

RT-Thread编程指南

RT-Thread开发组¹

2017-07-06

¹这个是RT-Thread编程指南的PDF版本，由github.com上最新提交内容自动生成。

前言

本书是RT-Thread的编程手册，用于指导在RT-Thread实时操作系统环境下如何进行编程。

本书结构

适合读者

本书用于指导在RT-Thread实时操作系统环境下如何进行编程。

变动

日期	修改人	修改说明
2013/5/14	Bernard	加入表格相关的内容；加入API说明例子

致谢

@larrycaiyou 提供本书电子版的初始模板；

@grissiom 解决本书电子版的代码语法高亮问题；

@aozima 提示pandoc 1.11.1版本已经开始支持表格操作；后续开始在文档中加入表格的文本。

@reynoldxu 补足了文档中的图片即API说明。

目录

前言	i
致谢	iii
目录	v
1 简介	1
1.1 RT-Thread 的软件结构	1
1.2 开发、维护	2
2 线程调度与管理	3
2.1 实时系统的需求	3
2.2 线程调度器	3
2.3 线程控制块	5
2.4 线程状态	6
2.5 空闲线程	7
2.6 调度器相关接口	8
2.6.1 调度器初始化	8
2.6.2 启动调度器	8
2.6.3 执行调度	8
2.6.4 设置调度器钩子	9
2.7 线程相关接口	10
2.7.1 线程创建	10
2.7.2 线程删除	12
2.7.3 线程初始化	15
2.7.4 线程脱离	17
2.7.5 线程启动	19
2.7.6 当前线程	20
2.7.7 线程让出处理器	20
2.7.8 线程睡眠	22
2.7.9 线程挂起	23
2.7.10 线程恢复	25
2.7.11 线程控制	27
2.7.12 初始化空闲线程	27
2.7.13 设置空闲线程钩子	28
2.8 线程设计	28
2.8.1 程序的运行上下文	28

2.8.2	线程设计要点	29
3	定时器	31
3.1	定时器管理	31
3.2	定时器超时函数	32
3.3	定时器管理控制块	33
3.4	定时器管理接口	33
3.4.1	定时器管理系统初始化	33
3.4.2	创建定时器	34
3.4.3	删除定时器	35
3.4.4	初始化定时器	36
3.4.5	脱离定时器	38
3.4.6	启动定时器	38
3.4.7	停止定时器	39
3.4.8	控制定时器	39
3.5	合理使用定时器	40
3.5.1	定时器执行上下文	41
3.5.2	OS tick与定时器精度	42
4	任务间同步及通信	43
4.1	关闭中断	43
4.1.1	使用场合	45
4.2	调度器锁	46
4.2.1	使用场合	46
4.3	信号量	46
4.3.1	信号量控制块	47
4.3.2	信号量相关接口	47
	创建信号量	47
	删除信号量	51
	初始化信号量	51
	脱离信号量	55
	获取信号量	55
	无等待获取信号量	56
	释放信号量	56
4.3.3	使用场合	60
	线程同步	60
	锁	61
	中断与线程的同步	61
	资源计数	62
4.4	互斥量	62
4.4.1	互斥量控制块	63
4.4.2	互斥量相关接口	63
	创建互斥量	63
	删除互斥量	64
	初始化互斥量	64
	脱离互斥量	65

获取互斥量	65
释放互斥量	66
4.4.3 使用场合	70
4.5 事件	71
4.5.1 事件控制块	72
4.5.2 事件相关接口	72
创建事件	72
删除事件	72
初始化事件	73
脱离事件	73
接收事件	74
发送事件	74
4.5.3 使用场合	78
4.6 邮箱	79
4.6.1 邮箱控制块	80
4.6.2 邮箱相关接口	80
创建邮箱	80
删除邮箱	81
初始化邮箱	81
脱离邮箱	82
发送邮件	82
等待方式发送邮件	82
接收邮件	83
4.6.3 使用场合	87
4.7 消息队列	87
4.7.1 消息队列控制块	88
4.7.2 消息队列相关接口	89
创建消息队列	89
删除消息队列	89
初始化消息队列	90
脱离消息队列	90
4.7.3 发送消息	91
发送紧急消息	91
接收消息	92
4.7.4 使用场合	96
典型使用	96
同步消息	97
5 内存管理	99
5.1 静态内存池管理	99
5.1.1 静态内存池工作原理	100
静态内存池控制块	100
5.1.2 静态内存池接口	101
创建内存池	101
删除内存池	101
初始化内存池	103

脱离内存池	103
分配内存块	104
释放内存块	104
5.2 动态内存管理	108
5.2.1 小内存管理模块	108
5.2.2 SLAB内存管理模块	109
5.2.3 动态内存接口	110
初始化系统堆空间	110
分配内存块	111
重分配内存块	111
分配多内存块	112
释放内存块	112
设置分配钩子函数	112
设置内存释放钩子函数	113
5.3 更改情况	115
6 I/O设备管理	117
6.1 块设备	118
6.2 I/O设备控制块	119
6.3 I/O设备管理接口	120
6.3.1 注册设备	120
6.3.2 移除设备	121
6.3.3 初始化所有设备	121
6.3.4 查找设备	122
6.3.5 初始化设备	122
6.3.6 打开设备	122
6.3.7 关闭设备	123
6.3.8 读设备	124
6.3.9 写设备	124
6.3.10 控制设备	125
6.3.11 设置数据接收指示	125
6.3.12 设置发送完成指示	126
6.4 设备驱动	126
6.4.1 设备驱动必须实现的接口	126
6.4.2 设备驱动实现的步骤	128
6.4.3 STM32F10x的串口驱动	128
6.4.4 finsh使用uart设备分析	148
7 异常与中断	151
7.1 中断处理过程	151
7.2 中断栈	152
7.3 中断的底半处理	152
7.3.1 底半处理实现范例	153
7.4 中断相关接口	154
7.4.1 装载中断服务例程	154
7.4.2 屏蔽中断源	155

7.4.3	打开被屏蔽的中断源	155
7.4.4	关闭中断	156
7.4.5	打开中断	156
7.4.6	与OS相关的中断接口	157
7.5	ARM Cortex-M中的中断与异常	157
7.6	外设中的中断模式与轮询模式	158
8	应用模块	161
8.1	功能和限制	161
8.2	使用应用模块	161
8.2.1	编译主程序	161
8.2.2	使用应用模块	162
8.3	应用模块API	164
9	移植	167
9.1	使用移植还是自己移植	167
9.2	移植前的准备	167
9.3	RT-Thread在ARM Cortex M3上的移植	169
9.3.1	建立RealView MDK工程	169
9.3.2	添加源文件	172
9.4	RT-Thread/STM32其他部分说明	183
9.5	RT-Thread在ARM Cortex M4上的移植	184
9.5.1	生成MDK工程模板	184
9.5.2	仿照并修改scons相关文件	184
9.5.3	添加其他相关文件	188
9.5.4	Cortex-M4中的内核相关代码分析	190
10	SCons构建系统	193
10.1	什么是构建工具(software construction tool)	193
10.2	RT-Thread构建	193
10.3	安装SCons环境	194
10.3.1	Linux、BSD环境	194
10.3.2	Windows环境	194
10.4	SCons基本使用	194
10.4.1	配置编译器	197
10.4.2	SCons基本命令	198
	scons	198
	scons -jN	198
	scons -c	199
	scons -target=XXX -s	199
	scons -verbose	199
10.5	SCons进阶	200
10.5.1	修改编译器选项	200
10.5.2	内置函数	201
10.5.3	SConscript示例1	203
10.5.4	SConscript示例2	203

10.5.5 SConscript示例3	204
10.5.6 SConscript示例4	205
10.5.7 添加库	206
10.5.8 增加一个SCons命令	207
10.5.9 RT-Thread building脚本	207
10.6 简单的SConstruct	207
10.7 SConstruct与SConscript	208
11 finsh shell	209
11.1 简介	209
11.2 工作模式	209
11.3 什么是shell?	210
11.4 初识finsh	210
11.4.1 finsh(C-Style)	210
11.4.2 finsh(msh)	212
11.4.3 finsh中的按键	212
11.5 finsh特性	212
11.5.1 finsh(c-style)的数据类型	212
11.6 finsh(c-style)中增加命令/变量	213
11.6.1 宏方式	213
11.6.2 函数方式	214
11.7 msh中增加命令	215
11.7.1 添加内置命令	215
11.8 RT-Thread内置命令	217
11.8.1 finsh(c-style)	217
11.8.2 finsh(msh) 内置命令	222
11.9 移植	223
11.10宏选项	223
12 文件系统	225
12.1 简介	225
12.2 文件系统、文件与文件夹	225
12.3 文件系统接口	226
12.3.1 打开文件	226
12.3.2 关闭文件	227
12.3.3 读取数据	228
12.3.4 写入数据	228
12.3.5 更改名称	231
12.3.6 取得状态	232
12.4 目录操作接口	232
12.4.1 创建目录	232
12.4.2 打开目录	233
12.4.3 读取目录	234
12.4.4 取得目录流的读取位置	235
12.4.5 设置下次读取目录的位置	235
12.4.6 重设读取目录的位置为开头位置	236

12.4.7 关闭目录	236
12.4.8 删除目录	237
12.4.9 格式化文件系统	237
12.5 底层驱动接口	237
12.5.1 文件系统初始化	237
12.6 FatFs	238
12.6.1 FatFs 相关宏	239
12.7 NFS	241
12.7.1 RT-Thread中使用NFS	241
主机配置	241
开发板配置	241
12.8 UFFS	243
12.8.1 UFFS配置	243
12.8.2 UFFS 内存精简	244
12.8.3 MTD NAND驱动	245
读写页	247
擦除块	247
块状态检查	247
标记坏块	248
移动页	248
注册MTD NAND设备	249
12.8.4 UFFS示例驱动	249
12.9 jffs2	249
12.10 yaffs	249
13 lwIP - 轻型TCP/IP协议栈	251
13.1 简介	251
13.2 协议分层	251
13.3 lwIP不遵循严格的分层	252
13.4 进程模型 (process model)	253
13.5 操作系统模拟层 (OS emulation layer)	253
13.6 RT-Thread中的lwIP	254
13.6.1 lwIP版本	254
13.6.2 RT-Thread 网络设备管理	254
13.6.3 RT-Thread lwIP有哪些变化	257
13.6.4 RT-Thread lwIP相关代码补充说明	259
13.7 网络编程示例	259
13.7.1 UDP使用示例	260
13.7.2 TCP使用示例	263
14 lwIP-IPv6 支持	269
14.1 lwIP-IPV6 概况	269
14.2 IPv6基础知识	269
14.2.1 IPv6报文格式	269
14.2.2 IPv6扩展报头	270
14.2.3 IPv4与IPv6对比	271

14.2.4 IPv6地址	271
IPv6地址格式	271
IPv6地址分类	271
14.2.5 邻居发现协议	272
14.3 RT-Thread中如何使用IPv6	272
14.3.1 使用IPv4/v6双栈	272
14.3.2 仅使用IPv4	273
14.3.3 对开发板进行Ping测试	273
14.4 IPv6 Socket API实例	275
14.4.1 IPv4/v6 Socket编程异同	275
14.4.2 PC测试程序	276
14.4.3 TCP Server例子	276
14.4.4 TCP Client例子	278
14.4.5 UDP Server例子	281
14.4.6 UDP Client例子	282
15 POSIX接口	285
15.1 简介	285
15.2 在RT-Thread中使用POSIX	285
15.3 POSIX Thread介绍	285
15.3.1 栏杆: barrier	285
16 图像用户界面引擎	287
16.1 介绍	287
16.2 引擎初始化	287
16.3 绘图设备上下文	287
16.4 绘图渲染	288
16.5 基本的GUI引擎应用例子	289
16.6 事件传递机制	289
16.7 控件和剪切域	289
16.8 字体API	289
16.9 图像API	289
附录 A 电子书markdown入门	291
A.1 标题、段落、区块代码	291
A.2 修辞和强调	291
A.3 列表	291
A.4 链接	292
A.5 图片	293
A.6 代码	294
A.7 API说明	294
A.8 注意事项	295

第 1 章

简介

RT-Thread 是一款由中国开源社区主导开发的开源嵌入式实时操作系统（遵循GPLv2+许可协议，当标识产品使用了RT-Thread时可以按照自有代码非开源的方式应用在商业产品中），它包含实时嵌入式系统相关的各个组件：实时操作系统内核，TCP/IP协议栈、文件系统、libc接口、图形引擎等。

本手册是RT-Thread开源嵌入式实时操作系统的使用手册。

1.1 RT-Thread 的软件结构

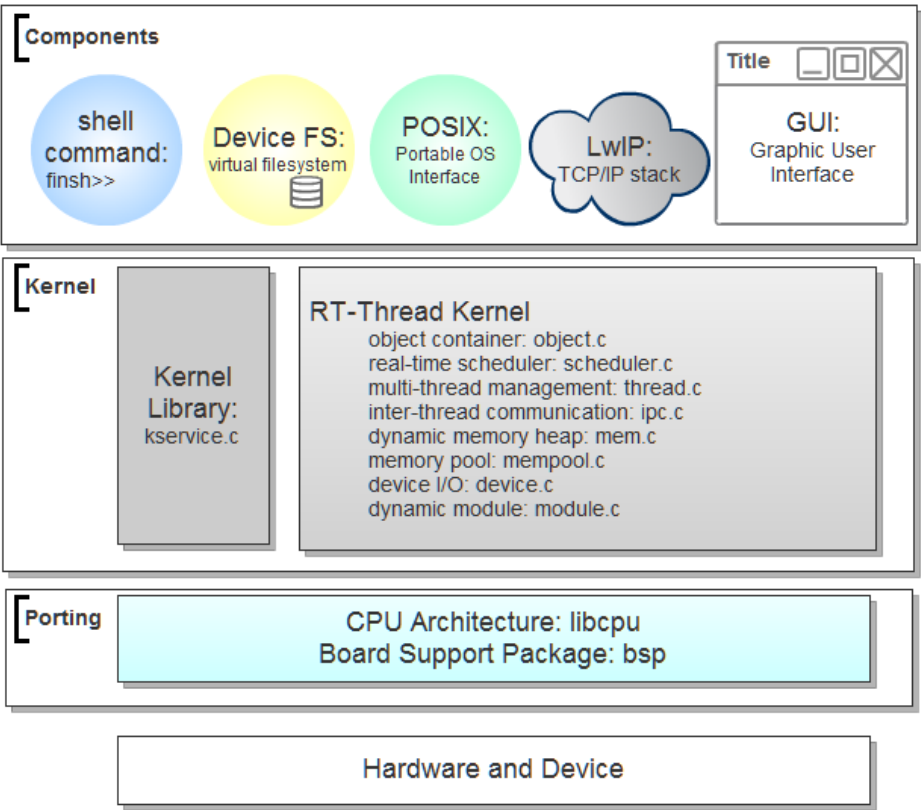


图 1.1: RT-Thread软件结构

RT-Thread实时操作系统是一个分层的操作系统，它包括了：

- 底层移植、驱动层，这层与硬件密切相关，由Drivers和CPU移植相构成。
- 硬实时内核，这层是RT-Thread的核心，包括了内核系统中对象的实现，例如多线程及其调度，信号量，邮箱，消息队列，内存管理，定时器等实现。
- 组件层，这些是基于RT-Thread核心基础上的外围组件，例如文件系统，命令行shell接口，lwIP轻型TCP/IP协议栈，GUI图形引擎等。

RT-Thread在设计及后续的发展方向上会力图保持RT-Thread自己本身的特色：

- 小巧的内核及周边组件；
- 清晰、简单、低耦合的系统结构；
- 面向对象，类UNIX的编程风格；
- 尽可能兼容POSIX可移植操作系统接口的方式；

1.2 开发、维护

RT-Thread的主要开发成员来自中国，大家主要利用业余时间进行RT-Thread的开发和维护，同时也接受开发者，爱好者，以及专业嵌入式领域公司向RT-Thread捐赠代码。在上海也有一家专业提供RT-Thread技术服务的服务公司：[上海睿赛德电子科技有限公司](#)。

RT-Thread以一年为开发、发布周期。RT-Thread的每一个版本都会设定一个目标，而后的开发周期以这个为目标进行开发、演化改进，同时按照每个季度一个测试版本的形式进行推进。发布的版本包括两种：

- 一种是正式版本（或者说稳定版本，维护版本），例如2.0.x正式版本，它是2.0.0正式版本的bug fix版本。在功能上并不添加新的功能，而着重于对已有bug的修正；
- 一种是测试版本（或者说开发版本），例如2.1.0 beta版本。它是以一年期设定目标而演进，完善的版本，相对来说不那么稳定，但具备新的功能，对新的路线的探索；

每个开发版本会提前设定出开发目标，一般是通过邮件、论坛进行沟通后进行；同时每年在中国也会有一到两次的开发者会议，会议上会讨论新版本的目标，或者大版本新的方向。

在开发活动上，RT-Thread相类似的按照上面的软件体系结构划分成三个部分：

- 内核（kernel），这个是RT-Thread的核心，也是根本；
- 组件（component），基于核心之上，把一些功能模块划分成独立的一个个组件模块，做到组件与组件之间的低耦合，组件内部的高内聚；
- 分支（porting），这个是RT-Thread支持的一个个芯片移植，外设驱动等；

这三部分每部分都有维护人，维护人应切实地保证相关部分的正常运行。当前的RT-Thread开发版本放在[github.com](#)上，欢迎每个开发者、爱好者向RT-Thread提交pull request。每个组件、分支的维护人在收到pull request后，会决定是否合并到开发分支中。开发者、爱好者提交的代码应该符合RT-Thread的编程规范，并尽量少地影响到其他组件。

第 2 章

线程调度与管理

一个典型的简单程序会设计成一个串行的系统运行：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“同时”处理多个输入输出，实时软件应用程序应该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。当合理的划分任务，正确的并行执行时，这种设计能够让系统满足实时系统的性能及时间的要求。

2.1 实时系统的需求

如第二章里描述的，系统的实时性指的是在固定的时间内正确地对外部事件做出响应。这个“时间内” (英文叫做deadline、有时中文也翻译成时间约束)，系统内部会做一些处理，例如输入数据的分析计算，加工处理等。而在这段时间之外，系统可能会空闲下来，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，例如听音乐，玩游戏等。

从上面的例子我们可以看出，实时系统是一种需求倾向性的系统，对于实时的事件需要在第一时间内做出回应，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等级系统，不同重要性的任务具有不同的优先等级：重要的事件能够优先被响应执行，非重要的事件可以适当往后推迟。

在RT-Thread实时操作系统中，任务采用了线程来实现，线程是RT-Thread中最基本的调度单位，它描述了一个任务执行的上下文关系，也描述了这个任务所处的优先等级。重要的任务能拥有相对较高的优先级，非重要的任务优先级可以放低，并且可以类似Linux一样具备分时的效果。

2.2 线程调度器

RT-Thread中提供的线程调度器是基于优先级的全抢占式调度：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，数值越小的优先级越高，0为最高优先级，255分配给空闲线程使用，一般用户不使用。在一些资源比较紧张的系统，可以根据实际情况选择只支持8个或32个优先级的系统配置)。在系统中，当有比

当前线程优先级更高的线程就绪时，当前线程将立刻被换出，高优先级线程抢占处理器运行。

如图 线程就绪优先级队列 所示，在RT-Thread调度器的实现中,包含了一个共256个优先级队列的数组(如果系统最大支持32个优先级，那么这里将是一个包含了32个优先级队列的数组)，每个数组元素中放置相同优先级链表的表头。这些相同优先级的列表形成一个双向环形链表，最低优先级线程链表一般只包含一个idle线程。

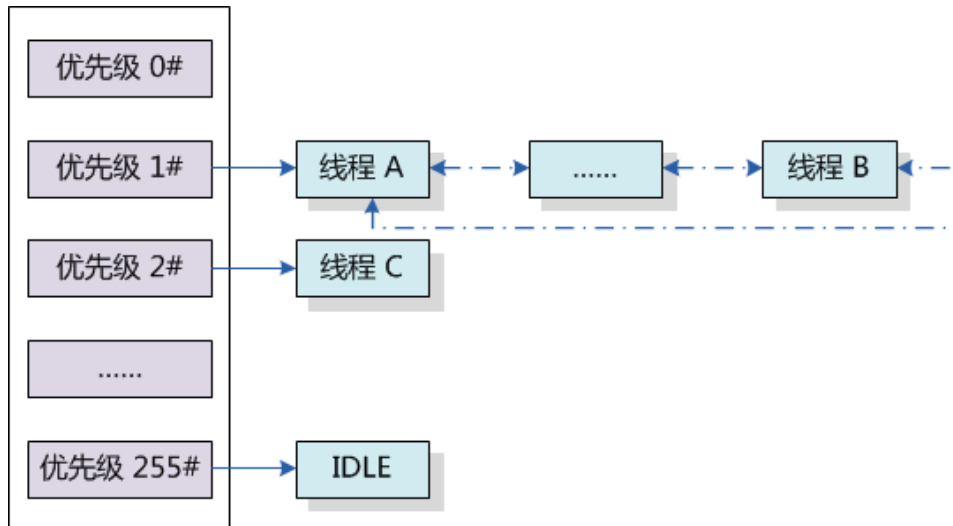


图 2.1: 线程就绪优先级队列

在优先级队列1#和2#中，可以看到三个线程：线程A、线程B和线程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，必须要等待优先级队列1#的中所有线程（因为阻塞）都让出处理器后才能得到执行。

一个操作系统如果只是具备了高优先级任务能够“立即”获得处理器并得到执行的特点，那么它仍然不算是实时操作系统。因为这个查找最高优先级线程的过程决定了调度时间是否具有确定性，例如一个包含n个就绪任务的系统中，如果仅仅从头找到尾，那么这个时间将直接和n相关，而下一个就绪线程抉择时间的长短将会极大的影响系统的实时性。当所有就绪线程都链接在它们对应的优先级队列中时，抉择过程就将演变为在优先级数组中寻找具有最高优先级线程的非空链表。RT-Thread内核中采用了基于位图的优先级算法（时间复杂度 $O(1)$ ，即与就绪线程的多少无关），通过位图的定位快速的获得优先级最高的线程。

RT-Thread内核中也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方式进行调度（也就是通常说的分时调度器），时间片轮转调度仅在当前系统中无更高优先级就绪线程存在的情况下才有效。例如在 线程就绪优先级队列 图中，我们假设线程A和线程B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将在线程A的时间片结束（10个时钟节拍）后才能运行，但如果中途线程A被挂起了，即线程A在运行的途中，因为试图去持有不可用的资源，而导致线程状态从就绪状态更改为阻塞状态，那么线程B会因其优先级成为系统中就绪线程中最高的而马上运行。每个线程的时间片大小都可以在初始化或创建这个线程时指定。

因为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）。

2.3 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连接用的链表结构，线程等待事件集合等。

在RT-Thread实时操作系统中，线程控制块由结构体struct rt_thread表示。另外一种C表达方式rt_thread_t，表示的是线程的句柄，在C语言中的实现是指向线程控制块的指针，详细定义情况见以下代码：

线程控制块结构如下所示

```
/* rt_thread_t线程句柄，指向线程控制块的指针 */
typedef struct rt_thread* rt_thread_t;

/*
 * 线程控制块
 */
struct rt_thread
{
    /* RT-Thread根对象定义 */
    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*/
#ifdef RT_USING_MODULE
    void *module_id;                  /* 线程所在的模块ID*/
#endif
    rt_list_t list;                   /* 对象链表*/

    rt_list_t tlist;                  /* 线程链表*/

    /* 栈指针及入口 */
    void* sp;                         /* 线程的栈指针*/
    void* entry;                      /* 线程入口*/
    void* parameter;                  /* 线程入口参数*/
    void* stack_addr;                 /* 线程栈地址*/
    rt_uint16_t stack_size;           /* 线程栈大小*/

    rt_err_t error;                   /* 线程错误号*/

    rt_uint8_t stat;                  /* 线程状态 */

    /* 优先级相关域 */
    rt_uint8_t current_priority;      /* 当前优先级*/
    rt_uint8_t init_priority;         /* 初始线程优先级*/
#ifdef RT_THREAD_PRIORITY_MAX > 32
    rt_uint8_t number;
    rt_uint8_t high_mask;
#endif
#endif
```

```
rt_uint32_t number_mask;

#ifdef RT_USING_EVENT
    /* 事件相关域 */
    rt_uint32_t event_set;
    rt_uint8_t event_info;
#endif

    rt_ubase_t init_tick;          /* 线程初始tick*/
    rt_ubase_t remaining_tick;     /* 线程当次运行剩余tick */

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

    /* 当线程退出时，需要执行的清理函数 */
    void (*cleanup)(struct rt_thread *tid);
    rt_uint32_t user_data;         /* 用户数据*/
};
```

其中init_priority是线程创建时指定的线程优先级，在线程运行过程当中是不会被改变的（除非用户执行线程控制函数进行手动调整线程优先级）。cleanup成员是RT-Thread 1.0.0中新引入的成员，它会在线程退出时，被idle线程回调一次以执行用户设置的清理现场等工作。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，以提供类似线程私有数据的实现，例如lwIP线程中用于放置定时器链表的表头。

2.4 线程状态

线程运行的过程中，一个时间内只允许一个线程在处理器中运行，从运行的过程上划分，线程有多种不同的运行状态，如运行态，非运行态等。在RT-Thread实时操作系统中，线程包含五种状态，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中的五种线程状态如下所示：

状态	描述
RT_THREAD_INIT	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；在这个状态下，线程不参与调度
RT_THREAD_SUSPEND	挂起态、阻塞态。线程此时被挂起：它可能因为资源不可用而挂起等待；或线程主动延时一段时间而被挂起。在这个状态下，线程不参与调度
RT_THREAD_READY	就绪态。线程正在运行；或当前线程运行完让出处理器后，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。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_CLOSE	线程结束态。当线程运行结束时将处于这个状态。这个状态的线程不参与线程的调度。

RT-Thread实时操作系统提供一系列的操作系统调用接口，使得线程的状态在这五个状态之间来回的变换。例如一个就绪态的线程由于申请一个资源（例如使用`rt_sem_take`），而可能进入挂起态。又例如因为一个外部中断发生了，系统转入中断服务例程，在中断服务例程中释放了相应的资源，导致把等待在这个资源上的高优先级线程唤醒，改变其状态为就绪态，导致当前运行线程切换等等。

几种状态间的转换关系如 线程转换图 所示：

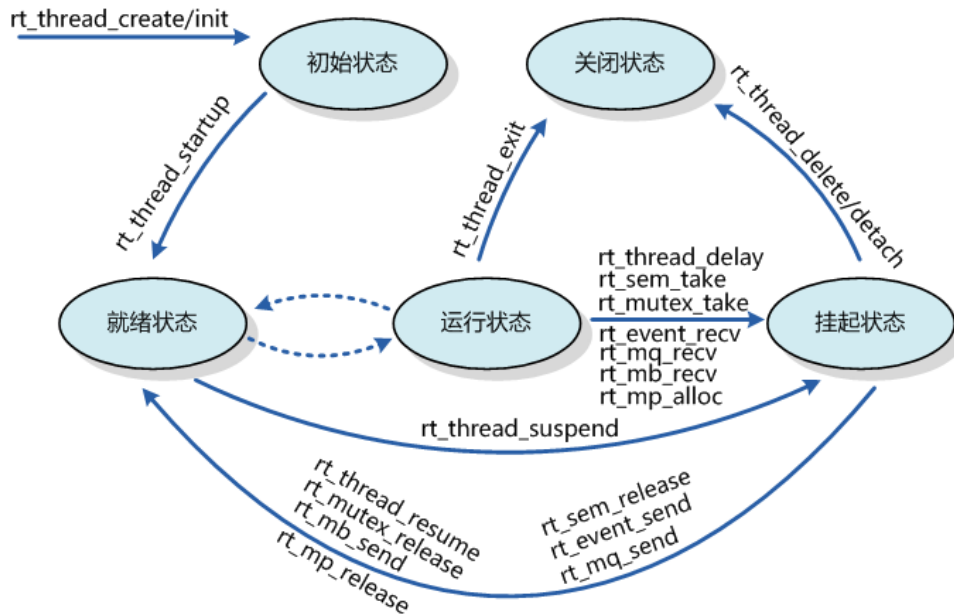


图 2.2: 线程转换图

线程通过调用函数`rt_thread_create/init`进入到初始状态（`RT_THREAD_INIT`）；再通过调用函数`rt_thread_startup`进入到就绪状态（`RT_THREAD_READY`）；当处于就绪状态的线程调用`rt_thread_delay`，`rt_sem_take`，`rt_mb_rcv`等函数或由于获取不到资源时，将进入到挂起状态（`RT_THREAD_SUSPEND`）；处于挂起状态的线程，如果等待超时依然未能获得资源或由于其他线程释放了资源，那么它将返回到就绪状态。挂起状态的线程，如果调用`rt_thread_delete/detach`将更改为关闭状态（`RT_THREAD_CLOSE`）；而运行状态的线程，如果运行结束会在线程最后部分执行`rt_thread_exit`函数而更改为关闭状态（`RT_THREAD_CLOSE`）。

2.5 空闲线程

空闲线程是系统线程中一个比较特殊的线程，它具有最低的优先级，当系统中无其他线程可运行时，调度器将调度到空闲线程。空闲线程通常是一个死循环，永远不被挂起。

RT-Thread实时操作系统为空闲线程提供了钩子函数（钩子函数：用户提供的一段代码，在系统运行的某一路径上设置一个钩子，当系统经过这个位置时，转而执行这个钩子函数，然后再返回到它的正常路径上），可以让系统在空闲的时候执行一些特定的任务，例如系统运行指示灯闪烁，电源管理等。除了调用钩子函数，RT-Thread也把线程清理（`rt_thread->cleanup`回调函数）函数、真正的线程删除动作放到了空闲线程中（在删除线程时，仅改变线程的状态为关闭状态不再参与系统调度）。

2.6 调度器相关接口

2.6.1 调度器初始化

在系统启动时需要执行调度器的初始化，以初始化系统调度器用到的一些全局变量。调度器初始化可以调用下面的函数接口。

```
void rt_system_scheduler_init(void);
```

线程安全
不安全
中断例程
不可调用
函数参数
无
函数返回
无

2.6.2 启动调度器

在系统完成初始化后切换到第一个线程，可以调用下面的函数接口。

```
void rt_system_scheduler_start(void);
```

在调用这个函数时，它会查找系统中优先级最高的就绪态线程，然后切换过去执行。另外在调用这个函数前，必须先做idle线程的初始化，即保证系统至少能够找到一个就绪状态的线程执行。此函数是永远不会返回的。

线程安全
不安全
中断例程
不可调用
函数参数
无
函数返回
无

2.6.3 执行调度

让调度器执行一次线程的调度可通过下面的函数接口。

```
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，系统将切换到高优先级的线程去。上层应用程序一般不需要调用这个函数。

线程安全
安全
中断例程
可调用
函数参数
无
函数返回
无

- 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需要切换到的更高优先级线程，在中断服务例程处理完毕后执行真正的线程上下文切换（即中断中的线程上下文切换），最终切换到目标线程去。

2.6.4 设置调度器钩子

在整个系统的运行时，系统都处于线程运行、中断触发-响应中断、切换到其他线程，甚至是线程间的切换过程中，或者说系统的上下文切换是系统中最普遍的事件。有时用户可能会想知道在一个时刻发生了什么样的线程切换，可以通过调用下面的函数接口设置一个相应的钩子函数。在系统线程切换时，这个钩子函数将被调用：

```
void rt_scheduler_sethook(void (*hook)(struct rt_thread* from, struct rt_thread* to));
```

这个函数用于把用户提供的hook函数设置到系统调度器钩子中，当系统进行上下文切换时，这个hook函数将会被系统调用。

线程安全
安全
中断例程
可调用
函数参数

参数	描述
hook	表示用户定义的钩子函数指针；

这个hook函数的声明如下：

```
void hook(struct rt_thread* from, struct rt_thread* to);
```

线程安全
安全
中断例程
可调用

函数参数

参数	描述
from	表示系统所要切换出的线程控制块指针；
to	表示系统所要切换到的线程控制块指针。

函数返回

无

- 注：请仔细编写你的钩子函数，稍有不慎将很可能导致整个系统运行不正常（在这个钩子函数中，基本上不允许调用系统API，更不应该导致当前运行的上下文挂起）。

2.7 线程相关接口

2.7.1 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的内核来为它创建（初始化）一个线程句柄。可以通过如下的函数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);
```

调用这个函数时，系统会从动态堆内存中分配一个线程句柄（即TCB，线程控制块）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。分配出来的栈空间是按照rtconfig.h中配置的RT_ALIGN_SIZE方式对齐。

线程安全
安全
中断例程
不可调用
函数参数

参数	描述
name	线程的名称；线程名称的最大长度由rtconfig.h中定义的RT_NAME_MAX宏指定，多余部分会被自动截掉。
entry	线程入口函数
parameter	线程入口函数参数；
stack_size	线程栈大小，单位是字节。在大多数系统中需要做栈空间地址对齐（例如ARM体系结构中需要向4字节地址对齐）。
priority	线程的优先级。优先级范围根据系统配置情况（rtconfig.h中的RT_THREAD_PRIORITY_MAX宏定义），如果支持的是256级优先级，那么范围是从0 ~ 255，数值越小优先级越高，0代表最高优先级。

tick	线程的时间片大小。时间片（tick）的单位是操作系统的时钟节拍。当系统中存在相同优先级线程时，这个参数指定线程一次调度能够运行的最大时间长度。这个时间片运行结束时，调度器自动选择下一个就绪态的同优先级线程进行运行。
------	---

函数返回

创建成功返回线程句柄；否则返回RT_NULL。

- 注：确定一个线程的栈空间大小，是一件令人头痛的事情。在RT-Thread中，可以先指定一个稍微大的栈空间，例如指定大小为1024或2048，然后在FinSH shell中通过list_thread()命令查看线程运行的过程中线程所使用的栈的大小，通过此命令，能够看到从线程启动运行时，到当前时刻点，线程使用的最大栈深度，从而可以确定栈空间的大小并加以修改。

下面举例创建一个线程加以说明：

```
/*
 * 程序清单：动态线程
 *
 * 这个程序会初始化2个动态线程：
 * 它们拥有共同的入口函数，相同的优先级
 * 但是它们的入口参数不相同
 */
#include <rtthread.h>

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
/* 线程入口 */
static void thread_entry(void* parameter)
{
    rt_uint32_t count = 0;
    rt_uint32_t no = (rt_uint32_t) parameter; /* 获得线程的入口参数 */

    while (1)
    {
        /* 打印线程计数值输出 */
        rt_kprintf("thread%d count: %d\n", no, count ++);

        /* 休眠10个OS Tick */
        rt_thread_delay(10);
    }
}
```

```
/* 用户应用入口 */
int rt_application_init()
{
    /* 创建线程1 */
    tid1 = rt_thread_create("t1",
        thread_entry, (void*)1, /* 线程入口是thread_entry, 入口参数是1 */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);
    else
        return -1;

    /* 创建线程2 */
    tid2 = rt_thread_create("t2",
        thread_entry, (void*)2, /* 线程入口是thread_entry, 入口参数是2 */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
    else
        return -1;

    return 0;
}
```

2.7.2 线程删除

对于一些使用rt_thread_create创建出来的线程，当不需要使用，或者运行出错时，我们可以使用下面的函数接口来从系统中把线程完全删除掉：

```
rt_err_t rt_thread_delete(rt_thread_t thread);
```

调用该函数后，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，收回的空间将重新用于其他的内存分配。实际上，用rt_thread_delete函数删除线程接口，仅仅是把相应的线程状态更改为RT_THREAD_CLOSE状态，然后放入到rt_thread_defunct队列中；而真正的删除动作（释放线程控制块和释放线程栈）需要到下一次执行idle线程时，由idle线程完成最后的线程删除动作。用rt_thread_init初始化的静态线程则不能使用此接口删除。

- 线程安全
- 安全
- 中断例程
- 可调用
- 函数参数

参数	描述
----	----

thread	要删除的线程句柄;
--------	-----------

函数返回

返回RT_EOK

- 注：在线程运行完成，自动结束的情况下，系统会自动删除线程，不需要再调用rt_thread_delete()函数接口。这个接口不应由线程本身来调用以删除线程自身，一般只能由其他线程调用或在定时器超时函数中调用。

这个函数仅在使用了系统动态堆时才有效（即RT_USING_HEAP宏定义已经定义了）。
下面举一个删除线程的例子，如下代码：

```
/*
 * 程序清单：删除线程
 *
 * 这个例子会创建两个线程，在一个线程中删除另外一个线程。
 */
#include <rtthread.h>

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

/*
 * 线程删除(rt_thread_delete)函数仅适合于动态线程，为了在一个线程
 * 中访问另一个线程的控制块，所以把线程块指针声明成全局类型以供全
 * 局访问
 */
static rt_thread_t tid1 = RT_NULL, tid2 = RT_NULL;

/* 线程1的入口函数 */
static void thread1_entry(void* parameter)
{
    rt_uint32_t count = 0;

    while (1)
    {
        /* 线程1采用低优先级运行，一直打印计数值 */
        rt_kprintf("thread count: %d\n", count ++);
    }
}

/* 线程2的入口函数 */
static void thread2_entry(void* parameter)
{
```

```

/* 线程2拥有较高的优先级，以抢占线程1而获得执行 */

/* 线程2启动后先睡眠10个OS Tick */
rt_thread_delay(10);

/*
 * 线程2唤醒后直接删除线程1，删除线程1后，线程1自动脱离就绪线程
 * 队列
 */
rt_thread_delete(tid1);
tid1 = RT_NULL;

/*
 * 线程2继续休眠10个OS Tick然后退出，线程2休眠后应切换到idle线程
 * idle线程将执行真正的线程1控制块和线程栈的删除
 */
rt_thread_delay(10);

/*
 * 线程2运行结束后也将自动被删除(线程控制块和线程栈依然在idle线
 * 程中释放)
 */
tid2 = RT_NULL;
}

/* 应用入口 */
int rt_application_init()
{
    /* 创建线程1 */
    tid1 = rt_thread_create("t1", /* 线程1的名称是t1 */
        thread1_entry, RT_NULL, /* 入口是thread1_entry, 参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL) /* 如果获得线程控制块，启动这个线程 */
        rt_thread_startup(tid1);
    else
        return -1;

    /* 创建线程2 */
    tid2 = rt_thread_create("t2", /* 线程2的名称是t2 */
        thread2_entry, RT_NULL, /* 入口是thread2_entry, 参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    if (tid2 != RT_NULL) /* 如果获得线程控制块，启动这个线程 */
        rt_thread_startup(tid2);
    else
        return -1;
}

```

```
    return 0;
}
```

2.7.3 线程初始化

线程的初始化可以使用下面的函数接口完成：

```
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);
```

rt_thread_init函数用来初始化静态线程对象。而线程句柄（或者说线程控制块指针），线程栈由用户提供。静态线程是指，线程控制块、线程运行栈一般都设置为全局变量，在编译时就被确定、被分配处理，内核不负责动态分配内存空间。需要注意的是，用户提供的栈首地址需做系统对齐（例如ARM上需要做4字节对齐）。

线程安全
安全
中断例程
可调用
函数参数

参数	描述
thread	线程句柄。线程句柄由用户提供出来，并指向对应的线程控制块内存地址。
name	线程的名称；线程名称的最大长度由rtconfig.h中定义的RT_NAME_MAX宏指定，多余部分会被自动截掉。
entry	线程入口函数；
parameter	线程入口函数参数；
stack_start	线程栈起始地址；
stack_size	线程栈大小，单位是字节。在大多数系统中需要做栈空间地址对齐（例如ARM体系结构中需要向4字节地址对齐）。
priority	线程的优先级。优先级范围根据系统配置情况（rtconfig.h中的RT_THREAD_PRIORITY_MAX宏定义），如果支持的是256级优先级，那么范围是从0 ~ 255，数值越小优先级越高，0代表最高优先级。
tick	线程的时间片大小。时间片（tick）的单位是操作系统的时钟节拍。当系统中存在相同优先级线程时，这个参数指定线程一次调度能够运行的最大时间长度。这个时间片运行结束时，调度器自动选择下一个就绪态的同优先级线程进行运行。

函数返回

返回值：返回RT_EOK；

下面给出一个线程初始化的例子，如下代码：

```

/*
 * 程序清单：初始化静态线程
 *
 * 这个程序会初始化2个静态线程，它们拥有共同的入口函数，但参数不相同
 */
#include <rtthread.h>

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

/* 线程1控制块 */
static struct rt_thread thread1;
/* 线程1栈 */
ALIGN(4)
static rt_uint8_t thread1_stack[THREAD_STACK_SIZE];
/* 线程2控制块 */
static struct rt_thread thread2;
/* 线程2栈 */
ALIGN(4)
static rt_uint8_t thread2_stack[THREAD_STACK_SIZE];

/* 线程入口 */
static void thread_entry(void* parameter)
{
    rt_uint32_t count = 0;
    rt_uint32_t no = (rt_uint32_t) parameter; /* 获得正确的入口参数 */

    while (1)
    {
        /* 打印线程计数值输出 */
        rt_kprintf("thread%d count: %d\n", no, count ++);

        /* 休眠10个OS Tick */
        rt_thread_delay(10);
    }
}

/* 用户应用入口 */
int rt_application_init()
{
    rt_err_t result;

```

```
/* 初始化线程1 */
result = rt_thread_init(&thread1, "t1", /* 线程名:t1 */
    thread_entry, (void*)1, /* 线程的入口是thread_entry, 入口参数是1 */
    &thread1_stack[0], sizeof(thread1_stack), /* 线程栈是thread1_stack */
    THREAD_PRIORITY, 10);
if (result == RT_EOK) /* 如果返回正确, 启动线程1 */
    rt_thread_startup(&thread1);
else
    return -1;

/* 初始化线程2 */
result = rt_thread_init(&thread2, "t2", /* 线程名:t2 */
    thread_entry, (void*)2, /* 线程的入口是thread_entry, 入口参数是2 */
    &thread2_stack[0], sizeof(thread2_stack), /* 线程栈是thread2_stack */
    THREAD_PRIORITY + 1, 10);
if (result == RT_EOK) /* 如果返回正确, 启动线程2 */
    rt_thread_startup(&thread2);
else
    return -1;

return 0;
}
```

2.7.4 线程脱离

线程脱离将使线程对象在线程队列和内核对象管理器中被删除。线程脱离使用下面的函数：

```
rt_err_t rt_thread_detach (rt_thread_t thread);
```

线程安全
安全
中断例程
可调用
函数参数

参数	描述
thread	线程句柄，它应该是由rt_thread_init进行初始化的线程句柄。

函数返回
返回RT_EOK

- 注：这个函数接口是和rt_thread_delete()函数相对应的， rt_thread_delete()函数操作

的对象是rt_thread_create()创建的句柄，而rt_thread_detach()函数操作的对象是使用rt_thread_init()函数初始化的线程控制块。同样，线程本身不应调用这个接口脱离线程本身。

线程脱离的例子如下所示：

```
/*
 * 程序清单：线程脱离
 *
 * 这个例子会创建两个线程(t1和t2)，在t2中会对t1进行脱离操作；
 * t1脱离后将不在运行，状态也更改为初始状态
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 线程1控制块 */
static struct rt_thread thread1;
/* 线程1栈 */
static rt_uint8_t thread1_stack[THREAD_STACK_SIZE];
/* 线程2控制块 */
static struct rt_thread thread2;
/* 线程2栈 */
static rt_uint8_t thread2_stack[THREAD_STACK_SIZE];

/* 线程1入口 */
static void thread1_entry(void* parameter)
{
    rt_uint32_t count = 0;

    while (1)
    {
        /* 线程1采用低优先级运行，一直打印计数值 */
        rt_kprintf("thread count: %d\n", count ++);
    }
}

/* 线程2入口 */
static void thread2_entry(void* parameter)
{
    /* 线程2拥有较高的优先级，以抢占线程1而获得执行 */

    /* 线程2启动后先睡眠10个OS Tick */
    rt_thread_delay(10);

    /*
     * 线程2唤醒后直接执行线程1脱离，线程1将从就绪线程队列中删除
     */
}
```



```

rt_thread_detach(&thread1);

/*
 * 线程2继续休眠10个OS Tick然后退出
 */
rt_thread_delay(10);

/*
 * 线程2运行结束后也将自动被从就绪队列中删除，并脱离线程队列
 */
}

int rt_application_init(void)
{
    rt_err_t result;

    /* 初始化线程1 */
    result = rt_thread_init(&thread1, "t1", /* 线程名：t1 */
        thread1_entry, /* 线程的入口是thread1_entry */
        RT_NULL, /* 入口参数是RT_NULL */
        &thread1_stack[0], /* 线程栈是thread1_stack */
        sizeof(thread1_stack),
        THREAD_PRIORITY, 10);
    if (result == RT_EOK) /* 如果返回正确，启动线程1 */
        rt_thread_startup(&thread1);

    /* 初始化线程2 */
    result = rt_thread_init(&thread2, "t2", /* 线程名：t2 */
        thread2_entry, /* 线程的入口是thread2_entry */
        RT_NULL, /* 入口参数是RT_NULL */
        &thread2_stack[0], /* 线程栈是thread2_stack */
        sizeof(thread2_stack),
        THREAD_PRIORITY - 1, 10);
    if (result == RT_EOK) /* 如果返回正确，启动线程2 */
        rt_thread_startup(&thread2);

    return 0;
}

```

2.7.5 线程启动

创建（初始化）的线程对象的状态处于初始态，并未进入就绪线程的调度队列，我们可以调用下面的函数接口启动一个线程：

```
rt_err_t rt_thread_startup(rt_thread_t thread);
```

当调用这个函数时，将把线程的状态更改为就绪状态，并放到相应优先级队列中等待调度。如果新启动的线程优先级比当前线程优先级高，将立刻切换到这个线程。

线程安全
安全
中断例程
可调用
函数参数

参数	描述
thread	线程句柄。

函数返回
返回RT_EOK

2.7.6 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函数接口获得当前执行的线程句柄。

```
rt_thread_t rt_thread_self(void);
```

线程安全
安全
中断例程
可调用
函数参数

参数	描述
thread	线程句柄。

函数返回
返回当前运行的线程句柄。如果调度器还未启动，将返回RT_NULL。

- 注：请不要在中断服务程序中调用此函数，因为它并不能准确获得当前的执行线程。当调度器未启动时，这个接口返回RT_NULL。

2.7.7 线程让出处理器

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，它不再占有处理器，调度器会选择相同优先级的下一个线程执行。线程调用这个接口后，这个线程仍然在就绪队列中。线程让出处理器使用下面的函数接口：

```
rt_err_t rt_thread_yield(void);
```

调用该函数后，当前线程首先把自己从它所在的就绪优先级线程队列中删除，然后把自己挂到这个优先级队列链表的尾部，然后激活调度器进行线程上下文切换（如果当前优先级只有这一个线程，则这个线程继续执行，不进行上下文切换动作）。

线程安全

安全

中断例程

可调用

函数参数

无

函数返回

无

线程让出处理器代码的例子如下示：

```
/*
 * 程序清单：线程让出处理器
 * 在这个例子中，将创建两个相同优先级的线程，它们会通过rt_thread_yield
 * 接口把处理器相互让给对方进行执行。
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
/* 线程1入口 */
static void thread1_entry(void* parameter)
{
    rt_uint32_t count = 0;

    while (1)
    {
        /* 打印线程1的输出 */
        rt_kprintf("thread1: count = %d\n", count ++);

        /* 执行yield后应该切换到thread2执行 */
        rt_thread_yield();
    }
}

/* 线程2入口 */
static void thread2_entry(void* parameter)
{
    rt_uint32_t count = 0;

    while (1)
    {
```

```

    /* 打印线程2的输出 */
    rt_kprintf("thread2: count = %d\n", count ++);

    /* 执行yield后应该切换到thread1执行 */
    rt_thread_yield();
}
}

int rt_application_init(void)
{
    /* 创建线程1 */
    tid1 = rt_thread_create("thread",
        thread1_entry,      /* 线程入口是thread1_entry */
        RT_NULL,           /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    /* 创建线程2 */
    tid2 = rt_thread_create("thread",
        thread2_entry,      /* 线程入口是thread2_entry */
        RT_NULL,           /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);

    return 0;
}

```

- 注：rt_thread_yield()函数和rt_schedule()函数比较相像，但在有相同优先级的其他就绪态线程存在时，系统的行为却完全不一样。执行rt_thread_yield()函数后，当前线程被换出，相同优先级的下一个就绪线程将被执行。而执行rt_schedule()函数后，当前线程并不一定被换出，即使被换出，也不会被放到就绪线程链表的尾部，而是在系统中选取就绪的优先级最高的线程执行（如果系统中没有比当前线程优先级更高的线程存在，那么执行完rt_schedule()函数后，系统将执行当前线程）。

2.7.8 线程睡眠

在实际应用中，我们有时需要让运行的当前线程延迟一段时间，在指定的时间到达后重新运行，这就叫做“线程睡眠”。线程睡眠可使用以下两个函数接口：

```

rt_err_t rt_thread_sleep(rt_tick_t tick);
rt_err_t rt_thread_delay(rt_tick_t tick);

```

这两个函数接口的作用相同，调用它们可以使当前线程挂起一段指定的时间，当这个时间过后，线程会被唤醒并再次进入就绪状态。这个函数接受一个参数，该参数指定了线程的

休眠时间（单位是OS Tick时钟节拍）。

线程安全
安全
中断例程
不可调用
函数参数

参数	描述
tick	线程睡眠的时间。

函数返回
返回RT_EOK

2.7.9 线程挂起

当线程调用rt_thread_delay，调用线程将主动挂起，当调用rt_sem_take，rt_mb_recv等函数时，资源不可使用也将导致调用线程挂起。处于挂起状态的线程，如果其等待的资源超时（超过其设定的等待时间），那么该线程将不再等待这些资源，并返回到就绪状态；或者，当其它线程释放掉该线程所等待的资源时，该线程也会返回到就绪状态。

线程挂起使用下面的函数接口：

```
rt_err_t rt_thread_suspend (rt_thread_t thread);
```

线程安全
安全
中断例程
可调用
函数参数

参数	描述
thread	线程句柄。

函数返回
如果这个线程的状态并不是就绪状态，将返回-RT_ERROR，否则返回RT_EOK

- 注：通常不应该使用这个函数来挂起线程本身，如果确实需要采用rt_thread_suspend函数挂起当前任务，需要在调用rt_thread_suspend()函数后立刻调用rt_schedule()函数进行手动的线程上下文切换。

线程挂起的例子如下：

```
/*  
 * 程序清单：挂起线程  
 */
```

```

* 这个例子中将创建两个动态线程(t1和t2)
* 低优先级线程t1在启动后将一直持续运行；
* 高优先级线程t2在一定时刻后唤醒并挂起低优先级线程。
*/
#include <rtthread.h>
#include "tc_comm.h"

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
/* 线程1入口 */
static void thread1_entry(void* parameter)
{
    rt_uint32_t count = 0;

    while (1)
    {
        /* 线程1采用低优先级运行，一直打印计数值 */
        rt_kprintf("thread count: %d\n", count ++);
    }
}

/* 线程2入口 */
static void thread2_entry(void* parameter)
{
    /* 延时10个OS Tick */
    rt_thread_delay(10);

    /* 挂起线程1 */
    rt_thread_suspend(tid1);

    /* 延时10个OS Tick */
    rt_thread_delay(10);

    /* 线程2自动退出 */
}

int rt_application_init(void)
{
    /* 创建线程1 */
    tid1 = rt_thread_create("t1",
        thread1_entry, /* 线程入口是thread1_entry */
        RT_NULL,       /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)

```

```
    rt_thread_startup(tid1);

    /* 创建线程2 */
    tid2 = rt_thread_create("t2",
        thread2_entry, /* 线程入口是thread2_entry */
        RT_NULL,      /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);

    return 0;
}
```

2.7.10 线程恢复

线程恢复就是让挂起的线程重新进入就绪状态，如果被恢复线程在所有就绪态线程中，位于最高优先级链表的第一位，那么系统将进行线程上下文的切换。线程恢复使用下面的函数接口：

```
rt_err_t rt_thread_resume (rt_thread_t thread);
```

线程安全
安全
中断例程
可调用
函数参数

参数	描述
thread	要恢复的线程句柄。

函数返回

如果恢复的线程状态并不是RT_THREAD_SUSPEND状态，将返回-RT_ERROR；否则返回RT_EOK。

线程恢复的例子如下代码所示：

```
/*
 * 程序清单：唤醒线程
 *
 * 这个例子中将创建两个动态线程(t1和t2),
 * 低优先级线程t1将挂起自身
 * 高优先级线程t2将在一定时刻后唤醒低优先级线程。
 */
#include <rtthread.h>
#include "tc_comm.h"
```

```

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
/* 线程1入口 */
static void thread1_entry(void* parameter)
{
    /* 低优先级线程1开始运行 */
    rt_kprintf("thread1 startup%d\n");

    /* 挂起自身 */
    rt_kprintf("suspend thread self\n");
    rt_thread_suspend(tid1);
    /* 主动执行线程调度 */
    rt_schedule();

    /* 当线程1被唤醒时 */
    rt_kprintf("thread1 resumed\n");
}

/* 线程2入口 */
static void thread2_entry(void* parameter)
{
    /* 延时10个OS Tick */
    rt_thread_delay(10);

    /* 唤醒线程1 */
    rt_thread_resume(tid1);

    /* 延时10个OS Tick */
    rt_thread_delay(10);

    /* 线程2自动退出 */
}

int rt_application_init(void)
{
    /* 创建线程1 */
    tid1 = rt_thread_create("t1",
        thread1_entry,      /* 线程入口是thread1_entry */
        RT_NULL,           /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);
}

```



```
/* 创建线程2 */
tid2 = rt_thread_create("t2",
    thread2_entry,      /* 线程入口是thread2_entry */
    RT_NULL,           /* 入口参数是RT_NULL */
    THREAD_STACK_SIZE, THREAD_PRIORITY - 1,
    THREAD_TIMESLICE);
if (tid2 != RT_NULL)
    rt_thread_startup(tid2);

return 0;
}
```

2.7.11 线程控制

当需要对线程进行一些其他控制时，例如动态更改线程的优先级，可以调用如下函数接口：

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

线程安全
安全
中断例程
可调用
函数参数

参数	描述
thread	线程句柄
cmd	指示控制命令：
arg	控制参数

指示控制命令cmd当前支持的命令包括

- RT_THREAD_CTRL_CHANGE_PRIORITY - 动态更改线程的优先级；
- RT_THREAD_CTRL_STARTUP - 开始运行一个线程，等同于rt_thread_startup()函数调用；
- RT_THREAD_CTRL_CLOSE - 关闭一个线程，等同于rt_thread_delete()函数调用。

函数返回

控制执行正确返回RT_EOK，否则返回RT_ERROR。

2.7.12 初始化空闲线程

根据前面的描述，系统运行过程中必须存在一个最终可运行的线程，可以调用如下函数初始化空闲线程：

```
void rt_thread_idle_init(void);
```

线程安全
不安全
中断例程
不可调用
函数参数
无
函数返回
无

2.7.13 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_sethook(void (*hook)(void));
```

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，所以只有在空闲的时候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态的线程，因此设置的钩子函数必须保证空闲线程在任何时刻都不会处于挂起状态，例如 `rt_thread_delay()`，`rt_sem_take()` 等可能会导致线程挂起的函数都不能使用。

线程安全
不安全
中断例程
不可调用
函数参数

参数	描述
hook	设置的钩子函数

函数返回
无

2.8 线程设计

2.8.1 程序的运行上下文

采用RT-Thread的实时系统，程序运行状态只有几种类型，但这几种类型构成了程序运行的上下文状态，当程序员清晰的知道自己编写的程序处于何种状态时，对于程序中应该注意什么要点将非常清晰了然。

RT-Thread中程序运行的上下文包括：

- 中断服务例程;

- 普通线程;
- 空闲线程;

空闲线程

空闲线程是RT-Thread系统中没有其他工作进行时自动进入的系统线程。开发者可以通过idle线程钩子方式，在idle线程上钩入自己的功能函数。通常这个空闲线程钩子能够完成一些额外的特殊功能，例如系统运行状态的指示，系统省电模式等。

除了空闲线程钩子，RT-Thread系统还把idle线程用于一些其他的功能，比如当系统删除一个线程或一个动态线程运行结束时，会先行更改线程状态为非调度状态，然后挂入一个僵尸队列中，真正的系统资源回收工作在idle线程完成。所以，对于空闲线程钩子上挂接的程序，它应该：

- 不会挂起idle线程；
- 不应该陷入死循环，需要留出部分时间用于系统处理僵尸线程的系统资源回收。

中断服务例程

中断服务例程是一种需要特别注意的上下文环境，它运行在非线程的执行环境下（一般为芯片的一种特殊运行模式（特权模式）），在这个上下文环境中不能使用挂起当前线程的操作，因为当前线程并不存在，执行相关的操作会有类似：

```
Function[abc_func] shall not used in ISR
```

的打印信息（其中abc_func就是你不应该在中断服务例程中调用的函数）。另外需要注意的是，中断服务程序最好保持精简短小，因为中断服务是一种高于任何线程的存在。

普通线程

普通线程看似没有什么限制程序执行的因素，似乎所有的操作都可以执行。但是做为一个实时系统，一个优先级明确的实时系统，如果一个线程中的程序执行了死循环操作，那么比它优先级低的线程都将不能够得到执行，当然也包括了idle线程。这个是在实时操作系统中必须注意的一点。

2.8.2 线程设计要点

在实时系统的章节中提到过，实时系统多数是一种被动的系统，被动的响应外部事件，当外部事件触发后执行设定的工作内容。所以在对系统进行线程设计时，需要考虑到：

上下文环境

对于工作内容，首先需要思考它的执行环境是什么。工作内容与工作内容间是否有重叠的部分，是否能够合并到一起进行处理，或者单独划分开进行处理。例如对键盘事件的处理：在正常情况下，键盘可以采用中断服务例程直接产生RT-Thread/GUI所要求的键盘事件，然后在中断服务例程中把它发送给应用线程；但是在STM32 Radio中，由于硬件的限制，系统需要自行查询按键的状态，即不能够在中断服务的上下文中执行，所以应单独开辟一个key线程来处理按键。

线程的状态跃迁

这里说的状态跃迁指的是线程运行中状态的变化，从就绪态过渡到挂起态。实时系统一般被设计成一种优先级的系统，如果一个线程只有就绪态而无阻塞态，势必会影响到其他低优先级线程的执行。所以在进行线程设计时，就应该保证线程在不活跃的时候，必须让出

理器，即线程能够主动让出处理器资源，进入到阻塞状态。这需要设计者在设计线程的时候就明确的知道什么情况下需要让线程从就绪态跃迁到阻塞态。

线程运行时间长度

线程运行时间长度被定义为，在线程所关心的一种事件或多种事件触发状态下，线程由阻塞态跃迁为就绪态执行设定的工作，再从就绪态跃迁为阻塞态所需要的时间（一般还应加上这段时间内，这个线程不会被其它线程所抢占的先决条件）。线程运行时间长度将和线程的优先级设计密切相关，同时也决定着设计的系统是否能够满足预计的实时响应的指标。

例如，对于事件A对应的服务线程Ta，系统要求的实时响应指标是1ms，而Ta的最大运行时间是500us。此时，系统中还存在着以50ms为周期的另一线程Tb，它每次运行的最大时间长度是100us。在这种情况下，即使把线程Tb的优先级抬到比Ta更高的位置，对系统的实时性指标也没什么影响（因为即使在Ta的运行过程中，Tb抢占了Ta的资源，但在规定的时间内(1ms)，Ta也能够完成对事件A的响应）。

第 3 章

定时器

3.1 定时器管理

定时器，是指从指定的时刻开始，经过一个指定时间，然后触发一个事件，类似定个时间提醒第二天能够按时起床。定时器有硬件定时器和软件定时器之分：

- 硬件定时器是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，并且是中断触发方式。
- 软件定时器是由操作系统提供的一类系统接口（函数），它构建在硬件定时器基础之上，使系统能够提供不受数目限制的定时器服务。

在操作系统中，通常软件定时器以系统节拍（tick）为单位。节拍长度指的是周期性硬件定时器两次中断间的间隔时间长度。这个周期性硬件定时器也称之为操作系统时钟。软件定时器以这个节拍时间长度为单位，数值必须是这个节拍的整数倍，例如节拍是10ms，那么上层软件定时器只能是10ms，20ms，100ms等，而不能取值为15ms。由于节拍定义了系统中定时器能够分辨的精确度，系统可以根据实际系统CPU的处理能力和实时性需求设置合适的数值，tick值设置越小，精度越高，但是系统开销也将越大（在1秒中系统用于处理时钟中断的次数也就越多）。RT-Thread的定时器也基于类似的系统节拍，提供了基于节拍整数倍的定时能力。

RT-Thread的定时器提供两类定时器机制：第一类是单次触发定时器，这类定时器在启动后只会触发一次定时器事件，然后定时器自动停止。第二类是周期触发定时器，这类定时器会周期性的触发定时器事件，直到用户手动的停止定时器否则将永远持续执行下去。

下面以一个实际部分代码来说明RT-Thread软件定时器的基本工作原理。在RT-Thread定时器模块中维护着两个重要的全局变量：

- 当前系统经过的tick时间rt_tick（当硬件定时器中断来临时，它将加1）；
- 定时器链表rt_timer_list。系统新创建并激活的定时器都会按照以超时时间排序的方式插入到rt_timer_list链表中。

如上图所示，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，分别是定时时间为50个tick的Timer1、100个tick的Timer2和500个tick的Timer3，这三个定时器分别加上系统当前时间rt_tick = 20，从小到大排序链接在rt_timer_list链表中，形成如上图所示的定时器链表结构。

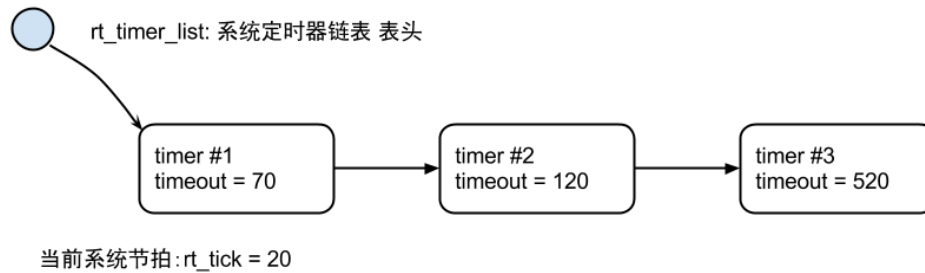


图 3.1: 定时器链表

而 rt_tick 随着硬件定时器的触发一直在增长（每一次硬件定时器中断来临， rt_tick 变量会加1），50个tick以后， rt_tick 从20增长到70，与Timer1的timeout值相等，这时会触发与Timer1定时器相关联的超时函数，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，与Timer2和Timer3定时器相关联的超时函数会被触发，接着将Timer2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在10个tick以后（ $rt_tick = 30$ ）有一个任务新创建了一个tick值为300的Timer4定时器，由于Timer4定时器的 $timeout = rt_tick + 300 = 330$ ，因此它将被插入到Timer2和Timer3定时器中间，形成如图3-2所示链表结构：

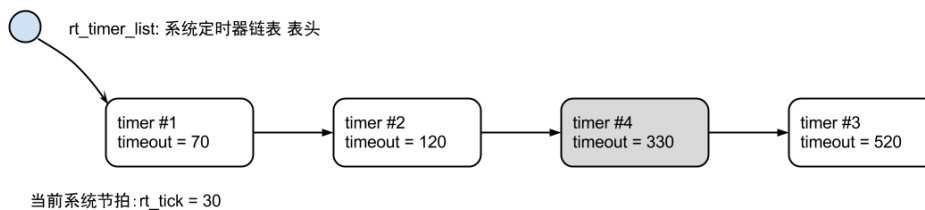


图 3.2: 插入Timer #4后的示意图

3.2 定时器超时函数

软定时器最主要的目的是在经过设定的时间后，系统能够自动执行用户设定的动作。当定时器设定的时间到了，即超时，执行的动作函数称之为定时器的超时函数。与线程不同的是，超时函数的执行上下文环境并未用显式给出。

在RT-Thread实时操作系统中，定时器超时函数存在着两种情况：

- 超时函数在（系统时钟）中断上下文环境中执行；
- 超时函数在线程的上下文环境中执行。

如果超时函数是在中断上下文环境中执行，显然对于超时函数的要求与中断服务例程的要求相同：执行时间应该尽量短，执行时不应导致当前上下文挂起、等待。例如在中断上下文中执行的超时函数它不应该试图去申请动态内存、释放动态内存等（其中一个就包括 rt_timer_delete 函数调用）。

而超时函数在线程上下文中执行，则不会有这个限制，但是通常也要求超时函数执行时间应该足够短，而不应该影响到其他定时器或本次定时器的下一次周期性超时。这两种情况在RT-Thread定时器控制块中分别使用参数： $RT_TIMER_FLAG_HARD_TIMER$ 和 $RT_TIMER_FLAG_SOFT_TIMER$ 指定。 $HARD_TIMER$ 代表的是定时器超时函数执行上下文是在

中断上下文环境中执行；SOFT_TIMER代表的是定时器函数执行的上下文是timer线程（在rtconfig.h头文件中应该定义宏RT_USING_TIMER_SOFT使timer线程能被使用）。

3.3 定时器管理控制块

```
struct rt_timer
{
    struct rt_object parent;

    rt_list_t row[RT_TIMER_SKIP_LIST_LEVEL]; /* 定时器列表算法用到的队列 */

    void (*timeout_func)(void *parameter); /* 定时器超时调用的函数 */
    void *parameter; /* 超时函数用到的入口参数 */

    rt_tick_t init_tick; /* 定时器初始超时节拍数 */
    rt_tick_t timeout_tick; /* 定时器实际超时时的节拍数 */
};
typedef struct rt_timer *rt_timer_t;
```

定时器控制块由struct rt_timer结构体定义，并形成定时器内核对象再链接到内核对象容器中进行管理。list成员则用于把一个激活的（已经启动的）定时器链接到rt_timer_list链表中。

3.4 定时器管理接口

3.4.1 定时器管理系统初始化

初始化定时器管理系统，可以通过下面的函数接口完成：

```
void rt_system_timer_init(void);
```

函数参数

无

函数返回

无

如果需要使用SOFT_TIMER，则系统初始化时，应该调用下面这个函数接口：

```
void rt_system_timer_thread_init(void);
```

函数参数

无

函数返回

无

3.4.2 创建定时器

当动态创建一个定时器时，可使用下面的函数接口：

```
rt_timer_t rt_timer_create(const char* name,
                           void (*timeout)(void* parameter), void* parameter,
                           rt_tick_t time, rt_uint8_t flag);
```

调用该函数接口后，内核首先从动态内存堆中分配一个定时器控制块，然后对该控制块进行基本的初始化。

函数参数

参数	描述
const char* name	定时器的名称；
void (timeout)(void parameter)	定时器超时函数指针（当定时器超时时，系统会调用这个函数）；
void* parameter	定时器超时函数的入口参数（当定时器超时时，调用超时回调函数 会把这个参数做为入口参数传递给超时函数）；
rt_tick_t time	定时器的超时时间，单位是系统节拍；
rt_uint8_t flag	定时器创建时的参数，支持的值包括（可以用“或”关系取多个值）；

include/rtddef.h中定义了一些定时器相关的宏，如下。

```
#define RT_TIMER_FLAG_DEACTIVATED 0x0    /* 定时器为非激活态 */
#define RT_TIMER_FLAG_ACTIVATED   0x1    /* 定时器为激活状态 */
#define RT_TIMER_FLAG_ONE_SHOT    0x0    /* 单次定时 */
#define RT_TIMER_FLAG_PERIODIC    0x2    /* 周期定时 */
#define RT_TIMER_FLAG_HARD_TIMER  0x0    /* 硬件定时器 */
#define RT_TIMER_FLAG_SOFT_TIMER  0x4    /* 软件定时器 */
```

当指定的flag为RT_TIMER_FLAG_HARD_TIMER时，如果定时器超时，定时器的回调函数将在时钟中断的服务例程上下文中被调用；当指定的flag为RT_TIMER_FLAG_SOFT_TIMER时，如果定时器超时，定时器的回调函数将在系统时钟timer线程的上下文被调用。

函数返回

如果定时器创建成功，则返回定时器的句柄；如果创建失败，会返回RT_NULL（通常会由于系统内存不够用而返回RT_NULL）。

创建定时器的例子如下所示：

```
/*
 * 程序清单：动态定时器例程
 *
 * 这个例程会创建两个动态定时器对象，一个是单次定时，一个是周期性的定时
```



```

*/
#include <rtthread.h>

/* 定时器的控制块 */
static rt_timer_t timer1;
static rt_timer_t timer2;

/* 定时器1超时函数 */
static void timeout1(void* parameter)
{
    rt_kprintf("periodic timer is timeout\n");
}

/* 定时器2超时函数 */
static void timeout2(void* parameter)
{
    rt_kprintf("one shot timer is timeout\n");
}

int rt_application_init(void)
{
    /* 创建定时器1 */
    timer1 = rt_timer_create("timer1", /* 定时器名字是 timer1 */
                             timeout1, /* 超时时回调的处理函数 */
                             RT_NULL, /* 超时函数的入口参数 */
                             10,      /* 定时长度, 以OS Tick为单位, 即10个OS Tick */
                             RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */

    /* 启动定时器 */
    if (timer1 != RT_NULL) rt_timer_start(timer1);

    /* 创建定时器2 */
    timer2 = rt_timer_create("timer2", /* 定时器名字是 timer2 */
                             timeout2, /* 超时时回调的处理函数 */
                             RT_NULL, /* 超时函数的入口参数 */
                             30,      /* 定时长度为30个OS Tick */
                             RT_TIMER_FLAG_ONE_SHOT); /* 单次定时器 */

    /* 启动定时器 */
    if (timer2 != RT_NULL) rt_timer_start(timer2);

    return 0;
}

```

3.4.3 删除定时器

系统不再使用特定定时器时, 可使用下面的函数接口:

```
rt_err_t rt_timer_delete(rt_timer_t timer);
```

调用这个函数接口后，系统会把这个定时器从rt_timer_list链表中删除，然后释放相应的定时器控制块占有的内存。

函数参数

参数	描述
rt_timer_t timer	定时器句柄，指向要删除的定时器。

函数返回

返回RT_EOK（如果参数timer句柄是一个RT_NULL，将会导致一个ASSERT断言）
删除定时器的例子请参考例9-1中删除定时器的代码。

3.4.4 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，函数接口如下：

```
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);
```

使用该函数接口时会初始化相应的定时器控制块，初始化相应的定时器名称，定时器超时函数等等。

函数参数

参数	描述
rt_timer_t timer	定时器句柄，指向要初始化的定时器控制块；
const char* name	定时器的名称；
void (timeout)(void parameter)	定时器超时函数指针（当定时器超时时，系统会调用这个函数）；
void* parameter	定时器超时函数的入口参数（当定时器超时时，调用超时回调函数 会把这个参数做为入口参数传递给超时函数）；
rt_tick_t time	定时器的超时时间，单位是系统节拍；
rt_uint8_t flag	定时器创建时的参数，支持的值包括（可以用“或”关系取多个值）；

```
#define RT_TIMER_FLAG_ONE_SHOT    0x0    /* 单次定时    */
#define RT_TIMER_FLAG_PERIODIC    0x2    /* 周期定时    */
```

```
#define RT_TIMER_FLAG_HARD_TIMER    0x0    /* 硬件定时器 */
#define RT_TIMER_FLAG_SOFT_TIMER    0x4    /* 软件定时器 */
```

当指定的flag为RT_TIMER_FLAG_HARD_TIMER时，如果定时器超时，定时器的回调函数将在时钟中断的服务例程上下文中被调用；当指定的flag为RT_TIMER_FLAG_SOFT_TIMER时，如果定时器超时，定时器的回调函数将在系统时钟timer线程的上下文中被调用。

初始化定时器的例子如下代码所示：

```
/*
 * 程序清单：静态定时器例程
 *
 * 这个程序会初始化2个静态定时器，一个是单次定时，一个是周期性的定时
 */
#include <rtthread.h>

/* 定时器的控制块 */
static struct rt_timer timer1;
static struct rt_timer timer2;

/* 定时器1超时函数 */
static void timeout1(void* parameter)
{
    rt_kprintf("periodic timer is timeout\n");
}

/* 定时器2超时函数 */
static void timeout2(void* parameter)
{
    rt_kprintf("one shot timer is timeout\n");
}

int rt_application_init(void)
{
    /* 初始化定时器 */
    rt_timer_init(&timer1, "timer1", /* 定时器名字是 timer1 */
        timeout1, /* 超时回调的处理函数 */
        RT_NULL, /* 超时函数的入口参数 */
        10, /* 定时长度，以OS Tick为单位，即10个OS Tick */
        RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */
    rt_timer_init(&timer2, "timer2", /* 定时器名字是 timer2 */
        timeout2, /* 超时回调的处理函数 */
        RT_NULL, /* 超时函数的入口参数 */
        30, /* 定时长度为30个OS Tick */
        RT_TIMER_FLAG_ONE_SHOT); /* 单次定时器 */

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

```

    rt_timer_start(&timer2);

    return 0;
}

```

3.4.5 脱离定时器

当一个静态定时器不需要再使用时，可以使用下面的函数接口：

```
rt_err_t rt_timer_detach(rt_timer_t timer);
```

脱离定时器时，系统会把定时器对象从系统容器的定时器链表中删除，但是定时器对象所占有的内存不会被释放。

函数参数

参数	描述
rt_timer_t timer	定时器句柄，指向要脱离的定时器控制块。

函数返回

返回RT_EOK。

脱离定时器的例子可参考9-2例程代码中的脱离部分。

3.4.6 启动定时器

当定时器被创建或者初始化以后，并不会被立即启动，必须在调用启动定时器函数接口后，才开始工作，启动定时器函数接口如下：

```
rt_err_t rt_timer_start(rt_timer_t timer);
```

调用定时器启动函数接口后，定时器的状态将更改为激活状态（RT_TIMER_FLAG_ACTIVATED），并按照超时顺序插入到rt_timer_list队列链表中。

函数参数

参数	描述
rt_timer_t timer	定时器句柄，指向要启动的定时器控制块。

函数返回

如果timer已经处于激活状态，则返回-RT_ERROR；否则返回RT_EOK。

启动定时器的例子请参考9-1例程代码中的定时器代码。

3.4.7 停止定时器

启动定时器以后，若想使它停止，可以使用下面的函数接口：

```
rt_err_t rt_timer_stop(rt_timer_t timer);
```

调用定时器停止函数接口后，定时器状态将更改为停止状态，并从rt_timer_list链表中脱离出来不参与定时器超时检查。当一个（周期性）定时器超时时，也可以调用这个函数接口停止这个（周期性）定时器本身。

函数参数

参数	描述
rt_timer_t timer	定时器句柄，指向要停止的定时器控制块。

函数返回

如果timer已经处于停止状态，返回-RT_ERROR；否则返回RT_EOK。

3.4.8 控制定时器

除了上述提供的一些编程接口，RT_thread也额外提供了定时器控制函数接口，以获取或设置更多定时器的信息。控制定时器函数接口如下：

```
rt_err_t rt_timer_control(rt_timer_t timer, rt_uint8_t cmd, void* arg);
```

控制定时器函数接口可根据命令类型参数，来查看或改变定时器的设置。

函数参数

参数	描述
rt_timer_t timer	定时器句柄，指向要进行控制的定时器控制块;
rt_uint8_t cmd	用于控制定时器的命令，当前支持四个命令接口，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;
void* arg	与command相对应的控制命令参数;

```
#define RT_TIMER_CTRL_SET_TIME      0x0    /* 设置定时器超时时间 */
#define RT_TIMER_CTRL_GET_TIME      0x1    /* 获得定时器超时时间 */
#define RT_TIMER_CTRL_SET_ONESHOT   0x2    /* 设置定时器为单一超时型 */
#define RT_TIMER_CTRL_SET_PERIODIC  0x3    /* 设置定时器为周期型定时器 */
```

函数返回

函数返回RT_EOK

使用定时器控制接口的代码如下所示:

```

/*
 * 程序清单：定时器控制接口示例
 *
 * 这个例程会创建1个动态周期型定时器对象，然后控制它进行更改定时器的时间长度。
 */
#include <rtthread.h>

/* 定时器的控制块 */
static rt_timer_t timer1;
static rt_uint8_t count;

/* 定时器超时函数 */
static void timeout1(void* parameter)
{
    rt_kprintf("periodic timer is timeout\n");

    count ++;
    /* 当超过8次时，更改定时器的超时长度 */
    if (count >= 8)
    {
        int timeout_value = 50;
        /* 控制定时器更改定时器超时时间长度 */
        rt_timer_control(timer1, RT_TIMER_CTRL_SET_TIME, (void*)&timeout_value);
        count = 0;
    }
}

int rt_application_init(void)
{
    /* 创建定时器1 */
    timer1 = rt_timer_create("timer1", /* 定时器名字是 timer1 */
                             timeout1, /* 超时回调的处理函数 */
                             RT_NULL, /* 超时函数的入口参数 */
                             10, /* 定时长度，以OS Tick为单位，即10个OS Tick */
                             RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */

    /* 启动定时器 */
    if (timer1 != RT_NULL)
        rt_timer_start(timer1);

    return 0;
}

```

3.5 合理使用定时器

3.5.1 定时器执行上下文

RT-Thread的定时器与其他实时操作系统的定时器实现稍微有些不同（特别是RT-Thread早期版本的实现中），因为RT-Thread里定时器默认的方式是HARD_TIMER定时器，即定时器超时后，超时函数是在系统时钟中断的上下文环境中运行的。在中断上下文中的执行方式决定了定时器的超时函数不应该调用任何会让当前上下文挂起的系统函数；也不能够执行非常长的时间，否则会导致其他中断的响应时间加长或抢占了其他线程执行的时间。

另外，在第二章第三节线程控制块中，你是否有留意到每个线程控制块中都包含了一个定时器：thread_timer。这个thread_timer也是一个HARD_TIMER定时器。它被用于当线程需要执行一些带时间特性的系统调用中，例如带超时特性的试图持有信号量，接收事件、接收消息等，而当相应的条件不能够被满足时线程就将被挂起，在线程挂起前，这个内置的定时器将会被激活并启动（超时函数设定为rt_thread_timeout）。当线程定时器超时时，这个线程依然还未被唤醒，rt_thread_timeout函数仍将继续被调用，接着设置线程的error代码为-RT_ETIMEOUT，接着唤醒这个线程。所以从某个意义上说，在线程中执行rt_thread_sleep/rt_thread_delay函数，也可以算是另一种意义的超时。

回到上一段对HARD_TIMER定时器描述中来，可以看到HARD_TIMER定时器超时函数工作于中断的上下文环境中，这种在中断中执行的方式显得非常麻烦，因此开发人员需要时刻关心超时函数究竟执行了哪些操作；相反如果定时器超时函数是在线程中执行，显然会好很多，如果有更高优先级的线程就绪，依然可以抢占这个定时器执行线程从而获得优先处理权。如果是想要使用rt_thread_sleep/rt_thread_delay的方式实现定时器超时操作，那么可以使用如下图的方式：

```
void timer_thread(void* parameter)
{
    while (1)
    {
        /* ... */
        /* 进行一个n个OS Tick的延迟 */
        rt_thread_delay(n);

        /* 超时达到后, 执行相应的动作处理 */
    }
}
```

图 3.3: 线程定时器

在上面的例子中，timer_thread是一个线程入口函数，在线程中执行rt_thread_delay(n)后，可以实现n个OS tick的定时，当执行rt_thread_delay时，线程的内置定时器将会被激活并启动；当线程定时器超时时，这个线程将被唤醒，并接着rt_thread_delay运行后续的代码。

上述描述的都是HARD_TIMER的特性。另外，在RT-Thread中，我们也可以在创建定时器时，把定时器指定成SOFT_TIMER的方式，这样可以使得定时器超时函数完全运行在timer系统线程上下文环境中。如果系统在初始化时需要使用SOFT_TIMER特性，需要在系统配置中打开RT_USING_TIMER_SOFT宏定义，那么调用rt_system_timer_thread_init函数就可以启动timer系统线程。这里值得注意的是，SOFT_TIMER定时器的精度由RT_TIMER_TICK_PER_SECOND定义的值所决定（每秒触发的timer tick次数是多少），这个值必须是OS tick的整数倍。

3.5.2 OS tick与定时器精度

系统中HARD_TIMER定时器的最小精度是由系统时钟节拍所决定的（1 OS tick = 1/RT_TICK_PERSECOND秒，RT_TICK_PER_SECOND值在rtconfig.h文件中定义），定时器设定的时间必须是OS tick的整数倍。当需要实现更短时间长度的系统定时时，例如OS tick是10ms，而程序需要实现1ms的定时或延时，这种时候操作系统定时器将不能够满足要求，只能通过读取系统某个硬件定时器的计数器或直接使用硬件定时器的方式。

在Cortex-M3中，SysTick已经被RT-Thread用于作为OS tick使用，它被配置成1/RT_TICK_PER_SECOND秒后触发一次中断的方式，中断处理函数使用Cortex-M3默认的SysTick_Handler名字。在Cortex-M3的CMSIS（Cortex Microcontroller Software Interface Standard）规范中规定了SystemCoreClock代表芯片的主频，所以基于SysTick以及SystemCoreClock，我们能够使用SysTick获得一个精确的延时函数，如下例所示，Cortex-M3上的基于SysTick的精确延时（需要系统在使能SysTick后使用）：

高精度延时的例程如下所示

```
#include <core_cm3.h>
void rt_hw_us_delay(int us)
{
    rt_uint32_t delta;

    /* 获得延时经过的tick数 */
    us = us * (SysTick->LOAD/(1000000/RT_TICK_PER_SECOND));

    /* 获得当前时间 */
    delta = SysTick->VAL;

    /* 循环获得当前时间，直到达到指定的时间后退出循环 */
    while (delta - SysTick->VAL < us);
}
```

其中入口参数us指示出需要延时的微秒数目，这个函数只能支持低于1 OS tick的延时，否则SysTick会出现溢出而不能够获得指定的延时时间。

第 4 章

任务间同步及通信

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协调的方式共同来完成，例如一个任务从传感器中接收数据并且将数据写到共享内存中，同时另一个任务周期性的从共享内存中读取数据并发送去显示（如图 两个线程间的数据传递）。

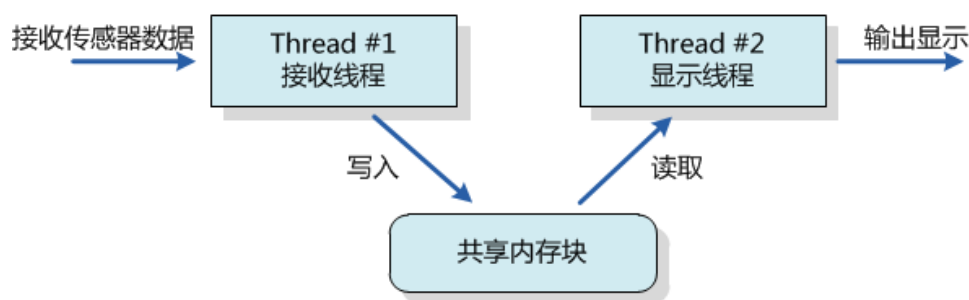


图 4.1: 两个线程间的数据传递

如果对共享内存的访问不是排他性的，那么各个线程间可能同时访问它。这将引起数据一致性的问题，例如，在显示线程试图显示数据之前，传感器线程还未完成数据的写入，那么显示将包含不同时间采样的数据，造成显示数据的迷惑。

将传感器数据写入到共享内存的代码是接收线程的关键代码段；将传感器数据从共享内存中读出的代码是显示线程的关键代码段；这两段代码都会访问共享内存。正常的操作序列应该是在一个线程对共享内存块操作完成后，才允许另一个线程去操作。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种，其核心思想都是：在访问临界区的时候只允许一个(或一类)任务运行。

4.1 关闭中断

关闭中断也叫中断锁，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。当中断关闭的时候，就意味着当前任务不会被其他事件打断（因为整个系统已经不再响应那些可以触发线程重新调度的外部事件），也就是当前线程不会被抢占，除非这个任务主动放弃了处理器控制权。关闭中断/恢复中断API接口由BSP实现，根据平台的不同其实现方式也大不相同。

关闭、打开中断接口由两个函数完成：

- 关闭中断

```
rt_base_t rt_hw_interrupt_disable(void);
```

这个函数用于关闭中断并返回关闭中断前的中断状态。

函数参数

无

函数返回

返回调用这个函数前的中断状态。

- 恢复中断

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

这个函数“使能”中断，它采用恢复调用rt_hw_interrupt_disable()函数前的中断状态，进行“使能”中断状态，如果调用rt_hw_interrupt_disable() 函数前是关中断状态，那么调用此函数后依然是关中断状态。level参数是上一次调用rt_hw_interrupt_disable()时的返回值。

函数参数

参数	描述
level	前一次rt_hw_interrupt_disable返回的中断状态。

函数返回

无

使用开关中断进行线程间同步的例子代码如下例代码所示：

```
/* 代码清单：关闭中断进行全局变量的访问 */
#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_t) 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t1线程 */
    thread = rt_thread_create("t1", thread_entry, (void*)10,
        512, 10, 5);
    if (thread != RT_NULL) rt_thread_startup(thread);

    /* 创建t2线程 */
    thread = rt_thread_create("t2", thread_entry, (void*)20,
        512, 20, 5);
    if (thread != RT_NULL) rt_thread_startup(thread);
}

```

- 警告: 由于关闭中断会导致整个系统不能响应外部中断, 所以在使用关闭中断做为互斥访问临界区的手段时, 首先必须需要保证关闭中断的时间非常短, 例如数条机器指令。

4.1.1 使用场合

使用中断锁来操作系统的方法可以应用于任何场合, 且其他几类同步方式都是依赖于中断锁而实现的, 可以说中断锁是最强大的和最高效的同步方法。只是使用中断锁最主要的问题在于, 在中断关闭期间系统将不再响应任何中断, 也就不能响应外部的事件。所以中断锁对系统的实时性影响非常巨大, 当使用不当的时候会导致系统完全无实时性可言 (可能导致系统完全偏离要求的时间需求); 而使用得当, 则会变成一种快速、高效的同步方式。

例如, 为了保证一行代码 (例如赋值) 的互斥运行, 最快速的方法是使用中断锁而不是信号量或互斥量:

```

/* 关闭中断*/
level = rt_hw_interrupt_disable();
a = a + value;
/* 恢复中断*/
rt_hw_interrupt_enable(level);

```

在使用中断锁时, 需要确保关闭中断的时间非常短, 例如上面代码中的 `a = a + value;` 也可换成另外一种方式, 例如使用信号量:

```

/* 获得信号量锁*/
rt_sem_take(sem_lock, RT_WAITING_FOREVER);
a = a + value;
/* 释放信号量锁*/
rt_sem_release(sem_lock);

```

这段代码在 `rt_sem_take`、`rt_sem_release` 的实现中, 已经存在使用中断锁保护信号量内部变量的行为, 所以对于简单如 `a = a + value;` 的操作, 使用中断锁将更为简洁快速。

4.2 调度器锁

同中断锁一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。但和中断锁有一点不相同的是，对调度器上锁，系统依然能响应外部中断，中断服务例程依然能进行相应的响应。所以在使用调度器上锁的方式进行任务同步时，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进行同步。RT-Thread提供的调度锁操作API为：

```
void rt_enter_critical(void); /* 进入临界区*/
```

函数参数

无

函数返回

无

调用这个函数后，调度器将被上锁。在锁住调度器期间，系统依然响应中断，如果中断唤醒了更高优先级的线程，调度器并不会立刻执行它，直到调用解锁调度器函数时才会尝试进行下一次调度。

```
void rt_exit_critical(void); /* 退出临界区*/
```

函数参数

无

函数返回

无

当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线程就绪，如果有比当前线程更高优先级的线程就绪，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，将继续执行当前任务。

- 注：rt_enter_critical/rt_exit_critical可以多次嵌套调用，但每调用一次rt_enter_critical就必须相对地调用一次rt_exit_critical退出操作，嵌套的最大深度是65535。

4.2.1 使用场合

调度器锁能够方便地使用于一些线程与线程间同步的场合，由于轻型，它不会对系统中断响应造成负担；但它的缺陷也很明显，就是它不能被用于中断与线程间的同步或通知，并且如果执行调度器锁的时间过长，会对系统的实时性造成影响（因为使用了调度器锁后，系统将不再具备优先级的关系，直到它脱离了调度器锁的状态）。

4.3 信号量

信号量是一种轻型的用于解决线程间同步问题的内核对象，线程可以获取或释放它，从而达到同步或互斥的目的。信号量就像一把钥匙，把一段临界区给锁住，只允许有钥匙的线程进行访问：线程拿到了钥匙，才允许它进入临界区；而离开后把钥匙传递给排队在后面的等待线程，让后续线程依次进入临界区。

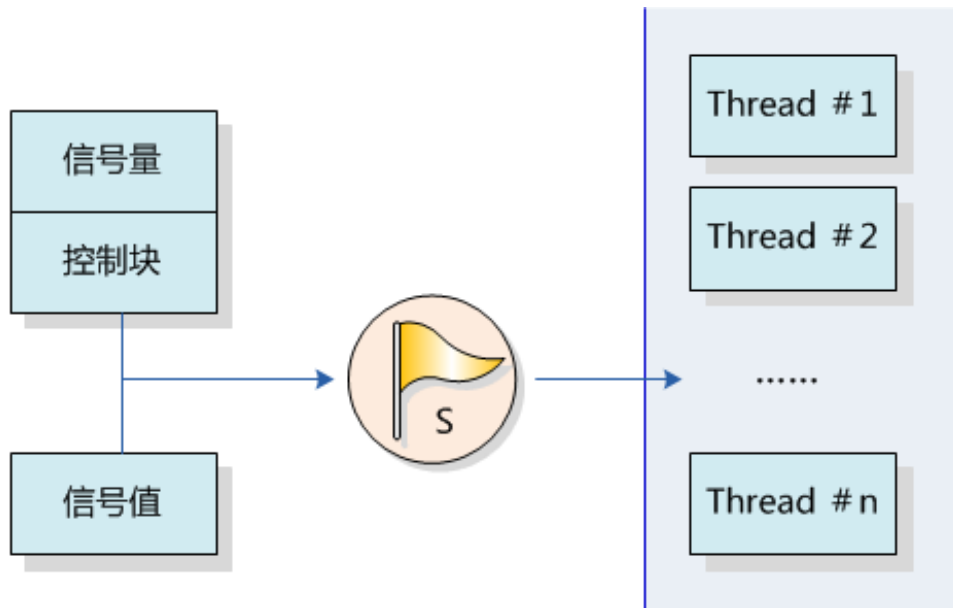


图 4.2: 信号量工作示意图

信号量工作示意图如图 信号量工作示意图 所示，每个信号量对象都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、资源数目，假如信号量值为 5，则表示共有 5 个信号量实例（资源）可以被使用，当信号量实例数目为零时，再申请该信号量的线程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（资源）。

4.3.1 信号量控制块

```
struct rt_semaphore
{
    struct rt_ipc_object parent; /*继承自ipc_object类*/
    rt_uint16_t value; /* 信号量的值 */
};
/* rt_sem_t是指向semaphore结构体的指针类型 */
typedef struct rt_semaphore* rt_sem_t;
```

rt_semaphore对象从rt_ipc_object中派生，由IPC容器所管理。信号量的最大值是 65535。

4.3.2 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，创建信号量使用下面的函数接口：

```
rt_sem_t rt_sem_create (const char* name, rt_uint32_t value, rt_uint8_t flag);
```

当调用这个函数时，系统将先分配一个semaphore对象，并初始化这个对象，然后初始化IPC对象以及与semaphore相关的部分。在创建信号量指定的参数中，信号量标志参数决定了当信号量不可用时，多个线程等待的排队方式。当选择FIFO方式时，那么等待线程队列将按照先进先出的方式排队，先进入的线程将先获得等待的信号量；当选择PRIO（优先级等待）方式时，等待线程队列将按照优先级进行排队，优先级高的等待线程将先获得等待的信号量。

函数参数

参数	描述
name	信号量名称；
value	信号量初始值；
flag	信号量标志，取值可以使用如下类型：

```
#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*/
```

函数返回

创建成功返回创建的信号量控制块指针；否则返回RT_NULL。
创建信号量的例程如下例所示：

```
/*
 * 程序清单：动态信号量
 */
/* 这个例子中将创建一个动态信号量(初始值为0)及一个动态线程，在这个动态线程中
 * 将试图采用超时方式去持有信号量，应该超时返回。然后这个线程释放一次信号量，
 * 并在后面继续采用永久等待方式去持有信号量，成功获得信号量后返回。
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 指向线程控制块的指针 */
static rt_thread_t tid = RT_NULL;
/* 指向信号量的指针 */
static rt_sem_t sem = RT_NULL;
/* 线程入口 */
static void thread_entry(void* parameter)
{
    rt_err_t result;
    rt_tick_t tick;

    /* 获得当前的OS Tick */
    tick = rt_tick_get();

    /* 试图持有一个信号量，如果10个OS Tick依然没拿到，则超时返回 */
    result = rt_sem_take(sem, 10);
```

```

if (result == -RT_ETIMEOUT)
{
    /* 判断是否刚好过去10个OS Tick */
    if (rt_tick_get() - tick != 10)
    {
        /* 如果失败，则测试失败 */
        tc_done(TC_STAT_FAILED);
        rt_sem_delete(sem);
        return;
    }
    rt_kprintf("take semaphore timeout\n");
}
else
{
    /* 因为并没释放信号量，应该是超时返回，否则测试失败 */
    tc_done(TC_STAT_FAILED);
    rt_sem_delete(sem);
    return;
}

/* 释放一次信号量 */
rt_sem_release(sem);

/* 继续持有信号量，并永远等待直到持有到信号量 */
result = rt_sem_take(sem, RT_WAITING_FOREVER);
if (result != RT_EOK)
{
    /* 返回不正确，测试失败 */
    tc_done(TC_STAT_FAILED);
    rt_sem_delete(sem);
    return;
}

/* 测试成功 */
tc_done(TC_STAT_PASSED);
/* 删除信号量 */
rt_sem_delete(sem);
}

int semaphore_dynamic_init()
{
    /* 创建一个信号量，初始值是0 */
    sem = rt_sem_create("sem", 0, RT_IPC_FLAG_FIFO);
    if (sem == RT_NULL)
    {

```

```

        tc_stat(TC_STAT_END | TC_STAT_FAILED);
        return 0;
    }

    /* 创建线程 */
    tid = rt_thread_create("thread",
        thread_entry, RT_NULL, /* 线程入口是thread_entry, 参数RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid != RT_NULL)
        rt_thread_startup(tid);
    else
        tc_stat(TC_STAT_END | TC_STAT_FAILED);

    return 0;
}

#ifdef RT_USING_TC
static void _tc_cleanup()
{
    /* 调度器上锁, 上锁后, 将不再切换到其他线程, 仅响应中断 */
    rt_enter_critical();

    /* 删除线程 */
    if (tid != RT_NULL && tid->stat != RT_THREAD_CLOSE)
    {
        rt_thread_delete(tid);

        /* 删除信号量 */
        rt_sem_delete(sem);
    }

    /* 调度器解锁 */
    rt_exit_critical();

    /* 设置TestCase状态 */
    tc_done(TC_STAT_PASSED);
}

int _tc_semaphore_dynamic()
{
    /* 设置TestCase清理回调函数 */
    tc_cleanup(_tc_cleanup);
    semaphore_dynamic_init();

    /* 返回TestCase运行的最长时间 */

```



```
        return 100;
    }
    /* 输出函数命令到finsh shell中 */
    FINSH_FUNCTION_EXPORT(_tc_semaphore_dynamic, a dynamic semaphore example);
    #else
    /* 用户应用入口 */
    int rt_application_init()
    {
        semaphore_dynamic_init();

        return 0;
    }
    #endif
```

删除信号量

系统不再使用信号量时，可通过删除信号量以释放系统资源。删除信号量使用下面的函数接口：

```
rt_err_t rt_sem_delete (rt_sem_t sem);
```

调用这个函数时，系统将删除这个信号量。如果删除该信号量时，有线程正在等待该信号量，那么删除操作会先唤醒等待在该信号量上的线程（等待线程的返回值是-RT_ERROR），然后再释放信号量的内存资源。

参数	描述
sem	rt_sem_create创建处理的信号量对象。

函数返回
RT_EOK

初始化信号量

对于静态信号量对象，它的内存空间在编译时期就被编译器分配出来，放在数据段或ZI段上，此时使用信号量就不再需要使用rt_sem_create接口来创建它，而只需在使用前对它进行初始化即可。初始化信号量对象可使用下面的函数接口：

```
rt_err_t rt_sem_init (rt_sem_t sem, const char* name, rt_uint32_t value, rt_uint8_t flag);
```

当调用这个函数时，系统将对这个semaphore对象进行初始化，然后初始化IPC对象以及与semaphore相关的部分。在初始化信号量指定的参数中，信号量标志参数决定了当信号量不可用时，多个线程等待的方式。当选择FIFO方式时，那么等待线程队列将按照先进先出的方式排队，先进入的线程将先获得等待的信号量；当选择PRIO（优先级等待）方式时，等待线程队列将按照优先级进行排队，优先级高的等待线程将先获得等待的信号量。

函数参数

参数	描述
sem	信号量对象的句柄;
name	信号量名称;
value	信号量初始值;
flag	信号量标志。

```
#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*/
```

函数返回

RT_EOK

初始化信号量的例程如下例所示:

```
/*
 * 程序清单：静态信号量
 */
/* 这个例子中将创建一个静态信号量（初始值为0）及一个静态线程，在这个静态线程中
 * 将试图采用超时方式去获取信号量，应该超时返回。然后这个线程释放一次信号量，并
 * 在后面继续采用永久等待方式去获取信号量，成功获得信号量后返回。
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 线程控制块及栈 */
static struct rt_thread thread;
static rt_uint8_t thread_stack[THREAD_STACK_SIZE];
/* 信号量控制块 */
static struct rt_semaphore sem;

/* 线程入口 */
static void thread_entry(void* parameter)
{
    rt_err_t result;
    rt_tick_t tick;

    /* 获得当前的OS Tick */
    tick = rt_tick_get();

    /* 试图持有信号量，最大等待10个OS Tick后返回 */
    result = rt_sem_take(&sem, 10);
    if (result == -RT_ETIMEOUT)
    {

```

```

        /* 超时后判断是否刚好是10个OS Tick */
        if (rt_tick_get() - tick != 10)
        {
            tc_done(TC_STAT_FAILED);
            rt_sem_detach(&sem);
            return;
        }
        rt_kprintf("take semaphore timeout\n");
    }
else
{
    /* 因为没有其他地方释放信号量，所以不应该成功持有信号量，否则测试失败 */
    tc_done(TC_STAT_FAILED);
    rt_sem_detach(&sem);
    return;
}

/* 释放一次信号量 */
rt_sem_release(&sem);

/* 永久等待方式持有信号量 */
result = rt_sem_take(&sem, RT_WAITING_FOREVER);
if (result != RT_EOK)
{
    /* 不成功则测试失败 */
    tc_done(TC_STAT_FAILED);
    rt_sem_detach(&sem);
    return;
}

/* 测试通过 */
tc_done(TC_STAT_PASSED);
/* 脱离信号量对象 */
rt_sem_detach(&sem);
}

int semaphore_static_init()
{
    rt_err_t result;

    /* 初始化信号量，初始值是0 */
    result = rt_sem_init(&sem, "sem", 0, RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
    {
        tc_stat(TC_STAT_END | TC_STAT_FAILED);
    }
}

```

```

        return 0;
    }

    /* 初始化线程1 */
    result = rt_thread_init(&thread, "thread", /* 线程名: thread */
        thread_entry, RT_NULL, /* 线程的入口是thread_entry, 参数是RT_NULL*/
        &thread_stack[0], sizeof(thread_stack), /* 线程栈thread_stack */
        THREAD_PRIORITY, 10);
    if (result == RT_EOK) /* 如果返回正确, 启动线程1 */
        rt_thread_startup(&thread);
    else
        tc_stat(TC_STAT_END | TC_STAT_FAILED);

    return 0;
}

#ifdef RT_USING_TC
static void _tc_cleanup()
{
    /* 调度器上锁, 上锁后, 将不再切换到其他线程, 仅响应中断 */
    rt_enter_critical();

    /* 执行线程脱离 */
    if (thread.stat != RT_THREAD_CLOSE)
    {
        rt_thread_detach(&thread);

        /* 执行信号量对象脱离 */
        rt_sem_detach(&sem);
    }

    /* 调度器解锁 */
    rt_exit_critical();

    /* 设置TestCase状态 */
    tc_done(TC_STAT_PASSED);
}

int _tc_semaphore_static()
{
    /* 设置TestCase清理回调函数 */
    tc_cleanup(_tc_cleanup);
    semaphore_static_init();

    /* 返回TestCase运行的最长时间 */

```

```
        return 100;
    }
    /* 输出函数命令到finsh shell中 */
    FINSH_FUNCTION_EXPORT(_tc_semaphore_static, a static semaphore example);
#else
    /* 用户应用入口 */
    int rt_application_init()
    {
        thread_static_init();

        return 0;
    }
#endif
```

脱离信号量

脱离信号量就是让信号量对象从内核对象管理器中移除掉。脱离信号量使用下面的函数接口：

```
rt_err_t rt_sem_detach (rt_sem_t sem);
```

使用该函数后，内核先唤醒所有挂在该信号量等待队列上的线程，然后将该信号量从内核对象管理器中删除。原来挂起在信号量上的等待线程将获得-RT_ERROR 的返回值。

参数	描述
sem	信号量对象的句柄。

函数返回

RT_EOK

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，线程将获得信号量，并且相应的信号量值都会减1，获取信号量使用下面的函数接口：

```
rt_err_t rt_sem_take (rt_sem_t sem, rt_int32_t time);
```

在调用这个函数时，如果信号量的值等于零，那么说明当前信号量资源实例不可用，申请该信号量的线程将根据time参数的情况选择直接返回、或挂起等待一段时间、或永久等待，直到其他线程或中断释放该信号量。如果在参数time指定的时间内依然得不到信号量，线程将超时返回，返回值是-RT_ETIMEOUT。

函数参数

参数	描述
sem	信号量对象的句柄；
time	指定的等待时间，单位是操作系统时钟节拍（OS Tick）。

函数返回

成功获得信号量返回RT_EOK；超时依然未获得信号量返回-RT_ETIMEOUT；其他错误返回-RT_ERROR。

无等待获取信号量

当用户不想在申请的信号量上挂起线程进行等待时，可以使用无等待方式获取信号量，无等待获取信号量使用下面的函数接口：

```
rt_err_t rt_sem_trytake(rt_sem_t sem);
```

这个函数与rt_sem_take(sem, 0)的作用相同，即当线程申请的信号量资源实例不可用的时候，它不会等待在该信号量上，而是直接返回-RT_ETIMEOUT。

函数参数

参数	描述
sem	信号量对象的句柄。

函数返回

成功获取信号量返回RT_EOK；否则返回RT_ETIMEOUT。

释放信号量

当线程完成资源的访问后，应尽快释放它持有的信号量，使得其他线程能获得该信号量。释放信号量使用下面的函数接口：

```
rt_err_t rt_sem_release(rt_sem_t sem);
```

当信号量的值等于零时，并且有线程等待这个信号量时，将唤醒等待在该信号量线程队列中的第一个线程，由它获取信号量。否则将把信号量的值加一。

函数参数

参数	描述
sem	信号量对象的句柄。

函数返回

RT_EOK

下面是一个使用信号量的例程，如下例所示：

```

/*
 * 程序清单：生产者消费者例子
 *
 * 这个例子中将创建两个线程用于实现生产者消费者问题
 */
#include <rtthread.h>
#include "tc_comm.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];
/* 指向生产者、消费者在array数组中的读写位置 */
static rt_uint32_t set, get;

/* 指向线程控制块的指针 */
static rt_thread_t producer_tid = RT_NULL;
static rt_thread_t consumer_tid = RT_NULL;

struct rt_semaphore sem_lock;
struct rt_semaphore sem_empty, sem_full;

/* 生产者线程入口 */
void producer_thread_entry(void* parameter)
{
    rt_int32_t 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 = 0;

    /* 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");
}

```



```

}

int semaphore_producer_consumer_init()
{
    /* 初始化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);

    /* 创建线程1 */
    producer_tid = rt_thread_create("producer",
        producer_thread_entry, /* 线程入口是producer_thread_entry */
        RT_NULL, /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    if (producer_tid != RT_NULL)
        rt_thread_startup(producer_tid);
    else
        tc_stat(TC_STAT_END | TC_STAT_FAILED);

    /* 创建线程2 */
    consumer_tid = rt_thread_create("consumer",
        consumer_thread_entry, /* 线程入口是consumer_thread_entry */
        RT_NULL, /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY + 1, THREAD_TIMESLICE);
    if (consumer_tid != RT_NULL)
        rt_thread_startup(consumer_tid);
    else
        tc_stat(TC_STAT_END | TC_STAT_FAILED);

    return 0;
}

#ifdef RT_USING_TC
static void _tc_cleanup()
{
    /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
    rt_enter_critical();

    /* 删除线程 */
    if (producer_tid != RT_NULL && producer_tid->stat != RT_THREAD_CLOSE)
        rt_thread_delete(producer_tid);
    if (consumer_tid != RT_NULL && consumer_tid->stat != RT_THREAD_CLOSE)
        rt_thread_delete(consumer_tid);

    /* 调度器解锁 */

```

```

    rt_exit_critical();

    /* 设置TestCase状态 */
    tc_done(TC_STAT_PASSED);
}

int _tc_semaphore_producer_consumer()
{
    /* 设置TestCase清理回调函数 */
    tc_cleanup(_tc_cleanup);
    semaphore_producer_consumer_init();

    /* 返回TestCase运行的最长时间 */
    return 100;
}

/* 输出函数命令到finsh shell中 */
FINSH_FUNCTION_EXPORT(_tc_semaphore_producer_consumer, producer and consumer example);
#else
/* 用户应用入口 */
int rt_application_init()
{
    semaphore_producer_consumer_init();

    return 0;
}
#endif

```

在这个例子中，semaphore是作为一种锁的形式存在，当要访问临界资源：ring buffer时，通过持有semaphore的形式阻止其他线程进入（如果其他线程也打算进入，将在这里被挂起）。

4.3.3 使用场合

信号量是一种非常灵活的同步方式，可以运用在多种场合中。形成锁，同步，资源计数等关系，也能方便的用于线程与线程，中断与线程的同步中。

线程同步

线程同步是信号量最简单的一类应用。例如，两个线程用来进行任务间的执行控制转移，信号量的值初始化成具备0个信号量资源实例，而等待线程先直接在这个信号量上进行等待。

当信号线程完成它处理的工作时，释放这个信号量，以把等待在这个信号量上的线程唤醒，让它执行下一部分工作。这类场合也可以看成把信号量用于工作完成标志：信号线程完成它自己的工作，然后通知等待线程继续下一部分工作。

锁

锁，单一的锁常应用于多个线程间对同一临界区的访问。信号量在作为锁来使用时，通常应将信号量资源实例初始化成1，代表系统默认有一个资源可用。当线程需要访问临界资源时，它需要先获得这个资源锁。当这个线程成功获得资源锁时，其他打算访问临界区的线程将被挂起在该信号量上，这是因为其他线程在试图获取这个锁时，这个锁已经被锁上（信号量值是0）。当获得信号量的线程处理完毕，退出临界区时，它将会释放信号量并把锁解开，而挂起在锁上的第一个等待线程将被唤醒从而获得临界区的访问权。

因为信号量的值始终在1和0之间变动，所以这类锁也叫做二值信号量，如图 锁 所示：

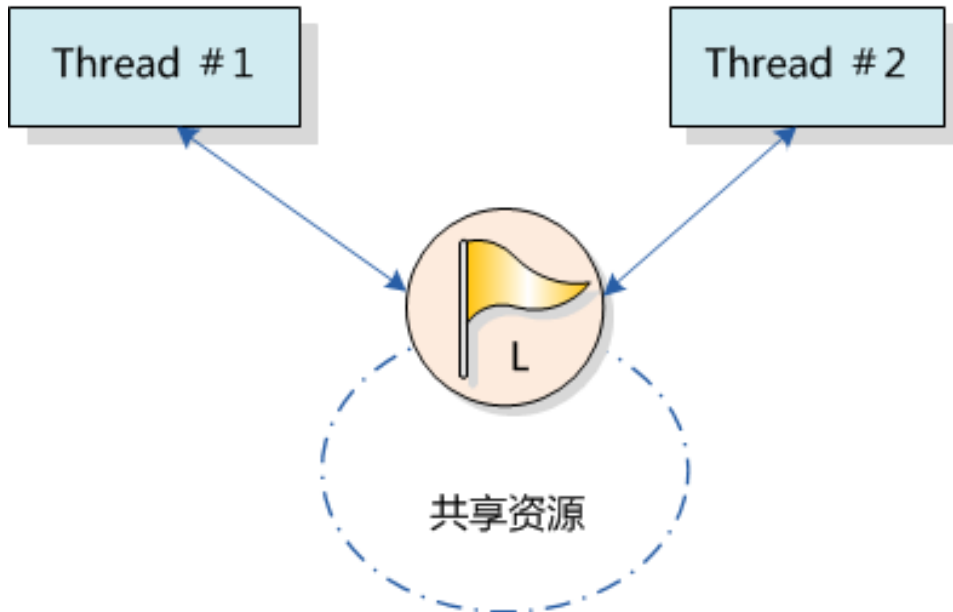


图 4.3: 锁

中断与线程的同步

信号量也能够方便的应用于中断与线程间的同步，例如一个中断触发，中断服务例程需要通知线程进行相应的数据处理。这个时候可以设置信号量的初始值是0，线程在试图持有这个信号量时，由于信号量的初始值是0，线程直接在这个信号量上挂起直到信号量被释放。当中断触发时，先进行与硬件相关的动作，例如从硬件的I/O口中读取相应的数据，并确认中断以清除中断源，而后释放一个信号量来唤醒相应的线程以做后续的数据处理。例如finsh shell线程的处理方式，如图 finsh shell的中断、线程间同步 所示：

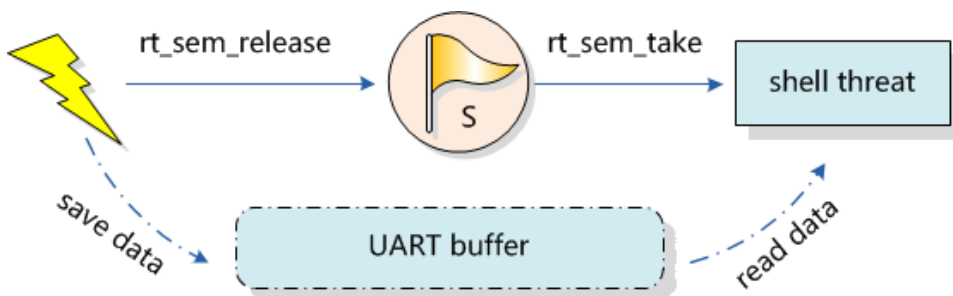


图 4.4: finsh shell的中断、线程间同步

semaphore先初始为0，而后shell线程试图取得信号量，因为信号量值是0，所以它会

被挂起。当console设备有数据输入时，将产生中断而进入中断服务例程。在中断服务例程中，它会读取console设备的数据，并把读得的数据放入uart buffer中进行缓冲，而后释放信号量，释放信号量的操作将唤醒shell线程。在中断服务例程运行完毕后，如果系统中没有比shell线程优先级更高的就绪线程存在时，shell线程将持有信号量并运行，从uart buffer缓冲区中获取输入的数据。

- 警告：中断与线程间的互斥不能采用信号量（锁）的方式，而应采用中断锁。

资源计数

资源计数适合于线程间速度不匹配的场所，这个时候信号量可以做为前一线程工作完成的计数，而当调度到后一线程时，它可以以一种连续的方式一次处理数个事件。例如，生产者与消费者问题中，生产者可以对信号进行多次释放，而后消费者被调度到时能够一次处理多个资源。

- 注：一般资源计数类型多是混合方式的线程间同步，因为对于单个的资源处理依然存在线程的多重访问，这就需要对一个单独的资源进行访问、处理，并进行锁方式的互斥操作。

4.4 互斥量

互斥量又叫相互排斥的信号量，是一种特殊的二值性信号量。它和信号量不同的是，它支持互斥量所有权、递归访问以及防止优先级翻转的特性。互斥量工作如互斥量的工作示意图所示。

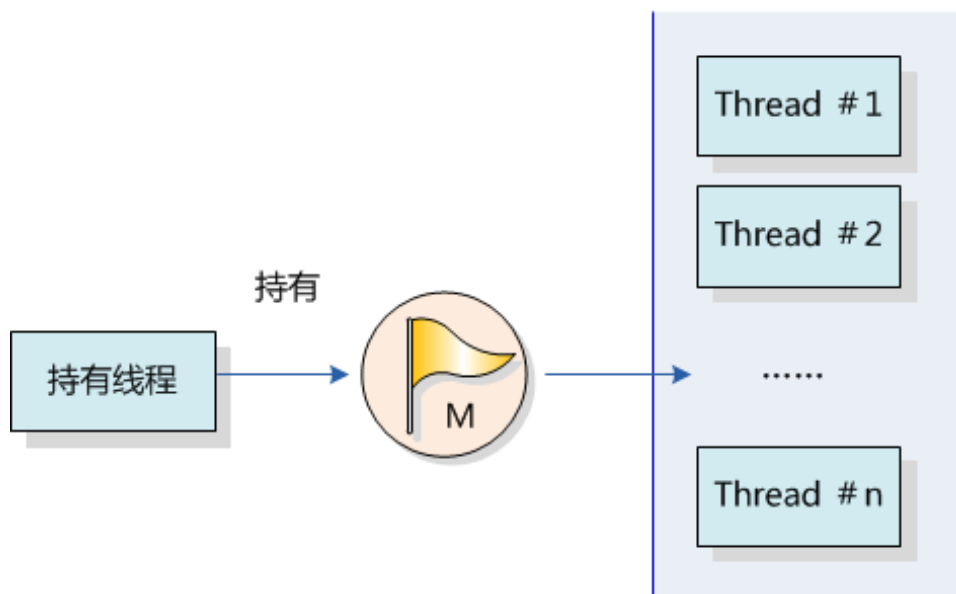


图 4.5: 互斥量的工作示意图

互斥量的状态只有两种，开锁或闭锁（两种状态值）。当有线程持有它时，互斥量处于闭锁状态，由这个线程获得它的所有权。相反，当这个线程释放它时，将对互斥量进行开锁，失去它的所有权。当一个线程持有互斥量时，其他线程将不能够对它进行开锁或持有它，持有该互斥量的线程也能够再次获得这个锁而不被挂起。这个特性与一般的二值信号量有很大的不同，在信号量中，因为已经不存在实例，线程递归持有会发生主动挂起（最终形成死锁）。

使用信号量会导致的另一个潜在问题是线程优先级翻转。所谓优先级翻转问题即当一个高优先级线程试图通过信号量机制访问共享资源时，如果该信号量已被一低优先级线程持有，而这个低优先级线程在运行过程中可能又被其它一些中等优先级的线程抢占，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，等待某一事件触发，线程C正在运行，此时线程C开始使用某一共享资源M。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M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M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操作系统中实现的是优先级继承算法。优先级继承是通过在线程A被阻塞的期间内，将线程C的优先级提升到线程A的优先级别，从而解决优先级翻转引起的问题。这样能够防止C（间接地防止A）被B抢占。优先级继承协议是指，提高某个占有某种资源的低优先级线程的优先级，使之与所有等待该资源的线程中优先级最高的那个线程的优先级相等，然后执行，而当这个低优先级线程释放该资源时，优先级重新回到初始设定。因此，继承优先级的线程避免了系统资源被任何中间优先级的线程抢占。

- 警告：在获得互斥量后，请尽快释放互斥量，并且在持有互斥量的过程中，不得再行更改持有互斥量线程的优先级。

4.4.1 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;           /* 继承自ipc_object类 */

    rt_uint16_t      value;                /* 互斥量的值 */
    rt_uint8_t       original_priority;    /* 持有线程的原始优先级 */
    rt_uint8_t       hold;                 /* 持有线程的持有次数 */
    struct rt_thread *owner;               /* 当前拥有互斥量的线程 */
};
/* rt_mutex_t为指向互斥量结构体的指针 */
typedef struct rt_mutex* rt_mutex_t;
```

rt_mutex对象从rt_ipc_object中派生，由IPC容器管理。

4.4.2 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。创建互斥量使用下面的函数接口：

```
rt_mutex_t rt_mutex_create (const char* name, rt_uint8_t flag);
```

可以调用rt_mutex_create函数创建一个互斥量，它的名字有name所指定。创建的互斥量由于指定的flag不同，而有不同的意义：使用PRIO优先级flag创建的IPC对象，在多个线程等待资源时，将由优先级高的线程优先获得资源。而使用FIFO先进先出flag创建的IPC对象，在多个线程等待资源时，将按照先来先得的顺序获得资源。

函数参数

参数	描述
name	互斥量的名称；
flag	互斥量标志，可以取如下类型的数值：

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

函数返回

创建成功返回指向互斥量的互斥量句柄；否则返回RT_NULL。

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。删除互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_delete (rt_mutex_t mutex);
```

当删除一个互斥量时，所有等待此互斥量的线程都将被唤醒，等待线程获得的返回值是-RT_ERROR。然后系统将该互斥量从内核对象管理器链表中删除并释放互斥量占用的内存空间。

函数参数

参数	描述
mutex	互斥量对象的句柄；

函数返回

RT_EOK

初始化互斥量

静态互斥量对象的内存是在系统编译时由编译器分配的，一般放于数据段或ZI段中。在使用这类静态互斥量对象前，需要先进行初始化。初始化互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_init (rt_mutex_t mutex, const char* name, rt_uint8_t flag);
```

使用该函数接口时，需指定互斥量对象的句柄（即指向互斥量控制块的指针），互斥量名称以及互斥量标志。互斥量标志可用上面创建互斥量函数里提到的标志。

函数参数

参数	描述
mutex	互斥量对象的句柄，它由用户提供，并指向互斥量对象的内存块；
name	互斥量名称；
flag	互斥量标志，可以取如下类型的数值：

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

函数返回

RT_EOK

脱离互斥量

脱离互斥量将把互斥量对象从内核对象管理器中删除。脱离互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_detach (rt_mutex_t mutex);
```

使用该函数接口后，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），然后系统将该互斥量从内核对象管理器链表中删除。

函数参数

参数	描述
mutex	互斥量对象的句柄；

函数返回

RT_EOK

获取互斥量

线程通过互斥量申请服务获取互斥量的所有权。线程对互斥量的所有权是独占的，某一个时刻一个互斥量只能被一个线程持有。获取互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_take (rt_mutex_t mutex, rt_int32_t 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该互斥量。如果互斥量已经被当前线程控制，则该互斥量的持有计数加1，当前线程也不会挂起等待。如果互斥量已经被其他线程占有，则当前线程在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。

函数参数

参数	描述
mutex	互斥量对象的句柄；
time	指定等待的时间。

函数返回

成功获得互斥量返回RT_EOK；超时返回-RT_ETIMEOUT；其他返回-RT_ERROR。

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。释放互斥量使用下面的函数接口：

```
rt_err_t rt_mutex_release(rt_mutex_t mutex);
```

使用该函数接口时，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，它的持有计数就减1。当该互斥量的持有计数为零时（即持有线程已经释放所有的持有操作），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，线程恢复为持有互斥量前的优先级。

函数参数

参数	描述
mutex	互斥量对象的句柄；

函数返回

RT_EOK

使用互斥量的例程如下例所示：

```
/*
 * 程序清单：互斥量使用例程
 *
 * 这个例子将创建3个动态线程以检查持有互斥量时，持有的线程优先级是否
 * 被调整到等待线程优先级中的最高优先级。
 */
```



```

* 线程1, 2, 3的优先级从高到低分别被创建,
* 线程3先持有互斥量, 而后线程2试图持有互斥量, 此时线程3的优先级应该
* 被提升为和线程2的优先级相同。线程1用于检查线程3的优先级是否被提升
* 为与线程2的优先级相同。
*/
#include <rtthread.h>
#include "tc_comm.h"

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t tid3 = RT_NULL;
static rt_mutex_t mutex = RT_NULL;

/* 线程1入口 */
static void thread1_entry(void* parameter)
{
    /* 先让低优先级线程运行 */
    rt_thread_delay(10);

    /* 此时thread3持有mutex, 并且thread2等待持有mutex */

    /* 检查thread2与thread3的优先级情况 */
    if (tid2->current_priority != tid3->current_priority)
    {
        /* 优先级不相同, 测试失败 */
        tc_stat(TC_STAT_END | TC_STAT_FAILED);
        return;
    }
}

/* 线程2入口 */
static void thread2_entry(void* parameter)
{
    rt_err_t result;

    /* 先让低优先级线程运行 */
    rt_thread_delay(5);

    while (1)
    {
        /*
         * 试图持有互斥锁, 此时thread3持有, 应把thread3的优先级提升
         * 到thread2相同的优先级
         */
    }
}

```

```

        result = rt_mutex_take(mutex, RT_WAITING_FOREVER);

        if (result == RT_EOK)
        {
            /* 释放互斥锁 */
            rt_mutex_release(mutex);
        }
    }
}

/* 线程3入口 */
static void thread3_entry(void* parameter)
{
    rt_tick_t tick;
    rt_err_t result;

    while (1)
    {
        result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
        result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
        if (result != RT_EOK)
        {
            tc_stat(TC_STAT_END | TC_STAT_FAILED);
        }

        /* 做一个长时间的循环, 总共50个OS Tick */
        tick = rt_tick_get();
        while (rt_tick_get() - tick < 50) ;

        rt_mutex_release(mutex);
        rt_mutex_release(mutex);
    }
}

int mutex_simple_init()
{
    /* 创建互斥锁 */
    mutex = rt_mutex_create("mutex", RT_IPC_FLAG_FIFO);
    if (mutex == RT_NULL)
    {
        tc_stat(TC_STAT_END | TC_STAT_FAILED);
        return 0;
    }

    /* 创建线程1 */

```

```

    tid1 = rt_thread_create("t1",
        thread1_entry, /* 线程入口是thread1_entry */
        RT_NULL, /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);
    else
        tc_stat(TC_STAT_END | TC_STAT_FAILED);

    /* 创建线程2 */
    tid2 = rt_thread_create("t2",
        thread2_entry, /* 线程入口是thread2_entry */
        RT_NULL, /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
    else
        tc_stat(TC_STAT_END | TC_STAT_FAILED);

    /* 创建线程3 */
    tid3 = rt_thread_create("t3",
        thread3_entry, /* 线程入口是thread3_entry */
        RT_NULL, /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY + 1, THREAD_TIMESLICE);
    if (tid3 != RT_NULL)
        rt_thread_startup(tid3);
    else
        tc_stat(TC_STAT_END | TC_STAT_FAILED);

    return 0;
}

#ifdef RT_USING_TC
static void _tc_cleanup()
{
    /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
    rt_enter_critical();

    /* 删除线程 */
    if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid1);
    if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid2);
    if (tid3 != RT_NULL && tid3->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid3);
}

```

```

    if (mutex != RT_NULL)
    {
        rt_mutex_delete(mutex);
    }

    /* 调度器解锁 */
    rt_exit_critical();

    /* 设置TestCase状态 */
    tc_done(TC_STAT_PASSED);
}

int _tc_mutex_simple()
{
    /* 设置TestCase清理回调函数 */
    tc_cleanup(_tc_cleanup);
    mutex_simple_init();

    /* 返回TestCase运行的最长时间 */
    return 100;
}

/* 输出函数命令到finsh shell中 */
FINSH_FUNCTION_EXPORT(_tc_mutex_simple, sime mutex example);
#else
/* 用户应用入口 */
int rt_application_init()
{
    mutex_simple_init();

    return 0;
}
#endif

```

4.4.3 使用场合

互斥量的使用比较单一，因为它是信号量的一种，并且它是以锁的形式存在。在初始化的时候，互斥量永远都处于开锁的状态，而被线程持有的时候则立刻转为闭锁的状态。互斥量更适合于：

- 线程多次持有互斥量的情况下。这样可以避免同一线程多次递归持有而造成死锁的问题；
- 可能会由于多线程同步而造成优先级翻转的情况；

另外需要切记的是互斥量不能在中断服务例程中使用。

4.5 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。即一个线程可等待多个事件的触发：可以是其中任意一个事件唤醒线程进行事件处理的操作；也可以是几个事件都到达后才唤醒线程进行后续的处理；同样，事件也可以是多个线程同步多个事件，这种多个事件的集合可以用一个32位无符号整型变量来表示，变量的每一位代表一个事件，线程通过“逻辑与”或“逻辑或”与一个或多个事件建立关联，形成一个事件集。事件的“逻辑或”也称为是独立型同步，指的是线程与任何事件之一发生同步；事件“逻辑与”也称为是关联型同步，指的是线程与若干事件都发生同步。

RT-Thread定义的事件有以下特点：

- 事件只与线程相关，事件间相互独立：每个线程拥有32个事件标志，采用一个32 bit 无符号整型数进行记录，每一个bit代表一个事件。若干个事件构成一个事件集；
- 事件仅用于同步，不提供数据传输功能；
- 事件无排队性，即多次向线程发送同一事件(如果线程还未来得及读走)，其效果等同于只发送一次。

在RT-Thread实现中，每个线程都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，可以通过32个事件标志和这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。

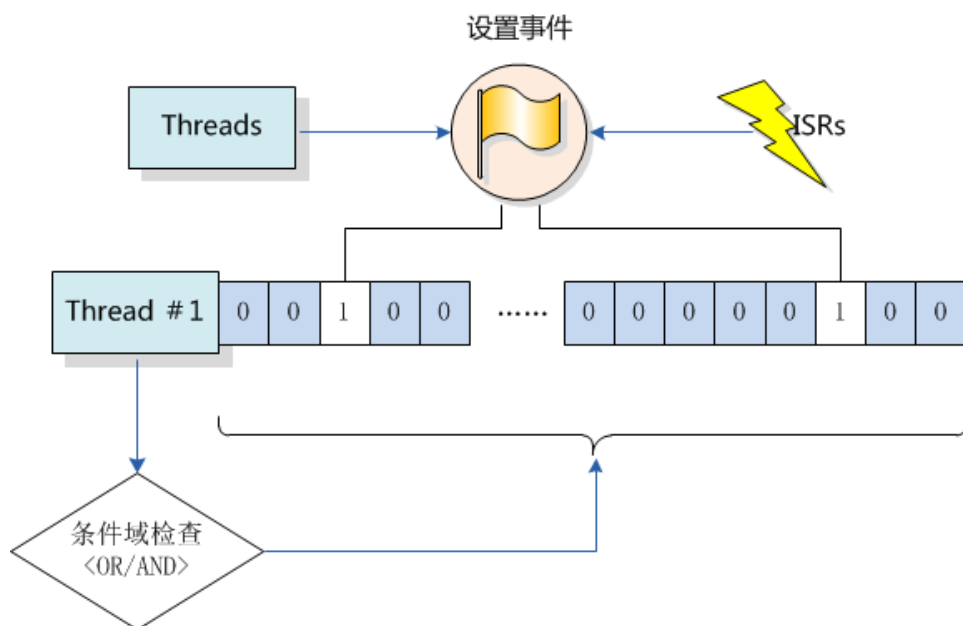


图 4.6: 事件工作示意图

如图 事件工作示意图 所示，线程#1的事件标志中第2位和第29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1和事件29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1或事件29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，则当线程#1唤醒后将主动把事件1和事件29清为零，否则事件标志将依然存在（即置1）。

4.5.1 事件控制块

```
struct rt_event
{
    struct rt_ipc_object parent; /* 继承自ipc_object类 */
    rt_uint32_t set;           /* 事件集合 */
};
/* rt_event_t是指向事件结构体的指针 */
typedef struct rt_event* rt_event_t;
```

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

4.5.2 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，创建事件使用下面的函数接口：

```
rt_event_t rt_event_create (const char* name, rt_uint8_t flag);
```

调用该函数接口时，系统会从动态内存堆中分配事件对象，然后进行对象的初始化，IPC对象初始化，并把set设置成0。

函数参数

参数	描述
name	事件的名称；
flag	事件的标志，可以使用如下的数值：

```
#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*/
```

函数返回

创建成功返回事件对象的句柄；创建失败返回RT_NULL。

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来释放系统资源。删除事件可以使用下面的函数接口：

```
rt_err_t rt_event_delete (rt_event_t event);
```

在调用rt_event_delete函数删除一个事件对象时，应该确保该事件不再被使用。在删除前会唤醒所有挂起在该事件上的线程（线程的返回值是-RT_ERROR），然后释放事件对象占用的内存块。

函数参数

参数	描述
event	事件对象的句柄。

函数返回

RT_EOK

初始化事件

静态事件对象的内存是在系统编译时由编译器分配的，一般放于数据段或ZI段中。在使用静态事件对象前，需要先行对它进行初始化操作。初始化事件使用下面的函数接口：

```
rt_err_t rt_event_init(rt_event_t event, const char* name, rt_uint8_t flag);
```

调用该接口时，需指定静态事件对象的句柄（即指向事件控制块的指针），然后系统会初始化事件对象，并加入到系统对象容器中进行管理。

函数参数

参数	描述
event	事件对象的句柄。
name	事件名称；
flag	事件的标志，可以使用如下的数值：

```
#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*/
```

函数返回

RT_EOK

脱离事件

脱离事件是将事件对象从内核对象管理器中删除。脱离事件使用下面的函数接口：

```
rt_err_t rt_event_detach(rt_event_t event);
```

用户调用这个函数时，系统首先唤醒所有挂在该事件等待队列上的线程（线程的返回值是- RT_ERROR ），然后将该事件从内核对象管理器中删除。

函数参数

参数	描述
event	事件对象的句柄。

函数返回
RT_EOK

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才激活线程，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。接收事件使用下面的函数接口：

```
rt_err_t rt_event_rcv(rt_event_t event, rt_uint32_t set, rt_uint8_t option,
                      rt_int32_t timeout, rt_uint32_t* recved);
```

当用户调用这个接口时，系统首先根据set参数和接收选项来判断它要接收的事件是否发生，如果已经发生，则根据参数option上是否设置有RT_EVENT_FLAG_CLEAR来决定是否重置事件的相应标志位，然后返回（其中recved参数返回收到的事件）；如果没有发生，则把等待的set和option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的超时时间。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就不等待，而直接返回-RT_TIMEOUT。

函数参数

参数	描述
event	事件对象的句柄。
set	接收线程感兴趣的事件；
option	接收选项；
timeout	指定超时时间；
recved	指向收到的事件；

函数返回
正确接收返回RT_EOK，超时返回-RT_TIMEOUT，其他返回-RT_ERROR。

发送事件

通过发送事件服务，可以发送一个或多个事件。发送事件可以使用下面的函数接口：

```
rt_err_t rt_event_send(rt_event_t event, rt_uint32_t set);
```

使用该函数接口时，通过参数set指定的事件标志来设定event对象的事件标志值，然后遍历等待在event事件对象上的等待线程链表，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，则唤醒该线程。

函数参数

参数	描述
event	事件对象的句柄。
set	发送的事件集；

函数返回
RT_EOK
使用事件的例程如下例所示：

```
/*
 * 程序清单：事件例程
 *
 * 这个程序会创建3个动态线程及初始化一个静态事件对象
 * 一个线程等待在事件对象上以接收事件；
 * 一个线程定时发送事件（事件3）
 * 一个线程定时发送事件（事件5）
 */
#include <rtthread.h>
#include "tc_comm.h"

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t tid3 = RT_NULL;

/* 事件控制块 */
static struct rt_event event;

/* 线程1入口函数 */
static void thread1_entry(void *param)
{
    rt_uint32_t e;

    while (1)
    {
        /* 以逻辑与的方式接收事件 */
        if (rt_event_recv(&event, ((1 << 3) | (1 << 5)),
            RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
            RT_WAITING_FOREVER, &e) == RT_EOK)
        {
            rt_kprintf("thread1: AND recv event 0x%x\n", e);
        }

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

```

    rt_thread_delay(10);

    /* 以逻辑或的方式接收事件 */
    if (rt_event_recv(&event, ((1 << 3) | (1 << 5)),
        RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
        RT_WAITING_FOREVER, &e) == RT_EOK)
    {
        rt_kprintf("thread1: OR recv event 0x%x\n", e);
    }

    rt_thread_delay(5);
}

/* 线程2入口函数 */
static void thread2_entry(void *param)
{
    /* 线程2持续地发送事件#3 */
    while (1)
    {
        rt_kprintf("thread2: send event1\n");
        rt_event_send(&event, (1 << 3));

        rt_thread_delay(10);
    }
}

/* 线程3入口函数 */
static void thread3_entry(void *param)
{
    /* 线程3持续地发送事件#5 */
    while (1)
    {
        rt_kprintf("thread3: send event2\n");
        rt_event_send(&event, (1 << 5));

        rt_thread_delay(20);
    }
}

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

```

```

/* 创建线程1 */
tid1 = rt_thread_create("t1",
    thread1_entry, /* 线程入口是thread1_entry */
    RT_NULL, /* 入口参数是RT_NULL */
    THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
if (tid1 != RT_NULL)
    rt_thread_startup(tid1);
else
    tc_stat(TC_STAT_END | TC_STAT_FAILED);

/* 创建线程2 */
tid2 = rt_thread_create("t2",
    thread2_entry, /* 线程入口是thread2_entry */
    RT_NULL, /* 入口参数是RT_NULL */
    THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
if (tid2 != RT_NULL)
    rt_thread_startup(tid2);
else
    tc_stat(TC_STAT_END | TC_STAT_FAILED);

/* 创建线程3 */
tid3 = rt_thread_create("t3",
    thread3_entry, /* 线程入口是thread3_entry */
    RT_NULL, /* 入口参数是RT_NULL */
    THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
if (tid3 != RT_NULL)
    rt_thread_startup(tid3);
else
    tc_stat(TC_STAT_END | TC_STAT_FAILED);

return 0;
}

#ifdef RT_USING_TC
static void _tc_cleanup()
{
    /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
    rt_enter_critical();

    /* 删除线程 */
    if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid1);
    if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid2);
    if (tid3 != RT_NULL && tid3->stat != RT_THREAD_CLOSE)

```

```

        rt_thread_delete(tid3);

    /* 执行事件对象脱离 */
    rt_event_detach(&event);

    /* 调度器解锁 */
    rt_exit_critical();

    /* 设置TestCase状态 */
    tc_done(TC_STAT_PASSED);
}

int _tc_event_simple()
{
    /* 设置TestCase清理回调函数 */
    tc_cleanup(_tc_cleanup);
    event_simple_init();

    /* 返回TestCase运行的最长时间 */
    return 100;
}

/* 输出函数命令到finsh shell中 */
FINSH_FUNCTION_EXPORT(_tc_event_simple, a simple event example);
#else
/* 用户应用入口 */
int rt_application_init()
{
    event_simple_init();

    return 0;
}
#endif

```

4.5.3 使用场合

事件可使用于多种场合，它能够在一定程度上替代信号量，用于线程间同步。一个线程或中断服务例程发送一个事件给事件对象，而后等待的线程被唤醒并对相应的事件进行处理。但是它与信号量不同的是，事件的发送操作在事件未清除前，是不可累计的，而信号量的释放动作是累计的。事件另外一个特性是，接收线程可等待多种事件，即多个事件对应一个线程或多个线程。同时按照线程等待的参数，可选择是“逻辑或”触发还是“逻辑与”触发。这个特性也是信号量等所不具备的，信号量只能识别单一的释放动作，而不能同时等待多种类型的释放。如图 多事件接收 所示：

各个事件类型可分别发送或一起发送给事件对象，而事件对象可以等待多个线程，它们仅对它们感兴趣的事件进行关注。当有它们感兴趣的事件发生时，线程就将被唤醒并进行后续的处理动作。

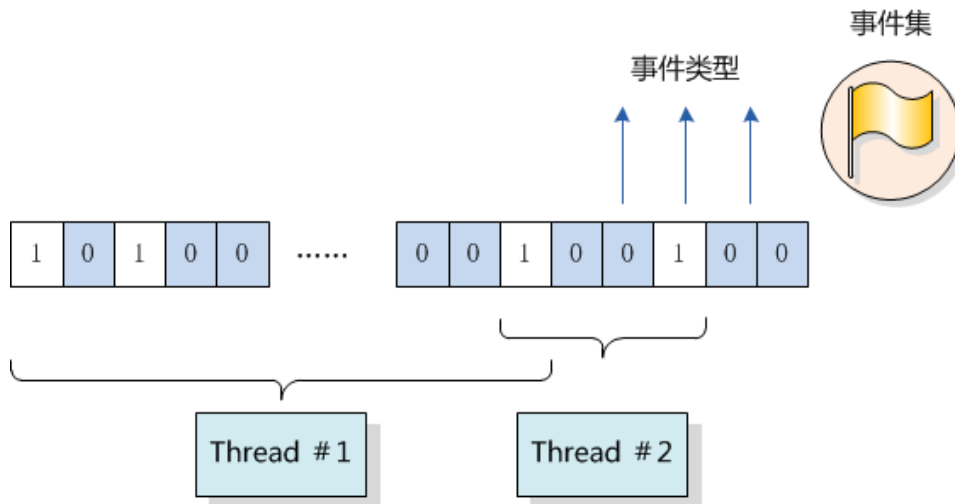


图 4.7: 多事件接收

4.6 邮箱

邮箱服务是实时操作系统中一种典型的任务间通信方法，特点是开销比较低，效率较高。邮箱中的每一封邮件只能容纳固定的4字节内容（针对32位处理系统，指针的大小即为4个字节，所以一封邮件恰好能够容纳一个指针）。典型的邮箱也称作交换消息，如图6-8所示，线程或中断服务例程把一封4字节长度的邮件发送到邮箱中。而一个或多个线程可以从邮箱中接收这些邮件进行处理。

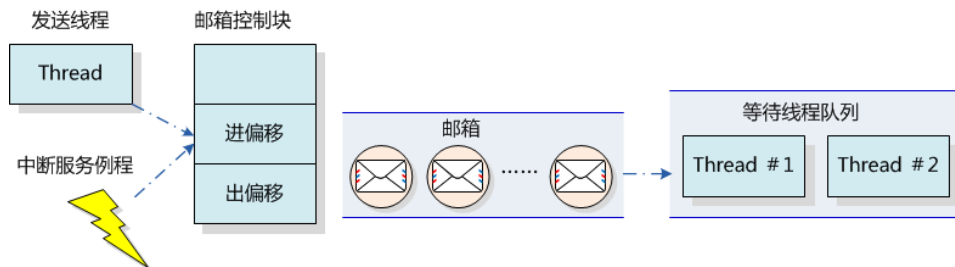


图 4.8: 邮箱工作示意图

RT-Thread操作系统采用的邮箱通信机制有点类似于传统意义上的管道，用于线程间通讯。非阻塞方式的邮件发送过程能够安全的应用于中断服务中，是线程，中断服务，定时器向线程发送消息的有效手段。通常来说，邮件收取过程可能是阻塞的，这取决于邮箱中是否有邮件，以及收取邮件时设置的超时时间。当邮箱中不存在邮件且超时时间不为0时，邮件收取过程将变成阻塞方式。所以在这类情况下，只能由线程进行邮件的收取。

RT-Thread操作系统的邮箱中可存放固定条数的邮件，邮箱容量在创建/初始化邮箱时设定，每个邮件大小为4字节。当需要在线程间传递比较大的消息时，可以把指向一个缓冲区的指针作为邮件发送到邮箱中。

在一个线程向邮箱发送邮件时，如果邮箱没满，将把邮件复制到邮箱中。如果邮箱已经满了，发送线程可以设置超时时间，选择是否等待挂起或直接返回-RT_EFULL。如果发送线程选择挂起等待，那么当邮箱中的邮件被收取而空出空间来时，等待挂起的发送线程将被唤醒继续发送的过程。

在一个线程从邮箱中接收邮件时，如果邮箱是空的，接收线程可以选择是否等待挂起直到收到新的邮件而唤醒，或设置超时时间。当达到设置的超时时间，邮箱依然未收到邮件

时，这个选择超时等待的线程将被唤醒并返回-RT_ETIMEOUT。如果邮箱中存在邮件，那么接收线程将复制邮箱中的4个字节邮件到接收线程中。

4.6.1 邮箱控制块

```
struct rt_mailbox
{
    struct rt_ipc_object parent;

    rt_uint32_t* msg_pool;           /* 邮箱缓冲区的开始地址 */
    rt_uint16_t size;                /* 邮箱缓冲区的大小 */

    rt_uint16_t entry;               /* 邮箱中邮件的数目 */
    rt_uint16_t in_offset, out_offset; /* 邮箱缓冲的进出指针 */
    rt_list_t suspend_sender_thread; /* 发送线程的挂起等待队列 */
};
typedef struct rt_mailbox* rt_mailbox_t;
```

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

4.6.2 邮箱相关接口

创建邮箱

创建邮箱对象可以调用如下的函数接口：

```
rt_mailbox_t rt_mb_create (const char* name, rt_size_t size, rt_uint8_t flag);
```

创建邮箱对象时会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，这块内存的大小等于邮件大小（4字节）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。

函数参数

参数	描述
name	邮箱名称；
size	邮箱容量；
flag	邮箱标志，它可以取如下数值：

```
#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*/
```

函数返回

创建成功返回邮箱对象的句柄；否则返回-RT_ERROR。

删除邮箱

当邮箱不再被使用时，应该删除它来释放相应的系统资源，一旦操作完成，邮箱将被永久性的删除。删除邮箱的函数接口如下：

```
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，内核先唤醒挂起在该邮箱上的所有线程（线程获得返回值是-RT_ERROR），然后再释放邮箱使用的内存，最后删除邮箱对象。

函数参数

参数	描述
mb	邮箱对象的句柄。

函数返回

RT_EOK

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邮箱对象的初始化。其他与创建邮箱不同的是，此处静态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。函数接口如下：

```
rt_err_t rt_mb_init(rt_mailbox_t mb, const char* name, void* msgpool,  
rt_size_t size, rt_uint8_t flag)
```

初始化邮箱时，该函数接口需要获得用户已经申请获得的邮箱对象控制块，缓冲区的指针，以及邮箱名称和邮箱容量。

函数参数

参数	描述
mb	邮箱对象的句柄；
name	邮箱名称；
msgpool	缓冲区指针；
size	邮箱容量；
flag	邮箱标志，它可以取如下数值：

```
#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*/
```

函数返回
RT_EOK

- 注：这里的size参数指定的是邮箱的容量，即如果msgpool的字节数是N，那么邮箱容量应该是N/4。

脱离邮箱

脱离邮箱将把邮箱对象从内核对象管理器中删除。脱离邮箱使用下面的接口：

```
rt_err_t rt_mb_detach(rt_mailbox_t mb);
```

使用该函数接口后，内核先唤醒所有挂在该邮箱上的线程（线程获得返回值是-RT_ERROR ），然后将该邮箱对象从内核对象管理器中删除。

函数参数

参数	描述
mb	邮箱对象的句柄。

函数返回
RT_EOK

发送邮件

线程或者中断服务程序可以通过邮箱给其他线程发送邮件，发送邮件函数接口如下：

```
rt_err_t rt_mb_send (rt_mailbox_t mb, rt_uint32_t value);
```

发送的邮件可以是32位任意格式的数据，一个整型值或者一个指向缓冲区的指针。当邮箱中的邮件已经满时，发送邮件的线程或者中断程序会收到-RT_EFULL 的返回值。

函数参数

参数	描述
mb	邮箱对象的句柄；
value	邮件内容。

函数返回
发送成功返回RT_EOK；如果邮箱已经满了，返回-RT_EFULL。

等待方式发送邮件

用户也可以通过如下的函数接口向指定邮箱发送邮件：


```
rt_err_t rt_mb_send_wait (rt_mailbox_t mb, rt_uint32_t value, rt_int32_t timeout);
```

rt_mb_send_wait与rt_mb_send的区别在于，如果邮箱已经满了，那么发送线程将根据设定的timeout参数等待邮箱中因为收取邮件而空出空间。如果设置的超时时间到达依然没有空出空间，这时发送线程将被唤醒返回错误码。

函数参数

参数	描述
mb	邮箱对象的句柄；
value	邮件内容；
timeout	超时时间。

函数返回

发送成功返回RT_EOK；如果设置的时间超时依然未发送成功，返回-RT_ETIMEOUT，其他情况返回-RT_ERROR。

接收邮件

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并返回RT_EOK的返回值，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。接收邮件函数接口如下：

```
rt_err_t rt_mb_recv (rt_mailbox_t mb, rt_uint32_t* value, rt_int32_t timeout);
```

接收邮件时，接收者需指定接收邮件的邮箱句柄，并指定接收到的邮件存放位置以及最多能够等待的超时时间。如果接收时设定了超时，当指定的时间内依然未收到邮件时，将返回-RT_ETIMEOUT。

函数参数

参数	描述
mb	邮箱对象的句柄；
value	邮件内容；
timeout	超时时间。

函数返回

成功收到返回RT_EOK，超时返回-RT_ETIMEOUT，其他返回-RT_ERROR。
使用邮箱的例程如下例所示：

/*

* 程序清单：邮箱例程

```

*
* 这个程序会创建2个动态线程，一个静态的邮箱对象，其中一个线程往邮箱中发送邮件，
* 一个线程从邮箱中收取邮件。
*/
#include <rtthread.h>
#include "tc_comm.h"

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

/* 邮箱控制块 */
static struct rt_mailbox mb;
/* 用于放邮件的内存池 */
static char mb_pool[128];

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

/* 线程1入口 */
static void thread1_entry(void* parameter)
{
    unsigned char* str;

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

        /* 从邮箱中收取邮件 */
        if (rt_mb_recv(&mb, (rt_uint32_t*)&str, RT_WAITING_FOREVER)
            == RT_EOK)
        {
            /* 显示邮箱内容 */
            rt_kprintf("thread1: get a mail, the content:%s\n", str);

            /* 延时10个OS Tick */
            rt_thread_delay(10);
        }
    }
}

/* 线程2入口 */
static void thread2_entry(void* parameter)
{
    rt_uint8_t count;

```

```

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

    /* 延时20个OS Tick */
    rt_thread_delay(20);
}
}

int mbox_simple_init()
{
    /* 初始化一个mailbox */
    rt_mb_init(&mb,
        "mbt",          /* 名称是mbt */
        &mb_pool[0],    /* 邮箱用到的内存池是mb_pool */
        sizeof(mb_pool)/4, /* 大小是mb_pool/4, 因为每封邮件的大小是4字节 */
        RT_IPC_FLAG_FIFO); /* 采用FIFO方式进行线程等待 */

    /* 创建线程1 */
    tid1 = rt_thread_create("t1",
        thread1_entry,    /* 线程入口是thread1_entry */
        RT_NULL,          /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);
    else
        tc_stat(TC_STAT_END | TC_STAT_FAILED);

    /* 创建线程2 */
    tid2 = rt_thread_create("t2",
        thread2_entry,    /* 线程入口是thread2_entry */
        RT_NULL,          /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);

```

```

    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
    else
        tc_stat(TC_STAT_END | TC_STAT_FAILED);

    return 0;
}

#ifdef RT_USING_TC
static void _tc_cleanup()
{
    /* 调度器上锁, 上锁后, 将不再切换到其他线程, 仅响应中断 */
    rt_enter_critical();

    /* 删除线程 */
    if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid1);
    if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid2);

    /* 执行邮箱对象脱离 */
    rt_mb_detach(&mb);

    /* 调度器解锁 */
    rt_exit_critical();

    /* 设置TestCase状态 */
    tc_done(TC_STAT_PASSED);
}

int _tc_mbox_simple()
{
    /* 设置TestCase清理回调函数 */
    tc_cleanup(_tc_cleanup);
    mbox_simple_init();

    /* 返回TestCase运行的最长时间 */
    return 100;
}

/* 输出函数命令到finsh shell中 */
FINSH_FUNCTION_EXPORT(_tc_mbox_simple, a simple mailbox example);
#else
/* 用户应用入口 */
int rt_application_init()
{

```

```

    mbox_simple_init();

    return 0;
}
#endif

```

4.6.3 使用场合

邮箱是一种简单的线程间消息传递方式，在RT-Thread操作系统的实现中能够一次传递4字节邮件，并且邮箱具备一定的存储功能，能够缓存一定数量的邮件数(邮件数由创建、初始化邮箱时指定的容量决定)。邮箱中一封邮件的最大长度是4字节，所以邮箱能够用于不超过4字节的消息传递，当传送的消息长度大于这个数目时就不能再采用邮箱的方式。最重要的是，在32位系统上4字节的内容恰好适合放置一个指针，所以邮箱也适合那种仅传递指针的情况，例如：

```

struct msg
{
    rt_uint8_t *data_ptr;
    rt_uint32_t data_size;
};

```

对于这样一个消息结构体，其中包含了指向数据的指针data_ptr和数据块长度的变量data_size。当一个线程需要把这个消息发送给另外一个线程时，可以采用如下的操作：

```

struct msg* msg_ptr;

msg_ptr = (struct msg*)rt_malloc(sizeof(struct msg));
msg_ptr->data_ptr = ...; /* 指向相应的数据块地址*/
msg_ptr->data_size = len; /* 数据块的长度*/
/* 发送这个消息指针给mb邮箱*/
rt_mb_send(mb, (rt_uint32_t)msg_ptr);

```

而在接收线程中，因为收取过来的是指针，而msg_ptr是一个新分配出来的内存块，所以在接收线程处理完毕后，需要释放相应的内存块：

```

struct msg* msg_ptr;
if (rt_mb_recv(mb, (rt_uint32_t*)&msg_ptr) == RT_EOK)
{
    /* 在接收线程处理完毕后，需要释放相应的内存块*/
    rt_free(msg_ptr);
}

```

4.7 消息队列

消息队列是另一种常用的线程间通讯方式，它能够接收来自线程或中断服务例程中不固定长度的消息，并把消息缓存在自己的内存空间中。其他线程也能够从消息队列中读取相应的消息，而当消息队列是空的时候，可以挂起读取线程。当有新的消息到达时，挂起的线程将被唤醒以接收并处理消息。消息队列是一种异步的通信方式。

如图 消息队列的工作示意图 所示，通过消息队列服务，线程或中断服务例程可以将一条或多条消息放入消息队列中。同样，一个或多个线程可以从消息队列中获得消息。当有多个消息发送到消息队列时，通常应将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)。

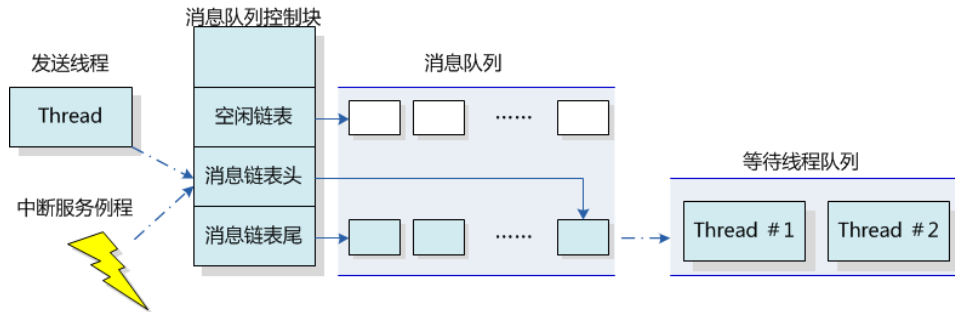


图 4.9: 消息队列的工作示意图

RT-Thread操作系统的消息队列对象由多个元素组成，当消息队列被创建时，它就被分配了消息队列控制块：消息队列名称、内存缓冲区、消息大小以及队列长度等。同时每个消息队列对象中包含着多个消息框，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消息链表头和消息链表尾，对应于消息队列控制块中的msg_queue_head和msg_queue_tail；有些消息框可能是空的，它们通过msg_queue_free形成一个空闲消息框链表。所有消息队列中的消息框总数即是消息队列的长度，这个长度可在消息队列创建时指定。

4.7.1 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_uint16_t msg_size; /* 每个消息的长度*/
    rt_uint16_t max_msgs; /* 最大能够容纳的消息数*/

    rt_uint16_t entry; /* 队列中已有的消息数*/

    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*/
};

typedef struct rt_messagequeue* rt_mq_t;
```

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

4.7.2 消息队列相关接口

创建消息队列

消息队列在使用前，应该被创建出来，或对已有的静态消息队列对象进行初始化，创建消息队列的函数接口如下所示：

```
rt_mq_t rt_mq_create(const char* name, rt_size_t msg_size, rt_size_t max_msgs, rt_uint8_t flag);
```

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，这块内存的大小等于[消息大小+消息头（用于链表连接）]与消息队列容量的乘积，接着再初始化消息队列，此时消息队列为空。

函数参数

参数	描述
name	消息队列的名称；
msg_size	消息队列中一条消息的最大长度；
max_msgs	消息队列的最大容量；
flag	消息队列采用的等待方式，可以取值：

```
#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*/
```

函数返回

成功创建返回消息队列对象的句柄；否则返回-RT_ERROR。

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。删除消息队列的函数接口如下：

```
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，则内核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。

函数参数

参数	描述
mq	消息队列对象的句柄；

函数返回

RT_EOK

初始化消息队列

初始化静态消息队列对象跟创建消息队列对象类似，只是静态消息队列对象的内存是在系统编译时由编译器分配的，一般放于数据段或ZI段中。在使用这类静态消息队列对象前，需要进行初始化。初始化消息队列对象的函数接口如下：

```
rt_err_t rt_mq_init(rt_mq_t mq, const char* name, void *msgpool, rt_size_t msg_size, rt_size_t pool_size,
                    rt_uint8_t flag);
```

初始化消息队列时，该接口需要获得消息队列对象的句柄（即指向消息队列对象控制块的指针）、消息队列名、消息缓冲区指针、消息大小以及消息队列容量。如图 消息队列的工作示意图 所示，消息队列初始化后所有消息都挂在空闲消息链表上，消息队列为空。

函数参数

参数	描述
mq	指向静态消息队列对象的句柄；
name	消息队列的名称；
msgpool	用于存放消息的缓冲区；
msg_size	消息队列中一条消息的最大长度；
pool_size	存放消息的缓冲区大小；
flag	消息队列采用的等待方式，可以取值：

```
#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*/
```

函数返回

RT_EOK

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。脱离消息队列使用下面的接口：

```
rt_err_t rt_mq_detach(rt_mq_t mq);
```

使用该函数接口后，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。

函数参数

参数	描述
mq	指向静态消息队列对象的句柄；

函数返回

RT_EOK

4.7.3 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，消息队列对象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。当且仅当空闲消息链表上有可用的空闲消息块时，发送者才能成功发送消息；当空闲消息链表上无可用消息块，说明消息队列已满，此时，发送消息的线程或者中断程序会收到一个错误码（-RT_EFULL）。发送消息的函数接口如下：

```
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的消息队列的对象句柄（即指向消息队列控制块的指针），并且指定发送的消息内容以及消息大小。如图 消息队列的工作示意图 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。

函数参数

参数	描述
mq	消息队列对象的句柄；
buffer	消息内容；
size	消息大小。

函数返回

发送成功返回RT_EOK，如果消息队列已满返回-RT_EFULL。

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，接收者就能够优先接收到紧急消息，从而及时进行消息处理。发送紧急消息的函数接口如下：

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

参数：

函数参数

参数	描述
mq	消息队列对象的句柄；
buffer	消息内容；
size	消息大小。

函数返回

发送成功返回RT_EOK，如果消息队列已满返回-RT_EFULL。

接收消息

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。接收消息函数接口如下：

```
rt_err_t rt_mq_rcv (rt_mq_t mq, void* buffer, rt_size_t size, rt_int32_t timeout);
```

接收消息时，接收者需指定存储消息的消息队列对象句柄，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里。此外，还需指定未能及时取到消息时的超时时间。如图 消息队列的工作示意图 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。

函数参数

参数	描述
mq	消息队列对象的句柄；
buffer	用于接收消息的数据块；
size	消息大小；
timeout	指定的超时时间。

函数返回

成功收到返回RT_EOK，超时返回-RT_ETIMEOUT，其他返回-RT_ERROR。

使用消息队列的例程如下例所示：

```
/*
 * 程序清单：消息队列例程
 *
 * 这个程序会创建3个动态线程：
 * 一个线程会从消息队列中收取消息；
 * 一个线程会定时给消息队列发送消息；
 * 一个线程会定时给消息队列发送紧急消息。
 */
#include <rtthread.h>
```

```

#include "tc_comm.h"

/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t tid3 = RT_NULL;

/* 消息队列控制块 */
static struct rt_messagequeue mq;
/* 消息队列中用到的放置消息的内存池 */
static char msg_pool[2048];

/* 线程1入口函数 */
static void thread1_entry(void* parameter)
{
    char buf[128];

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

        /* 从消息队列中接收消息 */
        if (rt_mq_rcv(&mq, &buf[0], sizeof(buf), RT_WAITING_FOREVER)
            == RT_EOK)
        {
            /* 输出内容 */
            rt_kprintf("thread1: rcv a msg, the content:%s\n", buf);
        }

        /* 延迟10个OS Tick */
        rt_thread_delay(10);
    }
}

/* 线程2入口函数 */
static void thread2_entry(void* parameter)
{
    int i, result;
    char buf[] = "this is message No.x";

    while (1)
    {
        for (i = 0; i < 10; i++)
        {
            buf[sizeof(buf) - 2] = '0' + i;

```

```

        rt_kprintf("thread2: send message - %s\n", buf);
        /* 发送消息到消息队列中 */
        result = rt_mq_send(&mq, &buf[0], sizeof(buf));
        if ( result == -RT_EFULL)
        {
            /* 消息队列满, 延迟1s时间 */
            rt_kprintf("message queue full, delay 1s\n");
            rt_thread_delay(100);
        }
    }

    /* 延时10个OS Tick */
    rt_thread_delay(10);
}

/* 线程3入口函数 */
static void thread3_entry(void* parameter)
{
    char buf[] = "this is an urgent message!";

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

        /* 发送紧急消息到消息队列中 */
        rt_mq_urgent(&mq, &buf[0], sizeof(buf));

        /* 延时25个OS Tick */
        rt_thread_delay(25);
    }
}

int messageq_simple_init()
{
    /* 初始化消息队列 */
    rt_mq_init(&mq, "mq",
        &msg_pool[0], /* 内存池指向msg_pool */
        128 - sizeof(void*), /* 每个消息的大小是 128 - void* */
        sizeof(msg_pool), /* 内存池的大小是msg_pool的大小 */
        RT_IPC_FLAG_FIFO); /* 如果有多个线程等待, 按照FIFO的方法分配消息 */

    /* 创建线程1 */
    tid1 = rt_thread_create("t1",

```

```

        thread1_entry, /* 线程入口是thread1_entry */
        RT_NULL,      /* 入口参数是RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
if (tid1 != RT_NULL)
    rt_thread_startup(tid1);
else
    tc_stat(TC_STAT_END | TC_STAT_FAILED);

/* 创建线程2 */
tid2 = rt_thread_create("t2",
    thread2_entry, /* 线程入口是thread2_entry */
    RT_NULL,      /* 入口参数是RT_NULL */
    THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
if (tid2 != RT_NULL)
    rt_thread_startup(tid2);
else
    tc_stat(TC_STAT_END | TC_STAT_FAILED);

/* 创建线程3 */
tid3 = rt_thread_create("t3",
    thread3_entry, /* 线程入口是thread3_entry */
    RT_NULL,      /* 入口参数是RT_NULL */
    THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
if (tid3 != RT_NULL)
    rt_thread_startup(tid3);
else
    tc_stat(TC_STAT_END | TC_STAT_FAILED);

return 0;
}

#ifdef RT_USING_TC
static void _tc_cleanup()
{
    /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
    rt_enter_critical();

    /* 删除线程 */
    if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid1);
    if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid2);
    if (tid3 != RT_NULL && tid3->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid3);
}

```

```

    /* 执行消息队列对象脱离 */
    rt_mq_detach(&mq);

    /* 调度器解锁 */
    rt_exit_critical();

    /* 设置TestCase状态 */
    tc_done(TC_STAT_PASSED);
}

int _tc_messageq_simple()
{
    /* 设置TestCase清理回调函数 */
    tc_cleanup(_tc_cleanup);
    messageq_simple_init();

    /* 返回TestCase运行的最长时间 */
    return 100;
}

/* 输出函数命令到finsh shell中 */
FINSH_FUNCTION_EXPORT(_tc_messageq_simple, a message queue example);
#else
/* 用户应用入口 */
int rt_application_init()
{
    messageq_simple_init();

    return 0;
}
#endif

```

4.7.4 使用场合

消息队列可以应用于发送不定长消息的场合，包括线程与线程间的消息交换，以及中断服务例程中发送给线程的消息（中断服务例程不可能接收消息）。

典型使用

消息队列和邮箱的明显不同是消息的长度并不限定在4个字节以内，另外消息队列也包括了一个发送紧急消息的函数接口。但是当创建的是一个所有消息的最大长度是4字节的消息队列时，消息队列对象将蜕化成邮箱。这个不限定长度的消息，也及时的反应到了代码编写的场合上，同样是类似邮箱的代码：

```

struct msg
{
    rt_uint8_t *data_ptr;    /* 数据块首地址 */

```

```

    rt_uint32_t data_size;          /* 数据块大小 */
};

```

和邮箱例子相同的消息结构定义，假设依然需要发送这么一个消息给接收线程。在邮箱例子中，这个结构只能够发送指向这个结构的指针（在函数指针被发送过去后，接收线程能够正确的访问指向这个地址的内容，通常这块数据需要留给接收线程来释放）。而使用消息队列的方式则大不相同：

```

void send_op(void *data, rt_size_t length)
{
    struct msg msg_ptr;

    msg_ptr.data_ptr = data; /* 指向相应的数据块地址 */
    msg_ptr.data_size = length; /* 数据块的长度 */

    /* 发送这个消息指针给mq消息队列 */
    rt_mq_send(mq, (void*)&msg_ptr, sizeof(struct msg));
}

```

注意，上面的代码中，是把一个局部变量的数据内容发送到了消息队列中。在接收线程中，同样也采用局部变量进行消息接收的结构体：

```

void message_handler()
{
    struct msg msg_ptr; /* 用于放置消息的局部变量 */

    /* 从消息队列中接收消息到msg_ptr中 */
    if (rt_mq_recv(mq, (void*)&msg_ptr, sizeof(struct msg)) == RT_EOK)
    {
        /* 成功接收到消息，进行相应的数据处理 */
    }
}

```

因为消息队列是直接的数据内容复制，所以在上面的例子中，都采用了局部变量的方式保存消息结构体，这样也就免去动态内存分配的烦恼了（也就不用担心，接收线程在接收到消息时，消息内存空间已经被释放）。

同步消息

在一般的系统设计中会经常遇到要发送同步消息的问题，这个时候就可以根据当时的状态选择相应的实现：两个线程间可以采用[消息队列+信号量或邮箱]的形式实现。发送线程通过消息发送的形式发送相应的消息给消息队列，发送完毕后希望获得接收线程的收到确认，工作示意图如图 同步消息发送 所示：

根据消息确认的不同，可以把消息结构体定义成：

```

struct msg
{
    /* 消息结构其他成员 */
    struct rt_mailbox ack;
}

```

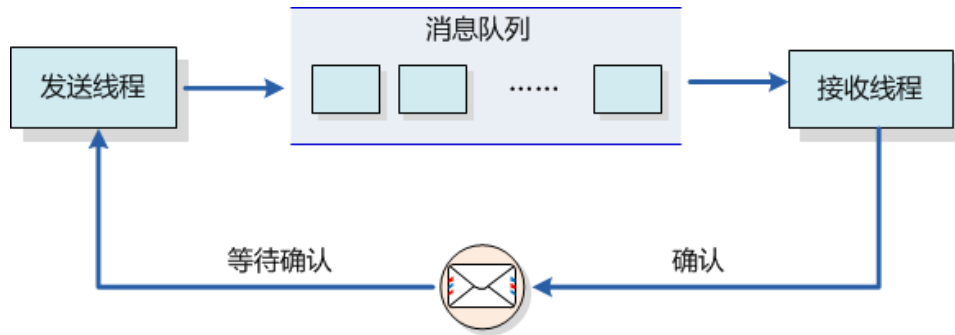


图 4.10: 同步消息发送

```

};
/* 或者 */
struct msg
{
    /* 消息结构其他成员 */
    struct rt_semaphore ack;
};

```

第一种类型的消息使用了邮箱来作为确认标志，而第二种类型的消息采用了信号量来作为确认标志。邮箱做为确认标志，代表着接收线程能够通知一些状态值给发送线程；而信号量作为确认标志只能够单一的通知发送线程，消息已经确认接收。

第 5 章

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，只有在实际使用时才将它们从存储空间调入到中央处理器内部进行运算。通常存储空间可以分为两种：内部存储空间和外部存储空间。内部存储空间访问速度比较快，能够按照变量地址随机地访问，也就是我们通常所说的RAM（随机存储器），或电脑的内存；而外部存储空间内所保存的内容相对来说比较固定，即使掉电后数据也不会丢失，这就是通常所讲的ROM（只读存储器），也可以把它理解为电脑的硬盘。在这一章中我们主要讨论内部存储空间的管理。

由于实时系统中对时间要求的严格性，内存分配往往要比通用操作系统要求苛刻得多。

- 首先，分配内存的时间必须是确定的。一般内存管理算法是根据需要存储的数据的长度在内存中去寻找一个与这段数据相适应的空闲内存块，然后将数据存储在里面。而寻找这样一个空闲内存块所耗费的时间是不确定的，因此对于实时系统来说，这就是不可接受的，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，否则实时任务对外部事件的响应也将变得不可确定。
- 其次，随着内存不断被分配和释放，整个内存区域会产生越来越多的碎片（因为在使用过程中，申请了一些内存，其中一些释放了，导致内存空间中存在一些小的内存块，它们地址不连续，不能够作为一整块的大内存分配出去），系统中还有足够的空闲内存，但因为它们地址并非连续，不能组成一块连续的完整内存块，会使得程序不能申请到大的内存。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(每个月或者数月进行一次)，但是对于那些需要常年不间断地工作于野外的嵌入式系统来说，就变得让人无法接受了。
- 最后，嵌入式系统的资源环境也是不尽相同，有些系统的资源比较紧张，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，如何为这些不同的系统，选择适合它们的高效率的内存分配算法，就将变得复杂化。

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，有针对性的提供了不同的内存分配管理算法。总体上可分为两类：静态分区内存管理与动态内存管理，而动态内存管理又根据可用内存的多少划分为两种情况：一种是针对小内存块的分配管理（小内存管理算法），另一种是针对大内存块的分配管理（SLAB管理算法）。

5.1 静态内存池管理

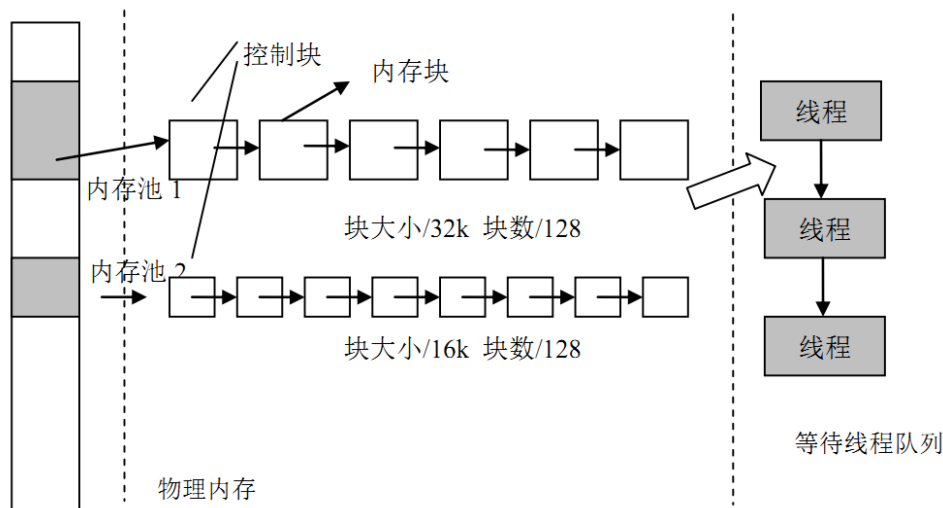


图 5.1: 内存池管理结构示意图

5.1.1 静态内存池工作原理

内存池管理结构示意图 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/释放的速度。

内存池在创建时先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，小内存块直接通过链表连接起来（此链表也称为空闲链表）。每次分配的时候，从空闲链表中取出链头上第一个内存块，提供给申请者。从图中可以看到，物理内存中允许存在多个大小不同的内存池，每一个内存池又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给了一个内存池控制块，内存控制块的参数包括内存池名，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的分配内存块申请，当获得这些信息后，内核就可以从内存池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。

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 */

    /* 内存池数据区域中能够容纳的最大内存块数 */
    rt_size_t block_total_count;
    /* 内存池中空闲的内存块数 */
    rt_size_t block_free_count;
    /* 因为内存块不可用而挂起的线程列表 */
}
```

```
    rt_list_t      suspend_thread;
    /* 因为内存块不可用而挂起的线程数 */
    rt_size_t      suspend_thread_count;
};
typedef struct rt_mempool* rt_mp_t;
```

每一个内存池对象由上述结构组成，其中suspend_thread形成了一个申请线程等待列表，即当内存池中无可用内存块，并且申请线程允许等待时，申请线程将挂起在suspend_thread链表上。

5.1.2 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从堆上分配一个内存池。创建内存池是从对应内存池中分配和释放内存块的先决条件，创建内存池后，线程便可以从内存池中执行申请、释放等操作。创建内存池使用下面的函数接口，该函数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);
```

使用该函数接口可以创建一个与需求的内存块大小、数目相匹配的内存池，前提当然是在系统资源允许的情况下（最主要的是动态堆内存资源）才能创建成功。创建内存池时，需要给内存池指定一个名称。然后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缓冲区，接着初始化内存池对象，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。

函数参数

参数	描述
name	内存池名；
block_count	内存块数量；
block_size	内存块容量。

函数返回

创建内存池对象成功，将返回内存池的句柄；否则返回RT_NULL。

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。使用下面的函数接口：

```
rt_err_t rt_mp_delete(rt_mp_t mp);
```

删除内存池时，会首先唤醒等待在该内存池对象上的所有线程（返回-RT_ERROR），然后再释放已从内存堆上分配的内存池数据存放区域，然后删除内存池对象。

函数参数

参数	描述
mp	rt_mp_create返回的内存池对象句柄。

函数返回

返回RT_EOK

初始化内存池

初始化内存池跟创建内存池类似，只是初始化内存池用于静态内存管理模式，内存池控制块来源于用户在系统中申请的静态对象。另外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。函数接口如下：

```
rt_err_t rt_mp_init(rt_mp_t mp, const char* name, void *start, rt_size_t size, rt_size_t block size);
```

初始化内存池时，把需要进行初始化的内存池对象传递给内核，同时需要传递的还有内存池用到的内存空间，以及内存池管理的内存块数目和块大小，并且给内存池指定一个名称。这样，内核就可以对该内存池进行初始化，将内存池用到的内存空间组织成可用于分配的空闲块链表。

函数参数

参数	描述
mp	内存池对象；
name	内存池名；
start	内存池的起始位置；
size	内存池数据区域大小；
block_size	内存块容量。

函数返回

初始化成功返回RT_OK；否则返回-RT_ERROR。

脱离内存池

脱离内存池将把内存池对象从内核对象管理器中删除。脱离内存池使用下面的函数接口：

```
rt_err_t rt_mp_detach(rt_mp_t mp);
```

使用该函数接口后，内核先唤醒所有等待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。

函数参数

参数	描述
mp	内存池对象。

函数返回

返回RT_EOK。

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_t 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；若等待时间大于零，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。

函数参数

参数	描述
mp	内存池对象；
time	超时时间。

函数返回

成功时返回分配的内存块地址，失败时返回RT_NULL。

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);
```

使用该函数接口时，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（或所属于的）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。

函数参数

参数	描述
block	内存块指针。

函数返回

无

内存池使用的例程如下所示：

```

/*
 * 程序清单：内存池例程
 *
 * 这个程序会创建一个静态的内存池对象，2 个动态线程。
 * 两个线程会试图分别从内存池中获得内存块
 */
#include <rtthread.h>
#include "tc_comm.h"
static rt_uint8_t *ptr[48];
static rt_uint8_t mempool[4096];
static struct rt_mempool mp; /* 静态内存池对象 */
/* 指向线程控制块的指针 */
static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    int i;
    char *block;
    while(1)
    {
        for (i = 0; i < 48; i++)
        {
            /* 申请内存块 */
            rt_kprintf("allocate No.%d\n", i);
            if (ptr[i] == RT_NULL)
            {
                ptr[i] = rt_mp_alloc(&mp, RT_WAITING_FOREVER);
            }
        }

        /* 继续申请一个内存块，因为已经没有内存块，线程应该被挂起 */
        block = rt_mp_alloc(&mp, RT_WAITING_FOREVER);
        rt_kprintf("allocate the block mem\n");
        /* 释放这个内存块 */
        rt_mp_free(block);
        block = RT_NULL;
    }
}

/* 线程 2 入口，线程 2 的优先级比线程 1 低，应该线程 1 先获得执行。*/
static void thread2_entry(void *parameter)

```

```

{
    int i;
    while(1)
    {
        rt_kprintf("try to release block\n");
        for (i = 0 ; i < 48; i ++ )
        {
            /* 释放所有分配成功的内存块 */
            if (ptr[i] != RT_NULL)
            {
                rt_kprintf("release block %d\n", i);
                rt_mp_free(ptr[i]);
                ptr[i] = RT_NULL;
            }
        }

        /* 休眠 10 个 OS Tick */
        rt_thread_delay(10);
    }
}

int mempool_simple_init()
{
    int i;
    for (i = 0; i < 48; i ++ ) ptr[i] = RT_NULL;
    /* 初始化内存池对象 */
    rt_mp_init(&mp, "mp1", &mempool[0], sizeof(mempool), 80);

    /* 创建线程 1 */
    tid1 = rt_thread_create("t1",
                            thread1_entry, /* 线程入口是 thread1_entry */
                            RT_NULL, /* 入口参数是 RT_NULL */
                            THREAD_STACK_SIZE, THREAD_PRIORITY,
                            THREAD_TIMESLICE);

    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);
    else
        tc_stat(TC_STAT_END | TC_STAT_FAILED);

    /* 创建线程 2 */
    tid2 = rt_thread_create("t2",
                            thread2_entry, /* 线程入口是 thread2_entry */
                            RT_NULL, /* 入口参数是 RT_NULL */
                            THREAD_STACK_SIZE, THREAD_PRIORITY + 1,
                            THREAD_TIMESLICE);
}

```



```

    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
    else
        tc_stat(TC_STAT_END | TC_STAT_FAILED);
    return 0;
}
#ifdef RT_USING_TC
static void _tc_cleanup()
{
    /* 调度器上锁，上锁后，将不再切换到其他线程，仅响应中断 */
    rt_enter_critical();
    /* 删除线程 */
    if (tid1 != RT_NULL && tid1->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid1);
    if (tid2 != RT_NULL && tid2->stat != RT_THREAD_CLOSE)
        rt_thread_delete(tid2);
    /* 执行内存池脱离 */
    rt_mp_detach(&mp);
    /* 调度器解锁 */
    rt_exit_critical();
    /* 设置 TestCase 状态 */
    tc_done(TC_STAT_PASSED);
}
int _tc_mempool_simple()
{
    /* 设置 TestCase 清理回调函数 */
    tc_cleanup(_tc_cleanup);
    mempool_simple_init();
    /* 返回 TestCase 运行的最长时间 */
    return 100;
}
/* 输出函数命令到 finsh shell 中 */
FINSH_FUNCTION_EXPORT(_tc_mempool_simple, a memory pool
example);
#else
/* 用户应用入口 */
int rt_application_init()
{
    mempool_simple_init();
    return 0;
}
#endif

```

5.2 动态内存管理

动态内存管理是一个真实的堆（Heap）内存管理模块，可以在当前资源满足的情况下，根据用户的需求分配任意大小的内存块。而当用户不需要再使用这些内存块时，又可以释放回堆中供其他应用分配使用。RT-Thread系统为了满足不同需求，提供了两套不同的动态内存管理算法，分别是小堆内存管理算法和SLAB内存管理算法。

小堆内存管理模块主要针对系统资源比较少，一般用于小于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，提供了一种近似多内存池管理算法的快速算法。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。这两种管理模块提供的API接口完全相同。

- 警告：因为动态内存管理器要满足多线程情况下的安全分配，会考虑多线程间的互斥问题，所以请不要在中断服务例程中分配或释放动态内存块。因为它可能会引起当前上下文被挂起等待。

5.2.1 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。初始时，它是一块大的内存。当需要分配内存块时，将从这个大的内存块上分割出相匹配的内存块，然后把分割出来的空闲内存块还回给堆管理系统中。每个内存块都包含一个管理用的数据头，通过这个头把使用块与空闲块用双向链表的方式链接起来，如 内存块链表 图所示：

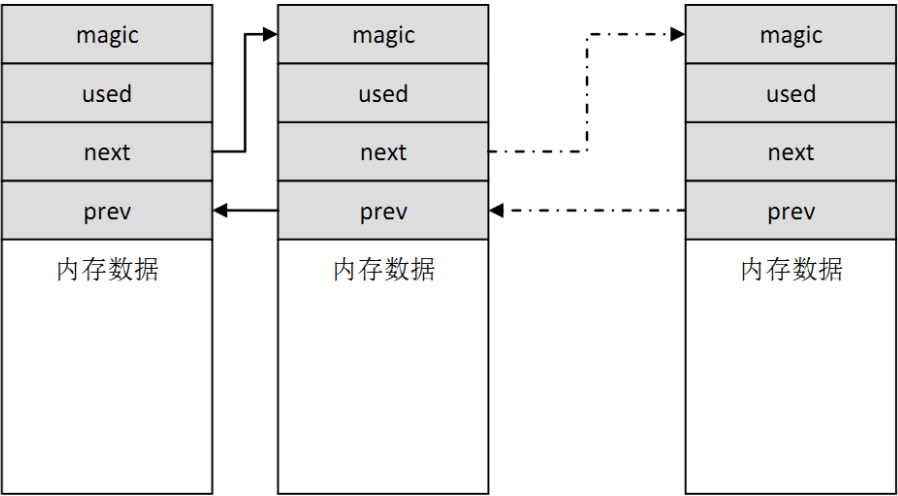


图 5.2: 内存块链表

每个内存块（不管是已分配的内存块还是空闲的内存块）都包含一个数据头，其中包括：

magic - 变数（或称为幻数），它会被初始化成0x1ea0（即英文单词heap），用于标记这个内存块是一个内存管理用的内存数据块；

used - 指示出当前内存块是否已经分配。

magic变数不仅仅用于标识这个数据块是一个内存管理用的内存数据块，实质也是一个内存保护字：如果这个区域被改写，那么也就意味着这块内存块被非法改写（正常情况下只有内存管理器才会去碰这块内存）。

内存管理的在表现主要体现在内存的分配与释放上，小型内存管理算法可以用以下例子体现出来。

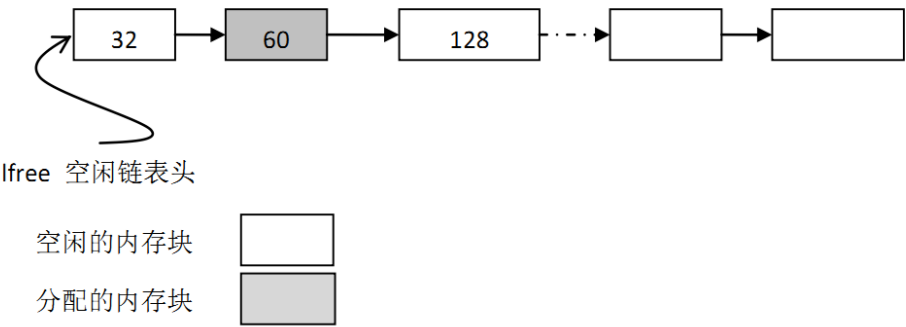


图 5.3: 小内存管理算法链表结构示意图

如 小内存管理算法链表结构示意图 所示的内存分配情况，空闲链表指针lfree初始指向32字节的内存块。当用户线程要再分配一个64字节的内存块时，但此lfree指针指向的内存块只有32字节并不能满足要求，内存管理器会继续寻找下一内存块，当找到再下一块内存块，128字节时，它满足分配的要求。因为这个内存块比较大，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 分配64 字节后的链表结构 所示。

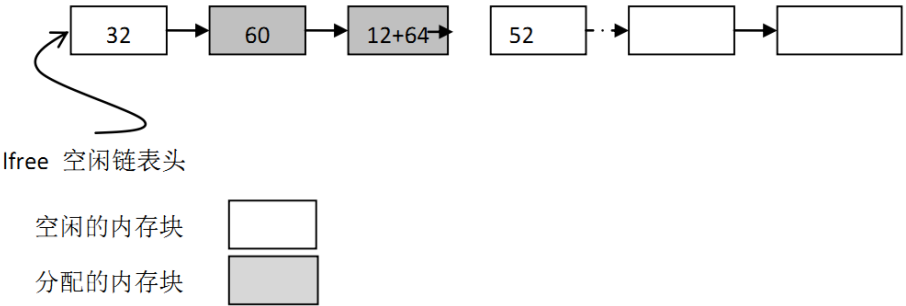


图 5.4: 分配64 字节后的链表结构

另外，在每次分配内存块前，都会留出12字节数据头用于magic，used信息及链表节点使用。返回给应用的地址实际上是这块内存块12字节以后的地址，前面的12字节数据头是用户永远不应该碰的部分。（注：12字节数据头长度会与系统对齐差异而有所不同）

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。

5.2.2 SLAB内存管理模块

RT-Thread的SLAB分配器是在DragonFly BSD创始人Matthew Dillon实现的SLAB分配器基础上，针对嵌入式系统优化的内存分配算法。最原始的SLAB算法是Jeff Bonwick为Solaris操作系统而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 SLAB 内存分配器结构 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

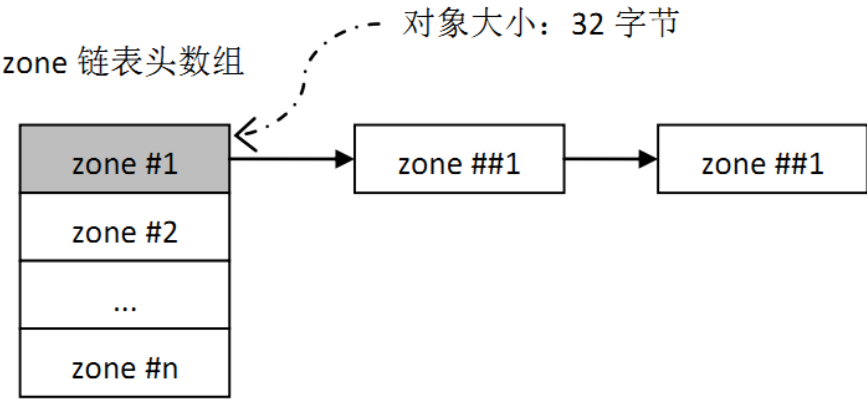


图 5.5: SLAB 内存分配器结构

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 array）中统一管理。

下面是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），那么就取相应的空闲块。如果分配完成后，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，那么当zone链表中全空闲zone达到一定数目后，系统就会把这个全空闲的zone释放到页面分配器中去。

5.2.3 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，可以通过下面的函数接口完成：

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

这个函数会把参数begin_addr，end_addr区域的内存空间作为内存堆来使用。

函数参数

参数	描述
----	----

begin_addr	堆内存区域起始地址；
end_addr	堆内存区域结束地址。

函数返回
无

分配内存块

从内存堆上分配用户指定大小的内存块，函数接口如下：

```
void* rt_malloc(rt_size_t nbytes);
```

rt_malloc函数会从系统堆空间中找到合适大小的内存块，然后把内存块可用地址返回给用户。

函数参数

参数	描述
nbytes	申请的内存大小。

函数返回
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），可以通过下面的函数接口完成：

```
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）。

函数参数

参数	描述
rmem	指向已分配的内存块；
newsize	重新分配的内存大小。

函数返回
返回重新分配的内存块地址；

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，可以通过下面的函数接口完成：

```
void *rt_calloc(rt_size_t count, rt_size_t size);
```

函数参数

参数	描述
count	内存块数量；
size	内存块容量。

函数返回

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，释放内存块的函数接口如下：

```
void rt_free (void *ptr);
```

rt_free函数会把待释放的内存还回给堆管理器中。在调用这个函数时用户需传递待释放的内存块指针，如果是空指针直接返回。

函数参数

参数	描述
ptr	待释放的内存块指针。

函数返回

无

设置分配钩子函数

在分配内存块过程中，用户可设置一个钩子函数，调用的函数接口如下：

```
void rt_malloc_sethook(void (*hook)(void *ptr, rt_size_t size));
```

设置的钩子函数会在内存分配完成后进行回调。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。

函数参数

参数	描述
hook	钩子函数指针。

函数返回
无
其中hook函数接口如下：

```
void hook(void *ptr, rt_size_t size);
```

函数参数

参数	描述
ptr	分配到的内存块指针；
size	分配到的内存块的大小。

函数返回
无

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，调用的函数接口如下：

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

设置的钩子函数会在调用内存释放完成前进行回调。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）。

函数参数

参数	描述
hook	钩子函数指针。

函数返回
无
其中hook函数接口如下：

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

函数参数

参数	描述
ptr	待释放的内存块指针。

函数返回

无

动态内存堆使用 的例程如下所示：

```
/* 线程TCB和栈*/
struct rt_thread_t 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()
{
    rt_err_t result;

    /* 初始化线程对象*/
```



```
    result = rt_thread_init(&thread1,
                           "thread1",
                           thread1_entry, RT_NULL,
                           &thread1_stack[0], sizeof(thread1_stack),
                           200, 100);

    if (result == RT_EOK)
        rt_thread_startup(&thread1);

    return 0;
}
```

5.3 更改情况

- RT-Thread 1.2.0中引入RT_USING_MEMHEAP_AS_HEAP选项，可以把多个memheap（地址可不连续）粘合起来用于系统的heap分配；
- RT-Thread 1.2.0中引入rt_memheap_realloc函数，用于在memheap中进行memory重新分配；

第 6 章

I/O设备管理

绝大部分的嵌入式系统都包括一些输入输出(I/O)设备，例如仪器上的数据显示，工业设备上的串口通信，数据采集设备上用于保存数据的flash或SD卡，以及网络设备的以太网接口都是嵌入式系统中容易找到的I/O设备例子。嵌入式系统通常都是针对具有专有特殊需求的设备而设计的，例如移动电话、MP3播放器就是典型地为处理I/O设备而建造的嵌入式系统例子。

在RT-Thread实时操作系统中，RT-Thread提供了一套简单的I/O设备管理框架，如图RT-Thread I/O 设备结构 所示，它把I/O设备分成了三层进行处理：

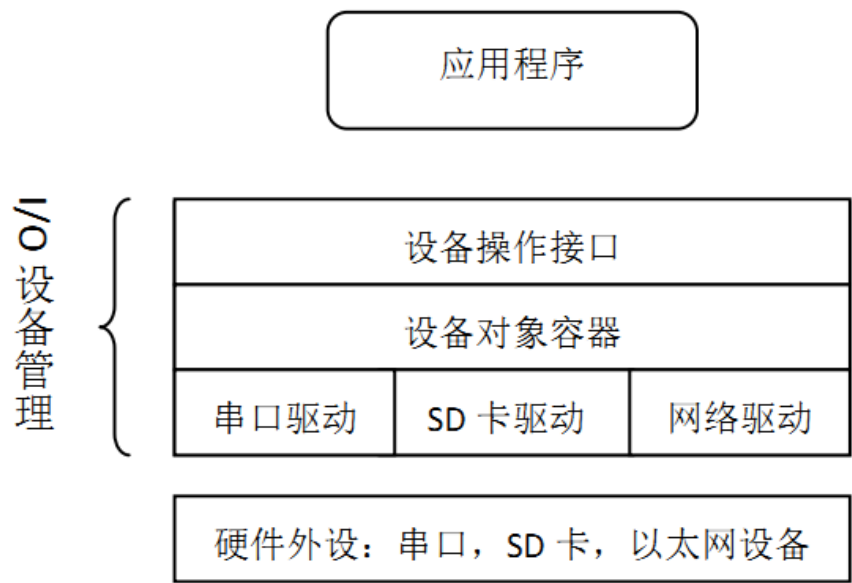


图 6.1: RT-Thread I/O 设备结构

应用程序通过RT-Thread的设备操作接口获得正确的设备驱动，然后通过这个设备驱动与底层I/O硬件设备进行数据（或控制）交互。RT-Thread提供给上层应用的是一个抽象的设备接口，给下层设备提供的是底层驱动框架。从系统整体位置来说I/O设备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设备模块实现了对设备驱动程序的封装。应用程序通过I/O设备模块提供的标准接口访问底层设备，设备驱动程序的升级、更替不会对上层应用产生影响。这种方式使得设备的硬件操作相关的代码能够独立于应用程序而存在，双方只需关注各自的功能实现，从而降低

了代码的耦合性、复杂性，提高了系统的可靠性。

RT-Thread的设备模型是建立在内核对象模型基础之上的。在第4章中我们已经介绍过RT-Thread的内核对象管理器，读者若对这部分还不太了解，可以再回顾下这一章节。在RT-Thread中，设备也被认为是一类对象，被纳入对象管理器的范畴。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。设备继承关系图是设备对象的继承和派生关系示意图。

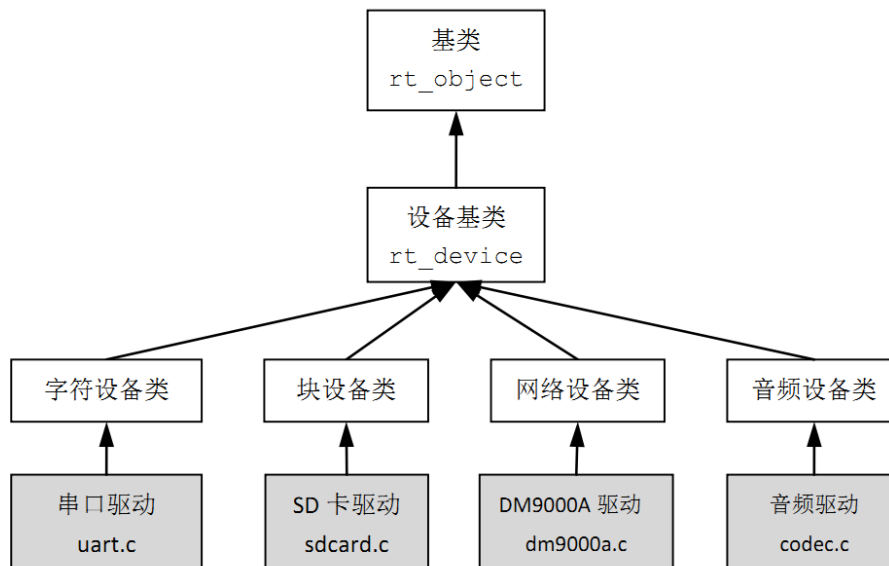


图 6.2: 设备继承关系图

6.1 块设备

在传统操作系统中一般将I/O设备分成字符设备、块设备和网络接口，分类的依据是设备数据与系统之间的传输处理方式。

字符模式设备允许非结构的数据传输，即通常数据传输采用串行的形式，每次一个字节。字符设备通常是一些简单设备，如串口、按键。

块设备每次传输一个数据块，例如每次传输512个字节数据。这个数据块是硬件强制性的，数据块可能使用某类数据接口或某些强制性的传输协议，否则就可能发生错误。因此，有时块设备驱动程序对读或写操作必须执行附加的工作，如图 块设备 所示。

当系统服务于一个具有大量数据的写操作时，设备驱动程序必须首先将数据划分为多个包，每个包采用设备指定的数据尺寸。而在实际过程中，最后一部分数据尺寸有可能小于正常的设备块尺寸。如图 块设备 中每个块使用单独的写请求写入到设备中，头3个直接进行写操作。但最后一个数据块尺寸小于设备块尺寸，设备驱动程序必须使用不同于前3个块的方式处理最后的数据块。通常情况下，设备驱动程序需要首先执行相对应的设备块的读操作，然后把写入数据覆盖到读出数据上，然后再把这个“合成”的数据块做为一整块写回到设备中。例如图 块设备 中的块4，驱动程序需要先把块4所对应的设备块读出来，然后将需要写入的数据覆盖至从设备块读出的数据上，使其合并成一个新的块，最后再写回到块设备中。

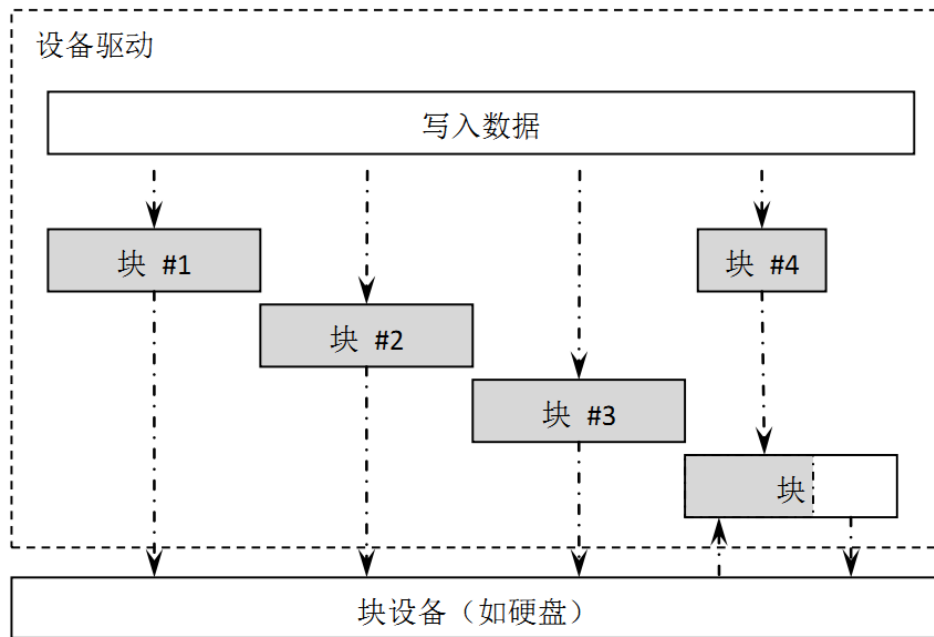


图 6.3: 块设备

6.2 I/O设备控制块

```

struct rt_device
{
    struct rt_object parent;

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

    /* 提供给上层应用的回调函数 */
    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(由驱动程序提供) */
    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);

    /* 设备的私有数据 */
    void* user_data;
};

typedef struct rt_device* rt_device_t;

```

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_RTC,              /* RTC设备       */
    RT_Device_Class_Sound,            /* 声音设备      */
    RT_Device_Class_Graphic,          /* 图形设备      */
    RT_Device_Class_I2CBUS,           /* I2C总线       */
    RT_Device_Class_USBDevice,        /* USB device设备 */
    RT_Device_Class_USBHost,          /* USB host设备  */
    RT_Device_Class_SPIBUS,           /* SPI总线       */
    RT_Device_Class_SPIDevice,        /* SPI设备       */
    RT_Device_Class_SDIO,             /* SDIO设备      */
    RT_Device_Class_PM,               /* 电源管理设备  */
    RT_Device_Class_Pipe,             /* 管道设备      */
    RT_Device_Class_Portal,           /* 双向管道设备  */
    RT_Device_Class_Timer,            /* 定时器设备    */
    RT_Device_Class_Miscellaneous,    /* 杂类设备      */
    RT_Device_Class_Unknown           /* 未知设备      */
};
```

从设备控制块，我们可以看到，每个设备对象都会在内核中维护一个设备控制块结构，这种结构使设备对象继承rt_object基类，然后形成rt_device设备类型。

6.3 I/O设备管理接口

6.3.1 注册设备

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，并添加一些相应的一些属性。这些注册的设备均可以通过设备名，采用“查找设备接口”的方式从系统中查找，从而获得该设备控制块（或设备句柄）。注册设备的函数接口如下：

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

函数参数

参数	描述
dev	设备句柄；
name	设备名称；
flags	设备模式标志：

flags参数支持下列参数(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 */
```

设备流模式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。

函数返回

返回RT_EOK

- 警告：应当避免重复注册已经注册的设备，以及注册相同名字的设备。

6.3.2 移除设备

将设备从设备系统中移除，被卸载的设备将不能再通过“查找设备接口”被查找到。卸载设备的函数接口如下所示：

```
rt_err_t rt_device_unregister(rt_device_t dev)
```

函数参数

参数	描述
dev	设备句柄。

函数返回

返回RT_EOK

- 注：卸载设备并不会释放设备控制块所占用的内存。

6.3.3 初始化所有设备

初始化所有注册到设备对象管理器中的未初始化的设备，可以通过如下函数接口完成：

```
rt_err_t rt_device_init_all(void)
```

函数参数

无

函数返回

返回RT_EOK

- 注：此函数将逐渐废弃，不推荐在应用程序中调用。当一个设备初始化完成后它的flags域中的RT_DEVICE_FLAG_ACTIVATED应该被置位。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。

6.3.4 查找设备

根据指定的设备名称查找设备，可以通过如下接口完成：

```
rt_device_t rt_device_find(const char* name)
```

使用这个函数接口时，系统会在设备对象类型所对应的对象容器中遍历寻找设备对象，然后返回该设备的句柄，如果没有找到相应的设备对象，则返回RT_NULL。

函数参数

参数	描述
name	设备名称。

函数返回

查找到对应设备将返回相应的设备句柄；否则返回RT_NULL

6.3.5 初始化设备

初始化指定设备，可以通过如下函数接口完成：

```
rt_err_t rt_device_init(rt_device_t dev)
```

函数参数

参数	描述
dev	设备句柄；

函数返回

返回驱动的init函数返回值

6.3.6 打开设备

根据设备控制块来打开设备，可以通过如下函数接口完成：


```
rt_err_t rt_device_open (rt_device_t dev, rt_uint16_t oflags)
```

函数参数

参数	描述
dev	设备句柄;
oflags	访问模式。

其中oflags支持以下列表中的参数:

```
#define RT_DEVICE_OFLAG_CLOSE    0x000 /* 设备已经关闭（内部使用） */
#define RT_DEVICE_OFLAG_RDONLY  0x001 /* 以只读方式打开设备      */
#define RT_DEVICE_OFLAG_WRONLY  0x002 /* 以只写方式打开设备      */
#define RT_DEVICE_OFLAG_RDWR    0x003 /* 以读写方式打开设备      */
#define RT_DEVICE_OFLAG_OPEN    0x008 /* 设备已经打开（内部使用） */

#define RT_DEVICE_FLAG_STREAM    0x040 /* 设备以流模式打开        */
#define RT_DEVICE_FLAG_INT_RX    0x100 /* 设备以中断接收模式打开  */
#define RT_DEVICE_FLAG_DMA_RX    0x200 /* 设备以DMA接收模式打开   */
#define RT_DEVICE_FLAG_INT_TX    0x400 /* 设备以中断发送模式打开  */
#define RT_DEVICE_FLAG_DMA_TX    0x800 /* 设备以DMA发送模式打开   */
```

函数返回

返回驱动的open函数返回值

- 注: 如果设备注册时指定的参数中包括RT_DEVICE_FLAG_STANDALONE参数, 此设备将不允许重复打开, 返回-RT_EBUSY。
- 注: 如果上层应用程序需要设置设备的接收回调函数, 则必须以INT_RX或者DMA_RX的方式打开设备, 否则不会回调函数。

6.3.7 关闭设备

根据设备控制块来关闭设备, 可以通过如下函数接口完成:

```
rt_err_t rt_device_close(rt_device_t dev)
```

函数参数

参数	描述
dev	设备句柄。

函数返回

返回驱动的close函数返回值

6.3.8 读设备

从设备中读取，或获得数据，可以通过如下函数接口完成：

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

调用这个函数，会从设备dev中获得数据，并存放在buffer缓冲区中。这个缓冲区的最大长度是size。pos根据不同的设备类别存在不同的意义。

函数参数

参数	描述
dev	设备句柄；
pos	读取数据偏移量；
buffer	内存缓冲区指针，读取的数据将会被保存在缓冲区中；
size	读取数据的大小。

函数返回

返回读到数据的实际大小（如果是字符设备，返回大小以字节为单位；如果是块设备，返回的大小以块为单位）；如果返回0，则需要读取当前线程的errno来判断错误状态。

6.3.9 写设备

向设备中写入数据，可以通过如下函数接口完成：

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

调用这个函数，会把缓冲区buffer中的数据写入到设备dev中。写入数据的最大长度是size。pos根据不同的设备类别存在不同的意义。

函数参数

参数	描述
dev	设备句柄；
pos	写入数据偏移量；
buffer	内存缓冲区指针，放置要写入的数据；
size	写入数据的大小。

函数返回

返回写入数据的实际大小(如果是字符设备，返回大小以字节为单位；如果是块设备，返回的大小以块为单位)；如果返回0，则需要读取当前线程的errno来判断错误状态

- 注：在RT-Thread的块设备中，从1.0.0版本开始，rt_device_read()/rt_device_write()接口的pos、size参数以块为单位。0.3.x以前的版本则以字节为单位。

6.3.10 控制设备

根据设备控制块来控制设备，可以通过下面的函数接口完成：

```
rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)
```

函数参数

参数	描述
dev	设备句柄；
cmd	命令控制字，这个参数通常与设备驱动程序相关；
arg	控制的参数。

函数返回

返回驱动控制接口的返回值

6.3.11 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以通知用程序有数据到达。可以通过如下函数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 size))
```

在调用这个函数时，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后，立刻从设备中读取数据。

函数参数

参数	描述
dev	设备句柄；
rx_ind	接收回调函数。

函数返回

返回RT_EOK

6.3.12 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的发送完成后(例如DMA发送完成或FIFO已经写入完毕产生完成中断时)被调用。可以通过如下函数接口设置设备发送完成指示：

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

调用这个函数时，回调函数tx_done参数由调用者提供，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内存块或将其做为下一个写数据的缓存。

函数参数

参数	描述
dev	设备句柄；
tx_done	发送回调函数。

函数返回

返回RT_EOK

6.4 设备驱动

上一节说到了如何使用RT-Thread的设备接口，但对于底层来说，如何编写一个设备驱动程序可能会更为重要，本节将详细描述如何编写一个设备驱动程序，并以STM32上的一个串口设备为例子进行说明。

6.4.1 设备驱动必须实现的接口

在6.1节中提及了RT-Thread设备接口类，我们着重看看其中包含的一套公共设备接口(类似上节说的设备访问接口，但面向的层次已经不一样，这里是面向底层驱动)：

```
/* 公共的设备接口(由驱动程序提供) */
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);
```

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（如 设备操作接口与设备驱动程序接口的映射）：

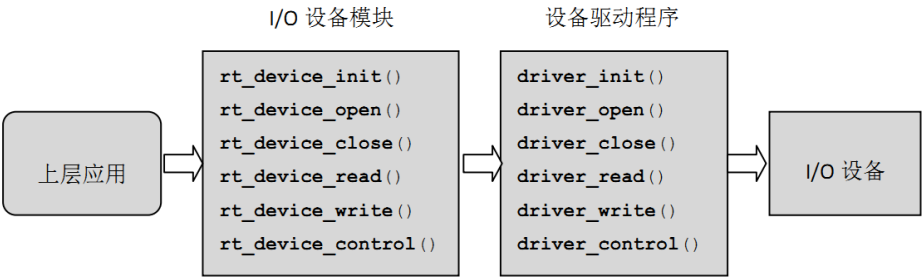


图 6.4: 设备操作接口与设备驱动程序接口的映射

即这些驱动实现的底层接口是上层应用最终访问的落脚点，例如上层应用调用 `rt_device_read` 接口进行设备读取数据操作，上层应先调用 `rt_device_find` 获得相对应的设备句柄，而在调用 `rt_device_read` 时，就是使用这个设备句柄所对应驱动的 `driver_read`。上述的接口是一一对应关系。

I/O设备模块提供的这六个接口（`rt_device_init/open/read/write/control`），对应到设备驱动程序的六个接口（`driver_init/open/read/write/control`等），可以认为是底层设备驱动必须提供的接口：

方法名称	方法描述
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)。如果设备控制块中的flag标志已经设置成激活状态，那么再运行初始化接口时，会立刻返回，而不会重新进行初始化。
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；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并开始接收数据。所以建议在写底层驱动程序时，在调用open接口时才使能设备。
close	关闭设备。建议在打开设备时，设备驱动自行维护一个打开计数，在打开设备时进行+1操作，在关闭设备时进行-1操作，当计数器变为0时，进行真正的关闭操作。
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指定偏移量，例如串口设备，设备驱动应忽略这个参数。而对于块设备来说，pos以及size都是以块设备的数据块大小做为单位的。例如块设备的数据块大小是512，而参数中pos = 10, size = 2，那么驱动应该返回设备中第10个块（从第0个块做为起始），共计2个块的数据。这个接口返回的类型是rt_size_t，即读到的字节数或块数目。正常情况下应该会返回参数中size的数值，如果返回零请设置对应的errno值
write	向设备中写入数据。参数pos指出写入数据的偏移量。与读操作类似，对于块设备来说，pos以及size都是以块设备的数据块大小做为单位的。这个接口返回的类型是rt_size_t，即真实写入数据的字节数或块数目。正常情况下应该会返回参数中size的数值，如果返回零请设置对应的errno值。

control	根据不同的cmd命令控制设备。命令往往是由底层各类设备驱动自定义实现。例如参数RT_DEVICE_CTRL_BLK_GETGEOME，意思是获取块设备的大小信息。
---------	---

6.4.2 设备驱动实现的步骤

在实现一个RT-Thread设备时，可以按照如下的步骤进行（对于一些复杂的设备驱动，例如以太网接口驱动、图形设备驱动，请参看网络组件、GUI部分章节）：

- 按照RT-Thread的对象模型，扩展一个对象有两种方式：
 - 定义自己的私有数据结构，然后赋值到RT-Thread设备控制块的用户_data指针上；
 - 从struct rt_device结构中进行派生。
- 实现RT-Thread I/O设备模块中定义的6个公共设备接口，开始可以是空函数(返回类型是rt_err_t的可默认返回RT_EOK)；
- 根据自己的设备类型定义自己的私有数据域。特别是在可能有多个相类似设备的情况下（例如串口1、2），设备接口可以共用同一套接口，不同的只是各自的数据域(例如寄存器基地址)；
- 根据设备的类型，注册到RT-Thread设备框架中。

6.4.3 STM32F10x的串口驱动

以下例子详细分析了STM32F10x的串口驱动，也包括上层应该如何使用这个设备的代码。STM32F10x串口驱动代码，详细的中文注释已经放在其中了。

目前的串口驱动采用了从struct rt_device结构中进行派生的方式，派生出rt_serial_device。STM32F10x的串口驱动包括公用的rt_serial_device串口驱动框架和属于STM32F10x的uart驱动两部分。串口驱动框架位于components/drivers/serial/serial.c中，向上层提供如下函数：

- rt_serial_init
- rt_serial_open
- rt_serial_close
- rt_serial_read
- rt_serial_write
- rt_serial_control

uart驱动位于bsp/stm32f10x/drivers/usart.c中，向上层提供如下函数：

- stm32_configure
- stm32_control
- stm32_putc
- stm32_getc

uart驱动位于底层，实际运行中串口驱动框架将调用uart驱动提供的函数。例如：应用程序调用rt_device_write时，实际调用关系为：

```
rt_device_write ==> rt_serial_write ==> stm32_putc
```

下面将首先列出串口驱动框架的代码，由于我们仅以中断接收和轮询发送方式来举例，其他接收和发送方式的代码将省略。驱动框架代码如下：

```
/* RT-Thread设备驱动框架接口 */

/* serial.h部分内容开始 */
/* Default config for serial_configure structure */
#define RT_SERIAL_CONFIG_DEFAULT \
{ \
    BAUD_RATE_115200, /* 115200 bits/s */ \
    DATA_BITS_8,     /* 8 databits */ \
    STOP_BITS_1,      /* 1 stopbit */ \
    PARITY_NONE,      /* No parity */ \
    BIT_ORDER_LSB,    /* LSB first sent */ \
    NRZ_NORMAL,       /* Normal mode */ \
    RT_SERIAL_RB_BUFSZ, /* Buffer size */ \
    0 \
}

/* 串口配置结构体 */
struct serial_configure
{
    rt_uint32_t baud_rate;

    rt_uint32_t data_bits      :4;
    rt_uint32_t stop_bits     :2;
    rt_uint32_t parity         :2;
    rt_uint32_t bit_order      :1;
    rt_uint32_t invert         :1;
    rt_uint32_t bufsz          :16;
    rt_uint32_t reserved       :4;
};

/*
 * Serial FIFO mode
 */
struct rt_serial_rx_fifo
{
    /* software fifo */
    rt_uint8_t *buffer;

    rt_uint16_t put_index, get_index;
};

/* 串口设备结构体 */
struct rt_serial_device
```

```

{
    struct rt_device      parent;

    const struct rt_uart_ops *ops;
    struct serial_configure  config;

    void *serial_rx;
    void *serial_tx;
};
typedef struct rt_serial_device rt_serial_t;

/**
 * uart operators
 * 函数的具体实现在bsp/stm32f10x/drivers/usart.c中
 */
struct rt_uart_ops
{
    rt_err_t (*configure)(struct rt_serial_device *serial, struct serial_configure *cfg);
    rt_err_t (*control)(struct rt_serial_device *serial, int cmd, void *arg);

    int (*putc)(struct rt_serial_device *serial, char c);
    int (*getc)(struct rt_serial_device *serial);

    rt_size_t (*dma_transmit)(struct rt_serial_device *serial, const rt_uint8_t *buf, rt_size_t size);
};
/* serial.h部分内容结束 */

/* 以下为serial.c的内容 */
/* 轮询接收 */
rt_inline int _serial_poll_rx(struct rt_serial_device *serial, rt_uint8_t *data, int length)
{
    /* 代码省略 */
}

/* 轮询发送 */
rt_inline int _serial_poll_tx(struct rt_serial_device *serial, const rt_uint8_t *data, int length)
{
    int size;
    RT_ASSERT(serial != RT_NULL);

    size = length;
    while (length)
    {
        /*
         * to be polite with serial console add a line feed

```



```

        * to the carriage return character
        */
    if (*data == '\n' && (serial->parent.open_flag & RT_DEVICE_FLAG_STREAM))
    {
        serial->ops->putc(serial, '\r');
    }

    /* 实际调用usart.c中的stm32_putc,
     * serial->ops中包含uart驱动中的4个函数指针,
     * 在usart.c的rt_hw_usart_init函数中, 向系统注册设备之前对其赋值 。
     */
    serial->ops->putc(serial, *data);

    ++ data;
    -- length;
}

return size - length;
}

/* 中断接收
 * 中断处理函数rt_hw_serial_isr将收到的数据放到接收buffer里,
 * 此函数从接收buffer里取出数据
 */
rt_inline int _serial_int_rx(struct rt_serial_device *serial, rt_uint8_t *data, int length)
{
    int size;
    struct rt_serial_rx_fifo* rx_fifo;

    RT_ASSERT(serial != RT_NULL);
    size = length;

    rx_fifo = (struct rt_serial_rx_fifo*) serial->serial_rx;
    RT_ASSERT(rx_fifo != RT_NULL);

    /* read from software FIFO */
    while (length)
    {
        int ch;
        rt_base_t level;

        /* disable interrupt */
        level = rt_hw_interrupt_disable();
        if (rx_fifo->get_index != rx_fifo->put_index)
        {

```

```

        ch = rx_fifo->buffer[rx_fifo->get_index];
        rx_fifo->get_index += 1;
        if (rx_fifo->get_index >= serial->config.bufsz) rx_fifo->get_index = 0;
    }
    else
    {
        /* no data, enable interrupt and break out */
        rt_hw_interrupt_enable(level);
        break;
    }

    /* enable interrupt */
    rt_hw_interrupt_enable(level);

    *data = ch & 0xff;
    data++; length--;
}

return size - length;
}

/* 中断发送 */
rt_inline int _serial_int_tx(struct rt_serial_device *serial, const rt_uint8_t *data, int length)
{
    /* 代码省略 */
}

/* DMA接收 */
rt_inline int _serial_dma_rx(struct rt_serial_device *serial, rt_uint8_t *data, int length)
{
    /* 代码省略 */
}

/* DMA发送 */
rt_inline int _serial_dma_tx(struct rt_serial_device *serial, const rt_uint8_t *data, int length)
{
    /* 代码省略 */
}

static rt_err_t rt_serial_init (struct rt_device *dev)
{
    rt_err_t result = RT_EOK;
    struct rt_serial_device *serial;

    RT_ASSERT(dev != RT_NULL);

```

```

/* 获得真实的serial设备对象 */
serial = (struct rt_serial_device *)dev;

/* initialize rx/tx */
serial->serial_rx = RT_NULL;
serial->serial_tx = RT_NULL;

/* 实际调用usart.c中stm32_configure函数，对串口波特率等进行配置。
 * serial->config包含配置数据，在usart.c的rt_hw_usart_init函数中，向系统注册设备之前进行赋值。
 */
if (serial->ops->configure)
    result = serial->ops->configure(serial, &serial->config);

return result;
}

/* 打开设备 */
static rt_err_t rt_serial_open(struct rt_device *dev, rt_uint16_t oflag)
{
    struct rt_serial_device *serial;

    RT_ASSERT(dev != RT_NULL);
    serial = (struct rt_serial_device *)dev;

    /* check device flag with the open flag */
    if ((oflag & RT_DEVICE_FLAG_DMA_RX) && !(dev->flag & RT_DEVICE_FLAG_DMA_RX))
        return -RT_EIO;
    if ((oflag & RT_DEVICE_FLAG_DMA_TX) && !(dev->flag & RT_DEVICE_FLAG_DMA_TX))
        return -RT_EIO;
    if ((oflag & RT_DEVICE_FLAG_INT_RX) && !(dev->flag & RT_DEVICE_FLAG_INT_RX))
        return -RT_EIO;
    if ((oflag & RT_DEVICE_FLAG_INT_TX) && !(dev->flag & RT_DEVICE_FLAG_INT_TX))
        return -RT_EIO;

    /* get open flags */
    dev->open_flag = oflag & 0xff;

    /* initialize the Rx/Tx structure according to open flag */
    if (serial->serial_rx == RT_NULL)
    {
        if (oflag & RT_DEVICE_FLAG_DMA_RX)
        {
            /* 代码省略 */
        }
        else if (oflag & RT_DEVICE_FLAG_INT_RX)

```

```

{
    struct rt_serial_rx_fifo* rx_fifo;

    /* 创建中断接收buffer */
    rx_fifo = (struct rt_serial_rx_fifo*) rt_malloc (sizeof(struct rt_serial_rx_fifo) +
        serial->config.bufsz);
    RT_ASSERT(rx_fifo != RT_NULL);
    rx_fifo->buffer = (rt_uint8_t*) (rx_fifo + 1);
    rt_memset(rx_fifo->buffer, 0, serial->config.bufsz);
    rx_fifo->put_index = 0;
    rx_fifo->get_index = 0;

    /* 保存指向接收buffer的指针, 以便在接收中断函数rt_hw_serial_isr中使用 */
    serial->serial_rx = rx_fifo;
    dev->open_flag |= RT_DEVICE_FLAG_INT_RX;
    /* 调用usart.c中的stm32_control函数
    * 开启uart接收中断
    */
    serial->ops->control(serial, RT_DEVICE_CTRL_SET_INT, (void *)RT_DEVICE_FLAG_INT_RX);
}
else
{
    serial->serial_rx = RT_NULL;
}
}

if (serial->serial_tx == RT_NULL)
{
    if (oflag & RT_DEVICE_FLAG_DMA_TX)
    {
        /* 代码省略 */
    }
    else if (oflag & RT_DEVICE_FLAG_INT_TX)
    {
        /* 代码省略 */
    }
    else
    {
        serial->serial_tx = RT_NULL;
    }
}

return RT_EOK;
}

```

```

/* 关闭设备 */
static rt_err_t rt_serial_close(struct rt_device *dev)
{
    struct rt_serial_device *serial;

    RT_ASSERT(dev != RT_NULL);
    serial = (struct rt_serial_device *)dev;

    /* 实际上只有ref_count为0时本函数才会被调用到,
     * 即最后一个打开此设备的应用调用rt_device_close时, 本函数才会被调用
     * 所以下面一条语句并不起作用
     */
    if (dev->ref_count > 1) return RT_EOK;

    if (dev->open_flag & RT_DEVICE_FLAG_INT_RX)
    {
        struct rt_serial_rx_fifo* rx_fifo;

        rx_fifo = (struct rt_serial_rx_fifo*)serial->serial_rx;
        RT_ASSERT(rx_fifo != RT_NULL);
        /* 释放中断接收buffer */
        rt_free(rx_fifo);
        serial->serial_rx = RT_NULL;
        dev->open_flag &= ~RT_DEVICE_FLAG_INT_RX;
        /* 关闭接收中断 */
        serial->ops->control(serial, RT_DEVICE_CTRL_CLR_INT, (void*)RT_DEVICE_FLAG_INT_RX);
    }
    else if (dev->open_flag & RT_DEVICE_FLAG_DMA_RX)
    {
        /* 代码省略 */
    }

    if (dev->open_flag & RT_DEVICE_FLAG_INT_TX)
    {
        /* 代码省略 */
    }
    else if (dev->open_flag & RT_DEVICE_FLAG_DMA_TX)
    {
        /* 代码省略 */
    }

    return RT_EOK;
}

/* 从设备中读取数据 */

```

```

static rt_size_t rt_serial_read (struct rt_device *dev,
                                rt_off_t         pos,
                                void              *buffer,
                                rt_size_t         size)
{
    struct rt_serial_device *serial;

    RT_ASSERT(dev != RT_NULL);
    if (size == 0) return 0;

    serial = (struct rt_serial_device *)dev;

    if (dev->open_flag & RT_DEVICE_FLAG_INT_RX)
    {
        /* 调用中断接收函数 */
        return _serial_int_rx(serial, buffer, size);
    }
    else if (dev->open_flag & RT_DEVICE_FLAG_DMA_RX)
    {
        return _serial_dma_rx(serial, buffer, size);
    }

    return _serial_poll_rx(serial, buffer, size);
}

/* 向设备中写入数据 */
static rt_size_t rt_serial_write(struct rt_device *dev,
                                 rt_off_t         pos,
                                 const void        *buffer,
                                 rt_size_t         size)
{
    struct rt_serial_device *serial;

    RT_ASSERT(dev != RT_NULL);
    if (size == 0) return 0;

    serial = (struct rt_serial_device *)dev;

    if (dev->open_flag & RT_DEVICE_FLAG_INT_TX)
    {
        return _serial_int_tx(serial, buffer, size);
    }
    else if (dev->open_flag & RT_DEVICE_FLAG_DMA_TX)
    {
        return _serial_dma_tx(serial, buffer, size);
    }
}

```

```

    }
    else
    {
        /* 轮询模式发送 */
        return _serial_poll_tx(serial, buffer, size);
    }
}

/* 设备控制操作 */
static rt_err_t rt_serial_control (struct rt_device *dev,
                                   rt_uint8_t cmd,
                                   void *args)
{
    struct rt_serial_device *serail;

    RT_ASSERT(dev != RT_NULL);

    /* 获得真正的串口对象 */
    serial = (struct rt_serial_device *)dev;

    switch (cmd)
    {
    case RT_DEVICE_CTRL_SUSPEND:
        /* 挂起设备 */
        break;
        dev->flag |= RT_DEVICE_FLAG_SUSPENDED;

    case RT_DEVICE_CTRL_RESUME:
        /* 唤醒设备 */
        dev->flag &= ~RT_DEVICE_FLAG_SUSPENDED;
        break;
    case RT_DEVICE_CTRL_CONFIG:
        /* 配置设备, 设置波特率、数据位数等 */
        serial->ops->configure(serial, (struct serial_configure *)args);
        break;

    default :
        /* control device */
        serial->ops->control(serial, cmd, args);
        break;
    }

    return RT_EOK;
}

```

```

/*
 * 向系统中注册串口设备
 */
rt_err_t rt_hw_serial_register(struct rt_serial_device *serial,
                               const char             *name,
                               rt_uint32_t            flag,
                               void                   *data)
{
    struct rt_device *device;
    RT_ASSERT(serial != RT_NULL);

    device = &(serial->parent);

    /* 设置设备驱动类型 */
    device->type      = RT_Device_Class_Char;
    device->rx_indicate = RT_NULL;
    device->tx_complete = RT_NULL;
    /* 设置设备驱动公共接口函数 */
    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;
    /* 在使用stm32f10x时, 此处传给data的是指向stm32_uart结构体的指针 */
    device->user_data   = data;

    /* 注册一个字符设备 */
    return rt_device_register(device, name, flag);
}

/* ISR for serial interrupt */
void rt_hw_serial_isr(struct rt_serial_device *serial, int event)
{
    switch (event & 0xff)
    {
        /* 接收中断 */
        case RT_SERIAL_EVENT_RX_IND:
        {
            int ch = -1;
            rt_base_t level;
            struct rt_serial_rx_fifo* rx_fifo;

            /* 获取中断接收buffer, serial->serial_rx在rt_serial_open里进行赋值 */
            rx_fifo = (struct rt_serial_rx_fifo*)serial->serial_rx;

```



```

RT_ASSERT(rx_fifo != RT_NULL);

while (1)
{
    /* 实际调用stm32_getc */
    ch = serial->ops->getc(serial);
    if (ch == -1) break;

    /* disable interrupt */
    level = rt_hw_interrupt_disable();

    rx_fifo->buffer[rx_fifo->put_index] = ch;
    rx_fifo->put_index += 1;
    if (rx_fifo->put_index >= serial->config.bufsz) rx_fifo->put_index = 0;

    /* if the next position is read index, discard this 'read char' */
    if (rx_fifo->put_index == rx_fifo->get_index)
    {
        /* 丢弃最旧的数据, buffer里能保存最多bufsz - 1个字节 */
        rx_fifo->get_index += 1;
        if (rx_fifo->get_index >= serial->config.bufsz) rx_fifo->get_index = 0;
    }

    /* enable interrupt */
    rt_hw_interrupt_enable(level);
}

/* 调用接收回调函数
 * rx_indicate在device.c文件rt_device_set_rx_indicate函数中赋值
 * 对于finsh在finsh_thread_entry中进行设置
 */
if (serial->parent.rx_indicate != RT_NULL)
{
    rt_size_t rx_length;

    /* get rx length */
    level = rt_hw_interrupt_disable();
    rx_length = (rx_fifo->put_index >= rx_fifo->get_index)? (rx_fifo->put_index - rx_fifo->get_index) :
        (serial->config.bufsz - (rx_fifo->get_index - rx_fifo->put_index));
    rt_hw_interrupt_enable(level);

    serial->parent.rx_indicate(&serial->parent, rx_length);
}
break;

```

```

    }
    case RT_SERIAL_EVENT_TX_DONE:
    {
        /* 代码省略 */
    }
    case RT_SERIAL_EVENT_TX_DMADONE:
    {
        /* 代码省略 */
    }
    case RT_SERIAL_EVENT_RX_DMADONE:
    {
        /* 代码省略 */
    }
}

}

```

uart驱动位于bsp/stm32f10x/drivers/usart.c中，代码如下：

```

/* STM32 uart driver */
struct stm32_uart
{
    USART_TypeDef* uart_device;
    IRQn_Type irq;
};

static rt_err_t stm32_configure(struct rt_serial_device *serial, struct serial_configure *cfg)
{
    struct stm32_uart* uart;
    USART_InitTypeDef USART_InitStructure;

    RT_ASSERT(serial != RT_NULL);
    RT_ASSERT(cfg != RT_NULL);
    /* serial->parent.user_data即device->user_data在设备注册时进行赋值 */
    uart = (struct stm32_uart *)serial->parent.user_data;

    USART_InitStructure.USART_BaudRate = cfg->baud_rate;

    if (cfg->data_bits == DATA_BITS_8){
        USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    } else if (cfg->data_bits == DATA_BITS_9) {
        USART_InitStructure.USART_WordLength = USART_WordLength_9b;
    }

    if (cfg->stop_bits == STOP_BITS_1){
        USART_InitStructure.USART_StopBits = USART_StopBits_1;
    } else if (cfg->stop_bits == STOP_BITS_2){

```

```

        USART_InitStructure.USART_StopBits = USART_StopBits_2;
    }

    if (cfg->parity == PARITY_NONE){
        USART_InitStructure.USART_Parity = USART_Parity_No;
    } else if (cfg->parity == PARITY_ODD) {
        USART_InitStructure.USART_Parity = USART_Parity_Odd;
    } else if (cfg->parity == PARITY_EVEN) {
        USART_InitStructure.USART_Parity = USART_Parity_Even;
    }

    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
    USART_Init(uart->uart_device, &USART_InitStructure);

    /* Enable USART */
    USART_Cmd(uart->uart_device, ENABLE);

    return RT_EOK;
}

static rt_err_t stm32_control(struct rt_serial_device *serial, int cmd, void *arg)
{
    struct stm32_uart* uart;

    RT_ASSERT(serial != RT_NULL);
    uart = (struct stm32_uart *)serial->parent.user_data;

    switch (cmd)
    {
        /* disable interrupt */
        case RT_DEVICE_CTRL_CLR_INT:
            /* disable rx irq */
            UART_DISABLE_IRQ(uart->irq);
            /* 关闭接收中断 */
            USART_ITConfig(uart->uart_device, USART_IT_RXNE, DISABLE);
            break;
            /* enable interrupt */
        case RT_DEVICE_CTRL_SET_INT:
            /* enable rx irq */
            UART_ENABLE_IRQ(uart->irq);
            /* 打开接收中断 */
            USART_ITConfig(uart->uart_device, USART_IT_RXNE, ENABLE);
            break;
    }
}

```

```

    return RT_EOK;
}

static int stm32_putc(struct rt_serial_device *serial, char c)
{
    struct stm32_uart* uart;

    RT_ASSERT(serial != RT_NULL);
    uart = (struct stm32_uart *)serial->parent.user_data;

    uart->uart_device->DR = c;
    while (!(uart->uart_device->SR & USART_FLAG_TC));

    return 1;
}

static int stm32_getc(struct rt_serial_device *serial)
{
    int ch;
    struct stm32_uart* uart;

    RT_ASSERT(serial != RT_NULL);
    uart = (struct stm32_uart *)serial->parent.user_data;

    ch = -1;
    if (uart->uart_device->SR & USART_FLAG_RXNE)
    {
        ch = uart->uart_device->DR & 0xff;
    }

    return ch;
}

static const struct rt_uart_ops stm32_uart_ops =
{
    stm32_configure,
    stm32_control,
    stm32_putc,
    stm32_getc,
};

#ifdef RT_USING_UART1
/* UART1 device driver structure */
struct stm32_uart uart1 =

```

```

{
    USART1,
    USART1_IRQn,
};
struct rt_serial_device serial1;

void USART1_IRQHandler(void)
{
    struct stm32_uart* uart;

    uart = &uart1;

    /* enter interrupt */
    rt_interrupt_enter();
    /* 接收中断 Read data register not empty */
    if(USART_GetITStatus(uart->uart_device, USART_IT_RXNE) != RESET)
    {
        /* 调用中断处理函数, 处理接收中断 */
        rt_hw_serial_isr(&serial1, RT_SERIAL_EVENT_RX_IND);
        /* clear interrupt */
        USART_ClearITPendingBit(uart->uart_device, USART_IT_RXNE);
    }
    /* 发送完成中断 Transmission complete */
    if (USART_GetITStatus(uart->uart_device, USART_IT_TC) != RESET)
    {
        /* clear interrupt */
        USART_ClearITPendingBit(uart->uart_device, USART_IT_TC);
    }
    /* Overrun error */
    if (USART_GetFlagStatus(uart->uart_device, USART_FLAG_ORE) == SET)
    {
        stm32_getc(&serial1);
    }
    /* leave interrupt */
    rt_interrupt_leave();
}
#endif

/* UART2至UART4相关代码省略 */

static void RCC_Configuration(void)
{
    #if defined(RT_USING_UART1)
        /* Enable UART GPIO clocks */
        RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
    #endif
}

```

```

    /* Enable UART clock */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
#endif /* RT_USING_UART1 */

/* UART2至UART4相关代码省略 */
}

static void GPIO_Configuration(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;

#ifdef RT_USING_UART1
    /* Configure USART Rx/tx PIN */
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Pin = USART1_GPIO_RX;
    GPIO_Init(USART1_GPIO, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Pin = USART1_GPIO_TX;
    GPIO_Init(USART1_GPIO, &GPIO_InitStructure);
#endif /* RT_USING_UART1 */

/* UART2至UART4相关代码省略 */
}

static void NVIC_Configuration(struct stm32_uart* uart)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    /* Enable the USART Interrupt */
    NVIC_InitStructure.NVIC_IRQChannel = uart->irq;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

void rt_hw_usart_init(void)
{
    struct stm32_uart* uart;
    /* 配置默认波特率、数据位等 */
    struct serial_configure config = RT_SERIAL_CONFIG_DEFAULT;

```

```

    RCC_Configuration();
    GPIO_Configuration();

#if defined(RT_USING_UART1)
    uart = &uart1;
    config.baud_rate = BAUD_RATE_115200;

    /* 接口函数赋值 */
    serial1.ops = &stm32_uart_ops;
    /* 配置赋值 */
    serial1.config = config;

    NVIC_Configuration(&uart1);

    /* 注册设备uart1
     * 第4个参数uart最终被传给device->user_data
     */
    rt_hw_serial_register(&serial1, "uart1",
                        RT_DEVICE_FLAG_RDWR | RT_DEVICE_FLAG_INT_RX ,
                        uart);
#endif /* RT_USING_UART1 */

/* UART2至UART4相关代码省略 */
}

```

对于包含中断发送、接收的情况的驱动程序，以下例子给出了具体的使用代码。在这个例子中，用户线程将从两个设备上(uart1, uart2)读取数据，然后再写到uart1设备中。

```

/*
 * 程序清单：串口设备操作例程
 *
 * 在这个例程中，将启动一个devt线程，然后打开串口1和2
 * 当串口1和2有输入时，将读取其中的输入数据然后写入到
 * 串口1设备中。
 *
 */
#include <rtthread.h>

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

/* 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];

/* 数据到达回调函数*/
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;

    /* 查找系统中的串口1设备 */
    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|RT_DEVICE_FLAG_INT_RX);
    }
    /* 设置写设备为uart1设备 */
    write_device = device;

    /* 查找系统中的串口2设备 */
    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_rcv(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，系统将先查找是否存在uart1, uart2这两个设备，如果存在则设置数据接收回调函数。在数据接收回调函数中，系统将对应的设备句柄、接收到的数据长度填充到一个消息结构体（struct rx_msg）上，然后发送到消息队列rx_mq中。devt线程在打开设备后，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，直到达到唤醒条件被唤醒，被唤醒的条件是devt线程收到消息或0.5秒(50 OS tick, 在RT_TICK_PER_SECOND设置为100时)内都没收到消息。可以根据rt_mq_rcv函数返回值的不同，区分出devt线程是因为什么原因而被唤醒。如果devt线程是因为接收到消息而被唤醒（rt_mq_rcv函数的返回值是RT_EOK），那么它将主动调用rt_device_read去读取消息，然

后写入uart1设备中。

6.4.4 finsh使用uart设备分析

1.注册设备

我们首先要注册设备，才能用rt_device_find查找到设备。然后就可以进行初始化、打开、读取等操作。首先在各线程启动之前进行设备注册，函数调用关系如下：

```
main ==> rt_thread_startup ==> rt_hw_board_init ==> rt_hw_uart_init ==> rt_hw_serial_register ==> rt_device_register
```

在函数rt_thread_startup中最后才调用了rt_system_scheduler_start各线程才开始运行。可以确定线程在查找设备时设备已经存在。

在rtconfig.h中有#define RT_CONSOLE_DEVICE_NAME "uart1",注册完设备后设置了终端设备

```
rt_hw_board_init ==> rt_console_set_device(RT_CONSOLE_DEVICE_NAME)
```

finsh将使用此终端设备。

2.启动finsh线程

在shell.c中有代码INIT_COMPONENT_EXPORT(finsh_system_init)，采用宏INIT_COMPONENT_EXPORT导出的函数将在rt_components_init函数中被调用。调用关系：

```
rt_thread_startup ==> rt_application_init ==> rt_init_thread_entry
```

启动初始化线程，然后在初始化线程中启动了finsh线程：

```
rt_init_thread_entry ==> rt_components_init ==> finsh_system_init ==> finsh_thread_entry
```

3.打开设备

在finsh线程中打开设备，相关代码如下：

```
shell->device = rt_console_get_device();
RT_ASSERT(shell->device);
rt_device_set_rx_indicate(shell->device, finsh_rx_ind);
rt_device_open(shell->device, (RT_DEVICE_OFLAG_RDWR | RT_DEVICE_FLAG_STREAM | RT_DEVICE_FLAG_INT_RX)
```

如果打开时设备没有进行初始化将首先进行初始化

```
rt_device_open ==> rt_serial_init ==> stm32_configure
```

stm32_configure进行了波特率等参数的配置。
然后打开设备

```
rt_device_open ==> rt_serial_open ==> stm32_control
```

stm32_control中将打开接收中断。

4.接收数据

finsh采用了中断接收方式打开设备，当uart收到数据时将产生中断

```
USART1_IRQHandler ==> rt_hw_serial_isr ==> stm32_getc
```

将收到的数据放到接收buffer里，然后调用回调函数finsh_rx_ind，finsh_rx_ind释放信号量，finsh线程得到信号量后读取数据。

```
finsh_thread_entry ==> rt_device_read ==> rt_serial_read ==> _serial_int_rx
```

_serial_int_rx函数从接收buffer中读取数据。

5.发送数据

在finsh中调用rt_kprintf函数进行输出

```
rt_kprintf ==> rt_device_write ==> rt_serial_write ==> _serial_poll_tx ==> stm32_putc
```

以上即finsh中对uart设备的使用分析。

第 7 章

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，如果不及时进行处理，轻则系统出错，重则会导致系统毁灭性地瘫痪。所以正确地处理异常，避免错误的发生是提高软件鲁棒性（稳定性）非常重要的一环，对于实时系统更是如此。

异常通常可以分成两类：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。同步异常与异步异常的区别还在于，同步异常触发后，系统必须立刻进行处理而不能够依然执行原有的程序指令步骤；而异步异常则可以延缓处理甚至是忽略，例如按键中断异常，虽然中断异常触发了，但是系统可以忽略它继续运行（同样也忽略了相应的按键事件）。

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。

7.1 中断处理过程

中断处理的一般过程如下图所示：

中断处理过程

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CPU连续不断地查询是否有相应的事件发生。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（在关闭中断期间，通常处理器会把新产生的中断挂起，当中断打开时立刻进行响应）。在执行中断服务例程的过程中，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，比如新的中断等待挂起直到当前中断处理离开后再行响应；或新的高优先级中断打断当前中断处理过程，而去直接响应这个更高优先级的新中断源。后面这种情况，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，不能使响应中断的时间尽量短。而在软件处理（软实时环境）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优先级更高的中断，过程如下图所示：

当正在执行一个中断服务例程（中断1）时，如果有更高的中断（中断2、中断3）触发，那么操作系统将先保存当前中断服务例程的上下文环境，然后转向中断2的中断服务例程，依此类推，直至中断3。当中断3的中断服务例程运行完成后，系统才恢复中断2的上下文环境，然后转回到中断2的中断服务例程去接着执行，依此类推，直至中断1。

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的一些特定时间量（图7-3）：

中断延迟TB定义为，从中断开始的时刻到中断服务例程开始执行的时刻之间的时间段。而中断服务例程的处理时间TC主要取决于中断服务例程处理的方法，对于不同的系统因其对中断处理的方法不同，其相应服务例程的时间需求也不一样。中断响应时间 $TD = TB + TC$ 。

7.2 中断栈

从上节的中断处理过程中，我们看到，在系统响应中断前，软件代码（或处理器）需要把当前任务的上下文保存下来（通常保存在当前任务的任务栈中），再调用中断服务例程进行中断响应、处理。在进行中断处理时（实质是调用用户的中断服务例程函数），中断处理函数中很可能会有自己的局部变量，这些都需要相应的栈空间来保存，所以中断响应依然需要一个栈空间来做为上下文运行中断处理函数。中断栈可以保存在打断任务的栈中，当中断中退出时，返回相应的任务继续执行。

中断栈也可以与打断任务栈完全分离开来，即每次进入中断时，在保存完打断任务上下文后，切换到新的中断栈中独立运行。在中断退出时，再做相应的上下文恢复。使用独立中断栈相对来说更容易实现，并且对于任务栈使用情况也比较容易了解掌握（否则必须要为中断栈预留空间，如果系统支持中断嵌套，还需要考虑应该为嵌套中断预留多大的空间）。

RT-Thread采用的方式是提供独立的中断栈，即中断发生时，中断的前期处理程序会将用户的栈指针更换到系统事先留出的中断栈空间中，等中断退出时再恢复用户的栈指针。这样中断就不会占用任务的栈空间，从而提高了内存空间的利用率，且随着任务的增加，这种减少内存占用的效果也越明显。

7.3 中断的底半处理

RT-Thread不对中断服务例程所需要的处理时间做任何假设、限制，但如同其它实时操作系统或非实时操作系统一样，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（相当于中断服务例程在系统中拥有最高的优先级，会抢占所有线程优先执行）。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，或自身中断源的下一次中断信号。

当一个中断发生时，中断服务例程需要取得相应的硬件状态或者数据。如果中断服务例程接下来要对状态或者数据进行简单处理，比如CPU时钟中断，中断服务例程只需对一个系统时钟tick变量进行加一操作，然后就结束中断服务例程。这类中断需要的运行时间往往都比较短。对于另外一些中断，中断服务例程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、底半部分（Bottom Half）。在Top Half中，取得硬件状态和数据后，打开被屏蔽的中断，给相关线程发送一条通知（可以是RT-Thread所提供的semaphore, event, mailbox或message queue等方式），然后结束中断服务例程；而接下来，相关的线程在接收到通知后，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（底半处理）。

7.3.1 底半处理实现范例

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（例7-1a），并假设接收到数据报文后，系统对报文的分析、处理是一个相对耗时的，比外部中断源信号重要性小许多的，而且在不屏蔽中断源信号情况下也能处理的过程。

```
/*
 * 程序清单：中断底半处理例子
 */

/* 用于唤醒线程的信号量 */
rt_sem_t demo_nw_isr;

/* 数据读取、分析的线程 */
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);

    /* ..其他的一些操作.. */

    /* 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);
}

```

这个例子的程序创建了一个nwt线程，这个线程在启动运行后，将阻塞在nw_bh_sem信号上，一旦这个信号量被释放，将执行接下来的nw_packet_parser过程，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，如下例。

```

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();

    /*..其他一些数据操作等..*/

    /* 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 */
}

```

从上面例子的两个代码片段可以看出，中断服务例程通过对一个信号量对象的等待和释放，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，必须认真考虑中断服务的处理时间是否大于给Bottom Half发送通知并处理的时间。

7.4 中断相关接口

为了尽量的使用户和系统底层异常、中断隔离开来，RT-Thread把中断和异常封装起来，以更友好的接口的形式提供给用户。（注：这部分的API由BSP提供，在某些处理器支持分支中并不一定存在，例如ARM Cortex-M0/M3分支中，请查询相关移植分支获得详细的实现情况）

7.4.1 装载中断服务例程

可调用如下的接口挂载一个新的中断服务例程：

```

rt_isr_handler_t rt_hw_interrupt_install(int vector,
                                         rt_isr_handler_t handler,
                                         void *param,
                                         char *name);

```


调用rt_hw_interrupt_install后，系统将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，程序则返回之前关联的这个中断服务例程。

函数参数

参数	描述
int vector	vector是挂载的中断号；
rt_isr_handler_t handler	新挂载的中断服务例程；
void *param	param会作为参数传递给中断服务例程；
char *name	中断的名称。

函数返回

函数返回挂载这个中断服务例程之前挂载的中断服务例程。

注意事项

这个API并不会出现在每一个移植分支中，例如通常Cortex-M0/M3/M4的移植分支中就没有这个API。

7.4.2 屏蔽中断源

通常在ISR准备处理某个中断信号之前，我们需要先屏蔽该中断源，以保证在接下来的处理过程中硬件状态或者数据不会受到干扰，我们可调用下面这个函数接口：

```
void rt_hw_interrupt_mask(int vector);
```

调用rt_hw_interrupt_mask函数接口后，相应的中断将会被屏蔽（通常当这个中断触发时，中断状态寄存器会有相应的变化，但并不送达到处理器进行处理）。

函数参数

参数	描述
int vector	vector是要屏蔽的中断号。

注意事项

这个API并不会出现在每一个移植分支中，例如通常Cortex-M0/M3/M4的移植分支中就没有这个API。

7.4.3 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，我们可调用下面的函数接口：

```
void rt_hw_interrupt_umask(int vector);
```

调用rt_hw_interrupt_umask函数接口后，如果中断（及对应外设）被正确时，中断触发后，将送到处理器进行处理。

函数参数

参数	描述
int vector	vector是要打开屏蔽的中断号。

注意事项

这个API并不会出现在每一个移植分支中，例如通常Cortex-M0/M3/M4的移植分支中就没有这个API。

7.4.4 关闭中断

当需要关闭整个系统的中断时，可调用下面的函数接口：

```
rt_base_t rt_hw_interrupt_disable(void);
```

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（因为整个系统已经不再对外部事件响应），也就是当前任务不会被抢占，除非这个任务主动让出处理器。

函数返回

函数返回中断前的系统中断状态。

7.4.5 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。调用的函数接口如下：

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

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

函数参数

参数	描述
rt_base_t level	level 是rt_hw_interrupt_disable函数返回的中断状态。

注意事项

调用这个接口并不代表着肯定打开中断，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。

7.4.6 与OS相关的中断接口

当整个系统被中断打断，进入中断处理函数时，OS需要知道当前已经进入到中断状态。针对这种情况，OS提供了两个函数：

```
void rt_interrupt_enter(void);  
void rt_interrupt_leave(void);
```

rt_interrupt_enter函数用于通知OS，当前已经进入了中断状态；rt_interrupt_leave函数用于通知OS，已经离开中断状态。通常来说，OS需要知道这样的运行状态，这样在中断服务例程中，如果调用了OS相关的调用，OS好及时调整相应的行为，例如进行任务切换时应该采取中断中任务切换的策略，而不是立即进行切换。但是如果中断服务例程很显然、很必然地，它不会去调用OS相关的函数，这个时候，也可以不调用rt_interrupt_enter/leave函数。

```
rt_uint8_t rt_interrupt_get_nest(void);
```

这个函数提供给上层应用，当前嵌套的中断深度。即，如果当前是中断上下文环境中，这个函数的返回值会大于0。

函数返回

函数返回当前系统中中断嵌套的深度，如果当前系统处于中断上下文环境中，返回值大于0。如果返回值大于1，说明当前出现了中断嵌套。

7.5 ARM Cortex-M中的中断与异常

ARM Cortex-M系列处理器与以往的ARM7TDMI、ARM920T相差很多，以往中断控制器都由IP授权的各家芯片厂商自行定义，而ARM Cortex-M则把中断控制器统一起来，命名为NVIC（嵌套向量中断控制）。正如其名，ARM Cortex-M NVIC支持中断嵌套功能：当一个中断触发并且系统进行响应时，处理器硬件会将当前运行的部分上下文寄存器自动压入中断栈中，这部分的寄存器包括PSR，R0，R1，R2，R3以及R12寄存器。当系统正在服务一个中断时，如果有一个更高优先级的中断触发，那么处理器同样的会打断当前运行的中断服务例程，然后把老的中断服务例程上下文的PSR，R0，R1，R2，R3和R12寄存器自动保存到中断栈中。这些部分上下文寄存器保存到中断栈的行为完全是硬件行为，这一点是与其他ARM处理器最大的区别（以往都需要依赖于软件保存上下文）。

另外，在ARM Cortex-M系列处理器上，所有中断都采用中断向量表的方式进行处理，即当一个中断触发时，处理器将直接判定是哪个中断源，然后直接跳转到相应的固定位置进行处理。而在ARM7、ARM9中，一般是先跳转进入IRQ入口，然后再由软件进行判断是哪个中断源触发，获得了相对应的中断服务例程入口地址后，再进行后续的中断处理。ARM7、ARM9的好处在于，所有中断它们都有统一的入口地址，便于OS的统一管理。而ARM Cortex-M系列处理器则恰恰相反，每个中断服务例程必须排列在一起放在统一的地址上（这个地址必须要设置到NVIC的中断向量偏移寄存器中）。

中断向量表一般由一个数组定义（或在起始代码中给出）。在STM32上，默认采用起始代码给出：

代码清单：初始化代码中的中断向量表

```
__Vectors      DCD      __initial_sp                ; Top of Stack
                DCD      Reset_Handler              ; Reset Handler
                DCD      NMI_Handler                 ; NMI Handler
                DCD      HardFault_Handler           ; Hard Fault Handler
                DCD      MemManage_Handler           ; MPU Fault Handler
                DCD      BusFault_Handler            ; Bus Fault Handler
                DCD      UsageFault_Handler          ; Usage Fault Handler
                DCD      0                           ; Reserved
                DCD      0                           ; Reserved
                DCD      0                           ; Reserved
                DCD      0                           ; Reserved
                DCD      SVC_Handler                 ; SVCall Handler
                DCD      DebugMon_Handler           ; Debug Monitor Handler
                DCD      0                           ; Reserved
                DCD      PendSV_Handler              ; PendSV Handler
                DCD      SysTick_Handler             ; SysTick Handler

... ..

NMI_Handler    PROC
                EXPORT NMI_Handler                  [WEAK]
                B       .
                ENDP

HardFault_Handler  PROC
                EXPORT HardFault_Handler            [WEAK]
                B       .
                ENDP

... ..
```

请注意代码后面的[WEAK]标识，它是符号弱化标识，在[WEAK]前面的符号如NMI_Handler、HardFault_Handler将被执行弱化处理，如果整个代码在链接时遇到了名称相同的符号（例如与NMI_Handler相同名称的函数），那么代码将使用未被弱化定义的符号（与NMI_Handler相同名称的函数），而与弱化符号相关的代码将被自动丢弃。

RT-Thread在Cortex-M系列上也遵循这样的方法，当用户需要使用自定义的中断服务例程时，只需要定义相同名称的函数覆盖弱化符号即可。例如用户需要自定义自己的串口2中断处理函数，那么可以在代码中自己实现USART2_IRQHandler函数，在系统编译链接时，中断向量表中将只保留这份USART2_IRQHandler函数，而不是经过WAEK修饰的USART2_IRQHandler函数。

7.6 外设中的中断模式与轮询模式

当编写一个外设驱动时，其编程模式到底采用中断模式触发还是轮询模式触发往往是驱动开发人员首先要考虑的问题，并且这个问题在实时操作系统与分时操作系统中差异还非常大。因为轮询模式本身采用顺序执行的方式：查询到相应的事件然后进行对应的处理。所以轮询模式从实现上来说，相对简单清晰。例如往串口中写入数据，仅当串口控制器写完一

个数据时，程序代码才写入下一个数据（否则这个数据丢弃掉）。相应的代码可以是这样的：

```
/* 轮询模式向串口写入数据 */
while (size)
{
    /* 判断UART外设中数据是否发送完毕 */
    while (!(uart->uart_device->SR & USART_FLAG_TXE));
    /* 当所有数据发送完毕后，才发送下一个数据 */
    uart->uart_device->DR = (*ptr & 0xFF);

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

但是在实时系统中轮询模式可能会出现非常大问题，因为在实时操作系统中，当一个程序持续地执行时（轮询时），它所在的线程会一直运行，比它优先级低的线程都不会得到运行。而分时系统中，这点恰恰相反，几乎没有优先级之分，可以在一个时间片运行这个程序，然后在另外一段时间片上运行另外一段程序。

所以通常情况下，实时系统中更多采用的是中断模式来驱动外设。当数据达到时，由中断唤醒相关的处理线程，再进行后续的动作。例如一些携带FIFO（包含一定数据量的先进先出队列）的串口外设，其写入过程可以是这样的，如下图所示：

线程先向串口的FIFO中写入数据，当FIFO满时，线程主动挂起。串口控制器持续地从FIFO中取出数据并以配置的波特率（例如115200bps）发送出去。当FIFO中所有数据都发送完成时，将向处理器触发一个中断；当中断服务例程得到执行时，可以唤醒这个线程。这里举例的是FIFO类型的设备，在现实中也有DMA类型的设备，原理类似。

对于低速设备这种模式非常好，因为在串口外设把FIFO中的数据发送出去前，处理器可以运行其他的线程，这样就提高了系统的整体运行效率（甚至对于分时系统来说，这样的设计也是非常必要）。但是对于一些高速设备，例如传输速度达到10Mbps的时候，假设一次发送的数据量是32字节，我们可以计算出发送这样一段数据量需要的时间是：

$$(32 \times 8) \times 1/10\text{Mbps} = 25\mu\text{s}$$

当数据需要持续传输时，系统将在25us后触发一个中断以唤醒上层线程继续下次传递。假设系统的任务切换时间是8us（通常实时操作系统的任务上下文切换时间只有几个us），那么当整个系统运行时，对于数据带宽利用率将只有 $25/(25 + 8) = 75.8\%$ 。但是采用轮询模式，数据带宽的利用率则可能达到100%。这个也是大家普遍认为实时系统中数据吞吐量不足的缘故，系统开销消耗在了任务切换上（有些实时系统甚至会如本章前面说的，采用底半处理，分级的中断处理方式，相当于再行拉长中断到发送线程的时间开销，效率会更进一步下降）。

通过上述的计算过程，我们可以看出其中的一些关键因素：发送数据量越小，发送速度越快，对于数据吞吐量的影响也将越大。归根结底，系统中产生中断的频度如何。当一个实时系统想要提升数据吞吐量时，可以考虑的几种方式：

- 增加每次数据量发送的长度，每次尽量让外设尽量多地发送数据；
- 必要情况下更改中断模式为轮询模式。同时为了解决轮询方式一直抢占处理机，其他低优先级线程得不到运行的情况，可以把轮询线程的优先级适当降低。

第 8 章

应用模块

在传统桌面操作系统中，用户空间和内核空间是分开的，应用程序运行在用户空间，内核以及内核模块则运行于内核空间，其中内核模块可以动态加载与删除以扩展内核功能，而在小型嵌入式设备领域，通常并不区分内核态与用户态，并且整个系统通常编译成一个单独的固件下载到单片机芯片的Flash中。自RT-Thread 0.4.0版开始引入了一种称为RT-Thread application module（应用模块）的技术，它提供了一种动态加载或卸载应用程序的功能，应用程序可以独立编译，并存储外部存储介质上，如SD卡、SPI Flash，甚至可以通过网络传输。但RT-Thread依然没有区分用户空间与内核空间，应用模块兼顾应用程序和内核模块的属性，是两者的结合体，所以称作应用模块。为书写简单，下文将RT-Thread application module简称为应用模块或模块。

[TODO] 更新使用应用模块为使用[rtthread-apps](#)的方式；加入RTM_EXPORT宏描述。

8.1 功能和限制

应用模块为RT-Thread提供一种类似桌面系统安装卸载应用程序的功能，功能十分灵活。从实现上讲，这是一种将内核和应用分开的机制，通过这种机制，内核和应用可以分开编译，并在运行时通过内核中的模块加载器将编译好的应用加载到内核中运行。

当前RT-Thread支持应用模块的架构包括ARM7、ARM9、Cortex-M3/M4/M7。当前RT-Thread内核可使用多种编译器，如GCC、ARMCC、IAR等工具链，但是模块编译只支持GCC工具链，因此编译RT-Thread模块需下载GCC工具，例如CodeSourcery的arm-none-eabi工具链。

应用模块也带来了一定的限制，它仅支持加载到RAM中运行，而不能直接在flash上运行，因此，RAM耗费的会多一些。

8.2 使用应用模块

要想在板子测试使用应用模块，需要编译一个支持应用模块的RT-Thread主程序以及独立编译的应用模块程序。下面将分为两部分介绍。

8.2.1 编译主程序

在rtconfig.h中打开如下宏（如果不存在则手动添加）

```
#define RT_USING_MODULE
```

然后重新编译主工程。读者需要参考SCons构建系统那一章学习如何编译RT-Thread源代码。将编译好的主程序下载到芯片中运行。

注意：

1. 如果是手动创建RT-Thread的MDK工程，需要在链接选项中应加入--keep __rtm-sym_*参数
2. 某些分支bsp目录下的startup.c可能未加入rt_module_system_init()函数

8.2.2 使用应用模块

RT-Thread源码中提供了几个应用模块的基本例子，它们位于RT-Thread源码树的example/module目录下，目前该目录下共有如下目录和文件

- example/module/basicapp/ 一个简单的应用模块工程
- example/module/tetris/ 一个使用RTGUI的俄罗斯方块的应用模块工程
- example/module/SConstruct 应用模块工程的构建脚本
- example/module/rtconfig.py 应用模块工程的配置脚本，需要配置工具链路径和bsp，默认为mini2440分支
- example/module/rtconfig_lm3s.py lm3s8962分支的应用工程配置脚本模板，使用时需更名为rtconfig.py
- example/module/README 使用说明

这里以stm32f10x bsp为例，编译一个stm32的应用模块程序，起名为test。假如工作目录为D:/work，复制以下文件或目录到work目录下，

- \$RTT_ROOT/example/module/basicapp/ --> work/basicapp/ --> work/test
- \$RTT_ROOT/example/module/SConstruct --> work/SConstruct

并将work目录下的basicapp重命名为test。

由于stm32f10x属于arm-cortex M3架构，因此复制rtconfig_lm32s.py到work目录下，并重命名为rtconfig.py，如下所示

- \$RTT_ROOT/example/module/rtconfig_lm3s.py --> work/rtconfig.py

打开work/rtconfig.py文件，并做如下修改

1. 修改BSP为stm32f10x
2. 首先确保ARM GCC已经安装，并修改EXEC_PATH为你的编译器路径

在笔者的机器上，修改后的work/rtconfig.py如下所示：

```
# bsp name
BSP = 'stm32f10x'

# toolchains
```



```

EXEC_PATH = r'C:\Program Files (x86)\CodeSourcery\Sourcery_CodeBench_Lite_for_ARM_EABI\bin'
PREFIX = 'arm-none-eabi-'
CC = PREFIX + 'gcc'
CXX = PREFIX + 'g++'
AS = PREFIX + 'gcc'
AR = PREFIX + 'ar'
LINK = PREFIX + 'gcc'
TARGET_EXT = '.so'
SIZE = PREFIX + 'size'
OBJDUMP = PREFIX + 'objdump'
OBJCOPY = PREFIX + 'objcopy'

DEVICE = '-mcpu=cortex-m3'
CFLAGS = DEVICE + ' -mthumb -mlong-calls -Dsourceyrgxx -O0 -fPIC'
AFLAGS = '-c' + DEVICE + ' -x assembler-with-cpp'
LFLAGS = DEVICE + ' -mthumb -Wl,-z,max-page-size=0x4 -shared -fPIC -e main -nostdlib'

CPATH = ''
LPATH = ''

```

在work目录下打开命令行，执行如下命令编译应用模块

```
scons --app=test
```

如果没有错误，会在work/test下生成build/stm32f10x/目录，其中test.so为RT-Thread应用模块。

将test.so拷贝到SD或其他设备中。然后在开发板中运行第一步编译的支持了RT-Thread应用模块的主程序。加入test.so已经被置于SD卡，并且将SD卡挂载到RT-Thread根目录下，则在finsh Shell中运行

```
finsh>>exec("/test.so")
```

可以看到如下效果，test.so正确运行。

```

Hello RT-Thread 1 101
Hello RT-Thread 2 102
Hello RT-Thread 3 103
Hello RT-Thread 4 104
Hello RT-Thread 5 105
Hello RT-Thread 6 106
Hello RT-Thread 7 107
Hello RT-Thread 8 108
Hello RT-Thread 9 109
Hello RT-Thread 10 110
Hello RT-Thread 11 111
Hello RT-Thread 12 112
Hello RT-Thread 13 113
.....

```

读者也可以尝试example/module目录下的其他应用模块示例，也可以自己编写应用模块程序。尽情享受应用模块带来的灵活吧~，限制你的只有想象力！

8.3 应用模块API

除了可以通过finsh手动加载应用模块外，也可以在主程序中使用RT-Thread提供的应用模块API来加载或卸载应用模块。

```
rt_module_t rt_module_open(const char *path)
```

函数参数

参数	描述
path	模块完整路径名；

函数返回

正确加载返回模块指针，否则返回NULL

这个函数从文件系统中加载应用模块到内存中运行，若正确加载返回该模块的指针。

```
rt_module_t rt_module_find(const char *name)
```

函数参数

参数	描述
name	模块名；

函数返回

如果找到则返回模块指针，否则返回NULL

这个函数根据模块名查找系统已加载的模块，若找到返回该模块的指针。

```
rt_err_t rt_module_destroy(rt_module_t module)
```

函数参数

参数	描述
module	模块指针；

函数返回

成功返回RT_EOK；失败返回-RT_ERROR。
这个函数会销毁应用模块占用的RT-Thread内核对象，如信号量、互斥量、mempool等，如果它使用了的话。

```
rt_err_t rt_module_unload(rt_module_t module)
```

函数参数

参数	描述
module	模块指针；

函数返回

成功返回RT_EOK；失败返回-RT_ERROR。
这个函数会销毁应用模块的线程以及子线程。

第 9 章

移植

想要让RT-Thread在某款芯片上运行起来，必须先要把RT-Thread在这款芯片上移植好。本章讲述RT-Thread移植相关事项，同时以ARM Cortex-M3为例给出如何移植RT-Thread到一个新的芯片上。

9.1 使用移植还是自己移植

通常大多数人还没分清楚是使用一个移植还是自己进行移植，以为要把RT-Thread跑在自己的板子上都需要进行移植。例如RT-Thread官方已经提供了STM32F103ZET6的移植，而自己的目标板使用的是STM32F103C8T6的芯片，所以觉得自己应该把RT-Thread“移植”到自己的板子上。

孰不知，STM32F103ZET6和STM32F103C8T6对操作系统的内核差别仅在于SRAM的容量大小。如果使用了系统动态内存堆（即上层应用需要使用rt_malloc函数），仅仅是这个系统动态内存堆的结束地址不一样。其他部分，例如ARM Cortex-M3部分是完全一样的。还有不一样，需要自己仔细检查的地方包括：

- 主晶振使用的震荡频率是多少；
- 外设的pin引脚是否相同；

明白这些之后就知道，要想把已经支持STM32F103ZET6的RT-Thread，在STM32F103C8T6上运行起来，只需要修改这些相关的地方即可。和传统意义的移植相差甚远，可以重用STM32F103ZET6的大部分移植代码。

所以在想移植RT-Thread到一款新型号的芯片之前，应该了解下，RT-Thread是否已经支持类似的处理器，如果已经支持了，那么仅需要把不同的地方修改即可。

9.2 移植前的准备

在移植之前，应该对RT-Thread的目录结构有一定的了解：

```
RT-Thread
+---bsp /* 板级支持包,所有支持的芯片可以在这里面找到 */
|   \--- stm32f40x
|   \--- simulator
|   \--- stm32f0x
```

```
| \--- stm32f107
+---components /* RT-Thread支持的组件文件夹 */
+---include
+---libcpu /* CPU内核的代码 */
| \---arm
|     \---cortex-m4
|     \---cortex-m3
+---src /* RT-Thread内核代码 */
+---tools /* 工具脚本文件夹 */
```

其中，include和src目录用于放置RT-Thread的实时核心代码文件；components目录用于放置各类组件；tools是用于放置RT-Thread的构建环境scons的一些扩展脚本；bsp和libcpu则是移植相关的部分。

注

通常来说，对于一个移植除了bsp、libcpu目录以外，其他的目录和文件不应该被修改，而且对于一种已知完成移植的内核(比如Cortex-M3,Cortex-M4等),其libcpu部分也已经存在,完全没有重写的必要,只要完成相关bsp移植部分即可。

只有在你需要支持一种新的编译器时，才可能修改到include\rtdef.h和finsh等相关的代码。当要支持一种新的编译器，同时希望包括在开发分支内时，请联系内核的维护人以解决相关的问题，或给与适当的指导。

在了解了RT-Thread的目录，以及知道自己应该修改哪里的代码后，应该了解RT-Thread移植的两种模式：

- 使用RT-Thread中的libcpu目录：这个时候，和CPU相关的移植放在libcpu目录下的相对应的子目录中，自己的移植通过scons的SConscript脚本或工程文件使用这个目录下的libcpu文件；
- 不使用RT-Thread中的libcpu目录：例如希望使用自己的CPU移植，或这份CPU移植不会放到开发分支上。

对于第二种情况，可以按照如下的方式组织自己的移植：

```
your_board
+---applications
+---components
+---cpu
+---drivers
+---documents
+---Libraries
\---rt-thread
    +---tools
    +---include
    +---components
    \---src
```

从这个目录结构可以看到，rt-thread的相关目录被做为一个相对独立的目录放在工程目录下面。同时在自己的工程目录中，包含：

- applications，用于放置用户应用的目录；

- components, 用于放置用户自己的组件;
- cpu, 替代原来的libcpu目录, 放置芯片移植相关的代码、驱动;
- driver, 用户自行编写的驱动;
- documents, 用户文档;
- Libraries, 一些相对固定的库文件;

需要注意的一点是, 按照这样的使用方式, 需要在SConstruct文件中加入has_libcpu = True的选项:

```
# prepare building environment
objs = PrepareBuilding(env, RTT_ROOT, has_libcpu=True)
```

这样后续不管是使用scons进行编译或者使用scons生成工程文件去编译, 都将不会使用在rt-thread\libcpu中芯片相关的这部分代码。

9.3 RT-Thread在ARM Cortex M3上的移植

下面将以RT-Thread在STM32F103ZE为例, 介绍纯手工, 利用现有框架在Keil MDK上移植RT-Thread的过程, 当然, 如果觉得许多配置等麻烦, 可以参考 RT-Thread在ARM Cortex M4上的移植 里面介绍了基于模板的自动配置工程方法。

9.3.1 建立RealView MDK工程

在bsp目录下新建project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bsp\your_board目录下。创建工程时一次的选项如 MDK CPU 选择示意图: CPU选择STMicroelectronics的STM32F103ZE:

提问复制STM32的启动代码到工程目录, 选择No, 我们需要使用库中的启动文件

然后选择工程的属性, 如 MDK工程Target设置图

Select Folder for Objects目录选择到bsp\your_board\build, Name of Executable为rtthread-stm32

同样Select Folder for Listings选择bsp\your_board\objs目录, 如 MDK工程Listing设置图 所示:

如 MDK工程编译设置图 所示:

C/C++编译选项标签页中, 因为在项目中使用了ST的STM32固件库, 需要在Define中添加如下2个定义

- STM32F10X_HD
- USE_STDPERIPH_DRIVER

在Include Paths (头文件搜索路径) 中加上ST固件库中需要Include的目录:

- Libraries\STM32F10x_StdPeriph_Driver\inc;
- Libraries\CMSIS\CM3\CoreSupport;
- Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x

以及RT-Thread的头文件路径和STM32移植目录的路径:

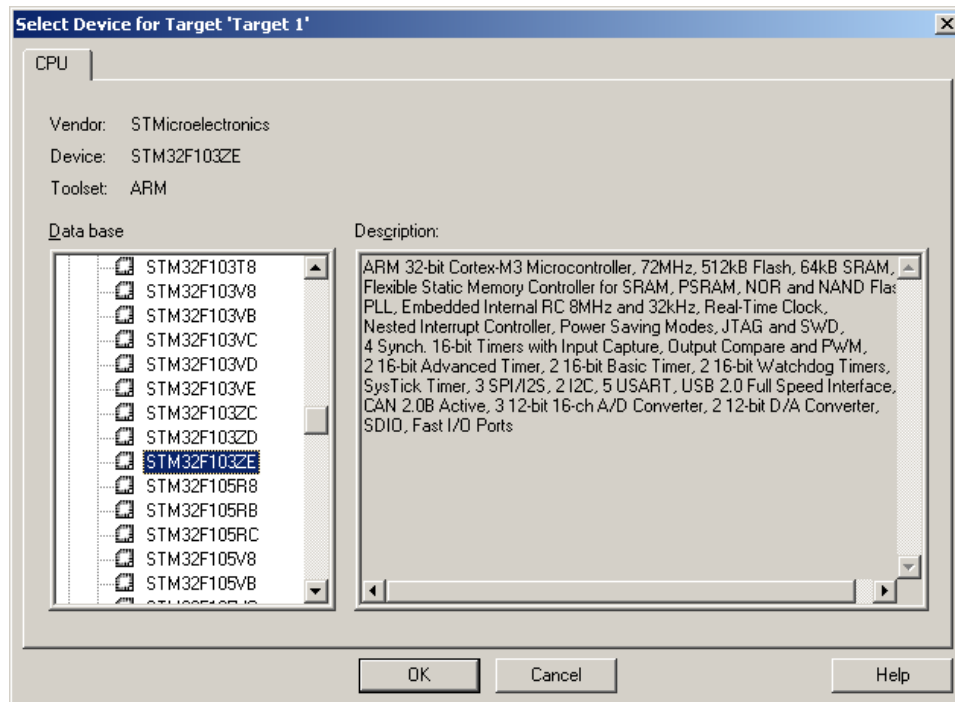


图 9.1: MDK CPU 选择示意图

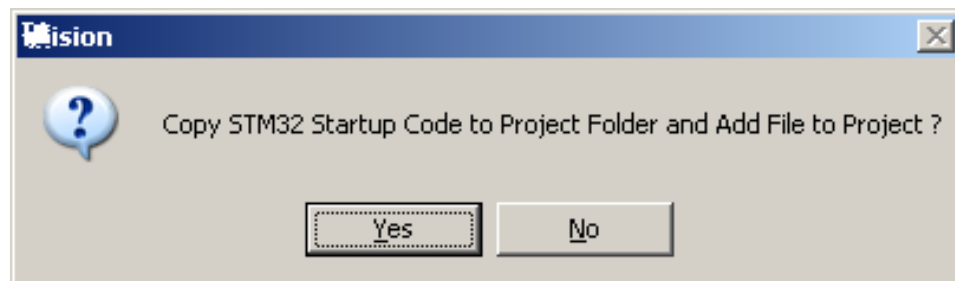


图 9.2: MDK增加启动代码

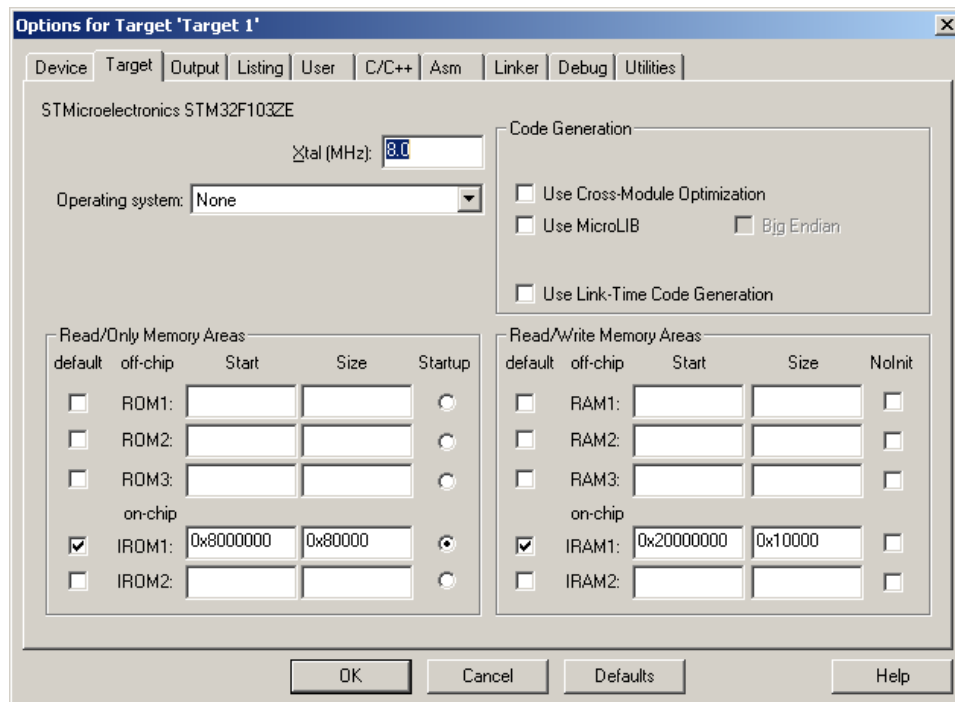


图 9.3: MDK工程Target设置图

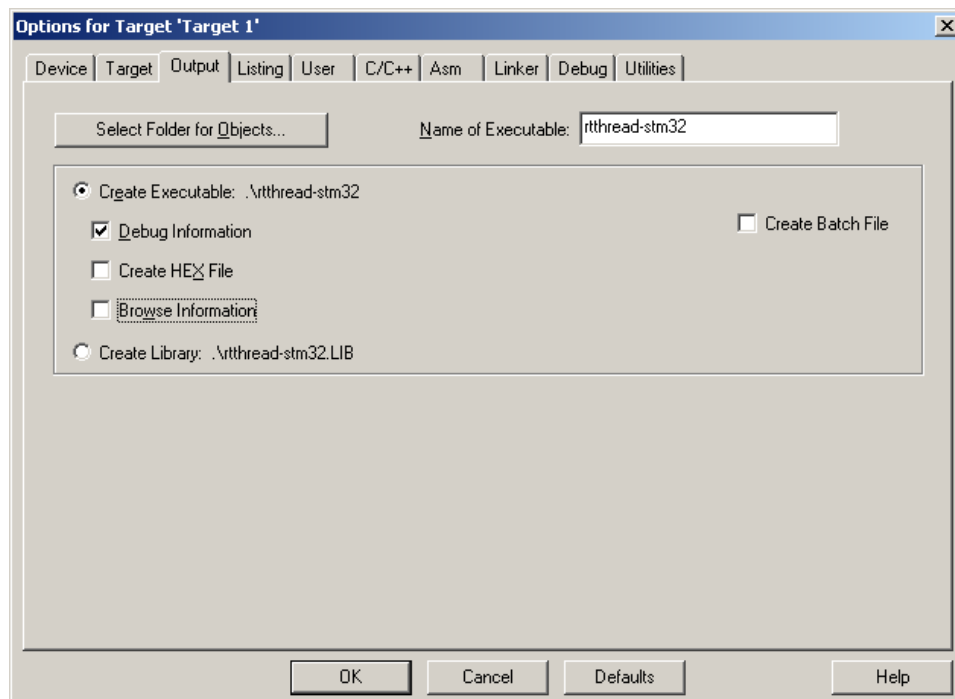


图 9.4: MDK工程OutPut设置图

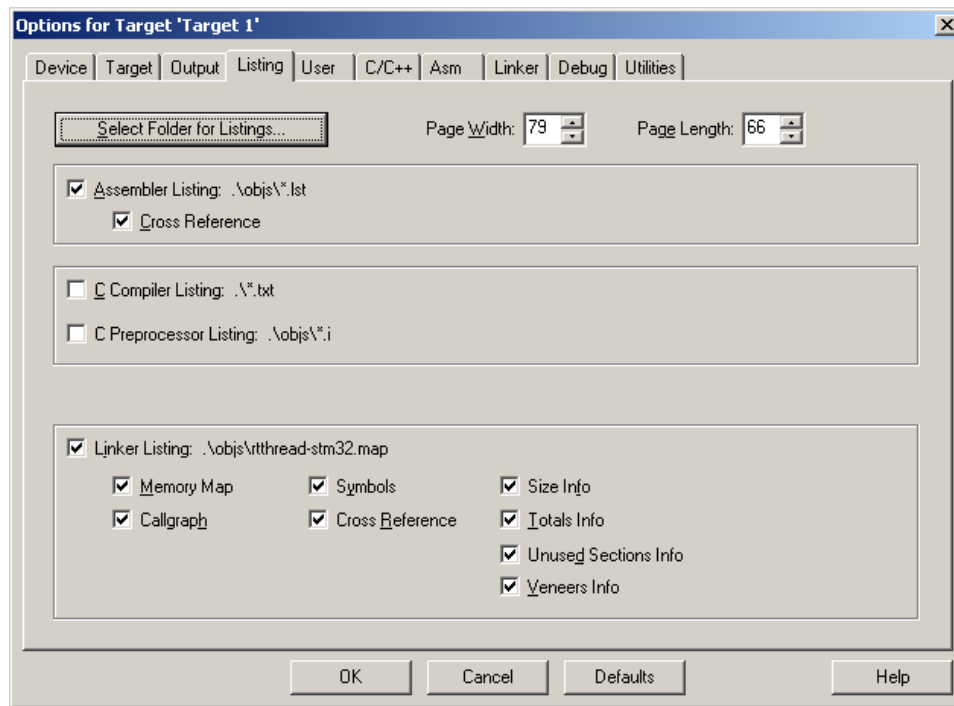


图 9.5: MDK工程Listing设置图

- ..\..\include;
- ..\..\libcpu\arm\cortex-m3,

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

9.3.2 添加源文件

- 添加STM32固件库源文件,新添加Group: STM32_StdPeriph, 然后把以下2个目录中的所有C源文件都添加到Group中

1.bsp\your_board\Libraries\CMSIS 2.bsp\your_board\Libraries\STM32F10x_StdPeriph_Driver
 \src 3.bsp\your_board\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm
 \startup_stm32f10x_hd.s

- 添加RT-Thread相关源文件

对工程中初始添加的Source Group1改名为Applications, 并添加Kernel, Cortex-M3, Drivers的Group

在Cortex-M3分组中,加入libcpu\arm\cortex-m3里的context_rvds.S和cpuport.c文件,以及libcpu\arm\common中的backtrace.c, div0.c, showmem.c

Kernel Group中添加所有\src下的C源文件;

Applications Group中添加startup.c文件 (放于bsp\your_board\applications目录中);

Drivers Group中添加board.c文件 (放于bsp\your_board\drivers目录中);

- 添加RT-Thread配置头文件

在bsp\your_board目录中添加rtconfig.h文件, 内容如下:

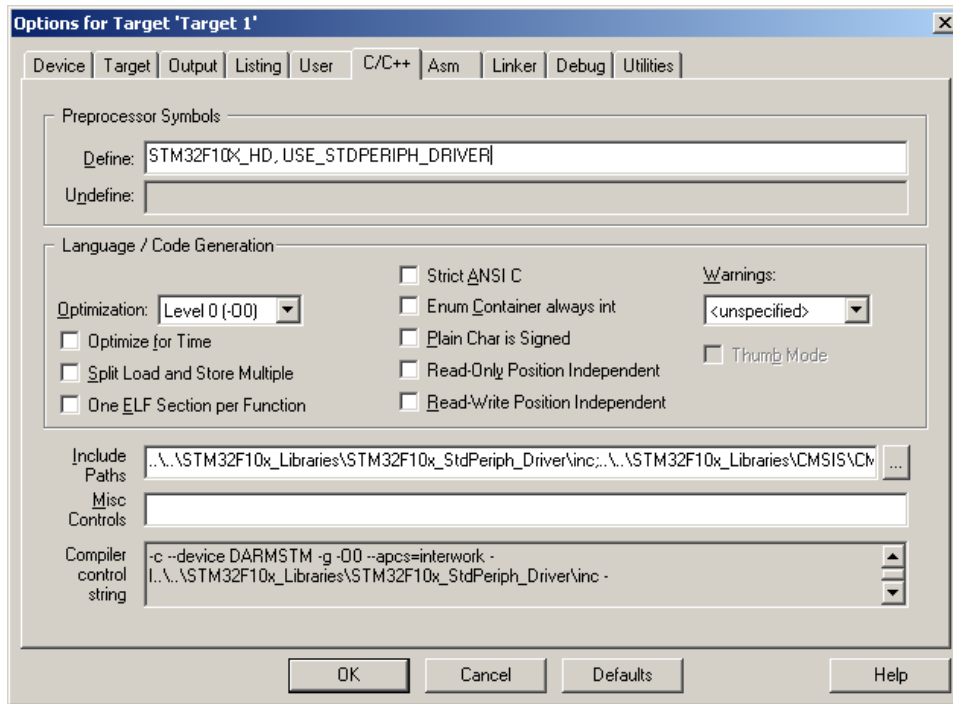


图 9.6: MDK工程编译设置图

```

/* RT-Thread配置文件 */
#ifndef __RTTHREAD_CFG_H__
#define __RTTHREAD_CFG_H__

/* 内核对象名称最大长度 */
#define RT_NAME_MAX          8
/* 数据对齐长度 */
#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 */
/* 打开RT-Thread的ASSERT选项 */
#define RT_DEBUG
/* 打开RT-Thread的线程栈溢出检查 */
#define RT_USING_OVERFLOW_CHECK
/* 使用钩子函数 */
#define RT_USING_HOOK

/* SECTION: 线程间通信选项 */
/* 支持信号量 */
#define RT_USING_SEMAPHORE
/* 支持互斥锁 */
#define RT_USING_MUTEX

```

```

/* 支持事件标志 */
#define RT_USING_EVENT
/* 支持邮箱 */
#define RT_USING_MAILBOX
/* 支持消息队列 */
#define RT_USING_MESSAGEQUEUE

/* SECTION: 内存管理 */
/* 支持静态内存池 */
#define RT_USING_MEMPOOL
/* 支持动态内存堆管理 */
#define RT_USING_HEAP
/* 使用小型内存管理算法 */
#define RT_USING_SMALL_MEM

/* SECTION: 设备模块选项 */
/* 支持设备模块 */
#define RT_USING_DEVICE
/* 支持串口1设备 */
#define RT_USING_UART1
/* SECTION: 控制台选项 */
#define RT_USING_CONSOLE
/* 控制台缓冲区大小 */
#define RT_CONSOLEBUF_SIZE 128

#endif

```

- 启动代码

在Keil MDK自动生成的启动代码中, 由于STM32的中断处理方式是以中断向量表的方式进行, 所以将不再使用中断统一入口的方式进行, 启动代码可以大部分使用这份启动代码。主要修改在:

对于大多数已知的CPU, 尤其是内核相同的CPU, 他们的启动代码非常相似, 可以直接使用标准启动代码。比如说STM32F103ZET6是Cortex-M3内核的, 那就可以直接使用已有的启动代码。好处是, 这些启动代码都是官方给出的, 而且, 对于同一种内核来说, 基本上都是相同的。比如说异常的入口地址, 一般的异常Handler名称等。RT-Thread默认这些Handler都是固定(一般来说都是这样)在内核中, 已经有相关的Handler处理函数, 可以被异常直接调用, 省去了修改的麻烦。

一般来说, 在移植过程中需要用确认几个异常入口以及变量是否正确:

1. 栈尺寸

如果中断服务例程使用的栈尺寸需要不高, 可以使用默认值。Stack_Size EQU 0x00000200

2. PendSV异常

PendSV_Handler在context_rvds.S中实现, 完成上下文切换。

3. HardFault_Handler异常

HardFault异常直接保留代码也没关系，只是当系统出现了fault异常时，并不容易看到。为完善代码起见，在context_rvds.S中有相关Fault信息输出代码,入口名称为Hard-Fault_Handler异常。

4. 时钟中断

OS时钟在Cortex-M3中使用了统一的中断方式：SysTick_Handler。需要在bsp的drivers的board.c中调用rt_tick_increase();

相应的启动代码如下：

```
; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors      DCD      __initial_sp          ; Top of Stack
                DCD      Reset_Handler        ; Reset Handler
                DCD      NMI_Handler          ; NMI Handler
                DCD      HardFault_Handler    ; Hard Fault Handler
                DCD      MemManage_Handler    ; MPU Fault Handler
                DCD      BusFault_Handler     ; Bus Fault Handler
                DCD      UsageFault_Handler   ; Usage Fault Handler
                DCD      0                    ; Reserved
                DCD      0                    ; Reserved
                DCD      0                    ; Reserved
                DCD      0                    ; Reserved
                DCD      SVC_Handler          ; SVCcall Handler
                DCD      DebugMon_Handler     ; Debug Monitor Handler
                DCD      0                    ; Reserved
                DCD      PendSV_Handler       ; PendSV Handler
                DCD      SysTick_Handler      ; SysTick Handler
```

• 栈初始化代码

栈的初始化代码用于创建线程或初始化线程，“手工”的构造一份线程运行栈，相当于在线程栈上保留了一份线程从初始位置运行的上下文信息。在Cortex-M3体系结构中，当系统进入异常时，CPU Core会自动进行R0 – R3以及R12、psr、pc、lr等压栈，所以手工构造这个初始化栈时，也相应的把这些寄存器初始值放置到正确的位置。

libcpu\arm\cortex-m3\cpuport.c程序清单：

```
#include <rtthread.h>

/*
 * 这个函数用于初始化线程栈
 *
 * @param tentry 线程的入口函数
 * @param parameter 线程入口函数的参数
```

```

* @param stack_addr 栈的初始化地址
* @param texit 当线程退出时的处理函数
*
* @return 返回准备好的栈初始指针
*/
rt_uint8_t *rt_hw_stack_init(void *tentry, void *parameter,
                             rt_uint8_t *stack_addr, void *texit)
{
    struct stack_frame *stack_frame;
    rt_uint8_t *stk;
    unsigned long i;

    stk = stack_addr + sizeof(rt_uint32_t);
    stk = (rt_uint8_t *)RT_ALIGN_DOWN((rt_uint32_t)stk, 8);
    stk -= sizeof(struct stack_frame);

    stack_frame = (struct stack_frame *)stk;

    /* init all register */
    for (i = 0; i < sizeof(struct stack_frame) / sizeof(rt_uint32_t); i++)
    {
        ((rt_uint32_t *)stack_frame)[i] = 0xdeadbeef;
    }

    stack_frame->exception_stack_frame.r0 = (unsigned long)parameter; /* r0 : argument */
    stack_frame->exception_stack_frame.r1 = 0;                          /* r1 */
    stack_frame->exception_stack_frame.r2 = 0;                          /* r2 */
    stack_frame->exception_stack_frame.r3 = 0;                          /* r3 */
    stack_frame->exception_stack_frame.r12 = 0;                         /* r12 */
    stack_frame->exception_stack_frame.lr = (unsigned long)texit;      /* lr */
    stack_frame->exception_stack_frame.pc = (unsigned long)tentry;     /* entry point, pc */
    stack_frame->exception_stack_frame.psr = 0x01000000L;              /* PSR */

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

/* @} */

```

最终形成的线程栈情况如 堆栈压入情况图：

- 上下文切换代码

代码清单libcpu\arm\cortex-m3\context_rvds.S：在RT-Thread中，中断锁是完全由芯片移植来实现的，参见 线程间同步与通信章节。以下是Cortex-M3上的开关中断实现，它是使用CPSID指令来实现的。

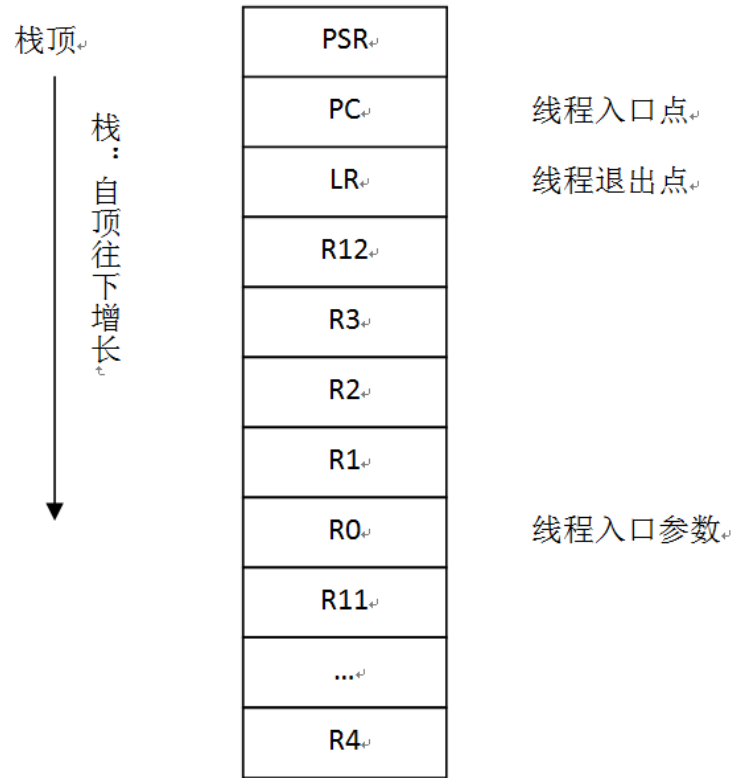


图 9.7: 堆栈压入情况图

```
; 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
```

在Cortex M3微处理器中，当系统进入异常时，CPU Core会自动进行R0 – R3以及R12、PSR、PC、LR等压栈，所以CM3的线程上下文切换正可以利用硬件压栈的特点，让机器自动帮助完成部分工作：在线程正常的上下文切换时，触发一个PenSV中断，从而进入到PenSV处理程序中。

```
; void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);
```

```

; r0 --> from
; r1 --> to
; 在Cortex M3移植中, 这两个函数的内容都是相同的,
; 因为正常模式的切换也采取了触发PendSV异常的方式进行
rt_hw_context_switch_interrupt
    EXPORT rt_hw_context_switch_interrupt
rt_hw_context_switch    PROC
    EXPORT rt_hw_context_switch

    ; 设置参数rt_thread_switch_interrupt_flag为1,
    ; 代表将要发起线程上下文切换
    LDR    r2, =rt_thread_switch_interrupt_flag
    LDR    r3, [r2]
    CMP    r3, #1                                ; 参数已经置1, 说明线程切换已经触发
    BEQ    _reswitch
    MOV    r3, #1
    STR    r3, [r2]

    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)

_reswitch
    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
    BX     LR
    ENDP

; PendSV异常处理
; r0 --> swith from thread stack
; r1 --> swith to thread stack
; psr, pc, lr, r12, r3, r2, r1, r0 等寄存器已经被自动压栈
; 到切换出线程栈中
rt_hw_pend_sv    PROC
    EXPORT rt_hw_pend_sv

    ; 为了保护线程切换, 先关闭中断
    MRS    r2, PRIMASK
    CPSID  I

    ; 获得rt_thread_switch_interrupt_flag参数,
    ; 以判断pendsv是否已经处理过

```



```

    LDR    r0, =rt_thread_switch_interrupt_flag
    LDR    r1, [r0]
    CBZ    r1, pendsv_exit          ; pendsv已经被处理, 直接退出

    ; 清除参数: rt_thread_switch_interrupt_flag为0
    MOV    r1, #0x00
    STR    r1, [r0]

    LDR    r0, =rt_interrupt_from_thread
    LDR    r1, [r0]
    CBZ    r1, swtich_to_thread     ; 如果切换出线程为0, 这是第一次上下文切换

    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寄存器

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

pendsv_exit
    ; 恢复中断
    MSR    PRIMASK, r2

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

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

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

LDR    r0, =NVIC_INT_CTRL
LDR    r1, =NVIC_PENDSVSET
STR    r1, [r0]             ; 触发PendSV异常

; restore MSP
LDR    r0, =SCB_VTOR
LDR    r0, [r0]
LDR    r0, [r0]
MSR    msp, r0

CPSIE  I                    ; 使能中断以使PendSV能够正常处理
ENDP
```

正常模式下的线程上下文切换的过程可以用下 正常模式上下文切换图 示：

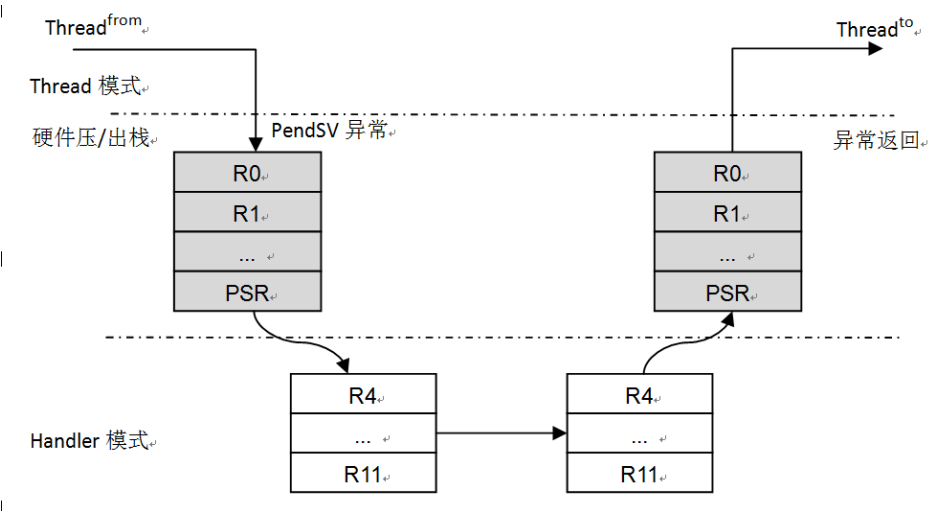


图 9.8: 正常模式上下文切换图

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自动压入当前线程的栈中，然后切换到PenSV异常处理。到PenSV异常后，Cortex M3工作模式

(从Thread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(Thread from)和切换到线程(Thread to)的栈指针, 如果切换出线程的栈指针是0那么表示这是系统启动时的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 中断模式上下文切换图 表示:

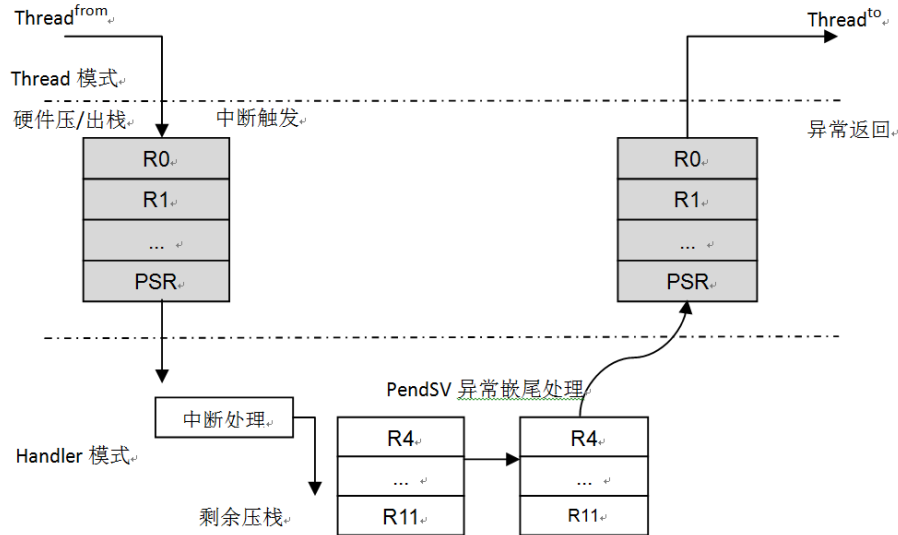


图 9.9: 中断模式上下文切换图

当中断达到时, 当前线程会被中断并把PC, PSR, R0 - R3, R12等压到当前线程栈中, 工作模式(从Thread模式)切换到Handler模式。

在运行中断服务例程期间, 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), 如果是则调用rt_hw_contextswitch_interrupt函数进行伪切换: 在rt_hw_contextswitch_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_interrupt_flag为1。在最后一个中断服务例程结束时, Cortex M3将去处理PendSV异常, 因为PendSV异常的优先级是最低的, 所以只要触发过PendSV异常, 它将总是在最后得到处理。Fault处理代码 Fault处理代码并不是必须的, 为了系统的完整性, 实现fault处理代码无疑对系统出错时定位问题提供非常有利的帮助。libcpu\arm\cortex-m3\context_rvds.S程序清单:

```

IMPORT rt_hw_hard_fault_exception
EXPORT HardFault_Handler
HardFault_Handler    PROC

; get current context
TST    lr, #0x04                ; if(!EXC_RETURN[2])
MRSNE  r0, msp                  ; get fault context from handler.
MRSEQ  r0, psp                  ; get fault context from thread.

STMFD  r0!, {r4 - r11}          ; push r4 - r11 register

```

```

STMFD    r0!, {lr}                ; push exec_return register

MSRNE    msp, r0                  ; update stack pointer to MSP.
MSREQ    psp, r0                  ; update stack pointer to PSP.

PUSH     {lr}
BL       rt_hw_hard_fault_exception
POP      {lr}

ORR      lr, lr, #0x04             ; 从fault中返回
BX       lr
ENDP

END

```

libcpu\arm\cortex-m3\cpuport.c程序清单：

```

#include <rtthread.h>

/* CM3硬件压栈时的寄存器结构 */
struct exception_stack_frame
{
    rt_uint32_t r0;
    rt_uint32_t r1;
    rt_uint32_t r2;
    rt_uint32_t r3;
    rt_uint32_t r12;
    rt_uint32_t lr;
    rt_uint32_t pc;
    rt_uint32_t psr;
};

void rt_hw_hard_fault_exception(struct exception_info * exception_info)
{
    extern long list_thread(void);
    struct stack_frame* context = &exception_info->stack_frame;

    if (rt_exception_hook != RT_NULL)
    {
        rt_err_t result;

        result = rt_exception_hook(exception_info);
        if (result == RT_EOK)
            return;
    }

    rt_kprintf("psr: 0x%08x\n", context->exception_stack_frame.psr);
}

```

```

    rt_kprintf("r00: 0x%08x\n", context->exception_stack_frame.r0);
    rt_kprintf("r01: 0x%08x\n", context->exception_stack_frame.r1);
    rt_kprintf("r02: 0x%08x\n", context->exception_stack_frame.r2);
    rt_kprintf("r03: 0x%08x\n", context->exception_stack_frame.r3);
    rt_kprintf("r04: 0x%08x\n", context->r4);
    rt_kprintf("r05: 0x%08x\n", context->r5);
    rt_kprintf("r06: 0x%08x\n", context->r6);
    rt_kprintf("r07: 0x%08x\n", context->r7);
    rt_kprintf("r08: 0x%08x\n", context->r8);
    rt_kprintf("r09: 0x%08x\n", context->r9);
    rt_kprintf("r10: 0x%08x\n", context->r10);
    rt_kprintf("r11: 0x%08x\n", context->r11);
    rt_kprintf("r12: 0x%08x\n", context->exception_stack_frame.r12);
    rt_kprintf(" lr: 0x%08x\n", context->exception_stack_frame.lr);
    rt_kprintf(" pc: 0x%08x\n", context->exception_stack_frame.pc);

    if(exception_info->exc_return & (1 << 2) )
    {
        rt_kprintf("hard fault on thread: %s\r\n\r\n", rt_thread_self()->name);

#ifdef RT_USING_FINSH
        list_thread();
#endif /* RT_USING_FINSH */
    }
    else
    {
        rt_kprintf("hard fault on handler\r\n\r\n");
    }

#ifdef RT_USING_FINSH
    hard_fault_track();
#endif /* RT_USING_FINSH */

    while (1);
}

```

9.4 RT-Thread/STM32其他部分说明

RT-Thread/STM32移植是基于RealView MDK开发环境进行移植的（GNU GCC编译器和IAR ARM编译亦支持），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，建议可以考虑使用成熟的库（例如针对STM32芯片，可以采用ST官方的固件库）。RT-Thread/STM32工程中已经包含了STM32f10x系列3.1.x的库代码，可以配套使用。和中断相关的rt_hw函数（异常与中断章节

中的大多数函数) 本移植中并不具备, 所以可以跳过OS层直接操作硬件。在编写中断服务例程时, 推荐使用如下的模板:

```
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_interrupt_enter()函数会通知OS进入到中断处理模式(相应的线程切换行为会有些变化); rt_interrupt_leave()函数会通知OS离开了中断处理模式。

9.5 RT-Thread在ARM Cortex M4上的移植

RT-thread支持scons工具,可以简单的创建工程文件,编译,下面将以移植MK60开发板为目标讲解。

scons创建MDK工程是以给定的Keil工程的模板为基础,通过scons把一些配置信息,文件,包含目录等加入到最后生成的工程中。

必要的,首先需要创建一个模板工程,就像创建普通MDK工程那样,但是一定要命名为 template.uvproj,作为基础模板

9.5.1 生成MDK工程模板

同移植Cortex-M3的一些基本步骤,同样创建MDK工程,CPU选择 freescale Semiconductor 中的 MK60FN1M0xxx12,询问是否添加启动代码 startup_MK60F12.s 时选择No,因为所有文件的加入,头文件的Include这些都是根据scons来加入的,不用MDK模板操心,所以,启动文件也应该由我们自己掌控。

在 Output 选项卡中,选择工程的属性,Select Folder for Objects目录选择到 bsp\your_board\build, Name of Executable 为rtthread-mk60f120m

在 Output 选项卡中,可以勾选 Browse Information 这样可以支持右键函数,宏定义跳转。

在 Listing 选项卡中,同样 Select Folder for Listings 选择 bsp\your_board\build 目录。

在 Debug 和 Utilities 选项卡中选择K60相应的 pemicro_0SJtag Asm, Linker,选项使用初始配置即可。

删除project目录下所有文件与文件夹,保证所有的文件结构增加由scons管理,保存此模板,确认名称为template.uvproj

9.5.2 仿照并修改scons相关文件

有了MDK的模板,那生成MDK工程文件的基石就有了,接下来需要scons的相关脚本文件来完成工程的生成。

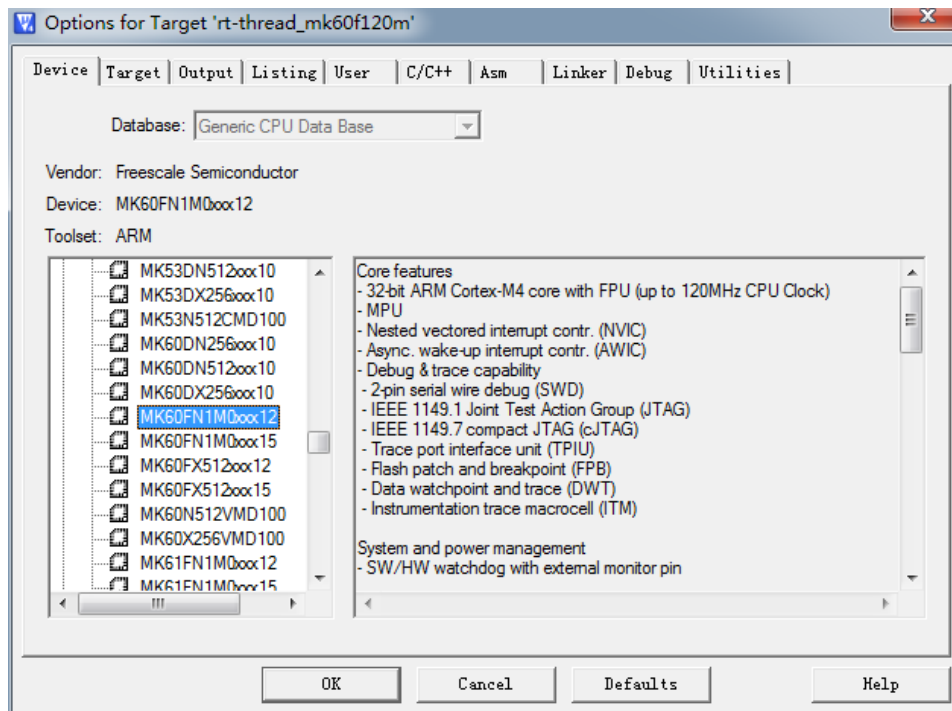


图 9.10: K60 CPU 选择示意图

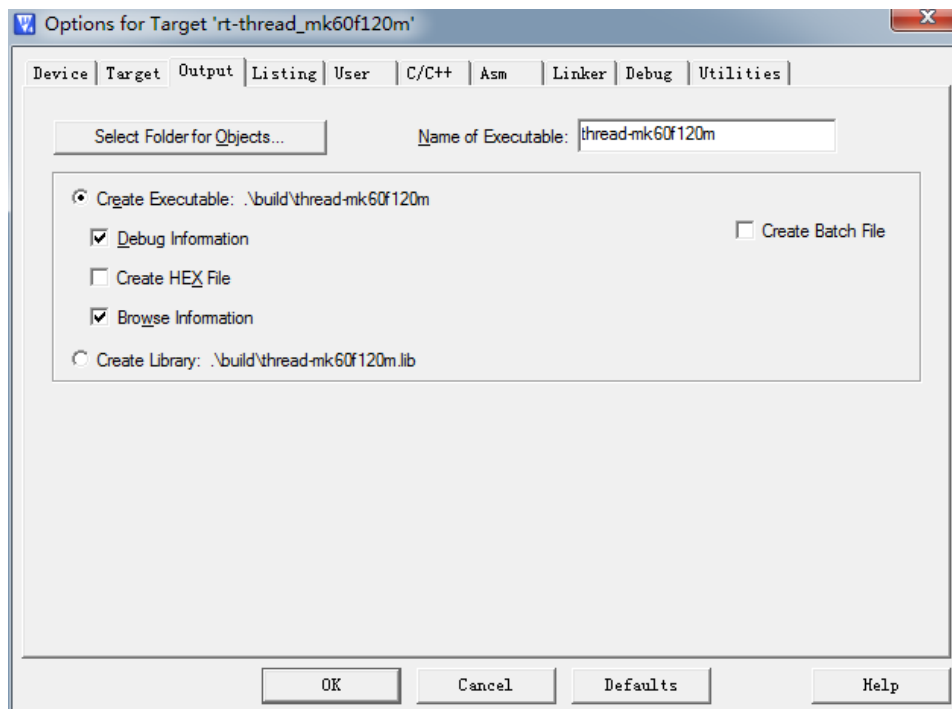


图 9.11: K60 Output 选项卡示意图

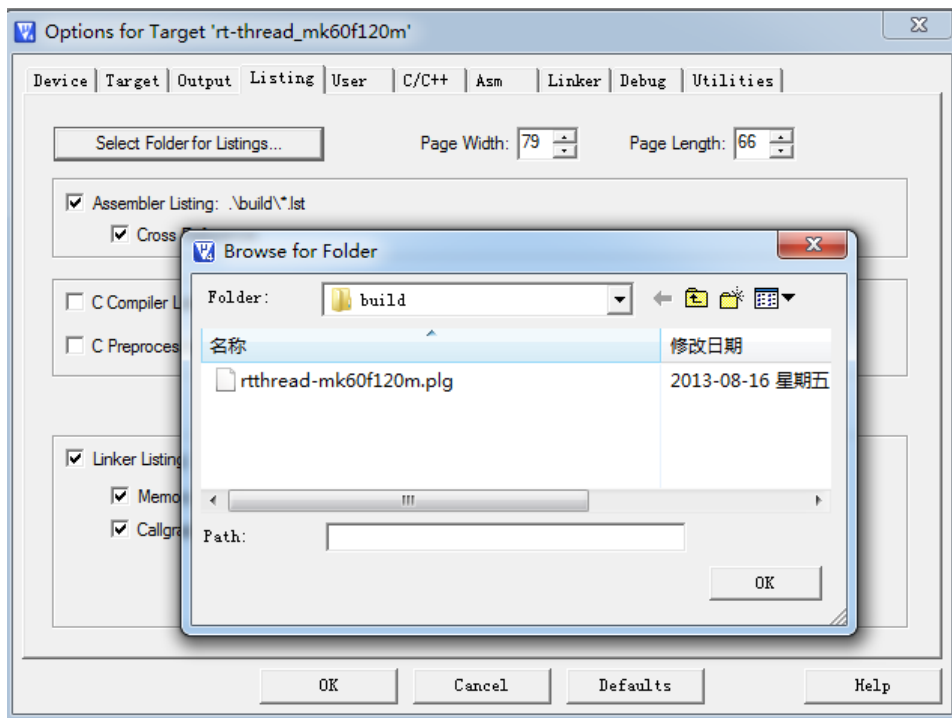


图 9.12: K60 Listing 选项卡示意图

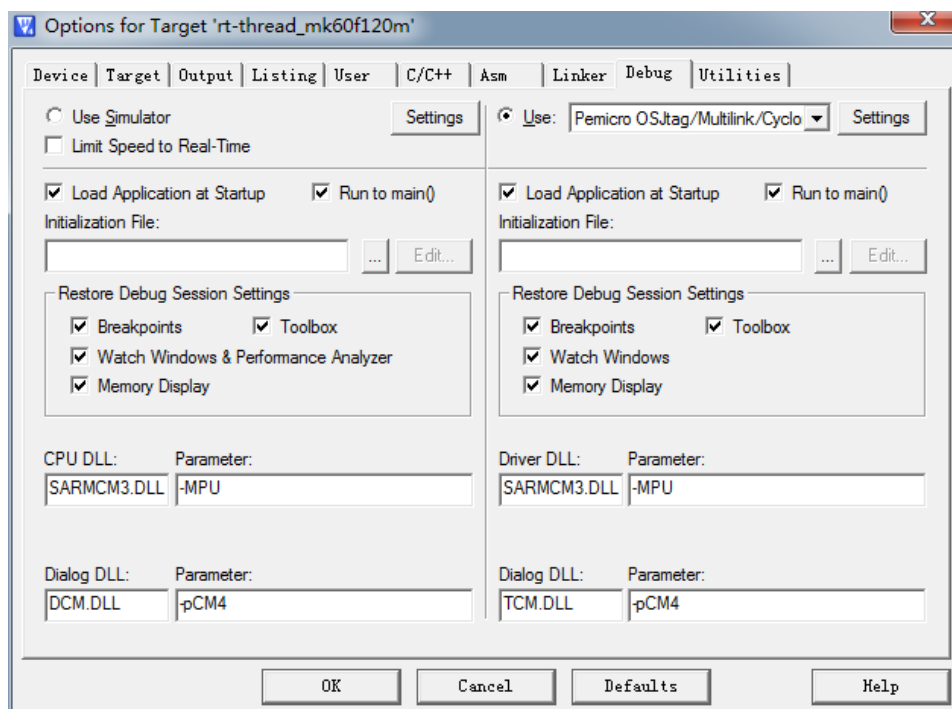


图 9.13: K60 Debug 选项卡示意图

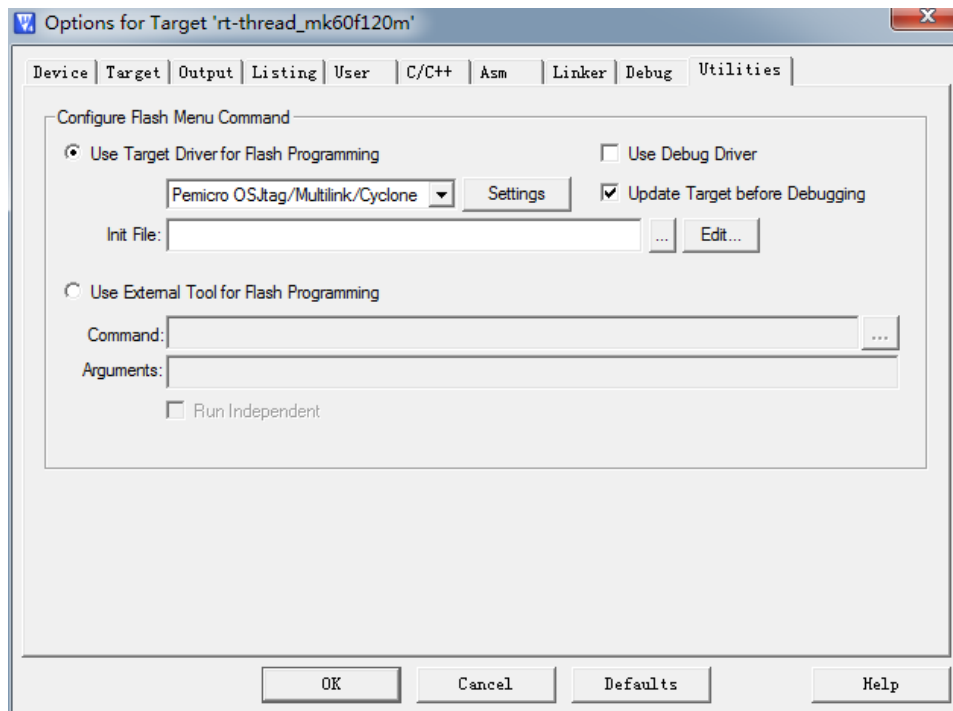


图 9.14: K60 Utilities 选项卡示意图

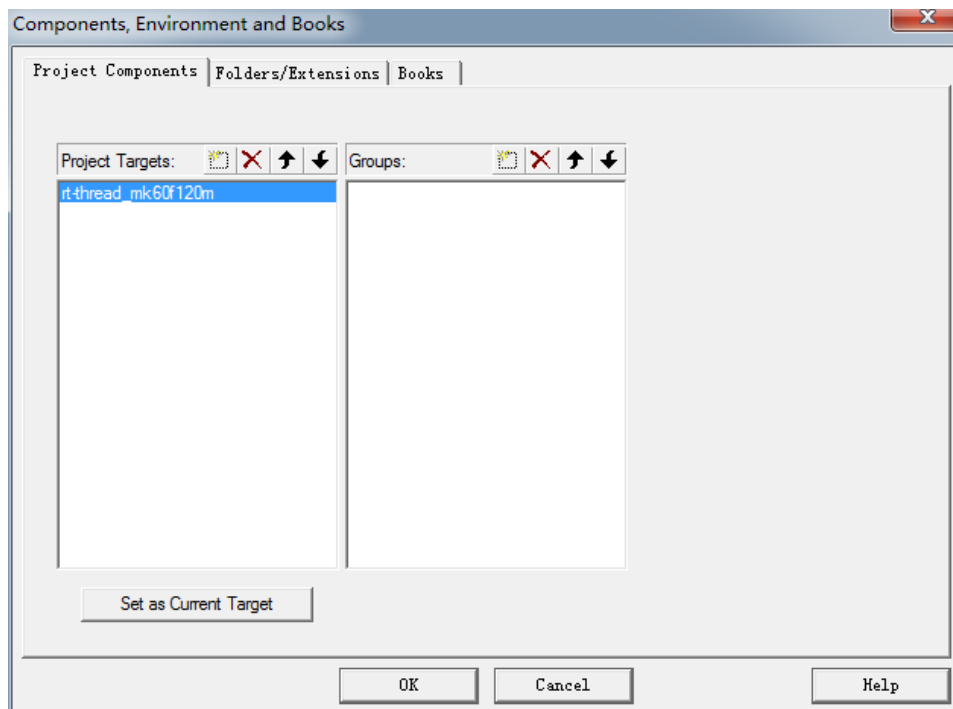


图 9.15: K60项目文件示意图

由于是Cortex-M4内核,所以可以直接从相同内核的bsp中拷贝以下这几个文件来使用的scons能够生成MDK工程,比如,从 bsp\stm32f40x 中拷贝:

- rtconfig.py
- SConscript
- SConstruct

相关文件的具体意义和作用可以参考[这篇文章](#)

直接修改 rtconfig.py 中相关段落,本次移植主要关注MDK上的移植,可以忽略gcc和iar部分,与MDK相关的重点如下:

- 确认如下字段

```
# toolchains options
ARCH='arm'
CPU='cortex-m4'
CROSS_TOOL='keil'          /* 使用keil */
```

- 设置Keil路径

```
elif CROSS_TOOL == 'keil':
    PLATFORM      = 'armcc'
    EXEC_PATH     = 'C:/Keil'
```

- 修改map文件名

```
LFLAGS = DEVICE + ' --info sizes --info totals --info unused --info veneers
              --list rtthread-k60.map --scatter k60_rom.sct'
```

其中,CPU类型告诉RT-Thread会去采用哪种CPU类型去加入libcpu中的文件, CROSS_TOOL则指定了编译工具链

9.5.3 添加其他相关文件

添加rtconfig.h到bsp目录下, scons会根据rtconfig.h中的宏定义来加载相关 .h 和 .c 文件,具体可以参考 RT-Thread在ARM Cortex M3上的移植 中的 添加RT-Thread配置头文件 部分的注释,只是使用的话可以直接参考类似架构的CPU,进行小部分修改即可,比如,与MK60同是Cortex-M4内核的STM32F4等bsp的文件,然后根据自己的需要进行更改。

在工程目录下增加启动文件(K60的启动文件 startup_MK60F12.s 可以在 Keil\ARM\Startup\Freescale\Kinetis 中找到)

在工程文件夹下新建drivers文件夹,用于存放bsp的相关驱动,并且在drivers目录下新建SConscript脚本,使得SCONS能够自动加载drivers中的文件到MDK工程中,同样可以参考stm32f40x/drivers中的SConscript:

```
Import('RTT_ROOT')
Import('rtconfig')
from building import *

cwd = os.path.join(str(Dir('#')), 'drivers')
src = Glob('*.c') # 将.c文件加入工程
```

```
src += Glob('*.*')          # 将.s文件加入工程,启动脚本也放在了drivers中
CPPPATH = [cwd]

# 在MDK工程中加入Drivers文件夹
group = DefineGroup('Drivers', src, depend = [''], CPPPATH = CPPPATH)

Return('group')
```

在drivers目录下,新建 board.h 和 board.c 文件, board.h 里需要有bsp的相关定义和相关特性定义;而 board.c 中需要提供系统时钟, NVIC等bsp基本硬件初始化,一般还会提供 rt_hw_board_init() 完成所有板级基本硬件初始化,和 SysTick_Handler() 系统tick时钟ISR(板级相关)

在工程文件夹下新建applications文件夹,拷贝SConscript脚本,创建 startup.c , 其中应该有main(startup汇编文件中定义的入口),这部分可以参考其他bsp步骤来完成(包括 application.c的实现)

注意:

- 板级相关的宏定义等应该在 board.h 中,而不是 rtconfig.h 中, rtconfig.h 中的宏定义是OS层使用的,所以所有的bsp相关定义应当与 rtconfig.h 去相关。
- 由于属于通用Cortex-M4,所以内核部分可以不用过多的关心,对于任何一款Cortex-M4/M3的CPU,只需要处理系统时钟初始化(主要是systick初始化),NVIC基础配置,并且完成systick中断的ISR即可,相关部分的代码会在下节中分析.在Cortex-M3移植中谈到的 PendSV_Handler() PenSV() HardFault_Handler() 已经被 \libcpu\arm\cortex-m4\context_rvds.S 中 PendSV_Handler PROC 所接管(只要在startup文件中相关ISR名字正确)
- 对于 systick 中断,RT-Thread已经有相关的中断处理函数,对于不同的bsp,只需要在bsp中相关 SysTick_Handler() 调用即可,相对是很固定的模式,就像9.4 RT-Thread/STM32 其他部分说明中的模板那样:

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

    rt_tick_increase();

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

- 对于Systick Timer和NVIC的配置,可以使用ARM标准CMSIS库中的函数进行配置,相关函数和帮助可以参考 Keil\ARM\CMSIS\index.html
- 需要根据不同的CPU检查堆栈的配置,在 rt_system_heap_init(...); 时给出正确的地址RAM段范围。

这样,基本的模板就已经实现了,可以在命令行中执行 scons --target=mdk4 -S , 得到项目文件。

9.5.4 Cortex-M4中的内核相关代码分析

下面将以MDK环境为主,分析RT-Thread与Cortex-M4相关的主要的代码(context_rvds.S)
context_rvds.S中主要包括了一下几个部分:

- 中断控制系列(中断开关)

- 上下文切换系列

```
void rt_hw_context_switch(rt_uint32 from, rt_uint32 to); void rt_hw_context_switch_to(rt_uint32 to);
```

- Handler系列(PendSV_Handler , HardFault_Handler)

与Cortex-M3一样,Cortex-M4的开关中断同样也是使用CPSID指令来实现的,参考本文Cortex-M3分析部分

与Cortex-M3不同,上下文切换部分和pendSV部分的代码多了诸如 IF {FPU} != "Soft-VFP" 这样的代码段,这是在MDK编译选项中使用不同FPU类型产生的。

打开Target Options-Target选项卡,其中Code Generation中有Floating Point Hardware选项,如 MDK中FPU设置图 所示

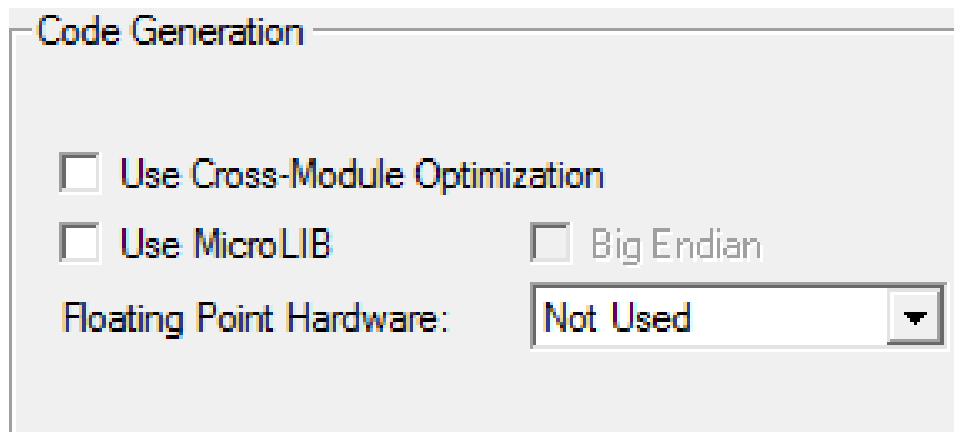


图 9.16: MDK中FPU设置图

有如下2个选项: Not Used和Use FPU选项

- Not Used 会在C/C++选项卡,编译控制中产生-cpu Cortex-M4,即使用Cortex-M4编译
- Use FPU在C/C++选项卡,编译控制中产生-cpu Cortex-M4.fp

FPU是Cortex-M4浮点运算的可选单元。它是一个专用于浮点任务的单元。这个单元通过硬件提升性能,能处理单精度浮点运算,并与IEEE 754标准 兼容。当选择了Not Used时,即不使用硬件FPU,编译时FPU就等于SoftVFP;而选择USE FPU时,cpu变为了Cortex-M4.fp,此时,会默认按照 --fpu=vfpv4-spvfpv4-sp_d16 来进行编译,即使用了硬件FPU,此时, FPU!= “SoftVFP”, 于是IF语句块成立。

关于更多fpu编译的信息可以参考MDK的帮助文件:

ARM Compiler toolchain v5.02 for µVision Using the Compiler

Home > Compiler Coding Practices > Processors and their implicit Floating-Point Units (FPUs)

PendSV_Handler中增量代码分析:

```
IF      {FPU} != "SoftVFP"
TST     lr, #0x10          ; 判断EXC_RETURN[4]是否置位
VSTMFD  r1!, {d8 - d15}    ; EXC_RETURN[4]置位则push{d8~d15}, !表示自增, STMFD=STMDB
ENDIF
```

当系统产生异常(中断)时, 首先会自动进行硬件入栈行为, 对于Cortex-M3, 会入栈 xPSR、PC、LR、R12、R3~R0, 这些寄存器被称为basic frame(参考ARMv7-M Architecture Reference Manual第648页), 对于Cortex-M4, 在入栈basic frame之前, 还会入栈extended frame, 即把d0_{d8}入栈, 然后入栈xPSR、PC、LR、R12、R3~R0。所以, 如果要完整的保护上下文切换的状态, 还需要入栈d8~d15, 如 启用FPU后自动入栈顺序图 所示

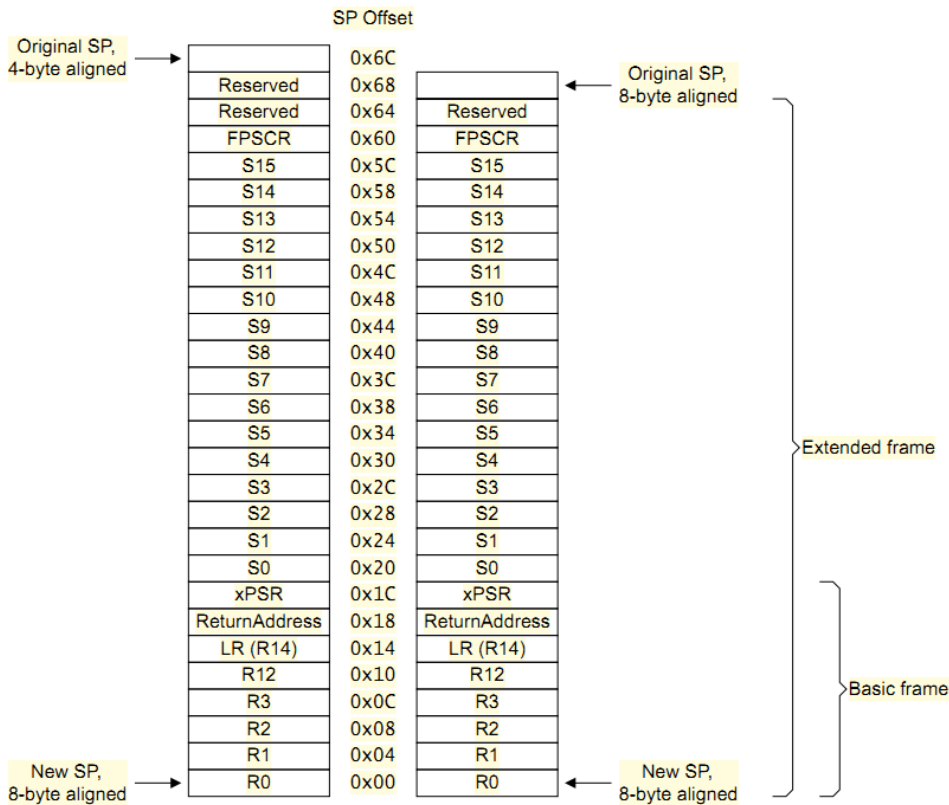


图 9.17: 启用FPU后自动入栈顺序图

并且对于FPU需要使用V指令(参考 ARM Cortex-M4 Processor Technical Reference Manual 第69页7.2.3 FPU instruction set)

```
;以下代码为了保存CONTROL.FPCA
IF      {FPU} != "SoftVFP"
MOV     r4, #0x00          ; 先r4清零

TST     lr, #0x10          ; if(!EXC_RETURN[4])
MOVEQ   r4, #0x01          ; CONTROL.FPCA=1的话,r4=1

STMFDD  r1!, {r4}          ; push CONTROL.FPCA
ENDIF
```

有了保存现场,那么当切回时应该还原:

```
;还原CONTROL.FPCA
IF      {FPU} != "SoftVFP"
LDMFD  r1!, {r3}          ; pop flag
ENDIF
```

其他部分代码与Cortex-M3几乎一样, 请参考Cortex-M3移植分析部分。
至此RT-Thread在MK60上的内核移植完成。

第 10 章

SCons构建系统

SCons是一套由Python语言编写的开源构建系统，类似于GNU Make。它采用不同于通常Makefile文件的方式，而使用SConstruct和SConscript文件来替代。这些文件也是Python脚本，能够使用标准的Python语法来编写。所以在SConstruct、SConscript文件中可以调用Python标准库进行各类复杂的处理，而不局限于Makefile设定的规则。

在[SCons](#)的网站上可以找到详细的SCons用户手册，本章节讲述SCons的基本用法，以及如何在RT-Thread中用好SCons工具。

10.1 什么是构建工具(software construction tool)

构建工具是一种软件，它可以根据一定的规则或指令，将源代码编译成可执行的二进制程序。这是构建工具最基本也是最重要的功能。实际上，构建工具的功能不止于此，通常这些规则有一定的语法，并组织成文件。这些文件用于来控制构建工具的行为，在完成软件构建之外，也可以做其他事情。

目前最流行的构建工具是GNU Make。很多知名开源软件，如Linux内核就采用Make构建。Make通过读取Makefile文件来检测文件的组织结构和依赖关系，并完成Makefile中所指定的命令。

由于历史原因，Makefile的语法比较混乱，不利于初学者学习。此外，在Windows平台上使用Make也不方便，需要安装Cygwin环境。为了克服Make的种种缺点，人们开发了其他构建工具，如CMake和SCons等。

10.2 RT-Thread构建

RT-Thread早期使用Make/Makefile构建。从0.3.x开始，RT-Thread开发团队逐渐引入了SCons构建系统，引入SCons唯一的目的是：使大家从复杂的Makefile配置、IDE配置中脱离出来，把精力集中在RT-Thread功能开发上。

有些读者可能会有些疑惑，这里介绍的构建工具与IDE有什么不同。

通常IDE有自己的管理源码的方式，一些IDE使用XML来组织文件，并解决依赖关系。大部分IDE会根据用户所添加的源码生成类似Makefile或SConscript的脚本文件，在底层调用类似Make与SCons的工具来构建源码。IDE通过图形化的操作来完成构建。

10.3 安装SCons环境

在使用SCons系统前需要在PC主机中安装它，因为它是Python语言编写的，所以在使用SCons之前需要安装Python运行环境。需要注意的是，由于目前SCons还不支持Python 3.x，所以需要安装Python 2.x环境，可以选择Python 2.x的最新版本进行安装。

10.3.1 Linux、BSD环境

在Linux、BSD环境中Python应该是已经默认安装了，一般也是2.x版本系列的Python环境。这时只需要安装SCons即可，例如在Ubuntu中可以使用如下命令：

```
sudo apt-get install scons
```

10.3.2 Windows环境

请到[Python网站](#)下载Python 2.x系列安装包，当前推荐使用Python 2.7.x系列的Python版本。

请到[SCons网站](#)下载SCons安装包，从RT-Thread使用经验来看，SCons的各个版本（1.0.0 - 2.3.x）都可以在RT-Thread上正常使用。

在Windows下安装完成Python 和 SCons后，需要把scons命令添加到系统的PATH环境变量中，假设Python默认安装在

```
C:\Python27
```

目录下，可以把C:\Python27\Scripts目录加入到PATH环境变量中。在Windows的我的电脑中，单击右键把系统属性设置窗口点出来，如下图所示：

点击其中的高级设置

选择PATH项，然后点击编辑按钮，然后把C:\Python27\Scripts目录添加到PATH最后的位置。添加完成后，可以按Win键+R，然后输入cmd回车打开Windows命令行窗口，在其中输入：

```
scons
```

如果能够见到下面的输出，说明Python和SCons安装正确。

10.4 SCons基本使用

初次使用SCons编译某个bsp之前，需要先为bsp指定编译器。这需要修改该bsp目录下的rtconfig.py文件。

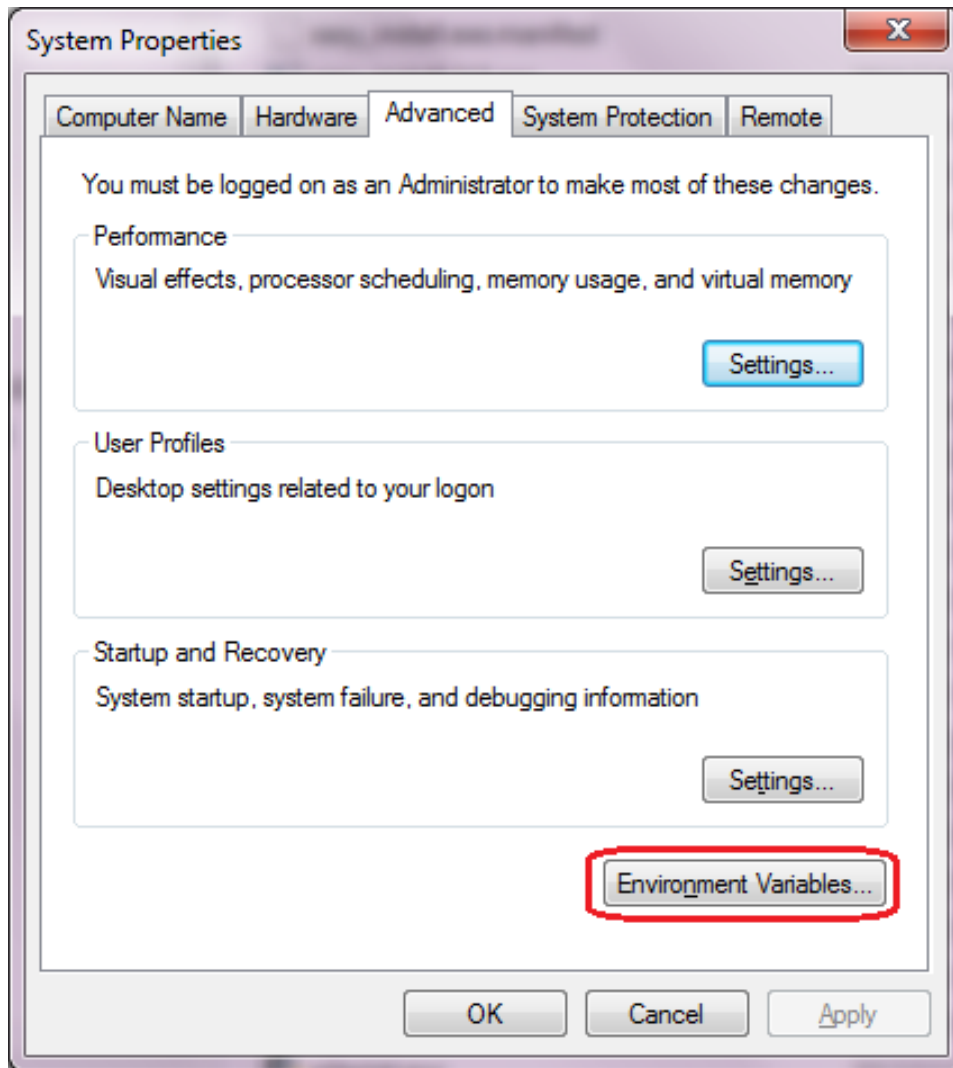


图 10.1: 我的电脑系统属性设置

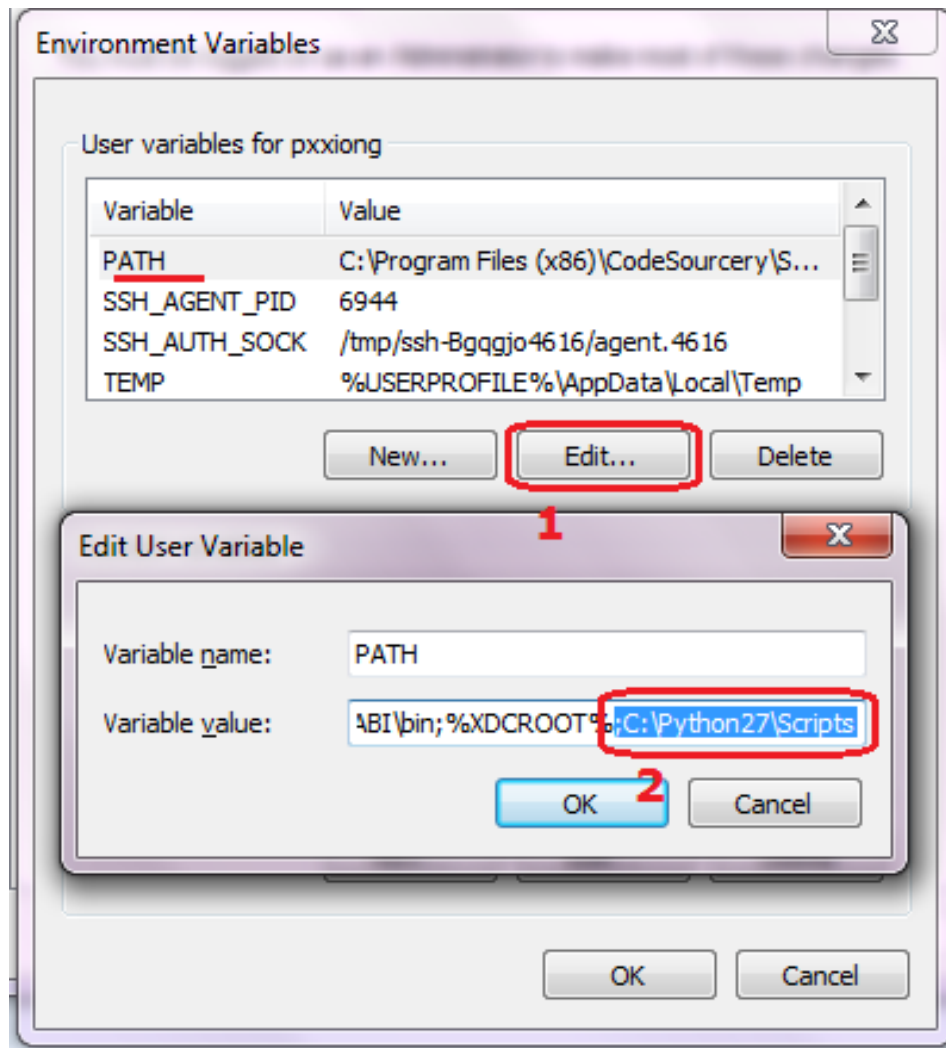


图 10.2: 修改PATH环境变量

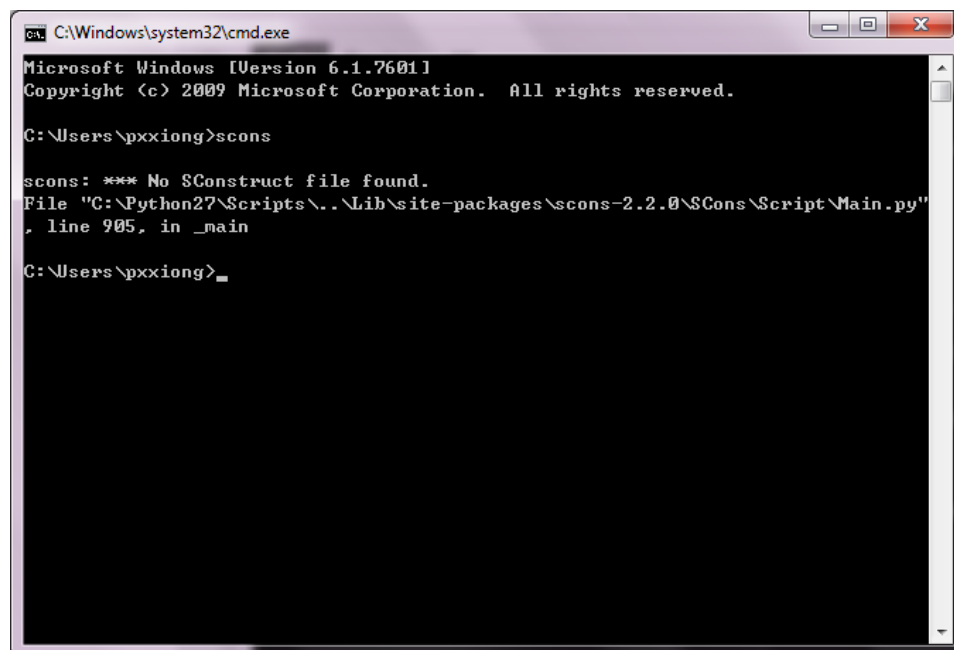


图 10.3: SCons命令行输出

10.4.1 配置编译器

rtconfig.py是一个RT-Thread标准的编译器配置文件，主要用于完成以下工作：

- 指定编译器（从支持多个编译器中选择一个你现在使用的编译器）
- 指定编译器参数，如编译选项、链接选项等

首先确保你的系统上已经安装了编译器。RT-Thread构建系统支持多种编译器。目前支持的编译器包括arm gcc，MDK，IAR，VisualStudio，Visual DSP。主流的ARM Cortex M0、M3、M4平台，基本上ARM GCC、MDK、IAR都是支持的。有一些bsp可能仅支持一种，读者可以阅读该bsp目录下的rtconfig.py查看当前支持的编译器。

这里以bsp/stm32f10x为例，其rtconfig.py如下所示

```
ARCH='arm'
CPU='cortex-m3'
CROSS_TOOL='keil'

if CROSS_TOOL == 'gcc':
    PLATFORM = 'gcc'
    EXEC_PATH = r'D:\SourceryGCC\bin'
elif CROSS_TOOL == 'keil':
    PLATFORM = 'armcc'
    EXEC_PATH = r'C:\Keil'
elif CROSS_TOOL == 'iar':
    PLATFORM = 'iar'
    IAR_PATH = r'E:/Program Files/IAR Systems/Embedded Workbench 6.0'
.....
```

一般来说，我们只需要修改CROSS_TOOL和下面的EXEC_PATH两个选项。

• CROSS_TOOL

指定编译器，可选的值为'keil', 'gcc', 'iar'。大致浏览rtconfig.py可以查看当前bsp所支持的编译器。

• EXEC_PATH

编译器的安装路径。

如果您的机器上安装了MDK，那么将CROSS_TOOL修改为'keil'，并修改EXEC_PATH = r'C:/Keil'为您的MDK的安装路径。

这里有两点需要注意：

1. 安装编译器时（如MDK，GNU GCC，IAR等），不要安装到带有中文或者空格的路径中。否则，某些解析路径时会出现错误。有些程序默认会安装到C:\Program Files目录下，中间带有空格。建议安装时选择其他路径，养成良好的开发习惯。
2. 修改EXEC_PATH时，需要注意路径的格式。在windows平台上，默认的路径分割符号是反斜杠\，而这个符号在C语言以及Python中都是用于转义字符的。所以修改路径时，可以将\改为/，或者在前面加r（python特有的语法，表示原始数据）。

假如某编译器安装位置为D:\Dir1\Dir2下。下面几种是正确的写法:

```
EXEC_PATH = r'D:\Dir1\Dir2'  注意, 字符串前带有r, 则可正常使用“\”  
EXEC_PATH = 'D:/Dir1/Dir2'   注意, 改用“/”, 前面没有r  
EXEC_PATH = 'D:\\Dir1\\Dir2'  注意, 这里使用“\\”的转义性来转义“\”自己。
```

下面是错误的写法:

```
EXEC_PATH = 'D:\Dir1\Dir2'
```

编译器配置完成之后, 我们就可以使用SCons来编译RT-Thread的bsp了。

在当前目录打开命令行窗口, 执行scons, 就会启动编译过程。

小技巧:

在WIN7上, 在当前目录按下SHIFT+鼠标右键, 弹出的菜单中, 会有“在此处打开命令行窗口”的菜单项。点击可以快速打开CMD窗口。

10.4.2 SCons基本命令

本节介绍RT-Thread中常用的SCons命令。SCons不仅完成基本的编译, 还可以生成MDK/IAR/VS工程。

scons

```
scons
```

这个命令用于直接编译目标。如果执行过scons后修改一些文件, 再次执行scons时, 则SCons会进行增量编译, 仅编译修改过的文件并链接。

注:

如果在Windows上执行scons输出以下的警告信息,

```
scons: warning: No version of Visual Studio compiler found - C/ C +  
+ compilers most likely not set correctly
```

说明scons并没在你的机器上找到Visual Studio编译器, 但实际上我们主要是针对设备开发, 和Windows本地没什么关系。请直接忽略掉它。

scons -jN

多线程编译目标, 在多核计算机上可以加快编译速度。一般来说, 一颗cpu核心可以支持2个线程。双核机器上使用-j4即可。

```
scons -j4
```

*** 注：如果你只是想看看编译错误或警告，最好是不使用-j参数，这样错误信息不会因为多个文件并行编译而导致出错信息夹杂在一起 ****

```
scons -c
```

清除编译目标。这个命令会清除执行scons时生成的临时文件和目标文件。

```
scons -target=XXX -s
```

```
scons --target=mdk4 -s
```

可以在当前目录生成一个新的名为project.uvproj文件。双击它打开，就可以使用MDK来编译、调试。不习惯SCons的同学可以使用这种方式。

当修改了rtconfig.h打开或者关闭某些组件时，也需要使用这个命令重新生成对应的定制化的工程。

注意：

要生成Keil MDK的工程文件，前提条件是当前目录存在一个工程模版文件，然后scons才会根据这份模版文件加入相关的源码，头文件搜索路径，编译参数，链接参数等。而至于这个工程是针对哪颗芯片的，则直接由这份工程模版文件指定。所以大多数情况下，这个模版文件是一份空的工程文件，用于辅助SCons生成project.uvproj。

如果打开project.uvproj失败，请删除project.uvopt后，重新生成工程。

```
scons --target=iar -s
```

自动生成IAR工程；

```
scons --target=vs2012 -s  
Scons --target=vs2005 -s
```

在bsp/simulator下，可以使用这个命令生成vs2012的工程或vs2005的工程。

```
scons -verbose
```

默认情况下，scons编译的输出不会显示编译参数，如下所示：

```
F:\Project\git\rt-thread\bsp\stm32f10x>scons  
scons: Reading SConscript files ...  
scons: done reading SConscript files.
```

```
scons: Building targets ...
scons: building associated VariantDir targets: build
CC build\applications\application.o
CC build\applications\startup.o
CC build\components\drivers\serial\serial.o
...
```

使用scons -verbose的效果

```
armcc -o build\src\mempool.o -c --device DARMSTM --apcs=interwork -ID:/Keil/ARM/
RV31/INC -g -O0 -DUSE_STDPERIPH_DRIVER -DSTM32F10X_HD -Iapplications -IF:\Projec
t\git\rt-thread\applications -I. -IF:\Project\git\rt-thread -Idrivers -IF:\Proje
ct\git\rt-thread\drivers -ILibraries\STM32F10x_StdPeriph_Driver\inc -IF:\Project
\git\rt-thread\Libraries\STM32F10x_StdPeriph_Driver\inc -ILibraries\STM32_USB-FS
_Device_Driver\inc -IF:\Project\git\rt-thread\Libraries\STM32_USB-FS_Device_Driv
er\inc -ILibraries\CMSIS\CM3_DeviceSupport\ST\STM32F10x -IF:\Project\git\rt-thre
ad\Libraries\CMSIS\CM3_DeviceSupport\ST\STM32F10x -IF:\Project\git\rt-thread\com
ponents\CMSIS\Include -Iusb -IF:\Project\git\rt-thread\usb -I. -IF:\Project\git\
rt-thread -IF:\Project\git\rt-thread\include -IF:\Project\git\rt-thread\libcpu\arm
cortex-m3 -IF:\Project\git\rt-thread\libcpu\arm\common -IF:\Project\git\rt-th
read\components\drivers\include -IF:\Project\git\rt-thread\components\drivers\in
clude -IF:\Project\git\rt-thread\components\finsh -IF:\Project\git\rt-thread\com
ponents\init F:\Project\git\rt-thread\src\mempool.c
...
```

10.5 SCons进阶

SCons使用SConscript和SConstruct文件来组织源码结构，通常来说一个项目只有一个SConstruct，但是会有多个SConscript。一般情况下，每个存放有源代码的子目录下都会放置一个SConscript，这些SCons的脚本文件组成如下所示的等级结构。

[图片待补充]

为了使RT-Thread更好的支持多种编译器，以及方便的调整编译参数，RT-Thread为每个bsp单独创建了一个名为rtconfig.py的文件。因此每一个RT-Thread bsp目录下都会存在下面三个文件，它们具体控制BSP的编译。

```
rtconfig.py
SConstruct
SConscript
```

大部分组件源码文件夹下存在SConscript文件，这些文件会被BSP目录下的SConscript文件“找到”从而将rtconfig.h中定义的组件加入编译器来。一个BSP中只有一个SConstruct文件，但是却会有多个SConscript文件，可以说SConscript文件是组织源码的主力军。

10.5.1 修改编译器选项

在rtconfig.py中控制了大部分编译选项。下面以stm32f10x/rtconfig.py为例（部分）

```

elif PLATFORM == 'armcc':
    # toolchains
    CC = 'armcc'
    AS = 'armasm'
    AR = 'armar'
    LINK = 'armlink'
    TARGET_EXT = 'axf'

    DEVICE = ' --device DARMSTM'
    CFLAGS = DEVICE + ' --apcs=interwork'
    AFLAGS = DEVICE
    LFLAGS = DEVICE + ' --info sizes --info totals --info unused --info veneers --list rtthread-
stm32.map --scatter stm32_rom.sct'

    CFLAGS += ' -I' + EXEC_PATH + '/ARM/RV31/INC'
    LFLAGS += ' --libpath ' + EXEC_PATH + '/ARM/RV31/LIB'

    EXEC_PATH += '/arm/bin40/'

    if BUILD == 'debug':
        CFLAGS += ' -g -O0'
        AFLAGS += ' -g'
    else:
        CFLAGS += ' -O2'

    POST_ACTION = 'fromelf --bin $TARGET --output rtthread.bin \nfromelf -z $TARGET'

```

其中CFLAGS存储C文件的编译选项，AFLAGS 则是汇编文件的编译选项，LFLAGS 是链接选项。BUILD 变量控制代码优化的级别。默认 BUILD 变量取值为'debug'，即使用debug方式编译，优化级别O。如果将这个变量修改为其他值，就会使用优化级别2编译。下面几种都是可行的写法（总之只要不是'debug'就可以了）。

```

BUILD = ''
BUILD = 'release'
BUILD = 'hello, world'

```

建议在开发阶段都使用debug方式编译，不开优化，等产品稳定之后再考虑优化。

关于这些选项的具体含义需要参考编译器手册，如上面使用的armcc是MDK的底层编译器。其编译选项的含义在MDK help中有详细说明。

10.5.2 内置函数

如果想要将自己的一些源代码加入到SCons编译环境中，一般可以创建或修改已有SConscript文件。SConscript文件可以控制源码文件的加入，并且可以指定文件的Group（与MDK/IAR等IDE中的Group的概念类似）。

SCons提供了很多内置函数可以帮助我们快速添加源码程序。简单介绍一些常用函数。

```
GetCurrentDir()
```

获取当前路径

```
Glob('*.*')
```

获取当前目录下的所有C文件。修改参数的值为其他后缀就可以匹配当前目录下的所有某类型的文件。

```
GetDepend(macro)
```

在tools/目录下的脚本文件中定义，它会从rtconfig.h文件读取组件配置信息，其参数为rtconfig.h中的宏名。如果rtconfig.h打开了某个宏，则这个方法（函数）返回真，否则返回假。

```
Split(str)
```

将字符串str分割成一个list

```
DefineGroup(name, src, depend, **parameters)
```

这是RT-Thread基于SCons扩展的一个方法（函数）。DefineGroup用于定义一个组件。组件可以是一个目录（下的文件或子目录），也是后续一些IDE工程文件中的一个Group或文件夹。

- name来定义这个group的名字
- src用于定义这个Group中包含的文件，一般指的是C/C++源文件。方便起见，也能够通过Glob函数采用通配符的方式列出SConscript文件所在目录中匹配的文件。
- depend 用于定义这个Group编译时所依赖的选项（例如finsh组件依赖于RT_USING_FINSH宏定义）。编译选项一般指rtconfig.h中定义的RT_USING_xxx宏。当在rtconfig.h配置文件中定义了相应宏时，那么这个Group才会被加入到编译环境中进行编译。如果依赖的宏并没在rtconfig.h中被定义，那么这个Group将不会被加入编译。相类似的，在使用scons生成为IDE工程文件时，如果依赖的宏未被定义，相应的Group也不会出现在工程文件中出现。
- parameters则可以输入一组字符串，后面还可以加入的参数包括：
 - CCFLAGS - C源文件编译参数；
 - CPPPATH - 头文件路径；
 - CPPDEFINES - 添加预定义宏；
 - LINKFLAGS - 链接时参数。
 - LIBRARY - 包含此参数，则会将组件生成的目标文件打包成库文件

可见DefineGroup的功能十分强大，实际使用时不需要配置所有参数。


```
SConscript(dirs, variant_dir, duplicate)
```

SCons内置函数。其参数包括三个：

- dirs指明SConscript文件路径，
- variant_dir指定生成的目标文件的存放路径，
- duplicate的作用是设定是否拷贝或链接源文件到variant_dir

利用这些函数，再配合一些简单的Python语句我们就能随心所欲向项目中添加或者删除源码了。下一节我们将介绍几个典型的SConscript示例文件来学习，并达到举一反三的目的。

10.5.3 SConscript示例1

bsp/stm32f10x/application/SConscript

```

1
2 Import('RTT_ROOT')
3 Import('rtconfig')
4 from building import *
5
6 src = Glob('*.*')
7 cwd = GetCurrentDir()
8 include_path = [cwd]
9
10 group = DefineGroup('Applications', src, depend = [''], CPPPATH = include_path)
11
12 Return('group')
```

上面这个脚本完成如下功能：

src = Glob('*.*')得到当前目录下所有的C文件，cwd = GetCurrentDir()将当前路径赋值给cwd，注意cwd是一个字符串；include_path = [cwd]将当前头文件路径保存为一个list变量。最后一行使用DefineGroup创建一个组。组名为Applications。depend为空，表示该组不依赖任何rtconfig.h的任何宏。CPPPATH = include_path表示将当前目录添加到系统的头文件路径中。

总结：这个源程序会将当前目录下的所有c程序加入到组Applications中，并将这个目录添加到系统头文件搜索路径中。因此，如果在这个目录下增加或者删除文件，就可以将文件加入工程或者从工程中删除。

它适用于批量添加源码文件。

10.5.4 SConscript示例2

component/finsh/SConscript

```

1
2 Import('rtconfig')
```

```

3  from building import *
4
5  cwd      = GetCurrentDir()
6  src      = Glob('*.c')
7  CPPPATH = [cwd]
8  if rtconfig.CROSS_TOOL == 'keil':
9      LINKFLAGS = ' --keep __fsym_* --keep __vsym_* '
10 else:
11     LINKFLAGS = ''
12
13 group = DefineGroup('finsh', src, depend = ['RT_USING_FINSH'], CPPPATH = CPPPATH,
14     LINKFLAGS = LINKFLAGS)
15
16 Return('group')

```

从第7行开始，与示例1有些区别。

```

if rtconfig.CROSS_TOOL == 'keil':
    LINKFLAGS = ' --keep __fsym_* --keep __vsym_* '
else:
    LINKFLAGS = ''

```

这是Python的条件判断语句，如果编译工具是keil，则变量LINKFLAGS = --keep __fsym_* --keep __vsym_*，否则置空。

DefinGroup同样将finsh目录下的所有文件创建为finsh组。

depend = ['RT_USING_FINSH']表示这个组依赖rtconfig.h中的RT_USING_FINSH。即，当rtconfig.h中打开宏RT_USING_FINSH时，finsh组内的源码才会被实际编译，否则SCons不会编译。

CPPPATH = CPPPATH，左边的CPPPATH是DefineGroup中内置参数，右边的CPPPATH是本文第6行定义的，意思是将finsh目录加入到系统头文件目录中。这样我们就可以在其他源码中引用finsh目录下的头文件了，如finsh.h。

LINKFLAGS = LINKFLAGS的含义与CPPPATH = CPPPATH类似。左边的LINKFLAGS表示链接参数，右边的LINKFLAGS则是前面if else语句所设定的值。

10.5.5 SConscript示例3

bsp/stm32f10x/SConscript

```

1
2 # for module compiling
3 import os
4 Import('RTT_ROOT')
5
6 cwd = str(Dir('#'))
7 objs = []
8 list = os.listdir(cwd)

```

```

9
10 for d in list:
11     path = os.path.join(cwd, d)
12     if os.path.isfile(os.path.join(path, 'SConscript')):
13         objs = objs + SConscript(os.path.join(d, 'SConscript'))
14
15 Return('objs')

```

`cwd = str(Dir('#'))` 获取工程的顶级目录，也就是工程的SConstruct所在的目录，在这里它的效果与 `cwd = GetCurrentDir()` 相同。随后定义了一个空的list型变量`objs`。第6行 `list = os.listdir(cwd)` 得到当前目录下的所有子目录，并保存到变量`list`中。随后是一个python的for循环，其含义是取出一个当前目录的子目录，利用`os.path.join(cwd,d)`拼接成一个完整路径，然后判断这个子目录是否存在一个名为`SConscript`的文件，若存在，则执行

```
objs = objs + SConscript(os.path.join(d, 'SConscript'))
```

上面这一句中使用了SCons提供的一个内置函数`SConscript`，它可以读入一个新的`SConscript`文件，并将`SConscript`文件中所指明的源码加入编译列表中来。

10.5.6 SConscript示例4

stm32f10x/drivers/SConscript

```

1
2 Import('RTT_ROOT')
3 Import('rtconfig')
4 from building import *
5
6 cwd = GetCurrentDir()
7
8 # add the general drivers.
9 src = Split(''
10 board.c
11 stm32f10x_it.c
12 led.c
13 usart.c
14 '')
15
16 # add Ethernet drivers.
17 if GetDepend('RT_USING_LWIP'):
18     src += ['dm9000a.c']
19
20 # add Ethernet drivers.
21 if GetDepend('RT_USING_DFS'):
22     src += ['sdcard.c']
23

```

```

24 # add Ethernet drivers.
25 if GetDepend('RT_USING_RTC'):
26     src += ['rtc.c']
27
28 # add Ethernet drivers.
29 if GetDepend('RT_USING_RTGUI'):
30     src += ['touch.c']
31     if rtconfig.RT_USING_LCD_TYPE == 'ILI932X':
32         src += ['ili_lcd_general.c']
33     elif rtconfig.RT_USING_LCD_TYPE == 'SSD1289':
34         src += ['ssd1289.c']
35
36 CPPPATH = [cwd]
37
38 group = DefineGroup('Drivers', src, depend = [''], CPPPATH = CPPPATH)
39
40 Return('group')

```

第8行使用Split方法来将一个文件字符串分割成一个list，其效果等价于

```
src = ['board.c', 'stm32f10x_it.c', 'led.c', 'usart.c']
```

第15行到第33行使用了GetDepend方法检查rtconfig.h中的某个宏是否打开，如果打开，则使用src += [src_name]来添加源码。最后使用DefineGroup创建组。

10.5.7 添加库

在进行编译时添加一个额外的库，需要注意不同的工具链对二进制库的命名。例如GCC工具链，它识别的是libabc.a这样的库名称，在指定库时是指定abc，而不是libabc。所以在链接额外库时需要在SConscript文件中特别注意。另外，在指定额外库时，也最好指定相应的库搜索路径，以下是一个示例：

```

1 # RT-Thread building script for component
2
3 Import('rtconfig')
4 from building import *
5
6 cwd = GetCurrentDir()
7 src = Split(''
8 '')
9
10 LIBPATH = [cwd + '/libs']
11 LIBS = ['abc']
12
13 group = DefineGroup('ABC', src, depend = [''], LIBS = LIBS, LIBPATH=LIBPATH)

```

如果工具链是GCC，则库的名称应该是libabc.a；如果工具链是armcc，则库的名称应该是abc.lib。库的搜索路径是当前目录下的‘libs’目录。

10.5.8 增加一个SCons命令

10.5.9 RT-Thread building脚本

在RT-Thread tools目录下存放有RT-Thread自己定义的一些辅助building的脚本，例如用于自动生成RT-Thread针对一些IDE集成开发环境的工程文件。其中最主要的是building.py脚本。

10.6 简单的SConstruct

例如针对一个hello world的简单程序，假设它的源文件是：

```
/* file: hello.c */
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Hello, world!\n");
}
```

只需要在这个文件目录下添加一个如下内容的SConstruct文件：

```
Program('hello.c')
```

然后在这个目录下执行命令：

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -o hello.o -c hello.c
cc -o hello hello.o
scons: done building targets.
```

将会在当前目录下生成hello的应用程序。所以相比于Makefile，一个简单的hello.c到hello的转换，只需要一句话。如果hello是由两个文件编译而成，也只需要把SConstruct文件修改成：

```
Program(['hello.c', 'file1.c'])
```

同时也可以指定编译出的目标文件名称：

```
Program('program', ['hello.c', 'file1.c'])
```

有的时候也可以偷偷懒，例如把当前目录下的所有C文件都作为源文件来编译：

```
Program('program', Glob('*.c'))
```

Glob函数就是用于使用当前目录下的所有C文件。除了Glob函数以外，也有Split函数。Split函数写的脚本具备更好的可读性以及更精确的可定制性：

```
src = Split('''
    hello.c
    file1.c
    ''')
Program('program', src)
```

它的效果与 `Program('program', ['hello.c', 'file1.c'])` 是一致的，但具有更清晰的可读性。

10.7 SConstruct与SConscript

对于复杂、大型的系统，显然不仅仅是一个目录下的几个文件就可以搞定的，很可能是由数个文件夹一级级组合而成。

在SCons中，可以编写SConscript脚本文件来编译这些相对独立目录中的文件，同时也可以使用SCons中的Export和Import函数在SConstruct与SConscript文件之间共享数据（也就是Python中的一个对象数据）。

第 11 章

finsh shell

11.1 简介

finsh是RT-Thread的命令行外壳（shell），提供一套供用户在命令行的操作接口，主要用于调试或查看系统信息。finsh支持两种模式：

1. C语言解释器模式，为行文方便称之为c-style；
2. 传统命令行模式，此模式又称为msh(module shell)。

C语言表达式解释模式下，finsh能够解析执行大部分C语言的表达式，并使用类似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。

在msh模式下，finsh运行方式类似于dos/bash等传统shell。

在本章的最后一节宏选项中介绍如何配置finsh，读者可以根据自己的喜好配置finsh。

11.2 工作模式

用户由设备端口输入命令行，finsh 通过对设备输入的读取，解析输入内容，然后自动扫描内部段（内部函数表），寻找对应函数名，执行函数后输出回应。

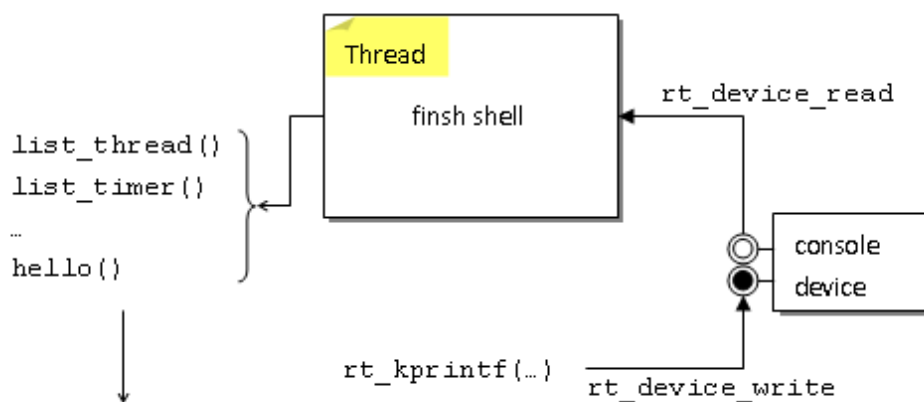


图 11.1: finsh的数据流结构

11.3 什么是shell?

在计算机发展的早期，图形系统出现之前，没有鼠标，甚至没有键盘。那时候人们如何与计算机交互呢？最早期的计算机使用打孔的纸条向计算机输入命令，编写程序。后来计算机不断发展，显示器、键盘成为计算机的标准配置，但此时的操作系统还不支持图形界面，计算机先驱们开发了一种软件，它接受用户输入的命令，解释之后，传递给操作系统，并将操作系统执行的结果返回给用户。这个程序像一层外壳包裹在操作系统的外面，所以它被称为shell。

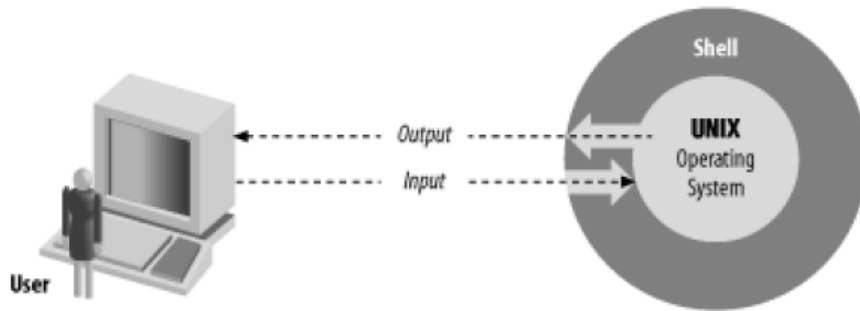


图 11.2: 系统结构图

在图形界面系统出现之前，shell这个命令程序曾经统治计算机的交互接口数十年之久，并由大名鼎鼎的Unix系统发扬光大，诞生了多种shell软件，如bsh、bash、csh、ksh、zsh。这些shell功能都非常强大，不仅可以供用户输入命令，它们还都支持shell编程语言，可以完成复杂的操作。这些shell目前都可以在*nix系统上使用。即使后来windows统治PC，对于一般人来说，shell的光彩逐渐暗淡，它并未因此退出操作系统。在windows上，cmd 可以认为就是一种shell，同时在windows的后续版本发展出更强大的powershell。

11.4 初识finsh

在大部分嵌入式系统中，一般开发调试都使用硬件调试器和printf日志打印，在有些情况下，这两种方式并不是那么好用。比如对于RT-Thread这个多线程系统，我们想知道某个时刻系统中的线程运行状态、手动控制系统状态。如果有一个shell，就可以输入命令，直接执行相应的函数获得需要的信息，或者控制程序的行为，这无疑会十分方便。

11.4.1 finsh(C-Style)

在嵌入式领域，C语言是最常用的开发语言，如果shell程序的命令是C语言的风格，那无疑是非常易用且有趣。

嵌入式设备通常采用交叉编译，一般需要将开发板与PC机连接起来通讯，常见连接方式包括，串口、USB、以太网、wifi等。一个灵活的shell也应该可以在多种连接方式上工作。

finsh正是基于这些考虑而诞生的，finsh可以发音为[ɪfɪnʃ]。finsh运行于开发板，它可以使用串口/以太网/USB等与PC机进行通信。其运行时的finsh工作原理图 所示。

下图是finsh的实际运行运行效果图。开发板运行RT-Thread，并使能了finsh组件，通过串口与PC机连接，PC上运行Secure CRT。

按下回车，然后输入list_thread()将会打印系统当前所有线程，及其状态。关于这个命令的详细解释请参考本章最后一节。

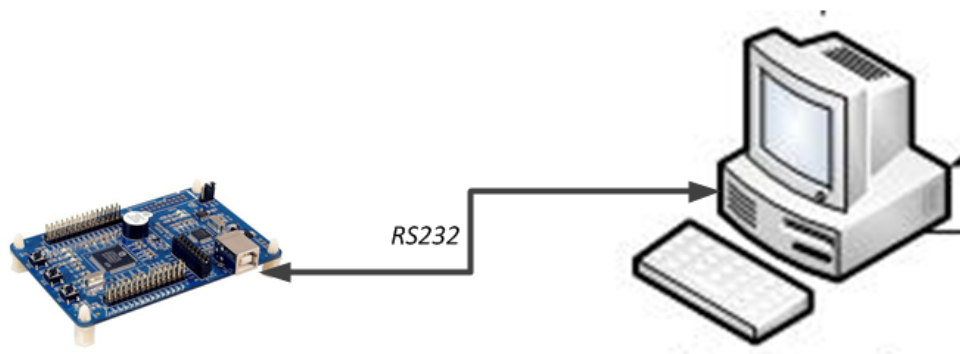


图 11.3: finsh工作原理

```
serial-com4 - SecureCRT
File Edit View Options Transfer Script Tools Window Help
Enter host <Alt+R>
serial-com4 x
- RT - Thread operating System
  /   \ 1.2.0 build oct '20 2013
2006 - 2013 Copyright by rt-thread team
finsh>>
finsh>>
--function:
led          -- set led[0 - 1] on[1] or off[0].
list_mem     -- list memory usage information
hello        -- say hello world
version      -- show RT-Thread version information
list_thread  -- list thread
list_sem     -- list semaphore in system
list_event   -- list event in system
list_mutex   -- list mutex in system
list_mailbox -- list mail box in system
list_msgqueue -- list message queue in system
list_mempool -- list memory pool in system
list_timer   -- list timer in system
list_device  -- list device in system
list         -- list all symbol in system
--variable:
dummy        -- dummy variable for finsh
finsh>>
finsh>>
finsh>>
finsh>>he
--function:
hello        -- say hello world
finsh>>hello()
Hello RT-Thread!
0, 0x00000000
finsh>>
```

图 11.4: finsh运行界面

WARNING: 此模式下，finsh是一个C语言风格的Shell，与Linux/Unix以及Windows下的cmd 的风格不同。在finsh shell中使用命令（即C语言中的函数），必须类似C语言中的函数调用方式，即必须携带()符号。最后finsh命令的输出为此函数的返回值。对于一些不存在返回值的函数（void返回值），这个打印输出没有意义。

11.4.2 finsh(msh)

实际上，一开始finsh仅支持C-Style模式。后来随着RT-Thread的不断发展，尤其支持app module之后（请参考本书应用模块一章了解相关内容），C-Style模式执行app module时操作起来不太方便，而传统的shell则更方便些。另外，C-Style模式下，finsh占用体积较大。出于这些考虑，在RT-Thread 1.2.0中为finsh增加了msh模式。

msh模式下，finsh与传统shell（dos/bash）执行方式一致，其命令执行格式如下

```
command [arg1] [arg2] [...]
```

其中command既可以RT-Thread内置的命令，也可以是编译出的app module（类似于dos里的exe可执行文件）。finsh(msh)内置的命令风格采用bash的风格，内置命令将在本章后续章节详细介绍。

11.4.3 finsh中的按键

finsh支持TAB键自动补全，当没有输入任何字符时按下TAB键将会打印当前所有的符号，包括当前导出的所有命令和变量。若已经输入部分字符时按下TAB键，将会查找匹配的符号，并自动补全，并可以继续输入，多次补全。如果是msh状态，输入一个字符后，不仅仅会按系统导出函数命令方式自动补全，也会按照文件系统的当前目录下的文件名进行补全。

上下键可以回溯最近输入的历史命令，左右键可移动光标，退格键删除。

目前finsh的按键处理还比较薄弱。不支持CTRL+C等控制键中断命令，也不支持DELETE键删除。

11.5 finsh特性

11.5.1 finsh(c-style)的数据类型

finsh支持基本的C语言数据类型，包括：

数据类型	描述
void	空数据格式，只用于创建指针变量
char, unsigned char	（带符号）字符型变量
int, unsigned int	（带符号）整数型变量
short, unsigned short	（带符号）短整型变量
long, unsigned long	（带符号）长整型变量
char, short, long, void	指针型变量

在finsh的命令行上，输入上述数据类型的C表达式可以被识别。浮点类型以及复合数据类型unin与struct等暂不支持。此外，finsh也不支持if, for, while, goto, switch等分支、跳转语句。

11.6 finsh(c-style)中增加命令/变量

finsh支持两种方式向finsh中输出符号（函数或变量），下面将分别介绍这两种方式。

11.6.1 宏方式

需要在rtconfig.h中定义宏FINSH_USING_SYMTAB。

```
#include <finsh.h>
FINSH_FUNCTION_EXPORT(name, desc)
```

参数	描述
name	函数指针，一般为函数名；
desc	针对这个函数命令的描述信息；

desc一般为一段字符串，中间不可以有逗号，两边也没有引号。

```
FINSH_VAR_EXPORT(name, type, desc)
```

参数	描述
name	变量名；
type	变量类型；
desc	变量描述；

type的包括以下类型：

```
enum finsh_type{
    finsh_type_unknown = 0,           /**< unknown data type */
    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 */
};
```

此外FINSH还提供了另外一个宏，它在输出函数为命令时，可以指定命令的名字。

```
#include <finsh.h>
FINSH_FUNCTION_EXPORT_ALIAS(name, alias, desc)
```

参数	描述
name	函数指针，一般为函数名；
alias	输出到finsh中的命令名；
desc	函数描述；

当函数名字过长时，可以利用上面这个宏导出一个简短的名字以方便输入。

- 说明: FINSH的函数名字长度有一定限制，它由finsh.h中的宏定义FINSH_NAME_MAX控制，默认是16字节。意味着finsh命令长度不会超过16字节。这里有个潜在的问题。当一个函数名长度超过FINSH_NAME_MAX时，使用FINSH_FUNCTION_EXPORT导出这个函数到命令表中后，在finsh符号表中看到完整的函数名，但是完整输入执行会出现null node错误。这是因为虽然显示了完整的函数名，但是实际上finsh中却保存了前16字节作为命令，过多的输入会导致无法正确找到命令，这时就可以使用FINSH_FUNCTION_EXPORT_ALIAS来对导出的命令进行重命名。

一个简单的输出函数和变量到finsh的例子：

```
#include <finsh.h>

int var;

int hello_rtt(int a)
{
    rt_kprintf("hello, world! I am %d\n", a);
    return a;
}
FINSH_FUNCTION_EXPORT(hello_rtt, say hello to rtt)
FINSH_FUNCTION_EXPORT_ALIAS(hello_rtt, hr, say hello to rtt)

FINSH_VAR_EXPORT(var, finsh_type_int, just a var for test)
```

编译后运行，可以看到finsh中增加了两个命令，一个变量var。

11.6.2 函数方式

```
#include <finsh.h>
void finsh_syscall_append(const char* name, syscall_func func)
```

参数	描述
name	函数在finsh中访问的名称，即命令名;
func	函数地址，一般为函数名;

这个函数可以输出一个函数到finsh中，使之可以在finsh命令行中使用。

```
#include <finsh.h>
void finsh_sysvar_append(const char* name, u_char type, void* addr)
```

参数	描述
name	变量在finsh中访问的名称，即命令名;
type	变量的类型;
addr	变量地址;

这个函数用于输出一个变量到finsh中。

11.7 msh中增加命令

当使用msh模式时，finsh不支持C表达式，因此只能添加（函数）命令，并不能向c-style模式下动态创建变量。msh模式添加命令仅支持下面两种宏方式添加命令。

```
#include <finsh.h>
MSH_CMD_EXPORT(command, desc);
FINSH_FUNCTION_EXPORT_ALIAS(name, alias, desc);
```

11.7.1 添加内置命令

下面是一种向msh导出命令的方式：

```
#include <finsh.h>

int mycmd(void)
{
    printf("hello!\n");
    return 0;
}

MSH_CMD_EXPORT(mycmd, my command test);
```

在上面的代码例子中，定义了自己的命令函数 mycmd ，这个函数会在命令行中输出一句

hello!

当我们想在msh中能够调用这个命令时，就可以使用MSH_CMD_EXPORT的宏来定义这个函数导出到msh中。把以上的代码加入到系统中，并进行编译，当系统运行起来进入命令行后，我们可以在命令行下输入：

```
msh /> mycmd  
hello!
```

来调用这个mycmd命令。我们可以看到，这个命令方式和finsh的方式并不一样，在它的后面不需要加入()括号（类似C代码中调用一个C函数那样），而是直接回车即可执行这个命令。

同样的，在msh下也可以使用参数，例如如下的示例代码：

```
#include <finsh.h>  
  
int mycmdarg(int argc, char** argv)  
{  
    printf("argv[0]: %s\n", argv[0]);  
  
    if (argc > 1)  
        printf("argv[1]: %s\n", argv[1]);  
  
    return 0;  
}  
MSH_CMD_EXPORT(mycmdarg, my command with args);
```

当我们在命令行下运行这个命令时，特别是以不同的参数运行时，会发现：

```
msh /> mycmdarg  
argv[0]: mycmdarg  
  
msh /> mycmdarg 0  
argv[0]: mycmdarg  
argv[1]: 0  
  
msh /> mycmdarg str 1 2 3  
argv[0]: mycmdarg  
argv[1]: str
```

即，

- argc - 反映的是总计有多少个命令行参数（也包含命令行自身）；
- argv - 反映的是命令行参数组，且都是以字符形式存储；

从上的例子可以看出，一个msh导出函数很类似于一个main函数：

```
int main(int argc, char** argv);
```

最初的msh设计确实就是按照主函数方式进行的，所以其命令行参数传递风格也和main函数完全一致。当对命令行参数进行完整的校验时，就可以确保参数的合法性，并对非法的参数提供出相应的错误信息出来。

使用宏导出命令的形式，

```
MSH_CMD_EXPORT(cmd, cmd description);
```

可以导出到msh下。实际上，使用finsh的函数导出宏也可以导出成msh的命令，两者的差别是函数命令在实际存放时，msh的命令名字上会多出__cmd_的前缀。例如以下的finsh导出宏定义也同样的可以在msh中导出对应的命令：

```
FINSH_FUNCTION_EXPORT_ALIAS(cmd_ls, __cmd_ls, List information about the FILEs.);
```

这里面就是把cmd_ls函数重命名成__cmdls导出到shell中。当执行这个命令时，它会被特殊对待，只能当成msh命令使用。实际上，纯粹的finsh shell在显示命令时，对__开头的函数名并不显示，会被当成一类特殊的命令对待（例如提供给msh的函数命令）。

11.8 RT-Thread内置命令

在RT-Thread中默认内置了一些finsh命令，在finsh中按下TAB键可以打印则会当前系统支持所有符号，也可以输入list()回车，二者效果相同。

11.8.1 finsh(c-style)

注意：在finsh(c-style)中使用命令（即C语言中的函数），必须类似C语言中的函数调用方式，即必须携带“()”符号。finsh shell的输出为此函数的返回值，对于那些不存在返回值的函数，这个打印输出没有意义。要查看命令行信息必须定义对应相应的宏。

```
finsh>>list()
```

显示当前系统中存在的命令及变量，执行结果如下：

```
--Function List:
list_mem      -- list memory usage information
hello         -- say hello world
version       -- show RT-Thread version information
list_thread   -- list thread
list_sem      -- list semaphore in system
list_event    -- list event in system
list_mutex    -- list mutex in system
list_mailbox  -- list mail box in system
list_magqueue -- list messgae queue in system
list_mempool  -- list memory pool in system
list_timer    -- list timer in system
list_device   -- list device in system
list          -- list all symbol in system
--Variable List:
dummy         -- dummy variable for finsh
```

```
0, 0x00000000

finsh>>list_thread()

thread  pri  status   sp      stack size  max used   left tick  error
-----
tidle   0x1f  ready  0x00000058  0x00000100  0x00000058  0x0000000b  000
shell   0x14  ready  0x00000080  0x00000800  0x000001b0  0x00000006  000
```

显示当前系统中线程状态：

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

```
finsh>>list_sem()

semaphore  v   suspend thread
-----
```

显示系统中信号量状态：

字段	描述
semaphore	信号量的名称；
v	信号量当前的值；
suspend thread	等待这个信号量的线程数目；

```
finsh>>list_event()

event  set   suspend thread
-----
```


显示系统中事件状态：

字段	描述
event	事件的名称；
set	事件当前的值；
suspend thread	等待这个事件的线程数目；

```
finsh>>list_mutex()
```

```
mutex    owner    hold    suspend thread
-----
fslock   (NULL)    0000    0
lock     (NULL)    0000    0
```

显示系统中互斥量状态：

字段	描述
mutxe	互斥量的名称；
owner	当前持有互斥量的线程；
hold	持有者在这个互斥量上嵌套持有的次数；
suspend thread	等待这个互斥量的线程数目；

```
finsh>>list_mb()
```

```
mailbox  entry  size  suspend thread
-----
```

显示系统中信箱状态：

字段	描述
mailbox	信箱的名称；
entry	信箱中包含的信件数目；
size	信箱能够容纳的最大信件数目；
suspend thread	等这个信箱上的线程数目；

```
finsh>>list_mq()

msgqueue  entry  suspend thread
-----  -

```

显示系统中消息队列状态：

字段	描述
msgqueue	消息队列的名称；
entry	消息队列当前包含的消息数目；
suspend thread	等待这个消息队列上的线程数目；

```
finsh>>list_memp()

mempool  block  total free  suspend thread
-----  -

```

显示系统中内存池状态：

字段	描述
mempool	内存池的名称；
block	内存池中内存块大小；
total	内存块总数量；
free	空余内存块数量；
suspend thread	挂起线程数目；

```
finsh>>list_timer()

timer  periodic  timeout  flag
-----
tidle  0x00000000  0x00000000  deactivated
tshell 0x00000000  0x00000000  deactivated
current tick:0x0000d7e
```

显示系统中定时器状态：

字段	描述
----	----

timer	定时器的名称；
periodic	定时器是否是周期性的；
timeout	定时器超时时的节拍数；
flag	定时器的状态，activated表示活动的，deactivated表示不活动的；
current tick	当前系统的节拍；

```
finsh>> list_device()
```

```
device      type
-----
uart3      Character Device
uart2      Character Device
uart1      Character Device
```

显示系统中设备状态：

字段	描述
device	设备的名称；
type	设备的类型；

type输出下列数据类型：

```
char * const device_type_str[]={
    "Character Device",
    "Block Device",
    "Network Interface",
    "MTD Device",
    "CAN Device",
    "RTC",
    "Sound Device",
    "Graphic Device",
    "I2C Bus",
    "USB Slave Device",
    "USB Host Bus",
    "SPI Bus",
    "SPI Device",
    "SDIO Bus",
    "PM Pseudo Device",
    "Unknown",
};
```

RT-Thread的各个组件会向finsh输出一些命令。如当打开DFS组件时，还会增加如下命令，各个命令详细介绍参见文件系统一章。

```
mkfs      -- make a file system
df        -- get disk free
ls        -- list directory contents
rm        -- remove files or directories
cat       -- print file
copy      -- copy source file to destination file
mkdir     -- create a directory
```

11.8.2 finsh(msh) 内置命令

msh模式下，内置命令风格与bash类似，按下tab键后可以列出当前支持的所有命令。

```
RT-Thread shell commands:
list_timer    - list timer in system
list_device   - list device in system
version       - show RT-Thread version information
list_thread   - list thread
list_sem      - list semaphore in system
list_event    - list event in system
list_mutex    - list mutex in system
list_mailbox  - list mail box in system
list_msgqueue - list message queue in system
ls            - List information about the FILEs.
cp            - Copy SOURCE to DEST.
mv            - Rename SOURCE to DEST.
cat           - Concatenate FILE(s)
rm            - Remove (unlink) the FILE(s).
cd            - Change the shell working directory.
pwd           - Print the name of the current working directory.
mkdir         - Create the DIRECTORY.
ps            - List threads in the system.
time          - Execute command with time.
free          - Show the memory usage in the system.
exit          - return to RT-Thread shell mode.
help          - RT-Thread shell help.
```

执行方式与传统shell相同，因此不详细赘述，以cat为例简单介绍。如果打开DFS，并正确挂载了文件系统，则可以执行ls查看列出的当前目录。

```
finsh>ls
Directory /:
..          <DIR>
a.txt       1119
```

当前目录下存在名为a.txt的文件，则可执行如下命令打印a.txt的内容。

```
finsh>> cat a.txt
```

11.9 移植

finsh完全采用ANSI C编写，具备极好的移植性；内存占用少，如果不使用前面章节中介绍的函数方式动态地向finsh添加符号，finsh将不会动态申请内存。finsh源码位于components/finsh目录下。移植finsh需要注意以下几个方面：

- 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。当定义RT_USING_FINSH宏时，就可以在初始化线程中调用finsh_system_init()初始化finsh shell线程。RT-Thread 1.2.0之后的版本中可以不使用finsh_set_device(const char* device_name)函数去显式指定使用的设备，而是会自动调用rt_console_get_device()函数去使用console设备（RT-Thread 1.1.x及以下版本中必须使用finsh_set_device(const char* device_name)指定finsh shell使用的设备）。finsh shell线程在函数finsh_system_init()函数中被创建，它将一直等待rx_sem信号量。

- finsh的输出：

finsh的输出依赖于系统的输出，在RT-Thread中依赖rt_kprintf输出。在启动函数rt_hw_board_init()中，rt_console_set_device(const char* name)函数设置了finsh的打印输出设备。

- finsh的输入：

finsh shell线程在获得了rx_sem信号量后，调用rt_device_read()函数从设备(选用串口设备)中获得一个字符然后处理。所以finsh的移植需要rt_device_read()函数的实现。而rx_sem信号量的释放通过调用rx_indicate()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是（即串口有输入），接受中断服务例程调用rx_indicate()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。

11.10 宏选项

finsh有一些宏定义可以简单配置。

```
#define RT_USING_FINSH
```

此宏定义在rtconfig.h中，用于在RT-Thread中打开finsh，并将其作为shell。

```
#define FINSH_USING_SYMTAB
```

```
#define FINSH_USING_DESCRIPTION
```

此宏定义在rtconfig.h中。打开FINSH_USING_SYMTAB可以在finsh中使用符号表，打开FINSH_USING_DESCRIPTION需要给每个finsh的符号添加一段字符串描述。这两个宏一般都需要打开。

```
#define FINSH_USING_HISTORY
```

此宏定义在rtconfig.h中，打开后可以在finsh中使用方向键（上下）回溯历史指令。

```
#define FINSH_USING_MSH
```

此宏定义在rtconfig.h中，打开后finsh将支持传统shell模式。

```
#define FINSH_USING_MSH_ONLY
```

此宏定义在rtconfig.h中，打开后finsh仅支持msh模式。

如果打开了FINSH_USING_MSH而没有打开FINSH_USING_MSH_ONLY，finsh同时支持两种c-style模式与msh模式，但是默认进入c-style模式，执行 msh()即可切换到msh模式，在msh模式下执行 exit后即退回到c-style模式。

```
#define DFS_USING_WORKDIR
```

此宏定义在rtconfig.h中，它实际上是DFS组件的宏，但由于它与finsh有一定关系，因此在这里也介绍一下。打开此宏后finsh可以支持工作目录。当使用msh时，建议打开此宏。

```
#define FINSH_USING_AUTH
```

此宏定义在rtconfig.h中，打开则开启权限验证功能。系统在启动后，只有权限验证（目前仅支持密码验证）通过，才会开启finsh功能，提升系统输入的安全性。

```
#define FINSH_DEFAULT_PASSWORD "rtthread"
```

此宏定义在rtconfig.h中，设置finsh在密码验证模式下的默认密码。密码长度大于等于FINSH_PASSWORD_MIN（默认6），小于等于FINSH_PASSWORD_MAX（默认RT_NAME_MAX）。

第 12 章

文件系统

12.1 简介

RT-Thread 的文件系统采用了三层的结构，如图 文件系统结构图 所示：

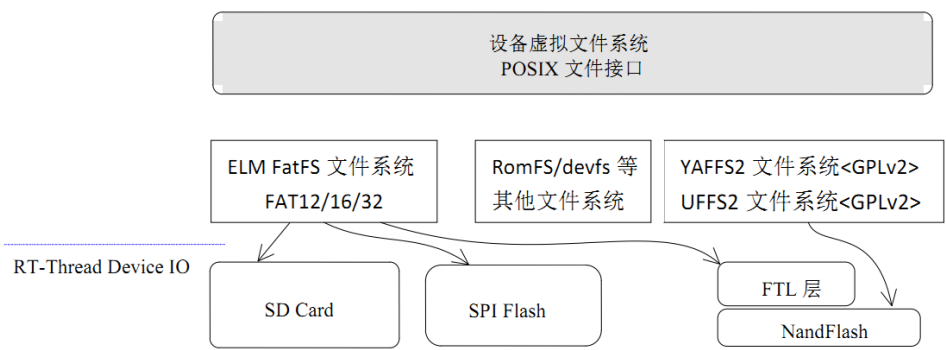


图 12.1: 文件系统结构图

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，RT-thread 操作系统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见的flash 文件系统（YAFFS2、JFFS2 等）。

接下来中间的一层是各种文件系统的实现，例如支持FAT文件系统的DFS-ELM、支持NandFlash 的YAFFS2，只读文件系统ROMFS 等。（RT-Thread 1.0.0版本中包含了ELM FatFS，ROMFS以及网络文件系统NFS v3实现，YAFFS2等flash 文件系统则包含在了RT-Thread 1.1.0 版本中）

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 1.1.0 版本也将在NandFlash 上构建一层转换层(FTL)，以使得NandFlash 能够支持Flash 文件系统。

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，这样也能够保证程序可以在PC 上编写、调试，然后再移植到RT-Thread 操作系统上。

12.2 文件系统、文件与文件夹

文件系统是一套实现了数据的存储、分级组织、访问和获取等操作的抽象数据类型 (Abstract data type)，是一种用于向用户提供底层数据访问的机制。文件系统通常存储的基本单位是文件，即数据是按照一个个文件的方式进行组织。当文件比较多时，将导致文件繁

多，不易分类、重名的问题。而文件夹作为一个容纳多个文件的容器而存在。

在 RT-Thread 中，文件系统名称使用上类似UNIX 文件、文件夹的风格，例如如图 目录结构 的目录结构：

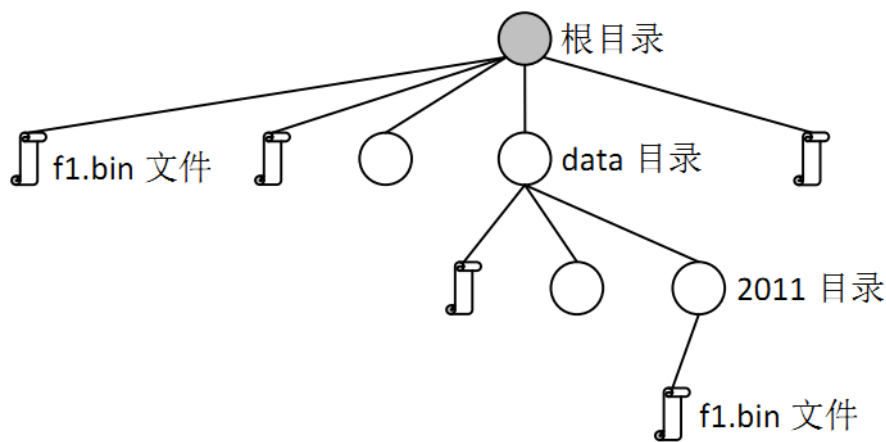


图 12.2: 目录结构

在RT-Thread 操作系统中，文件系统有统一的根目录，使用’ / ’ 来表示。而在根目录下的f1.bin 文件则使用’ /f1.bin’ 来表示，2011 目录下的f1.bin目录则使用’ /data/2011/f1.bin’ 来表示。即目录的分割符号是’ / ’，这与UNIX/Linux 完全相同的，与Windows 则不相同（Windows 操作系统上使用” 来作为目录的分割符）。

默认情况下，RT-Thread 操作系统为了获得较小的内存占用，宏定义DFS_USING_WORKDIR 并不会被定义。当它不定义时，那么在使用文件、目录 接口进行操作时应该使用绝对目录进行（因为此时系统中不存在当前工作的目录）。如果需要使用当前工作目录以及相对目录，可以在rtconfig.h头文件中定义DFS_USING_WORKDIR 宏。

12.3 文件系统接口

12.3.1 打开文件

打开或创建一个文件可以调用下面的open 函数接口：

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

参数：

- pathname - 打开或创建的文件名；
- oflag - 指定打开文件的方式，当前支持的打开方式有：

参数	描述
O_RDONLY	只读方式打开文件
O_WRONLY	只写方式打开文件

O_RDWR	以读写方式打开文件
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，也就是所写入的数据会以附加的方式添加到文件的尾部。

- mode - 与POSIX 标准接口像兼容的参数（目前没有意义，传入0即可）。
- 返回值：

打开成功时返回打开文件的描述符序号，否则返回负数。可以参考 @@以下@@ 代码，看看如何去打开一个文件。

```
#include <rtthread.h>
#include <dfs_posix.h> /* 当需要使用文件操作时，需要包含这个头文件 */

/* 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);
    }

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

12.3.2 关闭文件

当使用完文件后若不再需要使用则可使用close()函数接口关闭该文件，而close()会让数据写回磁盘，并释放该文件所占用的资源。关闭文件的函数接口如下：

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

- 参数：
fd - open()函数所返回的文件描述字。
- 返回值： 无

12.3.3 读取数据

读取数据可使用下面的函数接口：

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

- 参数：
fd - 文件描述词； buf - 内存指针； count - 预读取文件的字节数。
- 返回值：

实际读取到的字节数。read()函数接口会把参数fd 所指的文件的count 个字节传送到buf 指针所指的内存中。返回值为实际读取到的字节数，有两种情况会返回0 值，一是读取数据已到达文件结尾，二是无可读取的数据（例如设定count为0），此外，文件的读写位置会随读取到的字节移动。

12.3.4 写入数据

写入数据可使用下面的函数接口：

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

- 参数：
fd - 文件描述词； buf - 内存指针； count - 预写入文件的字节数。
- 返回值： 实际写入的字节数。

write()函数接口会把buf 指针所指向的内存中count 个字节写入到参数fd 所指的文件内。返回值为实际写入文件的字节数，返回值为0 时表示写入出错，错误代码存入当前线程的errno中，此外，文件的读写位置会写入的字节移动。

可以参考 @@以下@@ 代码，看看一个完整的文件读写流程：

```
/*  
 * 代码清单：文件读写例子  
 * 这个例子演示了如何读写一个文件，特别是写的时候应该如何操作。  
 */  
  
#include <rtthread.h>  
#include <dfs_posix.h> /* 当需要使用文件操作时，需要包含这个头文件 */  
  
#define TEST_FN    "/test.dat"
```

```
/* 测试用的数据和缓冲 */
static char test_data[120], buffer[120];

/* 文件读写测试 */
void readwrite(const char* filename)
{
    int fd;
    int index, length;

    /* 只写 & 创建 打开 */
    fd = open(TEST_FN, O_WRONLY | O_CREAT | O_TRUNC, 0);
    if (fd < 0)
    {
        rt_kprintf("open file for write failed\n");
        return;
    }

    /* 准备写入数据 */
    for (index = 0; index < sizeof(test_data); index++)
    {
        test_data[index] = index + 27;
    }

    /* 写入数据 */
    length = write(fd, test_data, sizeof(test_data));
    if (length != sizeof(test_data))
    {
        rt_kprintf("write data failed\n");
        close(fd);
        return;
    }

    /* 关闭文件 */
    close(fd);

    /* 只写并在末尾添加打开 */
    fd = open(TEST_FN, O_WRONLY | O_CREAT | O_APPEND, 0);
    if (fd < 0)
    {
        rt_kprintf("open file for append write failed\n");
        return;
    }

    length = write(fd, test_data, sizeof(test_data));
}
```

```
if (length != sizeof(test_data))
{
    rt_kprintf("append write data failed\n");
    close(fd);
    return;
}
/* 关闭文件 */
close(fd);

/* 只读打开进行数据校验 */
fd = open(TEST_FN, O_RDONLY, 0);
if (fd < 0)
{
    rt_kprintf("check: open file for read failed\n");
    return;
}

/* 读取数据(应该为第一次写入的数据) */
length = read(fd, buffer, sizeof(buffer));
if (length != sizeof(buffer))
{
    rt_kprintf("check: read file failed\n");
    close(fd);
    return;
}

/* 检查数据是否正确 */
for (index = 0; index < sizeof(test_data); index++)
{
    if (test_data[index] != buffer[index])
    {
        rt_kprintf("check: check data failed at %d\n", index);
        close(fd);
        return;
    }
}

/* 读取数据(应该为第二次写入的数据) */
length = read(fd, buffer, sizeof(buffer));
if (length != sizeof(buffer))
{
    rt_kprintf("check: read file failed\n");
    close(fd);
    return;
}
```

```

/* 检查数据是否正确 */
for (index = 0; index < sizeof(test_data); index++)
{
    if (test_data[index] != buffer[index])
    {
        rt_kprintf("check: check data failed at %d\n", index);
        close(fd);
        return;
    }
}

/* 检查数据完毕, 关闭文件 */
close(fd);
/* 打印结果 */
rt_kprintf("read/write done.\n");
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出函数到finsh shell 命令行中 */
FINSH_FUNCTION_EXPORT(readwrite, perform file read and write test);
#endif

```

12.3.5 更改名称

更改文件的名称可使用下面的函数接口：

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

- 参数： oldpath - 需更改的文件名； newpath - 更改成的文件名。
- 返回值： 无

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。若 newpath 所指定的文件已经存在，则该文件将会被覆盖。可以参考 @@以下@@ 代码，如何进行文件名改名。

```

#include <dfs_posix.h>

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");
}

```

这个示例函数会把文件’ /text1.txt’ 改名成’ /text2.txt’ 。

12.3.6 取得状态

获取文件状态可使用下面的stat 函数接口：

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

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

- 参数： file_name - 文件名； buf - 结构指针，指向获取文件状态的结构。

返回值： 无 可以参考 @@以下@@ 代码了解如何使用stat 函数。

```

void file_thread(void* parameter)
{
    struct stat buf;
    stat("/text.txt", &buf);
    rt_kprintf("text.txt file size = %d\n", buf.st_size);
}

```

12.4 目录操作接口

12.4.1 创建目录

创建目录可使用下面的函数接口：

```
int mkdir(const char *path, mode_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。

- 参数： path - 目录名； mode - 创建模式。
- 返回值： 创建成功返回0，创建失败返回-1。

可以参考 @@以下@@ 代码了解如何使用mkdir 函数：

```

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");
}
}

```

12.4.2 打开目录

打开目录可使用下面的函数接口：

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

opendir()函数用来打开一个目录，参数name 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。

- 参数： name - 目录路径名。
- 返回值： 打开文件成功，返回指向目录的DIR 结构指针，否则返回RT_NULL。

可以参考 @@以下@@ 代码了解如何使用opendir()函数：

```

#include <dfs_posix.h>

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

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

```

```

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

```

12.4.3 读取目录

读取目录可使用下面的函数接口：

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

readdir()函数用来读取目录，参数d 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，每读取一次目录，目录流的指针位置将自动往后递推1 个位置。

- 参数：

d - 目录路径名。

- 返回值：

读取成功返回指向目录entry 的结构指针，否则返回RT_NULL。

可以参考 @@以下@@ 代码了解如何使用readdir 函数：

```

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

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

```


12.4.4 取得目录流的读取位置

获取目录流的读取位置可使用下面的函数接口：

```
off_t telldir(DIR *d);
```

- 参数：
d - 目录路径名。
- 返回值：
无

12.4.5 设置下次读取目录的位置

设置下次读取目录的位置可使用下面的函数接口：

```
void seekdir(DIR *d, off_t offset);
```

- 参数：
d - 目录路径名； offset - 偏移值，距离本次目录的位移。
- 返回值：
无 可以参考 @@以下@@代码了解如何使用seekdir 函数：

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

    /* 打开根目录 */
    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);
}

```

12.4.6 重设读取目录的位置为开头位置

重设读取目录为开头位置可使用下面的函数接口：

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

- 参数：
d - 目录路径名；
- 返回值：
无

12.4.7 关闭目录

关闭目录可使用下面的函数接口：

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

closedir()函数用来关闭一个目录。该函数必须和opendir()函数成对出现。

- 参数：
d - 目录路径名；
- 返回值：
关闭成功返回0，否则返回-1；

12.4.8 删除目录

删除目录可使用下面的函数接口：

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

- 参数：
d - 目录路径名；
- 返回值：
删除目录成功返回0，否则返回-1。

12.4.9 格式化文件系统

```
int mkfs(const char * fs_name, const char * device)
```

- 参数：
fs_name - 文件系统名； device - 设备名；
- 返回值：
格式化成功返回0，否则返回-1。

RT-Thread中目前支持的文件系统参见本章最后一节。

12.5 底层驱动接口

RT-Thread DFS 文件系统针对下层媒介使用的是RT-Thread 的设备IO系统，其中主要包括设备读写等操作。但是有些文件系统并不依赖于RT-Thread 的设备系统，例如1.0.x分支引入的只读文件系统、网络文件系统等。对于常使用的FAT 文件系统，下层驱动必须用块设备的形式来实现。

12.5.1 文件系统初始化

在使用文件系统接口前，需要对文件系统进行初始化，代码如下：

```
#ifdef RT_USING_DFS
/* 包含DFS 的头文件 */
#include <dfs_fs.h>
#include <dfs_elm.h>
#endif

/* 初始化线程 */
void rt_init_thread_entry(void *parameter)
{
```

```

/* 文件系统初始化 */
#ifdef RT_USING_DFS
{
/* 初始化设备文件系统 */
dfs_init();
#ifdef RT_USING_DFS_ELMFAT
/* 如果使用的是ELM 的FAT 文件系统，需要对它进行初始化 */
elm_init();

/* 调用dfs_mount 函数对设备进行装载 */
if (dfs_mount("sd0", "/", "elm", 0, 0) == 0)
    rt_kprintf("File System initialized!\n");
else
    rt_kprintf("File System init failed!\n");
#endif
}
#endif
}

```

其主要包括的函数接口为：

```

int dfs_mount(const char* device_name, const char* path, const char* filesystemtype,
rt_uint32_t rwflag, const void* data);

```

dfs_mount 函数用于把以device_name 为名称的设备挂接到path 路径中。filesystemtype 指定了设备上的文件系统的类型（如上面代码所述的elm、rom、nfs 等文件系统）。data参数对某些文件系统是有意义的，如nfs，对elm 类型系统则没有意义。

- 参数：

device_name - 设备名； path - 挂接路径； filesystemtype - 文件系统的类型； rwflag - 文件系统的标志； data - 文件系统的数据。

- 返回值：

装载成功将返回0，否则返回-1。具体的错误需要查看errno。

12.6 FatFs

FatFs是专为小型嵌入式设备开发的一个兼容微软fat的文件系统，采用ANSI C编写，采用抽象的硬件I/O层以及提供持续的维护，因此具有良好的硬件无 关性以及可移植性。

FatFs官方网址 http://elm-chan.org/fsw/ff/00index_e.html

RT-Thread将FatFs整合为一个RT-Thread组件，并置于DFS层之下。因此可以 非常方便的在RT-Thread中使用FatFs。

12.6.1 FatFs 相关宏

在RT-Thread中使用Elm FatFs，需要在rtconfig.h打开此宏。

```
/* DFS: ELM FATFS options */
#define RT_USING_DFS_ELMFAT
```

FAT文件系统扇区大小

```
/* Maximum sector size to be handled. */
#define RT_DFS_ELM_MAX_SECTOR_SIZE 512
```

这个宏用于指定FatFs的内部扇区大小，注意，这个宏需要大于等于实际硬件驱动的扇区大小。例如，某spi flash芯片扇区为4096字节，则上述宏需要修改为4096，否则FatFs从驱动读入数据时就会发生数组越界而导致系统崩溃（新版本在系统执行时给出警告信息）。

```
/* Number of volumes (logical drives) to be used. */
#define RT_DFS_ELM_DRIVES 2
```

FatFs支持多分区，默认支持一个分区，如果想要在多个设备上挂载FatFs，可以修改上述宏定义。

```
/* Reentrancy (thread safe) of the FatFs module. */
#define RT_DFS_ELM_REENTRANT
```

Elm FatFs充分考虑了多线程安全读写安全的情况，当在多线程中读写FatFs时，为了避免重入带来的问题，需要打开上述宏。如果系统仅有一个线程操作文件系统，不会出现重入问题，则可以关闭上述宏以节省资源。

```
#define RT_DFS_ELM_USE_LFN 3
#define RT_DFS_ELM_MAX_LFN 255
#define RT_DFS_ELM_CODE_PAGE 437
```

默认情况下，FatFs使用8.3方式的文件命名规则，这种方式具有如下缺点：

- 文件名（不含后缀）最长不超过8个字符，后缀最长不超过3个字符。文件名和后缀超过限制后将会被截断。
- 文件名不支持大小写（显示为大写）

如果需要在支持长文件名，则需要打开上述宏。注意，Elm FatFs支持三种方式的长文件名

- 1 采用静态缓冲区支持长文件名，多线程操作文件名时将会带来重入问题。
- 2 采用栈内临时缓冲区支持长文件名。对栈空间需求较大。
- 3 使用heap（malloc申请）缓冲区存放长文件名。最安全。

在RT-Thread中，如果对需要使用长文件名，建议使用长文件名模式3，即按照如下方式定义

```
#define RT_DFS_ELM_USE_LFN 3
```

当打开长文件名支持时，FatFs内部会使用Unicode编码文件名，而完整Unicode字库较大，不利于嵌入式设备上使用，FatFs可以单独配置文件名编码，在rtconfig.h指定宏RT_DFS_ELM_CODE_PAGE的值可以配置FatFs的编码。如果需要存储中文文件名，可以使用936编码（GBK编码），如下所示

```
#define RT_DFS_ELM_CODE_PAGE 936
```

当打开长文件名宏RT_DFS_ELM_USE_LFN时，RT-Thread/FatFs默认使用936编码。936编码需要一个大约180KB的字库。如果仅使用英文字符作为文件，则可以设置宏为437（美国英语），这样就可以节省这180KB的Flash空间。

FatFs所支持的文件编码如下所示。

```
/* The _CODE_PAGE specifies the OEM code page to be used on the target system.
 / Incorrect setting of the code page can cause a file open failure.
 /
 / 932 - Japanese Shift-JIS (DBCS, OEM, Windows)
 / 936 - Simplified Chinese GBK (DBCS, OEM, Windows)
 / 949 - Korean (DBCS, OEM, Windows)
 / 950 - Traditional Chinese Big5 (DBCS, OEM, Windows)
 / 1250 - Central Europe (Windows)
 / 1251 - Cyrillic (Windows)
 / 1252 - Latin 1 (Windows)
 / 1253 - Greek (Windows)
 / 1254 - Turkish (Windows)
 / 1255 - Hebrew (Windows)
 / 1256 - Arabic (Windows)
 / 1257 - Baltic (Windows)
 / 1258 - Vietnam (OEM, Windows)
 / 437 - U.S. (OEM)
 / 720 - Arabic (OEM)
 / 737 - Greek (OEM)
 / 775 - Baltic (OEM)
 / 850 - Multilingual Latin 1 (OEM)
 / 858 - Multilingual Latin 1 + Euro (OEM)
 / 852 - Latin 2 (OEM)
 / 855 - Cyrillic (OEM)
 / 866 - Russian (OEM)
 / 857 - Turkish (OEM)
 / 862 - Hebrew (OEM)
 / 874 - Thai (OEM, Windows)
```

12.7 NFS

NFS是Network File System（网络文件系统）的简称。NFS允许一个系统在网络上与他人共享目录和文件。通过NFS，用户和程序可以像访问本地文件一样访问远端系统上的文件。NFS可以加速程序的开发调试，在嵌入式系统中应用十分广泛。

NFS主要由部分组成：一台服务器和一台或多台客户机。客户机远程访问存放在服务器上的资源。

PC上运行NFS server程序，运行RT-Thread的嵌入式系统做NFS Client，可以将PC上的目录通过以太网挂载到RT-Thread中。

12.7.1 RT-Thread中使用NFS

NFS通过以太网实现数据传输，因此要想在RT-Thread中使用NFS，需要一些基本条件：

- 开发板硬件带有以太网支持
- 以太网驱动测试通过，可以在RT-Thread中正确运行Lwip

当具备这两个条件，我们就可以在RT-Thread上使用NFS了。

主机配置

Windows上可以使用FreeNFS搭建一个简单的NFS Server。[下载FreeNFS](#)。

双击即可运行。运行以后，程序会在系统托盘中，右击选择setting，可以主机的目录（如下）

```
C:\Users\Administrator\Documents\FreeNFS
```

被作为共享目录，读者可以根据需要修改此路径。

开发板配置

需要在rtconfig.h中开DFS以及LWIP，并根据开发板的网络环境修改网络配置，并确保PC机与开发板可以正常ping通。

```
#define RT_USING_LWIP

#define RT_LWIP_IPADDR0 192
#define RT_LWIP_IPADDR1 168
#define RT_LWIP_IPADDR2 1
#define RT_LWIP_IPADDR3 30

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

/* mask address of target*/
#define RT_LWIP_MSKADDR0 255
```

```
#define RT_LWIP_MSKADDR1 255
#define RT_LWIP_MSKADDR2 255
#define RT_LWIP_MSKADDR3 0
#define RT_USING_DFS_NFS
```

NFS属于DFS组件，因此同样需要打开DFS宏，以及DFS_NFS宏，并配置RT_NFS_HOST_EXPORT宏。

```
#define RT_USING_DFS
#define RT_USING_DFS_NFS
#define RT_NFS_HOST_EXPORT "192.168.1.2:/"
```

RT_NFS_HOST_EXPORT宏用于指定NFS Server的ip地址以及共享目录路径。

如果读者使用的bsp中的rtconfig.h中没有上述宏，则需要自己添加上。

NFS相关的启动代码，如下所示（以stmf10x作为参考平台）。

```
/* LwIP Initialization */
#ifdef RT_USING_LWIP
{
    extern void lwip_sys_init(void);

    /* register ethernetif device */
    eth_system_device_init();

#ifdef STM32F10X_CL
    rt_hw_stm32_eth_init();
#else
    /* STM32F103 */
    #if STM32_ETH_IF == 0
        rt_hw_enc28j60_init();
    #elif STM32_ETH_IF == 1
        rt_hw_dm9000_init();
    #endif
#endif

    /* re-init device driver */
    rt_device_init_all();

    /* init lwip system */
    lwip_sys_init();
    rt_kprintf("TCP/IP initialized!\n");
}
#endif

#if defined(RT_USING_DFS) && defined(RT_USING_LWIP) && defined(RT_USING_DFS_NFS)
{
```



```

/* NFSv3 Initialization */
rt_kprintf("begin init NFSv3 File System ...\n");
nfs_init();

if (dfs_mount(RT_NULL, "/", "nfs", 0, RT_NFS_HOST_EXPORT) == 0)
    rt_kprintf("NFSv3 File System initialized!\n");
else
    rt_kprintf("NFSv3 File System initialization failed!\n");
}
#endif

```

上述代码可以分为两部分，第一部分是初始化lwip组件。第二部分包括NFS初始化，以及挂在NFS Server上的共享目录。示例代码中将NFS挂载到了系统根目录下，读者也可以选择其他文件系统挂在根目录，并将NFS挂载到其他子路径下。

编译烧录程序后，就可以在finsh中使用ls、mkdir、cat等命令操作NFS共享的目录中的文件了。

12.8 UFFS

UFFS是Ultra-low-cost Flash File System（超低功耗的闪存文件系统）的简称。它是国人开发的、专为嵌入式设备等小内存环境中使用Nand Flash的开源文件系统。与嵌入式中常用的yaffas文件系统相比具有资源占用少、启动速度快、免费等优势。

UFFS官方代码仓库 <http://sourceforge.net/projects/uffs/>

12.8.1 UFFS配置

首先来介绍rtconfig.h中的UFFS中的相关宏。

```

#define RT_USING_MTD_NAND
#define RT_USING_DFS
#define RT_USING_DFS_UFFS

```

在RT-Thread中的UFFS使用了MTD NAND的接口，因此需要打开RT_USING_MTD_NAND。此外，要想正确使用UFFS还必须提供NAND的驱动程序，它需要符合RT-Thread的MTD NAND接口规范。该驱动程序的实现将在后面的章节介绍。后面两个宏必须打开。

更多配置参考dfs_uffs.h与uffs_config.h

UFFS配置相关宏。在nand flash芯片上通常使用ECC进行数据校验（ECC是一种数据校验与纠错机制）。UFFS支持多种校验方式，包括如下几种：

1. UFFS_ECC_SOFT
2. UFFS_ECC_HW_AUTO
3. UFFS_ECC_NONE

方式一为软件校验方式，主要用于一些不支持硬件ECC的情况下，ECC校验由UFFS完成。由于ECC数据校验比较耗时，因此这种方式会导致读写速度降低，不推荐使用。

方式二为硬件自动方式。这种方式下，ECC校验由NAND驱动程序完成，UFFS不做任何ECC校验工作。这种方式比较灵活，驱动程序可以自行决定ECC数据的存放位置。

方式三为无ECC校验方式。在这种方式下，UFFS不使用ECC校验，由于NAND芯片可能出现数据写入错误，并且ECC可以识别并纠正一定bit的错误（一般ECC可以纠正一个bit的错误，可以识别2个bit的错误但是无法纠正，但这并不绝对，ECC bits越多其纠错能力越强）。在NAND设备上通常会有一定的安全风险。

综上，当在NAND设备上使用UFFS时推荐使用方式二UFFS_ECC_HW_AUTO。

注意：UFFS不仅可以使用在NAND设备上，也可以使用NOR FLASH、SPI FLASH设备等。不过目前RT-Thread中的UFFS仅支持在NAND上使用，未来可能会考虑增加对NOR FLASH以及SPI FLASH的支持。

```
#define RT_CONFIG_UFFS_ECC_MODE UFFS_ECC_HW_AUTO
```

rtconfig.h中定义，用于配置UFFS的校验方式。

```
#define RT_UFFS_USE_CHECK_MARK_FUNCITON
```

rtconfig.h中定义。NAND容易产生坏块，一般NAND文件系统(如yaffs)都需要提供检测坏块和标记坏块的功能。为了简化UFFS驱动编写，UFFS提供了上面这个宏。当打开这个宏时，NAND驱动需要提供坏块检测与坏块标记这两个函数。如果关闭这个宏，UFFS将会借助NAND驱动提供的页读写函数实现坏块检查与标记。

12.8.2 UFFS 内存精简

UFFS本身支持非常多的配置选项，配置非常灵活。在下面文件中有大量的配置选项。可以修改这个文件来定制UFFS实现精简内存占用。

```
components/dfs/filesystems/uffs/uffs_config.h
```

此文件配置选项中多，内存占用较大的几个宏如下所示。

```
/**
 * \def MAX_CACHED_BLOCK_INFO
 * \note uffs cache the block info for opened directories and files,
 *       a practical value is 5 ~ MAX_OBJECT_HANDLE
 */
#define MAX_CACHED_BLOCK_INFO 6//50

/**
 * \def MAX_PAGE_BUFFERS
 * \note the bigger value will bring better read/write performance.
```

```

*      but few writing performance will be improved when this
*      value is become larger than 'max pages per block'
*/
#define MAX_PAGE_BUFFERS 10//40

/**
 * \def MAX_DIRTY_PAGES_IN_A_BLOCK
 * \note this value should be between '2' and the lesser of
 * 'max pages per block' and (MAX_PAGE_BUFFERS - CLONE_BUFFERS_THRESHOLD - 1).
 *
 *      the smaller the value the frequently the buffer will be flushed.
 */
#define MAX_DIRTY_PAGES_IN_A_BLOCK 7//32

/**
 * \def MAX_OBJECT_HANDLE
 * maximum number of object handle
 */
#define MAX_OBJECT_HANDLE 8//50

```

按照上面修改后，可以显著降低内存占用。注意，这样可能会降低UFFS的读写性能。究竟该配置什么样的参数，还需要读者根据自己板子的实际情况配置并测试，合理配置参数才能找到最理想的配置方案。

12.8.3 MTD NAND驱动

TOADD: NAND结构简介

在RT-Thread上使用UFFS，还需要提供NAND驱动。RT-Thread针对NAND芯片设计了一层简单的MTD NAND接口层。MTD NAND接口对NAND芯片做了简单的抽象和封装，为一个NAND芯片编写符合MTD接口的程序后就可以在NAND上使用RT-Thread支持的NAND组件，如UFFS、Yaffs以及NFTL等。

NFTL即Nand Flash Translate Layer，利用它就在NAND上安全的使用FatFs文件系统。这个组件目前仅面向商业客户提供。关于NFTL的相关信息，请参考RT-Thread商业支持网站<http://www.rt-thread.com/>

MTD NAND接口源码位于components/drivers/mtd目录下，其中最重要的数据结构包括两个。

1. struct rt_mtd_nand_device

```

struct rt_mtd_nand_device
{
    struct rt_device parent;

    rt_uint16_t page_size; /* The Page size in the flash */
    rt_uint16_t oob_size; /* Out of bank size */
    rt_uint16_t oob_free; /* the free area in oob that flash driver not use */
    rt_uint16_t plane_num; /* the number of plane in the NAND Flash */

    rt_uint32_t pages_per_block; /* The number of page a block */
    rt_uint16_t block_total;

```

```
rt_uint32_t block_start; /* The start of available block */
rt_uint32_t block_end; /* The end of available block */

/* operations interface */
const struct rt_mtd_nand_driver_ops* ops;
};
```

- page_size 页大小，指页数据区字节数目
- oob_size 页spare区（或称为oob区）字节大小
- oob_free 表示页spare区中可能空间大小，MTD驱动通常会将ECC数据校验以及坏块标志放在Spare区，oob_free指除这些数据之外的spare区的剩余空间大小。
- plane_num NAND flash的plane数目
- pages_per_block 每个NAND FLASH块的页数。
- block_total 块数目
- block_start 起始块号
- block_end 结束块号
- ops 用来填充MTD NAND的操作函数名

2. struct rt_mtd_nand_driver_ops

```
struct rt_mtd_nand_driver_ops
{
    rt_err_t (*read_id)(struct rt_mtd_nand_device* device);

    rt_err_t (*read_page)(struct rt_mtd_nand_device* device,
                          rt_off_t page,
                          rt_uint8_t* data, rt_uint32_t data_len,
                          rt_uint8_t * spare, rt_uint32_t spare_len);

    rt_err_t (*write_page)(struct rt_mtd_nand_device * device,
                          rt_off_t page,
                          const rt_uint8_t * data, rt_uint32_t data_len,
                          const rt_uint8_t * spare, rt_uint32_t spare_len);
    rt_err_t (*move_page)(struct rt_mtd_nand_device *device, rt_off_t src_page, rt_off_t dst_page);

    rt_err_t (*erase_block)(struct rt_mtd_nand_device* device, rt_uint32_t block);
    rt_err_t (*check_block)(struct rt_mtd_nand_device* device, rt_uint32_t block);
    rt_err_t (*mark_badblock)(struct rt_mtd_nand_device* device, rt_uint32_t block);
};
```

这是MTD NAND定义的一组用于操作NAND FLASH的方法。接下来分别介绍各个函数的作用。

readid用于返回MTD NAND设备的id。

读写页

```
rt_err_t (*read_page)(struct rt_mtd_nand_device* device,
                      rt_off_t page,
                      rt_uint8_t* data, rt_uint32_t data_len,
                      rt_uint8_t * spare, rt_uint32_t spare_len);
rt_err_t (*write_page)(struct rt_mtd_nand_device * device,
                       rt_off_t page,
                       const rt_uint8_t * data, rt_uint32_t data_len,
                       const rt_uint8_t * spare, rt_uint32_t spare_len);
```

• 参数:

- device - 设备指针;
- page - 页号, 此页号为块内页号, 即某页在其块内的页号
- data - 页数据缓冲区地址, 如果不读/写data区则设置为NULL
- data_len - 页数据长度;
- spare - 页SPARE (OOB) 缓冲区地址, 若不读/写SPARE区则设置为NULL
- spare_len- 页SPARE缓冲区长度;

• 返回值:

- RT_EOK - 读写成功
- -RT_MTD_EECC - ECC错误
- -RT_EIO - 参数错误

擦除块

```
rt_err_t (*erase_block)(struct rt_mtd_nand_device* device, rt_uint32_t block);
```

• 参数:

- device - 设备指针;
- block - 块号

• 返回值:

- RT_EOK - 擦除成功

块状态检查

```
rt_err_t (*check_block)(struct rt_mtd_nand_device* device, rt_uint32_t block);
```

用于检查一个块是否为坏块。当使用UFFS并打开宏RT_UFFS_USE_CHECK_MARK_FUNCITON时, 必须实现这个函数。

- 参数：
 - device - 设备指针；
 - block - 块号
- 返回值：
 - RT_EOK - 好块
 - -1 - 坏块

标记坏块

```
rt_err_t (*mark_badblock)(struct rt_mtd_nand_device* device, rt_uint32_t block);
```

用于标记一个块为坏块。当使用UFFS并打开宏RT_UFFS_USE_CHECK_MARK_FUNCITON时，必须实现这个函数。

- 参数：
 - device - 设备指针；
 - block - 块号
- 返回值：
 - RT_EOK - 标记成功
 - 非0值 - 标记失败

移动页

```
rt_err_t (*move_page) (struct rt_mtd_nand_device *device, rt_off_t src_page, rt_off_t dst_page);
```

将NAND FLASH中的一个页移动到另一个页中。NAND FLASH控制器通常硬件命令实现此功能。注意：此函数UFFS与YAFFS并不需要。NFTL需要实现。

- 参数：
 - device - 设备指针；
 - src_page - 块内源页号
 - dst_page - 块内目的页号
- 返回值：
 - RT_EOK - 操作成功
 - -RT_EIO - 参数错误或其硬件错误等

注册MTD NAND设备

```
rt_err_t rt_mtd_nand_register_device(const char* name, struct rt_mtd_nand_device* device);
```

调用此函数向RT-Thread系统注册MTD NAND设备，在UFFS文件系统中就可以使用这个设备挂载文件系统。

12.8.4 UFFS示例驱动

目前RT-Thread中使用UFFS还是比较容易的，stm32f10x，stm32f40x上实现了k9f1g08 NAND的支持，并且在bsp/simulator恶意是用文件来模拟NAND，并支持UFFS模拟。

读者可以参考这些学习MTD NAND驱动的写法。

1. stm32f40x的k9f2g08.c驱动

<https://github.com/RT-Thread/realtouch-stm32f4/blob/master/software/examples/drivers/k9f2g08u0b.c>

2. stm32f10x的k9f1g08.c驱动

https://github.com/prife/stm32f10x_demo/blob/master/wdrivers/k9f_nand.c

12.9 jffs2

12.10 yaffs

第 13 章

lwIP - 轻型TCP/IP协议栈

13.1 简介

lwIP (light-weight IP)最初由瑞典计算机科学院 (Swedish Institute of Computer Science) 的Adam Dunkels开发, 现在由Kieran Mansley领导的一个全球开发团队开发、维护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协议的基础上实现了小型化的资源占用, 因此它十分适合于应用到嵌入式设备中, 其占用的资源体积RAM大概为几十kB, ROM大概为40KB。

lwIP结构精简, 功能完善, 因而用户群较为广泛。RT-Thread实时操作系统就采用了lwIP做为默认的TCP/IP协议栈, 同时根据小型设备的特点对lwIP进行了再次优化, 使其资源占用体积进一步地缩小, RAM 的占用可缩小到5kB附近 (未计算上层应用使用TCP/IP协议时的空间占用量)。本章内容将为您讲述lwIP在RT-Thread中的使用方法。

主要特性 (摘自lwIP官方网站, 翻译如有错误请指正) :

- 协议: IP, ICMP, UDP, TCP, IGMP, ARP, PPPoS, PPPoE
- DHCP client, DNS client, AutoIP/APIPA(Zeroconf), SNMP agent(private MIB support)
- APIs: 专门针对增强性能实现的API接口, 可选的类BSD的Socket API。
- 延伸的特点: 多个网络接口的IP转发, TCP拥塞控制, RTT往返时间估算、快速恢复和快速重传

13.2 协议分层

可能大家对OSI七层模型并不陌生, 它将网络协议很细致地从逻辑上分为了7层。但是实际运用中并不是按七层模型, 一般大家都只使用5层模型。如下: 物理层: 一般包括物理媒介, 电信号, 光信号等, 主要对应于PHY芯片, PHY芯片将数据传送给物理媒介 (RJ45座->双绞线), 如图:

- 数据链路层: 一般简单称为MAC层, 因为MAC芯片处于这层, 对应于代码中的网卡驱动层。数据包在这一层一般称之为“以太网帧”。
- 网络层: 对应于代码中IP层。
- 传输层: 对应于代码中TCP层。
- 应用层: 对应于代码中应用层数据, 即SOCKET通信,recv()/send()的数据。



图 13.1: 物理层

对于一个以太网数据包，我们在代码中能真实看到的包括4部分，分别对应链路层、网络层、传输层、应用层，如下图：



图 13.2: 以太网包格式

注：有几个概念需要解释一下，从网卡收到的数据，此时是一个完整的包含各层头的数据包，此时称之为“以太网帧”；当解开以太网帧头到达IP层，称之为“IP Packet（IP数据包）”；当解开IP头到达TCP层，称之为“TCP Segment（TCP分片）”；当解开TCP头时到达应用层，就是我们socket通信看到的数据了。

这种分层的设计作为一个协议设计与实现的向导，在这种方式下，每个协议可以分离地实现，互不干扰。然而严格的分层设计，各层间的通讯可能会导致总体的性能下降。为了克服这些问题，协议的某些内部细节可以被其他的协议共享，但是必须注意，只有重要的信息才能在各层间共享。

大部分的TCP/IP协议栈实现在应用层到底层之间都遵循严格的分层设计，然而底层或多或少可以有交叉。在大多数操作系统中，所有的底层协议都与操作系统的内核绑定在一起（成为OS内核的一部分），内核提供入口点（API）与应用层的进程通信。此时，应用程序可认为是TCP/IP协议栈的一个抽象，不用关心底层的细节，对于支持SOCKET的系统，直接使用SOCKET进行网络通信即可，这些操作基本和文件IO的操作差别不大。这意味着应用程序对底层一无所知，比如底层使用buffer缓冲数据，而应用层无法对buffer一无所知，如果有应用层有一部分数据频繁使用，而它是无法操作buffer将频繁使用的数据缓冲起来。

在最小系统中，一般不会严格地在内核和应用程序中间加一道保护屏障，如此应用程序可以使用共享内存（底层在内核中，与内核共享内存）的方式更轻松地与底层通信。具体来讲，应用层知道底层使用的缓冲处理机制，因此，应用层可以更高效的重用buffer。既然应用层可以和底层协议使用同一段内存，这样也可以节省拷贝带来的开销。

13.3 lwIP不遵循严格的分层

前面提到过TCP/IP的标准实现一般使用严格的分层，这对lwIP的设计与实现提供了指导意义。每个协议作为一个单独地模块，提供一些API作为协议的入口点。尽管这些协议都单独地实现，但是一些层（协议之间）违背了严格的分层标准，这样做是为了提高处理的速度和内存的占用。比如：在TCP分片的报文中，为了计算TCP校验和，我们需要知道IP协议层

的源IP地址和目的IP地址，一般我们会构造一个伪的IP头（包含IP地址信息等），而常规的做法是通过IP协议层提供的API去获得这些IP地址，但是lwIP是拿到数据报文的IP头，从中解析得到IP地址。

13.4 进程模型 (process model)

以下将process翻译为“进程”只是便于说明问题，在不同的OS中也可能是线程。

TCP/IP的进程模型可以被设计为以下两种：（当然也可以是其他的模型）1. 每层的协议都作为一个独立的进程在运行，这种方式的好处在于代码易于理解和调试；同样也有不好之处，每数据报文经过每一层时，都需要进行一次上下文的切换（因为每层协议都在不同的进程中），在操作系统中，上下文的切换耗费资源比较大。2. lwIP使用单进程模型，所有的协议都运行在一个进程中，并且与操作系统内核是分开的。应用程序可以驻留在lwIP进程中或者运行在一个分离的进程中，当应用程序与lwIP在一个进程中，协议栈和应用层通讯通过函数调用即可。

这两种方法各有优缺点，lwIP之所以运行在一个分离地，单独地进程中，是因为这样易于不同的操作系统移植，为啥？与内核耦合性小。为了方便移植，lwIP加了一层操作系统模拟层，只要实现这里面的提供的函数移植的基本工作就完成了。下一章将介绍操作系统模拟层。

13.5 操作系统模拟层 (OS emulation layer)

不同的操作系统，提供不同的通信机制，而且这些通信的方法实现也不同，增加操作系统模拟层，将操作系统相关的功能函数和数据结构放在这一层中（对应于代码sys.c/h），这一层提供诸如创建lwIP进程，延时，互斥锁，信号量，邮箱等相关的函数。如下：

```
// Creates a new thread
sys_thread_t sys_thread_new(const char *name, lwip_thread_fn thread,
                             void *arg, int stacksize, int prio);

/** Create a new mutex
 * @param mutex pointer to the mutex to create
 * @return a new mutex */
err_t sys_mutex_new(sys_mutex_t *mutex);
/** Delete a semaphore
 * @param mutex the mutex to delete */
void sys_mutex_free(sys_mutex_t *mutex);
#ifdef sys_msleep
void sys_msleep(u32_t ms); /* only has a (close to) 1 jiffy resolution. */
#endif
/* Mailbox functions. */
/** Create a new mbox of specified size
 * @param mbox pointer to the mbox to create
 * @param size (mimum) number of messages in this mbox
 * @return ERR_OK if successful, another err_t otherwise */
err_t sys_mbox_new(sys_mbox_t *mbox, int size);
```

一般说来，移植到其他操作系统上时，实现这些接口即可，但是在实际的移植过程中还需要做一些细节处理。具体的一个移植的实现请看[RT-Thread源码](#)对于sys.c/h如何处理的。

13.6 RT-Thread中的lwIP

由于原版的lwIP更适合于在无操作系统的情况下运行，所以RT-Thread在移植lwIP的过程中根据RT-Thread的特点进行了适当调整。其结构如下图所示：

RT-Thread操作系统中的lwIP是从lwIP发布原始版本移植过来，然后添加了设备层以替换原来的驱动层。不同于原版，这里RT-Thread对于以太网数据的收发采用了独立的双线程（erx线程与etx线程）结构：

- erx线程用于以太网报文的接收—当以太网硬件设备收到网络报文产生中断时，中断服务例程将会通过邮箱的形式唤醒erx线程，让erx线程主动进行以太网报文收取过程，当erx线程收到有效网络报文后，它通过邮箱的形式通知给LwIP的主线程（tcp线程）；
- tcp的发送操作则是通过邮箱的形式唤醒etx线程进行实际的以太网硬件写入。在正常情况下，erx线程和etx线程的优先级是相同的，用户可以根据自身实际要求进行微调以侧重接收或发送。

13.6.1 lwIP版本

RT-Thread lwIP包含三个版本，分别为：“1.3.2”，“1.4.0”，“1.4.1”，其中“1.4.0”的文件夹没有标名版本号，查看具体的版本号可以在src/include/lwip/init.h中查询。如下：

```
/** X.x.x: Major version of the stack */
#define LWIP_VERSION_MAJOR    1U
/** x.X.x: Minor version of the stack */
#define LWIP_VERSION_MINOR    4U
/** x.x.X: Revision of the stack */
#define LWIP_VERSION_REVISION 1U
```

RT-Thread通过宏去指定使用哪个版本的lwIP，熟悉RT-Thread的朋友都知道一般都是使用scons工具（类linux下的make工具）生成项目工程文件（MDK工程、IAR工程等），因此在每个版本的文件夹中包含了一个SConscript文件，该文件中会依赖与相应的宏加入到工程文件中，以lwIP1.4.1中的SConscript为例：

```
group = DefineGroup('LwIP', src, depend = ['RT_USING_LWIP', 'RT_USING_LWIP141'], CPPPATH = path)
```

大家可以看到加入该版本下的所有文件依赖与（RT_USING_LWIP、RT_USING_LWIP141）两个宏，这两个宏在RT-Thread源码的rtconfig.h中，这个文件与实际的项目（或者说BSP、开发板相关），点开“bsp”目录下任何一个文件夹都可以找到rtconfig.h。因为这些宏是自己定义的，所以你可能在这个文件中找不到这些宏，如果你需要使用，请自行添加吧，然后使用scons重新生成工程。

13.6.2 RT-Thread 网络设备管理

RT-Thread有一套自己的设备框架，这里只作一个简单的描述，具体请参考《RT-Thread编程指南第六章-I/O设备管理》，可以在RT-Thread入门帖中找到。RT-Thread中包含很多设备，为了更简单的添加或者管理这些设备，使用面向对象的思想将设备抽象成了一个类，基于这个“设备类”，我们派生出不同类型的设备类，如：网络设备类、字符设备类、块设备类、音频设备类等等，它们的关系图如下：

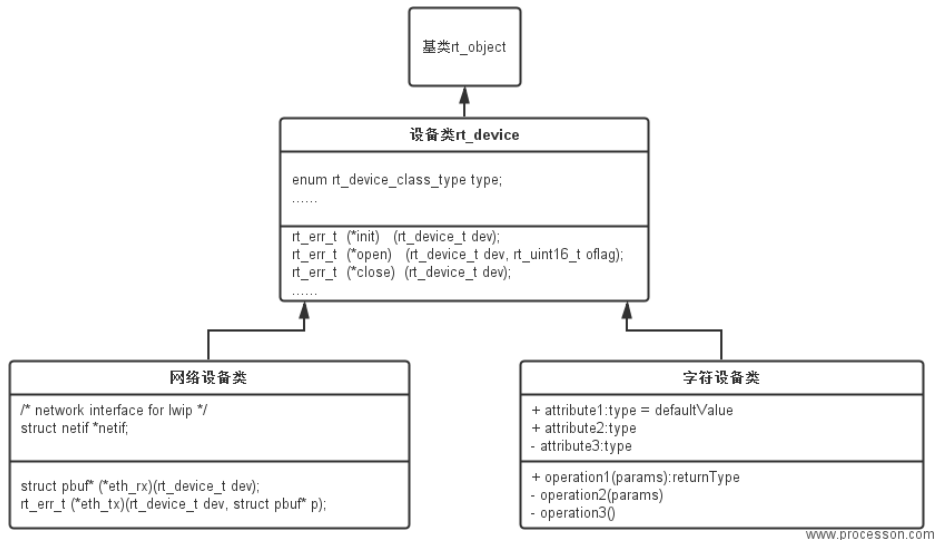


图 13.3: RTT设备继承关系

除基类以外，其他继承自基类的类分别加上了与基类不同的属性和接口，比如设备类中就添加了基类没有的设备初始化，打开，关闭的接口和设备类型的属性。

有了这个概念接着说RT-Thread中设备的管理，RT-Thread中有一个数组，里面为每一种对象（信号、邮箱、设备、定时器）分配了一个链表（用结构体封装了），如下：

```

struct rt_object_information
{
    enum rt_object_class_type type;          /**< object class type*/
    rt_list_t object_list;                  /**< object list */
    rt_size_t object_size;                  /**< object size */
};

struct rt_object_information rt_object_container[RT_Object_Class_Unknown] =
{
    /* initialize object container - thread */
    {RT_Object_Class_Thread, _OBJ_CONTAINER_LIST_INIT(RT_Object_Class_Thread),
        sizeof(struct rt_thread)},
#ifdef RT_USING_SEMAPHORE
    /* initialize object container - semaphore */
    {RT_Object_Class_Semaphore,
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Class_Semaphore),
        sizeof(struct rt_semaphore)},
#endif
#ifdef RT_USING_MUTEX
    /* initialize object container - mutex */
    {RT_Object_Class_Mutex, _OBJ_CONTAINER_LIST_INIT(RT_Object_Class_Mutex),
        sizeof(struct rt_mutex)},
#endif
#ifdef RT_USING_EVENT
    /* initialize object container - event */

```

```

    {RT_Object_Class_Event, _OBJ_CONTAINER_LIST_INIT(RT_Object_Class_Event),
        sizeof(struct rt_event)},
#endif
#ifdef RT_USING_MAILBOX
    /* initialize object container - mailbox */
    {RT_Object_Class_MailBox, _OBJ_CONTAINER_LIST_INIT(RT_Object_Class_MailBox),
        sizeof(struct rt_mailbox)},
#endif
#ifdef RT_USING_MESSAGEQUEUE
    /* initialize object container - message queue */
    {RT_Object_Class_MessageQueue,
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Class_MessageQueue),
        sizeof(struct rt_messagequeue)},
#endif
#ifdef RT_USING_MEMHEAP
    /* initialize object container - memory heap */
    {RT_Object_Class_MemHeap, _OBJ_CONTAINER_LIST_INIT(RT_Object_Class_MemHeap),
        sizeof(struct rt_memheap)},
#endif
#ifdef RT_USING_MEMPOOL
    /* initialize object container - memory pool */
    {RT_Object_Class_MemPool, _OBJ_CONTAINER_LIST_INIT(RT_Object_Class_MemPool),
        sizeof(struct rt_mempool)},
#endif
#ifdef RT_USING_DEVICE
    /* initialize object container - device */
    {RT_Object_Class_Device, _OBJ_CONTAINER_LIST_INIT(RT_Object_Class_Device),
        sizeof(struct rt_device)},
#endif
/* initialize object container - timer */
{RT_Object_Class_Timer, _OBJ_CONTAINER_LIST_INIT(RT_Object_Class_Timer),
    sizeof(struct rt_timer)},
#ifdef RT_USING_MODULE
    /* initialize object container - module */
    {RT_Object_Class_Module, _OBJ_CONTAINER_LIST_INIT(RT_Object_Class_Module),
        sizeof(struct rt_module)},
#endif
};

```

具体地讲，RT-Thread中使用一个链表来维护所有的设备，当需要往系统中注册设备时，需要将设备添加到对应的链表中（当然如何添加，RT-Thread提供了相应的接口）。如果对代码不了解，简单点的理解方式请看下图（图中并不对应实际的代码，代码中用的双向链表）：

从图中可知，当系统需要操作网卡时，直接遍历这个链表即可。

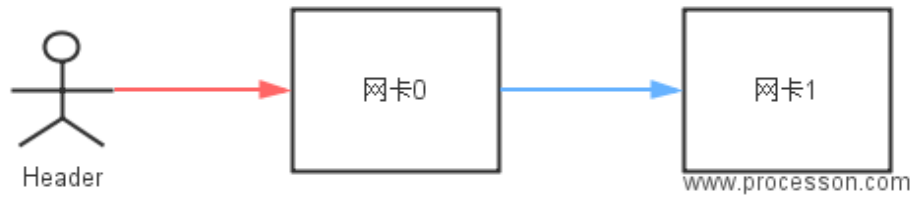


图 13.4: RTT网络设备管理

13.6.3 RT-Thread lwIP有哪些变化

- 上面提到过，将sys.c/h中的接口实现基本的移植工作就完成了，细心的读者可能会拿RT-Thread中lwIP这部分源码与lwIP官方的源码做一个对比，然后会发现RT-Thread增加一个“arch”目录，这部分代码主要实现了前面提到的信号量、互斥锁、邮箱等sys.h文件中的接口，另外RT-Thread根据其系统自身增加了lwIP的初始化工作，如下：

```

/**
 * LwIP system initialization
 */
void lwip_system_init(void)
{
    rt_err_t rc;
    struct rt_semaphore done_sem;

    /* set default netif to NULL */
    netif_default = RT_NULL;

    // 初始化信号量
    rc = rt_sem_init(&done_sem, "done", 0, RT_IPC_FLAG_FIFO);

    if (rc != RT_EOK)
    {
        LWIP_ASSERT("Failed to create semaphore", 0);

        return;
    }

    // 这是关键代码，调用sys_thread_new()创建lwIP线程，并回调tcipip_init_done_callback()初始化网卡设备
    tcipip_init(tcipip_init_done_callback, (void *)&done_sem);

    //等待tcipip_init_done_callback()初始化完成
    /* waiting for initialization done */
    if (rt_sem_take(&done_sem, RT_WAITING_FOREVER) != RT_EOK)
    {
        rt_sem_detach(&done_sem);
    }
}

```



```

    return;
}
// 将此信号量从系统的信号量对象链表中删除
rt_sem_detach(&done_sem);

/* set default ip address */
#if !LWIP_DHCP //如果未启用DHCP, 即表示使用静态IP, 则配置默认网卡的IP、子网掩码、网关
if (netif_default != RT_NULL) //上面提到过, 如果此时系统还未注册网卡设备, 这部分代码也不执行。
{
    struct ip_addr ipaddr, netmask, gw;

    IP4_ADDR(&ipaddr, RT_LWIP_IPADDR0, RT_LWIP_IPADDR1, RT_LWIP_IPADDR2,
              RT_LWIP_IPADDR3);
    IP4_ADDR(&gw, RT_LWIP_GWADDR0, RT_LWIP_GWADDR1, RT_LWIP_GWADDR2,
              RT_LWIP_GWADDR3);
    IP4_ADDR(&netmask, RT_LWIP_MSKADDR0, RT_LWIP_MSKADDR1, RT_LWIP_MSKADDR2,
              RT_LWIP_MSKADDR3);

    netifapi_netif_set_addr(netif_default, &ipaddr, &netmask, &gw);
}
#endif
}

```

这段代码的解释通过注释的方式, 大家请参照代码旁边的注释。

- 单纯在RT-Thread中完成lwIP初始化和创建lwIP线程的工作还是不够的, 因为要让协议栈与外界通信, 系统必须可以收发数据, 所以还需要硬件驱动的支持, 这时牵扯到RT-Thread收发包的设计和网卡驱动。这部分的整体框架如下图:

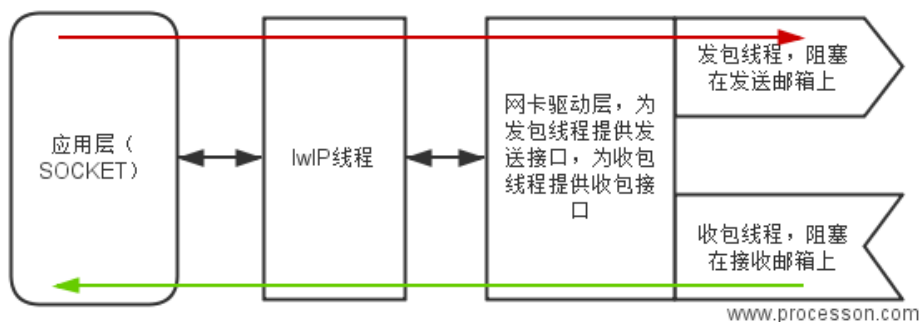


图 13.5: RTT收发包设计

由此可知, RT-Thread中将lwIP应用起来主要包括三个核心步骤: 1. 创建收发包线程, 调用接口eth_system_device_init(). 2. 提供网卡驱动, 调用网卡初始化函数, 注册网卡设备。(驱动不同相应的接口函数可能不同) 3. 初始化lwIP, 创建lwIP线程, 调用接口lwip_sys_init() (实际调用的lwip_system_init())。

至此, 三个步骤完成之后, 应用层便可以直接与外界通讯。

13.6.4 RT-Thread lwIP相关代码补充说明

前面我们讲解过lwip_system_init()中，当系统中没有网卡设备时，有一部分初始化工作（为网卡初始化IP、子网掩码、网关等）是不会进行的。此时lwIP线程已经创建，如果需要和外界通讯，那么必须为系统添加网卡设备，而在网卡驱动中，网卡设备初始化时，会向系统注册，此时网卡设备就添加到系统中了。以[RT-Thread双网口开发板网卡驱动例程](#)为例，参考以下代码：

```
#ifdef USING_MAC0
    /* set autonegotiation mode */
    fm3_emac_device0.phy_mode = EMAC_PHY_AUTO;
    fm3_emac_device0.FM3_ETHERNET_MAC = FM3_ETHERNET_MAC0;
    fm3_emac_device0.ETHER_MAC_IRQ = ETHER_MAC0_IRQn;

    // OUI 00-00-0E FUJITSU LIMITED
    fm3_emac_device0.dev_addr[0] = 0x00;
    fm3_emac_device0.dev_addr[1] = 0x00;
    fm3_emac_device0.dev_addr[2] = 0x0E;
    /* set mac address: (only for test) */
    fm3_emac_device0.dev_addr[3] = 0x12;
    fm3_emac_device0.dev_addr[4] = 0x34;
    fm3_emac_device0.dev_addr[5] = 0x56;

    fm3_emac_device0.parent.parent.init = fm3_emac_init;
    fm3_emac_device0.parent.parent.open = fm3_emac_open;
    fm3_emac_device0.parent.parent.close = fm3_emac_close;
    fm3_emac_device0.parent.parent.read = fm3_emac_read;
    fm3_emac_device0.parent.parent.write = fm3_emac_write;
    fm3_emac_device0.parent.parent.control = fm3_emac_control;
    fm3_emac_device0.parent.parent.user_data = RT_NULL;

    fm3_emac_device0.parent.eth_rx = fm3_emac_rx;
    fm3_emac_device0.parent.eth_tx = fm3_emac_tx;

    /* init tx buffer free semaphore */
    rt_sem_init(&fm3_emac_device0.tx_buf_free, "tx_buf0", EMAC_TXBUFNB,
               RT_IPC_FLAG_FIFO);

    // 关键代码，驱动向系统注册网卡设备
    eth_device_init(&(fm3_emac_device0.parent), "e0");
#endif /* #ifdef USING_MAC0 */
```

eth_device_init()调用eth_device_init_with_flag()接口初始化网卡设备（为网卡添加名称，IP、子网掩码、网关，网卡设备使用的发包和收包接口函数等），并向系统注册网卡设备。到此，解释了一个现象：网卡驱动初始化和lwIP的初始化顺序互换并无影响。

13.7 网络编程示例

13.7.1 UDP使用示例

下面是一个在RT-Thread上使用BSD socket接口的UDP服务端例子，当把这个代码加入到RT-Thread操作系统时，它会自动向finsh命令行添加一个udpserv命令，在finsh上执行udpserv()函数即可启动这个UDP服务端，该UDP服务端在端口5000上进行监听。

当服务端接收到数据时，它将把数据打印到控制终端中；如果服务端接收到exit字符串时，那么服务端将退出服务。

```
/*
 * 代码清单：UDP服务端例子
 */
#include <rtthread.h>
#include <lwip/sockets.h> /* 使用BSD socket, 需要包含sockets.h头文件 */

void udpserv(void* paramemter)
{
    int sock;
    int bytes_read;
    char *recv_data;
    rt_uint32_t addr_len;
    struct sockaddr_in server_addr, client_addr;

    /* 分配接收用的数据缓冲 */
    recv_data = rt_malloc(1024);
    if (recv_data == RT_NULL)
    {
        /* 分配内存失败, 返回 */
        rt_kprintf("No memory\n");
        return;
    }

    /* 创建一个socket, 类型是SOCK_DGRAM, UDP类型 */
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
        rt_kprintf("Socket error\n");

        /* 释放接收用的数据缓冲 */
        rt_free(recv_data);
        return;
    }

    /* 初始化服务端地址 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(5000);
    server_addr.sin_addr.s_addr = INADDR_ANY;
    rt_memset(&(server_addr.sin_zero), 0, sizeof(server_addr.sin_zero));
```

```

/* 绑定socket到服务端地址 */
if (bind(sock, (struct sockaddr *) &server_addr, sizeof(struct sockaddr))
    == -1)
{
    /* 绑定地址失败 */
    rt_kprintf("Bind error\n");

    /* 释放接收用的数据缓冲 */
    rt_free(recv_data);
    return;
}

addr_len = sizeof(struct sockaddr);
rt_kprintf("UDPServer Waiting for client on port 5000...\n");

while (1)
{
    /* 从sock中收取最大1024字节数据 */
    bytes_read = recvfrom(sock, recv_data, 1024, 0,
                          (struct sockaddr *) &client_addr, &addr_len);
    /* UDP不同于TCP, 它基本不会出现收取的数据失败的情况, 除非设置了超时等待 */

    recv_data[bytes_read] = '\0'; /* 把末端清零 */

    /* 输出接收的数据 */
    rt_kprintf("\n(%s , %d) said : ", inet_ntoa(client_addr.sin_addr),
              ntohs(client_addr.sin_port));
    rt_kprintf("%s", recv_data);

    /* 如果接收数据是exit, 退出 */
    if (strcmp(recv_data, "exit") == 0)
    {
        lwip_close(sock);

        /* 释放接收用的数据缓冲 */
        rt_free(recv_data);
        break;
    }
}

return;
}

#ifdef RT_USING_FINSH

```

```
#include <finsh.h>
/* 输出udpserv函数到finsh shell中 */
FINSH_FUNCTION_EXPORT(udpserv, startup udp server);
#endif
```

下面是另一个在RT-Thread上使用BSD socket接口的UDP客户端例子。当把这个代码加入到RT-Thread时，它会自动向finsh命令行添加一个udpclient命令，在finsh上执行udpclient(url,port)函数即可启动这个UDP客户端，url指定了这个客户端连接到的服务端地址或域名，port是相应的端口号。

当UDP客户端启动后，它将连续发送5次“This is UDP Client from RT-Thread.”的字符串给服务端，然后退出。

```
/*
 * 程序清单：UDP客户端例子
 */
#include <rtthread.h>
#include <lwip/netdb.h> /* 为了解析主机名，需要包含netdb.h头文件 */
#include <lwip/sockets.h> /* 使用BSD socket，需要包含sockets.h头文件 */

/* 发送用到的数据 */
ALIGN(4)
const char send_data[] = "This is UDP Client from RT-Thread.\n";
void udpclient(const char* url, int port, int count)
{
    int sock;
    struct hostent *host;
    struct sockaddr_in server_addr;

    /* 通过函数入口参数url获得host地址（如果是域名，会做域名解析） */
    host = (struct hostent *) gethostbyname(url);

    /* 创建一个socket，类型是SOCK_DGRAM，UDP类型 */
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
        rt_kprintf("Socket error\n");
        return;
    }

    /* 初始化预连接的服务端地址 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    server_addr.sin_addr = *((struct in_addr *) host->h_addr);
    rt_memset(&(server_addr.sin_zero), 0, sizeof(server_addr.sin_zero));

    /* 总计发送count次数据 */
    while (count)
    {
```

```

        /* 发送数据到服务远端 */
        sendto(sock, send_data, strlen(send_data), 0,
                (struct sockaddr *) &server_addr, sizeof(struct sockaddr));

        /* 线程休眠一段时间 */
        rt_thread_delay(50);

        /* 计数值减一 */
        count--;
    }

    /* 关闭这个socket */
    lwip_close(sock);
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出udpclient函数到finsh shell中 */
FINSH_FUNCTION_EXPORT(udpclient, startup udp client);
#endif

```

13.7.2 TCP使用示例

下面是一个在RT-Thread上使用BSD socket接口的TCP服务端例子。当把这个代码加入到RT-Thread时，它会向finsh命令行添加一个tcpserv命令，在finsh上执行tcpserv()函数即可启动这个TCP服务端，该TCP服务端在端口5000上进行监听。当有TCP客户向这个服务端进行连接后，只要服务端接收到数据，它就会立即向客户端发送“This is TCP Server from RT-Thread.”的字符串。

如果服务端接收到q或Q字符串时，服务器将主动关闭这个TCP连接。如果服务端接收到exit字符串时，那么将退出服务。

```

/*
 * 程序清单：TCP服务端例子
 */
#include <rtthread.h>
#include <lwip/sockets.h> /* 使用BSD Socket接口必须包含sockets.h这个头文件 */

/* 发送用到的数据 */
ALIGN(4)
static const char send_data[] = "This is TCP Server from RT-Thread.";
void tcpserv(void* parameter)
{
    char *recv_data; /* 用于接收的指针，后面会做一次动态分配以请求可用内存 */
    rt_uint32_t sin_size;
    int sock, connected, bytes_received;
    struct sockaddr_in server_addr, client_addr;
    rt_bool_t stop = RT_FALSE; /* 停止标志 */

```

```

recv_data = rt_malloc(1024); /* 分配接收用的数据缓冲 */
if (recv_data == RT_NULL)
{
    rt_kprintf("No memory\n");
    return;
}

/* 一个socket在使用前, 需要预先创建出来, 指定SOCK_STREAM为TCP的socket */
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    /* 创建失败的错误处理 */
    rt_kprintf("Socket error\n");

    /* 释放已分配的接收缓冲 */
    rt_free(recv_data);
    return;
}

/* 初始化服务端地址 */
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(5000); /* 服务端工作的端口 */
server_addr.sin_addr.s_addr = INADDR_ANY;
rt_memset(&(server_addr.sin_zero), 8, sizeof(server_addr.sin_zero));

/* 绑定socket到服务端地址 */
if (bind(sock, (struct sockaddr *) &server_addr, sizeof(struct sockaddr))
    == -1)
{
    /* 绑定失败 */
    rt_kprintf("Unable to bind\n");

    /* 释放已分配的接收缓冲 */
    rt_free(recv_data);
    return;
}

/* 在socket上进行监听 */
if (listen(sock, 5) == -1)
{
    rt_kprintf("Listen error\n");

    /* release recv buffer */
    rt_free(recv_data);
    return;
}

```

```

}

rt_kprintf("\nTCPServer Waiting for client on port 5000...\n");
while (stop != RT_TRUE)
{
    sin_size = sizeof(struct sockaddr_in);

    /* 接受一个客户端连接socket的请求, 这个函数调用是阻塞式的 */
    connected = accept(sock, (struct sockaddr *) &client_addr, &sin_size);
    /* 返回的是连接成功的socket */

    /* 接受返回的client_addr指向了客户端的地址信息 */
    rt_kprintf("I got a connection from (%s , %d)\n", inet_ntoa(
        client_addr.sin_addr), ntohs(client_addr.sin_port));

    /* 客户端连接的处理 */
    while (1)
    {
        /* 发送数据到connected socket */
        send(connected, send_data, strlen(send_data), 0);

        /*
         * 从connected socket中接收数据, 接收buffer是1024大小,
         * 但并不一定能够收到1024大小的数据
         */
        bytes_received = recv(connected, recv_data, 1024, 0);
        if (bytes_received < 0)
        {
            /* 接收失败, 关闭这个connected socket */
            lwip_close(connected);
            break;
        }

        /* 有接收到数据, 把末端清零 */
        recv_data[bytes_received] = '\0';
        if (strcmp(recv_data, "q") == 0 || strcmp(recv_data, "Q") == 0)
        {
            /* 如果是首字母是q或Q, 关闭这个连接 */
            lwip_close(connected);
            break;
        }
        else if (strcmp(recv_data, "exit") == 0)
        {
            /* 如果接收的是exit, 则关闭整个服务端 */
            lwip_close(connected);
        }
    }
}

```

```

        stop = RT_TRUE;
        break;
    }
    else
    {
        /* 在控制终端显示收到的数据 */
        rt_kprintf("RECIEVED DATA = %s \n", recv_data);
    }
}

/* 退出服务 */
lwip_close(sock);

/* 释放接收缓冲 */
rt_free(recv_data);

return;
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出tcpserv函数到finsh shell中 */
FINSH_FUNCTION_EXPORT(tcpserv, startup tcp server);
#endif

```

下面则是另一个如在RT-Thread上使用BSD socket接口的TCP客户端例子。当把这个代码加入到RT-Thread时，它会自动向finsh 命令行添加一个tcpclient命令，在finsh上执行tcpclient(url,port)函数即可启动这个TCP服务端，url指定了这个客户端连接到的服务端地址或域名，port是相应的端口号。当TCP客户端连接成功时，它会接收服务端传过来的数据。当有数据接收到时，如果是以q或Q开头，它将主动断开这个连接；否则会把接收的数据在控制终端中打印出来，然后发送“This is TCP Client from RT-Thread.”的字符串。

```

/*
 * 程序清单：TCP客户端例子
 */
#include <rtthread.h>
#include <lwip/netdb.h> /* 为了解析主机名，需要包含netdb.h头文件 */
#include <lwip/sockets.h> /* 使用BSD socket，需要包含sockets.h头文件 */

/* 发送用到的数据 */
ALIGN(4)
static const char send_data[] = "This is TCP Client from RT-Thread.";
void tcpclient(const char* url, int port)
{
    char *recv_data;
    struct hostent *host;

```



```
int sock, bytes_received;
struct sockaddr_in server_addr;

/* 通过函数入口参数url获得host地址（如果是域名，会做域名解析） */
host = gethostbyname(url);

/* 分配用于存放接收数据的缓冲 */
recv_data = rt_malloc(1024);
if (recv_data == RT_NULL)
{
    rt_kprintf("No memory\n");
    return;
}

/* 创建一个socket, 类型是SOCKET_STREAM, TCP类型 */
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    /* 创建socket失败 */
    rt_kprintf("Socket error\n");

    /* 释放接收缓冲 */
    rt_free(recv_data);
    return;
}

/* 初始化预连接的服务端地址 */
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(port);
server_addr.sin_addr = *((struct in_addr *) host->h_addr);
rt_memset(&(server_addr.sin_zero), 0, sizeof(server_addr.sin_zero));

/* 连接到服务端 */
if (connect(sock, (struct sockaddr *) &server_addr,
    sizeof(struct sockaddr)) == -1)
{
    /* 连接失败 */
    rt_kprintf("Connect error\n");

    /* 释放接收缓冲 */
    rt_free(recv_data);
    return;
}

while (1)
{
```

```

    /* 从sock连接中接收最大1024字节数据 */
    bytes_received = recv(sock, recv_data, 1024, 0);
    if (bytes_received < 0)
    {
        /* 接收失败, 关闭这个连接 */
        lwip_close(sock);

        /* 释放接收缓冲 */
        rt_free(recv_data);
        break;
    }

    /* 有接收到数据, 把末端清零 */
    recv_data[bytes_received] = '\0';

    if (strcmp(recv_data, "q") == 0 || strcmp(recv_data, "Q") == 0)
    {
        /* 如果是首字母是q或Q, 关闭这个连接 */
        lwip_close(sock);

        /* 释放接收缓冲 */
        rt_free(recv_data);
        break;
    }
    else
    {
        /* 在控制终端显示收到的数据 */
        rt_kprintf("\nRecieved data = %s ", recv_data);
    }

    /* 发送数据到sock连接 */
    send(sock, send_data, strlen(send_data), 0);
}

return;
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出tcpclient函数到finsh shell中 */
FINSH_FUNCTION_EXPORT(tcpclient, startup tcp client);
#endif

```

第 14 章

lwIP-IPv6 支持

14.1 lwIP-IPV6 概况

lwIP的[git开发分支](#)支持IPv4/v6双栈，并已支持绝大多数的IPv6特性，开发者认为lwIP的实现已基本稳定([详见邮件](#))，但是由于部分开发者认为lwIP并未实现6LowPAN和RPL协议，所以近期内并不会发布lwIP 1.5.0-Beta ([原文](#))。lwIP的git开发分支已支持以下功能([原文](#))：

1. 支持IPv6层协议
2. 在tcp/udp/raw 协议控制块中支持IPv6
3. Netconn API支持IPv6
4. Socket API支持IPv6
5. 支持ICMPv6
6. 支持邻居发现协议(Neighbor Discovery)
7. 支持组播侦听发现模式(Multicast Listener Discovery)
8. 支持无状态地址自动配置
9. 支持IPv6数据包分片与重组
10. 网络接口层支持IPv6

尽管lwIP-head的IPv6支持已基本稳定，但是仍有部分功能待开发：

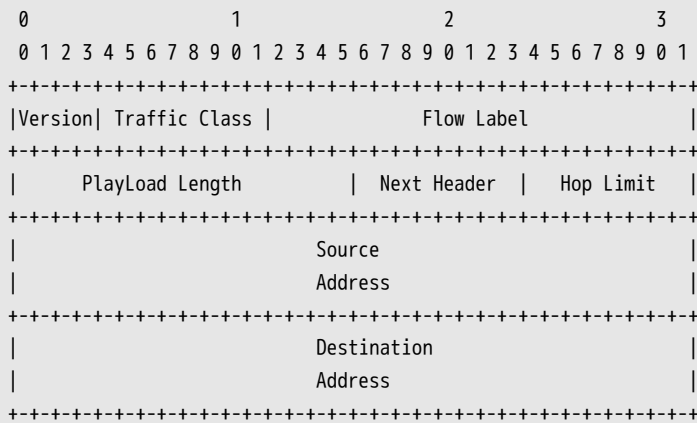
1. 在不同的netif结构体中添加Scope id的支持，在利用link-local地址通信时，Scope id可提供路由信息
2. 在BSD Socket API中有多个函数实现不完善

14.2 IPv6基础知识

互联网通信协议第6版 (Internet Protocol version 6, 简写: IPv6) 是互联网协议的最新版本，用于数据包交换互联网络的网络层协议，旨在解决IPv4地址枯竭的问题，IPv6意图取代IPv4，而IPv4在网络上仍然占据着绝大多数的份额。

14.2.1 IPv6报文格式

IPv6报文有8个字段，固定大小为40字节，每一个IPv6数据包都必须包含报头，基本报头结构如下图所示：



- Version: 版本号，长度为4bit，对于IPv6，其值为6
- Traffic Class: 流类别，长度为8bit，用于表示数据报的优先级，用于QoS
- Flow Label: 流标签，长度为20bit，用于区分实时流量，不同的流标签+源地址可以唯一确定一条数据流
- Payload Length: 有效载荷长度，长度为16bit，指的是紧跟IPv6报头的数据包的其他部分（扩展报头和上层协议数据单元）。该字段只能表示最大长度为65535字节的有效载荷。如果长度超过这个值，该字段会置0，而有效载荷长度用逐跳选项扩展报头中的超大有效载荷选项来表示
- Next Header: 下一个报头，长度为8bit，该字段定义了紧跟在IPv6报头后面的第一个扩展报头的类型，或者上层协议数据单元中的协议类型
- Hop Limit: 跳数限制，长度为8bit，该字段类似于IPv4中的TTL字段，定义了IP数据包经过的最大跳数。
- Source Address: 源地址，128bit，发送方的地址
- Destination Address: 目的地址，128bit，接收方的地址

14.2.2 IPv6扩展报头

IPv4中，报头包含了可选字段，因而其报头长度是不定长的，在路由转发过程中，处理携带可选字段的IPv4报文会占用很大的资源，因而在IPv6中将扩展报头放在了IPv6报头和上层协议数据单元之间。IPv6可拥有0个、1个或多个的扩展头，扩展报头由前一段的next hdr标识。

IPv6支持的扩展报头有：逐跳选项扩展报头、路由扩展报头、分片扩展报头、目的选项扩展报头、身份验证扩展报头、封装安全有效载荷扩展报头，由于每一个扩展头的内容可以决定是否处理下一个报头，所以扩展报头出现的次序必须遵循一定的原则，一般来说会按照如下的顺序排列：1) IPv6报头 2) 逐跳选项扩展报头 3) 目的选项扩展报头 4) 路由扩展报头 5) 分片扩展报头 6) 身份认证扩展报头 7) 封装安全有效载荷扩展报头 8) 目的选项扩展报头 9) 上层应用报头，除了目的选项扩展报头外，其他扩展报头只能出现一次，每个扩展头的含义如下：

1. 逐跳选项扩展报头，定义了转发路径中每个节点需要处理的信息
2. 目的选项扩展报头，目的节点需要处理的信息
3. 路由扩展报头，可用于强制数据包经过的特定设备

4. 分片扩展报头，发送大于MTU的包，不同于IPV4，IPV6只在源节点进行数据的分片
5. 身份认证扩展报头，确保数据来源于可信任的源点
6. 封装安全有效载荷扩展报头，可以有效避免在数据传输过程中被窃听、抓取内容等行为

14.2.3 IPv4与IPv6对比

1. 地址空间，IPv6拥有比IPv4多得多的地址空间，由于IPv6采用了128位的地址，可支持（43亿×43亿×43亿×43亿）个地址，无疑地址空间是IPv6的最大优势。
2. 报文格式，IPv6报头中删除了首部长度、DSCP、标识符、标志、分片偏移、首部检验和的6个域，修改了traffic class（流量分类）、payload length（负载长度）、hop limit（跳数限制）域，对应于IPv4中的协议、全长、存活时间，增加了Flow Label（流标签）域，并且由于IPv6采用了固定长度的报头，IPv6的报文处理效率较IPv4更加高效。
3. 安全性，在IPv4中安全机制往往在应用层实现，而在IPv6中增加了IPsec的加密与认证，可以保证端对端传输的安全。
4. 服务质量的保证，由于IPv6报头中增加了流标签域，可提供QoS的支持，这符合多媒体网络迅速发展的趋势。
5. 移动性的改进，IPv6的地址自动配置协议简化了移动节点的转交地址的分配，从而避免了外地代理的使用，并由于IPv6扩展报头的设计，可解决移动IPv4的三角路由、源地址过滤问题，移动通信处理效率更高。

14.2.4 IPv6地址

IPv6地址格式 IPv6地址包含128比特，以16位为一组，每组以冒号“:”隔开，可分为8组，每组以4位十六进制方式表示，以下为合法的IPv6地址：“2001:0DB8:02de:0000:0000:0000:0000:0e13”，同时IPv6还支持省略规则：

规则1：每项数字前导0可省略，省略后前导数字仍为0的可继续，上述的地址可等效为：“2001:DB8:2de:0:0:0:0:e13”

规则2：可用双冒号“::”来表示一组或多组的0，但只可出现一次，上述的地址可等效为：“2001:DB8:2de::e13”

IPv6地址分类 IPv6地址可分为三类：

- 单播(unicast)地址

单播地址标示一个网络接口。协议会把送往地址的数据包投送其接口，IPv6的单播地址包括了未指定地址、环回地址、全球单播地址、链路本地地址、唯一本地地址ULA(Unique Local Address)。

- 任播(anycast)地址

任播是IPv6特有的数据发送方式，它拥有一组接收节点的地址栏表，但指定为Anycast的数据包，只会发送给距离最近或者发送成本最低的其中一个接收地址。任播地址设计用来给多个主机或者节点提供相同服务时提供冗余功能和负载分担功能，任播地址与单播地址拥有相同的地址空间，主要用于移动IPv6中。

- 多播(multicast)地址

多播地址也称为组播地址。多播地址被指定为一群不同的接口，发送到多播地址的数据包会被发送到所有的地址。多播地址由皆为1的字节起始，它们的前置为FF00::/8。其中第二个字节的最后四个比特用以表示“范畴”，有node-local(0x1),link-local(0x2),site-local(0x5),organization-local(0x8),global(0xE)。

14.2.5 邻居发现协议

邻居发现协议（Neighbor Discovery Protocol，简写：NDP或ND），是用于替代IPv4中的ARP协议的，用于实现网络层地址与链路层地址之间的映射，NDP实现效率要比ARP高。IPv6邻居发现协议可提供以下功能：1）无服务器的自动配置 2）路由发现 3）地址解析 4）邻居不可达检测 5）链路MTU发现 6）下一跳决定 7）重复地址检测等功能。邻居发现协议定义了5种ICMPv6类型：

- 路由请求：

当节点不愿意等到下一次周期性的路由器宣告时，可发起一次路由器请求的多播包。正在初始化的节点可使用路由器请求，这样即可得到路由相关参数
- 路由通告：

路由器可周期性的发送路由器宣告包，这样链路内的节点就可获得相关的路由配置信息，路由器宣告包的跳数限制为255，这样防止非本链路的路由发送路由器宣告包来干扰本链路的通信
- 邻居请求

用于确定邻居的链路层地址，判断缓存中的链路层地址是否可达，判断链路中是否存在重复的IP地址。这里的跳数限制仍然为255，防止邻居请求包通过路由器
- 邻居通告

一种情况是应答邻居请求，另一种情况是当节点发生改变时发送多播包给本链路中的节点通知链路层地址改变信息。
- 重定向

由路由器发送，用于把数据包重定向到两路中链路质量更好的节点

14.3 RT-Thread中如何使用IPv6

要使用IPv6，需使用lwip-head版本的协议栈，需要在相应的BSP包中的rtconfig.h文件中添加“#define RT_USING_LWIP_HEAD”。

14.3.1 使用IPv4/v6双栈

要在RT-Thread中使用IPv4/v6双栈，需在相应的BSP包中的rtconfig.h文件添加“#define RT_LWIP_IPV6”，这样当网络初始化后就会为网卡创建link-local address，用于局域网内的通信。当然也可以为网卡创建全球单播地址，这里提供了两种方式：

1）无状态地址自动配置，只要在rtconfig.h中添加“#define RT_LWIP_IPV6_AUTOCONFIG”，这样将开发板接入支持IPv6的路由器时（如极路由和其他支持openwrt系统的路由器）即可完成IPv6地址的自动配置。

2）手动配置，在你的应用程序中完成了网络的初始化后，可调用“void set_if6(char* netif_name, char* ip6_addr)”函数设置网络地址，如你想要为网卡“e0”设置“2001::1”的地址，则调用set_if6(“e0”，“2001::1”)即可。

14.3.2 仅使用IPv4

在lwip-head中网络层仅使用IPv4协议栈，只要不在相应的BSP包中的rtconfig.h文件添加“#define RT_LWIP_IPV6”即可，RT-Thread不会将IPv6相关的源文件、头文件编译进去。IPv4中仍然支持使用DHCP协议与静态IP地址配置。

14.3.3 对开发板进行Ping测试

要了解PC和开发板之间的网络连接状态时，需使用Ping的测试方法。

首先，要在串口调试工具中输入list_if()获取开发板的IPv4/IPv6地址。如下图所示：

```
network interface: e0
MTU: 1500
MAC: 00 00 0e 12 34 56
FLAGS: UP LINK_UP DHCP ETHARP
ip address: 192.168.199.134
gw address: 192.168.199.1
net mask : 255.255.255.0
link-local address: FE80::200:EFF:FE12:3456
ipv6[1] address: 4006:E024:680:C6E:200:EFF:FE12:3456
```

图 14.1: IP_Information

IPv4地址为192.168.199.134，IPv6的link-local地址为fe80::200:eff:fe12:3456,IPv6地址为4006:e024:680:c6e:200:eff:fe12:3456。

需要注意的是，测试前，要通过使用路由器或者静态IP的方法使得PC与开发板处于同一网段，这里为简单起见使用路由器。

- IPv4 ping测试

IPv4的ping测试十分简单，只需打开命令提示符输入ping 192.168.199.134 -t即可

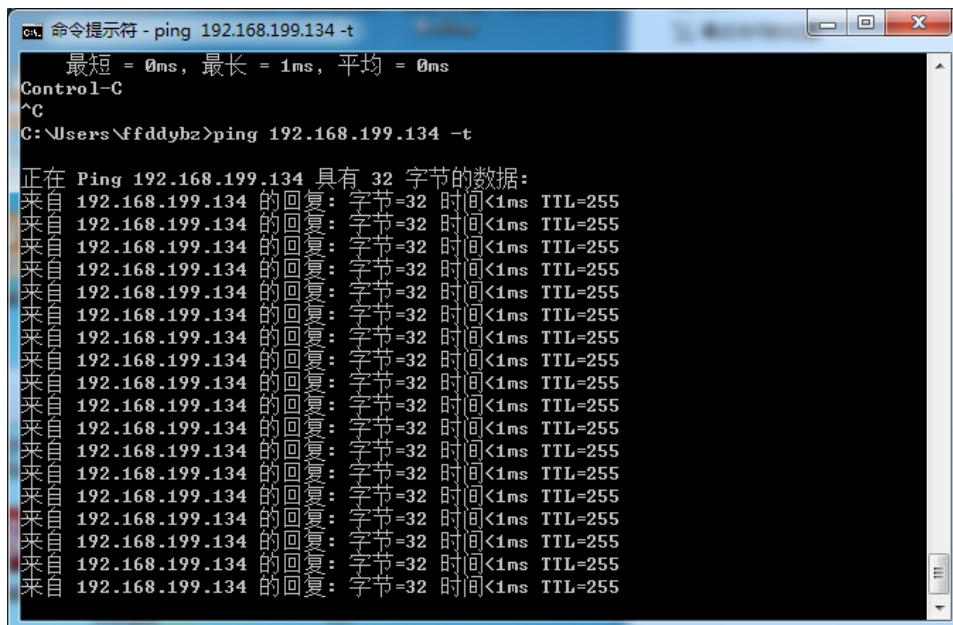


图 14.2: ping4

- IPv6 link-local address ping测试

首先需在命令提示符中输入ipconfig，获得以下信息：

之后在命令行提示符中输入ping fe80::200:eff:fe12:3456%12 -t

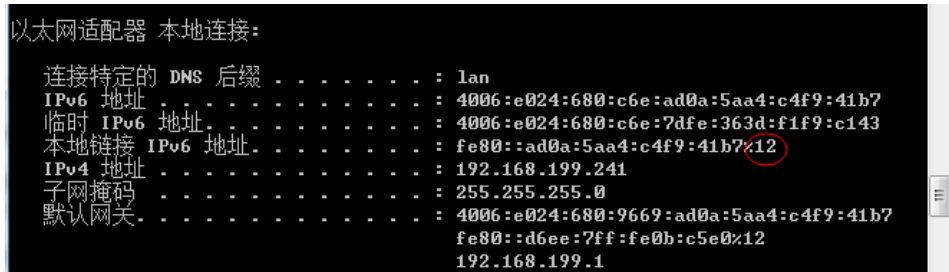


图 14.3: Scope ID

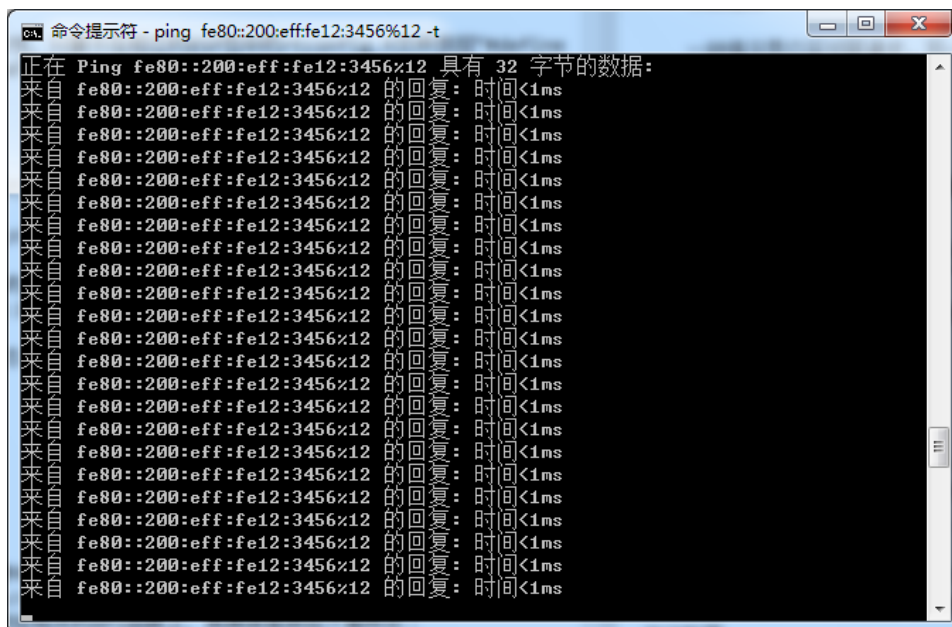


图 14.4: Link-local Address

• IPv6地址的测试

在命令提示符中输入ping 4006:e024:680:c6e:200:eff:fe12:3456 -t

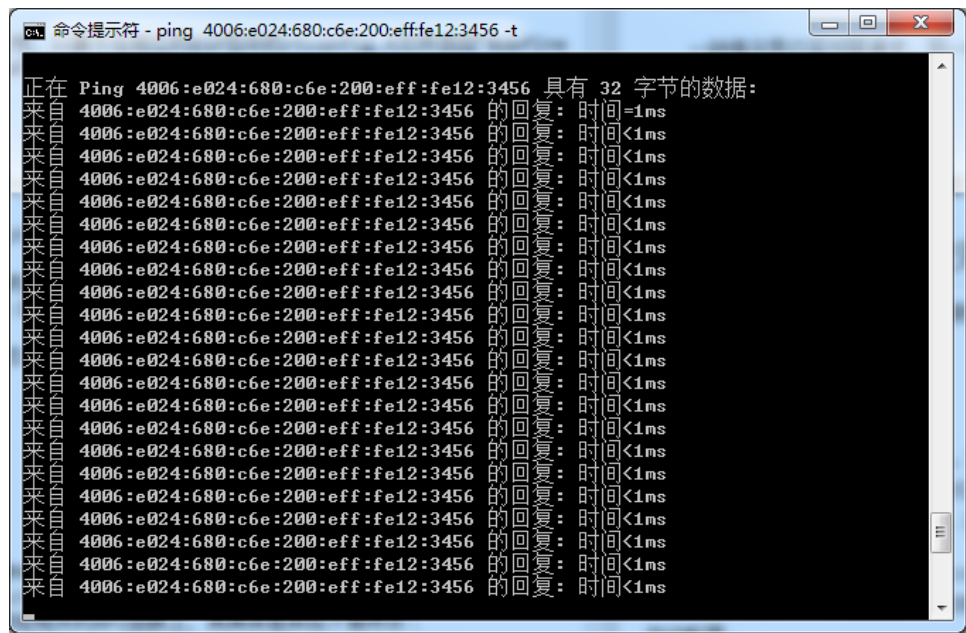


图 14.5: ping6

至此你已进入IPv6的世界了。

14.4 IPv6 Socket API实例

14.4.1 IPv4/v6 Socket编程异同

在基于Socket API进行开发的应用程序中，IPv4/v6都使用基本相同的编程模型；如connect、accept、listen、send/sendto、recv/recvfrom等Socket API函数在IPv4/v6中用法也基本一致，而IPv4与IPv6间的主要差异体现在了与地址相关的API函数上，其具体差异如下图所示：

	IPv4	IPv6
地址族	AF_INET	AF_INET6
协议族	PF_INET	PF_INET6
数据结构	in_addr	in6_addr
结构体成员	sockaddr_in	sockaddr_in6
结构体长度	sin_len	sin6_len
协议族	sin_family	sin6_family
端口号	sin_port	sin6_port
地址	sin_addr	sin6_addr
	INADDR_ANY	IP6_ADDR_ANY
	inet_aton()/inet_addr()	inet_pton() inet_ntop()
	inet_ntoa() gethostbyname()	gethostbyname()

getaddrinfo()函数在lwIP中实现不完善，这里只介绍inet_pton()函数的用法：

函数原型：inet_pton(int family, const char strptr, void addrptr)

返回值：若成功返回1，若输入的不是有效表达式返回0，若出错则返回-1

参数：family：可以是AF_INET,也可以是AF_INET6,如果以不被支持的地址族作为参数，

函数将返回错误，并置errno为EAFNOSUPPORT

作用：该函数尝试转换由strptr指向的字符串，并通过addrptr指针存放二进制结果，完成了表达式到IPv6地址的转换。

关于getaddrinfo()、inet_ntop()函数的用法请参见Unix网络编程卷一。

14.4.2 PC测试程序

往往对IPv4的TCP/UDP的Server/Client进行测试时，Windows下有许多网络调试工具，而IPv6的测试主要是在Linux下利用Socket API编写的测试程序，具体请见[GitHub](#)，请在Linux的命令行下完成测试。

14.4.3 TCP Server例子

下面是一个在RT-Thread上使用BSD socket接口的TCP服务端例子。当把这个代码加入到RT-Thread时，它会自动向finsh命令行添加一个tcpserver6命令，在finsh上执行tcpserver6()函数即可启动这个TCP服务端，该TCP服务端在端口10001上进行监听。

当有TCP客户向这个服务端进行连接后，只要服务端接收到数据，它就会立即向客户端发送“This is TCP Server from RT-Thread.”的字符串。

如果服务端接收到q或Q字符串时，服务器将主动关闭这个TCP连接。如果服务端接收到exit字符串时，那么将退出服务。

```
#include <rtthread.h>
#include <lwip/sockets.h> /* 使用BSD Socket需包含socket.h */

#define SERV_PORT 10001 /* 服务器使用的端口号 */
#define BUF_SIZE 1024 /* 接收缓冲区的长度 */
#define BACKLOG 5 /* 请求队列的长度 */

static const char send_data[] = "This is TCP Server from RT-Thread.";

void tcpserver6(void)
{
    int sockfd, clientfd;
    struct sockaddr_in6 server_addr6, client_addr6;
    int bytes_received;
    char *recv_data;
    rt_uint32_t sin_size;
    rt_bool_t stop = RT_FALSE;

    /* 分配接收用的数据缓冲 */
    recv_data = rt_malloc(BUF_SIZE);
    if(recv_data == RT_NULL)
    {
        /* 分配内存失败,返回 */
        rt_kprintf("No memory\n");
        return ;
    }
}
```

```

/* 创建一个socket,family类型为PF_INET6,套接字类型为SOCK_STREAM */
if((sockfd = socket(PF_INET6, SOCK_STREAM, 0)) == -1)
{
    rt_kprintf("Socket error\n");
    rt_free(recv_data);
    return ;
}

/* 初始化服务端地址 */
server_addr6.sin6_family = AF_INET6;
memcpy(server_addr6.sin6_addr.s6_addr, IP6_ADDR_ANY, 16);
server_addr6.sin6_port = htons(SERV_PORT);

/* 绑定Socket到服务端地址 */
if(bind(sockfd, (struct sockaddr *)&server_addr6, sizeof(struct sockaddr)) == -1)
{
    rt_kprintf("Bind error\n");
    rt_free(recv_data);
    return ;
}

/* 在Socket上进行监听 */
if(listen(sockfd, BACKLOG) == -1)
{
    rt_kprintf("Listen error\n");
    rt_free(recv_data);
    return ;
}

rt_sprintf(recv_data, "%4d", SERV_PORT);
rt_kprintf("\nTCPServer Waiting for client on port %s...\n", recv_data);

while(stop != RT_TRUE)
{
    sin_size = sizeof(struct sockaddr_in6);
    /*接收客户端的连接*/
    clientfd = accept(sockfd, (struct sockaddr *)&client_addr6, &sin_size);
    rt_kprintf("I got a connection from (IP:%s, PORT:%d\n)", inet6_ntoa(client_addr6.sin6_addr), client_addr6.sin6_port);
    /* 客户端连接的处理 */
    while(1)
    {
        /* 发送数据到connected socket */
        send(clientfd, send_data, strlen(send_data), 0);

        /* 接收数据,但并不一定能够收到BUF_SIZE大小的数据 */

```

```

        bytes_received = recv(clientfd, recv_data, BUF_SIZE, 0);
        if(bytes_received <= 0)
        {
            /* 接收失败,关闭这个Connected Socket */
            closesocket(clientfd);
            break;
        }
        /* 收到数据,加入字符串结束符*/
        recv_data[bytes_received] = '\0';
        if(strcmp(recv_data, "q") == 0 || strcmp(recv_data, "Q") == 0)
        {
            /* 关闭连接 */
            closesocket(clientfd);
            break;
        }
        else if(strcmp(recv_data, "exit") == 0)
        {
            /* 关闭服务端 */
            closesocket(clientfd);
            stop = RT_TRUE;
            break;
        }
        else
        {
            /* 打印收到的数据 */
            rt_kprintf("RECEIVED DATA = %s\n", recv_data);
        }
    }

    closesocket(sockfd);
    rt_free(recv_data);

    return ;
}

#ifdef RT_USING_FINSH
#include <finsh.h>
FINSH_FUNCTION_EXPORT(tcpserver6, start tcp server via ipv6 );
#endif

```

14.4.4 TCP Client例子

下面则是另一个如在RT-Thread上使用BSD socket接口的TCP客户端例子。当把这个代码加入到RT-Thread时, 它会自动向finsh 命令行添加一个tcpclient6命令, 在finsh上执行tcpclient6()函数即可启动这个TCP服务端。

当TCP客户端连接成功时, 它会接收服务端传过来的数据。当有数据接收到时, 如果是

以q或Q开头，它将主动断开这个连接；否则会把接收的数据在控制终端中打印出来，然后发送“This is TCP Client from RT-Thread.”的字符串。

```
#include <rtthread.h>
#include <lwip/netdb.h> /*为解析主机名, 需包含netdb.h */
#include <lwip/sockets.h>

#define SERV_PORT 12345 /* 服务器端口号 */
#define SERVADDR "4006:e024:680:c6e:223:8bff:fe59:de90" /* 由于未实现Scope id, 请不要使用link-local */
#define BUF_SIZE 1024 /* 缓冲区的长度 */
static const char send_data[] = "This is TCP Client from RT-Thread.";

void tcpclient6(void)
{
    char* recv_data;
    int sockfd, bytes_received;
    struct sockaddr_in6 server_addr6;
    int status = 0;

    /* 分配接收用的数据缓冲 */
    recv_data = rt_malloc(BUF_SIZE);
    if(recv_data == RT_NULL)
    {
        /* 分配内存失败,返回 */
        rt_kprintf("No memory\n");
        return ;
    }

    /* 创建一个socket,family类型为PF_INET6,套接字类型为SOCK_STREAM */
    if((sockfd = socket(PF_INET6, SOCK_STREAM, 0)) == -1)
    {
        rt_kprintf("Socket error\n");
        rt_free(recv_data);
        return ;
    }

    /* 初始化预连接的服务端地址 */
    memset(&server_addr6, 0, sizeof(server_addr6));
    server_addr6.sin6_family = AF_INET6;
    server_addr6.sin6_port = htons(SERV_PORT);
    /* 将字符串转换为IPv6地址 */
    if(inet_pton(AF_INET6, SERVADDR, &server_addr6.sin6_addr.s6_addr) != 1)
    {
        rt_kprintf("inet_pton() error\n");
        rt_free(recv_data);
        return ;
    }
}
```

```

    }

    /* 连接到服务端 */
    status = connect(sockfd, (struct sockaddr *)&server_addr6, sizeof(server_addr6));
    if(status < 0)
    {
        /* 连接失败 */
        rt_kprintf("Connect error:%d\n", status);
        rt_free(recv_data);
        return ;
    }

    while(1)
    {
        /* 从socket连接中接收最大BUF_SIZE字节数据 */
        bytes_received = recv(sockfd, recv_data, BUF_SIZE -1, 0);
        if(bytes_received <= 0)
        {
            /* 接收失败, 关闭这个连接 */
            closesocket(sockfd);
            rt_free(recv_data);
            break;
        }
        /* 收到数据, 加入字符串结束符 */
        recv_data[bytes_received] = '\0';
        if(strcmp(recv_data, "q") == 0 || strcmp(recv_data, "Q") == 0)
        {
            /* 关闭这个连接 */
            closesocket(sockfd);
            rt_free(recv_data);
            break;
        }
        else
        {
            /* 打印收到的数据 */
            rt_kprintf("\nReceived data = %s ", recv_data);
        }
        /* 发送数据到服务端 */
        send(sockfd, send_data, strlen(send_data), 0);
    }

    return;
}

#ifdef RT_USING_FINSH

```

```
#include <finsh.h>
FINSH_FUNCTION_EXPORT(tcpclient6, startup tcp client via ipv6);
#endif
```

14.4.5 UDP Server例子

下面是一个在RT-Thread上使用BSD socket接口的UDP服务端例子，当把这个代码加入到RT-Thread操作系统时，它会自动向finsh命令行添加一个udpserver6命令，在finsh上执行udpserver6()函数即可启动这个UDP服务端，该UDP服务端在端口10012上进行监听。

当服务端接收到数据时，它将把数据打印到控制终端中；如果服务端接收到exit字符串时，那么服务端将退出服务。

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

#define SERV_PORT 10012
#define BUF_SIZE 1024

static const char send_data[] = "This is UDP Server from RT-Thread.";

void udpserver6(void)
{
    int sockfd;
    struct sockaddr_in6 server_addr6;
    struct sockaddr_in6 client_addr6;
    int bytes_read;
    char *recv_data;
    rt_uint32_t addr_len;

    /* 分配接收用的数据缓冲 */
    recv_data = rt_malloc(BUF_SIZE);
    if(recv_data == RT_NULL)
    {
        /* 分配内存失败, 返回 */
        rt_kprintf("No memory\n");
        return ;
    }

    /* 创建一个socket,family类型为PF_INET6,套接字类型为SOCK_DGRAM */
    if((sockfd = socket(AF_INET6, SOCK_DGRAM, 0)) == -1)
    {
        rt_kprintf("Socket error\n");
        rt_free(recv_data);
        return ;
    }

    /* 初始化服务端地址 */
```

```

server_addr6.sin6_family = AF_INET6;
server_addr6.sin6_port = htons(SERV_PORT);
memcpy(server_addr6.sin6_addr.s6_addr, IP6_ADDR_ANY, 16);

/* 绑定Socket到服务端地址 */
if(bind(sockfd, (struct sockaddr *)&server_addr6, sizeof(struct sockaddr)) == -1)
{
    /* 绑定地址失败 */
    rt_kprintf("Bind error\n");
    rt_free(recv_data);
    return ;
}
rt_sprintf(recv_data, "%4d", SERV_PORT);
rt_kprintf("UDPServer Waiting for client on port %s...\n", recv_data);

addr_len = sizeof(struct sockaddr);
while(1)
{
    /* 从socket中收取最大BUF_SIZE字节数据 */
    bytes_read = recvfrom(sockfd, recv_data, BUF_SIZE - 1, 0, (struct sockaddr *)&client_addr6, &client_addr_len);
    /* 收到数据,加入字符串结束符*/
    recv_data[bytes_read] = '\0';
    /* 输出接收的数据 */
    rt_kprintf("\n(%s, %d) said:", inet6_ntoa(client_addr6.sin6_addr), ntohs(client_addr6.sin6_port));
    rt_kprintf("%s", recv_data);

    if(strcmp(recv_data, "exit") == 0)
    {
        /* 关闭服务端 */
        closesocket(sockfd);
        rt_free(recv_data);
        break;
    }
}
return ;
}

#ifdef RT_USING_FINSH
#include <finsh.h>
FINSH_FUNCTION_EXPORT(udpserver6, startup udp server via ipv6);
#endif

```

14.4.6 UDP Client例子

下面是另一个在RT-Thread上使用BSD socket接口的UDP客户端例子。当把这个代码加入到RT-Thread时, 它会自动向finsh命令行添加一个udpclient6命令, 在finsh上执行

udpclient6()函数即可启动这个UDP客户端。当UDP客户端启动后，它将发送“This is UDP Client from RT-Thread.”的字符串给服务端，然后接收数据并打印数据。

```
#include <rtthread.h>
#include <lwip/netdb.h> /*为解析主机名, 需包含netdb.h */
#include <lwip/sockets.h>

#define SERV_PORT 22345 /* 服务器端口号 */
#define SERVADDR "4006:e024:680:c6e:223:8bff:fe59:de90" /* 由于未实现Scope id, 请不要使用lin
#define BUF_SIZE 1024 /* 缓冲区的长度 */
static const char send_data[] = "This is UDP Client from RT-Thread.";

void udpclient6(void)
{
    char *recv_data;
    int sockfd;
    struct sockaddr_in6 server_addr6, client_addr6;
    socklen_t clientlen;

    /* 分配接收用的数据缓冲 */
    recv_data = rt_malloc(BUF_SIZE);
    if(recv_data == RT_NULL)
    {
        rt_kprintf("No memory\n");
        return ;
    }

    /* 创建一个socket, family类型为PF_INET6, 套接字类型为SOCK_DGRAM */
    if((sockfd = socket(PF_INET6, SOCK_DGRAM, 0)) == -1)
    {
        rt_kprintf("Socket error\n");
        rt_free(recv_data);
        return ;
    }

    /* 初始化预连接的服务端地址 */
    memset(&server_addr6, 0, sizeof(server_addr6));
    server_addr6.sin6_family = AF_INET6;
    server_addr6.sin6_port = htons(SERV_PORT);
    /* 将字符串转换为IPv6地址 */
    if(inet_pton(AF_INET6, SERVADDR, &server_addr6.sin6_addr.s6_addr) != 1)
    {
        rt_kprintf("inet_pton() error\n");
        rt_free(recv_data);
        return ;
    }
}
```

```

/* 发送数据到服务端 */
if(sendto(sockfd, send_data, sizeof(recv_data), 0, (struct sockaddr *)&server_addr6, sizeof(recv_data)))
{
    rt_kprintf("Sendto error\n");
    rt_free(recv_data);
    return ;
}

rt_kprintf("Waiting for a reply...\n");

clientlen = sizeof(client_addr6);
/* 接收数据 */
if(recvfrom(sockfd, recv_data, BUF_SIZE, 0, (struct sockaddr *)&client_addr6, &clientlen) < 0)
{
    /* 接收失败 */
    rt_kprintf("Recvfrom error\n");
    rt_free(recv_data);
    return ;
}
/* 打印数据 */
rt_kprintf("got '%s'\n", recv_data);
closesocket(sockfd);
}
#ifdef RT_USING_FINSH
#include <finsh.h>
FINSH_FUNCTION_EXPORT(udpclient6, start udp server via ipv6);
#endif

```

第 15 章

POSIX接口

15.1 简介

[描述POSIX的历史情况，POSIX API情况]

[RT-Thread中对POSIX API的支持情况]

15.2 在RT-Thread中使用POSIX

在RT-Thread中使用POSIX API接口包括几个部分：libc（例如newlib），file system，pthread等。

需要在rtconfig.h中打开相关的选项：

```
#define RT_USING_LIBC
#define RT_USING_DFS
#define RT_USING_DFS_DEVFS
#define RT_USING_PTHREADS
```

15.3 POSIX Thread介绍

[API介绍]

15.3.1 栏杆: barrier

pthread_barrier 系列函数在中定义，用于多线程的同步，它包含三个函数：

```
--pthread_barrier_init()
--pthread_barrier_wait()
--pthread_barrier_destroy()
```

pthread_barrier_*实现一个类似栏杆的功能（barrier意为栏杆）。形象的说就是把先后到达的多个线程挡在同一栏杆前，直到所有线程到齐，然后撤下栏杆同时放行。其中：

- pthread_barrier_init函数负责指定要等待的线程个数；
- pthread_barrier_wait函数由每个线程主动调用，它告诉栏杆“我到起跑线前了”。pthread_barrier_wait函数执行末尾栏杆会检查是否所有人都到栏杆前了，如果是，栏杆就消失所有线程继

续执行下一句代码；如果不是，则所有已到pthread_barrier_wait函数的线程停在该函数不动，剩下没执行到pthread_barrier_wait函数的线程继续执行；

- pthread_barrier_destroy函数释放init申请的资源。

使用场景举例：

这种“栏杆”机制最大的特点就是最后一个执行wait的动作最为重要，就像赛跑时的起跑枪一样，它来之前所有人都必须等着。所以实际使用中，pthread_barrier_*常常用来让所有线程等待“起跑枪”响起后再一起行动。比如我们可以用pthread_create()生成100个线程，每个子线程在被create出的瞬间就会自顾自的立刻进入回调函数运行。但我们可能不希望它们这样做，因为这时主进程还没准备好，和它们一起配合的其它线程还没准备好，我们希望它们在回调函数中申请完线程空间、初始化后停下来，一起等待主进程释放一个“开始”信号，然后所有线程再开始执行业务逻辑代码。

解决方法：

为了解决上述场景问题，我们可以在init时指定n+1个等待，其中n是线程数。而在每个线程执行函数的首部调用wait()。这样100个pthread_create()结束后所有线程都停下来等待最后一个wait()函数被调用。这个wait()由主进程在它觉得合适的时候调用就好。最后这个wait()就是鸣响的起跑枪。

函数原型：

```
#include <pthread.h>
```

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,  
    const pthread_barrierattr_t *restrict attr, unsigned count);  
int pthread_barrier_wait(pthread_barrier_t *barrier);  
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

参数解释：

- pthread_barrier_t，是一个计数锁，对该锁的操作都包含在三个函数内部，我们不用关心也无法直接操作。只需要实例化一个对象丢给它就好。
- pthread_barrierattr_t，锁的属性设置，设为NULL让函数使用默认属性即可。
- count，你要指定的等待个数。

第 16 章

图像用户界面引擎

16.1 介绍

GUI engine是一套基本的绘图引擎，由C代码编写而成，代码主要放于rt-thread/components/gui/目录中。主要的功能包括：

- 基于绘图设备DC的绘图操作，包括点，线，圆，椭圆，多边形（填充）等；
- 各类图像格式加载（从文件系统中加载，需要DFS文件系统）及绘图；
- 各种字体的文本显示；
- GUI的C/S架构及基础的窗口机制，事件框架机制等。

当需要使用GUI engine时，需要在rtconfig.h中定义：

```
#define RT_USING_GUIENGINE
```

打开GUI engine。在rtconfig.h中打开GUI engine后，可以使用命令行重新生成工程，或使用scons来进行编译。

16.2 引擎初始化

在打开GUI engine后，需要在启动时对它进行初始化，把UI的服务打开起来，如果程序中已经使用了组件自动初始化，则不再需要额外进行单独的初始化，否则需要在初始化任务中调用初始化函数：

```
rtgui_system_server_init();
```

GUI engine初始化后，它会创建基础的GUI服务，包括任务：rtgui。虽然GUI engine是基础的引擎，但它依然还带了多窗口系统，绘图主要以GUI应用的模式而存在（创建应用，创建窗口（制定窗口位置，以及大小），在窗口中进行绘图，关闭窗口，删除应用）。

16.3 绘图设备上下文

为了划分不同的绘图区域（而不是直接和硬件打交道），在GUI engine中抽象出了DC（设备上下文，Device Context），而GUI engine中提供的绘图API基本上都是基于DC的API接口。DC根据不同的面向领域，主要包括这三类DC：

- Hardware DC，硬件设备上下文；
- Client DC，客户端设备上下文（每个窗口，控件都会携带一个Client DC）；
- Buffer DC，基于软件缓冲区的软件虚拟设备上下文；

16.4 绘图渲染

基于DC，UI引擎提供了基本的绘图操作，例如画点、线、矩形、圆、多边形等。这里统一的操作设备是DC，例如：

```
void rtgui_dc_draw_line(struct rtgui_dc *dc, int x1, int y1, int x2, int y2);
```

这个函数是在dc上绘制一条从 (x1, y1) - (x2, y2) 的线。线的颜色则是dc的图形上下文 (gc, graphic context) 中定义的前景色。

除了基本的绘图操作以外，GUI引擎也提供了DC上的字体文本绘制，例如：

```
void rtgui_dc_draw_text(struct rtgui_dc *dc, const char *text, struct rtgui_rect *rect);
```

这个函数就是在dc上，在由rect参数定义区域内显示text文本。文本颜色、字体依然是由图形上下文来定义的，并且图形上下文也定义了文本的对齐方式textalign，对齐方式取值是在RTGUI_ALIGN枚举类型中定义。

```
enum RTGUI_ALIGN
{
    RTGUI_ALIGN_NOT            = 0x00,
    RTGUI_ALIGN_CENTER_HORIZONTAL = 0x01,
    RTGUI_ALIGN_LEFT           = RTGUI_ALIGN_NOT,
    RTGUI_ALIGN_TOP             = RTGUI_ALIGN_NOT,
    RTGUI_ALIGN_RIGHT           = 0x02,
    RTGUI_ALIGN_BOTTOM          = 0x04,
    RTGUI_ALIGN_CENTER_VERTICAL = 0x08,
    RTGUI_ALIGN_CENTER          = RTGUI_ALIGN_CENTER_HORIZONTAL | RTGUI_ALIGN_CENTER_VERTICAL,
    RTGUI_ALIGN_EXPAND          = 0x10,
    RTGUI_ALIGN_STRETCH         = 0x20,
};
```

对于一个DC而言，可以通过以下方式获得DC的图形上下文：

```
rtgui_gc_t *rtgui_dc_get_gc(struct rtgui_dc *dc);
```

也可以通过以下简单的方式来访问DC的前景、背景、字体、对齐方式等信息：

```
#define RTGUI_DC_FC(dc)      (rtgui_dc_get_gc(RTGUI_DC(dc))->foreground)
#define RTGUI_DC_BC(dc)      (rtgui_dc_get_gc(RTGUI_DC(dc))->background)
#define RTGUI_DC_FONT(dc)    (rtgui_dc_get_gc(RTGUI_DC(dc))->font)
#define RTGUI_DC_TEXTALIGN(dc) (rtgui_dc_get_gc(RTGUI_DC(dc))->textalign)
```

同样的，一幅图像也可以在DC上渲染出来，例如使用以下API：

```
void rtgui_image_blit(struct rtgui_image *image, struct rtgui_dc *dc, struct rtgui_rect *rect);
```

这个API用于把一副image图像，绘制到dc的rect区域上，如果rect区域超出dc的范围，则会被自动裁剪。

16.5 基本的GUI引擎应用例子

16.6 事件传递机制

16.7 控件和剪切域

16.8 字体API

16.9 图像API

附录 A

电子书markdown入门

本章节附录主要描述电子书环境下书写markdown文件的规则，包括markdown本身的规则，也包括用于电子书而扩展的一些规则。

A.1 标题、段落、区块代码

电子书中的每个章节，章节标题行首都由一个 # 包围，例如：# 章节名称 #

每个章节文件有且仅有一个章节标题，其余的都是它的子标题。每个子标题由两个到六个 # 包围而成，从而形成标题2到标题6阶。

一个段落是由一个以上的连接的行句组成，而一个以上的空行则会划分出不同的段落（空行的定义是显式上看起来是空行，就被视为空行，例如有一行只有空白和 tab，那该行也会被视为空行），一般的段落不需要用空白或换行缩进。

在markdown电子书中不存在引用的情况，相替换的，建议使用代码方式风格来替换引用。代码风格可以在文本前加入4个空格，例如：

```
引用的区块#1
应用的区块#2
```

A.2 修辞和强调

Markdown 使用星号和底线来标记需要强调的区段。例如：

```
这是一个 **强调** 的文本，这是一个加 __底线__ 的文本。
```

这是一个 强调 的文本，这是一个加 底线 的文本。

A.3 列表

无序列表使用星号、加号和减号来做为列表的项目标记，这些符号是都可以使用的，使用星号：

```
* Candy.
* Gum.
* Booze.
```

加号：

```
+ Candy.
+ Gum.
+ Booze.
```

和减号

```
- Candy.
- Gum.
- Booze.
```

有序的列表则是使用一般的数字接着一个英文句点作为项目标记：

```
1. Red
2. Green
3. Blue
```

如果你在项目之间插入空行，那项目的内容会形成段落，可以在一个项目内放上多个段落，只要在它前面缩排 4 个空白或 1 个 tab。

```
* A list item.
With multiple paragraphs.

* Another item in the list.
```

- A list item. With multiple paragraphs.
- Another item in the list.

A.4 链接

Markdown 支援两种形式的链接语法：行内 和 参考 两种形式，两种都是使用角括号来把文字转成连结。

行内形式是直接后面用括号直接接上链接：

```
This is an [example link](http://example.com/).
```

实际效果如：

This is an [example link](http://example.com/).

你也可以选择性的加上 title 属性：

```
This is an [example link](http://example.com/ "With a Title").
```

实际效果如：

This is an [example link](http://example.com/).

参考形式的链接让你可以为链接定一个名称，之后你可以在文件的其他地方定义该链接的内容：

```
I get 10 times more traffic from [Google][1] than from  
[Yahoo][2] or [MSN][3].
```

```
[1]: http://google.com/ "Google"  
[2]: http://search.yahoo.com/ "Yahoo Search"  
[3]: http://search.msn.com/ "MSN Search"
```

实际效果如：

I get 10 times more traffic from [Google](http://google.com/) than from [Yahoo](http://search.yahoo.com/) or [MSN](http://search.msn.com/).

title 属性是选择性的，链接名称可以用字母、数字和空格，但是不分大小写：

```
I start my morning with a cup of coffee and  
[The New York Times][NY Times].
```

```
[ny times]: http://www.nytimes.com/
```

实际效果如：

I start my morning with a cup of coffee and [The New York Times](http://www.nytimes.com/).

A.5 图片

图片的语法和链接很像，同时图也可以选择一个标题，标题序号会在生成PDF时自动加上序号，例如：

```
![标题](../../figures/logo.png)
```

其中，请确保指向的图形文件在figures目录下存在，否则在生成PDF文件时会报错。



图 A.1: 标题

A.6 代码

在电子书中，当转换成PDF时，可以支持代码的语法高亮，可以使用如下的形式（也可以根据实际排版情况，在代码前加入4个或2个空格）：

```
```c
#include <stdio.h>

int main(int argc, char** argv)
{
 printf("hello\n");
 return 0;
}
```
```

它的效果类似于这样：

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("hello\n");
    return 0;
}
```

A.7 API说明

可以使用下面的格式来说明系统中的API接口。例如：

```
int func(int a, int b);
```

这个函数用于什么目的，完成了什么事。

函数参数

| 参数 | 描述 |
|-------|---------------|
| int a | a的意义是balabala |
| int b | b的意义是balabala |

这是一个用于什么的例子，例子的大致描述。

```
int result;  
  
/* 调用函数 */  
result = function(1, 2);
```

例子是否需要详细解释。

A.8 注意事项

在markdown向PDF转换过程中，pandoc会生成中间的LaTeX文件，如果markdown文本中（不是引用、代码中使用）使用了反斜杠，会导致转化PDF文件报错。所以最好的方式是使用双反斜杠，如下所示：

```
C:\\Python27路径balabala
```