

# RT-Thread 移植

本章描述了 RT-Thread 的移植说明，主要包括 ARM 体系结构下采用 GNU GCC 和 RealView MDK 两种不同的编译器环境下的移植说明。











## 1.1 RT-Thread 目录结构

表格 A - 1 rtt 0.2.4 目录下的文件

名称	大小/B	最后修改时间	说明
 config/		2009-01-20 21:00:04	各种移植的配置文件及配置脚本（Python 脚本）。
 kernel/		2009-01-20 21:00:04	RT-Thread 代码（包含各个组件）
 tools/		2009-01-20 21:00:04	一些工具，例如一些模拟器。
 AUTHORS	389	2008-10-18 15:10:49	
 ChangeLog	1546	2008-10-18 15:10:49	
 COPYING	17992	2008-10-18 15:10:49	
 Makefile	548	2008-10-18 15:10:49	

表格 A - 2 kernel 目录下的文件

名称	大小/B	最后修改时间	说明
 bsp/		2009-01-20 21:00:04	板级支持包，包含 target 板的一些配置及驱动。
 cplusplus/		2009-01-20 21:00:04	C++组件
 finsh/		2009-01-20 21:00:04	finsh shell 组件

 include/		2009-01-20 21:00:04	RT-Thread kernel 头文件
 lib/		2009-01-20 21:00:04	编译产生的库文件
 libc/		2009-01-20 21:00:04	标准 C 库组件
 libcpu/		2009-01-20 21:00:04	各种 CPU 架构相关代码，一些 SoC 片内外设驱动
 net/		2009-01-20 21:00:04	TCP/IP 网络协议栈组件
 src/		2009-01-20 21:00:04	RT-Thread kernel 代码
 testsuite/		2009-01-20 21:00:04	测试代码以及一些例子代码
 config.mk	3216	2008-10-18 15:10:49	
 config.target	850	2008-10-18 15:10:49	
 Makefile	844	2008-10-18 15:10:49	

如表格 A - 1 rtt 0.2.4 目录下的文件和表格 A - 2 kernel 目录下的文件所示，和 RT-Thread 移植相关的主要包含两个目录：**bsp** 和 **libcpu**

**bsp** — 放置的是和具体板子相关的文件；

**libcpu** — 放置的具体的 CPU/MCU 相关文件，由 CPU/MCU 将决定整个系统架构。

这两个目录都是和移植、硬件驱动相关的，都可能包含硬件驱动，但不同的是，和 CPU/MCU 相关的驱动都放在 **libcpu** 中，即不同的目标板但是具备相同的 CPU/MCU，可以共享同一份 **libcpu** 中的移植代码。**bsp** 目录是和具体开发板密切相关的，例如 **memory** 的布局，开发板上通过 IO 接口等外扩的设备等驱动文件将放在这里，同时链接脚本也会放在这里（链接脚本通常是和整个 **memory** 布局是密切相关的）。

**libcpu** 目录下是各个具体芯片相关的实现，但其分类首先是按照体系结构来分类的，例如 **arm**, **ia32** 等等，再下面才是具体的芯片描述，如 **ARM7TDMI** 中的 **s3c4510**, **m68k** 中的 **coldfire** 系列芯片等。

## 1.2 ARM 编程模式

本文讲述的是 RT-Thread 的 ARM 移植，分别介绍 **s3c4510** 在 GNU GCC 环境下的移植和

AMTEL AT91SAM7S64 在 RealView MDK 环境下的移植。这两款芯片都是采用 ARM7TDMI 架构的芯片。

### 1.2.1 ARM 的工作状态

从编程的角度看，ARM微处理器的工作状态一般有两种，ARM状态，此时处理器执行32位的字对齐的ARM指令；Thumb状态，此时处理器执行16位的、半字对齐的Thumb指令。

RT-Thread也支持Thumb模式运行，其中汇编代码运行于ARM状态，C代码运行于Thumb状态。

### 1.2.2 ARM 处理器模式

ARM微处理器支持7种运行模式，分别为：

模式	说明
用户模式（usr）	ARM处理器正常的程序执行状态。
快速中断模式（fiq）	用于高速数据传输或通道处理。
外部中断模式（irq）	用于通用的中断处理。
管理模式（svc）	操作系统使用的保护模式，系统复位后的缺省模式。
指令终止模式（abt）	当指令预取终止时进入该模式。
数据访问终止模式（abt）	当数据访问终止时进入该模式。
系统模式（sys）	运行具有特权的操作系统任务。

ARM微处理器的运行模式可以通过软件改变，也可以通过外部中断或异常处理改变。除用户模式以外，其余的所有6种模式称之为非用户模式，或特权模式（Privileged Modes）；其中除去用户模式和系统模式以外的5种又称为异常模式（Exception Modes），常用于处理中断或异常，以及需要访问受保护的系统资源等情况。RT-Thread线程选择的是运行于SVC模式，这样进行系统函数时可不用通过SWI方式陷入到SVC模式中。

### 1.2.3 ARM 的寄存器组织

本节主要说明 ARM 状态下的寄存器组织，因为 Thumb 的移植中，汇编代码部分都是在 ARM 状态下运行，所以 Thumb 状态下的寄存器组织不做过多的详细说明。

ARM 体系结构中包括通用寄存器和特殊寄存器。通用寄存器包括 R0～R15，可以分为三类：

- 未分组寄存器R0～R7；
- 分组寄存器R8～R14
- 程序计数器PC(R15)

未分组寄存器 R0～R7 在所有运行模式中，代码中指向的寄存器在物理上都是唯一的，他们未被系统用作特殊的用途，因此，在中断或异常处理进行运行模式转换时，由于不同的处理器运行模式均使用相同的物理寄存器，可能会造成寄存器中数据的破坏。

分组寄存器 R8～R14 则是和运行模式相关，代码中指向的寄存器和处理器当前运行的模式

密切相关。

对于 R8~R12 来说，每个寄存器对应两个不同的物理寄存器，当使用 FIQ 模式时，访问寄存器 R8\_fiq~R12\_fiq；当使用除 FIQ 模式以外的其他模式时，访问寄存器 R8\_usr~R12\_usr。对于 R13、R14 来说，每个寄存器对应 6 个不同的物理寄存器，其中的一个为用户模式与系统模式共用，另外 5 个物理寄存器对应于其他 5 种不同的运行模式。

采用以下的记号来区分不同的物理寄存器：

R13\_<mode>

R14\_<mode>

其中，mode 为以下几种模式之一：usr、fiq、irq、svc、abt、und。

由于处理器的每种运行模式均有自己独立的物理寄存器 R13，在用户应用程序的初始化部分都需要初始化每种模式下的 R13，使其指向该运行模式的栈空间，这样，当程序的运行进入异常模式时，可以将需要保护的寄存器放入 R13 所指向的堆栈，而当程序从异常模式返回时，则从对应的堆栈中恢复，采用这种方式可以保证异常发生后程序的正常执行。

R14 也称作子程序连接寄存器 (Subroutine Link Register) 或连接寄存器 LR。当执行 BL 子程序调用指令时，R14 中得到 R15 (程序计数器 PC) 的备份。其他情况下，R14 用作通用寄存器。与之类似，当发生中断或异常时，对应的分组寄存器 R14\_svc、R14\_irq、R14\_fiq、R14\_abt 和 R14\_und 用来保存 R15 的返回值。

寄存器 R14 常用在如下的情况：

在每一种运行模式下，都可用 R14 保存子程序的返回地址，当用 BL 或 BLX 指令调用子程序时，将 PC 的当前值拷贝给 R14，执行完子程序后，又将 R14 的值拷贝回 PC，即可完成子程序的调用返回。

**程序计数器 PC (R15)** 用作程序计数器 (PC)。在 ARM 状态下，位 [1:0] 为 0，位 [31:2] 用于保存 PC；在 Thumb 状态下，位 [0] 为 0，位 [31:1] 用于保存 PC；在 ARM 状态下，PC 的 0 和 1 位是 0，在 Thumb 状态下，PC 的 0 位是 0。

**CPSR (Current Program Status Register, 当前程序状态寄存器)**，CPSR 可在任何运行模式下被访问，它包括条件标志位、中断禁止位、当前处理器模式标志位，以及其他一些相关的控制和状态位。

每一种运行模式下又都有一个专用的物理状态寄存器，称为 SPSR (Saved Program Status Register, 备份的程序状态寄存器)，当异常发生时，SPSR 用于保存 CPSR 的当前值，从异常退出时则可由 SPSR 来恢复 CPSR。

图 A - 1 ARM 寄存器组织

ARM状态下的通用寄存器与程序计数器					
System & User	FIQ	Supervisor	About	IRQ	Undefined
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	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

ARM状态下的程序状态寄存器					
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

 = 分组寄存器

1.2.4 ARM 的中断处理（IRQ 中断）

RT-Thread 的 ARM 体系结构移植只涉及到 IRQ 中断，所以本节只讨论 IRQ 中断模式，主要包括 ARM 微处理器在硬件上是如何响应中断及如何从中断中返回的。

当中断产生时，ARM7TDMI 将执行的操作

- 1. 把当前 CPSR 寄存器的内容拷贝到相应的 SPSR 寄存器。这样当前的工作模式、中断屏蔽位和状态标志被保存下来。
- 2. 转入相应的模式，并关闭 IRQ 中断。
- 3. 把 PC 值减 4 后，存入相应的 LR 寄存器。
- 4. 将 PC 寄存器指向 IRQ 中断向量位置。

由中断返回时，ARM7TDMI 将完成的操作

- 1. 将备份程序状态寄存器的内容拷贝到当前程序状态寄存器，恢复中断前的状态。
- 2. 清除相应禁止中断位（如果已设置的话）。
- 3. 把连接寄存器的值拷贝到程序计数器，继续运行原程序。

1.3 RT-Thread 在 GNU GCC 环境下的移植

GNU GCC 是 GNU 的多平台编译器，也是开源项目中的首选编译环境，支持 ARM 各个版本的指令集，MIPS，x86 等多个体系结构，也为一些知名操作系统作为官方编译器（例如主流的几种 BSD 操作系统，Linux 操作系统，vxWorks 实时操作系统等），所以作为开源项

目的 RT-Thread 首选编译器是 GNU GCC，甚至在一些地方会对 GCC 做专门的优化。






以下就以 s3c4510、lunit4510 开发板为例，描述如何进行 RT-Thread 的移植。

### 1.3.1 CPU 相关移植

s3c4510 是一款 SAMSUNG 公司早期面向网络应用的 SoC 芯片，包括 16/32 位 ARM7TDMI 核，8KB 指令/数据共用的 cache，I<sup>2</sup>C 总线控制器，双通道缓冲 DMA 的以太网控制器，2 个带有 4 通道缓冲 DMA 的 HDLC，2 通道 GDMA，2 个 UART，两个 32 位定时器，18 个可编程的 I/O 端口，中断控制器，一个系统管理器等。

和通用平台中的 GCC 不同，编译操作系统会生成单独的目标文件，一些基本的算术操作例如除法，必须在链接的时候选择使用 gcc 库 (libgcc.a)，还是自身实现。RT-Thread 推荐选择后者，自己实现一些基本的算术操作，因为这样能够让生成的目标文件体积缩小一些。这些基本的算术操作统一放在各自体系结构目录下的 common 目录。另外 ARM 体系结构中 ARM 模式下的一些过程调用也是标准的，所以也放置了一些栈回溯的代码例程（在 Thumb 模式下这部分代码将不可用）。

表格 A- 3 kernel/libcpu/arm/common 目录下的文件

名称	大小/B	最后修改时间	说明
 backtrace.c	1233	2008-10-18 15:10:49	栈回溯实现
 div0.c	36	2008-10-18 15:10:49	
 divsi3.S	8424	2008-10-18 15:10:49	
 Makefile	265	2008-10-18 15:10:49	
 showmem.c	823	2008-10-18 15:10:49	

目前 common 目录下这些文件都已经存在，其他的 ARM 芯片移植基本上不需要重新实现或修改。

在 RT-Thread 核心中，和芯片相关的主要有两部分：关/开中断，及线程上下文切换（rt\_hw\_context\_switch、rt\_hw\_context\_switch\_to 和 rt\_hw\_context\_switch\_interrupt）。线程上下文切换的三个函数会在调度器中调用，主要是实现线程中的上下文保存和恢复，通常这部分是和硬件密切相关的，需要采用汇编实现。

在 kernel/libcpu/arm 目录下新建一个目录 s3c4510（建议取芯片的名称为目录名）：

a) 添加一个文件 context.S，分别实现如下函数：

rt\_hw\_interrupt\_disable

rt\_hw\_interrupt\_enable

中断关闭及中断使能函数，代码如下（操作 CPSR 寄存器屏蔽/使能所有中断）：

## 代码 A - 1 开关 ARM 中断

```
/*
 * rt_base_t rt_hw_interrupt_disable();
 */
.globl rt_hw_interrupt_disable
rt_hw_interrupt_disable:
    mrs r0, cpsr          ; 保存CPSR寄存器的值到R0寄存器
    orr r1, r0, #0xc0     ; R0寄存器的值或上0xc0（2、3位置1），结果放到r1中
    msr cpsr_c, r1        ; 把R1的值存放到CPSR寄存器中
    bx lr                 ; 返回调用rt_hw_interrupt_disable函数处，返回值在R0中

/*
 * void rt_hw_interrupt_enable(rt_base_t level);
 */
.globl rt_hw_interrupt_enable
rt_hw_interrupt_enable:
    msr cpsr, r0          ; 把R0的值保存到CPSR中
    bx lr                 ; 返回调用rt_hw_interrupt_enable函数处
```

实现 `rt_hw_context_switch` 和 `rt_hw_context_switch_to` 函数，代码如代码 A - 2 上下文切换代码

## 代码 A - 2 上下文切换代码

```
/*
 * void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
 */
.globl rt_hw_context_switch
rt_hw_context_switch:
    stmfd sp!, {lr}        ; 把LR寄存器压入栈，也就是从这个函数返回后的下一个执行处
    stmfd sp!, {r0-r12, lr}; ; 把R0 - R12以及LR压入栈

    mrs r4, cpsr           ; 读取CPSR寄存器到R4寄存器
    stmfd sp!, {r4}        ; 把R4寄存器压栈（即上一指令取出的CPSR寄存器）
    mrs r4, spsr           ; 读取SPSR寄存器到R4寄存器
    stmfd sp!, {r4}        ; 把R4寄存器压栈（即SPSR寄存器）

    str sp, [r0]           ; 把栈指针更新到TCB的sp，是由R0传入此函数

    ; 到这里换出线程的上下文都保存在栈中

    ldr sp, [r1]           ; 载入切换到线程的TCB的sp，即此线程换出时保存的sp寄存器

    ; 从切换到线程的栈中恢复上下文，次序和保存的时候刚好相反

    ldmsd sp!, {r4}        ; 出栈到R4寄存器（保存了SPSR寄存器）
    msr spsr_cxsf, r4      ; 恢复SPSR寄存器
    ldmsd sp!, {r4}        ; 出栈到R4寄存器（保存了CPSR寄存器）
    msr cpsr_cxsf, r4      ; 恢复CPSR寄存器

    ldmsd sp!, {r0-r12, lr, pc} ; 对R0 - R12及LR、PC进行恢复

/*
 * void rt_hw_context_switch_to(rt_uint32 to);
 * 此函数只在系统进行第一次发生任务切换时使用，因为是从没有线程的状态进行切换
 * 实现上，刚好是rt_hw_context_switch的下半截
 */
.globl rt_hw_context_switch_to
```

```

rt_hw_context_switch_to:
    ldr sp, [r0]                ; 获得切换到线程的SP指针

    ldmfd sp!, {r4}             ; 出栈R4寄存器（保存了SPSR寄存器值）
    msr spsr_cxsf, r4           ; 恢复SPSR寄存器
    ldmfd sp!, {r4}             ; 出栈R4寄存器（保存了CPSR寄存器值）
    msr cpsr_cxsf, r4           ; 恢复CPSR寄存器

    ldmfd sp!, {r0-r12, lr, pc} ; 恢复R0 - R12, LR及PC寄存器

```

在 RT-Thread 中，如果中断服务例程触发了上下文切换（即执行了一次 `rt_schedule` 函数试图进行一次调度），它会设置标志 `rt_thread_switch_interrupt_flag` 为真。而后在所有中断服务例程都处理完毕向线程返回的时候，如果 `rt_thread_switch_interrupt_flag` 为真，那么中断结束后就必须进行线程的上下文切换。这部分上下文切换代码和上面会有些不同，这部分在下一个文件中叙述，但设置 `rt_thread_switch_interrupt_flag` 标志的代码以及保存切换出和切换到线程的函数在这里给出，如代码 A-3。

#### 代码 A - 3 中断中上下文切换汇编代码

```

/*
 * void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);
 * 此函数会在调度器中调用，在调度器做上下文切换前会判断是否处于中断服务模式中，如果
 * 是则调用rt_hw_context_switch_interrupt函数（设置中断中任务切换标志）
 * 否则调用 rt_hw_context_switch函数（进行真正的线程上线文切换）
 */
.globl rt_thread_switch_interrput_flag
.globl rt_interrupt_from_thread
.globl rt_interrupt_to_thread
.globl rt_hw_context_switch_interrupt
rt_hw_context_switch_interrupt:
    ldr r2, =rt_thread_switch_interrput_flag
    ldr r3, [r2]                ; 载入中断中切换标志地址
    cmp r3, #1                 ; 等于 1 ?
    beq _reswitch              ; 如果等于1，跳转到_reswitch
    mov r3, #1                 ; 设置中断中切换标志位1
    str r3, [r2]               ; 保存到标志变量中
    ldr r2, =rt_interrupt_from_thread
    str r0, [r2]               ; 保存切换出线程栈指针
_reswitch:
    ldr r2, =rt_interrupt_to_thread
    str r1, [r2]               ; 保存切换到线程栈指针
    bx lr

```

上面的代码等价于如下的 C 代码，代码 A-4：

#### 代码 A - 4 中断中上下文切换 C 代码

```

/*
 * void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);
 * 此函数会在调度器中调用，在调度器做上下文切换前会判断是否处于中断服务模式中，如果
 * 是则调用rt_hw_context_switch_interrupt函数（设置中断中任务切换标志）
 * 否则调用 rt_hw_context_switch函数（进行真正的线程上线文切换）
 */
rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to)
{
    if (rt_thread_switch_interrput_flag == 1)
        rt_interrupt_from_thread = from;
}

```



```

        else
            rt_thread_switch_interrupt_flag = 1;

        rt_interrupt_to_thread = to;
    }

```

## b) 添加文件 stack.c

现在线程上下文切换的代码已经有了，那么创建一个线程时，栈也一定需要按照入栈的顺序来创建初始的栈了，否则就不可能把线程正确的切过去。

### 代码 A - 5 stack.c 初始化栈函数

```

#include <rtthread.h>
#define SVCMODE          0x13

/**
 * 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) = (unsigned long)tentry;          /* 线程入口，等价于线程的PC */
    *(--stk) = (unsigned long)texit;         /* lr */
    *(--stk) = 0;                           /* r12 */
    *(--stk) = 0;                           /* r11 */
    *(--stk) = 0;                           /* r10 */
    *(--stk) = 0;                           /* r9 */
    *(--stk) = 0;                           /* r8 */
    *(--stk) = 0;                           /* r7 */
    *(--stk) = 0;                           /* r6 */
    *(--stk) = 0;                           /* r5 */
    *(--stk) = 0;                           /* r4 */
    *(--stk) = 0;                           /* r3 */
    *(--stk) = 0;                           /* r2 */
    *(--stk) = 0;                           /* r1 */
    *(--stk) = (unsigned long)parameter;     /* r0 : 入口函数参数 */
    *(--stk) = SVCMODE;                     /* cpsr, 采用SVC模式运行 */
    *(--stk) = SVCMODE;                     /* spsr */

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

```

## c) 添加启动文件 start.S

接下来是系统启动的代码。因为 ARM 体系结构异常的触发总是置于 0 地址的（或者说异常向量表总是置于 0 地址），所以操作系统要捕获异常（最重要的是中断异常）就必须放置上自己的向量表。

此外，由于系统刚上电可能一些地方也需要进行初始化（RT-Thread 推荐，和板子相关的初始化最好放在 bsp 目录中），对 ARM 体系结构，另一个最重要的就是（各种模式下）栈的设置。

代码 A-6 是 s3c4510 的启动文件清单。

#### 代码 A - 6 start.S 启动文件

```
.equ USERMODE,    0x10
.equ FIQMODE,     0x11
.equ IRQMODE,     0x12
.equ SVCMODE,     0x13
.equ ABORTMODE,   0x17
.equ UNDEFMODE,   0x1b
.equ MODEMASK,    0x1f
.equ NOINT,       0xc0

.section .init, "ax"
.code 32
.globl _start
_start:
    b    reset
    ldr pc, _vector_undef
    ldr pc, _vector_swi
    ldr pc, _vector_pabt
    ldr pc, _vector_dabt
    ldr pc, _vector_resv
    ldr pc, _vector_irq
    ldr pc, _vector_fiq

_vector_undef:    .word vector_undef
_vector_swi:      .word vector_swi
_vector_pabt:     .word vector_pabt
_vector_dabt:     .word vector_dabt
_vector_resv:     .word vector_resv
_vector_irq:      .word vector_irq
_vector_fiq:      .word vector_fiq

    .balignl 16, 0xdeadbeef    ; 以当前地址开始到16倍数地址之间填充4字节内容0xdeadbeef

_TEXT_BASE:      .word TEXT_BASE

; /*
; * rtthread kernel start and end
; * which are defined in linker script
; */
.globl _rtthread_start
_rtthread_start: .word _start
.globl _rtthread_end
_rtthread_end:   .word _end

; /*
; * rtthread bss 段的开始和结束，都在链接脚本中定义
; */
.globl _bss_start
_bss_start: .word __bss_start
.globl _bss_end
```

```

_bss_end:    .word __bss_end

; /*
; * 各种模式下的栈内存空间
; */
UNDSTACK_START: .word _undefined_stack_start + 128
ABTSTACK_START: .word _abort_stack_start + 128
FIQSTACK_START: .word _fiq_stack_start + 1024
IRQSTACK_START: .word _irq_stack_start + 1024
SVCSTACK_START: .word _svc_stack_start + 4096

.equ INTMSK, 0x3ff4008

; /*
; * 系统入口点
; */
reset:
    ; 先初始化为SVC模式
    mrs r0, cpsr
    bic r0, r0, #0x1f
    orr r0, r0, #0x13
    msr cpsr, r0

    ; 屏蔽所有s3c4510的中断
    ldr r1, =INTMSK
    ldr r0, =0xffffffff
    str r0, [r1]

    ; 设置异常向量表
    ldr r0, _TEXT_BASE      ; 源地址是_TEXT_BASE, 0x8000, 在编译时指定
    mov r1, #0x00          ; 目标地址是0x0地址, 这样异常才能正确捕捉到

    ldmia r0!, {r3-r10}    ; 载入8 x 4 bytes
    stmia r1!, {r3-r10}    ; 存入
    ldmia r0!, {r3-r10}    ; 载入8 x 4 bytes
    stmia r1!, {r3-r10}    ; 存入

    ; 各种模式下的栈设置
    bl stack_setup

    ; 对bss段进行清零
    mov r0, #0              ; 置R0为0
    ldr r1, =__bss_start    ; 获得bss段开始位置
    ldr r2, =__bss_end      ; 获得bss段结束位置
bss_loop:
    cmp r1, r2              ; 确认是否已经到结束位置
    strlo r0, [r1], #4      ; 清零
    blo bss_loop           ; 循环直到结束

    ; 对C++的全局对象进行构造
    ldr r0, =__ctors_start__ ; 获得ctors开始位置
    ldr r1, =__ctors_end__   ; 获得ctors结束位置
ctor_loop:
    cmp r0, r1
    beq ctor_end
    ldr r2, [r0], #4
    stmfd sp!, {r0-r1}
    mov lr, pc
    bx r2
    ldmbd sp!, {r0-r1}

```

```

        b        ctor_loop
ctor_end:

        ; 跳转到rtthread_startup中, 这是RT-Thread的第一C函数
        ; 也可以看成是RT-Thread的入口点
        ldr pc, _rtthread_startup

_rtthread_startup: .word rtthread_startup

; 异常处理
vector_undef:    bl    rt_hw_trap_undef
vector_swi:      bl    rt_hw_trap_swi
vector_pabt:     bl    rt_hw_trap_pabt
vector_dabt:     bl    rt_hw_trap_dabt
vector_resv:     bl    rt_hw_trap_resv

.globl rt_interrupt_enter
.globl rt_interrupt_leave
.globl rt_thread_switch_interrput_flag
.globl rt_interrupt_from_thread
.globl rt_interrupt_to_thread
        ; IRQ异常处理
vector_irq:
        stmfd    sp!, {r0-r12,lr}          ; 先把R0 - R12, LR寄存器压栈保存
        bl      rt_interrupt_enter          ; 调用rt_interrupt_enter以确认进入中断处理
        bl      rt_hw_trap_irq              ; 调用C函数的中断处理函数进行处理
        bl      rt_interrupt_leave          ; 调用rt_interrupt_leave表示离开中断处理

        ; 如果设置了rt_thread_switch_interrput_flag, 进行中断中的线程上下文处理
        ldr r0, =rt_thread_switch_interrput_flag
        ldr r1, [r0]
        cmp r1, #1                          ; 判断是否设置了中断中线程切换标志
        beq _interrupt_thread_switch        ; 是则跳转到_interrupt_thread_switch

        ldmfd    sp!, {r0-r12,lr}          ; R0 - R12, LR出栈
        subs     pc, lr, #4                  ; 中断返回

        ; FIQ异常处理
        ; 在这里仅仅进行了简单的函数回调, OS并没对FIQ做特别处理。
        ; 如果在FIQ中要用到OS的一些服务, 需要做IRQ异常类似处理。
vector_fiq:
        stmfd    sp!, {r0-r7,lr}           ; R0 - R7, LR寄存器入栈,
                                           ; FIQ模式下, R0 - R7是通用寄存器,
                                           ; 其他的都是分组寄存器

        bl      rt_hw_trap_fiq              ; 跳转到rt_hw_trap_fiq进行处理
        ldmfd    sp!, {r0-r7,lr}
        subs     pc, lr, #4                  ; FIQ异常返回

        ; 进行中断中的线程切换
_interrupt_thread_switch:
        mov      r1, #0                      ; 清除切换标识
        str      r1, [r0]

        ldmfd    sp!, {r0-r12,lr}          ; 载入保存的R0 - R12及LR寄存器
        stmfd    sp!, {r0-r3}              ; 先保存R0 - R3寄存器
        mov      r1, sp                    ; 保存一份IRQ模式下的栈指针到R1寄存器
        add      sp, sp, #16                ; IRQ栈中保持了R0 - R4, 加16后刚好到栈底
                                           ; 后面会直接跳出IRQ模式, 相当于恢复IRQ的栈
        sub      r2, lr, #4                  ; 保存中断前线程的PC到R2寄存器

```

```

mrs      r3,  spsr                                ; 保存中断前的CPSR到R3寄存器
orr      r0,  r3,  #NOINT                          ; 关闭中断前线程的中断
msr      spsr_c, r0

ldr      r0,  =.+8                                ; 把当前地址+8载入到R0寄存器中
movs     pc,  r0                                    ; 退出IRQ模式，由于SPSR被设置成关中断模式，
                                                    ; 所以从IRQ返回后，中断并没有打开

                                                    ; R0寄存器中的位置实际就是下一条指令，
                                                    ; 即PC继续往下走

                                                    ; 此时
                                                    ; 模式已经换成中断前的SVC模式，
                                                    ; SP寄存器也是SVC模式下的栈寄存器
                                                    ; R1保存IRQ模式下的栈指针
                                                    ; R2保存切换出线程的PC
                                                    ; R3保存切换出线程的CPSR

stmfd    sp!, {r2}                                ; 对R2寄存器压栈，即前面保存的切换出线程PC
stmfd    sp!, {r4-r12,lr}                          ; 对LR, R4 - R12寄存器进行压栈（切换出线程的）
mov      r4,  r1                                    ; R4寄存器为IRQ模式下的栈指针，
                                                    ; 栈中保存了切换出线程的R0 - R3
mov      r5,  r3                                    ; R5中保存了切换出线程的CPSR
ldmfd    r4!, {r0-r3}                              ; 恢复切换出线程的R0 - R3寄存器
stmfd    sp!, {r0-r3}                              ; 对切换出线程的R0 - R3寄存器进行压栈
stmfd    sp!, {r5}                                  ; 对切换出线程的CPSR进行压栈
mrs      r4,  spsr                                ; 读取切换出线程的SPSR寄存器
stmfd    sp!, {r4}                                  ; 对切换出线程的SPSR进行压栈

ldr      r4,  =rt_interrupt_from_thread
ldr      r5,  [r4]
str      sp,  [r5]                                ; 更新切换出线程的sp指针（存放在TCB中）

ldr      r6,  =rt_interrupt_to_thread
ldr      r6,  [r6]
ldr      sp,  [r6]                                ; 获得切换到线程的栈指针

ldmfd    sp!, {r4}                                  ; 恢复切换到线程的SPSR寄存器
msr      SPSR_cxsf, r4
ldmfd    sp!, {r4}                                  ; 恢复切换到线程的CPSR寄存器
msr      CPSR_cxsf, r4

ldmfd    sp!, {r0-r12,lr,pc}                      ; 恢复切换到线程的R0 - R12, LR及PC寄存器

; 各种模式下的栈设置
stack_setup:
; 未定义指令模式
msr      cpsr_c, #UNDEFMODE|NOINT
ldr      sp, UNDSTACK_START

; 异常模式
msr      cpsr_c, #ABORTMODE|NOINT
ldr      sp, ABTSTACK_START

; FIQ模式
msr      cpsr_c, #FIQMODE|NOINT
ldr      sp, FIQSTACK_START

; IRQ模式

```

```

msr cpsr_c, #IRQMODE|NOINT
ldr sp, IRQSTACK_START

; SVC模式
msr cpsr_c, #SVCMODE|NOINT
ldr sp, SVCSTACK_START

bx lr

```

#### d) 添加 interrupt.c 文件

ARM7TDMI 具备中断表直接跳转方式，即在放置 IRQ 中断的地方直接使用一个跳转指令跳转到相应中断服务例程中。但其弊端是每个中断服务例程入口和出口都不被操作系统感知，正向 start.C 文件中所示，IRQ 中断来了后是先进入一段操作系统的代码，然后调用 rt\_hw\_trap\_irq 函数进行中断服务例程的调度，这个函数就可以在 interrupt.c 这个文件中实现。

#### 代码 A - 7 interrupt.c 文件

```

#include <rtthread.h>
#include "s3c4510.h"

#define MAX_HANDLERS 21      /* s3c4510支持的最大外部中断数是21 */

extern rt_uint32_t rt_interrupt_nest;

/* 存放中断向量的表 */
rt_isr_handler_t isr_table[MAX_HANDLERS];

/* 中断中切换用到的存放切换出和切换到线程的栈指针 */
rt_uint32_t rt_interrupt_from_thread, rt_interrupt_to_thread;
/* 中断中切换用到的是否需要发生切换的标志 */
rt_uint32_t rt_thread_switch_interrput_flag;

/* 默认的中断服务例程 */
void rt_hw_interrupt_handle(int vector)
{
    rt_kprintf("Unhandled interrupt %d occured!!!\n", vector);
}

/**
 * 初始化中断
 */
void rt_hw_interrupt_init()
{
    register int i;

    /* 屏蔽所有中断*/
    INTMASK = 0x3fffffff;

    /* 清除所有悬挂的中断 */
    INTPEND = 0x1fffffff;

    /* all=IRQ mode */
    INTMODE = 0x0;

    /* 初始化中断表 */
    for(i=0; i<MAX_HANDLERS; i++)
    {

```

```

        isr_table[i] = rt_hw_interrupt_handle;
    }

    /* 初始化中断嵌套层数和中断中切换用到的变量 */
    rt_interrupt_nest = 0;
    rt_interrupt_from_thread = 0;
    rt_interrupt_to_thread = 0;
    rt_thread_switch_interrput_flag = 0;
}

/**
 * 屏蔽一个中断，移植底层提供的API
 * @param vector 中断号
 */
void rt_hw_interrupt_mask(int vector)
{
    INTMASK |= 1 << vector;
}

/**
 * 去屏蔽一个中断，移植底层提供的API
 * @param vector 中断号
 */
void rt_hw_interrupt_umask(int vector)
{
    INTMASK &= ~(1 << vector);
}

/**
 * 安装一个中断服务例程到对应的中断号
 * @param vector 中断号
 * @param new_handler 新的中断服务例程
 * @param old_handler 用于存放老的中断服务例程，允许为RT_NULL
 */
void rt_hw_interrupt_install(int vector, rt_isr_handler_t new_handler,
rt_isr_handler_t *old_handler)
{
    if(vector < MAX_HANDLERS)
    {
        if (old_handler != RT_NULL) *old_handler = isr_table[vector];
        if (new_handler != RT_NULL) isr_table[vector] = new_handler;
    }
}

```

#### e) 添加 trap.c 文件

在 start.S 中设定的异常向量表中还会调用其他的几个异常函数，在 trap.c 中依次实现。

#### 代码 A - 8 trap.c 文件

```

#include "s3c4510.h"

/**
 * Undef异常处理，入口参数是系统寄存器，由于入口处并没把系统寄存器压栈，所以参数忽略
 *
 * @param regs system registers
 */
void rt_hw_trap_undef(struct rt_hw_register *regs)
{

```

```

        rt_kprintf("undefined instruction\n");
        rt_hw_cpu_shutdown();
    }

/**
 * SWI异常处理，入口参数是系统寄存器，由于入口处并没把系统寄存器压栈，所以参数忽略
 *
 * @param regs system registers
 */
void rt_hw_trap_swi(struct rt_hw_register *regs)
{
    rt_kprintf("software interrupt\n");

    rt_hw_cpu_shutdown();
}

/**
 * Abort异常，入口参数是系统寄存器，由于入口处并没把系统寄存器压栈，所以参数忽略。
 *
 * @param regs system registers
 */
void rt_hw_trap_pabt(struct rt_hw_register *regs)
{
    rt_kprintf("prefetch abort\n");

    rt_hw_cpu_shutdown();
}

/**
 * Data Abort异常，入口参数是系统寄存器，由于入口处并没把系统寄存器压栈，所以参数忽略。
 *
 * @param regs system registers
 */
void rt_hw_trap_dabt(struct rt_hw_register *regs)
{
    rt_kprintf("data abort\n");

    rt_hw_cpu_shutdown();
}

/**
 * Resv异常，入口参数是系统寄存器，由于入口处并没把系统寄存器压栈，所以参数忽略
 *
 * @param regs system registers
 */
void rt_hw_trap_resv(struct rt_hw_register *regs)
{
    rt_kprintf("not used\n");

    rt_hw_cpu_shutdown();
}

/**
 * IRQ中断处理
 */
void rt_hw_trap_irq()
{
    /* 使用SkyEye模拟时的中断处理代码 */
    register int offset;
    unsigned long pend;

```



```

rt_isr_handler_t isr_func;
extern rt_isr_handler_t isr_table[];

/* 获得悬挂的中断 */
pend = INTPEND;

/* 计算中断号 */
for (offset = 0; offset < INTGLOBAL; offset++)
{
    if (pend & (1 << offset)) break;
}
if (offset == INTGLOBAL) return;

/* 获得中断服务例程 */
isr_func = isr_table[offset];

/* 调用中断服务例程 */
isr_func(offset);

/* 清除悬挂的中断 */
INTPEND = 1 << offset;
}

/**
 * FIQ中断处理，未使用，直接打印显示
 */
void rt_hw_trap_fiq()
{
    rt_kprintf("fast interrupt request\n");
}

```

#### f) 添加 cpu.c 文件

s3c4510 具备 8KB 的片内指令/数据 cache，在 cpu.c 文件中实现相应代码。

#### 代码 A - 9 cpu.c 文件

```

#include <rtthread.h>
#include "s3c4510.h"

/**
 * 使能I-Cache
 */
void rt_hw_cpu_icache_enable()
{
    rt_base_t reg;
    volatile int i;

    /* flush cache before enable */
    rt_uint32_t *tagram = (rt_uint32_t *)S3C4510_SRAM_ADDR;

    SYSCFG &= ~(1<<1); /* Disable cache */

    for(i=0; i < 256; i++)
    {
        *tagram = 0x00000000;
        tagram += 1;
    }
}

```

```

        /*
         * Enable chache
         */
        reg = SYSCFG;
        reg |= (1 << 1);
        SYSCFG = reg;
    }

/**
 * 关闭I-Cache
 */
void rt_hw_cpu_icache_disable()
{
    rt_base_t reg;

    reg = SYSCFG;
    reg &= ~(1<<1);
    SYSCFG = reg;
}

/**
 * 获得I-Cache的状态
 */
rt_base_t rt_hw_cpu_icache_status()
{
    return 0;
}

/**
 * 使能D-Cache
 */
void rt_hw_cpu_dcache_enable()
{
    rt_hw_cpu_icache_enable();
}

/**
 * 关闭D-Cache
 */
void rt_hw_cpu_dcache_disable()
{
    rt_hw_cpu_icache_disable();
}

/**
 * 获得D-Cache的状态
 */
rt_base_t rt_hw_cpu_dcache_status()
{
    return rt_hw_cpu_icache_status();
}

/**
 * Reset CPU, 使用WDT, 当定时器超时时自动重启
 */
void rt_hw_cpu_reset()
{
}

/**

```

```

    * 进行CPU停机，先关闭中断而后竟如死循环
    */
void rt_hw_cpu_shutdown()
{
    rt_uint32_t level;

    level = rt_hw_interrupt_disable();
    rt_kprintf("shutdown...\n");

    while (1);
}

```

#### g) 添加 serial.c 文件

串口是 s3c4510 芯片集成的外设，所以串口驱动也在这里实现，代码清单如下。

#### 代码 A - 10 serial.c 文件

```

#include <rthw.h>
#include <rtthread.h>

#include "s3c4510.h"

#define USTAT_RCV_READY    0x20    /* receive data ready */
#define USTAT_TXB_EMPTY    0x40    /* tx buffer empty */

void rt_serial_init(void);

/**
 * 这个函数用于在控制台上显示一个字符串，会被rt_kprintf调用。
 *
 * @param str 显示的字符串
 */
void rt_console_puts(const char* str)
{
    while (*str)
    {
        /* 调用rt_serial_putc依次显示 */
        rt_serial_putc (*str++);
    }
}

/**
 * 初始化串口，只初始化了串口0，波特率设置为19200
 */
void rt_serial_init()
{
    rt_uint32_t divisor = 0;

    divisor = 0x500; /* 19200 baudrate, 50MHz */

    UARTLCON0 = 0x03;
    UARTCONT0 = 0x09;
    UARTBRD0 = divisor;

    UARTLCON1 = 0x03;
    UARTCONT1 = 0x09;
    UARTBRD1 = divisor;
}

```

```

        for(divisor=0; divisor<100; divisor++);
    }

    /**
     * 从串口中获得一个字符
     *
     * @return 读到的字符
     */
    char rt_serial_getc()
    {
        while ((UARTSTAT0 & USTAT_RCV_READY) == 0);

        return UARTRXB0;
    }

    /**
     * 往串口中输出一个字符，如果遇到'\n'字符将会在前面插入一个'\r'字符
     *
     * @param c 需要写的字符
     */
    void rt_serial_putc(const char c)
    {
        if (c=='\n')rt_serial_putc('\r');

        /* wait for room in the transmit FIFO */
        while(!(UARTSTAT0 & USTAT_TXB_EMPTY));

        UARTTXH0 = (rt_uint8_t)c;
    }

```

## 1.3.2 板级相关移植

bsp 目录下放置了各类开发板/平台的具体实现，包括开发板/平台的初始化，外设的驱动等。由于 limit4510 开发中并没特别外扩设备外设，所以这个目录中只需要进行简单的初始化即可。

### a) RT-Thread 配置头文件

RT-Thread 代码中默认包含 rtconfig.h 头文件作为它的配置文件，会定义 RT-Thread 中各种选项设置，例如内核对象名称长度，是否支持线程间通信中的信箱，消息队列，快速事件等。详细情况请参看附录 B RT-Thread 配置，和此移植相关的配置文件代码如下：

#### 代码 A - 11 配置头文件

```

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

/* RT_NAME_MAX*/
#define RT_NAME_MAX 8

/* RT_ALIGN_SIZE*/
#define RT_ALIGN_SIZE 4

/* PRIORITY_MAX*/
#define RT_THREAD_PRIORITY_MAX 256

/* Tick per Second*/

```

```

#define RT_TICK_PER_SECOND    100

/* SECTION: RT_DEBUG */
/* Thread Debug*/
/* #define RT_THREAD_DEBUG */

/* 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 Faset Event*/
/* #define RT_USING_FASTEVENT */

/* 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 */

/* Using SLAB Allocator*/
#define RT_USING_SLAB

/* SECTION: Device System */
/* Using Device System*/
#define RT_USING_DEVICE

/* SECTION: Console options */
/* the buffer size of console*/
#define RT_CONSOLEBUF_SIZE    128

/* SECTION: FinSH shell options */
/* Using FinSH as Shell*/
#define RT_USING_FINSH

/* SECTION: emulator options */
/* Using QEMU or SkyEye*/
/* #define RT_USING_EMULATOR */

/* SECTION: a mini libc */
/* Using mini libc library*/
/* #define RT_USING_MINILIBC */

```

```

/* SECTION: C++ support */
/* Using C++ support*/
#define RT_USING_CPLUSPLUS

/* SECTION: lwip, a lightweight TCP/IP protocol stack */
/* Using lightweight TCP/IP protocol stack*/
/* #define RT_USING_LWIP */

/* Trace LwIP protocol*/
/* #define RT_LWIP_DEBUG */

/* Enable ICMP protocol*/
#define RT_LWIP_ICMP

/* Enable IGMP protocol*/
#define RT_LWIP_IGMP

/* Enable UDP protocol*/
#define RT_LWIP_UDP

/* Enable TCP protocol*/
#define RT_LWIP_TCP

/* Enable SNMP protocol*/
/* #define RT_LWIP_SNMP */

/* Using DHCP*/
/* #define RT_LWIP_DHCP */

/* ip address of target*/
#define RT_LWIP_IPADDR0 192
#define RT_LWIP_IPADDR1 168
#define RT_LWIP_IPADDR2 0
#define RT_LWIP_IPADDR3 30

/* gateway address of target*/
#define RT_LWIP_GWADDR0 192
#define RT_LWIP_GWADDR1 168
#define RT_LWIP_GWADDR2 0
#define RT_LWIP_GWADDR3 1

/* mask address of target*/
#define RT_LWIP_MSKADDR0 255
#define RT_LWIP_MSKADDR1 255
#define RT_LWIP_MSKADDR2 255
#define RT_LWIP_MSKADDR3 0

#endif

```

## b) 添加启动文件 startup.c

### 代码 A - 12 startup.c 文件

```

#include <rthw.h>
#include <rtthread.h>

#include <s3c4510.h>
#include <board.h>

```

```

/* 几个模式中用到的栈 */
char _undefined_stack_start[128];
char _abort_stack_start[128];
char _irq_stack_start[1024];
char _fiq_stack_start[1024];
char _svc_stack_start[4096];

extern void rt_hw_interrupt_init(void);
extern void rt_hw_board_init(void);
extern void rt_serial_init(void);
extern void rt_system_timer_init(void);
extern void rt_system_scheduler_init(void);
extern void rt_system_heap_init(void* begin_addr, void* end_addr);
extern void rt_thread_idle_init(void);
extern void rt_hw_cpu_icache_enable(void);
extern void rt_show_version(void);
extern int  rt_application_init(void);

extern unsigned char __bss_start;
extern unsigned char __bss_end;

/**
 * RT-Thread启动函数
 */
void rtthread_startup(void)
{
    /* 使能I/D Cache */
    rt_hw_cpu_icache_enable();
    rt_hw_cpu_dcache_enable();

    /* 初始化中断 */
    rt_hw_interrupt_init();

    /* 初始化开发板硬件 */
    rt_hw_board_init();

    /* 初始化串口硬件 */
    rt_serial_init();

    /* 显示RT-Thread版本信息 */
    rt_show_version();

    /* 初始化系统节拍 */
    rt_system_tick_init();

    /* 初始化内核对象 */
    rt_system_object_init();

    /* 初始化系统定时器 */
    rt_system_timer_init();

    /* 初始化堆内存, __bss_end在链接脚本中定义 */
#ifdef RT_USING_HEAP
    rt_system_heap_init((void*)&__bss_end, (void*)0x1000000);
#endif

    /* 初始化系统调度器 */
    rt_system_scheduler_init();

```

```

#ifdef RT_USING_HOOK
    /* 设置空闲线程的钩子函数 */
    rt_thread_idle_sethook(rt_hw_led_flash);
#endif

    /* 初始化用户程序 */
    rt_application_init();

    /* 初始化IDLE线程 */
    rt_thread_idle_init();

    /* 去屏蔽全局中断，使系统能够相应中断 */
    rt_hw_interrupt_umask( INTGLOBAL );

    /* 开始启动系统调度器，切换到第一个线程中 */
    rt_system_scheduler_start();

    /* 此处应该是永远不会达到的 */
    return ;
}

```

### c) 添加开发板初始化文件 board.c

#### 代码 A - 13 board.c 文件

```

#include <rthw.h>
#include <rtthread.h>

#include <s3c4510.h>

#define DATA_COUNT    0x7a120

/* 系统节拍用的定时器 */
void rt_timer_handler(int vector)
{
    /* reset TDATA0 */
    TDATA0 = DATA_COUNT;

    /* 调用rt_tick_increas去触发一个系统节拍 */
    rt_tick_increas();
}

/**
 * lunit4510开发板初始化
 */
void rt_hw_board_init()
{
    /* 设置定时器0为系统节拍定时器 */
    TDATA0 = DATA_COUNT;
    TCNT0 = 0x0;
    TMOD= 0x3;

    /* 装载定时器0中断服务例程 */
    rt_hw_interrupt_install(INTTIMER0, rt_timer_handler, RT_NULL);
    rt_hw_interrupt_umask( INTTIMER0 );
}

/**
 * 设置LED灯

```



```

    * @param led the led status
    */
void rt_hw_led_set(rt_uint32_t led)
{
    IOPDATA = 1 << led;
}

/* LED闪烁 */
void rt_hw_led_flash(void)
{
    int i;

    rt_hw_led_set(4);
    for ( i = 0; i < 2000000; i++);

    rt_hw_led_set(5);
    for ( i = 0; i < 2000000; i++);

    rt_hw_led_set(6);
    for ( i = 0; i < 2000000; i++);

    rt_hw_led_set(17);
    for ( i = 0; i < 2000000; i++);
}

```

#### d) 添加用户初始化文件 application.c

##### 代码 A - 14 application.c 文件

```

/* 只建立一个空内核，直接返回即可 */
int rt_application_init()
{
    return 0;    /* empty */
}

```

#### e) 链接脚本

由于只进行运行于 RAM 的移植，所以只需要添加 `lunit4510_ram.lds` 即可。

##### 代码 A - 15 lunit4510\_ram.lds 文件

```

/* 输出格式为ARM小端模式 */
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)

/* 定义入口点为_start*/
ENTRY(_start)
SECTIONS
{
    /* 生成的代码从0x0开始 */
    . = 0x00000000;

    /* text段 */
    . = ALIGN(4);
    .text : {
        *(.init)                /* .init即start.S中代码放置的位置，初始的地方是向量表 */
        *(.text)
        *(.gnu.linkonce.t*)
    }
}

```

```

/* rodata段 */
. = ALIGN(4);
.rodata : { *(.rodata) *(.rodata.*) *(.gnu.linkonce.r*) *(.eh_frame) }

/* C++中的全局对象构造部分 */
. = ALIGN(4);
.ctors :
{
    PROVIDE(__ctors_start__ = .);
    KEEP(*(SORT(.ctors.*)))
    KEEP(*(.ctors))
    PROVIDE(__ctors_end__ = .);
}

/* C++中的全局对象析构部分 */
.dtors :
{
    PROVIDE(__dtors_start__ = .);
    KEEP(*(SORT(.dtors.*)))
    KEEP(*(.dtors))
    PROVIDE(__dtors_end__ = .);
}

/* data段 */
. = ALIGN(4);
.data :
{
    *(.data)
    *(.data.*)
    *(.gnu.linkonce.d*)
}

/* bss段，在bss段前后分别放置__bss_start和__bss_end */
. = ALIGN(4);
__bss_start = .;
.bss : { *(.bss) }
__bss_end = .;

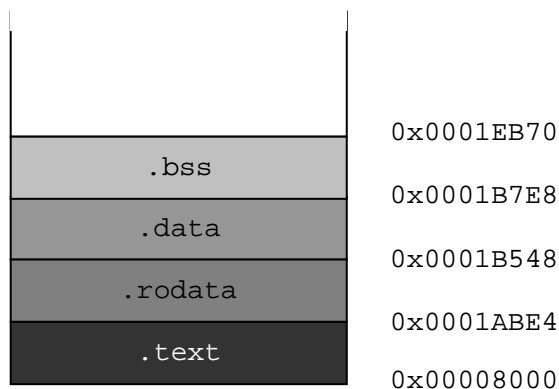
/* 调试信息段 */
.stab 0 : { *(.stab) }
.stabstr 0 : { *(.stabstr) }
.stab.excl 0 : { *(.stab.excl) }
.stab.exclstr 0 : { *(.stab.exclstr) }
.stab.index 0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment 0 : { *(.comment) }
.debug_abbrev 0 : { *(.debug_abbrev) }
.debug_info 0 : { *(.debug_info) }
.debug_line 0 : { *(.debug_line) }
.debug_pubnames 0 : { *(.debug_pubnames) }
.debug_aranges 0 : { *(.debug_aranges) }

_end = .;
}

```

最后生成的映像文件如图所示。

图 A - 2 rtthread-lumit.elf 文件结构



## 1.4 RT-Thread 在 RealView MDK 环境下的移植

本节用到的 RealView MDK 版本是 3.40 评估版，因为生成的代码小于<32k，可以正常编译调试。移植的目标板是 AT91SAM7S64，由 icdev.cn 网站提供，感谢 icdev.cn 网站的支持。（由于 RealView MDK 评估版和 RealView MDK 专业版的差异，专业版会生成更小的代码尺寸，推荐使用专业版）

### 1.4.1 AT91SAM7S64 概述

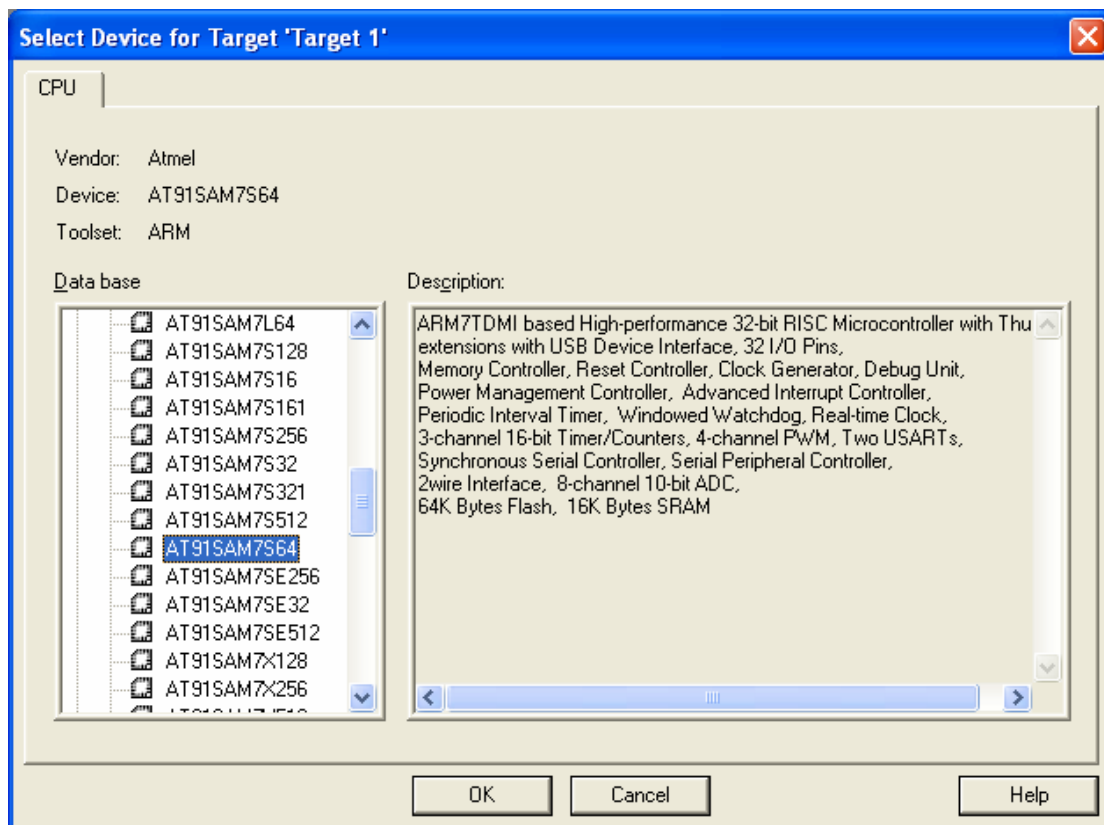
AT91SAM7S64 是 Atmel 32 位 ARM RISC 处理器小引脚数 Flash 微处理器家族的一员。它拥有 64K 字节的高速 Flash 和 16K 字节的 SRAM，丰富的外设资源，包括一个 USB 2.0 设备，使外部器件数目减至最低的完整系统功能集。

它包含一个 ARM7TDMI 高性能 RISC 核心，64KB 片上高速 flash（512 页，每页包含 128 字节），16KB 片内高速静态 RAM，2 个同步串口（USART），USB 2.0 全速 Device 设备，3 个 16bit 定时器/计数器通道。

### 1.4.2 建立 RealView MDK 工程

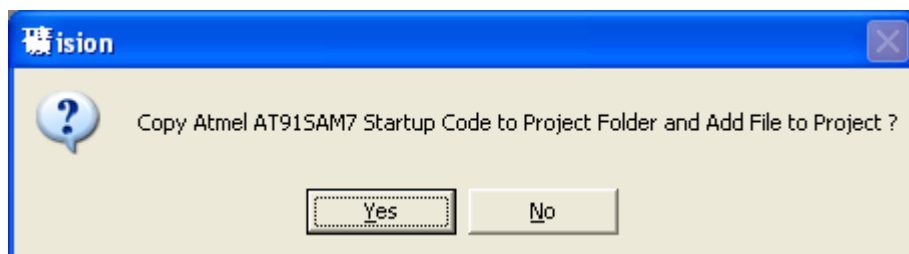
在 kernel/bsp 目录下新建 sam7s 目录。在 RealView MDK 中新建立一个工程文件（用菜单创建），名称为 project，保存在 kernel/bsp/sam7s 目录下。创建工程时一次的选项如下：  
CPU 选择 Atmel 的 AT91SAM7S64

图 A - 3 创建工程



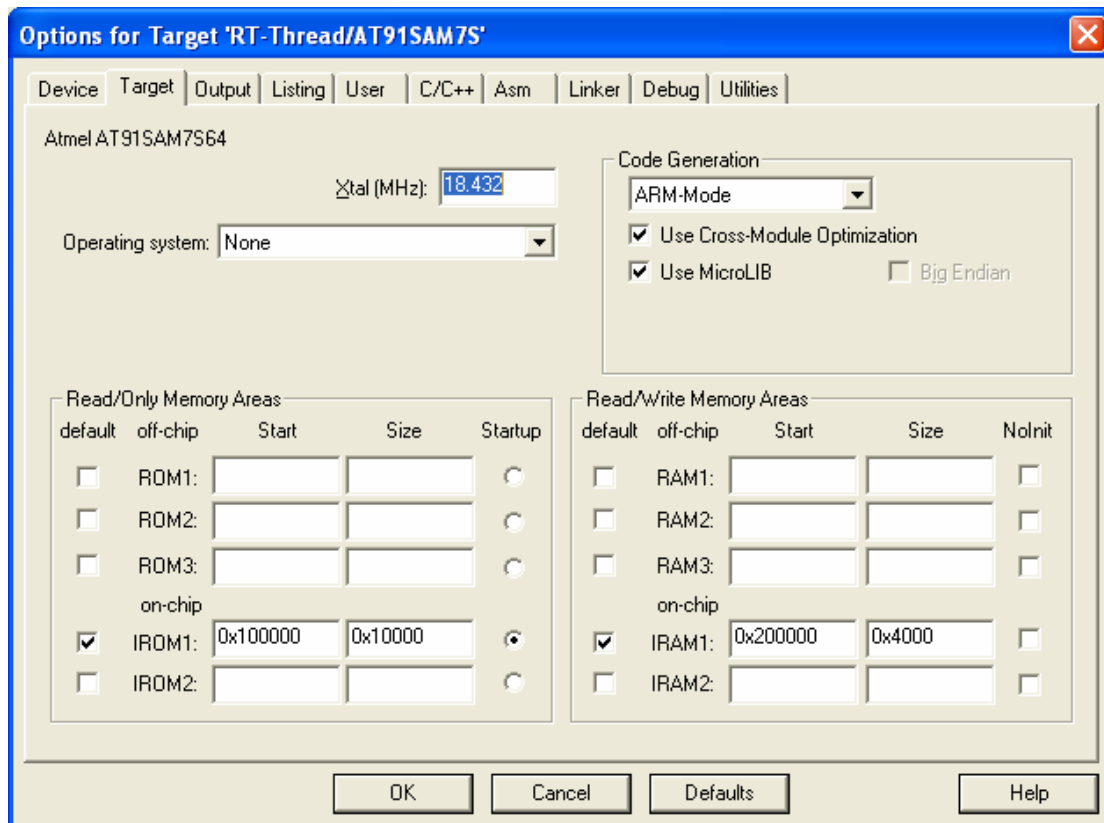
提问复制 Atmel AT91SAM7S 的启动代码到工程目录，确认 Yes

图 A - 4 添加启动汇编代码



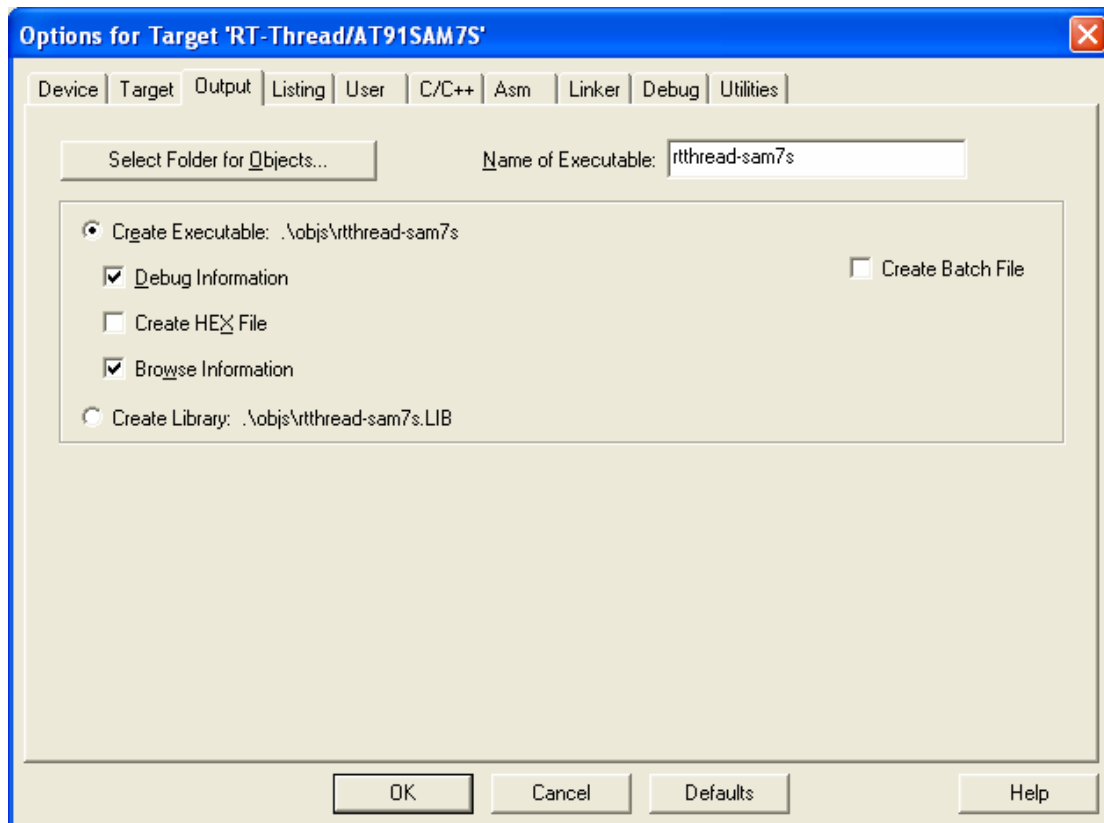
然后选择工程的属性，Code Generation 选择 ARM-Mode，如果希望产生更小的代码选择 Use Cross-Module Optimization 和 Use MicroLIB，如下图

图 A - 5 工程选项 - Target



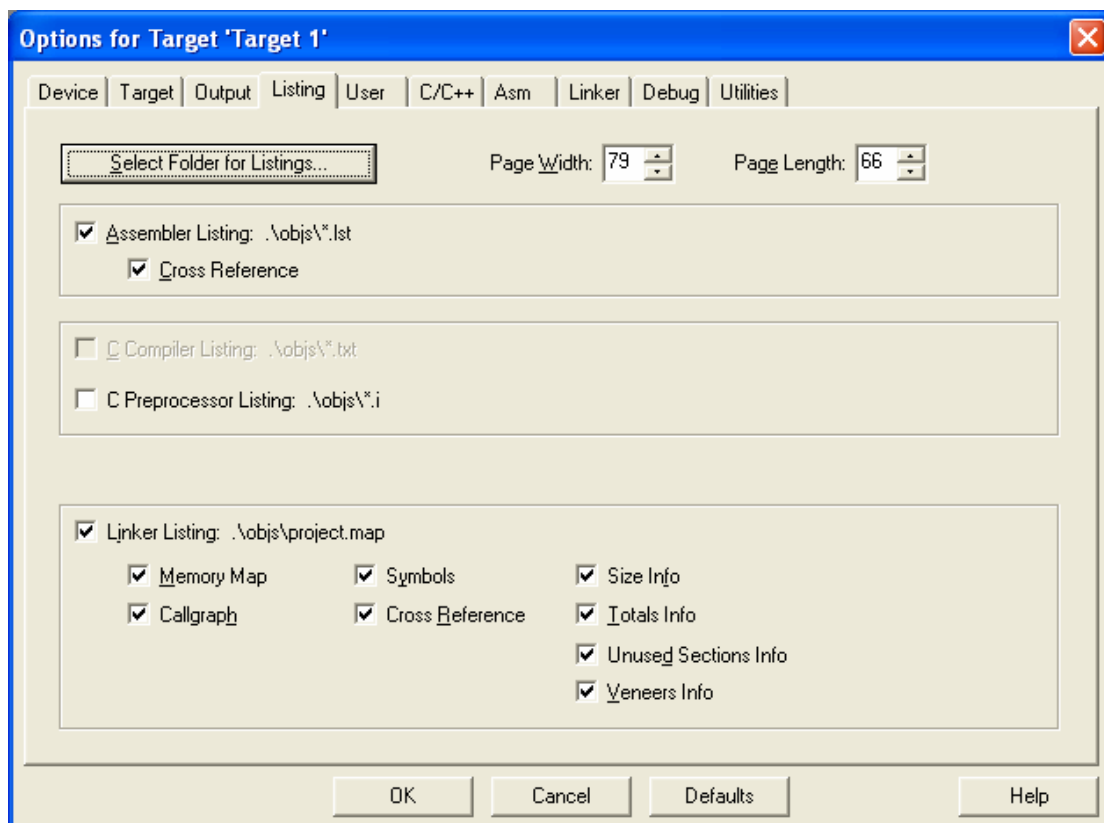
Select Folder for Objects 目录选择到 kernel/bsp/sam7s/objs，Name of Executable 为 rtthread-sam7s。

图 A - 6 工程选项 - Output



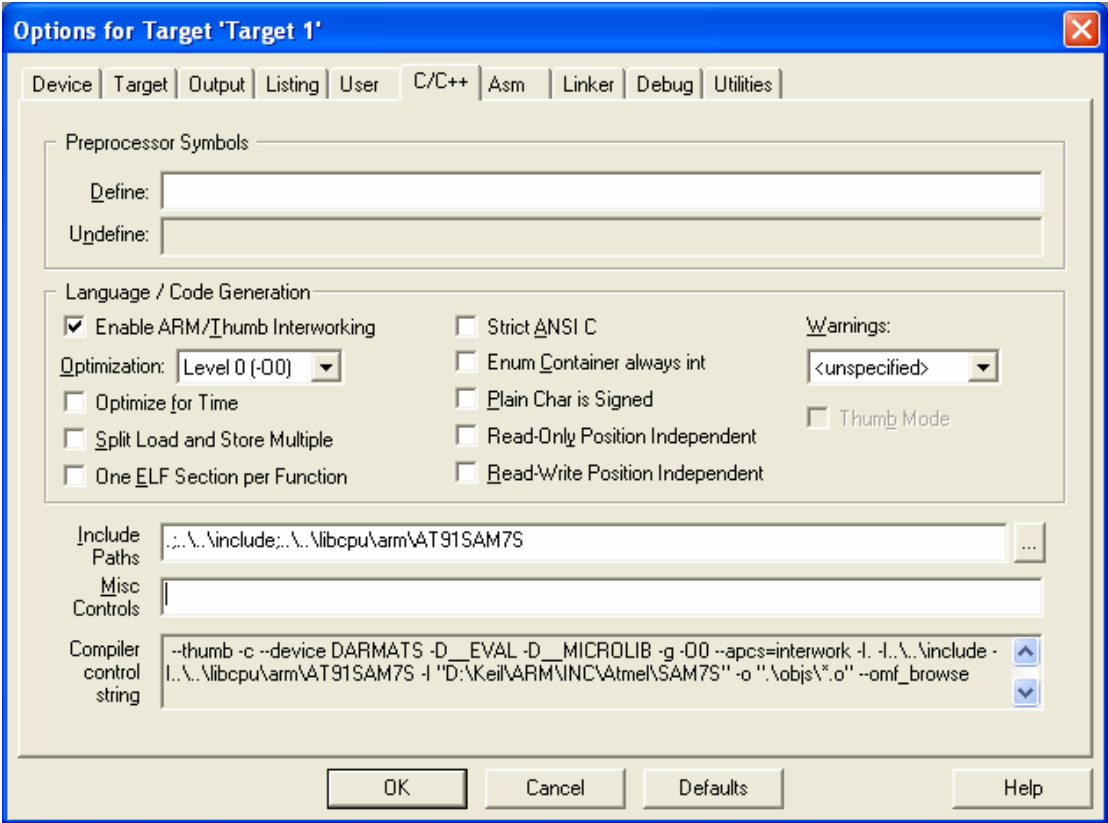
同样 Select Folder for Listings 选择 kernel/bsp/sam7s/objs 目录，如下图所示：

图 A - 7 工程选项 - Listing



C/C++编译选项标签页中，选择 Enable ARM/Thumb Interworking，Include Paths（头文件搜索路径）中加上目录 kernel\include，kernel\libcpu\arm\AT91SAM7S 以及 kernel\bsp\sam7s 目录，如下图所示：

图 A - 8 工程选项 – C/C++



Asm, Linker, Debug 和 Utilities 选项使用初始配置。

### 1.4.3 添加 RT-Thread 的源文件

对工程中初始添加的 Source Group1 改名为 Startup，并添加 Kernel，AT91SAM7S 的 Group，开始建立工程时产生的 SAM7.s 充命名为 start\_rvds.s 并放到 kernel\libcpu\AT91SAM7S 目录中。

Kernel Group 中添加所有 kernel\src 下的 C 源文件；  
Startup Group 中添加 startup.c，board.c 文件（放于 kernel\bsp\sam7s 目录中）；  
AT91SAM7S Group 中添加 context\_rvds.s，stack.c，trap.c，interrupt.c 等文件（放于 kernel\libcpu\sam7s 目录中）；

在 kernel/bsp/sam7s 目录中添加 rtconfig.h 文件，内容如下（详细的 RT-Thread 内核配置详见附录 B）：

代码 A - 16 AT91SAM7S64 的配置文件 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_THREAD_DEBUG */

/* 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 Faset Event*/
/* #define RT_USING_FASTEVENT */

/* 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 */

/* SECTION: Console options */
/* the buffer size of console*/
#define RT_CONSOLEBUF_SIZE 128

```



```

/* SECTION: FinSH shell options */
/* Using FinSH as Shell*/
/* #define RT_USING_FINSH */

/* SECTION: a mini libc */
/* Using mini libc library*/
#define RT_USING_MINILIBC

/* SECTION: C++ support */
/* Using C++ support*/
/* #define RT_USING_CPLUSPLUS */

#endif

```

## 1.4.4 线程上下文切换

### 代码 A - 17 context\_rvds.s

```

NOINT    EQU        0xc0; disable interrupt in psr

        AREA |.text|, CODE, READONLY, ALIGN=2
        ARM
        REQUIRE8
        PRESERVE8

; rt_base_t rt_hw_interrupt_disable();
; 关闭中断, 关闭前返回CPSR寄存器值
rt_hw_interrupt_disable PROC
    EXPORT rt_hw_interrupt_disable
    MRS r0, cpsr
    ORR r1, r0, #NOINT
    MSR cpsr_c, r1
    BX  lr
    ENDP

; void rt_hw_interrupt_enable(rt_base_t level);
; 恢复中断状态
rt_hw_interrupt_enable PROC
    EXPORT rt_hw_interrupt_enable
    MSR cpsr_c, r0
    BX  lr
    ENDP

; void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; r0 --> from
; r1 --> to
; 进行线程的上下文切换
rt_hw_context_switch PROC
    EXPORT rt_hw_context_switch
    STMFD sp!, {lr}          ; 把LR寄存器压入栈（这个函数返回后的下一个执行处）
    STMFD sp!, {r0-r12, lr}  ; 把R0 - R12以及LR压入栈

    MRS      r4, cpsr          ; 读取CPSR寄存器到R4寄存器
    STMFD    sp!, {r4}         ; 把R4寄存器压栈（即上一指令取出的CPSR寄存器）
    MRS      r4, spsr          ; 读取SPSR寄存器到R4寄存器
    STMFD    sp!, {r4}         ; 把R4寄存器压栈（即SPSR寄存器）

```

```

STR sp, [r0] ; 把栈指针更新到TCB的sp, 是由R0传入此函数

; 到这里换出线程的上下文都保存在栈中

LDR sp, [r1] ; 载入切换到线程的TCB的sp

; 从切换到线程的栈中恢复上下文, 次序和保存的时候刚好相反

LDMFD sp!, {r4} ; 出栈到R4寄存器 (保存了SPSR寄存器)
MSR spsr_cxsf, r4 ; 恢复SPSR寄存器
LDMFD sp!, {r4} ; 出栈到R4寄存器 (保存了CPSR寄存器)
MSR cpsr_cxsf, r4 ; 恢复CPSR寄存器

LDMFD sp!, {r0-r12, lr, pc} ; 对R0 - R12及LR、PC进行恢复
ENDP

; void rt_hw_context_switch_to(rt_uint32 to);
; r0 --> to
; 此函数只在系统进行第一次发生任务切换时使用, 因为是从没有线程的状态进行切换
; 实现上, 刚好是rt_hw_context_switch的下半截
rt_hw_context_switch_to PROC
EXPORT rt_hw_context_switch_to
LDR sp, [r0] ; 获得切换到线程的SP指针

LDMFD sp!, {r4} ; 出栈R4寄存器 (保存了SPSR寄存器值)
MSR spsr_cxsf, r4 ; 恢复SPSR寄存器
LDMFD sp!, {r4} ; 出栈R4寄存器 (保存了CPSR寄存器值)
MSR cpsr_cxsf, r4 ; 恢复CPSR寄存器

LDMFD sp!, {r0-r12, lr, pc} ; 恢复R0 - R12, LR及PC寄存器
ENDP

IMPORT rt_thread_switch_interrupt_flag
IMPORT rt_interrupt_from_thread
IMPORT rt_interrupt_to_thread

; void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);
; 此函数会在调度器中调用, 在调度器做上下文切换前会判断是否处于中断服务模式中, 如果
; 是则调用rt_hw_context_switch_interrupt函数 (设置中断中任务切换标志)
; 否则调用 rt_hw_context_switch函数 (进行真正的线程上线文切换)
rt_hw_context_switch_interrupt PROC
EXPORT rt_hw_context_switch_interrupt
LDR r2, =rt_thread_switch_interrupt_flag
LDR r3, [r2] ; 载入中断中切换标志地址
CMP r3, #1 ; 等于 1 ?
BEQ _reswitch ; 如果等于1, 跳转到_reswitch
MOV r3, #1 ; 设置中断中切换标志位1
STR r3, [r2] ; 保存到标志变量中
LDR r2, =rt_interrupt_from_thread
STR r0, [r2] ; 保存切换出线程栈指针
_reswitch
LDR r2, =rt_interrupt_to_thread
STR r1, [r2] ; 保存切换到线程栈指针
BX lr
ENDP

END

```

## 1.4.5 启动汇编文件

启动汇编文件可直接在 RealView MDK 新创建的 SAM7.s 文件上进行修改得到，把它重命名（为了和 RT-Thread 的文件命名规则保持一致）为 start\_rvds.s。修改主要有几点：

- 默认 IRQ 中断是由 RealView 的库自己处理的，RT-Thread 需要截获下来进行做操作系统级的调度；
- 自动生成的 SAM7.s 默认对 Watch Dog 不做处理，修改成 disable 状态（否则需要在代码中加入相应代码）；
- 在汇编文件最后跳转到 RealView 的库函数 \_\_main 时，会提前转到 ARM 的用户模式，RT-Thread 需要维持在 SVC 模式；
- 和 GNU GCC 的移植类似，需要添加中断结束后的线程上下文切换部分代码。

代码 A-8 是启动汇编的代码清单，其中加双下划线部分是修改的部分。

### 代码 A - 18 start\_rvds.s

```
;/*****  
;/* SAM7.S: Startup file for Atmel AT91SAM7 device series */  
;/*****  
;/* <<< Use Configuration Wizard in Context Menu >>> */  
;/*****  
;/* This file is part of the uVision/ARM development tools. */  
;/* Copyright (c) 2005-2006 Keil Software. All rights reserved. */  
;/* This software may only be used under the terms of a valid, current, */  
;/* end user licence from KEIL for a compatible version of KEIL software */  
;/* development tools. Nothing else gives you the right to use this software. */  
;/*****  
  
; /*  
; * The SAM7.S code is executed after CPU Reset. This file may be  
; * translated with the following SET symbols. In uVision these SET  
; * symbols are entered under Options - ASM - Define.  
; *  
; * REMAP: when set the startup code remaps exception vectors from  
; * on-chip RAM to address 0.  
; *  
; * RAM_INTVEC: when set the startup code copies exception vectors  
; * from on-chip Flash to on-chip RAM.  
; */  
  
; Standard definitions of Mode bits and Interrupt (I & F) flags in PSRs  
  
Mode_USR      EQU      0x10  
Mode_FIQ      EQU      0x11  
Mode_IRQ      EQU      0x12  
Mode_SVC      EQU      0x13  
Mode_ABT      EQU      0x17  
Mode_UND      EQU      0x1B  
Mode_SYS      EQU      0x1F  
  
I_Bit         EQU      0x80      ; when I bit is set, IRQ is disabled  
F_Bit         EQU      0x40      ; when F bit is set, FIQ is disabled
```

```

; Internal Memory Base Addresses
FLASH_BASE    EQU    0x00100000
RAM_BASE      EQU    0x00200000

;/// <h> Stack Configuration (Stack Sizes in Bytes)
;/// <o0> Undefined Mode    <0x0-0xFFFFFFFF:8>
;/// <o1> Supervisor Mode  <0x0-0xFFFFFFFF:8>
;/// <o2> Abort Mode        <0x0-0xFFFFFFFF:8>
;/// <o3> Fast Interrupt Mode <0x0-0xFFFFFFFF:8>
;/// <o4> Interrupt Mode    <0x0-0xFFFFFFFF:8>
;/// <o5> User/System Mode  <0x0-0xFFFFFFFF:8>
;/// </h>

UND_Stack_Size EQU    0x00000000
SVC_Stack_Size EQU    0x00000080
ABT_Stack_Size EQU    0x00000000
FIQ_Stack_Size EQU    0x00000000
IRQ_Stack_Size EQU    0x00000080
USR_Stack_Size EQU    0x00000400

ISR_Stack_Size EQU    (UND_Stack_Size + SVC_Stack_Size + ABT_Stack_Size + \
                        FIQ_Stack_Size + IRQ_Stack_Size)

                AREA    STACK, NOINIT, READWRITE, ALIGN=3

Stack_Mem      SPACE   USR_Stack_Size
__initial_sp   SPACE   ISR_Stack_Size
Stack_Top

;/// <h> Heap Configuration
;/// <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF>
;/// </h>

Heap_Size      EQU    0x00000000

                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem       SPACE   Heap_Size
__heap_limit

; Reset Controller (RSTC) definitions
RSTC_BASE      EQU    0xFFFFFD00    ; RSTC Base Address
RSTC_MR        EQU    0x08         ; RSTC_MR Offset

;/**
;/// <e> Reset Controller (RSTC)
;/// <o1.0>    URSTEN: User Reset Enable
;///          <i> Enables NRST Pin to generate Reset
;/// <o1.8..11> ERSTL: External Reset Length <0-15>
;///          <i> External Reset Time in 2^(ERSTL+1) Slow Clock Cycles
;/// </e>
;*/
RSTC_SETUP     EQU    1
RSTC_MR_Val     EQU    0xA5000401

```

```

; Embedded Flash Controller (EFC) definitions
EFC_BASE      EQU      0xFFFFF00      ; EFC Base Address
EFC0_FMR      EQU      0x60           ; EFC0_FMR Offset
EFC1_FMR      EQU      0x70           ; EFC1_FMR Offset

;/// <e> Embedded Flash Controller 0 (EFC0)
;/// <01.16..23> FMCN: Flash Microsecond Cycle Number <0-255>
;/// <i> Number of Master Clock Cycles in lus
;/// <01.8..9> FWS: Flash Wait State
;/// <0=> Read: 1 cycle / Write: 2 cycles
;/// <1=> Read: 2 cycle / Write: 3 cycles
;/// <2=> Read: 3 cycle / Write: 4 cycles
;/// <3=> Read: 4 cycle / Write: 4 cycles
;/// </e>
EFC0_SETUP    EQU      1
EFC0_FMR_Val  EQU      0x00320100

;/// <e> Embedded Flash Controller 1 (EFC1)
;/// <01.16..23> FMCN: Flash Microsecond Cycle Number <0-255>
;/// <i> Number of Master Clock Cycles in lus
;/// <01.8..9> FWS: Flash Wait State
;/// <0=> Read: 1 cycle / Write: 2 cycles
;/// <1=> Read: 2 cycle / Write: 3 cycles
;/// <2=> Read: 3 cycle / Write: 4 cycles
;/// <3=> Read: 4 cycle / Write: 4 cycles
;/// </e>
EFC1_SETUP    EQU      0
EFC1_FMR_Val  EQU      0x00320100

; Watchdog Timer (WDT) definitions
WDT_BASE      EQU      0xFFFFFD40    ; WDT Base Address
WDT_MR        EQU      0x04          ; WDT_MR Offset

;/// <e> Watchdog Timer (WDT)
;/// <01.0..11> WDV: Watchdog Counter Value <0-4095>
;/// <01.16..27> WDD: Watchdog Delta Value <0-4095>
;/// <01.12> WDFIEN: Watchdog Fault Interrupt Enable
;/// <01.13> WDRSTEN: Watchdog Reset Enable
;/// <01.14> WDRPROC: Watchdog Reset Processor
;/// <01.28> WDBGHLT: Watchdog Debug Halt
;/// <01.29> WDIDLEHLT: Watchdog Idle Halt
;/// <01.15> WDDIS: Watchdog Disable
;/// </e>
WDT_SETUP     EQU      1
WDT_MR_Val    EQU      0x00008000

; Power Mangement Controller (PMC) definitions
PMC_BASE      EQU      0xFFFFFC00    ; PMC Base Address
PMC_MOR       EQU      0x20          ; PMC_MOR Offset
PMC_MCFR      EQU      0x24          ; PMC_MCFR Offset
PMC_PLLR      EQU      0x2C          ; PMC_PLLR Offset
PMC_MCKR      EQU      0x30          ; PMC_MCKR Offset
PMC_SR        EQU      0x68          ; PMC_SR Offset
PMC_MOSCEN    EQU      (1<<0)        ; Main Oscillator Enable
PMC_OSCBYPASS EQU      (1<<1)        ; Main Oscillator Bypass
PMC_OSCOUNT    EQU      (0xFF<<8)    ; Main OSCillator Start-up Time
PMC_DIV       EQU      (0xFF<<0)    ; PLL Divider

```

```

PMC_PLLCOUNT    EQU    (0x3F<<8)        ; PLL Lock Counter
PMC_OUT           EQU    (0x03<<14)       ; PLL Clock Frequency Range
PMC_MUL           EQU    (0x7FF<<16)      ; PLL Multiplier
PMC_USBDIV        EQU    (0x03<<28)       ; USB Clock Divider
PMC_CSS           EQU    (3<<0)            ; Clock Source Selection
PMC_PRES          EQU    (7<<2)            ; Prescaler Selection
PMC_MOSCS         EQU    (1<<0)            ; Main Oscillator Stable
PMC_LOCK          EQU    (1<<2)            ; PLL Lock Status
PMC_MCKRDY        EQU    (1<<3)            ; Master Clock Status

; // <e> Power Mangement Controller (PMC)
; // <h> Main Oscillator
; // <01.0>      MOSCEN: Main Oscillator Enable
; // <01.1>      OSCBYPASS: Oscillator Bypass
; // <01.8..15>  OSCCOUNT: Main Oscillator Startup Time <0-255>
; // </h>
; // <h> Phase Locked Loop (PLL)
; // <02.0..7>   DIV: PLL Divider <0-255>
; // <02.16..26> MUL: PLL Multiplier <0-2047>
; //             <i> PLL Output is multiplied by MUL+1
; // <02.14..15> OUT: PLL Clock Frequency Range
; //             <0=> 80..160MHz <1=> Reserved
; //             <2=> 150..220MHz <3=> Reserved
; // <02.8..13>  PLLCOUNT: PLL Lock Counter <0-63>
; // <02.28..29> USBDIV: USB Clock Divider
; //             <0=> None <1=> 2 <2=> 4 <3=> Reserved
; // </h>
; // <03.0..1>   CSS: Clock Source Selection
; //             <0=> Slow Clock
; //             <1=> Main Clock
; //             <2=> Reserved
; //             <3=> PLL Clock
; // <03.2..4>   PRES: Prescaler
; //             <0=> None
; //             <1=> Clock / 2 <2=> Clock / 4
; //             <3=> Clock / 8 <4=> Clock / 16
; //             <5=> Clock / 32 <6=> Clock / 64
; //             <7=> Reserved
; // </e>
PMC_SETUP         EQU    1
PMC_MOR_Val       EQU    0x00000601
PMC_PLLR_Val      EQU    0x00191C05
PMC_MCKR_Val      EQU    0x00000007

```

PRESERVE8

```

; Area Definition and Entry Point
; Startup Code must be linked first at Address at which it expects to run.

```

```

AREA    RESET, CODE, READONLY
ARM

```

```

; Exception Vectors
; Mapped to Address 0.
; Absolute addressing mode must be used.
; Dummy Handlers are implemented as infinite loops which can be modified.

```

```

Vectors      LDR      PC,Reset_Addr
              LDR      PC,Undef_Addr
              LDR      PC,SWI_Addr
              LDR      PC,PAbt_Addr
              LDR      PC,DAbt_Addr
              NOP                               ; Reserved Vector
              LDR      PC,IRQ_Addr
              LDR      PC,FIQ_Addr

Reset_Addr    DCD      Reset_Handler
Undef_Addr    DCD      Undef_Handler
SWI_Addr      DCD      SWI_Handler
PAbt_Addr     DCD      PAbt_Handler
DAbt_Addr     DCD      DAbt_Handler
              DCD      0                        ; Reserved Address
IRQ_Addr      DCD      IRQ_Handler
FIQ_Addr      DCD      FIQ_Handler

Undef_Handler B      Undef_Handler
SWI_Handler   B      SWI_Handler
PAbt_Handler  B      PAbt_Handler
DAbt_Handler  B      DAbt_Handler

; IRQ和FIQ的处理由操作系统截获，需要重新实现
; IRQ_Handler   B      IRQ_Handler

FIQ_Handler   B      FIQ_Handler

; Reset Handler

EXPORT Reset_Handler
Reset_Handler

; Setup RSTC
IF RSTC_SETUP != 0
LDR R0, =RSTC_BASE
LDR R1, =RSTC_MR_Val
STR R1, [R0, #RSTC_MR]
ENDIF

; Setup EFC0
IF EFC0_SETUP != 0
LDR R0, =EFC_BASE
LDR R1, =EFC0_FMR_Val
STR R1, [R0, #EFC0_FMR]
ENDIF

; Setup EFC1
IF EFC1_SETUP != 0
LDR R0, =EFC_BASE
LDR R1, =EFC1_FMR_Val
STR R1, [R0, #EFC1_FMR]
ENDIF

; Setup WDT
IF WDT_SETUP != 0
LDR R0, =WDT_BASE
LDR R1, =WDT_MR_Val

```

```

        STR    R1, [R0, #WDT_MR]
    ENDIF

; Setup PMC
        IF     PMC_SETUP != 0
        LDR    R0, =PMC_BASE

; Setup Main Oscillator
        LDR    R1, =PMC_MOR_Val
        STR    R1, [R0, #PMC_MOR]

; Wait until Main Oscillator is stablilized
        IF     (PMC_MOR_Val:AND:PMC_MOSCEN) != 0
MOSCS_Loop    LDR    R2, [R0, #PMC_SR]
        ANDS   R2, R2, #PMC_MOSCS
        BEQ    MOSCS_Loop
        ENDIF

; Setup the PLL
        IF     (PMC_PLLR_Val:AND:PMC_MUL) != 0
        LDR    R1, =PMC_PLLR_Val
        STR    R1, [R0, #PMC_PLLR]

; Wait until PLL is stabilized
PLL_Loop      LDR    R2, [R0, #PMC_SR]
        ANDS   R2, R2, #PMC_LOCK
        BEQ    PLL_Loop
        ENDIF

; Select Clock
        IF     (PMC_MCKR_Val:AND:PMC_CSS) == 1      ; Main Clock Selected
        LDR    R1, =PMC_MCKR_Val
        AND    R1, #PMC_CSS
        STR    R1, [R0, #PMC_MCKR]
WAIT_Rdy1     LDR    R2, [R0, #PMC_SR]
        ANDS   R2, R2, #PMC_MCKRDY
        BEQ    WAIT_Rdy1
        LDR    R1, =PMC_MCKR_Val
        STR    R1, [R0, #PMC_MCKR]
WAIT_Rdy2     LDR    R2, [R0, #PMC_SR]
        ANDS   R2, R2, #PMC_MCKRDY
        BEQ    WAIT_Rdy2
        ELIF   (PMC_MCKR_Val:AND:PMC_CSS) == 3      ; PLL Clock Selected
        LDR    R1, =PMC_MCKR_Val
        AND    R1, #PMC_PRESC
        STR    R1, [R0, #PMC_MCKR]
WAIT_Rdy1     LDR    R2, [R0, #PMC_SR]
        ANDS   R2, R2, #PMC_MCKRDY
        BEQ    WAIT_Rdy1
        LDR    R1, =PMC_MCKR_Val
        STR    R1, [R0, #PMC_MCKR]
WAIT_Rdy2     LDR    R2, [R0, #PMC_SR]
        ANDS   R2, R2, #PMC_MCKRDY
        BEQ    WAIT_Rdy2
        ENDIF  ; Select Clock
        ENDIF  ; PMC_SETUP

; Copy Exception Vectors to Internal RAM

```



```

        IF      :DEF:RAM_INTVEC
        ADR      R8, Vectors          ; Source
        LDR      R9, =RAM_BASE        ; Destination
        LDMIA    R8!, {R0-R7}         ; Load Vectors
        STMIA    R9!, {R0-R7}         ; Store Vectors
        LDMIA    R8!, {R0-R7}         ; Load Handler Addresses
        STMIA    R9!, {R0-R7}         ; Store Handler Addresses
        ENDIF

; Remap on-chip RAM to address 0

MC_BASE EQU     0xFFFFF00           ; MC Base Address
MC_RCR EQU      0x00                ; MC_RCR Offset

        IF      :DEF:REMAP
        LDR      R0, =MC_BASE
        MOV      R1, #1
        STR      R1, [R0, #MC_RCR]   ; Remap
        ENDIF

; Setup Stack for each mode

        LDR      R0, =Stack_Top

; Enter Undefined Instruction Mode and set its Stack Pointer
        MSR      CPSR_c, #Mode_UND:OR:I_Bit:OR:F_Bit
        MOV      SP, R0
        SUB      R0, R0, #UND_Stack_Size

; Enter Abort Mode and set its Stack Pointer
        MSR      CPSR_c, #Mode_ABT:OR:I_Bit:OR:F_Bit
        MOV      SP, R0
        SUB      R0, R0, #ABT_Stack_Size

; Enter FIQ Mode and set its Stack Pointer
        MSR      CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit
        MOV      SP, R0
        SUB      R0, R0, #FIQ_Stack_Size

; Enter IRQ Mode and set its Stack Pointer
        MSR      CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
        MOV      SP, R0
        SUB      R0, R0, #IRQ_Stack_Size

; Enter Supervisor Mode and set its Stack Pointer
        MSR      CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit
        MOV      SP, R0
        SUB      R0, R0, #SVC_Stack_Size

; Enter User Mode and set its Stack Pointer
; 在跳转到__main函数前, 维持在SVC模式
; MSR      CPSR_c, #Mode_USR
IF      :DEF:__MICROLIB

EXPORT __initial_sp

ELSE

```

```

MOV     SP, R0
SUB     SL, SP, #USR_Stack_Size

ENDIF

; Enter the C code

IMPORT __main
LDR     R0, =__main
BX      R0

IMPORT rt_interrupt_enter
IMPORT rt_interrupt_leave
IMPORT rt_thread_switch_interrput_flag
IMPORT rt_interrupt_from_thread
IMPORT rt_interrupt_to_thread
IMPORT rt_hw_trap_irq

; IRQ处理的实现
IRQ_Handler PROC
EXPORT IRQ_Handler
stmfd   sp!, {r0-r12,lr}      ; 对R0 - R12, LR寄存器压栈
bl      rt_interrupt_enter     ; 通知RT-Thread进入中断模式
bl      rt_hw_trap_irq        ; 相应中断服务例程处理
bl      rt_interrupt_leave     ; 通知RT-Thread要离开中断模式

; 判断中断中切换是否置位, 如果是, 进行上下文切换
ldr     r0, =rt_thread_switch_interrput_flag
ldr     r1, [r0]
cmp     r1, #1
beq     rt_hw_context_switch_interrupt_do ; 中断中切换发生
; 如果跳转了, 将不会回来

ldmfd   sp!, {r0-r12,lr}      ; 恢复栈
subspc, lr, #4                ; 从IRQ中返回
ENDP

; void rt_hw_context_switch_interrupt_do(rt_base_t flag)
; 中断结束后的上下文切换
rt_hw_context_switch_interrupt_do PROC
EXPORT rt_hw_context_switch_interrupt_do
mov     r1, #0                ; 清楚中断中切换标志
str     r1, [r0]

ldmfd   sp!, {r0-r12,lr}      ; 先恢复被中断线程的上下文
stmfd   sp!, {r0-r3}          ; 对R0 - R3压栈, 因为后面会用到
mov     r1, sp                ; 把此处的栈值保存到R1
add     sp, sp, #16           ; 恢复IRQ的栈, 后面会跳出IRQ模式
sub     r2, lr, #4            ; 保存切换出线程的PC到R2

mrs     r3, spsr              ; 获得SPSR寄存器值
orr     r0, r3, #I_Bit|F_Bit
msr     spsr_c, r0            ; 关闭SPSR中的IRQ/FIQ中断

; 切换到SVC模式
msr     cpsr_c, #Mode_SVC

stmfd   sp!, {r2}             ; 保存切换出任务的PC

```

```

stmfd    sp!, {r4-r12,lr}      ; 保存R4 - R12, LR寄存器
mov r4,  r1                    ; R1保存有压栈R0 - R3处的栈位置
mov r5,  r3                    ; R3切换出线程的CPSR
ldmfd    r4!, {r0-r3}          ; 恢复R0 - R3
stmfd    sp!, {r0-r3}          ; R0 - R3压栈到切换出线程
stmfd    sp!, {r5}             ; 切换出线程CPSR压栈
mrs r4,  spsr
stmfd    sp!, {r4}             ; 切换出线程SPSR压栈

ldr r4,  =rt_interrupt_from_thread
ldr r5,  [r4]
str sp,  [r5]                  ; 保存切换出线程的SP指针

ldr r6,  =rt_interrupt_to_thread
ldr r6,  [r6]
ldr sp,  [r6]                  ; 获得切换到线程的栈

ldmfd    sp!, {r4}             ; 恢复SPSR
msr SPSR_cxsf, r4
ldmfd    sp!, {r4}             ; 恢复CPSR
msr CPSR_cxsf, r4

ldmfd    sp!, {r0-r12,lr,pc}   ; 恢复R0 - R12, LR及PC寄存器
ENDP

IF      :DEF: __MICROLIB

EXPORT  __heap_base
EXPORT  __heap_limit

ELSE

; User Initial Stack & Heap
AREA   |.text|, CODE, READONLY

IMPORT  __use_two_region_memory
EXPORT  __user_initial_stackheap
__user_initial_stackheap

LDR     R0, = Heap_Mem
LDR     R1, =(Stack_Mem + USR_Stack_Size)
LDR     R2, =(Heap_Mem +      Heap_Size)
LDR     R3, = Stack_Mem
BX      LR
ENDIF

END

```

## 1.4.6 中断处理文件

代码 A - 19 interrupt.c

```

#include <rtthread.h>
#include "AT91SAM7S.h"

#define MAX_HANDLERS 32

```

```

extern rt_uint32_t rt_interrupt_nest;

rt_uint32_t rt_interrupt_from_thread, rt_interrupt_to_thread;
rt_uint32_t rt_thread_switch_interrput_flag;

/* 默认的中断处理 */
void rt_hw_interrupt_handler(int vector)
{
    rt_kprintf("Unhandled interrupt %d occured!!!\n", vector);
}

/* 初始化中断控制器 */
void rt_hw_interrupt_init()
{
    rt_base_t index;

    /* 每个中断服务例程都设置到默认的中断处理上 */
    for (index = 0; index < MAX_HANDLERS; index++)
    {
        AT91C_AIC_SVR(index) = (rt_uint32_t)rt_hw_interrupt_handler;
    }

    /* 初始化线程在中断中切换的一些变量 */
    rt_interrupt_nest = 0;
    rt_interrupt_from_thread = 0;
    rt_interrupt_to_thread = 0;
    rt_thread_switch_interrput_flag = 0;
}

/* 屏蔽某个中断的API */
void rt_hw_interrupt_mask(int vector)
{
    /* disable interrupt */
    AT91C_AIC_IDCR = 1 << vector;

    /* clear interrupt */
    AT91C_AIC_ICCR = 1 << vector;
}

/* 去屏蔽某个中断的API */
void rt_hw_interrupt_umask(int vector)
{
    AT91C_AIC_IECR = 1 << vector;
}

/* 在相应的中断号上装载中断服务例程 */
void rt_hw_interrupt_install(int vector, rt_isr_handler_t new_handler,
rt_isr_handler_t *old_handler)
{
    if(vector >= 0 && vector < MAX_HANDLERS)
    {
        if (*old_handler != RT_NULL) *old_handler =
(rt_isr_handler_t)AT91C_AIC_SVR(vector);
        if (new_handler != RT_NULL) AT91C_AIC_SVR(vector) =
(rt_uint32_t)new_handler;
    }
}

```

## 1.4.7 操作系统时钟节拍处理

代码 A - 20 操作系统时钟节拍 (board.c 文件中)

```
#define TCK 1000                                /* Timer Clock */
#define PIV ((MCK/TCK/16)-1)                    /* Periodic Interval Value */

/* 时钟中断服务例程 */
void rt_hw_timer_handler(int vector)
{
    if (AT91C_PITC_PISR & 0x01)
    {
        /* 递增一个系统节拍 */
        rt_tick_increase();

        /* 确认中断 */
        AT91C_AIC_EOICR = AT91C_PITC_PIVR;
    }
    else
    {
        /* end of interrupt */
        AT91C_AIC_EOICR = 0;
    }
}

/* 目标板初始化函数 */
void rt_hw_board_init()
{
    /* 装载时钟中断*/
    rt_hw_interrupt_install(AT91C_ID_SYS, rt_hw_timer_handler, RT_NULL);

    /* 初始PIT以提供操作系统时钟节拍 */
    AT91C_PITC_PIMR = (1 << 25) | (1 << 24) | PIV;
    /* 去屏蔽中断以保证时钟节拍中断能够正常被处理 */
    rt_hw_interrupt_umask(AT91C_ID_SYS);
}
```