

强实时嵌入式系统开发

——基于 RS-RTOS 开发强实时应用系统

Rev1.00

2007 年 1 月

致电子版读者

《强实时嵌入式系统开发》一书（以下简称本书）系作者辛勤劳动的成果，本着知识共享的开源精神，本书以电子版的形式免费提供阅读，请尊重作者的版权。为合法拥有本书电子版的拷贝，本书作者特与电子版读者约定以下条款：

- ◆ 条款 1：电子版读者必须加入本书的读者邮件列表，本邮件列表是免费的，邮件列表的地址和加入方法请参考网址 <http://www.RS-RTOS.org/books/>。
- ◆ 条款 2：本书电子版可以自由传播，但必须保持完整性，并且不得以之盈利。

如你接受以上条款，将视为你合法拥有本书电子版的拷贝，并且，你也应同时遵守包括但不限于（作者保留在修订版本中增删它们的权利）以下权利和义务。否则，请勿保留本书电子版拷贝。

读者权利：

- ◆ 通过本书读者邮件列表，读者可以获得本书配套例子所有源代码；
- ◆ 通过邮件列表，读者将继续免费获得本书电子版的修订版本；
- ◆ 本书所有读者和作者均通过邮件列表进行交流和问题讨论。

读者义务：

- ◆ 读者有义务协助作者查找、反馈本书中字、句、源代码等错漏；
- ◆ 读者有义务与其他读者一起讨论、解决在阅读、调试本书代码时产生的疑问；
- ◆ 读者有义务维护本书版权不被第三方侵犯。

作者：阮海深
2007 年 1 月

前 言

嵌入式系统泛指一切以计算机技术为核心的专用系统。而嵌入式操作系统是嵌入式系统的核心，它是包裹在硬件上的第一层软件平台，所有的应用都是依赖操作系统实现运行的。而嵌入式操作系统和应用的开发是分别进行的，应用开发者通常在第三方操作系统上完成他们的任务。那么，如何能够更好的利用操作系统的特性？用更短的时间完成产品的设计，这是应用开发所关心的问题。

本书试图以应用的角度，自上而下，对嵌入式操作系统作全面的介绍，理清在应用操作系统时常常遇到的问题和误区，使应用开发工程师使用操作系统时能操纵自如。我在从事嵌入式应用和操作系统开发过程中，发现了一个有趣的现象，硬件、操作系统、应用是嵌入式系统三大技术层，但彼此缺乏沟通。软件开发不能充分参与硬件设计中，往往增加了硬件成本，还使得软件设计更加复杂。应用开发也一样，因为对操作系统的运行机制不了解，不能充分发挥操作系统的特性，遇到应用与操作系统牵连的问题更是一筹莫展。另一方面，操作系统是一个十分复杂的系统，不可能也没有必要要求应用开发工程师了解其实现细节。而应该从运行机制、实现原理上观其大略，识其大体，这也是我写这本书的目的。那些对嵌入式操作系统有浓厚兴趣的人，则可以此为基础，继续深入研究源代码，定能事半功倍。

目 录

致电子版读者 1

前 言 3

目 录 5

第 1 章 建立调试环境 1

 1.1 构建调试环境 1

 1.2 示例代码 1

 1.3 调试示例 2

 1.4 RS-RTOS入口 6

 1.5 示例APP_SAMPLE_01A代码 8

第 2 章 RTOS基本概念 15

 2.1 什么是嵌入式系统 15

 2.2 实时系统 16

 2.3 任 务 16

 2.4 上下文切换 16

 2.5 任务调度 17

 2.6 非抢占式调度 17

 2.7 抢占式调度 17

 2.8 优先级 18

 2.9 资 源 18

 2.10 资源冲突 18

 2.11 死 锁 18

 2.12 中断处理 19

 2.13 线程饥饿 19

 2.14 优先级翻转 19

第3章 内核组件	21
3.1 介 绍	21
3.2 数据类型	22
3.3 任 务	23
3.4 中断管理	24
3.5 定时器	24
3.6 互斥量	25
3.7 信号量	25
3.8 二值信号量	26
3.9 事件标志	27
3.10 邮 箱	27
3.11 队 列	28
3.12 内存池集	29
第4章 内核服务	31
4.1 介 绍	31
4.2 内核服务综述	31
4.3 基础服务接口	35
4.4 建立对象名字	36
4.5 返回内核版本	36
4.6 返回内核版本字符串	37
4.7 锁定内核	37
4.8 解除内核锁	37
4.9 返回内核锁状态	38
4.10 获取内核信息	38
4.11 总 结	40
第5章 任 务	41
5.1 介 绍	41
5.2 任务控制块	41
5.3 重入函数	42

5.4 任务调用综述	43
5.5 创建任务	44
5.6 删除任务	51
5.7 更改任务优先级	53
5.8 更改任务运行优先级	54
5.9 重置运行优先级	57
5.10 识别任务	58
5.11 任务伪优先级	59
5.12 当前运行优先级	60
5.13 任务休眠	61
5.14 唤醒任务	62
5.15 挂起任务	63
5.16 恢复任务	65
5.17 任务保护	66
5.18 解除保护	69
5.19 获取任务信息	69
5.20 任务执行综述	71
5.21 任务状态	72
5.22 任务设计	73
5.23 任务的内部结构	73
5.24 总 结	73
第 6 章 互斥量	75
6.1 介 绍	75
6.2 保护临界区	75
6.3 共享资源的互斥访问	76
6.4 互斥量控制块	76
6.5 互斥量服务综述	77
6.6 创建互斥量	78
6.7 删除互斥量	79
6.8 获取（等待）互斥量	80

6.9 获取（无等待）互斥量.....	81
6.10 释放互斥量.....	83
6.11 获取互斥量信息.....	85
6.12 避免死锁	86
6.13 互斥量示例.....	88
6.14 示例APP_SAMPLE_06E代码.....	89
6.15 互斥量的内部结构.....	94
6.16 总 结	95
第 7 章 信号量.....	97
7.1 介 绍	97
7.2 信号量属性.....	98
7.3 避免死锁	99
7.4 防止优先级翻转	99
7.5 信号量服务综述	99
7.6 创建信号量.....	100
7.7 删除信号量.....	101
7.8 获取（等待）信号量	102
7.9 获取（无等待）信号量.....	103
7.10 释放信号量.....	105
7.11 获取信号量信息.....	106
7.12 信号量与互斥量异同	108
7.13 生产者—消费者问题	109
7.14 信号量内部结构	111
7.15 总 结	112
第 8 章 二值信号量.....	113
8.1 介 绍	113
8.2 二值信号量属性	114
8.3 二值信号量服务综述	114
8.4 创建二值信号量	115

8.5 删除二值信号量	116
8.6 获取（等待）二值信号量	116
8.7 获取（无等待）二值信号量	118
8.8 释放二值信号量	119
8.9 获取二值信号量信息	121
8.10 使用二值信号量代替互斥量	122
8.11 二值信号量内部结构	128
8.12 总 结	128
第 9 章 邮 箱	131
9.1 介 绍	131
9.2 邮箱消息	132
9.3 邮箱属性	132
9.4 邮箱服务综述	133
9.5 创建邮箱	133
9.6 删除邮箱	134
9.7 发送邮箱消息	136
9.8 接收邮箱消息	137
9.9 清空邮箱	138
9.10 间接邮箱消息	139
9.11 示例APP_SAMPLE_09E代码	140
9.12 邮箱内部结构	146
9.13 总 结	146
第 10 章 队 列	149
10.1 介 绍	149
10.2 队列属性	150
10.3 队列服务综述	151
10.4 创建队列	151
10.5 删除队列	155
10.6 发送队列消息	156

10.7 接收队列消息	157
10.8 清空队列	159
10.9 间接队列消息	160
10.10 示例APP_SAMPLE_10F代码	161
10.11 队列内部结构	166
10.12 总 结	167
第 11 章 事件标志	169
11.1 介 绍	169
11.2 事件标志中的事件	170
11.3 事件标志属性	172
11.4 事件标志服务综述	172
11.5 创建事件标志	173
11.6 删除事件标志	174
11.7 获取（等待）事件	175
11.8 获取（无等待）事件	179
11.9 设置事件	180
11.10 获取事件标志信息	181
11.11 使用事件标志实现同步	182
11.12 示例APP_SAMPLE_11G代码	184
11.13 事件标志内部结构	188
11.14 总 结	189
第 12 章 时 钟	191
12.1 介 绍	191
12.2 时钟服务综述	192
12.3 获取系统节拍	192
12.4 设置系统节拍	192
12.5 总 结	193
第 13 章 定时器	195
13.1 介 绍	195

13.2 定时器属性.....	196
13.3 定时器服务综述.....	197
13.4 创建定时器.....	198
13.5 删除定时器.....	200
13.6 启动定时器.....	201
13.7 停止定时器.....	202
13.8 定时器控制.....	203
13.9 定时器内部结构	204
13.10 总 结	205
第 14 章 中 断.....	207
14.1 介 绍.....	207
14.2 多重中断	208
14.3 中断向量	209
14.4 中断服务综述.....	210
14.5 注册中断服务.....	211
14.6 注销中断服务.....	212
14.7 允许中断	213
14.8 禁止中断	214
14.9 捕获中断	215
14.10 总 结	217
第 15 章 内存管理	219
15.1 介 绍.....	219
15.2 内存碎片	219
15.3 内存池集	221
15.4 内存管理服务综述.....	222
15.5 申请内存池.....	223
15.6 释放内存池.....	224
15.7 获取内存池信息	225
15.8 内存池集内部结构.....	227

15.9 总 结	228
第 16 章 设备管理.....	229
16.1 介 绍	229
16.2 设备驱动	230
16.3 设备管理服务综述.....	231
16.4 创建设备标识	231
16.5 安装设备驱动	232
16.6 卸载设备驱动	235
16.7 设备初始化.....	236
16.8 打开设备	237
16.9 关闭设备	238
16.10 设备读	239
16.11 设备写	241
16.12 设备控制	243
16.13 总 结	244
附 录 RS-KERNEL API服务接口.....	247
附录A 初始化服务	248
A.1 <i>hardware_initialize</i>	248
A.2 <i>application_initialize</i>	249
附录B 基础服务	250
B.1 <i>build_name</i>	250
B.2 <i>kernel_version</i>	251
B.3 <i>kernel_version_s</i>	252
B.4 <i>kernel_lock</i>	253
B.5 <i>kernel_unlock</i>	254
B.6 <i>kernel_islock</i>	255
B.7 <i>kernel_info</i>	256
附录C 任务服务	257
C.1 <i>task_create</i>	257

C.2	<i>task_delete</i>	258
C.3	<i>task_change_prio</i>	259
C.4	<i>task_change_runprio</i>	260
C.5	<i>task_reset_runprio</i>	261
C.6	<i>task_ident</i>	262
C.7	<i>task_self</i>	263
C.8	<i>task_current</i>	264
C.9	<i>task_sleep</i>	265
C.10	<i>task_wake</i>	266
C.11	<i>task_suspend</i>	267
C.12	<i>task_resume</i>	268
C.13	<i>task_protect</i>	269
C.14	<i>task_unprotect</i>	270
C.15	<i>task_info</i>	271
附录D	时钟服务	272
D.1	<i>tick_get</i>	272
D.2	<i>tick_set</i>	273
附录E	中断服务.....	274
E.1	<i>interrupt_attach</i>	274
E.2	<i>interrupt_detach</i>	275
E.3	<i>interrupt_enable</i>	275
E.4	<i>interrupt_disable</i>	276
E.5	<i>interrupt_catch</i>	277
附录F	定时器服务	278
F.1	<i>timer_create</i>	278
F.2	<i>timer_delete</i>	280
F.3	<i>timer_enable</i>	281
F.4	<i>timer_disable</i>	282
F.5	<i>timer_control</i>	283
附录G	互斥量服务.....	284
G.1	<i>mutex_create</i>	284

G.2	<i>mutex_delete</i>	285
G.3	<i>mutex_wait</i>	286
G.4	<i>mutex_trywait</i>	287
G.5	<i>mutex_release</i>	288
G.6	<i>mutex_info</i>	289
附录H	信号量服务.....	290
H.1	<i>semaphore_create</i>	290
H.2	<i>semaphore_delete</i>	291
H.3	<i>semaphore_wait</i>	292
H.4	<i>semaphore_trywait</i>	293
H.5	<i>semaphore_post</i>	294
H.6	<i>semaphore_info</i>	295
附录I	二值信号量服务.....	296
I.1	<i>sembinary_create</i>	296
I.2	<i>sembinary_delete</i>	297
I.3	<i>sembinary_wait</i>	298
I.4	<i>sembinary_trywait</i>	299
I.5	<i>sembinary_post</i>	300
I.6	<i>sembinary_info</i>	301
附录J	邮箱服务.....	302
J.1	<i>mailbox_create</i>	302
J.2	<i>mailbox_delete</i>	303
J.3	<i>mailbox_send</i>	304
J.4	<i>mailbox_receive</i>	305
J.5	<i>mailbox_flush</i>	306
附录K	队列服务.....	307
K.1	<i>queue_create</i>	307
K.2	<i>queue_delete</i>	308
K.3	<i>queue_send</i>	309
K.4	<i>queue_receive</i>	310
K.5	<i>queue_flush</i>	311

附录L 事件标志服务	312
L.1 <i>event_create</i>	312
L.2 <i>event_delete</i>	313
L.3 <i>event_wait</i>	314
L.4 <i>event_trywait</i>	315
L.5 <i>event_post</i>	316
L.6 <i>event_info</i>	317
附录M 内存服务	318
M.1 <i>mpool_alloc</i>	318
M.2 <i>mpool_free</i>	319
M.3 <i>mpool_info</i>	320
附录N 设备管理服务	321
N.1 <i>build_device</i>	321
N.2 <i>driver_install</i>	322
N.3 <i>driver_uninstall</i>	323
N.4 <i>device_init</i>	324
N.5 <i>device_open</i>	325
N.6 <i>device_close</i>	326
N.7 <i>device_ioctl</i>	327
N.8 <i>device_read</i>	328
N.9 <i>device_write</i>	329

第 1 章

建立调试环境

1.1 构建调试环境

RS-RTOS 支持多种编译环境，包括通用的 Unix/Linux gcc 编译器，以及特定的开发环境如 TI CCS2000。这些编译器大多针对特定芯片环境，在编辑调试方面限制较多，特别对于初学者，使用起来并不十分顺手。幸运的是，RS-RTOS 直接支持 Microsoft Visual C/C++ 系列开发平台。这样，基于 RS-RTOS 的应用可以直接在 WIN32 平台上进行编译调试，这使得开发人员能够在易于使用，并非常流行的 Windows 环境下对其目标应用进行原型设计。在本书的讨论中，也将使用 Microsoft Visual C/C++ Version 6.0 作为调试平台。通过 RS-RTOS 完善的 WIN32 仿真环境，让面向 RS-RTOS 的应用程序在 WIN32 上和最终目标硬件以高度一致的方式运行。这样带来的好处是，可以在实际目标硬件就绪前，就能很好的开展嵌入式软件的开发和调试工作，为项目争取宝贵的时间。

1.2 示例代码

在本书附带光盘有一个 RS-RTOS 演示版本，读者可以通过查看其中的 Readme.txt 文件，了解示例的安装和使用信息。

First Out, LIFO) 的模式进行, 因此称为任务栈。每个任务栈需要一个连续的内存空间, 可以从内存池中分配得到, 也可以通过定义一个全局的数组获得。前一方法在任务结束时, 任务栈占用的内存空间可以被内存池回收, 称为动态内存分配; 后一方法则称为静态分配。当系统任务数目比较多, 并且不固定时, 使用动态分配的任务栈比较灵活; 反之, 如系统任务数目固定, 则使用静态分配的任务栈比较简单。此外, 内存池也可以用于其他的 RS-RTOS 内核对象。在该示例中使用定义全局数组的方式。

在示例中还使用到另一个内核对象——互斥量 (Mutex), 其用途是防止两个任务在并行运行的时候, 同时使用到一个共有的 LED 数码管, 同时抢占资源就会产生冲突。在这个例子中, 任务 TA1 控制 LED 以 1Hz 的频率闪烁, 持续时间 5 秒, 任务 TA2 的控制 LED 以 1Hz 的频率闪烁, 持续时间 4 秒。显然, 当任务 TA1 在使用 LED 的时候, 不希望任务 TA2 打扰; 同样, 任务 TA2 在使用 LED 的时候, 也不希望被 TA1 打断。如果没有适当的限制措施, 这两个任务对共有资源 LED 的使用是无序的, 得到的结果是两个任务对 LED 操作叠加的效果。

为了得到正确的输出结果, 可以这样规定: 对数码管资源的操作代码, 在同一时刻只能由一个任务执行。这样, 一个任务对数码管输出时, 另一个任务就不能同时进行这项操作, 因而避免了冲突。习惯上, 这种对公共资源操作的代码称为临界区。解决临界区冲突的方法有很多, 使用互斥量是其中一个非常有效的方法。

互斥量是一个类似令牌或者门卫的对象。要进入临界区, 任务必须拥有互斥量的所有权, 而一旦互斥量被一个任务拥有, 别的任务就不能拥有该互斥量。任何时刻, 只能有一个任务能够拥有互斥量, 故互斥量又被称为互斥锁。利用互斥量这个性质来实现共享资源的互斥保

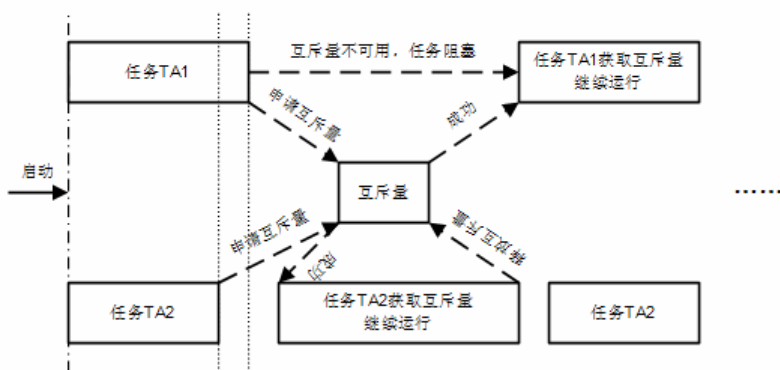


图1.3 互斥量性质

护。比如，如果 TA2 拥有了互斥量，尽管任务 TA1 的优先级比较高，任务 TA1 进入临界区时也必须等待，直到任务 TA2 释放互斥量的所有权。任务一旦获得互斥量的所有权，它将一直持有该所有权，除非任务自己主动释放该互斥量。也就是说，即使存在着优先级的差别，任何任务也不能抢占被其他任务持有的互斥量。这个特性对于实现任务间的互斥至关重要。

示例中的两个任务都是一个不断重复的活动过程，其结构是一个无穷循环：

【例 APP_SAMPLE_01A】

file: example\WIN32\app_sample_01a.cpp

```
.....
00133     for (;;)
00134     {
            .....
00160     }
.....
```

这是嵌入式系统任务的典型形式。节 1.5 给出了该示例的代码清单，包含了任务 TA1 和任务 TA2 入口函数的全部源程序。示例中 L00141~L00152 (L00141 表示行 00141)，L00179~L00190 分别是任务 TA1，TA2 对 LED 操作的临界段代码，任务通过下面的语句查询并获得互斥量 MU1 的所有权：

【例 APP_SAMPLE_01A】

file: example\WIN32\app_sample_01a.cpp

```
.....
00135         /**
00136         * 获取互斥量 MU1 的控制权,如互斥量无效,
00137         * 将任务进入等待状态. */
00138         mutex_wait(&led_mu1, RS_WAIT_FOREVER);
.....
```

mutex_wait 是 RS-RTOS 内核提供的服务接口，如果互斥量所有权已经被别的任务获得，任务将一直等待（通过参数 RS_WAIT_FOREVER 指出），直到获得互斥量所有权。当任务完成资源的访问，将通过下面语句释放对互斥量 MU1 的持有权：

【例 APP_SAMPLE_01A】

file: example\WIN32\app_sample_01a.cpp

```
.....
00154      /* 释放互斥量 MU1 的控制权. */
00155      mutex_release(&led_mu1);
.....
```

这条语句被执行之后，任务释放了互斥量 **MU1**，使之恢复可用状态，如果其他任务等待这个互斥量，它将有机会获得互斥量的所有权。

系统启动时,较高优先级的任务 TA1 首先获得运行,遇到临界段 L00138,通过 mutex_wait 申请互斥量 MU1,此时互斥量 MU1 可用,任务 TA1 将取得 MU1 所有权,进入临界段,控制 LED 以 1Hz 的频率闪烁,并维持 5 秒。在 L00146、L00151 处,任务 TA1 调用 task_sleep 服务进入休眠状态,释放出处理器使用权。任务 TA2 将有机会获得处理器使用权,当任务 TA2 要进入临界段时(L00176 处),申请互斥量 MU1,但此时 MU1 所有权已经被任务 TA1 获得,任务 TA2 进入阻塞状态。一直到任务 TA1 释放互斥量 MU1 (在 L00155 处),任务 TA2 才能获得 MU1 的所有权。在互斥量 MU1 干预下,任务 TA1, TA2 有秩序的完成对 LED 显示过程,图 1.4 给出了这个活动流程。

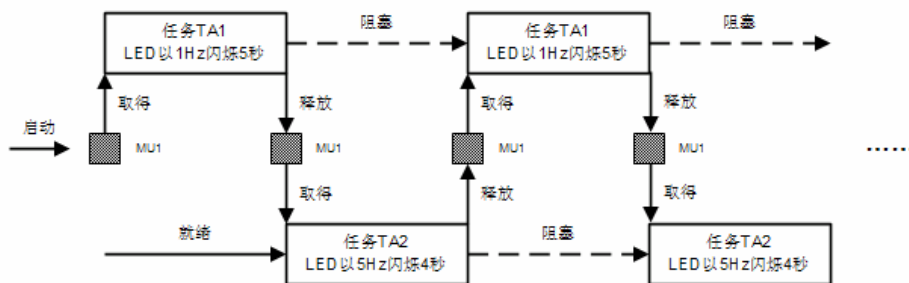


图1.4 示例执行过程

本书附带光盘中还给出了另一个示例 APP_SAMPLE_01B，这个例子将互斥量 MU1 从例子 APP_SAMPLE_01A 中去除，我们可以通过对比这两个例子的运行结果，加深理解。

提示: 如何切换到示例 APP_SAMPLE_01B?

在 VC6.0 工程文件视图中, 展开文件夹 **External Dependencies**, 找到 **app_sample.h**,

或者直接在 RS-RTOS 的源代码目录 `example\WIN32\` 中找到它，这是示例包含的头文件，找到以下这一行：

```
#define APP_SAMPLE_01A
```

更改为：

```
#define APP_SAMPLE_01B
```

按 F5 重新编译运行，即可看到示例 `app_sample_01b` 结果，宏 `APP_SAMPLE_01B` 就是对应例子的名字大写，本书所有的例子都可以通过这个方法切换。

1.4 RS-RTOS 入口

我们在编写 c 语言程序时，总是从 `main` 函数开始执行。RS-RTOS 上的应用程序，情形就不同了。在例子 `app_sample_01a` 中，有两个非常重要的函数：

【例 APP_SAMPLE_01A】

file: `example\WIN32\app_sample_01a.cpp`

```
.....
00081 /* 硬件初始化 */
00082 void hardware_initialize(void)
00083 {
00084
00085 }
.....
00088 /* 应用初始化 */
00089 void application_initialize(void)
00090 {
.....
00123 }
.....
```


这两个函数由应用编写，被 RS-RTOS 系统调用。不能直接在应用中调用这两个函数，就像不直接调用 `main` 函数一样，只要实现它们就可以了。读者根据这两个函数名字，大概也能猜出其作用：`hardware_initialize` 用来初始化硬件，而 `application_initialize` 就是应用的入口点，它的地位相当于 PC 程序的 `main` 函数，是整个应用程序运行的起始点。关于这两个函数，还将在后面的章节中详细讨论。现在只要记住，`application_initialize` 就是应用的入口点就够了。

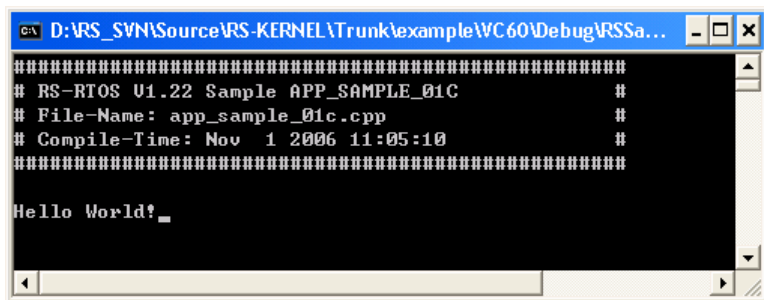
示例 `app_sample_01c` 显示了一个最简单的 Hello World! 例子，程序清单在下面列出，在随书配套光盘也能找到。当然，这个例子没有任何应用价值，仅仅是为了理解嵌入式系统应用与传统的 PC 程序的异同，在很多程序开发人员的脑子里，PC 程序模式已经是根深蒂固的了。

【例 APP_SAMPLE_01C】

file: example\WIN32\app_sample_01c.cpp

```
.....  
00056 void application_initialize(void)  
00057 {  
.....  
00061     printk("Hello World!");  
00062 }  
.....
```

例子 `app_sample_01c` 中，并没有创建任何内核对象，只是简单输出了一行字符串信息，结果如下：



```
G:\ D:\RS_SVN\Source\RS-KERNEL\Trunk\example\WC60\Debug\RSSa...  
#####  
# RS-RTOS V1.22 Sample APP_SAMPLE_01C #  
# File-Name: app_sample_01c.cpp #  
# Compile-Time: Nov 1 2006 11:05:10 #  
#####  
  
Hello World!_
```

当信息打印完毕，`application_initialize` 函数返回后，RS-RTOS 系统将进入“空闲状态”，这个状态被称为：IDLE LOOP。

在实际应用中，通常利用 `application_initialize` 创建若干应用任务，由各个任务完成应用功能。通过这种方法将复杂的系统划分为多个简单的子模块，这是应用嵌入式操作系统的一个重要意义。

1.5 示例 APP_SAMPLE_01A 代码

【例 APP_SAMPLE_01A】

file: example\WIN32\app_sample_01a.cpp

```
.....
00038 #include "inc/kapi.h"
00039 #include "app_sample.h"
00040
00041 #ifdef APP_SAMPLE_01A
00042
00043 /**
00044  * 例子 APP_SAMPLE_01A 初始化创建了两个任务,分别以不同频率
00045  * 控制 LED 闪烁.任务通过使用互斥量获取对 LED 控制权,并对 LED
00046  * 维持一定时间的控制输出,成功运行该程序将看到 LED 指示灯
00047  * 以不同频率交替闪烁一段时间.
00048  */
00049
00050
00051 /* 任务栈大小 */
00052 #define APP_STACK_SIZE      1024 * 32
00053
00054
00055 enum {
00056     /* 互斥量优先级 */
```

```
00057     APP_PRIO_MU1     = 0,
00058
00059     /* 任务优先级 */
00060     APP_PRIO_TA1     = 1,
00061     APP_PRIO_TA2     = 2,
00062 };
00063
00064
00065 /* LED 控制 */
00066 #define LED_OFF()          printk("\r  - - ")
00067 #define LED_ON()           printk("\r  @ @ ")
00068
00069 /* 任务栈 */
00070 stack_t app_stack_ta1[APP_STACK_SIZE];
00071 stack_t app_stack_ta2[APP_STACK_SIZE];
00072
00073 /* 信号量 */
00074 mutex_t led_mu1;
00075
00076
00077 void app_task_1(arg_t arg);
00078 void app_task_2(arg_t arg);
00079
00080
00081 /* 硬件初始化 */
00082 void hardware_initialize(void)
00083 {
00084
00085 }
00086
00087
```

```
00088 /* 应用初始化 */
00089 void application_initialize(void)
00090 {
00091     status_t status;
00092
00093     /* 打印关于示例一些信息 */
00094     APP_SAMPLE_INFO;
00095
00096     /* 创建任务 TA1 */
00097     status = task_create(
00098         APP_PRIO_TA1,
00099         build_name('T', 'A', '1', '\0'),
00100         app_task_1, 0,
00101         app_stack_ta1, APP_STACK_SIZE,
00102         0);
00103
00104     ASSERT(status == RS_EOK);
00105
00106     /* 创建任务 TA2 */
00107     status = task_create(
00108         APP_PRIO_TA2,
00109         build_name('T', 'A', '2', '\0'),
00110         app_task_2, 0,
00111         app_stack_ta2, APP_STACK_SIZE,
00112         0);
00113
00114     ASSERT(status == RS_EOK);
00115
00116     /* 初始化互斥量 MU1 */
00117     status = mutex_create(
00118         &led_mu1,
```

```
00119         build_name('M', 'U', '1', '\0'),
00120         APP_PRIO_MU1);
00121
00122     ASSERT(status == RS_EOK);
00123 }
00124
00125 /* 任务 TA1 入口 */
00126 void app_task_1(arg_t arg)
00127 {
00128     int i;
00129
00130     /* 防止编译器警告 */
00131     arg = arg;
00132
00133     for (;;)
00134     {
00135         /**
00136          * 获取互斥量 MU1 的控制权,如互斥量无效,
00137          * 将任务进入等待状态. */
00138         mutex_wait(&led_mu1, RS_WAIT_FOREVER);
00139
00140         /* LED 以 1Hz 频率闪烁,维持时间 5s */
00141         for (i = 0; i < 5; i++)
00142         {
00143             /* 点亮 LED. */
00144             LED_ON();
00145             /* 延时 500ms. */
00146             task_sleep(RS_TICK_FREQ * 1 / 2);
00147
00148             /* 关闭 LED. */
00149             LED_OFF();
```

```
00150          /* 延时 500ms. */
00151          task_sleep(RS_TICK_FREQ * 1 / 2);
00152      }
00153
00154      /* 释放互斥量 MU1 的控制权. */
00155      mutex_release(&led_mu1);
00156
00157      /* 休眠 1 时钟单位,
00158       * 这将允许低优先级的任务有机会运行. */
00159      task_sleep(1);
00160  }
00161 }
00162
00163 /* 任务 TA2 入口 */
00164 void app_task_2(arg_t arg)
00165 {
00166     int i;
00167
00168     /* 防止编译器警告 */
00169     arg = arg;
00170
00171     for (;;)
00172     {
00173         /**
00174          * 获取互斥量 MU1 的控制权,如互斥量无效,
00175          * 将任务进入等待状态. */
00176         mutex_wait(&led_mu1, RS_WAIT_FOREVER);
00177
00178         /* LED 以 5Hz 频率闪烁,维持时间 4s */
00179         for (i = 0; i < 25; i++)
00180         {
```

```
00181      /* 点亮 LED. */
00182      LED_ON();
00183      /* 延时 100ms. */
00184      task_sleep(RS_TICK_FREQ * 1 / 10);
00185
00186      /* 关闭 LED. */
00187      LED_OFF();
00188      /* 延时 100ms. */
00189      task_sleep(RS_TICK_FREQ * 1 / 10);
00190  }
00191
00192      /* 释放互斥量 MU1 的控制权. */
00193      mutex_release(&led_mu1);
00194
00195      /* 休眠 1 时钟单位,
00196      * 这将允许低优先级的任务有机会运行. */
00197      task_sleep(1);
00198  }
00199 }
00200
00201
00202 #endif
.....
```


第 2 章

RTOS 基本概念

嵌入式系统无疑是当今最热门的技术术语之一。其在制造工业、过程控制、通讯、仪器、仪表、汽车、船舶、航空、航天、军事装备、消费电子等方面广泛应用，取得了令人瞩目的成就，同时极大的促进了嵌入式系统及其相关技术的发展。

2.1 什么是嵌入式系统

嵌入式系统是一个以计算机软硬件技术为基础，并附以为特定应用的模块（机械，生物电子等），针对特定应用的系统。

与通用计算机系统不同，嵌入式系统针对特定的领域，因而在功耗、体积、成本、可靠性、速度、处理能力、电磁兼容性等方面均受到应用要求的制约。然而随着嵌入式技术的发展，有越来越多的通用计算机技术在嵌入式领域获得应用，嵌入式系统与通用计算机系统之间的界限逐渐变得模糊，但还是可以从以下特征区分这两种系统：

- 1) 是否是以微处理器为控制或处理核心；
- 2) 是否采用了计算机软件架构；
- 3) 是否针对特定应用领域。

具备以上特性的系统就可以称之为嵌入式系统。如一个以 AT89C51 为控制核心的数据采集系统，具备了第 1 和第 3 个特征。是否可称为嵌入式系统呢？取决于是否具备第 2 个特征，如果这个系统的应用软件采用传统的软件设计理念，直接在裸片上编写应用代码，这个系统只能被称为单片机系统；如果软件设计是以内嵌操作系统做为运行平台，则这个系统具备了

嵌入式系统的典型特征。

2.2 实时系统

嵌入式系统通常运行在特定的时间约束下。有时间约束的嵌入式系统称为实时系统（**Real-Time System**）。实时意味着系统必须在预定的时间限制内对输入事件作出响应并输出结果。因此，一个实时系统不但要得到正确的结果，还要在符合的时间内得到正确的结果。

实时约束有两种类型：强实时（或硬实时 **Hard Real-Time**）和弱实时（或软实时 **Soft Real-Time**）。强实时是指实时约束要求严格，即使只出现超时都是不可接受的，因为可能会导致灾难性的系统错误。强实时系统的例子有卫星发射系统、导弹控制系统、汽车紧急刹车系统、机床动力控制系统。弱实时系统的时间约束不像强实时系统那样苛刻，只要求尽可能的符合预定的时间即可，大多数消费类应用均属于这种系统。

2.3 任务

任务是具有完整独立的运行轨迹和确定的运行环境的逻辑集合。任务包含处理器运行轨迹与一组特定处理器寄存器状态集合，从而可确定处理器运行状态。打个比方，任务好比旅客，具有当前的位置和目的地，而处理器好比出租车。旅客在恰当的时机截获出租车，通过出租车达到目的地，而出租车因为承载乘客才有确定的运行轨迹。不同的乘客通过同一个出租车达到各自的目的地。

在操作系统的管理下，一个处理器可以同时运行多个不同的任务，这样的系统称为多任务系统，而在同一时刻，只能有一个任务占有当前处理器，占有处理器的任务称为当前任务，等待处理器资源的任务称为就绪任务，还有一些暂时不能运行称为阻塞任务。

2.4 上下文切换

上下文是任务当前的执行状态。包括一组用以指示当前处理器运行状态的寄存器信息，通常，它包含程序地址寄存器（**Program Counter, PC**），用以保存计算过程的通用寄存器，专有寄存器，存放堆栈地址的堆栈指针（**Stack Pointer, SP**）。上下文切换（**Context Switch**）

是指保存一个任务的上下文和恢复另一个任务的上下文，以便处理器从一个任务切换到另一个任务运行。

上下文切换通常发上在对处理器的抢占控制、中断处理、时间片切换、任务挂起或者任务申请暂时不可用的资源而被阻塞等事件。当一个任务被停止，它的上下文就会被保存起来，当一个任务的上下文被恢复，它就会从被停止的地方继续运行。内核负责完成上下文的保存和恢复操作。在上下文切换中需要完成的实际指令因处理器而异。

2.5 任务调度

任务只有在得到处理器资源之后才能真正获得运行。一个就绪的任务怎样才能获得处理器的控制权呢？多个任务之间如何分配有限的处理器资源呢？这是由任务调度实现的。任务调度的工作就是完成任务从就绪状态到运行状态的转化，协调多个就绪任务对处理器的控制以及运行时间。任务调度的基本方式可分为非抢占方式和抢占方式。

2.6 非抢占式调度

在非抢占式调度（Nonpreemptive）方式下，一旦一个任务被选中运行，它就一直运行下去，直到它完成工作，自愿释放 CPU，或者因等待某一个事件而被阻塞为止，才把 CPU 出让给其他任务。即得到 CPU 的任务不管要运行多长时间，都一直运行下去，决不会因为时钟中断或者外部事件等原因而被迫让出 CPU。

2.7 抢占式调度

与非抢占式调度相反，抢占式（Preemptive）允许调度程序根据某种策略中止当前运行任务的执行，将其移入就绪队列，并选择另一个任务投入运行。出现抢占调度的情况有：创建一个新任务，任务由阻塞转变为就绪状态等。

抢占式调度比非抢占式调度的开销大，其好处是可以防止一个任务长期占用处理器，高优先级的任务可以及时得到处理器资源，获得更佳的整体性能。

2.8 优先级

在任务调度时，系统采用优先级作为任务之间重要性的标准。优先级一般用某个固定范围内的整数表示，例如 0~255。优先级数值与任务优先级别的关系因系统而已，在有些系统中优先级数值越大，表示的优先级别越高；而另一些系统则恰恰相反，优先级数值越小，优先级别越高。如 RS-RTOS 系统就是采用优先级数值越小表示优先级别越高的方式。

任务优先级有静态方式和动态方式两种：静态优先级是在任务创建时确定下来，在运行期间保持不变；而动态优先级则可以在任务运行期更改。

2.9 资源

任何为应用程序所占用的实体都可称为资源。例如，内存中存放数据的变量、数组、结构体、处理器的中断资源。此外，还有常见的输入输出设备，包括打印机、键盘、鼠标、显示器、硬盘等，都是一种资源。

2.10 资源冲突

可以被一个以上任务使用的资源叫做共享资源（Shared Resource）。在同一时刻只能允许一个任务使用的资源称为临界资源（Critical Resource）或者互斥资源（Mutual Resource）。多个任务使用共享的互斥资源时，如发生一个以上的任务同时使用一个互斥资源，就会引发资源冲突。资源冲突或者引发输出错误的结果，或者破坏重要的数据，应该避免发生这种情况。为了防止数据被破坏，每个任务在与共享资源打交道时，必须独占该资源，这叫做互斥访问。

2.11 死锁

所谓死锁，是指在一个任务集合中的每个任务都在等待仅由该集合中的另一个任务才能引发的事件而无限期的僵持下去的状态。在多数情况下，任务是在等待该集合中的另一个任

务释放其所占有的资源。也就是说，每个任务都在期待获得另一个任务正在占用的资源。由于集合中所有的任务都不能运行，因此谁也不会释放资源。

死锁主要发生在大型多任务系统中，在嵌入式系统中比较少见。

2.12 中断处理

嵌入式应用的一个重要需求是对同步事件（比如硬件中断和软件中断）提供快速响应的能力。当中断发生时，当前执行的任务上下文被保存，控制权转交到对应的中断向量。所谓中断向量，是指中断服务例程（Interrupt Service Routine, ISR）的入口地址。通常 ISR 是用户编写的用于提供特殊中断处理和服务的软件。系统中一般会有多个 ISR，取决于多少个中断需要处理。

2.13 线程饥饿

基于优先级的抢占式调度算法会出现一种称为线程饥饿（Starvation）的危险。这种情况是指由于处理器实际被大量花费在较高优先级的任务上，导致较低优先级的任务几乎没有机会被执行。解决这个问题一个办法是：确保较高优先级的任务不会垄断对处理器的使用；另一个办法是，通过逐渐提高饥饿任务的优先级，以便他们能有机会执行。

2.14 优先级翻转

如果两个不同优先级的任务共享了一个公共资源，有时候会出现一种叫优先级翻转（Priority inversion）情况——当较低优先级的任务已经获得了较高优先级也需要的资源，较高优先级的任务就会被阻塞，以便等待该资源有效。而这个过程，一个中优先级的任务可以抢占较低优先级的任务，结果导致了较高优先级的任务也不得不等待比它优先级低的中优先级任务，这种现象称为优先级翻转。

第 3 章

内核组件

3.1 介绍

RTOS 为实时嵌入式系统应用的开发提供多种不同的服务，这些服务允许应用开发者可以创建、操作和管理系统资源以及组件，以便完成各种应用的开发。本章的主要内容是介绍 RS-RTOS 提供的这些服务和组件。图 3.1 给出了 RS-RTOS 内核组件结构图。

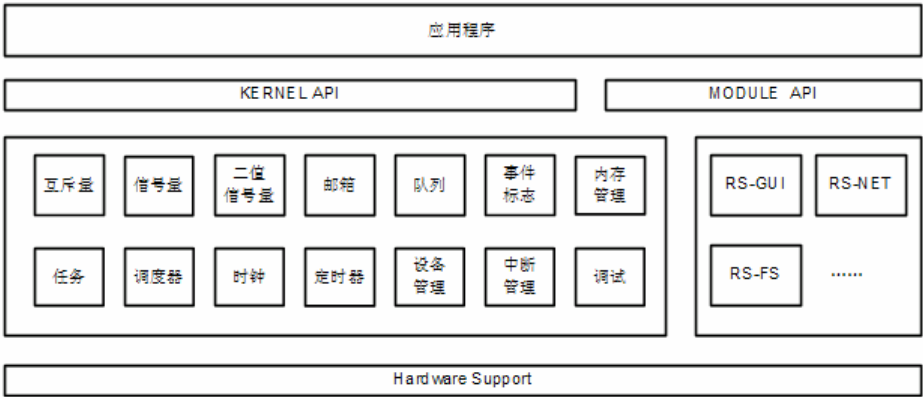


图3.1 RS-RTOS 内核组件

3.2 数据类型

RS-RTOS 使用一些的基本数据类型,而且可以直接映射为标准 c 编译器可识别的数据类型,这样做是为了保证程序在不同 c 编译器之间的可移植性。表 3.1 是 RS-RTOS 各种服务用到的数据类型介绍。

表 3.1 RS-RTOS 基本数据类型

数据类型	描 述
int8u	8 位无符号整型
int8s	8 位有符号整型
int16u	16 位无符号整型
int16s	16 位有符号整型
int32u	32 位无符号整型
int32s	32 位有符号整型
uint	寄存器无符合整型, 在 8 位处理器上是 int8u, 16 位是 int16u, 32 位上是 int32u
sint	寄存器有符合整型, 在 8 位处理器上是 int8s, 16 位是 int16s, 32 位上是 int32s
void	无类型
bool	布尔型 true 或者 false
char	字符类型
byte	字节, 8 位无符号
word	字, 16 位无符号

除了这些基本数据类型, RS-RTOS 使用系统数据类型来定义和声明系统资源, 如任务入口和内核名字。表 3.2 是这些数据类型的介绍。

表 3.2 RS-RTOS 系统数据类型

数据类型	描 述
------	-----

stack_t	任务栈类型，任务栈的类型
sp_t	任务栈指针类型，指向任务栈的指针
mail_t	消息邮箱类型，一般为32位，能容纳一个指针大小
mmsz_t	buffer长度类型，与c标准库中size_t等同
tick_t	系统时钟类型，一般为32位
count_t	计数器类型，一般为16位，用在任务的引用计数，如信号量的资源计数，互斥锁的引用计数
name_t	内核对象名字，32位大小，可保存四个字符
prio_t	任务优先级，8位大小
arg_t	接口参数，一个指针大小（一般为32位）
entry_t	任务入口函数
status_t	返回状态，有符号

3.3 任 务

任务是一段功能逻辑独立的程序集。一个处理器可以同时运行多个不同的任务，而在同一时刻，只能有一个任务拥有处理器资源。占有处理器的任务称为当前任务，等待处理器资源的任务称为就绪任务，还有一些暂时不能运行称为阻塞任务。

多个任务可以共享相同的内存空间，但每个任务必须有自己的堆栈。任务是嵌入式应用程序的基本组成部分，因为他们包括了应用程序中大部分的程序逻辑。**RS-RTOS** 可以创建的任务数量依赖于具体处理器和内存资源，而且每个任务拥有的堆栈可以有不同的大小。多个任务执行时，他们可以互相独立运行而互不干扰。

任务具有若干属性，表 3.3 列出了详细的属性项，在任务创建时，这些属性需要被定义。

表 3.3 任务属性
任务优先级
任务名字
任务入口函数
任务入口参数
任务栈起始地址
任务栈大小
任务选项

所有这些属性被定义在一个结构中，这个结构称为任务控制块（Task Control Block，TCB），其中包含了一些关键的系统信息，大多数应用程序没有必要访问 TCB 的内容。每个任务都被分配一个名字，主要是方便我们区分任务信息，但内核识别一个任务不是通过任务的名字，而是使用任务的优先级。RS-RTOS 中一个优先级只能有一个任务，任务的优先级可以在运行时动态修改，直接使用优先级作为任务的唯一标识，有利用提高内核效率。任务的入口函数是任务应用代码的实际入口，任务入口参数是任务开始时传入任务入口函数的参数，参数值的意义完全由应用开发人员来定义，任务入口参数可以作为区分具有相同任务入口的不同任务的标志。每个任务必须有一个任务栈，栈空间的起始地址可以由应用确定，或者由系统自动分配，但必须指出任务栈的所占用空间的大小。任务的选项用来确定一些扩展控制属性，如指示被创建的任务是否马上运行，任务被删除时是否主动释放栈空间等。

3.4 中断管理

对于实时嵌入式系统应用来说，最重要的功能就是对外部异步事件的快速响应。能够对外设中断请求信号（Interrupt Request line，IRQ）作出迅速的响应是现代处理器一项重要的特性。对 IRQ 响应的处理例程，称之为中断服务例程（Interrupt Service Routines，ISR）。ISR 例程和处理器密切相关，需要充分掌握处理器细节，编写与维护难度较大。RS-RTOS 的中断管理是对 ISR 的封装，屏蔽了对 CPU 依赖的部分处理代码，可以用普通 c 语言方便的编写 ISR 服务。表 3.4 给出了中断管理的属性列表。

表 3.4 中断管理属性
中断到期函数
到期函数参数

3.5 定时器

对于实时嵌入式系统应用来说，最重要的功能就是对外部异步事件的快速响应。很多应用还要求在指定的间隔进行周期性的计算处理。借助应用定时器，应用程序可以在指定的间隔时间执行定时器服务函数。它也可以用来为应用程序提供仅有一次的定时，这类定时器被称为单次定时器（One-shot Timer）；而周期性定时器被称为周期定时器（Periodic Timer）。

表 3.5 给出了应用定时器的属性列表。每个应用定时器有一个控制块来存放必要的系统信息。每个应用程序定时器都会分配一个名字，主要用于人机交互的标识，系统不将其作为

区分定时器标识。其他的属性包括定时器到期执行的服务函数，传递给到期服务函数的参数，该参数的意义是由开发者定义的。

应用定时器是由硬件驱动的（通常是使用一个周期性的硬件中断来实现），其精度是比较高的。应用定时器的属性与 **ISR**（中断服务例程）非常相似。应用程序通常使用定时器实现超时、周期计算、看门狗（**Warchdog**）服务。应用定时器也常用于中断任务的执行，实现如任务管理和统计任务运行信息等应用。此外，应用定时器之间不能互相中断执行，这是与中断服务例程的不同之处。

表 3.5 定时器属性
内核名字
定时器到期函数
到期函数参数
定时器周期
定时器选项

3.6 互斥量

互斥量（**Mutex**）是实现任务互斥访问的工具，用于互斥多个任务对临界区资源的访问。互斥量是一种可以被某个任务独自占有的资源。对于可以定义的互斥量的数量 **RS-RTOS** 没有任何限制，仅仅受限于硬件所能提供的 **RAM** 资源。

图 3.6 给出了互斥量的属性列表。互斥量有一个控制块用于保存该互斥量相关信息。每个互斥量会分配一个内核名字，主要用于人机交互的标识，系统不使用内核名字区分互斥量。优先级选项用于表明该互斥量是否支持优先级继承。当高优先级任务等待一个被低优先级任务占用的互斥量时，优先级继承允许低优先级任务暂时以高优先级任务的优先级别运行。这样，避免中优先级任务抢占拥有互斥量的任务，防止出现优先级翻转。互斥量是唯一支持优先级继承的 **RS-RTOS** 资源。

表 3.6 互斥量属性
内核名字
优先级继承项

3.7 信号量

信号量是用来解决任务同步（**Task Synchronization**）和互斥（**Mutual exclusion**）的通用的工具。信号量也可以代替互斥量用作资源互斥访问，但信号量没有所有权的概念。在应

用范围上比互斥量广，而互斥量更专注互斥功能。

信号量包含一个计数值，用来表示资源实例（Instance）的数目。RS-RTOS 系统提供 32 位的信号量，计数的范围是 0 到 $2^{32}-1$ （包括 0 和 $2^{32}-1$ ）。计数值位宽是可配置的，在资源受限的嵌入式系统，可配置为 8 位（0 到 255）或 16 位（0 到 65535）。这样可以节约一些内存资源，但会失去一定的兼容性。当一个信号量被创建后，计数值必须被初始化为以上范围的一个值。信号量的值是该信号量的实例数目。也就是说，如果信号量的计数值为 10，那么就有 10 个该信号量实例。

图 3.7 给出了信号量的属性。信号量有一个控制块用于保存该信号量的运行信息。每个信号量都会分配一个内核名字，这个名字用于人机交互的标识，系统不使用它作为区分信号量的标识。信号量计数值表示了目前有多少个资源实例。如上述说明，信号量的计数值必须符合范围。在使用信号量之前，首先需要创建它。RS-RTOS 系统没有限制可创建的信号量个数。

虽然信号量比互斥量功能简单，并且不支持优先级继承。但信号量是一种轻量级的内核对象，占用资源很少，使用非常灵活。因此，在很多场合下也使用信号量作资源互斥，只在 对互斥要求严格的应用中，才使用互斥量。

表 3.7 信号量属性
内核名字
信号量计数值

3.8 二值信号量

二值信号量是信号量的一种特殊形式，其特性与信号量一致，唯一的区别是二值信号量只能取值为 false 和 true（0 和 1）。当信号量值为 false（或 0）时，表示该二值信号量无效，当信号量值为 true（或 1）时，表示该二值信号量有效。二值信号量更多的被用于标识一个互斥的资源。

图 3.8 给出二值信号量的属性。和信号量一样，系统为每个二值信号量分配一个内核名字，用于人机交互的标识，系统不使用它区分二值信号量。二值信号量只能被初始化为 false 或 true（0 或 1）两种值。二值信号量属性存放在其控制块中。

表 3.8 二值信号量属性
内核名字
信号量二元值

二值信号量比信号量占用上更少的资源，速度也更快，是 RS-RTOS 系统中最轻巧的对

象。

3.9 事件标志

事件标志（Event Flags）为任务同步提供了强有力的武器。每个事件由一位数据表示，而每个事件标志由 32 位数据组成，可以表示互相独立的 32 个事件，事件意义完全由应用开发者定义。事件位宽是可配置的，在资源受限的系统中，可配置为 8 位（最大表示 8 个独立事件）或 16 位（最大表示 16 个独立事件）。这样可以节约内存资源，但会失去一定的兼容性。当一个事件标志被创建时，其中所包含的事件标志值都被初始化为 0。

图 3.9 给出事件标志的属性。事件标志包括一个控制块，用于保存该事件标志的状态信息。每个事件标志都会分配一个内核名字，主要用于人机交互的标识，系统不使用它来区分事件标志。事件标志还包含一个由 32 位数据构成的事件组结构。

表 3.9 事件标志属性
内核名字
事件组结构

任务可以同时 对 32 个独立的事件进行操作。事件标志组必须在使用前初始化。图 3.2 给出一个初始化过的事件组结构。RS-RTOS 系统没有限制创建事件标志的个数，这个能力仅受限于系统的硬件资源。

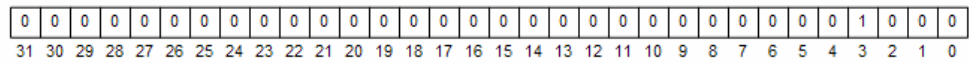


图3.2 事件标志位组

3.10 邮 箱

在多任务系统中，各任务之间免不了要进行的通讯。邮箱（Mail Box）是实现任务间的通信用工具。每个消息邮箱只存放一条消息，所以，使用邮箱一次只能传递一条信息。任务可以往一个空的邮箱发送消息，另一个任务从邮箱中取得该信息。当任务取走邮箱中的信

息后，邮箱被清空，其他任务就可以往该邮箱发送一条信息。邮箱接收到一条信息后，必须等到一个任务从该邮箱中取走该信息，方可继续接收一条信息。

图 3.10 给出邮箱的属性。邮箱包含一个控制块，用于保存该邮箱的状态信息，每个邮箱都会分配一个内核名字，这个名字用于方便人机交互信息，系统不使用名字来区别不同的邮箱。邮箱消息是一个宽 32 位的整型结构，用来存放消息内容。当需要存放的消息大于这个结构尺寸时，可以存放该消息的指针，间接实现消息的传递。邮箱状态指明该邮箱是否包含有效的消息。邮箱为空时，可以给邮箱发送信息；邮箱包含有效信息时，可以邮箱消息的读取。

表 3.10 邮箱属性
内核名字
邮箱消息
邮箱状态

邮箱为任务之间提供一种轻量级的通讯机制。**RS-RTOS** 系统没有限制能够创建的邮箱数目，唯一的限制是系统的硬件资源。任务可以给一个邮箱发送消息，同时也接收该邮箱的消息。邮箱必须在使用前初始化。

3.11 队 列

队列（**Queue**）是实现任务间的通讯主要的，也是最常用的方法。邮箱一次只能传递一条信息，而队列同时可以传递多个消息。消息在队列中按一定的顺序排列，后到的消息被置于队列的尾部，并从队列头部删除多余的消息。

图 3.11 给出队列的属性。队列包含一个控制块结构，用来保存队列的状态信息。每个队列都会分配一个内核名字，作为人机交互的标识，系统不使用它区分不同的队列。消息大小指定每条消息的长度。队列长度表示队列能容纳的消息数目。队列地址是指向容纳该队列消息缓冲区的起始地址，队列的消息缓冲区可以由系统自动分配，也可以由应用为其指定一个起始地址。

表 3.11 队列属性
内核名字
消息大小
队列长度
队列地址

图 3.3 给出一个队列的示例，队列中每个消息的长度是相等的，并按照固定的顺序存放。

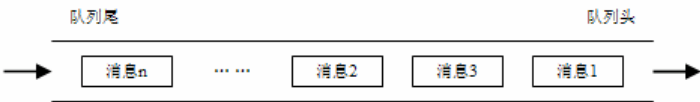


图3.3 RS-RTOS 队列结构

消息存放顺序有两种方式：先进先出（First In First Out, FIFO）和后进先出（Last In First Out, LIFO）。

3.12 内存池集

内存池集是一种动态内存分配策略。该模式的基本思想是将一块连续的内存划分为大小不同内存池，每个内存池包含一组单一尺寸的内存块，内存块是内存分配的单位。图 3.4 表示了固定尺寸内存池集的模式。

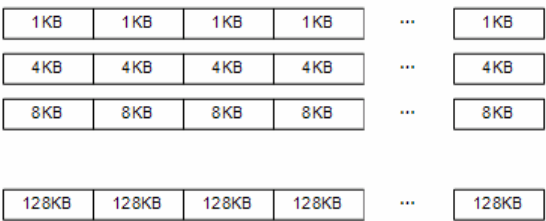


图3.4 内存池集

内存池集模式不会产生内存碎片化（Fragmentation），并且内存分配速度很快，非常适合对实时性能要求苛刻的系统。使用内存池集的缺点在于前期建模较为繁琐，要求手工配置池块的大小和数量，需要设计者具备较完整的系统知识，内存池尺寸划分不合理会导致内存利用率降低，甚至是内存分配失败。虽然内存池集配置较为麻烦，但瑕不掩瑜，内存池集作为一种高性能的内存管理策略，仍是实时嵌入式系统的最佳选择。

在使用上，内存池与 c 语言标准库中的堆（Heap）分配没有任何区别，同样需要在使用前申请内存，在使用完成后，需要释放该内存，否则会导致内存泄漏（Memory Leak）。

第 4 章

内核服务

4.1 介绍

内核是 RS-RTOS 系统最基本运行平台，它包含了一组可选的组件。在这个平台上可以构建以任务为核心的应用。内核提供一组服务接口供应用程序调用，这组接口称为应用编程接口（Application Programming Interface, API）。应用程序通过 API 接口使用内核功能，包括系统基本的运行控制，获取内核相关状态信息，创建并使用内核对象。RS-RTOS 所有内核服务接口均在 RS-RTOS 目录下的 inc/kapi.h 中声明。

本章首先给出内核服务接口概要，然后详细介绍内核控制相关的一组通用服务接口，还将给出一些例子和相关图示说明。

4.2 内核服务综述

附录 A~N 给出了 RS-KERNEL API 服务接口用户指南，包含了所有内核组件提供的接口，这些接口按照不同的组件分组编排，方便用户查阅。服务接口信息包括函数的原型、简要介绍、参数、返回值、调用者、阻塞状态和其他注意项。表 4.1 列出了 RS-RTOS 提供的所有内核服务的列表。在以后的章节，将逐个介绍这些服务，考察这些服务各方面的特性，还将给出一些例子和图示说明。

表 4.1 内核服务接口

	服 务	描 述
基 础 服 务	build_name	建立内核名字
	kernel_version	返回内核版本
	kernel_version_s	获得内核版本的字符串
	kernel_lock	锁定内核
	kernel_unlock	解除内核锁
	kernel_islock	返回内核锁状态
	kernel_info	获取内核信息
任 务	task_create	创建任务
	task_delete	删除任务
	task_change_prio	更改任务优先级
	task_change_runprio	更改任务运行优先级
	task_reset_runprio	重置任务运行优先级
	task_ident	获取任务标识
	task_self	获取任务伪标识
	task_current	获取当前任务运行优先级
	task_sleep	任务休眠指定时间
	task_wake	唤醒休眠中的任务
	task_suspend	挂起任务
	task_resume	恢复任务
	task_protect	保护任务
	task_unprotect	退出任务保护
	task_info	获取任务信息
时	tick_get	获取系统节拍

钟	tick_set	设置系统节拍
中断	interrupt_attach	注册中断服务
	interrupt_detach	注销中断服务
	interrupt_enable	允许中断
	interrupt_disable	禁止中断
	interrupt_catch	捕获中断
	intnake_attach	注册纯中断服务
	intnake_detach	注销纯中断服务
	intnake_enable	允许纯中断
	intnake_disable	禁止纯中断
	intnake_catch	捕获纯中断
定时器	timer_create	创建定时器
	timer_delete	删除定时器
	timer_enable	启动定时器
	timer_disable	停止定时器
	timer_control	定时器控制
互斥量	mutex_create	创建互斥量
	mutex_delete	删除互斥量
	mutex_wait	获取（等待）互斥量
	mutex_trywait	获取（无等待）互斥量
	mutex_release	释放互斥量
	mutex_info	获取互斥量信息
信号量	semaphore_create	创建信号量
	semaphore_delete	删除信号量
	semaphore_wait	获取（等待）信号量

	semaphore_trywait	获取（无等待）信号量
	semaphore_post	释放信号量
	semaphore_info	获取信号量信息
二 值 信 号 量	seminary_create	创建二值信号量
	seminary_delete	删除二值信号量
	seminary_wait	获取（等待）二值信号量
	seminary_trywait	获取（无等待）二值信号量
	seminary_post	释放二值信号量
	seminary_info	获取二值信号量信息
邮 箱	mailbox_create	创建邮箱
	mailbox_delete	删除邮箱
	mailbox_send	发送邮箱消息
	mailbox_receive	接收邮箱消息
	mailbox_flush	清空邮箱
队 列	queue_create	创建队列
	queue_delete	删除队列
	queue_send	发送队列消息
	queue_receive	接收队列消息
	queue_flush	清空队列
事 件	event_create	创建事件标志
	event_delete	删除事件标志
	event_wait	获取（等待）事件
	event_trywait	获取（无等待）事件
	event_post	设置事件
	event_info	获取事件标志信息

内存管理	mpool_alloc	申请内存池
	mpool_free	释放内存池
	mpool_info	获取内存池信息
设备管理	build_device	创建设备标识
	driver_install	安装设备驱动
	driver_uninstall	卸载设备驱动
	device_init	设备初始化
	device_open	打开设备
	device_close	关闭设备
	device_write	设备读
	device_read	设备写
	device_ioctl	设备控制

4.3 基础服务接口

基础服务接口是内核服务接口的一部分，它提供内核基础控制，以及内核运行信息。表 4.2 列出了基础服务接口，本章以下小节将逐个介绍这些接口。

表 4.2 基础服务接口

服 务	描 述
build_name	建立内核名字
kernel_version	返回内核版本
kernel_version_s	获得内核版本的字符串
kernel_lock	锁定内核
kernel_unlock	解除内核锁
kernel_islock	返回内核锁状态

kernel_info	获取内核信息
-------------	--------

4.4 建立对象名字

在 RS-RTOS 系统中，最基本的结构是内核对象。如一个任务或者一个信号量都是内核对象，每个内核对象都有一个名字，也可以不设置。名字只作为描述方便，不是内核区分对象的标识。因此，允许有两个名字相同的内核对象。

内核对象名字由英文字母(a~z, A~Z)、数字(0~9)以及下划线(_)组成，并且以字母或下划线开头，名字长度不能超过四个字符。通过接口：

```
name_t build_name(char c1, char c2, char c3, char c4);
```

创建一个内核对象名字。内核对象名字在 RS-RTOS 内部采用一个 32 位的整型保存，故有很高的效率。以下代码建立一个内核名字 TA1：

```
build_name('T', 'A', '1', '\0');
```

本书所提供的例子中的内核名字均以终结字符'\0'结尾，仅仅是为了在调试器中查看方便，不是强制性要求，完全可以不添加终结字符'\0'，如以下代码：

```
build_name('M', 'U', 'T', '1');
```

将创建一个内核对象名字 MUT1。如果在调试器中查看该对象名字，可能紧在接着 MUT1 后面显示乱码。这是调试器不能完全识别内核名字的原因，不会带来任何的问题。

4.5 返回内核版本

接口 kernel_version 返回 RS-RTOS 内核版本号（也就是 RS-KERNEL 版本号），接口原型如下：

```
int16u kernel_version(void);
```

接口返回值是一个两字节的 16 进制整数，高字节表示主版本号，低字节表示次版本号。

如 0x0122 表示的版本号为 V1.22。

4.6 返回内核版本字符串

接口 `kernel_version_s` 返回 RS-RTOS 内核版本的字符串形式，接口原型如下：

```
char __p_* kernel_version_s(void);
```

接口返回以终止符'\0'结尾的 c 语言字符串，如在内核版本 V1.22 中调用该接口将返回字符串"1.22"。

4.7 锁定内核

某些情况下，任务可能需要完成一项重要的工作，并且不希望被其他任务打断，例如系统在调电时存储重要的数据资料。这个时候，可以暂时性的禁止任务调度，待任务完成该工作后，再次允许任务调度。禁止任务调度通过锁定内核服务完成，其接口原型如下：

```
void kernel_lock(void);
```

该接口没有返回值，因而，锁定内核在任何情况下都是成功的。锁定内核将禁止调度器进行任务调度运算，禁止一切任务切换操作。在锁定内核后应该避免可能引起任务阻塞系统调用，如获取互斥量会使任务进入等待状态（产生任务切换），否则将返回错误信息。

锁定内核不影响内核对中断服务和定时器应用的响应。另外，锁定内核会降低系统的实时性，因为紧迫的任务不能得到及时的运行。所以，一般只在系统的调试阶段使用它，在设计中尽量避免使用内核锁定功能，并尽可能的缩短内核锁定的时间，通过接口 `kernel_unlock` 解除内核的锁定状态。

注意：不要在 RS-RTOS 内核初始化阶段使用内核锁。

4.8 解除内核锁

内核在锁定状态下，必须通过接口 `kernel_unlock` 解除内核的锁定状态，其接口原型如下：

```
void kernel_unlock(void);
```

内核使用一个引用计数器计算内核被锁定的次数，该计数器的宽度与处理器的位宽一致，在 8 位处理器上最大值为 255，16 位处理器上最大值是 65535，32 位处理器上最大值是 $2^{32}-1$ 。这允许内核锁被嵌套使用，嵌套的最大深度不能超过以上值。注意，在某些情况下，内核也会进行自动锁定（例如发生中断时）。所以，实际应用中，内核锁的嵌套层数应该预留一定的空间。

`kernel_lock` 与 `kernel_unlock` 要严格成对使用。在嵌套使用内核锁时，只有在最外层的 `kernel_unlock` 解除内核的锁定状态。

4.9 返回内核锁状态

接口 `kernel_islock` 返回内核锁的状态，接口原型如下：

```
uint kernel_islock(void);
```

返回值类型 `uint` 是与处理器位宽一致的无符号整型。返回值为 0 时，表示内核处于未锁定状态，如果返回值大于 0，表示内核在锁定状态，其值是内核被锁定的引用计数。

4.10 获取内核信息

通过调用 `kernel_info` 可以获取内核运行时（Run-time）的状态信息。获取的信息包括系统处理器的利用率、任务使用情况。获取内核信息的接口原型如下：

```
status_t kernel_info(kinfo_t __p_* info);
```

该接口传入一个 `kinfo_t` 结构（`kinfo_t` 是 `kernel information` 的缩写）指针参数。`kinfo_t` 结构使用的内存由调用者负责分配，用来存放服务返回的信息。表 4.3 给出了 `kinfo_t` 结构详细说明：

表 4.3 接口 `kernel_info` 参数 `kinfo_t` 结构说明

参 数	说 明
-----	-----

cpu_usage	返回处理器利用率，返回值为 8 位宽的整数，单位为百分之一
task_no	返回系统的任务数

下面给出一个示例 APP_SAMPLE_04A，该例子演示了利用接口 `kernel_info` 获取处理器的利用率信息，并在控制台打印该信息。

【例 APP_SAMPLE_04A】

file: example\WIN32\app_sample_04a.cpp

```
.....
00090     status_t status;
00091     kinfo_t  info;
.....
00096     for (;;)
00097     {
00098         /* 获取内核信息 */
00099         status = kernel_info(&info);
00100
00101         /* 如果返回信息成功,打印内核信息 */
00102         if (status == RS_EOK) {
00103
00104             /* 显示 CPU 利用率信息 */
00105             printk("\rCPU Usage: %3d%%", info.cpu_usage);
00106         }
00107
00108         task_sleep(RS_TICK_FREQ);
00109     }
.....
```

4.11 总 结

本章首先介绍了 RS-RTOS 应用编程接口 (Application Programming Interface, API), 并给出了完整的内核服务接口列表。应用程序通过内核 API 使用 RS-RTOS 的内核功能, 进行基本的运行控制, 获取内核相关信息, 创建和使用内核对象等。通过应用编程接口, 可以轻易的构建用户所需的应用程序。本章的后半部分介绍了基础服务部分的 API, 并给出了相应例子。在后面的章节, 将陆续介绍其余部分 API 接口。

第 5 章

任 务

5.1 介 绍

在前面的章节，已经介绍了任务各个方面的性质，包括任务的应用、创建、组成和用途，本章将详细介绍与任务相关系统服务，这些服务涉及了任务的创建、删除、休眠、唤醒、更改优先级等各方面。在介绍这些服务之前，首先回顾一下任务控制块，然后逐个介绍任务的系统服务，并给出示例说明这些服务的特点和功能。

5.2 任务控制块

任务控制块（Task Control Block，TCB）是用来保存运行时（Run-time）任务状态的数据结构，它也用来在任务切换时保存任务信息。表 5.1 给出了构成 TCB 的主要属性项。

TCB 可以存在于内存的任意位置，通常把 TCB 定义成全局变量，以便它能够被所有相关函数访问。需要注意的是，如果控制块在一个 c 函数内部被声明，存储该 TCB 的内存将从该函数的栈中分配。通常应该避免这种情况，因为一旦该函数返回，它使用过的栈将被释放掉，任务不能继续使用该 TCB。

大多数情况下，开发者不需要了解 TCB 的内容。但某些时候，特别是在调试时，观察某些属性项的值非常必要。表 5.1 解析了这些关键属性项的意义。

TCB 中还有很多有用的项，包括任务栈指针、任务状态、优先级别等。开发者可以观察 TCB 成员变量的值，但是绝对不可以人为的修改它。TCB 中没有成员项可以表明该任务当前是否正在被执行。在某一给定时刻，只有一个任务被执行，RS-RTOS 用其他的变量记录当前正在被执行的任务。注意，正在被执行的任务是任务就绪队列中之一。

表 5.1 任务主要属性

项	描 述
prio	任务优先级
name	任务名字
entry	任务入口函数
arg	任务入口参数
stack_base	任务栈起始地址
stack_size	任务栈大小
options	任务选项

5.3 重入函数

多任务编程的一个特点是，同一个函数可以被多个任务执行。这个特征给应用程序提供了很大的灵活性，并且有助于节省代码空间。但是，凡事有利必有弊，要写好这样一个的函

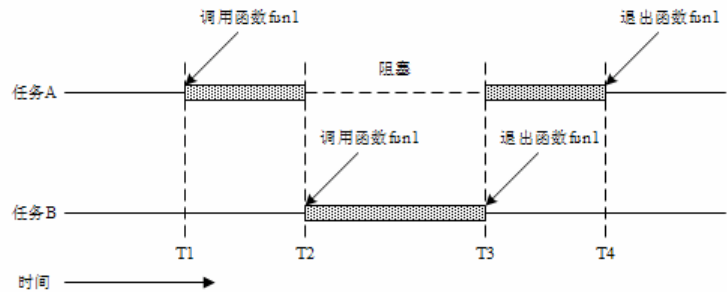


图5.1 函数重入

数并不十分容易，一不留神就会掉进暗藏的陷阱，这个陷阱就是函数的重入（**Reentrant**）。重入是一种什么样的状态呢？当一个任务调用一个函数时，发生了任务切换，新执行任务也调用了该函数。也就是说，在一个函数的调用还没退出时，发生了再次调用，这种情况叫做重入。其实，不仅是多任务环境，在函数递归时也会发生函数重入。图 5.1 演示了多任务环境下出现的函数重入的情况。

无论函数在那个时刻发生了重入，都能得到正确的运行结果，这种函数称为可重入函数。可重入的函数在执行时仍可以被其他任务安全地调用。根据重入函数的定义，可重入函数需要满足以下条件：

- 1) 函数内部不能使用全局或者静态对象；
- 2) 不返回指向静态或者全局的对象的指针；
- 3) 不调用其他的不可重入的函数。

满足这些条件的函数使用的数据都由调用它的任务提供，所有在函数内使用的变量在任务栈中分配。

一个不可重入函数的例子是，标志 **C** 函数库中的字符处理函数 **strtok**。该函数利用静态变量保存上一次调用是的指向字符的指针。如果这个函数被多个任务调用，它很可能会返回无效指针。

5.4 任务调用综述

附录 C 包含与任务相关的服务接口的详细信息，包括服务的接口原型、简要介绍、参数、返回值、调用者、阻塞状态、其他注意项。表 5.2 包括系统提供的任务服务的列表。在本章后续部分，将对它们逐个的介绍。在考察每个服务的同时，会给出一些例子和图示，帮助开发者理解该服务各个方面的性质，以及使用时要注意的一些事项。

表 5.2 任务服务接口

服 务	描 述
task_create	创建一个任务
task_delete	删除一个任务
task_change_prio	更改任务优先级
task_change_runprio	更改任务运行优先级

task_reset_runprio	重置任务运行优先级
task_ident	获取任务标识
task_self	获取任务伪标识
task_current	获取当前任务运行优先级
task_sleep	任务休眠指定时间
task_wake	唤醒休眠中的任务
task_suspend	挂起任务
task_resume	恢复任务
task_protect	保护任务
task_unprotect	退出任务保护
task_info	获取任务状态信息

5.5 创建任务

通过接口 `task_create` 创建一个任务。每个任务都必须有一个优先级，优先级是一个 8 位大小的数字，范围是从 0 到 254。值越小代表的优先级越高，每个优先级只能创建一个任务，不允许运行两个相同优先级的任务。此外，每个任务必须有自己的任务栈，开发者决定任务栈的大小和栈内存的分配方式。图 5.2 是一个任务栈的结构实例。

有很多为任务栈分配内存的方法，包括使用内存池、建立全局的数组、或者由系统自动分配、也可以直接指定任务栈的起始物理地址。无论使用那一种分配方法，任务栈的大小必须足够大，适应最坏情况下的函数嵌套、分配局部变量、以及任务上下文切换所需要的空间。如果任务栈分配过小，将直接影响任务的正常运行。因为深层次的函数调用、或者函数内过大的局部变量，会导致任务栈越界使用。任务栈越界使用是非常危险的，不仅涉及到自身任务的安全运行，很可能危及到别的任务、或者其他的资源。在系统资源充足时，宁可浪费一些内存，使用大的任务栈。根据实际的经验，任务栈预留 30%~50% 空间比较合适。

在系统设计初期，可以把任务栈的大小设计的大一些。在程序调试完毕后，再把任务栈调整到合适的大小。有一种非常简单又实用的方法，可以确定任务栈的实际大小。首先声明足够大的任务栈空间，用一个特征数据（如 0xCA），将整个任务栈填满，创建任务并运行测试，待完成测试后，从栈末端扫描特征数据的长度（没有被使用过的空间），就能确定实际需

要的栈大小了。这个方法的关键是要保证测试的覆盖比较充分，尽可能把任务栈使用的最大深度覆盖到。RS-RTOS 内核已经提供了测试任务栈使用量的方法，应用程序只需要的调用服务接口 `task_info` 就可以获得任务栈的使用信息，这个接口将在后面的章节介绍。

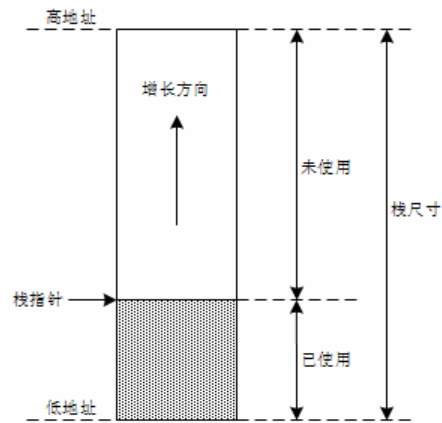


图5.2 任务栈

任务可能需要很大的栈空间，所以在设计程序是，应该创建合理数目的任务，同时避免对栈的过分使用。开发者应该尽量避免使用递归算法以及在函数内使用大的局部变量。

如果任务栈空间不足，会发生什么情况呢？默认情况下，运行环境假设任务栈是足够大的。也就是说，运行环境不会主动检查任务栈是否已经到达边界。因此，如果当前的任务栈已经到达临界状态，任务就会继续使用栈后续的内存空间。最终的结果是，任务栈临近内存中的内容被破坏了。被更改的内存可能是另一个任务栈的空间，或者是系统数据区。总之，不属于任务自身的栈空间被非法改动了，后果是难以预料的，通常会发生系统崩溃，表现为系统莫名其妙的当掉！避免出现危险的唯一办法就是使用一个足够大的任务栈。

前面的章节展示了任务控制块的属性项，这些属性是创建一个任务所需要的参数项。以下例子演示了使用接口 `task_create` 创建多个任务。通过这些实例，可以进一步的了解这些属性项的意义。

作为第一个例子，我们创建一个优先级为 10，任务入口 `app_task_1` 的任务。任务栈的尺寸为 $1024 \times 32 = 32768$ 字节，使用全局数组的方式分配栈空间。在创建任务的同时把它设置为就绪状态（Ready）。为了运行这个例子，还要实现任务的入口函数。下面列出了创建任务所需要的代码以及对应的任务入口函数。

【例 APP_SAMPLE_05A】

file: example\WIN32\app_sample_05a.cpp

```
.....
00048 /* 任务栈大小 */
00049 #define APP_STACK_SIZE      1024 * 32
00050
00051 /* 任务优先级 */
00052 #define APP_PRIO_TA1         10
00053
00054 /* 任务栈 */
00055 stack_t app_stack_ta1[APP_STACK_SIZE];
00056
00057 /* 任务入口函数 */
00058 void app_task_1(arg_t arg);
.....
00068 /* 应用初始化 */
00069 void application_initialize(void)
00070 {
00071     status_t status;
.....
00076     /* 创建任务 TA1 */
00077     status = task_create(
00078         APP_PRIO_TA1,
00079         build_name('T', 'A', '1', '\0'),
00080         app_task_1, (arg_t)0x1234abcd,
00081         app_stack_ta1, APP_STACK_SIZE,
00082         0);
.....
00085 }
00086
00087 /* 任务 TA1 入口 */
```



```
00088 void app_task_1(arg_t arg)
00089 {
00090     /**
00091      * 这里是任务的开始，
00092      * 通常在这里进行一些相关的模块初始化工作。
00093      */
00094
00095     for (;;)
00096     {
00097         /**
00098          * 这里是任务的主循环，用以实现任务的主体功能，
00099          * 在任务的主循环中，通常要包括能够引起任务阻塞的接口，
00100          * 如 task_sleep, semaphore_wait, queue_receive 等，
00101          * 否则比该任务优先级低的任务将无法得到运行机会。
00102          */
00103
00104         task_sleep(RS_TICK_FREQ);
00105     }
00106 }
.....
```

在这个例子中，代码：

```
00077     status = task_create(...);
```

创建了一个名为 TA1 的任务，在 RS-RTOS 系统中，所有服务接口都是以小写字母开头，并使用下划线“_”连接成的。声明：

```
00071     status_t status;
```

用来存放内核服务调用的返回值，大部分 RS-RTOS 服务接口都使用 `status_t` 类型返回接口执行结果的状态信息。返回值用来指示服务执行的结果，如果服务执行失败，返回值小于 0，通过附录的接口手册可以获得失败代码的详细信息。根据应用的需要，也可以不理睬

服务接口的返回值，比如，可以使用以下调用方式：

```
00077 task_create(...);
```

但作为一个良好的编程习惯，应尽可能的检查服务接口的返回值，确定调用是否成功。如本示例中使用以下语句来确定是否成功创建了任务 TA1。

```
00084 ASSERT(status == RS_EOK);
```

ASSERT 是一个预定义宏，在文件 app_sample.h 中被定义为 assert，assert 是标准 c 语言库的错误判定宏。如果任务 TA1 创建失败，接口返回值 status 不等于 RS_EOK，assert 将中断程序运行，并打印出错代码所在的文件和出错的行号。这样，开发者可以快速定位出错位置。关于 assert 的使用已经超出了本书讨论范围，这里不作深入探讨，请读者朋友参考资料。回到创建任务的讨论上，通过 task_create 创建任务，一共使用 7 个参数，这些参数指定了任务各方面的特性。表 5.5 给出了这些参数的说明：

表 5.3 例子 APP_SAMPLE_05A 中创建任务的参数说明

参 数	说 明
APP_PRIO_TA1	任务的优先级，范围是 0~254
build_name('T', 'A', '1', '\0')	任务名字，由四个字符组成，build_name 是创建内核对象名字的接口，所有的内核对象名字必须由 build_name 建立。
app_task_1	任务入口函数
(arg_t)0x1234abcd	任务入口函数的参数，是一个 32 位宽的整数。
app_stack_ta1	任务栈的起始地址。
APP_STACK_SIZE	任务栈大小。
0	任务选项。

任务入口函数必须声明为以下形式：

```
00058 void app_task_1(arg_t arg);
```

任务函数没有返回值，在前面的章节中提到，典型的任务是一个无限的循环，永远不会返回。但是，这不是强制的规则，在 RS-RTOS 系统中，是允许任务返回的。如果任务返回，

任务将调用删除任务服务（接口 `task_delete`）删除自己，这个调用是隐含的，也就是说，不需要在任务返回时编写删除任务的调用代码。删除任务接口的使用将在稍后的介绍中继续研究。采用无限循环还是带返回的形式，完全是取决于任务本身的应用设计。

附录 C 包含了创建任务接口的使用参数和返回值的详细说明。返回值说明接口调用是否成功，如果不成功，返回值说明了失败原因，返回值的意义在附录中均有详细的描述。

任务入口参数的具体意义完全是由应用决定的，如本例的入口参数为 `0x1234abcd`。例如，在工程中常用任务入口参数区分不同的任务。

例子 `APP_SAMPLE_05B` 演示了这种用法，这里创建了两个任务 `TA1` 和任务 `TA2`，这个例子的特别之处在于，两个任务都使用同一个入口函数。这样做理由是，举个例子，为蛋糕店编写一个系统，用来接待客户。有两个客户，客户 A 需要全麦面包，客户 B 需要奶油蛋糕。因为两个客户采用一套相同下单、确认和付费流程。这样可以用相同的代码完成这个过程，并使用一个任务完成这个过程。因此，两个客户只需要执行相同的任务代码，但是，还得区分不同的客户，利用任务入口参数，就可以解决这个问题。

任务的入口参数是一个 32 位的整型数据，它的大小恰好能容纳一个指针（`sizeof(void *)`），应用程序通过地址的方式传递任意大小的数据。当传递信息量比较少，用一个 32 位的整型就能容纳时，直接的值传递是最方便的，如例子 `APP_SAMPLE_05B`，参数值 1 表示任务 `TA1`，参数值 2 表示是任务 `TA2`。以下给出任务创建部分代码：

【例 APP_SAMPLE_05B】

file: example\WIN32\app_sample_05b.cpp

```
.....
00075     status_t status;
00076
00077     /* 打印关于示例一些信息 */
00078     APP_SAMPLE_INFO;
00079
00080     /* 创建任务 TA1 */
00081     status = task_create(
00082         APP_PRIO_TA1,
00083         build_name('T', 'A', '1', '\0'),
00084         app_task_1n2, (arg_t)0x00000001,
00085         app_stack_ta1, APP_STACK_SIZE,
```

```

00086         0);
00087
00088     ASSERT(status == RS_EOK);
00089
00090     /* 创建任务 TA2 */
00091     status = task_create(
00092         APP_PRIO_TA2,
00093         build_name('T', 'A', '2', '\0'),
00094         app_task_1n2, (arg_t)0x00000002,
00095         app_stack_ta2, APP_STACK_SIZE,
00096         0);
00097
00098     ASSERT(status == RS_EOK);
00099     .....

```

这个示例演示创建两个任务，任务 TA1 与任务 TA2，并且，两个任务使用同一个入口函数：app_task_1n2。任务 TA1 的入口参数是 0x00000001，任务 TA2 的入口参数是 0x00000002。以下是任务 TA1 和任务 TA2 的入口函数定义：

【例 APP_SAMPLE_05B】

file: example\WIN32\app_sample_05b.cpp

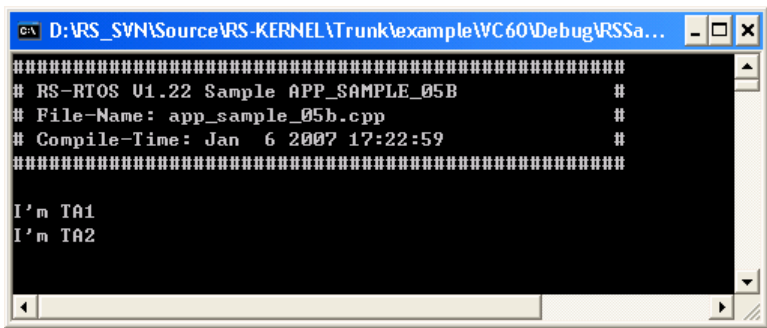
```

.....
00101 /* 任务 TA1,任务 TA2 的入口 */
00102 void app_task_1n2(arg_t arg)
00103 {
00104     /* 打印出任务名字. */
00105     printf("I'm TA%d\n", arg);
00106     .....
00113 }
.....

```

通过入口函数参数 arg 取得任务创建时的入口参数值。在上面给出的代码中，任务 TA1

的 arg 值是 0x00000001，任务 TA2 的 arg 值是 0x00000002。printk 是控制台格式字符输出命令，它与 printf 的用法基本是一样的，但功能上弱一些，不支持浮点数据的格式输出。以下为例 APP_SAMPLE_05B 的输出结果：



任务 TA1 的优先级比任务 TA2 高，因而任务 TA1 首先打印出结果，然后任务 TA2 输出打印结果。有兴趣的读者不妨将它们的优先级别颠倒一下，再运行更改后的示例，对比更改前后的输出结果。

这个例子也展示了任务代码与任务之间的关系，请读者体会两者的联系与区别。

5.6 删除任务

当任务已经完成它的工作，或者任务处于一种错误状态，并且不能通过其他方法恢复正常的运行时。需要将任务删除，释放任务占用的系统资源。删除一个任务通过调用接口：

```
status_t task_delete(prio_t prio);
```

从删除任务服务的接口原型可以看出，删除一个任务比创建它要简单的多，事实上确实如此。下面的示例 APP_SAMPLE_05C 将演示删除一个任务的操作过程。该示例在初始化时创建了两个任务，任务 TA1 与 TA2。在任务 TA2 中完成对任务 TA1 的删除操作，以下是删除任务 TA1 的操作代码：

【例 APP_SAMPLE_05C】

file: example\WIN32\app_sample_05c.cpp

```
.....
```

```
00134      status_t status;
.....
00143      /* 删除任务 TA1. */
00144      status = task_delete(APP_PRIO_TA1);
.....
```

该例子完整的代码请看本书附带的光盘。删除任务服务不仅可以删除其他的任务，也可以调用 `task_delete` 删除任务自身。通过 `task_delete` 删除自身时（其他对当前任务的操作也同样适用），为了安全，通常使用任务的伪优先级。使用接口 `task_self` 获得任务的伪优先级，注意，`task_self` 返回的优先级与任务的真实优先级的值是不同的，`task_self` 返回值只能用于该当前任务。关于 `task_self` 详细介绍请参考本章中任务伪优先级一节。这里仅仅给出它的使用方法，以下示例给出了删除任务本身的代码：

【例 APP_SAMPLE_05D】

file: example\WIN32\app_sample_05d.cpp

```
.....
00091      status_t status;
.....
00101      /* 删除自身 */
00102      status = task_delete(task_self());
.....
```

任务调用 `task_delete` 成功删除自己后，紧接的代码将不能继续运行，因为任务被系统删除后，该任务对应的资源（包括处理器资源）已经被系统回收。因此，上面例子中，紧接着 L00102 后的代码是不会继续执行的，这个语句也不能输出到控制台上。

【例 APP_SAMPLE_05D】

file: example\WIN32\app_sample_05d.cpp

```
.....
00104      printk("After deleted.\n");
.....
```

需要强调的是，删除任务自身的服务调用，如果成功，接口将不会返回；相反，如果接

口调用返回了，则代表这个服务没有成功执行。

另外，使用删除任务接口还需要提防出现任务拥有的资源泄漏。假设任务申请了一段内存，并且在还没有释放它之前，删除该任务，任务申请的内存将无法回收。在任务还拥有待释放的资源时，删除该任务是非常危险的。在删除一个任务之前，确保该任务申请的资源可以安全被释放，这非常重要。在稍后的章节，还将介绍一对接口，用来确保删除任务的安全，再回过头来讨论这个问题。

5.7 更改任务优先级

任务被创建时必须指定一个唯一的优先级，在 RS-RTOS 系统中，优先级是一个宽 8 位的整数，取值范围是 0~244，优先级的值越大，代表的优先级越低。任务的优先级可以在运行中随时更改。通过接口：

```
status_t task_change_prio(prio_t prio, prio_t new_prio);
```

更改任务的优先级，表 5.4 给出该接口用到的参数说明：

表 5.4 接口 task_change_prio 参数说明

参 数	说 明
prio	任务的优先级
new_prio	更改后的任务优先级

例子 APP_SAMPLE_05E 演示修改任务优先级接口的使用，应用初始化创建三个任务：任务 TA1，任务 TA2，任务 TA3，对应优先级分别 TA1(10)，TA2(11)，TA3(12)；任务 TA1 启动后，将任务 TA2 的优先级修改为 15，如下所示：

TA1(10), TA2(11), TA3(12) --> TA1(10), TA2(15), TA3(12)

修改后任务 TA3 优先级比 TA2 高，TA3 比 TA2 先输出打印信息，下面是本例子修改任务 TA2 优先级的代码片段：

【例 APP_SAMPLE_05E】

file: example\WIN32\app_sample_05e.cpp

```
.....
00123     status_t status;
        .....
00128     /**
00129         * 更改任务 TA2 的优先级:
00130         * APP_PRIO_TA2A --> APP_PRIO_TA2B.
00131         * 更改成功后,任务 TA2 优先级比任务 TA3 优先级低
00132         */
00133     status = task_change_prio(APP_PRIO_TA2A, APP_PRIO_TA2B);
.....
```

除了可以修改其他任务的优先级，也可以修改任务自己的优先级。通常使用任务的伪优先级表示当前任务，如，使用以下代码修改任务自己的优先级：

```
task_change_prio(task_self(), 10);
```

注意，`task_self` 返回值与任务的真实优先级是不同的，`task_self` 返回值只能用于表示当前任务。

更改任务的优先级是永久性的，当一个任务的优先级被修改后，对该任务进行的操作需要使用修改后的优先级。如上面例子中，在任务 TA2 优先级被修改后，对任务 TA2 的操作均以 APP_PRIO_TA2B 作为标识。例如，使用以下语句删除任务 TA2：

```
task_delete(APP_PRIO_TA2B);
```

如果需要临时性的更改任务的优先级，更好的方法是更改任务的运行优先级，下一节将介绍更改任务运行优先级的方法。

5.8 更改任务运行优先级

任务的优先级有时会被内核调度做临时性的修改。比如发生优先级继承时，任务实际运行的优先级会被提升，这就会出现任务的优先级与实际运行的优先级不一致的情况。任务实际运行的优先级称为任务的运行优先级，它永远是表示任务真实的、确切的在运行中处在优

先级别。而任务的优先级是对任务进行操作的标识，是相对稳定的，它只能通过更改任务优先级接口修改。任务运行优先级可以被应用程序修改，通过使用接口：

```
status_t task_change_runprio(prio_t prio, prio_t runprio);
```

更改任务的运行优先级，该接口用到的参数说明在表 5.5 给出：

表 5.5 接口 task_change_runprio 参数说明

参 数	说 明
prio	任务的优先级
runprio	任务的目标运行优先级

例子 APP_SAMPLE_05F 演示了更改任务运行优先级服务的使用方法。该例子在初始化时创建两个任务，任务 TA1 和 TA2，优先级分别是 10、12。在任务 TA1 中，将自身运行优先级更改为 15，更改任务运行优先级的代码如下：

【例 APP_SAMPLE_05F】

file: example\WIN32\app_sample_05f.cpp

```
.....
00109      status_t status;
.....
00114      /**
00115       * 更改任务 TA1 的运行优先级:
00116       * TA1(10) -->> TA1(15).
00117       * 更改成功后,任务 TA1 运行优先级比任务 TA2 运行优先级低,
00118       * 导致了任务 TA2 马上抢占任务 TA1 而获得的运行.
00119       */
00120      status = task_change_runprio(task_self(), APP_PRIO_TA1R1);
.....
```

更改任务的运行优先级可能引起任务切换。在上面的演示中，任务 TA1 的运行优先级成功更改为 APP_PRIO_TA1RUN（该值定义为 15）后，由于该优先级比任务 TA2 的优先级 12 低（优先级的值越大，优先级别越低）。结果，已经处在就绪状态的任务 TA2 马上抢占任

务 TA1，获得处理器资源而运行。图 5.3 演示了任务 TA2 对任务 TA1 的抢占过程。

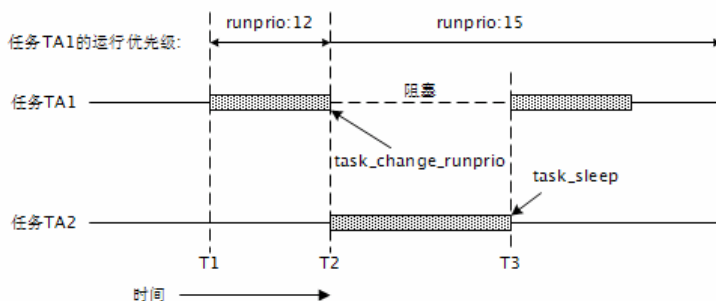


图 5.3 更改运行优先级

为了验证上面分析的是否正确，在调用修改任务运行优先级接口前后分别安排了一行输出信息。代码如下：

【例 APP_SAMPLE_05F】

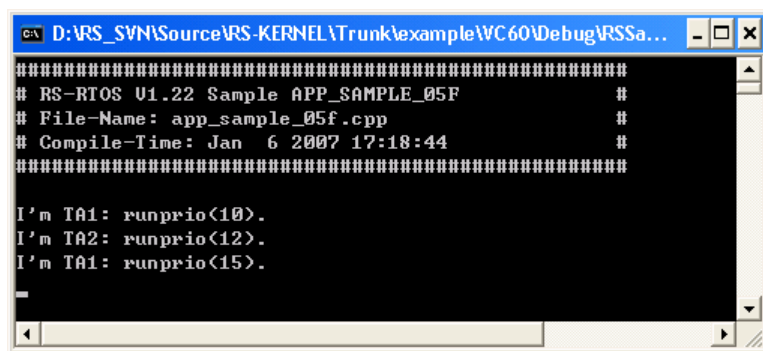
file: example\WIN32\app_sample_05f.cpp

```
.....
00111      /* 更改前的任务运行优先级信息 */
00112      printf("I'm TA1: runprio(%d).\n", task_current());
.....
00120      status = task_change_runprio(task_self(), APP_PRIO_TA1R1);
00121
00122      /* 更改后的任务运行优先级信息 */
00123      printf("I'm TA1: runprio(%d).\n", task_current());
.....
```

请注意行 L00112、L00122，这里使用另一个服务接口：

```
prio_t task_current(void);
```

该接口返回当前任务的运行优先级。这个例子的输出结果如下图。通过例子的输出结果可以看到，第一和第三行分别对应任务 TA1 行 L00112、L00122 的输出信息，而第二行信息是任务 TA2 的输出信息。



```
D:\RS_SVN\Source\RS-KERNEL\Trunk\example\WC60\Debug\RSSa...
#####
# RS-RTOS V1.22 Sample APP_SAMPLE_05F #
# File-Name: app_sample_05f.cpp #
# Compile-Time: Jan 6 2007 17:18:44 #
#####

I'm TA1: runprio(10).
I'm TA2: runprio(12).
I'm TA1: runprio(15).
```

任务的运行优先级被更改后，如果更改任务的运行优先级为任务优先级相同的值，将恢复任务在常态优先级下运行，如以下代码：

```
task_change_runprio(10, 10);
```

将恢复优先级为 10 的任务的运行优先级。虽然使用 `task_change_runprio` 可以恢复任务的运行优先级为常态，但更好的方法是，使用重置任务运行优先级（`task_reset_runprio`）服务实现这个目的，`task_reset_runprio` 服务重置任务的运行优先级为常态优先级，请参考下一节重置任务运行优先级的介绍。

一般情况下，通过更改任务的运行优先级（`task_change_runprio`），而不是通过更改任务的优先级（`task_change_prio`），来实现任务实际优先级的动态调整。理由是，如果修改一个任务的优先级（`task_change_prio`），并且不再恢复到原先的优先级，那么在一开始就应该将任务的优先级设定为最终的目标。此外，更改任务的优先级比更改任务的运行优先级消耗更多的系统资源。

5.9 重置运行优先级

接口 `task_reset_runprio` 提供了重置任务的运行优先级的能力。任务运行优先级被修改后（通过 `task_change_runprio`），通过接口 `task_reset_runprio` 重置任务的运行优先级，该服务将任务的运行优先级恢复到最初的状态。接口原型如下：

```
status_t task_reset_runprio(prio_t prio);
```

接口仅有一个传入参数，表 5.6 给出了参数说明：

表 5.6 接口 `task_reset_runprio` 参数说明

参 数	说 明
prio	需要重置运行优先级的任务的优先级

该接口仅影响任务的运行优先级。重置成功后，任务的运行优先级恢复到与任务优先级相同的值。重置任务的运行优先级可能引起任务切换，如果任务的运行优先级没有进行过修改，调用重置运行优先级服务不进行任何操作，不会对任务运行状态产生影响，并得到一个返回值指示操作失败。此外，只能对被应用修改的运行优先级（通过 `task_change_runprio`）进行重置，由内核更改的运行优先级是不能重置的。

5.10 识别任务

`task_ident` 返回当前正在执行的任务的优先级，作为任务的标识。在 RS-RTOS 系统中，同一个优先级只能创建一个任务，因此，这个值在系统中是唯一的。识别任务服务的接口原型：

```
prio_t task_ident(void);
```

如果在中断服务程序调用该服务，则返回值是被中断服务打断的那个任务的优先级。例子 APP_SAMPLE_05G 演示该接口的使用：

【例 APP_SAMPLE_05G】

file: example\WIN32\app_sample_05g.cpp

```
.....
00106      /* 输出任务优先级信息 */
00107      printf("I'm TA1: prio(%d).\n", task_ident());
.....
```

5.11 任务伪优先级

接口 `task_self` 返回一个指向当前任务的伪优先级。伪优先级并不是任务的真实优先级，伪优先级只对当前任务有效，用来识别当前任务。其接口原型如下：

```
prio_t task_self(void);
```

在前面多个例子中曾演示了伪优先级的使用方法，如例子 APP_SAMPLE_05F：

【例 APP_SAMPLE_05F】

file: example\WIN32\app_sample_05f.cpp

```
.....
00109     status_t status;
.....
00114     /**
00115      * 更改任务 TA1 的运行优先级:
00116      * TA1(10) -->> TA1(15).
00117      * 更改成功后,任务 TA1 运行优先级比任务 TA2 运行优先级低,
00118      * 导致了任务 TA2 马上抢占任务 TA1 而获得的运行.
00119      */
00120     status = task_change_runprio(task_self(), APP_PRIO_TA1R1);
.....
```

为了安全识别当前任务，通常使用伪优先级接口 `task_self`，而不要使用 `task_ident`。因为 `task_ident` 返回当前任务的优先级，而当前任务的优先级可能会被其他的任务所修改（通过接口 `task_change_prio`）。假设使用 `task_ident` 获得任务的优先级之后，当前任务优先级被其他任务所修改——这种情况很可能发生在被更高优先级的任务抢占之后。因此，使用 `task_ident` 返回值标识当前任务将不可靠，除非可以确定系统中没有（以后也不会有）更改该任务优先级的调用。既然有更好的选择，为什么还要种下一颗可能长成恶果的种子呢？

所以，接口 `task_ident` 只能反映服务执行时的任务优先级。然而，在任何情况下，使用 `task_self` 都是安全的，因为它的返回值不会随任务优先级的更改发生改变。

另一个示例 APP_SAMPLE_05H 显示了 `task_self` 的返回值，有兴趣的读者可以运行该

示例查看任务的伪优先级值。

5.12 当前运行优先级

当前任务运行优先级是指当前任务正在运行的优先级，使用接口 `task_current` 获得当前任务运行优先级，接口原型如下：

```
prio_t task_current(void);
```

示例 APP_SAMPLE_05I 演示了获取当前任务运行优先级服务的使用：

【例 APP_SAMPLE_05I】

file: example\WIN32\app_sample_05i.cpp

```
.....
00093     status_t status;
00094
00095     /* 更改前的任务运行优先级信息 */
00096     printf("I'm TA1: prio(%d) runprio(%d).\n",
00097           task_ident(), task_current());
00098
00099     /**
00100      * 更改任务 TA1 的运行优先级:
00101      * TA1(10) --> TA1(15).
00102      */
00103     status = task_change_runprio(task_self(), APP_PRIO_TA1R1);
00104
00105     /* 更改后的任务运行优先级信息 */
00106     printf("I'm TA1: prio(%d) runprio(%d).\n",
00107           task_ident(), task_current());
.....
```

5.13 任务休眠

实际应用中，经常需要将任务延时一段时间，到达指定的时间后继续执行。使用任务休眠服务接口就能够满足这个需求。该服务的接口原型如下：

```
status_t task_sleep(tick_t ticks);
```

调用任务休眠服务可以使得该任务暂停运行指定的时间。该接口接受一个传入参数，它指定了任务休眠的时间，表 5.7 给出接口参数说明：

表 5.7 接口 task_sleep 参数说明

参 数	说 明
ticks	指定的休眠时间，长度为 32 位的整数，单位为 tick，取值范围为 0 ~ 4 294 967 295

tick 是 RS-RTOS 系统的时钟，一个时钟节拍代表的实际时间，不同的系统通常是不一样的。通常一个 tick 代表的时间范围在几个毫秒到一百毫秒之间，在一个确定的系统这个值是确定的。为了计算方便，时钟值通常配置为 10 毫秒或者 20 毫秒。

时钟的值可以通过宏 RS_TICK_FREQ 换算，宏 RS_TICK_FREQ 定义了系统时钟的频率，其意义是 1 秒钟内的时钟节拍数，其换算关系如下：

1 时钟节拍 = 1 / RS_TICK_FREQ 秒

根据以上公式，可以换算出定时 1 秒所需要的时钟节拍数。例如以下代码（该代码可以在示例 APP_SAMPLE_05I 中找到），它的作用是使任务休眠 1 秒。

【例 APP_SAMPLE_05I】

file: example\WIN32\app_sample_05i.cpp

```
.....
00112      /* 休眠一秒 */
00113      task_sleep(RS_TICK_FREQ);
.....
```

任务休眠接口参数 `ticks` 是一个长度为 32 位的整数,取值范围为 0 ~ 4 294 967 295(包括 0 和 4 294 967 295), 当 `ticks` 为 0 时, 如下调用:

```
task_sleep(0);
```

接口将立即返回, 也就是说, 当传入 `ticks` 为 0 时, 调用任务休眠接口不会让任务进入休眠状态, 也不会引起任务切换。在系统时钟节拍为 20 毫秒情况下, 任务休眠服务能提供的最大休眠时间 (`ticks` 取值 4 294 967 295) 约为 995.2 天。在大多数情况下, 这个时间是足够的。如果需要更长的时间延时, 可以通过扩展任务休眠接口的方法实现。

5.14 唤醒任务

如果在一个任务的休眠时间还没到达之前, 需要马上唤醒该任务, 可以通过唤醒任务服务来实现, 其接口原型如下:

```
status_t task_wake(prio_t prio);
```

当某个任务正在休眠 (通过调用 `task_sleep`), 调用唤醒任务服务后, 该任务将从休眠状态中恢复运行。表 5.8 给出了参数说明:

表 5.8 接口 `task_wake` 参数说明

参 数	说 明
prio	需要唤醒的任务的优先级

例子 APP_SAMPLE_05J 演示了如何唤醒一个休眠中的任务。通过该示例运行结果, 也能说明唤醒任务可能会引起任务切换。

【例 APP_SAMPLE_05J】

file: example\WIN32\app_sample_05j.cpp

```
.....
00103 void app_task_1(arg_t)
00104 {
.....
```



```
00108      /**
00109      * 任务休眠 100 秒
00110      */
00111      task_sleep(RS_TICK_FREQ * 100);
00112      .....
00120 }
00121
00122 /* 任务 TA2 入口 */
00123 void app_task_2(arg_t)
00124 {
00125     status_t status;
00126
00127     /**
00128     * 马上唤醒休眠中的任务 TA1
00129     */
00130     status = task_wake(APP_PRIO_TA1);
00131     .....
00138 }
00139
00140 .....
```

`task_sleep` 和 `task_wake` 是一对颇有意思的接口，一个任务只能休眠自己，唤醒别人，永远不要试图使用 `task_wake` 唤醒自己。这是为什么呢？

5.15 挂起任务

挂起任务就是使任务的一切活动处于暂停状态，但是保留任务所占用系统资源以及状态信息，以便继续恢复任务的运行。挂起任务的接口如下：

```
status_t task_suspend(prio_t prio);
```

挂起任务接口需要一个传入参数，它指定了需要挂起的任务的优先级。表 5.9 给出该接

口的参数说明：

表 5.9 接口 `task_suspend` 参数说明

参 数	说 明
prio	待挂起的任务的优先级

当一个任务处于挂起状态，任务所有的活动都停止，就像在一种凝固的状态那样，包括任务的休眠时间也会被暂停。任务可以挂起自身，也可以挂起别的任务，当任务被挂起之后，如果需要重新唤醒它，必须通过 `task_resume` 服务完成。

例子 APP_SAMPLE_05K 演示了挂起任务服务的使用方法：

【例 APP_SAMPLE_05K】

file: example\WIN32\app_sample_05k.cpp

```
.....
00102 /* 任务 TA1 入口 */
00103 void app_task_1(arg_t)
00104 {
00105     status_t status;
.....
00109     /**
00110      * 挂起任务 TA2
00111      */
00112     status = task_suspend(APP_PRIO_TA2);
.....
00119 }
00120
00121 /* 任务 TA2 入口 */
00122 void app_task_2(arg_t)
00123 {
.....
00131 }
.....
```

使用挂起任务服务时，有些情况是需要注意的。在任务拥有互斥量、信号量等资源时，挂起该任务，可能会引起死锁。因为被挂起任务占用的资源无法得到及时释放，从而引致需要这些资源的其他任务进入阻塞状态，而这种等待状态可能是无限期的。在应用时必须对这些影响进行评估。

5.16 恢复任务

当创建任务时使用了 RS_OPT_SUSPEND 选项，该任务就会被设置成挂起状态。另外，使用 task_suspend 服务，也可以挂起一个任务。令挂起任务恢复执行的唯一方法就是通过调用接口：

```
status_t task_resume(prio_t prio);
```

使任务从挂起状态恢复。表 5.10 给出了该接口参数说明：

表 5.10 接口 task_resume 参数说明

参 数	说 明
prio	待恢复的任务的优先级

例子 APP_SAMPLE_05L 演示了恢复任务服务接口的使用方法：

【例 APP_SAMPLE_05L】

file: example\WIN32\app_sample_05l.cpp

```
.....
00120 /* 任务 TA2 入口 */
00121 void app_task_2(arg_t)
00122 {
00123     status_t status;
.....
00127     /**
00128     * 恢复任务 TA1
```

```
00129      */
00130      status = task_resume(APP_PRIO_TA1);
.....
00136 }
.....
```

任务不能使用 `task_resume` 恢复自身运行。挂起任务与恢复任务服务常用于系统的调试。

5.17 任务保护

前面曾经提到，如果某个任务申请了一段内存，在还没有释放它之前，删除该任务，任务申请的内存将无法回收，导致内存泄漏。如下代码：

【例 APP_SAMPLE_05M】

file: example\WIN32\app_sample_05m.cpp

```
.....
00105      char * p;
.....
00112      p = (char*)mpool_alloc(100);
00113
00114      if (p != NULL) {
00115
00116          /**
00117           * 这里任务申请了一段内存.
00118           * 在还没有释放它之前,如该任务被删除将导致内存泄漏.
00119           * 以防任务被不安全的删除,应置于任务保护区间
00120           */
00121
00122          mpool_free(p);
00123      }
.....
```

如何确保任务能够被安全删除呢？从另一个角度考虑，只要确保在任务使用关键资源时，不被其他任务删除，就可以达到保护资源的目的。为此，RS-RTOS 系统提供一对的接口，任务保护与解除保护，它们的接口原型是：

```
void task_protect(void);
```

```
void task_unprotect(void);
```

调用 `task_protect` 后，任务将进入保护状态。在任务保护状态下，任何删除该任务的操作都将失败，调用者将得到一个错误信息。任务保护与解除保护服务必须成对使用，例子 APP_SAMPLE_05M 演示了它们的使用方法：

【例 APP_SAMPLE_05M】

file: example\WIN32\app_sample_05m.cpp

```
.....
00105     char * p;
00106
00107     /**
00108      * 保护任务,在任务被保护期间,防止任务被删除,
00109      */
00110     task_protect();
00111
00112     p = (char*)mpool_alloc(100);
00113
00114     if (p != NULL) {
00115
00116         /**
00117          * 这里任务申请了一段内存.
00118          * 在还没有释放它之前,如该任务被删除将导致内存泄漏.
00119          * 以防任务被不安全的删除,应置于任务保护区间
00120          */
00121
```

```
00122         mpool_free(p);
00123     }
00124
00125     /**
00126      * 退出保护
00127      */
00128     task_unprotect();
.....
```

在任务保护状态下，任务不会被意外删除，这种保护是强制性的。当任务使用一些重要的资源时，这种强制的保护措施，可以确保系统稳定运行。

此外，任务保护服务还能保护任务的优先级不被修改。在任务保护状态下，修改该任务的优先级会返回一个错误信息。例子 APP_SAMPLE_05N 演示了任务保护的另一个用途：

【例 APP_SAMPLE_05N】

file: example\WIN32\app_sample_05n.cpp

```
.....
00093     prio_t myslef;
00094
00095     /**
00096      * 保护任务,在任务被保护期间,防止任务被删除或修改优先级.
00097      */
00098     task_protect();
00099
00100     /* 获得任务标识 */
00101     myslef = task_ident();
00102
00103     /**
00104      * 在任务保护状态下,确保任务优先级不会发生改变
00105      */
00106
00107     task_change_prio(myslef, APP_PRIO_TA1R1);
```

```
00108
00109     /**
00110     * 退出保护
00111     */
00112     task_unprotect();
.....
```

任务保护和解除保护服务必须成对使用，并且允许嵌套使用，嵌套的最大深度不得超过 256，当最后一层解除服务接口被调用时，才能退出任务保护状态。

5.18 解除保护

请参考上一节 5.17 任务保护。

5.19 获取任务信息

在 RS-RTOS 系统种，大部分内核对象都提供外界获取该对象运行信息的服务。通过调用 task_info 可以获取任务的信息。获得的信息包括任务当前的状态、优先级、栈指针、栈起始地址、栈尺寸以及栈的使用情况等。获取任务信息服务的接口原型如下：

```
status_t task_info(tinfo_t __p_* info);
```

该接口传入一个 tinfo_t 结构（tinfo_t 是 task information 的缩写）的指针，用来存放服务接口返回的各信息值，tinfo_t 结构参数详细说明在表 5.11 中给出：

表 5.11 接口 task_info 参数 tinfo_t 结构说明

参 数	说 明
prio	任务的优先级，作为接口唯一的传入参数，同时也作为传出参数
name	任务名字

state	任务运行状态
stack_base	任务栈起始地址
stack_size	任务栈尺寸
stack_used	任务栈使用字节数
stack_free	任务栈空闲字节数

tinfo_t 结构中，第一个参数 prio 是传入值，其他成员变量存放对应信息的返回值。例子 APP_SAMPLE_05O 演示了如何获得任务运行状态等信息：

【例 APP_SAMPLE_05O】

file: example\WIN32\app_sample_05o.cpp

```

.....
00092     tinfo_t info;    /* 用于存放任务信息的结构 */
00093     status_t status; /* 返回状态 */
00094
00095
00096     /* 需要获取的任务 */
00097     info.prio = task_self();
00098
00099     /* 获取任务信息 */
00100     status = task_info(&info);
00101
00102     if (status == RS_EOK) {
00103
00104         /* 显示任务信息 */
00105         printk("==Task Infomation=====\n");
00106         printk("Task Name   : %s\n", info.name);
00107         printk("Task Prio   : %d\n", info.prio);
00108         printk("Task State : %#04p\n", info.state);
00109         printk("Stack-Base : %#p\n", info.stack_base);
00110         printk("Stack-Size : %d\n", info.stack_size);
00111         printk("Stack-Used : %d\n", info.stack_used);

```



```
00112      printk("Stack-Free : %d\n", info.stack_free);
00113      }
.....
```

当返回值 `status` 等于 `RS_EOK` 时，就表明成功获得了信息。

5.20 任务执行综述

在 RS-RTOS 系统中可以运行 4 类程序：初始化、任务、中断服务、定时器服务。图 5.4 显示了这 4 类程序的关系。

初始化程序是第一种类型，它从系统启动开始运行。初始化结束后，RS-RTOS 系统进入任务调度循环（Task Scheduling Loop）。

任务调度循环查找就绪队列中最高优先级的任务，并把控制权交给它。在任务结束后，或者是有更高优先级的任务准备执行时，任务调度循环再次执行，继续查找下一个最高优先级的任务。这种任务与调度循环交替执行的过程是实时系统普遍采用的方式。

中断服务是实时系统重要的程序类型。假如没有中断服务，要对外界变化作出及时反应是非常困难的。中断服务是怎样产生的呢？硬件检测到中断信号后，把它发送到处理器中断处理单元，处理器中断服务机构保存当前正在执行程序的环境信息（通常保存在任务栈里），然后控制权被转移到一段预先定义的代码，这段代码被称为中断服务例程（ISR）。多数情况下，中断在任务执行或者调度循环执行时发生，但也有可能在中断服务例程或者定时器服务运行时发生。

定时器和中断非常相似，在应用程序看来，定时器的硬件实现（通常使用硬件发生周期性的中断信号）被隐藏了。应用可以使用定时器进行超时控制，完成周期性工作或者清除看门狗服务。和中断服务程序一样，定时器服务可以中断任务的执行，和中断服务不同的是，定时器服务程序之间不会相互打断。

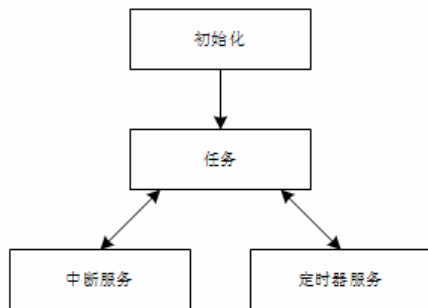


图 5.4 RS-RTOS 系统程序类型

5.21 任务状态

理解任务的状态是理解多任务编程的关键。任务有 4 种状态，分别是就绪态、阻塞态、挂起态、运行态。图 5.5 给出了这些状态的转移图。

任务在准备好被调度执行是处于就绪态。一个处于就绪态的任务不会被调度执行，直到它成为就绪队列（Ready Queue）中具有最高优先级的任务时，那么系统调度该任务执行，这个任务就转入执行态；如果另一个具有更高优先级的任务被加入就绪队列，那么当前执行的任务就再次转入就绪态，而高优先级的任务转入执行态。每次抢占发生，就会有就绪态和执行态之间的转换。

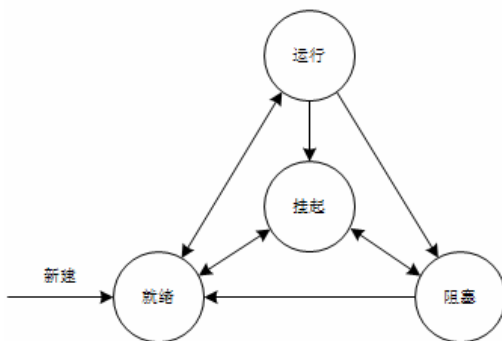


图 5.5 任务状态

在同一时刻，只能有一个任务处于执行态。处于执行态的任务将占用处理器资源。阻塞态的任务不会被调度执行，处在阻塞态的原因可能是任务需要休眠指定的时间（休眠也可以理解为在等待时间资源）、或者任务正在等待消息、信号量、互斥量、事件等资源。一旦休眠完成，或者等待的资源就绪，任务就将转入到就绪状态。

处在就绪态、阻塞态、运行态的任务，都能被 `task_suspend` 服务挂起，任务因此转入挂起态。处于挂起态的任务的一切活动都会暂停，系统不会调度挂起状态的任务，但是保留任务所占用系统资源及其状态。

系统中只有就绪队列（Ready Queue）中的任务可以被调度执行。当系统调度器需要选择一个任务执行时，它就会从就绪队列中选择优先级最高的任务。无论是什么原因，如果一个任务在执行时被打断，则它执行的上下文环境会被保存，该任务被放回到就绪队列中，等待下一次调度。而阻塞队列（Blocked Queue）中的线程不会被执行，因为它们在等待尚不可得的资源，它们或者在休眠（等待时间资源），或者在等待消息、事件等资源。只有任务的阻塞条件被清除后，才能回到就绪队列中去，等待再次调度运行。

5.22 任务设计

嵌入式系统应用软件设计过程中，任务设计是非常关键的一环，它涉及到整个应用框架，设计者需要较强系统的掌控能力。但是，依然可以通过遵循以下原则，达到更佳的性能和更小的目标代码。

- ◆ 最小化应用使用的任务的数目；
- ◆ 谨慎的安排任务优先级；
- ◆ 任务功能尽可能的独立；
- ◆ 避免运行时更改任务优先级。

5.23 任务的内部结构

创建一个任务时，系统给任务将分配一个任务控制块（TCB）资源。TCB 位于一个以优先级为索引的数组里，如图 5.6 所示。

调度器重新调度一个任务执行时，需要在就绪队列中选择最高优先级的任务。如何能够快速的查找指定任务，是加速调度器进行任务切换的关键。在 RS-RTOS 系统中，为了实现任务的快速定位。调度器根据优先级位图表，查找任务的优先级，然后通过图 5.6 控制块数组定位 TCB 的位置。

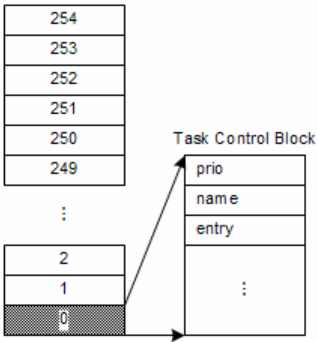


图5.6 任务控制块数组

5.24 总结

任务是动态的数据结构，它构成了实时操作系统 RS-RTOS 的基础。

RS-RTOS 系统提供了 15 个和任务相关的系统调用，涉及任务的创建、删除、修改和各种控制。

任务控制块（Task Control Block，TCB）是用来保存任务运行时信息的数据结构。当任务切换时，它也被用来保存任务的上下文信息。

应用开发人员可以在创建任务时指定多个选项，包括任务的优先级，是否创建挂起任务。

并且大多数参数允许在任务创建后进行修改。

一个任务可以主动把控制权让给其他任务，也可以挂起和恢复其他任务的执行，也可以对任务自身进行挂起。

任务可以自我保护，防止任务被其他任务删除，或者被更改优先级。

任务有 4 种状态：就绪态、阻塞态、运行态、挂起态。任一时刻只能有一个任务在运行。其他的任务在就绪队列或者阻塞队列中。就绪队列中的任务可以被调度执行，阻塞队列的任务不能被调度执行。

第 6 章

互斥量

6.1 介 绍

很多情况下，需要保障任务独自访问共享资源或者临界区。可能几个任务都想访问这些资源，所以必须同步它们的行为，实现独占式的访问。互斥量是专门设计用来提供互斥访问，避免任务之间冲突和资源交叉访问的工具。本章将讨论互斥量各方面的特性。

互斥量是一种独占资源，在同一时刻，它只能被一个任务占有，而且占有它的任务可以反复申请这个互斥量。也就是说，互斥量可以被嵌套使用，最大嵌套深度是 4 294 967 295 层（也就是 $2^{32} - 1$ 次）。同理，该任务也必须释放这个互斥量同样次数，互斥量才能再次变为可用。

只有任务能够拥有互斥量，不能在初始化程序、中断服务程序、定时器服务程序中获取和释放互斥量。

6.2 保护临界区

临界区是一段程序代码，在执行时不能被其他任务打断。为了实现这个目标，可以使用互斥量对临界区进行保护。图 6.1 显示了一个临界区。为了进入该临界区，任务必须首先得到一个互斥量。当任务进入临界区时，它首先申请互斥量，如果成功申请到互斥量，则继续

执行临界区程序，在退出临界区后释放互斥量。



图6.1 访问临界区资源

6.3 共享资源的互斥访问

互斥量也可以提供对共享资源的互斥访问。一个任务访问受保护的资源前必须得到互斥量。如果任务想对多个资源进行独立访问，那么必须为每个资源提供一个互斥量。这样，任务首先要申请到对应的互斥量，然后才能访问特定的资源，图 6.2 显示了这个过程。为了访问两个不同的共享资源，任务需要申请两个互斥量，才能使用共享资源，完成访问后，依次把两个互斥量释放掉。



图6.2 访问共享资源

6.4 互斥量控制块

互斥量控制块（Mutex Control Block，MCB）是用来保存运行时（Run-time）互斥量状态信息的数据结构。表 6.1 给出了构成 MCB 的主要属性项。

MCB 可以存放在内存的任意位置，通常把 MCB 定义成全局变量，以便它能够被所有相

关函数访问。需要注意的是，如果 **MCB** 在一个函数内部被创建（定义为函数内部变量），存储该 **MCB** 的内存空间将从函数栈中分配。通常应该避免这种情况，因为一旦该函数返回，它使用过的栈将被释放掉。

大多数情况下，开发者不需要了解 **MCB** 的内容。但某些时候，特别在调试时，观察某些属性项的值就显得非常的必要，但是绝对不可以手动修改它的值。

表 6.1 互斥量控制块 **MCB** 主要属性

项	描 述
name	互斥量的名字
ceiling	互斥量的优先级
owner	互斥量所有者

6.5 互斥量服务综述

附录 G 包含了与互斥量相关的各项服务接口的详细信息。包括函数的原型、简要介绍、参数、返回值、调用者、阻塞状态、注意项。表 6.2 包括系统提供的互斥量服务接口的列表。在本章后面的部分，将逐个介绍这些服务，考察这些服务各方面的特性，还将给出一些例子和图示说明。

表 6.2 互斥量服务接口

服 务	描 述
mutex_create	创建互斥量
mutex_delete	删除互斥量
mutex_wait	获取（等待）互斥量
mutex_trywait	获取（无等待）互斥量
mutex_release	释放互斥量
mutex_info	获取互斥量状态信息

6.6 创建互斥量

每个互斥量必须声明为 `mutex_t` 数据类型。当定义一个互斥量时，系统就会创建一个互斥量结构。在使用一个互斥量时，必须对其进行初始化，使用接口：

```
status_t  mutex_create(mutex_t __p_* mut, name_t name, prio_t prio);
```

完成了初始化一个互斥量的所有工作。创建一个互斥量需要给互斥量指定一个名字，指定互斥量的优先级。表 6.3 给出了该接口参数说明：

表 6.3 接口 `mutex_create` 参数说明

参 数	说 明
mut	指向互斥量结构的指针
name	互斥量的内核名字
prio	互斥量的最高限优先级

例子 APP_SAMPLE_06A 演示了如何创建一个互斥量：

【例 APP_SAMPLE_06A】

file: example\WIN32\app_sample_06a.cpp

```
.....
00052 enum {
.....
00056     /* 互斥量优先级 */
00057     APP_PRIO_MU1     = 9,
00058 };
.....
00064 /* 互斥量 */
00065 mutex_t app_mu1;
.....
00100     status_t status;
00101
```



```
00102      /* 初始化互斥量 MU1 */
00103      status = mutex_create(
00104          &app_mu1,
00105          build_name('M', 'U', '1', '\0'),
00106          APP_PRIO_MU1);
.....
```

创建互斥量时，必须指定一个合法的互斥量结构指针，并且不允许传入一个空指针（NULL），否则将引发一个断言错误。互斥量的内核名字通过 `build_name` 建立，也可以指定为 0，表示不使用内核对象名字。互斥量的优先级也叫最高限优先级，当某个任务获得该互斥量时，如满足特定的条件，该任务的运行优先级将被提升为互斥量的最高限优先级。互斥量的优先级应高于所有使用该互斥量的任务之中最高优先级的那个任务。

6.7 删除互斥量

当互斥量不再被使用，通过删除互斥量，释放不必要的资源开销。删除互斥量服务的接口原型如下：

```
status_t  mutex_delete(mutex_t __p_* mut);
```

删除一个互斥量，必须确保该互斥量不再被使用，并且没有任务锁定它。表 6.4 给出了该接口参数说明：

表 6.4 接口 `mutex_delete` 参数说明

参 数	说 明
mut	指向待删除的互斥量结构的指针

6.8 获取（等待）互斥量

任务通过获取互斥量服务申请对互斥量的所有权。互斥量的所有权是独占性的，同一时刻只能有一个任务拥有互斥量。获取互斥量服务接口如下：

```
status_t mutex_wait(mutex_t __p_* mut, tick_t ticks);
```

如果没有其他任务拥有这个互斥量，调用该服务的任务将获得这个互斥量；如果互斥量已经被调用的服务拥有，则该服务增加拥有互斥量的引用计数，并返回成功；如果互斥量已经被其他任务拥有，则调用该服务的任务将进入等待状态，直到互斥量有效，或者指定的超时返回。表 6.5 给出了该接口参数说明：

表 6.5 接口 `mutex_wait` 参数说明

参 数	说 明
<code>mut</code>	指向互斥量结构的指针
<code>ticks</code>	指定的超时时间，单位为系统节拍（Tick），如指定 <code>RS_WAIT_FOREVER</code> 表示无限期等待互斥量有效。

注意，表中 `RS_WAIT_FOREVER` 是内核定义的宏，其值为 0，也就是说，当超时 `ticks` 参数设置为 0 时，如果互斥量不可用，调用该接口的任务将一直等待，直到所需的互斥量可用。这个特性使得任务必须获得了互斥量之后，才能进入临界区或使用互斥资源。

例子 APP_SAMPLE_06B 演示了获取互斥量服务的使用方法：

【例 APP_SAMPLE_06B】

file: example\WIN32\app_sample_06b.cpp

```
.....
00064 /* 互斥量 */
00065 mutex_t app_mu1;
.....
00097 /* 任务 TA1 入口 */
00098 void app_task_1(arg_t)
```

```
00099 {
00100     status_t status;
00101
00102     /* 创建互斥量 MU1 */
.....
00112     /* 获取互斥量 MU1 的控制权,
00113      * 如互斥量无效,任务将进入等待状态. */
00114     status = mutex_wait(&app_mu1, RS_WAIT_FOREVER);
00115
00116     if (status == RS_EOK) {
00117
00118         /**
00119          * 使用临界资源
00120          */
00121
00122         /* 释放互斥量 MU1 的控制权. */
00123         mutex_release(&app_mu1);
00124     }
.....
00130 }
.....
```

6.9 获取（无等待）互斥量

顾名思义，无等待获取互斥量服务使得任务无需等待就可以获得互斥量。该服务的接口原型为：

```
status_t  mutex_trywait(mutex_t __p_ mut);
```

接口仅使用一个参数，表 6.6 给出了参数说明：

表 6.6 接口 `mutex_trywait` 参数说明

参 数	说 明
<code>mut</code>	指向互斥量结构的指针

`mutex_trywait` 提供了不同于 `mutex_wait` 的另一种获取互斥量的方法：任务调用该服务不会进入等待状态，即便该互斥量不可用，接口也会立刻返回。但是，该服务不能抢占已经被其他任务占有的互斥量，在这种情况下，将返回错误代码，指示互斥量处于不可用状态。例子 APP_SAMPLE_06C 演示了无等待获取互斥量服务的使用方法：

【例 APP_SAMPLE_06C】

file: example\WIN32\app_sample_06c.cpp

```
.....
00064 /* 互斥量 */
00065 mutex_t app_mu1;
.....
00097 /* 任务 TA1 入口 */
00098 void app_task_1(arg_t)
00099 {
00100     status_t status;
00101
00102     /* 创建互斥量 MU1 */
.....
00112     /* 获取互斥量 MU1 的控制权,
00113      * 如互斥量无效,接口将返回错误代码. */
00114     status = mutex_trywait(&app_mu1);
00115
00116     if (status == RS_EOK) {
00117
00118         /**
00119          * 使用临界资源
00120          */
```

```
00121
00122          /* 释放互斥量 MU1 的控制权. */
00123          mutex_release(&app_mu1);
00124      }
.....
00130 }
.....
```

使用 `mutex_trywait` 永远不用担心进入等待状态。只有任务能够拥有互斥量，不能在初始化程序、中断服务程序、定时器服务程序中获取和释放互斥量。

6.10 释放互斥量

当一个任务完成互斥资源的使用后，要尽快释放所占用的互斥量，以便资源能被其他任务及时的访问。释放一个互斥量使用接口：

```
status_t  mutex_release(mutex_t __p_* mut);
```

表 6.7 给出了参数说明：

表 6.7 接口 `mutex_release` 参数说明

参 数	说 明
mut	指向待释放互斥量结构的指针

假设任务拥有一个互斥量，任务将其释放，将会使得互斥量的引用计数器减 1，如果引用计数变成 0，则互斥量就会变得可用。如果任务的运行优先级被互斥量提升，那么释放该互斥量后，任务将自动恢复为占有互斥量之前的优先级运行。

例子 APP_SAMPLE_06B 演示释放一个互斥量的方法：

【例 APP_SAMPLE_06B】

file: example\WIN32\app_sample_06b.cpp

```
.....
```

```
00064 /* 互斥量 */
00065 mutex_t app_mu1;
.....
00097 /* 任务 TA1 入口 */
00098 void app_task_1(arg_t)
00099 {
00100     status_t status;
00101
00102     /* 创建互斥量 MU1 */
.....
00112     /* 获取互斥量 MU1 的控制权,
00113      * 如互斥量无效,任务将进入等待状态. */
00114     status = mutex_wait(&app_mu1, RS_WAIT_FOREVER);
00115
00116     if (status == RS_EOK) {
00117
00118         /**
00119          * 使用临界资源
00120          */
00121
00122         /* 释放互斥量 MU1 的控制权. */
00123         mutex_release(&app_mu1);
00124     }
.....
00130 }
.....
```

6.11 获取互斥量信息

通过调用 `mutex_info` 可以获取互斥量的信息。得到的信息包括互斥量所有者的优先级，互斥量的优先级。获取互斥量信息服务的接口原型为：

```
status_t mutex_info(mutex_t __p_* mut, mutinfo_t __out_* info);
```

该接口传入一个 `mutinfo_t` 结构（`mutinfo_t` 是 `mutex information` 的缩写）指针参数，用来存放服务接口返回的各信息值，表 6.8 给出 `mutex_info` 服务接口参数详细说明：

表 6.8 接口 `mutex_info` 参数说明

参 数		说 明
mut		指向互斥量结构的指针
info	name	返回互斥量名字
	prio	返回互斥量的优先级
	owner	返回互斥量的所有者优先级
	reference	返回互斥量引用计数

例子 APP_SAMPLE_06D 演示了利用接口 `mutex_info` 获取互斥量的信息：

【例 APP_SAMPLE_06D】

file: example\WIN32\app_sample_06d.cpp

```
.....
00064 /* 互斥量 */
00065 mutex_t app_mu1;
.....
00097 /* 任务 TA1 入口 */
00098 void app_task_1(arg_t)
00099 {
00100     status_t status;
00101     mutinfo_t info;
00102
```

```

00103    /* 创建互斥量 MU1 */
.....
00113    /* 获取互斥量 MU1 信息. */
00114    status = mutex_info(&app_mu1, &info);
00115
00116    if (status == RS_EOK) {
00117
00118        /* 显示互斥量信息 */
00119        printk("==Mutex Infomation=====\n");
00120        printk("Mutex Name   : %s\n", info.name);
00121        printk("Mutex Prio   : %d\n", info.prio);
00122        printk("Mutex Owner  : %d\n", info.owner);
00123        printk("Mutex Refer. : %d\n", info.reference);
00124    }
.....
00132 }
.....

```

6.12 避免死锁

使用信号量带来的一个潜在问题是死锁。死锁时，两个或多个任务由于互相等待占用的资源而永远挂起。图 6.3 显示了一种死锁的情况，图中

- 1) 任务 TA1 占有互斥量 MU1；
- 2) 任务 TA2 占有互斥量 MU2；
- 3) 任务 TA1 因为申请互斥量 MU2 而阻塞；
- 4) 任务 TA2 因为申请互斥量 MU1 而阻塞。

这样，两个任务陷入死锁，因为它们都将永远等待另一个任务拥有的互斥量。

为了清楚地说明死锁情况，下面再列举一个例子说

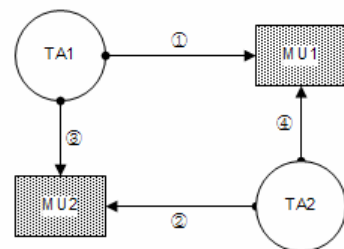


图6.3 导致死锁的操作

明。设系统有打印机、读卡机各一台，它们被任务 A 和任务 B 共享。两个任务并发执行，它们按下列次序请求和释放资源：

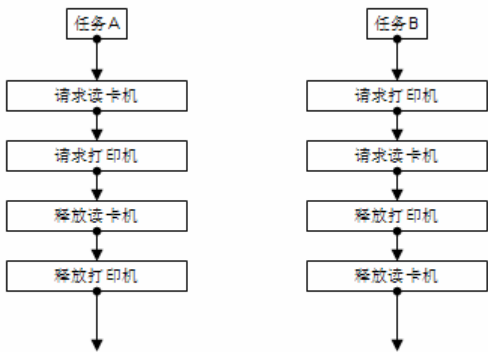


图6.4 可能导致死锁的资源请求

它们执行时，相对速度无法预知，当出现任务 A 占用了读卡机，任务 B 占用了打印机后，任务 A 又请求打印机，但因打印机被任务 B 占用，故任务 A 处于等待资源状态；这时，任务 B 执行，它又请求读卡机，但因读卡机被任务 A 占用而也只好处于等待资源状态。它们分别等待对方占用的资源，致使无法结束这种等待，产生了死锁。

为了避免出现死锁，必须破坏死锁产生的条件，原理上，死锁必需满足以下条件才能发生：

- 1) 资源互斥；
- 2) 资源被持有的同时，其他使用者在等待该资源；
- 3) 允许占先持有资源；
- 4) 存在一个循环等待条件，例如：R1 等待 R2，R2 等待 R3，R3 等待 R1。

解决死锁问题，必须确保以上条件至少有一个不会发生，才能保证死锁不会发生。有以下几个策略可以解决这个问题，只要实现其中的一条就可以达到目的。

- 1) 强制每一个任务在同一时刻只能占有一个互斥量；
- 2) 任务必须按相同的顺序申请互斥量；
- 3) 获取互斥量时使用超时属性，让任务能从死锁中恢复。

6.13 互斥量示例

下面给出一个示例程序来展示如何使用互斥量保护共享资源。这个例子中有两个任务：Waveform Draw 和 Message Display，分别负责显示波形和文本信息。它们共享资源是显示器 Display。任务的优先级别安排顺序是：Waveform Draw（中优先级，Medium），Message Display（低优先级，Low），因为任务 Waveform Draw 和 Message Display 共享资源 Display，而共享资源的优先级应该高于使用它们的任务中最高的优先级。所以 Display 的最高限优先级为比任务 Waveform Draw 的优先级要高（即高优先级，High）。

该示例的运行过程是这样的：首先，最低优先级的任务 Message Display 运行，调用操作 DisplayMsg()。因为 Display 有一个互斥量，它锁定了资源，拥有 Display 资源后，任务 Waveform Draw 将以高优先级运行。在 Message Display 对资源操作期间，某个事件发生(示例中通过 task_resume 唤醒)，唤醒任务 Waveform Draw。此时任务 Message Display 优先级比任务 Waveform Draw 高，任务 Waveform Draw 将进入就绪队列等待。任务 Message Display 输出完成，释放资源 Display，运行优先级恢复为低优先级。守候在就绪队列中的任务 Waveform Draw 将获得处理器资源，进入运行状态。而低优先级的任务 Message Display 则被抢占，图 6.5 显示了这个过程。

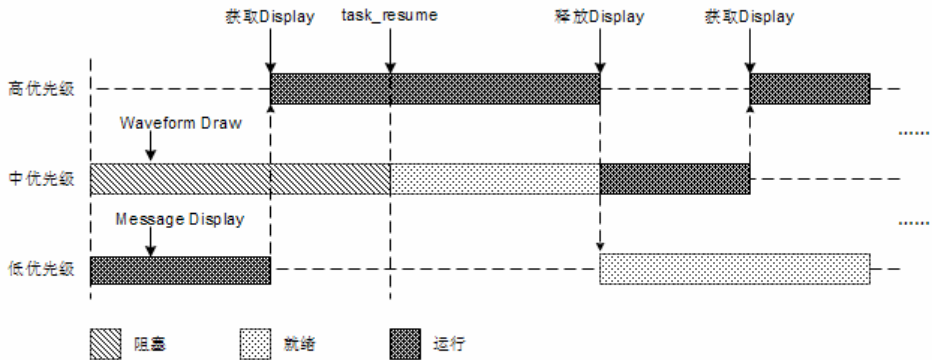


图6.5 互斥量应用示例

任务 Message Display 在取得互斥量 Display 后，运行优先级即被提升到互斥量的优先级，这种模式称为优先级的最高限模式（ceiling pattern）。在 RS-RTOS 系统中，互斥量还可以配置为另一种工作模式：优先级继承模式（inheritable pattern）。在优先级继承模式下，当

一个任务获取一个互斥量时，并不会马上提高其运行优先级，仅当另一个更高优先级的任务试图获取该互斥量时，才提高拥有该互斥量的任务的运行优先级。

最高限模式一个很有意义特性是，它破坏了死锁产生的必要条件，可以防止任务出现死锁。相对来说，优先级继承模式实现的要精致一些，它避免了最高限模式下频繁发生的不必要的优先级切换。在不是很复杂的系统，只要合理安排共享资源的使用，任务死锁完全是可以避免的，那么，优先级继承模式是更好的选择。

节 6.14 给出了该示例的代码清单，请读者运行该示例，观察其输出结果。

6.14 示例 APP_SAMPLE_06E 代码

【例 APP_SAMPLE_06E】

file: example\WIN32\app_sample_06e.cpp

```
.....
00038 #include "inc/kapi.h"
00039 #include "app_sample.h"
00040
00041 #ifdef APP_SAMPLE_06E
00042
00043 /**
00044  * 例子 APP_SAMPLE_06E 演示如何使用互斥量保护共享资源
00045  */
00046
00047
00048 /* 任务栈大小 */
00049 #define APP_STACK_SIZE      1024 * 32
00050
00051
00052 enum {
00053     /* 任务优先级 */
00054     APP_PRIO_WAVEFORM      = 11,
```

```
00055     APP_PRIO_MESSAGE    = 12,
00056
00057     /* 互斥量优先级 */
00058     APP_PRIO_DISPLAY      = 10,
00059 };
00060
00061
00062 /* 任务栈 */
00063 stack_t app_stack_message[APP_STACK_SIZE];
00064 stack_t app_stack_waveform[APP_STACK_SIZE];
00065
00066 /* 互斥量 */
00067 mutex_t app_mut_display;
00068
00069
00070 void app_task_message_display(arg_t arg);
00071 void app_task_waveform_draw(arg_t arg);
00072
00073
00074 /* 硬件初始化 */
00075 void hardware_initialize(void)
00076 {
00077
00078 }
00079
00080
00081 /* 应用初始化 */
00082 void application_initialize(void)
00083 {
00084     status_t status;
00085
```

```
00086    /* 打印关于示例一些信息 */
00087    APP_SAMPLE_INFO;
00088
00089
00090    /* 初始化互斥量 Display */
00091    status = mutex_create(
00092        &app_mut_display,
00093        build_name('D', 'I', 'S', '\0'),
00094        APP_PRIO_DISPLAY);
00095
00096    ASSERT(status == RS_EOK);
00097
00098
00099    /* 创建任务 Message Display */
00100    status = task_create(
00101        APP_PRIO_MESSAGE,
00102        build_name('M', 'S', 'G', '\0'),
00103        app_task_message_display, 0,
00104        app_stack_message, APP_STACK_SIZE,
00105        0);
00106
00107    ASSERT(status == RS_EOK);
00108
00109
00110    /* 创建任务 Waveform Draw */
00111    status = task_create(
00112        APP_PRIO_WAVEFORM,
00113        build_name('W', 'A', 'V', '\0'),
00114        app_task_waveform_draw, 0,
00115        app_stack_waveform, APP_STACK_SIZE,
00116        RS_OPT_SUSPEND);
```

```
00117
00118     ASSERT(status == RS_EOK);
00119 }
00120
00121 /* 任务 Message Display 入口 */
00122 void app_task_message_display(arg_t)
00123 {
00124     /* 显示信息 step.1 */
00125     printk("MESSAGE: step 1.\n");
00126
00127     /* 获取 Display 资源 */
00128     mutex_wait(&app_mut_display, RS_WAIT_FOREVER);
00129
00130     /**
00131      * 获取 Display 资源,显示 DisplayMsg()
00132      */
00133
00134     /* 显示信息 step.2 */
00135     printk("MESSAGE: step 2.\n");
00136
00137     /* 唤醒任务 Waveform Draw */
00138     task_resume(APP_PRIO_WAVEFORM);
00139
00140     /* 显示信息 step.3 */
00141     printk("MESSAGE: step 3.\n");
00142
00143
00144     /* 释放 Display 资源 */
00145     mutex_release(&app_mut_display);
00146
00147     /* 显示信息 step.4 */
```

```
00148     printk("MESSAGE: step 4.\n");
00149
00150     for (;;)
00151     {
00152         task_sleep(RS_TICK_FREQ);
00153     }
00154 }
00155
00156 /* 任务 Waveform Draw 入口 */
00157 void app_task_waveform_draw(arg_t)
00158 {
00159     /* 显示信息 step.1 */
00160     printk("WAVEFORM: step 1.\n");
00161
00162     /* 获取 Display 资源 */
00163     mutex_wait(&app_mut_display, RS_WAIT_FOREVER);
00164
00165     /**
00166      * 获取 Display 资源,显示 DisplayWave()
00167      */
00168
00169     /* 显示信息 step.2 */
00170     printk("WAVEFORM: step 2.\n");
00171
00172
00173     /* 释放 Display 资源 */
00174     mutex_release(&app_mut_display);
00175
00176     /* 显示信息 step.3 */
00177     printk("WAVEFORM: step 3.\n");
00178
```

```
00179     for (;;)
00180     {
00181         task_sleep(RS_TICK_FREQ);
00182     }
00183 }
00184
00185
00186 #endif
.....
```

6.15 互斥量的内部结构

每定义一个 `mutex_t` 类型互斥量，编译器就会分配一个互斥量控制块（MCB）。互斥量控制块用于保存互斥量运行时的状态信息，表 6.9 列出了 MCB 控制块结构说明：

表 6.9 互斥量控制块 MCB 结构说明

项	描 述
name	互斥量的名字
ceiling	互斥量的优先级
owner	互斥量所有者
reference	拥有互斥量的引用计数
init	初始化标志
waits	互斥量的等待队列

大多数情况下，开发者不需要了解 MCB 的内容。但某些时候，特别是在调试时，观察 MCB 项的值就很有用。尽管 RS-RTOS 允许观测 MCB 各项的值，但绝对不允许修改它。

6.16 总 结

互斥量用于对临界区代码或者共享资源的独占式访问。互斥量是一种独占资源，在同一时刻，只能被一个任务占有。

RS-RTOS 提供了多个对互斥量操作的服务接口，包括创建互斥量、删除互斥量、获取互斥量、无等待获取互斥量、释放互斥量、取得互斥量信息六个服务接口。

互斥量拥有优先级属性，该属性可以提高拥有互斥量的任务的运行优先级，从而避免出现优先级翻转。

互斥量可以被嵌套使用，最大嵌套深度是 $2^{32} - 1$ 。使用互斥量带来的一个副作用是死锁，本章讨论了死锁的产生必要条件，以及避免死锁的方法。

第 7 章

信号量

7.1 介绍

信号量（Semaphore）方法是荷兰学者 E.W.Dijkstra 在 1965 年提出的一种解决任务同步、互斥问题的通用的工具。

信号量的实例（Instance）是一个整型数据，其值代表实例数目。举例来说，如果信号量的值为 10，那么该信号量就有 10 个实例；如果为 0，那么该信号量就没有实例。施加在信号量上主要有两种操作，称做 PV 操作。P 操作测试信号量的值是否大于 0，如果是，则信号量值减 1，程序继续执行；如果不大于 0，则循环测试。V 操作只是简单的把信号量的值加 1。

信号量是一种非常灵活的工具，利用信号量的 PV 操作可以方便的解决任务通讯、同步、互斥等问题。

RS-RTOS 系统提供 32 位(可剪裁为 16 位或者 8 位)的信号量,计数的范围是 0 到 4 294 967 295 (或 0 到 $2^{32} - 1$, 包括 4 294 967 295 或 $2^{32} - 1$)。内核提供了较为丰富的操作接口,包括信号量的创建、删除、等待获取、无等待获取、释放、获取信号量信息。其中,等待获取、无等待获取信号量属于 P 操作,释放信号量属于 V 操作。

信号量也常用于提供互斥访问。用在互斥访问的信号量和互斥量功能类似,两者的主要区别是:信号量不支持所有权,不能解决优先级翻转问题,而所有权是互斥量的核心;但信号量使用比互斥量更加灵活,他可以被用来提供事件通知和多个任务间同步;信号量可以在

中断服务，定时器应用中使用，而互斥量只能用在普通任务空间。

互斥访问用来控制任务对某些特定资源的访问。提供互斥访问时，信号量的值代表资源的数目，也就是同时能够被几个任务访问。多数情况下，提供互斥访问的信号量的初值为 1，这意味着在任一时刻只有一个应用可以访问该资源。其值只可能为 1 或者 0 的信号量又称为二值信号量。

如果使用二值信号量实现互斥访问，则使用者必须避免已经控制了信号量的任务再次调用 P 服务（连续多次调用 P 服务）。第二次调用 P 服务会失败，如果没有设置超时时间，该任务将被永久挂起，令资源永远处于不可用状态。

信号量还可以被用来在生产者—消费者模式的应用中提供事件通知。在这种应用中，消费者在消费资源（例如队列中的数据）前试图获取信号量，生产者一旦生产了资源就增加信号量的值。换句话说，生产者把实例交给信号量，而消费者把实例从信号量中拿走。这类信号量通常初始值为 0，其直到生产者生产了资源后才会增加。

程序可以在初始化或者运行时创建信号量，信号量的初始值在创建信号量时指定。
RS-RTOS 系统中使用信号量的数量不受限制。

任务在申请当前值为 0 的信号量后可以被挂起，这取决于程序使用 P 操作及其选项。

当一个实例被释放给信号量并且当前有一个任务因等待该信号量而阻塞时，该任务将被唤醒并取走信号量一个实例，如果有多个任务因等待信号量而阻塞，则最高优先级的任务将被唤醒并取走信号量一个实例。

7.2 信号量属性

当定义一个信号量时，就创建了一个信号量控制块（SCB）。SCB 用于存储信号量运行时的状态信息，包括信号量的计数值、任务阻塞队列等。SCB 可以被放在内存中任何位置，可以通过申请动态内存获得，但通常把它定义为全局变量，以便不同的任务和程序能够访问。

信号量的属性可以在其控制块中找到。表 7.1 列出了信号量的主要属性：

表 7.1 信号量控制块 SCB 主要属性

项	描 述
name	信号量的名字
count	信号量的值（范围为 0 到 $2^{32} - 1$ ）

信号量的值是一个 32 位的整数，在资源受限的嵌入式系统，可配置为 8 位（可表示 0 到 255 个实例）或 16 位（可表示 0 到 65535 个实例），以节约内存资源，但会失去一定的兼容性。

7.3 避免死锁

使用信号量提供互斥访问时，最危险的陷阱是死锁。死锁时，两个或多个任务因为等待其他任务占有的信号量而阻塞，而这种阻塞是永久的，除非人为干预，否则无法自动恢复。请参考第 6 章互斥量关于死锁的讨论，已经讨论了如何避免死锁，以及从死锁中恢复的方法，这些方法对解决信号量死锁问题同样有效。

7.4 防止优先级翻转

与提供互斥访问的信号量相关的另一个陷阱是优先级翻转。优先级翻转发生的前提条件是：一个低优先级的任务占有了一个高优先级任务申请的互斥信号量。这种翻转是相当普遍的，并且不会带来严重的问题，但是，如果还有处在中间优先级的就绪任务，这种翻转就会持续不确定的时间。应用开发人员可以通过谨慎设置任务的优先级来避免这种情况，也可以暂时提高拥有信号量的任务的运行优先级来避免出现优先级翻转。不幸的是，信号量不能主动提高任务的运行优先级，而这恰恰是互斥量特有的性质。

7.5 信号量服务综述

附录 H 包含了与信号量相关的各项服务接口的详细信息。包括函数的原型、简要介绍、参数、返回值、调用者、阻塞状态、注意事项。

表 7.2 包括系统提供的信号量服务接口的列表。在本章后面的部分，将逐个介绍这些服务，还将给出一些例子和图示说明。

表 7.2 信号量服务接口

服 务	描 述
semaphore_create	创建信号量
semaphore_delete	删除信号量
semaphore_wait	获取（等待）信号量
semaphore_trywait	获取（无等待）信号量
semaphore_post	释放信号量
semaphore_info	获取信号量状态信息

7.6 创建信号量

每个信号量必须声明为 `sem_t` 数据类型。当定义一个信号量时，系统就会创建一个信号量结构（SCB）。在使用一个信号量前，必须对其进行初始化，使用接口：

```
status_t semaphore_create(sem_t __p_* sem, name_t name, count_t value);
```

对一个信号量进行初始化。创建一个信号量需要给信号量指定一个名字，和一个初始值。表 7.3 给出了该接口参数说明：

表 7.3 接口 semaphore_create 参数说明

参 数	说 明
sem	指向信号量的指针
name	信号量的名字
value	信号量初始值

例子 APP_SAMPLE_07A 演示了如何创建一个信号量：

【例 APP_SAMPLE_07A】

file: example\WIN32\app_sample_07a.cpp

```
.....
00048 /* 信号量 */
00049 sem_t app_sem1;
```

```
.....
00062      status_t status;
.....
00067      /* 创建信号量 SE1,初始值为 0 */
00068      status = semaphore_create(
00069          &app_sem1,
00070          build_name('S', 'E', '1', '\0'),
00071          0);
.....
```

创建信号量时，必须指定一个合法的信号量结构指针，并且不允许传入一个空指针（NULL），否则将引发一个断言错误。信号量的名字通过 `build_name` 建立，也可以指定为 0，表示不使用内核对象名字。信号量的初始值意义完全由应用定义，通常该值表示对应资源的初始实例数，信号量初始值是非负的整数。

7.7 删除信号量

当一个信号量不再被使用，通过删除信号量，释放不必要的资源开销。删除一个信号量服务的接口原型如下：

```
status_t semaphore_delete(sem_t __p_* sem);
```

表 7.4 给出了该接口参数说明：

表 7.4 接口 `semaphore_delete` 参数说明

参 数	说 明
sem	指向待删除的信号量的指针

注意：当删除一个信号量时，如果有任务正在等待该信号量资源，则所有等待该信号量的任务将被唤醒，并获得一个成功信息。要避免删除一个正在使用的信号量，以防可能由此产生的逻辑错误。

7.8 获取（等待）信号量

通过获取信号量服务申请信号量的资源实例。如果成功获取了信号量的实例，则信号量的值会被减 1。获取（等待）信号量服务的接口原型如下：

```
status_t semaphore_wait(sem_t __p_* sem, tick_t ticks);
```

如果信号量的值大于 0，调用该服务的将使信号量的值减 1；如果信号量的值为 0，表示信号量的资源实例已经用完，则任务将进入阻塞状态，直到信号量有效，或者指定的超时返回。表 7.5 给出了该接口参数说明：

表 7.5 接口 `semaphore_wait` 参数说明

参 数	说 明
<code>sem</code>	指向信号量的指针
<code>ticks</code>	指定的超时时间，单位为系统节拍（Tick），如指定 <code>RS_WAIT_FOREVER</code> 表示无限期等待，直到信号量有可用实例。

注意，表中 `RS_WAIT_FOREVER` 是内核定义的宏，其值为 0，也就是说，当超时 `ticks` 参数设置为 0 时，如果信号量的值为 0，调用该服务的任务将一直等待，直到所需信号量有实例可用。

例子 APP_SAMPLE_07B 演示了获取信号量实例的使用方法：

【例 APP_SAMPLE_07B】

file: example\WIN32\app_sample_07b.cpp

```
.....
00061 /* 信号量 */
00062 sem_t app_sem1;
.....
00094 /* 任务 TA1 入口 */
00095 void app_task_1(arg_t)
00096 {
```



```
00097     status_t status;
.....
00100     /* 创建信号量 SE1,初始值为 10 */
00101     status = semaphore_create(
00102         &app_sem1,
00103         build_name('S', 'E', '1', '\0'),
00104         10);
.....
00108     for (;;)
00109     {
00110         /* 获取信号量 SE1 实例,
00111          * 如信号量 SE1 无可实例,任务将进入等待状态. */
00112         status = semaphore_wait(&app_sem1, RS_WAIT_FOREVER);
.....
00125     }
00126 }
.....
```

7.9 获取（无等待）信号量

某些情况下，程序不希望由于信号量无实例可用而进入阻塞状态，或者程序不能被阻塞（如中断服务程序）。无等待获取信号量服务使得程序在获取信号量实例时无需等待。无等待获取信号量服务的接口原型为：

```
status_t semaphore_trywait(sem_t __p_* sem);
```

接口仅使用一个参数，表 7.6 给出了参数说明：

表 7.6 接口 `semaphore_trywait` 参数说明

参 数	说 明
-----	-----

sem	指向信号量的指针
-----	----------

`semaphore_trywait` 提供了无需等待而获取信号量实例的方法: 任务调用该服务不会进入等待状态, 即便该信号量没有实例可用, 接口也会立刻返回, 在这种情况下, 将返回错误代码, 指示信号量无实例可用。例子 **APP_SAMPLE_07C** 演示了无等待获取互斥量服务的使用方法:

【例 APP_SAMPLE_07C】

file: example\WIN32\app_sample_07c.cpp

```
.....
00061 /* 信号量 */
00062 sem_t app_sem1;
.....
00094 /* 任务 TA1 入口 */
00095 void app_task_1(arg_t)
00096 {
00097     status_t status;
00098     int i = 0;
00099
00100     /* 创建信号量 SE1,初始值为 10 */
.....
00108     for (;;)
00109     {
00110         /* 无等待获取信号量 SE1 实例,
00111          * 如信号量 SE1 无可实例,服务接口将返回错误代码. */
00112         status = semaphore_trywait(&app_sem1);
00113
00114         if (status == RS_EOK) {
00115
00116             /* 打印取得 SE1 的实例数目 */
00117             printk("Get SE1: %d.\n", ++i);
00118
```

```
00119         } else {
00120
00121             /* 信号量已无可实例 */
00122             printk("SE1 is not available.\n");
00123         }
00124
00125         .....
00128     }
00129 }
00130
00131 .....
```

与互斥量很重要的一个区别：除了普通任务，中断服务例程（ISR）、定时器应用也可以通过 `semaphore_trywait` 获得信号量实例，而互斥量只能被任务占有。任务间的互斥访问可以使用信号量或者互斥量；而任务、中断服务例程、定时器之间的互斥访问只能使用信号量，而不能使用互斥量。

7.10 释放信号量

释放信号量也叫投递信号量，将使信号量的计数值加 1。释放一个信号实例时，如果有任务因等待该信号量而阻塞，则阻塞队列中最高优先级的任务将被唤醒。

```
status_t semaphore_post(sem_t __p_* sem);
```

表 7.7 给出了参数说明：

表 7.7 接口 `semaphore_post` 参数说明

参 数	说 明
sem	指向待释放信号量的指针

例子 APP_SAMPLE_07D 演示释放信号量实例的方法：

【例 APP_SAMPLE_07D】

file: example\WIN32\app_sample_07d.cpp

```
.....
00061 /* 信号量 */
00062 sem_t app_sem1;
.....
00094 /* 任务 TA1 入口 */
00095 void app_task_1(arg_t)
00096 {
00097     status_t status;
00098
00099     /* 创建信号量 SE1,初始值为 0 */
.....
00107     /* 释放信号量 SE1 实例,
00108      * 成功调用该服务将使信号量 SE1 的值加 1. */
00109     status = semaphore_post(&app_sem1);
.....
00118 }
.....
```

使用信号量实现资源的互斥访问时，释放信号量与释放互斥量同义。值得注意的是：释放信号量另外一个语义是释出、投递，这个语义体现在生产——消费者模式，生产者不断释出信号量实例，但并不需要拥有信号量实例。而释放互斥量的任务必定是已经拥有了该互斥量。这就是为什么释放互斥量称 **release**，而释放信号量称 **post** 的原因，从中也可以体会到两者的区别。

7.11 获取信号量信息

通过调用 **semaphore_info** 可以获取信号量运行时的状态信息。得到的信息包括信号量名字，信号量的计数值（实例值）。获取信号量信息的接口原型为：

```
status_t semaphore_info(sem_t __p_* sem, seminfo_t __out_* info);
```

该接口传入一个 `seminfo_t` 结构（`seminfo_t` 是 `semaphore information` 的缩写）指针参数，用以存放服务接口返回的各信息值，表 7.8 给出 `semaphore_info` 服务接口参数详细说明：

表 7.8 接口 `semaphore_info` 参数说明

参 数		说 明
sem		指向信号量的指针
info	name	返回信号量的名字
	count	返回信号量的计数值

例子 APP_SAMPLE_07E 演示了利用接口 `semaphore_info` 获取信号量的信息：

【例 APP_SAMPLE_07E】

file: example\WIN32\app_sample_07e.cpp

```
.....
00061 /* 信号量 */
00062 sem_t app_sem1;
.....
00097     status_t status;
00098     seminfo_t info;
00099
00100     /* 创建信号量 SE1,初始值为 8 */
00101     status = semaphore_create(
00102         &app_sem1,
00103         build_name('S', 'E', '1', '\0'),
00104         8);
.....
00109     /* 获取信号量 SE1 信息. */
00110     status = semaphore_info(&app_sem1, &info);
00111
00112     if (status == RS_EOK) {
00113
00114         /* 显示信号量信息 */
```

```

00115      printk("==Semaphore Infomation=====\\n");
00116      printk("Semaphore Name   : %s\\n", info.name);
00117      printk("Semaphore Value : %d\\n", info.count);
00118      }
.....

```

7.12 信号量与互斥量异同

信号量和互斥量在很多方面类似，但也有区别，各有优点。表 7.9 对互斥量和信号量各方面特性的差异做了一个综合的比较：

表 7.9 互斥量与信号量特性对比

	互斥量	信号量
速度	比信号量稍慢	很快
资源消耗	比信号量高	极少
优先级翻转	可以避免优先级翻转问题	不能避免
互斥	互斥量的专项特性	通过二值信号量实现
任务同步	不支持	可以用信号量实现
事件通知	不支持	可以用信号量实现
中断服务	不能在中断服务中使用	可以在中断服务使用（通过无等待接口）
定时器应用	互斥量不能在定时器应用中使用	可以在定时器应用中使用信号量（通过无等待接口）

互斥量异常健壮，如果互斥对程序很关键，那么使用互斥量是好主意。不过，如果互斥不是很重要的因素，就使用信号量，因为它速度快，也耗费很少的资源。

第 8 章二值信号量将给出一个例子，演示如何使用信号量代替互斥量。

7.13 生产者—消费者问题

信号量主要用来提供互斥、事件通知以及同步。前面主要围绕信号量的互斥用法，对信号量和互斥量的特性做了比较。接下来，将展示一个使用信号量实现同步的例子。该例子应用了生产者—消费者模式。

一个生产者好比一个生产厂家，如康师傅，而一个消费者，例如小明。生产者和消费者之间还需要一个缓冲区——零售商，如沃尔玛商场。只有当厂家把生产出来的产品冰红茶放到商场上销售后，小明才可以从商场里买到冰红茶。这就是一个简单的同步问题。图 7.1 给出了该示例图示。

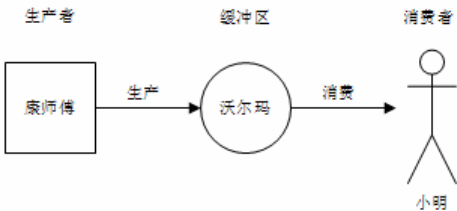


图7.1 生产者—消费者

用信号量实现如下：

<div>生产者：</div> <div>for(;;) { post(v) }</div>	<div>消费者：</div> <div>for(;;) { wait(v) }</div>
--	--

这里，信号量是产品的商场，它的值代表产品的数量；对信号量的释放（post）操作是生产的过程；而获取（wait）操作是产品的消费过程。其完整示例代码如下：

【例 APP_SAMPLE_07F】

file: example\WIN32\app_sample_07f.cpp

```
.....  
00063 /* 信号量 */
```

```
00064 sem_t app_sem1;
.....
00078 /* 应用初始化 */
00079 void application_initialize(void)
00080 {
00081     status_t status;
.....
00087     /* 创建信号量 SE1,初始值为 0 */
00088     status = semaphore_create(
00089         &app_sem1,
00090         build_name('S', 'E', '1', '\0'),
00091         0);
00092
00093     ASSERT(status == RS_EOK);
.....
00114 }
00115
00116 /* 任务 TA1 入口 */
00117 void app_task_1(arg_t)
00118 {
00119     for (;;)
00120     {
00121         /* 生产者:释放信号量 SE1. */
00122         semaphore_post(&app_sem1);
00123
00124         printk("TA1: produced.\n");
00125
00126         task_sleep(RS_TICK_FREQ);
00127     }
00128 }
00129
```



```
00130 /* 任务 TA2 入口 */
00131 void app_task_2(arg_t)
00132 {
00133     for (;;)
00134     {
00135         /* 消费者:获取信号量 SE1. */
00136         semaphore_wait(&app_sem1, RS_WAIT_FOREVER);
00137
00138         printf("TA2: consumed.\n");
00139     }
00140 }
.....
```

7.14 信号量内部结构

信号量结构为 `sem_t` 类型，定义一个 `sem_t` 类型的信号量，编译器就会分配一个信号量控制块（SCB）。信号量控制块用于保存信号量运行时的状态信息，表 7.10 列出了 SCB 结构说明：

表 7.10 信号量控制块 SCB 结构说明

项	描 述
name	信号量的名字
count	信号量的值
init	初始化标志
waits	信号的等待队列

大多数情况下，开发者不需要了解 SCB 的内容。但某些时候，特别是在调试时，观察 SCB 项的值就很有用。尽管 RS-RTOS 允许观测 SCB 各项的值，但绝对不允许修改它。

7.15 总 结

信号量和互斥量都可以被用来提供互斥访问。不过，互斥量只能用来提供互斥，而信号量灵活得多，还可以提供事件通知以及任务同步。

互斥量使用严格，因此也更加可靠。如果互斥对程序很关键，那么就应该使用互斥量；如果互斥不是程序的重要因素，那么就使用信号量，因为它比互斥量稍快，并且消耗更少的资源。

二值信号量是信号量的特殊形式，其值只能取 0 或者 1 (**false** 或者 **true**)。如果使用信号量来实现互斥，使用二值信号量是最好的选择。

信号量的值代表了资源的可用实例数，获取信号量使得信号量的值减少，释放信号量使得信号量的值增加。与互斥量不同，获取和释放信号量的顺序没有严格规定，完全取决于应用的实际意义。

信号量没有所有权概念，使用信号量不会自动提升任务的优先级，不能解决优先级翻转问题。此外，信号量还可以在中断服务，定时器应用中使用，但是不能使用可以引起阻塞的接口。

第 8 章

二值信号量

8.1 介绍

二值信号量(Binary Semaphore)是信号量的特殊形式。二值信号量只能取 0 或者 1(false 或者 true) 两个值。大多数情况下, 初值被设为 1 (true), 这意味着同一时刻, 只能有一个程序可以访问二值信号量, 这种特性特别是被用来代替互斥量用来提供互斥访问。

使用二值信号量实现互斥访问时, 使用者必须避免已经获取二值信号量的任务再次调用获取信号量的服务。第二次调用获取信号量的服务会失败, 如果没有设置超时时间, 该任务将被永久挂起, 同时令资源永远不可用。

二值信号量还常常被用于任务间的同步。信号量不像互斥量那样, 有严格的获取、释放顺序。用于同步的二值信号量常常被初始化为 0, 任务 A 执行获取二值信号量的服务时被阻塞(此时二值信号量值为 0), 直到任务 B 释放该二值信号量有效信号(1), 任务 A 将继续运行, 本章将对这个例子进行讨论。

二值信号量的特性与普通信号量一致, 唯一的区别是, 二值信号量只能取 0 或者 1(false 或者 true) 两种状态, 因此, 二值信号量比普通信号量消耗更少的资源。

8.2 二值信号量属性

当定义一个二值信号量时，就创建了一个二值信号量控制块（SBCB）。SBCB 用于存储信号量运行时的状态信息，包括信号量的值、任务阻塞队列等。SBCB 可以被放在内存中任何位置，可以通过申请动态内存获得，但通常把它定义为全局变量，以便不同的任务和程序能够访问。

二值信号量的属性可以在其控制块中找到。表 8.1 列出了二值信号量的主要属性：

表 8.1 二值信号量控制块 SBCB 主要属性

项	描 述
name	二值信号量的名字
value	二值信号量的值（取值 0 或者 1）

8.3 二值信号量服务综述

附录 I 包含了与二值信号量相关的各项服务接口的详细信息。其中包括接口原型、对调用的描述、参数、返回值、注释和警告、允许在何处调用、被抢占的可能性等。

表 8.2 列出系统提供的与信号量相关的服务接口列表。在本章后面的部分，将逐个介绍这些服务。

表 8.2 二值信号量服务接口

服 务	描 述
sembinary_create	创建二值信号量
sembinary_delete	删除二值信号量
sembinary_wait	获取（等待）二值信号量
sembinary_trywait	获取（无等待）二值信号量
sembinary_post	释放二值信号量
sembinary_info	获取二值信号量信息

8.4 创建二值信号量

每个二值信号量必须声明为 `semb_t` 数据类型。当定义一个信号量时，系统就会创建一个二值信号量结构（SBCB）。在使用一个信号量前，必须创建该信号量并对其进行初始化，创建二值信号量服务的接口原型如下：

```
status_t  sembinary_create(semb_t __p_* semb, name_t name, bool value);
```

创建一个二值信号量需要给信号量指定一个名字，和一个初始值。表 8.3 给出了该接口参数说明：

表 8.3 接口 `sembinary_create` 参数说明

参 数	说 明
<code>semb</code>	指向二值信号量的指针
<code>name</code>	二值信号量的名字
<code>value</code>	二值信号量初始值

例子 APP_SAMPLE_08A 演示了如何创建一个二值信号量：

【例 APP_SAMPLE_08A】

file: example\WIN32\app_sample_08a.cpp

```
.....
00048 /* 二值信号量 */
00049 semb_t app_semb1;
.....
00062     status_t status;
.....
00067     /* 创建二值信号量 SB1,初始值为 true */
00068     status = sembinary_create(
00069         &app_semb1,
```

```
00070      build_name('S', 'B', '1', '\0'),
00071      true);
.....
```

创建二值信号量时，必须指定一个合法的信号量结构指针，并且不允许传入一个空指针（NULL），否则将引发一个断言错误。信号量的名字通过 `build_name` 建立，也可以指定为 0，表示不使用内核对象名字。信号量的初始值意义完全由应用定义，二值信号量初始值只能为 0 或者 1（`false` 或者 `true`）。

8.5 删除二值信号量

当一个信号量不再被使用，通过删除信号量，释放不必要的资源开销。删除一个二值信号量服务的接口原型如下：

```
status_t sembinary_delete(sem_t __p_* semb);
```

表 8.4 给出了该接口参数说明：

表 8.4 接口 `sembinary_delete` 参数说明

参 数	说 明
semb	指向待删除的二值信号量的指针

注意：当删除一个二值信号量时，如果有任务正在等待该信号量资源，则所有等待该信号量的任务将被唤醒，并获得一个成功信息。要避免删除一个正在使用的信号量，以防可能由此产生的逻辑错误。

8.6 获取（等待）二值信号量

与普通信号量类似，通过调用 `sembinary_wait` 取得二值信号量的实例。所不同的是，普通信号量可用有多个实例，而二值信号量只有一个实例。

获取（等待）二值信号量服务的接口原型如下：

```
status_t  sembinary_wait(sem_t __p_* semb, tick_t ticks);
```

如果二值信号量值为 1（true），成功获取信号量，则信号量值变为 0（false）；如果信号量值为 0（false），调用获取二值信号量服务，则调用任务将进入阻塞状态，直到信号量有效，或者指定的超时返回。表 8.5 给出了该接口参数说明：

表 8.5 接口 sembinary_wait 参数说明

参 数	说 明
semb	指向二值信号量的指针
ticks	指定的超时时间，单位为系统节拍（Tick），如指定 RS_WAIT_FOREVER 表示无限期等待，直到信号量有可用实例。

注意，表中 RS_WAIT_FOREVER 是内核定义的宏，其值为 0，也就是说，当超时 ticks 参数设置为 0 时，如果二值信号量的值为 0，调用该服务的任务将一直等待，直到所需信号量有实例可用。

例子 APP_SAMPLE_08B 演示了获取二值信号量实例的使用方法：

【例 APP_SAMPLE_08B】

file: example\WIN32\app_sample_08b.cpp

```
.....
00061 /* 二值信号量 */
00062 sem_t app_semb1;
.....
00094 /* 任务 TA1 入口 */
00095 void app_task_1(arg_t)
00096 {
00097     status_t status;
00098
00099     /* 创建信号量 SB1,初始值为 true */
00100     status = sembinary_create(
```

```
00101      &app_semb1,
00102      build_name('S', 'B', '1', '\0'),
00103      true);
.....
00108      /* 获取信号量 SB1 实例,
00109      * 如信号量 SB1 无可实例,任务将进入等待状态. */
00110      status = sembinary_wait(&app_semb1, RS_WAIT_FOREVER);
.....
00123 }
.....
```

8.7 获取（无等待）二值信号量

如果希望在获取信号量时，不会由于信号量无效而进入阻塞状态，需要使用无等待获取二值信号量服务。无等待获取二值信号量服务的接口原型为：

```
status_t sembinary_trywait(sem_t __p *semb);
```

表 8.6 给出了接口参数说明：

表 8.6 接口 **sembinary_trywait** 参数说明

参 数	说 明
semb	指向二值信号量的指针

使用无等待获取二值信号量，调用者不会进入等待状态，接口会马上返回，而不管该二值信号量是否有效。当二值信号量有效时（1 或者 true），该服务将使信号量的值变为无效（0 或者 false），并且返回成功；当二值信号量无效时（0 或者 false），调用该接口将返回错误代码。

例子 APP_SAMPLE_08C 演示了无等待获取二值互斥量服务的使用方法：

【例 APP_SAMPLE_08C】

file: example\WIN32\app_sample_08c.cpp

```
.....
00061 /* 二值信号量 */
00062 sem_t app_semb1;
.....
00094 /* 任务 TA1 入口 */
00095 void app_task_1(arg_t)
00096 {
00097     status_t status;
00098
00099     /* 创建二值信号量 SB1,初始值为 false */
00100     status = sembinary_create(
00101         &app_semb1,
00102         build_name('S', 'B', '1', '\0'),
00103         false);
.....
00108     /* 无等待获取二值信号量 SB1 实例,
00109      * 如二值信号量 SB1 无可实例,服务接口将返回错误代码. */
00110     status = sembinary_trywait(&app_semb1);
.....
00128 }
.....
```

与普通信号量一样，`sembinary_trywait` 也能被用于中断服务例程（ISR）、定时器应用。该接口常常被用来提供任务、中断服务例程、定时器之间的互斥访问。

8.8 释放二值信号量

释放二值信号量将使信号量的值变为 1（true），如果有任务因等待该信号量而阻塞，则其中最高优先级的任务将被唤醒。释放二值信号量服务的接口原型如下：

status_t sembinary_post(sem_t __p_* semb);

表 8.7 给出了参数说明：

表 8.7 接口 sembinary_post 参数说明

参 数	说 明
semb	指向待释放二值信号量的指针

例子 APP_SAMPLE_08D 演示释放信号量实例的方法：

【例 APP_SAMPLE_08D】

file: example\WIN32\app_sample_08d.cpp

```
.....
00061 /* 二值信号量 */
00062 sem_t app_semb1;
.....
00097     status_t status;
00098
00099     /* 创建信号量 SB1,初始值为 false */
.....
00108     /* 释放二值信号量 SB1 实例,
00109      * 成功调用该服务将使信号量 SB1 有效. */
00110     status = sembinary_post(&app_semb1);
.....
```

二值信号量通常被用来实现资源的互斥访问。任务（或者是中断服务例程、定时器应用）必须首先获得信号量，完成临界资源访问后，然后释放信号量。这个顺序是严格的，用法与互斥量类似。但是，使用二值信号量进行任务间的同步或者事件通知时，获取信号量和释放信号量并不需要按照这个顺序进行，而完全由应用的性质决定。因而，信号量比互斥量更为灵活。

8.9 获取二值信号量信息

通过调用 `sembinary_info` 可以获取二值信号量运行时的状态信息。得到的信息包括信号量名字，信号量的值。获取二值信号量信息的接口原型为：

```
status_t sembinary_info(sem_t __p_ * semb, seminfo_t __out_ * info);
```

该接口传入一个 `seminfo_t` 结构指针参数，用以存放服务接口返回的各信息值，表 8.8 给出 `sembinary_info` 服务接口参数详细说明：

表 8.8 接口 `semaphore_info` 参数说明

参 数		说 明
semb		指向二值信号量的指针
info	name	返回二值信号量的名字
	value	返回二值信号量的值（0 或者 1）

例子 APP_SAMPLE_08E 演示了利用接口 `sembinary_info` 获取信号量的信息：

【例 APP_SAMPLE_08E】

file: example\WIN32\app_sample_08e.cpp

```
.....
00061 /* 二值信号量 */
00062 sem_t app_semb1;
.....
00097     status_t status;
00098     seminfo_t info;
00099
00100     /* 创建二值信号量 SB1,初始值为 true */
.....
00109     /* 获取信号量 SB1 信息. */
00110     status = sembinary_info(&app_semb1, &info);
00111
```

```

00112     if (status == RS_EOK) {
00113
00114         /* 显示信号量信息 */
00115         printk("==Binary Semaphore Infomation=====\n");
00116         printk("Semaphore Name   : %s\n", info.name);
00117         printk("Semaphore Value : %s\n", info.value ? "true" : "false" );
00118     }
.....

```

8.10 使用二值信号量代替互斥量

在第 6 章对互斥量的介绍中，演示过这样一个例子：有两个任务 **Waveform Draw** 和 **Message Display**，分别负责显示波形和文本信息，它们共享资源是显示器 **Display**，在例子中使用互斥量保护共享资源。现在我们将这个例子作了改动，使用二值信号量来代替互斥量，而没有改动其他的结构。从中对比二值信号量和互斥量的相似之处和它们的区别。下面是改动之后的示例图解。

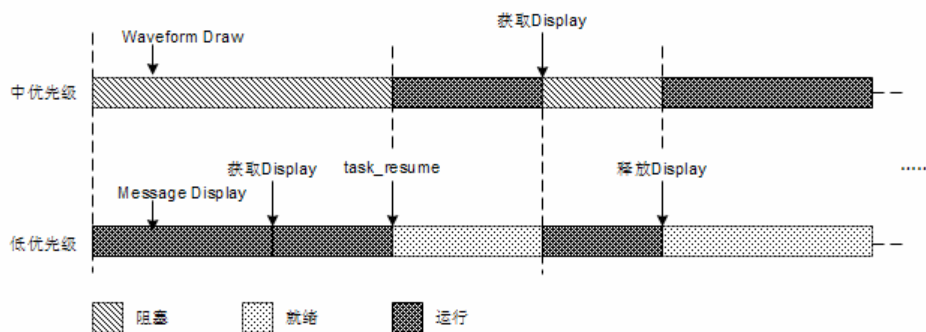


图8.1 二值信号量代替互斥量示例

任务 **Waveform Draw** 被唤醒后，马上就抢占任务 **Message Display** 进入运行状态。对比图 6.5 可以看出，使用互斥量时，任务 **Message Display** 对 **display** 资源的使用没有被 **Waveform Draw** 中断（虽然它是在 **Message Display** 使用 **display** 过程中被唤醒的）；使用信

号量时，任务 Message Display 在访问 display 资源时，被任务 Waveform Draw 抢占，任务 Waveform Draw 继续运行，在需要使用共享资源 display 时，它试图申请信号量，此时资源被任务 Message Display 占用，任务 Waveform Draw 再次进入阻塞状态，而任务 Message Display 继续执行，释放资源 Display，任务 Waveform Draw 又重新进入运行态。

虽然两个例子的代码结构一样，使用信号量实现了互斥量相同的功能，但两个任务的运行方式却有很大的差别。改动之后的例子代码如下：

【例 APP_SAMPLE_08F】

file: example\WIN32\app_sample_08f.cpp

```
.....
00038 #include "inc/kapi.h"
00039 #include "app_sample.h"
00040
00041 #ifdef APP_SAMPLE_08F
00042
00043 /**
00044  * 例子 APP_SAMPLE_08F 演示如何使用信号量保护共享资源
00045  */
00046
00047
00048 /* 任务栈大小 */
00049 #define APP_STACK_SIZE      1024 * 32
00050
00051
00052 enum {
00053     /* 任务优先级 */
00054     APP_PRIO_WAVEFORM      = 11,
00055     APP_PRIO_MESSAGE       = 12,
00056
00057     /* 互斥量优先级 */
00058     APP_PRIO_DISPLAY       = 10,
00059 };
```

```
00060
00061
00062 /* 任务栈 */
00063 stack_t app_stack_message[APP_STACK_SIZE];
00064 stack_t app_stack_waveform[APP_STACK_SIZE];
00065
00066 /* 信号量 */
00067 sem_t app_semb_display;
00068
00069
00070 void app_task_message_display(arg_t arg);
00071 void app_task_waveform_draw(arg_t arg);
00072
00073
00074 /* 硬件初始化 */
00075 void hardware_initialize(void)
00076 {
00077
00078 }
00079
00080
00081 /* 应用初始化 */
00082 void application_initialize(void)
00083 {
00084     status_t status;
00085
00086     /* 打印关于示例一些信息 */
00087     APP_SAMPLE_INFO;
00088
00089
00090     /* 初始化二值信号量 Display */
```

```
00091     status = sembinary_create(
00092         &app_semb_display,
00093         build_name('D', 'I', 'S', '\0'),
00094         true);
00095
00096     ASSERT(status == RS_EOK);
00097
00098
00099     /* 创建任务 Message Display */
00100     status = task_create(
00101         APP_PRIO_MESSAGE,
00102         build_name('M', 'S', 'G', '\0'),
00103         app_task_message_display, 0,
00104         app_stack_message, APP_STACK_SIZE,
00105         0);
00106
00107     ASSERT(status == RS_EOK);
00108
00109
00110     /* 创建任务 Waveform Draw */
00111     status = task_create(
00112         APP_PRIO_WAVEFORM,
00113         build_name('W', 'A', 'V', '\0'),
00114         app_task_waveform_draw, 0,
00115         app_stack_waveform, APP_STACK_SIZE,
00116         RS_OPT_SUSPEND);
00117
00118     ASSERT(status == RS_EOK);
00119 }
00120
00121 /* 任务 Message Display 入口 */
```

```
00122 void app_task_message_display(arg_t)
00123 {
00124     /* 显示信息 step.1 */
00125     printk("MESSAGE: step 1.\n");
00126
00127     /* 获取 Display 资源 */
00128     sembinary_wait(&app_semb_display, RS_WAIT_FOREVER);
00129
00130     /**
00131      * 获取 Display 资源,显示 DisplayMsg()
00132      */
00133
00134     /* 显示信息 step.2 */
00135     printk("MESSAGE: step 2.\n");
00136
00137     /* 唤醒任务 Waveform Draw */
00138     task_resume(APP_PRIO_WAVEFORM);
00139
00140     /* 显示信息 step.3 */
00141     printk("MESSAGE: step 3.\n");
00142
00143
00144     /* 释放 Display 资源 */
00145     sembinary_post(&app_semb_display);
00146
00147     /* 显示信息 step.4 */
00148     printk("MESSAGE: step 4.\n");
00149
00150     for (;;)
00151     {
00152         task_sleep(RS_TICK_FREQ);
```



```
00153     }
00154 }
00155
00156 /* 任务 Waveform Draw 入口 */
00157 void app_task_waveform_draw(arg_t)
00158 {
00159     /* 显示信息 step.1 */
00160     printk("WAVEFORM: step 1.\n");
00161
00162     /* 获取 Display 资源 */
00163     sembinary_wait(&app_semb_display, RS_WAIT_FOREVER);
00164
00165     /**
00166      * 获取 Display 资源,显示 DisplayWave()
00167      */
00168
00169     /* 显示信息 step.2 */
00170     printk("WAVEFORM: step 2.\n");
00171
00172
00173     /* 释放 Display 资源 */
00174     sembinary_post(&app_semb_display);
00175
00176     /* 显示信息 step.3 */
00177     printk("WAVEFORM: step 3.\n");
00178
00179     for (;;)
00180     {
00181         task_sleep(RS_TICK_FREQ);
00182     }
00183 }
```

```
00184
00185
00186 #endif
.....
```

8.11 二值信号量内部结构

二值信号量使用 `semb_t` 类型，定义一个 `semb_t` 类型的信号量，编译器就会分配一个二值信号量控制块（SBCB）。二值信号量控制块用于保存信号量运行时的状态信息，表 8.9 列出了 SBCB 结构说明：

表 8.9 二值信号量控制块 SBCB 结构说明

项	描 述
name	二值信号量的名字
value	二值信号量的值
init	初始化标志
waits	二值信号的等待队列

这里给出 SBCB 各项的意义，仅是为了在调试时方便检查二值信号量的状态。但是不要在程序中直接更改控制块的值。

8.12 总 结

二值信号量是信号量的特殊形式，其值只能取 0 或者 1（false 或者 true）。如果使用信号量来实现互斥，使用二值信号量是最好的选择。

使用二值信号量实现互斥访问时，必须避免已经获取二值信号量的任务再次调用获取信号量的服务，将会引起任务死锁，如果没有设置超时时间，死锁将不可恢复。

二值信号量还可以用于任务间的同步，用于同步的二值信号量常常被初始化为 0。

二值信号量的特性与普通信号量一致，唯一的区别是，二值信号量只能取 0 或者 1 (**false** 或者 **true**) 两种状态，此外，二值信号量比普通信号量消耗更少的资源。

第9章

邮 箱

9.1 介绍

邮箱（Mail Box）是任务间常用的、轻量级的通讯方式。它提供了由一个任务（或者中断服务、定时器应用）向另一个任务发送信息的手段。

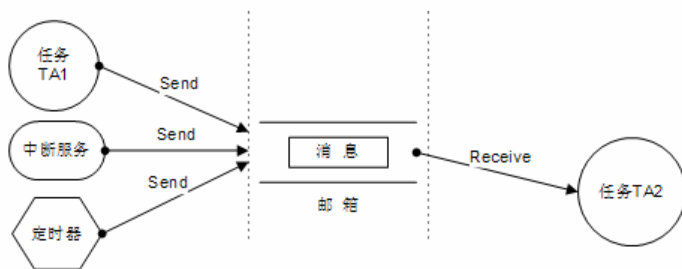


图9.1 RS-RTOS 邮箱

邮箱是一个公共组件，一个邮箱能够容纳一条信息，发送者将信息发送到一个空的邮箱中，接收者从一个非空的邮箱取走信息。如果邮箱已满（包含一个消息），往该邮箱发送消息将返回一个错误；如果邮箱为空，试图取得该邮箱消息的任务将被阻塞或者获得失败代码（取决于使用的接口和选项）。

除了任务，中断服务例程，定时器应用也能够向邮箱的发送信息，但只有任务能够接收

邮箱消息。

邮箱消息的格式和意义完全由应用开发者定义，消息可以是一个数字、字符串、缓冲区、二进制数据等。

9.2 邮箱消息

一个邮箱能够容纳一条信息，消息是任务间的通讯内容的载体，也是邮箱存储的单元对象。在 RS-RTOS 系统中，邮箱消息定义类型为 `mail_t`，它是一个宽为 32 位的无符合整型数据，如图 9.2 所示。



图9.2 RS-RTOS 邮箱消息

一个邮箱消息可以存放 4 字节大小的数据，这刚好等于一个指针所占的空间大小（在 8 位或者 16 位的系统，指针也可能是 2 字节，甚至 1 个字节）。邮箱消息大小取为和一个指针大小相等，理由是，当实际需要发送的信息比较大时，可以通过传递一个缓冲的地址实现。

虽然邮箱消息被定义为 32 位的整型数据，但是，能够存放的数据格式是不受限制的，可以是一个字符，一个有符号的整数，或者一个指针数据，通过强制的类型转换成特定的格式即可。

9.3 邮箱属性

邮箱主要属性保存在邮箱控制块（Mailbox Control Block，MBCB）中，MBCB 是用来保存运行时（Run-time）邮箱状态信息的数据结构。表 9.1 描述了构成 MBCB 的主要项。

MBCB 可以存在于内存的任意位置，通常被定义为全局变量，以便它能够被多个任务访问。

表 9.1 邮箱控制块 MBCB 主要属性

项	描 述
---	-----

name	邮箱的名字
mail	邮箱的消息
avail	消息是否有效

9.4 邮箱服务综述

附录 J 包含了与邮箱相关的各项服务接口的详细信息。包括函数的原型、简要介绍、参数、返回值、调用者、阻塞状态、注意事项。

表 9.2 包括系统提供的邮箱服务接口的列表。在本章后面的部分，将继续对这些服务进行详细的研究。

表 9.2 邮箱服务接口

服 务	描 述
mailbox_create	创建邮箱
mailbox_delete	删除邮箱
mailbox_send	发送邮箱消息
mailbox_receive	接收邮箱消息
mailbox_flush	清空邮箱

9.5 创建邮箱

在使用一个邮箱前，首先需要创建该邮箱，并对其初始化。邮箱必须声明为 `mbox_t` 类型，当定义一个 `mbox_t` 类型的变量时，将获得一个邮箱控制块。邮箱控制块必须被初始化后才能使用。通过接口：

```
status_t mailbox_create(mbox_t __p_* mbox, name_t name);
```

完成邮箱的创建和初始化工作。该接口传入两个参数用以指定邮箱的属性，表 9.3 说明

了创建邮箱接口的传入参数。

表 9.3 接口 `mailbox_create` 参数说明

参 数	描 述
<code>mbox</code>	指向邮箱的指针
<code>name</code>	邮箱的名字

例子 APP_SAMPLE_09A 演示了如何创建一个邮箱：

【例 APP_SAMPLE_09A】

file: example\WIN32\app_sample_09a.cpp

```
.....
00048 /* 邮箱 */
00049 mbox_t app_mbox1;
.....
00062     status_t status;
.....
00067     /* 创建邮箱 MB1 */
00068     status = mailbox_create(
00069         &app_mbox1,
00070         build_name('M', 'B', '1', '\0'));
.....
```

创建邮箱时，必须指定一个合法的邮箱结构指针，不允许传入一个空指针（NULL），否则将引发一个断言错误。邮箱的名字通过 `build_name` 建立，如果指定为 0，表示创建一个未命名的邮箱。邮箱一旦创建成功，即可获得一个空的邮箱，表示邮箱内没有任何消息，并可以随时接收发送者投递过来的消息。

9.6 删除邮箱

删除邮箱可以节约一些系统资源，前提是该邮箱已经不再使用。删除邮箱服务的接口原

型如下：

```
status_t mailbox_delete(mbox_t __p_* mbox);
```

删除一个邮箱，只需要给出指向该邮箱的指针为参数，且该参数不允许为空指针（NULL）
表 9.4 是删除邮箱的接口参数说明。

表 9.4 接口 mailbox_delete 参数说明

参 数	描 述
mbox	指向邮箱的指针

例子 APP_SAMPLE_09B 演示了如何删除一个不用的邮箱：

【例 APP_SAMPLE_09B】

file: example\WIN32\app_sample_09b.cpp

```
.....
00048 /* 邮箱 */
00049 mbox_t app_mbox1;
.....
00062     status_t status;
.....
00067     /* 创建邮箱 MB1 */
.....
00074     /* 删除邮箱 MB1 */
00075     status = mailbox_delete(&app_mbox1);
.....
```

删除一个邮箱之前，必须确保该邮箱不再被使用，并且没有任务等待该邮箱的消息，否则将返回失败代码。如果邮箱包含了有效消息，邮箱消息将被删除。但是，如果邮箱消息内容是指向某个缓冲区的指针时，情况变得稍微复杂，删除的是指针本身，那么，指向的缓冲区被正确处理了吗？

9.7 发送邮箱消息

使用邮箱作为中介，可以将信息从一个任传递到另一个任务。信息传递第一步是将信息发送到邮箱，通过接口：

```
status_t mailbox_send(mbox_t __p_* mbox, mail_t mail);
```

将信息发送到指定的邮箱。发送邮箱消息服务的接口参数在表 9.5 中给出：

表 9.5 接口 mailbox_send 参数说明

参 数	描 述
mbox	指向邮箱的指针
mail	邮箱消息

例子 APP_SAMPLE_09C 演示了向指定的邮箱发送信息：

【例 APP_SAMPLE_09C】

file: example\WIN32\app_sample_09c.cpp

```
.....
00063 /* 邮箱 */
00064 mbox_t app_mbox1;
.....
00142     int i = 0;
.....
00153         /* 发送邮箱消息 */
00154         mailbox_send(&app_mbox1, (mail_t)i++);
.....
```

每个邮箱只能存放一条信息。当邮箱已满，表示该邮箱包含了有效信息，这时，往邮箱发送信息将返回失败。只有当邮箱为空时，才能成功发送一条信息到该邮箱。邮箱消息的传递方式是值复制，发送的消息会被复制到邮箱的消息缓冲区中。

邮箱消息是 32 位的整型数据，能够存放最大 4 个字节的数据。当消息大于 4 个字节时，

则将需要传递的数据放到一个缓冲区中，通过发送缓冲区的地址实现间接传递。

9.8 接收邮箱消息

如果邮箱包含有效消息，任务即可通过接收邮箱消息服务取得该消息。当一条邮箱消息被成功取走，邮箱就变为空，等待存放下一个消息。接收邮箱消息服务的接口原型如下：

```
status_t mailbox_receive(mbox_t __p_* mbox, mail_t __out_* mail, tick_t ticks);
```

参数 **mail** 是一个传出参数，它是一个指针，指向存放返回消息的地址。表 9.6 中给出接收邮箱消息服务的接口参数说明：

表 9.6 接口 **mailbox_receive** 参数说明

参 数	描 述
mbox	指向邮箱的指针
mail	指向存放邮箱消息的地址
ticks	指定的超时时间，单位为系统节拍（Tick），如指定 RS_WAIT_FOREVER 表示无限期等待，直到邮箱获得有效消息。

例子 APP_SAMPLE_09C 演示了接收邮箱消息服务的使用：

【例 APP_SAMPLE_09C】

file: example\WIN32\app_sample_09c.cpp

```
.....
00063 /* 邮箱 */
00064 mbox_t app_mbox1;
.....
00118     status_t status;
00119     mail_t mail;
.....
00127     /**
```

```
00128      * 接收邮箱消息,
00129      * 如果邮箱为空,任务进入阻塞,等待消息 */
00130      status = mailbox_receive(
00131          &app_mbox1,
00132          &mail,
00133          RS_WAIT_FOREVER);
.....
```

当邮箱保存有效消息时，接收邮箱消息服务马上返回，并取走一个邮箱消息。当邮箱为空时，接收邮箱消息的任务将进入阻塞状态，直到邮箱收到一个有效消息，或者设定的超时时间到达为止。

注意，内核预定义宏 RS_WAIT_FOREVER 的值为 0，当参数 ticks 指定为 0 时，表示无限期等待，直到邮箱获得有效消息。

9.9 清空邮箱

清空邮箱服务是把邮箱信息清空。无论邮箱是否包含了消息，清空邮箱操作都会使得邮箱恢复到原始空闲的状态。清空邮箱服务的接口原型如下：

```
status_t mailbox_flush(mbox_t __p_ * mbox);
```

表 9.7 中给出清空邮箱服务的接口参数说明：

表 9.7 接口 mailbox_flush 参数说明

参 数	描 述
mbox	指向待清空邮箱的指针

例子 APP_SAMPLE_09D 演示了清空一个指定的邮箱：

【例 APP_SAMPLE_09D】

file: example\WIN32\app_sample_09d.cpp

```
.....
```

```

00048 /* 邮箱 */
00049 mbox_t app_mbox1;
.....
00062     status_t status;
.....
00074     /* 清除邮箱 MB1 */
00075     status = mailbox_flush(&app_mbox1);
.....

```

清空邮箱时，如果邮箱包含了有效消息，邮箱消息将被删除。在某些情况下需要小心，假设邮箱消息内容是指向某个缓冲区的指针，要谨慎地考虑这个问题：该缓冲区是否能被正确释放？

9.10 间接邮箱消息

使用邮箱传递消息，理想的情况是，消息的内容能够直接存放在邮箱消息结构 `mail_t` 中。这种情况下，由于不需要借助额外的缓冲区，邮箱消息的处理非常简单。当消息的大小超过 `mail_t` 能够容纳的大小时，就需要把消息数据放到缓冲区中，而将缓冲区的地址作为邮箱消息发送，这种邮箱消息称为间接邮箱消息，对应的，前面一种称为直接邮箱消息。图 9.3 给出这两种邮箱消息的图示。

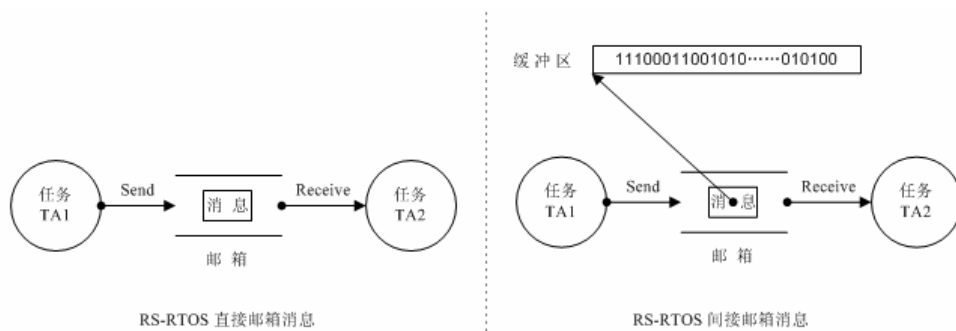


图9.3 直接邮箱消息和间接邮箱消息

使用间接邮箱消息时，仅仅是传递缓冲区的地址，而没有复制缓冲区内容的开销，极大的提高了消息传递的效率。不过，使用缓冲区也带来一些问题，如缓冲区内存的管理，潜在的缓冲区存储冲突，幸运的是，这些问题都很容易避免。

示例 APP_SAMPLE_09E 演示了如通过邮箱传递缓冲区消息，为了实现这个目的，我们使用了间接邮箱消息模式。

应用初始化时，创建了两个任务和一个邮箱，分别是任务 TA1、任务 TA2 和邮箱 MB1。任务 TA1 负责发送邮箱消息，任务 TA2 接收邮箱消息。任务 TA1 发送给 TA2 的消息是一个字符串信息，通过一个缓冲区来存放。该缓冲区使用动态内存，由任务 TA1 负责申请，由任务 TA2 负责释放。在使用缓冲区传递消息时，明确各自任务对缓冲区内存管理的职责很重要。通常做法是，消息发起者负责申请缓冲区，消息的处理者负责释放缓冲区，这个思路非常简单清晰。

任务 TA1 以一个固定的周期（100 毫秒）向邮箱 MB1 发送消息。任务 TA1 实际发送的邮箱消息是一个指针，该指针指向真正缓冲区消息。在发送邮箱消息时，需要将指针做强制类型转换，转换为 mail_t 结构。因为指针占用的字节刚好等于 mail_t 的结构的大小，这个转换并不会丢失数据。

任务 TA2 接收到邮箱消息后，需要将 mail_t 转换成指针类型，这样就可以访问缓冲区，取得真正需要消息。例子中，任务 TA2 在收到缓冲区消息后，所做的处理就是在控制台上输出消息的内容，然后释放该缓冲区内存。

节 9.11 给出了该示例的代码清单。

9.11 示例 APP_SAMPLE_09E 代码

【例 APP_SAMPLE_09E】

file: example\WIN32\app_sample_09e.cpp

```
.....
00038 #include <stdio.h>
00039 #include "inc/kapi.h"
00040 #include "app_sample.h"
00041
00042 #ifdef APP_SAMPLE_09E
```

```
00043
00044 /**
00045  * 例子 APP_SAMPLE_09E 演示如何使用邮箱传递缓冲区消息
00046  */
00047
00048
00049 /* 任务栈大小 */
00050 #define APP_STACK_SIZE      1024 * 32
00051
00052
00053 enum {
00054     /* 任务优先级 */
00055     APP_PRIO_TA1      = 10,
00056     APP_PRIO_TA2      = 11,
00057 };
00058
00059
00060 /* 任务栈 */
00061 stack_t app_stack_ta1[APP_STACK_SIZE];
00062 stack_t app_stack_ta2[APP_STACK_SIZE];
00063
00064 /* 邮箱 */
00065 mbox_t app_mbox1;
00066
00067
00068 void app_task_1(arg_t arg);
00069 void app_task_2(arg_t arg);
00070
00071
00072 /* 硬件初始化 */
00073 void hardware_initialize(void)
```

```
00074 {
00075
00076 }
00077
00078
00079 /* 应用初始化 */
00080 void application_initialize(void)
00081 {
00082     status_t status;
00083
00084     /* 打印关于示例一些信息 */
00085     APP_SAMPLE_INFO;
00086
00087     /* 创建任务 TA1 */
00088     status = task_create(
00089         APP_PRIO_TA1,
00090         build_name('T', 'A', '1', '\0'),
00091         app_task_1, 0,
00092         app_stack_ta1, APP_STACK_SIZE,
00093         0);
00094
00095     ASSERT(status == RS_EOK);
00096
00097     /* 创建任务 TA2 */
00098     status = task_create(
00099         APP_PRIO_TA2,
00100         build_name('T', 'A', '2', '\0'),
00101         app_task_2, 0,
00102         app_stack_ta2, APP_STACK_SIZE,
00103         0);
00104
```



```
00105     ASSERT(status == RS_EOK);
00106
00107     /* 创建邮箱 MB1 */
00108     status = mailbox_create(
00109         &app_mbox1,
00110         build_name('M', 'B', '1', '\0'));
00111
00112     ASSERT(status == RS_EOK);
00113
00114 }
00115
00116 /* 任务 TA1 入口 */
00117 void app_task_1(arg_t)
00118 {
00119     int      i = 0;
00120     char*    p;
00121     status_t status;
00122
00123     for (;;)
00124     {
00125         /**
00126          * 任务 TA1 定期向邮箱发送缓冲区消息
00127          */
00128
00129         /* 申请缓冲区,用以存放消息 */
00130         p = (char*)mpool_alloc(50);
00131
00132         /* 判定内存申请是否成功是个好习惯 */
00133         if (p != NULL) {
00134
00135             /* 缓冲区内容,使用增量 id 以区别不同的消息 */
```

```
00136         sprintf(p, "A message from TA1 (id:%d)", i++);
00137
00138         /* 发送邮箱消息,将缓冲区的地址作为邮箱消息 */
00139         status = mailbox_send(&app_mbox1, (mail_t)p);
00140
00141         /* 显示发送信息 */
00142         printk("TA1 Send: %s\n", p);
00143
00144         /**
00145          * 如果发送不成功,记得释放缓冲区
00146          * 这是容易忽略的地方! */
00147         if (status != RS_EOK)
00148             mpool_free(p);
00149     }
00150
00151     /* 休眠 1/10 秒 */
00152     task_sleep(RS_TICK_FREQ/10);
00153 }
00154 }
00155
00156 /* 任务 TA2 入口 */
00157 void app_task_2(arg_t)
00158 {
00159     mail_t  mail;
00160     char*   p;
00161     status_t status;
00162
00163     for (;;)
00164     {
00165         /**
00166          * 任务 TA2 负责接收消息,并输出消息内容
```

```
00167         */
00168
00169         /* 接收邮箱消息,
00170          * 如果邮箱为空,任务进入阻塞,等待消息 */
00171         status = mailbox_receive(
00172             &app_mbox1,
00173             &mail,
00174             RS_WAIT_FOREVER);
00175
00176         if (status == RS_EOK) {
00177
00178             /* 邮箱消息是缓冲区的地址 */
00179             p = (char*)mail;
00180
00181             /* 在控制台输出缓冲区的内容 */
00182             printk("TA2 Received: %s\n", p);
00183
00184             /* 释放缓冲区 */
00185             mpool_free(p);
00186         }
00187     }
00188 }
00189
00190
00191 #endif
.....
```

9.12 邮箱内部结构

使用 `mbox_t` 定义一个邮箱，每个邮箱包含一个邮箱控制块（MBCB），用来保存信号量运行时的状态信息。定义一个 `mbox_t` 类型的邮箱时，就创建了一个邮箱控制块（MBCB）。表 9.8 列出了 MBCB 结构说明：

表 9.8 邮箱控制块 MBCB 结构说明

项	描 述
name	邮箱的名字
mail	邮箱消息
avail	消息是否有效
init	初始化标志
waits	邮箱的等待队列

了解 MBCB 各项的意义，有助于理解邮箱的内部机制，方便程序测试。注意，邮箱控制块必须通过系统提供的服务来管理，千万不要直接更改控制块各项的值，否则，将引起系统的异常。

9.13 总 结

本章首先介绍了邮箱机制，分析了邮箱消息的结构，然后对 RS-RTOS 邮箱相关的服务逐一做了介绍，并给出了一些代码和图示，最后通过一个例子演示了使用邮箱传递缓冲区信息的方法。

邮箱是任务间常用的、轻量级的通讯方式。发送者将信息发送到邮箱中，接收者从邮箱中取走信息，信息的传递顺序是：任务—邮箱—任务。能够发送信息的程序包括：任务、中断服务例程、定时器应用；但只有任务能够接收邮箱消息。

一个邮箱能够容纳一条信息。当邮箱没有消息时，邮箱是一个空邮箱，空邮箱可以接收信息；当邮箱已经包含一个消息，则邮箱已满，这时，可以从邮箱中取走该消息。往一个满邮箱发送消息会失败；接收空邮箱中的消息会引起任务阻塞或者返回失败代码，取决于使用

的接口选项。

邮箱消息定义类型为 `mail_t`，是一个宽为 32 位的无符号整型数据。邮箱消息的格式和意义完全由应用开发者定义，消息可以是一个数字、字符串、缓冲区、二进制数据等。邮箱消息和应用数据类型之间转换要注意数据安全，防止数据丢失。

使用邮箱传递消息有两种方法：直接邮箱消息和间接邮箱消息。直接邮箱消息适合传递信息量极少的信息，当消息内容超过邮箱消息结构的大小时，需要使用间接邮箱消息传递。使用间接邮箱消息，仅仅是传递消息缓冲区的地址，因为没有复制缓冲区内容的开销，故能够获得非常好的传送速度。

第 10 章

队 列

10.1 介 绍

队列（Queue）又称为消息队列，是任务间通讯常用的、主要的方式。一个消息队列中可以含有一个或多个消息，并遵循一定的顺序（先进先出 FIFO、后进先出 LIFO）存储。

图 10.1 是一个消息队列的结构，队列中每个消息的长度是相等的，队列能够容纳的消息的数目称为队列的长度。在有些系统中，队列可以容纳尺寸不相等的消息，使用一个队列可方便的支持不同尺寸的消息。但是，队列容纳相等尺寸的消息却有明显的优势——消息索引

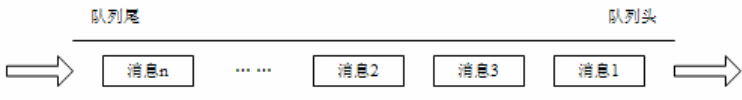


图10.1 RS-RTOS 消息队列 FIFO

的速度非常快。在 RS-RTOS 系统中，队列只容纳相等尺寸的消息，因为简练和高速才是强实时系统最关心的。

在先进先出（FIFO）的顺序中，从队列尾部插入消息，从队列头取走消息。而后进先出（LIFO）的队列中，则从队列尾插入消息，同样也从队列尾取出消息。图 10.2 是一个后进先出的消息队列。当队列中消息数目已经达到队列容纳的上限时，就不能往队列插入消息，队

列为满状态（Full）；另一个极端，当队列一个消息也没有时，就不能从队列取得消息，队列为空状态（Empty）。

消息的大小在队列创建时指定，队列创建以后不能进行修改。只要有足够的内存，消息的大小不受限制，根据消息的所包含的内容来确定消息的大小。



图10.2 RS-RTOS 消息队列 LIFO

消息进出队列的传递方式是复制。假设任务 TA1 将一个消息发送到队列 Q1 中，消息将被复制到队列 Q1 中；取出队列中的消息时，则从队列中把消息复制到指定的缓冲区中。如果消息的内容比较大，则可以通过发送指针来实现，这种消息称为间接队列消息。比如消息超过 64 字节，则可以创建一个消息大小为 4 个字节（刚好容纳一个指针）的队列，然后发送或者接收指针消息，而不是整个消息。

一个消息队列占用的存储器空间大小，取决于消息的大小和队列能容纳消息的数量。为了计算消息队列占用的存储器空间，可以将消息的大小乘以队列能容纳消息的数量。计算公式如下：

队列大小 = 消息的大小 x 队列长度

例如，如果创建一个能容纳 100 个消息的队列，每个消息大小为 16 字节，那么队列所占用的存储器空间为 1600 字节。

10.2 队列属性

创建一个队列时，同时也创建了一个队列控制块（Queue Control Block，QCB），QCB 是用来保存运行时（Run-time）队列状态信息的数据结构。队列的主要属性保存在 QCB 中，表 10.1 描述了队列的主要属性。

表 10.1 队列控制块 QCB 主要属性

项	描 述
---	-----

name	队列的名字
entries	队列的长度
size	消息尺寸
options	队列选项

消息队列的控制块可以存在于内存的任意位置，通常被定义为全局变量，以便它能够被多个任务访问。

10.3 队列服务综述

附录 K 包含了与队列相关的各项服务接口的详细信息。包括函数的原型、简要介绍、参数、返回值、调用者、阻塞状态、注意事项。

表 10.2 包括系统提供的队列服务接口的列表。在本章后面的部分，将逐一介绍这些服务，并给出一些代码示例。

表 10.2 队列服务接口

服 务	描 述
queue_create	创建队列
queue_delete	删除队列
queue_send	发送队列消息
queue_receive	接收队列消息
queue_flush	清空队列

10.4 创建队列

在使用队列前，首先需要创建该队列，并对其初始化。队列必须声明为 `queue_t` 类型，当定义一个 `queue_t` 类型的变量时，将分配一个队列控制块 **QCB**。队列控制块必须被初始化后才能使用。通过接口：

```
status_t queue_create(  
    queue_t __p_* queue,  
    name_t      name,  
    void  __p_* buff,  
    mmsz_t      size,  
    mmsz_t      entries,  
    int8u       options  
);
```

完成队列的创建和初始化工作。该接口包含六个参数，用来指定需要创建的队列的属性，表 10.3 时创建队列服务接口的参数说明。

表 10.3 接口 `queue_create` 参数说明

参 数	描 述
queue	指向队列的指针
name	队列的名字
buff	队列缓冲区的指针
size	消息的大小（字节）
entries	队列的长度
options	队列选项

消息的大小只受限于系统能够提供的存储器资源，单位是字节（Bytes）。队列缓冲区指针是指向队列存储器空间的起始地址，如果该地址为空（NULL），系统将会为队列自动分配一块合适的内存空间。队列的长度是队列能够容纳的消息的数目，如果为队列指定缓冲区，则缓冲区的大小可以通过以下公式计算：

队列缓冲区大小 = 消息的大小 x 队列的长度

例子 APP_SAMPLE_10A 演示了如何创建一个队列。该示例使用指定的队列缓冲区，注意缓冲区的定义方法，（L00067）使用了静态变量。静态变量的空间不会因为函数的退出而释放，而函数局部变量的空间在函数返回时将被释放。因为队列是一个全局对象，不能使用局部变量。除了使用静态定义队列的缓冲区，还可以将缓冲区定义为全局数组或者使用动态

内存分配。静态变量除了拥有全局变量在程序的生命期有效的特性外，还具备了封装特性，避免了程序错误的访问，另外，也使得代码看起来比较清晰。

【例 APP_SAMPLE_10A】

file: example\WIN32\app_sample_10a.cpp

```
.....
00048 /* 队列 */
00049 queue_t app_queue1;
.....
00062     status_t status;
00063
00064     /**
00065      * 队列缓冲区
00066      * 这里定义成静态变量,不能使用栈变量 */
00067     static char qbuff[1600];
.....
00072     /**
00073      * 创建一个先进先出(FIFO)队列 Q1:
00074      * 消息大小为 16 字节,队列长度能容纳 100 条消息 */
00075     status = queue_create(
00076         &app_queue1,
00077         build_name('Q', '1', '\0', '\0'),
00078         qbuff,
00079         16,
00080         100,
00081         RS_OPT_FIFO);
.....
```

创建队列时，必须指定一个合法的队列结构指针，不允许传入一个空指针（NULL），否则将引发一个断言错误。队列的名字通过 `build_name` 建立，如果指定为 0，表示创建一个未命名的队列。

如果希望系统自动为队列缓冲区分配存储器空间，将队列缓冲区指定为空指针（NULL），

系统将自动计算并分配一个合适大小的存储器空间。例子 APP_SAMPLE_10B 演示了由系统自动分配队列缓冲区的用法：

【例 APP_SAMPLE_10B】

file: example\WIN32\app_sample_10b.cpp

```
.....
00048 /* 队列 */
00049 queue_t app_queue1;
.....
00062     status_t status;
.....
00067     /**
00068      * 创建一个先进先出(FIFO)队列 Q1:
00069      * 消息大小为 16 字节,队列长度能容纳 100 条消息 */
00070     status = queue_create(
00071         &app_queue1,
00072         build_name('Q', '1', '\0', '\0'),
00073         NULL,
00074         16,
00075         100,
00076         RS_OPT_FIFO);
.....
```

实际工程中，队列常常被用于传递某些特定大小的消息，比如尺寸为 4 个字节大小的消息。对此，RS-RTOS 采取了某些优化措施，来提高消息的存取速度，这些尺寸被预定义成以下的宏：

RS_MSG_1ULONG（4 字节）

RS_MSG_2ULONG（8 字节）

RS_MSG_4ULONG（16 字节）

RS_MSG_8ULONG（32 字节）

在应用设计时，建议尽可能的采取以上尺寸的消息，有利于提高队列消息的存取速度。在例子 APP_SAMPLE_10D 中，就是采取优化的消息尺寸。

虽然创建队列服务接口看上去稍微复杂，但经过一些示例练习之后，掌握好每个参数的意义并不难。而且，创建队列服务是一个非常灵活而且精致的接口。在附录 K，也给出这个接口一些详细项目供参考。

10.5 删除队列

如果消息队列已经不再使用，那么删除该消息队列可以节约一些系统资源。删除队列服务的接口原型如下：

```
status_t queue_delete(queue_t __p_ * queue);
```

删除一个队列的接口比创建它要简单的多，只需要给出指向该队列的指针为参数，且该参数不允许为空指针（NULL）表 10.4 是删除队列的接口参数说明。

表 10.4 接口 queue_delete 参数说明

参 数	描 述
queue	指向队列的指针

例子 APP_SAMPLE_10C 演示了如何删除一个不再使用的队列：

【例 APP_SAMPLE_10C】

file: example\WIN32\app_sample_10c.cpp

```
.....
00048 /* 队列 */
00049 queue_t app_queue1;
.....
00062     status_t status;
.....
00067     /**
00068     * 创建一个先进先出(FIFO)队列 Q1:
00069     * 消息大小为 16 字节,队列长度能容纳 100 条消息 */
.....
```

```
00081      /**
00082      * 使用队列.....
00083      */
.....
00086      /* 删除队列 Q1 */
00087      status = queue_delete(&app_queue1);
.....
```

删除一个队列之前，必须确保该队列不再被使用，并且没有任务等待该队列的消息。如果有一个或者多个任务正在等待队列中的消息时，删除该队列是危险的，调用删除队列服务将获得一个失败代码，指示队列删除失败。

10.6 发送队列消息

一个刚被创建的队列，不包含任何消息，为了将消息传递到目的任务，首先需要将消息发送到队列中，完成这一工作的接口是发送队列消息服务，它的接口原型是：

```
status_t queue_send(queue_t __p_* queue, const void __p_* buff);
```

发送队列消息需要指定目的队列和待发送的消息缓冲区的首地址。在表 10.5 中给出这些参数的说明：

表 10.5 接口 queue_send 参数说明

参 数	描 述
queue	指向队列的指针
buff	指向队列消息的指针

虽然使用消息的首地址作为接口参数，但是，发送队列消息是将整个消息复制到队列的消息缓冲区中，而不是复制消息的地址。如果希望队列仅仅传递消息的地址而不是内容（在消息内容比较大时，这是非常聪明的策略），那么就创建一个消息大小为一个指针的队列，这种模式称为间接队列消息。

例子 APP_SAMPLE_10D 演示了向指定的队列发送信息的方法。

【例 APP_SAMPLE_10D】

file: example\WIN32\app_sample_10d.cpp

```
.....  
00063 /* 队列 */  
00064 queue_t app_queue1;  
.....  
00150     unsigned long i = 0;  
.....  
00158         /* 发送队列消息 */  
00159         queue_send(&app_queue1, (void*)&i);  
.....
```

当队列容纳的消息已经到达队列能容纳的上限，往队列发送的消息将失败，并返回一个失败代码。通过查看返回值，确定服务是否正确执行是个好习惯。如果经常出现队列满的情况，考虑队列长度是否设置过小？如果是，通过增大队列长度解决问题。

有些开发人员认为，增大队列的长度会使得队列的速度变慢。这是个误解，队列的速度与队列长度无关，而与消息的大小有关。但是，盲目的增加队列长度只会浪费宝贵的内存资源。

发送队列消息不会引起程序阻塞。除了任务，中断服务例程，定时器应用也能够发送队列信息。

10.7 接收队列消息

当队列包含有效消息时，任务通过接收队列消息服务获取队列中的消息。该服务成功找到队列消息后，将消息从队列缓冲区中复制到指定的目标缓冲区，并从队列中删除该消息。接收队列消息服务的接口原型如下：

```
status_t queue_receive(queue_t __p_* queue, void __out_* buff, tick_t ticks);
```

参数 **buff** 是一个传出参数，它是一个指针，指向存放返回消息的地址。表 10.6 中给出接

收队列消息服务的接口参数说明：

表 10.6 接口 `queue_receive` 参数说明

参 数	描 述
<code>queue</code>	指向队列的指针
<code>buff</code>	指向存放队列消息的地址
<code>ticks</code>	指定的超时时间，单位为系统节拍（Tick），如指定 <code>RS_WAIT_FOREVER</code> 表示无限期等待，直到获得有效消息。

例子 APP_SAMPLE_10D 演示了接收队列消息服务的使用：

【例 APP_SAMPLE_10D】

file: example\WIN32\app_sample_10d.cpp

```
.....
00063 /* 队列 */
00064 queue_t app_queue1;
.....
00123     status_t status;
00124     unsigned long message;
.....
00132         /**
00133          * 接收队列消息,
00134          * 如果队列为空,任务进入阻塞,等待消息 */
00135         status = queue_receive(
00136             &app_queue1,
00137             (void*)&message,
00138             RS_WAIT_FOREVER);
.....
```

当队列包含有效消息时，接收队列消息服务马上返回，并取走一个队列消息。当队列为空时，接收队列消息的任务将进入阻塞状态，直到队列收到一个有效消息，或者设定的超时

时间到达为止。

注意，内核预定义宏 `RS_WAIT_FOREVER` 的值为 0，当参数 `ticks` 指定为 0 时，表示无限期等待，直到获得有效消息。

接收队列消息可能会引起任务进入阻塞，不能在中断服务例程、定时器应用中使用接收队列消息的服务。

10.8 清空队列

清空队列服务将删除一个消息队列中的所有消息，并使得队列恢复到创建时的状态。如果队列是空的，这个服务将不会产生任何操作。清空队列服务的接口原型如下：

```
status_t queue_flush(queue_t __p_* queue);
```

表 10.7 中给出清空队列服务的接口参数说明：

表 10.6 接口 `queue_flush` 参数说明

参 数	描 述
queue	指向待清空队列的指针

例子 APP_SAMPLE_10E 演示了如何清空一个指定的队列：

【例 APP_SAMPLE_10E】

file: example\WIN32\app_sample_10e.cpp

```
.....
00048 /* 队列 */
00049 queue_t app_queue1;
.....
00062     status_t status;
.....
00067     /**
00068      * 创建一个先进先出(FIFO)队列 Q1:
00069      * 消息大小为 4 字节,队列长度能容纳 50 条消息 */
```

```
.....
00081      /* 清空队列 Q1 */
00082      status = queue_flush(&app_queue1);
.....
```

清空队列时，如果队列包含了有效消息，队列消息将被删除。在某些情况下需要谨慎，假设队列消息内容是指向某个缓冲区的指针，并且该缓冲区使用了动态内存，那么要确保该缓冲区能被正确释放。

10.9 间接队列消息

由于消息在进出队列时均需要进行复制过程，当消息尺寸比较小时，尤其是符合以下预定义宏尺寸的消息：

- RS_MSG_1ULONG（4 字节）
- RS_MSG_2ULONG（8 字节）
- RS_MSG_4ULONG（16 字节）
- RS_MSG_8ULONG（32 字节）

在使用小尺寸消息时，队列传输消息的效率是非常高的。当消息尺寸比较大时（习惯上取 64 字节作为分界点），更好的方法是，传递消息地址而不是消息内容本身。当队列传递的内容就是消息本身时，称为直接队列消息；当队列传递的内容是消息的缓冲区地址时，称为

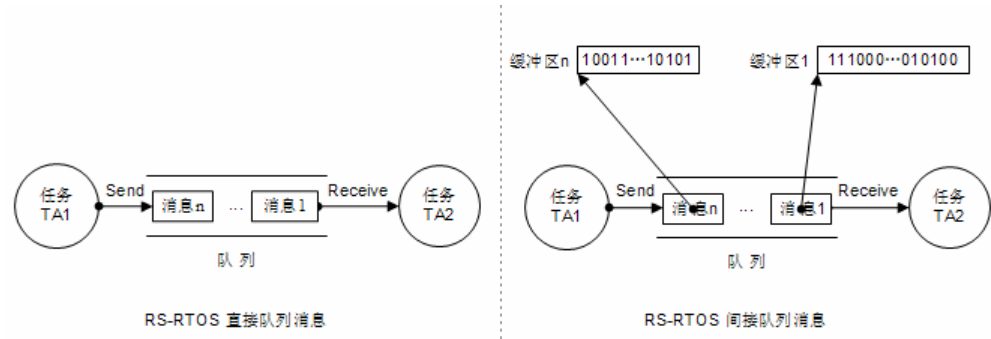


图10.3 直接队列消息和间接队列消息

间接队列消息。图 10.3 给出了两种消息的图示。

使用直接队列消息的优点是简单，不需要借助额外的缓冲区，尤其适合短小的消息。而间接队列消息合适传递大尺寸的消息，因为仅仅是传递消息缓冲区的地址，不需要复制消息的内容，消息的传递速度非常快，缺点是需要小心的管理消息缓冲区。

示例 APP_SAMPLE_10F 演示了如通过队列传递缓冲区消息例子，该例子与使用邮箱的示例 APP_SAMPLE_09E 非常相似，实际上，仅仅是使用队列代替了邮箱，这里不在赘述。

节 10.10 给出了该示例的代码清单。有一个细节的地方需要说明，行 L00144 参数使用缓冲区指针的地址，由于队列的消息内容是指针数据（指向缓冲区的指针），而接口需要的参数是消息的地址，所以这里就是缓冲区指针的地址。同样，L00177 在接收数据时需要做相应的处理。

10.10 示例 APP_SAMPLE_10F 代码

【例 APP_SAMPLE_10F】

file: example\WIN32\app_sample_10F.cpp

```
.....
00038 #include <stdio.h>
00039 #include "inc/kapi.h"
00040 #include "app_sample.h"
00041
00042 #ifdef APP_SAMPLE_10F
00043
00044 /**
00045  * 例子 APP_SAMPLE_10F 演示如何使用队列传递缓冲区消息
00046  */
00047
00048
00049 /* 任务栈大小 */
00050 #define APP_STACK_SIZE      1024 * 32
00051
```

```
00052
00053 enum {
00054     /* 任务优先级 */
00055     APP_PRIO_TA1    = 10,
00056     APP_PRIO_TA2    = 11,
00057 };
00058
00059
00060 /* 任务栈 */
00061 stack_t app_stack_ta1[APP_STACK_SIZE];
00062 stack_t app_stack_ta2[APP_STACK_SIZE];
00063
00064 /* 队列 */
00065 queue_t app_queue1;
00066
00067
00068 void app_task_1(arg_t arg);
00069 void app_task_2(arg_t arg);
00070
00071
00072 /* 硬件初始化 */
00073 void hardware_initialize(void)
00074 {
00075
00076 }
00077
00078
00079 /* 应用初始化 */
00080 void application_initialize(void)
00081 {
00082     status_t status;
```

```
00083
00084     /* 打印关于示例一些信息 */
00085     APP_SAMPLE_INFO;
00086
00087     /* 创建任务 TA1 */
00088     status = task_create(
00089         APP_PRIO_TA1,
00090         build_name('T', 'A', '1', '\0'),
00091         app_task_1, 0,
00092         app_stack_ta1, APP_STACK_SIZE,
00093         0);
00094
00095     ASSERT(status == RS_EOK);
00096
00097     /* 创建任务 TA2 */
00098     status = task_create(
00099         APP_PRIO_TA2,
00100         build_name('T', 'A', '2', '\0'),
00101         app_task_2, 0,
00102         app_stack_ta2, APP_STACK_SIZE,
00103         0);
00104
00105     ASSERT(status == RS_EOK);
00106
00107     /**
00108      * 创建一个先进先出(FIFO)队列 Q1:
00109      * 消息大小为 4 字节,队列长度能容纳 80 条消息 */
00110     status = queue_create(
00111         &app_queue1,
00112         build_name('Q', '1', '\0', '\0'),
00113         NULL,
```

```
00114         RS_MSG_1ULONG,
00115         80,
00116         RS_OPT_FIFO);
00117
00118     ASSERT(status == RS_EOK);
00119 }
00120
00121 /* 任务 TA1 入口 */
00122 void app_task_1(arg_t)
00123 {
00124     int      i = 0;
00125     char*    p;
00126     status_t status;
00127
00128     for (;;)
00129     {
00130         /**
00131          * 任务 TA1 定期向队列发送缓冲区消息
00132          */
00133
00134         /* 申请缓冲区,用来存放消息 */
00135         p = (char*)mpool_alloc(50);
00136
00137         /* 判定内存申请是否成功是个好习惯 */
00138         if (p != NULL) {
00139
00140             /* 缓冲区内容,使用增量 id 以区别不同的消息 */
00141             sprintf(p, "A message from TA1 (id:%d)", i++);
00142
00143             /* 发送队列消息,将缓冲区的地址作为消息 */
00144             status = queue_send(&app_queue1, (void*)&p);
```

```
00145
00146         /* 显示发送信息 */
00147         printk("TA1 Send: %s\n", p);
00148
00149         /**
00150          * 如果发送不成功,记得释放缓冲区
00151          * 这是容易忽略的地方! */
00152         if (status != RS_EOK)
00153             mpool_free(p);
00154     }
00155
00156     /* 休眠 1/10 秒 */
00157     task_sleep(RS_TICK_FREQ/10);
00158 }
00159 }
00160
00161 /* 任务 TA2 入口 */
00162 void app_task_2(arg_t)
00163 {
00164     char*   p;
00165     status_t status;
00166
00167     for (;;)
00168     {
00169         /**
00170          * 任务 TA2 负责接收消息,并输出消息内容
00171          */
00172
00173         /* 接收队列消息,
00174          * 如果队列为空,任务进入阻塞,等待消息 */
00175         status = queue_receive(
```

```

00176                &app_queue1,
00177                (void*)&p,
00178                RS_WAIT_FOREVER);
00179
00180        if (status == RS_EOK) {
00181
00182            /* 在控制台输出缓冲区的内容 */
00183            printk("TA2 Received: %s\n", p);
00184
00185            /* 释放缓冲区 */
00186            mpool_free(p);
00187        }
00188    }
00189 }
00190
00191
00192 #endif
.....

```

10.11 队列内部结构

使用 `queue_t` 定义一个队列，每个队列包含一个队列控制块（QCB），用来保存队列运行时的状态信息。定义一个 `queue_t` 类型的队列时，就创建了一个队列控制块（QCB）。表 10.7 列出了 QCB 结构说明：

表 10.7 队列控制块 QCB 结构说明

项	描 述
name	队列的名字
avail	有效消息数

msgsz	消息尺寸
entries	队列的长度
pstart	队列缓冲区的起始地址
pend	队列缓冲区的末地址
phead	指向队列头的指针
ptail	指向队列尾的指针
init	初始化标志
waits	队列的等待队列

了解 QCB 各项的意义，有助于理解队列的内部机制，方便程序测试。注意，队列控制块必须通过系统提供的服务来管理，千万不要直接更改控制块各项的值，否则，将引起系统的异常。

10.12 总 结

本章首先介绍了队列主要属性，分析了队列的结构，然后对队列相关的服务逐一做了介绍，并给出了一些示例代码和图示，最后，给出了一个例子演示了使用队列传递缓冲区信息的方法。

队列是任务间首选的通讯方式。队列可以存放一个或者多个消息，队列中的消息以一定的顺序存放。按照消息进出队列的顺序，消息队列包括先进先出（First In First Out，FIFO）和后进先出（Last In First Out，LIFO）两种。

消息的大小不受限制，只要有足够的内存空间，可以存放任何尺寸大小的消息。但是，在传递大尺寸消息时，为了提高消息进出队列的速度，最好的方法是使用间接队列消息的方式。

使用队列传递消息有两种方法：直接队列消息和间接队列消息。直接队列消息适合传递信息量较少的消息；间接队列消息适合传递大尺寸消息，使用间接队列消息，仅仅是传递消息缓冲区的地址，因为没有复制缓冲区内容的开销，故传送速度非常快。

直接队列消息的优点是简单直观，缺点是传送大尺寸消息时速度较慢；间接队列消息在大尺寸消息时也能获得同样快的速度，缺点是需要管理消息缓冲区，容易出错。

队列的速度与队列长度无关，而与消息的大小有关。在传送小尺寸消息时，队列传输消

息的效率非常高，RS-RTOS 内核对几个常用的消息尺寸进行了特别优化：

RS_MSG_1ULONG

RS_MSG_2ULONG

RS_MSG_4ULONG

RS_MSG_8ULONG

分别对应 4 字节、8 字节、16 字节和 32 字节的消息尺寸。在应用设计时，要优先考虑使用以上尺寸的消息，有助于进一步提高消息队列的吞吐效率。比如，消息的内容只需要 7 字节就足够时，应该使用大小为 8 字节的消息。

第 11 章

事件标志

11.1 介绍

事件标志（Event Flags）是任务同步的强有力的工具。事件标志包含一个宽 32 位的数据结构，这个结构称为事件标志位组。事件标志位组中每一位代表一个事件，一共可以表示 32 个不同的事件，并且这些事件互相独立，互不干扰。

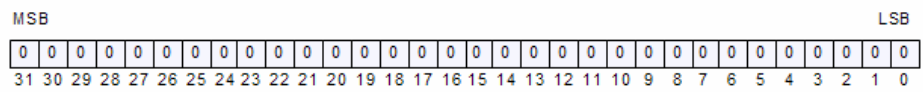


图 11.1 RS-RTOS 事件标志位组

当标志位被置 1，表示对应的事件发生；当标志位为 0，表示对应的事件清除。程序可以对事件标志中的一个或者多个事件置位或者清除。

事件标志中的单个事件位可以任意组合，任务可以等待事件标志中的某一事件或者一个事件组合，并可指定事件满足的条件。如果等待的事件条件不符合，任务将进入阻塞状态。直到事件标志满足条件，任务将会被恢复。

11.2 事件标志中的事件

事件标志位组是一个 32 位的数据结构，每一位都可以代表一个事件。位与位之间既可以互相独立，又可以任意组合，这样一来，事件的形式就非常灵活丰富。比如，图 11.2 显示了发生标志 5 被置位的事件。

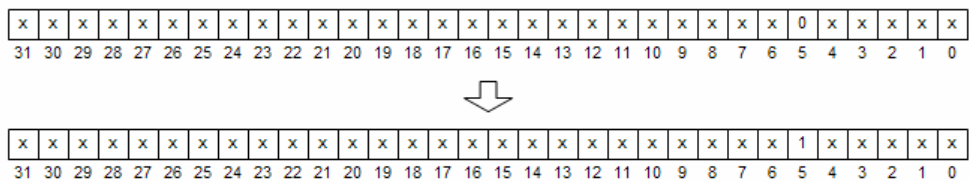


图11.2 标志位5置位

事件标志位可以任意组合。如图 11.3 所示，表示了标志位 0、6、28、30 同时被置位的情况。

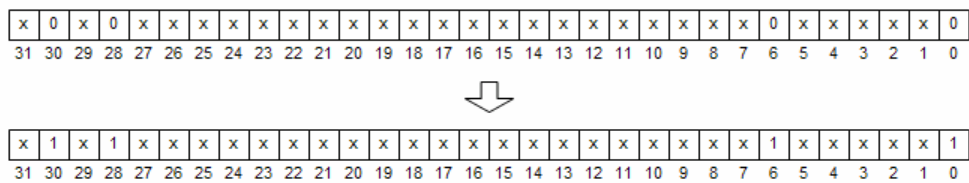


图11.3 标志位0、6、28、30置位

不仅仅标志位置 1 可以表示事件发生，同样，标志位由 1 被清为 0 也可以表达一个事件。事件是一个动作而不是一个状态。所以，只要标志位组的值发生了改变，就认为发生了事件。每个任务都可以表达对不同的事件的关注，并且可以规定事件的组合逻辑。被关心的标志位组合成为标志位组，并且需要满足一定的逻辑运算，才能宣告事件成立。标志位的逻辑运算有以下方式：

1) 正逻辑与 (RS_OPT_AND)

仅当所有位被置 1 时成立。例如，标志 0、6 组合，使用正逻辑与运算，仅当标志 0、

6 同时为 1 时成立。

2) 正逻辑或 (RS_OPT_OR)

任何一位或多位被置 1 时成立。例如，标志 0、6 组合，使用正逻辑或运算，当标志 0、6 中任意位为 1 时成立。

3) 负逻辑与 (RS_OPT_NAND)

仅当所有位被置 0 时成立。例如，标志 0、6 组合，使用负逻辑与运算，仅当标志 0、6 同时为 0 时成立。

4) 负逻辑或 (RS_OPT_NOR)

任何一位或多位被置 0 时成立。例如，标志 0、6 组合，使用负逻辑或运算，当标志 0、6 中任意位为 0 时成立。

5) 反逻辑与 (RS_OPT_XAND)

仅当所有位被置反时成立。例如，标志 0、6 组合，原值为 0、1，当值被置为 1、0 时成立。图 11.4 显示了标志 0、6 反逻辑与成立的一种情况。

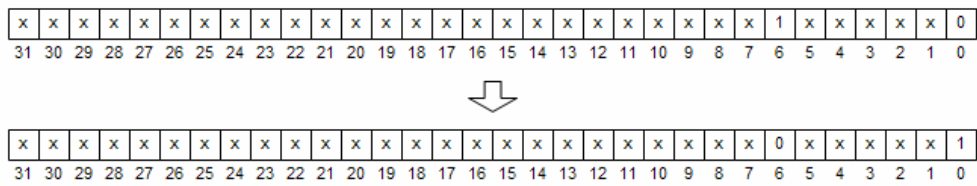


图11.4 标志位0、6满足反逻辑与

6) 反逻辑或 (RS_OPT_XOR)

任何一位或多位被置反时成立。例如，标志 0、6 组合，原值为 0、1，如果标志 0 被置 1，则条件成立。

反逻辑或和反逻辑与是 RS-RTOS 特有的事件类型。反逻辑关心事件的改变，而不关心事件改变前后的状态，这一特性在处理信号改变时非常奏效。

11.3 事件标志属性

事件标志的属性被保存在事件标志控制块中（Event Control Block, ECB），ECB 是用来保存运行时（Run-time）事件标志状态信息的数据结构。表 11.1 是事件标志的主要属性说明。

表 11.1 事件标志控制块 ECB 主要属性

项	描 述
name	事件标志的名字
bits	事件标志位组

事件标志控制块可以存在于内存的任意位置，通常被定义为全局变量，以便它能够被多个任务访问。

11.4 事件标志服务综述

附录 L 包含了与事件标志相关的各项服务接口的详细信息。包括函数的原型、简要介绍、参数、返回值、调用者、阻塞状态、注意事项。

表 11.2 包括系统提供的事件标志服务接口的列表。在本章后面的部分，将对这些服务进行详细的介绍，同时给出一些演示代码。

表 11.2 邮箱服务接口

服 务	描 述
event_create	创建事件标志
event_delete	删除事件标志
event_wait	获取（等待）事件
event_trywait	获取（无等待）事件
event_post	设置事件
event_info	获取事件标志信息

11.5 创建事件标志

使用事件标志前，首先需要创建该事件标志，并对其初始化。事件标志声明为 `event_t` 类型，当定义一个 `event_t` 类型的事件标志时，编译器将分配事件标志控制块 `ECB`。获得一个事件标志控制块后，需要调用接口：

```
status_t event_create(event_t __p_* event, name_t name, bits_t bits);
```

完成事件标志的创建和初始化工作。表 11.3 说明了创建邮箱接口的传入参数。

表 11.3 接口 `event_create` 参数说明

参 数	描 述
event	指向事件标志的指针
name	事件标志的名字
bits	事件标志位组的初始值

事件标志位组的结构请参考图 11.1，例子 APP_SAMPLE_11A 演示了如何创建一个事件标志：

【例 APP_SAMPLE_11A】

file: example\WIN32\app_sample_11a.cpp

```
00047
00048 /* 事件标志 */
00049 event_t app_event1;
.....
00062     status_t status;
.....
00067     /**
00068      * 创建一个事件标志 EV1
00069      * 事件标志位组初始化为 0 */
00070     status = event_create(
00071         &app_event1,
```

```
00072      build_name('E', 'V', '1', '\0'),
00073      0);
.....
```

创建事件标志时，必须指定一个合法的事件标志结构指针，不允许指定为空指针(NULL)，否则，将引发一个断言错误。事件标志的名字通过 **build_name** 建立，如果指定为 0，表示创建一个未命名的事件标志。可以为事件标志位组指定一个初始值，默认情况下，事件标志位组的所有位被初始化为 0。

11.6 删除事件标志

删除事件标志可以为系统节约一些资源开销，但是，不能删除一个正在使用中的事件标志。删除事件标志服务的接口原型如下：

```
status_t  event_delete(event_t __p_* event);
```

删除一个事件标志只需要给出指向该事件标志的指针，该参数不允许为空指针(NULL)，表 11.4 是删除事件标志的接口参数说明。

表 11.4 接口 **event_delete** 参数说明

参 数	描 述
event	指向事件标志的指针

例子 APP_SAMPLE_11B 演示了如何删除一个不再使用的事件标志：

【例 APP_SAMPLE_11B】

file: example\WIN32\app_sample_11b.cpp

```
00047
00048 /* 事件标志 */
00049 event_t app_event1;
.....
00062      status_t status;
```



```
.....
00067      /**
00068      * 创建一个事件标志 EV1
00069      * 事件标志位组初始化为 0 */
.....
00078      /**
00079      * 使用事件标志.....
00080      */
.....
00083      /* 删除事件标志 EV1 */
00084      status = event_delete(&app_event1);
.....
```

删除一个事件标志之前，必须确保该事件标志不再被使用，并且没有任务等待该事件标志的事件。如果有一个或者多个任务正在等待事件标志中的事件时，删除该事件标志是危险的，调用删除事件标志服务将获得一个失败代码，指示删除失败。

当事件标志被成功删除后，不能再继续使用该事件标志，除非重新创建它。

11.7 获取（等待）事件

事件标志服务中，任务常常需要判定是否发生了某个事件。通过获取事件服务，当事件标志位组中的指定事件标志位被置位，则服务满足。如果事件不满足，则任务进入等待状态或者返回失败代码。获取（等待）事件服务的接口原型如下：

```
status_t event_wait(event_t __p_* event, bits_t bits, int8u options, tick_t ticks);
```

表 11.5 给出了该接口参数说明：

表 11.5 接口 event_wait 参数说明

参 数	说 明
event	指向事件标志的指针

bits	请求的事件标志位组
options	服务选项
ticks	指定的超时时间，单位为系统节拍（Tick），如指定 RS_WAIT_FOREVER 表示无限期等待，直到信号量有可用实例。

表中 RS_WAIT_FOREVER 是内核定义的宏，其值为 0，也就是说，当超时 ticks 参数设置为 0 时，如果信号量的值为 0，调用该服务的任务将一直等待，直到所需信号量有实例可用。

请求的事件标志位组是事件标志位的组合，该参数指出了调用者关心的事件标志位。例如，当希望得到标志 0、2 的事件时，则指定为十六进制 0x0005；如果希望得到标志 0、3、4 的事件，则对应的十六进制是 0x0019。

服务选项 options 则给出服务的操作选项，其中最重要的功能是，给出标志位组的逻辑运算规则。表 11.6 列出了的这些选项：

表 11.6 事件标志逻辑运算选项说明

参 数	说 明
RS_OPT_AND	正逻辑与，所有位被置 1 时成立
RS_OPT_OR	正逻辑或，任何一位或多位被置 1 时成立
RS_OPT_NAND	负逻辑与，所有位被置 0 时成立
RS_OPT_NOR	负逻辑或，任何一位或多位被置 0 时成立
RS_OPT_XAND	反逻辑与，所有位被置反时成立
RS_OPT_XOR	反逻辑或，任何一位或多位被置反时成立

例如，如果希望得到标志 3 被置 1 的事件，则 bits 的取值为十六进制 0x0008，options 可以取 RS_OPT_AND 或者 RS_OPT_OR，图 11.5 显示了标志 3 被置位。

例子 APP_SAMPLE_11C 给出了代码。

【例 APP_SAMPLE_11C】

file: example\WIN32\app_sample_11c.cpp

```
00062
```

```
00063 /* 事件标志 */
00064 event_t app_event1;
.....
00120     status_t status;
00121
00122     /**
00123      * 等待事件标志位 3 被置 1 的事件
00124      * 如果事件不满足,任务将一直阻塞 */
00125     status = event_wait(
00126         &app_event1,
00127         0x0008,
00128         RS_OPT_AND,
00129         RS_WAIT_FOREVER
00130     );
.....
```

在获取事件的服务中，有一个选项是 RS_OPT_CONSUME（事件消费）。它的特殊之处在于能够改变事件标志位的值。

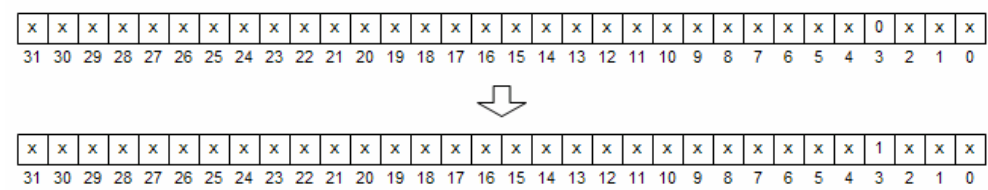


图11.5 标志位3被置位

RS_OPT_CONSUME 选项能够将事件标志复位，如例子 APP_SAMPLE_11C，如果使用事件消费选项，则当事件满足后，标志 3 将被复位为 0。选项 RS_OPT_CONSUME 与事件标志逻辑运算选项组合使用，如示例代码 APP_SAMPLE_11D 所示。

【例 APP_SAMPLE_11D】

file: example\WIN32\app_sample_11d.cpp

```

00062
00063 /* 事件标志 */
00064 event_t app_event1;
.....
00120     status_t status;
00121
00122     /**
00123      * 等待事件标志位 3 被置 1 的事件
00124      * 如果事件不满足,任务将一直阻塞 */
00125     status = event_wait(
00126         &app_event1,
00127         0x0008,
00128         RS_OPT_AND | RS_OPT_CONSUME,
00129         RS_WAIT_FOREVER
00130     );
.....

```

使用 RS_OPT_CONSUME 选项时,当服务条件满足时,将使服务接口参数 bits (请求事件标志位组) 中指定的标志位复位 (置 0),而不是将标志位恢复到事件前的状态。

举个例子,标志 3、4、5 的初始值均为 1,使用选项 RS_OPT_NOR (负逻辑或) 和

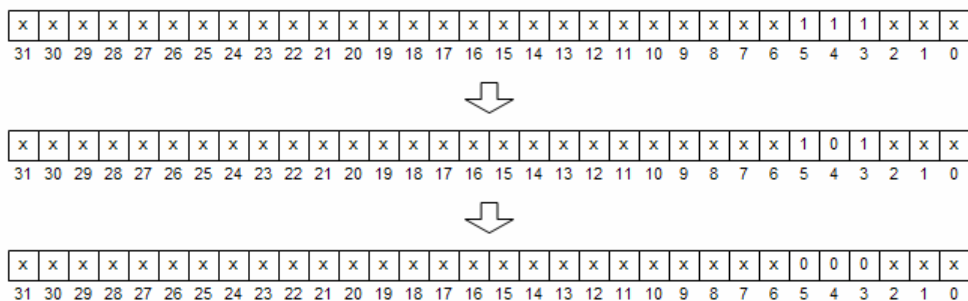


图11.6 使用事件消费选项

RS_OPT_CONSUME，当事件满足时（例如标志 4 被置 0），则 `event_wait` 服务将成功返回，并将标志 3、4、5 复位，在这个过程中事件标志位组值的变化如图 11.6 所示。

11.8 获取（无等待）事件

某些情况下，程序不希望由于事件条件不满足而进入阻塞状态，或者程序不能被阻塞（中断服务程序、定时器应用程序）。使用无等待获取事件服务，程序无需等待，亦可获得事件。无等待获取事件服务的接口原型为：

```
status_t event_trywait(event_t __p_ * event, bits_t bits, int8u options);
```

表 11.7 给出该接口的参数说明：

表 11.7 接口 `event_trywait` 参数说明

参 数	说 明
event	指向事件标志的指针
bits	请求的事件标志位组
options	服务选项

任务调用 `event_trywait` 服务不会进入等待状态，如果指定的事件没有满足，接口也会立刻返回，这种情况下，将返回错误代码，指示事件不满足。例子 APP_SAMPLE_11E 演示了无等待获取事件服务的使用方法：

【例 APP_SAMPLE_11E】

file: example\WIN32\app_sample_11e.cpp

```
00062
00063 /* 事件标志 */
00064 event_t app_event1;
.....
00120     status_t status;
.....
00125     /**
```

```
00126      * 等待事件标志位 3 被置 1 的事件
00127      * 使用该接口任务不会阻塞 */
00128      status = event_trywait(
00129          &app_event1,
00130          0x0008,
00131          RS_OPT_AND
00132      );
.....
```

普通任务、中断服务例程（ISR）、定时器应用可以通过 `event_trywait` 获得事件标志，而 `event_wait` 只能在任务中使用。

11.9 设置事件

通过设置事件服务，可以对一个或者多个标志位置位（置 1）或复位（清 0）。设置事件服务的接口原型如下：

```
status_t event_post(event_t __p_* event, bits_t bits, int8u options);
```

在表 11.8 列出这些参数的说明：

表 11.8 接口 `event_post` 参数说明

参 数	说 明
event	指向事件标志的指针
bits	设置的事件标志位组
options	设置选项

设置事件标志位组是事件标志位的组合，该参数给出了调用者关心的事件标志位。例如，当希望设置标志 0、1、4 的事件时，则指定为十六进制 `0x0013`。

选项 `options` 给出服务的操作选项，该选项说明对标志位组进行置位还是复位操作。表 11.9 列出了设置选项说明：

表 11.9 设置事件选项

参 数	说 明
RS_OPT_SET	将事件位置为 1
RS_OPT_CLR	将事件位置为 0

例子 APP_SAMPLE_11E 使用了设置事件服务，行 L00154 对标志 3 设置置位事件。

【例 APP_SAMPLE_11E】

file: example\WIN32\app_sample_11e.cpp

```
00062
00063 /* 事件标志 */
00064 event_t app_event1;
.....
00153      /* 置位事件标志位 3 */
00154      event_post(&app_event1, 0x0008, RS_OPT_SET);
.....
```

11.10 获取事件标志信息

调用 event_info 可以获取事件标志运行时（Run-time）的状态信息。得到的信息包括事件标志的名字，事件标志位组信息。获取事件标志信息的接口原型为：

```
status_t event_info(event_t __p_* event, evinfo_t __out_* info);
```

该接口传入一个 evinfo_t 结构（evinfo_t 是 event information 的缩写）指针参数，用来存放服务接口返回的各信息值，表 11.10 给出 event_info 服务接口参数详细说明：

表 11.10 接口 event_info 参数说明

参 数	说 明
event	指向事件标志的指针

info	name	返回事件标志的名字
	bits	返回事件标志位组状态

例子 APP_SAMPLE_11F 演示了利用接口 event_info 获取事件标志的信息：

【例 APP_SAMPLE_11F】

file: example\WIN32\app_sample_11f.cpp

```
00060
00061 /* 事件标志 */
00062 event_t app_event1;
.....
00107     status_t status;
00108     evinfo_t info;
00109
00110     /* 获取事件标志 EV1 信息. */
00111     status = event_info(&app_event1, &info);
00112
00113     if (status == RS_EOK) {
00114
00115         /* 显示事件标志信息 */
00116         printf("==Event Infomation=====\n");
00117         printf("Event Name : %s\n", info.name);
00118         printf("Event Bits : %#06x\n", info.bits);
00119     }
.....
```

11.11 使用事件标志实现同步

考虑一个缓冲区信息存储的问题：由两个任务 TA1、TA2 完成读卡器信息的输入。任务 TA1 负责从读卡器上把卡片信息读到一个缓冲区中，任务 TA2 负责把该缓冲区中的信息进行

加工并写到外部存储器中。要实现两者的协同工作，两个任务必须满足如下制约关系：只有当取空该缓冲区中的内容时，任务 TA1 才能其中写入新的信息；只有当写满该缓冲区时，任务 TA2 才能从中取出内容做进一步的加工和转送工作。可见，在缓冲区内容尚未取走时，任务 TA1 应该等待，防止把原有的信息冲掉，造成信息丢失。针对该缓冲区，任务 TA1、TA2 就是一种同步关系。

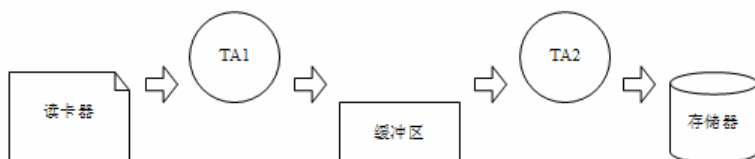


图11.7 缓冲区同步问题

可以看出，使用缓冲区的两个任务需要交换两个信息：缓冲区空和缓冲区满的状态。当缓冲区空时，任务 TA1 才能把信息存入缓冲区中；当缓冲区满时，表示其中有可加工的信息，任务 TA2 才能从中取出信息。信息使用者（任务 TA2）不能超前信息提供者（任务 TA1），即缓冲区中未存入信息时不能从中取出信息；如信息提供者已经把缓冲区写满，但使用者尚未取走信息时，信息提供者不能又写入信息，避免冲掉前面的信息。

为此，设置两个标志位：

标志 0 表示缓冲区是否空（置 0 表示不空，置 1 表示空）。

标志 1 表示缓冲区是否满（置 0 表示不满，置 1 表示满）。

规定标志 0、1 的初值分别为 1、0，则对缓冲区的信息提供者和使用者的同步的关系如图 11.8 所示。

该演示完整代码在例 APP_SAMPLE_11G 中给出。



图11.8 缓冲区读写流程

11.12 示例 APP_SAMPLE_11G 代码

【例 APP_SAMPLE_11G】

file: example\WIN32\app_sample_11g.cpp

```
.....
00038 #include <stdio.h>
00039 #include "inc/kapi.h"
00040 #include "app_sample.h"
00041
00042 #ifdef APP_SAMPLE_11G
00043
00044 /**
00045  * 例子 APP_SAMPLE_11G 演示了如何使用事件标志实现同步
00046  */
00047
00048
00049 /* 任务栈大小 */
00050 #define APP_STACK_SIZE      1024 * 32
00051
00052 /* 缓冲区大小 */
00053 #define APP_BUFF_SIZE      100
00054
00055 enum {
00056     /* 任务优先级 */
00057     APP_PRIO_TA1      = 10,
00058     APP_PRIO_TA2      = 11,
00059 };
00060
00061
00062 /* 任务栈 */
```

```
00063 stack_t app_stack_ta1[APP_STACK_SIZE];
00064 stack_t app_stack_ta2[APP_STACK_SIZE];
00065
00066 /* 事件标志 */
00067 event_t app_event1;
00068
00069 /* 缓冲区 */
00070 char app_buff[APP_BUFF_SIZE];
00071
00072
00073 void app_task_1(arg_t arg);
00074 void app_task_2(arg_t arg);
00075
00076
00077 /* 硬件初始化 */
00078 void hardware_initialize(void)
00079 {
00080
00081 }
00082
00083
00084 /* 应用初始化 */
00085 void application_initialize(void)
00086 {
00087     status_t status;
00088
00089     /* 打印关于示例一些信息 */
00090     APP_SAMPLE_INFO;
00091
00092     /* 创建任务 TA1 */
00093     status = task_create(
```

```
00094     APP_PRIO_TA1,
00095     build_name('T', 'A', '1', '\0'),
00096     app_task_1, 0,
00097     app_stack_ta1, APP_STACK_SIZE,
00098     0);
00099
00100     ASSERT(status == RS_EOK);
00101
00102     /* 创建任务 TA2 */
00103     status = task_create(
00104         APP_PRIO_TA2,
00105         build_name('T', 'A', '2', '\0'),
00106         app_task_2, 0,
00107         app_stack_ta2, APP_STACK_SIZE,
00108         0);
00109
00110     ASSERT(status == RS_EOK);
00111
00112     /**
00113      * 创建一个事件标志 EV1
00114      * 事件标志位组初始化为 0x0001 */
00115     status = event_create(
00116         &app_event1,
00117         build_name('E', 'V', '1', '\0'),
00118         0x0001);
00119
00120     ASSERT(status == RS_EOK);
00121 }
00122
00123 /* 任务 TA1 入口 */
00124 void app_task_1(arg_t)
```

```
00125 {
00126     int i = 0;
00127
00128     for (;;)
00129     {
00130         /**
00131          * 等待事件标志位 0 被置 1 */
00132         event_wait(
00133             &app_event1,
00134             0x0001,
00135             RS_OPT_AND | RS_OPT_CONSUME,
00136             RS_WAIT_FOREVER);
00137
00138         /**
00139          * 读入卡片信息,写入缓冲区 */
00140         sprintf(app_buff, "Read a message (id:%d)\n", i++);
00141         /**
00142          * 读卡信息需要的时间 */
00143         task_sleep(RS_TICK_FREQ / 10);
00144
00145         /**
00146          * 缓冲区已满,将标志 1 置位 */
00147         event_post(&app_event1, 0x0002, RS_OPT_SET);
00148     }
00149 }
00150
00151 /* 任务 TA2 入口 */
00152 void app_task_2(arg_t)
00153 {
00154     for (;;)
00155     {
```

```

00156      /**
00157      * 等待事件标志位 1 被置 1 */
00158      event_wait(
00159          &app_event1,
00160          0x0002,
00161          RS_OPT_AND | RS_OPT_CONSUME,
00162          RS_WAIT_FOREVER);
00163
00164      /**
00165      * 写入存取区(使用输出缓冲区信息到控制台代替) */
00166      printk(app_buff);
00167
00168      /**
00169      * 缓冲区已空,将标志 0 置位 */
00170      event_post(&app_event1, 0x0001, RS_OPT_SET);
00171  }
00172 }
00173
00174
00175 #endif
.....

```

11.13 事件标志内部结构

使用 `event_t` 定义一个事件标志，每个事件标志包含一个事件标志控制块（ECB），用来保存事件标志运行时（Run-time）的状态信息。定义一个 `event_t` 类型的事件标志时，就创建了一个事件标志控制块（ECB）。表 11.11 列出了 ECB 结构说明：

表 11.11 事件标志控制块 ECB 结构说明

项	描 述
name	队列的名字
bits	事件标志位组
cares	指定标志位组
init	初始化标志
waits	事件标志的等待队列

了解 ECB 各项的意义，有助于理解事件标志的内部机制，方便程序测试。注意，事件标志控制块必须通过系统提供的服务来管理，不允许直接更改控制块各项的值，否则，将引起系统的异常。

11.14 总 结

事件标志为任务同步提供了强有力的工具。事件标志不支持所有权，也不限定同时访问事件标志的任务数。

事件标志可以独立表示多个独立的事件，也可以任意的组合标志位组，处理复杂形式的事件。

事件标志不仅可以在任务中使用，也可以被中断服务例程、定时器应用程序访问。

获取事件时，允许应用指定所关心的标志位的组合，并设定事件成立的逻辑运算条件，当标志位组满足事先指定的逻辑运算后，才能宣告事件成立。逻辑运算操作大大的丰富了事件的表现形式。

任务在等待的事件未满足时，可以被阻塞或者返回失败代码，取决于所选用的服务接口和选项。

系统能够创建的事件标志的数量不受设计限制，只要有足够的存储器资源，可以使用任何数量的事件标志。

第 12 章

时 钟

12.1 介 绍

RS-RTOS 内部提供了一个宽 32 位的时钟节拍计数器，该计数器称为系统时钟或系统节拍。在每次系统定时器中断产生时，系统节拍值加 1。

系统节拍值的范围是 0 到 4 294 967 295（或 $2^{32} - 1$ ），当系统节拍到达最大值后，在下一个系统定时器中断产生时将溢出，从 0 开始新一轮的计数过程。

在应用初始化时，系统节拍被初始化为 0。应用程序可以通过时钟服务接口取得和设置系统节拍的值。

系统时钟的值反应了系统定时器的中断次数，但应用往往更加关心系统节拍流逝的真实时间。在 RS-RTOS 内核中预定义了一个宏：

RS_TICK_FREQ

这个宏定义了系统时钟的频率，单位是 ticks/s。代表了在单位时间（1 秒），系统定时器发生的中断次数。通过这个宏，可以得到系统节拍所代表的时间转换公式：

$$\text{时 间} = \text{系统时钟} / \text{RS_TICK_FREQ}$$

12.2 时钟服务综述

与系统时钟服务相关的接口有两个，表 12.1 给出了系统时钟服务接口的列表。另外，在附录 D 中有这些接口的详细使用规则。下面将介绍系统时钟服务的使用：

表 12.1 时钟服务接口

参 数	描 述
tick_get	获取系统节拍
tick_set	设置系统节拍

12.3 获取系统节拍

在应用初始化时，系统时钟初始化为 0，在以后的每个系统定时器中断发生时，系统时钟的值加 1。通过获取系统节拍服务可以获得当前系统时钟的值，获取系统节拍服务的接口原型如下：

```
int32u  tick_get(void);
```

该接口没有传入参数，其返回值是系统当前的时钟值。

12.4 设置系统节拍

系统时钟主要是提供给应用程序开发者使用，RS-RTOS 不使用系统时钟做任何事情。任何时候，出于应用系统的需要，都可以对当前时钟进行修改，并不会影响系统内部的运行状态。如果需要设置系统时间节拍，通过接口：

```
void  tick_set(int32u ticks);
```

该接口传入参数如表 12.2。

表 12.2 接口 tick_set 参数说明

参 数	说 明
ticks	指定的系统节拍值

设置系统节拍后，系统内部时钟的值将被修改为指定的节拍，直到下一个系统定时器中断到来，系统时钟的计数值将加 1。

12.5 总 结

由系统时钟的内部机制可以知道，系统时钟的精确性取决于硬件定时器。只要硬件定时器的精确性有保障，就可以获得很高精度的系统时钟。

系统时钟的分辨率一般为毫秒级，常用的系统时钟的频率在 200 次/秒到 10 次/秒之间，这个值被定义在宏

RS_TICK_FREQ

系统时钟为应用程序提供了时间上的参照物，如果随意修改系统时钟的值，可能会破坏或者削弱这种参照作用。RS-RTOS 不依赖系统时钟做任何事情，修改系统时钟不会给内核带来副作用。

第 13 章

定时器

13.1 介绍

对外部事件进行快速响应，是实时嵌入式应用系统中最重要的特征。除此之外，应用还必须具备能够在预定的时间间隔内完成某项活动的能力。

使用定时器，能够在特定的时间间隔执行指定的应用程序。在嵌入式系统中，从定时器实现的载体区分，可分为硬件定时器和软件定时器。硬件定时器通过专用的时钟芯片、或者由微控制器的外围定时模块来产生。硬件定时器定时分辨率和精度很高，可以达到纳秒（ns）级甚至更高。

但是，在嵌入式应用中，往往不需要这么高分辨率的定时器，而且不能为每个定时器应用配备一个硬件定时器。这样，软件定时器因为简单、通用、易于扩展等优点，更容易被应用开发者所青睐。本章所介绍的定时器就是指 **RS-RTOS** 内建的软件定时器。

软件定时器本身也是基于硬件定时器实现的。所以，软件定时器的精度是有保障的，只是在分辨率上不如硬件定时器。

RS-RTOS 内建了定时器模块，可以实现单次定时器和周期定时器。单次定时器仅仅执行一次定时器服务函数；而周期定时器以指定的周期重复执行定时器的服务函数。

定时器指定的到期时间，称为定时器周期（用 **T** 表示）。定时器周期是系统定时器中断的整数倍，一个周期单位与一个系统时钟相等。例如，在系统时钟频率为 **100Hz** 时，最小的定时器周期为 **10ms**。系统时钟和定时器使用了统一的时间基准，这样，在使用定时器应用时显

得非常方便。

定时器启动后，每个系统时钟到来时，定时器的周期计数器会减 1，直到计数器减为 0 时，定时器到期。由于定时器启动的时刻，通常不是恰好在系统时钟到达的时刻，所以，第一个定时周期实际在 $T-1$ 到 T 之间，如图 13.1 所示。

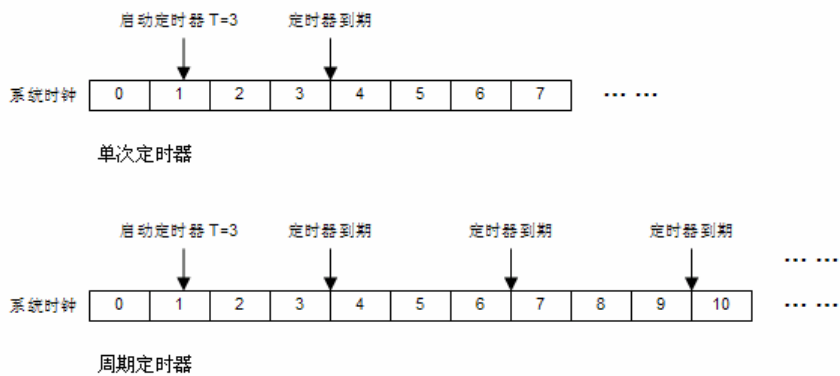


图 13.1 定时器

定时器应用的优先级比所有任务的优先级都高，因此，定时器应用能够抢占当前运行的任务。不过，定时器应用与中断服务例程的优先级高低没有作出规定，因为不同的系统对此需求不同，但是，在一个确定硬件体系的系统上，它们的优先级别是确定的。

当定时器指定的周期到达，系统产生定时器中断，调用指定的定时器到期函数。定时器到期函数在定时器创建时指定，定时器本质上也是一种中断，因此，不能在定时器到期函数中使用可能引起程序阻塞的系统服务。通过附录，可以了解各服务接口是否允许在定时器中使用。

13.2 定时器属性

定义一个定时器时，就创建了一个定时器控制块 (Timer Control Block, TMCB)。TMCB 用于存储定时器运行时 (Run-time) 的状态信息，包括定时器名字、到期服务函数、定时器周期等。TMCB 可以在内存中任何位置，可以通过申请动态内存获得，但通常把它定义为全局变量，以便不同的任务和程序能够访问。

定时器的属性保存在控制块 TMCB 中。表 13.1 列出了定时器的主要属性：

表 13.1 定时器控制块 TMCB 主要属性

项	描 述
name	定时器的名字
entry	定时器到期函数
arg	到期函数参数
ticks	定时器周期
options	定时器选项

当定时器指定的周期到达，系统产生一个定时器中断，调用指定的定时器到期函数。定时器周期指定了定时器到期的时间间隔，单位是系统的时钟节拍，因此，定时器周期的最小分辨率是系统时钟。定时器选项可以指定使用单次定时器或者周期定时器。

13.3 定时器服务综述

附录 F 描述了与定时器相关的各项服务接口的详细信息。包括函数的原型、简要介绍、参数、返回值、调用者、阻塞状态、注意事项。

表 13.2 包括系统提供的定时器服务的接口列表。在本章后面的部分，将对这些服务进行详细的介绍，同时给出一些演示代码和图示。

表 13.2 定时器服务接口

服 务	描 述
timer_create	创建定时器
timer_delete	删除定时器
timer_enable	启动定时器
timer_disable	停止定时器
timer_control	定时器控制

13.4 创建定时器

使用定时器前，首先需要创建该定时器，并对其初始化。定时器声明为 `timer_t` 类型，当定义一个 `timer_t` 类型的定时器时，编译器将分配定时器控制块 `TMCB`。

通过调用服务接口：

```
status_t timer_create(
    timer_t __p_* timer,
    name_t name,
    entry_t entry,
    arg_t arg,
    tick_t ticks,
    int8u options
);
```

完成定时器的创建和初始化工作。表 13.3 描述了创建定时器接口的传入参数。

表 13.3 接口 `timer_create` 参数说明

参 数	描 述
timer	指向定时器的指针
name	定时器的名字
entry	定时器到期函数
arg	到期函数参数
ticks	定时器周期
options	定时器选项

创建定时器时，必须指定一个合法的定时器结构指针，不允许指定为空指针（`NULL`），否则，将引发一个断言错误。定时器的名字通过 `build_name` 建立，如果指定为 0，表示创建一个未命名的定时器。定时器到期函数声明为以下形式：

```
void app_timer_1(arg_t arg)
```


到期函数没有返回值，入口参数 `arg` 在创建定时器时指定，注意，不能在到期函数中使用可能引起程序阻塞的系统服务。定时器最小周期为 1 个系统时钟，如果没有指定，默认为 1 个系统时钟。表 13.4 列出定时器选项：

表 13.4 定时器选项

参 数	描 述
RS_OPT_ENABLE	启动定时器，默认选项
RS_OPT_DISABLE	停止定时器
RS_OPT_SINGLE	单次定时器，默认选项
RS_OPT_REPEAT	周期定时器

如果不指定定时器选项，默认将创建并立刻启动一个单次的定时器。多个选项可以通过位或“|”指定。

例子 APP_SAMPLE_13A 演示了如何创建一个定时器：

【例 APP_SAMPLE_13A】

file: example\WIN32\app_sample_13a.cpp

```
.....
00048 /* 定时器 */
00049 timer_t app_timer1;
.....
00066     status_t status;
.....
00071     /**
00072      * 创建并启动一个名为 TM1 的周期定时器
00073      * 定时器周期为 20 个系统时钟周期 */
00074     status = timer_create(
00075         &app_timer1,
00076         build_name('T', 'M', '1', '\0'),
00077         app_timer_1,
00078         (arg_t)0x01234567,
00079         20,
```

```
00080      RS_OPT_ENABLE | RS_OPT_REPEAT
00081      );
.....
```

13.5 删除定时器

删除定时器可以为系统节约资源开销。删除定时器服务的接口原型为：

```
status_t timer_delete(timer_t __p_* timer);
```

删除一个定时器只需要给出指向该定时器的指针，并且不允许为空指针（NULL）。表 11.4 是删除定时器的接口参数说明。

表 13.5 接口 timer_delete 参数说明

参 数	描 述
timer	指向定时器的指针

如果定时器正在进行中断服务，删除定时器是不允许的，此时，调用删除定时器服务将返回失败代码。如果在任务中使用删除定时器服务，则不用担心这个问题，因为定时器的优先级比任务高，因此，任务不会抢占并删除一个正在进行到期服务的定时器。

例子 APP_SAMPLE_13B 演示了如何删除定时器：

【例 APP_SAMPLE_13B】

file: example\WIN32\app_sample_13b.cpp

```
.....
00061 /* 定时器 */
00062 timer_t app_timer1;
.....
00097 /**
00098      * 创建并启动一个名为 TM1 的周期定时器
00099      * 定时器周期为 20 个系统时钟周期 */
```

```
.....
00116      status_t status;
.....
00119      /**
00120      * 删除定时器 TM1 */
00121      status = timer_delete(&app_timer1);
.....
```

13.6 启动定时器

如果定时器在创建时使用 `RS_OPT_DISABLE` 选项，或者应用了停止定时器服务接口，定时器将会保持停止状态，通过启动定时器服务可以将定时器重新激活。启动定时器服务的接口原型为：

```
status_t timer_enable(timer_t __p_* timer);
```

如果定时器已经在活动状态，该服务不会产生任何操作。表 13.6 给出启动定时器服务的参数说明。

表 13.6 接口 `timer_enable` 参数说明

参 数	描 述
timer	指向定时器的指针

如果多个定时器同时到期，相应的到期服务函数将按照定时器启动时间的顺序执行。例子 APP_SAMPLE_13C 演示了如何启动定时器：

【例 APP_SAMPLE_13C】

file: example\WIN32\app_sample_13c.cpp

```
.....
00061 /* 定时器 */
00062 timer_t app_timer1;
```

```
.....
00117     status_t status;
00118
00119
00120     /**
00121      * 启动定时器 TM1 */
00122     status = timer_enable(&app_timer1);
.....
```

13.7 停止定时器

停止定时器使定时器变为非活动状态。如果定时器已经处于停止状态，这个服务将不会产生任何操作。停止定时器服务的接口原型为：

```
status_t timer_disable(timer_t __p* timer);
```

处于停止状态的定时器，可以通过启动定时器服务将其重新激活。表 13.7 给出停止定时器服务的参数说明。

表 13.7 接口 **timer_disable** 参数说明

参 数	描 述
timer	指向定时器的指针

例子 APP_SAMPLE_13C 演示了如何停止定时器：

【例 APP_SAMPLE_13C】

file: example\WIN32\app_sample_13c.cpp

```
.....
00061 /* 定时器 */
00062 timer_t app_timer1;
.....
```

```
00117      status_t status;
.....
00132      /**
00133      * 停止定时器 TM1 */
00134      status = timer_disable(&app_timer1);
.....
```

13.8 定时器控制

通过定时器控制服务可以修改定时器的选项，将定时器更改为单次或者周期定时器，其接口原型为：

```
status_t timer_control(timer_t __p_* timer, int8u options);
```

表 13.8 给出定时器控制接口的参数说明。

表 13.8 接口 timer_control 参数说明

参 数	描 述
timer	指向定时器的指针
options	定时器选项

定时器控制选项如表 13.9 所示，可修改的选项包括单次和周期定时器。

表 13.9 接口选项参数说明

参 数	描 述
RS_OPT_REPEAT	周期定时器
RS_OPT_SINGLE	单次定时器

例子 APP_SAMPLE_13D 演示了修改定时器选项服务的使用方法：

【例 APP_SAMPLE_13D】

file: example\WIN32\app_sample_13d.cpp

```
.....
00061 /* 定时器 */
00062 timer_t app_timer1;
.....
00097      /**
00098      * 创建并启动一个名为 TM1 的周期定时器,
00099      * 定时器周期为 20 个系统时钟周期 */
.....
00116      status_t status;
.....
00119      /**
00120      * 修改定时器 TM1 选项为单次定时器 */
00121      status = timer_control(&app_timer1, RS_OPT_SINGLE);
.....
```

13.9 定时器内部结构

使用 timer_t 定义定时器，每个定时器包含一个定时器控制块（TMCB），用来保存运行时（Run-time）的状态信息。定义一个 timer_t 类型的定时器时，就创建了一个定时器控制块结构（TMCB）。表 13.10 列出了 TMCB 结构说明：

表 13.10 定时器控制块 TMCB 结构说明

项	描 述
name	定时器的名字
init_ticks	定时器周期
hand_ticks	剩余时间
entry	定时器到期函数
arg	到期函数参数

state	状态标志
init	初始化标志
next	指向下一个定时器结构

了解 TMCB 各项的意义，有助于理解定时器的内部机制，方便程序测试。注意，定时器控制块必须通过系统提供的服务来管理，不允许直接更改控制块各项的值，否则，将引起系统的异常。

13.10 总 结

本章首先介绍了硬件定时器和软件定时器的区别，接着详细介绍了 RS-RTOS 内建的定时器模块，然后讨论了与定时器相关的系统服务，并给出了定时器内部结构。

定时器包括单次定时器和周期定时器。单次定时器仅仅产生一次到期服务；而周期定时器以指定的周期重复执行到期服务。

使用定时器时，首先需要创建定时器，创建定时器时可以立即启动，或者在随后的程序中启动它。

定时器与系统时钟同步，定时器到期中断总是与系统时钟步调一致。而且，定时器的周期必须为系统时钟周期的整数倍，当不指定定时器周期时，默认为一个系统时钟周期。

由于定时器启动的时刻，通常不是恰好在系统时钟到达的时刻，所以，定时器第一个周期实际在 $T-1$ 到 T 之间。从第二个周期开始才能得到精确的定时器周期。

定时器的优先级比所有任务的优先级高，所以，定时器能够抢占当前运行的任务。

第 14 章

中 断

14.1 介 绍

为了获得对外部事件快速响应的能力，在发生紧急事件时，处理器可以立即暂停当前运行的程序，响应外部事件的服务，在处理完外部事件后，重新恢复到被暂停的程序中继续运行。这种能力称为处理器的中断机制。如图 14.1 表示发生中断时 CPU 控制转移的轨迹。

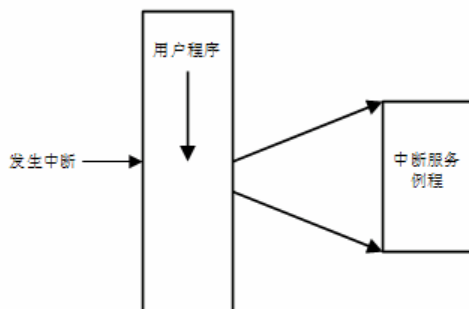


图 14.1 中断示意图

中断是实时嵌入式系统中最重要特征。毫不夸张地说，只有具备了中断处理能力的系统，才有资格称为实时系统。

14.2 多重中断

处理器的中断信号通常不止一个，多个中断可能同时出现。例如，一个程序正在从通讯线路上接收数据并打印结果。每当打印操作完成后，打印机就会产生一个中断。每当一个数据单位来到时，通讯线路控制器就会产生一个中断。数据单位可能是一个字符或者一个数据块，这取决于通讯规则。总之，在处理打印机中断的过程中有可能出现通讯中断。

处理多个中断的方法有顺序处理方式或者嵌套处理方式两种。

（1）顺序处理方式

当一个中断正在处理期间，暂时屏蔽其他的中断的响应，等到该中断处理完成，开放中断，由处理器查看有无尚未处理的中断。如果有，则依次响应中断的处理。这种方式为顺序处理方式，如图 14.2 所示。

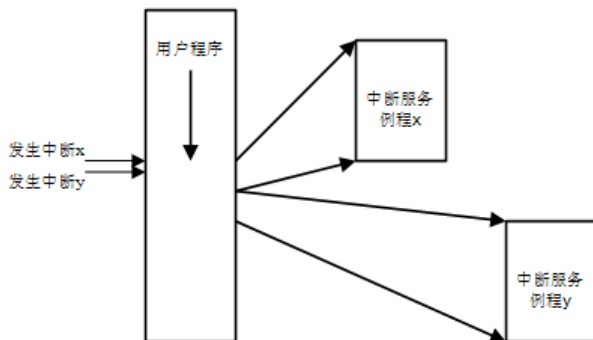


图 14.2 顺序中断处理

这种方式的缺点是没有考虑中断的相对优先级或者时间的紧迫程度。例如，输入数据从通讯线路上来到时，就需要迅速处理，腾出线路控制器的资源，供后面的输入使用。如果在后面的数据输入来到之前，前面的数据还未处理完，就会丢失数据。

（2）嵌套处理方式

这种方式对每种中断赋予不同的优先级，允许高优先级中断打断低优先级中断的处理程序，如图 14.3 所示。

这种方式使得高优先级的中断总能得到迅速的响应。但是嵌套中断也带来设计上的复杂，

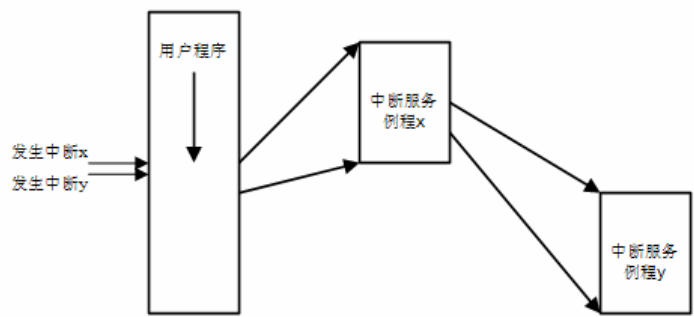


图14.3 嵌套中断处理

中断需要保存当前程序的上下文环境，过深的中断嵌套层数会消耗大量的栈空间。一旦出现栈溢出，系统崩溃将不可避免。

14.3 中断向量

引起中断的事件或发出中断请求的来源称为中断源。中断源向处理器提出的处理请求称为中断请求。发生中断时，被打断程序的暂停点称为断点。

对中断请求的整个处理过程是由硬件和软件结合起来而形成的一套中断机构实施的。发生中断时，处理器暂停执行当前的程序，转去处理中断。这个由硬件对中断请求做出反应的过程，称为中断响应。一般来说，处理器响应中断后依次执行下述动作：

- ① 中止当前程序的执行；
- ② 保存原程序断点的上下文信息（通用寄存器、程序寄存器 **PC** 和状态寄存器 **PS**）；
- ③ 修改程序计数器（**PC**）的值，跳转到相应的中断处理程序。

处理器的中断机制根据中断源信息，形成中断处理程序的入口地址，把它送入程序寄存器中 **PC**，并转入中断程序入口。通常，不同的中断有不同的入口。处理器接到中断后，就从中断控制器那里得到一个称为中断号的信息，通过它标识那个中断源产生了中断事件。它是检索中断向量表的索引，中断向量表的表项称为中断向量，存放中断服务程序的入口地址。

表 14.1 是 Intel Pentium 处理器中断向量表的设计情况。0~31 号中断是不可屏蔽的，用于对各种错误情况发出中断信号。32~255 号中断是可屏蔽的，用于表示设备产生的中断。

表 14.1 Intel Pentium 处理器中断向量表

中断号	描述	中断号	描述
0	除法错误	11	段不存在
1	调试异常	12	堆栈故障
2	空中断	13	一般性保护
3	断点	14	页面故障
4	INTO 检测溢出	15	(Intel 保留)
5	边界范围异常	16	浮点错误
6	无效操作码	17	调整检查
7	设备不可用	18	机器检查
8	双精度故障	19~31	(Intel 保留)
9	协调处理器段超限 (保留)	32~255	可屏蔽中断
10	无效任务状态段		

14.4 中断服务综述

附录 E 描述了与中断相关的各项服务接口的详细信息。包括函数的原型、简要介绍、参数、返回值、调用者、阻塞状态、注意事项。

表 14.2 包括系统提供的中断服务的接口列表。在本章后面的部分，将对这些服务进行详细的介绍，同时给出一些演示代码和图示。

表 14.2 中断服务接口

服 务	描 述
interrupt_attach	注册中断服务
interrupt_detach	注销中断服务
interrupt_enable	允许中断
interrupt_disable	禁止中断
interrupt_catch	捕获中断

14.5 注册中断服务

中断是处理器对外部事件的一种反应。通过注册中断服务，可以将中断服务程序和中断向量关联起来，当发生中断时，处理器暂停正在执行的程序，保留现场后执行相应的中断服务程序。注册中断服务的接口原型如下：

```
ientry_t interrupt_attach(vect_t vect, ientry_t ientry);
```

接口返回值是中断向量前一个中断服务入口，这是为了在需要时可以恢复到以前中断服务函数。表 14.3 描述了注册中断的传入参数。

表 14.3 接口 `interrupt_attach` 参数说明

参 数	描 述
<code>vect</code>	中断向量
<code>ientry</code>	中断服务入口

中断服务入口声明如下形式：

```
void app_i_handle_1(vect_t v)
```

中断服务函数没有返回值，入口参数 `vect_t` 为正在处理的中断向量号。通过中断向量号参数可以识别当前发生的中断源，这样，可以使用一个中断服务函数处理多个中断，而不至于产生混淆。需要注意，在中断服务中不能使用可能引起程序阻塞的系统服务，因为中断是不能被阻塞的。通过查看附录可以知道，那些服务可以在中断中使用，那些服务不可以。

例子 APP_SAMPLE_14A 演示了如何注册中断：

【例 APP_SAMPLE_14A】

file: example\WIN32\app_sample_14a.cpp

```
.....
00048 /* 中断向量 */
00049 #define APP_IRQ_1    1
```

```
00050
00051
00052 /* 中断服务例程 */
00053 void app_i_handle_1(vect_t v);
.....
00094      /* 注册 RS-RTOS 中断服务. */
00095      interrupt_attach(APP_IRQ_1, app_i_handle_1);
.....
```

14.6 注销中断服务

注销中断将对应中断向量与服务例程分离。那样，中断服务函数就不需要为对应的中断服务了。注销中断服务的接口原型如下：

```
ientry_t interrupt_detach(vect_t vect);
```

接口返回值是中断向量前一个中断服务入口，以便在需要时恢复到以前中断服务。表 14.4 描述了注销中断接口的传入参数。

表 14.4 接口 interrupt_detach 参数说明

参 数	描 述
vect	中断向量

例子 APP_SAMPLE_14B 演示了注销中断服务的使用方法：

【例 APP_SAMPLE_14B】

file: example\WIN32\app_sample_14b.cpp

```
.....
00051 /* 中断向量 */
00052 #define APP_IRQ_1          1
.....
```

```
00068 /* 中断服务例程 */
00069 void app_i_handle_1(vect_t v);
.....
00137      /**
00138      * 注销中断服务 */
00139      interrupt_detach(APP_IRQ_1);
.....
```

14.7 允许中断

注册中断服务后，默认中断服务是被禁止的。这时产生相应的中断，中断服务也不会被调用，只有允许中断后，才能响应中断服务。允许中断通过以下接口：

```
status_t interrupt_enable(vect_t vect);
```

表 14.5 描述了允许中断的传入参数。

表 14.5 接口 interrupt_enable 参数说明

参 数	描 述
vect	中断向量

例子 APP_SAMPLE_14A 和 APP_SAMPLE_14B 都包含了允许中断服务的使用，以下摘录了例子 APP_SAMPLE_14A 的代码片段：

【例 APP_SAMPLE_14A】

```
file: example\WIN32\app_sample_14b.cpp
```

```
.....
00048 /* 中断向量 */
00049 #define APP_IRQ_1    1
.....
00097      /* 允许中断 */
```

```
00098      interrupt_enable(APP_IRQ_1);
.....
```

14.8 禁止中断

如果禁止中断，相应的中断服务不会被调用。在需要中断服务，重新允许中断即可。禁止中断服务的接口原型如下：

```
status_t  interrupt_disable(vect_t vect);
```

表 14.6 给出了禁止中断的传入参数。

表 14.6 接口 `interrupt_disable` 参数说明

参 数	描 述
vect	中断向量

例子 APP_SAMPLE_14C 演示了禁止中断服务的使用：

【例 APP_SAMPLE_14C】

file: example\WIN32\app_sample_14c.cpp

```
.....
00051 /* 中断向量 */
00052 #define APP_IRQ_1          1
.....
00137      /**
00138      * 禁止中断 */
00139      interrupt_disable(APP_IRQ_1);
.....
```


14.9 捕获中断

如果关闭了中断，并在此过程发生了中断，中断处理机制将不会响应该中断，但是，中断信号已经被保留在中断源寄存器中。只要中断一旦被允许，将立刻得到响应。在中断被关闭的情况下，通过捕获中断服务，可以较为及时的响应已经发生的中断。这个过程如图 14.4 所示。

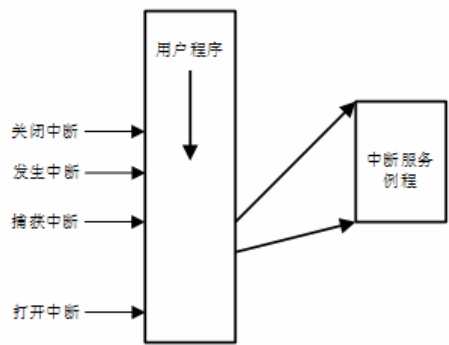


图14.4 捕获中断

捕获中断服务返回以后，将对应的中断重新恢复到禁止状态。因此，在中断允许的状态下，不应该使用该服务。捕获中断服务的接口原型如下：

```
status_t interrupt_catch(vect_t vect);;
```

表 14.7 是捕获中断的传入参数。

表 14.7 接口 interrupt_catch 参数说明

参 数	描 述
vect	中断向量

当应用系统在进行某项任务时，需要较长时间的禁止相关的中断服务。这样会降低系统对该中断响应的实时性，更重要的是，因为长时间的禁止中断，导致可能丢失重要的信息。通过捕获中断服务，可以在某些确定的时刻，响应发生的中断，提高中断的响应能力。

举个例子，任务 TA1 对一个数据链表进行清理工作，将过期的数据从链表中去除，这个任务不常发生。在任务 TA1 对链表清理的过程中，需要关闭中断 ISR1，因为该中断负责读

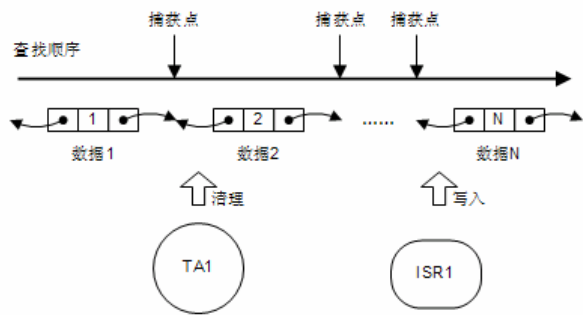


图14.5 中断捕获应用

入一条信息，并插入到链表中。如图 14.5，显然，任务完成所有数据清理的时间是不确定的，这依赖于数据链表的大小。如果任务在每个数据处理完成后，到下一个数据处理前，响应中断（此时，往链表插入数据是安全的）。这样，即确保了不会因为长时间关闭中断，导致数据丢失，又保证了数据链的完整性。

例子 APP_SAMPLE_14D 演示了禁止中断服务的使用：

【例 APP_SAMPLE_14D】

file: example\WIN32\app_sample_14d.cpp

```
.....
00051 /* 中断向量 */
00052 #define APP_IRQ_1          1
.....
00136      /**
00137      * 捕获中断 */
00138      interrupt_catch(APP_IRQ_1);
.....
```

捕获中断的特性依赖于硬件处理机制。有些处理器，禁止中断后，对应的中断源寄存器标志也会被禁止，并且中断源的触发是瞬间信号时（如，脉冲上升、下降沿）。这种情况下，

在禁止中断过程产生的中断信号是不能被保留下来的。那么，使用捕获中断就不能响应之前产生过的中断。

正如例子 APP_SAMPLE_14D 所表现一样，运行它没有任何输出，因为环境并没有模拟上述机制。

14.10 总 结

本章首先介绍了处理器的中断机制，指出中断能力是实时嵌入式系统重要的特征，介绍了中断向量的概念。然后讨论了 RS-RTOS 提供的与中断相关的服务接口。

使用中断前，首先需要注册相关的中断服务函数。中断服务函数要尽量短小精悍，将复杂的工作安排在任务中进行。

中断的优先级比任务要高，并且中断不能被阻塞。因此，在中断服务函数中，不能使用会引起阻塞的系统服务。

不同的中断之间有优先级的高低之分，高优先级的中断可以打断低优先级的中断服务，从而产生中断嵌套。中断嵌套层数最大不能超过 $2^{32} - 1$ （在 8 位处理器上是 255，16 位处理器上是 65535）。

第 15 章

内存管理

15.1 介绍

在嵌入式系统中，有一个必不可少的部件，就是存储器。存储器按照用途可分为主存储器和辅助存储器。主存储器（Main Memory 或者 Primary Memory）又称为内存储器，简称内存，是 CPU 能够直接存取指令和数据的存储器。硬盘、软盘等存储器，一般称为外存储器或辅助（Secondary Storage）。内存物理介质一般采用半导体存储技术，通常采用随机存储器（Random Access Memory, RAM）。

近年来，随着硬件技术和生产水平的发展，内存的成本迅速下降，容量不断扩大。在台式电脑中，1 兆字节（GB）大小的内存早已不是新鲜的事情。但随之而来的是日趋复杂的软件对存储空间急剧增长的需求。特别是在嵌入式领域，存储器资源依然是非常有限，往往由于设备体积和功耗的限制，要求尽量节省存储器资源。

因此，对内存的有效管理仍然是嵌入式系统中十分重要的问题。

15.2 内存碎片

使用内存的方式有很多种，按照内存的分配策略可分为静态分配和动态分配。静态分配是在编译期间就分配好所需内存，比如，前面所介绍的例子，使用数组来为任务栈分配一块

连续的内存空间。而动态分配是在程序运行过程中需要时才分配内存空间。

静态分配的优点是简单，但它通常不受欢迎，因为非常的不灵活——在运行中不能改变数组的大小，内存使用完毕空间不能释放。为了解决内存浪费的问题，系统负责管理一块连续的内存。程序开始不占有这些内存空间，在运行过程中，将空闲的内存划分为大小适应用程序需要的大小，并且内存使用完毕后，重新释放到空闲区中。这种技术称为动态内存分配。

使用动态内存分配技术，操作系统掌握一个表格，登记没格空闲区和已分配区，指出其大小、位置和对各个区的存取限制等。最初，全部空闲内存对应用程序是可用的，可划分为一大块。当程序需要内存时，系统就查表找到一个空闲区，它应该足以满足程序所需要的大小。如果大小恰好一样，则吧该区分给应用程序使用，在登记表中做响应的记录；如果这块空闲区比需要的还大，就将该区分成两部分：一部分给应用程序，另一部分就是剩下的较小的空闲区。当应用程序使用完内存，应该释放所占的区。系统要设法将它和相邻的空闲区合并起来，使它们成为一个连续的更大的空闲区。图 15.1 是内存分区的使用过程示例。

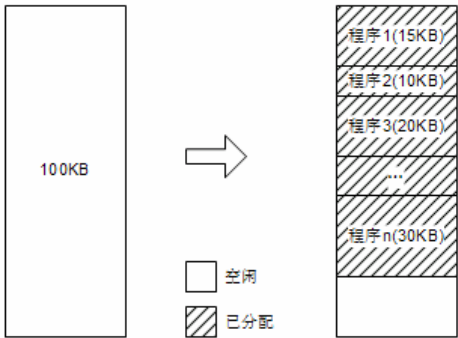


图15.1 动态内存分配

虽然动态分配技术的内存利用率高，但由于各个应用申请和释放内存的离散性，在内存中经常出现大量的分散的小空闲片区。如图 15.2 所示，由于应用分别申请大小为 5KB、20KB、15KB 和 30KB 内存空间，接着释放了大小为 5KB 和 15KB 的块。这时候，出现了 3 个空闲分区，它们大小分别是 5KB、15KB 和 25KB。三者总和是 45KB。如果此时应用需要分配 30KB 的内存空间。由于这三个空闲分区中任何一个均小于 30KB，因而应用无法得到所需大小的内存。内存中这种分散的、小容量、无法利用的小分区称作内存碎片。

内存碎片在嵌入式系统中是不可容忍的，因为在设备长期运行过程中，会有越来越多的内存碎片产生。这些碎片会慢慢蚕食掉有限的内存空间，最终导致系统崩溃。必须采取一种

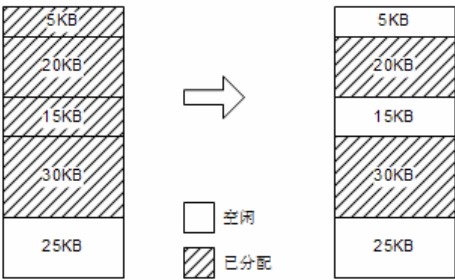


图15.2 内存碎片

策略避免内存碎片的产生，如 RS-RTOS 内核使用称为内存池集的技术来避免内存碎片的问题。

15.3 内存池集

在嵌入式系统中使用内存，最关键是要避免产生内存碎片，其次是内存分配的时间应尽量短。

有两种方法可以修正动态内存分配，使之不产生内存碎片：（1）是内存的分配和释放关联有序；（2）不允许内存以任意的尺寸分配，只能以少数几种固定尺寸的块来分配。

和应用的设计有关，实现内存分配和释放的顺序是不现实的。内存池的基本概念是实现内存以固定尺寸来分配。

内存池是由多个预先分配好的、固定大小的内存块组成。图 15.3 演示了一个尺寸为 1KB 的内存池。

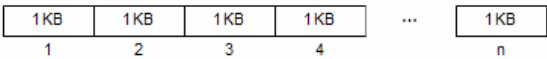


图15.3 内存池

设想一个系统，可以确定最坏的情况下所需的内存总数，以及所需内存的最大尺寸。将整个内存划分成若干块，每块的尺寸与请求的最大尺寸相同，那么可以保证只要有内存空闲，任何内存请求均可得到满足。

这个方案的最大的缺点是浪费一些内存。即使只需要一个字节的内存，也要分配最坏情况大小的内存块，浪费了块中的空间。如果系统需要的尺寸在 1 字节到最大尺寸之间随机均匀分布，内存池块全部分配后，会有一半的空间会被浪费掉。虽然，这是很浪费的，但是这种方法的优点仍是引人注目的，它永远不会因为内存的碎片化而崩溃。对于嵌入式系统，这点尤为重要。

为了减少这种浪费，名为内存池集的模式提供了内存池的有限集。如图 15.4 所示，它的基本策略是提供一组由不同尺寸的内存池组成的集合，每一个尺寸的内存池只为某一较小范围的内存需求服务。例如，当程序需要 3KB 大小的内存时，内存池集模式最先匹配到 4KB 尺寸的内存池。

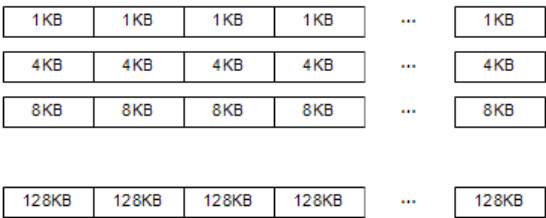


图15.4 内存池集

内存池集模式总以最小尺寸池作为匹配原则，即便命中的内存池中沒有空闲的内存块，也不会继续匹配更大尺寸的内存池。如果 4KB 内存池中沒有空闲块，也不会用 8KB 的空闲块满足大小为 3KB 的内存需求。这一规则避免产生隐形内存的问题：大尺寸的内存块被多数小尺寸需求占用，而真正的需求却得不到满足。

内存池集模式能够完美的避免内存碎片和内存浪费，并且具有非常优异的分配速度。但需要设计时更多的分析，要求能准确的统计应用系统对内存需求的尺寸分部和数量。

15.4 内存管理服务综述

附录 M 描述了与内存管理相关的各项服务接口的详细信息。包括函数的原型、简要介绍、参数、返回值、调用者、阻塞状态、注意事项。

表 15.1 包括系统提供的内存管理服务的接口列表。在本章后面的部分，将对这些服务进行详细的介绍，同时给出一些演示代码和图示。

表 15.1 内存管理服务接口

服 务	描 述
mpool_alloc	申请内存池
mpool_free	释放内存池
mpool_info	获取内存池信息

15.5 申请内存池

应用需要使用内存时，通过向内存池集提出申请服务。如果内存池中有合适的空闲内存块，则返回指向空闲内存首地址的指针，否则返回空指针（**NULL**）。申请内存池服务的接口原型如下：

```
void __p_* mpool_alloc(mmsz_t size);
```

该接口需要提供一个内存尺寸参数，用来指明所需分配的内存的字节数。表 15.2 描述了申请内存池接口的传入参数。

表 15.2 接口 mpool_alloc 参数说明

参 数	描 述
size	申请内存的尺寸，单位字节（bytes）

例子 APP_SAMPLE_15A 演示了如何使用申请内存池服务：

【例 APP_SAMPLE_15A】

file: example\WIN32\app_sample_15a.cpp

```
.....
00058      char* p;
.....
00064      /**
00065      * 申请大小为 100 字节得内存空间 */
```

```
00066      p = (char*)mpool_alloc(100);
.....
```

正如例子所示，当内存池没有合适的空闲内存块时，申请内存池服务返回一个空指针，因此，使用该服务时，判定是否返回有效内存地址十分必要。如果指定的内存尺寸参数为 0，则服务将试图查找内存池中最小的空闲块——申请一个尺寸为 0 的内存令人费解，而且没有实际意义，之所以这样做，仅仅是简化了设计。

15.6 释放内存池

使用完通过申请内存池服务获得的内存时，必须及时释放它，其他的程序才可以再次使用该内存。释放内存池服务的接口原型是：

```
void mpool_free(void __p_* p);
```

该接口指定内存地址为传入参数，表 15.3 描述了释放内存池接口的传入参数。

表 15.3 接口 mpool_free 参数说明

参 数	描 述
p	指向待释放的内存地址

例子 APP_SAMPLE_15A 演示了如何使用释放内存池服务：

【例 APP_SAMPLE_15A】

file: example\WIN32\app_sample_15a.cpp

```
.....
00058      char* p;
.....
00064      /**
00065      * 申请大小为 100 字节得内存空间 */
00066      p = (char*)mpool_alloc(100);
00067
```

```
00068      /* 内存申请是否成功 */
00069      if (p != NULL) {
00070
00071          /**
00072           * 内存使用... */
00073          .....
00076          /* 释放内存 */
00077          mpool_free(p);
00078
00079      } else {
00080          .....
00086      }
00087      .....
```

服务的传入参数必须指向合法的内存地址，必须是通过申请内存池服务 `mpool_alloc` 获得的有效内存。否则，将引起内部断言错误，甚至导致内存冲突，使系统崩溃。

15.7 获取内存池信息

通过获取内存池信息服务，可以获得内存池集运行时（Run-time）的状态信息。得到的信息包括内存池尺寸，内存块数，空闲块数目，池链表的头指针。获取内存池信息服务的接口原型是：

```
status_t mpool_info(mpinfo_t __out* info);
```

该接口传入一个 `mpinfo_t` 结构（`mpinfo_t` 是 `memory pool information` 的缩写）指针参数，用来存放服务接口返回的各信息值，表 15.4 给出 `mpool_info` 服务接口参数详细说明：

表 15.4 接口 `mpool_info` 参数说明

参 数		说 明
info	index	内存池的序号索引

	size	返回内存池尺寸
	blocks	内存块数目
	frees	空闲内存块数目
	pfree	指向空闲内存块表头的指针

例子 APP_SAMPLE_15B 演示了如何获取内存池信息服务：

【例 APP_SAMPLE_15B】

file: example\WIN32\app_sample_15b.cpp

```
.....
00094     status_t status;
00095     mpinfo_t info;
.....
00098     /* 内存池索引 */
00099     info.index = 0;
.....
00104     for (;;) {
00105
00106         /* 获取内存池信息. */
00107         status = mpool_info(&info);
00108
00109         if (status != RS_EOK)
00110             break;
00111
00112         /* 显示内存信息 */
00113         printf("%5d%8d%10d%8d%9d      %#010p\n",
00114             info.index,
00115             info.size,
00116             info.blocks,
00117             info.blocks - info.frees,
00118             info.frees,
00119             info.pfree);
```

```
00120
00121     info.index++;
00122 }
.....
```

15.8 内存池集内部结构

RS-RTOS 使用内存集作为首选内存管理策略。内存池中的块以单向链表组织，每个内存块头部包含一个指针，用来连接下一个内存块起始地址。因此，实际每块内存能够使用的空

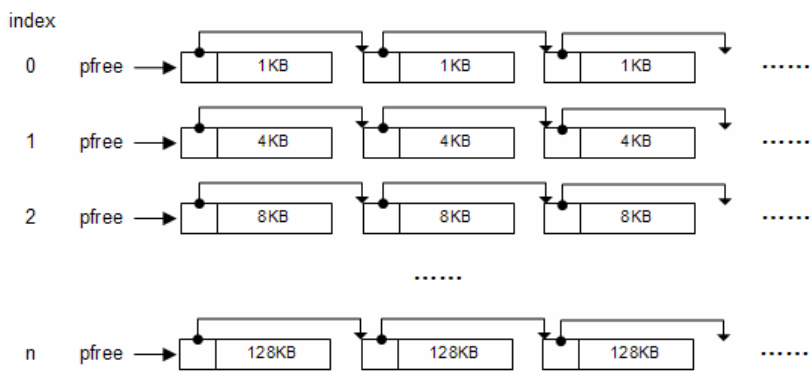


图15.5 RS-RTOS 内存池集

间大小是块的尺寸减去一个指针的大小，公式如下：

块最大使用空间 = 块尺寸 - 指针尺寸

例如，设计满足最大 1KB 的内存需求，需要配置的内存池尺寸为 1KB + sizeof(void*)，在 32 位系统上指针大小占 4 个字节，则应该配置内存池块的大小为：

1024 字节 + 4 字节 = 1028 字节

15.9 总 结

本章首先介绍了与内存池相关的几种动态内存管理策略，讨论了内存碎片的产生过程。然后重点讨论了内存池集的管理模式，最后对 RS-RTOS 内存管理相关的服务做了分析。

内存管理是嵌入式系统重要的组成部分。内存管理分为静态内存分配和动态内存分配策略。静态分配是在编译期间就分配好所需内存；而动态分配是在程序运行过程中需要时才分配内存空间。

内存碎片是内存中分散的、小容量、无法利用的小分区。内存碎片是动态内存分配和释放的离散性和内存分片的随意性共同作用的结果。内存碎片的产生会导致虽然有足够的内存空间，却无法得到一整块可用内存。随着设备长时间运行，这一现象会加剧，最终导致系统因内存耗尽而崩溃。

内存池集可以完全避免产生内存碎片，并且具有非常优异的分配速度。最大的缺点是需要对内存池尺寸和数量做精心的设计，对设计人员要求比较高。如果配置不当，潜在的代价是内存块利用率过低和内存分配失败。

内存池集优劣分明，尤其是避免碎片和极为优异的查找速度，是嵌入式系统不二的内存管理策略。但在使用内存池集时，依然要严格遵循申请——使用——释放的顺序。

第 16 章

设备管理

16.1 介绍

嵌入式系统需要用到各种各样的设备，其种类繁多，特性各异。常见的有打印机、鼠标、硬盘驱动器、网络驱动器等。这些设备各有不同的物理特性，操作方式各异，因此，很难按照一种算法统一的进行管理。设备管理是嵌入式系统中比较繁琐和复杂的一部分。

设备与硬件紧密相关，也是操作系统直接管理的资源。在大部分应用中，对设备操作往往不只一个任务，从而引起多个任务对设备的竞争。为了使系统有条不紊的工作，系统必须合理的分配设备，并对设备操作进行统一的调度。

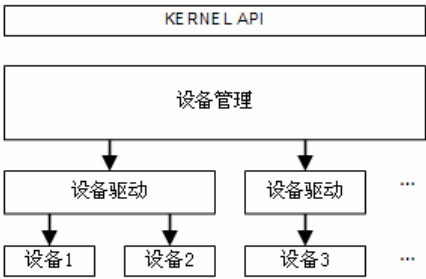


图 16.1 RS-RTOS 设备管理

设备管理的关键目标是：为应用提供一个统一的、简单的接口，方便使用；其次，对设备的操作进行统一的调度，避免设备操作的冲突。

为了能够操作各种的设备，并实现统一的应用接口，需要对设备的物理特性进行软件封装，实现这些封装的程序就是设备的驱动程序，设备驱动是物理设备的第一层软件。图 16.1 演示了 RS-RTOS 设备管理的层次结构。

16.2 设备驱动

设备驱动程序是控制设备动作（如设备的打开、关闭、读和写等）的核心模块，用来控制设备上的数据传输。一般来说，设备驱动程序的功能包括以下部分：

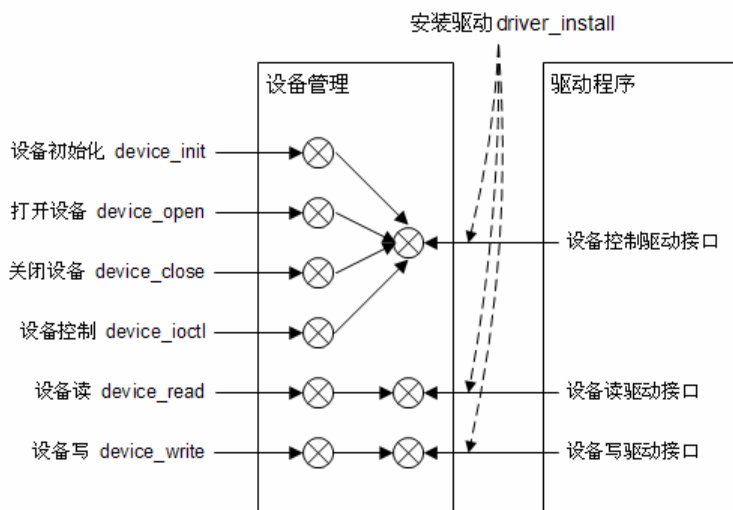


图16.2 RS-RTOS 安装驱动

- 1) 接收来自上层、与设备无关的抽象读写请求，并且检查请求的合法性。
- 2) 管理来自上层的并发请求，采用合适的算法顺序处理它们。
- 3) 将服务请求转化为合适的设备控制指令，启动设备工作，完成特定的 IO 操作。
- 4) 处理来自设备的中断。

通常，设备驱动程序与设备类型是一一对应的，即系统可由一个磁盘驱动程序控制所有

的磁盘机，一个终端驱动程序控制所有的终端。因为这些相同类型的设备有相同的命令控制序列。这样，一个设备驱动程序可以控制同一类型的多个物理设备。驱动程序能够区别它所控制的多台设备。为了管理方便，常采用主、从设备号方式。主设备号（Major）表示设备类型，而从设备号（Minor）表示该类型的一个设备。利用从设备号可以把一类设备中的多台设备互相区别开。

16.3 设备管理服务综述

附录 N 描述了与设备管理相关的各项服务接口的详细信息。包括函数的原型、简要介绍、参数、返回值、调用者、阻塞状态、注意事项。

表 16.1 包括系统提供的设备管理服务的接口列表。在本章后面的部分，将对这些服务进行详细的介绍，同时给出一些演示代码和图示。

表 16.1 设备管理服务接口

服 务	描 述
build_device	创建设备标识
driver_install	安装设备驱动
driver_uninstall	卸载设备驱动
device_init	设备初始化
device_open	打开设备
device_close	关闭设备
device_write	设备读
device_read	设备写
device_ioctl	设备控制

16.4 创建设备标识

系统用主设备号（Major）从设备号（Minor）来唯一标识一个设备。相同主设备号表示

同一类设备，例如都是硬盘；从设备号标识同类设备的序号。通过接口：

```
device_t build_device(int16u major, int16u minor);
```

创建设备的唯一标识，服务根据设备的主设备号和从设备号，返回一个类型为 `device_t` 的标识。`device_t` 是一个宽为 32 位的整型结构，高 16 位存放了主设备号，低 16 存放从设备号。通过这个标识就能识别系统中唯一的设备。表 16.2 描述了创建设备标识接口的传入参数。

表 16.2 接口 `build_device` 参数说明

参 数	描 述
major	主设备号
minor	从设备号

16.5 安装设备驱动

系统通过驱动程序操作具体的设备。每个设备都对应一个驱动程序，但一个驱动程序可以驱动不只一个设备。而拥有相同主设备号的设备使用同一个驱动程序，因此，驱动是针对主设备的，与从设备号无关。在设备使用之前必须安装与之对应的驱动程序，通常在系统初始化的时候就安装它。服务接口的原型如下：

```
status_t driver_install(int16u major, driver_table_t __p_* table);
```

表 16.3 描述了安装设备驱动接口的传入参数。

表 16.3 接口 `driver_install` 参数说明

参 数		说 明
major		主设备号，指定为 0 表示由系统分配
table	read	设备读驱动接口
	write	设备写驱动接口
	ioctl	设备控制驱动接口

参数 **major** 指定待注册的主设备号，如果该编号不指定（为 0），则由设备管理器分配一个空闲的主设备号，接口返回最后成功注册的主设备号（大于或者等于 0）或者操作失败代码（小于 0）；参数 **table** 是驱动的接口表，它是一个结构，包括设备驱动程序必须实现的三个接口：设备读接口、设备写接口和设备控制接口。所有应用对设备的操作均通过这三个接口实现（请参考图 16.2）。

1) 设备读驱动接口

设备读驱动接口声明为如下形式：

```
int  drv_device1_read(int16u minor, char __out_* buff, int size);
```

其中，第一个参数 **minor** 是从设备号；第二个参数 **buff** 用来存放从设备读入的数据的缓冲区；第三个参数 **size** 是缓冲区 **buff** 的大小。函数的返回值是成功读入的字节数，返回负值表示操作失败。

2) 设备写驱动接口

设备写驱动接口声明为如下形式：

```
int  drv_device1_write(int16u minor, const char __p_* buff, int size);
```

其中，第一个参数 **minor** 是从设备号；第二个参数 **buff** 用来存放写数据的缓冲区；第三个参数 **size** 是缓冲区 **buff** 的大小。函数的返回值是成功写入的字节数，如果返回负值表示操作失败。

3) 设备操作驱动接口

设备操作驱动接口声明为如下形式：

```
int  drv_device1_ioctl(int16u minor, int16u operate, arg_t arg);
```

其中，第一个参数 **minor** 是从设备号；第二个参数 **operate** 是设备控制选项，指示设备操作项目，这些项分别与设备管理其中 4 个服务接口一一对应，如表 16.4 所示；最后一个参数 **arg** 为操作的传入参数，完全由驱动开发人员定义，系统不对该参数做任何操作。

表 16.4 设备操作驱动接口 operate 选项

选 项	描 述
RS_DEVICE_INIT	设备初始化操作
RS_DEVICE_OPEN	打开设备操作
RS_DEVICE_CLOSE	关闭设备操作
RS_DEVICE_IOCTL	设备控制操作

一旦定义好以上几个驱动接口，就可以通过安装设备驱动服务，将驱动注册到系统的设备管理中。例子 APP_SAMPLE_16A 演示了如何安装设备驱动程序：

【例 APP_SAMPLE_16A】

file: example\WIN32\app_sample_16a.cpp

```
.....
00048 /* 主设备号 */
00049 #define DRV_DEVICE1          1
00050
00051
00052 /* 设备驱动接口 */
00053 int drv_device1_read(int16u minor, char* buff, int size);
00054 int drv_device1_write(int16u minor, const char* buff, int size);
00055 int drv_device1_ioctl(int16u minor, int16u operate, arg_t arg);
.....
00063 /* 应用初始化 */
00064 void application_initialize(void)
00065 {
00066     status_t status;
00067     driver_table_t drv_tbl;
00068
00069     /* 打印关于示例一些信息 */
00070     APP_SAMPLE_INFO;
00071
00072     /* 取得驱动接口 */
00073     drv_tbl.read = drv_device1_read;
```

```
00074     drv_tbl.write = drv_device1_write;
00075     drv_tbl.ioctl = drv_device1_ioctl;
00076
00077     /* 安装设备的驱动 */
00078     status = driver_install(DRV_DEVICE1, &drv_tbl);
00079
00080     /* 返回值相等表示注册成功 */
00081     ASSERT(status == DRV_DEVICE1);
00082 }
.....
```

16.6 卸载设备驱动

热插拔（Hot Plugging）允许用户在不关闭系统的情况，对设备进行取出或者更换。相应的设备驱动程序应该从系统中卸载，以免错误的操作一个不存在的硬件设备。最直接的方法就是通过卸载设备驱动服务，将设备驱动从设备管理器中卸载。其服务接口原型如下：

```
status_t driver_uninstall(int16u major);
```

卸载设备驱动只需传入对应的主设备号，如表 16.5 所示。

表 16.5 接口 driver_install 参数说明

参 数	说 明
major	主设备号

例子 APP_SAMPLE_16B 演示了卸载设备驱动服务的使用方法：

【例 APP_SAMPLE_16B】

file: example\WIN32\app_sample_16b.cpp

```
.....
00051 /* 主设备号 */
```

```
00052 #define DRV_DEVICE1          1
.....
00113     status_t status;
00114
00115     /* 卸载设备驱动 */
00116     status = driver_uninstall(DRV_DEVICE1);
.....
```

16.7 设备初始化

大多设备在使用之前都要进行初始化操作。设备的初始化工作一般在系统初始化或者设备加载的时候进行。对设备进行初始化的服务接口原型如下：

```
int  device_init(device_t device, arg_t arg);
```

表 16.6 描述了设备初始化接口的传入参数。

表 16.6 接口 **device_init** 参数说明

参 数	说 明
device	设备标识
arg	操作参数

例子 APP_SAMPLE_16C 演示了设备初始化服务的使用方法：

【例 APP_SAMPLE_16C】

file: example\WIN32\app_sample_16c.cpp

```
.....
00052 /* 主设备号 */
00053 #define DRV_DEVICE1          1
.....
00114     int status;
```

```
00115     device_t device;
00116
00117
00118     /* 建立设备标识 */
00119     device = build_device(DRV_DEVICE1, 0);
00120
00121     /* 初始化设备 */
00122     status = device_init(device, (arg_t)0);
.....
```

16.8 打开设备

在对设备进行读写的时候，首先要获得设备的操作权限。通过打开设备获得对设备的操作权限，特别是独占类（同一时刻只能有一个使用者，如打印机）的设备，打开设备可以防止多个使用者进行设备读写冲突。打开设备服务的接口原型如下：

```
int  device_open(device_t device, arg_t arg);
```

表 16.7 描述了打开设备接口的传入参数。

表 16.7 接口 **device_open** 参数说明

参 数	说 明
device	设备标识
arg	操作参数

例子 APP_SAMPLE_16D 演示了打开设备服务的使用方法：

【例 APP_SAMPLE_16D】

file: example\WIN32\app_sample_16d.cpp

```
.....
00052 /* 主设备号 */
```

```
00053 #define DRV_DEVICE1          1
.....
00114     int status;
00115     device_t device;
00116
00117
00118     /* 建立设备标识 */
00119     device = build_device(DRV_DEVICE1, 0);
.....
00124     /* 打开设备 */
00125     status = device_open(device, (arg_t)0);
.....
```

16.9 关闭设备

对设备读写完成后，要释放设备的操作权限。通过关闭设备服务完成这一任务，其接口原型如下：

```
int device_close(device_t device, arg_t arg);
```

关闭设备服务必须与打开设备服务（`device_open`）成对使用。表 16.8 描述了关闭设备接口的传入参数。

表 16.8 接口 `device_close` 参数说明

参 数	说 明
device	设备标识
arg	操作参数

例子 APP_SAMPLE_16D 演示了打开设备服务的使用方法：

【例 APP_SAMPLE_16D】

file: example\WIN32\app_sample_16d.cpp

```
.....
00052 /* 主设备号 */
00053 #define DRV_DEVICE1          1
.....
00114     int status;
00115     device_t device;
00116
00117
00118     /* 建立设备标识 */
00119     device = build_device(DRV_DEVICE1, 0);
.....
00127     /* 关闭设备 */
00128     status = device_close(device);
.....
```

16.10 设备读

设备读写服务是设备与系统缓冲区互交数据的主要手段，可以提供大容量的数据交换服务。设备读服务负责从设备中读入数据到指定的缓冲区中。其接口原型如下：

```
int  device_read(device_t device, char __out_* buff, int size);
```

接口返回值是从设备中成功读入数据的大小（大于或等于 0），或者返回操作失败代码（小于 0），表 16.9 描述了设备读接口的传入参数。

表 16.9 接口 device_read 参数说明

参 数	说 明
device	设备标识
buff	存放读入数据的缓冲区

size	缓冲区尺寸
------	-------

例子 APP_SAMPLE_16E 演示了设备读服务的使用方法：

【例 APP_SAMPLE_16E】

file: example\WIN32\app_sample_16e.cpp

```
.....
00052 /* 主设备号 */
00053 #define DRV_DEVICE1      1
.....
00111 /* 任务 TA1 入口 */
00112 void app_task_1(arg_t)
00113 {
00114     int status;
00115     device_t device;
00116
00117
00118     /* 建立设备标识 */
00119     device = build_device(DRV_DEVICE1, 0);
00120
00121     /* 初始化设备 */
00122     status = device_init(device, (arg_t)0);
00123
00124     for (;;)
00125     {
00126         int bytes;
00127         char buff[100];
00128
00129         /* 打开设备 */
00130         status = device_open(device, (arg_t)0);
00131
00132         /* 从设备读入数据 */
```

```
00133         bytes = device_read(device, buff, 100);
00134
00135         if (bytes > 0)
00136             printk("TA1: read from device (%s)\n", buff);
00137
00138         /* 关闭设备 */
00139         status = device_close(device);
00140
00141         task_sleep(RS_TICK_FREQ);
00142     }
00143 }
.....
```

16.11 设备写

设备写服务负责将缓冲区数据写到指定的设备。与设备读服务一样，设备写服务提供了大容量数据的写入操作。其接口原型如下：

```
int device_write(device_t device, const char __p_* buff, int size);
```

接口返回值是成功写入设备的数据的大小（大于或等于 0），或者返回操作失败代码（小于 0），表 16.10 描述了设备写接口的传入参数。

表 16.10 接口 device_write 参数说明

参 数	说 明
device	设备标识
buff	输出数据的缓冲区
size	缓冲区尺寸

例子 APP_SAMPLE_16F 演示了设备写服务的使用方法：

【例 APP_SAMPLE_16F】

file: example\WIN32\app_sample_16f.cpp

```
.....
00052 /* 主设备号 */
00053 #define DRV_DEVICE1          1
.....
00111 /* 任务 TA1 入口 */
00112 void app_task_1(arg_t)
00113 {
00114     int status;
00115     device_t device;
00116
00117
00118     /* 建立设备标识 */
00119     device = build_device(DRV_DEVICE1, 0);
00120
00121     /* 初始化设备 */
00122     status = device_init(device, (arg_t)0);
00123
00124     for (;;)
00125     {
00126         int bytes;
00127         char buff[100];
00128
00129         /* 待写数据 */
00130         sprintf(buff, "Hello world!");
00131
00132         /* 数据长度 */
00133         bytes = strlen(buff);
00134
00135         /* 打开设备 */
```

```
00136         status = device_open(device, (arg_t)0);
00137
00138         /* 设备写 */
00139         device_write(device, buff, bytes);
00140
00141         /* 关闭设备 */
00142         status = device_close(device);
00143
00144         task_sleep(RS_TICK_FREQ);
00145     }
00146 }
.....
```

16.12 设备控制

设备控制提供对设备状态和属性操作的通道。通过设备控制，可以读入设备相关状态信息，或者设定设备的属性，工作模式，控制选项等。除了设备的初始化、打开、关闭和设备读写。其他操作都在设备控制服务中完成。设备控制服务的接口原型如下：

```
int device_ioctl(device_t device, arg_t arg);
```

表 16.11 描述了设备控制接口的传入参数。

表 16.11 接口 device_ioctl 参数说明

参 数	说 明
device	设备标识
arg	操作参数

例子 APP_SAMPLE_16G 演示了设备控制服务的使用方法：

【例 APP_SAMPLE_16G】

file: example\WIN32\app_sample_16g.cpp

```
.....
00052 /* 主设备号 */
00053 #define DRV_DEVICE1          1
.....
00114     int status;
00115     device_t device;
00116
00117
00118     /* 建立设备标识 */
00119     device = build_device(DRV_DEVICE1, 0);
00120
00121     /* 初始化设备 */
00122     status = device_init(device, (arg_t)0);
00123
00124     /* 更改设备控制项 */
00125     status = device_ioctl(device, (arg_t)0);
.....
```

16.13 总 结

本章首先介绍了设备以及设备驱动的概念，然后介绍了 RS-RTOS 设备管理与设备驱动的层次结构，最后讨论了与设备管理相关的各项服务。

嵌入式系统用到设备种类繁多，特性各异，很难按照一种方法统一的进行管理。设备管理是嵌入式系统中较为繁琐和复杂的部分。

设备管理的主要目标是为应用提供一个统一、简单、方便使用接口，其次，对设备的操作进行统一的调度，避免多任务对设备操作的冲突。

应用对设备的标准操作有六个，分别是：设备初始化、打开设备、关闭设备、设备读、设备写、设备控制。在对设备操作前，必须首先安装对应的设备驱动程序。

设备通过主设备号和从设备号进行唯一标识。主设备号表示同一类设备，拥有相同的驱动接口。在驱动服务接口中，使用从设备号来区别应用操作的具体设备。

设备从系统中移除后，通过卸载不再使用的设备驱动，可以节省系统资源，也避免了错误的操作一个不存在的硬件设备。

附录

RS-KERNEL API 服务接口

附录包括了 RS-RTOS 内核模块——RS-KERNEL 所有服务接口的参考手册。描述了各项服务接口的详细信息。包括函数的原型、简要介绍、参数、返回值、调用者、阻塞状态、注意事项。

附录 A~N 分别描述了 RS-KERNEL 各组件提供的接口，它们都遵循以下格式：

- 原型 —— 服务接口的 C 语言标准原型，由接口名称、参数、数据类型、返回值类型组成；
- 描述 —— 关于服务的简单介绍；
- 参数 —— 服务接口的输入参数，有些参数是一个预先分配好空间的结构，用来存放服务返回数据；
- 返回值 —— 服务接口返回的状态和数据；
- 调用者 —— 合法调用该服务的程序类型；
- 阻塞 —— 是否会引起调用者进入阻塞状态；
- 其他 —— 其他的注意事项。

附录 A 初始化服务

A.1 hardware_initialize

硬件初始化。

原 型

```
void hardware_initialize(void);
```

描 述

应用程序必须实现该接口，该接口在 RS-RTOS 内核初始化之前被调用，用来完成一些特定的硬件初始化工作，如处理器外围功能寄存器的设置，硬件时钟设定等。

参 数

无。

返回值

无。

调用者

被内核调用。

阻 塞

无。

其 他

不能在应用程序中直接调用该接口。不能在该接口中调用 RS-KERNEL API 服务接口。

A.2 application_initialize

应用初始化。

原 型

```
void application_initialize(void);
```

描 述

应用的入口函数，用来初始化应用程序。如创建任务，创建信号量、队列，注册中断等。在 `application_initialize` 中创建的任务不会被马上启动，内核在完成 `application_initialize` 后，才真正进入多任务环境，这时优先级最高的就绪任务获得运行机会。

参 数

无。

返回值

无。

调用者

被内核调用。

阻 塞

无。

其 他

不能在应用程序中直接调用该接口。

附录 B 基础服务

B.1 build_name

建立内核对象名字。

原 型

```
name_t build_name(char c1, char c2, char c3, char c4);
```

描 述

创建一个内核对象名字，内核对象名字由英文字母(a~z, A~Z)、数字(0~9)以及下划线(_)组成，并且以字母或下划线开头，名字长度不能超过四个字符。在 RS-RTOS 系统，名字只作为描述方便，不是内核对象唯一标识，并允许有两个名字相同的内核对象。

参 数

- c1 —— 内核名字第一个字符。
- c2 —— 内核名字第二个字符。
- c3 —— 内核名字第三个字符。
- c4 —— 内核名字第四个字符。

返回值

内核对象名字。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

B.2 kernel_version

获取内核版本号。

原 型

```
int16u kernel_version(void);
```

描 述

返回 RS-RTOS 内核版本号（也就是 RS-KERNEL 版本号），返回值是一个两字节的 16 进制整数，高字节表示主版本号，低字节表示次版本号，如 0x0122 表示版本号为 V1.22。

参 数

无。

返回值

两字节的 16 进制整数，高字节表示主版本号，低字节表示次版本号。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

B.3 kernel_version_s

获取内核版本号的字符串。

原型

```
char __p_* kernel_version_s(void);
```

描述

返回 RS-RTOS 内核版本号（也就是 RS-KERNEL 版本号）的字符串形式，如在内核版本 V1.22 中调用该接口将返回字符串“1.22”。

参数

无。

返回值

返回以终止符'\0'结尾的 C 语言字符串。

调用者

应用初始化、中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

无。

B.4 kernel_lock

锁定内核。

原 型

```
void kernel_lock(void);
```

描 述

锁定内核将禁止调度器进行任务调度算法，禁止一切任务切换操作。但不影响中断的响应与定时器的响应服务。

参 数

无。

返回值

无。

调用者

中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

B.5 kernel_unlock

解除内核锁。

原 型

```
void kernel_unlock(void);
```

描 述

解除内核的锁定状态，允许内核调度器重新进行任务调度。`kernel_lock` 与 `kernel_unlock` 要严格成对使用。

参 数

无。

返回值

无。

调用者

中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

内核使用一个引用计数器计算内核被锁定的次数，该计数器的宽度与处理器的位宽一致，在 8 位处理器上最大值为 255，16 位处理器上最大值是 65535，32 位处理器上最大值是 $2^{32} - 1$ 。这允许内核锁被嵌套使用，嵌套的最大深度不能超过以上值。

另外，在某些情况下，内核也会进行自动锁定（如在中断服务），所以实际应用中，使用嵌套层数应该有所保留。

B.6 kernel_islock

返回内核锁的状态。

原 型

```
uint kernel_islock(void);
```

描 述

返回内核锁的状态。返回值类型 `uint` 是与处理器位宽一致的无符号整型，当返回值为 0 时，表示内核未锁定状态，如果返回值大于 0，则表示内核在锁定状态，其值表示了内核被锁定的引用计数。

参 数

无。

返回值

0	—— 内核未锁定
大于 0	—— 内核锁定状态

调用者

中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

B.7 kernel_info

获取内核信息。

原型

```
status_t kernel_info(kinfo_t __p_* info);
```

描述

获取内核的运行时（Run-time）的状态信息。获取的信息包括系统当前 CPU 利用率、任务使用情况。

参数

info —— 存储返回信息的结构指针。

返回值

RS_EOK —— 操作成功，info 保存返回的内核信息。

调用者

中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

无。

附录 C 任务服务

C.1 task_create

创建一个任务。

原 型

```
status_t task_create(prio_t prio, name_t name, entry_t entry, arg_t arg,
                    sp_t stack_base, mmsz_t stack_size, int8u options);
```

描 述

该服务创建一个任务，被创建的任务从指定的入口函数开始执行，可以通过参数指定任务的入口参数、优先级别、任务栈等，另外可以通过选项参数决定任务被创建后是否立即执行，还是处于挂起状态，手动恢复运行。

参 数

- prio — 任务的优先级，范围是 0~254，0 为最高优先级。
- name — 任务的名字，指定为 0 则不使用名字。
- entry — 任务入口函数。
- arg — 任务入口函数的参数。
- stack_base — 任务栈的起始地址，指定为 NULL，则由系统动态创建栈空间。
- stack_size — 任务栈大小（字节）。
- options — 任务选项：
 - 1) RS_OPT_SUSPEND 创建一个挂起的任务，通过手动恢复运行。
 - 2) RS_OPT_KFREE 任务被删除时自动释放任务栈；如果指定该选项，stack_base 传入地址必须由动态内存分配 mpool_alloc 获得；如果 stack_base 指定为 NULL，则无需指定该选项。

返回值

RS_EOUTRANGE	—— 任务优先级超限。
RS_EEXIST	—— 相同优先级的任务已经存在。
RS_ENULL	—— 任务栈自动分配失败。
RS_EOK	—— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

无。

C.2 task_delete

删除一个任务。

原型

```
status_t task_delete(prio_t prio);
```

描述

调用该服务将删除指定的任务。任务可以调用 **task_delete** 删除自身。如果任务返回，任务将隐式的调用该服务删除自身。

参数

prio —— 待删除的任务的优先级，指定为 **task_self()**删除自身任务。

返回值

- RS_EOUTRANGE — 指定优先级超限。
- RS_ENOTEXIST — 指定任务不存在。
- RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

删除一个任务时请确保已经释放该任务申请的资源。

C.3 task_change_prio

更改任务优先级。

原 型

```
status_t task_change_prio(prio_t prio, prio_t new_prio);
```

描 述

该服务改变指定任务的优先级别，任务合法优先级范围是 0~254（包括 0 和 254），值越低代表任务的优先级别越高。任务优先级的更改是永久的，这意味着任务将以新的优先级运行和控制，除非再次更改任务的优先级。

参 数

- prio — 任务优先级。
- new_prio — 新的任务优先级。

返回值

RS_EOUTRANGE	—— 指定参数错误，如优先级超出范围。
RS_ENOTEXIST	—— 指定任务不存在。
RS_EEXIST	—— 新的任务优先级已经被占用。
RS_ELOCK	—— 任务被锁定，如任务拥有互斥量时会被暂时锁定。
RS_EOK	—— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

如果是临时性的更改任务的优先级，更好的方法是更改任务的运行优先级。

C.4 task_change_runprio

更改任务运行优先级。

原型

```
status_t task_change_runprio(prio_t prio, prio_t run_prio);
```

描述

该服务改变指定任务的运行优先级别。任务运行优先级是任务实际运行的优先级别，其合法范围是 0~254（包括 0 和 254）。与更改任务优先级不同，更改任务的运行优先级后，任务仍然占有优先级资源，可以随时恢复在任务优先级下运行。

参数

prio —— 任务优先级。

run_prio — 目标运行优先级。

返回值

RS_EO RANGE — 指定参数错误，如优先级超出范围。
RS_ENOTEXIST — 指定任务不存在。
RS_EINVAL — 期望修改的优先级已经被别的任务占用，或者由于任务已被互斥量等资源控制了运行优先级，此时更改任务运行优先级操作将失败。
RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

C.5 task_reset_runprio

重置任务运行优先级。

原 型

```
status_t task_reset_runprio(prio_t prio);
```

描 述

任务运行优先级被更改后，通过该服务重置任务的运行优先级，将任务的运行优先级恢复到最初的状态。

参 数

prio —— 任务优先级。

返回值

RS_EOUTRANGE —— 指定参数错误，优先级超出范围。
RS_ENOTEXIST —— 指定任务不存在。
RS_EINVAL —— 任务运行优先级被互斥量等其他资源控制，不能重置任务运行优先级。
RS_EOK —— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

C.6 task_ident

获取任务标识。

原 型

```
prio_t task_ident(void);
```

描 述

该服务返回当前正在执行的任务的优先级。每个优先级只能有一个任务，因此，任务优

优先级通常用来作为任务的标识。

参 数

无。

返回值

当前的任务优先级。

调用者

任务。

阻 塞

非阻塞。

其 他

注意，该接口只反映被调用时刻的任务优先级，如果任务优先级可以被更改，该返回值只作为应用参考使用。

C.7 task_self

获取任务伪标识。

原 型

```
prio_t task_self(void);
```

描 述

该服务返回当前正在执行的任务的伪优先级。伪优先级并不是任务的真实优先级，伪优先级只对当前任务有效，只能用来识别当前任务。

参 数

无。

返回值

当前任务的伪优先级。

调用者

任务。

阻 塞

非阻塞。

其 他

C.8 task_current

获取任务伪标识。

原 型

```
prio_t task_current(void);
```

描 述

该服务返回当前正在执行的任务的运行优先级。

参 数

无。

返回值

当前任务的运行优先级。

调用者

任务。

阻 塞

非阻塞。

其 他

无。

C.9 task_sleep

任务休眠指定时间。

原 型

```
status_t task_sleep(tick_t ticks);
```

描 述

该服务使任务进入休眠状态，到达指定的时间后继续执行任务。时间单位是系统时钟。正在休眠的任务可以被唤醒任务服务提前唤醒。

参 数

ticks —— 指定需要休眠的系统节拍数，系统节拍的范围是 0 到 $2^{32} - 1$ 。

返回值

RS_EOK —— 操作成功。

调用者

任务。

阻 塞

可能引起阻塞。

其 他

无。

C.10 task_wake

唤醒休眠中的任务。

原 型

```
status_t task_wake(prio_t prio);
```

描 述

该服务唤醒一个正在休眠中的任务。调用该服务可以提前唤醒休眠中的任务，恢复任务为就绪状态。

参 数

prio —— 任务优先级。

返回值

RS_EOUTRANGE	—— 指定参数错误，优先级超出范围。
RS_ENOTEXIST	—— 指定任务不存在。
RS_EUNKNOW	—— 任务不在休眠状态。
RS_EOK	—— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

C.11 task_suspend

挂起任务。

原 型

```
status_t task_suspend(prio_t prio);
```

描 述

该服务挂起指定的任务。处于挂起状态的任务，所有的活动都停止，任务处于凝固状态，包括任务的休眠时间也会被暂停。一个任务可以挂起自身，也可以挂起别的任务，当任务被挂起之后，必须调用 **task_resume** 服务接口恢复。

参 数

prio — 任务优先级。

返回值

RS_EOUTRANGE	—— 指定参数错误，优先级超出范围。
RS_ENOTEXIST	—— 指定任务不存在。
RS_EOK	—— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

如挂起自身，可能引起阻塞。

其 他

无。

C.12 task_resume

恢复任务。

原 型

```
status_t task_resume(prio_t prio);
```

描 述

该服务恢复被 `task_suspend` 挂起，或者创建任务时使用 `RS_OPT_SUSPEND` 选项而挂起的任务。

参 数

`prio` —— 任务优先级。

返回值

<code>RS_EOUTRANGE</code>	—— 指定参数错误，优先级超出范围。
<code>RS_ENOTEXIST</code>	—— 指定任务不存在。
<code>RS_EOK</code>	—— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

C.13 task_protect

任务保护。

原 型

```
void task_protect(void);
```

描 述

该服务使当前正在运行的任务进入保护状态。在任务保护状态下，将禁止对该任务进行的删除操作和修改任务的优先级操作，确保了任务不会被意外删除，这样，允许任务在安全状态下使用关键资源。

参 数

无。

返回值

无。

调用者

任务。

阻 塞

非阻塞。

其 他

无。

C.14 task_unprotect

解除任务保护。

原 型

```
void task_unprotect(void);
```

描 述

该服务解除当前运行的任务的保护状态。

参 数

无。

返回值

无。

调用者

任务。

阻 塞

非阻塞。

其 他

无。

C.15 task_info

获取任务信息。

原 型

```
status_t task_info(tinfo_t __p_* info);
```

描 述

通过该服务取得任务的运行时的状态信息。获取的信息包括任务当前的状态、优先级、栈指针、栈起始地址、栈尺寸以及栈的使用情况等。

参 数

info —— 存放任务信息的结构。

返回值

RS_EOUTRANGE	—— 指定参数错误，优先级超出范围。
RS_ENOTEXIST	—— 指定任务不存在。
RS_EOK	—— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

附录 D 时钟服务

D.1 tick_get

获取系统节拍。

原 型

```
int32u tick_get(void);
```

描 述

该服务取得当前系统时钟的值。应用初始化时，系统时钟初始化为 0，在每个系统定时器中断发生时，系统时钟的值加 1。

参 数

无。

返回值

系统时钟的值。

调用者

中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

D.2 tick_set

设置系统节拍。

原 型

```
void tick_set(int32u ticks);
```

描 述

该服务设置系统内部时钟的值，在以后每个系统定时器中断到来时，系统时钟的计数值将以此为基础增加 1。

参 数

新的系统时钟值。

返回值

无。

调用者

中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

附录 E 中断服务

E.1 interrupt_attach

注册中断服务。

原 型

```
ientry_t interrupt_attach(vect_t vect, ientry_t ientry);
```

描 述

注册中断服务到指定的中断向量。通过该服务，可以将中断服务程序和中断向量关联起来，当发生中断时，处理器暂停正在执行的程序，保留现场后执行相应的中断服务程序。

参 数

vect — 中断向量。
ientry — 中断服务函数入口。

返回值

该中断向量的前一个中断服务函数入口。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

E.2 interrupt_detach

注销中断服务。

原 型

```
ientry_t interrupt_detach(vect_t vect);
```

描 述

注销中断将对应中断向量与服务例程分离。

参 数

vect —— 中断向量。

返回值

当前的中断服务函数入口。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

E.3 interrupt_enable

允许中断服务。

原 型

```
status_t interrupt_enable(vect_t vect);
```

描 述

允许指定的中断服务。注册中断服务后，默认中断服务是被禁止的。这时产生相应的中断，中断服务函数也不会被调用，只有允许中断后，才能响应中断服务

参 数

vect —— 中断向量。

返回值

RS_EOUTRANGE —— 指定参数错误，中断向量号超限。
RS_EOK —— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

E.4 interrupt_disable

禁止中断服务。

原 型

```
status_t interrupt_disable(vect_t vect);
```

描 述

禁止指定的中断服务。在中断服务禁止状态，即使产生相应的中断，中断服务函数也不会被调用。

参 数

`vect` —— 中断向量。

返回值

`RS_EOUTRANGE` —— 指定参数错误，中断向量号超限。

`RS_EOK` —— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

E.5 interrupt_catch

捕获中断。

原 型

```
status_t interrupt_catch(vect_t vect);
```

描 述

该服务临时允许指定的中断，响应在中断关闭过程中发生的中断信号。在中断关闭的情况下，通过捕获中断服务，可以较为及时的响应已经发生的中断。

参 数

`vect` —— 中断向量。

返回值

`RS_EOUTRANGE` —— 指定参数错误，中断向量号超限。
`RS_EOK` —— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

1. 该服务依赖于硬件特性的支持，某些硬件可能不支持。
2. 指定的中断在允许的状态下，使用该服务将导致中断被禁止。

附录 F 定时器服务

F.1 timer_create

创建定时器。

原 型

```
status_t timer_create(timer_t __p_* timer, name_t name, entry_t entry, arg_t arg,
```



```
tick_t ticks, int8u options);
```

描 述

该服务创建一个定时器。通过指定定时器到期函数与定时器周期等参数，能够在特定的时间执行指定的服务程序。

参 数

timer — 指向定时器的指针。
name — 定时器名字。
entry — 定时器到期函数。
arg — 到期函数参数。
ticks — 定时器周期。
options — 定时器选项：
 1) RS_OPT_ENABLE 自动启动；RS_OPT_DISABLE 禁止自动启动。
 2) RS_OPT_SINGLE 单次定时器；RS_OPT_REPEAT 重复定时器。
选项 1) 与 2) 可以通过按位或运算指定；如，指定一个手动启动的重复定时器使用以下运算：RS_OPT_DISABLE | RS_OPT_REPEAT；指定为 0 相当于 RS_OPT_ENABLE | RS_OPT_SINGLE。

返回值

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

F.2 timer_delete

删除定时器。

原 型

```
status_t timer_delete(timer_t __p_* timer);
```

描 述

该服务删除指定的定时器。删除定时器可以为系统节约资源开销，并且相应的定时器服务将不在被调用。

参 数

timer —— 指向定时器的指针。

返回值

RS_ELOCK —— 定时器被系统锁定，比如定时器正在执行服务函数。

RS_EOK —— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

F.3 timer_enable

启动定时器。

原 型

```
status_t timer_enable(timer_t __p_* timer);
```

描 述

该服务启动指定的定时器。如果定时器在创建时使用 **RS_OPT_DISABLE** 选项，或者应用了停止定时器服务，通过启动定时器服务可以将定时器重新激活。如果多个定时器同时到期，相应的到期服务函数将按照定时器启动时间的顺序执行。

参 数

timer — 指向定时器的指针。

返回值

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

F.4 timer_disable

停止定时器。

原型

```
status_t timer_disable(timer_t __p_* timer);
```

描述

该服务停止指定的定时器。处于停止状态的定时器，可以通过启动定时器服务将其重新激活。

参数

`timer` — 指向定时器的指针。

返回值

`RS_EOK` — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

无。

F.5 timer_control

定时器控制。

原 型

```
status_t timer_control(timer_t __p_* timer, int8u options);
```

描 述

该服务用来修改定时器属性。控制项包括将定时器更改为单次或者周期定时器。

参 数

timer — 指向定时器的指针。
options — 定时器选项：
 1) RS_OPT_SINGLE 单次定时器；
 2) RS_OPT_REPEAT 重复定时器。

返回值

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

附录 G 互斥量服务

G.1 mutex_create

创建互斥量。

原 型

```
status_t mutex_create(mutex_t __p_* mut, name_t name, prio_t prio);
```

描 述

该服务创建并初始化一个互斥量。互斥量被用来提供资源互斥访问，互斥量拥有优先级属性，需要占用一个系统优先级资源。

参 数

mut — 指向互斥量结构的指针。
name — 互斥量名字，指定为 0 不使用名字。
prio — 互斥信号量的优先级。

返回值

RS_EOUTRANGE — 指定参数错误，如优先级超出范围。
RS_EEXIST — 优先级已经被占用。
RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

G.2 mutex_delete

删除互斥量。

原 型

```
status_t mutex_delete(mutex_t __p_* mut);
```

描 述

该服务删除一个指定的互斥量。删除一个互斥量，必须确保该互斥量不再被使用，并且没有任务占用它。

参 数

mut —— 指向互斥量结构的指针。

返回值

RS_EEXIST —— 互斥量被任务占用。

RS_EOK —— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

G.3 mutex_wait

等待并获取一个互斥量。

原 型

```
status_t mutex_wait(mutex_t __p_* mut, tick_t ticks);
```

描 述

该服务等待并获得一个互斥量。如果互斥量可用，则获得互斥量；如果互斥量已经被调用的服务拥有，则该服务增加拥有互斥量的引用计数，并返回成功；如果互斥量已经被其他任务拥有，则调用该服务的任务将进入等待状态，直到互斥量有效，或者指定的超时返回。

参 数

mut —— 指向互斥量结构的指针。
ticks —— 指定超时时间，单位为系统节拍（Tick）。

返回值

RS_ELOCK —— 系统被锁定。
RS_EAGAIN —— 试图重复取得互斥量（配置为不支持嵌套模式）。
RS_ETIMEOUT —— 等待超时。
RS_EOK —— 操作成功。

调用者

任务。

阻 塞

可能引起阻塞。

其 他

无。

G.4 mutex_trywait

无等待获取一个互斥量。

原 型

```
status_t mutex_trywait(mutex_t __p_ * mut);
```

描 述

该服务无等待获得一个互斥量。如果互斥量可用，则获得互斥量；如果互斥量已经被其他任务拥有，则调用该服务的任务将获得一个错误代码，并立刻返回。无论互斥量在处于何种状态，调用该服务都不会引起阻塞。

参 数

mut —— 指向互斥量结构的指针。

返回值

RS_ELOCK	—— 系统被锁定。
RS_ENAVAIL	—— 互斥量不可用。
RS_EOK	—— 操作成功。

调用者

任务。

阻 塞

非阻塞。

其 他

无。

G.5 mutex_release

释放互斥量。

原 型

```
status_t mutex_release(mutex_t __p_ * mut);
```

描 述

该服务释放一个互斥量。被释放的互斥量的引用计数器将减 1，如果引用计数变成 0，则互斥量就会变得可用。如果任务的运行优先级被互斥量提升，那么释放该互斥量后，任务将恢复为它占有互斥量之前的优先级运行。

参 数

mut —— 指向互斥量结构的指针。

返回值

RS_ELOCK	—— 系统被锁定。
RS_EINVAL	—— 错误的释放互斥量，如该任务没有占用该信号量。
RS_EOK	—— 操作成功。

调用者

任务。

阻 塞

非阻塞。

其 他

无。

G.6 mutex_info

获取互斥量信息。

原 型

```
status_t mutex_info(mutex_t __p_* mut, mutinfo_t __out_* info);
```

描 述

通过该服务取得互斥量的运行时的状态信息。获取的信息包括互斥量名字、优先级、所有者、引用计数。

参 数

mut —— 指向互斥量结构的指针。
info —— 存放返回信息的结构。

返回值

RS_EOK —— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

附录 H 信号量服务

H.1 semaphore_create

创建信号量。

原型

```
status_t semaphore_create(sem_t __p_* sem, name_t name, count_t value);
```

描述

该服务创建并初始化一个信号量。信号量在使用前必须初始化，通过接口参数可以指定信号量的初始实例值和名字。

参数

sem — 指向信号量结构的指针。
name — 信号量名字，指定为 0 不使用名字。
value — 信号量初始值。

返回值

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

无。

H.2 semaphore_delete

删除信号量。

原 型

```
status_t semaphore_delete(sem_t __p_* sem);
```

描 述

该服务删除指定的信号量。删除不用的信号量，可以释放资源。避免删除一个正在使用的信号量，以防产生逻辑错误。

参 数

sem —— 指向信号量结构的指针。

返回值

RS_EOK —— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

H.3 semaphore_wait

等待并获取指定的信号量。

原型

```
status_t semaphore_wait(sem_t __p_* sem, tick_t ticks);
```

描述

该服务等待并获得指定的信号量。如果信号量的值不为 0，则信号量的值会被减 1。如果信号量的值为 0，表示信号量的资源实例已经用完，则任务将进入阻塞状态，直到信号量有效，或者指定的超时返回。

参数

sem —— 指向信号量结构的指针。
ticks —— 指定超时时间，单位为系统节拍（Tick）。

返回值

RS_ELOCK —— 系统被锁定。
RS_ETIMEOUT —— 等待超时。
RS_EOK —— 操作成功。

调用者

任务。

阻塞

可能引起阻塞。

其他

无。

H.4 semaphore_trywait

无等待获取指定的信号量。

原 型

```
status_t semaphore_trywait(sem_t __p_* sem);
```

描 述

该服务无等待获得指定的信号量。如果信号量的值不为 0，则信号量的值会被减 1。如果信号量的值为 0，表示信号量的资源实例已经用完，则任务将返回失败代码。无论信号量是否有效，调用该服务都不会引起阻塞。

参 数

sem — 指向信号量结构的指针。

返回值

RS_ENAVAIL — 信号量无效。
RS_EOK — 操作成功。

调用者

中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

H.5 semaphore_post

释放信号量实例。

原型

```
status_t semaphore_post(sem_t __p_* sem);
```

描述

该服务释放信号量实例，使指定信号量的值加 1。释放一个信号实例时，如果有任务因等待该信号量而阻塞，则阻塞队列中最高优先级的任务将被唤醒。

参数

sem —— 指向信号量结构的指针。

返回值

RS_EOK —— 操作成功。

调用者

中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

无。

H.6 semaphore_info

获取信号量信息。

原 型

```
status_t semaphore_info(sem_t __p_* sem, seminfo_t __out_* info);
```

描 述

通过该服务取得信号量运行时的状态信息。获取的信息包括信号量名字、实例计数。

参 数

sem —— 指向信号量结构的指针。

info —— 存放返回信息的结构。

返回值

RS_EOK —— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

附录 I 二值信号量服务

I.1 sembinary_create

创建二值信号量。

原 型

```
status_t sembinary_create(sem_t __p_* semb, name_t name, bool value);
```

描 述

该服务创建并初始化一个二值信号量。二值信号量在使用前必须初始化，通过接口参数可以指定二值信号量的初始实例值和名字。

参 数

semb — 指向二值信号量结构的指针。
name — 二值信号量名字，指定为 0 不使用名字。
value — 二值信号量初始值。

返回值

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

I.2 sembinary_delete

删除二值信号量。

原 型

```
status_t sembinary_delete(sem_t __p_* sem);
```

描 述

该服务删除指定的二值信号量。删除不用的二值信号量，可以释放资源。避免删除一个正在使用的二值信号量，以防产生逻辑错误。

参 数

`semb` — 指向二值信号量结构的指针。

返回值

`RS_EOK` — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

1.3 sembinary_wait

等待并获取指定的二值信号量。

原 型

```
status_t sembinary_wait(sem_t __p_* sem, tick_t ticks);
```

描 述

该服务等待并获得指定的二值信号量。如果二值信号量值为 1 (**true**)，成功获取信号量，则信号量值变为 0 (**false**)；如果信号量值为 0 (**false**)，调用获取二值信号量服务，则调用任务将进入阻塞状态，直到信号量有效，或者指定的超时返回。

参 数

sem —— 指向二值信号量结构的指针。
ticks —— 指定超时时间，单位为系统节拍 (Tick)。

返回值

RS_ELOCK —— 系统被锁定。
RS_ETIMEOUT —— 等待超时。
RS_EOK —— 操作成功。

调用者

任务。

阻 塞

可能引起阻塞。

其 他

无。

I.4 sembinary_trywait

无等待获取指定的二值信号量。

原 型

```
status_t sembinary_trywait(sem_t __p_* sem);
```

描 述

该服务无等待获得指定的二值信号量。当二值信号量有效时（1 或者 **true**），该服务将使信号量的值变为无效（0 或者 **false**），并且返回成功；当二值信号量无效时（0 或者 **false**），调用该接口将返回错误代码。

参 数

semb — 指向二值信号量结构的指针。

返回值

RS_ENAVAIL — 二值信号量无效。
RS_EOK — 操作成功。

调用者

中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

1.5 sembinary_post

释放二值信号量。

原 型

```
status_t sembinary_post(sem_t __p_* sem);
```

描 述

该服务释放信号量实例，使指定信号量变为有效。如果有任务因等待该信号量而阻塞，则其中最高优先级的任务将被唤醒。

参 数

`semb` — 指向二值信号量结构的指针。

返回值

`RS_EOK` — 操作成功。

调用者

中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

I.6 sembinary_info

获取二值信号量信息。

原 型

```
status_t sembinary_info(sem_t __p_* semb, seminfo_t __out_* info);
```

描 述

通过该服务取得二值信号量运行时的状态信息。获取的信息包括二值信号量名字、实例是否有效。

参 数

semb — 指向二值信号量结构的指针。
info — 存放返回信息的结构。

返回值

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

附录 J 邮箱服务

J.1 mailbox_create

创建邮箱。

原 型

```
status_t mailbox_create(mbox_t __p_* mbox, name_t name);
```

描 述

该服务创建并初始化一个邮箱。邮箱是任务间常用的、轻量级的通讯方式。一个邮箱能够容纳一条消息，它提供了由一个任务（或者中断服务、定时器应用）向另一个任务发送信息的手段。

参 数

mbox — 指向邮箱的指针。
name — 邮箱的名字，指定为 0 不使用名字。

返回值

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

J.2 mailbox_delete

删除指定的邮箱。

原 型

```
status_t mailbox_delete(mbox_t __p_* mbox);
```

描 述

该服务删除一个指定的邮箱。删除一个邮箱之前，必须确保该邮箱不再被使用，并且没有任务等待该邮箱的消息，否则将返回失败代码。

参 数

mbox — 指向邮箱的指针。

返回值

RS_EEXIST — 邮箱的等待队列不为空。

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

J.3 mailbox_send

向指定的邮箱发送消息。

原 型

```
status_t mailbox_send(mbox_t __p_* mbox, mail_t mail);
```

描 述

该服务向指定的邮箱发送一条消息。邮箱消息是 32 位的整型数据，能够存放最大 4 个字节的数据。当消息大于 4 个字节时，则将需要传递的数据放到一个缓冲区中，通过发送缓冲区的地址实现间接传递。

参 数

mbox — 指向邮箱的指针。
mail — 邮箱消息内容。

返回值

RS_EFULL — 邮箱已满。
RS_EOK — 操作成功。

调用者

中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

J.4 mailbox_receive

接收邮箱消息。

原 型

```
status_t mailbox_receive(mbox_t __p_* mbox, mail_t __out_* mail, tick_t ticks);
```

描 述

该服务接收指定邮箱的信息。当邮箱保存有效消息时，接收邮箱消息服务马上返回，并取走一个邮箱消息。当邮箱为空时，接收邮箱消息的任务将进入阻塞状态，直到邮箱收到一个有效消息，或者设定的超时时间到达。

参 数

mbox — 指向邮箱的指针。
mail — 存放邮箱消息的起始地址。
ticks — 指定超时时间，单位为系统节拍（Tick）。

返回值

RS_ELOCK — 系统被锁定。
RS_ETIMEOUT — 等待超时。
RS_EOK — 操作成功。

调用者

任务。

阻 塞

可能引起阻塞。

其 他

无。

J.5 mailbox_flush

清空指定邮箱。

原 型

```
status_t mailbox_flush(mbox_t __p_* mbox);
```

描 述

该服务把指定的邮箱信息清空。无论邮箱是否包含了消息，清空邮箱操作都会使得邮箱恢复到原始空闲的状态。

参 数

mbox — 指向邮箱的指针。

返回值

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

附录 K 队列服务

K.1 queue_create

创建一个队列。

原 型

```
status_t queue_create(queue_t __p_* queue, name_t name, void __p_* buff,  
                      mmsz_t size, mmsz_t entries, int8u options);
```

描 述

该服务创建并初始化一个队列。一个队列可以含有一个或多个消息，每个消息的长度是相等的，并遵循先进先出 FIFO 或者后进先出 LIFO 顺序。

参 数

queue — 指向队列的指针。

name — 队列的名字，指定为 0 不使用名字。

buff — 队列缓冲区的起始地址，指定为 NULL，则由系统自动分配缓冲区空间。

size — 消息大小（字节）。

entries — 队列长度。

options — 队列选项：

- 1) RS_OPT_FIFO 先进先出，RS_OPT_LIFO 后进先出。
- 2) RS_OPT_KFREE 队列被删除时自动释放队列缓冲区；如果指定该选项，buff 地址必须由动态内存分配 mpool_alloc 获得；如果 buff 指定为 NULL，则无需指定该选项。

返回值

RS_ENULL — 队列缓冲区分配失败。

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

无。

K.2 queue_delete

删除指定的队列。

原型

```
status_t queue_delete(queue_t __p_* queue);
```

描述

该服务删除一个指定的队列。删除一个队列之前，必须确保该队列不再被使用，并且没有任务等待该队列的消息。如果有一个或者多个任务正在等待队列中的消息时，删除该队列是危险的，调用删除队列服务将获得一个失败代码，指示队列删除失败。

参数

queue — 指向队列的指针。

返回值

RS_EEXIST — 至少有一个任务正在等待该队列的消息。

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

无。

K.3 queue_send

向指定的队列发送消息。

原型

```
status_t queue_send(queue_t __p_* queue, const void __p_* buff);
```

描述

该服务向一个指定的队列发送消息。发送队列消息是将整个消息复制到队列的消息缓冲区中，而不是复制消息的地址。

参数

queue — 指向队列的指针。
buff — 指向队列消息的指针。

返回值

RS_EFULL — 队列缓冲区已满。
RS_EOK — 操作成功。

调用者

中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

无。

K.4 queue_receive

接收指定的队列的消息。

原型

```
status_t queue_receive(queue_t __p_* queue, void __out_* buff, tick_t ticks);
```

描述

该服务接收指定队列中的消息。如队列中包含有效消息，该服务将消息从队列缓冲区中复制到指定的目标缓冲区，并从队列中删除该消息。如队列为空，接收队列消息的任务将进入阻塞状态，直到队列收到一个有效消息，或者设定的超时时间到达。

参数

queue — 指向队列的指针。
buff — 指向存放队列消息的地址。
ticks — 指定超时时间，单位为系统节拍（Tick）。

返回值

RS_ELOCK — 系统被锁定。
RS_ETIMEOUT — 等待超时。
RS_EOK — 操作成功。

调用者

任务。

阻 塞

可能引起阻塞。

其 他

无。

K.5 queue_flush

清空指定的队列。

原 型

```
status_t queue_flush(queue_t __p_* queue);
```

描 述

该服务删除一个队列中的所有消息，并使得队列恢复到创建时的状态。

参 数

queue —— 指向队列的指针。

返回值

RS_EOK —— 操作成功。

调用者

任务。

阻 塞

非阻塞。

其 他

无。

附录 L 事件标志服务

L.1 event_create

创建一个事件标志。

原 型

```
status_t event_create(event_t __p_* event, name_t name, bits_t bits);
```

描 述

该服务创建并初始化一个事件标志。事件标志是任务同步的强有力的工具。事件标志包含一个宽 32 位的事件标志位组，可以表示 32 个互相独立的事件。

参 数

event — 指向事件标志的指针。
name — 事件标志的名字，指定为 0 不使用名字。
bits — 事件标志位组的初始值。

返回值

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

L.2 event_delete

删除指定的事件标志。

原 型

```
status_t event_delete(event_t __p_* event);
```

描 述

该服务一个指定的事件标志。如果有一个或者多个任务正在等待事件标志中的事件时，删除该事件标志是不允许的，调用删除事件标志服务将获得一个失败代码。

参 数

event —— 指向事件标志的指针。

返回值

RS_EEXIST —— 至少有一个任务正在等待该事件标志。

RS_EOK —— 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

L.3 event_wait

等待并获取指定的事件标志。

原 型

```
status_t event_wait(event_t __p_* event, bits_t bits, int8u options, tick_t ticks);
```

描 述

该服务等待并获得指定的事件标志。当事件标志位组中的指定事件标志位被置位，则服务满足。如果事件不满足，则任务进入等待状态或者返回失败代码。通过指定的条件与标志位组合，支持丰富灵活的事件形式。

参 数

- event — 指向事件标志的指针。
- bits — 请求的事件标志位组。
- options — 服务选项：
 - 1) RS_OPT_AND 正逻辑与，所有位被置 1 时成立；
 - 2) RS_OPT_OR 正逻辑或，任何一位或多位被置 1 时成立；
 - 3) RS_OPT_NAND 负逻辑与，所有位被置 0 时成立；
 - 4) RS_OPT_NOR 负逻辑或，任何一位或多位被置 0 时成立；
 - 5) RS_OPT_XAND 反逻辑与，所有位被置反时成立；
 - 6) RS_OPT_XOR 反逻辑或，任何一位或多位被置反时成立。
- ticks — 指定超时时间，单位为系统节拍（Tick）。

返回值

- RS_ELOCK — 系统被锁定。
- RS_ETIMEOUT — 等待超时。

RS_EINVAL — 参数无效。
RS_EOK — 操作成功。

调用者

任务。

阻 塞

可能引起阻塞。

其 他

无。

L.4 event_trywait

无等待获取指定的事件标志。

原 型

status_t event_trywait(event_t __p_* event, bits_t bits, int8u options);

描 述

该服务无需等待获得指定的事件标志。当事件标志位组中的指定事件标志位被置位，则服务满足；如果事件不满足，则返回失败代码。

参 数

event — 指向事件标志的指针。
bits — 请求的事件标志位组。
options — 服务选项：
 1) RS_OPT_AND 正逻辑与，所有位被置 1 时成立；
 2) RS_OPT_OR 正逻辑或，任何一位或多位被置 1 时成立；

- 3) RS_OPT_NAND 负逻辑与，所有位被置 0 时成立；
- 4) RS_OPT_NOR 负逻辑或，任何一位或多位被置 0 时成立；
- 5) RS_OPT_XAND 反逻辑与，所有位被置反时成立；
- 6) RS_OPT_XOR 反逻辑或，任何一位或多位被置反时成立。

返回值

RS_EINVAL	-- 参数无效。
RS_ENAVAIL	-- 指定事件不满足。
RS_EOK	-- 操作成功。

调用者

中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

无。

L.5 event_post

设置指定的事件标志。

原型

```
status_t event_post(event_t __p_* event, bits_t bits, int8u options);
```

描述

该服务对指定的事件标志中的一个或者多个标志位置位（置 1）或复位（清 0）。

参数

event — 指向事件标志的指针。
bits — 设置的事件标志位组。
options — 设置选项：
 1) RS_OPT_SET 将事件位置为 1；
 2) RS_OPT_CLR 将事件位置为 0。

返回值

RS_EINVAL — 参数无效。
RS_EOK — 操作成功。

调用者

中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

L.6 event_info

获取事件标志信息。

原 型

```
status_t event_info(event_t __p_* event, evinfo_t __out_* info);
```

描 述

通过该服务取得指定的事件标志运行时的状态信息。获取的信息包括事件标志的名字，事件标志位组的值。

参 数

event — 指向事件标志的指针。
info — 存放返回信息的结构。

返回值

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

附录 M 内存服务

M.1 mpool_alloc

向内存池集申请指定大小的内存。

原 型

```
void __p_* mpool_alloc(mmsz_t size);
```

描 述

通过该服务向内存池集申请指定大小的内存。如果内存池中有合适的空闲内存块，则返

回指向空闲内存首地址的指针，否则返回空指针（NULL）。

参 数

size —— 申请内存的尺寸，单位字节（bytes）。

返回值

NULL —— 申请失败，无可用内存。

非 NULL —— 申请成功，返回内存首地址。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

M.2 mpool_free

释放内存。

原 型

```
void mpool_free(void __p_* p);
```

描 述

通过该服务释放从内存池中申请得到的内存块。服务将内存释放到空闲内存池中，其他应用可以再次使用该内存。

参 数

p —— 指向待释放的内存地址。

返回值

无。

调用者

应用初始化、中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

该服务释放的内存必须是通过申请内存池服务 `mpool_alloc` 获得的有效内存。

M.3 mpool_info

获取内存池信息。

原型

```
status_t mpool_info(mpinfo_t __out_* info);
```

描述

通过该服务取得指定的内存池运行时的状态信息。得到的信息包括内存池尺寸，内存块数，空闲块数目，池链表的头指针。

参数

info —— 存放返回信息的结构，属性项 **index** 是传入参数，用来指定内存池的索引。

返回值

RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

附录 N 设备管理服务

N.1 build_device

创建设备标识。

原 型

```
device_t build_device(int16u major, int16u minor);
```

描 述

该服务根据设备的主设备号和从设备号，返回一个类型为 `device_t` 的标识。`device_t` 是一个宽为 32 位的整型结构，高 16 位存放了主设备号，低 16 存放从设备号。通过这个标识能识别系统中唯一的设备。

参 数

major — 主设备号。

minor — 从设备号。

返回值

设备的唯一标识

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

N.2 driver_install

安装设备驱动。

原 型

```
status_t driver_install(int16u major, driver_table_t __p_* table);
```

描 述

通过该服务安装设备驱动程序。系统通过驱动程序操作具体的设备。每个设备都对应一个驱动程序，但一个驱动程序可以同时驱动多个设备。

参 数

major	—— 主设备号，指定为 0 时将由系统自动分配一个设备号。
table	—— 设备驱动接口表。

返回值

- RS_EOUTRANGE — 参数超限，主设备号超范围。
- RS_EFULL — 对应设备号驱动已经安装，或无可用设备号（自动分配）。
- 大于 0 — 操作成功，返回成功安装的主设备号。

调用者

应用初始化、中断服务、定时器应用、任务。

阻 塞

非阻塞。

其 他

无。

N.3 driver_uninstall

卸载设备驱动。

原 型

status_t driver_uninstall(int16u major);

描 述

通过该服务将设备驱动从设备管理器中卸载。

参 数

major — 主设备号。

返回值

- RS_EOUTRANGE — 参数超限，主设备号超范围。
- RS_ENOTEXIST — 对应设备驱动未安装。

RS_ELOCK — 设备驱动被锁定（正忙）。
RS_EOK — 操作成功。

调用者

应用初始化、中断服务、定时器应用、任务。

阻塞

非阻塞。

其他

无。

N.4 device_init

初始化指定设备。

原型

```
int device_init(device_t device, arg_t arg);
```

描述

通过该服务初始化指定的设备。设备的初始化工作一般在系统初始化或者设备加载的时候进行。

参数

device — 设备标识。
arg — 操作参数。

返回值

RS_EOUTRANGE — 参数超限，无效的设备标识。

- RS_EUNKNOW —— 未知错误，可能对应设备驱动未安装。
- 其他 —— 由驱动程序定义。

调用者

由驱动程序定义。

阻 塞

由驱动程序定义。

其 他

无。

N.5 device_open

打开指定设备。

原 型

```
int device_open(device_t device, arg_t arg);
```

描 述

该服务打开指定的设备。在对设备进行读写的时候，通常通过该服务获得对设备的操作权限，打开设备可以防止多个使用者进行设备读写冲突。

参 数

- device —— 设备标识。
- arg —— 操作参数。

返回值

- RS_EOUTRANGE —— 参数超限，无效的设备标识。

- RS_EUNKNOW —— 未知错误，可能对应设备驱动未安装。
- 其他 —— 由驱动程序定义。

调用者

由驱动程序定义。

阻 塞

由驱动程序定义。

其 他

无。

N.6 device_close

关闭指定设备。

原 型

```
int device_close(device_t device);
```

描 述

该服务关闭指定的设备，与打开设备服务配合使用。

参 数

device —— 设备标识。

返回值

- RS_EOUTRANGE —— 参数超限，无效的设备标识。
- RS_EUNKNOW —— 未知错误，可能对应设备驱动未安装。
- 其他 —— 由驱动程序定义。

调用者

由驱动程序定义。

阻 塞

由驱动程序定义。

其 他

无。

N.7 device_ioctl

更改或控制设备的属性。

原 型

```
int device_ioctl(device_t device, arg_t arg);
```

描 述

该服务提供对设备状态和属性操作的通道。通过设备控制，可以读入设备相关状态信息，或者设定设备的属性，工作模式，控制选项等。

参 数

- device — 设备标识。
- arg — 操作参数。

返回值

- RS_EOUTRANGE — 参数超限，无效的设备标识。
- RS_EUNKNOW — 未知错误，可能对应设备驱动未安装。
- 其他 — 由驱动程序定义。

调用者

由驱动程序定义。

阻塞

由驱动程序定义。

其他

无。

N.8 device_read

从指定设备读入数据。

原型

```
int device_read(device_t device, char __out_* buff, int size);
```

描述

该服务负责从设备中读入数据到指定的缓冲区中。设备读写服务是设备与系统缓冲区互交数据的主要手段，可以提供大容量的数据交换服务。

参数

device — 设备标识。
buff — 存放读入数据的缓冲区。
size — 缓冲区尺寸。

返回值

RS_EOUTRANGE — 参数超限，无效的设备标识。
RS_EUNKNOWN — 未知错误，可能对应设备驱动未安装。

大于或等于 0 —— 从设备中成功读入数据的大小。

调用者

由驱动程序定义。

阻 塞

由驱动程序定义。

其 他

无。

N.9 device_write

往指定设备写入数据。

原 型

```
int device_write(device_t device, const char __p_* buff, int size);
```

描 述

该服务负责将缓冲区数据写到指定的设备。与设备读服务一样，设备写服务提供了大容量数据的写入操作。

参 数

device —— 设备标识。
buff —— 输出数据的缓冲区。
size —— 缓冲区尺寸。

返回值

RS_EOUTRANGE —— 参数超限，无效的设备标识。

- RS_EUNKNOWN

大于或等于 0
- 未知错误，可能对应设备驱动未安装。

—— 成功写入设备的数据的大小。

调用者

由驱动程序定义。

阻 塞

由驱动程序定义。

其 他

无。