附录 D

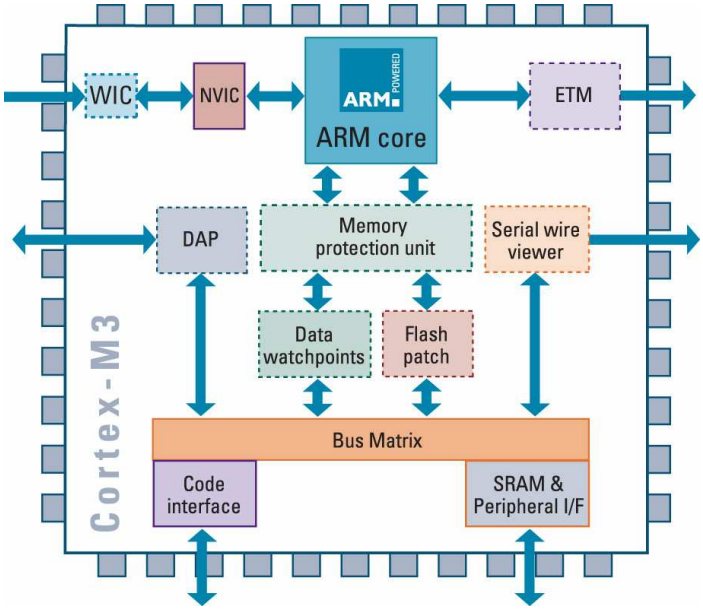
RT-Thread/STM32 说明

本文是 RT-Thread 的 STM32 移植的说明。STM32 是一款 ARM Cortex M3 芯片，本文也对 RT-Thread 关于 ARM Cortex M3 体系结构移植情况进行详细说明。

D.1 ARM Cortex M3 概况

Cortex M3 微处理器是 ARM 公司于 2004 年推出的基于 ARMv7 架构的新一代微处理器，它的速度比目前广泛使用的 ARM7 快三分之一，功耗则低四分之三，并且能实现更小芯片面积，利于将更多功能整合在更小的芯片尺寸中。

Cortex-M3 微处理器包含了一个 ARM core，内置了嵌套向量中断控制器、存储器保护等系统外设。ARM core 内核基于哈佛架构，3 级流水线，指令和数据分别使用一条总线，由于指令和数据可以从存储器中同时读取，所以 Cortex-M3 处理器对多个操作并行执行，加快了应用程序的执行速度。



Cortex-M3 微处理器是一个 32 位处理器，包括 13 个通用寄存器，两个堆栈指针，一个链接寄存器，一个程序计数器和一系列包含编程状态寄存器的特殊寄存器。Cortex-M3 微处理器的指令集则是 Thumb-2 指令，是 16 位 Thumb 指令的扩展集，可使用于多种场合。BFI 和 BFC 指令为位字段指令，在网络信息包处理等应用中可大派用场；SBFX 和 UBFX 指令

改进了从寄存器插入或提取多个位的能力，这一能力在汽车应用中的表现相当出色；RBIT 指令的作用是将一个字中的位反转，在 DFT 等 DSP 运算法则的应用中非常有用；表分支指令 TBB 和 TBH 用于平衡高性能和代码的紧凑性；Thumb-2 指令集还引入了一个新的 If-Then 结构，意味着可以有多达 4 个后续指令进行条件执行。

Cortex-M3 微处理器支持两种工作模式（线程模式（Thread）和处理模式（Handler））和两个等级的访问形式（有特权或无特权），在不牺牲应用程序安全的前提下实现了对复杂的开放式系统的执行。无特权代码的执行限制或拒绝对某些资源的访问，如某个指令或指定的存储器位置。Thread 是常用的工作模式，它同时支持享有特权的代码以及没有特权的代码。当异常发生时，进入 Handler 模式，在该模式中所有代码都享有特权。这两种模式中分别使用不同的两个堆栈指针寄存器。

Cortex-M3 微处理器的异常模型是基于堆栈方式的。当异常发生时，程序计数器、程序状态寄存器、链接寄存器和 R0 - R3、R12 四个通用寄存器将被压进堆栈。在数据总线对寄存器压栈的同时，指令总线从向量表中识别出异常向量，并获取异常代码的第一条指令。一旦压栈和取指完成，中断服务程序或故障处理程序就开始执行。当处理完毕后，前面压栈的寄存器自动恢复，中断了的程序也因此恢复正常的执行。由于可以在硬件中处理堆栈操作，Cortex-M3 处理器免去了在传统的 C 语言中断服务程序中为了完成堆栈处理所要编写的汇编代码。

Cortex-M3 微处理器内置的中断控制器支持中断嵌套（压栈），允许通过提高中断的优先级对中断进行优先处理。正在处理的中断会防止被进一步激活，直到中断服务程序完成。而中断处理过程中，它使用了 tail-chaining 技术来防止当前中断和和未决中断处理之间的压出栈。

D.2 ARM Cortex M3 移植要点

ARM Cortex M3 微处理器可以说是和 ARM7TDMI 微处理器完全不同的体系结构，在进行 RT-Thread 移植时首先需要把线程的上下文切换移植好。

通常的 ARM 移植，RT-Thread 需要手动的保存当前模式下几乎所有寄存器，R0 – R13，LR，PC，CPSR，SPSR 等。在 Cortex M3 微处理器中，

代码 D - 1 线程切换代码

```
; rt_base_t rt_hw_interrupt_disable();
; 关闭中断
rt_hw_interrupt_disable    PROC
    EXPORT  rt_hw_interrupt_disable
    MRS     r0, PRIMASK                 ; 读出PRIMASK值，即返回值
    CPSID   I                           ; 关闭中断
    BX      LR
    ENDP

; void rt_hw_interrupt_enable(rt_base_t level);
; 恢复中断
rt_hw_interrupt_enable    PROC
    EXPORT  rt_hw_interrupt_enable
    MSR     PRIMASK, r0                 ; 恢复R0寄存器的值到PRIMASK中
```

```
    BX      LR
    ENDP

; void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; r0 --> from
; r1 --> to
; 上下文切换函数；在Cortex M3中，由于采用的上下文切换方式都是在Handler模式中处理，
; 上下文切换统一使用原来RT-Thread中断中切换的方式来处理
rt_ hw_context_switch    PROC
    EXPORT rt_hw_context_switch
    LDR     r2, =rt_interrupt_from_thread    ; 保存切换出线程栈指针
    STR     r0, [r2]                         ; （切换过程中需要更新到当前位置）

    LDR     r2, =rt_interrupt_to_thread      ; 保存切换到线程栈指针
    STR     r1, [r2]

    LDR     r0, =NVIC_INT_CTRL
    LDR     r1, =NVIC_PENDSVSET
    STR     r1, [r0]                         ; 触发PendSV异常
    CPSIE   I                                ; 使能中断以使PendSV能够正常处理
    BX      LR
    ENDP

; void rt_hw_context_switch_to(rt_uint32 to);
; r0 --> to
; 切换到函数，仅在第一次调度时调用
rt_hw_context_switch_to    PROC
    EXPORT rt_hw_context_switch_to
    LDR     r1, =rt_interrupt_to_thread      ; 设置切换到线程
    STR     r0, [r1]

    LDR     r1, =rt_interrupt_from_thread    ; 设置切换出线程栈为0
    MOV     r0, #0x0
    STR     r0, [r1]

    LDR     r0, =NVIC_SYSPRI2                ; 设置优先级
    LDR     r1, =NVIC_PENDSV_PRI
    STR     r1, [r0]

    LDR     r0, =NVIC_INT_CTRL
    LDR     r1, =NVIC_PENDSVSET
    STR     r1, [r0]                         ; 触发PendSV异常
    CPSIE   I                                ; 使能中断以使PendSV能够正常处理
    ENDP

; 在异常处理过程中发生线程切换时的上下文处理函数
rt_hw_context_switch_interrupt    PROC
    EXPORT rt_hw_context_switch_interrupt
    LDR r2, =rt_thread_switch_interrput_flag
    LDR r3, [r2]
    CMP r3, #1
    BEQ _reswitch                          ; 中断中切换标识已置位，则跳到_reswitch
    MOV r3, #1                             ; 设置中断中切换标识
    STR r3, [r2]
    LDR r2, =rt_interrupt_from_thread      ; 设置切换出线程栈指针
    STR r0, [r2]
_reswitch
    LDR r2, =rt_interrupt_to_thread        ; 设置切换到线程栈指针
    STR r1, [r2]
    BX      lr
```

```
    ENDP

; 中断结束后是否进行线程上下文切换函数
rt_hw_interrupt_thread_switch    PROC
    EXPORT rt_hw_interrupt_thread_switch
    LDR     r0, =rt_thread_switch_interrput_flag
    LDR     r1, [r0]
    CBZ     r1, _no_switch       ; 如果中断中切换标志未置位，直接返回

    MOV     r1, #0x00            ; 清楚中断中切换标志
    STR     r1, [r0]

    ; 触发PendSV异常进行上下文切换
    LDR    r0, =NVIC_INT_CTRL
    LDR    r1, =NVIC_PENDSVSET
    STR    r1, [r0]

_no_switch
    BX      lr

; PendSV异常处理
; r0 --> swith from thread stack
; r1 --> swith to thread stack
; psr, pc, lr, r12, r3, r2, r1, r0 are pushed into [from thread] stack
rt_hw_pend_sv   PROC
    EXPORT rt_hw_pend_sv
    LDR     r0, =rt_interrupt_from_thread
    LDR     r1, [r0]
    CBZ     r1, swtich_to_thread     ; 如果切换出线程栈为零，直接切换到切换到线程

    MRS     r1, psp                  ; 获得切换出线程栈指针
    STMFD   r1!, {r4 - r11}          ; 对剩余的R4 – R11寄存器压栈
    LDR     r0, [r0]
    STR     r1, [r0]                 ; 更新切换出线程栈指针

swtich_to_thread
    LDR     r1, =rt_interrupt_to_thread
    LDR     r1, [r1]
    LDR     r1, [r1]                 ; 载入切换到线程的栈指针到R1寄存器

    LDMFD   r1!, {r4 - r11}          ; 恢复R4 – R11寄存器
    MSR     psp, r1                  ; 更新程序栈指针寄存器

    ORR     lr, lr, #0x04            ; 构造LR以返回到Thread模式
    BX      lr                       ; 从PendSV异常中返回
    ENDP
```

线程切换的过程可以用来**图 D - 1 正常模式下的线程上下文切换**表示。

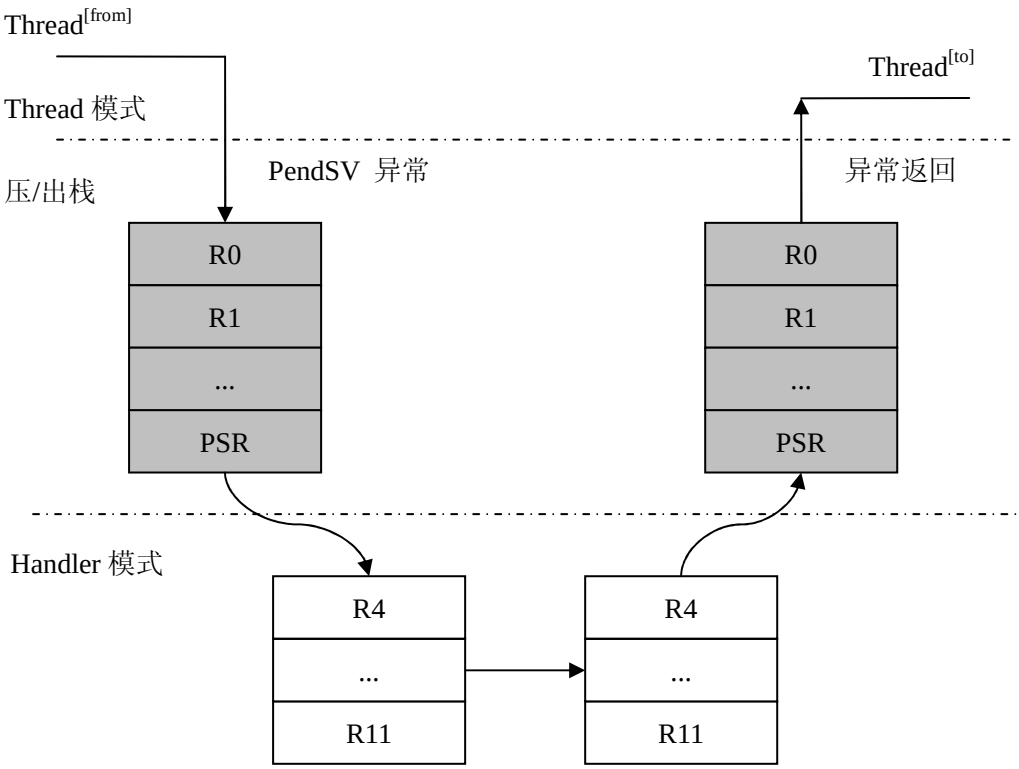


图 D - 1 正常模式下的线程上下文切换

当要进行切换时（假设从 Thread [from] 切换到 Thread [to]），通过 rt_hw_context_switch 函数触发一个 PenSV 异常。异常产生时，Cortex M3 会把 PSR，PC，LR，R0 – R3，R12 压入当前线程的栈中，然后切换到 PenSV 异常。到 PenSV 异常后，Cortex M3 工作模式切换到 Handler 模式，由函数 rt_hw_pend_sv 进行处理。rt_hw_pend_sv 函数会载入切换出线程和切换到线程的栈指针，如果切换出线程的栈指针是 0 那么表示这是第一次线程上下文切换，不需要对切换出线程做压栈动作。如果切换出线程栈指针非零，则把剩余未压栈的 R4 – R11 寄存器依次压栈；然后从切换到线程栈中恢复 R4 – R11 寄存器。当从 PendSV 异常返回时，PSR，PC，LR，R0 – R3，R12 等寄存器由 Cortex M3 自动恢复。

因为中断而导致的线程切换可用**图 D - 2 中断中线程上下文切换**表示。

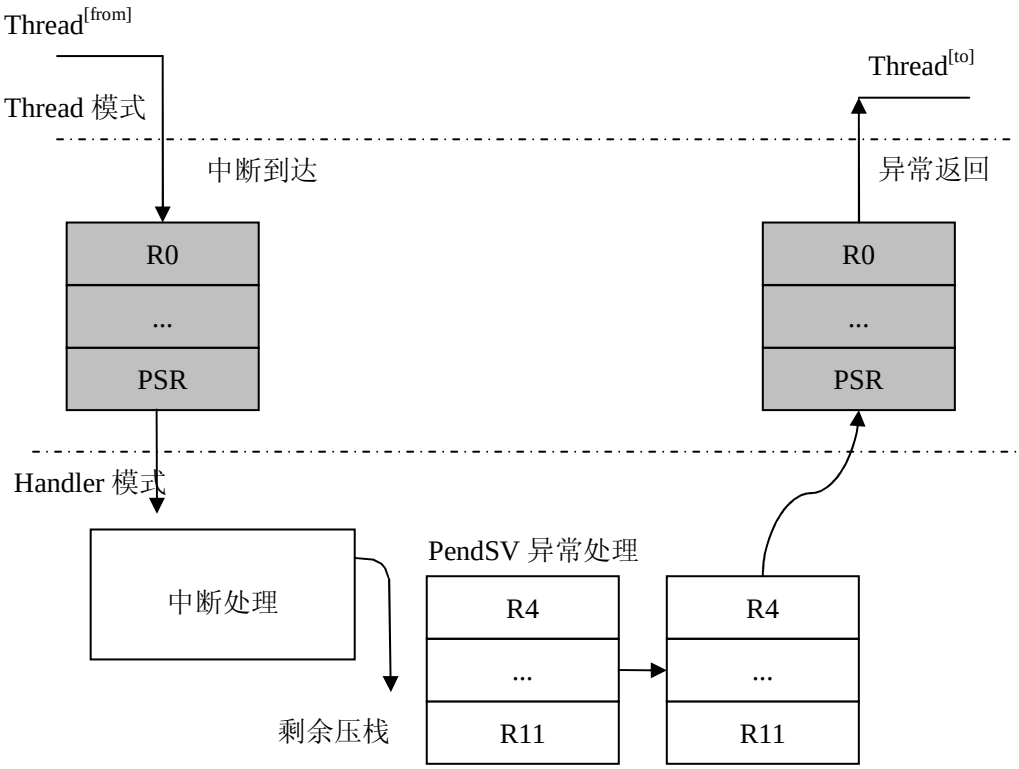


图 D - 2 中断中线程上下文切换

当中断达到时，当前线程会被中断并把 PC，PSR，R0 – R3 等压到当前线程栈中，工作模式切换到 Handler 模式。

在运行中断服务例程时，如果发生了线程切换（调用 rt_schedule），会先判断当前工作模式是否是 Handler 模式（依赖于全局变量 rt_interrupt_nest），如果是调用 rt_hw_context_switch_interrupt 函数进行伪切换。

在 rt_hw_context_switch_interrupt 函数中，将把当前线程栈指针赋值到 rt_interrupt_from_thread 变量上，把要切换过去的线程栈指针赋值到 rt_interrupt_to_thread 变量上，并设置中断中线程切换标志 rt_thread_switch_interrput_flag 为 1。

在最后一个中断服务例程结束时，会去检查中断中线程切换标志是否置位，如果置位则触发一个 PendSV 异常。PendSV 异常会在最后进行处理。

D.3 RT-Thread/STM32 说明

RT-Thread/STM32 移植是基于 RealView MDK 开发环境进行移植的，和 STM32 相关的代码大多采用 RealView MDK 中的代码，例如 start_rvds.s 是从 RealView MDK 自动添加的启动代码中修改而来。

和 RT-Thread 以往的 ARM 移植不一样的是，系统底层提供的 rt_hw_系列函数相对要少些，

建议可以考虑使用一些成熟的库。RT-Thread/STM32 工程中已经包含了 STM32f10x 系列的库代码，可以酌情使用。

和中断相关的 rt_hw_函数（RT-Thread 编程指南第 5 章大多数函数）本移植中并不具备，可以直接操作硬件。在编写中断服务例程时，如果中断服务例程可能导致线程切换请使用如下模式来编写（尽管有时并不会导致线程切换，但这里还是推荐使用此种方式编写中断服务例程）：

代码 D - 2 中断服务例程模板

```
void rt_hw_interrupt_xx_handler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();

    /* do interrupt service routine */

    /* leave interrupt */
    rt_interrupt_leave();
    rt_hw_interrupt_thread_switch();
}
```

D.4 RT-Thread/STM32 移植默认配置参数

- 线程优先级支持，32 优先级
- 内核对象支持命名，4 字符
- 操作系统节拍单位，10 毫秒
- 支持钩子函数
- 支持信号量、互斥锁
- 支持事件、邮箱、消息队列
- 支持内存池，不支持 RT-Thread 自带的动态堆内存分配器