RT-Thread Programming Guide

Release 0.3.0

Bernard Xiong

June 29, 2009

CONTENTS

Contents:

CHAPTER

ONE

文档简介

RT-Thread做为国内有较大影响力的开源实时操作系统，本文是RT-Thread实时操作系统的编程指南文档，它旨在说明如何在RT-Thread实时操作系统上进行编程、把它使用到具体的应用中去。它分成几个部分分别讲述了:

- 实时系统概念：实时系统是一个什么样的系统，它的特点是什么；
- RT-Thread快速入门，在无硬件平台的情况下，如何迅速地了解RT-Thread实时操作系统，如何使用RT-Thread实时操作系统最基本的一些元素；
- RT-Thread作为一个完整的实时操作系统，它能够满足各种实时系统的需求，所以接下来详细地介绍了各个模块的结构，以及编程时的注意事项。
- RT-Thread外围组件的编程说明，RT-Thread不仅包括了一个强实时的内核，也包括外围的一些组件，例如shell，文件系统，协议栈等。这部分对外围组件编程进行了描述。
- RT-Thread中一些其他部分说明，包含了如何使用GNU GCC工具搭建RT-Thread的开发环境及RT-Thread在Cortex-M3系统上的说明。

本书面向使用RT-Thread系统的开发人员，并假定开发人员具备基本的C语言基础知识，如具备基本的实时操作系统知识将能更好的理解书中的一些概念。本书是一本使用RT-Thread进行编程的书籍，对于RT-Thread的内部实现并不做过多、过细节性的分析。

本书中异常与中断由王刚编写，定时器与系统时钟，I/O设备管理，文件系统由邱祎编写，其他部分由熊谱翔编写。

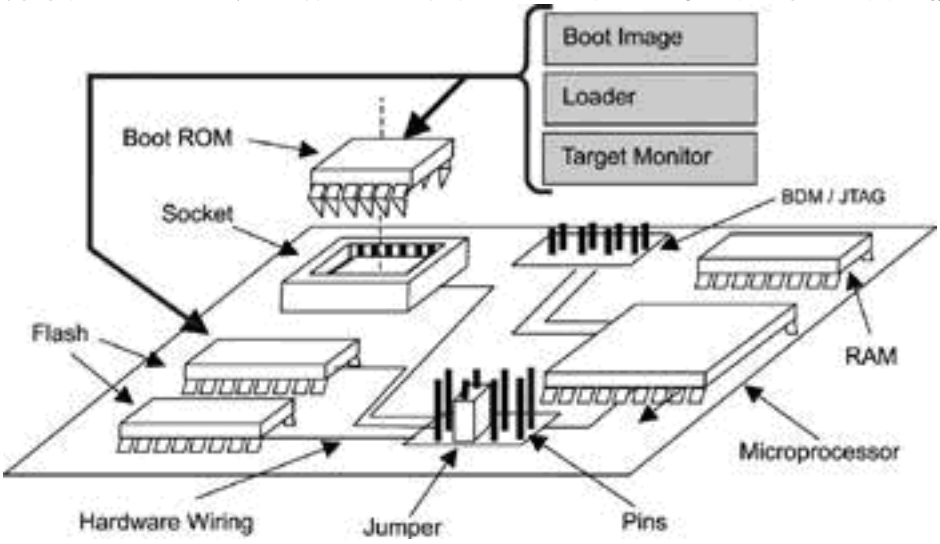
CHAPTER

TWO

实时系统

2.1 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。



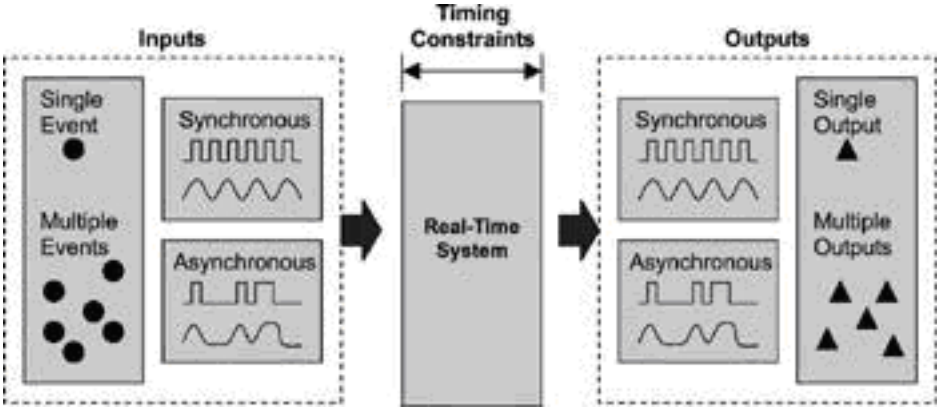
嵌入式系统

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。

2.2 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：



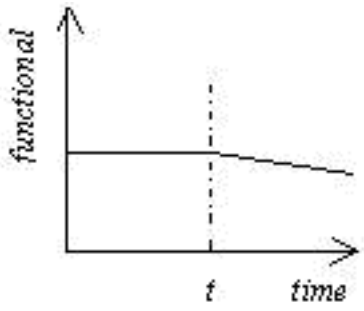
对于输入的信号、事件，实时系统应该能够在规定的时间内得到正确的响应，而不管这些事件是单一事件、多重事件还是同步信号或异步信号。对于一个具体的例子，可以考虑子弹射向玻璃杯的问题：一颗子弹从20米处射出，射向一个玻璃杯。假设子弹的速度是v米/秒，那么经过t1=20/v秒后，子弹将击碎玻璃杯。而有一系统在看见子弹射出后，将把玻璃杯拿走，假设这整个过程将持续t2秒的事件。如果$t2 < t1$，那么这个系统可以看成是一个实时系统。

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。

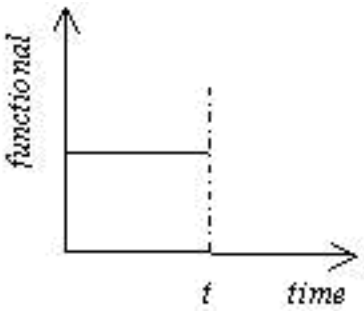
嵌入式系统的应用领域十分广泛，并不是其所针对的专用功能都要求实时性的，只有当系统中对任务有严格时间限定时，才有系统的实时性问题。具体的例子包括实验控制、过程控制设备、机器人、空中交通管制、远程通信、军事指挥与控制系统等。而对打印机这样一个嵌入式应用系统，人们并没有严格的时间限定，只有一个"尽可能快的"期望要求，因此，这样的系统称不上是实时系统。

2.3 软实时与硬实时

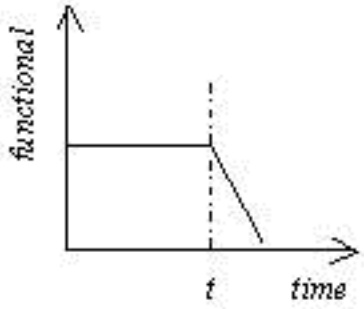
从上所述，实时系统是非常强调两点的：时间和功能的正确性。判断一个实时系统的正确性也正是这样，在给定的时间内正确地完成相应的任务。但也会有这种系统，偶尔也会在给定时间之外才能正确地完成任务，这种系统通常称为软实时系统。这也就构成了硬实时系统和软实时系统的区别。硬实时系统严格限定在给定的时间内完成任务，否则就可能导致灾难的发生，例如导弹的拦截，汽车引擎系统等。而软实时系统，可以允许一定的偏差，但是随着时间的偏移，整个系统的正确性也随之下降，例如一个DVD播放系统可以看成一个软实时系统，可以允许它偶尔的画面或声音延迟。



none real-time system



hard real-time system



soft real-time system

如上图所示，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

THREE

快速入门

一般嵌入式操作系统因为它的特殊性，往往和硬件平台密切相关连，具体的嵌入式操作系统往往只能在特定的硬件上运行。对于刚接触的读者并不容易马上就获得一个和RT-Thread相配套的硬件模块。在科技发展的今天，还有一种技术叫做仿真运行，下面我们将选择RealView MDK开发环境作为目标平台来看看RT-Thread是如何运行的。RealView MDK开发环境因为其完全的AT91SAM7S64软件仿真环境，让我们有机会在不使用真实硬件平台的情况下运行目标代码。这个软件仿真能够完整地虚拟出ARM7TDMI的各种运行模式，几乎和真实的硬件平台完全一致。实践也证明，这份软件仿真的RT-Thread代码能够在无任何修改的情况下在真实硬件平台中正常运行。

3.1 准备环境

在运行RT-Thread前，我们需要安装RealView MDK(正式版或评估版) 3.5+，它不仅是软件仿真工具，也是编译器链接器。这里采用了16k编译代码限制的评估版3.50版本。

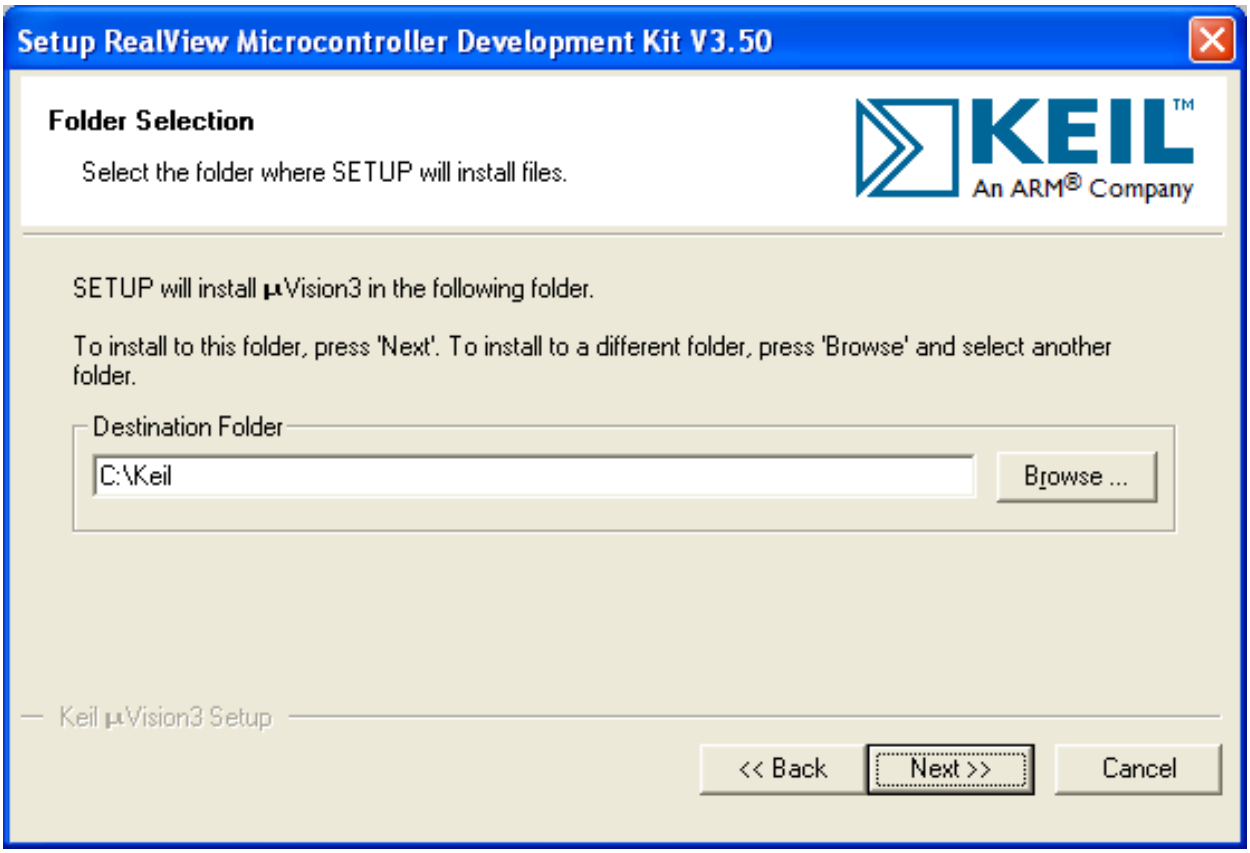
先从Keil官方网站下载 RealView MDK评估版 。

会需要先填一些个人基本信息，然后就可以下载了。

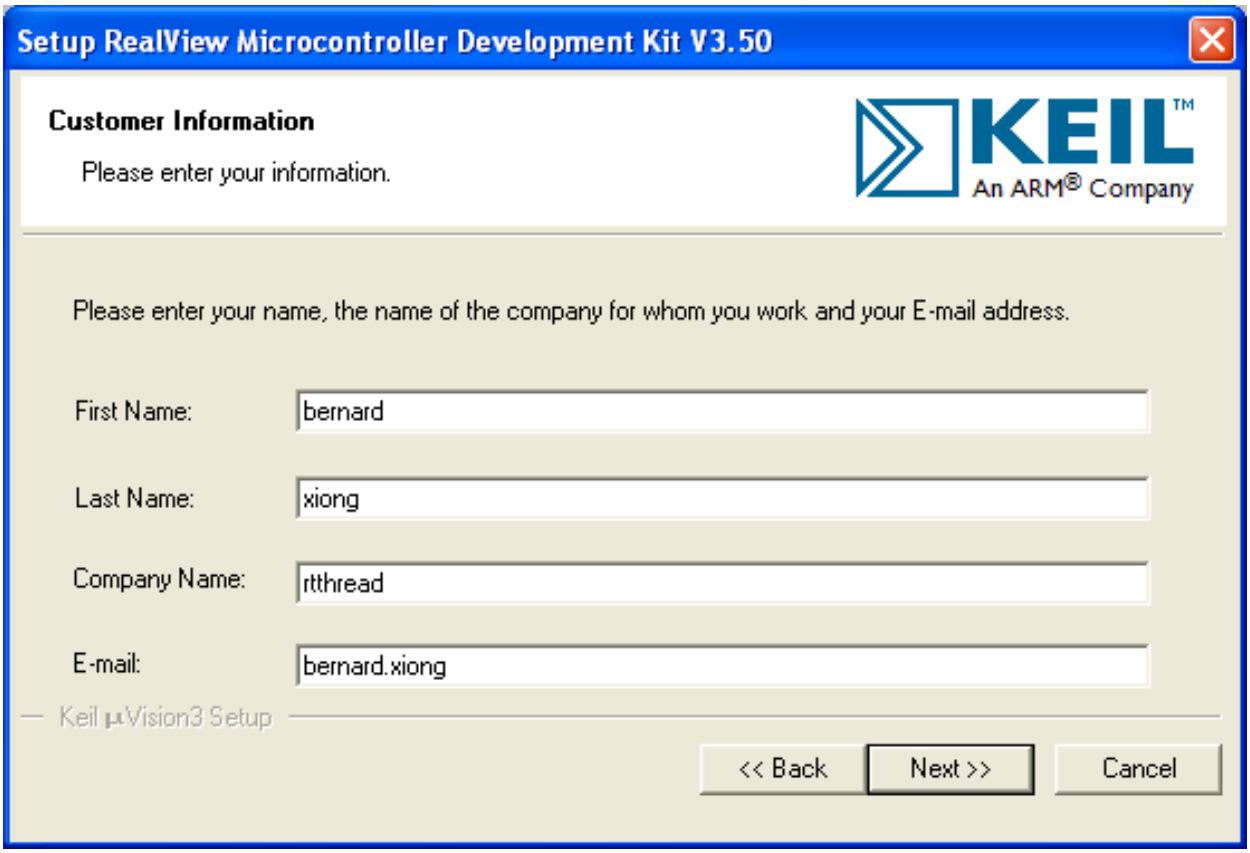
运行它，它会出现如下画面

这个是RealView MDK的安装说明，点”Next >>”进入下一画面

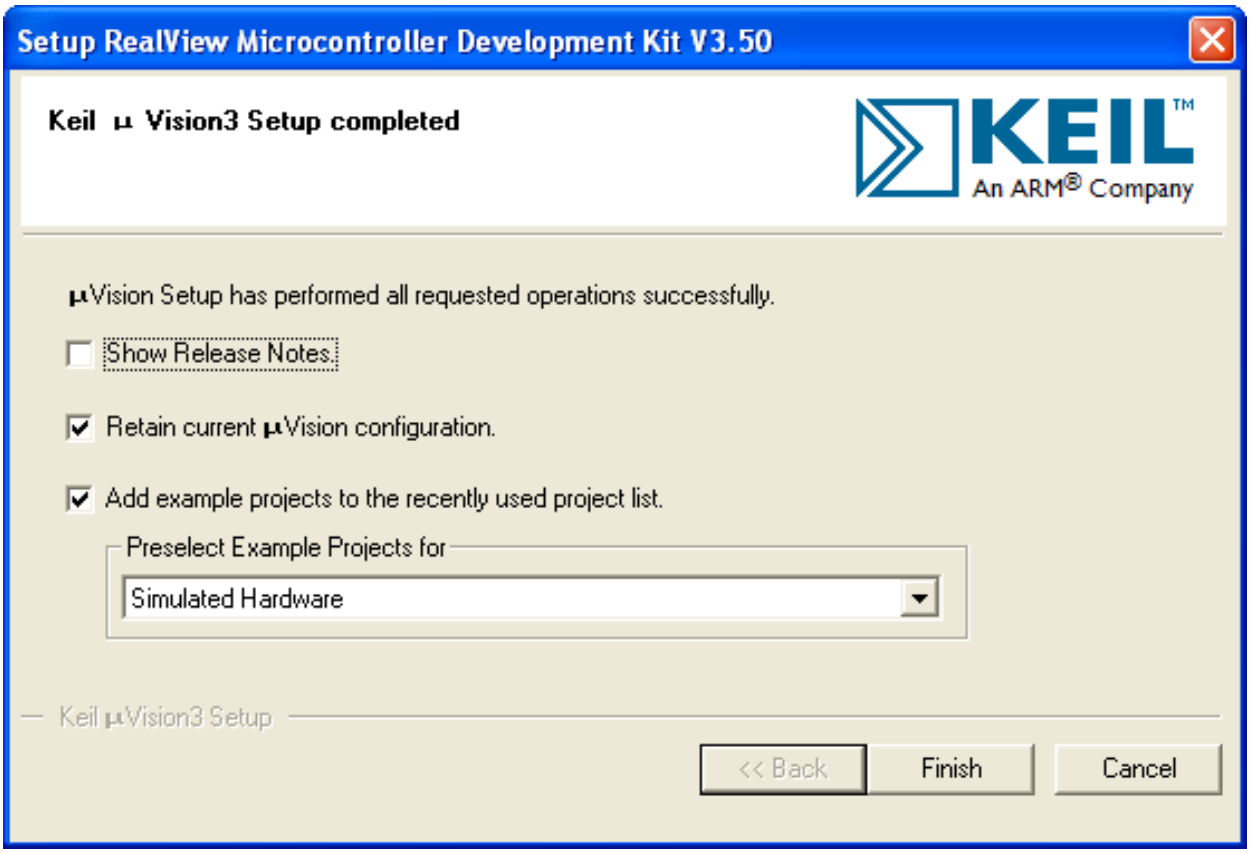
为了能够正常安装，需要同意它的条款(这是一款评估版)，选择"Next >>"



选择RealView MDK对于的安装目录，默认C:Keil即可，选择”Next >>”



输入您的名字及邮件地址，选择”Next >>”



安装完成，选择"Finish"

有了RealView MDK的利器，就可以轻松开始RT-Thread之旅，一起探索实时操作系统的奥秘。请注意RealView MDK正式版是收费的，如果您希望能够编译出更大体积的二进制文件，请购买RealView MDK正式版。RT-Thread也支持自由基金会的GNU GCC编译器，这是一款开源的编译器，想要了解如何使用GNU的相关工具搭建RT-Thread的开发环境请参考本书后面的附录部分，其中有在Windows/Linux环境下搭建采用GNU GCC做为RT-Thread开发环境的详细说明。

3.2 初识RT-Thread

RT-Thread做为一个操作系统，它会不会和Windows或Linux一样很庞大？代码会不会达到惊人的上百万行代码级别？弄清楚这些之前，我们先要做的就是获得本书 配套的RT-Thread代码 ，它可以看成是目前RT-Thread正在开发中的0.3.0主线版本中的一个分支，内核核心的代码与0.3.0正式版RT-Thread是完全一致的，只是把不必要的外围组件先行移除掉。

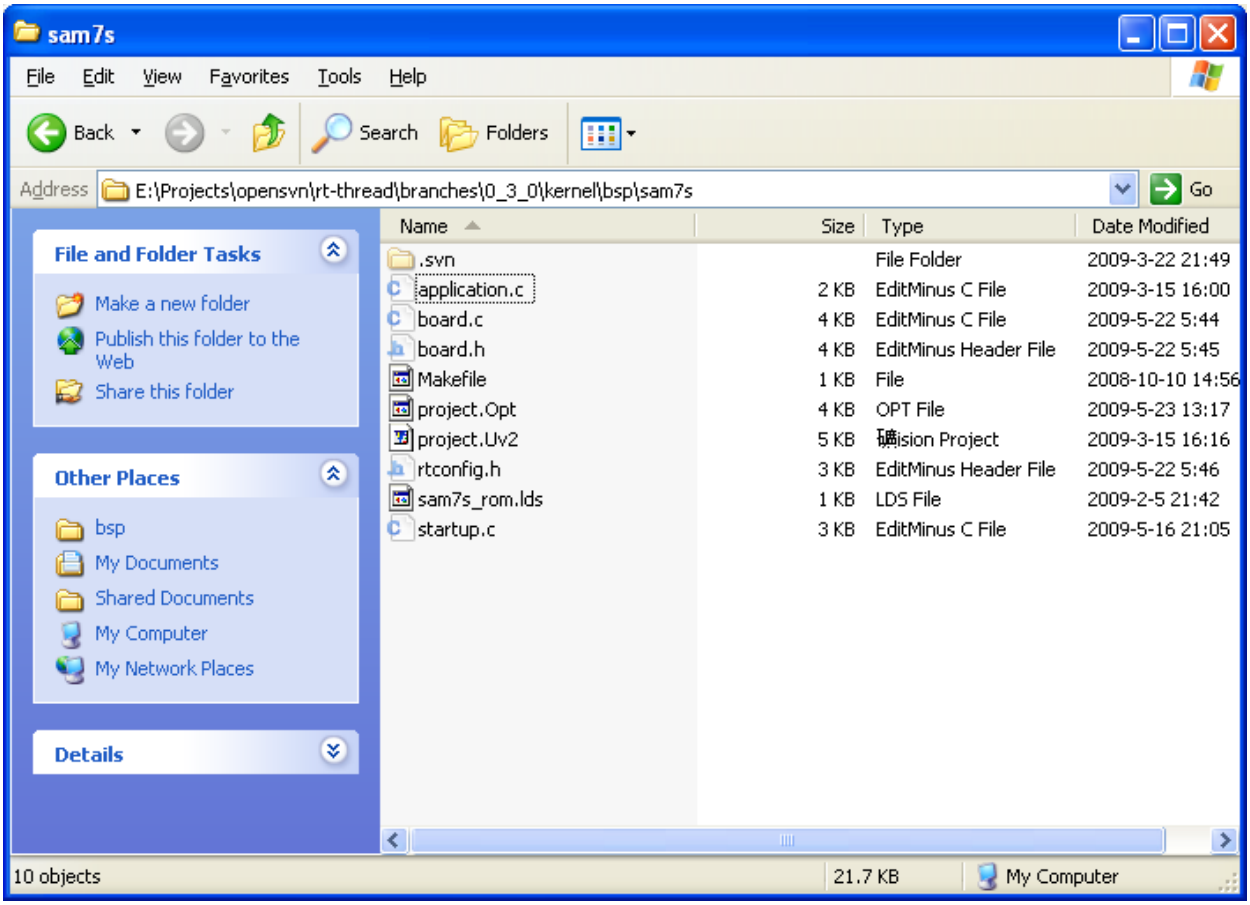
这个是一个压缩包文件，请解压到一个目录，这里把它解压到D:目录中。解压完成后的目录结构是这样的:

```
D:\rtthread-0.3.0
 -kernel
```

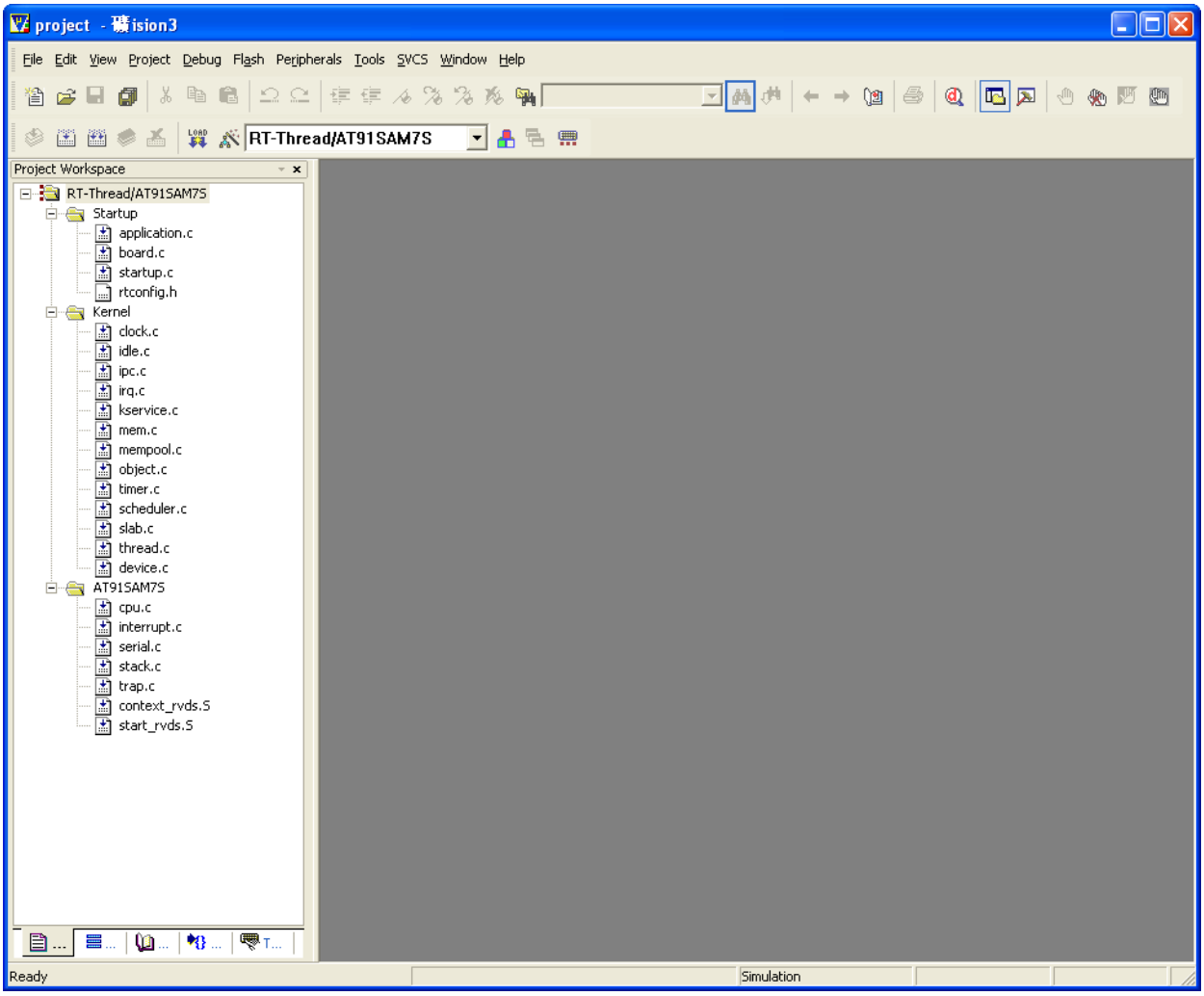
```
-src
-include
-finsh
-bsp
  -sam7s
-libcpu
    -arm
        -AT91SAM7S
```

kernel目录就是RT-Thread的源代码目录，有chapter前缀的则是和本书每个章节相配套的代码。

在目录\bsp\sam7s下，有一个project.Uv2文件



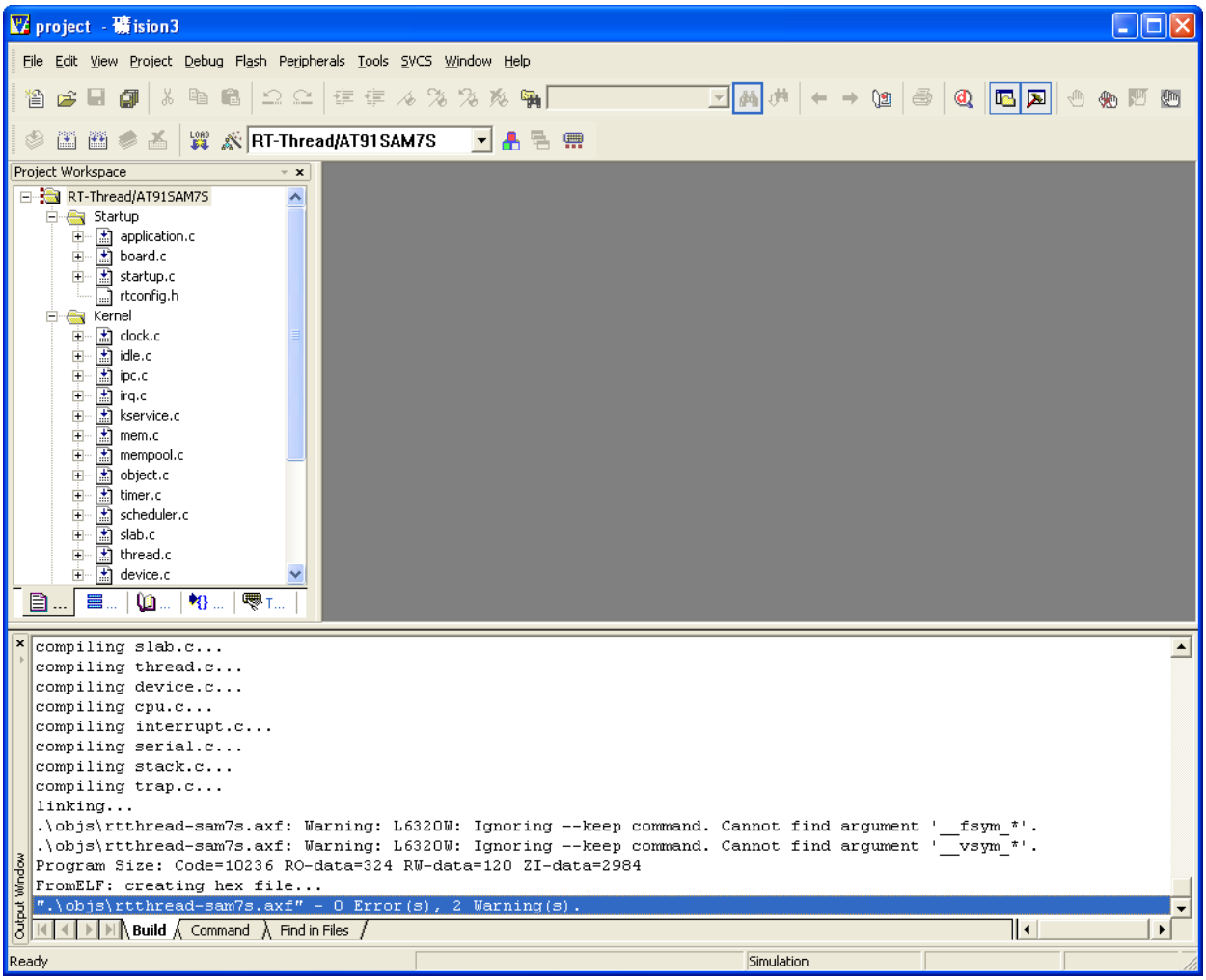
它是一个RealView MDK的工程文件，如果按照上节的步骤正确的安装了RealView MDK，那么在这里直接双击鼠标可以打开这个文件。打开后会出现如下的画面：



这个就是RT-Thread工程文件夹画面，在工程文件列表中总共存在如下几个组

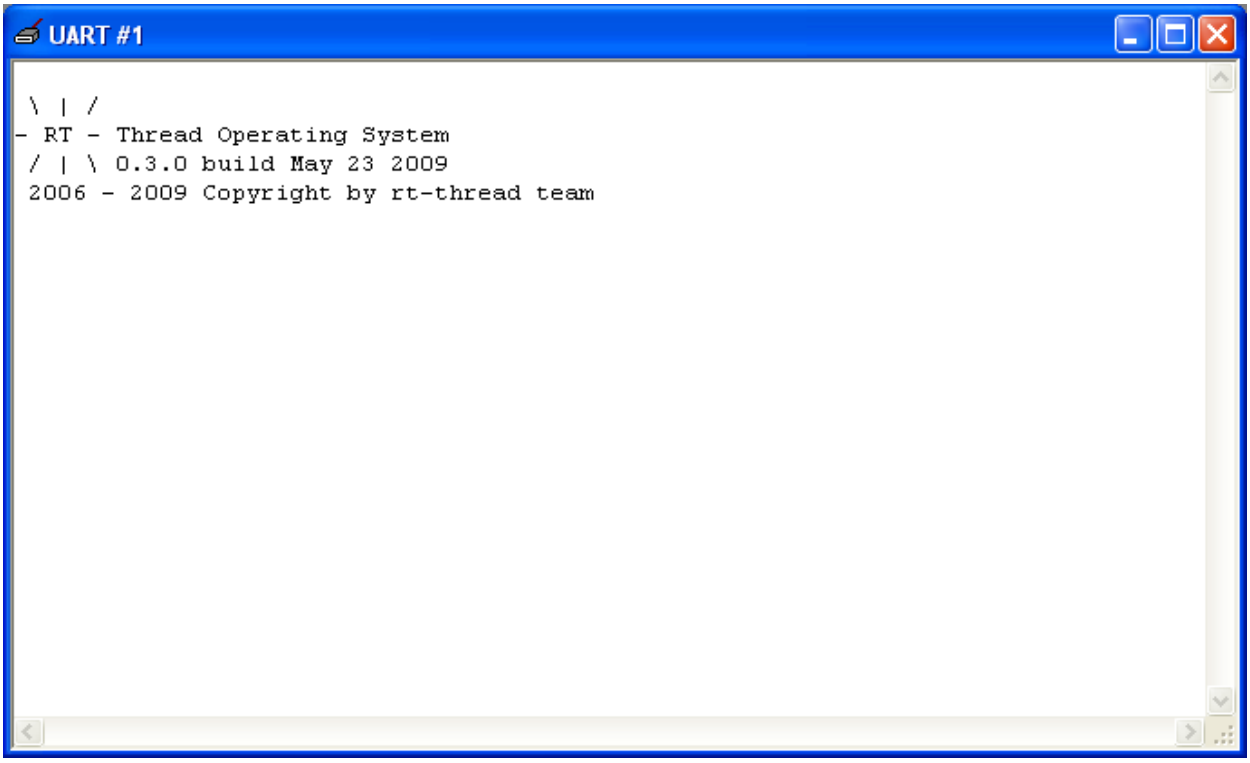
- Startup 用户开发板相关文件及启动文件（对应kernelbsp目录）
- Kernel RT-Thread内核核心实现（对应kernelsrc目录）
- AT91SAM7S 针对ATMEL AT91SAM7S64移植的代码（对应kernellibcpuAT91SAM7S目录）

我们先让RealView MDK编译运行试试：



没什么意外，最后会出现类似画面上的结果，虽然有一些警告但关系不大。

在编译完RT-Thread/AT91SAM7S后，我们可以通过RealView MDK的模拟器来仿真运行RT-Thread。模拟运行的结果如下图所示。



因为用户代码是空的，所以只是显示了RT-Thread的LOGO。

3.3 系统启动代码

一般了解一份代码大多从启动部分开始，同样这里也采用这种方式，先寻找启动的源头：因为RealView MDK的用户程序入口采用了main()函数，所以先看看main()函数在哪个文件中：startup.c，它位于Startup组中，是在AT91SAM7S64的启动汇编代码（启动汇编在AT91SAM7S Group的start_rvds.S中，在后面章节的移植一节中会详细讨论）后跳转到C代码的入口位置。

```
int main (void)
{
    /* 调用RT-Thread的启动函数，rtthread_startup */
    rtthread_startup();

    return 0;
}
```

很简单，main()函数仅仅调用了rtthread_startup()函数。RT-Thread因为支持多种平台，多种编译器，rtthread_startup()函数是RT-Thread的统一入口点。从rtthread_startup()函数中我们将可以看到RT-Thread的启动流程：

```
/* 这个函数将启动RT-Thread RTOS */
void rtthread_startup(void)
```

```
{
    /* 初始化硬件中断控制器 */
    rt_hw_interrupt_init();

    /* 初始化硬件开发板 */
    rt_hw_board_init();

    /* 显示RT-Thread的版本号 */
    rt_show_version();

    /* 初始化系统节拍，用于操作系统的时间技术 */
    rt_system_tick_init();

    /* 初始化系统对象 */
    rt_system_object_init();

    /* 初始系统定时器 */
    rt_system_timer_init();

    /*
     * 如果定义了宏RT_USING_HEAP，即RT-Thread使用动态堆
     * AT91SAM7S64的SRAM总共是16kB，地址范围是
     *   0x200000 - 0x204000
     * 所以在调用rt_system_heap_init函数时的最后一个参数是尾地址0x204000
     * 前面的初始地址则根据编译器环境的不同而略微不同
     */
#ifdef RT_USING_HEAP
#ifdef __CC_ARM
    rt_system_heap_init((void*)&Image$$RW_IRAM1$$ZI$$Limit, (void*)0x204000);
#elif __ICCARM__
    rt_system_heap_init(__segment_end("HEAP"), (void*)0x204000);
#else
    rt_system_heap_init((void*)&__bss_end, (void*)0x204000);
#endif
#endif

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

    /* 如果使用了钩子函数，把rt_hw_led_flash函数挂到idle线程的执行中去 */
#ifdef RT_USING_HOOK
    /* set idle thread hook */
    rt_thread_idle_sethook(rt_hw_led_flash);
#endif

    /* 如果使用了设备框架 */
#ifdef RT_USING_DEVICE
```

```
    /* 注册/初始化硬件串口 */
    rt_hw_serial_init();
    /* 初始化所有注册了的设备 */
    rt_device_init_all();
#endif

    /* 初始化上层应用 */
    rt_application_init();

    /* 如果系统中使用了shell系统 */
#ifdef RT_USING_FINSH
    /* 初始化finsh */
    finsh_system_init();
    /* finsh的输入设备是uart1设备 */
    finsh_set_device("uart1");
#endif

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

    /* 启动调度器，将进行系统的第一次调度 */
    rt_system_scheduler_start();

    /* 这个地方应该是永远都不应该达到的 */
    return ;
}
```

这部分启动代码，可以分为几个部分：

- 初始化系统相关的硬件
- 初始化系统组件，例如定时器，调度器
- 初始化系统设备，这个主要是为RT-Thread的设备框架做的初始化
- 初始化各个应用线程，并启动调度器

3.4 用户入口代码

上面的启动代码基本上可以说都是和RT-Thread系统相关的，那么用户代码在什么地方初始化？

```
/* 初始化上层应用 */
rt_application_init();
```

这里，用户代码入口位置是rt_application_init()，在这个函数中可以初始化用户应用程序的线程，当后面打开调度器后，用户线程也将得到执行。

在工程中，rt_application_init()的实现在application.c文件中，目前跑的RT-Thread是最简单的，仅包含一个空的rt_application_init()实现：

```
/* 包含RT-Thread的头文件，每一个需要用到RT-Thread服务的文件都需要包含这个文件 */
#include <rtthread.h>

/* 用户应用程序入口点 */
int rt_application_init()
{
    return 0;
}
```

空的实现就意味着，系统中不存在用户的代码，系统只会运行和系统相关的一些代码。在以后的例子中，如果没有特殊的说明，我们都将在这个文件中实现代码，并在rt_application_init()函数中进行初始化工作。

3.5 跑马灯的例子

对于从事电子方面开发的技术工程师来说，跑马灯大概是最简单的例子，就类似于每种编程语言中的Hello World。所以第一个例子就从跑马灯例子开始：创建一个线程，让它不定时的对LED进行更新（关或灭）

```
/* 因为要使用RT-Thread的线程服务，需要包含RT-Thread的头文件 */
#include <rtthread.h>

/* 线程用到的栈，由于ARM是4字节对齐的，所以栈的空间必须是4字节对齐 */
static char led_thread_stack[512];

/* 线程的TCB控制块 */
static struct rt_thread led_thread;

/* 线程的入口点，当线程运行起来后，它将从这里开始执行 */
static void led_thread_entry(void* parameter)
{
    int i;

    /* 这个线程是一个永远循环执行的线程 */
    while (1)
    {
        /* 开LED，然后延时10个tick */
        led_on();
        rt_thread_delay(10);
```

```
        /* 关LED，然后延时10个tick */
        led_off();
        rt_thread_delay(10);
    }
}

/* 用户应用程序入口点 */
int rt_application_init()
{
    /*
     * 初始化一个线程
     * 名称是`led`
     * 入口位置是led_thread_entry
     * 入口参数是空，这个参数会传递给入口函数的，可以是一个指针或一个数
     * 优先级是25（AT91SAM7S64配置的最大优先级数是32，这里使用25）
     * 时间片是8（如果有相同优先级的线程存在，时间片才会真正起作用）
     */
    rt_thread_init(&led_thread,
        "led",
        led_thread_entry, RT_NULL,
        &led_thread_stack[0], sizeof(led_thread_stack),
        25, 8);

    /*
     * 上一步仅仅是初始化一个线程，也就是为一个线程的运行做准备，
     * 这里则是启动这个线程
     *
     * 注：这个函数并不代表线程立刻就运行起来，当调度器启动起来后，
     * 线程才得到真正的调度。如果此时，调度器已经运行了，那么则取决于新启
     * 动的线程优先级是否高于当前任务优先级，如果高于，则立刻执行新线程。
     */
    rt_thread_startup(&led_thread);
    return 0;
}
```

在代码中rt_thread_delay(10)函数的作用是延时一段时间，即让led线程休眠10个tick（按照rtconfig.h中的配置，1秒 = RT_TICK_PER_SECOND个tick = 100tick，即这份代码中是延时100ms）。在休眠的这段时间内，如果没有其他线程运行，操作系统会切换到idle线程运行。

3.6 生产者消费者问题

生产者消费者问题是操作系统中的一个经典问题，在嵌入式操作系统中也经常能够遇到，例如串口中接收到数据，然后由一个任务统一的进行数据的处理：串口产生数据，任务作

为一个消费者消费数据。

在下面的例子中，将用RT-Thread的编程模式来实现一个生产者、消费者问题的解决代码。

```
#include <rtthread.h>

/* 定义最大5个元素能够被产生 */
#define MAXSEM 5

/* 用于放置生产的整数数组 */
rt_uint32_t array[MAXSEM];

struct rt_semaphore sem_lock;
struct rt_semaphore sem_empty, sem_full;
/* 指向生产者、消费者在array数组中的读写位置 */
static rt_uint32_t set, get;

/* 生成者线程入口 */
void producer_thread_entry(void* parameter)
{
    int cnt = 0;

    /* 运行100次 */
    while( cnt < 100)
    {
        /* 获取一个空位 */
        rt_sem_take(&sem_empty, RT_WAITING_FOREVER);

        /* 修改array内容，上锁 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        array[set%MAXSEM] = cnt + 1;
        rt_kprintf("the producer generates a number: %d\n", array[set%MAXSEM]);
        set++;
        rt_sem_release(&sem_lock);

        /* 发布一个满位 */
        rt_sem_release(&sem_full);
        cnt++;

        /* 暂停一段时间 */
        rt_thread_delay(50);
    }

    rt_kprintf("the producer exit!\n");
}

/* 消费者线程入口 */
```

```
void consumer_thread_entry(void* parameter)
{
    rt_uint32_t no;
    rt_uint32_t sum;

    /* 第n个线程，由入口参数传进来 */
    no = (rt_uint32_t)parameter;

    while(1)
    {
        /* 获取一个满位 */
        rt_sem_take(&sem_full, RT_WAITING_FOREVER);

        /* 临界区，上锁进行操作 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        sum += array[get%MAXSEM];
        rt_kprintf("the consumer[%d] get a number: %d\n", no, array[get%MAXSEM] );
        get++;
        rt_sem_release(&sem_lock);

        /* 释放一个空位 */
        rt_sem_release(&sem_empty);

        /* 生产者生产到100个数目，停止，消费者线程相应停止 */
        if (get == 100) break;

        /* 暂停一小会时间 */
        rt_thread_delay(10);
    }

    rt_kprintf("the consumer[%d] sum is %d \n ", no, sum);
    rt_kprintf("the consumer[%d] exit!\n");
}

/**
 * This function will be invoked to initalize user application when system startup.
 */
int rt_application_init()
{
    rt_thread_t p, s;

    /* 初始3个信号量 */
    rt_sem_init(&sem_lock , "lock",     1,      RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_empty, "empty",    MAXSEM, RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_full , "full",     0,      RT_IPC_FLAG_FIFO);

    /* 创建 生产者 线程 */
```

```
    p = rt_thread_create("p",
        producer_thread_entry, RT_NULL,
        1024, 18, 5);
    rt_thread_startup(p);

    /* 创建 消费者 线程，入口相同，入口参数不同，优先级相同 */
    s = rt_thread_create("s1",
        consumer_thread_entry, (void *)1,
        1024, 20, 5);
    if (s != RT_NULL) rt_thread_startup(s);

    s = rt_thread_create("s2",
        consumer_thread_entry, (void *)2,
        1024, 20, 5);
    if (s != RT_NULL) rt_thread_startup(s);

    return 0;
}
```

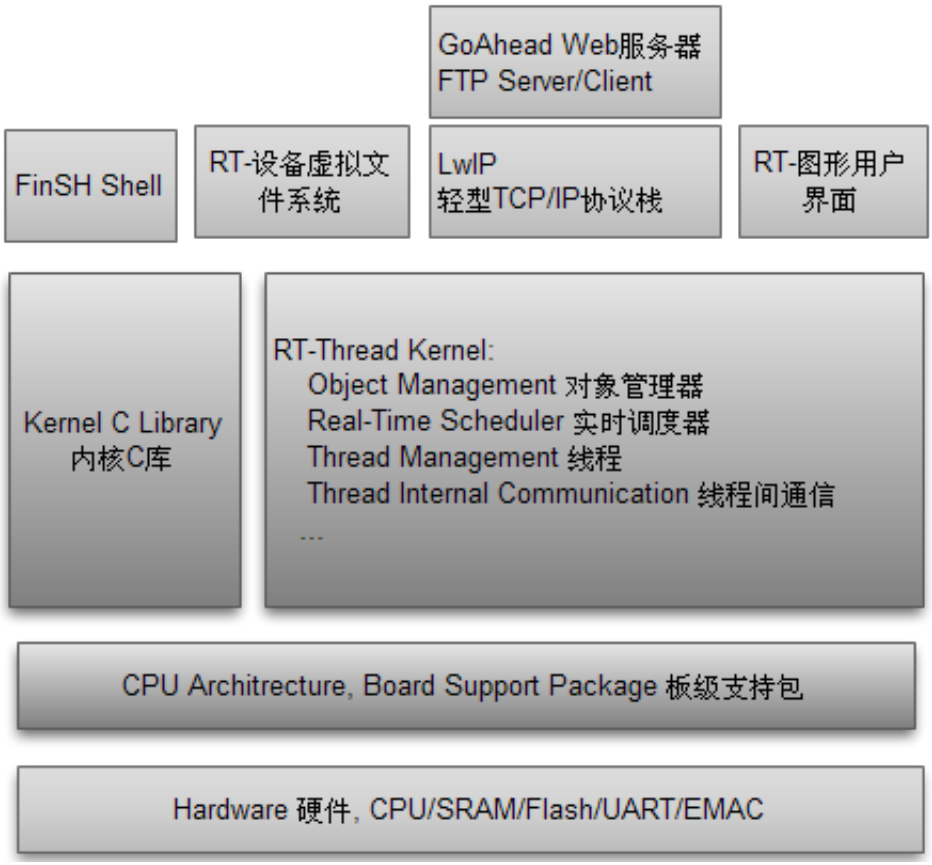
在上面的例子中，系统启动了一个生产者线程p，用于向仓库（array数组）中产生一个整数（1 到 100）；启动了两个消费者线程s1和s2，它们的入口函数是同一个，只是通过入口参数分辨它们是第一个消费者线程还是第二个消费者线程。这两个消费者线程将同时从仓库中获取生成的整数，然后把它打印出来。

CHAPTER

FOUR

RT-THREAD简介

RT-Thread是一个源代码公开的实时操作系统，并且商业许可证非常宽松的实时操作系统。下图是RT-Thread及外围组件的基本框架图：



RT-Thread Kernel内核部分包括了RT-Thread的核心代码，包括对象管理器，线程管理及调度，线程间通信等的微小内核实现。内核C库是为了保证内核能够独立运作的一套小型C库（在RealView MDK等系统自带部分C库函数的情况下，这部分不会被使能）。CPU及板级支持包包含了RT-Thread支持的各个平台移植代码，通常会包含两个汇编文件，一个是系统启动初始化文件，一个是线程进行上下文切换的文件，其他的都是C源文件。

4.1 实时内核

4.1.1 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（也能通过选项选择最大支持32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(时间复杂度是1，即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。

4.1.2 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。

4.1.3 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，效率较消息队列更为高效。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。

4.1.4 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；定时器支持一次性超时及周期性超时。

4.1.5 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，当内存池为空时，可把申请内存块的线程阻塞(或立刻返回，或等待一段时间后仍未获得内存块返回。这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，将把相应阻塞线程唤醒。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。

4.1.6 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，当设备事件触发时，通知给上层的应用程序。

4.2 虚拟文件系统

RT-Thread提供的文件系统称为设备文件系统，它主要包含了一个非常轻型的虚拟文件系统。虚拟文件系统的好处就是，不管下层采用了什么文件系统，例如内存虚拟文件系统，FAT32文件系统还是YAFFS2闪存文件系统，对上层应用程序提供的接口都是统一的。

4.3 轻型IP协议栈

LwIP 是瑞士计算机科学院（Swedish Institute of Computer Science）的Adam Dunkels等开发的一套用于嵌入式系统的开放源代码TCP/IP协议栈，它在包含完整的TCP协议实现基础上实现了小型的资源占用，因此它十分适合于使用到嵌入式设备中，RT-Thread 采用 LwIP 做为默认的 TCP/IP 协议栈，同时根据小型设备的特点对其进行再优化，体积相对进一步减小，**RAM 占用缩小到5kB附近**（依据上层应用使用情况会有浮动）。

4.4 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。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。

4.5 支持的平台

- ARM7TDMI，如ATMTEL的AT91SAM7系列处理器，NXP的LPC2xxx系列处理器，SAMSUNG的44B0处理器
- ARM920T，如SAMSUNG的 S3C2410/S3C2440 系列处理器，ATMEL 的 AT91RM9200 处理器
- Cortex-M3，如 ST 的 STM32F103/F105/F107 系列处理器
- x86
- ColdFire

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语言一般都称为一种面向过程的语言。面向对象源于人类对世界的认知大多偏向于类别模式，根据世界中不同物品的特性分门类别的组织在一起学习，归纳。在计算机领域一般实现面向对象的方式是依赖于一门新的，具备面向对象特征的语言，例如常见的编程语言C++，Java，Python等。那么RT-Thread既然有意引入对象系统，为什么不直接采用C++来实现？这个需要从C++的实现说起，用过C++的开发人员都知道，C++的对象系统中会引入很多未知的东西，例如虚拟重载表，命名粉碎，模板展开等。对于一个需要精确控制的系统，这不是一个很好的方式，假于它人之手不如握入己手！

面向对象有它非常优越的地方，取其精华(即面向对象思想，面向对象设计)，也就是RT-Thread内核对象模型的来源。RT-Thread中包含一个小型的，非常紧凑的对象系统，这个对象系统完全采用C语言实现。在了解RT-Thread内部或采用RT-Thread编程时有必要先熟悉它，它是RT-Thread实现的基础。

5.1 C语言的对象化模型

面向对象的特征主要包括：

- 封装，隐藏内部实现
- 继承，复用现有代码
- 多态，改写对象行为

采用C语言实现的关键是如何运用C语言本身的特性来实现上述面向对象的特征。

5.1.1 封装

一般属于某个类的对象会有一个统一的创建，析构过程。在RT-Thread中这些分为两类(以semaphore对象为例)：

- 对象内存已经创建，需要对它进行初始化 – rt_sem_init;

- 对象内存数据快未分配，需要从头创建 – rt_sem_create。

可以这么认为，对象的创建(create)是以对象的初始化(init)为基础的，创建动作相比较而言多了个内存分配的动作。

相对应的两类析构方式：

- 由rt_sem_init初始化的semaphore对象 – rt_sem_detach;
- 由rt_sem_create创建的semaphore对象 – rt_sem_delete.

在C语言中，大多数函数的命名方式是动词+名词的形式，例如要获取一个semaphore，会命名成take_semaphore，重点在take这个动作上。在RT-Thread的面向对象编程中刚好相反，命名为rt_sem_take，即名词+动词的形式，重点在名词上，体现了一个对象的方法。另外对于某些方法，仅仅是在对象内部使用，它们将采用static修辞把作用范围局限在一个文件的内部：

```
static rt_err_t _rt_sem_init()
{
    ...
}
```

通过方法的命名规则和static修辞，把对象的方法进行一定的封装，形成了面向对象中最基本的对象封装实现。

5.1.2 继承

我们知道C语言中的指针是非常强大的，不仅可以指向变量而且还可以泛指向任何数据，例如不同类型的变量，函数等。不同变量间的指针只需要强制转换类型就可实现不同类型数据的访问。例如如下代码：

```
struct parent_class
{
    int a, b;
    char *str;
};

struct child_class
{
    struct parent_class p;
    int a, b;
};

void func()
{
    struct child_class obj, *obj_ptr;
```

```
    struct parent_class *parent_ptr;

    obj_ptr = &obj;
    /* 获得父指针 */
    parent_ptr = (struct parent*) &obj;

    parent_ptr->a = 1;
    parent_ptr->b = 5;

    obj_ptr->a = 10;
    obj_ptr->b = 100;
}
```

在上面代码中，注意subobject结构中第一个成员p，这种声明方式代表subobject类型的数据中开始的位置包含一个parent类型的变量。在函数func中obj是一个subobject对象，正向这个结构类型指示的，它前面的数据应该包含一个parent类型的数据。在第行的强制类型赋值中parent_ptr指向了obj变量的首地址，实际上也就是obj变量中的p对象。好了，现在parent_ptr指向的是一个真真实实的parent类型的结构，那么可以按照parent的方式访问其中的成员，当然也包括可以使用和parent结构相关的函数来处理内部数据，因为一个正常的，正确的代码，它是不会越界访问parent结构体以外的数据的。

经过这基本的结构体层层相套包含，对象简单的继存关系就体现出来了：父对象放于数据块的最前方，代码中可以通过强制类型转换获的父对象指针。

5.1.3 多态

多态是面向对象设计中极为重用的一环。对象系统是一个归类的系统，类与类之间是有联系的，例如RT-Thread中的设备和串口设备，串口是设备的一种。设备一般支持读写接口，但是统一的，串口是设备的一种，也应该支持设备的读写。但串口的读写操作是串口所特有的，不应和其他设备操作完全相同，例如操作串口的操作不应应用于SD卡设备中。

RT-Thread对象模型采用结构封装中使用指针的形式达到面向对象中多态的效果，例如：

```
struct base_class
{
    int a;
    void (*vfunc)(int a);
}

void base_class_vfunc(struct base_class *self, int a)
{
    assert(self != NULL);
    assert(slef->vfunc != NULL);

    self->vfunc(a);
}
```

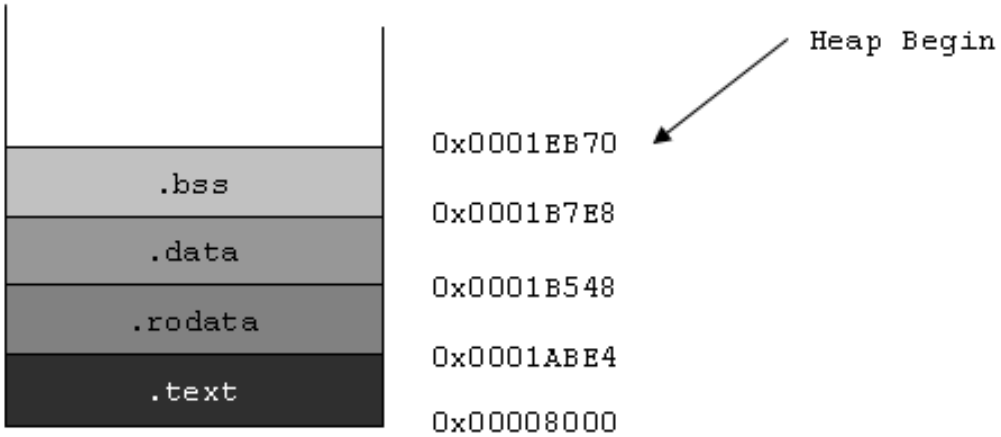
```
struct child_class
{
    struct base_class parent;
    int b;
};

void child_class_init(struct child_class* self)
{
    struct base_class* parent;
    parent = (struct base_class*) self;

    assert(parent != NULL);

    parent->vfunc = child_class_vfunc;
}

static void _child_class_vfunc(struct child_class*self, int a)
{
    self->b = a + 10;
}
```

5.2 内核对象模型

5.2.1 静态对象和动态对象

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AT91SAM7S64为例）：
其中主要包括了这么几段：

Segment	Description
.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



上图为AT91SAM7S64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，如下代码：

```
rt_uint8_t* msg_ptr;
msg_ptr = (rt_uint8_t*) rt_malloc (128);
rt_memset(msg_ptr, 0, 128);
```

msg_ptr 指向的128字节内存空间就是位于堆空间中的。

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），如下代码：

```
#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。

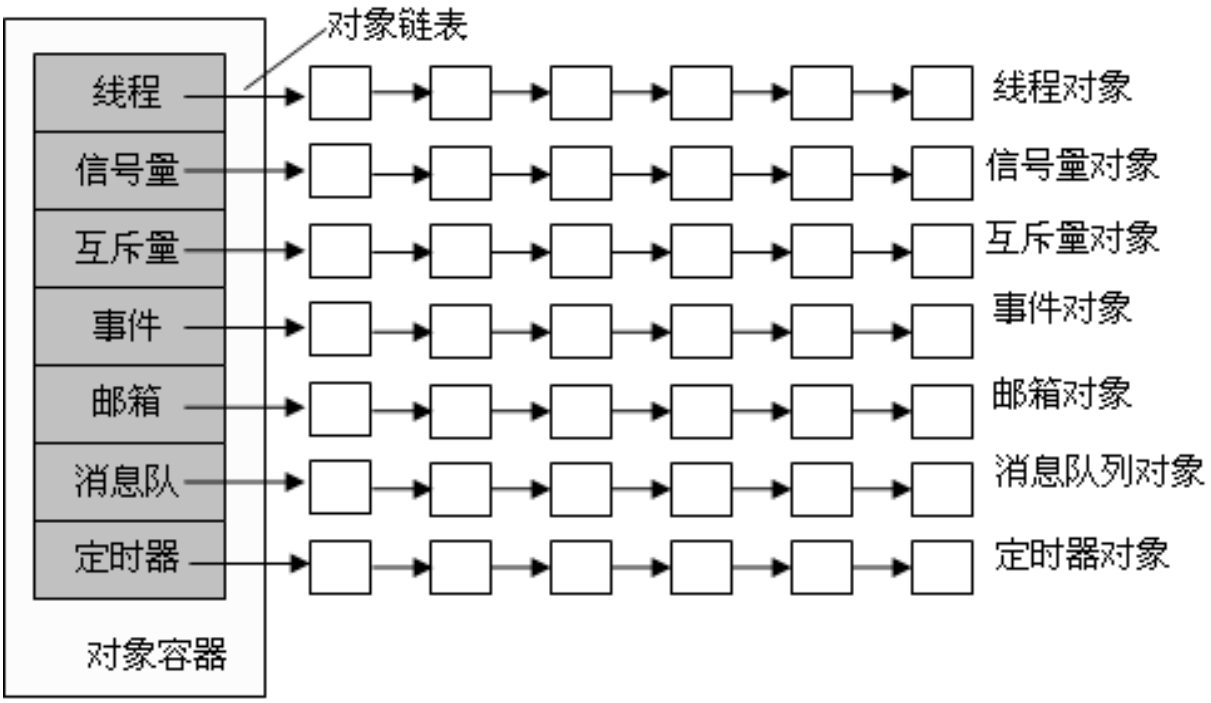
RT-Thread中操作系统级的设施都是一种内核对象，例如线程，信号量，互斥量，定时器等。以下的代码所示的即为静态线程和动态线程的例子：

```
/* 线程1的对象和运行时用到的栈 */
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);
    rt_thread_startup(&thread1);

    thread2_ptr = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
```

```
        512, 250, 25);
    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，包括线程控制块 thread1，栈空间thread1_stack都是编译时决定的，因为代码中都不存在初始值，都统一放在.bss段中。thread2运行中用到的空间都是动态分配的，包括线程控制块（thread2_ptr指向的内容）和栈空间。

5.2.2 内核对象管理工作模式

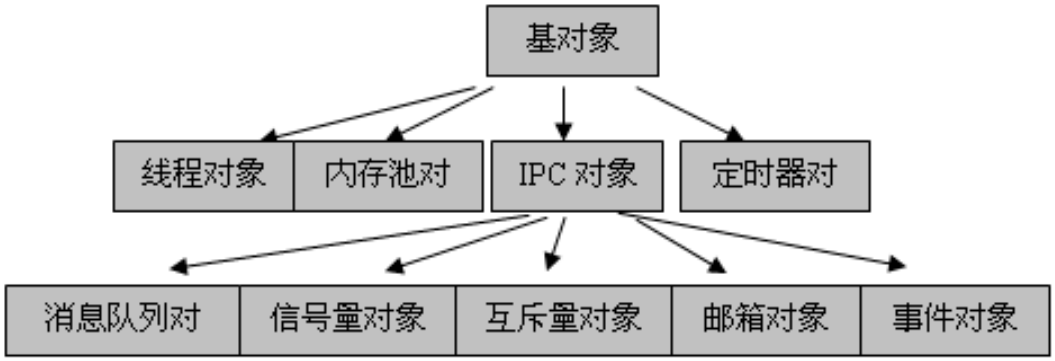


RT-Thread采用内核对象管理系统来访问/管理所有内核对象。内核对象包含了内核中绝大部分设施，而这些内核对象可以是静态分配的静态对象，也可以是从系统内存堆中分配的动态对象。通过内核对象系统，RT-Thread做到了不依赖于具体的内存分配方式，系统的灵活性得到极大的提高。

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。

下图显示了RT-Thread中各类内核对象的派生和继承关系。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操作中不会用到，只有在与具体线程相关的操作中才会使用。因此从面向对象的观点，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扩展了与自己相关的属性。



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。

5.2.3 对象控制块

```
struct rt_object
{
    /* 内核对象名称      */
    char       name[RT_NAME_MAX];
    /* 内核对象类型      */
    rt_uint8_t  type;
    /* 内核对象的参数    */
    rt_uint8_t  flag;
    /* 内核对象管理链表 */
    rt_list_t   list;
};
```

目前内核对象支持的类型如下：

```
enum rt_object_class_type
{
    RT_Object_Class_Thread = 0,      /* 对象为线程类型        */
#ifdef RT_USING_SEMAPHORE
    RT_Object_Class_Semaphore,       /* 对象为信号量类型      */
#endif
#ifdef RT_USING_MUTEX
    RT_Object_Class_Mutex,           /* 对象为互斥锁类型      */
```

```
#endif
#ifdef RT_USING_FASTEVENT
    RT_Object_Class_FastEvent,      /* 对象为快速事件类型    */
#endif
#ifdef RT_USING_EVENT
    RT_Object_Class_Event,          /* 对象为事件类型        */
#endif
#ifdef RT_USING_MAILBOX
    RT_Object_Class_MailBox,        /* 对象为邮箱类型        */
#endif
#ifdef RT_USING_MESSAGEQUEUE
    RT_Object_Class_MessageQueue,   /* 对象为消息队列类型    */
#endif
#ifdef RT_USING_MEMPOOL
    RT_Object_Class_MemPool,        /* 对象为内存池类型      */
#endif
#ifdef RT_USING_DEVICE
    RT_Object_Class_Device,         /* 对象为设备类型        */
#endif
    RT_Object_Class_Timer,          /* 对象为定时器类型      */
    RT_Object_Class_Unknown,        /* 对象类型未知          */
    RT_Object_Class_Static = 0x80   /* 对象为静态对象        */
};
```

5.2.4 内核对象接口

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; /* 对象类型 */
    rt_list_t object_list;          /* 对象链表 */
    rt_size_t object_size;          /* 对象大小 */
};
```

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。

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。

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。

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。

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。

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。

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。但是这种方法对于复杂一些的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。

6.1 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，系统应当及时发出振铃、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。

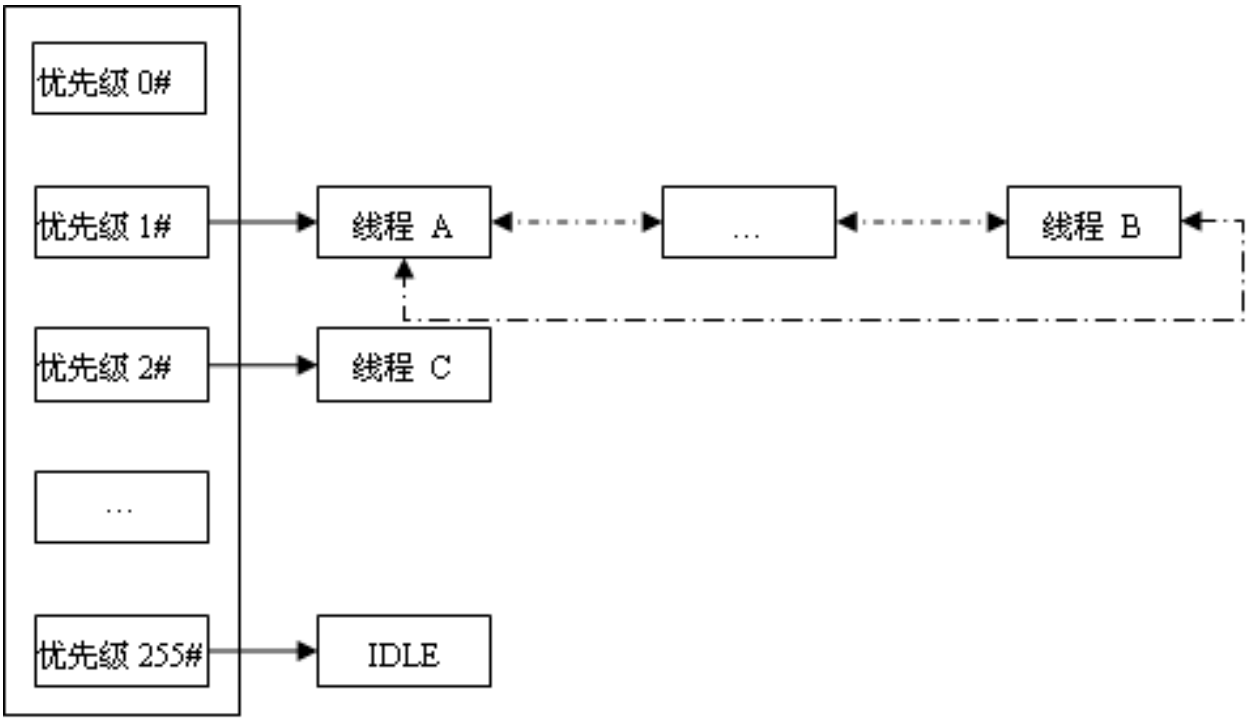
从上面的例子我们可以看出，实时系统是一种倾向性的系统，对于实时的事件需要在第一时间做出回应，而对非实时任务则可以在实时事件到达时为之让路 – 被抢占。所以实时系统也可以看成是一个分等级的系统，不同重要性的任务具有不同的优先级。

6.2 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。

如下图所示，RT-Thread调度器实现中包含一组，总共256个优先级队列数组(如果系统最大支持32个优先级，那么这里将是32个优先级队列数组)，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。



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（如果中途线程A被挂起了，线程B因为变成系统中就绪线程中最高的优先级，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。

6.3 线程控制块

线程控制块是操作系统用于控制线程的一个数据结构，它会存放线程的一些信息，例如优先级，线程名称等，也包含线程与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。

```
typedef struct rt_thread* rt_thread_t;

struct rt_thread
{
    /* rt object */
    char        name[RT_NAME_MAX];      /* 对象的名称 */
    rt_uint8_t  type;                   /* 对象的类型 */
    rt_uint8_t  flags;                  /* 对象的参数 */

    rt_list_t   list;                   /* 对象链表 */

    rt_thread_t tid;                    /* 线程ID */
    rt_list_t   tlist;                  /* 线程链表 */

    /* 栈指针和线程入口 */
    void*       sp;                     /* 线程的栈指针 */
    void*       entry;                  /* 线程的入口位置 */
    void*       parameter;              /* 入口参数 */
    void*       stack_addr;             /* 栈起始地址 */
    rt_uint16_t stack_size;             /* 栈大小 */

    rt_err_t    error;                  /* 线程运行过程的错误号 */

    /* priority */
    rt_uint8_t  current_priority;       /* 线程当前的优先级 */
    rt_uint8_t  init_priority;          /* 线程的初始优先级 */
#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)
    /* 线程事件 */
    rt_uint32_t event_set;
    rt_uint8_t  event_info;
#endif

    rt_uint8_t  stat;                   /* 线程状态 */

    rt_ubase_t  init_tick;              /* 线程初始分配的时间片 */
    rt_ubase_t  remaining_tick;         /* 线程当前运行时剩余的时间片 */

    struct rt_timer thread_timer;       /* 线程定时器 */

    rt_uint32_t user_data;              /* 用户数据 */
```

```
};
```

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。

6.4 线程状态

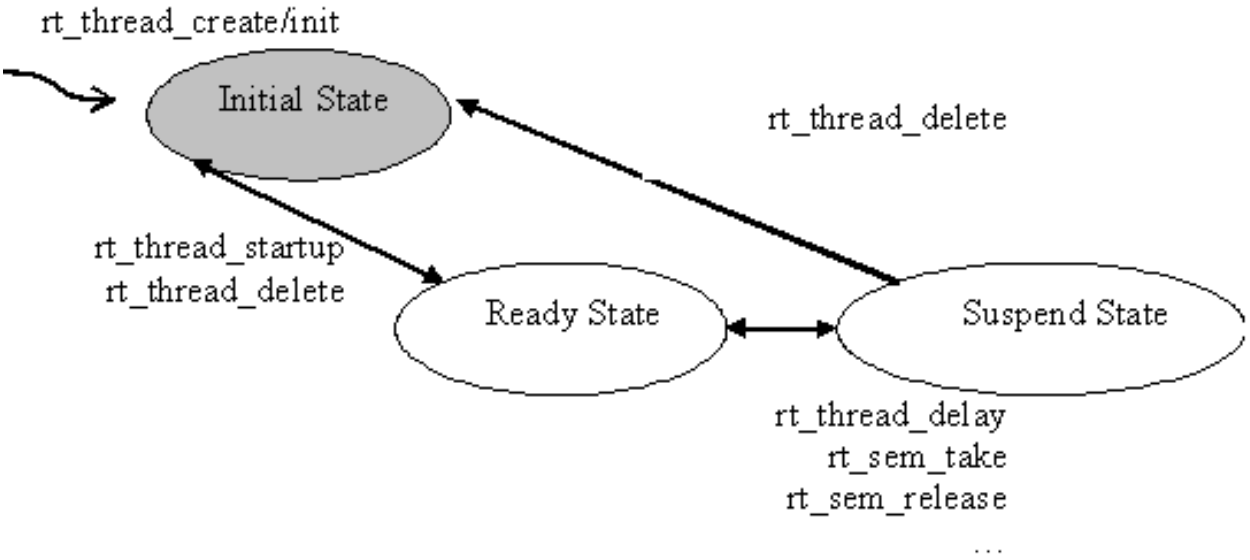
在线程运行的过程中，在一个时间内只允许一个线程中处理器中运行，即线程会有多种不同的线程状态，如运行态，非运行态等。在RT-Thread中，线程包含四种状态，操作系统会自动根据它运行的情况而动态调整它的状态。

RT-Thread中的四种线程状态：

State	Description
RT_THREAD_INIT 初始状态	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	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_RUNNING 运行态	运行态。线程当前正在运行，在单核系统中，只有rt_thread_self()函数返回的线程处于这个状态；在多核系统中则不受这个限制。

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四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：



线程通过调用函数rt_thread_create/init调用进入到初始状态（RT_THREAD_INIT），通过函数rt_thread_startup调用后进入到就绪状态（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（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。

6.5 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，可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，电源管理等。

6.6 调度器相关接口

6.6.1 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)
```

6.6.2 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。

6.6.3 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。

6.7 线程相关接口

6.7.1 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。

```
#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，入口是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;
}
```

6.7.2 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。

Note: 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：

```
#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;
}
```

6.7.3 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：

```
#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 %d\n", 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;
}
```

6.7.4 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。

6.7.5 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)
```

6.7.6 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)
```

Note: 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。

6.7.7 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：

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

Note: 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。

6.7.8 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）。

6.7.9 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 */

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

```
    while (1)
    {
        /* 获得当前线程 */
        thread = rt_thread_self();

        /* 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()
{
    rt_thread_t thread;

    /* 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;
}
```

6.7.10 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。

```
#include <rtthread.h>

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

```
    rt_thread_resume(thread);
    ...
}
```

6.7.11 线程控制

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

6.7.12 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)
```

6.7.13 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

SEVEN

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，那么就很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，称之为临界区。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。

7.1 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（因为整个系统已经不再响应外部事件，如果自身不主动放弃处理机，它也不会产生内部事件），也就是当前任务不会被抢占，除非这个任务主动放弃了处理机。关闭中断/恢复中断 API是由BSP实现的，根据不同的平台实现方式也不大相同。

关闭、打开中断由两个函数完成：

- 关闭中断

```
rt_base_t rt_hw_interrupt_disable()
```

关闭中断并返回关闭中断前的中断状态

- 恢复中断

```
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。

使用的例子代码如下：

```
#include <rtthread.h>

/* 同时访问的全局变量 */
static rt_uint32_t cnt;
void thread_entry(void* parameter)
{
    rt_uint32_t no;
    rt_uint32_t level;

    no = (rt_uint32) parameter;
    while(1)
    {
        /* 关闭中断 */
        level = rt_hw_interrupt_disable();
        cnt += no;
        /* 恢复中断 */
        rt_hw_interrupt_enable(level);

        rt_kprintf("thread[%d]'s counter is %d\n", no, cnt);
        rt_thread_delay(no);
    }
}

/* 用户应用程序入口 */
void rt_application_init()
{
    rt_thread_t thread;

    /* 创建t10线程 */
    thread = rt_thread_create("t10", thread_entry, (void*)10,
        512, 10, 5);
    if (thread != RT_NULL) rt_thread_startup(thread);

    /* 创建t20线程 */
    thread = rt_thread_create("t20", thread_entry, (void*)20,
        512, 20, 5);
    if (thread != RT_NULL) rt_thread_startup(thread);
}
```

7.2 调度器上锁

同样把调度器锁住也能让当前运行的任务不被换出，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，如果可能会被修改，那么将不适合采用此种方式作为同步的方法。

RT-Thread提供的调度器操作API为:: rt_enter_critical – 进入临界区

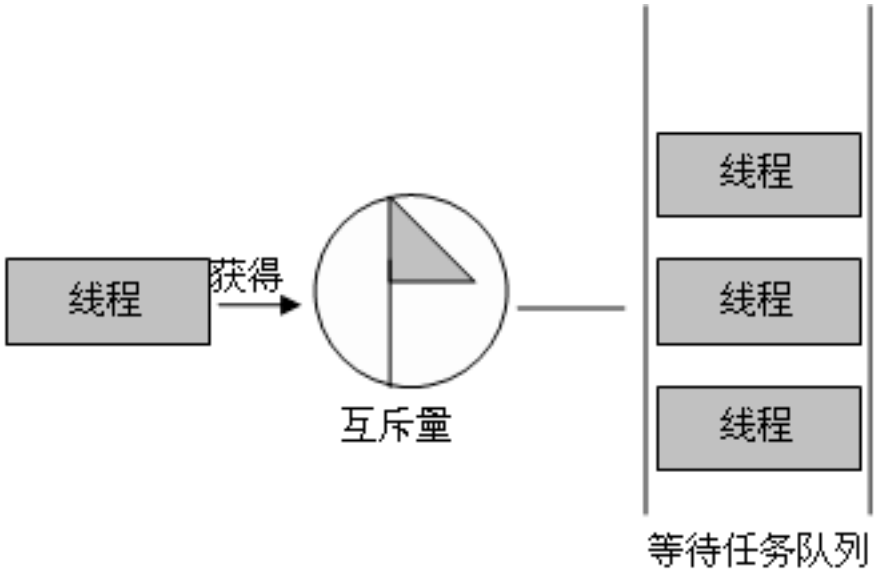
调用这个函数后，调度器将被上锁。在系统锁住调度器的期间，系统依然响应中断，但因为中断而可能
rt_exit_critical – 退出临界区

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，如果有比当前线程更高优先级的线程就绪，那么将切换到这个高优先级线程中执行；如果无更高优先级线程就绪，那么将继续执行当前任务。

Note: 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。嵌套的最大深度是256。

7.3 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下图所示，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。

7.3.1 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;                /* 互斥量的数值             */
    struct rt_thread* owner;        /* 当前拥有互斥量的线程     */
    rt_uint8_t original_priority;   /* 线程原始优先级           */
    rt_base_t hold;                 /* 等待线程数量             */
};
```

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

7.3.2 互斥量相关接口

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，需要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。

```
#define RT_IPC_FLAG_FIFO    0x00    /* IPC参数采用FIFO方式      */
#define RT_IPC_FLAG_PRIO    0x01    /* IPC参数采用优先级方式    */
```

采用基于优先级flag创建的IPC对象，将在多个线程等待资源时，由优先级高的线程优先获得资源。而采用基于FIFO flag创建的IPC对象，在多个线程等待资源时，按照先来先得的顺序获得资源。

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，所有等待此互斥量的线程都将被唤醒，等待线程获得返回值是-RT_ERROR。

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。

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链表中删除。

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。

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。

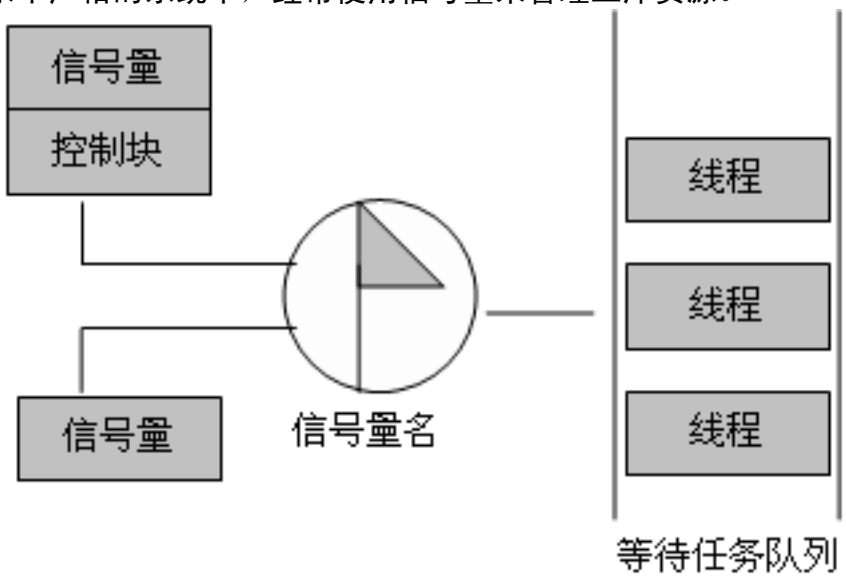
使用互斥量的例子如下

```
#include <rtthread.h>

/* 静态互斥锁对象 */
struct rt_mutex mutex;

/* 线程1入口 */
void thread1_entry(void *param)
{
    while(1)
    {
        /* 试图获得mutex互斥锁 */
        rt_mutex_take(&mutex, RT_WAITING_FOREVER);
        /* 再获得mutex互斥锁，因为mutex允许相同线程多次获得互斥锁，所以这步将立刻返回 */
        rt_mutex_take(&mutex, RT_WAITING_FOREVER);

        /* 打印获得mutex的信息，并休眠5秒 */
        rt_kprintf("thread1: taken mutex\n");
        rt_thread_delay(500);

        /*
         * 释放mutex互斥锁，因为在前面对于mutex对象调用了两次rt_mutex_take，
         * 所以这里也相应的释放两次。
         */
        rt_mutex_release(&mutex);
        rt_mutex_release(&mutex);
```

```
        /* 打印释放了mutex */
        rt_kprintf("thread1: release mutex\n");
    }
}

/* 线程2入口 */
void thread2_entry(void *param)
{
    while(1)
    {
        /* 先休眠1秒钟 */
        rt_kprintf("thread2: delay 1second\n");
        rt_thread_delay(100);

        /* 试图获得mutex互斥锁 */
        rt_kprintf("thread2: taken mutex\n");
        rt_mutex_take(&mutex, RT_WAITING_FOREVER);

        /* 休眠10秒钟 */
        rt_thread_delay(1000);

        /* 释放mutex */
        rt_mutex_release(&mutex);
    }
}

int rt_application_init()
{
    rt_thread_t thread;

    rt_mutex_init(&mutex, "mutext", RT_IPC_FLAG_FIFO);

    thread = rt_thread_create("t1",
        thread1_entry, RT_NULL,
        512, 250, 10);
    if (thread != RT_NULL) rt_thread_startup();

    thread = rt_thread_create("t2",
        thread2_entry, RT_NULL,
        512, 200, 10);
    if (thread != RT_NULL) rt_thread_startup();

    return 0;
}
```

7.4 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。

7.4.1 信号量控制块

```
struct rt_semaphore
{
    struct rt_ipc_object parent;     /* 信号量对象继承自ipc对象   */
    rt_base_t value;                 /* 信号量的值                */
};
```

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

7.4.2 信号量相关接口

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。

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）。

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。

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。

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。

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。

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。

使用信号量的例子

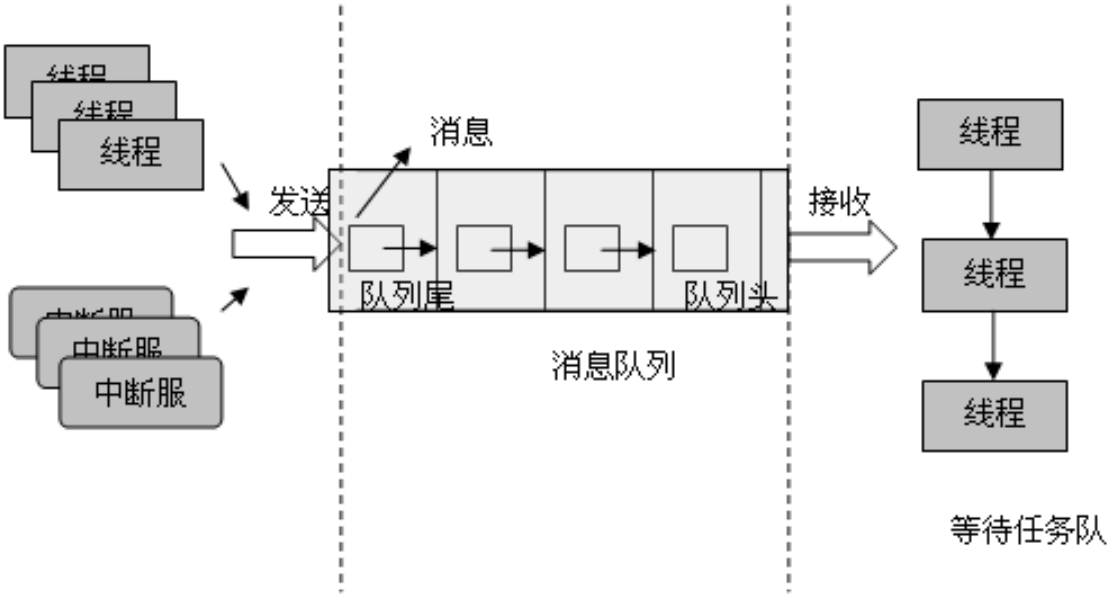
```
#include <rtthread.h>

/* 信号量对象 */
struct rt_semaphore sem;

/* thread1线程入口 */
void thread1_entry(void* parameter)
{
    while (1)
    {
        /* 采用永久等待方式获取信号量 */
        if (rt_sem_take(&sem, RT_WAITING_FOREVER) == RT_EOK)
        {
            /* 获取到信号量 */
            rt_kprintf("thread1:taken semaphore\n");

            rt_thread_delay(50);
            rt_sem_release(&sem);
        }
    }
}

/* thread2线程入口 */
void thread2_entry(void* parameter)
{
```

```
    while (1)
    {
        /* 采用永久等待方式获取信号量 */
        if (rt_sem_take(&sem, RT_WAITING_FOREVER) == RT_EOK)
        {
            rt_kprintf("thread2:taken semaphore\n");

            rt_thread_delay(10);
            rt_sem_release(&sem);
        }

        /* 以非等待方式试图获取信号量 */
        if (rt_sem_take(&sem, RT_WAITING_NO) == RT_EOK)
        {
            rt_kprintf("thread2:taken semaphore ok with no waiting\n");

            rt_thread_delay(10);
            rt_sem_release(&sem);
        }
        else
        {
            rt_kprintf("thread2: take semaphore failed with no waiting\n");
        }
    }
}

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

    /* 初始化sem对象 */
    rt_sem_init(sem, "semt", RT_IPC_FLAG_FIFO);

    /* 创建线程，入口是thread1_entry */
    thread = rt_thread_create("t1",
        thread1_entry, RT_NULL,
        512, 250, 10);
    if (thread != RT_NULL) rt_thread_startup();

    /* 创建线程，入口是thread2_entry */
    thread = rt_thread_create("t2",
        thread2_entry, RT_NULL,
        512, 200, 10);
    if (thread != RT_NULL) rt_thread_startup();

    return 0;
```

```
}
```

7.5 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，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。如上图所示，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。

7.5.1 消息队列控制块

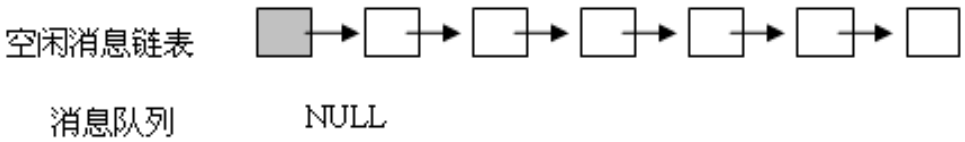
```
struct rt_messagequeue
{
```

```
    struct rt_ipc_object parent;

    void* msg_pool;          /* 存放消息的消息池开始地址 */

    rt_size_t msg_size;      /* 每个消息的长度           */
    rt_size_t max_msgs;      /* 最大运行的消息数         */

    void* msg_queue_head;    /* 消息链表头               */
    void* msg_queue_tail;    /* 消息链表尾               */
    void* msg_queue_free;    /* 空闲消息链表             */

    rt_ubase_t entry;        /* 队列中已经存放的消息数   */
};
```

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

7.5.2 消息队列相关接口

创建消息队列



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。如上图所示，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。

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。

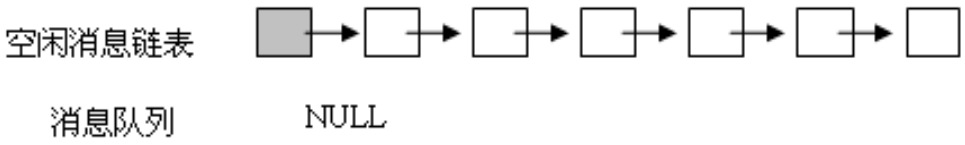
脱离消息队列

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。

初始化消息队列

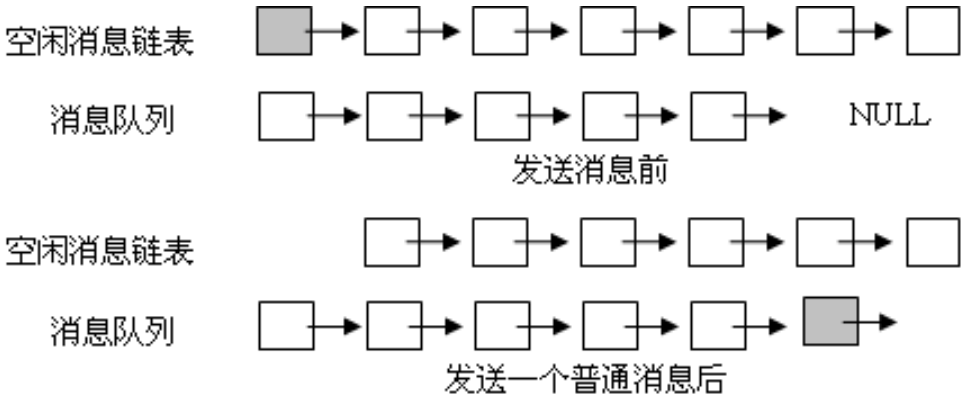


初始化消息队列跟创建消息队列类似，只是初始化消息队列用于静态内存管理模式，消息队列控制块来源于用户线程在系统中申请的静态对象。还与创建消息队列不同的是，此处消息队列对象所使用的内存空间是由用户线程提供的一个缓冲区空间，其余的初始化工作与创建消息队列时相同。初始化消息队列接口如下：

```
rt_err_t rt_mq_init(rt_mq_t mq, const char* name, void *msgpool, rt_size_t msg_size, rt_size_t poo
```

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。如上图所示，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。

发送消息

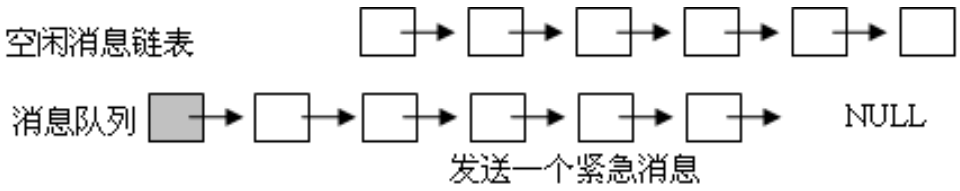


线程或者中断服务程序都可以给消息队列发送消息，当发送消息时，内核先从空闲消息链表上取下一个空闲消息块，把线程或者中断服务程序发送的消息内容复制到消息块上，然后把该消息块挂到消息队列的尾部。发送者成功发送消息当且仅当空闲消息链表上有可用的空闲消息块；当自由消息链表上无可用消息块，说明消息队列中的消息已满，此时，发送消息的的线程或者中断程序会收到一个错误码。发送消息接口如下：

```
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。

发送紧急消息

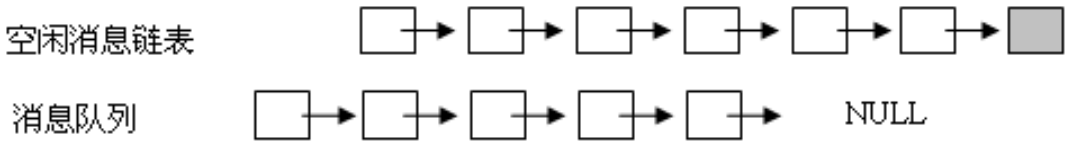


发送紧急消息的过程与发送消息几乎一样，唯一的不同是，当发送紧急消息时，从空闲消息链表上取下来的消息块不是挂到消息队列的队尾，而是挂到队首，这样，接收者能够优先接收到紧急消息，从而及时进行消息处理。发送紧急消息的接口如下：

```
rt_err_t rt_mq_urgent(rt_mq_t mq, void* buffer, rt_size_t size)
```

如上图所示，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。

接收消息



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。如上图所示，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。

使用消息队列的例子

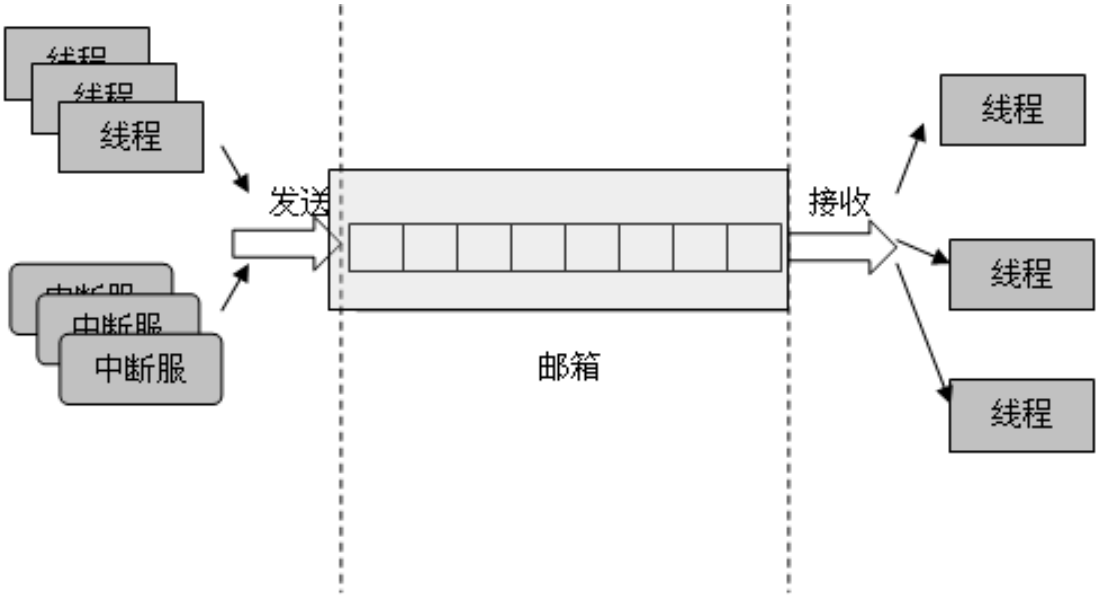
```
#include <rtthread.h>

/* 消息队列对象 */
struct rt_messagequeue mq;
/* 用于放置消息的内存池 */
char msg_pool[2048];

/* t1线程入口 */
void thread1_entry(void* parameter)
{
    char buf[128];

    while (1)
    {
        rt_memset(&buf[0], 0, sizeof(buf));

        /* 采用永久等待的方式从消息队列中读取消息 */
        if (rt_mq_recv(&mq, &buf[0], sizeof(buf), RT_WAITING_FOREVER) == RT_EOK)
        {
            rt_kprintf("thread1: recv msg from message queue, the content:%s\n", buf);
        }

        rt_thread_delay(100);
    }
}

/* t2线程入口 */
void thread2_entry(void* parameter)
{
    int i, result;
    char buf[] = "this is message No.x";

    while (1)
    {
        for (i = 0; i < 10; i++)
        {
            /* 构造测试数据 */
            buf[sizeof(buf) - 2] = '0' + i;

            rt_kprintf("thread2: send message - %s\n", buf);
            result = rt_mq_send(&mq, &buf[0], sizeof(buf));
            if ( result == -RT_EFULL);
            {
                /* 如果消息队列满的处理 */
                rt_kprintf("message queue full, delay 10s\n");
```

```
                rt_thread_delay(1000);
            }
        }

        rt_thread_delay(100);
    }
}

/* t3线程入口 */
void thread3_entry(void* parameter)
{
    char buf[] = "this is an urgent message!";

    while (1)
    {
        rt_kprintf("thread3: send an urgent message\n");

        /* 采用紧急消息的形式发送出去 */
        rt_mq_urgent(&mq, &buf[0], sizeof(buf));

        rt_thread_delay(250);
    }
}

int rt_application_init()
{
    rt_thread_t thread;

    /*
     * 初始化消息队列对象
     * 消息长度是128 - sizeof(void*)
     * 消息队列的总大小为sizeof(msg_pool)/4
     */
    rt_mq_init(&mq, "mqt", &msg_pool[0], 128 - sizeof(void*),
        sizeof(msg_pool)/4, RT_IPC_FLAG_FIFO);

    /* 创建t1线程 */
    thread = rt_thread_create("t1",
        thread1_entry, RT_NULL,
        512, 250, 10);
    if (thread != RT_NULL) rt_thread_startup();

    /* 创建t2线程 */
    thread = rt_thread_create("t2",
        thread2_entry, RT_NULL,
        512, 200, 10);
    if (thread != RT_NULL) rt_thread_startup();
```

```
    /* 创建t3线程 */
    thread = rt_thread_create("t3",
        thread2_entry, RT_NULL,
        512, 220, 10);
    if (thread != RT_NULL) rt_thread_startup();

    return 0;
}
```

7.6 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，如下图所示，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。

7.6.1 邮箱控制块

```
struct rt_mailbox
{
    struct rt_ipc_object parent;

    rt_uint32_t* msg_pool;              /* 邮件池地址，用于存放邮件 */

    rt_size_t size;                     /* 邮件池大小               */

    rt_ubase_t entry;                   /* 邮箱中接收到的邮件数目   */
    rt_ubase_t in_offset, out_offset;   /* 邮件池的进/出偏移位置    */
};
```

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

7.6.2 邮箱相关接口

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。

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。

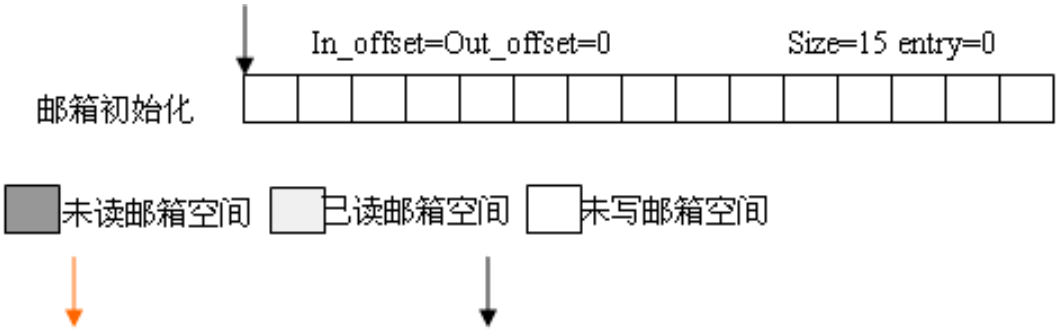
脱离邮箱

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。

初始化邮箱

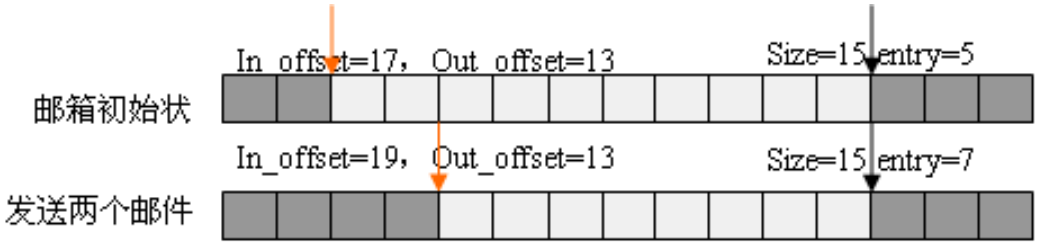


初始化邮箱跟创建邮箱类似，只是初始化邮箱用于静态内存管理模式，邮箱控制块来源于用户线程在系统中申请的静态对象。还与创建邮箱不同的是，此处邮箱对象所使用的内存空间是由用户线程指定的一个缓冲区空间，用户把缓冲区的指针传递给邮箱对象控制块，其余的初始化工作与创建邮箱时相同。接口如下：

```
rt_err_t rt_mb_init(rt_mailbox_t mb, const char* name, void* msgpool, rt_size_t size, rt_uint8 fl
```

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。如上图所示，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。

发送邮件



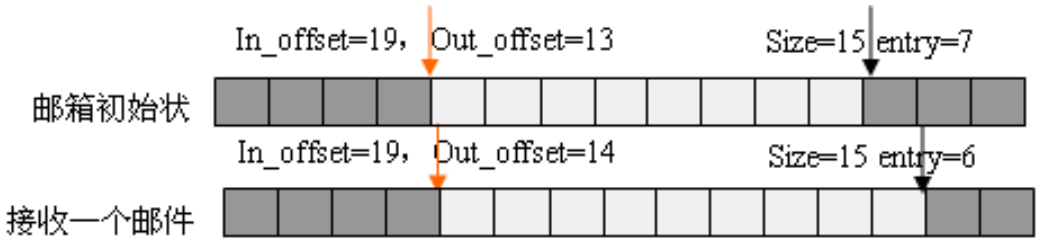
线程或者中断服务程序通过邮箱可以给其他线程发送邮件，发送的邮件可以是32位任意格式的数据，一个整型值或者指向一个缓冲区的指针，发送者成功发送邮件当且仅当邮箱未满，当邮箱中的邮件满时，发送邮件的线程或者中断程序会收到一个错误码。发送邮件接口如下：

```
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。如上图所示，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。

接收邮件



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。如上图所示，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。

使用邮箱的例子

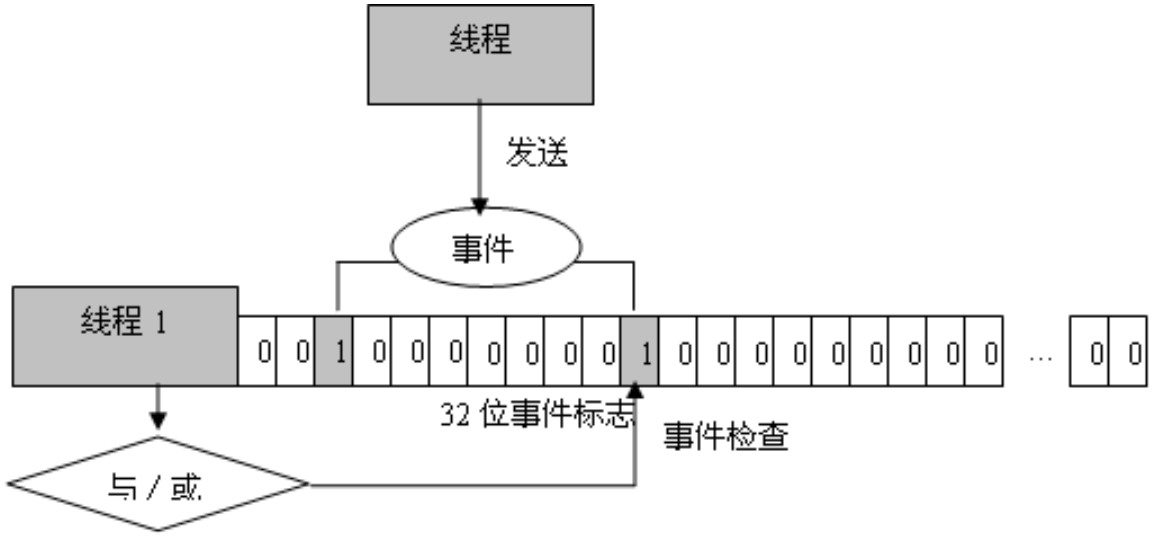
```
#include <rtthread.h>

/* 邮箱对象 */
struct rt_mailbox mb;
/* 用于存放邮件的内存池 */
char mb_pool[128];

char mb_str1[] = "I'm a mail!";
char mb_str2[] = "this is another mail!";

/* t1线程入口 */
void thread1_entry(void* parameter)
{
    unsigned char* str;

    while (1)
    {
        rt_kprintf("thread1: try to recv a mail\n");

        /* 从邮箱中收取信件，如果邮箱中为空，则直到等到收到邮件为止 */
```

```
        if (rt_mb_recv(&mb, (rt_uint32_t*)&str, RT_WAITING_FOREVER) == RT_EOK)
        {
            rt_kprintf("thread1: get a mail from mailbox, the content:%s\n", str);

            rt_thread_delay(100);
        }
    }
}

/* t2线程入口 */
void thread2_entry(void* parameter)
{
    rt_uint8_t count;

    count = 0;
    while (1)
    {
        count ++;
        if (count & 0x1)
        {
            /* 为奇数，发送mb_str1的首地址到邮箱中 */
            rt_mb_send(&mb, (rt_uint32_t)&mb_str1[0]);
        }
        else
        {
            /* 为偶数，发送mb_str2的首地址到邮箱中 */
            rt_mb_send(&mb, (rt_uint32_t)&mb_str2[0]);
        }

        rt_thread_delay(200);
    }
}

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

    /* 初始化邮箱对象，大小是128/4封邮件（每封邮件大小为4字节） */
    rt_mb_init(&mb, "mbt", &mb_pool[0], 128/4, RT_IPC_FLAG_FIFO);

    /* 创建t1线程 */
    thread = rt_thread_create("t1",
        thread1_entry, RT_NULL,
        512, 250, 10);
    if (thread != RT_NULL) rt_thread_startup();
```

```
    /* 创建t2线程 */
    thread = rt_thread_create("t2",
        thread2_entry, RT_NULL,
        512, 200, 10);
    if (thread != RT_NULL) rt_thread_startup();

    return 0;
}
```

7.7 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）。



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。

如上图所示，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。

7.7.1 事件控制块

```
struct rt_event
{
    struct rt_ipc_object parent;

    rt_uint32_t set;                /* 事件集合 */
};
```

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

7.7.2 事件相关接口

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。

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。

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。

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。

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_uint
```

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。

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。

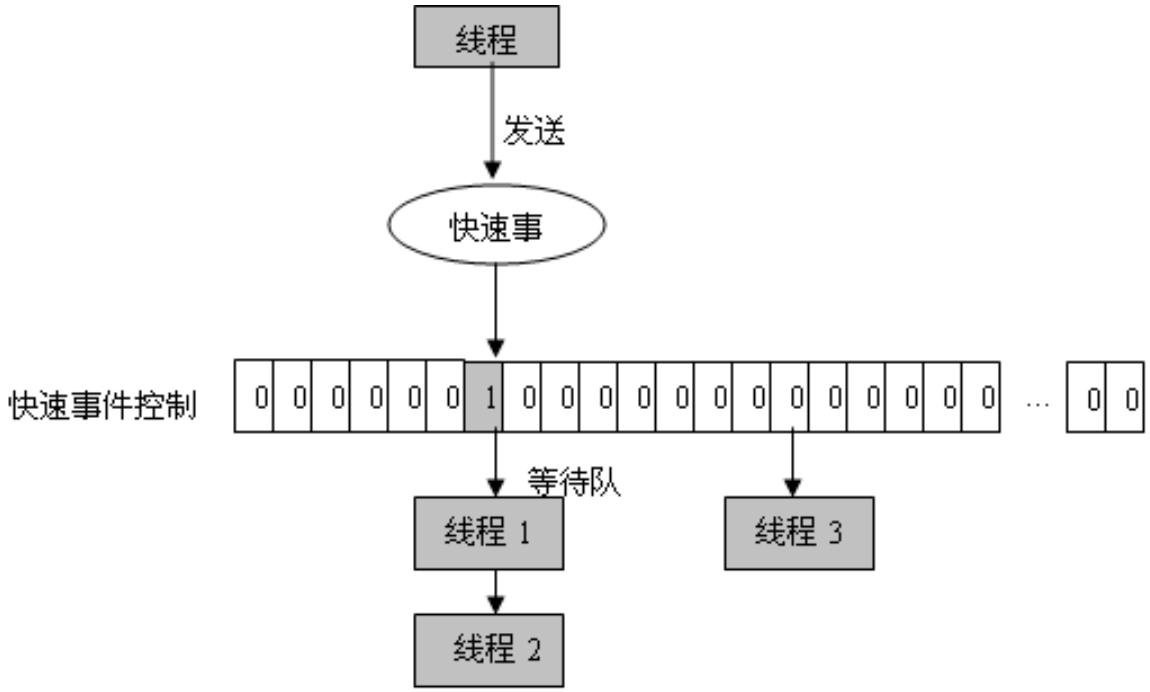
使用事件的例子

```
#include <rtthread.h>

/* 事件对象 */
struct rt_event event;

/* t1线程入口 */
void thread1_entry(void *param)
{
```

```
    rt_uint32_t e;

    while (1)
    {
        /* 收取第一个事件，采用 "与" 及收到并清除的方式 */
        if (rt_event_recv(&event, ((1 << 3) | (1 << 5)),
            RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR, RT_WAITING_FOREVER, &e) == RT_EOK)
        {
            /* 当收到3及5 bit位置位事件时，线程被唤醒 */
            rt_kprintf("thread1: AND recv event 0x%x\n", e);
        }

        rt_kprintf("thread1: delay 1s to prepare second event\n");
        rt_thread_delay(100);

        /* 收取第二个时间，采用 "或" 及收到并清除的方式 */
        if (rt_event_recv(&event, ((1 << 3) | (1 << 5)),
            RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR, RT_WAITING_FOREVER, &e) == RT_EOK)
        {
            /* 当收到3或5 bit位置位事件时，线程被唤醒 */
            rt_kprintf("thread1: OR recv event 0x%x\n", e);
        }

        rt_thread_delay(50);
    }
}

/* t2线程入口 */
void thread2_entry(void *param)
{
    while (1)
    {
        rt_kprintf("thread2: send event1\n");
        rt_event_send(&event, (1 << 3));

        rt_thread_delay(100);
    }
}

/* t3线程入口 */
void thread3_entry(void *param)
{
    while (1)
    {
        rt_kprintf("thread3: send event2\n");
        rt_event_send(&event, (1 << 5));
```

```
        rt_thread_delay(200);
    }
}

/* 用户应用程序 */
int rt_application_init()
{
    rt_thread_t thread;

    /* 初始化事件对象 */
    rt_event_init(&event, "event", RT_IPC_FLAG_FIFO);

    /* 创建t1线程 */
    thread = rt_thread_create("t1",
        thread1_entry, RT_NULL,
        512, 250, 10);
    if (thread != RT_NULL) rt_thread_startup();

    /* 创建t2线程 */
    thread = rt_thread_create("t2",
        thread2_entry, RT_NULL,
        512, 200, 10);
    if (thread != RT_NULL) rt_thread_startup();

    /* 创建t3线程 */
    thread = rt_thread_create("t3",
        thread2_entry, RT_NULL,
        512, 220, 10);
    if (thread != RT_NULL) rt_thread_startup();

    return 0;
}
```

7.8 快速事件



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。如上图所示，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。

7.8.1 快速事件控制块

```
struct rt_fast_event
{
    struct rt_object parent;                /* 继承于object对象 */
    rt_uint32 set;                          /* 事件集           */
    rt_list_t thread_list[RT_EVENT_LENGTH]; /* 线程队列         */
};
```

7.8.2 快速相关接口

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。

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。

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。

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。

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。

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。

使用快速事件的例子

```
#include <rtthread.h>

/* 快速事件对象 */
struct rt_fast_event fevent;

/* t1线程入口 */
void thread1_entry(void *param)
{
    while (1)
    {
        rt_kprintf("thread1: try to recv fast event\n");

        /* 接收3bit位事件，当收到时，时间集合自动清除 */
        if (rt_fast_event_recv(&fevent, 3,
            RT_EVENT_FLAG_CLEAR, RT_WAITING_FOREVER) == RT_EOK)
        {
            rt_kprintf("thread1: recv fase event\n");
        }

        rt_kprintf("thread1: delay 1s\n");
        rt_thread_delay(100);
    }
}

/* t2线程入口 */
void thread2_entry(void *param)
{
    while (1)
    {
        rt_kprintf("thread2: send fast event\n");
```

```
        /* 发送3bit位事件 */
        rt_fast_event_send(&fevent, 3);

        rt_thread_delay(100);
    }
}
#endif

/* 用户应用程序 */
int rt_application_init()
{
    rt_thread_t thread;

    /* 初始化快速事件对象 */
    rt_fast_event_init(&fevent, "fevent", RT_IPC_FLAG_FIFO);

    /* 创建t1线程 */
    thread = rt_thread_create("t1",
        thread1_entry, RT_NULL,
        512, 250, 10);
    if (thread != RT_NULL) rt_thread_startup();

    /* 创建t2线程 */
    thread = rt_thread_create("t2",
        thread2_entry, RT_NULL,
        512, 200, 10);
    if (thread != RT_NULL) rt_thread_startup();

    return 0;
}
```

CHAPTER

EIGHT

内存管理

在计算系统中，变量、中间数据一般存放在系统存储空间中，当实际使用时才从存储空间调入到中央处理器内部进行运算。存储空间，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去寻找适合长度的空闲内存块。这个适当，造成了搜索时间的不确定性，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过程在可预测的确定时间内完成，否则实时任务在对外部事响应也将变得时间不可确定性，例如一个处理数据的例子。

当一个外部数据达到时（通过传感器或网络数据包），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（可能搜索了1次，也可能搜索了若干次才能找到匹配的空间），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，但因为它们非连续性，不能组成一块连续的完整内存块，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)，但是这个对于可能需要数年工作于野外的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下去。

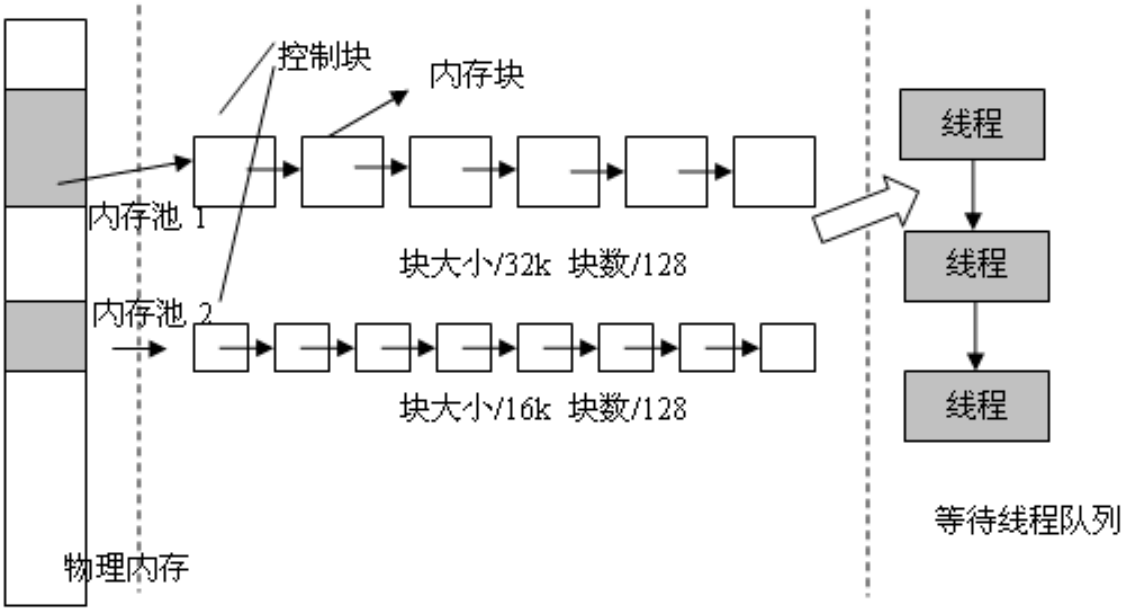
- 最后，嵌入式系统的资源环境也不是都相同，有些系统中资源比较紧张，只有数十KB的内存可供分配，而有些系统则存在数MB的内存。

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，有针对性的提供了数种内存分配管理算法：静态分区内存管理及动态内存管理。动态内存管理又更加可用内

存多少划分为两种情况，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。

8.1 静态内存池管理

8.1.1 静态内存池工作模式



上图是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。如上图所示，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。内存池一旦初始化完成，内部的内存块大小将不能再做调整。

8.1.2 静态内存池控制块

```
struct rt_mempool
{
    struct rt_object parent;
```

```
    void* start_address;              /* 内存池数据区域开始地址                          */
    rt_size_t size;                   /* 内存池数据区域大小                              */

    rt_size_t block_size;             /* 内存块大小                                      */
    rt_uint8_t* block_list;           /* 内存块列表                                      */

    rt_size_t block_total_count;      /* 内存池数据区域中能够容纳的最大内存块数          */
    rt_size_t block_free_count;       /* 内存池中空闲的内存块数                          */

    rt_list_t suspend_thread;         /* 因为内存块不可用而挂起的线程列表                */
    rt_size_t suspend_thread_count;   /* 因为内存块不可用而挂起的线程数                  */
};
```

8.1.3 静态内存池接口

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。

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。

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
```

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。

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，然后将内存池对象从内核对象管理器中删除。

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。

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。

内存池的使用例子如下

```
#include <rtthread.h>

/* 两个线程用到的TCB和栈 */
struct rt_thread thread1;
struct rt_thread thread2;
```

```
char thread1_stack[512];
char thread2_stack[512];

/* 内存池数据存放区域 */
char mempool[4096];

/* 内存池TCB */
struct rt_mempool mp;

/* 测试用指针分配头 */
char *ptr[48];

/* 测试线程1入口 */
void thread1_entry(void* parameter)
{
    int i;
    char *block;

    while(1)
    {
        /* 分配48个内存块 */
        for (i = 0; i < 48; i++)
        {
            rt_kprintf("allocate No.%d\n", i);
            ptr[i] = rt_mp_alloc(&mp, RT_WAITING_FOREVER);
        }

        /* 再分配一个内存块 */
        block = rt_mp_alloc(&mp, RT_WAITING_FOREVER);
        rt_kprintf("allocate the block mem\n");
        /* 是否分配的内存块 */
        rt_mp_free(block);
        block = RT_NULL;
    }
}

/* 测试线程2入口 */
void thread2_entry(void *parameter)
{
    int i;

    while(1)
    {
        rt_kprintf("try to release block\n");

        /* 释放48个已经分配的内存块 */
        for (i = 0 ; i < 48; i ++)
```

```
        {
            /* 非空才释放 */
            if (ptr[i] != RT_NULL)
            {
                rt_kprintf("release block %d\n", i);
                rt_mp_free(ptr[i]);

                /* 释放完成，把指针清零 */
                ptr[i] = RT_NULL;
            }
        }
    }
}

int rt_application_init()
{
    int i;
    for (i = 0; i < 48; i ++) ptr[i] = RT_NULL;

    /* 初始化一个内存池对象，每个内存块的大小是80个字节 */
    rt_mp_init(&mp, "mp1", &mempool[0],
        sizeof(mempool), 80);

    /* 初始化两个测试线程对象 */
    rt_thread_init(&thread1,
        "thread1",
        thread1_entry, RT_NULL,
        &thread1_stack[0], sizeof(thread1_stack),
        20, 10);

    rt_thread_init(&thread2,
        "thread2",
        thread2_entry, RT_NULL,
        &thread2_stack[0], sizeof(thread2_stack),
        25, 7);
    rt_thread_startup(&thread1);
    rt_thread_startup(&thread2);

    return 0;
}
```

8.2 动态内存管理

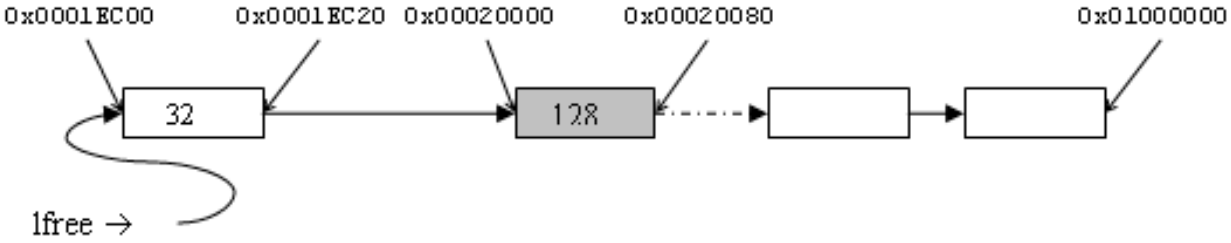
动态内存管理是一个真实的堆内存管理模块，可以根据用户的需求（在当前资源满足的情况下）分配任意大小的内存块。RT-Thread系统中为了满足不同的需求，提供了两套动态

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。

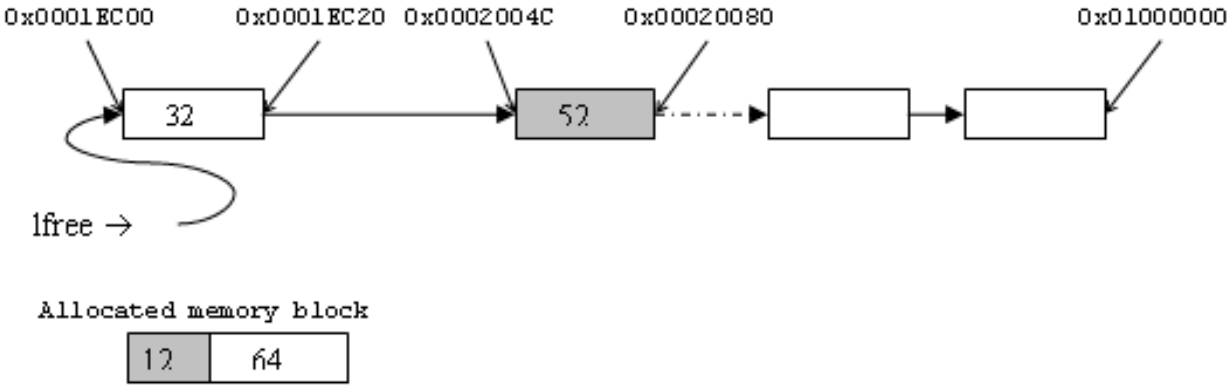
注：不要在中断服务例程中分配或释放动态内存块。

8.2.1 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。如下图所示



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。

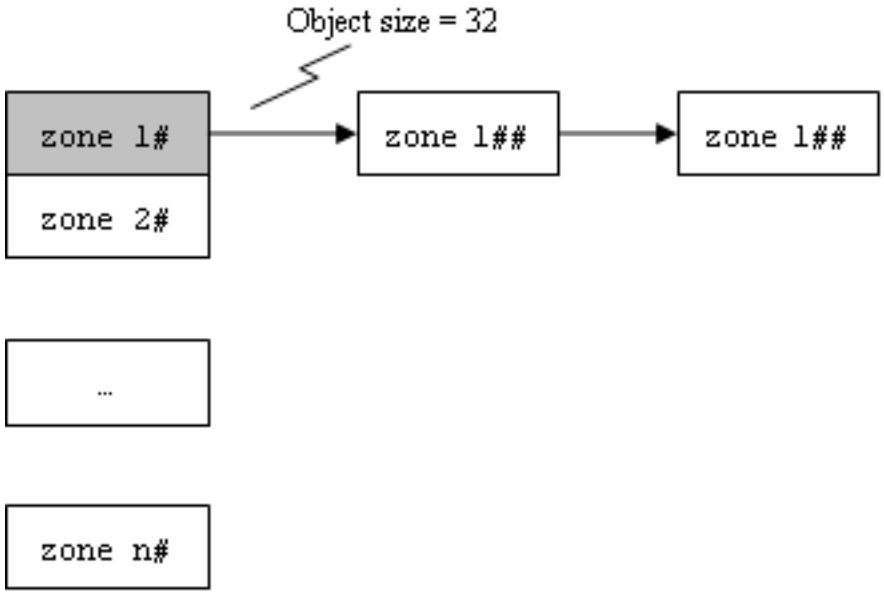
释放时则是相反的过程，但分配器会查看前后相邻的内存块是否空闲，如果空闲则合并成一个大的空闲内存块。

8.2.2 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。

8.2.3 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：

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

入口参数分别为堆内存的起始地址和结束地址。

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）。

动态内存分配的例子

```
/* 线程TCB和栈 */
struct rt_thread thread1;
char thread1_stack[512];

/* 线程入口 */
void thread1_entry(void* parameter)
{
    int i;
    char *ptr[20]; /* 用于放置20个分配内存块的指针 */

    /* 对指针清零 */
    for (i = 0; i < 20; i ++) ptr[i] = RT_NULL;

    while(1)
    {
        for (i = 0; i < 20; i++)
        {
            /* 每次分配(1 << i)大小字节数的内存空间 */
            ptr[i] = rt_malloc(1 << i);

            /* 如果分配成功 */
```

```
            if (ptr[i] != RT_NULL)
            {
                rt_kprintf("get memory: 0x%x\n", ptr[i]);

                /* 释放内存块 */
                rt_free(ptr[i]);
                ptr[i] = RT_NULL;
            }
        }
    }
}

int rt_application_init()
{
    /* 初始化线程对象 */
    rt_thread_init(&thread1,
        "thread1",
        thread1_entry, RT_NULL,
        &thread1_stack[0], sizeof(thread1_stack),
        200, 100);

    rt_thread_startup(&thread1);

    return 0;
}
```

CHAPTER

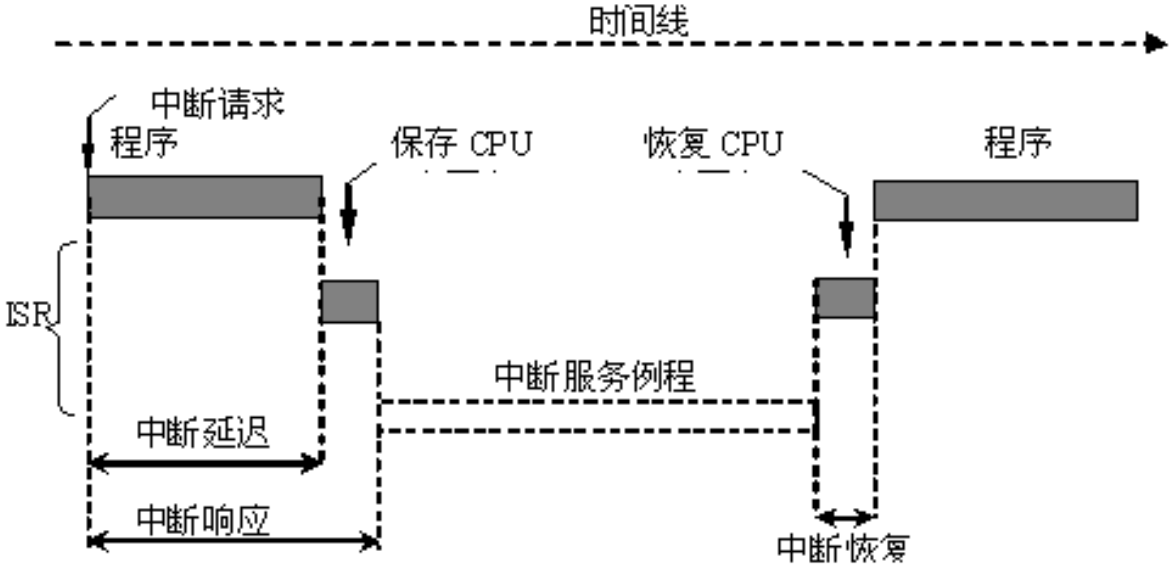
NINE

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。

9.1 中断处理过程

中断处理的一般过程如下图所示：

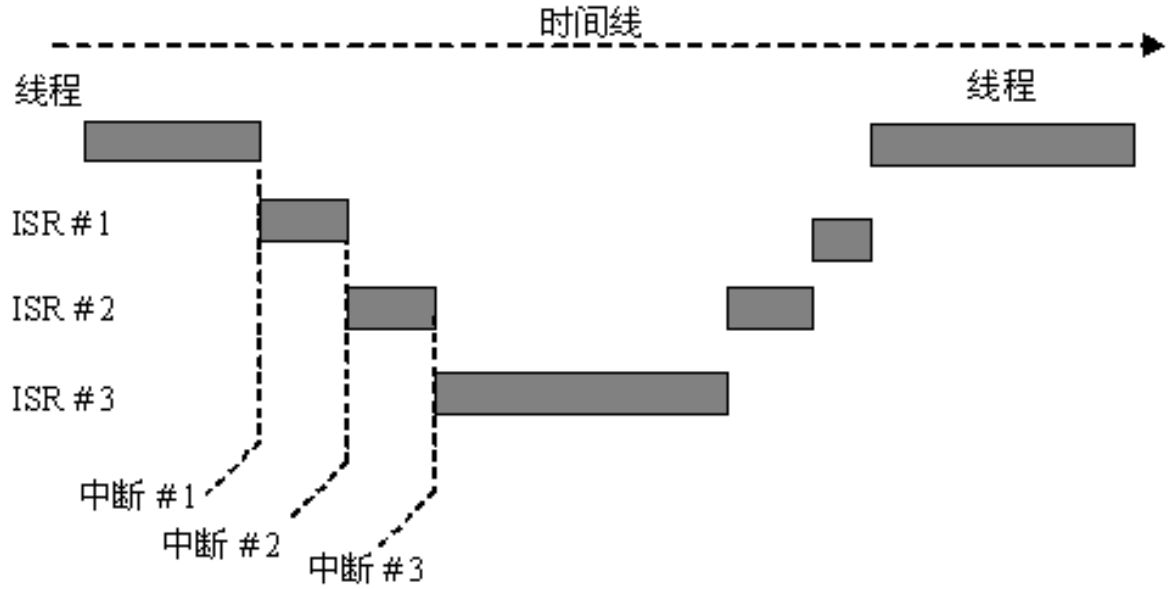


当中断产生时，处理机将按如下的顺序执行：

- 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

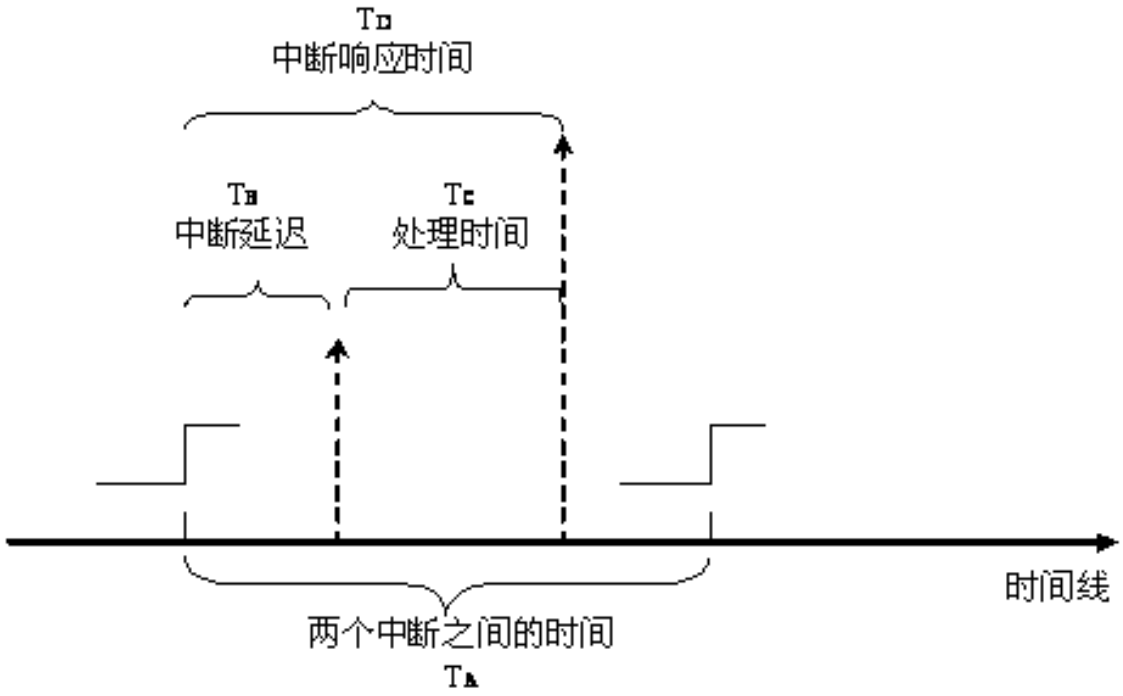
- 把控制权转交给处理函数并开始执行
- 当处理函数执行完成时，恢复处理器状态信息
- 从异常或中断中返回到前一个程序执行点

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相应事件发生。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。在执行中断服务例程过程中，如果有更高优先级别的中断源触发中断，由于当前处于中断处理上下文环境中，根据不同的处理器构架可能有不同的处理方式：新的中断等待挂起直到当前中断处理离开或打断当前中断处理过程，让处理器相应这个更高优先级的中断源。后面这种情况，一般称之为中断嵌套。在硬实时环境中，前一种情况是不允许发生的，关闭中断响应的时间应尽量的短。在软件处理上，RT-Thread允许中断嵌套，即在一个中断服务例程期间，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提供独立的系统栈，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，提高了内存空间的利用率，且随着任务的增加，这种技术的效果也越明显。

9.2 中断的底半处理

RT-Thread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，即上半部分（Top Half）和下半部分（Bottom Half）。在Top Half中，取得硬件状态和数据后，打开被屏蔽的中断，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，这一过程称之为Bottom Half。

9.2.1 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。

```
rt_sem_t demo_nw_isr;
void demo_nw_thread(void *param)
{
    /* 首先对设备进行必要的初始化工作 */
    device_init_setting();

    /* 装载中断服务例程 */
    rt_hw_interrupt_install(NW_IRQ_NUMBER, demo_nw_isr, RT_NULL);
    rt_hw_interrupt_umask(NW_IRQ_NUMBER);

    [...]

    /* 创建一个semaphore来响应Bottom Half的事件 */
    nw_bh_sem = rt_sem_create("bh_sem", 1, RT_IPC_FLAG_FIFO);

    while(1)
    {
        /* 最后，让demo_nw_thread等待在nw_bh_sem上 */
        rt_sem_take(nw_bh_sem, RT_WAITING_FOREVER);

        /* 接收到semaphore信号后，开始真正的Bottom Half处理过程 */
        nw_packet_parser (packet_buffer);
        nw_packet_process(packet_buffer);
    }
}

int rt_application_init()
{
    rt_thread_t thread;

    /* 创建处理线程 */
    thread = rt_thread_create("nwt",
    demo_nw_thread, RT_NULL,
    1024, 20, 5);

    if (thread != RT_NULL)
        rt_thread_startup(thread);
}
```

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。

```
void demo_nw_isr(int vector)
{
    /* 当network设备接收到数据后，陷入中断异常，开始执行此ISR */
    /* 开始Top Half部分的处理，如读取硬件设备的状态以判断发生了何种中断 */
    nw_device_status_read();

    [...]

    /* 释放nw_bh_sem，发送信号给demo_nw_thread，准备开始Bottom Half */
    rt_sem_release(nw_bh_sem);

    /* 然后退出中断的Top Half部分，结束device的ISR */
}
```

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。

9.3 中断相关接口

RT-Thread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，在某些支持处理器支持分支中并不一定存在，例如STM32）

9.3.1 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_handl
```

9.3.2 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)
```

9.3.3 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)
```

9.3.4 关闭中断

当需要关闭中断以屏蔽整个系统的事件处理时，调用如下接口：

```
rt_base_t rt_hw_interrupt_disable()
```

当系统关闭了中断时，就意味着当前任务/代码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。

9.3.5 打开中断

打开中断往往是和关闭中断成对使用的，用于恢复关闭中断前的状态。接口如下：

```
void rt_hw_interrupt_enable(rt_base_t level)
```

调用这个接口将恢复调用rt_hw_interrupt_disable前的中断状态，level是上一次关闭中断时返回的值。

注：调用这个接口并不代表着肯定会打开中断，而是恢复关闭中断前的状态，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。

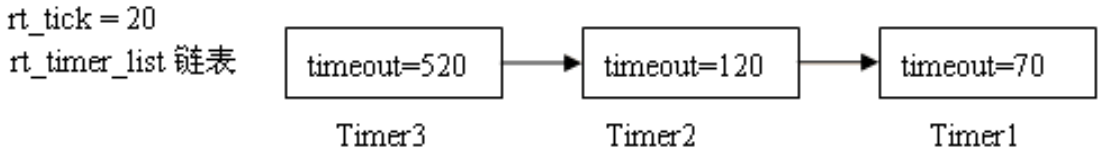
CHAPTER

TEN

定时器与系统时钟

10.1 定时器管理

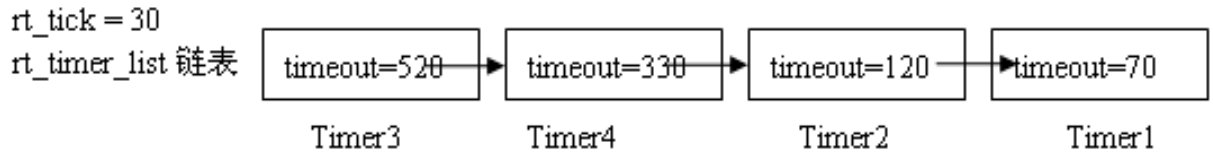
定时器，故名思义，是指时间到达一个设置的值后触发一个事件，定时器有硬件定时器和软件定时器之分，硬件定时器是指芯片本身提供的定时功能，一般是由外部晶振提供给芯片输入时钟，芯片向软件模块提供一组配置寄存器，接受控制输入，到达设定时间值后芯片中断控制器产生时钟中断。硬件定时器的精度一般很高，可以达到纳秒级别。在RT-Thread RTOS中，定时器模块以tick为时间单位，tick的时间长度为两次硬件定时器中断间隔的时间，这个时间可以根据不同的系统MIPS和实时性需求设置不同的值。RT-Thread是以tick作为任务调度的时间单位的，tick值设置越小，实时精度越高，但是系统开销也越大。一个嵌入式系统中，往往对定时器的需求比较广泛，有时会同时使用几个甚至几十个定时器，而这类定时器精度要求也并不苛刻，显然，纯粹硬件不能满足这需求，那么，软件定时器就有了表现舞台了，软件定时器是操作系统提供的一类系统接口，使系统能够提供不受数目限制的定时器服务。RT-Threa的类定时器以tick为单位，提供两类定时器机制，第一类是单次触发定时器，这类定时器只会触发一次定时器事件，第二类则是周期触发定时器，这类定时器会周期性的触发定时器事件。下面以实际例子来说明RT-Thread软件定时器的基本工作原理,在RT-Thread定时器模块维护两个重要的全局变量，一个是当前系统的时间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



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。



10.2 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. */
};
```

10.3 定时器管理接口

10.3.1 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()
```

10.3.2 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周期定时。创建定时器的例子如下代码所示。

```
#include <rtthread.h>

/* 定时器timer1的触发函数 */
void timeout1(void* parameter)
{
        rt_kprintf("periodic timer is timeout\n");
}

/* 定时器timer2的触发函数 */
void timeout2(void* parameter)
{
        rt_kprintf("one shot timer is timeout\n");
}

/* 用户应用程序入口 */
int rt_application_init()
{
        rt_timer_t timer1;
        rt_timer_t timer2;

        /* 创建周期性定时器timer1 */
        timer1 = rt_timer_create("timer1", timeout1, RT_NULL, 200, RT_TIMER_FLAG_PERIODIC);
        /* 创建单次定时器timer2 */
        timer2 = rt_timer_create("timer2", timeout2, RT_NULL, 200, RT_TIMER_FLAG_ONE_SHOT);

        /*......*/

        return 0;
}
```

10.3.3 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)
```

删除定时器的例子如下代码所示。

```
/* 用户应用程序入口 */
int rt_application_init()
{
```

```
        rt_timer_t timer1;
        rt_timer_t timer2;

        /* 创建周期性定时器timer1 */
        timer1 = rt_timer_create("timer1", timeout1, RT_NULL, 200, RT_TIMER_FLAG_PERIODIC);
        /* 创建单次定时器timer2 */
        timer2 = rt_timer_create("timer2", timeout2, RT_NULL, 200, RT_TIMER_FLAG_ONE_SHOT);

        /*......*/

        /* 删除定时器timer1 */
        rt_timer_delete(&timer1);
        /* 删除定时器timer2 */
        rt_timer_delete(&timer2);

        return 0;
}
```

10.3.4 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周期定时。初始化定时器的例子如下代码所示。

```
#include <rtthread.h>

struct rt_timer timer1;
struct rt_timer timer2;

/* 定时器timer1的触发函数 */
void timeout1(void* parameter)
{
        rt_kprintf("periodic timer is timeout\n");
}

/* 定时器timer2的触发函数 */
void timeout2(void* parameter)
{
        rt_kprintf("one shot timer is timeout\n");
}
```

```
/* 用户应用程序入口 */
int rt_application_init()
{
        /* 初始化timer1为周期性定时器 */
        rt_timer_init(&timer1, "timer1", timeout1, RT_NULL, 200, RT_TIMER_FLAG_PERIODIC);
        /* 初始化timer2为单次定时器 */
        rt_timer_init(&timer2, "timer2", timeout2, RT_NULL, 200, RT_TIMER_FLAG_ONE_SHOT);

        /* 启动定时器timer1 */
        rt_timer_start(&timer1);
        /* 启动定时器timer2 */
        rt_timer_start(&timer2);

        return 0;
}
```

10.3.5 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)
```

脱离定时器的例子如下代码所示。

```
/* 用户应用程序入口 */
int rt_application_init()
{
        /* 初始化timer1为周期性定时器 */
        rt_timer_init(&timer1, "timer1", timeout1, RT_NULL, 200, RT_TIMER_FLAG_PERIODIC);
        /* 初始化timer2为单次定时器 */
        rt_timer_init(&timer2, "timer2", timeout2, RT_NULL, 200, RT_TIMER_FLAG_ONE_SHOT);

        /*......*/

        /* 脱离定时器timer1 */
        rt_timer_detach(&timer1);
        /* 脱离定时器timer2 */
        rt_timer_start(&timer2);

        return 0;
}
```

10.3.6 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)
```

启动定时器的例子如下代码所示。

```
#include <rtthread.h>

/* 用户应用程序入口 */
int rt_application_init()
{
        rt_timer_t timer1;
        rt_timer_t timer2;

        /* 创建周期性定时器timer1 */
        timer1 = rt_timer_init("timer1", timeout1, RT_NULL, 200, RT_TIMER_FLAG_PERIODIC);
        /* 创建单次定时器timer2 */
        timer2 = rt_timer_init("timer2", timeout2, RT_NULL, 200, RT_TIMER_FLAG_ONE_SHOT);

        /* 启动定时器timer1 */
        rt_timer_start(&timer1);
        /* 启动定时器timer2 */
        rt_timer_start(&timer2);

        return 0;
}
```

10.3.7 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)
```

停止定时器的例子如下代码所示。

```
#include <rtthread.h>

/* 用户应用程序入口 */
int rt_application_init()
{
        rt_timer_t timer1;
        rt_timer_t timer2;
```

```
        /* 创建周期性定时器timer1 */
        timer1 = rt_timer_init("timer1", timeout1, RT_NULL, 200, RT_TIMER_FLAG_PERIODIC);
        /* 创建单次定时器timer2 */
        timer2 = rt_timer_init("timer2", timeout2, RT_NULL, 200, RT_TIMER_FLAG_ONE_SHOT);

        /*......*/

        /* 停止定时器timer1 */
        rt_timer_stop(&timer1);
        /* 停止定时器timer2 */
        rt_timer_stop(&timer2);

        return 0;
}
```

10.3.8 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。控制定时器的例子如下代码所示。

```
#include <rtthread.h>

/* 用户应用程序入口 */
int rt_application_init()
{
        rt_timer_t timer1;
        rt_timer_t timer2;

        /* 创建周期性定时器timer1 */
        timer1 = rt_timer_init("timer1", timeout1, RT_NULL, 200, RT_TIMER_FLAG_PERIODIC);
        /* 创建单次定时器timer2 */
        timer2 = rt_timer_init("timer2", timeout2, RT_NULL, 200, RT_TIMER_FLAG_ONE_SHOT);
```

```
        /*......*/

        /* 控制定时器timer1，重新设置timer1定时时间 */
        rt_timer_control(timer1, RT_TIMER_CTRL_SET_TIME, 100);

        /* 控制定时器timer2，重新设置timer2为单次触发类型，重新设置定时时间 */
        rt_timer_control(timer2，RT_TIMER_CTRL_SET_ONESHOT，100 );

        return 0;
}
```

10.3.9 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()
```

此函数为系统内部接口，不对外开放

CHAPTER

ELEVEN

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上层应用之间的一个中间层。

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访问底层设备，设备驱动程序的升级不会对上层应用产生影响。这种方式使得与设备的硬件操作相关的代码与应用相隔离，双方只需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。下图即为设备对象的继承和派生关系示意图。

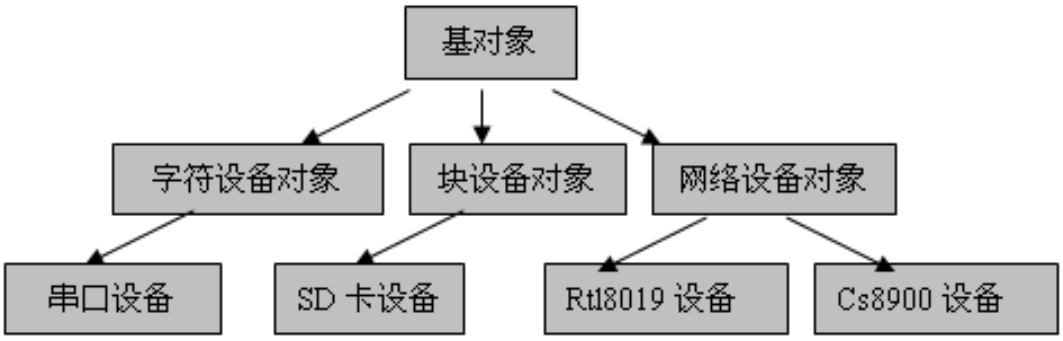


Figure 11.1: 设备对象的继承和派生关系示意图

11.1 I/O设备管理控制块

```
struct rt_device
{
```

```
    struct rt_object parent;

    /* 设备类型 */
    enum rt_device_class_type type;
    /* 设备参数及打开时参数 */
    rt_uint16_t flag, open_flag;

    /* 设备回调接口 */
    rt_err_t (*rx_indicate)(rt_device_t dev, rt_size_t size);
    rt_err_t (*tx_complete)(rt_device_t dev, void* buffer);

    /* 设备公共接口 */
    rt_err_t  (*init)   (rt_device_t dev);
    rt_err_t  (*open)   (rt_device_t dev, rt_uint16_t oflag);
    rt_err_t  (*close)  (rt_device_t dev);
    rt_size_t (*read)   (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size);
    rt_size_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);

#ifdef RT_USING_DEVICE_SUSPEND
    rt_err_t (*suspend) (rt_device_t dev);
    rt_err_t (*resumed) (rt_device_t dev);
#endif

    /* 设备私有数据 */
    void* private;
};
```

当前RT-Thread支持的设备类型包括：

```
enum rt_device_class_type
{
        RT_Device_Class_Char = 0,       /* 字符设备     */
        RT_Device_Class_Block,          /* 块设备       */
        RT_Device_Class_NetIf,          /* 网络接口设备 */
        RT_Device_Class_MTD,            /* 内存设备     */
        RT_Device_Class_CAN,            /* CAN设备      */
        RT_Device_Class_Unknown         /* 未知设备     */
};
```

Note: suspend、resume回调函数只会在“RT_USING_DEVICE_SUSPEND“宏使能的情况下才会有效。

11.2 I/O设备管理接口

11.2.1 注册设备

在一个设备能够被上层应用访问前，需要先把这个设备注册到系统中，并添加一些相应的属性。这些注册的设备均可以采用“查找设备接口”通过设备名来查找设备，获得该设备控制块。注册设备的原始如下：

```
rt_err_t rt_device_register(rt_device_t dev, const char* name, rt_uint8_t flags)
```

其中调用的flags参数支持如下列表中的参数(可以采用或的方式支持多种参数)：

```
#define RT_DEVICE_FLAG_DEACTIVATE       0x000   /* 未初始化设备                 */

#define RT_DEVICE_FLAG_RDONLY           0x001   /* 只读设备                     */
#define RT_DEVICE_FLAG_WRONLY           0x002   /* 只写设备                     */
#define RT_DEVICE_FLAG_RDWR             0x003   /* 读写设备                     */

#define RT_DEVICE_FLAG_REMOVABLE        0x004   /* 可移除设备                   */
#define RT_DEVICE_FLAG_STANDALONE       0x008   /* 独立设备                     */
#define RT_DEVICE_FLAG_ACTIVATED        0x010   /* 已激活设备                   */
#define RT_DEVICE_FLAG_SUSPENDED        0x020   /* 挂起设备                     */
#define RT_DEVICE_FLAG_STREAM           0x040   /* 设备处于流模式               */

#define RT_DEVICE_FLAG_INT_RX           0x100   /* 设备处于中断接收模式         */
#define RT_DEVICE_FLAG_DMA_RX           0x200   /* 设备处于DMA接收模式          */
#define RT_DEVICE_FLAG_INT_TX           0x400   /* 设备处于中断发送模式         */
#define RT_DEVICE_FLAG_DMA_TX           0x800   /* 设备处于DMA发送模式          */
```

RT_DEVICE_FLAG_STREAM参数用于向串口终端输出字符串，当输出的字符是”n”时，自动在前面补一个”r”做分行。

11.2.2 卸载设备

将设备从设备系统中卸载，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)
```

Note: 卸载设备并不会释放设备控制块所占用的内存。

11.2.3 初始化所有设备

初始化所有注册到设备对象管理器中的未初始化的设备，可以通过如下接口完成：

```
rt_err_t rt_device_init_all(void)
```

Note: 如果设备的flags域已经是RT_DEVICE_FLAG_ACTIVATED，调用这个接口将不再重复做初始化，一个设备初始化完成后它的flags域RT_DEVICE_FLAG_ACTIVATED应该被置位。

11.2.4 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，如果没有找到相应的设备对象，则返回RT_NULL。

11.2.5 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, rt_uint16_t oflags)
```

其中oflags支持以下列表中的参数：

```
#define RT_DEVICE_OFLAG_RDONLY      0x001   /* 只读模式访问     */
#define RT_DEVICE_OFLAG_WRONLY      0x002   /* 只写模式访问     */
#define RT_DEVICE_OFLAG_RDWR        0x003   /* 读写模式访问     */
```

Note: 如果设备flags域包含RT_DEVICE_FLAG_STANDALONE参数，将不允许重复打开。

11.2.6 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)
```

11.2.7 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_size_t rt_device_read (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)
```

根据底层驱动的实现，通常这个接口并不会阻塞上层应用线程。返回值是读到数据的大小(以字节为单位)，如果返回值是0，需要读取当前线程的errno来判断错误状态。

11.2.8 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_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)
```

根据底层驱动的实现，通常这个接口并不会阻塞上层应用线程。返回值是写入数据的大小(以字节为单位)，如果返回值是0，需要读取当前线程的errno来判断错误状态。

11.2.9 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)
```

cmd命令参数通常是和设备驱动程序相关的。

11.2.10 设置数据接收指示

设置一个回调函数，当硬件设备收到数据时回调给应用程序以通知有数据达到。可以通过如下接口完成设置接收指示：

```
rt_err_t rt_device_set_rx_indicate(rt_device_t dev, rt_err_t (*rx_ind )(rt_device_t dev, rt_size_t
```

回调函数rx_ind由调用者提供，当硬件设备接收到数据时，会回调这个函数并把收到的数据长度放在size参数中传递给上层应用。上层应用线程应在收到指示时，立刻从设备中读取数据。

11.2.11 设置发送完成指示

在上层应用调用rt_device_write写入数据时，如果底层硬件能够支持自动发送，那么上层应用可以设置一个回调函数。这个回调函数会在底层硬件给出发送完成时(例如DMA传送完成或FIFO已经写入完毕产生完成中断时)被调用。可以通过如下接口完成设备发送完成指示：

```
rt_err_t rt_device_set_tx_complete(rt_device_t dev, rt_err_t (*tx_done)(rt_device_t dev, void *buf
```

回调函数tx_done由调用者提供，当硬件设备发送完数据时，由驱动程序回调这个函数并把发送完成的数据块地址buffer做为参数传递给上层应用。上层应用（线程）在收到指示时应根据发送buffer的情况，释放buffer内存块或为下一个写数据做缓存。

11.3 设备驱动

上一节说到了如何使用RT-Thread的设备接口，但对于开发人员来说，如何编写一个驱动设备可能会更加重要。

11.3.1 设备驱动必须实现的接口

我们先来看看/解析下RT-Thread的设备控制块：

```
struct rt_device
{
    struct rt_object parent;

    /* 设备类型 */
    enum rt_device_class_type type;
    /* 设备参数及打开时的参数 */
    rt_uint16_t flag, open_flag;

    /* 设备回调函数 */
    rt_err_t (*rx_indicate)(rt_device_t dev, rt_size_t size);
    rt_err_t (*tx_complete)(rt_device_t dev, void* buffer);

    /* 公共的设备接口 */
    rt_err_t  (*init)   (rt_device_t dev);
    rt_err_t  (*open)   (rt_device_t dev, rt_uint16_t oflag);
    rt_err_t  (*close)  (rt_device_t dev);
    rt_size_t (*read)   (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size);
    rt_size_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);

            /* 当使用了设备挂起功能时的接口 */
#ifdef RT_USING_DEVICE_SUSPEND
    rt_err_t (*suspend) (rt_device_t dev);
    rt_err_t (*resumed) (rt_device_t dev);
#endif

    /* device private data */
    void* private;
};
```

其中包含了一个套公共的设备接口(类似上节说的设备访问接口，但面向的层次已经不一样了)：

```
/* 公共的设备接口 */
rt_err_t  (*init)   (rt_device_t dev);
rt_err_t  (*open)   (rt_device_t dev, rt_uint16_t oflag);
rt_err_t  (*close)  (rt_device_t dev);
rt_size_t (*read)   (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size);
rt_size_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);

    /* 当使用了设备挂起功能时的接口 */
#ifdef RT_USING_DEVICE_SUSPEND
rt_err_t (*suspend) (rt_device_t dev);
rt_err_t (*resumed) (rt_device_t dev);
#endif
```

这些接口也是上层应用通过RT-Thread设备接口进行访问的实际底层接口，在满足一定的条件下，都会调用到这套接口。其中suspend和resume接口是应用于RT-Thread的电源管理部分，目前的0.3.0版本并不支持，预留给以后使用。

其他的六个接口，可以看成是底层设备驱动必须提供的接口。

Name	Description
init	设备的初始化。设备初始化完成后，设备控制块的flag会被置成已激活状态(RT_DEVICE_FLAG_ACTIVATED)。如果设备控制块的flag不是已激活状态，那么在设备框架调用rt_device_init_all接口时调用此设备驱动的init接口进行设备初始化。
open	打开设备。有些设备并不是系统一启动就已经打开开始运行的，或者设备需要进行数据接收，但如果上层应用还未准备好，设备也不应默认已经使能开始接收数据。所以建议底层驱动程序，在调用open接口时进行设备的使能。
close	关闭设备。在打开设备时，设备框架中会自动进行打开计数(设备控制块中的ocount数据域)，只有当打开计数为零的时候，底层设备驱动的close接口才会被调用。
read	从设备中读取数据。参数pos指出读取数据的偏移量，但是有些设备并不一定需要制定偏移量，例如串口设备，那么忽略这个参数即可。这个接口返回的类型是rt_size_t即读到的字节数，如果返回零建议检查errno值。如果errno值并不是RT_EOK，那么或者已经读取了所有数据，或者有错误发生。
write	往设备中写入数据。同样pos参数在一些情况下是不必要的，略过即可。
control	根据不同的cmd命令控制设备。命令往往是由底层设备驱动自定义实现的。

11.3.2 设备驱动实现的步骤

上节中比较详细介绍了RT-Thread的设备控制块，那么实现一个设备驱动的步骤是如何的：

1. 实现RT-Thread中定义的设备公共接口，开始可以是空函数(返回类型是rt_err_t的可默认返回RT_EOK)。
2. 根据自己的设备类型定义自己的私有数据域。特别是可以有多个相同设备的情况下，设备接口可以用同一套，不同的只是各自的数据域(例如寄存器基地址)。
3. 按照RT-Thread的对象模型，扩展一个对象有两种方式：
   (a) 定义自己的私有数据结构，然后赋值到RT-Thread设备控制块的private指针上。
   (b) 从struct rt_device结构中进行派生。
4. 根据设备的类型，注册到RT-Thread设备框架中。

Note: 异步设备，通俗的说就是，对设备进行读写并不是采用轮询的方式的，而是采用中断方式。例如接收中断产生代表接收到数据，发送中断产生代表数据已经真实的发送完毕。

11.3.3 AT91SAM7S64串口驱动

做为一个例子，这里仔细分析了AT91SAM7S64的串口驱动，也包括上层应该如何使用这个设备的代码。

AT91SAM7S64串口驱动代码，详细的中文注释已经放在其中了。

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

#include "AT91SAM7X.h"
#include "serial.h"

/* 串口寄存器结构 */
typedef volatile rt_uint32_t REG32;
struct rt_at91serial_hw
{
    REG32    US_CR;     // Control Register
    REG32    US_MR;     // Mode Register
    REG32    US_IER;    // Interrupt Enable Register
    REG32    US_IDR;    // Interrupt Disable Register
    REG32    US_IMR;    // Interrupt Mask Register
    REG32    US_CSR;    // Channel Status Register
    REG32    US_RHR;    // Receiver Holding Register
    REG32    US_THR;    // Transmitter Holding Register
    REG32    US_BRGR;   // Baud Rate Generator Register
    REG32    US_RTOR;   // Receiver Time-out Register
    REG32    US_TTGR;   // Transmitter Time-guard Register
    REG32    Reserved0[5];  //
```

```
    REG32    US_FIDI;   // FI_DI_Ratio Register
    REG32    US_NER;    // Nb Errors Register
    REG32    Reserved1[1];  //
    REG32    US_IF;     // IRDA_FILTER Register
    REG32    Reserved2[44];     //
    REG32    US_RPR;    // Receive Pointer Register
    REG32    US_RCR;    // Receive Counter Register
    REG32    US_TPR;    // Transmit Pointer Register
    REG32    US_TCR;    // Transmit Counter Register
    REG32    US_RNPR;   // Receive Next Pointer Register
    REG32    US_RNCR;   // Receive Next Counter Register
    REG32    US_TNPR;   // Transmit Next Pointer Register
    REG32    US_TNCR;   // Transmit Next Counter Register
    REG32    US_PTCR;   // PDC Transfer Control Register
    REG32    US_PTSR;   // PDC Transfer Status Register
};

/* AT91串口设备 */
struct rt_at91serial
{
    /* 采用从设备基类中继承 */
    struct rt_device parent;

    /* 串口设备的私有数据 */
    struct rt_at91serial_hw* hw_base;   /* 寄存器基地址 */
    rt_uint16_t peripheral_id;          /* 外设ID */
    rt_uint32_t baudrate;               /* 波特率 */

    /* 用于接收的域 */
    rt_uint16_t save_index, read_index;
    rt_uint8_t  rx_buffer[RT_UART_RX_BUFFER_SIZE];
};
/* 串口类的实例化，serial 1/2 */
#ifdef RT_USING_UART1
struct rt_at91serial serial1;
#endif
#ifdef RT_USING_UART2
struct rt_at91serial serial2;
#endif

/* 串口外设的中断服务例程 */
static void rt_hw_serial_isr(int irqno)
{
    rt_base_t level;
    struct rt_device* device;
    struct rt_at91serial* serial = RT_NULL;
```

```
    /* 确定对应的外设对象 */
#ifdef RT_USING_UART1
    if (irqno == AT91C_ID_US0)
    {
        /* serial 1 */
        serial = &serial1;
    }
#endif
#ifdef RT_USING_UART2
    if (irqno == AT91C_ID_US1)
    {
        /* serial 2 */
        serial = &serial2;
    }
#endif
    RT_ASSERT(serial != RT_NULL);

    /* 获得设备基类对象指针 */
    device = (rt_device_t)serial;

    /* 关闭中断以更新接收缓冲 */
    level = rt_hw_interrupt_disable();

    /* 读取一个字符 */
    serial->rx_buffer[serial->save_index] = serial->hw_base->US_RHR;

    /* 把存放索引移到下一个位置 */
    serial->save_index ++;
    if (serial->save_index >= RT_UART_RX_BUFFER_SIZE)
        serial->save_index = 0;

    /* 如果存放索引指向的位置已经到了读索引位置，则丢掉一个数据 */
    if (serial->save_index == serial->read_index)
    {
        serial->read_index ++;
        if (serial->read_index >= RT_UART_RX_BUFFER_SIZE)
            serial->read_index = 0;
    }

    /* 使能中断 */
    rt_hw_interrupt_enable(level);

    /* 调用回调函数指示给上层收到了数据 */
    if (device->rx_indicate != RT_NULL)
        device->rx_indicate(device, 1);
}
```

```
/* 以下的是设备基类的公共接口(虚拟函数) */
static rt_err_t rt_serial_init (rt_device_t dev)
{
    rt_uint32_t bd;
    struct rt_at91serial* serial = (struct rt_at91serial*) dev;

    RT_ASSERT(serial != RT_NULL);
    /* 确认外设标识必需为US0或US1 */
    RT_ASSERT((serial->peripheral_id != AT91C_ID_US0) &&
              (serial->peripheral_id != AT91C_ID_US1));

    /* 使能时钟 */
    AT91C_PMC_PCER = 1 << serial->peripheral_id;

    /* 设置pinmux以使能Rx/Tx数据引脚 */
    if (serial->peripheral_id == AT91C_ID_US0)
    {
        AT91C_PIO_PDR = (1 << 5) | (1 << 6);
    }
    else if (serial->peripheral_id == AT91C_ID_US1)
    {
        AT91C_PIO_PDR = (1 << 21) | (1 << 22);
    }

    /* 重置外设 */
    serial->hw_base->US_CR = AT91C_US_RSTRX      |   /* Reset Receiver      */
                             AT91C_US_RSTTX      |   /* Reset Transmitter   */
                             AT91C_US_RXDIS      |   /* Receiver Disable    */
                             AT91C_US_TXDIS;         /* Transmitter Disable */

    /* 默认都设置为8-N-1 */
    serial->hw_base->US_MR = AT91C_US_USMODE_NORMAL |   /* Normal Mode */
                             AT91C_US_CLKS_CLOCK    |   /* Clock = MCK */
                             AT91C_US_CHRL_8_BITS   |   /* 8-bit Data  */
                             AT91C_US_PAR_NONE      |   /* No Parity   */
                             AT91C_US_NBSTOP_1_BIT;     /* 1 Stop Bit  */

    /* 设置波特率，注：主时钟（MCK）在board.h中定义 */
    bd =  ((MCK*10)/(serial->baudrate * 16));
    if ((bd % 10) >= 5) bd = (bd / 10) + 1;
    else bd /= 10;

    serial->hw_base->US_BRGR = bd;
    serial->hw_base->US_CR = AT91C_US_RXEN |    /* 使能接收 */
                             AT91C_US_TXEN;     /* 使能发送 */

    /* 重置读写索引 */
```

```
    serial->save_index = 0;
    serial->read_index = 0;

    /* 重置接收缓冲 */
    rt_memset(serial->rx_buffer, 0, RT_UART_RX_BUFFER_SIZE);

    return RT_EOK;
}

static rt_err_t rt_serial_open(rt_device_t dev, rt_uint16_t oflag)
{
    struct rt_at91serial *serial = (struct rt_at91serial*)dev;
    RT_ASSERT(serial != RT_NULL);

    /* 如果是中断方式接收，打开中断并装载中断 */
    if (dev->flag & RT_DEVICE_FLAG_INT_RX)
    {
        /* enable UART rx interrupt */
        serial->hw_base->US_IER = 1 << 0;       /* 使能RxReady中断 */
        serial->hw_base->US_IMR |= 1 << 0;      /* 激活RxReady中断 */

        /* 转载UART中断服务例程 */
        rt_hw_interrupt_install(serial->peripheral_id, rt_hw_serial_isr, RT_NULL);
        AT91C_AIC_SMR(serial->peripheral_id) = 5 | (0x01 << 5);
        rt_hw_interrupt_umask(serial->peripheral_id);
    }

    return RT_EOK;
}

static rt_err_t rt_serial_close(rt_device_t dev)
{
    struct rt_at91serial *serial = (struct rt_at91serial*)dev;
    RT_ASSERT(serial != RT_NULL);

    /* 如果是中断方式接收，关闭中断 */
    if (dev->flag & RT_DEVICE_FLAG_INT_RX)
    {
        /* disable interrupt */
        serial->hw_base->US_IDR = 1 << 0;       /* 关闭RxReady中断 */
        serial->hw_base->US_IMR &= ~(1 << 0);   /* 屏蔽RxReady中断 */
    }

    /* 重置外设 */
    serial->hw_base->US_CR = AT91C_US_RSTRX     |   /* Reset Receiver      */
                             AT91C_US_RSTTX     |   /* Reset Transmitter   */
                             AT91C_US_RXDIS     |   /* Receiver Disable    */
```

```
                                AT91C_US_TXDIS;         /* Transmitter Disable */

    return RT_EOK;
}

static rt_size_t rt_serial_read (rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)
{
    rt_uint8_t* ptr;
    struct rt_at91serial *serial = (struct rt_at91serial*)dev;
    RT_ASSERT(serial != RT_NULL);

    /* ptr指向读取的缓冲 */
    ptr = (rt_uint8_t*) buffer;

    if (dev->flag & RT_DEVICE_FLAG_INT_RX)
    {
        /* 中断模式接收 */
        while (size)
        {
            rt_base_t level;

            /* serail->rx_buffer是和ISR共享的，需要关闭中断保护 */
            level = rt_hw_interrupt_disable();
            if (serial->read_index != serial->save_index)
            {
                *ptr = serial->rx_buffer[serial->read_index];

                serial->read_index ++;
                if (serial->read_index >= RT_UART_RX_BUFFER_SIZE)
                    serial->read_index = 0;
            }
            else
            {
                /* rx buffer中无数据 */

                /* 使能中断 */
                rt_hw_interrupt_enable(level);
                break;
            }

            /* 使能中断 */
            rt_hw_interrupt_enable(level);

            ptr ++;
            size --;
        }
```

```
        return (rt_uint32_t)ptr - (rt_uint32_t)buffer;
    }
    else if (dev->flag & RT_DEVICE_FLAG_DMA_RX)
    {
        /* DMA模式接收，目前不支持 */
        RT_ASSERT(0);
    }
    else
    {
        /* 轮询模式 */
        while (size)
        {
            /* 等待数据达到 */
            while (!(serial->hw_base->US_CSR & AT91C_US_RXRDY));

            /* 读取一个数据 */
            *ptr = serial->hw_base->US_RHR;
            ptr ++;
            size --;
        }

        return (rt_size_t)ptr - (rt_size_t)buffer;
    }

    return 0;
}

static rt_size_t rt_serial_write (rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t siz
{
    rt_uint8_t* ptr;
    struct rt_at91serial *serial = (struct rt_at91serial*)dev;
    RT_ASSERT(serial != RT_NULL);

    ptr = (rt_uint8_t*) buffer;
    if (dev->open_flag & RT_DEVICE_OFLAG_WRONLY)
    {
        if (dev->flag & RT_DEVICE_FLAG_STREAM)
        {
            /* STREAM模式发送 */
            while (size)
            {
                /* 遇到'\n'进入STREAM模式， 在前面添加一个'\r' */
                if (*ptr == '\n')
                {
                    while (!(serial->hw_base->US_CSR & AT91C_US_TXRDY));
                    serial->hw_base->US_THR = '\r';
                }
```

```
                /* 等待发送就绪 */
                while (!(serial->hw_base->US_CSR & AT91C_US_TXRDY));

                /* 发送单个字符 */
                serial->hw_base->US_THR = *ptr;
                ptr ++;
                size --;
            }
        }
        else
        {
            while (size)
            {
                /* 等待发送就绪 */
                while (!(serial->hw_base->US_CSR & AT91C_US_TXRDY));

                /* 发送单个字符 */
                serial->hw_base->US_THR = *ptr;
                ptr ++;
                size --;
            }
        }
    }

    return (rt_size_t)ptr - (rt_size_t)buffer;
}

static rt_err_t rt_serial_control (rt_device_t dev, rt_uint8_t cmd, void *args)
{
    return RT_EOK;
}

/* 串口设备硬件初始化，它会根据配置情况进行串口设备注册 */
rt_err_t rt_hw_serial_init()
{
    rt_device_t device;

#ifdef RT_USING_UART1
    device = (rt_device_t) &serial1;

    /* 初始化AT91串口设备私有数据 */
    serial1.hw_base         = (struct rt_at91serial_hw*)AT91C_BASE_US0;
    serial1.peripheral_id   = AT91C_ID_US0;
    serial1.baudrate        = 115200;

    /* 设置设备基类的虚拟函数接口 */
```

```
    device->init    = rt_serial_init;
    device->open    = rt_serial_open;
    device->close   = rt_serial_close;
    device->read    = rt_serial_read;
    device->write   = rt_serial_write;
    device->control = rt_serial_control;

    /* 在设备子系统中注册uart1设备 */
    rt_device_register(device, "uart1", RT_DEVICE_FLAG_RDWR | RT_DEVICE_FLAG_INT_RX);
#endif

#ifdef RT_USING_UART2
    /* 初始化AT91串口设备私有数据 */
    device = (rt_device_t) &serial2;

    serial2.hw_base         = (struct rt_at91serial_hw*)AT91C_BASE_US1;
    serial2.peripheral_id   = AT91C_ID_US1;
    serial2.baudrate        = 115200;

    /* 设置设备基类的虚拟函数接口 */
    device->init    = rt_serial_init;
    device->open    = rt_serial_open;
    device->close   = rt_serial_close;
    device->read    = rt_serial_read;
    device->write   = rt_serial_write;
    device->control = rt_serial_control;

    /* 在设备子系统中注册uart2设备 */
    rt_device_register(device, "uart2", RT_DEVICE_FLAG_RDWR | RT_DEVICE_FLAG_INT_RX);
#endif

    return RT_EOK;
}
```

这个驱动程序中是包含中断发送、接收的情况，所以针对这些，下面给出了具体的使用代码。在这个例子中，线程将在两个设备上(UART1, UART2)读取数据，然后再写到其中的一个设备中。

```
#include <rtthread.h>

/* UART接收消息结构 */
struct rx_msg
{
    rt_device_t dev;
    rt_size_t size;
};
```

```c
/* 用于接收消息的消息队列 */
static rt_mq_t rx_mq;
/* 接收线程的接收缓冲区 */
static char uart_rx_buffer[64];

/* 数据达到回调函数 */
rt_err_t uart_input(rt_device_t dev, rt_size_t size)
{
    struct rx_msg msg;
    msg.dev  = dev;
    msg.size = size;

    /* 发送消息到消息队列中 */
    rt_mq_send(rx_mq, &msg, sizeof(struct rx_msg));

    return RT_EOK;
}

void device_thread_entry(void* parameter)
{
    struct rx_msg msg;
    int count = 0;

    rt_device_t device, write_device;
    rt_err_t result = RT_EOK;

    device = rt_device_find("uart1");
    if (device != RT_NULL)
    {
        /* 设置回调函数及打开设备 */
        rt_device_set_rx_indicate(device, uart_input);
        rt_device_open(device, RT_DEVICE_OFLAG_RDWR);
    }

    /* 设置写设备 */
    write_device = device;
    device = rt_device_find("uart2");
    if (device != RT_NULL)
    {
        /* 设置回调函数及打开设备 */
        rt_device_set_rx_indicate(device, uart_input);
        rt_device_open(device, RT_DEVICE_OFLAG_RDWR);
    }

    while (1)
    {
```

```
        /* 从消息队列中读取消息 */
        result = rt_mq_recv(rx_mq, &msg, sizeof(struct rx_msg), 50);
        if (result == -RT_ETIMEOUT)
        {
            /* 接收超时 */
            rt_kprintf("timeout count:%d\n", ++count);
        }

        /* 成功收到消息 */
        if (result == RT_EOK)
        {
            rt_uint32_t rx_length;

            rx_length = (sizeof(uart_rx_buffer) - 1) > msg.size ?
                msg.size : sizeof(uart_rx_buffer) - 1;

            /* 读取消息 */
            rx_length = rt_device_read(msg.dev, 0, &uart_rx_buffer[0], rx_length);
            uart_rx_buffer[rx_length] = '\0';

            /* 写到写设备中 */
            if (write_device != RT_NULL)
                rt_device_write(write_device, 0, &uart_rx_buffer[0], rx_length);
        }
    }
}

int rt_application_init()
{
    /* 创建devt线程 */
    rt_thread_t thread = rt_thread_create("devt",
        device_thread_entry, RT_NULL,
        1024, 25, 7);

    /* 创建成功则启动线程 */
    if (thread != RT_NULL)
        rt_thread_startup(&thread);
}
```

线程devt启动后，将先查找是否有存在uart1, uart2两个设备，如果存在则设置数据接收到回调函数。在数据接收到的回调函数中，将把对应的设备句柄，接收到的数据长度填充到一个消息结构（struct rx_msg）上，然后发送到消息队列中。devt线程在打开完设备后，将在消息队列中等待消息的到来。如果消息队列是空的，devt线程将被阻塞，直到它接收到消息被唤醒，或在0.5秒(50 OS Tick)内都没收到消息而唤醒。两者唤醒时，从rt_mq_recv函数的返回值中是不相同的。当devt线程因为接收到消息而唤醒时（rt_mq_recv函数的返回值是RT_EOK），它将主动调用rt_device_read去读取消息，然后写

入到write_device设备中。

CHAPTER

TWELVE

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。

12.1 基本数据类型

finsh支持基本的C语言数据类型，包括：

Data Type	Description
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。

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AT91SAM7S64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。

12.2 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：

12.2.1 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

12.2.2 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

12.2.3 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

12.2.4 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

12.2.5 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

12.2.6 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

12.3 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，使用时应包含如下头文件：

```
#include <finsh.h>
```

注：另外一种添加函数及变量的方式，请参看本章选项一节。

12.3.1 添加函数

```
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

12.3.2 添加变量

```
void finsh_sysvar_append(const char* name, u_char type, void* addr)
```

这个接口用于在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。当前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

12.4 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。

12.5 选项

要开启finsh的支持，在RT-Thread的配置中必须定义RT_USING_FINSH宏。

```
/* SECTION: FinSH shell options */
/* Using FinSH as Shell*/
#define RT_USING_FINSH
/* Using symbol table */
#define FINSH_USING_SYMTAB
#define FINSH_USING_DESCRIPTION
```

采用FINSH_USING_SYMTAB将可以支持FINSH_FUNCTION_EXPORT和FINSH_VAR_EXPORT 的方式输出函数或变量到shell中调用。例如：

```
long hello()
{
        rt_kprintf("Hello RT-Thread!\n");

        return 0;
}
FINSH_FUNCTION_EXPORT(hello, say hello world)

static int dummy = 0;
FINSH_VAR_EXPORT(dummy, finsh_type_int, dummy variable for finsh)
```

hello函数、counter变量将自动输出到shell中，既可以在shell中调用、访问hello函数、counter变量：

采用FINSH_USING_DESCRIPTION将可以在list()列出函数、变量列表时显示相应的描述。

CHAPTER

THIRTEEN

文件系统

RT-Thread的文件系统采用了三层的结构，如下图所示：

Figure 13.1: 文件系统结构

- 最顶层的是一套面向嵌入式系统专门优化过的虚拟文件系统（接口），通过它能够适配下层不同的文件系统格式，例如个人电脑上常使用的FAT文件系统，或者是嵌入式设备中常用的flash文件系统。
- 接下来的是各种文件系统的实现，例如支持FAT文件系统的DFS-FAT、DFS-EFSL，支持NandFlash的YAFFS2也即将移植进这套虚拟文件系统框架中。
- 最底层的是各类存储驱动，例如SD卡驱动，IDE硬盘驱动等。

RT-Thread的文件系统对上层提供的接口主要以POSIX标准接口为主，这样这部分代码也容易调试通过。

13.1 文件系统接口

13.1.1 打开文件

打开或创建一个文件可以调用如下接口

```
int open(const char *pathname, int oflag)
```

pathname是要打开或创建的文件名，oflag指出打开文件的选项，当前可以支持：

Name	Description
O_RDONLY	RD只读方式打开
O_WRONLY	WR只写方式打开
O_RDWR	RD-WR读写方式打开
O_CREAT	如果要打开的文件不存在，则建立该文件
O_APPEND	当读写文件时会从文件尾开始移动，也就是所定入的数据会以附加的方式加入到文件后面
O_DIRECTORY	如果参数pathname所指的文件并非为一个目录，则会令打开文件失败

打开成功时返回打开文件的描述符序号，使用open的例子如下

```
/* 假设文件操作是在一个线程中完成 */
void file_thread()
{
   int fd, size;
   char s[] = "RT-Thread Programmer!\n", buffer[80];

   /* 打开 /text.txt 作写入，如果该文件不存在则建立该文件 */
   fd = open("/text.txt", O_WRONLY | O_CREAT);
   if (fd >= 0)
   {
       write(fd, s, sizeof(s));
       close(fd);
   }

   /* 打开 /text.txt 准备作读取动作 */
   fd = open("/text.txt", O_RDONLY);
   if (fd >= 0)
   {
       size=read(fd, buffer, sizeof(buffer));
       close(fd);
   }

   printf("%s", buffer);
}
```

13.1.2 关闭文件

```
int close(int fd)
```

当使用完文件后若已不再需要则可使用close()关闭该文件，而close()会让数据写回磁盘，并释放该文件所占用的资源。参数fd为先前由open()或creat()所返回的文件描述词。

13.1.3 读取数据

```
ssize_t read(int fd, void *buf, size_t count)
```

read()函数会把参数fd所指的文件传送count个字节到buf指针所指的内存中。若参数count为0，则read()不会有作用并返回0。返回值为实际读取到的字节数，如果返回0，表示已到达文件尾或是无可读取的数据，此外文件读写位置随读取到的字节移动。

13.1.4 写入数据

```
size_t write(int fd, const void *buf, size_t count)
```

write()会把参数buf所指的内存写入count个字节到参数fd所指的文件内。当然，文件读写位置也会随之移动。如果顺利write()会返回实际写入的字节数。当有错误发生时则返回-1，错误代码存入当前线程的errno中。

13.1.5 更改名称

```
int rename(const char *oldpath, const char *newpath)
```

rename()会将参数oldpath所指定的文件名称改为参数newpath所指的文件名称。或newpath所指定的文件已经存在，则会被删除。

例子代码如下：

```
void file_thread(void* parameter)
{
    rt_kprintf("%s =>  %s ", "/text1.txt", "/text2.txt");

    if(rename("/text1.txt", "/text2.txt") <0 )
     rt_kprintf("[error!]\n");
    else
     rt_kprintf("[ok!]\n");
}
```

13.1.6 取得状态

```
int  stat(const char *file_name, struct stat *buf)
```

stat()函数用来将参数file_name所指的文件状态，复制到参数buf所指的结构中(struct stat)。

例子如下：

```
void file_thread(void* parameter)
{
      struct stat buf;
      stat("/text.txt", &buf);

      rt_kprintf("text.txt file size = %d\n", buf.st_size);
}
```

13.2 目录操作接口

13.2.1 创建目录

```
int mkdir (const char *path, rt_uint16_t mode)
```

mkdir()函数用来创建一个目录，参数path为目录名，参数mode在当前版本未启用，输入0x777即可，若目录创建成功，返回0，否则返回-1.

例子如下：

```
void file_thread(void* parameter)
{
    int ret;

    /* 创建目录 */
    ret = mkdir("/web", 0x777)
    if(ret < 0)
    {
        /* 创建目录失败 */
        rt_kprintf("[mkdir error!]\n");
    }
    else
    {   /* 创建目录成功 */
        rt_kprintf("[mkdir ok!]\n");
    }
}
```

13.2.2 打开目录

```
DIR* opendir(const char* name)
```

opendir()函数用来打开一个目录，参数为目录路径名，若读取目录成功，返回该目录结构，若读取目录失败，返回RT_NULL。

例子如下：

```
void dir_operation(void* parameter)
{
    int result;
    DIR *dirp;

    /* 创建 /web 目录 */
    dirp = opendir("/web");
    if(dirp == RT_NULL)
    {
        rt_kprintf("[error!]\n");
    }
    else
    {
        /* 在这儿进行读取目录相关操作 */
        /* ..... */
    }
}
```

13.2.3 读取目录

```
struct dirent* readdir(DIR *d)
```

readdir()函数用来读取目录，参数为目录路径名，返回值为读到的目录项结构，如果返回值为RT_NULL，则表示已经读到目录尾。此外，没读取一次目录，目录流的指针位置为自动往后递推1个位置。

例子如下：

```
void dir_operation(void* parameter)
{
    int result;
    DIR *dirp;
    struct dirent *d

    /* 打开 /web 目录 */
    dirp = opendir("/web");
```

```
    if(dirp == RT_NULL)
    {
        rt_kprintf("[error!]\n");
    }
    else
    {
        /* 读取目录 */
        while ((d = readdir(dirp)) != RT_NULL)
        {
            rt_kprintf("found %s\n",d->d_name);
        }
    }
}
```

13.2.4 取得目录流的读取位置

```
rt_off_t telldir(DIR *d)
```

telldir()函数用来取得当前目录流的读取位置。

13.2.5 设置下次读取目录的位置

```
void seekdir(DIR *d, rt_off_t offset)
```

seekdir()函数用来设置下回目录读取的位置。

例子如下：

```
void dir_operation(void* parameter)
{
    DIR * dirp;
    int save3 = 0;
    int cur;
    int i = 0;
    struct dirent *dp;

    dirp = opendir (".");
    for (dp = readdir (dirp); dp != RT_NULL; dp = readdir (dirp))
    {
        /* 保存第三个目录项的目录指针 */
        if (i++ == 3)
        save3 = telldir (dirp);

        rt_kprintf ("%s\n", dp->d_name);
```

```
    }

    /* 回到刚才保存的第三个目录项的目录指针 */
    seekdir (dirp, save3);

    /* 检查当前目录指针是否等于保存过的第三个目录项的指针.  */
    cur = telldir (dirp);
    if (cur != save3)
    {
        rt_kprintf ("seekdir (d, %ld); telldir (d) == %ld\n", save3, cur);
    }

    /* 从第三个目录项开始打印 */
    for (dp = readdir (dirp); dp != NULL; dp = readdir (dirp))
    rt_kprintf ("%s\n", dp->d_name);

    /* 关闭目录 */
    closedir (dirp);
}
```

13.2.6 重设读取目录的位置为开头位置

```
void rewinddir(DIR *d)
```

rewinddir()函数用来设置读取的目录位置为开头位置。

13.2.7 关闭目录

```
int closedir(DIR* d)
```

closedir()函数用来关闭一个目录，如果关闭目录成功返回0，否则返回-1，该函数必须和opendir()函数成对出现。

13.2.8 删除目录

```
int rmdir(const char *pathname)
```

rmdir()函数用来删除一个目录，如果删除目录成功返回0，否则返回-1。

13.3 下层驱动接口

TODO

CHAPTER

FOURTEEN

TCP/IP协议栈

LwIP 是瑞士计算机科学院（Swedish Institute of Computer Science）的Adam Dunkels等开发的一套用于嵌入式系统的开放源代码TCP/IP协议栈，它在包含完整的TCP协议实现基础上实现了小型的资源占用，因此它十分适合于使用到嵌入式设备中，占用的体积大概在几十kB RAM和40KB ROM代码左右。

由于 LwIP 出色的小巧实现，而功能也相对完善，用户群比较广泛，RT-Thread 采用 LwIP 做为默认的 TCP/IP 协议栈，同时根据小型设备的特点对其进行再优化，体积相对进一步减小，**RAM 占用缩小到5kB附近**（依据上层应用使用情况会有浮动）。本章主要讲述了 LwIP 在 RT-Thread 中的使用。

14.1 协议初始化

在使用 LwIP 协议栈之前，需要初始化协议栈。协议栈本身会启动一个 TCP 的线程，和协议相关的处理都会放在这个线程中完成。

```
#include <rtthread.h>

#ifdef RT_USING_LWIP
#include <lwip/sys.h>
#endif

/* 初始化线程入口 */
void rt_init_thread_entry(void *parameter)
{
/* LwIP 初始化 */
#ifdef RT_USING_LWIP
    {
        extern void lwip_sys_init(void);

        /* 初始化LwIP系统 */
        lwip_sys_init();
        rt_kprintf("TCP/IP initialized!\n");
    }
```

```
#endif
}

int rt_application_init()
{
    rt_thread_t init_thread;

    /* 创建初始化线程 */
    init_thread = rt_thread_create("init",
        rt_init_thread_entry, RT_NULL,
        2048, 10, 5);
    /* 启动线程 */
    if (init_thread != RT_NULL) rt_thread_startup(init_thread);

    return 0;
}
```

另外，在RT-Thread中为了使用 LwIP 协议栈需要在 rtconfig.h 头文件中定义使用 LwIP的宏

```
/* 使用 lighweight TCP/IP 协议栈 */
#define RT_USING_LWIP
```

LwIP 协议栈的主线程 TCP 的参数（优先级，信箱大小，栈空间大小）也可以在 rtconfig.h 头文件中定义

```
/* tcp线程选项 */
#define RT_LWIP_TCPTHREAD_PRIORITY      120
#define RT_LWIP_TCPTHREAD_MBOX_SIZE     4
#define RT_LWIP_TCPTHREAD_STACKSIZE     1024
```

默认的 IP 地址，网关地址，子网掩码也可以在 rtconfig.h 头文件中定义（如果要使用 DHCP 方式分配，则需要定义RT_USING_DHCP宏）

```
/* 目标板IP地址 */
#define RT_LWIP_IPADDR0     192
#define RT_LWIP_IPADDR1     168
#define RT_LWIP_IPADDR2     1
#define RT_LWIP_IPADDR3     30

/* 网关地址 */
#define RT_LWIP_GWADDR0     192
#define RT_LWIP_GWADDR1     168
#define RT_LWIP_GWADDR2     1
#define RT_LWIP_GWADDR3     1
```

```
/* 子网掩码 */
#define RT_LWIP_MSKADDR0    255
#define RT_LWIP_MSKADDR1    255
#define RT_LWIP_MSKADDR2    255
#define RT_LWIP_MSKADDR3    0
```

14.2 缓冲区函数

14.2.1 netbuf_new() 原型声明

```
struct netbuf *netbuf_new(void)
```

分配一个 netbuf 结构，该函数并不会分配实际的缓冲区空间，只创建顶层的结构。netbuf 用完后，必须使用 netbuf_delete()回收。

14.2.2 netbuf_delete() 原型声明

```
void netbuf_delete(struct netbuf*)
```

回收先前通过调用 netbuf_new()函数创建的 netbuf 结构，任何通过 netbuf_alloc()函数分配给 netbuf 的缓冲区内存同样也会被回收。

例子：这个例子显示了使用 netbufs 的基本代码结构

```
void lw_thread(void *parameter)
{
    struct netbuf *buf;
    buf = netbuf_new();      /* 建立一个新的netbuf */
    netbuf_alloc(buf, 100); /* 为这个buf分配100 bytes */

    /* 对netbuf数据做一些处理 */
    /* [...] */

    netbuf_delete(buf);      /* 删除buf */
}
```

14.2.3 netbuf_alloc()原型声明

```
void *netbuf_alloc(struct netbuf *buf, int size)
```

为 netbuf buf 分配指定字节（bytes）大小的缓冲区内存。这个函数返回一个指针指向已分配的内存，任何先前已分配给 netbuf buf 的内存会被回收。刚分配的内存可以在以后使用 netbuf_free()函数回收。因为协议头应该要先于数据被发送，所以这个函数即为协议头也为实际的数据分配内存。

14.2.4 netbuf_free() 原型声明

```
int netbuf_free(struct netbuf *buf)
```

回收与 netbuf buf 相关联的缓冲区。如果还没有为 netbuf 分配缓冲区，这个函数不做任何事情。

14.2.5 netbuf_ref() 原型声明

```
int netbuf_ref(struct netbuf *buf, void *data, int size)
```

使数据指针指向的外部存储区与 netbuf buf 关联起来。外部存储区大小由 size 参数给出。任何先前已分配给 netbuf 的存储区会被回收。使用 netbuf_alloc()函数为 netbuf 分配存储区与先分配存储区——比如使用 malloc()函数——然后再使用 netbuf_ref()函数引用这块存储区相比，不同的是前者还要为协议头分配空间这样会使处理和发送缓冲区速度更快。

下面这个例子显示了 netbuf_ref()函数的简单用法

```
void lw_thread(void *parameter)
{
    struct netbuf *buf;
    char string[] = "A string";

    buf = netbuf_new();
    netbuf_ref(buf, string, sizeof(string));    /*  引用这个字符串 */

    /* [...] */

    netbuf_delete(buf);
}
```

14.2.6 netbuf_len() 原型声明

```
int netbuf_len(struct netbuf *buf)
```

返回 netbuf buf 中的数据长度，即使 netbuf 被分割为数据片断。对数据片断状的 netbuf 来说，通过调用这个函数取得的长度值并不等于 netbuf 中的第一个数据片断的长度，而

是所有分片的长度。

14.2.7 netbuf_data() 原型声明

```
int netbuf_data(struct netbuf *buf, void **data, int *len)
```

这个函数用于获取一个指向 netbuf buf 中的数据的指针，同时还取得数据块的长度。参数 data 和 len 为结果参数，参数 data 用于接收指向数据的指针值，len 指针接收数据块长度。如果 netbuf 中的数据被分割为片断，则函数给出的指针指向 netbuf 中的第一个数据片断。应用程序必须使用片断处理函数 netbuf_first()和 netbuf_next()来取得 netbuf 中的完整数据。

关于如何使用 netbuf_data()函数请参阅下面 netbuf_next()函数说明中给出的例子。

14.2.8 netbuf_next() 原型声明

```
int netbuf_next(struct netbuf *buf)
```

函数修改netbuf中数据片断的指针以便指向netbuf中的下一个数据片断。返回值为0表明netbuf中还有数据片断存在，大于0表明指针现在正指向最后一个数据片断，小于0表明已经到了最后一个数据片断的后面的位置，netbuf中已经没有数据片断了。

例子：这个例子显示了如何使用 netbuf_next()函数。我们假定这是一个函数的中间部分，并且其中的 buf 就是一个 netbuf 类型的变量。

```
void lw_thread(void* parameter)
{
    /* [...] */
    do
    {
        char *data;
        int len;

        netbuf_data(buf, &data, &len);  /* 获取一个指针指向数据片段中的数据 */

        do_something(data, len);        /* 对这些数据进行一些处理 */
    } while(netbuf_next(buf) >= 0);
    /* [...] */
}
```

14.2.9 netbuf_first() 原型声明

```
int netbuf_first(struct netbuf *buf)
```

复位netbuf buf中的数据片断指针，使其指向netbuf中的第一个数据片断。

14.2.10 netbuf_copy() 原型声明

```
void netbuf_copy(struct netbuf *buf, void *data, int len)
```

将netbuf buf中的所有数据复制到data指针指向的存储区，即使netbuf buf中的数据被分割为片断。len参数指定要复制数据的最大值。

下面的例子显示了 netbuf_copy()函数的简单用法。这里，协议栈分配 200 个字节的存储区用以保存数据，即使 netbuf buf 中的数据大于 200 个字节，也只会复制 200 个字节的数据。

```
void example_function(struct netbuf *buf)
{
    char data[200];
    netbuf_copy(buf, data, 200);

    /* 对这些数据进行一些处理 */
}
```

14.2.11 netbuf_chain()原型声明

```
void netbuf_chain(struct netbuf *head, struct netbuf *tail)
```

将两个netbufs的首尾链接在一起，以使首部netbuf的最后一个数据片断成为尾部netbuf的第一个数据片断。函数被调用后，尾部netbuf会被回收，不能再使用。

14.2.12 netbuf_fromaddr() 原型声明

```
struct ip_addr *netbuf_fromaddr(struct netbuf *buf)
```

返回接收到的netbuf buf的主机IP地址。如果指定的netbuf还没有从网络收到，函数返回一个未定义值。netbuf_fromport()函数用于取得远程主机的端口号。

14.2.13 netbuf_fromport() 原型声明

```
unsigned short netbuf_fromport(struct netbuf *buf)
```

返回接收到的netbuf buf的主机端口号。如果指定的netbuf还没有从网络收到，函数返回一个不确定值。netbuf_fromaddr()函数用于取得远程主机的IP地址。

14.3 网络连接函数

14.3.1 netconn_new() 原型声明

```
struct netconn *netconn_new(enum netconn_type type)
```

建立一个新的连接数据结构，根据是要建立TCP还是UDP连接来选择参数值是NETCONN_TCP还是NETCONN_UCP。调用这个函数并不会建立连接并且没有数据被发送到网络中。

14.3.2 netconn_delete() 原型声明

```
void netconn_delete(struct netconn *conn)
```

删除连接数据结构conn，如果连接已经打开，调用这个函数将会关闭这个连接。

14.3.3 netconn_type() 原型声明

```
enum netconn_type netconn_type(struct netconn *conn)
```

返回指定的连接conn的连接类型。返回的类型值就是前面netconn_new()函数说明中提到的 NETCONN_TCP或者NETCONN_UDP。

14.3.4 netconn_peer() 原型声明

```
int netconn_peer(struct netconn *conn, struct ip_addr *addr, unsigned short *port)
```

这个函数用于获取连接的远程终端的IP地址和端口号。addr和port为结果参数，它们的值由函数设置。如果指定的连接conn并没有连接任何远程主机，则获得的结果值并不确定。

14.3.5 netconn_addr() 原型声明

```
int netconn_addr(struct netconn *conn, struct ip_addr **addr, unsigned short *port)
```

这个函数用于获取由conn指定的连接的本地IP地址和端口号。

14.3.6 netconn_bind() 原型声明

```
int netconn_bind(struct netconn *conn, struct ip_addr *addr, unsigned short port)
```

为参数conn指定的连接绑定本地IP地址和TCP或UDP端口号。如果addr参数为NULL则本地IP 地址由网络系统确定。

14.3.7 netconn_connect() 原型声明

```
int netconn_connect(struct netconn *conn, struct ip_addr *addr, unsigned short port)
```

对UDP连接，该函数通过addr和port参数设定发送的UDP消息要到达的远程主机的IP地址和端口号。对TCP，netconn_connect()函数打开与指定远程主机的连接。

14.3.8 netconn_listen() 原型声明

```
int netconn_listen(struct netconn *conn)
```

使参数conn指定的连接进入TCP监听（TCP LISTEN）状态。

14.3.9 netconn_accept() 原型声明

```
struct netconn *netconn_accept(struct netconn *conn)
```

阻塞进程直至从远程主机发出的连接请求到达参数conn指定的连接。这个连接必须处于监听（LISTEN）状态，因此在调用netconn_accept()函数之前必须调用netconn_listen()函数。与远程主机的连接建立后，函数返回新连接的结构。

例子：这个例子显示了如何在 2000 端口上打开一个 TCP 服务器。

```
void lw_thread(void* parameter)
{
    struct netconn *conn, *newconn;

    /*  建立一个连接结构 */
```

```
    conn = netconn_new(NETCONN_TCP);

    /* 将连接绑定到一个本地任意IP地址的2000端口上 */
    netconn_bind(conn, NULL, 2000);

    /* 告诉这个连接监听进入的连接请求 */
    netconn_listen(conn);

    /* 阻塞直至得到一个进入的连接 */
    newconn = netconn_accept(conn);

    /* 处理这个连接 */
    process_connection(newconn);

    /* 删除连接 */
    netconn_delete(newconn);
    netconn_delete(conn);
}
```

14.3.10 netconn_recv() 原型声明

```
struct netbuf *netconn_recv(struct netconn *conn)
```

阻塞进程，等待数据到达参数conn指定的连接。如果连接已经被远程主机关闭，则返回NULL，其它情况，函数返回一个包含着接收到的数据的netbuf。

例子：这是一个小例子，显示了 netconn_recv()函数的假定用法。我们假定在调用这个例子函数 example_function()之前连接已经建立。

```
void example_function(struct netconn *conn)
{
    struct netbuf *buf;

    /* 接收数据直到其它主机关闭连接 */
    while((buf = netconn_recv(conn)) != NULL)
    {
                    /* 对这些数据进行一些处理 */
        do_something(buf);
    }

    /* 连接现在已经被其它终端关闭，因此也关闭我们自己的连接 */
    netconn_close(conn);
}
```

14.3.11 netconn_write() 原型声明

```
int netconn_write(struct netconn *conn, void *data, int len, unsigned int flags)
```

这个函数只用于TCP连接。它把data指针指向的数据放在属于conn连接的输出队列。Len参数指定数据的长度，这里对数据长度没有任何限制。这个函数不需要应用程序明确的分配缓冲区（buffers），因为这由协议栈来负责。flags参数有两种可能的状态，如下所示：

```
#define NETCONN_NOCOPY  0x00
#define NETCONN_COPY    0x01
```

当flags值为NETCONN_COPY时，data指针指向的数据被复制到为这些数据分配的内部缓冲区。这就允许这些数据在函数调用后可以直接修改，但是这会在执行时间和内存使用率方面降低效率。如果flags值为NETCONN_NOCOPY，数据不会被复制而是直接使用data指针来引用。这些数据在函数调用后不能被修改，因为这些数据可能会被放在当前指定连接的重发队列，并且会在里面逗留一段不确定的时间。当要发送的数据在ROM中因而数据不可变时这很有用。如果需要更多的控制数据的修改，则可以联合使用复制和不复制数据，如下面的例子所示。

这个例子显示了 netconn_write() 函数的基本用法。这里假定程序里的 data 变量在后面编辑修改，因此它被复制到内部缓冲区，方法是前文所讲的在调用 netconn_write() 函数时将 flags 参数值设为 NETCONN_COPY。text 变量包含了一个不能被编辑修改的字符串，因此它采用指针引用的方式以代替复制。

```
void lw_thread(void* parameter)
{
    struct netconn *conn;
    char data[10];
    char text[] = "Static text";
    int i;

    /*  设置连接conn */
    /* [...] */
    /*  构造一些测试数据 */
    for(i = 0; i < 10; i++)
        data[i] = i;

    netconn_write(conn, data, 10, NETCONN_COPY);
    netconn_write(conn, text, sizeof(text), NETCONN_NOCOPY);

    /*  这些数据可以被修改 */
    for(i = 0; i < 10; i++)
        data[i] = 10 - i;

    /*  关闭连接 */
```

```
    netconn_close(conn);
}
```

14.3.12 netconn_send() 原型声明

```
int netconn_send(struct netconn *conn, struct netbuf *buf)
```

使用参数conn指定的UDP连接发送参数buf中的数据。netbuf中的数据不能太大，因为没有使用IP分段。数据长度不能大于发送网络接口（outgoing network interface）的最大传输单元值（MTU）。因为目前还没有获取这个值的方法，这就需要采用其它的途径来避免超过MTU值，所以规定了一个上限，就是netbuf中包含的数据不能大于1000个字节。函数对要发送的数据大小没有进行校验，无论是非常小还是非常大，因而函数的执行结果是不确定的。

例子：这个例子显示了如何向 IP 地址为 10.0.0.1，UDP 端口号为 7000 的远程主机发送 UDP 数据。

```
void lw_thread(void* parameter)
{
    struct netconn *conn;
    struct netbuf *buf;
    struct ip_addr addr;
    char *data;
    char text[] = "A static text";
    int i;

    /*  建立一个新的连接 */
    conn = netconn_new(NETCONN_UDP);

    /*  设置远程主机的IP地址，执行这个操作后，addr.addr的值为0x0100000a */
    addr.addr = htonl(0x0a000001);

    /*  连接远程主机 */
    netconn_connect(conn, &addr, 7000);

    /*  建立一个新的netbuf    */
    buf = netbuf_new();
    data = netbuf_alloc(buf, 10);

    /*  构造一些测试数据 */
    for(i = 0; i < 10; i++)
        data[i]= i;

    /*  发送构造的数据 */
    netconn_send(conn, buf);
```

```
    /* 引用这个文本给netbuf */
    netbuf_ref(buf, text, sizeof(text));

    /* 发送文本 */
    netconn_send(conn, buf);

    /* 删除conn和buf */
    netconn_delete(conn);
    netconn_delete(buf);
}
```

14.3.13 netconn_close() 原型声明

```
int netconn_close(struct netconn *conn)
```

关闭参数conn指定的连接。

14.3.14 代码示例

这一节介绍了使用LwIP API编写的一个简单的web服务器，代码将在后面给出。这个简单的web服务器仅实现了HTTP/1.0协议的基本功能，显示了如何使用LwIP API实现一个实际地应用。

这个应用由单一线程组成，它负责接收来自网络的连接，响应HTTP请求，以及关闭连接。在这个应用中的线程lw_thread()负责必要的初始化及连接设置工作；连接设置过程是一个相当简单的例子，显示了如何使用最小限度API初始化连接。使用netconn_new()函数建立一个连接后，这个连接被绑定在TCP 80端口并且进入监听（LISTEN）状态，等待连接。一旦一个远程主机连接进来，netconn_accept()函数将返回连接的netconn结构。当这个连接已经被 process_connection() 函数处理后，必须使用 netconn_delete() 函数删除这个netconn。

在process_connection()函数，调用netconn_recv()函数接收一个netbuf，然后通过netbuf_data()函数获取一个指向实际的请求数据指针。这个指针指向netbuf中的第一个数据片断，并且我们希望它包含这个请求。这并不是一个不合实际的想法，因为我们只读取这个请求的前七个字节。如果我们想读取更多的数据，简单的方法是使用netbuf_copy()函数复制这个请求到一个连续的内存区然后在那里处理它。

这个简单的web服务器只响应HTTP GET对文件“/”的请求，并且检测到请求就会发出响应。这里，我们既需要发送针对HTML数据的HTTP头，还要发送HTML数据，所以对netconn_write()函数调用了两次。因为我们不需要修改HTTP头和HTML数据，所以我们将 netconn_write()函数的flags参数值设为NETCONN_NOCOPY以避免复制。

最后，连接被关闭并且process_connection()函数返回。连接结构也会在这个调用后被删除。

下面就是这个应用的C代码：

```
/* 使用最小限度API实现的一个简单的HTTP/1.0服务器 */

#include "api.h"

/*  这是实际的web页面数据。大部分的编译器会将这些数据放在ROM里 */
const static char indexdata[] = "<html> \
    <head><title>A test page</title></head> \
    <body> \
    This is a small test page. \
    </body> \
    </html>";

const static char http_html_hdr[] = "Content-type: text/html\r\n\r\n";

/*  这个函数处理进入的连接 */
static void process_connection(struct netconn *conn)
{
    struct netbuf *inbuf;
    char *rq;
    int len;

    /*  从这个连接读取数据到inbuf，我们假定在这个netbuf中包含完整的请求 */
    inbuf = netconn_recv(conn);

    /*  获取指向netbuf中第一个数据片断的指针，在这个数据片段里我们希望包含这个请求 */
    netbuf_data(inbuf, &rq, &len);

    /*  检查这个请求是不是HTTP "GET /\r\n"  */
    if( rq[0] == 'G' &&
        rq[1] == 'E' &&
        rq[2] == 'T' &&
        rq[3] == ' ' &&
        rq[4] == '/' &&
        rq[5] == '\r' &&
        rq[6] == '\n')
    {
        /*  发送头部数据 */
        netconn_write(conn, http_html_hdr, sizeof(http_html_hdr), NETCONN_NOCOPY);

        /*  发送实际的web页面 */
        netconn_write(conn, indexdata, sizeof(indexdata), NETCONN_NOCOPY);

        /*  关闭连接 */
        netconn_close(conn);
    }
```

```
}

/* 线程入口 */
void lw_thread(void* paramter)
{
    struct netconn *conn, *newconn;

    /* 建立一个新的TCP连接句柄 */
    conn = netconn_new(NETCONN_TCP);

    /* 将连接绑定在任意的本地IP地址的80端口上 */
    netconn_bind(conn, NULL, 80);

    /* 连接进入监听状态 */
    netconn_listen(conn);

    /* 循环处理 */
    while(1)
    {
        /* 接受新的连接请求 */
        newconn = netconn_accept(conn);

        /* 处理进入的连接 */
        process_connection(newconn);

        /* 删除连接句柄 */
        netconn_delete(newconn);
    }

    return 0;
}
```

14.4 BSD Socket库

BSD Socket是在Unix下进行网络通信编程的API接口之一，也是网络编程的事实标准。

LwIP提供了一个轻型BSD Socket API的实现，为大量已有的网络应用程序提供了兼容的接口。LwIP的socket接口实现都在函数名前加有lwip_前缀，同时在头文件中把它采用宏定义的方式定义成标准的BSD Socket API接口。

14.4.1 分配一个socket

```
int lwip_socket(int domain, int type, int protocol);
int socket(int domain, int type, int protocol);
```

应用程序在使用socket前，首先必须拥有一个socket，调用socket()函数可以为应用程序分配一个socket。socket()函数的参数用于指定所需要的socket的类型。

domain参数可以支持如下类型：

```
#define AF_UNSPEC       0
#define AF_INET         2
#define PF_INET         AF_INET
#define PF_UNSPEC       AF_UNSPEC
```

type参数可以支持如下类型：

```
/* Socket protocol types (TCP/UDP/RAW) */
#define SOCK_STREAM     1
#define SOCK_DGRAM      2
#define SOCK_RAW        3
```

protocol参数可以支持如下类型：

```
#define IPPROTO_IP      0
#define IPPROTO_TCP     6
#define IPPROTO_UDP     17
#define IPPROTO_UDPLITE 136
```

14.4.2 绑定socket到地址

```
int lwip_bind(int s, struct sockaddr *name, socklen_t namelen);
int bind(int s, struct sockaddr *name, socklen_t namelen);
```

bind()调用为socket绑定一个本地地址。本地地址由name指定，其长度由namelen指定。sockaddr结构定义如下：

```
struct sockaddr {
  u8_t sa_len;
  u8_t sa_family;
  char sa_data[14];
};
```

14.4.3 建立到远端socket的连接

```
int lwip_connect(int s, const struct sockaddr *name, socklen_t namelen);
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

调用connect()函数连接socket到一个远端地址。调用时需要指定一个远端连接的地址。

14.4.4 接收一个连接

```
int lwip_accept(int s, struct sockaddr *addr, socklen_t *addrlen);
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

accept()函数等待一个连接请求到达指定的TCP socket，而这个socket先前已经通过调用listen()函数进入监听状态。

对accept()函数的调用会根据socket选项情况阻塞线程直至与远程主机建立连接或非阻塞方式返回。这个addr参数是一个结果参数，它的值由accept()函数设置，这个值其实就是远程主机的地址。当新的连接已经建立，LwIP将把远程主机的IP地址和端口号保存到addr参数后，分配一个新的socket标识符，然后accept函数返回这个标识符。

14.4.5 监听连接

```
int lwip_listen(int s, int backlog);
int listen(int s, int backlog);
```

调用listen()函数相当于调用LwIP API函数netconn_listen()，且只能用于TCP连接。与BSD Socket API中listen函数不同的是，BSD Socket允许应用程序指定等待连接队列的大小（backlog参数指定）。LwIP协议栈只支持范围在0 - 255内值的backlog参数。

14.4.6 发送数据

```
int lwip_send(int s, const void *dataptr, int size, unsigned int flags);
int lwip_sendto(int s, const void *dataptr, int size, unsigned int flags,
    struct sockaddr *to, socklen_t tolen);
int lwip_write(int s, const void *dataptr, int size);

int send(int s, const void *dataptr, int size, unsigned int flags);
int sendto(int s, const void *dataptr, int size, unsigned int flags,
    struct sockaddr *to, socklen_t tolen);
int write(int s, const void *dataptr, int size);
```

send()调用用于在参数s指定的已连接的数据报或流socket上发送输出数据，其中参数s为已连接的socket描述符；dataptr指向存有发送数据的缓冲区的指针，其长度由 size 指定。flags支持的参数包括：

```
/* Flags we can use with send and recv. */
#define MSG_PEEK       0x01    /* 查看当前数据       */
#define MSG_WAITALL    0x02    /* 等到所有的信息到达时才返回，不支持 */
#define MSG_OOB        0x04    /* 带外数据，不支持   */
#define MSG_DONTWAIT   0x08    /* 非阻塞模式         */
#define MSG_MORE       0x10    /* 发送更多           */
```

sendto()调用用于将数据由指定的socket传给远方主机。参数to用来指定要传送到的网络地址，结构sockaddr请参考bind()。参数tolen为sockaddr的结果长度。

write()调用用于往参数s指定的已连接的socket中写入数据，其中参数s为已连接的本地socket描述符；dataptr指向存有发送数据的缓冲区的指针，其长度由 size 指定。

14.4.7 接收数据

recv()调用用于在参数s指定的已连接的数据报或流socket上接收输入数据。其中参数s为已连接的socket描述符；mem指向用于保存接收数据的的缓冲区指针，其能够存放的最大长度由 len 指定。flags支持的参数包括：

```
/* Flags we can use with send and recv. */
#define MSG_PEEK       0x01    /* 查看当前数据       */
#define MSG_WAITALL    0x02    /* 等到所有的信息到达时才返回，不支持 */
#define MSG_OOB        0x04    /* 带外数据，不支持   */
#define MSG_DONTWAIT   0x08    /* 非阻塞模式         */
#define MSG_MORE       0x10    /* 发送更多           */
```

recvfrom()调用用于在指定的socket上接收远方主机传递过来的数据。参数from用来指定欲传送的网络地址，结构sockaddr请参考bind()函数。参数fromlen为sockaddr的结构长度。

read()调用用于在参数s指定的已连接的socket中读取数据，其中参数s为已连接的本地socket描述符；mem指向用于保存接收数据的的缓冲区指针，其能够存放的最大长度由 len 指定。

14.4.8 输入/输出多路复用

```
int lwip_select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
                struct timeval *timeout);
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
                struct timeval *timeout);
```

select()调用用来检测一个或多个socket的状态。对每一个socket来说，这个调用可以请求读、写或错误状态方面的信息。请求给定状态的socket集合由一个fd_set结构指示。在返回时，此结构被更新，以反映那些满足特定条件的socket的子集，同时 select() 调用返回满足条件的socket的数目。

Note: select函数在RT-Thread中不能用于文件描述符操作。

14.4.9 关闭socket

```
int lwip_close(int s);
int close(int s);
```

close()关闭socket s，并释放分配给该socket的资源；如果s涉及一个打开的TCP连接，则该连接被释放。

14.4.10 TCP服务器例子

```
#include <rtthread.h>
#include <lwip/sockets.h>

#define MAX_SERV                5           /* 最大chargen服务数目   */
#define CHARGEN_THREAD_NAME     "chargen"   /* chargen线程名称       */
#define CHARGEN_PRIORITY        200         /* 线程优先级            */
#define CHARGEN_THREAD_STACKSIZE 1024

/* 一个chargen客户端控制块结构 */
struct charcb
{
    struct charcb *next;
    int socket;
    struct sockaddr_in cliaddr;
    socklen_t clilen;
    char nextchar;
};

/* 放置chargen客户端的链表 */
static struct charcb *charcb_list = 0;

static int do_read(struct charcb *p_charcb);
static void close_chargen(struct charcb *p_charcb);

/* chargen线程入口 */
static void chargen_thread_entry(void *arg)
{
    int listenfd;
    struct sockaddr_in chargen_saddr;
    fd_set readset;
    fd_set writeset;
    int i, maxfdp1;
    struct charcb *p_charcb;
```

```
/* 创建一个TCP的套接字 */
listenfd = lwip_socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

LWIP_ASSERT("chargen_thread(): Socket create failed.", listenfd >= 0);

/* 对服务器地址做初始化，端口19 */
memset(&chargen_saddr, 0, sizeof(chargen_saddr));
chargen_saddr.sin_family = AF_INET;
chargen_saddr.sin_addr.s_addr = htonl(INADDR_ANY);
chargen_saddr.sin_port = htons(19);

/* 绑定listenfd套接字到服务器地址 */
if (lwip_bind(listenfd, (struct sockaddr *) &chargen_saddr, sizeof(chargen_saddr)) == -1)
    LWIP_ASSERT("chargen_thread(): Socket bind failed.", 0);

/* 让listenfd进入监听模式 */
if (lwip_listen(listenfd, MAX_SERV) == -1)
    LWIP_ASSERT("chargen_thread(): Listen failed.", 0);

/* 用于处理数据的死循环 */
for (;;)
{
    maxfdp1 = listenfd+1;

    /* Determine what sockets need to be in readset */
    FD_ZERO(&readset);
    FD_ZERO(&writeset);
    FD_SET(listenfd, &readset);
    for (p_charcb = charcb_list; p_charcb; p_charcb = p_charcb->next)
    {
        if (maxfdp1 < p_charcb->socket + 1)
            maxfdp1 = p_charcb->socket + 1;
        FD_SET(p_charcb->socket, &readset);
        FD_SET(p_charcb->socket, &writeset);
    }

    /* 等待数据传送或一个新的连接 */
    i = lwip_select(maxfdp1, &readset, &writeset, 0, 0);

    /* 未有新的就绪，继续循环 */
    if (i == 0) continue;

    /* 否则至少有个一个就绪 */
    if (FD_ISSET(listenfd, &readset))
    {
        /* 有新的连接请求，创建一个控制块 */
```

```
        p_charcb = (struct charcb *)rt_calloc(1, sizeof(struct charcb));
        if (p_charcb)
        {
            /* 接受新的连接，此处应是马上返回并会阻塞 */
            p_charcb->socket = lwip_accept(listenfd,
                                    (struct sockaddr *) &p_charcb->cliaddr,
                                    &p_charcb->clilen);
            if (p_charcb->socket < 0)
                /* 如果返回的socket依然小于0，则释放控制块 */
                rt_free(p_charcb);
            else
            {
                /* 加入新的控制块到链表中 */
                p_charcb->next = charcb_list;
                charcb_list = p_charcb;
                p_charcb->nextchar = 0x21;
            }
        }
        else
        {
            /* 分配控制块失败，立马关闭连接 */
            int sock;
            struct sockaddr cliaddr;
            socklen_t clilen;

            sock = lwip_accept(listenfd, &cliaddr, &clilen);
            if (sock >= 0) lwip_close(sock);
        }
    }

    /* 遍历链表并处理数据 */
    for (p_charcb = charcb_list; p_charcb; p_charcb = p_charcb->next)
    {
        if (FD_ISSET(p_charcb->socket, &readset))
        {
            /* 读就绪，执行数据读取 */
            if (do_read(p_charcb) < 0) break;
        }
        if (FD_ISSET(p_charcb->socket, &writeset))
        {
            /* 写就绪 */

            char line[80];
            char setchar = p_charcb->nextchar;

            /* 构造发送数据 */
            for( i = 0; i < 59; i++)
```

```
                {
                    line[i] = setchar;
                    if (++setchar == 0x7f)
                        setchar = 0x21;
                }
                line[i] = 0;
                strcat(line, "\n\r");

                /* 发送到对端 */
                if (lwip_write(p_charcb->socket, line, strlen(line)) < 0)
                {
                    /* 写失败，关闭相应控制块 */
                    close_chargen(p_charcb);
                    break;
                }
                if (++p_charcb->nextchar == 0x7f)
                    p_charcb->nextchar = 0x21;
            }
        }
    }
}

/* 关闭一个chargen连接 */
static void close_chargen(struct charcb *p_charcb)
{
    struct charcb *p_search_charcb;

    /* 关闭套接字 */
    lwip_close(p_charcb->socket);

    /* 从charcb列表中把p_charcb删除 */
    if (charcb_list == p_charcb)
        charcb_list = p_charcb->next;   /* 如果是头节点，直接删除 */
    else
        /* 查找链表以找到它的前一个 */
        for (p_search_charcb = charcb_list; p_search_charcb; p_search_charcb = p_search_charcb->n
        {
            if (p_search_charcb->next == p_charcb)
            {
                /* 把节点从链表中删除 */
                p_search_charcb->next = p_charcb->next;
                break;
            }
        }

    /* 释放p_charcb占用的资源 */
    rt_free(p_charcb);
```

```
}

/* 读操作 */
static int do_read(struct charcb *p_charcb)
{
    char buffer[80];
    int readcount;

    /* 从socket中读取最大80个字节的数据 */
    readcount = lwip_read(p_charcb->socket, &buffer, 80);
    if (readcount <= 0)
    {
        close_chargen(p_charcb);
        return -1;
    }
    return 0;
}

/* chargen初始函数 */
void chargen_init(void)
{
    rt_thread_t chargen;

    /* 创建chargen线程 */
    chargen = rt_thread_create(CHARGEN_THREAD_NAME,
        chargen_thread_entry, RT_NULL,
        CHARGEN_THREAD_STACKSIZE,
        CHARGEN_PRIORITY, 5);
    if (chargen != RT_NULL) rt_thread_startup(chargen);
}
```

CHAPTER

FIFTEEN

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。

15.1 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
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；

15.2 内核配置

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

15.3 手工配置

除了使用python配置工具来配置RT-Thread以外，也能通过修改rtconfig.h的方式来修改系统的配置。下面的是RT-Thread/STM32F103ZE的配置头文件。

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

/* 系统中对象名称大小 */
#define RT_NAME_MAX     8

/* 对齐方式 */
#define RT_ALIGN_SIZE   4

/* 最大支持的优先级：32或256 */
#define RT_THREAD_PRIORITY_MAX  256
```

```
/* 每秒的节拍数 */
#define RT_TICK_PER_SECOND      100

/* SECTION: 调试选项 */
/* 调试 */
#define RT_THREAD_DEBUG

/* 线程栈的溢出检查 */
#define RT_USING_OVERFLOW_CHECK

/* 支持钩子函数 */
#define RT_USING_HOOK

/* SECTION: 线程间通信 */
/* 支持信号量 */
#define RT_USING_SEMAPHORE

/* 支持互斥锁 */
#define RT_USING_MUTEX

/* 支持事件 */
#define RT_USING_EVENT

/* 支持快速事件 */
/* #define RT_USING_FASTEVENT */

/* 支持邮箱 */
#define RT_USING_MAILBOX

/* 支持消息队列 */
#define RT_USING_MESSAGEQUEUE

/* SECTION: 内存管理 */
/* 支持内存池管理 */
#define RT_USING_MEMPOOL

/* 支持动态堆内存管理 */
#define RT_USING_HEAP

/* 使用小型内存模型 */
#define RT_USING_SMALL_MEM

/* 支持SLAB管理器 */
/* #define RT_USING_SLAB */

/* SECTION: 设备IO系统 */
/* 支持设备IO管理系统 */
```

```
#define RT_USING_DEVICE
/* 支持UART1、2、3 */
#define RT_USING_UART1
/* #define RT_USING_UART2 */
/* #define RT_USING_UART3 */

/* SECTION: console选项 */
/* console缓冲长度 */
#define RT_CONSOLEBUF_SIZE      128

/* SECTION: FinSH shell 选项 */
/* 支持finsh作为shell */
#define RT_USING_FINSH
/* 使用符合表 */
#define FINSH_USING_SYMTAB
#define FINSH_USING_DESCRIPTION

/* SECTION: mini libc库 */
/* 使用小型libc库 */
/* #define RT_USING_MINILIBC */

/* SECTION: C++ 选项 */
/* 支持C++ */
/* #define RT_USING_CPLUSPLUS */

/* 支持RTGUI */
/* #define RT_USING_RTGUI */

/* SECTION: 设备虚拟文件系统 */
#define RT_USING_DFS
/* 支持最大的文件系统数目 */
#define DFS_FILESYSTEMS_MAX                     2
/* 最大同时打开文件数             */
#define DFS_FD_MAX                                      8
/* 最大扇区缓冲数目               */
#define DFS_CACHE_MAX_NUM               8

/* SECTION: 轻型TCP/IP协议栈选项 */
/* 支持LwIP协议栈 */
#define RT_USING_LWIP
/* 支持WebServer */
#define RT_USING_WEBSERVER

/* 打开LwIP调试信息 */
/* #define RT_LWIP_DEBUG */

/* 使能ICMP协议 */
```

```
#define RT_LWIP_ICMP

/* 使能IGMP协议 */
/* #define RT_LWIP_IGMP */

/* 使能 UDP 协议/
#define RT_LWIP_UDP

/* 使能 TCP protocol*/
#define RT_LWIP_TCP

/* 同时支持的TCP连接数 */
#define RT_LWIP_TCP_PCB_NUM     5

/* TCP发送缓冲空间 */
#define RT_LWIP_TCP_SND_BUF     1500

/* 使能 SNMP 协议 */
/* #define RT_LWIP_SNMP */

/* 使能 DHCP*/
/* #define RT_LWIP_DHCP */

/* 使能 DNS */
#define RT_LWIP_DNS

/* 本机IP地址 */
#define RT_LWIP_IPADDR0 192
#define RT_LWIP_IPADDR1 168
#define RT_LWIP_IPADDR2 1
#define RT_LWIP_IPADDR3 30

/* 网关地址 */
#define RT_LWIP_GWADDR0 192
#define RT_LWIP_GWADDR1 168
#define RT_LWIP_GWADDR2 1
#define RT_LWIP_GWADDR3 1

/* 本机掩码地址 */
#define RT_LWIP_MSKADDR0        255
#define RT_LWIP_MSKADDR1        255
#define RT_LWIP_MSKADDR2        255
#define RT_LWIP_MSKADDR3        0

/* TCP线程选项 */
#define RT_LWIP_TCPTHREAD_PRIORITY              120
#define RT_LWIP_TCPTHREAD_MBOX_SIZE             4
```

```
#define RT_LWIP_TCPTHREAD_STACKSIZE             1024

/* 以太网线程选项 */
#define RT_LWIP_ETHTHREAD_PRIORITY              128
#define RT_LWIP_ETHTHREAD_MBOX_SIZE             4
#define RT_LWIP_ETHTHREAD_STACKSIZE             512

#endif
```

CHAPTER

SIXTEEN

ARM基本知识

本章及后面的两章讲述的是RT-Thread的ARM移植，分别介绍AT91SAM7S64在GNU GCC环境下和RealView MDK环境下的移植。ATMEL AT91SAM7S64这款芯片采用了ARM7TDMI架构，在进行移植之前很有必要对ARM的编程模式进行详细的了解。

16.1 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，并可在两种状态之间切换：

- 第一种为ARM状态，此时处理器执行32位的字对齐的ARM指令；
- 第二种为Thumb状态，此时处理器执行16位的、半字对齐的Thumb指令。

当ARM微处理器执行32位的ARM指令集时，工作在ARM状态；当ARM微处理器执行16位的Thumb指令集时，工作在Thumb状态。在程序的执行过程中，微处理器可以随时在两种工作状态之间切换，并且，处理器工作状态的转变并不影响处理器的工作模式和相应寄存器中的内容。采用Thumb指令，生成的代码体积相对ARM指令要小，但Thumb指令也有一些局限性，例如它并不标准的程序调用栈，从软件上很难实现栈的回溯等。

在目前RT-Thread对ARM的支持上，RT-Thread只能工作于ARM状态。

16.2 ARM处理器模式

ARM微处理器支持7种运行模式，分别为：

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。

在用户模式下，支持的指令集是受限的，需要使用特权指令时需要通过一定的方式（例如软中断）切换到系统模式进行处理。这对RTOS来说，模式的切换增加了不必要的开销，所以绝大多数RTOS使用了系统模式。同样RT-Thread应用线程选择的是运行于系统模式，这样进行系统函数调用时可不用通过软中断方式陷入到系统模式中。

16.3 ARM的寄存器组织

因为目前RT-Thread还不支持运行Thumb状态，所以本节主要说明ARM状态下的寄存器组织。

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切换时，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。

ARM状态下的通用寄存器与程序计数器

System & User	FIQ	Supervisor	About	IRG	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

ARM状态下的程序状态寄存器

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

= 分组寄存器

16.4 ARM的异常

当正常的程序执行流程发生暂时的停止时，称之为异常，例如处理一个外部的中断请求。在处理异常之前，当前处理器的状态必须保留，这样当异常处理完成之后，当前程序可以继续执行。处理器允许多个异常同时发生，它们将会按固定的优先级进行处理。

当一个异常出现以后，ARM微处理器会执行以下几步操作：

- 将下一条指令的地址存入相应连接寄存器LR，以便程序在处理异常返回时能从正确的位置重新开始执行。若异常是从 ARM状态进入，LR寄存器中保存的是下一条指

令的地址（当前PC+4或PC+8，与异常的类型有关）；若异常是从Thumb状态进入，则在LR寄存器中保存当前PC的偏移量，这样，异常处理程序就不需要确定异常是从何种状态进入的。例如：在软件中断异常SWI，指令 MOV PC，R14_svc总是返回到下一条指令，不管SWI是在ARM状态执行，还是在Thumb状态执行。

- 将CPSR复制到相应的SPSR中。
- 根据异常类型，强制设置CPSR的运行模式位。
- 强制PC从相关的异常向量地址取下一条指令执行，从而跳转到相应的异常处理程序处。

还可以设置中断禁止位，以禁止中断发生。如果异常发生时，处理器处于Thumb状态，则当异常向量地址加载入PC时，处理器自动切换到ARM状态。

ARM微处理器对异常的响应过程用伪码可以描述为：

```
R14_<Exception_Mode> = Return Link

SPSR_<Exception_Mode> = CPSR

CPSR[4:0] = Exception Mode Number

CPSR[5] = 0 ; 当运行于 ARM 工作状态时

If <Exception_Mode> == Reset or FIQ then ; 当响应 FIQ 异常时，禁止新的 FIQ 异常

CPSR[6] = 1

CPSR[7] = 1

PC = Exception Vector Address
```

异常处理完毕之后，ARM微处理器会执行以下几步操作从异常返回：

- 将连接寄存器LR的值减去相应的偏移量后送到PC中。
- 将SPSR复制回CPSR中。
- 若在进入异常处理时设置了中断禁止位，要在此清除。

可以认为应用程序总是从复位异常处理程序开始执行的，因此复位异常处理程序不需要返回。当系统运行时，异常可能会随时发生，为保证在 ARM 处理器发生异常时不至于处于未知状态，在应用程序的设计中，首先要进行异常处理，采用的方式是在异常向量表中的特定位置放置一条跳转指令，跳转到异常处理程序，当 ARM 处理器发生异常时，程序计数器 PC 会被强制设置为对应的异常向量，从而跳转到异常处理程序，当异常处理完成以后，返回到主程序继续执行。

16.5 ARM的IRQ中断处理

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。

16.6 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/计数器通道等。

CHAPTER

SEVENTEEN

GNU GCC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AT91SAM7S核心板为例，描述如何进行RT-Thread的移植。

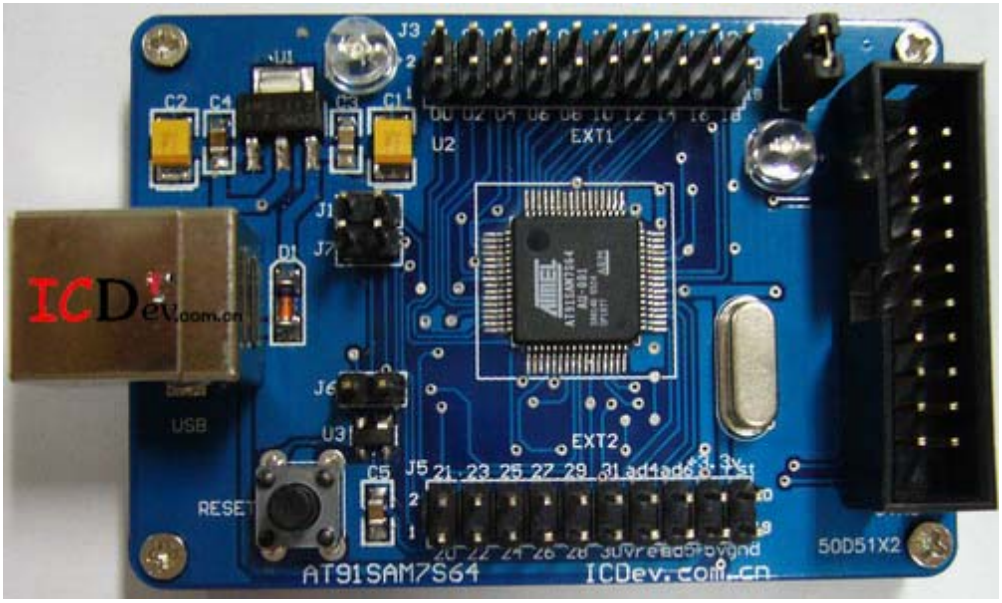


Figure 17.1: AT91SAM7S64核心板(由icdev.net提供)

17.1 CPU相关移植

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），还是自身的实现。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更小一些。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）。

kernel/libcpu/arm/common目录下的文件

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。

17.1.1 上下文切换代码

任务切换代码是移植一个操作系统首先要考虑的，因为它关系到线程间的正常运行，是核心中的核心。

在目录中添加context_gcc.S代码，代码如下

```
/*
 * void rt_hw_context_switch(rt_uint32 from, rt_uint32 to)
 * 上下文切换函数，
 * r0寄存器指向保存当前线程栈位置
 * r1寄存器指向切换到线程的栈位置
 */
.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，如下代码所示。

```
/*
 * 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:
            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C代码：

```
;/*
; * void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);
; */
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;
}
```

除了上下文切换外，也在这个文件中实现了中断（IRQ & FIQ）的关闭和恢复（操作CPSR寄存器屏蔽/使能所有中断）。

```
/*
 * rt_base_t rt_hw_interrupt_disable()
 * 关闭IRQ和FIQ中断，返回关闭中断前的状态
 */
.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 */
    mov pc, 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)
 * 恢复中断状态，中断状态在R0寄存器中
 */
.globl rt_hw_interrupt_enable
rt_hw_interrupt_enable:
    msr cpsr, r0        /* 把R0的值保存到CPSR中 */
    mov pc, lr          /* 函数返回 */
```

17.1.2 系统启动代码

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，另一个最重要的地方就是（各种模式下）栈的设置。下面的代码（文件start_gcc.S）列出了系统启动部分的汇编代码：

```
    /* AT91SAM7S64内部Memory基地址 */
    .equ    FLASH_BASE,     0x00100000
    .equ    RAM_BASE,       0x00200000
```

```
    /* 栈顶及各个栈大小的配置 */
    .equ    TOP_STACK,      0x00204000
    .equ    UND_STACK_SIZE, 0x00000000
    .equ    SVC_STACK_SIZE, 0x00000400
    .equ    ABT_STACK_SIZE, 0x00000000
    .equ    FIQ_STACK_SIZE, 0x00000100
    .equ    IRQ_STACK_SIZE, 0x00000100
    .equ    USR_STACK_SIZE, 0x00000004

    /* ARM模式的定义 */
    .equ    MODE_USR, 0x10
    .equ    MODE_FIQ, 0x11
    .equ    MODE_IRQ, 0x12
    .equ    MODE_SVC, 0x13
    .equ    MODE_ABT, 0x17
    .equ    MODE_UND, 0x1B
    .equ    MODE_SYS, 0x1F

    .equ    I_BIT, 0x80    /* IRQ位 */
    .equ    F_BIT, 0x40    /* FIQ位 */

.section .init, "ax"
.code 32
.align 0
.globl _start
_start:
    b   reset
    ldr pc, _vector_undef
    ldr pc, _vector_swi
    ldr pc, _vector_pabt
    ldr pc, _vector_dabt
    nop                          /* 保留的异常项 */
    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

/*
 * RT-Thread BSS段起始、结束位置，这个在链接脚本中定义
 */
```

```
.globl _bss_start
_bss_start: .word __bss_start
.globl _bss_end
_bss_end:   .word __bss_end

/* 系统入口 */
reset:
    /* 关闭看门狗 */
    ldr r0, =0xFFFFFD40
    ldr r1, =0x00008000
    str r1, [r0, #0x04]

    /* 使能主晶振 */
    ldr r0, =0xFFFFFC00
    ldr r1, =0x00000601
    str r1, [r0, #0x20]

    /* 等待晶振稳定 */
moscs_loop:
    ldr r2, [r0, #0x68]
    ands r2, r2, #1
    beq moscs_loop

    /* 设置PLL */
    ldr r1, =0x00191C05
    str r1, [r0, #0x2C]

    /* 等待PLL上锁 */
pll_loop:
    ldr r2, [r0, #0x68]
    ands r2, r2, #0x04
    beq pll_loop

    /* 选择clock */
    ldr r1, =0x00000007
    str r1, [r0, #0x30]

    /* 设置各个模式下的栈 */
    ldr r0, =TOP_STACK

    /* 设置栈 */
    /* undefined模式 */
    msr cpsr_c, #MODE_UND|I_BIT|F_BIT
    mov sp, r0
    sub r0, r0, #UND_STACK_SIZE

    /* abort模式 */
```

```
    msr cpsr_c, #MODE_ABT|I_BIT|F_BIT
    mov sp, r0
    sub r0, r0, #ABT_STACK_SIZE

    /* FIQ模式 */
    msr cpsr_c, #MODE_FIQ|I_BIT|F_BIT
    mov sp, r0
    sub r0, r0, #FIQ_STACK_SIZE

    /* IRQ模式 */
    msr cpsr_c, #MODE_IRQ|I_BIT|F_BIT
    mov sp, r0
    sub r0, r0, #IRQ_STACK_SIZE

    /* 系统模式 */
    msr cpsr_c, #MODE_SVC
    mov sp, r0

#ifdef __FLASH_BUILD__
    /* 如果是FLASH模式build，从ROM中复制数据段到RAM中 */
    ldr     r1, =_etext
    ldr     r2, =_data
    ldr     r3, =_edata
data_loop:
    cmp     r2, r3
    ldrlo   r0, [r1], #4
    strlo   r0, [r2], #4
    blo     data_loop
#else
    /* 重映射SRAM到零地址 */
    ldr r0, =0xFFFFFF00
    mov r1, #0x01
    str r1, [r0]
#endif

    /* 屏蔽所有IRQ中断 */
    ldr r1, =0xFFFFF124
    ldr r0, =0XFFFFFFFF
    str r0, [r1]

    ; 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RT-Thread Kernel */
    ldr pc, _rtthread_startup

_rtthread_startup: .word rtthread_startup

/* 异常处理 */
vector_undef: b vector_undef
vector_swi  : b vector_swi
vector_pabt : b vector_pabt
vector_dabt : b vector_dabt
vector_resv : b vector_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 */
```

17.1.3 线程初始栈构造

在创建一个线程并把它放到就绪队列，调度器选择了这个线程开始运行时，调度器只知道要切换到这个线程中，它并不知道线程应该从什么地方开始运行，也不知道线程入口处应该放置哪些参数。为了解决这个问题，那么移植就需要"手工地"设置初始栈。添加一个stack.c文件以实现线程栈的初始化工作，代码如下。

```
#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;
}
```

17.1.4 中断处理

当一个中断触发时，从上面初始化代码中可以看到，它的处理流程是这样的：

rt_hw_trap_irq是实际的中断服务例程调用函数，它在trap.c文件中实现。

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

#include "AT91SAM7S.h"

/* 实际的中断处理函数 */
```

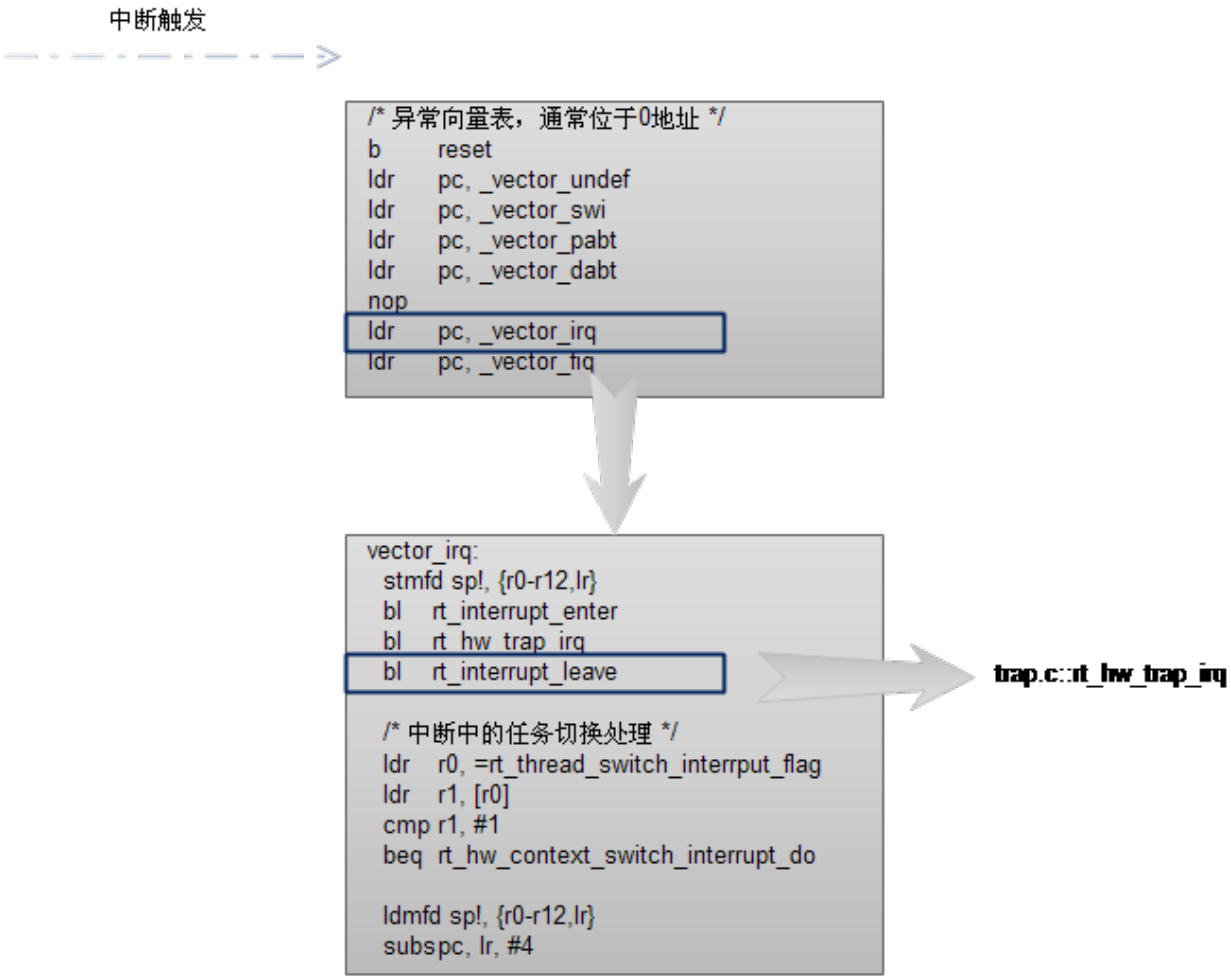


Figure 17.2: 启动文件中的汇编处理

```
void rt_hw_trap_irq()
{
    /* 从IVR寄存器中获得当前设定的中断服务例程函数入口 */
    rt_isr_handler_t hander = (rt_isr_handler_t)AT91C_AIC_IVR;

    /* 调用中断服务例程函数，ISR寄存器指示出是第几号中断 */
    hander(AT91C_AIC_ISR);

    /* 写EOICR寄存器以指示出中断服务结束 */
    AT91C_AIC_EOICR = 0;
}

/* FIQ异常处理函数，目前未使用到 */
void rt_hw_trap_fiq()
{
    rt_kprintf("fast interrupt request\n");
}
```

在rt_hw_trap_irq函数中，它只是负责找到当前产生的中断应该调用哪个中断服务例程，而并没给出如何去设置每个中断所对应的中断服务例程。中断控制器及中断服务例程的设定在interrupt.c中实现。

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

/* 总共32个中断 */
#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_handl
{
    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;
    }
}
```

17.1.5 串口设备驱动

为了能够使用finsh shell，系统中必须实现一个相应的设备，这里实现了一个基本的串口类设备（文件: serial.c），详细的代码解释请参考 *AT91SAM7S64串口驱动* 。

17.2 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AT91SAM7S64核心板中并没特别外扩设备，所以这个目录中只包含了相应的初始化即可。

Tip: 通常SoC芯片已经集成了很多外设，所以这个目录中文件相对比较少，比较简单，而这些SoC芯片的外设驱动则放置在CPU相关的实现中，以方便不同的开发板复用。

17.2.1 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 内核配置_ ，和此移植相关的配置文件代码如下：

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

/* 系统中对象名称大小 */
#define RT_NAME_MAX              4

/* 对齐方式 */
#define RT_ALIGN_SIZE            4

/* 最大支持的优先级：32 */
#define RT_THREAD_PRIORITY_MAX  32

/* 每秒的节拍数 */
#define RT_TICK_PER_SECOND      100

/* SECTION: 调试选项 */
/* 调试 */
/* #define RT_THREAD_DEBUG */

/* 支持线程栈的溢出检查 */
#define RT_USING_OVERFLOW_CHECK

/* 支持钩子函数 */
#define RT_USING_HOOK

/* SECTION: 线程间通信 */
/* 支持信号量 */
#define RT_USING_SEMAPHORE

/* 支持互斥锁 */
```

```
#define RT_USING_MUTEX

/* 支持事件 */
#define RT_USING_EVENT

/* 支持快速事件 */
/* #define RT_USING_FASTEVENT */

/* 支持邮箱 */
#define RT_USING_MAILBOX

/* 支持消息队列 */
#define RT_USING_MESSAGEQUEUE

/* SECTION: 内存管理 */
/* 支持内存池管理 */
#define RT_USING_MEMPOOL

/* 支持动态堆内存管理 */
#define RT_USING_HEAP

/* 使用小型内存模型 */
#define RT_USING_SMALL_MEM

/* 支持SLAB管理器 */
/* #define RT_USING_SLAB */

/* SECTION: 设备IO系统 */
/* 支持设备IO管理系统 */
#define RT_USING_DEVICE
/* 支持UART1、2、3 */
#define RT_USING_UART1
/* #define RT_USING_UART2 */
/* #define RT_USING_UART3 */

/* SECTION: console选项 */
/* console缓冲长度 */
#define RT_CONSOLEBUF_SIZE  128

/* SECTION: FinSH shell 选项 */
/* 支持finsh作为shell */
#define RT_USING_FINSH
/* 使用符合表 */
#define FINSH_USING_SYMTAB
#define FINSH_USING_DESCRIPTION

/* SECTION: mini libc库 */
```

```
/* 使用小型libc库 */
#define RT_USING_MINILIBC

/* SECTION: C++ 选项 */
/* 支持C++ */
/* #define RT_USING_CPLUSPLUS */

#endif
```

由于采用了GNU GCC作为编译器，并没有编译代码的限制，所以这里打开了finsh shell组件、C++支持组件。另外由于GNU GCC并不存在C库，所以在配置文件中使能了小型C库支持。

17.2.2 Kernel启动

做为系统启动汇编后进入的第一个C函数，startup.c文件实现了相应的rtthread_startup函数（针对RealView MDK，则实现了main函数）。

```
/* 为了获得heap的起始地址，把bss段的末地址通过链接器定义的方式给出 */
#ifdef __CC_ARM
extern int Image$$RW_IRAM1$$ZI$$Limit;
#endif

#ifdef __GNUC__
extern unsigned char __bss_start;
extern unsigned char __bss_end;
#endif

extern void rt_hw_interrupt_init(void);
extern int  rt_application_init(void);
#ifdef RT_USING_DEVICE
extern rt_err_t rt_hw_serial_init(void);
#endif

/**
 * RT-Thread启动函数
 */
void rtthread_startup(void)
{
    /* 初始化中断 */
    rt_hw_interrupt_init();

    /* 初始化开发板硬件 */
    rt_hw_board_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
#ifdef __CC_ARM
    rt_system_heap_init((void*)&Image$$RW_IRAM1$$ZI$$Limit, (void*)0x204000);
#elif __ICCARM__
    rt_system_heap_init(__segment_end("HEAP"), (void*)0x204000);
#else
    rt_system_heap_init((void*)&__bss_end, (void*)0x204000);
#endif
#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

#ifdef RT_USING_DEVICE
    /* init hardware serial device */
    rt_hw_serial_init();
    /* init all device */
    rt_device_init_all();
#endif

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

#ifdef RT_USING_FINSH
    /* init finsh */
    finsh_system_init();
    finsh_set_device("uart1");
#endif
```

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

    /* 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 ;
}

int main (void)
{
    /* 在main函数中调用rtthread_startup函数 */
    rtthread_startup();

    return 0;
}
```

17.2.3 开发板初始化

和开发板硬件相关的实现放于board.c文件中，代码及注释如下：

```
/* Periodic Interval Value */
#define PIV   (((MCK/16)/1000)*(1000/RT_TICK_PER_SECOND))

/* OS时钟定时器中断服务例程，这个是移植中必须添加的部分 */
void rt_hw_timer_handler(int vector)
{
    if (AT91C_PITC_PISR & 0x01)
    {
        /* 定时器中断到了，调用rt_tick_increase通知OS一个时钟节拍达到 */
        rt_tick_increase();

        /* 确认中断结束 */
        AT91C_AIC_EOICR = AT91C_PITC_PIVR;
    }
    else
    {
        /* 确认中断结束 */
        AT91C_AIC_EOICR = 0;
    }
}

/* AT91SAM7S64核心板上PIO8连接了一个LED灯 */
                        /* PIO    Flash     PA     PB    PIN */
```

```
#define LED     (1 << 8)/* PA8          & TWD  NPCS3  43 */

/* 开发板中LED的初始化化 */
static void rt_hw_board_led_init()
{
    /* 使能PIO的时钟 */
    AT91C_PMC_PCER = 1 << AT91C_ID_PIOA;

    /* 配置PIO8为输出 */
    AT91C_PIO_PER = LED;
    AT91C_PIO_OER = LED;
}

/* 点亮LED灯 */
void rt_hw_board_led_on()
{
    AT91C_PIO_CODR = LED;
}

/* 熄灭LED灯 */
void rt_hw_board_led_off()
{
    AT91C_PIO_SODR = LED;
}

/* 采用循环延时的方式对LED进行闪烁 */
void rt_hw_led_flash()
{
    int i;

    rt_hw_board_led_off();
    for (i = 0; i < 2000000; i ++);

    rt_hw_board_led_on();
    for (i = 0; i < 2000000; i ++);
}

/*
 * RT-Thread Console 接口，由rt_kprintf使用
 */
/* 在console上显示一段字符串 str，它不应触发一个硬件中断 */
void rt_hw_console_output(const char* str)
{
    while (*str)
    {
        /* 如果是'\n'，在前面插入一个'\r'。 */
        if (*str == '\n')
```

```
        {
            while (!(AT91C_US0_CSR & AT91C_US_TXRDY));
            AT91C_US0_THR = '\r';
        }

        /* 等待发送空 */
        while (!(AT91C_US0_CSR & AT91C_US_TXRDY));

        /* 发送一个字符 */
        AT91C_US0_THR = *str;

        /* 移动显示字符串指针到下一个位置 */
        str ++;
    }
}

/* console的初始化函数 */
static void rt_hw_console_init()
{
    /* 使能USART0时钟 */
    AT91C_PMC_PCER = 1 << AT91C_ID_US0;
    /* 设置相应的接收、发送Pin脚 */
    AT91C_PIO_PDR = (1 << 5) | (1 << 6);

    /* 先重置控制器 */
    AT91C_US0_CR = AT91C_US_RSTRX    |       /* Reset Receiver      */
                   AT91C_US_RSTTX    |       /* Reset Transmitter   */
                   AT91C_US_RXDIS    |       /* Receiver Disable    */
                   AT91C_US_TXDIS;           /* Transmitter Disable */

    /* 初始化控制器 */
    AT91C_US0_MR = AT91C_US_USMODE_NORMAL |     /* Normal Mode */
                   AT91C_US_CLKS_CLOCK    |     /* Clock = MCK */
                   AT91C_US_CHRL_8_BITS   |     /* 8-bit Data  */
                   AT91C_US_PAR_NONE      |     /* No Parity   */
                   AT91C_US_NBSTOP_1_BIT;       /* 1 Stop Bit  */

    /* 设置波特率 */
    AT91C_US0_BRGR = BRD;

    /* 使能接收和发送 */
    AT91C_US0_CR = AT91C_US_RXEN |  /* Receiver Enable     */
                   AT91C_US_TXEN;   /* Transmitter Enable  */
}

/* AT91SAM7S64核心板初始化 */
void rt_hw_board_init()
```

```
{
    /* 初始化console，在使用rt_kprintf前，必须先要初始化console */
    rt_hw_console_init();

    /* 初始化LED */
    rt_hw_board_led_init();

    /* 初始化PITC */
    AT91C_PITC_PIMR = (1 << 25) | (1 << 24) | PIV;
    /* 装置OS定时器中断服务例程 */
    rt_hw_interrupt_install(AT91C_ID_SYS, rt_hw_timer_handler, RT_NULL);
    AT91C_AIC_SMR(AT91C_ID_SYS) = 0;
    rt_hw_interrupt_umask(AT91C_ID_SYS);
}
```

17.2.4 用户初始化文件

作为基本的移植，只需要一个空的rt_application_init函数实现即可。

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

17.2.5 链接脚本

由于AT91SAM7S64只包含较小的内存，所以才有直接在片内flash中运行的方式，在文件sam7s_rom.lds中实现。

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
MEMORY
{
    CODE (rx) : ORIGIN = 0x00000000, LENGTH = 0x00010000
    DATA (rw) : ORIGIN = 0x00200000, LENGTH = 0x00004000
}
ENTRY(_start)
SECTIONS
{
    .text :
    {
        *(.init)
```

```
        *(.text)
    } > CODE = 0

    . = ALIGN(4);
    .rodata :
    {
        *(.rodata .rodata.*)
    } > CODE

    _etext = . ;
    PROVIDE (etext = .);

    /* .data section which is used for initialized data */

    .data : AT (_etext)
    {
        _data = . ;
        *(.data)
        SORT(CONSTRUCTORS)
    } >DATA
    . = ALIGN(4);

    _edata = . ;
    PROVIDE (edata = .);

    . = ALIGN(4);
    __bss_start = .;
    .bss :
    {
        *(.bss)
    } > DATA
    __bss_end = .;

    _end = .;
}
```

通过这个链接脚本文件，主要生成了几个section：

*. .text，从0x00000000开始，放置可执行代码部分。 *. .rodata，紧接着.text后面放置，其中包含了只读数据；并在后面插入了_etext符合，指向结束地址。 *. .data，在映像文件中放置于_etext位置，其中包含了可读写的数据，但在运行状态下则从0x00200000地址开始。 *. .bss，紧接着.data放置，并在开始位置及结束位置放置了__bss_start和__bss_end以指向相应的位置。

CHAPTER

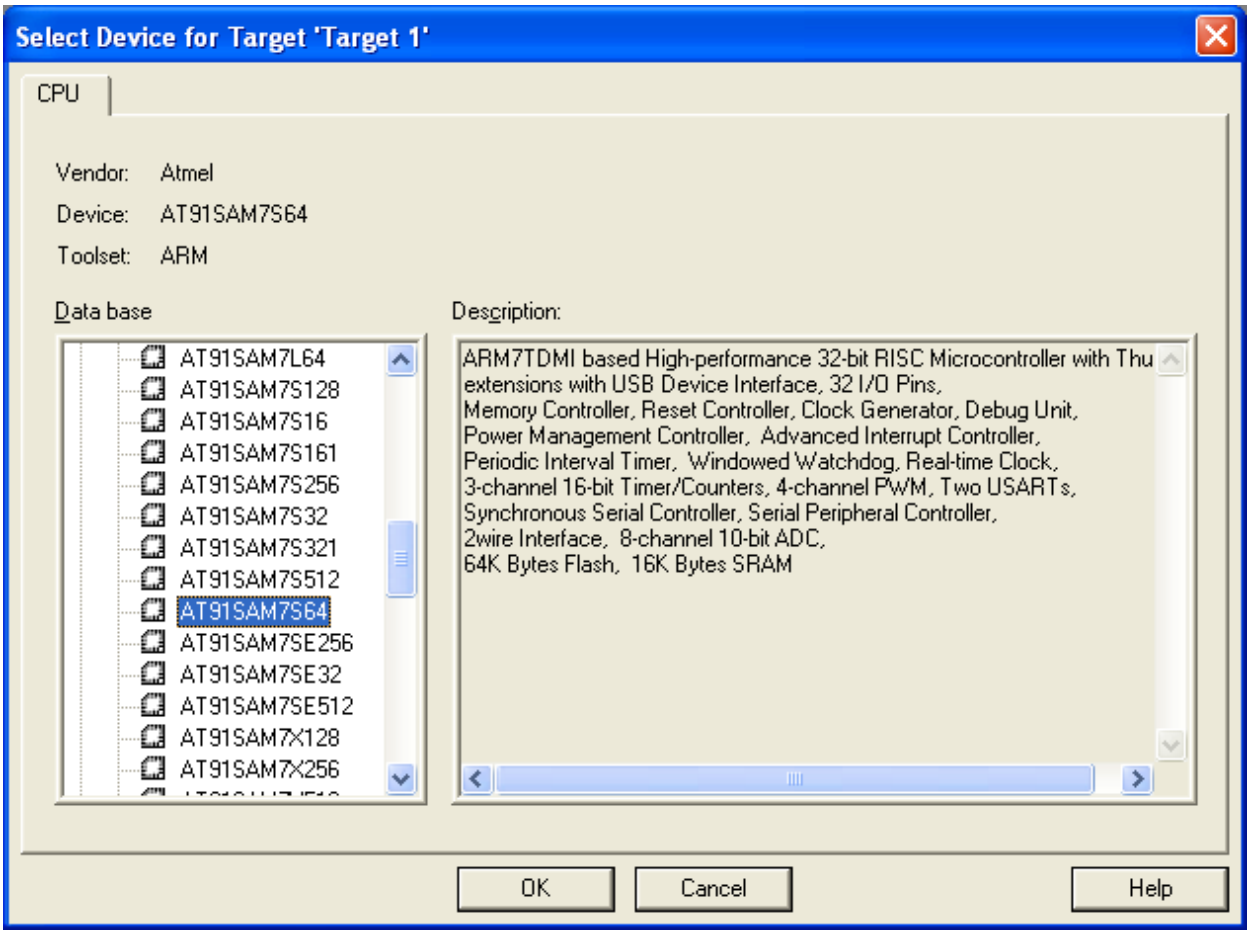
EIGHTEEN

REALVIEW MDK移植

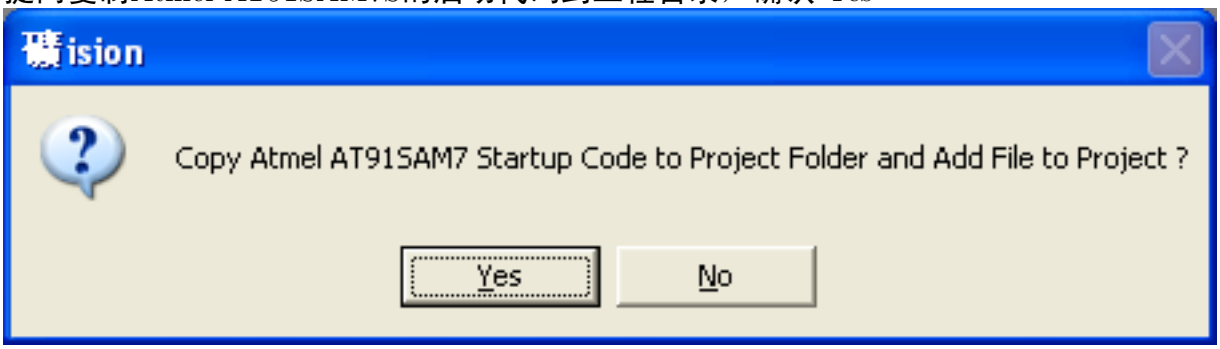
本节用到的RealView MDK版本是3.50评估版，因为生成的代码小于<16k，可以正常编译调试。（由于RealView MDK评估版和RealView MDK专业版的差异，专业版会生成更小的代码尺寸，推荐使用专业版）

18.1 建立RealView MDK工程

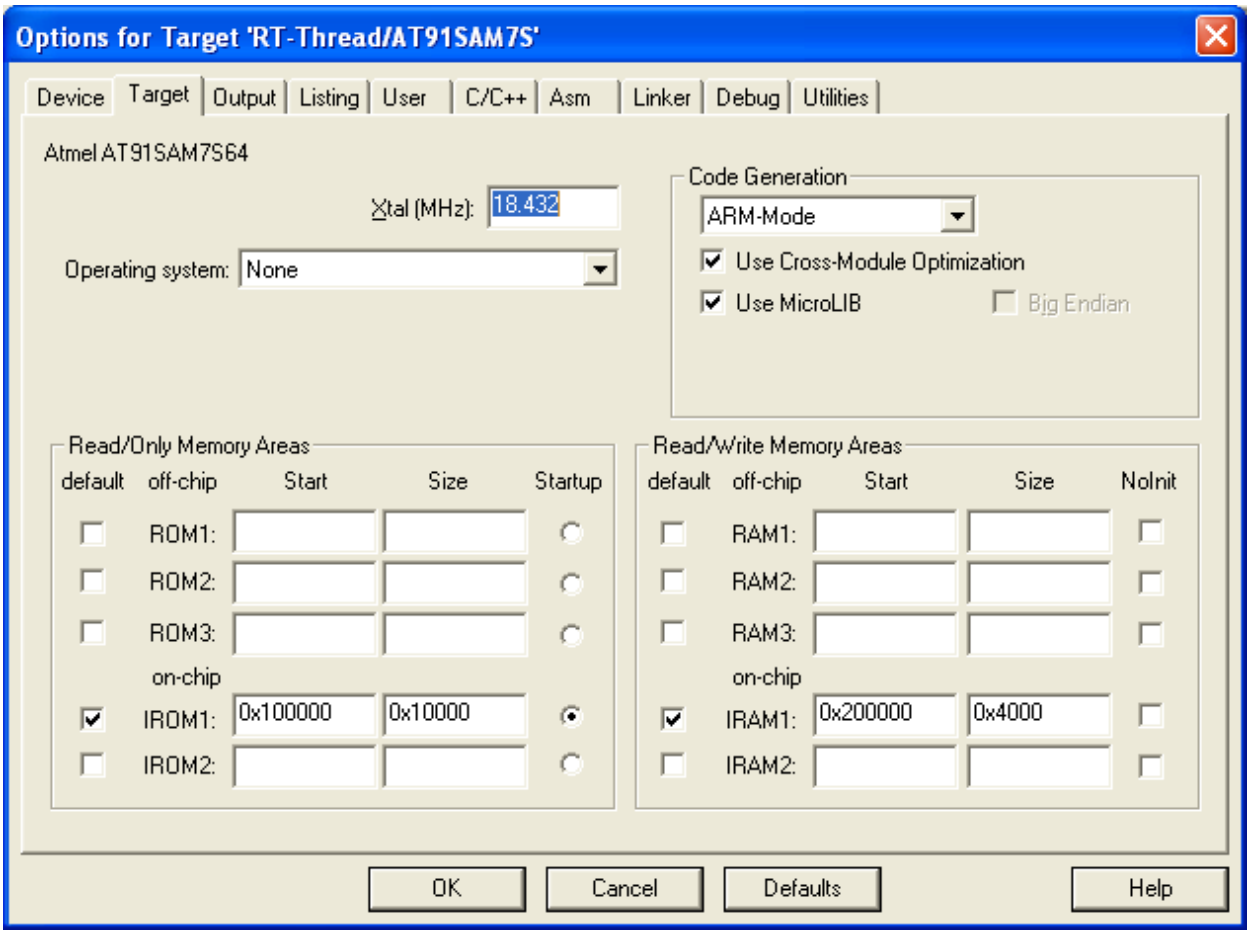
在kernel/bsp目录下新建sam7s目录。在RealView MDK中新建立一个工程文件（用菜单创建），名称为project，保存在kernel/bsp/sam7s目录下。创建工程时一次的选项如下：CPU选择Atmel的AT91SAM7S64



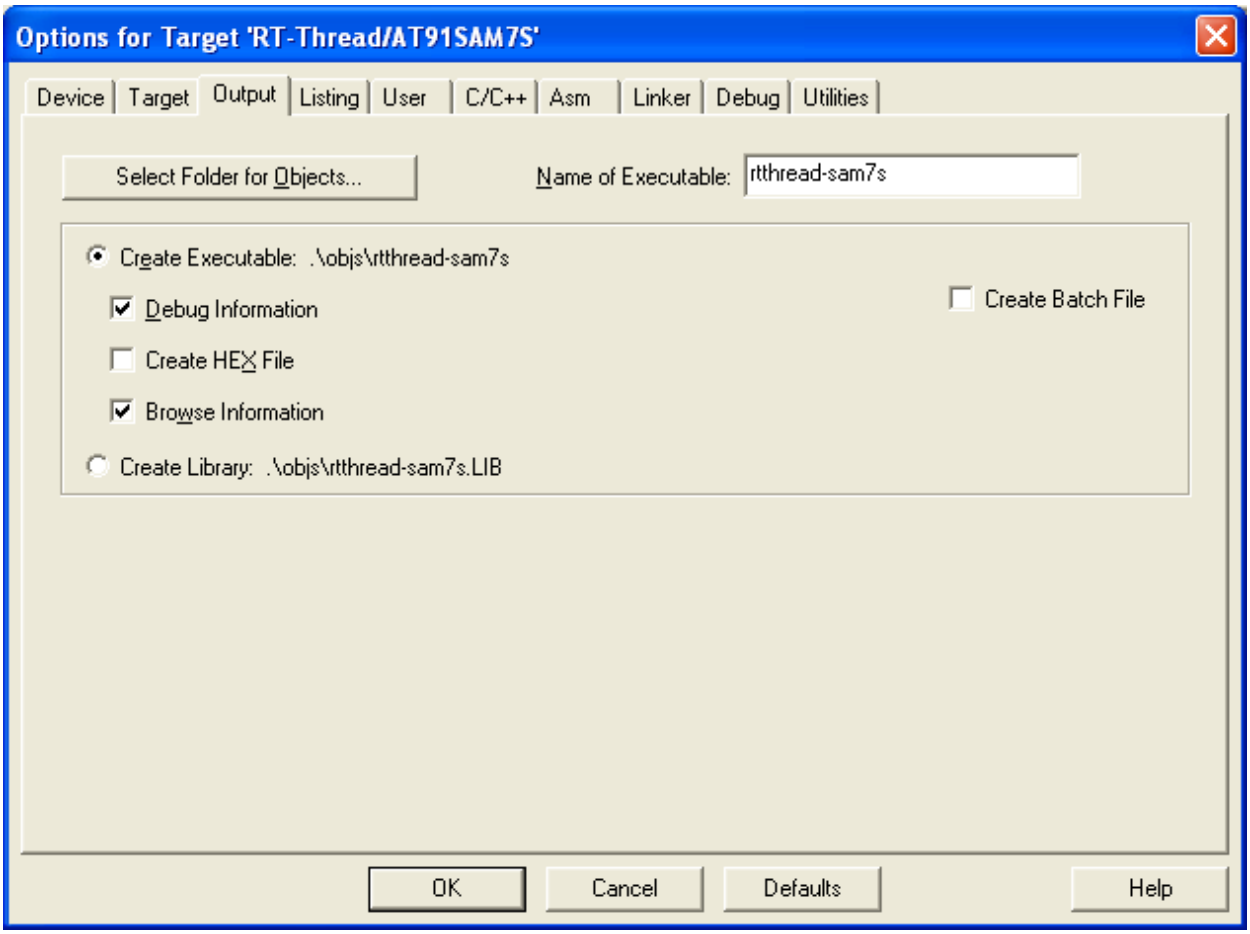
提问复制Atmel AT91SAM7S的启动代码到工程目录，确认 Yes



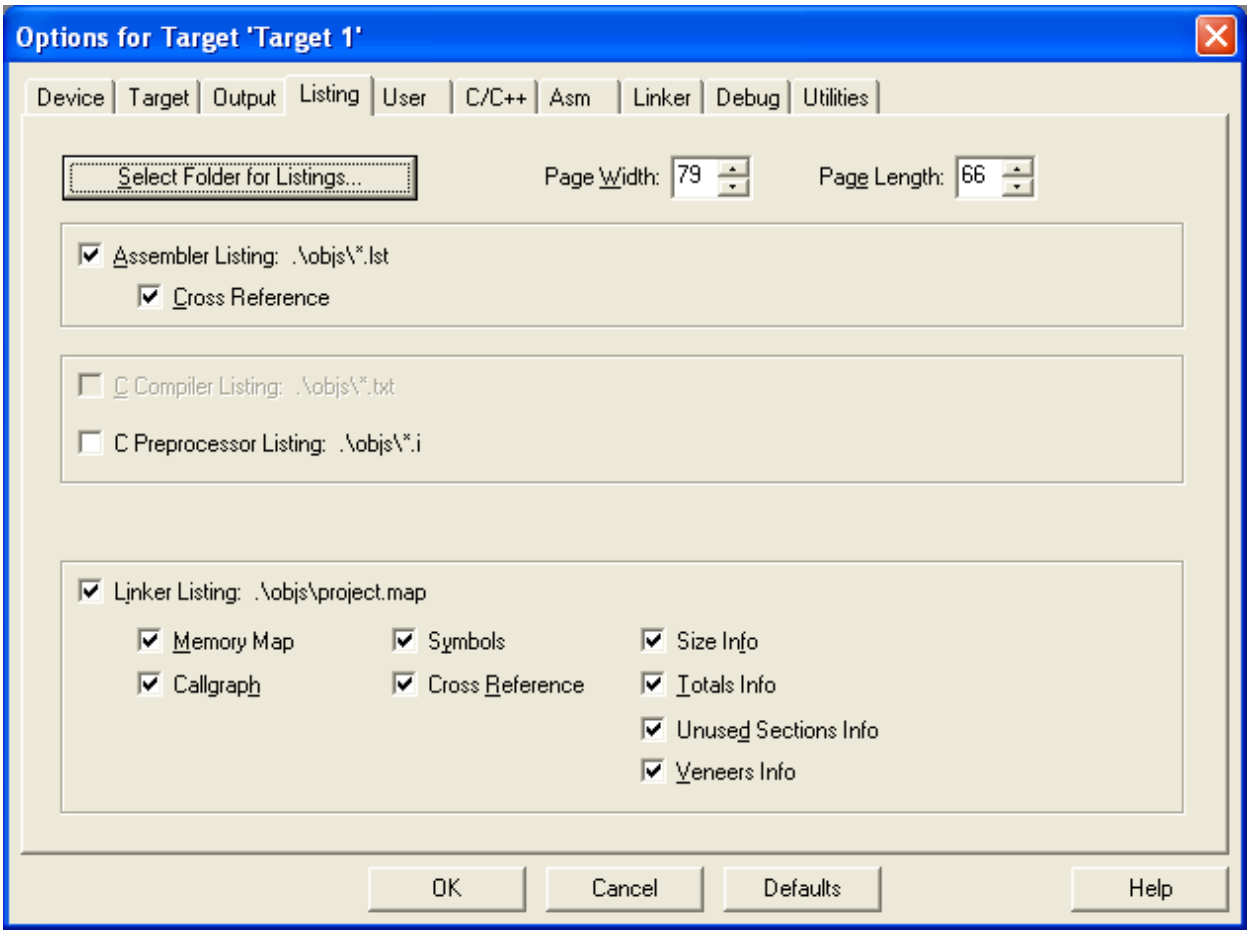
然后选择工程的属性，Code Generation选择ARM-Mode，如果希望产生更小的代码选择Use Cross-Module Optimization和Use MicroLIB，如下图



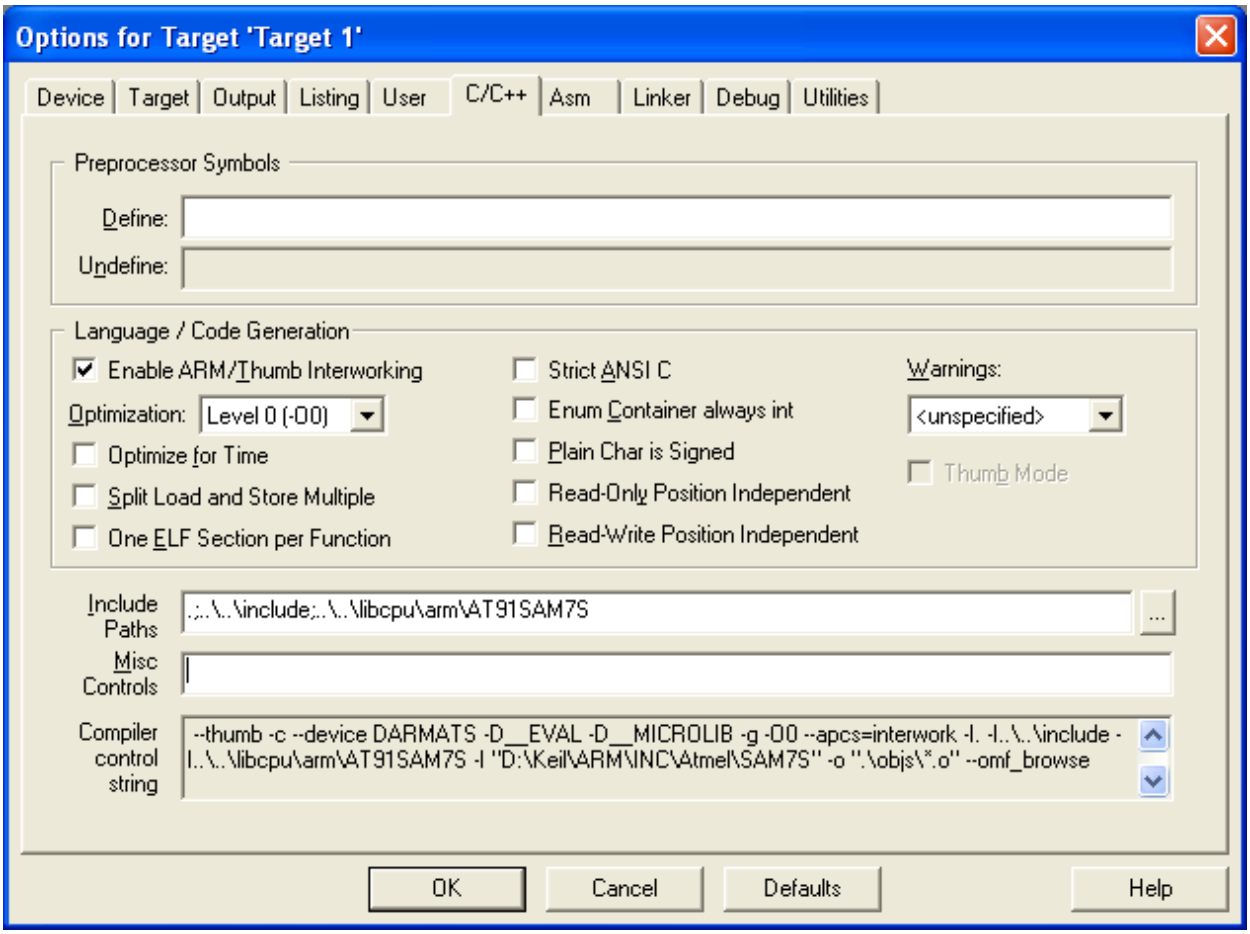
Select Folder for Objects目 录 选 择 到kernel/bsp/sam7s/objs， Name of Executable为rtthread-sam7s。



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



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kernel\include，kernel\libcpu\arm\AT91SAM7S以及kernel\bsp\sam7s目录，如下图所示：



Asm，Linker，Debug和Utilities选项使用初始配置。

18.2 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重命名为start_rvds.s并放到kernellibcpuAT91SAM7S目录中。

Kernel Group中添加所有kernel\src下的C源文件； Startup Group中添加startup.c，board.c文件（放于kernel\bsp\sam7s目录中）； AT91SAM7S Group中添加context_rvds.s，stack.c，trap.c，interrupt.c等文件（放于kernellibcpusam7s目录中）；

在kernel/bsp/sam7s目录中添加rtconfig.h文件，内容如下：

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

/* 系统中对象名称大小 */
#define RT_NAME_MAX             4
```

```
/* 对齐方式 */
#define RT_ALIGN_SIZE           4

/* 最大支持的优先级：32 */
#define RT_THREAD_PRIORITY_MAX  32

/* 每秒的节拍数 */
#define RT_TICK_PER_SECOND      100

/* SECTION: 调试选项 */
/* 调试 */
/* #define RT_THREAD_DEBUG */

/* 支持线程栈的溢出检查 */
#define RT_USING_OVERFLOW_CHECK

/* 支持钩子函数 */
#define RT_USING_HOOK

/* SECTION: 线程间通信 */
/* 支持信号量 */
#define RT_USING_SEMAPHORE

/* 支持互斥锁 */
#define RT_USING_MUTEX

/* 支持事件 */
#define RT_USING_EVENT

/* 支持快速事件 */
/* #define RT_USING_FASTEVENT */

/* 支持邮箱 */
#define RT_USING_MAILBOX

/* 支持消息队列 */
#define RT_USING_MESSAGEQUEUE

/* SECTION: 内存管理 */
/* 支持内存池管理 */
#define RT_USING_MEMPOOL

/* 支持动态堆内存管理 */
#define RT_USING_HEAP

/* 使用小型内存模型 */
```

```
#define RT_USING_SMALL_MEM

/* 支持SLAB管理器 */
/* #define RT_USING_SLAB */

/* SECTION: 设备IO系统 */
/* 支持设备IO管理系统 */
#define RT_USING_DEVICE
/* 支持UART1、2、3 */
#define RT_USING_UART1
/* #define RT_USING_UART2 */
/* #define RT_USING_UART3 */

/* SECTION: console选项 */
/* console缓冲长度 */
#define RT_CONSOLEBUF_SIZE  128

/* SECTION: FinSH shell 选项 */
/* 支持finsh作为shell */
/* #define RT_USING_FINSH */
/* 使用符合表 */
#define FINSH_USING_SYMTAB
#define FINSH_USING_DESCRIPTION

/* SECTION: mini libc库 */
/* 使用小型libc库 */
/* #define RT_USING_MINILIBC */

/* SECTION: C++ 选项 */
/* 支持C++ */
/* #define RT_USING_CPLUSPLUS */

#endif
```

由于采用的是RealView MDK评估版本，加入finsh shell将超出代码限制，所以此处把RT_USING_FINSH的宏定义移除了。

18.3 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
```

18.4 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                     ; 清除中断中切换标志
                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
```

18.5 中断处理

中断处理部分和GNU GCC中的移植是相同的，详细请参见:ref: *AT91SAM7S64-interrupt-c*

18.6 开发板初始化

此部分代码也和GNU GCC中断移植相同，详细请参见:ref: *AT91SAM7S64-board-c*

CHAPTER

NINETEEN

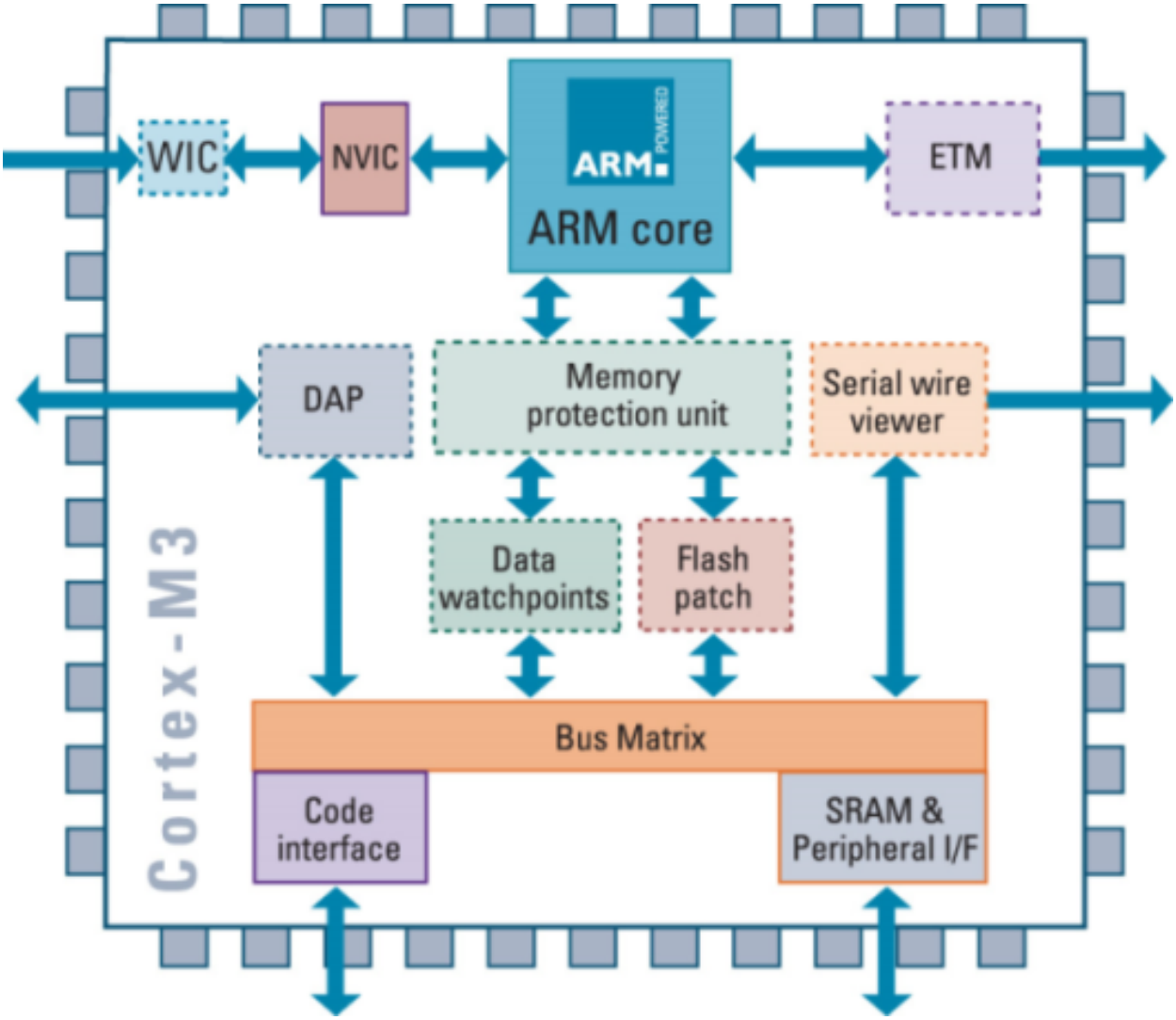
RT-THREAD/STM32说明

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。

19.1 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。

19.2 ARM Cortex M3移植要点

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

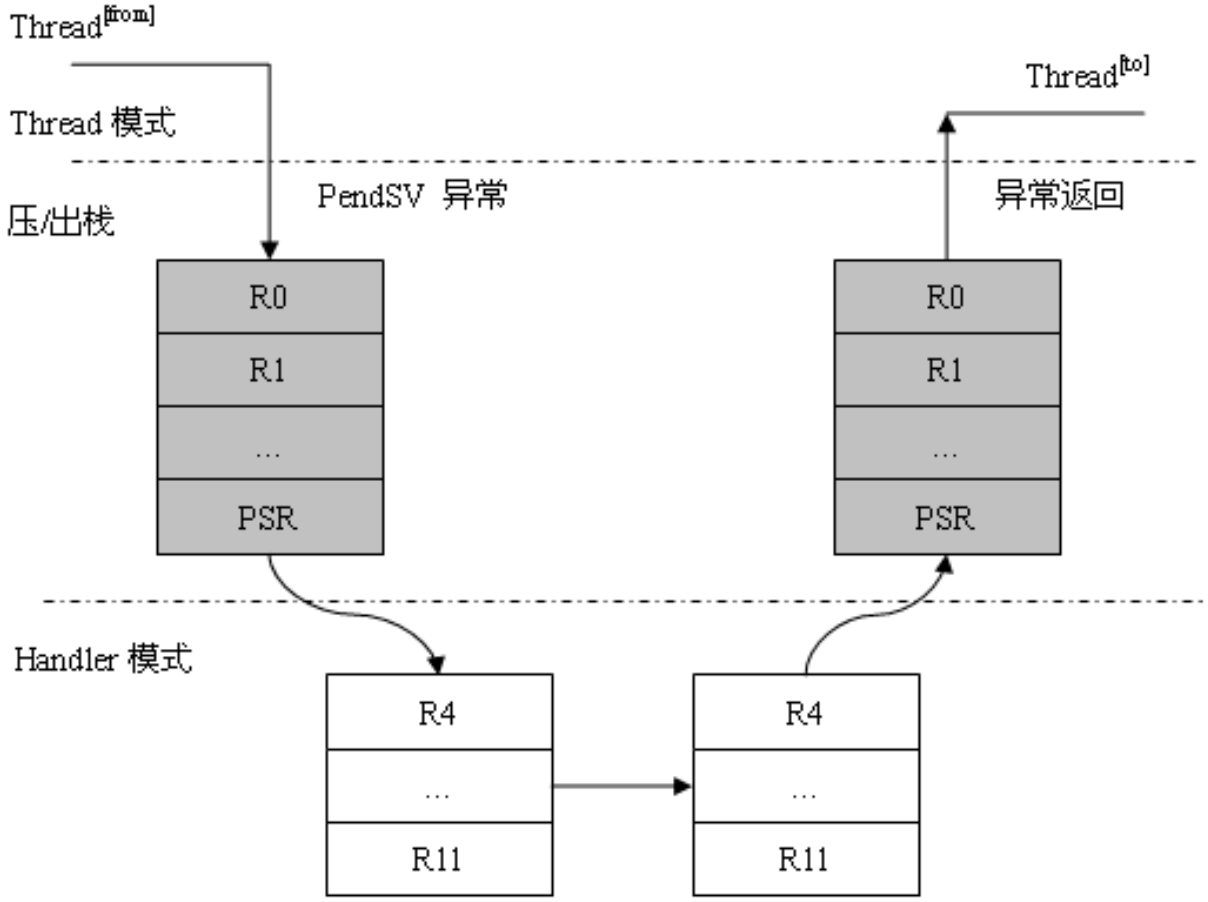
通常的ARM移植，RT-Thread需要手动的保存当前模式下几乎所有寄存器，R0 – R13，LR，PC，CPSR，SPSR等。在Cortex M3微处理器中，代码 D - 1 线程切换代码

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

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

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

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

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

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

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

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

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

; 在异常处理过程中发生线程切换时的上下文处理函数
rt_hw_context_switch_interrupt    PROC
        EXPORT rt_hw_context_switch_interrupt
        LDR     r2, =rt_thread_switch_interrput_flag
        LDR     r3, [r2]
        CMP     r3, #1
        BEQ     _reswitch                                       ; 中断中切换标识已置位，则跳到_re
        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

        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

        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

        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
```

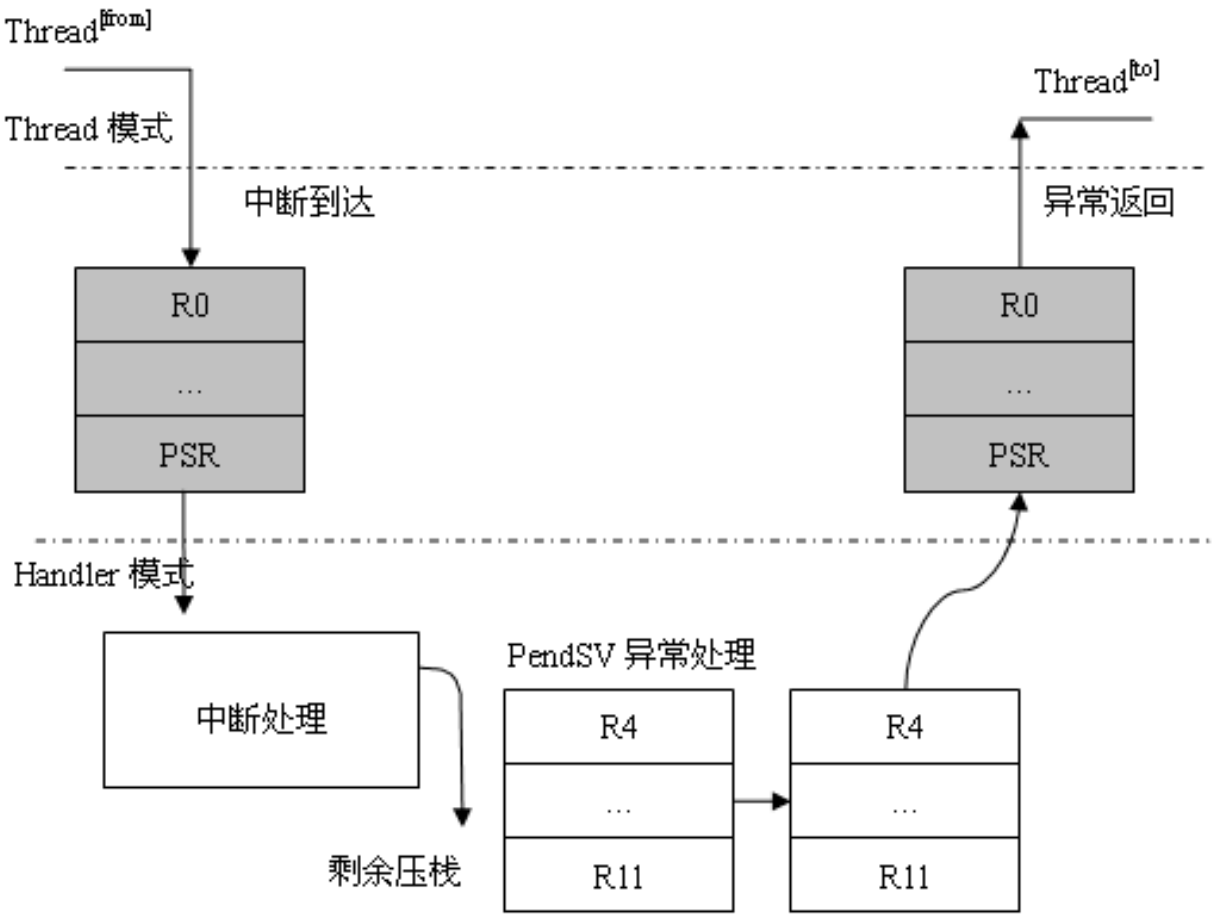
线程切换的过程可以用来图 D - 1 正常模式下的线程上下文切换表示。



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。



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。

19.3 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)
{
        /* 通知RT-Thread进入中断模式 */
        rt_interrupt_enter();

        /* ... 中断处理 */

        /* 通知RT-Thread离开中断模式 */
        rt_interrupt_leave();
        rt_hw_interrupt_thread_switch();
}
```

19.4 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

TWENTY

INDICES AND TABLES

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