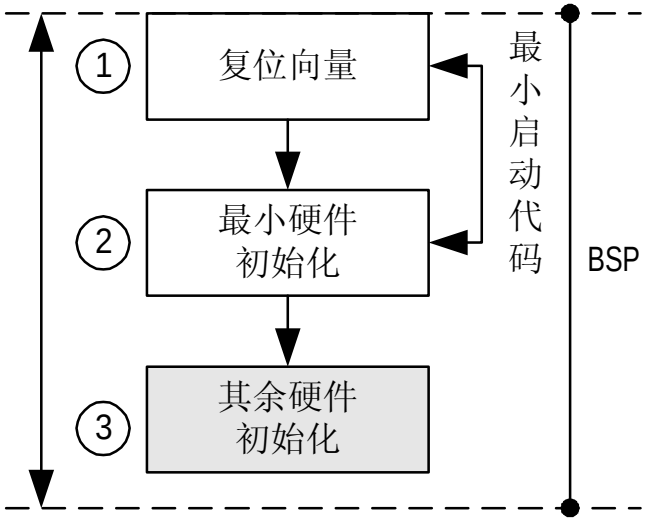
2）禁止中断和高速缓存

3）初始化内存控制器、内存芯片和高速缓存单元，包括：

- 得到内存的开始地址；
- 得到内存的大小；
- 如果有要求，则还需要进行主存测试；

嵌入式系统的初始化过程（3）

硬件初始化阶段



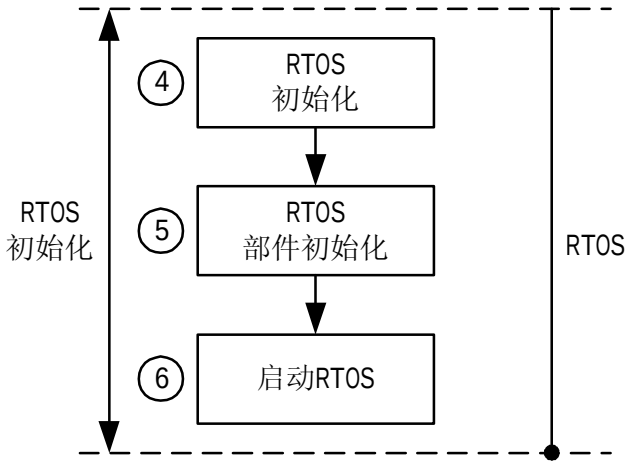
3、其余硬件初始化

1）引导代码调用合适的函数对目标机系统上的全部硬件部件进行初始化，包括：

- 建立执行处理程序
- 初始化中断处理程序
- 初始化总线接口
- 初始化板级外设得到内存的开始地址；

嵌入式系统的初始化过程（4）

RTOS初始化阶段



4、RTOS初始化

1）RTOS初始化

2）RTOS对象和服务初始化

- 任务
- 信号量
- 定时器
- 中断
- 内存管理

3）RTOS任务堆栈初始化

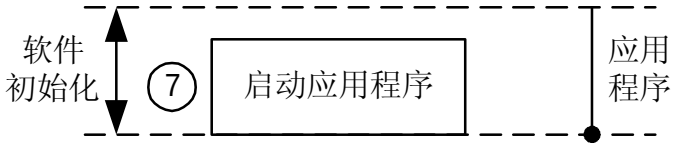
4）RTOS扩展部件初始化

5）启动RTOS

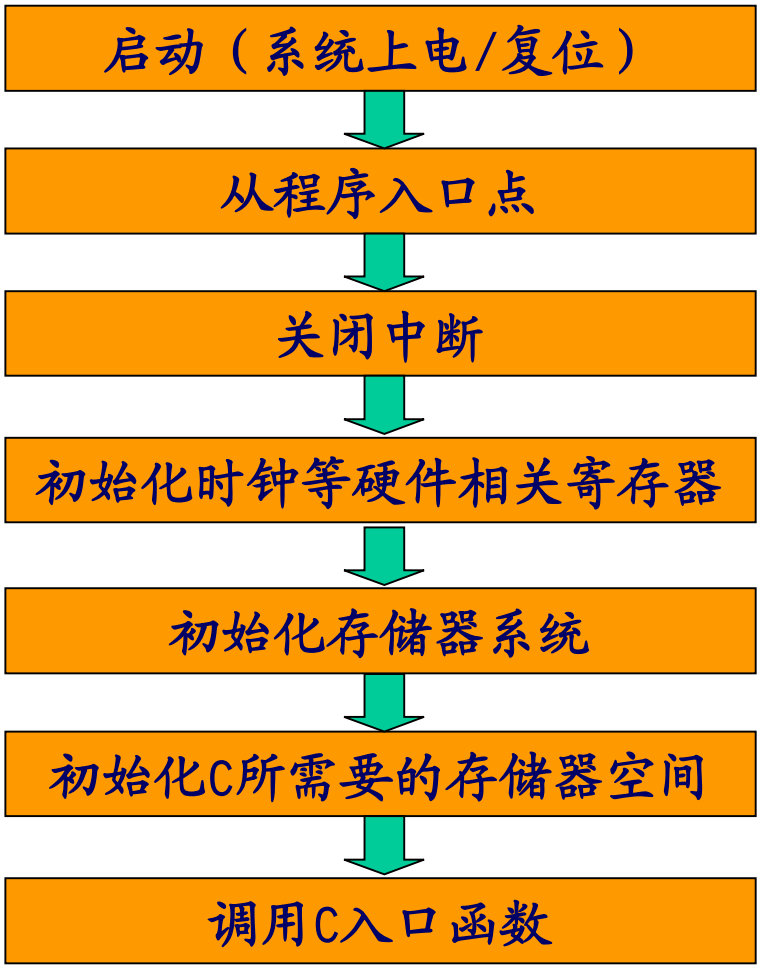
嵌入式系统的初始化过程（5）

4、应用程序初始化

应用程序初始化阶段



ARM7TDMI系统初始化的一般过程



一、设置程序入口指针

- 上电复位后直接到程序入口点执行，入口点一般为一个跳转表，跳转到复位处理程序处开始执行ARM7TDMI系统的初始化;

- 启动程序首先必须定义入中指针，而且整个应用程序只有一个入口指针

例: AREA Boot,CODE,READONLY

ENTRY /*设置程序入口指针*/

二、设置中断向量

- ARM要求中断向量必须设置在从OX00000000地址开始，连续8*4字节的地址空间；
- 向量表包含一系列跳转指令，跳转到相应的中断服务程序；
- 对各未用中断，使其指向一个含返回指令的哑函数，以防止错误中断引起系统的混乱；

中断向量表

0x1C	FIQ	外部快速中断
0x18	IRQ	一般外部中断
0x14	(Reserved)	保留
0x10	Data Abort	数据异常
0x0C	Frefetch Abort	预取指异常
0x08	Software int	软件中断
0x04	Undef	未定义指令中断
0x00	Reset	复位中断

中断向量表的程序

```
AREA Boot,CODE,READONLY
ENTRY
B  Reset_handler
B  Undef_Handler
B  SWI_Handler
B  PreAbort_Handler
B  .                    ;for reserved interrupt,stop here
B  IRQ_handler
B  FIQ_handler
```

三、初始化时钟和设置相关的寄存器

- 通过设置时钟控制器来确定CPU的工作频率，设置中断控制寄存器屏蔽中断

四、初始化存储器系统

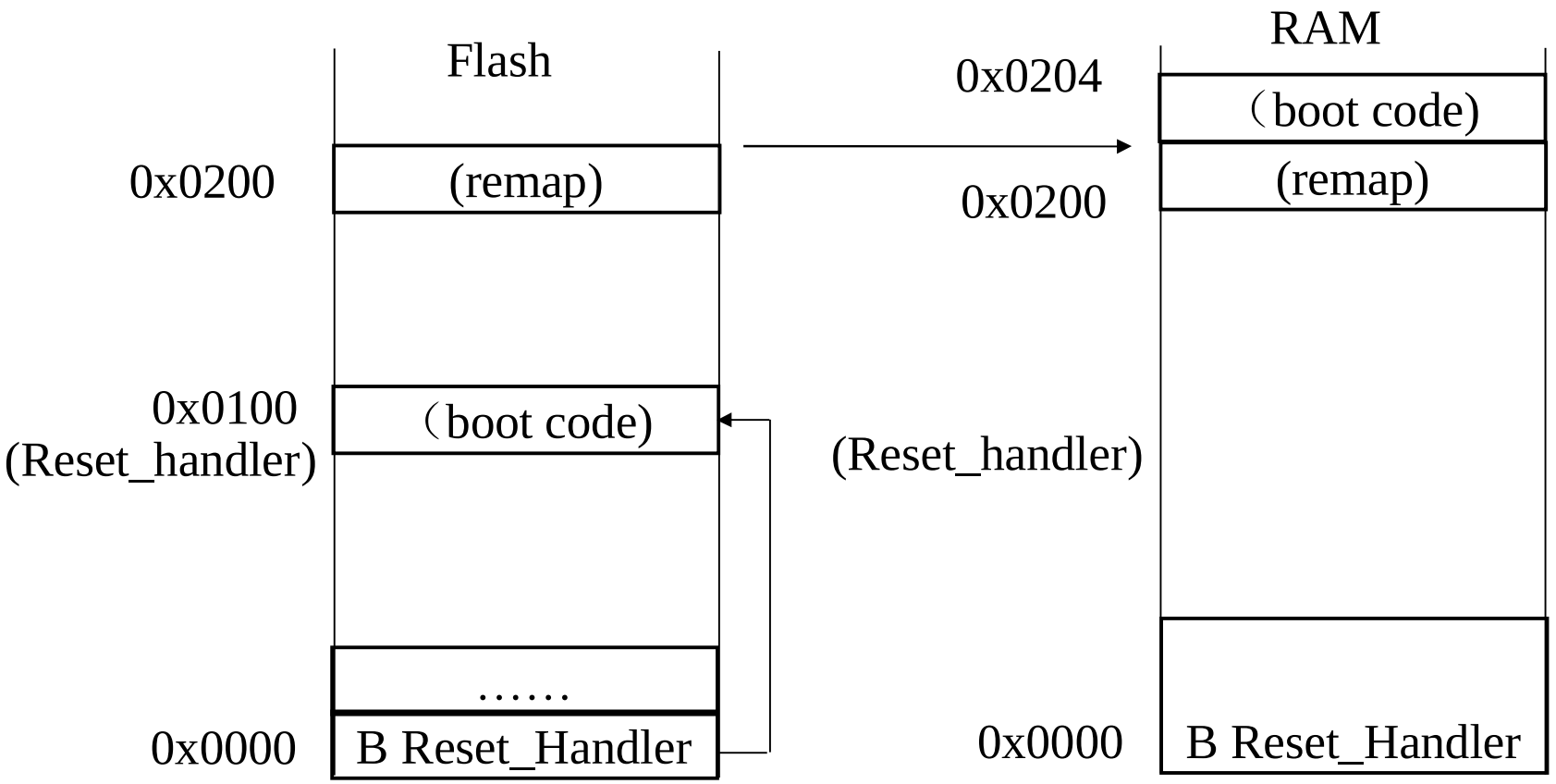
存储器类型和时序配置(参考芯片手册，设置与内存映射相关的寄存器)

- 一个复杂的系统可能存在多种存储器类型的接口，需要根据实际的系统设计对此加以正确配置。对同一种存储器类型来说，也因为访问速度的差异，需要不同的时序设置。
- 通常Flash 和SRAM 同属于静态存储器类型，可以合用同一个存储器端口;
- 而DRAM 因为动态刷新和地址线复用等特性，通常配有专用的存储器端口。
- 存储器端口的接口时序优化是非常重要的，影响到整个系统的性能。因为一般系统运行的速度瓶颈都存在于存储器访问，所以存储器访问时序应尽可能地快；但同时又要考虑由此带来的稳定性问题。只有根据具体选定的芯片，进行多次的测试之后，才能确定最佳的时序配置。

存储器地址分布

- 有些系统具有非常灵活的存储器地址分配特性，进行存储器初始化设计的时候一定要根据应用程序的具体要求来完成地址分配。
- 一种典型的情况是启动ROM 的地址重映射（remap）。当一个系统上电后程序将自动从0 地址处开始执行，因此在系统的初始状态，必须保证在0 地址处存在正确的代码，即要求0 地址开始处的存储器是非易性的ROM 或Flash 等。但是因为ROM 或Flash 的访问速度相对较慢，每次中断发生后都要从读取ROM 或Flash 上面的向量表开始，影响了中断响应速度。因此有的系统便提供一种灵活的地址重映射方法，可以把0 地址重新指向到RAM中去。在这种地址映射的变化过程当中，程序员需要仔细考虑的是程序的执行流程不能被这种变化所打断。

ROM地址的重映射



ROM地址重映射的实现

为保证重映射之后提供正确的中断入口地址，在重映射之前就必须把中断和异常向量表拷贝到内部RAM中。其程序实现如下：

```
mov    r8,#RAM_BASE_BOOT      //RAM_BASE_BOOT是重映射前内部RAM区地址
add    r9, pc,#-(8+.-VectorTable)  //VectorTale是异常向量表入口
ldmia  r9!, {r0-r7}        //读8个异常向量
stmia  r8!, {r0-r7}        //保存8个异常向量到RAM区
ldmia  r9!, {r0-r4}        //读5个异常处理程序绝对地址
stmia  r8!, {r0-r4}        //保存5个异常处理程序绝对地址到RAM区
```

五、初始化堆栈

- ARM处理器有好几种运行状态（模式），各种状态都需要有自己的堆栈，所以需要分别为这些堆栈分配空间并设置好各自的堆栈指针
- 每一种状态的堆栈指针寄存器（SP）都是独立的（System 和User 模式使用相同的SP 寄存器）。因此对程序中需要用到的每一种模式都要给SP 寄存器定义一个堆栈地址。方法是改变状态寄存器CPSR内的状态位，使处理器切换到不同的状态，然后给SP 赋值。（意不要切换到User模式进行User 模式的堆栈设置，因为进入User 模式后就不能再操作CPSR 回到别的模式了。可能会对接下去的程序执行造成影响。）
- 一般堆栈的大小要根据需要而定，但是要尽可能给堆栈分配快速和高带宽的存储器。堆栈性能的提高对系统整体性能的影响是非常明显的。

堆栈初始化代码示例

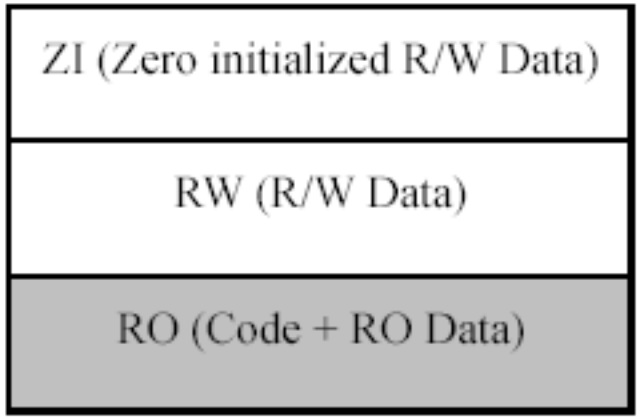
```
MRS R0, CPSR                ; CPSR -> R0
BIC R0, R0, #MODEMASK       ; 安全起见，屏蔽模式位以外的其它位
ORR R1, R0, #IRQMODE        ; 把设置模式位设置成需要的模式（IRQ）
MSR CPSR_cxsf, R1           ; 转到IRQ 模式
LDR SP, =UndefStack         ; 设置SP_irq

ORR R1,R0,#FIQMODE
MSR CPSR_cxsf, R1           ; FIQMode
LDR SP, =FIQStack

ORR R1, R0, #SVCMODE
MSR CPSR_cxsf, R1           ; SVCMode
LDR SP, =SVCStack
```

链接器产生的符号表

- 符号由链接器自动产生，只读段（read-only RO）就是代码段，读写段（read-write RW）是已经初始化的全局变量，而零初始化段（zero-initialized section ZI）中存放未初始化的全局变量；

符号表示	意义
Image$$RO$$ Base	只读段的起始地址
Image$$RO$$Limit	只读段结束的后首地址
Image$$RW$$Base	读写段的起始地址
Image$$RW$$Limit	读写段结束的后首地址
Image$$ZI$$ Base	零初始化段的起始地址
Image$$ZI$$Limit	零初始化段结束的后首地址

六、初始化应用程序执行环境



只定义了变量名的全局变量

定义时带初始值的全局变量

编译结果

- 映像一开始总是存储在ROM/Flash 里面的，其RO 部分既可以在ROM/Flash里面执行，也可以转移到速度更快的RAM 中去；而RW 和ZI 这两部分必须是需要转移到可写的RAM 里去的。所谓应用程序执行环境的初始化，就是完成必要的从ROM 到RAM 的数据传输和内容清零。

六、初始化C环境

- 在目标文件中，代码、数据放在不同的段中。源文件编译链接生成含.data、.text段的目标文件，且链接器生成的.data段是以系统RAM为参考地址

- 故在系统启动时需要拷贝ROM或FLASH中的.data段到RAM，以完成对RAM的初始化。在初始化期间应将系统需要读写的数据和变量从ROM拷贝到RAM里运行

初始化C环境（2）

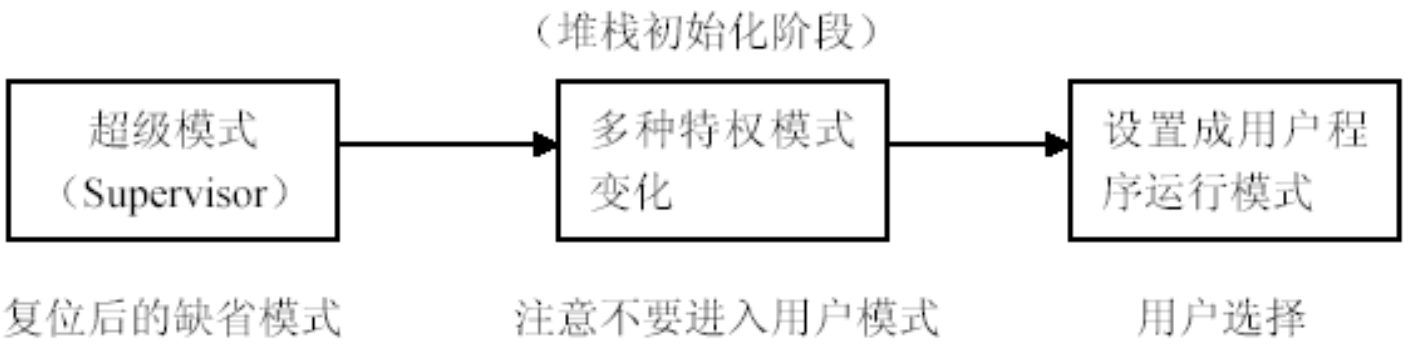
- C环境初始化，就是利用上述符号初始化RW和ZI段以使后面使用的全局变量的C程序正常运行;
- 这里有两个循环，第一个循环把预初始化的数据段RW（位于代码段的后面）复制到RAM中，另一个循环把未初始化的数据段ZI初始化为0，也就是实现把从ROM中的.data段拷贝到RAM，对ZI段内的数据初始化为0，以完成对C环境的实始化;

初始化C环境（3）

```
;*****************************************************************
;*     初始化 C 环境          *
;*****************************************************************
    LDR     r0, =|Image$$RO$$Limit|  ; 得到 ROM 中 DATA 段的起始位置载入到 r0
    LDR     r1, =|Image$$RW$$Base|   ;  RAM 区的起始位置载入到 r1
    LDR     r3, =|Image$$ZI$$Base|   ;ZI 段的起始位置载入到 r3
    ; ZI 段的起始地址>=初始化 DATA 段的顶端
    CMP     r0, r1      ; 检查两者是否相同，若相同表示 DATA 段已经在 RAM 区
    BEQ     %F1
0
//复制初始化的 DATA 到 RAM
    CMP     r1, r3      ;
    LDRCC    r2, [r0], #4      ;--> LDRCC r2, [r0] + ADD r0, r0, #4
    STRCC    r2, [r1], #4      ;--> STRCC r2, [r1] + ADD r1, r1, #4
    BCC     %B0       ;循环直到初始化 DATA 段复制完成，r1 到达 ZI 段的起始位置 r3
1
    LDR     r1, =|Image$$ZI$$Limit| ; //r1 等于 ZI 段结束地址
    MOV     r2, #0
2
//对 ZI 段（未初始化的 DATA）初始化为 0
    CMP     r3, r1      ;判断是否到达 ZI 段结束地址
    STRCC    r2, [r3], #4 ;
    BCC     %B2
```

改变处理器模式

- 除用户模式以外，其他6 种模式都是特权模式。因为在初始化过程中许多操作需要在特权模式下才能进行（比如CPSR 的修改），所以要特别注意不能过早地进入用户模式。一般地，在初始化过程中会经历以下一些模式变化：



七、呼叫C程序

- 对main函数的调用进入uc/OS的入口，通过这个入口就进入uC/OS的主函数，启动对uC/OS的初始化

- 例

```
IMPORT Main
b Main                  ;C Entry
```

uC/OS系统的初始化

- 完成了前面的硬件初始化和运行环境的相关设置后，进入Main()， Main()是uC/OS的入口函数，启动对uC/OS的初始化

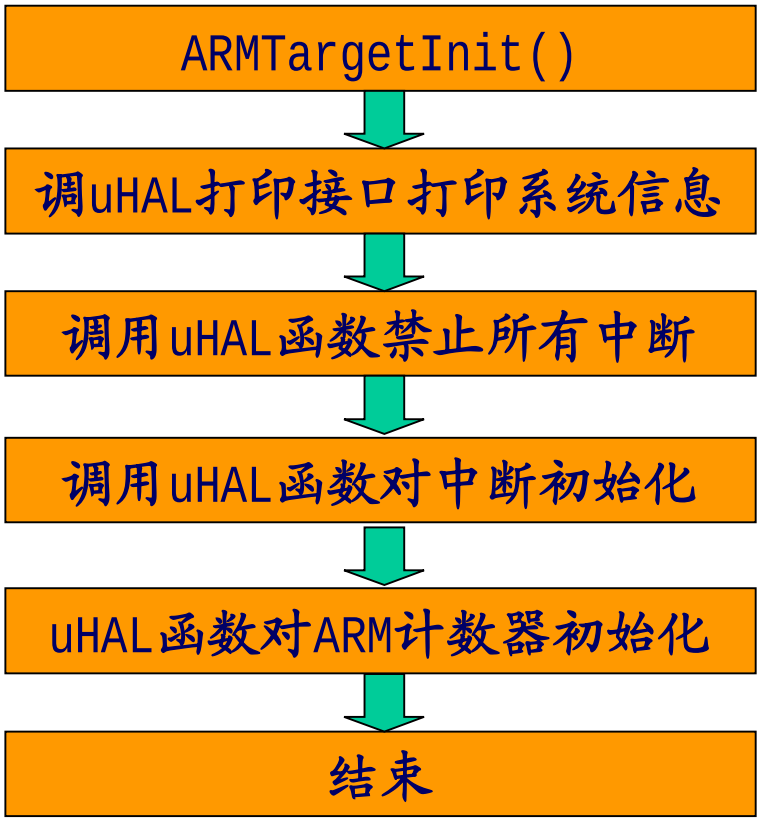
ARM的硬件抽象层——uHALuC/OS

- ARM公司为操作系统的开发提供了一个硬件抽象层HAL，称为uHAL；
- 从结构上看，uHAL是一组库程序，需要说明的是，uHAL并不是专门为uC/OS准备的，甚至也不是专为操作系统内核准备的；
- uHAL只是个针对ARM核的函数库；
- uC/OS是建立在uHAL的基础之上的；

uC/OS系统的初始化（2）

```
void main (void)
{
  ..................
    ARMTargetInit();初始化系统的硬件;
    OSInit();
.....................
    任务的建立,消息机制的建立;
    ARMTargetStart();
    OSStart();
}
```

ARMTargetInit()函数结构



uHAL的功能

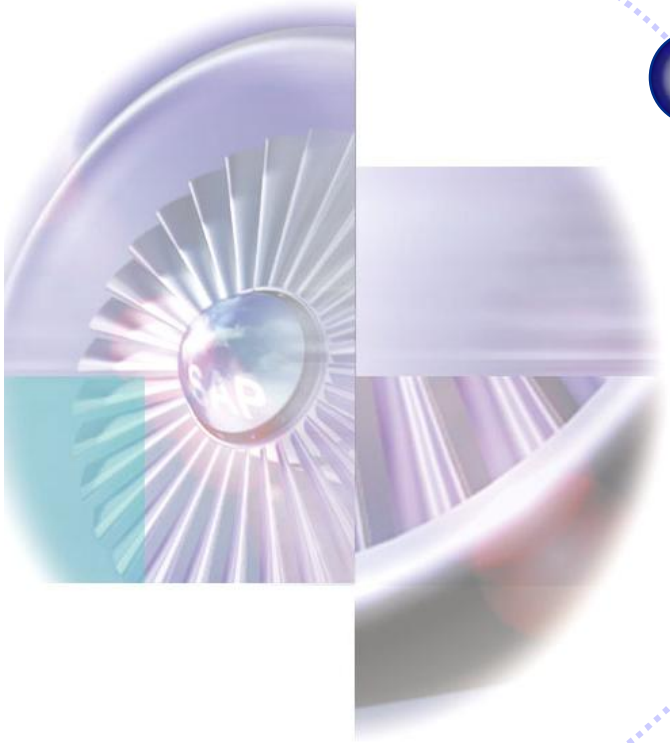
- uHAL的作用之一是在操作系统本身进入正常运行之前，为系统提供基本的输入输出手段，例如uHALr_printf()等;
- uHAL还要为操作系统的运行准备一个基本的运行环境，具体包括下列各种初始化:
  - 通过uHAL_ResetMMU()，将MMU设置在一个确定的初始状态;
  - 通过ARMDisable()关闭中断;
  - 通过uHAL_InitInterrupts()设置中断向量处理程序;
  - 通过uHAL_InitTimer()对系统使用的计数器进行初始化

ARMTargetStart()的分析

- 创建了任务之后，ARMTargetStart()调用uHALr_InstallSystemTimer()创建一个系统时钟，为时钟中断做好准备;

本节提要

1. 嵌入式系统的初始化
2. mC/OS-Ⅱ的移植



操作系统移植的概念

- 所谓操作系统的移植，是指使一个实时操作系统能够在某个微处理器平台上运行。
- mCOS-II的主要代码都是由标准的C语言写成的，移植方便。
- 移植的主要工作是修改部分与处理器硬件相关的代码。

移植的层次

操作系统的移植大体可以分为两个层次：

- 跨体系结构的移植
- 针对特定处理器的移植

移植μCOS-II满足的条件

- 处理器的C编译器能产生可重入代码
- 在程序中可以打开或者关闭中断
- 处理器支持中断，并且能产生定时中断（通常在10—100Hz之间）
- 处理器支持能够容纳一定量数据的硬件堆栈
- 处理器有将堆栈指针和其他CPU寄存器存储和读出到堆栈（或者内存）的指令

什么是可重入代码

- 可重入的代码指的是一段代码（比如：一个函数）可以被多个任务同时调用，而不必担心会破坏数据。

- 也就是说，可重入型函数在任何时候都可以被中断执行，过一段时间以后又可以继续运行，而不会因为在函数中断的时候被其他的任务重新调用，影响函数中的数据。

可重入代码举例

程序1：可重入型函数

```
void swap(int *x, int *y)
{
        int temp;
        temp=*x;
        *x=*y;
        *y=temp;
}
```

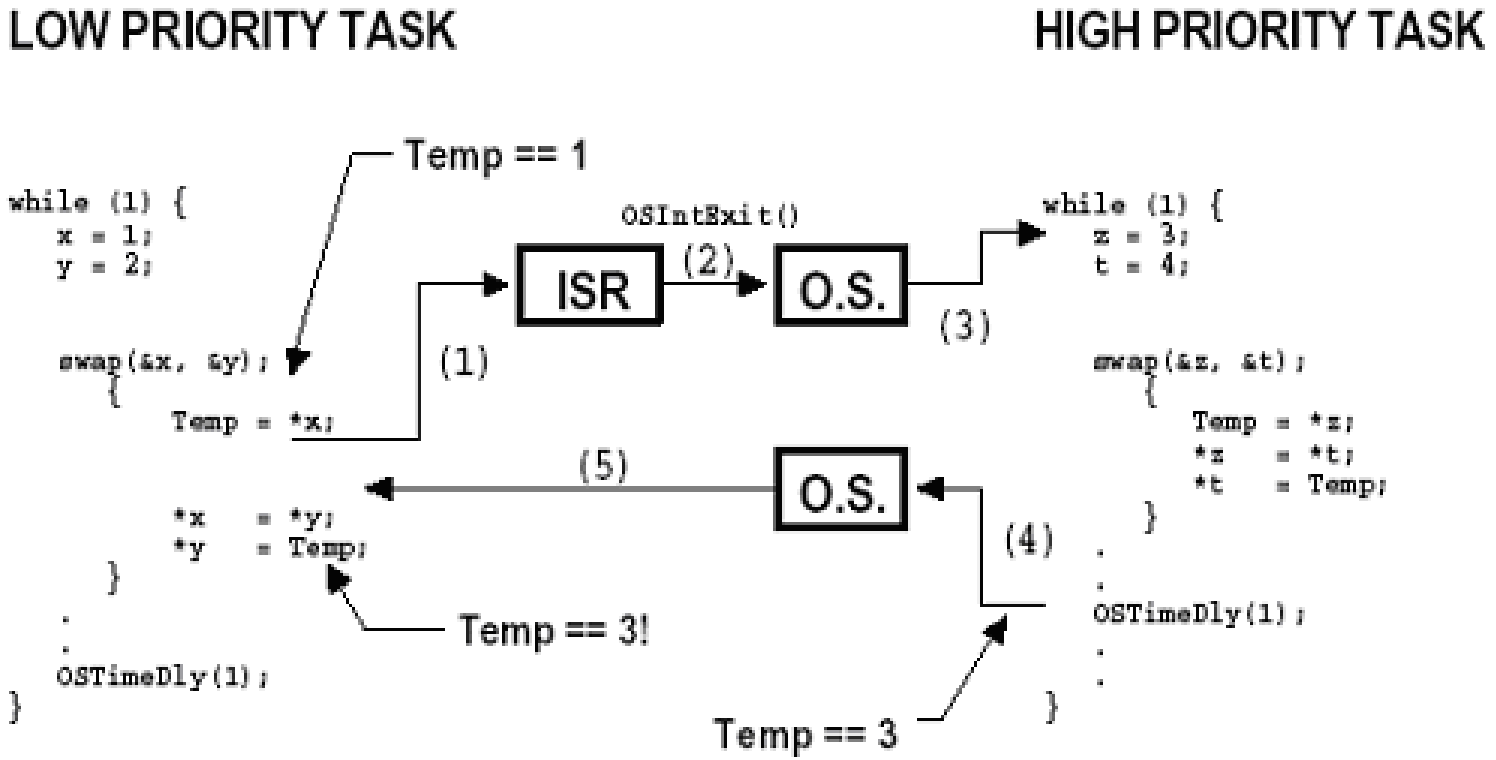
非可重入代码举例

程序2：非可重入型函数

```
int temp;
void swap(int *x, int *y)
{
        temp=*x;
        *x=*y;
        *y=temp;
}
```

返回

不可重入函数被中断破坏



如何使函数具有可重入性

使Swap()函数具有可重入性:

- 把Temp定义为局部变量;
- 调用Swap()函数之前关中断，调动后再开中断;
- 用信号量禁止该函数在使用过程中被再次调用;

打开/关闭中断

在mCOS-II中，可以通过:

OS_ENTER_CRITICAL()

OS_EXIT_CRITICAL()

宏来控制系统关闭或者打开中断。这需要处理器的支持。

在ARM7TDMI的处理器上，可以设置相应的寄存器来关闭或者打开系统的所有中断。

处理器支持中断并且能产生定时中断

- mCOS-II是通过处理器产生的定时器的中断来实现多任务之间的调度的。
- ARM7TDMI的处理器上可以产生定时器中断。

处理器支持硬件堆栈

- mCOS-II进行任务调度的时候，会把当前任务的CPU寄存器存放到此任务的堆栈中，然后，再从另一个任务的堆栈中恢复原来的工作寄存器，继续运行另一个任务。所以，寄存器的入栈和出栈是mCOS-II多任务调度的基础。
- ARM7处理器中有专门的指令处理堆栈，可以灵活的使用堆栈。

移植对开发工具的要求

- 移植mCOS-II需要一个标准的C交叉编译器;
- 由于移植时需要对CPU的寄存器进行操作，所以需要C交叉编译器能够支持汇编语言程序;
- 嵌入式C编译器一般都包括汇编器、链接器和定位器。链接器是用来将不同的模块（编译或汇编过的文件）链接成目标文件；定位器则允许将代码和数据放置在目标处理器的指定内存空间中;

移植uCOS-II要点(1)

开关中断的方式。推荐使用method3

```
{
#if OS_CRITICAL_METHOD == 3
  OS_CPU_SR  cpu_sr;
#endif
…
   OS_ENTER_CRITICAL();
…
   OS_EXIT_CRITICAL();
}
```

使用method3方式的开关中断

```
#define  OS_ENTER_CRITICAL()
        { cpu_sr = INTS_OFF(); }
#define  OS_EXIT_CRITICAL()
        { if(cpu_sr == 0) INTS_ON(); }
```

ARM的中断模式

- 设备的中断在ARM中被映射到了两个异常中——FIQ和IRQ
- 通过控制CPSR中的对应数据位，可以开启或者关闭中断

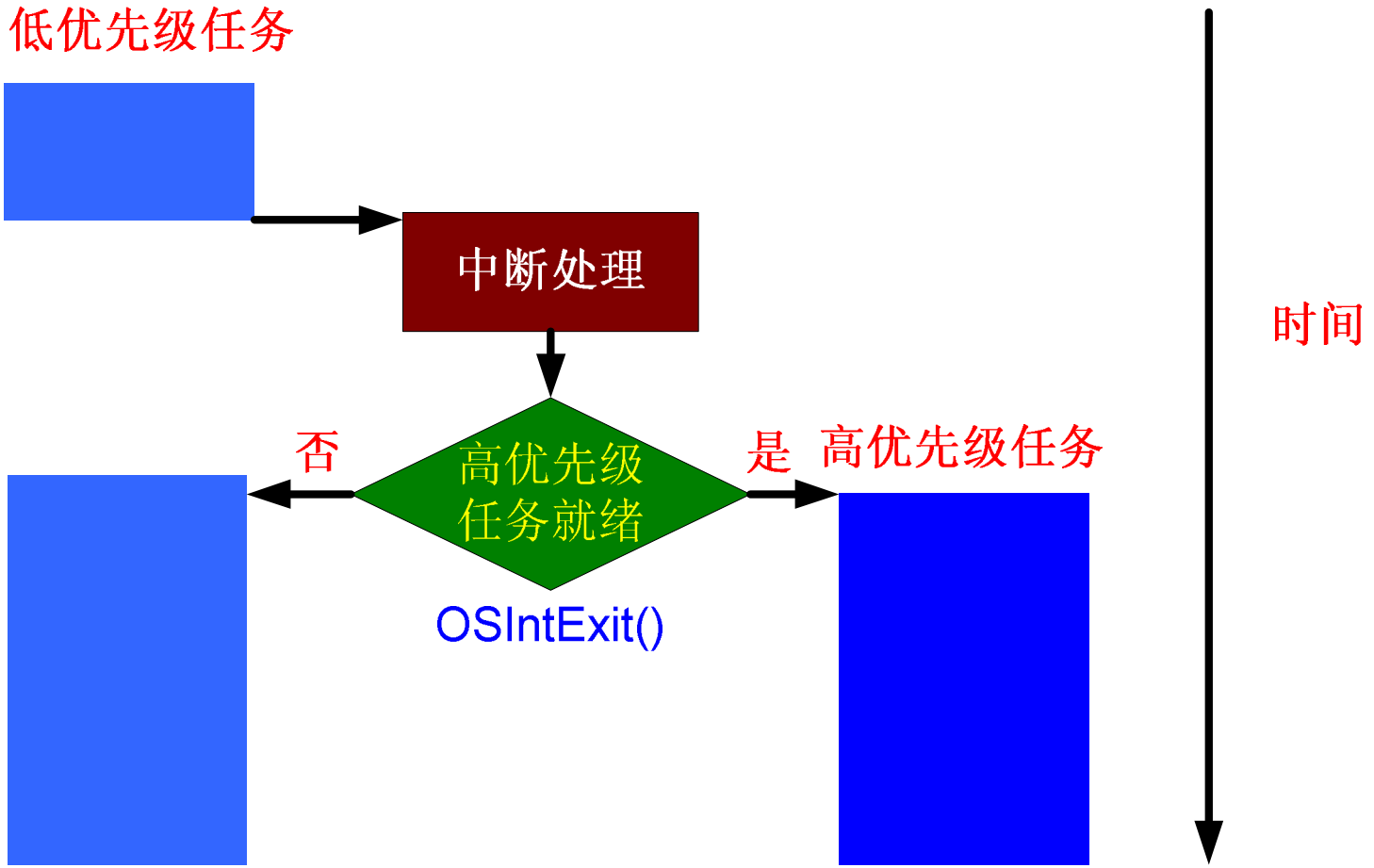
为了方便和统一uCOS-II系统中断的处理，只使用了IRQ模式的中断。

移植uCOS-II要点(2)——系统中断的处理

- 所有中断的调用都需要经过系统的接管。中断处理函数调用前后需要通知系统。例如:

```
OSIntEnter();
    yourInterruptFun();
OSIntExit();
```

OSIntExit的意义



ARM的工作模式

- ARM处理器有7种操作模式:
  - 用户模式(usr)
    - 正常的程序执行模式
  - 快速中断模式(fiq)
    - 支持高速数据传输或通道处理
  - 中断模式(irq)
    - 用于通用中断处理
  - 管理员模式(svc)
    - 操作系统的保护模式.
  - 中止模式(abt)
    - 支持虚拟内存和/或内存保护等异常
  - 系统模式(sys)
    - 支持操作系统的特殊用户模式(运行操作系统任务)
  - 未定义模式(und)
    - 支持硬件协处理器的软件仿真
- 除了用户模式外，其他模式均可视为特权模式

ARM的寄存器（1）

- 37个寄存器
  - 31 个通用32位寄存器，包括程序计数器PC
  - 6 个状态寄存器
  - 15个通用寄存器 (R0 to R14)，以及2个状态寄存器和程序计数器（PC）在任何时候都中可见的
- 可见的寄存器取决于处理器的模式，不同的模式映射了不同的工作寄存器

ARM寄存器的组织

User32	Fiq32	Supervisor32	Abort32	IRQ32	Undefined32
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13(SP)	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14(LR)	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

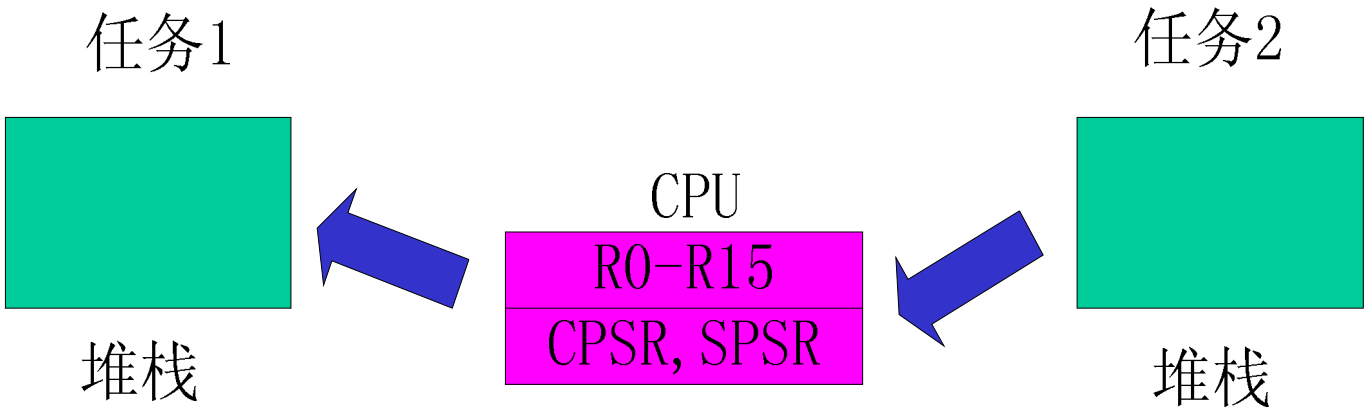
注:表明用户或系统模式使用的正常寄存器已经被异常模式指定的另一个寄存器取代

ARM的寄存器（2）

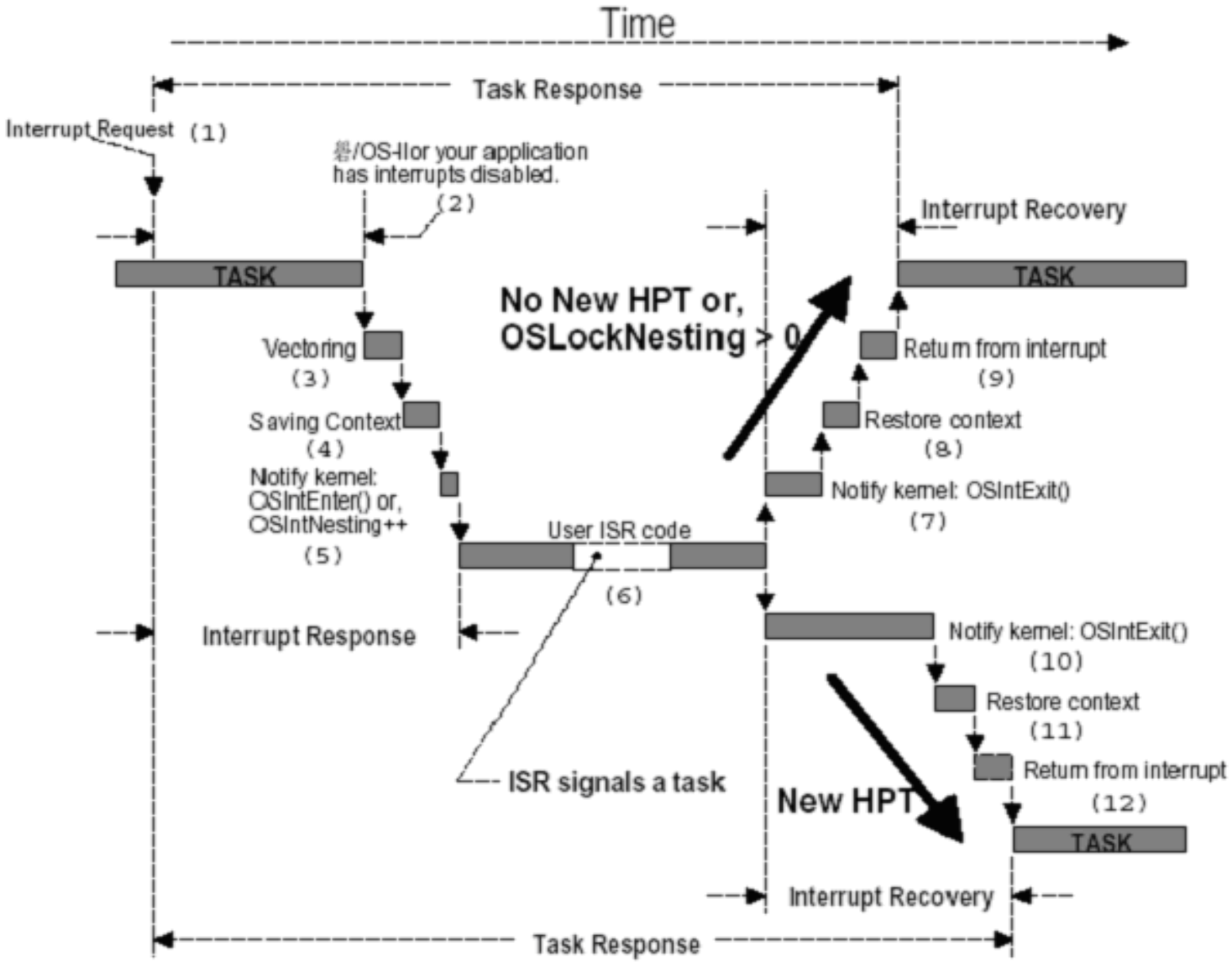
- R0 到 R15 可以直接访问
- R0 到 R14 是通用寄存器
- R13：堆栈指针（sp）（通常）
  - 每种处理器模式都有单独的堆栈
- R14：链接寄存器（lr）
- R15：程序计数器（PC）
- CPSR – 当前程序状态寄存器，包括代码标志状态和当前模式位
- 5个SPSR--(程序状态保存寄存器) 当异常发生时保存CPSR状态

uCOS-II在ARM上的任务切换

- 任务级的任务切换
- 中断级的任务切换



中断处理过程

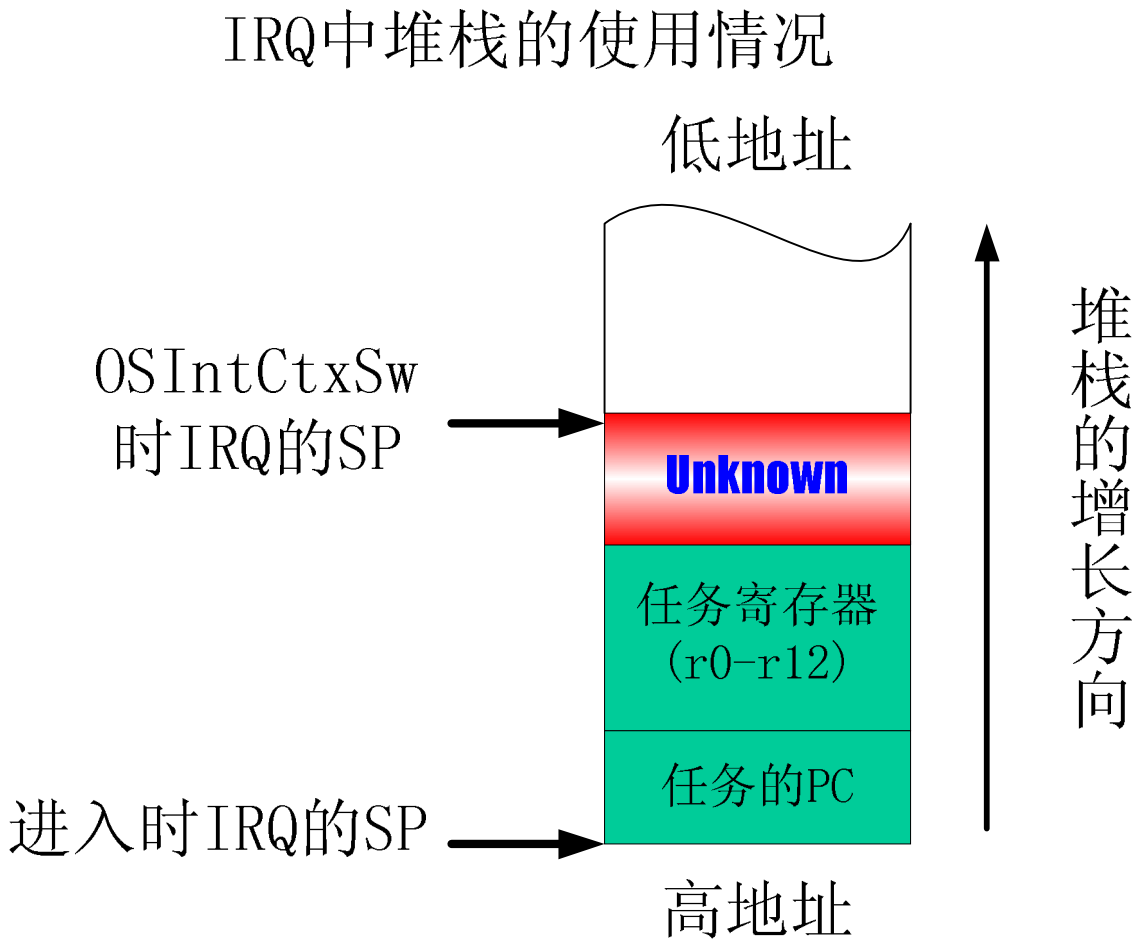


OSIntExit()

```
void OSIntExit (void){
    OS_ENTER_CRITICAL();                                    (1)
   if ((--OSIntNesting | OSLockNesting) == 0) {             (2)
         OSIntExitY    = OSUnMapTbl[OSRdyGrp];             (3)
         OSPrioHighRdy = (INT8U)((OSIntExitY << 3) +
               OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
    if (OSPrioHighRdy != OSPrioCur) {
         OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
         OSCtxSwCtr++;
         OSIntCtxSw();                                      (4)
         }
     }
   OS_EXIT_CRITICAL();
}
```

OSIntExit的关键——OSIntCtxSw

- 实现中断级的任务切换
- ARM在栈指针调整过程中的优势



移植uCOS-II要点(3)——Thumb带来的问题

- 很多ARM内核集成了16位thumb指令集
- Thumb可以在一定程度上节省代码空间，提高系统效率
- Thumb会给中断级的任务切换带来麻烦
  - CPSR中的T位不能直接操作
  - Thumb状态将导致CPSR恢复以后的指令不能运行
- 解决办法:
  - 对Thumb的使用必须保证原子操作
  - 专门对任务切换中Thumb的情况作处理

- 建议，小心使用C编译器。尽量不使用Thumb。

移植uCOS-II的要点(4)——何时启动系统定时器

- 如果在OSStart之前启动定时器，则系统可能无法正确执行完OSStartHighRdy
- OSStart函数直接调用OSStartHighRdy去执行最高优先级的任务，OSStart不返回。
- 系统定时器应该在系统的最高优先级任务中启动
- 使用OSRunning变量来控制操作系统的运行

- 在我们的移植版本中，使用了uCOS-II中的保留任务1作为系统任务。负责启动定时器

例：μC/OS-II在S3C44B0X上的移植

- 设置OS_CPU.H中与处理器和编译器相关的代码
- 用C语言编写六个操作系统相关的函数（OS_CPU_C.C）
- 用汇编语言编写四个与处理器相关的函数（OS_CPU.ASM）

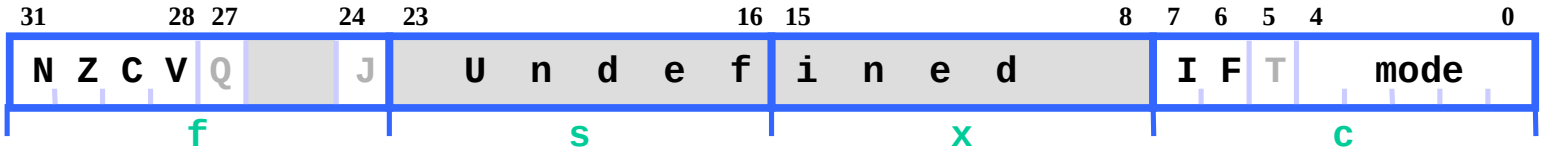
设置与处理器和编译器相关的代码

- OS_CPU.H中定义了与编译器相关的数据类型。比如：INT8U、INT8S等。
- 与 ARM处理器相关的代码，使用OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 宏开启／关闭中断
- 设置堆栈的增长方向 ： 堆栈由高地址向低地址增长

设置includes.h

```
typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;
typedef signed   char  INT8S;
typedef unsigned int   INT16U;
typedef signed   int   INT16S;
typedef unsigned long  INT32U;
typedef signed   long  INT32S;
typedef float          FP32;
typedef double         FP64;
typedef unsigned long   OS_STK;
typedef unsigned long   OS_CPU_SR;
extern int  INTS_OFF(void);
extern void INTS_ON(void);
#define  OS_ENTER_CRITICAL()  { cpu_sr = INTS_OFF(); }
#define  OS_EXIT_CRITICAL()   { if(cpu_sr == 0) INTS_ON(); }
#define  OS_STK_GROWTH        1       /*从高向低*/
```

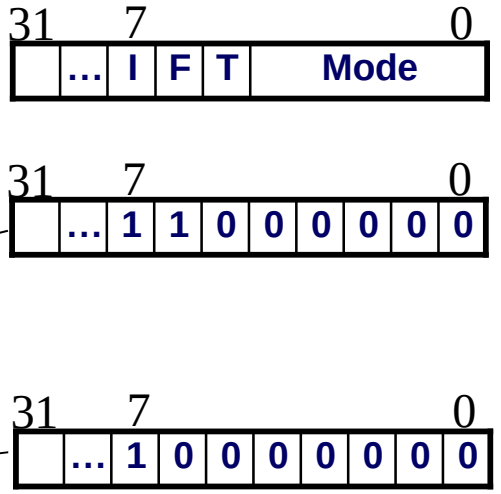
程序状态寄存器



- 条件位:
  - N = 1-结果为负,0-结果为正或0
  - Z = 1-结果为0,0-结果不为0
  - C =1-进位，0-借位
  - V =1-结果溢出，0结果没溢出
- Q 位:
  - 仅ARM 5TE/J架构支持
  - 指示增强型DSP指令是否溢出
- J 位
  - 仅ARM 5TE/J架构支持
  - J = 1: 处理器处于Jazelle状态
- 中断禁止位:
  - I = 1: 禁止 IRQ.
  - F = 1: 禁止 FIQ.
- T Bit
  - 仅ARM xT架构支持
  - T = 0: 处理器处于 ARM 状态
  - T = 1: 处理器处于 Thumb 状态
- Mode位(处理器模式位):
  - 0b10000 User
  - 0b10001 FIQ
  - 0b10010 IRQ
  - 0b10011 Supervisor
  - 0b10111 Abort
  - 0b11011 Undefined
  - 0b11111 System

打开/关闭中断

```
    EXPORT   INTS_OFF
    EXPORT   INTS_ON

INTS_OFF
   mrs   r0, cpsr          ; current CSR
   mov   r1, r0            ; make a copy for
    masking
   orr   r1, r1, #0xC0     ; mask off int bits
   msr   CPSR_cxsf, r1     ; disable ints
   and   r0, r0, #0x80     ; return IRQ bit
   mov   pc,lr             ; return

INTS_ON
   mrs   r0, cpsr          ; current CSR
   bic   r0, r0, #0x80     ; mask on ints
   msr   CPSR_cxsf, r0     ; enable ints
   mov   pc,lr             ; return
```



设置OS_STK_GROWTH

- 绝大多数的微处理器和微控制器的堆栈是从上往下长的。但是某些处理器是用另外一种方式工作的。mC/OS-Ⅱ被设计成两种情况都可以处理，只要在结构常量OS_STK_GROWTH中指定堆栈的生长方式就可以了。

- 置OS_STK_GROWTH为0表示堆栈从下往上长。

- 置OS_STK_GROWTH为1表示堆栈从上往下长。

用C语言编写六个操作系统相关的函数

- `void *OSTaskStkInit (void (*task)(void *pd),void *pdata, void *ptos, INT16U opt)`
- `void OSTaskCreateHook (OS_TCB *ptcb)`
- `void OSTaskDelHook (OS_TCB *ptcb)`
- `void OSTaskSwHook (void)`
- `void OSTaskStatHook (void)`
- `void OSTimeTickHook (void)`

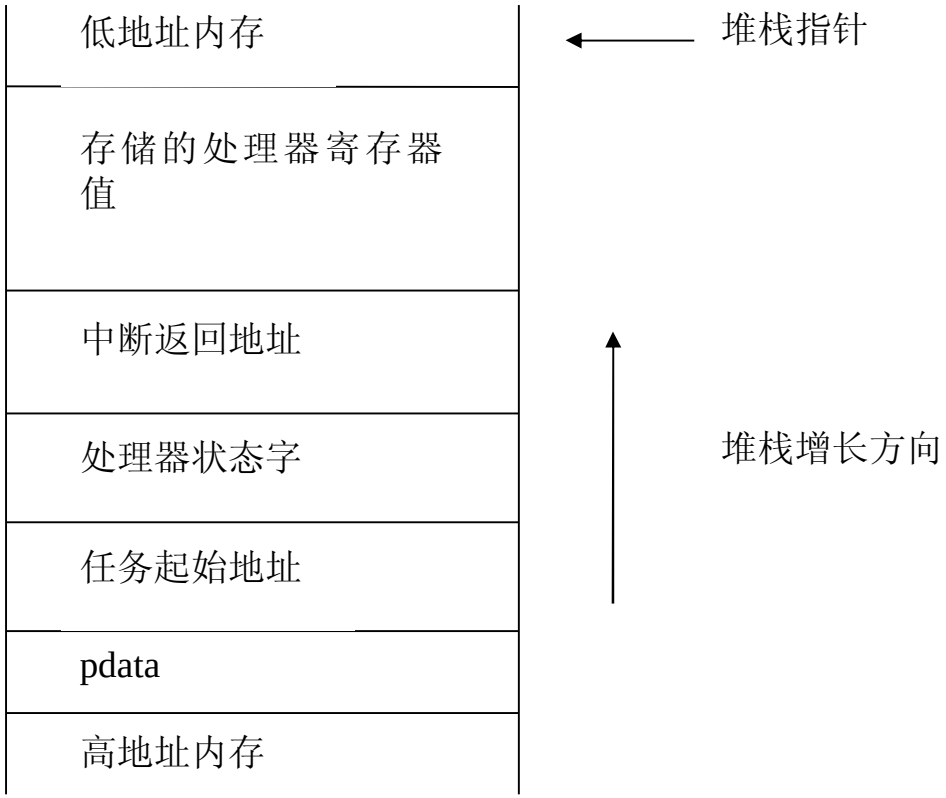
后5个函数为接口函数，可以不加代码

OSTaskStkInit

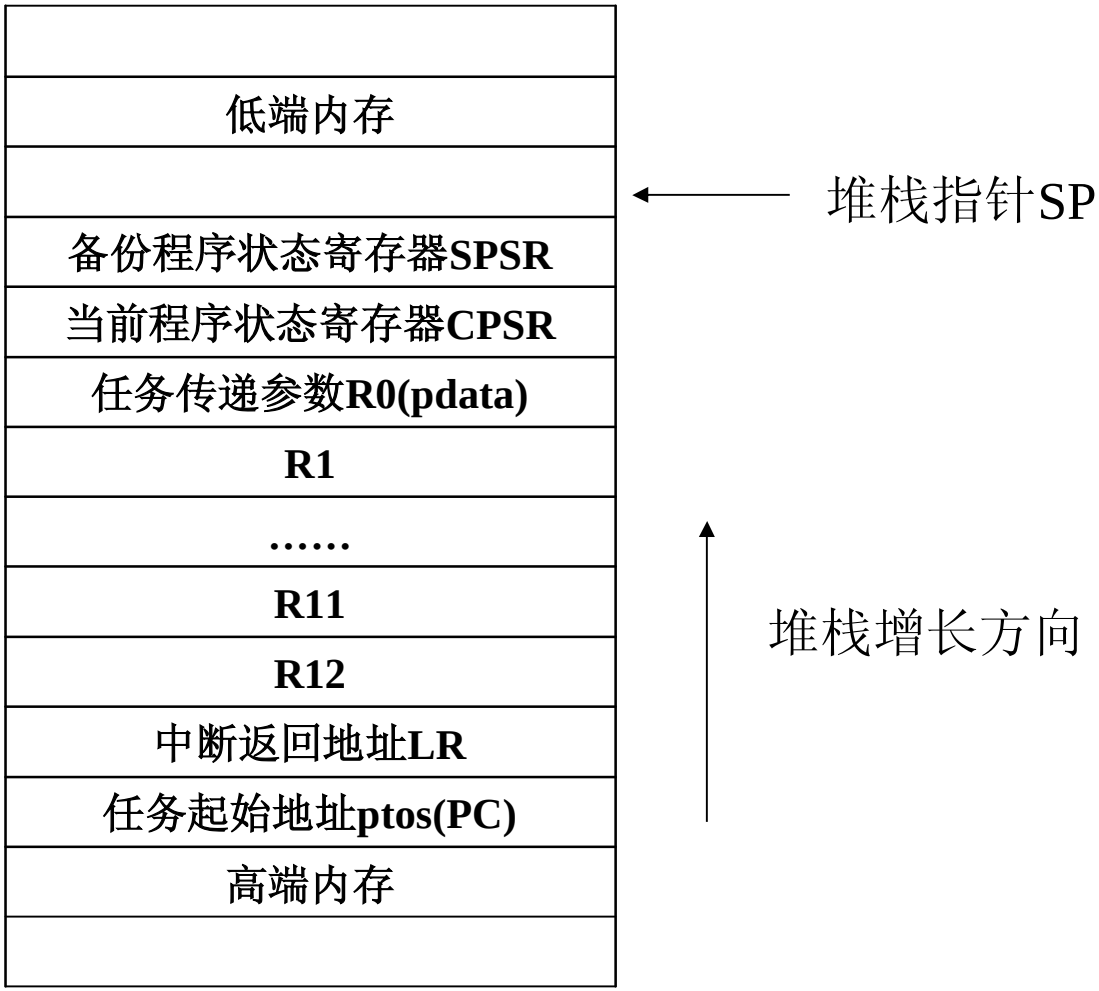
- OSTaskCreate（）和OSTaskCreateExt（）通过调用OSTaskStkInit（）来初始化任务的堆栈结构。因此，堆栈看起来就像刚发生过中断并将所有的寄存器保存到堆栈中的情形一样。上图显示了OSTaskStkInt（）放到正被建立的任务堆栈中的内容。这里我们定义了堆栈是从上往下长的。

- 在用户建立任务的时候，用户传递任务的地址、pdata指针、任务的堆栈栈顶和任务的优先级给OSTaskCreate（）和OSTaskCreateExt（）。一旦用户初始化了堆栈，OSTaskStkInit（）就需要返回堆栈指针所指的地址。OSTaskCreate（）和OSTaskCreateExt（）会获得该地址并将它保存到任务控制块（OS_TCB）中。

堆栈初始化



ARM系统的堆栈初始化



OSTaskStkInit

```
OS_STK * OSTaskStkInit (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT16U opt)
{
    unsigned int * stk;
    stk    = (unsigned int *)ptos;        /* Load stack pointer */
    //USE_ARG(opt);
    opt++;
    /* build a stack for the new task */
    *--stk = (unsigned int) task;     /* pc */
    *--stk = (unsigned int) task;     /* lr */
    *--stk = 12;                      /* r12 */
    *--stk = 11;                      /* r11 */
    *--stk = 10;                      /* r10 */
    *--stk = 9;                       /* r9 */
    *--stk = 8;                       /* r8 */
    *--stk = 7;                       /* r7 */
    *--stk = 6;                       /* r6 */
    *--stk = 5;                       /* r5 */
    *--stk = 4;                       /* r4 */
    *--stk = 3;                       /* r3 */
    *--stk = 2;                       /* r2 */
    *--stk = 1;                       /* r1 */
    *--stk = (unsigned int) pdata;    /* r0 */
    *--stk = (SUPMODE);                                          /* cpsr */
    *--stk = (SUPMODE);                          /* spsr */
    return ((OS_STK *)stk);
}
```

OSTaskCreateHook

- 当用OSTaskCreate（）和OSTaskCreateExt（）建立任务的时候就会调用OSTaskCreateHook（）。该函数允许用户或使用移植实例的用户扩展mC/OS-Ⅱ功能。当mC/OS-Ⅱ设置完了自己的内部结构后，会在调用任务调度程序之前调用OSTaskCreateHook（）。该函数被调用的时候中断是禁止的。因此用户应尽量减少该函数中的代码以缩短中断的响应时间。
- 当OSTaskCreateHook（）被调用的时候，它会收到指向已建立任务的OS_TCB的指针，这样它就可以访问所有的结构成员了。
- 函数原型：

```
void OSTaskCreateHook (OS_TCB *ptcb)
{
    ptcb=ptcb;
}
```

OSTaskDelHook

- 当任务被删除的时候就会调用OSTaskDelHook（）。该函数在把任务从mC/OS-Ⅱ的内部任务链表中删除之前被调用。当OSTaskDelHook（）被调用的时候，它会收到指向正被删除任务的OS_TCB的指针，这样它就可以访问所有的结构成员了。OSTaskDelHook（）可以来检验TCB扩展是否被建立（一个非空指针），并进行一些清除操作。

- 函数原型：void OSTaskDelHook (OS_TCB *ptcb)

```
{
    ptcb=ptcb;
 }
```

OSTaskSwHook

- 当发生任务切换的时候就会调用OSTaskSwHook（）。OSTaskSwHook（）可以直接访问OSTCBCur和OSTCBHighRdy，因为它们是全局变量。OSTCBCur指向被切换出去的任务OS_TCB，而OSTCBHighRdy指向新任务OS_TCB。注意在调用OSTaskSwHook（）期间中断一直是被禁止的。因此用户应尽量减少该函数中的代码以缩短中断的响应时间。

- 函数原型：

```
void OSTaskSwHook (void)
{
 #if 0
     if(OSRunning==TRUE){
         /*保存拟被挂起任务的寄存器;
     }
         /*恢复拟被运行任务的寄存器;
 #endif
 }
```

OSTaskStatHook

- OSTaskStatHook（）每秒钟都会被OSTaskStat（）调用一次。用户可以用OSTaskStatHook（）来扩展统计功能。例如，用户可以保持并显示每个任务的执行时间，每个任务所用的CPU份额，以及每个任务执行的频率等。

- 函数原型：void OSTaskStatHook (void)

OSTimeTickHook

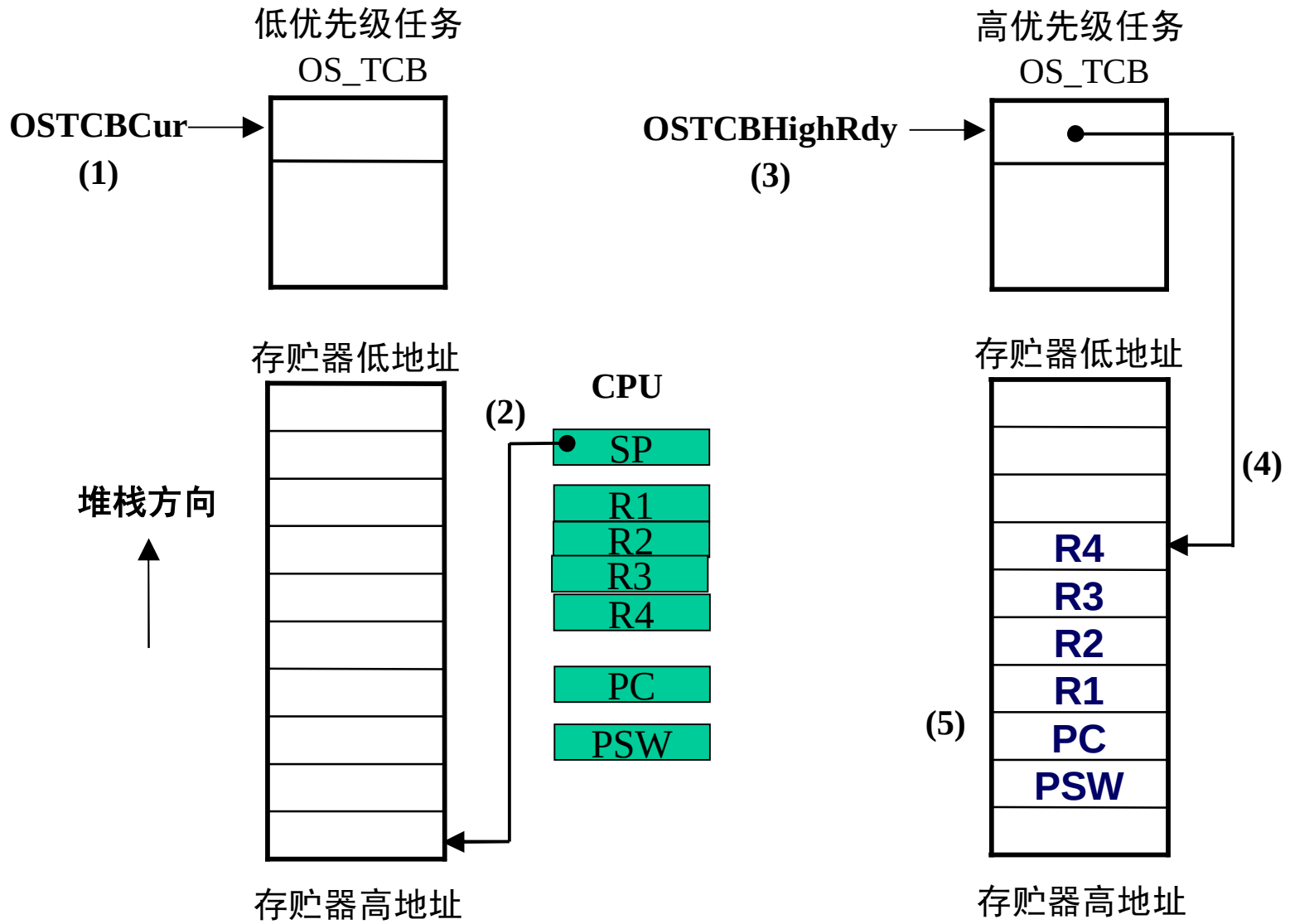
- OSTimeTickHook（）在每个时钟节拍都会被OSTaskTick（）调用。实际上，OSTimeTickHook（）是在节拍被mC/OS-Ⅱ真正处理，并通知用户的移植实例或应用程序之前被调用的。

- 函数原型：void OSTimeTickHook (void)

用汇编语言编写四个与处理器相关的函数

- OSStartHighRdy()
- OSCtxSw()
- OSIntCtxSw()
- OSTickISR()

数据结构



OSStartHighRdy()：运行优先级最高的就绪任务

```
OSStartHighRdy

LDR  r4, addr_OSTCBCur       ; 得到当前任务TCB地址
LDR  r5, addr_OSTCBHighRdy   ; 得到最高优先级任务TCB地址
LDR  r5, [r5]                ; 获得最高优先级任务堆栈栈顶指针
LDR  sp, [r5]                ; 转移到新的堆栈中
STR  r5, [r4]                ; 设置新的当前任务TCB地址
LDMFD       sp!, {r4}        ;
MSR SPSR, r4
LDMFD       sp!, {r4}        ; 从栈顶获得新的状态
MSR CPSR, r4                 ; CPSR 处于 SVC32Mode摸式
LDMFD       sp!, {r0-r12, lr, pc }      ; 运行新的任务
```

OSCtxSw()的原型

```
void OSCtxSw(void)
{
    保存处理器寄存器;
    将当前任务的堆栈指针保存到当前任务的OS_TCB中:
        OSTCBCur->OSTCBStkPtr = Stack pointer;
    调用用户定义的OSTaskSwHook();
    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    得到需要恢复的任务的堆栈指针:
        Stack pointer = OSTCBHighRdy->OSTCBStkPtr;
    将所有处理器寄存器从新任务的堆栈中恢复出来;
    执行中断返回指令;
}
```

OS_TASK_SW（）：任务级的任务切换函数（1）

```
OS_TASK_SW
STMFD       sp!, {lr}          ; 保存 pc
STMFD       sp!, {lr}          ; 保存 lr
STMFD       sp!, {r0-r12}      ; 保存寄存器和返回地址
MRS         r4, CPSR
STMFD       sp!, {r4}                  ; 保存当前的PSR
MRS         r4, SPSR
STMFD       sp!, {r4}                  ; 保存SPSR
; OSPrioCur = OSPrioHighRdy
LDR         r4, addr_OSPrioCur
LDR         r5, addr_OSPrioHighRdy
LDRB        r6, [r5]
STRB        r6, [r4]
```

OS_TASK_SW()：任务级的任务切换函数（2）

```
    ; 得到当前任务TCB地址
    LDR   r4, addr_OSTCBCur
    LDR   r5, [r4]
    STR   sp, [r5]                ; 保存sp在被占先的任务的 TCB
    ; 得到最高优先级任务TCB地址
    LDR   r6, addr_OSTCBHighRdy
    LDR   r6, [r6]
    LDR   sp, [r6]                ; 得到新任务堆栈指针
    ; OSTCBCur = OSTCBHighRdy
    STR   r6, [r4]                ; 设置新的当前任务的TCB地址
;保存任务方式寄存器
    LDMFD           sp!, {r4}
    MSR  SPSR, r4
    LDMFD           sp!, {r4}
    MSR  CPSR, r4
; 返回到新任务的上下文
    LDMFD           sp!, {r0-r12, lr, pc}
```

中断服务

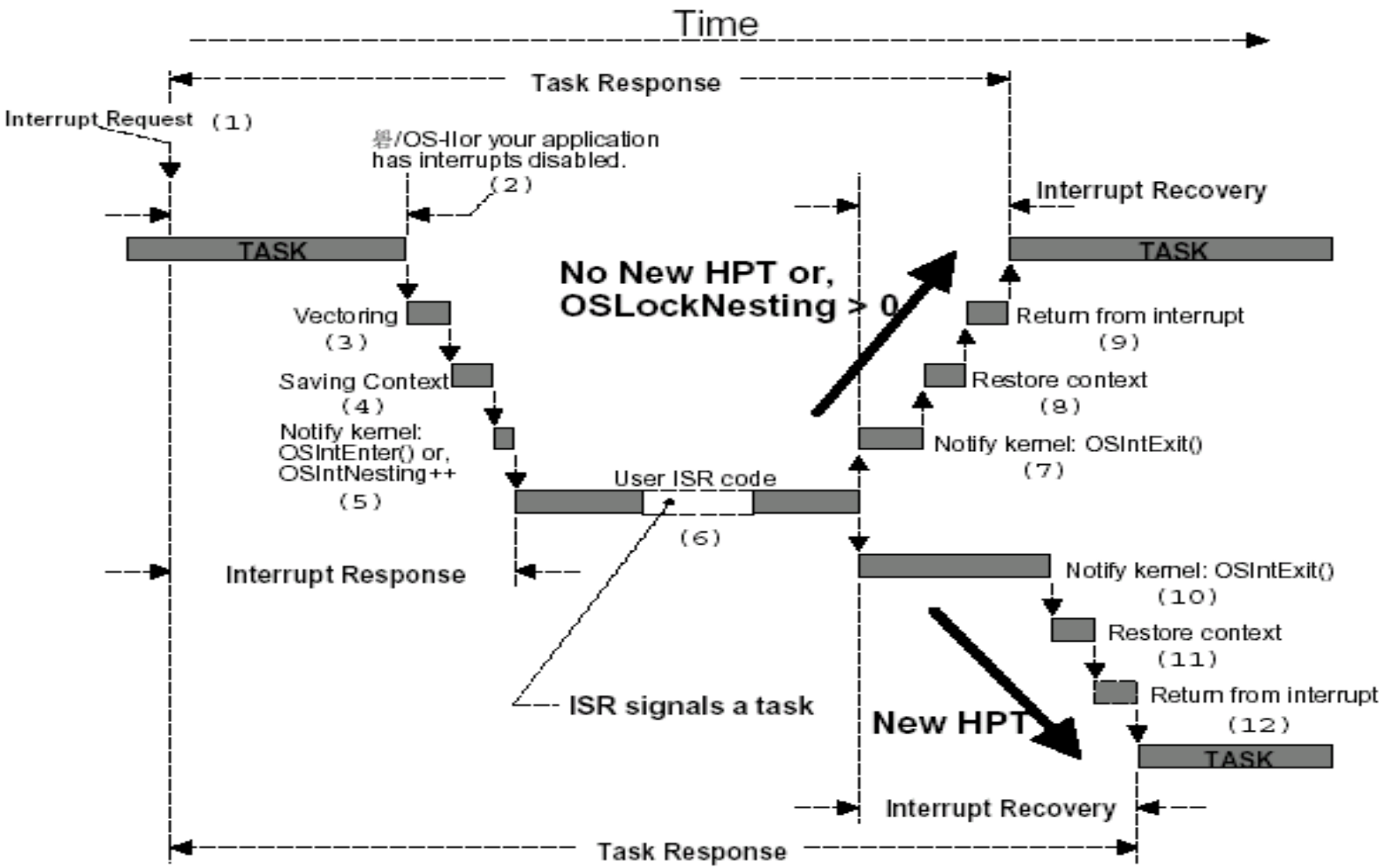
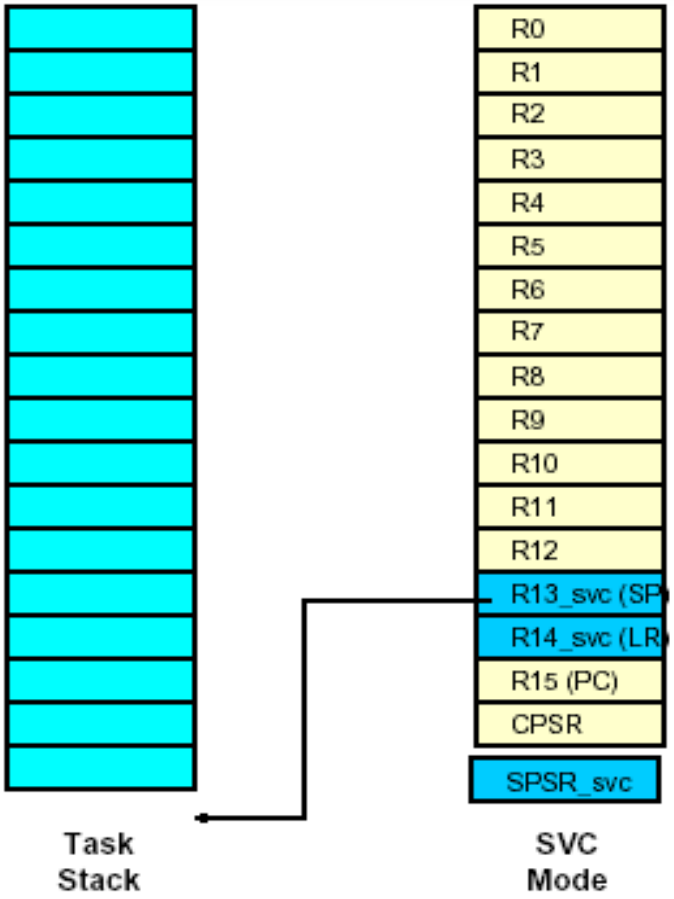


Figure 3-5, Servicing an interrupt

中断服务程序



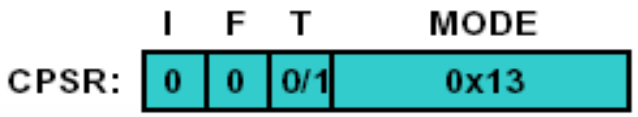
It is assumed that a task is running (in SVC mode) when an interrupt occurs.

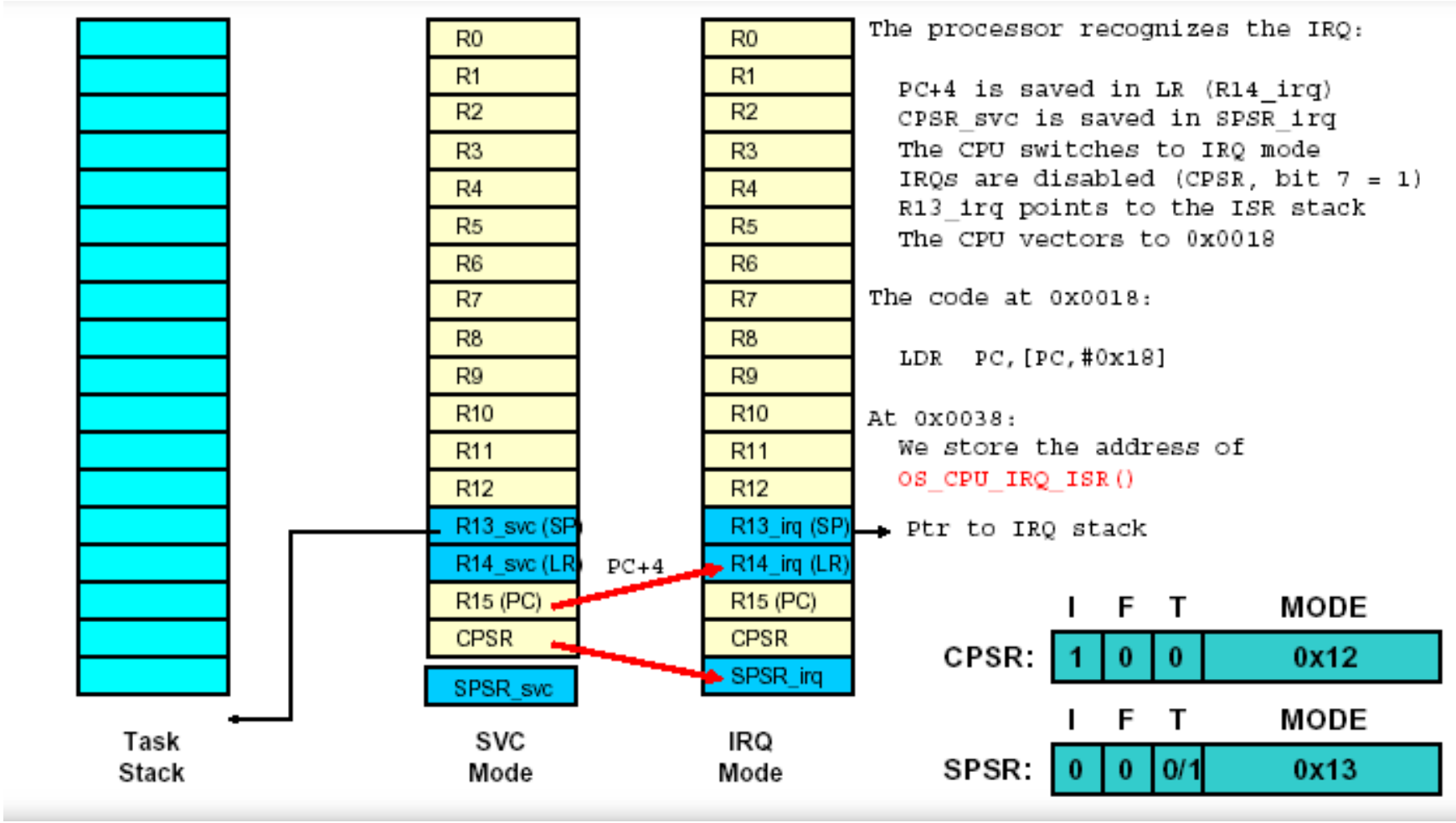
The processor's SP (R13) points to 'some' location into the current task's stack.

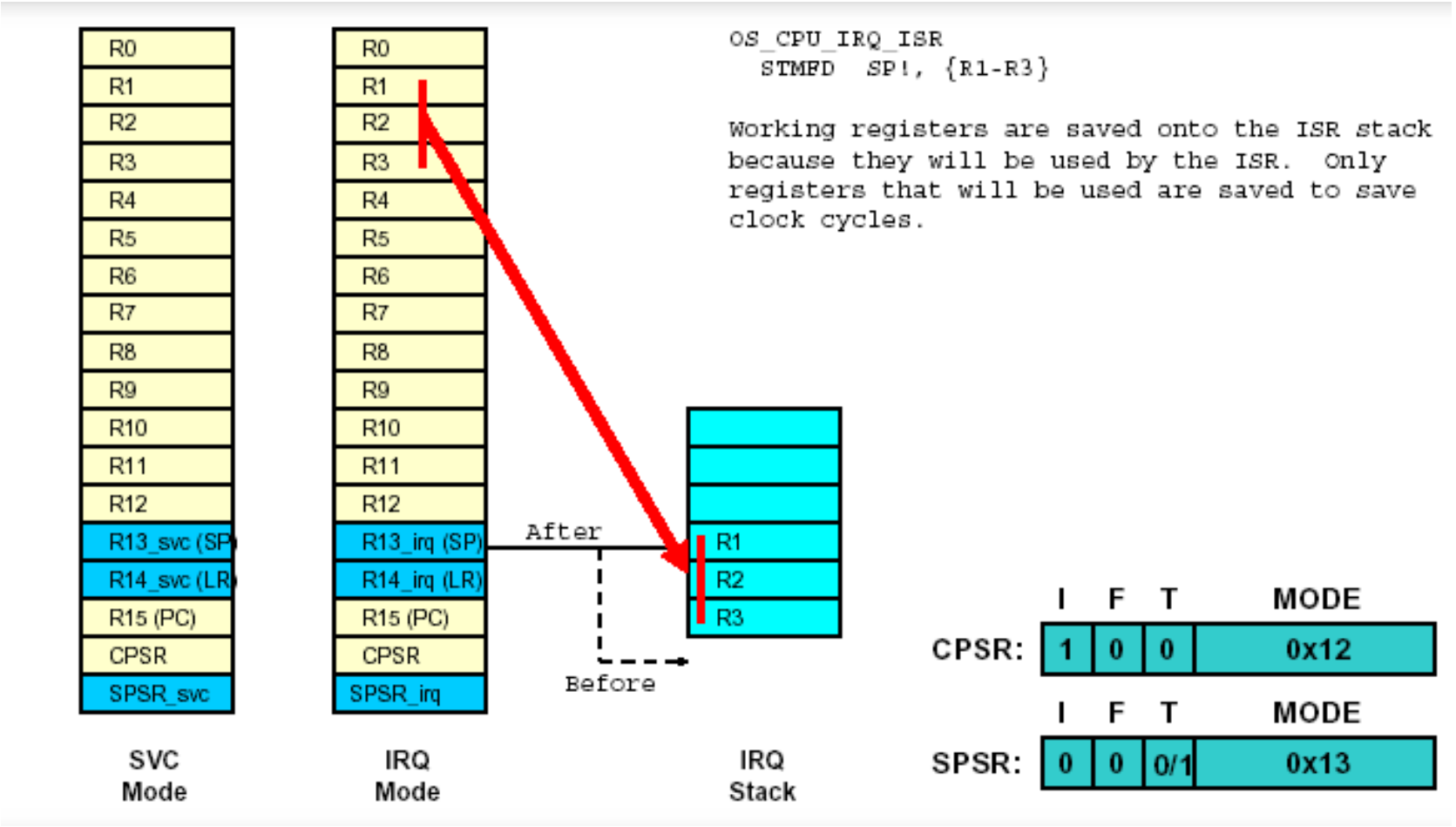
The ISR is implemented as outlined in Jean J. Labrosse's book.

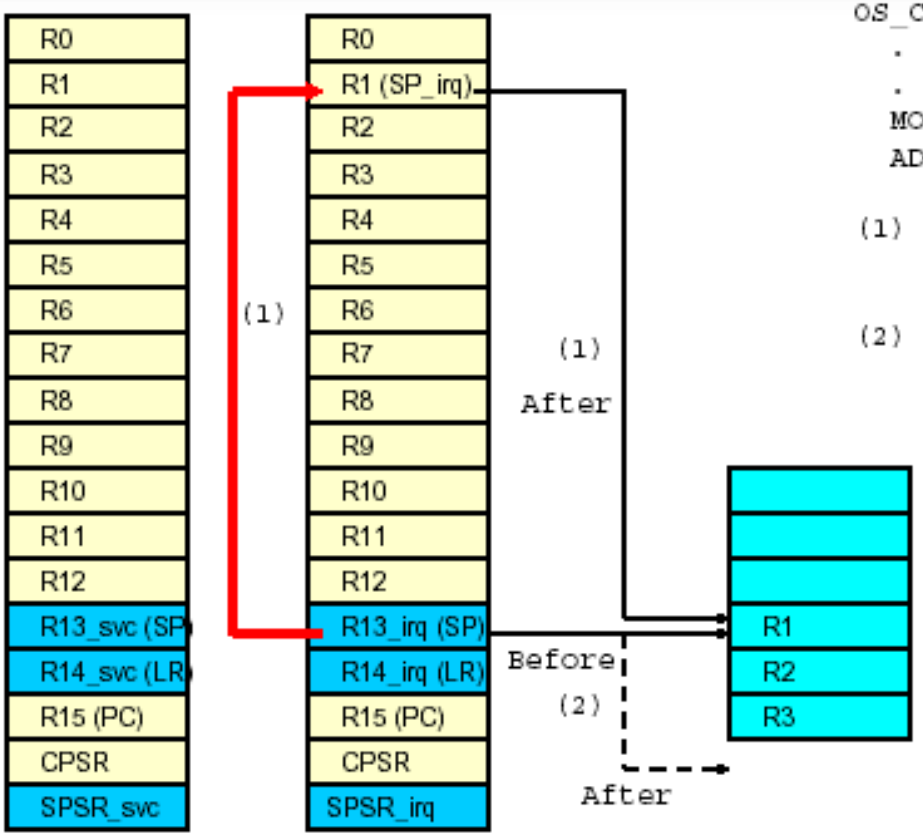
Specifically, your ISRs need to be written as follows:

```
Save CPU registers onto the current task's stack;
OSIntNesting++;
if (OSIntNesting == 1) {
  OSTCBCur->OSTCBStkPtr = SP;
}
Call OS_CPU_???_ISR_Handler in C;      // ??? Is IRQ or FIQ
OSIntExit();
Restore the registers from the current task's stack;
Return from interrupt;
```









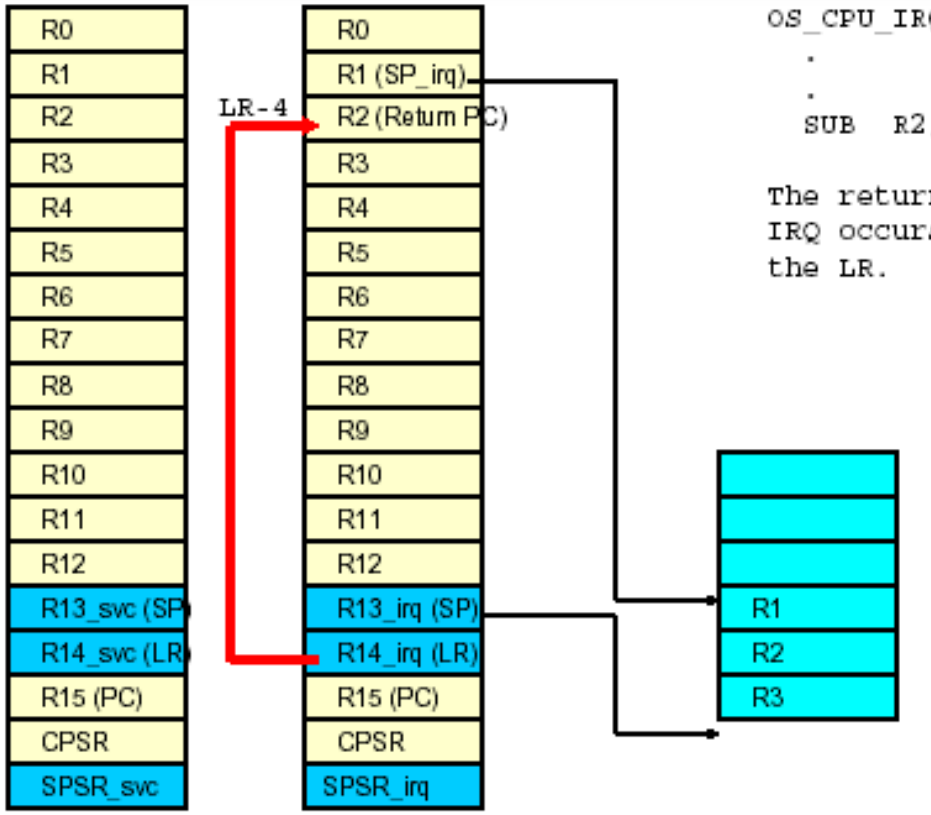

```
OS_CPU_IRQ_ISR
  .
  .
  MOV   R1,SP           ; (1)
  ADD   SP,SP,#(3*4)    ; (2)
```

(1) The IRQ stack pointer is copied to the R1 for future use.

(2) The IRQ stack pointer is adjusted to 'remove' the stacked data. Note that other IRQ interrupts are currently disabled so there is no danger to write over R1-R3.

	I	F	T	MODE
CPSR:	1	0	0	0x12

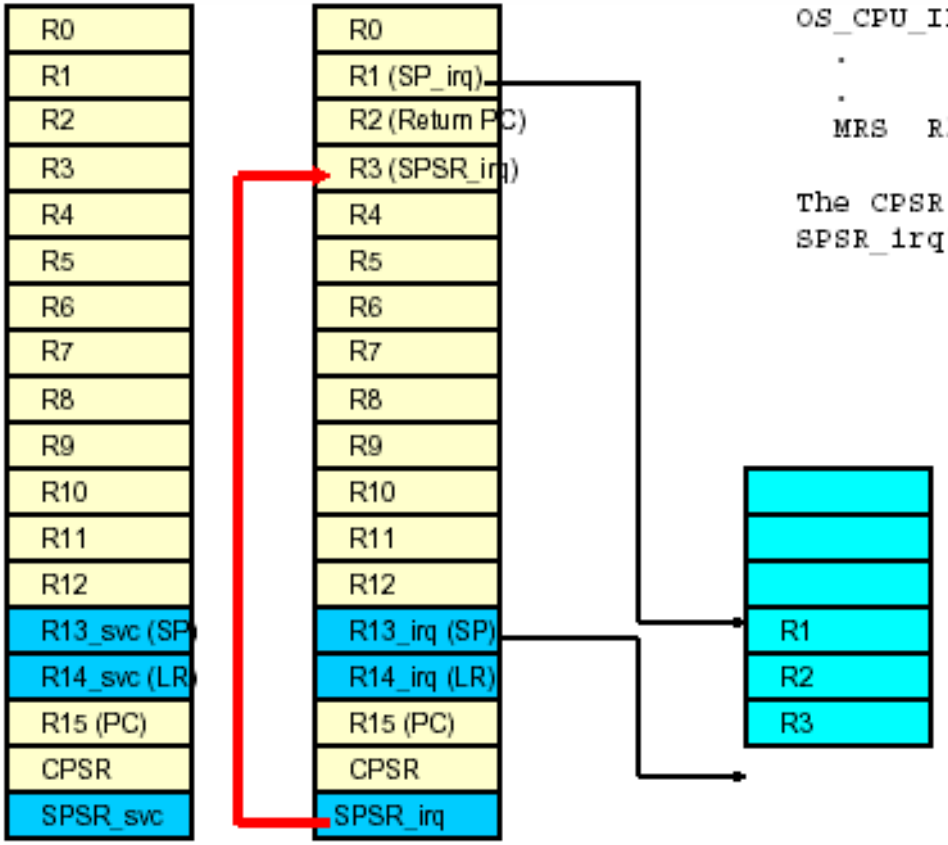
	I	F	T	MODE
SPSR:	0	0	0/1	0x13




```
OS_CPU_IRQ_ISR
   .
   .
   SUB   R2,LR,#4
```

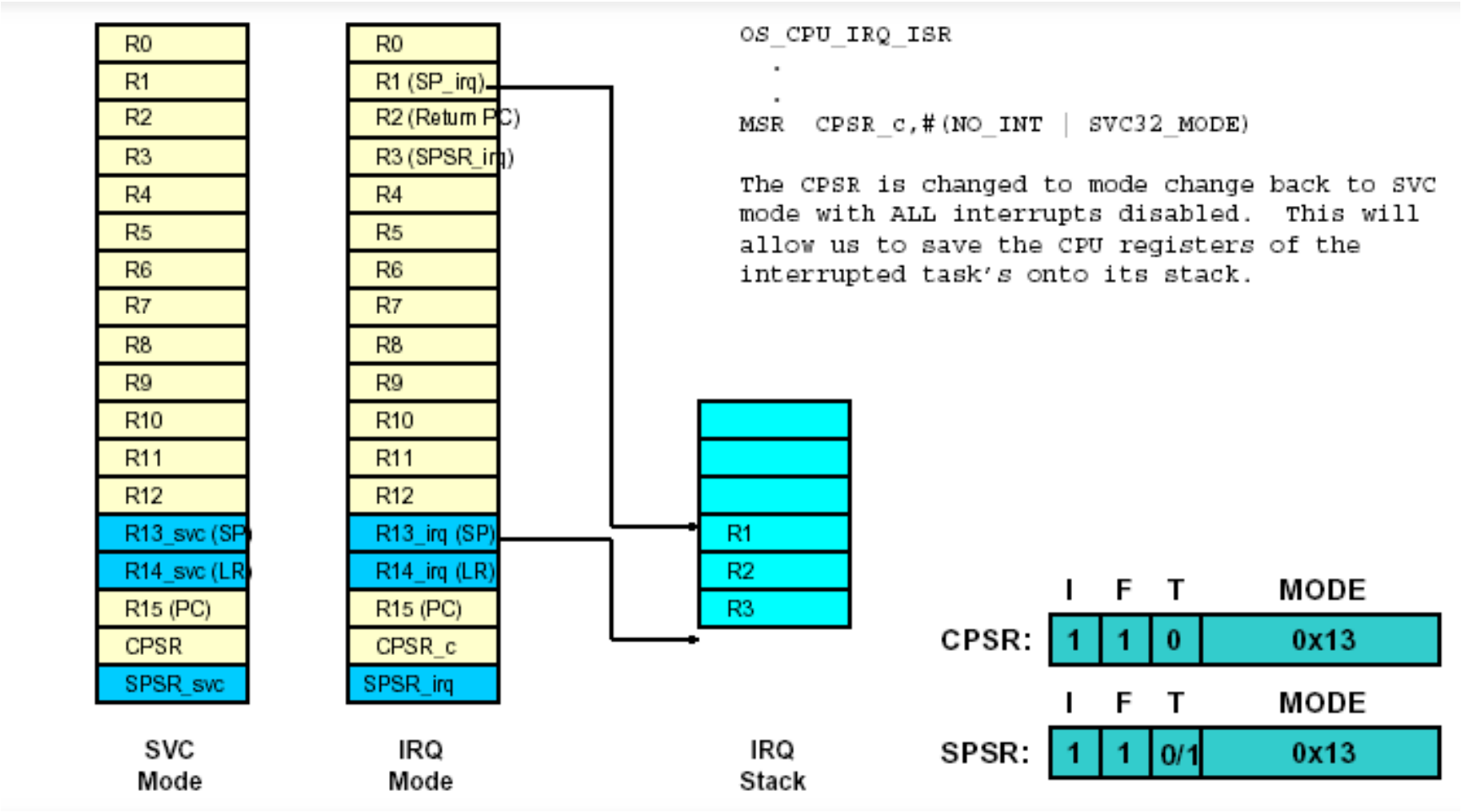
The return address is adjusted because when an IRQ occurs, the CPU saves the return PC+4 into the LR.

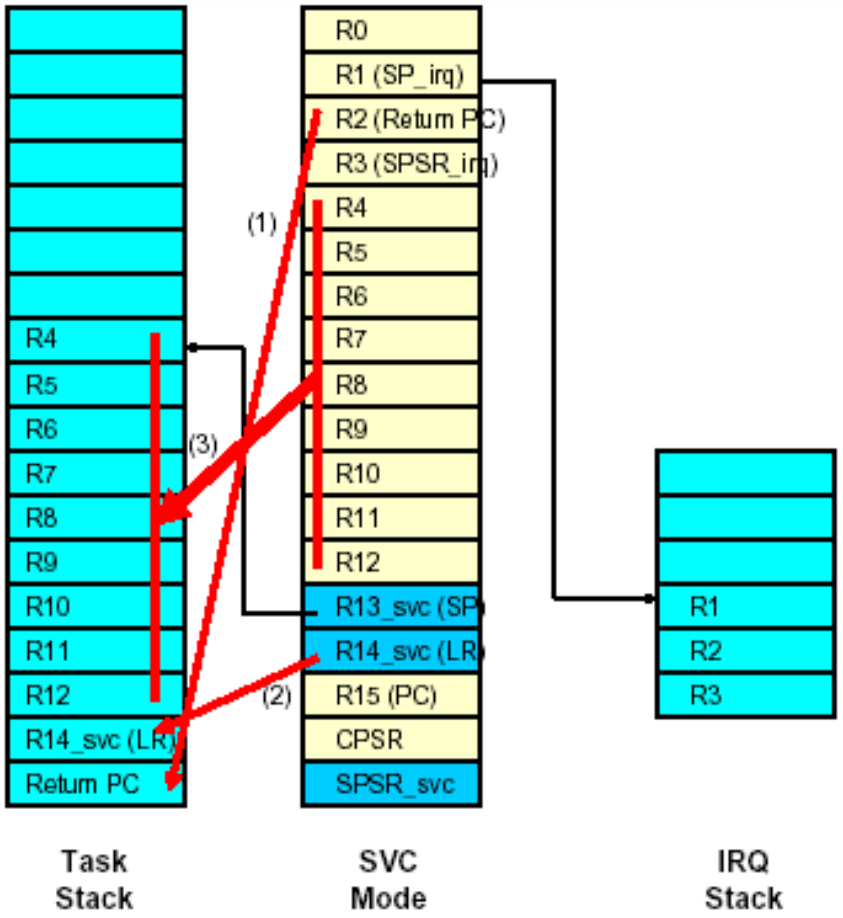
	I	F	T	MODE
CPSR:	1	0	0	0x12
SPSR:	0	0	0/1	0x13




```
OS_CPU_IRQ_ISR
   .
   .
   MRS   R3,SPSR

The CPSR of the interrupted task (i.e. the
SPSR_irq) is saved for future use.
```

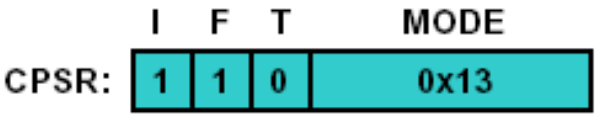
	I	F	T	MODE
CPSR:	1	0	0	0x12
SPSR:	0	0	0/1	0x13

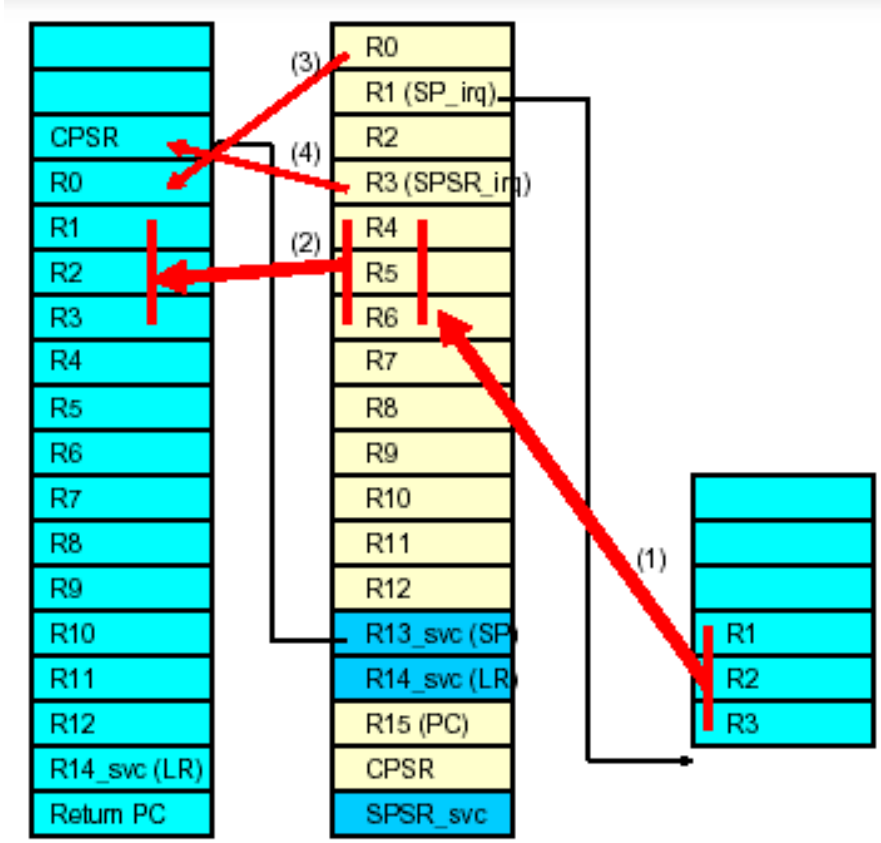





```
OS_CPU_IRQ_ISR
  :
  STMFD   SP!, {R2}       ; (1)
  STMFD   SP!, {LR}       ; (2)
  STMFD   SP!, {R4-R12}   ; (3)
```

We now save the interrupted task's registers to its stack.



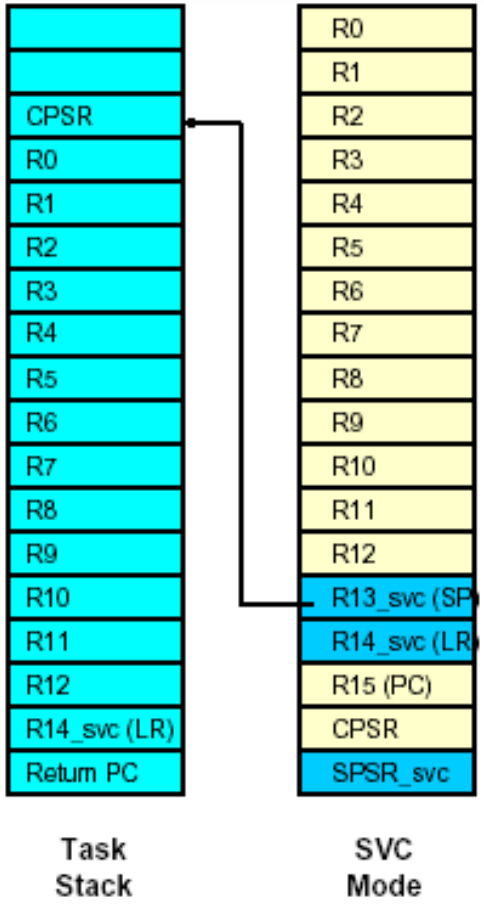



```
OS_CPU_IRQ_ISR
   :
   LDMFD    R1!, {R4-R6}   ; (1)
   STMFD    SP!, {R4-R6}   ; (2)

   STMFD    SP!, {R0}      ; (3)
   STMFD    SP!, {R3}      ; (4)
```

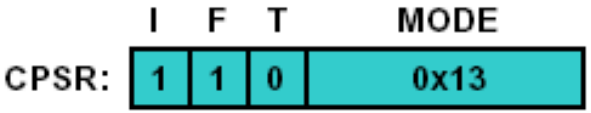
We now save the remaining interrupted task registers and the interrupted task's CPSR.

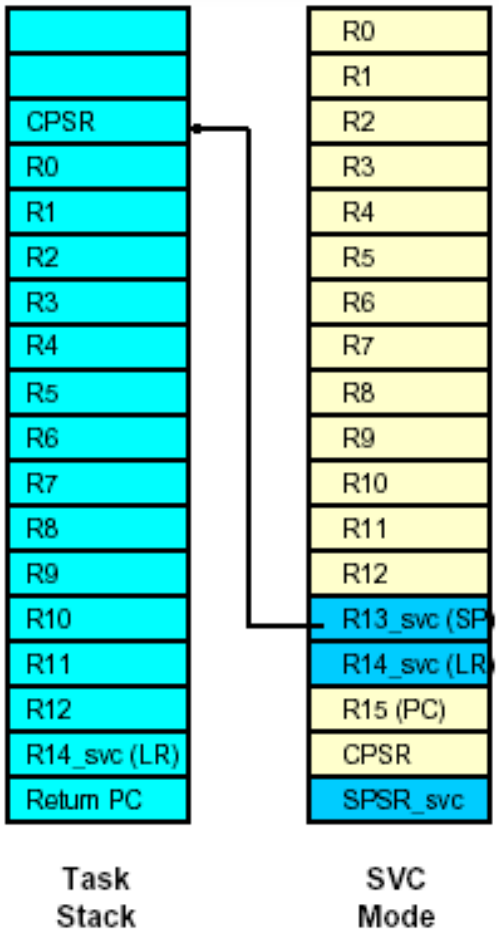
At this point, we saved the interrupted task's context onto its stack.

	I	F	T	MODE
CPSR:	1	1	0	0x13



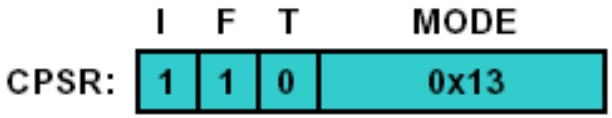

```
OS_CPU_IRQ_ISR
  :
  LDR   R0,??OSIntNesting  ; OSIntNesting++
  LDRB  R1,[R0]
  ADD   R1,R1,#1
  STRB  R1,[R0]

We now increment OSIntNesting to tell µC/OS-II
that we are starting an ISR.
```

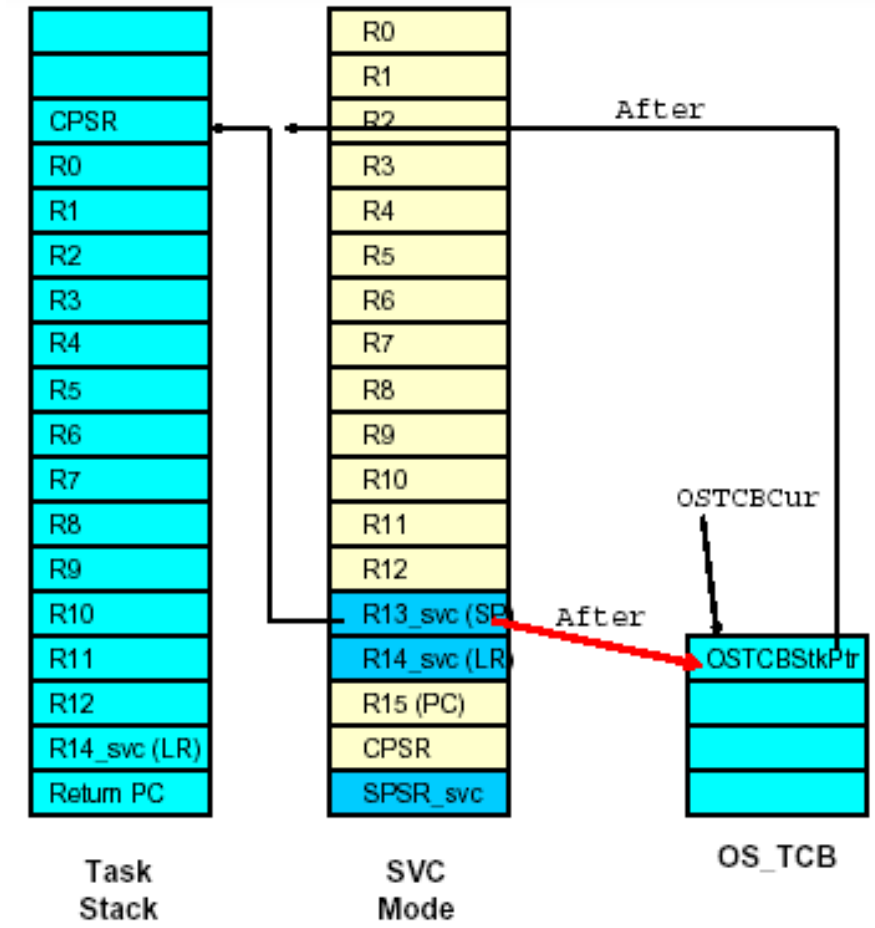





```
OS_CPU_IRQ_ISR
  :
  CMP   R1,#1           ; if (OSIntNesting == 1) {
  BNE   OS_CPU_IRQ_ISR_1
```

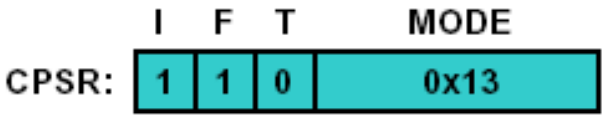
We now check to see if this is the first ISR and if not, we branch around the code shown on the next slide.

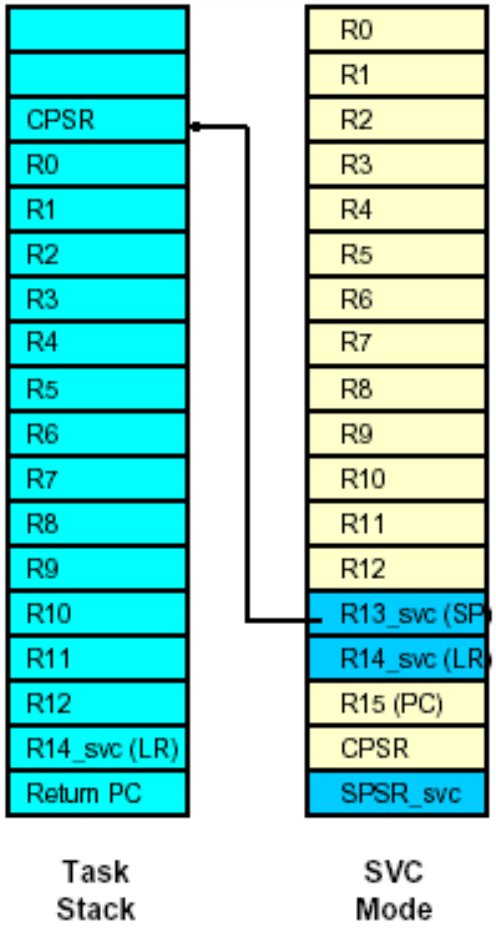





```
OS_CPU_IRQ_ISR
   :
;        OSTCBCur->OSTCBStkPtr = SP
   LDR    R4,??OSTCBCur
   LDR    R5,[R4]
   STR    SP,[R5]
```

If this is the first nested ISR then we save the SP of the current task into its OS_TCB.





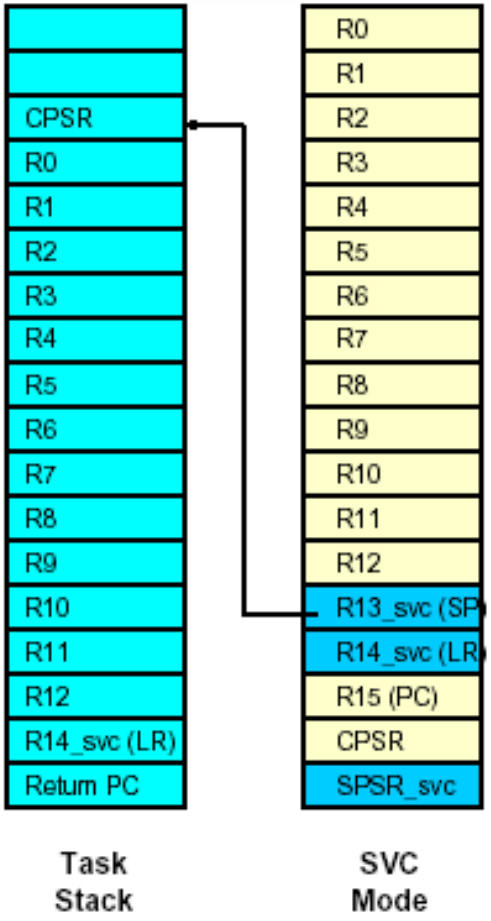

```
OS_CPU_IRQ_ISR
  :
OS_CPU_IRQ_ISR_1
  MSR    CPSR_c,#(NO_INT | IRQ32_MODE)   (1)
;
  LDR    R0,??OS_CPU_IRQ_ISR_Handler     (2)
  MOV    LR,PC
  BX     R0
```

We now switch back to IRQ mode in order to process the ISR using the IRQ stack. This allows to reduce the RAM requirements on the task stack because all ISRs are processed on the IRQ stack.

We now call the code that will handle the ISR. We do this because it's typically easier to write this code in C instead of assembly language.

On a 32-bit bus, the CPU takes about 50 clock cycles to get to this point in the code.

	I	F	T	MODE
CPSR:	1	1	0	0x12

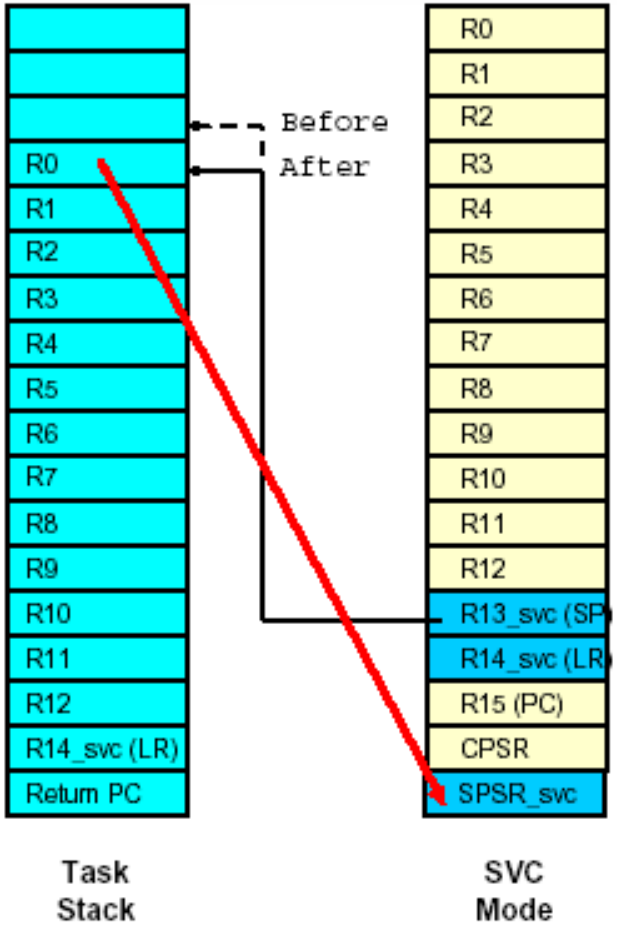



```
OS_CPU_IRQ_ISR:
  :
  MSR    CPSR_c,#(NO_INT | SVC32_MODE)   (1)

  LDR    R0,??OS_IntExit                 (2)
  MOV    LR,PC
  BX     R0
```

We now switch back to SVC mode because we are about to return to task level code.

We now call the µC/OS-II scheduler to determine whether this (or any other nested interrupt) made a higher priority task ready to run. If this is the case, OSIntExit() will NOT return to OS_CPU_IRQ_ISR() but instead, will context switch to the new, more important task (via OSIntCtxSw() (described later).

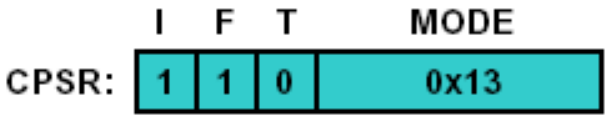
	I	F	T	MODE
CPSR:	1	1	0	0x13

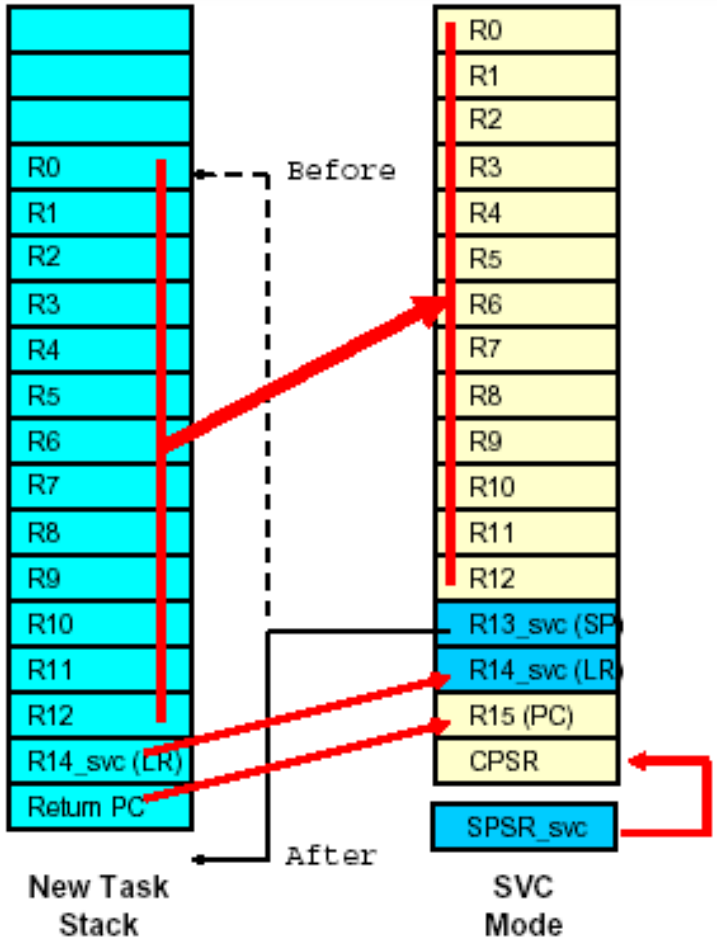



```
OS_CPU_IRQ_ISR:
   :
   LDMFD    SP!, {R4}          ; pop new task's CPSR
   MSR      SPSR_cxsf, R4
```

This code is executed if the interrupted task is still the most important task.

The CPSR of the interrupted task is thus retrieved from the interrupted task's stack and placed in the SPSR register (and NOT the CPSR).





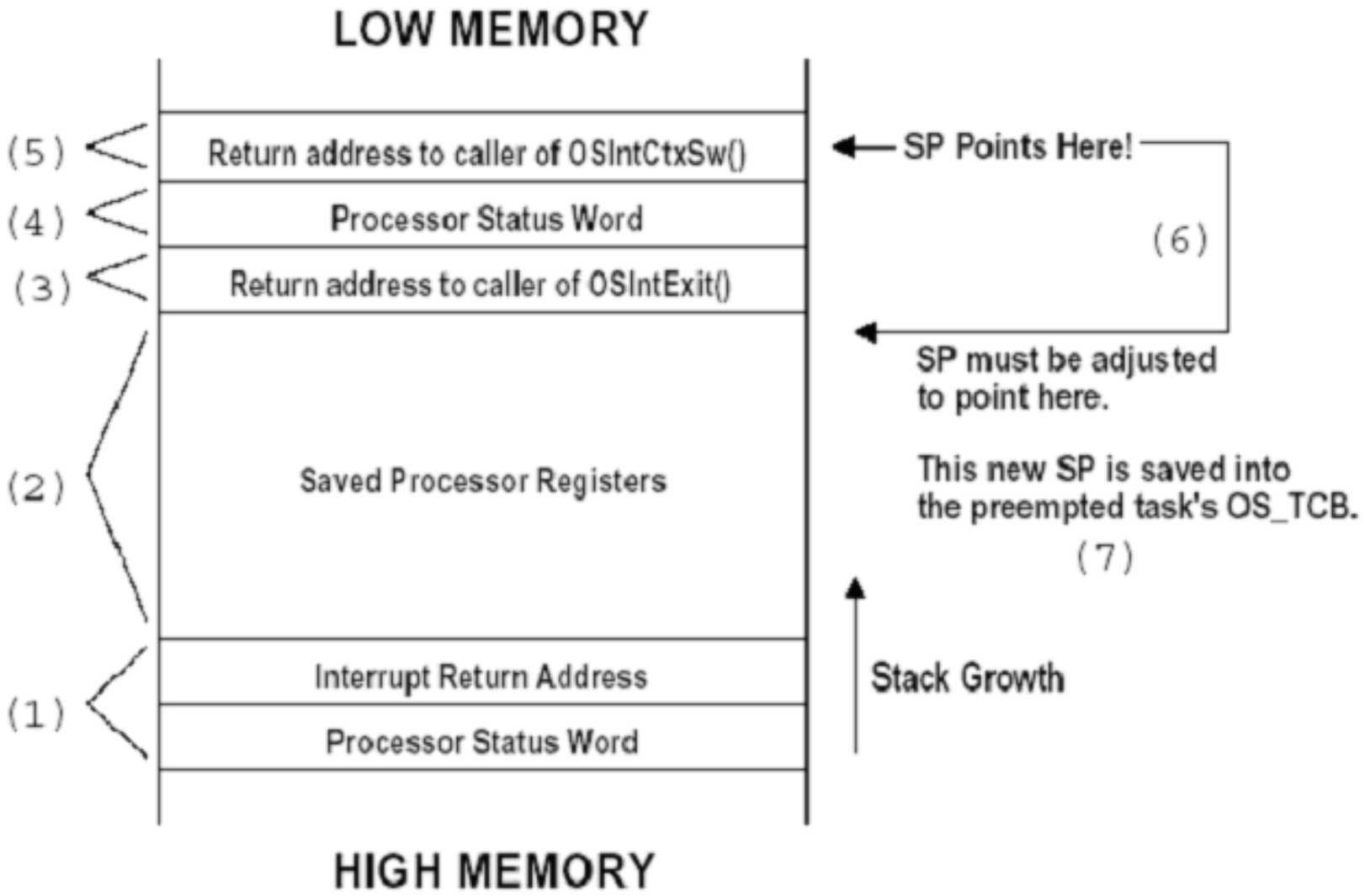

```
OS_CPU_IRQ_ISR:
  :
  LDMFD   SP!, {R0-R12,LR,PC}^
```

This single instruction retrieves the task's registers from the new task's stack and copies the SPSR into the CPSR.

If the task was executing in Thumb mode, it will resume in Thumb mode. If the task was executing in ARM mode, it will resume in ARM mode.

	I	F	T	MODE
CPSR:	1	1	0	0x13

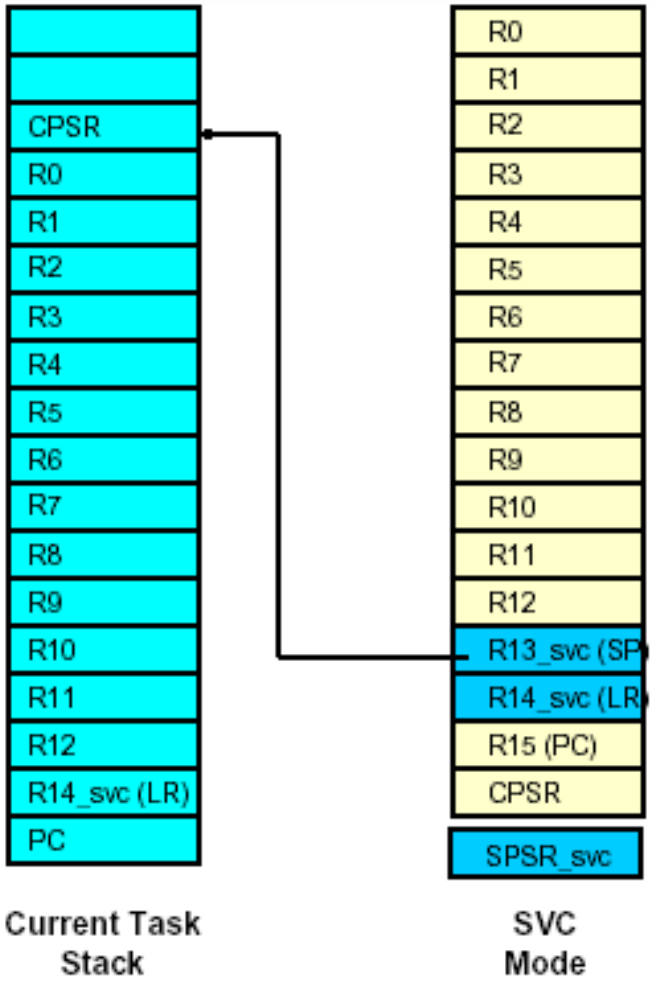
关于栈指针调整



栈指针调整

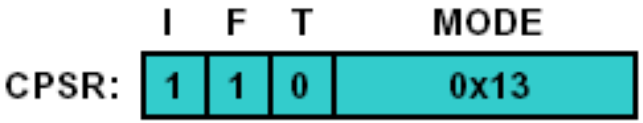
- 调整堆栈指针（加一个数在堆栈指针上）来完成的。加在堆栈指针上的数必须是明确的，而这个数主要依赖于移植的目标处理器(地址空间可能是16，32或64位)，所用的编译器，编译器选项，内存模式等等。另外，处理器状态字可能是8，16，32甚至64位宽，并且OSIntExit()可能会分配局部变量。有些处理器允许用户直接增加常量到堆栈指针中，而有些则不允许。在后一种情况下，可以通过简单的执行一定数量的pop（出栈）指令来实现相同的功能。一旦堆栈指针完成调整，新的堆栈指针会被保存到被切换出去的任务的OS_TCB中

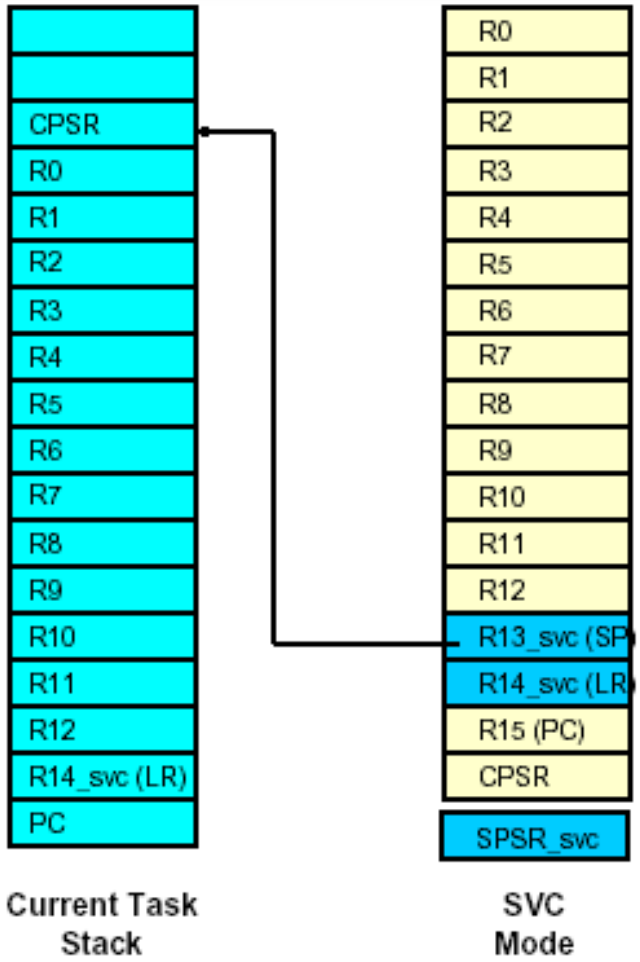
OSIntCtxSW的实现



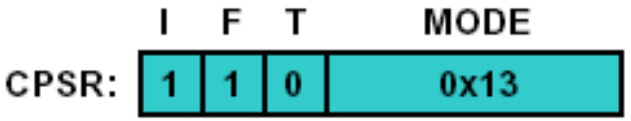
OSIntCtxSw() is called by OSIntExit() if µC/OS-II determines that there is a more important task to run than the interrupted task. In this case, the CPU is in SVC mode with interrupts disabled and the SP is pointing to the interrupted task's stack.

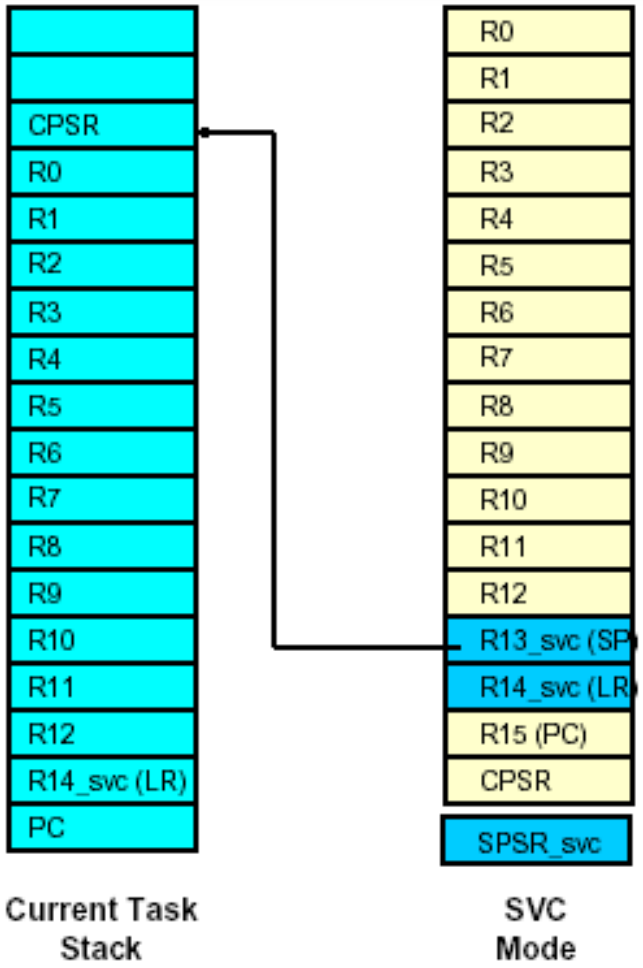
Note that the ISR has already saved the SP into the interrupted task's OS_TCB.





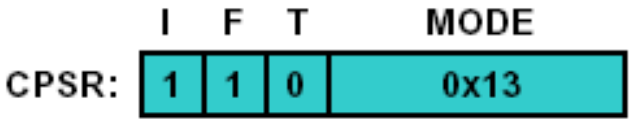

```
OSIntCtxSw:
  LDR   R0, ??OS_TaskSwHook
  MOV   LR,PC
  BX    R0

The task switch hook is called.
```

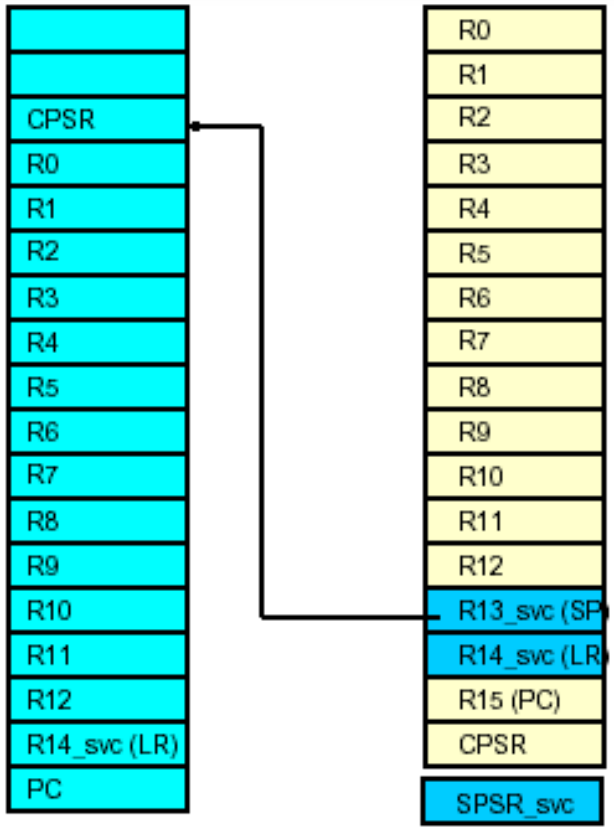





```
OSIntCtxSw:
   .
   .
;          OSPrioCur = OSPrioHighRdy
   LDR     R4,??OS_PrioCur
   LDR     R5,??OS_PrioHighRdy
   LDRB    R5,[r5]
   STRB    R5,[r4]
```

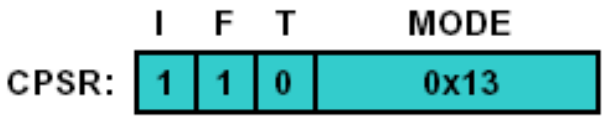
The new high priority is copied to the current priority.

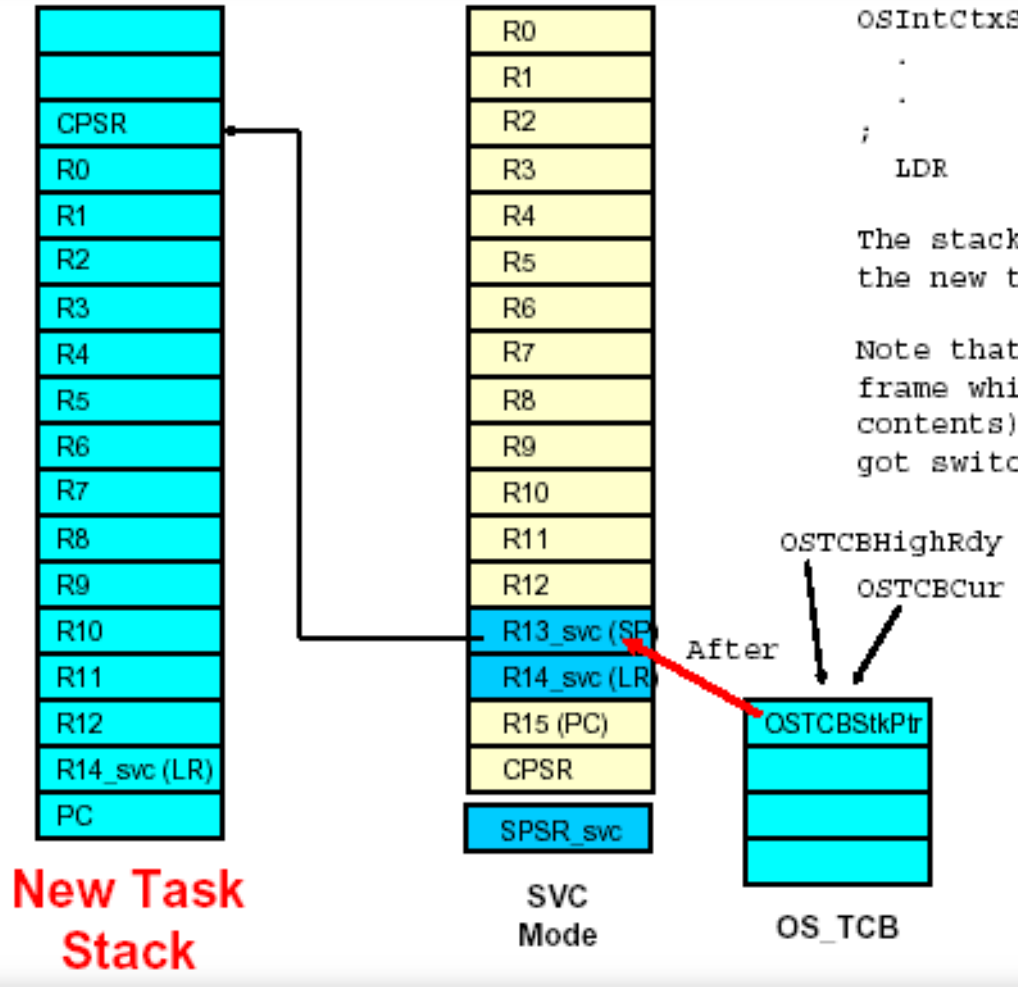





```
OSIntCtxSw:
   .
   .
;          OSTCBCur = OSTCBHighRdy
   LDR     R6,??OS_TCBHighRdy
   LDR     R4,??OS_TCBCur
   LDR     R6,[R6]
   STR     R6,[R4]
```

The pointer to the current OS_TCB is updated to point to the OS_TCB of the new task.



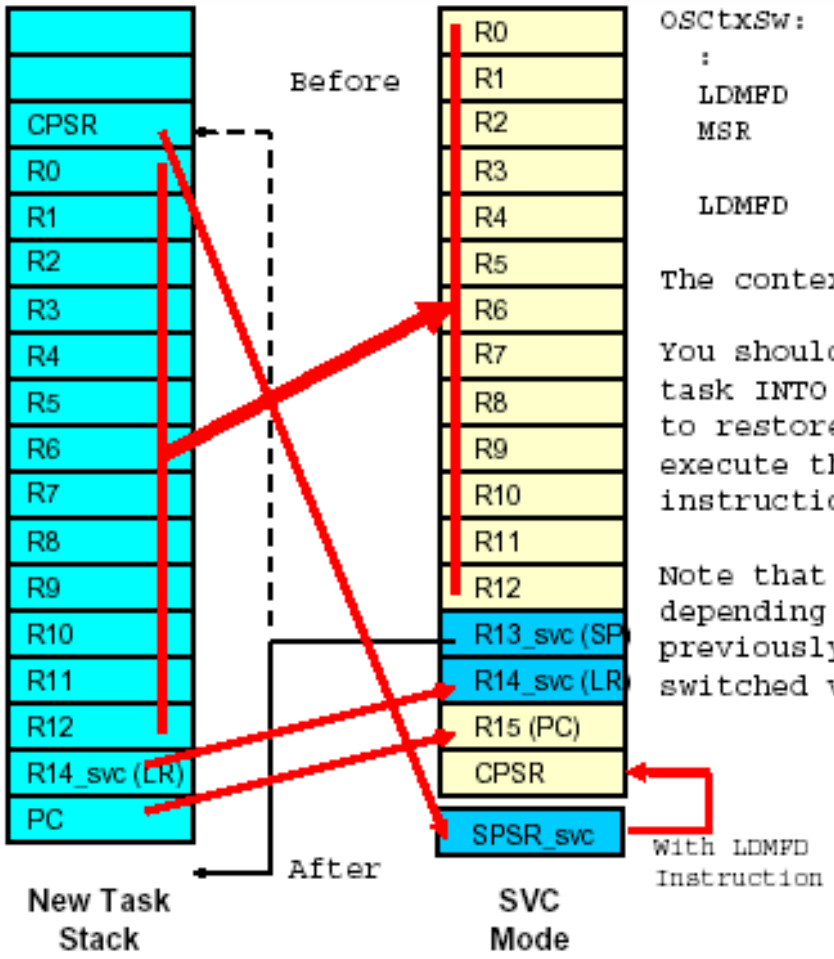



```
OSIntCtxSw:
   .
   .
;          SP = OSTCBHighRdy->OSTCBStkPtr
   LDR     SP,[R6]
```

The stack pointer is loaded from the OS_TCB of the new task.

Note that SP now points to the new task's stack frame which looks identical (except for the contents) to the stack frame of the task that got switched out.

	I	F	T	MODE
CPSR:	1	1	0	0x13

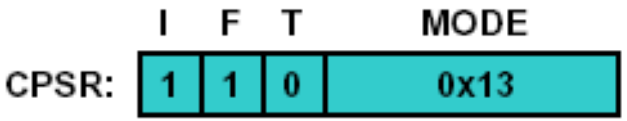



```
OSCtxSw:
  :
  LDMFD    SP!, {R4}              ; Pop new task's CPSR
  MSR      SPSR_cxsf, R4

  LDMFD    SP!, {R0-R12,LR,PC}^ ; Pop new task's context
```

The context of the new task is pulled off the stack.

You should notice that we restore the CPSR of the new task INTO the SPSR register. The reason we do this is to restore both the CPSR and PC at the same time when we execute the LDMFD instruction. After the last instruction, the CPU resumes the new task.

Note that the interrupts are either enabled or disabled depending on whether we return to a task that was previously interrupted or, a task that was context switched via OSCtxSw().



OSIntCtxSw()的原型

```
void OSIntCtxSw(void)
{
    调整堆栈指针来去掉在调用:
        OSIntExit(),
        OSIntCtxSw()过程中压入堆栈的多余内容;
    将当前任务堆栈指针保存到当前任务的OS_TCB中:
        OSTCBCur->OSTCBStkPtr = 堆栈指针;
    调用用户定义的OSTaskSwHook();
    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;
    得到需要恢复的任务的堆栈指针:
        堆栈指针 = OSTCBHighRdy->OSTCBStkPtr;
    将所有处理器寄存器从新任务的堆栈中恢复出来;
    执行中断返回指令;
}
```

OSIntCtxSW ()：中断级的任务切换函数（1）

```
OSIntCtxSw
LDR      sp, =IRQStack    ;FIQ_STACK
Sub      r7,sp,#4
mrs      r1, SPSR              ; 得到暂停的 PSR
orr      r1, r1, #0xC0         ; 关闭 IRQ, FIQ.
msr      CPSR_cxsf, r1         ; 转换模式 (应该是 SVC_MODE)

ldr      r0, [r7]    ; 从IRQ堆栈中得到IRQ's LR (任务 PC)
sub      r0, r0, #4       ; 当前PC地址是(saved_LR - 4)
STMFD  sp!, {r0}                  ; 保存任务 PC
STMFD  sp!, {lr}                  ; 保存 LR

sub      lr, r7 ,#52          ; 保存 FIQ 堆栈 ptr in LR (转到 nuke r7)
ldmfd    lr!, {r0-r12}     ; 从FIQ堆栈中得到保存的寄存器
STMFD  sp!, {r0-r12}             ;在任务堆栈中保存寄存器
```

OSIntCtxSW ()：中断级的任务切换函数（2）

```
;在任务堆栈上保存PSR 和任务 PSR
MRS      r4, CPSR
bic   r4, r4, #0xC0  ; 使中断位处于使能态
STMFD  sp!, {r4}              ; 保存任务当前 PSR
MRS      r4, SPSR
STMFD  sp!, {r4}              ; SPSR
; OSPrioCur = OSPrioHighRdy   // 改变当前程序
LDR      r4, addr_OSPrioCur
LDR      r5, addr_OSPrioHighRdy
LDRB     r6, [r5]
STRB     r6, [r4]
; 得到被占先的任务TCB
LDR      r4, addr_OSTCBCur
LDR      r5, [r4]
STR      sp, [r5]             ; 保存sp 在被占先的任务的 TCB
```

OSIntCtxSW ()：中断级的任务切换函数（3）

```
; 得到新任务 TCB 地址
LDR     r6, addr_OSTCBHighRdy
LDR     r6, [r6]
LDR     sp, [r6]            ; 得到新任务堆栈指针
; OSTCBCur = OSTCBHighRdy
STR     r6, [r4]              ; 设置新的当前任务的TCB地址
LDMFD sp!, {r4}
MSR     SPSR, r4
LDMFD sp!, {r4}
BIC   r4, r4, #0xC0         ; 必须退出新任务通过允许中断
MSR     CPSR, r4
LDMFD sp!, {r0-r12, lr, pc}
```

测试uCOS-II（1）

```
/*任务定义*/

OS_STK SYS_Task_Stack[STACKSIZE]= {0, }; //system task刷新任务
    堆栈
#define SYS_Task_Prio                                    1
void SYS_Task(void *Id);

OS_STK Task1_Stack[STACKSIZE]={0, };
void Task1(void *Id);
#define Task1_Prio     12

OS_STK Task2_Stack[STACKSIZE]= {0, };
void Task2(void *Id);
#define  Task2_Prio      13
```

测试uCOS-II（2）

```
int Main(int argc, char **argv)
{
    ARMTargetInit();        // do target (uHAL based ARM system)
    initialisation //
    OSInit();
    OSTaskCreate(SYS_Task,  (void *)0,  (OS_STK
    *)&SYS_Task_Stack[STACKSIZE-1],  SYS_Task_Prio);
    OSTaskCreate(Task2,  (void *)0,  (OS_STK
    *)&Task2_Stack[STACKSIZE-1],  Task2_Prio);
    OSTaskCreate(Task1,  (void *)0,  (OS_STK
    *)&Task1_Stack[STACKSIZE-1],  Task1_Prio  );
    OSStart();              // start the game //
    // never reached //
    return 0;
}//
```

测试uCOS-II（3）

```
void Task1(void *ld)
{
	for(;;){
		Uart_Printf("run task1\n");
		OSTimeDly(1000);
	}
}

void Task2(void *ld)
{
	for (;;)
	{
		Uart_Printf("run task2\n");
		OSTimeDly(2000);
	}
}

void SYS_Task(void *ld)
{
	uHALr_InstallSystemTimer();
	Uart_Printf("start system task.\n");
	for (;;)
	{
		OSTimeDly(10000);
	}
}
```

关于移植

相对于其他的嵌入式操作系统，uCOS-II的移植虽然是一个很简单的过程，但是，对于不熟悉uCOS-II的开发者，移植还是有一定难度的。

移植要点

- 定义函数OS_ENTER_CRITICAL和OS_ENTER_CRITICAL。
- 定义函数OS_TASK_SW执行任务切换。
- 定义函数OSCtxSw实现用户级上下文切换，用纯汇编实现。
- 定义函数OSIntCtxSw实现中断级任务切换，用纯汇编实现。
- 定义函数OSTickISR。
- 定义OSTaskStkInit来初始化任务的堆栈。

uC/OS的完善

- 固定的基于优先级的调度，不支持时间片，使用起来不方便。一个任务的基础上增加一个基于时间片的微型调度核
- 系统时钟中断，没有提供用户使用定时器，可以借鉴linux的定时器加以修改
- 在对临界资源的访问上使用关闭中断实现，没有使用CPU提供的硬件指令，例如测试并置位。
- 只是一个实时多任务内核，没有图形用户接口（GUI）、文件系统（FS）和TCP/IP协议栈

谢谢各位