[野火]RT-Thread 内核实现与应用开发实战
——基于 RT1052

EmbedFire

野火电子

2021 年 01 月 21 日

Contents

关于本项目 **1**
关于野火 **2**
开源共享，共同进步 2
联系方式 2
第 1 章 推荐序 **3**
第 2 章 前言 **6**
2.1 关于本书 6
2.2 如何学习本书 7
2.3 本书的参考资料 7
2.4 本书的编写风格 8
2.5 本书的配套硬件 8
2.6 本书的技术论坛 12
TODO 和悬赏任务 **13**
第 3 章 如何学习 RTOS **14**
3.1 为什么要学习 RTOS 14
3.2 如何学习 RTOS 15
3.3 选择什么 RTOS 15
第 4 章 初识 RT-Thread **16**
4.1 RT-Thread 简介 16
4.1.1 版权 16
4.2 RT-Thread 的意义 16
第 5 章 第一部分：从 0 到 1 教你写 RT-Thread 内核 **18**
第 6 章 新建 RT-Thread 工程—软件仿真 **19**
6.1 新建本地工程文件夹 19
6.2 使用 KEIL 新建工程 21
6.2.1 New Progect 21
6.2.2 Select Device For Target 21

6.2.3 Manage Run-Time Environment . . . 24
6.2.4 在 KEIL 工程里面新建文件组 . . . 26
6.2.5 在 KEIL 工程里面添加文件 . . . 27
6.3 编写 main 函数 . . . 28
6.4 调试配置 . . . 29
6.4.1 设置软件仿真 . . . 29
6.4.2 修改时钟大小 . . . 30
6.4.3 添加头文件路径 . . . 31
第 7 章 裸机系统与多线程系统 . . . 33
7.1 裸机系统 . . . 33
7.1.1 轮询系统 . . . 33
7.1.2 前后台系统 . . . 34
7.2 多线程系统 . . . 36
第 8 章 线程的定义与线程切换的实现 . . . 40
8.1 本章目标 . . . 40
8.2 什么是线程 . . . 42
8.3 创建线程 . . . 43
8.3.1 定义线程栈 . . . 43
8.3.2 定义线程函数 . . . 46
8.3.3 定义线程控制块 . . . 48
8.3.4 实现线程创建函数 . . . 49
8.4 实现就绪列表 . . . 66
8.4.1 定义就绪列表 . . . 66
8.4.2 将线程插入到就绪列表 . . . 68
8.5 实现调度器 . . . 71
8.5.1 调度器初始化 . . . 71
8.5.2 启动调度器 . . . 74
8.5.3 第一次线程切换 . . . 76
8.5.4 系统调度 . . . 86
8.6 main 函数 . . . 90
8.7 实验现象 . . . 95

第 9 章 临界段的保护 100
9.1 什么是临界段 100
9.2 Cortex-M 内核快速关中断指令 100
9.3 关中断 101
9.4 开中断 102
9.5 临界段代码的应用 103
9.6 实验现象 109
第 10 章 对象容器的实现 110
10.1 什么是对象 110
10.1.1 对象枚举定义 110
10.1.2 对象数据类型定义 111
10.1.3 在线程控制块中添加对象成员 111
10.2 什么是容器 112
10.2.1 容器的定义 113
10.3 容器的接口实现 121
10.3.1 获取指定类型的对象信息 121
10.3.2 对象初始化 122
10.3.3 调用对象初始化函数 126
10.4 实验现象 127
第 11 章 空闲线程与阻塞延时的实现 128
11.1 实现空闲线程 128
11.1.1 定义空闲线程的栈 128
11.1.2 定义空闲线程的线程控制块 129
11.1.3 定义空闲线程函数 129
11.1.4 空闲线程初始化 130
11.2 实现阻塞延时 131
11.3 SysTick_Handler 中断服务函数 136
11.3.1 系统时基更新函数 138
11.4 main 函数 141
11.5 实验现象 147

第 12 章 支持多优先级 149
12.1 就绪列表 149
12.1.1 线程就绪优先级组 149
12.1.2 线程优先级表 155
12.2 修改代码，支持多优先级 159
12.2.1 修改线程控制块 160
12.2.2 修改调度器初始化函数 rt_system_scheduler_init() 162
12.2.3 修改线程初始化函数 rt_thread_init() 163
12.2.4 添加线程启动函数 rt_thread_startup() 165
12.2.5 修改空闲线程初始化函数 rt_thread_idle_init() 168
12.2.6 修改启动系统调度器函数 rt_system_scheduler_start() 169
12.2.7 修改系统调度函数 rt_schedule () 170
12.2.8 修改阻塞延时函数 rt_thread_delay() 175
12.2.9 修改时基更新函数 rt_tick_increase() 176
12.3 main 函数 178
12.4 实验现象 181
第 13 章 定时器的实现 183
13.1 实现定时器 183
13.1.1 系统定时器列表 184
13.1.2 系统定时器列表初始化 184
13.1.3 定义定时器结构体 185
13.1.4 在线程控制块中内置定时器 186
13.1.5 定时器初始化函数 187
13.1.6 定时器删除函数 191
13.1.7 定时器停止函数 191
13.1.8 定时器控制函数 192
13.1.9 定时器启动函数 194
13.1.10 定时器扫描函数 202
13.2 修改代码，支持定时器 206
13.2.1 修改线程初始化函数 206
13.2.2 修改线程延时函数 208

13.2.3 修改系统时基更新函数 213
13.2.4 修改 main.c 文件 214
13.3 实验现象 220
第 14 章 支持时间片 222
14.1 实现时间片 222
14.1.1 在线程控制块中添加时间片相关成员 222
14.1.2 修改线程初始化函数 223
14.1.3 修改系统时基更新函数 226
14.2 修改 main.c 文件 229
14.3 实验现象 235
第 15 章 第二部分：RT-Thread 内核应用开发 237
第 16 章 移植 RT-Thread 到 i.MX RT 238
16.1 获取 i.MX RT 的裸机工程模板 238
16.2 下载 RT-Thread Nano 源码 239
16.3 安装 RT-Thread Package 239
16.4 往裸机工程添加 RT-Thread 源码 241
16.4.1 拷贝 RT-Thread Package 到裸机工程根目录 241
16.4.2 拷贝 rtconfig.h 文件到 user 文件夹 242
16.4.3 拷贝 board.c 文件到 user 文件夹 242
16.4.4 RT-Thread 文件夹内容简介 242
16.4.5 bsp 文件夹简介 242
16.4.6 components 文件夹简介 243
16.4.7 include 文件夹简介 244
16.4.8 libcpu 文件夹简介 244
16.4.9 src 文件夹简介 244
16.4.10 新建 rtt/source 和 rtt/ports 组 245
16.4.11 指定 RT-Thread 头文件的路径 247
16.5 修改 rtconfig.h 249
16.6 修改 board.c 263
16.6.1 rt_hw_board_init() 函数 267
16.6.2 SysTick_Handler() 函数 267

16.7 修改 main.c 275
16.8 下载验证 277
第 17 章 创建线程 278
17.1 硬件初始化 278
17.2 创建单线程—SRAM 静态内存 282
17.2.1 定义线程函数 282
17.2.2 定义线程栈 283
17.2.3 定义线程控制块 284
17.2.4 初始化线程 285
17.2.5 启动线程 286
17.2.6 main.c 文件内容全貌 286
17.3 下载验证 289
17.4 创建单线程—SRAM 动态内存 290
17.4.1 动态内存空间的堆从哪里来 290
17.4.2 定义线程函数 293
17.4.3 定义线程栈 294
17.4.4 定义线程控制块指针 294
17.4.5 创建线程 294
17.4.6 启动线程 295
17.4.7 main.c 文件内容全貌 296
17.5 下载验证 299
17.6 创建多线程—SRAM 动态内存 299
17.7 下载验证 303
第 18 章 重映射串口到 rt_kprintf 函数 304
18.1 rt_kprintf() 函数定义 304
18.2 自定义 rt_hw_console_output 函数 306
18.3 测试 rt_kprintf 函数 307
18.3.1 硬件初始化 307
18.3.2 编写 rt_kprintf 测试代码 309
18.3.3 下载验证 310

第 19 章 RT-Thread 的启动流程 312
19.1 万事俱备，只欠东风 312
19.2 小心翼翼，十分谨慎 314
19.3 孰优孰劣 316
19.4 RT-Thread 的启动流程 316
19.4.1 $Sub$$main 函数 318
19.4.2 rtthread_startup() 函数 320
19.4.3 rt_application_init() 函数 322
19.4.4 $Super$$main() 函数 325
19.4.5 main 函数 326
第 20 章 线程管理 329
20.1 线程的基本概念 329
20.2 线程调度器的基本概念 329
20.3 线程状态的概念 330
20.4 线程状态迁移 331
20.5 常用的线程函数讲解 332
20.5.1 线程挂起函数 rt_thread_suspend() 332
20.5.2 线程恢复函数 rt_thread_resume() 335
20.6 线程的设计要点 337
20.7 线程管理实验 339
20.8 实验现象 344
第 21 章 消息队列 346
21.1 消息队列的基本概念 346
21.2 消息队列的运作机制 347
21.3 消息队列的阻塞机制 348
21.4 消息队列的应用场景 349
21.5 消息队列控制块 349
21.6 常用消息队列的函数讲解 350
21.6.1 消息队列创建函数 rt_mq_create() 350
21.6.2 消息队列删除函数 rt_mq_delete() 354
21.6.3 消息队列发送消息函数 rt_mq_send() 356

21.6.4 消息队列接收消息函数 rt_mq_recv() 360
21.7 消息队列使用注意事项 366
21.8 消息队列实验 367
21.9 实验现象 372
第 22 章 信号量 374
22.1 信号量基本概念 374
22.2 二值信号量的应用场景 375
22.3 二值信号量的运作机制 376
22.4 计数型信号量的运作机制 377
22.5 信号量控制块 378
22.6 常用信号量函数接口讲解 379
22.6.1 信号量创建函数 rt_sem_create() 379
22.6.2 信号量删除函数 rt_sem_delete() 382
22.6.3 信号量释放函数 rt_sem_release() 384
22.6.4 信号量获取函数 rt_sem_take() 386
22.7 信号量实验 391
22.7.1 二值信号量同步实验 391
22.7.2 计数信号量实验 396
22.8 实验现象 401
22.8.1 二值信号量同步实验现象 401
22.8.2 计数信号量实验现象 402
第 23 章 互斥量 404
23.1 互斥量的基本概念 404
23.2 互斥量的优先级继承机制 405
23.3 互斥量的应用场景 408
23.4 互斥量的运作机制 408
23.5 互斥量控制块 410
23.6 信号量函数接口讲解 411
23.6.1 互斥量创建函数 rt_mutex_create() 411
23.6.2 互斥量删除函数 rt_mutex_delete() 413
23.6.3 互斥量释放函数 rt_mutex_release() 415

23.6.4 互斥量获取函数 rt_mutex_take() 419
23.7 互斥量使用注意事项 425
23.8 互斥量实验 426
23.9 实验现象 431
第 24 章 事件 432
24.1 事件的基本概念 432
24.2 事件的应用场景 433
24.3 事件的运作机制 434
24.4 事件控制块 436
24.5 事件函数接口讲解 436
24.5.1 事件创建函数 rt_event_create() 436
24.5.2 事件删除函数 rt_event_delete() 439
24.5.3 事件发送函数 rt_event_send() 441
24.5.4 事件接受函数 rt_event_recv() 445
24.6 事件实验 451
24.7 实验现象 456
第 25 章 软件定时器 458
25.1 软件定时器的基本概念 458
25.2 软件定时器的应用场景 460
25.3 软件定时器的精度 460
25.4 软件定时器的运作机制 460
25.5 定时器超时函数 464
25.6 软件定时器的使用 469
25.6.1 软件定时器的创建函数 470
25.7 软件定时器实验 473
25.8 实验现象 477
第 26 章 邮箱 478
26.1 邮箱的基本概念 478
26.2 邮箱的运作机制 479
26.3 邮箱的应用场景 480
26.4 邮箱的应用技巧 480

26.5 邮箱控制块 482
26.6 邮箱的函数接口讲解 483
26.6.1 邮箱创建函数 rt_mb_create() 483
26.6.2 邮箱删除函数 rt_mb_delete() 486
26.6.3 邮箱邮件发送函数 rt_mb_send_wait()（阻塞） 488
26.6.4 邮箱邮件发送函数 rt_mb_send ()（非阻塞） 495
26.6.5 邮箱邮件接收函数 rt_mb_recv() 498
26.7 邮箱的实验 504
26.8 邮箱的实验现象 510
第 27 章 内存管理 511
27.1 内存管理的基本概念 511
27.2 内存管理的运作机制 513
27.2.1 静态内存管理 513
27.2.2 动态内存管理 514
27.3 内存管理的应用场景 518
27.4 静态内存管理的函数接口讲解 519
27.4.1 静态内存控制块 519
27.4.2 静态内存创建函数 rt_mp_create() 520
27.4.3 静态内存删除函数 rt_mp_delete() 525
27.4.4 静态内存初始化函数 rt_mp_init() 528
27.4.5 静态内存申请函数 rt_mp_alloc() 532
27.4.6 静态内存释放函数 rt_mp_free() 538
27.5 动态内存管理的函数接口讲解 541
27.5.1 系统堆内存初始化 rt_system_heap_init() 542
27.5.2 系统堆内存申请函数 rt_malloc() 546
27.5.3 系统堆内存释放函数 rt_free() 554
27.6 内存管理的实验 559
27.6.1 静态内存管理实验 559
27.6.2 动态内存管理实验 564
27.7 内存管理的实验现象 569
27.7.1 静态内存管理实验现象 569

27.7.2 静态内存管理实验现象 570
第 28 章 中断管理 571
28.1 异常与中断的基本概念 571
28.1.1 中断的介绍 572
28.1.2 和中断相关的名词解释 573
28.2 中断管理的运作机制 573
28.3 中断延迟的概念 575
28.4 中断管理的应用场景 576
28.5 中断管理讲解 577
28.6 中断管理的实验 580
28.7 中断管理的实验现象 589
第 29 章 双向链表 590
29.1 双向链表的基本概念 590
29.2 双向链表的函数接口讲解 590
29.2.1 链表初始化函数 rt_list_init() 591
29.2.2 向链表中插入节点 593
29.2.3 从链表删除节点函数 rt_list_remove() 596
29.3 双向链表的实验 597
29.4 双向链表的实验现象 603
第 30 章 CPU 利用率统计 605
30.1 CPU 利用率的基本概念 605
30.2 CPU 利用率的作用 605
30.3 CPU 利用率统计 606
30.4 CPU 利用率实验 612
30.5 CPU 利用率实验现象 616
第 31 章 rst 投稿示例文件 618
如何参与项目 619
项目源码 619
文档在线阅读与 pdf 619
常见问题 620
版权说明 621

关于本项目

本项目通过 git 开源：

- github 仓库地址：https://github.com/Embedfire-rtthread/ebf_rtthread_tutorial_rt1052.git
- gitee 仓库地址：https://gitee.com/Embedfire-rtthread/ebf_rtthread_tutorial_rt1052.git

点击右侧链接可在线阅读本项目文档：《RT-Thread 内核实现与应用开发实战—基于 i.MX RT 》

本书主要面向有 MCU 开发经验，希望从零开始学习 rt-thread 系统开发的嵌入式软件工程师及在校学生。

关于野火

开源共享，共同进步

野火在发布第一块 STM32 开发板之初，就喊出 **开源共享，共同进步**的口号，把代码和文档教程都免费提供给用户下载，而我们也一直把这个理念贯穿至今。

目前我们的产品已经包括 *STM32*、*i.MX RT* 系列、*GD32V*、*FPGA*、*Linux*、*emXGUI*、操作系统、网络、下载器等分支，覆盖电子工程应用领域的各种常用技术，其中教学类产品的代码和文档一直保持着开源的姿态发布到网络上，为电子工程师排忧解难，让嵌入式没有难用的技术是我们最大的愿望。

联系方式

- 官网：http://www.embedfire.com
- 论坛：http://www.firebbs.cn
- github 主页：https://github.com/Embedfire
- gitee 主页：https://gitee.com/wildfireteam
- 淘宝：https://yehuosm.tmall.com
- 邮箱：embedfire@embedfire.com
- 电话：0769-33894118

第 1 章　推荐序

RT-Thread 从 2006 年走来，已经经过了十多个年头。早年就和野火相识于网络，记得那个时候野火还没那么火（才开始做开发板），RT-Thread 也还只是国内一个小众的 RTOS。直至 2018 年年中，野火过来上海才有缘相见，互道年轻。

在 2017 年底，欣闻野火有著 RT-Thread 书籍的计划，非常欣喜、感谢。这将是 RT-Thread 的首本书籍，在 RT-Thread 的基础生态推广上有着举足轻重的作用！不仅如此，作为 RT-Thread 的官方合作伙伴，野火也在其全系列的 STM32 开发板上适配了 RT-Thread 例程并配套了手把手的教程。

野火的这本书籍，第一部分先从 0 到 1 的方式，从最底层的汇编开始，一步一步构造出一个 RT-Thread 操作系统内核，向大家揭示了任务如何定义、任务如何切换，也讲解了任务的延时如何实现，讲解了如何支持多优先级，讲解了如何实现定时器和讲解如何实现时间片等 RT- Thread 操作系统的核心知识点；第二部分，则讲解 RT-Thread 内核设施的应用，使得大家学习 RT-Thread 不再困难，使用 RT-Thread 不再困难。

整本书由浅入深，层层叠加，与初学者的入门路径完全吻合，是学习 RT-Thread 物联网操作系统的不二之选书籍。同时整本书也兼顾深度，对于要了解操作系统内核原理的使用者，也非常值得一看。

RT-Thread 创始人

本书电子版和配套的实验源码均可到野火电子 / RT-Thread / 芯片之家公众号下载。

野火电子公众号

RT-Thread 公众号

芯片之家公众号

第 2 章　前言

2.1 关于本书

本书是野火嵌入式教学丛书“RTOS 内核实现与应用开发实战指南的”RT-Thread 版本，其中机械工业出版社出版的《FreeRTOS 内核实现与应用开发实战指南—基于 i.MX RT》是该书的姐妹篇。因为两本书均是写 RTOS 的书籍，只是针对的 RTOS 对象不一样，所以书中凡是涉及到 RTOS 抽象层的理论部分均会有很多相同的地方，具体的不同就体现为这两个 RTOS 的实现原理和代码层面上。

本书分为两个部分，第一部分为“从 0 到 1 叫你写 RT- Thread 内核”，该部分的章节名字与其姐妹篇《FreeRTOS 内核实现与应用开发实战指南—基于 i.MX RT》的章节名字基本类似，均采取循序渐进，层层叠加的方式来编排章节的序号和名字，但是在具体的内容上是不一样的，因为针对的 RTOS 不一样。其中“新建 RT-Thread 工程—软件仿真”和“裸机系统与多任务系统”这两章属于理论的抽象层，不具体针对哪一个 RTOS，所以基本一样。

本书第二部分为“RT-Thread 内核应用开发”，主要讲解 RT-Thread 内核组件的源码实和 API 的使用，同样在章节命名和排序上也与该书的姐妹篇《FreeRTOS 内核实现与应用开发实战指南—基于 i.MX RT》类似。对于什么是任务、阻塞延时和信号量的应用等这些 RTOS 的共同抽象层的理论讲解也基本类似，但是具体涉及到这两个 RTOS 的原理实现和代码讲解的时候是完全不一样的，往往这部分的知识点才是读者要真正关心的内容。

我把这两本书归为姐妹篇是因为两书的编写风格、内容框架和章节命名与排序基本一致，语言阐述也类似，且涉及到 RTOS 抽象层的理论部分均相同，但不同的是凡是涉及到 RTOS 的实现原理、内核源码的讲解和上层 API 使用时，均是不一样的，而且这些内容才是本书的重点部分，是读者朋友关心和学习的核心。两本书又无法合并在一起讲解，所以分为姐妹篇两本书。

如果读者已经学习了其中一本书，再学习另外一本的话，那么涉及到 RTOS 抽象层的理论部分可跳过，只需把精力放在 RTOS 内核的实现和源码 API 的应用方面。因为现有的 RTOS，在理论层基本都是想通的，但在具体的码实现上又各有不同且各有特点，所以两书可以作为互补学习，只要掌握了其中一本的知识，再学习另外一本定会得心应手，如有神助。

2.2 如何学习本书

本书是首本系统讲解 RT-Thread 的中文书籍，共分为两个部分，第一部分为“从 0 到 1 教你写 RT-Thread 内核”，即重点讲解 RT-Thread 的原理实现，从 0 开始，不断迭代，教你怎么把 RT-Thread 的内核写出来，让你彻底学会线程是如何定义的，系统是如何调度的（包括底层的汇编代码讲解），多优先级是如何实现的等等操作系统最深层次的知识。当你拿到本书开始学习的时候你一定会惊讶，原来 RTOS 的学习并没有那么复杂，反而是那么的有趣，原来自己也可以写 RTOS，成就感立马爆棚。

当第一部分知识你彻底掌握好之后，再切换到其它 RTOS 的学习，那简直就是易如反掌，纵观现在市面上流行的几种 RTOS，它们的内核的实现基本都差不多，只需要深入研究其中一种即可，没有必要每一种 RTOS 都深入的研究源码，如果你时间允许，看看那也无所谓。第二部分是“RT-Thread 内核应用开发”，重点讲解 RT-Thread 的移植，内核每个组件的应用，比起第一部分，这部分内容掌握起来比较容易。

全书内容循序渐进，不断迭代，前一章都是后一章的基础，必须从头开始阅读，不能进行跳跃式的阅读。在学习的时候务必做到两点：一是不能一味地看书，要把代码和书本结合起来学习，一边看书，一边调试代码。看书倒是很简单，那如何调试代码？即单步执行每一条程序，看看程序的执行流程和执行的效果与自己大脑所想是不是一样；二是在每学完一章之后，必须将配套的例程重写一遍（切记不要复制，哪怕是一个分号，但可以抄），做到举一反三，确保真正理解。在自己写的时候肯定会错漏百出，这个时候要珍惜这些错误，好好调试，这是你提高编程能力的最好的机会。记住，程序不是一气呵成写出来的，而是一步一步调试出来的。

2.3 本书的参考资料

1. RT-Thread 官方源代码
2. RT-Thread_manual.zh.pdf（电子版）
3. 《i.MX RT 库开发实战指南》（电子版）

2.4 本书的编写风格

本书第一部分以 RT-Thread Nano 3.0.3 官方源码为蓝本，抽丝剥茧，不断迭代，教你怎么从 0 开始把 RT-Thread 内核写出来。书中涉及到的数据类型，变量名称、函数名称，文件名称，文件存放的位置都完全按照 RT-Thread 官方的方式来实现，当你学完这本书之后可以无缝地切换到原版的 RT-Thread 的使用。要注意的是，在实现的过程中，某些函数我会去掉一些形参和一些冗余的代码，只保留核心的功能，但这并不会影响我们学习。

本书第二部分主要讲 RT-Thread 的移植和内核组件的使用，不会再去深入讲解源码，着重讲应用，如果对第一部分不感兴趣的朋友，可以跳过第一部分，直接进入第二部分的学习，两者之间没有必然的联系。

2.5 本书的配套硬件

本书支持野火 i.MX RT 开发板全套系列，具体型号见表格 1-1，具体图片见图 1-1、图 1-2、图 1-3、图 1-4。学习的时候如果配套这些硬件平台做实验，学习必会达到事半功倍的效果，可以省去中间硬件不一样时移植遇到的各种问题。

表格 1-1 野火 i.MX RT 开发板型号汇总

型号	区别		
•	内核	引脚	RAM
RT1021	Cortex-M7	144	256KB
RT1052	Cortex-M7	196	512KB

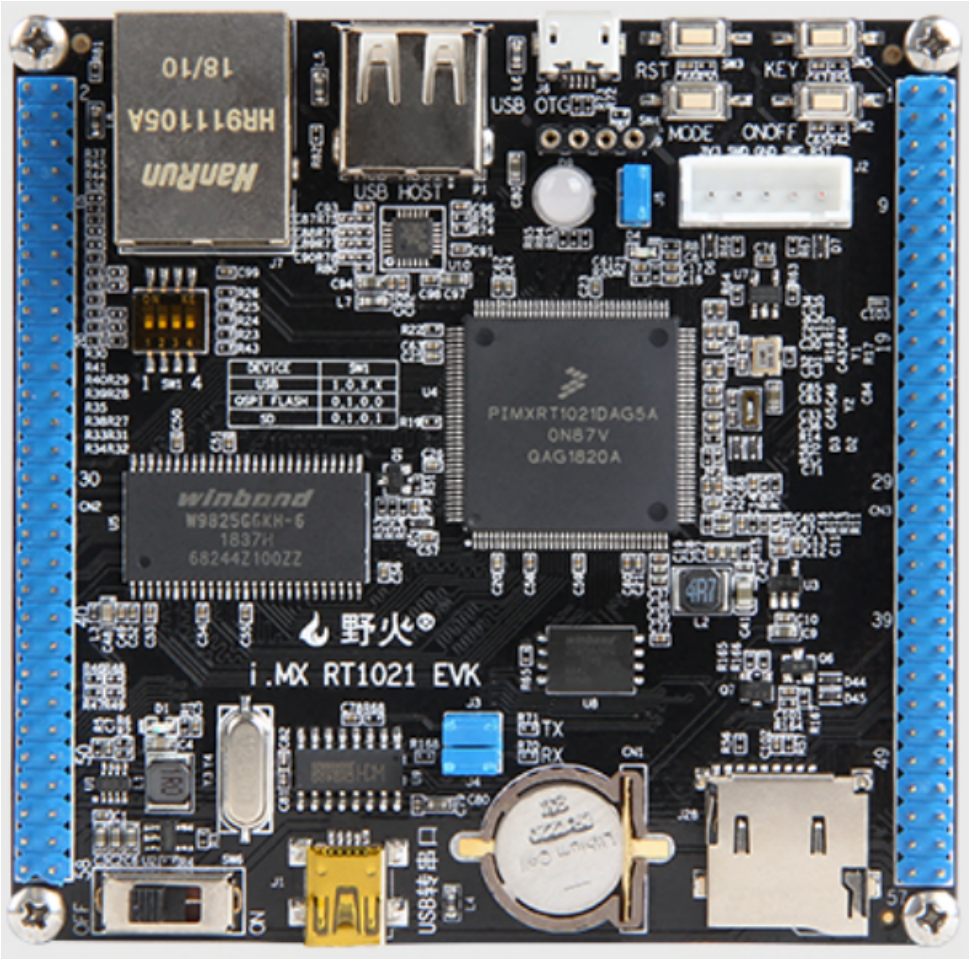


图 1-1 野火【i.MX RT1021 EVK 】开发板

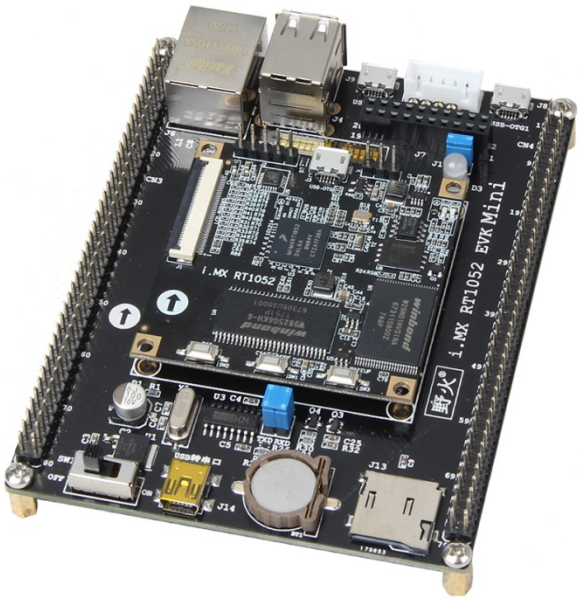


图 1-2 野火【i.MX RT1052 MINI】开发板

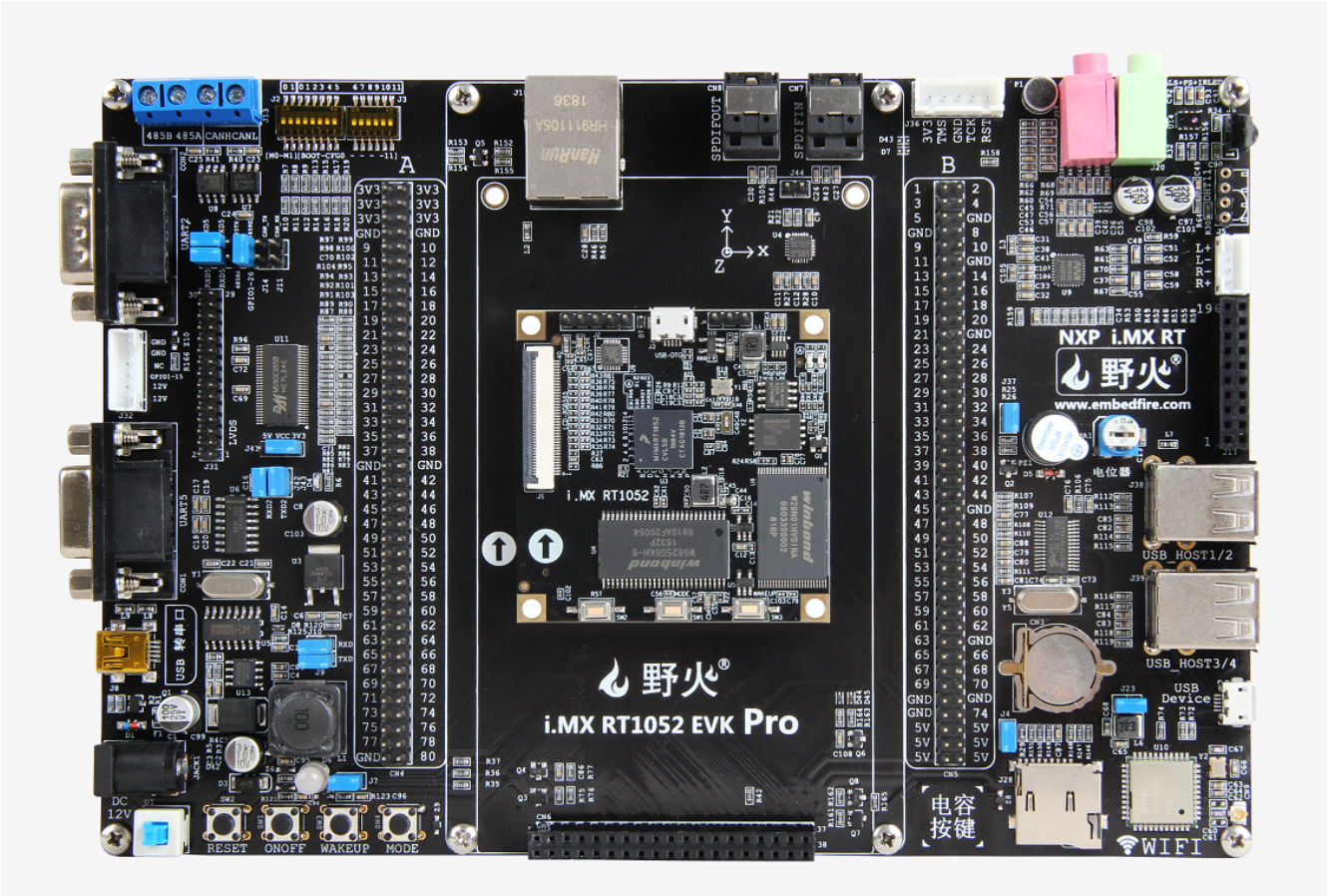


图 1-3 野火【i.MX RT1052 Pro】开发板

图 1-4 野火【i.MX RT1052 邮票孔】核心板

2.6 本书的技术论坛

如果在学习过程中遇到问题，可以到野火电子论坛：www.firebbs.cn 发帖交流 ，开源共享，共同进步。

鉴于水平有限，本书难免有纰漏，热心的读者也可把勘误发到论坛好让我们改进做得更好，祝您学习愉快，RT-Thread 的世界，野火与您同行。

TODO 和悬赏任务

本页记录待改进的事项，以后会在此页发布任务清单，完成任务可获得 **奖励**，敬请期待！

如果您想参与到项目，可按 如何参与野火开源项目 的说明操作。

- 格式问题
  - 代码改成使用 rst 的格式
  - 图片的引用需要修改
  - 把图片改成居中形式，去掉缩放
- 悬赏任务

若想领取悬赏任务，请发送要参与的任务和预计时间到邮箱：embedfire@embedfire.com。

第 3 章　如何学习 RTOS

3.1 为什么要学习 RTOS

当我们进入嵌入式这个领域的时候，往往首先接触的都是单片机编程，单片机编程又首选 51 单片机来入门。这里面说的单片机编程通常都是指裸机编程，即不加入任何 RTOS（Real Time Operation System 实时操作系统）的程序。常用的 RTOS 有国外的 FreeRTOS、μC/OS、RTX 和国内的 RT-Thread、Huawei LiteOS 和 AliOS-Things 等，其中尤以国外开源且免费的 FreeRTOS 的市场占有率最高。如今国产的 RT-Thread 经过 10 来年的发展，声势迅猛，在国产 RTOS 中占据鳌头。

在裸机系统中，所有的程序基本都是自己写的，所有的操作都是在一个无限的大循环里面实现。现实生活中的很多中小型的电子产品用的都是裸机系统，而且也能够满足需求。但是为什么还要学习 RTOS 编程，偏偏还要整个操作系统进来。一是项目需要，随着产品要实现的功能越来越多，单纯的裸机系统已经不能够完美地解决问题，反而会使编程变得更加复杂，如果想降低编程的难度，我们可以考虑引入 RTOS 实现多线程管理，这是使用 RTOS 的最大优势。二是学习的需要，必须学习更高级的东西，实现更好的职业规划，为将来走向人生巅峰迎娶白富美做准备，而不是一味的在裸机编程上面死磕。作为一个合格的嵌入式软件工程师，学习是永远不能停歇的事，时刻都得为将来准备。书到用时方恨少，我希望机会来临时你不要有这种感觉。

为了帮大家理清 RTOS 编程的套路，我们会在“裸机系统与多线程系统”章节简单的分析下这两种编程方式的区别，这个区别我称它为学习 RTOS 的命门，只要打通这个任督二脉，以后的 RTOS 学习可以说是易如反掌。在讲解这两种编程方法的区别的时候，我们主要讲方法论，不会涉及到具体的代码编程，主要还是通过伪代码来讲解。

3.2 如何学习 RTOS

裸机编程和 RTOS 编程的风格有些不一样，而且有很多人说 RTOS 的学习很难，这就导致学习的人一听到 RTOS 编程就在心理面忌惮三分，结果就是出师未捷身先死。

那么到底如何学习一个 RTOS？最简单的就是在别人移植好的系统之上，看看 RTOS 里面的 API 使用说明，然后调用这些 API 实现自己想要的功能即可。完全，不用关心底层的移植，这是最简单快速的入门方法。这种方法各有利弊，如果是做产品，好处是可以快速的实现功能，将产品推向市场，赢得先机，弊端是当程序出现问题的时候，因对这个 RTOS 不够了解，会导致调试困难，焦头烂额，一筹莫展。如果是学习，那么只会简单的调用 API，那是不可取的，我们应该深入的学习其中一款 RTOS。

目前市场上现有的 RTOS，它们的内核实现方式都差不多，我们只需要深入学习其中一款就行。万变不离其宗，以后换到其它型号的 RTOS，使用起来，那自然是得心应手。那如何深入的学习一款 RTOS？这里有一个最有效也是最难的方法，就是阅读 RTOS 的源码，深究内核和每个组件的实现方式，这个过程枯燥且痛苦。但为了能够学到 RTOS 的精华，你不入地狱谁入地狱？

市面上虽然有一些讲解相关 RTOS 源码的书，如果你基础不够，且先前没有使用过该款 RTOS，那么源码看起来还是会非常枯燥，且并不能从全局掌握整个 RTOS 的构成和实现。

现在，我们采用一种全新的方法来教大家学习一款 RTOS，即不是单纯的讲里面的 API 如何使用，也不是单纯的拿里面的源码一句句来讲解。而是，从 0 开始，层层叠加，不断地完善，教大家怎么把一个 RTOS 从 0 到 1 写出来，让你在每一个阶段都能享受到成功的喜悦。在这个 RTOS 实现的过程中，只需要你具备 C 语言的基础就行，然后就是跟着野火这个教程笃定前行，最后定有所成。

3.3 选择什么 RTOS

这个用来教学的 RTOS，我们不会完全自己写一个，不会再重复的造轮子，而是选取目前国内最红火的国产 RT-Thread 为蓝本，将其抽丝剥茧，层层叠加，从 0 到 1 写出来。在实现的过程中，数据类型、变量名、函数名称、文件类型等都完全按照 RT-Thread 里面的写法，不会自己再重新命名。这样学完我们这个课程之后，就可以无缝地过度到 RT-Thread 的使用。

第 4 章　初识 RT-Thread

4.1 RT-Thread 简介

4.1.1 版权

RT-Thread 版权属于上海睿赛德电子科技有限公司，于 2006 年 1 月首次发布，初始版本号为 0.1.0，经过 10 来年的发展，如今主版本号已经升级到 3.0，累计开发者达到数百万，在各行各业产品中装机量达到了惊人的 2000 多万，占据国产 RTOS 的鳌头。

4.1.1.1 收费问题

RT-Thread 是一款“开源免费”的实时操作系统，遵循的是 GPLv2+ 的许可协议。这里说到的开源，指的是你可以免费得获取到 RT-Thread 的源代码，且当你的产品使用了 RT-Thread 且没有修改 RT-Thread 内核源码的时候，你的产品的全部代码都可以闭源，不用开源，但是当你修改了 RT-Thread 内核源码的时候，就必须将修改的这部分开源，反馈给社区，其它应用部分不用开源。免费的意思是无论你是个人还是公司，都可以免费地使用，不需要掏一分钱。

4.2 RT-Thread 的意义

不知你是否发现，在 RTOS 领域，我们能接触到的实时操作基本都来自国外，很少见到有国内的厂家在 RTOS 领域崭露头角。从早年最火的 μC/OS，到如今市场占有率最高的 FreeRTOS，到获得安全验证最多的 RTX（KEIL 自家的），再到盈利能力最强的 ThreadX，均来自国外。可偏偏在 10 几年前，在中国，有一个天赋异禀，倔强不屈的极客，他叫熊谱翔，编写了 RT-Thread 初代内核，并联合中国开源社区的极客不断完善，推陈出新，经过 10 几年的发展，如今豪据国产 RTOS 的鳌头，每年递增数十万的开发者，加上如今 AI 和物联网等风口，让 RT-Thread 有一统江湖之

势，从今年完成 A 轮数百万美元的融资就可以看出，在未来不出 5 年，RT-Thread 将是你学习和做产品的不二之选。

那 RT-Thread 的意义到底是啥？RT-Thread 来自中国，让我们看到了国内的技术开发者也能写出如此优秀的 RTOS，并不是什么都是外国的好。以我们 10 多年搞电子的经验来看，在以往接触到很多国内的开发者，一看到一个稍微高大上的东西，就问是不是外国的，如果是国产的，就问是不是参考了外国的，很多人骨子里头天生就有这种软骨病，这种病很难治，有一种浑然天生的感觉，如今 RT-Thread 就是治疗这种软骨病的一副良药，这是我个人认为 RT-Thread 的最大意义。当然，RT-Thread 作为一款国产的物联网操作系统，简单易用，低功耗设计，组件丰富等特性也将让 RT-Thread 大放异彩。野火，作为一个国内嵌入式的教育品牌，能为国产 RTOS 出一份力也是我们的荣幸，希望这本书能够帮助大家快速的入门和掌握 RT-Thread。

第 5 章　第一部分：从 0 到 1 教你写 RT-Thread 内核

本书第一部分以 RT-Thread Nano 为蓝本，抽丝剥茧，不断迭代，教大家怎么从 0 开始把 RT-thread 写出来。这一部分是着重讲解 RTOS 怎么实现的过程，当你学完这部分之后，再来重新使用 RT-Thread 或者其它 RTOS，那将会得心应手，不仅知其然，而且知其所以然。在源码实现的过程中，涉及到的数据类型，变量名称，函数名称，文件名称以及文件的存放目录都会完全按照 RT-Thread 的来实现，有些不必要的代码我会剔除，但并不会影响我们理解整个 OS 的功能。

这部分每一章都是前一章的基础，环环相扣，逐渐为你揭开 OS 的神秘面纱，读起来会有一种豁然开朗的感觉。如果你再稍微上进一点，把代码都自己敲一遍，然后再仿真的时候出来的效果要是跟我的一样的话，那从心里油然而生的成就感简直就要爆棚，恨不得一下子把本书读完，真是让人看了还想看，读了还想读，流连忘返之际，你不禁会歇斯底里大喊一句：火哥，我爱你。

第 6 章　新建 RT-Thread 工程—软件仿真

在开始写 RT-Thread 内核之前，我们先新建一个 RT-Thread 的工程，Device 选择 Cortex-M3（Cortex-M4 或 Cortex-M7）内核的处理器，调试方式选择软件仿真，然后我们再开始一步一步地教大家把 RT-Thread 内核从 0 到 1 写出来，让大家彻底搞懂 RT-Thread 的内部实现和设计的哲学思想。最后我们再把 RT-Thread 移植到野火 i.MX RT 开发板上，到了最后的移植其实已经非常简单，只需要换一下启动文件和添加 bsp 驱动就行。

6.1 新建本地工程文件夹

在开始新建工程之前，我们先在本地电脑端新建一个文件夹用于存放工程。文件夹名字我们取为“新建 RT-Thread 工程—软件仿真”（名字可以随意取），然后再在该文件夹下面新建各个文件夹和文件，有关这些文件夹的包含关系和作用具体见表格 4-1。

表格 4-1 工程文件夹根目录下的文件夹的作用

文件夹名称	文件夹	文件夹作用
Doc		用于存放对整个工程的说明文件，如 readme.txt。通常情况下，我们都要对整个工程实现的功能，如何编译，如何使用等做一个简要的说明。
libraries		存放库文件
Project		用于存放新建的工程文件。
rt-thread/3.0.3	bsp	存放板级支持包，暂时为空。
	components/finsh	存放 RT-Thread 组件，暂时未空。
	include	存放头文件，暂时为空。
	include/libc	
	libcpu/arm/cortex-m0	存放与处理器相关的接口文件，暂时为空。
	libcpu/arm/cortex-m3	
	libcpu/arm/cortex-m4	
	libcpu/arm/cortex-m7	
	src	存放 RT-Thread 内核源码，暂时为空。
User		存放 main.c 和其它的用户编写的程序，main.c 第一次使用需要用户自行新建。

6.2 使用 KEIL 新建工程

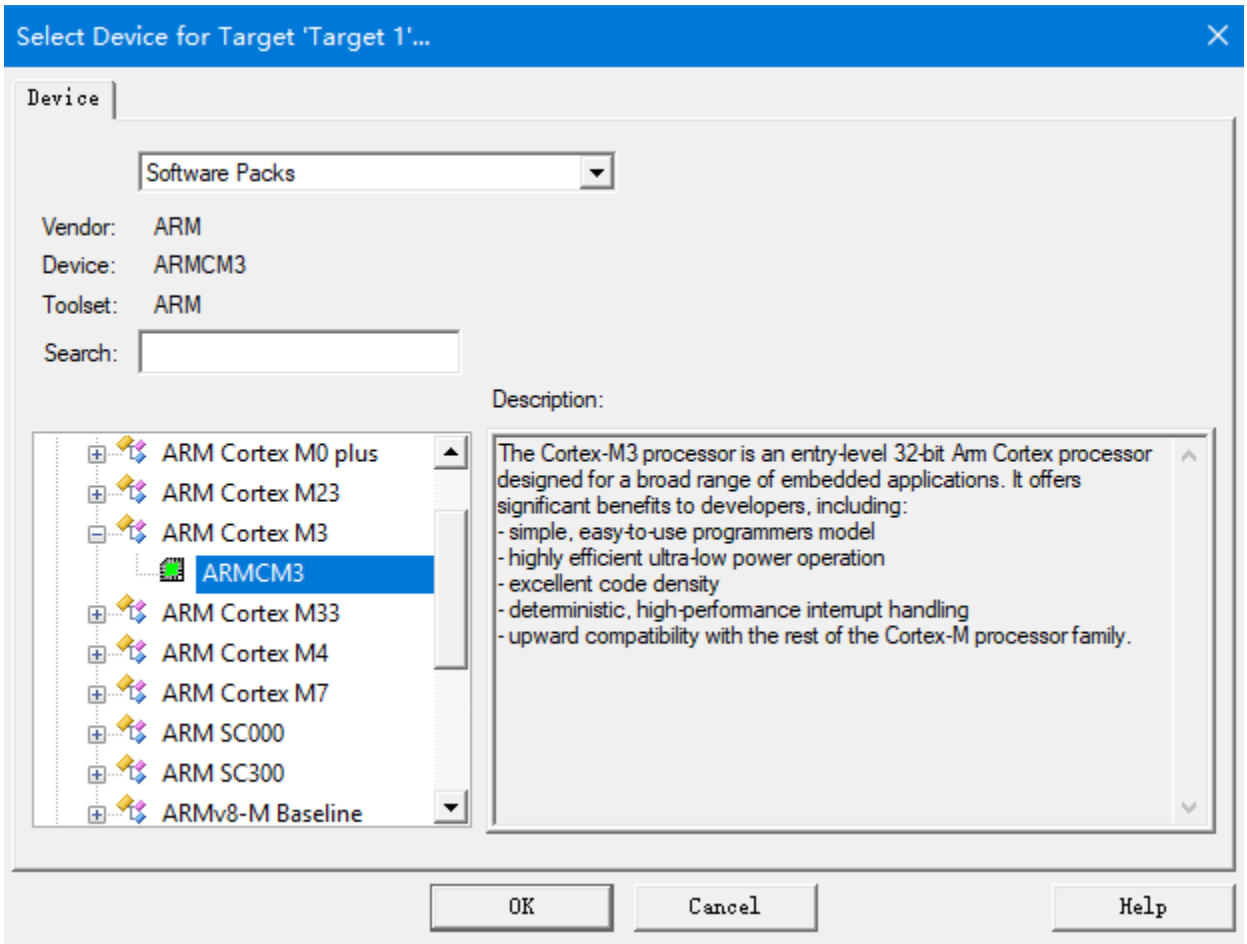
开发环境我们使用 KEIL5，版本为 5.23，高于或者低于 5.23 都行，只要是版本 5 就行。

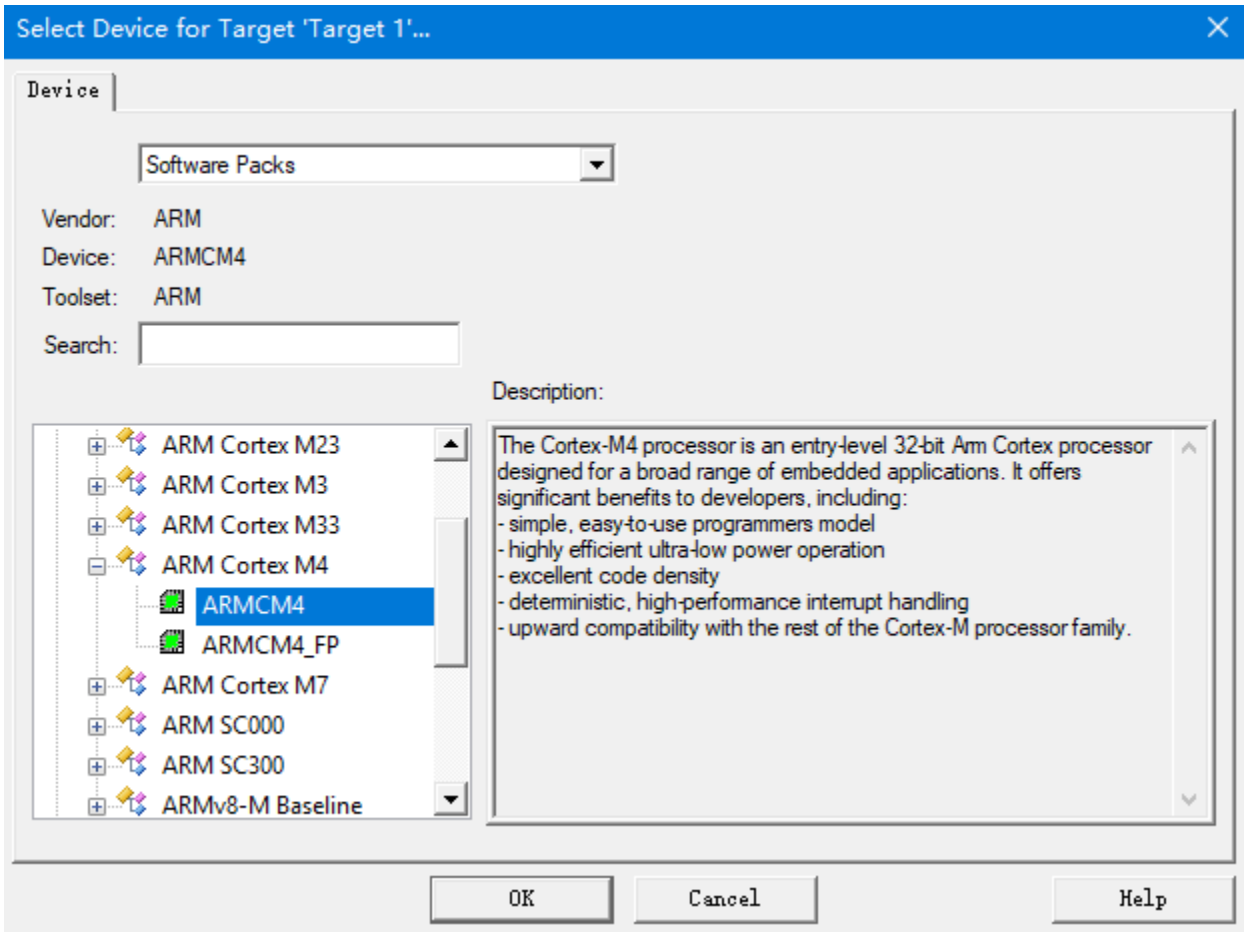
6.2.1 New Progect

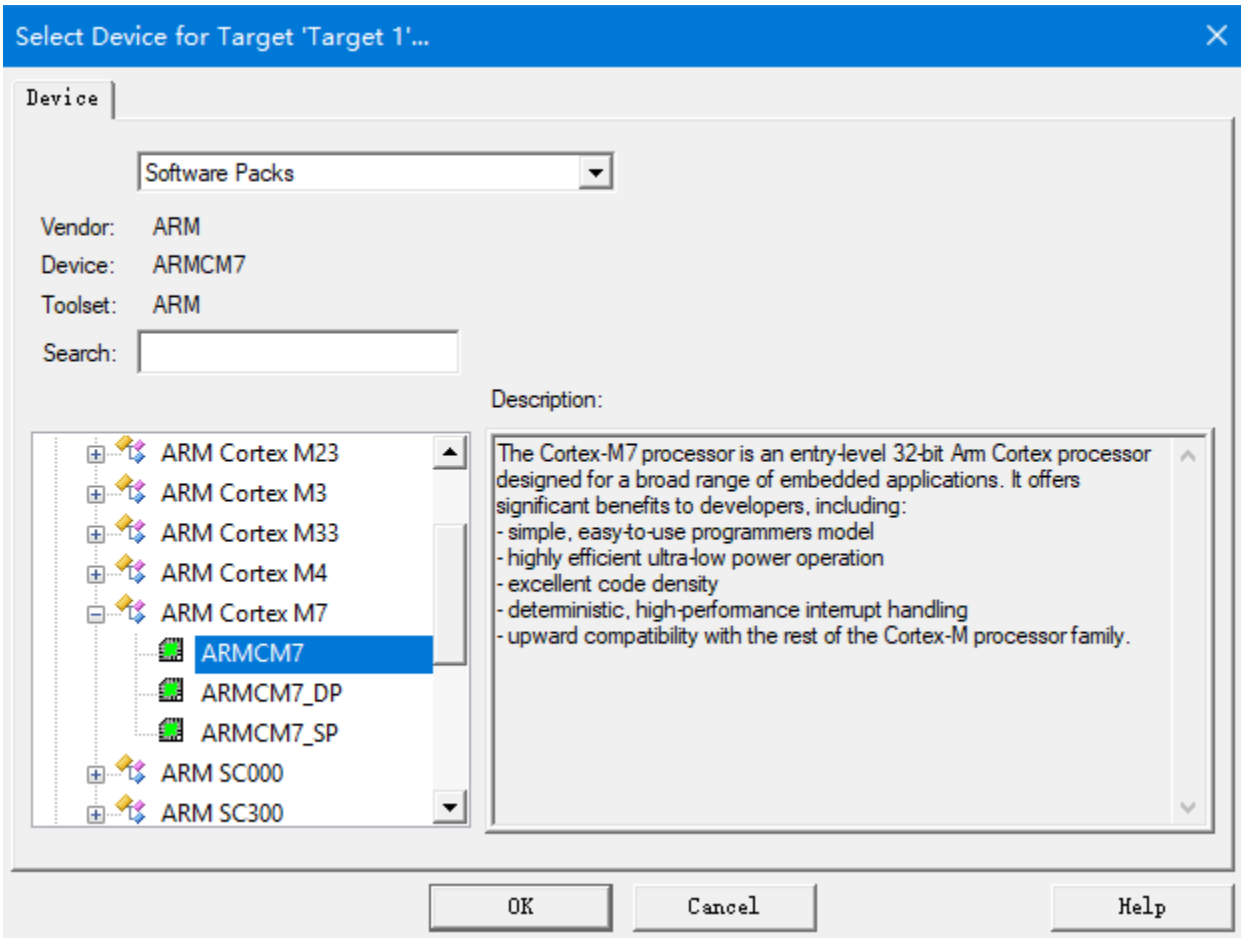
首先打开 KEIL5 软件，新建一个工程，工程文件放在目录 Project 下面，名称命名为 Fire_RT-Thread，其中 Fire 表示野火的意思，当然你也可以换成其它名称，但是必须是英文，不能是中文，切记。

6.2.2 Select Device For Target

当命名好工程名称，点击确定之后会弹出 Select Device for Target 的选项框，让我们选择处理器，这里我们选择 ARMCM7（ARMCM4 或 ARMCM3），具体见图 4-3_（图 *4-2* 或图 *4-1* ）。

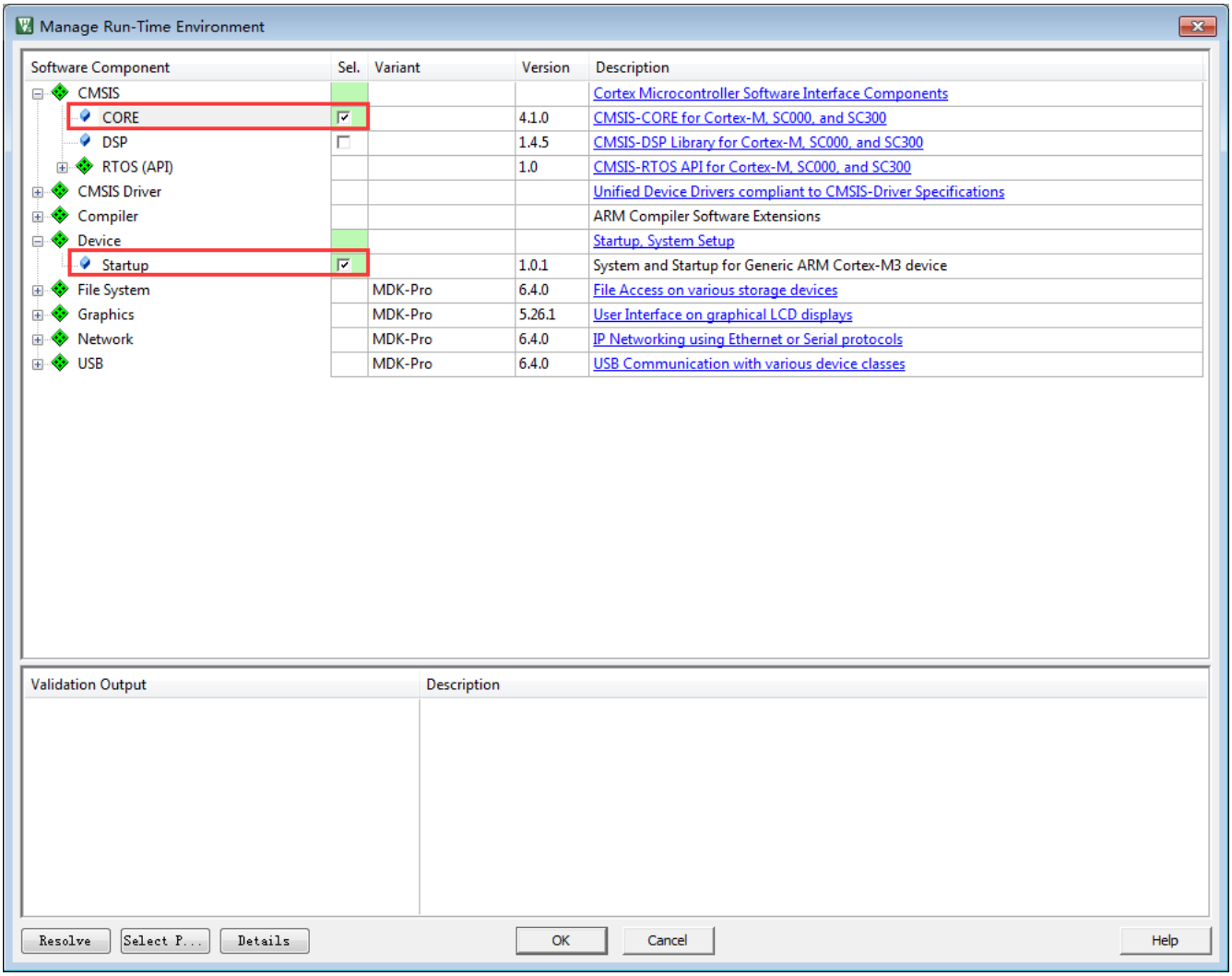

Select Device for Target 'Target 1'...
Device
Software Packs
Vendor: ARM
Device: ARMCM3
Toolset: ARM
Search:
Description:
ARM Cortex M0 plus
ARM Cortex M23
ARM Cortex M3
ARMCM3
ARM Cortex M33
ARM Cortex M4
ARM Cortex M7
ARM SC000
ARM SC300
ARMv8-M Baseline
The Cortex-M3 processor is an entry-level 32-bit Arm Cortex processor designed for a broad range of embedded applications. It offers significant benefits to developers, including:
- simple, easy-to-use programmers model
- highly efficient ultra-low power operation
- excellent code density
- deterministic, high-performance interrupt handling
- upward compatibility with the rest of the Cortex-M processor family.
OK
Cancel
Help


Select Device for Target 'Target 1'...
Device
Software Packs
Vendor: ARM
Device: ARMCM4
Toolset: ARM
Search:
Description:
ARM Cortex M23
ARM Cortex M3
ARM Cortex M33
ARM Cortex M4
ARMCM4
ARMCM4_FP
ARM Cortex M7
ARM SC000
ARM SC300
ARMv8-M Baseline
The Cortex-M4 processor is an entry-level 32-bit Arm Cortex processor designed for a broad range of embedded applications. It offers significant benefits to developers, including:
- simple, easy-to-use programmers model
- highly efficient ultra-low power operation
- excellent code density
- deterministic, high-performance interrupt handling
- upward compatibility with the rest of the Cortex-M processor family.
OK
Cancel
Help

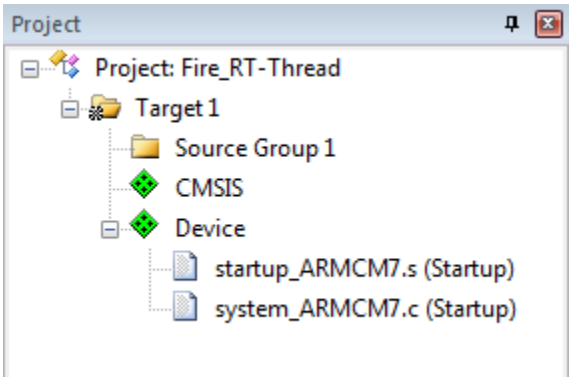


6.2.3 Manage Run-Time Environment

选择好处理器，点击 OK 按钮后会弹出 Manage Run-Time Environment 选项框。这里我们在 CMSIS 栏选中 CORE 和 Device 栏选中 Startup 这两个文件即可，具体见图 *4-4*。



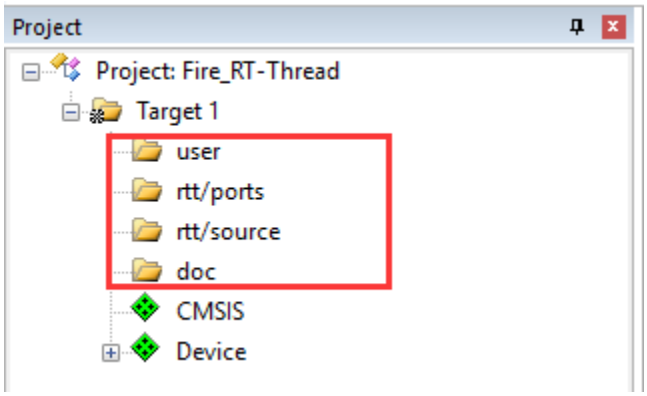
点击 OK，关闭 Manage Run-Time Environment 选项框之后，刚刚我们选择的 CORE 和 Startup 这两个文件就会添加到我们的工程组里面，具体见图 *4-5*。



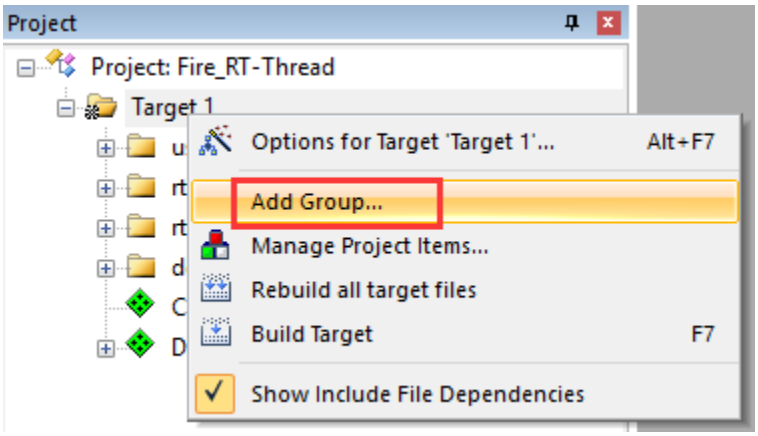
其实这两个文件刚开始都是存放在 KEIL 的安装目录下，当我们配置 Manage Run-Time Environment 选项框之后，软件就会把选中好的文件从 KEIL 的安装目录拷贝到我们的工程目录：ProjectRTEDeviceARMCM7（ARMCM4 或 ARMCM3）下面。其中 startup_ARMCM7.s（startup_ARMCM4.s 或 startup_ARMCM3.s）是汇编编写的启动文件，system_ARMCM7.c（startup_ARMCM4.c 或 startup_ARMCM3.c）是 C 语言编写的跟时钟相关的文件。更加具体的可直接阅读这两个文件的源码。只要是 Cortex-M7（ARMCM4 或 ARMCM3）内核的单片机，这两个文件都适用。

6.2.4 在 KEIL 工程里面新建文件组

在工程里面添加 user、rtt/ports、rtt/source 和 doc 这几个文件组，用于管理文件，具体见图 *4-6*。

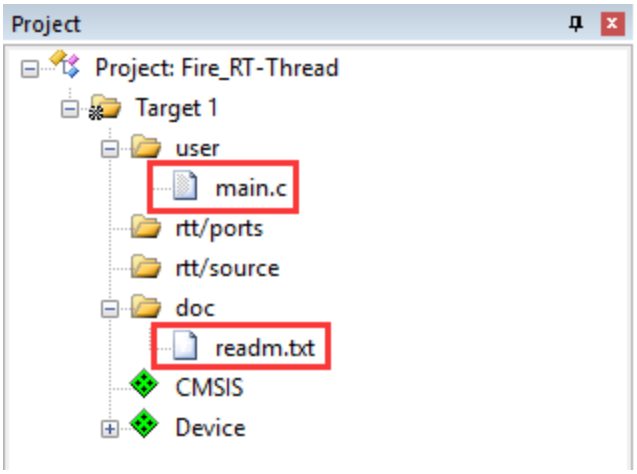


对于新手，这里有个问题就是如何添加文件组？具体的方法为鼠标右键 Target1，在弹出的选项里面选择 Add Group…即可，具体见图 *4-7*，需要多少个组就鼠标右击多少次 Target1。

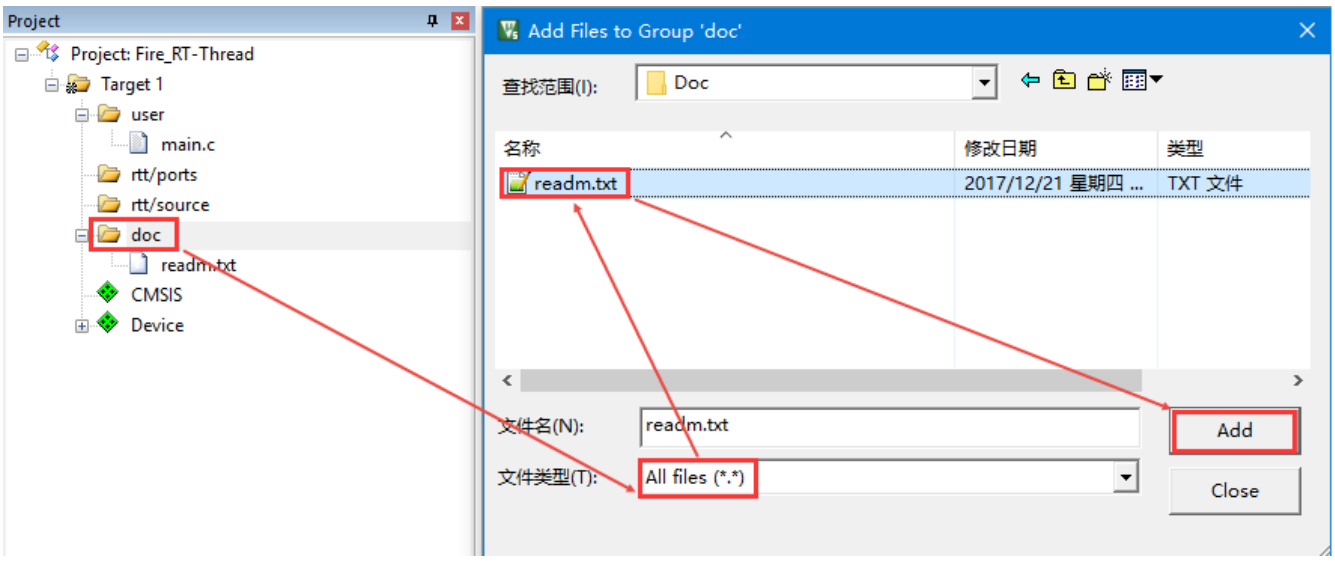


6.2.5 在 KEIL 工程里面添加文件

在工程里面添加好组之后，我们需要把本地工程里面新建好的文件添加到工程里面。具体为把 readme.txt 文件添加到 doc 组，main.c 添加到 user 组，至于 RT-Thread 相关的文件我们还没有编写，那么 RT-Thread 相关的组就暂时为空，具体见图 *4-8*。



对于新手，这里有个问题就是如何将本地工程里面的文件添加到工程组里里面？具体的方法为鼠标左键双击相应的组，在弹出的文件选择框中找到要添加的文件，默认的文件类型是 C 文件，如果要添加的是文本或者汇编文件，那么此时将看不到，这个时候就需要把文件类型选择为 All Files，最后点击 Add 按钮即可，具体见图 *4-9*。



6.3 编写 main 函数

一个工程如果没有 main 函数是编译不成功的，会出错。因为系统在开始执行的时候先执行启动文件里面的复位程序，复位程序里面会调用 C 库函数 __main，__main 的作用是初始化好系统变量，如全局变量，只读的，可读可写的等等。__main 最后会调用 __rtentry，再由 __rtentry 调用 main 函数，从而由汇编跳入到 C 的世界，这里面的 main 函数就需要我们手动编写，如果没有编写 main 函数，就会出现 main 函数没有定义的错误，具体见图 *4-10*。

```
Build Output
*** Using Compiler 'V5.06 update 4 (build 422)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Build target 'Target 1'
assembling startup_ARMCM3.s...
compiling main.c...
compiling system_ARMCM3.c...
linking...
.\Objects\YH-uCOS-III.axf: Error: L6218E: Undefined symbol main (referred from __rtentry2.o).
Not enough information to list image symbols.
Finished: 1 information, 0 warning and 1 error messages.
".\Objects\YH-uCOS-III.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:02
```

main 函数我们写在 main.c 文件里面，因为是刚刚新建工程，所以 main 函数暂时为空，具体见*代码清单 4-1*。

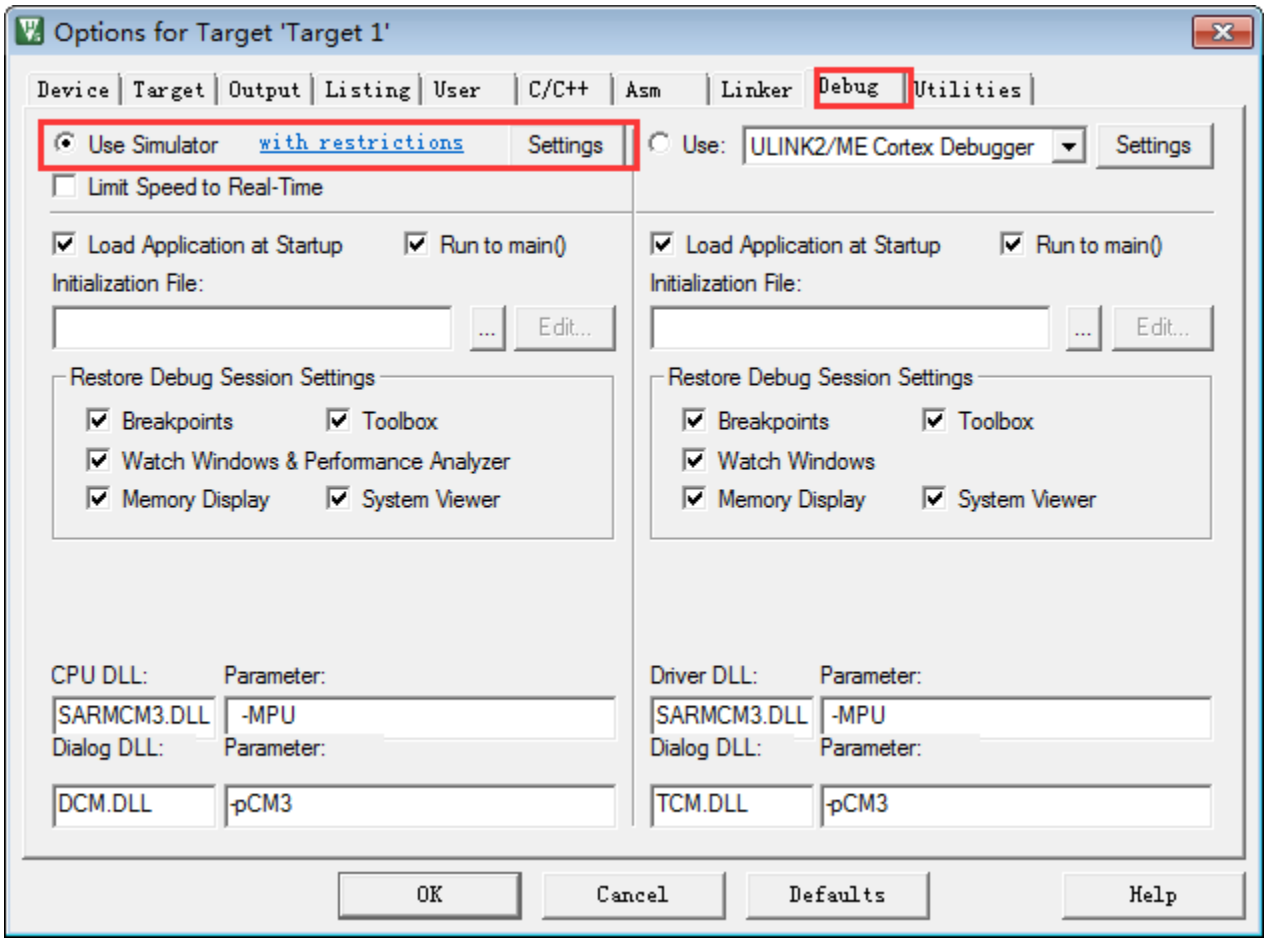
列表 1: 代码清单 4-1main 函数

```
int main(void)
{
    for (;;)
    {
        /* 啥事不干 */
    }
}
```

6.4 调试配置

6.4.1 设置软件仿真

最后，我们再配置下调试相关的配置即可。为了方便，我们全部代码都用软件仿真，即不需要开发板也不需要仿真器，只需要一个 KEIL 软件即可，有关软件仿真的配置具体见图 *4-11*。



6.4.2 修改时钟大小

在时钟相关文件 system_ARMCM7.c（system_ARMCM4.c 或 system_ARMCM3.c）的开头，有一段代码定义了系统时钟的大小为 25M，具体见*代码清单 4-2*。在软件仿真的时候，确保时间的准确性，代码里面的系统时钟跟软件仿真的时钟必须一致，所以 Options forTarget->Target 的时钟应该由默认的 12M 改成 25M，具体见图 *4-12*。

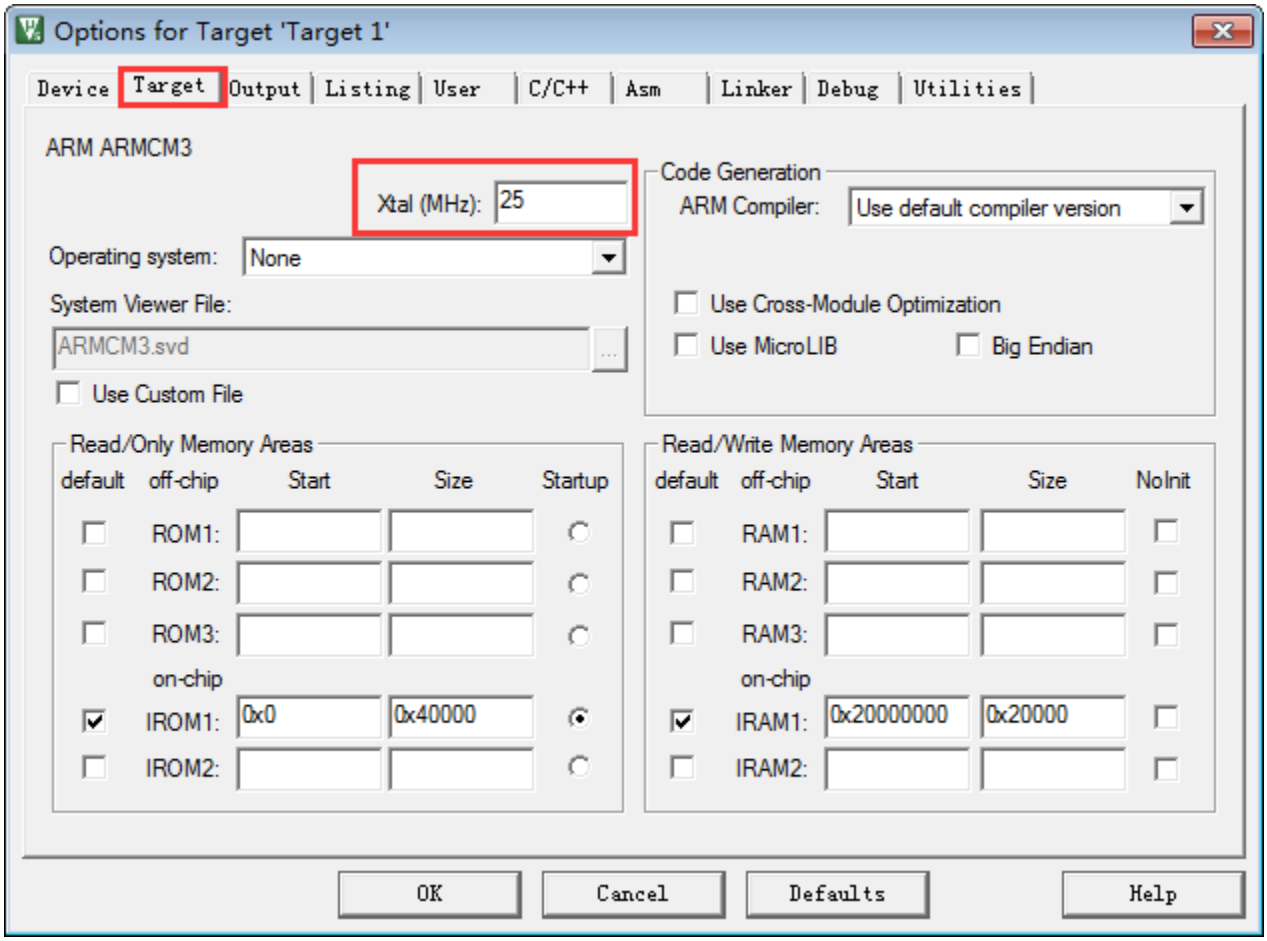
列表 2: 代码清单 4-2 时钟相关宏定义

```
#define __HSI ( 8000000UL)
#define __XTAL ( 5000000UL)
```

(下页继续)

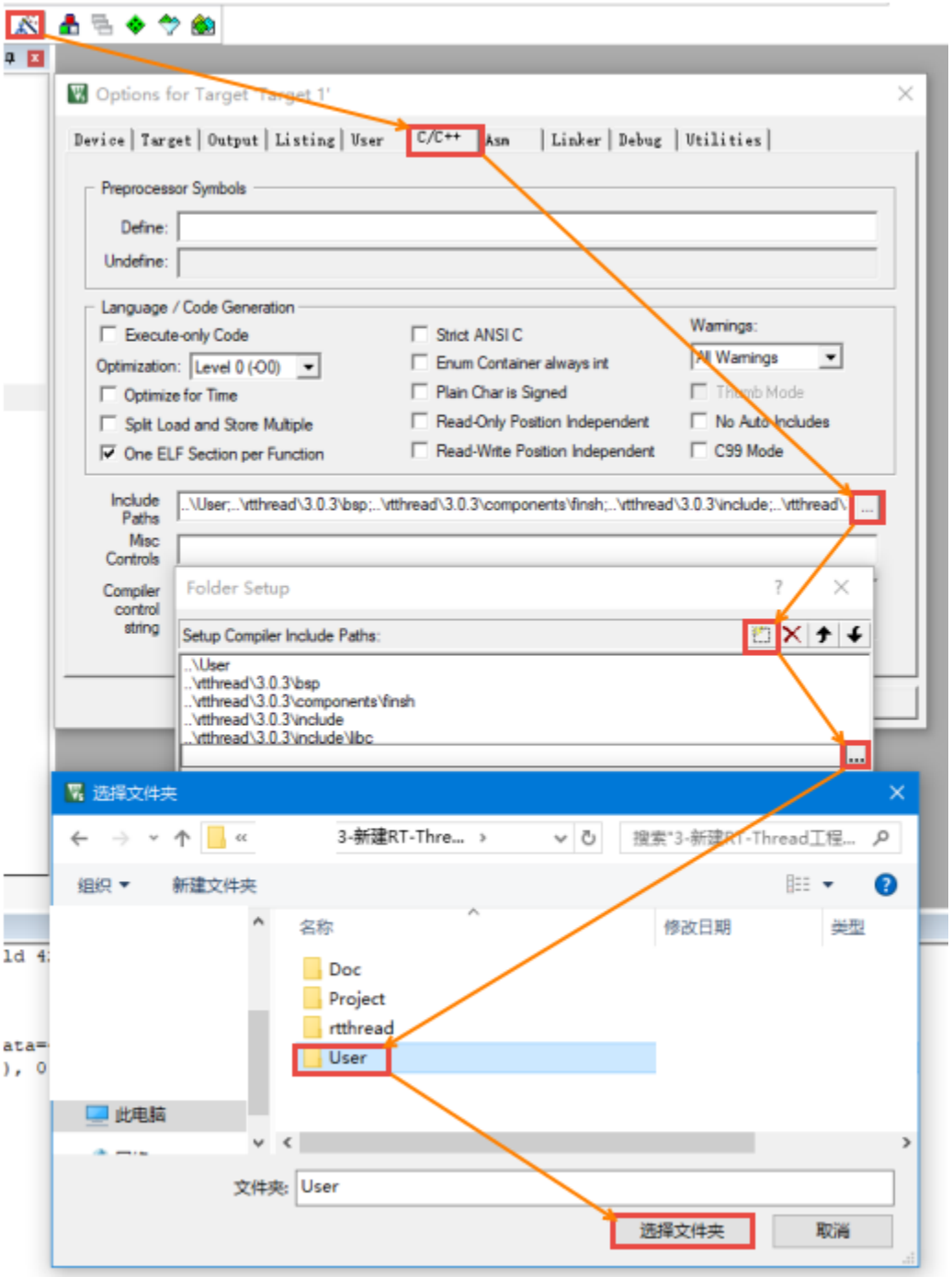
(续上页)

```
3
4 #define __SYSTEM_CLOCK (5*__XTAL) /* 5*5000000 = 25M */
```



6.4.3 添加头文件路径

在 C/C++ 选项卡里面指定工程头文件的路径，不然编译会出错，头文件路径的具体指定方法见图 *4-13*。



至此，一个完整的基于 Cortex-M7（Cortex-M4 或 Cortex-M3）内核的 RT-Thread 软件仿真的工程就建立完毕。

第 7 章　裸机系统与多线程系统

在真正开始动手写 RTOS 之前，我们先来讲解下单片机编程中的裸机系统和多线程系统的区别。

7.1 裸机系统

裸机系统通常分成轮询系统和前后台系统，有关这两者的具体实现方式请看下面的讲解。

7.1.1 轮询系统

轮询系统即是在裸机编程的时候，先初始化好相关的硬件，然后让主程序在一个死循环里面不断循环，顺序地做各种事情，大概的伪代码具体见*代码清单 5-1*。轮询系统是一种非常简单的软件结构，通常只适用于那些只需要顺序执行代码且不需要外部事件来驱动的就能完成的事情。在代码清单 5-1 中，如果只是实现 LED 翻转，串口输出，液晶显示等这些操作，那么使用轮询系统将会非常完美。但是，如果加入了按键操作等需要检测外部信号的事件，用来模拟紧急报警，那么整个系统的实时响应能力就不会那么好了。假设 DoSomething3 是按键扫描，当外部按键被按下，相当于一个警报，这个时候，需要立马响应，并做紧急处理，而这个时候程序刚好执行到 DoSomething1，要命的是 DoSomething1 需要执行的时间比较久，久到按键释放之后都没有执行完毕，那么当执行到 DoSomething3 的时候就会丢失掉一次事件。足见，轮询系统只适合顺序执行的功能代码，当有外部事件驱动时，实时性就会降低。

列表 1: 代码清单 5-1 轮询系统伪代码

```
int main(void)
{
    /* 硬件相关初始化 */
    HardWareInit();
    /* 无限循环 */
```

(下页继续)

(续上页)

```
    for (;;)
    {
        /* 处理事情 1 */
        DoSomething1();
        /* 处理事情 2 */
        DoSomething2();
        /* 处理事情 3 */
        DoSomething3();
    }
}
```

7.1.2 前后台系统

相比轮询系统，前后台系统是在轮询系统的基础上加入了中断。外部事件的响应在中断里面完成，事件的处理还是回到轮询系统中完成，中断在这里我们称为前台，main 函数里面的无限循环我们称为后台，大概的伪代码见代码清单 5-2。

列表 2: 代码清单 5-2 前后台系统伪代码

```
int flag1 = 0;
int flag2 = 0;
int flag3 = 0;

int main(void)
{
    /* 硬件相关初始化 */
    HardWareInit();
    /* 无限循环 */
    for (;;)
    {
        if (flag1)
```

(下页继续)

(续上页)

```
        {
            /* 处理事情 1 */
            DoSomething1();
        }

        if (flag2)
        {
            /* 处理事情 2 */
            DoSomething2();
        }

        if (flag3)
        {
            /* 处理事情 3 */
            DoSomething3();
        }
    }
}

void ISR1(void)
{
    /* 置位标志位 */
    flag1 = 1;
    /* 如果事件处理时间很短，则在中断里面处理如果事件处理时间比较长，在回到前台处理 */
    DoSomething1();
}

void ISR2(void)
{
    /* 置位标志位 */
    flag2 = 1;
    /* 如果事件处理时间很短，则在中断里面处理如果事件处理时间比较长，在回到前台处理 */
```

(下页继续)

(续上页)

```
    DoSomething2();
}

void ISR3(void)
{
    /* 置位标志位 */
    flag3 = 1;
    /* 如果事件处理时间很短，则在中断里面处理如果事件处理时间比较长，在回到前台处理 */
    DoSomething3();
}
```

在顺序执行后台程序的时候，如果有中断来临，那么中断会打断后台程序的正常执行流，转而去执行中断服务程序，在中断服务程序里面标记事件，如果事件要处理的事情很简短，则可在中断服务程序里面处理，如果事件要处理的事情比较多，则返回到后台程序里面处理。虽然事件的响应和处理是分开了，但是事件的处理还是在后台里面顺序执行的，但相比轮询系统，前后台系统确保了事件不会丢失，再加上中断具有可嵌套的功能，这可以大大的提高程序的实时响应能力。在大多数的中小型项目中，前后台系统运用的好，堪称有操作系统的效果。

7.2 多线程系统

相比前后台系统，多线程系统的事件响应也是在中断中完成的，但是事件的处理是在线程中完成的。在多线程系统中，线程跟中断一样，也具有优先级，优先级高的线程会被优先执行。当一个紧急的事件在中断被标记之后，如果事件对应的线程的优先级足够高，就会立马得到响应。相比前后台系统，多线程系统的实时性又被提高了。多线程系统大概的伪代码具体见*代码清单 5-3*。

列表 3: 代码清单 5-3 多线程系统伪代码

```
int flag1 = 0;
int flag2 = 0;
int flag3 = 0;
```

(下页继续)

(续上页)

```
4
5  int main(void)
6  {
7      /* 硬件相关初始化 */
8      HardWareInit();
9      /* OS 初始化 */
10     RTOSInit();
11     /* OS 启动，开始多线程调度，不再返回 */
12     RTOSStart();
13 }
14
15 void ISR1(void)
16 {
17     /* 置位标志位 */
18     flag1 = 1;
19 }
20
21 void ISR2(void)
22 {
23     /* 置位标志位 */
24     flag2 = 2;
25 }
26
27 void ISR3(void)
28 {
29     /* 置位标志位 */
30     flag3 = 1;
31 }
32
33 void DoSomething1(void)
34 {
35     /* 无限循环，不能返回 */
```

(下页继续)

(续上页)

```
    for (;;)
    {
        /* 线程实体 */
        if (flag1) {
        }
    }
}

void DoSomething2(void)
{
    /* 无限循环，不能返回 */
    for (;;)
    {
        /* 线程实体 */
        if (flag2) {
        }
    }
}

void DoSomething3(void)
{
    /* 无限循环，不能返回 */
    for (;;)
    {
        /* 线程实体 */
        if (flag3) {
        }
    }
}
```

相比前后台系统中后台顺序执行的程序主体，在多线程系统中，根据程序的功能，我们把这个程序主体分割成一个个独立的，无限循环且不能返回的小程序，这个小程序我们称之为线程。每个

线程都是独立的，互不干扰的，且具备自身的优先级，它由操作系统调度管理。加入操作系统后，我们在编程的时候不需要精心地去设计程序的执行流，不用担心每个功能模块之间是否存在干扰。加入了操作系统，我们的编程反而变得简单了。整个系统随之带来的额外开销就是操作系统占据的那一丁点的 FLASH 和 RAM。现如今，单片机的 FLASH 和 RAM 是越来越大，完全足以抵挡 RTOS 那点开销。

无论是裸机系统中的轮询系统、前后台系统和多线程系统，我们不能一锤子的敲定孰优孰劣，它们是不同时代的产物，在各自的领域都还有相当大的应用价值，只有合适才是最好。有关这三者的软件模型区别具体见表格 5-1。

表格 5-1 轮询、前后台和多线程系统软件模型区别

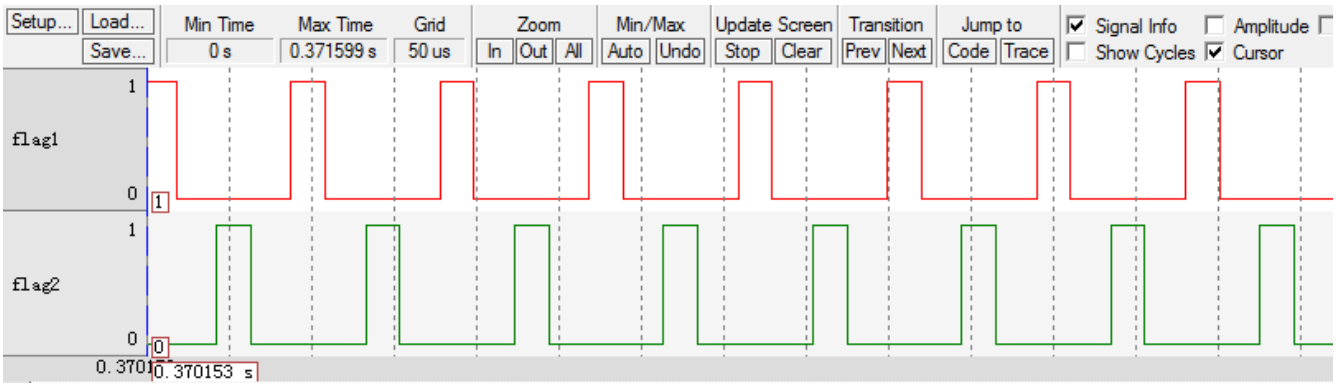
模型	事件响应	事件处理	特点
轮询系统	主程序	主程序	轮询响应事件，轮询处理事件
前后台系统	中断	主程序	实时响应事件，轮询处理事件
多线程系统	中断	线程	实时响应事件，实时处理事件

第 8 章　线程的定义与线程切换的实现

8.1 本章目标

本章是我们真正从从 0 到 1 写 RT-Thread 的第一章，属于基础中的基础，必须要学会创建线程，并重点掌握线程是如何切换的。因为线程的切换是由汇编代码来完成的，所以代码看起来比较难懂，但是我会尽力把代码讲得透彻。如果本章内容学不会，后面的内容根本无从下手。

在这章中，我们会创建两个线程，并让这两个线程不断地切换，线程的主体都是让一个变量按照一定的频率翻转，通过 KEIL 的软件仿真功能，在逻辑分析仪中观察变量的波形变化，最终的波形图具体见图 *6-1*。



其实，图 *6-1* 的波形图的效果，并不是真正的多线程系统中线程切换的效果图，这个效果其实可以完全由裸机代码来实现，具体见代码清单 *6-1*。

列表 1: 代码清单 6-1 裸机系统中两个变量轮流翻转

```
/* flag 必须定义成全局变量才能添加到逻辑分析仪里面观察波形在逻
 * 辑分析仪中要设置以 bit 的模式才能看到波形，不能用默认的模拟量
 */
uint32_t flag1;
uint32_t flag2;
```

(下页继续)

(续上页)

```

/* 软件延时 */
void delay (uint32_t count)
{
    for(; count!====0; count--);
}

int main(void)
{
    /* 无限循环，顺序执行 */
    for (;;)
    {
        flag1 ==== 1;
        delay( 100 );
        flag1 ==== 0;
        delay( 100 );
        flag2 ==== 1;
        delay( 100 );
        flag2 ==== 0;
        delay( 100 );
    }
}
```

在多线程系统中，两个线程不断切换的效果图应该像图 *6-2* 所示那样，即两个变量的波形是完全一样的，就好像 CPU 在同时干两件事一样，这才是多线程的意义。虽然两者的波形图一样，但是，代码的实现方式是完全不一样的，由原来的顺序执行变成了线程的主动切换，这是根本区别。这章只是开始，我们先掌握好线程是如何切换，在后面章节中，我们会陆续的完善功能代码，加入系统调度，实现真正的多线程。千里之行，始于本章节，不要急。

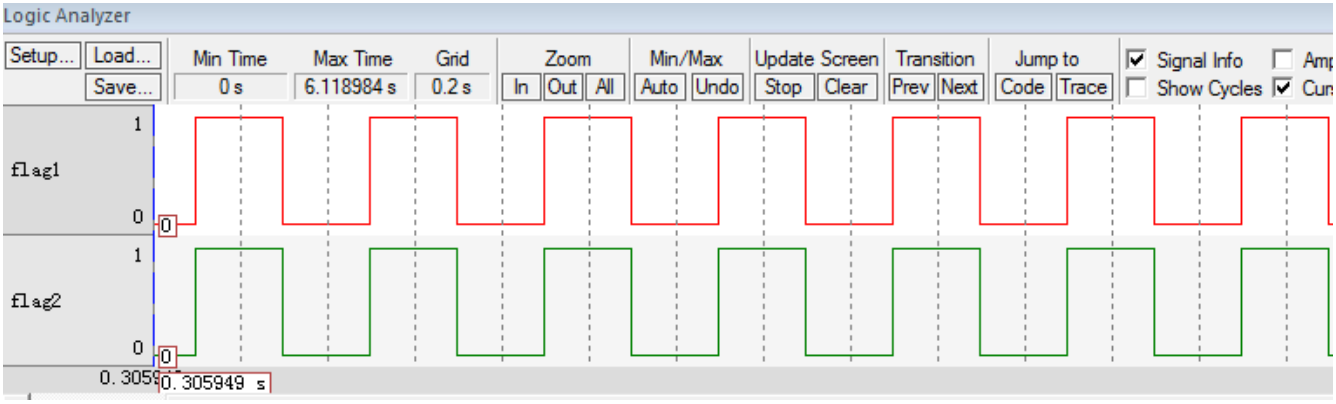


图 *6-2* 多线程系统线程切换波形图

8.2 什么是线程

在裸机系统中，系统的主体就是 main 函数里面顺序执行的无限循环，这个无限循环里面 CPU 按照顺序完成各种事情。在多线程系统中，我们根据功能的不同，把整个系统分割成一个个独立的且无法返回的函数，这个函数我们称为线程。线程的大概形式具体见*代码清单 6-2*。

列表 2: 代码清单 6-2 多线程系统中线程的形式

```
void thread_entry (void *parg)
{
    /* 线程主体，无限循环且不能返回 */
    for (;;)
    {
        /* 线程主体代码 */
    }
}
```

8.3 创建线程

8.3.1 定义线程栈

我们先回想下，在一个裸机系统中，如果有全局变量，有子函数调用，有中断发生。那么系统在运行的时候，全局变量放在哪里，子函数调用时，局部变量放在哪里，中断发生时，函数返回地址发哪里。如果只是单纯的裸机编程，它们放哪里我们不用管，但是如果要写一个 RTOS，这些种种环境参数，我们必须弄清楚他们是如何存储的。在裸机系统中，他们统统放在一个叫栈的地方，栈是单片机 RAM 里面一段连续的内存空间，栈的大小一般在启动文件或者链接脚本里面指定，最后由 C 库函数 _main 进行初始化。

但是，在多线程系统中，每个线程都是独立的，互不干扰的，所以要为每个线程都分配独立的栈空间，这个栈空间通常是一个预先定义好的全局数组，也可以是动态分配的一段内存空间，但它们都存在于 RAM 中。

本章我们要实现两个变量按照一定的频率轮流的翻转，每个变量对应一个线程，那么就需要定义两个线程栈，具体见*代码清单 6-3*。在多线程系统中，有多少个线程就需要定义多少个线程栈。

列表 3: 代码清单 6-3 定义线程栈

```
ALIGN(RT_ALIGN_SIZE)//     (2)
/* 定义线程栈 */
rt_uint8_t rt_flag1_thread_stack[512];//    (1)
rt_uint8_t rt_flag2_thread_stack[512];
```

代码清单 6-3 **(1)**：线程栈其实就是一个预先定义好的全局数据，数据类型为 rt_uint8_t，大小我们设置为 512。在 RT-Thread 中，凡是涉及到数据类型的地方，RT-Thread 都会将标准的 C 数据类型用 typedef 重新取一个类型名，以“rt”前缀开头。这些经过重定义的数据类型放在 rtdef.h（rtdef.h 第一次使用需要在 include 文件夹下面新建然后添加到工程 rtt/source 这个组文件）这个头文件，具体见代码清单 6-4。代码清单 6-4 中除了 rt_uint8_t 外，其它数据类型重定义是本章后面内容需要使用到，这里统一贴出来，后面将不再赘述。

列表 4: 代码清单 6-4 rtdef.h 中的数据类型

```
#ifndef __RT_DEF_H__
#define __RT_DEF_H__

/*
*          数据类型
*/

/* RT-Thread 基础数据类型重定义 */
typedef signed   char                   rt_int8_t;
typedef signed   short                  rt_int16_t;
typedef signed   long                   rt_int32_t;
typedef unsigned char                   rt_uint8_t;
typedef unsigned short                  rt_uint16_t;
typedef unsigned long                   rt_uint32_t;
typedef int                             rt_bool_t;

```

(下页继续)

(续上页)

```
/* 32bit CPU*/
typedef long                          rt_base_t;
typedef unsigned long                 rt_ubase_t;
typedef rt_base_t                     rt_err_t;
typedef rt_uint32_t                   rt_time_t;
typedef rt_uint32_t                   rt_tick_t;
typedef rt_base_t                     rt_flag_t;
typedef rt_ubase_t                    rt_size_t;
typedef rt_ubase_t                    rt_dev_t;
typedef rt_base_t                     rt_off_t;


/* 布尔数据类型重定义 */
#define RT_TRUE                       1
#define RT_FALSE                      0

#ifdef __CC_ARM
      #define rt_inline                 static __inline
      #define ALIGN(n)                  __attribute__((aligned(n)))

#elif defined (__IAR_SYSTEMS_ICC__)
   #define rt_inline                 static inline
      #define ALIGN(n)                  PRAGMA(data_alignment====n)

#elif defined (__GNUC__)
   #define rt_inline                 static __inline
      #define ALIGN(n)                  __attribute__((aligned(n)))
#else
   #error not supported tool chain
#endif

#define RT_ALIGN(size, align)           (((size) + (align) - 1) & ~((align)␣
↪- 1))
```

(下页继续)

(续上页)

```
#define RT_ALIGN_DOWN(size, align)      ((size) & ~((align) - 1))

#define RT_NULL                         (0)

#endif /* __RT_DEF_H__ */
```

代码清单 6-3 **(2)**：设置变量需要多少个字节对齐，对在它下面的变量起作用。ALIGN 是一个带参宏，在 rtdef.h 中定义，具体见代码清单 6-4。RT_ALIGN_SIZE 是一个在 rtconfig.h （rtconfig.h 第一次使用需要在 User 文件夹下面新建然后添加到工程 user 这个组文件）中定义的宏，默认为 4，表示 4 个字节对齐，具体见代码清单 6-5。

列表 5: 代码清单 6-5 RT_ALIGN_SIZE 宏定义

```
#ifndef __RTTHREAD_CFG_H__
#define __RTTHREAD_CFG_H__

#define RT_THREAD_PRIORITY_MAX  32      /* 最大优先级 */
#define RT_ALIGN_SIZE           4       /* 多少个字节对齐 */

#endif /* __RTTHREAD_CFG_H__ */
```

8.3.2 定义线程函数

线程是一个独立的函数，函数主体无限循环且不能返回。本章我们在 main.c 中定义的两个线程具体见代码清单 6-6。

列表 6: 代码清单 6-6 线程函数

```
/* 软件延时 */
void delay (uint32_t count)
{
```

(下页继续)

(续上页)

```
    for(; count!====0; count--);
}

/* 线程 1 */
void flag1_thread_entry( void *p_arg )//    (1)
{
    for( ;; )
    {
        flag1 ==== 1;
        delay( 100 );
        flag1 ==== 0;
        delay( 100 );

        /* 线程切换，这里是手动切换 */
        rt_schedule();
    }
}

/* 线程 2 */
void flag2_thread_entry( void *p_arg )//    (2)
{
    for( ;; )
    {
        flag2 ==== 1;
        delay( 100 );
        flag2 ==== 0;
        delay( 100 );

        /* 线程切换，这里是手动切换 */
        rt_schedule();
    }
}
```

代码清单 6-6 **(1)**、**(2)**：正如我们所说的那样，线程是一个独立的、无限循环且不能返回的函数。

8.3.3 定义线程控制块

在裸机系统中，程序的主体是 CPU 按照顺序执行的。而在多线程系统中，线程的执行是由系统调度的。系统为了顺利的调度线程，为每个线程都额外定义了一个线程控制块，这个线程控制块就相当于线程的身份证，里面存有线程的所有信息，比如线程的栈指针，线程名称，线程的形参等。有了这个线程控制块之后，以后系统对线程的全部操作都可以通过这个线程控制块来实现。定义一个线程控制块需要一个新的数据类型，该数据类型在 rtdef.h 这个头文件中声明，具体的声明见代码清单 6-7，使用它可以为每个线程都定义一个线程控制块实体。

列表 7: 代码清单 6-7 线程控制块类型声明

```
struct rt_thread//     (1)
{
    void        *sp;                /* 线程栈指针 */
    void        *entry;                 /* 线程入口地址 */
    void        *parameter;           /* 线程形参 */
    void        *stack_addr;        /* 线程起始地址 */
    rt_uint32_t stack_size;         /* 线程栈大小，单位为字节 */

    rt_list_t   tlist;              /* 线程链表节点 */
};
typedef struct rt_thread *rt_thread_t;//     (2)
```

代码清单 6-7 **(1)**：目前线程控制块结构体里面的成员还比较少，往后我们会慢慢在里面添加成员。

代码清单 6-7 **(2)**：在 RT-Thread 中，都会给新声明的数据结构重新定义一个指针。往后如果要定义线程控制块变量就使用 struct rt_thread xxx 的形式，定义线程控制块指针就使用 rt_thread_t xxx 的形式。

在本章实验中，我们在 main.c 文件中为两个线程定义的线程控制块，具体见代码清单 6-8。

列表 8: 代码清单 6-8 线程控制块定义

```
/* 定义线程控制块 */
struct rt_thread rt_flag1_thread;
struct rt_thread rt_flag2_thread;
```

8.3.4 实现线程创建函数

线程的栈，线程的函数实体，线程的控制块最终需要联系起来才能由系统进行统一调度。那么这个联系的工作就由线程初始化函数 rt_thread_init() 来实现，该函数在 thread.c（thread.c 第一次使用需要自行在文件夹 rtthread/3.0.3/src 中新建并添加到工程的 rtt/source 组）中定义，在 rtthread.h 中声明，所有跟线程相关的函数都在这个文件定义。rt_thread_init() 函数的实现见*代码清单 6-9*。

列表 9: 代码清单 6-9 rt_thread_init() 函数

```
rt_err_t rt_thread_init(struct rt_thread *thread,//          (1)
                        void (*entry)(void *parameter),//    (2)
                        void             *parameter,//        (3)
                        void             *stack_start,//      (4)
                        rt_uint32_t       stack_size)//       (5)
{
    rt_list_init(&(thread->tlist));//                         (6)

    thread->entry ==== (void *)entry;//                          (7)
    thread->parameter ==== parameter;//                          (8)

    thread->stack_addr ==== stack_start;//                       (9)
    thread->stack_size ==== stack_size;//                        (10)

    /* 初始化线程栈，并返回线程栈指针 */ //                    (11)
    thread->sp ==== (void *)rt_hw_stack_init( thread->entry,
```

(下页继续)

(续上页)

```
                                        thread->parameter,
                                    (void *)((char *)thread->stack_addr +␣
↪thread->stack_size - 4) );

    return RT_EOK;//                                                (12)
}
```

代码清单 6-9：rt_thread_init 函数遵循 RT-Thread 中的函数命名规则，以小写的 rt 开头，表示这是一个外部函数，可以由用户调用，以 _rt 开头的函数表示内部函数，只能由 RT-Thread 内部使用。紧接着是文件名，表示该函数放在哪个文件，最后是函数功能名称。

代码清单 6-9 **(1)**：thread 是线程控制块指针。

代码清单 6-9 **(2)**：entry 是线程函数名，表示线程的入口。

代码清单 6-9 **(3)**：parameter 是线程形参，用于传递线程参数。

代码清单 6-9 **(4)**：stack_start 用于指向线程栈的起始地址。

代码清单 6-9 **(5)**：stack_size 表示线程栈的大小，单位为字节。

8.3.4.1 实现链表相关函数

代码清单 6-9 **(6)**：初始化线程链表节点，往后我们要把线程插入到各种链表中，就是通过这个节点来实现的，它就好像是线程控制块里面的一个钩子，可以把线程控制块挂在各种链表中。在初始化之前我们需要在线程控制块中添加一个线程链表节点，具体见代码清单 6-10 的高亮部分代码。

列表 10: 代码清单 6-10 在线程控制块中添加线程链表节点

```
struct rt_thread
{
    void        *sp;            /* 线程栈指针 */
```

(下页继续)

(续上页)

```
    void        *entry;                 /* 线程入口地址 */
    void        *parameter;           /* 线程形参 */
    void        *stack_addr;        /* 线程起始地址 */
    rt_uint32_t stack_size;         /* 线程栈大小，单位为字节 */

    rt_list_t   tlist;              /* 线程链表节点 *///    (1)
};
typedef struct rt_thread *rt_thread_t;
```

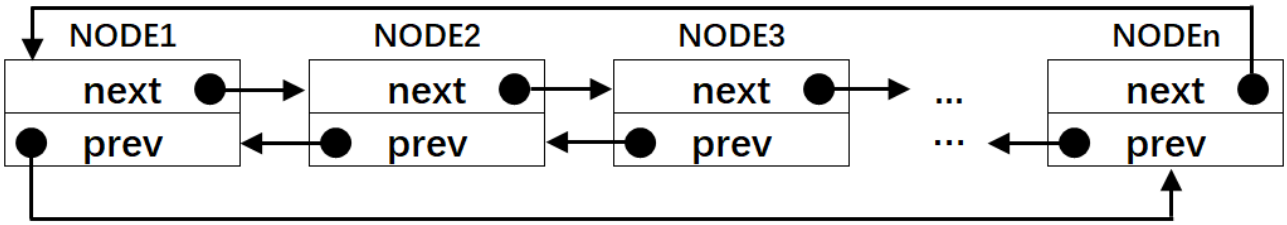
代码清单 6-10(**1**)：线程链表节点 tlist 的数据类型是是 rt_list_t，该数据类型在 rtdef.h 中定义，具体见代码清单 6-11。

8.3.4.1.1 定义链表节点数据类型

列表 11: 代码清单 6-11 双向链表节点数据类型 rt_list_t 定义

```
struct rt_list_node
{
    struct rt_list_node *next;                /* 指向后一个节点 */
    struct rt_list_node *prev;                /* 指向前一个节点 */
};
typedef struct rt_list_node rt_list_t;
```

rt_list_t 类型的节点里面有两个 rt_list_t 类型的节点指针 next 和 prev，分别用来指向链表中的下一个节点和上一个节点。由 rt_list_t 类型的节点构成的双向链表示意图具体见图 *6-3*。



现在我们详细讲解下双向链表的相关操作，这些函数均在 rtservice.h 中实现，rtservice.h 第一次使用需要自行在 rtthread/3.0.3/include 文件夹下新建，然后添加到工程的 rtt/source 组中。

8.3.4.1.2 初始化链表节点

rt_list_t 类型的节点的初始化，就是将节点里面的 next 和 prev 这两个节点指针指向节点本身，具体的代码实现见*代码清单 6-12*，具体的示意图见图 *6-4*。

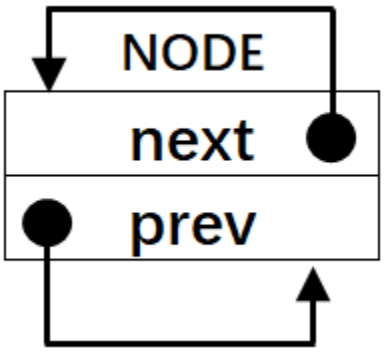
列表 12: 代码清单 6-12 初始化 rt_list_t 类型的链表节点

```
rt_inline void rt_list_init(rt_list_t *l)
{
    l->next ==== l->prev ==== l;
```

(下页继续)

(续上页)

```
}
```

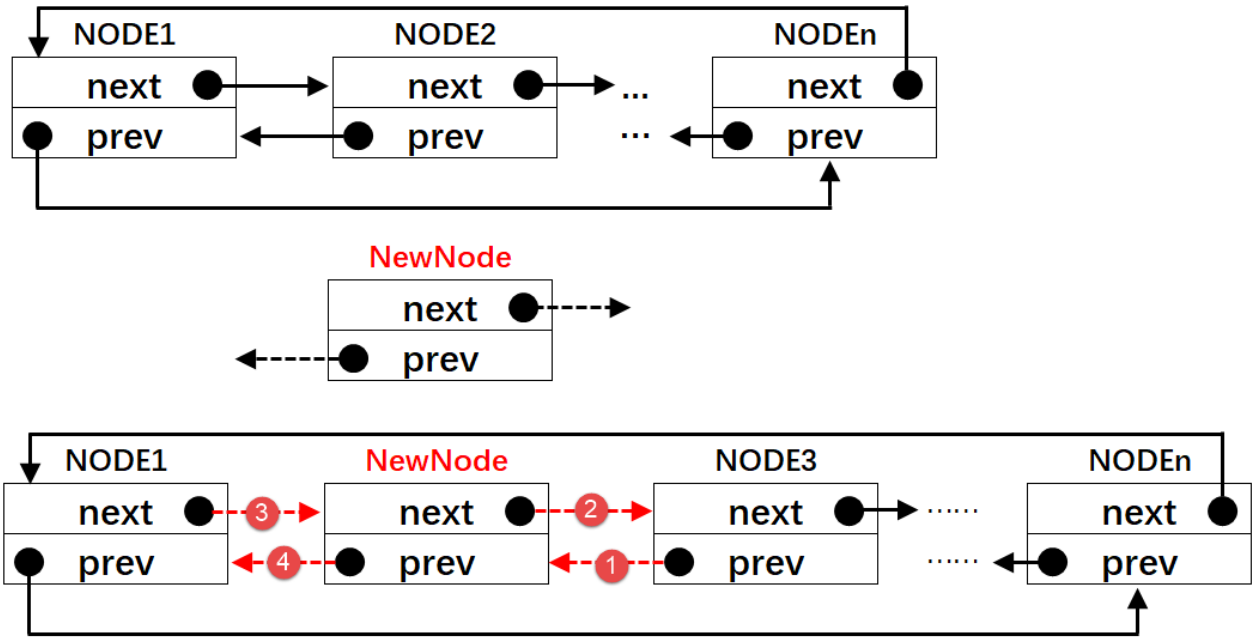


8.3.4.1.3 在双向链表表头后面插入一个节点

在双向链表头部插入一个节点，具体代码实现见*代码清单 6-13*，主要处理分为 4 步，插入前和插入后的示意图具体见图 *6-5*。

列表 13: 代码清单 6-13 在双向链表表头后面插入一个节点

```
/* 在双向链表头部插入一个节点 */
rt_inline void rt_list_insert_after(rt_list_t *l, rt_list_t *n)
{
    l->next->prev ==== n; /* 第 1 步 */
    n->next ==== l->next; /* 第 2 步 */
    l->next ==== n; /* 第 3 步 */
    n->prev ==== l; /* 第 4 步 */
}
```

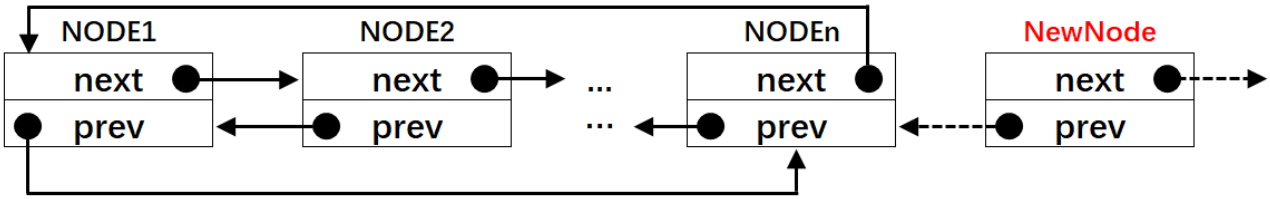


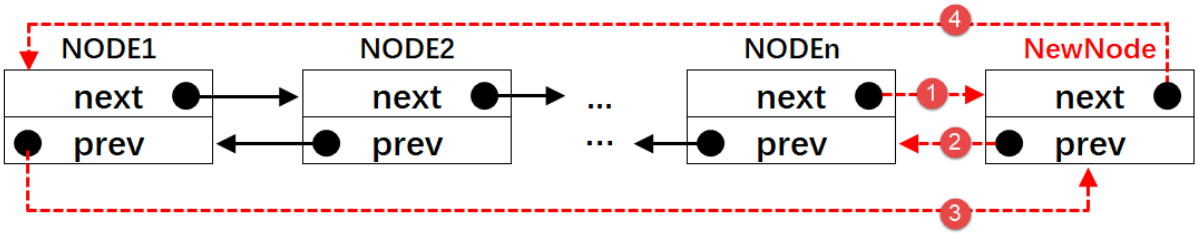
8.3.4.1.4 在双向链表表头前面插入一个节点

在双向链表尾部插入一个节点，具体代码实现见*代码清单 6-14*，主要处理分为 4 步，插入前和插入后的示意图具体见图 *6-6*。

列表 14: 代码清单 6-14 在双向链表表头前面插入一个节点

```
rt_inline void rt_list_insert_before(rt_list_t *l, rt_list_t *n)
{
    l->prev->next ==== n; /* 第 1 步 */
    n->prev ==== l->prev; /* 第 2 步 */
    l->prev ==== n; /* 第 3 步 */
    n->next ==== l; /* 第 4 步 */
}
```



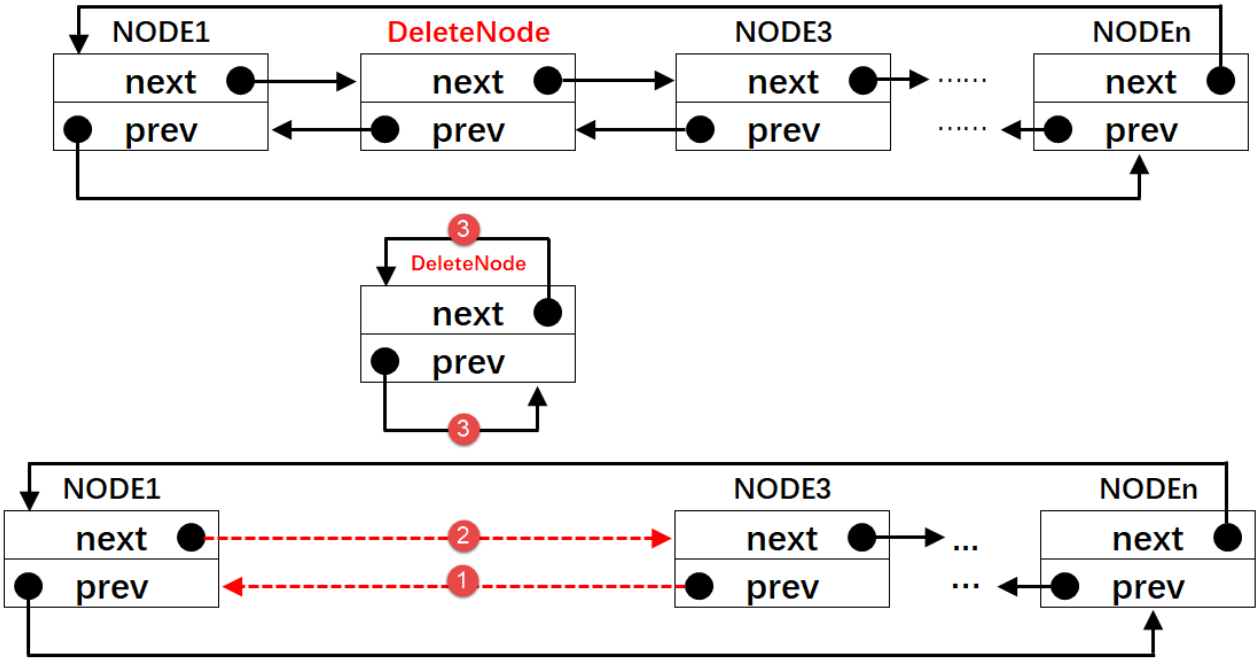


8.3.4.1.5 从双向链表删除一个节点

从双向链表删除一个节点，具体代码实现见代码清单 6-15，主要处理分为 3 步，删除前和删除后的示意图具体见图 6-7。

列表 15: 代码清单 6-15 从双向链表删除一个节点

```
rt_inline void rt_list_remove(rt_list_t *n)
{
    n->next->prev ==== n->prev; /* 第 1 步 */
    n->prev->next ==== n->next; /* 第 2 步 */
    n->next ==== n->prev ==== n; /* 第 3 步 */
}
```



代码清单 6-9 **(7)**：将线程入口保存到线程控制块的 entry 成员中。

代码清单 6-9 **(8)**：将线程入口形参保存到线程控制块的 parameter 成员中。

代码清单 6-9 **(9)**：将线程栈起始地址保存到线程控制块的 stack_start 成员中。

代码清单 6-9 **(10)**：将线程栈起大小保存到线程控制块的 stack_size 成员中。

代码清单 6-9 **(11)**：初始化线程栈，并返回线程栈顶指针。rt_hw_stack_init() 用来初始化线程栈，当线程第一次运行的时候，加载到 CPU 寄存器的参数就放在线程栈里面，该函数在 cpuport.c 中实现，具体见代码清单 6-16。cpuport.c 第一次使用需要自行在 rtthread/3.0.3/ libcpu/arm/cortex-m3 (cortex-m4 或 cortex-m7) 文件夹下新建，然后添加到工程的 rtt/ports 组中。

8.3.4.2 rt_hw_stack_init() 函数

列表 16: 代码清单 6-16 rt_hw_stack_init() 函数

```
/* 线程栈初始化 */
rt_uint8_t *rt_hw_stack_init(void        *tentry,//                    (1)
                             void        *parameter,//                  (2)
                             rt_uint8_t  *stack_addr)//                 (3)
{

    struct stack_frame *stack_frame;//                                  (4)
    rt_uint8_t         *stk;
    unsigned long       i;

    /* 获取栈顶指针
    rt_hw_stack_init 在调用的时候，传给 stack_addr 的是 (栈顶指针)*/
    stk  ==== stack_addr + sizeof(rt_uint32_t);//                       (5)

    /* 让 stk 指针向下 8 字节对齐 */
    stk  ==== (rt_uint8_t *)RT_ALIGN_DOWN((rt_uint32_t)stk, 8);//       (6)

    /* stk 指针继续向下移动 sizeof(struct stack_frame) 个偏移 */
    stk -==== sizeof(struct stack_frame);//                             (7)

    /* 将 stk 指针强制转化为 stack_frame 类型后存到 stack_frame */
    stack_frame ==== (struct stack_frame *)stk;//                       (8)
```

(下页继续)

(续上页)

```
23
24      /* 以 stack_frame 为起始地址，将栈空间里面的 sizeof(struct stack_frame)
25      个内存初始化为 0xdeadbeef */
26      for (i ==== 0; i < sizeof(struct stack_frame) / sizeof(rt_uint32_t); i
↪++)//    (9)
27      {
28              ((rt_uint32_t *)stack_frame)[i] ==== 0xdeadbeef;
29      }
30
31      /* 初始化异常发生时自动保存的寄存器 *////                                (10)
32      stack_frame->exception_stack_frame.r0  ==== (unsigned long)parameter; /*
↪r0 : argument */
33      stack_frame->exception_stack_frame.r1  ==== 0;                        /*
↪r1 */
34      stack_frame->exception_stack_frame.r2  ==== 0;                        /*
↪r2 */
35      stack_frame->exception_stack_frame.r3  ==== 0;                        /*
↪r3 */
36      stack_frame->exception_stack_frame.r12 ==== 0;                        /*
↪r12 */
37      stack_frame->exception_stack_frame.lr  ==== 0;                        /*
↪lr */
38      stack_frame->exception_stack_frame.pc  ==== (unsigned long)tentry;    /*
↪entry point, pc */
39      stack_frame->exception_stack_frame.psr ==== 0x01000000L;              /*
↪PSR */
40
41      /* 返回线程栈指针 */
42      return stk;//                                                        (11)
43  }
```

代码清单 6-16(1)：线程入口。

代码清单 6-16(2)：线程形参。

代码清单 6-16(3)：线程栈顶地址-4，在该函数调用的时候传进来的是线程栈的栈顶地址-4。

代码清单 6-16(4)：定义一个 struct stack_frame 类型的结构体指针 stack_frame，该结构体类型在 cpuport.c 中定义，具体见代码清单 6-17。

列表 17: 代码清单 6-17 struct stack_frame 类型结构体定义

```
struct exception_stack_frame
{
    /* 异常发生时自动保存的寄存器 */
    rt_uint32_t r0;
    rt_uint32_t r1;
    rt_uint32_t r2;
    rt_uint32_t r3;
    rt_uint32_t r12;
    rt_uint32_t lr;
    rt_uint32_t pc;
    rt_uint32_t psr;
};

struct stack_frame
{
    /* r4 ~ r11 register
    异常发生时需手动保存的寄存器 */
    rt_uint32_t r4;
    rt_uint32_t r5;
    rt_uint32_t r6;
    rt_uint32_t r7;
    rt_uint32_t r8;
    rt_uint32_t r9;
    rt_uint32_t r10;
```

(下页继续)

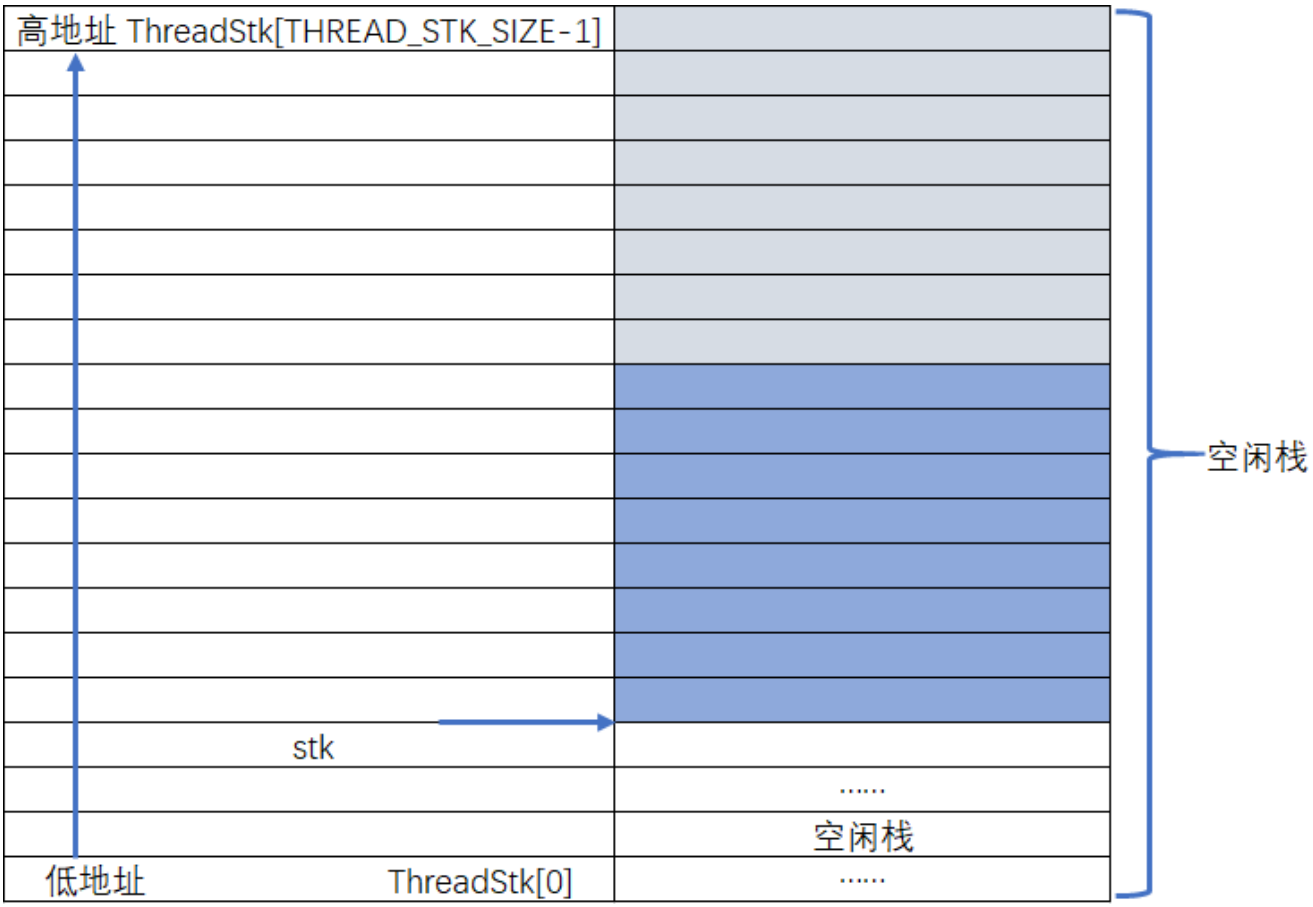
(续上页)

```
    rt_uint32_t r11;

    struct exception_stack_frame exception_stack_frame;
};
```

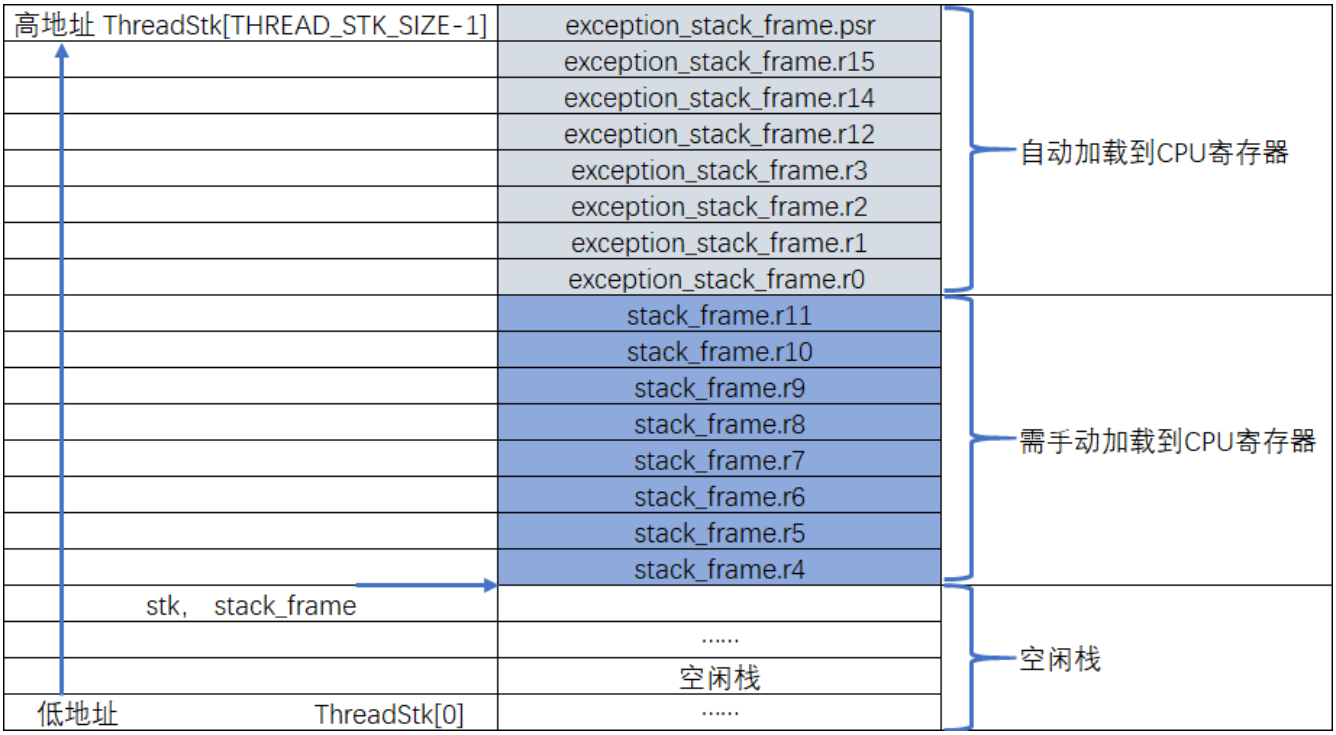
代码清单 6-16**(5)**：获取栈顶指针，将栈顶指针传给指针 stk。rt_hw_stack_init() 函数在 rt_thread_init () 函数中调用的时候传给形参 stack_addr 的值是栈顶指针减去 4，所以现在加上 sizeof(rt_uint32_t) 刚好与减掉的 4 相互抵消，即传递给 stk 的是栈顶指针。

代码清单 6-16**(6)**：让 stk 这个指针向下 8 个字节对齐，确保 stk 是 8 字节对齐的地址。在 Cortex-M3（Cortex-M4 或 Cortex-M7）内核的单片机中，因为总线宽度是 32 位的，通常只要栈保持 4 字节对齐就行，可这样为啥要 8 字节？难道有哪些操作是 64 位的？确实有，那就是浮点运算，所以要 8 字节对齐（但是目前我们都还没有涉及到浮点运算，只是为了后续兼容浮点运行的考虑）。如果栈顶指针是 8 字节对齐的，在进行向下 8 字节对齐的时候，指针不会移动，如果不是 8 字节对齐的，在做向下 8 字节对齐的时候，就会空出几个字节，不会使用，比如当 stk 是 33，明显不能整除 8，进行向下 8 字节对齐就是 32，那么就会空出一个字节不使用。

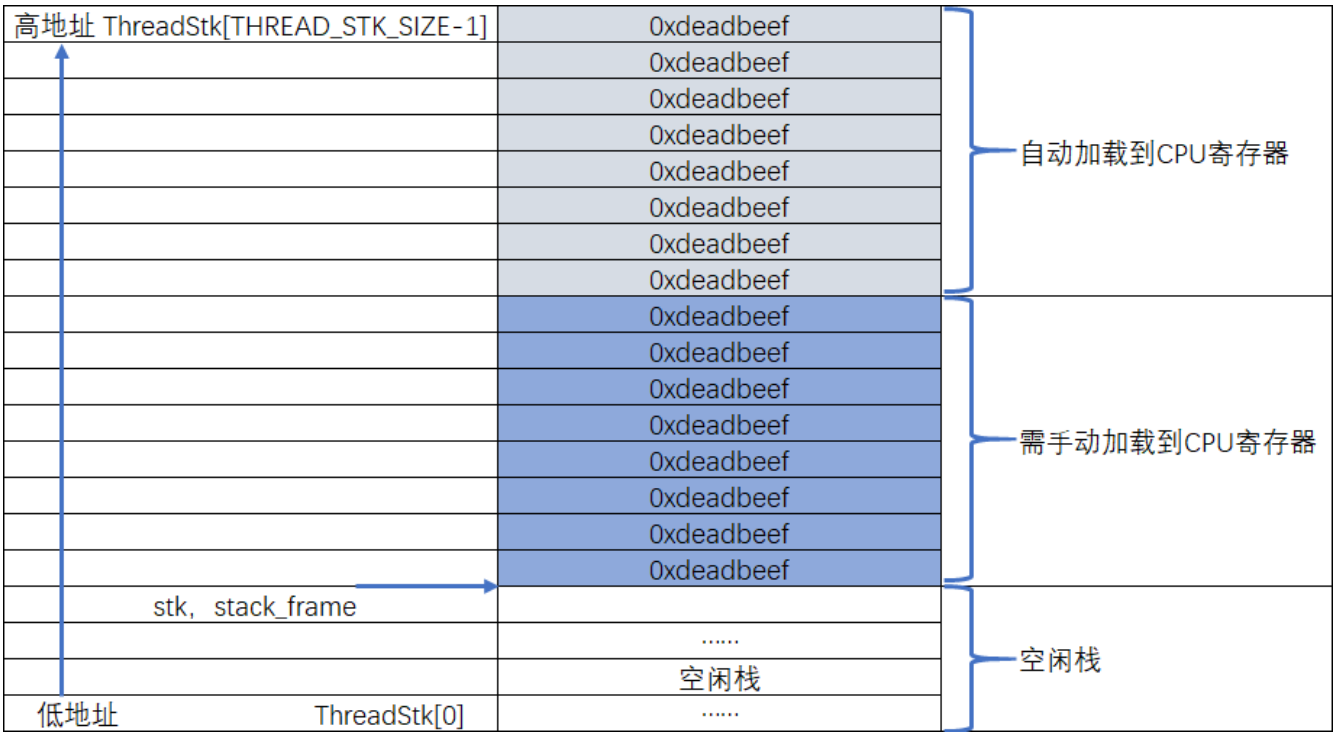
代码清单 6-16**(7)**：stk 指针继续向下移动 sizeof(struct stack_frame) 个偏移，即 16 个字的大小。如果栈顶指针一开始都是 8 字节对齐的，那么 stk 现在在线程栈里面的指向具体见图 6-8。



代码清单 6-16(**8**)：将 stk 指针强制转化为 stack_frame 类型后存到指针变量 stack_frame 中，这个时候 stack_frame 在线程栈里面的指向具体见图 *6-9*。



代码清单 6-16(**9**)：以 stack_frame 为起始地址，将栈空间里面的 sizeof(struct stack_frame) 个内存初始化为 0xdeadbeef ，这个时候栈空间的内容分布具体见图 *6-10*。



代码清单 6-16(**10**)：线程第一次运行的时候，加载到 CPU 寄存器的环境参数我们要预先初始化好。从栈顶开始，初始化的顺序固定，首先是异常发生时自动保存的 8 个寄存器，即 xPSR、R15、R14、R12、R3、R2、R1 和 R0。其中 xPSR 寄存器的位 24 必须是 1，R15PC 指针必须存的是线程的入口地址，R0 必须是线程形参，剩下的 R14、R12、R3、R2 和 R1 我们初始化为 0。

剩下的是 8 个需要手动加载到 CPU 寄存器的参数，即 R4~R11，默认初始化为 0xdeadbeaf，具体见图 *6-11*。

代码清单 6-16(**11**)：返回线程栈指针 stk，这个时候 stk 指向剩余栈的栈顶。

代码清单 6-9 (**12**)：线程初始化成功，返回错误码 RT_EOK。RT-Thread 的错误码在 rtdef.h 中定义，具体见*代码清单 6-18*。

列表 18: 代码清单 6-18 错误码宏定义

```
/* RT-Thread 错误码重定义 */
#define RT_EOK                          0               /**< There is no error */
#define RT_ERROR                        1               /**< A generic error happens */
#define RT_ETIMEOUT                     2               /**< Timed out */
#define RT_EFULL                        3               /**< The resource is full */
```

(下页继续)

(续上页)

```
#define RT_EEMPTY                    4                /**< The resource is empty */
#define RT_ENOMEM                    5                /**< No memory */
#define RT_ENOSYS                    6                /**< No system */
#define RT_EBUSY                     7                /**< Busy */
#define RT_EIO                       8                /**< IO error */
#define RT_EINTR                     9                /**< Interrupted system call */
#define RT_EINVAL                    10               /**< Invalid argument */
```

在本章实验中，我们在 main 函数中创建两个 flag 相关的线程，具体见代码清单 6-19。

列表 19: 代码清单 6-19 初始化线程

```
/* 初始化线程 */
rt_thread_init(&rt_flag1_thread,                    /* 线程控制块 */
               flag1_thread_entry,                  /* 线程入口地址 */
               RT_NULL,                             /* 线程形参 */
               &rt_flag1_thread_stack[0],           /* 线程栈起始地址 */
               sizeof(rt_flag1_thread_stack) );     /* 线程栈大小，单位为字节 */
/* 将线程插入到就绪列表 */

/* 初始化线程 */
rt_thread_init(&rt_flag2_thread,                    /* 线程控制块 */
               flag2_thread_entry,                  /* 线程入口地址 */
               RT_NULL,                             /* 线程形参 */
               &rt_flag2_thread_stack[0],           /* 线程栈起始地址 */
               sizeof(rt_flag2_thread_stack) );     /* 线程栈大小，单位为字节 */
```

8.4 实现就绪列表

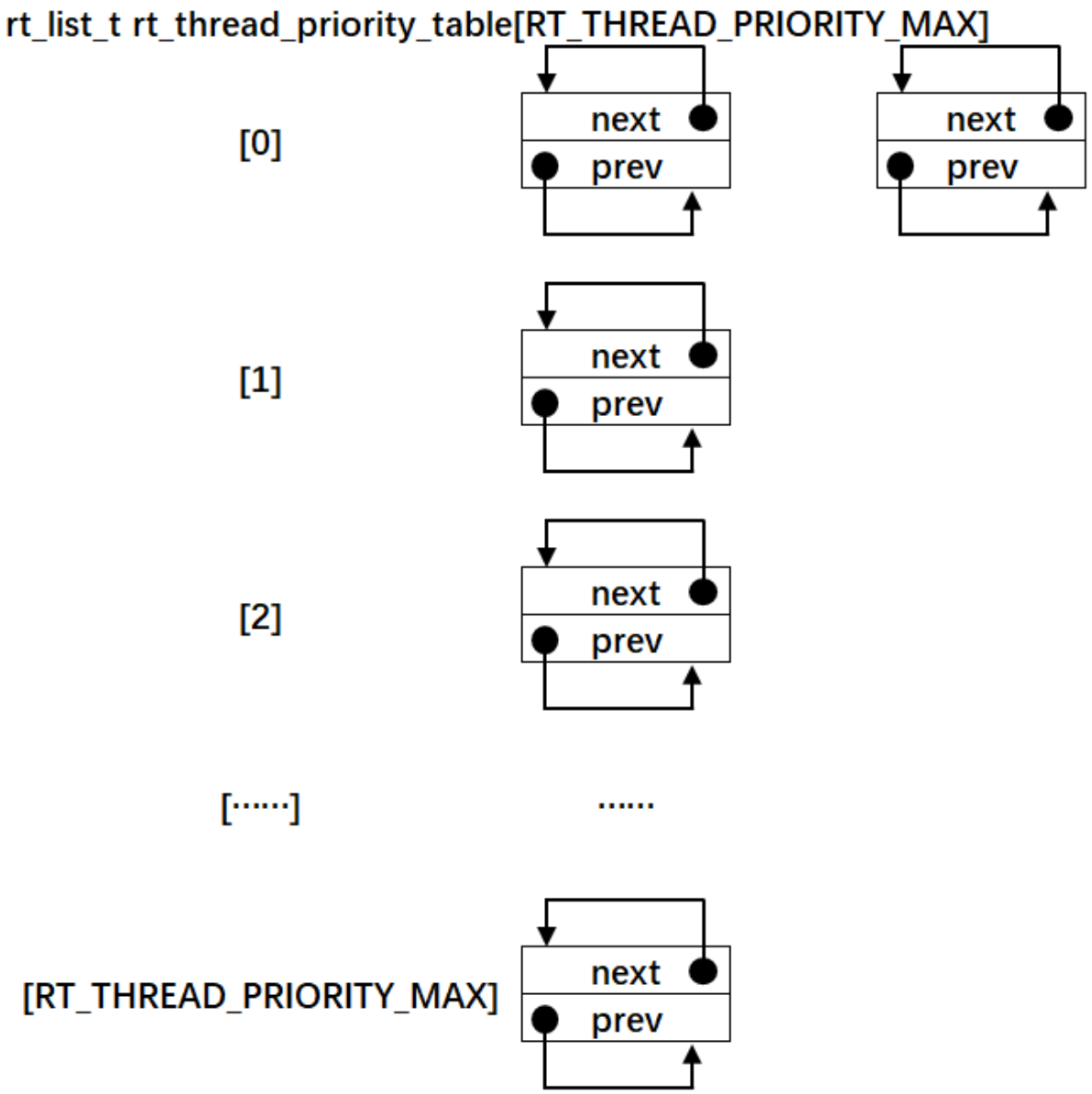
8.4.1 定义就绪列表

线程创建好之后，我们需要把线程添加到就绪列表里面，表示线程已经就绪，系统随时可以调度。就绪列表在 scheduler.c 中定义（scheduler.c 第一次使用需要在 rtthread3.0.3src 目录下新建，然后添加到工程的 rtt/source 组中），具体见*代码清单 6-20*。

列表 20: 代码清单 6-20 定义就绪列表

```
/* 线程就绪列表 */
rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX]; //  (1)
```

代码清单 6-20(**1**)：就绪列表实际上就是一个 rt_list_t 类型的数组，数组的大小由决定最大线程优先级的宏 RT_THREAD_PRIORITY_MAX 决定，RT_THREAD_PRIORITY_MAX 在 rtconfig.h 中默认定义为 32。数组的下标对应了线程的优先级，同一优先级的线程统一插入到就绪列表的同一条链表中。一个空的就绪列表具体见图 *6-12*。



8.4.2 将线程插入到就绪列表

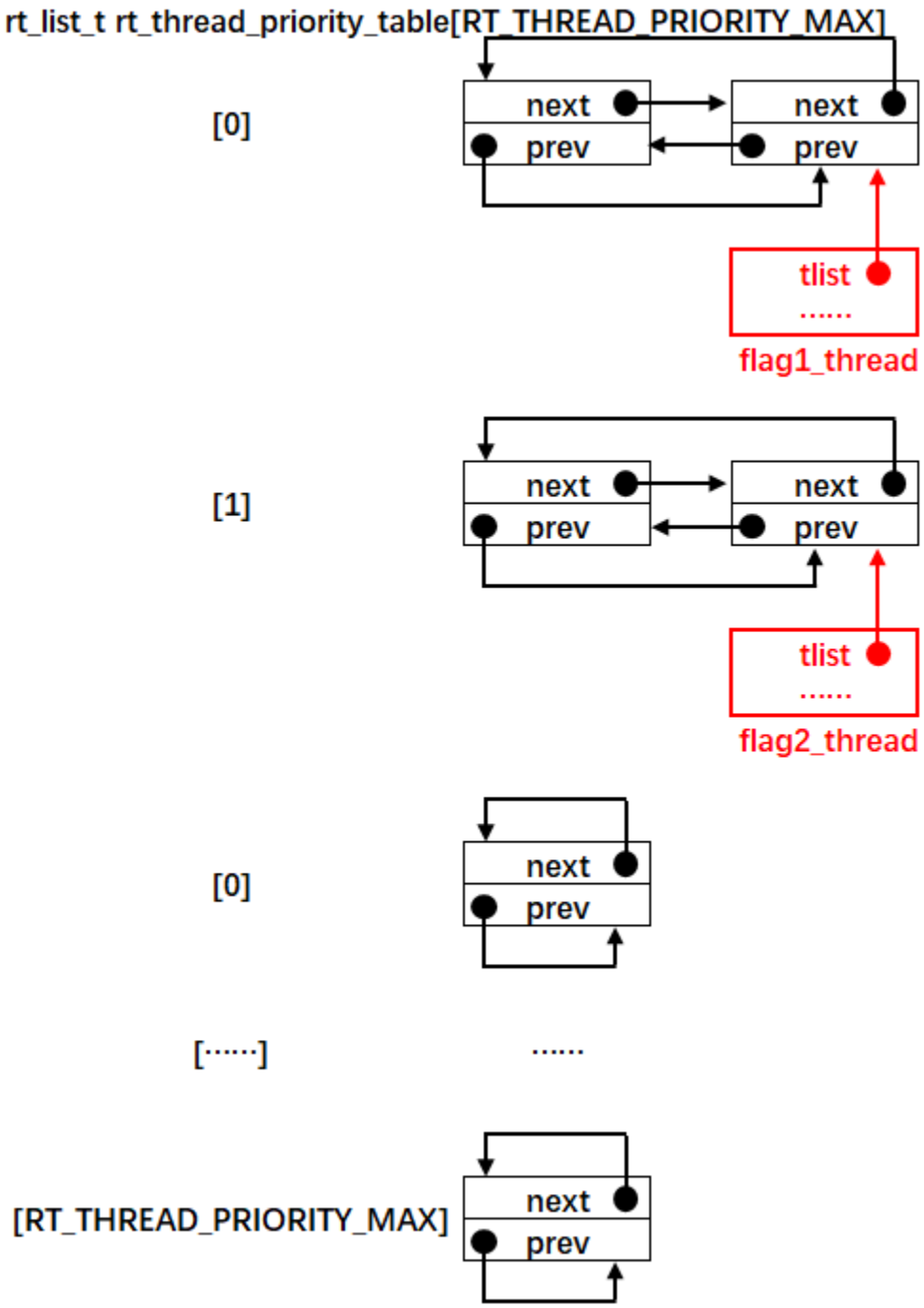
线程控制块里面有一个 tlist 成员，数据类型为 rt_list_t，我们将线程插入到就绪列表里面，就是通过将线程控制块的 tlist 这个节点插入到就绪列表中来实现的。如果把就绪列表比作是晾衣杆，线程是衣服，那 tlist 就是晾衣架，每个线程都自带晾衣架，就是为了把自己挂在各种不同的链表中。

在本章实验中，我们在线程创建好之后，紧跟着将线程插入到就绪列表，具体实现见代码清单 *6-21* 的高亮部分。

列表 21: 代码清单 6-21 将线程插入到就绪列表

```
/* 初始化线程 */
rt_thread_init( &rt_flag1_thread,                    /* 线程控制块 */
                flag1_thread_entry,                 /* 线程入口地址 */
                RT_NULL,                            /* 线程形参 */
                &rt_flag1_thread_stack[0],          /* 线程栈起始地址 */
                sizeof(rt_flag1_thread_stack) );    /* 线程栈大小，单位为字节 */
/* 将线程插入到就绪列表 */
rt_list_insert_before( &(rt_thread_priority_table[0]),&(rt_flag1_thread.
↪tlist) );

/* 初始化线程 */
rt_thread_init( &rt_flag2_thread,                    /* 线程控制块 */
                flag2_thread_entry,                 /* 线程入口地址 */
                RT_NULL,                            /* 线程形参 */
                &rt_flag2_thread_stack[0],          /* 线程栈起始地址 */
                sizeof(rt_flag2_thread_stack) );    /* 线程栈大小，单位为字节 */
/* 将线程插入到就绪列表 */
rt_list_insert_before( &(rt_thread_priority_table[1]),&(rt_flag2_thread.
↪tlist) );
```

就绪列表的下标对应的是线程的优先级，但是目前我们的线程还不支持优先级，有关支持多优先级的知识点我们后面会讲到，所以 flag1 和 flag2 线程在插入到就绪列表的时候，可以随便选择插入的位置。在代码清单 *6-21* 中，我们选择将 flag1 线程插入到就绪列表下标为 0 的链表中，flag2 线程插入到就绪列表下标为 1 的链表中，具体的示意图见图 *6-13*。


rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX]
[0]
next
prev
next
prev
tlist
.....
flag1_thread
[1]
next
prev
next
prev
tlist
.....
flag2_thread
[0]
next
prev
[.....]
.....
[RT_THREAD_PRIORITY_MAX]
next
prev

8.5 实现调度器

调度器是操作系统的核心，其主要功能就是实现线程的切换，即从就绪列表里面找到优先级最高的线程，然后去执行该线程。从代码上来看，调度器无非也就是由几个全局变量和一些可以实现线程切换的函数组成，全部都在 scheduler.c 文件中实现。

8.5.1 调度器初始化

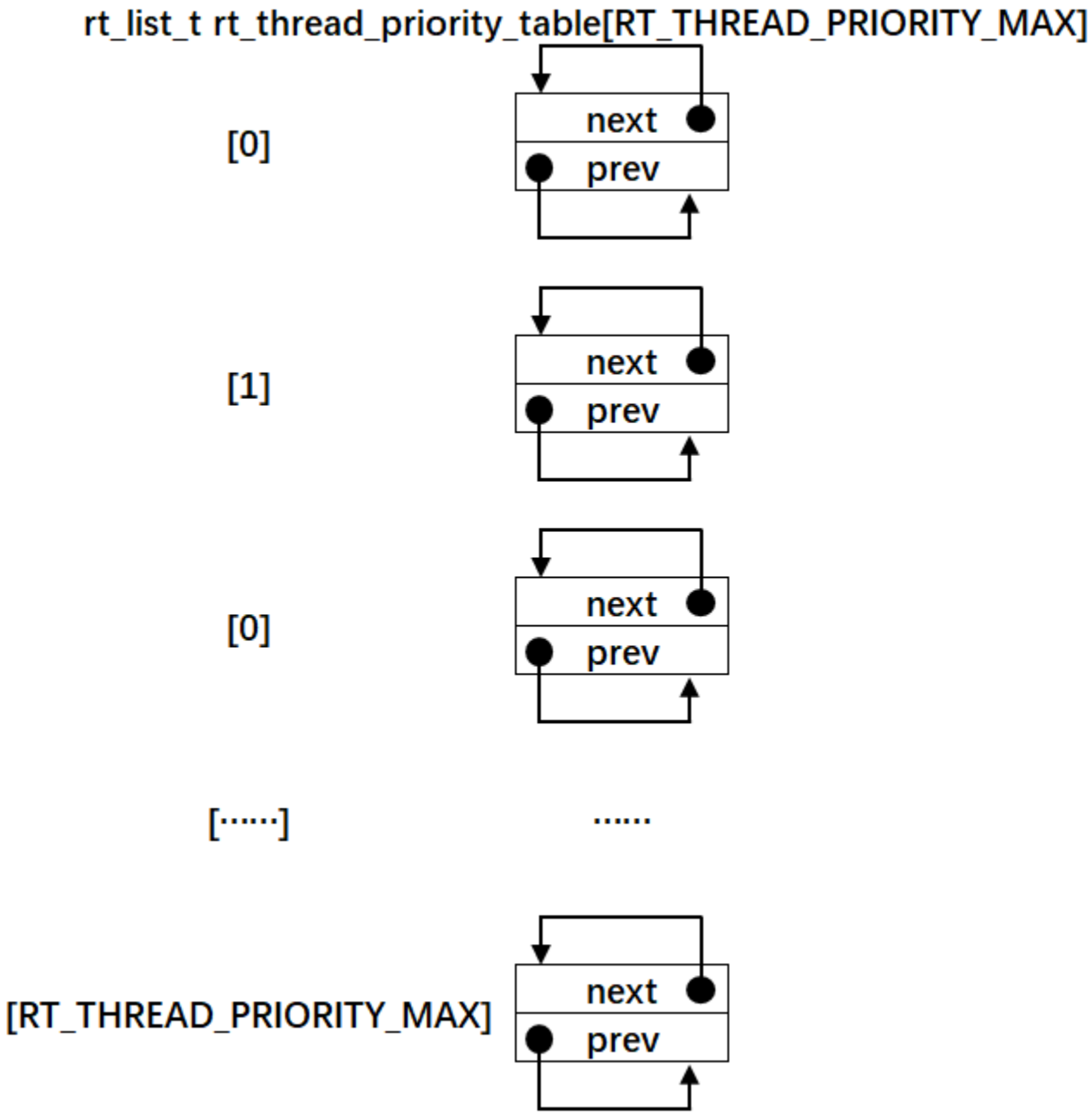
调度器在使用之前必须先初始化，具体见代码清单 6-22。

列表 22: 代码清单 6-22 调度器初始化函数

```
/* 初始化系统调度器 */
void rt_system_scheduler_init(void)
{
    register rt_base_t offset;//                                        (1)


    /* 线程就绪列表初始化 */
    for (offset ==== 0; offset < RT_THREAD_PRIORITY_MAX; offset ++)//     (2)
    {
          rt_list_init(&rt_thread_priority_table[offset]);
    }

    /* 初始化当前线程控制块指针 */
    rt_current_thread ==== RT_NULL;//                                   (3)
}
```

代码清单 6-22**(1)**：定义一个局部变量，用 C 语言关键词 register 修饰，防止被编译器优化。

代码清单 6-22**(2)**：初始化线程就绪列表，初始化完后，整个就绪列表为空，具体见图 *6-14*。



代码清单 6-22(**3**)：初始化当前线程控制块指针为空。rt_current_thread 是在 scheduler.c 中定义的一个 struct rt_thread 类型的全局指针，用于指向当前正在运行的线程的线程控制块。

在本章实验中，我们把调度器初始化放在硬件初始化之后，线程创建之前，具体见*代码清单 6-23* 的加粗部分。

列表 23: 代码清单 6-23 调度器初始化

```
int main(void)
{
    /* 硬件初始化 */
    /* 将硬件相关的初始化放在这里，如果是软件仿真则没有相关初始化代码 */

    /* 调度器初始化 */
    rt_system_scheduler_init();

    /* 初始化线程 */
    rt_thread_init( &rt_flag1_thread,                    /* 线程控制块 */
                    flag1_thread_entry,                 /* 线程入口地址 */
                    RT_NULL,                            /* 线程形参 */
                    &rt_flag1_thread_stack[0],          /* 线程栈起始地址 */
                    sizeof(rt_flag1_thread_stack) );    /* 线程栈大小，单位为字节 */
    /* 将线程插入到就绪列表 */
    rt_list_insert_before( &(rt_thread_priority_table[0]),&(rt_flag1_thread.
↪tlist) );

    /* 初始化线程 */
    rt_thread_init( &rt_flag2_thread,                    /* 线程控制块 */
                    flag2_thread_entry,                 /* 线程入口地址 */
                    RT_NULL,                            /* 线程形参 */
                    &rt_flag2_thread_stack[0],          /* 线程栈起始地址 */
                    sizeof(rt_flag2_thread_stack) );    /* 线程栈大小，单位为字节 */
    /* 将线程插入到就绪列表 */
    rt_list_insert_before( &(rt_thread_priority_table[1]),&(rt_flag2_thread.
↪tlist) );

    /* 启动系统调度器 */
    rt_system_scheduler_start();
```

(下页继续)

(续上页)

```
}
```

8.5.2 启动调度器

调度器启动由函数 rt_system_scheduler_start() 来完成，具体实现见代码清单 6-24。

列表 24: 代码清单 6-24 启动调度器函数

```
/* 启动系统调度器 */
void rt_system_scheduler_start(void)
{
    register struct rt_thread *to_thread;


    /* 手动指定第一个运行的线程 */// (1)
    to_thread ==== rt_list_entry(rt_thread_priority_table[0].next,
                        struct rt_thread,
                        tlist);
    rt_current_thread ==== to_thread;// (2)

    /* 切换到第一个线程，该函数在 context_rvds.S 中实现，在 rthw.h 声明，
       用于实现第一次任务切换。当一个汇编函数在 C 文件中调用的时候，
       如果有形参，则执行的时候会将形参传入到 CPU 寄存器 r0。*/
    rt_hw_context_switch_to((rt_uint32_t)&to_thread->sp); // (3)
}
```

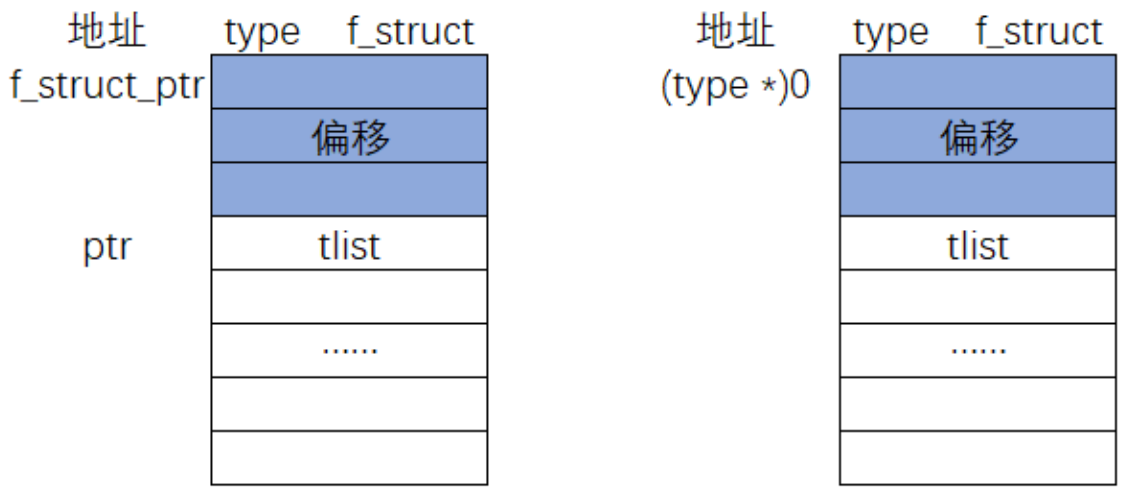
代码清单 6-24 **(1)**：调度器在启动的时候会从就绪列表中取出优先级最高的线程的线程控制块，然后切换到该线程。但是目前我们的线程还不支持优先级，那么就手动指定第一个运行的线程为就绪列表下标为 0 这条链表里面挂着的线程。rt_list_entry() 是一个已知一个结构体里面的成员的地址，反推出该结构体的首地址的宏，在 scheduler.c 开头定义，具体实现见代码清单 6-25。

列表 25: 代码清单 6-25 rt_list_entry 宏定义

```
/* 已知一个结构体里面的成员的地址，反推出该结构体的首地址 */
#define rt_container_of(ptr, type, member) \//                    (2)
    ((type *)((char *)(ptr) - (unsigned long)(&((type *)0)->member)))

#define rt_list_entry(node, type, member) \//                     (1)
    rt_container_of(node, type, member)
```

代码清单 6-25**(1)**：node 表示一个节点的地址，type 表示该节点所在的结构体的类型，member 表示该节点在该结构体中的成员名称。

代码清单 6-25**(2)**：rt_container_of() 的实现算法具体见图 *6-15*。



在图 *6-15* 中，我们知道了一个节点 tlist 的地址 ptr，现在要推算出该节点所在的 type 类型的结构体的起始地址 f_struct_ptr。我们可以将 ptr 的值减去图中灰色部分的偏移的大小就可以得到 f_struct_ptr 的地址，现在的关键是如何计算出灰色部分的偏移大小。这里采取的做法是将 0 地址强制类型类型转换为 type，即 (type*)0，然后通过指针访问结构体成员的方式获取到偏移的大小，即 (&((type*)0)->member)，最后即可算出 f_struct_ptr ==== ptr - (&((type*)0)->member)。

代码清单 6-24 **(2)**：将获取到的第一个要运行的线程控制块指针传到全局变量 rt_current_thread 中。

8.5.3 第一次线程切换

8.5.3.1 rt_hw_context_switch_to() 函数

代码清单 6-24 **(3)**：第一次切换到新的线程，该函数在 context_rvds.s 中实现（context_rvds.S 文件第一次使用需要在 rtthread3.0.3libcpuarmcortex-m3(cortex-m4 或者 cortex-m7) 中新建，然后添加到工程 rtt/ports 组中），在 rthw.h 声明，用于实现第一次线程切换。当一个汇编函数在 C 文件中调用的时候，如果有一个形参，则执行的时候会将这个形参传入到 CPU 寄存器 r0，如果有两个形参，第二个则传入到 r1。rt_hw_context_switch_to() 的具体实现见*代码清单 6-26*。context_rvds.s 文件中涉及到的 ARM 汇编指令具体参考表格 6-1。

表格 6-1 ARM 常用汇编指令讲解

指令名称	作用
EQU	给数字常量取一个符号名，相当于 C 语言中的 define
AREA	汇编一个新的代码段或者数据段
SPACE	分配内存空间
PRESERVE8	当前文件栈需按照 8 字节对齐
EXPORT	声明一个标号具有全局属性，可被外部的文件使用
DCD	以字为单位分配内存，要求 4 字节对齐，并要求初始化这些内存
PROC	定义子程序，与 ENDP 成对使用，表示子程序结束
WEAK	弱定义，如果外部文件声明了一个标号，则优先使用外部文件定义的标号，如果外部文
IMPORT	声明标号来自外部文件，跟 C 语言中的 EXTERN 关键字类似
B	跳转到一个标号
ALIGN	编译器对指令或者数据的存放地址进行对齐，一般需要跟一个立即数，缺省表示 4 字节
END	到达文件的末尾，文件结束
IF,ELSE,ENDIF	汇编条件分支语句，跟 C 语言的 if else 类似
MRS	加载特殊功能寄存器的值到通用寄存器
MSR	存储通用寄存器的值到特殊功能寄存器
CBZ	比较，如果结果为 0 就转移

指令名称	作用
CBNZ	比较，如果结果非 0 就转移
LDR	从存储器中加载字到一个寄存器中
LDR[伪指令]	加载一个立即数或者一个地址值到一个寄存器。举例：LDR Rd, ==== label，如果 label 是
LDRH	从存储器中加载半字到一个寄存器中
LDRB	从存储器中加载字节到一个寄存器中
STR	把一个寄存器按字存储到存储器中
STRH	把一个寄存器存器的低半字存储到存储器中
STRB	把一个寄存器的低字节存储到存储器中
LDMIA	加载多个字，并且在加载后自增基址寄存器
STMIA	存储多个字，并且在存储后自增基址寄存器
ORR	按位或
BX	直接跳转到由寄存器给定的地址
BL	跳转到标号对应的地址，并且把跳转前的下条指令地址保存到 LR
BLX	跳转到由寄存器 REG 给出的的地址，并根据 REG 的 LSB 切换处理器状态，还要把转移

列表 26: 代码清单 6-26 rt_hw_context_switch_to() 函数

```
;*****************************************************************************
;                                    全局变量                                 (4)
;*****************************************************************************
    IMPORT rt_thread_switch_interrupt_flag
    IMPORT rt_interrupt_from_thread
    IMPORT rt_interrupt_to_thread

;*****************************************************************************
;                                    常量                                     (5)
;*****************************************************************************
;-----------------------------------------------------------------------------
; 系统控制块外设 SCB 地址范围: 0xE000ED00-0xE000ED3F
```

(下页继续)

(续上页)

```
;-------------------------------------------------------------------------
SCB_VTOR        EQU     0xE000ED08      ; 向量表偏移寄存器
NVIC_INT_CTRL   EQU     0xE000ED04      ; 中断控制状态寄存器
NVIC_SYSPRI2    EQU     0xE000ED20      ; 系统优先级寄存器(2)
NVIC_PENDSV_PRI EQU     0x00FF0000      ; PendSV 优先级值 (lowest)
NVIC_PENDSVSET  EQU     0x10000000      ; 触发 PendSV exception 的值

;*************************************************************************
;                                代码产生指令                                (1)
;*************************************************************************

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

;/*
; *-----------------------------------------------------------------------
; * 函数原型: void rt_hw_context_switch_to(rt_uint32 to);
; * r0 --> to
; * 该函数用于开启第一次线程切换
; *-----------------------------------------------------------------------
; */

rt_hw_context_switch_to    PROC                                            (6)

   ; 导出 rt_hw_context_switch_to, 让其具有全局属性, 可以在 C 文件调用
   EXPORT rt_hw_context_switch_to                                          (7)

   ; 设置 rt_interrupt_to_thread 的值                                        (8)
   ; 将 rt_interrupt_to_thread 的地址加载到 r1
   LDR     r1, ====rt_interrupt_to_thread
↪(8)-1
```

(下页继续)

(续上页)

```
    ; 将 r0 的值存储到 rt_interrupt_to_thread
    STR     r0, [r1]                                                    (8)-2

    ; 设置 rt_interrupt_from_thread 的值为 0，表示启动第一次线程切换          (9)
    ; 将 rt_interrupt_from_thread 的地址加载到 r1
    LDR     r1, ====rt_interrupt_from_thread                            ␣
↪(9)-1
    ; 配置 r0 等于 0
    MOV     r0, #0x0                                                    (9)-2
    ; 将 r0 的值存储到 rt_interrupt_from_thread
    STR     r0, [r1]                                                    (9)-3

    ; 设置中断标志位 rt_thread_switch_interrupt_flag 的值为 1              (10)
    ; 将 rt_thread_switch_interrupt_flag 的地址加载到 r1
    LDR     r1, ====rt_thread_switch_interrupt_flag                     ␣
↪(10)-1
    ; 配置 r0 等于 1
    MOV     r0, #1                                                      (10)-2
    ; 将 r0 的值存储到 rt_thread_switch_interrupt_flag
    STR     r0, [r1]                                                    (10)-3

    ; 设置 PendSV 异常的优先级
    LDR     r0, ====NVIC_SYSPRI2
    LDR     r1, ====NVIC_PENDSV_PRI
    LDR.W   r2, [r0,#0x00]          ; 读
    ORR     r1,r1,r2                ; 改
    STR     r1, [r0]                ; 写

    ; 触发 PendSV 异常（产生上下文切换）                                    (12)
    LDR     r0, ====NVIC_INT_CTRL
    LDR     r1, ====NVIC_PENDSVSET
    STR     r1, [r0]
```

(下页继续)

(续上页)

```
75
76      ; 开中断
77      CPSIE   F                                                    (13)
78      CPSIE   I
79
80      ; 永远不会到达这里
81      ENDP                                                         (14)
82
83      ALIGN   4                                                    (3)
84
85      END                                                          (2)
```

代码清单 6-26 **(1)**：汇编代码产生指令，当我们新建一个汇编文件写代码时，必须包含类似的指令。AERA 表示汇编一个新的数据段或者代码段，.text 表示段名字，如果段名不是以字母开头，而是以其它符号开头则需要在段名两边加上‘|’，CODE 表示为代码，READONLY 表示只读，ALIGN====2，表示当前文件指令要 2:sup:*2* 字节对齐。THUMB 表示 THUMB 指令代码，REQUIRE8 和 PRESERVE8 均表示当前文件的栈按照 8 字节对齐。

代码清单 6-26 **(2)**：汇编文件结束，每个汇编文件都需要一个 END。

代码清单 6-26 **(3)**：当前文件指令代码要求 4 字节对齐，不然会有警告。

代码清单 6-26 **(4)**：使用 IMPORT 关键字导入一些全局变量，这三个全局变量在 cpuport.c 中定义，具体见代码清单 6-27，每个变量的含义具体看注释。

列表 27: 代码清单 6-27 汇编文件导入的三个全局变量定义

```
1 /* 用于存储上一个线程的栈的 sp 的指针 */
2 rt_uint32_t rt_interrupt_from_thread;
3
4 /* 用于存储下一个将要运行的线程的栈的 sp 的指针 */
5 rt_uint32_t rt_interrupt_to_thread;
```

(下页继续)

(续上页)

```

/* PendSV 中断服务函数执行标志 */
rt_uint32_t rt_thread_switch_interrupt_flag;
```

代码清单 6-26 **(5)**：定义了一些常量，这些都是内核里面的寄存器，等下触发 PendSV 异常会用到。有关内核外设寄存器定义可参考官方文档：STM32F10xxx Cortex-M3 programming manual—4 Core peripherals，无论是 M3/4/7 内核均可以参考该文档。

代码清单 6-26 **(6)**：PROC 用于定义子程序，与 ENDP 成对使用，表示 rt_hw_context_switch_to() 函数开始。

代码清单 6-26 **(7)**：使用 EXPORT 关键字导出 rt_hw_context_switch_to，让其具有全局属性，可以在 C 文件调用（但也要先在 rthw.h 中声明）。

代码清单 6-26 **(8)**：设置 rt_interrupt_to_thread 的值。

代码清单 6-26 **(8)-1**：将 rt_interrupt_to_thread 的地址加载到 r1。

代码清单 6-26 **(8)-2**：将 r0 的值存储到 rt_interrupt_to_thread，r0 存的是下一个将要运行的线程的 sp 的地址，由 rt_hw_context_switch_to((rt_uint32_t)&to_thread->sp) 调用的时候传到 r0。

代码清单 6-26 **(9)**：设置 rt_interrupt_from_thread 的值为 0，表示启动第一次线程切换。

代码清单 6-26 **(9)-1**：将 rt_interrupt_from_thread 的地址加载到 r1。

代码清单 6-26 **(9)-2**：配置 r0 等于 0。

代码清单 6-26 **(9)-3**：将 r0 的值存储到 rt_interrupt_from_thread。

代码清单 6-26 **(10)**：设置中断标志位 rt_thread_switch_interrupt_flag 的值为 1，当执行了 PendSVC Handler 时，rt_thread_switch_interrupt_flag 的值会被清 0。

代码清单 6-26 **(10)-1**：将 rt_thread_switch_interrupt_flag 的地址加载到 r1。

代码清单 6-26 **(10)-2**：配置 r0 等于 1。

代码清单 6-26 **(10)-3**：将 r0 的值存储到 rt_thread_switch_interrupt_flag。

代码清单 6-26 **(11)**：设置 PendSV 异常的优先级为最低。

代码清单 6-26 **(12)**：触发 PendSV 异常 (产生上下文切换)。如果前面关了，还要等中断打开才能去执行 PendSV 中断服务函数。

代码清单 6-26 **(13)**：开中断。

代码清单 6-26 **(14)**：rt_hw_context_switch_to() 程序结束，与 PROC 成对使用。

8.5.3.2 PendSV_Handler() 函数

PendSV_Handler() 函数是真正实现线程上下文切换的地方，具体实现见代码清单 6-28。

列表 28: 代码清单 6-28 PendSV_Handler() 函数

```
;/*
; *------------------------------------------------------------------------
; * void PendSV_Handler(void);
; * r0 --> switch from thread stack
; * r1 --> switch to thread stack
; * psr, pc, lr, r12, r3, r2, r1, r0 are pushed into [from] stack
; *------------------------------------------------------------------------
; */

PendSV_Handler   PROC
EXPORT PendSV_Handler

; 失能中断，为了保护上下文切换不被中断                                    (1)
MRS      r2, PRIMASK
CPSID    I

; 获取中断标志位，看看是否为 0                                            (2)
; 加载 rt_thread_switch_interrupt_flag 的地址到 r0
LDR      r0, ====rt_thread_switch_interrupt_flag                          (2)-1
; 加载 rt_thread_switch_interrupt_flag 的值到 r1
LDR      r1, [r0]                                                         (2)-2
```

(下页继续)

(续上页)

```
; 判断 r1 是否为 0，为 0 则跳转到 pendsv_exit
CBZ     r1, pendsv_exit                                    (2)-3

; r1 不为 0 则清 0                                            (3)
MOV     r1, #0x00
; 将 r1 的值存储到 rt_thread_switch_interrupt_flag，即清 0
STR     r1, [r0]
; 判断 rt_interrupt_from_thread 的值是否为 0                  (4)
; 加载 rt_interrupt_from_thread 的地址到 r0
LDR     r0, ====rt_interrupt_from_thread                     (4)-1
; 加载 rt_interrupt_from_thread 的值到 r1
LDR     r1, [r0]                                           (4)-2
; 判断 r1 是否为 0，为 0 则跳转到 switch_to_thread
; 第一次线程切换时 rt_interrupt_from_thread 肯定为 0，则跳转到 switch_to_thread
CBZ     r1, switch_to_thread                               (4)-3

; ================ 上文保存 ====================================    (6)
; 当进入 PendSVC Handler 时，上一个线程运行的环境即：
; xPSR，PC（线程入口地址），R14，R12，R3，R2，R1，R0（线程的形参）
; 这些 CPU 寄存器的值会自动保存到线程的栈中，剩下的 r4~r11 需要手动保存
; 获取线程栈指针到 r1
MRS     r1, psp                                            (6)-1
; 将 CPU 寄存器 r4~r11 的值存储到 r1 指向的地址(每操作一次地址将递减一次)
STMFD   r1!, {r4 - r11}                                    (6)-2
; 加载 r0 指向值到 r0，即 r0====rt_interrupt_from_thread
LDR     r0, [r0]                                           (6)-3
; 将 r1 的值存储到 r0，即更新线程栈 sp
STR     r1, [r0]                                           (6)-4

; ================= 下文切换 ================================   (5)
switch_to_thread
; 加载 rt_interrupt_to_thread 的地址到 r1
```

(下页继续)

(续上页)

```
; rt_interrupt_to_thread 是一个全局变量，里面存的是线程栈指针 SP 的指针
LDR     r1, ====rt_interrupt_to_thread                          (5)-1
; 加载 rt_interrupt_to_thread 的值到 r1，即 sp 指针的指针
LDR     r1, [r1]                                                (5)-2
; 加载 rt_interrupt_to_thread 的值到 r1，即 sp
LDR     r1, [r1]                                                (5)-3

; 将线程栈指针 r1(操作之前先递减)指向的内容加载到 CPU 寄存器 r4~r11
LDMFD   r1!, {r4 - r11}                                         (5)-4
; 将线程栈指针更新到 PSP
MSR     psp, r1                                                 (5)-5

pendsv_exit
; 恢复中断
MSR     PRIMASK, r2                                             (7)

; 确保异常返回使用的栈指针是 PSP，即 LR 寄存器的位 2 要为 1
ORR     lr, lr, #0x04                                           (8)
; 异常返回，这个时候栈中的剩下内容将会自动加载到 CPU 寄存器:
; xPSR，PC (线程入口地址)，R14，R12，R3，R2，R1，R0 (线程的形参)
; 同时 PSP 的值也将更新，即指向线程栈的栈顶
BX      lr                                                      (9)

; PendSV_Handler 子程序结束
ENDP                                                            (10)
```

代码清单 6-28 **(1)**：失能中断，为了保护上下文切换不被中断。

代码清单 6-28 **(2)**：获取中断标志位 rt_thread_switch_interrupt_flag 是否为 0，如果为 0 则退出 PendSV Handler，如果不为 0 则继续往下执行。

代码清单 6-28 **(2)-1**：加载 rt_thread_switch_interrupt_flag 的地址到 r0。

代码清单 6-28 **(2)-2**：加载 rt_thread_switch_interrupt_flag 的值到 r1。

代码清单 6-28 **(2)-3**：判断 r1 是否为 0，为 0 则跳转到 pendsv_exit，退出 PendSV Handler 函数。

代码清单 6-28 **(3)**：中断标志位 rt_thread_switch_interrupt_flag 清 0。

代码清单 6-28 **(4)**：判断 rt_interrupt_from_thread 的值是否为 0，如果为 0 则表示第一次线程切换，不用做上文保存的工作，直接跳转到 switch_to_thread 执行下文切换即可。如果不为 0 则需要先保存上文，然后再切换到下文。

代码清单 6-28 **(4)-1**：加载 rt_interrupt_from_thread 的地址到 r0。

代码清单 6-28 **(4)-2**：加载 rt_interrupt_from_thread 的值到 r1。

代码清单 6-28 **(4)-3**：判断 r1 是否为 0，为 0 则跳转到 switch_to_thread，第一次线程切换时 rt_interrupt_from_thread 肯定为 0，则跳转到 switch_to_thread。

代码清单 6-28 **(5)**：下文切换。下文切换实际上就是把接下来要运行的线程栈里面的内容加载到 CPU 寄存器，更改 PC 指针和 PSP 指针，从而实现程序的跳转。

代码清单 6-28 **(5)-1**：加载 rt_interrupt_to_thread 的地址到 r1，rt_interrupt_to_thread 是一个全局变量，里面存的是线程栈指针 SP 的指针。

代码清单 6-28 **(5)-2**：加载 rt_interrupt_to_thread 的值到 r1，即 sp 的指针。

代码清单 6-28 **(5)-3**：加载 rt_interrupt_to_thread 的值到 r1，即 sp。

代码清单 6-28 **(5)-4**：将线程栈指针 r1(操作之前先递减) 指向的内容加载到 CPU 寄存器 r4~r11。

代码清单 6-28 **(5)-5**：将线程栈指针更新到 PSP。

代码清单 6-28 **(6)**：rt_interrupt_from_thread 的值不为 0 则表示不是第一次线程切换，需要先保存上文。当进入 PendSVCHandler 时，上一个线程运行的环境即：xPSR，PC（线程入口地址），R14，R12，R3，R2，R1，R0（线程的形参）这些 CPU 寄存器的值会自动保存到线程的栈中，并更新 PSP 的值，剩下的 r4~r11 需要手动保存。

代码清单 6-28 **(6)-1**：获取线程栈指针到 r1。

代码清单 6-28 **(6)-2**：将 CPU 寄存器 r4~r11 的值存储到 r1 指向的地址 (每操作一次地址将递减一次)。

代码清单 6-28 **(6)-3**：加载 r0 指向值到 r0，即 r0====rt_interrupt_from_thread。

代码清单 6-28 **(6)-4**：将 r1 的值存储到 r0，即更新线程栈 sp。

代码清单 6-28 **(7)**：上下文切换完成，恢复中断。

代码清单 6-28 **(8)**：确保异常返回使用的栈指针是 PSP，即 LR 寄存器的位 2 要为 1。

代码清单 6-28 **(9)**：异常返回，这个时候接下来将要运行的线程栈中的剩下内容将会自动加载到 CPU 寄存器：xPSR，PC（线程入口地址），R14，R12，R3，R2，R1，R0（线程的形参）同时 PSP 的值也将更新，即指向线程栈的栈顶。

代码清单 6-28 **(10)**：上下文切换完成，恢复中断。

8.5.4 系统调度

系统调度就是在就绪列表中寻找优先级最高的就绪线程，然后去执行该线程。但是目前我们还不支持优先级，仅实现两个线程轮流切换，系统调度函数 rt_schedule 具体实现见代码清单 6-29。

8.5.4.1 rt_schedule() 函数

列表 29: 代码清单 6-29 rt_schedule() 函数

```
/* 系统调度 */
void rt_schedule(void)
{
    struct rt_thread *to_thread;
    struct rt_thread *from_thread;

    /* 两个线程轮流切换 */// (1)
    if( rt_current_thread ======== rt_list_entry( rt_thread_priority_table[0].next,
                                                  struct rt_thread,
                                                  tlist) )
    {
        from_thread ==== rt_current_thread;
```

（下页继续）

(续上页)

```
13        to_thread ==== rt_list_entry( rt_thread_priority_table[1].next,
14                                                        struct rt_thread,
15                                                        tlist);
16    rt_current_thread ==== to_thread;
17    }
18    else//                                                        (2)
19    {
20        from_thread ==== rt_current_thread;
21        to_thread ==== rt_list_entry( rt_thread_priority_table[0].next,
22                                                        struct rt_thread,
23                                                        tlist);
24    rt_current_thread ==== to_thread;
25    }//                                                           (3)
26    /* 产生上下文切换 */
27    rt_hw_context_switch((rt_uint32_t)&from_thread->sp,(rt_uint32_t)&to_
↪thread->sp);
28 }
```

代码清单 6-29**(1)**：如果当前线程为线程 1，则把下一个要运行的线程改为线程 2。

代码清单 6-29**(2)**：如果当前线程为线程 2，则把下一个要运行的线程改为线程 1。

8.5.4.2 rt_hw_contex_switch() 函数

代码清单 6-29**(3)**：产生上下文切换。rt_hw_contex_switch() 函数用于产生上下文切换，在 context_rvds.S 中实现，在 rthw.h 声明。当一个汇编函数在 C 文件中调用的时候，如果有两个个形参，则执行的时候会将这个形参传传入到 CPU 寄存器 r0、r1。rt_hw_context_switch() 具体实现见代码清单 6-30。

列表 30: 代码清单 6-30 rt_hw_context_switch() 函数

```
;/*
; *-----------------------------------------------------------------------
; * void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);
; * r0 --> from
; * r1 --> to
; *-----------------------------------------------------------------------
; */
rt_hw_context_switch    PROC
EXPORT rt_hw_context_switch

; 设置中断标志位 rt_thread_switch_interrupt_flag 为 1          (1)
; 加载 rt_thread_switch_interrupt_flag 的地址到 r2
LDR     r2, ====rt_thread_switch_interrupt_flag              (1)-1
; 加载 rt_thread_switch_interrupt_flag 的值到 r3
LDR     r3, [r2]                                            (1)-2
;r3 与 1 比较，相等则执行 BEQ 指令，否则不执行
CMP     r3, #1                                              (1)-3
BEQ     _reswitch
; 设置 r3 的值为 1
MOV     r3, #1                                              (1)-4
; 将 r3 的值存储到 rt_thread_switch_interrupt_flag，即置 1
STR     r3, [r2]                                            (1)-5

; 设置 rt_interrupt_from_thread 的值                          (2)
; 加载 rt_interrupt_from_thread 的地址到 r2
LDR     r2, ====rt_interrupt_from_thread                     (2)-1
; 存储 r0 的值到 rt_interrupt_from_thread，即上一个线程栈指针 sp 的指针
STR     r0, [r2]                                            (2)-2

_reswitch
```

(下页继续)

(续上页)

```
; 设置 rt_interrupt_to_thread 的值                                (3)
; 加载 rt_interrupt_from_thread 的地址到 r2
LDR     r2, ====rt_interrupt_to_thread                            (3)-1
; 存储 r1 的值到 rt_interrupt_from_thread，即下一个线程栈指针 sp 的指针
STR     r1, [r2]                                                 (3)-2

; 触发 PendSV 异常，实现上下文切换                                (4)
LDR     r0, ====NVIC_INT_CTRL
LDR     r1, ====NVIC_PENDSVSET
STR     r1, [r0]
; 子程序返回
BX      LR                                                       (5)
; 子程序结束
ENDP                                                             (6)
```

代码清单 6-30**(1)**：设置中断标志位 rt_thread_switch_interrupt_flag 为 1。

代码清单 6-30**(1)-1**：加载 rt_thread_switch_interrupt_flag 的地址到 r2。

代码清单 6-30**(1)-2**：加载 rt_thread_switch_interrupt_flag 的值到 r3。

代码清单 6-30**(1)-3**：r3 与 1 比较，相等则执行 BEQ 指令，否则不执行。

代码清单 6-30**(1)-4**：设置 r3 的值为 1。

代码清单 6-30**(1)-5**：将 r3 的值加载到 rt_thread_switch_interrupt_flag，即置 1。

代码清单 6-30**(2)**：设置 rt_interrupt_from_thread 的值。

代码清单 6-30**(2)-1**：加载 rt_interrupt_from_thread 的地址到 r2。

代码清单 6-30**(2)-2**：存储 r0 的值到 rt_interrupt_from_thread，即上一个线程栈指针 sp 的指针。r0 存储的是函数调用 rt_hw_context_switch((rt_uint32_t)&from_thread->sp,(rt_uint32_t)&to_th read->sp) 时的第一个形参，即上一个线程栈指针 sp 的指针。

代码清单 6-30**(3)**：设置 rt_interrupt_to_thread 的值

代码清单 6-30**(3)-1**：加载 rt_interrupt_from_thread 的地址到 r2

代码清单 6-30**(3)-2**：存储 r1 的值到 rt_interrupt_from_thread，即下一个线程栈指针 sp 的指针。r1 存储的是函数调用 rt_hw_context_switch((rt_uint32_t)&from_thread->sp,(rt_uint32_t)&to_th read->sp) 时的第二个形参，即下一个线程栈指针 sp 的指针。

代码清单 6-30**(4)**：触发 PendSV 异常，在 PendSV Handler 里面实现上下文切换。

代码清单 6-30**(5)**：子程序返回，返回到调用 rt_hw_context_switch_to() 函数的地方。

代码清单 6-30**(6)**：汇编程序结束。

8.6 main 函数

线程的创建，就绪列表的实现，调度器的实现均已经讲完，现在我们把全部的测试代码都放到 main.c 里面，具体见代码清单 6-31。

列表 31: 代码清单 6-31 main.c 代码

```
/**
*****************************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2019-xx-xx
* @brief    《RT-Thread 内核实现与应用开发实战指南》书籍例程
*
*****************************************************************************
* @attention
*
* 实验平台: 野火 i.MX RT 系列开发板
*
* 官网     :www.embedfire.com
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
```

(下页继续)

(续上页)

```
17  *
18  ************************************************************************
19  */
20
21  /*
22  ************************************************************************
23  *                                包含的头文件
24  ************************************************************************
25  */
26
27  #include <rtthread.h>
28  #include "ARMCM7.h"
29
30
31  /*
32  ************************************************************************
33  *                                全局变量
34  ************************************************************************
35  */
36  rt_uint8_t flag1;
37  rt_uint8_t flag2;
38
39  extern rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX];
40
41  /*
42  ************************************************************************
43  *                        线程控制块 & STACK & 线程声明
44  ************************************************************************
45  */
46
47
48  /* 定义线程控制块 */
```

(下页继续)

(续上页)

```
struct rt_thread rt_flag1_thread;
struct rt_thread rt_flag2_thread;

ALIGN(RT_ALIGN_SIZE)
/* 定义线程栈 */
rt_uint8_t rt_flag1_thread_stack[512];
rt_uint8_t rt_flag2_thread_stack[512];

/* 线程声明 */
void flag1_thread_entry(void *p_arg);
void flag2_thread_entry(void *p_arg);

/*
*************************************************************************
*                                 函数声明
*************************************************************************
*/
void delay(uint32_t count);

/*************************************************************************
* @brief  main 函数
* @param  无
* @retval 无
*
* @attention
*************************************************************************
*/
int main(void)
{
    /* 硬件初始化 */
    /* 将硬件相关的初始化放在这里，如果是软件仿真则没有相关初始化代码 */

```

(下页继续)

(续上页)

```
    /* 调度器初始化 */
    rt_system_scheduler_init();


    /* 初始化线程 */
    rt_thread_init( &rt_flag1_thread,                    /* 线程控制块 */
                    flag1_thread_entry,                 /* 线程入口地址 */
                    RT_NULL,                            /* 线程形参 */
                    &rt_flag1_thread_stack[0],          /* 线程栈起始地址 */
                    sizeof(rt_flag1_thread_stack) );    /* 线程栈大小，单位为字节 */
    /* 将线程插入到就绪列表 */
    rt_list_insert_before( &(rt_thread_priority_table[0]),&(rt_flag1_thread.tlist) );

    /* 初始化线程 */
    rt_thread_init( &rt_flag2_thread,                    /* 线程控制块 */
                    flag2_thread_entry,                 /* 线程入口地址 */
                    RT_NULL,                            /* 线程形参 */
                    &rt_flag2_thread_stack[0],          /* 线程栈起始地址 */
                    sizeof(rt_flag2_thread_stack) );    /* 线程栈大小，单位为字节 */
    /* 将线程插入到就绪列表 */
    rt_list_insert_before( &(rt_thread_priority_table[1]),&(rt_flag2_thread.tlist) );

    /* 启动系统调度器 */
    rt_system_scheduler_start();
}

/*
*****************************************************************************
*                                    函数实现
*****************************************************************************
```

(下页继续)

(续上页)

```
*/
/* 软件延时 */
void delay (uint32_t count)
{
    for(; count!====0; count--);
}

/* 线程 1 */
void flag1_thread_entry( void *p_arg )
{
    for( ;; )
    {
        flag1 ==== 1;
        delay( 100 );
        flag1 ==== 0;
        delay( 100 );

        /* 线程切换，这里是手动切换 */
        rt_schedule();//                                    (注意)
    }
}

/* 线程 2 */
void flag2_thread_entry( void *p_arg )
{
    for( ;; )
    {
        flag2 ==== 1;
        delay( 100 );
        flag2 ==== 0;
        delay( 100 );

```

(下页继续)

(续上页)

```
        /* 线程切换，这里是手动切换 */
        rt_schedule();//                                    (注意)
    }
}
```

代码清单 6-31 中的每个局部的代码均已经讲解过，剩下的看代码注释即可。

代码清单 6-31**(注意)**：因为目前还不支持优先级，每个线程执行完毕之后都主动调用系统调度函数 rt_schedule() 来实现线程的切换。

8.7 实验现象

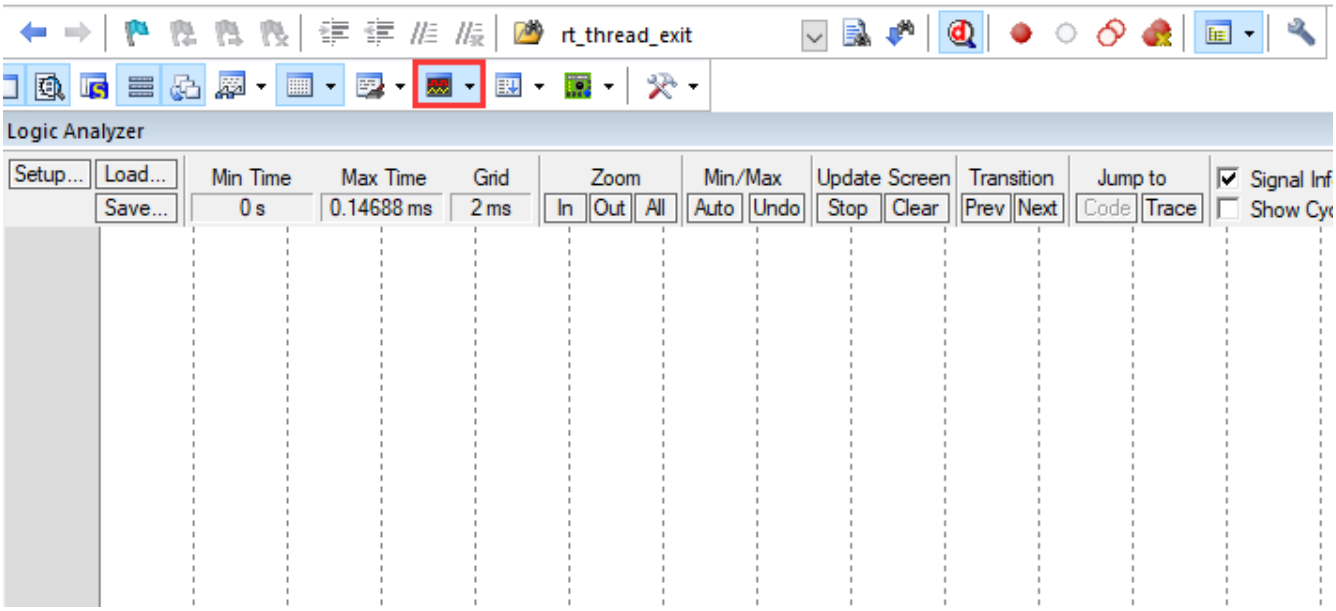
本章代码讲解完毕，接下来是软件调试仿真，具体过程见图 *6-16*、图 *6-17*、图 *6-18*、图 *6-19* 和图 *6-20*。

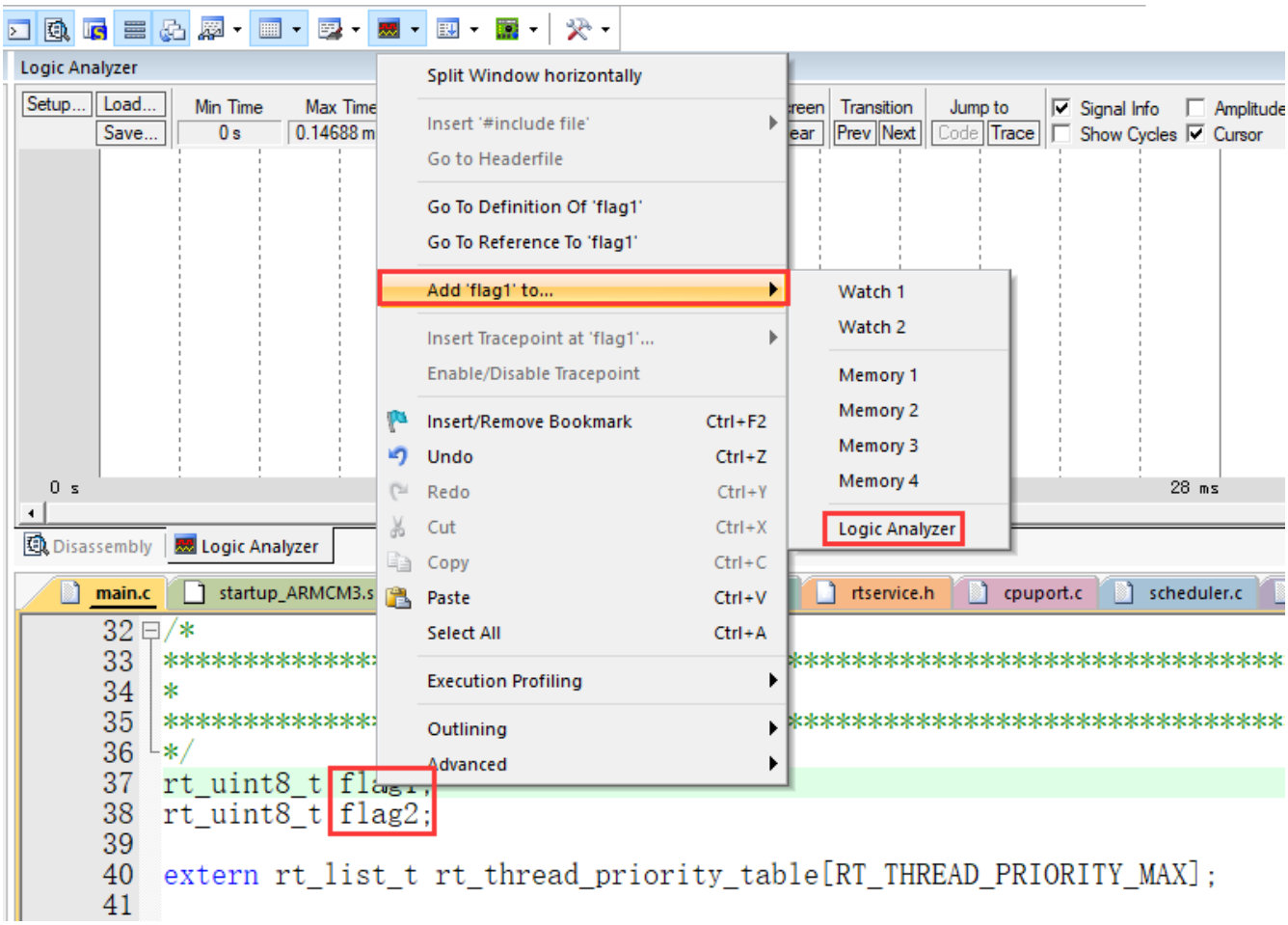
Tools SVCS Window Help

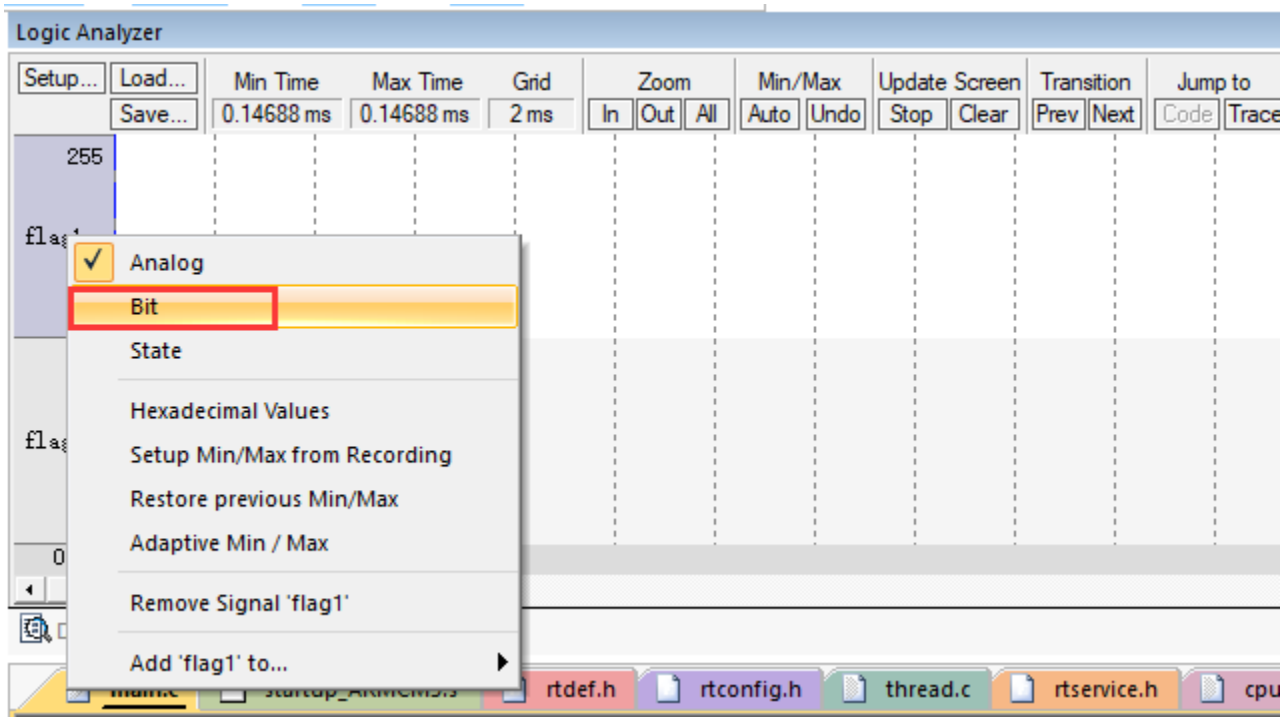
DHCP_GET

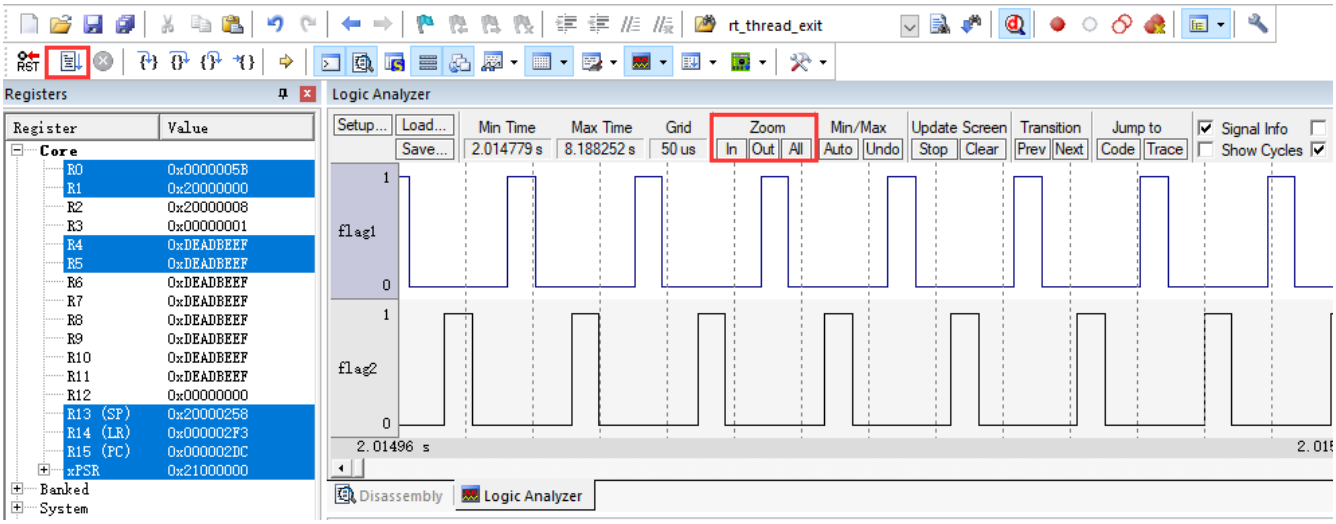
main.c | startup_ARMCM3.s | rtdef.h | rtconfig.h | thread.c | rtservice.h | cpuport.c | scheduler.c | context_rvds.s

```
  ***********************************************************************
  */
int main(void)
{
  /* 硬件初始化 */
  /* 将硬件相关的初始化放在这里，如果是软件仿真则没有相关初始化代码 */

  /* 调度器初始化 */
  rt_system_scheduler_init();


  /* 初始化线程 */
  rt_thread_init( &rt_flag1_thread,                    /* 线程控制块 */
                  flag1_thread_entry,                  /* 线程入口地址 */
                  RT_NULL,                             /* 线程形参 */
                  &rt_flag1_thread_stack[0],           /* 线程栈起始地址 */
                  sizeof(rt_flag1_thread_stack) );     /* 线程栈大小，单位为字节 */
  /* 将线程插入就绪列表 */
  rt_list_insert_before( &(rt_thread_priority_table[0]),&(rt_flag1_thread.tlist) );

  /* 初始化线程 */
  rt_thread_init( &rt_flag2_thread,                    /* 线程控制块 */
                  flag2_thread_entry,                  /* 线程入口地址 */
                  RT_NULL,                             /* 线程形参 */
                  &rt_flag2_thread_stack[0],           /* 线程栈起始地址 */
                  sizeof(rt_flag2_thread_stack) );     /* 线程栈大小，单位为字节 */
  /* 将线程插入就绪列表 */
  rt_list_insert_before( &(rt_thread_priority_table[1]),&(rt_flag2_thread.tlist) );

  /* 启动系统调度器 */
  rt_system_scheduler_start();
}
```


rt_thread_exit
Logic Analyzer
Setup...
Load...
Save...
Min Time
0 s
Max Time
0.14688 ms
Grid
2 ms
Zoom
In
Out
All
Min/Max
Auto
Undo
Update Screen
Stop
Clear
Transition
Prev
Next
Jump to
Code
Trace
Signal Inf
Show Cyc


Logic Analyzer
Split Window horizontally
Insert '#include file'
Go to Headerfile
Go To Definition Of 'flag1'
Go To Reference To 'flag1'
Add 'flag1' to...
Insert Tracepoint at 'flag1'...
Enable/Disable Tracepoint
Insert/Remove Bookmark Ctrl+F2
Undo Ctrl+Z
Redo Ctrl+Y
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Select All Ctrl+A
Execution Profiling
Outlining
Advanced
Watch 1
Watch 2
Memory 1
Memory 2
Memory 3
Memory 4
Logic Analyzer
Signal Info
Show Cycles
Amplitude
Cursor
Disassembly
main.c
startup_ARMCM3.s
rtservice.h
cpuport.c
scheduler.c
rt_uint8_t flag1;
rt_uint8_t flag2;
extern rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX];





至此，本章讲解完毕。但是，只是把本章的内容看完，然后再仿真看看波形是远远不够的，应该是把当前线程控制块指针 rt_current_thread、就绪列表 rt_thread_priority_table、每个线程的控制块、线程的入口函数和线程的栈这些变量统统添加到观察窗口，然后单步执行程序，看看这些变量是怎么变化的。特别是线程切换时，CPU 寄存器、线程栈和 PSP 这些是怎么变化的，让机器执行代码的过程在自己的脑子里面过一遍。图 *6-21* 就是我在仿真调试时的观察窗口。

Watch 1

Name	Value	Type
rt_current_thread	0x2000003C &rt_flag2_thread	struct rt_thread *
rt_thread_priority_table	0x20000458 rt_thread_priority_table	struct rt_list_node[32]
rt_flag1_thread	0x20000020 &rt_flag1_thread	struct rt_thread
flag1_thread_entry	0x000002E4	void f(void *)
rt_flag1_thread_stack	0x20000058 rt_flag1_thread_stack[] ""	unsigned char[512]
rt_flag2_thread	0x2000003C &rt_flag2_thread	struct rt_thread
flag2_thread_entry	0x00000308	void f(void *)
rt_flag2_thread_stack	0x20000258 rt_flag2_thread_stack[] ""	unsigned char[512]
<Enter expression>		

Call Stack + Locals | Watch 1 | Memory 1

第 9 章　临界段的保护

9.1 什么是临界段

临界段用一句话概括就是一段在执行的时候不能被中断的代码段。在 RT-Thread 里面，这个临界段最常出现的就是对全局变量的操作，全局变量就好像是一个枪把子，谁都可以对他开枪，但是我开枪的时候，你就不能开枪，否则就不知道是谁命中了靶子。

那么什么情况下临界段会被打断？一个是系统调度，还有一个就是外部中断。在 RT-Thread，系统调度，最终也是产生 PendSV 中断，在 PendSV Handler 里面实现线程的切换，所以还是可以归结为中断。既然这样，RT-Thread 对临界段的保护就处理的很干脆了，直接把中断全部关了，NMIFAULT 和硬 FAULT 除外。

9.2 Cortex-M 内核快速关中断指令

为了快速地开关中断，Cortex-M 内核专门设置了一条 CPS 指令，有 4 种用法，具体见*代码清单 7-1*。

列表 1: 代码清单 7-1 CPS 指令用法

```
CPSID I ;PRIMASK=1 ; 关中断
CPSIE I ;PRIMASK=0 ; 开中断
CPSID F ;FAULTMASK=1 ; 关异常
CPSIE F ;FAULTMASK=0 ; 开异常
```

代码清单 7-1 中 PRIMASK 和 FAULTMAST 是 Cortex-M 内核里面三个中断屏蔽寄存器中的两个，还有一个是 BASEPRI，有关这三个寄存器的详细用法见表格 7-1。

表格 7-1 Cortex-M 内核中断屏蔽寄存器组描述

名字	功能描述
PRIMASK	这是个只有单一比特的寄存器。在它被置 1 后，就关掉所有可屏蔽的异常，只剩下 NMI 和硬 FAULT 可以响应。它的缺省值是 0，表示没有关中断。
FAULTMASK	这是个只有 1 个位的寄存器。当它置 1 时，只有 NMI 才能响应，所有其它的异常，甚至是硬 FAULT，也通通闭嘴。它的缺省值也是 0，表示没有关异常。
BASEPRI	这个寄存器最多有 9 位（由表达优先级的位数决定）。它定义了被屏蔽优先级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。但若被设成 0，则不关闭任何中断，0 也是缺省值。

9.3 关中断

RT-Thread 关中断的函数在 contex_rvds.s 中定义，在 rthw.h 中声明，具体实现见代码清单 7-2。

列表 2: 代码清单 7-2 关中断

```
;/\*
; \* rt_base_t rt_hw_interrupt_disable();
; \*/
rt_hw_interrupt_disable PROC;      (1)
EXPORT rt_hw_interrupt_disable;    (2)
MRS r0, PRIMASK;                   (3)
CPSID I;                           (4)
BX LR;                             (5)
ENDP;                              (6)
```

代码清单 7-2**(1)**：关键字 PROC 表示汇编子程序开始。

代码清单 7-2**(2)**：使用 EXPORT 关键字导出标号 rt_hw_interrupt_disable，使其具有全局属性，在外部头文件声明后（在 rthw.h 中声明），就可以在 C 文件中调用。

代码清单 7-2**(3)**：通过 MRS 指令将特殊寄存器 PRIMASK 寄存器的值存储到通用寄存器 r0。当在 C 中调用汇编的子程序返回时，会将 r0 作为函数的返回值。所以在 C 中调用 rt_hw_interrupt_disable()

的时候，需要事先声明一个变量用来存储 rt_hw_interrupt_disable() 的返回值，即 r0 寄存器的值，也就是 PRIMASK 的值。

代码清单 7-2**(4)**：关闭中断，即使用 CPS 指令将 PRIMASK 寄存器的值置 1。在这里，我敢肯定，一定会有人有这样一个疑问：关中断，不就是直接使用 CPSID I 指令就行了嘛，为什么还要第三步，即在执行 CPSIDI 指令前，要先把 PRIMASK 的值保存起来？这个疑问接下来在“临界段代码的应用”这个小结揭晓。

代码清单 7-2**(5)**：子程序返回。

代码清单 7-2**(6)**：ENDP 表示汇编子程序结束，与 PROC 成对使用。

9.4 开中断

RT-Thread 开中断的函数在 contex_rvds.s 中定义，在 rthw.h 中声明，具体实现见代码清单 7-3。

列表 3: 代码清单 7-3 开中断

```
;/\*
; \* void rt_hw_interrupt_enable(rt_base_t level);
; \*/
rt_hw_interrupt_enable PROC;      (1)
EXPORT rt_hw_interrupt_enable;    (2)
MSR PRIMASK, r0                   (3)
BX LR;                            (4)
ENDP;                             (5)
```

代码清单 7-3**(1)**：关键字 PROC 表示汇编子程序开始。

代码清单 7-3**(2)**：使用 EXPORT 关键字导出标号 rt_hw_interrupt_enable，使其具有全局熟悉，在外部头文件声明后（在 rthw.h 中声明），就可以在 C 文件中调用。

代码清单 7-2**(3)**：通过 MSR 指令将通用寄存器 r0 的值存储到特殊寄存器 PRIMASK。当在 C 中调用汇编的子程序返回时，会将第一个形参传入到通用寄存器 r0。所以在 C 中调用 rt_hw_interrupt_enable() 的时候，需要传入一个形参，该形参是进入临界段之前保存的 PRIMASK

的值。这个时候又有人会问，开中断，不就是使用 CPSIE I 指令就行了嘛，为啥跟我等凡人想的不一样？其中奥妙将在接下来“临界段代码的应用”这个小结揭晓

代码清单 7-2**(4)**：子程序返回。

代码清单 7-2**(5)**：ENDP 表示汇编子程序结束，与 PROC 成对使用。

9.5 临界段代码的应用

在进入临界段之前，我们会先把中断关闭，退出临界段时再把中断打开。而且 Cortex-M 内核设置了快速关中断的 CPS 指令，那么按照我们的第一思维，开关中断的函数的实现和临界段代码的保护应该是像代码清单 7-4_ 那样的。

列表 4: 代码清单 7-4 开关中断的函数的实现和临界段代码的保护

```
; 开关中断函数的实现
;/*
; * void rt_hw_interrupt_disable();
; */
rt_hw_interrupt_disable    PROC
    EXPORT  rt_hw_interrupt_disable
    CPSID   I                                   //(1)
    BX      LR
    ENDP

;/*
; * void rt_hw_interrupt_enable(void);
; */
rt_hw_interrupt_enable    PROC
    EXPORT  rt_hw_interrupt_enable
    CPSIE   I                                  // (2)
    BX      LR
```

(下页继续)

(续上页)

```
18      ENDP
19
20  PRIMASK = 0;        /* PRIMASK 初始值为 0, 表示没有关中断 */ (3)
21
22  /* 临界段代码保护 */
23  {
24      /* 临界段开始 */
25      rt_hw_interrupt_disable();      /* 关中断,PRIMASK = 1 */            (4)
26      {
27          /* 执行临界段代码, 不可中断 */                                  (5)
28      }
29      /* 临界段结束 */
30      rt_hw_interrupt_enable();       /* 开中断,PRIMASK = 0 */            (6)
31  }
```

代码清单 7-4**(1)**：关中断直接使用了 CPSID I，没有跟代码清单 7-2 一样事先将 PRIMASK 的值保存在 r0 中。

代码清单 7-4**(2)**：开中断直接使用了 CPSIE I，而不是像代码清单 7-3 那样从传进来的形参来恢复 PRIMASK 的值。

代码清单 7-4**(3)**：假设 PRIMASK 初始值为 0，表示没有关中断。

代码清单 7-4**(4)**：临界段开始，调用关中断函数 rt_hw_interrupt_disable()，此时 PRIMASK 的值等于 1，确实中断已经关闭。

代码清单 7-4**(5)**：执行临界段代码，不可中断。

代码清单 7-4**(6)**：临界段结束，调用开中断函数 rt_hw_interrupt_enable()，此时 PRIMASK 的值等于 0，确实中断已经开启。

乍一看，代码清单 7-4 的这种实现开关中断的方法确实有效，没有什么错误，但是我们忽略了一种情况，就是当临界段是出现嵌套的时候，这种开关中断的方法就不行了，具体怎么不行具体见代码清单 7-5。

列表 5: 代码清单 7-5 开关中断的函数的实现和嵌套临界段代码的保护（有错误，只为讲解）

```
; 开关中断函数的实现
;/*
; * void rt_hw_interrupt_disable();
; */
rt_hw_interrupt_disable    PROC
    EXPORT  rt_hw_interrupt_disable
    CPSID   I
    BX      LR
    ENDP

;/*
; * void rt_hw_interrupt_enable(void);
; */
rt_hw_interrupt_enable    PROC
    EXPORT  rt_hw_interrupt_enable
    CPSIE   I
    BX      LR
    ENDP

PRIMASK = 0;                            /* PRIMASK 初始值为 0, 表示没有关中断␣
↪*/
/* 临界段代码 */
{
    /* 临界段 1 开始 */
    rt_hw_interrupt_disable();          /* 关中断,PRIMASK = 1 */
    {
        /* 临界段 2 */
        rt_hw_interrupt_disable();      /* 关中断,PRIMASK = 1 */
        {
```

(下页继续)

(续上页)

```
        }
        rt_hw_interrupt_enable();          /* 开中断,PRIMASK = 0 */   //(注意)
    }
    /* 临界段 1 结束 */
    rt_hw_interrupt_enable();              /* 开中断,PRIMASK = 0 */
}
```

代码清单 7-5**(注意)**：当临界段出现嵌套的时候，这里以一重嵌套为例。临界段 1 开始和结束的时候 PRIMASK 分别等于 1 和 0，表示关闭中断和开启中断，这是没有问题的。临界段 2 开始的时候，PRIMASK 等于 1，表示关闭中断，这是没有问题的，问题出现在临界段 2 结束的时候，PRIMASK 的值等于 0，如果单纯对于临界段 2 来说，这也是没有问题的，因为临界段 2 已经结束，可是临界段 2 是嵌套在临界段 1 中虽然临界段 2 已经结束，但是临界段 1 还没有结束，中断是不能开启的，如果此时有外部中断来临，那么临界段 1 就会被中断，违背了我们的初衷，那应该怎么办？正确的做法具体见。

列表 6: 代码清单 7-6 开关中断的函数的实现和嵌套临界段代码的保护（正确）

```
;/*
; * rt_base_t rt_hw_interrupt_disable();
; */
rt_hw_interrupt_disable    PROC
    EXPORT  rt_hw_interrupt_disable
    MRS     r0, PRIMASK
    CPSID   I
    BX      LR
    ENDP

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

(下页继续)

(续上页)

```
rt_hw_interrupt_enable    PROC
    EXPORT  rt_hw_interrupt_enable
    MSR     PRIMASK, r0
    BX      LR
    ENDP

PRIMASK = 0;        /* PRIMASK 初始值为 0, 表示没有关中断 */                //(1)
rt_base_t level1;                                                           //(2)
rt_base_t level2;

/* 临界段代码 */
{
    /* 临界段 1 开始 */
    level1 = rt_hw_interrupt_disable();   /* 关中断,level1=0,PRIMASK=1 */   //(3)
    {
        /* 临界段 2 */
        level2 = rt_hw_interrupt_disable(); /* 关中断,level2=1,PRIMASK=1 */// (4)
        {

        }
        rt_hw_interrupt_enable(level2);     /* 开中断,level2=1,PRIMASK=1 */// (5)
    }
    /* 临界段 1 结束 */
    rt_hw_interrupt_enable(level1);      /* 开中断,level1=0,PRIMASK=0 */   //(6)
}
```

代码清单 7-6 **(1)**：假设 PRIMASK 初始值为 0, 表示没有关中断。

代码清单 7-6 **(2)**：定义两个变量，留着后面用。

代码清单 7-6 **(3)**：临界段 1 开始，调用关中断函数 rt_hw_interrupt_disable()，rt_hw_interrupt_disable() 函数先将 PRIMASK 的值存储在通用寄存器 r0，一开始我们假设 PRIMASK 的值等于 0，所以此时 r0 的值即为 0。然后执行汇编指令 CPSID I 关闭中断，即设置 PRIMASK 等于 1，在返回的时候 r0 当做函数的返回值存储在 level1，所以 level1 等于 r0 等于 0。

代码清单 7-6 **(4)**：临界段 2 开始，调用关中断函数 rt_hw_interrupt_disable()，rt_hw_interrupt_disable() 函数先将 PRIMASK 的值存储在通用寄存器 r0，临界段 1 开始的时候我们关闭了中断，即设置 PRIMASK 等于 1，所以此时 r0 的值等于 1。然后执行汇编指令 CPSIDI 关闭中断，即设置 PRIMASK 等于 1，在返回的时候 r0 当做函数的返回值存储在 level2，所以 level2 等于 r0 等于 1。

代码清单 7-6 **(5)**：临界段 2 结束，调用开中断函数 rt_hw_interrupt_enable(level2)，level2 作为函数的形参传入到通用寄存器 r0，然后执行汇编指令 MSR r0, PRIMASK 恢复 PRIMASK 的值。此时 PRIAMSK = r0 = level2 =1。关键点来了，为什么临界段 2 结束了，PRIMASK 还是等于 1，按道理应该是等于 0。因为此时临界段 2 是嵌套在临界段 1 中的，还是没有完全离开临界段的范畴，所以不能把中断打开，如果临界段是没有嵌套的，使用当前的开关中断的方法的话，那么 PRIMASK 确实是等于 1，具体举例见代码清单 7-7。

列表 7: 代码清单 7-7 开关中断的函数的实现和一重临界段代码的保护（正确）

```
;/*
; * rt_base_t rt_hw_interrupt_disable();
; */
rt_hw_interrupt_disable    PROC
    EXPORT  rt_hw_interrupt_disable
    MRS     r0, PRIMASK
    CPSID   I
    BX      LR
    ENDP

;/*
```

（下页继续）

(续上页)

```
; * void rt_hw_interrupt_enable(rt_base_t level);
; */
rt_hw_interrupt_enable    PROC
    EXPORT  rt_hw_interrupt_enable
    MSR     PRIMASK, r0
    BX      LR
    ENDP
PRIMASK = 0;                          /* PRIMASK 初始值为 0, 表示没有关中断 */
rt_base_t level1;

/* 临界段代码 */
{
/* 临界段开始 */
level1 = rt_hw_interrupt_disable();/* 关中断,level1=0,PRIMASK=1 */
{

}
/* 临界段结束 */
rt_hw_interrupt_enable(level1);     /* 开中断,level1=0,PRIMASK=0 *///(注意点)
}
```

代码清单 7-6 **(6)**：临界段 1 结束，PRIMASK 等于 0，开启中断，与进入临界段 1 遥相呼应。

9.6 实验现象

本章没有实验，充分理解本章内容即可，这么简单，其实也没啥好理解的。

第 10 章　对象容器的实现

10.1 什么是对象

在 RT-Thread 中，所有的数据结构都称之为对象。

10.1.1 对象枚举定义

其中线程，信号量，互斥量、事件、邮箱、消息队列、内存堆、内存池、设备和定时器在 rtdef.h 中有明显的枚举定义，即为每个对象打上了一个数字标签，具体见*代码清单 8-1*。

列表 1: 代码清单 8-1 对象类型枚举定义

```
enum rt_object_class_type
{
    RT_Object_Class_Thread = 0,         /* 对象是线程 */
    RT_Object_Class_Semaphore,          /* 对象是信号量 */
    RT_Object_Class_Mutex,              /* 对象是互斥量 */
    RT_Object_Class_Event,              /* 对象是事件 */
    RT_Object_Class_MailBox,            /* 对象是邮箱 */
    RT_Object_Class_MessageQueue,       /* 对象是消息队列 */
    RT_Object_Class_MemHeap,            /* 对象是内存堆 */
    RT_Object_Class_MemPool,            /* 对象是内存池 */
    RT_Object_Class_Device,             /* 对象是设备 */
    RT_Object_Class_Timer,              /* 对象是定时器 */
    RT_Object_Class_Module,             /* 对象是模块 */
    RT_Object_Class_Unknown,            /* 对象未知 */
    RT_Object_Class_Static = 0x80       /* 对象是静态对象 */
};
```

10.1.2 对象数据类型定义

在 rtt 中，为了方便管理这些对象，专门定义了一个对象类型数据结构，具体见代码清单 8-2。

列表 2: 代码清单 8-2 对象数据类型定义

```
struct rt_object
{
    char name[RT_NAME_MAX];    (1) /* 内核对象的名字 */
    rt_uint8_t type;           (2) /* 内核对象的类型 */
    rt_uint8_t flag;           (3) /* 内核对象的状态 */
    rt_list_t list;            (4) /* 内核对象的列表节点 */
};
typedef struct rt_object *rt_object_t; (5) /* 内核对象数据类型重定义 */
```

代码清单 8-2**(1)**：对象名字，字符串形式，方便调试，最大长度由 rt_config.h 中的宏 RT_NAMA_MAX 决定，默认定义为 8。

代码清单 8-2**(2)**：对象的类型，RT-Thread 为每一个对象都打上了数字标签，取值由 rt_object_class_type 枚举类型限定，具体见代码清单 8-1。

代码清单 8-2**(3)**：对象的状态。

代码清单 8-2**(4)**：对象的列表节点，每个对象都可以通过自己的列表节点 list 将自己挂到容器列表中，什么是容器接下来会讲解到。

代码清单 8-2**(5)**：对象数据类型，RT-Thread 中会为每一个新的结构体用 typedef 重定义一个指针类型的数据结构。

10.1.3 在线程控制块中添加对象成员

在 RT-Thread 中，每个对象都会有对应的一个结构体，这个结构体叫做该对象的控制块。如线程会有一个线程控制块，定时器会有一个定时器控制块，信号量会有信号量控制块等。这些控制块的开头都会包含一个内核对象结构体，或者直接将对象结构体的成员放在对象控制块结构体的开头。其中线程控制块的开头放置的就是对象结构体的成员，具体见代码清单 8-3 开头的高亮部

分代码。这里我们只讲解往线程控制块里面添加对象结构体成员，其它内核对象的都是直接在其开头使用 struct rt_object 直接定义一个内核对象变量。

列表 3: 代码清单 8-3 在线程控制块中添加对象成员

```
struct rt_thread
{
    /* rt 对象 */
    char        name[RT_NAME_MAX];     /* 对象的名字 */
    rt_uint8_t  type;                  /* 对象类型 */
    rt_uint8_t  flags;                 /* 对象的状态 */
    rt_list_t   list;                  /* 对象的列表节点 */
    rt_list_t   tlist;                 /* 线程链表节点 */
    void        *sp;                       /* 线程栈指针 */
    void        *entry;                    /* 线程入口地址 */
    void        *parameter;            /* 线程形参 */
    void        *stack_addr;           /* 线程起始地址 */
    rt_uint32_t stack_size;            /* 线程栈大小，单位为字节 */
};
```

10.2 什么是容器

在 rtt 中，每当用户创建一个对象，如线程，就会将这个对象放到一个叫做容器的地方，这样做的目的是为了方便管理，这时用户会问，管理什么？在 RT- Thread 的组件 finsh 的使用中，就需要使用到容器，通过扫描容器的内核对象来获取各个内核对象的状态，然后输出调试信息。目前，我们只需要知道所有创建的对象都会被放到容器中即可。

那什么是容器，从代码上看，容器就是一个数组，是一个全局变量，数据类型为 struct rt_object_information，在 object.c 中定义，具体见*代码清单 8-4*，示意图具体见图 *8-1*。

10.2.1 容器的定义

列表 4: 代码清单 8-4 rtt 容器的定义

```
static struct rt_object_information                          (1)
rt_object_container[RT_Object_Info_Unknown] =                (2)
{
    /* 初始化对象容器 - 线程 */                              (3)-1
    {
        RT_Object_Class_Thread,                              (3)-2
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Info_Thread),
            (1)
        sizeof(struct rt_thread)                             (3)-3
    },

#ifdef RT_USING_SEMAPHORE                                    (4)
    /* 初始化对象容器 - 信号量 */
    {
        RT_Object_Class_Semaphore,
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Info_Semaphore),
        sizeof(struct rt_semaphore)
    },
#endif

#ifdef RT_USING_MUTEX                                        (5)
    /* 初始化对象容器 - 互斥量 */
    {
        RT_Object_Class_Mutex,
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Info_Mutex),
        sizeof(struct rt_mutex)
    },
#endif

```

(下页继续)

(续上页)

```
#ifdef RT_USING_EVENT                                    (6)
    /* 初始化对象容器 - 事件 */
    {
        RT_Object_Class_Event,
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Info_Event),
        sizeof(struct rt_event)
    },
#endif

#ifdef RT_USING_MAILBOX                                  (7)
    /* 初始化对象容器 - 邮箱 */
    {
        RT_Object_Class_MailBox,
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Info_MailBox),
        sizeof(struct rt_mailbox)
    },
#endif

#ifdef RT_USING_MESSAGEQUEUE                             (8)
    /* 初始化对象容器 - 消息队列 */
    {
        RT_Object_Class_MessageQueue,
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Info_MessageQueue),
        sizeof(struct rt_messagequeue)
    },
#endif

#ifdef RT_USING_MEMHEAP                                  (9)
    /* 初始化对象容器 - 内存堆 */
    {
        RT_Object_Class_MemHeap,
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Info_MemHeap),
```

(下页继续)

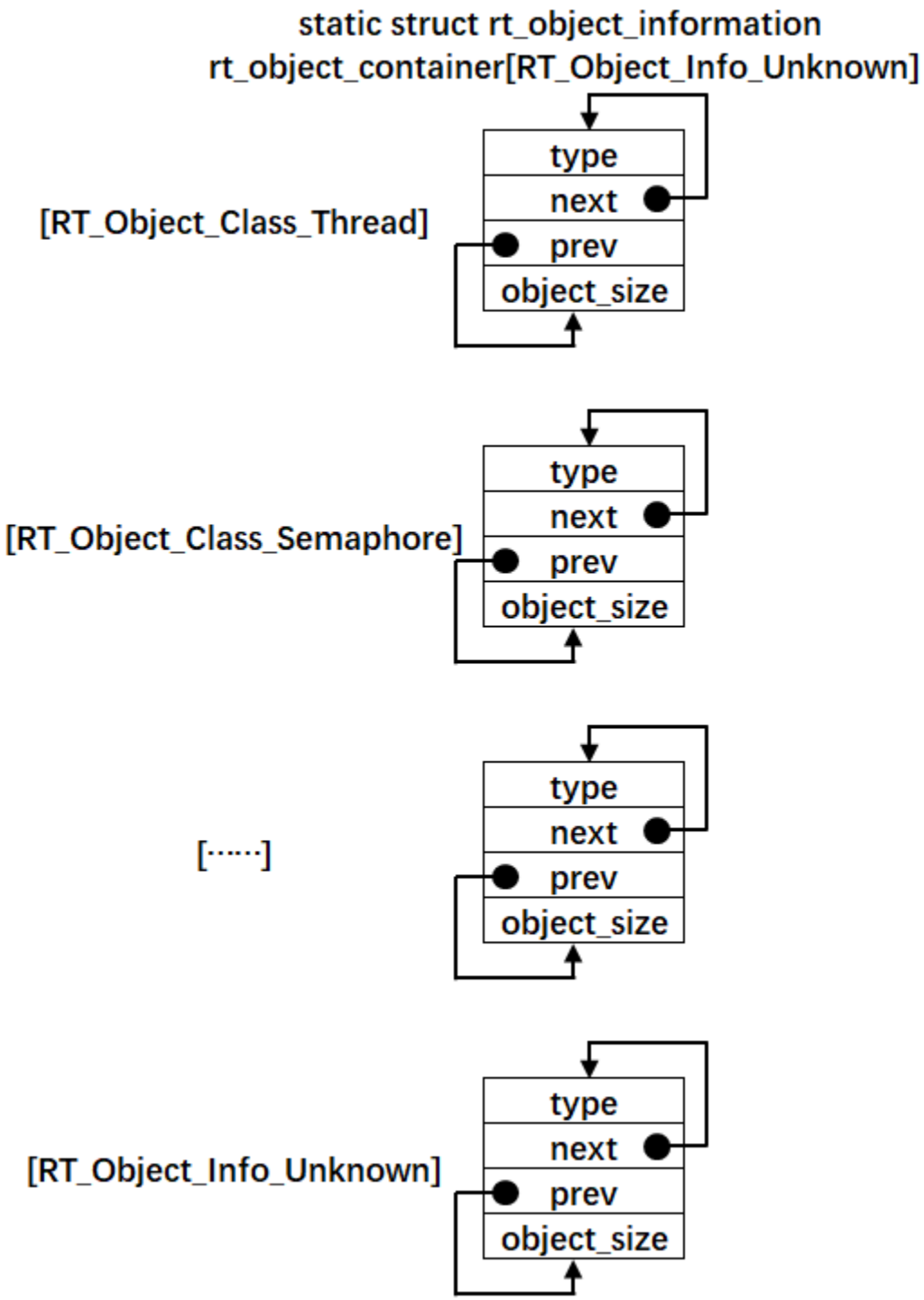
(续上页)

```
        sizeof(struct rt_memheap)
    },
#endif

#ifdef RT_USING_MEMPOOL                                    (10)
    /* 初始化对象容器 - 内存池 */
    {
        RT_Object_Class_MemPool,
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Info_MemPool),
        sizeof(struct rt_mempool)
    },
#endif

#ifdef RT_USING_DEVICE                                     (11)
    /* 初始化对象容器 - 设备 */
    {
        RT_Object_Class_Device,
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Info_Device), sizeof(struct rt_
↪device)},
#endif
    /* 初始化对象容器 - 定时器 */                          (12)
    /*
    {
        RT_Object_Class_Timer,
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Info_Timer),
        sizeof(struct rt_timer)
    },
    */
#ifdef RT_USING_MODULE                                     (13)
    /* 初始化对象容器 - 模块 */
    {
        RT_Object_Class_Module,
```

(下页继续)

(续上页)

```
        _OBJ_CONTAINER_LIST_INIT(RT_Object_Info_Module),
        sizeof(struct rt_module)
    },
#endif
```



代码清单 8-4 **(1)**：容器是一个全部变量的数组，数据类型为 struct rt_object_information，这是一个

结构体类型，包含对象的三个信息，分别为对象类型、对象列表节点头和对象的大小，在 rtdef.h 中定义，具体实现见代码清单 8-5。

列表 5: 代码清单 8-5 内核对象信息结构体定义

```
struct rt_object_information
{
    enum rt_object_class_type type;//   (1) /* 对象类型 */
    rt_list_t object_list;//            (2) /* 对象列表节点头 */
    rt_size_t object_size;//            (3) /* 对象大小 */
};
```

代码清单 8-5 **(1)**：对象的类型，取值只能是 rt_object_class_type 枚举类型，具体取值见代码清单 8-1。

代码清单 8-5 **(2)**：对象列表节点头，每当对象创建时，对象就会通过他们控制块里面的 list 节点将自己挂到对象容器中的对应列表，同一类型的对象是挂到对象容器中同一个对象列表的，容器数组的小标对应的就是对象的类型。

代码清单 8-5**(3)**：对象的大小，可直接通过 sizeof(对象控制块类型) 获取。

代码清单 8-4 **(2)**：容器的大小由 RT_Object_Info_Unknown 决定，RT_Object_Info_Unknown 是一个枚举类型的变量，在 rt_object_info_type 这个枚举结构体里面定义，具体见代码清单 8-6。

列表 6: 代码清单 8-6 对象容器数组的下标定义

```
/*
* 对象容器数组的下标定义，决定容器的大小
*/
enum rt_object_info_type
{
    RT_Object_Info_Thread = 0,              /* 对象是线程 */
#ifdef RT_USING_SEMAPHORE
    RT_Object_Info_Semaphore,               /* 对象是信号量 */
#endif
```

(下页继续)

(续上页)

```
#ifdef RT_USING_MUTEX
    RT_Object_Info_Mutex,                    /* 对象是互斥量 */
#endif
#ifdef RT_USING_EVENT
    RT_Object_Info_Event,                    /* 对象是事件 */
#endif
#ifdef RT_USING_MAILBOX
    RT_Object_Info_MailBox,                  /* 对象是邮箱 */
#endif
#ifdef RT_USING_MESSAGEQUEUE
    RT_Object_Info_MessageQueue,             /* 对象是消息队列 */
#endif
#ifdef RT_USING_MEMHEAP
    RT_Object_Info_MemHeap,                  /* 对象是内存堆 */
#endif
#ifdef RT_USING_MEMPOOL
    RT_Object_Info_MemPool,                  /* 对象是内存池 */
#endif
#ifdef RT_USING_DEVICE
    RT_Object_Info_Device,                   /* 对象是设备 */
#endif
    RT_Object_Info_Timer,                    /* 对象是定时器 */
#ifdef RT_USING_MODULE
    RT_Object_Info_Module,                   /* 对象是模块 */
#endif
    RT_Object_Info_Unknown,                  /* 对象未知 */
};
```

从代码清单 8-6 可以看出 RT_Object_Info_Unknown 位于枚举结构体的最后，它的具体取值由前面的成员多少决定，前面的成员是否有效都是通过宏定义来决定的，只有当在 rtconfig.h 中定义了相应的宏，对应的枚举成员才会有效，默认在这些宏都没有定义的情况下只有 RT_Object_Info_Thread 和 RT_Object_Info_Timer 有效，此时 RT_Object_Info_Unknown 的值等于

2。当这些宏全部有效，RT_Object_Info_Unknown 的值等于 11，即容器的大小为 12，此时是最大。C 语言知识：如果枚举类型的成员值没有具体指定，那么后一个值是在前一个成员值的基础上加 1。

代码清单 *8-4* **(3)**：初始化对象容器—线程，线程是 rtt 里面最基本的对象，是必须存在的，跟其它的对象不一样，没有通过宏定义来选择，接下来下面的信号量、邮箱都通过对应的宏定义来控制是否初始化，即只有在创建了相应的对象后，才在对象容器里面初始化。

代码清单 *8-4* **(3)-1**：初始化对象类型为线程。

代码清单 *8-4* **(3)-2**：初始化对象列表节点头里面的 next 和 prev 两个节点指针分别指向自身，具体见图 *8-1*。_OBJ_CONTAINER_LIST_INIT() 是一个带参宏，用于初始化一个节点 list，在 object.c 中定义，具体见代码清单 *8-7*。

列表 7: 代码清单 8-7 _OBJ_CONTAINER_LIST_INIT() 宏定义

```
#define _OBJ_CONTAINER_LIST_INIT(c) \
{&(rt_object_container[c].object_list), &(rt_object_container[c].object_
↪list)}
```

代码清单 *8-4* **(3)-3**：获取线程对象的大小，即整个线程控制块的大小。

代码清单 *8-4* **(4)**：初始化对象容器—信号量，由宏 RT_USING_SEMAPHORE 决定。

代码清单 *8-4* **(5)**：初始化对象容器—互斥量，由宏 RT_USING_MUTEX 决定。

代码清单 *8-4* **(6)**：初始化对象容器—事件，由宏 RT_USING_EVENT 决定。

代码清单 *8-4* **(7)**：初始化对象容器—邮箱，由宏 RT_USING_MAILBOX 决定。

代码清单 *8-4* **(8)**：初始化对象容器—消息队列，由宏 RT_USING_MESSAGEQUEUE 决定。

代码清单 *8-4* **(9)**：初始化对象容器—内存堆，由宏 RT_USING_MEMHEAP 决定。

代码清单 *8-4* **(10)**：初始化对象容器—内存池，由宏 RT_USING_MEMPOOL 决定。

代码清单 *8-4* **(11)**：初始化对象容器—设备，由宏 RT_USING_DEVICE 决定。

代码清单 *8-4* **(12)**：初始化对象容器—定时器，每个线程在创建的时候都会自带一个定时器，但

是目前我们还没有在线程中加入定时器，所以这部分初始化我们先注释掉，等加入定时器的时候再释放。

代码清单 8-4 **(13)**：初始化对象容器—模块，由宏 RT_USING_MODULE 决定。

10.3 容器的接口实现

容器接口相关的函数均在 object.c 中实现。

10.3.1 获取指定类型的对象信息

从容器中获取指定类型的对象的信息由函数 rt_object_get_information() 实现，具体定义见代码清单 8-8。

列表 8: 代码清单 8-8 rt_object_get_information() 函数定义

```
struct rt_object_information *rt_object_get_information(enum rt_object_class_type type)
{
    int index;

    for (index = 0; index < RT_Object_Info_Unknown; index ++)
        if (rt_object_container[index].type == type) return &rt_object_container[index];

    return RT_NULL;
}
```

我们知道，容器在定义的时候，大小是被固定的，由 RT_Object_Info_Unknown 这个枚举值决定，但容器里面的成员是否初始化就不一定了，其中线程和定时器这两个对象默认会被初始化，剩下的其它对象由对应的宏决定。rt_object_get_information() 会遍历整个容器对象，如果对象的类型等于我们指定的类型，那么就返回该容器成员的地址，地址的类型为 struct rt_object_information。

10.3.2 对象初始化

每创建一个对象，都需要先将其初始化，主要分成两个部分的工作，首先将对象控制块里面与对象相关的成员初始化，然后将该对象插入到对象容器中，具体的代码实现见代码清单 8-9。

列表 9: 代码清单 8-9 对象初始化 rt_object_init() 函数定义

```
/**
* 该函数将初始化对象并将对象添加到对象容器中
*
* @param object 要初始化的对象
* @param type 对象的类型
* @param name 对象的名字，在整个系统中，对象的名字必须是唯一的
*/

void rt_object_init(struct rt_object         *object,
                    enum rt_object_class_type type,
                    const char               *name)
{
    register rt_base_t temp;
    struct rt_object_information *information;

    /* 获取对象信息，即从容器里拿到对应对象列表头指针 */
    information = rt_object_get_information(type);

    /* 设置对象类型为静态 */
    object->type = type | RT_Object_Class_Static;

    /* 拷贝名字 */
    rt_strncpy(object->name, name, RT_NAME_MAX);

    /* 关中断 */
```

(下页继续)

(续上页)

```
    temp = rt_hw_interrupt_disable();

    /* 将对象插入到容器的对应列表中，不同类型的对象所在的列表不一样 */
    rt_list_insert_after(&(information->object_list), &(object->list));

    /* 使能中断 */
    rt_hw_interrupt_enable(temp);
}
```

代码清单 8-9**(1)**：要初始化的对象。我们知道每个对象的控制块开头的成员都是对象信息相关的成员，比如一个线程控制块，它的开头前面 4 个成员都是与对象信息相关的，在调用 rt_object_init() 函数的时候，只需将线程控制块强制类型转化为 structrt_object 作为第一个形参即可。

代码清单 8-9**(2)**：对象的类型，是一个数字化的枚举值，具体见代码清单 8-1。

代码清单 8-9**(3)**：对象的名字，字符串形式，在整个系统中，对象的名字必须是唯一的。

代码清单 8-9**(4)**：获取对象信息，即从容器里拿到对应对象列表头指针。容器是一个定义好的全局数组，可以直接操作。

代码清单 8-9**(5)**：设置对象类型为静态。

代码清单 8-9**(6)**：拷贝名字。rt_strncpy() 是字符串拷贝函数，在 kservice.c（kservice.c 第一次使用需要在 rtthread3.0.3src 下新建，然后添加到工程 rtt/source 组中）中定义，在 rtthread.h 声明，具体代码实现见代码清单 8-10。

列表 10: 代码清单 8-10 rt_strncpy() 函数定义

```
/**
* 该函数将指定个数的字符串从一个地方拷贝到另外一个地方
*
* @param dst 字符串拷贝的目的地
* @param src 字符串从哪里拷贝
* @param n 要拷贝的最大长度
```

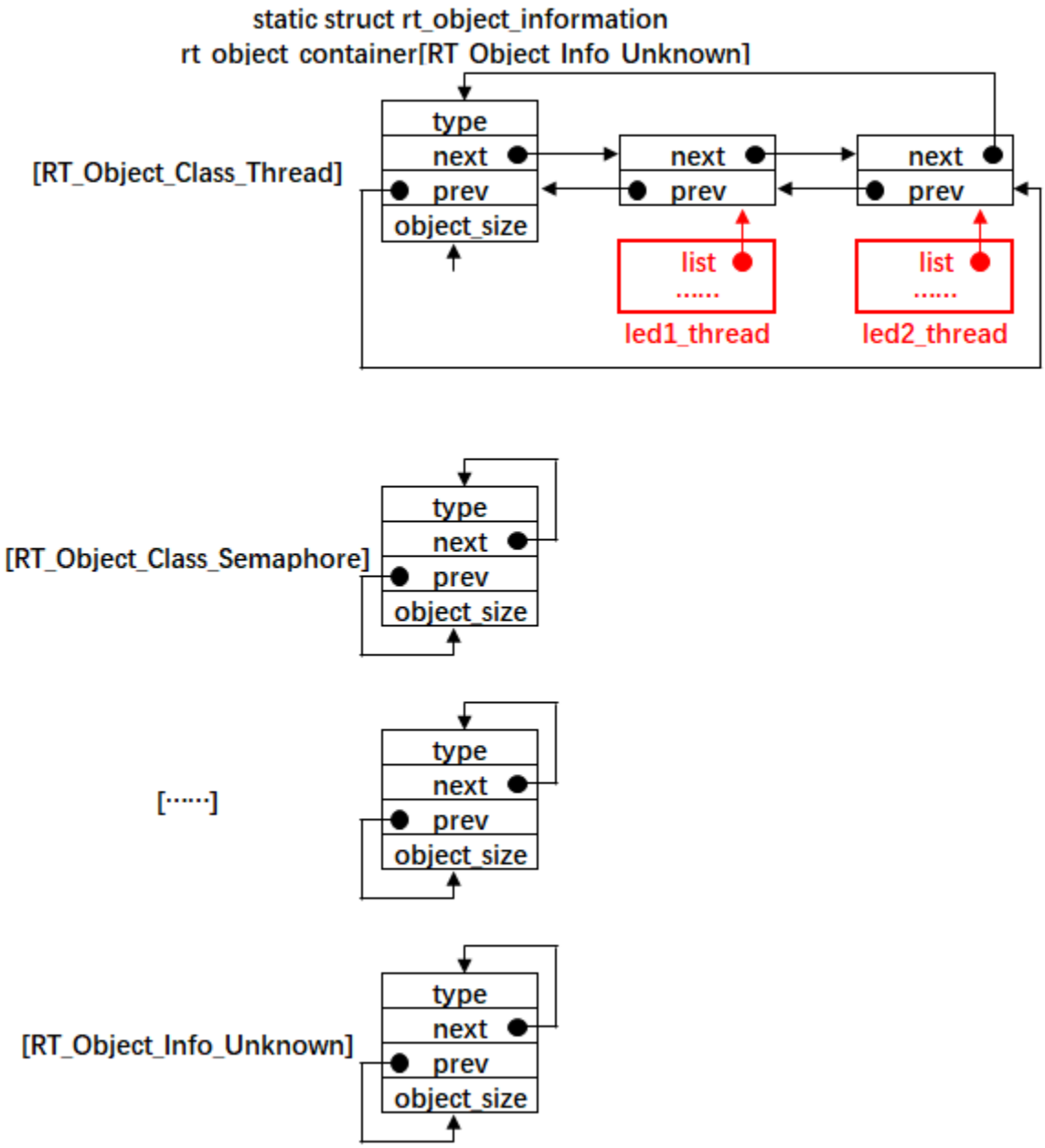
(下页继续)

(续上页)

```
 *
 * @return the result
 */
char *rt_strncpy(char *dst, const char *src, rt_ubase_t n)
{
    if (n != 0)
    {
        char *d = dst;
        const char *s = src;

        do
        {
            if ((*d++ = *s++) == 0)
            {
                /* NUL pad the remaining n-1 bytes */
                while (--n != 0)
                    *d++ = 0;
                break;
            }
        } while (--n != 0);
    }

    return (dst);
}
```

代码清单 8-9**(7)**：关中断，接下来链表的操作不希望被中断。

代码清单 8-9**(8)**：将对象插入到容器的对应列表中，不同类型的对象所在的列表不一样。比如创建了两个线程，他们在容器列表中的示意图具体见。



代码清单 8-9**(9)**：使能中断。

10.3.3 调用对象初始化函数

对象初始化函数在线程初始化函数里面被调用，具体见*代码清单 8-11* 的高亮部分。如果创建了两个线程，在线程初始化之后，线程通过自身的 list 节点将自身挂到容器的对象列表中，在容器中的示意图具体见图 *8-2*。

列表 11: 代码清单 8-11 在线程初始化中添加对象初始化功能

```
rt_err_t rt_thread_init(struct rt_thread *thread,
                        const char       *name,
                        void (*entry)(void *parameter),
                        void             *parameter,
                        void             *stack_start,
                        rt_uint32_t       stack_size)
{
    /* 线程对象初始化 */
    /* 线程结构体开头部分的成员就是 rt_object_t 类型 */
    rt_object_init((rt_object_t)thread, RT_Object_Class_Thread, name);
    rt_list_init(&(thread->tlist));

    thread->entry = (void *)entry;
    thread->parameter = parameter;

    thread->stack_addr = stack_start;
    thread->stack_size = stack_size;

    /* 初始化线程栈，并返回线程栈指针 */
    thread->sp = (void *)rt_hw_stack_init( thread->entry,
                                           thread->parameter,
                                           (void *)((char *)thread->stack_addr␣
↪+ thread->stack_size - 4) );

```

(下页继续)

(续上页)

```
    return RT_EOK;
}
```

10.4 实验现象

本章没有实验，充分理解本章内容即可。

第 11 章　空闲线程与阻塞延时的实现

在上一章节中，线程体内的延时使用的是软件延时，即还是让 CPU 空等来达到延时的效果。使用 RTOS 的很大优势就是榨干 CPU 的性能，永远不能让它闲着，线程如果需要延时也就不能再让 CPU 空等来实现延时的效果。RTOS 中的延时叫阻塞延时，即线程需要延时的时候，线程会放弃 CPU 的使用权，CPU 可以去干其它的事情，当线程延时时间到，重新获取 CPU 使用权，线程继续运行，这样就充分地利用了 CPU 的资源，而不是干等着。

当线程需要延时，进入阻塞状态，那 CPU 又去干什么事情了？如果没有其它线程可以运行，RTOS 都会为 CPU 创建一个空闲线程，这个时候 CPU 就运行空闲线程。在 RT-Thread 中，空闲线程是系统在初始化的时候创建的优先级最低的线程，空闲线程主体主要是做一些系统内存的清理工作。但是为了简单起见，我们本章实现的空闲线程只是对一个全局变量进行计数。鉴于空闲线程的这种特性，在实际应用中，当系统进入空闲线程的时候，可在空闲线程中让单片机进入休眠或者低功耗等操作。

11.1 实现空闲线程

11.1.1 定义空闲线程的栈

空闲线程的栈在 idle.c（idle.c 第一次使用需要自行在文件夹 rtthread3.0.3src 中新建并添加到工程的 rtt/source 组）文件中定义，具体见*代码清单 9-1*。

列表 1: 代码清单 9-1 定义空闲线程的栈

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

#define IDLE_THREAD_STACK_SIZE      512

```

（下页继续）

(续上页)

```
ALIGN(RT_ALIGN_SIZE)
static rt_uint8_t rt_thread_stack[IDLE_THREAD_STACK_SIZE];
```

代码清单 9-1 (1)：空闲线程的栈是一个定义好的数组，大小由 IDLE_THREAD_STACK_SIZE 这个宏控制，默认为 512，即 128 个字。

11.1.2 定义空闲线程的线程控制块

线程控制块是每一个线程必须的，空闲线程的的线程控制块在 idle.c 中定义，是一个全局变量，具体见代码清单 9-2。

列表 2: 代码清单 9-2 定义空闲线程的线程控制块

```
/* 空闲线程的线程控制块 */
struct rt_thread idle;
```

11.1.3 定义空闲线程函数

在 RT-Thread 中空闲线程函数主要是做一些系统内存的清理工作，但是为了简单起见，我们本章实现的空闲线程只是对一个全局变量 rt_idletask_ctr 进行计数，rt_idletask_ctr 在 idle.c 中定义，默认初始值为 0。空闲线程函数在 idle.c 定义，具体实现见代码清单 9-3。

列表 3: 代码清单 9-3 空闲线程函数

```
rt_ubase_t  rt_idletask_ctr = 0;

void rt_thread_idle_entry(void *parameter)
{
    parameter = parameter;
    while (1)
    {
```

(下页继续)

(续上页)

```
        rt_idletask_ctr ++;
    }
}
```

11.1.4 空闲线程初始化

当定义好空闲线程的栈，线程控制块和函数主体之后，我们需要空闲线程初始化函数将这三者联系在一起，这样空闲线程才能够被系统调度，空闲线程初始化函数在 idle.c 定义，具体实现见*代码清单 9-4*。

列表 4: 代码清单 9-4 空闲线程初始化

```
void rt_thread_idle_init(void)
{

    /* 初始化线程 */                                    (1)
    rt_thread_init(&idle,
                   "idle",
                   rt_thread_idle_entry,
                   RT_NULL,
                   &rt_thread_stack[0],
                   sizeof(rt_thread_stack));

    /* 将线程插入到就绪列表 */                          (2)
    rt_list_insert_before( &(rt_thread_priority_table[RT_THREAD_PRIORITY_
↪MAX-1]),&(idle.tlist) );
}
```

代码清单 9-4(1)：创建空闲线程。

代码清单 9-4(2)：将空闲线程插入到就绪列表的末尾。在下一章我们会支持优先级，空闲线程默认的优先级是最低的，即排在就绪列表的最后面。

11.2 实现阻塞延时

阻塞延时的阻塞是指线程调用该延时函数后，线程会被剥离 CPU 使用权，然后进入阻塞状态，直到延时结束，线程重新获取 CPU 使用权才可以继续运行。在线程阻塞的这段时间，CPU 可以去执行其它的线程，如果其它的线程也在延时状态，那么 CPU 就将运行空闲线程。阻塞延时函数在 thread.c 中定义，具体代码实现见*代码清单 9-5*。

列表 5: 代码清单 9-5 阻塞延时代码

```
void rt_thread_delay(rt_tick_t tick)
{
    struct rt_thread *thread;

    /* 获取当前线程的线程控制块 */
    thread = rt_current_thread;                    (1)

    /* 设置延时时间 */
    thread->remaining_tick = tick;                 (2)

    /* 进行系统调度 */
    rt_schedule();                                 (3)
}
```

代码清单 9-5 **(1)**：获取当前线程的线程控制块。rt_current_thread 是一个在 scheduler.c 定义的全局变量，用于指向当前正在运行的线程的线程控制块。

代码清单 9-5 **(2)**：remaining_tick 是线程控制块的一个成员，用于记录线程需要延时的时间，单位为 SysTick 的中断周期。比如我们本书当中 SysTick 的中断周期为 10ms，调用 rt_thread_delay(2) 则完成 2*10ms 的延时。线程的定义具体见*代码清单 9-6*。

列表 6: 代码清单 9-6 remaining_tick 定义

```
struct rt_thread
{
    /* rt 对象 */
    char        name[RT_NAME_MAX];      /* 对象的名字 */
    rt_uint8_t  type;                   /* 对象类型 */
    rt_uint8_t  flags;                  /* 对象的状态 */
    rt_list_t   list;                   /* 对象的列表节点 */
    rt_list_t   tlist;                  /* 线程链表节点 */
    void        *sp;                        /* 线程栈指针 */
    void        *entry;                     /* 线程入口地址 */
    void        *parameter;             /* 线程形参 */
    void        *stack_addr;            /* 线程起始地址 */
    rt_uint32_t stack_size;             /* 线程栈大小，单位为字节 */

    rt_ubase_t  remaining_tick;         /* 用于实现阻塞延时 */
};
```

代码清单 9-5 **(3)**：系统调度。这个时候的系统调度与上一章节的不一样，具体见代码清单 9-7，其中高亮部分为上一章节的代码，现已用条件编译屏蔽掉。

列表 7: 代码清单 9-7 系统调度

```
extern struct rt_thread idle;
extern struct rt_thread rt_flag1_thread;
extern struct rt_thread rt_flag2_thread;

/* 系统调度 */
void rt_schedule(void)
{
    struct rt_thread *to_thread;
    struct rt_thread *from_thread;
```

（下页继续）

(续上页)

```
#if 0
    /* 两个线程轮流切换 */
    if( rt_current_thread == rt_list_entry( rt_thread_priority_table[0].next,
                                            struct rt_thread,
                                            tlist) )
    {
        from_thread = rt_current_thread;
        to_thread = rt_list_entry( rt_thread_priority_table[1].next,
                                   struct rt_thread,
                                   tlist);
    rt_current_thread = to_thread;
    }
    else
    {
        from_thread = rt_current_thread;
        to_thread = rt_list_entry( rt_thread_priority_table[0].next,
                                   struct rt_thread,
                                   tlist);
    rt_current_thread = to_thread;
    }
#else

    /* 如果当前线程是空闲线程，那么就去尝试执行线程 1 或者线程 2，
    看看他们的延时时间是否结束，如果线程的延时时间均没有到期，
    那就返回继续执行空闲线程 */
    if( rt_current_thread == &idle )                                    (1)
    {
        if(rt_flag1_thread.remaining_tick == 0)
        {
            from_thread = rt_current_thread;
```

(下页继续)

(续上页)

```
            to_thread = &rt_flag1_thread;
            rt_current_thread = to_thread;
        }
        else if(rt_flag2_thread.remaining_tick == 0)
        {
            from_thread = rt_current_thread;
            to_thread = &rt_flag2_thread;
            rt_current_thread = to_thread;
        }
        else
        {
            return;          /* 线程延时均没有到期则返回，继续执行空闲线程 */
        }
    }
    else /* 当前线程不是空闲线程则会执行到这里 */                    (2)
    {
        /* 如果当前线程是线程 1 或者线程 2 的话，检查下另外一个线程，如果另外的线程不在延时中，就切换到该线程
        否则，判断下当前线程是否应该进入延时状态，如果是的话，就切换到空闲线程。否则就不进行任何切换 */
        if(rt_current_thread == &rt_flag1_thread)
        {
            if(rt_flag2_thread.remaining_tick == 0)
            {
                from_thread = rt_current_thread;
                to_thread = &rt_flag2_thread;
                rt_current_thread = to_thread;
            }
            else if(rt_current_thread->remaining_tick != 0)
            {
                from_thread = rt_current_thread;
                to_thread = &idle;
```

(下页继续)

(续上页)

```
                rt_current_thread = to_thread;
            }
            else
            {
                return;             /* 返回，不进行切换，因为两个线程都处于延时中 */
            }
        }
        else if(rt_current_thread == &rt_flag2_thread)
        {
            if(rt_flag1_thread.remaining_tick == 0)
            {
                from_thread = rt_current_thread;
                to_thread = &rt_flag1_thread;
                rt_current_thread = to_thread;
            }
            else if(rt_current_thread->remaining_tick != 0)
            {
                from_thread = rt_current_thread;
                to_thread = &idle;
                rt_current_thread = to_thread;
            }
            else
            {
                return;             /* 返回，不进行切换，因为两个线程都处于延时中 */
            }
        }
    }
#endif
    /* 产生上下文切换 */
    rt_hw_context_switch((rt_uint32_t)&from_thread->sp,(rt_uint32_t)&to_thread->sp);
```

(下页继续)

(续上页)

```
}
```

代码清单 9-7**(1)**：如果当前线程是空闲线程，那么就去尝试执行线程 1 或者线程 2，看看他们的延时时间是否结束，如果线程的延时时间均没有到期，那就返回继续执行空闲线程。

代码清单 9-7**(2)**：如果当前线程是线程 1 或者线程 2 的话，检查下另外一个线程，如果另外的线程不在延时中，就切换到该线程。否则，判断下当前线程是否应该进入延时状态，如果是的话，就切换到空闲线程，否则就不进行任何切换。

代码清单 9-7**(3)**：系统调度，实现线程的切换。

11.3 SysTick_Handler 中断服务函数

在系统调度函数 rt_schedule() 中，会判断每个线程的线程控制块中的延时成员 remaining_tick 的值是否为 0，如果为 0 就要将对应的线程就绪，如果不为 0 就继续延时。如果一个线程要延时，一开始 remaining_tick 肯定不为 0，当 remaining_tick 变为 0 的时候表示延时结束，那么 remaining_tick 是以什么周期在递减？在哪里递减？在 RT-Thread 中，这个周期由 SysTick 中断提供，操作系统里面的最小的时间单位就是 SysTick 的中断周期，我们称之为一个 tick，SysTick 中断服务函数我们放在 main.c 中实现，具体见代码清单 9-8。

列表 8: 代码清单 9-8 SysTick_Handler 中断服务函数

```
/* 关中断 */
rt_hw_interrupt_disable();                                  (1)

/* SysTick 中断频率设置 */
SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );     (2)

void SysTick_Handler(void)                                  (3)
{
    /* 进入中断 */
    rt_interrupt_enter();                                   (3)-1
```

(下页继续)

(续上页)

```
    /* 时基更新 */
    rt_tick_increase();                                    (3)-2
    /* 离开中断 */
    rt_interrupt_leave();                                  (3)-3
}
```

代码清单 9-8**(1)**：关中断。在程序开始的时候把中断关闭是一个好习惯，等系统初始化完毕，线程创建完毕，启动系统调度的时候会重新打开中断。如果一开始不关闭中断的话，接下来 SysTick 初始化完成，然后再初始化系统和创建线程，如果系统初始化和线程创建的时间大于 SysTick 的中断周期的话，那么就会出现系统或者线程都还没有准备好的情况下就先执行了 SysTick 中断服务函数，进行了系统调度，显然，这是不科学的。

代码清单 9-8**(2)**：初始化 SysTick，调用固件库函数 SysTick_Config 来实现，配置中断周期为 10ms，中断优先级为最低（无论中断优先级分组怎么分都是最低，因为这里把表示 SysTick 中断优先级的四个位全部配置为 1，即 15，在 Cortex-M 内核中，优先级数值越大，逻辑优先级越低），RT_TICK_PER_SECOND 是一个在 rtconfig.h 中定义的宏，目前等于 100。

列表 9: 代码清单 9-9 SysTick 初始化函数（在 core_cm3.h 中定义）

```
__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks)
{
    /* 非法的重装载寄存器值 */
    if ((ticks - 1UL) > SysTick_LOAD_RELOAD_Msk)
    {
        return (1UL);
    }
    /* 设置重装载寄存器的值 */
    SysTick->LOAD = (uint32_t)(ticks - 1UL);
    /* 设置 SysTick 的中断优先级 */
    NVIC_SetPriority (SysTick_IRQn, (1UL << __NVIC_PRIO_BITS) - 1UL);
    /* 加载 SysTick 计数器值 */
```

(下页继续)

(续上页)

```
    SysTick->VAL = 0UL;
    /* 设置系统定时器的时钟源为 AHBCLK
    使能 SysTick 定时器中断
    使能 SysTick 定时器 */
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
                    SysTick_CTRL_TICKINT_Msk |
                    SysTick_CTRL_ENABLE_Msk;
    return (0UL);
}
```

代码清单 9-8**(3)-2**：更新系统时基，该函数在 clock.c（clock.c 第一次使用需要自行在文件夹 rt-thread3.0.3src 中新建并添加到工程的 rtt/source 组）中实现，具体见代码清单 9-10。

11.3.1 系统时基更新函数

列表 10: 代码清单 9-10 时基更新函数

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

static rt_tick_t rt_tick = 0; /* 系统时基计数器 */                    (1)
extern rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX];


void rt_tick_increase(void)
{
    rt_ubase_t i;
    struct rt_thread *thread;
    rt_tick ++;                                                        (2)

    /* 扫描就绪列表中所有线程的 remaining_tick，如果不为 0，则减 1 */
```

(下页继续)

(续上页)

```
    for(i=0; i<RT_THREAD_PRIORITY_MAX; i++)                    (3)
    {
        thread = rt_list_entry( rt_thread_priority_table[i].next,
                                struct rt_thread,
                                tlist);
        if(thread->remaining_tick > 0)
        {
            thread->remaining_tick --;
        }
    }

    /* 系统调度 */
    rt_schedule();                                             (4)
}
```

代码清单 9-10 **(1)**：系统时基计数器，是一个全局变量，用来记录产生了多少次 SysTick 中断。

代码清单 9-10 **(2)**：系统时基计数器加一操作。

代码清单 9-10 **(3)**：扫描就绪列表中所有线程的 remaining_tick，如果不为 0，则减 1。

代码清单 9-10 **(4)**：进行系统调度。

代码清单 9-8**(3)-1 和 3**：进入中断和离开中断，这两个函数在 irq.c（irq.c 第一次使用需要自行在文件夹 rtthread3.0.3src 中新建并添加到工程的 rtt/source 组）中实现，具体见代码清单 9-11。

列表 11: 代码清单 9-11 进入中断和离开中断函数

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

/* 中断计数器 */
volatile rt_uint8_t rt_interrupt_nest;              (1)

```

(下页继续)

(续上页)

```
/**
* 当 BSP 文件的中断服务函数进入时会调用该函数
*
* @note 请不要在应用程序中调用该函数
*
* @see rt_interrupt_leave
*/
void rt_interrupt_enter(void)                          (2)
{
    rt_base_t level;

    /* 关中断 */
    level = rt_hw_interrupt_disable();

    /* 中断计数器 ++ */
    rt_interrupt_nest ++;

    /* 开中断 */
    rt_hw_interrupt_enable(level);
}


/**
* 当 BSP 文件的中断服务函数离开时会调用该函数
*
* @note 请不要在应用程序中调用该函数
*
* @see rt_interrupt_enter
*/
void rt_interrupt_leave(void)                        (3)
{
    rt_base_t level;
```

(下页继续)

(续上页)

```
    /* 关中断 */
    level = rt_hw_interrupt_disable();

    /* 中断计数器-- */
    rt_interrupt_nest --;

    /* 开中断 */
    rt_hw_interrupt_enable(level);
}
```

代码清单 9-11(1)：中断计数器，是一个全局变量，用了记录中断嵌套次数。

代码清单 9-11(2)：进入中断函数，中断计数器 rt_interrupt_nest 加一操作。当 BSP 文件的中断服务函数进入时会调用该函数，应用程序不能调用，切记。

代码清单 9-11(3)：离开中断函数，中断计数器 rt_interrupt_nest 减一操作。当 BSP 文件的中断服务函数离开时会调用该函数，应用程序不能调用，切记。

11.4 main 函数

main 函数和线程代码变动不大，具体见代码清单 9-11，有变动部分代码已高亮。

列表 12: 代码清单 9-12 main 函数

```
/*
**************************************************************************
*                                包含的头文件
**************************************************************************
*/
#include <rtthread.h>
#include <rthw.h>                                                    (1)
```

(下页继续)

(续上页)

```
#include "ARMCM4.h"
/*
*************************************************************************
*                                 全局变量
*************************************************************************
*/
rt_uint8_t flag1;
rt_uint8_t flag2;
extern rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX];
/*
*************************************************************************
*                        线程控制块 & STACK & 线程声明
*************************************************************************
*/
/* 定义线程控制块 */
struct rt_thread rt_flag1_thread;
struct rt_thread rt_flag2_thread;

ALIGN(RT_ALIGN_SIZE)
/* 定义线程栈 */
rt_uint8_t rt_flag1_thread_stack[512];
rt_uint8_t rt_flag2_thread_stack[512];

/* 线程声明 */
void flag1_thread_entry(void *p_arg);
void flag2_thread_entry(void *p_arg);
/*
*************************************************************************
*                                 函数声明
*************************************************************************
*/
void delay(uint32_t count);
```

(下页继续)

(续上页)

```
/*****************************************************************************
* @brief  main 函数
* @param  无
* @retval 无
*
* @attention
*****************************************************************************
*/
int main(void)
{
    /* 硬件初始化 */
    /* 将硬件相关的初始化放在这里，如果是软件仿真则没有相关初始化代码 */

    /* 关中断 */
    rt_hw_interrupt_disable();                                  (2)

    /* SysTick 中断频率设置 */
    SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );   (3)

    /* 调度器初始化 */
    rt_system_scheduler_init();

    /* 初始化空闲线程 */
    rt_thread_idle_init();                                      (4)

    /* 初始化线程 */
    rt_thread_init( &rt_flag1_thread,                   /* 线程控制块 */
                    "rt_flag1_thread",                  /* 线程名字，字符串形式␣
↪*/
                    flag1_thread_entry,                 /* 线程入口地址 */
                    RT_NULL,                            /* 线程形参 */
```

(下页继续)

(续上页)

```
                    &rt_flag1_thread_stack[0],          /* 线程栈起始地址 */
                    sizeof(rt_flag1_thread_stack) );    /* 线程栈大小，单位为字节 */
    /* 将线程插入到就绪列表 */
    rt_list_insert_before( &(rt_thread_priority_table[0]),&(rt_flag1_thread.tlist) );

    /* 初始化线程 */
    rt_thread_init( &rt_flag2_thread,                   /* 线程控制块 */
                    "rt_flag2_thread",                  /* 线程名字，字符串形式 */
                    flag2_thread_entry,                 /* 线程入口地址 */
                    RT_NULL,                            /* 线程形参 */
                    &rt_flag2_thread_stack[0],          /* 线程栈起始地址 */
                    sizeof(rt_flag2_thread_stack) );    /* 线程栈大小，单位为字节 */
    /* 将线程插入到就绪列表 */
    rt_list_insert_before( &(rt_thread_priority_table[1]),&(rt_flag2_thread.tlist) );

    /* 启动系统调度器 */
    rt_system_scheduler_start();
}

/*
*********************************************************************
*                              函数实现
*********************************************************************
*/
/* 软件延时 */
void delay (uint32_t count)
{
```

(下页继续)

(续上页)

```
    for(; count!=0; count--);
}

/* 线程 1 */
void flag1_thread_entry( void *p_arg )
{
    for( ;; )
    {
#if 0
        flag1 = 1;
        delay( 100 );
        flag1 = 0;
        delay( 100 );

        /* 线程切换，这里是手动切换 */
        rt_schedule();
#else
        flag1 = 1;
        rt_thread_delay(2);                                    (5)
        flag1 = 0;
        rt_thread_delay(2);
#endif
    }
}

/* 线程 2 */
void flag2_thread_entry( void *p_arg )
{
    for( ;; )
    {
#if 0
        flag2 = 1;
```

(下页继续)

(续上页)

```
        delay( 100 );
        flag2 = 0;
        delay( 100 );

        /* 线程切换，这里是手动切换 */
        rt_schedule();
#else
        flag2 = 1;
        rt_thread_delay(2);                                    (6)
        flag2 = 0;
        rt_thread_delay(2);
#endif
    }
}


void SysTick_Handler(void)                                     (7)
{
    /* 进入中断 */
    rt_interrupt_enter();

    rt_tick_increase();

    /* 离开中断 */
    rt_interrupt_leave();
}
```

代码清单 9-11(1)：新包含的两个头文件。

代码清单 9-11(2)：关中断。

代码清单 9-11(3)：初始化 SysTick。

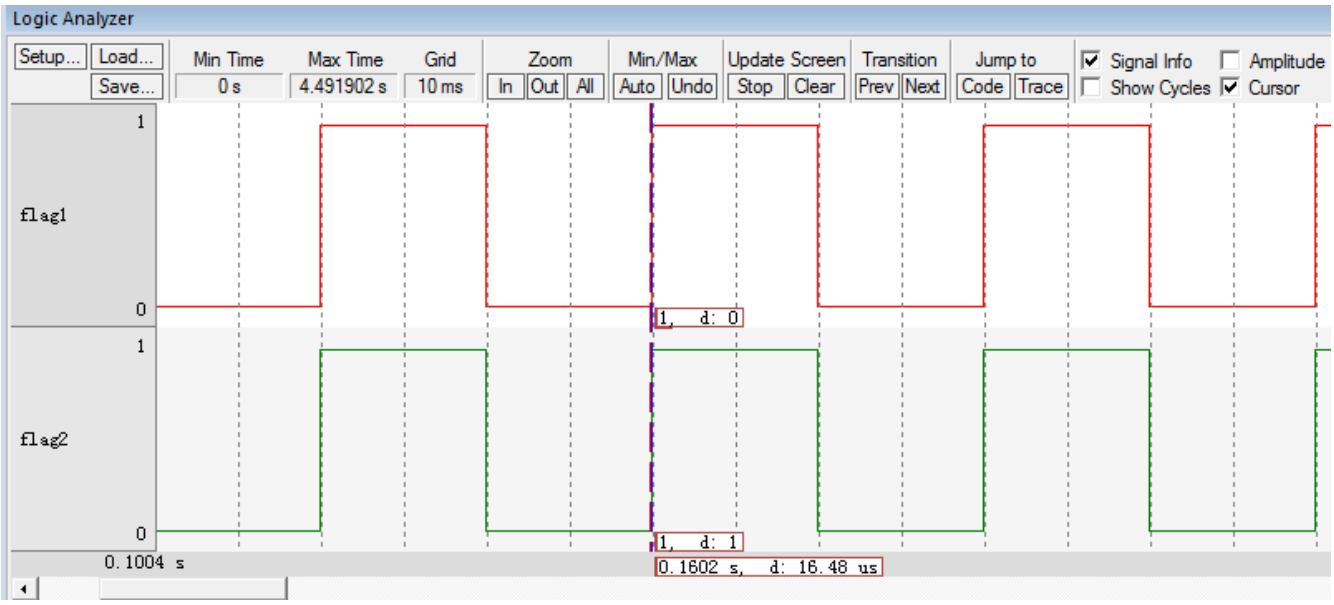
代码清单 9-11(4)：创建空闲线程。

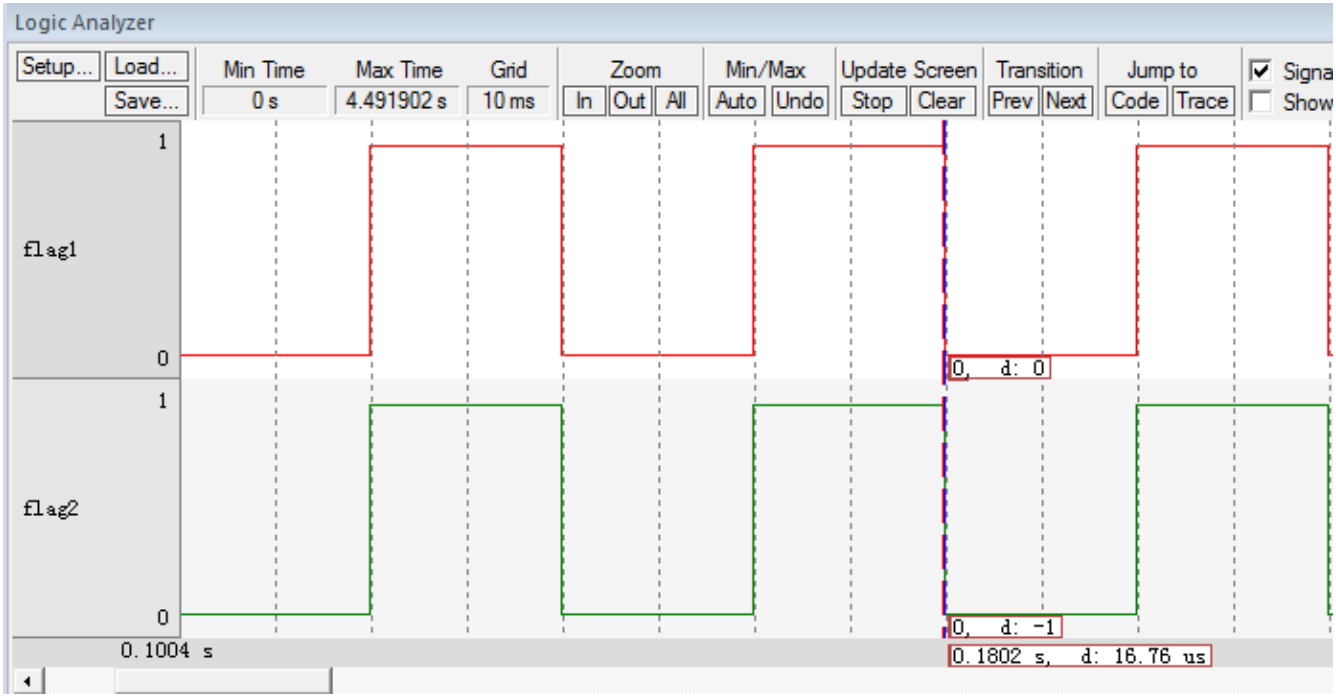
代码清单 *9-11*(**5**) 和 (**6**)：延时函数均由原来的软件延时替代为阻塞延时，延时时间均为 2 个 SysTick 中断周期，即 20ms。

代码清单 *9-11*(**7**)：SysTick 中断服务函数。

11.5 实验现象

进入软件调试，全速运行程序，从逻辑分析仪中可以看到两个线程的波形是完全同步，就好像 CPU 在同时干两件事情，具体仿真的波形图见图 *9-1* 和图 *9-2* 。





从图 *9-1* 和图 *9-2* 可以看出，flag1 和 flag2 的高电平的时间为 (0.1802-0.1602)s，刚好等于阻塞延时的 20ms，所以实验现象跟代码要实现的功能是一致的。

第 12 章　支持多优先级

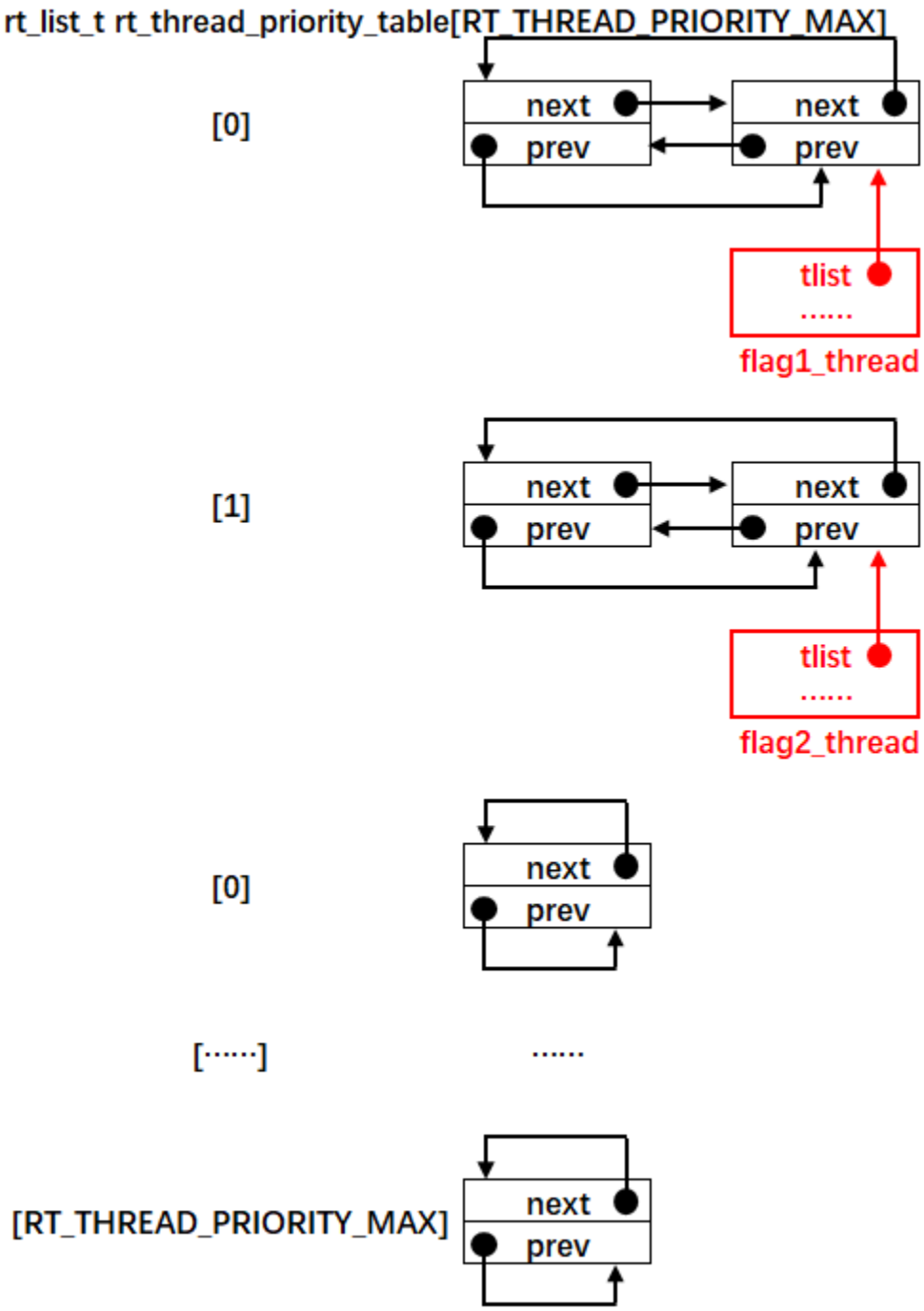
在本章之前，RT-Thread 还没有支持多优先级，只支持两个线程互相切换，从本章开始，线程中我们开始加入优先级的功能。在 RT-Thread 中，数字优先级越小，逻辑优先级越高。

12.1 就绪列表

RT-Thread 要支持多优先级，需要靠就绪列表的支持，从代码上看，就绪列表由两个在 scheduler.c 文件定义的全局变量组成，一个是线程就绪优先级组 rt_thread_ready_priority_group，另一个是线程优先级表 rt_thread_priority_table[RT_THREAD_PRIORITY_MAX]，接下来我们将详细讲解这两个变量是如何帮系统实现多优先级的。

12.1.1 线程就绪优先级组

在本章以前，我们将创建好的两个线程手动插入到线程优先级表（之前我们一直将线程优先级表称为就绪列表，其实不具体区分，这种叫法也没有错）rt_thread_priority_table[RT_THREAD_PRIORITY_MAX] 的索引 0 和索引 1 所在的链表，并没有考虑线程的优先级，具体见图 *10-1*。线程优先级表的索引对应的是线程的优先级，从本章开始，线程将支持优先级，线程启动的时候将根据优先级来决定插入到线程优先级表的位置。



为了快速的找到线程在线程优先级表的插入和移除的位置，RT-Thread 专门设计了一个线程就绪优先级组。从代码上看，线程就绪优先级组就是一个 32 位的整形数，每一个位对应一个优先级。一个就绪优先级组最多只能表示 32 个优先级，如果优先级超过 32 个怎么办，则可以定义一个线

程就绪优先级数组，每一个数组成员都可以表示 32 个优先级，具体支持到少由系统的 RAM 的大小决定，这里我们只讲解支持到 32 个优先级的情况。线程就绪优先级组在 scheduler.c 文件中定义，具体见*代码清单 10-1*。

列表 1: 代码清单 10-1 线程就绪优先级组

```
/* 线程就绪优先级组 */
rt_uint32_t rt_thread_ready_priority_group;
```

那么线程就绪优先级组是如何帮助系统快速地找到线程在线程优先级表的插入和移除的位置? 线程就绪优先级组的每一个位对应一个优先级，位 0 对应优先级 0，位 1 对应优先级 1，以此类推。比如，当优先级为 10 的线程已经准备好，那么就将线程就绪优先级组的位 10 置 1，表示线程已经就绪，然后根据 10 这个索引值，在线程优先级表 10（rt_thread_priority_table[10]）的这个位置插入线程。有关线程就绪优先级组的位号与线程优先级对应的关系具体见图 *10-2*。

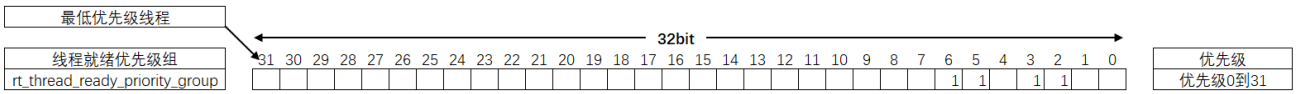
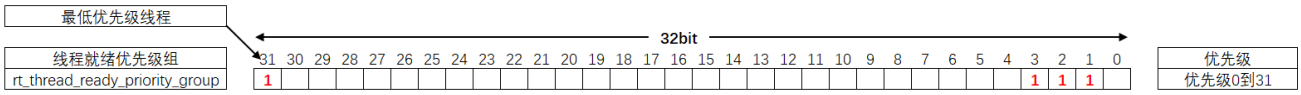
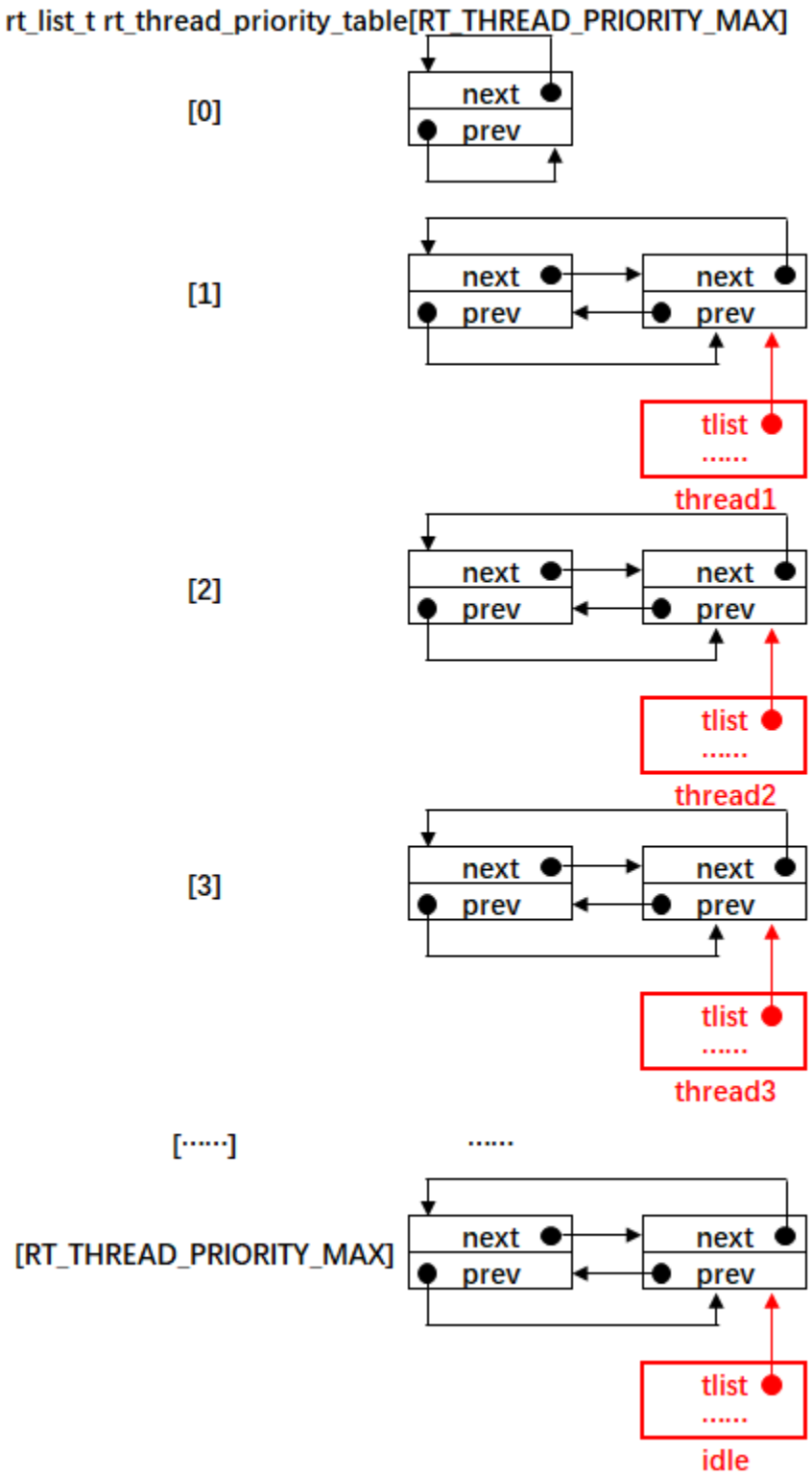


图 *10-2* 线程就绪优先级组的位号与线程优先级对应的关系

12.1.1.1 寻找优先级最高的线程

RT-Thread 是一个根据优先级来调度的抢占式实时操作系统，即在每个系统周期到来时，调度器都会扫描就绪列表，选取优先级最高的线程去执行。假设目前系统中，创建了优先级分别为 1、2 和 3 的线程 1、线程 2 和线程 3，再加上系统默认创建的空闲线程，那么此时线程就绪优先级组的位设置情况和线程优先级表的链表挂载情况就如图 *10-3* 和图 *10-4* 所示那样。




rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX]
[0]
next
prev
[1]
next
prev
next
prev
tlist
.....
thread1
[2]
next
prev
next
prev
tlist
.....
thread2
[3]
next
prev
next
prev
tlist
.....
thread3
[.....]
.....
[RT_THREAD_PRIORITY_MAX]
next
prev
next
prev
tlist
.....
idle

在下一个系统周期来临时，调度器需要选取优先级最高的线程去运行，从图 *10-3* 我们一眼就可以看出线程就绪优先级组从右往左开始数，第一个置 1 的位是位 1，即表示此时就绪的线程当中，优先级最高的是线程 1，然后调度器从线程优先级表的索引 1 下取出线程 1 的线程控制块，从而切换到线程 1。但是，单片机没有眼睛，并不能跟人一样一眼就从线程就绪优先级组中看到那个第一个置 1 的位，怎么办？RT-Thread kservice.c 文件中，有一个专门的函数 __rt_ffs，用来寻找 32 位整形数第一个（从低位开始）置 1 的位号，具体实现见*代码清单 10-2*。

列表 2: 代码清单 10-2 __rt_ffs 函数实现

```
/**
* 该函数用于从一个 32 位的数中寻找第一个被置 1 的位 (从低位开始),
* 然后返回该位的索引 (即位号)
*
* @return 返回第一个置 1 位的索引号。如果全为 0, 则返回 0。
*/
int __rt_ffs(int value)
{
    /* 如果值为 0, 则直接返回 0 */
    if (value == 0) return 0;                                   (1)

    /* 检查 bits [07:00]
    这里加 1 的原因是避免当第一个置 1 的位是位 0 时
    返回的索引号与值都为 0 时返回的索引号重复 */
    if (value & 0xff)                                           (2)
        return __lowest_bit_bitmap[value & 0xff] + 1;

    /* 检查 bits [15:08] */
    if (value & 0xff00)                                         (3)
        return __lowest_bit_bitmap[(value & 0xff00) >> 8] + 9;

    /* 检查 bits [23:16] */
    if (value & 0xff0000)                                       (4)
        return __lowest_bit_bitmap[(value & 0xff0000) >> 16] + 17;
```

(下页继续)

(续上页)

```

    /* 检查 bits [31:24] */                                    (5)
    return __lowest_bit_bitmap[(value & 0xff000000) >> 24] + 25;
}
```

代码清单 10-2(**1**)：如果值为 0，则直接返回 0。

代码清单 10-2(**2**)：检查 bits [07:00]，然后通过 __lowest_bit_bitmap[value & 0xff] + 1 返回第一个置 1 的位号，这里加 1 的原因是避免当第一个置 1 的位是位 0 时返回的索引号与值都为 0 时返回的索引号重复，返回 1 表示优先级为 0 就绪，使用这个索引号的时候再减 1 即可。现在我们在具体分析下 __lowest_bit_bitmap[] 这个数组，该数组在 kservice.c 中定义，具体见代码清单 10-3。

列表 3: 代码清单 10-3 数组 __lowest_bit_bitmap[] 定义

```
/*
* __lowest_bit_bitmap[] 数组的解析
* 将一个 8 位整形数的取值范围 0~255 作为数组的索引，索引值第一个出现 1(从最低位开始) 的位号作为该数组索引下的成员值。
* 举例：十进制数 10 的二进制为：0000 1010，从最低位开始，第一个出现 1 的位号为 bit1，则有__lowest_bit_bitmap[10]=1
* 注意：只需要找到第一个出现 1 的位号即可
*/
const rt_uint8_t __lowest_bit_bitmap[] =
{
    /* 00 */ 0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 10 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 20 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 30 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 40 */ 6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 50 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 60 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 70 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 80 */ 7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
```

(下页继续)

(续上页)

```
    /* 90 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* A0 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* B0 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* C0 */ 6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* D0 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* E0 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* F0 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};
```

代码清单 10-3：要从一个 8 位整形数中从低位开始找出第一个置 1 的位，常规的方法是从低位开始一位一位的判断，优点是逻辑简单好理解，缺点是耗时，这里采取一种空间换时间的方法，即：将 8 位整形数的取值范围 0~255 作为数组 __lowest_bit_bitmap[] 的索引，索引值第一个出现 1(从最低位开始) 的位号作为该数组索引下的成员值。举例：十进制数 10 的二进制为：0000 1010，从最低位开始，第一个出现 1 的位号为 bit1，则有 __lowest_bit_bitmap[10]=1。注意：只需要找到第一个出现 1 的位号即可。

代码清单 10-2(3)、(4)、(5)：依次检查剩下的位，原理同 (2) 。

12.1.2 线程优先级表

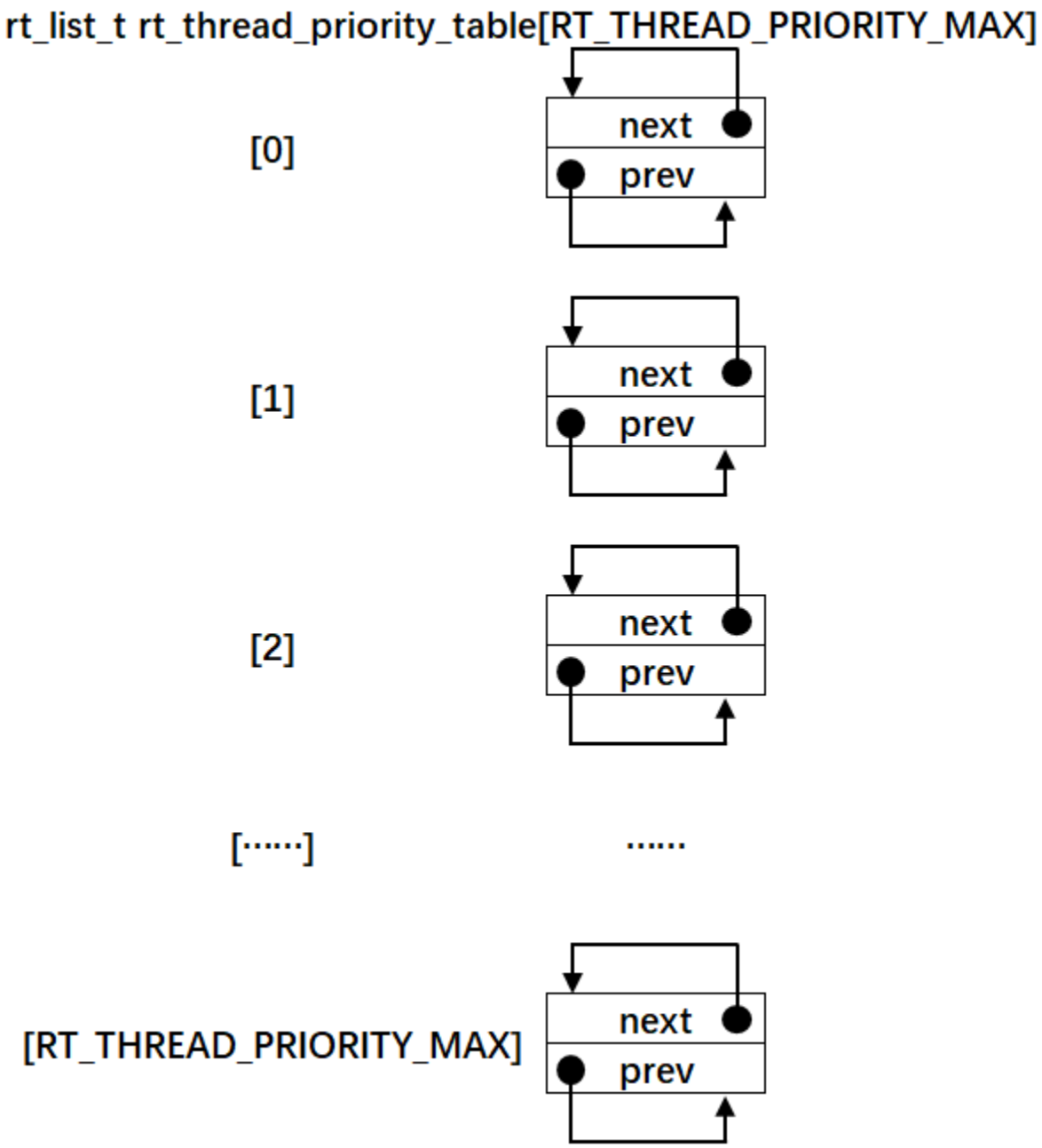
线程优先级表就是我们本章之前讲的就绪列表，因为本章为了讲解方便将就绪列表分为了线程就绪优先级组和线程优先级表，除了本章，往后我们说到的就绪列表就是线程优先级表。线程优先级表是一个在 scheduler.c 中定义的全局数组，具体见*代码清单 10-4*

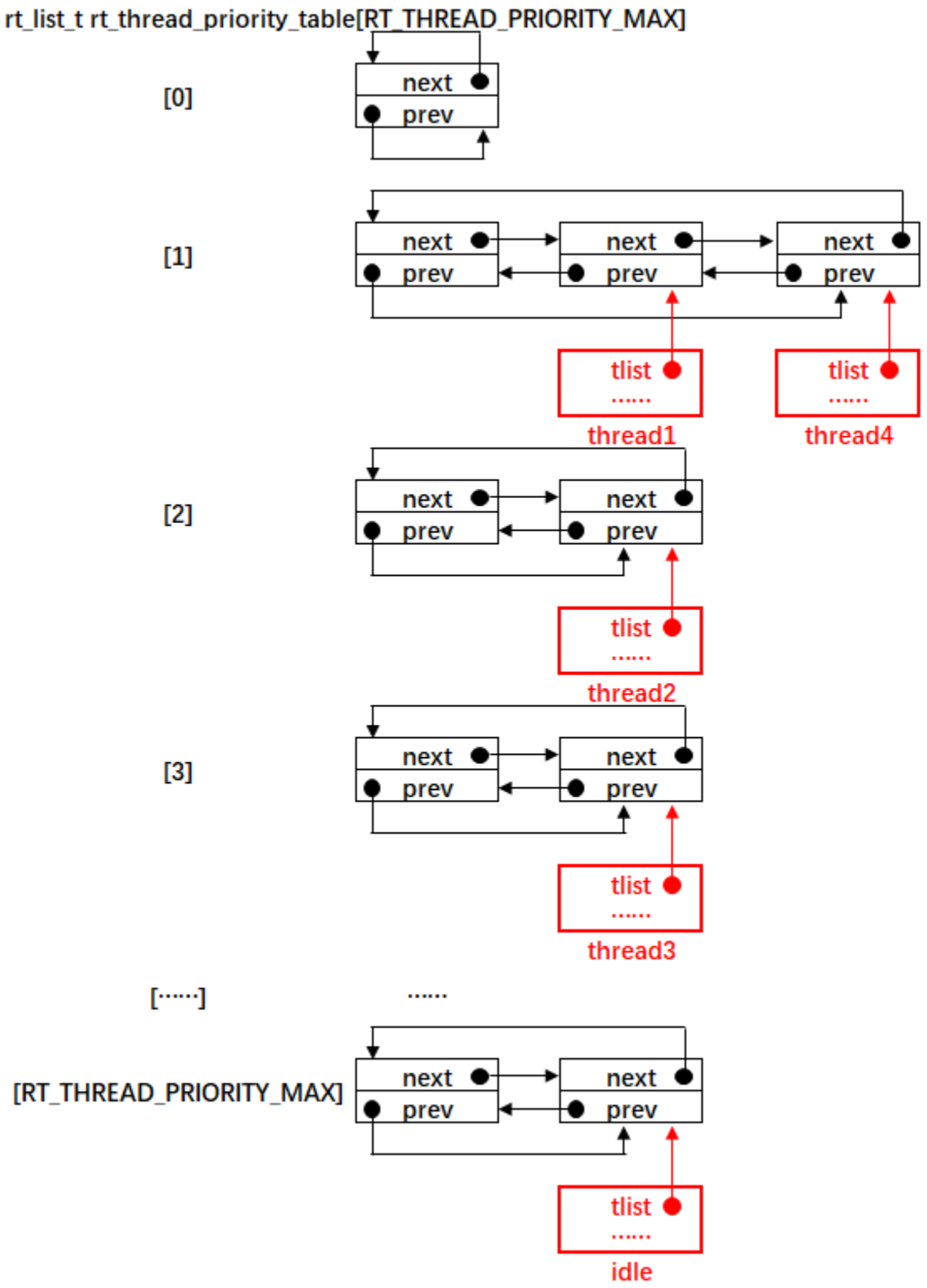
列表 4: 代码清单 10-4 线程优先级表定义

```
/* 线程优先级表 */
rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX];
```

线程优先级表的数据类型为 rt_list，每个索引号对应线程的优先级，该索引下维护着一条双向链表，当线程就绪时，线程就会根据优先级插入到对应索引的链表，同一个优先级的线程都会被插入到同一条链表中（当同一个优先级下有多个线程时，需要时间片的支持，目前本章暂时不支持

时间片，后面的章节再讲解）。一个空的就绪列表和一个有 4 个线程就绪的就绪列表示意图具体见图 *10-5* 和图 *10-6*。




rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX]
[0]
next
prev
[1]
next
prev
next
prev
next
prev
tlist
.....
thread1
tlist
.....
thread4
[2]
next
prev
next
prev
tlist
.....
thread2
[3]
next
prev
next
prev
tlist
.....
thread3
[.....]
.....
[RT_THREAD_PRIORITY_MAX]
next
prev
next
prev
tlist
.....
idle

将线程插入到线程优先级表和移除分别由 scheduler.c 的 rt_schedule_insert_thread() 和 rt_schedule_remove_thread() 这两个函数实现，它们的具体定义见代码清单 10-5 和代码清单 10-6。

12.1.2.1 调度器插入线程

列表 5: 代码清单 10-5 调度器插入线程

```
void rt_schedule_insert_thread(struct rt_thread *thread)
{
    register rt_base_t temp;

    /* 关中断 */
    temp = rt_hw_interrupt_disable();

    /* 改变线程状态 */
    thread->stat = RT_THREAD_READY;

    /* 将线程插入就绪列表 */
    rt_list_insert_before(&(rt_thread_priority_table[thread->current_priority]),
                          &(thread->tlist));

    /* 设置线程就绪优先级组中对应的位 */
    rt_thread_ready_priority_group |= thread->number_mask;

    /* 开中断 */
    rt_hw_interrupt_enable(temp);
}
```

12.1.2.2 调度器删除线程

列表 6: 代码清单 10-6 调度器删除线程

```
void rt_schedule_remove_thread(struct rt_thread *thread)
{
    register rt_base_t temp;


    /* 关中断 */
    temp = rt_hw_interrupt_disable();

    /* 将线程从就绪列表删除 */
    rt_list_remove(&(thread->tlist));

    if (rt_list_isempty(&(rt_thread_priority_table[thread->current_priority])))
    {
        rt_thread_ready_priority_group &= ~thread->number_mask;
    }

    /* 开中断 */
    rt_hw_interrupt_enable(temp);
}
```

12.2 修改代码，支持多优先级

接下来我们在上一章的代码上，继续迭代修改，从而实现多优先级。

12.2.1 修改线程控制块

在线程控制块中增加与优先级相关的成员，具体见高亮部分，其中还增加了错误码和线程状态成员，具体见代码清单 10-7。

列表 7: 代码清单 10-7 修改线程控制块代码

```
struct rt_thread
{
    /* rt 对象 */
    char        name[RT_NAME_MAX];     /* 对象的名字 */
    rt_uint8_t  type;                  /* 对象类型 */
    rt_uint8_t  flags;                 /* 对象的状态 */
    rt_list_t   list;                  /* 对象的列表节点 */

    rt_list_t   tlist;                 /* 线程链表节点 */

    void        *sp;                       /* 线程栈指针 */
    void        *entry;                    /* 线程入口地址 */
    void        *parameter;            /* 线程形参 */
    void        *stack_addr;           /* 线程起始地址 */
    rt_uint32_t stack_size;            /* 线程栈大小，单位为字节 */

    rt_ubase_t  remaining_tick;        /* 用于实现阻塞延时 */

    rt_uint8_t  current_priority;      /* 当前优先级 */          (1)
    rt_uint8_t  init_priority;         /* 初始优先级 */          (2)
    rt_uint32_t number_mask;           /* 当前优先级掩码 */     (3)

    rt_err_t    error;                 /* 错误码 */              (4)
    rt_uint8_t  stat;                  /* 线程的状态 */          (5)
};
```

代码清单 10-7**(1)**：当前优先级。

代码清单 10-7(2)：初始优先级。

代码清单 10-7(3)：当前优先级掩码，即线程就绪优先级组的索引。

代码清单 10-7(4)：错误码，具体取值在 rtdef.h 中定义，具体见代码清单 10-8。

列表 8: 代码清单 10-8 RT-Thread 错误码重定义

```
/* RT-Thread 错误码重定义 */
#define RT_EOK                          0               /* 没有错误 */
#define RT_ERROR                        1               /* 一个常规错误 */
#define RT_ETIMEOUT                     2               /* 超时 */
#define RT_EFULL                        3               /* 资源已满 */
#define RT_EEMPTY                       4               /* 资源为空 */
#define RT_ENOMEM                       5               /* 没有内存 */
#define RT_ENOSYS                       6               /* No system */
#define RT_EBUSY                        7               /* 忙 */
#define RT_EIO                          8               /* IO 错误 */
#define RT_EINTR                        9               /* 中断系统调用 */
#define RT_EINVAL                       10              /* 无效形参 */
```

代码清单 10-7(5)：线程的状态，具体取值在 rtdef.h 中定义，具体见代码清单 10-9。

列表 9: 代码清单 10-9 线程状态定义

```
/*
* 线程状态定义
*/
#define RT_THREAD_INIT                  0x00                /* 初始态 */
#define RT_THREAD_READY                 0x01                /* 就绪态 */
#define RT_THREAD_SUSPEND               0x02                /* 挂起态 */
#define RT_THREAD_RUNNING               0x03                /* 运行态 */
#define RT_THREAD_BLOCK                 RT_THREAD_SUSPEND   /* 阻塞态 */
#define RT_THREAD_CLOSE                 0x04                /* 关闭态 */
#define RT_THREAD_STAT_MASK             0x0f
```

(下页继续)

(续上页)

```

#define RT_THREAD_STAT_SIGNAL           0x10
#define RT_THREAD_STAT_SIGNAL_READY     (RT_THREAD_STAT_SIGNAL | RT_THREAD_
↪READY)
#define RT_THREAD_STAT_SIGNAL_SUSPEND   0x20
#define RT_THREAD_STAT_SIGNAL_MASK      0xf0
```

12.2.2 修改调度器初始化函数 rt_system_scheduler_init()

列表 10: 代码清单 10-10 系统调度器初始化

```
/* 初始化系统调度器 */
void rt_system_scheduler_init(void)
{
#if 0
    register rt_base_t offset;

    /* 线程就绪列表初始化 */
    for (offset = 0; offset < RT_THREAD_PRIORITY_MAX; offset ++)
    {
            rt_list_init(&rt_thread_priority_table[offset]);
    }

    /* 初始化当前线程控制块指针 */
    rt_current_thread = RT_NULL;
#else
    register rt_base_t offset;


    /* 线程优先级表初始化 */                                    (1)
    for (offset = 0; offset < RT_THREAD_PRIORITY_MAX; offset ++)
```

(下页继续)

(续上页)

```
    {
            rt_list_init(&rt_thread_priority_table[offset]);
    }

    /* 初始化当前优先级为空闲线程的优先级 */
    rt_current_priority = RT_THREAD_PRIORITY_MAX - 1;                (2)

    /* 初始化当前线程控制块指针 */
    rt_current_thread = RT_NULL;                                     (3)

    /* 初始化线程就绪优先级组 */
    rt_thread_ready_priority_group = 0;                              (4)
#endif
}
```

代码清单 10-10**(1)**：线程优先级表初始化。

代码清单 10-10**(2)**：初始化当前优先级为空闲线程的优先级，rt_current_priority 是在 scheduler.c 中定义的全局变量，表示当前运行线程的优先级。

代码清单 10-10**(3)**：初始化当前线程控制块指针为 0，rt_current_thread 是在 scheduler.c 中定义的全局指针，表示指向当前正在运行的线程的线程控制块。

代码清单 10-10**(4)**：初始化线程就绪优先级组为 0，即还没有一个线程就绪。

12.2.3 修改线程初始化函数 rt_thread_init()

在线程初始化函数中添加优先级形参，在函数中初始化线程控制块中优先级、错误码和线程状态成员，具体见代码清单 10-11 高亮部分。

列表 11: 代码清单 10-11 线程初始化函数 rt_thread_init()

```
rt_err_t rt_thread_init(struct rt_thread *thread,
                        const char       *name,
                        void (*entry)(void *parameter),
                        void             *parameter,
                        void             *stack_start,
                        rt_uint32_t       stack_size,
                        rt_uint8_t        priority)
{
    /* 线程对象初始化 */
    /* 线程结构体开头部分的成员就是 rt_object_t 类型 */
    rt_object_init((rt_object_t)thread, RT_Object_Class_Thread, name);
    rt_list_init(&(thread->tlist));

    thread->entry = (void *)entry;
    thread->parameter = parameter;

    thread->stack_addr = stack_start;
    thread->stack_size = stack_size;

    /* 初始化线程栈，并返回线程栈指针 */
    thread->sp = (void *)rt_hw_stack_init( thread->entry,
                                            thread->parameter,
                                            (void *)((char *)thread->stack_addr␣
↪+ thread->stack_size - 4) );

    thread->init_priority    = priority;
    thread->current_priority = priority;
    thread->number_mask = 0;

    /* 错误码和状态 */
```

(下页继续)

(续上页)

```
    thread->error = RT_EOK;
    thread->stat  = RT_THREAD_INIT;

    return RT_EOK;
}
```

12.2.4 添加线程启动函数 rt_thread_startup()

在本章之前，创建好线程之后会调用函数 rt_list_insert_before() 将线程插入到线程优先级表，本章开始我们另外独立添加一个函数 rt_thread_startup() 来实现该功能，该函数在 thread.c 中定义，具体实现见*代码清单 10-12*。

列表 12: 代码清单 10-12 线程启动函数 rt_thread_startup()

```
/**
* 启动一个线程并将其放到系统的就绪列表中
*
* @param thread 待启动的线程
*
* @return 操作状态, RT_EOK on OK, -RT_ERROR on error
*/
rt_err_t rt_thread_startup(rt_thread_t thread)
{
    /* 设置当前优先级为初始优先级 */
    thread->current_priority = thread->init_priority;          (1)
    thread->number_mask = 1L << thread->current_priority;      (2)

    /* 改变线程的状态为挂起状态 */
    thread->stat = RT_THREAD_SUSPEND;                          (3)
    /* 然后恢复线程 */
    rt_thread_resume(thread);                                  (4)
```

(下页继续)

(续上页)

```

    if (rt_thread_self() != RT_NULL)                                (5)
    {
        /* 系统调度 */
        rt_schedule();                                              (6)
    }

    return RT_EOK;
}
```

代码清单 10-12(1)：设置当前优先级为初始优先级。

代码清单 10-12(2)：根据优先级计算线程就绪优先级组的掩码值。

代码清单 10-12(3)：设置线程的状态为挂起态，等下会恢复。

代码清单 10-12(4)：恢复线程，即将线程插入到就绪列表，由单独的 rt_thread_resume() 函数来实现，该函数在 thread.c 中定义，具体实现见代码清单 10-13。

列表 13: 代码清单 10-13 线程恢复函数 rt_thread_resume()

```
/**
* 该函数用于恢复一个线程然后将其放到就绪列表
*
* @param thread 需要被恢复的线程
*
* @return 操作状态, RT_EOK on OK, -RT_ERROR on error
*/
rt_err_t rt_thread_resume(rt_thread_t thread)
{
    register rt_base_t temp;

    /* 将被恢复的线程必须在挂起态，否则返回错误码 */
```

(下页继续)

(续上页)

```
    if ((thread->stat & RT_THREAD_STAT_MASK) != RT_THREAD_SUSPEND)
    {
        return -RT_ERROR;
    }

    /* 关中断 */
    temp = rt_hw_interrupt_disable();

    /* 从挂起队列移除 */
    rt_list_remove(&(thread->tlist));

    /* 开中断 */
    rt_hw_interrupt_enable(temp);

    /* 插入就绪列表 */
    rt_schedule_insert_thread(thread);

    return RT_EOK;
}
```

代码清单 10-12**(5)**：rt_thread_self() 是一个在 thread.c 中定义的函数，具体见代码清单 10-14。用于返回全局指针 rt_current_thread，该指针指向当前正在运行的线程的线程控制块，在系统没有启动之前，rt_current_thread 的值为 RT_NULL，是在代码清单 10-10 调度器初始化函数中初始化的，所以不会执行系统调度。

列表 14: 代码清单 10-14 rt_thread_self() 函数

```
rt_thread_t rt_thread_self(void)
{
    return rt_current_thread;
}
```

代码清单 10-12(6)：系统调度，暂时不会执行，因为系统还没有启动，rt_current_thread 的值还是 RT_NULL。系统调度函数接下来也需要修改。

12.2.5 修改空闲线程初始化函数 rt_thread_idle_init()

修改空闲线程初始化函数，将原先的将线程插入到就绪列表的部分代码修改为 rt_thread_startup() 代替，具体见的高亮部分代码清单 10-15。

列表 15: 代码清单 10-15 空闲线程初始化函数 rt_thread_idle_init()

```
void rt_thread_idle_init(void)
{
    /* 初始化线程 */
    rt_thread_init(&idle,
                   "idle",
                   rt_thread_idle_entry,
                   RT_NULL,
                   &rt_thread_stack[0],
                   sizeof(rt_thread_stack),
                   RT_THREAD_PRIORITY_MAX-1);

    /* 将线程插入到就绪列表 */
    //rt_list_insert_before( &(rt_thread_priority_table[RT_THREAD_PRIORITY_
↪MAX-1]),&(idle.tlist) );
```

(下页继续)

(续上页)

```
    rt_thread_startup(&idle);
}
```

12.2.6 修改启动系统调度器函数 rt_system_scheduler_start()

修改系统调度器函数 rt_system_scheduler_start()，不再是手动指定第一个需要运行的线程，而是根据优先级来决定第一个运行的线程，具体修改见*代码清单 10-16* 的高亮部分。

列表 16: 代码清单 10-16 系统调度器函数 rt_system_scheduler_start()

```
/* 启动系统调度器 */
void rt_system_scheduler_start(void)
{
#if 0
    register struct rt_thread *to_thread;

    /* 手动指定第一个运行的线程 */
    to_thread = rt_list_entry(rt_thread_priority_table[0].next,
                              struct rt_thread,
                              tlist);
    rt_current_thread = to_thread;

    /* 切换到第一个线程，该函数在 context_rvds.S 中实现，在 rthw.h 声明，
    用于实现第一次线程切换。当一个汇编函数在 C 文件中调用的时候，
    如果有形参，则执行的时候会将形参传入到 CPU 寄存器 r0。*/
    rt_hw_context_switch_to((rt_uint32_t)&to_thread->sp);
#else
    register struct rt_thread *to_thread;
    register rt_ubase_t highest_ready_priority;

```

(下页继续)

(续上页)

```
    /* 获取就绪的最高优先级 */                                    (1)
    highest_ready_priority = __rt_ffs(rt_thread_ready_priority_group) - 1;

    /* 获取将要运行线程的线程控制块 */                            (2)
    to_thread = rt_list_entry(rt_thread_priority_table[highest_ready_
↪priority].next,
                              struct rt_thread,
                              tlist);

    rt_current_thread = to_thread;                                (3)

    /* 切换到新的线程 */
    rt_hw_context_switch_to((rt_uint32_t)&to_thread->sp);         (4)

    /* 永远不会返回 */                                            (5)
#endif
}
```

代码清单 10-12(1)：从线程就绪优先级组中获取到就绪的最高优先级。

代码清单 10-12(2)：根据就绪的最高优先级从线程优先级表中获取到线程控制块。

代码清单 10-12(3)：更新全局指针 rt_current_thread 的值。

代码清单 10-12(4)：切换到新的线程。

代码清单 10-12(5)：永远不会返回，以后将在线程之间不断切换。

12.2.7 修改系统调度函数 rt_schedule ()

系统调度函数 rt_schedule() 将不再像本章之前的那样，在三个线程之间轮流切换，而是需要根据优先级来实现，即系统选择就绪线程当中优先级最高的来运行，具体修改见代码清单 10-17 的高亮部分。

列表 17: 代码清单 10-17 系统调度函数 rt_schedule ()

```
void rt_schedule(void)
{
#if 0
    struct rt_thread *to_thread;
    struct rt_thread *from_thread;

    /* 如果当前线程是空闲线程，那么就去尝试执行线程 1 或者线程 2，
    看看他们的延时时间是否结束，如果线程的延时时间均没有到期，
    那就返回继续执行空闲线程 */
    if ( rt_current_thread == &idle )
    {
        if (rt_flag1_thread.remaining_tick == 0)
        {
            from_thread = rt_current_thread;
            to_thread = &rt_flag1_thread;
            rt_current_thread = to_thread;
        }
        else if (rt_flag2_thread.remaining_tick == 0)
        {
            from_thread = rt_current_thread;
            to_thread = &rt_flag2_thread;
            rt_current_thread = to_thread;
        }
        else
        {
            return;          /* 线程延时均没有到期则返回，继续执行空闲线程 */
        }
    }
    else
    {
```

(下页继续)

(续上页)

```
        /* 如果当前线程是线程 1 或者线程 2 的话，
        检查下另外一个线程，如果另外的线程不在延时中，
        就切换到该线程。否则，判断下当前线程是否应该进入延时状态，
        如果是的话，就切换到空闲线程。否则就不进行任何切换 */
        if (rt_current_thread == &rt_flag1_thread)
        {
            if (rt_flag2_thread.remaining_tick == 0)
            {
                from_thread = rt_current_thread;
                to_thread = &rt_flag2_thread;
                rt_current_thread = to_thread;
            }
            else if (rt_current_thread->remaining_tick != 0)
            {
                from_thread = rt_current_thread;
                to_thread = &idle;
                rt_current_thread = to_thread;
            }
            else
            {
                return;     /* 返回，不进行切换，因为两个线程都处于延时中 */
            }
        }
        else if (rt_current_thread == &rt_flag2_thread)
        {
            if (rt_flag1_thread.remaining_tick == 0)
            {
                from_thread = rt_current_thread;
                to_thread = &rt_flag1_thread;
                rt_current_thread = to_thread;
            }
            else if (rt_current_thread->remaining_tick != 0)
```

(下页继续)

(续上页)

```
            {
                from_thread = rt_current_thread;
                to_thread = &idle;
                rt_current_thread = to_thread;
            }
            else
            {
                return;     /* 返回，不进行切换，因为两个线程都处于延时中 */
            }
        }
    }
#else
    rt_base_t level;
    register rt_ubase_t highest_ready_priority;
    struct rt_thread *to_thread;
    struct rt_thread *from_thread;

    /* 关中断 */
    level = rt_hw_interrupt_disable();

    /* 获取就绪的最高优先级 */                          (1)
    highest_ready_priority = __rt_ffs(rt_thread_ready_priority_group) - 1;
    /* 获取就绪的最高优先级对应的线程控制块 */                (2)
    to_thread = rt_list_entry(rt_thread_priority_table[highest_ready_
↪priority].next,
                              struct rt_thread,
                              tlist);

    /* 如果目标线程不是当前线程，则要进行线程切换 */
    if (to_thread != rt_current_thread)                (3)
    {
        rt_current_priority = (rt_uint8_t)highest_ready_priority;
```

(下页继续)

(续上页)

```
        from_thread         = rt_current_thread;
        rt_current_thread   = to_thread;

        rt_hw_context_switch((rt_uint32_t)&from_thread->sp,
                             (rt_uint32_t)&to_thread->sp);

        /* 开中断 */
        rt_hw_interrupt_enable(level);

    }
    else
    {
        /* 开中断 */
        rt_hw_interrupt_enable(level);
    }
#endif

    /* 产生上下文切换 */
    rt_hw_context_switch((rt_uint32_t)&from_thread->sp,(rt_uint32_t)&to_
↪thread->sp);
}
```

代码清单 10-17(1)：从线程就绪优先级组中获取到就绪的最高优先级。

代码清单 10-17(2)：获取就绪的最高优先级对应的线程控制块，并存在 to_thread 中。

代码清单 10-17(3)：如果 to_thread 不是当前线程，则要进行线程切换，否则则开启中断退出继续执行当前线程。

12.2.8 修改阻塞延时函数 rt_thread_delay()

修改阻塞延时函数 rt_thread_delay()，具体修改见*代码清单 10-18* 的高亮部分。

列表 18: 代码清单 10-18 阻塞延时函数 rt_thread_delay()

```
void rt_thread_delay(rt_tick_t tick)
{
#if 0
    struct rt_thread *thread;

    thread = rt_current_thread;
    thread->remaining_tick = tick;

    /* 进行任务调度 */
    rt_schedule();
#else
    register rt_base_t temp;
    struct rt_thread *thread;

    /* 失能中断 */
    temp = rt_hw_interrupt_disable();

    thread = rt_current_thread;
    thread->remaining_tick = tick;

    /* 改变线程状态 */
    thread->stat = RT_THREAD_SUSPEND;                          (1)
    rt_thread_ready_priority_group &= ~thread->number_mask;    (2)

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

```

(下页继续)

(续上页)

```
    /* 进行系统调度 */
    rt_schedule();
#endif
}
```

代码清单 10-18(1)：将线程的状态改为挂起，接下来将进入延时，暂时放弃 CPU 的使用权。

代码清单 10-18(2)：根据优先级将线程就绪优先级组中对应的位清零。严格来说，还需要将线程从线程优先级表中移除，但是鉴于我们目前的时基更新函数 rt_tick_increase() 还是需要通过扫描线程优先级表中来判断线程的延时时间是否到期，所以不能讲线程从就绪列表移除。当我们在接下来的“定时器”章节中，会有一个全新的延时方法，到时候延时的时候除了根据优先级将线程就绪优先级组中对应的位清零外，还需要线程从线程优先级表中移除。

12.2.9 修改时基更新函数 rt_tick_increase()

修改时基更新函数 rt_tick_increase()，具体修改见代码清单 10-19 的高亮部分。

列表 19: 代码清单 10-19 时基更新函数 rt_tick_increase()

```
:linenos:

void rt_tick_increase(void)
{
    rt_ubase_t i;
    struct rt_thread *thread;
    rt_tick ++;
#if 0
    /* 扫描就绪列表中所有线程的 remaining_tick，如果不为 0，则减 1 */
    for(i=0; i<RT_THREAD_PRIORITY_MAX; i++)
    {
        thread = rt_list_entry( rt_thread_priority_table[i].next,
                                struct rt_thread,
```

(下页继续)

(续上页)

```
                                tlist);
        if(thread->remaining_tick > 0)
        {
            thread->remaining_tick --;
        }
    }
#else
    /* 扫描就绪列表中所有线程的 remaining_tick，如果不为 0，则减 1 */
    for(i=0; i<RT_THREAD_PRIORITY_MAX; i++)
    {
        thread = rt_list_entry( rt_thread_priority_table[i].next,
                                struct rt_thread,
                                tlist);
        if(thread->remaining_tick > 0)
        {
            thread->remaining_tick --;
            if(thread->remaining_tick == 0)
            {
                //rt_schedule_insert_thread(thread);
                rt_thread_ready_priority_group |= thread->number_mask; (1)
            }
        }
    }
#endif
    /* 任务调度 */
    rt_schedule();
}
```

代码清单 10-19(1)：如果线程的延时时间 remaining_tick 递减为 0，则表示延时时间结束，需要将线程插入到线程优先级表，即简单的根据优先级将线程就绪优先级组中对应的位置 1 即可。因为在阻塞延时函数中，我们是通过清除线程就绪优先级组中对应的位来让线程挂起的，并没有将线程从线程优先级表里面移除，所以这里将 rt_schedule_insert_thread(thread); 注释掉。

12.3 main 函数

本章 main 函数与上一章基本一致，修改不大，具体修改见*代码清单 10-20* 的高亮部分。

列表 20: 代码清单 10-20 main 函数

```
int main(void)
{
    /* 硬件初始化 */
    /* 将硬件相关的初始化放在这里，如果是软件仿真则没有相关初始化代码 */

    /* 关中断 */
    rt_hw_interrupt_disable();

    /* SysTick 中断频率设置 */
    SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );

    /* 调度器初始化 */
    rt_system_scheduler_init();

    /* 初始化空闲线程 */
    rt_thread_init();

    /* 初始化线程 */
    rt_thread_init( &rt_flag1_thread,                    /* 线程控制块 */
                    "rt_flag1_thread",                   /* 线程名字，字符串形式 */
                    flag1_thread_entry,                  /* 线程入口地址 */
                    RT_NULL,                             /* 线程形参 */
                    &rt_flag1_thread_stack[0],           /* 线程栈起始地址 */
                    sizeof(rt_flag1_thread_stack),       /* 线程栈大小，单位为字节 */
                    2);                                  /* 优先级 */  (1)
```

(下页继续)

(续上页)

```
    /* 将线程插入到就绪列表 */                                          (2)
    //rt_list_insert_before( &(rt_thread_priority_table[0]),&(rt_flag1_thread.tlist) );
    rt_thread_startup(&rt_flag1_thread);

    /* 初始化线程 */
    rt_thread_init( &rt_flag2_thread,                    /* 线程控制块 */
                    "rt_flag2_thread",                   /* 线程名字，字符串形式 */
                    flag2_thread_entry,                  /* 线程入口地址 */
                    RT_NULL,                             /* 线程形参 */
                    &rt_flag2_thread_stack[0],           /* 线程栈起始地址 */
                    sizeof(rt_flag2_thread_stack),       /* 线程栈大小，单位为字节 */
                    3);                                  /* 优先级 */  (3)
    /* 将线程插入到就绪列表 */                                          (4)
    //rt_list_insert_before( &(rt_thread_priority_table[1]),&(rt_flag2_thread.tlist) );
    rt_thread_startup(&rt_flag2_thread);

    /* 启动系统调度器 */
    rt_system_scheduler_start();
}

/*
*********************************************************************
*                              函数实现
*********************************************************************
*/
/* 软件延时 */
void delay (uint32_t count)
{
```

(下页继续)

(续上页)

```
    for(; count!=0; count--);
}

/* 线程 1 */
void flag1_thread_entry( void *p_arg )
{
    for( ;; )
    {
        flag1 = 1;
        rt_thread_delay(2);
        flag1 = 0;
        rt_thread_delay(2);
    }
}

/* 线程 2 */
void flag2_thread_entry( void *p_arg )
{
    for( ;; )
    {
        flag2 = 1;
        rt_thread_delay(2);
        flag2 = 0;
        rt_thread_delay(2);
    }
}

void SysTick_Handler(void)
{
    /* 进入中断 */
    rt_interrupt_enter();

```

(下页继续)

(续上页)

```
    /* 更新时基 */
    rt_tick_increase();

    /* 离开中断 */
    rt_interrupt_leave();
}
```

代码清单 *10-20*(**1**)：设置线程 1 的优先级为 2，数字优先级越高，逻辑优先级越低。

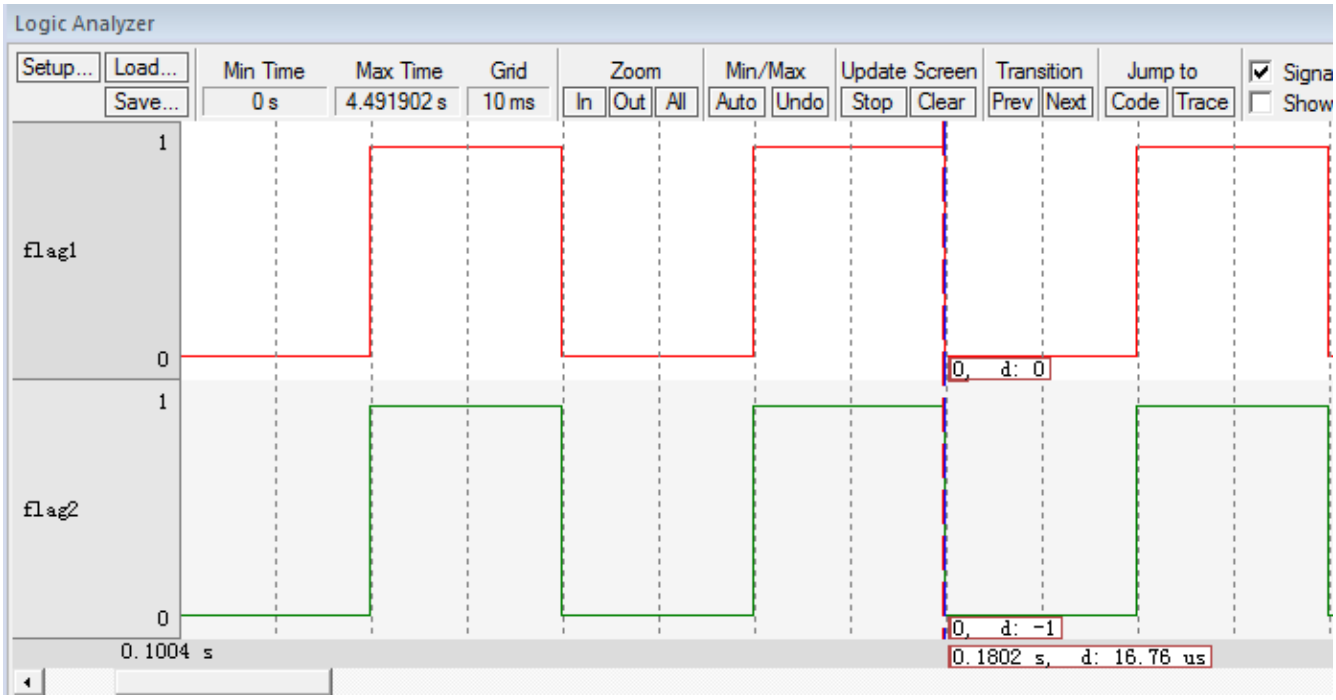
代码清单 *10-20*(**2**)：启动线程，即将线程插入到就绪列表，但是还不会运行，因为系统还没有启动。

代码清单 *10-20*(**3**)：设置线程 1 的优先级为 3，数字优先级越高，逻辑优先级越低。

代码清单 *10-20*(**4**)：启动线程，即将线程插入到就绪列表，但是还不会运行，因为系统还没有启动。

12.4 实验现象

进入软件调试，全速运行程序，从逻辑分析仪中可以看到两个线程的波形是完全同步，就好像 CPU 在同时干两件事情，具体仿真的波形图见图 *10-7* 和图 *10-8*。



从图 *10-7* 和图 *10-8* 可以看出，flag1 和 flag2 的高电平的时间为 (0.1802-0.1602)s，刚好等于阻塞延时的 20ms，所以实验现象跟代码要实现的功能是一致的。。

第 13 章　定时器的实现

在本章之前，为了实现线程的阻塞延时，在线程控制块中内置了一个延时变量 remaining_tick。每当线程需要延时的时候，就初始化 remaining_tick 需要延时的时间，然后将线程挂起，这里的挂起只是将线程在线程就绪优先级组中对应的位清 0，并不会将线程从线程优先级表（即就绪列表）中删除。当每次时基中断（SysTick 中断）来临时，就扫描就绪列表中的每个线程的 remaining_tick，如果 remaining_tick 大于 0 则递减一次，然后判断 remaining_tick 是否为 0，如果为 0 则表示延时时间到，将该线程就绪（即将线程在线程就绪优先级组中对应的位置位），然后等待系统下一次调度。这种延时的缺点是，在每个时基中断中需要对所有线程都扫描一遍，费时，优点是容易理解。之所以先这样讲解是为了慢慢地过度到 RT-Thread 定时器的讲解。

在 RT-Thread 中，每个线程都内置一个定时器，当线程需要延时的时候，则先将线程挂起，然后内置的定时器就会启动，并且将定时器插入到一个全局的系统定时器列表 rt_timer_list，这个全局的系统定时器列表维护着一条双向链表，每个节点代表了正在延时的线程的定时器，节点按照延时时间大小做升序排列。当每次时基中断（SysTick 中断）来临时，就扫描系统定时器列表的第一个定时器，看看延时时间是否到，如果到则让该定时器对应的线程就绪，如果延时时间不到，则退出扫描，因为定时器节点是按照延时时间升序排列的，第一个定时器延时时间不到期的话，那后面的定时器延时时间自然不到期。比起第一种方法，这种方法就大大缩短了寻找延时到期的线程的时间。

13.1 实现定时器

接下来具体讲解下 RT-Thread 定时器的实现，彻底掀开定时器这层朦胧的面纱，要知道这部分功能的有些细节方面可是花了我好些功夫才完全看明白。

13.1.1 系统定时器列表

在 RT-Thread 中，定义了一个全局的系统定时器列表，当线程需要延时时，就先把线程挂起，然后线程内置的定时器将线程挂起到这个系统定时器列表中，系统定时器列表维护着一条双向链表，节点按照定时器的延时时间的大小做升序排列。该系统定时器在 timer.c （timer.c 第一次使用需要在 rtthread3.0.3src 目录下新建，然后添加到工程的 rtt/source 组中）中定义，具体实现见*代码清单 11-1*。

列表 1: 代码清单 11-1 系统定时器列表

```
/* 硬件定时器列表 */
static rt_list_t rt_timer_list[RT_TIMER_SKIP_LIST_LEVEL];      (1)
```

代码清单 11-1(1)：系统定时器列表是一个 rt_list 类型的数组，数组的大小由在 rtdef.h 中定义的宏 RT_TIMER_SKIP_LIST_LEVEL 决定，默认定义为 1，即数组只有一个成员。

13.1.2 系统定时器列表初始化

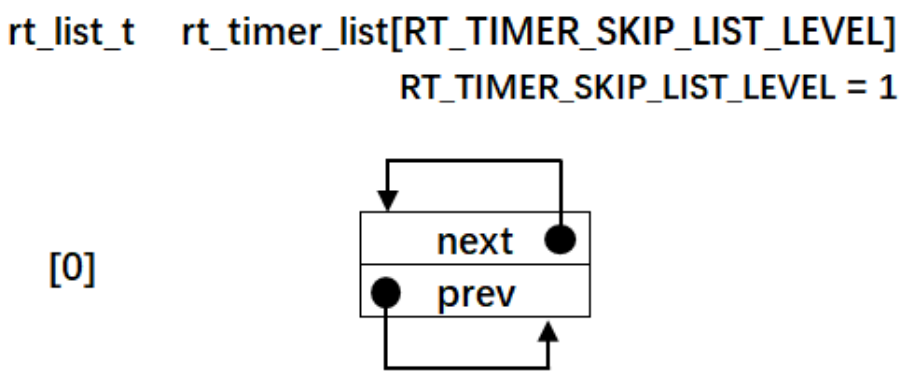
系统定时器列表初始化由函数 rt_system_timer_init() 来完成，在初始化调度器前需要先初始化系统定时器列表。该函数在 timer.c 中定义，具体实现见*代码清单 11-2*。

列表 2: 代码清单 11-2 系统定时器列表初始化

```
void rt_system_timer_init(void)
{
    int i;
    for (i = 0; i < sizeof(rt_timer_list) / sizeof(rt_timer_list[0]); i++) (1)
    {
        rt_list_init(rt_timer_list + i);     (2)
    }
}
```

代码清单 *11-2*(**1**)：系统定时器列表是一个 rt_list 节点类型的数组，rt_timer_list 里面的成员就是一个双向链表的根节点，有多少个成员就初始化多少个根节点，目前只有一个，所以该 for 循环只执行一次。

代码清单 *11-2*(**2**)：初始化节点，即初始化节点的 next 和 prev 这两个指针指向节点本身。初始化好的系统定时器列表的示意图具体见图 *11-1*。



13.1.3 定义定时器结构体

定时器统一由一个定时器结构体来管理，该结构体在 rtdef.h 中定义，具体实现见代码清单 *11-3*。

列表 3: 代码清单 11-3 定时器结构体

```
/*
* 定时器结构体
*/
struct rt_timer
{
    struct rt_object parent; /* 从 rt_object 继承 */        (1)
    rt_list_t row[RT_TIMER_SKIP_LIST_LEVEL]; /* 节点 */    (2)
    void (*timeout_func)(void *parameter); /* 超时函数 */ (3)
    void *parameter; /* 超时函数形参 */                    (4)
    rt_tick_t init_tick; /* 定时器实际需要延时的时间 */    (5)
```

(下页继续)

(续上页)

```
    rt_tick_t timeout_tick; /* 定时器实际超时时的系统节拍数 */ (6)
};
typedef struct rt_timer *rt_timer_t;                          (7)
```

代码清单 11-3(**1**)：定时器也属于内核对象，也会在自身结构体里面包含一个内核对象类型的成员，通过这个成员可以将定时器挂到系统对象容器里面。

代码清单 11-3(**2**)：定时器自身的节点，通过该节点可以实现将定时器插入到系统定时器列表。RT_TIMER_SKIP_LIST_LEVEL 在 rtdef.h 中定义，默认为 0。

代码清单 11-3 (**3**)：定时器超时函数，当定时器延时到期时，会调用相应的超时函数，该函数接下来会讲解。

代码清单 11-3 (**4**)：定时器超时函数形参。

代码清单 11-3 (**5**)：定时器实际需要延时的时间，单位为 tick。

代码清单 11-3 (**6**)：定时器实际超时时的系统节拍数。这个如何理解？我们知道系统定义了一个全局的系统时基计数器 rt_tick（在 clock.c 中定义），每产生一次系统时基中断（即 SysTick 中断）时，rt_tick 计数加一。假设线程要延时 10 个 tick，即 init_tick 等于 10 ，此时 rt_tick 等于 2，那么 timeout_tick 就等于 10 加 2 等于 12，当 rt_tick 递增到 12 的时候，线程延时到期，这个就是 timeout_tick 的实际含义。

13.1.4 在线程控制块中内置定时器

每个线程都会内置一个定时器，具体是在线程控制块中添加一个定时器成员，具体实现见代码清单 11-4 的高亮部分。

列表 4: 代码清单 11-4 在线程控制块中内置定时器

```
struct rt_thread
{
    /* rt 对象 */
    char        name[RT_NAME_MAX];     /* 对象的名字 */
```

(下页继续)

(续上页)

```
    rt_uint8_t  type;                   /* 对象类型 */
    rt_uint8_t  flags;                  /* 对象的状态 */
    rt_list_t   list;                   /* 对象的列表节点 */

    rt_list_t   tlist;                  /* 线程链表节点 */

    void        *sp;                        /* 线程栈指针 */
    void        *entry;                     /* 线程入口地址 */
    void        *parameter;             /* 线程形参 */
    void        *stack_addr;            /* 线程起始地址 */
    rt_uint32_t stack_size;             /* 线程栈大小，单位为字节 */

    rt_ubase_t  remaining_tick;         /* 用于实现阻塞延时 */

    rt_uint8_t  current_priority;       /* 当前优先级 */
    rt_uint8_t  init_priority;          /* 初始优先级 */
    rt_uint32_t number_mask;            /* 当前优先级掩码 */

    rt_err_t    error;                  /* 错误码 */
    rt_uint8_t  stat;                   /* 线程的状态 */

    struct rt_timer thread_timer;       /* 内置的线程定时器 */
};
```

13.1.5 定时器初始化函数

定时器初始化函数 rt_timer_init 在 timer.c 中定义，具体实现见*代码清单 11-5*。

列表 5: 代码清单 11-5 rt_timer_init 函数

```
/**
* 该函数用于初始化一个定时器，通常该函数用于初始化一个静态的定时器
*
* @param timer 静态定时器对象
* @param name 定时器的名字
* @param timeout 超时函数
* @param parameter 超时函数形参
* @param time 定时器的超时时间
* @param flag 定时器的标志
*/
void rt_timer_init(rt_timer_t  timer,
                   const char *name,
                   void (*timeout)(void *parameter),
                   void       *parameter,
                   rt_tick_t   time,
                   rt_uint8_t  flag)
{
    /* 定时器对象初始化 */
    rt_object_init((rt_object_t)timer, RT_Object_Class_Timer, name); (1)

    /* 定时器初始化 */
    _rt_timer_init(timer, timeout, parameter, time, flag);           (2)
}
```

代码清单 11-5(**1**)：定时器对象初始化，即将定时器插入到系统对象容器列表。有关对象相关的知识点请参考“对象容器的实现”章节。

代码清单 11-5(**2**)：定时器初始化函数 rt_timer_init 将定时器具体的初始化由封装在了一个内部函数 _rt_timer_init（函数开头的“_rt”表示该函数是一个内部函数）中，该函数在 timer.c 中定义，具体实现见*代码清单 11-6*。

列表 6: 代码清单 11-6 _rt_timer_init 函数

```
static void _rt_timer_init(rt_timer_t timer,                          (1)
                           void (*timeout)(void *parameter),          (2)
                           void      *parameter,                      (3)
                           rt_tick_t  time,                           (4)
                           rt_uint8_t flag)                           (5)
{
    int i;

    /* 设置标志 */
    timer->parent.flag  = flag;                            (6)

    /* 先设置为非激活态 */
    timer->parent.flag &= ~RT_TIMER_FLAG_ACTIVATED;        (7)

    timer->timeout_func = timeout;                         (8)
    timer->parameter    = parameter;                       (9)

    /* 初始化 定时器实际超时时的系统节拍数 */
    timer->timeout_tick = 0;                               (10)
    /* 初始化 定时器需要超时的节拍数 */
    timer->init_tick    = time;                            (11)

    /* 初始化定时器的内置节点 */
    for (i = 0; i < RT_TIMER_SKIP_LIST_LEVEL; i++)         (12)
    {
        rt_list_init(&(timer->row[i]));
    }
}
```

代码清单 11-6(1)：定时器控制块指针。

代码清单 11-6(2)：定时器超时函数。

代码清单 11-6(3)：定时器超时函数形参。

代码清单 11-6(4)：定时器实际需要延时的时间。

代码清单 11-6(5)：设置定时器的标志，取值在 rtdef.h 中定义，具体见代码清单 11-7。

列表 7: 代码清单 11-7 定时器状态宏定义

```
#define RT_TIMER_FLAG_DEACTIVATED        0x0      /* 定时器没有激活 */
#define RT_TIMER_FLAG_ACTIVATED          0x1      /* 定时器已经激活 */
#define RT_TIMER_FLAG_ONE_SHOT           0x0      /* 单次定时 */
#define RT_TIMER_FLAG_PERIODIC           0x2      /* 周期定时 */

#define RT_TIMER_FLAG_HARD_TIMER         0x0      /* 硬件定时器，定时器回调函数在tick isr 中调用 */

#define RT_TIMER_FLAG_SOFT_TIMER         0x4      /* 软件定时器，定时器回调函数在定时器线程中调用 */
```

代码清单 11-6(6)：设置标志。

代码清单 11-6(7)：初始时设置为非激活态。

代码清单 11-6(8)：设置超时函数，超时函数接下来会讲。

代码清单 11-6(9)：定时器超时函数形参。

代码清单 11-6(10)：初始化定时器实际超时时的系统节拍数。

代码清单 11-6(11)：初始化定时器需要超时的节拍数。

代码清单 11-6(12)：初始化定时器的内置节点，即将节点的 next 和 prev 这两个指针指向节点本身。当启动定时器的时候，定时器就通过该节点将自身插入到系统定时器列表 rt_timer_list 中。

13.1.6 定时器删除函数

定时器删除函数 _rt_timer_remove 在 timer.c 中定义，实现算法是将定时器自身的节点从系统定时器列表 rt_timer_list 脱离即可，具体实现见*代码清单 11-8*。

列表 8: 代码清单 11-8 _rt_timer_remove 函数定义

```
rt_inline void _rt_timer_remove(rt_timer_t timer)
{
    int i;

    for (i = 0; i < RT_TIMER_SKIP_LIST_LEVEL; i++)
    {
        rt_list_remove(&timer->row[i]);
    }
}
```

13.1.7 定时器停止函数

定时器停止函数 rt_timer_stop 在 timer.c 中定义，实现的算法也很简单，主要分成两步，先将定时器从系统定时器列表删除，然后改变定时器的状态为非 active 即可，具体代码实现见*代码清单 11-9*。

列表 9: 代码清单 11-9 rt_timer_stop 函数定义

```
/**
* 该函数将停止一个定时器
*
* @param timer 将要被停止的定时器
*
* @return 操作状态, RT_EOK on OK, -RT_ERROR on error
*/
```

(下页继续)

(续上页)

```
rt_err_t rt_timer_stop(rt_timer_t timer)
{
    register rt_base_t level;

    /* 只有 active 的定时器才能被停止，否则退出返回错误码 */
    if (!(timer->parent.flag & RT_TIMER_FLAG_ACTIVATED))
        return -RT_ERROR;

    /* 关中断 */
    level = rt_hw_interrupt_disable();

    /* 将定时器从定时器列表删除 */
    _rt_timer_remove(timer);

    /* 开中断 */
    rt_hw_interrupt_enable(level);

    /* 改变定时器的状态为非 active */
    timer->parent.flag &= ~RT_TIMER_FLAG_ACTIVATED;

    return RT_EOK;
}
```

13.1.8 定时器控制函数

定时器控制函数 rt_timer_control 在 timer.c 中定义，具体实现算法是根据不同的形参来设置定时器的状态和初始时间值，具体代码实现见*代码清单 11-10*。

列表 10: 代码清单 11-10 rt_timer_control 函数定义

```
/**
* 该函数将获取或者设置定时器的一些选项
*
* @param timer 将要被设置或者获取的定时器
* @param cmd 控制命令
* @param arg 形参
*
* @return RT_EOK
*/                                          (1)      (2)       (3)
rt_err_t rt_timer_control(rt_timer_t timer, int cmd, void *arg)
{
    switch (cmd)
    {
    case RT_TIMER_CTRL_GET_TIME:                                  (4)
        *(rt_tick_t *)arg = timer->init_tick;
        break;

    case RT_TIMER_CTRL_SET_TIME:                                  (5)
        timer->init_tick = *(rt_tick_t *)arg;
        break;

    case RT_TIMER_CTRL_SET_ONESHOT:
        timer->parent.flag &= ~RT_TIMER_FLAG_PERIODIC;            (6)
        break;

    case RT_TIMER_CTRL_SET_PERIODIC:
        timer->parent.flag |= RT_TIMER_FLAG_PERIODIC;             (7)
        break;
    }

```

(下页继续)

(续上页)

```
    return RT_EOK;
}
```

代码清单 11-10 (1)：timer 表示要控制的定时器。

代码清单 11-10 (2)：cmd 表示控制命令，取值在 rtdef.h 中定义，具体见代码清单 11-11。

列表 11: 代码清单 11-11 定时器控制命令宏定义

```
#define RT_TIMER_CTRL_SET_TIME          0x0     /* 设置定时器定时时间 */
#define RT_TIMER_CTRL_GET_TIME          0x1     /* 获取定时器定时时间 */
#define RT_TIMER_CTRL_SET_ONESHOT       0x2     /* 修改定时器为一次定时 */
#define RT_TIMER_CTRL_SET_PERIODIC      0x3     /* 修改定时器为周期定时 */
```

代码清单 11-10 (3)：控制定时器的形参，参数取值的含义根据第二个形参 cmd 来决定。

代码清单 11-10 (4)：获取定时器延时的初始时间。

代码清单 11-10 (5)：重置定时器的延时时间。

代码清单 11-10 (6)：设置定时器为一次延时，即延时到期之后定时器就停止了。

代码清单 11-10 (7)：设置定时器为周期延时，即延时到期之后又重新启动定时器。

13.1.9 定时器启动函数

定时器启动函数 rt_timer_start 在 timer.c 中定义，核心实现算法是将定时器按照延时时间做升序排列插入到系统定时器列表 rt_timer_list 中，具体代码实现见代码清单 11-12。

列表 12: 代码清单 11-12 rt_timer_start 函数定义

```
/**
* 启动定时器
*
* @param timer 将要启动的定时器
```

(下页继续)

(续上页)

```
*
* @return 操作状态, RT_EOK on OK, -RT_ERROR on error
*/
rt_err_t rt_timer_start(rt_timer_t timer)
{
    unsigned int row_lvl = 0;
    rt_list_t *timer_list;
    register rt_base_t level;
    rt_list_t *row_head[RT_TIMER_SKIP_LIST_LEVEL];
    unsigned int tst_nr;
    static unsigned int random_nr;


    /* 关中断 */
    level = rt_hw_interrupt_disable();                          (1)

    /* 将定时器从系统定时器列表移除 */
    _rt_timer_remove(timer);

    /* 改变定时器的状态为非 active */
    timer->parent.flag &= ~RT_TIMER_FLAG_ACTIVATED;

    /* 开中断 */
    rt_hw_interrupt_enable(level);

    /* 获取 timeout tick,
    最大的 timeout tick 不能大于 RT_TICK_MAX/2 */
    timer->timeout_tick = rt_tick_get() + timer->init_tick;      (2)

    /* 关中断 */
    level = rt_hw_interrupt_disable();

```

(下页继续)

(续上页)

```
    /* 将定时器插入到定时器列表 */
    /* 获取系统定时器列表根节点地址，rt_timer_list 是一个全局变量 */
    timer_list = rt_timer_list;                                    (3)


    /* 获取系统定时器列表第一条链表根节点地址 */
    row_head[0]  = &timer_list[0];                                 (4)

    /* 因为 RT_TIMER_SKIP_LIST_LEVEL 等于 1，这个循环只会执行一次 */
    for (row_lvl = 0; row_lvl < RT_TIMER_SKIP_LIST_LEVEL; row_lvl++)   (5)
    {
        /* 列表不为空，当没有定时器被插入到系统定时器列表时，该循环不执行 */ (6)
        for (; row_head[row_lvl] != timer_list[row_lvl].prev; row_head[row_
↪lvl]  = row_head[row_lvl]->next)
        {
            struct rt_timer *t;

            /* 获取定时器列表节点地址 */
            rt_list_t *p = row_head[row_lvl]->next;                (6)-1

            /* 根据节点地址获取父结构的指针 */                        (6)-2
            t = rt_list_entry(p,                    /* 节点地址 */
                              struct rt_timer,    /* 节点所在父结构的数据类型 */
                              row[row_lvl]);      /* 节点在父结构中叫什么，即名字 */

            /* 两个定时器的超时时间相同，则继续在定时器列表中寻找下一个节点 */
            if ((t->timeout_tick - timer->timeout_tick) == 0)      (6)-3
            {
                continue;
            }
            /* 两个定时器的超时时间相同，则继续在定时器列表中寻找下一个节点 */
```

(下页继续)

(续上页)

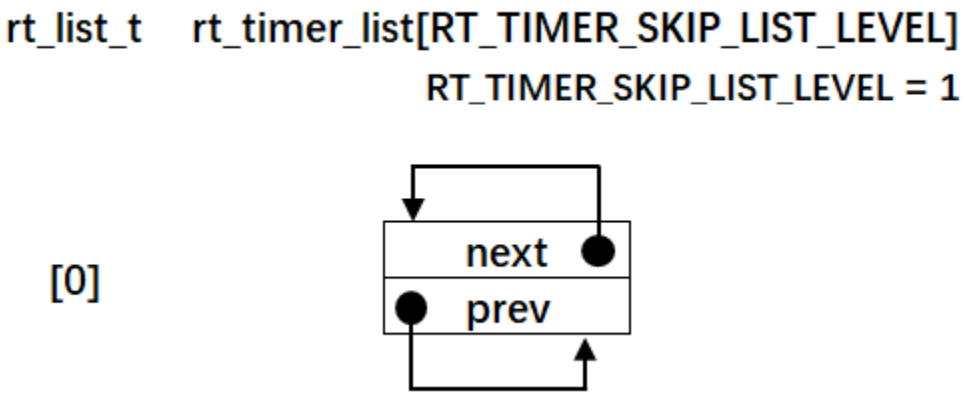
```
            else if ((t->timeout_tick - timer->timeout_tick) < RT_TICK_MAX / 2)
            {
                break;
            }

        }
        /* 条件不会成真，不会被执行 */
        if (row_lvl != RT_TIMER_SKIP_LIST_LEVEL - 1)
        {
        row_head[row_lvl + 1] = row_head[row_lvl] + 1;
        }
    }

    /* random_nr 是一个静态变量，用于记录启动了多少个定时器 */
    random_nr++;
    tst_nr = random_nr;

    /* 将定时器插入到系统定时器列表 */                                (7)
    rt_list_insert_after(row_head[RT_TIMER_SKIP_LIST_LEVEL - 1],        /* 双向列表根节点地址 */
                         &(timer->row[RT_TIMER_SKIP_LIST_LEVEL - 1])); /* 要被插入的节点的地址 */

    /* RT_TIMER_SKIP_LIST_LEVEL 等于 1，该 for 循环永远不会执行 */
    for (row_lvl = 2; row_lvl <= RT_TIMER_SKIP_LIST_LEVEL; row_lvl++)
    {
        if (!(tst_nr & RT_TIMER_SKIP_LIST_MASK))
            rt_list_insert_after(row_head[RT_TIMER_SKIP_LIST_LEVEL - row_lvl],
                                 &(timer->row[RT_TIMER_SKIP_LIST_LEVEL - row_lvl]));
```

(下页继续)

(续上页)

```
        else
            break;

        tst_nr >>= (RT_TIMER_SKIP_LIST_MASK + 1) >> 1;
    }

    /* 设置定时器标志位为激活态 */
    timer->parent.flag |= RT_TIMER_FLAG_ACTIVATED;                (8)

    /* 开中断 */
    rt_hw_interrupt_enable(level);

    return -RT_EOK;
}
```

在阅读代码清单 11-12 的内容时，配套一个初始化好的空的系统定时器列表示意图会更好理解，该图具体见。



代码清单 11-12**(1)**：关中断，进入临界段，启动定时器之前先将定时器从系统定时器列表删除，状态改为非 active。

代码清单 11-12**(2)**：计算定时器超时结束时的系统时基节拍计数器的值，当系统时基节拍计数器

rt_tick 的值等于 timeout_tick 时，表示定时器延时到期。在 RT-Thread 中，timeout_tick 的值要求不能大于 RT_TICK_MAX/2，RT_TICK_MAX 是在 rtdef.h 中定义的宏，具体为 32 位整形的最大值 0xffffffff。

代码清单 11-12(**3**)：获取系统定时器列表 rt_timer_list 的根节点地址，rt_timer_list 是一个全局变量。

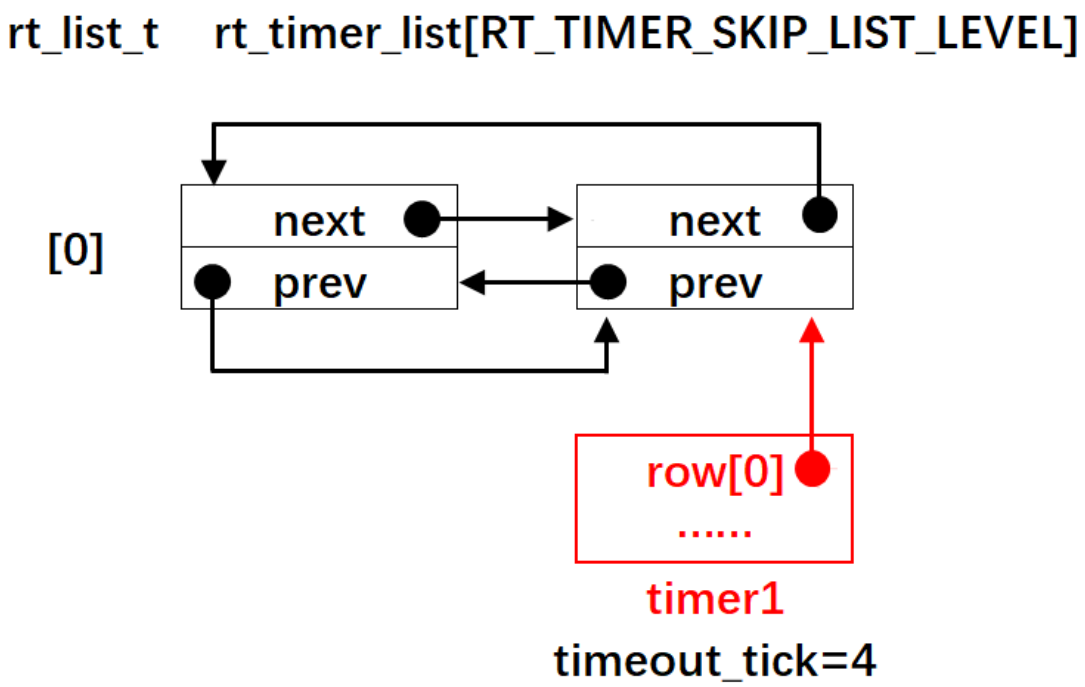
代码清单 11-12(**4**)：获取系统定时器列表第一条链表根节点地址。

代码清单 11-12(**5**)：因为 RT_TIMER_SKIP_LIST_LEVEL 等于 1，这个 for 循环只会执行一次，即只有一条定时器双向链表。首先 row_lvl 等于 0，因为 RT_TIMER_SKIP_LIST_LEVEL 等于 1，所以 row_lvl < RT_TIMER_SKIP_LIST_LEVEL 条件成立，for 循环体会被执行，当执行完 for 函数体时，执行 row_lvl++ 变成 1，再执行判断 row_lvl < RT_TIMER_SKIP_LIST_LEVEL，此时两者相等，条件不成立，则跳出 for 循环，只执行一次。

代码清单 11-12(**6**)：当系统定时器列表 rt_timer_list 为空时，该循环体不执行。rt_timer_list 为空是什么样，具体见图 *11-2*，用代码表示就是 row_head[row_lvl] = timer_list[row_lvl].prev （此时 row_lvl 等于 0）。现在我们假设有三个定时器需要插入到系统定时器列表 rt_timer_list，定时器 1 的 timeout_tick 等于 4，定时器 2 的 timeout_tick 等于 2，定时器 3 的 timeout_tick 等于 3，插入的顺序为定时器 1 先插入，然后是定时器 2，再然后是定时器 3。接下来我们看看这三个定时器是如何插入到系统定时器列表的。

13.1.9.1 插入定时器 1（timeout_tick=4）

当启动定时器 1 之前，系统定时器列表为空，*代码清单 11-12*(**6**) 跳过不执行，紧接着执行到代码清单 11-12(**7**)，定时器 1 作为第一个节点插入到系统定时器列表，示意图具体见图 *11-3*。



定时器 1 插入到系统定时器之后，会执行到*代码清单 11-12*(**8**) 将定时器的状态改变为非 active 态，至此，定时器 1 顺利完成插入。

13.1.9.2 插入定时器 2（timeout_tick=2）

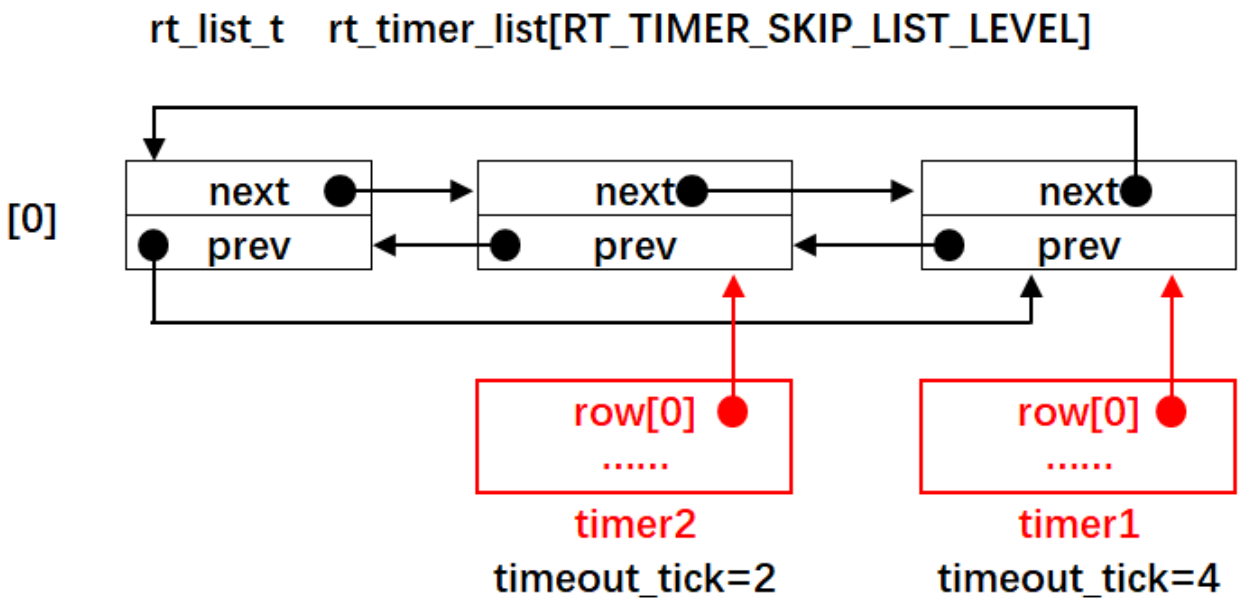
此时要插入定时器 2，定时器启动函数 rt_timer_start 会重新被调用，*代码清单 11-12*(**1**) ~(**5**) 的执行过程与定时器 1 插入时是一样的，有区别的是*代码清单 11-12*(**6**) 部分。此时系统定时器列表里面有定时器 1，所以不为空，该 for 循环体会被执行。

代码清单 11-12(**6**)-**1**：获取定时器列表节点地址，此时 p 的值等于定时器 1 里面 row[0] 的地址。

代码清单 11-12(**6**)-**2**：根据节点地址 p 获取父结构的指针，即根据 row[0] 的地址获取到 row[0] 所在定时器的地址，即定时器 1 的地址。

代码清单 11-12(**6**)-**3**：比较两个定时器的 timeout_tick 值，如果相等则继续与下一个节点的定时器比较。定时器 1 的 timeout_tick 等于 4，定时器 2 的 timeout_tick 等于 2，4 减 2 等于 2，小于 RT_TICK_MAX / 2，则跳出（break）当前的 for 循环，当前 for 循环里面的 row_head[row_lvl] = row_head[row_lvl]->next 语句不会被执行，即 row_head[row_lvl=0] 存的还是系统定时器列表

rt_timer_list 的根节点。然后执行代码清单 11-12**(7)**，将定时器 2 插入到系统定时器列表根节点的后面，即定时器 1 节点的前面，实现了按照 timeout_tick 的大小做升序排列，示意图具体见图 *11-4*。



13.1.9.3 插入定时器 3（timeout_tick=3）

此时要插入定时器 3，定时器启动函数 rt_timer_start 会重新被调用，*代码清单 11-12*(1) ~(5) 的执行过程与定时器 1 和 2 插入时是一样的，有区别的是*代码清单 11-12*(6) 部分。此时系统定时器列表里面有定时器 1 和定时器 2，所以不为空，该 for 循环体会被执行。

代码清单 11-12(6)-1：获取定时器列表节点地址，此时 p 的值等于定时器 2 里面 row[0] 的地址。

代码清单 11-12(6)-2：根据节点地址 p 获取父结构的指针，即根据 row[0] 的地址获取到 row[0] 所在定时器的地址，即定时器 2 的地址。

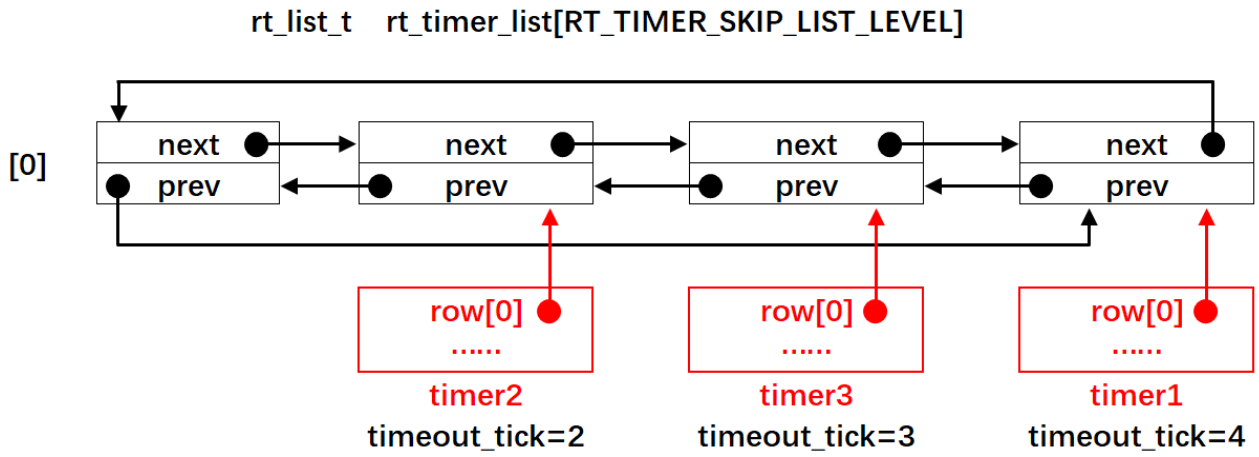
代码清单 11-12(6)-3：比较两个定时器的 timeout_tick 值，如果相等则继续与下一个节点的定时器比较。定时器 2 的 timeout_tick 等于 2，定时器 3 的 timeout_tick 等于 3，2 减 3 等于-1，-1 的补码为 0xfffffffe，大于 RT_TICK_MAX /2，表示定时器 3 应该插入到定时器 2 之后，但是定时器 2 之后还有节点，需要继续比较，则继续执行 for 循环：执行 row_head[row_lvl] = row_head[row_lvl]-

>next 语句，得到 row_head[row_lvl=0] 等于定时器 2 里面 row[0] 的地址，重新执行代码代码清单 11-12**(6)-1~3**：

*代码清单 11-12***(6)-1**：获取定时器列表节点地址，此时 p 的值等于定时器 1 里面 row[0] 的地址。

*代码清单 11-12***(6)-2**：根据节点地址 p 获取父结构的指针，即根据 row[0] 的地址获取到 row[0] 所在定时器的地址，即定时器 1 的地址。

*代码清单 11-12***(6)-3**：比较两个定时器的 timeout_tick 值，如果相等则继续与下一个节点的定时器比较。定时器 1 的 timeout_tick 等于 4，定时器 3 的 timeout_tick 等于 3，4 减 3 等于 1，1 小于 RT_TICK_MAX /2，则跳出当前的 for 循环，表示定时器 3 应该插入到定时器 1 之前，要插入的位置找到。然后执行*代码清单 11-12***(7)**，将定时器 3 插入到定时器 2 后面，实现了按照 timeout_tick 的大小做升序排列，示意图具体见图 *11-5*。



13.1.10 定时器扫描函数

定时器扫描函数 rt_timer_check 在 timer.c 中定义，用于扫描系统定时器列表，查询定时器的延时是否到期，如果到期则让对应的线程就绪，具体实现见*代码清单 11-13*。

列表 13: 代码清单 11-13rt_timer_check 函数定义

```
/**
* 该函数用于扫描系统定时器列表，当有超时事件发生时
* 就调用对应的超时函数
*
* @note 该函数在操作系统定时器中断中被调用
*/
void rt_timer_check(void)
{
    struct rt_timer *t;
    rt_tick_t current_tick;
    register rt_base_t level;

    /* 获取系统时基计数器 rt_tick 的值 */
    current_tick = rt_tick_get();                                   (1)

    /* 关中断 */
    level = rt_hw_interrupt_disable();                              (2)

    /* 系统定时器列表不为空，则扫描定时器列表 */                    (3)
    while (!rt_list_isempty(&rt_timer_list[RT_TIMER_SKIP_LIST_LEVEL - 1]))
    {
        /* 获取第一个节点定时器的地址 */                             (4)
        t = rt_list_entry
        (rt_timer_list[RT_TIMER_SKIP_LIST_LEVEL - 1].next,    /* 节点地址 */
        struct rt_timer,                                      /* 节点所在的父结构的数据类型 */
        row[RT_TIMER_SKIP_LIST_LEVEL - 1]);                   /* 节点在父结构的成员名 */

        if ((current_tick - t->timeout_tick) < RT_TICK_MAX / 2)     (5)
```

(下页继续)

(续上页)

```
        {
            /* 先将定时器从定时器列表移除 */
            _rt_timer_remove(t);                                    (6)

            /* 调用超时函数 */
            t->timeout_func(t->parameter);                          (7)

            /* 重新获取 rt_tick */
            current_tick = rt_tick_get();                           (8)

            /* 周期定时器 */                                        (9)
            if ((t->parent.flag & RT_TIMER_FLAG_PERIODIC) &&
                (t->parent.flag & RT_TIMER_FLAG_ACTIVATED))
            {
                /* 启动定时器 */
                t->parent.flag &= ~RT_TIMER_FLAG_ACTIVATED;
                rt_timer_start(t);
            }
            /* 单次定时器 */                                       (10)
            else
            {
                /* 停止定时器 */
                t->parent.flag &= ~RT_TIMER_FLAG_ACTIVATED;
            }
        }
        else
            break;                                                  (11)
    }

    /* 开中断 */
    rt_hw_interrupt_enable(level);                                  (12)
}
```

代码清单 11-13(1)：获取系统时基计数器 rt_tick 的值，rt_tick 是一个在 clock.c 中定义全局变量，用于记录系统启动至今经过了多少个 tick。

代码清单 11-13(2)：关中断，接下来扫描系统时基列表 rt_timer_list 的过程不能被中断。

代码清单 11-13(3)：系统定时器列表不为空，则扫描整个定时器列表，如果列表的第一个节点的定时器延时不到期，则退出，因为列表中的定时器节点是按照延时时间做升序排列的，第一个延时不到期，则后面的肯定不到期。

代码清单 11-13(4)：获取第一个节点定时器的地址。

代码清单 11-13(5)：定时器超时时间到。

代码清单 11-13(6)：将定时器从系统定时器列表 rt_timer_list 移除，表示延时时间到。

代码清单 11-13(7)：调用超时函数 rt_thread_timeout，将线程就绪。该函数在 thread.c 中定义，具体实现见代码清单 11-14。

列表 14: 代码清单 11-14 rt_thread_timeout 函数定义

```
/**
* 线程超时函数
* 当线程延时到期或者等待的资源可用或者超时时，该函数会被调用
*
* @param parameter 超时函数的形参
*/
void rt_thread_timeout(void *parameter)
{
    struct rt_thread *thread;

    thread = (struct rt_thread *)parameter;

    /* 设置错误码为超时 */                                  (1)
    thread->error = -RT_ETIMEOUT;

    /* 将线程从挂起列表中删除 */                            (2)
```

(下页继续)

(续上页)

```
    rt_list_remove(&(thread->tlist));

    /* 将线程插入到就绪列表 */                                    (3)
    rt_schedule_insert_thread(thread);

    /* 系统调度 */                                                (4)
    rt_schedule();
}
```

代码清单 11-14(1)：设置线程错误码为超时。

代码清单 11-14(2)：将线程从挂起列表中删除，前提是线程在等待某些资源而被挂起到挂起列表，如果只是延时到期，则这个只是空操作。

代码清单 11-14(3)：将线程就绪。

代码清单 11-14(4)：因为有新的线程就绪，需要执行系统调度。

代码清单 11-13(8)：重新获取系统时基计数器 rt_tick 的值。

代码清单 11-13(9)：如果定时器是周期定时器则重新启动定时器。

代码清单 11-13(10)：如果定时器为单次定时器则停止定时器。

代码清单 11-13(11)：第一个节点定时器延时没有到期，则跳出 while 循环，因为链表中的定时器节点是按照延时的时间做升序排列的，第一个定时器延时不到期，则后面的肯定不到期，不用再继续扫描。

代码清单 11-13(12)：系统定时器列表扫描完成，开中断。

13.2 修改代码，支持定时器

13.2.1 修改线程初始化函数

在线程初始化函数中，需要将自身内置的定时器初始化好，具体见代码清单 11-15 的高亮部分。

列表 15: 代码清单 11-15 修改线程初始化函数

```
rt_err_t rt_thread_init(struct rt_thread *thread,
                        const char       *name,
                        void (*entry)(void *parameter),
                        void             *parameter,
                        void             *stack_start,
                        rt_uint32_t       stack_size,
                        rt_uint8_t        priority)
{
    /* 线程对象初始化 */
    /* 线程结构体开头部分的成员就是 rt_object_t 类型 */
    rt_object_init((rt_object_t)thread, RT_Object_Class_Thread, name);
    rt_list_init(&(thread->tlist));

    thread->entry = (void *)entry;
    thread->parameter = parameter;

    thread->stack_addr = stack_start;
    thread->stack_size = stack_size;

    /* 初始化线程栈，并返回线程栈指针 */
    thread->sp = (void *)rt_hw_stack_init( thread->entry,
                                           thread->parameter,
                                           (void *)((char *)thread->stack_addr␣
↪+ thread->stack_size - 4) );

    thread->init_priority    = priority;
    thread->current_priority = priority;
    thread->number_mask = 0;

    /* 错误码和状态 */
```

(下页继续)

(续上页)

```
    thread->error = RT_EOK;
    thread->stat  = RT_THREAD_INIT;

    /* 初始化线程定时器 */
    rt_timer_init(&(thread->thread_timer),      /* 静态定时器对象 */
                  thread->name,                 /* 定时器的名字，直接使用的是线程的名字 */
                  rt_thread_timeout,            /* 超时函数 */
                  thread,                       /* 超时函数形参 */
                  0,                            /* 延时时间 */
                  RT_TIMER_FLAG_ONE_SHOT);      /* 定时器的标志 */

    return RT_EOK;
}
```

13.2.2 修改线程延时函数

线程延时函数 rt_thread_delay 具体修改见*代码清单 11-16* 的高亮部分，整个函数的实体由 rt_thread_sleep 代替。

列表 16: 代码清单 11-16 修改线程延时函数

```
#if 0
void rt_thread_delay(rt_tick_t tick)
{
    register rt_base_t temp;
    struct rt_thread *thread;

    /* 失能中断 */
    temp = rt_hw_interrupt_disable();

```

(下页继续)

(续上页)

```
    thread = rt_current_thread;
    thread->remaining_tick = tick;

    /* 改变线程状态 */
    thread->stat = RT_THREAD_SUSPEND;
    rt_thread_ready_priority_group &= ~thread->number_mask;

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

    /* 进行系统调度 */
    rt_schedule();
}
#else
rt_err_t rt_thread_delay(rt_tick_t tick)
{
    return rt_thread_sleep(tick);                    (1)
}
#endif
```

代码清单 11-16**(1)**：rt_thread_sleep 函数在 thread.c 定义，具体实现见代码清单 11-17。

列表 17: 代码清单 11-17 rt_thread_sleep 函数定义

```
/**
* 该函数将让当前线程睡眠一段时间，单位为 tick
*
* @param tick 睡眠时间，单位为 tick
*
* @return RT_EOK
*/
rt_err_t rt_thread_sleep(rt_tick_t tick)
```

(下页继续)

(续上页)

```
{
    register rt_base_t temp;
    struct rt_thread *thread;

    /* 关中断 */
    temp = rt_hw_interrupt_disable();                                    (1)

    /* 获取当前线程的线程控制块 */
    thread = rt_current_thread;                                          (2)

    /* 挂起线程 */
    rt_thread_suspend(thread);                                           (3)

    /* 设置线程定时器的超时时间 */
    rt_timer_control(&(thread->thread_timer), RT_TIMER_CTRL_SET_TIME, &
↪tick);    (4)

    /* 启动定时器 */
    rt_timer_start(&(thread->thread_timer));                             (5)

    /* 开中断 */
    rt_hw_interrupt_enable(temp);                                        (6)

    /* 执行系统调度 */
    rt_schedule();                                                       (7)

    return RT_EOK;
}
```

代码清单 *11-17*(**1**)：关中断。

代码清单 *11-17*(**2**)：获取当前线程的线程控制块，rt_current_thread 是一个全局的线程控制块指针，用于指向当前正在运行的线程控制块。

代码清单 11-17(3)：在启动定时器之前，先把线程挂起来，线程挂起函数 rt_thread_suspend 在 thread.c 实现，具体实现见代码清单 11-18。

列表 18: 代码清单 11-18 rt_thread_suspend 函数定义

```
/**
* 该函数用于挂起指定的线程
* @param thread 要被挂起的线程
*
* @return 操作状态, RT_EOK on OK, -RT_ERROR on error
*
* @note 如果挂起的是线程自身，在调用该函数后，
* 必须调用 rt_schedule() 进行系统调度
*
*/
rt_err_t rt_thread_suspend(rt_thread_t thread)
{
    register rt_base_t temp;


    /* 只有就绪的线程才能被挂起，否则退出返回错误码 */    (1)
    if ((thread->stat & RT_THREAD_STAT_MASK) != RT_THREAD_READY)
    {
        return -RT_ERROR;
    }

    /* 关中断 */
    temp = rt_hw_interrupt_disable();                    (2)

    /* 改变线程状态 */
    thread->stat = RT_THREAD_SUSPEND;                    (3)
    /* 将线程从就绪列表删除 */
    rt_schedule_remove_thread(thread);                   (4)
```

(下页继续)

(续上页)

```

    /* 停止线程定时器 */
    rt_timer_stop(&(thread->thread_timer));                (5)

    /* 开中断 */
    rt_hw_interrupt_enable(temp);                          (6)

    return RT_EOK;
}
```

代码清单 *11-18*(**1**)：只有就绪的线程才能被挂起，否则退出返回错误码。

代码清单 *11-18*(**2**)：关中断。

代码清单 *11-18*(**3**)：将线程的状态改为挂起态。

代码清单 *11-18*(**4**)：将线程从就绪列表删除，这里面包含了两个动作，一是将线程从线程优先级表里面删除，二是将线程在线程就绪优先级组中对应的位清零。

代码清单 *11-18*(**5**)：停止定时器。

代码清单 *11-18*(**6**)：开中断。

代码清单 *11-17*(**4**)：设置定时器的超时时间。

代码清单 *11-17*(**5**)：启动定时器。

代码清单 *11-17*(**6**)：开中断。

代码清单 *11-17*(**7**)：执行系统调度，因为当前线程要进入延时，接下来需要寻找就绪线程中优先级最高的线程来执行。

13.2.3 修改系统时基更新函数

系统时基更新函数 rt_thread_delay 具体修改见*代码清单 11-19* 的高亮部分，整个函数的实体由 rt_timer_check() 代替。

列表 19: 代码清单 11-19 修改系统时基更新函数

```
#if 0
void rt_tick_increase(void)
{
    rt_ubase_t i;
    struct rt_thread *thread;
    rt_tick ++;

    /* 扫描就绪列表中所有线程的 remaining_tick，如果不为 0，则减 1 */
    for(i=0; i<RT_THREAD_PRIORITY_MAX; i++)
    {
        thread = rt_list_entry( rt_thread_priority_table[i].next,
                                struct rt_thread,
                                tlist);
        if(thread->remaining_tick > 0)
        {
            thread->remaining_tick --;
            if(thread->remaining_tick == 0)
            {
                //rt_schedule_insert_thread(thread);
                rt_thread_ready_priority_group |= thread->number_mask;
            }
        }
    }

    /* 任务调度 */
    rt_schedule();
```

(下页继续)

(续上页)

```
}

#else
void rt_tick_increase(void)
{
    /* 系统时基计数器加 1 操作,rt_tick 是一个全局变量 */
    ++ rt_tick;                                    (1)

    /* 扫描系统定时器列表 */
    rt_timer_check();                              (2)
}
#endif
```

代码清单 11-18(**1**)：系统时基计数器加 1 操作，rt_tick 是一个在 clock.c 中定义的全局变量，用于记录系统启动至今经过了多少个 tick。

代码清单 11-18(**2**)：扫描系统定时器列表 rt_timer_list，检查是否有定时器延时到期，如果有则将定时器从系统定时器列表删除，并将对应的线程就绪，然后执行系统调度。

13.2.4 修改 main.c 文件

为了演示定时器的插入，我们新增加了一个线程 3，在启动调度器初始化前，我们新增了定时器初始化 rt_system_timer_init()，这两个改动具体见的代码清单 11-20 高亮部分。

列表 20: 代码清单 11-20 main.c 文件内容

```
/*
************************************************************************
*                                包含的头文件
************************************************************************
*/

```

(下页继续)

(续上页)

```
#include <rtthread.h>
#include <rthw.h>
#include "ARMCM7.h"


/*
*************************************************************************
*                                 全局变量
*************************************************************************
*/
rt_uint8_t flag1;
rt_uint8_t flag2;
rt_uint8_t flag3;

extern rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX];

/*
*************************************************************************
*                         线程控制块 & STACK & 线程声明
*************************************************************************
*/


/* 定义线程控制块 */
struct rt_thread rt_flag1_thread;
struct rt_thread rt_flag2_thread;
struct rt_thread rt_flag3_thread;

ALIGN(RT_ALIGN_SIZE)
/* 定义线程栈 */
rt_uint8_t rt_flag1_thread_stack[512];
rt_uint8_t rt_flag2_thread_stack[512];
```

(下页继续)

(续上页)

```
rt_uint8_t rt_flag3_thread_stack[512];

/* 线程声明 */
void flag1_thread_entry(void *p_arg);
void flag2_thread_entry(void *p_arg);
void flag3_thread_entry(void *p_arg);

/*
*************************************************************************
*                                 函数声明
*************************************************************************
*/
void delay(uint32_t count);

/*************************************************************************
* @brief  main 函数
* @param  无
* @retval 无
*
* @attention
*************************************************************************
*/
int main(void)
{
    /* 硬件初始化 */
    /* 将硬件相关的初始化放在这里，如果是软件仿真则没有相关初始化代码 */

    /* 关中断 */
    rt_hw_interrupt_disable();

    /* SysTick 中断频率设置 */
    SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );
```

(下页继续)

(续上页)

```
    /* 系统定时器列表初始化 */
    rt_system_timer_init();

    /* 调度器初始化 */
    rt_system_scheduler_init();

    /* 初始化空闲线程 */
    rt_thread_idle_init();

    /* 初始化线程 */
    rt_thread_init( &rt_flag1_thread,                    /* 线程控制块 */
                    "rt_flag1_thread",                   /* 线程名字，字符串形式␣
↪*/
                    flag1_thread_entry,                  /* 线程入口地址 */
                    RT_NULL,                             /* 线程形参 */
                    &rt_flag1_thread_stack[0],           /* 线程栈起始地址 */
                    sizeof(rt_flag1_thread_stack),       /* 线程栈大小，单位为字
节 */
                    2);                                  /* 优先级 */
    /* 将线程插入到就绪列表 */
    rt_thread_startup(&rt_flag1_thread);

    /* 初始化线程 */
    rt_thread_init( &rt_flag2_thread,                    /* 线程控制块 */
                    "rt_flag2_thread",                   /* 线程名字，字符串形式␣
↪*/
                    flag2_thread_entry,                  /* 线程入口地址 */
                    RT_NULL,                             /* 线程形参 */
                    &rt_flag2_thread_stack[0],           /* 线程栈起始地址 */
                    sizeof(rt_flag2_thread_stack),       /* 线程栈大小，单位为字
节 */
```

(下页继续)

(续上页)

```
                    3);                                 /* 优先级 */
    /* 将线程插入到就绪列表 */
    rt_thread_startup(&rt_flag2_thread);

    /* 初始化线程 */
    rt_thread_init( &rt_flag3_thread,                   /* 线程控制块 */
                    "rt_flag3_thread",                  /* 线程名字，字符串形式␣
↪*/
                    flag3_thread_entry,                 /* 线程入口地址 */
                    RT_NULL,                            /* 线程形参 */
                    &rt_flag3_thread_stack[0],          /* 线程栈起始地址 */
                    sizeof(rt_flag3_thread_stack),      /* 线程栈大小，单位为字
节 */
                    4);                                 /* 优先级 */
    /* 将线程插入到就绪列表 */
    rt_thread_startup(&rt_flag3_thread);

    /* 启动系统调度器 */
    rt_system_scheduler_start();
}

/*
*********************************************************************
*                              函数实现
*********************************************************************
*/
/* 软件延时 */
void delay (uint32_t count)
{
    for(; count!=0; count--);
}

```

(下页继续)

(续上页)

```
/* 线程 1 */
void flag1_thread_entry( void *p_arg )
{
    for( ;; )
    {
        flag1 = 1;
        rt_thread_delay(4);
        flag1 = 0;
        rt_thread_delay(4);
    }
}

/* 线程 2 */
void flag2_thread_entry( void *p_arg )
{
    for( ;; )
    {
        flag2 = 1;
        rt_thread_delay(2);
        flag2 = 0;
        rt_thread_delay(2);
    }
}

/* 线程 3 */
void flag3_thread_entry( void *p_arg )
{
    for( ;; )
    {
        flag3 = 1;
        rt_thread_delay(3);
        flag3 = 0;
```

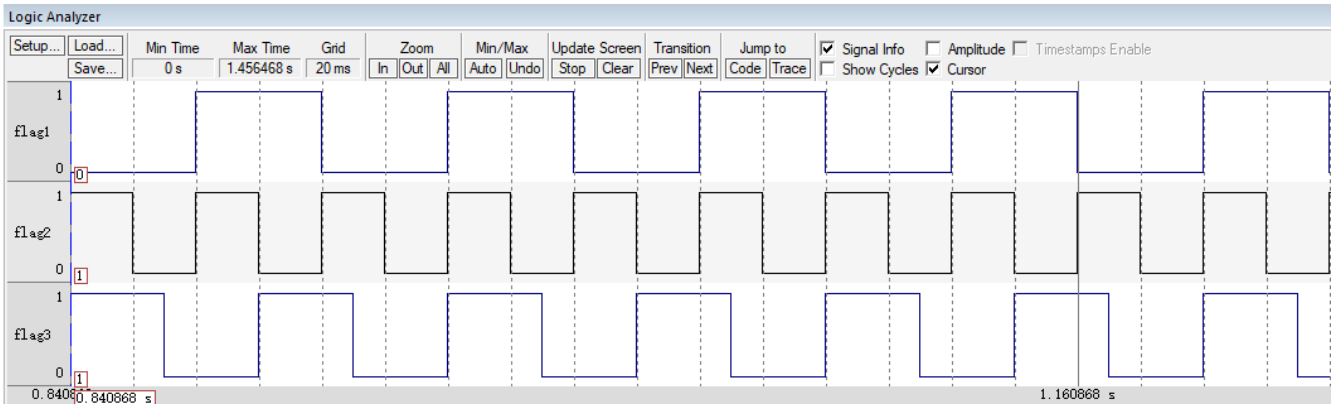
(下页继续)

(续上页)

```
        rt_thread_delay(3);
    }
}


void SysTick_Handler(void)
{
    /* 进入中断 */
    rt_interrupt_enter();

    /* 更新时基 */
    rt_tick_increase();

    /* 离开中断 */
    rt_interrupt_leave();
}
```

13.3 实验现象

进入软件调试，全速运行程序，逻辑分析仪中的仿真波形图具体见图 *11-6*。



从图 *11-6* 中可以看出线程 1、线程 2 和线程 3 的高低电平的延时时间分别为 4、2 和 3 个 tick，与

代码控制的完全一致，说明我们的定时器起作用了，搞定。

第 14 章　支持时间片

在 RT-Thread 中，当同一个优先级下有两个或两个以上线程的时候，线程支持时间片功能，即我们可以指定线程持续运行一次的时间，单位为 tick。假如有两个线程分别为线程 2 和线程 3，他们的优先级都为 3，线程 2 的时间片为 2，线程 3 的时间片为 3。当执行到优先级为 3 的线程时，会先执行线程 2，直到线程 2 的时间片耗完，然后再执行线程 3，具体的实验波形图看本章最后的实验现象即可。

14.1 实现时间片

14.1.1 在线程控制块中添加时间片相关成员

在线程控制块中添加时间片相关的成员，init_tick 表示初始时间片，remaining_tick 表示还剩下多少时间片，具体见*代码清单 12-1* 的高亮部分。

列表 1: 代码清单 12-1 在线程控制块中添加时间片相关成员

```
struct rt_thread
{
    /* rt 对象 */
    char        name[RT_NAME_MAX];     /* 对象的名字 */
    rt_uint8_t  type;                  /* 对象类型 */
    rt_uint8_t  flags;                 /* 对象的状态 */
    rt_list_t   list;                  /* 对象的列表节点 */

    rt_list_t   tlist;                 /* 线程链表节点 */
    void        *sp;                       /* 线程栈指针 */
```

(下页继续)

(续上页)

```
    void        *entry;                  /* 线程入口地址 */
    void        *parameter;          /* 线程形参 */
    void        *stack_addr;         /* 线程起始地址 */
    rt_uint32_t stack_size;          /* 线程栈大小，单位为字节 */

    rt_ubase_t  init_tick;           /* 初始时间片 */
    rt_ubase_t  remaining_tick;      /* 剩余时间片 */

    rt_uint8_t  current_priority;    /* 当前优先级 */
    rt_uint8_t  init_priority;       /* 初始优先级 */
    rt_uint32_t number_mask;         /* 当前优先级掩码 */

    rt_err_t    error;               /* 错误码 */
    rt_uint8_t  stat;                /* 线程的状态 */

    struct rt_timer thread_timer;    /* 内置的线程定时器 */
};
```

14.1.2 修改线程初始化函数

在线程初始化函数 rt_thread_init 中添加时间片相关形参，并将线程控制块中与时间片相关的成员初始化好，具体见*代码清单 12-2* 的高亮部分。

列表 2: 代码清单 12-2 修改线程初始化函数

```
rt_err_t rt_thread_init(struct rt_thread *thread,
                        const char       *name,
                        void (*entry)(void *parameter),
                        void             *parameter,
                        void             *stack_start,
                        rt_uint32_t       stack_size,
```

(下页继续)

(续上页)

```
                        rt_uint8_t        priority,
                        rt_uint32_t       tick)
{
    /* 线程对象初始化 */
    /* 线程结构体开头部分的成员就是 rt_object_t 类型 */
    rt_object_init((rt_object_t)thread, RT_Object_Class_Thread, name);
    rt_list_init(&(thread->tlist));

    thread->entry = (void *)entry;
    thread->parameter = parameter;

    thread->stack_addr = stack_start;
    thread->stack_size = stack_size;

    /* 初始化线程栈，并返回线程栈指针 */
    thread->sp = (void *)rt_hw_stack_init( thread->entry,
                                           thread->parameter,
                                           (void *)((char *)thread->stack_addr␣
↪+ thread->stack_size - 4) );

    thread->init_priority    = priority;
    thread->current_priority = priority;
    thread->number_mask = 0;

    /* 错误码和状态 */
    thread->error = RT_EOK;
    thread->stat  = RT_THREAD_INIT;

    /* 时间片相关 */
    thread->init_tick      = tick;
    thread->remaining_tick = tick;

```

(下页继续)

(续上页)

```
    /* 初始化线程定时器 */
    rt_timer_init(&(thread->thread_timer),      /* 静态定时器对象 */
                  thread->name,                 /* 定时器的名字，直接使用的是线程的名字 */
                  rt_thread_timeout,            /* 超时函数 */
                  thread,                       /* 超时函数形参 */
                  0,                            /* 延时时间 */
                  RT_TIMER_FLAG_ONE_SHOT);      /* 定时器的标志 */

    return RT_EOK;
}
```

14.1.2.1 修改空闲线程初始化函数

在空闲线。程初始化函数中指定空闲线程的时间片，通常很少线程的优先级会与空闲线程的优先级一样，时间片我们可以随便设置，这里我们示意性的设置为 2，具体见*代码清单 12-3* 的高亮部分。

列表 3: 代码清单 12-3 修改空闲线程初始化函数

```
void rt_thread_idle_init(void)
{

    /* 初始化线程 */
    rt_thread_init(&idle,
                   "idle",
                   rt_thread_idle_entry,
                   RT_NULL,
                   &rt_thread_stack[0],
                   sizeof(rt_thread_stack),
                   RT_THREAD_PRIORITY_MAX-1,
```

(下页继续)

(续上页)

```
                2);/* 时间片 */

    /* 启动空闲线程 */
    rt_thread_startup(&idle);
}
```

14.1.3 修改系统时基更新函数

在系统时基更新函数中添加与时间片相关的代码，具体见代码清单 12-4 的高亮部分。

列表 4: 代码清单 12-4 修改系统时基更新函数

```
void rt_tick_increase(void)
{
    struct rt_thread *thread;

    /* 系统时基计数器加 1 操作,rt_tick 是一个全局变量 */
    ++ rt_tick;

    /* 获取当前线程线程控制块 */
    thread = rt_thread_self();                          (1)

    /* 时间片递减 */
    -- thread->remaining_tick;                          (2)

    /* 如果时间片用完，则重置时间片，然后让出处理器 */
    if (thread->remaining_tick == 0)                    (3)
    {
        /* 重置时间片 */
        thread->remaining_tick = thread->init_tick;     (4)

```

(下页继续)

(续上页)

```
        /* 让出处理器 */
        rt_thread_yield();                                  (5)
    }

    /* 扫描系统定时器列表 */
    rt_timer_check();
}
```

代码清单 12-4**(1)**：获取当前线程线程控制块。

代码清单 12-4**(2)**：递减当前线程的时间片。

代码清单 12-4**(3)**：如果时间片用完，则重置时间片，然后让出处理器，具体是否真正的要让出处理器还要看当前线程下是否有两个以上的线程。

代码清单 12-4**(4)**：如果时间片耗完，则重置时间片。

代码清单 12-4**(5)**：调用 rt_thread_yield 让出处理器，该函数在 thread.c 中定义，具体实现见代码清单 12-5。

列表 5: 代码清单 12-5 rt_thread_yield 函数定义

```
/**
* 该函数将让当前线程让出处理器，调度器选择最高优先级的线程运行。当前让出处理器之后，
* 当前线程还是在就绪态。
*
* @return RT_EOK
*/
rt_err_t rt_thread_yield(void)
{
    register rt_base_t level;
    struct rt_thread *thread;

    /* 关中断 */
```

(下页继续)

(续上页)

```
    level = rt_hw_interrupt_disable();

    /* 获取当前线程的线程控制块 */
    thread = rt_current_thread;                                          (1)

    /* 如果线程在就绪态，且同一个优先级下不止一个线程 */
    if ((thread->stat & RT_THREAD_STAT_MASK) == RT_THREAD_READY &&       (2)
        thread->tlist.next != thread->tlist.prev)
    {
        /* 将时间片耗完的线程从就绪列表移除 */
        rt_list_remove(&(thread->tlist));                                (3)

        /* 将线程插入到该优先级下的链表的尾部 */                          (4)
        rt_list_insert_before(&(rt_thread_priority_table[thread->current_
↪priority]),
                              &(thread->tlist));

        /* 开中断 */
        rt_hw_interrupt_enable(level);                                   (5)

        /* 执行调度 */
        rt_schedule();

        return RT_EOK;
    }

    /* 开中断 */
    rt_hw_interrupt_enable(level);

    return RT_EOK;
}
```

代码清单 12-5(1)：获取当前线程线程控制块。

代码清单 12-5(2)：如果线程在就绪态，且同一个优先级下不止一个线程，则执行 if 里面的代码，否则函数返回。

代码清单 12-5(3)：将时间片耗完的线程从就绪列表移除。

代码清单 12-5(4)：将时间片耗完的线程插入到该优先级下的链表的尾部，把机会让给下一个线程。

代码清单 12-5(5)：执行调度。

14.2 修改 main.c 文件

main.c 文件的修改内容具体见代码清单 12-6 的高亮部分。

列表 6: 代码清单 12-6 main.c 文件内容

```
/*
 ************************************************************************
 *                              包含的头文件
 ************************************************************************
 */

#include <rtthread.h>
#include <rthw.h>
#include "ARMCM7.h"


/*
 ************************************************************************
 *                              全局变量
 ************************************************************************
 */
```

(下页继续)

(续上页)

```
rt_uint8_t flag1;
rt_uint8_t flag2;
rt_uint8_t flag3;

extern rt_list_t rt_thread_priority_table[RT_THREAD_PRIORITY_MAX];

/*
 ***************************************************************************
 *                          线程控制块 & STACK & 线程声明
 ***************************************************************************
 */


/* 定义线程控制块 */
struct rt_thread rt_flag1_thread;
struct rt_thread rt_flag2_thread;
struct rt_thread rt_flag3_thread;

ALIGN(RT_ALIGN_SIZE)
/* 定义线程栈 */
rt_uint8_t rt_flag1_thread_stack[512];
rt_uint8_t rt_flag2_thread_stack[512];
rt_uint8_t rt_flag3_thread_stack[512];

/* 线程声明 */
void flag1_thread_entry(void *p_arg);
void flag2_thread_entry(void *p_arg);
void flag3_thread_entry(void *p_arg);

/*
 ***************************************************************************
 *                                    函数声明
```

(下页继续)

(续上页)

```
 ****************************************************************************
 */
void delay(uint32_t count);

/****************************************************************************
 * @brief  main 函数
 * @param  无
 * @retval 无
 *
 * @attention
 ****************************************************************************
 */
int main(void)
{
    /* 硬件初始化 */
    /* 将硬件相关的初始化放在这里，如果是软件仿真则没有相关初始化代码 */

    /* 关中断 */
    rt_hw_interrupt_disable();

    /* SysTick 中断频率设置 */
    SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );

    /* 系统定时器列表初始化 */
    rt_system_timer_init();

    /* 调度器初始化 */
    rt_system_scheduler_init();

    /* 初始化空闲线程 */
    rt_thread_idle_init();

```

(下页继续)

(续上页)

```
    /* 初始化线程 */
    rt_thread_init( &rt_flag1_thread,                    /* 线程控制块 */
                    "rt_flag1_thread",                   /* 线程名字，字符串形式␣
↪*/
                    flag1_thread_entry,                  /* 线程入口地址 */
                    RT_NULL,                             /* 线程形参 */
                    &rt_flag1_thread_stack[0],           /* 线程栈起始地址 */
                    sizeof(rt_flag1_thread_stack),       /* 线程栈大小，单位为字
节 */
                    2,                                   /* 优先级 */
                    4);                                  /* 时间片 */
    /* 将线程插入到就绪列表 */
    rt_thread_startup(&rt_flag1_thread);

    /* 初始化线程 */
    rt_thread_init( &rt_flag2_thread,                    /* 线程控制块 */
                    "rt_flag2_thread",                   /* 线程名字，字符串形式␣
↪*/
                    flag2_thread_entry,                  /* 线程入口地址 */
                    RT_NULL,                             /* 线程形参 */
                    &rt_flag2_thread_stack[0],           /* 线程栈起始地址 */
                    sizeof(rt_flag2_thread_stack),       /* 线程栈大小，单位为字
节 */
                    3,                                   /* 优先级 */
                    2);                                  /* 时间片 */
    /* 将线程插入到就绪列表 */
    rt_thread_startup(&rt_flag2_thread);

    /* 初始化线程 */
    rt_thread_init( &rt_flag3_thread,                    /* 线程控制块 */
                    "rt_flag3_thread",                   /* 线程名字，字符串形式␣
↪*/
```

(下页继续)

(续上页)

```
                    flag3_thread_entry,                 /* 线程入口地址 */
                    RT_NULL,                            /* 线程形参 */
                    &rt_flag3_thread_stack[0],          /* 线程栈起始地址 */
                    sizeof(rt_flag3_thread_stack),      /* 线程栈大小，单位为字节 */
                    3,                                  /* 优先级 */
                    3);                                 /* 时间片 */
    /* 将线程插入到就绪列表 */
    rt_thread_startup(&rt_flag3_thread);

    /* 启动系统调度器 */
    rt_system_scheduler_start();
}

/*
 ******************************************************************************
 *                                 函数实现
 ******************************************************************************
 */

/* 软件延时 */
void delay (uint32_t count)
{
    for(; count!=0; count--);
}

/* 线程 1 */
void flag1_thread_entry( void *p_arg )
{
    for( ;; )
    {
        flag1 = 1;
```

(下页继续)

(续上页)

```
        rt_thread_delay(3);                    /*(阻塞延时)*/
        flag1 = 0;
        rt_thread_delay(3);
    }
}

/* 线程 2 */
void flag2_thread_entry( void *p_arg )
{
    for( ;; )
    {
        flag2 = 1;
        //rt_thread_delay(2);
        delay( 100 );                          /*(软件延时)*/
        flag2 = 0;
        //rt_thread_delay(2);
        delay( 100 );
    }
}

/* 线程 3 */
void flag3_thread_entry( void *p_arg )
{
    for( ;; )
    {
        flag3 = 1;
        //rt_thread_delay(3);
        delay( 100 );                          /*(软件延时)*/
        flag3 = 0;
        //rt_thread_delay(3);
        delay( 100 );
    }
```

(下页继续)

(续上页)

```
}


void SysTick_Handler(void)
{
    /* 进入中断 */
    rt_interrupt_enter();

    /* 更新时基 */
    rt_tick_increase();

    /* 离开中断 */
    rt_interrupt_leave();
}
```

代码清单 *12-6*(**优先级**)：线程 1 的优先级修改为 2，线程 2 和线程 3 的优先级修改为 3。

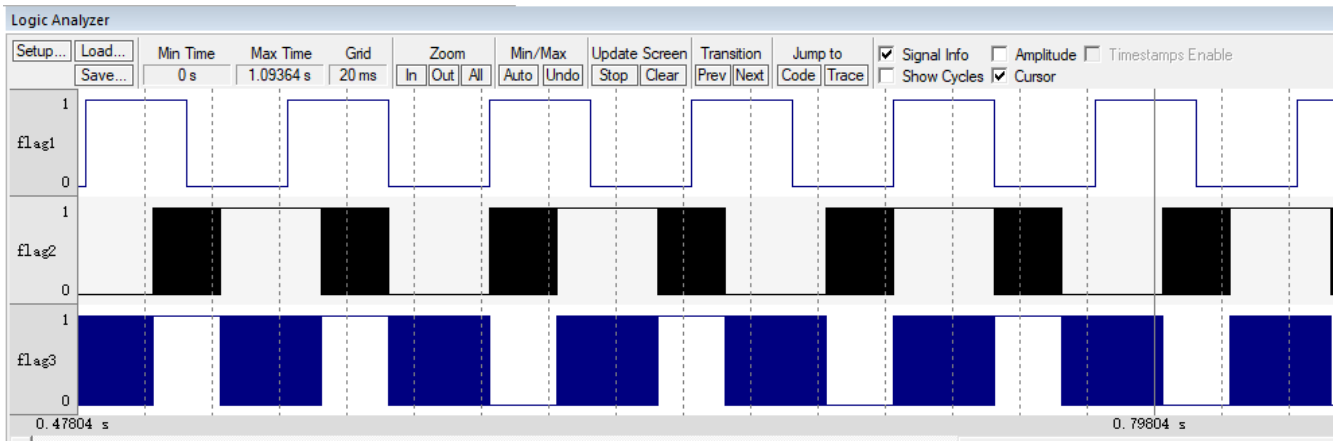
代码清单 *12-6*(**时间片**)：线程 1 的时间片设置为 4（可是与线程 1 同优先级的线程没有，这里设置了时间片也没有什么鸟用，不信等下看实验现象），线程 2 和线程 3 的时间片设置为 3。

代码清单 *12-6*(**阻塞延时**)：设置线程 1 高低电平的时间为 3 个 tick，且延时要使用阻塞延时。

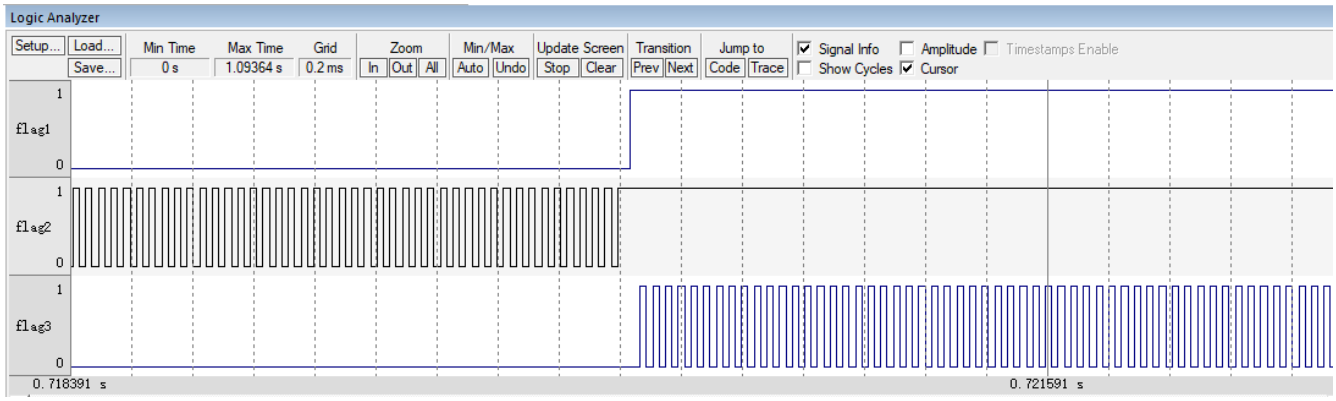
代码清单 *12-6*(**软件延时**)：将线程 2 和线程 3 的延时改成软件延时，因为这两个线程的优先级是相同的，当他们的时间片耗完的时候让出处理器进行系统调度，不会一直的占有 CPU，所以可以使用软件延时，但是线程 1 却不可以，因为与线程 1 同优先级的线程没有，时间片功能不起作用，当时间片耗完的时候不会让出 CPU，会一直的占有 CPU，所以不能使用软件延时。

14.3 实验现象

进入软件调试，全速运行程序，逻辑分析仪中的仿真波形图具体见图 *12-1*。



从图 *12-1* 中可以看出线程 1 运行一个周期的时间为 6 个 tick，与线程 1 初始化时设置的 4 个时间片不符，说明同一个优先级下只有一个线程时时间片不起作用。线程 2 和线程 3 运行一个周期的时间分别为 2 个 tick 和 3 个 tick，且线程 2 运行的时候线程 3 是不运行的，从而说明我们的时间片功能起作用了，搞定。图 12-1 线程 2 和线程 3 运行的波形图现在是太密集了，一团黑，看不出代码的执行效果，我们将波形图放大之后，可以在线程要求的时间片内 flag2 和 flag3 进行了很多很多次的翻转，具体见图 *12-2*。



第 15 章　第二部分：RT-Thread 内核应用开发

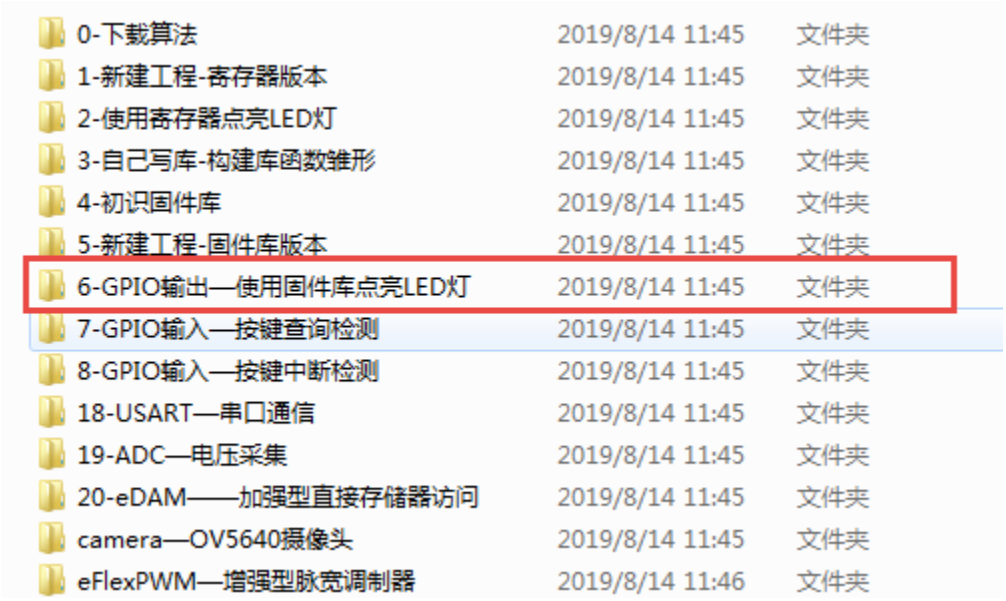
本书第二部分以野火 i.MX RT 全系列开发板为硬件平台，来讲解 RT-Thread 的内核应用。这部分不会再深究源码的实现，着重讲解 RT-Thread 各个内核对象的使用，例如线程如何创建、优先级如何分配、内部 IPC 通信机制如何使用等 RTOS 知识点。

第 16 章　移植 RT-Thread 到 i.MX RT

本章开始，先新建一个基于野火 i.MX RT 全系列（包含 M3/4/7）开发板的的 RT-Thread 的工程模板，让 RT-Thread 先跑起来。以后所有的 RT-Thread 相关的例程我们都在此模板上修改和添加代码，不用再反反复复地新建。在本书配套的例程中，每一章的例程对野火 i.MX RT 的每一个板子都会有一个对应的例程，但是区别都很小，如果有区别的地方我会在教程里面详细指出，如果没有特别备注那么都是一样的。

16.1 获取 i.MX RT 的裸机工程模板

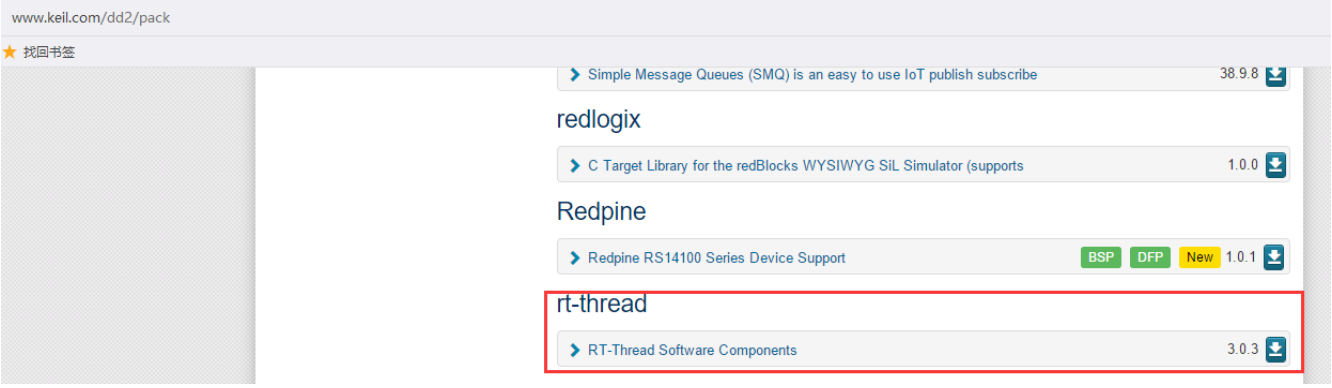
i.MX RT 的裸机工程模板我们直接使用野火 i.MX RT 开发板配套的固件库例程即可。这里我们选取比较简单的例程—“GPIO 输出—使用固件库点亮 LED”作为裸机工程模板。该裸机工程模板均可以在对应板子的书籍配套例程的目录下获取到，下面以野火 RT1052PRO 板子的光盘目录为例，具体见图 *13-1*。



16.2 下载 RT-Thread Nano 源码

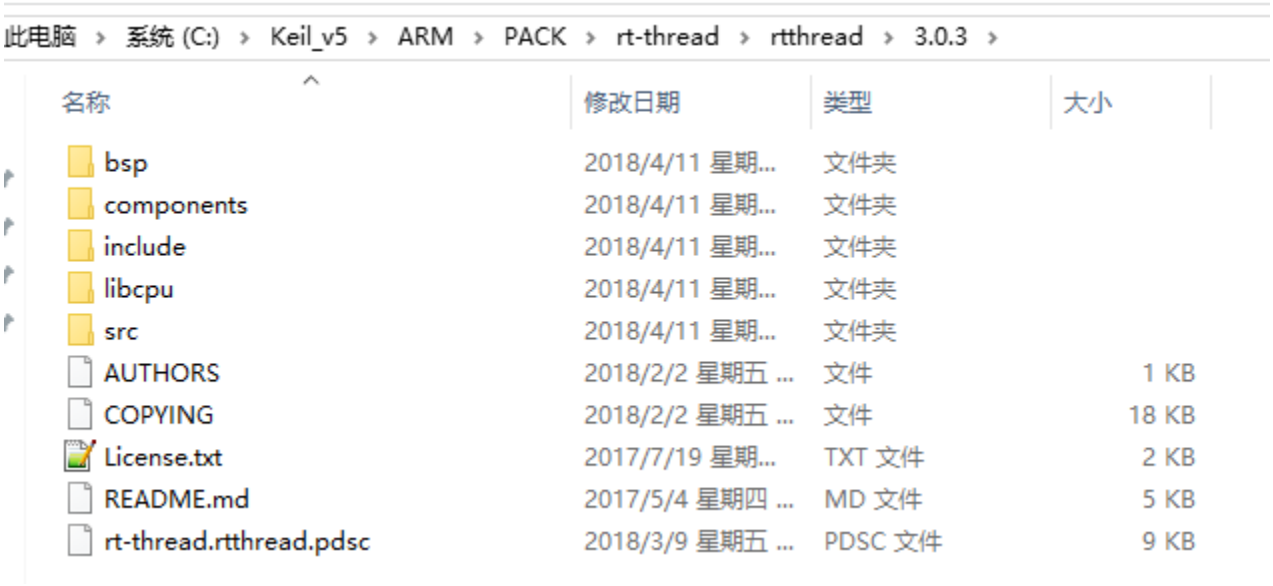
Nano 是 Master 的精简版，去掉了一些组件和各种开发板的 BSP，保留了 OS 的核心功能，但足够我们使用。版本已经更新到了 3.0.3 版本，与 Master 的版本号一致。

RT-Thread Master 的源码可从 RT-Thread GitHub 仓库地址：https://github.com/RT-Thread/rt-thread下载到，Nano 就是从里面扣出来的。RT-Thread 官方并没有将抠出来的 Nano 放到他们的官方网站，而是作为一个 Package 放在了 KEIL 网站—http://www.keil.com/dd2/pack/中，供用户下载，具体见图 *13-2*，目前的版本号是 3.0.3，如果以后更新到更高的版本则以最新的版本为准。

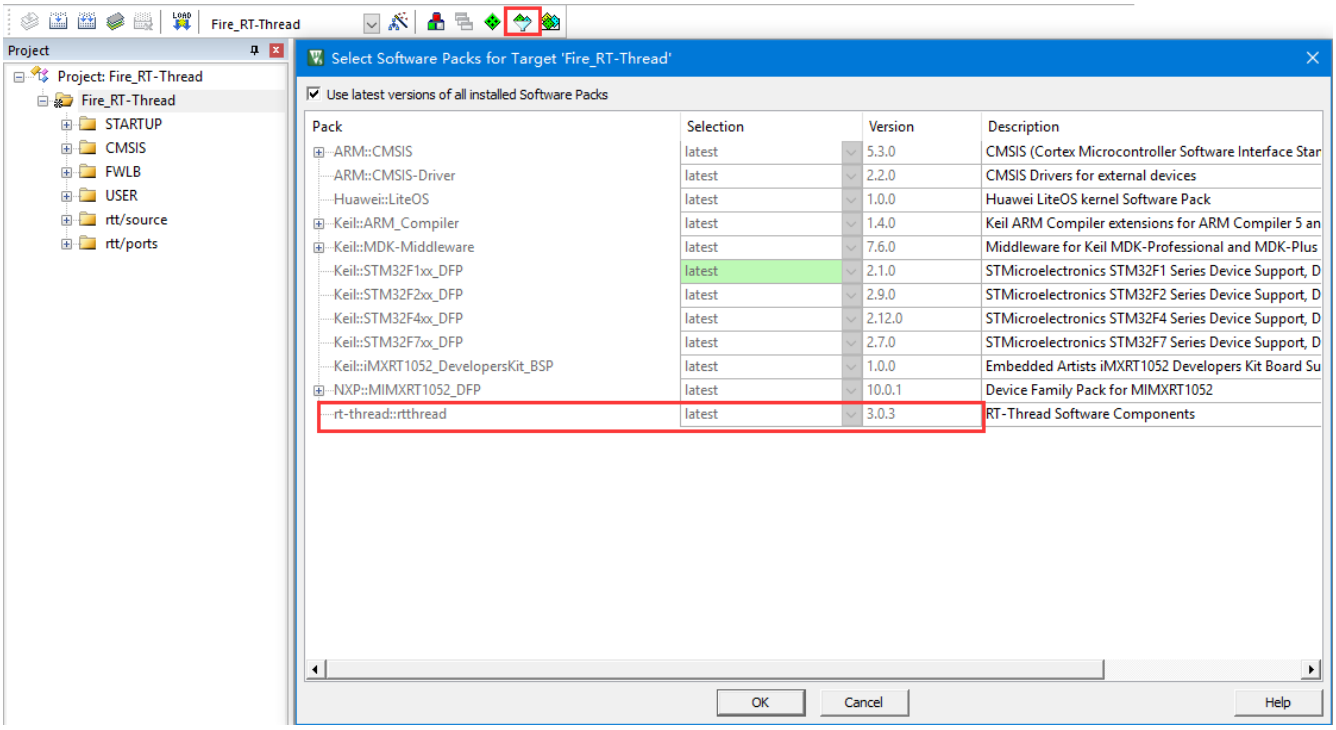


16.3 安装 RT-Thread Package

下载下来之后是一个以 exe 为后缀的文件，点击安装即可，安装目录与你的 KEIL 安装目录一样，安装成功之后，可以在 KEIL 的 PACK 目录下找到刚刚安装的 Package 的所有的文件，具体见图 *13-3*。



这样安装成功之后，就可以在 KEIL 里面的软件包管理器中将 RT-Thread Nano 直接添加到工程里面，具体见图 *13-4*。



16.4 往裸机工程添加 RT-Thread 源码

16.4.1 拷贝 RT-Thread Package 到裸机工程根目录

使用这种方法打包的 RT-Thread 工程，拷贝到一台没有安装 RT-Thread Package 的电脑上面是使用不了的，会提示找不到 RT-Thread 的源文件。鉴于 RT-Thread Package 容量很小，我们直接将安装在 KEIL PACK 目录下的整个 RT-Thread 文件夹拷贝到我们的 i.MX RT 裸机工程里面，让整个 RT-Thread Package 跟随我们的工程一起发布，具体见图 *13-5*。

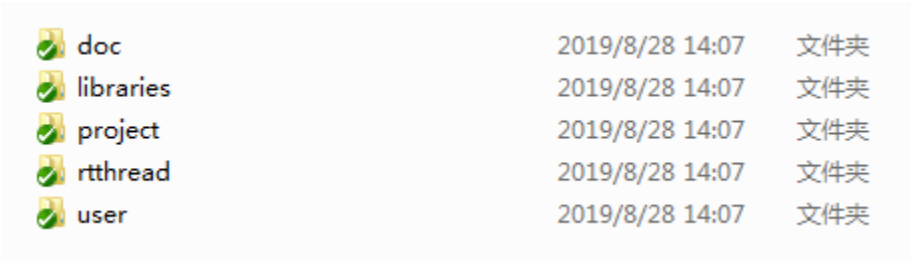


图 *13-5* 中 RT-Thread 文件夹下就是 RT-Thread Nano 的所有东西，该文件夹下的具体内容见表格 13-1。

表格 13-1 RT-dhread 文件夹内容组成

文件夹	文件夹	描述
rtthread/3.0.3	bsp	板级支持包
	components/finsh	RT-Thread 组件
	include	头文件
	include/libc	
	libcpu/arm/cortex-m0	与处理器相关的接口文件
	libcpu/arm/cortex-m3	
	libcpu/arm/cortex-m4	
	libcpu/arm/cortex-m7	
	src	RT-Thread 内核源码

16.4.2 拷贝 rtconfig.h 文件到 user 文件夹

将 RT-Thread/3.0.3/bsp 文件夹下面的 rtconfig.h 配套文件拷贝到工程根目录下面的 user 文件夹，等下我们需要对这个文件进行修改。

用户可以通过修改这个 RT-Thread 内核的配置头文件来裁剪 RT-Thread 的功能，所以我们把它拷贝一份放在 user 这个文件夹下面。user，见名之义我们就可以知道里面存放的文件都是用户自己编写的。

16.4.3 拷贝 board.c 文件到 user 文件夹

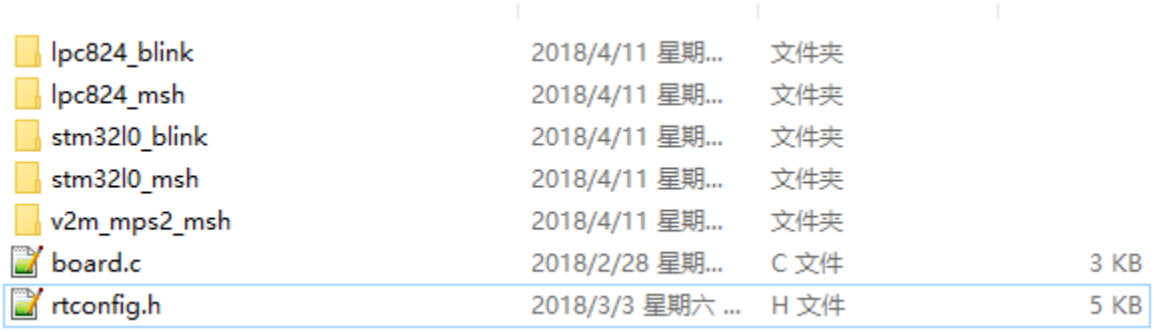
将 RT-Thread/3.0.3/bsp 文件夹下面的 board.c 配套文件拷贝到工程根目录下面的 user 文件夹，等下我们需要对这个 board.c 进行修改。

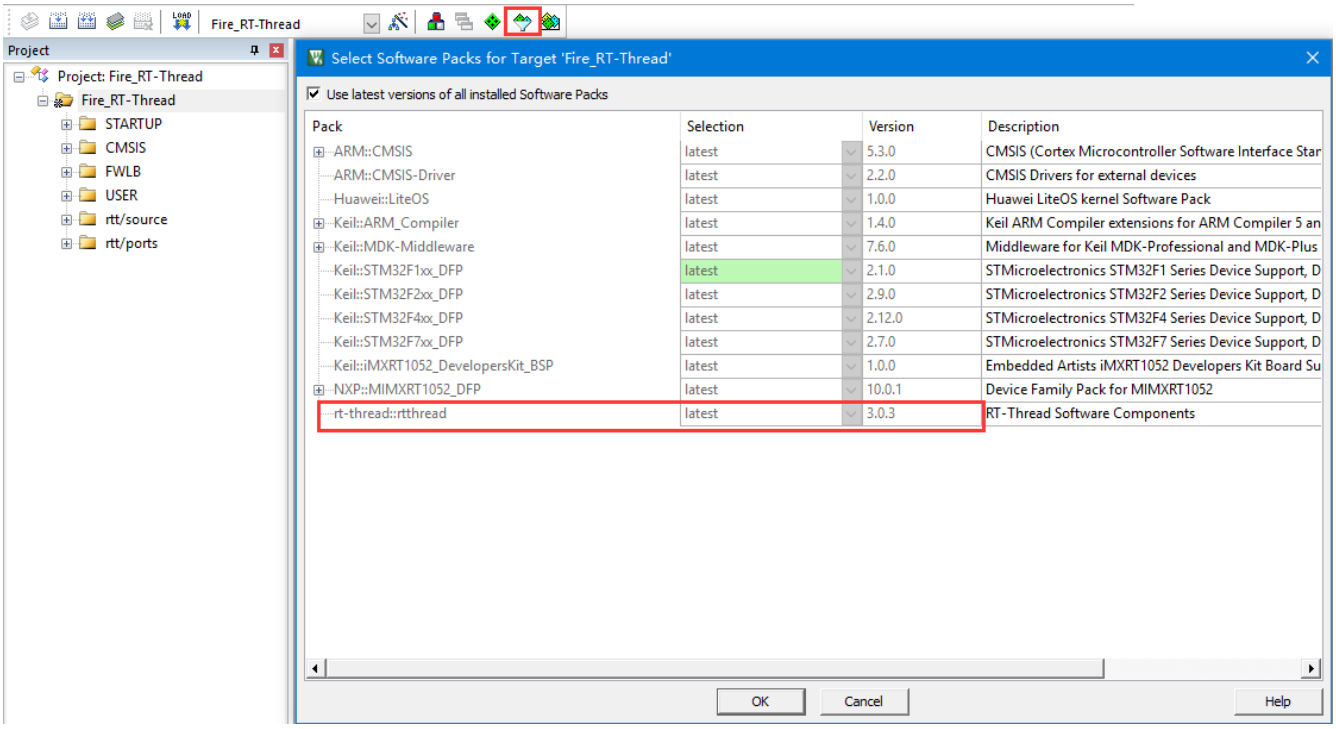
16.4.4 RT-Thread 文件夹内容简介

接下来我们对 RT-Thread 文件夹下面的内容做个简单的介绍，好让我们能够更顺心地使用 RT-Thread。

16.4.5 bsp 文件夹简介

bsp 文件夹里面存放的是板级支持包，即 board support package 的英文缩写。RT-Thread 为了推广自己，会给各种半导体厂商的评估板写好驱动程序，这些驱动程序就放在 bsp 这个目录下，我们这里用的是 nano 版本，只有几款开发板的驱动，具体见图 *13-6*，如果是 Master 版本，则存放了非常多的开发板的驱动，具体见图 *13-7*。bsp 文件夹下面的 board.c 这是 RT-Thread 用来初始化开发板硬件的相关函数。rtconfig.h 是 RT-Thread 功能的配置头文件，里面定义了很多宏，通过这些宏定义，我们可以裁剪 RT-Thread 的功能。用户在使用 RT-Thread 的时候，用户只需要修改 board.c 和 rtconfig.h 这两个文件的内容即可，其它文件我们不需要改动。如果为了减小工程的大小，bsp 文件夹下面除了 board.c 和 rtconfig.h 这两个文件要保留外，其它的统统可以删除。





16.4.6 components 文件夹简介

在 RT-Thread 看来，除了内核，其它第三方加进来的软件都是组件，比如 gui、fatfs、lwip 和 finsh 等。那么这些组件就放在 components 这个文件夹内，目前 nano 版本只放了 finsh，其它的都被删除了，master 版本则放了非常多的组件。finsh 是 RT-Thread 组件里面最具特色的，它通过串口打印的方式来输出各种信息，方便我们调试程序。

16.4.7 include 文件夹简介

include 目录下面存放的是 RT-Thread 内核的头文件，是内核不可分割的一部分。

16.4.8 libcpu 文件夹简介

RT-Thread 是一个软件，单片机是一个硬件，RT- Thread 要想运行在一个单片机上面，它们就必须关联在一起，那么怎么关联？还是得通过写代码来关联，这部分关联的文件叫接口文件，通常由汇编和 C 联合编写。这些接口文件都是跟硬件密切相关的，不同的硬件接口文件是不一样的，但都大同小异。编写这些接口文件的过程我们就叫移植，移植的过程通常由 RT-Thread 和 mcu 原厂的人来负责，移植好的这些接口文件就放在 libcpu 这个文件夹的目录下。RT-Thread nano 目前在 libcpu 目录下只放了 cortex-m0、m3、m4 和 m7 内核的单片机的接口文件，只要是使用了这些内核的 mcu 都可以使用里面的接口文件。通常网络上出现的叫“移植某某某 RTOS 到某某某 MCU”的教程，其实准确来说，不能够叫移植，应该叫使用官方的移植，因为这些跟硬件相关的接口文件，RTOS 官方都已经写好了，我们只是使用而已。我们本章讲的移植也是使用 RT-Thread 官方的移植，关于这些底层的移植文件我们已经在第一部分“从 0 到 1 教你写 RT-Thread 内核”有非常详细的讲解，这里我们直接使用即可。

16.4.9 src 文件夹简介

src 目录下面存放的是 RT-Thread 内核的源文件，是内核的核心，我们在第一部分“从 0 到 1 教你写 RT-Thread 内核”里面讲解的就是这里面内容。

16.4.9.1 添加 RT-Thread 源码到工程组文件夹

在上一步我们只是将 RT-Thread 的源码放到了本地工程目录下，还没有添加到开发环境里面的组文件夹里面。

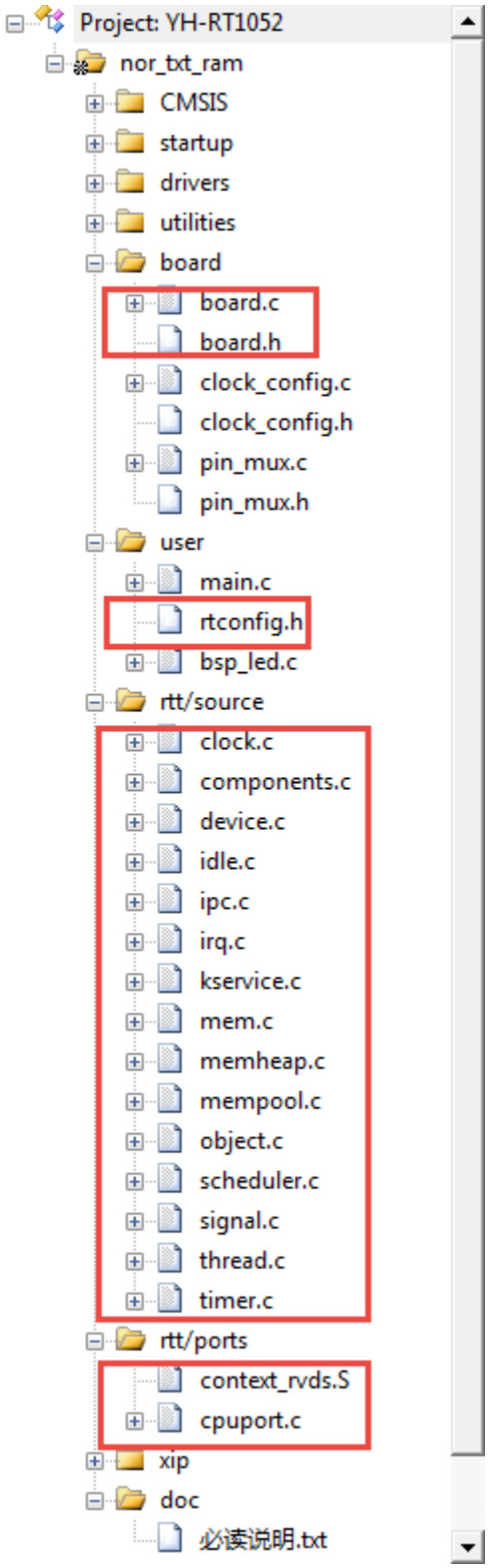
16.4.10 新建 rtt/source 和 rtt/ports 组

接下来我们在开发环境里面新建 rtt/source 和 rtt/ports 两个组文件夹，其中 rtt/source 用于存放 src 文件夹的内容，rtt/ports 用于存放 libcpu/arm/cortex-m? 文件夹的内容，“?”表示 3、4 或者 7，具体选择哪个得看你使用的是野火哪个型号的 i.MX RT 开发板，具体见表格 13-2。

表格 13-2 野火 i.MX RT 开发板型号对应 RT-Thread 的接口文件

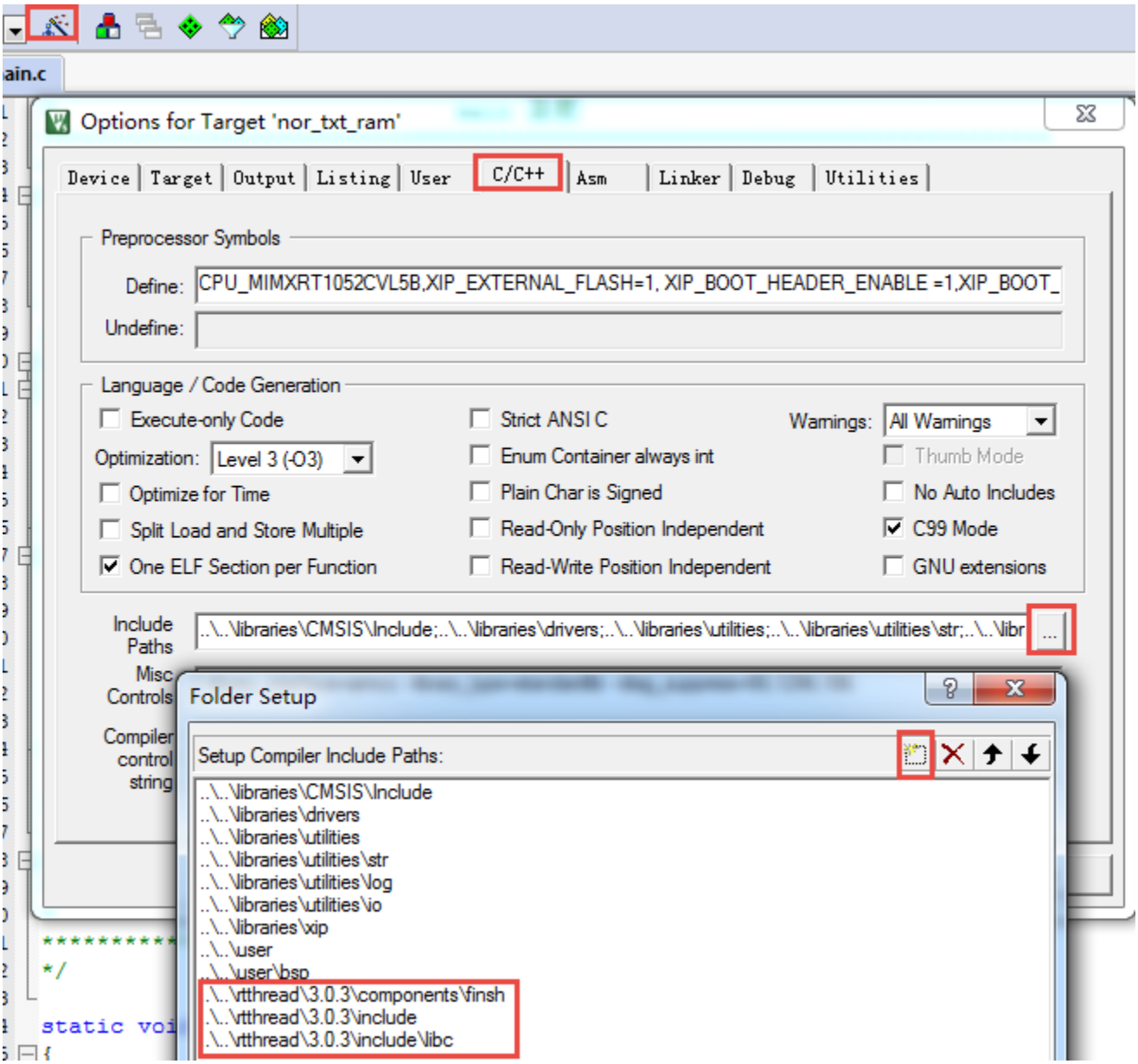
野火 i.MX RT 开发板型号	具体芯片型号	RT-Thread 不同内核的接口文件
RT1021	PIMXRT1021DAG5A	libcpu/arm/cortex-m7
RT1052	MIMXRT1052CVL5B	

bsp 里面的 rtconfig.h 和 board.c 添加到 user 组文件夹下，其中 rtconfig.h 用于配置 RT-Thread 的功能，board.c 用于存放硬件相关的初始化函数。源码添加完毕之后，具体见图 *13-8*。


Project: YH-RT1052
nor_txt_ram
CMSIS
startup
drivers
utilities
board
board.c
board.h
clock_config.c
clock_config.h
pin_mux.c
pin_mux.h
user
main.c
rtconfig.h
bsp_led.c
rtt/source
clock.c
components.c
device.c
idle.c
ipc.c
irq.c
kservice.c
mem.c
memheap.c
mempool.c
object.c
scheduler.c
signal.c
thread.c
timer.c
rtt/ports
context_rvds.S
cpuport.c
xip
doc
必读说明.txt

16.4.11 指定 RT-Thread 头文件的路径

RT-Thread 的源码已经添加到开发环境的组文件夹下面，编译的时候需要为这些源文件指定头文件的路径，不然编译会报错。RT-Thread 的源码里面只有 RT-Thread3.0.3componentsfinsh、RT-Thread3.0.3include 和 RT-Thread3.0.3includelibc 这三个文件夹下面有头文件，只需要将这三个头文件的路径在开发环境里面指定即可。同时我们还将 RT-Thread3.0.3bsp 里面的 rtconfig.h 这个头文件拷贝到了工程根目录下的 user 文件夹下，所以 user 的路径也要加到开发环境里面。RT-Thread 头文件的路径添加完成后的效果具体见图 *13-9*。


Options for Target 'nor_txt_ram'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Preprocessor Symbols
Define: CPU_MIMXRT1052CVL5B,XIP_EXTERNAL_FLASH=1, XIP_BOOT_HEADER_ENABLE =1,XIP_BOOT_
Undefine:
Language / Code Generation
Execute-only Code
Strict ANSI C
Warnings: All Warnings
Optimization: Level 3 (-O3)
Enum Container always int
Thumb Mode
Optimize for Time
Plain Char is Signed
No Auto Includes
Split Load and Store Multiple
Read-Only Position Independent
C99 Mode
One ELF Section per Function
Read-Write Position Independent
GNU extensions
Include Paths
..\..\libraries\CMSIS\Include;..\..\libraries\drivers;..\..\libraries\utilities;..\..\libraries\utilities\str;..\..\libr
Misc Controls
Compiler control string
Folder Setup
Setup Compiler Include Paths:
..\..\libraries\CMSIS\Include
..\..\libraries\drivers
..\..\libraries\utilities
..\..\libraries\utilities\str
..\..\libraries\utilities\log
..\..\libraries\utilities\io
..\..\libraries\xip
..\..\user
..\..\user\bsp
..\..\rtthread\3.0.3\components\finsh
..\..\rtthread\3.0.3\include
..\..\rtthread\3.0.3\include\libc

16.5 修改 rtconfig.h

rtconfig.h 是直接从 RT-Thread/3.0.3/bsp 文件夹下面拷贝过来的，该头文件对裁剪整个 RT-Thread 所需的功能的宏均做了定义，有些宏定义被使能，有些宏定义被失能，一开始我们只需要配置最简单的功能即可。要想随心所欲的配置 RT-Thread 的功能，我们必须对这些宏定义的功能有所掌握，下面我们先简单的介绍下这些宏定义的含义，然后再对这些宏定义进行修改。

列表 1: 代码清单 13-1 rtconfig.h 文件内容

```
/* RT-Thread config file */

#ifndef __RTTHREAD_CFG_H__
#define __RTTHREAD_CFG_H__

#include "RTE_Components.h"                                  (1)

// <<< Use Configuration Wizard in Context Menu >>>          (2)
// <h>Basic Configuration                                    (3)
// <o>Maximal level of thread priority <8-256>
//   <i>Default: 32
#define RT_THREAD_PRIORITY_MAX  32                           (3)-1
// <o>OS tick per second
//   <i>Default: 1000   (1ms)
#define RT_TICK_PER_SECOND  100                              (3)-2
// <o>Alignment size for CPU architecture data access
//   <i>Default: 4
#define RT_ALIGN_SIZE   4                                    (3)-3
// <o>the max length of object name<2-16>
//   <i>Default: 8
#define RT_NAME_MAX    8                                     (3)-4
// <c1>Using RT-Thread components initialization
//   <i>Using RT-Thread components initialization
```

(下页继续)

(续上页)

```
#define RT_USING_COMPONENTS_INIT                                (3)-5
// </c>
// <c1>Using user main
//  <i>Using user main
#define RT_USING_USER_MAIN                                      (3)-6
// </c>
// <o>the size of main thread<1-4086>
//  <i>Default: 512
#define RT_MAIN_THREAD_STACK_SIZE     256                       (3)-7

// </h>

// <h>Debug Configuration                                       (4)
// <c1>enable kernel debug configuration
//  <i>Default: enable kernel debug configuration
//#define RT_DEBUG
// </c>
// <o>enable components initialization debug configuration<0-1>
//  <i>Default: 0
#define RT_DEBUG_INIT 0
// <c1>thread stack over flow detect
//  <i> Diable Thread stack over flow detect
//#define RT_USING_OVERFLOW_CHECK
// </c>
// </h>

// <h>Hook Configuration                                         (5)
// <c1>using hook
//  <i>using hook
//#define RT_USING_HOOK
// </c>
// <c1>using idle hook
```

(下页继续)

(续上页)

```
//  <i>using idle hook
//#define RT_USING_IDLE_HOOK
// </c>
// </h>

// <e>Software timers Configuration                        (6)
// <i> Enables user timers
#define RT_USING_TIMER_SOFT         0
#if RT_USING_TIMER_SOFT == 0
#undef RT_USING_TIMER_SOFT
#endif
// <o>The priority level of timer thread <0-31>
//  <i>Default: 4
#define RT_TIMER_THREAD_PRIO                4
// <o>The stack size of timer thread <0-8192>
//  <i>Default: 512
#define RT_TIMER_THREAD_STACK_SIZE  512
// <o>The soft-timer tick per second <0-1000>
//  <i>Default: 100
#define RT_TIMER_TICK_PER_SECOND    100
// </e>

// <h>IPC(Inter-process communication) Configuration    (7)
// <c1>Using Semaphore
//  <i>Using Semaphore
#define RT_USING_SEMAPHORE                              (7)-1
// </c>
// <c1>Using Mutex
//  <i>Using Mutex
//#define RT_USING_MUTEX                                (7)-2
// </c>
// <c1>Using Event
```

(下页继续)

(续上页)

```
//  <i>Using Event
//#define RT_USING_EVENT                                  (7)-3
// </c>
// <c1>Using MailBox
//  <i>Using MailBox
#define RT_USING_MAILBOX                                  (7)-5
// </c>
// <c1>Using Message Queue
//  <i>Using Message Queue
//#define RT_USING_MESSAGEQUEUE                           (7)-5
// </c>
// </h>

// <h>Memory Management Configuration                     (8)
// <c1>Using Memory Pool Management
//  <i>Using Memory Pool Management
//#define RT_USING_MEMPOOL                                (8)-1
// </c>
// <c1>Dynamic Heap Management
//  <i>Dynamic Heap Management
//#define RT_USING_HEAP                                   (8)-2
// </c>
// <c1>using small memory
//  <i>using small memory
#define RT_USING_SMALL_MEM                                (8)-3
// </c>
// <c1>using tiny size of memory
//  <i>using tiny size of memory
//#define RT_USING_TINY_SIZE                              (8)-4
// </c>
// </h>

```

(下页继续)

(续上页)

```
// <h>Console Configuration                                (9)
// <c1>Using console
//   <i>Using console
#define RT_USING_CONSOLE
// </c>
// <o>the buffer size of console <1-1024>
//   <i>the buffer size of console
//   <i>Default: 128  (128Byte)
#define RT_CONSOLEBUF_SIZE          128
// <s>The device name for console
//   <i>The device name for console
//   <i>Default: uart1
#define RT_CONSOLE_DEVICE_NAME      "uart2"
// </h>


#if defined(RTE_FINSH_USING_MSH)                            (10)
#define RT_USING_FINSH
#define FINSH_USING_MSH
#define FINSH_USING_MSH_ONLY
// <h>Finsh Configuration
// <o>the priority of finsh thread <1-7>
//   <i>the priority of finsh thread
//   <i>Default: 6
#define __FINSH_THREAD_PRIORITY     5
#define FINSH_THREAD_PRIORITY       (RT_THREAD_PRIORITY_MAX / 8 * __FINSH_
↪THREAD_PRIORITY + 1)
// <o>the stack of finsh thread <1-4096>
//   <i>the stack of finsh thread
//   <i>Default: 4096  (4096Byte)
#define FINSH_THREAD_STACK_SIZE     512
// <o>the history lines of finsh thread <1-32>
```

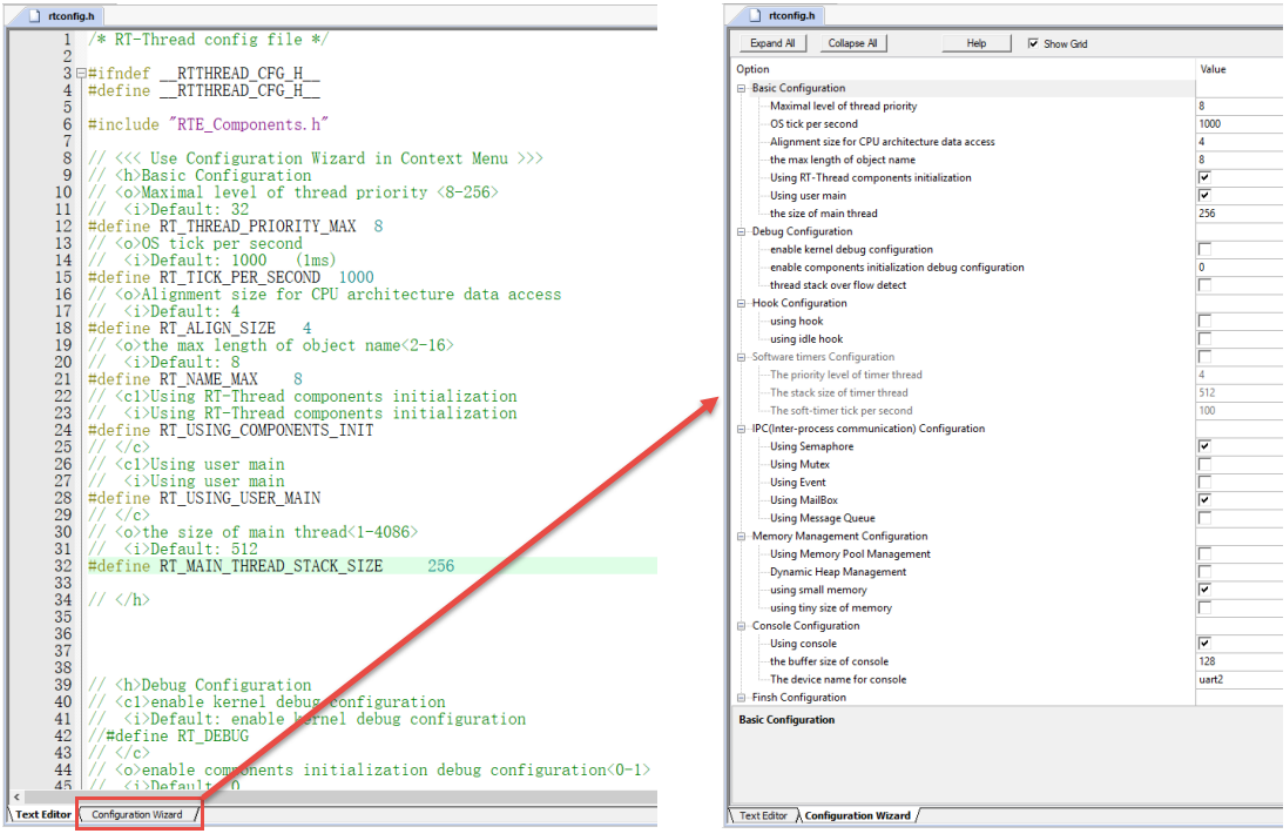
(下页继续)

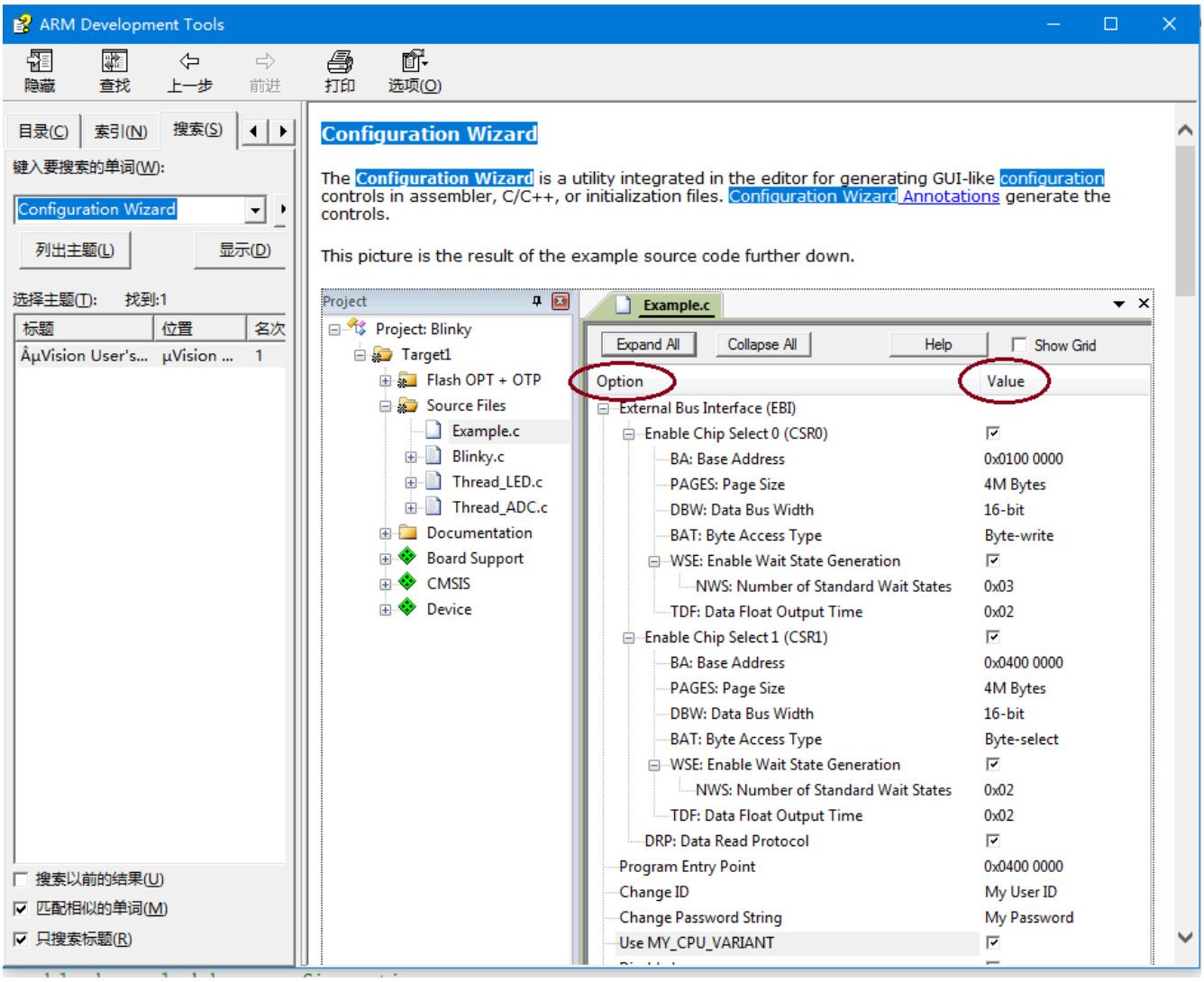
(续上页)

```
//  <i>the history lines of finsh thread
//  <i>Default: 5
#define FINSH_HISTORY_LINES         1
// <c1>Using symbol table in finsh shell
//  <i>Using symbol table in finsh shell
#define FINSH_USING_SYMTAB
// </c>
// </h>
#endif

#if defined(RTE_USING_DEVICE)                          (11)
#define RT_USING_DEVICE
#endif

// <<< end of configuration section >>>                (12)

#endif
```

代码清单 13-1(1)： 头文件 RTE_Components.h 是在 MDK 中添加 RT-Thead Package 时由 MDK 自动生成的，目前我们没有使用 MDK 中自带的 RT-Thread 的 Package，所以这个头文件不存在，如果包含了该头文件，编译的时候会报错，等下修改 rtconfig.h 的时候需要注释掉该头文件。

代码清单 13-1(2)： **Use Configuration Wizard in Context Menu：**在上下文中使用配置向导来配置 rtconfig.h 中的宏定义。接下来代码中夹杂的“<h> </h>”、“<o>”“<i>”、“<c1> </c>”和“<e> </e>”这些符号是 MDK 自带的配置向导控制符号，使用这些符号控制的代码可以生成一个对应的图形界面的配置向导，rtconfig.h 对应的配置向导具体见图 *13-10*。有关配置向导的语法，可在 MDK 的帮助文档里面找到，在搜索栏输入 Configuration Wizard 即可搜索到，具体见图 *13-11*。具体每一个符号的语法我们这里不做细讲，有兴趣的可以深究下。对于我个人，还是倾向于直接修改 rtconfig.h 中的源码，而不是通过这个配置向导来修改，就好比一个老烟枪抽烟的时候你要给他加个过滤嘴，那是不可能的，这辈子都是不可能的。


rtconfig.h
/* RT-Thread config file */
#ifndef __RTTHREAD_CFG_H__
#define __RTTHREAD_CFG_H__
#include "RTE_Components.h"
// <<< Use Configuration Wizard in Context Menu >>>
// <h>Basic Configuration
// <o>Maximal level of thread priority <8-256>
// <i>Default: 32
#define RT_THREAD_PRIORITY_MAX 8
// <o>OS tick per second
// <i>Default: 1000 (1ms)
#define RT_TICK_PER_SECOND 1000
// <o>Alignment size for CPU architecture data access
// <i>Default: 4
#define RT_ALIGN_SIZE 4
// <o>the max length of object name<2-16>
// <i>Default: 8
#define RT_NAME_MAX 8
// <c1>Using RT-Thread components initialization
// <i>Using RT-Thread components initialization
#define RT_USING_COMPONENTS_INIT
// </c>
// <c1>Using user main
// <i>Using user main
#define RT_USING_USER_MAIN
// </c>
// <o>the size of main thread<1-4086>
// <i>Default: 512
#define RT_MAIN_THREAD_STACK_SIZE 256
// </h>
// <h>Debug Configuration
// <c1>enable kernel debug configuration
// <i>Default: enable kernel debug configuration
//#define RT_DEBUG
// </c>
Text Editor
Configuration Wizard
rtconfig.h
Expand All
Collapse All
Help
Show Grid
Option
Value
Basic Configuration
Maximal level of thread priority 8
OS tick per second 1000
Alignment size for CPU architecture data access 4
the max length of object name 8
Using RT-Thread components initialization
Using user main
the size of main thread 256
Debug Configuration
enable kernel debug configuration
enable components initialization debug configuration 0
thread stack over flow detect
Hook Configuration
using hook
using idle hook
Software timers Configuration
The priority level of timer thread 4
The stack size of timer thread 512
The soft-timer tick per second 100
IPC(Inter-process communication) Configuration
Using Semaphore
Using Mutex
Using Event
Using MailBox
Using Message Queue
Memory Management Configuration
Using Memory Pool Management
Dynamic Heap Management
using small memory
using tiny size of memory
Console Configuration
Using console
the buffer size of console 128
The device name for console uart2
Finsh Configuration
Basic Configuration
Text Editor
Configuration Wizard



代码清单 13-1(3)：RT-Thread 的基本配置，要想 RT-Thread 准确无误的跑起来，这些基本配置必须得有且正确。

代码清单 13-1(3)-1：RT_THREAD_PRIORITY_MAX 这个宏表示 RT-Thread 支持多少个优先级，取值范围为 8~~~256，默认为 32。

代码清单 13-1(3)-2：RT_TICK_PER_SECOND 表示操作系统每秒钟有多少个 tick，tick 即是操作系统的时钟周期，默认为 1000，即操作系统的时钟周期 tick 等于 1ms。

代码清单 13-1(3)-3：RT_ALIGN_SIZE 这个宏表示 CPU 处理的数据需要多少个字节对齐，默认为 4 个字节。

代码清单 13-1**(3)-4**：RT_NAME_MAX 这个宏表示内核对象名字的最大长度，取值范围为 2~~~16，默认为 8。

代码清单 13-1**(3)-5**：使用 RT-Thread 组件初始化，默认使能。

代码清单 13-1**(3)-6**：使用用户 main 函数，默认打开。

代码清单 13-1**(3)-7**：main 线程栈大小，取值范围为 1~~~4086，单位为字节，默认为 512。

代码清单 13-1**(4)**：调试配置。包括了内核调试配置，组件调试配置和线程栈溢出检测，目前全部关闭。

代码清单 13-1**(5)**：钩子函数配置，目前全部关闭。

代码清单 13-1**(6)**：软件定时器配置，目前关闭，不使用软件定时器。

代码清单 13-1**(7)**：内部通信配置，包括信号量、互斥量、事件、邮箱和消息队列，根据需要配置。

代码清单 13-1**(8)**：内存管理配置。

代码清单 13-1**(8)-1**：RT_USING_MEMPOOL 这个宏用于表示是否使用内存池，目前关闭，不使用内存池。

代码清单 13-1**(8)-2**：RT_USING_HEAP 这个宏用于表示是否堆，目前关闭，不使用堆。

代码清单 13-1**(8)-3**：RT_USING_SMALL_MEM 这个宏用于表示是否使用小内存，目前使能。

代码清单 13-1**(8)-4**：RT_USING_TINY_SIZE 这个宏用于表示是否使用极小内存，目前关闭，不使用。

代码清单 13-1**(9)**：控制台配置。控制台即是 rt_kprintf() 函数调试输出的设备，通常使用串口。

代码清单 13-1**(10)**：FINSH 配置。

代码清单 13-1**(11)**：设备配置。

代码清单 13-1**(12)**：rtconfig.h 配置结束。

rtconfig.h 头文件的内容修改的不多，具体是：注释掉头文件 RTE_Components.h、修改了 RT_THREAD_PRIORITY_MAX、RT_TICK_PER_SECOND 和 RT_MAIN_THREAD_STACK_SIZE 这三个宏的大小，具体见代码清单 13-2 的高亮部分。

列表 2: 代码清单 13-2 rtconfig.h 文件修改

```
/* RT-Thread config file */

#ifndef __RTTHREAD_CFG_H__
#define __RTTHREAD_CFG_H__

//#include "RTE_Components.h"

// <<< Use Configuration Wizard in Context Menu >>>
// <h>Basic Configuration
// <o>Maximal level of thread priority <8-256>
//   <i>Default: 32
#define RT_THREAD_PRIORITY_MAX  8
// <o>OS tick per second
//   <i>Default: 1000   (1ms)
#define RT_TICK_PER_SECOND  1000
// <o>Alignment size for CPU architecture data access
//   <i>Default: 4
#define RT_ALIGN_SIZE   4
// <o>the max length of object name<2-16>
//   <i>Default: 8
#define RT_NAME_MAX    8
// <c1>Using RT-Thread components initialization
//   <i>Using RT-Thread components initialization
#define RT_USING_COMPONENTS_INIT
// </c>
// <c1>Using user main
//   <i>Using user main
#define RT_USING_USER_MAIN
// </c>
// <o>the size of main thread<1-4086>
```

(下页继续)

(续上页)

```
//  <i>Default: 512
#define RT_MAIN_THREAD_STACK_SIZE     512

// </h>

// <h>Debug Configuration
// <c1>enable kernel debug configuration
//  <i>Default: enable kernel debug configuration
//#define RT_DEBUG
// </c>
// <o>enable components initialization debug configuration<0-1>
//  <i>Default: 0
#define RT_DEBUG_INIT 0
// <c1>thread stack over flow detect
//  <i> Diable Thread stack over flow detect
//#define RT_USING_OVERFLOW_CHECK
// </c>
// </h>

// <h>Hook Configuration
// <c1>using hook
//  <i>using hook
//#define RT_USING_HOOK
// </c>
// <c1>using idle hook
//  <i>using idle hook
//#define RT_USING_IDLE_HOOK
// </c>
// </h>

// <e>Software timers Configuration
// <i> Enables user timers
```

(下页继续)

(续上页)

```
#define RT_USING_TIMER_SOFT         0
#if RT_USING_TIMER_SOFT == 0
#undef RT_USING_TIMER_SOFT
#endif
// <o>The priority level of timer thread <0-31>
//  <i>Default: 4
#define RT_TIMER_THREAD_PRIO                4
// <o>The stack size of timer thread <0-8192>
//  <i>Default: 512
#define RT_TIMER_THREAD_STACK_SIZE  512
// <o>The soft-timer tick per second <0-1000>
//  <i>Default: 100
#define RT_TIMER_TICK_PER_SECOND    100
// </e>

// <h>IPC(Inter-process communication) Configuration
// <c1>Using Semaphore
//  <i>Using Semaphore
#define RT_USING_SEMAPHORE
// </c>
// <c1>Using Mutex
//  <i>Using Mutex
//#define RT_USING_MUTEX
// </c>
// <c1>Using Event
//  <i>Using Event
//#define RT_USING_EVENT
// </c>
// <c1>Using MailBox
//  <i>Using MailBox
#define RT_USING_MAILBOX
// </c>
```

(下页继续)

(续上页)

```
// <c1>Using Message Queue
//   <i>Using Message Queue
//#define RT_USING_MESSAGEQUEUE
// </c>
// </h>

// <h>Memory Management Configuration
// <c1>Using Memory Pool Management
//   <i>Using Memory Pool Management
//#define RT_USING_MEMPOOL
// </c>
// <c1>Dynamic Heap Management
//   <i>Dynamic Heap Management
#define RT_USING_HEAP
// </c>
// <c1>using small memory
//   <i>using small memory
#define RT_USING_SMALL_MEM
// </c>
// <c1>using tiny size of memory
//   <i>using tiny size of memory
//#define RT_USING_TINY_SIZE
// </c>
// </h>

// <h>Console Configuration
// <c1>Using console
//   <i>Using console
#define RT_USING_CONSOLE
// </c>
// <o>the buffer size of console <1-1024>
//   <i>the buffer size of console
```

(下页继续)

(续上页)

```
//  <i>Default: 128  (128Byte)
#define RT_CONSOLEBUF_SIZE          128
// <s>The device name for console
//  <i>The device name for console
//  <i>Default: uart1
#define RT_CONSOLE_DEVICE_NAME      "uart2"
// </h>

#if defined(RTE_FINSH_USING_MSH)
#define RT_USING_FINSH
#define FINSH_USING_MSH
#define FINSH_USING_MSH_ONLY
// <h>Finsh Configuration
// <o>the priority of finsh thread <1-7>
//  <i>the priority of finsh thread
//  <i>Default: 6
#define __FINSH_THREAD_PRIORITY     5
#define FINSH_THREAD_PRIORITY       (RT_THREAD_PRIORITY_MAX / 8 * __FINSH_
↪THREAD_PRIORITY + 1)
// <o>the stack of finsh thread <1-4096>
//  <i>the stack of finsh thread
//  <i>Default: 4096  (4096Byte)
#define FINSH_THREAD_STACK_SIZE     512
// <o>the history lines of finsh thread <1-32>
//  <i>the history lines of finsh thread
//  <i>Default: 5
#define FINSH_HISTORY_LINES         1
// <c1>Using symbol table in finsh shell
//  <i>Using symbol table in finsh shell
#define FINSH_USING_SYMTAB
// </c>
// </h>
```

(下页继续)

(续上页)

```
#endif

#if defined(RTE_USING_DEVICE)
#define RT_USING_DEVICE
#endif

// <<< end of configuration section >>>

#endif
```

16.6 修改 board.c

board.c 是直接从 RT-Thread/3.0.3/bsp 文件夹下面拷贝过来的，里面存放的是与硬件相关的初始化函数，整个 board.c 中的内容具体见代码清单 13-3。

列表 3: 代码清单 13-3 board.c 文件内容

```
#ifdef RT_USING_HEAP        //(1)

#if defined(RT_USING_MEMHEAP) && defined(RT_USING_MEMHEAP_AS_HEAP)

/* 从外部 SDRAM 里面分配一部分静态内存来作为 rtt 的堆空间，这里配置为 30MB */
#define SDRAM_BEGIN         (0x80000000u)
#define SDRAM_END           (0x81D00000u)
#define LCD_FB_BASE         (0x81D10000u)

#elif defined(RT_USING_SMALL_MEM)
/* 从内部 SRAM (即 DTCM) 里面分配一部分静态内存来作为 rtt 的堆空间，这里配置为 4KB */
#define RT_HEAP_SIZE 1024
static uint32_t rt_heap[RT_HEAP_SIZE];
```

(下页继续)

(续上页)

```
RT_WEAK void *rt_heap_begin_get(void)
{
    return rt_heap;
}

RT_WEAK void *rt_heap_end_get(void)
{
    return rt_heap + RT_HEAP_SIZE;
}
#endif

#endif

/**
* @brief  开发板硬件初始化函数
* @param  无
* @retval 无
*
* @attention
* RTT 把开发板相关的初始化函数统一放到 board.c 文件中实现,
* 当然, 你想把这些函数统一放到 main.c 文件也是可以的。
*/
void rt_hw_board_init(void)                          //(2)
{
    /* 初始化内存管理单元 */
    BOARD_ConfigMPU();

    /* 初始化开发板引脚 */
    BOARD_InitPins();

    /* 初始化开发板时钟 */
    BOARD_BootClockRUN();
```

(下页继续)

(续上页)

```

    /* 初始化调试串口 */
    BOARD_InitDebugConsole();
    PRINTF("***** 欢迎使用 野火 i.MX RT1052 开发板 *****\r\n\n\n\n\n");
    LED_GPIO_Config();
    /* 将开发板硬件相关的初始化放上面 */

    /* 初始化 SysTick */
    SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );


    /* Call components board initial (use INIT_BOARD_EXPORT()) */

ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();                    // (2)-□
endif

if defined(RT_USING_CONSOLE) && defined(RT_USING_DEVICE)
    rt_console_set_device(RT_CONSOLE_DEVICE_NAME);    // (2)-□
endif

ifdef RT_USING_HEAP
if defined(RT_USING_MEMHEAP) && defined(RT_USING_MEMHEAP_AS_HEAP)
PRINTF("sdram heap, begin: 0x%p, end: 0x%p\n", SDRAM_BEGIN, SDRAM_END);    //
↪ (2)-□
rt_system_heap_init((void *)SDRAM_BEGIN, (void *)SDRAM_END);
elif defined(RT_USING_SMALL_MEM)
PRINTF("sram heap, begin: 0x%p, end: 0x%p\n", HEAP_BEGIN, HEAP_END);
rt_system_heap_init((void *)rt_heap_begin_get, (void *)rt_heap_end_get);
#endif
#endif
}
```

(下页继续)

(续上页)

```


/**
* @brief  SysTick 中断服务函数
* @param  无
* @retval 无
*
* @attention
*
*/                                      //(3)
void SysTick_Handler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();

    rt_tick_increase();

    /* leave interrupt */
    rt_interrupt_leave();
}
```

代码清单 13-3(1)：RT-Thread 堆配置，如果同时定义了 RT_USING_MEMHEAP 和 RT_USING_MEMHEAP_AS_HEAP 这两个宏，表示 RT-Thread 里面创建内核对象时使用动态内存分配方案。堆可以是内部的 SRAM 也可以是外部的 SRAM 或 SDRAM，目前的方法是从外部 SDRAM 里面分配一部分静态内存来作为堆空间，这里配置为 30MB。rt_heap_begin_get() 和 rt_heap_end_get() 这两个函数表示堆的起始地址和结束地址。这两个函数前面的宏 RT_WEAK 的原型是关键字 __weak，表示若定义，即其它地方定义了 rt_heap_begin_get() 和 rt_heap_end_get() 这两个函数实体，被 __weak 修饰的函数就会被覆盖。

RT_USING_USER_MAIN 和 RT_USING_HEAP 这两个宏在 rtconfig.h 中定义，RT_USING_USER_MAIN 默认使能，通过使能或者失能 RT_USING_HEAP 这个宏来选择使用静态或者动态内存。无论是使用静态还是动态内存方案，使用的都是内部的 SRAM，区别是

使用的内存是在程序编译的时候分配还是在运行的时候分配。

16.6.1 rt_hw_board_init() 函数

代码清单 13-3(**2**)：RT-Thread 启动的时候会调用一个名为 rt_hw_board_init() 的函数，从函数名称我们可以知道它是用来初始化开发板硬件的，比如时钟，比如串口等，具体初始化什么由用户选择。当这些硬件初始化好之后，RT-Thread 才继续往下启动。至于 RT-Thread 是哪个文件里面的哪个函数会调用 rt_hw_board_init()，我们在本章先不细讲，留到接下来的“RT-Thread 的启动流程”章节再深究，这里我们只需要知道我们用户要自己编写一个 rt_hw_board_init() 的函数供 RT-Thread 启动的时候调用即可。

代码清单 13-3(**2**)**-** □：这部分是 RT-Thread 为开发板组件提供的一个初始化函数，该函数在 components.c 里面实现，由 rtconfig.h 里面的宏 RT_USING_COMPONENTS_INIT 决定是否调用，默认是开启。

代码清单 13-3(**2**)**-** □：rt_console_set_device() 是 RT- Thread 提供的一个控制台设置函数，它将指定 rt_kprintf() 函数的输出内容具体从什么设备打印出来。该函数在 kservice.c 里面实现，由 rtconfig.h 里面的 RT_USING_CONSOLE 和 RT_USING_DEVICE 这两个宏决定是否调用，目前我们暂时不用。

代码清单 13-3(**2**)**-** □：rt_system_heap_init() 是 RT-Thread 提供的一个内存初始化函数，只有在使用 RT-Thread 提供的动态内存分配函数时才需要使用到。该函数在 mem.c 里面实现，由 rtconfig.h 里面的 RT_USING_HEAP 和 RT_USING_USER_MAIN 这两个决定是否调用，目前我们暂时不用。

16.6.2 SysTick_Handler() 函数

代码清单 13-3(**3**)：SysTick 中断服务函数是一个非常重要的函数，RT-Thread 所有跟时间相关的事情都在里面处理，具体实现见*代码清单 13-4*。

列表 4: 代码清单 13-4 SysTick_Handler() 函数

```
/**
* @brief  SysTick 中断服务函数
```

(下页继续)

(续上页)

```
* @param  无
* @retval 无
*
*/




void SysTick_Handler(void)
{
    /* 进入中断 */
    rt_interrupt_enter();                    (1)

    /* 更新时基 */
    rt_tick_increase();                      (2)

    /* 离开中断 */
    rt_interrupt_leave();                    (3)
}
```

代码清单 13-4 **(1)**：进入中断，对中断计数器 rt_interrupt_nest 加 1 操作。

代码清单 13-4**(2)**：rt_tick_increase() 用于更新时基，实现时间片，扫描系统定时器。

代码清单 13-4**(3)**：退出中断，对中断计数器 rt_interrupt_nest 减 1 操作。

16.6.2.1 board.c 文件修改

board.c 文件内容修改的并不多，具体见代码清单 13-5。

列表 5: 代码清单 13-5 board.c 文件修改

```
#include "board.h"                    //修改 (1)
#include "fsl_common.h"
#include "fsl_debug_console.h"
#include "board.h"
#include "pin_mux.h"

#include <rthw.h>
#include <rtthread.h>
#include "./led/bsp_led.h"
#include "fsl_lpuart.h"
#if 0                                 //修改 (2)
#define _SCB_BASE       (0xE000E010UL)
#define _SYSTICK_CTRL   (*(rt_uint32_t *)(_SCB_BASE + 0x0))
#define _SYSTICK_LOAD   (*(rt_uint32_t *)(_SCB_BASE + 0x4))
#define _SYSTICK_VAL    (*(rt_uint32_t *)(_SCB_BASE + 0x8))
#define _SYSTICK_CALIB  (*(rt_uint32_t *)(_SCB_BASE + 0xC))
#define _SYSTICK_PRI    (*(rt_uint8_t  *)(0xE000ED23UL))

// Updates the variable SystemCoreClock and must be called
// whenever the core clock is changed during program execution.
extern void SystemCoreClockUpdate(void);

// Holds the system core clock, which is the system clock
// frequency supplied to the SysTick timer and the processor
// core clock.
extern uint32_t SystemCoreClock;

static uint32_t _SysTick_Config(rt_uint32_t ticks)
{
    if ((ticks - 1) > 0xFFFFFF) {
```

(下页继续)

(续上页)

```
        return 1;
    }

    _SYSTICK_LOAD = ticks - 1;
    _SYSTICK_PRI = 0xFF;
    _SYSTICK_VAL  = 0;
    _SYSTICK_CTRL = 0x07;

    return 0;
}
#endif
/*****************************************************************************
* Variables
*****************************************************************************/

/*****************************************************************************
* Code
*****************************************************************************/

/* Get debug console frequency. */
uint32_t BOARD_DebugConsoleSrcFreq(void)//增加板级系统时钟相关初始化
{
    /*
    省略配置时钟内容
    */
}

/* Initialize debug console. */
void BOARD_InitDebugConsole(void)
{
    uint32_t uartClkSrcFreq = BOARD_DebugConsoleSrcFreq();

```

(下页继续)

(续上页)

```
    DbgConsole_Init(BOARD_DEBUG_UART_BASEADDR, BOARD_DEBUG_UART_BAUDRATE, BOARD_DEBUG_UART_TYPE, uartClkSrcFreq);
}

/* MPU configuration. */
void BOARD_ConfigMPU(void)
{
    /*
    省略配置 MPU 内容
    */
}


#if 0   //修改 (3)
#if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
#define RT_HEAP_SIZE 1024
static uint32_t rt_heap[RT_HEAP_SIZE];  // heap default size: 4K(1024 * 4)
RT_WEAK void *rt_heap_begin_get(void)
{
    return rt_heap;
}

RT_WEAK void *rt_heap_end_get(void)
{
    return rt_heap + RT_HEAP_SIZE;
}
#endif
#endif
#ifdef RT_USING_HEAP

#if defined(RT_USING_MEMHEAP) && defined(RT_USING_MEMHEAP_AS_HEAP)
/* 从外部 SDRAM 里面分配一部分静态内存来作为 rtt 的堆空间，这里配置为 30MB */
```

(下页继续)

(续上页)

```
#define SDRAM_BEGIN          (0x80000000u)
#define SDRAM_END            (0x81D00000u)
#define LCD_FB_BASE          (0x81D10000u)

#elif defined(RT_USING_SMALL_MEM)
/* 从内部 SRAM (即 DTCM) 里面分配一部分静态内存来作为 rtt 的堆空间, 这里配置为 4KB */
#define RT_HEAP_SIZE 1024
static uint32_t rt_heap[RT_HEAP_SIZE];
RT_WEAK void *rt_heap_begin_get(void)
{
    return rt_heap;
}

RT_WEAK void *rt_heap_end_get(void)
{
    return rt_heap + RT_HEAP_SIZE;
}
#endif

#endif
/**
* This function will initial your board.
*/
void rt_hw_board_init()
{
#if 0                //修改 (4)
    /* System Clock Update */
    SystemCoreClockUpdate();

    /* System Tick Configuration */
    _SysTick_Config(SystemCoreClock / RT_TICK_PER_SECOND);
#endif
```

(下页继续)

(续上页)

```
    //    /* 初始化内存管理单元 */
    /* 初始化内存管理单元 */
    BOARD_ConfigMPU();

    /* 初始化开发板引脚 */
    BOARD_InitPins();

    /* 初始化开发板时钟 */
    BOARD_BootClockRUN();

    /* 初始化调试串口 */
    BOARD_InitDebugConsole();
    PRINTF("***** 欢迎使用 野火 i.MX RT1052 开发板 *****\r\n");
    LED_GPIO_Config();
    /* 将开发板硬件相关的初始化放上面 */

    /* 初始化 SysTick */
    SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );
    /* Call components board initial (use INIT_BOARD_EXPORT()) */
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif

#if defined(RT_USING_CONSOLE) && defined(RT_USING_DEVICE)
    rt_console_set_device(RT_CONSOLE_DEVICE_NAME);
#endif
#if 0
#if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
    rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get());
#endif
#endif

```

(下页继续)

(续上页)

```
#ifdef RT_USING_HEAP
#if defined(RT_USING_MEMHEAP) && defined(RT_USING_MEMHEAP_AS_HEAP)
    //PRINTF("sdram heap, begin: 0x%p, end: 0x%p\n", SDRAM_BEGIN, SDRAM_END);
    rt_system_heap_init((void *)SDRAM_BEGIN, (void *)SDRAM_END);
#elif defined(RT_USING_SMALL_MEM)
    PRINTF("sram heap, begin: 0x%p, end: 0x%p\n", HEAP_BEGIN, HEAP_END);
    rt_system_heap_init((void *)rt_heap_begin_get, (void *)rt_heap_end_get);
#endif
#endif
}
void SysTick_Handler(void)
{
    /* enter interrupt */
    rt_interrupt_enter();

    rt_tick_increase();

    /* leave interrupt */
    rt_interrupt_leave();
}
```

代码清单 13-5**修改 (1)**：在 user 目录下新建一个 board.h 头文件，用来包含固件库和 BSP 相关的头文件和存放 board.c 里面的函数声明。并且添加原有的头文件。

代码清单 13-5**修改 (2)**：SysTick 相关的寄存器和初始化函数统统屏蔽掉，将由固件库文件 core_cm3/4/7 里面的替代。并增加现有板子的 SysTick 的配置。

代码清单 13-5**修改 (3)**：将原有的堆地址空间全部注释掉，从新定义新的堆空间起始地址及基地址。

代码清单 13-5**修改 (4)**：增加板级驱动。

16.7 修改 main.c

我们将原来裸机工程里面 main.c 的文件内容全部删除，新增如下内容，具体见*代码清单 13-6*。

列表 6: 代码清单 13-6 main.c 文件内容

```c
/**
*****************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2019-xx-xx
* @brief   RT-Thread 系统 工程模板
*****************************************************************
* @attention
*
* 实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
******************************************************************
*/

/*
*******************************************************************
*                              包含的头文件
*******************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"
#include "rtthread.h"

/*
```

(下页继续)

(续上页)

```
**************************************************************************
*                                 变量
**************************************************************************
*/


/*
**************************************************************************
*                               函数声明
**************************************************************************
*/



/*
**************************************************************************
*                               main 函数
**************************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /* 暂时没有在 main 线程里面创建线程应用线程 */
}


/******************************END OF FILE****************************/
```

16.8 下载验证

将程序编译好，用 DAP 仿真器把程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），一看，啥现象都没有，一脸懵逼，我说，你急个肾，目前我们还没有在 main 线程里面创建应用线程，但是系统是已经跑起来了，只有默认的空闲线程和 main 线程。要想看现象，得自己在 main 创建里面应用线程，如果创建线程，请看下一章“创建线程”。

第 17 章　创建线程

在上一章，我们已经基于野火 i.MX RT 开发板创建好了 RT-Thread 的工程模板，这章开始我们将真正进入如何使用 RT-Thread 的征程，先从最简单的创建线程开始，点亮一个 LED，以慰藉下尔等初学者弱小的心灵。

17.1 硬件初始化

本章创建的线程需要用到开发板上的 LED，所以先要将 LED 相关的函数初始化好，具体是在 board.c 的 rt_hw_board_init() 函数中初始化，具体见*代码清单 14-1* 的高亮部分。

列表 1: 代码清单 14-1rt_hw_board_init() 中添加硬件初始化函数

```
/**
* @brief  开发板硬件初始化函数
* @param  无
* @retval 无
*
* @attention
* RTT 把开发板相关的初始化函数统一放到 board.c 文件中实现，
* 当然，你想把这些函数统一放到 main.c 文件也是可以的。
*/
void rt_hw_board_init(void)
{
    /* 初始化内存管理单元 */
    BOARD_ConfigMPU();

    /* 初始化开发板引脚 */
```

(下页继续)

(续上页)

```
    BOARD_InitPins();

    /* 初始化开发板时钟 */
    BOARD_BootClockRUN();

    /* 初始化调试串口 */
    BOARD_InitDebugConsole();
    PRINTF("***** 欢迎使用 野火 i.MX RT1052 开发板 *****\r\n\n\n\n\n");

    /* 将开发板硬件相关的初始化放上面 */

    /* 初始化 SysTick */
    SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );
    /* 初始化开发板的 LED */
    LED_GPIO_Config();

    /* Call components board initial (use INIT_BOARD_EXPORT()) */
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif

#if defined(RT_USING_CONSOLE) && defined(RT_USING_DEVICE)
    rt_console_set_device(RT_CONSOLE_DEVICE_NAME);
#endif

#ifdef RT_USING_HEAP
#if defined(RT_USING_MEMHEAP) && defined(RT_USING_MEMHEAP_AS_HEAP)
    PRINTF("sdram heap, begin: 0x%p, end: 0x%p\n", SDRAM_BEGIN, SDRAM_END);
    rt_system_heap_init((void *)SDRAM_BEGIN, (void *)SDRAM_END);
#elif defined(RT_USING_SMALL_MEM)
    PRINTF("sram heap, begin: 0x%p, end: 0x%p\n", HEAP_BEGIN, HEAP_END);
    rt_system_heap_init((void *)rt_heap_begin_get, (void *)rt_heap_end_get);
```

(下页继续)

(续上页)

```
#endif
#endif
}
```

执行到 rt_hw_board_init() 函数的时候，操作系统完全都还没有涉及到，即 rt_hw_board_init() 函数所做的工作跟我们以前编写的裸机工程里面的硬件初始化工作是一模一样的。运行完 rt_hw_board_init() 函数，接下来才慢慢启动操作系统，最后运行创建好的线程。有时候线程创建好，整个系统跑起来了，可想要的实验现象就是出不来，比如 LED 不会亮，串口没有输出，LCD 没有显示等等。如果是初学者，这个时候就会心急如焚，四处求救，那怎么办？这个时候如何排除是硬件的问题还是系统的问题，这里面有个小小的技巧，即在硬件初始化好之后，顺便测试下硬件，测试方法跟裸机编程一样，具体实现见*代码清单 14-2* 的高亮部分。

列表 2: 代码清单 14-2 rt_hw_board_init() 中添加硬件测试函数

```
/**
* @brief  开发板硬件初始化函数
* @param  无
* @retval 无
*
* @attention
* RTT 把开发板相关的初始化函数统一放到 board.c 文件中实现，
* 当然，你想把这些函数统一放到 main.c 文件也是可以的。
*/
void rt_hw_board_init(void)
{
    /* 初始化内存管理单元 */
    BOARD_ConfigMPU();

    /* 初始化开发板引脚 */
    BOARD_InitPins();

```

(下页继续)

(续上页)

```
    /* 初始化开发板时钟 */
    BOARD_BootClockRUN();

    /* 初始化调试串口 */
    BOARD_InitDebugConsole();
    PRINTF("***** 欢迎使用 野火 i.MX RT1052 开发板 *****\r\n\n\n\n\n");

    /* 将开发板硬件相关的初始化放上面 */

    /* 初始化 SysTick */
    SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );
    /* 初始化开发板的 LED */
    LED_GPIO_Config();                    //(1)

    /* 测试硬件是否正常工作 */
    LED1_ON;

    /* 其它硬件初始化和测试 */                //(2)

    /* 让程序停在这里，不再继续往下执行 */    //(3)
    while (1);


    /* Call components board initial (use INIT_BOARD_EXPORT()) */
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif

#if defined(RT_USING_CONSOLE) && defined(RT_USING_DEVICE)
    rt_console_set_device(RT_CONSOLE_DEVICE_NAME);
#endif

```

(下页继续)

(续上页)

```
#ifdef RT_USING_HEAP
#if defined(RT_USING_MEMHEAP) && defined(RT_USING_MEMHEAP_AS_HEAP)
    PRINTF("sdram heap, begin: 0x%p, end: 0x%p\n", SDRAM_BEGIN, SDRAM_END);
    rt_system_heap_init((void *)SDRAM_BEGIN, (void *)SDRAM_END);
#elif defined(RT_USING_SMALL_MEM)
    PRINTF("sram heap, begin: 0x%p, end: 0x%p\n", HEAP_BEGIN, HEAP_END);
    rt_system_heap_init((void *)rt_heap_begin_get, (void *)rt_heap_end_get);
#endif
#endif
}
```

代码清单 14-2(**1**)：初始化硬件后，顺便测试硬件，看下硬件是否正常工作。

代码清单 14-2(**2**)：可以继续添加其它的硬件初始化和测试。硬件确认没有问题之后，硬件测试代码可删可不删，因为 rt_hw_board_init() 函数只执行一遍。

代码清单 14-2(**3**)：方便测试硬件好坏，让程序停在这里，不再继续往下执行，当测试完毕后，这个 while(1); 必须删除。

17.2 创建单线程—SRAM 静态内存

这里，我们创建一个单线程，线程使用的栈和线程控制块都使用静态内存，即预先定义好的全局变量，这些预先定义好的全局变量都存在内部的 SRAM 中。

17.2.1 定义线程函数

线程实际上就是一个无限循环且不带返回值的 C 函数。目前，我们创建一个这样的线程，让开发板上面的 LED 灯以 500ms 的频率闪烁，具体实现见*代码清单 14-3*。

列表 3: 代码清单 14-3 20.6.1 定义线程函数

```
static void led1_thread_entry(void* parameter)
{
    while (1)
    {
        LED1_ON;
        rt_thread_delay(500);   /* 延时 500 个 tick */      (1)

        LED1_OFF;
        rt_thread_delay(500);   /* 延时 500 个 tick */

    }
}
```

代码清单 14-3(**1**)：线程里面的延时函数必须使用 RT-Thread 里面提供的延时函数，并不能使用我们裸机编程中的那种延时。这两种的延时的区别是 RT-Thread 里面的延时是阻塞延时，即调用 rt_thread_delay() 函数的时候，当前线程会被挂起，调度器会切换到其它就绪的线程，从而实现多线程。如果还是使用裸机编程中的那种延时，那么整个线程就成为了一个死循环，如果恰好该线程的优先级是最高的，那么系统永远都是在这个线程中运行，根本无法实现多线程。

目前我们只创建了一个线程，当线程进入延时的时候，因为没有另外就绪的用户线程，那么系统就会进入空闲线程，空闲线程是 RT-Thread 系统自己启动的一个线程，优先级最低。当整个系统都没有就绪线程的时候，系统必须保证有一个线程在运行，空闲线程就是为这个设计的。当用户线程延时到期，又会从空闲线程切换回用户线程。

17.2.2 定义线程栈

在 RT-Thread 系统中，每一个线程都是独立的，他们的运行环境都单独的保存在他们的栈空间当中。那么在定义好线程函数之后，我们还要为线程定义一个栈，目前我们使用的是静态内存，所以线程栈是一个独立的全局变量，具体见*代码清单 14-4*。线程的栈占用的是 MCU 内部的 RAM，当线程越多的时候，需要使用的栈空间就越大，即需要使用的 RAM 空间就越多。一个 MCU 能

够支持多少线程，就得看你的 RAM 空间有多少。

列表 4: 代码清单 14-4 定义线程栈

```
/* 定义线程控栈时要求 RT_ALIGN_SIZE 个字节对齐 */
ALIGN(RT_ALIGN_SIZE)
/* 定义线程栈 */
static rt_uint8_t rt_led1_thread_stack[1024];
```

在大多数系统中需要做栈空间地址对齐，例如在 ARM 体系结构中需要向 4 字节地址对齐。实现栈对齐的方法为，在定义栈之前，放置一条 ALIGN(RT_ALIGN_SIZE) 语句，指定接下来定义的变量的地址对齐方式。其中 ALIGN 是在 rtdef.h 里面定义的一个宏，根据编译器不一样，该宏的具体定义是不一样的，在 ARM 编译器中，该宏的定义具体见*代码清单 14-5*。ALIGN 宏的形参 RT_ALIGB_SIZE 是在 rtconfig.h 中的一个宏，目前定义为 4。

列表 5: 代码清单 14-5ALIGN 宏定义

```
/* 只针对 ARM 编译器，在其它编译器，该宏的实现会不一样 */
#define ALIGN(n) __attribute__((aligned(n)))
```

17.2.3 定义线程控制块

定义好线程函数和线程栈之后，我们还需要为线程定义一个线程控制块，通常我们称这个线程控制块为线程的身份证。在 C 代码上，线程控制块就是一个结构体，里面有非常多的成员，这些成员共同描述了线程的全部信息，具体见*代码清单 14-6*。

列表 6: 代码清单 14-6 定义线程控制块

```
/* 定义线程控制块 */
static struct rt_thread led1_thread;
```

17.2.4 初始化线程

一个线程的三要素是线程主体函数，线程栈，线程控制块，那么怎么样把这三个要素联合在一起？RT-Thread 里面有一个叫线程初始化函数 rt_thread_init()，它就是干这个活的。它将线程主体函数，线程栈（静态的）和线程控制块（静态的）这三者联系在一起，让线程可以随时被系统启动，具体见*代码清单 14-7*。

列表 7: 代码清单 14-7 初始化线程

```
rt_thread_init(&led1_thread,                    /* 线程控制块 */      (1)
               "led1",                          /* 线程名字 */        (2)
               led1_thread_entry,               /* 线程入口函数 */    (3)
               RT_NULL,                         /* 线程入口函数参数 */ (4)
               &rt_led1_thread_stack[0],        /* 线程栈起始地址 */  (5)
               sizeof(rt_led1_thread_stack),    /* 线程栈大小 */      (6)
               3,                               /* 线程的优先级 */    (7)
               20);                             /* 线程时间片 */      (8)
```

代码清单 14-7(**1**)：线程控制块指针，在使用静态内存的时候，需要给线程初始化函数 rt_thread_init() 传递预先定义好的线程控制块的指针。在使用动态内存的时候，线程创建函数 rt_thread_create() 会返回一个指针指向线程控制块，该线程控制块是 rt_thread_create() 函数里面动态分配的一块内存。

代码清单 14-7(**2**)：线程名字，字符串形式，最大长度由 rtconfig.h 中定义的 RT_NAME_MAX 宏指定，多余部分会被自动截掉。

代码清单 14-7(**3**)：线程入口函数，即线程函数的名称。

代码清单 *14-7*(**4**)：线程入口函数形参，不用的时候配置为 0 即可。

代码清单 *14-7*(**5**)：线程栈起始地址，只有在使用静态内存的时候才需要提供，在使用动态内存的时候会根据提供的线程栈大小自动创建。

代码清单 *14-7*(**6**)：线程栈大小，单位为字节。

代码清单 *14-7*(**7**)：线程的优先级。优先级范围根据 rtconfig.h 中的宏 RT_THREAD_PRIORITY_MAX 决定，最多支持 256 个优先级，目前配置为 32。在 RT-Thread 中，数值越小优先级越高，0 代表最高优先级。

代码清单 *14-7*(**8**)：线程的时间片大小。时间片的单位是操作系统的时钟节拍。当系统中存在相同优先级线程时，这个参数指定线程一次调度能够运行的最大时间长度。这个时间片运行结束时，调度器自动选择下一个就绪态的同优先级线程进行运行。如果同一个优先级下只有一个线程，那么时间片这个形参就不起作用。

17.2.5 启动线程

当线程初始化好后，是处于线程初始态（RT_THREAD_INIT），并不能够参与操作系统的调度，只有当线程进入就绪态（RT_THREAD_READY）之后才能参与操作系统的调度。线程由初始态进入就绪态可由函数 rt_thread_startup() 来实现，具体见代码清单 *14-8*。

列表 8: 代码清单 14-8 启动线程

```
/* 启动线程，开启调度 */
rt_thread_startup(&led1_thread);
```

17.2.6 main.c 文件内容全貌

现在我们把线程主体，线程栈，线程控制块这三部分代码统一放到 main.c 中，具体内容见代码清单 *14-9*。

列表 9: 代码清单 14-9 main.c 文件内容全貌

```
/*
*************************************************************************
*                                 包含的头文件
*************************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"
/* RT-Thread 头文件 */
#include "rtthread.h"
/*
*************************************************************************
*                                   变量
*************************************************************************
*/
/* 定义线程控制块 */
static struct rt_thread led1_thread;

/* 定义线程控栈时要求 RT_ALIGN_SIZE 个字节对齐 */
ALIGN(RT_ALIGN_SIZE)
/* 定义线程栈 */
static rt_uint8_t rt_led1_thread_stack[1024];
/*
*************************************************************************
*                                 函数声明
*************************************************************************
*/
static void led1_thread_entry(void* parameter);


/*
```

(下页继续)

(续上页)

```
*****************************************************************************
*                                main 函数
*****************************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */

    rt_thread_init(&led1_thread,                    /* 线程控制块 */
                   "led1",                          /* 线程名字 */
                   led1_thread_entry,               /* 线程入口函数 */
                   RT_NULL,                         /* 线程入口函数参数 */
                   &rt_led1_thread_stack[0],        /* 线程栈起始地址 */
                   sizeof(rt_led1_thread_stack),    /* 线程栈大小 */
                   3,                               /* 线程的优先级 */
                   20);                             /* 线程时间片 */
    rt_thread_startup(&led1_thread);                /* 启动线程，开启调度 */
}

/*
*****************************************************************************
*                                线程定义
*****************************************************************************
```

(下页继续)

(续上页)

```
*/

static void led1_thread_entry(void* parameter)
{
    while (1)
    {
        LED1_ON;
        rt_thread_delay(500);   /* 延时 500 个 tick */

        LED1_OFF;
        rt_thread_delay(500);   /* 延时 500 个 tick */

    }
}

/******************************END OF FILE****************************/
```

17.3 下载验证

将程序编译好，用 DAP 仿真器把程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），可以看到板子上面的 LED 灯已经在闪烁，说明我们创建的单线程（使用静态内存）已经跑起来了。

在当前这个例程，线程的栈，线程的控制块用的都是静态内存，必须由用户预先定义，这种方法我们在使用 RT-Thread 的时候用的比较少，通常的方法是在线程创建的时候动态的分配线程栈和线程控制块的内存空间，接下来我们讲解下“创建单线程—SRAM 动态内存”的方法。

17.4 创建单线程—SRAM 动态内存

这里，我们创建一个单线程，线程使用的栈和线程控制块是在创建线程的时候 RT-Thread 动态分配的，并不是预先定义好的全局变量。那这些动态的内存堆是从哪里来？继续往下看。

17.4.1 动态内存空间的堆从哪里来

在创建单线程—SRAM 静态内存的例程中，线程控制块和线程栈的内存空间都是从内部的 SRAM 里面分配的，具体分配到哪个地址由编译器决定。现在我们开始使用动态内存，即堆，其实堆也是内存，也属于 SRAM。现在我们的做法是在 SRAM 里面定义一个大数组供 RT-Thread 的动态内存分配函数使用，这些代码在 board.c 开头实现，具体见*代码清单 14-10*。

列表 10: 代码清单 14-10 定义 RT-Thread 的堆到内部 SRAM

```
#if defined(RT_USING_MEMHEAP) && defined(RT_USING_MEMHEAP_AS_HEAP)//(1)
/* 从外部 SDRAM 里面分配一部分静态内存来作为 rtt 的堆空间，这里配置为 30MB */
#define SDRAM_BEGIN          (0x80000000u)
#define SDRAM_END            (0x81D00000u)
#define LCD_FB_BASE          (0x81D10000u)

#elif defined(RT_USING_SMALL_MEM)
/* 从内部 SRAM (即 DTCM) 里面分配一部分静态内存来作为 rtt 的堆空间，这里配置为 4KB */
#define RT_HEAP_SIZE 1024
static uint32_t rt_heap[RT_HEAP_SIZE]; //(2)
RT_WEAK void *rt_heap_begin_get(void)  //(3)
{
    return rt_heap;
}

RT_WEAK void *rt_heap_end_get(void)    //(4)
{
```

(下页继续)

(续上页)

```
    return rt_heap + RT_HEAP_SIZE;
}
#endif


/**
* @brief  开发板硬件初始化函数
* @param  无
* @retval 无
*
* @attention
* RTT 把开发板相关的初始化函数统一放到 board.c 文件中实现,
* 当然, 你想把这些函数统一放到 main.c 文件也是可以的。
*/
void rt_hw_board_init(void)
{
    /* 初始化内存管理单元 */
    BOARD_ConfigMPU();

    /* 初始化开发板引脚 */
    BOARD_InitPins();

    /* 初始化开发板时钟 */
    BOARD_BootClockRUN();

    /* 初始化调试串口 */
    BOARD_InitDebugConsole();
    PRINTF("***** 欢迎使用 野火 i.MX RT1052 开发板 *****\r\n\n\n\n\n");
    LED_GPIO_Config();
    /* 将开发板硬件相关的初始化放上面 */

    /* 初始化 SysTick */
```

(下页继续)

(续上页)

```
    SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );


    /* Call components board initial (use INIT_BOARD_EXPORT()) */
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif

#if defined(RT_USING_CONSOLE) && defined(RT_USING_DEVICE)
    rt_console_set_device(RT_CONSOLE_DEVICE_NAME);
#endif

#ifdef RT_USING_HEAP
#if defined(RT_USING_MEMHEAP) && defined(RT_USING_MEMHEAP_AS_HEAP)
    PRINTF("sdram heap, begin: 0x%p, end: 0x%p\n", SDRAM_BEGIN, SDRAM_END);
    rt_system_heap_init((void *)SDRAM_BEGIN, (void *)SDRAM_END);
#elif defined(RT_USING_SMALL_MEM)
    PRINTF("sram heap, begin: 0x%p, end: 0x%p\n", HEAP_BEGIN, HEAP_END);
    rt_system_heap_init((void *)rt_heap_begin_get, (void *)rt_heap_end_get);
↪ //(5)

#endif
#endif
}
```

代码清单 14-10 **(1)**：RT_USING_USER_MAIN 和 RT_USING_HEAP 这两个宏，在 rtconfig.h 定义，RT_USING_USER_MAIN 默认开启，RT_USING_HEAP 在使用动态内存时需要开启。

代码清单 14-10**(2)**：从内部 SDRAM 里面定义一个静态数组 rt_heap，大小由 RT_HEAP_SIZE 这个宏决定，目前定义为 30MB。定义的堆大小不能超过内部 SDRAM 的总大小。

代码清单 14-10**(3)**：rt_heap_begin_get() 用于获取堆的起始地址。

代码清单 14-10**(4)**：rt_heap_end_get() 用于获取堆的结束地址。

代码清单 14-10(5) ：rt_system_heap_init() 根据堆的起始地址和结束地址进行堆的初始化。rt_system_heap_init() 需要两个形参，一个是堆的起始地址，另外一个是堆的结束地址，如果我们使用外部 SDRAM 作为堆，这两个形参直接传入外部 SDRAM 地址范围内的地址即可。

17.4.2 定义线程函数

使用动态内存的时候，线程的主体函数与使用静态内存时是一样的，具体见代码清单 14-11。

列表 11: 代码清单 14-11 定义线程函数

```
static void led1_thread_entry(void* parameter)
{
    while (1)
    {
        LED1_ON;
        rt_thread_delay(500);   /* 延时 500 个 tick */

        LED1_OFF;
        rt_thread_delay(500);   /* 延时 500 个 tick */

    }
}
```

17.4.3 定义线程栈

使用动态内存的时候，线程栈在线程创建的时候创建，不用跟使用静态内存那样要预先定义好一个全局的静态的栈空间。

17.4.4 定义线程控制块指针

使用动态内存时候，不用跟使用静态内存那样要预先定义好一个全局的静态的线程控制块空间。线程控制块是在线程创建的时候创建，线程创建函数会返回一个指针，用于指向线程控制块，所以要预先为线程栈定义一个线程控制块指针，具体见*代码清单 14-12*。

列表 12: 代码清单 14-12 定义线程控制块指针

```
/* 定义线程控制块指针 */
static  rt_thread_t led1_thread = RT_NULL;
```

17.4.5 创建线程

使用静态内存时，使用 rt_thread_init() 来初始化一个线程，使用动态内存的时，使用 rt_thread_create() 函数来创建一个线程，两者的函数名不一样，具体的形参也有区别，具体见*代码清单 14-13*。

列表 13: 代码清单 14-13 创建线程

```
led1_thread =                                /* 线程控制块指针 */    (1)
rt_thread_create( "led1",                    /* 线程名字 */          (2)
                led1_thread_entry,    /* 线程入口函数 */          (3)
                RT_NULL,              /* 线程入口函数参数 */       (4)
                512,                  /* 线程栈大小 */            (5)
                3,                    /* 线程的优先级 */          (6)
                20);                  /* 线程时间片 */            (7)
```

代码清单 14-13(1)：线程控制块指针，在使用静态内存的时候，需要给线程初始化函数 rt_thread_init() 传递预先定义好的线程控制块的指针。在使用动态内存的时候，线程创建函数 rt_thread_create() 会返回一个指针指向线程控制块，该线程控制块是 rt_thread_create() 函数里面动态分配的一块内存。

代码清单 14-13(2)：线程名字，字符串形式，最大长度由 rtconfig.h 中定义的 RT_NAME_MAX 宏指定，多余部分会被自动截掉。

代码清单 14-13(3)：线程入口函数，即线程函数的名称。

代码清单 14-13(4)：线程入口函数形参，不用的时候配置为 0 即可。

代码清单 14-13(5)：线程栈大小，单位为字节。使用动态内存创建线程时，与使用静态内存线程初始化函数不一样，不再需要提供线程栈的起始地址，只需要知道线程栈的大小即可，因为它是在线程创建时动态分配的。

代码清单 14-13(6)：线程的优先级。优先级范围根据 rtconfig.h 中的宏 RT_THREAD_PRIORITY_MAX 决定，最多支持 256 个优先级，目前配置为 32。在 RT-Thread 中，数值越小优先级越高，0 代表最高优先级。

代码清单 14-13(7)：线程的时间片大小。时间片的单位是操作系统的时钟节拍。当系统中存在相同优先级线程时，这个参数指定线程一次调度能够运行的最大时间长度。这个时间片运行结束时，调度器自动选择下一个就绪态的同优先级线程进行运行。如果同一个优先级下只有一个线程，那么时间片这个形参就不起作用。

17.4.6 启动线程

当线程创建好后，是处于线程初始态（RT_THREAD_INIT），并不能够参与操作系统的调度，只有当线程进入就绪态（RT_THREAD_READY）之后才能参与操作系统的调度。线程由初始态进入就绪态可由函数 rt_thread_startup() 来实现，具体见代码清单 14-14。

列表 14: 代码清单 14-14 启动线程

```
if (led1_thread != RT_NULL)
    rt_thread_startup(led1_thread);          /* 启动线程，开启调度 */
```

(下页继续)

(续上页)

```
else
    return -1;
```

17.4.7 main.c 文件内容全貌

现在我们把线程主体，线程栈指针，线程控制块这三部分代码统一放到 main.c 中，具体见代码清单 14-15。

列表 15: 代码清单 14-15main.c 文件内容全貌

```
#if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
#define RT_HEAP_SIZE 1024
/* 从内部 SRAM 里面分配一部分静态内存来作为 rtt 的堆空间，这里配置为 4KB */
static uint32_t rt_heap[RT_HEAP_SIZE];
RT_WEAK void *rt_heap_begin_get(void)
{
    return rt_heap;
}

RT_WEAK void *rt_heap_end_get(void)
{
    return rt_heap + RT_HEAP_SIZE;
}
#endif

/* 该部分代码截取自函数 rt_hw_board_init() */
#if defined(RT_USING_USER_MAIN) && defined(RT_USING_HEAP)
//rt_system_heap_init((void*)HEAP_BEGIN, (void*)SRAM_END);
rt_system_heap_init(rt_heap_begin_get(), rt_heap_end_get());
#endif

```

(下页继续)

(续上页)

```
/*
*************************************************************************
*                              包含的头文件
*************************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"

#include "rtthread.h"
/*
*************************************************************************
*                                 变量
*************************************************************************
*/
/* 定义线程控制块指针 */
static rt_thread_t led1_thread = RT_NULL;

/*
*************************************************************************
*                              函数声明
*************************************************************************
*/
static void led1_thread_entry(void* parameter);


/*
*************************************************************************
*                              main 函数
*************************************************************************
*/
/**
```

(下页继续)

(续上页)

```
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */

    led1_thread =                           /* 线程控制块指针 */
    rt_thread_create( "led1",               /* 线程名字 */
                      led1_thread_entry,    /* 线程入口函数 */
                      RT_NULL,              /* 线程入口函数参数 */
                      512,                  /* 线程栈大小 */
                      3,                    /* 线程的优先级 */
                      20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (led1_thread != RT_NULL)
        rt_thread_startup(led1_thread);
    else
        return -1;
}

/*
**************************************************************************
*                              线程定义
**************************************************************************
*/
```

(下页继续)

(续上页)

```

static void led1_thread_entry(void* parameter)
{
    while (1)
    {
        LED1_ON;
        rt_thread_delay(50);   /* 延时 50 个 tick */

        LED1_OFF;
        rt_thread_delay(50);   /* 延时 50 个 tick */

    }
}

/*****************************END OF FILE****************************/
```

17.5 下载验证

将程序编译好，用 DAP 仿真器把程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），可以看到板子上面的 LED 灯已经在闪烁，说明我们创建的单线程（使用动态内存）已经跑起来了。在往后的实验中，我们创建内核对象均采用动态内存分配方案。

17.6 创建多线程—SRAM 动态内存

创建多线程只需要按照创建单线程的套路依葫芦画瓢即可，接下来我们创建两个线程，线程 1 让一个 LED 灯闪烁，线程 2 让另外一个 LED 闪烁，两个 LED 闪烁的频率不一样，具体实现见*代码清单 14-16* 的高亮部分，两个线程的优先级不一样。

列表 16: 代码清单 14-16 创建多线程—SRAM 动态内存

```
/*
*************************************************************************
*                              包含的头文件
*************************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"

#include "rtthread.h"
/*
*************************************************************************
*                                 变量
*************************************************************************
*/
/* 定义线程控制块指针 */
static rt_thread_t led1_thread = RT_NULL;
static rt_thread_t led2_thread = RT_NULL;

/*
*************************************************************************
*                              函数声明
*************************************************************************
*/
static void led1_thread_entry(void* parameter);
static void led2_thread_entry(void* parameter);


/*
*************************************************************************
*                              main 函数
```

(下页继续)

(续上页)

```
*****************************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */

    led1_thread =                           /* 线程控制块指针 */
    rt_thread_create( "led1",               /* 线程名字 */
                      led1_thread_entry,    /* 线程入口函数 */
                      RT_NULL,              /* 线程入口函数参数 */
                      512,                  /* 线程栈大小 */
                      3,                    /* 线程的优先级 */
                      20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (led1_thread != RT_NULL)
        rt_thread_startup(led1_thread);
    else
        return -1;

    led2_thread =                           /* 线程控制块指针 */
    rt_thread_create( "led2",               /* 线程名字 */
                      led2_thread_entry,    /* 线程入口函数 */
```

(下页继续)

(续上页)

```
                    RT_NULL,                /* 线程入口函数参数 */
                    512,                    /* 线程栈大小 */
                    4,                      /* 线程的优先级 */
                    20);                    /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (led2_thread != RT_NULL)
        rt_thread_startup(led2_thread);
    else
        return -1;
}

/*
*****************************************************************************
*                                 线程定义
*****************************************************************************
*/

static void led1_thread_entry(void* parameter)
{
    while (1)
    {
        LED1_ON;
        rt_thread_delay(500);   /* 延时 500 个 tick */

        LED1_OFF;
        rt_thread_delay(500);   /* 延时 500 个 tick */

    }
}

static void led2_thread_entry(void* parameter)
```

(下页继续)

(续上页)

```
{
    while (1)
    {
        LED2_ON;
        rt_thread_delay(300);   /* 延时 300 个 tick */

        LED2_OFF;
        rt_thread_delay(300);   /* 延时 300 个 tick */

    }
}
/***************************END OF FILE***************************/
```

目前多线程我们只创建了两个，如果要创建 3 个、4 个甚至更多都是同样的套路，容易忽略的地方是线程栈的大小，每个线程的优先级。大的线程，栈空间要设置大一点，重要的线程优先级要设置的高一点。

17.7 下载验证

将程序编译好，用 DAP 仿真器把程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），可以看到板子上面的两个 LED 灯以不同的频率在闪烁，说明我们创建的单线程（使动态内存）已经跑起来了。在往后的实验中，我们创建内核对象均采用动态内存分配方案。

第 18 章　重映射串口到 rt_kprintf 函数

在 RT-Thread 中，有一个打印函数 rt_kprintf() 供用户使用，方便在调试的时候输出各种信息。如果要想使用 rt_kprintf()，则必须将控制台重映射到 rt_kprintf()，这个控制台可以是串口、CAN、USB、以太网等输出设备，用的最多的就是串口，接下来我们讲解下如何将串口重定向到 rt_kprintf()。

18.1 rt_kprintf() 函数定义

rt_kprintf() 函数在 kservice.c 中实现，是属于内核服务类的函数，具体实现见*代码清单 15-1*。

列表 1: 代码清单 15-1 rt_kprintf() 函数定义

```
/*
* @brief 这个函数用于向控制台打印特定格式的字符串
*
* @param fmt 指定的格式
*/

void rt_kprintf(const char *fmt, ...)
{
va_list args;
rt_size_t length;
static char rt_log_buf[RT_CONSOLEBUF_SIZE];                    (1)

va_start(args, fmt);

/* rt_vsnprintf 的返回值 length 表示按照 fmt 指定的格式写入到 rt_log_buf 的字符长度 */
length = rt_vsnprintf(rt_log_buf, sizeof(rt_log_buf) - 1, fmt, args); (2)
```

(下页继续)

(续上页)

```
/* 如果 length 超过 RT_CONSOLEBUF_SIZE，则进行截短即最多只能输出 RT_CONSOLEBUF_
↪SIZE 个字符 */
if (length > RT_CONSOLEBUF_SIZE - 1)
    length = RT_CONSOLEBUF_SIZE - 1;

/* 使用设备驱动 */
#ifdef RT_USING_DEVICE                                                    (3)
    if (_console_device == RT_NULL)
    {
        rt_hw_console_output(rt_log_buf);
    }
    else
    {
        rt_uint16_t old_flag = _console_device->open_flag;

        _console_device->open_flag |= RT_DEVICE_FLAG_STREAM;
        rt_device_write(_console_device, 0, rt_log_buf, length);
        _console_device->open_flag = old_flag;
    }
#else
    /* 没有使用设备驱动则由 rt_hw_console_output 函数处理，该函数需要用户自己实现 */
    rt_hw_console_output(rt_log_buf);                                     (4)
#endif
    va_end(args);
}
```

代码清单 15-1(1)：先定义一个字符缓冲区，大小由 rt_config.h 中的宏 RT_CONSOLEBUF_SIZE 定义，默认为 128。

代码清单 15-1(2)：调用 rt_vsnprintf 函数，将要输出的字符按照 fmt 指定的格式打印到预先定义好的 rt_log_buf 缓冲区，然后我们将缓冲区的内容输出到控制台就行了，接下来就是选择使用什么控制台。

代码清单 15-1(3)：如果使用设备驱动，则通过设备驱动函数将 rt_log_buf 缓冲区的内容输出到控制台。如果设备控制台打开失败则由 rt_hw_console_output 函数处理，这个函数需要用户单独实现。

代码清单 15-1(4)：不使用设备驱动，rt_log_buf 缓冲区的内容则由 rt_hw_console_output() 函数处理，这个函数需要用户单独实现。

18.2 自定义 rt_hw_console_output 函数

目前，我们不使用 RT-Thread 的设备驱动，那通过 rt_kprintf 输出的内容则由 rt_hw_console_output 函数处理，这个函数需要用户单独实现。其实，实现这个函数也很简单，只需要通过一个控制台将 rt_log_buf 缓冲区的内容发送出去即可，这个控制台可以是 USB、串口、CAN 等，使用的最多的控制台则是串口。这里我们只讲解如何将串口控制台重映射到 rt_kprintf 函数，rt_hw_console_output 函数在 board.c 实现，具体见代码清单 15-2。

列表 2: 代码清单 15-2 重映射串口控制台到 rt_kprintf 函数

```
/**
* @brief  重映射串口 DEBUG_USARTx 到 rt_kprintf() 函数
*   Note: DEBUG_USARTx 是在 bsp_usart.h 中定义的宏，默认使用串口 1
* @param  str: 要输出到串口的字符串
* @retval 无
*
* @attention
*
*/
void rt_hw_console_output(const char *str)
{
    /* 进入临界段 */
    rt_enter_critical();

```

(下页继续)

(续上页)

```
    /* 直到字符串结束 */
    while (*str!='\0')
    {
        /* 换行 */
        if (*str=='\n')
        {
        PUTCHAR('\r');

        }

        PUTCHAR(*str++);

    }

    /* 退出临界段 */
    rt_exit_critical();
}
```

18.3 测试 rt_kprintf 函数

18.3.1 硬件初始化

rt_kprintf 函数输出的控制台使用的是开发板上的串口（野火 i.MX RT 全系列的开发板都板载了 USB 转串口，然后通过跳帽默认接到了 i.MX RT 的串口 1），所以需要先要将裸机的串口驱动添加到工程并在开发环境中指定串口驱动头文件的编译路径，然后在 board.c 的 rt_hw_board_init() 函数中对串口初始化，具体见*代码清单 15-3* 的高亮部分。

列表 3: 代码清单 15-3 在 rt_hw_board_init 中添加串口初始化代码

```
/**
* @brief  开发板硬件初始化函数
* @param  无
* @retval 无
*
* @attention
* RTT 把开发板相关的初始化函数统一放到 board.c 文件中实现,
* 当然, 你想把这些函数统一放到 main.c 文件也是可以的。
*/
void rt_hw_board_init(void)
{
    /* 初始化内存管理单元 */
    BOARD_ConfigMPU();

    /* 初始化开发板引脚 */
    BOARD_InitPins();

    /* 初始化开发板时钟 */
    BOARD_BootClockRUN();

    /* 初始化调试串口 */
    BOARD_InitDebugConsole();
    PRINTF("***** 欢迎使用 野火 i.MX RT1052 开发板 *****\r\n\n\n\n\n");
    LED_GPIO_Config();
    /* 将开发板硬件相关的初始化放上面 */

    /* 初始化 SysTick */
    SysTick_Config( SystemCoreClock / RT_TICK_PER_SECOND );

```

(下页继续)

(续上页)

```

    /* Call components board initial (use INIT_BOARD_EXPORT()) */
#ifdef RT_USING_COMPONENTS_INIT
    rt_components_board_init();
#endif

#if defined(RT_USING_CONSOLE) && defined(RT_USING_DEVICE)
    rt_console_set_device(RT_CONSOLE_DEVICE_NAME);
#endif

#ifdef RT_USING_HEAP
#if defined(RT_USING_MEMHEAP) && defined(RT_USING_MEMHEAP_AS_HEAP)
    PRINTF("sdram heap, begin: 0x%p, end: 0x%p\n", SDRAM_BEGIN, SDRAM_END);
    rt_system_heap_init((void *)SDRAM_BEGIN, (void *)SDRAM_END);
#elif defined(RT_USING_SMALL_MEM)
    PRINTF("sram heap, begin: 0x%p, end: 0x%p\n", HEAP_BEGIN, HEAP_END);
    rt_system_heap_init((void *)rt_heap_begin_get, (void *)rt_heap_end_get);
#endif
#endif
}
```

18.3.2 编写 rt_kprintf 测试代码

当 rt_kprintf 函数对应的输出控制台初始化好之后（在 rt_hw_board_init() 完成），系统接下来会调用函数 rt_show_version() 来打印 RT-Thread 的版本号，该函数在 kservice.c 中实现，具体见*代码清单 15-4*。

列表 4: 代码清单 15-4 rt_show_version 函数实现

```
void rt_show_version(void)
{
```

(下页继续)

(续上页)

```
    rt_kprintf("\n \\ | /\n");
    rt_kprintf("- RT -     Thread Operating System\n");
    rt_kprintf(" / | \\     %d.%d.%d build %s\n",
            RT_VERSION, RT_SUBVERSION, RT_REVISION, __DATE__);
    rt_kprintf(" 2006 - 2018 Copyright by rt-thread team\n");
}
```

我们也可以在线程中用 rt_kprintf 打印一些辅助信息，具体见*代码清单 15-5* 的高亮部分。

列表 5: 代码清单 15-5 使用 rt_kprintf 在线程中打印调试信息

```
static void led1_thread_entry(void* parameter)
{
    while (1)
    {
        LED1_ON;
        rt_thread_delay(500);   /* 延时 500 个 tick */
        rt_kprintf("led1_thread running,LED1_ON\r\n");

        LED1_OFF;
        rt_thread_delay(500);   /* 延时 500 个 tick */
        rt_kprintf("led1_thread running,LED1_OFF\r\n");
    }
}
```

18.3.3 下载验证

将程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息，具体见图 *15-1*。

野火串口调试助手V1.3 技术论坛：www.fireBBS.cn 官网：www.embedFire.com
串口调试功能
端口 COM11
波特率 115200
校验位 None
数据位 8
停止位 1
关闭串口
清空接收区
停止显示
自动清空
十六进制显示
接收转向文件
自动发送周期 1000 毫秒
自动发送
十六进制发送
清空重填
加载文件
手动发送
文件路径 还没选择要发送的文件
串口已开启
接收字节数： 5186
发送字节数：
计数清零
*****欢迎使用 野火i.MX RT1052 开发板*****
\ | /
- RT - Thread Operating System
/ | \ 3.0.3 build Aug 29 2019
2006 - 2018 Copyright by rt-thread team
led1_thread running,LED1_ON
led1_thread running,LED1_OFF
led1_thread running,LED1_ON
led1_thread running,LED1_OFF
led1_thread running,LED1_ON
led1_thread running,LED1_OFF
led1_thread running,LED1_ON

第 19 章　RT-Thread 的启动流程

在目前的 RTOS 中，主要有两种比较流行的启动方式，暂时还没有看到第三种，接下来我将通过伪代码的方式来讲解下这两种启动方式的区别，然后再具体分析下 RT-Thread 的启动流程。

19.1 万事俱备，只欠东风

第一种我称之为万事俱备，只欠东风法。这种方法是在 main 函数中将硬件初始化，RTOS 系统初始化，所有线程的创建这些都弄好，这个我称之为万事都已经准备好。最后只欠一道东风，即启动 RTOS 的调度器，开始多线程的调度，具体的伪代码实现见*代码清单 16-1*。

列表 1: 代码清单 16-1 万事俱备，只欠东风法伪代码实现

```
int main (void)
{
    /* 硬件初始化 */
    HardWare_Init();                                                    (1)

    /* RTOS 系统初始化 */
    RTOS_Init();                                                        (2)

    /* 创建线程 1，但线程 1 不会执行，因为调度器还没有开启 */            (3)
    RTOS_ThreadCreate(Task1);
    /* 创建线程 2，但线程 2 不会执行，因为调度器还没有开启 */
    RTOS_ThreadCreate(Task2);

    /* ...... 继续创建各种线程 */

```

(下页继续)

(续上页)

```
    /* 启动 RTOS，开始调度 */
    RTOS_Start();                                                    (4)
}

void Thread1( void *arg )                                            (5)
{
    while(1)
    {
        /* 线程实体，必须有阻塞的情况出现 */
    }
}

void Thread2( void *arg )                                            (6)
{
    while(1)
    {
        /* 线程实体，必须有阻塞的情况出现 */
    }
}
```

代码清单 16-1 **(1)**：硬件初始化。硬件初始化这一步还属于裸机的范畴，我们可以把需要使用到的硬件都初始化好而且测试好，确保无误。

代码清单 16-1 **(2)**：RTOS 系统初始化。比如 RTOS 里面的全局变量的初始化，空闲线程的创建等。不同的 RTOS，它们的初始化有细微的差别。

代码清单 16-1 **(3)**：创建各种线程。这里把所有要用到的线程都创建好，但还不会进入调度，因为这个时候 RTOS 的调度器还没有开启。

代码清单 16-1 **(4)**：启动 RTOS 调度器，开始线程调度。这个时候调度器就从刚刚创建好的线程中选择一个优先级最高的线程开始运行。

代码清单 16-1 **(5) (6)**：线程实体通常是一个不带返回值的无限循环的 C 函数，函数体必须有阻塞的情况出现，不然线程（如果优先权恰好是最高）会一直在 while 循环里面执行，导致其它线程

没有执行的机会。

19.2 小心翼翼，十分谨慎

第二种我称之为小心翼翼，十分谨慎法。这种方法是在 main 函数中将硬件和 RTOS 系统先初始化好，然后创建一个启动线程后就启动调度器，然后在启动线程里面创建各种应用线程，当所有线程都创建成功后，启动线程把自己删除，具体的伪代码实现见*代码清单 16-2*。

列表 2: 代码清单 16-2 小心翼翼，十分谨慎法伪代码实现

```
int main (void)
{
    /* 硬件初始化 */
    HardWare_Init();                                        (1)

    /* RTOS 系统初始化 */
    RTOS_Init();                                            (2)

    /* 创建一个线程 */
    RTOS_ThreadCreate(AppThreadStart);                      (3)

    /* 启动 RTOS，开始调度 */
    RTOS_Start();                                           (4)
}

/* 起始线程，在里面创建线程 */
void AppThreadStart( void *arg )                            (5)
{
    /* 创建线程 1，然后执行 */
    RTOS_ThreadCreate(Thread1);                             (6)

```

(下页继续)

(续上页)

```
    /* 当线程 1 阻塞时，继续创建线程 2，然后执行 */
    RTOS_ThreadCreate(Thread2);

    /* ...... 继续创建各种线程 */

    /* 当线程创建完成，关闭起始线程 */
    RTOSThreadClose(AppThreadStart);                    (7)
}

void Thread1( void *arg )                               (8)
{
    while (1)
    {
        /* 线程实体，必须有阻塞的情况出现 */
    }
}

void Thread2( void *arg )                               (9)
{
    while (1)
    {
        /* 线程实体，必须有阻塞的情况出现 */
    }
}
```

代码清单 16-2 **(1)**：硬件初始化。来到硬件初始化这一步还属于裸机的范畴，我们可以把需要使用到的硬件都初始化好而且测试好，确保无误。

代码清单 16-2 **(2)**：RTOS 系统初始化。比如 RTOS 里面的全局变量的初始化，空闲线程的创建等。不同的 RTOS，它们的初始化有细微的差别。

代码清单 16-2 **(3)**：创建一个开始线程。然后在这个初始线程里面创建各种应用线程。

代码清单 16-2 **(4)**：启动 RTOS 调度器，开始线程调度。这个时候调度器就去执行刚刚创建好的

初始线程。

代码清单 16-2 **(5)**：我们通常说线程是一个不带返回值的无限循环的 C 函数，但是因为初始线程的特殊性，它不能是无限循环的，只执行一次后就关闭。在初始线程里面我们创建我们需要的各种线程。

代码清单 16-2 **(6)**：创建线程。每创建一个线程后它都将进入就绪态，系统会进行一次调度，如果新创建的线程的优先级比初始线程的优先级高的话，那将去执行新创建的线程，当新的线程阻塞时再回到初始线程被打断的地方继续执行。反之，则继续往下创建新的线程，直到所有线程创建完成。

代码清单 16-2 **(7)**：各种应用线程创建完成后，初始线程自己关闭自己，使命完成。

代码清单 16-2 **(8) (9)**：线程实体通常是一个不带返回值的无限循环的 C 函数，函数体必须有阻塞的情况出现，不然线程（如果优先权恰好是最高）会一直在 while 循环里面执行，其它线程没有执行的机会。

19.3 孰优孰劣

那有关这两种方法孰优孰劣？我暂时没发现，我个人还是比较喜欢使用第一种。ucos 第一种和第二种都可以使用，由用户选择，freertos 和 RT-Thread 则默认使用第二种。接下来我们详细讲解下 RT-Thread 的启动流程，虽然说 RT-Thread 用的是第二种，但是 RT-Thread 又拓展了 main 函数，稍微又高级了点。

19.4 RT-Thread 的启动流程

当你拿到一个移植好的 RT-Thread 工程的时候，你去看 main 函数，只能在 main 函数里面看到创建线程和启动线程的代码，硬件初始化，系统初始化，启动调度器等信息都看不到。那是因为 RT-Thread 拓展了 main 函数，在 main 函数之前把这些工作都做好了。

我们知道，在系统上电的时候第一个执行的是启动文件里面由汇编编写的复位函数 Reset_Handler，具体见*代码清单 16-3*。复位函数的最后会调用 C 库函数 __main，具体见*代码清单*

16-3 的高亮部分。__main 函数的主要工作是初始化系统的堆和栈，最后调用 C 中的 main 函数，从而去到 C 的世界。

列表 3: 代码清单 16-3 Reset_Handler 函数

```
Reset_Handler   PROC
                EXPORT  Reset_Handler             [WEAK]
                IMPORT  SystemInit
                IMPORT  __main

                CPSID   I               ; 关中断
                LDR     R0, =0xE000ED08
                LDR     R1, =__Vectors
                STR     R1, [R0]
                LDR     R2, [R1]
                MSR     MSP, R2
                LDR     R0, =SystemInit
                BLX     R0
                CPSIE   i               ; 开中断
                LDR     R0, =__main
                BX      R0
                ENDP
```

但当我们硬件仿真 RT-Thread 工程的时候，单步执行完 __main 之后，并不是跳转到 C 中的 main 函数，而是跳转到 component.c 中的 $Sub$$main 函数，这是为什么？因为 RT-Thread 使用编译器（这里仅讲解 KEIL，IAR 或者 GCC 稍微有点区别，但是原理是一样的）自带的 $Sub$$ 和 $Super$$ 这两个符号来扩展了 main 函数，使用 $Sub$$main 可以在执行 main 之前先执行 $Sub$$main，在 $Sub$$main 函数中我们可以先执行一些预操作，当做完这些预操作之后最终还是要执行 main 函数，这个就通过调用 $Super$$main 来实现。当需要扩展的函数不是 main 的时候，只需要将 main 换成你要扩展的函数名即可，即 $Sub$$function 和 $Super$$function，具体如何使用这两个扩展符号的伪代码见*代码清单 16-4*。

列表 4: 代码清单 16-4 $Sub$$ 和 $Super$$ 的使用方法

```
extern void ExtraFunc(void);      /* 用户自己实现的外部函数 */

void $Sub$$function(void)
{
    ExtraFunc();                  /* 做一些其它的设置工作 */
    $Super$$function();           /* 回到原始的 function 函数 */
}

/* 在执行 function 函数执行会先执行 function 的扩展函数$Sub$$function，在扩展函数里面执行一些扩展的操作，
当扩展操作完成后，最后必须调用$Super$$function 函数通过它回到我们原始的 function 函数 */
void function(void)
{
    /* 函数实体 */
}
```

19.4.1 $Sub$$main 函数

知道了 $Sub$$ 和 $Super$$ 的用法之后，我们回到 RT-Thread component.c 文件中的的 $Sub$$main，具体实现见*代码清单 16-5*。

列表 5: 代码清单 16-5 main 的扩展函数 $Sub$$main

```
int $Sub$$main(void)
{
    rt_hw_interrupt_disable();                    (1)
    rtthread_startup();                           (2)
    return 0;
}
```

代码清单 16-5 (1)：关闭中断，除了硬 FAULT 和 NMI 可以响应外，其它统统关掉。该函数是在接口文件 contex_rvds.S 中由汇编实现的，具体见代码清单 16-6。

列表 6: 代码清单 16-6 硬件中断失能和使能函数定义

```
;/*
; * rt_base_t rt_hw_interrupt_disable();
; */
rt_hw_interrupt_disable    PROC
    EXPORT  rt_hw_interrupt_disable
    MRS     r0, PRIMASK
    CPSID   I
    BX      LR
    ENDP

;/*
; * void rt_hw_interrupt_enable(rt_base_t level);
; */
rt_hw_interrupt_enable    PROC
    EXPORT  rt_hw_interrupt_enable
    MSR     PRIMASK, r0
    BX      LR
    ENDP
```

在 Cortex-M 内核中，为了快速地开关中断，专门设置了一条 CPS 指令，有 4 种用法，具体见代码清单 16-7。很显然，RT-Thread 里面快速关中断的方法就是用了 Cortex-M 中的 CPS 指令。

列表 7: 代码清单 16-7 Cortex-M 内核中快速关中断指令 CPS 的用法

```
CPSID I ;PRIMASK=1,  ; 关中断，只有 FAULT 和 NMI 可以响应
CPSIE I ;PRIMASK=0,  ; 开中断，只有 FAULT 和 NMI 可以响应
CPSID F ;FAULTMASK=1, ; 关异常，只有 NMI 可以响应
CPSIE F ;FAULTMASK=0 ; 开异常，只有 NMI 可以响应
```

19.4.2 rtthread_startup() 函数

代码清单 16-5 **(2)**：rtthread_startup() 函数也在 componet.c 里面实现，具体实现见*代码清单 16-8*。

列表 8: 代码清单 16-8 rtthread_startup() 函数定义

```
int rtthread_startup(void)
{
    /* 关闭中断 */
    rt_hw_interrupt_disable();                                  (1)

    /* 板级硬件初始化
    * 注意：在板级硬件初始化函数中把要堆初始化好（前提是使用动态内存）
    */
    rt_hw_board_init();                                         (2)

    /* 打印 RT-Thread 版本号 */
    rt_show_version();                                          (3)

    /* 定时器初始化 */
    rt_system_timer_init();                                     (4)

    /* 调度器初始化 */
```

(下页继续)

(续上页)

```
    rt_system_scheduler_init();                                    (5)

#ifdef RT_USING_SIGNALS
    /* 信号量初始化 */
    rt_system_signal_init();                                       (6)
#endif

    /* 创建初始线程 */
    rt_application_init();                                         (7)

    /* 定时器线程初始化 */
    rt_system_timer_thread_init();                                 (8)

    /* 空闲线程初始化 */
    rt_thread_idle_init();                                         (9)

    /* 启动调度器 */
    rt_system_scheduler_start();                                   (10)

    /* 绝对不会回到这里 */
    return 0;                                                      (11)
}
```

代码清单 16-8 **(1)**：关中断。在硬件初始化之前把中断关闭是一个很好的选择，如果没有关闭中断，在接下来的硬件初始化中如果某些外设开启了中断，那么它就有可能会响应，可是后面的 RTOS 系统初始化，调度器初始化这些都还没有完成，显然这些中断我们是不希望响应的。

代码清单 16-8 **(2)**：板级硬件初始化。RT-Thread 把板级硬件相关的初始化都放在 rt_hw_board_int() 函数里面完成，该函数需要用户在 board.c 实现。我们通常在还没有进入系统相关的操作前把硬件都初始化好且测试好，然后在继续往下执行系统相关的操作。

代码清单 16-8 **(3)**：打印 RT-Thread 的版本号，该函数在 kservice.c 中实现，具体见代码清单 16-9。rt_show_version() 函数是通过调用 rt_kprintf 函数向控制台打印 RT-Thread 版本相关的信息，要想成

功打印，必须重映射一个控制台到 rt_kprintf 函数，具体实现参考上一章《重映射串口到 rt_kprintf 函数》。如果没有重映射控制台到 rt_kprintf 函数，该函数也不会阻塞，而是打印输出为空。

列表 9: 代码清单 16-9 rt_show_version() 函数

```
void rt_show_version(void)
{
    rt_kprintf("\n \\ | /\n");
    rt_kprintf("- RT -     Thread Operating System\n");
    rt_kprintf(" / | \\     %d.%d.%d build %s\n",
            RT_VERSION, RT_SUBVERSION, RT_REVISION, __DATE__);
    rt_kprintf(" 2006 - 2018 Copyright by rt-thread team\n");
}
```

代码清单 16-8 **(4)**：定时器初始化，实际上就是初始化一个全局的定时器列表，列表里面存放的是处于延时状态的线程。

代码清单 16-8 **(5)**：调度器初始化。

代码清单 16-8 **(6)**：信号初始化，RT_USING_SIGNALS 这个宏默认不定义。

代码清单 16-8 **(7)**：创建初始线程。前面我们说过，RT-Thread 的启动流程是这样的：即先创建一个初始线程，等调度器启动之后，在这个初始线程里面创建各种应用线程，当所有应用线程都成功创建好后，初始线程就把自己关闭。那么这个初始线程就在 rt_application_init() 里面创建，该函数也在 component.c 里面定义，具体实现见*代码清单 16-10*。

19.4.3 rt_application_init() 函数

列表 10: 代码清单 16-10 创建初始线程

```
/* 使用动态内存时需要用到的宏：rt_config.h 中定义 *///                    (2)
#define RT_USING_USER_MAIN
#define RT_MAIN_THREAD_STACK_SIZE     256
#define RT_THREAD_PRIORITY_MAX        32
```

(下页继续)

(续上页)

```
5
6   /* 使用静态内存时需要用到的宏和变量: 在 component.c 定义 */ //            (4)
7   #ifdef RT_USING_USER_MAIN
8   #ifndef RT_MAIN_THREAD_STACK_SIZE
9   #define RT_MAIN_THREAD_STACK_SIZE     2048
10  #endif
11  #endif
12
13  #ifndef RT_USING_HEAP
14  ALIGN(8)
15  static rt_uint8_t main_stack[RT_MAIN_THREAD_STACK_SIZE];
16  struct rt_thread main_thread;
17  #endif
18
19  void rt_application_init(void)
20  {
21      rt_thread_t tid;
22
23  #ifdef RT_USING_HEAP
24      /* 使用动态内存 */ //                                              (1)
25      tid =
26          rt_thread_create("main",
27                           main_thread_entry,
28                           RT_NULL,
29                           RT_MAIN_THREAD_STACK_SIZE,
30                           RT_THREAD_PRIORITY_MAX / 3,     (初始线程优先级)
31                           20);
32      RT_ASSERT(tid != RT_NULL);
33  #else
34      /* 使用静态内存 */ //                                              (3)
35      rt_err_t result;
36
```

(下页继续)

(续上页)

```
    tid = &main_thread;
    result =
        rt_thread_init(tid,
                       "main",
                       main_thread_entry,
                       RT_NULL,
                       main_stack,
                       sizeof(main_stack),
                       RT_THREAD_PRIORITY_MAX / 3,       (初始线程优先级)
                       20);
    RT_ASSERT(result == RT_EOK);
    (void)result;
#endif

    /* 启动线程 */
    rt_thread_startup(tid);//                                    (6)
}

/* main 线程 */
void main_thread_entry(void *parameter)//                        (5)
{
    extern int main(void);
    extern int $Super$$main(void);

    /* RT-Thread 组件初始化 */
    rt_components_init();

    /* 调用$Super$$main()函数，去到 main */
    $Super$$main();
}
```

代码清单 16-10 **(1)**：创建初始线程的时候，分使用动态内存和静态内存两种情况，通常我们使用

动态内存，有关动态内存需要用到的宏定义具体见*代码清单 16-10* **(2)**。

代码清单 16-10 **(3)**：创建初始线程的时候，分使用动态内存和静态内存两种情况，这里是使用静态内存，有关静态内存需要用到的宏定义具体见*代码清单 16-10* **(4)**。

19.4.4 $Super$$main() 函数

代码清单 16-10 **(5)**：初始线程入口。该函数除了调用 rt_components_init() 函数进行 RT-Thread 的组件初始化外，最终是调用 main 的扩展函数 $Super$$main() 回到 main 函数。这个是必须的，因为我们一开始在进入 main 函数之前，通过 $Sub$$main() 函数扩展了 main 函数，做了一些硬件初始化，RTOS 系统初始化的工作，当这些工作做完之后最终还是要回到 main 函数，那只能通过调用 $Super$$main() 函数来实现。$Sub$$ 和 $Super$$ 是 MDK 自带的用来扩展函数的符号，通常是成对使用。

代码清单 16-10 **(6)**：启动初始线程，这个时候初始线程还不会立即被执行，因为调度器还没有启动。

代码清单 16-10 **(初始线程优先级)**：初始线程的优先级默认配置为最大优先级/3。控制最大优先级的宏 RT_THREAD_PRIORITY_MAX 在 rt_config.h 中定义，目前配置为 32 ，那初始线程的优先级即是 10，那在初始线程里面创建的各种应用线程的优先级又该如何配置？分三种情况：1、应用线程的优先级比初始线程的优先级高，那创建完后立马去执行刚刚创建的应用线程，当应用线程被阻塞时，继续回到初始线程被打断的地方继续往下执行，直到所有应用线程创建完成，最后初始线程把自己关闭，完成自己的使命；2、应用线程的优先级与初始线程的优先级一样，那创建完后根据线程的时间片来执行，直到所有应用线程创建完成，最后初始线程把自己关闭，完成自己的使命；3、应用线程的优先级比初始线程的优先级低，那创建完后线程不会被执行，如果还有应用线程紧接着创建应用线程，如果应用线程的优先级出现了比初始线程高或者相等的情况，请参考 1 和 2 的处理方式，直到所有应用线程创建完成，最后初始线程把自己关闭，完成自己的使命。

19.4.5 main 函数

当我们拿到一个移植好 RT-Thread 的例程的时候，不出意外，你首先看到的是 main 函数，当你认真一看 main 函数里面只是创建并启动一些线程，那硬件初始化，系统初始化，这些统统在哪里？这些 RT-Thread 通过扩展 main 函数的方式都在 component.c 里面实现了，具体过程往回看本章的其它小节的详细讲解。

列表 11: 代码清单 16-11 main 函数

```
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 开发板硬件初始化，RT-Thread 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。(1)
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    (2)
    thread1 =                                /* 线程控制块指针 */
    rt_thread_create("thread1",              /* 线程名字，字符串形式 */
                     thread1_entry,         /* 线程入口函数 */
                     RT_NULL,               /* 线程入口函数参数 */
                     HREAD1_STACK_SIZE,     /* 线程栈大小，单位为字节 */
                     THREAD1_PRIORITY,      /* 线程优先级，数值越大，优先级越小 */
                     THREAD1_TIMESLICE);    /* 线程时间片 */

    if (thread1 != RT_NULL)
        rt_thread_startup(thread1);
```

(下页继续)

(续上页)

```
    else
        return -1;
    (3)
    thread2 =                                   /* 线程控制块指针 */
    rt_thread_create("thread2",                 /* 线程名字，字符串形式 */
                     thread2_entry,            /* 线程入口函数 */
                     RT_NULL,                  /* 线程入口函数参数 */
                     THREAD2_STACK_SIZE,       /* 线程栈大小，单位为字节 */
                     THREAD2_PRIORITY,         /* 线程优先级，数值越大，优先级越小 */
                     THREAD2_TIMESLICE);       /* 线程时间片 */

    if (thread2 != RT_NULL)
        rt_thread_startup(thread2);
    else
        return -1;
    (4)
    thread3 =                                   /* 线程控制块指针 */
    rt_thread_create("thread3",                 /* 线程名字，字符串形式 */
                     thread3_entry,             /* 线程入口函数 */
                     RT_NULL,                   /* 线程入口函数参数 */
                     THREAD3_STACK_SIZE,        /* 线程栈大小，单位为字节 */
                     THREAD3_PRIORITY,          /* 线程优先级，数值越大，优先级越小 */
                     THREAD3_TIMESLICE);        /* 线程时间片 */

    if (thread3 != RT_NULL)
        rt_thread_startup(thread3);
    else
        return -1;

    /* 执行到最后，通过 LR 寄存器执行的地址返回 */        (5)
}
```

代码清单 16-11 **(1)**：开发板硬件初始化，RT-Thread 系统初始化已经在 main 函数之前完成，即在 component.c 文件中的 rtthread_startup() 函数中完成了，所以在 main 函数中，只需要创建线程和启动线程即可。

代码清单 16-11 **(2) (3) (4)**：创建各种应用线程，当创建的应用线程的优先级比 main 线程的优先级高、低或者相等时候，程序是如何执行的？具体看*代码清单 16-10* **(初始线程优先级)** 的分析。

代码清单 16-11 **(5)**：main 线程执行到最后，通过 LR 寄存器指定的链接地址退出，在创建 main 线程的时候，线程栈对应 LR 寄存器的内容是 rt_thread_exit() 函数，在 rt_thread_exit 里面会把 main 线程占用的内存空间都释放掉。

至此，RT-Thread 的整个启动流程我们就讲完了。

第 20 章　线程管理

20.1 线程的基本概念

从系统的角度看，线程是竞争系统资源的最小运行单元。RT-Thread 是一个支持多线程的操作系统。在 RT-Thread 中，线程可以使用或等待 CPU、使用内存空间等系统资源，并独立于其它线程运行。

简而言之：RT-Thread 的线程可认为是一系列独立线程的集合。每个线程在自己的环境中运行。在任何时刻，只有一个线程得到运行，RT-Thread 调度器决定运行哪个线程。调度器会不断启动、停止每一个线程，宏观看上去所有的线程都在同时在执行。作为线程，不需要对调度器的活动有所了解，在线程切入切出时保存上下文环境（寄存器值、堆栈内容）是调度器主要的职责。为了实现这点，每个 RT-Thread 线程都需要有自己的堆栈。当线程切出时，它的执行环境会被保存在该线程的堆栈中，这样当线程再次运行时，就能从堆栈中正确的恢复上次的运行环境。

RT-Thread 的线程模块可以给用户提供多个线程，实现了线程之间的切换和通信，帮助用户管理业务程序流程。这样用户可以将更多的精力投入到业务功能的实现中。

RT-Thread 中的线程是抢占式调度机制，同时支持时间片轮转调度方式。

高优先级的线程可打断低优先级线程，低优先级线程必须在高优先级线程阻塞或结束后才能得到调度。

20.2 线程调度器的基本概念

RT-Thread 中提供的线程调度器是基于优先级的全抢占式调度：在系统中除了中断处理函数、调度器上锁部分的代码和禁止中断的代码是不可抢占的之外，系统的其他部分都是可以抢占的，包括线程调度器自身。系统总共支持 256 个优先级 (0 ~ 255，数值越小的优先级越高，0 为最高优先级，255 分配给空闲线程使用，一般用户不使用。在一些资源比较紧张的系统中，可以根据实

际情况选择只支持 8 个或 32 个优先级的系统配置)。在系统中，当有比当前线程优先级更高的线程就绪时，当前线程将立刻被换出，高优先级线程抢占处理器运行。

一个操作系统如果只是具备了高优先级线程能够“立即”获得处理器并得到执行的特点，那么它仍然不算是实时操作系统。因为这个查找最高优先级线程的过程决定了调度时间是否具有确定性，例如一个包含 n 个就绪线程的系统中，如果仅仅从头找到尾，那么这个时间将直接和 n 相关，而下一个就绪线程抉择时间的长短将会极大的影响系统的实时性。当所有就绪线程都链接在它们对应的优先级队列中时，抉择过程就将演变为在优先级数组中寻找具有最高优先级线程的非空链表。RT-Thread 内核中采用了基于位图的优先级算法（时间复杂度 O(1)，即与就绪线程的多少无关），通过位图的定位快速的获得优先级最高的线程，具体见 10.1.1 1 小节。

RT-Thread 内核中也允许创建相同优先级的线程。相同优先级的线程采用时间片轮转方式进行调度（也就是通常说的分时调度器），时间片轮转调度仅在当前系统中无更高优先级就绪线程存在的情况下才有效。因为 RT-Thread 调度器的实现是采用优先级链表的方式，所以系统中的总线程数不受限制，只和系统所能提供的内存资源相关。为了保证系统的实时性，系统尽最大可能地保证高优先级的线程得以运行。线程调度的原则是一旦线程状态发生了改变，并且当前运行的线程优先级小于优先级队列组中线程最高优先级时，立刻进行线程切换（除非当前系统处于中断处理程序中或禁止线程切换的状态）。

20.3 线程状态的概念

RT-Thread 系统中的每一线程都有多种运行状态。系统初始化完成后，创建的线程就可以在系统中竞争一定的资源，由内核进行调度。

线程状态通常分为以下四种：

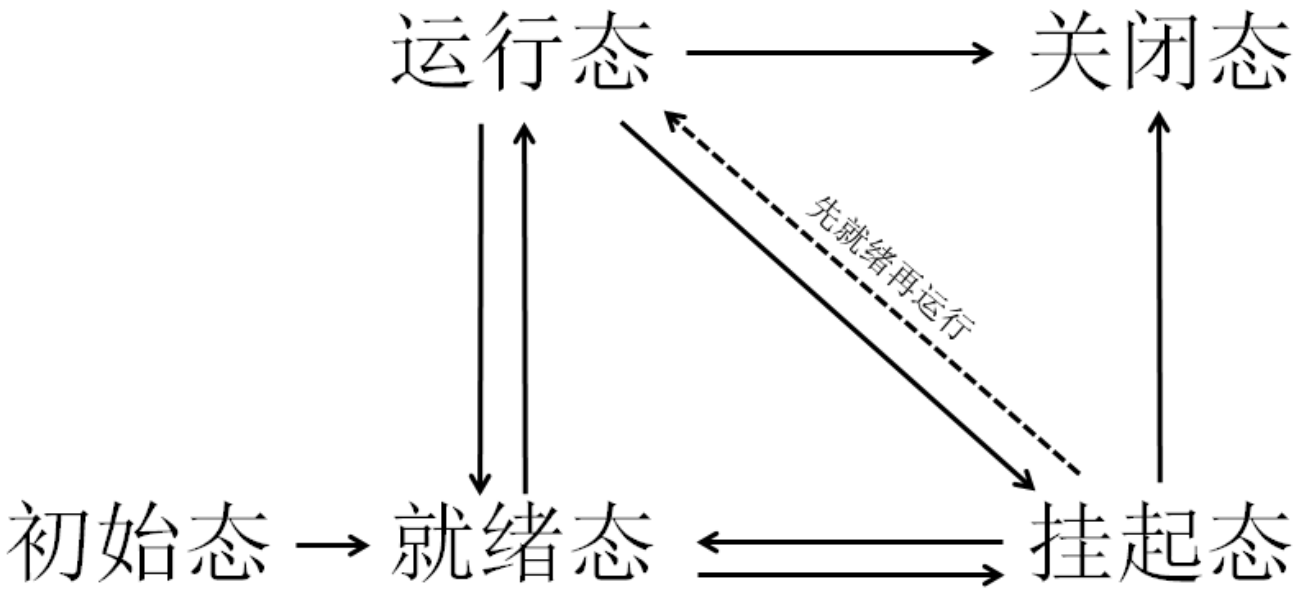
- 初始态（RT_THREAD_INIT）：创建线程的时候会将线程的状态设置为初始态。
- 就绪态（RT_THREAD_READY）：该线程在就绪列表中，就绪的线程已经具备执行的能力，只等待 CPU。
- 运行态（RT_THREAD_RUNNING）：该线程正在执行，此时它占用处理器。
- 挂起态（RT_THREAD_SUSPEND）：如果线程当前正在等待某个时序或外部中断，我们就说这个线程处于挂起状态，该线程不在就绪列表中。包含线程被挂起、线程被延时、线程

正在等待信号量、读写队列或者等待读写事件等。

- 关闭态（RT_THREAD_CLOSE）：该线程运行结束，等待系统回收资源。

20.4 线程状态迁移

RT-Thread 系统中的每一个线程都有多种运行状态，他们之间的转换关系是怎么样的呢？从运行态线程变成阻塞态，或者从阻塞态变成就绪态，这些线程状态是怎么迁移的呢，下面就让我们一起了解线程状态迁移吧，具体见图 *17-1*



初始态 → 就绪态：线程创建后进入初始态（RT_THREAD_INIT），在线程启动的时候（调用 rt_thread_startup() 函数）会将初始态转变为就绪态，表明线程已启动，线程可以进行调度。

就绪态（RT_THREAD_READY）→ 运行态：发生线程切换时，就绪列表中最高优先级的线程被执行，从而进入运行态（RT_THREAD_RUNNING）。

运行态（RT_THREAD_RUNNING）→ 挂起态：正在运行的线程发生阻塞（挂起、延时、读信号量等待）时，该线程会从就绪列表中删除，线程状态由运行态变成挂起态，然后发生线程切换，运行就绪列表中最高优先级线程。

挂起态（RT_THREAD_SUSPEND）→ 就绪态（阻塞态 → 运行态）：阻塞的线程被恢复后（线程恢

复、延时时间超时、读信号量超时或读到信号量等），此时被恢复的线程会被加入就绪列表，从而由挂起态变成就绪态；此时如果被恢复线程的优先级高于正在运行线程的优先级，则会发生线程切换，将该线程由就绪态变成运行态。

就绪态（RT_THREAD_READY）→ 挂起态：线程也有可能在就绪态时被挂起，此时线程状态会由就绪态转变为挂起态，该线程从就绪列表中删除，不会参与线程调度，直到该线程被恢复。

运行态（RT_THREAD_RUNNING）→ 就绪态：有更高优先级线程创建或者恢复后，会发生线程调度，此刻就绪列表中最高优先级线程变为运行态，那么原先运行的线程由运行态变为就绪态，依然在就绪列表中。

挂起态 → 关闭态（RT_THREAD_CLOSE）：处于挂起的线程被调用删除接口，线程状态由挂起态变为关闭态。

运行态 → 关闭态：运行状态的线程，如果运行结束会在线程最后部分执行 rt_thread_exit() 函数而更改为关闭状态（RT_THREAD_CLOSE）。

20.5 常用的线程函数讲解

相信大家通过第一部分章节的学习，对线程创建以及线程调度的实现已经掌握了，下面就补充一些 RT-Thread 提供给我们对线程操作的一些常用函数。

20.5.1 线程挂起函数 rt_thread_suspend()

挂起指定线程。被挂起的线程绝不会得到处理器的使用权，不管该线程具有什么优先级。

线程挂起可以由多种方法实现：线程调用 rt_thread_delay()、rt_thread_suspend() 等函数接口可以使得线程主动挂起，放弃 CPU 使用权，当线程调用 rt_sem_take()，rt_mb_recv() 等函数时，资源不可使用也会导致调用线程被动挂起。

当线程已经是挂起态的时候无法调用 rt_thread_suspend() 函数，已经是挂起态的线程调用 rt_thread_suspend() 将返回错误代码，挂起的线程想要恢复可以调用 rt_thread_resume() 函数。线程挂起是我们经常使用的一个函数，下面一起看看线程挂起的源码，了解其工作过程，具体见*代码清单 17-1*。

列表 1: 代码清单 17-1 线程挂起函数 rt_thread_suspend()源码

```
rt_err_t rt_thread_suspend(rt_thread_t thread)
{
    register rt_base_t temp;

    /* 线程检查 */
    RT_ASSERT(thread != RT_NULL);                                (1)

    RT_DEBUG_LOG(RT_DEBUG_THREAD, ("thread suspend:  %s\n", thread->name));

    if ((thread->stat & RT_THREAD_STAT_MASK) != RT_THREAD_READY) {(2)
        RT_DEBUG_LOG(RT_DEBUG_THREAD, ("thread suspend: thread disorder, 0x
↪%2x\n",
                                       thread->stat));

        return -RT_ERROR;
    }

    /* 关中断 */
    temp = rt_hw_interrupt_disable();

    /* 改变状态 */
    thread->stat = RT_THREAD_SUSPEND | (thread->stat &
                    ~RT_THREAD_STAT_MASK);  (3)
    rt_schedule_remove_thread(thread);
    /* 停止线程计时器 */
    rt_timer_stop(&(thread->thread_timer));          (4)

    /* 开中断 */
    rt_hw_interrupt_enable(temp);
```

(下页继续)

(续上页)

```
    RT_OBJECT_HOOK_CALL(rt_thread_suspend_hook, (thread));
    return RT_EOK;
}
```

代码清单 17-1 **(1)**：判断线程是否有效，如果是没被创建的线程，那么无法挂起。

代码清单 17-1 **(2)**：判断要挂起线程的状态，如果是已经挂起了，会返回错误码，用户可以在恢复线程后再挂起。

代码清单 17-1 **(3)**：将线程的状态变为挂起态。

代码清单 17-1 **(4)**：停止线程计时器。

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

线程的挂起与恢复函数在很多时候都是很有用的，比如我们想暂停某个线程运行一段时间，但是我们又需要在其恢复的时候继续工作，那么删除线程是不可能的，因为删除了线程的话，线程的所有的信息都是不可能恢复的了，删除是完完全全删除了，里面的资源都被系统释放掉，但是挂起线程就不会这样子，调用挂起线程函数，仅仅是将线程进入阻塞态，其内部的资源都会保留下来，同时也不会参与线程的调度，当调用恢复函数的时候，整个线程立即从阻塞态进入就绪态，参与线程的调度，如果该线程的优先级是当前就绪态优先级最高的线程，那么立即会按照挂起前的线程状态继续执行该线程，从而达到我们需要的效果，注意，是继续执行，也就是说，暂停线程之前是什么状态，都会被系统保留下来，在恢复的瞬间，继续执行。这个线程函数的使用方法是很简单的，只需把线程控制块传递进来即可，rt_thread_suspend() 会根据线程控制块的信息将对应的线程挂起，具体见代码清单 17-2 高亮部分。

列表 2: 代码清单 17-2 线程挂起函数 rt_thread_suspend() 实例

```
rt_kprintf("挂起 LED1 线程! \n");
uwRet = rt_thread_suspend(led1_thread);/* 挂起 LED1 线程 */
if (RT_EOK == uwRet)
{
    rt_kprintf("挂起 LED1 线程成功! \n");
} else
{
    rt_kprintf("挂起 LED1 线程失败! 失败代码: 0x%lx\n",uwRet);
}
```

20.5.2 线程恢复函数 rt_thread_resume()

既然有线程的挂起，那么当然一样有恢复，不然线程怎么恢复呢，线程恢复就是让挂起的线程重新进入就绪状态，恢复的线程会保留挂起前的状态信息，在恢复的时候根据挂起时的状态继续运行。如果被恢复线程在所有就绪态线程中，位于最高优先级链表的第一位，那么系统将进行线程上下文的切换。下面一起看看线程恢复函数 rt_thread_resume() 的源码，具体见*代码清单 17-3*。

列表 3: 代码清单 17-3 线程恢复函数 rt_thread_resume() 源码

```
rt_err_t rt_thread_resume(rt_thread_t thread)
{
    register rt_base_t temp;

    /* 线程检查 */
    RT_ASSERT(thread != RT_NULL);

    RT_DEBUG_LOG(RT_DEBUG_THREAD, ("thread resume:  %s\n", thread->name));
```

(下页继续)

(续上页)

```

    if ((thread->stat & RT_THREAD_STAT_MASK) != RT_THREAD_SUSPEND) {(1)
        RT_DEBUG_LOG(RT_DEBUG_THREAD, ("thread resume: thread disorder, %d\n",
                                        thread->stat));

        return -RT_ERROR;
    }

    /* 关中断 */
    temp = rt_hw_interrupt_disable();

    /* 从列表删除 */
    rt_list_remove(&(thread->tlist));                        (2)

    rt_timer_stop(&thread->thread_timer);

    /* 开中断 */
    rt_hw_interrupt_enable(temp);

    /* 加入就绪列表 */
    rt_schedule_insert_thread(thread);                       (3)

    RT_OBJECT_HOOK_CALL(rt_thread_resume_hook, (thread));
    return RT_EOK;
}
```

代码清单 17-3 **(1)**：判断线程是否有效，如果是没被创建的线程，那么无法恢复。并且检查当前线程是否已经挂起，要恢复的线程当然是必须是挂起态的，如果不是挂起态的根本不需要进行恢复。

代码清单 17-3 **(2)**：将线程从挂起列表中删除。

代码清单 17-3 (3)：将恢复的线程加入就绪列表，但是此时线程能不能立即运行是根据其优先级决定的，如果该线程的优先级在就绪列表中最高，那么是可以立即运行的。

线程的恢复是十分简单的，简单来说就是将线程状态从挂起列表移到就绪列表中，当线程的优先级为最高的时候，就发起线程切换。下面来看看线程恢复函数 rt_thread_resume() 的使用实例，具体见代码清单 17-4 高亮部分。

列表 4: 代码清单 17-4 线程恢复函数 rt_thread_resume() 实例

```
rt_kprintf("恢复 LED1 线程! \n");
uwRet = rt_thread_resume(led1_thread);/* 恢复 LED1 线程! */
if (RT_EOK == uwRet)
{
    rt_kprintf("恢复 LED1 线程成功! \n");
}
else
{
    rt_kprintf("恢复 LED1 线程失败! 失败代码: 0x%lx\n",uwRet);
}
```

20.6 线程的设计要点

作为一个嵌入式开发人员，要对自己设计的嵌入式系统要了如指掌，线程的优先级信息，线程与中断的处理，线程的运行时间、逻辑、状态等都要知道，才能设计出好的系统，所以，在设计的时候需要根据需求制定框架。在设计之初就应该考虑下面几点因素：线程运行的上下文环境、线程的执行时间合理设计。

RT-Thread 中程序运行的上下文包括：

- 中断服务函数。
- 普通线程。
- 空闲线程。

1. 中断服务函数：

中断服务函数是一种需要特别注意的上下文环境，它运行在非线程的执行环境下（一般为芯片的一种特殊运行模式（也被称作特权模式）），在这个上下文环境中不能使用挂起当前线程的操作，不允许调用任何会阻塞运行的 API 函数接口。另外需要注意的是，中断服务程序最好保持精简短小，快进快出，一般在中断服务函数中只做标记事件的发生，让对应线程去执行相关处理，因为中断服务函数的优先级高于任何优先级的线程，如果中断处理时间过长，将会导致整个系统的线程无法正常运行。所以在设计的时候必须考虑中断的频率、中断的处理时间等重要因素，以便配合对应中断处理线程的工作。

2. 线程：

线程看似没有什么限制程序执行的因素，似乎所有的操作都可以执行。但是做为一个优先级明确的实时系统，如果一个线程中的程序出现了死循环操作（此处的死循环是指没有不带阻塞机制的线程循环体），那么比这个线程优先级低的线程都将无法执行，当然也包括了空闲线程，因为死循环的时候，线程不会主动让出 CPU，低优先级的线程是不可能得到 CPU 的使用权的，而高优先级的线程就可以抢占 CPU。这个情况在实时操作系统中是必须注意的一点，所以在线程中不允许出现死循环。如果一个线程只有就绪态而无阻塞态，势必会影响到其他低优先级线程的执行，所以在进行线程设计时，就应该保证线程在不活跃的时候，线程可以进入阻塞态以交出 CPU 使用权，这就需要我们自己明确知道什么情况下让线程进入阻塞态，保证低优先级线程可以正常运行。在实际设计中，一般会将紧急的处理事件的线程优先级设置得高一些。

3. 空闲线程：

空闲线程（idle 线程）是 RT-Thread 系统中没有其他工作进行时自动进入的系统线程。用户可以通过空闲线程钩子方式，在空闲线程上钩入自己的功能函数。通常这个空闲线程钩子能够完成一些额外的特殊功能，例如系统运行状态的指示，系统省电模式等。除了空闲线程钩子，RT-Thread 系统还把空闲线程用于一些其他的功能，比如当系统删除一个线程或一个动态线程运行结束时，会先行更改线程状态为非调度状态，然后挂入一个待回收队列中，真正的系统资源回收工作在空闲线程完成，空闲线程是唯一不允许出现阻塞情况的线程，因为 RT-Thread 需要保证系统用于都有一个可运行的线程。

对于空闲线程钩子上挂接的空闲钩子函数，它应该满足以下的条件：

- 不会挂起空闲线程；

- 不应该陷入死循环，需要留出部分时间用于系统处理系统资源回收。

线程的执行时间：

线程的执行时间一般是指两个方面，一是线程从开始到结束的时间，二是线程的周期。

在系统设计的时候这两个时间候我们都需要考虑，例如，对于事件 A 对应的服务线程 Ta，系统要求的实时响应指标是 10ms，而 Ta 的最大运行时间是 1ms，那么 10ms 就是线程 Ta 的周期了，1ms 则是线程的运行时间，简单来说线程 Ta 在 10ms 内完成对事件 A 的响应即可。此时，系统中还存在着以 50ms 为周期的另一线程 Tb，它每次运行的最大时间长度是 100us。在这种情况下，即使把线程 Tb 的优先级抬到比 Ta 更高的位置，对系统的实时性指标也没什么影响，因为即使在 Ta 的运行过程中，Tb 抢占了 Ta 的资源，等到 Tb 执行完毕，消耗的时间也只不过是 100us，还是在事件 A 规定的响应时间内 (10ms)，Ta 能够安全完成对事件 A 的响应。但是假如系统中还存在线程 Tc，其运行时间为 20ms，假如将 Tc 的优先级设置比 Ta 更高，那么在 Ta 运行的时候，突然间被 Tc 打断，等到 Tc 执行完毕，那 Ta 已经错过对事件 A（10ms）的响应了，这是不允许的。所以在我们设计的时候，必须考虑线程的时间，一般来说处理时间更短的线程优先级应设置更高一些。

20.7 线程管理实验

线程管理实验是将线程常用的函数进行一次实验，在野火 i.MX RT 开发板上进行该试验，通过创建两个线程，一个是 LED 线程，另一个是按键线程，LED 线程是显示线程运行的状态，而按键线程是通过检测按键的按下与否来进行对 LED 线程的挂起与恢复，具体见*代码清单 17-5* 高亮部分。

列表 5: 代码清单 17-5 线程管理实验源码

```
/**
*****************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2018-xx-xx
```

(下页继续)

(续上页)

```
* @brief   RT-Thread 系统 线程管理
*****************************************************************
* @attention
*
* 实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
*****************************************************************
*/

/*
*************************************************************************
*                              包含的头文件
*************************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"

#include "rtthread.h"
/*
*************************************************************************
*                                 变量
*************************************************************************
*/
/* 定义线程控制块 */
static rt_thread_t led1_thread = RT_NULL;
static rt_thread_t key_thread = RT_NULL;
/*
*************************************************************************
*                              函数声明
*************************************************************************
```

(下页继续)

(续上页)

```
*/
static void led1_thread_entry(void* parameter);
static void key_thread_entry(void* parameter);

/*
*************************************************************************
*                              main 函数
*************************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]-i.MX RT 全系列开发板 RTT 线程管理实验！\n\n");
    rt_kprintf("按下 K1 挂起线程，按下 K2 恢复线程\n");
    led1_thread =                           /* 线程控制块指针 */
        rt_thread_create( "led1",               /* 线程名字 */
                          led1_thread_entry,    /* 线程入口函数 */
                          RT_NULL,              /* 线程入口函数参数 */
                          512,                  /* 线程栈大小 */
                          3,                    /* 线程的优先级 */
                          20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
```

(下页继续)

(续上页)

```
    if (led1_thread != RT_NULL)
        rt_thread_startup(led1_thread);
    else
        return -1;

    key_thread =                            /* 线程控制块指针 */
        rt_thread_create( "key",                /* 线程名字 */
                        key_thread_entry,   /* 线程入口函数 */
                        RT_NULL,            /* 线程入口函数参数 */
                        512,                /* 线程栈大小 */
                        2,                  /* 线程的优先级 */
                        20);                /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (key_thread != RT_NULL)
        rt_thread_startup(key_thread);
    else
        return -1;
}

/*
*****************************************************************************
*                                 线程定义
*****************************************************************************
*/

static void led1_thread_entry(void* parameter)
{

    while (1) {
        LED1_ON;
        rt_thread_delay(500);   /* 延时 500 个 tick */
```

(下页继续)

(续上页)

```
        rt_kprintf("led1_thread running,LED1_ON\r\n");

        LED1_OFF;
        rt_thread_delay(500);   /* 延时 500 个 tick */
        rt_kprintf("led1_thread running,LED1_OFF\r\n");
    }
}

static void key_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
    while (1) {/* K1 被按下 */
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_GPIO_PIN) == KEY_ON ) {
            rt_kprintf("挂起 LED1 线程! \n");
            uwRet = rt_thread_suspend(led1_thread);/* 挂起 LED1 线程 */
            if (RT_EOK == uwRet) {
                rt_kprintf("挂起 LED1 线程成功! \n");
            } else {
                rt_kprintf("挂起 LED1 线程失败! 失败代码: 0x%lx\n",uwRet);
            }
        }/* K2 被按下 */
        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_GPIO_PIN) == KEY_ON ) {
            rt_kprintf("恢复 LED1 线程! \n");
            uwRet = rt_thread_resume(led1_thread);/* 恢复 LED1 线程!  */
            if (RT_EOK == uwRet) {
                rt_kprintf("恢复 LED1 线程成功! \n");
            } else {
                rt_kprintf("恢复 LED1 线程失败! 失败代码: 0x%lx\n",uwRet);
            }
        }
        rt_thread_delay(20);
    }
```

(下页继续)

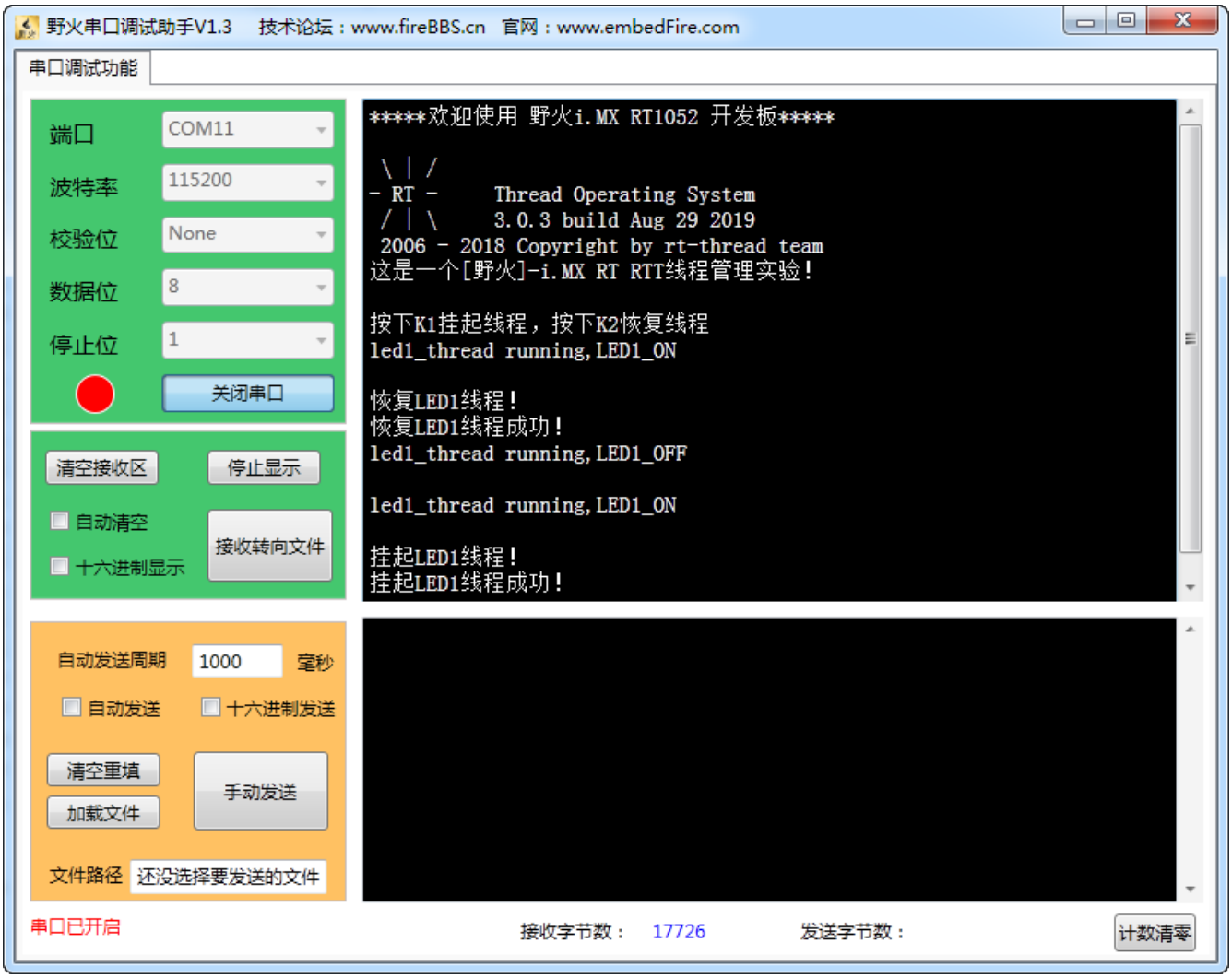
(续上页)

```
}
/**********************END OF FILE***************************/
```

20.8 实验现象

将程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息，在开发板可以看到，LED 在闪烁，按下开发版的 K1 按键挂起线程，按下 K2 按键恢复线程；我们按下 K1 试试，可以看到开发板上的灯也不闪烁了，同时在串口调试助手也输出了相应的信息，说明线程已经被挂起，我们按下 K2 试试，可以看到开发板上的灯也恢复闪烁了，同时在串口调试助手也输出了相应的信息，说明线程已经被恢复，具体见图 17-2。

注意： 由于 RT-Thread 中挂起线程函数不允许将已经在阻塞态的线程进行操作，而我们的实验中 LED 线程的延时函数会将线程挂起进入阻塞态，所以，在挂起的时候可能会挂起失败，多尝试几次即可。我们一般调用挂起函数是在线程就绪或者运行的时候将其挂起，而不是在挂起态再将线程挂起，本实验仅为演示与介绍如何使用 RT-Thread 的挂起与恢复函数。


野火串口调试助手V1.3 技术论坛：www.fireBBS.cn 官网：www.embedFire.com
串口调试功能
端口 COM11
波特率 115200
校验位 None
数据位 8
停止位 1
关闭串口
清空接收区
停止显示
自动清空
十六进制显示
接收转向文件
自动发送周期 1000 毫秒
自动发送
十六进制发送
清空重填
加载文件
手动发送
文件路径 还没选择要发送的文件
*****欢迎使用 野火i.MX RT1052 开发板*****
\ | /
- RT - Thread Operating System
/ | \ 3.0.3 build Aug 29 2019
2006 - 2018 Copyright by rt-thread team
这是一个[野火]-i.MX RT RTT线程管理实验！
按下K1挂起线程，按下K2恢复线程
led1_thread running,LED1_ON
恢复LED1线程！
恢复LED1线程成功！
led1_thread running,LED1_OFF
led1_thread running,LED1_ON
挂起LED1线程！
挂起LED1线程成功！
串口已开启
接收字节数： 17726
发送字节数：
计数清零

第 21 章　消息队列

同学们，回想一下，在我们裸机的编程中，我们是怎么样用全局的一个数组的呢?

21.1 消息队列的基本概念

队列又称消息队列，是一种常用于线程间通信的数据结构，队列可以在线程与线程间、中断和线程间传送信息，实现了线程接收来自其他线程或中断的不固定长度的消息，并根据不同的接口选择传递消息是否存放在线程自己的空间。线程能够从队列里面读取消息，当队列中的消息是空时，挂起读取线程，用户还可以指定挂起的线程时间 timeout；当队列中有新消息时，挂起的读取线程被唤醒并处理新消息，消息队列是一种异步的通信方式。

通过消息队列服务，线程或中断服务例程可以将一条或多条消息放入消息队列中。同样，一个或多个线程可以从消息队列中获得消息。当有多个消息发送到消息队列时，通常是将先进入消息队列的消息先传给线程，也就是说，线程先得到的是最先进入消息队列的消息，即先进先出原则 (FIFO)。同时 RT-Thread 中的消息队列支持优先级，也就是说在所有等待消息的线程中优先级最高的会先获得消息。

用户在处理业务时，消息队列提供了异步处理机制，允许将一个消息放入队列，但并不立即处理它，同时队列还能起到缓冲消息作用。

RT-Thread 中使用队列数据结构实现线程异步通信工作，具有如下特性:

- 消息支持先进先出方式排队与优先级排队方式，支持异步读写工作方式。
- 读队列支持超时机制。
- 支持发送紧急消息，这里的紧急消息是往队列头发送消息。
- 可以允许不同长度（不超过队列节点最大值）的任意类型消息。
- 一个线程能够从任意一个消息队列接收和发送消息。

- 多个线程能够从同一个消息队列接收和发送消息。
- 当队列使用结束后，需要通过删除队列操作释放内存函数回收。

21.2 消息队列的运作机制

创建消息队列时先创建一个消息队列对象控制块，然后给消息队列分配一块内存空间，组织成空闲消息链表，这块内存的大小等于 [消息大小 + 消息头（用于链表连接）] 与消息队列容量的乘积，接着再初始化消息队列，此时消息队列为空。

RT-Thread 操作系统的消息队列对象由多个元素组成，当消息队列被创建时，它就被分配了消息队列控制块：消息队列名称、内存缓冲区、消息大小以及队列长度等。同时每个消息队列对象中包含着多个消息框，每个消息框可以存放一条消息；消息队列中的第一个和最后一个消息框被分别称为消息链表头和消息链表尾，对应于消息队列控制块中的 msg_queue_head 和 msg_queue_tail；有些消息框可能是空的，它们通过 msg_queue_free 形成一个空闲消息框链表。所有消息队列中的消息框总数即是消息队列的长度，这个长度可在消息队列创建时指定。

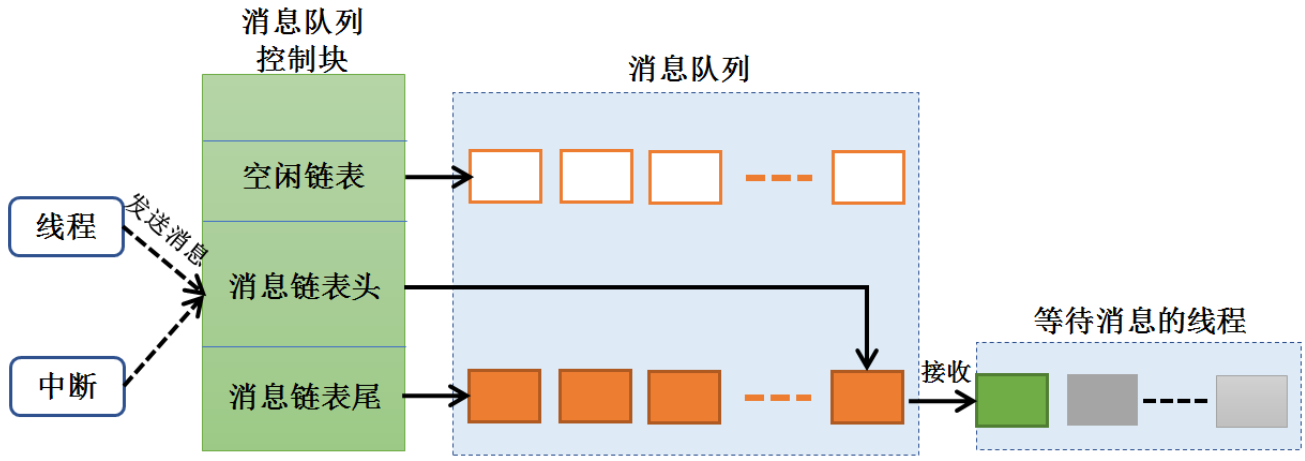
线程或者中断服务程序都可以给消息队列发送消息。当发送消息时，消息队列对象先从空闲消息链表上取下一个空闲消息块，把线程或者中断服务程序发送的消息内容复制到消息块上，然后把该消息块挂到消息队列的尾部。当且仅当空闲消息链表上有可用的空闲消息块时，发送者才能成功发送消息；当空闲消息链表上无可用消息块，说明消息队列已满，此时，发送消息的的线程或者中断程序会收到一个错误码（-RT_EFULL）。

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

读取消息时，根据 msg_queue_head 找到最先入队列中的消息节点进行读取。根据消息队列控制块中的 entry 判断队列是否有消息读取，对全部空闲（entry 为 0）队列进行读消息操作会引起线程挂起。

当消息队列不再被使用时，应该删除它以释放系统资源，一旦操作完成，消息队列将被永久性的删除。

队列的运作过程具体见图 *18-1*。



21.3 消息队列的阻塞机制

我们使用的消息队列一般不是属于某个线程的队列，在很多时候，我们创建的队列，是每个线程都可以去对他进行读写操作的，但是为了保护每个线程对它进行读写操作的过程，我们必须要有阻塞机制，在某个线程对它读写操作的时候，必须保证该线程能正常完成读写操作，而不受后来的线程干扰，凡事都有先来后到嘛！

那么，如何实现这个先来后到的机制呢，很简单，因为 RT-Thread 已经为我们做好了，我们直接使用就好了，每个对消息队列读写的函数，都有这种机制，我称之为阻塞机制。假设有一个线程 A 对某个队列进行读操作的时候（也就是我们所说的出队），发现它没有消息，那么此时线程 A 有 3 个选择：第一个选择，线程 A 扭头就走，既然队列没有消息，那我也不等了，干其它事情去，这样子线程 A 不会进入阻塞态；第二个选择，线程 A 还是在这里等等吧，可能过一会队列就有消息，此时线程 A 会进入阻塞状态，在等待着消息的道来，而线程 A 的等待时间就由我们自己定义，比如设置 1000 个 tick 的等待，在这 1000 个 tick 到来之前线程 A 都是处于阻塞态，当阻塞的这段时间线程 A 等到了队列的消息，那么线程 A 就会从阻塞态变成就绪态，如果此时线程 A 比当前运行的线程优先级还高，那么，线程 A 就会得到消息并且运行；假如 1000 个 tick 都过去了，队列还没消息，那线程 A 就不等了，从阻塞态中唤醒，返回一个没等到消息的错误代码，然后继续执行线程 A 的其他代码；第三个选择，线程 A 死等，不等到消息就不走了，这样子线程 A 就会进入阻塞态，直到完成读取队列的消息。

而在发送消息操作的时候，为了保护数据，当且仅当空闲消息链表上有可用的空闲消息块时，发

送者才能成功发送消息；当空闲消息链表上无可用消息块，说明消息队列已满，此时，发送消息的的线程或者中断程序会收到一个错误码（-RT_EFULL），发送消息并不带有阻塞机制的，因为发送消息的环境可能是在中断中，不允许有阻塞的情况。

21.4 消息队列的应用场景

消息队列可以应用于发送不定长消息的场合，包括线程与线程间的消息交换，以及在中断服务函数中给线程发送消息（中断服务例程不可能接收消息）。

21.5 消息队列控制块

消息队列控制块包含了每个消息队列的信息，如消息队列名称、内存缓冲区、消息大小以及队列长度等，是很重要的一个内核对象控制块，具体见*代码清单 18-1*。

列表 1: 代码清单 18-1 消息队列控制块

```
struct rt_messagequeue {
    struct rt_ipc_object parent;                    (1)

    void                *msg_pool;                  (2)

    rt_uint16_t          msg_size;                  (3)
    rt_uint16_t          max_msgs;                  (4)

    rt_uint16_t          entry;                     (5)

    void                *msg_queue_head;            (6)
    void                *msg_queue_tail;            (7)
    void                *msg_queue_free;            (8)
};
typedef struct rt_messagequeue *rt_mq_t;
```

代码清单 *18-1* **(1)**：消息队列属于内核对象，会在自身结构体里面包含一个内核对象类型的成员，通过这个成员可以将消息队列挂到系统对象容器里面。

代码清单 *18-1* **(2)**：存放消息的消息池开始地址。

代码清单 *18-1* **(3)**：每条消息大小，消息队列中也就是节点的大小，单位为字节。

代码清单 *18-1* **(4)**：能够容纳的最大消息数量。

代码清单 *18-1* **(5)**：队列中的消息索引，记录消息队列的消息个数。

代码清单 *18-1* **(6)**：链表头指针，指向即将读取数据的节点。

代码清单 *18-1* **(7)**：链表尾指针，指向允许写入数据的节点

代码清单 *18-1* **(8)**：指向队列的空闲节点的指针。

21.6 常用消息队列的函数讲解

使用队列模块的典型流程如下：

- 创建消息队列 rt_mq_create。
- 写队列操作函数 rt_mq_send。
- 读队列操作函数 rt_mq_recv。
- 删除队列 rt_mq_delete。

21.6.1 消息队列创建函数 rt_mq_create()

消息队列创建函数，顾名思义，就是创建一个队列，与线程一样，都是需要先创建才能使用的东西，RT-Thread 肯定不知道我们需要什么样的队列，所以，我们需要怎么样的队列我们就自己创建就行了，比如队列的长度，队列句柄，节点的大小这些信息都是我们自己定义的，RT-Thread 提供给我们这个创建函数，爱怎么搞都是我们自己来自定义的，创建队列的函数源码具体见代码清单 *18-2*。

列表 2: 代码清单 18-2 消息队列创建函数 rt_mq_create() 源码

```
rt_mq_t rt_mq_create(const char *name,
                     rt_size_t   msg_size,
                     rt_size_t   max_msgs,
                     rt_uint8_t  flag)
{
    struct rt_messagequeue *mq;
    struct rt_mq_message *head;
    register rt_base_t temp;

    RT_DEBUG_NOT_IN_INTERRUPT;

    /* 分配消息队列对象 */                                          (1)
    mq = (rt_mq_t)rt_object_allocate(RT_Object_Class_MessageQueue, name);
    if (mq == RT_NULL)
        return mq;

    /* 设置 parent */
    mq->parent.parent.flag = flag;                                  (2)

    /* 初始化消息队列内核对象 */
    rt_ipc_object_init(&(mq->parent));                              (3)

    /* 初始化消息队列 */

    /* 获得正确的消息队列大小 */
    mq->msg_size = RT_ALIGN(msg_size, RT_ALIGN_SIZE);               (4)
    mq->max_msgs = max_msgs;

    /* 分配消息内存池 */
```

(下页继续)

(续上页)

```
    mq->msg_pool = RT_KERNEL_MALLOC((mq->msg_size +
                                    sizeof(struct rt_mq_message)) * mq->max_msgs);
    if (mq->msg_pool == RT_NULL) {                                  (5)
        rt_mq_delete(mq);

        return RT_NULL;
    }

    /* 初始化消息队列头尾链表 */
    mq->msg_queue_head = RT_NULL;                                   (6)
    mq->msg_queue_tail = RT_NULL;

    /* 初始化消息队列空闲链表 */
    mq->msg_queue_free = RT_NULL;
    for (temp = 0; temp < mq->max_msgs; temp ++) {                  (7)
        head = (struct rt_mq_message *)((rt_uint8_t *)mq->msg_pool +
            temp * (mq->msg_size + sizeof(struct rt_mq_message)));
        head->next = mq->msg_queue_free;
        mq->msg_queue_free = head;
    }

    /* 消息队列的个数为 0（清零）*/
    mq->entry = 0;                                                  (8)

    return mq;
}
RTM_EXPORT(rt_mq_create);
```

代码清单 18-2 **(1)**：分配消息队列对象，调用 rt_object_allocate 此函数将从对象系统分配对象，为创建的消息队列分配一个消息队列的对象，并且命名对象名称，在系统中，对象的名称必须是唯一的。

代码清单 18-2 **(2)**：设置消息队列的阻塞唤醒模式，创建的消息队列由于指定的 flag 不同，而有不

同的意义：使用 RT_IPC_FLAG_PRIO 优先级 flag 创建的 IPC 对象，在多个线程等待消息队列资源时，将由优先级高的线程优先获得资源。而使用 RT_IPC_FLAG_FIFO 先进先出 flag 创建的 IPC 对象，在多个线程等待消息队列资源时，将按照先来先得的顺序获得资源。RT_IPC_FLAG_PRIO 与 RT_IPC_FLAG_FIFO 均在 rtdef.h 中有定义。

代码清单 18-2 **(3)**：初始化消息队列内核对象。此处会初始化一个链表，用于记录访问此队列而阻塞的线程，通过这个链表，可以找到对应的阻塞线程的控制块，从而能恢复线程。

代码清单 18-2 **(4)**：设置消息队列的节点大小与消息队列的最大容量，节点大小要按 RT_ALIGN_SIZE 字节对齐，消息队列的容量由用户自己定义。

代码清单 18-2 **(5)**：给此消息队列分配内存。这块内存的大小为 [消息大小 + 消息头大小] 与消息队列容量的乘积，每个消息节点中都有一个消息头，用于链表链接，指向下一个消息节点，作为消息的排序。

代码清单 18-2 **(6)**：初始化消息队列头尾链表。

代码清单 18-2 **(7)**：将所有的消息队列的节点连接起来，形成空闲链表。

代码清单 18-2 **(8)**：消息队列的个数为 0（清零）。

在创建消息队列的时候，是需要用户自己定义消息队列的句柄的，但是注意了，定义了队列的句柄并不等于创建了队列，创建队列必须是调用 rt_mq_create() 函数进行创建，否则，以后根据队列句柄使用队列的其它函数的时候会发生错误，在创建队列的时候是会返回创建的情况的，如果创建成功则返回消息队列句柄，如果是返回 RT_NULL，则表示失败，消息队列创建函数 rt_mq_create() 使用实例具体见*代码清单 18-3* 高亮部分。

列表 3: 代码清单 18-3 消息队列创建函数 rt_mq_create() 实例

```
/* 创建一个消息队列 */
test_mq = rt_mq_create("test_mq",    /* 消息队列名字 */
                       40,           /* 消息的最大长度 */
                       20,           /* 消息队列的最大容量 */
                       RT_IPC_FLAG_FIFO);/* 队列模式 FIFO(0x00)*/
if (test_mq != RT_NULL)
```

(下页继续)

(续上页)

```
    rt_kprintf("消息队列创建成功! \n\n");
```

21.6.2 消息队列删除函数 rt_mq_delete()

队列删除函数是根据消息队列句柄直接删除的，删除之后这个消息队列的所有信息都会被系统回收清空，而且不能再次使用这个消息队列了，但是需要注意的是，如果某个消息队列没有被创建，那也是无法被删除的，动脑子想想都知道，没创建的东西就不存在，怎么可能被删除。删除消息队列的时候会把所有由于访问此消息队列而进入阻塞态的线程都从阻塞链表中删除，mq 是 rt_mq_delete 传入的参数，是消息队列句柄，表示的是要删除哪个想队列，其函数源码具体见*代码清单 18-4*。

列表 4: 代码清单 18-4 消息队列删除函数 rt_mq_delete() 源码

```
rt_err_t rt_mq_delete(rt_mq_t mq)
{
    RT_DEBUG_NOT_IN_INTERRUPT;

    /* 检查消息队列 */
    RT_ASSERT(mq != RT_NULL);                                   (1)

    /* 恢复所有因为访问此队列而阻塞的线程 */
    rt_ipc_list_resume_all(&(mq->parent.suspend_thread));    (2)

#if defined(RT_USING_MODULE) && defined(RT_USING_SLAB)
    /*  消息队列对象属于应用程序模块 ，此处不使用 */
    if (mq->parent.parent.flag & RT_OBJECT_FLAG_MODULE)
        rt_module_free(mq->parent.parent.module_id, mq->msg_pool);
    else
#endif

```

(下页继续)

(续上页)

```
        /* 释放消息队列内存 */
        RT_KERNEL_FREE(mq->msg_pool);                    (3)

    /* 删除消息队列对象 */
    rt_object_delete(&(mq->parent.parent));              (4)

    return RT_EOK;
}
```

代码清单 18-4 **(1)**：检测消息队列是否被创建了，如果是则可以进行删除操作。

代码清单 18-4 **(2)**：调用 rt_ipc_list_resume_all() 函数将所有因为访问此队列的而阻塞的线程从阻塞态中恢复过来，线程得到队列返回的错误代码。在实际情况一般不这样子使用，在删除的时候，应先确认所有的线程都无需再次访问此队列，并且此时没有线程被此队列阻塞，才进行删除操作。

代码清单 18-4 **(3)**：删除了消息队列，那肯定要把消息队列的内存释放出来，毕竟嵌入式设备的内存是很珍贵的。

代码清单 18-4 **(4)**：删除消息队列对象并且释放消息队列内核对象的内存，释放内核对象内存在 rt_object_delete() 函数中实现。

消息队列删除函数 rt_mq_delete() 的使用也是很简单的，只需传入要删除的消息队列的句柄即可，调用这个函数时，系统将删除这个消息队列。如果删除该消息队列时，有线程正在等待消息，那么删除操作会先唤醒等待在消息队列量上的线程（等待线程的返回值是-RT_ERROR），具体见代码清单 18-5 高亮部分。

列表 5: 代码清单 18-5 消息队列删除函数 rt_mq_delete() 实例

```
/* 定义消息队列控制块 */
static rt_mq_t test_mq = RT_NULL;

rt_err_t uwRet = RT_EOK;
```

(下页继续)

(续上页)

```

uwRet = rt_mq_delete(test_mq);
if (RT_EOK == uwRet)
    rt_kprintf("消息队列删除成功! \n\n");
```

21.6.3 消息队列发送消息函数 rt_mq_send()

线程或者中断服务程序都可以给消息队列发送消息。当发送消息时，消息队列对象先从空闲消息链表上取下一个空闲消息块，把线程或者中断服务程序发送的消息内容复制到消息块上，然后把该消息块挂到消息队列的尾部。当且仅当空闲消息链表上有可用的空闲消息块时，发送者才能成功发送消息；当空闲消息链表上无可用消息块，说明消息队列已满，此时，发送消息的的线程或者中断程序会收到一个错误码（-RT_EFULL），消息队列发送消息函数 rt_mq_send() 源码具体见*代码清单 18-6*。

列表 6: 代码清单 18-6 消息队列发送消息函数 rt_mq_send() 源码

```
rt_err_t rt_mq_send(rt_mq_t mq, void *buffer, rt_size_t size)(1)
{
    register rt_ubase_t temp;
    struct rt_mq_message *msg;

    RT_ASSERT(mq != RT_NULL);                                   (2)
    RT_ASSERT(buffer != RT_NULL);
    RT_ASSERT(size != 0);

    /* 判断消息的大小 */
    if (size > mq->msg_size)                                    (3)
        return -RT_ERROR;

    RT_OBJECT_HOOK_CALL(rt_object_put_hook, (&(mq->parent.parent)));
```

(下页继续)

(续上页)

```
    /* 关中断 */
    temp = rt_hw_interrupt_disable();

    /* 获取一个空闲链表，必须有一个空闲链表项 */
    msg = (struct rt_mq_message *)mq->msg_queue_free;       (4)
    /* 消息队列满 */
    if (msg == RT_NULL) {
        /* 开中断 */
        rt_hw_interrupt_enable(temp);

        return -RT_EFULL;
    }
    /* 移动空闲链表指针 */
    mq->msg_queue_free = msg->next;                         (5)

    /* 开中断 */
    rt_hw_interrupt_enable(temp);

    /* 这个消息是新的链表尾部，其下一个指针为 RT_NULL /
    msg->next = RT_NULL;
    /* 拷贝数据 */
    rt_memcpy(msg + 1, buffer, size);                       (6)

    /* 关中断 */
    temp = rt_hw_interrupt_disable();
    /* 将消息挂载到消息队列尾部 */
    if (mq->msg_queue_tail != RT_NULL) {                    (7)
        /* 如果已经存在消息队列尾部链表 */
        ((struct rt_mq_message *)mq->msg_queue_tail)->next = msg;
    }

```

(下页继续)

(续上页)

```
    /* 设置新的消息队列尾部链表指针 */
    mq->msg_queue_tail = msg;                                   (8)
    /*  如果头部链表是空的，设置头部链表指针 */
    if (mq->msg_queue_head == RT_NULL)                          (9)
        mq->msg_queue_head = msg;

    /* 增加消息数量记录 */
    mq->entry ++;                                               (10)

    /* 恢复挂起线程 */
    if (!rt_list_isempty(&mq->parent.suspend_thread)) {         (11)
        rt_ipc_list_resume(&(mq->parent.suspend_thread));

        /* 开中断 */
        rt_hw_interrupt_enable(temp);

        rt_schedule();                                          (12)

        return RT_EOK;
    }

    /* 开中断 */
    rt_hw_interrupt_enable(temp);

    return RT_EOK;
}
RTM_EXPORT(rt_mq_send);
```

代码清单 18-6 **(1)**：在发送消息的时候需要传递一些参数：rt_mq_t mq 是已经创建的消息队列句柄；void *buffer 是即将发送消息的存储地址；rt_size_t size 是即将发送消息的大小。

代码清单 18-6 **(2)**：检测传递进来的参数，如果这些参数之中有一个是无效的，都无法发送消息。

代码清单 18-6 **(3)**：判断消息的大小，其大小不能超过创建时候设置的消息队列的大小 mq->msg_size，用户可以自定义大小的，如果 mq->msg_size 不够，可以在创建时候设置大一些。

代码清单 18-6 **(4)**：获取一个空闲链表指针，必须有一个空闲链表节点用于存放要发送的消息。如果消息队列已经满了，则无法发送消息。

代码清单 18-6 **(5)**：移动空闲链表指针。

代码清单 18-6 **(6)**：拷贝数据，将即将发送的数据拷贝到空闲链表的节点中，因为空闲节点有消息头，所以其真正存放消息的地址是 msg + 1。

代码清单 18-6 **(7)**：将空闲队列的消息挂载到消息队列尾部，如果此时消息队列已经有消息，也就是尾部链表不为空，那么就直接将发送的消息挂载到尾部链表后面。

代码清单 18-6 **(8)**：重置消息队列尾链表指针，指向当前发送的消息，无论当前消息队列中尾链表是否有消息，都需要重置尾链表指针的指向。

代码清单 18-6 **(9)**：如果连头链表是空的，就需要设置头部链表指针指向当前要发送的消息，也就是指向消息自身。

代码清单 18-6 **(10)**：记录当前消息队列的消息个数，自加 1。

代码清单 18-6 **(11)**：恢复挂起线程。如果当前有线程因为访问队列而进入阻塞，现在有消息了则可以将该线程从阻塞中恢复。

代码清单 18-6 **(12)**：发起一次线程调度。

发送消息时，发送者需指定发送到的消息队列的对象句柄（即指向消息队列控制块的指针），并且指定发送的消息内容以及消息大小，在发送一个普通消息之后，空闲消息链表上的消息被转移到了消息队列尾链表上，消息队列发送消息函数 rt_mq_send() 的实例具体见代码清单 18-7 高亮部分。

列表 7: 代码清单 18-7 消息队列发送消息函数 rt_mq_send() 实例

```
static void send_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
```

(下页继续)

(续上页)

```
    uint32_t send_data1 = 1;
    uint32_t send_data2 = 2;
    while (1) {/* K1 被按下 */
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_GPIO_PIN) == KEY_ON ) {
            /* 将数据写入（发送）到队列中，等待时间为 0  */
            uwRet = rt_mq_send(test_mq,      /* 写入（发送）队列的 ID(句柄) */
                               &send_data1, /* 写入（发送）的数据 */
                               sizeof(send_data1)); /* 数据的长度 */
            if (RT_EOK != uwRet) {
                rt_kprintf("数据不能发送到消息队列！错误代码：%lx\n",uwRet);
            }
        }/* K1 被按下 */
        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_GPIO_PIN) == KEY_ON ) {
            /* 将数据写入（发送）到队列中，等待时间为 0  */
            uwRet = rt_mq_send(test_mq,      /* 写入（发送）队列的 ID(句柄) */
                                &send_data2, /* 写入（发送）的数据 */
                                sizeof(send_data2)); /* 数据的长度 */
            if (RT_EOK != uwRet) {
                rt_kprintf("数据不能发送到消息队列！错误代码：%lx\n",uwRet);
            }
        }
        rt_thread_delay(20);
    }
}
```

21.6.4 消息队列接收消息函数 rt_mq_recv()

当消息队列中有消息时，接收线程才能接收到消息，接收消息是有阻塞机制的，用户可以自定义等待时间，RT-Thread 的接收消息过程是：接收一个消息后消息队列的头链表消息被转移到了空闲消息链表中，其源码实现具体见*代码清单 18-8*。

列表 8: 代码清单 18-8 消息队列接收消息函数 rt_mq_recv() 源码

```
rt_err_t rt_mq_recv(rt_mq_t     mq,                                (1)
                    void       *buffer,                            (2)
                    rt_size_t   size,                              (3)
                    rt_int32_t timeout)                            (4)
{
    struct rt_thread *thread;
    register rt_ubase_t temp;
    struct rt_mq_message *msg;
    rt_uint32_t tick_delta;

    RT_ASSERT(mq != RT_NULL);
    RT_ASSERT(buffer != RT_NULL);
    RT_ASSERT(size != 0);                                          (5)

    tick_delta = 0;
    /* 获取当前的线程 */
    thread = rt_thread_self();                                     (6)
    RT_OBJECT_HOOK_CALL(rt_object_trytake_hook, (&(mq->parent.parent)));

    /* 关中断 */
    temp = rt_hw_interrupt_disable();

    /* 非阻塞情况 */
    if (mq->entry == 0 && timeout == 0) {                          (7)
        rt_hw_interrupt_enable(temp);

        return -RT_ETIMEOUT;
    }

```

(下页继续)

(续上页)

```
    /* 消息队列为空 */
    while (mq->entry == 0) {                                    (8)
        RT_DEBUG_IN_THREAD_CONTEXT;

        /* 重置线程中的错误号 */
        thread->error = RT_EOK;                                 (9)

        /* 不等待 */
        if (timeout == 0) {
            /* 开中断 */
            rt_hw_interrupt_enable(temp);

            thread->error = -RT_ETIMEOUT;

            return -RT_ETIMEOUT;
        }

        /* 挂起当前线程 */
        rt_ipc_list_suspend(&(mq->parent.suspend_thread),       (10)
                            thread,
                            mq->parent.parent.flag);

        /* 有等待时间，启动线程计时器 */
        if (timeout > 0) {                                      (11)
            /* 获取 systick 定时器时间 */
            tick_delta = rt_tick_get();

            RT_DEBUG_LOG(RT_DEBUG_IPC, ("set thread:%s to timer list\n",
                                        thread->name));

            /* 重置线程计时器的超时并启动它 */
            rt_timer_control(&(thread->thread_timer),           (12)
```

(下页继续)

(续上页)

```
                                RT_TIMER_CTRL_SET_TIME,
                                &timeout);
            rt_timer_start(&(thread->thread_timer));
        }

        /* 开中断 */
        rt_hw_interrupt_enable(temp);

        /* 发起线程调度 */
        rt_schedule();                                          (13)


        if (thread->error != RT_EOK) {
            /* 返回错误 */
            return thread->error;
        }

        /* 关中断 */
        temp = rt_hw_interrupt_disable();

        /* 如果它不是永远等待，然后重新计算超时滴答 */
        if (timeout > 0) {
            tick_delta = rt_tick_get() - tick_delta;
            timeout -= tick_delta;
            if (timeout < 0)
                timeout = 0;
        }
    }

    /* 获取消息 */
    msg = (struct rt_mq_message *)mq->msg_queue_head;           (14)

```

(下页继续)

(续上页)

```
    /* 移动消息队列头链表指针 */
    mq->msg_queue_head = msg->next;                          (15)
    /* 到达队列尾部，设置为 NULL  */
    if (mq->msg_queue_tail == msg)                           (16)
        mq->msg_queue_tail = RT_NULL;

    /* 记录消息个数，自减一 */
    mq->entry --;                                            (17)

    /* 开中断 */
    rt_hw_interrupt_enable(temp);

    /* 拷贝消息到指定存储地址 */
    rt_memcpy(buffer, msg + 1, size > mq->msg_size ? mq->msg_size : size);␣
↪(18)

    /* 关中断 */
    temp = rt_hw_interrupt_disable();
    /* 移到空闲链表 */
    msg->next = (struct rt_mq_message *)mq->msg_queue_free; (19)
    mq->msg_queue_free = msg;
    /* 开中断 */
    rt_hw_interrupt_enable(temp);

    RT_OBJECT_HOOK_CALL(rt_object_take_hook, (&(mq->parent.parent)));

    return RT_EOK;
}
RTM_EXPORT(rt_mq_recv);
```

代码清单 18-8 **(1)**：消息队列对象的句柄。

代码清单 18-8 **(2)**：buffer 是用于接收消息的数据存储地址，必须在接收之前就定义了，确保地址

有效。

代码清单 *18-8* **(3)**：消息大小。

代码清单 *18-8* **(4)**：指定超时时间。

代码清单 *18-8* **(5)**：检测传递进来的参数是否有效，有效才进行消息队列的数据读取。

代码清单 *18-8* **(6)**：获取当前运行的线程。

代码清单 *18-8* **(7)**：如果当前消息队列中没有消息并且设置了不等待，则立即返回错误代码。

代码清单 *18-8* **(8)**：如果消息队列为空，但是用户设置了等待时间，则进入循环中。

代码清单 *18-8* **(9)**：重置线程中的错误码。

代码清单 *18-8* **(10)**：挂起当前线程，因为当前线程是由于消息队列为空，并且用户设置了超时时间，直接将当前线程挂起，进入阻塞状态。

代码清单 *18-8* **(11)**：用户有设置等待时间，需要启动线程计时器，并且调用 rt_tick_get() 函数获取当前系统 systick 时间。

代码清单 *18-8* **(12)**：重置线程计时器的超时并启动它，调用 rt_timer_control() 函数改变当前线程阻塞时间，阻塞的时间根据用户自定义的 timeout 设置，并且调用 rt_timer_start() 函数开始定时。

代码清单 *18-8* **(13)**：发起一次线程调度。当前线程都已经挂起了，需要进行线程切换。

代码清单 *18-8* **(14)**：如果当前消息队列中有消息，那么获取消息队列的线程可以直接从消息队列的 msg_queue_head 链表获取到消息，并不会进入阻塞态中。

代码清单 *18-8* **(15)**：移动消息队列头链表指针。重置消息队列的 msg_queue_head 指向当前消息的下一个消息。因为当前的消息被取走了，下一个消息才是可获取的有效消息。

代码清单 *18-8* **(16)**：如果到达队列尾部，则将消息队列的 msg_queue_tail 设置为 NULL。

代码清单 *18-8* **(17)**：记录当前消息队列中消息的个数，entry 减一，消息就是获取了一个就少一个。

代码清单 *18-8* **(18)**：拷贝消息到指定存储地址 buffer，拷贝消息的大小为 size，其大小最大不能超过创建消息队列时候已经定义的消息大小 msg_size。

代码清单 18-8 **(19)**：获取一个消息后，消息队列上的头链表消息被转移到空闲消息链表中，相当消息的删除操作，这样子可以保证消息队列的循环利用，而不会导致头链表指针移动到队列尾部时没有可用的消息节点。

根据这些函数源码，我们能很轻松对它进行使用操作，下面让我们来进行队列接收操作吧，这个函数用于读取指定队列中的数据，并将获取的数据存储到 buffer 指定的地址。要读取的数据的地址和大小为 size，由用户定义，具体使用实例见代码清单 18-9 高亮部分。

列表 9: 代码清单 18-9 消息队列接收消息函数 rt_mq_recv() 实例

```
/* 队列读取（接收），等待时间为一直等待 */
uwRet = rt_mq_recv(test_mq, /* 读取（接收）队列的 ID(句柄) */
                   &r_queue,    /* 读取（接收）的数据保存位置 */
                   sizeof(r_queue), /* 读取（接收）的数据的长度 */
                   RT_WAITING_FOREVER); /* 等待时间：一直等 */
if (RT_EOK == uwRet)
{
    rt_kprintf("本次接收到的数据是：%d\n",r_queue);
} else
{
    rt_kprintf("数据接收出错，错误代码：0x%lx\n",uwRet);
}
```

21.7 消息队列使用注意事项

在使用 RT-Thread 提供的消息队列函数的时候，需要了解以下几点：

1. 使用 rt_mq_recv()、rt_mq_send()、rt_mq_delete() 等这些函数之前应先创建需消息队列，并根据队列句柄进行操作。

2. 队列读取采用的是先进先出（FIFO）模式，会首先读取出首先存储在队列中的数据。当然也有例外，RT-Thread 给我们提供了另一个函数，可以发送紧急消息的，那么读取的时候就会读取到

紧急消息的数据。

3. 必须要我们定义一个存储读取出来的数据的地方，并且把存储数据的起始地址传递给 rt_mq_recv() 函数，否则，将发生地址非法的错误。

4. 接收消息队列中的消息是拷贝的方式，读取消息时候定义的地址必须保证能存放下即将读取消息的大小。

21.8 消息队列实验

消息队列实验是在 RT-Thread 中创建了两个线程，一个是发送消息线程，一个是获取消息线程，两个线程独立运行，发送消息线程是通过检测按键的按下情况来发送消息，假如发送消息不成功，就把返回的错误情代码在串口打印出来，另一个线程是获取消息线程，在消息队列没有消息之前一直等待消息，一旦获取到消息就把消息打印在串口调试助手里，具体见*代码清单 18-10* 高亮部分。

注意：在使用消息队列时候请确保在 rtconfig.h 中打开 RT_USING_MESSAGEQUEUE 这个宏定义。

列表 10: 代码清单 18-10 消息队列实验

```
/**
*****************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2019-xx-xx
* @brief   RT-Thread 消息队列
*****************************************************************
* @attention
*
* 实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-i.MX RT.taobao.com
```

(下页继续)

(续上页)

```
14  *
15  ************************************************************************
16  */
17
18  /*
19  ************************************************************************
20  *                                包含的头文件
21  ************************************************************************
22  */
23  #include "board.h"
24  #include "fsl_debug_console.h"
25  /* RT-Thread 头文件 */
26  #include "rtthread.h"
27  /*
28  ************************************************************************
29  *                                  变量
30  ************************************************************************
31  */
32  /* 定义线程控制块 */
33  static rt_thread_t receive_thread = RT_NULL;
34  static rt_thread_t send_thread = RT_NULL;
35  /* 定义消息队列控制块 */
36  static rt_mq_t test_mq = RT_NULL;
37  /*
38  ************************************************************************
39  *                                函数声明
40  ************************************************************************
41  */
42  static void receive_thread_entry(void* parameter);
43  static void send_thread_entry(void* parameter);
44
45  /*
```

(下页继续)

(续上页)

```
*****************************************************************************
*                                main 函数
*****************************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]-i.MX RT 全系列开发板 RTT 消息队列实验! \n");
    rt_kprintf("按下 K1 或者 K2 发送队列消息\n");
    rt_kprintf("receive 线程接收到消息在串口回显\n");
    /* 创建一个消息队列 */
    test_mq = rt_mq_create("test_mq",/* 消息队列名字 */
                        40,      /* 消息的最大长度 */
                        20,     /* 消息队列的最大容量 */
                        RT_IPC_FLAG_FIFO);/* 队列模式 FIFO(0x00)*/
    if (test_mq != RT_NULL)
        rt_kprintf("消息队列创建成功! \n\n");

    receive_thread =                          /* 线程控制块指针 */
        rt_thread_create( "receive",              /* 线程名字 */
                        receive_thread_entry,   /* 线程入口函数 */
                        RT_NULL,              /* 线程入口函数参数 */
                        512,                  /* 线程栈大小 */
```

(下页继续)

(续上页)

```
                        3,                      /* 线程的优先级 */
                        20);                    /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (receive_thread != RT_NULL)
        rt_thread_startup(receive_thread);
    else
        return -1;

    send_thread =                           /* 线程控制块指针 */
        rt_thread_create( "send",               /* 线程名字 */
                        send_thread_entry,      /* 线程入口函数 */
                        RT_NULL,                /* 线程入口函数参数 */
                        512,                    /* 线程栈大小 */
                        2,                      /* 线程的优先级 */
                        20);                    /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (send_thread != RT_NULL)
        rt_thread_startup(send_thread);
    else
        return -1;
}

/*
*****************************************************************
*                              线程定义
*****************************************************************
*/

static void receive_thread_entry(void* parameter)
{
```

(下页继续)

(续上页)

```
    rt_err_t uwRet = RT_EOK;
    uint32_t r_queue;
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        /* 队列读取（接收），等待时间为一直等待 */
        uwRet = rt_mq_recv(test_mq, /* 读取（接收）队列的 ID(句柄) */
                           &r_queue,        /* 读取（接收）的数据保存位置 */
                           sizeof(r_queue), /* 读取（接收）的数据的长度 */
                           RT_WAITING_FOREVER); /* 等待时间：一直等 */
        if (RT_EOK == uwRet) {
            rt_kprintf("本次接收到的数据是：%d\n",r_queue);
        } else {
            rt_kprintf("数据接收出错，错误代码：0x%lx\n",uwRet);
        }
        rt_thread_delay(200);
    }
}

static void send_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
    uint32_t send_data1 = 1;
    uint32_t send_data2 = 2;
    while (1) { /* K1 被按下 *
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_GPIO_PIN) == KEY_ON ) {/
            /* 将数据写入（发送）到队列中，等待时间为 0  */
            uwRet = rt_mq_send(test_mq,/* 写入（发送）队列的 ID(句柄) */
                               &send_data1,/* 写入（发送）的数据 */
                               sizeof(send_data1)); /* 数据的长度 */
            if (RT_EOK != uwRet) {
                rt_kprintf("数据不能发送到消息队列！错误代码：%lx\n",uwRet);
            }
```

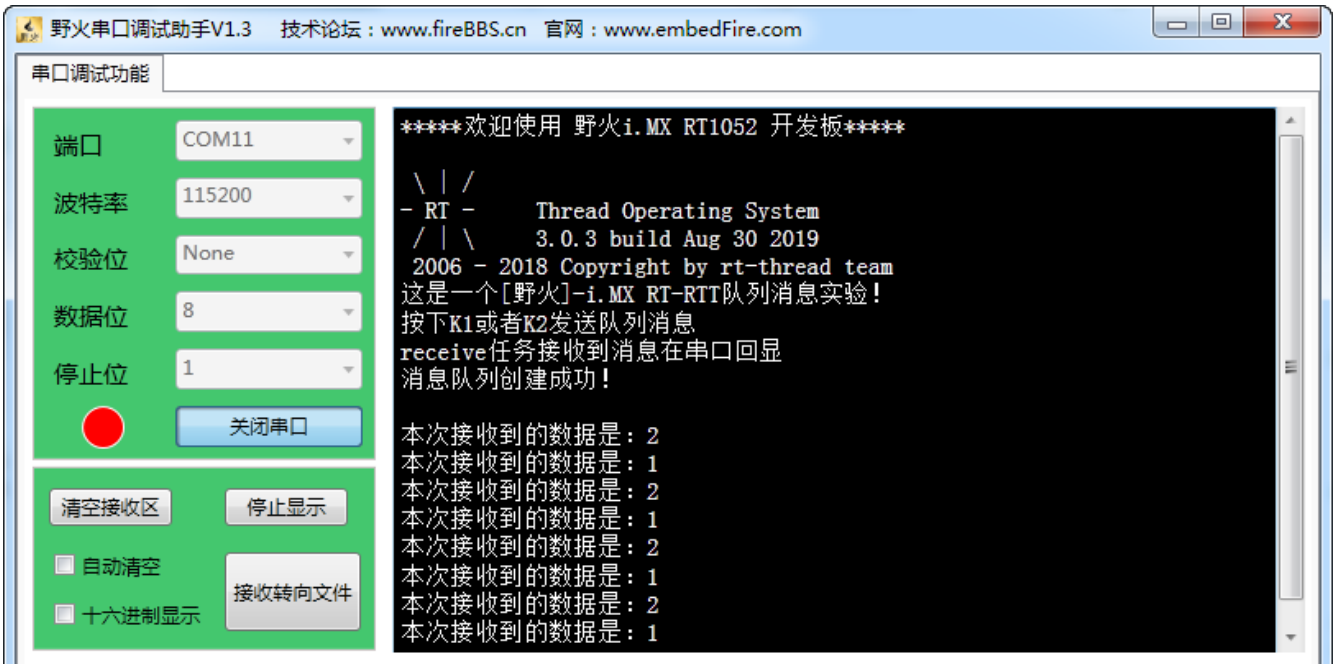
(下页继续)

(续上页)

```
        }/* K2 被按下 */
        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_GPIO_PIN) == KEY_ON ) {
            /* 将数据写入（发送）到队列中，等待时间为 0  */
            uwRet = rt_mq_send(test_mq,      /* 写入（发送）队列的 ID(句柄) */
                               &send_data2,         /* 写入（发送）的数据 */
                               sizeof(send_data2));         /* 数据的长度 */
            if (RT_EOK != uwRet) {
                rt_kprintf("数据不能发送到消息队列！错误代码：%lx\n",uwRet);
            }
        }
        rt_thread_delay(20);
    }
}
/***************************END OF FILE************************/
```

21.9 实验现象

将程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息，按下开发版的 K1 按键发送消息 1，按下 K2 按键发送消息 2；我们按下 K1 试试，在串口调试助手中可以看到接收到消息 1，我们按下 K2 试试，在串口调试助手中可以看到接收到消息 2，具体见图 *18-2*。


野火串口调试助手V1.3 技术论坛：www.fireBBS.cn 官网：www.embedFire.com
串口调试功能
端口
COM11
波特率
115200
校验位
None
数据位
8
停止位
1
关闭串口
清空接收区
停止显示
自动清空
十六进制显示
接收转向文件
*****欢迎使用 野火i.MX RT1052 开发板*****
\ | /
- RT - Thread Operating System
/ | \ 3.0.3 build Aug 30 2019
2006 - 2018 Copyright by rt-thread team
这是一个[野火]-i.MX RT-RTT队列消息实验！
按下K1或者K2发送队列消息
receive任务接收到消息在串口回显
消息队列创建成功！
本次接收到的数据是：2
本次接收到的数据是：1
本次接收到的数据是：2
本次接收到的数据是：1
本次接收到的数据是：2
本次接收到的数据是：1
本次接收到的数据是：2
本次接收到的数据是：1

第 22 章　信号量

同志们，回想一下，你是否在裸机编程中这样使用过一个变量：用于标记某个事件是否发生，或者标志一下某个东西是否正在被使用，如果是被占用了的或者没发生，我们就不对它进行操作。

22.1 信号量基本概念

信号量（Semaphore）是一种实现线程间通信的机制，实现线程之间同步或临界资源的互斥访问，常用于协助一组相互竞争的线程来访问临界资源。在多线程系统中，各线程之间需要同步或互斥实现临界资源的保护，信号量功能可以为用户提供这方面的支持。

通常一个信号量的计数值用于对应有效的资源数，表示剩下的可被占用的互斥资源数。其值的含义分两种情况：

- 0：表示没有积累下来的 release 释放信号量操作，且有可能有在此信号量上阻塞的线程。
- 正值，表示有一个或多个 release 释放信号量操作。

以同步为目的的信号量和以互斥为目的的信号量在使用有如下不同：

- 用作互斥时，信号量创建后可用信号量个数应该是满的，线程在需要使用临界资源时，先获取信号量，使其变空，这样其他线程需要使用临界资源时就会因为无法获取信号量而进入阻塞，从而保证了临界资源的安全。但是这样子有一个缺点就是有可能产生优先级翻转，优先级翻转的危害具体会在互斥量章节中详细讲解。
- 用作同步时，信号量在创建后被置为空，线程 1 取信号量而阻塞，线程 2 在某种条件发生后，释放信号量，于是线程 1 得以进入就绪态，如果线程 1 的优先级是最高的，那么就会立即切换线程，从而达到了两个线程间的同步。同样的，在中断服务函数中释放信号量，也能达到线程与中断间的同步。

在操作系统中，我们使用信号量的目的是为了给临界资源建立一个标志，信号量表示了该临界资源被占用情况。这样，当一个线程在访问临界资源的时候，就会先对这个资源信息进行查询，从

而在了解资源被占用的情况之后，再做处理，从而使得临界资源得到有效的保护。

还记得我们经常说的中断要快进快出吗，在裸机开发中我们经常是在中断中做一个标记，然后在退出的时候进行轮询处理，这个就是类似我们使用信号量进行同步的，当标记发生了，我们再做其他事情。在 RT-Thread 我们用信号量用于同步，线程与线程的同步，中断与线程的同步，可以大大提高效率。

信号量还有计数型信号量，计数型信号量允许多个线程对其进行操作，但限制了线程的数量。比如有一个停车场，里面只有 100 个车位，那么能停的车只有 100 辆，也相当于我们的信号量有 100 个，假如一开始停车场的车位还有 100 个，那么每进去一辆车就要消耗一个停车位，车位的数量就要减一，对应的，我们的信号量在使用之后也需要减一，当停车场停满了 100 辆车的时候，此时的停车位为 0，再来的车就不能停进去了，否则将造成事故，也相当于我们的信号量为 0，后面的线程对这个停车场资源的访问也无法进行，当有车从停车场离开的时候，车位又空余出来了，那么，后面的车就能停进去了，在我们信号量的操作也是一样的，当我们释放了这个资源，后面的线程才能对这个资源进行访问。

22.2 二值信号量的应用场景

在嵌入式操作系统中二值信号量是线程间、线程与中断间同步的重要手段。为什么叫二值信号量呢？因为信号量资源被获取了，信号量值就是 0，信号量资源被释放，信号量值就是 1，把这种只有 0 和 1 两种情况的信号量称之为二值信号量。

在线程系统中，我们经常会使用这个二值信号量，比如，某个线程需要等待一个标记，那么线程可以在轮询中查询这个标记有没有被置位，这样子做，就会很消耗 CPU 资源，其实根本不需要在轮询中查询这个标记，只需要使用二值信号量即可，当二值信号量没有的时候，线程进入阻塞态等待二值信号量到来即可，当得到了这个信号量（标记）之后，在进行线程的处理即可，这样子么就不会消耗太多资源了，而且实时响应也是最快的。

再比如某个线程使用信号量在等中断的标记的发生，在这之前线程已经进入了阻塞态，在等待着中断的发生，当在中断发生之后，释放一个信号量，也就是我们常说的标记，当它退出中断之后，操作系统进行线程的调度，如果这个线程能够运行，系统就会把等待这个线程运行起来，这样子就大大提高了我们的效率。

二值信号量在线程与线程中同步的应用场景：假设我们有一个温湿度的传感器，假设是 1s 采集

一次数据，那么我们让他在液晶屏中显示数据出来，这个周期也是要 1s 一次的，如果液晶屏刷新的周期是 100ms 更新一次，那么此时的温湿度的数据还没更新，液晶屏根本无需刷新，只需要在 1s 后温湿度数据更新的时候刷新即可，否则 CPU 就是白白做了多次的无效数据更新，CPU 的资源就被刷新数据这个线程占用了大半，造成 CPU 资源浪费，如果液晶屏刷新的周期是 10s 更新一次，那么温湿度的数据都变化了 10 次，液晶屏才来更新数据，那拿这个产品有啥用，根本就是不准确的，所以，还是需要同步协调工作，在温湿度采集完毕之后，进行液晶屏数据的刷新，这样子，才是最准确的，并且不会浪费 CPU 的资源。

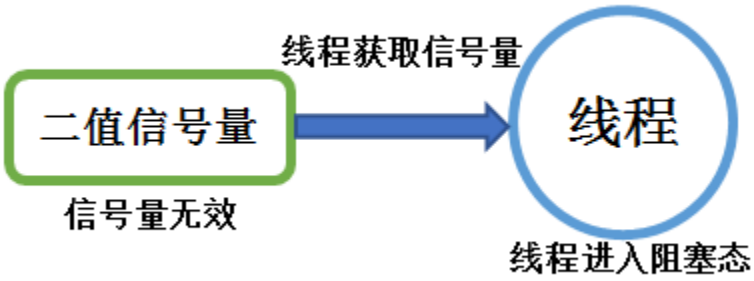
同理，二值信号量在线程与中断同步的应用场景：我们在串口接收中，我们不知道啥时候有数据发送过来，有一个线程是做接收这些数据处理，总不能在线程中每时每刻都在线程查询有没有数据到来，那样会浪费 CPU 资源，所以在这种情况下使用二值信号量是很好的办法，当没有数据到来的时候，线程就进入阻塞态，不参与线程的调度，等到数据到来了，释放一个二值信号量，线程就立即从阻塞态中解除，进入就绪态，然后运行的时候处理数据，这样子系统的资源就会很好的被利用起来。

22.3 二值信号量的运作机制

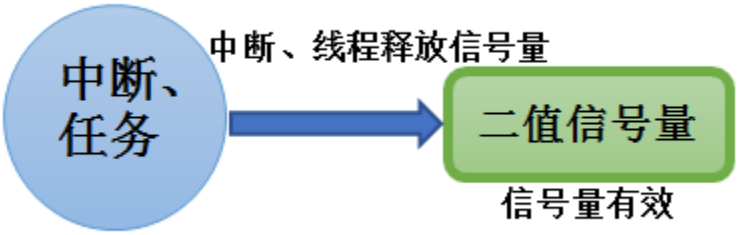
创建二值信号量，为创建的信号量对象分配内存，并把可用信号量初始化为用户自定义的个数，二值信号量的最大可用信号量个数为 1。

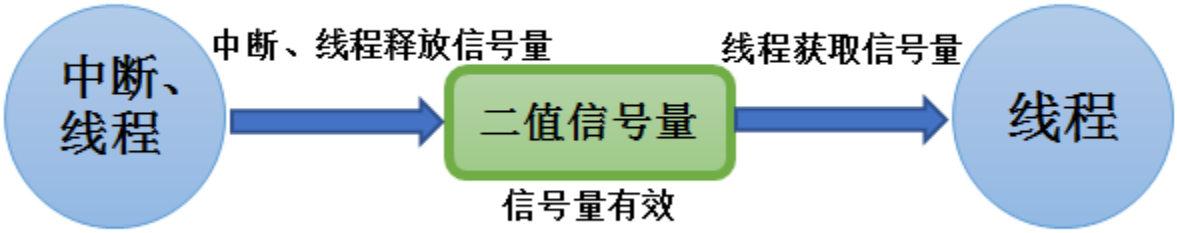
信号量获取，从创建的信号量资源中获取一个信号量，获取成功返回正确。否则线程会等待其它线程释放该信号量，超时时间由用户设定。当线程获取信号量失败时，线程将进入阻塞态，系统将线程挂到该信号量的阻塞列表中。

在二值信号量无效的时候，假如此时有线程获取该信号量的话，那么线程将进入阻塞状态，具体见图 *19-1*。



假如某个时间中断/线程释放了信号量，其过程具体见图 *19-2*，那么，由于获取无效信号量而进入阻塞态的线程将获得信号量并且恢复为就绪态，其过程具体见图 *19-3*。

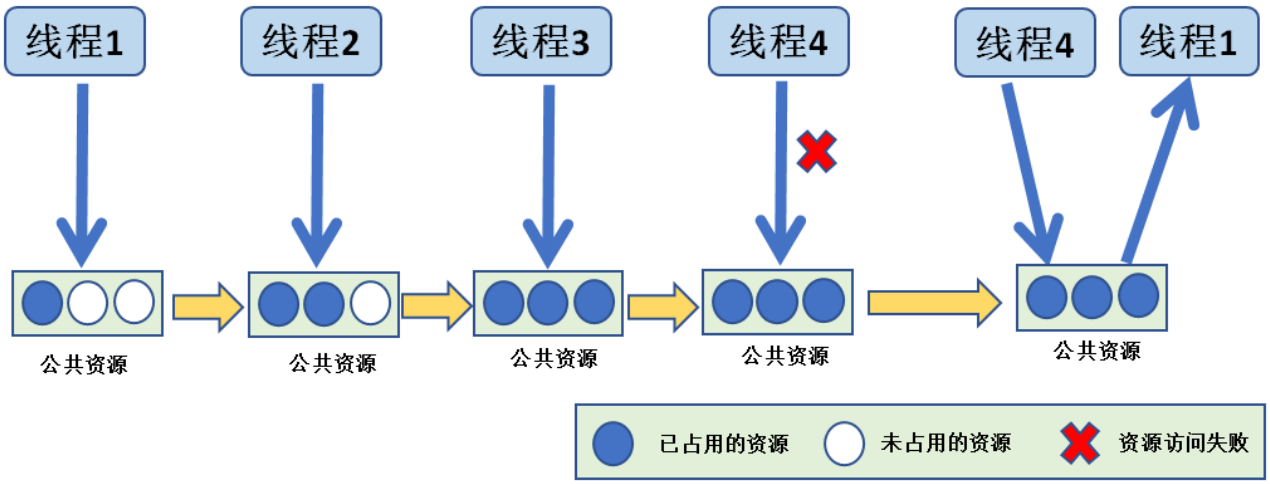




22.4 计数型信号量的运作机制

计数型信号量与二值信号量其实都是差不多的，一样用于资源保护，不过计数信号量则允许多个线程获取信号量访问共享资源，但会限制线程的最大数目。访问的线程数达到信号量可支持的最大数目时，会阻塞其他试图获取该信号量的线程，直到有线程释放了信号量。这就是计数型信号

量的运作机制，虽然计数信号量允许多个线程访问同一个资源，但是也有限定，比如某个资源限定只能有 3 个线程访问，那么第 4 个线程访问的时候，会因为获取不到信号量而进入阻塞，等到有线程（比如线程 1）释放掉该资源的时候，第 4 个线程才能获取到信号量从而进行资源的访问，其运作的机制具体见图 *19-4*。



22.5 信号量控制块

说到信号量的使用就不得不说一下信号量的控制块了，信号量控制块与线程控制类似，每一个信号量都有自己的信号量控制块，信号量控制块中包含了信号量的所有信息，比如信号量的一些状态信息，使用情况等，具体见*代码清单 19-1*。

列表 1: 代码清单 19-1 信号量控制块

```
struct rt_semaphore {
    struct rt_ipc_object parent; /**< 继承自 ipc_object 类 */

    rt_uint16_t          value; /**< 信号量的值，最大为 65535 */
};
typedef struct rt_semaphore *rt_sem_t;
```

信号量属于内核对象，也会在自身结构体里面包含一个内核对象类型的成员，通过这个成员可以将信号量挂到系统对象容器里面。rt_semaphore 对象从 rt_ipc_object 中派生，由 IPC 容器管理。

22.6 常用信号量函数接口讲解

在 RT-Thread 中，无论是二值信号量还是计数信号量，都是由我们自己创建的，二值信号量的最大计数值为 1，并且都是使用 RT-Thread 的同一个释放与获取函数，所以在将信号量当二值信号量使用的时候要注意：用完信号量及时释放，并且不要调用多次信号量释放函数。

22.6.1 信号量创建函数 rt_sem_create()

二值信号量的创建很简单，因为创建的是二值的信号量，所以该信号量的容量只有一个，其可用信号量个数要么是 0，要么是 1，而计数信号量则可以由用户决定在创建的时候初始化多少个可用信号量，其源码具体见*代码清单 19-2*。

列表 2: 代码清单 19-2 信号量创建函数 rt_sem_create() 源码

```
rt_sem_t rt_sem_create(const char *name,                    (1)
                       rt_uint32_t value,                   (2)
                       rt_uint8_t flag)                     (3)
```

(下页继续)

(续上页)

```
{
    rt_sem_t sem;

    RT_DEBUG_NOT_IN_INTERRUPT;

    /* 分配内核对象 */
    sem = (rt_sem_t)rt_object_allocate(RT_Object_Class_Semaphore, name);
    if (sem == RT_NULL)                                          (4)
        return sem;

    /* 初始化信号量对象 */
    rt_ipc_object_init(&(sem->parent));                          (5)

    /* 设置可用信号量的值 */
    sem->value = value;                                          (6)

    /* 设置信号量模式 */
    sem->parent.parent.flag = flag;                              (7)

    return sem;                                                  (8)
}
```

代码清单 *19-2* **(1)**：信号量名称。

代码清单 *19-2* **(2)**：可用信号量初始值。

代码清单 *19-2* **(3)**：信号量标志。

代码清单 *19-2* **(4)**：分配消息队列对象，调用 rt_object_allocate 此函数将从对象系统分配对象，为创建的消息队列分配一个消息队列的对象，并且命名对象名称，在系统中，对象的名称必须是唯一的。

代码清单 *19-2* **(5)**：初始化信号量对象。此处会初始化一个链表用于记录访问此信号量而阻塞的线程。

代码清单 *19-2* **(6)**：设置可用信号量的初始值。表示在创建成功的时候有多少个信号量可用，如果创建的是二值信号量，其取值范围为 [0,1]，如果是计数信号量，其取值范围为 [0,65535]。

代码清单 *19-2* **(7)**：设置信号量的阻塞唤醒模式，创建的信号量由于指定的 flag 不同，而有不同的意义：使用 RT_IPC_FLAG_PRIO 优先级 flag 创建的 IPC 对象，在多个线程等待信号量资源时，将由优先级高的线程优先获得资源。而使用 RT_IPC_FLAG_FIFO 先进先出 flag 创建的 IPC 对象，在多个线程等信号量待资源时，将按照先来先得的顺序获得资源。RT_IPC_FLAG_PRIO 与 RT_IPC_FLAG_FIFO 均在 rtdef.h 中有定义。

代码清单 *19-2* **(8)**：创建成功返回信号量句柄。

通过上面的信号量创建的学习，在创建信号量的时候，我们只需要传入我们的信号量名称、初始化的值和阻塞唤醒发生即可。在创建信号量的时候，是需要用户自己定义信号量的句柄的，但是注意了，定义了信号量的句柄并不等于创建了信号量，创建信号量必须是调用 rt_sem_create() 函数进行创建，需要注意的是：二值信号量可用个数的取值范围是 0~1，计数信号量可用个数的取值范围是 0~65535，用户可以根据需求选择。信号量创建的实例具体见代码清单 *19-3* 高亮部分。

列表 3: 代码清单 19-3 信号量创建函数 rt_sem_create() 实例

```
/* 定义信号量控制块 */
static rt_sem_t test_sem = RT_NULL;
/* 创建一个信号量 */
test_sem = rt_sem_create("test_sem",/* 信号量名字 */
                         1,      /* 信号量初始值，默认有一个信号量 */
                         RT_IPC_FLAG_FIFO); /* 信号量模式 FIFO(0x00)*/
if (test_sem != RT_NULL)
    rt_kprintf("信号量创建成功！\n\n");
```

22.6.2 信号量删除函数 rt_sem_delete()

信号量删除函数是根据信号量句柄直接删除的，删除之后这个信号量的所有信息都会被系统回收，并且用户无法再次使用这个信号量。但是需要注意的是，如果某个信号量没有被创建，那是无法被删除的，动脑子想想都知道，没创建的东西就不存在，怎么可能被删除，删除信号量的时候会把所有由于访问此信号量而阻塞的线程从阻塞链表中删除，并且返回一个错误代码。sem 是 rt_sem_delete() 传入的参数，是信号量句柄，表示的是要删除哪个信号量，其函数源码见*代码清单 19-4*

列表 4: 代码清单 19-4 信号量删除函数 rt_sem_delete() 源码

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

    RT_ASSERT(sem != RT_NULL);                                    (1)

    /* 恢复所有阻塞在此信号量的线程 */
```

(下页继续)

(续上页)

```
    rt_ipc_list_resume_all(&(sem->parent.suspend_thread));          (2)

    /* 删除信号量对象 */
    rt_object_delete(&(sem->parent.parent));                        (3)

    return RT_EOK;
}
RTM_EXPORT(rt_sem_delete);
```

代码清单 19-4 **(1)**：检查信号量是否被创建了，如果是则可以进行删除操作。

代码清单 19-4 **(2)**：调用 rt_ipc_list_resume_all() 函数将所有因为访问此信号量的而阻塞的线程从阻塞态中恢复过来，线程得到信号量返回的错误代码，在实际情况一般不这样子使用，在删除的时候，应先确认所有的线程都无需再次访问此信号量，并且此时没有线程被此信号量阻塞，才进行删除操作。

代码清单 19-4 **(3)**：删除信号量对象并且释放信号量内核对象的内存，释放内核对象内存在 rt_object_delete() 函数中实现。

调用这个函数时，系统将删除这个信号量。如果删除该信号量时，有线程正在等待该信号量，那么删除操作会先唤醒等待在该信号量上的线程（等待线程的返回值是-RT_ERROR），信号量删除的实例代码，具体见加代码清单 19-5 粗部分。

列表 5: 代码清单 19-5 信号量删除函数 rt_sem_delete() 实例

```
/* 定义信号量控制块 */
static rt_sem_t test_sem = RT_NULL;

rt_err_t uwRet = RT_EOK;

uwRet = rt_sem_delete(test_sem);
if (RT_EOK == uwRet)
    rt_kprintf("信号量删除成功！\n\n");
```

22.6.3 信号量释放函数 rt_sem_release()

在前面的讲解中，我们知道，当信号量有效的时候，线程才能获取信号量，那么，是什么函数使得信号量变得有效？其实有两个方式，一个是在创建的时候进行初始化，将它可用的信号量个数设置一个初始值；在二进制信号量中，该初始值的范围是 0~1，假如初始值为 1 个可用的信号量的话，被申请一次就变得无效了，那就需要我们释放号量，RT-Thread 提供了信号量释放函数 rt_sem_release()，每调用一次该函数就释放一个信号量。但是有个问题，能不能一直释放呢？很显然，这是不能的，无论是你的信号量是用作二值信号量还是计数信号量，都要注意可用信号量的范围，当用作二值信号量的时候，必须确保其可用值在 0~1 范围内，所以使用二值信号量的时候要在使用完毕应及时释放信号量；而用作计数信号量的话，其范围是 0~65535，不允许超过释放 65535 个信号量，这代表我们不能一直调用 rt_sem_release() 函数来释放信号量。下面我们一起来看看信号量释放函数 rt_sem_release() 的源码，具体见*代码清单 19-6*。

列表 6: 代码清单 19-6 信号量释放函数 rt_sem_release() 源码

```
rt_err_t rt_sem_release(rt_sem_t sem)                                (1)
{
    register rt_base_t temp;
    register rt_bool_t need_schedule;

    RT_OBJECT_HOOK_CALL(rt_object_put_hook, (&(sem->parent.parent)));

    need_schedule = RT_FALSE;                                        (2)

    /* 关中断 */
    temp = rt_hw_interrupt_disable();

    RT_DEBUG_LOG(RT_DEBUG_IPC,("thread %s releases sem:%s, which value is: %d\n",
                                rt_thread_self()->name,
                                ((struct rt_object *)sem)->name,
```

(下页继续)

(续上页)

```
                                sem->value));

    if (!rt_list_isempty(&sem->parent.suspend_thread)) {
        /* 恢复阻塞线程 */
        rt_ipc_list_resume(&(sem->parent.suspend_thread));   (3)
        need_schedule = RT_TRUE;                             (4)
    } else
        sem->value ++; /* 记录可用信号量个数 */               (5)

    /* 开中断 */
    rt_hw_interrupt_enable(temp);

    /* 如果需要调度，则发起一次线程调度 */
    if (need_schedule == RT_TRUE)                            (6)
        rt_schedule();

    return RT_EOK;
}
RTM_EXPORT(rt_sem_release);
```

代码清单 19-6 **(1)**：根据信号量句柄（sem）释放信号量。

代码清单 19-6 **(2)**：定义一个记录是否需要进行系统调度的变量 need_schedule，默认为不需要调度。

代码清单 19-6 **(3)**：恢复阻塞线程。如果当前有线程等待这个信号量时，那么现在进行信号量释放的时候，将唤醒等待在该信号量线程队列中的第一个线程，由它获取信号量，并且将其从阻塞中恢复。恢复的过程是：将线程从阻塞列表中删除，添加到就绪列表中。

代码清单 19-6 **(4)**：恢复线程需要进行线程调度，所以此变量应该为真（RT_TRUE）。

代码清单 19-6 **(5)**：如果当前没有线程因为访问此信号量而进入阻塞，则不需要恢复线程，将该信号量的可用个数加一即可。此处应注意信号量的范围。

代码清单 19-6 **(6)**：如果需要进行调度，则调用 rt_schedule() 函数进行一次线程切换。

当线程完成资源的访问后，应尽快释放它持有的信号量，使得其他线程能获得该信号量，我们学习了信号量释放过程，那用起来简直就是异常爽快，循序渐进，学习了再去使用，这简直就是享受的过程，面一起来看看怎么使用信号量释放函数 rt_sem_release()，具体见*代码清单 19-7* 高亮部分。

补充：在中断中一样可以这样子调用信号量释放函数 rt_sem_release()，因为这个函数是非阻塞的。

列表 7: 代码清单 19-7 信号量释放函数 rt_sem_release()
实例

```
static void send_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
    /* 线程都是一个无限循环，不能返回 */
    while (1) { //如果 KEY2 被单击
        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_GPIO_PIN) == KEY_ON ) {
            /* 释放一个计数信号量 */
            uwRet = rt_sem_release(test_sem);
            if ( RT_EOK == uwRet )
                rt_kprintf ( "KEY2 被单击：释放 1 个停车位。\r\n" );
            else
                rt_kprintf ( "KEY2 被单击：但已无车位可以释放！\r\n" );
        }
        rt_thread_delay(20);      //每 20ms 扫描一次
    }
}
```

22.6.4 信号量获取函数 rt_sem_take()

与释放信号量对应的是获取信号量，我们知道，当信号量有效的时候，线程才能获取信号量，当线程获取了某个信号量的时候，该信号量的有效值就会减一，也就是说该信号量的可用个数就减一，当它减到 0 的时候，线程就无法再获取了，并且获取的线程会进入阻塞态（假如使用了等待时间的话）。在二进制信号量中，该初始值的范围是 0~1，假如初始值为 1 个可用的信号量的话，

被获取一次就变得无效了，那么此时另外一个线程获取该信号量的时候，就会无法获取成功，该线程便会进入阻塞态。每调用一次 rt_sem_take() 函数获取信号量的时候，信号量的可用个数便减少一个，直至为 0 的时候，线程就无法成功获取信号量了，具体见*代码清单 19-8*。

列表 8: 代码清单 19-8 信号量获取函数 rt_sem_take() 源码

```
rt_err_t rt_sem_take(rt_sem_t sem, rt_int32_t time)          (1)
{
    register rt_base_t temp;
    struct rt_thread *thread;

    RT_ASSERT(sem != RT_NULL);                               (2)

    RT_OBJECT_HOOK_CALL(rt_object_trytake_hook, (&(sem->parent.parent)));

    /* 关中断 */
    temp = rt_hw_interrupt_disable();

    RT_DEBUG_LOG(RT_DEBUG_IPC, ("thread %s take sem:%s, which value is: %d\n
↪",
                                rt_thread_self()->name,
                                ((struct rt_object *)sem)->name,
                                sem->value));

    if (sem->value > 0) {                                    (3)
        /* 有可用信号量 */
        sem->value --;

        /* 关中断 */
        rt_hw_interrupt_enable(temp);
    } else {
        /* 不等待，返回超时错误 */
```

(下页继续)

(续上页)

```
        if (time == 0) {                                        (4)
            rt_hw_interrupt_enable(temp);

            return -RT_ETIMEOUT;
        } else {
            /* 当前上下文检查 */
            RT_DEBUG_IN_THREAD_CONTEXT;

            /* 信号不可用，挂起当前线程 */
            /* 获取当前线程 */
            thread = rt_thread_self();                          (5)

            /* 设在线程错误代码 */
            thread->error = RT_EOK;

            RT_DEBUG_LOG(RT_DEBUG_IPC, ("sem take: suspend thread - %s\n",
                                        thread->name));

            /* 挂起线程 */
            rt_ipc_list_suspend(&(sem->parent.suspend_thread),(6)
                                thread,
                                sem->parent.parent.flag);

            /* 有等待时间，开始计时 */
            if (time > 0) {                                     (7)
                RT_DEBUG_LOG(RT_DEBUG_IPC, ("set thread:%s to timer list\n",
                                            thread->name));

                /* 设置线程超时时间，并且启动定时器 */
                rt_timer_control(&(thread->thread_timer),       (8)
                                 RT_TIMER_CTRL_SET_TIME,
                                 &time);
```

(下页继续)

(续上页)

```
                rt_timer_start(&(thread->thread_timer));     (9)
            }

            /* 开中断 */
            rt_hw_interrupt_enable(temp);

            /* 发起线程调度 */
            rt_schedule();                                   (10)

            if (thread->error != RT_EOK) {
                return thread->error;
            }
        }
    }

    RT_OBJECT_HOOK_CALL(rt_object_take_hook, (&(sem->parent.parent)));

    return RT_EOK;                                           (11)
}
RTM_EXPORT(rt_sem_take);
```

代码清单 *19-8* **(1)**：sem 信号量对象的句柄；time 指定的等待时间，单位是操作系统时钟节拍（tick）。

代码清单 *19-8* **(2)**：检查信号量是否有效，如果有效则进行获取操作。

代码清单 *19-8* **(3)**：如果当前有可用的信号量，那么线程获取信号量成功，信号量可用个数减一，然后直接跳到 **(11)** 返回成功。

代码清单 *19-8* **(4)**：_ **(4)~(10)** 都是表示当前没有可用信号量，此时无法获取到信号量，如果用户设定的等待时间为 0，那么线程获取信号量不成功，直接返回错误码-RT_ETIMEOUT。

代码清单 *19-8* **(5)**：如果用户设置了等待时间，那么在获取不到信号量的情况下，可以将获取信号量的线程挂起，进行等待，这首先获取到当前线程，调用 rt_thread_self() 函数就是为了得到当前线程控制块。

代码清单 *19-8* **(6)**：将线程挂起，rt_ipc_list_suspend() 此函数将线程挂起到指定列表。IPC 对象（rt_ipc_object）结构体中包含一个挂起列表，此处将当前线程挂起到信号量的挂起列表中。

代码清单 *19-8* **(7)**：如果有等待时间，那么需要计时，在时间到的时候恢复线程。

代码清单 *19-8* **(8)**：调用 rt_timer_control() 函数设置当前线程的挂起的时间，时间 time 由用户设定。

代码清单 *19-8* **(9)**：启动定时器开始计时。

代码清单 *19-8* **(10)**：发起一次线程调度，因为当前线程已经被挂起了吗，需要进行线程的切换。

线程通过获取信号量来获得信号量资源，当信号量值大于零时，线程将获得信号量，并且相应的信号量值都会减 1；如果信号量的值等于零，那么说明当前信号量资源不可用，获取该信号量的线程将根据 time 参数的情况选择直接返回、或挂起等待一段时间、或永久等待，直到其他线程或中断释放该信号量。如果在参数 time 指定的时间内依然得不到信号量，线程将超时返回，返回值是-RT_ETIMEOUT，其使用实例具体见代码清单 *19-9* 高亮部分。

列表 9: 代码清单 19-9 信号量获取函数 rt_sem_take() 实例

```
rt_sem_take(test_sem,                          /* 获取信号量 */
        RT_WAITING_FOREVER);          /* 等待时间: 一直等 */

uwRet = rt_sem_take(test_sem,                  /* 获取一个计数信号量 */
                    0);               /* 等待时间: 0 */
if ( RT_EOK == uwRet )
    rt_kprintf( "获取信号量成功\r\n" );
```

22.7 信号量实验

22.7.1 二值信号量同步实验

信号量同步实验是在 RT-Thread 中创建了两个线程，一个是获取信号量线程，一个是释放互斥量线程，两个线程独立运行，获取信号量线程是一直在等待信号量，其等待时间是 RT_WAITING_FOREVER，等到获取到信号量之后，线程处理完毕时它又马上释放信号量。

释放互斥量线程利用延时模拟占用信号量，延时的这段时间，获取线程无法获得信号量，等到释放线程使用完信号量，然后释放信号量，此时释放信号量会唤醒获取线程，获取线程开始运行，然后形成两个线程间的同步，若是线程正常同步，则在串口打印出信息，具体见*代码清单 19-10* 高亮部分。

列表 10: 代码清单 19-10 二值信号量同步实验

```
/**
*****************************************************************
* @file    main.c
* @author  fire
* @version V1.0
```

(下页继续)

(续上页)

```
* @date    2019-xx-xx
* @brief   RT-Thread  二值信号量
*****************************************************************
* @attention
*
* 实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
*******************************************************************
*/

/*
********************************************************************
*                              包含的头文件
********************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"
/* RT-Thread 头文件 */
#include "rtthread.h"
/*
****************************************************************
*                              变量
****************************************************************
*/
/* 定义线程控制块 */
static rt_thread_t receive_thread = RT_NULL;
static rt_thread_t send_thread = RT_NULL;
/* 定义信号量控制块 */
static rt_sem_t test_sem = RT_NULL;

```

(下页继续)

(续上页)

```
/************************* 全局变量声明 ****************************/
/*
* 当我们在写应用程序的时候，可能需要用到一些全局变量。
*/
uint8_t ucValue [ 2 ] = { 0x00, 0x00 };
/*
**********************************************************************
*                              函数声明
**********************************************************************
*/
static void receive_thread_entry(void* parameter);
static void send_thread_entry(void* parameter);

/*
**********************************************************************
*                              main 函数
**********************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]- i.MX RT 全系列开发板 RTT 二值信号量同步实验！\n");
    rt_kprintf("同步成功则输出 Successful，反之输出 Fail\n");
```

(下页继续)

(续上页)

```
    /* 创建一个信号量 */
    test_sem = rt_sem_create("test_sem",/* 信号量名字 */
                             1,      /* 信号量初始值，默认有一个信号量 */
                             RT_IPC_FLAG_FIFO); /* 信号量模式 FIFO(0x00)*/
    if (test_sem != RT_NULL)
        rt_kprintf("信号量创建成功! \n\n");

    receive_thread =                            /* 线程控制块指针 */
        rt_thread_create( "receive",              /* 线程名字 */
                        receive_thread_entry, /* 线程入口函数 */
                        RT_NULL,              /* 线程入口函数参数 */
                        512,                  /* 线程栈大小 */
                        3,                    /* 线程的优先级 */
                        20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (receive_thread != RT_NULL)
        rt_thread_startup(receive_thread);
    else
        return -1;

    send_thread =                           /* 线程控制块指针 */
        rt_thread_create( "send",               /* 线程名字 */
                        send_thread_entry,    /* 线程入口函数 */
                        RT_NULL,              /* 线程入口函数参数 */
                        512,                  /* 线程栈大小 */
                        2,                    /* 线程的优先级 */
                        20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (send_thread != RT_NULL)
        rt_thread_startup(send_thread);
```

(下页继续)

(续上页)

```
    else
        return -1;
}

/*
*****************************************************
*                              线程定义
*****************************************************
*/

static void receive_thread_entry(void* parameter)
{
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        rt_sem_take(test_sem,                /* 获取信号量 */
                    RT_WAITING_FOREVER); /* 等待时间：一直等 */
        if ( ucValue [ 0 ] == ucValue [ 1 ] ) {
            rt_kprintf ( "Successful\n" );
        } else {
            rt_kprintf ( "Fail\n" );
        }
        rt_sem_release(     test_sem        );    //释放二值信号量

        rt_thread_delay ( 100 );             //每 1s 读一次
    }
}



static void send_thread_entry(void* parameter)
{
    /* 线程都是一个无限循环，不能返回 */
```

(下页继续)

(续上页)

```
    while (1) {
        rt_sem_take(test_sem,        /* 获取信号量 */
                    RT_WAITING_FOREVER); /* 等待时间: 一直等 */
        ucValue [ 0 ] ++;
        rt_thread_delay ( 10 );         /* 延时 100ms */
        ucValue [ 1 ] ++;
        rt_sem_release(test_sem);     //释放二值信号量
        rt_thread_yield();                  //放弃剩余时间片，进行一次线程切换
    }
}
/***********************END OF FILE***************************/
```

22.7.2 计数信号量实验

计数型信号量实验是模拟停车场工作运行。在创建信号量的时候初始化 5 个可用的信号量，并且创建了两个线程：一个是获取信号量线程，一个是释放信号量线程，两个线程独立运行，获取信号量线程是通过按下 K1 按键进行信号量的获取，模拟停车场停车操作，其等待时间是 0，在串口调试助手输出相应信息。

释放信号量线程则是信号量的释放，释放信号量线程也是通过按下 K2 按键进行信号量的释放，模拟停车场取车操作，在串口调试助手输出相应信息，实验源码具体见*代码清单 19-11* 高亮部分。

列表 11: 代码清单 19-11 计数信号量实验

```
/**
*******************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2019-xx-xx
* @brief   RT-Thread 计数信号量
```

(下页继续)

(续上页)

```
**********************************************************************
* @attention
*
* 实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
**********************************************************************
*/

/*
**************************************************************************
*                                包含的头文件
**************************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"
/* RT-Thread 头文件 */
#include "rtthread.h"
/*
*********************************************************************
*                                变量
*********************************************************************
*/
/* 定义线程控制块 */
static rt_thread_t receive_thread = RT_NULL;
static rt_thread_t send_thread = RT_NULL;
/* 定义消息队列控制块 */
static rt_sem_t test_sem = RT_NULL;

/************************* 全局变量声明 ****************************/
/*
```

(下页继续)

(续上页)

```
* 当我们在写应用程序的时候，可能需要用到一些全局变量。
*/
/*
*************************************************************************
*                              函数声明
*************************************************************************
*/
static void receive_thread_entry(void* parameter);
static void send_thread_entry(void* parameter);

/*
*************************************************************************
*                              main 函数
*************************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]- i.MX RT-霸道-RTT 计数信号量实验！\n");
    rt_kprintf("车位默认值为 5 个，按下 K1 申请车位，按下 K2 释放车位！\n\n");
    /* 创建一个信号量 */
    test_sem = rt_sem_create("test_sem",/* 计数信号量名字 */
                             5,     /* 信号量初始值，默认有 5 个信号量 */
```

(下页继续)

(续上页)

```
                                RT_IPC_FLAG_FIFO); /* 信号量模式 FIFO(0x00)*/
    if (test_sem != RT_NULL)
        rt_kprintf("计数信号量创建成功! \n\n");

    receive_thread =                            /* 线程控制块指针 */
        rt_thread_create( "receive",                /* 线程名字 */
                        receive_thread_entry,    /* 线程入口函数 */
                        RT_NULL,                /* 线程入口函数参数 */
                        512,                    /* 线程栈大小 */
                        3,                      /* 线程的优先级 */
                        20);                    /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (receive_thread != RT_NULL)
        rt_thread_startup(receive_thread);
    else
        return -1;

    send_thread =                           /* 线程控制块指针 */
        rt_thread_create( "send",               /* 线程名字 */
                        send_thread_entry,    /* 线程入口函数 */
                        RT_NULL,                /* 线程入口函数参数 */
                        512,                    /* 线程栈大小 */
                        2,                      /* 线程的优先级 */
                        20);                    /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (send_thread != RT_NULL)
        rt_thread_startup(send_thread);
    else
        return -1;
}
```

(下页继续)

(续上页)

```
/*
*****************************************************************
*                                线程定义
******************************************************************
*/

static void receive_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
    /* 线程都是一个无限循环，不能返回 */
    while (1) {//如果 KEY2 被单击
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_GPIO_PIN) == KEY_ON ) {
            /* 获取一个计数信号量 */
            uwRet = rt_sem_take(test_sem,
                                0);          /* 等待时间: 0 */
            if ( RT_EOK == uwRet )
                rt_kprintf( "KEY1 被单击: 成功申请到停车位。\r\n" );
            else
                rt_kprintf( "KEY1 被单击: 不好意思，现在停车场已满！\r\n" );
        }
        rt_thread_delay(2);      //每 20ms 扫描一次
    }
}

static void send_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
    /* 线程都是一个无限循环，不能返回 */
    while (1) {//如果 KEY2 被单击
        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_GPIO_PIN) == KEY_ON ) {
            /* 释放一个计数信号量 */
```

(下页继续)

(续上页)

```
            uwRet = rt_sem_release(test_sem);
            if ( RT_EOK == uwRet )
                rt_kprintf ( "KEY2 被单击：释放 1 个停车位。\r\n" );
            else
                rt_kprintf ( "KEY2 被单击：但已无车位可以释放！\r\n" );
        }
        rt_thread_delay(2);      //每 20ms 扫描一次
    }
}
/****************************END OF FILE****************************/
```

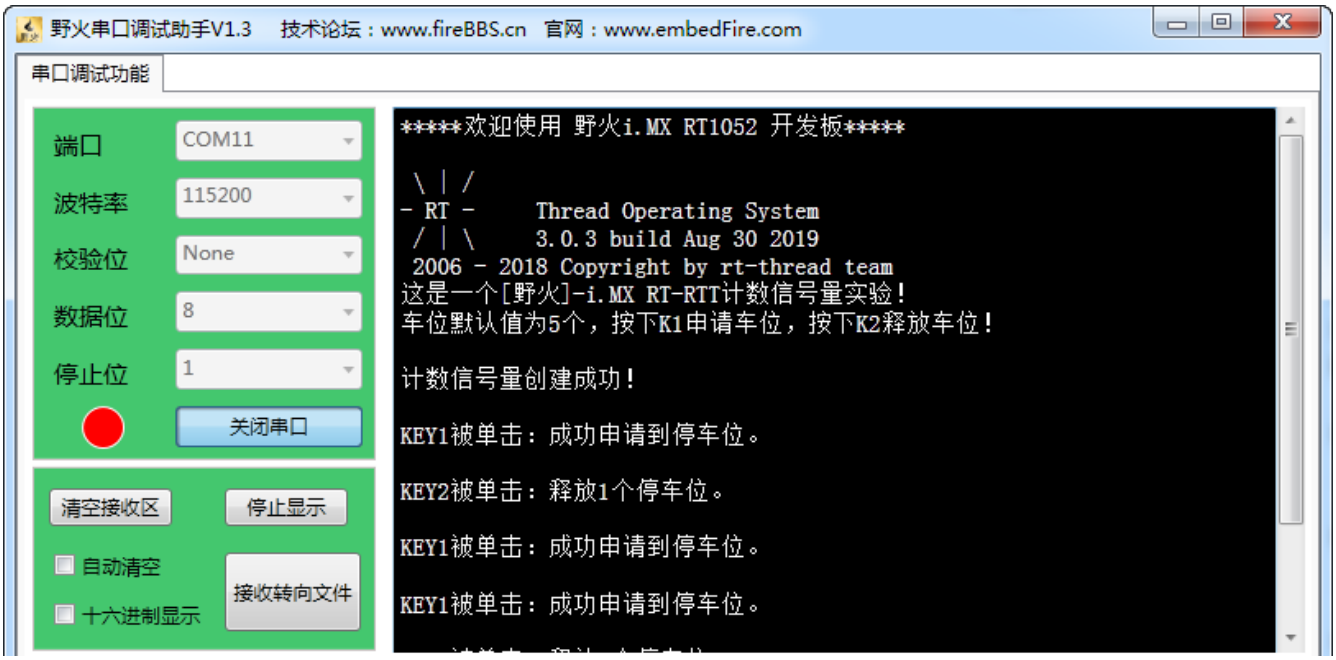
22.8 实验现象

22.8.1 二值信号量同步实验现象

将程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息，它里面输出了信息表明线程正在运行中，当输出信息为 Successful 的时候，则表面两个线程同步成功，具体见图 *19-5*。

22.8.2 计数信号量实验现象

将程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息，按下开发版的 K1 按键获取信号量，按下 K2 按键释放信号量；我们按下 K1 与 K2 试试，在串口调试助手中可以看到运行结果，具体见图 *19-6*。


野火串口调试助手V1.3 技术论坛：www.fireBBS.cn 官网：www.embedFire.com
串口调试功能
端口
COM11
波特率
115200
校验位
None
数据位
8
停止位
1
关闭串口
清空接收区
停止显示
自动清空
十六进制显示
接收转向文件
*****欢迎使用 野火i.MX RT1052 开发板*****
\ | /
- RT - Thread Operating System
/ | \ 3.0.3 build Aug 30 2019
2006 - 2018 Copyright by rt-thread team
这是一个[野火]-i.MX RT-RTT计数信号量实验！
车位默认值为5个，按下K1申请车位，按下K2释放车位！
计数信号量创建成功！
KEY1被单击：成功申请到停车位。
KEY2被单击：释放1个停车位。
KEY1被单击：成功申请到停车位。
KEY1被单击：成功申请到停车位。

第 23 章　互斥量

23.1 互斥量的基本概念

互斥量又称互斥型信号量，是一种特殊的二值信号量，它和信号量不同的是，它支持互斥量所有权、递归访问以及防止优先级翻转的特性，用于实现对临界资源的独占式处理。任意时刻互斥量的状态只有两种，开锁或闭锁。当互斥量被线程持有时，该互斥量处于闭锁状态，这个线程获得互斥量的所有权。当该线程释放这个互斥量时，该互斥量处于开锁状态，线程失去该互斥量的所有权。当一个线程持有互斥量时，其他线程将不能再对该互斥量进行开锁或持有。持有该互斥量的线程也能够再次获得这个锁而不被挂起，这就是递归访问，这个特性与一般的二值信号量有很大的不同，在信号量中，由于已经不存在可用的信号量，线程递归获取信号量时会发生主动挂起（最终形成死锁）。

如果想要用于实现同步（线程之间或者线程与中断之间），二值信号量或许是更好的选择，虽然互斥量也可以用于线程与线程、线程与中断的同步，但是互斥量更多的是用于保护资源的互锁。

用于互锁的互斥量可以充当保护资源的令牌。当一个线程希望访问某个资源时，它必须先获取令牌。当线程使用完资源后，必须还回令牌，以便其它线程可以访问该资源。是不是很熟悉，在我们的二值信号量里面也是一样的，用于保护临界资源，保证多线程的访问井然有序。当线程获取到信号量的时候才能开始使用被保护的资源，使用完就释放信号量，下一个线程才能获取到信号量从而可用使用被保护的资源。但是信号量会导致的另一个潜在问题，那就是线程优先级翻转（具体会在下文讲解）。而 RT-Thread 提供的互斥量通过优先级继承算法，可用降低优先级翻转问题产生的影响，所以，用于临界资源的保护一般建议使用互斥量。

23.2 互斥量的优先级继承机制

在 RT-Thread 操作系统中为了降低优先级翻转问题利用了优先级继承算法。优先级继承算法是指，暂时提高某个占有某种资源的低优先级线程的优先级，使之与在所有等待该资源的线程中优先级最高那个线程的优先级相等，而当这个低优先级线程执行完毕释放该资源时，优先级重新回到初始设定值。因此，继承优先级的线程避免了系统资源被任何中间优先级的线程抢占。

互斥量与二值信号量最大的不同是：互斥量具有优先级继承机制，而信号量没有。也就是说，某个临界资源受到一个互斥量保护，如果这个资源正在被一个低优先级线程使用，那么此时的互斥量是闭锁状态，也代表了没有线程能申请到这个互斥量，如果此时一个高优先级线程想要对这个资源进行访问，去申请这个互斥量，那么高优先级线程会因为申请不到互斥量而进入阻塞态，那么系统会将现在持有该互斥量的线程的优先级临时提升到与高优先级线程的优先级相同，这个优先级提升的过程叫做优先级继承。这个优先级继承机制确保高优先级线程进入阻塞状态的时间尽可能短，以及将已经出现的“优先级翻转”危害降低到最小。

没有理解？没问题，结合过程示意图再说一遍。我们知道线程的优先级在创建的时候就已经是设置好的，高优先级的线程可以打断低优先级的线程，抢占 CPU 的使用权。但是在很多场合中，某些资源只有一个，当低优先级线程正在占用该资源的时候，即便高优先级线程也只能乖乖的等待低优先级线程使用完该资源后释放资源。这里高优先级线程无法运行而低优先级线程可以运行的现象称为“优先级翻转”。

为什么说优先级翻转在操作系统中是危害很大？因为在我们一开始创造这个系统的时候，我们就已经设置好了线程的优先级了，越重要的线程优先级越高。但是发生优先级翻转，对我们操作系统是致命的危害，会导致系统的高优先级线程阻塞时间过长。

举个例子，现在有 3 个线程分别为 H 线程（High）、M 线程（Middle）、L 线程（Low），3 个线程的优先级顺序为 H 线程 >M 线程 >L 线程。正常运行的时候 H 线程可以打断 M 线程与 L 线程，M 线程可以打断 L 线程，假设系统中有一个资源被保护了，此时该资源被 L 线程正在使用中，某一刻，H 线程需要使用该资源，但是 L 线程还没使用完，H 线程则因为申请不到资源而进入阻塞态，L 线程继续使用该资源，此时已经出现了“优先级翻转”现象，高优先级线程在等着低优先级的线程执行，如果在 L 线程执行的时候刚好 M 线程被唤醒了，由于 M 线程优先级比 L 线程优先级高，那么会打断 L 线程，抢占了 CPU 的使用权，直到 M 线程执行完，再把 CUP 使用权归还给 L 线程，L 线程继续执行，等到执行完毕之后释放该资源，H 线程此时才从阻塞态解除，使用

该资源。这个过程，本来是最高优先级的 H 线程，在等待了更低优先级的 L 线程与 M 线程，其阻塞的时间是 M 线程运行时间 +L 线程运行时间，这只是只有 3 个线程的系统，假如很多个这样子的线程打断最低优先级的线程，那这个系统最高优先级线程岂不是崩溃了，这个现象是绝对不允许出现的，高优先级的线程必须能及时响应。所以，没有优先级继承的情况下，使用资源保护，其危害极大，具体见图 *20-1*。

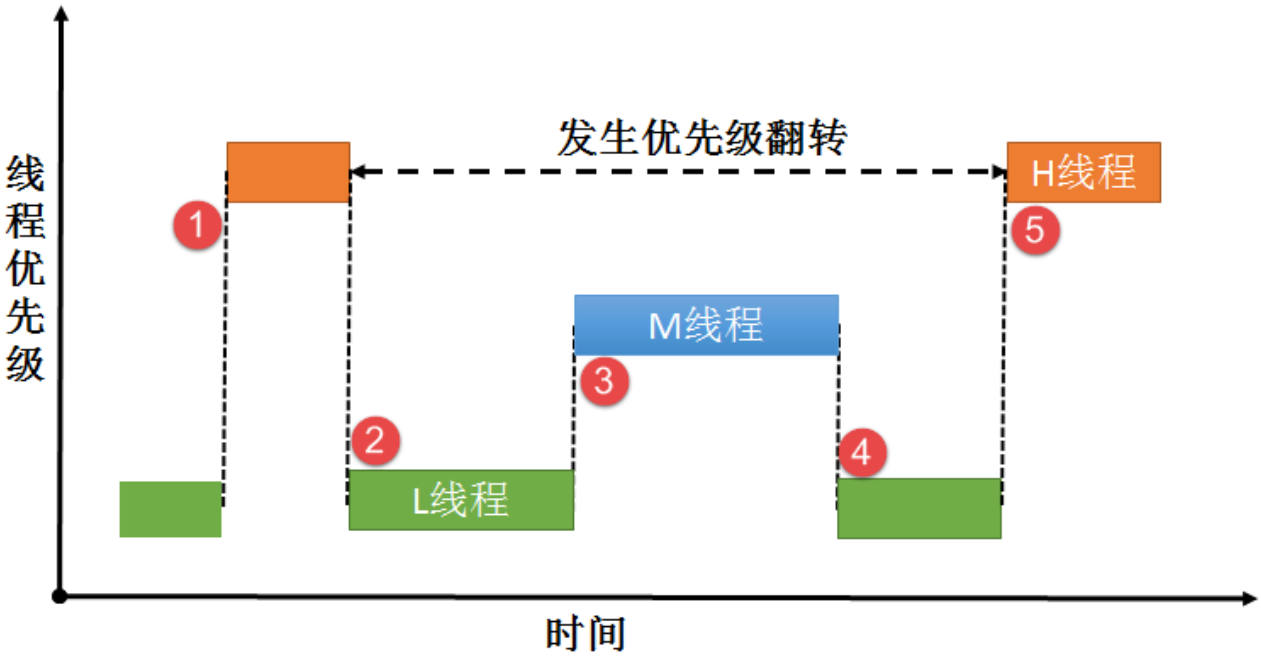


图 *20-1* **(1)**：L 线程正在使用某临界资源，H 线程被唤醒，执行 H 线程。但 L 线程并未执行完毕，此时临界资源还未释放。

图 *20-1* **(2)**：这个时刻 H 线程也要对该临界资源进行访问，但 L 线程还未释放资源，由于保护机制，H 线程进入阻塞态，L 线程得以继续运行，此时已经发生了优先级翻转现象。

图 *20-1* **(3)**：某个时刻 M 线程被唤醒，由于 M 线程的优先级高于 L 线程，M 线程抢占了 CPU 的使用权，M 线程开始运行，此时 L 线程尚未执行完，临界资源还没被释放。

图 *20-1* **(4)**：M 线程运行结束，归还 CPU 使用权，L 线程继续运行。

图 *20-1* **(5)**：L 线程运行结束，释放临界资源，H 线程得以对资源进行访问，H 线程开始运行。

在这过程中，H 线程的等待时间过长，这对系统来说这是很致命的，所以这种情况不允许出现，而互斥量就是用来降低优先级翻转的产生的危害。

假如有优先级继承呢？那么，在 H 线程申请该资源的时候，由于申请不到资源会进入阻塞态，那么系统就会把当前正在使用资源的 L 线程的优先级临时提高到与 H 线程优先级相同，此时 M 线程被唤醒了，因为它的优先级比 H 线程低，所以无法打断 L 线程，因为此时 L 线程的优先级被临时提升到 H，所以当 L 线程使用完该资源了，进行释放，那么此时 H 线程优先级最高，将接着抢占 CPU 的使用权，H 线程的阻塞时间仅仅是 L 线程的执行时间，此时的优先级的危害降到了最低，看！这就是优先级继承的优势，具体见图 *20-2*。

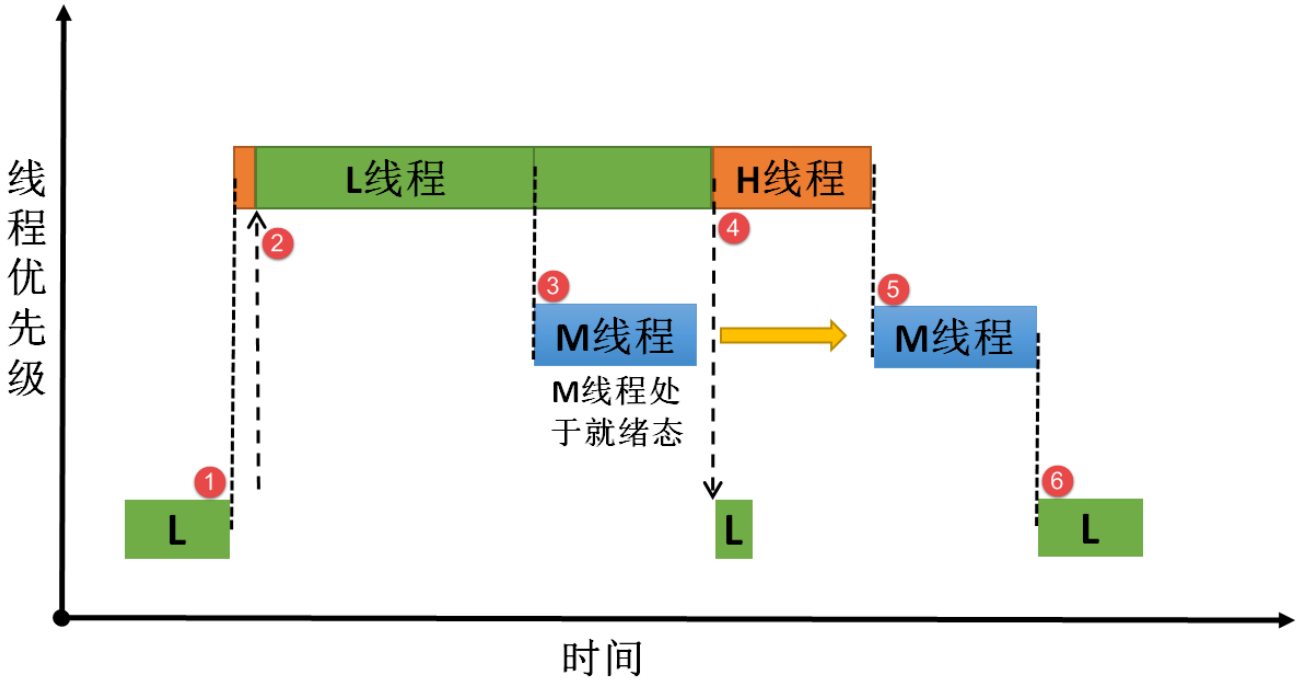


图 *20-2* **(1)**：L 线程正在使用某临界资源，L 线程正在使用某临界资源，H 线程被唤醒，执行 H 线程。但 L 线程并未执行完毕，此时临界资源还未释放。

图 *20-2* **(2)**：某一时刻 H 线程也要对该资源进行访问，由于保护机制，H 线程进入阻塞态。此时发生优先级继承，系统将 L 线程的优先级暂时提升到与 H 线程优先级相同，L 线程继续执行。

图 *20-2* **(3)**：在某一时刻 M 线程被唤醒，由于此时 M 线程的优先级暂时低于 L 线程，所以 M 线程仅在就绪态，而无法获得 CPU 使用权。

图 *20-2* **(4)**：L 线程运行完毕，H 线程获得对资源的访问权，H 线程从阻塞态变成运行态，此时 L 线程的优先级会变回原来的优先级。

图 *20-2* **(5)**：当 H 线程运行完毕，M 线程得到 CPU 使用权，开始执行。

图 *20-2* **(6)**：系统正常运行，按照设定好的优先级运行。

但是使用互斥量的时候一定需要注意：在获得互斥量后，请尽快释放互斥量，同时需要注意的是在线程持有互斥量的这段时间，不得更改线程的优先级。

23.3 互斥量的应用场景

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

- 线程可能会多次获取互斥量的情况下。这样可以避免同一线程多次递归持有而造成死锁的问题；
- 可能会引起优先级翻转的情况；

多线程环境下往往存在多个线程竞争同一临界资源的应用场景，互斥量可被用于对临界资源的保护从而实现独占式访问。另外，互斥量可以降低信号量存在的优先级翻转问题带来的影响。

比如有两个线程需要对串口进行发送数据，其硬件资源只有一个，那么两个线程肯定不能同时发送啦，不然导致数据错误，那么，就可以用互斥量对串口资源进行保护，当一个线程正在使用串口的时候，另一个线程则无法使用串口，等到线程使用串口完毕之后，另外一个线程才能获得串口的使用权。

另外需要注意的是互斥量不能在中断服务函数中使用。

23.4 互斥量的运作机制

多线程环境下会存在多个线程访问同一临界资源的场景，该资源会被线程独占处理。其他线程在资源被占用的情况下不允许对该临界资源进行访问，这个时候就需要用到 RT-Thread 的互斥量来进行资源保护，那么互斥量是怎样来避免这种冲突？

用互斥量处理不同线程对临界资源的同步访问时，线程想要获得互斥量才能进行资源访问，如果一旦有线程成功获得了互斥量，则互斥量立即变为闭锁状态，此时其他线程会因为获取不到互斥量而不能访问这个资源，线程会根据用户自定义的等待时间进行等待，直到互斥量被持有的线程

释放后，其他线程才能获取互斥量从而得以访问该临界资源，此时互斥量再次上锁，如此一来就可以确保每个时刻只有一个线程正在访问这个临界资源，保证了临界资源操作的安全性。

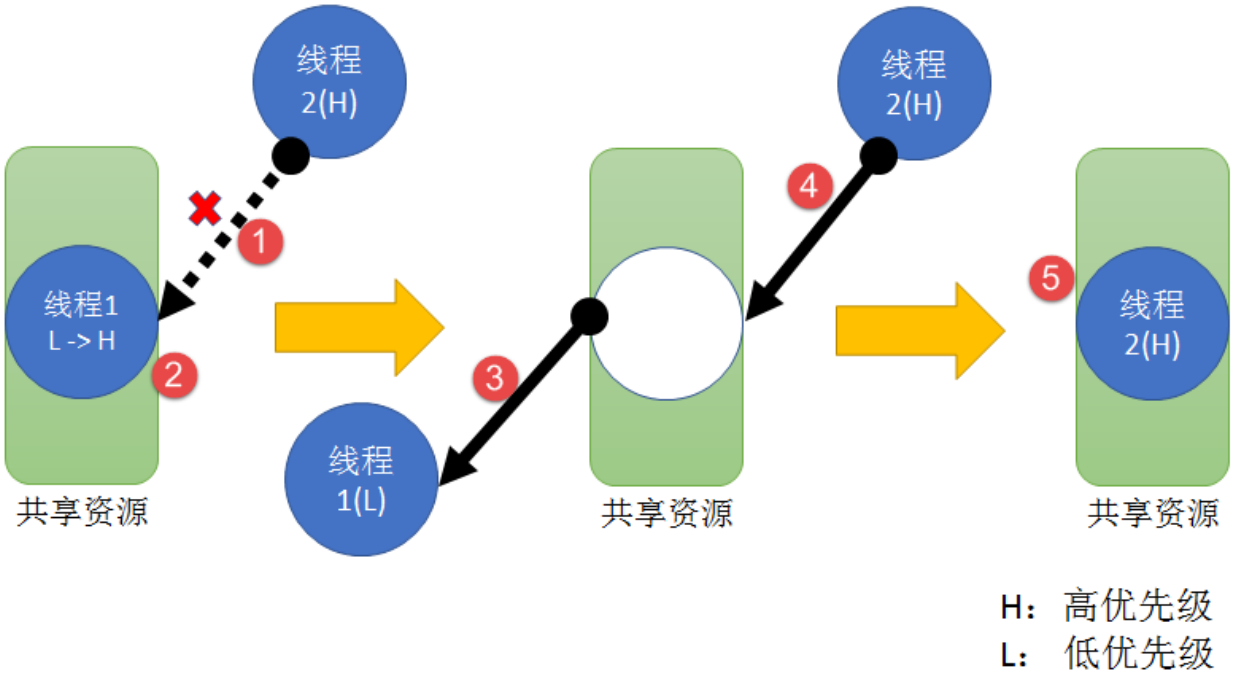


图 *20-3* **(1)**：因为互斥量具有优先级继承机制，一般选择使用互斥量对资源进行保护，如果资源被占用的时候，无论是什么优先级的线程想要使用该资源都会被阻塞。

图 *20-3* **(2)**：假如正在使用该资源的线程 1 比阻塞中的线程 2 的优先级还低，那么线程 1 将被系统临时提升到与高优先级线程 2 相等的优先级（线程 1 的优先级从 L 变成 H）。

图 *20-3* **(3)**：当线程 1 使用完资源之后，释放互斥量，此时线程 1 的优先级会从 H 变回原来的 L。

图 *20-3* **(4)-(5)**：线程 2 此时可以获得互斥量，然后进行资源的访问，当线程 2 访问了资源的时候，该互斥量的状态又为闭锁状态，其他线程无法获取互斥量。

23.5 互斥量控制块

上面说的互斥量是不是很有用？在我们操作系统中，如果不会使用的话会就会遇到很多麻烦的问题，如果使用得好的话，这种上操作系统的开发简直就跟喝水一样简单，但是喝水也能呛死人的，凡事还是要小心谨慎，特别是使用操作系统的时候。说到互斥量的使用就不得不说一下互斥量的控制块了，互斥量控制块与线程控制类似，每一个互斥量都有自己的互斥量控制块，互斥量控制块中包含了互斥量的所有信息，比如互斥量的一些状态信息，使用情况等，具体见*代码清单 20-1*。

列表 1: 代码清单 20-1 互斥量控制块

```
struct rt_mutex {
struct rt_ipc_object parent;                    (1)

rt_uint16_t          value;                     (2)

rt_uint8_t           original_priority;         (3)
rt_uint8_t           hold;            (4)

struct rt_thread    *owner;                     (5)
};
typedef struct rt_mutex *rt_mutex_t;
```

代码清单 20-1 **(1)**：互斥量属于内核对象，也会在自身结构体里面包含一个内核对象类型的成员，通过这个成员可以将互斥量挂到系统对象容器里面。互斥量从 rt_ipc_object 中派生，由 IPC 容器管理。

代码清单 20-1 **(2)**：互斥量的值。初始状态下互斥量的值为 1，因此，如果值大于 0，表示可以使用互斥量。

代码清单 20-1 **(3)**：持有互斥量线程的原始优先级，用来做优先级继承的保存。

代码清单 20-1 **(4)**：持有互斥量的线程的持有次数，用于记录线程递归调用了多少次获取互斥量。

代码清单 20-1 **(5)**：当前持有互斥量的线程。

23.6 信号量函数接口讲解

23.6.1 互斥量创建函数 rt_mutex_create()

互斥量的创建很简单，因为互斥量是用于保护临界资源的，它就只有两种状态，要么是闭锁，要么是开锁，创建的时候需要我们自己定义互斥量句柄，以后对互斥量的操作也是通过这互斥量句柄进行操作的，RT-Thread 官方给我们提供了一个创建函数 rt_mutex_create()，

我们可以调用 rt_mutex_create 函数创建一个互斥量，它的名字由 name 所指定。创建成功返回指向互斥量的互斥量句柄，否则返回 RT_NULL，具体见*代码清单 20-2*。

列表 2: 代码清单 20-2 互斥量创建函数 rt_mutex_create()源码

```
rt_mutex_t rt_mutex_create(const char *name, rt_uint8_t flag)
{
struct rt_mutex *mutex;

RT_DEBUG_NOT_IN_INTERRUPT;

/* 分配对象 */
mutex = (rt_mutex_t)rt_object_allocate(RT_Object_Class_Mutex, name);
if (mutex == RT_NULL)                                          (1)
    return mutex;

/* 初始化 IPC 对象 */
rt_ipc_object_init(&(mutex->parent));                          (2)

mutex->value              = 1;                                 (3)
mutex->owner              = RT_NULL;                           (4)
mutex->original_priority  = 0xFF;                              (5)
mutex->hold               = 0;                                 (6)

```

(下页继续)

(续上页)

```
/* 设置互斥量的等待模式 */
mutex->parent.parent.flag = flag;                                    (7)

return mutex;                                                        (8)
}
RTM_EXPORT(rt_mutex_create);
```

代码清单 20-2 **(1)**：分配互斥量对象，调用 rt_object_allocate 此函数将从对象系统分配对象，为创建的互斥量分配一个互斥量的对象，并且命名对象名称，在系统中，对象的名称必须是唯一的。

代码清单 20-2 **(2)**：初始化互斥量内核对象。调用 rt_ipc_object_init 会初始化一个链表用于记录访问此互斥量而阻塞的线程。

代码清单 20-2 **(3)**：将互斥量的值初始化为 1，

代码清单 20-2 **(4)**：初始化持有互斥量线程为 RT_NULL，因为现在是创建互斥量，肯定没有线程持有该互斥量。

代码清单 20-2 **(5)**：持有互斥量线程的原始优先级默认为 0xFF，在获取的时候这个值就会被重置为获取线程的优先级。

代码清单 20-2 **(6)**：持有互斥量的线程的持有次数为 0。

代码清单 20-2 **(7)**：设置互斥量的阻塞唤醒模式，创建的互斥量由于指定的 flag 不同，而有不同的意义：使用 RT_IPC_FLAG_PRIO 优先级 flag 创建的 IPC 对象，在多个线程等待资源时，将由优先级高的线程优先获得资源。而使用 RT_IPC_FLAG_FIFO 先进先出 flag 创建的 IPC 对象，在多个线程等待资源时，将按照先来先得的顺序获得资源。RT_IPC_FLAG_PRIO 与 RT_IPC_FLAG_FIFO 均在 rtdef.h 中有定义。

代码清单 20-2 **(8)**：返回互斥量句柄。

理解了创建互斥量的源码，用起来就会很简单了，在创建互斥的时候需要我们自己定义互斥量的句柄，也就是互斥量控制块指针，具体见代码清单 20-3 加粗部分。

列表 3: 代码清单 20-3 互斥量创建函数 rt_mutex_create() 实例

```
/* 定义互斥量控制块 */
static rt_mutex_t test_mux = RT_NULL;
/* 创建一个互斥量 */
test_mux = rt_mutex_create("test_mux",RT_IPC_FLAG_PRIO);

if (test_mux != RT_NULL)
    rt_kprintf("互斥量创建成功! \n\n");
```

23.6.2 互斥量删除函数 rt_mutex_delete()

互斥量删除函数是根据互斥量句柄（mutex）直接删除的，删除之后这个互斥量的所有信息都会被系统回收清空，而且不能再次使用这个互斥量。但是需要注意的是，如果互斥量没有被创建，那是无法被删除的，动脑子想想都知道，没创建的东西就不存在，怎么可能被删除。删除互斥量的时候会把所有阻塞在互斥量的线程唤醒，被唤醒的线程则会得到一个错误码-RT_ERROR；mutex 是 rt_sem_delete() 传入的参数，是互斥量句柄，表示的是要删除哪个互斥量，其函数源码见*代码清单 20-4*。

列表 4: 代码清单 20-4 互斥量删除函数 rt_mutex_delete() 源码

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

    RT_ASSERT(mutex != RT_NULL);                                    (1)

    /* 解除所有挂起线程 */
    rt_ipc_list_resume_all(&(mutex->parent.suspend_thread));        (2)
```

(下页继续)

(续上页)

```
    /* 删除互斥量对象 */
    rt_object_delete(&(mutex->parent.parent));                    (3)

    return RT_EOK;
}
RTM_EXPORT(rt_mutex_delete);
```

代码清单 20-4 **(1)**：检查互斥量是否被创建了，如果是则可以进行删除操作。

代码清单 20-4 **(2)**：调用 rt_ipc_list_resume_all() 函数将所有因为访问此互斥量的而阻塞的线程从阻塞态中恢复过来，线程将得到互斥量返回的错误代码，在实际情况一般不这样子使用，在删除的时候，应先确认所有的线程都无需再次访问此互斥量，并且此时没有线程被此互斥量阻塞，才进行删除操作。

代码清单 20-4 **(3)**：删除互斥量对象并且释放互斥量内核对象的内存，释放内核对象内存在 rt_object_delete() 函数中实现。

当删除一个互斥量时，所有等待此互斥量的线程都将被唤醒，等待线程获得的返回值是-RT_ERROR。然后系统将该互斥量从内核对象管理器链表中删除并释放互斥量占用的内存空间，互斥量删除函数 rt_mutex_delete() 使用实例具体见代码清单 20-5 加粗部分。

列表 5: 代码清单 20-5 互斥量删除函数 rt_mutex_delete() 实例

```
/* 定义消息队列控制块 */
static rt_mutex_t test_mutex = RT_NULL;

rt_err_t uwRet = RT_EOK;

uwRet = rt_mutex_delete(test_mutex);
if (RT_EOK == uwRet)
    rt_kprintf("互斥量删除成功！\n\n");
```

23.6.3 互斥量释放函数 rt_mutex_release()

线程想要访问某个资源的时候，需要先获取互斥量，然后进行资源访问，在线程使用完该资源的时候，必须要及时归还互斥量，这样别的线程才能对资源进行访问。在前面的讲解中，我们知道，当互斥量有效的时候，线程才能获取互斥量，那么，是什么函数使得信号量变得有效呢？RT-Thread 给我们提供了互斥量释放函数 rt_mutex_release()，线程可以调用 rt_mutex_release() 函数进行释放互斥量，表示我已经用完了，别人可以申请使用。

使用该函数接口时，只有已持有互斥量所有权的线程才能释放它，每释放一次该互斥量，它的持有计数就减 1。当该互斥量的持有计数为零时（即持有线程已经释放所有的持有操作），互斥量则变为开锁状态，等待在该互斥量上的线程将被唤醒。如果线程的优先级被互斥量的优先级翻转机制临时提升，那么当互斥量被释放后，线程的优先级将恢复为原本设定的优先级，具体见*代码清单 20-6*。

列表 6: 代码清单 20-6 互斥量释放函数 rt_mutex_release()
源码

```
rt_err_t rt_mutex_release(rt_mutex_t mutex)                          (1)
{
    register rt_base_t temp;
    struct rt_thread *thread;
    rt_bool_t need_schedule;

    need_schedule = RT_FALSE;

    /* 只有持有的线程可以释放互斥量，因为需要测试互斥量的所有权 */
    RT_DEBUG_IN_THREAD_CONTEXT;

    /* 获取当前线程 */
    thread = rt_thread_self();                                       (2)

    /* 关中断 */
    temp = rt_hw_interrupt_disable();
```

(下页继续)

(续上页)

```

    RT_DEBUG_LOG(RT_DEBUG_IPC,
        ("mutex_release:current thread %s, mutex value: %d, hold: %d\n",
                thread->name, mutex->value, mutex->hold));

    RT_OBJECT_HOOK_CALL(rt_object_put_hook, (&(mutex->parent.parent)));

    /* 互斥量只能被持有者释放 */
    if (thread != mutex->owner) {                                   (3)
        thread->error = -RT_ERROR;

        /* 开中断 */
        rt_hw_interrupt_enable(temp);

        return -RT_ERROR;
    }

    /* 减少持有量 */
    mutex->hold --;                                                 (4)
    /* 如果没有持有量了 */
    if (mutex->hold == 0) {                                         (5)
        /* 将持有者线程更改为原始优先级 */
        if (mutex->original_priority != mutex->owner->current_priority) {
            rt_thread_control(mutex->owner,
                              RT_THREAD_CTRL_CHANGE_PRIORITY,
                              &(mutex->original_priority));         (6)
        }

        /* 唤醒阻塞线程 */
        if (!rt_list_isempty(&mutex->parent.suspend_thread)) {      (7)
            /* 获取阻塞线程 */
            thread = rt_list_entry(mutex->parent.suspend_thread.next,
```

(下页继续)

(续上页)

```
                                struct rt_thread,
                                tlist);                                 (8)

        RT_DEBUG_LOG(RT_DEBUG_IPC, ("mutex_release: resume thread: %s\n",
                                    thread->name));

            /* 设置新的持有者线程与其优先级 */
            mutex->owner             = thread;                          (9)
            mutex->original_priority = thread->current_priority;        (10)
            mutex->hold ++;                                             (11)

            /* 恢复线程 */
            rt_ipc_list_resume(&(mutex->parent.suspend_thread));        (12)

            need_schedule = RT_TRUE;
        } else {
            /* 记录增加 value 的值 */
            mutex->value ++;                                            (13)

            /* 清除互斥量信息 */
            mutex->owner             = RT_NULL;                         (14)
            mutex->original_priority = 0xff;                            (15)
        }
    }

    /* 开中断 */
    rt_hw_interrupt_enable(temp);

    /* 执行一次线程调度 */
    if (need_schedule == RT_TRUE)
        rt_schedule();                                                  (16)

```

(下页继续)

(续上页)

```
    return RT_EOK;
}
RTM_EXPORT(rt_mutex_release);
```

代码清单 20-6 **(1)**：根据传递进来的互斥量句柄释放互斥量。

代码清单 20-6 **(2)**：获取当前线程。只有已持有互斥量所有权的线程才能释放它，不然的话，会出现问题，就像被盗版一样，每个人都可以做到根本不知道哪个是真的哪个是假的，这样子互斥量就无法保证资源访问是绝对安全的。

代码清单 20-6 **(3)**：判断当前线程与互斥量持有者是不是同一个线程，只有互斥量持有者才能释放互斥量，如果不是则返回错误代码-RT_ERROR。

代码清单 20-6 **(4)**：如果当前线程是该互斥量的持有者，那么互斥量可以被释放，将持有量减一。

代码清单 20-6 **(5)**：如果线程释放了互斥量，则会进行 **(5)-(15)** 操作。

代码清单 20-6 **(6)**：如果当前线程初始设置的优先级与互斥量保存的优先级不一样，那么则要恢复线程初始化设定的优先级，调用 rt_thread_control 函数重置线程的优先级。

代码清单 20-6 **(7)**：如果有由于获取不到互斥量而进入阻塞的线程，那么此时互斥量为开锁状态就需要将这些线程唤醒，执行 **(8)-(12)** 操作。

代码清单 20-6 **(8)**：获取当前被阻塞的线程，。

代码清单 20-6 **(9)**：设置互斥量持有者为从阻塞中恢复的新线程，将 mutex->owner 执行新线程的线程控制块。

代码清单 20-6 **(10)**：保存新持有互斥量线程的优先级。

代码清单 20-6 **(11)**：持有互斥量次数加一。

代码清单 20-6 **(12)**：恢复被阻塞的线程，并且需要进行一次线程调度，然后执行 **(16)**，进行线程调度。

代码清单 20-6 **(13)**：如果释放了信号量，但此时没有线程被阻塞的话，将互斥量的值加一，表示此时互斥量处于开锁状态，其他线程可以获取互斥量。

代码清单 20-6 **(14)**：清除互斥量信息，恢复互斥量的初始状态，因为没有线程持有互斥量。

代码清单 20-6 **(15)**：持有互斥量的线程优先级恢复默认 0xff。

代码清单 20-6 **(16)**：进行一次线程调度。

使用该函数接口时，只有已经拥有互斥量控制权的线程才能释放它，每释放一次该互斥量，它的持有计数就减 1。当该互斥量的持有计数为零时（即持有线程已经释放所有的持有操作），它变为可用，等待在该信号量上的线程将被唤醒。如果线程的运行优先级被互斥量提升，那么当互斥量被释放后，线程恢复为持有互斥量前的优先级。学习是一个循序渐进的过程，我们学习了互斥量释放过程的源码，那么，接下来就是要学会怎么去使用这个互斥量释放函数 rt_mutex_release()，具体见代码清单 20-7 加粗部分。

列表 7: 代码清单 20-7 互斥量释放函数 rt_mutex_release() 实例

```
/* 定义消息队列控制块 */
static rt_mutex_t test_mutex = RT_NULL;

rt_err_t uwRet = RT_EOK;

uwRet = rt_mutex_release(test_mutex);
if (RT_EOK == uwRet)
rt_kprintf("互斥量释放成功！\n\n");
```

23.6.4 互斥量获取函数 rt_mutex_take()

释放互斥量对应的是获取互斥量，我们知道，当互斥量处于开锁的状态，线程才能获取互斥量成功，当线程持有了某个互斥量的时候，其它线程就无法获取这个互斥量，需要等到持有互斥量的线程进行释放后，其他线程才能获取成功，线程通过互斥量 rt_mutex_take() 函数获取互斥量的所有权。线程对互斥量的所有权是独占的，任意时刻互斥量只能被一个线程持有，如果互斥量处于开锁状态，那么获取该互斥量的线程将成功获得该互斥量，并拥有互斥量的使用权；如果互斥量处于闭锁状态，获取该互斥量的线程将无法获得互斥量，线程将被挂起，直到持有互斥量线程释放它，而如果线程本身就持有互斥量，再去获取这个互斥量却不会被挂起，只是将该互斥量的持有值加 1，下面一起来看看互斥量获取函数 rt_mutex_take()，具体见代码清单 20-8。

列表 8: 代码清单 20-8 互斥量获取函数 rt_mutex_take() 源码

```
rt_err_t rt_mutex_take(rt_mutex_t mutex,                                (1)
                       rt_int32_t time)                                 (2)
{
    register rt_base_t temp;
    struct rt_thread *thread;

    /* 即使 time = 0, 也不得在中断中使用此功能 */
    RT_DEBUG_IN_THREAD_CONTEXT;

    RT_ASSERT(mutex != RT_NULL);                                        (3)

    /* 获取当前线程 */
    thread = rt_thread_self();                                          (4)

    /* 关中断 */
    temp = rt_hw_interrupt_disable();                                   (5)

    RT_OBJECT_HOOK_CALL(rt_object_trytake_hook, (&(mutex->parent.parent)));

    RT_DEBUG_LOG(RT_DEBUG_IPC,
        ("mutex_take: current thread %s, mutex value: %d, hold: %d\n",
                thread->name, mutex->value, mutex->hold));

    /* 设置线程错误码 */
    thread->error = RT_EOK;

    if (mutex->owner == thread) {
        /* 如果是同一个线程 */
        mutex->hold ++;                                                 (6)
```

(下页继续)

(续上页)

```
    } else {
__again:
        /*
        * 初始状态下互斥量的值为 1。 因此，如果该值大于 0，则表示可以使用互斥量。
        */
        if (mutex->value > 0) {                                    (7)
            /* 互斥量可用 */
            mutex->value --;

            /* 记录申请互斥量的线程与它的初始化优先级 */
            mutex->owner             = thread;                     (8)
            mutex->original_priority = thread->current_priority;   (9)
            mutex->hold ++;                                        (10)
        } else {
            /* 如果不等待，返回超时错误代码 */
            if (time == 0) {                                       (11)
                /* 设置线程错误码 */
                thread->error = -RT_ETIMEOUT;

                /* 开中断 */
                rt_hw_interrupt_enable(temp);

                return -RT_ETIMEOUT;
            } else {                                               (12)
                /* 互斥量不可用，挂起线程 */
        RT_DEBUG_LOG(RT_DEBUG_IPC, ("mutex_take: suspend thread: %s\n",
                                    thread->name));

                /* 判断申请互斥量线程与持有互斥量线程的优先级关系 */
        if (thread->current_priority < mutex->owner->current_priority) {
                    /* 改变持有互斥量的线程优先级 */                (13)
                    rt_thread_control(mutex->owner,
```

(下页继续)

(续上页)

```
                                RT_THREAD_CTRL_CHANGE_PRIORITY,
                                &thread->current_priority);      (14)
            }

            /* 挂起当前线程 */
            rt_ipc_list_suspend(&(mutex->parent.suspend_thread),
                                thread,
                                mutex->parent.parent.flag);     (15)

            /* 有等待时间，开始计时 */
            if (time > 0) {                                     (16)
                RT_DEBUG_LOG(RT_DEBUG_IPC,
                    ("mutex_take: start the timer of thread:%s\n",
                            thread->name));

                /* 重置线程计时器的超时时间并启动它 */
                rt_timer_control(&(thread->thread_timer),
                                RT_TIMER_CTRL_SET_TIME,
                                &time);                         (17)
                rt_timer_start(&(thread->thread_timer));        (18)
            }

            /* 开中断 */
            rt_hw_interrupt_enable(temp);

            /* 发起线程调度 */
            rt_schedule();                                      (19)

            if (thread->error != RT_EOK) {
                /* 再试一次 */
                if (thread->error == -RT_EINTR) goto __again;   (20)

```

(下页继续)

(续上页)

```
                    /* 返回错误代码 */
                    return thread->error;
                } else {
                    /* 获取信号量成功 */
                    /* 关中断 */
                    temp = rt_hw_interrupt_disable();                (21)
                }
            }
        }
    }

    /* 开中断 */
    rt_hw_interrupt_enable(temp);

    RT_OBJECT_HOOK_CALL(rt_object_take_hook, (&(mutex->parent.parent)));

    return RT_EOK;
}
RTM_EXPORT(rt_mutex_take);
```

代码清单 20-8 **(1)**：mutex 是互斥量对象句柄，在使用获取信号量之前必须先创建互斥量。

代码清单 20-8 **(2)**：time 是指定等待的时间。

代码清单 20-8 **(3)**：判断互斥量是否有效。必须是已经创建的互斥量才能进行获取操作。

代码清单 20-8 **(4)**：获取当前线程。系统要知道是哪个线程获取互斥量。

代码清单 20-8 **(5)**：关中断，防止下面的操作被打断。

代码清单 20-8 **(6)**：如果持有互斥量的线程与当前获取互斥量的线程是同一个线程的话，则不会发生阻塞，将互斥量的持有次数加一，此处是互斥量的递归调用，不会造成死锁。如果不是同一个线程的话则执行 **(7)-(21)** 操作。

代码清单 20-8 **(7)**：如果线程能获取到互斥量，则执行 **(7)-(10)** 操作。初始状态下互斥量 value 的

值为 1。因此，如果该值大于 0，则表示此互斥量处于开锁状态，线程可以获取互斥量。当获取到互斥量的时候，value 的值会减一，表示超时互斥量处于闭锁状态，其他线程无法获取互斥量。

代码清单 20-8 **(8)**：记录当前获取到互斥量的线程信息，重置 owner 指向当前线程。

代码清单 20-8 **(9)**：记录当前获取到互斥量的线程的优先级信息，original_priority 设置为当前线程的优先级。

代码清单 20-8 **(10)**：互斥量的持有次数加一。

代码清单 20-8 **(11)**：如果没有申请到互斥量，执行 **(11)-(21)** 操作。如果不设置等待时间，则直接返回错误码。

代码清单 20-8 **(12)**：如果没有申请到，但是设置了等待时间，那么可以根据等待时间将线程挂起。

代码清单 20-8 **(13)**：判断当前线程与持有互斥量线程的优先级关系，如果当前线程优先级比持有互斥量线程优先级高，这时候已经发生了优先级翻转了，因为互斥量的原因，高优先级的线程（当前线程）被阻塞着，所以，需要进行优先级继承操作。

代码清单 20-8 **(14)**：发生优先级翻转，需要暂时改变持有互斥量的线程优先级，将其优先级暂时提高到当前线程的优先级。

代码清单 20-8 **(15)**：挂起当前线程，进行等待。

代码清单 20-8 **(16)**：有等待时间，开始计时。

代码清单 20-8 **(17)**：重置线程计时器的超时时间。

代码清单 20-8 **(18)**：启动定时器。

代码清单 20-8 **(19)**：发起线程调度。

代码清单 20-8 **(20)**：回到 __again。

代码清单 20-8 **(21)**：获取信号量成功。

下面来学习一下任何使用互斥量获取函数 rt_mutex_take()，实例具体见*代码清单 20-9* 加粗部分。

列表 9: 代码清单 20-9 互斥量获取函数 rt_mutex_take() 实例

```
/* 定义消息队列控制块 */
static rt_mutex_t test_mutex = RT_NULL;

rt_err_t uwRet = RT_EOK;

rt_mutex_take(test_mux,                /* 获取互斥量 */
            RT_WAITING_FOREVER);     /* 等待时间: 一直等 */
if (RT_EOK == uwRet)
    rt_kprintf("互斥量获取成功! \n\n");
```

23.7 互斥量使用注意事项

使用互斥量时候需要注意几点:

1. 两个线程不能对同时持有同一个互斥量。如果某线程对已被持有的互斥量进行获取，则该线程会被挂起，直到持有该互斥量的线程将互斥量释放成功，其他线程才能申请这个互斥量。

2. 互斥量不能在中断服务程序中使用。

3. RT-Thread 作为实时操作系统需要保证线程调度的实时性，尽量避免线程的长时间阻塞，因此在获得互斥量之后，应该尽快释放互斥量。

4. 持有互斥量的过程中，不得再调用 rt_thread_control() 等函数接口更改持有互斥量线程的优先级。

23.8 互斥量实验

互斥量同步实验是在 RT-Thread 中创建了两个线程，一个是申请互斥量线程，一个是释放互斥量线程，两个线程独立运行，申请互斥量线程是一直在等待互斥量线程的释放互斥量，其等待时间是 RT_WAITING_FOREVER，一直在等待，等到获取到互斥量之后，进行处理完它又马上释放互斥量。

释放互斥量线程模拟占用互斥量，延时的时间接收线程无法获得互斥量，等到线程使用互斥量完毕，然后进行互斥量的释放，接收线程获得互斥量，然后形成两个线程间的同步，若是线程正常同步，则在串口打印出信息，具体见代码清单 20-10 加粗部分。

列表 10: 代码清单 20-10 互斥量实验

```
/**
*********************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2019-xx-xx
* @brief   RT-Thread 互斥量同步
*********************************************************************
* @attention
*
* 实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
**********************************************************************
*/

/*
**********************************************************************
*                              包含的头文件
```

(下页继续)

(续上页)

```
**********************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"
/* RT-Thread 头文件 */
#include "rtthread.h"
/*
*****************************************************************
*                                 变量
*****************************************************************
*/
/* 定义线程控制块 */
static rt_thread_t receive_thread = RT_NULL;
static rt_thread_t send_thread = RT_NULL;
/* 定义互斥量控制块 */
static rt_mutex_t test_mux = RT_NULL;

/************************* 全局变量声明 ****************************/
/*
* 当我们在写应用程序的时候，可能需要用到一些全局变量。
*/
uint8_t ucValue [ 2 ] = { 0x00, 0x00 };
/*
**********************************************************************
*                                 函数声明
**********************************************************************
*/
static void receive_thread_entry(void* parameter);
static void send_thread_entry(void* parameter);

/*
**********************************************************************
```

(下页继续)

(续上页)

```
*                                   main 函数
*****************************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]- i.MX RT-RTT 互斥量同步实验！\n");
    rt_kprintf("同步成功则输出 Successful，反之输出 Fail\n");
    /* 创建一个互斥量 */
    test_mux = rt_mutex_create("test_mux",RT_IPC_FLAG_PRIO);

    if (test_mux != RT_NULL)
        rt_kprintf("互斥量创建成功！\n\n");

    receive_thread =                          /* 线程控制块指针 */
        rt_thread_create( "receive",              /* 线程名字 */
                        receive_thread_entry,   /* 线程入口函数 */
                        RT_NULL,              /* 线程入口函数参数 */
                        512,                  /* 线程栈大小 */
                        3,                    /* 线程的优先级 */
                        20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
```

(下页继续)

(续上页)

```
    if (receive_thread != RT_NULL)
        rt_thread_startup(receive_thread);
    else
        return -1;

    send_thread =                           /* 线程控制块指针 */
        rt_thread_create( "send",               /* 线程名字 */
                        send_thread_entry,   /* 线程入口函数 */
                        RT_NULL,             /* 线程入口函数参数 */
                        512,                 /* 线程栈大小 */
                        2,                   /* 线程的优先级 */
                        20);                 /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (send_thread != RT_NULL)
        rt_thread_startup(send_thread);
    else
        return -1;
}

/*
***********************************************************
*                              线程定义
*********************************************************
*/

static void receive_thread_entry(void* parameter)
{
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        rt_mutex_take(test_mux,                /* 获取互斥量 */
                      RT_WAITING_FOREVER);     /* 等待时间：一直等 */
```

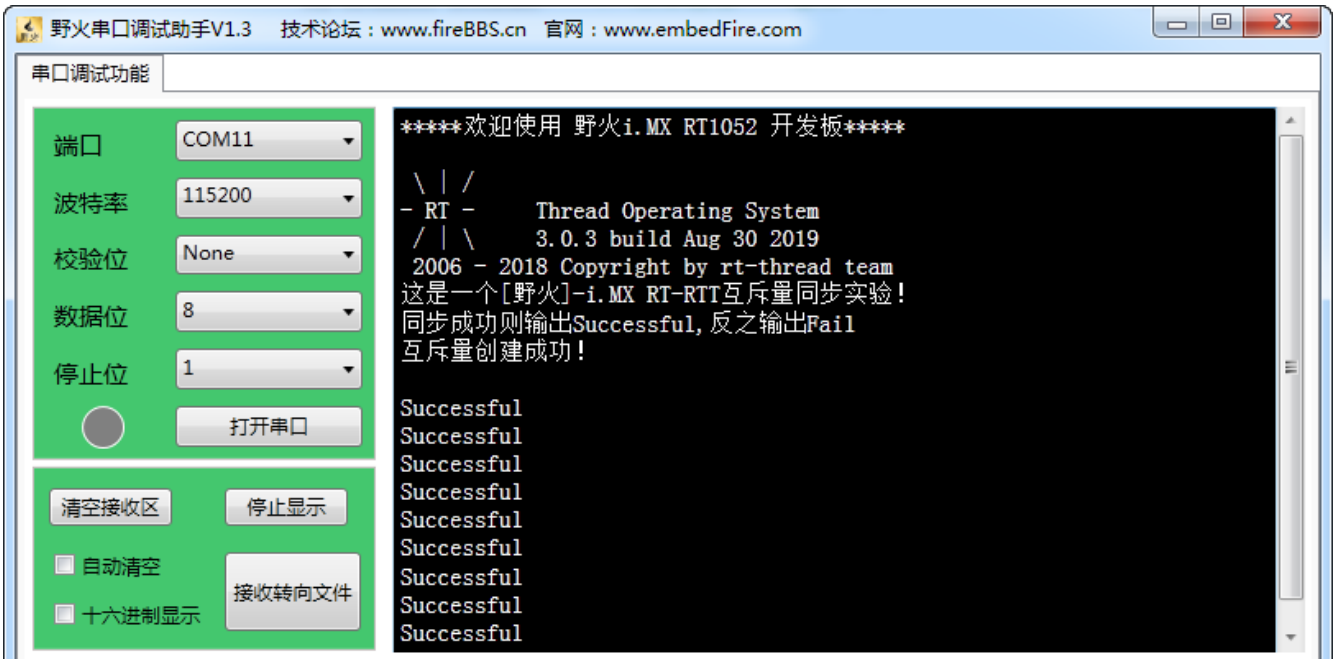
(下页继续)

(续上页)

```
        if ( ucValue [ 0 ] == ucValue [ 1 ] ) {
            rt_kprintf ( "Successful\n" );
        } else {
            rt_kprintf ( "Fail\n" );
        }
        rt_mutex_release(   test_mux        );                //释放互斥量

        rt_thread_delay ( 100 );                    //每 1s 读一次
    }
}



static void send_thread_entry(void* parameter)
{
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        rt_mutex_take(test_mux,             /* 获取互斥量 */
                    RT_WAITING_FOREVER);    /* 等待时间：一直等 */
        ucValue [ 0 ] ++;
        rt_thread_delay ( 10 );             /* 延时 100ms */
        ucValue [ 1 ] ++;
        rt_mutex_release(test_mux);                 //释放互斥号量
        rt_thread_yield();  //放弃剩余时间片，进行一次线程切换
    }
}
/**********************END OF FILE***************************/
```

23.9 实验现象

将程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息，它里面输出了信息表明线程正在运行中，当输出信息为 Successful 的时候，则表面两个线程同步成功，具体见图 *20-4*。



第 24 章　事件

24.1 事件的基本概念

事件是一种实现线程间通信的机制，主要用于实现线程间的同步，但事件通信只能是事件类型的通信，无数据传输。与信号量不同的是，它可以实现一对多，多对多的同步。即一个线程可以等待多个事件的发生：可以是任意一个事件发生时唤醒线程进行事件处理；也可以是几个事件都发生后才唤醒线程进行事件处理。同样，事件也可以是多个线程同步多个事件。

事件集合用 32 位无符号整型变量来表示，每一位代表一个事件，线程通过“逻辑与”或“逻辑或”与一个或多个事件建立关联，形成一个事件集。事件的“逻辑或”也称作是独立型同步，指的是线程感兴趣的所有事件任一件发生即可被唤醒；事件“逻辑与”也称为是关联型同步，指的是线程感兴趣的若干事件都发生时才被唤醒。

多线程环境下，线程之间往往需要同步操作，一个事件发生即是一个同步。事件可以提供一对多、多对多的同步操作。一对多同步模型：一个线程等待多个事件的触发；多对多同步模型：多个线程等待多个事件的触发。

线程可以通过创建事件来实现事件的触发和等待操作。RT-Thread 的事件仅用于同步，不提供数据传输功能。

RT-Thread 提供的事件具有如下特点：

- 事件只与线程相关联，事件相互独立，一个 32 位的事件集合（set 变量），用于标识该线程发生的事件类型，其中每一位表示一种事件类型（0 表示该事件类型未发生、1 表示该事件类型已经发生），一共 32 种事件类型。
- 事件仅用于同步，不提供数据传输功能。
- 事件无排队性，即多次向线程发送同一事件 (如果线程还未来得及读走)，等效于只发送一次。
- 允许多个线程对同一事件进行读写操作。

- 支持事件等待超时机制。

在 RT-Thread 实现中，每个线程都拥有一个事件信息标记，它有三个属性，分别是 RT_EVENT_FLAG_AND(逻辑与)，RT_EVENT_FLAG_OR(逻辑或）以及 RT_EVENT_FLAG_CLEAR(清除标记）。当线程等待事件同步时，可以通过 32 个事件标志和这个事件信息标记来判断当前接收的事件是否满足同步条件。

24.2 事件的应用场景

RT-Thread 的事件用于事件类型的通讯，无数据传输，也就是说，我们可以用事件来做标志位，判断某些事件是否发生了，然后根据结果做处理，那很多人又会问了，为什么我不直接用变量做标志呢，岂不是更好更有效率？非也非也，若是在裸机编程中，用全局变量是最为有效的方法，这点我不否认，但是在操作系统中，使用全局变量就要考虑以下问题了：

- 如何对全局变量进行保护呢，防止多线程同时对它进行访问？
- 如何让内核对事件进行有效管理呢？使用全局变量的话，就需要在线程中轮询查看事件是否发送，这简直就是在浪费 CPU 资源啊，还有等待超时机制，使用全局变量的话需要用户自己去实现。

所以，在操作系统中，还是使用操作系统给我们提供的通信机制就好了，简单方便还实用。

在某些场合，可能需要多个时间发生了才能进行下一步操作，比如一些危险机器的启动，需要检查各项指标，当指标不达标的时候，无法启动，但是检查各个指标的时候，不能一下子检测完毕啊，所以，需要事件来做统一的等待，当所有的事件都完成了，那么机器才允许启动，这只是事件的其中一个应用。

事件可使用于多种场合，它能够在一定程度上替代信号量，用于线程间同步。一个线程或中断服务例程发送一个事件给事件对象，而后等待的线程被唤醒并对相应的事件进行处理。但是它与信号量不同的是，事件的发送操作是不可累计的，而信号量的释放动作是可累计的。事件另外一个特性是，接收线程可等待多种事件，即多个事件对应一个线程或多个线程。同时按照线程等待的参数，可选择是“逻辑或”触发还是“逻辑与”触发。这个特性也是信号量等所不具备的，信号量只能识别单一同步动作，而不能同时等待多个事件的同步。

各个事件可分别发送或一起发送给事件对象，而线程可以等待多个事件，线程仅对感兴趣的事件

进行关注。当有它们感兴趣的事件发生时并且符合感兴趣的条件，线程将被唤醒并进行后续的处理动作。

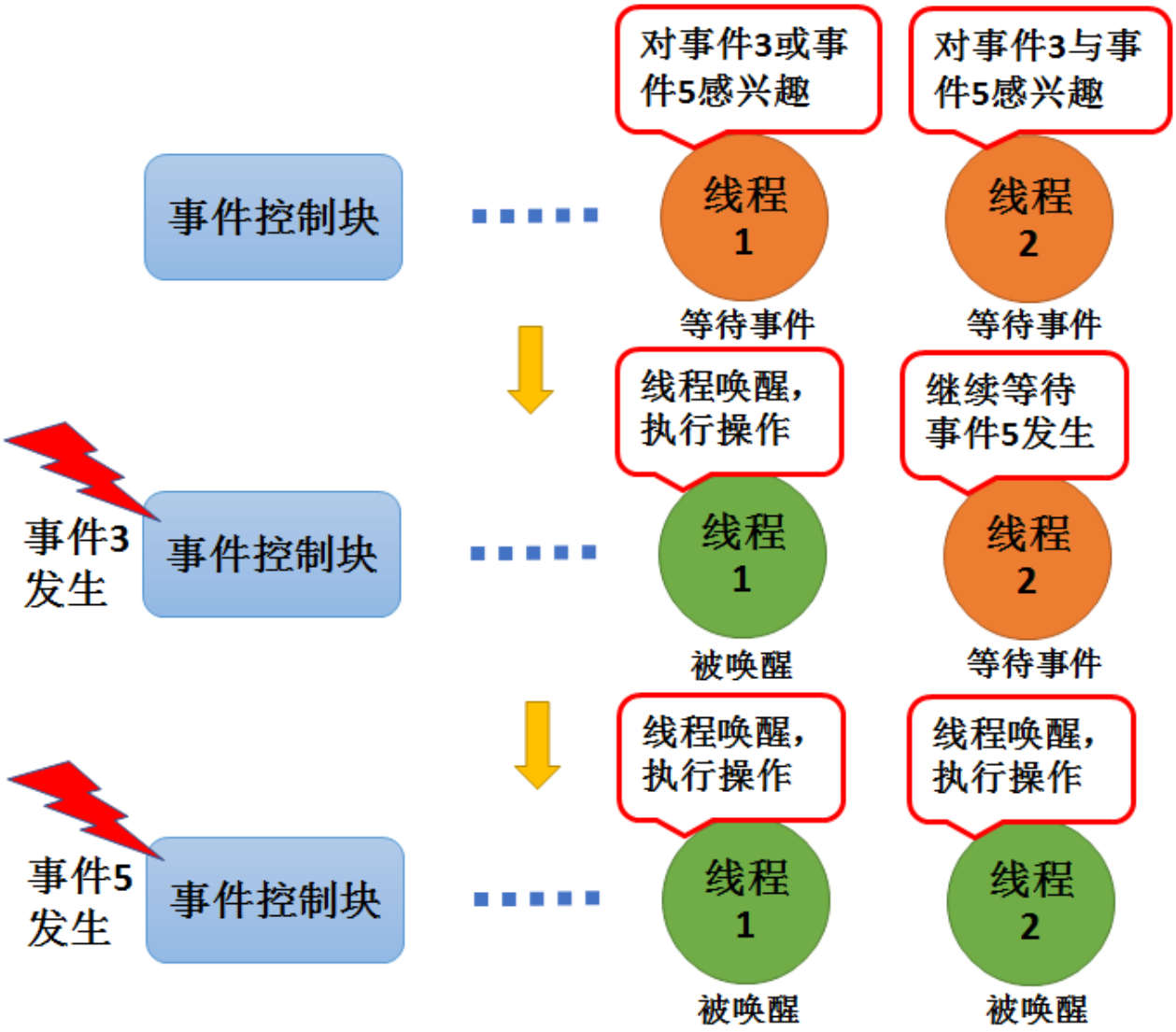
24.3 事件的运作机制

接收事件时，可以根据入感兴趣的参事件类型接收事件的单个或者多个事件类型。事件接收成功后，必须使用 RT_EVENT_FLAG_CLEA 选项来清除已接收到的事件类型，否则不会清除已接收到的事件。用户可以自定义通过传入参数选择读取模式 option，是等待所有感兴趣的事件还是等待感兴趣的任意一个事件。

发送事件时，对指定事件写入指定的事件类型，设置事件集合 set 的对应事件位为 1，可以一次同时写多个事件类型，发送事件会触发线程调度。

清除事件时，根据入参数事件句柄和待清除的事件类型，对事件对应位进行清 0 操作。事件不与线程相关联，事件相互独立，一个 32 位的变量（事件集合 set），用于标识该线程发生的事件类型，其中每一位表示一种事件类型（0 表示该事件类型未发生、1 表示该事件类型已经发生），一共 32 种事件类型具体见图 *21-1*。

事件唤醒机制，当线程因为等待某个或者多个事件发生而进入阻塞态，当事件发生的时候会被唤醒，其过程具体见图 *21-2*。



线程 1 对事件 3 或事件 5 感兴趣（逻辑或 RT_EVENT_FLAG_OR），当发生其中的某一个事件都会被唤醒，并且执行相应操作。而线程 2 对事件 3 与事件 5 感兴趣（逻辑与 RT_EVENT_FLAG_AND），当且仅当事件 3 与事件 5 都发生的时候，线程 2 才会被唤醒，如果只有一个其中一个事件发生，那么线程还是会继续等待事件发生。如果接在收事件函数中 option 设置了清除事件位，那么当线程唤醒后将把事件 3 和事件 5 的事件标志清零，否则事件标志将依然存在。

24.4 事件控制块

事件的使用很简单，每个对事件的操作的函数都是根据事件控制块来进行操作的，事件控制块包含了一个 32 位的 set 变量，其变量的各个位表示一个事件，每一位代表一个事件的发生，利用逻辑或、逻辑与等实现不同事件的不同唤醒处理，具体见*代码清单 21-1*。

列表 1: 代码清单 21-1 事件控制块

```
struct rt_event {
    struct rt_ipc_object parent;

    rt_uint32_t          set;                    /**< 事件标志位 */
};
typedef struct rt_event *rt_event_t; /* rt_event_t 是指向事件结构体的指针 */
```

事件属于内核对象，也会在自身结构体里面包含一个内核对象类型的成员，通过这个成员可以将事件挂到系统对象容器里面。rt_event 对象从 rt_ipc_object 中派生，由 IPC 容器管理。

24.5 事件函数接口讲解

24.5.1 事件创建函数 rt_event_create()

事件创建函数，顾名思义，就是创建一个事件，与其他内核对象一样，都是需要先创建才能使用的资源，RT-Thread 给我们提供了一个创建事件的函数 rt_event_create()，当创建一个事件时，内核首先创建一个事件控制块，然后对该事件控制块进行基本的初始化，创建成功返回事件句柄；创建失败返回 RT_NULL。所以，在使用创建函数之前，我们需要先定义有个事件的句柄，事件创建的源码具体见*代码清单 21-2*。

列表 2: 代码清单 21-2 事件创建函数 rt_event_create() 源码

```
rt_event_t rt_event_create(const char *name, rt_uint8_t flag)        (1)
{
    rt_event_t event;                                                 (2)

    RT_DEBUG_NOT_IN_INTERRUPT;

    /* 分配对象 */
    event = (rt_event_t)rt_object_allocate(RT_Object_Class_Event, name);
    if (event == RT_NULL)                                             (3)
        return event;

    /* 设置阻塞唤醒的模式 */
    event->parent.parent.flag = flag;                                 (4)

    /* 初始化事件对象 */
    rt_ipc_object_init(&(event->parent));                             (5)

    /* 事件集合清零 */
    event->set = 0;                                                   (6)

    return event;                                                     (7)
}
RTM_EXPORT(rt_event_create);
```

代码清单 21-2 **(1)**：name： 事件的名称，由用户自己定义。flag： 事件阻塞唤醒模式。

代码清单 21-2 **(2)**： 创建一个事件控制块。

代码清单 21-2 **(3)**：分配事件对象，调用 rt_object_allocate() 函数将从对象系统分配对象，为创建的事件分配一个事件的对象，并且命名对象名称，在系统中，对象的名称必须是唯一的。

代码清单 21-2 **(4)**：设置事件的阻塞唤醒模式，创建的事件由于指定的 flag 不同，而有不同的意义：使用 RT_IPC_FLAG_PRIO 优先级 flag 创建的 IPC 对象，在多个线程等待资源时，将由优先级高的线程优先获得资源。而使用 RT_IPC_FLAG_FIFO 先进先出 flag 创建的 IPC 对象，在多个线程等待资源时，将按照先来先得的顺序获得资源。RT_IPC_FLAG_PRIO 与 RT_IPC_FLAG_FIFO 均在 rtdef.h 中有定义。

代码清单 21-2 **(5)**：初始化事件内核对象。调用 rt_ipc_object_init() 函数会初始化一个链表用于记录访问此事件而阻塞的线程。

代码清单 21-2 **(6)**：事件集合清零，因为现在是创建事件，还没有事件发生，所以事件集合中所有位都为 0。

代码清单 21-2 **(7)**：创建成功返回事件对象的句柄，创建失败返回 RT_NULL。。

事件创建函数的源码都那么简单，其使用更为简单，不过需要我们在使用前定义一个指向事件控制块的指针，也就是常说的事件句柄，当事件创建成功，我们就可以根据我们定义的事件句柄来调用 RT-Thread 的事件函数进行操作，具体见代码清单 21-3 高亮部分。

列表 3: 代码清单 21-3 事件创建函数 rt_event_create() 实例

```
/* 定义事件控制块（句柄） */
static rt_event_t test_event = RT_NULL;
/* 创建一个事件 */
test_event = rt_event_create("test_event",/* 事件标志组名字 */
                             RT_IPC_FLAG_PRIO); /* 事件模式 FIFO(0x00)*/
if (test_event != RT_NULL)
    rt_kprintf("事件创建成功! \n\n");
```

24.5.2 事件删除函数 rt_event_delete()

在很多场合，某些事件只用一次的，就好比在事件应用场景说的危险机器的启动，假如各项指标都达到了，并且机器启动成功了，那这个事件之后可能就没用了，那就可以进行销毁了。想要删除事件怎么办呢？RT-Thread 给我们提供了一个删除事件的函数——rt_event_delete()，使用它就能将事件进行删除了。当系统不再使用事件对象时，可以通过删除事件对象控制块来释放系统资源，具体见*代码清单 21-4*。

列表 4: 代码清单 21-4 事件删除函数 rt_event_delete() 源码

```
rt_err_t rt_event_delete(rt_event_t event)                          (1)
{
    /* 事件句柄检查 */
    RT_ASSERT(event != RT_NULL);                                    (2)

    RT_DEBUG_NOT_IN_INTERRUPT;

    /* 恢复所有阻塞在此事件的线程 */
    rt_ipc_list_resume_all(&(event->parent.suspend_thread));        (3)
```

(下页继续)

(续上页)

```
    /* 删除事件对象 */
    rt_object_delete(&(event->parent.parent));                    (4)

    return RT_EOK;                                                (5)
}
RTM_EXPORT(rt_event_delete);
```

代码清单 21-4 **(1)**：event 是我们自己定义的事件句柄，根据事件句柄进行删除操作。

代码清单 21-4 **(2)**：检查事件句柄 event 是否有效，如果它是未定义或者未创建的事件句柄，那么是无法进行删除操作的。

代码清单 21-4 **(3)**：调用 rt_ipc_list_resume_all() 函数将所有因为访问此事件的而阻塞的线程从阻塞态中唤醒，所有被唤醒的线程的返回值是-RT_ERROR，一般不这样子使用，所以在删除的时候，应先确认所有的线程都无需再次使用这个事件，并且所有线程都没被此事件阻塞时候才进行删除，否则删除之后线程需要再次使用此事件的话那也会发生错误。

代码清单 21-4 **(4)**：删除事件对象，释放事件对象占用的内存资源。

代码清单 21-4 **(5)**：删除成功返回 RT_EOK。

事件的删除函数使用是很简单的，只需要传递进我们创建的事件对象句柄，其使用方法具体见代码清单 21-5 高亮部分。

列表 5: 代码清单 21-5 事件删除函数 rt_event_delete() 使用实例

```
/* 定义事件控制块（句柄）*/
static rt_event_t test_event = RT_NULL;
rt_err_t uwRet = RT_EOK;
/* 删除一个事件 */
uwRet = rt_event_delete(test_event);
if (RT_EOK == uwRet)
    rt_kprintf("事件删除成功！\n\n");
```

24.5.3 事件发送函数 rt_event_send()

使用该函数接口时，通过参数 set 指定的事件标志来设定事件的标志位，然后遍历等待在 event 事件对象上的等待线程链表，判断是否有线程的事件激活要求与当前事件对象标志值匹配，如果有，则唤醒该线程。简单来说，就是设置我们自己定义的事件标志位为 1，并且看看有没有线程在等待这个事件，有的话就唤醒它，其源码具体见代码清单 21-6。

列表 6: 代码清单 21-6 事件发送函数 rt_event_send() 源码

```
rt_err_t rt_event_send(rt_event_t event,                                (1)
                       rt_uint32_t set)                                 (2)
{
    struct rt_list_node *n;
    struct rt_thread *thread;
    register rt_ubase_t level;
    register rt_base_t status;
    rt_bool_t need_schedule;

    /* 事件对象检查 */
    RT_ASSERT(event != RT_NULL);                                        (3)
    if (set == 0)
        return -RT_ERROR;

    need_schedule = RT_FALSE;                                           (4)
    RT_OBJECT_HOOK_CALL(rt_object_put_hook, (&(event->parent.parent)));

    /* 关中断 */
    level = rt_hw_interrupt_disable();

    /* 设置事件 */
    event->set |= set;                                                  (5)

    if (!rt_list_isempty(&event->parent.suspend_thread)) {              (6)
```

(下页继续)

(续上页)

```
25          /* 搜索线程列表以恢复线程 */
26          n = event->parent.suspend_thread.next;
27          while (n != &(event->parent.suspend_thread)) {
28              /* 找到要恢复的线程 */
29              thread = rt_list_entry(n, struct rt_thread, tlist);      (7)
30
31              status = -RT_ERROR;
32              if (thread->event_info & RT_EVENT_FLAG_AND) {            (8)
33                  if ((thread->event_set & event->set)
34                      == thread->event_set) {                          (9)
35                      /* 收到了一个 AND */
36                      status = RT_EOK;                                 (10)
37                  }
38              } else if (thread->event_info & RT_EVENT_FLAG_OR) {      (11)
39                  if (thread->event_set & event->set) {
40                      /* 保存收到的事件集 */
41                  thread->event_set = thread->event_set & event->set; (12)
42
43                      /* 收到一个 OR */
44                      status = RT_EOK;                                 (13)
45                  }
46              }
47
48              /* 将节点移动到下一个节点 */
49              n = n->next;                                             (14)
50
51              /* 条件满足，恢复线程 */
52              if (status == RT_EOK) {                                  (15)
53                  /* 清除事件标志位 */
54                  if (thread->event_info & RT_EVENT_FLAG_CLEAR)        (16)
55                      event->set &= ~thread->event_set;
56
```

(下页继续)

(续上页)

```
                /* 恢复线程 */
                rt_thread_resume(thread);                          (17)

                /* 需要进行线程调度 */
                need_schedule = RT_TRUE;                           (18)
            }
        }
    }

    /* 开中断 */
    rt_hw_interrupt_enable(level);

    /* 发起一次线程调度 */
    if (need_schedule == RT_TRUE)
        rt_schedule();                                             (19)

    return RT_EOK;
}
RTM_EXPORT(rt_event_send);
```

代码清单 21-6 **(1)**：event：事件发送操作的事件句柄，由用户自己定义，并且需要在创建后使用。

代码清单 21-6 **(2)**：set：设置事件集合中的具体事件，也就是设置 set 中的某些位。

代码清单 21-6 **(3)**：检查事件句柄 event 是否有效，如果它是未定义或者未创建的事件句柄，那么是无法进行发送事件操作的。

代码清单 21-6 **(4)**：need_schedule 用于记录是否进行线程调度，默认不进行线程调度。

代码清单 21-6 **(5)**：设置事件发生的标志位，利用‘|’操作即保证不干扰其他事件位又能同事对多个事件位一次性标记，即使是多次向线程发送同一事件 (如果线程还未来得及读走)，也等效于只发送一次。

代码清单 21-6 **(6)**：如果当前有线程因为等待某个事件进入阻塞态，则在阻塞列表中搜索线程，并且执行 **(7)-(18)**，

代码清单 21-6 **(7)**：从等待的线程中获取对应的线程控制块。

代码清单 21-6 **(8)**：如果线程等待事件的模式是 RT_EVENT_FLAG_AND（逻辑与），那么需要等待的事件都发生时才动作。

代码清单 21-6 **(9)**：判断线程等待的事件是否都发生了，如果事件激活要求与事件标志值匹配，则唤醒事件。

代码清单 21-6 **(10)**：当等待的事件都发生的时候，进行标记 status 动作，表示事件已经等待到了。

代码清单 21-6 **(11)**：如果线程等待事件的模式是 RT_EVENT_FLAG_OR（逻辑或），那么线程等待的所有事件标记中只要有一个或多个事件发生了就表示事件已发生，可以唤醒线程。

代码清单 21-6 **(12)**：保存收到的事件，这个很重要，因为在接收事件函数的时候，这个值是要用来进行判断的，假设有一个线程等待接收 3 个事件，采用 RT_EVENT_FLAG_OR（逻辑或）的方式等待接收，那么有其中一个事件发生，该线程就会解除阻塞，但是我们假如没保存收到的事件的话，我们怎么知道是哪个事件发生呢?

代码清单 21-6 **(13)**：当等待的事件发生的时候，进行标记 status 动作，表示事件已经等待到了

代码清单 21-6 **(14)**：将节点移动到下一个节点，因为这是搜索所有等待的线程。

代码清单 21-6 **(15)**：当等待的事件发生的时候，条件满足，需要恢复线程。

代码清单 21-6 **(16)**：如果在接收中设置了 RT_EVENT_FLAG_CLEAR，那么在线程被唤醒的时候，系统会进行事件标志位的清除操作，防止一直响应事件。采用 event->set &= ~thread->event_set 操作仅仅是清除对应事件标志位，不影响其他事件标志位。

代码清单 21-6 **(17)**：恢复阻塞的线程。

代码清单 21-6 **(18)**：标记一下 need_schedule 表示需要进行线程调度。

代码清单 21-6 **(19)**：发起一次线程调度。

举个例子，比如我们要记录一个事件的发生，这个事件在事件集合的位置是 bit0，当它还未发生的时候，那么事件集合 bit0 的值也是 0，当它发生的时候，我们往事件集合 bit0 中写入这个事件，也就是 0x01，那这就表示事件已经发生了，为了便于理解，一般操作我们都是用宏定义来实现 #define EVENT (0x01 « x)，“« x”表示写入事件集合的 bit x ，具体见代码清单 21-7 高亮部分。

列表 7: 代码清单 21-7 事件发送函数 rt_event_send() 实例

```
#define KEY1_EVENT  (0x01 << 0)//设置事件掩码的位 0
#define KEY2_EVENT  (0x01 << 1)//设置事件掩码的位 1
static void send_thread_entry(void* parameter)
{
    /* 线程都是一个无限循环，不能返回 */
    while (1) {//如果 KEY2 被单击
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_GPIO_PIN) == KEY_ON ) {
            rt_kprintf ( "KEY1 被单击\n" );
            /* 发送一个事件 1 */
            rt_event_send(test_event,KEY1_EVENT);
        }
        //如果 KEY2 被单击
        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_GPIO_PIN) == KEY_ON ) {
            rt_kprintf ( "KEY2 被单击\n" );
            /* 发送一个事件 2 */
            rt_event_send(test_event,KEY2_EVENT);
        }
        rt_thread_delay(20);      //每 20ms 扫描一次
    }
}
```

24.5.4 事件接受函数 rt_event_recv()

既然标记了事件的发生，那么我怎么知道他到底有没有发生，这也是需要一个函数来获取事件发生的标记，RT-Thread 提供了一个接收指定事件的函数——rt_event_recv()，通过这个函数，我们知道事件集合中的哪一位，哪一个事件发生了，我们可以通过“逻辑与”、“逻辑或”等操作对感兴趣的事件进行接收，并且这个函数实现了等待超时机制，如果此刻该事件没有发生，那么线程可以进入阻塞态进行等待，等到事件发生了就会被唤醒，很有效的体现了操作系统的实时性，如果事件正确接收则返回 RT_EOK，事件接收超时则返回-RT_ETIMEOUT，其他情况返

回-RT_ERROR，事件接受函数 rt_event_recv() 源码具体见代码清单 21-8。

列表 8: 代码清单 21-8 事件接受函数 rt_event_recv() 源码

```
rt_err_t rt_event_recv(rt_event_t   event,                    (1)
                       rt_uint32_t  set,                      (2)
                       rt_uint8_t   option,                   (3)
                       rt_int32_t   timeout,                  (4)
                       rt_uint32_t *recved)                   (5)
{
    struct rt_thread *thread;
    register rt_ubase_t level;
    register rt_base_t status;

    RT_DEBUG_IN_THREAD_CONTEXT;

    /* 检查事件句柄 */
    RT_ASSERT(event != RT_NULL);                              (6)
    if (set == 0)
        return -RT_ERROR;

    /* 初始化状态 */
    status = -RT_ERROR;
    /* 获取当前线程 */
    thread = rt_thread_self();                                (7)
    /* 重置线程错误码 */
    thread->error = RT_EOK;

    RT_OBJECT_HOOK_CALL(rt_object_trytake_hook, (&(event->parent.parent)));

    /* 关中断 */
    level = rt_hw_interrupt_disable();

```

(下页继续)

(续上页)

```
    /* 检查事件接收选项& 检查事件集合 */
    if (option & RT_EVENT_FLAG_AND) {                          (8)
        if ((event->set & set) == set)
            status = RT_EOK;
    } else if (option & RT_EVENT_FLAG_OR) {                    (9)
        if (event->set & set)
            status = RT_EOK;
    } else {
        /* 应设置 RT_EVENT_FLAG_AND 或 RT_EVENT_FLAG_OR */
        RT_ASSERT(0);                                          (10)
    }

    if (status == RT_EOK) {
        /* 返回接收的事件 */
        if (recved)
            *recved = (event->set & set);                      (11)

        /* 接收事件清除 */
        if (option & RT_EVENT_FLAG_CLEAR)                      (12)
            event->set &= ~set;
    } else if (timeout == 0) {                                 (13)
        /* 不等待 */
        thread->error = -RT_ETIMEOUT;
    } else {
        /* 设置线程事件信息 */
        thread->event_set  = set;                              (14)
        thread->event_info = option;

        /* 将线程添加到阻塞列表中 */
        rt_ipc_list_suspend(&(event->parent.suspend_thread),(15)
                            thread,
                            event->parent.parent.flag);
```

(下页继续)

(续上页)

```
        /* 如果有等待超时，则启动线程计时器 */
        if (timeout > 0) {
            /* 重置线程超时时间并且启动定时器 */
            rt_timer_control(&(thread->thread_timer),          (16)
                             RT_TIMER_CTRL_SET_TIME,
                             &timeout);
            rt_timer_start(&(thread->thread_timer));           (17)
        }

        /* 开中断 */
        rt_hw_interrupt_enable(level);

        /* 发起一次线程调度 */
        rt_schedule();                                         (18)

        if (thread->error != RT_EOK) {
            /* 返回错误代码 */
            return thread->error;                              (19)
        }

        /* 接收一个事件，失能中断 */
        level = rt_hw_interrupt_disable();

        /* 返回接收到的事件 */
        if (recved)
            *recved = thread->event_set;                       (20)
    }

    /* 开中断 */
    rt_hw_interrupt_enable(level);

```

(下页继续)

(续上页)

```
    RT_OBJECT_HOOK_CALL(rt_object_take_hook, (&(event->parent.parent)));

    return thread->error;                                          (21)
}
RTM_EXPORT(rt_event_recv);
```

代码清单 21-8 **(1)**：event：事件发送操作的事件句柄，由用户自己定义，并且需要在创建事件后使用。

代码清单 21-8 **(2)**：set：事件集合中的事件标志，在这里是指线程对哪些事件标志感兴趣。

代码清单 21-8 **(3)**：option：接收选项，有 RT_EVENT_FLAG_AND、RT_EVENT_FLAG_OR，可以与 RT_EVENT_FLAG_CLEAR 通过“|”按位或操作符连接使用。

代码清单 21-8 **(4)**：timeout 是设置等待的超时时间。

代码清单 21-8 **(5)**：recved 用于保存接收到的事件标志结果，用户通过它的值判断是否成功接收到事件。

代码清单 21-8 **(6)**：检查事件句柄 event 是否有效，如果它是未定义或者未创建的事件句柄，那么是无法接收事件的。

代码清单 21-8 **(7)**：获取当前线程信息，即获取调用接收事件的线程。

代码清单 21-8 **(8)**：如果指定的 option 接收选项是 RT_EVENT_FLAG_AND，那么判断事件集合里面的信息与线程感兴趣的信息是否全部吻合，如果满足条件则标记接收成功。

代码清单 21-8 **(9)**：如果指定的 option 接收选项是 RT_EVENT_FLAG_OR，那么判断事件集合里面的信息与线程感兴趣的信息是否有吻合的部分（有其中一个满足即可），如果满足条件则标记接收成功。

代码清单 21-8 **(10)**：其他情况，接收选项应设置 RT_EVENT_FLAG_AND 或 RT_EVENT_FLAG_OR，他们无法同时使用，也不能不使用。

代码清单 21-8 **(11)**：满足接收事件的条件，则返回接收的事件，读取 recved 即可知道接收到了哪个事件。

代码清单 21-8 **(12)**：如果指定的 option 接收选项选择了 RT_EVENT_FLAG_CLEAR，在接收完成的时候会清除对应的事件集合的标志位。

代码清单 21-8 **(13)**：如果 timeout= 0，那么接收不到事件就不等待，直接返回-RT_ETIMEOUT 错误码。

代码清单 21-8 **(14)**：timeout 不为 0，需要等待，那么需要配置线程接收事件的信息，event_set 与 event_info 在线程控制块中有定义，event_set 表示当前线程等待哪些感兴趣的事件，event_info 表示事件接收选项 option。

代码清单 21-8 **(15)**：将等待的线程添加到阻塞列表中。

代码清单 21-8 **(16)**：根据 timeout 的值重置线程超时时间。

代码清单 21-8 **(17)**：启动定时器开始计时。

代码清单 21-8 **(18)**：发起一次线程调度。

代码清单 21-8 **(19)**：返回错误代码。

代码清单 21-8 **(20)**：返回接收到的事件

代码清单 21-8 **(21)**：返回接收成功结果。

当用户调用这个接口时，系统首先根据 set 参数和接收选项来判断它要接收的事件是否发生，如果已经发生，则根据参数 option 上是否设置有 RT_EVENT_FLAG_CLEAR 来决定是否清除事件的相应标志位，其中 recved 参数用于保存收到的事件；如果事件没有发生，则把线程感兴趣的事件和接收选项填写到线程控制块中，然后把线程挂起在此事件对象的阻塞列表上，直到事件发生或等待时间超时，事件接受函数 rt_event_recv() 具体用法见代码清单 21-9 高亮部分。

列表 9: 代码清单 21-9 事件接受函数 rt_event_recv() 实例

```
static void receive_thread_entry(void* parameter)
{
    rt_uint32_t recved;
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        /* 等待接收事件标志 */
```

(下页继续)

(续上页)

```
        rt_event_recv(test_event,            /* 事件对象句柄 */
                      KEY1_EVENT|KEY2_EVENT,  /* 接收线程感兴趣的事件 */
                      RT_EVENT_FLAG_AND|RT_EVENT_FLAG_CLEAR,/* 接收选项 */
                      RT_WAITING_FOREVER,     /* 指定超时事件，一直等 */
                      &recved);               /* 指向接收到的事件 */
        if (recved == (KEY1_EVENT|KEY2_EVENT)) { /* 如果接收完成并且正确 */
            rt_kprintf ( "Key1 与 Key2 都按下\n");
            LED1_TOGGLE;        //LED1        反转
        } else
            rt_kprintf ( "事件错误！\n");
    }
}
```

24.6 事件实验

事件标志组实验是在 RT-Thread 中创建了两个线程，一个是发送事件线程，一个是接收事件线程，两个线程独立运行，发送事件线程通过检测按键的按下情况发送不同的事件，接收事件线程则接收这两个事件，并且判断两个事件是否都发生，如果是则输出相应信息，LED 进行翻转。接收线程的等待时间是 RT_WAITING_FOREVER，一直在等待事件的发生，接收事件之后进行清除事件标记，具体见*代码清单 21-10* 高亮部分。

列表 10: 代码清单 21-10 事件实验

```
/**
*****************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2019-xx-xx
* @brief   RT-Thread 事件标志组
```

(下页继续)

(续上页)

```
**********************************************************************
* @attention
*
* 实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
**********************************************************************
*/

/*
**********************************************************************
*                              包含的头文件
**********************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"
/* RT-Thread 头文件 */
#include "rtthread.h"
/*
*********************************************************************
*                              变量
*********************************************************************
*/
/* 定义线程控制块 */
static rt_thread_t receive_thread = RT_NULL;
static rt_thread_t send_thread = RT_NULL;
/* 定义事件控制块 (句柄) */
static rt_event_t test_event = RT_NULL;

/************************* 全局变量声明 ****************************/
/*
```

(下页继续)

(续上页)

```
* 当我们在写应用程序的时候，可能需要用到一些全局变量。
*/
#define KEY1_EVENT  (0x01 << 0)//设置事件掩码的位 0
#define KEY2_EVENT  (0x01 << 1)//设置事件掩码的位 1
/*
**************************************************************************
*                                 函数声明
**************************************************************************
*/
static void receive_thread_entry(void* parameter);
static void send_thread_entry(void* parameter);

/*
**************************************************************************
*                                 main 函数
**************************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]- i.MX RT 全系列开发板-RTT 事件标志组实验！\n");
    /* 创建一个事件 */
    test_event = rt_event_create("test_event",/* 事件标志组名字 */
```

(下页继续)

(续上页)

```
                        RT_IPC_FLAG_PRIO); /* 事件模式 FIFO(0x00)*/
    if (test_event != RT_NULL)
        rt_kprintf("事件创建成功! \n\n");

    receive_thread =                            /* 线程控制块指针 */
        rt_thread_create( "receive",                /* 线程名字 */
                        receive_thread_entry,    /* 线程入口函数 */
                        RT_NULL,                /* 线程入口函数参数 */
                        512,                    /* 线程栈大小 */
                        3,                      /* 线程的优先级 */
                        20);                    /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (receive_thread != RT_NULL)
        rt_thread_startup(receive_thread);
    else
        return -1;

    send_thread =                           /* 线程控制块指针 */
        rt_thread_create( "send",               /* 线程名字 */
                        send_thread_entry,    /* 线程入口函数 */
                        RT_NULL,                /* 线程入口函数参数 */
                        512,                    /* 线程栈大小 */
                        2,                      /* 线程的优先级 */
                        20);                    /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (send_thread != RT_NULL)
        rt_thread_startup(send_thread);
    else
        return -1;
}
```

(下页继续)

(续上页)

```

/*
*****************************************************************
*                                线程定义
*****************************************************************
*/

static void receive_thread_entry(void* parameter)
{
    rt_uint32_t recved;
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        /* 等待接收事件标志 */
        rt_event_recv(test_event,  /* 事件对象句柄 */
                      KEY1_EVENT|KEY2_EVENT,/* 接收线程感兴趣的事件 */
                      RT_EVENT_FLAG_AND|RT_EVENT_FLAG_CLEAR,/* 接收选项 */
                      RT_WAITING_FOREVER,/* 指定超时事件，一直等 */
                      &recved);    /* 指向接收到的事件 */
        if (recved == (KEY1_EVENT|KEY2_EVENT)) { /* 如果接收完成并且正确 */
            rt_kprintf ( "Key1 与 Key2 都按下\n");
            LED1_TOGGLE;        //LED1       反转
        } else
            rt_kprintf ( "事件错误！\n");
    }
}

static void send_thread_entry(void* parameter)
{
    /* 线程都是一个无限循环，不能返回 */
    while (1) { //如果 KEY2 被单击
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_GPIO_PIN) == KEY_ON ) {
            rt_kprintf ( "KEY1 被单击\n" );
```

(下页继续)

(续上页)

```
            /* 发送一个事件 1 */
            rt_event_send(test_event,KEY1_EVENT);
        }
        //如果 KEY2 被单击
        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_GPIO_PIN) == KEY_ON ) {
            rt_kprintf ( "KEY2 被单击\n" );
            /* 发送一个事件 2 */
            rt_event_send(test_event,KEY2_EVENT);
        }
        rt_thread_delay(2);      //每 20ms 扫描一次
    }
}
/*********************END OF FILE***************************/
```

24.7 实验现象

程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息，按下开发版的 K1 按键发送事件 1，按下 K2 按键发送事件 2；我们按下 K1 与 K2 试试，在串口调试助手中可以看到运行结果，并且当时间 1 与事件 2 都发生的时候，开发板的 LED 会进行翻转，具体见图 *21-3*。

野火串口调试助手V1.3 技术论坛：www.fireBBS.cn 官网：www.embedFire.com
串口调试功能
端口
COM11
波特率
115200
校验位
None
数据位
8
停止位
1
关闭串口
清空接收区
停止显示
自动清空
十六进制显示
接收转向文件
*****欢迎使用 野火i.MX RT1052 开发板*****
\ | /
- RT - Thread Operating System
/ | \ 3.0.3 build Aug 30 2019
2006 - 2018 Copyright by rt-thread team
这是一个[野火]-i.MX RT-RTT事件标志组实验！
事件创建成功！
KEY1被单击
KEY2被单击
Key1与Key2都按下
KEY1被单击
KEY1被单击
KEY2被单击
Key1与Key2都按下

第 25 章　软件定时器

25.1 软件定时器的基本概念

定时器，是指从指定的时刻开始，经过一个指定时间，然后触发一个超时事件，用户可以自定义定时器的周期与频率。类似生活中的闹钟，我们可以设置闹钟每天什么时候响，还能设置响的次数，是响一次还是每天都响。

定时器有硬件定时器和软件定时器之分：

硬件定时器是芯片本身提供的定时功能。一般是由外部晶振提供给芯片输入时钟，芯片向软件模块提供一组配置寄存器，接受控制输入，到达设定时间值后芯片中断控制器产生时钟中断。硬件定时器的精度一般很高，可以达到纳秒级别，并且是中断触发方式。

软件定时器，软件定时器是由操作系统提供的一类系统接口，它构建在硬件定时器基础之上，使系统能够提供不受硬件定时器资源限制的定时器服务，它实现的功能与硬件定时器也是类似的。

使用硬件定时器时，每次在定时时间到达之后就会自动触发一个中断，用户在中断中处理信息；而使用软件定时器时，需要我们在创建软件定时器时指定时间到达后要调用的函数（也称超时函数/回调函数，为了统一，下文均用超时函数描述），在超时函数中处理信息。

软件定时器在被创建之后，当经过设定的时钟计数值后会触发用户定义的超时函数。定时精度与系统时钟的周期有关。一般系统利用 SysTick 作为软件定时器的基础时钟，超时函数类似硬件的中断服务函数，所以，超时函数也要快进快出，而且超时函数中不能有任何阻塞线程运行的情况，比如 rt_thread_delay() 以及其它能阻塞线程运行的函数，两次触发超时函数的时间间隔 Tick 叫定时器的定时周期。

RT-Thread 操作系统提供软件定时器功能，软件定时器的使用相当于扩展了定时器的数量，允许创建更多的定时业务。RT-Thread 软件定时器功能上支持：

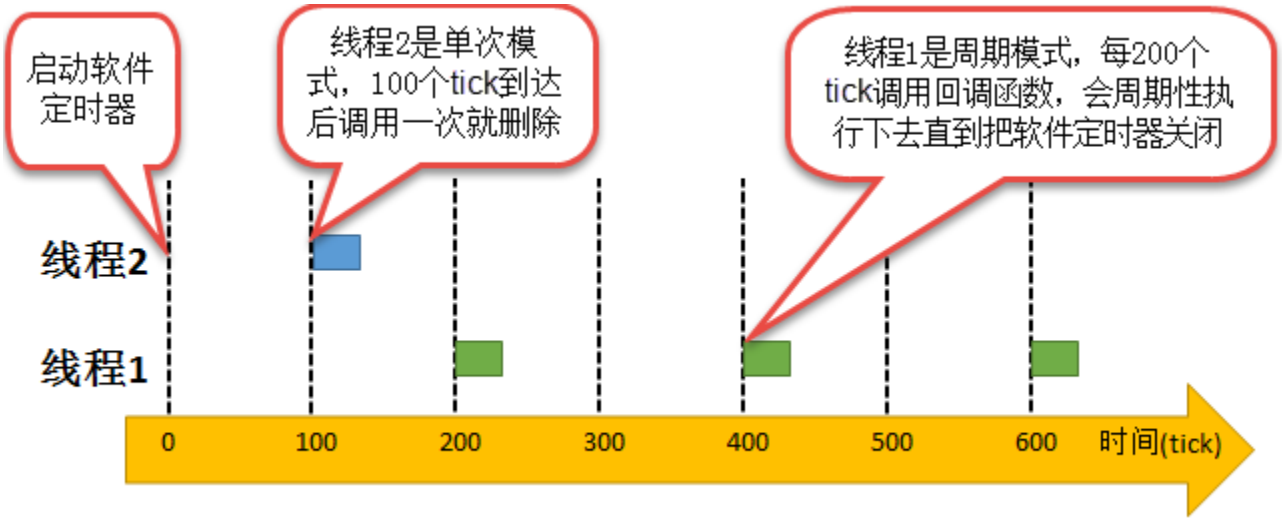
- 静态裁剪：能通过宏关闭软件定时器功能。
- 软件定时器创建。

- 软件定时器启动。
- 软件定时器停止。
- 软件定时器删除。

RT-Thread 提供的软件定时器支持单次模式和周期模式，单次模式和周期模式的定时时间到之后都会调用定时器的超时函数，用户可以在超时函数中加入要执行的工程代码。

单次模式：当用户创建了定时器并启动了定时器后，定时时间到了，只执行一次超时函数之后就将该定时器删除，不再重新执行。

周期模式：这个定时器会按照设置的定时时间循环执行超时函数，直到用户将定时器删除，具体见图 *22-1*。



注意： 在 RT-Thread 中创建定时器 API 接口可以选择软件定时器与硬件定时器，但是硬件定时器超时函数的上下文环境中断，而软件定时器超时函数的上下文是线程。下文所说的定时器均为软件定时器工作模式，RT-Thread 中在 rtdef.h 中定义了相关的宏定义来选择定时器的工作模式：

- RT_TIMER_FLAG_HARD_TIMER 为硬件定时器。
- RT_TIMER_FLAG_SOFT_TIMER 为软件定时器。

25.2 软件定时器的应用场景

在很多应用中，我们需要一些定时器线程，硬件定时器受硬件的限制，数量上不足以满足用户的实际需求，无法提供更多的定时器，那么可以采用软件定时器来完成，由软件定时器代替硬件定时器线程。但需要注意的是软件定时器的精度是无法和硬件定时器相比的，因为在软件定时器的定时过程中是极有可能被其它的线程所打断，因为软件定时器的线程优先级是 RT_TIMER_THREAD_PRIO，默认为 4。所以，软件定时器更适用于对时间精度要求不高的线程，一些辅助型的线程。

25.3 软件定时器的精度

在操作系统中，通常软件定时器以系统节拍周期为计时单位。系统节拍是系统的心跳节拍，表示系统时钟的频率，就类似人的心跳，1s 能跳动多少下，系统节拍配置为 RT_TICK_PER_SECOND，该宏在 rtconfig.h 中有定义，默认是 1000。那么系统的时钟节拍周期就为 1ms（1s 跳动 1000 下，每一下就为 1ms）。软件定时器的所定时数值必须是这个节拍周期的整数倍，例如节拍周期是 10ms，那么上层软件定时器定时数值只能是 10ms，20ms，100ms 等，而不能取值为 15ms。由于节拍定义了系统中定时器能够分辨的精确度，系统可以根据实际系统 CPU 的处理能力和实时性需求设置合适的数值，系统节拍周期的值越小，精度越高，但是系统开销也将越大，因为在 1 秒中系统进入时钟中断的次数也就越多。

25.4 软件定时器的运作机制

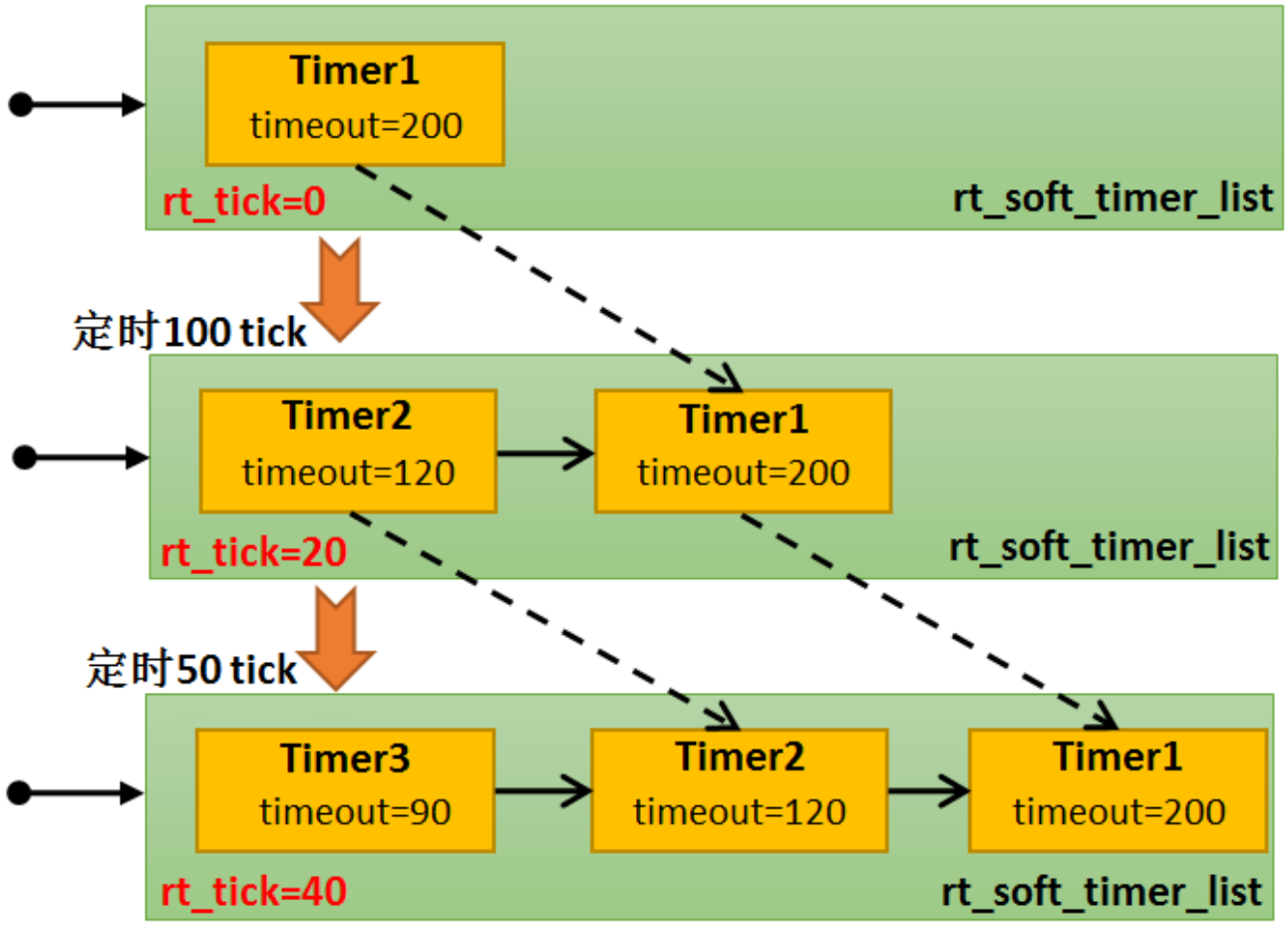
软件定时器是系统资源，在创建定时器的时候会分配一块内存空间。当用户创建并启动一个软件定时器时，RT-Thread 会根据当前系统 rt_tick 时间及用户设置的定时确定该定时器唤醒时间 timeout，并将该定时器控制块挂入软件定时器列表 rt_soft_timer_list。

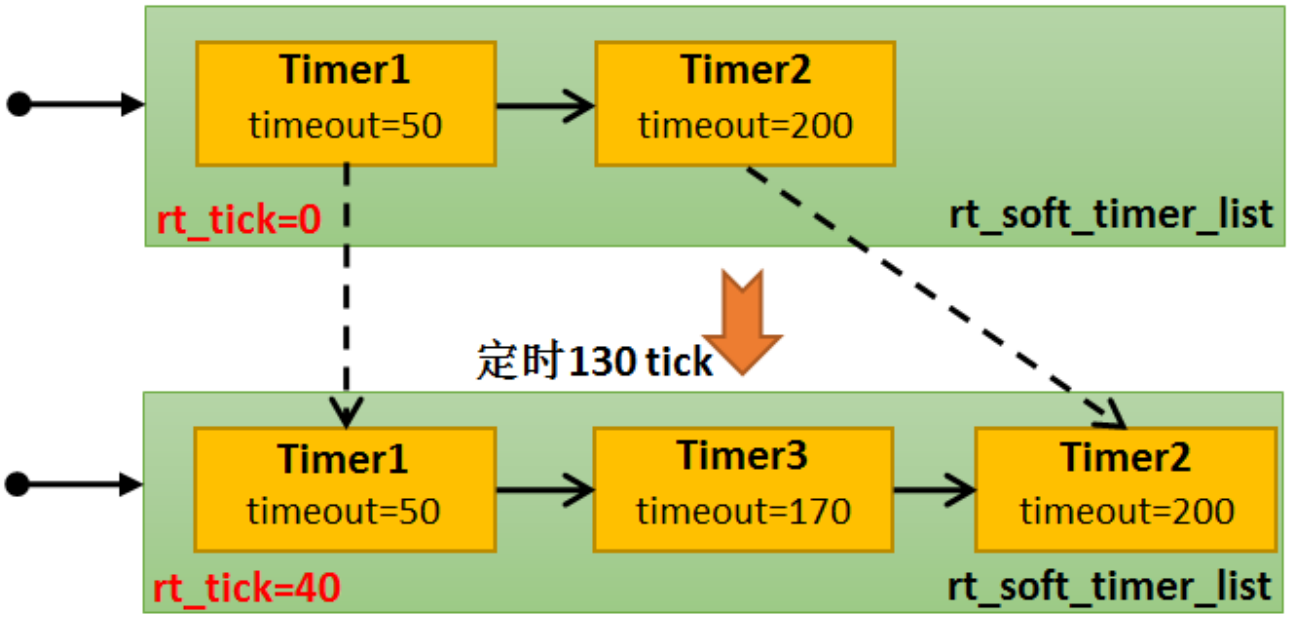
在 RT-Thread 定时器模块中维护着两个重要的全局变量：

- rt_tick，它是一个 32 位无符号的变量，用于记录当前系统经过的 tick 时间，当硬件定时器中断来临时，它将自动增加 1。

- 软件定时器列表 rt_soft_timer_list。系统新创建并激活的定时器都会以超时时间升序的方式插入到 rt_soft_timer_list 列表中。系统在定时器线程中扫描 rt_soft_timer_list 中的第一个定时器，看是否已超时，若已经超时了则调用软件定时器超时函数。否则出软件定时器线程，因为定时时间是升序插入软件定时器列表的，列表中第一个定时器的定时时间都还没到的话，那后面的定时器定时时间自然没到。

例如：系统当前时间 rt_tick 值为 0，在当前系统中已经创建并启动了 1 个定时器 Timer1；系统继续运行，当系统的时间 rt_tick 为 20 的时候，用户创建并且启动一个定时时间为 100 的定时器 Timer2，此时 Timer2 的溢出时间 timeout 就为定时时间 + 系统当前时间（100+20=120），然后将 Timer2 按 timeout 升序插入软件定时器列表中；假设当前系统时间 rt_tick 为 40 的时候，用户创建并且启动了一个定时时间为 50 的定时器 Timer3，那么此时 Timer3 的溢出时间 timeout 就为 40+50=90，同样安装 timeout 的数值升序插入软件定时器列表中，在定时器链表中插入过程具体见图 *22-2*。同理创建并且启动在已有的两个定时器中间的定时器也是一样的，具体见图 *22-3*。


Timer1
timeout=200
rt_tick=0
rt_soft_timer_list
定时100 tick
Timer2
timeout=120
Timer1
timeout=200
rt_tick=20
rt_soft_timer_list
定时50 tick
Timer3
timeout=90
Timer2
timeout=120
Timer1
timeout=200
rt_tick=40
rt_soft_timer_list



那么系统如何处理软件定时器列表？系统在不断运行，而 rt_tick 随着 SysTick 的触发一直在增长（每一次硬件定时器中断来临，rt_tick 变量会加 1），在软件定时器线程中扫描 rt_soft_timer_list，比较当前系统时间 rt_tick 是否大于或等于 timeout，若是则表示超时，定时器线程调用对应定时器的超时函数，否则退出软件定时器线程。以图 *22-3* 为例，讲解软件定时器调用超时函数的过程，在创建定 Timer1 并且启动后，假如系统经过了 50 个 tick，rt_tick 从 0 增长到 50，与 Timer1 的 timeout 值相等，这时会触发与 Timer1 对应的超时函数，从而转到超时函数中执行用户代码，同时将 Timer1 从 rt_timer_list 删除。同理，在 rt_tick=40 的时候创建的 Timer3，在经过 130 个 tick 后（此时系统时间 rt_tick 是 40，130 个 tick 就是系统时间 rt_tick 为 170 的时候），与 Timer3 定时器对应的超时函数会被触发，接着将 Timer3 从 rt_timer_list 删除。

使用软件定时器时候要注意以下几点：

- 软件定时器的超时函数中应快进快出，绝对不允许使用任何可能引软件定时器起线程挂起或者阻塞的 API 接口，在超时函数中也绝对不允许出现死循环。
- 软件定时器使用了系统的一个队列和一个线程资源，软件定时器线程的优先级默认为 RT_TIMER_THREAD_PRIO。
- 创建单次软件定时器，该定时器超时执行完超时函数后，系统会自动删除该软件定时器，并回收资源。

- 定时器线程的堆栈大小默认为 RT_TIMER_THREAD_STACK_SIZE，512 个字节。

25.5 定时器超时函数

定时器最主要的目的是在经过指定的定时时间后，系统能够自动执行用户设定的动作，也就是超时函数。该函数在创建定时器的时候需要用户自己定义，并且编写对应的执行代码。

在 RT-Thread 实时操作系统中，定时器超时函数存在着两种情况：

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

如果超时函数是在中断上下文环境中执行，显然对于超时函数的要求与中断服务例程的要求相同：执行时间应该尽量短，执行时不应导致当前上下文挂起、等待。例如在中断上下文中执行的超时函数它不应该试图去申请动态内存、释放动态内存等，也不允许调用 rt_thread_delay() 等导致上下文挂起的 API 接口，其具体的实现过程见*代码清单 22-1* 加粗部分。因为定时器超时函数包括软硬件定时器，所以此处仅对硬件定时器做简单两个介绍，本章节主要讲解软件定时器的实现。

列表 1: 代码清单 22-1 硬件定时器超时在 systick 的 isr 中的实现

```
void rt_tick_increase(void)
{
struct rt_thread *thread;

/* 系统时间全局变量自加 */
++ rt_tick;

/* 检查时间片 */
thread = rt_thread_self();

-- thread->remaining_tick;
```

（下页继续）

(续上页)

```
if (thread->remaining_tick == 0) {
    /* 更改为初始化的时间 */
    thread->remaining_tick = thread->init_tick;

    /* 强制切换 */
    rt_thread_yield();
}

/* 检查定时器时间 */
rt_timer_check();                                    (1)
}
```

代码清单 22-1 **(1)**：rt_timer_check() 是具体的检查定时器是否超时的函数。在第一部分的第 11 章有详细讲解此函数的实现过程，这里就不再赘述，具体见代码清单 11-13。

而软件定时器的超时函数在线程上下文中执行，则不会有这个限制，但是通常也要求超时函数执行时间应该足够短，不允许在超时函数中有阻塞的情况出现，更不允许有死循环，也不应该影响到其他定时器执行超时函数或本定时器的下一次超时回调。软件定时器的超时函数在线程中执行，下面一起来看看软件定时器超时函数是怎么样实现的。

我们知道，在 RT-Thread 启动的时候，会创建几个必要的线程，有 main_thread_entry 线程、rt_thread_idle_entry 线程、rt_thread_timer_entry 线程。rt_thread_timer_entry 是定时器线程，用于扫描软件定时器列表中是否有超时的定时器，然后执行其对应的超时函数，具体见代码清单 22-2。

列表 2: 代码清单 22-2 rt_thread_timer_entry 线程

```
/* system timer thread entry */
static void rt_thread_timer_entry(void *parameter)
{
rt_tick_t next_timeout;

while (1) {
    /* 获取软件定时器列表中下一个定时器的到达时间 */
```

(下页继续)

(续上页)

```
    next_timeout = rt_timer_list_next_timeout(rt_soft_timer_list);   (1)
    if (next_timeout == RT_TICK_MAX) {
            /* 如果没有软件定时器，则挂起线程自身 */
            rt_thread_suspend(rt_thread_self());                     (2)
            rt_schedule();
    } else {
            rt_tick_t current_tick;

            /* 获取当前系统时间 */
            current_tick = rt_tick_get();                            (3)

            if ((next_timeout - current_tick) < RT_TICK_MAX / 2) {   (4)
            /* 计算下一个定时器溢出时间与当前时间的间隔 */
            next_timeout = next_timeout - current_tick;              (5)
            rt_thread_delay(next_timeout);                           (6)
            }
    }

    /* 检查软件定时器列表 */
    rt_soft_timer_check();                                           (7)
}
}
#endif
```

代码清单 22-2 **(1)**：rt_thread_timer_entry 是一个线程，所以也是需要死循环的，线程在运行的时候扫描软件定时器列表获取下一个定时器定时到达的时间。

代码清单 22-2 **(2)**：如果此时软件定时器列表中没有软件定时器，就把线程自身挂起。因为软件定时器线程的运行是会占用 CPU 的，当没有开启软件定时器的时候就不要经常进入线程扫描，直接挂起线程即可。挂起自身之后要发起一次线程调度，让出 CPU。

代码清单 22-2 **(3)**：如果启动了软件定时器，那么就获取当前系统时间 current_tick。

代码清单 22-2 **(4)**：下一个定时器溢出时间与系统当前时间比较，如果时间还没到，执行 **(5) (6)** 。

代码清单 22-2 **(5)**：计算还有多长时间到达下一个定时器溢出的时间，记录在 next_timeout 中。

代码清单 22-2 **(6)**：将定时器线程延时 next_timeout，这样子做就不需要经常进入定时器线程查找定时器，直到下一个定时器需要唤醒的时候才进来处理，这样子大大提高 CPU 的利用率，这也是软件定时器为什么不够精确的原因，线程在唤醒的时候不一定能得到 CPU 的使用权。

代码清单 22-2 **(7)**：软件定时器扫描函数 rt_soft_timer_check()，其实现过程具体见代码清单 22-3。

列表 3: 代码清单 22-3 软件定时器扫描函数 rt_soft_timer_check()

```
void rt_soft_timer_check(void)
{
    rt_tick_t current_tick;
    rt_list_t *n;
    struct rt_timer *t;

    RT_DEBUG_LOG(RT_DEBUG_TIMER, ("software timer check enter\n"));

    current_tick = rt_tick_get();

    /* 锁定调度程序 */
    rt_enter_critical();

    for (n = rt_soft_timer_list[RT_TIMER_SKIP_LIST_LEVEL - 1].next;
        n != &(rt_soft_timer_list[RT_TIMER_SKIP_LIST_LEVEL - 1]);) {
        t = rt_list_entry(n, struct rt_timer, row[RT_TIMER_SKIP_LIST_LEVEL -
↪ 1]);

        /*
        * 判断是超时
        *
        */
        if ((current_tick - t->timeout_tick) < RT_TICK_MAX / 2) {    (1)
```

(下页继续)

(续上页)

```
            RT_OBJECT_HOOK_CALL(rt_timer_timeout_hook, (t));

            /* 移动节点到下一个 */
            n = n->next;

            /* 首先从定时器列表中删除定时器 */
            _rt_timer_remove(t);                                    (2)

            /* 执行超时功能时不锁定调度程序 */
            rt_exit_critical();
            /* 调用超时函数 */
            t->timeout_func(t->parameter);                          (3)

            /* 重新获取当前系统时间 tick */
            current_tick = rt_tick_get();                           (4)

            RT_DEBUG_LOG(RT_DEBUG_TIMER, ("current tick: %d\n", current_
↪tick));

            /* 锁定调度程序 */
            rt_enter_critical();

            if ((t->parent.flag & RT_TIMER_FLAG_PERIODIC) &&
                (t->parent.flag & RT_TIMER_FLAG_ACTIVATED)) {       (5)
                /* 开始，设置定时器状态为可用 */
                t->parent.flag &= ~RT_TIMER_FLAG_ACTIVATED;
                rt_timer_start(t);
            } else {
                /* 停止，设置定时器状态为不可用 */
                t->parent.flag &= ~RT_TIMER_FLAG_ACTIVATED;         (6)
            }
        } else break; /* 不再检查了 */                             (7)
```

(下页继续)

(续上页)

```
    }

    /* 解锁调度程序 */
    rt_exit_critical();

    RT_DEBUG_LOG(RT_DEBUG_TIMER, ("software timer check leave\n"));
}
```

代码清单 22-3 **(1)**：判断是系统时间是否到达定时器溢出时间。

代码清单 22-3 **(2)**：如果到达了定时器溢出时间，首先移动软件定时器列表的表头指针，指向下一个定时器，然后从软件定时器列表中删除当前时间溢出的定时器。

代码清单 22-3 **(3)**：执行定时器的超时函数。

代码清单 22-3 **(4)**：重新获取当前系统时间 current_tick。

代码清单 22-3 **(5)**：如果这个定时器是周期定时器的话，那么需要根据初始设置的定时时间重新加入定时器链表中，设置定时器状态为可用，然后调用启动定时器函数 rt_timer_start 将定时器重新添加到软件定时器列表中去，插入定时器列表会按定时器溢出时间 timeout 进行排序。

代码清单 22-3 **(6)**：如果软件定时器是单次模式的话，则将软件定时器设置为不可用状态。

代码清单 22-3 **(7)**：退出。

25.6 软件定时器的使用

由于在第一部分：第 11 章中已经详细讲解了定时器的函数接口与实现过程，现在不再赘述。直接讲解如何使用软件定时器。

25.6.1 软件定时器的创建函数

RT-Thread 给我们提供的只是一些基础函数，使用任何一个内核的资源都需要我们自己去创建，就像线程、信号量等这些 RT-Thread 的资源，所以，使用软件定时器也是需要我们自己去创建的，下面来看看软件定时器创建函数 rt_timer_create() 的源码，具体见代码清单 22-4。

列表 4: 代码清单 22-4 软件定时器的创建函数 rt_timer_create() 源码

```
rt_timer_t rt_timer_create(const char *name,                    (1)
                           void (*timeout)(void *parameter),    (2)
                           void        *parameter,              (3)
                           rt_tick_t    time,                   (4)
                           rt_uint8_t   flag)                   (5)
{
    struct rt_timer *timer;

    /* 分配定时器对象 */
    timer = (struct rt_timer *)rt_object_allocate(RT_Object_Class_Timer,␣
↪name);
    if (timer == RT_NULL) {                                     (6)
        return RT_NULL;
    }

    _rt_timer_init(timer, timeout, parameter, time, flag);      (7)

    return timer;                                               (8)
}
```

代码清单 22-4 **(1)**：定时器的名称，由用户自定义。

代码清单 22-4 **(2)**：定时器超时函数指针（当定时器超时时，系统会调用这个指针指向的函数），函数主体由用户自己实现。

代码清单 22-4 **(3)**：定时器超时函数的入口参数（当定时器超时时，调用超时函数会把这个参数

做为入口参数传递给超时函数)。

代码清单 22-4 **(4)**：定时器的超时时间，单位是 tick。

代码清单 22-4 **(5)**：定时器创建时的参数，支持的值具体见代码清单 22-5（可以用“或”关系取多个值，但是需要注意的是互斥关系的不能共用，同一个定时器不能是无效的又是可用的，不能既是硬件定时器的同时又是软件定时器），当指定的 flag 为 RT_IMER_FLAG_HARD_TIMER 时，如果定时器超时，定时器的超时函数将在中断中被调用；当指定的 flag 为 RT_TIMER_FLAG_SOFT_TIMER 时，如果定时器超时，定时器的超时函数将在线程中被调用。

列表 5: 代码清单 22-5 定时器创建时的参数（在 rtdef.h 文件中定义）

```
#define RT_TIMER_FLAG_DEACTIVATED        0x0     /**< 计时器是无效的 */
#define RT_TIMER_FLAG_ACTIVATED          0x1    /**< 定时器是可用的 */
#define RT_TIMER_FLAG_ONE_SHOT           0x0    /**< 单次定时器 */
#define RT_TIMER_FLAG_PERIODIC           0x2    /**< 周期定时器 */

#define RT_TIMER_FLAG_HARD_TIMER         0x0   /**< 硬定时器，定时器的超时函数将在tick isr 中调用。*/
#define RT_TIMER_FLAG_SOFT_TIMER         0x4   /**< 软定时器，定时器的超时函数将在定时器线程中调用。*/
```

代码清单 22-4 **(6)**：分配软件定时器对象，调用 rt_object_allocate 此函数将从对象系统分配对象，为创建的软件定时器分配一个软件定时器的对象，并且命名对象名称，在系统中，对象的名称必须是唯一的。

代码清单 22-4 **(7)**：调用 _rt_timer_init 初始化函数进行定时器的初始化，在第一部分中详细讲解过，此处不再赘述，具体见代码清单 11-6。

代码清单 22-4 **(8)**：如果定时器创建成功，则返回定时器的句柄，如果创建失败，会返回 RT_NULL（通常会由于系统内存不够用而返回 RT_NULL）。

软件定时器的创建函数使用起来是很简单的，软件定时器的超时函数需要自己实现，软件定时器的工作模式以及定时器的定时时间按需选择即可，具体见代码清单 22-6 加粗部分。

列表 6: 代码清单 22-6 软件定时器的创建函数 rt_timer_create() 实例

```
/* 创建一个软件定时器 */
swtmr1 = rt_timer_create("swtmr1_callback", /* 软件定时器的名称 */
                         swtmr1_callback,/* 软件定时器的超时函数 */
                         0,          /* 定时器超时函数的入口参数 */
                         5000,    /* 软件定时器的超时时间 (周期超时时间) */
                     RT_TIMER_FLAG_ONE_SHOT | RT_TIMER_FLAG_SOFT_TIMER);
                     /* 一次模式    软件定时器模式 */
/* 启动定时器 */
if (swtmr1 != RT_NULL)
    rt_timer_start(swtmr1);

/* 创建一个软件定时器 */
swtmr2 = rt_timer_create("swtmr2_callback",          /* 软件定时器的名称 */
                         swtmr2_callback,     /* 软件定时器的超时函数 */
                         0,          /* 定时器超时函数的入口参数 */
                         1000,    /* 软件定时器的超时时间 (周期超时时间) */
                     RT_TIMER_FLAG_PERIODIC | RT_TIMER_FLAG_SOFT_TIMER);
             /* 软件定时器模式 周期模式 */
/* 启动定时器 */
if (swtmr2 != RT_NULL)
    rt_timer_start(swtmr2);
```

软件定时器的其他相关函数均在第一部分的第 11 章中详细介绍了，现在就不再赘述，因为这些函数的实现都是一样的，只不过在第一部分中使用的是硬件定时器资源，在 systick 中断服务函数中实现定时器的扫描是否超时，而现在使用的是软件定时器资源，在定时器线程中扫描是否超时，原理都是一样的。

25.7 软件定时器实验

软件定时器实验是在 RT-Thread 中创建了两个软件定时器，其中一个软件定时器是单次模式，5000 个 tick 调用一次超时函数，另一个软件定时器是周期模式，1000 个 tick 调用一次超时函数，在超时函数中输出相关信息，具体见代码清单 22-7 加粗部分。

列表 7: 代码清单 22-7 软件定时器实验

```
/**
*****************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2019-xx-xx
* @brief   RT-Thread 软件定时器
*****************************************************************
* @attention
*
* 实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
******************************************************************
*/

/*
*******************************************************************
*                              包含的头文件
*******************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"
/* RT-Thread 头文件 */
```

(下页继续)

(续上页)

```
#include "rtthread.h"
/*
*****************************************************************
*                                 变量
*****************************************************************
*/
/* 定义线软件定时器制块 */
static rt_timer_t swtmr1 = RT_NULL;
static rt_timer_t swtmr2 = RT_NULL;
/************************* 全局变量声明 ****************************/
/*
* 当我们在写应用程序的时候，可能需要用到一些全局变量。
*/
static uint32_t TmrCb_Count1 = 0;
static uint32_t TmrCb_Count2 = 0;

/*
*********************************************************************
*                                函数声明
*********************************************************************
*/
static void swtmr1_callback(void* parameter);
static void swtmr2_callback(void* parameter);

/*
*********************************************************************
*                               main 函数
*********************************************************************
*/
/**
* @brief  主函数
* @param  无
```

(下页继续)

(续上页)

```
* @retval 无
*/
int main(void)
{

    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]- i.MX RT 全系列开发板-RTT 软件定时器实验！\n");
    rt_kprintf("定时器超时函数 1 只执行一次就被销毁\n");
    rt_kprintf("定时器超时函数 2 则循环执行\n");
    /* 创建一个软件定时器 */
    swtmr1 = rt_timer_create("swtmr1_callback", /* 软件定时器的名称 */
                             swtmr1_callback,/* 软件定时器的超时函数 */
                             0,       /* 定时器超时函数的入口参数 */
                             5000,    /* 软件定时器的超时时间 (周期超时时间) */
                     RT_TIMER_FLAG_ONE_SHOT | RT_TIMER_FLAG_SOFT_TIMER);
            /* 软件定时器模式 一次模式 */
    /* 启动定时器 */
    if (swtmr1 != RT_NULL)
        rt_timer_start(swtmr1);

    /* 创建一个软件定时器 */
    swtmr2 = rt_timer_create("swtmr2_callback", /* 软件定时器的名称 */
                             swtmr2_callback,/* 软件定时器的超时函数 */
                             0,              /* 定时器超时函数的入口参数

                             1000,    /* 软件定时器的超时时间 (周期超时时间) */
                     RT_TIMER_FLAG_PERIODIC | RT_TIMER_FLAG_SOFT_TIMER);
            /* 软件定时器模式 周期模式 */
```

(下页继续)

(续上页)

```
    /* 启动定时器 */
    if (swtmr2 != RT_NULL)
        rt_timer_start(swtmr2);
}

/*
*************************************************************************
*                                线程定义
*************************************************************************
*/

static void swtmr1_callback(void* parameter)
{
    uint32_t tick_num1;

    TmrCb_Count1++;                          /* 每调用一次加一 */

    tick_num1 = (uint32_t)rt_tick_get();     /* 获取滴答定时器的计数值 */

    rt_kprintf("swtmr1_callback 函数执行 %d 次\n", TmrCb_Count1);
    rt_kprintf("滴答定时器数值 =%d\n", tick_num1);
}

static void swtmr2_callback(void* parameter)
{
    uint32_t tick_num2;

    TmrCb_Count2++;                          /* 每调用一次加一 */

    tick_num2 = (uint32_t)rt_tick_get();     /* 获取滴答定时器的计数值 */

    rt_kprintf("swtmr2_callback 函数执行 %d 次\n", TmrCb_Count2);
```

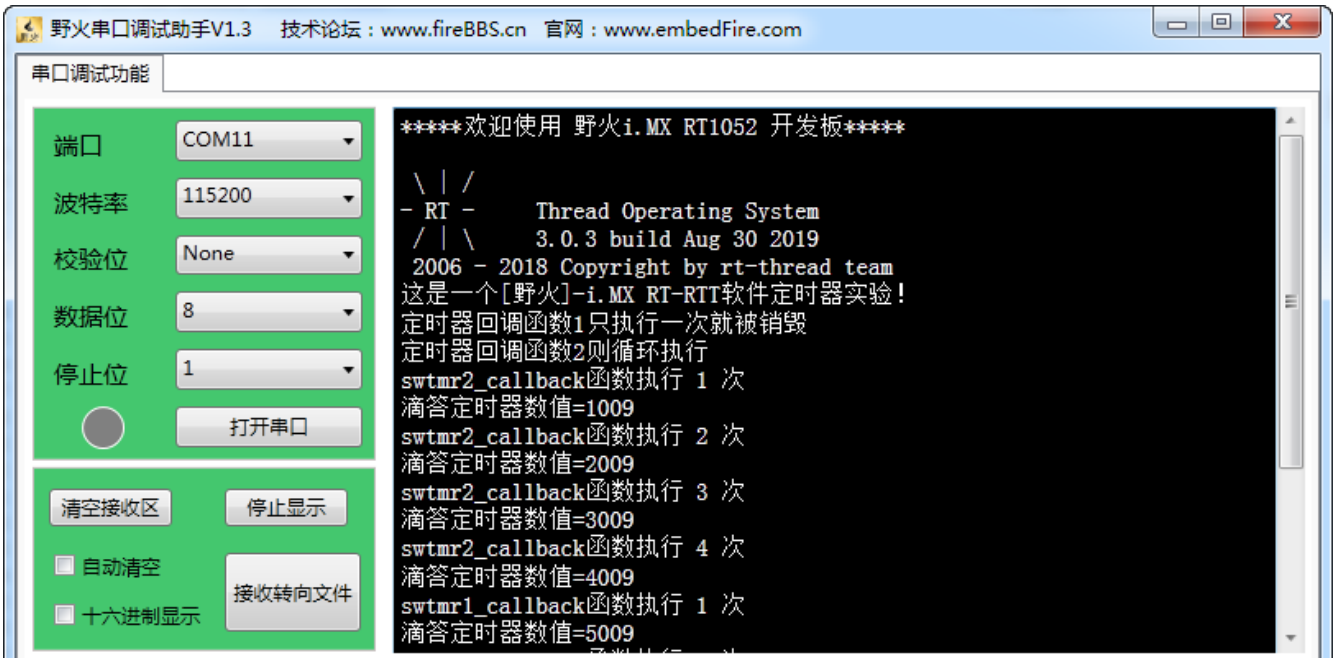
(下页继续)

(续上页)

```
    rt_kprintf("滴答定时器数值 =%d\n", tick_num2);
}

/****************************END OF FILE****************************/
```

25.8 实验现象

程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息，在串口调试助手中可以看到运行结果我们可以看到，每 1000 个 tick 时候软件定时器就会触发一次超时函数，当 5000 个 tick 到来的时候，触发软件定时器单次模式的超时函数，之后便不会再次调用了，具体见图 *22-4*。



第 26 章　邮箱

26.1 邮箱的基本概念

邮箱在操作系统中是一种常用的 IPC 通信方式，邮箱可以在线程与线程之间。中断与线程之间进行消息的传递，此外，邮箱相比于信号量与消息队列来说，其开销更低，效率更高，所以常用来做线程与线程、中断与线程间的通信。邮箱中的每一封邮件只能容纳固定的 4 字节内容（i.MX RT 是 32 位处理系统，一个指针的大小即为 4 个字节，所以一封邮件恰好能够容纳一个指针），当需要在线程间传递比较大的消息时，可以把指向一个缓冲区的指针作为邮件发送到邮箱中。

线程能够从邮箱里面读取邮件消息，当邮箱中的邮件是空时，根据用户自定义的阻塞时间决定是否挂起读取线程；当邮箱中有新邮件时，挂起的读取线程被唤醒，邮箱也是一种异步的通信方式。

通过邮箱，线程或中断服务函数可以将一个或多个邮件放入邮箱中。同样，一个或多个线程可以从邮箱中获得邮件消息。当有多个邮件发送到邮箱时，通常应将先进入邮箱的邮件先传给线程，也就是说，线程先得到的是最先进入邮箱的消息，即先进先出原则 (FIFO)，同时 RT-Thread 中的邮箱支持优先级，也就是说在所有等待邮件的线程中优先级最高的会先获得邮件。

RT-Thread 中使用邮箱实现线程异步通信工作，具有如下特性：

- 邮件支持先进先出方式排队与优先级排队方式，支持异步读写工作方式。
- 发送与接收邮件均支持超时机制。
- 一个线程能够从任意一个消息队列接收和发送邮件。
- 多个线程能够向同一个邮箱发送邮件和从中接收邮件。
- 邮箱中的每一封邮件只能容纳固定的 4 字节内容（可以存放地址）。
- 当队列使用结束后，需要通过删除邮箱以释放内存。

邮箱与消息队列很相似，消息队列中消息的长度是可以由用户配置的，但邮箱中邮件的大小却只能是固定容纳 4 字节的内容，所以，使用邮箱的开销是很小的，因为传递的只能是 4 字节以内的

内容，那么其效率会更高。

26.2 邮箱的运作机制

创建邮箱对象时会先创建一个邮箱对象控制块，然后给邮箱分配一块内存空间用来存放邮件，这块内存的大小等于邮件大小（4 字节）与邮箱容量的乘积，接着初始化接收邮件和发送邮件在邮箱中的偏移量，接着再初始化消息队列，此时消息队列为空。

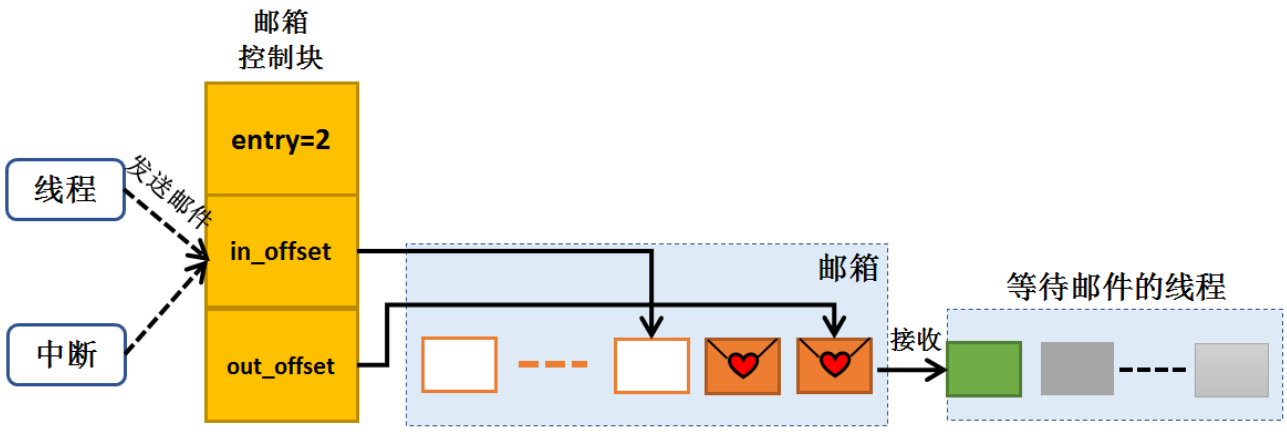
RT-Thread 操作系统的邮箱对象由多个元素组成，当邮箱被创建时，它就被分配了邮箱控制块：邮箱名称、邮箱缓冲区起始地址、邮箱大小等。同时每个邮箱对象中包含着多个邮件框，每个邮件框可以存放一封邮件；所有邮箱中的邮件框总数即是邮箱的大小，这个大小可在邮箱创建时指定。

线程或者中断服务程序都可以给邮箱发送邮件，非阻塞方式的邮件发送过程能够安全的应用于中断服务中，中断服务函数、定时器向线程发送消息的有效手段，而阻塞方式的邮件发送只能应用于线程中。当发送邮件时，当且仅当邮箱还没满邮件的时候才能进行发送，如果邮箱已满，可以根据用户设定的等待时间进行等待，当邮箱中的邮件被收取而空出空间来时，等待挂起的发送线程将被唤醒继续发送的过程，当等待时间到了还未完成发送邮件，或者未设置等待时间，此时发送邮件失败，发送邮件的线程或者中断程序会收到一个错误码（-RT_EFULL）。线程发送邮件可以带阻塞，但在中断中不能采用任何带阻塞的方式发送邮件。

接收邮件时，根据邮箱控制块中的 entry 判断队列是否有邮件，如果邮箱的邮件非空，那么可以根据 out_offset 找到最先发送到邮箱中的邮件进行接收。在接收时如果邮箱为空，如果用户设置了等待超时时间，系统会将当前线程挂起，当达到设置的超时时间，邮箱依然未收到邮件时，那么线程将被唤醒并返回-RT_ETIMEOUT。如果邮箱中存在邮件，那么接收线程将复制邮箱中的 4 个字节邮件到接收线程中。通常来说，邮件收取过程可能是阻塞的，这取决于邮箱中是否有邮件，以及收取邮件时设置的超时时间。

当消邮箱不再被使用时，应该删除它以释放系统资源，一旦操作完成，邮箱将被永久性的删除。

邮箱的运作机制具体见图 *23-1*。



26.3 邮箱的应用场景

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

与系统其它通信方式相比，邮箱的通信开销更低，效率更高，无论是什么消息，传递的都是 4 个字节的邮件，所以经常应用在众多领域，另外其实现的发送/接收阻塞机制，能很好应用于线程与线程，中断与线程之间的通讯。

其实邮箱中每封邮件的大小为 4 个字节，在 32 位系统中，刚好能存放一个指针，所以，邮箱也特别适合那种仅传递地址情况。

26.4 邮箱的应用技巧

可能很多人会问，在实际应用中，有很多结构体，那怎么用邮箱进行传递呢？其实这个很简单的，只是一个指针的传递，具体见*代码清单 23-1*。

列表 1: 代码清单 23-1 例子—邮箱传递的结构体

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

对于这样一个消息结构体，其中包含了指向数据的指针 data_ptr 和数据块长度的变量 data_size。当一个线程需要把这个消息发送给另外一个线程时，可以采用如下的操作，具体见代码清单 23-2。

列表 2: 代码清单 23-2 对结构体进行发送操作

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

申请结构体大小的内存空间，返回的指针指向了结构体，当结构体中的信息处理完，那么可以将指向结构体的指针作为邮件发送到邮箱中，而在接收邮件的线程中完成对结构体信息的读取操作，在完成操作后应当释放内存，因为收取过来的是指针，而 msg_ptr 是一个新分配出来的内存块，所以在接收线程处理完毕后，需要释放相应的内存块，具体见代码清单 23-3。

列表 3: 代码清单 23-3 对结构体进行接收操作

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

26.5 邮箱控制块

邮箱控制块包含了每个使用中邮箱的所有信息，如邮箱名称、内存缓冲区、邮箱大小以及队邮箱中邮件的数量等，是邮箱的很重要的控制块，具体见*代码清单 23-4*。

列表 4: 代码清单 23-4 邮箱控制块

```
struct rt_mailbox {
    struct rt_ipc_object parent;                          (1)

    rt_uint32_t         *msg_pool;                        (2)

    rt_uint16_t          size;                            (3)

    rt_uint16_t          entry;                           (4)
    rt_uint16_t          in_offset;                       (5)
    rt_uint16_t          out_offset;                      (6)

    rt_list_t            suspend_sender_thread;           (7)
};
typedef struct rt_mailbox *rt_mailbox_t;
```

代码清单 23-4 **(1)**：邮箱属于内核对象，也会在自身结构体里面包含一个内核对象类型的成员，通

过这个成员可以将邮箱挂到系统对象容器里面。

代码清单 23-4 **(2)**：邮箱缓冲区的开始地址。

代码清单 23-4 **(3)**：邮箱缓冲区的大小，也就是邮箱的大小，它的大小决定了能存放多少封 4 字节大小的邮件。

代码清单 23-4 **(4)**：邮箱中当前邮件的数目。

代码清单 23-4 **(5)**：邮箱邮件的进偏移指针，指向空的邮件。

代码清单 23-4 **(6)**：邮箱邮件的出偏移指针，如果邮箱中有邮件，则指向先进来的邮件。

代码清单 23-4 **(7)**：发送线程的挂起等待链表。

26.6 邮箱的函数接口讲解

26.6.1 邮箱创建函数 rt_mb_create()

邮箱创建函数，顾名思义，就是创建一个邮箱，与消息队列一样，都是需要先创建才能使用的内核资源，我们需要怎么样的邮箱我们就自己创建就行了，邮箱的大小，邮箱的名称这些信息都是我们自己定义的，RT-Thread 提供给我们这个创建函数，爱怎么搞都是我们自己来决定的。

创建邮箱对象时会先创建一个邮箱对象控制块，然后给邮箱分配一块内存空间用来存放邮件，这块内存的大小等于邮件大小（4 字节）与邮箱容量的乘积，接着初始化接收邮件和发送邮件在邮箱中的偏移量。创建邮箱的函数源码具体见代码清单 23-5。

列表 5: 代码清单 23-5 邮箱创建函数 rt_mb_create() 源码

```
rt_mailbox_t rt_mb_create(const char *name,                    (1)
                          rt_size_t size,                      (2)
                          rt_uint8_t flag)                     (3)
{
    rt_mailbox_t mb;

```

(下页继续)

(续上页)

```
    RT_DEBUG_NOT_IN_INTERRUPT;

    /* 分配邮箱对象 */
    mb = (rt_mailbox_t)rt_object_allocate(RT_Object_Class_MailBox, name);
    if (mb == RT_NULL)                                          (4)
        return mb;

    /* 设置接收线程等待模式 */
    mb->parent.parent.flag = flag;                              (5)

    /* 初始化邮箱对象 */
    rt_ipc_object_init(&(mb->parent));                          (6)

    /* 初始化邮箱 */
    mb->size     = size;                                        (7)
    mb->msg_pool = RT_KERNEL_MALLOC(mb->size * sizeof(rt_uint32_t));
    if (mb->msg_pool == RT_NULL) {                              (8)
        /* 删除邮箱对象 */
        rt_object_delete(&(mb->parent.parent));                 (9)

        return RT_NULL;
    }
    mb->entry      = 0;                                         (10)
    mb->in_offset  = 0;
    mb->out_offset = 0;

    /* 初始化发送邮件挂起线程的链表 */
    rt_list_init(&(mb->suspend_sender_thread));                 (11)

    return mb;
}
RTM_EXPORT(rt_mb_create);
```

代码清单 23-5 **(1)**：name 邮箱名称。

代码清单 23-5 **(2)**：size 邮箱容量，这个邮箱能存放多少封邮件。

代码清单 23-5 **(3)**：flag 用于设置邮箱的阻塞唤醒模式。

代码清单 23-5 **(4)**：分配邮箱对象，调用 rt_object_allocate 此函数将从对象系统分配对象，为创建的邮箱分配一个邮箱的对象，并且命名对象名称，在系统中，对象的名称必须是唯一的。

代码清单 23-5 **(5)**：设置邮箱的阻塞唤醒模式，创建的邮箱由于指定的 flag 不同，而有不同的意义：使用 RT_IPC_FLAG_PRIO 优先级 flag 创建的 IPC 对象，在多个线程等待资源时，将由优先级高的线程优先获得资源。而使用 RT_IPC_FLAG_FIFO 先进先出 flag 创建的 IPC 对象，在多个线程等待资源时，将按照先来先得的顺序获得资源。RT_IPC_FLAG_PRIO 与 RT_IPC_FLAG_FIFO 均在 rtdef.h 中有定义。

代码清单 23-5 **(6)**：初始化邮箱内核对象。调用 rt_ipc_object_init 会初始化一个链表用于记录访问此事件而阻塞的线程。

代码清单 23-5 **(7)**：初始化邮箱，设置邮箱的大小。

代码清单 23-5 **(8)**：申请邮箱内存，其内存大小为邮箱容量乘以 4 个字节，因为每封邮件的大小为 4 个字节。

代码清单 23-5 **(9)**：如果内存申请失败，则需要删除邮箱对象。

代码清单 23-5 **(10)**：申请内存成功，则初始化相关信息，将当前邮件的数量清零，邮件的进出偏移指针也为 0。

代码清单 23-5 **(11)**：初始化发送邮件挂起线程的链表。

在创建邮箱的时候，是需要用户自己定义邮箱的句柄的，但是注意了，定义了邮箱的句柄并不等于创建了邮箱，创建邮箱必须是调用 rt_mb_create() 函数进行创建，否则，以后根据邮箱句柄使用邮箱的其它函数的时候都会发生错误，在创建邮箱的时候是会返回创建的情况的，如果创建成功则返回创建的邮箱句柄，如果是返回 RT_NULL，则表示失败，邮箱创建函数 rt_mb_create() 使用实例具体见代码清单 23-6 高亮部分。

列表 6: 代码清单 23-6 邮箱创建函数 rt_mb_create() 实例

```
/* 定义邮箱控制块 */
static rt_mailbox_t test_mail = RT_NULL;

/* 创建一个邮箱 */
test_mail = rt_mb_create("test_mail", /* 消息队列名字 */
                         10,                  /* 邮箱大小 */
                         RT_IPC_FLAG_FIFO);/* 信号量模式 FIFO(0x00)*/
if (test_mail != RT_NULL)
rt_kprintf("邮箱创建成功! \n\n");
```

26.6.2 邮箱删除函数 rt_mb_delete()

在不想用邮箱的时候，想要删除邮箱怎么办呢？RT-Thread 给我们提供了一个删除邮箱的函数——rt_mb_delete()，使用它就能将邮箱进行删除了。当系统不再使用邮箱对象时，可以通过删除邮箱对象控制块来释放系统资源，一旦操作完成，邮箱将被永久性的删除，具体见*代码清单 23-7*。

列表 7: 代码清单 23-7 邮箱删除函数 rt_mb_delete() 源码

```
rt_err_t rt_mb_delete(rt_mailbox_t mb)                          (1)
{
    RT_DEBUG_NOT_IN_INTERRUPT;

    /* 邮箱句柄检查 */
    RT_ASSERT(mb != RT_NULL);                                   (2)

    /* 恢复所有阻塞在接收邮件的线程 */
    rt_ipc_list_resume_all(&(mb->parent.suspend_thread));       (3)

    /* 也恢复所有阻塞在发送邮件的线程  */
```

(下页继续)

(续上页)

```
    rt_ipc_list_resume_all(&(mb->suspend_sender_thread));    (4)

#if defined(RT_USING_MODULE) && defined(RT_USING_SLAB)       (5)
    /* 邮箱对象属于应用程序模块 */
    if (mb->parent.parent.flag & RT_OBJECT_FLAG_MODULE)
        rt_module_free(mb->parent.parent.module_id, mb->msg_pool);
    else
#endif

        /* 释放邮箱内存 */
        RT_KERNEL_FREE(mb->msg_pool);                         (6)

    /* 删除邮箱对象 */
    rt_object_delete(&(mb->parent.parent));                   (7)

    return RT_EOK;
}
RTM_EXPORT(rt_mb_delete);
```

代码清单 23-7 **(1)**：mb 是我们自己定义的邮箱句柄，删除哪个邮箱就把该邮箱句柄传进来即可。

代码清单 23-7 **(2)**：检查邮箱句柄 mb 是否有效，如果它是未定义或者未创建的邮箱句柄，那么是无法进行删除操作的。

代码清单 23-7 **(3)**：调用 rt_ipc_list_resume_all() 函数将所有因为接收不到邮件的而阻塞的线程从阻塞态中唤醒，所有被唤醒的线程的返回值是-RT_ERROR。与所有对象资源的删除函数一样，我们一般不会直接就删除一个邮箱，所以在删除邮箱的时候，应先确认所有的线程都无需接收邮件，并且都没被此邮箱阻塞时候才进行删除，否则删除之后线程需要发送/接收此邮箱邮件的话那也会发生错误。

代码清单 23-7 **(4)**：同理，也应该调用 rt_ipc_list_resume_all() 函数将所有因为邮箱满了发送不到邮件的而阻塞的线程从阻塞态中恢复过来，所有被唤醒的线程的返回值是-RT_ERROR。

代码清单 23-7 **(5)**：如果使能了 RT_USING_SLAB 这个宏定义，表示使用 slab 分配内存机制，那

么需要使用 rt_module_free 函数进行释放内存，在这里我们并未使用 slab。

代码清单 23-7 **(6)**：释放邮箱内存，一旦释放，将永久性被删除。

代码清单 23-7 **(7)**：删除邮箱对象。

邮箱的删除函数使用是很简单的，只需要传递进我们创建的邮箱对象句柄，其使用方法具体见代码清单 23-8 高亮部分。

列表 8: 代码清单 23-8 邮箱删除函数 rt_mb_delete() 实例

```
/* 定义邮箱控制块 */
static rt_mailbox_t test_mail = RT_NULL;
rt_err_t uwRet = RT_EOK;

/* 删除一个邮箱 */
uwRet = rt_mbt_delete(test_mail);
if (RT_EOK == uwRet)
    rt_kprintf("邮箱创建成功！\n\n");
```

26.6.3 邮箱邮件发送函数 rt_mb_send_wait()（阻塞）

邮箱的邮件发送可以从线程发送到线程，当发送邮件时候，邮箱发送的邮件可以是 4 字节以内任意格式的数据或者是一个指向缓冲区的指针。当且仅当邮箱还未满时，发送者才能成功发送邮件；当邮箱中的邮件已经满时，用户可以设置阻塞时间，进行发送邮件等待，当邮箱为满的时候将发送邮件线程挂起指定时间，当发送超时的时候，发送邮件的线程会收到一个错误代码-RT_EFULL，表示发送邮件失败，邮箱发送邮件函数 rt_mb_send_wait() 源码具体见代码清单 23-9。

列表 9: 代码清单 23-9 邮箱邮件发送函数 rt_mb_send_wait()（阻塞）源码

```
/**
* 如果这个邮箱对象是空的话，这个函数会发送一个邮件到邮箱对象.
* 如果这个邮箱对象是满的话，将会挂起当前线程
```

(下页继续)

(续上页)

```
 *
 * @param 邮箱对象
 * @param 邮箱大小
 * @param 等待时间
 *
 * @return 错误代码
 */
rt_err_t rt_mb_send_wait(rt_mailbox_t mb,                    (1)
                         rt_uint32_t  value,                 (2)
                         rt_int32_t   timeout)               (3)
{
    struct rt_thread *thread;
    register rt_ubase_t temp;
    rt_uint32_t tick_delta;

    /* 检查邮箱对象 */
    RT_ASSERT(mb != RT_NULL);                                (4)

    /* 初始化系统时间差 */
    tick_delta = 0;
    /* 获取当前线程 */
    thread = rt_thread_self();                               (5)

    RT_OBJECT_HOOK_CALL(rt_object_put_hook, (&(mb->parent.parent)));

    /* 关中断 */
    temp = rt_hw_interrupt_disable();

    /* 无阻塞调用 */
    if (mb->entry == mb->size && timeout == 0) {             (6)
        rt_hw_interrupt_enable(temp);

```

(下页继续)

(续上页)

```
        return -RT_EFULL;
    }

    /* 邮箱满了 */
    while (mb->entry == mb->size) {                          (7)
        /* 重置线程错误代码 */
        thread->error = RT_EOK;

        /* 不等待，返回错误 */
        if (timeout == 0) {                                  (8)
            /* 开中断 */
            rt_hw_interrupt_enable(temp);

            return -RT_EFULL;
        }

        RT_DEBUG_IN_THREAD_CONTEXT;
        /* 挂起当前线程 */
        rt_ipc_list_suspend(&(mb->suspend_sender_thread),    (9)
                            thread,
                            mb->parent.parent.flag);

        /* 有等待时间 */
        if (timeout > 0) {                                   (10)
            /* 获取当前系统时间 */
            tick_delta = rt_tick_get();

    RT_DEBUG_LOG(RT_DEBUG_IPC, ("mb_send_wait: start timer of thread:%s\n",
                                thread->name));

            /* 重置线程超时时间并开始定时 */
            rt_timer_control(&(thread->thread_timer),        (11)
```

(下页继续)

(续上页)

```
                                RT_TIMER_CTRL_SET_TIME,
                                &timeout);
            rt_timer_start(&(thread->thread_timer));          (12)
        }

        /* 开中断 */
        rt_hw_interrupt_enable(temp);

        /* 进行线程调度 */
        rt_schedule();                                        (13)

        /* 从挂起状态恢复 */
        if (thread->error != RT_EOK) {                        (14)
            /* 返回错误代码 */
            return thread->error;
        }

        /* 关中断 */
        temp = rt_hw_interrupt_disable();

        /* 如果它不是永远等待 */
        if (timeout > 0) {
            tick_delta = rt_tick_get() - tick_delta;
            timeout -= tick_delta;
            if (timeout < 0)
                timeout = 0;
        }
    }

    /* 将要发送的信息放入邮件中 */
    mb->msg_pool[mb->in_offset] = value;                      (15)
    /* 邮件进指针偏移 */
```

(下页继续)

(续上页)

```
    ++ mb->in_offset;                                          (16)
    if (mb->in_offset >= mb->size)                             (17)
        mb->in_offset = 0;
    /* 记录邮箱中邮件的数量 */
    mb->entry ++;                                              (18)

    /* 恢复线程 */
    if (!rt_list_isempty(&mb->parent.suspend_thread)) {        (19)
        rt_ipc_list_resume(&(mb->parent.suspend_thread));

        /* 开中断 */
        rt_hw_interrupt_enable(temp);

        rt_schedule();                                         (20)

        return RT_EOK;
    }

    /* 开中断 */
    rt_hw_interrupt_enable(temp);

    return RT_EOK;                                             (21)
}
RTM_EXPORT(rt_mb_send_wait);
```

代码清单 23-9 **(1)**：mb 邮箱对象的句柄。

代码清单 23-9 **(2)**：value 邮件内容，可以是 4 字节大小以内的任意内容，也可以是一个指针。

代码清单 23-9 **(3)**：timeout 超时时间。

代码清单 23-9 **(4)**：检查邮箱句柄 mb 是否有效，如果它是未定义或者未创建的邮箱句柄，那么是无法进行发送邮件操作的。

代码清单 23-9 **(5)**：先获取当前线程，在后面需要用到当前线程的信息。

代码清单 23-9 **(6)**：如果邮箱已满，并且是无阻塞调用（timeout=0），那么发送失败，直接退出发送。

代码清单 23-9 **(7)**：如果邮箱满了，进入死循环中。

代码清单 23-9 **(8)**：timeout=0，用户不等待，返回错误码。

代码清单 23-9 **(9)**：_ **(9)-(17)** 的内容都是邮箱满了并且 timeout 不为 0 的情况。因为用户设置了阻塞时间，不管三七二十一直接先将当前线程挂起。

代码清单 23-9 **(10)**：有等待时间（并非一直等待的情况，因为 RT_WAITING_FOREVER 的值为（-1），在 rtdef.h 中有定义），现在是设置了某个等待的时间。

代码清单 23-9 **(11)**：重置线程定时器的超时时间，调用 rt_timer_control() 函数改变当前线程阻塞时间 thread_timer。

代码清单 23-9 **(12)**：启动定时器，开始计时。

代码清单 23-9 **(13)**：因为现在线程是等待着了，要进行线程切换，所以需要进行一次线程调度。

代码清单 23-9 **(14)**：超时时间到了，线程被唤醒，但此时还没发送邮件完成，那么将返回错误码。

代码清单 23-9 **(15)**：如果邮箱还未满，那么可以将要发送的邮件放入邮箱。

代码清单 23-9 **(16)**：更新发送邮件指针的进偏移地址，因为邮箱是一个内存池，其存放邮件的地址在 32 位机器中指针下标加 1 偏移刚好是 4 个字节，指向了下一个空闲邮件地址。

代码清单 23-9 **(17)**：判断邮箱是否满了，若它满了，将 in_offset 设置为 0。

代码清单 23-9 **(18)**：记录邮箱中邮件的数量，邮箱控制块需要知道邮箱中邮件的实时数量。

代码清单 23-9 **(19)**：如果有线程因为接收不到邮件进入阻塞的话，那么需要恢复该线程，调用 rt_ipc_list_resume 函数将该线程恢复。

代码清单 23-9 **(20)**：恢复线程后进行一次线程调度。

代码清单 23-9 **(21)**：返回发送邮件结果。

发送邮件时，发送者需指定发送到的邮箱的对象句柄（即指向邮箱控制块的指针），并且指定发送的邮件内容，如果内容大于 4 个字节，可以将内容的地址作为邮件发送出去，邮箱发送邮件函

数 rt_mb_send_wait() 的实例具体见代码清单 23-10 高亮内容。

列表 10: 代码清单 23-10 邮箱邮件发送函数 rt_mb_send_wait()（阻塞）实例

```
/* 定义邮箱控制块 */
static rt_mailbox_t test_mail = RT_NULL;
/************************* 全局变量声明 ****************************/
/*
* 当我们在写应用程序的时候，可能需要用到一些全局变量。
*/
char test_str1[] = "this is a mail test 1";/* 邮箱消息 test1 */
char test_str2[] = "this is a mail test 2";/* 邮箱消息 test2 */

static void send_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        //如果 KEY1 被单击
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_GPIO_PIN) == KEY_ON ) {
            rt_kprintf ( "KEY1 被单击\n" );
            /* 发送一个邮箱消息 1 */
            uwRet = rt_mb_send_wait(test_mail,/* 邮箱对象句柄 */
                                    (rt_uint32_t)&test_str1,/* 邮件内容（地址）*/
                                    10); /* 超时时间 */
            if (RT_EOK == uwRet)
                rt_kprintf ( "邮箱消息发送成功\n" );
            else
                rt_kprintf ( "邮箱消息发送失败\n" );
        }
        //如果 KEY2 被单击
        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_GPIO_PIN) == KEY_ON ) {
```

(下页继续)

(续上页)

```
            rt_kprintf ( "KEY2 被单击\n" );
            /* 发送一个邮箱消息 2 */
            uwRet = rt_mb_send_wait(test_mail,/* 邮箱对象句柄 */
                            (rt_uint32_t)&test_str1,/* 邮件内容 (地址) */
                                10);  /* 超时时间 */
            if (RT_EOK == uwRet)
                rt_kprintf ( "邮箱消息发送成功\n" );
            else
                rt_kprintf ( "邮箱消息发送失败\n" );
        }
        rt_thread_delay(20);      //每 20ms 扫描一次
    }
}
```

发送的邮件可以是 4 字节任意格式的数据，当邮箱中的邮件已经满时，发送邮件的线程或者中断程序会收到-RT_EFULL 的返回值。

26.6.4 邮箱邮件发送函数 rt_mb_send ()（非阻塞）

RT-Thread 给我们提供了两个邮箱发送函数，一个是带阻塞的 rt_mb_send_wait()，另一个是非阻塞的 rt_mb_send()，那么这两个函数有什么不一样呢？其实，看了源码你就会知道，原来没啥差别，下面一起来看看 rt_mb_send ()（非阻塞）的源码，具体见*代码清单 23-11*。

列表 11: 代码清单 23-11 邮箱邮件发送函数 rt_mb_send ()（非阻塞）源码

```
/**
* 此函数将邮件发送到邮箱对象，
* 如果有邮件对象挂起，则会被唤醒。
* 此函数将立即返回，如果要阻塞发送，请改用 rt_mb_send_wait。
*
```

(下页继续)

(续上页)

```
* @param 邮箱对象
* @param 要发送的邮件内容
*
* @return 返回的错误码
*/
rt_err_t rt_mb_send(rt_mailbox_t mb, rt_uint32_t value)
{
    return rt_mb_send_wait(mb, value, 0);
}
RTM_EXPORT(rt_mb_send);
```

其实 rt_mb_send 真正调用的函数是 rt_mb_send_wait，但是它却是不等待的（因为 timeout=0），这个函数多用于中断与线程的通信，因为中断中不允许阻塞。而 rt_mb_send_wait() 却比较灵活，多用于线程与线程的通信。

既然 rt_mb_send() 函数源码实际上就是调用 rt_mb_send_wai()，连实现都是一样的，那么使用当然也是一样啦，只不过 rt_mb_send() 传递的参数少了一个 timeout 而已，具体实例见*代码清单 23-12* 高亮部分。

列表 12: 代码清单 23-12 邮箱邮件发送函数 rt_mb_send ()（非阻塞）实例

```
/* 定义邮箱控制块 */
static rt_mailbox_t test_mail = RT_NULL;
/************************ 全局变量声明 ***************************/
/*
* 当我们在写应用程序的时候，可能需要用到一些全局变量。
*/
char test_str1[] = "this is a mail test 1";/* 邮箱消息 test1 */
char test_str2[] = "this is a mail test 2";/* 邮箱消息 test2 */

static void send_thread_entry(void* parameter)
```

(下页继续)

(续上页)

```
{
    rt_err_t uwRet = RT_EOK;
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        //如果 KEY1 被单击
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_GPIO_PIN) == KEY_ON ) {
            rt_kprintf ( "KEY1 被单击\n" );
            /* 发送一个邮箱消息 1 */
            uwRet = rt_mb_send(test_mail,/* 邮箱对象句柄 */
                               (rt_uint32_t)&test_str1)/* 邮件内容（地址） */
                    if (RT_EOK == uwRet)
                        rt_kprintf ( "邮箱消息发送成功\n" );
            else
                rt_kprintf ( "邮箱消息发送失败\n" );
        }
        //如果 KEY2 被单击
        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_GPIO_PIN) == KEY_ON ) {
            rt_kprintf ( "KEY2 被单击\n" );
            /* 发送一个邮箱消息 2 */
            uwRet = rt_mb_send(test_mail,/* 邮箱对象句柄 */
                               (rt_uint32_t)&test_str1)/* 邮件内容（地址） */
                    if (RT_EOK == uwRet)
                        rt_kprintf ( "邮箱消息发送成功\n" );
            else
                rt_kprintf ( "邮箱消息发送失败\n" );
        }
        rt_thread_delay(20);      //每 20ms 扫描一次
    }
}
```

26.6.5 邮箱邮件接收函数 rt_mb_recv()

邮件的收发与我们现实生活中的邮件收发其实是一样的道理，既然别人给我们发了一份邮件，那么我们肯定要看看有什么事情发生，然后进行处理。在 RT-Thread 中，官方给我们提供了一个函数接口——邮箱的邮件接收函数 rt_mb_recv()，我们可以使用该函数访问指定的邮箱，看看是否有邮件发送过来，接收到邮件就去处理信息，如果还没有邮件发送过来，那我们可以不等这个邮件或者指定等待时间去接收这个邮件，如果超时了还是没有收到邮件，就返回错误代码。

只有当接收者接收的邮箱中有邮件时，接收线程才能立即取到邮件，否则接收线程会根据指定超时时间将线程挂起，直到接收完成或者超时，下面一起来看看邮件的接收函数，具体见*代码清单 23-13*。

列表 13: 代码清单 23-13 邮箱邮件接收函数 rt_mb_recv() 源码

```
rt_err_t rt_mb_recv(rt_mailbox_t mb,                          (1)
                    rt_uint32_t *value,                       (2)
                    rt_int32_t timeout)                       (3)
{
    struct rt_thread *thread;
    register rt_ubase_t temp;
    rt_uint32_t tick_delta;

    /* 邮箱检查 */
    RT_ASSERT(mb != RT_NULL);                                 (4)

    /* 初始化系统时间差变量 */
    tick_delta = 0;
    /* 获取当前线程 */
    thread = rt_thread_self();                                (5)

    RT_OBJECT_HOOK_CALL(rt_object_trytake_hook, (&(mb->parent.parent)));

```

(下页继续)

(续上页)

```
    /* 关中断 */
    temp = rt_hw_interrupt_disable();

    /* 非阻塞调用 */
    if (mb->entry == 0 && timeout == 0) {                    (6)
        rt_hw_interrupt_enable(temp);

        return -RT_ETIMEOUT;
    }

    /* 邮箱是空的 */
    while (mb->entry == 0) {                                 (7)
        /* 重置线程错误 */
        thread->error = RT_EOK;

        /* 不等待，返回错误码-RT_ETIMEOUT */
        if (timeout == 0) {
            /* 开中断 */
            rt_hw_interrupt_enable(temp);

            thread->error = -RT_ETIMEOUT;

            return -RT_ETIMEOUT;
        }

        RT_DEBUG_IN_THREAD_CONTEXT;
        /* 挂起当前线程 */
        rt_ipc_list_suspend(&(mb->parent.suspend_thread),    (8)
                            thread,
                            mb->parent.parent.flag);

        /* 有等待时间，开始等待 */
```

(下页继续)

(续上页)

```
        if (timeout > 0) {
            /* 获取开始时候的系统时间 */
            tick_delta = rt_tick_get();                    (9)

        RT_DEBUG_LOG(RT_DEBUG_IPC, ("mb_recv: start timer of thread:%s\n",
                                    thread->name));

            /* 重置线程超时时间，并且开始定时器 */
            rt_timer_control(&(thread->thread_timer),      (10)
                             RT_TIMER_CTRL_SET_TIME,
                             &timeout);
            rt_timer_start(&(thread->thread_timer));       (11)
        }

        /* 开中断 */
        rt_hw_interrupt_enable(temp);

        /* 发起线程调度 */
        rt_schedule();                                     (12)

        /* 解除阻塞了 */
        if (thread->error != RT_EOK) {
            /* 返回错误代码 */
            return thread->error;
        }

        /* 关中断 */
        temp = rt_hw_interrupt_disable();

    /* 如果它不是永远等待 */
        if (timeout > 0) {
            tick_delta = rt_tick_get() - tick_delta;
```

(下页继续)

(续上页)

```
            timeout -= tick_delta;
            if (timeout < 0)
                timeout = 0;
        }
    }

    /* 将邮件内容放到接收邮件的地址中 */
    *value = mb->msg_pool[mb->out_offset];                    (13)

    /* 接收邮件偏移指针自加 */
    ++ mb->out_offset;                                        (14)
    if (mb->out_offset >= mb->size)                           (15)
        mb->out_offset = 0;
    /* 记录当前邮件数量 */
    mb->entry --;                                             (16)

    /* 恢复挂起的线程 */
    if (!rt_list_isempty(&(mb->suspend_sender_thread))) {     (17)
        rt_ipc_list_resume(&(mb->suspend_sender_thread));

        /* 开中断 */
        rt_hw_interrupt_enable(temp);

        RT_OBJECT_HOOK_CALL(rt_object_take_hook, (&(mb->parent.parent)));

        rt_schedule();                                        (18)

        return RT_EOK;
    }

    /* 关中断 */
    rt_hw_interrupt_enable(temp);
```

(下页继续)

(续上页)

```

    RT_OBJECT_HOOK_CALL(rt_object_take_hook, (&(mb->parent.parent)));

    return RT_EOK;                                                  (19)
}
RTM_EXPORT(rt_mb_recv);
```

代码清单 23-13 **(1)**：mb 邮箱对象的句柄。

代码清单 23-13 **(2)**：value 用于存放邮件内容的地址，在调用接收函数前需要用户自己定义一个用于保存数据的变量，并且将该变量的地址作为参数传递进来。

代码清单 23-13 **(3)**：timeout 超时时间。

代码清单 23-13 **(4)**：检查邮箱句柄 mb 是否有效，如果它是未定义或者未创建的邮箱句柄，那么是无法进行接收邮件操作的。

代码清单 23-13 **(5)**：先获取当前线程，在后面需要用到当前线程的信息。

代码清单 23-13 **(6)**：如果邮箱是空的，并且是无阻塞调用（timeout=0）接收函数，那么接收邮件失败。

代码清单 23-13 **(7)**：如果邮箱是空的，进入死循环中。

代码清单 23-13 **(8)**：_ **(8)-(12)** 的内容都是邮箱是空的并且 timeout 不为 0 的情况。因为用户设置了阻塞时间，不管三七二十一直接先将当前线程挂起。

代码清单 23-13 **(9)**：获取阻塞开始时候的系统时间。

代码清单 23-13 **(10)**：重置线程计时器的超时时间，调用 rt_timer_control() 函数改变当前线程阻塞时间 thread_timer。

代码清单 23-13 **(11)**：启动定时器，开始计时。

代码清单 23-13 **(12)**：因为现在线程是等待着了，要进行线程切换，所以进行一次线程调度。

代码清单 23-13 **(13)**：将接收到的邮件内容放到接收地址中，在接收线程中用户可以自己定义接收的类型，可以是 4 字节内的任意内容，也可以是指针。

代码清单 23-13 **(14)**：更新接收邮件指针的偏移地址，因为邮箱是一个内存池，其存放邮件的地址在 32 位机器中指针下标自加 1 偏移刚好是 4 个字节，如果有邮件的话指向下一个邮件的地址（如果没有邮件，那么就是空闲地址）。

代码清单 23-13 **(15)**：判断接收邮件指针的偏移地址是否到达邮箱最大容量，如果是，则重置为 0。

代码清单 23-13 **(16)**：记录当前邮件数量，每接收一封邮件就要减少一封邮件。

代码清单 23-13 **(17)**：如果有线程因为发送邮件不成功而被阻塞的话，那么需要恢复该线程，调用 rt_ipc_list_resume() 函数将该线程恢复。

代码清单 23-13 **(18)**：进行一次线程调度。

代码清单 23-13 **(19)**：返回接收邮件结果。

接收邮件时，接收者需指定接收邮件的邮箱句柄，并指定接收到的邮件存放位置以及设置指定超时时间，成功收到邮件则返回 RT_EOK；当指定的时间内依然未收到邮件时，将返回-RT_ETIMEOUT。接收是允许带阻塞的，所以仅在线程中接收邮件，邮件接收函数 rt_mb_recv() 实例具体见代码清单 23-14 高亮部分。

列表 14: 代码清单 23-14 邮箱邮件接收函数 rt_mb_recv() 实例

```
/* 定义邮箱控制块 */
static rt_mailbox_t test_mail = RT_NULL;

static void receive_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
    char *r_str;
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        /* 等待接邮箱消息 */
        uwRet = rt_mb_recv(test_mail,                /* 邮箱对象句柄 */
                           (rt_uint32_t*)&r_str, /* 接收邮箱消息 */
```

（下页继续）

(续上页)

```
                        RT_WAITING_FOREVER);        /* 指定超时事件，一直等 */

        if (RT_EOK == uwRet) { /* 如果接收完成并且正确 */
            rt_kprintf ( "邮箱的内容是:%s\n\n",r_str);
            LED1_TOGGLE;        //LED1        反转
        } else
            rt_kprintf ( "邮箱接收错误！错误码是 0x%x\n",uwRet);
    }
}
```

26.7 邮箱的实验

邮箱实验是在 RT-Thread 中创建了两个线程，一个是发送邮件线程，一个是接收邮件线程，两个线程独立运行，发送邮件线程是通过检测按键的按下情况来发送邮件，假如发送邮件错误，就把发送邮件错误情况在串口打印出来，另一个线程是接收邮件线程，在没有接收到邮件之前一直等待邮件，一旦接收到邮件就通过串口调试助手把邮件里面的数据信息打印出来，具体见*代码清单 23-15* 高亮部分。

注意： 在使用邮箱时候请确保在 rtconfig.h 中打开 RT_USING_MAILBOX 这个宏定义。

列表 15: 代码清单 23-15 邮箱的实验

```
/**
*****************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2019-xx-xx
```

(下页继续)

(续上页)

```
* @brief   RT-Thread 邮箱
*******************************************************************
* @attention
*
* 实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
********************************************************************
*/

/*
*********************************************************************
*                              包含的头文件
*********************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"
/* RT-Thread 头文件 */
#include "rtthread.h"
/*
******************************************************************
*                              变量
******************************************************************
*/
/* 定义线程控制块 */
static rt_thread_t receive_thread = RT_NULL;
static rt_thread_t send_thread = RT_NULL;
/* 定义邮箱控制块 */
static rt_mailbox_t test_mail = RT_NULL;

/************************ 全局变量声明 ***************************/
```

(下页继续)

(续上页)

```
/*
* 当我们在写应用程序的时候，可能需要用到一些全局变量。
*/
char test_str1[] = "this is a mail test 1";/* 邮箱消息 test1 */
char test_str2[] = "this is a mail test 2";/* 邮箱消息 test2 */
/*
**********************************************************************
*                              函数声明
**********************************************************************
*/
static void receive_thread_entry(void* parameter);
static void send_thread_entry(void* parameter);

/*
**********************************************************************
*                              main 函数
**********************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]-i.MX RT 全系列开发板-RTT 邮箱消息实验! \n");
    rt_kprintf("按下 K1 | K2 进行邮箱实验测试!\n");
```

(下页继续)

(续上页)

```
    /* 创建一个邮箱 */
    test_mail = rt_mb_create("test_mail", /* 邮箱名字 */
                             10,          /* 邮箱大小 */
                             RT_IPC_FLAG_FIFO);/* 信号量模式 FIFO(0x00)*/
    if (test_mail != RT_NULL)
        rt_kprintf("邮箱创建成功! \n\n");

    receive_thread =                          /* 线程控制块指针 */
        rt_thread_create( "receive",              /* 线程名字 */
                          receive_thread_entry,   /* 线程入口函数 */
                          RT_NULL,              /* 线程入口函数参数 */
                          512,                  /* 线程栈大小 */
                          3,                    /* 线程的优先级 */
                          20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (receive_thread != RT_NULL)
        rt_thread_startup(receive_thread);
    else
        return -1;

    send_thread =                           /* 线程控制块指针 */
        rt_thread_create( "send",               /* 线程名字 */
                          send_thread_entry,    /* 线程入口函数 */
                          RT_NULL,              /* 线程入口函数参数 */
                          512,                  /* 线程栈大小 */
                          2,                    /* 线程的优先级 */
                          20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (send_thread != RT_NULL)
        rt_thread_startup(send_thread);
```

(下页继续)

(续上页)

```
    else
        return -1;
}

/*
*****************************************************************
*                              线程定义
*****************************************************************
*/

static void receive_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
    char *r_str;
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        /* 等待接邮箱消息 */
        uwRet = rt_mb_recv(test_mail, /* 邮箱对象句柄 */
                           (rt_uint32_t*)&r_str, /* 接收邮箱消息 */
                           RT_WAITING_FOREVER);/* 指定超时事件，一直等 */

        if (RT_EOK == uwRet) { /* 如果接收完成并且正确 */
            rt_kprintf ( "邮箱的内容是:%s\n\n",r_str);
            LED1_TOGGLE;        //LED1       反转
        } else
            rt_kprintf ( "邮箱接收错误！错误码是 0x%x\n",uwRet);
    }
}

static void send_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
```

(下页继续)

(续上页)

```
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        //如果 KEY1 被单击
        if ( Key_Scan(KEY1_GPIO_PORT,KEY1_GPIO_PIN) == KEY_ON ) {
            rt_kprintf ( "KEY1 被单击\n" );
            /* 发送一个邮箱消息 1 */
            uwRet = rt_mb_send(test_mail,(rt_uint32_t)&test_str1);
            if (RT_EOK == uwRet)
                rt_kprintf ( "邮箱消息发送成功\n" );
            else
                rt_kprintf ( "邮箱消息发送失败\n" );
        }
        //如果 KEY2 被单击
        if ( Key_Scan(KEY2_GPIO_PORT,KEY2_GPIO_PIN) == KEY_ON ) {
            rt_kprintf ( "KEY2 被单击\n" );
            /* 发送一个邮箱 2 */
            uwRet = rt_mb_send(test_mail,(rt_uint32_t)&test_str2);
            if (RT_EOK == uwRet)
                rt_kprintf ( "邮箱消息发送成功\n" );
            else
                rt_kprintf ( "邮箱消息发送失败\n" );
        }
        rt_thread_delay(20);      //每 20ms 扫描一次
    }
}
/***************************END OF FILE***************************/
```

26.8 邮箱的实验现象

程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息，按下开发版的 K1 按键发送邮件 1，按下 K2 按键发送邮件 2；我们按下 K1 与 K2 试试，在串口调试助手中可以看到运行结果，具体见图 *23-2*。

第 27 章　内存管理

27.1 内存管理的基本概念

在计算系统中，变量、中间数据一般存放在系统存储空间中，只有在实际使用时才将它们从存储空间调入到中央处理器内部进行运算。通常存储空间可以分为两种：内部存储空间和外部存储空间。内部存储空间访问速度比较快，能够按照变量地址随机地访问，也就是我们通常所说的 RAM（随机存储器），或电脑的内存；而外部存储空间内所保存的内容相对来说比较固定，即使掉电后数据也不会丢失，可以把它理解为电脑的硬盘。在这一章中我们主要讨论内部存储空间（RAM）的管理——内存管理。

RT-Thread 操作系统将内核与内存管理分开实现，操作系统内核仅规定了必要的内存管理函数原型，而不关心这些内存管理函数是如何实现的，所以在 RT-Thread 中提供了多种内存分配算法（分配策略），但是上层接口（API）却是统一的。这样做可以增加系统的灵活性：用户可以选择对自己更有利的内存管理策略，在不同的应用场合使用不同的内存分配策略。

在嵌入式程序设计中内存分配应该是根据所设计系统的特点来决定选择使用动态内存分配还是静态内存分配算法，一些可靠性要求非常高的系统应选择使用静态的，而普通的业务系统可以使用动态来提高内存使用效率。静态可以保证设备的可靠性但是需要考虑内存上限，内存使用效率低，而动态则是相反。

RT-Thread 的内存管理模块管理系统的内存资源，它是操作系统的核心模块之一。主要包括内存的初始化、分配以及释放。

很多人会有疑问，什么不直接使用 C 标准库中的内存管理函数呢？在电脑中我们可以用 malloc() 和 free() 这两个函数动态的分配内存和释放内存。但是，在嵌入式实时操作系统中，调用 malloc() 和 free() 却是危险的，原因有以下几点：

- 这些函数在小型嵌入式系统中并不总是可用的，小型嵌入式设备中的 RAM 不足。
- 它们的实现可能非常的大，占据了相当大的一块代码空间。
- 他们几乎都不是线程安全的。

- 它们并不是确定的，每次调用这些函数执行的时间可能都不一样。
- 它们有可能产生碎片。
- 这两个函数会使得链接器配置得复杂。
- 如果允许堆空间的生长方向覆盖其他变量占据的内存，它们会成为 debug 的灾难。

在一般的实时嵌入式系统中，由于实时性的要求，很少使用虚拟内存机制。所有的内存都需要用户参与分配，直接操作物理内存，所分配的内存不能超过系统的物理内存，所有的系统堆栈的管理，都由用户自己管理。

同时，在嵌入式实时操作系统中，对内存的分配时间要求更为苛刻，分配内存的时间必须是确定的。一般内存管理算法是根据需要存储的数据的长度在内存中去寻找一个与这段数据相适应的空闲内存块，然后将数据存储在里面。而寻找这样一个空闲内存块所耗费的时间是不确定的，因此对于实时系统来说，这就是不可接受的，实时系统必须要保证内存块的分配过程在可预测的确定时间内完成，否则实时线程对外部事件的响应也将变得不可确定。

而在嵌入式系统中，内存是十分有限而且是十分珍贵的，用一块内存就少了一块内存，而在分配中随着内存不断被分配和释放，整个系统内存区域会产生越来越多的碎片，因为在使用过程中，申请了一些内存，其中一些释放了，导致内存空间中存在一些小的内存块，它们地址不连续，不能够作为一整块的大内存分配出去，所以一定会在某个时间，系统已经无法分配到合适的内存了，导致系统瘫痪。其实系统中实际是还有内存的，但是因为小块的内存的地址不连续，导致无法分配成功，所以我们需要一个优良的内存分配算法来避免这种情况的出现。

不同的嵌入式系统具有不同的内存配置和时间要求。所以单一的内存分配算法只可能适合部分应用程序。因此，RT-Thread 将内存分配作为可移植层面（相对于基本的内核代码部分而言），RT-Thread 有针对性的提供了不同的内存分配管理算法，这使得应用于不同场景的设备可以选择适合自身内存算法。

RT-Thread 的内存管理模块的算法总体上可分为两类：静态内存管理与动态内存管理，而动态内存管理又根据可用内存的多少划分为两种情况：一种是针对小内存块的分配管理（小内存管理算法），另一种是针对大内存块的分配管理（SLAB 管理算法），需要使用的时候开启其对应的宏定义即可。

RT-Thread 的内存管理模块通过对内存的申请、释放操作，来管理用户和系统对内存的使用，使内存的利用率和使用效率达到最优，同时最大限度地解决系统的内存碎片问题。

RT-Thread 的内存管理分为静态内存管理和动态内存管理，提供内存初始化、分配、释放等功能但是各有优缺点。

动态内存：在动态内存池中分配用户指定大小的内存块。

- 优点：按需分配，在设备中灵活使用。
- 缺点：内存池中可能出现碎片。

静态内存：在静态内存池中分配用户初始化时预设（固定）大小的内存块。

- 优点：分配和释放效率高，静态内存池中无碎片。
- 缺点：只能申请到初始化预设大小的内存块，不能按需申请。

27.2 内存管理的运作机制

首先，在使用内存分配前，必须明白自己在做什么，这样做与其它的方法有什么不同，特别是会产生哪些负面影响，在自己的产品面前，应当选择哪种分配策略。

动态分配内存与静态分配内存的区别：静态内存一旦创建就指定了内存块的大小，分配只能以内存块大小粒度进行分配；动态内存分配则根据运行时环境确定需要的内存块大小，按照需要分配内存。

静态分配内存适合于可以确定需要占用内存多少的情况，静态分配内存的效率比动态分配内存的效率要高，但静态内存的利用率却比动态内存低，因为只能按照已经定义的内存块粒度大小进行分配，而动态内存则可以按需分配。

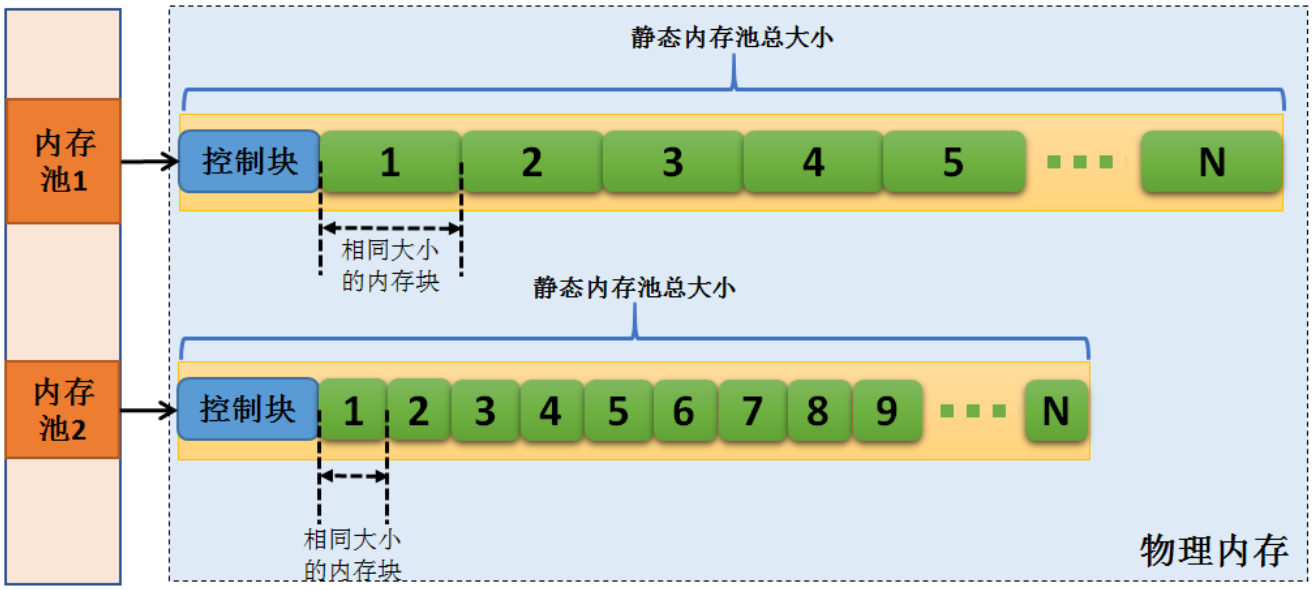
27.2.1 静态内存管理

内存池（Memory Pool）是一种用于分配大量大小相同的小内存对象的技术。它可以极大加快内存分配/释放的速度。

内存池在创建时先向系统申请一大块内存，然后分成大小相等的多个小内存块，小内存块直接通过链表连接起来（此链表也称为空闲内存链表）。每次分配的时候，从空闲内存链表中取出表头上第一个内存块，提供给申请者。物理内存中允许存在多个大小不同的内存池，每一个内存池又

由多个大小相同的空闲内存块组成。当一个内存池对象被创建时，内存池对象就被分配给了一个内存池控制块，内存控制块的参数包括内存池名，内存缓冲区，内存块大小，块数以及一个等待线程列表。

内核负责给内存池分配内存池对象控制块，它同时也接收用户线程的分配内存块申请，当获得申请信息后，内核就可以从内存池中为线程分配内存块。内存池一旦初始化完成，内部的内存块大小将不能再做调整，具体见图 *24-1*。



27.2.2 动态内存管理

动态内存管理是一个真实的堆（Heap）内存管理模块。动态内存管理，即在内存资源充足的情况下，从系统配置的一块比较大的连续内存，根据用户需求，在这块内存中分配任意大小的内存块。当用户不需要该内存块时，又可以释放回系统供下一次使用。与静态内存相比，动态内存管理的好处是按需分配，缺点是内存池中容易出现碎片（在申请与释放的时候由于内存不对齐会导致内存碎片）。RT-Thread 系统为了满足不同的需求，提供了两套不同的动态内存管理算法，分别是小堆内存管理算法和 SLAB 内存管理算法。

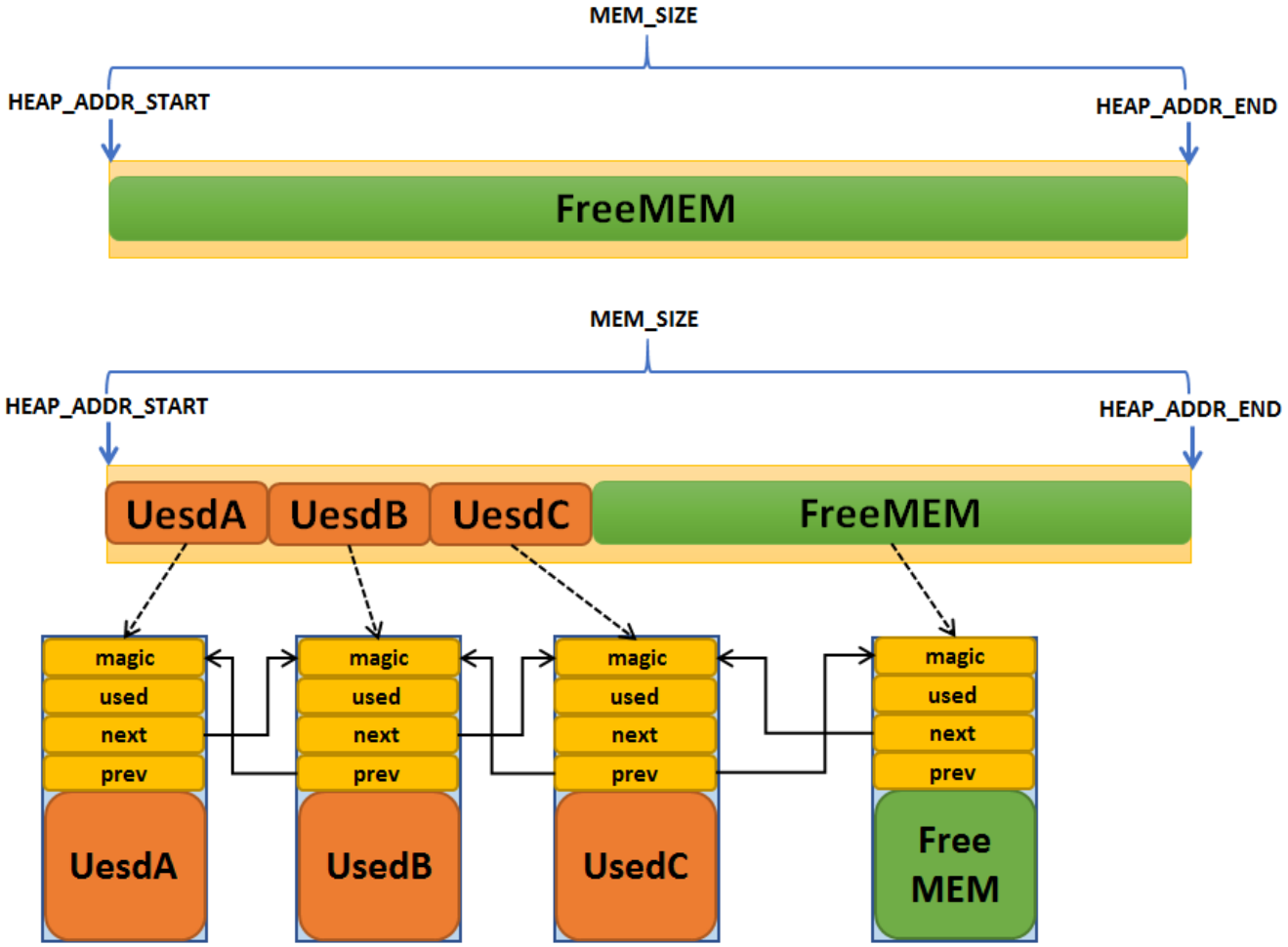
小堆内存管理模块主要针对系统资源比较少，一般用于小于 2M 内存空间的系统；而 SLAB 内存管理模块则主要是在系统资源比较丰富时，提供了一种近似多内存池管理算法的快速算法。两种内存管理模块在系统运行时只能选择其中之一或者完全不使用动态堆内存管理器，这两种内存

管理模块提供的 API 接口完全相同。

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

27.2.2.1 小内存管理模块

小内存管理算法是一个简单的内存分配算法。初始时，它是一块大的内存，其大小为(MEM_SIZE)，当需要分配内存块时，将从这个大的内存块上分割出相匹配的内存块，然后把分割出来的空闲内存块还回给堆管理系统中。每个内存块都包含一个管理用的数据头，通过这个头把使用块与空闲块用双向链表的方式链接起来（内存块链表），具体见图 *24-2* 和图 *24-3*。



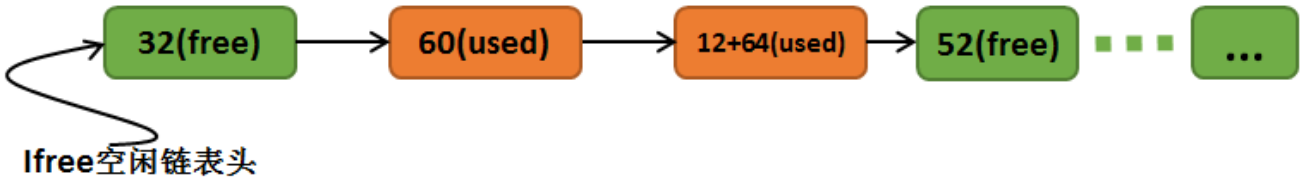
每个内存块（不管是已分配的内存块还是空闲的内存块）都包含一个数据头，其中包括：

- magic –变数（或称为幻数），它会被初始化成 0x1ea0（即英文单词 heap），用于标记这个内存块是一个内存管理用的内存数据块；
- used - 指示出当前内存块是否已经分配。

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

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

如图 *24-4* 所示的内存分配情况，空闲链表指针 lfree 初始指向 32 字节的内存块。当用户线程要再分配一个 64 字节的内存块时，但此 lfree 指针指向的内存块只有 32 字节并不能满足要求，内存管理器会继续寻找下一内存块，当找到再下一块内存块，128 字节时，它满足分配的要求。因为这个内存块比较大，分配器将把此内存块进行拆分，余下的内存块（52 字节）继续留在 lfree 链表中，具体见图 *24-5*。



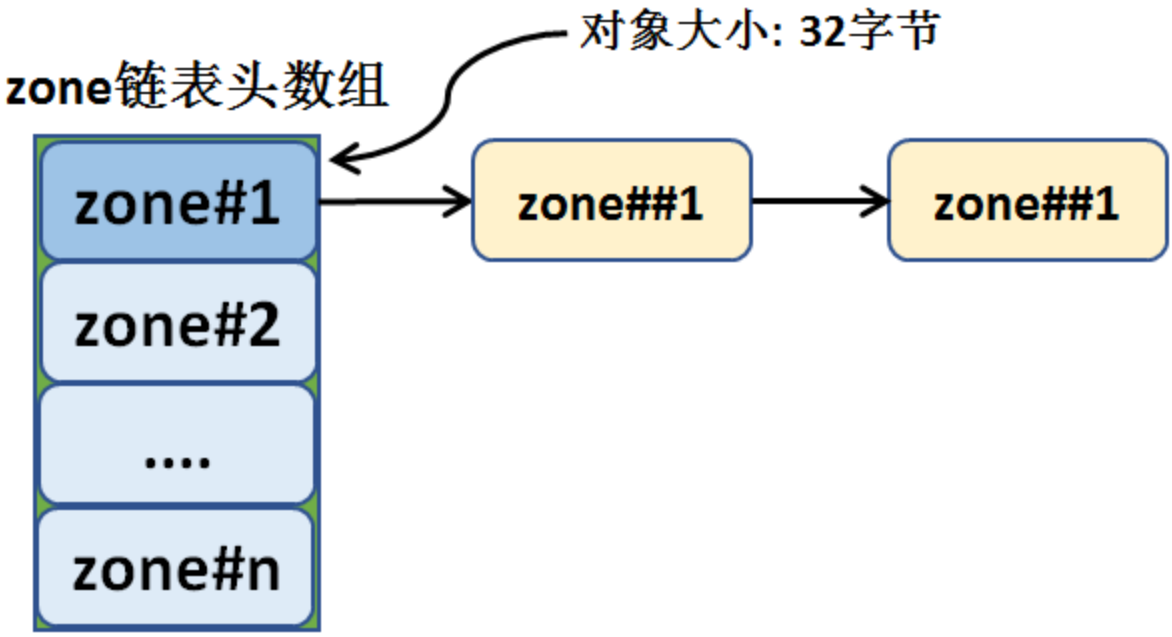
另外，在每次分配内存块前，都会留出 12 字节数据头用于 magic，used 信息及链表节点使用。返回给应用的地址实际上是这块内存块 12 字节以后的地址，而数据头部分是用户永远不应该改变的部分。（注：12 字节数据头长度会与系统对齐差异而有所不同）。

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

27.2.2.2 SLAB 内存管理模块

RT-Thread 的 SLAB 分配器是在 DragonFly BSD 创始人 Matthew Dillon 实现的 SLAB 分配器基础上，针对嵌入式系统优化的内存分配算法。最原始的 SLAB 算法是 Jeff Bonwick 为 Solaris 操作系统而引入的一种高效内核内存分配算法。

RT-Thread 的 SLAB 分配器实现主要是去掉了其中的对象构造及析构过程，只保留了纯粹的缓冲型的内存池算法。SLAB 分配器会根据对象的类型（主要是大小）分成多个区（zone），也可以看成每类对象有一个内存池，具体见图 *24-6*。



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

下面是动态内存分配器主要的两种操作：

- 内存分配：假设分配一个 32 字节的内存，SLAB 内存分配器会先按照 32 字节的值，从 zone array 链表表头数组中找到相应的 zone 链表。如果这个链表是空的，则向页分配器分配一个新的 zone，然后从 zone 中返回第一个空闲内存块。如果链表非空，则这个 zone 链表中的第

一个 zone 节点必然有空闲块存在（否则它就不应该放在这个链表中），那么就取相应的空闲块。如果分配完成后，zone 中所有空闲内存块都使用完毕，那么分配器需要把这个 zone 节点从链表中删除。

- 内存释放：分配器需要找到内存块所在的 zone 节点，然后把内存块链接到 zone 的空闲内存块链表中。如果此时 zone 的空闲链表指示出 zone 的所有内存块都已经释放，即 zone 是完全空闲的，那么当 zone 链表中全空闲 zone 达到一定数目后，系统就会把这个全空闲的 zone 释放到页面分配器中去。

27.3 内存管理的应用场景

RT-Threadd 操作系统将内核与内存管理分开实现，操作系统内核仅规定了必要的内存管理函数原型，而不关心这些内存管理函数是如何实现的。这样做大有好处，可以增加系统的灵活性：不同的应用场合可以使用不同的内存分配实现，用户也能自己通过 API 接口进行对内存的管理，选择对自己更有利的内存管理策略。

内存管理的主要工作是动态划分并管理用户分配好的内存区间，主要是在用户需要使用大小不等的内存块的场景中使用，当用户需要分配内存时，可以通过操作系统的动态内存申请函数索取指定大小内存块，一旦使用完毕，通过动态内存释放函数归还所占用内存，使之可以重复使用。

静态内存管理是当用户需要使用固定长度的内存时，可以使用静态内存分配的方式获取内存，一旦使用完毕，通过静态内存释放函数归还所占用内存，使之可以重复使用。

例如我们需要定义一个 float 型数组：float Arr[];

但是，在使用数组的时候，总有一个问题困扰着我们：数组应该有多大？在很多的情况下，你并不能确定要使用多大的数组，可能为了避免发生错误你就需要把数组定义得足够大。即使你知道想利用的空间大小，但是如果因为某种特殊原因空间利用的大小有增加或者减少，你又必须重新去修改程序，扩大数组的存储范围。这种分配固定大小的内存分配方法称之为静态内存分配。这种内存分配的方法存在比较严重的缺陷，在大多数情况下会浪费大量的内存空间，在少数情况下，当你定义的数组不够大时，可能引起下标越界错误，甚至导致严重后果。

我们用动态内存分配就可以解决上面的问题。所谓动态内存分配就是指在程序执行的过程中动态地分配或者回收存储空间的分配内存的方法。动态内存分配不象数组等静态内存分配方法那

样需要预先分配存储空间，而是由系统根据程序的需要即时分配，且分配的大小就是程序要求的大小。

27.4 静态内存管理的函数接口讲解

对于一些安全型的嵌入式系统，通常不允许动态内存分配，那么可以采用非常简单的内存管理策略，一经申请的内存，甚至不允许被释放，在满足设计要求的前提下，系统越简单越容易做的更安全。RT-Threadd 也提供了静态内存管理的函数，下面一起来看看静态内存管理函数的使用。

静态内存的典型场景开发流程：

1. 规划一片内存区域作为静态内存池。
2. 调用 rt_mp_create() 函数。进行静态内存使用前的创建。
3. 调用 rt_mp_alloc() 函数。系统内部将会从空闲链表中获取第一个空闲块，并返回该块的用户空间地址。
4. 调用 rt_mp_free() 函数。将该块内存加入空闲块链表，进行内存的释放。

27.4.1 静态内存控制块

RT-Threadd 对内存的控制很严格，哪个线程哪个模块用了哪些内存都要知道，我们知道控制块常用于保存使用信息，所以，静态内存的管理也一样离不开控制块。每一个静态内存池都有一个内存控制块保存信息，下面一起来看看内存池控制块吧，具体见*代码清单 24-1*。

列表 1: 代码清单 24-1 静态内存控制块

```
struct rt_mempool {
    struct rt_object parent;      /**< 继承自 rt_object */                    (1)

    void            *start_address;  /**< 内存池起始地址 */                  (2)
    rt_size_t        size;                    /**< 内存池大小 */             (3)

```

(下页继续)

(续上页)

```
    rt_size_t        block_size; /**< 内存块大小 */                    (4)
    rt_uint8_t      *block_list;  /**< 内存块链表 */                   (5)

    rt_size_t        block_total_count; /**< 内存块总数量 */           (6)
    rt_size_t        block_free_count; /**< 空闲内存块数量 */          (7)

    rt_list_t        suspend_thread;                                  (8)
    rt_size_t        suspend_thread_count;                             (9)
};
typedef struct rt_mempool *rt_mp_t;
```

代码清单 24-1 **(1)**：静态内存会在自身结构体里面包含一个对象类型的成员，通过这个成员可以将内存挂到系统对象容器里面。

代码清单 24-1 **(2)**：内存池开始地址。

代码清单 24-1 **(3)**：内存池大小。

代码清单 24-1 **(4)**：内存块大小，也就是我们实际申请内存块的大小，单位为字节。

代码清单 24-1 **(5)**：内存块链表，所有可用的内存块都挂载在此链表上。

代码清单 24-1 **(6)**：内存池数据区域中能够容纳的最大内存块数。

代码清单 24-1 **(7)**：内存池中空闲的内存块数。

代码清单 24-1 **(8)**：挂起在内存池的线程列表。

代码清单 24-1 **(9)**：挂起在内存池的线程数量。

27.4.2 静态内存创建函数 rt_mp_create()

在使用静态内存的时候首先要创建一个内存池，从堆上分配划分一块连续的区域作为静态内存池。创建内存池后，线程才可以从内存池中申请、释放内存，RT-Thread 提供静态内存池创建函数 rt_mp_create()，该函数返回一个已创建的内存池对象，内存池创建函数 rt_mp_create() 源码具体见代码清单 24-2。

列表 2: 代码清单 24-2 静态内存创建函数 rt_mp_create() 源码

```
/**
* 此函数将创建一个 mempool 对象并从堆中分配内存池。
*
*
* @param name 内存池名称
* @param block_count 内存块数量
* @param block_size 内存块大小
*
* @return 已创建的内存池对象
*/
rt_mp_t rt_mp_create(const char *name,                          (1)
                     rt_size_t   block_count,                   (2)
                     rt_size_t   block_size)                    (3)
{
    rt_uint8_t *block_ptr;
    struct rt_mempool *mp;
    register rt_base_t offset;

    RT_DEBUG_NOT_IN_INTERRUPT;

    /* 分配对象 */
    mp = (struct rt_mempool *)rt_object_allocate(RT_Object_Class_MemPool,␣
↪name);
    /* 分配对象失败 */
    if (mp == RT_NULL)                                          (4)
        return RT_NULL;

    /* 初始化内存池信息 */
    block_size     = RT_ALIGN(block_size, RT_ALIGN_SIZE);       (5)
```

(下页继续)

(续上页)

```
    mp->block_size = block_size;                                (6)
    mp->size  = (block_size + sizeof(rt_uint8_t *)) * block_count; (7)

    /* 分配内存 */
    mp->start_address = rt_malloc((block_size + sizeof(rt_uint8_t *)) *
                                  block_count);                 (8)
    if (mp->start_address == RT_NULL) {
        /* 没有足够内存，删除内存池对象句柄 */
        rt_object_delete(&(mp->parent));                        (9)

        return RT_NULL;
    }

    mp->block_total_count = block_count;                        (10)
    mp->block_free_count  = mp->block_total_count;              (11)

    /* 初始化阻塞链表 */
    rt_list_init(&(mp->suspend_thread));                        (12)
    mp->suspend_thread_count = 0;

    /* 初始化空闲内存块链表 */
    block_ptr = (rt_uint8_t *)mp->start_address;                (13)
    for (offset = 0; offset < mp->block_total_count; offset ++) { (14)
        *(rt_uint8_t **)(block_ptr + offset * (block_size + sizeof(rt_uint8_
↪t *)))
        = block_ptr + (offset + 1) * (block_size + sizeof(rt_uint8_t *));
    }                                                           (15)

    *(rt_uint8_t **)(block_ptr + (offset - 1) * (block_size + sizeof(rt_
↪uint8_t *)))
        = RT_NULL;                                              (16)

```

(下页继续)

(续上页)

```
    mp->block_list = block_ptr;                                    (17)

    return mp;                                                     (18)
}
RTM_EXPORT(rt_mp_create);
```

代码清单 24-2 **(1)**：name 内存池名称。

代码清单 24-2 **(2)**：block_count 初始化内存池中可分配内存块最大数量。

代码清单 24-2 **(3)**：block_size 初始化内存块的大小，单位为字节。

代码清单 24-2 **(4)**：分配内存池对象，调用 rt_object_allocate() 函数将从对象系统分配内存池对象，并且命名内存池对象名称，在系统中，对象的名称必须是唯一的。

代码清单 24-2 **(5)**：初始化内存池信息，初始化内存块大小，使其对齐方式与系统内存对齐方式一致，配置 block_size 以 4 字节对齐，如果不满足对齐倍数将返回其最小的对齐倍数，如想要对齐 13 字节大小的内存块，RT_ALIGN（13,4），将返回 16（字节）。

代码清单 24-2 **(6)**：内存块大小按传递进来的 block_size 来进行初始化配置。

代码清单 24-2 **(7)**：计算得出内存池需要的内存大小，其大小为 (block_size + sizeof(rt_uint8_t *)) * block_count，也就是 [内存块大小 +4 个字节大小（指向内存池控制块）] 乘以内存块的数量。

代码清单 24-2 **(8)**：分配内存池，调用 rt_malloc () 函数将从系统管理的堆中划分一块连续的内存，作为静态内存池，分配的内存大小为内存池大小。很多人会问了，都没创建，怎么分配内存？此处分配的内存是调用 rt_malloc() 进行动态内存分配，因为 RT-Thread 必须采用动态内存分配的方式，所以此处只是划分一块堆内存区域给我们当做静态内存池使用，初始化内存池之后，这块区域的内存就是静态的，只能使用静态内存管理接口访问。

代码清单 24-2 **(9)**：系统已经没有足够的内存了，分配失败，需要删除内存池对象句柄，所以在静态内存池创建的时候一定要考虑到系统的内存大小。

代码清单 24-2 **(10)**：分配成功，静态内存控制块的 block_total_count（内存块总数量）就是创建时候由用户定义的 block _count。

代码清单 24-2 **(11)**：初始化空闲内存块数量。

代码清单 24-2 **(12)**：初始化线程的阻塞列表和在此列表上线程的数量。

代码清单 24-2 **(13)**：初始化第一个内存块的起始地址。

代码清单 24-2 **(14)**：在 for 循环中初始化空闲内存块列表，循环执行次数为空闲内存块的数量值。

代码清单 24-2 **(15)**：将所有的内存块都连接起来，在分配的时候更容易管理，其初始化结果具体见图 *24-7*。

初始化内存

代码清单 24-2 **(16)**：最后一块内存块的下一个内存是没有了，就是 NULL。

代码清单 24-2 **(17)**：内存块列表指向第一块可用内存块。

代码清单 24-2 **(18)**：创建成功返回内存池对象句柄。

使用该函数接口可以创建一个静态内存池，前提是在系统资源允许的情况下（最主要的是动态堆内存资源）才能创建成功。创建内存池时，需要给内存池指定一个名称，从系统中申请一个内存池对象，然后从堆内存中划分一块连续的内存区域作为静态内存池，并将内存区域组织成用于静态分配的空闲块列表，创建内存池成功将返回内存池的句柄，否则返回 RT_NULL。静态内存创建函数 rt_mp_create() 使用实例具体见代码清单 *24-3* 高亮部分。

列表 3: 代码清单 24-3 静态内存创建函数 rt_mp_create() 实例

```
/* 定义内存池控制块 */
static rt_mp_t test_mp = RT_NULL;
/* 定义申请内存的指针 */
static rt_uint32_t *p_test = RT_NULL;
/* 相关宏定义 */
#define  BLOCK_COUNT   20            //内存块数量
```

（下页继续）

(续上页)

```
#define  BLOCK_SIZE   3                //内存块大小
/* 创建一个静态内存池 */
test_mp = rt_mp_create("test_mp",
                    BLOCK_COUNT,
                    BLOCK_SIZE);
if (test_mp != RT_NULL)
    rt_kprintf("静态内存池创建成功! \n\n");
```

27.4.3 静态内存删除函数 rt_mp_delete()

删除内存池时，会首先唤醒等待在该内存池对象上的所有线程（返回-RT_ERROR），然后再释放已从内存堆上分配的内存池数据存放区域，然后删除内存池对象。删除内存池后将无法向内存池申请内存块，静态内存删除函数 rt_mp_delete() 源码具体见*代码清单 24-4*。

列表 4: 代码清单 24-4 静态内存删除函数 rt_mp_delete() 源码

```
/**
* 这个函数会删除内存池对象并且释放内存池对象的内存
*
* @param mp 内存池对象句柄
*
* @return 删除成功返回 RT_EOK
*/
rt_err_t rt_mp_delete(rt_mp_t mp)                                (1)
{
    struct rt_thread *thread;
    register rt_ubase_t temp;

    RT_DEBUG_NOT_IN_INTERRUPT;

```

(下页继续)

(续上页)

```
    /* 检查内存池对象 */
    RT_ASSERT(mp != RT_NULL);                                   (2)

    /* 唤醒所有在阻塞中的线程 */
    while (!rt_list_isempty(&(mp->suspend_thread))) {           (3)
        /* 关中断 */
        temp = rt_hw_interrupt_disable();

        /* 获取阻塞线程 */                                       (4)
        thread = rt_list_entry(mp->suspend_thread.next, struct rt_thread,␣
↪tlist);
        /* 返回线程错误 */
        thread->error = -RT_ERROR;

        /*
        * 恢复线程
        * 在 rt_thread_resume 函数中，它将从挂起列表中删除当前线程
        *
        */
        rt_thread_resume(thread);                               (5)

        /* 挂起线程数减一 */
        mp->suspend_thread_count --;                            (6)

        /* 开中断 */
        rt_hw_interrupt_enable(temp);
    }

#if defined(RT_USING_MODULE) && defined(RT_USING_SLAB)          (7)

    if (mp->parent.flag & RT_OBJECT_FLAG_MODULE)
        rt_module_free(mp->parent.module_id, mp->start_address);
```

(下页继续)

(续上页)

```
    else
#endif

        /* 释放申请的内存池 */
        rt_free(mp->start_address);                          (8)

    /* 删除内存池对象 */
    rt_object_delete(&(mp->parent));                         (9)

    return RT_EOK;                                           (10)
}
RTM_EXPORT(rt_mp_delete);
```

代码清单 24-4 **(1)**：mp 内存池对象句柄，根据内存池对象句柄决定要删除的是哪个内存池。

代码清单 24-4 **(2)**：检查内存池对象句柄 mp 是否有效。

代码清单 24-4 **(3)**：如果当前有线程挂在内存池的阻塞列表中，需要将该线程唤醒，直到没有线程阻塞的时候才退出 while 循环。

代码清单 24-4 **(4)**：获取阻塞的线程。

代码清单 24-4 **(5)**：调用 rt_thread_resume 线程恢复函数，将该线程恢复，该函数会将线程从阻塞链表中删除。

代码清单 24-4 **(6)**：将内存池控制块中记录线程挂起数量的 suspend_thread_count 变量减一。

代码清单 24-4 **(7)**：在这里我们并没有使用 slab 分配机制，未使能 RT_USING_SLAB 这个宏定义，所以还不需要使用 rt_module_free 释放内存函数。

代码清单 24-4 **(8)**：释放内存池的内存，因为这个内存池是从系统堆内存动态划分的，删除后要进行释放。

代码清单 24-4 **(9)**：调用 rt_object_delete() 函数删除内存池对象。

代码清单 24-4 **(10)**：返回删除结果 RT_EOK。

内存池的删除函数实现过程我们都已经了解了，用 rt_mp_delete() 来删除我们需要删除的内存池，那岂不是手到擒来，但是需要注意的是：删除的时候会将所有因为申请不到内存块而进入阻塞的线程恢复，被恢复的线程会得到一个-RT_ERROR，所以，建议在删除内存池之前我们应确保所有的线程没有阻塞，并且以后也不会再向这个内存池申请内存块，才进行删除操作，这样子才是最稳妥的办法，rt_mp_delete() 的使用实例具体见*代码清单 24-5* 高亮部分。

列表 5: 代码清单 24-5 静态内存删除函数 rt_mp_delete() 实例

```
/* 定义内存池控制块 */
static rt_mp_t test_mp = RT_NULL;

rt_err_t uwRet = RT_EOK;

/* 删除一个静态内存池 */
uwRet = rt_mp_delete(test_mp);
if (RT_EOK == uwRet)
    rt_kprintf("静态内存池删除成功！\n\n");
```

27.4.4 静态内存初始化函数 rt_mp_init()

初始化内存池跟创建内存池类似，只是初始化内存池用于静态内存管理模式，内存池控制块来源于用户在系统中申请的静态对象。另外与创建内存池不同的是，此处内存池对象所使用的内存空间是由用户指定的一个缓冲区空间，用户把缓冲区的指针传递给内存池对象控制块，其余的初始化工作与创建内存池相同，具体见*代码清单 24-6*。

列表 6: 代码清单 24-6 静态内存初始化函数 rt_mp_init() 源码

```
/**
*
* 此函数将初始化内存池对象，通常用于静态对象。
```

(下页继续)

(续上页)

```
 *
 * @param mp 内存池对象
 * @param name 内存池名称
 * @param start 内存池起始地址
 * @param size 内存池总大小
 * @param block_size 每个内存块的大小
 *
 * @return RT_EOK
 */
rt_err_t rt_mp_init(struct rt_mempool *mp,                          (1)
                    const char        *name,                        (2)
                    void              *start,                       (3)
                    rt_size_t          size,                        (4)
                    rt_size_t          block_size)                  (5)
{
    rt_uint8_t *block_ptr;
    register rt_base_t offset;

    /* 检查内存池 */
    RT_ASSERT(mp != RT_NULL);                                       (6)

    /* 初始化内存池对象 */
    rt_object_init(&(mp->parent), RT_Object_Class_MemPool, name);   (7)

    /* 初始化内存池 */
    mp->start_address = start;                                      (8)
    mp->size = RT_ALIGN_DOWN(size, RT_ALIGN_SIZE);                  (9)

    /* 内存块大小对齐 */
    block_size = RT_ALIGN(block_size, RT_ALIGN_SIZE);               (10)
    mp->block_size = block_size;

```

(下页继续)

(续上页)

```
36
37 mp->block_total_count = mp->size / (mp->block_size + sizeof(rt_uint8_t *));
38     mp->block_free_count  = mp->block_total_count;                    (11)
39
40     /* 初始化阻塞链表 */
41     rt_list_init(&(mp->suspend_thread));                              (12)
42     mp->suspend_thread_count = 0;
43
44     /* 初始化内存块空闲链表 */
45     block_ptr = (rt_uint8_t *)mp->start_address;                      (13)
46     for (offset = 0; offset < mp->block_total_count; offset ++) {     (14)
47     *(rt_uint8_t **)(block_ptr + offset * (block_size + sizeof(rt_uint8_t␣
↪*))) =
48     (rt_uint8_t *)(block_ptr + (offset + 1) * (block_size + sizeof(rt_uint8_
↪t *)));
49     }                                                                 (15)
50
51 *(rt_uint8_t **)(block_ptr + (offset - 1) * (block_size + sizeof(rt_uint8_t␣
↪*))) =
52         RT_NULL;                                                      (16)
53
54     mp->block_list = block_ptr;                                       (17)
55
56     return RT_EOK;                                                    (18)
57 }
58 RTM_EXPORT(rt_mp_init);
```

代码清单 24-6 **(1)**：mp 内存池对象句柄。

代码清单 24-6 **(2)**：name 内存池名称，是字符串常量类型。

代码清单 24-6 **(3)**：start 内存池起始地址，由用户自己定义的具体的起始地址。

代码清单 24-6 **(4)**：size 初始化内存池总容量大小。

代码清单 24-6 **(5)**：block_size 每个内存块的大小。

代码清单 24-6 **(6)**：检查内存池对象句柄 mp 是否有效。

代码清单 24-6 **(7)**：初始化内存池内核对象。调用 rt_object_init() 此函数将初始化内存池对象并将其添加到对象管理系统，在系统中，对象的名称必须是唯一的。

代码清单 24-6 **(8)**：初始化内存池，内存池的地址是由用户传递进来的地址。

代码清单 24-6 **(9)**：初始化内存池容量 size，使其以 4 字节对齐方式对齐，如果不满足对齐倍数将返回其最小的对齐倍数，如想要对齐 13 字节大小的内存块，RT_ALIGN（13,4），将返回 16（字节）。

代码清单 24-6 **(10)**：初始化内存块大小 block_size，使其对齐方式与系统内存对齐方式一致，配置 block_size 以 4 字节对齐。

代码清单 24-6 **(11)**：通过计算得出内存池中最大内存块数量，例如内存池大小为 200 个字节，内存块的大小为 16 个字节，但是需要再加上 4 个字节大小的内存头（指向内存池控制），很显然，内存块的数量最大为 5=200/(16+4)，并且初始化可用空闲内存块个数。

代码清单 24-6 **(12)**：初始化线程的阻塞列表和线程阻塞的数量。

代码清单 24-6 **(13)**：初始化第一个内存块的起始地址。

代码清单 24-6 **(14)**：在 for 循环中初始化空闲内存块链表，循环执行次数为空闲内存块的数量值。

代码清单 24-6 **(15)**：将所有的空闲内存块都连接起来，在分配的时候更容易管理，其初始化结果具体见图 *24-8*。

初始化内存

代码清单 24-6 **(16)**：最后一块内存块的下一个内存是没有了，就是 NULL。

代码清单 24-6 **(17)**：内存控制块的 block_list 内存块链表指向第一块可用内存块。

代码清单 24-6 **(18)**：创建成功返回内存池对象句柄。

其实静态内存初始化函数 rt_mp_init() 与静态内存创建函数 rt_mp_create() 的过程差不多，初始化内存池时，调用 rt_mp_init() 函数一般需要我们定义一个区域作为静态内存池，这个区域一般为一个大数组，这样，系统就可以对该内存池进行初始化，将内存池用到的内存空间组织成可用于分配的空闲块列表，其具体使用方法见代码清单 24-7 高亮部分。

列表 7: 代码清单 24-7 静态内存初始化函数 rt_mp_init() 实例

```
/* 定义内存池控制块 */
static rt_mp_t test_mp = RT_NULL;
static rt_uint8_t mempool[4096];

rt_err_t uwRet = RT_EOK;

/* 初始化内存池对象 */
uwRet = rt_mp_init(&test_mp,                    /** 内存池对象 **/
                "test_mp",              /** 内存池名称 **/
                &mempool[0],            /** 内存池起始地址 **/
                sizeof(mempool),        /** 内存池总大小 **/
                80);                    /** 每个内存块的大小 **/
if (RT_EOK == uwRet)
    rt_kprintf("初始化内存成功！\n");
```

27.4.5 静态内存申请函数 rt_mp_alloc()

这个函数用于申请固定大小的内存块，从指定的内存池中分配一个内存块给用户使用，该内存块的大小在内存池初始化的时候就已经决定的。如果内存池中有可用的内存块，则从内存池的内存块列表上取下一个内存块；如果内存池中已经没有可用内存块，则根据用户设定的超时时间把当前线程挂在内存池的阻塞列表中，直到内存池中有可用内存块，其源码具体见代码清单 24-8。

列表 8: 代码清单 24-8 静态内存申请函数 rt_mp_alloc()

```
/**
* 这个函数用于从指定内存池分配内存块
*
* @param mp 内存池对象
* @param time 超时时间
*
* @return 分配成功的内存块地址或 RT_NULL 表示分配失败
*/
void *rt_mp_alloc(rt_mp_t mp, rt_int32_t time)                    (1)
{
    rt_uint8_t *block_ptr;
    register rt_base_t level;
    struct rt_thread *thread;
    rt_uint32_t before_sleep = 0;

    /* 获取当前线程 */
    thread = rt_thread_self();                                    (2)

    /* 关中断 */
    level = rt_hw_interrupt_disable();

    while (mp->block_free_count == 0) {                           (3)
        /* 无内存块可用 */
        if (time == 0) {                                          (4)
            /* 开中断 */
            rt_hw_interrupt_enable(level);

            rt_set_errno(-RT_ETIMEOUT);

            return RT_NULL;
```

(下页继续)

(续上页)

```
        }

        RT_DEBUG_NOT_IN_INTERRUPT;

        thread->error = RT_EOK;

        /* 需要挂起当前线程 */
        rt_thread_suspend(thread);                                  (5)
        rt_list_insert_after(&(mp->suspend_thread), &(thread->tlist));
        mp->suspend_thread_count++;                                 6)

        if (time > 0) {
            /* 获取当前系统时间 */
            before_sleep = rt_tick_get();                           (7)

            /* 重置线程超时时间并且启动定时器 */
            rt_timer_control(&(thread->thread_timer),               (8)
                             RT_TIMER_CTRL_SET_TIME,
                             &time);
            rt_timer_start(&(thread->thread_timer));                (9)
        }

        /* 开中断 */
        rt_hw_interrupt_enable(level);

        /* 发起线程调度 */
        rt_schedule();                                              (10)

        if (thread->error != RT_EOK)
            return RT_NULL;

        if (time > 0) {
```

(下页继续)

(续上页)

```
            time -= rt_tick_get() - before_sleep;
            if (time < 0)
                time = 0;
        }
        /* 关中断 */
        level = rt_hw_interrupt_disable();
    }

    /* 内存块可用，记录当前可用内存块个数，申请之后空闲内存块数量减一 */
    mp->block_free_count--;                                    (11)

    /* 获取内存块指针 */
    block_ptr = mp->block_list;                                (12)
    RT_ASSERT(block_ptr != RT_NULL);

    /* 设置下一个空闲内存块为可用内存块 */
    mp->block_list = *(rt_uint8_t **)block_ptr;                (13)


    *(rt_uint8_t **)block_ptr = (rt_uint8_t *)mp;              (14)

    /* 开中断 */
    rt_hw_interrupt_enable(level);

    RT_OBJECT_HOOK_CALL(rt_mp_alloc_hook,
                        (mp, (rt_uint8_t *)(block_ptr + sizeof(rt_uint8_t␣
↪*))));

    return (rt_uint8_t *)(block_ptr + sizeof(rt_uint8_t *));(15)
}
RTM_EXPORT(rt_mp_alloc);
```

代码清单 *24-8* **(1)**：mp 内存池对象，time 超时时间。

代码清单 *24-8* **(2)**：获取当前线程。

代码清单 *24-8* **(3)**：如果无内存块可用，进入 while 循环。

代码清单 *24-8* **(4)**：如果用户不设置等待时间，则直接返回错误码。

代码清单 *24-8* **(5)**：因为能到这一步，用户肯定设置了等待时间的，那么，不管三七二十一将当前线程挂起。

代码清单 *24-8* **(6)**：记录因为挂起的线程数量。

代码清单 *24-8* **(7)**：获取当前系统时间。

代码清单 *24-8* **(8)**：重置线程计时器的超时时间，调用 rt_timer_control() 函数改变当前线程阻塞时间 thread_timer。

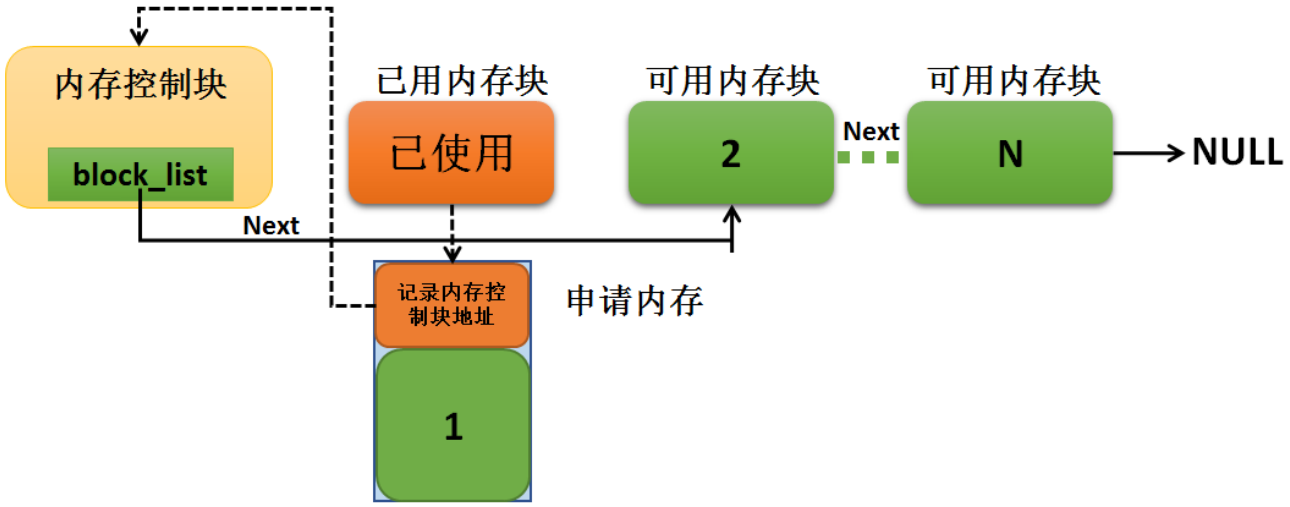
代码清单 *24-8* **(9)**：启动定时器开始计时。

代码清单 *24-8* **(10)**：因为现在线程是等待着了，要进行线程切换，所以进行一次线程调度。

代码清单 *24-8* **(11)**：当前内存池中还有内存块可用，记录当前可用内存块个数，申请之后可用内存块数量减一。

代码清单 *24-8* **(12)**：获取内存块指针，指向空闲的内存块。

代码清单 *24-8* **(13)**：设置当前申请内存块的下一个内存块为可用内存块，将 mp->block_list 的指针指向下一个内存块，具体见图 *24-9*。



代码清单 24-8 **(14)**：如图 24-9 所示，每一个内存块的前 4 个字节是指向内存池控制块的指针，为的是让我们在释放内存的时候能找到内存控制块。为什么要记录内存控制块指针呢？因为 block_list 是单链表，在申请成功内存的时候，已使用的内存块相当于脱离了内存块列表，那么在释放内存块的时候就没办法正常释放，所以需要保存内存控制块的指针。

代码清单 24-8 **(15)**：返回用户真正能读写操作的内存地址，其地址向下偏移了 4 个字节。

静态内存申请函数 rt_mp_alloc() 的使用是很简单的，仅需配置申请静态内存池句柄与超时时间即可，申请成功后返回指向用户可以操作的内存块地址，所以我们需要定义一个可以对内存块地址进行读写的指针，对申请的内存块进行访问，具体见代码清单 24-9 高亮部分。

列表 9: 代码清单 24-9 静态内存申请函数 rt_mp_alloc() 的使用实例

```
/* 定义申请内存的指针 */
static rt_uint32_t *p_test = RT_NULL;
rt_kprintf("正在向内存池申请内存...........\n");

p_test = rt_mp_alloc(test_mp,0);
if (RT_NULL == p_test) /* 没有申请成功 */
    rt_kprintf("静态内存申请失败! \n");
else
```

(下页继续)

(续上页)

```
    rt_kprintf("静态内存申请成功，地址为%d! \n\n",p_test);

rt_kprintf("正在向 p_test 写入数据...........\n");
*p_test = 1234;
rt_kprintf("已经写入 p_test 地址的数据\n");
rt_kprintf("*p_test = %.4d , 地址为:%d \n\n", *p_test,p_test);
```

27.4.6 静态内存释放函数 rt_mp_free()

入式系统的内存对我们来说是十分珍贵的，任何内存块使用完后都必须被释放，否则会造成内存泄露，导致系统发生致命错误。RT-Thread 提供了 rt_mp_free() 函数进行静态内存的释放管理，使用该函数接口时，根据内存块得到该内存块所在的（或所属于的）内存池对象，然后增加该内存池的可用内存块数目，并把该被释放的内存块加入内存块列表，接着判断该内存池对象上是否有挂起的线程，如果有，则唤醒线程，其源码具体见*代码清单 24-10*。

列表 10: 代码清单 24-10 静态内存释放函数 rt_mp_free() 源码

```
/**
* 这个函数会释放一个内存块
*
* @param block 要释放的内存块的地址
*/
void rt_mp_free(void *block)                                    (1)
{
    rt_uint8_t **block_ptr;
    struct rt_mempool *mp;
    struct rt_thread *thread;
    register rt_base_t level;

    /* 获取块所属的池的控制块 */
```

(下页继续)

(续上页)

```
    block_ptr = (rt_uint8_t **)((rt_uint8_t *)block - sizeof(rt_uint8_t *));
↪(2)
    mp    = (struct rt_mempool *)*block_ptr;                    (3)

    RT_OBJECT_HOOK_CALL(rt_mp_free_hook, (mp, block));

    /* 关中断 t */
    level = rt_hw_interrupt_disable();

    /* 增加可以的内存块数量 */
    mp->block_free_count ++;                                     (4)

    /* 将释放的内存块添加到 block_list 链表中 */
    *block_ptr = mp->block_list;                                 (5)
    mp->block_list = (rt_uint8_t *)block_ptr;                    (6)

    if (mp->suspend_thread_count > 0) {                          (7)
        /* 获取阻塞的线程 */
        thread = rt_list_entry(mp->suspend_thread.next,          (8)
                               struct rt_thread,
                               tlist);

        /* 重置线程错误为 RT_EOK */
        thread->error = RT_EOK;

        /* 恢复线程 */
        rt_thread_resume(thread);                                (9)

        /* 记录阻塞线程数量，减一 */
        mp->suspend_thread_count --;                             (10)

        /* 开中断 */
```

(下页继续)

(续上页)

```
        rt_hw_interrupt_enable(level);

        /* 发起线程调度 */
        rt_schedule();                                          (11)

        return;
    }

    /* 开中断 */
    rt_hw_interrupt_enable(level);
}
RTM_EXPORT(rt_mp_free);
```

代码清单 24-10 **(1)**：block 要释放的内存块的地址。

代码清单 24-10 **(2)**：每个内存块中前 4 个字节保存的信息就是指向内存池控制块指针，所以，需要进行指针的偏移，为了获得内存池控制块的地址。

代码清单 24-10 **(3)**：获取内存块所属的内存池对象 mp。

代码清单 24-10 **(4)**：记录当前可用内存块数量。

代码清单 24-10 **(5)**：将释放的内存块添加到 block_list 链表中，内存控制块的指向当前可用内存链表头具体过程见图 *24-10* **(1)**。

代码清单 24-10 **(6)**：内存控制块的 block_list 指向刚释放的内存块，具体见图 *24-10* _ **(2)**。

释放内存

代码清单 24-10 **(7)**：如果当前有线程因为无法申请内存进入阻塞的话，会执行 while 循环中的代码。

代码清单 24-10 **(8)**：获取阻塞的线程。

代码清单 24-10 **(9)**：调用 rt_ipc_list_resume 函数将该线程恢复。

代码清单 24-10 **(10)**：记录阻塞线程数量，suspend_thread_count 减一。

代码清单 24-10 **(11)**：恢复挂起的线程，需要发起一次线程调度。

内存释放的使用是非常很简单的，仅将需要释放的内存块地址传递进去即可，系统会根据内存块前 4 字节的内容自动找到对应的内存池控制块，然后根据内存池控制块来进行释放内存操作，具体见代码清单 24-11。

列表 11: 代码清单 24-11 静态内存释放函数 rt_mp_free() 实例

```
/* 定义申请内存的指针 */
static rt_uint32_t *p_test = RT_NULL;

rt_kprintf("正在释放内存...........\n");
rt_mp_free(p_test);
```

27.5 动态内存管理的函数接口讲解

动态内存使用，在 RT-Thread 运用极多，为了尽可能让 RT-Thread 易于使用，信号量、队列、互斥量、软件定时器、线程这些内核对象并不是在编译时静态分配的，而是在运行时动态分配的。内核对象创建时 RT-Thread 分配内存空间，在内核对象删除时释放内存。这样的策略减少了设计和计划上的努力，简化了 API，并且减少了 RAM 的占用，提高内存的利用率，更能灵活运用内存。

动态内存的典型场景开发流程：

1. 初始化系统堆内存空间：rt_system_heap_init()。
2. 申请任意大小的动态内存：rt_malloc()。
3. 释放动态内存 rt_free()。回收系统内存，供下一次使用。

27.5.1 系统堆内存初始化 rt_system_heap_init()

在使用堆内存时，必须要在系统初始化的时候进行堆内存的初始化，一般在系统初始化的时候就分配一大块内存作为堆内存，然后调用 rt_system_heap_init() 函数进行系统堆内存初始化，之后我们才能去申请内存，在初始化的时候需要用户自己知道初始化的是哪段内存，所以必须知道内存的起始地址与结束地址，这个函数会把参数 begin_addr，end_addr 区域的内存空间作为内存堆来使用，系统堆内存初始化 rt_system_heap_init() 源码具体见*代码清单 24-12*。

列表 12: 代码清单 24-12 系统堆内存初始化 rt_system_heap_init() 源码

```
void rt_system_heap_init(void *begin_addr, void *end_addr)          (1)
{
    struct heap_mem *mem;
    rt_uint32_t begin_align = RT_ALIGN((rt_uint32_t)begin_addr, RT_ALIGN_
↪SIZE); (2)
    rt_uint32_t end_align = RT_ALIGN_DOWN((rt_uint32_t)end_addr, RT_ALIGN_
↪SIZE); (3)

    RT_DEBUG_NOT_IN_INTERRUPT;

    /* 对齐地址 */
    if ((end_align > (2 * SIZEOF_STRUCT_MEM)) &&
        ((end_align - 2 * SIZEOF_STRUCT_MEM) >= begin_align)) {     (4)
        /* 计算对齐的内存大小 */
        mem_size_aligned = end_align - begin_align - 2 * SIZEOF_STRUCT_MEM;
    } else {
        rt_kprintf("mem init, error begin address 0x%x, and end address 0x
↪%x\n",
                   (rt_uint32_t)begin_addr, (rt_uint32_t)end_addr);

        return;
    }
```

(下页继续)

(续上页)

```
    /* 指向堆的起始地址 */
    heap_ptr = (rt_uint8_t *)begin_align;                                (5)

    RT_DEBUG_LOG(RT_DEBUG_MEM, ("mem init, heap begin address 0x%x, size %d\n",
                                (rt_uint32_t)heap_ptr, mem_size_aligned));

    /* 初始化起始地址 */
    mem        = (struct heap_mem *)heap_ptr;                            (6)
    mem->magic = HEAP_MAGIC;
    mem->next  = mem_size_aligned + SIZEOF_STRUCT_MEM;
    mem->prev  = 0;
    mem->used  = 0;
#ifdef RT_USING_MEMTRACE
    rt_mem_setname(mem, "INIT");
#endif

    /* 初始化结束地址 */
    heap_end        = (struct heap_mem *)&heap_ptr[mem->next];           (7)
    heap_end->magic = HEAP_MAGIC;
    heap_end->used  = 1;
    heap_end->next  = mem_size_aligned + SIZEOF_STRUCT_MEM;
    heap_end->prev  = mem_size_aligned + SIZEOF_STRUCT_MEM;
#ifdef RT_USING_MEMTRACE
    rt_mem_setname(heap_end, "INIT");
#endif

    rt_sem_init(&heap_sem, "heap", 1, RT_IPC_FLAG_FIFO);                 (8)

    /* 初始化指向堆起始的最低空闲指针 */
    lfree = (struct heap_mem *)heap_ptr;                                 (8)
```

(下页继续)

(续上页)

```
}
```

代码清单 24-12 **(1)**：begin_addr 内存的起始地址，end_addr 结束地址，这个函数会把参数 begin_addr，end_addr 区域的内存空间作为堆内存来使用。

代码清单 24-12 **(2)**：起始地址对齐，按 4 字节对齐，其地址要能被 4 整除，如果不对齐会向下进行对齐，例如 RT_ALIGN（13，4）会将其地址改为 16。

代码清单 24-12 **(3)**：：结束地址对齐，按 4 字节对齐，其地址要能被 4 整除，如果不对齐会向上进行对齐，例如 RT_ALIGN_DOWN（13，4）会将其地址改为 12。

代码清单 24-12 **(4)**：如果对齐后的内存大于两个数据头，则此内存是有效的，可以进行初始化内存，数据头是每个动态分配的内存块都包含的一个结构体，与静态内存的前 4 字节内容一样，用于保存内存块的信息，内存管理器能根据数据头进行内存的释放与回收，其数据类型具体见代码清单 24-13。

列表 13: 代码清单 24-13 内存管理的数据头 heap_mem

```
struct heap_mem {

    rt_uint16_t magic;              (1)
    rt_uint16_t used;                       (2)

    rt_size_t next, prev;           (3)

#ifdef RT_USING_MEMTRACE
    rt_uint8_t thread[4];   /* thread name */
#endif
};
```

代码清单 24-13 **(1)**：magic 变数（或称为幻数），它会被初始化成 0x1ea0（即英文单词 heap），用于标记这个内存块是一个内存管理用的内存数据块。

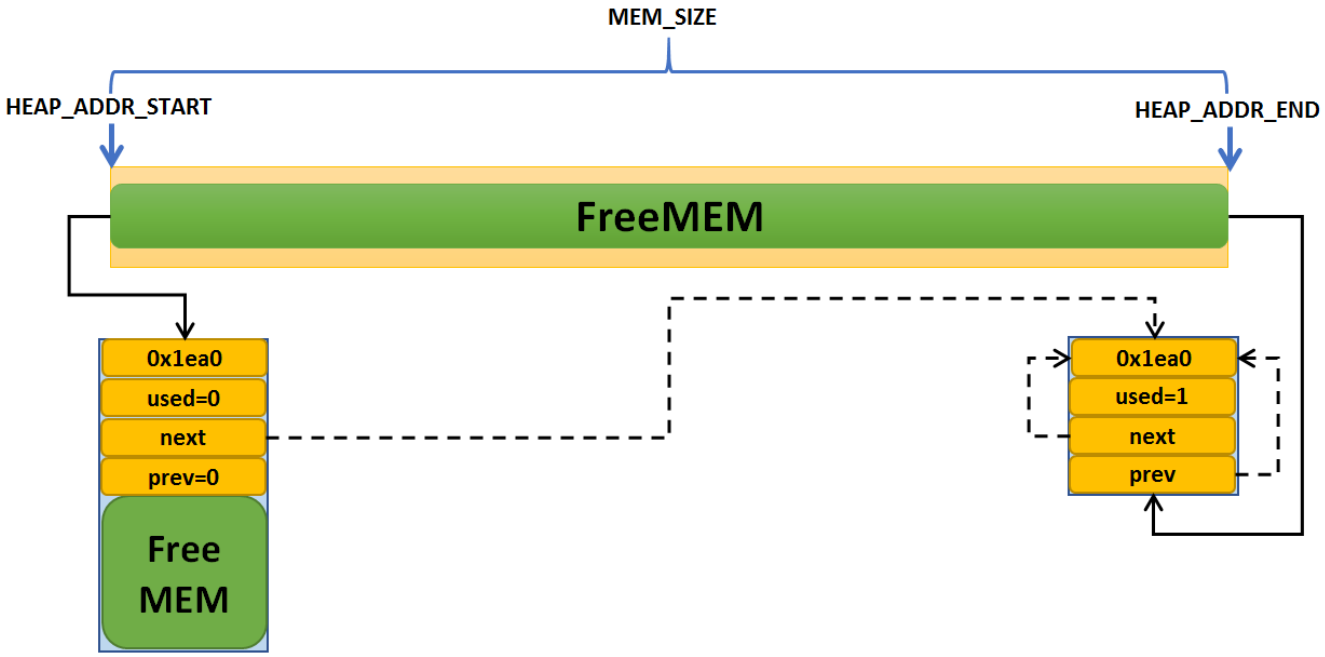
代码清单 24-13 **(2)**：used 指示出当前内存块是否已经分配，1 代表内存已经分配了，0 代表内存可用。

代码清单 24-13 **(3)**：两个指针，用于将内存块形成双线链表，便于管理，具体见图 *24-3*。

代码清单 24-12 **(5)**：获取地址 heap_ptr，指向堆内存的起始地址。

代码清单 24-12 **(6)**：初始化起始地址数据头，magic 初始化成 0x1ea0（即英文单词 heap），mem->next 下一个内存块指向结束地址的数据头，当前内存没有被分割，只有从起始地址到结束地址的一整块内存。

代码清单 24-12 **(7)**：同理一样初始化结束地址数据头，因为结束地址之后没有内存了，所以 used 的值要为 1，表示在这个地址之后没有内存空间可以分配了，具体见图 *24-11*。



代码清单 24-12 **(8)**：初始化一个二值信号量，因为申请内存需要进行资源保护，总不能这个线程在申请内存的时候被另一个线程打断，这样子就乱套了。

初始化系统内存的使用是很简单的，一般系统在初始化的时候就已经将内存初始化完成了，并不需要我们再次初始化，在 rt_hw_board_init 函数中（board.c 文件）已经进行初始化了，当然我们也能从新将内存初始化一次。具体见代码清单 24-14 高亮部分。

列表 14: 代码清单 24-14 系统堆内存初始化 rt_system_heap_init() 实例

```
#define RT_HEAP_SIZE 1024
/* 从内部 SRAM 里面分配一部分静态内存来作为 rtt 的堆空间，这里配置为 4KB */
static uint32_t rt_heap[RT_HEAP_SIZE];
RT_WEAK void *rt_heap_begin_get(void)
{
    return rt_heap;
}

RT_WEAK void *rt_heap_end_get(void)
{
    return rt_heap + RT_HEAP_SIZE;
}

rt_system_heap_init(rt_heap_begin_get(),rt_heap_end_get());
```

27.5.2 系统堆内存申请函数 rt_malloc()

rt_malloc 函数会从系统堆空间中找到合适用户指定大小的内存块，然后把该内存块可用地址返回给用户，rt_malloc 函数的源码实现具体见*代码清单 24-15*。

列表 15: 代码清单 24-15 系统堆内存申请函数 rt_malloc() 源码

```
void *rt_malloc(rt_size_t size)                                    // (1)
{
    rt_size_t ptr, ptr2;
    struct heap_mem *mem, *mem2;

```

(下页继续)

(续上页)

```
    RT_DEBUG_NOT_IN_INTERRUPT;

    if (size == 0)
        return RT_NULL;

    if (size != RT_ALIGN(size, RT_ALIGN_SIZE))
        RT_DEBUG_LOG(RT_DEBUG_MEM, ("malloc size %d, but align to %d\n",
↪size, RT_ALIGN(size, RT_ALIGN_SIZE)));
    else
        RT_DEBUG_LOG(RT_DEBUG_MEM, ("malloc size %d\n", size));

    /* 对齐内存 */
    size = RT_ALIGN(size, RT_ALIGN_SIZE);                          // (2)

    if (size > mem_size_aligned) {                                 // (3)
        RT_DEBUG_LOG(RT_DEBUG_MEM, ("no memory\n"));

        return RT_NULL;
    }

    /* 每个数据块的长度必须至少为 MIN_SIZE_ALIGNED  */
    if (size < MIN_SIZE_ALIGNED)                                   // (4)
        size = MIN_SIZE_ALIGNED;

    /* 获取信号量 */
    rt_sem_take(&heap_sem, RT_WAITING_FOREVER);                    // (5)

    for (ptr = (rt_uint8_t *)lfree - heap_ptr;
        ptr < mem_size_aligned - size;
        ptr = ((struct heap_mem *)&heap_ptr[ptr])->next) {         // (6)
        mem = (struct heap_mem *)&heap_ptr[ptr];

```

(下页继续)

(续上页)

```
        if ((!mem->used) && (mem->next - (ptr + SIZEOF_STRUCT_MEM)) >=␣
↪size) {//(7)
            /* 该内存没有被使用并且可以满足用户的需要申请的内存大小 */

            if (mem->next - (ptr + SIZEOF_STRUCT_MEM) >=
                (size + SIZEOF_STRUCT_MEM + MIN_SIZE_ALIGNED)) {   // (8)

    / * (除了上面的内容，我们测试是否包含另一个 struct heap_mem SIZEOF_STRUCT_MEM)
        * 至少 MIN_SIZE_ALIGNED 数据也适合用户数据空间 mem 的大小 )
        * 拆分大块，创建空余数，余数必须足够大才能包含 MIN_SIZE_ALIGNED 数据:
        * if mem-> next - (ptr + (2 * SIZEOF_STRUCT_MEM)) == size,
        * struct heap_mem 适合但 mem2 和 mem2-> next 之间没有数据
        * 我们可以省略 MIN_SIZE_ALIGNED。 我们会创建一个空的内存块
        * 虽然无法保存数据的区域，但是当 mem-> next 被释放时，两个区域将合并，从而产生
更多的可用内存
                ptr2 = ptr + SIZEOF_STRUCT_MEM + size;             // (9)

                /* 创建一个数据头结构体 */
                mem2       = (struct heap_mem *)&heap_ptr[ptr2];   // (10)
                mem2->magic = HEAP_MAGIC;
                mem2->used = 0;
                mem2->next = mem->next;                            // (11)
                mem2->prev = ptr;                                  // (12)
#ifdef RT_USING_MEMTRACE
                rt_mem_setname(mem2, "    ");
#endif

                /* 并将其插入 mem 和 mem-> next 之间 */
                mem->next = ptr2;
                mem->used = 1;                                     // (13)

        if (mem2->next != mem_size_aligned + SIZEOF_STRUCT_MEM) {
```

(下页继续)

(续上页)

```
            ((struct heap_mem *)&heap_ptr[mem2->next])->prev = ptr2;
                }
#ifdef RT_MEM_STATS
                used_mem += (size + SIZEOF_STRUCT_MEM);            // (14)
                if (max_mem < used_mem)
                    max_mem = used_mem;
#endif
            } else {
            /* (mem2 结构不适合下次用户申请的数据空间大小，此时将始终使用:
            * 如果不是，我们连续有 2 个未使用的结构，在之前就处理了这种情况
            * 当前内存块是最适合用户申请的内存的，直接分配即可。
            *
            * 不进行分裂，没有 mem2 创建也无法移动 mem
            * 接下来直接在 mem 后面，因为 mem-> next 将始终用于此点!
            */
                mem->used = 1;                                     // (15)
#ifdef RT_MEM_STATS
            used_mem += mem->next - ((rt_uint8_t *)mem - heap_ptr);// (16)
            if (max_mem < used_mem)
                max_mem = used_mem;
#endif
            }
            /* 设置内存块数据头的变幻数 */
            mem->magic = HEAP_MAGIC;                               // (17)
#ifdef RT_USING_MEMTRACE
            if (rt_thread_self())
                rt_mem_setname(mem, rt_thread_self()->name);
            else
                rt_mem_setname(mem, "NONE");
#endif

            if (mem == lfree) {                                    // (18)
```

(下页继续)

(续上页)

```
                /* 在 mem 之后找到下一个空闲块并更新最低空闲指针 */
                while (lfree->used && lfree != heap_end)            // (19)
                    lfree = (struct heap_mem *)&heap_ptr[lfree->next];

                RT_ASSERT(((lfree == heap_end) || (!lfree->used)));
            }

            rt_sem_release(&heap_sem);
            RT_ASSERT((rt_uint32_t)mem + SIZEOF_STRUCT_MEM + size <= (rt_uint32_t)heap_end);
            RT_ASSERT((rt_uint32_t)((rt_uint8_t *)mem +
            SIZEOF_STRUCT_MEM) % RT_ALIGN_SIZE == 0);
            RT_ASSERT((((rt_uint32_t)mem) & (RT_ALIGN_SIZE - 1)) == 0);
            RT_DEBUG_LOG(RT_DEBUG_MEM,("allocate memory at 0x%x, size: %d\n",(rt_uint32_t)((rt_uint8_t *)mem + SIZEOF_STRUCT_MEM),(rt_uint32_t)(mem->next - ((rt_uint8_t *)mem - heap_ptr))));
            RT_OBJECT_HOOK_CALL(rt_malloc_hook,(((void *)((rt_uint8_t *)mem + SIZEOF_STRUCT_MEM)), size));

            /*  返回除内存块数据头结构之外的内存数据 */
            return (rt_uint8_t *)mem + SIZEOF_STRUCT_MEM;           // (20)
        }
    }

    rt_sem_release(&heap_sem);                                      // (21)

    return RT_NULL;
}
RTM_EXPORT(rt_malloc);
```

代码清单 24-15 **(1)**：size 表示申请多大的内存块，单位为字节。

代码清单 24-15 **(2)**：初始化 size，配置 size 以 4 字节对齐，使其对齐方式与系统内存对齐方式一

致。

代码清单 *24-15* **(3)**：如果 size 大于当前系统管理的最大空闲内存，内存不足，返回错误。

代码清单 *24-15* **(4)**：每个内存块的大小必须至少为 MIN_SIZE_ALIGNED，否则连内存块的数据头部分都放不下，更别谈放下我们的数据了。

代码清单 *24-15* **(5)**：获取信号量，此信号量是一个二值信号量，用于对内存资源的保护，当一个线程申请内存的时候，其他线程就不能申请，否则内存就会变得很混乱。

代码清单 *24-15* **(6)**：在 for 循环中遍历寻找合适的内存资源。

代码清单 *24-15* **(7)**：该内存没有被使用并且可以满足用户的需要申请的内存大小，表示已经找到了合适用户申请的大小的内存。

代码清单 *24-15* **(8)**：当内存满足用户需要并且在分割后剩下的内存块也适合存放数据，那么则进行内存块分割。说白了就是找到的这块内存很大，给用户申请了一部分，剩下的内存块也能存放数据，那就把剩下的内存也利用起来，进行内存分割。

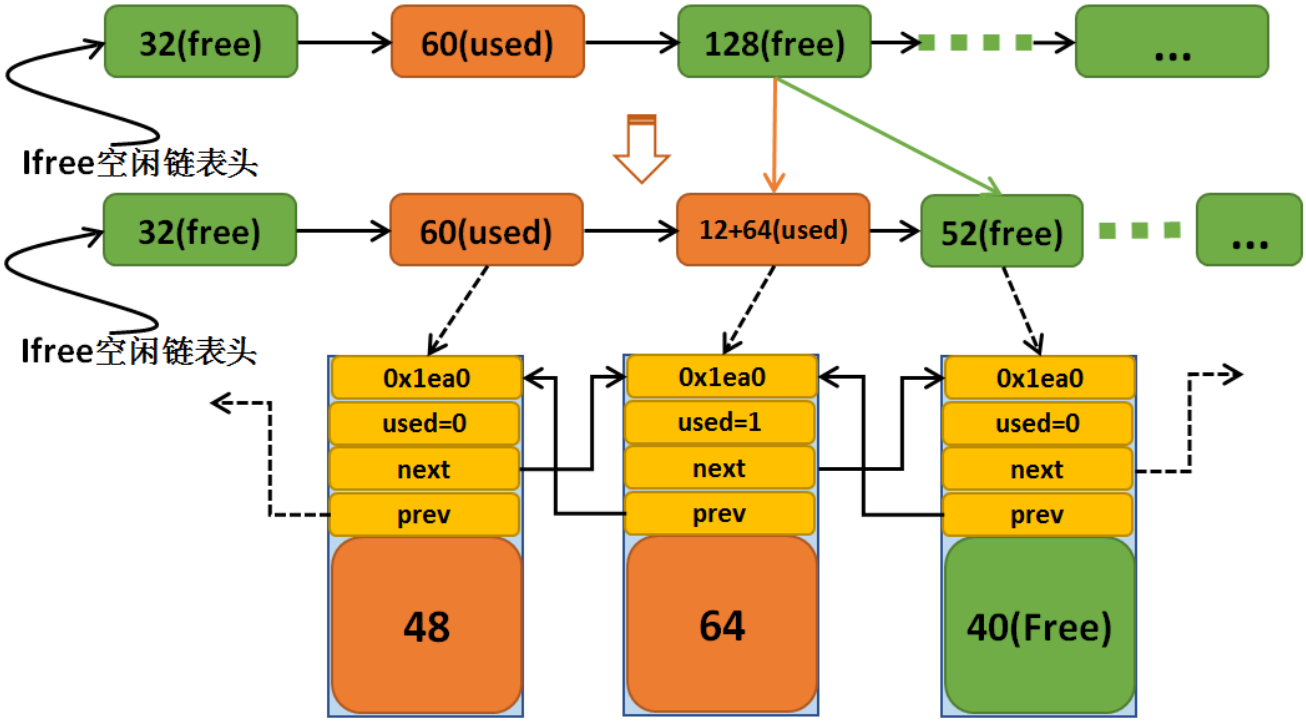
代码清单 *24-15* **(9)**：获取分割后的空闲内存块地址 ptr2。它的起始地址就是当前内存块地址 ptr+12 字节的内存块数据头 + 用户申请的内存块大小 size。

代码清单 *24-15* **(10)**：为分割后的空闲内存块创建一个数据头。在 RT-Thread 中，不管是已使用还是空闲的内存块都要有数据头，因为这样子便于管理、申请与释放，其过程见图 *24-12*。

代码清单 *24-15* **(11)**：ptr2 是空闲内存块，used 为 0，它的 next 指针指向下一个内存块，也就是本内存块没分割之前的下一个的内存块，其过程见图 *24-12*。

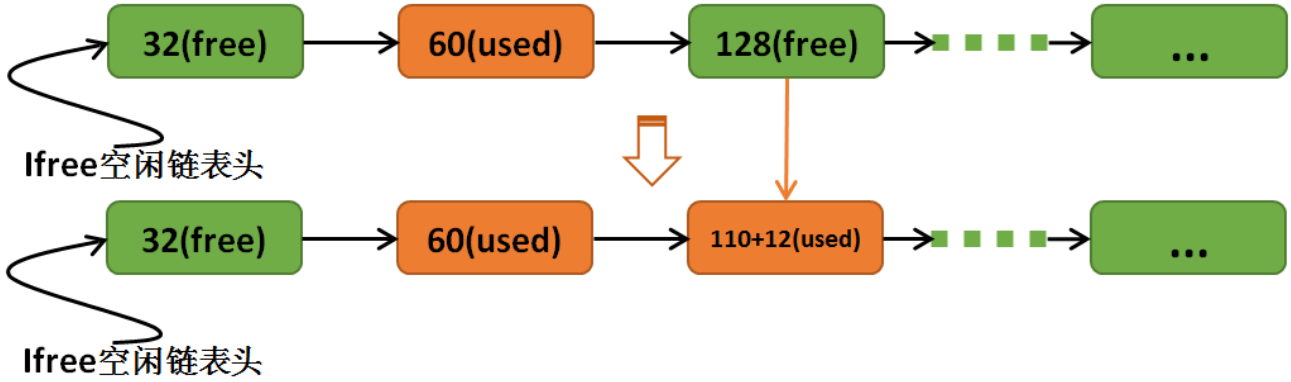
代码清单 *24-15* **(12)**：很显然 ptr2 的上一个内存块就是当前申请的内存块，利用双向链表将内存块连接起来。

代码清单 *24-15* **(13)**：申请的内存块 used 设置 1 表示已使用，具体见图 *24-12*。



代码清单 24-15 **(14)**：计算得出当前内存块大小为用户要申请的内存大小 size+ 内存块中数据头大小（12 字节）。

代码清单 24-15 **(15)**：else 里面的内容是不进行内存块分割，因为剩下的内存块太小，都无法保存数据，没必要进行内存块分割了，直接将当前内存块作为已使用的内存块即可，具体见图 *24-13*。



代码清单 24-15 **(16)**：计算得到使用的内存块大小。

代码清单 24-15 **(17)**：设置内存块数据头的变幻数为 0x1ea0（即英文单词 heap）。

代码清单 *24-15* **(18)**：如果当前申请的内存块是 lfree 指向的内存块，那么现在申请成功了，内存块肯定不是空闲的内存块，需要重新更新一下 lfree 的指针。

代码清单 *24-15* **(19)**：找到下一个空闲块并更新最小空闲内存块指针。

代码清单 *24-15* **(20)**：返回用户需要的内存地址，因为数据头是内存管理器处理的地方，无需用户理会的，同时用户也不应该去修改数据头的内容。

代码清单 *24-15* **(21)**：申请完成，释放二值信号量，让别的线程也能申请内存。

申请内存的源码其实不要求用户很清楚，因为这是内存管理器要做的事情，我们需要注意的是在用户用完内存的时候将内存释放掉即可，别看内存申请过程那么复杂，其实使用起来是很简单的，用户需要定义一个指针，因为申请内存返回的是内存块的地址，系统堆内存申请函数 rt_malloc() 实例具体见代码清单 *24-16* 高亮部分。

列表 16: 代码清单 24-16 系统堆内存申请函数 rt_malloc() 实例

```
/* 定义申请内存的指针 */
static rt_uint32_t *p_test = RT_NULL;

rt_kprintf("正在向内存池申请内存...........\n");
p_test = rt_malloc(TEST_SIZE);     /* 申请动态内存 */
if (RT_NULL == p_test) /* 没有申请成功 */
    rt_kprintf("动态内存申请失败! \n");
else
    rt_kprintf("动态内存申请成功，地址为%d! \n\n",p_test);

rt_kprintf("正在向 p_test 写入数据...........\n");
*p_test = 1234;
rt_kprintf("已经写入 p_test 地址的数据\n");
rt_kprintf("*p_test = %.4d , 地址为:%d \n\n", *p_test,p_test);
```

27.5.3 系统堆内存释放函数 rt_free()

嵌入式系统的内存对我们来说是十分珍贵的，当不使用的时候就应该把内存释放出来，不然很容易造成内存不足的问题，导致系统发生致命错误。RT-Thread 提供了 rt_free() 函数进行动态内存的释放管理，rt_free() 函数会把待释放的内存还回给堆管理器中。在调用这个函数时用户需传递待释放的内存块指针，如果是空指针直接返回，其源码具体见*代码清单 24-17*。

列表 17: 代码清单 24-17 系统堆内存释放函数 rt_free() 源码

```
/**
*   此函数将释放先前利用 rt_malloc 分配的内存块
* 释放的内存块将被恢复到系统堆。
```

(下页继续)

(续上页)

```
 *
 * @param rmem 即将释放的内存块指针
 */
void rt_free(void *rmem)                                                    (1)
{
    struct heap_mem *mem;

    RT_DEBUG_NOT_IN_INTERRUPT;

    if (rmem == RT_NULL)                                                    (2)
        return;
    RT_ASSERT((((rt_uint32_t)rmem) & (RT_ALIGN_SIZE - 1)) == 0);
    RT_ASSERT((rt_uint8_t *)rmem >= (rt_uint8_t *)heap_ptr &&
            (rt_uint8_t *)rmem < (rt_uint8_t *)heap_end);

    RT_OBJECT_HOOK_CALL(rt_free_hook, (rmem));

    if ((rt_uint8_t *)rmem < (rt_uint8_t *)heap_ptr ||
        (rt_uint8_t *)rmem >= (rt_uint8_t *)heap_end) {                     (3)
        RT_DEBUG_LOG(RT_DEBUG_MEM, ("illegal memory\n"));

        return;
    }

    /* 获取相应的 heap_mem 结构体... */
    mem = (struct heap_mem *)((rt_uint8_t *)rmem - SIZEOF_STRUCT_MEM); (4)

    RT_DEBUG_LOG(RT_DEBUG_MEM,
                ("release memory 0x%x, size: %d\n",
                (rt_uint32_t)rmem,
            (rt_uint32_t)(mem->next - ((rt_uint8_t *)mem - heap_ptr))));

```

(下页继续)

(续上页)

```
    /* 获取信号量，保护堆免受并发访问 */
    rt_sem_take(&heap_sem, RT_WAITING_FOREVER);                    (5)

    /* ...... 必须处于使用状态...... */
    if (!mem->used || mem->magic != HEAP_MAGIC) {                  (6)
        rt_kprintf("to free a bad data block:\n");
        rt_kprintf("mem: 0x%08x, used flag: %d, magic code: 0x%04x\n",
                mem, mem->used, mem->magic);
    }
    RT_ASSERT(mem->used);
    RT_ASSERT(mem->magic == HEAP_MAGIC);

    mem->used  = 0;                                                (7)
    mem->magic = HEAP_MAGIC;
#ifdef RT_USING_MEMTRACE
    rt_mem_setname(mem, "    ");
#endif

    if (mem < lfree) {                                             (8)
        /* 新释放的内存大小现在是最小的 */
        lfree = mem;
    }

#ifdef RT_MEM_STATS
    used_mem -= (mem->next - ((rt_uint8_t *)mem - heap_ptr));
#endif

    /* 最后，看看 prev 与 next 也是不是空闲的，看看是否能合并 */
    plug_holes(mem);                                               (9)
    rt_sem_release(&heap_sem);                                     (10)
}
```

(下页继续)

(续上页)

```
RTM_EXPORT(rt_free);
```

代码清单 24-17 **(1)**：rmem 即将释放的内存块指针，由用户传递进来。

代码清单 24-17 **(2)**：检查内存块指针是否有效，如果无效，直接退出释放函数。

代码清单 24-17 **(3)**：检查 rmem 的地址是否属于系统管理的内存范围，如果 rmem 地址比系统管理的起始地址还小或者比系统管理的结束地址还大，那么肯定是地址非法了，打印出错误信息，退出释放函数。

代码清单 24-17 **(4)**：根据 rmem 偏移，获取内存块中数据头信息，这样子做的好处是可用知道要释放的内存是不是由内存管理器管理的，所以用户一般不允许改变数据头的内容。

代码清单 24-17 **(5)**：获取信号量，保护堆免受并发访问。

代码清单 24-17 **(6)**：如果需要释放的内存块还不是使用的状态，那么无需释放，或者内存块中变幻数 magic 不是 HEAP_MAGIC，那么也不能释放该内存。

代码清单 24-17 **(7)**：释放内存，将 used 变为 0，表明内存未使用，但是注意了，该内存的真正数据是没有释放的，used 为 0 只是表明该内存块能被申请而已。

代码清单 24-17 **(8)**：新释放的内存大小现在是最小的，那么 lfree 必须要指向刚释放的内存块。

代码清单 24-17 **(9)**：最后，调用 plug_holes() 函数看看释放的内存块相邻的两个内存块也是不是空闲的，看看是否能合并成一个大的内存块。

代码清单 24-17 **(10)**：释放信号量，保证别的线程能释放内存。

内存释放函数的使用是很简单，一般来说，只要用户传递正确的内存块指针即可，而且，在使用完内存的时候一定要及时释放内存，提高内存的利用，系统堆内存释放函数 rt_free() 实例具体见代码清单 24-18 高亮部分。

列表 18: 代码清单 24-18 系统堆内存释放函数 rt_free() 实例

```
/* 定义申请内存的指针 */
static rt_uint32_t *p_test = RT_NULL;
```

(下页继续)

(续上页)

```

rt_kprintf("正在向内存池申请内存.....\n");
p_test = rt_malloc(TEST_SIZE);     /* 申请动态内存 */
if (RT_NULL == p_test) /* 没有申请成功 */
    rt_kprintf("动态内存申请失败! \n");
else
    rt_kprintf("动态内存申请成功，地址为%d! \n\n",p_test);

rt_kprintf("正在向 p_test 写入数据.....\n");
*p_test = 1234;
rt_kprintf("已经写入 p_test 地址的数据\n");
rt_kprintf("*p_test = %.4d , 地址为:%d \n\n", *p_test,p_test);

rt_kprintf("正在释放内存.....\n");
rt_free(p_test);
rt_kprintf("释放内存成功! \n\n");
```

动态内存的使用有几点要注意的地方：

- 由于系统中动态内存管理需要消耗管理控制块结构的内存，故实际用户可使用空间总量小于堆内存的实际大小，假设以堆内存的 begin_addr 作为起始地址，end_addr 作为结束地址，那么实际内存大小应为（end_addr - begin_addr），而用户不可能完全利用到那么多内存的，因为内存管理器也是要用内存的。
- 系统中地址为了对齐可能会丢弃部分空间，故存在一些内存碎片。
- 系统中进行内存释放时调用 rt_free() 函数，只有在内存还没释放的时候才能进行释放并且返回成功，当内存已经被释放掉了还继续调用 rt_free() 会提示出错。

27.6 内存管理的实验

27.6.1 静态内存管理实验

静态内存管理实验是在 RT-Thread 中创建了两个线程，其中一个线程是申请内存，另一个线程是清除内存块中的内容以及释放内存。划分静态内存池区域可以通过定义全局数组或调用动态内存分配接口方式获取。在不需内存时，注意要及时释放该段内存，避免内存泄露。具体见代码清单 24-19 高亮部分。

列表 19: 代码清单 24-19 静态内存管理实验

```
/**
*********************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2019-xx-xx
* @brief   RT-Thread 静态内存管理
*********************************************************************
* @attention
*
* 实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
**********************************************************************
*/

/*
***********************************************************************
*                              包含的头文件
***********************************************************************
```

(下页继续)

(续上页)

```
22 */
23 #include "board.h"
24 #include "fsl_debug_console.h"
25 /* RT-Thread 头文件 */
26 #include "rtthread.h"
27 /*
28 ****************************************************************
29 *                                 变量
30 ****************************************************************
31 */
32 /* 定义线程控制块 */
33 static rt_thread_t alloc_thread = RT_NULL;
34 static rt_thread_t free_thread = RT_NULL;
35 /* 定义内存池控制块 */
36 static rt_mp_t test_mp = RT_NULL;
37 /* 定义申请内存的指针 */
38 static rt_uint32_t *p_test = RT_NULL;
39
40
41 /************************* 全局变量声明 ****************************/
42 /*
43 * 当我们在写应用程序的时候，可能需要用到一些全局变量。
44 */
45
46 /* 相关宏定义 */
47 #define  BLOCK_COUNT   20            //内存块数量
48 #define  BLOCK_SIZE    3               //内存块大小
49
50
51 /*
52 ********************************************************************
53 *                                 函数声明
```

(下页继续)

(续上页)

```
*****************************************************************************
*/
static void alloc_thread_entry(void* parameter);
static void free_thread_entry(void* parameter);

/*
*****************************************************************************
*                                main 函数
*****************************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]- i.MX RT 全系列开发板-RTT 静态内存管理实验！\n");
    rt_kprintf("正在创建一个内存池..........\n");
    /* 创建一个静态内存池 */
    test_mp = rt_mp_create("test_mp",
                           BLOCK_COUNT,
                           BLOCK_SIZE);
    if (test_mp != RT_NULL)
        rt_kprintf("静态内存池创建成功！\n\n");

    /* 创建一个线程 */
```

(下页继续)

(续上页)

```
    alloc_thread =                          /* 线程控制块指针 */
        rt_thread_create( "alloc",              /* 线程名字 */
                          alloc_thread_entry,   /* 线程入口函数 */
                          RT_NULL,              /* 线程入口函数参数 */
                          512,                  /* 线程栈大小 */
                          1,                    /* 线程的优先级 */
                          20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (alloc_thread != RT_NULL)
        rt_thread_startup(alloc_thread);
    else
        return -1;

    free_thread =                           /* 线程控制块指针 */
        rt_thread_create( "free",               /* 线程名字 */
                          free_thread_entry,    /* 线程入口函数 */
                          RT_NULL,              /* 线程入口函数参数 */
                          512,                  /* 线程栈大小 */
                          2,                    /* 线程的优先级 */
                          20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (free_thread != RT_NULL)
        rt_thread_startup(free_thread);
    else
        return -1;
}

/*
*****************************************************************
*                              线程定义
```

(下页继续)

(续上页)

```
*****************************************************************
*/

static void alloc_thread_entry(void* parameter)
{
    rt_kprintf("正在向内存池申请内存...........\n");

    p_test = rt_mp_alloc(test_mp,0);
    if (RT_NULL == p_test) /* 没有申请成功 */
        rt_kprintf("静态内存申请失败! \n");
    else
        rt_kprintf("静态内存申请成功, 地址为%d! \n\n",p_test);

    rt_kprintf("正在向 p_test 写入数据...........\n");
    *p_test = 1234;
    rt_kprintf("已经写入 p_test 地址的数据\n");
    rt_kprintf("*p_test = %.4d , 地址为:%d \n\n", *p_test,p_test);

    /* 线程都是一个无限循环, 不能返回 */
    while (1) {
        LED2_TOGGLE;
        rt_thread_delay(1000);      //每 1000ms 扫描一次
    }
}

static void free_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
    rt_kprintf("正在释放内存...........\n");
    rt_mp_free(p_test);
    rt_kprintf("释放内存成功! \n\n");
    rt_kprintf("正在删除内存...........\n");
```

(下页继续)

(续上页)

```
    uwRet = rt_mp_delete(test_mp);
    if (RT_EOK == uwRet)
        rt_kprintf("删除内存成功! \n");
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        LED1_TOGGLE;
        rt_thread_delay(500);      //每 500ms 扫描一次
    }
}
/****************************END OF FILE****************************/
```

27.6.2 动态内存管理实验

动态内存管理实验是在 RT-Thread 中创建了两个线程，其中一个线程是申请内存，另一个线程是清除内存块中的内容以及释放内存。在不需内存时，注意要及时释放该段内存，避免内存泄露，具体见*代码清单 24-20* 高亮部分。

列表 20: 代码清单 24-20 动态内存管理实验

```
/**
*******************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2019-xx-xx
* @brief   RT-Thread 动态内存管理
*******************************************************************
* @attention
*
* 实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
```

(下页继续)

(续上页)

```
* 淘宝     :https://fire-stm32.taobao.com
*
*****************************************************************
*/

/*
*********************************************************************
*                              包含的头文件
*********************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"
/* RT-Thread 头文件 */
#include "rtthread.h"
/*
*****************************************************************
*                              变量
*****************************************************************
*/
/* 定义线程控制块 */
static rt_thread_t alloc_thread = RT_NULL;
static rt_thread_t free_thread = RT_NULL;

/* 定义申请内存的指针 */
static rt_uint32_t *p_test = RT_NULL;

/************************* 全局变量声明 ****************************/
/*
* 当我们在写应用程序的时候，可能需要用到一些全局变量。
*/

/* 相关宏定义 */
```

(下页继续)

(续上页)

```
#define  TEST_SIZE   100              //内存大小（字节）


/*
*************************************************************************
*                              函数声明
*************************************************************************
*/
static void alloc_thread_entry(void* parameter);
static void free_thread_entry(void* parameter);

/*
*************************************************************************
*                              main 函数
*************************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]- i.MX RT 全系列开发板-RTT 动态内存管理实验！\n");

    /* 创建一个线程 */
    alloc_thread =                          /* 线程控制块指针 */
```

(下页继续)

(续上页)

```
        rt_thread_create( "alloc",                /* 线程名字 */
                          alloc_thread_entry,    /* 线程入口函数 */
                          RT_NULL,              /* 线程入口函数参数 */
                          512,                  /* 线程栈大小 */
                          1,                    /* 线程的优先级 */
                          20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (alloc_thread != RT_NULL)
        rt_thread_startup(alloc_thread);
    else
        return -1;

    free_thread =                            /* 线程控制块指针 */
        rt_thread_create( "free",                 /* 线程名字 */
                          free_thread_entry,    /* 线程入口函数 */
                          RT_NULL,              /* 线程入口函数参数 */
                          512,                  /* 线程栈大小 */
                          2,                    /* 线程的优先级 */
                          20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (free_thread != RT_NULL)
        rt_thread_startup(free_thread);
    else
        return -1;
}

/*
*****************************************************************
*                              线程定义
*****************************************************************
```

(下页继续)

(续上页)

```
*/

static void alloc_thread_entry(void* parameter)
{
    rt_kprintf("正在向内存池申请内存...........\n");
    p_test = rt_malloc(TEST_SIZE);     /* 申请动态内存 */
    if (RT_NULL == p_test) /* 没有申请成功 */
        rt_kprintf("动态内存申请失败! \n");
    else
        rt_kprintf("动态内存申请成功, 地址为%d! \n\n",p_test);

    rt_kprintf("正在向 p_test 写入数据...........\n");
    *p_test = 1234;
    rt_kprintf("已经写入 p_test 地址的数据\n");
    rt_kprintf("*p_test = %.4d , 地址为:%d \n\n", *p_test,p_test);

    /* 线程都是一个无限循环, 不能返回 */
    while (1) {
        LED2_TOGGLE;
        rt_thread_delay(1000);      //每 1000ms 扫描一次
    }
}

static void free_thread_entry(void* parameter)
{
    rt_kprintf("正在释放内存...........\n");
    rt_free(p_test);
    rt_kprintf("释放内存成功! \n\n");

    /* 线程都是一个无限循环, 不能返回 */
    while (1) {
        LED1_TOGGLE;
```

(下页继续)

(续上页)

```
        rt_thread_delay(500);      //每 500ms 扫描一次
    }
}
/***************************END OF FILE****************************/
```

27.7 内存管理的实验现象

27.7.1 静态内存管理实验现象

程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息与运行结果，开发板的 LED 也在闪烁，具体见图 *24-14*。

27.7.2 静态内存管理实验现象

程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息与运行结果，开发板的 LED 也在闪烁，具体见图 *24-15*。

第 28 章　中断管理

28.1 异常与中断的基本概念

异常是导致处理器脱离正常运行转向执行特殊代码的任何事件，如果不及时进行处理，轻则系统出错，重则会导致系统毁灭性瘫痪。所以正确地处理异常，避免错误的发生是提高软件鲁棒性（稳定性）非常重要的一环，对于实时系统更是如此。

异常是指任何打断处理器正常执行，并且迫使处理器进入一个由有特权的特殊指令执行的事件。异常通常可以分成两类：同步异常和异步异常。由内部事件（像处理器指令运行产生的事件）引起的异常称为同步异常，例如造成被零除的算术运算引发一个异常，又如在某些处理器体系结构中，对于确定的数据尺寸必须从内存的偶数地址进行读和写操作。从一个奇数内存地址的读或写操作将引起存储器存取一个错误事件并引起一个异常，（称为校准异常）。

异步异常主要是指由于外部异常源产生的异常，是一个由外部硬件装置产生的事件引起的异步异常。同步异常不同于异步异常的地方是事件的来源，同步异常事件是由于执行某些指令而从处理器内部产生的，而异步异常事件的来源是外部硬件装置。例如按下设备某个按钮产生的事件。同步异常与异步异常的区别还在于，同步异常触发后，系统必须立刻进行处理而不能够依然执行原有的程序指令步骤；而异步异常则可以延缓处理甚至是忽略，例如按键中断异常，虽然中断异常触发了，但是系统可以忽略它继续运行（同样也忽略了相应的按键事件）。

中断，中断属于异步异常。所谓中断是指中央处理器 CPU 正在处理某件事的时候，外部发生了某一事件，请求 CPU 迅速处理，CPU 暂时中断当前的工作，转入处理所发生的事件，处理完后，再回到原来被中断的地方，继续原来的工作，这样的过程称为中断。

中断能打断线程的运行，无论该线程具有什么样的优先级，因此中断一般用于处理比较紧急的事件，而且只做简单处理，例如标记该事件，在使用 RT-Thread 系统时，一般建议使用信号量、消息或事件标志组等标志中断的发生，将这些内核对象发布给处理线程，处理线程再做具体处理。

通过中断机制，在外设不需要 CPU 介入时，CPU 可以执行其它线程，而当外设需要 CPU 时通过产生中断信号使 CPU 立即停止当前线程转而来响应中断请求。这样可以使 CPU 避免把大量时间

耗费在等待、查询外设状态的操作上，因此将大大提高系统实时性以及执行效率。

此处读者要知道一点，RT-Thread 源码中有许多处临界段的地方，临界段虽然保护了关键代码的执行不被打断，但也会影响系统的实时，任何使用了操作系统的中断响应都不会比裸机快。比如，某个时候有一个线程在运行中，并且该线程部分程序将中断屏蔽掉，也就是进入临界段中，这个时候如果有一个紧急的中断事件被触发，这个中断就会被挂起，不能得到及时响应，必须等到中断开启才可以得到响应，如果屏蔽中断时间超过了紧急中断能够容忍的限度，危害是可想而知的。所以，操作系统的中断在某些时候会有适当的中断延迟，因此调用中断屏蔽函数进入临界段的时候，也需快进快出。

RT-Thread 的中断管理支持：

- 开/关中断。
- 恢复中断。
- 中断使能。
- 中断屏蔽。

28.1.1 中断的介绍

与中断相关的硬件可以划分为三类：外设、中断控制器、CPU 本身。

外设：当外设需要请求 CPU 时，产生一个中断信号，该信号连接至中断控制器。

中断控制器：中断控制器是 CPU 众多外设中的一个，它一方面接收其它外设中断信号的输入，另一方面，它会发出中断信号给 CPU。可以通过对中断控制器编程实现对中断源的优先级、触发方式、打开和关闭源等设置操作。在 Cortex-M 系列控制器中常用的中断控制器是 NVIC（内嵌向量中断控制器 NestedVectored Interrupt Controller）。

CPU：CPU 会响应中断源的请求，中断当前正在执行的线程，转而执行中断处理程序。NVIC 最多支持 240 个中断，每个中断最多 256 个优先级。

28.1.2 和中断相关的名词解释

中断号：每个中断请求信号都会有特定的标志，使得计算机能够判断是哪个设备提出的中断请求，这个标志就是中断号。

中断请求："紧急事件"需向 CPU 提出申请，要求 CPU 暂停当前执行的线程，转而处理该"紧急事件"，这一申请过程称为中断请求。

中断优先级：为使系统能够及时响应并处理所有中断，系统根据中断时间的重要性和紧迫程度，将中断源分为若干个级别，称作中断优先级。

中断处理程序：当外设产生中断请求后，CPU 暂停当前的线程，转而响应中断申请，即执行中断处理程序。

中断触发：中断源发出并送给 CPU 控制信号，将中断触发器置"1"，表明该中断源产生了中断，要求 CPU 去响应该中断，CPU 暂停当前线程，执行相应的中断处理程序。

中断触发类型：外部中断申请通过一个物理信号发送到 NVIC，可以是电平触发或边沿触发。

中断向量：中断服务程序的入口地址。

中断向量表：存储中断向量的存储区，中断向量与中断号对应，中断向量在中断向量表中按照中断号顺序存储。

临界段：代码的临界段也称为临界区，一旦这部分代码开始执行，则不允许任何中断打断。为确保临界段代码的执行不被中断，在进入临界段之前须关中断，而临界段代码执行完毕后，要立即开中断。RT-Thread 支持中断屏蔽和中断使能。

28.2 中断管理的运作机制

当中断产生时，处理机将按如下的顺序执行：

1. 保存当前处理机状态信息
2. 载入异常或中断处理函数到 PC 寄存器
3. 把控制权转交给处理函数并开始执行

4. 当处理函数执行完成时，恢复处理器状态信息
5. 从异常或中断中返回到前一个程序执行点

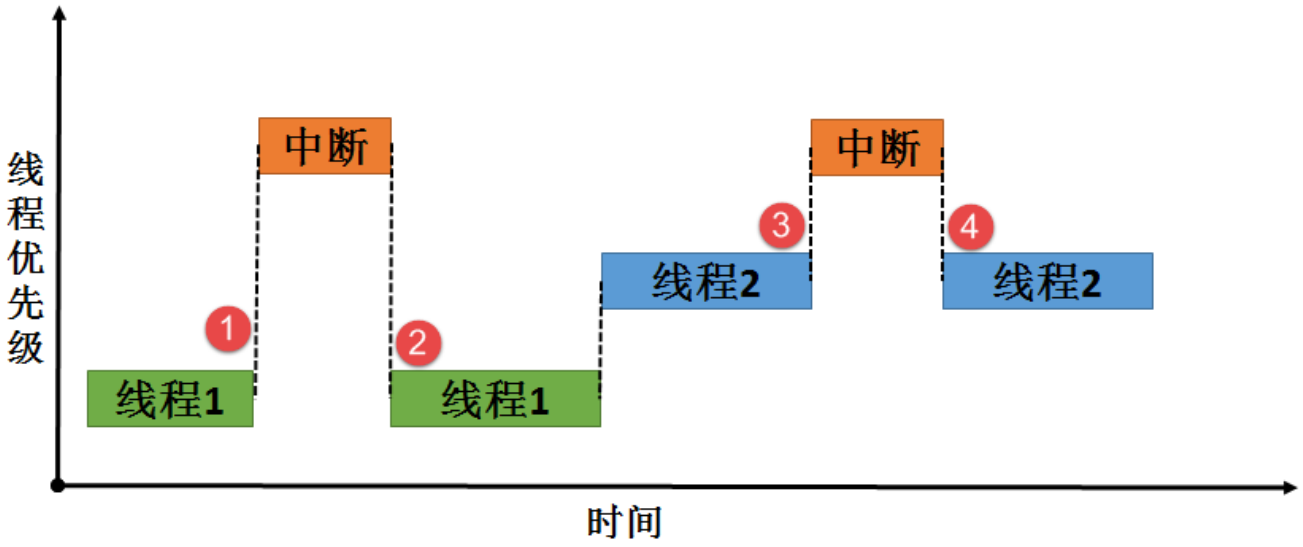
中断使得 CPU 可以在事件发生时才给予处理，而不必让 CPU 连续不断地查询是否有相应的事件发生。通过两条特殊指令：关中断和开中断可以让处理器不响应或响应中断，在关闭中断期间，通常处理器会把新产生的中断挂起，当中断打开时立刻进行响应，所以会有适当的延时响应中断，故用户在进入临界区的时候应快进快出。

中断发生的环境有两种情况：在线程的上下文中，在中断服务函数处理上下文中。

- 线程在工作的时候，如果此时发生了一个中断，无论中断的优先级是多大，都会打断当前线程的执行，从而转到对应的中断服务函数中执行，其过程具体见图 *25-1*。

图 *25-1* **(1)**、**(3)**：在线程运行的时候发生了中断，那么中断会打断线程的运行，那么操作系统将先保存当前线程的上下文环境，转而去处理中断服务函数。

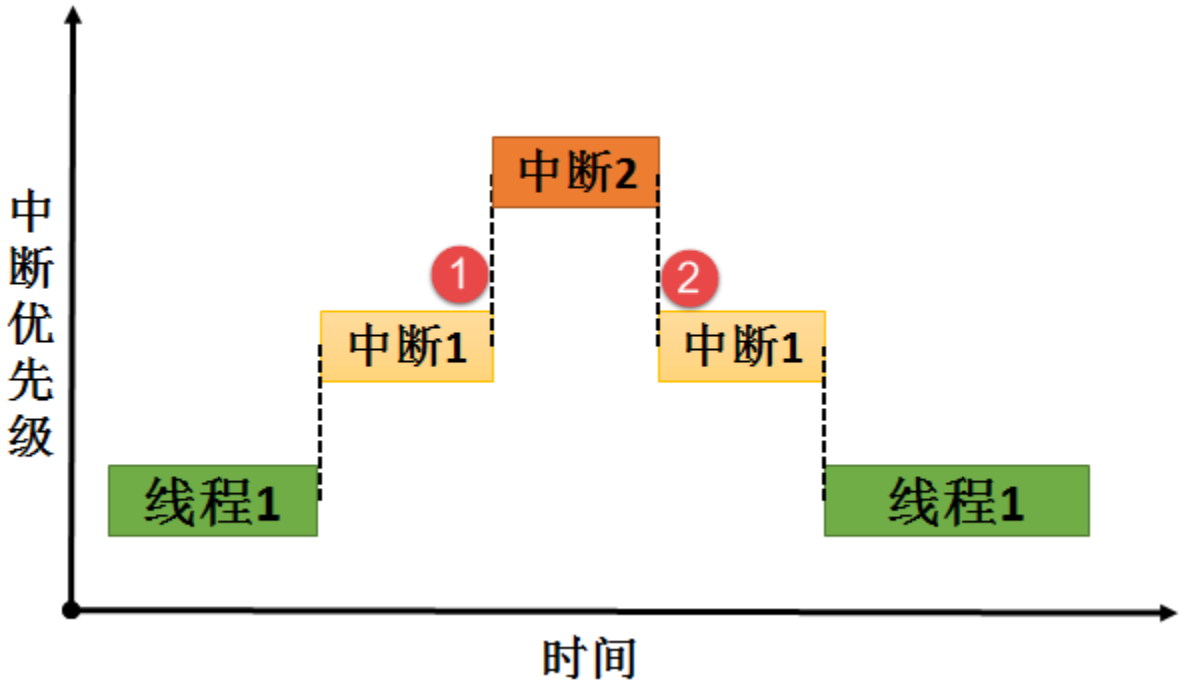
图 *25-1* **(2)**、**(4)**：当且仅当中断服务函数处理完的时候才恢复线程的上下文环境，继续运行线程。



- 在执行中断服务例程的过程中，如果有更高优先级别的中断源触发中断，由于当前处于中断处理上下文环境中，根据不同的处理器构架可能有不同的处理方式，比如新的中断等待挂起直到当前中断处理离开后再行响应；或新的高优先级中断打断当前中断处理过程，而去直接响应这个更高优先级的新中断源。后面这种情况，称之为中断嵌套。在硬实时环境中，前一种情况是不允许发生的，不能使响应中断的时间尽量的短。而在软件处理（软实

时环境）上，RT-Thread 允许中断嵌套，即在一个中断服务例程期间，处理器可以响应另外一个优先级更高的中断，过程如图 *25-2* 所示。

图 *25-2* **(1)**：当中断 1 的服务函数在处理的时候发生了中断 2，由于中断 2 的优先级比中断 1 更高，所以发生了中断嵌套，那么操作系统将先保存当前中断服务函数的上下文环境，并且转向处理中断 2，当且仅当中断 2 执行完的时候图 *25-2* **(2)**，才能继续执行中断 1。



28.3 中断延迟的概念

即使操作系统的响应很快了，但对于中断的处理仍然存在着中断延迟响应的问题，我们称之为中断延迟 (Interrupt Latency) 。

中断延迟是指从硬件中断发生到开始执行中断处理程序第一条指令之间的这段时间。也就是：系统接收到中断信号到操作系统作出响应，并完成换到转入中断服务程序的时间。也可以简单地理解为：（外部）硬件（设备）发生中断，到系统执行中断服务子程序（ISR）的第一条指令的时间。

中断的处理过程是：外界硬件发生了中断后，CPU 到中断处理器读取中断向量，并且查找中断向量表，找到对应的中断服务子程序（ISR）的首地址，然后跳转到对应的 ISR 去做相应处理。这

部分时间，我称之为：识别中断时间。

在允许中断嵌套的实时操作系统中，中断也是基于优先级的，允许高优先级中断抢断正在处理的低优先级中断，所以，如果当前正在处理更高优先级的中断，即使此时有低优先级的中断，也系统不会立刻响应，而是等到高优先级的中断处理完之后，才会响应。而即使在不支持中断嵌套，即中断是没有优先级的，中断是不允许被中断的，所以，如果当前系统正在处理一个中断，而此时另一个中断到来了，系统也是不会立即响应的，而只是等处理完当前的中断之后，才会处理后来的中断。此部分时间，我称其为：等待中断打开时间。

在操作系统中，很多时候我们会主动进入临界段，系统不允许当前状态被中断打断，故而在临界区发生的中断会被挂起，直到退出临界段时候打开中断。此部分时间，我称其为：关闭中断时间。

中断延迟可以定义为，从中断开始的时刻到中断服务例程开始执行的时刻之间的时间段。中断延迟 = 识别中断时间 + [等待中断打开时间] + [关闭中断时间]。

注意："[ ]" 的时间是不一定都存在的，此处为最大可能的中断延迟时间。

28.4 中断管理的应用场景

中断在嵌入式处理器中应用非常之多，没有中断的系统不是一个好系统，因为有中断，才能启动或者停止某件事情，从而转去做另一间事情。我们可以举一个日常生活中的例子来说明，假如你正在给朋友写信，电话铃响了，这时你放下手中的笔去接电话，通话完毕再继续写信。这个例子就表现了中断及其处理的过程：电话铃声使你暂时中止当前的工作，而去处理更为急需处理的事情——接电话，当把急需处理的事情处理完毕之后，再回过头来继续原来的事情。在这个例子中，电话铃声就可以称为"中断请求"，而你暂停写信去接电话就叫作"中断响应"，那么接电话的过程就是"中断处理"。由此我们可以看出，在计算机执行程序的过程中，由于出现某个特殊情况(或称为"特殊事件")，使得系统暂时中止现行程序，而转去执行处理这一特殊事件的程序，处理完毕之后再回到原来程序的中断点继续向下执行。

为什么说吗没有中断的系统不是好系统呢？我们可以再举一个例子来说明中断的作用。假设有一个朋友来拜访你，但是由于不知何时到达，你只能在门口等待，于是什么事情也干不了；但如果在门口装一个门铃，你就不必在门口等待而可以在家里去做其他的工作，朋友来了按门铃通知你，这时你才中断手中的工作去开门，这就避免了不必要的等待。CPU 也是一样，如果时间都浪

费在查询的事情上，那这个 CPU 啥也干不了，要他何用。在嵌入式系统中合理利用中断，能更好利用 CPU 的资源。

28.5 中断管理讲解

ARM Cortex-M 内核的中断是不受 RT-Thread 管理的，所以 RT-Thread 中的中断使用其实跟裸机差不多的，需要我们自己配置中断，并且使能中断，编写中断服务函数，在中断服务函数中使用内核 IPC 通信机制，一般建议使用信号量、消息或事件标志组等标志事件的发生，将事件发布给处理线程，等退出中断后再由相关处理线程具体处理中断。由于中断不受 RT-Thread 管理，所以不需要使用 RT-Thread 提供的函数（中断屏蔽与使能除外）。

ARM Cortex-M NVIC 支持中断嵌套功能：当一个中断触发并且系统进行响应时，处理器硬件会将当前运行的部分上下文寄存器自动压入中断栈中，这部分的寄存器包括 PSR，R0，R1，R2，R3 以及 R12 寄存器。当系统正在服务一个中断时，如果有一个更高优先级的中断触发，那么处理器同样的会打断当前运行的中断服务例程，然后把老的中断服务例程上下文的 PSR，R0，R1，R2，R3 和 R12 寄存器自动保存到中断栈中。这些部分上下文寄存器保存到中断栈的行为完全是硬件行为，这一点是与其他 ARM 处理器最大的区别（以往都需要依赖于软件保存上下文）。

另外，在 ARM Cortex-M 系列处理器上，所有中断都采用中断向量表的方式进行处理，即当一个中断触发时，处理器将直接判定是哪个中断源，然后直接跳转到相应的固定位置进行处理。而在 ARM7、ARM9 中，一般是先跳转进入 IRQ 入口，然后再由软件进行判断是哪个中断源触发，获得了相对应的中断服务例程入口地址后，再进行后续的中断处理。ARM7、ARM9 的好处在于，所有中断它们都有统一的入口地址，便于 OS 的统一管理。而 ARM Cortex-M 系列处理器则恰恰相反，每个中断服务例程必须排列在一起放在统一的地址上（这个地址必须要设置到 NVIC 的中断向量偏移寄存器中）。中断向量表一般由一个数组定义（或在起始代码中给出），在 i.MX RT 上，默认采用起始代码给出：具体见*代码清单 25-1*。

列表 1: 代码清单 25-1 中断向量表（部分）

```
__Vectors       DCD     |Image$$ARM_LIB_STACK$$ZI$$Limit| ; Top of Stack
                DCD     Reset_Handler  ; Reset Handler
                DCD     NMI_Handler                        ;NMI Handler
```

(下页继续)

(续上页)

```
                DCD     HardFault_Handler                   ;Hard Fault Handler
                DCD     MemManage_Handler                   ;MPU Fault Handler
                DCD     BusFault_Handler                    ;Bus Fault Handler
                DCD     UsageFault_Handler                  ;Usage Fault Handler
                DCD     0                                   ;Reserved
                DCD     0                                   ;Reserved
                DCD     0                                   ;Reserved
                DCD     0                                   ;Reserved
                DCD     SVC_Handler                         ;SVCall Handler
                DCD     DebugMon_Handler                    ;Debug Monitor Handler
                DCD     0                                   ;Reserved
                DCD     PendSV_Handler                      ;PendSV Handler
                DCD     SysTick_Handler                     ;SysTick Handler

                                                            ;External Interrupts
                DCD     DMA0_DMA16_IRQHandler               ;DMA channel 0/16 transfer complete
                DCD     DMA1_DMA17_IRQHandler               ;DMA channel 1/17 transfer complete
                DCD     DMA2_DMA18_IRQHandler               ;DMA channel 2/18 transfer complete
                DCD     DMA3_DMA19_IRQHandler               ;DMA channel 3/19 transfer complete
                DCD     DMA4_DMA20_IRQHandler               ;DMA channel 4/20 transfer complete
                DCD     DMA5_DMA21_IRQHandler               ;DMA channel 5/21 transfer complete
```

(下页继续)

(续上页)

```
                DCD     DMA6_DMA22_IRQHandler                ;DMA channel 6/22 transfer complete
                DCD     DMA7_DMA23_IRQHandler                ;DMA channel 7/23 transfer complete
                DCD     DMA8_DMA24_IRQHandler                ;DMA channel 8/24 transfer complete
                DCD     DMA9_DMA25_IRQHandler                ;DMA channel 9/25 transfer complete
                DCD     DMA10_DMA26_IRQHandler               ;DMA channel 10/26 transfer complete
                DCD     DMA11_DMA27_IRQHandler               ;DMA channel 11/27 transfer complete
                DCD     DMA12_DMA28_IRQHandler               ;DMA channel 12/28 transfer complete
                DCD     DMA13_DMA29_IRQHandler               ;DMA channel 13/29 transfer complete
                DCD     DMA14_DMA30_IRQHandler               ;DMA channel 14/30 transfer complete
                DCD     DMA15_DMA31_IRQHandler               ;DMA channel 15/31 transfer complete
                DCD     DMA_ERROR_IRQHandler                 ;DMA error interrupt channels 0-15 / 16-31
                DCD     CTI0_ERROR_IRQHandler                ;CTI0_Error
                DCD     CTI1_ERROR_IRQHandler                ;CTI1_Error
                DCD     CORE_IRQHandler                      ;CorePlatform exception IRQ
                DCD     LPUART1_IRQHandler                   ;LPUART1 TX interrupt and RX interrupt
                DCD     LPUART2_IRQHandler                   ;LPUART2 TX interrupt and RX interrupt
```

RT-Thread 在 Cortex-M 系列处理器上也遵循与裸机中断一致的方法，当用户需要使用自定义的中

断服务例程时，只需要定义相同名称的函数覆盖弱化符号即可。所以，RT-Thread 在 Cortex-M 系列处理器的中断控制其实与裸机没什么差别。

28.6 中断管理的实验

中断管理实验是在 RT-Thread 中创建了两个线程分别获取信号量与消息队列，并且定义了两个按键 KEY1 与 KEY2 的触发方式为中断触发，其触发的中断服务函数则跟裸机一样，在中断触发的时候通过消息队列将消息传递给线程，线程接收到消息就将信息通过串口调试助手显示出来。而且中断管理实验也实现了一个串口的 DMA 传输 + 空闲中断功能，当串口接收完不定长的数据之后产生一个空闲中断，在中断中将信号量传递给线程，线程在收到信号量的时候将串口的数据读取出来并且在串口调试助手中回显，具体见*代码清单 25-2* 高亮部分。

列表 2: 代码清单 25-2 中断管理的实验

```
/*********************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2019-xx-xx
* @brief   RT-Thread 中断管理
*********************************************************************
* @attention
*
*   实验平台: 基于野火 i.MX RT 全系列开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
**********************************************************************
*/

/*
**********************************************************************
```

(下页继续)

(续上页)

```
*                                包含的头文件
*****************************************************************************
*/
#include "board.h"
#include "fsl_debug_console.h"
/* RT-Thread 头文件 */
#include "rtthread.h"
#include <string.h>
/*****************************************************************
*                                变量
*****************************************************************
*/

/* 定义线程控制块 */
static rt_thread_t key_thread = RT_NULL;
static rt_thread_t usart_thread = RT_NULL;
/* 定义消息队列控制块 */
rt_mq_t test_mq = RT_NULL;
/* 定义信号量控制块 */
rt_sem_t test_sem = RT_NULL;

/************************* 全局变量声明 ****************************/
/*
* 当我们在写应用程序的时候，可能需要用到一些全局变量。
*/

/* 相关宏定义 */
extern char Usart_Rx_Buf[USART_RBUFF_SIZE];

/*
*****************************************************************************
*                                函数声明
```

(下页继续)

(续上页)

```
*************************************************************************
*/
static void key_thread_entry(void* parameter);
static void usart_thread_entry(void* parameter);

/*
*************************************************************************
*                              main 函数
*************************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]- i.MX RT 全系列开发板-RTT 中断管理实验！\n");
    rt_kprintf("按下 KEY1 | KEY2 触发中断！\n");
    rt_kprintf("串口发送数据触发中断，线程处理数据！\n");
    /* 创建一个消息队列 */
    test_mq = rt_mq_create("test_mq",/* 消息队列名字 */
                           4,      /* 消息的最大长度 */
                           2,     /* 消息队列的最大容量 */
                           RT_IPC_FLAG_FIFO);/* 队列模式 FIFO(0x00)*/
    if (test_mq != RT_NULL)
        rt_kprintf("消息队列创建成功！\n\n");
```

(下页继续)

(续上页)

```
    /* 创建一个信号量 */
    test_sem = rt_sem_create("test_sem",/* 消息队列名字 */
                             0,      /* 信号量初始值，默认有一个信号量 */
                             RT_IPC_FLAG_FIFO); /* 信号量模式 FIFO(0x00)*/
    if (test_sem != RT_NULL)
        rt_kprintf("信号量创建成功! \n\n");

    /* 创建一个线程 */
    key_thread =                            /* 线程控制块指针 */
        rt_thread_create( "key",                /* 线程名字 */
                          key_thread_entry,    /* 线程入口函数 */
                          RT_NULL,             /* 线程入口函数参数 */
                          512,                 /* 线程栈大小 */
                          1,                   /* 线程的优先级 */
                          20);                 /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (key_thread != RT_NULL)
        rt_thread_startup(key_thread);
    else
        return -1;

    usart_thread =                          /* 线程控制块指针 */
        rt_thread_create( "usart",              /* 线程名字 */
                          usart_thread_entry,   /* 线程入口函数 */
                          RT_NULL,             /* 线程入口函数参数 */
                          512,                 /* 线程栈大小 */
                          2,                   /* 线程的优先级 */
                          20);                 /* 线程时间片 */

    /* 启动线程，开启调度 */
```

(下页继续)

(续上页)

```
    if (usart_thread != RT_NULL)
        rt_thread_startup(usart_thread);
    else
        return -1;
}

/*
*************************************************************************
*                                 线程定义
*************************************************************************
*/

static void key_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
    uint32_t r_queue;
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        /* 队列读取（接收），等待时间为一直等待 */
        uwRet = rt_mq_recv(test_mq, /* 读取（接收）队列的 ID(句柄) */
                           &r_queue,        /* 读取（接收）的数据保存位置 */
                           sizeof(r_queue), /* 读取（接收）的数据的长度 */
                           RT_WAITING_FOREVER); /* 等待时间：一直等 */
        if (RT_EOK == uwRet) {
            rt_kprintf("触发中断的是 KEY%d!\n",r_queue);
        } else {
            rt_kprintf("数据接收出错，错误代码：0x%lx\n",uwRet);
        }
        LED1_TOGGLE;
    }
}

```

(下页继续)

(续上页)

```
static void usart_thread_entry(void* parameter)
{
    rt_err_t uwRet = RT_EOK;
    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        uwRet = rt_sem_take(test_sem,        /* 获取串口中断的信号量 */
                            0);        /* 等待时间: 0 */
        if (RT_EOK == uwRet) {
            rt_kprintf("收到数据:%s\n",Usart_Rx_Buf);
            memset(Usart_Rx_Buf,0,USART_RBUFF_SIZE);/* 清零 */
        }
    }
}
/****************************END OF FILE****************************/
```

而中断服务函数则需要我们自己编写，并且通过信号量告知线程，具体见代码清单 25-3。

列表 3: 代码清单 25-3 中断管理——中断服务函数 (按键中断)

```
/******************** 中断服务函数 **************************/
/**
* @brief  GPIO 输入中断服务函数
*         CORE_BOARD_WAUP_KEY_IRQHandler 只是一个宏，
*         在本例中它指代函数名 GPIO5_Combined_0_15_IRQHandler，
*         中断服务函数名是固定的，可以在启动文件中找到。
* @param  中断服务函数不能有输入参数
* @note   中断函数一般只使用标志位进行指示，完成后尽快退出，
*         具体操作或延时尽量不放在中断服务函数中
* @retval 中断服务函数不能有返回值
*/
void CORE_BOARD_WAUP_KEY_IRQHandler(void)
```

(下页继续)

(续上页)

```
{
    /* 进入中断 */
    rt_interrupt_enter();
    /*  检测按键中断标志  */
    if (GPIO_GetPinsInterruptFlags(CORE_BOARD_WAUP_KEY_GPIO)&(1U<<CORE_BOARD_WAUP_KEY_GPIO_PIN)) {
        CPU_TS_Tmr_Delay_MS(10);
        /*  判断按键是否 按下 */
        if (GPIO_PinRead(CORE_BOARD_WAUP_KEY_GPIO, CORE_BOARD_WAUP_KEY_GPIO_PIN) == 1) {
            /* 将数据写入（发送）到队列中，等待时间为 0  */
            rt_mq_send( test_mq,  /* 写入（发送）队列的 ID(句柄) */
                        &send_data1,      /* 写入（发送）的数据 */
                        sizeof(send_data1));      /* 数据的长度 */
        }
    }
    /* 清除中断标志位 */
    GPIO_PortClearInterruptFlags(CORE_BOARD_WAUP_KEY_GPIO,
                                 1U << CORE_BOARD_WAUP_KEY_GPIO_PIN);
#if defined __CORTEX_M && (__CORTEX_M == 4U)
    __DSB();
#endif
    /* 离开中断 */
    rt_interrupt_leave();
}

/**
* @brief  GPIO 输入中断服务函数
*         CORE_BOARD_MODE_KEY_IRQHandler 只是一个宏，
*         在本例中它指代函数名 GPIO1_Combined_0_15_IRQHandler，
*         中断服务函数名是固定的，可以在启动文件中找到。
* @param  中断服务函数不能有输入参数
```

(下页继续)

(续上页)

```
* @note    中断函数一般只使用标志位进行指示，完成后尽快退出，
*          具体操作或延时尽量不放在中断服务函数中
* @retval 中断服务函数不能有返回值
*/
void CORE_BOARD_MODE_KEY_IRQHandler(void)
{
    /* 进入中断 */
    rt_interrupt_enter();

    /*  检测按键中断标志  */
    if (GPIO_GetPinsInterruptFlags(CORE_BOARD_MODE_KEY_GPIO)&(1U<<CORE_BOARD_MODE_KEY_GPIO_PIN)) {
        CPU_TS_Tmr_Delay_MS(10);
        /*  判断按键是否 按下 */
        if (GPIO_PinRead(CORE_BOARD_MODE_KEY_GPIO, CORE_BOARD_MODE_KEY_GPIO_PIN) == 1) {
            /* 将数据写入（发送）到队列中，等待时间为 0  */
            rt_mq_send( test_mq,  /* 写入（发送）队列的 ID(句柄) */
                        &send_data2,      /* 写入（发送）的数据 */
                        sizeof(send_data2));      /* 数据的长度 */
        }
    }
    /* 清除中断标志位 */
    GPIO_PortClearInterruptFlags(CORE_BOARD_MODE_KEY_GPIO,
                                 1U << CORE_BOARD_MODE_KEY_GPIO_PIN);
#if defined __CORTEX_M && (__CORTEX_M == 4U)
    __DSB();
#endif
    /* 离开中断 */
    rt_interrupt_leave();
}
```

列表 4: 代码清单 25-4 中断管理——中断服务函数 (串口中断)

```
void DEBUG_UART_IRQHandler(void)
{
    /* 进入中断 */
    rt_interrupt_enter();
    /*  接收到数据满了触发中断 */
    if ((kLPUART_IdleLineFlag) & LPUART_GetStatusFlags(DEMO_LPUART)) {
        /* 清除空闲中断 */
        DEMO_LPUART->STAT |= LPUART_STAT_IDLE_MASK;
        /* 接收 eDMA 的数据量 */
        LPUART_TransferGetReceiveCountEDMA(DEMO_LPUART, &g_lpuartEdmaHandle, &data);
        Get_Uart_Data_flag=1;
        LPUART_TransferAbortReceiveEDMA(DEMO_LPUART, &g_lpuartEdmaHandle); //eDMA 终止接收数据
        LPUART_ReceiveEDMA(DEMO_LPUART, &g_lpuartEdmaHandle, &receiveXfer); //使用 eDMA 接收

        //给出二值信号量 ，发送接收到新数据标志，供前台程序查询
        rt_sem_release(test_sem);

    }
    /* 离开中断 */
    rt_interrupt_leave();
}
```

28.7 中断管理的实验现象

程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息，按下开发版的 K1 按键触发中断发送消息 1，按下 K2 按键发送消息 2；我们按下 K1 与 K2 试试，在串口调试助手中可以看到运行结果，然后通过串口调试助手发送一段不定长信息，触发中断会在中断服务函数发送信号量通知线程，线程接收到信号量的时候将串口信息打印出来，具体见图 *25-3*。

第 29 章　双向链表

29.1 双向链表的基本概念

双向链表也叫双链表，是链表的一种，是在操作系统中常用的数据结构，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱，其头指针 head 是唯一确定的。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点，这种数据结构形式使得双向链表在查找时更加方便，特别是大量数据的遍历。由于双向链表具有对称性，能方便地完成各种插入、删除等操作，但需要注意前后方向的操作。

29.2 双向链表的函数接口讲解

RT-Thread 为我们提供了很多操作链表的函数，如链表的初始化、添加节点、删除节点等。

RT-Thread 的链表节点结构体中只有两个指针，一个是指向上一个节点的指针，另一个是指向下一个节点的指针，具体见*代码清单 26-1*

列表 1: 代码清单 26-1 链表节点结构体

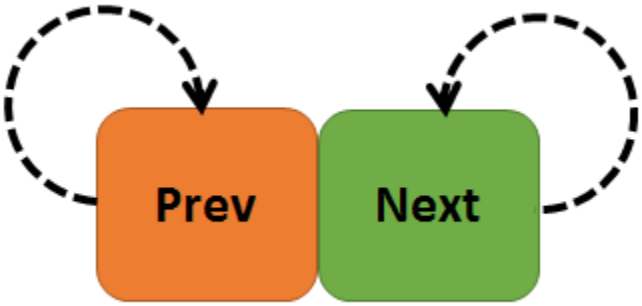
```
struct rt_list_node {
    struct rt_list_node *next;  /**< 指向下一个节点的指针. */
    struct rt_list_node *prev;  /**< 指向上一个节点的指针. */
};
typedef struct rt_list_node rt_list_t;
```

29.2.1 链表初始化函数 rt_list_init()

在使用链表的时候必须要进行初始化，将链表的指针指向自己，为以后添加节点做准备，链表的数据结构也是需要内存空间的，所以也需要进行内存的申请，链表初始化函数 rt_list_init() 的源码具体见*代码清单 26-2*，其结果具体见图 *26-1*。

列表 2: 代码清单 26-2 链表初始化函数 rt_list_init() 源码

```
/**
* @brief 初始化一个链表
*
* @param
*/
rt_inline void rt_list_init(rt_list_t *l)
{
    l->next = l->prev = l;
}
```



其初始化完成后可以检查一下链表初始化是否成功，判断链表是不是空的就行了，因为初始化完成的时候，链表肯定是空的，注意，在初始化链表的时候其实链表就是链表头，需要申请内存，链表的初始化实例具体见*代码清单 26-3*。

列表 3: 代码清单 26-3 链表初始化函数 rt_list_init() 实例

```
head = rt_malloc(sizeof(rt_list_t));/* 申请动态内存 */
if (RT_NULL == head) /* 没有申请成功 */
    rt_kprintf("动态内存申请失败! \n");
else
    rt_kprintf("动态内存申请成功，头结点地址为%d! \n",head);

```

(下页继续)

(续上页)

```
rt_kprintf("\n双向链表初始化中......\n");
rt_list_init(head);
if (rt_list_isempty(head))
    rt_kprintf("双向链表初始化成功!\n\n");
```

29.2.2 向链表中插入节点

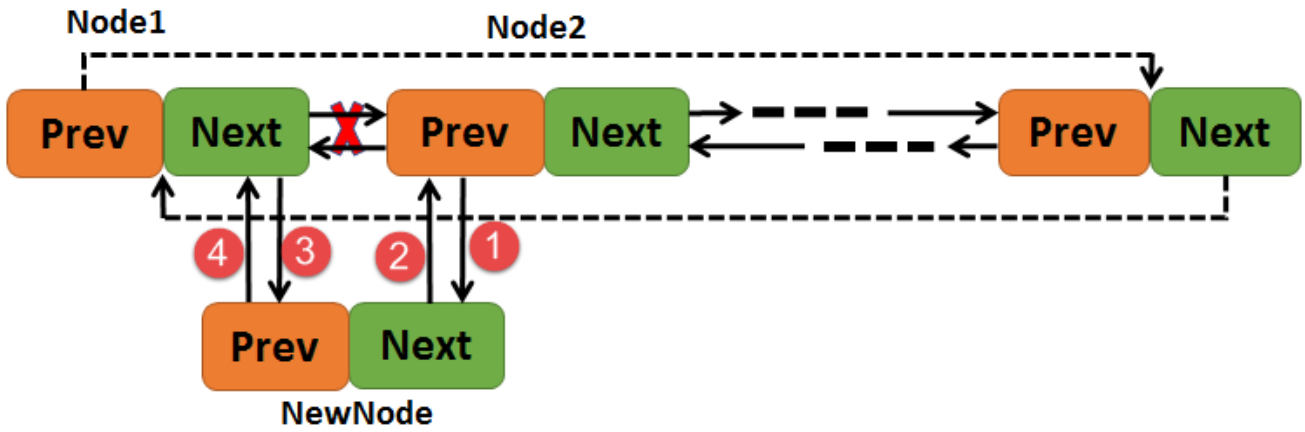
29.2.2.1 向链表指定节点后面插入节点 rt_list_insert_after()

插入节点需要先申请节点大小的内存，然后根据插入的位置（在某个节点（rt_list_t *l）后面）进行插入操作，具体见*代码清单 26-4*。

列表 4: 代码清单 26-4 向链表指定节点后面插入节点 rt_list_insert_after() 源码

```
rt_inline void rt_list_insert_after(rt_list_t *l, rt_list_t *n)
{
    l->next->prev = n;                          (1)
    n->next = l->next;                          (2)

    l->next = n;                        (3)
    n->prev = l;                        (4)
}
```

这是数据结构的基本使用方法，其过程具体见图 *26-2*。



29.2.2.2 向链表指定节点前面插入节点 rt_list_insert_before()

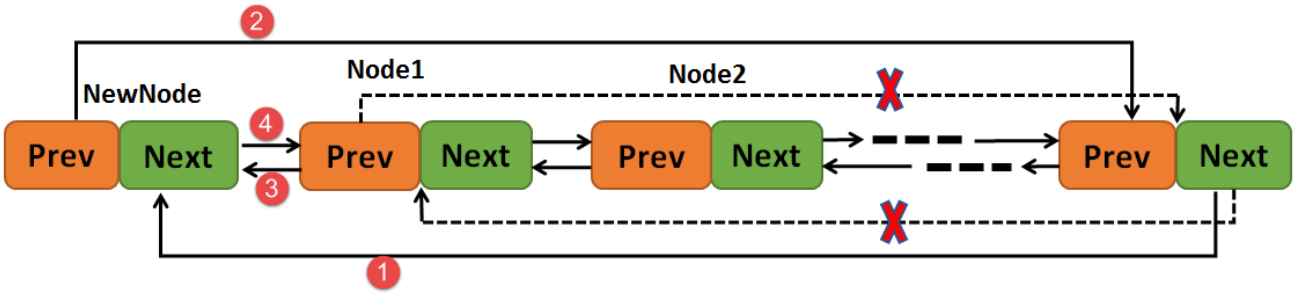
插入节点需要先申请节点大小的内存，然后根据插入的位置（在某个节点（rt_list_t *l）前面）进行插入操作，具体见*代码清单 26-5*。

列表 5: 代码清单 26-5 向链表指定节点前面插入节点

rt_list_insert_before() 源码

```
rt_inline void rt_list_insert_before(rt_list_t *l, rt_list_t *n)
{
    l->prev->next = n;                          (1)
    n->prev = l->prev;                          (2)

    l->prev = n;                        (3)
    n->next = l;                        (4)
}
```

这是数据结构的基本使用方法，其过程具体见图 *26-3*。



插入节点的实例也很简单，但是要注意的是要申请内存，具体见*代码清单 26-6* 高亮部分。

列表 6: 代码清单 26-6 向链表插入节点函数实例

```
/* 插入节点：顺序插入与从末尾插入 */

rt_kprintf("添加节点和尾节点添加......\n");

/* 动态申请第一个结点的内存 */
node1 = rt_malloc(sizeof(rt_list_t));

/* 动态申请第二个结点的内存 */
```

(下页继续)

(续上页)

```
node2 = rt_malloc(sizeof(rt_list_t));

rt_kprintf("添加第一个节点与第二个节点.....\n");

/* 因为这是在某个节点后面添加一个节点函数
为后面的 rt_list_insert_before (某个节点之前)
添加节点做铺垫, 两个函数添加完之后的顺序是
head -> node1 -> node2 */

rt_list_insert_after(head,node2);

rt_list_insert_before(node2,node1);

if ((node1->prev == head) && (node2->prev == node1))
    rt_kprintf("添加节点成功!\n\n");
else
    rt_kprintf("添加节点失败!\n\n");
```

29.2.3 从链表删除节点函数 rt_list_remove()

删除节点与添加节点一样，其实删除节点更简单，只需要知道删除哪个节点即可，把该节点前后的节点链接起来，那它就删除了，然后该节点的指针指向节点本身即可，不过要注意的是也要讲该节点的内存释放掉，因为该节点是动态分配内存的，否则会导致内存泄漏，源码具体见代码清单 *26-7*，其实现过程具体见图 *26-4*。

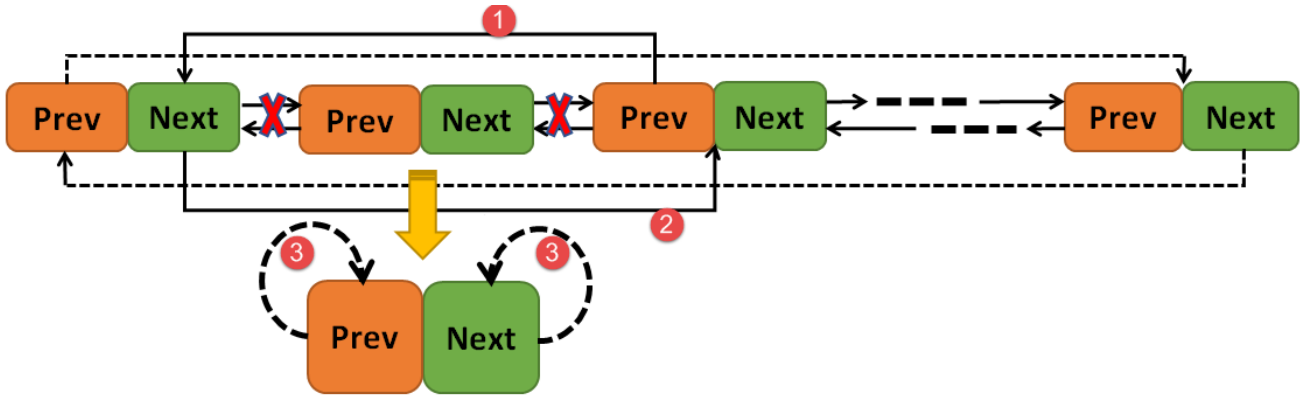
列表 7: 代码清单 26-7 从链表删除节点函数 rt_list_remove() 源码

```
rt_inline void rt_list_remove(rt_list_t *n)
{
    n->next->prev = n->prev;                    (1)
```

(下页继续)

(续上页)

```
    n->prev->next = n->next;                    (2)

    n->next = n->prev = n;              (3)
}
```



删除节点的用法也是简单，具体见代码清单 26-8

列表 8: 代码清单 26-8 从链表删除节点函数 rt_list_remove() 实例

```
rt_kprintf("删除节点......\n"); /* 删除已有节点 */
rt_list_remove(node1);
rt_free(node1);/* 释放第一个节点的内存 */
if (node2->prev == head)
    rt_kprintf("删除节点成功\n\n");
```

29.3 双向链表的实验

双向链表实验实现如下功能：

- 调用 rt_list_init 初始双向链表。
- 调用 rt_list_insert_after 与 rt_list_insert_ before 向链表中增加节点。

- 调用 rt_list_remove 删除指定节点。
- 调用 rt_list_isempty 判断链表是否为空。
- 测试操作是否成功。

删除节点的时候要注意释放掉内存，具体见代码清单 26-9 高亮部分。

列表 9: 代码清单 26-9 双向链表实验

```
/**
*****************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2018-xx-xx
* @brief   RT-Thread 3.0 + STM32 双向链表实验
*****************************************************************
* @attention
*
* 实验平台: 基于野火 STM32 全系列 (M3/4/7) 开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
******************************************************************
*/

/*
*******************************************************************
*                              包含的头文件
*******************************************************************
*/
#include "board.h"
#include "rtthread.h"

```

(下页继续)

(续上页)

```

/*
*****************************************************************
*                                 变量
*****************************************************************
*/
/* 定义线程控制块 */
static rt_thread_t test1_thread = RT_NULL;
static rt_thread_t test2_thread = RT_NULL;

/************************* 全局变量声明 ****************************/
/*
* 当我们在写应用程序的时候，可能需要用到一些全局变量。
*/

/*
*********************************************************************
*                                 函数声明
*********************************************************************
*/
static void test1_thread_entry(void* parameter);
static void test2_thread_entry(void* parameter);

/*
*********************************************************************
*                                 main 函数
*********************************************************************
*/
/**
* @brief  主函数
* @param  无
* @retval 无
```

(下页继续)

(续上页)

```
*/
int main(void)
{
    /*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */
    rt_kprintf("这是一个 [野火]- STM32 全系列开发板-RTT 双向链表操作实验！\n");

    test1_thread =                          /* 线程控制块指针 */
        rt_thread_create( "test1",              /* 线程名字 */
                        test1_thread_entry,   /* 线程入口函数 */
                        RT_NULL,              /* 线程入口函数参数 */
                        512,                  /* 线程栈大小 */
                        2,                    /* 线程的优先级 */
                        20);                  /* 线程时间片 */

    /* 启动线程，开启调度 */
    if (test1_thread != RT_NULL)
        rt_thread_startup(test1_thread);
    else
        return -1;

    test2_thread =                          /* 线程控制块指针 */
        rt_thread_create( "test2",              /* 线程名字 */
                        test2_thread_entry,   /* 线程入口函数 */
                        RT_NULL,              /* 线程入口函数参数 */
                        512,                  /* 线程栈大小 */
                        3,                    /* 线程的优先级 */
                        20);                  /* 线程时间片 */

```

(下页继续)

(续上页)

```
    /* 启动线程，开启调度 */
    if (test2_thread != RT_NULL)
        rt_thread_startup(test2_thread);
    else
        return -1;
}

/*
*****************************************************************************
*                                 线程定义
*****************************************************************************
*/

static void test1_thread_entry(void* parameter)
{
    rt_list_t *head;  /* 定义一个双向链表的头节点 */
    rt_list_t *node1;  /* 定义一个双向链表的头节点 */
    rt_list_t *node2;  /* 定义一个双向链表的头节点 */

    head = rt_malloc(sizeof(rt_list_t));/* 申请动态内存 */
    if (RT_NULL == head) /* 没有申请成功 */
        rt_kprintf("动态内存申请失败! \n");
    else
        rt_kprintf("动态内存申请成功，头结点地址为%d! \n",head);

    rt_kprintf("\n双向链表初始化中......\n");
    rt_list_init(head);
    if (rt_list_isempty(head))
        rt_kprintf("双向链表初始化成功!\n\n");

    /* 插入节点：顺序插入与从末尾插入 */
    rt_kprintf("添加节点和尾节点添加......\n");
```

(下页继续)

(续上页)

```
    /* 动态申请第一个结点的内存 */
    node1 = rt_malloc(sizeof(rt_list_t));

    /* 动态申请第二个结点的内存 */
    node2 = rt_malloc(sizeof(rt_list_t));

    rt_kprintf("添加第一个节点与第二个节点.....\n");

    /* 因为这是在某个节点后面添加一个节点函数
    为后面的 rt_list_insert_before (某个节点之前)
    添加节点做铺垫, 两个函数添加完之后的顺序是
    head -> node1 -> node2 */

    rt_list_insert_after(head,node2);

    rt_list_insert_before(node2,node1);

    if ((node1->prev == head) && (node2->prev == node1))
        rt_kprintf("添加节点成功!\n\n");
    else
        rt_kprintf("添加节点失败!\n\n");

    rt_kprintf("删除节点......\n");      /* 删除已有节点 */
    rt_list_remove(node1);
    rt_free(node1);/* 释放第一个节点的内存 */
    if (node2->prev == head)
        rt_kprintf("删除节点成功\n\n");

    /* 线程都是一个无限循环, 不能返回 */
    while (1) {
        LED1_TOGGLE;
```

(下页继续)

(续上页)

```
        rt_thread_delay(500);      //每 500ms 扫描一次
    }
}

static void test2_thread_entry(void* parameter)
{

    /* 线程都是一个无限循环，不能返回 */
    while (1) {
        rt_kprintf("线程运行中!\n");
        LED2_TOGGLE;
        rt_thread_delay(1000);      //每 1000ms 扫描一次
    }
}
/****************************END OF FILE****************************/
```

29.4 双向链表的实验现象

程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 i.MX RT 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序）点击编译，然后下载到开发板上，打开我们野火的串口调试助手，它里面输出了信息表明双向链表的操作已经全部完成，开发板上的 LED 也在闪烁，具体见图 *26-5*。

野火串口调试助手V1.3 技术论坛：www.fireBBS.cn 官网：www.embedFire.com
串口调试功能
端口
COM11
波特率
115200
校验位
None
数据位
8
停止位
1
打开串口
清空接收区
停止显示
自动清空
十六进制显示
接收转向文件
*****欢迎使用 野火i.MX RT1052 开发板*****
\ | /
- RT - Thread Operating System
/ | \ 3.0.3 build Aug 30 2019
2006 - 2018 Copyright by rt-thread team
这是一个[野火]-i.MX RT-RTT双向链表操作实验！
动态内存申请成功，头结点地址为-2147482264！
双向链表初始化中.....
双向链表初始化成功！
添加节点和尾节点添加.....
添加第一个节点与第二个节点.....
添加节点成功！
删除节点.....
删除节点成功
任务运行中！
任务运行中！
任务运行中！
任务运行中！

第 30 章　CPU 利用率统计

30.1 CPU 利用率的基本概念

CPU 使用率其实就是系统运行的程序占用的 CPU 资源，表示机器在某段时间程序运行的情况，如果这段时间中，程序一直在占用 CPU 的使用权，那么可以人为 CPU 的利用率是 100%。CPU 的利用率越高，说明机器在这个时间上运行了很多程序，反之较少。利用率的高低与 CPU 强弱有直接关系，就像一段一模一样的程序，如果使用运算速度很慢的 CPU，它可能要运行 1000ms，而使用很运算速度很快的 CPU 可能只需要 10ms，那么在 1000ms 这段时间中，前者的 CPU 利用率就是 100%，而后者的 CPU 利用率只有 1%，因为 1000ms 内前者都在使用 CPU 做运算，而后者只使用 10ms 的时间做运算，剩下的时间 CPU 可以做其他事情。

RT-Thread 是多线程操作系统，对 CPU 都是分时使用的：比如 A 进程占用 10ms，然后 B 进程占用 30ms，然后空闲 60ms，再又是 A 进程占 10ms，B 进程占 30ms，空闲 60ms; 如果在一段时间内都是如此，那么这段时间内的利用率为 40%，因为整个系统中只有 40% 的时间是 CPU 处理数据的时间。

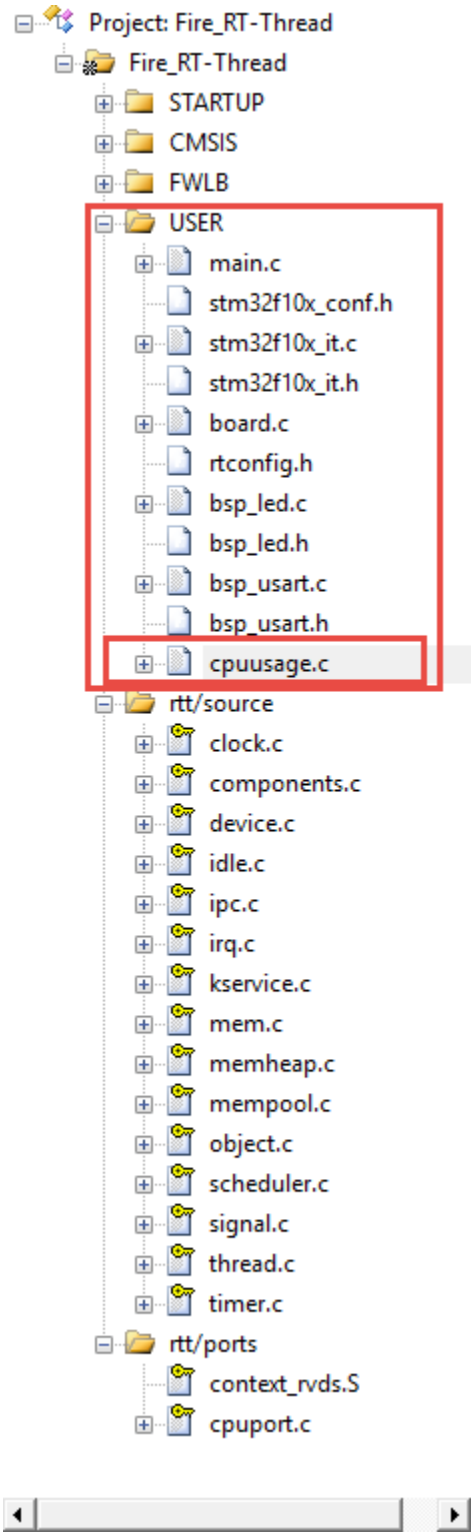
30.2 CPU 利用率的作用

一个系统设计的好坏，可以使用 CPU 使用率来衡量，一个好的系统必然是能完美响应急需的处理，并且系统的资源不会过于浪费（性价比高）。举个例子，假设一个系统的 CPU 利用率经常在 90%~100% 徘徊，那么系统就很少有空闲的时候，这时候突然有一些事情急需 CPU 的处理，但是此时 CPU 都很可能被其他线程在占用了，那么这个紧急事件就有可能无法被相应，即使能被相应，那么占用 CPU 的线程又处于等待状态，这种系统就是不够完美的，因为资源处理得太过于紧迫；反过来，假如 CPU 的利用率在 1% 以下，那么我们就可以认为这种产品的资源过于浪费，搞一个那么好的 CPU 去干着没啥意义的活（大部分时间处于空闲状态），使用，作为产品的设计，既不能让资源过于浪费，也不能让资源过于紧迫，这种设计才是完美的，在需要的时候能及时处理完突发事件，而且资源也不会过剩，性价比更高。

30.3 CPU 利用率统计

RT-Thread 给我们提供一个 CPU 统计的代码文件，该代码并非 RT-Thread 内核资源，只是利用 RT-Thread 中空闲线程来统计 CPU 的利用率，实现的算法原理很简单，在 RT-Thread 的空闲线程计算出在一段时间内处于空闲线程的时间，就知道 CPU 在有效干活的时间，从而得到 CPU 的利用率。下面来看看 CPU 利用率统计的源码文件，该文件在 rt-thread–masterexampleskernel 路径下，名字叫 cpuusage.c。

如果需要使用它来统计 CPU 利用率，那么需要先将该文件添加到我们的工程中，我们首先将 cpuusage.c 文件复制到我们模板工程的 User 文件夹下面，然后添加到我们的工程分组中，具体见图 *27-1*。


Project: Fire_RT-Thread
Fire_RT-Thread
STARTUP
CMSIS
FWLB
USER
main.c
stm32f10x_conf.h
stm32f10x_it.c
stm32f10x_it.h
board.c
rtconfig.h
bsp_led.c
bsp_led.h
bsp_usart.c
bsp_usart.h
cpuusage.c
rtt/source
clock.c
components.c
device.c
idle.c
ipc.c
irq.c
kservice.c
mem.c
memheap.c
mempool.c
object.c
scheduler.c
signal.c
thread.c
timer.c
rtt/ports
context_rvds.S
cpuport.c

添加到工程文件中的时候我们可以进行编译一次工程，会发现有错误，发现提示无法识别 rt_thread_idle_sethook() 函数，该函数在 cpuusage.c 文件中被调用，具体见图 *27-2*。

```
*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'D:\Keil_v5\ARM\ARMCC\Bin'
Build target 'Fire_RT-Thread'
linking...
..\..\Output\Fire_RT-Thread.axf: Error: L6218E: Undefined symbol rt_thread_idle_sethook (referred from cpuusage.o).
Not enough information to list image symbols.
Not enough information to list load addresses in the image map.
Finished: 2 information, 0 warning and 1 error messages.
"..\..\Output\Fire_RT-Thread.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed:  00:00:00
```

原来，rt_thread_idle_sethook() 函数是一个钩子函数，看函数的名字就知道函数的作用了，该函数是系统定义的函数，只有启用空闲钩子函数的时候编译器才会将空闲钩子函数相关的代码编译进来，我们就在 rtconfig.h 中将 RT_USING_IDLE_HOOK 宏定义打开，再次编译的时候就会发现错误已经消失了，这样子我们就能使用 cpuusage.c 文件提供的函数来统计我们的 CPU 利用率了。

为了使用方便，我们再创建一个与 cpuusage.c 源码文件对应的头文件 cpuusage.h，目的是为了声明 cpuusage.c 文件中对外提供的函数接口，方便调用这些函数，具体见*代码清单 27-1*，cpuusage.c 源码文件具体见*代码清单 27-2*。

列表 1: 代码清单 27-1cpuusage.h 文件（自己创建的）

```
#ifndef __CPUUSAGE_H__
#define __CPUUSAGE_H__


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


/* 获取 cpu 利用率 */
void cpu_usage_init(void);
void cpu_usage_get(rt_uint8_t *major, rt_uint8_t *minor);

#endif
```

列表 2: 代码清单 27-2cpuusage.c 文件（文件路径：rt-threadexampleskernel）

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

#define CPU_USAGE_CALC_TICK    1000
#define CPU_USAGE_LOOP        100

static rt_uint8_t  cpu_usage_major = 0, cpu_usage_minor= 0;
static rt_uint32_t total_count = 0;

static void cpu_usage_idle_hook()
{
    rt_tick_t tick;
    rt_uint32_t count;
volatile rt_uint32_t loop;

if (total_count == 0) {                                (1)
/* get total count */
        rt_enter_critical();                                   (2)
        tick = rt_tick_get();                                  (3)
while (rt_tick_get() - tick < CPU_USAGE_CALC_TICK) {
            total_count ++;                            (4)
            loop = 0;
while (loop < CPU_USAGE_LOOP) loop ++;
        }
        rt_exit_critical();
    }

    count = 0;
```

(下页继续)

(续上页)

```
/* get CPU usage */
    tick = rt_tick_get();                                   (5)
while (rt_tick_get() - tick < CPU_USAGE_CALC_TICK) {
        count ++;                                           (6)
        loop  = 0;
while (loop < CPU_USAGE_LOOP) loop ++;
    }

/* calculate major and minor */
if (count < total_count) {                                  (7)
        count = total_count - count;
        cpu_usage_major = (count * 100) / total_count;
        cpu_usage_minor = ((count * 100) % total_count) * 100 / total_count;
    } else {
        total_count = count;                                        (8)

/* no CPU usage */
        cpu_usage_major = 0;
        cpu_usage_minor = 0;
    }
}

void cpu_usage_get(rt_uint8_t *major, rt_uint8_t *minor)
{
    RT_ASSERT(major != RT_NULL);
    RT_ASSERT(minor != RT_NULL);

    *major = cpu_usage_major;                                   (9)
    *minor = cpu_usage_minor;
}

void cpu_usage_init()
```

(下页继续)

(续上页)

```
{
/* 设置空闲线程钩子函数 */
    rt_thread_idle_sethook(cpu_usage_idle_hook);     (10)
}
```

代码清单 27-2**(1)**：在第一次进入该函数的时候，total_count 为 0，那么就在指定的时间段中 CPU 全速运算，看看能将 total_count 加一运算加到多大，并以 total_count 的值作为 CPU 利用率 100% 的运算标准。

代码清单 27-2**(2)**：进入临界段，不响应中断，CPU 全速运行。

代码清单 27-2**(3)**：获取当前时间 tick，也就是作为运算起始的时间点。

代码清单 27-2**(4)**：在一个相对时间 rt_tick_get() - tick < CPU_USAGE_CALC_TICK 里面，循环将 total_count 自加，CPU_USAGE_CALC_TICK 的大小由宏定义指定，用户可以修改其值，我们以 1000 个 tick 作为计算。时间到达之后，退出循环，我们也得到一个 CPU 全速运算的值 total_count。

代码清单 27-2**(5)**：获取当前时间 tick，也就是作为运算起始的时间点，这个获取当前系统时间是为了计算在指定的 CPU_USAGE_CALC_TICK 相对时间内，计算空闲任务占到的相对时间。

代码清单 27-2**(6)**：不进入临界段的 count 自加，可能 count 的运算会被系统其它任务或中断打断，这样子的运算我称之为空闲的 CPU 运算，只在空闲时间占用 CPU，因为空闲线程是永远处于运行的，而空闲任务是可以被我们粗略认为是做无用功的，CPU 没有被利用上。

代码清单 27-2**(7)**：假设在 CPU 全速运行的时候，total_count 自加到 100，而在有线程运行的时候，空闲线程是不能获得 CPU 的使用权，那么自然 count 也无法一直自加，所以 count 的值往往是比 total_count 要小的，假设某段时间 count 的值为 80，那么我们可以认为空闲线程占了系统的 80%CPU 所使用权，其他线程占用了 20%，而这 20% 是有用的，所以可以看做 CPU 的利用率是 20%，按照这个思想，将得到某个相对时间段中 CPU 的利用率，CPU 利用率的结果将保留两位小数，cpu_usage_major 是 CPU 利用率的整数部分，cpu_usage_minor 是 CPU 利用率的小数部分。

代码清单 27-2**(8)**：如果 count 的值大于等于 total_count 的值，那么就说明了这段时间 CPU 没有处理其他事情，基本都在空闲线程中做运算。

代码清单 27-2**(9)**：获取 CPU 利用率，并保存在传入的参数中。

代码清单 27-2**(10)**：CPU 利用率统计的初始化函数，设置空闲线程钩子函数，让空闲线程能调用到空闲钩子函数，从而能进行 CPU 利用率的统计。

> **注意：** 在使用 CPU 利用率统计之前必须先调用 cpu_usage_init() 函数，我们已经在 board.c 中进行初始化了。

30.4 CPU 利用率实验

CPU 利用率实验是是在 RT-Thread 中创建了两个线程，其中一个线程是模拟占用 CPU，另一个线程用于获取 CPU 利用率并通过串口打印出来。具体见加粗部分。

列表 3: 代码清单 27-3CPU 利用率实验

```
/**
*****************************************************************
* @file    main.c
* @author  fire
* @version V1.0
* @date    2018-xx-xx
* @brief   RT-Thread 3.0 + STM32 CPU 利用率统计
*****************************************************************
* @attention
*
* 实验平台: 野火  STM32 开发板
* 论坛     :http://www.firebbs.cn
* 淘宝     :https://fire-stm32.taobao.com
*
******************************************************************
*/

```

(下页继续)

(续上页)

```
/*
**************************************************************************
*                              包含的头文件
**************************************************************************
*/
#include"board.h"
#include"rtthread.h"


/*
**************************************************************************
*                                 变量
**************************************************************************
*/
/* 定义线程控制块 */
static rt_thread_t led1_thread = RT_NULL;
static rt_thread_t get_cpu_use_thread = RT_NULL;
/*
**************************************************************************
*                              函数声明
**************************************************************************
*/
static void led1_thread_entry(void* parameter);
static void get_cpu_use_thread_entry(void* parameter);

/*
**************************************************************************
*                             main 函数
**************************************************************************
*/
/**
* @brief  主函数
```

(下页继续)

(续上页)

```
* @param  无
* @retval 无
*/
int main(void)
{
/*
    * 开发板硬件初始化，RTT 系统初始化已经在 main 函数之前完成，
    * 即在 component.c 文件中的 rtthread_startup() 函数中完成了。
    * 所以在 main 函数中，只需要创建线程和启动线程即可。
    */

    rt_kprintf("这是一个 [野火]-STM32 全系列开发板-RTT-CPU 利用率统计实验\r\n");

    led1_thread =                           /* 线程控制块指针 */
        rt_thread_create( "led1",               /* 线程名字 */
                        led1_thread_entry,   /* 线程入口函数 */
                        RT_NULL,             /* 线程入口函数参数 */
                        512,                 /* 线程栈大小 */
                        3,                   /* 线程的优先级 */
                        20);                 /* 线程时间片 */

/* 启动线程，开启调度 */
if (led1_thread != RT_NULL)
        rt_thread_startup(led1_thread);
else
return -1;

    get_cpu_use_thread =                          /* 线程控制块指针 */
        rt_thread_create( "get_cpu_use",              /* 线程名字 */
                        get_cpu_use_thread_entry,   /* 线程入口函数 */
                        RT_NULL,             /* 线程入口函数参数 */
                        512,                 /* 线程栈大小 */
```

(下页继续)

(续上页)

```
                        5,                      /* 线程的优先级 */
                        20);                    /* 线程时间片 */

/* 启动线程，开启调度 */
if (get_cpu_use_thread != RT_NULL)
        rt_thread_startup(get_cpu_use_thread);
else
return -1;
}

/*
*****************************************************************************
*                                 线程定义
*****************************************************************************
*/

static void led1_thread_entry(void* parameter)
{
    rt_uint16_t i;

while (1) {
        LED1_TOGGLE;

/* 模拟占用 CPU 资源，修改数值作为模拟测试 */
for (i = 0; i < 10000; i++);

        rt_thread_delay(5);    /* 延时 5 个 tick */
    }
}

static void get_cpu_use_thread_entry(void* parameter)
{
```

(下页继续)

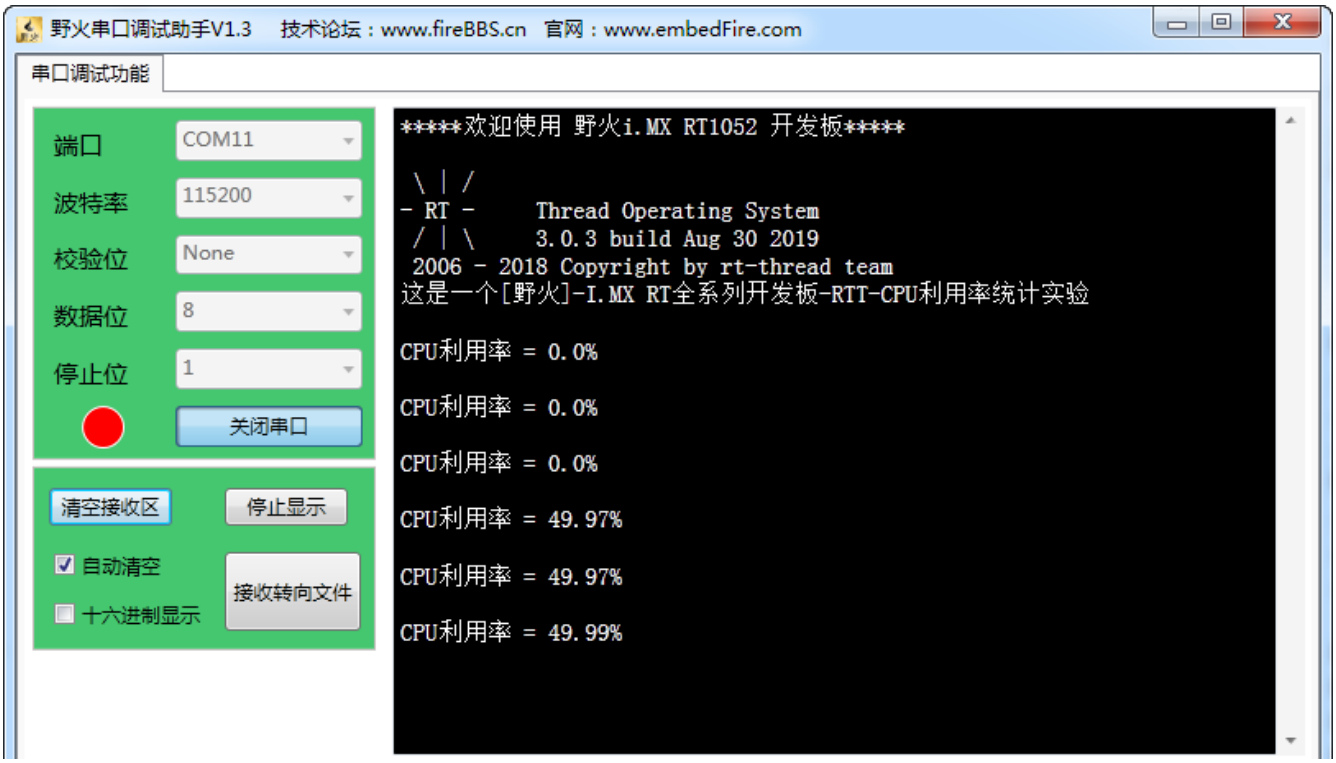
(续上页)

```
    rt_uint8_t major,minor;

while (1) {
/* 获取 CPU 利用率数据 */
        cpu_usage_get(&major,&minor);

/* 打印 CPU 利用率 */
        rt_kprintf("CPU 利用率 = %d.%d%\r\n",major,minor);

        rt_thread_delay(1000);   /* 延时 1000 个 tick */

    }
}

/*****************************END OF FILE****************************/
```

30.5 CPU 利用率实验现象

程序编译好，用 USB 线连接电脑和开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（具体型号根据你买的板子而定，每个型号的板子都配套有对应的程序），在电脑上打开串口调试助手，然后复位开发板就可以在调试助手中看到 rt_kprintf 的打印信息，具体见图 *27-3*。

注意：在开始的时候调用获取 CPU 利用率函数 cpu_usage_get() 是会进行一个计算参考值的，所以刚开始的时候 CPU 利用率为 0，后面的就是统计后的真正数据。


野火串口调试助手V1.3 技术论坛：www.fireBBS.cn 官网：www.embedFire.com
串口调试功能
端口
COM11
波特率
115200
校验位
None
数据位
8
停止位
1
关闭串口
清空接收区
停止显示
自动清空
十六进制显示
接收转向文件
*****欢迎使用 野火i.MX RT1052 开发板*****
\ | /
- RT - Thread Operating System
/ | \ 3.0.3 build Aug 30 2019
2006 - 2018 Copyright by rt-thread team
这是一个[野火]-I.MX RT全系列开发板-RTT-CPU利用率统计实验
CPU利用率 = 0.0%
CPU利用率 = 0.0%
CPU利用率 = 0.0%
CPU利用率 = 49.97%
CPU利用率 = 49.97%
CPU利用率 = 49.99%

第 31 章　rst 投稿示例文件

如何参与项目

本项目从代码至文档全部开源，期待大家的参与！

无论是小到发现错别字、发现代码写得不规范、有 bug 抑或是想参与到文档的编辑、投稿完整的文章，我们都热烈欢迎。

具体参与方式请阅读如下说明：如何参与野火开源项目

项目源码

本项目代码及文档源码托管在 github 和 gitee，维护时主要以 github 开发，gitee 主要是方便国内用户下载，可在这两个平台提交 **pull request** 参与到本项目。

- github 仓库地址：https://github.com/Embedfire-rtthread/ebf_rtthread_tutorial_rt1052.git
- gitee 仓库地址：https://gitee.com/Embedfire-rtthread/ebf_rtthread_tutorial_rt1052.git

文档在线阅读与 pdf

本项目的文档是托管在 **readthedocs** 网站的，可在该网站在线浏览文档或者生成 **pdf** 文档离线阅读。

常见问题

版权说明

野火电子保留本项目的所有版权。

公司组织有生存的压力，因为开源而导致生存不下去，是有 GEEK 精神的人不愿看到的事情。由于我们还不熟悉各种版权条例，所以关于版权的问题还在选择中。

目前我们保留本项目的所有版权，但我们秉承开源的心是不变的。如果你使用本项目不是用于商业目的，基本不需要考虑版权问题。

我们主要担忧的商业目的如下，列出来的意思是如果用于以下目的，我们极有可能会追究版权。

- 同业竞争，目前开发板是我们主要的盈利来源，禁止把本项目用于其它开发板项目。
- 文档出版，项目中包含的文档我们随时会提交至出版社出版，所以我们保留文档出版的权利。如果你只是在自己的文章中使用了本项目文档中的内容，需要注明来源。