FUJITSU

MCU-AN-510003-C-10

FM3

32 位微处理器

MB9B500 系列

RT-THREAD 实时操作系统

在 MB9BF500R 上的移植

应用指南

关于FM3 微处理器的相关资料，可访问以下网址:
http://www.fujitsu.com/global/services/microelectronics/product/micom/roadmap/industrial/fm3/

ARM™ ARM 和 Cortex-M3 是 ARM 公司在欧盟和其它国家的注册商标。

ALL RIGHTS RESERVED

本手册的记载内容如有变动，恕不另行通知。

建议用户订购前先咨询销售代表。

本手册记载的信息仅作参考，诸如功能概要和应用电路示例，旨在说明FUJITSU SEMICONDUCTOR半导体器件的使用 方法和操作示例。对于建立在该信息基础上的器件使用，FUJITSU SEMICONDUCTOR不保证器件的正常工作。如果用户根据该信息在开发产品中使用该器件，用户应对该信息的使用负责。基于上述信息的使用引起的任何损失，FUJITSU SEMICONDUCTOR概不承担任何责任。

本手册内的任何信息，包括功能介绍和原理图，不应理解为使用和执行任何知识产权的许可，诸如专利权或著作权，或FUJITSU SEMICONDUCTOR的其他权利或第三方权利，FUJITSU SEMICONDUCTOR也不保证使用该信息不侵犯任 何第三方知识产权或其他权利。因使用该信息引起的第三方知识产权或其他权利的侵权行为，FUJITSU SEMICONDUCTOR不承担任何责任。

本手册所介绍的产品旨在一般用途而设计、开发和制造，包括但并不限于一般的工业使用、通常办公使用、个人使用和家庭使用。在以下设计、开发和制造(1)使用中伴随着致命风险或危险，若不加以特别高度安全保障，有可能导致对公众产生危害，甚至直接死亡、人身伤害、严重物质损失或其他损失(即核设施的核反应控制、航空飞行控制、空中交通控制、公共交通控制、医用维系生命系统、核武器系统的导弹发射控制)，(2)需要极高可靠性的应用领域(比如海底中转器和人造卫星)。

注意上述领域内使用该产品引起的用户和/或第三方的任何索赔或损失，FUJITSU SEMICONDUCTOR不承担任何责任。

半导体器件存在一定的故障发生概率。请用户对器件和设备采取冗余设计、消防设计、过电流等级防护措施，其他异常操作防护措施等安全设计，保证即使半导体器件发生故障的情况下，也不会造成人身伤害、社会损害或重大损失。

本手册内记载的任何产品的出口/发布可能需要根据日本外汇及外贸管理法和/或美国出口管理法条例办理必要的手续。

本手册内记载的公司名称和商标名称是各个公司的商标或注册商标

修改记录

日期	版本	修改记录
2011-04-01	V1.0	第一版

目录

1 概述

该应用指南介绍**RT-Thread**在FM3简易开发套件上的'官方'移植。图 1-1的框图，列出了应用、**RT-Thread**、移植代码和BSP(板级支持包)的关系。可参考该图查阅本指南相关章节的内容。

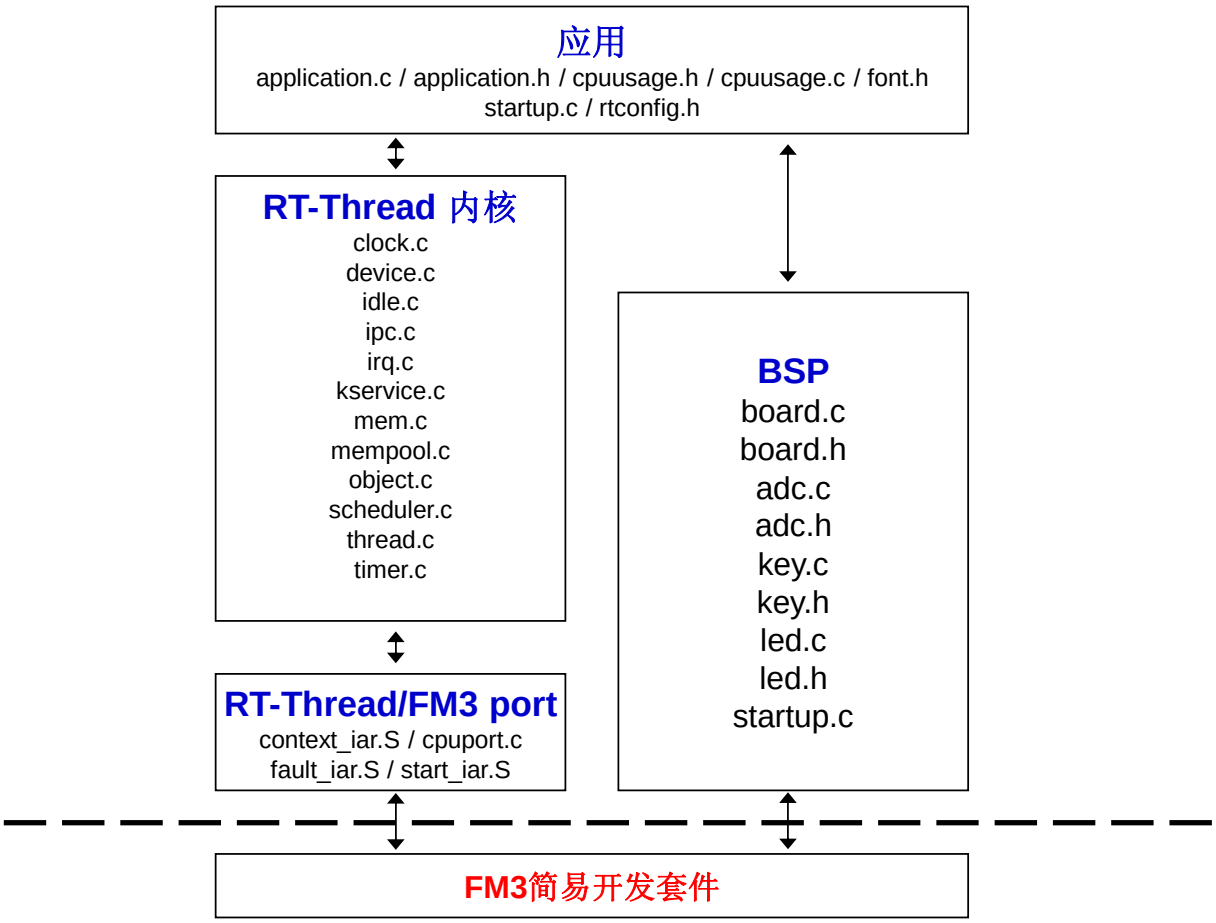

图 1-1 模块关系图

2 ARM Cortex-M3 处理器模型

ARM Cortex-M3 处理器的寄存器如图 2-1 所示。ARM Cortex-M3 一共有 20 个寄存器。每个寄存器 32 位宽。

R0 - R12	R0 到 R12 是通用寄存器，用于保存数据及指针。
R13	通常指定为堆栈指针(也称作 SP)，也可用于接收算术运算。有两个堆栈指针(SP_process 和 SP_main)，但只有一个有效。SP_process 用于任务级代码，SP_main 用于异常中断处理。
R14	称作连接寄存器(LR)，执行跳转指令(BL)时用于存储 PC 内容。LR 用于使子程序返回调用者。
R15	专用作程序计数器(PC)，并指向当前正在执行的指令。指令执行时，依据指令的不同，PC 以 2 或 4 递增。

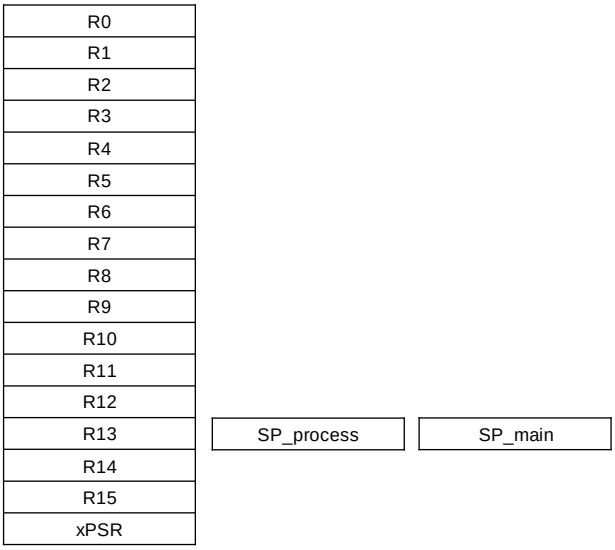

图 2-1 ARM Cortex-M3 寄存器模型

xPSR　　以下三个单独的寄存器用于保持 CPU 的状态: APSR, IPSR 和 EPSR。

APSR 包含如图 2-2 所示的应用状态。

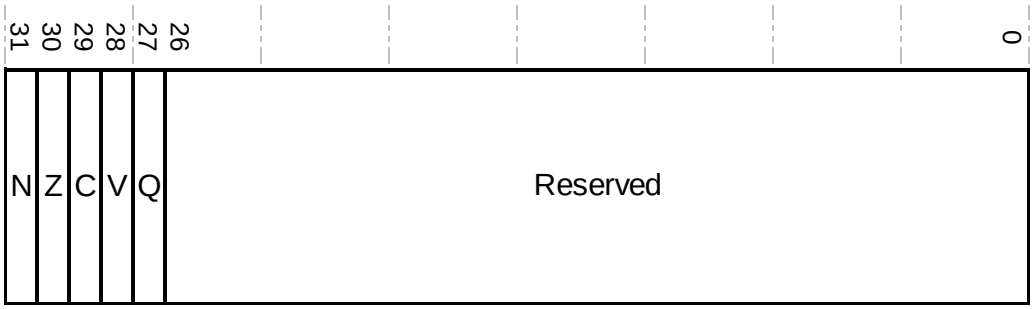

图 2-2 APSR 寄存器

N

Bit 31 是‘负数标志’位，最后一次 ALU 操作产生结果是负数时进行设置(也就是 32 位结果的最高位是 1)。

Z

Bit 30 是‘零标志’位，最后一次 ALU 操作产生结果是零时进行设置(32 位结果的每个位都是零。)。

C

Bit 29 是‘进位标志’位，不论是由于 ALU 算术运算的结果还是由于移位器动作引发的结果，最后一次 ALU 操作产生结果是进位输出时进行设置。

V

Bit 28 是‘溢出标志’位，最后一次 ALU 操作在符号位产生溢出时进行设置。

Q

Bit 27 是”饱和’标志。

图 2-3 所示，中断 PSR(**IPSR**)包含当前异常中断启动的 ISR 号。

图 2-3 IPSR 寄存器

异常中断 PSR (**EPSR**)含有两个异常中断区域:

- 可被中断 (ICI)的区域，用于可被中断的多次载入指令和多次存储指令
- 执行状态区域，用于 If-Then (IT)指令和 T-bit (Thumb 状态位)

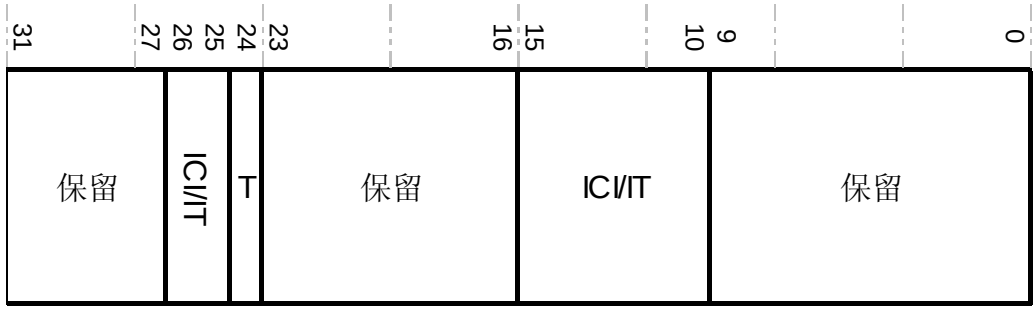

图 2-4 EPSR 寄存器

进入异常中断后，存储器把三个状态寄存器(称为 xPSR)的综合信息保存至堆栈。

3 FM3 (MB9B500)处理器的RT-Thread移植

使用 IAR EWARM V6.10 (ARM 嵌入式集成开发环境)测试移植。EWARM 包括一个编辑器、一个 C/C++编译器、一个汇编器、一个连接器/定址器和 C-Spy 调试器。C-Spy 调试器通常有一个 ARM Cortex-M3 仿真器，该仿真器可容许工作人员在实际运行硬件前来测试代码。

该移植基于FM3简易开发套件(如图3-1所示的FSSDC-9B506-EK)设计，简化了外围设备，用于研究MB9BF500特性。

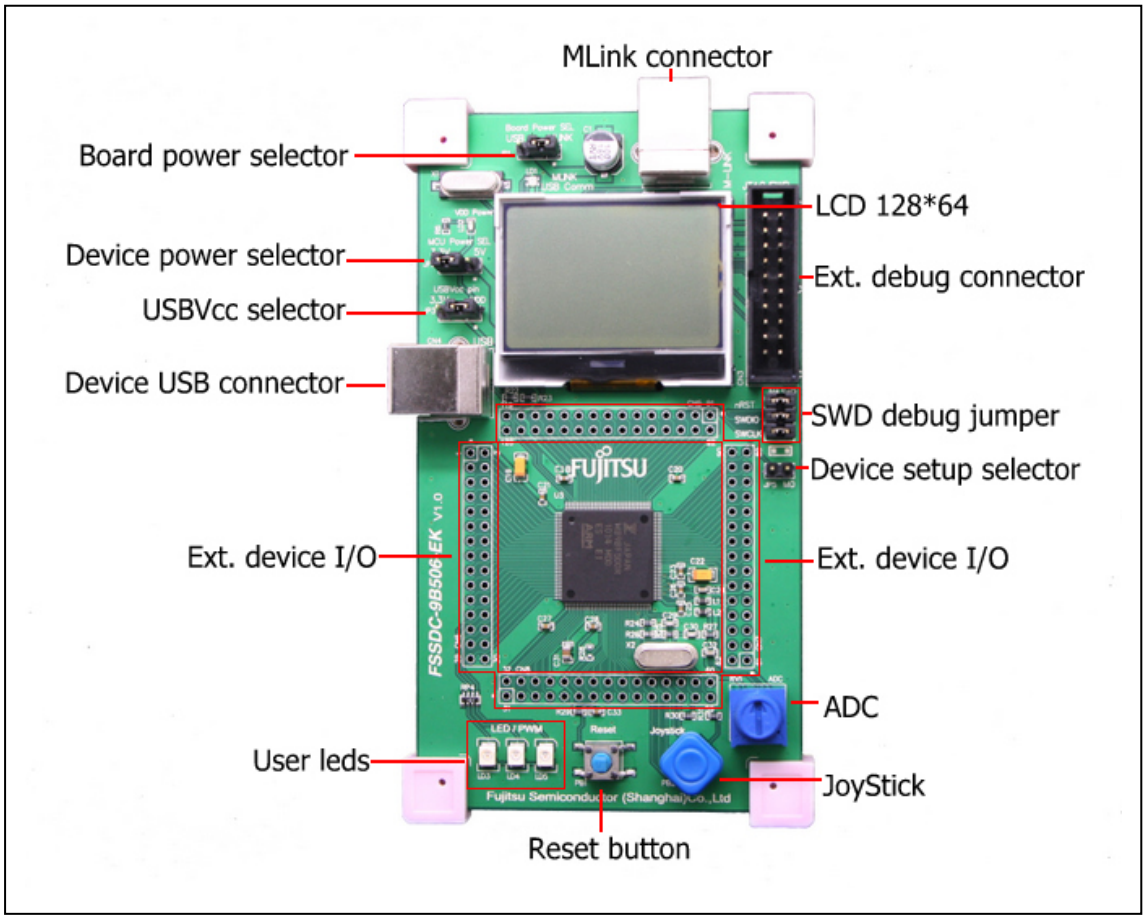

图 3-1 FSSDC-9B506-EK

调整应用指南中的移植可应用于其它基于ARM Cortex-M3编译器。指令(也就是代码)是一样的，只需调整移植以适合不同的编译器规格。

该移植需要使用RT-Thread v0.4.0或者更高版本。

3.1 目录和文件

可在以下目录中找到与本应用指南配套的软件:

fm3_rt_thread\

可在以下文件中 fm3_rt_thread \RT-Thread\fm3 目录下找到本移植的源代码:

start_iar.S

context_iar.S

fault_iar.S

cpuport.c

可在相应的目录中找到测试代码和构成文件并在下面加以说明。

3.2 start_iar.S (异常中断向量表)

ARM cortex-M3 包含异常中断向量表(也称作中断向量表)，其地址起于 0x0000 0000。该表的条目多达 256 条(每个条目是一个 32 位的指针，所以高达 1KB)。表中的每个条目是对应相应异常中断或者中断处理程序的指针。

ARM Cortex-M3 的异常中断向量表如表 3-2-1 所示:

位置	异常/中断	优先级	向量地址
0		\	0x0000 0000
1	复位	-3 (最高)	0x0000 0004
2	不可屏蔽中断	-2	0x0000 0008
3	硬件故障	-1	0x0000 000C
4	内存管理	可设	0x0000 0010
5	总线故障	可设	0x0000 0014
6	使用故障	可设	0x0000 0018
7	保留	\	0x0000 001C
8	保留	\	0x0000 0020
9	保留	\	0x0000 0024
10	保留	\	0x0000 0028
11	SVCall	可设	0x0000 002C
12	调试监视器	可设	0x0000 0030
13	保留	\	0x0000 0034
14	**PendSV**	可设	0x0000 0038
15	**SysTick**	可设	0x0000 003C
16	INTSIR[239]	可设	0x0000 0040
17	INTSIR[238]	可设	0x0000 0044
…	…	可设	…
255	INTSIR[0]	可设	0x0000 03FC

表 3-2-1 ARM Cortex-M3异常中断向量表

RT-Thread 使用 PendSV 处理程序切换上下文，使用 SysTick 处理程序处理系统节拍(时钟节拍)。PendSV 处理程序可禁止中断，中断因此可以自动执行。

ARM Cortex-M3 有一个内置的定时器，专为 RTOS 使用而设计。在任何节拍率下可配置定时器。该应用的 BSP 应把定时器设置成 RT_TICK_PER_SECOND。

注意：由应用代码决定异常中断向量表的设置。每项工程中用户对 start_iar.S 中的异常中断向量表进行编辑，以协助用户完成任务。

```
__vector_table
        DCD     sfe(CSTACK)
        DCD     __iar_program_start
        DCD     NMI_Handler               ; NMI Handler
        DCD     rt_hw_hard_fault          ; Hard Fault Handler
        DCD     MemManage_Handler         ; MPU Fault Handler
        DCD     BusFault_Handler          ; Bus Fault Handler
        DCD     UsageFault_Handler        ; Usage Fault Handler
        DCD     0                         ; Reserved
        DCD     0                         ; Reserved
        DCD     0                         ; Reserved
        DCD     0                         ; Reserved
        DCD     SVC_Handler               ; SVCall Handler
        DCD     DebugMon_Handler          ; Debug Monitor Handler
        DCD     0                         ; Reserved
        DCD     rt_hw_pend_sv             ; PendSV Handler
        DCD     rt_hw_timer_handler       ; SysTick Handler
```

3.2.1 异常中断/中断处理顺序

调用异常中断或者中断处理程序时，CPU 会自动把 xPSR, PC, LR, R12 及 R0-R3 寄存器压入 SP_process 堆栈。

此时，CPU 读取向量表可以摘取异常中断/中断处理程序的地址，还能使用该地址更新 PC。CPU 创建的异常中断堆栈帧包括旧 PC。实际上 LR 得到一个譬如 0xFFFF FFF9 的特殊值。这说明它处于处理模式，当 CM-3 发现该值试图载入 PC (正如在 BX LR)时，它把该动作识别为进入异常中断，当返回异常中断时，它把之前寄存器保存的 PC 值载入到 PC。CPU 转而使用 SP_main 堆栈指针。

3.2.2 中断控制器

ARM Cortex-M3 还有一个集成的嵌套向量中断控制器 (NVIC)。

3.2.3 中断服务例程

需要使用 RT-Thread 服务的中断服务例程(ISRs)应如富士通 FM3 的列表 3-2-2 一样编写。

列表 3-2-2 使用 RT-Thread 服务的中断服务例程

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

        /* handle the interrupt */

        /* leave interrupt */
        rt_interrupt_leave();
}

void rt_interrupt_enter()
{
        rt_base_t level;

        level = rt_hw_interrupt_disable();
        rt_interrupt_nest ++;
        rt_hw_interrupt_enable(level);
}

void rt_interrupt_leave()
{
        rt_base_t level;

        level = rt_hw_interrupt_disable();
        rt_interrupt_nest --;
        rt_hw_interrupt_enable(level);
}
```

3.3 context_iar.S

RT-Thread 移植要求编写相对简单的汇编函数:

rt_hw_interrupt_disable()

rt_hw_interrupt_enable()

rt_hw_context_switch()

rt_hw_context_switch_interrupt()

rt_hw_context_switch_to()

3.3.1 rt_hw_interrupt_disable() and rt_hw_interrupt_enable()

使用 rt_hw_interrupt_disable()函数禁止中断。中断禁止意味着其它事件(整个系统不会对应外部事件)不会中断当前的任务或者代码，还不会抢占当前任务。

列表 3-3-1 context_iar.S, rt_hw_interrupt_disable()

```
;/*
; * rt_base_t rt_hw_interrupt_disable();
; */
        EXPORT rt_hw_interrupt_disable
rt_hw_interrupt_disable:
        MRS          r0, PRIMASK
        CPSID        I
        BX           LR
```

rt_hw_interrupt_enable()用于恢复状态，经常和 rt_hw_interrupt_disable()成对使用。调用该函数意味着不会使能中断，但会把机器恢复到调用 rt_hw_interrupt_disable()前的状态。如果调用 rt_hw_interrupt_disable()前已经禁止中断状态，那么在调用 rt_hw_interrupt_enable()后，中断状态依旧禁止。

列表 3-3-2 context_iar.S, rt_hw_interrupt_enable()

```
;/*
; * void rt_hw_interrupt_enable(rt_base_t level);
; */
        EXPORT  rt_hw_interrupt_enable
rt_hw_interrupt_enable:
        MSR             PRIMASK, r0
        BX              LR
```

3.3.2 rt_hw_context_switch_interrupt()和rt_hw_conext_switch()

在富士通 FM3 移植中，这两个函数相同，这是因为 PendSV 异常也触发正常的上下文切换。

```
;/*
; * void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; * r0 --> from
; * r1 --> to
; */
        EXPORT rt_hw_context_switch_interrupt
        EXPORT rt_hw_context_switch
rt_hw_context_switch_interrupt:
rt_hw_context_switch:
        ; set rt_thread_switch_interrput_flag to 1
        LDR     r2, =rt_thread_switch_interrput_flag
        LDR     r3, [r2]
        CMP     r3, #1
        BEQ     _reswitch
        MOV     r3, #1
        STR     r3, [r2]

        LDR     r2, =rt_interrupt_from_thread     ; set rt_interrupt_from_thread
        STR     r0, [r2]

_reswitch
        LDR     r2, =rt_interrupt_to_thread       ; set rt_interrupt_to_thread
        STR     r1, [r2]

        LDR      r0, =NVIC_INT_CTRL; trigger the PendSV exception (causes context switch)
        LDR      r1, =NVIC_PENDSVSET
        STR      r1, [r0]
        BX       LR
```

3.3.3 rt_hw_context_switch_to()

该函数仅在第一次被调度程序调用。

```
;/*
; * void rt_hw_context_switch_to(rt_uint32 to);
; * r0 --> to
; */
        EXPORT rt_hw_context_switch_to
rt_hw_context_switch_to:
        LDR             r1, =rt_interrupt_to_thread
```

```
    STR          r0, [r1]

    ; set from thread to 0
    LDR          r1, =rt_interrupt_from_thread
    MOV          r0, #0x0
    STR          r0, [r1]

    ; set interrupt flag to 1
    LDR    r1, =rt_thread_switch_interrput_flag
    MOV    r0, #1
    STR    r0, [r1]

    ; set the PendSV exception priority
    LDR     r0, =NVIC_SYSPRI2
    LDR     r1, =NVIC_PENDSV_PRI
    LDR.W   r2, [r0,#0x00]       ; read
    ORR     r1,r1,r2             ; modify
    STR     r1, [r0]             ; write-back

    LDR     r0, =NVIC_INT_CTRL       ; trigger the PendSV exception (causes context switch)
    LDR     r1, =NVIC_PENDSVSET
    STR     r1, [r0]

    CPSIE   I                        ; enable interrupts at processor level
```

3.3.4 rt_hw_pend_sv()

rt_hw_pend_sv()函数是 PendSV 异常中断处理程序，用于处理 RT-thread 的所有上下文切换。在 ARM Cortext-M3 中推荐使用该方法进行上下文切换。这是因为 ARM Cortex-M3 自动保存任何异常中断的处理器上下文的一半，并在从异常中断恢复后还原相同的寄存器。PendSV 处理程序仅需要保存 R4-R11 并调整堆栈指针。使用 PendSV 异常中断表示上下文保存和还原使用的是相同的方法，无论是由任务初始化的还是因中断或异常中断引发的。

请注意在异常中断向量表中的第 14 个位置中必须放入指向 rt_hw_pend_sv()的指针(根据向量表 + 4*14 或偏移 56)。

```
; r0 --> swith from thread stack
; r1 --> swith to thread stack
; psr, pc, lr, r12, r3, r2, r1, r0 are pushed into [from] stack
    EXPORT rt_hw_pend_sv
rt_hw_pend_sv:
    ; disable interrupt to protect context switch
    MRS    r2, PRIMASK
    CPSID  I

    ; get rt_thread_switch_interrupt_flag
    LDR    r0, =rt_thread_switch_interrput_flag
    LDR    r1, [r0]
    CBZ    r1, pendsv_exit                  ; pendsv already handled

    ; clear rt_thread_switch_interrput_flag to 0
    MOV    r1, #0x00
    STR    r1, [r0]

    LDR    r0, =rt_interrupt_from_thread
    LDR    r1, [r0]
    CBZ    r1, swtich_to_thread     ; skip register save at the first time
```

```
        MRS     r1, psp                         ; get from thread stack pointer
        STMFD   r1!, {r4 - r11}                 ; push r4 - r11 register
        LDR     r0, [r0]
        STR     r1, [r0]                ; update from thread stack pointer

swtich_to_thread
        LDR     r1, =rt_interrupt_to_thread
        LDR     r1, [r1]
        LDR     r1, [r1]                ; load thread stack pointer

        LDMFD   r1!, {r4 - r11}                 ; pop r4 - r11 register
        MSR     psp, r1                 ; update stack pointer

pendsv_exit
        ; restore interrupt
        MSR     PRIMASK, r2

        ORR     lr, lr, #0x04
        BX      lr
```

正常上下文切换如图3-3-1所示。

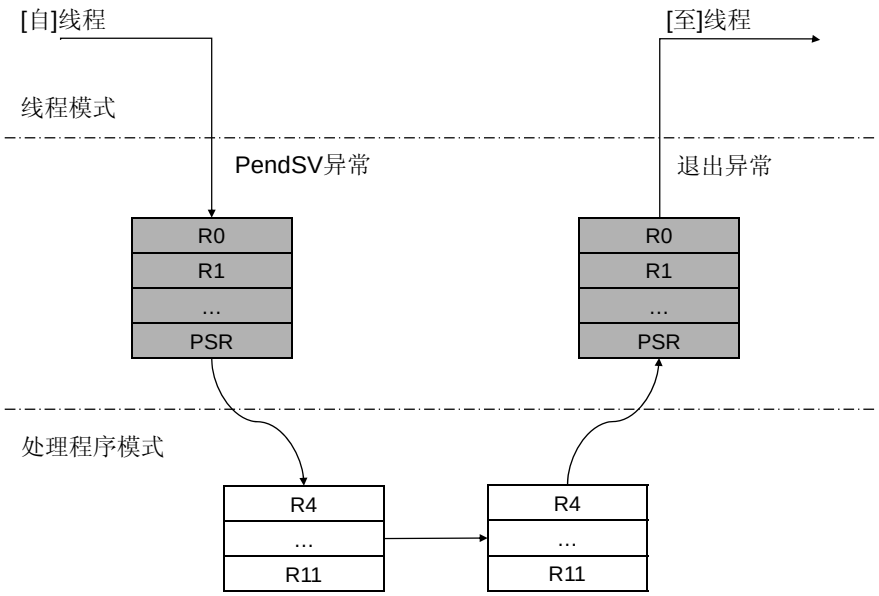

图3-3-1 正常上下文切换流程

运行上下文切换(即从[自]线程切换到[至]线程)时，使用rt_hw_context_switch()函数触发PendSV异常中断。异常中断发生时，Cortext-M3自动地将PSR、PC、LR、R0-R3和R12推入目前线程的堆栈。运行rt_hw_pend_sv()可将Cortext-M3切换到处理程序模式。rt_hw_pend_sv()函数可将[自]线程和[至]线程的堆栈指针还原。如果[自]线程的堆栈指针是0，这表示该上下文切换是第一次并且没有必要将[自]线程推入堆栈。如果[自]线程的堆栈指针不等于0，那么应该将R4-R11寄存器推入堆栈；并将R4-R11寄存器从[至]线程堆栈还原。PendSV异常中断退出时，PSR、PC、LR、R0-R3和R12寄存器将自动还原。

中断引发的上下文切换如图3-3-2所示。

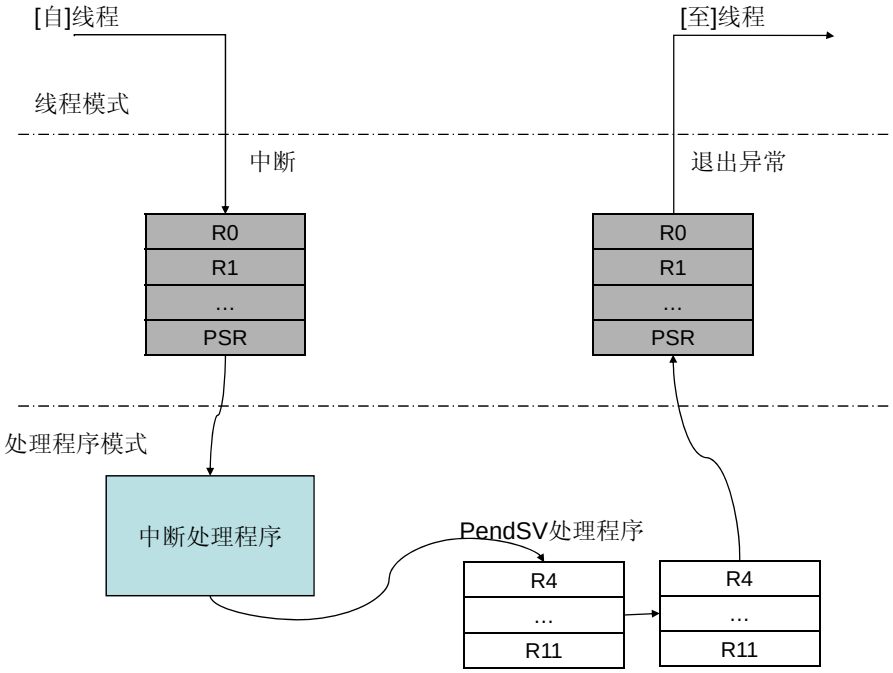

图3-3-2 中断引发的上下文切换

中断发生时，当前线程将被中断且PC、PSR、R0-R3和R12寄存器被推入当前线程的堆栈内。处理器模式切换到处理模式。

运行中断例程时，如果需要上下文切换(通过调用中断服务路径中的rt_schedule()函数)，则使用全局变数rt_interrupt_nest确认处理器是否在处理模式下。如果rt_interrupt_nest != 0，则调用rt_hw_context_switch_ interrupt()进行上下文切换。

在rt_hw_context_switch_interrupt()函数中，将当前线程的堆栈指针配置到变数rt_interrupt_from_thread，[自]线程的堆栈指针配置到变数rt_interrupt_to_thread，然后将标志rt_thread_switch_interrupt_flag置1。

最后的中断例程退出后，Cortex-M3开始处理PendSV异常中断，这是因为PendSV异常中断的优先级最低。

3.4 cpuport.c

列表 3-4-1 中的代码为创建的线程初始化堆栈框。线程收到选项参数数‘parameter’。对于 ARM 编译器(Cortex-M3 也同样)，将函数的第一个变数传递到 R0 寄存器是典型的做法。这就是为什么创建线程时，‘parameter’传递到 R0。大多数 CPU 寄存器的初始值并不重要，所以首次创建线程时，我们将寄存器初始化为 0。当然在执行线程代码时，寄存器值很可能会变化。

列表 3-4-1 cpuport.c rt_hw_stack_init()

```
/**
 * This function will initialize thread stack
 *
 * @param tentry the entry of thread
 * @param parameter the parameter of entry
 * @param stack_addr the beginning stack address
 * @param texit the function will be called when thread exit
 *
```

```
 * @return stack address
 */
rt_uint8_t *rt_hw_stack_init(void *tentry, void *parameter,
       rt_uint8_t *stack_addr, void *texit)
{
       unsigned long *stk;

       stk     = (unsigned long *)stack_addr;
       *(stk)   = 0x01000000L;                          /* PSR */
       *(--stk) = (unsigned long)tentry;                /* entry point, pc */
       *(--stk) = (unsigned long)texit;                 /* lr */
       *(--stk) = 0;                                    /* r12 */
       *(--stk) = 0;                                    /* r3 */
       *(--stk) = 0;                                    /* r2 */
       *(--stk) = 0;                                    /* r1 */
       *(--stk) = (unsigned long)parameter;             /* r0 : argument */

       *(--stk) = 0;                                    /* r11 */
       *(--stk) = 0;                                    /* r10 */
       *(--stk) = 0;                                    /* r9 */
       *(--stk) = 0;                                    /* r8 */
       *(--stk) = 0;                                    /* r7 */
       *(--stk) = 0;                                    /* r6 */
       *(--stk) = 0;                                    /* r5 */
       *(--stk) = 0;                                    /* r4 */

       /* return task's current stack address */
       return (rt_uint8_t *)stk;
}
```

4 应用

应用示例运行 6 个线程:

led1 thread -- 闪烁发光二极管 1

led2 thread --闪烁发光二极管 2

key thread -- 用户关键处理程序

adc thread -- 取得 ADC 值并发送到应用

app thread -- 应用线程

4.1 目录和文件

可在以下目录中找到与本应用指南配套的软件:

fm3_rt_thread\Example\source

可在以下文件中找到本应用的源代码:

adc.c

application.c

board.c

cpuusage.c

key.c

lcd.c

led.c

startup.c

rtconfig.h

可在相应的目录中找到测试代码和构成文件并在下面加以说明。

4.2 rtconfig.h

可通过rtconfig.h使能或禁止部分RT-thread元件以减少存储器的使用。

列表 4-2-1 rtconfig.h

```
/* RT-Thread config file */
#ifndef __RTTHREAD_CFG_H__
#define __RTTHREAD_CFG_H__

/* RT_NAME_MAX*/
#define RT_NAME_MAX       4

/* RT_ALIGN_SIZE*/
#define RT_ALIGN_SIZE     4

/* PRIORITY_MAX */
#define RT_THREAD_PRIORITY_MAX   32

/* Tick per Second */
#define RT_TICK_PER_SECOND       100
```

```
/* SECTION: RT_DEBUG */
/* Thread Debug */
#define RT_DEBUG
#define RT_USING_OVERFLOW_CHECK

/* Using Hook */
#define RT_USING_HOOK

/* SECTION: IPC */
/* Using Semaphore */
#define RT_USING_SEMAPHORE

/* Using Mutex */
#define RT_USING_MUTEX

/* Using Event */
#define RT_USING_EVENT

/* Using MailBox */
#define RT_USING_MAILBOX

/* Using Message Queue */
#define RT_USING_MESSAGEQUEUE

/* SECTION: Memory Management */
/* Using Memory Pool Management*/
#define RT_USING_MEMPOOL

/* Using Dynamic Heap Management */
#define RT_USING_HEAP

/* Using Small MM */
#define RT_USING_SMALL_MEM

/* SECTION: Device System */
/* Using Device System */
#define RT_USING_DEVICE
/* RT_USING_UART */
#define RT_USING_UART0
#define RT_UART_RX_BUFFER_SIZE   64

/* SECTION: Console options */
#define RT_TINY_SIZE
#define RT_USING_CONSOLE
/* the buffer size of console */
#define RT_CONSOLEBUF_SIZE128

/* SECTION: RTGUI support */
/* using RTGUI support */
//#define RT_USING_RTGUI

/* name length of RTGUI object */
#define RTGUI_NAME_MAX           16
/* support 16 weight font */
//#define RTGUI_USING_FONT16
/* support 12 weight font */
#define RTGUI_USING_FONT12
/* support Chinese font */
//#define RTGUI_USING_FONTHZ
/* use DFS as file interface */
//#define RTGUI_USING_DFS_FILERW
/* use font file as Chinese font */
/* #define RTGUI_USING_HZ_FILE */
/* use Chinese bitmap font */
//#define RTGUI_USING_HZ_BMP
/* use small size in RTGUI */
```

```
//#define RTGUI_USING_SMALL_SIZE
/* use mouse cursor */
/* #define RTGUI_USING_MOUSE_CURSOR */
#define RTGUI_DEFAULT_FONT_SIZE 12

#endif
```

4.3 startup.c

rtthread_startup()函数是 RT-thread 的进入点。通过查看 rtthread_startup()，可以了解 RT-thread 的启动过程。

可分为几个部分:

- 初始化硬件
- 初始化某些 RT-thread 元件，例如：定时器、调度程序…
- 初始化器件，用于初始化 RT-thread 设备框架
- 初始化应用线程并启动调度程序

列表 4-3-1 startup.c, rtthread_startup()

```
/**
 * This function will startup RT-Thread RTOS.
 */
void rtthread_startup(void)
{
	/* init board */
	rt_hw_board_init();

	/* show version */
	rt_show_version();

	/* init tick */
	rt_system_tick_init();

	/* init kernel object */
	rt_system_object_init();

	/* init timer system */
	rt_system_timer_init();

#ifdef RT_USING_HEAP
	#ifdef __CC_ARM
		rt_system_heap_init((void*)&Image$$RW_IRAM1$$ZI$$Limit,
(void*)FM3_SRAM_END);
	#elif __ICCARM__
		rt_system_heap_init(__segment_end("HEAP"), (void*)FM3_SRAM_END);
	#else
		/* init memory system */
		rt_system_heap_init((void*)&__bss_end, (void*)FM3_SRAM_END);
	#endif
#endif

	/* init scheduler system */
	rt_system_scheduler_init();

#ifdef RT_USING_DEVICE
```

```
        /* init all device */
        rt_device_init_all();
#endif

        /* init application */
        rt_application_init();

        /* init timer thread */
        rt_system_timer_thread_init();

        /* init idle thread */
        rt_thread_idle_init();

        /* start scheduler */
        rt_system_scheduler_start();

        /* never reach here */
        return ;
}
```

5 结论

本应用指南介绍了 FM3 (MB9B500 系列)处理器的移植。该移植应该可以简单地适用于不同的编译器(代码本身相似)。当然，在其他实际硬件上使用 RT-thread 移植时，必须适当地初始化并处理硬件中断。

6 使用许可

RT-thread 实时操作系统作为公开源实时操作系统在 GNU GPLv2 许可下发布。如果用户有意将 RT-thread 用于商品，请务必联系 RT-thread 组织将 GPLv2 许可转换为商用许可。感谢合作。

7 参考文献

<<RT-thread 手册>>