RT-Thread Documentation

Release 0.3.0

Bernard Xiong

April 15, 2009

CONTENTS

Contents:

CHAPTER

ONE

文档简介

CHAPTER

TWO

RT-THREAD快速入门

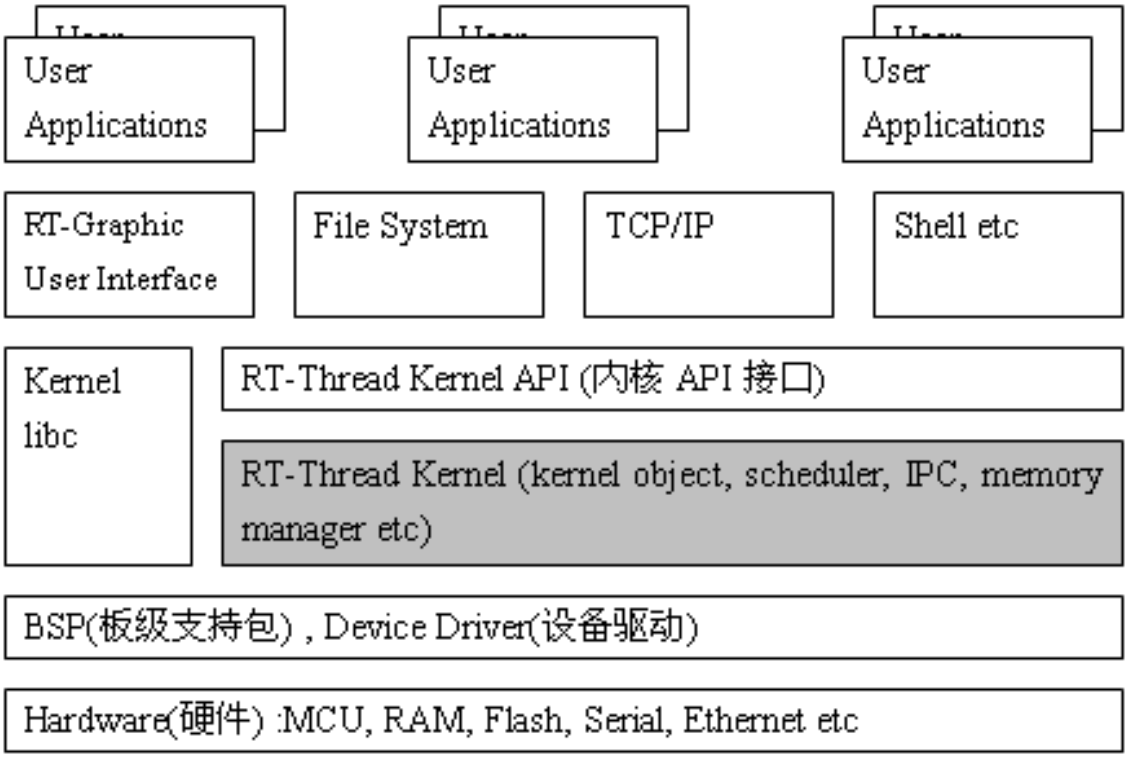
RT-Thread是一个开放源代码的实时操作系统，是由国内RT-Thread工作室发起的一个开源项目，许可证授权是GPL。读者可通过访问http://www.rt-thread.org来了解该项目的最新进展，包括在线的内核API参考及访问代码仓库获得最新的代码。

RT-Thread是一个面向实时的操作系统，同时兼顾嵌入式设备大多具备一定实时系统的概念，所以它也是一个嵌入式操作系统。

2.1 RT-Thread概述

在RT-Thread设计之初，就定下了宽度可伸缩的目标，既适合于一些资源非常紧张的系统，也适合于一些资源丰富对性能要求比较高的场合，甚至未来会支持多核高性能的系统。截止目前（2009年初），RT-Thread已经发展到了0.3.x版本，0.2.x系列版本也在稳步完善中。

RT-Thread包括了我们自主研发的强实时核心，它能够为上层服务、应用提供基础的实时支持，其总体结构如图所示：



本文主要介绍RT-Thread的实时核心以及一些移植方面的细节，各个组件会在后续的文档中陆续介绍。

2.2 RT-Thread内核功能概览

2.2.1 任务/线程调度

在RT-Thread中线程是最小的调度单位，调度算法是基于优先级的全抢占式多线程调度，支持256个线程优先级，0优先级代表最高优先级，255优先级留给空闲线程使用；支持创建相同优先级线程，相同优先级的线程采用可设置时间片的轮转调度算法；调度器寻找下一个最高优先级就绪线程的时间是恒定的(O(1))。系统不限制线程数量的多少，只和物理平台的具体内存相关。

2.2.2 任务同步机制

系统支持信号量、互斥锁作为线程间同步机制。互斥锁采用优先级继存方式以防止优先级翻转问题。信号量的释放动作可安全用于中断服务例程中。同步机制支持线程按优先级等待或按先进先出方式获取信号量或互斥锁。

2.2.3 任务间通信机制

系统支持事件、快速事件、邮箱和消息队列等通信机制。事件支持多事件"或触发"及"与触发"，适合于线程等待多个事件情况。快速事件支持事件队列，事件发生时确定哪个线程阻塞在相应事件上的时间是确定的。邮箱中一封邮件的长度固定为4字节，效率较消息队列要高效。通信设施中的发送动作可安全用于中断服务例程中。通信机制支持线程按优先级等待或按先进先出方式获取。

2.2.4 时间管理

系统使用时钟节拍来完成同优先级任务的时间片轮转调度；线程对内核对象的时间敏感性是通过系统定时器来实现的，此外，定时器也支持一次性超时及周期性超时。

2.2.5 内存管理

系统支持静态内存池管理及动态内存堆管理。从静态内存池中获取内存块时间恒定，而当内存池空时，可把申请内存块的线程阻塞(或立刻返回，或等待一段时间仍未获得返回，取决于内存块申请时设置的等待时间)，当其他线程释内存块到内存池时，将把阻塞线程唤醒。动态堆内存管理对于不同的系统资源情况，提供了面向小内存系统的管理算法及大内存系统的SLAB内存管理算法。

2.2.6 设备管理

系统实现了按名称访问的设备管理子系统，可按照统一的API界面访问硬件设备。在设备驱动接口上，根据嵌入式系统的特点，对不同的设备可以挂接相应的事件。

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targe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和表格 A - 2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。

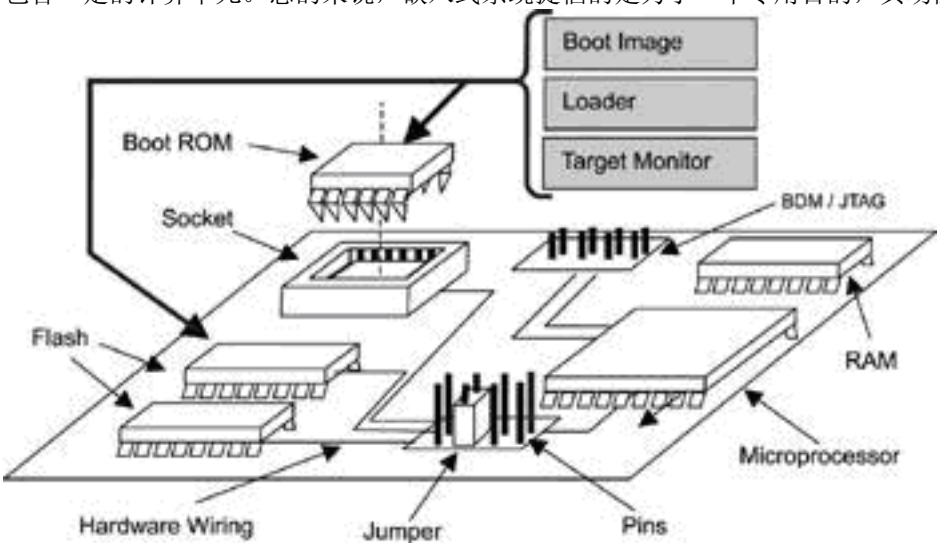
CHAPTER

THREE

实时系统

3.1 嵌入式系统

嵌入式系统是具备有特殊目的的计算系统，它具有特殊的需求，并运行预先定义好的任务。如常见的嵌入式系统：电视用的机顶盒，网络中的路由器等，它们都是为了一专用目的而设计的。从硬件资源上来讲，为了完成这一专有功能，嵌入式系统提供有限的资源，一般是恰到好处，在成本上满足一定的要求。从电子产品的角度来说，嵌入式系统最终会由一些芯片及电路组成，有时会包含一定的机械控制等，在控制芯片当中会包含一定的计算单元。总的来说，嵌入式系统提倡的是为了一个专用目的，其功能够用就好。

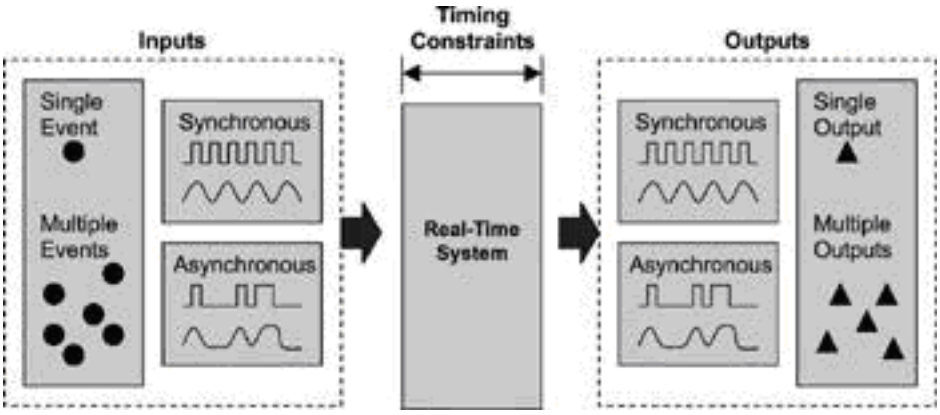


嵌入式系统

嵌入式系统中会包含微控制器，用于存放代码的Flash，Boot Rom，运行时代码用到的内存，调试时需要的JTAG接口等。

3.2 实时系统

实时计算可以定义成这样一类计算，即系统的正确性不仅取决于计算的逻辑结果，而且还依赖于产生结果的时间，关键有两点：正确地完成和在给定的时间内完成，而且两者重要性是等同的。而针对于在给定的时间内功能性的要求可以划分出常说的两类实时系统，软实时和硬实时系统。可以先看一个示例图：



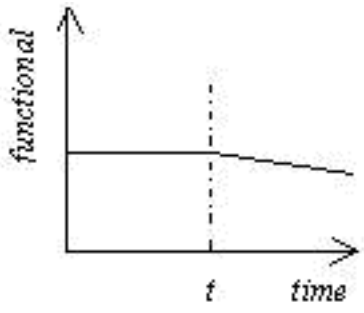
对于输入的信号、事件，实时系统应该能够在规定的时间内得到正确的响应，而不管这些事件是单一事件、多重事件还是同步信号或异步信号。对于一个具体的例子，可以考虑子弹射向玻璃杯的问题：一颗子弹从20米处射出，射向一个玻璃杯。假设子弹的速度是v米/秒，那么经过t1=20/v秒后，子弹将击碎玻璃杯。而有一系统在看见子弹射出后，将把玻璃杯拿走，假设这整个过程将持续t2秒的事件。如果t2 < t1，那么这个系统可以看成是一个实时系统。

和嵌入式系统类似，实时系统上也存在一定的计算单元，对系统的环境、里面的应用有所预计，也就是很多实时系统所说的确定性：对一个给定事件，在一给定的事件t秒内做出响应。对多个事件、多个输入的响应的确定性构成了整个实时系统的确定性。

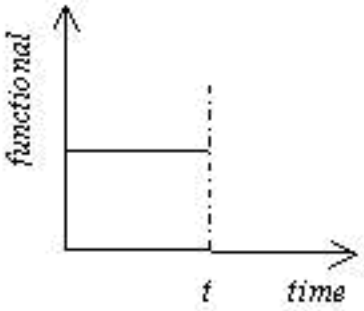
嵌入式系统的应用领域十分广泛，并不是其所针对的专用功能都要求实时性的，只有当系统中对任务有严格时间限定时，才有系统的实时性问题。具体的例子包括实验控制、过程控制设备、机器人、空中交通管制、远程通信、军事指挥与控制系统等。而对打印机这样一个嵌入式应用系统，人们并没有严格的时间限定，只有一个"尽可能快的"期望要求，因此，这样的系统称不上是实时系统。

3.3 软实时与硬实时

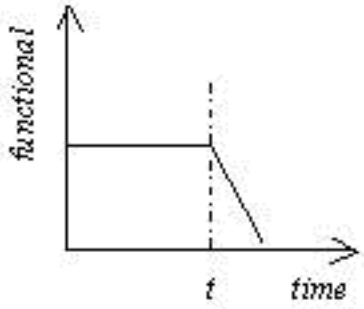
从上所述，实时系统是非常强调两点的：时间和功能的正确性。判断一个实时系统的正确性也正是这样，在给定的时间内正确地完成相应的任务。但也会有这种系统，偶尔也会在给定时间之外才能正确地完成任务，这种系统通常称为软实时系统。这也就构成了硬实时系统和软实时系统的区别。硬实时系统严格限定在给定的时间内完成任务，否则就可能导致灾难的发生，例如导弹的拦截，汽车引擎系统等。而软实时系统，可以允许一定的偏差，但是随着时间的偏移，整个系统的正确性也随之下降，例如一个DVD播放系统可以看成一个软实时系统，可以允许它偶尔的画面或声音延迟。



none real-time system



hard real-time system



soft real-time system

如下图1-2所示，从功能性、效用的角度上，实时系统可以看成：非实时系统随着给定时间t的推移，效用缓慢的下降。硬实时系统在给定时间t之后，马上变为零值。软实时系统随着给定时间t的推移，效用迅速的走向零值。

CHAPTER

FOUR

线程调度与管理

一个典型的简单软件系统会被设计成串行地运行：按照准确的指令步骤一次一个指令的运行。但是这种方法对于实时应用是不可行的，因为它们通常需要在固定的时间内处理多个输入输出，实时软件应用程序必须设计成一个并行的系统。

并行设计需要开发人员把一个应用分解成一个个小的，可调度的，序列化的程序单元。当正确的这样做时，并行设计能够让系统满足实时系统的性能及时间的要求。

4.1 实时系统的需求

实时系统中主要就是指固定的时间内正确的对外部事件做出响应。这个“时间内”，系统内部会做一些处理，例如获取数据后进行分析计算，加工处理等。而在这段时间之外，系统可能会闲下来，做一些空余的事。

例如一个手机终端，当一个电话拨入的时候，系统应当在固定的时间内发出振铃或声音提示以通知主人有来电，询问是否进行接听。而在非电话拨入的时候，人们可以用它进行一些其它工作，例如听听音乐，玩玩游戏。

从上面的例子我们可以看出，实时系统是一种倾向性的系统，对于实时的事件需要在第一时间做出回应，而对非实时任务则可以在实时事件到达时为之让路 – 被抢占。所以实时系统也可以看成是一个分级的系统，不同重要性的任务具有不同的优先级。

4.2 线程调度器

RT-Thread中提供的线程调度器是基于全抢占式优先级的调度，在系统中除了中断处理函数、调度器上锁部分的代码和禁止中断的代码是不可抢占的之外，系统的其他部分都是可以抢占的，包括线程调度器自身。系统总共支持256个优先级(0 ～ 255，255分配给空闲线程使用，一般不使用。在一些资源比较紧张的系统中，可以根据情况选择只支持32个优先级)。在系统中，当有比当前线程优先级还要高的线程就绪时，当前线程将立刻被换出，高优先级线程抢占处理机进行运行。

如图 2 - 1所示，RT-Thread调度器实现中包含一组，总共256个优先级队列组，每个优先级队列采用双向环形链表的方式链接，255优先级队列中一般只包含一个idle线程。如图例在优先级队列1#和2#中，分别有线程A – 线程C。由于线程A、B的优先级比线程C的高，所以此时线程C得不到运行，必须要等待线程A – 线程B都让出处理机后才能得到执行。图 2 - 1线程优先级队列

RT-Thread中允许创建相同优先级的线程。相同优先级的线程采用时间片轮转进行调度（也就是通常说的分时调度器）。如上图例中所示的线程A 和线程B，假设它们一次最大允许运行的时间片分别是10个时钟节拍和7个时钟节拍。那么线程B的运行需要在线程A运行完它的时间片（10个时钟节拍）后才能获得运行（当然如果线程A被挂起了，它会马上获得运行）。

由于RT-Thread调度器的实现是采用优先级链表的方式，所以系统中的总线程数不受限制，只和系统所能提供的内存资源相关。

为了保证系统的实时性，系统尽最大可能地保证高优先级的线程得以运行。线程调度的原则是一旦任务状态发生了改变，并且当前运行的线程优先级小于优先级队列组中线程最高优先级时，立刻进行线程切换（除非当前系统处于中断处理程序中或禁止线程切换的状态）。RT-Thread的调度器算法是基于优先级位图的算法，其时间复杂度是O(1)，即和线程的多少是不相关的。

4.3 线程控制块

线程控制块是操作系统用于控制线程的一块数据结构，它会存放线程的一些信息，例如优先级，线程名称等，也包含线程于线程之间的链表结构，线程等待事件集合等。

在RT-Thread中，线程控制块由结构体struct rt_thread（如下代码中所示）表示，另外一种写法是rt_thread_t，表示的是线程的句柄，在C的实现上是指向线程控制块的指针。代码 2 - 1 线程控制块

```
typedef struct rt_thread* rt_thread_t;

struct rt_thread
{
        /* rt object */
        char            name[RT_NAME_MAX];                      /* the name of thread */
        rt_uint8_t      type;                                   /* type of object */
        rt_uint8_t      flags;                                  /* thread's flags */

        rt_list_t       list;                                   /* the object list */

        rt_thread_t     tid;                                    /* the thread id */
        rt_list_t       tlist;                                  /* the thread list */

        /* stack point and entry */
        void*           sp;                                     /* stack point */
        void*           entry;                                  /* entry */
        void*           parameter;                              /* parameter */
        void*           stack_addr;                             /* stack address */
        rt_uint16_t stack_size;                                 /* stack size */

        /* error code */
        rt_err_t        error;                                  /* error code */

        /* priority */
        rt_uint8_t      current_priority;                       /* current priority */
        rt_uint8_t      init_priority;                          /* initialized priority */
#if RT_THREAD_PRIORITY_MAX > 32
        rt_uint8_t      number;
        rt_uint8_t      high_mask;
#endif
        rt_uint32_t     number_mask;

#if defined(RT_USING_EVENT) || defined(RT_USING_FASTEVENT)
        /* thread event */
        rt_uint32_t event_set;
        rt_uint8_t      event_info;
#endif

        rt_uint8_t      stat;                                   /* thread stat */

        rt_ubase_t      init_tick;                              /* thread's tick */
        rt_ubase_t      remaining_tick;                         /* remaining tick */
```

```
        struct rt_timer thread_timer;                           /* thread timer */

        rt_uint32_t user_data;                                  /* user data */
};
```

最后的一个成员user_data可由用户挂接一些数据信息到线程控制块中，以实现类似线程私有数据。

4.4 线程状态

在线程运行的过程中，在一个时间内只允许一个线程中处理器中运行，即线程会有多种不同的线程状态，如运行态，非运行态等。在RT-Thread中，线程包含三种状态，操作系统会自动根据它运行的情况而动态调整它的状态。

RT-Thread中的三种线程状态： RT_THREAD_INIT/CLOSE 线程初始状态。当线程刚开始创建还没开始运行时就处于这个状态；当线程运行结束时也处于这个状态。在这个状态下，线程会参与调度 RT_THREAD_SUSPEND 挂起态。线程此时被挂起：它可能因为资源不可用而等待挂起；或主动延时一段时间而被挂起。在这个状态下，线程不参与调度 RT_THREAD_READY 就绪态。线程正在运行；或当前线程运行完让出处理机后，操作系统寻找最高优先级的就绪态线程运行

RT-Thread RTOS提供一系列的操作系统调用接口，使得线程的状态在这三个状态之间来回的变换。例如一个就绪态的任务由于申请一个资源(例如使用rt_sem_take)，而有可能进入阻塞态。又如，一个外部中断发生，转入中断处理函数，中断处理函数释放了某个资源，导致了当前运行任务的切换，唤醒了另一阻塞态的任务，改变其状态为就绪态等等。

三种状态间的转换关系如下图所示：图 2 - 2线程状态转换图

线程通过调用函数rt_thread_create/init调用进入到初始状态（Initial State, RT_THREAD_INIT），通过函数rt_thread_startup调用后进入到就绪状态（Ready State，RT_THREAD_READY）。当这个线程调用rt_thread_delay，rt_sem_take，rt_mb_recv等函数时，将主动挂起或由于获取不到资源进入到挂起状态（Suspend State，RT_THREAD_SUSPEND）。在挂起状态的线程，如果它等待超时依然未获得资源或由于其他线程释放了资源，它将返回到就绪状态。

4.5 空闲线程

空闲线程是系统线程中一个比较特殊的线程，它具备最低的优先级，当系统中无其他线程可运行时，调度器将自动调度到空闲线程。空闲线程通常是一个死循环，永远不被挂起。在RT-Thread中空闲线程提供了钩子函数，以让系统在空闲的时候执行一定任务，例如系统运行指示灯闪烁等。

4.6 调度器相关接口

4.6.1 调度器初始化

在系统启动时需要执行调度器的初始化，以初始化调度器用到的一些全局变量。调度器初始化可以调用以下接口。 void rt_system_scheduler_init(void)

4.6.2 启动调度器

在系统完成初始化后切换到第一个线程，可以调用如下接口。 void rt_system_scheduler_start(void)

在调用这个函数时，它会查找系统中最高的就绪态的线程，然后切换过去运行。注：在调用这个函数前，必须先做idle线程的初始化。此函数是永远不会返回的。 2.6.3 执行调度让调度器执行一次线程的调度可通过如下接口。 void rt_schedule(void)

调用这个函数后，系统会计算一次系统中就绪态的线程，如果存在比当前线程更高优先级的线程时，系统将会切换到高优先级的线程去。通常情况下，用户不需要直接调用这个函数。

注：在中断服务例程中也可以调用这个函数，如果满足任务切换的条件，它会记录下中断前的线程及切换到更高优先级的线程，在中断服务例程处理完毕后执行正真的线程上下文切换。

4.7 线程相关接口

4.7.1 线程创建

一个线程要成为可执行的对象就必须由操作系统内核来为它创建/初始化一个线程句柄。可以通过如下的接口来创建一个线程。 rt_thread_t rt_thread_create (const char* name,

void (*entry)(void* parameter), void* parameter, rt_uint32_t stack_size, rt_uint8_t priority, rt_uint32_t tick)

在调用这个函数时，将为线程指定名称，线程入口位置，入口参数，线程栈大小，优先级及时间片大小。线程名称的最大长度由宏RT_NAME_MAX指定，多余部分会被自动截掉。栈大小的单位是字节，在大多数系统中需要做对齐（例如ARM体系结构中需要向4字节对齐）。线程的优先级范围从0 ～ 255，数值越小优先级越高。时间片的单位是操作系统的时钟节拍。

调用这个函数后，系统会从动态堆内存中分配一个线程句柄（或者叫做TCB）以及按照参数中指定的栈大小从动态堆内存中分配相应的空间。

创建一个线程的例子如下代码所示。代码 2 - 2 创建线程

```
#include <rtthread.h>

/* 线程入口 */
static void entry(void* parameter)
{
        rt_uint32_t count = 0;
        while (1)
        {
                rt_kprintf("count:%d\n", ++count);
                rt_thread_delay(50);
        }
}

/* 用户应用程序入口 */
int rt_application_init()
{
        /* 创建一个线程，入口时entry函数 */
        rt_thread_t thread = rt_thread_create("t1",
                entry, RT_NULL,
                1024, 200, 10);
        if (thread != RT_NULL)
                rt_thread_startup(thread);

        return 0;
}
```

4.7.2 线程删除

当需要删除用rt_thread_create创建出的线程时（例如线程出错无法恢复时），可以使用以下接口： rt_err_t rt_thread_delete (rt_thread_t thread)

调用该接口后，线程对象将会被移出线程队列并且从内核对象管理器中删除，线程占用的堆栈空间也会被释放。

注：线程运行完成，自动结束时，系统会自动删除线程。用rt_thread_init初始化的静态线程请不要使用此接口删除。

线程删除例子如下。代码 2 - 3 删除线程

```
#include <rtthread.h>

void thread1_entry(void* parameter)
{
        rt_uint32_t count = 0;
        while (1)
        {
                rt_kprintf("count:%d\n", ++count);
                rt_thread_delay(50);
        }
}

rt_uint32_t to_delete_thread1 = 0;
rt_thread_t thread1, thread2;
void thread2_entry(void* parameter)
{
        while (1)
        {
                if (to_delete_thread1 == 1)
                {
                        /* to_delete_thread1置位，删除thread1 */
                        rt_thread_delete(thread1);

                        /* 删除完成后退出 */
                        return ;
                }

                /* to_delete_thread1未置位，等待100个时钟节拍后再查询 */
                rt_thread_delay(100);
        }
}

int rt_application_init()
{
        /* 创建thread1并运行 */
        thread1 = rt_thread_create("t1",
                thread1_entry, RT_NULL,
                1024, 200, 10);
        if (thread1 != RT_NULL) rt_thread_startup(thread1);

        /* 创建thread2并运行 */
        thread2 = rt_thread_create("t2",
                thread2_entry, RT_NULL,
                1024, 120, 10);
        if (thread2 != RT_NULL) rt_thread_startup(thread2);
```

```
    return 0;
}
```

4.7.3 线程初始化

线程的初始化可以使用以下接口完成： rt_err_t rt_thread_init(struct rt_thread* thread,

const char* name, void (*entry)(void* parameter), void* parameter, void* stack_start, rt_uint32_t stack_size, rt_uint8_t priority, rt_uint32_t tick);

通常线程初始化函数用来初始化静态线程对象，线程句柄，线程栈由用户提供，一般都设置为全局变量在编译时被分配，内核不负责动态分配空间。

线程初始化例子如下所示。代码 2 - 4 线程初始化 #include <rtthread.h>

static rt_uint8_t thread_stack[512]; static struct rt_thread thread;

/* 线程入口 / static void entry(void* parameter) {

int i; rt_thread_t self;

/* 获得当前线程句柄 */ self = rt_thread_self();

while (1) {

rt_kprintf("thread[%s] count %dn", self->name, ++i); rt_thread_delay(100);

}

}

int rt_application_init() {

/* 初始化线程 */ rt_thread_init(&thread,

"thread1", entry, RT_NULL, &thread_stack[0], sizeof(thread_stack), 200, 10);

/* 启动线程 */ rt_thread_startup(&thread);

return 0;

}

4.7.4 线程脱离

脱离线程将使线程对象被从线程队列和内核对象管理器中删除。脱离线程使用以下接口。 rt_err_t rt_thread_detach (rt_thread_t thread)

注：这个函数接口是和rt_thread_delete相对应的，rt_thread_delete操作的对象是rt_thread_create创建的句柄，而rt_thread_detach操作的对象是使用rt_thread_init初始化的句柄。

4.7.5 线程启动

创建/初始化的线程对象的状态是RT_THREAD_INIT状态，并未进入调度队列，可以调用如下接口启动一个创建/初始化的线程对象： rt_err_t rt_thread_startup (rt_thread_t thread)

4.7.6 当前线程

在程序的运行过程中，相同的一段代码可能会被多个线程执行，在执行的时候可以通过下面的接口获得当前执行的线程句柄。 rt_thread_t rt_thread_self (void)

注：请不要在中断服务程序中调用此函数，因为它并不能准确获得当前的执行线程。

4.7.7 线程让出处理机

当前线程的时间片用完或者该线程自动要求让出处理器资源时，它不再占有处理机，调度器会选择下一个最高优先级的线程执行。这时，放弃处理器资源的线程仍然在就绪队列中。线程让出处理器使用以下接口：rt_err_t rt_thread_yield ()

调用该接口后，线程首先把自己从它所在的队列中删除，然后把自己挂到与该线程优先级对应的就绪线程链表的尾部，然后激活调度器切换到优先级最高的线程。

线程让出处理机代码例子如下所示。代码 2 - 5让出处理机

```
void funcion()
{
    ...
    rt_thread_yield();
    ...
}
```

注：rt_thread_yield函数和rt_schedule函数比较相像，但在相同优先级线程存在时，系统的行为是完全不一样的。当有相同优先级就绪线程存在，并且系统中不存在更高优先级的就绪线程时，执行rt_thread_yield函数后，当前线程被换出。而执行rt_schedule函数后，当前线程并不被换成。

4.7.8 线程睡眠

在实际应用中，经常需要线程延迟一段时间，指定的时间到达后，线程重新运行，线程睡眠使用以下接口：rt_err_t rt_thread_sleep(rt_tick_t tick) rt_err_t rt_thread_delay(rt_tick_t tick)

以上两个接口作用是相同的，调用该线程可以使线程暂时挂起指定时间，它接受一个参数，该参数指定了线程的休眠时间（时钟节拍数）。

4.7.9 线程挂起

当线程调用rt_thread_delay，rt_sem_take，rt_mb_recv等函数时，将主动挂起。或者由于线程获取不到资源，它也会进入到挂起状态。在挂起状态的线程，如果等待的资源超时或由于其他线程释放资源，它将返回到就绪状态。挂起线程使用以下接口： rt_err_t rt_thread_suspend (rt_thread_t thread)

注：如果挂起当前任务，需要在调用这个函数后，紧接着调用rt_schedule函数进行手动的线程上下文切换。

挂起线程的代码例子如下所示。代码 2 - 6 挂起线程代码

```
#include <rtthread.h>

/* 线程句柄 */
rt_thread_t thread = RT_NULL;

/* 线程入口 */
static void entry(void* parameter)
{
```

```
        while (1)
        {
                /* 挂起自己 */
                rt_thread_suspend(thread);

                /* 调用rt_thread_suspend虽然把thread从就绪队列中删除，
                 * 但代码依然在运行，需要手动让调度器调度一次
                 */
                rt_schedule();

                /* 运行到此处，相当于thread已经被唤醒 */
                rt_kprintf("thread is resumed\n");
        }
}

int rt_application_init()
{
        /* 创建线程 */
        thread = rt_thread_create("tid", entry, RT_NULL, 1024, 250, 20);

        /* 启动线程 */
        rt_thread_startup(thread);

        return 0;
}
```

4.7.10 线程恢复

线程恢复使得挂起的线程重新进入就绪状态。线程恢复使用以下接口： rt_err_t rt_thread_resume (rt_thread_t thread)

恢复挂起线程的代码例子如下所示。代码 2 - 7 恢复挂起线程

```
#include <rtthread.h>

rt_thread_t thread = RT_NULL;
void function ()
{
    ...
    rt_thread_resume(thread);
    ...
}
```

4.7.11 线程控制

rt_err_t rt_thread_control(rt_thread_t thread, rt_uint8_t cmd, void* arg)

4.7.12 初始化空闲线程

在系统调度器运行前，必须通过调用如下的函数初始化空闲线程。 void rt_thread_idle_init(void)

4.7.13 设置空闲线程钩子

可以调用如下的函数，设置空闲线程运行时执行的钩子函数。 void rt_thread_idle_set_hook(void (*hook)())

当空闲线程运行时会自动执行设置的钩子函数，由于空闲线程具有系统的最低优先级，所以只有在空闲时刻才会执行此钩子函数。空闲线程是一个线程状态永远为就绪状态的线程，因此挂入的钩子函数必须保证空闲线程永远不会处于挂起状态，例如rt_thread_delay，rt_sem_take等可能会导致线程挂起的函数不能使用。

CHAPTER

FIVE

内核对象模型

RT-Thread的内核对象模型是一种非常有趣的面向对象实现方式。通常操作系统核心都是采用C语言编写，而C语言一般称为是一种面向过程的语言。但人类对世界的认知更多偏向于类别模式，即按照世界中不同的物品的特性分门类别的方式组织。RT-Thread中包含了一个小型的对象系统，并且这个系统是采用C语言实现的。在了解RT-Thread内部或采用RT-Thread编程时非常有必要先熟悉RT-Thread的内核对象系统。

RT-Thread的内核映像文件在编译时会形成如下图所示的结构（以lumit4510为例）：其中主要包括了这么几段：表4-1　运行目标段段描述.text 代码正文段.data 数据段，用于放置带初始值的全局变量.rodata 只读数据段，用于放置只读的全局变量（常量）.bss bss段，用于放置未初始化的全局变量

图4-1　lumit4510运行时内存映像图

当系统运行时，这些段也会相应的映射到内存中。在RT-Thread系统初始化时，通常bss段会清零，而堆（Heap）则是除了以上这些段以外可用的内存空间（具体的地址空间在系统启动时由参数指定），系统运行时动态分配的内存块就在堆的空间中分配出来的，如程序4-1：程序4-1 rt_uint8_t* msg; msg = (rt_uint8_t*) rt_malloc (128); rt_memset(msg, 0, 128);

msg_ptr 指向的内存空间就是处于堆空间中的。

而一些全局变量则是存放于.data和.bss段中，.data存放的是具有初始值的全局变量（.rodata可以看成一个特殊的data段，是只读的），如程序4-2：程序4-2

```
#include <rtthread.h>

const static rt_uint32_t sensor_enable = 0x000000FE;
rt_uint32_t sensor_value;
rt_bool_t sensor_inited = RT_FALSE;

void sensor_init()
{
…
}
…
```

sensor_value存放在.bss段中，系统启动后会自动初始化成零。sensor_inited变量则存放在.data段中，而sensor_enable存放在.rodata端中。

在RT-Thread内核对象中分为两类：静态内核对象和动态内核对象。静态内核对象通常放在.data或.bss段中，在系统启动后在程序中初始化；动态内核对象则是从堆中创建的，而后手工做初始化。

内核对象API命名规则

rt_xxx_init

初始化系统静态内核对象，它只会对对象所在的内存块进行初始化，并添加到系统容器的链表中进行管理。

rt_xxx_detach 脱离内核对象容器管理，它会将对象从系统容器的链表中脱离出来，但不会去释放内存块。

rt_xxx_create 创建一个动态内核对象，它会先申请出一块空间，然后进行初始化并添加到系统容器的链表中进行管理。

rt_xxx_delete 它会将对象从系统容器的链表中删除出来，并是否所占用的内存块。

RT-Thread中操作系统级的设施都是一种内核对象，例如线程，信号量，互斥量，定时器等。如程序4-3中所示的静态线程和动态线程的例子：程序4-3 静态内核对象与动态内核对象

```
static rt_uint8_t thread1_stack[512];
static struct rt_thread thread1;

/* 线程1入口 */
void thread1_entry(void* parameter)
{
        int i;

        while (1)
        {
                for (i = 0; i < 10; i ++)
                {
                        rt_kprintf("%d\n", i);
                        rt_thread_delay(100);
                }
        }
}
/* 线程2入口 */
void thread2_entry(void* parameter)
{
        int count = 0;
        while (1)
        {
                rt_kprintf("Thread2 count:%d\n", ++count);
                rt_thread_delay(50);
        }
}

/* 用户应用程序入口 */
int rt_application_init()
{
        rt_thread_t thread2_ptr;

        rt_thread_init(&thread1,
                "thread1",
                thread1_entry, RT_NULL,
                &thread1_stack[0], sizeof(thread1_stack),
                200, 10);

        thread2_ptr = rt_thread_create("thread2",
                thread2_entry, RT_NULL,
                512, 250, 25);

        rt_thread_startup(&thread1);
        rt_thread_startup(thread2_ptr);

        return 0;
}
```

例子中，thread1是一个静态线程对象，而thread2是一个动态线程对象。thread1对象的内存空间，包括thread tcb: thread1，栈空间thread1_stack都是编译时刻决定的，因为都不存在初始值，都统一放在.bss段

中。thread2运行中用到的空间都是动态分配的，包括thread tcb（thread2_ptr指向的内容）和栈空间。

5.1 内核对象管理工作模式

图4-1 内核对象管理器结构图系统采用内核对象管理系统来访问/管理所有内核对象。内核对象包含了内核中绝大部分设施，而这些内核对象可以是静态分配的静态对象，也可以是从系统内存堆中分配的动态对象。通过内核对象系统，RT-Thread可以做到不依赖于具体的内存分配方式，伸缩性得到极大的扩展。RT-Thread RTOS使用C语言和汇编语言开发，本来不具备面向对象的特征。但这里，RT-Thread借用了面向对象语言中对象的概念，定义了对象管理模块。系统中线程同步和通信机制对象都是从object中继承而来，具备object的属性，并由系统的对象容器统一分配并管理。内核对象管理器结构如图4-1所示，RT-Thread内核对象包括：线程，信号量，互斥锁，事件，邮箱，消息队列和定时器，内存池。对象容器中包含了每类内核对象的信息，包括对象类型，大小等。对象容器给每类内核对象分配了一个链表，所有的内核对象都被链接到该链表上。对于每一种具体内核对象和对象控制块，除了基本结构外，还有自己的扩展属性（私有属性），例如，对于线程控制块，在此基类对象基础上进行扩展，增加了线程状态、优先级等属性。这些属性在基类对象的基本操作中不会用到，只有在与具体线程相关的函数才会使用。因此用面向对象的观点，我们可以认为每一种具体对象是抽象对象的派生，继承了基本对象的属性并在此基础上增加与自己相关的属性。图4-2显示了RT-Thread中各类内核对象的派生和继承关系。

图4-2 内核对象的继承和派生关系图在对象管理模块中，定义了通用的数据结构，用来保存各种对象的共同属性，各种具体对象只需要在此基础上加上自己的某些特别的属性，就可以清楚的表示自己的特征。这种设计方法的优点：

1. 提高了系统的可重用性和扩展性，增加新的对象类别很容易，只需要继承通用对象的属性再加少量扩展即可。
2. 提供统一的对象操作方式，简化了各种具体对象的操作，提高了系统的可靠性。

5.2 对象控制块

```
struct rt_object
{
        /* name of kernel object                */
        char      name[RT_NAME_MAX];
        /* type of kernel object                */
        rt_uint8_t  type;
        /* flag of kernel object1               */
        rt_uint8_t  flag;
        /* list pointer of kernel object        */
        rt_list_t       list;
};
```

5.2.1 内核对象接口

4.1.3.1 初始化系统对象在初始化各种内核对象之前，首先需对对象管理系统进行初始化。在系统中，每类内核对象都有一个静态对象容器，一个静态对象容器放置一类内核对象，初始化对象管理系统的任务就是初始化这些对象容器，使之能够容纳各种内核对象，初始化系统对象使用以下接口： void rt_system_object_init(void) 以下是对象容器的数据结构：

```
struct rt_object_information
{
```

```
        enum rt_object_class_type type;         /* object class type. */
        rt_list_t object_list;                  /* object list. */
        rt_size_t object_size;                  /* object size. */
};
```

一种类型的对象容器维护了一个对象链表object_list，所有对于内核对象的分配，释放操作均在该链表上进行。

4.1.3.2 初始化对象使用对象前须先对其进行初始化。初始化对象使用以下接口： void rt_object_init(struct rt_object* object, enum rt_object_class_type type, const char* name)

对象初始化，实际上就是把对象放入到其相应的对象容器中，即将对象插入到对象容器链表中。 4.1.3.3 脱离对象从内核对象管理器中脱离一个对象。脱离对象使用以下接口： void rt_object_detach(rt_object_t object)

使用该接口后，静态内核对象将从内核对象管理器中脱离，但是对象占用的内存不会被释放。 4.1.3.4 分配对象在当前内核中，共定义了十类内核对象，这些内核对象被广泛的用于线程的管理，线程之间的同步，通信等。因此，系统随时需要新的对象来完成这些操作，分配新对象使用以下接口： rt_object_t rt_object_allocate(enum rt_object_class_type type, const char* name)

使用以上接口，首先根据对象类型来获取对象信息，然后从内存堆中分配对象所需内存空间，然后对该对象进行必要的初始化，最后将其插入到它所在的对象容器链表中。 4.1.3.5 删除对象不再使用的对象应该立即被删除，以释放有限的系统资源。删除对象使用以下接口： void rt_object_delete(rt_object_t object)

使用以上接口时，首先从对象容器中脱离对象，然后释放对象所占用的内存。 4.1.3.6 查找对象通过指定的对象类型和对象名查找对象，查找对象使用以下接口： rt_object_t rt_object_find(enum rt_object_class_type type, const char* name)

使用以上接口时，在对象类型所对应的对象容器中遍历寻找指定对象，然后返回该对象，如果没有找到这样的对象，则返回空。 4.1.3.7 辨别对象判断指定对象是否是系统对象（静态内核对象）。辨别对象使用以下接口： rt_err_t rt_object_is_systemobject(rt_object_t object)

通常采用rt_object_init方式挂接到内核对象管理器中的对象是系统对象。

CHAPTER

SIX

线程间同步与通信

在多任务实时系统中，一项工作的完成往往可以通过多个任务来共同合作完成。例如，一个任务从数据采集器中读取数据，然后放到一个链表中进行保存。而另一个任务则从这个链表队列中把数据取出来进行分析处理，并把数据从链表中删除（一个典型的消费者与生产者的例子）。

当消费者任务取到链表的最末端的时候，此时生产者任务可能正在往末端添加数据，此时有可能生产者拿到的末节点被消费者任务给删除了。

正常的操作顺序应该是在一个任务删除或添加动作完成时再进行下一个动作，生产任务与消费任务之间需要协调动作，而对于操作/访问同一块区域，称为临界区。任务的同步方式有很多中，其核心思想是，在访问临界区的时候只允许一个任务运行。

6.1 关闭中断

关闭中断是禁止多任务访问临界区最简单的一种方式，即使是在分时操作系统中也是如此。当关闭中断的时候，就意味着当前任务不会被其他事件所打断（因为整个系统已经对外部事件不再响应），也就是当前任务不会被抢占，除非这个任务主动退出处理机。

关闭中断/恢复中断 API是由BSP提供。 - rt_base_t rt_hw_interrupt_disable() 关闭中断并返回关闭中断前的中断状态

void rt_hw_interrupt_enable(rt_base_t level) 使能中断，它采用恢复调用rt_hw_interrupt_disable前的中断状态进行恢复中断状态，如果调用rt_hw_interrupt_disable（）前是关中断状态，那么调用此函数后依然是关中断状态。

6.2 调度器上锁

同样把调度器锁住也能让任务不被换出，直到调度器解锁。但和关闭中断有一点不相同的是，对调度器上锁，系统依然能响应外部中断，中断服务例程依然有可能被运行。所以在使用调度器上锁的方式来做任务同步时，需要考虑好，任务访问的临界资源是否会被中断服务例程所修改。

RT-Thread提供的调度器操作API为： rt_enter_critical – 进入临界区调用这个函数后，调度器将被上锁。在系统锁住调度器的期间，系统依然响应中断，但因为中断而可能唤醒了高优先级的任务，调度依然不会选择高优先级的任务，直到调用解锁函数才尝试进行下一次调度。

rt_exit_critical – 退出临界区在系统退出临界区的时候，系统会计算当前是否有更高优先级的任务就绪，如果有将切换到新的任务中执行，否则继续执行当前任务。注：rt_enter_critical/rt_exit_critical可以多次嵌套调用，但有多少次rt_enter_critical就必须有成对的rt_exit_critical调用。

6.3 互斥量

互斥量是管理临界资源的一种有效手段。因为互斥量是独占的，所以在一个时刻只允许一个线程占有互斥量，利用这个性质来实现共享资源的互斥量保护。互斥量工作示意图如图4-5所示，任何时刻只允许一个线程获得互斥量对象，未能够获得互斥量对象的线程被挂起在该互斥量的等待线程队列上。

使用互斥量会导致的一个潜在问题就是线程优先级翻转。所谓优先级翻转问题即当一个高优先级线程通过互斥量机制访问共享资源时，该互斥量已被一低优先级线程占有，而这个低优先级线程在访问共享资源时可能又被其它一些中等优先级的线程抢先，因此造成高优先级线程被许多具有较低优先级的线程阻塞，实时性难以得到保证。例如：有优先级为A、B和C的三个线程，优先级A>B>C，线程A，B处于挂起状态，等待某一事件的发生，线程C正在运行，此时线程C开始使用某一共享资源S。在使用过程中，线程A等待的事件到来，线程A转为就绪态，因为它比线程C优先级高，所以立即执行。但是当线程A要使用共享资源S时，由于其正在被线程C使用，因此线程A被挂起切换到线程C运行。如果此时线程B等待的事件到来，则线程B转为就绪态。由于线程B的优先级比线程C高，因此线程B开始运行，直到其运行完毕，线程C才开始运行。只有当线程C释放共享资源S后，线程A才得以执行。在这种情况下，优先级发生了翻转，线程B先于线程A运行。这样便不能保证高优先级线程的响应时间。

在RT-Thread中实现的是优先级继承算法。优先级继承通过在线程C被阻塞期间提升线程A的优先级到线程C的优先级别从而解决优先级翻转引起的问题。这防止了A（间接地防止C）被B抢占。通俗地说，优先级继承协议使一个拥有资源的线程以等待该资源的线程中优先级最高的线程的优先级执行。当线程释放该资源时，它将返回到它正常的或标准的优先级。因此，继承优先级的线程避免了被任何中间优先级的线程抢占。

6.4 互斥量控制块

互斥量控制块的数据结构

```
struct rt_mutex
{
        struct rt_ipc_object parent;
        rt_base_t value;                /* value of mutex. */
        struct rt_thread* owner;        /* current owner of mutex. */
        rt_uint8_t original_priority;   /* priority of last thread hold the mutex. */
        rt_base_t hold;                 /* numbers of thread hold the mutex. */
};
```

rt_mutex对象从rt_ipc_object中派生，由IPC容器所管理。

6.5 互斥量相关接口

6.5.1 创建互斥量

创建一个互斥量时，内核首先创建一个互斥量控制块，然后完成对该控制块的初始化工作。创建互斥量使用以下接口： rt_mutex_t rt_mutex_create (const char* name, rt_uint8 flag)

使用该接口时，为互斥量指定一个名字，并指定互斥量标志，可以是基于优先级的或基于FIFO的。

6.5.2 删除互斥量

系统不再使用互斥量时，通过删除互斥量以释放系统资源。删除互斥量使用以下接口： rt_err_t rt_mutex_delete (rt_mutex_t mutex)

删除一个互斥量，必须确保该互斥量不再被使用。

6.5.3 初始化互斥量

系统选择静态内存管理方式时，系统会在编译时创建将会使用的各种内核对象，互斥量也会在此时被创建，此时使用互斥量就不再需要使用rt_mutex_create接口来创建它，而只需直接对内核在编译时创建的互斥量控制块进行初始化。初始化互斥量使用以下接口： rt_err_t rt_mutex_init (rt_mutex_t mutex, const char* name, rt_uint8 flag)

使用该接口时，需指定内核已在编译时分配的静态互斥量控制块，指定该互斥量名称以及互斥量标志。

6.5.4 脱离互斥量

脱离互斥量将使互斥量对象被从内核对象管理器中删除。脱离互斥量使用以下接口。 rt_err_t rt_mutex_detach (rt_mutex_t mutex)

使用该接口后，内核先唤醒所有挂在该互斥量上的线程（线程的返回值是-RT_ERROR），然后将该互斥量从内核对象管理器中删除。

6.5.5 获取互斥量

线程通过互斥量申请服务获取对互斥量的控制权。线程对互斥量的控制权是独占的，某一个时刻一个互斥量只能被一个线程控制。在RT-Thread中使用优先级继承算法来解决优先级翻转问题。成功获得该互斥量的线程的优先级将被提升到等待该互斥量资源的线程中优先级最高的线程的优先级，获取互斥量使用以下接口：rt_err_t rt_mutex_take (rt_mutex_t mutex, rt_int32 time)

如果互斥量没有被其他线程控制，那么申请该互斥量的线程将成功获得。如果互斥量已经被当前线程线程控制，则该互斥量的引用计数加一。如果互斥量已经被其他线程控制，则当前线程该互斥量上挂起等待，直到其他线程释放它或者等待时间超过指定的超时时间。

6.5.6 释放互斥量

当线程完成互斥资源的访问后，应尽快释放它占据的互斥量，使得其他线程能及时获取该互斥量。释放互斥量使用以下接口： rt_err_t rt_mutex_release(rt_mutex_t mutex)

使用该接口时，只有已经拥有互斥量控制权的线程才能释放它，每释放一次该互斥量，它的访问计数就减一。当该互斥量的访问计数为零时，它变为可用，等待在该信号量上的线程将被唤醒。如果线程的运行优先级被互斥量提升，那么当互斥量被释放后，线程恢复原先的优先级。

6.6 信号量

信号量是用来解决线程同步和互斥的通用工具，和互斥量类似，信号量也可用作资源互斥访问，但信号量没有所有者的概念，在应用上比互斥量更广泛。信号量比较简单，不能解决优先级翻转问题，但信号量是一种轻量级的对象，比互斥量小巧、灵活。因此在很多对互斥要求不严格的系统中，经常使用信号量来管理互斥资源。

信号量工作示意图如图4-6所示，每个信号量对象有一个信号量值和一个线程等待队列，信号量的值对应了信号量对象的实例数目，假如信号量值为5，则表示共有5个信号量实例可以被使用，当信号量实例数目为零时，再申请该信号量的对象就会被挂起在该信号量的等待队列上，等待可用的信号量实例。

6.6.1 信号量控制块

```
struct rt_semaphore
{
        struct rt_ipc_object parent;
        rt_base_t value;                /* value of semaphore. */
};
```

rt_semaphore对象从rt_ipc_object中派生，由IPC容器所管理。

6.7 信号量相关接口

6.7.1 创建信号量

当创建一个信号量时，内核首先创建一个信号量控制块，然后对该控制块进行基本的初始化工作，创建信号量使用以下接口： rt_sem_t rt_sem_create (const char* name, rt_uint32 value, rt_uint8 flag);

使用该接口时，需为信号量指定一个名称，并指定信号量初始值和信号量标志。

6.7.2 删除信号量

系统不再使用信号量时，通过删除信号量以释放系统资源。删除信号量使用以下接口： rt_err_t rt_sem_delete (rt_sem_t sem);

删除一个信号量，必须确保该信号量不再被使用。如果删除该信号量时，有线程正在等待该信号量，则先唤醒等待在该信号量上的线程（返回值为-RT_ERROR）。

6.7.3 初始化信号量

系统选择静态内存管理方式时，系统会在编译时创建将会使用的各种内核对象，信号量也会在此时被创建，此时使用信号量就不再需要使用rt_mutex_create接口来创建它，而只需直接对内核在编译时创建的互斥量控制块进行初始化。初始化互斥量使用以下接口： rt_err_t rt_sem_init (rt_sem_t sem, const char* name, rt_uint32 value, rt_uint8 flag)

使用该接口时，需指定内核分配的静态信号量控制块，指定信号量名称以及信号量标志。

6.7.4 脱离信号量

脱离信号量将使信号量对象被从内核对象管理器中删除。脱离信号量使用以下接口。 rt_err_t rt_mutex_detach (rt_mutex_t mutex)

使用该接口后，内核先唤醒所有挂在该信号量上的线程，然后将该信号量从内核对象管理器中删除。

6.7.5 获取信号量

线程通过获取信号量来获得信号量资源实例，当信号量值大于零时，它每次被申请获得，值都会减一，获取信号量使用以下接口： rt_err_t rt_sem_take (rt_sem_t sem, rt_int32 time)

如果信号量的值等于零，那么说明当前资源不可用，申请该信号量的线程就必须在此信号量上等待，直到其他线程释放该信号量或者等待时间超过指定超时时间。

6.7.6 获取无等待信号量

当用户不想在申请的信号量上等待时，可以使用无等待信号量，获取无等待信号量使用以下接口： rt_err_t rt_sem_trytake(rt_sem_t sem)

跟获取信号量接口不同的是，当线程申请的信号量资源不可用的时候，它不是等待在该信号量上，而是直接返回-RT_ETIMEOUT。

6.7.7 释放信号量

当线程完成信号量资源的访问后，应尽快释放它占据的信号量，使得其他线程能获得该信号量。释放信号量使用以下接口： rt_err_t rt_sem_release(rt_sem_t sem)

当信号量的值等于零时，信号量值加一，并且唤醒等待在该信号量上的线程队列中的首线程，由它获取信号量。

6.8 消息队列

消息队列是另一种常用的线程间通讯方式，它使得线程之间接收和发送消息不必同时进行，消息队列也有点类似管道，它临时保存线程从队列的一端发送来的消息，直到有接收者读取它们。

消息队列用于给线程发消息。如图4-13所示，通过内核提供的服务，线程或中断服务子程序可以将一条消息放入消息队列。同样，一个或多个线程可以通过内核服务从消息队列中得到消息。通常，先进入消息队列的消息先传给线程，也就是说，线程先得到的是最先进入消息队列的消息，即先进先出原则(FIFO)。消息队列由多个元素组成，内核用它们来管理队列。当消息队列被创建时，它就被分配了消息队列控制块，队列名，内存缓冲区，消息大小以及队列长度。内核负责给消息队列分配消息队列控制块，它同时也接收用户线程传入的参数，像消息队列名以及消息大小，队列长度，由这些来确定消息队列所需内存大小，当获得了这些信息后，内核就可以从内存堆或者线程私有内存空间中为消息队列分配内存。一个消息队列中本身包含着多个消息框，每个消息框可以存放一条消息，消息队列中的第一个和最后一个消息框被分别称为队首和队尾，对应了消息队列控制块中的msg_queue_head和msg_queue_tail，有些消息框中可能是空的，所有消息队列中的消息框总数就是消息队列的长度。用户线程可以在创建消息队列时指定这个长度。

6.8.1 消息队列控制块

```
struct rt_messagequeue
{
        struct rt_ipc_object parent;

        void* msg_pool;                 /* start address of message queue. */

        rt_size_t msg_size;             /* message size of each message. */
        rt_size_t max_msgs;             /* max number of messages. */

        void* msg_queue_head;   /* list head. */
        void* msg_queue_tail;   /* list tail. */
        void* msg_queue_free;   /* pointer indicated the free node of queue. */

        rt_ubase_t entry;               /* index of messages in the queue. */
};
```

rt_messagequeue对象从rt_ipc_object中派生，由IPC容器所管理。

6.9 消息队列相关接口

6.9.1 创建消息队列

图3-14 创建消息队列时的内部结构图创建消息队列时先创建一个消息队列对象控制块，然后给消息队列分配一块内存空间组织成空闲消息链表，这块内存大小等于消息大小与消息队列容量的乘积。然后再初始化消息队列，此时消息队列为空，如图所示。创建消息队列接口如下： rt_mq_t rt_mq_create (const char* name, rt_size_t msg_size, rt_size_t max_msgs, rt_uint8 flag)

创建消息队列时给消息队列指定一个名字，作为消息队列的标识，然后根据用户需求指定消息的大小以及消息队列的容量。如图3-14所示，消息队列被创建时所有消息都挂在空闲消息列表上，消息队列为空。

6.9.2 删除消息队列

当消息队列不再被使用时，应该删除它以释放系统资源，一旦操作完成，消息队列将被永久性的删除。删除消息队列接口如下： rt_err_t rt_mq_delete (rt_mq_t mq)

删除消息队列时，如果有线程被挂起在该消息队列等待队列上，则先唤醒挂起在该消息等待队列上的所有线程（返回值是-RT_ERROR），然后再释放消息队列使用的内存，最后删除消息队列对象。

6.9.3 脱离消息队列

脱离消息队列将使消息队列对象被从内核对象管理器中删除。脱离消息队列使用以下接口。 rt_err_t rt_mq_detach(rt_mq_t mq)

使用该接口后，内核先唤醒所有挂在该消息等待队列对象上的线程（返回值是-RT_ERROR），然后将该消息队列对象从内核对象管理器中删除。

6.9.4 初始化消息队列

图4-15 初始化消息队列时的内部结构图初始化消息队列跟创建消息队列类似，只是初始化消息队列用于静态内存管理模式，消息队列控制块来源于用户线程在系统中申请的静态对象。还与创建消息队列不同的是，此处消息队列对象所使用的内存空间是由用户线程提供的一个缓冲区空间，其余的初始化工作与创建消息队列时相同。初始化消息队列接口如下： rt_err_t rt_mq_init(rt_mq_t mq, const char* name, void *msgpool, rt_size_t msg_size, rt_size_t pool_size, rt_uint8 flag)

初始化消息队列时，该接口需要获得用户已经申请获得的消息队列控制块以及缓冲区指针参数，以及线程指定的消息队列名，消息大小以及消息队列容量。如图4-15所示，消息队列初始化后所有消息都挂在空闲消息列表上，消息队列为空。

6.9.5 发送消息

图3-16 消息队列在发送普通消息后的内部结构图线程或者中断服务程序都可以给消息队列发送消息，当发送消息时，内核先从空闲消息链表上取下一个空闲消息块，把线程或者中断服务程序发送的消息内容复制到消息块上，然后把该消息块挂到消息队列的尾部。发送者成功发送消息当且仅当空闲消息链表上有可用的空闲消息块；当自由消息链表上无可用消息块，说明消息队列中的消息已满，此时，发送消息的的线程或者中断程序会收到一个错误码。发送消息接口如下： rt_err_t rt_mq_send (rt_mq_t mq, void* buffer, rt_size_t size)

发送消息时，发送者需指定发送到哪个消息队列，并且指定发送的消息内容以及消息大小。如图3-16所示，在发送一个普通消息之后，空闲消息链表上的队首消息被转移到了消息队列尾。

6.9.6 发送紧急消息

图4-17 消息队列在发送紧急消息后的内部结构图发送紧急消息的过程与发送消息几乎一样，唯一的不同是，当发送紧急消息时，从空闲消息链表上取下来的消息块不是挂到消息队列的队尾，而是挂到队首，这样，接收者能够优先接收到紧急消息，从而及时进行消息处理。发送紧急消息的接口如下： rt_err_t rt_mq_urgent(rt_mq_t mq, void* buffer, rt_size_t size)

如图4-17所示，在发送一个紧急消息之后，空闲消息链表上的队首消息被转移到了消息队列首。

6.9.7 接收消息

图4-18 消息队列在接收消息后的内部结构图注：只有线程能够接收消息队列中的消息。只有当消息队列中有消息时，接收者才能接收消息，否则接收者会根据超时时间设置或挂起在消息队列的等待线程队列上，或直接返回。接收消息接口如下： rt_err_t rt_mq_recv (rt_mq_t mq, void* buffer, rt_size_t size, rt_int32 timeout)

接收消息时，接收者需指定存储消息的消息队列名,并且指定一个内存缓冲区，接收到的消息内容将被复制到该缓冲区，此外，还需指定未能及时取到邮件时的超时时间。如图4-18所示，接收一个消息后消息队列上的队首消息被转移到了空闲消息链表的尾部。

6.10 邮箱

通过内核服务可以给线程发送消息。典型的邮箱也称作交换消息，如图4-9所示，是用一个指针型变量，通过内核服务，一个线程或一个中断服务程序可以把一则消息(即一个指针)放到邮箱里去。同样，一个或多个线程可以通过内核服务接收这则消息。

RT-Thread采用的邮箱通信机制有点类型传统意义上的管道，用于线程间通讯。它是线程，中断服务，定时器向线程发送消息的有效手段。邮箱与线程对象等是独立的。线程，中断服务和定时器都可以向邮箱发送消息，但是只有线程能够接收消息。 RT-Thread的邮箱中共可存放固定条数的邮件，邮箱容量在创建邮箱时设定，每个邮件大小为32比特，刚好是一个整型数或一个指针大小。当实际需要发送的邮件较大时，可以传递指向一个缓冲区的指针。邮件格式不受限制，可以是任意格式的，可以是字符串，指针，二进制或其他格式。当邮箱满时，线程等不再发送新邮件。当邮箱空时，挂起正在试图接收邮件的线程，使其等待，当邮箱中有新邮件时，唤醒等待在邮箱上的线程，使其能够接收新邮件。

6.10.1 邮箱控制块

```
struct rt_mailbox
{
        struct rt_ipc_object parent;

        rt_uint32_t* msg_pool;  /* start address of message buffer. */

        rt_size_t size;                 /* size of message pool. */

        rt_ubase_t entry;               /* index of messages in msg_pool. */
        rt_ubase_t in_offset, out_offset;       /* in/output offset of the message buffer. */
};
```

rt_mailbox对象从rt_ipc_object中派生，由IPC容器所管理。 3.6.2 邮箱相关接口 3.6.2.1 创建邮箱创建邮箱对象时先创建一个邮箱对象控制块，然后给邮箱分配一块内存空间用来存放邮件，这块内存大小等于邮件大小与邮箱容量的乘积，接着初始化接收邮件和发送邮件在邮箱中的偏移量。创建邮箱的接口如下：rt_mailbox_t rt_mb_create (const char* name, rt_size_t size, rt_uint8 flag)

创建邮箱时给邮箱指定一个名称，作为邮箱的标识，并且指定邮箱的容量。 3.6.2.2 删除邮箱当邮箱不再被使用时，应该删除它以释放系统资源，一旦操作完成，邮箱将被永久性的删除。删除邮箱接口如下：rt_err_t rt_mb_delete (rt_mailbox_t mb)

删除邮箱时，如果有线程被挂起在该邮箱对象上，则先唤醒挂起在该邮箱上的所有线程，然后再释放邮箱使用的内存，最后删除邮箱对象。 3.6.2.3 脱离邮箱脱离邮箱将使邮箱对象被从内核对象管理器中删除。脱离邮箱使用以下接口。 rt_err_t rt_mb_detach(rt_mailbox_t mb)

使用该接口后，内核先唤醒所有挂在该邮箱上的线程，然后将该邮箱对象从内核对象管理器中删除。 3.6.2.4 初始化邮箱

图4-10 邮箱初始化后的内部结构初始化邮箱跟创建邮箱类似，只是初始化邮箱用于静态内存管理模式，邮箱控制块来源于用户线程在系统中申请的静态对象。还与创建邮箱不同的是，此处邮箱对象所使用的内存空间是由用户线程指定的一个缓冲区空间，用户把缓冲区的指针传递给邮箱对象控制块，其余的初始化工作与创建邮箱时相同。接口如下： rt_err_t rt_mb_init(rt_mailbox_t mb, const char* name, void* msgpool, rt_size_t size, rt_uint8 flag)

初始化邮箱时，该接口需要获得用户已经申请获得的邮箱对象控制块以及缓冲区指针参数，以及线程指定的邮箱名和邮箱容量。如图4-10所示，邮箱初始化后接收邮件偏移量In_offset，Out_offset均为零，邮箱容量size为15,邮箱中邮件数目entry为0。 3.6.2.5 发送邮件

图4-11 邮箱发送邮件后的内部结构线程或者中断服务程序通过邮箱可以给其他线程发送邮件，发送的邮件可以是32位任意格式的数据，一个整型值或者指向一个缓冲区的指针，发送者成功发送邮件当且仅当邮箱未满，当邮箱中的邮件满时，发送邮件的线程或者中断程序会收到一个错误码。发送邮件接口如下： rt_err_t rt_mb_send (rt_mailbox_t mb, rt_uint32 value)

发送邮件发送者需指定接收邮箱名称，并且指定发送的邮件内容。如图4-11所示，邮箱容量size为15，发送邮件之前邮件数entry为5，接收邮件偏移量为17，指向第三个邮件头。发送邮件偏移量为13，指向第13个邮件头。当连续发送两个邮件后，邮箱中邮件数变为7，接收邮件偏移量为19，指向第五个邮件头，发送邮件偏移量不变。 3.6.2.6 接收邮件

图4-12 邮箱接收邮件后的内部结构图只有线程能够接收邮箱中的邮件。只有当接收者接收的邮箱中有邮件时，接收者才能立即取到邮件，否则接收线程会根据超时时间设置或挂起在邮箱的等待线程队列上，或直接返回。接收邮件接口如下： rt_err_t rt_mb_recv (rt_mailbox_t mb, rt_uint32* value, rt_int32 timeout)

接收邮件时，接收者需指定接收邮件的邮箱,并指定接收到的邮件存放位置以及未能及时取到邮件时的超时时间。如图4-12所示，邮箱容量size为15，发送邮件之前邮件数entry为7，接收邮件偏移量为19，指向第五个邮件头。发送邮件偏移量为13，指向第13个邮件头。当连续发送两个邮件后，邮箱中邮件数变为6，接收邮件偏移量不变，接收邮件偏移量变为14，指向第14个邮件头。

3.7 事件事件主要用于线程间的同步，与信号量不同，它的特点是可以实现一对多，多对多的同步。可以是一个线程同步多个事件，也可以是多个线程同步多个事件。多个事件的集合用一个无符号整型变量来表示，变量中的一位代表一个事件，线程通过“逻辑与”或“逻辑或”与一个或多个事件建立关联。往往内核会定义若干个事件，称为事件集。事件在用于同步时有两种类型，一种是独立型同步，是指线程与任何事件之一发生同步（逻辑或关系），另一种是关联型同步，是指线程与若干事件都发生同步（逻辑与关系）。

图4-7 事件对象工作示意图 RT-Thread定义的事件有以下特点： 1. 事件只与线程相关，事件间相互独立，RT-Thread 定义的每个线程拥有32个事件标志，用一个32-bit无符号整形数记录，每一个bit代表一个事件。若干个事件构成一个事件集。 2. 事件仅用于同步，不提供数据传输功能 3. 事件无队列，即多次向线程发送同一事件，其效果等同于只发送一次。在RT-Thread中，每个线程还拥有一个事件信息标记，它有三个属性，分别是RT_EVENT_FLAG_AND（逻辑与），RT_EVENT_FLAG_OR（逻辑或）以及RT_EVENT_FLAG_CLEAR（清除标记）。当线程等待事件同步时，就可以通过32个事件标志和一个事件信息标记来判断当前接收的事件是否满足同步条件。如图4-7所示，线程1的事件标志中第三位和第十位被置位，如果事件信息标记位设为逻辑与，则表示线程1只有在事件3和事件10都发生以后才会被触发唤醒，如果事件信息标记位设为逻辑或，则事件3或事件10中的任意一个发生都会触发唤醒线程1。如果信息标记同时设置了清除标记位，则发生的事件会导致线程1的相应事件标志位被重新置位为零。

6.10.2 事件控制块

```
struct rt_event
{
        struct rt_ipc_object parent;

        rt_uint32_t set;                                /* event set. */
};
```

rt_event对象从rt_ipc_object中派生，由IPC容器所管理。 3.7.2 事件相关接口 3.7.2.1 创建事件当创建一个事件时，内核首先创建一个事件控制块，然后对该事件控制块进行基本的初始化，创建事件使用以下接口：rt_event_t rt_event_create (const char* name, rt_uint8 flag)

使用该接口时，需为事件指定名称以及事件标志。 3.7.2.2 删除事件系统不再使用事件对象时，通过删除事件对象控制块以释放系统资源。删除事件使用以下接口： rt_err_t rt_event_delete (rt_event_t event)

删除一个事件，必须确保该事件不再被使用，同时唤醒所有挂起在该事件上的线程。 3.7.2.3 脱离事件脱离信号量将使事件对象从内核对象管理器中删除。脱离事件使用以下接口。 rt_err_t rt_event_detach(rt_event_t event)

使用该接口后，内核首先唤醒所有挂在该事件上的线程，然后将该事件从内核对象管理器中删除。 3.7.2.4 初始化事件选择静态内存管理方式时，在编译时创建将会使用的各种内核对象，事件对象也会在此时被创建，此时使用事件就不再需要使用rt_event_create接口来创建它，而只需直接对内核在编译时创建的事件控制块进行初始化。初始化事件使用以下接口： rt_err_t rt_event_init(rt_event_t event, const char* name, rt_uint8 flag)

使用该接口时，需指定内核分配的静态事件对象，并指定事件名称和事件标志。 3.7.2.5 接收事件内核使用32位的无符号整型数来标识事件，它的每一位代表一个事件，因此一个事件对象可同时等待接收32个事件，内核可以通过指定选择参数“逻辑与”或“逻辑或”来选择如何激活线程，使用“逻辑与”参数表示只有当所有等待的事件都发生时激活线程，而使用“逻辑或”参数则表示只要有一个等待的事件发生就激活线程。接收事件使用以下接口： rt_err_t rt_event_recv(rt_event_t event, rt_uint32 set, rt_uint8 option, rt_int32 timeout, rt_uint32* recved)

用户线程首先根据选择参数和事件对象标志来判断它要接收的事件是否发生，如果已经发生，则根据线程要求选择是否重置事件对象的相应标志位，然后返回，如果没有发生，则把等待的事件标志位和选择参数填入线程本身的结构中，然后把线程挂起在此事件对象上，直到其等待的事件满足条件或等待时间超过指定超时时间。如果超时时间设置为零，则表示当线程要接受的事件没有满足其要求时不等待而直接返回。 3.7.2.6 发送事件通过发送事件服务，可以发送一个或多个事件。发送事件使用以下接口： rt_err_t rt_event_send(rt_event_t event, rt_uint32 set)

使用该接口时，通过set参数指定的事件标志重新设定event对象的事件标志值，然后遍历等待在event事件上的线程链表，判断是否有线程的事件激活要求与当前event对象事件标志值匹配，如果有，则激活该线程。 3.8 快速事件

图4-8 快速事件工作示意图快速事件对象和事件对象非常类似，快速事件对象也是32位整数，一个事件也是一个二进制位，每个事件位会拥有一个线程队列，当事件到来时，直接从队列中获得等待的线程并唤醒[所以寻找等待线程的时间是确定的]。快速事件对于通常的RTOS而言优势并不大，但会用于微内核中断处理中。如图4-8，线程1，和线程2因为等待事件7被挂起在事件7的线程队列上，线程3因为等待事件15而被挂在事件15的线程队列上，这时，有其他线程发送了一个事件7，于是线程1和线程2被从事件7的线程队列上唤醒，重新进入就绪队列。

6.10.3 快速事件控制块

```
struct rt_fast_event
{
        struct rt_object parent;
```

```
        rt_uint32 set;
        rt_list_t thread_list[RT_EVENT_LENGTH];
};
```

6.11 快速相关接口

6.11.1 创建快速事件

当创建一个快速事件时，内核首先创建一个快速事件控制块，并对该控制块进行必要的初始化。创建快速事件使用以下接口： rt_event_t rt_fast_event_create (const char* name, rt_uint8 flag)

使用该接口时，需指定一个事件名称及事件标志。

6.11.2 删除快速事件

系统不再使用快速事件对象时，通过删除事件对象控制块以释放系统资源。删除快速事件使用以下接口：rt_err_t rt_fast_event_delete (rt_fast_event_t event)

删除一个快速事件，必须确保该快速事件不再被使用，然后唤醒所有挂起在该快速事件上的线程。

6.11.3 脱离快速事件

脱离信号量将使快速事件对象被从内核对象管理器中删除。脱离快速事件使用以下接口。 rt_err_t rt_fast_event_detach(rt_fast_event_t event)

使用该接口后，内核先唤醒所有挂在该快速事件对象上的线程，然后将该快速事件对象从内核对象管理器中删除。

6.11.4 初始化快速事件

选择静态内存管理方式时，在编译时创建将会使用的各种内核对象，快速事件对象也会在此时被创建，此时使用事件就不再需要使用rt_fast_event_create接口来创建它，而只需直接对内核在编译时创建的快速事件控制块进行初始化。初始化事件使用以下接口： rt_err_t rt_fast_event_init(rt_fast_event_t event, const char* name, rt_uint8 flag)

使用该接口时，需要指定系统中已分配的静态快速事件对象，给该对象指定名称，并传入事件标志。

6.11.5 接收快速事件

系统使用32位的无符号整型数来标识快速事件，它的每一位代表一个快速事件，因此一个事件对象可同时等待接收32个事件，每个事件都对应有一个线程等待队列，接收快速事件使用以下接口： rt_err_t rt_fast_event_recv(rt_fast_event_t event, rt_uint8 bit, rt_uint8 option, rt_int32 timeout)

接收快速事件时，接收者首先根据快速事件对象标志来判断它要接收的事件是否发生，如果已经发生，则根据线程要求选择是否重置事件对象的相应标志位，然后返回，如果没有发生，则将自己挂起在此事件对应的线程等待队列上，直到其等待的事件发生或等待时间超过指定超时时间。如果超时时间设置为零，则表示当线程要接受的事件没有满足其要求时不等待而直接返回。

6.11.6 发送快速事件

通过发送快速事件服务，可以发送一个快速事件。发送快速事件使用以下接口： rt_err_t rt_fast_event_send(rt_fast_event_t event, rt_uint8 bit)

发送快速事件时，通过bit位来指定发送的事件，内核会遍历等待在该事件对应的等待线程队列，如果该队列上有等待的线程，则依次唤醒这些等待线程。

CHAPTER

SEVEN

内存管理

在计算系统中，一般需要一定的存储空间来存放变量或中间数据，按照存储方式来分类可以分为两种，内部存储空间和外部存储空间。内部存储空间通常访问速度比较快，能够随机访问（按照变量地址）。这一章主要讨论内部存储空间的管理。

实时系统中由于它对时间要求的严格性，其中的内存分配往往要比通用操作系统苛刻得多：首先，分配内存的时间必须是确定性的。一般内存管理算法是搜索一个适当范围的空间去寻找适合长度的空闲内存块，然而这对于实时系统是不可接受的，因为实时系统必须要保证内存块的分配必须在确定的时间内完成，否则实时任务在对外部事响应也将变得时间不可确定，例如一个处理外部数据的例子：当一个外部数据达到时（通过传感器，或者一些网络数据包），为了把它提交给上层的任务进行处理，它可能会先申请一块内存，把数据块的地址附加上，还可能有，数据长度，以及一些其他信息附加在一起（放在一个结构体中），然后提交给上层任务。

内存申请是当中的一个组成环节，如果因为使用的内存占用比较零乱，从而操作系统需要搜索一个不确定性长度的队列寻找适合的内存，那么申请时间将变得不可确定，进而对整个响应时间产生不可确定性。如果此时是一次导弹袭击，估计很可能会灰飞烟灭了！

其次，随着使用的内存分块被释放，整个内存区域会产生越来越多的碎片，从总体上来说，系统中还有足够的空闲内存，但因为它们不在一起，不能组成一块连续的内存块，从而造成程序不能申请到内存。对于通用系统而言，这种不恰当的内存分配算法可以通过重新启动系统来解决(每个月或者数个月进行一次)，但是这个对于嵌入式系统来说是不可接受的，他们通常需要连续不断地运行数年。

最后，嵌入式系统的资源环境也不是都相同，有些系统中资源比较紧张，只有数百KB的内存可供分配，而有些系统则存在数MB的内存。

RT-Thread操作系统在内存管理上，根据上层应用及系统资源的不同，有针对性的了数种内存分配算法：静态分区内存管理及动态内存管理。动态内存管理又划分为两种情况，一种是针对小内存块的分配管理，一种是针对大内存块的分配管理。

7.1 静态分区内存管理

7.1.1 内存池工作模式

图4-19 内存池管理结构示意图内存池（Memory Pool）是一种用于分配大量大小相同的小对象的技术。它可以极大加快内存分配/释放过程。

内存池在创建时向系统申请一大块内存，然后分成同样大小的多个小内存块，形成链表连接起来（此链表也称为空闲链表）。每次分配的时候，从空闲链表中取出头上一块，提供给申请者。如图4-19所示，物理内存中可以有多个大小不同的内存池，一个内存池由多个空闲内存块组成，内核用它们来进行内存管理。当一个内存池对象被创建时，内存池对象就被分配了内存池控制块：内存池名，内存缓冲区，内存块大小，块数以及一个等待线程队列。

内核负责给内存池分配内存池对象控制块，它同时也接收用户线程传入的参数，像内存块大小以及块数，由

这些来确定内存池对象所需内存大小，当获得了这些信息后，内核就可以从内存堆或者线程私有内存空间中为内存池分配内存。

7.1.2 内存池控制块

```
struct rt_mempool
{
        struct rt_object parent;

        void* start_address;                    /* memory pool start */
        rt_size_t size;                         /* size of memory pool */

        rt_size_t block_size;                   /* size of memory blocks */
        rt_uint8_t* block_list;                 /* memory blocks list */

        rt_size_t block_total_count;            /* numbers of memory block */
        rt_size_t block_free_count;             /* numbers of free memory block */

        rt_list_t suspend_thread;               /* threads pended on this pool */
        rt_size_t suspend_thread_count;         /* numbers of thread pended on this pool */
};
```

7.1.3 内存池接口

4.1.3.1 创建内存池创建内存池操作将会创建一个内存池对象并且从堆上分配一个内存池。创建内存池是分配，释放内存块的基础，创建该内存池后，线程便可以从内存池中完成申请，释放操作，创建内存池使用如下接口，接口返回一个已创建的内存池对象。 rt_mp_t rt_mp_create(const char* name, rt_size_t block_count, rt_size_t block_size);

使用该接口可以创建与需求相匹配的内存块大小和数目的内存池，前提是在系统资源允许的情况下。创建内存池时，需要给内存池指定一个名称。根据需要，内核从系统中申请一个内存池对象，然后从内存堆中分配一块由块数目和块大小计算得来的内存大小，接着初始化内存池对象结构，并将申请成功的内存缓冲区组织成可用于分配的空闲块链表。 4.1.3.2 删除内存池删除内存池将删除内存池对象并释放申请的内存。使用如下接口： rt_err_t rt_mp_delete(rt_mp_t mp)

删除内存池时，必须首先唤醒等待在该内存池对象上的所有线程，然后再释放已从内存堆上分配的内存，然后删除内存池对象。 4.1.3.3 初始化内存池初始化内存池跟创建内存池类似，只是初始化邮箱用于静态内存管理模式，内存池控制块来源于用户线程在系统中申请的静态对象。还与创建内存池不同的是，此处内存池对象所使用的内存空间是由用户线程指定的一个缓冲区空间，用户把缓冲区的指针传递给内存池对象控制块，其余的初始化工作与创建内存池相同。接口如下： rt_err_t rt_mp_init(struct rt_mempool* mp, const char* name, void *start, rt_size_t size, rt_size_t block_size)

初始化内存池时，把需要进行初始化的内存池对象传递给内核，同时需要传递的还有内存池用到的内存空间，以及内存池管理的内存块数目和块大小，并且给内存池指定一个名称。这样，内核就可以对该内存池进行初始化，将内存池用到的内存空间组织成可用于分配的空闲块链表。 4.1.3.4 脱离内存池脱离内存池将使内存池对象被从内核对象管理器中删除。脱离内存池使用以下接口。 rt_err_t rt_mp_detach(struct rt_mempool* mp)

使用该接口后，内核先唤醒所有挂在该内存池对象上的线程，然后将该内存池对象从内核对象管理器中删除。 4.1.3.5 分配内存块从指定的内存池中分配一个内存块，使用如下接口： void *rt_mp_alloc (rt_mp_t mp, rt_int32 time)

如果内存池中有可用的内存块，则从内存池的空闲块链表上取下一个内存块，减少空闲块数目并返回这个内存块，如果内存池中已经没有空闲内存块，则判断超时时间设置，若超时时间设置为零，则立刻返回空内存块，若等待大于零，则把当前线程挂起在该内存池对象上，直到内存池中有可用的自由内存块，或等待时间

到达。 4.1.3.6 释放内存块任何内存块使用完后都必须被释放，否则会造成内存泄露，释放内存块使用如下接口： void rt_mp_free (void *block)

使用以上接口时，首先通过需要被释放的内存块指针计算出该内存块所在的内存池对象，然后增加内存池对象的可用内存块数目，并把该被释放的内存块加入空闲内存块链表上。接着判断该内存池对象上是否有挂起的线程，如果有，则唤醒挂起线程链表上的首线程。

7.2 动态内存管理

动态内存管理是一个真实的堆内存管理模块，可以根据用户的需求（在当前资源满足的情况下）分配任意大小的内存块。RT-Thread系统中为了满足不同的需求，提供了两套动态内存管理算法，分别是小堆内存管理和SLAB内存管理。下堆内存管理模块主要针对系统资源比较少，一般小于2M内存空间的系统；而SLAB内存管理模块则主要是在系统资源比较丰富时，提供了一种近似的内存池管理算法。两种内存管理模块在系统运行时只能选择其中之一（或者完全不使用动态堆内存管理器），两种动态内存管理模块API形式完全相同。

注：不要在中断服务例程中分配或释放动态内存块。

7.2.1 小内存管理模块

小内存管理算法是一个简单的内存分配算法，当有可用内存的时候，会从中分割出一块来作为分配的内存，而余下的则返回到动态内存堆中。如图4-5所示

当用户线程要分配一个64字节的内存块时，空闲链表指针lfree初始指向0x0001EC00内存块，但此内存块只有32字节并不能满足要求，它会继续寻找下一内存块，此内存块大小为128字节满足分配的要求。分配器将把此内存块进行拆分，余下的内存块（52字节）继续留在lfree链表中。如图4-8所示

在分配的内存块前约12字节会存放内存分配器管理用的私有数据，用户线程不应访问修改它，这个头的大小会根据配置的对齐字节数稍微有些差别。

释放时则是相反的过程，但分配器会查看前后相邻的内存块是否空闲，如果空闲则合并成一个大的空闲内存块。

7.2.2 SLAB内存管理模块

RT-Thread 实现的SLAB分配器是在Matthew Dillon在DragonFly BSD中实现的SLAB分配器基础上针对嵌入式系统优化过的内存分配算法。原始的SLAB算法是Jeff Bonwick为 Solaris 操作系统首次引入的一种高效内核内存分配算法。

RT-Thread的SLAB分配器实现主要是去掉了其中的对象构造及析构过程，只保留了纯粹的缓冲型的内存池算法。SLAB分配器会根据对象的类型（主要是大小）分成多个区（zone），也可以看成每类对象有一个内存池，如图所示：

一个zone的大小在32k ～ 128k字节之间，分配器会在堆初始化时根据堆的大小自动调整。系统中最多包括72种对象的zone，最大能够分配16k的内存空间，如果超出了16k那么直接从页分配器中分配。每个zone上分配的内存块大小是固定的，能够分配相同大小内存块的zone会链接在一个链表中，而72种对象的zone链表则放在一个数组（zone_arry）中统一管理。

动态内存分配器主要的两种操作：内存分配：假设分配一个32字节的内存，SLAB内存分配器会先按照32字节的值，从zone_array链表表头数组中找到相应的zone链表。如果这个链表是空的，则向页分配器分配一个新的zone，然后从zone中返回第一个空闲内存块。如果链表非空，则这个zone链表中的第一个zone节点必然有空闲块存在（否则它就不应该放在这个链表中），然后取相应的空闲块。如果分配完成后，导致一个zone中所有空闲内存块都使用完毕，那么分配器需要把这个zone节点从链表中删除。

内存释放：分配器需要找到内存块所在的zone节点，然后把内存块链接到zone的空闲内存块链表中。如果此

时zone的空闲链表指示出zone的所有内存块都已经释放，即zone是完全空闲的zone。当中zone链表中，全空闲zone达到一定数目后，会把这个全空闲的zone释放到页面分配器中去。

7.2.3 动态内存接口

初始化系统堆空间

在使用堆内存时，必须要在系统初始化的时候进行堆内存的初始化，可以通过如下接口完成：.. code-block:: c

```
void rt_system_heap_init(void* begin_addr, void* end_addr);
```

入口参数分别为堆内存的其实地址和结束地址。

分配内存块

从内存堆上分配用户线程指定大小的内存块，接口如下： void* rt_malloc(rt_size_t nbytes);

用户线程需指定申请的内存空间大小，成功时返回分配的内存块地址，失败时返回RT_NULL。

重分配内存块

在已分配内存块的基础上重新分配内存块的大小（增加或缩小），可以通过如下接口完成： void *rt_realloc(void *rmem, rt_size_t newsize);

在进行重新分配内存块时，原来的内存块数据保持不变（缩小的情况下，后面的数据被自动截断）。

分配多内存块

从内存堆中分配连续内存地址的多个内存块，可以通过如下接口完成： void *rt_calloc(rt_size_t count, rt_size_t size);

返回的指针指向第一个内存块的地址，并且所有分配的内存块都被初始化成零。

释放内存块

用户线程使用完从内存分配器中申请的内存后，必须及时释放，否则会造成内存泄漏，释放接口如下： void rt_free (void *ptr);

用户线程需传递待释放的内存块指针，如果是空指针直接返回。

设置分配钩子函数

在分配内存块过程中，用户可申请一个钩子函数，它会在内存分配完成后回调，接口如下： void rt_malloc_sethook(void (*hook)(void *ptr, rt_size_t size));

回调时，会把分配到的内存块地址和大小做为入口参数传递进去。

设置内存释放钩子函数

在释放内存时，用户可设置一个钩子函数，它会在调用内存释放完成前进行回调，接口如下：

```
void rt_free_sethook(void (*hook)(void *ptr));
```

回调时，释放的内存块地址会做为入口参数传递进去（此时内存块并没有被释放）。

CHAPTER

EIGHT

异常与中断

异常是导致处理器脱离正常运行转向执行特殊代码的任何事件，如果系统不及时处理，系统轻则出错，重着导致系统毁灭性的瘫痪。所以正确地处理异常避免错误的发生是提高软件的鲁棒性重要的一方面，对于嵌入式系统更加如此。异常可以分成两类，同步异常和异步异常。同步异常主要是指由于内部事件产生的异常，例如除零错误。异步异常主要是指由于外部异常源产生的异常，例如按下设备某个按钮产生的事件。中断，通常也叫做外部中断。中断属于异步异常。当中断源产生中断时，处理器也将同样陷入到一个固定位置去执行指令。

8.1 中断处理过程

中断处理的一般过程如下图所示：

当中断产生时，处理机将按如下的顺序执行：

- 保存当前处理机状态信息
- 载入异常或中断处理函数到PC寄存器
- 把控制权转交给处理函数并开始执行
- 当处理函数执行完成时，恢复处理器状态信息
- 从异常或中断中返回到前一个程序执行点

中断使得CPU可以在事件发生时才予以处理，而不必让微处理器连续不断地查询是否有事件发生。通过两条特殊指令：关中断和开中断可以让处理器不响应或响应中断。而在执行中断服务例程时，大多数时候会屏蔽所有中断源，如果在执行中断服务例程过程中，有更高优先级别的中断源触发中断，很可能得不到正确的处理。而在硬实时环境中，这种情况是不允许发生的，关中断的时间应尽量的短。为了避免这种情况的发生，RT-Thread允许中断嵌套，即在一个中断服务例程期间，微处理器可以响应另外一个更重要的中断，过程如下图所示：

当正在执行一个中断服务例程（中断1）时，有更高的中断触发，将保存当前中断服务例程的上下文环境，转向中断2的中断服务例程。当所有中断服务例程都运行完成时，才又恢复上下文环境转回到中断1的中断服务例程中接着执行。即使如此，对于中断的处理仍然存在着（中断）时间响应的问题，先来看看中断处理过程中一个特定的时间量：

中断延迟TB定义为，从中断开始的时间到ISR程序开始执行的时间之间的时间段。而针对于处理时间TC，这主要取决于ISR程序如何处理，而不同的设备其相应的服务程序的时间需求也不一样。中断响应时间TD = TB + TC RT-Thread RTOS 提供系统栈，即中断发生时，中断的前期处理程序会将用户的堆栈指针更换为系统事先留出的空间中，等中断退出时再恢复用户的堆栈指针。这样中断将不再占任务的堆栈空间，大大提高内存空间的利用率，且随着任务的增加，这种技术的效果也越明显。

8.2 中断的Top/Bottom Half

RT-Thread RTOS不对ISR所需要的处理时间做任何限制，但如同其它RTOS或非RTOS一样，用户需要保证所有的中断服务例程在尽可能短的时间内完成。这样在发生中断嵌套，或屏蔽了相应中断源的过程中，不会耽误了嵌套的其它中断处理过程，或自己中断源的下一次中断信号。当一个中断信号发生时，ISR需要取得相应的硬件状态或者数据，如果ISR接下来要对状态或者数据进行简单处理，比如CPU时钟脉冲中断，ISR只需增加一个系统时钟tick，然后就结束ISR。这类中断往往所需的运行时间都比较短。对于另外一些中断，ISR在取得硬件状态或数据以后，还需要进行一系列更耗时的处理过程，通常需要将该中断分割为两部分，即Top Half和Bottom Half。在Top Half中，取得硬件状态和数据后，打开被屏蔽的中断（如果有的话），给相关的某个Thread发送一条通知（可以是RT-Thread所提供的任意一种IPC方式），然后结束ISR。而接下来，相关的Thread在接收到通知后，接着对状态或数据进行进一步的处理，这一过程称为Bottom Half。

8.2.1 Bottom Half实现范例

在这一节中，为了详细描述Bottom Half在RT-Thread中的实现，我们以一个虚拟的网络设备接收网络数据包作为范例，并假设接收到数据报文后，对报文的分析、处理是一个相对耗时，比外部中断源信号重要性小许多，而且在不屏蔽中断源信号后也能处理的过程。

在上面代码中，创建了demo_nw_thread，并将thread阻塞在nw_bh_sem上，一旦semaphore被释放，将执行接下来的nw_packet_parser，开始Bottom Half的事件处理。接下来让我们来看一下demo_nw_isr中是如何处理Top Half，并开启Bottom Half的。

由上面两个代码片段可以看出，通过一个IPC Object的等待和释放，来完成中断Bottom Half的起始和终结。由于将中断处理划分为Top和Bottom两个部分后，使得中断处理过程变为异步过程，这部分系统开销需要用户在使用RT-Thread RTOS时，必须认真考虑是否真正的处理时间大于给Bottom Half发送通知并处理的时间。

8.3 中断相关接口

RT-Thread RTOS为了把用户尽量的和系统底层异常、中断隔离开来，把中断和异常封装起来，提供给用户一个友好的接口。（注：这部分的API由BSP提供）

8.3.1 装载中断服务例程

通过调用rt_hw_interrupt_install，把用户的中断服务例程(new_handler)和指定的中断号关联起来，当这个中断源产生中断时，系统将自动调用装载的中断服务例程。如果old_handler不为空，则把之前关联的这个中断服务例程卸载掉。接口如下： void rt_hw_interrupt_install(int vector, rt_isr_handler_t new_handler, rt_isr_handler_t *old_handler)

8.3.2 屏蔽中断源

通常，在ISR准备处理某个中断信号之前，需要屏蔽该中断源，以保证在接下来的处理过程中硬件状态或者数据不会遭到干扰。接口如下： void rt_hw_interrupt_mask(int vector)

8.3.3 打开被屏蔽的中断源

在ISR处理完状态或数据以后，需要及时的打开之前被屏蔽的中断源，使得尽可能的不丢失硬件中断信号。接口如下： void rt_hw_interrupt_umask(int vector)

CHAPTER

NINE

定时器与系统时钟

6.1 定时器管理定时器是操作系统提供的一种系统操作，它能够在经过指定时间后依据用户的设置触发一定的动作，使一些需要时间控制的事件或者进程等得到可靠的控制。诸如任务控制块的时间片，实时的精度都需要相应的定时器模块的支持。在RT-Thread RTOS中，定时器模块以tick为时间单位。时间设备以一定的晶振频率（该频率可能是固定的或者可调节，要看硬件是否提供相应的支持，并且该频率与CPU的频率无关）提供一个规则的脉冲序列，若干个脉冲序列（通常称为初始计数值）完成后产生一次时钟中断。其中，脉冲序列的个数可以配置。tick的定义为发生一个时钟中断的时间长度。所以tick的时间长度 = 脉冲序列的个数 * 脉冲周期。在RT-Thread RTOS中维护着两个重要的全局变量，一个是当前系统的时间值rt_tick，另一个是定时器链表rt_timer_list，系统中新创建的定时期都会被排序插入到rt_timer_list链表中。

图6-1 定时器链表示意图a 如图6-1所示，系统当前tick值为20，在当前系统中已经创建并启动了三个定时器，分别为定时时间为50个tick的Timer1,100个tick的Timer2和500个tick的Timer3，这三个定时器分别被加上系统当前时间rt_tick = 20后从小到大排序插入到rt_timer_list链表中，形成图6-1所示的定时器链表结构，rt_tick一直在增长，50个tick以后，rt_tick从20增长到70,与Timer1的timeout值相等，这时会触发Timer1定时期相关连的超时函数，同时将Timer1从rt_timer_list链表上删除，同理，100个tick和500个tick过去后，Timer2和Timer3定时器相关联的超时函数会被触发，接着将Time2和Timer3定时器从rt_timer_list链表中删除。

如果系统当前定时器状态如图6-1所示，10个tick以后，rt_tick = 30, 此时有任务新创建一个tick值为300的Timer4定时器，则Timer4定时器的timeout = rt_tick + 300 =330, 然后被插入到Timer2和Timer3定时器中间，形成如图6-2所示链表结构。

图6-2 定时器链表示意图b 6.2 定时器管理控制块

```
struct rt_timer
{
        struct rt_object parent;
        rt_list_t list;                                 /* the node of timer list. */
        void (*timeout_func)(void* parameter);          /* timeout function. */
        void *parameter;                                /* timeout function's parameter. */
        rt_tick_t init_tick;                            /* timer timeout tick. */
        rt_tick_t timeout_tick;                         /* timeout tick. */
};
```

6.3 定时器管理接口 6.3.1 定时器管理系统初始化初始化定时器管理系统，可以通过如下接口完成： void rt_system_timer_init() 6.3.2 创建定时器当动态创建一个定时器时，内核首先创建一个定时器控制块，然后对该控制块进行基本的初始化，创建定时器使用以下接口： rt_timer_t rt_timer_create(const char* name, void (*timeout)(void* parameter), void* parameter, rt_tick_t time, rt_uint8_t flag) 使用该接口时，需要为定时器指定名称，提供定时器回调函数及参数，定时时间，并指定是单次定时还是循环周期定时。 6.3.3 删除定时器系统不再使用特定定时器时，通过删除该定时器以释放系统资源。删除定时器使用以下接口： rt_err_t rt_timer_delete(rt_timer_t timer) 6.3.4 初始化定时器当选择静态创建定时器时，可利用rt_timer_init接口来初始化该定时器，接口如下： void rt_timer_init(rt_timer_t timer, const char* name, void (*timeout)(void* parameter), void* parameter, rt_tick_t time, rt_uint8_t flag) 使用该接口时，需指定定时器对象，定时器

名称，提供定时器回调函数及参数，定时时间，并指定是单次定时还是循环周期定时。 6.3.5 脱离定时器脱离定时器使定时器对象被从系统容器的链表中脱离出来，但定时器对象所占有的内存不会被释放，脱离信号量使用以下接口。 rt_err_t rt_timer_detach(rt_timer_t timer) 6.3.6 启动定时器当定时器被创建或者初始化以后，不会被立即启动，必须在调用启动定时器接口后，才开始工作，启动定时器接口如下：rt_err_t rt_timer_start(rt_timer_t timer) 6.3.7 停止定时器启动定时器以后，若想使它停止，可以使用该接口：rt_err_t rt_timer_stop(rt_timer_t timer) 6.3.8 控制定时器控制定时器接口可以用来查看或改变定时器的设置，它提供四个命令接口，分别是设置定时时间，查看定时时间，设置单次触发，设置周期触发。命令如下：

```
#define RT_TIMER_CTRL_SET_TIME          0x0     /* set timer. */
#define RT_TIMER_CTRL_GET_TIME          0x1     /* get timer. */
#define RT_TIMER_CTRL_SET_ONESHOT       0x2     /* change timer to one shot. */
#define RT_TIMER_CTRL_SET_PERIODIC      0x3     /* change timer to periodic. */
```

控制定时器接口如下： rt_err_t rt_timer_control(rt_timer_t timer, rt_uint8_t cmd, void* arg) 使用该接口时，需指定定时器对象，控制命令及相应参数。

9.1 检查定时器

检查定时器接口是一个系统接口，当每一次系统时钟中断到来时，该接口就会被调用，它会检查该时刻是否有定时器到期，接口如下： void rt_timer_check()

CHAPTER

TEN

设备驱动

10.1 I/O设备管理

I/O管理模块为应用提供了一个对设备进行访问的通用接口，并通过定义的数据结构对设备驱动程序和设备信息进行管理。从位置来说I/O管理模块相当于设备驱动程序和内核之间的一个中间层。 I/O管理模块实现了对设备驱动程序的封装。设备驱动程序的实现与I/O管理模块独立，提高了模块的可移植性。应用程序通过I/O管理模块提供的标准接口调用设备驱动程序中的接口，设备驱动程序的升级不会对应用产生影响。此外，这种方式使与设备的硬件操作相关的代码与应用隔离，双方各自关注自己的功能，这降低了代码的复杂性，提高了系统的可靠性。在第四章中已经介绍过RT-Thread的内核对象管理器。读者若对这部分还不太了解，可以回顾一下这章。在RT-Thread中，设备也被认为是一类对象，被纳入对象管理器范畴。每个设备对象都是由基对象派生而来，每个具体设备都可以继承其父类对象的属性，并派生出其私有属性。图7-1即为设备对象的继承和派生关系示意图。

图7-1 设备对象的继承和派生关系示意图

10.2 I/O设备管理控制块

```
struct rt_device
{
        struct rt_object parent;
        enum rt_device_class_type type;                 /* device type */
        rt_uint8_t flag;                                /* device flag */
        rt_err_t (*init)        (rt_device_t dev);      /* common device interface */
        rt_err_t (*open)        (rt_device_t dev);
        rt_err_t (*close)       (rt_device_t dev);
        rt_err_t (*read)        (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size);
        rt_err_t (*write)       (rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size);
        rt_err_t (*control) (rt_device_t dev, rt_uint8_t cmd, void *args);
};
```

10.3 I/O设备管理接口

10.3.1 注册设备

将设备注册到设备对象管理器中，被注册的设备均可以通过“查找设备接口”来查找该设备，获得该设备控制块，可以通过如下接口完成： rt_err_t rt_device_register(rt_device_t dev, const char* name, rt_uint8_t flags)

10.3.2 卸载设备

将设备从设备对象管理器中卸载，被卸载的设备将不能通过“查找设备接口”找到该设备，可以通过如下接口完成： rt_err_t rt_device_unregister(rt_device_t dev)

10.3.3 初始化所有设备

初始化所有注册到设备对象管理器中的设备，可以通过如下接口完成： rt_err_t rt_device_init_all()

10.3.4 查找设备

根据指定的设备名称来查找设备，可以通过如下接口完成： rt_device_t rt_device_find(const char* name)

使用以上接口时，在设备对象类型所对应的对象容器中遍历寻找设备对象，然后返回该设备，如果没有找到这样的设备对象，则返回空。

10.3.5 打开设备

根据设备控制块来打开设备，可以通过如下接口完成： rt_err_t rt_device_open(rt_device_t dev)

10.3.6 关闭设备

根据设备控制块来关闭设备，可以通过如下接口完成： rt_err_t rt_device_close(rt_device_t dev)

10.3.7 读设备

根据设备控制块来读取设备，可以通过如下接口完成： rt_err_t rt_device_read (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)

10.3.8 写设备

根据设备控制块来写入设备，可以通过如下接口完成： rt_err_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)

10.3.9 控制设备

根据设备控制块来控制设备，可以通过如下接口完成： rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)

CHAPTER

ELEVEN

FINSH SHELL系统

RT-Thread的shell系统——finsh，提供了一套供用户在命令行操作的接口，主要用于调试、查看系统信息。finsh被设计成一个不同于传统命令行的解释器，由于很多嵌入式系统都是采用C语言来编写，所以finsh的输入对象也类似于C语言表达式的风格：它能够解析执行大部分C语言的表达式，也能够使用类似于C语言的函数调用方式访问系统中的函数及全局变量，此外它也能够通过命令行方式创建变量。

11.1 基本数据类型

finsh支持基本的C语言数据类型，包括： void – 空数据格式，只用于创建指针变量 char, unsigned char –（带符号）字符型数据 short, unsigned int – (带符号)整数型数据 long, unsigned long –(带符号)长整型数据 此外，finsh也支持指针类型（void *或int *等声明方式），如果指针做为函数指针类型调用，将自动按照函数方式执行。

11.2 内置命令

finsh中内建了一些命令函数，可以在命令行中调用：

```
list()
```

显示系统中存在的命令及变量，在lumit4510平台上执行结果如下

```
--Function List:
hello
version
list
list_thread
list_sem
list_mutex
list_event
list_mb
list_mq
list_memp
list_timer
--Variable List:
```

–Function List表示的是函数列表； –Variable List表示的是变量列表。

11.3 RT-Thread内置命令

针对RT-Thread RTOS，finsh也提供了一些基本的函数命令： list_thread() 列表显示当前系统中线程状态，lumit4510显示结果如下：

```
thread  pri  status      sp     stack size max used   left tick  error
------- ---- ------- ---------- ---------- ---------- ---------- ---
tidle    0xff ready   0x00000074 0x00000100 0x00000074 0x0000003b 000
tshell   0x14 ready   0x0000024c 0x00000800 0x00000418 0x00000064 000
```

thread字段表示线程的名称 pri字段表示线程的优先级 status字段表示线程当前的状态 sp字段表示线程当前的栈位置 stack size字段表示线程的栈大小 max used字段表示线程历史上使用的最大栈位置 left tick字段表示线程剩余的运行节拍数 error字段表示线程的错误号

list_sem() 列表显示系统中信号量状态，lumit4510显示结果如下： semaphore v suspend thread ——— —— ———— uart 000 0 semaphore字段表示信号量的名称 v字段表示信号量的当前值 suspend thread字段表示等在这个信号量上的线程数目

list_mb() 列表显示系统中信箱状态，lumit4510显示结果如下： mailbox entry size suspend thread ——— —- —- ———— mailbox字段表示信箱的名称 entry字段表示信箱中包含的信件数目 size字段表示信箱能够容纳的最大信件数目 suspend thread字段表示等在这个信箱上的线程数目

list_mq() 列表显示系统中消息队列状态，lumit4510显示结果如下： msgqueue entry suspend thread ——— —- ———— semaphore字段表示消息队列的名称 entry字段表示消息队列中当前包含的消息数目 size字段表示消息队列能够容纳的最大消息数目 suspend thread字段表示等在这个消息队列上的线程数目

list_event() 列表显示系统中事件状态，lumit4510显示结果如下： event set suspend thread ——— ———- ———— event字段表示事件的名称 set字段表示事件的值 suspend thread字段表示等在这个事件上的线程数目

list_timer() 列表显示系统中定时器状态，lumit4510显示结果如下： timer periodic timeout flag ——— ———- ——- ——— tidle 0x00000000 0x00000000 deactivated tshell 0x00000000 0x00000000 deactivated current tick:0x00000d7e timer字段表示定时器的名称 periodic字段表示定时器是否是周期性的 timeout字段表示定时器超时时的节拍数 flag字段表示定时器的状态，activated表示活动的，deactivated表示不活动的 current tick表示当前系统的节拍 12.4 应用程序接口 finsh的应用程序接口提供了上层注册函数或变量的接口，使用时应作如下头文件包含： #include <finsh.h>

原型：void finsh_syscall_append(const char* name, syscall_func func) 在finsh中添加一个函数。 name – 函数在finsh shell中访问的名称 func – 函数的地址

原型：void finsh_sysvar_append(const char* name, u_char type, void* addr) 在finsh中添加一个变量。 name – 变量在finsh shell中访问的名称 type – 数据类型，由枚举类型finsh_type给出finsh支持的数据类型：

```
enum finsh_type {
        finsh_type_unknown = 0,
        finsh_type_void,                /** void                */
        finsh_type_voidp,               /** void pointer        */
        finsh_type_char,                /** char                */
        finsh_type_uchar,               /** unsigned char       */
        finsh_type_charp,               /** char pointer        */
        finsh_type_short,               /** short               */
        finsh_type_ushort,              /** unsigned short      */
        finsh_type_shortp,              /** short pointer       */
        finsh_type_int,                 /** int                 */
        finsh_type_uint,                /** unsigned int        */
        finsh_type_intp,                /** int pointer         */
        finsh_type_long,                /** long                */
        finsh_type_ulong,               /** unsigned long       */
```

```
        finsh_type_longp                /** long pointer        */
};
```

addr – 变量的地址

12.5 移植由于finsh完全采用ANSI C编写，具备极好的移植性，同时在内存占用上也非常小，如果不使用上述提到的API函数，整个finsh将不会动态申请内存。

finsh shell线程：每次的命令执行都是在finsh shell线程的上下文中完成的，finsh shell线程在函数finsh_system_init()中创建，它将一直等待uart_sem信号量的释放。

finsh的输出： finsh的输出依赖于系统的输出，在RT-Thread中依赖的是rt_kprintf输出。

finsh的输入： finsh shell线程在获得了uart_sem信号量后调用rt_serial_getc()函数从串口中获得一个字符然后处理。所以finsh的移植需要rt_serial_getc()函数的实现。而uart_sem信号量的释放通过调用finsh_notify()函数以完成对finsh shell线程的输入通知。

通常的过程是，当串口接收中断发生时（即串口中有输入），接收中断服务例程调用finsh_notify()函数通知finsh shell线程有输入；而后finsh shell线程获取串口输入最后做相应的命令处理。 12.6 选项要开启finsh的支持，在RT-Thread的配置中必须定义RT_USING_FINSH宏。

CHAPTER

TWELVE

内核配置

RT-Thread内核是一个可配置的内核，可根据选项的不同以支持不同的特性。RT-Thread的内核是由rtconfig.h头文件控制，而此头文件又是由一些python脚本根据不同的配置动态产生。

每一种BSP的配置都是一个单独的文件，位于rtt-root /config文件夹下，例如lumit4510.py

```
PLATFORM = "lumit4510"
PREFIX = "arm-elf-"
BUILDTYPE = "RAM"
RELEASETYPE = "DEBUG"
TEXTBASE = "0x8000"
RT_USING_HOOK = "True"
RT_USING_EVENT = "True"
RT_USING_FASTEVENT = "False"
RT_USING_MESSAGEQUEUE = "True"
RT_USING_MEMPOOL = "True"
RT_USING_HEAP = "True"
RT_USING_FINSH = "True"
```

当需要生成此种配置的头文件及Makefile配置文件时，需要在此目录中执行

make_config.py lumit4510

（系统中必须安装有python执行环境）此文件主要分两部分，编译环境的设定及RT-Thread内核配置的设定。

12.1 编译环境配置

PLATFORM – 指定了生成哪个平台的配置文件，这个应该和rtt-root/bsp目录下的子目录一一对应。PREFIX – 指定gcc，binutil的前缀 BUILDTYPE – build类型，分为RAM及ROM两种 RELEASETYPE – 发布类型，分为DEBUG及RELEASE两种，DEBUG类型是带调试符号信息的内核，RELEASE是进行代码优化的内核； TEXTBASE – 正文段的基地址，由于在ld script中并没指定入口地址，所以在这里指定；

12.2 RT-Thread配置

RT_NAME_MAX – RT-Thread中对象的名称最大长度，默认 8 RT_ALIGN_SIZE – 对齐大小，默认 4 RT_THREAD_PRIORITY_MAX – 线程的最大优先级，默认 256 RT_TICK_PER_SECOND – 每秒的节拍数，默认 100 RT_DEBUG – 是否开启调试选项，默认 False RT_USING_HOOK – 是否启用钩子函数支持，默认 False RT_USING_SEMAPHORE – 是否支持信号量，默认 True RT_USING_MUTEX – 是否支持互斥锁，默认 True RT_USING_EVENT – 是否支持事件通信，默认 True RT_USING_FASTEVENT – 是否支持快速事

件通信，默认 True RT_USING_MAILBOX – 是否支持信箱通信，默认 True RT_USING_MESSAGEQUEUE – 是否支持消息队列，默认 True RT_USING_MEMPOOL – 是否支持内存池，默认 True RT_USING_HEAP – 是否支持动态堆内存分配，默认 True RT_USING_SMALL_MEM – 是否支持小内存动态分配算法，默认 False RT_USING_SLAB – 是否支持大内存SLAB动态分配算法，默认 True RT_USING_DEVICE – 是否启用设备管理系统，默认 True

CHAPTER

THIRTEEN

GNU GCC移植

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，其中的一个是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

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、lumit4510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，I2C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）。线程上线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 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

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

;/* ; * 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] ; 载入中断中切换标致地址 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_interrput_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 ldmfd 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_udef 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 = 0x3fffff;

/* 清除所有悬挂的中断 */ INTPEND = 0x1fffff;

/* 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_udef(struct rt_hw_register *regs) {

    rt_kprintf(“undefined instructionn”); 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 interruptn");
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 abortn");
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 abortn");
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 usedn");
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 requestn");

}
```

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lumit4510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 lighwight TCP/IP protocol stack / / Using lighweight 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_increase去触发一个系统节拍 */
        rt_tick_increase();
}

/**
 * lumit4510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lumit4510_ram.lds即可。代码 A - 15 lumit4510_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

CHAPTER

FOURTEEN

REALVIEW MDK移植

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）中添加上目录kernelinclude，kernellibcpuarmAT91SAM7S以及kernelbspsam7s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kernellibcpuAT91SAM7S目录中。

Kernel Group中添加所有kernelsrc下的C源文件； Startup Group中添加startup.c，board.c文件（放于kernelbspsam7s目录中）； AT91SAM7S Group中添加context_rvds.s，stack.c，trap.c，interrupt.c等文件（放于kernellibcpusam7s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_interrput_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_interrput_flag LDR r3, [r2] ; 载入中断中切换标致地址 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 0xFFFFFF00 ; EFC Base Address EFC0_FMR EQU 0x60 ; EFC0_FMR Offset EFC1_FMR EQU 0x70 ; EFC1_FMR Offset

;// <e> Embedded Flash Controller 0 (EFC0) ;// <o1.16..23> FMCN: Flash Microsecond Cycle Number <0-255> ;// <i> Number of Master Clock Cycles in 1us ;// <o1.8..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) ;// <o1.16..23> FMCN: Flash Microsecond Cycle Number <0-255> ;// <i> Number of Master Clock Cycles in 1us ;// <o1.8..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) ;// <o1.0..11> WDV: Watchdog Counter Value <0-4095> ;// <o1.16..27> WDD: Watchdog Delta Value <0-4095> ;// <o1.12> WDFIEN: Watchdog Fault Interrupt Enable ;// <o1.13> WDRSTEN: Watchdog Reset Enable ;// <o1.14> WDRPROC: Watchdog Reset Processor ;// <o1.28> WDDBGHLT: Watchdog Debug Halt ;// <o1.29> WDIDLEHLT: Watchdog Idle Halt ;// <o1.15> 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 ;// <o1.0> MOSCEN: Main Oscillator Enable ;// <o1.1> OSCBYPASS: Oscillator Bypass ;// <o1.8..15> OSCCOUNT: Main Oscillator Startup Time <0-255> ;// </h> ;// <h> Phase Locked Loop (PLL) ;// <o2.0..7> DIV: PLL Divider <0-255> ;// <o2.16..26> MUL: PLL Multiplier <0-2047> ;// <i> PLL Output is multiplied by MUL+1 ;// <o2.14..15> OUT: PLL Clock Frequency Range ;// <0=> 80..160MHz <1=> Reserved ;// <2=> 150..220MHz <3=> Reserved ;// <o2.8..13> PLLCOUNT: PLL Lock Counter <0-63> ;// <o2.28..29> USBDIV: USB Clock Divider ;// <0=> None <1=> 2 <2=> 4 <3=> Reserved ;// </h> ;// <o3.0..1> CSS: Clock Source Selection ;// <0=> Slow Clock ;// <1=> Main Clock ;// <2=> Reserved ;// <3=> PLL Clock ;// <o3.2..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_PRES 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 0xFFFFFF00 ; 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 subs pc, 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

```
/* 操作系统时钟节拍 (文件: board.c) */
#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);
}
```

CHAPTER

FIFTEEN

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。

15.1 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

        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

CHAPTER

SIXTEEN

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*