

第一章 RTEMS 硬实时操作系统简介

概述

本书讲述的 RTEMS 则属于硬实时嵌入式操作系统，与常见的 Windows、Linux、solaris 等等经常用到的桌面操作系统有很大的不同，RTEMS 硬实时嵌入式操作系统主要应用于工业控制、航空航天、机器人、武器装备、交通管理等实时系统领域。

实时系统

实时系统 (Real-Time System) 是一种很特殊的系统，一般应用于嵌入式领域，与嵌入式系统有许多交集。但它与嵌入式系统有所区别。从技术角度上看，嵌入式系统是将应用程序、操作系统和计算机硬件集成在一起的系统。嵌入式系统是指以应用为中心、以计算机技术为基础，软硬件可裁剪，其针对的用户应用对功能、可靠性、成本、体积、功耗和使用环境有特殊要求的专用计算机系统。根据 IEEE (美国电气工程师协会) 对嵌入式系统的定义是用于控制、监视或者辅助操作的机器、设备或装置” (原文为 devices used to control, monitor, or assist the operation of equipment, machinery or plants)。

而实时系统的核心特征是**实时性**。实时性的本质是任务处理所化费时间的可预测性，即任务需要在规定的时限内完成。简单来说，实时系统就是能够对外部事件及时响应的系统，其响应时间是有保证的。外部事件可能是突发的大量信息，这些信息既可能是一串毫不相关的异步信息，也可能是一些相互关联具有同步关系的信息，还可能是循环信息。响应时间包括识别外部事件、执行对应的处理程序以及输出结果的总时间。包括两个重要特征：逻辑和功能正确性，以及时间的精确性。实时系统定义为那些依赖于功能正确性和耗时正确性的系统，且对时间要求的准确性的的重要性至少不低于功能的正确性。IEEE 对实时系统的定义是“那些正确性不仅取决于计算的逻辑结果也取决于产生结果所花费时间的系统”。对实时系统的要求是其行为不仅要是可预测的而且是能够满足系统时间约束的。

实时系统对响应时间的限制有着严格的要求，依据超过限制时间后系统计算结果的有效性可以将实时系统分为**硬实时系统**和**软实时系统**两类，也可以说是系统对于超时限的可容忍度。硬实时系统中，一旦超过规定的时限，系统的计算结果将完全失效或者是高度失效，系统将遭受毁灭性的灾难；(例如：核电站控制、水坝控制)而在软实时系统中，系统的计算结果将大打折扣，但不致于像硬实时系统一样失效，而是主要体现在系统的性能会有所下降。

实时操作系统

在实时系统中的一个关键组成部分是实时操作系统 (Real-Time Operating System, 简称 RTOS)。实时操作系统在实时系统中起着核心作用，整个实时系统是在实时操作系统的控制下来管理和协调各项工作，为应用软件提供良好的运行软件环境及开发环境。一般而言，实时操作系统是事件驱动的，能对来自外界的激励作用在限定的时间范围内作出正确的响应。用户应用是运行于实时操作系统之上的各个任务 (一般也可称为线程或进程)，任务可以是实时或非实时的，实时操作系统根据各个任务的要求，进行资源 (包括存储器、外设等) 管理、消息管理、任务调度、异常处理等工作。在实时操作系统中，每个任务均有一个优先级，实时操作系统根据各个任务的优先级，动态地切换各个任务，保证对实时性的要求。实时操作系统强调的是实时性、可靠性和灵活性，并能够与实时应用软件相结合形成完整的实时软件系统，完成对硬件的相应控制。

从实时系统的应用特点来看实时操作系统可以分为两种：一般实时操作系统和嵌入式实时操作系统。一般实时操作系统应用于实时处理系统的主机和实时查询系统等实时性较弱的实时系统，并且提供了开发、调试、运用一致的环境。嵌入式实时操作系统应用于实时性要求高的实时控制系统，而且应用程序的开发过程是通过交叉编译开发环境来完成的，即开发环境与运行环境是不一致。嵌入式实时操作系统

具有所占存储空间小、可固化使用\实时性强(在毫秒或微秒数量级上)的特点。一般认为实时操作系统应具备以下几点:

- ❏ 异步事件响应能力
- ❏ 切换时间和中断延迟时间确定
- ❏ 基于优先级的中断和调度
- ❏ 抢占式调度
- ❏ 内存锁定
- ❏ 连续文件
- ❏ 同步互斥

实时操作系统也分为软实时操作系统和硬实时操作系统,不同的实时操作系统对于时限的要求是不一样的。从实践上说,软实时和硬实时之间的区别通常与系统的时间精度有关。由于这个原因,典型的软实时任务的调度精度必须大于千分之一秒,而典型的硬实时任务为微秒级。著名的硬实时操作系统包括 RTEMS、VxWorks、ThreadX、Nucleus、QNX、OSE、LynxOS、RTLinux、RTAI 等。软实时操作系统则有 WinCE, Linux 2.6.x。

相关的基本概念

本节描述的与实时操作系统相关的基本概念,这些概念对理解后续章节的 RTEMS 硬实时嵌入式操作系统设计与实现会有帮助。本节的内容节选自邵贝贝老师翻译的《uCOS-II 操作系统》一书的第二章,并进行了部分的改动。

操作系统内核 (Operating System Kernel)

操作系统是计算机系统上的系统软件,它是这样一些程序模块的集合:它们能有效地组织和管理计算机系统上的软硬件资源,合理地组织计算机工作流程,控制程序的执行,并向用户提供各种服务功能,使得用户能够灵活、方便、有效地使用计算机,使整个计算机系统能高效地运行。操作系统内核是操作系统中的核心部分,在多任务系统中,操作系统内核提供的最基本服务是任务调度与切换、中断服务。操作系统内核为每个任务分配 CPU 时间,并且负责任务之间的通信。任务调度与切换是实现多任务并发运行的重要机制,而中断服务是实现可抢占的任务调度与切换的高效手段(其中最主要一种中断服务的是时间中断管理),同时也是实现与外设交互的必要措施。扩展的服务包括同步互斥、内存管理、文件系统管理、网络管理等,在具体的实时操作系统中不一定必须提供。操作系统内核一般提供同步互斥等服务,如信号量、消息队列等。操作系统内核提供的文件管理为保存和读写需要长期保存的数据提供了一种有效的方法。操作系统内核提供的网络管理主要是包括基于 TCP/IP 或其它通信协议的协议栈实现。一般而言,操作系统内核本身对 CPU 的占用时间开销一般占整个应用系统开销的5%以下。

调度 (Scheduling)

调度 (Scheduling) 的作用是在多任务系统中,决定运行哪个任务。这是操作系统内核的主要职责之一。多数实时内核是基于优先级调度算法,即每个任务根据其重要程度的不同被赋予一定的优先级,由操作系统内核选择某一个优先级最高且可以马上执行的任务占用 CPU 来运行。根据何时让高优先级任务掌握 CPU 的使用权,可以把操作系统内核分为两种类型,即不可抢占型操作系统内核和可抢占型操作系统内核。

不可抢占型内核 (Non-Preemptive Kernel)

不可抢占型内核要求每个任务自我放弃 CPU 的所有权,所以不可抢占型调度也称作合作型多任务调度,其调度的完成需要基于各个任务对 CPU 主动“弃权”,只要任务自己不放弃 CPU,它就可以一直运行下去。虽然对于外设的异步事件还是由中断服务来处理,即中断服务可以使一个高优先级的任务由阻塞状态变为就绪状态,但中断服务以后控制权还是回到原来被中断了的那个任务。只有等该任务主动放弃 CPU 的使用权时,那个高优先级的任务才能获得 CPU 的使用权。

在不可抢占型内核中,由于每个任务要运行到自身结束时才释放 CPU 的控制权,所以可以在任务中使用不可重入函数(函数的可重入性在本节的靠后部分有介绍),而不必担心其它任务可能也会正在使用该

函数，从而造成对共享数据的破坏。当然该不可重入函数本身不能放弃 CPU 控制权。不可抢占型内核的另一个优点是，几乎不需要使用某种互斥机制（如信号量等）保护共享数据。这时由于运行着的任务始终占有 CPU，而不必担心被别的任务抢占。但这也不是绝对的，比如在处理 I/O 设备时，可能存在中断服务例程与应用任务共享某些资源，这就仍需要使用某种互斥机制来“安全”地访问这些共享资源。不可抢占型内核的主要缺点是响应时间过于缓慢。其原因在于虽然高优先级的任务已经进入就绪态，但还不能运行，要等到当前运行着的任务释放 CPU 后才能运行。这导致不可抢占型内核的任务级响应时间是不确定的，不知道什么时候最高优先级的任务才能获得 CPU 的控制权。正由于这个缺陷，使得实时操作系统中，不可抢占型内核用的比较少（但是由于非抢占型内核比较简单，一些开源的嵌入式系统，例如 NutOS 仍然在使用非抢占内核，FreeRTOS 也同时支持非抢占和可抢占内核）。

可抢占型内核（Preemptive Kernel）

绝大多数的实时操作系统内核都是可抢占型内核。在可抢占型内核中，处于最高优先级的任务一旦就绪（即就等 CPU 这个资源了），内核就会马上调度此任务，让这个最高优先级的任务总能得到 CPU 的控制权。我们也可以换一种说法，当一个运行着的任务使一个比它优先级高的任务进入了就绪态，当前任务的 CPU 使用权就被抢占了，或者说被阻塞了，那个高优先级的任务立刻得到了 CPU 的控制权。如果是中断服务例程使一个高优先级的任务进入就绪态，中断完成时，中断了的任务被阻塞，优先级最高的那个任务（不一定是那个被中断了的任务）开始运行。使用可抢占型内核，最高优先级的任务何时可以得到 CPU 的控制权并运行是可确定的。这样使用可抢占型内核使得任务级响应时间得以最优化。

使用可抢占型内核时，应用程序不应直接使用不可重入型函数。如果调用不可重入型函数时，低优先级的任务被高优先级任务抢占，那么不可重入型函数中的数据有可能被破坏。如果确实需要调用不可重入型函数，那么在调用不可重入型函数时，必须满足互斥条件，这一点可以用信号量等互斥机制来实现。

任务（task）

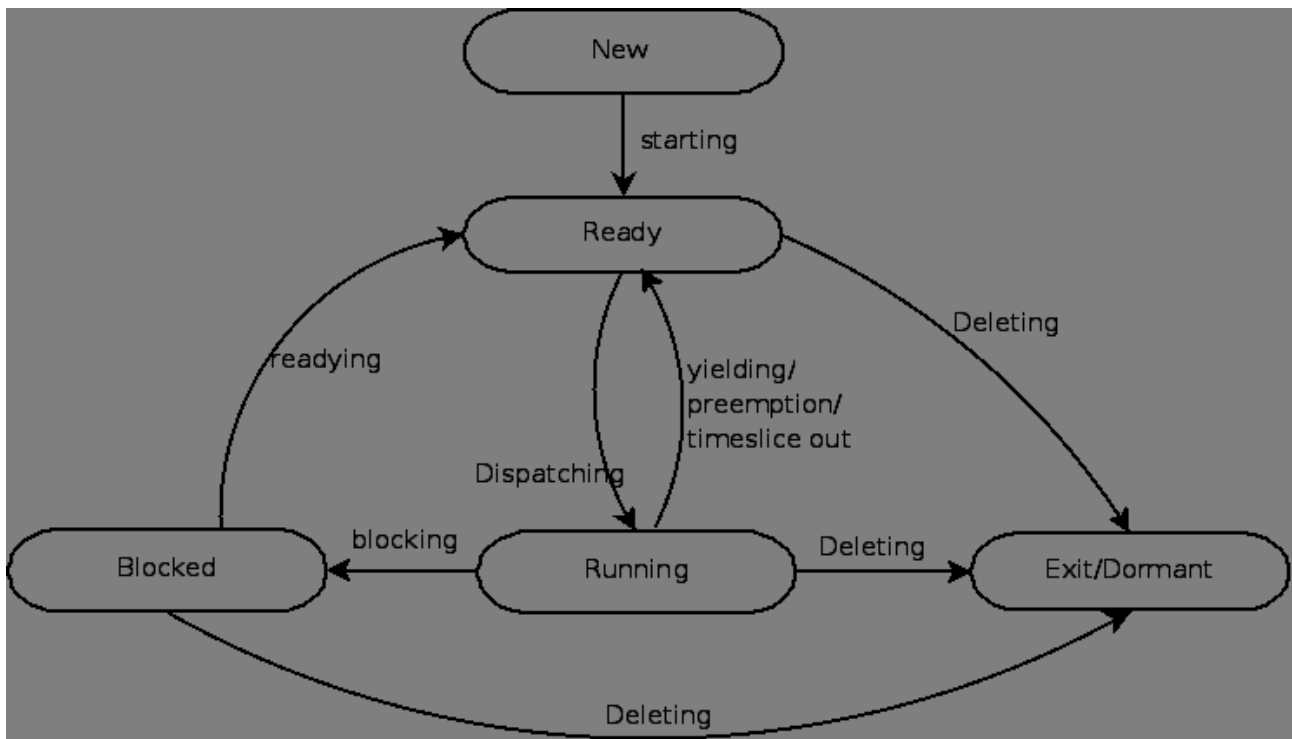
在前面章节已经多次提到过任务了，但任务到底是什么呢？从操作系统原理上看，一个任务是一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。任务的组成包括程序、数据和任务控制块（即任务状态信息）。在实时操作系统中，任务一般也称作线程（thread），因为在一般的实时操作系统中的所有任务共享操作系统提供的各种资源（内存、CPU、同步互斥资源等），且与实时操作系统一起位于同一地址空间，都处于特权态。每个任务都是整个应用的某一部分，每个任务被赋予一定的优先级，且每个任务在逻辑上维护了任务的运行状态信息，即与任务运行直接相关的 CPU 寄存器和栈空间（只有这样才能实现任务切换）。实时应用实际上是由多个任务组成，由实时操作系统根据当前任务的情况设置任务的状态，并根据任务的优先级进行调度。

任务状态

在任务的运行过程中，每个任务都处在以下五种状态之一：

- ❏ 创建（New）态：一个任务正在被创建，还没被转到就绪状态之前的状态。
- ❏ 就绪（Ready）态：该任务获得了除 CPU 之外的一切所需资源，已经准备好运行了，但由于还没有占用 CPU，所有还暂时不能运行，一旦得到处理机即可运行。
- ❏ 运行（Running）态：该任务获得了 CPU 使用权，正在运行中。
- ❏ 阻塞（Blocked）态：该任务在等待某一事件（如某个资源可用的事件等）发生，而暂停运行。
- ❏ 退出（Exit/Dormant）态：一个任务正在从系统中消失时的状态，这是因为任务结束或由于其他原因所导致。

这五种状态的关系如下图所示：



任务状态变化图

可能的状态变化如下所示：

- ❏ NULL→New：一个新任务被产生出来执行某种功能。
- ❏ New→Ready：当任务被创建完成并初始化后，准备运行时，变为就绪状态。
- ❏ Ready→Running：处于就绪状态的任务被任务调度程序选中后，就分配到 CPU 上来运行
- ❏ Running→Exit：当任务已经完成或者出错，操作系统会将其结束。
- ❏ Running→Ready：处于运行状态的任务在其运行过程中，由于分配给它的处理机时间片用完或被中断服务例程、高优先级任务抢占而让出 CPU。
- ❏ Running→Blocked：当任务请求某资源且必须等待。
- ❏ Blocked→Ready：当任务要等待的事件到来时，它从阻塞状态变到就绪状态。

多任务

多任务运行的实现实际上是靠操作系统利用中断机制和调度机制，让不同的任务按照优先级等因素分时占用 CPU。多任务运行使 CPU 得到充分利用，并使应用模块化。

任务调度算法

当多个就绪任务的优先级不同时，选择优先级高的任务获得 CPU 控制权，这种调度方法称为优先级调度法。当两个或两个以上任务有同样优先级，有两种方法可以采用。一个是 FIFO（先来先出）调度法，即按照任务的到达顺序来选择任务获得 CPU 控制权的顺序，只有当一个任务执行完毕或阻塞后，下一个同优先级的任务才能获得 CPU 控制权。内核允许一个任务运行确定好的一个时间片（quantum，也称时间额度），然后切换给另一个任务。这种调度方法称为时间片轮转调度法，也称时间片调度。内核在满足以下条件时，把 CPU 控制权交给下一个任务就绪态的任务：

- ❏ 当前任务的时间片用完了
- ❏ 当前任务在时间片还没结束时已经完成了

可重入性 (Reentrancy)

可重入型函数可以被一个以上的任务调用，而不必担心数据的破坏。可重入型函数任何时候都可以被中断，一段时间以后又可以继续运行，而相应数据不会丢失。可重入型函数一般只使用局部变量，即变量保存在 CPU 寄存器中或堆栈中。如果使用全局变量，则要对全局变量予以互斥保护。下面的程序片段是一个可重入型函数的例子。

```
void strcpy(char *dest, char *src)
{
    while (*dest++ = *src++) {};
    *dest = NUL;
}
```

函数 strcpy 实现字符串复制功能。因为参数是存在堆栈中的，故函数 strcpy 可以被多个任务调用，而不必担心各任务调用函数期间会互相破坏对方的指针。

不可重入型函数的例子如下所示。strcpy2 是一个简单函数，它也实现了字符串复制。为便于讨论，假定使用的是可抢占型内核，允许中断，temp 是全局变量。

```
char temp;

void strcpy2( char *dest, char *src)
{
    while (*src!=NULL) {
        temp=*src;
        src++;
        *dest=temp;
        dest++;
    }
    *dest = NUL;
}
```

假设 strcpy2 函数可以为任何任务所调用，如果一个低优先级的任务 A 正在执行 strcpy2 函数，把 temp 赋值为 'a' 后，而此时中断发生了，打断了任务 A 的执行，中断服务例程执行，当中断服务例程执行完毕后，操作系统内核使最高优先级的就绪任务 B 运行，任务 B 也调用 strcpy2 函数，且把 temp 赋值为 'b'。这对任务 B 本身来说，实现两个变量的交换是没有问题的。然后当任务 B 运行完毕后，释放了 CPU 的使用权，低优先级任务 A 得以继续运行。而此时 temp 的值仍为 'b'。这样导致任务 A 执行的结果出错。

使用以下技术之一即可使 strcpy2 函数具有可重入性：

- ❏ 把 temp 定义为局部变量
- ❏ 调用 strcpy2 函数之前屏蔽中断，调用后再使能中断
- ❏ 用信号量禁止该函数在使用过程中被再次调用

上下文切换 (Context Switch or Task Switch)

在操作系统中通常也把上下文切换称为任务切换 (task switch)。当操作系统决定运行另外的任务时，它需要保存当前正在运行任务的当前上下文 (Context，也可称“运行状态信息”)，即 CPU 寄存器中的全部内容。这些内容保存在任务的堆栈空间或特定的上下文保存区。完成保存工作后，操作系统就可以把下一个将要运行任务的当前上下文从该任务的栈或上下文保存区中恢复到 CPU 的寄存器中，这样就可以继续下一个任务的运行。这个过程叫做任务切换。一般有两种情况的上下文切换：

- ❏ 高优先级的任务因为需要某种临界资源，主动请求阻塞，让出处理器，此时将调度就绪状态的低优先级任务获得执行。这种切换称为任务级的上下文切换。
- ❏ 任务因为时钟中断或其它中断到来而被打断，在中断服务例程处理完毕后，内核发现有更高优先级任务处于就绪态，则在中断处理结束后直接切换到高优先级任务执行。这种调度也称为中断级的上下文切换。

任务优先级

每个任务都有其优先级。任务越重要，赋予的优先级应越高。

静态任务优先级

应用在执行过程中，各个任务优先级不变，则称任务的优先级为静态任务优先级。在这样的实时系统中，各个任务以及它们的时间约束在应用程序编译时是已知的。

动态任务优先级

应用执行过程中，各个任务的优先级会随着时间的流逝而改变，则称任务的优先级为动态任务优先级。

优先级反转

使用实时内核，优先级反转问题是实时系统中出现得最多的问题。例如在一个实时应用中，包含3个实时任务，任务1优先级最高，任务2其次，任务3优先级最低。任务1和任务2处于阻塞状态，等待某一事件的发生，任务3正在运行。下面是一个产生优先级反转问题的情况：

1. 开始时（时刻为 T1），任务3要使用共享资源 A。使用共享资源 A 之前，首先必须得到该资源的信号量(Semaphore)。假设任务3得到了该信号量，并开始使用该共享资源 A。
2. T2时，由于任务1优先级高，它等待的事件到来之后，会抢占任务3的 CPU 使用权，任务1开始运行。运行过程中任务1也要使用那个任务3正在使用着的资源 A，由于该资源的信号量还被任务3占用着，任务1只能进入阻塞状态，等待任务3释放该信号量。
3. T3时，任务3得以继续运行。由于任务2的优先级高于任务3，当任务2等待的事件发生后，任务2抢占了任务3的 CPU 的使用权并开始运行。处理它该处理的事件，直到处理完之后将 CPU 控制权还给任务3。
4. T4时，任务3接着运行，完成对共享资源 A 操作，释放信号量。
5. T5时，由于实时内核知道有个高优先级的任务在等待这个信号量，内核做任务切换，使任务1得到该信号量并接着运行。

在这种情况下，任务1优先级实际上降到了任务3 的优先级水平。因为任务1要等，直等到任务3释放占有的共享资源。由于任务2抢占任务3的 CPU 使用权，使任务1的状况更加恶化，任务2使任务1增加了额外的延迟时间。任务1和任务2的优先级发生了反转。

可以通过优先级继承(Priority inheritance)协议和优先级天花板(Priority Ceiling)协议解决优先级反转问题，但是这样会增加一些时间开销。RTEMS 支持优先级继承和优先级天花板方法。

同步互斥与任务间通信

资源

任何为任务所占用的实体都可称为资源。资源可以是 CPU、内存，也可以是 I/O 设备，还可以是一个变量，一个结构或一个数组等。

共享资源

可以被一个以上任务使用的资源叫做共享资源。为了防止数据被随意访问（特别是执行写操作），每个任务在与共享资源打交道时，必须独占该资源。这叫做互斥 (*mutual exclusion*)。需要互斥访问的共享资源称为临界资源。

竞争状态 (race condition)

竞争状态是指两个或多个任务对同一共享资源同时进行读写操作，而最后的结果是不可预测的，该结果取决于各个任务具体运行情况。

程序临界区

对共享资源的访问，可能导致竞争状态的出现。我们把可能出现竞争态的程序片断称为程序临界区。程序临界区在处理时不可以被中断，要保证其操作的原子性。为确保临界区程序执行过程中不被中断，在进入临界区之前要屏蔽中断，而临界区代码执行完以后要立即使能中断，以减少对中断处理延迟的影响。

互斥条件

当所有的任务都在一个单一地址空间下，实现任务间通信最简便的办法是使用基于全程变量的共享数据结构（指针、缓冲区、链表、循环缓冲区）。虽然共享数据区法简化了任务间的信息交换，但是必须保证每个任务在处理共享数据时的排它性和互斥性，以避免竞争和数据的破坏。解决临界区问题应满足的要求包括：

- ❏ 互斥：如果一个任务在其临界区执行，那么其它任务不能在其临界区内执行。
- ❏ 有空让进（前进要求）：如果没有任务在临界区，并且有任务希望进入临界区，那么只有那些不在剩余区内的任务能参加决策，且不能选择无限等待；
- ❏ 有限等待：在一个任务 T_i 做出进入临界区的请求到该请求被允许期间，其它任务 T_j 被允许进入其临界区的等待时间（ P_i 进入临界区的次数）存在一个上限（任务不能“死等”）。

与共享资源打交道时，使之满足互斥条件以避免临界区问题的最一般的方法有：

- ❏ 屏蔽中断（也称关中断）
- ❏ 使用测试并置位指令
- ❏ 禁止任务切换
- ❏ 信号量

屏蔽中断和使能中断

处理共享数据时保证互斥，最简便快捷的办法是屏蔽中断（关中断）和使能中断（开中断）。示意性代码如下所示：

```
屏蔽中断
读写基于全局变量的共享数据
使能中断
```

可是，必须十分小心，屏蔽中断的时间不能太长。因为它影响整个系统的中断响应时间，即中断延迟时间。当改变或复制某几个变量的值时，应想到用这种方法来做。这也是在中断服务例程中处理共享变量或共享数据结构的唯一方法。在任何情况下，屏蔽中断的时间都要尽量短。

如果使用某种实时内核，一般地说，屏蔽中断的最长时间不超过内核本身的屏蔽中断时间，就不会影响系统中断延迟。

测试并置位指令

如果不使用实时内核，当两个任务共享一个资源时，一定要约定好，先测试某一全程变量，如果该变量是0，允许该任务与共享资源打交道。为防止另一任务也要使用该资源，前者只要简单地将全程变量置为1，这通常称作测试并置位(Test-And-Set)操作，或称作 TAS 操作。TAS 操作一般是 CPU 提供的一条不会被中断的指令。通过这个指令可以实现对共享资源的互斥访问。TAS 指令读出标志后设置为 TRUE，下面是 TAS 指令的逻辑流程：

```
boolean TAS(boolean *lock) {
    boolean old;
    old = *lock; *lock = TRUE;
    return old;
}
```

lock 表示资源的两种状态：TRUE 表示正被占用，FALSE 表示空闲。

禁止任务切换，然后允许任务切换

如果任务不与中断服务例程共享变量或数据结构, 可以先禁止任务切换, 然后再允许任务切换的方法。这样两个或两个以上的任务可以共享数据而不发生冲突。注意, 此时虽然任务切换是禁止了, 但中断还是开着的。如果这时中断来了, 中断服务例程会在这一临界区内立即执行。中断服务例程结束时, 尽管有优先级高的任务已经进入就绪态, 由于设置了禁止任务切换, 所以内核还是返回到原来被中断了的任务。直到执行完允许任务切换函数后, 内核再看有没有优先级更高的就绪态任务, 如果有, 则做任务切换。虽然这种方法是可行的, 但应该尽量避免禁止任务切换之类操作, 因为内核最主要的功能就是做任务的调度与协调。禁止任务切换显然与内核的初衷相违。

信号量(Semaphores)

信号量和对应的 P/V 原语是二十世纪六十年代中期荷兰的计算机科学家 Edgser Dijkstra 提出的。所以 P、V 分别是荷兰语的 test (proberen) 和 increment (verhogen) 的意思。信号量实际上是一种同步约定机制, 在操作系统内核中普遍使用。信号量用于:

- 控制共享资源的使用权 (满足互斥条件)
- 标志某事件的发生
- 使两个任务的行为同步

信号像是一把钥匙, 任务要运行下去, 得先拿到这把钥匙。如果信号已被别的任务占用, 该任务只得被阻塞, 直到信号被当前使用者释放。换句话说, 申请信号的任务是在说: “把钥匙给我, 如果谁正在用着, 我只好等!”。

在实现上, 每个信号量 s 除一个整数值 s.count (计数值) 外, 还有一个任务等待队列 s.queue, 其中是阻塞在该信号量的各个任务的标识。信号量结构体的一般定义如下:

```
typedef struct {
    int count;          // 计数值变量
    struct TCB *queue; // 任务等待队列
} semaphore;
```

信号量有两种类型: 二值 (binary) 信号量, 计数 (counting) 信号量。二值信号量只有两个计数值 0 和 1 的信号量。计数信号量的计数值可以是大于 1 的, 具体的范围取决于信号量规约机制使用的是 8 位、16 位还是 32 位。

一般地说, 对信号量只能实施三种操作: 初始化 (INITIALIZE), 也可称作建立 (CREATE); 等信号 (WAIT) 也可称作阻塞 (SUSPEND); 给信号 (SIGNAL) 或发信号 (POST)。

信号量初始化时要给信号量赋初值, 初始化指定一个非负整数值, 表示空闲资源总数 (又称为“资源信号量”) —— 若为非负值表示当前的空闲资源数, 若为负值其绝对值表示当前等待临界区的任务数。等待信号量的任务等待队列应清为空。

想要得到信号量的任务执行等待 (WAIT) 操作。如果该信号量有效 (即信号量值大于 0), 则信号量值减 1, 任务得以继续运行。如果信号量的值为 0, 等待信号量的任务就被列入等待信号量的任务等待队列中。多数内核允许用户定义等待超时, 如果等待时间超过了某一设定值时, 该信号量还是无效, 则等待信号量的任务进入就绪态准备运行, 并返回出错代码 (指出发生了等待超时错误)。等待操作的实现逻辑如下所示:

```
wait(semaphore s) {
    --s.count;          //表示申请一个资源;
    if (s.count < 0) //表示没有空闲资源;
    {
        把任务放入任务等待队列 s.queue;
        阻塞调用任务;
    }
}
```

任务以发信号操作 (SIGNAL) 释放信号量。如果没有任务在等待信号量, 信号量的值仅仅是简单地加 1。如果有任务在等待该信号量, 那么就会有一个任务进入就绪态, 信号量的值也就不加 1。于是钥匙给了等待信号量的诸任务中的一个任务。至于给了那个任务, 要看内核是如何调度的。收到信号量的任务可能

是以下两者之一。

- ☒ 等待信号量任务中优先级最高的任务（基于优先级排队）
 - ☒ 最早开始等待信号量的那个任务（基于按先进先出的原则(First In First Out , FIFO)）
- 发信号操作的实现逻辑如下所示：

```
post(semaphore s){
    ++s.count;           //表示释放一个资源;
    if (s.count <= 0) //表示有任务处于阻塞状态;
    {
        从任务等待队列 s.queue 中取出一个任务 T;
        任务 T 进入任务就绪队列;
    }
}
```

处理简单的共享变量也使用信号量则是多余的。因为请求和释放信号量的过程是要花相当的时间。这种情况下只需要屏蔽中断、使能中断来处理简单共享变量，便可以有效地提高效率。如果屏蔽中断后的处理时间很长，会影响中断延迟时间，这种情况下就有必要使用信号量了。

死锁(或抱死) (Deadlock (or Deadly Embrace))

在多任务环境中，一组任务中的每个任务都占用着若干个资源，同时又在等待得到该组任务中另一任务所占用的资源，因而造成的所有任务都无法进展下去的现象，这种现象称为死锁(也称作抱死)，这一组任务就称为死锁任务。在死锁状态下，每个任务都动弹不得，既无法运行，也无法释放所占用的资源，它们互为因果、互相等待。例如，设任务 T1正独享资源 R1，任务 T2在独享资源 T2，而此时 T1又要独享 R2，T2也要独享 R1，于是哪个任务都没法继续执行了，发生了死锁。只有当以下四个条件同时成立时，才会出现死锁：

- ☒ 互斥：在任何时刻，每一个资源最多只能被一个任务所使用；
- ☒ 占有并等待：任务在占用若干个资源的同时又可以请求新的资源；
- ☒ 非抢占：任务已经占用的资源，不会被强制性拿走，而必须由该任务主动释放；
- ☒ 循环等待：存在一条由两个或多个任务所组成的环路链，其中每一个任务都在等待环路链中下一个任务所占用的资源。

最简单的防止发生死锁的方法是让每个任务都：

- ☒ 先得到全部需要的资源再做下一步的工作
- ☒ 用同样的顺序去申请多个资源
- ☒ 释放资源时使用相反的顺序

内核大多允许用户在申请信号量时定义等待超时，以此化解死锁。当等待时间超过了某一确定值，信号量还是无效状态，就会返回某种形式的出现超时错误的代码，这个出错代码告知该任务，无法得到资源使用权。死锁常出现在发生在复杂多任务系统中。

同步

同步指任务之间或任务与内核之间的执行流程中需要按照有固定的时序进行。例如一个任务向内核发出了一个系统调用，在内核没有完成系统调用之前，该任务一直处于等待的状态，只有等内核执行完系统调用后，该任务才能继续执行。一般可以利用信号量使某任务与中断服务同步(或者是与另一个任务同步，这两个任务间没有数据交换)。

如果内核支持计数信号量，那么计数信号量的值表示尚未得到处理的事件数。请注意，可能会有一个以上的任务在等待同一事件的发生，则这种情况下内核会根据以下原则之一发信号给相应的任务：

- ☒ 等待信号量任务中优先级最高的任务（基于优先级排队）
- ☒ 最早开始等待信号量的那个任务（基于按先进先出的原则(First In First Out , FIFO)）

任务间的通信(Intertask Communication)

多任务系统中，有时需要任务间的或任务与中断服务间传递消息。这种信息传递称为任务间的通信。

任务间信息的传递有两个途径：

- ☞ 通过全程变量或发消息给另一个任务
- ☞ 使用邮箱或消息队列等通信机制

用全程变量时，必须保证每个任务或中断服务程序独享该变量。中断服务中保证独享的唯一办法是屏蔽中断。如果两个任务共享某变量，各任务实现独享该变量的办法可以是先屏蔽中断再使能中断，或使用信号量(如前面提到的那样)。

需要注意的是，任务只能通过全程变量与中断服务程序通信，但是任务本身并不知道什么时候全程变量被中断服务程序修改了，除非中断程序以信号量方式向任务发信号或者是该任务以查询方式不断周期性地查询变量的值。如果需要避免这种情况，用户可以考虑使用邮箱或消息队列等通信机制。

中断

中断是一种硬件机制，用于通知 CPU 有个异步事件发生了。中断一旦被识别，CPU 保存部分(或全部)现场(Context)即部分或全部寄存器的值，跳转到操作系统提供的专门函数进行处理，称为中断服务例程(ISR)。中断服务例程对中断源进行相关处理，处理完成后，程序回到：

- ☞ 对不可抢占型内核而言，程序回到被中断了的任务
- ☞ 对可抢占型内核而言，让进入就绪态的优先级最高的任务开始运行

中断使得 CPU 可以在事件发生时才予以处理，而不必让微处理器连续不断地查询(Polling)是否有事件发生。通过两条特殊指令：屏蔽中断(Disable interrupt)和使能中断(Enable interrupt)可以分别让微处理器不响应或响应中断。在实时环境中，屏蔽中断的时间应尽可能的短。屏蔽中断影响中断延迟时间。屏蔽中断时间太长可能会引起中断丢失。微处理器一般允许中断嵌套，也就是说在中断服务期间，微处理器可以识别另一个更重要的中断，并服务于那个更重要的中断。对于实时操作系统而言，中断性能是一个很重要的评价指标。

中断延迟时间

实时内核最重要的指标之一就是中断关了多长时间。所有实时系统在进入临界区代码段之前都要屏蔽中断，执行完临界代码之后再使能中断。屏蔽中断的时间越长，中断延迟就越长。

中断延迟时间 = 屏蔽中断的最长时间 + 开始执行中断服务例程的第一条指令的时间

中断响应时间

中断响应定义为从中断发生到开始执行用户的中断服务例程代码来处理该中断的时间。中断响应时间包括开始处理这个中断前的全部开销。在典型情况下，执行用户代码之前要保护现场，将CPU的各寄存器推入堆栈。这段时间将被记作中断响应时间。

对于可抢占型内核，则要先调用一个特定的函数，该函数通知内核即将进行中断服务，使得内核可以跟踪中断的嵌套。可抢占型内核的中断响应时间的表达如下：

中断响应 = 中断延迟 + 保存 CPU 内部寄存器的时间 + 内核的进入中断服务函数的执行时间

中断响应是系统在最坏情况下的响应中断的时间，某系统100次中有99次在10 μs 之内响应中断，只有一次响应中断的时间是100 μs，只能认为中断响应时间是100 μs。

中断恢复时间(Interrupt Recovery)

中断恢复时间定义为微处理器返回到被中断的程序代码所需要的时间。对于可抢占型内核，中断的恢复要复杂一些。通常，在中断服务例程的末尾，要调用一个由实时内核提供的函数。这个函数用于判断中断是否脱离了所有的中断嵌套。如果脱离了嵌套(即已经可以返回到被中断了的任务级时)，内核要判断，由于中断服务例程的执行，是否使得一个优先级更高的任务进入了就绪态。如果是，则要让这个优先级更高的任务开始运行。在这种情况下，被中断了的任务只有重新成为优先级最高的任务而进入就绪态时才能继续运行。对于可抢占型内核，中断恢复时间的表达如下：

中断恢复时间 = 判断是否有优先级更高的任务进入了就绪态的时间 + 恢复高优先级任务的 CPU 内部寄

存器的时间 + 执行中断返回指令的时间

中断处理时间

虽然中断服务的处理时间应该尽可能的短，但是对处理时间并没有绝对的限制。不能说中断服务必须全部小于 $100\mu\text{s}$ ， $500\mu\text{s}$ 或 1ms 。如果中断服务是在任何给定的时间开始，且中断服务程序代码是应用程序中最重要的代码，则中断服务需要多长时间就应该给它多长时间。然而在大多数情况下，中断服务例程应识别中断来源，从触发中断的设备取得数据或状态，并通知等待该数据或者状态的任务。当然应该考虑到是否通知一个任务去做事件处理所花的时间比处理这个事件所花的时间还多。在中断服务中通知一个任务做时间处理(通过信号量、邮箱或消息队列)是需要一定时间的，如果事件处理需花的时间短于给一个任务发通知的时间，就应该考虑在中断服务例程中做事件处理并在中断服务例程中使能中断，以允许优先级更高的中断打入并优先得到服务。

时钟节拍(Clock Tick)

时钟节拍是特定的周期性中断。这个中断可以看作是系统心脏的脉动。中断之间的时间间隔取决于不同的应用，一般在 1ms 到 100ms 之间。时钟的节拍式中断使得内核可以将任务延时若干个整数时钟节拍，以及当任务等待事件发生时，提供超时等待。时钟节拍率越快，系统的额外开销就越大。各种实时内核都有将任务延时若干个时钟节拍的功能。然而这并不意味着延时的精度是1个时钟节拍，只是在每个时钟节拍中断到来时对任务延时做一次裁决而已。

对存储器的需求

使用多任务实时内核的应用所需空间大小取决于多种因素。代码空间总需求量为：

总代码量 = 应用程序代码 + 内核代码

由于每个任务都是独立运行的，所以至少需要给每个任务提供单独的栈空间。应用程序设计人员决定分配给每个任务多少栈空间时，应该尽可能使之接近实际需求量（有时，这是相当困难的一件事）。栈空间的大小不仅仅要计算任务本身的需求（局部变量、函数调用等等），还需要计算最多中断嵌套层数(保存寄存器、中断服务程序中的局部变量等)。根据不同的目标微处理器和内核的类型，任务栈和系统栈可以是分开的。系统栈专门用于处理中断级代码。这样做有许多好处，每个任务需要的栈空间可以大大减少。所有内核都需要额外的栈空间以保证内部变量、数据结构、队列等。如果内核支持任务栈和系统栈（用于中断），且二者分离，总 RAM 需求量为：

RAM 总需求 = 应用程序数据区的 RAM 需求 + 内核数据区的 RAM 需求 + 各任务栈之总和的 RAM 需求 + 最多中断嵌套栈的 RAM 需求

除非有特别大的 RAM 空间可以所用，对栈空间的分配与使用要非常小心。为减少应用程序需要的 RAM 空间，对每个任务栈空间的使用都要非常小心，特别要注意以下几点：

- ❏ 定义函数和中断服务例程中的局部变量，特别是定义大型数组和数据结构
- ❏ 函数(即子程序)的嵌套
- ❏ 中断嵌套
- ❏ 库函数需要的栈空间
- ❏ 多变元的函数调用

RTEMS 硬实时操作系统简介

RTEMS 是 Real-Time Executive for Multiprocessor Systems 的首字缩写。RTEMS 是前美国军方研制的嵌入式系统，最早用于美国国防系统，早期的名称为实时导弹系统 (Real Time Executive for Missile Systems)，后来改名为实时军用系统 (Real Time Executive for Military Systems)，由 OAR 公司接管维护后，全称改为目前的表示。它最突出的特点是非常的稳定，而且速度快。RTEMS 的性能不逊于老牌的商业系统 VxWorks，因此在通信、航空航天、工业控制、军事等领域有着非常广泛的应用。

RTEMS 诞生于二十世纪八十年代，最早用于美国国防部的导弹控制系统，它在诞生时就瞄准了高实时

性和高可用性的标准，并且采用了面向对象技术、构件技术等到现在也非常先进的设计理念，不但有非常好的实时性和稳定性，也具有非常好的可复用性。RTEMS 从设计初始就是为较少内存与较少 CPU 主频的嵌入式系统提供可靠的实时内核。该系统分为若干模块，用户可以根据实际应用需要进行剪裁。经过了20多年的持续开发，RTEMS 拥有非常广泛的客户，也具有丰富的开发资源。尤其在实时性、稳定性、开发速度和多处理器支持上面都是非常优秀的。

RTEMS 拥有三套 API 接口，其一是基于 RTEID (Real-Time Executive Interface Definition) 和 ORKID (Open Real-Time Kernel Interface Definition) 的接口，目前称为 RTEMS CLASS API 接口；同时也提供符合 POSIX 1003.1b 标准的 POSIX API 接口。这也就是说基于 POSIX API 的程序能运行在 RTEMS 上，这为应用程序的移植提供了很大的方便；还有一套接口是基于 ITRON 标准的，目前主要是支持 ITRON 3.0 API 标准，称为 IRON API 接口，不过此 API 的实现还不够完善。另外，RTEMS 上还移植了 FreeBSD 的 TCP/IP 协议栈，提供了完善的网络应用接口。

从体系结构上来看，RTEMS 是可配置的单体内核抢占式实时操作系统，它具有下面的优点：

- ⑩ 优秀的实时性能：支持硬实时和软实时；支持优先级继承协议和优先级天花板协议，防止优先级反转；支持速率单调实时调度 (rate mononitic scheduling, 简称 RMS)
- ⑩ 很好的稳定性：RTEMS 从设计初始就是为较少内存与较少 CPU 主频的嵌入式系统提供可靠的实时内核，该系统分为若干模块，在设计上充分考虑了对嵌入式系统的容错和高可靠的支持。
- ⑩ 支持多种 CPU：无论是 ARM, MIPS, PowerPC, i386 还是 DSP, AVR, Zilog, 都可以找到对应的板级支持包 (BSP)。
- ⑩ 高度可剪裁：内核在编译时可进行配置，可生成目标代码最小只有30KB。
- ⑩ 占用系统资源小，在32位系统中最小的内核只有30KB 左右。
- ⑩ 支持多处理器：RTEMS 中支持协作关系的多个处理器，不同于 SMP 架构。
- ⑩ 移植应用方便：提供了 POSIX API，这样 Linux/UNIX 下的程序可以方便移植
- ⑩ 完善的 TCP/IP 支持：提供完整的 BSD 的 TCP/IP 协议栈以及 FTP、WebServer、NFS 等服务
- ⑩ 先进的内核设计：使用面向对象思想设计，可以大大缩短开发周期。核心代码主要使用 C 由于编写，部分使用了 C++ 语言，可移植性好。支持 ISO/ANSI C 库，支持 ISO/ANSI C++ 库以及 STL 库。
- ⑩ 图形用户界面 (GUI)：目前在 RTEMS 上移植了 Microwindows/Nano-X。
- ⑩ 文件系统：目前支持 FAT 文件系统和内存文件系统 IMFS 等。
- ⑩ 多种调试模式：支持包括基于 GDB 的串口调试/以太网调试，基于仿真器的调试，基于模拟器 QEMU/SkyEye 的调试。

RTEMS 的版本

目前，RTEMS 的最新的稳定版是4.7.1，最新开发版是4.7.99.2，下一个稳定版发布将使用4.8的版本号。可以从其官方网站 (<http://www.rtems.org>) 直接免费获取最新的版本以及各历史版本。

RTEMS 的发展历史

- ❧ 1988年，美国军方提出需要一个无版税的开放源码的实时解决方案。
- ❧ 1990年，RTEMS 加入了对 Intel i386的支持，同年 RTEMS 有了第一个来自军方的用户——MICOM。
- ❧ 1991年，1.30版开始支持异构的多处理器平台，而1.31版加入了对 Intel i960的支持。
- ❧ 1992年，2.01版开始支持 GNU 工具，RTEMS 也开始有了第一个非军方用户——SSCL。
- ❧ 1993年，3.0版 RTEMS 加入了利用了 C 语言和 Ada 语言的应用。
- ❧ 1994年，3.1版本中 Ada 的应用开始支持多处理器平台。
- ❧ 1995年，RTEMS 加入了对 POSIX 1003.1b 基本支持，整个项目开始采用 GNU CVS 来做源代码的版本控制。从3.0到3.5版本，陆续加入了对一系列平台的支持，包括 PowerPC、SPARC、HP PA-RISC 等等。
- ❧ 1996年，RTEMS 加入了对 AMD A29K 和 MIPS 的支持，发布了最后一个军方版3.6，其中第一次加入了可用的 POSIX 1003.1b 支持。
- ❧ 1997年，RTEMS 的开发和应用开始遵循 FSF GPL 协议，同时 RTEMS 开始支持 TCP/IP 协议，并且建

立了自己的邮件列表。

- ✎ 1998年, OAR 公司开始接管 RTEMS 的开发, 新版4.0.0中支持稳定的 POSIX 标准, 而且加入了 FreeBSD TCP/IP 协议栈。
- ✎ 1999年, RTEMS 中加入了 ITRON 3.0的支持, 并且加入了基本的 IMFS 文件系统支持, 同时发布了 Linux 平台的 C/C++/Java 开发工具。
- ✎ 2000年, 新增了对 ARM、日立 H8、德州仪器 C3x/C4x 的支持, 开发平台也扩展到 Solaris、Cygwin、FreeBSD。
- ✎ 2001年, 修改的 MIPS 接口让 RTEMS 可以支持更多类型的 MIPS 芯片, 在这一年 RTEMS 管理委员会成立, 其成员都是对 RTEMS 的发展有过重要贡献的人。
- ✎ 2002年, RTEMS 加入了对 DOS (FAT12, FAT16, FAT32) 文件系统的支持, Till Straumann 发布了一份 RTEMS、RTLinux 和 VxWorks 的关于延时的测试报告, 报告显示出了 RTEMS 的优良性能。
- ✎ 2003年, RTEMS 得很多扩展功能进行了重大升级, 包括 GoAhead Webserver、ncurses 等等。
- ✎ 2004年, RTEMS 的 Wiki 建立, 这一年 RTEMS 也加入了对更多平台的支持。
- ✎ 2005年, RTEMS 完成了 PCI 总线统一标准 API 接口和 IRQ 统一标准 API 接口。
- ✎ 2006年, RTEMS 继续添加对多平台的支持, 同时 RTEMS 成功的应用在火星探测器和解密阿基米德的手稿中。
- ✎ 2007年, RTEMS 加入了对纳秒级时钟的支持, 4.7系列版本发布, RTEMS 代码的规模有显著的减小。

RTEMS 开发者社区

RTEMS 是由 OAR 公司为美国军方研制的, 所以 RTEMS 项目最早的相关维护和技术支持都是由 OAR 公司完成的, 随着 RTEMS 应用的逐渐广泛, RTEMS 社区也逐渐扩大。目前可以从 RTEMS 的官方网站获得及时而有效的技术支持。

RTEMS 的官方网站是 <http://www.rtems.com>, 中文社区为 <http://www.rtems.net>。

对于 RTEMS 的版权问题, 由于 RTEMS 包括: 核心操作系统, 应用软件, 协议等多个部分, RTEMS 中的一部分是 OAR 公司研发的, 也有一部分是移植其他开发人员的开源代码, 所以版权上, 不能笼统说是 GNU General Public License (简称 GPL) 或者 Berkeley Software Distribution License(简称 BSDL)。

它的版权包括下面几个部分:

- ✎ 核心代码 (包括操作系统, BSP) 等, 符合 RTEMS 版权许可, 该许可和 GPL 比较接近, 但是更适合嵌入式系统的商用应用, 因为 RTEMS 协议中也说了, 代码以二进制方式发布的时候, 如果在其中增加了自己的东西, 也可以不按照 GPL 发布, 也就是说, 可以不公开代码, 这点补充对于保密性质的软件来说, 是很有用的。
- ✎ TCP/IP 协议, 符合 BSD License (BSD 是比 GPL 更为开放的 License)
- ✎ RPC/XDR, 符合 SUN RPC 许可, 和 GPL 比较接近, 许可明确指出, 这些代码 SUN 公司不再对这些代码负有商业责任。
- ✎ WebServer 由 Go Ahead Software 软件公司开发, 协议规范参考代码中的 license.txt。其中要求在使用 Go Ahead WebServer 的时候, 网页上面必须要有 Go Ahead WebServer 字样。
- ✎ 需要指出, 上面所有 Licence 的共同特点就是可以免费使用, 但是修改原始代码的时候, 必须尊重原始作者。不要把代码里面的所有作者注释为你自己的名字, 由于是免费发送, 原始作者/公司对代码中可能存在的问题并不负有商业责任。

RTEMS 项目目前拥有两个可用的可支持搜索功能的邮件列表。

RTEMS Users 是主要的讨论区。

搜索页面: <http://www.rtems.com/ml/rtems-users/index.html>

索引界面: <http://rtems.rtems.org/pipermail/rtems-users>
申请界面: <http://www.rtems.com/mailman/listinfo/rtems-users>

RTEMS Announce 是用来发布即将发布的版本的信息以及其他重要事件的平台, 是一个只读的邮件列表。

搜索页面: <http://www.rtems.com/ml/rtems-announce/index.html>
索引界面: <http://rtems.rtems.org/pipermail/rtems-announce>
申请界面: <http://www.rtems.com/mailman/listinfo/rtems-announce>

小节

第二章 基本开发和调试过程

概述

本章主要介绍如何开发和调试基于 RTEMS 的嵌入式实时应用, 这主要是让读者在接触 RTEMS 的设计原理前, 能对系统有一个感性的认识, 知道如何基于 RTEMS 进行软件开发和调试。这也有助于读者通过运行调试 RTEMS 来分析 RTEMS 的内部实现。本章涉及的主要内容包括 RTEMS 源码的获取, 源码树结构介绍, 开发环境的建立以及如何运行 RTEMS 的测试用例、开发应用程序和调试应用程序。

RTEMS 源码的获取

RTEMS 的源码可以从它的官方网站上获取, 截至到2007年9月, RTEMS 最新的开发版本是4.7.99.2, 链接地址为 <ftp://ftp.rtems.com/pub/rtems/4.7.99.2/rtems-4.7.99.2.tar.bz2>。下载源码压缩包, 将其解压缩:

```
tar xjf rtems-4.7.99.2.tar.bz2
```

解压后的源代码位于 rtems-4.7.99.2 目录下, 这里把 RTEMS 源码的根目录记为 RTEMS_ROOT。

RTEMS 的源码树结构

首先, 我们看看 RTEMS_ROOT 目录下的一级子目录的大致内容:

`${RTEMS_ROOT}/aclocal/`

该目录存放了 autoconf 使用的 M4 宏。autoconf 解释 configure.ac 文件, 这些宏在定制、剪裁 RTEMS、为不同硬件体系产生不同目标文件这一系列过程中起着重要作用。

`${RTEMS_ROOT}/automake/`

该目录包含的*.am 文件是供 automake 使用的, 其他目录中的 Makefile.am 会包含这些文件。

`${RTEMS_ROOT}/c/`

该目录存放的文件是针对不同的主板、CPU 和芯片的代码, 这个目录保证了 RTEMS 的可移植性。

`${RTEMS_ROOT}/cpukit/`

该目录是 RTEMS 的库文件(例如 TCP/IP 协议栈 libnetworking)以及硬件无关的操作系统核心源代码, 该目录将在后面详细说明。

`${RTEMS_ROOT}/doc/`

存放 TeXinfo 和 HTML 格式的文档

`${RTEMS_ROOT}/make/`

Makefile 的配置文件, 例如 make/custom 子目录中每个 .cfg 文件都对应了一个 BSP 的编译器配置选项, 包括处理器模式、编译选项等。

`${RTEMS_ROOT}/tools/`

该目录包含 RTEMS 专用的一些工具。这些工具有目标硬件相关的，也有用来升级 RTEMS 源代码的。

`${RTEMS_ROOT}/testsuites/`

该目录包含了 RTEMS 自带的测试用例。

在上述目录中，`${RTEMS_ROOT}/cpukit/`和`${RTEMS_ROOT}/c/`目录中包括了 RTEMS 大部分代码，其中`${RTEMS_ROOT}/cpukit/`下存放了 RTEMS 的主要内核代码，下面分别对这两个目录做进一步介绍：

`${RTEMS_ROOT}/cpukit/aclocal/`

该目录包含了配置目标代码的并且生成 Makefile 文件的 M4宏脚本。

`${RTEMS_ROOT}/cpukit/ada/`

Ada API 的实现代码。

`${RTEMS_ROOT}/cpukit/include/`

RTEMS 内核代码的头文件。

`${RTEMS_ROOT}/cpukit/itron/`

ITRON API 的实现代码。

`${RTEMS_ROOT}/cpukit/libblock/`

使用块设备（硬盘、CD_ROM 等）所需的库文件实现代码。

`${RTEMS_ROOT}/cpukit/libcsupport/`

线程安全的 C 语言库，同时还提供了类 UNIX 系统调用（例如 open, chdir 等）的实现。

`${RTEMS_ROOT}/cpukit/libfs/`

文件系统的实现代码，包括 IMFS、miniIMFS、FAT 等。

`${RTEMS_ROOT}/cpukit/libnetworking/`

移植到 RTEMS 上的 BSD TCP/IP 代码的实现。

`${RTEMS_ROOT}/cpukit/librpc/`

该目录包含移植到 RTEMS 的 FreeBSD RPC/ XDR 源代码。

`${RTEMS_ROOT}/cpukit/posix/`

POSIX API 的 RTEMS 实现。

`${RTEMS_ROOT}/cpukit/rtems/`

RTEMS CLASSIC API 的实现。

`${RTEMS_ROOT}/cpukit/sapi/`

RTEMS 系统服务实现代码。

`${RTEMS_ROOT}/cpukit/score/`

RTEMS 的核心代码，各种 API 都是基于这层代码实现的。

`${RTEMS_ROOT}/cpukit/wrapup/`

将用户需要的库文件打包成单一的 RTEMS 库 `librtemscpu.a`。该库包含所有的目标嵌入式处理器模块与 BSP。

源码树中的`${RTEMS_ROOT}/c/src/`目录包含了支持各种 CPU 和板支持包 BSP 的代码

`${RTEMS_ROOT}/c/src/ada-tests/`

Ada 语言的测试用例

`${RTEMS_ROOT}/c/src/lib/libbsp`

包含了板级支持包（BSP）的源码

`${RTEMS_ROOT}/c/src/lib/libcpu`

包含了各种类型 CPU 相关的源码

`${RTEMS_ROOT}/c/src/libchip/`

包含了各处外设芯片的驱动源码

`${RTEMS_ROOT}/c/src/libnetworking/`

包含了 RTEMS 的网络部分源码，包括它的 telnetd, httpd 和 ftpd 服务

`${RTEMS_ROOT}/c/src/librdbg/`

包含了基于网络的远程调试栈

`${RTEMS_ROOT}/c/src/librtems++/`

包含了 CLASSIC API 的 C++实现

建立开发环境

建立工作目录结构

在使用 RTEMS 的过程中，需要一个综合的、容易使用的目录结构来组织各种软件包和文档。建议的工作目录组织如下：

❏ `rtems-4.7.99.2`: RTEMS 的内核代码

❏ `doc`: 将会用到的所有文档，包括 rtems 官方的帮助文档

❏ `tools`: 备份开发工具的源码包或者 RPM 包

❏ `build`: 用于编译内核和编译器的目录

❏ `examples-4.7.99.2`: RTEMS 的例子

❏ `test`: 用于测试的目录

上面的目录可存放于 `$HOME/rtems` 目录中，把这个目录记为 `PROJECT_ROOT`。`$HOME` 代表登录用户所在的缺省用户目录。可通过如下命令来声明宏 `PROJECT_ROOT`：

```
export PROJECT_ROOT=$HOME/rtems
```

使用 RPM 包安装

如果你使用的 Linux 发行版是 Fedora、RHEL 或者 Suse 都可以下载相应的 RPM 包直接安装，比用源码包编译安装更加方便！以 fedora 为例，首先到 <http://www.rtems.com/ftp/pub/rtems/linux/4.8/fedora/>

根据宿主机和目标机的硬件平台选择相应的 RPM 包（到2007年8月为止的最新版本，将来如有更新的版本，具体的处理方法是一样的，关键是要把各种软件包的对应的最新版本找对。）。如果宿主机是 X86-32, 且采用 Fedora 7 Linux Distribution, 目标机是基于 X86的平台，则选择以下 RPM 包下载：

`rtems-4.8-autoconf-2.60-5.fc7.noarch.rpm`

`rtems-4.8-automake-1.9.6-5.fc7.noarch.rpm`

`rtems-4.8-binutils-common-2.17.90-2.fc7.i386.rpm`

`rtems-4.8-gcc-common-4.2.1-23.fc7.i386.rpm`

`rtems-4.8-gdb-common-6.6-9.fc7.i386.rpm`


```
rtems-4.8-i386-rtems4.8-binutils-2.17.90-2.fc7.i386.rpm
rtems-4.8-i386-rtems4.8-gcc-4.2.1-23.fc7.i386.rpm
rtems-4.8-i386-rtems4.8-gcc-c++-4.2.1-23.fc7.i386.rpm
rtems-4.8-i386-rtems4.8-gdb-6.6-9.fc7.i386.rpm
rtems-4.8-i386-rtems4.8-newlib-1.15.0-23.fc7.i386.rpm
```

然后安装以上 rpm 包，注意，具体体系结构的 rpm 包依赖于“common”的 rpm 包。最后不要忘记执行如下命令或者把下面的行写入 ~/.bashrc：

```
export PATH=/opt/rtems-4.8/bin:$PATH
```

以确保安装好的编译工具在命令路径中。

使用 YUM 安装

YUM 是 RPM 包的自动安装、更新以及卸载的管理系统。RTEMS 的官方站点提供了 RTEMS 开发工具的 YUM 安装源，这样我们就可以利用 YUM 很方便地构建 RTEMS 的开发环境。

假定我们的宿主机上运行的是 Redhat/Fedora 系列的操作系统，那么首先在系统的 /etc/yum.repos.d 目录下创建 rtems.repo 这个文件，文件的内容如下：

```
[rtems-4.8]
name=rtems
baseurl=http://www.rtems.com/ftp/pub/rtems/linux/4.8/fedora/7/i386/
enabled=1
gpgcheck=0
```

上面的 baseurl 可能需要根据自己的宿主系统做相应的修改，然后执行下面的命令：

```
yum list | grep rtems
```

这样就可以看到 YUM 源里的针对不同目标体系结构的开发工具的安装包，复制下需要安装的包名，然后进行安装：

```
yum install rtems-4.8-i386-rtems4.8-gcc
```

YUM 会自动解决依赖关系，上面的操作还会安装 gcc-common、binutils-common、i386-binutils、newlib-common、i386-newlib 这五个包。继续安装 gdb：

```
yum install rtems-4.8-i386-rtems4.8-gdb
```

这样，就和上面的采用 RPM 包手动安装一样，在 /opt/rtems-4.8 目录下就安装好了 RTEMS 的开发环境，然后也需要把 /opt/rtems-4.8/bin 加入命令路径。

注意，采用 RPM 包或者 YUM 安装的交叉编译器的前缀是 i386-rtems4.8-，而在编译安装 RTEMS 的时候，如果指定 configure 的选项 --target=i386-rtems，那么编译的时候系统会查找编译器 i386-rtems-gcc，这样就会因为找不到编译器而无法编译。这时可以为 /opt/rtems-4.8/bin 下的程序做一个相应的链接，把前缀改为 i386-rtems，或者在 configure 时指定 --target=i386-rtems4.8。

使用 DEB 格式安装

另外，如果宿主系统是 Debian/Ubuntu 等基于 DEB 包安装系统的 Linux distribution，它们不支持基于 RPM 方式的安装。我们可以采用一种变通的办法，即用 alien 软件工具把 RPM 包转换成 DEB 包，然后再用 dpkg 安装转换后的 DEB 包。鲍 APT（Advanced Packaging Tool）安装，具体方法（要用 root 用户权限）如下：

1. 参考上节的描述，下载基于 Fedora7 的开发工具的 RPM 包到本地某安装目录下；
2. 在此安装目录下执行如下命令完成格式转换：

```
alien *.rpm
```

3. 在此安装目录下执行如下命令完成对 DEB 格式软件包的安装:

```
dpkg -i *.deb
```

注意: 在 Ubuntu7.04 中, 采用上述方法安装的 i386-rtems4.8-gdb 由于缺少与 Fedora7 相关的共享链接库而不能执行。这需要采用下节描述的源码安装的方式安装 i386-rtems4.8-gdb。

使用 MinGW 工具在 Windows 下安装

从 RTEMS4.7.99.1 开发版开始, RTEMS 也开始支持 Window 下图形界面的安装包。该安装包需要和 MinGW 工具配合使用。这里需要注意, 在 4.6 版本的时候, RTEMS 提供 Cygwin 的安装包, 但是目前 RTEMS 停止了对 Cygwin 工具链的支持, 基于 MinGW 的安装包与 Cygwin 系统并不兼容。

安装 Windows 工具链前需要安装设置好 MinGW, 并且安装 MYSYS 和 MY DTK, 然后在 <http://www.rtems.org/ftp/pub/rtems/windows/4.8/build-15/rtems4.8-tools-15.exe> 下载 Window 的工具链安装包, 在 Windows 下就可以在图形界面下安装设置 RTEMS 工具链了。

安装好好, 安装目录下面有几个 .bat 脚本, 其中 **rtems-env.bat** 用于配置工具链的环境变量, **rtems.bat** 用于打开一个 Windows command 窗口并且设置好环境变量, 最后一个是 **sh-run.bat**, 这个脚本可以让开发人员在编辑器 (例如 UltraEdit, Eclips, PSPad 等) 中运行编译指令。

使用源码包安装

这里介绍的是基于 2007 年 8 月为止的最新源码版本。将来如有更新的版本, 具体的处理方法是一样的, 关键是需要下载各种最新版本的源码包和对应的最新 diff 或 patch。

执行需要安装的交叉编译工具有: binutils, gcc, gdb, newlib 等。首先到 <ftp://ftp.rtems.com/pub/rtems/SOURCES> 下载以下源码包:

- autoconf-2.60.tar.bz2 : 一个产生可以自动配置源代码包, 生成 shell 脚本的工具
- automake-1.9.6.tar.bz2 : 一个产生可以自动生成 Makefile 脚本的工具
- binutils-2.17.tar.bz2 : 一组开发工具, 包括连接器, 汇编器和其他用于目标文件和档案的工具
- 🔗 [binutils-2.17-rtems4.8-20061021.diff](#): 针对 rtems 的 binutils 补丁
- 🔗 gcc-core-4.2.1.tar.bz2: C/C++ 编译器 gcc
- 🔗 [gcc-core-4.2.1-rtems4.8-20070804.diff](#): 针对 rtems 的 gcc 补丁
- 🔗 gdb-6.6.tar.bz2: GNU DEBUG 调试器
- 🔗 [gdb-6.6-rtems4.8-20070306.diff](#): 针对 rtems 的 gdb 补丁
- newlib-1.15.0.tar.gz: 支持多线程和可重入的面向嵌入式系统的 C 运行库
- 🔗 [newlib-1.15.0-rtems4.8-20070804.diff](#): 针对 rtems 的 newlib 补丁

然后解压缩、打补丁并编译安装, 具体的操作如下 (假定当前用户具有 root 权限, 上述软件包下载到 \$PROJECT_ROOT/tools 目录下, 要安装到 /opt/rtems 目录下):

```
cd $PROJECT_ROOT
mkdir build && cd build
```

编译、安装 autoconf 和 automake:

```
cd $PROJECT_ROOT/tools/automake-1.9.6/
./configure --prefix=/opt/rtems-4.8
make
make install
cd .. /autoconf-2.60/
./configure --prefix=/opt/rtems-4.8
make
make install
```

解压缩 binutils、gcc、gdb 和 newlib 源码包

```
cd $PROJECT_ROOT/tools/
tar jxf binutils-2.17.tar.bz2
tar zxf newlib-1.14.0.tar.gz
tar jxf gcc-core-4.2.1.tar.bz2
tar jxf gdb-6.6.tar.bz2
```

打补丁

```
cd newlib-1.14.0/
cat ../newlib-1.15.0-rtems4.8-20070804.diff |patch -p1
cd ../binutils-2.17/
cat ../binutils-2.17-rtems4.8-20061021.diff |patch -p1
cd ../gcc-4.2.1/
cat ../gcc-core-4.2.1-rtems4.8-20070804.diff |patch -p1
ln -s ../newlib-1.15.0/newlib
cd ../gdb-6.6/
cat ../gdb-6.6-rtems4.8-20070306.diff |patch -p1
```

编译和安装 binutils、gcc 和 gdb

```
cd $PROJECT_ROOT/build
mkdir build-binutils
mkdir build-gcc
mkdir build-gdb
cd build-binutils/
../../tools/binutils-2.17/configure --target=i386-rtems4.8 --program-prefix=i386-rtems4.8-
--prefix=/opt/rtems-4.8
```

```

make
make install
export PATH=/opt/rtems/bin:$PATH
cd ../build-gcc/
../../tools/gcc-4.2.1/configure --target=i386-rtems4.8 --with-gnu-as --with-gnu-ld --with-newlib --verbose --enable-threads --enable-languages="c" --program-prefix=i386-rtems4.8- --prefix=/opt/rtems-4.8
make
make install
cd ../build-gdb/
../../tools/gdb-6.6/configure --target=i386-rtems --program-prefix=i386-rtems4.8- --prefix=/opt/rtems-4.8
make
make install

```

需要注意：把安装好的软件加入到命令路径，如果你使用的是 bash 作为 shell，则建议写到 ~/.bashrc 中，否则每次打开新的 shell 还需要重新声明。这样，RTMES 的开发环境就已经安装完成。另外，在 build 目录中建立的各临时目录以及 tools 中的解压缩的源码可以删去以节省空间。可以在命令行执行下面的命令“ i386-rtems-gcc -v”，以确认：

```

$i386-rtems-gcc -v
使用内建 specs。
目标：i386-rtems
配置为：../../gcc-4.2.1/configure --target=i386-rtems4.8 --with-gnu-as --with-gnu-ld --with-newlib --verbose --enable-threads --enable-languages=c - program-prefix=i386-rtems4.8- --prefix=/opt/rtems-4.8
线程模型：rtems
gcc 版本 4.2.1

```

编译安装 RTEMS

在编译前要确认设定了 RTEMS 编译器的执行路径，即确认执行了如下 shell 命令：

```
export PATH=/opt/rtems-4.8/bin:$PATH
```

以确保安装好的编译工具在命令路径中。

通过执行如下命令可以完成编译安装 RTEMS：

```

cd $PROJECT_ROOT/build
tar jxf ../rtems-4.7.99.2.tar.bz2
mkdir build-rtems
cd build-rtems
../rtems-4.7.99.2/configure --target=i386-rtems4.8 --enable-posix --enable-networking --

```

```
enable-rdbg --enable-tests --enable-rtemsbsp="pc386" --prefix=/opt/rtems/rtems-4.8
make
make install
```

这样 RTEMS 就被安装到关于 configure 指定的参数的具体含义可以运行下面的命令来查看：

```
../rtems-4.7.99.2/configure --help
```

可以根据需要来指定相应的选项。

注意：

1. 这里的 build-rtems 不要删除，在以后开发应用程序时对我们还有帮助。
2. 如果执行 configure 出现错误需要重新执行 configure 时，应该先把 build-rtems 目录下的内容清除掉。
3. 如果你安装的用于 rtems 的 gcc 名称是 i386-rtems-gcc，则在执行 configure 时把 “--target=i386-rtems4.8” 改为 “--target=i386-rtems”。

运行测试用例

源码包中的测试用例

实际上在上面编译完的 RTEMS 中，就已经编译了 RTEMS 自带的一些测试用例。在 /opt/rtems/rtems-4.8/i386-rtems/c/pc386/testsuites 的各个子目录中分别存放着 CLASSIC、POSIX、ITRON 三种 API 的测试用例，还有一些关于时间和驱动力的测试。这里以 samples 目录下的 hello 为例：

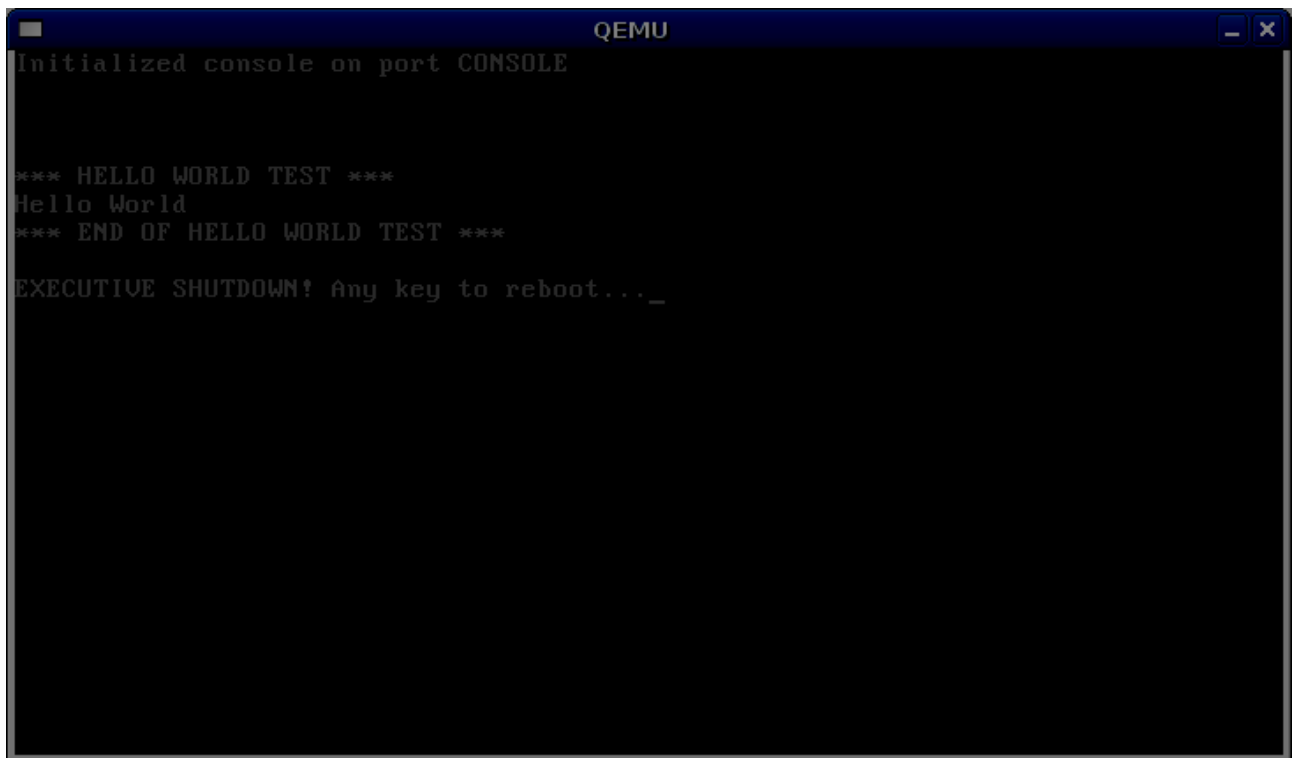
```
cd
build_image.sh fd.img /opt/rtems/rtems-4.8/i386-
rtems/c/pc386/testsuites/samples/hello/hello.exe
```

第一个命令 “cd” 将把当前目录设置为用户的 home 目录。build_image.sh 是一个生成软盘镜像的脚本文件，不属于 rtems 软件包的内容，此文件在本书提供的光盘中，建议把这个文件放在 /opt/rtems/bin 目录下。build_image.sh 需要用到 mformat 等命令，而这些命令属于 mtools 软件包，在不同的 Linux 发行版都有此软件包。如果 build_image.sh 执行错误或无法执行，一般都是由于缺少 mtools 软件包而造成的。

我们可以通过 PC 机来运行 RTEMS 操作系统。不过为了学习和调试的方便，我们可以用虚拟机软件来模拟硬件，让 RTEMS 操作系统在虚拟机上运行。目前模拟基于 X86 架构的虚拟机软件有很多，如 VMware、Bochs、VirtualBox、QEMU 等，模拟基于 ARM CPU 的嵌入式开发板的有 SkyEye 等。在本章中，我们将主要使用 QEMU 作为我们的 X86 模拟器。假定在你的系统中已经安装了 QEMU。我们然后就可以使用 QEMU 来运行这个 RTEMS 镜像：

```
qemu -fda fd.img
```

这样就可以在屏幕上看到 “Hello, world!” 了。下面是屏幕的截图：



运行 examples 中的例子

把下载的 examples-4.7.99.2.tar.bz2 解压缩，然后在 examples-4.7.99.2 里可以看到一些例子，还是以“hello world”为例：

```
cd $PROJECT_ROOT/examples-4.7.99.2/hello_world_c
export RTEMS_MAKEFILE_PATH="/opt/rtems/rtems-4.8/i386-rtems/pc386"
make
```

编译生成的可执行文件放在 o-optimize 目录里，采用和上面同样的方法就可以运行了。

开发自己的应用程序

在开发自己的应用程序的时候，可以在 RTEMS 自带的测试用例的基础上进行修改，这样可以省去自己编写 Makefile 的麻烦。所以在开发应用程序之前，最好是先找到一个在内容上或者是文件结构上类似的例子。

利用 build-rtems 目录

这种方法是利用前面编译 RTEMS 时在 build-rtems 目录下生成的内容。假设找到的参考用例是“hello”，如果不需要增加文件，那么只需要修改 \$RTEMS_ROOT/testsuites/samples/hello 中的源文件，然后执行如下命令：

```
cd $PROJECT_ROOT/build/build-rtems/i386-rtems/c/pc386/testsuites/samples/hello
make
```

也就是进入到编译 RTEMS 的目录中“hello”所在的目录，然后再执行 make，这时候就会编译修改后的源文件，生成新的可执行程序。

如果需要增加或者删除文件，那么需要修改“hello”源码目录中的 Makefile.am 文件，主要是修改：

```
hello_exe_SOURCES = hello.c system.h
```

这一行，在这增加自己的源文件或者头文件。因为 RTEMS 采用 autotools 作为编译工具，在修改完 Makefile.am 后还要更新 Makefile.in，这样在编译目录中才能生成新的 Makefile。具体的操作如下：

```
cd $RTEMS_ROOT/testsuites/samples/  
$RTEMS_ROOT/bootstrap
```

这样就更新了 Makefile.in，然后再进入 hello 的编译目录重新编译就可以生成新的可执行程序了。

下面通过一个实例来进一步说明这个过程。本来 init.c 文件本来就是打印一个 hello world，现在想让它可以计算两个数的和。为了说明如何增加一个文件，我们把 sum 这个函数写在这个目录下的 add.c 文件里，这个文件里的代码很简单：

```
int sum(int a, int b)  
{  
    return a+b;  
}
```

然后在原来的 system.h 中增加一个 sum 函数的声明，以便 init.c 中可以调用它。声明如下：

```
int sum(int a, int b);
```

因为 init.c 原来就包含了 system.h，所以就可以在这个文件里调用 sum 了，在函数 Init 中增加一行代码：

```
printf("the sum of 1 and 2 is %d\n", sum(1, 2));
```

接着就应该修改 Makefile.am，把第8行改为：

```
hello_exe_SOURCES = init.c add.c system.h
```

然后执行如下命令：

```
cd .. && $RTEMS_ROOT/bootstrap
```

来更新 Makefile.in。最后进入 hello 的编译目录重新编译、运行新的可执行程序就可以发现 sum 函数已经添加成功！

利用 examples 目录

利用 examples 目录会更加简单一些，首先要正确设置 RTEMS_MAKEFILE_PATH 环境变量，然后就可以修改源文件、编译了。如果需要增加新的文件，需要修改 Makefile 中的行：

```
CSRCS = test.c
```

在这一行中增加自己的源文件，然后再重新编译。

RTEMS 的调试

使能 RTEMS 的内核调试

在编译 RTEMS 时，如果在 configure 的选项中增加—enable-rtems-debug 选项，那么会定义 RTEMS_DEBUG 宏，这样打开内核中的 bug 检查开关，比如在 \$RTEMS_ROOT/cpukit/score/include/rtems/score/heap.h 中，有如下代码：

```
#if defined(RTEMS_DEBUG)
#define RTEMS_HEAP_DEBUG
#endif
```

而宏 RTEMS_HEAP_DEBUG 又会使函数 _Heap_Walk 具有检验内存并且打印相关信息的功能，如果没有定义这个宏，_Heap_Walk 就是空函数。

使用 QEMU 和 GDB 调试

以 /i386-rtems/c/pc386/testsuites/samples/hello 目录下的 hello.exe 为例，在此目录下，有 hello.nxe 这是带调试信息的 ELF 执行文件格式的代码，hello.exe 是可以被 grub 加载并执行的 elf 格式的代码（没有调试信息）。注意，在 \$RTEMS_ROOT/make/custom/pc386.cfg 文件中定义的编译优化选项已经打开了调试开关：

```
CFLAGS_OPTIMIZE_V = -O2 -g
```

所以，直接编译生成的 hello.elf 文件就带有调试信息，当然也可以在 hello 的编译目录中执行：

```
make clean
make CFLAGS="" -O0 -g
```

也就是不让编译器优化代码，这样 gdb 对源代码和机器码的映射关系定位得更加准确。

首先通过 build_image.sh 得到包含 grub 和 hello.exe 的软盘镜像 fd.img：

```
cd $PROJECT_ROOT/test
build_image.sh fd.img fd.img ../build-
rtems/i386rtems/c/pc386/testsuites/samples/hello/hello.exe
```

然后就可以用 QEMU 调试了。启动两个 shell 窗口，在其中一个窗口执行如下命令：

```
qemu -S -s -fda fd.img
```

这样，qemu 在完成初始化后，就启动了一个 gdbserver，并等待 gdb 的远程链接。在另外一个 shell 窗口执行如下命令，注意 (gdb) 后面是 gdb 中调试的命令：

```
cd $PROJECT_ROOT/test
i386-rtems-gdb ../build-rtems/i386-rtems/c/pc386/testsuites/samples/hello/hello.nxe
(gdb) target remote :1234
```



```

Remote debugging using :1234
0x0000ffff in ?? ()
(gdb) break Init
Breakpoint 1 at 0x1000ec: file ./c/src/../../testsuites/samples/hello /init.c, line 31.
(gdb) continue
...

```

RTEMS 可靠性和口控口助手段

在 RTEMS 操作系统中，由于采用了扁平（flat）内存管理机制，导致无法在硬件上利用当前 CPU 具有的 MMU 功能防止内存方面的非法访问，使得应用开发人员对堆栈溢出、运行时内存分配与应用的内存使用要求不一致等问题很难解决。但通过软件技术，在一定程度上还是能够解决上述问题的。而且 RTEMS 操作系统还提高了对内部系统功能对象的查看机制，这样可以在 RTEMS 操作系统运行的时候，检查当前各个内部系统功能对象的状态，这对于发现 RTEMS 应用和操作系统中隐含的问题很有帮助。

堆口溢出

使用方法

为了解决堆栈溢出问题。RTEMS 在内核中实现了堆栈溢出检查机制。在 RTEMS 源码中的 \${RTEMS_ROOT}/testsuites /libtests/stackchk 目录中包含了堆栈溢出检查的测试用例。此例子主要是让一个任务在过了15秒后，执行了一个破坏堆栈的函数blow_stack。由于此目录下的system.h文件中，包含了如下语句：

```
#define STACK_CHECKER_ON
```

使得 RTEMS 内核包含了堆栈溢出检查代码模块，从而可以在执行了一个破坏堆栈后，发现堆栈被破坏的情况，输出报错信息。此例子可能的运行结果如下所示：

```

*** TEST STACK CHECKER ***
TA1 - rtems_clock_get - 09:00:00 12/31/1988
TA2 - rtems_clock_get - 09:00:00 12/31/1988
TA3 - rtems_clock_get - 09:00:00 12/31/1988
TA1 - rtems_clock_get - 09:00:05 12/31/1988
TA1 - rtems_clock_get - 09:00:10 12/31/1988
TA2 - rtems_clock_get - 09:00:10 12/31/1988
TA1 - rtems_clock_get - 09:00:15 12/31/1988
---> error indication

```

实现方法

RTEMS 堆栈溢出检查的实现位于 \${RTEMS_ROOT}/cpukit/libmisc/stackchk/目录下。其主要工作是在上下文切换中检查堆栈溢出，以及提供了一个基于模式匹配的任务堆栈使用情况检测。首先建立了一个 RTEMS 扩展表，如下所示：

```

rtems_extensions_table rtems_stack_checker_extension_table = {
  rtems_stack_checker_create_extension, /* rtems_task_create */
  0, /* rtems_task_start */
  0, /* rtems_task_restart */
  0, /* rtems_task_delete */
  rtems_stack_checker_switch_extension, /* task_switch */
  rtems_stack_checker_begin_extension, /* task_begin */ /* task_exitted */
#ifdef DONT_USE_FATAL_EXTENSION
  0, /* fatal */

```

```

#else
    rtems_stack_checker_fatal_extension, /* fatal */
#endif
};

```

有关 RTEMS 扩展表的分析，可参考“线程管理与调度”一章的“任务扩展机制”一节。这样在任务创建（即执行 `rtems_task_create` 函数）的时候执行 `rtems_stack_checker_create_extension` 函数，此函数完成对任务的堆栈按照某种模式 A 进行填充。在任务启动（即执行 `rtems_task_begin` 函数）的时候，在堆栈的末端按照某种模式 B 填充16个字节。但每次执行任务切换（即执行 `task_switch` 函数）到改该任务的时候，会检查此任务堆栈的末端16个字节与16个字节的模式 B 是否一致，如果不一致，则表示堆栈溢出了。

性能□□与□□

实现代码在 `${RTEMS_ROOT}/cpukit/libmisc/capture` 目录。

性能监视模块尽可能的降低自身对系统的影响，在任务的创建、删除以及上下文切换中都尽量保证高性能。

检测模块为分层结构，最底层为信息捕获层，它的接口都定义在 `capture.h` 头文件中。

用户可以通过 RTEMS 主界面以命令交互的方式实现

命令包括：

```

copen    - 打开一个监测模块 Open the capture engine.
cclose   - 关闭一个监测模块 Close the capture engine.
cenable  - 使能一个监测模块 Enable the capture engine.
cdisable - 禁止一个监测模块 Disable the capture engine.
ctlist   - 列出监视模块可识别的任务 List the tasks known to the capture engine.
ctload   - 显示当前的任务运行情况 Display the current load (sort of top).
cwlist   - List the watch and trigger controls.
cwadd    - Add a watch.
cwdel    - Delete a watch.
cwctl    - Enable or disable a watch.
cwglob   - Enable or disable the global watch.
cwceil   - Set the watch ceiling.
cwfloor  - Set the watch floor.
ctrace   - Dump the trace records.
ctrig    - Define a trigger.

```

操作方法具体说明

命令：Open

使用方法： `copen [-i] size`

说明：

创建一个监视引擎。size 参数设定监视引擎的跟踪缓冲区大小。使用 `[-i]` 参数则在监视引擎创建

之后立即运行。每一条记录仅仅纪录一个事件。

命令: Close

使用方法: cclose

说明:

删除一个监视引擎并且释放所有占用的资源。

命令: Enable

使用方法: cenable

说明:

将一个已经创建的监视引擎转入运行态。

命令: Disable

使用方法: cdisable

说明:

阻塞一个运行态的监视引擎。Enable 和 Disable 操作，主要用来测试在有无监视引擎的不同情况下系统的负载情况。

命令: Task List

使用方法: ctlist

说明:

显示监视引擎可识别的任务列表，其中可能包括已删除的任务。

命令: Task Load

使用方法: ctload

说明:

依据占用 CPU 由高到底的顺序排列所有任务，列表将每5秒钟更新一次，用户输入回车键则终止显示。输出显示如下:

| | | | | | | | | | |
|----------------------|------|------|------|-------|---------|------|---------|------|------|
| Press ENTER to exit. | | | | | | | | | |
| PID | NAME | RPRI | CPRI | STATE | %CPU | %STK | FLGS | EXEC | TIME |
| 04010001 | IDLE | 255 | 255 | READY | 96.012% | 0% | a-----g | 1 | |
| 08010009 | CP1t | 1 | 1 | READY | 3.815% | 15% | a----- | 0 | |
| 08010003 | ntwk | 20 | 20 | Wevnt | 0.072% | 0% | at----g | 0 | |
| 08010004 | CSr0 | 20 | 20 | Wevnt | 0.041% | 0% | at----g | 0 | |
| 08010001 | main | 250 | 250 | DELAY | 0.041% | 0% | a-----g | 0 | |
| 08010008 | test | 100 | 100 | Wevnt | 0.000% | 20% | at-T--g | 0 | |

```
08010007 test 100 100 Wevnt 0.000% 0% at-T-+g 0
08010005 CSt0 20 20 Wevnt 0.000% 0% at----g 0
08010006 RMON 1 1 Wsem 0.000% 0% a----- 0
```

There are 7 flags and from left to right are:

- 1) 'a' the task is active, and 'd' the task has been deleted.
- 2) 't' the task has been traced.
- 3) 'F' the task has a from (TO_ANY) trigger.
- 4) 'T' the task has a to (FROM_ANY) trigger.
- 5) 'E' the task has an edge (FROM_TO) trigger.
- 6) '+' the task as a watch control attached, 'w' a watch is enabled.
- 7) 'g' the task is part of a global trigger.

🔗 %STK 是任务的堆栈使用情况，当前仅有任务自身创建的堆栈才可以被监视。

🔗 RPRI 是任务的实际优先级，在任务创建时设定的。

命令: Watch List

使用方法: cwlist

说明:

显示监视引擎所有的监视控制器，每一个被监视的任务都包含一个监视控制器。

命令: Watch Add

使用方法: cwadd [task name] [id]

说明:

为任务添加一个监视控制器。制定任务可以用任务名 (task name) 或者任务号 (id) 表示。要注意的是，如果使用任务名的方式，则所有使用该任务名的任务都将添加一个监视控制器，因为任务名不是惟一的；而任务号却是惟一的，如果只想给指定任务添加一个监视控制器，则使用任务号方式是适合的。但任务名方式还有一个优点，就是在任务创建而还未运行的情况下就可以添加。

命令: Watch Delete

使用方法: cwdel [task name] [id]

说明:

删除一个监视控制器。

命令: Watch Control

使用方法: cwctl [task name] [id] on/off

说明:

打开或关闭一个监视控制器，on/off 指定打开或者关闭。

命令: Global Watch

使用方法: cwglob on/off

说明:

打开或关闭一批监视控制器，如果任务的优先级在监控的范围之内的话。

命令: Watch Priority Ceiling

使用方法: cwceil priority

说明:

设定可以监控任务的优先级上限，优先级高于这个上限的任务将不在监视之列。这样保证了高优先级的系统任务或者驱动程序不在监视之列。

命令: Watch Priority Floor

使用方法: cwfloor priority

说明:

设定可以监控任务的优先级下限，优先级低于这个下限的任务将不在监视之列。这样保证了类似于 IDLE 之类的任务不在监视之列。

命令: Trace

使用方法: ctrace [-c] [-r records]

说明:

转储监视纪录。[-c]参数指定输出的变量以逗号间隔。[-r records]参数可以指定纪录的数量，避免进入死循环。

命令: Trigger

使用方法: ctrig type [from name] [from id] [to name] [to id]

说明:

设定一个任务切换的触发器。参数意义如下:

from : 指定切换前任务

to : 指定切换后任务

name : 任务名

id : 任务号

命令: Flush

使用方法: cflush [-n]

说明:

刷新监视纪录。

系□□□器

Windriver 公司在其 Tornado 集成开发环境中有一个功能强大的 SHELL 软件可供分析和调试基于 VxWork 操作系统的应用程序。其实在 RTEMS 操作系统中也有类似的一个 SHELL 软件，它作为一个任务在 RTEMS 中运行，任务的优先级为1，除非有其他优先级也为1的任务，否则这个监视器 SHELL 都能得到及时的响应。因为系统监视器也是以一个任务的形式在运行，所以在不需要的时候我们可以阻塞这个任务让 RTEMS 正常运行，在需要的时候再重新唤醒这个任务。

使用方法

在 RTEMS 源码中的\${RTEMS_ROOT}/testsuites /libtests/monitor 目录中包含了使用系统监视器的测试用例。由于系统监视器是作为一个任务存在的，且它的优先级很高，所以在配置的时候需要注意，大致使用方法包含配置部分，主要主要设置任务数和堆栈大小。如下面的代码片断：

```
.....
//六个用户任务+一个系统监视器任务
#define CONFIGURE_MAXIMUM_TASKS      7
.....
//降低 Init 任务的优先级为10
#define CONFIGURE_INIT_TASK_PRIORITY  10
#define CONFIGURE_INIT_TASK_INITIAL_MODES RTEMS_DEFAULT_MODES
.....
//设置六个用户任务+一个系统监视器任务的堆栈
#define CONFIGURE_EXTRA_TASK_STACKS \
(6 * (3 * RTEMS_MINIMUM_STACK_SIZE)) /* our tasks */ + \
(1 * RTEMS_MINIMUM_STACK_SIZE)      /* monitor tasks */
```

和初始化部分，如下面的代码片断：

```
.....
//创建并启动用户任务后
rtems_monitor_init( 0 );
//删除 Init 任务
status = rtems_task_delete( RTEMS_SELF );
.....
```

在 qemu 中运行测试用例，可以得到如下输出（粗斜体是用户敲的字符）：

```
*** MONITOR TASK TEST ***

Monitor ready, press enter to login.

rtems $ help
config  - Show the system configuration.
itask   - List init tasks for the system
mpci    - Show the MPCIE system configuration, if configured.
pause   - Monitor goes to "sleep" for specified ticks (default is 1).
         Monitor will resume at end of period or if explicitly
         awakened
         pause [ticks]
continue - Put the monitor to sleep waiting for an explicit wakeup from
         the program running.
go       - Alias for 'continue'
node     - Specify default node number for commands that take id's.
         node [ node number ]
```

```

symbol - Display value associated with specified symbol. Defaults to
        displaying all known symbols.
        symbol [ symbolname [symbolname ... ] ]
extension - Display information about specified extensions. Default is to
        display information about all extensions on this node.
        extension [id [id ...] ]
task - Display information about the specified tasks. Default is to
        display information about all tasks on this node.
        task [id [id ...] ]
queue - Display information about the specified message queues.
        Default is to display information about all queues on this
        node.
        queue [id [id ...] ]
object - Display information about specified RTEMS objects. Object id's
        must include 'type' information. (which may normally be
        defaulted)
        object [id [id ...] ]
driver - Display the RTEMS device driver table.
        driver [ major [ major ... ] ]
dname - Displays information about named drivers.
exit - Invoke 'rtems_fatal_error_occurred' with 'status' (default is
        RTEMS_SUCCESSFUL)
        exit [status]
fatal - 'exit' with fatal error; default error is RTEMS_TASK_EXITED
        fatal [status]
quit - Alias for 'exit'
help - Provide information about commands. Default is show basic
        command summary.
        help [ command [ command ] ]
rtems $ task
ID   NAME  PRIO  STAT  MODES  EVENTS  WAITID  WAITARG  NOTES
-----
0a010002 TA1    1  DELAY P:T:nA NONE
0a010003 TA2    1  DELAY P:T:nA NONE
0a010004 TA3    3  DELAY P:T:nA NONE
0a010005 TA4    4  DELAY P:T:nA NONE
0a010006 TA5    5  DELAY P:T:nA NONE
0a010007 RMON   1  READY P:T:nA NONE  1a010007 0x1

```

这个系统监视器比较简单，但是依然包含了基本的功能，支持的命令包括：

- help - 获得帮助信息
- pause - 暂停 Monitor 任务指定时间
- exit - 引发一个 Fatal RTEMS Error，退出
- symbol - 显示符号列表项
- continue - 阻塞 Monitor 任务等待唤醒，执行 RTEMS 其它任务
- config - 显示系统配置
- itask - 显示系统所有已经创建的任务
- mpci - 显示 MPCI 配置
- task - 显示任务的相关信息
- queue - 显示消息队列的相关信息

- extension - 显示 RTEMS 扩展
- driver - 显示设备驱动信息
- ❏

实现方法

系统监视器的实现代码在\${RTEMS_ROOT}/cpukit/libmisc/monitor/目录。

首先要进行 monitor 任务初始化，monitor 任务的函数定义如下：

```
void rtems_monitor_init(
    uint32_t  monitor_flags
)
```

输入参数 monitor_flags 指定系统监视器任务的启动模式，如果为0，表示正常启动； 如果为1（即 RTEMS_MONITOR_SUSPEND），表示启动此任务后直接阻塞，等待唤醒； 如果为2（即

RTEMS_MONITOR_GLOBAL），表示设定任务为全局模式。

函数执行流程如下：

1. 首先终止可能正在运行的 monitor 任务；
2. 创建新的 monitor 任务，任务优先级为1；
3. 加载符号列表 rtems_monitor_symbols_loadup；
4. 若任务为全局模式，则要启动一个 monitor_server 任务，由 rtems_monitor_server_init 创建；
5. 运行 monitor 任务，若包含 RTEMS_MONITOR_SUSPEND 标志则直接阻塞。

其它与 monitor 运行管理相关的函数还有：

❏ monitor 任务结束

```
void rtems_monitor_kill(void)
```

函数执行流程如下：

1. 调用 rtems_task_delete 终止并删除 Monitor 任务；
2. 若任务为全局模式，则终止并删除 Monitor_Server 任务。

❏ 阻塞 monitor 任务

```
rtems_status_code rtems_monitor_suspend(rtems_interval timeout)
```

❏ 唤醒 monitor 任务

```
void rtems_monitor_wakeup(void)
```

CEXP

CEXP 是一个由 stanford 大学开发的 C 语言表达式解释器的命令行版本工具，可以作为 RTEMS 的一个 SHELL 加载，能够访问程序模块的符号表，并且支持动态加载模块运行。具体信息可参见其主页

<http://www.slac.stanford.edu/~strauman/rtems/software.html>。下面我们将对其作简要的介绍。

在主页上提供了一个 RTEMS evaluation CD-ROM 镜像的下载，其中包括一个已经编译好的 RTEMS/GeSys 系统，可以运行在指定的几个 BSP 平台上，而且包括了完整的开发工具和文档。RTEMS/GeSys 是一个通用的系统，它包含了完整的 RTEMS 模块和网络配置模块，启动任务为 CEXP。这个系统与 RTEMS 相连，而且加载了完整的 newlib 库和 gcc 库，应用程序模块可以动态的加载或删除。这个开发环境与 Vxwork 的主机目标机开发模式非常相近。而且在这个系统上还可以加载 EPICS 应用（EPICS 是用于开发分布式实时控制系统的软件工具包，广泛应用于粒子加速器、高能物理实验、天文望远镜等大型科学实验设备的控制系统的开发）。

接下来为大家主要介绍一下 CEXP 这个极其有用的工具。CEXP 的主要目标平台是 RTEMS，但它也可以在支持 BFD library 的平台上运行。CEXP 能够识别基本的 C 语言变量类型，例如：

```
char,    char*
short,   short*
long,    long*
double,  double*
long     (*) ()
double  (*) ()
```

而且能够直接解释执行一些基本的 C 语句，例如：

```
Cexp> printf("Hello world")
Cexp> some_variable = 0xdeadbeef
Cexp> some_double_variable = *(double*)&some_variable
Cexp> a=printf, a && a("The square root of 2 is %g\n",sqrt(2.0))
```

符号表

CEXP 可以直接从可执行文件中读取符号表（Symbol Table）或者读取通过 xsyms 工具生成的 sym 文件。一旦成功读取符号表，CEXP 将从符号的大小来猜测符号的类型，但这仅仅局限于 ELF 格式的目标文件。有时候这种猜测是错误的，但用户可以通过强行转换来更正符号的类型。数组类型都是用 void* 来表示，用户使用的时候需要进行强制转换：

```
Cexp> *((long*)&some_array)+25)
```

或者使用符号重定义：

```
Cexp> long *some_array !
Cexp> *(some_array + 25)
```

‘!’ 表示将符号重定义为 ‘long *’ 型，随后就可以不用再强制转换就能直接使用，仅仅在新声明一个符号的时候不需要 ‘!’，这样能避免错误的重定义。

同时，CEXP 还提供了一些搜索函数

```
lkup(char *regex)
lkaddr(unsigned long addr)
```

C-Expression Parser / Interpreter

CEXP 是一个 C 语言表达式的解释器，但是也有一些高级操作是 CEXP 不支持的，例如：

- 不支持 ‘?’ ‘:’ 三元表达式
- 不支持 [] 下标操作和 ‘->’ 结构成员操作

```
- '.' 操作符有特殊的意义，在 CEXP 中 '.' 仅仅支持符号的成员函数 help() 操作
- 不支持多重指针。char** 在 CEXP 中是不可识别的，CEXP 也无法寻址 **addr 类型的表示，需要时用户必须这样定义：
    Cexp> *(char*)*(long*)some_char_pointer_pointer
```

但是其他大多数操作都是支持的，包括 '，' '&&' '||' 等等。条件表达式可以以类似如下的方式表示

```
Cexp> (dd=deviceDetect(_some_device_address)) && driverInit(dd)
```

对于函数调用操作，用户需要填入合适的输入参数，不使用的参数将被设置为0，这在大多数时候是安全的。因为在 ELF 格式的目标文件中，CEXP 使用 .stab 符号表，而不是 .symtab，所以符号表中没有提供函数返回值的类型，CEXP 假设所有的返回值都是 'long' 型。同样，用户也可以使用强制转换来修改返回值类型

```
Cexp> ((double(*)())sqrt)(2.345)
Or
Cexp> double (*sqrt)()
Cexp> sqrt(2.345)
```

Type Engine

CEXP 仅仅可以解析有限的一些变量类型。要注意的是 CEXP 将所有的整型都作为无符号类型处理，即使没有显示的添加 "unsigned" 关键字。

User Variables

CEXP 支持用户直接在命令行上定义全局变量，但是变量名不能与现有符号表中的变量相冲突。用户可以用简单的赋值语句来定义

```
hallo="hallo"
```

而且同样可以对自定义的全局变量进行各类操作

```
long hallo
chpnt = &hallo
*(char*)*chpnt
```

用户还可以定义函数指针，操作方法和标准 C 语言一致

```
s=((double(*)())sqrt)
printf("Let's print a double: %g\n", s(44.34))
```

Help Facility

在 cexp 中， '.' 仅仅支持符号的成员函数 help() 操作。

```
Cexp> some_symbol.help()
Cexp> some_symbol.help(1)
```

可以显示符号 some_symbol 的简单信息或者详细信息，包括变量的地址、当前值、类型描述等等。用户可以手动添加修改一个符号的附加信息：

```
Cexp> long myvar
```

```
Cexp> myvar.help("I just created a 'long' variable")
```

help() 信息是每个符号所特有的，对一个变量创建一个别名，原变量的 help() 信息并不会自动拷贝到新的别名，必要时需要进行手动拷贝

```
Cexp> s=(double (*)( ))sqrt  
Cexp> s.help(sqrt.help())
```

启动/调用 CEXP

CEXP 有两个入口函数

```
cexp_main(int argc, char **argv)  
cexp(char *arg1,...)
```

cexp 是调用 cexp_main 实现的。其实现过程是反复调用 cexp，执行启动脚本完成。调用语法如下：

```
cexp [-s <symbolfile>] [-h] [-d] [-q] [<scriptfile>]
```

[-h] 参数打印使用信息

[-d] 参数打开调试信息

以上两个参数只有在含有 --enable-YYDEBUG 是有效。

[-q] 参数将使 CEXP 不打印普通信息到 stdout，但是错误信息依然会打印到 stderr。

在第一次执行 cexp 时，[-s <symbolfile>] 是必需的，symbolfile 可以是包含符号表信息的可执行目标文件，也可以是一个单独的符号表文件，这个 symbolfile 将作为默认的全局符号表。随后的 cexp 调用如果没有 [-s <symbolfile>]，则加载默认的全局符号表。

CEXP 启动后，可以执行 [<scriptfile>] 脚本文件，或者直接从 stdin 读入命令操作。注释行用 ‘#’ 或者 ‘//’ 开头。

脚本文件的读取有如下一些方式：

```
Cexp> cexp("script_file")  
Cexp> < script_file  
  < a b c      --> read from file "a"  
  < 'a b' "c"  --> read from file "a b"  
  < "a'b"      --> read from file "a'b"  
  < "a\"b"     --> read from file "a\" (a followed by a single backslash)
```

Loadable Modules / Runtime Loader

针对 BFD library，CEXP 支持动态装载模块。CEXP 保持对装载模块的依赖性跟踪。如果有某个 B 模块依赖 A 模块的话，A 模块是不允许被卸载的。

一个模块 (someModule.o) 可以用如下的命令装载

```
Cexp> someModule=cexpModuleLoad("someModule.o")
```

装载操作返回一个模块 ID，在上述例子中返回值被保存在用户定义变量 someModule1。如果装载出错，则返回 NULL。

模块可以使用如下命令卸载

```
Cexp> cexpModuleUnload(someModule)
```

另外，还有一些常用的命令

```
cexpModuleInfo([ID])
```

输出指定模块的信息，如果 ID 为 NULL，则输出所有已装载模块的信息。

```
cexpModuleFindByName("regex")
```

在已装载模块中搜索指定表达式，返回值为第一个找到的模块 ID，如果没有找到任何结果则返回 NULL。

RTEMS 开发注意事项

下面列出了一些由于 RTEMS 操作系统的特点而带来的 RTEMS 应用开发中需要注意的问题。

无内存保护

对于 X86 体系系统结构和其它 CPU 架构，RTEMS 没有采用虚拟内存管理，也没有使用分段保护机制（把整个地址空间作为一个大段使用，相当于屏蔽了段），所以 RTEMS 是没有提供内存保护机制。这样做的好处在于实现简单、上下文切换的代价会变化，当然也存在很大的安全问题，应用程序中的一个错误的内存操作都可能使整个系统崩溃。但是考虑到嵌入式系统运行的程序相对确定，一般都是在调试完成后再发布产品，这样这个缺点在实际当中就没有那么可怕了。另外，由于它没有使用虚拟内存技术，它也只能使用有限的物理地址空间。

固定大小的堆栈

在基于通用操作系统（如 Windows、Linux）的应用开发中，一般不需要了解应用程序对堆栈的使用量。这主要是由于通用操作系统在中提供了虚拟内存管理功能，堆栈空间的按需扩展是由操作系统自动完成的，应用程序不需要了解其操作过程。但在 RTEMS 中，由于操作系统采用的是扁平内存管理模式，直接对物理内存进行访问，所以应用程序无法由操作系统来自动完成堆栈的扩展，只能由应用程序在编译前指定好堆栈的位置和大小，且由应用程序保证其在运行过程中，不会溢出堆栈空间。这给 RTEMS 应用程序开发人员提出了比较高的要求，在使用堆栈的时候要小心谨慎。

基于优先级的调度

rtems 是一个完全基于优先级调度的实时操作系统，它一共有 255 个优先级，每个优先级上又可以运行多个任务，在同一优先级上的任务可以指定 FIFO 和时间片轮转的调度策略。

并发与同步

由于 RTEMS 是一个支持多任务的，可抢占的硬实时操作系统，所以内核的很多共享数据都应该能够并发地被访问。这就要求有同步机制保证不出竞争条件。可以通过如下手段来达到目的：

- ❏ 屏蔽中断。因为中断完全是异步的，不管当前正在执行什么代码，中断都有可能发生。如果中断在代码访问共享资源时到来就可能造成竞态。所以，屏蔽中断是保证同步的一种方法，但是如果长时间屏蔽中断肯定会影响系统的实时性
- ❏ 禁止调度。如果始终只有一个线程访问共享资源，在没有中断发生的情况下，是不可能产生竞态的。
- ❏ 信号量。信号量也是内核提供的同步机制，它会对共享数据进行保护。

内核需要在不同情况下采用不同的同步机制来保证并发访问。

可移植性

RTEMS 具有很好的可移植性，大部分代码都是 C 语言实现的，与体系结构无关，在许多不同的体系结构中都可以编译执行。和体系结构相关的部分主要在：

```
$RTEMS_ROOT/c/src/lib/libcpu  
$RTEMS_ROOT/c/src/lib/libbsp  
$RTEMS_ROOT/cpukit/score/cpu
```

这样就很好地把系统结构相关的代码从内核代码树中分离出来，为移植提供了很大的方便。

C 运行库

newlib 是一个用于嵌入式系统的开放源代码的 C 语言程序库。特点是轻量级，速度快，可移植到很多 CPU 结构上。最初是由 Cygnus Solutions 收集组装的一个源代码集合，取名为 newlib，目前由 Red Hat 维护，最新的版本是 1.15。newlib 尽量符合 POSIX 标准，目前 newlib 主要在 I/O 和文件操作（open、close、read、write、lseek、fcntl、link、unlink、rename 等）、大内存堆需求（sbrk 等）、得当前系口的日期和时间（gettimeofday、times 等）、各种类型的任口管理函数（execve、fork、getpid、kill、wait、_exit 等）等方面有比口完善的支持。但由于 RTEMS 本身不支持口程，口致它并不象 linux 的口用程序广泛使用的 glibc 那口完善和全面。所以在编程的时候需要注意它与 glibc 的区别。

小口

第三章 总体结构和系统构成

概述

现在读者已经大致了解了实时操作系统的概念和特征，对 RTEMS 的历史和发展现状也有了一定的了解，对 RTEMS 的安装、编译和应用开发也有了初步的认识。为了让读者对 RTEMS 的内部设计有一个总体的感受，本章将对 RTEMS 的设计目标、总体设计思路、总体架构进行分析；并进一步对 RTEMS 的底层系统支持机制进行阐述；接下来对构成 RTEMS 主要功能的核心组件进行描述，并说明它们之间的交互关系；最后对 RTEMS 的核心代码结构进行描述。

设计目标

RTEMS 是一个典型的硬实时操作系统，所以它遵循并实现了如下的设计目标和规范：

- ❏ 提供对硬实时和软实时的支持：为防止优先级反转，提供优先级继承算法和优先级天花板算法；提供支持速率单调（Rate Monotonic）调度。其硬实时性能出众，与 VxWorks 等商用实时操作系统相比，其实时性处于同一个量级。
- ❏ 高度可移植性：核心代码使用 C 语言编写，采用 GCC 编译器来编译，可移植性好，支持主流嵌入式 32 bit CPU，包括 x86、ARM、PowerPC、MIPS、SH、ColdFire、m68k、Sparc、NIOS2、BlackFin、DSP（Texas Instruments C3x/C4x）等，对部分 64 bit CPU（MIPS64）也有一定的支持，并将支持 8 bit、16 bit CPU。
- ❏ 高度可裁减和可配置：通过宏定义可在编译时进行裁减和定制，选择把不同的内核组件组合在一起，实现特定需求的最小系统。在 32 bit 的嵌入式系统中最小的内核只有 30K Bytes 左右。
- ❏ 支持多处理器：这里的多处理器支持不同于通用操作系统的 SMP 支持功能，RTEMS 中多个处理器是协作关系，是一种非对称多处理（AMP）方式，这样每个 CPU 都需要维持一个 RTEMS 地址空间。
- ❏ 丰富的应用支持：采用 newlib 嵌入式 C 库、提供 POSIX API 接口/Itron API 接口/RTEID API 接口、支持图形用户界面（Microwindows/Nano-X）、支持文件系统（IMFS、FAT 等）、具有完整的 TCP/IP 协议栈以及 FTP、WebServer、NFS 等网络服务。

现在只是对 RTEMS 已经达到的设计目标的简单描述，随着对后面章节的深入展开，读者将会看到这些设计目标是如何有机组合在一起并实现在 RTEMS 内核中的。在分析 RTEMS 内部细节之前，让我们先来看看 RTEMS 的设计思路。

总体设计思路

RTEMS 基于典型的实时操作系统设计思路，与典型的通用操作系统（如 Linux、Solaris、Windows 等）在设计上差别很大。简单地说，RTEMS 是一种基于扁平内存管理的层次型单体内核，应用程序与 RTEMS 内核形成一个基于函数调用形式的整体功能单元，在一个地址空间下协同完成特定的功能。而对于通用操作系统而言，应用程序和操作系统是处于不同的特权状态和地址空间。代表应用程序的用户态进程运行

在 CPU 的用户态（又称非特权模式，用户模式），无直接访问系统硬件和操作系统中的系统数据，而操作系统运行在 CPU 的核心态（又称特权模式，内核模式），可以访问系统硬件和核心数据。下面分别从系统调用接口、进程/线程管理、内存管理、文件系统、I/O 管理等几个方面进行总体分析。

系统调用是应用程序访问操作系统的接口。在系统调用接口上，通用操作系统与基于此操作系统的应用程序处于两个不同的 CPU 特权态，操作系统处于核心态，而应用程序处于用户态。在核心态可以执行 CPU 特权指令，而用户态无法执行特权指令，且只能通过特定的指令或中断来访问操作系统提供的各种功能。这在一定程度上保证了系统整体的安全，避免应用程序对操作系统可能的破坏。RTEMS 没有区分用户态和操作系统核心态，RTEMS 内核和基于 RTEMS 的应用程序之间是简单的函数调用关系，且应用程序和 RTEMS 内核都处于核心态，这样的好处是执行速度更快了，但在安全性上则相对削弱。

在内存管理方面，通用操作系统采用了虚拟内存管理方式，这样可以让内存需求超过实际物理内存的进程/线程能够执行，其主要思想是把重要和常用的数据和执行代码放在物理内存中，把不常用的数据和执行代码放到二级存储（这里主要指的是硬盘等可在掉电后保存数据的存储介质），随时根据系统执行情况替换放在内存中的数据和代码。而且通过虚存管理可以实现对不同内存区域的保护，不同进程之间，或者应用程序和操作系统之间的地址空间相对隔离。这样一般情况下不同进程的地址空间不能直接访问，且应用程序不能直接访问内核地址空间。所以一个与错误的应用程序不会导致系统的崩溃，从而增加了系统的可靠性，但由于可能访问二级存储，导致内存访问的时间无法满足实时性的要求。RTEMS 操作系统没有采用虚拟内存管理，而是采用了简单的单一地址空间管理方式。这样应用程序和 RTEMS 内核在一个地址空间，相互之间可以直接访问，这样的好处是可以使得需要访问的数据和代码都在内存中，从而确保内存访问的确定性和实时性，而且还避免了内存管理的开销和地址切换的开销。但在安全性和可靠性方向则削弱了不少，需要靠应用程序开发人员保证应用程序的正确性和可靠性。一旦基于 RTEMS 的应用程序产生错误，很可能造成整个系统的崩溃。在内存分配和释放的管理上，RTEMS 相对实现得比较简单，主要有固定分区分配和可变分区分配等，在可变分区分配上主要采用首次适配算法，容易产生内存碎片，但实时性可以得到较好的保证。

在进程/线程管理方面，当前通用操作系统结合虚存管理，采用进程和线程结合的管理方式。进程代表了一个应用执行的过程以及其所占用的计算机资源（包括 CPU、内存、文件等），进程的执行流用线程来表示。操作系统的调度单位是线程。一个进程可以包含多个线程，属于同一进程的多个线程共享进程管理的资源，比如属于同一进程的多个线程共享进程所管理的内存，这样这些线程可以直接访问属于进程的全局地址空间。RTEMS 操作系统没有进程的概念，只有线程的概念，在 RTEMS 中把线程称为 task。由于 RTEMS 的应用程序和内核在一个统一的地址空间，且处于一个特权态，所以 RTEMS 的线程共享应用程序和 RTEMS 内核所拥有的整个地址空间。我们可以把应用程序和 RTEMS 内核理解为一个特殊的“进程”，在这个“进程”中，所有的线程共享其资源。这样的好处是线程开销小，资源管理和线程调度相对简单。在设计调度算法、同步互斥算法和线程间通信方面会比通用操作系统简化，这会进一步确保整个系统的实时性。

在文件系统管理方面，当前通用操作系统结合虚存管理，实现了多种复杂、高效且可靠的文件系统，且建立了一个统一的虚拟文件系统层，屏蔽不同文件系统的差异，对上层提供统一的接口。且与用户管理和进程管理结合，可实现安全管理，保证对文件的安全访问。RTEMS 操作系统没有用户和进程的概念，在安全性上的基本没有考虑。但 RTEMS 也与一个相对简单的虚拟文件系统层，支持数量有限的嵌入式文件系统，如 IMFS（基于内存的文件系统）和 DOSFS（即 FAT 文件系统）。

在 I/O 管理方面，RTEMS 操作系统与通用操作系统（特别是类 UNIX 操作系统）差别不是特别大，都把设备“看成”是一种特殊的设备文件，有设备号，用文件的访问接口来进行打开、关闭、读、写和控制等操作。在灵活性方面，RTEMS 驱动程序不能象通用操作系统那样根据硬件情况动态加载，而是在编译的时候就静态确定的，这是当前实时操作系统在灵活性方面普遍缺少的功能。在中断管理方面，强调确定性和实时性，对驱动程序中的中断处理例程的时限性方面要求高。

操作系统模型与总体结构

从操作系统模型上来看，RTEMS 是一个单地址空间的层次式单体内核，不是微内核（microkernel）模型的操作系统（如 Mach，QNX），与通用操作系统（如 Linux）的架构在地址空间和特权模式上也有一定的差别。这种操作系统模型是嵌入式操作系统（如 VxWorks、ThreadX、Nucleus）最常见的模型之一。

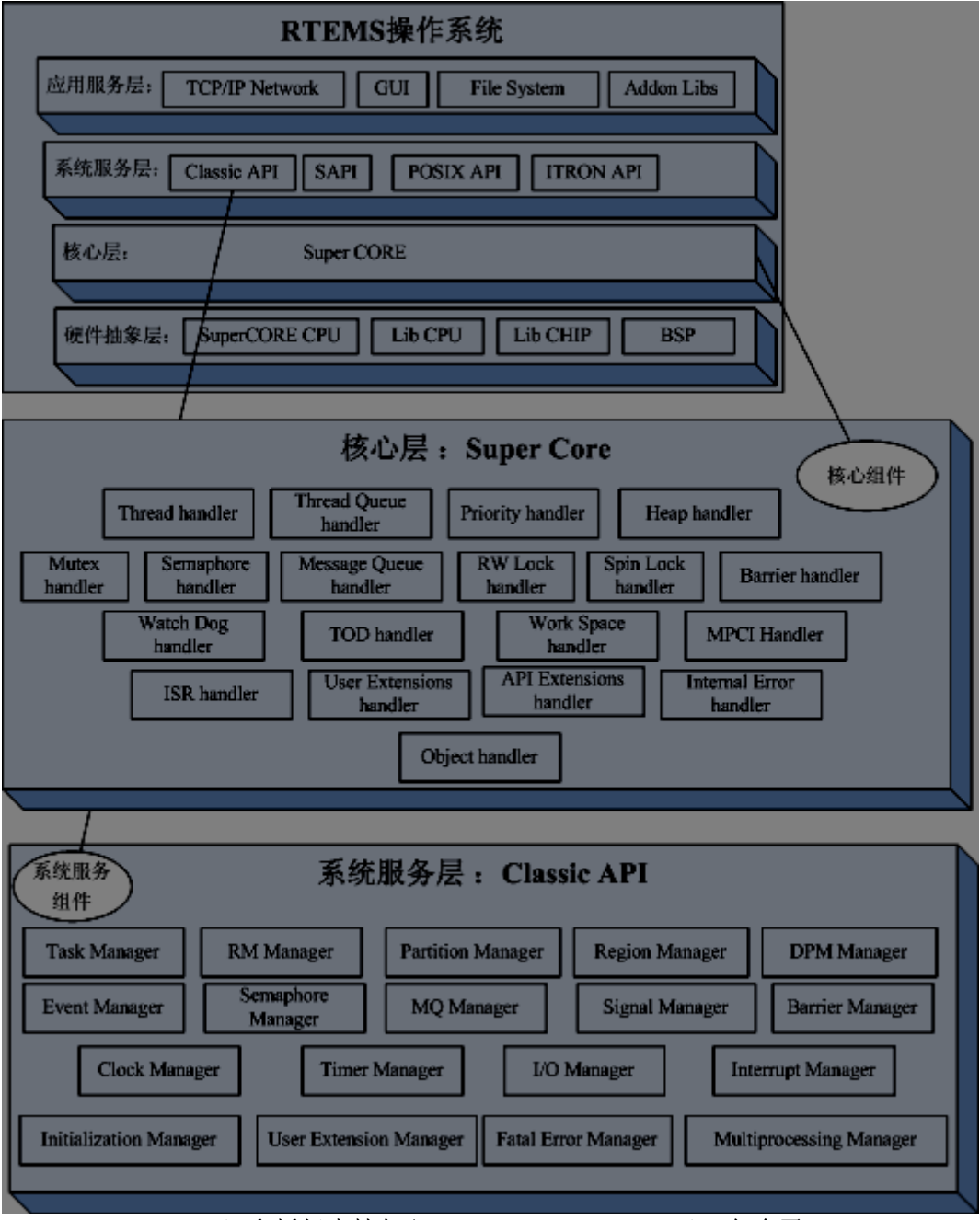
RTEMS 操作系统按照功能组件来组织整个内核。在组件的设计上采用了面向构件（Component oriented）和面向对象（Object oriented）的设计思想。在 RTEMS 操作系统中，任务、消息队列、互斥

量等都是以对象的形式存在，通过对象 ID 可以访问到对象的内容。与面向对象的编程类似，RTEMS 提供统一的接口函数表来动态的创建、删除、操作预定义的对象类型，屏蔽不同对象底层的细节差异。与面向构件编程的方式类似，RTEMS 操作系统通常不直接访问组件的数据结构，而是利用一系列的接口来访问和修改相应组件的数据结构。虽然 RTEMS 操作系统内部普遍使用了对象和构件来表示和设计，但从严格意义上说，RTEMS 不是面向对象或面向构件的操作系统，而只是借用了面向构件和面向对象的设计思想。不过想到 RTEMS 出现在二十世纪八十年代初期，我们不得不佩服当初的设计者的编程和设计思想很先进。

RTEMS 的总体结构如下所示：

从上图
出，
整个分
层次，
是与硬
相关的
象层

可以看
RTEMS
为四个
最底层
件直接
硬件抽



(Hardware Abstract Layer) 和板级支持包(Board Support Package), 包含了 SuperCore CPU (上层核心组件中与 CPU 相关的定义和功能实现), libcpu 库 (包括各种 CPU 初始化功能), libbsp 库 (包含各种外设的驱动程序)。

核心层

第二层是 RTEMS 的核心层，即 RTEMS 执行体 (RTEMS Executive) -- SuperScore, SuperScore 由一系列核心组件组成，每个组件提供了一个服务函数集合，用于不同组件之间的相互操作以及给高层的系统服务层提供服务。这一层主要的核心组件包括：

与线程管理和调度相关的组件

- ✎ thread（线程）handler 组件：RTEMS 为了支持不同高层 API 接口（Classic/Posix/Itron）的线程（thread）管理方式，实现了 thread handler 组件，它提供了 Thread_Control 结构体和以此配套的一组基本的线程管理的数据结构和管理函数，涉及线程初始化、线程启动、线程暂停、线程恢复执行、线程切换、线程属性管理等。不同的高层 API 接口实现只是对这些结构和函数进行封装，这样线程管理就可以分为两层，对 RTEMS 支持不同 API 接口扩展会很方便。详细分析内容在“线程管理与调度”一章讲述。
- ✎ thread queue（线程等待队列）handler 组件：在实现同步和互斥时，需要改变线程的执行状态，且需要把与此资源相关的线程管理（让线程睡眠或唤醒）起来。在 RTEMS 中，通过 thread queue handler 组件可以把阻塞在同一个资源上的各线程按不同的策略排队。线程等待队列是通过 Thread_queue_Control 结构体及相关操作来实现的。详细分析内容在“线程管理与调度”一章讲述。
- ✎ priority（优先级）handler 组件：此组件主要完成对优先级的高效计算和和读写操作，为上层的任务调度提供支持。
- ✎ watchdog（看门狗）handler 组件：watchdog 是一种定时器机制，可用于完成对超时的处理。对这种机制的使用在 RTEMS 中随处可见。RTEMS 在核心层实现了 Watchdog_Control 结构体和相关操作。详细分析内容在“时间服务”一章讲述。

与同步互斥相关的组件

- ✎ mutex（互斥）handler 组件：RTEMS 需要为每个要互斥访问的共享资源提供保护手段。互斥体（mutex）是完成这一功能的有效手段。在 RTEMS 中，互斥体通过 CORE_mutex_Control 结构体及相关操作实现了对线程的阻塞、调度等各项功能。详细分析内容在“同步互斥与通信机制”一章讲述。
- ✎ semaphore（信号）handler 组件：当多个线程需要共享的个数大于1的共享资源时，需要多值信号量机制的保护支持。RTEMS 在核心层实现了 semaphore handler 组件，信号量通过 CORE_semaphore_Control 结构体及相关操作来实现的。semaphore handler 组件会对线程的执行状态产生影响，详细分析内容在“同步互斥与通信机制”一章讲述。
- ✎ message queue（消息队列）handler 组件：消息队列是实现多个线程之间需要进行数据传递的有效手段。RTEMS 中的 message queue handler 组件主要包括 CORE_message_queue_Control 结构体及相关操作。message queue handler 组件会对线程的执行状态产生影响，详细分析内容在“同步互斥与通信机制”一章讲述。
- ✎ rwlock（读写锁）handler 组件：RTEMS 在核心层读写锁实现了基于读者优先的锁访问机制，在核心层提供了 CORE_RWLock_Control 结构体和相关操作，对超时也提供支持。目前的 RTEMS 版本提供的 rwlock handler 组件主要给高层的 POSIX API 的 posix rwlock manager 提供支持。
- ✎ spinlock（自旋锁）handler 组件：如果应用每次对某共享资源的访问时间很短，则采用自旋锁是完成这种特定情况下互斥访问的有效手段。RTEMS 在核心层通过 CORE_spinlock_Control 结构体及相关操作来实现的。目前的 RTEMS 版本提供的 spinlock handler 组件主要给高层的 POSIX API 的 posix spinlock manager 提供支持。
- ✎ barrier（屏障）handler 组件：此组件为多个线程在某执行点进行同步提供支持。RTEMS 在核心层通过 CORE_barrier_Control 结构体和 CORE_barrier_Attributes 结构体以及相关操作来实现的。屏障组件会对线程的执行状态产生影响，详细分析内容在“同步互斥与通信机制”一章讲述。

与内存管理相关的组件

- ✎ heap（堆管理） handler 组件：核心层的堆管理机制是实现高层的可变分区动态内存管理的基础。堆管理具体实现了 RTEMS 高层中动态分区内存管理的功能。RTEMS 在核心层实现了 Heap_Control 结构体和相关的堆空间分配、堆空间释放操作。详细分析内容在“内存管理”一章讲述。

其他重要组件

- ✎ object（对象）handler 组件：RTEMS 提供了动态创建、删除、操纵对象的接口。这些对象包括任务、

消息队列、同步互斥、内存管理对象等。通过对象 ID 就可以区分对象类别、对象节点号和对象索引，这样基于 object handler 组件设计高层的组件可以有效地重用代码，简化模块化编程的复杂度。RTEMS 在核心层通过 Objects_Control 结构体、Objects_Information 结构体及相关操作来实现的。详细分析内容在本章后续章节讲述。

- ✎ Time Of Day (时间, 简称 TOD) handler 组件: TOD handler 组件提供了对时间的基本支持, 主要完成了对系统启动后时间的更新维护, 并对高层的时间管理等提供支持。详细分析内容在“时间服务”一章讲述。
- ✎ ISR (中断处理) handler 组件: 这个组件主要完成处理函数的初始化, 建立中断上下文的堆栈, 为系统服务层的中断管理器组件提供服务。详细分析内容在“中断处理”一章讲述。
- ✎ workspace (工作区) handler 组件: 这个组件主要通过调用堆管理组件的初始化服务, 完成 RTEMS 的 workspace 的内存空间初始化等工作, 为系统服务层的初始化管理器提供支持。workspace 是 RTEMS 配置表中的核心数据的内存空间。同时借助 heap handler 组件提供在 workspace 分配内存空间的服务。
- ✎ user_extensions (用户扩展) handler 组件: 这个组件主要是在线程管理和错误管理的函数中增加扩展点支持服务, 可以让用户在扩展点实现特定的功能 (如统计线程启动的次数等)。
- ✎ API_extensions (API 扩展) handler 组件: 这个组件主要在初始化驱动程序前后和线程切换后提供扩展点支持服务, 这样可以让 RTEMS 实现特定的功能。详细分析内容在“线程管理与调度”一章讲述。
- ✎ MPCIE Handler 组件: 完成对多处理器通信接口的支持, 提供相关系统数据的初始化, 远程服务线程启动, 远程数据的接收和发送等服务。为系统服务层部分组件 (如信号量等) 的多处理器提供帮助。
- ✎ Internal_Error (内部错) handler 组件: 此组件主要是帮助系统服务层的 fatal_error_manager (致命错误管理器) 组件完成对错误的处理。

系统服务层

第三层是系统服务层, 主要完成给各种应用提供操作系统级的服务支持。借助与 RTEMS 核心层的帮助, 系统服务层主要提供了三种不同的 API 接口: Classic API、Posix API 和 Itron API。到目前 RTEMS 的最新版本为止, RTEMS 对 Classic API 和 Posix API 的支持比较完善, 而 Itron API 的支持相对较弱, 实现不全面。下面对 Classic API 提供的服务进行描述。

与任务调度相关的组件

- task_manager (任务管理器) 组件: 这里的 task (任务) 实际上是一种“轻量级线程”, 是对核心层的 Thread_Control 结构体的进一步封装。任务管理器借助于核心层与线程管理和调度相关的组件的帮助, 完成任务的创建、启动、删除、重启、阻塞、恢复运行、定时唤醒等多种服务。详细分析内容在“线程管理与调度”一章讲述。
- ✎ rate_monotonic_scheduling (速率单调实时调度 简称 RMS) 组件: RMS 组件通过速率单调实时调度算法来调度周期性任务。RMS 组件能保证计划的任务能够在指定时间范围内周期性执行。由于 RMS 组件对时间管理的需求, 需要核心层的看门狗组件的支持。详细分析内容在“线程管理与调度”一章讲述。

与同步互斥与通信相关的组件

- ✎ barrier_manager (屏障管理器) 组件: 屏障管理器组件用于多个任务在某执行点上同步。屏障管理器组件基于核心层的 CORE_barrier_Control 结构体和 CORE_barrier_Attributes 结构体以及相关函数来实现多任务同步。详细分析内容在“同步互斥与通信机制”一章讲述。
- event_manager (事件管理器) 组件: 事件管理器定义了事件标志 (用 bit 表示), 并高效实现被任务 (或 ISR) 用来通知另一个任务有事件发生。每个任务可以有三十三个相关的事件标志。事件管理器组件借助核心层的 Thread_Control 结构体的扩展域完成事件标志的发送和接收处理。详细分析

内容在“同步互斥与通信机制”一章讲述。

- message queue manager (消息队列管理器) 组件: 消息队列管理器实现在任务间或者任务与 ISR 之间传递可变长度的信息。消息队列管理器负责存储, 转发任务间通信的信息, 并更新消息状态改变任务的执行状态。事件管理器组件借助核心层的 message queue handler 组件来完成具体的管理工作。详细分析内容在“同步互斥与通信机制”一章讲述。
- signal manager (信号管理器) 组件: 信号管理器为任务提供异步通信的手段。信号管理器允许一个任务定义一个异步信号例程 (ASR), 当一个信号被发送到一个任务的时候, 该任务的执行将会转到 ASR。送一个信号给一个任务不会影响任务的状态。信号管理器组件借助核心层的 Thread_Control 结构体的扩展域完成信号的发送和接收处理。详细分析内容在“同步互斥与通信机制”一章讲述。
- semaphore manager (信号量管理器) 组件: 信号量管理器实现了标准的 Dijkstra 计数信号量和高效率的二值信号量, 提供信号量创建、删除、获取等操作, 来为任务间互斥同步访问共享资源提供支持。同时信号量管理器对二值信号量还提供优先级继承和优先级天花板机制的支持, 来解决优先级反转问题。信号量管理器组件借助核心层的信号组件来完成具体的操作。详细分析内容在“同步互斥与通信机制”一章讲述。

与内存管理相关的组件

- partition manager (分区管理器) 组件: 分区管理器主要负责对固定大小内存的动态分配和释放进行管理。分区管理器维护定长内存区, 定长内存区是一个连续的内存块, 内存块中有固定大小的 buffer, 通过链表链接起来, 通过简单对链表插入和删除等操作就可以完成对 buffer 的释放和分配。详细分析内容在“内存管理”一章讲述。
- region manger (区域管理器) 组件: 区域管理器主要负责对可变大小内存的动态分配和释放进行管理。区域管理器维护变长内存区 (region), 变长内存区是一个连续内存区, 区内有用户定义的变长内存块—page, page 的大小可变。区域管理器借助核心层的堆组件, 将变长内存区按照双向链表组织, 使用首次匹配算法来完成内存请求分配。详细分析内容在“内存管理”一章讲述。
- dual ported memory manager (双端口内存管理器) 组件: 双端口内存指的是可以由两个处理器同时访问的内存芯片。在 RTEMS 中的双端口内存管理器为需要访问这种内存的应用提供服务支持。双端口内存管理器针对双端口内存区 (dual ported memory, 简称 DPMA) 的特点, 用端口 (port) 定义为内部地址和外部地址的映射, 实现地址转换功能, 使得该区域可以被多个处理器访问。DPMA 的属主使用内部地址访问 DPMA, 其他处理器使用外部地址访问 DPMA。详细分析内容在“内存管理”一章讲述。

与外设处理相关的组件

- Interrupt manager (中断管理器) 组件: 为了保证 RTEMS 的实时性, 需要中断管理器能够支持 RTEMS 对外部中断的快速响应。中断管理器负责整个 RTEMS 的中断建立、中断执行流控制、屏蔽中断、使能中断等工作。在具体实现上, 中断管理器借助核心层的中断组件, 完成中断向量表的建立, 使得中断号和对应的中断服务例程 (ISR) 间形成一对一的映射。当中断产生后, CPU 会根据中断向量表执行 RTEMS 的中断服务例程。具体的中断服务例程和与硬件相关的实现由 BSP 提供。详细分析内容在“中断处理”一章讲述。
- Clock manager (时钟管理器) 组件: 时钟管理器组件主要提供对日期和其他的与时间相关的服务。其中发布时间信号的函数 rtems_clock_tick 很关键, 当时钟中断产生时候, 时钟中断服务例程会调用它, 而它会进一步调用核心层的 TOD 组件、看门狗组件和线程组件中与时间相关的函数, 为与时间相关的调度、同步互斥等提供时间更新支持。详细分析内容在“时间服务”一章讲述。
- Timer manager (定时器管理器) 组件: 定时器允许在特定的时间调用激活的任务。而定时器管理器提供对定时器管理服务。在具体实现上, 定时器管理器可以在中断层 (通过函数 rtems_clock_tick) 和任务层 (通过定时器服务器任务) 实现定时更新和相关处理操作。详细分析内容在“时间服务”一章讲述。
- I/O manager (I/O 管理器) 组件: I/O 管理器为了实现对设备的统一访问, 提供一种基于文件访问方式的结构化驱动程序接口设计。I/O 管理器通过设备驱动表 (Device Driver Table) 来描述驱动

程序的配置。使用 RTEMS I/O 管理器的应用在编译前需要指的设备驱动表以及配置表格的地址，这样在初始化管理器执行系统初始化时，完成设备驱动程序的加载和配置。且通过 I/O manager 的统一访问接口屏蔽了不同驱动程序的差异，给上层应用提供了一致的访问方式。有关这方面的内容和块设备，字符设备等的详细分析内容在“I/O 系统”一章讲述。

其他组件

- initialization manager（初始化管理器）组件：当板级支持包（BSP）完成硬件级的基本初始化工作后，将把控制权交给初始化管理器，初始化管理器负责启动和关闭 RTEMS。启动 RTEMS 包括创建并且启动所有的配置好的初始化任务，并且初始化 RTEMS 系统使用到的设备驱动程序。初始化管理器对系统初始化后，将把 CPU 控制移交到应用程序。但应用程序退出后，初始化管理器完成对 RTEMS 的关闭。详细分析内容在“初始化过程”一章讲述。
- fatal error manager（致命错误管理器）组件：当 RTEMS 在运行时出现致命错误时候，会通过调用 `rtems_fatal_error_occurred` 函数，把 CPU 控制权转给致命错误管理器，致命错误管理器借助核心层的内部错误组件调用缺省的内部错误处理函数或用户定义的内部错误处理函数，并停止系统运行。如果是 RTEMS 系统发现致命错误，RTEMS 将调用致命错误管理器；如果是应用程序发现致命错误，那么应用程序将调用致命错误管理器。
- User Extensions Manager（用户扩展管理器）组件：RTEMS 用户扩展管理器允许程序员对 RTEMS 进行扩展。扩展函数的作用其他操作系统中的钩子（hook）函数。这对于 RTEMS 系统功能的扩展用处很大。在 RTEMS 的某些执行点或在某些系统事件发生时，用户可以让 RTEMS 执行用户的扩展函数。用户扩展管理器利用核心层的线程组件中的 `Thread_Control` 结构体的 `extensions` 指针域存储用户扩展信息，同时还利用核心层的用户扩展组件在执行点调用具体的用户扩展函数。详细分析内容在“线程管理与调度”一章讲述。
- Multiprocessing Manager（多处理管理器）组件：为了在多个 CPU 间共享数据和全局资源，RTEMS 通过多处理管理器实现对远程资源的访问。多处理管理器适用于紧耦合多处理系统、松耦合多处理系统、混合耦合多处理系统、异构多处理系统。这主要通过对这些系统的抽象，定义了处理器结点（描述 CPU）、全局对象（描述远程资源），并通过全局对象表对远程资源统一管理，采用类似与 RPC（远程过程调用）的远程交互机制进行跨节点的资源访问。

第四层是应用服务层，这里主要提供了 TCP/IP 协议栈、GUI 系统、文件系统服务等。TCP/IP 协议栈移植于 FreeBSD 的 TCP/IP 协议栈，GUI 服务主要移植了 Microwin GUI 系统。文件系统主要支持 IMFS（内存文件系统）和 DosFS（FAT 文件系统），Addon Libs 是一些应用程序调用的软件库，包括 `zlib`、`libavl`、`tcl`、`ncurses`、`readline` 等。应用程序主要使用第三层和第四层提供的服务完成各种特定的功能。其中的文件系统服务的详细分析内容在“文件系统”一章讲述。

对象管理机制

RTEMS 设计了一种对象模型，以便为核心层和系统服务层的各种组件提供一致和安全的访问手段。这种对象模型的具体实现是位于核心层的 `object handler` 组件，它可以完成如下目标：

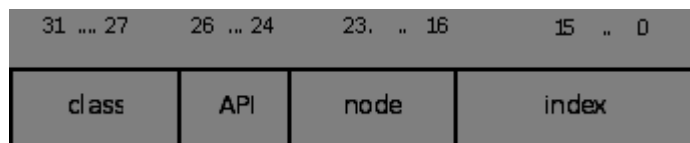
- ❧ 提供一种统一的机制来表示系统资源；
- ❧ 提供了一种对象命名方法，可以方便地实现对不同类型资源的扩展命名；
- ❧ 提供了一种统一的访问支撑机制来使用系统资源；
- ❧ 提供一种统一的机制来审计对象的使用情况，为系统管理对象的使用提供帮助。

下面将就对象的命名方法、对象的组织方法、基于类的对象管理方法和具体的实现进行更深入分析。

对象的命名方法

在 RTEMS 中，任务、消息队列、互斥量等组件都可以理解作为一种特殊的对象，且受核心层的 `object handler` 组件的管理。`object handler` 组件提供对象的接口函数来动态的创建、删除、操作预定义的对象类型。所有的对象由应用程序和 RTEMS 创建，由应用程序分配对象名，并由 `object handler` 组件分配对象 ID。这里需要注意的是对象名和对象 ID 都是 32 位的无符号数。在内核操作中，区分对象使用的是对象

ID，可以多个对象使用同样的对象名称。对象 ID 的结构如下图所示：



对象 ID 的结构表示

- ✎ Bits 0 .. 15 = index (up to 65535 objects of a type)
- ✎ Bits 16 .. 23 = node (up to 255 nodes)
- ✎ Bits 24 .. 26 = API (up to 7 API classes)
- ✎ Bits 27 .. 31 = class (up to 31 object types per API)

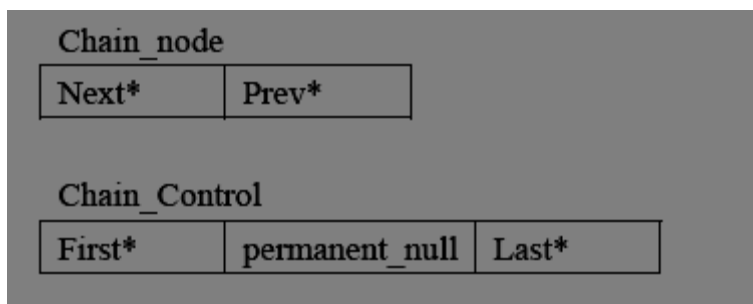
其中，高5位为 class 项，表示对象的属性类（此类属于某 API 类），例如是一任务或者消息；接着的3位表示 API 项，表示对象的 API 类，例如是 POSIX API 类或 CLASSIC API 类等，这样通过 API 类和属于此 API 类的属性类一起构成了对象的类；中间8位为 node 项，指的是对象所在的 CPU 节点号；低16位是 index 项，表示这个对象在它在所属类中的索引，由此可以看出同种对象类型最多有65535个对象。这样在一个复杂的多处理器系统中 也能很快找到指定的对象。

对象的组织方法

为了提高实时性，RTEMS 对象的管理采用了双向链表结构。为此，RTEMS 提供了链表结点 Chain_Node 和 Chain_Control 两种结构体。Chain_Node 的结构体定义如下：

```
struct Chain_Node_struct {
    Chain_Node *next;      //链上的下一个节点
    Chain_Node *previous;  //链上的前一个节点
};
```

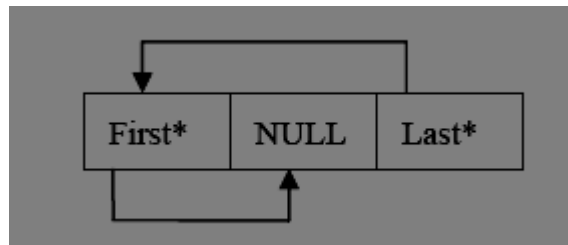
Chain_Node 结构如下图所示。



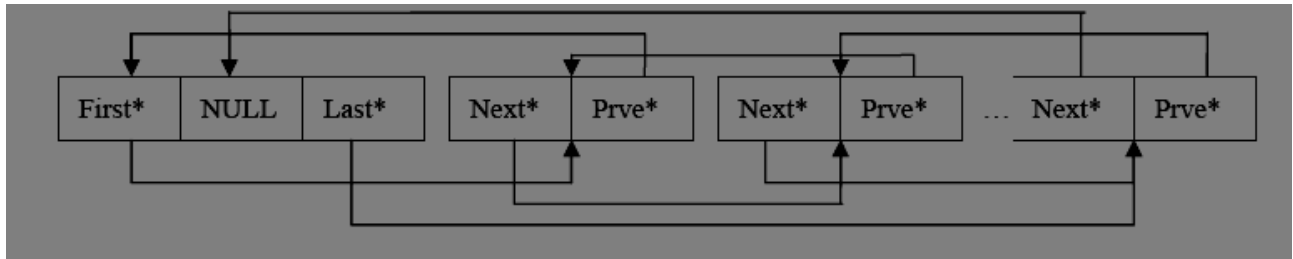
Chain_Control 的结构体定义如下：

```
typedef struct {
    Chain_Node *first;  //链上的第一个节点
    Chain_Node *permanent_null; //链上的空节点
    Chain_Node *last;   //链上的最后一个节点
} Chain_Control;
```

在链表为空的时，Chain_Control 如下图所示：



当链表不为空的时候，Chain_Control 如下图所示：



RTEMS 双向链表和普通的双向链表不同，它既可以看作是有头结点的双向循环链表。同时，此双向链表没有数据域。这使双向链表具有通用性和灵活性，对于不同的对象类型只需将链表结点和对象数据域构成新的节点即可，而对于链表的插入和删除等操作具有一般性，这在某种程度上体现了类似于面向对象语言的继承和多态的概念。而表示对象的数据结构 Objects_Control 的第一个成员就是 Chain_Node 结构，这样就可以把指向 Chain_Node 和 Objects_Control 的指针方便地进行转换，通过对象找到其在链表上的结点，或者通过结点找到对象。

基于类的对象管理方法

在 RTEMS 中，一个对象属于某一类，而所有对象的基类可以看做为 Objects_Control 结构体。Objects_Control 的定义如下：

```
typedef struct {
    Chain_Node    Node; // 一个对象的链节点
    Objects_Id    id;   // 对象的 ID
    Objects_Name  name; // 对象的名字
} Objects_Control;
```

我们可以把此基类看着是一个“继承”了 node 类的继承对象类。而象线程控制块（Thread_Control_struct）、定时器控制块（Timer_Control）等结构体都是“继承”了 Objects_Control 基类的继承类。它们“继承”Objects_Control 基类的实现方式也很简单，即把基于 Objects_Control 结构体的成员变量作为继承类的第一个成员变量即可。如定时器控制块的定义如下：

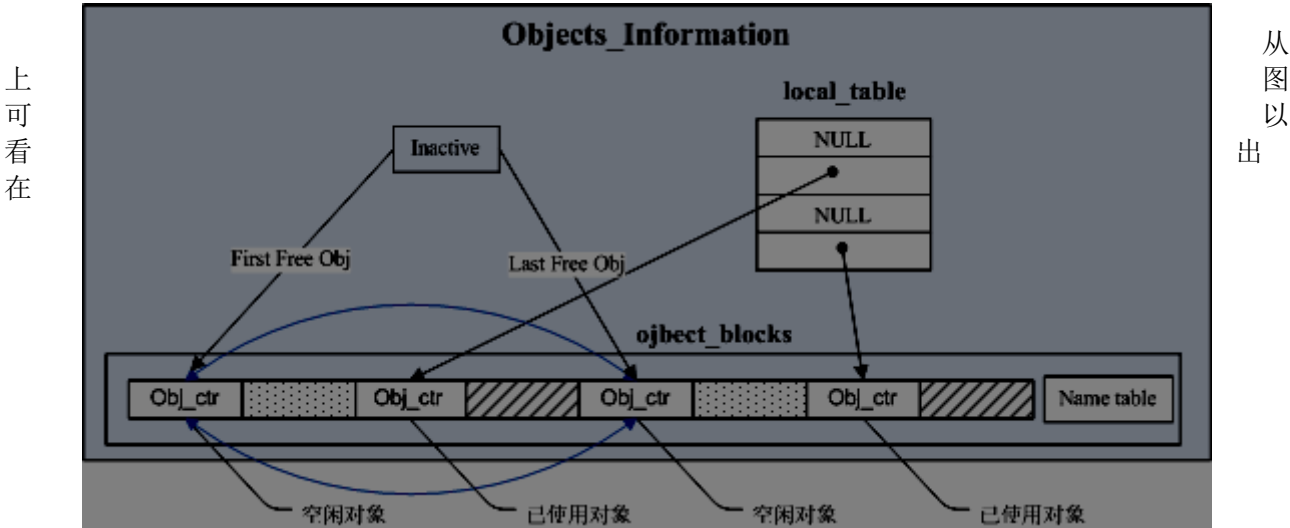
```
typedef struct {
    Objects_Control  Object;
    Watchdog_Control Ticker;
    Timer_Classes    the_class;
} Timer_Control;
```

而对每一类对象的描述和管理是通过 Objects_Information 这个结构体实现的。此结构体可以管理任何基于 Objects_Control 基类的继承类对象。基于 Objects_Control 基类的继承类必须把包含属于 Objects_Control 结构体的成员变量（且是第一个成员变量）。这样 Objects_Information 结构体就可以不用理会继承类的特性，只需要知道 Objects_Control 结构体的信息和其它辅助信息就可以管理各种继承此抽象类的继承类了。

下面我们来看看它是如何实现对不同类型的继承类对象进行管理的。Objects_Information 结构体包含的主要内容如下所示：

```
typedef struct {
Objects_APIs the_api;                //对象所属的 API 类
uint16_t the_class;                 //对象在所属 API 类下的属性类
Objects_Control **local_table;      //符合此类的本地对象列表
Objects_Name *name_table;           //本地对象名字的列表
Chain_Control Inactive;              //没有分配的对象列表
void **object_blocks;               //同类的对象列表所处内存块
#ifdef RTEMS_MULTIPROCESSING
Chain_Control *global_table;         //属于此类的全局对象列表
#endif
.....
} Objects_Information;
```

Objects_Information 结构体包含的信息很多，除了上述内容，还有对象的最大最小 ID、最大对象个数，此结构体是否可以自动扩展等域。下面是 Objects_Information 的结构图：



Objects_Information 结构体中，所有属于同一类的对象保存在一个连续地址空间的内存块 object_blocks 中，且此内存块的最后部分还保存了对象的名字列表。所有未被分配（即空闲的）对象由 Inactive 链表管理，所有已经被使用的对象可以由 local_table 列表（基于对象的 ID 中的 index）来查询。

因为 RTEMS 支持多种 API 类，每种 API 类可以提供不同的对象属性类，所以为了统一管理，RTEMS 用了一个基于 Objects_Information 结构体的可变大小的二维数组指针 Objects_Information_table 来描述所有对象群的信息，其定义如下：

```
Objects_Information **Objects_Information_table[OBJECTS_APIS_LAST + 1];
```

此数组指针是一个全局变量，它的一维表示 API 类，另一维表示属于此 API 类的对象类，由于不同 API 类包含的对象类个数不等，所以用指针表示会更加灵活和节省空间。

RTEMS 支持的 API 种类包括：

- OBJECTS_INTERNAL_API = 1,
- OBJECTS_CLASSIC_API = 2,
- OBJECTS_POSIX_API = 3,
- OBJECTS_ITRON_API = 4

而属于各种 API 类的对象种类定义如下：
属于 Internal API 类的对象种类

| | |
|--------------------------|------|
| OBJECTS_INTERNAL_THREADS | = 1, |
| OBJECTS_INTERNAL_MUTEXES | = 2 |

属于 RTEMS Classic API 类的对象种类:

| | |
|------------------------------|------|
| OBJECTS_RTEMS_TASKS | = 1, |
| OBJECTS_RTEMS_TIMERS | = 2, |
| OBJECTS_RTEMS_SEMAPHORES | = 3, |
| OBJECTS_RTEMS_MESSAGE_QUEUES | = 4, |
| OBJECTS_RTEMS_PARTITIONS | = 5, |
| OBJECTS_RTEMS_REGIONS | = 6, |
| OBJECTS_RTEMS_PORTS | = 7, |
| OBJECTS_RTEMS_PERIODS | = 8, |
| OBJECTS_RTEMS_EXTENSIONS | = 9 |
| OBJECTS_RTEMS_BARRIERS | = 10 |

属于 POSIX API 类的对象种类:

| | |
|-----------------------------------|-------|
| OBJECTS_POSIX_THREADS | = 1, |
| OBJECTS_POSIX_KEYS | = 2, |
| OBJECTS_POSIX_INTERRUPTS | = 3, |
| OBJECTS_POSIX_MESSAGE_QUEUE_FDS | = 4, |
| OBJECTS_POSIX_MESSAGE_QUEUES | = 5, |
| OBJECTS_POSIX_MUTEXES | = 6, |
| OBJECTS_POSIX_SEMAPHORES | = 7, |
| OBJECTS_POSIX_CONDITION_VARIABLES | = 8 |
| OBJECTS_POSIX_TIMERS | = 9, |
| OBJECTS_POSIX_BARRIERS | = 10, |
| OBJECTS_POSIX_SPINLOCKS | = 11, |
| OBJECTS_POSIX_RWLOCKS | = 12 |

属于 ITRON API 类的对象种类:

| | |
|-------------------------------------|------|
| OBJECTS_ITRON_TASKS | = 1, |
| OBJECTS_ITRON_EVENTFLAGS | = 2, |
| OBJECTS_ITRON_MAILBOXES | = 3, |
| OBJECTS_ITRON_MESSAGE_BUFFERS | = 4, |
| OBJECTS_ITRON_PORTS | = 5, |
| OBJECTS_ITRON_SEMAPHORES | = 6, |
| OBJECTS_ITRON_VARIABLE_MEMORY_POOLS | = 7, |
| OBJECTS_ITRON_FIXED_MEMORY_POOLS | = 8 |

从上面的定义可以 `_Objects_Information_table` 实际上描述了 RTEMS 支持的所有对象类的信息。也就是说给定一个对象类总可以在这个表中找到它所对应的 `Objects_Information` (就是属于这个类的所有对象信息), 而查找对象要用到的两个索引就是这个类属于哪种 API 类 (是 Internal、Classic、POSIX 还是 ITRON) 和这个 API 类中的哪种属性类 (是任务, 还是信号量等), 而这些信息都在对象的 ID 中。由于 `_Objects_Information_table` 列表中的 `Objects_Information` 管理的数据块 `object_blocks` 是一个连续地址的存储空间, 包含了所有被管理的特定类型的对象, 且 `Objects_Information_table` 知道每个特定类型对象的 `Objects_Control` 类型的基对象成员变量和单个特定类型对象所占的空间, 这样就可以把所有的对象管理起来: 首先通过 `_Objects_Information_table` 可找到对应类 (API 类+属性类) 的 `Objects_Information`, 而通过 `Objects_Information` 的 `local_table` 和 `global_table` 这两个 `Chain_Control` 就可以找到属于这个类的已经分配了的所有对象, 通过 `object_blocks` 可以访问属于这个类的没有分配的对象。为了方便地操作对象链表, RTEMS 提供了一系列双向列表操作函数来简化对对象的管理。

与对象节点/链表相关的操作支持函数

与对象节点和对象链表相关的操作包括对对象链表的初始化，对对象节点在对象链表中进行插入、添加、删除和获取等操作。这些操作是实现有效对象管理的基础。下面对相关函数实现进行介绍。

对象链表初始化

函数调用信息：

```
void _Chain_Initialize(  
    Chain_Control *the_chain,  
    void          *starting_address,  
    size_t        number_nodes,  
    size_t        node_size  
)
```

函数功能描述：

把从 starting_address 地址的开始 buffer 区域进行初始化，形成由地址上连续的 number_nodes 个 node 对象构成一个双向链表，每个对象的大小为 node_size，这些对象的结构体的第一个成员都是 Chain_Node 结构。同时把此链表的头一个节点赋值给链表变量 the_chain 的 first，把此链表的最后一个节点赋值给链表变量 the_chain 的 last。这里需要注意的是 _Chain_Head 宏完成了把 Chain_Control 结构体转变成 Chain_Node 结构体的工作，这样实际上使得对 Chain_Control 结构体的访问变成了对属于 Chain_Node 结构体的 first 成员变量的访问。_Chain_Head 宏的实现如下所示：

```
RTEMS_INLINE_ROUTINE Chain_Node *_Chain_Head(  
    Chain_Control *the_chain  
)  
{  
    return (Chain_Node *) the_chain;  
}
```

获取链表的第一个结点

函数调用信息：

```
Chain_Node *_Chain_Get(  
    Chain_Control *the_chain  
)
```

函数功能描述：

这个函数是从 the_chain 所表示的链获得第一个结点，且在进行此操作时屏蔽中断，完成此操作后再使能中断。与此类似的一个函数是 _Chain_Get_first_unprotected，不过它是内联函数，且在进行操作时不屏蔽中断。

对链表添加节点

函数调用信息：

```
void _Chain_Append(  
    Chain_Control *the_chain,  
    Chain_Node *node  
)
```


函数功能描述：

在链表的最后增加一个结点，且在进行此操作时屏蔽中断，完成此操作后再使能中断。与此类似的一个函数是_Chain_Append_unprotected，不过它是内联函数，且在进行操作时不屏蔽中断。

删除链表中的节点

函数调用信息：

```
void _Chain_Extract(  
Chain_Node *node  
)
```

在链表中删除给定结点，且在进行此操作时屏蔽中断，完成此操作后再使能中断。与此类似的一个函数是_Chain_Extract_unprotected，不过它是内联函数，且在进行操作时不屏蔽中断。

在链表中插入节点

函数调用信息：

```
void _Chain_Insert(  
Chain_Node *after_node,  
Chain_Node *node  
)
```

函数功能描述：

在链表的中的 after_node 后面添加一个结点且在进行此操作时屏蔽中断，完成此操作后再使能中断。与此类似的一个函数是_Chain_Insert_unprotected，不过它是内联函数，且在进行操作时不屏蔽中断。

对象管理的实现

使用对象的管理过程

对对象的管理主要体现在如何初始化_Objects_Information_table 全局变量和如何获得/删除一个对象上。对于一个具体的对象，例如要使用属于 CLASSIC API 类的 barrier 类的一个对象，那么大致的执行步骤如下：

1. 初始化：

对象初始化的来龙去脉如下所示：

```
rtems_initialize_executive_early()->_RTEMS_API_Initialize()->_Barrier_Manager_in  
itIALIZATION()->_Objects_Initialize_information()
```

_Barrier_Information 是属于 CLASSIC API 类的 barrier 类的 barrier 对象类信息表，在系统初始化时通过调用_Objects_Initialize_information 完成了对空闲 barrier 对象的空间分配。这一步是 rtems 内核负责，应用程序不用考虑。

2. 得到一个空闲对象：

```
rtems_barrier_create()->_Barrier_Allocate()->_Objects_Allocate()
```

调用 rtems_barrier_create 函数分配一个 barrier，这个函数会进一步调用_Barrier_Allocate 函数分配一个基于 Barrier_Control 结构体的对象，而_Barrier_Allocate 函数会进一步调用_Objects_Allocate(&Barrier_Information) 从_Barrier_Information 得到一个空闲的Barrier_Control 结构体对象返回。

3. 打开一个空闲对象：

```
rtems_barrier_create()->_Objects_Open()
```

得到空闲对象后，rtems_barrier_create 函数还要进一步通过执行

_Objects_Open(&_Barrier_Information, &the_barrier->Object, (Objects_Name) name) 函数，_Objects_Open 函数根据 the_barrier->Object.id 得到此具体对象 index，然后使得 _Barrier_Information 的 local_table[index] 指向这个被分配了的具体对象的 Object_Control 域，即这个对象的起始地址。到此这个 barrier 对象算是从空闲态变成了使用态。

4. 使用对象：

这是应用程序可以对这分配的对象进行特定的操作了。如果要释放这个分配了的对象，要执行如下两个步骤：

5. 关闭对象：

```
rtems_barrier_delete()->_Objects_Close()
```

调用 rtems_barrier_delete 函数，它会调用 _Objects_Close(&_Barrier_Information, &the_barrier->Object) 函数，_Objects_Close 函数根据 the_barrier->Object.id 得到此具体对象 index，然后使得 _Barrier_Information 的 local_table[index] 指向 NULL。

6. 释放对象：

```
rtems_barrier_delete()->_Barrier_Free()->_Objects_Free()
```

rtems_barrier_delete 函数会进一步调用 _Barrier_Free(the_barrier) 函数，_Barrier_Free 函数会调用 _Objects_Free(&_Barrier_Information, &the_barrier->Object) 来最终把此对象放到 _Barrier_Information.Inactive 空闲对象指针队列中，并更新相关域信息。

上述步骤1-6就是一个具体的对象从创建到最后释放的整个过程。

关键实现函数

下面是与对象管理相关的关键函数分析。

对象信息表初始化

函数调用信息：

```
void _Objects_Initialize_information(  
    Objects_Information *information,      对象类信息表  
    Objects_APIs        the_api,          对象的 API 类  
    uint32_t            the_class,        对象的属性类  
    uint32_t            maximum,          此类的最大对象个数  
    uint16_t            size,             对象控制块的大小  
    boolean             is_string,        对象的名字是否是字符串  
    uint32_t            maximum_name_length 对象名字的最大长度  
#if defined(RTEMS_MULTIPROCESSING)  
    ,  
    boolean             supports_global,  是否支持全局对象类  
    Objects_Thread_queue_Extract_callout extract threadq extract callout 函数指针  
#endif  
)
```

函数功能描述：

此函数主要完成对属于 (the_api +the_class) 类的对象类信息表初始化，分配空闲对象空间。下面是主要的步骤：

1. 初始化 information 部分成员

2. 根据 the_api 和 the_class 设置 information 在 _Objects_Information_table 中的入口
3. _Objects_Information_table[the_api][the_class] = information;
4. 设置 information 中存储对象的大小 size
5. 根据传入的 maximum 确定是否是 auto_extend, 如果传入的是 OBJECTS_UNLIMITED_OBJECTS 则设为 auto_extend
6. 设置 information 中的对象个数 allocation_size 为 maximum
7. 把 null_local_table 的地址(实际上是 NULL)传给 local_table
8. 计算最小和最大的对象 ID
9. 调用 _Objects_Extend_information 函数分配空闲对象空间 object_blocks
10. 如果支持多处理, 则根据 nodes (这里就是指 CPU 个数) 给 information 的 global_table 分配空间。

□某□对象分配或□展□象需要的内存空□

函数调用信息:

| | |
|--|-----------------|
| _Objects_Extend_information(Objects_Information *information | 要分配/扩展空间的对象类信息表 |
|) | |

函数功能描述:

此函数主要完成对象类信息表分配/扩展空闲对象内存空间。下面是主要的步骤:

1. 判断 information->maximum 的值, 如果 information->maximum 为 0, 表示是 _Objects_Initialize_information 调用, 执行第一次分配操作, 则表示当前空闲对象已经分配完了, 是要扩展空闲对象空间;
2. 根据 information->auto_extend 的值, 调用 _Workspace_Allocate 函数或 _Workspace_Allocate_or_fatal_error 函数分配一个区域, 用于给 information 的 object_blocks、inactive_per_block、name_table、local_table 域所需要的空间, 并初始化 information 的 maximum、maximum_id 域; 分配的空间大小如下

```
void *objects[block_count];

uint32_t inactive_count[block_count];

Objects_Name *name_table[block_count];

Objects_Control *local_table[maximum];
```

3. 如果是扩展操作, 需要把 information 中以前的 object_blocks、inactive_per_block、name_table、local_table 域的内容拷贝到新的对应的空间中;
4. 根据 information->auto_extend 的值, 再调用 _Workspace_Allocate_or_fatal_error 函数分配另外一个 block 区域, 用于 information 包含的多个空闲对象的名字空间和结构体空间;
5. 通过链表操作初始化处于此 block 区域的空闲对象 (建立), 把链表头放入 information->Inactive 中。这样通过查询 information->Inactive 就可以很容易得到空闲对象。

取对象指针

函数调用信息:

```
Objects_Control *_Objects_Get(  
Objects_Information *information, 属于此类的对象类信息表  
Objects_Id id,                  要查找对象的对象 ID  
Objects_Locations *location      返回值, 指明对象的位置  
)
```

函数功能描述:

此函数主要根据对象 ID 在对象类信息表 information 中查找符合此 ID 的对象指针。下面是主要的步骤:

1. 首先通过 ID 获得 index, 然后在 local_table 里找到相应对象。需要注意的是, 在寻找过程中, 执行了 _Thread_Disable_dispatch 函数, 这样找到对象并返回后, 在接下来的高层函数中是需要执行 _Thread_Enable_dispatch 函数, 来允许线程切换。
2. 如果支持多处理器, 则调用 _Objects_MP_Is_remote, 这个函数会在先通过 ID 获得它所在的 CPU 对应的 node 值, 然后在 global_table 中找到对应 node 的 Chain_Control, 在这个链表中根据 ID 找到对应的对象。

函数

- ✎ _Objects_Get_by_index 函数的作用和 _Objects_Get 函数类似, 只不过它是直接通过 index 在 local_table 里获取对象。
- ✎ _Objects_Get_next 函数会调用 _Objects_Get 函数找到给定 ID 的下一个 ID 对象(最接近的)。
- ✎ _Objects_Allocate 函数是从对应类的 information 的 Inactive 里分配一个对象空间, 如果这个类的 information 的 auto_extend 置位, 还会调用 _Objects_Extend_information 函数来更新这个 information。
- ✎ _Objects_Allocate_by_index 函数根据 index 在 Inactive 里分配一个对象空间。
- ✎ _Objects_Free 函数把对象变成空闲状态, 可这样可以被再次分配; 根据对象 ID 得到对象所处的内存块号 block, 把对应的 information 的 inactive_per_block[block]域和 inactive 域加一; 如果空闲的对象个数大于一个 object_block 可分配对象个数的 1.5 倍, 则执行 _Objects_Shrink_information 函数来压缩 information 管理的内存块。
- ✎ _Objects_Shrink_information 函数根据情况压缩 information 中的具体对象所占用的空间。需要注意的是, 对于 information 管理的每个内存块 object_block, 只有满足 information->inactive_per_block[block] == information->allocation_size 条件, 即此内存块的所有对象都是空闲的情况下, 才释放对应的内存空间。
- ✎ _Objects_Close 函数是把保存在 information 中的 local_table (即 Local Object Pointer Table) 对应项的 object_control 指针清空, 且把 object_control 结构体的 name 域清空。
- ✎ _Objects_Open 函数把已经分配的 object 的 object control 指针赋值给对应的 information 中的 local_table 项, 并对 object_control 结构体的 name 域赋值。

小节

第四章 初始化过程

概述

在操作系统课本中, 很少涉及到操作系统启动要做的事情, 以及操作系统是如何开始正常运行等问

题。导致读者虽然知道进程/线程管理、内存管理、同步互斥等抽象出来的操作系统原理，但对于让操作系统和上述机制开始正常工作的启动和初始化过程不太清楚。就好比虽然吃到了免淘洗的精制白米饭，感到饭也很好吃，但在把谷子变成糙米再变成精米的过程中，很多有营养的成分已经丢失了。下面各节将讲解计算机硬件的启动、引导加载程序、BIOS、加载 RTEMS 操作系统的 OS boot loader 和 RTEMS 操作系统的详细初始化启动过程。通过这部分的描述，让读者能够掌握如下内容：

- ❏ 计算机硬件加电后做的工作
- ❏ 引导加载程序如何加载 RTEMS 操作系统
- ❏ RTEMS 操作系统如何初始化硬件（CPU/外设）
- ❏ RTEMS 操作系统如何初始化自身核心组件
- ❏ RTEMS 操作系统如何进入多任务模式
- ❏ RTEMS 操作系统如何把控制权交给应用程序的

这样读者对操作系统的启动不再有陌生感，对后面章节的分析也会打下一个扎实的基础。这部分内容避免涉及部分硬件内容，我们将以应用较广泛的 PC386 计算机系统作为这部分内容的参考硬件平台。

计算机系统启动过程

相对于其他嵌入式 RISC CPU(如 ARM、MIPS 等)，PC386 是一种复杂的 32 bit CISC CPU，且它还需要向下兼容 8086 等老的 CPU 模式，导致在软件初始化方面的工作相对较多。这里我们不会对 PC386 的硬件架构进行详述，而只是把与 RTEMS 启动相关的部分进行分析和描述。

当计算机加电后，一般不直接执行操作系统，而是执行引导加载程序。简单地说，引导加载程序就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立系统的内存空间映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。最终引导加载程序把操作系统内核映像加载到 RAM 中，并将系统控制权传递给它。

对于绝大多数计算机系统而言，操作系统等软件都是存放在磁盘（硬盘/软盘）、光盘、EPROM、ROM、Flash 等可在掉电后继续保存数据的存储介质上。但计算机启动后，CPU 一开始会到一个特定的地址开始执行指令，这个特定的地址存放了系统软件（不仅是操作系统，还可能是引导加载程序等）。

引导加载程序是系统加电后运行的第一段软件代码。对于 PC386 的体系结构而言，PC 机中的引导加载程序由 BIOS(其本质是一个固化的软件)和位于软盘/硬盘引导扇区中的 OS Boot Loader（比如，LILO 和 GRUB 等）一起组成。而在嵌入式系统中，通常并没有像 BIOS 那样的固化软件（注，有的嵌入式 CPU 也会内嵌一段短小的启动程序），因此整个系统的加载启动任务就完全由 OS Boot Loader 来完成。比如在一个基于 ARM7TDMI CPU 的嵌入式系统中，系统在上电或复位时通常都从地址 0x00000000 处开始执行，而在这个地址处安排的通常就是 OS Boot Loader 程序。

以 PC386 为例，计算机加电后，CPU 从物理地址 0xFFFFF0（由初始化的 CS: EIP 确定，CS 和 IP 的值和也是 0xF000 和 0xFFFF0）开始执行。在 0xFFFFF0 这里只是存放了一条跳转指令，通过跳转指令跳到 BIOS（Basic Input Output System，即基本输入/输出系统）例行程序起始点。

注意：

BIOS 实际上是被固化在计算机 ROM（只读存储器）芯片上的一个特殊的软件，为上层软件提供最低级的、最直接的硬件控制与支持。更形象地说，BIOS 就是 PC 计算机硬件与上层软件程序之间的一个“桥梁”，负责访问和控制硬件。BIOS 的主要功能包括：

- ❏ 自检及初始化：开机后 BIOS 最先被启动，然后它会对电脑的硬件设备进行完全彻底的检验和测试。
- ❏ 程序服务：BIOS 直接与计算机的 I/O 设备打交道，通过特定的数据端口发出命令，传送或接收各种外部设备的数据，实现软件程序对硬件的操作。
- ❏ 设定中断：开机时，BIOS 会告诉 CPU 各硬件设备的中断号，当用户发出使用某个设备的指令后，CPU 就根据中断号使用相应的硬件完成工作，再根据中断号跳回原来的工作。

BIOS 做完计算机硬件自检和初始化后，会选择一个启动设备（可能是软盘、硬盘、光盘等），并且读取该设备的第一扇区（即启动扇区）到内存一个特定的地址，然后 CPU 控制权会转移到那个地址继续执行。至此 BIOS 的初始化工作做完了，进一步的工作交给了 OS Boot Loader。在第二章中，我们已经制作出了一个软盘镜像，这个软盘镜像的第一个扇区的内容就是 OS boot loader - grub (GRand Unified Bootloader) 的最初始的部分。我们这里假定采用 GRUB 来启动 RTEMS，下面将对 GRUB 的执行过程做一个简单介绍，看看 RTEMS 启动前，GRUB 要做什么事情，给 RTEMS 提供一个什么样的环境。

BIOS 把 GRUB 所在的第一个扇区（512 Byte）读到系统的 RAM 中的 0x7c00 处，然后将控制权交给 GRUB。GRUB 的主要运行任务就是将内核映像从软盘/硬盘上读到 RAM 中，然后跳转到内核的入口点去运行，也即开始启动操作系统。

以第二章讲解的“hello world” RTEMS 为例，来看看 GRUB 是如何加载 RTEMS 并把控制权给 RTEMS 的。首先 GRUB 会根据其加载配置文件 menu.lst 找到 RTEMS（即 hello.exe）的磁盘地址，并把它到内存的特定地址。这个特定地址是如何确定的呢？把 CPU 控制权转移到哪个地址呢？

hello.exe 是一个 GCC 生成 ELF 格式的执行文件，通过如下命令，我们可以看到一些有意思的信息。

```
$ i386-rtems-readelf -a hello.exe
```

ELF 头:

```

Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:                               ELF32
Data:                               2's complement, little endian
Version:                             1 (current)
OS/ABI:                             UNIX - System V
ABI Version:                         0
Type:                               EXEC (可执行文件)
Machine:                             Intel 80386
Version:                             0x1
入口点地址:                         0x10000c
程序头起点:                         52 (bytes into file)
Start of section headers:            185048 (bytes into file)
标志:                               0x0
本头的大小:                         52 (字节)
程序头大小:                         32 (字节)
程序头数量:                         1
节头大小:                           40 (字节)
节头数量:                           7
字符串表索引节头:                   6

```

节头:

| [Nr] | Name | Type | Addr | Off | Size | ES | Flg | Lk | Inf | Al |
|------|-------|----------|----------|--------|--------|----|-----|----|-----|----|
| [0] | | NULL | 00000000 | 000000 | 000000 | 00 | | 0 | 0 | 0 |
| [1] | .text | PROGBITS | 00100000 | 000060 | 0297cc | 00 | WAX | 0 | 0 | 32 |
| [2] | .init | PROGBITS | 001297cc | 02982c | 00000d | 00 | AX | 0 | 0 | 1 |

| | | | | | | |
|----------------|----------|---------------------------|----|---|---|----|
| [3] .fini | PROGBITS | 001297d9 029839 000008 00 | AX | 0 | 0 | 1 |
| [4] .data | PROGBITS | 001297e1 029841 003a6f 00 | WA | 0 | 0 | 32 |
| [5] .bss | NOBITS | 0012d250 02d2b0 003490 00 | WA | 0 | 0 | 32 |
| [6] .shstrtab | STRTAB | 00000000 02d2b0 000028 00 | | 0 | 0 | 1 |

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

0 (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

程序头:

| Type | Offset | VirtAddr | PhysAddr | FileSiz | MemSiz | Flg | Align |
|------|----------|------------|------------|---------|---------|-----|-------|
| LOAD | 0x000060 | 0x00100000 | 0x00100000 | 0x2d250 | 0x306e0 | RWE | 0x20 |

Section to Segment mapping:

段节...

00 .text .init .fini .data .bss

There is no dynamic section in this file.

该文件中没有重定位信息。

There are no unwind sections in this file.

No version information found in this file.

grub 包含解析 ELF 格式执行文件的功能，它首先分析 hello.exe 的 ELF 文件头，看到这个 ELF 文件设定的加载地址为 0x00100000，所以很自然第就把 ELF 文件的代码段、数据段等按照 ELF 文件头的信息放置在指定的内存位置。

grub 完成 RTEMS 内容数据拷贝后，就需要转移 CPU 控制权了。在上面的输出中，有“入口点地址：0x10000c”这么一行，这就是 CPU 要转移的执行地址。

实验:

准备工作:

编译 RTEMS 例子 hello.exe，用 build_image.sh 生成 fd.img，启动调试模式 QEMU（要加上 -S -s 参数），启动 *i386-rtems-gdb hello.nxe*，建立与调试模式 QEMU 的联系。

实际操作:

在 i386-rtems-gdb 下执行:

break start //在地址0x10000c，即 pc386BSP 中的 start.S 的87行设置断点

continue //这时就让 QEMU 继续执行，等一下，就会停到0x10000c 地址处

x /10i 0x10000c //反汇编0x10000c 处的代码，可以看到与 start.S 的代码一致

到这里，GRUB 的工作也完成了，下面就进入到 RTEMS 的初始化阶段。到这一阶段为止，RTEMS 在内存中布局如下所示:



系统初始化第一阶段

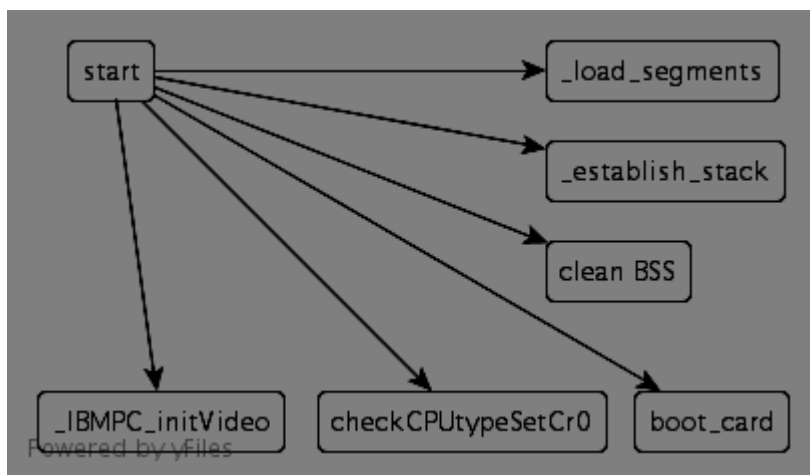
主要工作

首先我们考察一下 RTEMS 在最开始阶段的工作，这部分代码一般都是用汇编写的，属于 BSP 的一部分。RTEMS 当然想尽早进入基于 C 代码的初始化第二阶段，但在进入之前，必须要有一个可靠的运行环境。这就要通过初始化第一阶段的汇编代码完成了。总的来说，这一阶段的主要工作包括：

- ❏ 屏蔽中断
- ❏ 初始化 CPU 工作模式
- ❏ 建立内核堆栈
- ❏ 对 BSS 段清零
- ❏ 建立基本的内存布局

执行流程

由于我们分析的参考硬件是 PC386，所以可以根据上节的分析找到最开始的代码位于 `rtems/c/src/lib/libbsp/i386/pc386/start/start.S` 中。其中的 `start` 位置是 GNU default entry point，即 RTEMS 的启动位置。其函数调用图如下所示：



大致的执行流程包括：

1. 屏蔽中断
2. 读取 grub 传递来的 `multiboot_info` 信息（包括 `flags`, `mem_lower`, `mem_upper`），存放到 `_boot_multiboot_info` 结构中，
3. 跳转到 `_load_segments` 处，加载与 PC 硬件相关的全局描述符和中断描述符
4. 转跳到 `_establish_stack` 处，建立 RTEMS 内核栈空间
5. 清除 BSS
6. 调用 `_IBMPC_initVideo` 函数进行视频显示器初始化
7. 调用 `checkCPUtypeSetCr0` 函数查找 CPU 类型，设置 CR0 寄存器

8. 设定调用参数，调用第一个 C 函数 boot_card

因为一开始中断处理、内存管理等操作系统的基本机制还没有建立，所以在一开始就屏蔽中断是操作系统启动后首先要做的事情，这样可以避免潜在的硬件中断行为，避免不可预知的情况。

接下来 RTEMS 拷贝通过 GRUB 传来的 multiboot_info 信息，通过这部分信息可以知道这个计算机中的物理内存大小。对 multiboot_info 信息可以在如下代码中看到：

```
rtems/c/src/lib/libbsp/i386/pc386/startup/bspstart.c
函数 void bsp_pretasking_hook(void)
第116行的代码：
bsp_mem_size = _boot_multiboot_info.mem_upper * 1024;
```

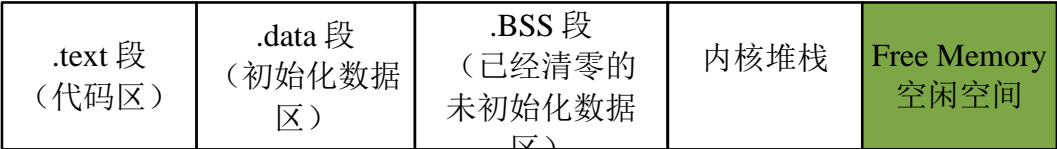
然后就是要设置内存管理模式。在 PC386上可同时提供段式和页式内存管理，但 RTEMS 只使用了段式内存管理，即没有采用虚存管理方式，而是简单的平面式的内存管理方式。在_load_segments 函数中，通过加载全局描述符和中断描述符设置 CPU 的段模式。_load_segments 函数位于 rtems/c/src/lib/libbsp/i386/pc386/startup/ldsegs.S。ldsegs.S 文件中定义了全局描述符表，相关的段描述及对应的操作。当加载 GDT 后，选择子 CS 指向代码段（包含所有物理内存空间），SS = DS = ES = FS = GS 指向数据段（包含所有物理内存空间）。这样，CPU 的地址空间处于平坦的段保护模式。接着把中断向量表加载的地址和长度加载到 IDT，这里只是定义了256个全为空的中断描述符，所以并不是的中断初始化。随着 RTEMS 的进一步运行，还会完成具体的中断初始化。详细分析内容在有关中断管理的章节讲述。

不知读者是否注意到，到目前为止 RTEMS 的执行过程中没有函数调用，只是用了 jmp 转跳指令。这是由于还没有设置内核堆栈的原因。为了执行函数调用，需要尽快设置中断。所以，接下来就要设定内核堆栈的起始地址和空间，RTEMS 把内核堆栈的地址设置在 RTEMS 的 BSS 段的结尾，堆栈段的大小为0X1000。

接下来是对 RTEMS 中没有初始化的 BSS 段的内容进行清零操作。在内核和应用编程中，对于没有初始化的变量，我们一般会认为操作系统已经我们清零了。所以这是一个操作系统的基本工作。

再接下来又是 CPU 硬件相关处理，先通过各种小手段判断 CPU 的类型（386/486/486+ 等）。在 QEMU 模拟器中执行，RTEMS 判断出是486+级别的 CPU，然后就设置 CR0寄存器为0x50033，简单地说，就是让 CPU 进入了没有页管理的保护模式。然后检查此 CPU 是否与硬件浮点协处理器。经过分析，QEMU 模拟的是 Intel Pentium II (Klamath) 处理器。

到此，第一阶段的初始化工作结束了，最主要是设定了CPU的工作模式，建立了堆栈，掌握了可用内存大小，清零了 BSS。这一切为进入以 C 语言为主的第二个初始化阶段建立了可靠的保障。最后，设定调用参数，调用第一个 C 函数 boot_card。到这一阶段为止，RTEMS 在内存中布局如下所示：



起始地址 0X100000结束地址 bsp_mem_size-0X100000

系统初始化第二阶段

主要工作

由于与初始化第一阶段的帮助，可以进入基于 C 语言的第二个初始化阶段。这一阶段的工作主要为初始化内核组件和驱动程序等做准备，与 BSP 联系紧密，其主要的工作包括：

- ❏ 初始化 rtems_cpu_table 结构体的 Cpu_table 全局变量
- ❏ 初始化 rtems_configuration_table 结构体的 Configuration 全局变量
- ❏ 设置 RTEMS workspace 区域
- ❏ 初始化中断和异常管理
- ❏ 初始化 PCI BIOS interface

下面我们先分析其中的关键数据结构，并看看当达到目标，具体需要做哪些事情。

关键数据结构

由于 RTEMS 的各种组件、驱动和应用都是通过某些关键的配置表确定的。这里面最重要的两个是 CPU 配置表—Cpu_table 和 RTEMS 配置表 Configuration。

与 CPU 相关的关键数据结构

CPU 配置表的结构与 CPU 硬件相关函数和数据，这使得 RTEMS 中与硬件无关的部分能够与与硬件相关的 BSP 部分建立有效的合作关系。面向 PC386 的 CPU 配置表结构体定义如下：

```
typedef struct {
    void      (*pretasking_hook)( void );
    void      (*predriver_hook)( void );
    void      (*idle_task)( void );
    boolean    do_zero_of_workspace;
    unsigned32 idle_task_stack_size;
    unsigned32 interrupt_stack_size;
    unsigned32 extra_mpci_receive_server_stack;
    void *     (*stack_allocate_hook)( unsigned32 );
    void      (*stack_free_hook)( void* );
    /* end of fields required on all CPUs */

    unsigned32 interrupt_segment;
    void      *interrupt_vector_table;
} rtems_cpu_table;
```

面向 PC386 的 CPU 配置表主要提供了一组了 BSP 包中的关键函数指针，下面对这些函数指针做一个说明：

- **pretasking_hook**：此函数指针指向 PC386 BSP 提供的 bsp_pretasking_hook 函数，pretasking_hook 函数在驱动程序初始化前调用，主要完成空闲堆空间的建立（free memory heap），为提供 malloc 和 free 函数做准备，初始化 libio 和文件系统，设置 libc（这里特指 newlibc）的可重入（reentrant）属性。需要注意的是，在执行此函数的时候，不能创建系统任务，需要屏蔽中断。
- **predriver_hook**：在 PC386 BSP 中，此函数指针指向 NULL。它在设备驱动和 MPC1 初始化之前调用。这设置为 NULL 的原因是 pretasking_hook 已经把设备驱动和 MPC1 初始化前的准备工作做完了。
- **postdriver_hook**：在 PC386 BSP 中，此函数指针指向 bsp_postdriver_hook 函数，这是一个与具体硬件无关的函数。它在设备驱动和 MPC1 初始化之后被立即调用。主要工作是打开/dev/console 设备文件。这样 RTEMS 就可以进行字符输入输出了。需要注意的是，在执行此函数的时候，RTEMS 已经初始化完毕了，但还处于屏蔽中断和禁止任务调用的阶段。
- **idle_task**：在 PC386 BSP 中，此函数指针指向 NULL。它是一个可选择的项，如果不为 NULL，表示采用应用程序指定的一个 idle task 运行；如果为 NULL，表示采用 RTEMS default IDLE task 作为 idle task。
- **do_zero_of_workspace**：表示 workspace 内存区是否清零了。

- `idle_task_stack_size`: idle task 的堆栈大小, 此值不能小于 `MINIMUM_STACK_SIZE`。
- `interrupt_stack_size`: 表示在处理中断时候的中断上下文堆栈大小, 此值不能小于 `MINIMUM_STACK_SIZE`。
- `extra_mpci_receive_server_stack`: 分配给 RTEMS MPCIE receive server 任务的堆栈空间大小, 这个变量的作用是为当 MPCIE 任务需要比较大的堆栈空间, 需要特别指定。在 PC386 BSP 中, 此值为零。
- `stack_allocate_hook`: 在 PC386 BSP 中, 此函数指针指向 NULL。这是应用程序提供的分配任务堆栈的钩子函数。如果此钩子函数指针不为 NULL, 则 `stack_free_hook` 钩子函数指针也不能为 NULL。
- `stack_free_hook`: 在 PC386 BSP 中, 此函数指针指向 NULL。这是应用程序提供的释放任务堆栈的钩子函数。如果此钩子函数指针不为 NULL, 则 `stack_allocate_hook` 钩子函数指针也不能为 NULL。
- `interrupt_segment`: 这是 PC386 所特有的, 用来存储指向中断描述符表 (IDT) 的段寄存器。
- `interrupt_vector_table`: 这是 PC386 所特有的, 这是与指向中断描述符表的段寄存器对应的中断描述符的起始地址。有关 IDT 的具体描述, 可参考与中断管理相关的章节。

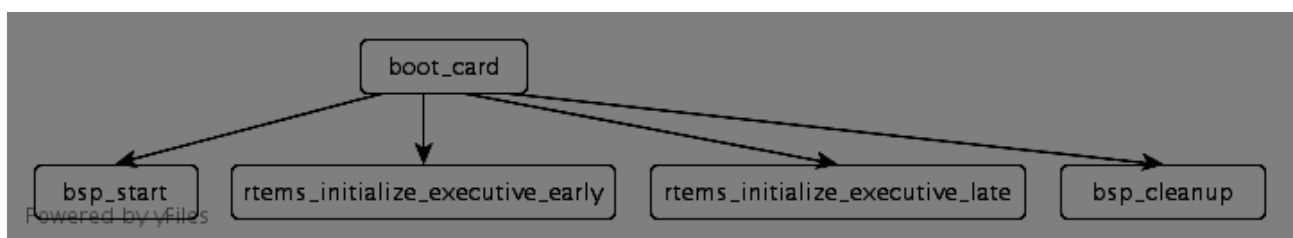
与 RTEMS 系统功能相关的关键数据结构

RTEMS 配置表描述了 RTEMS 的系统级功能和配置。关于 RTEMS 配置表的结构体 `rtems_configuration_table`, 主要用于对 RTEMS 内核和应用程序的裁减, 如最大的任务数、使用的 API 系统服务组件、初始化任务配置、驱动程序配置、用户扩展配置和多处理配置。此结构在有关 RTEMS 配置相关的章节有详细的描述。第二阶段的一个重要工作就是要根据 RTEMS 配置表对涉及到的 RTEMS 各个层次的功能组件进行初始化。

在这其中的两个重要全局变量是 RTEMS 配置表 `Configuration` 和 `BSP_Configuration`, 这二者有一定区别, RTEMS 在运行过程中主要使用 `BSP_Configuration`。`Configuration` 描述了应用程序指定的 RTEMS 配置表, 而 `BSP_Configuration` 先拷贝了 `Configuration`, 然后根据 BSP 的情况进行了部分修改, 主要是修改了对 CLASSIC API 系统服务配置 (由 `BSP_RTEMS_Configuration` 指定) 和 POSIX API 系统服务配置 (由 `BSP_POSIX_Configuration` 指定), 更新 RTEMS 的 workspace 起始地址和空间大小等。

执行流程

从 `boot_card` 函数开始, 大致的调用关系如下图所示:



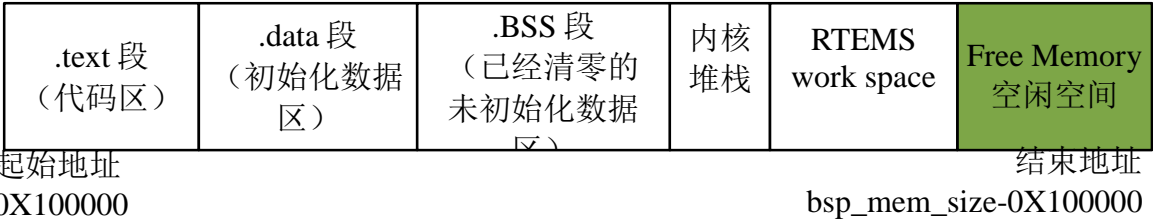
`boot_card` 首先要更新 CPU 配置表和 RTEMS 配置表, 因为下面的初始化工作很大程度上都是基于这两个配置表进行的。然后调用 `bsp_start` 函数, 在 PC386 中, 实际上此函数是 PC386 BSP 的 `bsp_start_default` 函数, 它主要完成如下与硬件相关的工作:

- ✎ 计算 1ms 时间额指令 loop 值
- ✎ 进一步更新 RTEMS 的 CPU 配置表
- ✎ 根据 RTEMS 配置表, 指定 RTEMS 的 workspace 的起始地址并分配空间
- ✎ 进一步初始化中断和异常管理 (与 PC386 相关)
- ✎ 初始化 PCI BIOS Interface

问题: (何时给 `BSP_Configuration.work_space_size` 赋值的????)

有屏蔽中断和异常的初始化分析请参考与中断管理相关的章节。这里需要注意的是只有初始化 PCI

BIOS Interface 正常完成后，基于 PCI 总线的一些外设驱动程序才能正常工作。完成 bsp_start 函数后，RTEMS 的 workspace 的起始地址和空间就确定了。执行完此函数后的 RTEMS 内存布局如下所示：



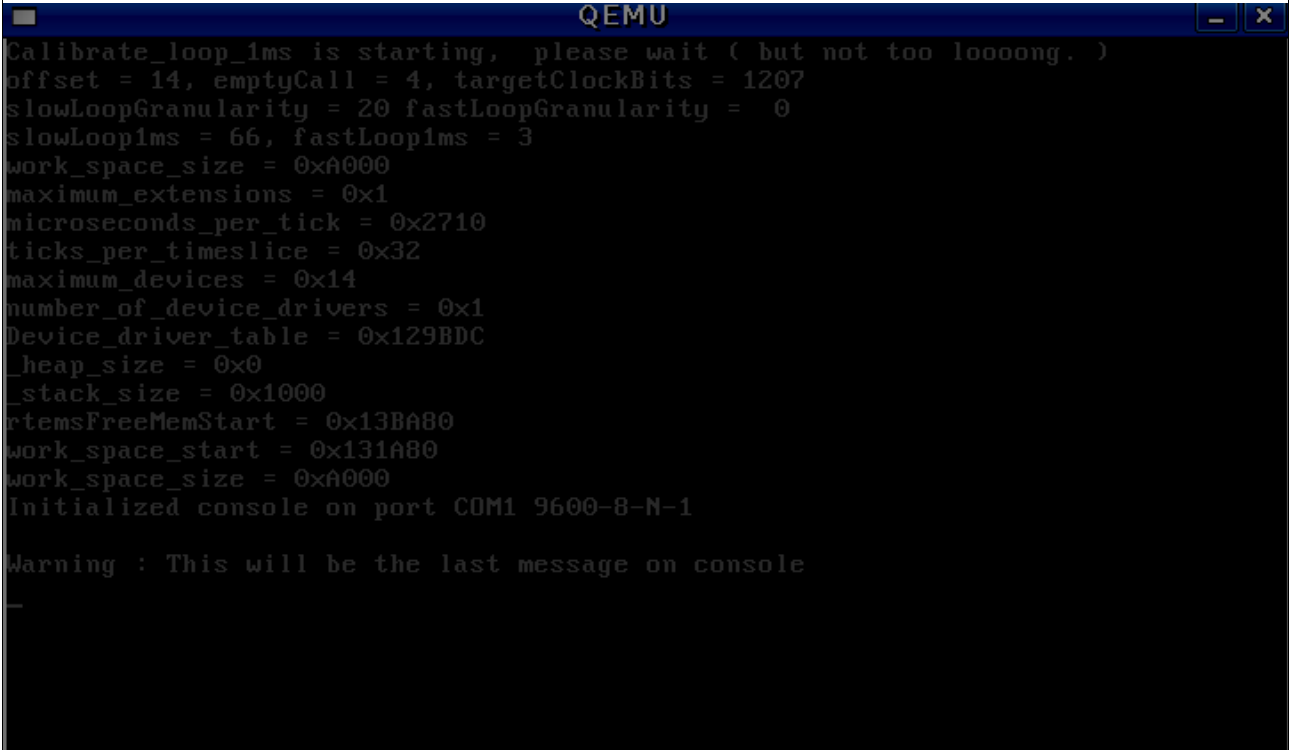
实验：

修改 rtems/c/src/lib/libbsp/i386/pc386/timer/timer.c 在第45行下增加两行，可以看到计算1ms 的 loop 值的结果：

```
#define DEBUG_CALIBRATE 1
#define DEBUG 1
```

修改 rtems/c/src/lib/libbsp/i386/pc386/ 把第223行和第241行注释掉，这样可以让配置信息显示出来。

然后重新编译 RTEMS 和其中的 helloworld 例子，则通过 QEMU 模拟执行，可得到如下显示：



```
QEMU
Calibrate_loop_1ms is starting, please wait ( but not too loooong. )
offset = 14, emptyCall = 4, targetClockBits = 1207
slowLoopGranularity = 20 fastLoopGranularity = 0
slowLoop1ms = 66, fastLoop1ms = 3
work_space_size = 0xA000
maximum_extensions = 0x1
microseconds_per_tick = 0x2710
ticks_per_timeslice = 0x32
maximum_devices = 0x14
number_of_device_drivers = 0x1
Device_driver_table = 0x129BDC
_heap_size = 0x0
_stack_size = 0x1000
rtemsFreeMemStart = 0x13BA80
work_space_start = 0x131A80
work_space_size = 0xA000
Initialized console on port COM1 9600-8-N-1

Warning : This will be the last message on console
-
```

执行完 bsp_start 函数后，就要进入系统初始化的第三个阶段，开始执行 rtems_initialize_executive_early 函数了。

系统初始化第三阶段

主要工作

系统初始化的第三个阶段的工作量相对前面两个阶段要大很多。其最终目标是完成多任务切换，并切换到用户提供的任务，完成整个应用系统的正常运行。这部分主要完成的工作是：

- ❏ 初始化 RTEMS 核心层和系统服务层等的功能组件
- ❏ 初始化驱动程序
- ❏ 进行多任务初始化，切换到用户提供的线程/任务执行

在介绍第三阶段的初始化工作前，首先需要对 RTEMS 的初始化管理器组件有一个全面的了解。

初始化管理器组件介绍

当板级支持包（BSP）完成硬件级的基本初始化工作后，把控制权交给初始化管理器，初始化管理器的工作主要是负责启动和关闭 RTEMS。启动 RTEMS 包括创建并且启动所有的配置好的初始化任务，并且初始化 RTEMS 系统使用到的设备驱动程序。初始化管理器对系统初始化后，将把 CPU 控制移交到应用程序。但应用程序退出后，初始化管理器完成对 RTEMS 的关闭。

为了完成上述工作，在 RTEMS 的初始化管理器中，管理了三种用于初始化管理的任务：系统初始化任务、用户初始化任务、空闲任务（idle task）。

系统初始化任务

系统初始化任务其实没有进入 RTEMS 的任务调度中，它不是一个严格意义上的可调度任务。为了理解方便，RTEMS 开始执行到把控制权交给用户初始化任务过程认为是系统初始化任务执行过程。系统初始化任务负责初始化所有 RTEMS 功能组件、设备驱动；在多处理器系统中，系统初始化任务还负责初始化多处理器通信接口层（MPCI）；创建空闲任务、用户初始化任务。在多处理器系统中，系统初始化任务在初始化设备驱动后，还创建多处理服务器（MPCI Receive Server）任务，该任务设定多处理器通信接口层，核查多处理器系统一致性，而且处理来自远端结点所有的请求。

系统初始化任务需要有充足的栈空间，否则无法初始化所有的设备驱动。在初始化多处理器系统时，需要初始化处理器配置表，该表包含一个成员变量，该成员变量允许动态增加分派给应用或 BSP 的栈空间的尺寸。

用户初始化任务

用户初始化任务定义在用户初始化任务表中，由应用程序具体实现，由系统初始化任务创建并且启动。由于用户初始化任务接受 RTEMS 任务管理器的调度，所以为了保证在其它应用任务启动前初始化任务能先启动，必须分配给它一个较高的优先级。。

标准的用户初始化任务创建并且启动配置好的应用程序，也可以创建应用任务需要的对象。一般来说，对于设计良好的初始化任务代码，在初始化结束的时候，必须将自己删除，并且释放自己占用的资源。初始化任务也可以转换成普通的应用任务。在做这样的转换的时候必须修改任务的优先级。

例如，在 `rtems/testsuites/samples/base_sp` 目录中 `init.c` 的函数 `Init` 是一个典型的例子，`Init` 函数的内容就是用户初始化任务的主要工作，它首先创建并启动了另外一个任务，然后把自身删除。其主要实现如下：

```
rtems_task Init(  
    rtems_task_argument argument  
)  
{  
    .....  
    status = rtems_task_create( task_name, 1, RTEMS_MINIMUM_STACK_SIZE,
```

```

        RTEMS_INTERRUPT_LEVEL(0), RTEMS_DEFAULT_ATTRIBUTES, &tid );

status = rtems_task_start( tid, Application_task, ARGUMENT );

status = rtems_task_delete( RTEMS_SELF );

}

```

那么用户初始化任务是如何开始运行呢？

在 cpukit/rtems/include/rtems/rtems/tasks.h 中定义了初始化任务表这样一个数据结构，它定义了初始化任务的名字、栈的大小、初始的优先级、属性、入口点、工作模式以及任务参数。

```

typedef struct {
    rtems_name      name;          /* task name */
    size_t          stack_size;    /* task stack size */
    rtems_task_priority initial_priority; /* task priority */
    rtems_attribute attribute_set; /* task attributes */
    rtems_task_entry entry_point;  /* task entry point */
    rtems_mode       mode_set;     /* task initial mode */
    rtems_task_argument argument;  /* task argument */
} rtems_initialization_tasks_table;

```

在 cpukit/sapi/include/confdefs.h 中定义了系统的初始化任务表，它只包含一个初始化任务。

```

rtems_initialization_tasks_table Initialization_tasks[] = {
{ CONFIGURE_INIT_TASK_NAME,
  CONFIGURE_INIT_TASK_STACK_SIZE,
  CONFIGURE_INIT_TASK_PRIORITY,
  CONFIGURE_INIT_TASK_ATTRIBUTES,
  CONFIGURE_INIT_TASK_ENTRY_POINT,
  CONFIGURE_INIT_TASK_INITIAL_MODES,
  CONFIGURE_INIT_TASK_ARGUMENTS
}
};

```

其中宏 CONFIGURE_INIT_TASK_ENTRY_POINT 的定义如下如下：

```

#ifndef CONFIGURE_INIT_TASK_ENTRY_POINT
#define CONFIGURE_INIT_TASK_ENTRY_POINT Init
#endif

```

也就是说如果用户没有定义 CONFIGURE_INIT_TASK_ENTRY_POINT，那么系统会把 Init 定义成为默认的初始化任务的入口点。在 RTEMS 的 CLASSIC API 的测试用例中可以看到，它的初始化任务的函数名都是 Init。另外，CONFIGURE_INIT_TASK_PRIORITY 的值为1，这也就是我们前面提到的初始化任务的优先级是很高的，这样才能保证它可以优先执行。这个初始化任务表是 rtems_api_configuration_table 的一个成员，而 API 的配置表，又是整个系统的配表的一部分，这样在系统启动时，初始化任务表就可以传入系统。在系统启动时会调用 _RTEMS_tasks_Manager_initialization 来初始化任务管理器，这个函数会调用 _API_extensions_Add 把 _RTEMS_tasks_API_extensions 加到 _API_extensions_List 上，_RTEMS_tasks_API_extensions 的定义如下：

```

API_extensions_Control _RTEMS_tasks_API_extensions = {
{ NULL, NULL },
NULL, /* predriver */
_RTEMS_tasks_Initialize_user_tasks, /* postdriver */
_RTEMS_tasks_Post_switch_extension /* post switch */
};

```

其中，_RTEMS_tasks_Initialize_user_tasks 定义在 cpukit/rtems/src/taskinitusers.c 上，它会遍历任务初始化表，根据前面提到的表中定义的任务的各种属性来创建并启动每一个初始化任务。

而 _API_extensions_List 中包含的 API 扩展结构的 postdriver 任务会在

`_API_extensions_Run_postdriver` 得到执行，这个函数会在系统初始化中执行，这样我们就可以理解用户的初始化任务是如何得到运行的。

空闲任务

空闲任务由系统初始化任务创建并且启动。只有当没有其他的任务准备好运行的时候，空闲任务是一个优先级最低的任务（该任务本身就是一个消耗 CPU 时间的无限循环）。当其他任务就绪时，空闲任务使用处理器的权力就会被抢占。空闲任务一般的工作是让 CPU “休息”，当有中断产生或其他任务就绪，则会转移 CPU 控制权。我们可以看看 PC386 BSP 的空闲任务的工作是什么，下面是其实现内容：

```
void _CPU_Thread_Idle_body ()
{
    while(1) {
        asm volatile ("hlt");
    }
}
```

是否很简单？实际上就是一个 CPU halt 的无限循环。

初始化管理器运行失口的情况

当初始化管理器碰到运行错误时，将会调用 `rtems_fatal_error_occurred` 函数，系统将停止运行。常见的初始化错误包括：

- ❏ 没有被提供 CPU 配置表或 RTEMS 配置表。
- ❏ 如果 RTEMS 配置表中提供的 RTEMS 在 workspace 中入口地址是 NULL 或者没有按照四字节的格式对齐（也就是说入口地址不是4的整数倍）。
- ❏ 如果 RTEMS 系统的 workspace 空间太小而不够设定初始化系统
- ❏ 如果被指定的中断栈大小太小
- ❏ 如果是多处理器系统但是多处理器配置表格的结点数目无效（不在1和 `maximum_nodes` 之间）
- ❏ 如果配置成多处理器系统但是没有分配多处理器通信接口层
- ❏ 如果没有配置用户初始化任务
- ❏ 如果有一个用户初始化任务但不能够被成功地创建或启动

主要操作

初始化 RTEMS

初始化管理器的主要操作体现在 `rtems_initialize_executive` 函数的实现机制中。在某种的情况下（RTEMS for PC386 就就是这种情况），如果不使用该机制，也可以采用 `rtems_initialize_executive_early` 函数和 `rtems_initialize_executive_late` 函数的实现机制来初始化，这样在 `rtems_initialize_executive_early` 函数和 `rtems_initialize_executive_late` 函数之间还可以插入一些其他工作。`rtems_initialize_executive_early` 函数完成在启动多任务处理之前的初始化工作，`rtems_initialize_executive_late` 函数用于启动多任务。在 RTEMS 中只能使用一种方案初始化系统，即不能重复初始化。

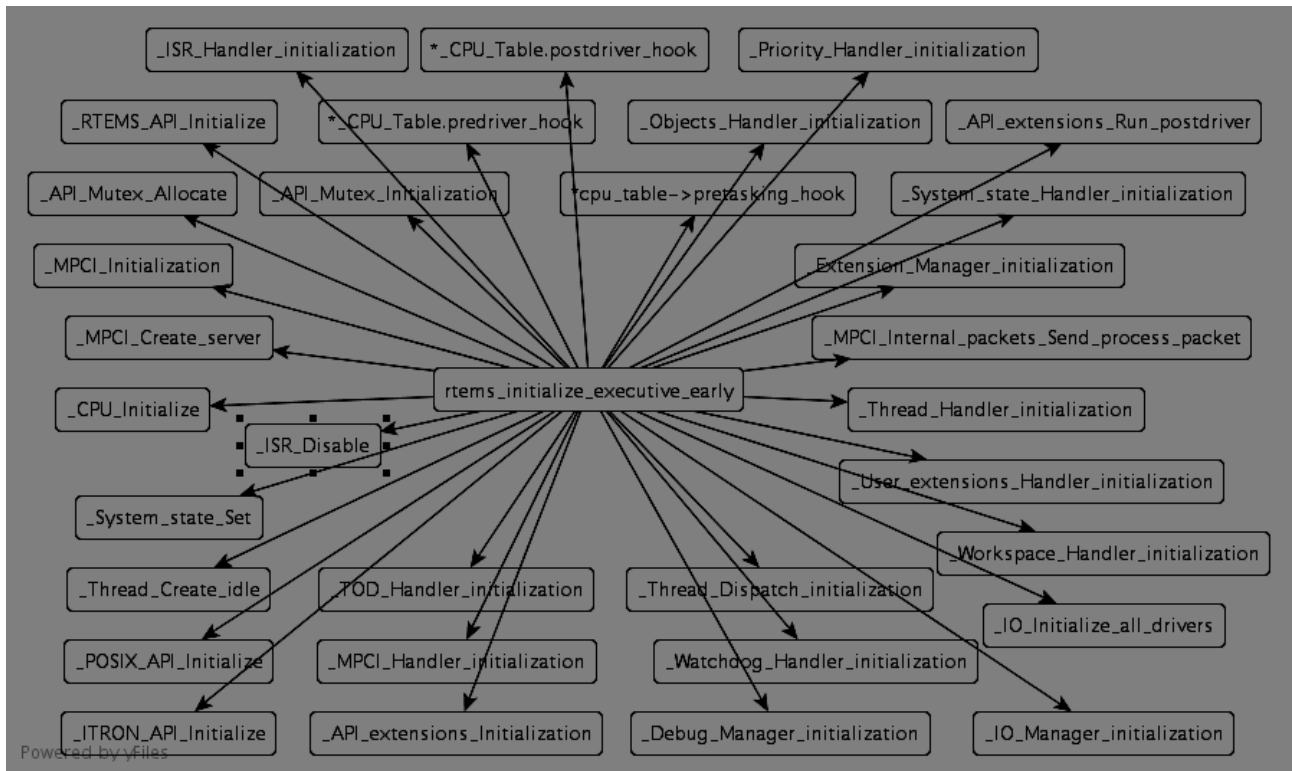
关闭 RTEMS

在应用任务结束的时候，将调用 `rtems_shutdown_executive` 函数结束多任务处理并且将控制权交还给 BSP。BSP 将执行善后工作，然后停机或重启。

执行流程

了解了 RTEMS 的初始化管理器组件的基本情况，再分析系统初始化第三阶段的执行流程就容易多了。在 RTEMS for PC386, 初始化管理器主要通过执行 `rtems_initialize_executive_early` 函数和 `rtems_initialize_executive_late` 函数来完成最后阶段的初始化，并启动多任务，并切换到用户初始化任务。至此完成整个 RTEMS 系统初始化过程。

我们先来看看 `rtems_initialize_executive_early` 函数，下面是它的调用关系图：



`rtems_initialize_executive_early` 函数一共调用了32个函数, 这也算是一个复杂的过程。不过结合 XX 章讲解的 RTEMS 功能组件的介绍, 可以将这些函数归类以方便理解。这32个函数都重要, `rtems_initialize_executive_early` 函数主要完成的工作是:

- ❏ 屏蔽中断: 在没有初始化完成前是不能使能中断的, 这通过 `_ISR_Disable` 函数完成;
- ❏ 设置系统状态为 `SYSTEM_STATE_BEFORE_INITIALIZATION`, 表明进入初始化状态;
- ❏ 初始化核心层的功能组件, 主要包括 API extension (API 扩展) 组件、workspace handler 组件、user extensions (用户扩展) handler 组件、ISR (中断服务例程) handler 组件、OBJECT (对象) handler 组件、Priority (优先级) handler 组件、Watchdog (看门狗) Handler 组件、TOD (时间) Handler 组件、Thread (线程) handler 组件、MPCI Handler 组件。
- ❏ 初始化系统服务层的功能组件: 主要包括属于 RTEMS CLASSIC API 和 SAPI 类的 task manager (任务管理器) 组件、Timer manager (定时器管理器) 组件、signal manager (信号管理器) 组件、event manager (事件管理器) 组件、message queue manager (消息队列管理器) 组件、semaphore manager (信号量管理器) 组件、partition manager (分区管理器) 组件、region manger (区域管理器) 组件、Dual ported memory manager (双端口内存管理器) 组件、barrier manager (屏障管理器) 组件、Debug_Manager 组件、Extension_Manager 组件、I/O manager (I/O 管理器) 组件; 如果定义了 RTEMS_POSIX_API 宏, 还要初始化属于 POSIX API 的各类组件;

如果定义了 RTEMS_ITRON_API 宏，还要初始化属于 ITRON API 的各类组件；

- ❏ 设置系统状态为 SYSTEM_STATE_BEFORE_MULTITASKING，表示功能组件初始化完毕，处于进入多任务状态的前夜；
- ❏ 初始化所有的驱动程序：这分成四个步骤，在具体执行初始化驱动程序之前，需要执行钩子函数 (*_CPU_Table.predriver_hook)，然后执行 _IO_Initialize_all_drivers() 来完成具体的驱动程序初始化过程，如果与多处理支持，还要执行 MPCIE 驱动程序，并进行初步的 MPCIE 验证，最后执行 API extension (API 扩展) 组件的 API_extensions_Run_postdriver 函数来创建用户初始化任务，并执行钩子函数 (*_CPU_Table.postdriver_hook)。

当 rtems_initialize_executive_early 函数完成繁重的工作后，接下来的 rtems_initialize_executive_late 函数的工作就轻松多了。它首先把当前的系统状态设置为 SYSTEM_STATE_BEGIN_MULTITASKING，然后通过执行 Thread_Start_multitasking 函数把当前系统状态设置为 SYSTEM_STATE_UP，这表明 RTEMS 进入多任务状态了，最后执行 Context_Switch 函数把 CPU 控制权转移到用户初始化任务去执行。至此整个 RTEMS 的系统初始化工作结束了。

小节

第五章 线程管理与调度

概述

RTEMS 是一个以线程（也称任务）为基本调度单位的实时操作系统。调度算法是基于优先级的抢占式线程调度，支持256个线程优先级。0级优先级代表最高优先级，主要用于 RTEMS 内部线程，255级代表最低优先级，是空闲（idle）线程的优先级，用户线程的优先级在1~254之间。RTEMS 支持创建同优先级线程，相同优先级的线程采用时间片轮转调度；调度器寻找下一个最高优先级就绪线程的时间是恒定的 ($O(1)$)，这也是实时性得到保障的一个关键机制。为了增强对用户应用需求的可扩展性，RTEMS 还实现了基于线程的任务扩展机制，可在创建任务、上下文切换或者是删除任务等系统事件发生时插入用户指定的函数（也称为钩子函数，hook 函数），完成用户设定的特殊功能。对于 RTEMS 这样的基于抢占式和优先级策略调度的操作系统来说，为保证其硬实时的特性，需要选择一个合适的调度算法。静态调度算法——速率单调调度（Rate Monotonic Scheduling，简称 RMS）是为周期性任务解决多任务调度的一种很好的方法。为此 RTEMS 实现了 RMS 调度算法来满足硬实时的要求。

RTEMS 在系统服务层支持 CLASSIC、POSIX、ITRON 三种 API，RTEMS 并没有从底层独立实现这三种 API，这三种 API 实际上都是在 RTEMS 的“super core”核心层上的 thread handler 组件进行封装得到的。RTEMS 中 threadXXX 文件中包含的 Thread_XXX 函数就是线程管理的“super core”层，即核心层 thread handler 组件的主要实现。而 RTEMS 中的 taskXXX 文件包含的 rtems_task_XXX 函数是封装了 thread handler 组件的系统服务层任务管理组件 task manager 的主要实现。本章将先介绍核心层的 thread handler 组件与调度的实现，然后介绍属于 RTEMS CLASSIC API 的系统服务层的 task manager 的接口和封装实现；接着进一步介绍任务扩展机制；最后对速率单调调度算法在 RTEMS 上的实现进行了分析。

核心层的 thread handler 组件

为了更好的理解 RTEMS 的线程管理与调度机制，关键是要理解 RTEMS 核心层的线程相关的数据结构和对数据结构的操作。描述一个线程的状态是通过线程控制块结构体来实现的。线程在从“出生”到“死亡”的运行过程中，会经历活跃的“运行态”，无所事事的“阻塞态”，到最后死亡的“退出态”，这都是通过 RTEMS 中的线程管理机制和调度机制来完成的。在线程的管理中各种类型的线程队列（如等待某个特定资源的线程形成的队列）是实现管理的关键。RTEMS 的调度相对通用操作系统的调度比较简单，主要是基于线程的优先级进行线程调度。除了对付优先级反转问题需要在运行时要改变线程的优先级外，其它情况下不会对线程的优先级进行改变。下面将首先对 RTEMS 核心层的线程管理与调度的关键部分进行分析。

RTEMS 核心层的 thread handler 组件提供了 Thread_Control 结构体和以此配套的一组基本的线程管理数据结构和管理函数，涉及线程初始化、线程启动、线程暂停、线程恢复执行、线程切换、线程属性管理等。不同的高层 API 接口实现只是对这些结构和函数进行封装，这样线程管理就可以分为两层，对 RTEMS 支持不同 API 接口扩展会很方便。

线程控制块

线程控制块 (Thread Control Block, 简称 TCB) 是 RTEMS 中 thread handler 组件对线程的静态基本情况和动态运行变化过程的描述。它包含了与管理线程运行所需要的所有信息集合，是线程存在的唯一标志。线程的创建主要体现在为该线程生成一个 TCB；线程的终止主要体现在回收它的 TCB；而线程的组织管理主要是通过对 TCB 的组织管理来实现的。那么 TCB 具体包含什么信息？如何被组织管理的呢？

RTEMS 的 TCB 定义在 \$RTEMS_ROOT/cpukit/score/include/rtems/score/thread.h 中，具体定义如下：

```
struct Thread_Control_struct {
    Objects_Control      Object;
    States_Control       current_state;
    Priority_Control      current_priority;
    Priority_Control      real_priority;
    uint32_t             resource_count;
    Thread_Wait_information Wait;
    Watchdog_Control     Timer;
#ifdef RTEMS_MULTIPROCESSING
    MP_packet_Prefix     *receive_packet;
#endif
    uint32_t             suspend_count;
    boolean              is_global;
    boolean              do_post_task_switch_extension;
    boolean              is_preemptible;
    void                 *rtems_ada_self;
    uint32_t             cpu_time_budget;
    Thread_CPU_budget_algorithms budget_algorithm;
    Thread_CPU_budget_algorithm_callout budget_callout;
    uint32_t             ticks_executed;
    Chain_Control        *ready;
    Priority_Information  Priority_map;
    Thread_Start_information Start;
    Context_Control      Registers;
#ifdef ( CPU_HARDWARE_FP == TRUE ) || ( CPU_SOFTWARE_FP == TRUE )
    void                 *fp_context;
#endif
    struct _reent        *libc_reent;
    void                 *API_Extensions[ THREAD_API_LAST + 1 ];
    void                 **extensions;
    rtems_task_variable_t *task_variables;
};
```

这是一个很重要的数据结构，它的各成员的含义在注释中已经说得比较清楚了，这里再把比较重要的或者难以理解的几个成员解释一下：

- ❧ Object 是这个线程作为对象而存在时对应的对象管理结构，在“总体结构和系统构成”一章中已经提到了 RTEMS 的对象管理机制，TCB 结构也是作为对象而存为的。
- ❧ real_priority 和 current_priority 分别是它的基础优先级和当前运行的优先级，

resource_count 代表线程获取到的资源个数，这些为了实现优先级继承算法和优先级天花板算法而设计的，具体的内容会在分析同步互斥管理机制的章节中描述。

- ✎ cpu_time_budget 是分配这个线程的时间片，而 budget_algorithm 是用来管理这个时间的方法，该成员会根据不同的调度策略设定，budget_callout 就是如果 budget_algorithm 被指定为时间片用完后需要执行一个指定的函数，那么这个成员就是那个需要执行的函数（具体可查看 _Thread_Tickle_timeslice() 函数的实现）。
- ✎ ready 是线程所在优先级的就绪线程等待队列
- ✎ Priority_map 是线程的优先级信息，本章后面的优先级管理相关小节中有进一步分析。
- ✎ Start 定义了一个线程的启动信息，包括需要执行的函数及其参数、堆栈的位置、调度参数等，这些都是在创建线程时初始化的。
- ✎ Registers 是这个线程执行的执行上下文，线程切换会用到这个结构，后面会详细分析。

TCB 中的成员变量主要包含：线程的名称、ID、当前的优先级、当前状态、执行模式、相关记事本位置、TCB 用户扩展指针的位置、调度控制结构以及阻塞线程所需控制结构。把上述成员变量进行分类，可以看出 TCB 有三大类信息：

- ✎ **线程标识信息：**线程的对象 ID
- ✎ **CPU 状态信息保存区：**主要包含了 CPU 的寄存器信息和堆栈指针，体现在 TCB 的成员变量 Registers、fp_context 和 Start。
- ✎ **线程控制信息：**主要包括了调度和状态信息、线程间通信信息和线程链接信息。调度和状态信息用于操作系统调度线程并占用处理机使用，体现在 TCB 的成员变量 current_state、current_priority、real_priority、cpu_time_budget、budget_algorithm、budget_callout、is_preemptible、Priority_map 等；RTEM 根据这些信息对线程进行调度，线程间通信信息主要为支持线程间通信的各种标识、信号量、事件等，对应 TCB 的成员变量 Timer、API_Extensions、Wait、resource_count 中；线程链接信息体现了不同状态和类型的线程的信息，这样线程可以连接到一个线程队列中，或连接到相关的其他线程的 PCB，这体现在 TCB 的成员变量 ready、Wait 中。

应用层任务发出的系统核心层提供的系统调用接口来改变 TCB 内部的成员变量。同时需要注意的是 TCB 是 RTEMS 中唯一的能经由用户扩展例程被应用访问的 RTEMS 内部的数据结构。

通常在嵌入式系统中，为了节省内存，内核所需要支持的最大线程数量是静态配置好的，因为嵌入式系统一般都针对确定的应用，所以开发人员应该对系统划分出的线程数量很明确。在 RTEMS 中是通过下面的宏定义完成这个配置的：

```
#define CONFIGURE_MAXIMUM_TASKS 3
```

这样在系统初始化的时候就可以根据线程的个数来为线程控制块分配分存，构成一个空闲的线程控制块链，创建线程的会从该链上取出空闲控制块进行初始化，在删除线程的时候将该控制块返回到空闲的链表上。

当发生线程调度的时候，线程的执行上下文信息储存在 TCB 的 Registers、fp_context 中。当线程重新获得处理器控制权时，它从 TCB 的 Registers、fp_context 中恢复执行上下文信息。

线程调度机制

在 RTEMS 中，对响应时间要求很严格，每个线程运行时间很短，基本上都采用可抢占的调度方式，相应地采用较小的调度单位（如线程）来实现快速上下文切换。为了实现更加灵活的调度手段，RTEMS 提供四种机制允许用户对线程调度进行干预：

- ✎ **用户制定线程优先级（并且可动态修改）：**在 RTEMS 中，线程的优先级决定了线程获取处理器的次序。RTEMS 支持的优先级从 0 到 255，共 256 级。线程的优先级使用数据类型 rtems_task_priority 来存储。用户可以在线程创建的时候为线程分配一个优先级，同时也允许用户在线程运行时改变线程的优先级。当一个线程进入 ready 状态链时，它前面应该是具有相同优先级的线程。该规则提供一个轮转的调度方式，轮转调度指在一组具有相等的优先级线程中，调度程序会根据他们准备好的次序（FIFO）依次调度入 CPU 中运行。虽然用户可以动态调整线程的优先级，但是程序员必须记住：RTEMS 调度程序总是选择准备态中的具有最高优先级线程进入 CPU 运

行。

- ❏ **线程抢占式控制**（该线程是否会被抢占）：用户可以通过改变线程的抢占模式标识 (RTEMS_PREEMPT_MASK) 影响调度的策略。如果一个线程的抢占位被置为无效 (RTEMS_NO_PREEMPT)，那么线程在运行的时候不会被抢占，除非线程结束或者被阻塞。在这种情况下，就算是有较高的优先级的线程就绪，线程也不会被抢占。需要注意的是，抢占位设置对于等待调度的线程是无效的，只有对于正在执行的线程才有用。
- ❏ **线程时间片控制**：基于优先级的时分复用，也被称为轮转调度。这是一种优化的优先级调度算法的附加算法。和抢占模式位一样，时分复用也是通过状态位标识的。时分复用使用时分复用模式标识 (RTEMS_TIMESLICE_MASK)。如果时分复用状态位通过 RTEMS_TIMESLICE 使能，那么 RTEMS 将会限制线程在处理器中的运行时间，如果到达该线程的运行时间片，它将会被其他线程取代。如果当前优先级就绪队列中没有其他的线程了，那么 RTEMS 将会增加当前线程的时间片并且让它继续执行。程序员必须注意，当一个较高的优先级线程就绪，它将会立刻抢占正在时间片轮转的线程，即使线程还没有用光它的整个时间片。
- ❏ **主动放弃处理器**：程序员可使用 `rtems_task_wake_after` 函数主动放弃处理器中低优先级的。在调用该指令时使用参数 `RTEMS_YIELD_PROCESSOR` 来表示时间间隔。该参数线程放弃处理器，进入系统的就绪队列中。如果没有其他同等优先级的线程就绪，那么该线程将不会放弃对处理器的占用。

上面的几方法都为定制线程属性提供了有力的手段，以满足不同场合的不同调度需要。每种干预手段彼此独立，它们的效果也有差异。调度程序对线程调度总是按照优先级，抢占模式和时间片 (timeslicing) 为调度次序。程序员在使用 timeslicing 和轮转的时候需要注意这两种调度中如果来了更高优先级的线程，让会产生抢占式调度，而不会顾及时间片因素。这样的实现机制即保证了调度的灵活性，也保证了在调度的过程中不会产生冲突。

线程的状态

线程的执行过程是一个动态，在 RTEMS 调度机制的管理下，在不同的时间段，线程会处于不同的状态。在 RTEMS 系统中，线程必须处于五种允许的线程状态之一。其状态变化情况在第一章中讲述的任务状态变化图基本一致，只是在细节上有所扩充，并在状态变化的起因上更加细致。这些状态是：

- ❏ **运行(Running)**：就绪线程如果被调度入处理器运行，就称它为运行状态，就绪线程可能因为下面的原因进入运行状态：
 - ❏ 该线程是优先级最高的就绪线程。
 - ❏ 正在运行的线程自己放弃了对处理器的控制，就绪队列中具有最高优先级的线程如果优先级相同，它们会轮流占用处理器。
 - ❏ 正在运行的线程将其强占位指为有效。此时如果就绪队列中有线程的优先级更高，就会将调高优先级的线程占用处理器。
 - ❏ 正在运行的线程降低它自己的优先级，这样优先级的线程会强占处理器运行。
 - ❏ 正在运行的线程提高，如果上升后新的优先级比正在运行线程的优先级高，高优先级的线程就有可能被调度到处理器进入运行态。
- ❏ **就绪(Ready)**：一个线程做好了一切准备工作，但是处理器被其他线程占用，则此时此线程处于就绪状态。一个线程由于下列任何一个的条件式进入就绪状态：
 - ❏ 接收到 `rtems_task_resume` 命令，线程从原来的挂起 (Suspended) 态唤醒。
 - ❏ 正在运行的线程调用 `rtems_message_queue_send` 或 `rtems_message_queue_broadcast` 或 `rtems_message_queue_urgent` 向阻塞的线程发送消息，而这个消息正好是阻塞线程所等待的。
 - ❏ 正在运行线程使用 `rtems_event_send` 函数向阻塞线程发送对应的事件。
 - ❏ 正在运行的线程调用了一个释放信号量的函数 `rtems_semaphore_release`，该信号量是被阻塞的线程正在等候的。
 - ❏ 被 `rtems_task_wake_after` 函数休眠的线程的休眠时间片到达，线程被唤醒，进入就绪队列。
 - ❏ 被 `rtems_task_wake_when` 函数阻塞的线程时间片到。
 - ❏ 正在运行的线程调用内存释放函数 `rtems_region_return_segment`，因为内存不足而阻塞的线程变成就绪。
 - ❏ 单调周期线程的执行时间到达，线程就绪。
 - ❏ 正在运行的线程使用 `rtems_task_restart` 函数唤醒阻塞线程。

- ☞ 在运行的线程，如果它的抢占位有效，可能会使用上面的指令将更高优先级的线程唤醒就绪，这样它自己就会被调度出处理器，变成就绪状态。同样 ISR 也会让正在运行的线程被中断，处理器转而执行中断服务例程。
- ☞ 挂起 (Suspended)：一个正在运行的线程可能挂起它本身，或者一个处于就绪态的线程被系统的其他线程挂起。在具体实现上，正在运行的线程可以通过 `rtems_task_suspend` 函数将自己挂起。同样一个线程能使用 `rtems_task_suspend` 函数挂起另外一个线程。
- ☞ 阻塞 (Blocked)：当一个线程由于要等待某个事件而无法运行，就称为它处于阻塞状态。一个线程进入阻塞状态可能由下面的原因所引起：
 - ☞ 正在运行的线程使用 `rtems_message_queue_receive` 函数接受消息队列中的信息，但是消息队列暂时为空，那么线程就会进入阻塞态。
 - ☞ 运行的线程等待的某个事件没有发生。
 - ☞ 运行的线程无法得到需要的信号量。
 - ☞ 运行的线程使用 `rtems_task_wake_after` 函数让自己阻塞。如果函数传入的参数代表的时间间隔是零，线程将进入就绪状态。
 - ☞ 运行的线程调用 `rtems_task_wake_when` 函数将自己阻塞，进入阻塞状态，直到睡眠结束
 - ☞ 运行的线程调用 `rtems_region_get_segment` 函数请求一个内存区段，但是暂时没有满足要求的区域，会进入阻塞状态。
 - ☞ 运行的线程调用 `rtems_rate_monotonic_period` 函数后必须等到指定的时间才能执行。
- ☞ 创建 (New)：在 RTEMS 中，线程创建后自动进入创建状态，等待变成就绪状态。
- ☞ 退出 (Exit/Dormant)：严格地说，系统中并没有一个线程真的处于这个状态，即使是正在被函数 `rtems_task_delete` 删除线程，它的状态也是 `STATES_TRANSIENT` (瞬时状态)，但是删除以后，线程就不再拥有有效的 TCB 和线程 ID 标识，因此别的线程也无法与该线程联系，此时线程就处于 Exit 状态。

与第一章的任务状态相比，在 RTEMS 中增加了一个 Suspended 状态，且此状态和 Blocked 状态不同。Suspended 状态的变化是因为需要把一个线程挂起或者恢复，而不是产生 Blocked 状态那样是因为需要某种条件得到满足而间接地引起线程的阻塞。

下面是 RTEMS 中对 Suspend 状态的定义：

```
#define STATES_SUSPENDED 0x00002
```

下面是 RTEMS 中对 Blocked 状态的定义：

```
#define STATES_LOCALLY_BLOCKED ( STATES_WAITING_FOR_BUFFER      |\
    STATES_WAITING_FOR_SEGMENT      |\
    STATES_WAITING_FOR_MESSAGE      |\
    STATES_WAITING_FOR_SEMAPHORE    |\
    STATES_WAITING_FOR_MUTEX        |\
    STATES_WAITING_FOR_CONDITION_VARIABLE |\
    STATES_WAITING_FOR_JOIN_AT_EXIT  |\
    STATES_WAITING_FOR_SIGNAL        |\
    STATES_WAITING_FOR_BARRIER      |\
    STATES_WAITING_FOR_RWLOCK        )

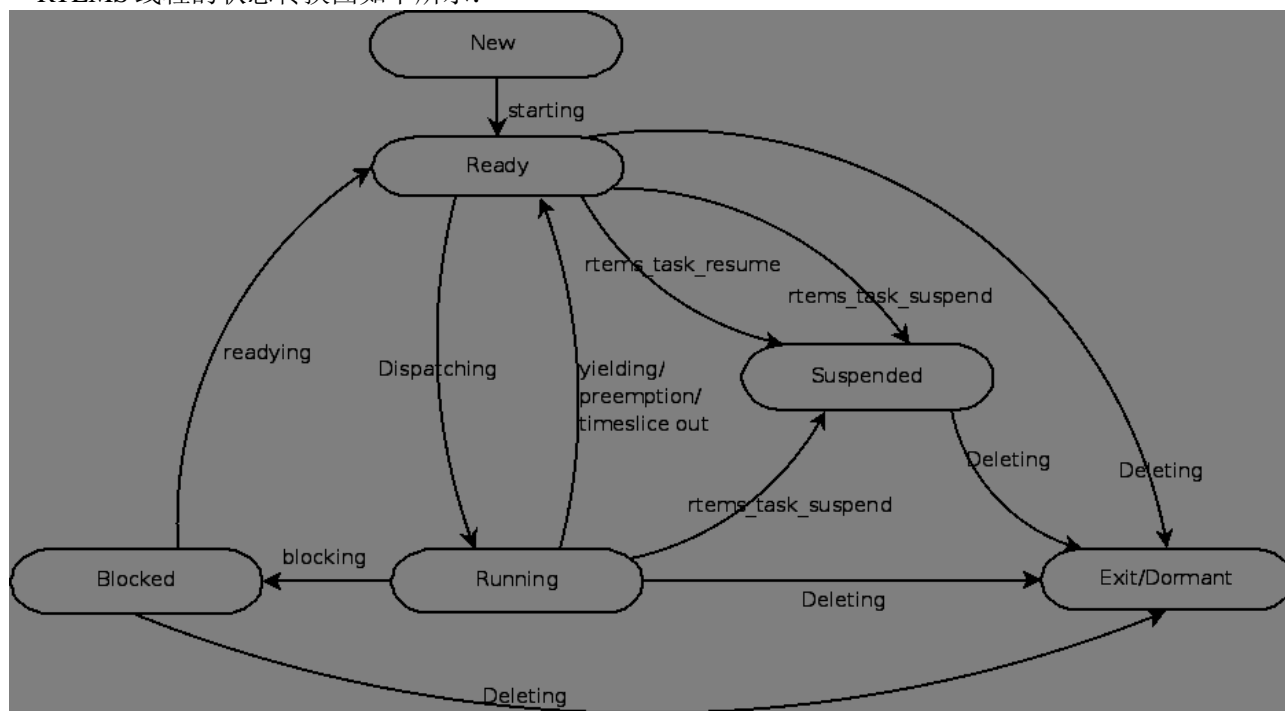
/** This macro corresponds to a task waiting which is blocked on
 * a thread queue. */
#define STATES_WAITING_ON_THREAD_QUEUE \
    ( STATES_LOCALLY_BLOCKED      |\
      STATES_WAITING_FOR_RPC_REPLY )

/** This macro corresponds to a task waiting which is blocked. */
#define STATES_BLOCKED ( STATES_DELAYING      |\
    STATES_WAITING_FOR_TIME      |\
    STATES_WAITING_FOR_PERIOD    |\
    STATES_WAITING_FOR_EVENT     |\
    STATES_WAITING_ON_THREAD_QUEUE |\
    STATES_INTERRUPTIBLE_BY_SIGNAL )
```

从上面的定义可以看出，RTEMS 线程的 Blocked 状态是许多具体状态的集合，这些具体的状态是用 bit 位

来表示的。也就是说 RTEMS 线程等待上述的某个条件都会变为 Blocked 状态。但反过来看，如果某一等待条件满足，这时相应的等待标志位会被清除，线程不一定会变为就绪态，因为这个线程可能还在等待其他条件。只有在所有的等待条件都满足后线程才会从 Blocked 状态才会变为就绪态。

RTEMS 线程的状态转换图如下所示：



RTEMS 线程的状态转换图

线程执行模式

RTEMS 可通过线程执行模式用来改变线程的调度方式和执行过程。可以通过设置 RTEMS_DEFAULT_MODES 来配置默认模式。RTEMS 使用数据类型 rtems_task_mode 来处理线程执行模式。线程的执行模式是下列四种状态的组合：

- ⑩ 抢占态 (preemption)
- ⑩ ASR 处理 (异步信号处理)
- ⑩ 时间片分片 (timeslicing)
- ⑩ 中断等级：中断等级包含如下几类：
 - ⑩ RTEMS_PREEMPT- 抢占模式有效 (默认)
 - ⑩ RTEMS_NO_PREEMPT- 非抢占模式
 - ⑩ RTEMS_NO_TIMESLICE- 时间片轮转失效 (默认)
 - ⑩ RTEMS_TIMESLICE- 时间片算法有效
 - ⑩ RTEMS_ASR- 异步处理有效 (默认)
 - ⑩ RTEMS_NO_ASR- 异步处理无效
 - ⑩ RTEMS_INTERRUPT_LEVEL-(0) 使所有的中断都有效 (默认)
 - ⑩ RTEMS_INTERRUPT_LEVEL(n)- 线程在第 n 级中断运行

线程上下文

在多线程系统中，运行的线程变为就绪或者阻塞时都需要保存当前线程的上下文，在线程下次得到 CPU 并运行时需要再恢复已保存的上下文，这样就能保证线程“连续”的运行下去。线程的上下文是和硬件的体系结构密切相关的，在 RTEMS 中，X86 平台的上下文定义如下：

```

typedef struct {
    uint32_t    eflags;    /* extended flags register */
}

```

```
void      *esp;      /* extended stack pointer register      */
void      *ebp;      /* extended base pointer register      */
uint32_t  ebx;      /* extended bx register      */
uint32_t  esi;      /* extended source index register      */
uint32_t  edi;      /* extended destination index flags register */
} Context_Control;
```

线程控制块中的 Registers 就是 Context_Control 类型的，也就是这个线程的上下文。

线程优先级管理

RTEMS 是基于优先级的抢占式调度，它支持256个优先级（0-255），数字越小优先级越高，0的优先级最高，这里需要注意，RTEM 的顺序和 POSIX 标准的规定刚好相反，，所以使用 RTEMS 的标准设定相优先级时 会转化成 POSIX 的优先级，转换方法是 Prtems=255-Pposix。 每个优先级的就绪的线程都是一个 FIFO 链表，TCB 里面会包含这个链表，便于在线程调度时的操作。_Thread_Ready_chain 就是由各个优先级的 FIFO 构成的数组，这是一个全局变量，函数_Thread_Handler_initialization(用来初始化线程管理的各个数据结构)会为此数组分配空间：

```
_Thread_Ready_chain = (Chain_Control *)
_Workspace_Allocate_or_fatal_error( (PRIORITY_MAXIMUM + 1) * sizeof(Chain_Control));
```

Chain_Control 结构在对象管理中已经进行过分析，它是对链表进行管理的数据结构，包含了一个链表头和尾，这样就可以很快地确定每个优先级的第一个线程和最后一个线程，而且当一个线程被切换出来的时候可以直接把它放到链表尾，不用沿着一条包含了所有就绪线程的链进行搜索(考虑到优先级)。但是这样做有一个问题，就是不太容易确定下一个优先级最高线程，如果被切换出来的线程它所在的优先级只有一个就绪线程，那么就需要在 _Thread_Ready_chain 里找出优先级最高的就绪线程，这是一个耗时的操作。RTEMS 采用位图的办法解决了这个问题。它把256个优先级 顺序地分成了16组，用一个16位的整数_Priority_Major_bit_map 的每一位来标志每一组优先级的状态：只要某一组优先级里面有一个 优先级有就绪的线程，那么相应的位就置1，每一组优先级中各个优先级的状态又可以具体的由一个16位的整数来表示，那么就应该有16个这样的整数，对应所有的优先级队列，系统里定义了_Priority_Bit_map[16]，它的每个元素就是每一组优先级的具体状态。下图更清楚地反映出这种关系(最上面的 16位数就是 _Priority_Major_bit_map)：

从
上
可
以
当
前
线
程
先
级
的
是

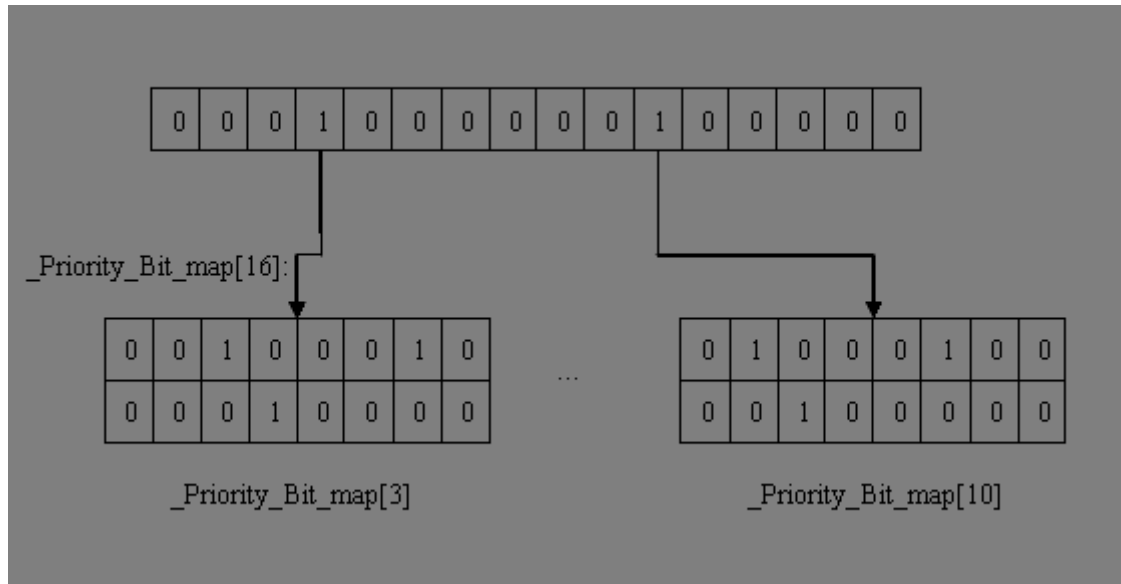


图
中
得
知
，
就
绪
中
优
先
级
最
高

3*16+2=50，这样就大大提高了搜索的速度。因为只有 256个优先级，所以一个优先级只有8位是有效的(仅管 i386中将优先级定义为32位，这是从 CPU 寻址的效率的角度考虑的)，这样一个 priority 就可以分

为 `major(priority/16)` 和 `minor(priority%16)` 两部分，`major` 就是它所在的优先级组在 `_Priority_Major_bit_map` 中的入口，而 `minor` 就是它在 `_Priority_Bit_map[major]` 中的入口。

那么一个线程如何与上面的两个结构交互的呢？在 TCB 里定义了一个 `Priority_Information` 类型的成员 `Priority_map`，`Priority_Information` 的定义如下：

```
typedef struct {
    Priority_Bit_map_control *minor;
    Priority_Bit_map_control ready_major;
    Priority_Bit_map_control ready_minor;
    Priority_Bit_map_control block_major;
    Priority_Bit_map_control block_minor;
} Priority_Information
```

`minor` 是这个线程对应的 `_Priority_Bit_map[major]` 的地址，后面的四个成员分别是对这个线程在 `_Priority_Major_bit_map` 和 `_Priority_Bit_map[major]` 中对应的位置1和置0用的掩码，这个结构在函数 `Priority_Initialize_information` 中初始化，它会在 `Thread_Set_priority` 函数中被调用，也就是说在线程初始化的时候或者改变线程的优先级的时候都会修改这个结构。有了这个结构，线程就可以通过它和 `_Priority_Bit_map[major]` 和 `_Priority_Major_bit_map` 打交道了，在线程就绪的时候要把自己添加到相应优先级的就绪线程链表上，同时也要修改 `_Priority_Bit_map[major]` 和 `_Priority_Major_bit_map`，把相应的位置，该操作通过函数 `_Priority_Add_to_bit_map` 实现，将 `Priority_Information` 的两个表示 `ready` 的掩码分别和 `_Priority_Bit_map[major]`、`_Priority_Major_bit_map` 按位取或。相反，如果它被阻塞（`block`）了，那么就需要通过 `_Priority_Remove_from_bit_map` 相应的位置0，具体地可以想出就是把它的 `Priority_Information` 的两个表示 `block` 的掩码分别和 `_Priority_Bit_map[major]`、`_Priority_Major_bit_map` 按位取与。

优先级位图最重要的就是方便系统在线程调度的时候能方便地找出最高优先级线程，这是通过函数 `_Priority_Get_highest` 实现的：

```
RTEMS_INLINE_ROUTINE Priority_Control _Priority_Get_highest( void )
{
    Priority_Bit_map_control minor;
    Priority_Bit_map_control major;

    _Bitfield_Find_first_bit( _Priority_Major_bit_map, major );
    _Bitfield_Find_first_bit( _Priority_Bit_map[major], minor );

    return ( _Priority_Bits_index( major ) << 4 ) +
           _Priority_Bits_index( minor );
}
```

`_Bitfield_Find_first_bit` 是一个宏，其功能主是从第一个参数中从低位开始找到第一个不为0的数，并把它赋给第二个参数，具体的实现和 CPU 相关，在 i386 中是通过汇编指令 `BSFW` 实现的，X86 CPU 把相对复杂的位操作封装成了一条指令。找到了 `major` 和 `minor` 后，把 `major*16+minor` 就是当前最高的优先级了（`_Priority_Bits_index` 返回的就是参数本身）。这个函数在每次调度的时候都会用到，系统有一个局变量 `Thread_Heir`，表示当前最重要的线程，实际上是在当前优先级最高的那个就绪线程链表的第一线程。

线程等待队列

在实现同步和互斥时，很多情况都要用到线程等待队列，它可以把阻塞在同一个资源上的各线程按不同的策略排队。RTEMS 是通过 `Thread_queue_Control` 结构体及对它的相关操作来实现等待队列的。而这个结构体一般是同步互斥资源对象（如 `posix API` 层中的条件变量 `condition` 对象，核心层的 `coremsg` 对象/`coremutex` 对象/`corerwlock` 对象/`coresem` 对象等）的重要成员变量，当线程访问这些对象，且没

有得到满足时会被阻塞，并挂在相关对象的线程等待队中。

关键数据结构

线程等待队列控制块

线程等待队列控制块的结构体定义如下：

```
typedef struct {
    union {
        Chain_Control Fifo;
        Chain_Control Priority[TASK_QUEUE_DATA_NUMBER_OF_PRIORITY_HEADERS];
    } Queues;
    Thread_queue_States      sync_state;
    Thread_queue_Disciplines discipline;
    States_Control           state;
    uint32_t                 timeout_status;
} Thread_queue_Control;
```

联合体 Queues 的 Fifo 成员是采用 FIFO 策略的线程等待队列，Priority 成员是采用 priority 策略实现的线程等待队列，它是一个链表的数组，这个数组有4个元素，每个元素对应的链表上最多有64个不同优先级的线程，也就是把256个优先级分成了4组，这样做的目的是为了在入队时加快搜索的速度，在后面章节的入队函数分析中会看到这一优势，使用联合体结构是因为一个线程等待队列只能采用一种排队策略，由 discipline 成员指定这种策略。其策略的定义如下：

```
typedef enum {
    THREAD_QUEUE_DISCIPLINE_FIFO, /* FIFO queue discipline */
    THREAD_QUEUE_DISCIPLINE_PRIORITY /* PRIORITY queue discipline */
} Thread_queue_Disciplines;
```

对 discipline 的设置决定了对应线程等待队列的线程插入和取出顺序的方式。sync_state 代表了对线程等待队列的处理中的不同状态。而 sync_state 的可选择值如下：

```
typedef enum {
    THREAD_QUEUE_SYNCHRONIZED,
    THREAD_QUEUE_NOTHING_HAPPENED,
    THREAD_QUEUE_TIMEOUT,
    THREAD_QUEUE_SATISFIED
} Thread_queue_States;
```

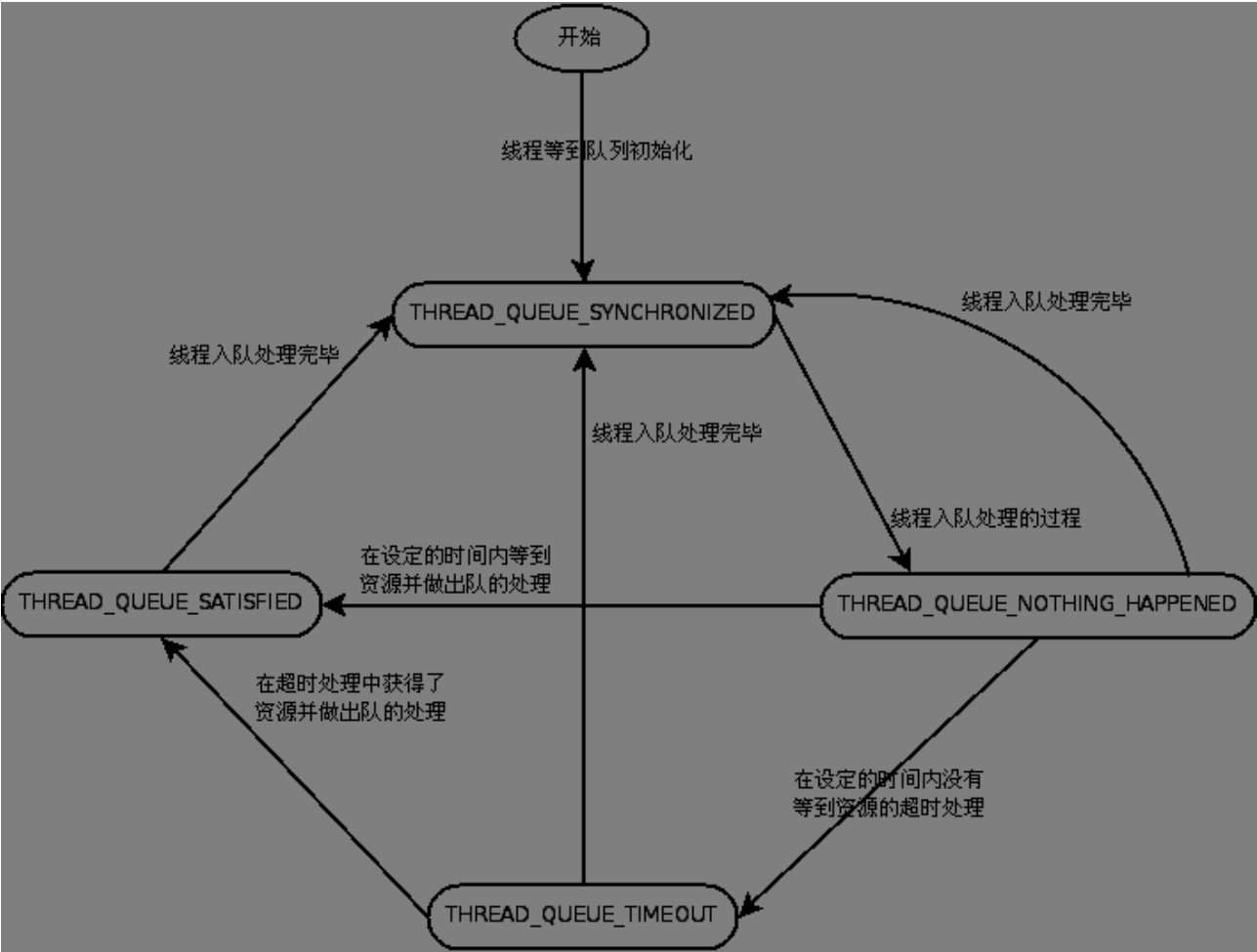
各状态的含义：

1. **THREAD_QUEUE_SYNCHRONIZED**：已同步状态。当调用函数 Thread_queue_Initialize 对线程等待队列初始化后，会把线程等待队列控制块的成员变量 sync_state 设置为该状态。在线程入队处理（即调用函数 Thread_queue_Enqueue_fifo 或 Thread_queue_Enqueue_priority）完毕，如果线程又会回到这个已同步状态。对线程队列的入队和出队操作会改变 sync_state 的已同步状态。
2. **THREAD_QUEUE_NOTHING_HAPPENED**：等待且无事状态。当线程申请资源（比如信号量等）时被阻塞，同时又指定了 timeout 时间希望等待一定时间（这个前件是可选的），会调用函数 Thread_queue_enter_critical_section，把线程等待队列的状态设置为等待且无事状态。此状态，表明在进行线程入队的操作，即线程挂在线程等待队列上在等待资源，且既没有到达等待期限，资源也没有得到满足。
3. **THREAD_QUEUE_TIMEOUT**：等待且超时状态。在进行线程入队的操作中，会指定一个 watchdog 对象和对应的超时处理函数。这样，如果在指定的 timeout 时间内没有获得需要的资源，watchdog 会触发，并调用超时处理函数进行超时处理。在超时处理函数 Thread_queue_Process_timeout 会把线程等待队列的

状态设为等待且超时状态，表明在进行线程在设定的时间内没有得到资源的处理。

4. THREAD_QUEUE_SATISFIED：满足状态。当线程可获得资源或在指定的时间内没有获得资源的时候，会调用函数_Thread_queue_Dequeue让线程出队并让线程从阻塞状态变成就绪态。当完成了出队列的处理后，会把线程等待队列的状态设置为满足状态。

线程等待队列的状态转换如下图所示：



线程等待队列状态转换图

timeout_status指定了在这个队列上等待的线程如果超时，这个线程的Wait结构体的return_code的值。

线程等待队列的管理

线程等待队列初始化

_Thread_queue_Initialize 用于线程等待队列结构体的初始化。

函数调用信息：

```
void _Thread_queue_Initialize(  
    Thread_queue_Control      *the_thread_queue,  
    Thread_queue_Disciplines  the_discipline,  
    States_Control             state,  
    uint32_t                  timeout_status
```

```
)
```

函数功能描述:

这个函数根据入口参数指定的 state、discipline 和 timeout_status 来创建一个线程等待队列结构体, 创建时会把同步状态初始化为 THREAD_QUEUE_SYNCHRONIZED, 然后会根据指定的排队策略来初始化相关的链表, 如果是 priority 策略, 需要初始化 Priority 的4个元素。

□程入口操作

如果一个线程想获得某共享对象, 但此时该对象被其他线程占有时, 那么它就会被阻塞, 并会被插入这个共享对象的等待队列里。线程指定等待最长的时间, _Thread_queue_Enqueue 宏就是完成这个入队操作的。此宏通过直接调用_Thread_queue_Enqueue_with_handler 函数来完成具体的工作。

函数调用信息:

```
void _Thread_queue_Enqueue_with_handler(  
    Thread_queue_Control *the_thread_queue,  
    Watchdog_Interval timeout,  
    Thread_queue_Timeout_callout handler  
)
```

函数功能描述:

此函数的具体实现如下:

- ⑩ 要加入队列的线程就是间接调用这个函数的当前线程
- ⑩ 把设置线程的 block 状态置为线程等待队列的状态, 比如, 如果等待 mutex, 就使用 STATES_WAITING_FOR_MUTEX 初始化
- ⑩ 如果设置了 timeout, 就为该线程初始化一个定时器, (每个线程结构体里都有一个定时器结构, 用于该和线程相关的定时操作), 并指定定时器到时运行的函数为_Thread_queue_Timeout
- ⑩ 把该定时器按照参数给定的 timeout 插入定时器的链表的相应位置, 开始计时
- ⑩ 最后根据不同的排队策, 调用 _Thread_queue_Enqueue_fifo 或者 _Thread_queue_Enqueue_priority 函数, 将该线程插入队列

_Thread_queue_Enqueue_fifo 函数将当前线程插入到线程等待队列的 Fifo 成员指向的链表上, 而 _Thread_queue_Enqueue_fifo 函数实现相对复杂, 它先通过调用 _Thread_queue_Header_number(priority) 函数来找到这个优先级所在的线程链表, 实际上就是把优先级除以64, 然后把优先级和0x20按位与, 如果等于0则说明它的低6位小于32, 也就是说该优先级应该位于队列的前半部分, 则应该从链表的前面搜索, 否则就从链表后端开始搜索, 这样对于任意一个优先级(0-255), 最坏的情况需要搜索32次, 这就是前面提到的按优先级排队时分组的好处, 接着这个函数会在相应的链表上查找到该线程应该插入的位置, 然后插入链表。如果这个链表上已经存在一个和该线程的优先级相同的线程了, 那么就在这个线程加入到已经存在的那个线程的 Wait.Block2n 成员上, Wait.Block2n 是一个指向所有和该线程阻塞在同一对象上相同优先级的其他线程的链表。

□程出口操作

函数_Thread_queue_Dequeue 执行出队操作, 把出队的线程作为它的返回参数。

函数调用信息:

```
Thread_Control *_Thread_queue_Dequeue(  
    Thread_queue_Control *the_thread_queue  
)
```

函数功能描述:

某一共享对象可用的时候就会从它的待待队列里按排队策略选择一个新的线程占有该对象, 这个函数就是简单地根据排队策略调用_Thread_queue_Dequeue_fifo 或者 _Thread_queue_Dequeue_priority, 前者就是把该线程从边等待队列里取出第一个节点, 如果这个线程指定了等待时间, 就把这个线程的定

时器停掉，因为它已经获得了这个资源，最后需要做的就是通过调用_Thread_Unblock 把获得共享对象的线程置为就绪状态。对于后者来说，就是找到 Queues.Priority 的第一个不为空的链表，然后取出它的 first 节点作为返回的线程，并且如果这个线程的 Wait.Block2n 指向的链表不为空，也就是说还有相同的优先级的线程也在等待这个共享对象，把这个链表上的第一个线程插入到取出的线程在等待队列 中的位置，并且把取出线程的 Wait.Block2n 链表上的其他线程放到新插入的线程的 Wait.Block2n 指向的链表上，最后也需要停掉定时器，使线程就绪。

线程等待队列到线程管理函数

如果在获到共享资源时指定了时间，那么在入队的时候就会初始化一个定时器，这个函数就是在定时器超时会调用的处理函数_Thread_queue_Timeout。

函数调用信息：

```
void _Thread_queue_Timeout(  
    Objects_Id id,  
    void *ignored  
)
```

函数功能描述：

它首先根据初始化传化的 id（启用定时器的线程的 ID），找到该定时器对应的线程，然后把这个线程的 Wait 结构体的 return_code 成员设置成该队列的 timeout_status，比如 CORE_MUTEX_TIMEOUT，然后调用 _Thread_queue_Extract 函数把这个线程从等待队列中取出，这个函数做的事情和 _Thread_queue_Dequeue 是十分类似的，只不过前者是取出第一个线程，而后者给定了要取出的线路程，最后也会删掉定时器，把取出的线程设为就绪状态参与调度。另外，函数_Thread_queue_First 是找到等待队列上的第一个线程，但不取出，_Thread_queue_Flush 是清空等待队列。

线程管理

RTEMS 的核心层中对线程的管理包括线程管理的初始化，线程的初始化、启动、就绪、调度、阻塞、恢复、退出等。其实线程管理就是对线程控制块 TCB 的管理。在线程管理中，是当前正在运行的线程线程控制块结构体的全局变量_Thread_Executing，表示_Thread_Heir 就绪列表中优先级最高的线程的？，这两个变量在线程管理中经常被使用。处于就绪态的线程是放置在线就绪队列链表_Thread_Ready_chain 中。所有的线程对象放置在线_Thread_Internal_information 中。下面是与线程管理相关的关键函数分析。

线程管理器初始化

在 RTEMS 启动的时候，需要对线程管理使用到的全局变量进行初始化，主要包括就绪队列链表，还有线程的对象信息管理表等。这部分功能是在启动由 rtems_initialize_executive_early 函数调用 _Thread_Handler_initialization 完成的。

函数调用信息：

```
void _Thread_Handler_initialization(  
    uint32_t ticks_per_timeslice,  
    uint32_t maximum_extensions,  
    uint32_t maximum_proxies  
)
```

函数功能描述：

此函数的主要流程如下：

1. 根据 configuration_table->ticks_per_timeslice 初始化_Thread_Ticks_per_timeslice，这个值是可以配置的；初始化线程管理相关的全局变量：

```
_Context_Switch_necessary = FALSE;  
_Thread_Executing = NULL;  
_Thread_Heir = NULL;  
#if ( CPU_HARDWARE_FP == TRUE ) || ( CPU_SOFTWARE_FP == TRUE )  
_Thread_Allocated_fp = NULL;
```

```
#endif
_Thread_Do_post_task_switch_extension = 0;
_Thread_Maximum_extensions = maximum_extensions;
_Thread_Ticks_per_timeslice = ticks_per_timeslice;
```

有关这些变量的含义在下面的函数分析中会描述

2. 为_Thread_Ready_chain 分配内存，并且对它的每个元素也就是每个优先级上的就绪线程链表进行初始化；
3. 调用_Objects_Initialize_information 初始化_Thread_Internal_information，用于线程对象管理。

线程初始化

在 RTEMS 中创建一个线程时首先需要做的就是根据创建线程时使用的参数来初始化线程控制块，_Thread_Initialize 函数就是用于完成线程的初始化。

函数调用信息：

```
boolean _Thread_Initialize(
  Objects_Information      *information,
  Thread_Control          *the_thread,
  void                   *stack_area,
  uint32_t               stack_size,
  boolean                is_fp,
  Priority_Control        priority,
  boolean                is_preemptible,
  Thread_CPU_budget_algorithm budget_algorithm,
  Thread_CPU_budget_algorithm_callout budget_callout,
  uint32_t               isr_level,
  Objects_Name           name
)
```

函数功能描述：

它的具体工作如下：

1. 首先会判断是否已经为线程分配栈，如果已经有了，只是简单的初始化一下，如果没有，就要给线程分配堆栈空间(调用 _Thread_Stack_Allocate 函数)，i386中定义的最小栈大小为4K，如果指定的栈大小小于4K，系统仍会提供4K 空间，线程的栈信息保存在 TCB 的 start 成员变量的域 stack 里；
1. 如果定义了浮点支持，则要分配线程的浮点上下文空间，并进行初始化；
2. 然后会初始化定时器，把 TCB 的 libc_reent 成员变量赋值为 NULL，初始化扩展区域，如果这个线程的 budget_algorithm 等于 THREAD_CPU_BUDGET_ALGORITHM_EXHAUST_TIMESLICE，则要先给这个线程分配时间片

```
the_thread->cpu_time_budget = _Thread_Ticks_per_timeslice);
```

3. 设置 state 为 STATES_DORMANT，设置优先级，调用_Thread_Set_priority 设定它的 Priority_map；根据调用参数，对 TCB 的其它成员变量进行初始化；
4. 然后调用_Objects_Open 把这个线程作为对象添加到描述这类对象（线程的）Object_Information 中，这样才能在接下来的线程管理中正常使用 TCB。具体工作包括根据这个对象的 id 找出它的 index，把它加到 information 的 localtable 里，而它的 id 是在进入_Thread_Initialize 以前给它分配对象（最终是调用_Objects_Allocate 实现的）的时候产生的。

线程启动

_Thread_Initialize 初始化了线程的 TCB 结构，但是线程还不能执行，还需要调用_Thread_Start

函数，使它变成就绪状态。

函数调用信息：

```
boolean _Thread_Start(  
    Thread_Control *the_thread,  
    Thread_Start_types the_prototype,  
    void *entry_point,  
    void *pointer_argument,  
    uint32_t numeric_argument  
)
```

函数功能描述：

其大致执行流程如下：

1. 它首先会判断线程的状态，是否是 STATES_DORMANT 态(已经创建但没有就绪)，如果是，就会根据传进的参数对 start 初始化，包括入口函数的地址，函数的类型，函数所使用的指针参数和数值参数；
2. 调用 _Thread_Load_environment 函数，该函数初始化线程的上下文（浮点、通用寄存器），初始化线程的堆栈，拷贝线程的初始化信息到线程控制块中。

☞ _Thread_Load_environment 函数调用 _Context_Initialize 函数，为线程初始化 Registers 结构，主要是堆栈指针，_Context_Initialize 函数会把 _Thread_Handler 的地址存放在堆栈，第一次线程切换时会调用这个函数，它会根据 start 中的入口函数类型以相应的方式来调用入口函数，入口函数类型就是这指针型的或数值型的；

3. 调用 _Thread_Ready 函数，_Thread_Ready 函数的工作是：

☞ 屏蔽中断；

☞ 将请求的线程的状态设置为 STATES_READY，表明线程处于就绪态；根据线程优先级设置此线程的 Priority_map；把此线程的对象体（即 TCB 的 Object.Node）添加到就绪线程链表（即 TCB 的 ready 成员变量）中；

☞ 由于计算线程新的优先级会屏蔽中断，且时间较长，所以先执行一下 _ISR_Flash 函数，看看是否有中断需要处理；

☞ 通过 _Thread_Calculate_heir 函数重新计算线程的新的最高优先级的线程；设置 _Context_Switch_necessary = TRUE。到此为止，一个线程的运行环境都已经建立好了，现在就可以把它加到它所在优先级的就绪队列里；

☞ 使能中断；

4. _User_extensions_Thread_start 函数是查找 User_extensions_List 中的所有节点中的 Callouts.thread_start 是否不为空，如果不为空，则执行

```
(*the_extension->Callouts.thread_start)(_Thread_Executing, the_thread);
```

线程就绪

线程启动执行过程中会调用线程就绪处理函数，线程就绪的主要工作是通过 _Thread_Ready 函数实现，此函数的用途是将指定线程加入到系统线程的就绪队列中。

函数调用信息：

```
void _Thread_Ready(  
    Thread_Control *the_thread  
)
```

函数功能描述：

该过程看似很简单，修改线程的状态为 READY，修改系统就绪队列的掩码（major、minor），将线程加入到就绪队列中，计算新的后备线程，计算线程切换需求。因为涉及到系统的就绪队列，所有对就绪队列的操作必须为中断安全的，（中断中可以激活线程，并涉及到就绪队列的操作）。因为这一系列的操作时间比较长，特别是计算后备线程这一操作在一些 CPU 平台特别费时（没有 ffs 位操作指令的平台），对于实时系统来讲，关闭这么长时间的中断是不容许的，所以 RTEMS 的处理是在函数中计算后备线程之前 Flash 中断（打开后立刻关闭），目的就是为了解决中断。另外，这个函数只有启动线程和重新启动线程

函数对它有调用，线程同步对象中的激活线程调用的是其它函数。

线程切换

需要进行线程切换当与高优先级的线程准备就绪时，或者当前运行的线程阻塞或退出时，。线程切换是通过 `_Thread_Dispatch` 函数完成的。

函数调用信息：

`void _Thread_Dispatch(void)`

在线程切换函数能够执行的可行性方面，涉及有两个重要的相关全局变量： ？

- ❏ `_Thread_Dispatch_disable_level`：此变量称为线程切换使能计数器。RTEMS 维护了线程切换使能计数器，在系统调用入口，将该计数器加1，在出口将计数器值减1，如果计数器减为0，则进行线程切换(`_Thread_Dispatch`)。这样，保证在系统调用期间没有线程切换。（这与UNIX系统对系统调用中关闭线程切换很相似）。
- `_Context_Switch_necessary`：此变量称为系统切换需求标志。如果系统切换需求标志为1，则表示线程切换需求存在，否则表示没有线程切换需求，不必进行线程切换。

RTEMS 在调用线程切换之前，中断必须认为没有屏蔽，线程切换使能计数器为0(允许切换)。需要注意的是此处有中断竞争现象。比如在屏蔽中断之前产生了中断，然后中断服务例程在处理中确定在接下来的执行中需要进行线程切换()，在中断退出时产生线程切换，再次切换到本线程时，线程切换需求标志为0，处理中屏蔽中断后首先检测切换需求标志，这样就可以解决这个竞争)

RTEMS 在切换浮点上下文时，可以有使用编译开关选择浮点延迟切换的算法，这个算法对于系统中只有少量有浮点运算的线程可以减少系统切换开销。在该算法使能时，RTEMS 系统维护一个浮点延迟线程指针，只有把当前运行线程（全局变量 `_Thread_Executing`）切换到的后备线程（全局变量 `_Thread_Heir`）有浮点运算并且后备线程不是浮点延迟线程时，系统才执行浮点上下文切换。在该算法不使能时，浮点切换比较简单。

函数功能描述：

当上层函数会调用 `_Thread_Enable_dispatch` 函数时，此函数先会检查当前是否允许调度，如果可以就调用 `_Thread_Dispatch` 函数进行调度。它先会判断 `_Context_Switch_necessary` 是不是真，如果是就调用 `_Context_Switch(&executing->Registers, &heir->Registers)`，把当前最重要线程的上下文切换进来，如果对线程的时间片指的是每次切换都重置，那在此以前还需要重新给它分配一个时间片，然后把新线程的 `ticks_executed` 加1。下面是 `_Thread_Dispatch` 函数的执行流程：

1. 取得当前运行线程指针为源线程；
2. 屏蔽中断； //注意一
3. 如果 `_Context_Switch_necessary == TRUE`，则进行线程切换： //注意二
 - ❏ 取得后备线程的指针 `heir` 为后备线程 `_Thread_Heir`；
 - ❏ 设置线程切换使能计数器为1，设置系统切换需求标志为 `FALSE`（这样就禁止了线程切换）；
 - ❏ 设置当前执行线程 `_Thread_Executing` 为后备线程；
- 如果后备线程时间片重置选项为 `THREAD_CPU_BUDGET_ALGORITHM_RESET_TIMESLICE`（切换时重置），则重置后备线程的时间片；
 - ❏ 使能中断；
 - ❏ 接下来交换 `libc`（即 `newlib`）中的相关数据；
 - ❏ 然后执行用户扩展的线程切换函数；
 - ❏ 如果支持浮点，则保存当前线程的浮点上下文；
 - ❏ 调用 `_Context_Switch` 函数完成上下文切换；
 - ❏ 如果支持浮点，则恢复后备线程的浮点上下文；

❏ 屏蔽中断;

4. 设置_Thread_Dispatch_disable_level 为0, 这样就运行在将来调用_Thread_Dispatch 函数;
5. 使能中断;
6. 执行用户扩展的线程切换后处理函数。

注意一：这段程序中平台相关的有：中断屏蔽/使能，浮点寄存器上下文的保存/恢复，通用寄存器上下文的保存/恢复。

注意二：在判断_Context_Switch_necessary 是否为 TRUE 用的是“while”，而不是“if”，且在 while 循环中已经把_Context_Switch_necessary 设置为了 FALSE。其原因在于在 while 的处理中，会使能中断，这样可能会产生中断处理，并改变_Context_Switch_necessary 为 TRUE，这样还会执行 while 循环。所以这里要有“while”而不是“if”。

切换上下文

真正的上下文切换是在_Context_Switch 中完成的，它会被宏替换成另外一个函数_CPU_Context_switch，定义如下：

```
#define _Context_Switch( _executing, _heir ) \
    _CPU_Context_switch( _executing, _heir )
```

函数_CPU_Context_switch 是由汇编语言实现的，因为它是和体系结构密切相关的，这里以 i386 为例分一下_CPU_Context_switch 的实现。在 cpukit/score/cpu/i386/cpu_asm.S 中定义了_CPU_Context_switch 函数：

```
SYM ( _CPU_Context_switch ) :
    movl    RUNCONTEXT_ARG(esp), eax    /* eax = running threads context */
    pushf                                /* push eflags */
    popl    REG_EFLAGS(eax)             /* save eflags */
    movl    esp, REG_ESP(eax)           /* save stack pointer */
    movl    ebp, REG_EBP(eax)           /* save base pointer */
    movl    ebx, REG_EBX(eax)           /* save ebx */
    movl    esi, REG_ESI(eax)           /* save source register */
    movl    edi, REG_EDI(eax)           /* save destination register */

    movl    HEIRCONTEXT_ARG(esp), eax   /* eax = heir threads context */

restore:
    pushl   REG_EFLAGS(eax)             /* push eflags */
    popf                                         /* restore eflags */
    movl    REG_ESP(eax), esp            /* restore stack pointer */
    movl    REG_EBP(eax), ebp            /* restore base pointer */
    movl    REG_EBX(eax), ebx            /* restore ebx */
    movl    REG_ESI(eax), esi            /* restore source register */
    movl    REG_EDI(eax), edi            /* restore destination register */
    ret
```

这个函数首先保存原来运行的线程的上下文到原来运行 TCB 的 Registers 成员里，然后从后备线程的 TCB 的 Registers 成员里恢复它的上下文，最后执行 ret 指令，这个指令会从栈中找到返回地址，因为这个函数是通过调用获得执行的，所以它的返回地址一定会保存在栈里，而在恢复了栈指针以后，也就是说 esp 指向的是后备线程的栈，这样在返回的时候就会返回到这个后备线程上次被换出的时的调用位置，它就可以继续运行了，实际上这个切换点就是在_Thread_Dispatch 函数的_Context_Switch 处，也就是说一个线程是在_Context_Switch 中被换，下次运行的时候从_Context_Switch 之后开始运行，线程的第一次运行除外。

线程起始运行点

在启动线程后，线程就可以被调度，一旦一个还未运行的线程得到CPU控制权后，线程的运行流程就会转到线程的起始运行点（_Thread_Handler）。为何会转跳到_Thread_Handler 函数作为线程的起始点来执行呢？这是由于在执行线程启动函数的时候，_Thread_load_environment_p 会把_Thread_Handler 的地址放在线程的上，这样第一次线程切换时根据线程的信息，会调用_Thread_Handler 函数。

函数调用信息：

```
void _Thread_Handler( void )
```

函数功能描述：

在这个函数中，系统根据线程的入口地址、调度策略和线程需要执行的函数输入参数跳转到用户线程的起始地址，线程如果间接结束时（以return指令退出），线程的运行又转向这个函数。此时，系统产生一个内部错误处理，该处理是一个永远不会返回的过程（也无处可以返回了）。具体实现如下：

1. 根据线程的中断等级属性设置CPU相关的中断等级；
2. 调用用户定义的线程起始的扩展函数；
3. 调用使能线程切换函数，会进一步调用线程切换函数；//注意一
4. 根据线程的输入参数，以不同的入口参数调用线程入口函数；
5. 调用用户定义的线程结束的扩展函数；//注意二
6. 系统内部错误；

注意一：这是必须注意的地方，调用这个函数的原因必须与线程切换的过程联合起来理解，普通的线程在切换回来后程序流程都在线程切换的系统调用中，但是还未运行的线程切换回来后跳转到本函数。线程切换流程中首先置系统的线程切换使能计数器为1，然后进行线程的上下文切换，普通切换必然运行到线程切换的后部分代码，它们将把线程切换时能计数器置为0，但是还未运行的线程则不是这样的流程

注意二：通常结束线程，流程会切换到其它线程，而不是继续执行到第5步和第6步。所以在第6步时，会认为RTEMS产生了内部错误，并进行调用处理致命错误的函数。

线程阻塞

函数_Thread_Suspend 用于阻塞一个线程，即把它的状态设置为阻塞态。它的入口参数是需要阻塞的线程控制块。

函数调用信息：

```
void _Thread_Suspend(  
    Thread_Control *the_thread  
)
```

函数功能描述：

具体实现如下：

1. 获得阻塞线程所在的就绪线程链表 ready；
2. 屏蔽中断；
3. 把该线程的阻塞次数 suspend_count 加1；
4. 如果该线程不是处于就绪状态，就在线程的状态中加入 STATES_SUSPENDED 位；
5. 如果该线程原来处理就绪状态，则直接把线程的状态设为 STATES_SUSPENDED；
6. 如果 ready 上只有这一个就绪线程，把 ready 设为空，并且调用_Priority_Remove_from_bit_map 清除优先级位图中这个线程的优先级所对应的位；
7. 如果 ready 链上还有其他线程，则直接把该线程从链上取下；

- 判断该线程是否是优先级最高的线程，如果是则由于它要阻塞，所以要调用 `_Thread_Calculate_heir` 函数重新计算最高优先级的线程；
- 如果该线程是当前运行的线程，将 `_Context_Switch_necessary` 设为真，表明需要线程调度；

线程恢复

函数 `_Thread_Resume` 用于恢复一个线程, 即把线程状态恢复为就绪态。

函数调用信息:

```
void _Thread_Resume(  
    Thread_Control *the_thread,  
    boolean        force  
)
```

它的入口参数包括:

❏ `the_thread` - 需要阻塞的线程的控制块

❏ `force` - 是否强制恢复

函数功能描述:

具体实现如下:

- 屏蔽中断；
- 如果要求强制恢复，则直接把该线程的阻塞次数 `suspend_count` 设为0，否则将其减1；
- 判断 `suspend_count` 是否大于0，如果大于0则使能中断并返回；
- 判断线程原来的状态是否是 `STATES_SUSPENDED`，如果不是，则使能中断并直接返回；
- 如果状态是否是 `STATES_SUSPENDED`，则把状态的 `STATES_SUSPENDED` 位清除，然后再判断线程的状态是否就绪，如果仍然不是就绪则使能中断直接返回；
- 如果已经就绪，调用 `_Priority_Add_to_bit_map` 添加优先级位图中这个线程的优先级所对应的位；
- 把这个线程添加到它所在的优先级对应的就绪线程链表 `ready` 上；
- 调用 `_ISR_Flash` 函数暂时使能中断；
- 比较该线程与当前运行线程的优先级，如果比当前运行线程的优先级高，则把要恢复的线程控制块 `the_thread` 赋给后备线程全局变量 `_Thread_Heir`；
- 如果当前运行的线程可抢占或者要恢复的线程的优先级为0（表明是优先级最高的系统线程），将 `_Context_Switch_necessary` 设为真，表明需要线程调度；
- 使能中断直接返回。

线程关闭

函数 `_Thread_Close` 用于关闭一个线程，主要是释放线程占用的内存，并把这个线程从它所在的本地对象链表中删除。

函数调用信息:

```
void _Thread_Close(  
    Objects_Information *information,  
    Thread_Control     *the_thread  
)
```

它的入口参数包括:

❏ `information` - 线程对象所属的对象信息管理结构

❏ `the_thread` - 需要关闭的线程控制块

函数功能描述:

具体实现如下：

1. 调用 `_User_extensions_Thread_delete` 函数删除线程的用户扩展；
2. 调用 `_Objects_Close` 函数把这个线程从它的对象信息管理表 `information` 中注销；
3. 调用 `Thread_Set_state` 函数把线程的状态设为 `STATES_TRANSIENT`，这个操作会引起系统重新计算最高优先级线程；
4. 调用 `_Thread_queue_Extract_with_proxy` 函数把该线程从等待队列中删除，如果这个函数返回0，即这个线程没有处于 `STATES_WAITING_ON_THREAD_QUEUE` 状态，则判断线程的定时器是否处于运行状态，如果是则删除定时器；
5. 调用 `_Thread_Stack_Free` 函数释放线程的堆栈；
6. 调用 `_Workspace_Free` 函数释放线程的 `extension` 成员变量。

线程放弃 CPU 控制权

函数 `_Thread_Yield_processor` 用于让当前执行此函数的线程主动让出 CPU，这时让出 CPU 控制权的线程的状态会变成就绪态。此函数不需要入口参数。

函数调用信息：

```
void _Thread_Yield_processor( void )
```

函数功能描述：

具体实现如下：

1. 判断当前运行线程的就绪链表上是否只有一个线程；
2. 如果不是，把运行线程从链表中取出放在链表尾，如果它是最高优先级就绪的队列中排在最前面的线程，那么把它的就绪链表上的第一个线程赋给后备线程全局变量 `_Thread_Heir`，并把 `_Context_Switch_necessary` 设为真；
3. 如果就绪链表上只有一个线程，则判断它是就绪线程，如果是则把 `_Context_Switch_necessary` 设为真，否则直接返回。

改变线程优先级

函数 `_Thread_Change_priority` 用于改变线程的优先级，该函数在执行过程中，有可能对线程就绪队列等的排列进行改变。

函数调用信息：

```
void _Thread_Change_priority(  
    Thread_Control *the_thread,  
    Priority_Control new_priority,  
    boolean        prepend_it  
)
```

函数功能描述：

具体实现如下：

1. 调用函数 `_Thread_Set_transient` 把线程状态在逻辑或上 `STATES_TRANSIENT`，这主要是让线程处于一个临时非执行状态，防止发生中断时该线程被调度到；
2. 如果线程老的优先级不等于输入参数 `new_priority`，则调用 `_Thread_Set_priority` 函数，根据 `new_priority`，改变线程的成员变量 `current_priority`，`ready` 和 `Priority_map`；
3. 屏蔽中断；
4. 如果线程的状态不仅仅是 `STATES_TRANSIENT`，

- ❏ 如果还有阻塞状态，则对线程等待队列重新排序；
 - ❏ 清除线程状态位 STATES_TRANSIENT；
 - ❏ 使能中断；
 - ❏ 返回；
1. 如果线程状态只是 STATES_TRANSIENT，则清除线程状态位 STATES_TRANSIENT；如果参数 prepend_it 为 TRUE，则把线程对象插入到线程就绪队列的前面；
 5. 如果参数 prepend_it 为 FALSE，则把线程对象插入到线程就绪队列的后面；
 6. 调用_ISR_Flash 暂时使能中断；
 7. 调用_Thread_Calculate_heir 函数，重新找到后备线程；
 8. 如果当前执行线程不是后备线程，且运行线程切换，则设置全局变量 _Context_Switch_necessary 为 TRUE；
 9. 使能中断；
 10. 返回。

系口服口的任口管理

RTEMS 的任务管理 (task manager) 组件提供了 CLASSIC、POSIX 和 ITRON 三种 API，其中 RTEMS CLASSIC API 是 RTEMS 缺省的 API 接口实现，是实现主要建立在核心层的 thread handler 组件基础之上。RTEMS CLASSIC API 接口函数主要包括：

- ❏ rtems_task_create - 创建任务
- ❏ rtems_task_ident - 获取任务的 ID
- ❏ rtems_task_start - 启动任务
- ❏ rtems_task_restart - 重启任务
- ❏ rtems_task_delete - 删除任务
- ❏ rtems_task_suspend - 阻塞任务
- ❏ rtems_task_resume - 恢复任务
- ❏ rtems_task_is_suspended - 确定任务是否被阻塞
- ❏ rtems_task_set_priority - 设置任务的优先级
- ❏ rtems_task_mode - 改变当前任务模式
- ❏ rtems_task_get_note - 获得任务记事本入口
- ❏ rtems_task_set_note - 设置任务记事本入口
- ❏ rtems_task_wake_after - 在指定的一段时间后唤醒任务
- ❏ rtems_task_wake_when - 在指定的时刻唤醒任务
- ❏ rtems_task_variable_add - 添加任务变量
- ❏ rtems_task_variable_get - 获取任务变量的值
- ❏ rtems_task_variable_delete - 删除任务变量

从上述函数的实现上看，在这样层没有对核心层的 TCB 进行扩展封装，而是直接使用了核心层的 TCB，主要考虑了对任务优先级（对应线程优先级）、任务执行模式（线程执行模式）的管理。扩展了 ASR 的处

理。有关 ASR 的内容请参考“同步互斥与通信机制”一章的“异步信号”一节。如果支持多处理器运行（AMP 方式），还要进一步增加对 MPC1 的支持。在数据结构上增加新的内容。

关□□□函数

任务管理器初始化

任务管理器初始化工作由函数 `_RTEMS_tasks_Manager_initialization` 完成。其来龙去脉如下：

```
rtems_initialize_executive_early-->_RTEMS_API_Initialize-->_RTEMS_tasks_Manager_initialization
```

其主要工作是对 API 的配置信息对保存任务相关信息的 `_RTEMS_tasks_Information` 进行初始化，增加 RTEMS USER extension 和 RTEMS API extension 的初始化。这里的 RTEMS USER extension 由全局变量 `_RTEMS_tasks_User_extensions` 表示：

```
User_extensions_Control _RTEMS_tasks_User_extensions = {
{ NULL, NULL },
{ { NULL, NULL }, _RTEMS_tasks_Switch_extension },
{ _RTEMS_tasks_Create_extension, /* create */
_RTEMS_tasks_Start_extension, /* start */
_RTEMS_tasks_Start_extension, /* restart */
_RTEMS_tasks_Delete_extension, /* delete */
_RTEMS_tasks_Switch_extension, /* switch */
NULL, /* begin */
NULL, /* exited */
NULL /* fatal */
}
};
```

通过在执行线程相对应的函数时，完成对任务的特定属性的管理。比如在执行任务创建函数 `rtems_task_creat` 时，会调用 `_Thread_Initialize` 函数，在此函数中，会进一步执行 `_RTEMS_tasks_Create_extension` 函数，完成对阻塞事件、异步信号服务例程 ASR 等的初始化工作等。这样就可以进一步支持同步互斥和任务间通信了。

RTEMS API extension 由全局变量 `_RTEMS_tasks_API_extensions` 表示：

```
API_extensions_Control _RTEMS_tasks_API_extensions = {
{ NULL, NULL },
NULL, /* predriver */
_RTEMS_tasks_Initialize_user_tasks, /* postdriver */
_RTEMS_tasks_Post_switch_extension /* post switch */
};
```

这里面定义了一个函数 `_RTEMS_tasks_Post_switch_extension`，此函数与异步信号服务例程（ASR）很相关。此函数会在执行线程切换函数 `_Thread_Dispatch` 的后期流程中被调用。此函数的工作就是完成对 ASR 的相关处理和函数调用。

函数调用信息

```
void _RTEMS_tasks_Manager_initialization(
uint32_t maximum_tasks, //最大任务数
uint32_t number_of_initialization_tasks, //要初始化的任务数
rtems_initialization_tasks_table *user_tasks //任务表
)
```

函数功能描述

1. 根据调用参数给任务相关全局变量赋值；

```
_RTEMS_tasks_Number_of_initialization_tasks = number_of_initialization_tasks;
```

```
_RTEMS_tasks_User_initialization_tasks = user_tasks;
```

2. 调用 `_Objects_Initialize_information` 函数初始化保存任务相关信息的 `_RTEMS_tasks_Information`;
3. 调用 `_User_extensions_Add_API_set` 函数增加 RTEMS USER extension;
4. 调用 `_API_extensions_Add` 函数增加 RTEMS API extension。

任务创建

函数调用信息

```
rtems_status_code rtems_task_create(  
  rtems_name      name,           //用户指定的任务名  
  rtems_task_priority initial_priority, //任务的初始优先级  
  size_t          stack_size,     //栈的大小（以字节为单位）  
  rtems_mode      initial_modes,  //初始的任务模式  
  rtems_attribute attribute_set,   //任务的属性  
  Objects_Id      *id             //用于存放任务 ID 的地址  
)
```

函数功能描述:

函数 `rtems_task_create` 是用于创建任务，主要作用是分配并初始化线程的控制块和栈，并把新建的线程设置为 `STATE_DORMANT` 状态。

具体实现如下:

1. 检查用于存放任务 ID 的地址及任务名的合法性;
2. 在任务的属性中设置 `ATTRIBUTES_REQUIRED`，清除 `ATTRIBUTES_NOT_SUPPORTED`;
3. 根据任务的属性集确定任务是否需要浮点运算;
4. 检查优先级的合法性;
5. 调用 `_RTEMS_tasks_Allocate` 分配线程控制块，具体实现可参考“总体结构和系统构成”一章的“对象管理机制”小节;
6. 调用 `_Thread_Initialize` 初始化线程控制块;
7. 根据任务的模式设置 ASR 是否使能，这里用到了 RTEMS 的 API Extension 机制;
8. 将线程控制块的对象 ID 作为任务 ID 保存在入口参数中指定的 ID 地址;
9. 如果支持多处理运行，还要调用 `_Objects_MP_Open` 函数和 `_RTEMS_tasks_MP_Send_process_packet` 函数进行与 MPC1 相关的处理;
10. 调用 `_Thread_Enable_dispatch` 函数运行线程调度。

获取任务 ID

函数调用信息

```
rtems_status_code rtems_task_ident(  
  rtems_name name,           //用户指定的任务名  
  uint32_t   node,          //指定搜索的节点范围  
  Objects_Id *id             //存放任务 ID 的地址  
)
```

函数功能描述:

函数 `rtems_task_ident` 用于获取任务 ID

具体实现如下:

1. 判断任务 ID 地址的合法性;
2. 根据 `name` 判断要获取 ID 的任务是否是调用这个函数的任务本身, 如果是则返回线程控制块中保存的线程对象的 ID, 代码中是通过 `name == OBJECTS_ID_OF_SELF` 这个条件来判断的, 而 `OBJECTS_ID_OF_SELF` 的定义为0, 也就是说如果想获得任务本身的 ID, 只需要把 `name` 指定为0;
3. 如果需要获取其他线程的 ID, 则调用 `_Objects_Name_to_id` 在 `RTEMS_tasks_Information` 所管理的对象中, 在入口参数 `node` 指定的搜索范围内, 根据 `name` 查找。

任务启动

函数调用信息:

```
rtems_status_code rtems_task_start(
    rtems_id id,           //被启动任务的 ID
    rtems_task_entry entry_point, //任务开始执行的入口地址
    rtems_task_argument argument //任务参数
)
```

函数功能描述:

函数 `rtems_task_start` 用于启动任务, 在调用 `rtems_task_create` 创建任务以后还需要调用这个函数使任务处于就绪状态, 从而可以参与调度。

具体实现如下:

1. 调用 `_Thread_Get` 函数, 通过 `id` 获得重启任务的控制块及其所在位置;
2. 如果是当前CPU上的任务, 则调用 `_Thread_Start` 启动任务。

任务重启

函数调用信息:

```
rtems_status_code rtems_task_restart(
    Objects_Id id,           //任务 ID
    uint32_t argument        //任务参数
)
```

函数功能描述:

函数 `rtems_task_restart` 用于任务重启, 它的具体实现如下:

1. 调用 `_Thread_Get` 函数, 通过 `id` 获得重启任务的控制块及其所在位置;
2. 如果是当前CPU上的任务, 则调用 `_Thread_Restart` 重启任务。

任务删除

函数调用信息

```
rtems_status_code rtems_task_delete(
    Objects_Id id           //被删除任务的 ID
)
```

函数功能描述:

函数 `rtems_task_delete` 用于删除任务, 使任务退出它的生命周期。

具体实现如下：

1. 调用_Thread_Get函数，通过id获得重启任务的控制块及其所在位置；
2. 如果是当前CPU上的任务，则调用_Objects_Get_information函数从全局变量_RTEMS_tasks_Information获取它的对象管理信息结构；
3. 调用_Thread_Close删除线程；
4. 调用_RTEMS_tasks_Free释放线程控制块。

任务阻塞

函数调用信息

```
rtems_status_code rtems_task_suspend(  
  Objects_Id id           //被阻塞任务的 ID  
)
```

函数功能描述

函数rtems_task_suspend用于阻塞一个任务，并把任务的状态改变为阻塞态。

具体实现如下：

1. 调用_Thread_Get函数，通过id获得重启任务的控制块及其所在位置；
2. 如果是当前CPU上的任务，则调用_States_Is_suspended判断是否已经被阻塞；
3. 如果没有阻塞，则调用_Thread_Suspend阻塞任务。

任务恢复

函数调用信息

```
rtems_status_code rtems_task_resume(  
  Objects_Id id           // 被恢复线程的 ID  
)
```

函数功能描述

函数rtems_task_resume用于任务恢复，在调用 rtems_task_suspend阻塞一个任务以后可以用rtems_task_resume来恢复任务，即把任务的状态改变成就绪态。

具体实现如下：

1. 调用_Thread_Get函数，通过id获得重启任务的控制块及其所在位置；
2. 如果是当前CPU上的任务，则调用_States_Is_suspended判断是否已经被阻塞；
3. 如果阻塞，则调用_Thread_Resume恢复任务。

设置任务优先级

函数调用信息

```
rtems_status_code rtems_task_set_priority(  
  Objects_Id id,          // 需要设置优先级的任务 ID  
  rtems_task_priority new_priority, // 线程的优先级 (0表示当前优先级)  
  rtems_task_priority *old_priority // 用于保存线程原来的优先级的指针  
)
```

函数rtems_task_set_priority用于设置任务的优先级，并且保存任务原来的优先级

具体操作如下：

1. 检查新的优先级是否有效保存原来的优先级的指针是否为空；

2. 调用 `_Thread_Get` 函数，通过 `id` 获得重启任务的控制块及其所在位置；
3. 如果是当前CPU上的任务，保存当前优先级到 `old_priority`；
4. 如果新的优先级不是当前优先级，则判断是否满足设置优先级的条件：

```
the_thread->resource_count == 0 || the_thread->current_priority >
new_priority
```

也就是说只有在这个任务不占有资源或者是占有资源但是想提高优先级的情况下才可以设置优先级，这个是为了避免出现优先级翻转；

5. 如果可以设置优先级，则调用 `_Thread_Change_priority` 设置任务优先级。

口置任口模式

函数 `rtems_task_mode` 用于设置任务模式，并保存原来的任务模式。

函数调用信息

```
rtems_status_code rtems_task_mode(
    rtems_mode mode_set,
    rtems_mode mask,
    rtems_mode *previous_mode_set
)
```

在 RTEMS 中一个任务的执行模式包括下面4个因素：

- ☞ `preemption` 抢占
- ☞ `ASR processing` (asynchronous signal processing) 异步信号处理
- ☞ `timeslicing` 时间片
- ☞ `interrupt level` 中断级别

任务模式可以改变 RTEMS 的调度处理和任务执行环境。数据类型 `rtems_task_mode` 用来描述任务的执行模式。这四个方面的作用如下：

- ☞ 抢占 (`preemption`) 用来控制任务是否可以被抢占。如果抢占被禁止了即 `preemption` 被设为 `disabled` (`RTEMS_NO_PREEMPT`)，只要任务处于执行状态，它就不会释放对处理器的控制权，直到执行结束，即使是有一个更高优先级的任务处于就绪状态。如果 `preemption` 设为 `enabled` (`RTEMS_PREEMPT`)，当有更高优先级的任务就绪时，RTEMS 收回当前任务的执行权，并且交给高优先级的任务。
- ☞ 时间片 (`timeslicing`) 用来决定当任务是同优先级时是如何分配处理器的。如果 `timeslicing` 设为 `enabled` (`RTEMS_TIMESLICE`)，RTEMS 就会限制每个任务的执行时间，超过了执行时间就分给下一个任务。时间片的长度是由应用决定的，在配置表中定义。如果 `timeslicing` 设为 `disabled` (`RTEMS_NO_TIMERSLICE`)，这个任务就一直执行直到有更高的优先级的任务处于就绪状态。如果 `RTEMS_NO_PREEMPT` 被选中，那么 `timeslicing` 就被忽略。
- ☞ 异步信号处理 (`ASR`) 用来决定接收到的信号何时被任务处理。如果信号处理设为 `enabled` (`RTEMS_ASR`)，那么在任务下一次得到执行时会检查是否收到信号并进行处理。如果信号处理设为 `disabled` (`RTEMS_NO_ASR`)，那么任务接收到的所有信号都保持悬挂，直到信号处理被使能。这个设置只对那些建立了处理异步信号程序的任务起作用。
- ☞ 中断 (`Interrupt level`) 用来决定当任务执行时哪些中断可以使能，`RTEMS_INTERRUPT_LEVEL(n)` 定义了任务执行使能的中断级别 `n`。

RTEMS 通过下面的宏定义组合成不同的任务模式

```
#define RTEMS_PREEMPT_MASK    0x00000100 /* preemption bit */
#define RTEMS_TIMESLICE_MASK  0x00000200 /* timeslice bit */
#define RTEMS_ASR_MASK        0x00000400 /* RTEMS_ASR enable bit */
#define RTEMS_INTERRUPT_MASK  CPU_MODES_INTERRUPT_MASK
```

```

#define RTEMS_PREEMPT      0x00000000    /* enable preemption      */
#define RTEMS_NO_PREEMPT   0x00000100    /* disable preemption     */

#define RTEMS_NO_TIMESLICE 0x00000000    /* disable timeslicing    */
#define RTEMS_TIMESLICE   0x00000200    /* enable timeslicing     */

#define RTEMS_ASR          0x00000000    /* enable RTEMS_ASR      */
#define RTEMS_NO_ASR       0x00000400    /* disable RTEMS_ASR     */

```

上面的CPU_MODES_INTERRUPT_MASK是不同的CPU体系结构定义的，RTEMS最多支持256级中断，任务模式的低8位都保留下来用来描述允许的中断等级。但具体的CPU中可能并不会使用那么多中断等级，在i386中只使两用了两级：使能、屏蔽中断。

函数调用信息

```

rtems_status_code rtems_task_mode(
  rtems_mode mode_set,      //新的模式
  rtems_mode mask,         //掩码，指定需要设置哪些内容。
  rtems_mode *previous_mode_set // 保存原来模式的地址
)

```

函数功能描述

具体实现如下：

1. 检查用来保存原来模式的地址是否为空；
2. 从线程控制块中获取异步信号管理信息asr；

```

api = executing->API_Extensions[ THREAD_API_RTEMS ];
asr = &api->Signal;

```

3. 根据线程控制块的相关成员来判断任务原来是否可抢占，是否是时间片轮转，是否使能异步信号，获得原来的允许中断的等级，保存在previous_mode_set中；
4. 根据掩码判断是否需要修改抢占，如果是，则根据新的模式把控制块的is_preemptible成员设置为真或假；
5. 根据掩码判断是否需要修改时间片，如果是，则根据新的模式把控制块的budget_algorithm成员设置为 THREAD_CPU_BUDGET_ALGORITHM_RESET_TIMESLICE 或者 THREAD_CPU_BUDGET_ALGORITHM_NONE；
6. 根据掩码判断是否需要允许中断等级，如果是，则调用_Modes_Set_interrupt_level修改允许的中断等级，这个函数最终会调用CPU的中断等级设置函数；
7. 根据掩码判断是否需要修改异步信号处理，如果是，获取新模式中指定的ASR使能状态，如果和原来的不同，则修改asr的is_enabled成员，并调用_ASR_Swap_signals交换asr的signals_pending和signals_posted，也就是说当ASR使能是信号集存入在signals_posted，当ASR禁用时信号集存放在signals_pending中；
8. 调用_ASR_Are_signals_pending判断是否有信号需要处理，如果有，则把needs_asr_dispatching和线程控制块的do_post_task_switch_extension置为真，表明需要处理信号；
9. 调用_System_state_Is_up判断系统当前是否处于正常处理状态，如果是，则进一步判断是否需要任务调度，判断的条件是：

```

_Thread_Evaluate_mode() || needs_asr_dispatching

```

函数_Thread_Evaluate_mode()的定义如下：

```

boolean _Thread_Evaluate_mode( void )
{
  Thread_Control *executing;
  executing = _Thread_Executing;
  if ( !_States_Is_ready( executing->current_state ) ||

```

```

        ( !_Thread_Is_heir( executing ) && executing->is_preemptible ) ) {
    _Context_Switch_necessary = TRUE;
    return TRUE;
}
return FALSE;
}

```

也就是说如果当前执行的任务不处于就绪态或者当前执行任务优先级不是最高并且可抢占，或者有信号需要处理，那么需要任务调度。

设置任务记事本

函数调用信息

```

rtems_status_code rtems_task_set_note(
    Objects_Id id,           //任务 ID
    uint32_t notepad,        //记事本号
    uint32_t note            //记录的信息
)

```

函数功能描述

RTEMS为每一个任务提供了16个记事本区域。每个记事本区域包含了4字节的记录信息。

函数rtems_task_set_note为指定任务的某个记事本设置信息。

具体实现如下：

1. 检查notepad合法性，是否超过最大值；
2. 判断需要设置的是否是当前任务，如果是，则通过线程控制块找到记事本的位置，写入信息：

```

api = _Thread_Executing->API_Extensions[ THREAD_API_RTEMS ];
api->Notepads[ notepad ] = note;

```

3. 如果不是，则通过调用函数_Thread_Get找到需要设置的线程，然后再执行上面的操作。

取任务记事本

函数调用信息

```

rtems_status_code rtems_task_get_note(
    Objects_Id id,           //任务 ID
    uint32_t notepad,        //记事本号
    uint32_t *note           //存入获取的信息
)

```

函数功能描述

函数 rtems_task_get_note 则是获取指定任务的某个记事本存的信息，在调用函数 rtems_task_set_note 设置了记事本信息后，可以通过这个函数来读取。

任务延时

函数 rtems_task_wake_after 使任务等待一定的时间后被唤醒。此函数涉及到定时器的管理。

函数调用信息

```

rtems_status_code rtems_task_wake_after(
    rtems_interval ticks      //等待的 tick 数
)

```

函数功能描述

具体实现如下：

1. 调用 `_Thread_Disable_dispatch`，禁止调度；
2. 如果需要等待的tick为0, 则直接调用 `Thread_Yield_processor` 让出CPU；
3. 如果不是0, 则调用 `_Thread_Set_state` 把任务的状态设为 `STATES_DELAYING`；
4. 调用 `_Watchdog_Initialize` 函数初始化线程的定时器，指定到时处理函数为 `_Thread_Delay_ended`，这个函数会使任务重新就绪；
5. 调用 `_Watchdog_Insert_ticks` 函数把线程的定时器根据指定的tick插入到系统的定时器队列，开始计时；
6. 调用 `_Thread_Enable_dispatch` 使能调度。

函数 `rtems_task_wake_when` 和 `rtems_task_wake_after` 类似，只不过它先把传入的唤醒时间转化成秒数和系统的当前时间 `_TOD_Seconds_since_epoch` 相减，然后以这个差值作为定时的时间插入定时器队列，不同的是插入以秒单位的定时器队列。RTEMS系统中维护了两个定时器队列，一个是以tick为单位的，一个是以秒为单位的。

任务变量

某些函数可能同时被多个任务调用，每个调用任务又期望函数中的全局变量或静态变量使用不同的值，这个时候就需要使用内核提供的任务变量，它存放在任务的上下文中，属于线程控制块的内容。在发生任务切换的时候，如果当前任务使用了任务变量，则需要把相应的全局变量保存到该任务的任务变量中，如果即将投入运行的任务使用了任务变量，则需要把任务变量的内容恢复到相应的全局变量中。可以看出，任务变量的存在会增加上下文切换的成本，特别是如果任务变量是比较大的结构体，那么情况会更加糟糕。为了解决这个问题，RTEMS中的任务变量采用指针变量，也就是说实际切换的时候只切换指针值，这样会减小成本。

添加任务变量

函数的调用信息

```
rtems_status_code rtems_task_variable_add(
    rtems_id tid,          //需要添加变量的任务 ID
    void **ptr,
    void (*dtor)(void *)   //变量的“析构函数”
)
```

其中 `ptr` 是指向添加变量的内存位置，这是一个二级指针，因为任务变量本身就是指针。

函数功能描述

函数 `rtems_task_variable_add` 用于添加任务变量，它的入口函数包括：

具体实现如下：

1. 检查 `ptr` 的合法性；
2. 调用 `_Thread_Get` 获取线程控制块及其位置；
3. 如果它是当前CPU上的对象则遍历TCB的 `task_variables` 所指向的所有任务变量，判断这个变量是否已经添加到任务变量；
4. 如果没有，调用 `_Workspace_Allocate` 分配一个 `rtems_task_variable_t` 结构体，并根据传入的参数进行初始化；
5. 把新的任务变量的 `next` 设置为 `task_variables`，并把 `task_variables` 指向新的任务变量，即把新的任务变量加入到该任务的任务变量链上。

获取任务变量的值

函数调用信息

```
rtems_status_code rtems_task_variable_get(
    rtems_id tid,          //需要获取变量的任务 ID
    void **ptr,            //指向需要获取的任务变量对应的全局变量的内存位置
    void **result          //保存任务变量的地址
)
```

函数功能描述:

函数 `rtems_task_variable_get` 用于添加任务变量

具体实现如下:

1. 检查 `ptr` 和 `result` 的合法性;
2. 调用 `_Thread_Get` 获取线程控制块及其位置;
3. 如果它是当前CPU上的对象则遍历TCB的 `task_variables` 所指向的所有任务变量, 从中查找任务变量对应的全局变量的内存地址和 `ptr` 相等的, 找到后把这个任务变量的 `tval` 成员放到 `result` 指向的地址中。

□除任□□量

函数调用信息

```
rtems_status_code rtems_task_variable_delete(
    rtems_id tid,           //需要删除变量的任务 ID
    void **ptr              //指向删除的任务变量对应的全局变量的内存位置
)
```

函数功能描述

函数 `rtems_task_variable_delete` 用于删除任务变量

具体实现如下:

1. 检查 `ptr` 和 `result` 的合法性;
2. 调用 `_Thread_Get` 获取线程控制块及其位置;
3. 如果它是当前CPU上的对象则遍历TCB的 `task_variables` 所指向的所有任务变量, 从中查找任务变量对应的全局变量的内存地址和 `ptr` 相等的, 然后把它从链中取下, 如果它原来处于链头, 那么就让 `task_variables` 指向它的下一个任务变量, 如果它处理链中, 那么就让它的前后任务变量相连;
4. 然后判断这个任务是否是当前运行的, 如果是则“析构” `ptr` 所指向的变量, 并把任务变量的 `gval` 成员赋给 `ptr` 所指的位置。因为当前任务正在运行, 所以 `ptr` 所指向的全局变量的值就是任务变量的值, RTEMS中的任务变量一般采用动态分配内存获得的指针, 所以“析构”一般采用 `free` 操作。如果这个任务不在运行, 则“析构”任务变量的 `tval`;
5. 最后调用 `_Workspace_Free` 释放这个任务变量结构体。

如果没有显式地删除一个任务变量, 在任务被删除时, 任务变量也会在函数 `_RTEMS_tasks_Delete_extension` 中被删除。另外, 任务变量的切换会在 `_RTEMS_tasks_Switch_extension` 中完成, 这个函数会在上下文切换时被调用。

应用举例

因为在实时操作系统中, 程序的设计和实现都离不开任务管理, 在后续各章节的举例中都涉及到任务管理的内容, 所以这里就不再单独举例说明。**还是要写例子!!!!**

系统扩展机制

RTEMS提供了两种系统扩展机制, 一个是API扩展, 一个是用户扩展。API扩展主要用于扩展核心层与API相关的行为, 这样保证了核心层不会直接调用RTEMS上层的服务, 从而确保了核心层与上层实现无关的特征。API扩展主要提供了在驱动程序初始化前后的扩展点以及上下文切换后的扩展点。用户扩展管理允许应用程序的开发人员增强程序的功能, 这是通过允许它们使用一些扩展程序来实现的, 这些程序在创建任务、上下文切换或者是删除任务等系统事件发生时被调用。

关键数据结构

RTEMS用 `API_extensions_Control` 结构体描述API扩展, 它包括一个把它挂在链表上的Node结点, 以及三个钩子 (hook) 函数指针, 其定义如下:

```
typedef struct {
    Chain_Node          Node;
    API_extensions_Predriver_hook predriver_hook; //驱动程序初始化前的钩子函数
    API_extensions_Postdriver_hook postdriver_hook; //驱动程序初始化后的钩子函数
    API_extensions_Postswitch_hook postswitch_hook; //在上下文切换后执行，运行在后备线程的上下文空间
} API_extensions_Control;
```

RTEMS用User_extensions_Control这个结构体来描述用户扩展，它包括一个把它挂在链表上的Node结点，还有一个User_extensions_Switch_control类型的Switch成员以及一个用户扩展表 Callouts，实际上在 User_extensions_Table中已经包含了任务切换的扩展，这里把它单独拿出来是出于效率上的考虑。所有的用户扩展结构会构成一个链，全局变量 User_extensions_List就指向这个链，而所有的任务切换扩展结构也会构成一个链，全局变量 User_extensions_Switches_list指向这个链。

User_extensions_Control 的定义：

```
typedef struct {
    Chain_Node          Node;
    User_extensions_Switch_control Switch;
    User_extensions_Table Callouts;
} User_extensions_Control;
```

User_extensions_Table 的定义

```
typedef struct {
    User_extensions_thread_create_extension thread_create;
    User_extensions_thread_start_extension thread_start;
    User_extensions_thread_restart_extension thread_restart;
    User_extensions_thread_delete_extension thread_delete;
    User_extensions_thread_switch_extension thread_switch;
    User_extensions_thread_begin_extension thread_begin;
    User_extensions_thread_exitted_extension thread_exitted;
    User_extensions_fatal_extension fatal;
} User_extensions_Table;
```

User_extensions_Switch_control 的定义

```
typedef struct {
    Chain_Node          Node;
    User_extensions_thread_switch_extension thread_switch;
} User_extensions_Switch_control;
```

关键实现函数

API 扩展管理相关函数

API 扩展管理相关函数包括：

- ❏ _API_extensions_Initialization: 对链表_API_extensions_List 初始化
- ❏ _API_extensions_Add: 在链表_API_extensions_List 上增加扩展点
- ❏ _API_extensions_Run_predriver: 驱动程序初始化前的 hook 函数
- ❏ _API_extensions_Run_postdriver: 驱动程序初始化后的 hook 函数
- ❏ _API_extensions_Run_postswitch: 上下文切换后的 hook 函数

用口口展管理初始化

函数调用信息

```
RTEMS_INLINE_ROUTINE void _User_extensions_Handler_initialization (
    uint32_t          number_of_extensions, //用户扩展的个数
    User_extensions_Table *initial_extensions //初始的用户扩展表
)
```

函数功能描述

函数 `_User_extensions_Handler_initialization` 用于完成对用户扩展的管理结构的初始化，在函数 `rtems_initialize_executive_early` 中有下面的代码：

```
_User_extensions_Handler_initialization(
    configuration_table->number_of_initial_extensions,
    configuration_table->User_extension_table
);
```

这个函数的两个参数的含义分别是初始的用户扩展个数和初始的任务扩展表的指针，这两个值都可以在系统的配置表中定义。

`_User_extensions_Handler_initialization` 的具体实现如下：

1. 初始化全局变量 `_User_extensions_List` 和 `_User_extensions_Switches_list`
2. 如果初始的用户扩展数不为0，则调用 `_Workspace_Allocate_or_fatal_error` 分配 `number_of_extensions` 个 `User_extensions_Control` 结构体。
3. 调用 `_User_extensions_Add_set` 把初始配置的用户扩展表加入到相应的 `User_extensions_Control` 结构体中。

添加用口口展

函数调用信息

```
RTEMS_INLINE_ROUTINE void _User_extensions_Add_set (
    User_extensions_Control *the_extension, //需要添加的用户扩展
    User_extensions_Table *extension_table //用户扩展集
)
```

函数功能描述

函数 `_User_extensions_Add_set` 用于把一个用户扩展集加到系统的用户扩展链表上。

具体实现如下：

1. 把用户扩展集 `extension_table` 赋给 `the_extension` 的 `Callouts` 成员
2. 把 `the_extension` 加入到 `_User_extensions_List` 链表上
3. 判断扩展集的 `thread_switch` 成员是否为空，如果不为空则把 `thread_switch` 赋给

`the_extension` 的 `Switch` 成员的 `thread_switch`，并把 `Switch` 成员加到 `_User_extensions_Switches_list` 链表上。

上面用户扩展初始化中增加的用户扩展集是静态配置的，在系统退出运行前不可删除。此外 RTEMS 还提供了动态增加用户扩展集的接口，包括：

- `rtems_extension_create` - 建立一个扩展集
- `rtems_extension_ident` - 获取扩展集的ID
- `rtems_extension_delete` - 删除一个扩展集

和静态配置的用户扩展集一样，动态创建的用户扩展也会加到 `_User_extensions_List` 链和 `_User_extensions_Switches_list` 链上。这些用户扩展之所以能够得到运行，是因为在创建任务、任务切换等系统函数的实现中都有一个函数来调用相应的扩展，比如 `_User_extensions_Thread_create` 是创建任务的扩展，这个函数会在 `_Thread_Initialize` 中调用，它会遍历 `_User_extensions_List` 链表，执行每个扩展集的 `thread_create` 函数。任务切换扩展 `_User_extensions_Thread_switch` 会在函数 `_Thread_Dispatch` 中调用，它会遍历 `_User_extensions_Switches_list` 链表，执行每个节点的 `thread_switch` 函数。

用户扩展管理其它函数

除了上述用户扩展管理函数外，还包括一系列用户扩展hook函数：

```
_User_extensions_Thread_begin
_User_extensions_Thread_create
_User_extensions_Thread_delete
_User_extensions_Thread_exitted
_User_extensions_Thread_restart
_User_extensions_Thread_start
_User_extensions_Fatal
```

应用举例

RTEMS 源码中的\${RTEMS_ROOT}/testsuites/sptests/sp07目录中包含了用户扩展的测试用例。这个用例定义了一个用户扩展集

```
rtems_extensions_table Extensions = {
  Task_create_extension, /* task create user extension */
  Task_start_extension, /* task start user extension */
  Task_restart_extension, /* task restart user extension */
  Task_delete_extension, /* task delete user extension */
  NULL, /* task switch user extension */
  NULL, /* begin user extension */
  Task_exit_extension, /* task exited user extension */
  NULL /* fatal error extension */
};
```

其中每个函数的实现就是打印一些和操作相关的信息，在初始任务中会调用 rtems_extension_create 把这个用户扩展集添加系统的用户扩展链表上。它的运行结果如下，其中注释内容非程序运行输出。

```
*** TEST 7 ***
TASK_CREATE - TA1 - created. //Task_create_extension 输出
TASK_CREATE - TA2 - created.
TASK_CREATE - TA3 - created.
TASK_CREATE - TA4 - created.
TASK_START - TA1 - started. //Task_start_extension 输出
TASK_START - TA2 - started.
TASK_START - TA3 - started.
TASK_START - TA4 - started.
TASK_RESTART - TA3 - restarted. //Task_restart_extension 输出
INIT - rtems_task_set_note - set TA1's RTEMS_NOTEPAD_8 to TA1's priority: 04
INIT - rtems_task_set_note - set TA2's RTEMS_NOTEPAD_8 to TA2's priority: 04
<pause>
TA1 - rtems_task_set_priority - get initial priority of self: 04
TA1 - rtems_task_get_note - get RTEMS_NOTEPAD_8 - current priority: 04
TA1 - rtems_task_set_note - set TA2's RTEMS_NOTEPAD_8: 03
TA1 - rtems_task_set_priority - set TA2's priority: 03
TA2 - rtems_task_get_note - get RTEMS_NOTEPAD_8 - current priority: 03
TA2 - rtems_task_set_note - set TA1's RTEMS_NOTEPAD_8: 02
TA2 - rtems_task_set_priority - set TA1's priority: 02
TA1 - rtems_task_get_note - get RTEMS_NOTEPAD_8 - current priority: 02
TA1 - rtems_task_set_note - set TA2's RTEMS_NOTEPAD_8: 01
TA1 - rtems_task_set_priority - set TA2's priority: 01
TA2 - rtems_task_set_priority - set TA1's priority: 02
TA1 - rtems_task_get_note - get RTEMS_NOTEPAD_8 - current priority: 02
TA1 - rtems_task_set_note - set TA2's RTEMS_NOTEPAD_8: 01
```



```

TA1 - rtems_task_set_priority - set TA2's priority: 01
TA2 - rtems_task_get_note - get RTEMS_NOTEPAD_8 - current priority: 01
TA2 - rtems_task_suspend - suspend TA1
TA2 - rtems_task_set_priority - set priority of TA1 ( blocked )
TASK_DELETE - TA2 deleting TA1      // Task_delete_extension 输出
TASK_DELETE - TA2 deleting TA3
TASK_DELETE - TA2 deleting TA2
TA4 - exiting task
RTEMS_TASK_EXITTED - extension invoked for TA4 //Task_exit_extension
*** END OF TEST 7 ***

```

RMS 实时调度

RTEMS 支持速率单调调度算法 RMS (Rate-Monotonic Scheduling algorithm), 这个算法是1973年 Liu 和 Layland 在 ACM 上发表的题为 “Scheduling Algorithms for Multitprogramming in Hard Real-Time Environment” 的论文中提出的。

RMS 是在基于以下假设的基础上进行分析的：（参考文献：嵌入式实时操作系统及应用开发 第二版）

1. 所有任务都是周期任务
2. 任务的相对截止时间等于任务的周期
3. 任务在每个周期内的计算时间都相等，保持一个常量
4. 任务之间不进行通信，也不需要同步
5. 任务可以在计算的任何位置被抢占，不存在临界区

RMS 是一个静态的固定优先级调度算法，任务的优先级与任务的周期表现为单调函数关系，任务周期越短，任务的优先级越高；任务周期越长，任务的优先级越低。RMS 是静态调度中的最优调度算法，即如果一组任务能够被任何静态调度算法所调度，则这些任务在 RMS 下也是可调度的。这里的静态是指任务的优先级是事先确定的，在任务运行过程中是不可改变的。

需要注意，在 RTEMS 当中，RMS 是作为一个单独的模块（RTEMS API）来实现的。其主要代码集中在 cpukit/rtems/src/ratemon*. *文件中，在此，RMS 是作为一个单独的 RTEMS API 小模块存在的，而不是作为 RTEMS 的最核心的代码 cpukit/score 的一部份。究其原因，主要是由于 RMS 的实现不会对核心层的 thread handler 产生影响，而 thread handler 负责完成基于优先级的调度，这样 RMS 实际上完全可以在 thread handler 的基础上实现。这也就没有必要在 RTEMS 核心层实现了。

关⌋数据⌋构

在 RTEMS 当中，当一个任务（线程）要用到 RMS 进行周期性调度时，它会在其线程执行的代码里声明一个和它相关联的 RMS 管理器。这个 RMS 管理器（Rate Monotonic Control）是 RTEMS 当中实现 RMS 的核心数据结构。它包含了它所关联的线程进行周期性调度的全部信息。该数据结构定义如下：

```

typedef struct {
Objects_Control Object;      // 结构体对应的对象
Watchdog_Control Timer;      //和这个结构体关联的定时器
rtems_rate_monotonic_period_states state; //RMS 的状态
uint32_t owner_ticks_executed_at_period; //当本周期开始时，关联线程执行了多少 tick
uint32_t time_at_period;     //本周期开始时的系统 tick 时间
uint32_t next_length;        //下一个周期的时长
Thread_Control *owner;       //和这个 Rate_monotonic_Control 结构关联的线程 TCB
} Rate_monotonic_Control;

```

关于 RMS 的状态：

```
typedef enum {  
RATE_MONOTONIC_INACTIVE, /* 对应线程还没有开始周期性任务 */  
RATE_MONOTONIC_OWNER_IS_BLOCKING, /* 对应线程要等待这个 RMS 唤醒它 */  
RATE_MONOTONIC_ACTIVE, /* 对应线程正在执行周期性任务 */  
RATE_MONOTONIC_EXPIRED_WHILE_BLOCKING,  
/* 对应线程正要等待这个 RMS 唤醒它，但是在还没有完全做好准备工作的  
时候，RMS 的计时器已经超时了，这是个中间状态，仅仅会短暂存在 */  
RATE_MONOTONIC_EXPIRED  
/* 对应线程开始执行了周期性任务，但是没有在所给的周期内完成（  
到达下一句 rtems_rate_monotonic_period），RMS 计时器超时 */  
} rtems_rate_monotonic_period_states;
```

关□□□函数

和其他对象管理一样，RTEMS 在启动的时候也要调用 `_Rate_monotonic_Manager_initialization` 来初始化 RMS 的管理器。这个函数只是调用 `_Objects_Initialize_information` 根据系统配置表来初始化 `_Rate_monotonic_Information`。

对于 RMS，RTEMS 提供了以下几个 API：

- ✎ `rtems_rate_monotonic_create` 创建一个 RMS
- ✎ `rtems_rate_monotonic_cancel` 取消一个 RMS，暂停其周期性调度
- ✎ `rtems_rate_monotonic_delete` 删除一个 RMS，永久取消其周期性调度
- ✎ `rtems_rate_monotonic_get_status` 获得 RMS 的状态情况
- ✎ `rtems_rate_monotonic_ident` 通过名字(name)获得 RMS 的对象 ID
- ✎ `rtems_rate_monotonic_period` 核心函数，用于启动一个周期，在这里发生 RMS 状态转换，以及关联线程的等待。
- ✎ `_Rate_monotonic_Timeout` 核心函数，当 RMS 定时器超时采取的操作，在这里也发生 RMS 状态转换。

RTEMS 中的 RMS 算法实现大致是：程序调用 `rtems_rate_monotonic_period`，此时系统认为线程开始做一项周期性的任务，并且其周期长度为 `length`，计这一时刻为 `time_at_period`。于是，当线程的执行流到达下一个 `rtems_rate_monotonic_period` 语句时（在通常的程序中，应该是回到之前调用的那句 `rtems_rate_monotonic_period`，这样体现出“周而复始”），系统会判断从 `time_at_period` 到当前时刻是否经过了 `length` 个 tick，如果已经超过了周期的时间，那么就重新开始一个周期；通常情况下，应该是还没有超过 `length` 个 tick，此时，这个线程会被阻塞，等待下一个周期开始，直到系统时刻确实到达 `time_at_period+length` 这个时刻的时候，这个线程才会被唤醒，开始一个新的周期。而这个计算超时以及唤醒线程的任务，是由 `_Rate_monotonic_Timeout` 来完成的。

创建一个 RMS 对象

函数调用信息

```
rtems_status_code rtems_rate_monotonic_create(  
  rtems_name name,           //用户指定的 RMS 对象的名字  
  Objects_Id *id             //存入返回 ID 的内存地址
```

```
)
```

函数功能描述

函数 `rtems_rate_monotonic_create` 用于创建一个 RMS 对象。

具体实现如下：

1. 检查 `name` 和 `id` 的合法；
2. 屏蔽调度；
3. 调用 `_Rate_monotonic_Allocate` 分配一个 `Rate_monotonic_Control` 结构体；
4. 把这个 RMS 的对象的 `owner` 设为 `_Thread_Executing`，把它的 `state` 设为 `RATE_MONOTONIC_INACTIVE`，调用 `_Watchdog_Initialize` 初始化它的 `timer`；
5. 调用 `_Objects_Open` 把这个对象注册到 `_Rate_monotonic_Information` 上；
6. 把这个 RMS 对象的 `id` 存在入口参数指定的地址；
7. 使能调度。

启动一个周期

函数调用信息

```
rtems_status_code rtems_rate_monotonic_period(  
  Objects_Id      id,           //RMS 对象的 ID  
  rtems_interval  length       //下一个周期的长度  
)
```

函数功能描述

函数 `rtems_rate_monotonic_period` 启动一个新的运行周期

具体实现如下：

1. 调用 `_Rate_monotonic_Get` 获取 RMS 的控制块结构和它的位置；
2. 如果这个对象在当前 CPU 上，判断 `length` 是否为 0，如果是 0，则根据 RMS 的状态返回相应的值；
3. 如果不是 0，根据原来的状态采取相应的操作；
4. 如果原来是 `RATE_MONOTONIC_INACTIVE` 状态，表明是创建后还没有开始周期任务，则把状态设置为 `RATE_MONOTONIC_ACTIVE`，然后调用 `_Watchdog_Initialize` 初始化这个 RMS 结构的定时器，并指定到时处理函数为 `_Rate_monotonic_Timeout`，然后再初始化 RMS 结构的其他成员，把 `owner_ticks_executed_at_period` 设置成 TCB 中的 `ticks_executed`，即执行的 tick 数，把 `time_at_period` 设置为 `_Watchdog_Ticks_since_boot`，把 `next_length` 设置为 `length`，最后调用 `_Watchdog_Insert_ticks` 设置定时器的到时时间为 `length`，把定时器另入到系统的定时器链表中开始计时；
5. 如果原来是 `RATE_MONOTONIC_ACTIVE` 状态，也就是说这个周期没有结束的时候任务已经完成，则把 RMS 的状态设成 `RATE_MONOTONIC_OWNER_IS_BLOCKING`，设置 `next_length` 为 `length`，调用 `_Thread_Set_state` 把任务的状态设为 `STATES_WAITING_FOR_PERIOD`，这个函数会使线程阻塞起来；
6. 如果原来是 `RATE_MONOTONIC_EXPIRED` 状态，表明执行 `rtems_rate_monotonic_period` 时已经超过一个周期，它会把开始一个新的周期，具体的操作和原来是 `RATE_MONOTONIC_INACTIVE` 基本相同，只不过不需要再注册超时处理函数。

RMS 的超时处理

函数调用信息

```
void _Rate_monotonic_Timeout(  

```

| | |
|----------------|--------------|
| Objects_Id id, | //RMS 对象的 ID |
| void *ignored | |
|) | |

函数功能描述

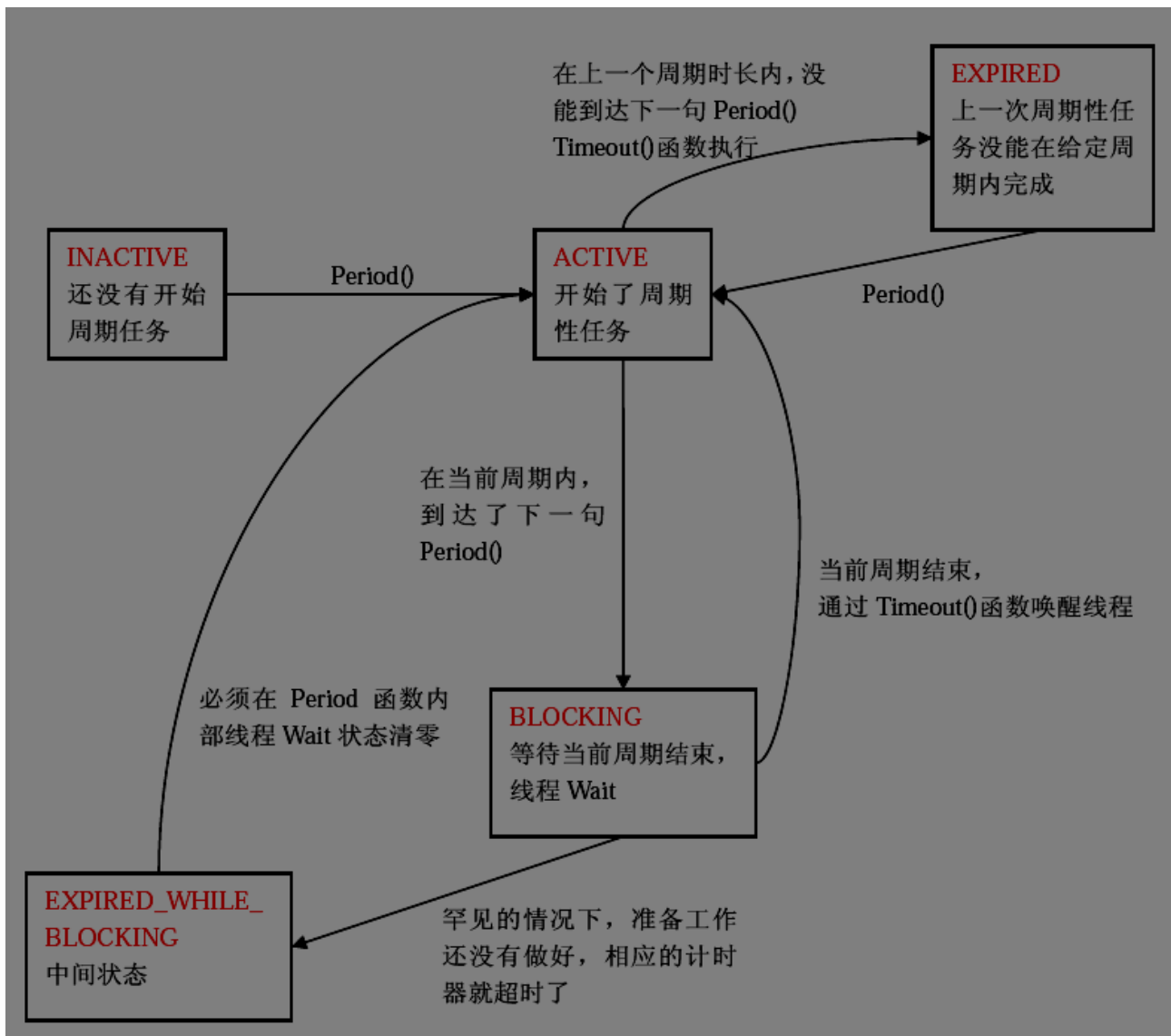
函数 `Rate_monotonic_Timeout` 是一个周期到达时的处理函数，在上一个周期内任务可能已经完成预定的工作在等待下一个周期的到来，也可能没有完成，即出现超时。

具体实现如下：

1. 调用 `_Rate_monotonic_Get` 获取 RMS 的控制块结构和它的位置；
2. 如果这个对象在当前 CPU 上，根据 RMS 的 `owner` 找到对应的任务，根据任务的状态采取相应的操作；
3. 如果原来是 `STATES_WAITING_FOR_PERIOD` 状态，表明是任务已经执行到等待下一个周期的到来，则调用 `_Thread_Unblock` 使任务处理就绪态，重新设置 RMS 结构：把 `owner_ticks_executed_at_period` 设置成 TCB 中的 `ticks_executed`，把 `time_at_period` 设置为 `_Watchdog_Ticks_since_boot`，把 `next_length` 设置为 `length`，最后调用 `_Watchdog_Insert_ticks` 设置定时器的到时时间为 `length`，把定时器另入到系统的定时器链表中开始新的周期的计时；
4. 如果原来是 `RATE_MONOTONIC_OWNER_IS_BLOCKING` 状态，会把 RMS 的状态设置为 `RATE_MONOTONIC_EXPIRED_WHILE_BLOCKING`，其他操作与上面相同；
5. 如果不是以上两种情况，则说明任务超时，即在这个周期内没有完成任务，会把 RMS 的状态设置成 `RATE_MONOTONIC_EXPIRED`。

RMS 的状□□□

RMS 工作的状态图如下：



□用□例

RTEMS 源码中的 `{RTEMS_ROOT}/testsuites/sptests/sp32` 目录中包含了 RMS 的测试用例。它的功能就是让任务运行5个确定的周期，在每个周期开始（上一个周期结束）和结束（下一个周期开始）的时刻记录时间，然后把这两个时间相减，看是否和事先指定的周期长度一样。每个周期指定的时间保存在数组 `wantintervals[5] = { 1, 50, 200, 25, 3 }`，运行时获取的每个周期的开始结束时间保存在数组 `timestamps[6]` 中，它的具体实现如下：

1. 调用 `rtems_rate_monotonic_create` 创建一个 RMS 对象，把它的 ID 保存在 `period_id` 中；
2. 调用 `rtems_rate_monotonic_period` 开始第一个周期，第一个周期的长度为 `wantintervals[0]`；
3. 调用 `rtems_clock_get` 获取系统时间，保存在 `timestamps[0]` 中；
4. 接下来进行4次循环（`loopy=1...4`），循环体中的工作是：
 - 调用 `rtems_rate_monotonic_period` 开始一个新的周期，周期长度为 `wantintervals[loopy]`；
 - 调用 `rtems_clock_get` 获取系统时间，保存在 `timestamps[loopy]` 中；
5. 调用 `rtems_rate_monotonic_period` 开始新的周期，周期长度为1, 调用 `rtems_clock_get` 获取最后一个周期的结束时间，最后这个周期并不是指定的五个周期以内的，只是为了获得最后一个周期的结束时间，因为如果任务在一个周期没有结束的时候遇到了 `rtems_rate_monotonic_period`，就会等待；

6. 调用 `rtems_rate_monotonic_cancel` 和 `rtems_rate_monotonic_delete` 取消和删除 RMS 对象；
7. 打印出每个周期的开始和结束时间的差值和预定的这个周期的长度；

这个任务每个周期内做的事情就是获得系统时间，然后就会执行到 `rtems_rate_monotonic_period` 函数，就会阻塞起来，等待下一个周期的开始。它的运行结果如下：

```
*** TEST 32 ***
period 0: measured 1 tick(s), wanted 1
period 1: measured 50 tick(s), wanted 50
period 2: measured 200 tick(s), wanted 200
period 3: measured 25 tick(s), wanted 25
period 4: measured 3 tick(s), wanted 3
*** END OF TEST SP32 ***
```

第六章 中断处理

概述

操作系统需要对计算机系统中的各种外设进行管理，这就需要 CPU 和外设能够相互通信才行。一般外设的速度远慢于 CPU 的速度。如果让操作系统通过 CPU “主动关心” 外设的事件，即采用通常的轮询（polling）机制，则太浪费 CPU 资源了。所以需要操作系统和 CPU 能够一起提供某种机制，让外设需要操作系统处理外设相关事件的时候，能够“主动通知”操作系统，即打断操作系统和应用的正常执行，让操作系统完成外设的相关处理，然后在恢复操作系统和应用的正常执行。在操作系统中，这种机制称为中断机制。中断机制给操作系统提供了处理意外情况的能力，同时它也是实现进程/线程抢占式调度的一个重要基石。本章主要讲解 RTEMS 中断处理的设计与实现，涉及中断和异常的介绍、中断的建立过程、中断的处理过程、中断的机制、中断上下文、嵌套中断等。在下面的各节中，我们将基于 PC 386来讲述 RTEMS 的中断处理。

中断与异常

在通用操作系统中，把三种特殊的中断事件。由 CPU 外部设备引起的外部事件如 I/O 中断、时钟中断、控制台中断等是异步产生的，与 CPU 的执行无关，我们称之为异步中断（asynchronous interrupt）也称外部中断，简称中断（interrupt）。而把在 CPU 执行指令期间检测到不正常的或非法的条件（如除零错、地址访问越界）所引起的内部事件称作同步中断（synchronous interrupt），也称内部中断，简称异常（exception）。把在程序中使用请求系统服务的系统调用而引发的事件，称作陷入中断（trap interrupt），也称软中断（soft interrupt），简称 trap。在后面的叙述中，如果没有特别指出，我们将用简称中断、异常、陷入来表示这三种特殊的中断事件。在 RTEMS 操作系统中，应用访问操作系统直接通过函数调用实现，所以没有陷入这种情况。下面的讲述主要以中断和异常为主。

i386中断

对于基于 PC 386的计算机系统，i386 CPU 有两根引脚 INTR 和 NMI 接受外部中断请求信号。INTR 接受可屏蔽中断请求。NMI 接受不可屏蔽中断请求。在 i386 CPU 中的标志寄存器 EFLAGS 的 IF 标志决定是否屏蔽可屏蔽中断请求。

当外部硬件在通过 INTR 发出中断请求信号的同时，还要向处理器给出一个8位的中断向量。i386 CPU 在响应可屏蔽中断请求时，读取这个由外部硬件给出的中断向量号。i386 CPU 并没有指定这个中断向量号。但系统必须通过软件和硬件的配合设置，使得给出的这个中断向量号不仅与外部中断源对应，而且要避免潜在的中断向量号冲突。i386 CPU 通过可编程中断控制器芯片8259A 来处理中断向量号。每个 8259A 芯片可以支持8路中断 请求信号，如果使用2个8259A 芯片（一个主片，一个从片），就可使 i386 CPU 在单个引脚 INTR 上接受多达15个中断源的中断请求信号。

处理器不屏蔽来自 NMI 的中断请求。处理器在响应 NMI 中断时，不从外部硬件接收中断向量号。在 i386 中，不可屏蔽中断所对应的中断向量号固定为2。为了实现不可屏蔽中断的嵌套，每当接受一个 NMI 中断，处理器就在内部屏蔽了再次响应 NMI，这一屏蔽过程直到 执行中断返回指令 IRET 后才结束。所以，NMI 处理程序应以 IRET 指令结束。

对于计算机来说，外设硬件通过电信号传播的方式把信息送给中断控制器，然后在由中断控制器向 CPU 发电信号，CPU 收到中断控制器发来的信号后，马上打断直接当前的工作，进入中断处理的准备阶段。而具体如何处理这个中断，必须得靠操作系统的中断服务例程来实现，操作系统支持所有事先能够预料到的中断信号的处理。下面我们来看看中断等级和中断服务例程。

中断等口

许多处理器支持不同等级的中断（或称为中断优先级），在 RTEMS 中，逻辑上支持256个中断等级，这些中断等级被映射到处理器的实际的中断等级上。对于中断如何映射到实际的处理器中断等级上，需要参考芯片文档。

屏蔽与使能中断

在执行系统调用的时候，有些关键代码是不应该被中断打扰的。当进入这些代码段的时候，需要尽可能的屏蔽外部中断，当关键代码执行完后，RTEMS 恢复外部中断。RTEMS 已经优化了中断屏蔽和恢复代码，所以在屏蔽和恢复中断上不会消耗太多时间。RTEMS 能够屏蔽中断的时间和目标处理器有一定的关系，需要阅读对应的处理器手册。

不可屏蔽中断（NMI）不能被屏蔽掉，同时，在不可屏蔽中断等级上的 ISR 不应该使用 RTEMS 系统调用。否则，将会有不可预料的后果。这主要是因为 RTEMS 的系统调用中没有屏蔽外部中断。

虽然 rtems 支持256级的中断嵌套，但是对于 i386来说只有使能中断，和屏蔽中断两种状态。全局中断函数 `_CPU_ISR_Disable(_level)` 和 `_CPU_ISR_Enable(_level)` 负责全局的使能/屏蔽中断，这实际上是通过 `cli` 和 `sti` 指令来实现的。在 PIC 层次(8259级别)的针对某条中断线屏蔽中断的函数是 `BSP_irq_disable_at_i8259s` 函数，在 PIC 层次(8259级别)的针对某条中断线使能中断的函数是 `BSP_irq_enable_at_i8259s` 函数，这都是通过对特定 I/O 地址写0或1来完成的。这两个函数的实现比较简单，在此不再进一步详述。

中断服务例程

当 CPU 把中断信号变成一个数字（我们称为中断号）后，操作系统会根据中断号执行对应的中断服务函数，该函数叫做中断服务例程（interrupt handler）或中断服务例程（interrupt service routine(ISR)）。产生中断的每个设备都有相应的中断服务例程。例如，由一个函数专门处理来自系统时钟的中断，而另外一个函数专门处理由键盘产生的中断。

这实际上都是通过中断号来区分不同的中断服务例程的。

中断服务例程不是一开始就存在的，需要操作系统做一系列的准备工作。RTEMS 是如何初始化其中断处理组件和完成对服务程序的管理的呢？

中断初始化

在 RTEMS 中，为完成中断处理，首先需要建立中断向量表。以 PC386为例。在系统启动时从 `start.S` 开始执行，它会调用 `ldsegs.S` 中的 `_load_segments` 函数，它会把定义的全局描述符表的地址和长度加载到 GDT 寄存器，把中断向量表加载的地址和长度加载到 IDT 寄存器，这里只是定义了256个全为空的中间描述符。`start.S` 在完成初始化后会跳到 `bsp_start` 函数，对于 pc386来说就是 `bsp_start_default` 函数，它又会调用 `rtems_irq_mngt_init` 函数来完成中断初始化。

执行流程

`rtems_irq_mngt_init` 函数的执行流程如下所示：

1. 获取 IDT 中保存的中断描述符表的地址和长度；
2. 用 `cli` 指令来屏蔽中断；
3. 用 `defaultRawIrq` 静态变量中包含的 `default_raw_idt_handler` 函数（定义在 `i386BSP` 的 `irq_asm.S` 中）来初始化256个 IDT 项；
4. 然后再用16个 `rtemsIrq` 数组变量中包含的 `rtems_irq_prologue_XX` 函数（定义在 `i386BSP` 的 `irq_asm.S` 中）来初始化前16个 IDT 项；
5. 通过初始化属于 `rtems_irq_global_settings` 结构体的 `initial_config` 静态变量来进行中断管理配置的初始化工作；

6. 调用 BSP_rtems_irq_mngt_set 函数，此函数会根据优先级表 irqPrioTable 计算8259的掩码，然后判断各个中断服务例程是否是默认的，如果不是就使能此中断线，并执行 on 函数，如果是就执行 off 函数，并屏蔽此中断线。

rtems_irq_mngt_init 函数是完成 PC386中断初始化的关键函数，其中有对部分相关的关键变量的操作。下面就对部分关键的数据结构和变量进行说明。

关键数据结构

中断配置数据结构和变量

RTEMS 386把中断分成了两个层次，一个是属于 CPU 级别的所有 IDT 中断向量（带用“raw”的），还有就是实际外设的中断（由8259控制）。针对这两个层次，分别定义了 rtems_raw_irq_global_settings 结构体和 rtems_irq_global_settings 结构体以及相关的变量。

rtems_raw_irq_global_settings 结构体的定义如下：

```
typedef struct {
    unsigned int          idtSize; //IDT 项的个数
    rtems_raw_irq_connect_data defaultRawEntry; //没有连接中断时的缺省中断处理数据
    rawIrqHdlTbl*         rawIrqHdlTbl; //包含初始中断处理数据的数组
} rtems_raw_irq_global_settings;
```

rtems_irq_global_settings 结构体的定义如下：

```
typedef struct {
    unsigned int          irqNb; //中断项的个数
    rtems_irq_connect_data defaultEntry; //没有连接中断时的缺省中断处理数据
    rtems_irq_connect_data* irqHdlTbl; //包含初始中断处理数据的数组
    rtems_irq_number       irqBase; //第一个中断线对应的中断向量
    rtems_irq_prio*        irqPrioTbl; //中断优先级表
} rtems_irq_global_settings;
```

RTEMS 对于 pc386定义了16个外设中断，从0x20开始，所以 BSP_ISA_IRQ_VECTOR_BASE=0x20。基于 rtems_raw_irq_global_settings 结构体的静态变量为 raw_initial_config，基于 rtems_irq_global_settings 结构体的静态变量为 initial_config，在函数 rtems_irq_mngt_init 中继续初始化。

中断描述数据口构和变量

rtems_raw_irq_connect_data 结构体与硬件相关，是针对 PC386 BSP 的 CPU 级的中断描述的数据结构，其定义如下所示：

```
typedef struct __rtems_raw_irq_connect_data__ {
    rtems_vector_offset    idtIndex; //IDT 向量偏移量 (IRQ line +
PC386_IRQ_VECTOR_BASE)
    rtems_raw_irq_hdl      hdl; //对应的中断服务例程
    rtems_raw_irq_enable   on; //使能中断的函数（在设备级和 PIC 级）
    rtems_raw_irq_disable  off; //屏蔽中断的函数（在设备级和 PIC 级）
    rtems_raw_irq_is_enabled isOn; //判断中断使能还是屏蔽的函数
} rtems_raw_irq_connect_data;
```


基于此结构体的静态变量数组为 `idtHdl[IDT_SIZE]`，描述了初始化阶段要对 IDT 项初始化所需要的数据。`idtHdl[IDT_SIZE]` 的初始值为 `defaultRawIrq` 的值，其定义如下：

```
static rtems_raw_irq_connect_data    defaultRawIrq = {
    /* vectorIdx, hdl, on, off      , isOn */
    0,    default_raw_idt_handler , nop_func , nop_func,    not_connected
};
```

`rtems_irq_connect_data` 结构体是通常情况下（与 CPU 无关）的中断描述的数据结构，其定义如下所示：

```
typedef struct __rtems_irq_connect_data__ {
    rtems_irq_number    name; //中断线号
    rtems_irq_hdl       hdl; //对应的中断服务例程
    rtems_irq_hdl_param handle; //对应的中断服务例程的参数
    rtems_irq_enable    on; //使能中断的函数
    rtems_irq_disable   off; //屏蔽中断的函数
    rtems_irq_is_enabled isOn; //判断中断使能还是屏蔽的函数

#ifdef BSP_SHARED_HANDLER_SUPPORT
    void *next_handler; //对共享中断的处理函数
#endif
} rtems_irq_connect_data;
```

基于此结构体的静态变量数组为 `rtemsIrq[BSP_IRQ_LINES_NUMBER]`，`rtemsIrq[BSP_IRQ_LINES_NUMBER]` 的初始值为 `defaultIrq`，其定义如下：

```
static rtems_irq_connect_data    defaultIrq = {
    /* vectorIdx, hdl , handle , on , off , isOn */
    0,    nop_func , 0 , nop_func , nop_func , not_connected
};
```

可以看出 `rtems_irq_connect_data` 结构体与 `rtems_raw_irq_connect_data` 结构体没有大的区别，只是多了一个中断服务例程的参数。如果用 `defaultIrq` 来初始化前 16 个 IDT 项，其实没有用处（因为用的是空函数 `nop_func`）。而真实的中断服务例程 `rtems_irq_prologue_XX`（ $0 \leq XX \leq 15$ ）定义在属于 `rtems_irq_connect_data` 结构体的 `rtemsIrq[BSP_IRQ_LINES_NUMBER]` 数组变量中。具体的处理在函数 `BSP_rtems_irq_mngt_set` 中完成。`rtemsIrq[BSP_IRQ_LINES_NUMBER]` 的定义如下：

```
static rtems_irq_connect_data    rtemsIrq[BSP_IRQ_LINES_NUMBER] = {
    {0, (rtems_irq_hdl)rtems_irq_prologue_0},
    .....
    {0, (rtems_irq_hdl)rtems_irq_prologue_15}
};
```

而 `rtems_irq_prologue_XX` 这些函数定义在 `libbsp/i386/shared/irq_asm.S` 里，具体定义如下：

```
#define DISTINCT_INTERRUPT_ENTRY(_vector) \
    .p2align 4                                ; \
    PUBLIC (rtems_irq_prologue_ ## _vector) ; \
SYM (rtems_irq_prologue_ ## _vector):        \
    pushl    eax                            ; \
    pushl    ecx                            ; \
    pushl    edx                            ; \
    movl     $_vector, ecx                 ; \
    jmp      SYM (_ISR_Handler) ;
```

```
DISTINCT_INTERRUPT_ENTRY(0)
.....
DISTINCT_INTERRUPT_ENTRY(15)
```

通过上述定义可以看出，`rtems_irq_prologue_XX` 函数都是把通用寄存器 `eax, ecx, edx` 压栈，然后把中断向量号放到 `ecx` 里，再跳转到 `_ISR_Handler`，可见它是所有中断的公共入口。

中断处理机制的实现

根据上面的分析，我们可以知道，从0~15号中断，其中断入口都为 `_ISR_Handler` 函数。此函数实现来 `irq_asm.S` 文件中，下面是此函数的大致执行流程：

1. 首先把 `i8259s` 的中断掩码 `i8259s_cache` 压栈；
2. 然后根据 `irq_mask_or_tbl` 计算出新的中断掩码，并把新的掩码写到8259的中断掩码寄存器里；
3. 向8259发送中断响应命令；
4. 如果没有中断嵌套，则把栈指针 `ESP` 切换到中断专用的堆栈，否则继续下面的处理；
5. 给 `_ISR_Nest_level` 加1,表明中断嵌套层次，给 `_Thread_Dispatch_disable_level` 加1,用来阻止线程调度；
6. 通过 `sti` 允许中断；
7. 把中断号做为下标，找到 `rtems_hdl_tbl` 数组对应的数组项，根据数组项的内容建立具体的中断服务函数地址和参数；
8. 调用具体的中断服务函数；
9. 中断服务函数返回后，屏蔽中断；
10. 恢复以前的堆栈；
11. 恢复8259的中断掩码；
12. 给 `_ISR_Nest_level` 减1,给 `_Thread_Dispatch_disable_level` 减1；
13. 判断是否需要调度，以及是否有信号需要处理，如果有则调用相应的处理函数；
14. 恢复相关寄存器，通过 `iret` 返回。

`irq_mask_or_tbl` 是系统定义的全局一个数组，它的每个元素对应于一个外设中断发生时的中断掩码，是一个16位的 `short` 类型。系统还定义了 `irqPrioTable`，其定义如下：

```
static rtems_irq_prio irqPrioTable[BSP_IRQ_LINES_NUMBER]={
    0,0, 255,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
```

如果是0表明只有当前中断被屏蔽，如果是255，所以中断都被屏蔽，为了保证8259的 `slave pic` 总是不被屏蔽，所以 `irqPrioTable[2]=255`，这是因为从片接在主片的2号中断线上。

一个外设的的实际的中断处理是在 `rtems_hdl_tbl` 中的注册的中断服务例程中完成的。在系统初始化以后这些例程都是空函数，而这个驱动程序的初始化程序会注册实际的中断服务例程。

注册中断处理服务例程

对于某个特定的驱动程序，会包含相应的中断服务例程，那么这个具体的中断服务例程是如何与具体的中断号绑定在一起的呢？在 `PC 386 BSP` 中，这实际上是通过 `BSP_install_rtems_irq_handler` (`rtems_irq_connect_data* irq`) 函数来完成这个绑定的。看此函数的名字就可以看出此函数是与硬件 `BSP` 相关的。这个函数就是用来向 `rtems_hdl_tbl` 注册中断的相关处理函数的。此函数的处理流程如下：

1. 判断 `irq->name` 是否是合法的中断，如果不是，就出错返回；
2. 判断 `rtems_hdl_tbl[irq->name].hdl` 是否是 `default_rtems_entry.hdl`，如果不是就出错返回；
3. 把 `rtems_hdl_tbl[irq->name]` 赋值为 `*irq`；
4. 在8259级别（PIC 级）使能中断；
5. 在设备级（device 级）使能中断；

我们可以从时钟中断这个实例来进一步了解它。在 `ckinit.c` 这个文件里的 `Clock_initialize` 调用了 `BSP_install_rtems_irq_handler` 来注册中断，这个函数是时钟驱动的初始化程序，在系统启动时会运行各个设备驱动程序的初始化函数，在各个初始化过程中完成相应的中断服务例程的注册。对于时钟，它是用 `clockIrqData` 这个变量来注册中断的，它的定义如下：

```
static rtems_irq_connect_data clockIrqData = {BSP_PERIODIC_TIMER,
                                              clockIsr,
                                              0,
                                              clockOn,
                                              clockOff,
                                              clockIsOn};
```

时钟是中断行号为0 的中断线，它的中断服务例程是 `clockIsr`，还有相应的 on 函数 `clockOn`，off 函数 `clockOff`，isOn 函数 `clockIsOn`，关于这些函数的具体作用将在分析时钟的部分章节进一步解释。

核心层中断管理

RTEMS 在核心层提供了 ISR handler 组件，用于给系统服务层的中断管理器组件提供服务，在中断管理器组件和 BSP 之间建立一个桥梁。

在 ISR handler 组件中，提供了 ISR handler 初始化函数 `_ISR_Handler_initialization`，其主要工作包括：

- ❏ 分配中断向量表的内存空间
- ❏ 调用 `_CPU_Initialize_vectors` 函数初始化中断向量，在 PC386 BSP 中，此函数为空
- ❏ 分配中断处理过程中的堆栈空间

ISR handler 组件除了完成中断管理初始化工作，还提供了一系列的宏定义帮助连接 BSP 提供的具体中断处理和系统服务层的中断管理器之间的联系。这些宏定义包括：

- ❏ `_ISR_Disable`：屏蔽中断
- ❏ `_ISR_Enable`：使能中断
- ❏ `_ISR_Flash`：暂时使能中断
- ❏ `_ISR_Get_level`：获得当前中断级别
- ❏ `_ISR_Set_level`：设置当前中断级别
- ❏ `_ISR_Install_vector`：注册某个中断向量的中断服务例程，对于 PC386 BSP 而言，等同于调用 BSP 提供的 `_CPU_ISR_install_vector` 函数
- ❏ `_ISR_Is_in_progress`：当前是否处于中断状态

系统服务层中断管理

RTEMS 在系统服务层中提供了中断管理器组件，用于提供一种快速响应外部中断的机制以满足实时性需求。中断管理器组件负责管理中断向量表，该表允许中断和对应的中断服务例程间形成一对一的映射。当中断产生，CPU 会执行 RTEMS 的对应此中断的中断服务例程。中断的中断服务例程负责处理中断，并操

作外部设备。

系统函数

当然中断管理器中断管理需要得到核心层的 interrupt handler 的支持允许中断服务例程抢占当前任务对处理器的占有权。下面是中断服务例程提供的系统函数。

- ✎ `rtems_interrupt_catch`: 注册中断服务例程 (ISR)，该函数将 RTEMS 的中断服务例程与中断向量表中对应的表项相关连，此函数会进一步调用 ISR handler 组件提供的 `_ISR_Install_vector` 函数
- ✎ `rtems_interrupt_disable`: 屏蔽中断，此宏定义即是 ISR handler 组件提供的 `_ISR_Disable`
- ✎ `rtems_interrupt_enable`: 激活中断，此宏定义即是 ISR handler 组件提供的 `_ISR_Enable`
- ✎ `rtems_interrupt_flash`: 清空中断，此宏定义即是 ISR handler 组件提供的 `_ISR_Flash`
- ✎ `rtems_interrupt_is_in_progress`: 判断处理器是否正在进行中断处理，此宏定义即是 ISR handler 组件提供的 `_ISR_Is_in_progress`

中断服务例程

`rtems_interrupt_catch` 函数将中断服务例程将中断向量和对应的中断服务例程 (ISR) 联系起来。中断向量使用 `rtems_vector_number` 数据类型表示。是中断服务例程的原型如下所示：

```
rtems_isr user_isr(  
rtems_vector_number vector  
);
```

为了尽量减少对低优先级中断的屏蔽时间，中断服务例程应该做得精简高效，可以推后执行的处理应该尽可能的放在上层（服务层或应用层任务）完成。当中断服务例程结束，RTMES 将恢复被中断任务的恢复工作。在处理中断的过程中，可能触发了某些条件导致有高优先级的任务变为就绪态。因此，当 ISR 完成的时候, 就需要进行任务调度与切换。

中断服务的准则

为了任务能在中断后正确的调度，并能保证中断处理的高效，中断处理必须遵循下面的准则：

- 所有的 ISR 必须由中断管理器统一管理。
- ✎ 中断优先级高的任务可以将中断优先级低的任务中断，这称为中断嵌套，RTMES 支持中断嵌套，为了提高中断嵌套的性能，在处理中断嵌套时，嵌套的中断中不会发生任务调度管理，只有当最外层中断处理结束，这些延时的中断处理才回开始执行。

允许调用的系统函数

中断服务例程需要使用 RTEMS 系统调用的时候必须通知 RTEMS。中断服务例程可能使用系统调用与应用层任务间实现同步。中断服务例程也可能使用消息、事件、信号等同步机制。中断服务例程调用的系统函数必须是本地节点上的函数。由于中断服务例程不能阻塞，所以衡量中断服务例程中可以调用的系统函数的重要指标是此函数不能被阻塞。中断服务例程可以调用的系统函数集合：

- ✎ 任务管理: `task_get_note` , `task_set_note` , `task_suspend`, `task_resume`
- ✎ 时钟管理: `clock_get`, `clock_tick`
- ✎ 消息、事件和信号管理: `message_queue_send`, `message_queue_urgent` , `event_send` , `signal_send`
- ✎ 信号量管理: `semaphore_release`
- ✎ 双端内存管理: `port_external_to_internal`, `port_internal_to_external`

- ☞ I/O 管理: `io_initialize` , `io_open` , `io_close` , `io_read` , `io_write`, `io_control`
- ☞ 致命错管理: `fatal_error_occurred`
- ☞ 多处理器: `multiprocessing_announce`

小节

第七章 时间服务

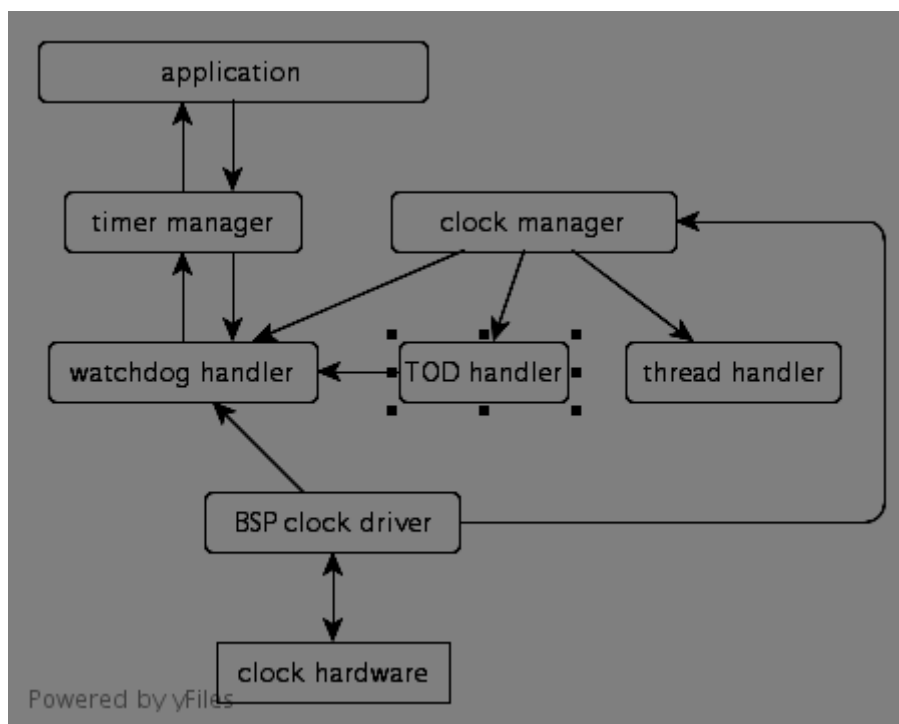
概述

为了得到精确的时间，并在确定时间点和时间段内完成确定的任务，除了在硬件上需要由周期性的时钟中断提供时间信息外，还需要操作系统根据系统时钟完成高效精确的时间服务。对于象 RTEMS 这样的硬实时操作系统而言，它能够对任何时间精确性要求苛刻的事件作出响应，在正确的时间做一个正确的动作。对于基于 RTEMS 的实时应用而言，时间服务在 RTEMS 内核的每个环节都起着关键作用。在 RTEMS 内核中，主要通过定时器管理器 (Timer Manager) 组件来给实时应用提供对日期和其他的与时间相关的服务。

定时器管理器中的核心概念是 `timer` (定时器)，它是指对将来在某个时间所发生事件的一次处理。为了能够让定时器管理器组件能够正常工作，RTEMS 还实现了一系列内核组件来协助定时器管理器组件。它们是：

- ☞ BSP 时钟驱动
- ☞ Watchdog handler 组件
- ☞ TOD handler 组件
- ☞ Clock Manager 组件

它们之间的关系如下图所示：



首先应用程序会通过 `timer manager` 组件创建定时器事件和到期执行例程。当硬件时钟产生中断后，BSP 的 `clock` 驱动程序中的时钟中断服务例程会接收此中断，并调用 `clock manager` 组件的 `rtems_clock_tick` 函数，来分别访问 `watchdog handler`、`TOD handler` 和 `thread handler` 组件，完成对到期事件的处理、当前时间的更新和基于时间片调度的线程时间片更新等工作。当 `watchdog handler` 组件发现有到期事件后，就会执行 `timer manager` 管理的相应的到期执行例程，完成应用程序的需求。

本章将对在核心层和系统服务层的相关组件为提供 RTEMS 时钟服务而进行的设计与实现进行分析，对 timer manager 组件提供的上层时间服务接口进行描述。从而让读者在设计实时应用是能够更好地实用时钟服务。

BSP 层的时钟中断服务例程

PC386 BSP 提供了时钟驱动程序，主要的相关函数在 rtems/c/src/lib/libbsp/i386/pc386/clock/ckinit.c 中实现。

关键数据结构

全局变量

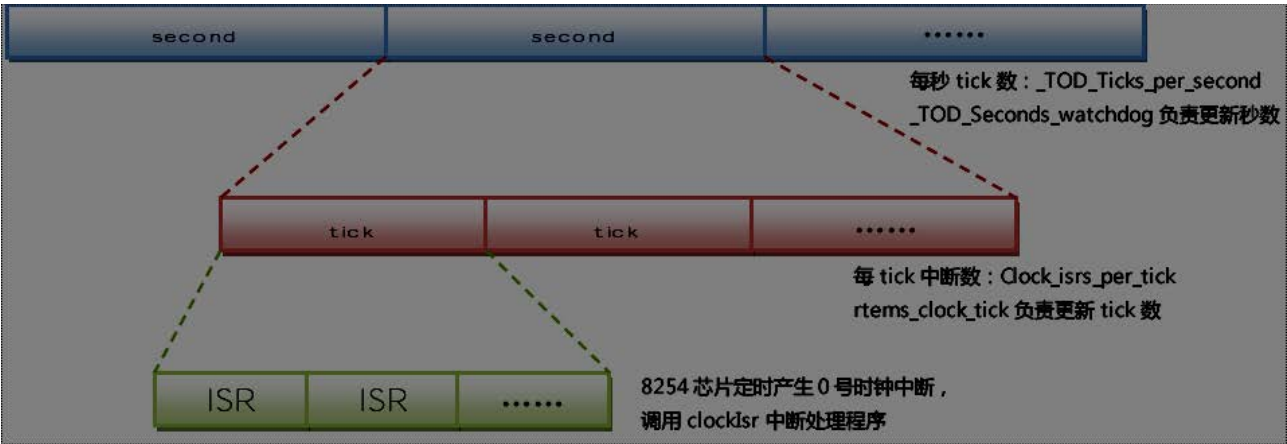
clockIrqData 保存了与时 PC386时钟有关的中断服务例程/使能时钟/屏蔽时钟/判断时钟是否使能的函数指针。其定义如下：

```
static rtems_irq_connect_data clockIrqData = {BSP_PERIODIC_TIMER,
                                             clockIsr,
                                             0,
                                             clockOn,
                                             clockOff,
                                             clockIsOn};
```

下面是与时钟驱动和时钟中断相关的全局变量：

```
volatile uint32_t Clock_driver_ticks; /* Tick (interrupt) counter. */
uint32_t Clock_isrs_per_tick; /* ISRs per tick. */
uint32_t Clock_isrs; /* ISRs until next tick. */
```

实际的定时器脉冲周期可以比一个 tick 单位时间还短，每次时钟中断到来时利用变量 Clock_isrs 计数，当经过了 Clock_isrs_per_tick 次时钟中断后才认为经过了一个 ticks 单位时间，这种机制提供了更加灵活的控制 tick 长度的方法。下图说明了各个计时单位的关系：



关键实现函数

时钟中断初始化函数 Clock_initialize

函数调用信息：

```
rtems_device_driver Clock_initialize(rtems_device_major_number major,
```

```
rtcms_device_minor_number minor,  
void *pargp)
```

函数功能描述:

此函数在系统初始化设备时调用，负责注册时钟中断向量。

时钟中断服务例程 clockIsr

函数调用信息:

```
static void clockIsr()
```

函数功能描述:

此函数是0号中断（时钟中断）的中断服务例程，当新的tick到来时负责调用rtcms_clock_tick函数。rtcms_clock_tick函数的分析在下面有关系统服务层时钟（clock）管理器的小节中。

时钟复位函数 clockOff

函数调用信息:

```
void clockOff(const rtcms_irq_connect_data* unused)
```

函数功能描述:

此函数在RTEMS退出时被调用，主要工作是将时钟复位。

时钟初始化函数 clockOn

函数调用信息:

```
static void clockOn(const rtcms_irq_connect_data* unused)
```

函数功能描述:

此函数初始化8254可编程定时器，调整中断触发周期和每tick中断数使得每秒tick数为系统设定值。

核心层的 watchdog Handler 组件

watchdog 是一种定时器机制，可用于完成对超时的处理。watchdog 机制是 RTEMS 实现“定时发生某个事件”的基础。对这种机制的使用在 RTEMS 中很普遍，其具体实现是通过核心层的 watchdog handler 组件来完成的。睡眠线程的唤醒、由于获取共享对象而阻塞的超时等用到的定时器都是采用 watchdog handler 组件来实现的。RTEMS 在核心层实现了 Watchdog_Control 结构体和相关操作。

从功能角度来看，watchdog 有两个属性：事件将要发生的时间，和要做的事（到期执行例程）。watchdog 总是按照其事件发生时间的顺序，从早到晚排成一条链。从早到晚排列的好处是：每次检查是否有 watchdog 事件发生，只需要检查链首的 watchdog。

在 RTEMS 中有两条 watchdog 链：_Watchdog_Ticks_chain 和 _Watchdog_Seconds_chain。二者的区别在于时间的粒度不同。对于 _Watchdog_Ticks_chain 而言，应该在每个 tick 检查一次链首的 watchdog 事件是否发生。而对于 _Watchdog_Seconds_chain 而言，应该在每秒检查一次链首的 watchdog 事件是否发生。这个“检查”工作实际上是通过调用函数 _Watchdog_Tickle 来进行的。

值得注意的是，每个 watchdog 有一个属性“delta_interval”。在一条 watchdog 链中，链首的 watchdog 的 delta_interval 就记录从当前时刻开始，还要经过多少时间这个 watchdog 事件才会发生。而链中其他的 watchdog 的 delta_interval 就记录从它上一个 watchdog 事件发生的那一时刻开始，还要经过多少时间这个 watchdog 事件才会发生。

另一个需要关注的是，一旦一个 watchdog 事件发生，并被成功地处理之后（执行了到期执行例程），就算是这个事件完成了，然后这个 watchdog 就会被从相应的链上取下。所以如果我们希望做一件周期性的事情，那么在 routine 子程序当中就需要在该干的事情干完后把这个 watchdog 重新加到对应的链里。

下面就分析一下 watchdog handler 组件的关键数据结构和具体实现。

关键数据结构

Watchdog_Control 结构体

Watchdog_Control 结构体的定义在 rtems/cpukit/score/include/rtems/score/watchdog.h 中。

Watchdog_Control 结构体的定义如下：

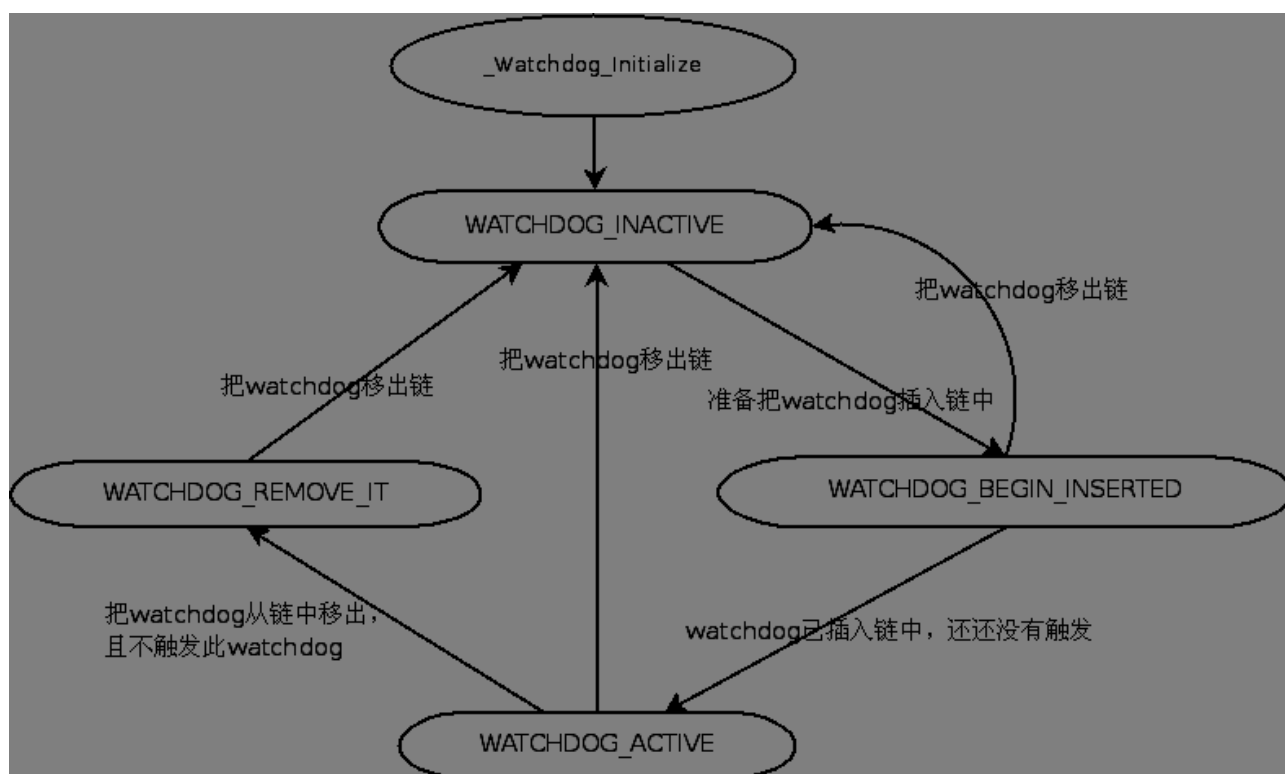
```
typedef struct {
    Chain_Node          Node; //chain node
    Watchdog_States     state; //watchdog 的状态
    Watchdog_Interval   initial; //watchdog 请求发起到定时结束之间的时间间隔
    Watchdog_Interval   delta_interval; //距离定时器链表中的前一定时器到期的时间
    Watchdog_Interval   start_time; //
    Watchdog_Interval   stop_time; //
    Watchdog_Service_routine_entry routine; //到定时时间后要执行的函数
    Objects_Id          id; //传递给函数的参数，当为一个线程创建定时器时这个 id
    //通常就是线程的 id，这样 routine 就可以找到对应的线程并进行相应处理
    void                *user_data; //传递给函数的 watchdog handler 例程的用户参数
} Watchdog_Control;
```

watchdog 的状态

watchdog 在做从创建到消亡的过程中，会经过四种状态，请定义如下：

```
typedef enum {
    WATCHDOG_INACTIVE, //watchdog 还没有挂到链上
    WATCHDOG_BEING_INSERTED, //正在将 watchdog 挂到链上
    WATCHDOG_ACTIVE, //watchdog 在链上，允许被触发
    WATCHDOG_REMOVE_IT //watchdog 在链上，需要把它移出，且不被触发
} Watchdog_States;
```

watchdog 的状态变化如下图所示：



watchdog 的状态变化图

关键实现函数

watchdog handler 初始化

函数调用信息：

```
void _Watchdog_Handler_initialization (void)
```

函数功能描述：

这个函数会在系统启动时调用 `rtems_initialize_executive_early` 时被运行，主要是设置同步信息，把 `_Watchdog_Ticks_since_boot` 置 0，然后初始化两个全局的链表 `Watchdog_Ticks_chain` 和 `_Watchdog_Seconds_chain`，这两个分别是以 tick 为单位和以 second 为单位的计时器链表，系统中所有创建的 watchdog 都会插入到这两个链表中。这里主要分析以 tick 为单位的定时操作，以 second 为单位的情况类似。

创建 watchdog

函数调用信息：

```
void _Watchdog_Initialize(
    Watchdog_Control      *the_watchdog,
    Watchdog_Service_routine_entry routine,
    Objects_Id            id,
    void                  *user_data
)
```

函数功能描述:

这个函数的作用就是根据给定的参数创建一个 watchdog，并把它状态设为 WATCHDOG_INACTIVE 状态。

把 watchdog 插入链表

函数调用信息:

```
void _Watchdog_Insert_ticks(  
    Watchdog_Control    *the_watchdog,  
    Watchdog_Interval    units  
)
```

函数功能描述:

在创建一个 watchdog 后，还需要根据定时时间把它插入到 _Watchdog_Ticks_chain 链表中，_Watchdog_Insert_ticks 就是用 来完成这个操作的，它只是把这个 watchdog 的 initial 设为给定的时间，然后就调用 _Watchdog_Insert 函数，把这个 watchdog 插入到 _Watchdog_Ticks_chain 中。

_Watchdog_Insert 函数中除了一些为了保证同步的操作以外，主要工作是根据 watchdog 的 initial 指定的时间找到插入的位置，链表排列的顺序是越早到期的 watchdog 排得越靠前，并且更新所有 watchdog 的 delta_interval，更新的结果就是链表上的第一个定时器的 delta_interval 是它的到期时间与当前时间的差值，其他 watchdog 的 delta_interval 都是 watchdog 的到期时间与它前一个 watchdog 的到期时间的差值。在找到这个位置以后就可以把这个 watchdog 插入链表。

更新 watchdog 链

函数调用信息:

```
void _Watchdog_Tickle( Chain_Control *header)
```

函数功能描述:

此函数被 _Watchdog_Tickle_ticks 函数封装（实际上二者没有区别），也被 _Watchdog_Tickle_seconds 函数封装。这两个函数的区别是，它们在封装 _Watchdog_Tickle 函数时，输入的参数是不一样的。一个是 _Watchdog_Ticks_chain，一个是 _Watchdog_Seconds_chain。

_Watchdog_Tickle 函数用来在 tick 添加一个单位时，对如 _Watchdog_Ticks_chain 或 _Watchdog_Seconds_chain 进行刷新，检查 WatchDog 链是否有 watchdog 事件发生，如果有 watchdog 到期，就执行此 watchdog 的到期函数。因为有以 tick 和 second 为单位的两种定时操作，所以 _Watchdog_Tickle 函数需要分别处理（体现在对两条 Watch Dog 链：_Watchdog_Ticks_chain 和 _Watchdog_Seconds_chain 的处理上）。

在具体实现上，此函数首先判断链表上的第一个定时器是否超时。如果第一个定时器超时了，那就把它移除。如果它的状态是 ACTIVE 就执行它的超时函数，接着操作链表上的下一个 watchdog，直到没有到期的 watchdog 为止，这样 watchdog 的函数就可以得到运行了。需要注意的是，每调用 _Watchdog_Tickle 一次，就认为 watchdog 链上的 WatchDog 度过了一个单位时间。所以 _Watchdog_Tickle 做的第一件事情就是把链首的 WatchDog 的 delta_interval 减一，并在减到0时处理事件。到期执行例程可以做许多有意义的事情，比如线程睡眠到期就执行 _Thread_Delay_ended 函数，它会把该线程设为 ready 状态。

那么 _Watchdog_Tickle_ticks 函数是在哪被调用呢，既然它是每个时间片到来是运行的，那么应该在时钟中断里。事实上也是这样，在时钟中断处 理程序 clockIsr 中会调用 rtems_clock_tick 函数，rtems_clock_tick 就会调用 _Watchdog_Tickle_ticks，除此以外它还会调用 _TOD_Tickle_ticks 更新系统时间，调用 _Thread_Tickle_timeslice 函数更新线程的时间片，然后发生线程调度。在 _TOD_Tickle

函数中，会进一步调用_Watchdog_Tickle_seconds 函数更新秒粒度的 watchdog。有关_TOD_Tickle 函数的分析可参考本章的“核心层的 TOD Handler 组件”小节。

调整 watchdog

_Watchdog_Adjust 函数将_Timer_Ticks_chain 或_Watchdog_Seconds_chain（由输入参数 header 指明）在 direction 方向上调整了 units 的秒数。在实现调整过程中，需要屏蔽中断。

_Watchdog_Adjust 函数会被系统服务层的 timer server 调用。同时_Watchdog_Adjust_seconds 函数和_Watchdog_Adjust_ticks 函数进一步封装了_Watchdog_Adjust 函数，二者的区别体现在输入参数 header 上，一个是_Watchdog_Seconds_chain，一个是_Watchdog_Ticks_chain。_Watchdog_Adjust_seconds 函数的定义在 rtems/cpukit/score/inline/rtems/score /watchdog.inl 中。

函数调用信息：

```
void _Watchdog_Adjust(  
    Chain_Control          *header,  
    Watchdog_Adjust_directions direction,  
    Watchdog_Interval      units  
)
```

函数功能描述：

_Watchdog_Adjust 函数如果向后调整时间（backward），也就是说，时光倒流，所有的 WatchDog 都“变年轻”了，那么所有 WatchDog 事件的发生时间都应该推迟，这比较简单，按照上面的分析，只要把链首的 WatchDog 的 delta_interval 加上 units 就可以了。

如果向前调整时间（forward），也就是说，需要令这些 WatchDog 渡过了 units 个单位时间。在物理时间流逝的整个过程中，可能会有 WatchDog 事件发生，于是会有 WatchDog 事件需要处理并从链中删除相应的 WatchDog。在时间流逝过程中，首先检查链首的 WatchDog 的 delta_interval 是否小于等于 units，如果是，那么就说明应该发生 watchdog 事件了。且对于这条 watchdog 链来而言，度过了 delta_interval 个单位时间，所以要通过从 units 中减去 delta_interval 来模拟“时间的流逝”，然后应该处理这个 WatchDog 的事件（调用_Watchdog_Tickle）。处理完此事件后，此 watchdog 链的链首会被删去，并进一步处理新的链首。这个过程一直继续，直到 watchdog 链空，或者链首的 delta_interval 大于 units（注意在模拟过程中 units 会不断减小），就不用处理事件了，而是直接把链首的 delta_interval 减掉 units 就模拟了时间的流逝。

核心层的 TOD Handler 组件

RTEMS 通过 watchdog handler 组件实现的 WatchDog 机制，其关键是对两条 watchdog 链（_Watchdog_Ticks_chain 和_Watchdog_Seconds_chain）上的 watchdog 事件的处理。但是应该看到：它的“时间”只是抽象意义上的，当调用一次_Watchdog_Tickle_ticks 函数，则_Watchdog_Ticks_chain 这条链上的所有 WatchDog 就相当于度过了一个 tick 单位时间。但是它缺乏和实际上的时间的对应关系。比如说，我们完全可以调用_Watchdog_Tickle_ticks 函数成千上万次，却一次也不调用_Watchdog_Tickle_seconds 函数，那么，在_Watchdog_Ticks_chain 链上的 WatchDog 就“感觉”到了时间的流逝，而在_Watchdog_Seconds_chain 链上的 WatchDog 就认为没有经过任何时间，或者说，时间是静止的。

而实际上，为了模拟实际的时间概念，这两条链上的时间应该是同步的。如果调用 _TOD_Ticks_per_second（这是一个常数，代表一秒钟对应多少次 tick）次_Watchdog_Tickle_ticks()，

就应该调用一次_Watchdog_Tickle_seconds()。

另外, WatchDog 没有直接提供对于周期性事件的处理机制。这个也需要系统服务层的相关组件和应用程序去实现。真正地处理周期性计时事物, 把两条 WatchDog 链联系起来, 并在此基础上提供关于实际“时间”的操作的机制, 是核心层的 TOD handler 组件。

TOD handler 组件提供了对时间的基本支持, 主要完成了对系统启动后时间的更新维护, 并对高层的时间管理等提供支持。TOD 相关的常量、数据结构定义和函数声明在 rtems/cpukit/score/include/rtems/score/tod.h 中。

关键数据结构

ScoreTODConstants 常量

常量定义中, 首先是名为 Score 中 TOD 的一组常量, 如下:

```
#define TOD_SECONDS_PER_MINUTE (uint32_t)60
#define TOD_MINUTES_PER_HOUR   (uint32_t)60
#define TOD_MONTHS_PER_YEAR    (uint32_t)12
#define TOD_DAYS_PER_YEAR      (uint32_t)365
#define TOD_HOURS_PER_DAY      (uint32_t)24
#define TOD_SECONDS_PER_DAY    (uint32_t) (TOD_SECONDS_PER_MINUTE * \
                                           TOD_MINUTES_PER_HOUR * \
                                           TOD_HOURS_PER_DAY)
#define TOD_SECONDS_PER_NON_LEAP_YEAR (365 * TOD_SECONDS_PER_DAY)
#define TOD_MILLISECONDS_PER_SECOND (uint32_t)1000
#define TOD_MICROSECONDS_PER_SECOND (uint32_t)1000000
#define TOD_NANOSECONDS_PER_SECOND (uint32_t)1000000000
#define TOD_NANOSECONDS_PER_MICROSECOND (uint32_t)1000
```

这组常量是与日期时间关系紧密的一组常量, 主要是各个单位之间的转化倍数, 在函数中经常使用, 所以在这里定义, 以便明确其意义。

TOD_BASE_YEAR

TOD_BASE_YEAR 是 rtesm 中可定义的最早的年份, 因为 rtems 中是以这一年作为起点, 从这一年开始计算从它到现在这一段时间的长度的。具体情况参考对数据结构和函数的分析。由 TOD_BASE_YEAR 的定义可以知道 rtems 是以1988年为计时的起点的, 如下:

```
#define TOD_BASE_YEAR 1988
```

时间常量数组

```
#ifdef SCORE_INIT
const uint32_t _TOD_Days_per_month[ 2 ][ 13 ] = {
    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
    { 0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};
const uint16_t _TOD_Days_to_date[2][13] = {
    { 0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334 },
    { 0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335 }
};
const uint16_t _TOD_Days_since_last_leap_year[4] = { 0, 366, 731, 1096 };
```

```
#else
extern const uint16_t  _TOD_Days_to_date[2][13]; /* Julian days */
extern const uint16_t  _TOD_Days_since_last_leap_year[4];
extern const uint32_t  _TOD_Days_per_month[2][13];
#endif
```

如果定义了 SCORE_INIT，那么将会定义一系列常量数组，有 _TOD_Days_per_month 实际、_TOD_Days_to_date、_TOD_Days_since_last_leap_year 等。_TOD_Days_per_month 是指各个月份中有多少天，由于有闰年的特殊情况，需要将其设置为2维数组，_TOD_Days_per_month [0] 是普通年份的数据，_TOD_Days_per_month [1] 是闰年年份的数据。_TOD_Days_to_date 是表示从本年度开始到各个月份开始之间的天数，同样由于有闰年的特殊情况，需要将其设置为2维数组。从 _TOD_Days_per_month 可以计算出 _TOD_Days_to_date，但是由于在函数中经常使用，所以设置后者，有利于加快速度。

_TOD_Days_since_last_leap_year 表示从最近的一次闰年开始到本年度开始之间的天数：_TOD_Days_since_last_leap_year [0] 表示本年度为闰年，所以值为 0；_TOD_Days_since_last_leap_year [1] 表示去年为闰年，所以值为366，依次类推。

如果没有定义 SCORE_INIT，那么将需要使用另外的一组常量，这组常量不是在这里定义，应该是由用户定义提供的。这组常量的名称和意义与定义了 SCORE_INIT 使用的常量相同。

TOD_Control

```
typedef struct {
    uint32_t  year;
    uint32_t  month;
    uint32_t  day;
    uint32_t  hour;
    uint32_t  minute;
    uint32_t  second;
    uint32_t  ticks;
} TOD_Control;
```

TOD_Control 是 TOD 的控制块，是 TOD 中的核心数据结构。其中有7个数据成员，分别是 year、month、day、hour、minute、second、ticks。前六个意义都很明确。ticks 可以理解为比 second 小的时间单位。在这里没有用 microsecond 是出于对程序运行的时间精度进行控制的考虑，将 ticks 作为程序运行的最小时间单位，并且 ticks 单位的大小可以通过改变 _TOD_Microseconds_per_tick 而改变，从而对程序的时间精度进行控制。

_TOD_Microseconds_per_tick 是 SCORE_EXTERN 的数据，分析见下面。对 ticks 的设置见 _TOD_Handler_initialization 函数的分析。

TOD handler 相关的全局变量

```
SCORE_EXTERN boolean _TOD_Is_set;
SCORE_EXTERN TOD_Control _TOD_Current;
SCORE_EXTERN Watchdog_Interval _TOD_Seconds_since_epoch;
SCORE_EXTERN uint32_t  _TOD_Microseconds_per_tick;
SCORE_EXTERN uint32_t  _TOD_Ticks_per_second;
SCORE_EXTERN Watchdog_Control _TOD_Seconds_watchdog;
```

_TOD_Is_set 是表明 TOD 是否已经设置的布尔值，在 _TOD_Set 这设置。_TOD_Current 为当前时间的控制块，函数即是通过改变它来实现对当前时间的控制的。_TOD_Seconds_since_epoch 表明从计时起点 TOD_BASE_YEAR((1988.1.1 0:0:0.00)) 开始到现在的秒数，是 Watchdog_Interval 类型。Watchdog_Interval 的定义在 rtems-4.7.99.2/cpukit/score/include/rtems/score/watchdog.h 中，如下：

```
typedef uint32_t Watchdog_Interval;
```

可见实际上就是 uint32_t 类型，但是为了在特定的使用环境中意义明确，所以定义为 Watchdog_Interval。

_TOD_Microseconds_per_tick 定义了一个 tick 节拍相当于多少微秒，表明了 ticks 单位的大小。
_TOD_Ticks_per_second 表明了一秒相当于多少 ticks。

_TOD_Seconds_watchdog 是 Watchdog_Control 结构体的全局变量。Watchdog_Control 结构体是 Watchdog handler 的控制块。Watchdog 是 rtems 中为了实现时间管理定时而引入的概念 有关 Watchdog_Control 结构体的进一步的分析见上节的内容。_TOD_Seconds_watchdog 负责周期性计时，并驱动_Watchdog_Seconds_chain 链的时间流动和超时处理。如果保证 _TOD_Seconds_watchdog 事件每秒发生一次，就可以完成计时工作，以及处理好_Watchdog_Seconds_chain 链的时间流动。
_TOD_Seconds_watchdog 执行的到期执行例程是 _TOD_Tickle 函数。

关键实现函数

TOD handler 初始化

函数调用信息：

```
void _TOD_Handler_initialization(  
    uint32_t    microseconds_per_tick )
```

函数功能描述：

函数对 TOD 的相关数据进行了初始化。传进的参数 microseconds_per_tick 给出了 tick 微秒有多少，并用于初始化 _TOD_Microseconds_per_tick。接下来将 _TOD_Current 初始化为 TOD_BASE_YEAR 的开始时刻，_TOD_Seconds_since_epoch 相应地设置为 0。如果 microseconds_per_tick 为 0，则 _TOD_Ticks_per_second 也设置为 0；否则计算得到 _TOD_Ticks_per_second 的整数值。新版本此模块已分开

_TOD_Is_set 需要设置为 FALSE，在 _TOD_Set 中才设置为 TRUE。最后调用 _TOD_Activate。

设置 TOD

函数调用信息：

```
void _TOD_Set(  
    TOD_Control *the_tod,  
  
)
```

函数功能描述：

此函数的作用为根据传进的参数 seconds_since_epoch 对 the_tod 指向的 TOD 控制块进行设置。重新设置并不是简单地把 _TOD_改成要设置的值，而是对所有的 WatchDog 事件的发生时间进行相应地调整。而且

在把时间向前调整的情况下，还有可能导致一批 WatchDog 事件发生，这也是需要处理的，好在 WatchDog 机制里面已经提供调整时间的实现方法。所以这里要做的就是判断设置的时间和当前时间的前后关系，并正确调用_Watchdog_Adjust_seconds。

在此函数的头尾，分别调用了_Thread_Disable_dispatch() 和_Thread_Enable_dispatch()，将中间操作保护起来。函数首先调用_TOD_Deactivate()。这个函数与_TOD_Activate() 对应。此函数然后根据 seconds_since_epoch 和_TOD_Seconds_since_epoch 的大小关系决定是将当前的时间相前调整还是向后调整。时间的调整调用了_Watchdog_Adjust_seconds 函数，实际上是调用了_Watchdog_Adjust 函数，将_Watchdog_Seconds_chain 在 direction 方向上调整了 units 的秒数，从而_TOD_SET 能将当前时间设置为所指定的时间。之后将_TOD_Current 设为 the_tod 指向的 TOD 控制块，并将_TOD_Seconds_since_epoch 设为 seconds_since_epoch 并将_TOD_Is_set 设为 TRUE。

函数最后调用了_TOD_Activate。传进的参数 ticks_until_next_second 是 _TOD_Ticks_per_second 减去 the_tod->ticks (**ticks_until_next_second<=the_tod->ticks 的情况应该不会出现，否则 TOD 将是不正确的，这里进行判断应该是为了避免在减去之后 ticks_until_next_second<=0 从而引起 _TOD_Activate() 调用不正确？**)，也就是距离下一次_TOD_Seconds_watchdog 调用_TOD_Tickle 进行对 TOD 进行更新的 ticks 数，这样就保证了 TOD 的持续更新。

检查 TOD 的合法性

函数调用信息：

```
boolean _TOD_Validate( TOD_Control *the_tod )
```

函数功能描述：

此函数检查 the_tod 中表示的时间是否有错的。主要是检查 the_tod 是否为空和各个数据成员是否在其范围之内。这个函数的设置主要是 TOD 控制块中的数据成员的范围随着年份的不同会有浮动，所以不能简单的通过定义各个数据成员相应的枚举类型来达到保证数据成员的正确性。

转换 TOD 为秒表示

函数调用信息：

```
uint32_t _TOD_To_seconds( TOD_Control *the_tod )
```

功能描述：

此函数将 the_tod 表示的时间转换为从计时起点开始到其的秒数。首先根据 the_tod->day 和本年度是否是闰年计算了从本年度开始到现在的天数，存到 time 中。然后根据_TOD_Days_since_last_leap_year 和从计时起点到现在过去了多少个4年循环（带一个闰年），令 time 加上从计时起点到本年度开始的天数，这样就得到了从计时开始到现在的天数。time 乘以 TOD_SECONDS_PER_DAY，再加上 hour 和 minute、second 表示的秒数，即得结果。

到期的 TOD 执行例程

函数调用信息：

```
void _TOD_Tickle(
```

```

    Objects_Id id,
    void      *ignored
)

```

函数功能描述:

此函数在每次_TOD_Seconds_watchdog 的 watchdog 到期时（一般是1秒一次，**可通过执行_TOD_SET 函数对到期时间进行修改**）被调用，这是在初始化_TOD_Seconds_watchdog（参见前面对_TOD_Handler_initialization 函数的分析）时设置的。此函数完成的功能是将 TOD 增加1秒，并将_TOD_Seconds_watchdog 重新放回 Watchdog chain 中，以便下次（1秒后）再进行更新。将 second 增加1是递增式考虑的，也就是如果秒数达到60，就需要重置为0，然后将 minute 增加1，如果分数达到60，就重置为0，将 hour 增加1，依次类推。在 TOD 增加1秒后，调用了_Watchdog_Tickle_seconds 函数，这个函数位于

rtems/cpukit/score/inline/rtems/score/watchdog.inl 中，定义如下:

```

RTEMS_INLINE_ROUTINE void _Watchdog_Tickle_seconds( void )
{ _Watchdog_Tickle( &_amp;Watchdog_Seconds_chain ); }

```

_Watchdog_Tickle_seconds() 驱动 _Watchdog_Seconds_chain 链的时间流动。这样，_Watchdog_Seconds_chain 链上的 WatchDog 才会发生事件。最后调用函数_Watchdog_Insert_ticks 将_TOD_Seconds_watchdog 的下一个时间间隔设置为_TOD_Ticks_per_second 个 ticks（也就是1秒），并插入链表中。

系统服务层的时钟（Clock）管理器组件

TOD 机制建立在_Watchdog_Ticks_chain 链正常工作的基础上，由 _TOD_Seconds_watchdog 来实现周期性计时，并驱动_Watchdog_Seconds_chain 链。但是，_Watchdog_Ticks_chain 链本身需要驱动者，必须在每个 tick 调用_Watchdog_Tickle_ticks 函数，才能保证_Watchdog_Ticks_chain 链的正常运作，而这个工作是通过系统服务层的时钟（Clock）管理器组件提供的 Clock 机制来完成的。

时钟管理器组件负责驱动整个系统的 tick 时钟（驱动_Watchdog_Ticks_chain 链的运作），负责提供线程的时间片机制，并提供了设置和获取当前时刻的函数。时钟管理器组件同时提供对日期和其他的与时间相关的服务。其中发布时间信号的函数 rtems_clock_tick 是关键，当时钟中断产生时候，时钟中断服务例程会调用它，而它会进一步调用核心层的 TOD 组件、看门狗组件和线程组件中与时间相关的函数，为与时间相关的调度、同步互斥等提供时间更新支持。

为了向钟管理器提供时钟，必须有一个周期型的时钟中断。因此系统中必须要有一个硬件时钟去产生这个中断。时钟 ISR 会调用 rtems_clock_tick 函数通知系统已经经过了一个特定的时间段。在 RTEMS 中，时间的最小单位是 tick，它的实际时间值在系统配置表中配置。

关键数据结构

时刻 rtems_clock_time_value 的定义如下:

```

typedef struct {
    uint32_t    seconds;
    uint32_t    microseconds;
} rtems_clock_time_value;

```

这是所谓“Unix 风格”的时刻，也就是给出从基准时刻开始到当前时刻的秒数以及微秒数。

关键实现函数

时钟管理器组件提供的函数主要是：

- ⑩ rtems_clock_set：设定系统日期和时间
- ⑩ rtems_clock_get：获取系统时间信息
- ⑩ rtems_clock_tick：发布时间信号

下面分别进行介绍。

获取系统时间信息

函数调用信息：

```
rtems_status_code rtems_clock_get(  
    rtems_clock_get_options option,  
    void *time_buffer  
)
```

函数功能描述：

此函数获得系统日期和时间 或其他与时间相关的信息，其中 option 与 time_buffer 的关系如下：

| option 取值 | time_buffer 的类型 | ime_buffer 返回值的含义 |
|-------------------------------------|-------------------------|-------------------|
| RTEMS_CLOCK_GET_TOD | rtems_time_of_day* | 系统日期和时间（绝对时间） |
| RTEMS_CLOCK_GET_SECONDS_SINCE_EPOCH | rtems_interval* | 自1988年1月1日零时起的秒数 |
| TEMS_CLOCK_GET_TICKS_SINCE_BOOT | rtems_interval* | 自RTEMS启动以来的tick数 |
| RTEMS_CLOCK_GET_TICKS_PER_SECOND | rtems_interval* | 每秒tick数 |
| RTEMS_CLOCK_GET_TIME_VALUE | rtems_clock_time_value* | 系统日期和时间（相对时间） |

设定系统日期和时间

函数调用信息：

```
rtems_status_code rtems_clock_set(  
    rtems_time_of_day *time_buffer  
)
```

函数功能描述：

此函数用来设置系统日期和时间，这是通过调用 TOD handler 组件提供的_TOD_Set 函数来具体完成的。

tick 到期服务例程

函数调用信息：

```
rtems_status_code rtems_clock_tick( void )
```

函数功能描述：

此函数被 BSP 的时钟中断服务例程（在 PC 386中，是 clockIsr 函数）调用，用于在每个 tick 到来时进行系统状态更新。其具体工作包括：调用_TOD_Tickle_ticks函数修改系统时间：_TOD_Current.ticks 和

_Watchdog_Ticks_since_boot 增1；调用_Watchdog_Tickle_ticks 函数来驱动_Watchdog_Ticks_chain 链，完成对 watchdog 的更新和到期事件处理；调用_Thread_Tickle_timeslice 函数，将当前运行任务时间片加1；如果使用时间片调度算法，还将检查剩余时间片以判断是否需要线程切换；如果任务需要切换标志已设置，调用_Thread_Dispatch 函数进行线程切换。

应用举例

RTEMS源码中的\${RTEMS_ROOT}/testsuites/sptests/sp01中包含了时钟管理的测试用例。它主要包括一个初始化任务和其他三个任务。

初始化任务：

1. 调用build_time生成一个确定的时间time
2. 调用rtems_clock_set把1中的时间设置为系统时间
3. 创建并启动三个优先级相同的任务，它们运行的任务函数都是Task_1_through_3
4. 删除自己，使得其他任务可以运行

Task_1_through_3是一个循环体，主要包括：

1. 调用rtems_clock_get获取系统时间
2. 如果它的秒数大于或等于35则退出测试
3. 否则打印时间，并调用rtems_task_wake_after等待5秒

运行结果如下：

```
*** TEST 1 ***
TA1 - rtems_clock_get - 09:00:00 12/31/1988
TA2 - rtems_clock_get - 09:00:00 12/31/1988
TA3 - rtems_clock_get - 09:00:00 12/31/1988
TA1 - rtems_clock_get - 09:00:05 12/31/1988
TA2 - rtems_clock_get - 09:00:10 12/31/1988
TA1 - rtems_clock_get - 09:00:10 12/31/1988
TA3 - rtems_clock_get - 09:00:15 12/31/1988
TA1 - rtems_clock_get - 09:00:15 12/31/1988
TA2 - rtems_clock_get - 09:00:20 12/31/1988
TA1 - rtems_clock_get - 09:00:20 12/31/1988
TA1 - rtems_clock_get - 09:00:25 12/31/1988
TA3 - rtems_clock_get - 09:00:30 12/31/1988
TA2 - rtems_clock_get - 09:00:30 12/31/1988
TA1 - rtems_clock_get - 09:00:30 12/31/1988
*** END OF TEST 1 ***
```

系统服务层的定时器(Timer) 管理器组件

系统服务层的定时器管理器提供对定时器管理服务。在具体实现上，定时器管理器可以在中断层（通过函数 rtems_clock_tick）和任务层（通过定时器服务器任务）实现定时更新和相关处理操作。系统服务层的定时器管理器提供了 Timer 实现机制和 Timer Server 实现机制来完成具体的定时操作。

Timer 机制可以简单的理解为定时器或者说是一个闹钟，它也是是 RTEMS 里的一个标准的 object。它可帮助任务定制在未来的某一时刻执行某一操作。Timer Server 机制专门实现了一个基于任务级的定时器（task-based Timer），这实际上是通过一个专门的 Timer Server 任务来完成具体的工作的。此任务是一个优先级比任何其他 RTEMS 任务的优先级都高的任务，且不可被抢占（non-preemptible）。正因为它是不可被抢占的，所以也可把也可以把 Timer Server 任务看作是一个优先级最低的”中断”。它的作用是管理与所有基于任务的与 Timer 有关的计时服务例程。

Timer 机制提供的定时功能有着诸多用途，其中比较重要的一个用途是在 Timer 的管理中制定若干的重置点（reset point），每次达到重置点，Timer 的触发（fire）被重置，也就不会触发了。但一旦到了预定的时间，Timer 仍然没有达到重置点，那么 Timer 就会触发。这可以用做为一种超时处理的实现机制，在 RTEMS 中有超时处理需求的组件中，此用途必不可少。

RTEMS classic Timer 从是否是 task-based 和是从现在开始一段时间后触发与在每天的固定时间触发这两个角度被分为了四种类型，其关系如下表所示：

| 类型 | 初始化函数 | 能否 reset | 需要 Timer Server |
|---------------------------|-------------------------------|----------|-----------------|
| TIMER_INTERVAL | rtems_timer_fire_after | 能 | 否 |
| TIMER_TIME_OF_DAY | rtems_timer_fire_when | 否 | 否 |
| TIMER_INTERVAL_ON_TASK | rtems_timer_server_fire_after | 能 | 是 |
| TIMER_TIME_OF_DAY_ON_TASK | rtems_timer_server_fire_when | 否 | 是 |

它们完成的功能都是在达到时间上的某一条件之后触发，即调用预定的某一函数。为了实现这一功能，每一个 Timer 需要一个 object 用于管理自身的创建，析构，以及资源分配，一个 watchdog 来记录自己的时序关系，一个表明自己当前属性的标记。从对 Timer 的分析中我们还可以看出，RTEMS 的一套计时功能均依赖于 watchdog 和 TOD 系统，watchdog 提供了一个链表的数据结构用于维护各需要时间概念的 object 之间的时序关系，TOD 提供了年月日，时分秒，而 Timer 就是建立在这个时序关系和标尺之上的报时机制。

关键数据结构

五类型的 Timer 状态

定义在 cpukit\rtems\src\Timer.h 中

```
typedef enum {
    TIMER_INTERVAL,
    TIMER_INTERVAL_ON_TASK,
    TIMER_TIME_OF_DAY,
    TIMER_TIME_OF_DAY_ON_TASK,
    TIMER_DORMANT
} Timer_Classes;
```

上述代码定义了一个 Timer 实例的五种状态类型。

Timer 控制结构体

```
typedef struct {
    Objects_Control  Object;
    Watchdog_Control Ticker;
    Timer_Classes   the_class;
} Timer_Control;
```

object 域使得 Timer 可以像标准的 rtems object 一样被管理也就是它作为一个 object 的 object 控制块，Ticker 为 Timer 添加 watch dog 控制块，class 为该 timer 所属的 timer 类型。

Timer 相关全局变量

```
RTEMS_EXTERN Objects_Information _Timer_Information;
_Timer_Information 用于管理这一类 Timer 对象。
```

```
RTEMS_EXTERN Thread_Control *_Timer_Server;
_Timer_Server 是指向时间服务器线程的 TCB 的指针。
```

```
extern Chain_Control_Timer_Ticks_chain;
extern Chain_Control_Timer_Seconds_chain;
```

定义了一个 Timer Server 上下文中所有 Timer 构成的链表，根据 Timer 类型不同分为两条链。

```
Watchdog_Interval_Timer_Server_seconds_last_time;
Watchdog_Interval_Timer_Server_ticks_last_time;
```

上一次 Timer Server 在链上处理的 Timer，同样分为两类。

```
Watchdog_Control_Timer_Seconds_timer;
```

用于控制 Timer Server 的 timer。

关键实现函数

定时器管理器初始化

函数调用信息：

```
void _Timer_Manager_initialization(
    uint32_t maximum_timers
)
```

函数功能描述：

此函数是定时器管理器的初始化函数，它完成了所有与定时器管理器的相关数据结构的初始化。它的功能主要通过调用 object 的初始化函数：

```
_Objects_Initialize_information(
    &_Timer_Information,    /* Timer object 信息表*/
    OBJECTS_CLASSIC_API,   /* object API 类 */
    OBJECTS_RTEMS_TIMERS,  /* object 属性类*/
    maximum_timers,        /* Timer object 最大个数 */
    sizeof( Timer_Control ), /* Timer_Control 的大小 */
    FALSE,                 /* 名字是否为 string*/
    RTEMS_MAXIMUM_NAME_LENGTH /* object 名字的最大长度*/
#ifdef(RTEMS_MULTIPROCESSING)
    ,
    FALSE,                 /* 是否是 global 类型*/
    NULL                   /* Proxy extraction support callout */
#endif
);
```

来初始化一个 Timer object 来实现。并为一个指向 Timer Server 的指针 _Timer_Server 为空。如果此指针为 NULL，则说明基于 timer server 任务还未被初始化。

创建定时器

函数调用信息：

```
rtems_status_code rtems_timer_create(
    rtems_name name,
    Objects_Id *id
)
```

函数功能描述：

通过配置 Timer 的计时器控制块（Timer control block TMCB）来创建一个 Timer。创建伊始，并没有与此 Timer 绑定的时间服务程序（Timer service routine），处于非活动的休眠态。主要的执行流程如下：

1. 首先判断 timer 的 name 和 id 的合法性
2. 屏蔽线程调度，确保后面操作的原子性。
3. 调用 _Timer_Allocate 函数为 timer object 分配资源。_Timer_Allocate 函数实际调用的是 _object_Allocate(& Timer_Information)
4. 判断如果没能分配成功，则使能线程调度，返回出错信息 TOO_MANY，表示没有可用的 timer object 了。
5. 否则即成功分配 timer object 资源，然后进行为 Timer 配置属性的操作：配置 Timer 类型为 TIMER_DORMANT（刚刚被创建的 Timer 为休眠态），初始化此 timer 上的 watchdog: the_timer->Ticke。
6. 调用 object 组件提供的 _Objects_Open 函数, 将 Timer 的 Object 指针和 Timer 的名字分别注册到 _Timer_Information 中的 local_table 项，并对 object_control 结构体的 name 域赋值，并给 id 赋值。这样系统才能正常使用这个 timer object。
7. Timer object 创建完毕。到此可允许线程调度。

根据 timer 的 name 查找其对象 id

函数调用信息：

```
rtems_status_code rtems_timer_ident(
    rtems_name  name,
    Objects_Id  *id
)
```

函数功能描述：

此函数主要通过调用函数：

```
_Objects_Name_to_id(
    &Timer_Information,
    (Objects_Name) name,
    OBJECTS_SEARCH_LOCAL_NODE,
    id
);
```

来根据 timer 的 name 在 _Timer_Information 表中查找 timer 的 object id。 _Timer_Information 表中的 object id 信息就是在每个 Timer 通过调用 rtems_timer_create 被创建的时候，通过 _Objects_Open 函数注册进去的。

取消 timer

函数调用信息：

```
rtems_status_code rtems_timer_cancel(
    Objects_Id id
)
```

函数功能描述：

根据 object id 取消某个特定的 Timer，一旦一个 Timer 被 cancel 了, 在重新初始化（重新初始化的操作可以通过调用函数 rtems_timer_reset 完成）之前将不起作用。主要执行流程如下：

1. 首先调用 _Timer_Get 函数，通过 Timer 的 object id 得到其的 object location;
2. 判断 location 的类型，如果是 REMOTE 类型或者 ERROR 一律为非法状态；

3. 只有在 location 为 LOCAL 时，并且 Timer 并没有处于休眠状态，则将 Timer 的 Watchdog 从 watchdog 的链上移除，并将其的状态置为非活动态 (inactive)，这样使 Timer 停止触发。

删除 timer

函数调用信息：

```
rtems_status_code rtems_timer_delete( Objects_Id id )
```

函数功能描述：

用于删除一个 Timer。首先调用 _Objects_Close 函数把保存在 _Timer_Information 中的 local_table (即 Local Object Pointer Table) 对应项的 object_control 指针清空，且把 object_control 结构体的 name 域清空；将 Timer 的 Watchdog 从 watchdog 的链上移除；调用 _Timer_Free 函数把 timer 对象变成空闲状态，可这样可以被再次分配。一旦一个 Timer 被删除后，所有对于它的 name 和 id 的操作将都是非法的。

设定触发定时器相对时间

函数调用信息：

```
rtems_status_code rtems_timer_fire_after(
    Objects_Id          id,
    rtems_interval      ticks,
    rtems_timer_service_routine_entry routine,
    void                *user_data
)
```

函数功能描述：

判断某特定 id 的 Timer 的合法性。如果合法，且如果此 Timer 正在运行，那么先将它的 watchdog 从 watchdog 链中取消掉，再重新初始化它的 watchdog，为其设置在某一时间间隔之后触发 (fire)。所谓触发就是在将来某时间到期后，RTEMS 会调用 rtems_timer_fire_after 函数设置的定时器服务例程 (timer service routine)。此函数的执行流程如下：

1. 进行一系列的合法性判断，如判断间隔时间的合法性，提供的计时器服务程序入口的合法性。
2. 将判断为 local 的 Timer 中的 Watchdog 从 watchdog 的链表中移除，并将其状态置为非活动状态。
3. 屏蔽中断，保证初始化操作的原子性。
4. 判断 watchdog 的状态是否为非活动态。这是字面的意思，而且看似重复，因为刚刚的操作刚将其状态设置为非活动的。其实不然，这步确保了在屏蔽中断之前，watchdog 是否被一个更高优先级的中断打断并修改，如果被打断则放弃后面的初始化操作使能中断，继续线程调度。否则，继续初始化操作。
5. 配置 TMCB 中的信息：将 Timer 的状态类型置为 TIMER_INTERVAL，初始化 watchdog 配置完毕使能中断。
6. 最后调用 _Watchdog_Insert_ticks 函数，将此 Timer 的 watchdog 插入到 watchdog chain 统一管理所有活动状态的 watchdog 的链表中。工作结束，允许线程调度。

设定触发定时器绝对时间

函数调用信息：

```

rtems_status_code rtems_timer_fire_when(
    Objects_Id          id,
    rtems_time_of_day   *wall_time,
    rtems_timer_service_routine_entry routine,
    void                *user_data
)

```

函数功能描述:

此函数与唯一与 `rtems_timer_fire_after` 函数的不同之处有三个:

- ❏ 合法性检查多了三条: 一条是检查系统是否设定了当前的时间; 另一条是检查当前的时间日期格式是否合法; 最后一条是检验计时是否准确, 通过计算从计时开始到现在时间的秒数与 `super core` 中存的数据进行比较。
- ❏ 由相对的时间间隔变成了绝对的墙上时间 (格式为年月日时分秒, 且此函数处理的最大精度为秒)。
- ❏ 将 Timer 类型状态置成 `TIMER_TIME_OF_DAY`
- ❏ 在最后调用 `_Watchdog_Insert_seconds`, 将此 Timer 的 watchdog 插入到 watchdog 链表中。

问题:

值得一提的一个比较奇怪的现象是为什么 `fire_after` 需要屏蔽中断/使能中断来保证原子性而 `fire_when` 却不需要呢?

重置定时器

函数调用信息:

```

rtems_status_code rtems_timer_reset( Objects_Id id )

```

函数功能描述:

用于一个被 `cancel` 的 Timer 的重启, 这个 Timer 被 `rtems_timer_fire_after` 或 `rtems_timer_server_fire_after` 初始化的。Reset 之后会用和以前相同的 `interval` 和 `routine` 继续工作。。 主要执行流程如下:

1. 调用 `_Timer_Get` 函数通过 `id` 得到 `location`, 如果不是 `local` 类型均报错, 屏蔽线程调度;
2. 然后根据 Timer 类型不同分别处理: 如果是 `TIMER_INTERVAL` 类型 (即通过 `rtems_timer_fire_after` 初始化的) 则通过连续执行 `_Watchdog_Remove` 和 `_Watchdog_Insert` 先将现有的 Timer 停掉, 再按照以往的配置, 重新注册 Timer。
3. 如果是 `TIMER_INTERVAL_ON_TASK` (即 `rtems_timer_server_fire_after` 初始化) 则在单纯的 Watchdog remove 和 insert 的操作基础上, 还须加入对 Timer server 的操作。在 remove 之前需要停掉时间服务器 (内部的操作是调用 `_Watchdog_Remove(&_amp;Timer_Server->Timer)` 将 Timer server 对应的 watchdog 置为 inactive)。在 remove 之后要调用 `_Timer_Server_process_ticks_chain()`; 调节当前的 interval timer 链表。在插入之后要重启 Timer Server。
4. 其他 Timer 类型不予理会 (`TIMER_TIME_OF_DAY`, `TIMER_TIME_OF_DAY_ON_TASK`, `TIMER_DORMANT`)
5. 主要操作完毕, 允许线程调度。

启动定时器服务后台任务

函数调用信息:

```

rtems_status_code rtems_timer_initiate_server(
unsigned32 stack_size,
rtems_attribute attribute_set
);

```

函数功能描述:

此函数创建并启动为基于 Timer Server 任务。如果 Timer Server 没有被初始化，那么任何基于线程的 Timer 都无法初始化。主要执行流程如下:

1. 用线程创建函数来 rtems_task_create 创建 Timer Server 任务。其任务名被叫做“TIME”，优先级被设置为0，栈大小为用户定义的大小，因为它的线程优先级为0（最高）且属性设置为 NO_PREEMPT，所以可把它看作是一个优先级最低的“中断”。
2. 如果初始化不成功，则恢复线程调度，返回具体出错状态。
3. 接下来执调用函数 rtems_task_start 来启动 timer server 任务。再判断是否成功，如果不成功，则恢复线程调度，返回具体的出错状态。
4. 指定指向“TIME”的 TCB 指针，这一指针的确定标志着 Timer Server 的建立，自此便可以开始基于 Timer Server 任务的初始化工作了。
5. 初始化 Timer Server 任务管理的 task-based Timer 组成的链表（分为两类 Timer_Ticks_chain 和 Timer_Seconds_chain）
6. 初始化 Timer Server 任务管理的两个 watchdog（_Timer_Server->Timer 和 _Timer_Seconds_timer）。
7. 主要操作完毕，允许线程调度。。

在间隔后触发定时器任务

函数调用信息:

```

rtems_status_code rtems_timer_server_fire_after(
    Objects_Id          id,
    rtems_interval      ticks,
    rtems_timer_service_routine_entry routine,
    void                *user_data
)

```

函数功能描述:

此函数就是 timer_fire_after 函数在 timer server 任务级的实现。与 timer_fire_after 函数的区别是：在到期后，要等到 timer server 任务运行，并执行具体的定时器服务例程。这里需要考虑 Timer Server 任务的管理的具体执行还是有些不同。主要执行流程如下:

1. 首先还是进行合法性的判断，与普通 timer 相比多了一条 Timer Server 的判断：判断 Timer Server 的状态是否正确。
2. 然后通过 id 得到 location，除 LOCAL 类型外一律出错处理。
3. 之后同样是 remove watchdog 然后和正常 Timer 一样的判断是否有更高优先级中断的打断。
4. 配置 TMCB 中的信息：这一次是将 Timer 的状态类型置为 TIMER_INTERVAL_ON_TASK，初始化 watchdog。配置完毕使能中断。
5. 下一步应该就执行 watchdog insert 功能了，但是因为这是 task-based Timer 还需要对 Timer Server 进行必要操作以便于后来它能够对此 Timer 进行管理。附加的操作为：先停掉 Timer

- Server, 调整 Timer Server 管理的 Timer 链表, 将被初始化 Timer 的 watchdog 插入链, Timer Server 重新启动运行。(即将控制 Timer Server 的 Timer reset)。
6. 初始化完毕, 继续线程调度。

在确定时间后触发定时器任务

函数调用信息:

```
rtems_status_code rtems_timer_server_fire_when(  
    Objects_Id          id,  
    rtems_time_of_day   *wall_time,  
    rtems_timer_service_routine_entry routine,  
    void                *user_data  
)
```

函数功能描述:

同样的与普通的 Timer 的 fire when 操作类似, 只不过加入了初始化时对 Timer Server 的操作。主要执行流程如下:

1. 执行过程: 同样是先进行合法性判断, 比 rtems_timer_fire_when 多判断了 Timer Server 的状态。
2. 后续操作仍然和 rtems_timer_fire_when 一样, 不过是在配制 TMCB 时, Timer 状态类型置为 TIMER_TIME_OF_DAY_ON_TASK。
3. 在 TMCB 配制好之后, 同样加入了对 Timer Server 的调整操作, 不同的是, 这次是对 second 那条链进行操作。不过同样要执行步骤: 停下, 调整, 插入, 重启。
4. 配制结束, 继续线程调度。

应用举例

RTEMS源码中的\${RTEMS_ROOT}/testsuites/sptests/sp22中包含了定时器的测试用例。

它包括初始化任务和任务1两部分。

初始化任务:

1. 调用build_time来生成一个确定的时间time
2. 调用rtems_clock_set将time设置为系统时间
3. 创建并启动任务1
4. 调用rtems_timer_create创建一个timer
5. 删除自己, 使任务1开始运行

任务1:

1. 调用rtems_timer_ident获取初始化任务中创建的timer的ID
2. 打印当前时间, 并调用rtems_timer_fire_after设定timer在3秒后到时, 到时运行的函数为Delayed_resume (这个函数会调用rtems_task_resume使任务恢复)
3. 调用rtems_task_suspend将自己阻塞
4. 打印当前时间, 这个操作如果能够执行, 说明定时器已经到期, 在到期后定时处理器函数Delayed_resume, 使得任务可以恢复运行。
5. 调用rtems_timer_fire_after设定timer在3秒后到时, 到时运行的函数为Delayed_resume
6. 调用rtems_task_wake_after等待1秒
7. 1秒到时后打印时间
8. 调用rtems_timer_reset重启timer
9. 调用rtems_task_suspend将自己阻塞
10. 3秒后, 任务恢复运行, 打印当前时间
11. 调用build_time来生成一个确定的时间time, 并调用rtems_clock_set将time设置为系统时间

12. 调用rtems_timer_fire_after设定timer在3秒后到时，到时运行的函数为Delayed_resume
 13. 调用rtems_timer_cancel取消timer, 并打印当前时间
 14. 调用rtems_clock_get获取当前时间，并把这个时间加上3秒，得到新的time
 15. 调用rtems_timer_fire_when设置在time时到时，并且到时处理函数为Delayed_resume
 16. 调用rtems_task_suspend将自己阻塞
 17. 3秒后，任务恢复运行，打印当前时间
 18. 调用rtems_clock_get获取当前时间，并把这个时间加上3秒，得到新的time
 19. 调用rtems_timer_fire_when设置在time时到时，并且到时处理函数为Delayed_resume
 20. 调用rtems_task_wake_after等待1秒
 21. 1秒到时后打印时间
 22. 调用rtems_timer_cancel取消timer
 23. 调用rtems_task_wake_after(RTEMS_YIELD_PROCESSOR)将这个任务放在就绪链表的最后，但实际上这个链表上只有一个任务，所以它会继续运行
 24. 调用rtems_timer_delete删除定时器
- 运行结果如下：

```
*** TEST 22 ***
INIT - rtems_timer_create - creating timer 1
INIT - timer 1 has id (0x12010001)
TA1 - rtems_timer_ident - identifying timer 1
TA1 - timer 1 has id (0x12010001)
TA1 - rtems_clock_get - 09:00:00 12/31/1988
TA1 - rtems_timer_fire_after - timer 1 in 3 seconds
TA1 - rtems_task_suspend( RTEMS_SELF )
TA1 - rtems_clock_get - 09:00:03 12/31/1988
TA1 - rtems_timer_fire_after - timer 1 in 3 seconds
TA1 - rtems_task_wake_after - 1 second
TA1 - rtems_clock_get - 09:00:04 12/31/1988
TA1 - rtems_timer_reset - timer 1
TA1 - rtems_task_suspend( RTEMS_SELF )
TA1 - rtems_clock_get - 09:00:07 12/31/1988
<pause>
TA1 - rtems_timer_fire_after - timer 1 in 3 seconds
TA1 - rtems_timer_cancel - timer 1
TA1 - rtems_clock_get - 09:00:07 12/31/1988
TA1 - rtems_timer_fire_when - timer 1 in 3 seconds
TA1 - rtems_task_suspend( RTEMS_SELF )
TA1 - rtems_clock_get - 09:00:10 12/31/1988
TA1 - rtems_timer_fire_when - timer 1 in 3 seconds
TA1 - rtems_task_wake_after - 1 second
TA1 - rtems_clock_get - 09:00:11 12/31/1988
TA1 - rtems_timer_cancel - timer 1
TA1 - rtems_task_wake_after - YIELD (only task at priority)
TA1 - timer_deleting - timer 1
*** END OF TEST 22 ***
```

小口

第八章 同步互斥与通信机制

概述

当两个或多个线程在执行一些关键性的临界区代码时（如对共享资源的访问），如何确保它们不会相互妨碍？当线程之间存在着某种依存关系时，如何来调整它们的运行次序？当线程经常需要与其它线程进

行通信，那么如何根据需要提供有效的通信手段？这实际上需要操作系统提供同步互斥与通信的手段才能解决上述问题。

在 RTEMS 中，内核提供了信号量（Semaphore）、栅栏（Barrier）自旋锁、（SpinLock）、读写锁（RWLock）等同步互斥机制以及消息队列（Message Queue）、事件（Event）等通信机制。这些机制的实现主要在系统服务层，并建立在核心层的基础之上，所以本章每部分机制实现的分析包括核心层和系统服务层两部分。先回顾一下

信号量

信号量用于任务和任务之间或者是任务和中断服务例程之间的同步和互斥。RTEMS 支持二值信号量和计数信号量。二值信号量只能取0和1 两个值。当二值信号量的值为0时，表示信号量不可使用；当二值信号量的值为1时，表示信号量可以使用。而计数信号量的取值可以是任何非负的整数。当计数信号量的值为0时，表示信号量不可使用；当计数信号量的值大于0时，表示信号量可以使用。二值信号量用于控制单个资源的互斥访问，而计数信号量控制资源集的访问。例如我们有三台打印机，这样就不能简单的把信号量设置为0 或1，而应该采用计数信号量，并且把值设为3。

另外，我们也可以通过信号量来实现任务之间的同步。例如，首先设信号量的初值为0，任务 A 到达了同步点后，需要等待任务 B, 于是就在同步点调用信号量获取操作，A 将发现信号量为0，则继续等待。任务 B 到达同步点后，调用释放信号量操作，将信号量值加1，这时系统会使任务 A 就绪，并会获得继续执行的机会，这样就实现了任务间的同步。

RTEMS 的核心层提供了 semaphore handler 组件和 mutex handler 组件，实现了核心信号量（core semaphore）和核心互斥体（core mutex）两种核心机制。RTEMS 的系统服务层提供了 semaphore manager 组件，实现了计数信号量，二值信号量和简单二值信号量，其中计数信号量是对核心信号量的封装，二值信号量主要用于互斥同步操作，严格限制取值为0或1，而简单二值信号量除了严格限制取值为0或1 外，主要用于支持嵌套使用，且如果简单二值信号量当前已被某线程占用时，允许被删除。semaphore manager 组件是基于核心信号量和核心互斥体等实现的。

其实也可以用计数信号量来实现二值信号量的功能，这里没有采用这种方法的原因是直接实现的二值信号量和简单二值信号量更加简洁和高效，并提供了优先级继承算法和优先级天花板算法来解决优先级反转问题，同时也有利于系统的剪裁。

信号量的嵌套访问

当已经占有一个互斥信号量的任务企图获得同一个互斥信号量时，有可能会发生死锁。这是由于互斥信号量已经分配给了某任务，且没有被释放，但同时该任务又在等待该任务互斥信号量被释放，因此该任务将永远不会再执行。而在编写应用程序时又可能有类似这样的需要，比如应用程序中的 A、B 两个函数都需要获得某一共享对象，如果函数 A 在获得了共享对象以后调用 B，这个时候 A 还没有释放，但是它们运行在同一个线程内不会发生竞争，这样内核还是应该让 B 获取该对象。RTEMS 通过允许任务嵌套获取资源的机制来解决这个问题，在持有信号量时可以继续获得同一个信号量，但获取和释放操作必须配对，且不能出现交叉。简单二值信号量不允许嵌套访问，因此不可用于嵌套互斥访问，而主要用于同步。

优先级反转

在实时操作系统中，高优先级任务需要访问的资源被低优先级的任务占用，导致高优先级的任务被阻塞，高优先级任务必须等待低优先级任务释放资源后才能唤醒。如果这个低优先级的任务还被其他优先级中等的任务抢占了 CPU，那么优先级反转现象还有可能进一步恶化。因为低优先级任务无法继续，导致资源无法释放，高优先级的任务可能会被无限制延迟。 RTEMS 采用了优先级继承算法和优先级天花板算法两种机制解决优先级反转问题。

优先级继承算法

优先级继承是一种动态提高获得互斥共享资源的任务的优先级的算法。如果获得互斥资源的任务由低优先级变为高优先级，那么中等优先级的任务就不会抢占该任务；也就不会导致需要同一资源的其他高优先级任务因为低优先级任务而阻塞不确定的时间。

RTEMS 为本地的二值信号量提供优先级继承算法。当高优先级任务因为低优先级占据资源而阻塞，那么占据资源的低优先级任务的优先级将会提升至此高优先级任务的优先级。当占据资源的任务释

放信号量后，此任务的优先级将会恢复到原始状态。

RTEMS 优先级继承算法也考虑到了实时任务拥有多个资源的信号量这一情况。此时占据资源的任务的优先级将改变，新的优先级是所有阻塞任务的优先级的最小值（值越小，优先级越高）。只有当任务释放所有的信号量，他的优先级才恢复到原始水平。

优先级天花板算法

优先级天花板算法中，每个任务的优先级是已知的，每个任务要求的资源在执行前也是已知的。资源具有优先级，其优先级与预计要求占用此资源的任务中的最高优先级相同。要求占用资源的任务的优先级在得到资源后，会动态提升任务自身的优先级，新的优先级是任务此时所占用的所有资源中最高的资源优先级。该算法通过一次性改变任务优先级的方式解决任务优先级反转的问题（优先级继承算法中，每次有任务阻塞就可能需要改变一次优先级）。

RTEMS 为本地的二值信号量提供优先级天花板算法。当一个低优先级的任务获取资源的信号量，他的优先级会自动提高到资源的优先级天花板（Ceiling）的水平。该任务执行完对应的操作，释放了信号量，其优先级就会降低到原来的水平。

在大型实时应用系统中获取优先级天花板的操作将会很复杂。由于优先级天花板算法只用改变一次任务的优先级，所以比优先级继承算法更为高效。但是如果系统复杂，扫描将会占据信号量的所有任务的优先级同样也会降低执行效率。相对而言，优先级继承算法不需要预先获取任务的相关信息，所以也显得更为灵活。

RTEMS 的优先级天花板算法同样也适用于占据多个信号量的任务。此时需要对每个二进制信号量分别进行优先级天花板算法，然后获取所有优先级天花板中的最高优先级（求两次最高优先级）。直到完成所有的对互斥资源的操作，他才会将优先级恢复到初始水平。还有一个需要注意的情况是，优先级继承算法不能防止死锁。优先级天花板算法可以，但是必须是在系统运行前知道所有任务对所有资源的访问情况。

核心层的 semaphore handler 组件

信号量机制是在 RTEMS 的核心层通过 semaphore handler 组件的核心信号量来辅助实现，其中涉及对 CORE_semaphore_Control 结构体以及相关操作。semaphore handler 组件会对线程的执行状态产生影响。

关键数据结构

在 rtems 中，核心信号量是 semaphore handler 组件的主体，负责 core_semaphore 处理的内核代码在 cpukit/score 目录下，有关核心信号量的内核代码都存入在文件 coresem*.c 上。在核心信号量的实现中定义了如下几个关键数据结构：

☞ CORE_semaphore_Control：用来描述一个核心信号量

```
typedef struct {
    Thread_queue_Control    Wait_queue; //等待获得此核心信号量的线程等待队列
    CORE_semaphore_Attributes Attributes; //核心信号量的属性
    uint32_t                count;      //核心信号量的计数值
} CORE_semaphore_Control;
```

☞ CORE_semaphore_Attributes：用于描述核心信号量的属性

```
typedef struct {
    uint32_t                maximum_count; //核心信号量的最大计数值
    CORE_semaphore_Disciplines discipline; //表明等待线程是按 FIFO 还是优先级来排队
} CORE_semaphore_Attributes;
```

- **Core_semaphore_Blocking_option**: 获取核心信号量的可选等待方式

```
typedef enum {
    CORE_SEMAPHORE_NO_WAIT, //不等待获取核心信号量
    CORE_SEMAPHORE_BLOCK_FOREVER, //死等获取核心信号量
    CORE_SEMAPHORE_BLOCK_WITH_TIMEOUT, //等待有限时间获取核心信号量
    CORE_SEMAPHORE_BAD_TIMEOUT, //获取核心信号量的等待时间有错（少见的情况）
} Core_semaphore_Blocking_option;
```

- **CORE_semaphore_Status**: 核心信号量操作的返回值

```
typedef enum {
    CORE_SEMAPHORE_STATUS_SUCCESSFUL,
    CORE_SEMAPHORE_STATUS_UNSATISFIED_NOWAIT,
    CORE_SEMAPHORE_WAS_DELETED,
    CORE_SEMAPHORE_TIMEOUT,
    CORE_SEMAPHORE_MAXIMUM_COUNT_EXCEEDED
} CORE_semaphore_Status;
```

关键实现函数

对核心信号量的操作主要包括：

- ✎ **_CORE_semaphore_Initialize**: 核心信号量的初始化
- ✎ **_CORE_semaphore_Seize**: 获取核心信号量
- ✎ **_CORE_semaphore_Surrender**: 释放核心信号量
- ✎ **_CORE_semaphore_Flush**: 清除核心信号量的任务等待队列

下面分别进行分析。

初始化核心信号量

函数 **_CORE_semaphore_Initialize** 用于在创建核心信号量时初始化一个核心信号量控制块。此函数会被 **rtems_semaphore_create** 函数调用，以支持 **RTEMS** 的信号量机制的实现。

函数调用信息

```
void _CORE_semaphore_Initialize(
    CORE_semaphore_Control *the_semaphore, //需要初始化的核心信号量控制块
    CORE_semaphore_Attributes *the_semaphore_attributes, //指定的核心信号量属性
    uint32_t initial_value //核心信号量的初始值
)
```

函数功能描述

具体实现如下：

1. 根据传入的 **the_semaphore_attributes** 和 **initial_value** 初始化核心信号量控制块的 **Attributes** 和 **count** 成员；
2. 调用 **_Thread_queue_Initialize** 初始化该核心信号量的等待队列 **Wait_queue**，并根据指定的属性值 **the_semaphore_attributes** 来确定任务的排队策略是 **FIFO** 还是基于优先级，指定线程等待队列的状态为 **STATES_WAITING_FOR_SEMAPHORE**，线程等待队列的超时状态设为 **CORE_SEMAPHORE_TIMEOUT**。

□取核心信号量

函数 `_CORE_semaphore_Seize` 用于获取一个核心信号量，如果可以等待，可能会使线程阻塞起来。

函数调用信息

```
void _CORE_semaphore_Seize(
    CORE_semaphore_Control *the_semaphore, //需要获取的核心信号量控制块
    Objects_Id id, //获取的核心信号量的 id
    Core_semaphore_Blocking_option wait, //指定是否可以等待
    Watchdog_Interval timeout //等待的 tick 数，0表示永远等待
)
```

函数功能描述

具体实现流程如下：

1. 把调用线程的等待的返回值 `Wait.return_code` 设为 `CORE_SEMAPHORE_STATUS_SUCCESSFUL`；
2. 屏蔽中断；
3. 判断核心信号量的值是否为0，如果不是0，则说明还有资源，那么就把核心信号量的值 `count` 减1，然后使能中断，成功返回；
4. 如果核心信号量的值已经为0，则进一步根据 `wait` 输入参数判断调用线程是否愿意等待等待时间是否有错，如果调用线程不想等待（`wait` 值为 `CORE_SEMAPHORE_NO_WAIT`）或者等待时间有错（`wait` 值为 `CORE_SEMAPHORE_BAD_TIMEOUT`），则使能中断，把它的等待返回值设为 `CORE_SEMAPHORE_STATUS_UNSATISFIED_NOWAIT` 或（`CORE_SEMAPHORE_BAD_TIMEOUT_VALUE`），然后返回；
5. 如果线程愿意等待（`wait` 值为 `CORE_SEMAPHORE_BLOCK_FOREVER` 或 `CORE_SEMAPHORE_BLOCK_WITH_TIMEOUT`），则调用函数 `_Thread_queue_Enter_critical_section`，使核心信号量的线程等待队列的同步状态变为 `THREAD_QUEUE_NOTHING_HAPPENED`；把调用线程的等待队列 `Wait.queue` 设为这个核心信号量的等待队列，把调用线程的等待对象的 `id`（`Wait.id`）设为输入参数 `id`；使能中断，调用 `_Thread_queue_Enqueue` 把调用线程加入到核心信号量的线程等待队列，并指定超时时间为 `timeout`；
6. 正常返回。

□放核心信号量

函数 `_CORE_semaphore_Surrender` 用于释放一个核心信号量。

函数调用信息

```
CORE_semaphore_Status _CORE_semaphore_Surrender(
    CORE_semaphore_Control *the_semaphore, //需要获取的核心信号量控制块
    Objects_Id id, //释放的对象的 id
    CORE_semaphore_API_mp_support_callout api_semaphore_mp_support
)
```

其中的 `id` 是封装了 `CORE_semaphore_Control` 结构体的同步对象（比如 `Semaphore_Control` 结构体）的 `id`。`api_semaphore_mp_support` 用于对多处理器核心信号量的支持，这里不详细分析。

函数功能描述

具体实现流程如下：

1. 设置返回状态为 `CORE_SEMAPHORE_STATUS_SUCCESSFUL`；
2. 如果核心信号量的等待队列中有线程在等待，则调用 `_Thread_queue_Dequeue` 函数取出一个线程

(出队操作会使线程就比绪)，然后返回；

3. 如果没有线程在等待该核心信号量，则屏蔽中断，判断核心信号量的值是否小于该核心信号量的属性当中指定的它的最大值，如果小于则把核心信号量的值加1，如是不小于则把返回状态设为 CORE_SEMAPHORE_MAXIMUM_COUNT_EXCEEDED；使能中断，返回。

清空核心信号量的线程等待队列

函数 _CORE_semaphore_Flush 用于清空挂在核心信号量的线程等待队列上的线程。

函数调用信息

```
void _CORE_semaphore_Flush(  
    CORE_semaphore_Control *the_semaphore, //需要清空队列的核心信号量控制块  
    Thread_queue_Flush_callout remote_extract_callout,  
    uint32_t status //传给线程的等待返回值  
)
```

函数功能描述

函数 _CORE_semaphore_Flush 用于清空核心信号量的线程等待队列，删除一个核心信号量时会调用这个函数。它的具体实现就是调用 _Thread_queue_Flush 把核心信号量的线程等待队列中的线程一一出队，并且把它们的等待状态设为 status。

核心层的 mutex handler 组件

RTEMS 需要为每个要互斥访问的共享资源提供保护手段。互斥体(mutex)是完成这一手段的有效手段。在 RTEMS 中，互斥体机制是通过表示核心互斥体的 CORE_mutex_Control 结构体及相关操作实现了对线程的阻塞、调度等各项功能。

关键数据结构

CORE_mutex_Control, 这个结构定义了一个互斥体的所有信息，其中包括在这个 核心互斥体上阻塞的线程的等待队列 wait_queue，这个核心互斥体的属性 Attributes，核心互斥体的状态 lock，这个核心互斥体目前被嵌套申请的次数 nest_count，另外还有等待线程数 blocked_count 以及目前持有这个锁的线程 holder 和线程的对象标识 holder_id。CORE_mutex_control 作为 RTEMS 当中核心互斥体的关键数据结构在核心互斥体的管理过程中被反复使用。

```
typedef struct {  
    Thread_queue_Control    Wait_queue;  
    CORE_mutex_Attributes  Attributes;  
    uint32_t                lock;  
    uint32_t                nest_count;  
    uint32_t                blocked_count;  
    Thread_Control          *holder;  
    Objects_Id              holder_id;  
} CORE_mutex_Control;
```

CORE_mutex_Disciplines 结构定义了核心互斥体的等待队列中的调度策略，从这里我们可以看到，核心互斥体等待队列中的调度策略有四个，分别是：FIFO（先进先出策略），Priority（优先级策略），Priority_Inherit（优先级继承策略），Priority_Ceiling（优先级天花板策略）。关于优先级继承策略和优先级天花板策略是为了解决优先级反转问题而设计的。

```
typedef enum {  
    CORE_MUTEX_DISCIPLINES_FIFO,
```

```

CORE_MUTEX_DISCIPLINES_PRIORITY,
CORE_MUTEX_DISCIPLINES_PRIORITY_INHERIT,
CORE_MUTEX_DISCIPLINES_PRIORITY_CEILING
}    CORE_mutex_Disciplines;

```

CORE_mutex_Attributes, 这个结构体定义了一个核心互斥体的各种属性, 包括: 嵌套获取该核心互斥体时的行为属性 lock_nesting_behavior, 是否只有拥有者才可以释放的属性 only_owner_release, 等待队列的排队策略属性 discipline, 天花板优先级属性 priority_ceiling。

```

typedef struct {
    CORE_mutex_Nesting_behaviors lock_nesting_behavior;
    boolean                      only_owner_release;
    CORE_mutex_Disciplines       discipline;
    Priority_Control              priority_ceiling;
}    CORE_mutex_Attributes;

```

关键实现函数

在具体实现中, RTEMS 为核心互斥体提供了如下几个函数:

- ✎ _CORE_mutex_Initialize: 初始化核心互斥体
- ✎ _CORE_mutex_Seize: 获取核心互斥体
- ✎ _CORE_mutex_Seize_interrupt_trylock: 尝试获取核心互斥体
- ✎ _CORE_mutex_Seize_interrupt_blocking: 等待获取核心互斥体
- ✎ _CORE_mutex_Surrender: 释放核心互斥体
- ✎ _CORE_mutex_Flush: 清空核心互斥体的线程等待队列

下面将分别对它们进行具体分析。

初始化核心互斥体

函数调用信息

```

void _CORE_mutex_Initialize(
    CORE_mutex_Control    *the_mutex, //初始化的核心互斥体控制块指针
    CORE_mutex_Attributes *the_mutex_attributes, //核心互斥体的属性
    uint32_t              initial_lock //初始的状态
)

```

函数功能描述

函数_CORE_mutex_Initialize 主要用于在创建信号量 (RTEMS CLASSIC API) 或互斥体 (POSIX API) 时对其封装的核心互斥体控制块初始化。

具体实现如下:

1. 把传入的 the_mutex_attributes 和 initial_lock 分别赋给核心互斥体控制块 Attributes 和 lock 成员, 并把 blocked_count 设为0。
2. 根据 initial_lock 的值对核心互斥体控制块的其它成员进行赋值, 如果 initial_lock 的值是 CORE_MUTEX_LOCKED, 这表示在初始化的时候, 创建它的线程就占用了它, 那么, 就把成员变量 nest_count 设为1, 且把成员变量 holder 设为现在正在运行的线程, 把成员变量 holder_id 设为这个线程的 id; 如果这个核心互斥体的等待队列采用了优先级继承或者优先级天花板策略, 则把该线程的成员变量 resource_count 加1, 表明当前调用线程又获取了一个资源;
3. 如果 initial_lock 值不是 CORE_MUTEX_LOCKED, 那么表明要建立一个空闲的核心互斥体, 即建立之初并没有线程马上占用它。可能程序执行的后续过程中会用到, 那么就把相应的成员变量赋值为0或者 NULL;

4. 调用 `_Thread_queue_Initialize` 函数来初始化核心互斥体的线程等待队列，在初始化线程等待队列的参数设定中，根据指定的属性值 `the_mutex_attributes` 来确定任务的排队策略是 FIFO 还是基于优先级，指定线程等待队列的状态为 `STATES_WAITING_FOR_MUTEX`，线程等待队列的超时状态设为 `CORE_MUTEX_TIMEOUT`；
5. 正常返回。

□取核心互斥体

函数 `_CORE_mutex_Seize` 用于获得一个核心互斥体，出于性能方面的考虑，它是通过宏定义来实现的。
函数调用信息

```
#define _CORE_mutex_Seize(\
    _the_mutex, _id, _wait, _timeout, _level )
...
```

其中，

- ✎ `_the_mutex`：尝试获取的核心互斥体控制块
- ✎ `_id`：获取的核心互斥体对象的 id
- ✎ `_wait`：线程是否愿意等待
- ✎ `_timeout`：可以等待的最大 tick 数
- ✎ `_level`：临时变量用于保存中断使能等级

函数功能描述

具体实现如下：

1. 判断当前状态，如果是在禁止调度的状态下且线程愿意等待且处于开始启动多任务的系统状态（`>= SYSTEM_STATE_BEGIN_MULTITASKING`）调用这个函数，则发生异常；
2. 否则调用 `_CORE_mutex_Seize_interrupt_trylock` 函数尝试获取这个核心互斥体，如果这个函数返回 0，表明获得了核心互斥体，则直接返回；
3. 如果 `_CORE_mutex_Seize_interrupt_trylock` 的值为 1，表明没有获得核心互斥体，则需要进一步判断调用线程是否愿意等待，如果不能等待，则使能中断，把线程的等待返回值设为 `CORE_MUTEX_STATUS_UNSATISFIED_NOWAIT`，返回；
4. 如果需要等待，则调用 `_Thread_queue_Enter_critical_section` 把等待线程队列的同步状态设置为 `THREAD_QUEUE_NOTHING_HAPPENED`，为使核心互斥体的等待队列对调用线程进行入队处理做准备，把调用线程的等待的队列 `Wait.queue` 设为这个核心互斥体的队列，把线程的等待对象的 id，即 `Wait.id` 设为 `_id`，然后禁止调度，使能中断，调用 `_CORE_mutex_Seize_interrupt_blocking` 函数将调用线程阻塞起来；
5. 正常返回。

函数 `_CORE_mutex_Seize_interrupt_blocking` 的实现请参考下面的“等待获取核心互斥体”分析。

等待□取核心互斥体

函数 `_CORE_mutex_Seize_interrupt_blocking` 用于在获取一个核心互斥体失败的时候阻塞获取线程，也就是目前正在执行的线程，该函数是核心互斥体操作中的一个核心函数。

函数□用信息

```
void _CORE_mutex_Seize_interrupt_blocking(
```

```

CORE_mutex_Control *the_mutex, //获取的核心互斥体控制块指针
Watchdog_Interval timeout //可以等待的时间, 0代表永久等待
)

```

函数功能描述

具体实现如下:

1. 如果核心互斥体使用的是优先级继承算法, 并且目前执行的线程的优先级高于核心互斥体的持有者的优先级, 那么把核心互斥体持有者的优先级提高到和目前执行的线程的优先级一样的高度;
2. 把核心互斥体的成员变量 blocked_count 加1, 调用_Thread_queue_Enqueue 函数把正在执行的线程放到核心互斥体的等待队列当中;
3. 调用_Thread_Enable_dispatch 函数重新调度;
4. 返回。

□□□取核心互斥体

函数_CORE_mutex_Seize_interrupt_trylock 尝试获取核心互斥体, 如果无法获取则返回, 不会被阻塞起来。

函数调用信息

```

RTEMS_INLINE_ROUTINE int _CORE_mutex_Seize_interrupt_trylock(
CORE_mutex_Control *the_mutex, //需要获取的核心互斥体控制块的指针
ISR_Level          *level_p    //保存中断使能等级的变量的地址
)

```

函数功能描述

具体实现如下:

1. 将调用线程的等待返回值 Wait.return_code 设为 CORE_MUTEX_STATUS_SUCCESSFUL;
2. 判断这个核心互斥体是否已经被锁, 如果没有被锁, 则获取线程可以获得这个核心互斥体, 那么需要执行的操作包括: 将核心互斥体成员变量—状态 lock 设为 CORE_MUTEX_LOCKED, 核心互斥体的成员变量—所有者 holder 设为_Thread_Executing (当前运行线程), 核心互斥体的成员变量—所有者的对象 id 设为当前运行线程的 id, 核心互斥体的成员变量—嵌套次数 nest_count 设为1, 如果采用了优先级继承或者优先级天花板算法, 则将调用线程的 resource_count 加1, 表明调用线程又获取了一个资源;
3. 如果这个核心互斥体的等待策略使用的不是优先级天花板算法, 则使能中断, 返回0;
4. 如果采用了优先级天花板算法, 则比较调用线程的当前优先级和核心互斥体的天花板优先级 Attributes.priority_ceiling, 如果二者相等, 则使能中断, 返回0;
5. 如果前者大, 则先屏蔽调度, 使能中断, 然后调用_Thread_Change_priority 函数, 把调用线程的优先级提高到核心互斥体的天花板优先级, 使能调度, 返回0;
6. 否则 (即说明当前线程的优先级超出核心互斥体的天花板优先级), 是一种错误情况, 不能获得这个核心互斥体, 这时要把调用线程的成员变量 Wait.return_code 设为 CORE_MUTEX_STATUS_CEILING_VIOLATED, 再把核心互斥体的成员变量 nest_count 恢复为0, 把调用线程的成员变量 resource_count 减1, 然后返回0;
7. 执行到此, 表示如果这个核心互斥体已经被锁; 如果这个核心互斥体已经被锁, 则判断当前运行的线程是否是这个核心互斥体的所有者, 如果是, 则根据这个核心互斥体的 lock_nesting_behavior 属性值采取不同的操作: 如果是 CORE_MUTEX_NESTING_ACQUIRES, 则把核心互斥体成员变量 nest_count 加1, 返回0; 如果是 CORE_MUTEX_NESTING_IS_ERROR, 则把线程的成员变量—等待返回值 Wait.return_code 设为 CORE_MUTEX_STATUS_NESTING_NOT_ALLOWED, 然后

返回0；如果是 CORE_MUTEX_NESTING_BLOCKS，则返回1；

8. 如果这个核心互斥体已经被锁，并且它的所有者不是当前运行的线程，那么返回1。

当函数 `_CORE_mutex_Seize` 发现 `_CORE_mutex_Seize_interrupt_trylock` 这个函数的返回值为1时，会调用函数 `_CORE_mutex_Seize_interrupt_blocking` 把线程阻塞起来。

□放核心互斥体

函数 `_Core_mutex_surrender` 用于释放一个核心互斥体。但有线程在等待获取这个核心互斥体时，此函数会让等待线程处于就绪态，并把核心互斥体给此线程。

函数调用信息

```
CORE_mutex_Status _CORE_mutex_Surrender(  
CORE_mutex_Control *the_mutex, //需要释放的核心互斥体  
Objects_Id id, //核心互斥体的对象 id  
CORE_mutex_API_mp_support_callout api_mutex_mp_support  
)
```

其中，`api_mutex_mp_support` 是释放一个远程的核心互斥体会被调用的回调函数，这里不详细分析。

函数功能描述

具体实现如下：

1. 判断这个核心互斥体是否只允许它的持有者释放（即表示是否不允许其它的线程或者同步信号例程强行释放它），如果是，则进一步判断当前运行线程是否是该核心互斥体的持有者，如果不是，那么返回 `CORE_MUTEX_STATUS_NOT_OWNER_OF_RESOURCE`；
2. 如果这个核心互斥体可以由其他线程释放或者持有者正在运行，那么判断核心互斥体的成员变量 `nest_count`，如果 `nest_count` 为0，那么表明该核心互斥体还没有被持有，返回 `CORE_MUTEX_STATUS_SUCCESSFUL`；
3. 如果核心互斥体的成员变量 `nest_count` 不为0，把 `nest_count` 减1，如果仍不为0，则根据核心互斥体的 `lock_nesting_behavior` 值设置不同的返回值返回；
4. 如果 `nest_count` 减1后为0，且这个核心互斥体采用了优先级继承或者优先级天花板策略，则把持有者的 `resource_count` 减1，然后把 `holder` 和 `holder_id` 清空；继续判断是否使用了优先级继承或者优先级天花板策略，如果是，则进一步判断，在持有者线程的成员变量 `resource_count` 为0并且持有者线程的成员变量 `real_priority` 不等于持有者线程的成员变量 `current_priority` 情况下，需要调用函数 `_Thread_Change_priority` 把持有者线程的优先级改为原来的值；
5. 如果核心互斥体的等待队列不为空，则取出一个线程，重新设置核心互斥体的成员变量——所有者线程和所有者线程 `id`，并把成员变量 `nest_count` 设为1；
6. 如果等待策略是基于 FIFO 或者基于优先级的，则直接返回 `CORE_MUTEX_STATUS_SUCCESSFUL`；如果等待策略是优先级继承的，则把获得核心互斥体的线程的 `resource_count` 加1，然后返回 `CORE_MUTEX_STATUS_SUCCESSFUL`；如果等待策略是优先级天花板的，那么把获得核心互斥体的线程的 `resource_count` 加1，然后判断这个线程的优先级是否比天花板优先级低，如果是则调用函数 `_Thread_Change_priority` 把这个线程提高到天花板优先级，然后返回 `CORE_MUTEX_STATUS_SUCCESSFUL`；
7. 如果核心互斥体的等待队列为空，则把这个核心互斥体的成员变量状态 `lock` 设为 `CORE_MUTEX_UNLOCKED`，然后返回 `CORE_MUTEX_STATUS_SUCCESSFUL`。

清空核心互斥体□程等待□列

函数调用信息

```
void _CORE_mutex_Flush(
    CORE_mutex_Control    *the_mutex, //需要处理的核心互斥体控制块
    Thread_queue_Flush_callout remote_extract_callout,
    uint32_t              status      //传给线程的等待返回值
)
```

函数功能描述

函数 `_CORE_mutex_Flush` 用于清空核心互斥体的线程等待队列，删除一个核心互斥体时会调用这个函数，它的具体实现就是调用 `_Thread_queue_Flush` 把核心互斥体的线程等待队列中的线程一一出队，并且把它们的等待状态设为 `status`。

系统服务层的 semaphore manager 组件

信号量的系统服务层对核心层的 `_CORE_mutex` 和 `_CORE_semaphore` 进行封装，提供统一的接口。在创建一个 semaphore 时，根据指定的属性来决定它的核心对象采用 `_CORE_mutex` 还是 `_CORE_semaphore`，对这个信号量的操作也会根据这个信号量的属性来选择相应的核心对象的操作函数。

关键数据结构

CLASSIC API 中用于管理信号量的结构是 `Semaphore_Control`，它的定义如下：

```
typedef struct {
    Objects_Control      Object;
    rtems_attribute      attribute_set;

    union {
        CORE_mutex_Control    mutex;
        CORE_semaphore_Control semaphore;
    } Core_control;
} Semaphore_Control;
```

它的成员包括代表这个信号量的对象 `Object` 成员，信号量的属性集 `attribute_set` 以及它的核心组件 `Core_control`，它是一个由 `CORE_mutex_Control` 和 `CORE_semaphore_Control` 组成的联合，这说明了系统会根据用户创建的是一个多值信号量还是一个二值信号量来选择使用它们中的一个。`Semaphore_Control` 代表了一个信号量对象，如果核心信号量和核心互斥体也是对象，则 `Semaphore_Control` 结构体就隐含了三个 `Object`，不利用管理。这也是为何没有把 `CORE_mutex_Control` 和 `CORE_semaphore_Control` 基于对象构建的一个原因。

其中信号量的属性包括：

- ☒ `RTEMS_FIFO`：等待信号量的任务按照 FIFO 排序（默认）
- ☒ `RTEMS_PRIORITY`：等待信号量的任务按照优先级排序
- ☒ `RTEMS_BINARY_SEMAPHORE`：信号量是二进制信号量
- ☒ `RTEMS_COUNTING_SEMAPHORE`：信号量是计数信号量（默认）
- ☒ `RTEMS_SIMPLE_BINARY_SEMAPHORE`：二进制信号量，并且不允许嵌套，允许对锁住的信号量进行删除操作。
- ☒ `RTEMS_NO_INHERIT_PRIORITY`：不使用优先级继承（默认）
- ☒ `RTEMS_INHERIT_PRIORITY`：使用优先级继承
- ☒ `RTEMS_PRIORITY_CEILING`：使用优先级天花板
- ☒ `RTEMS_NO_PRIORITY_CEILING`：不使用优先级天花板（默认）

☞ RTEMS_LOCAL: 本地任务（默认）

☞ RTEMS_GLOBAL: 全局任务

上面的各个属性分量都是正交的单位向量。所以对他们进行或运算和进行加法操作的效果是一样的。有的分量被标明是“默认”表明信号量创建的时候就是用这些缺省值。所有缺省值的累加可以用向量 RTEMS_DEFAULT_ATTRIBUTES 表示。同时，用户在创建信号量时可以指定不同的属性构成它的属性集。

关键实现函数

CLASSIC API 中对信号量的操作主要包括：

- ☞ rtems_semaphore_create: 创建一个信号量
- ☞ rtems_semaphore_ident: 获取信号量的 ID
- ☞ rtems_semaphore_delete: 删除一个信号量
- ☞ rtems_semaphore_obtain: 获得信号量
- ☞ rtems_semaphore_release: 释放信号量
- ☞ rtems_semaphore_flush: 解除所有等待该信号量的任务的阻塞

这些操作都是提供给用户来操作信号量的，但是在系统中还需要有一个结构对所有的信号量进行管理，这个变量就是 Objects_Information 类型的 _Semaphore_Information。在系统启动的时候会调用 _Semaphore_Manager_initialization 来初始化它，这个函数就是调用了函数 _Objects_initialize_information 来初始化信号量的对象信息管理。

□ 建一个信号量

函数调用信息

```
rtems_status_code rtems_semaphore_create(
    rtems_name      name,      //用户定义的信号量名
    uint32_t        count,     //信号量的初始值
    rtems_attribute  attribute_set, //信号量的属性集
    rtems_task_priority priority_ceiling, //信号量的优先级天花板
    rtems_id        *id        //信号量 id 的指针
)
```

函数功能描述

函数 rtems_semaphore_create 用于创建信号量并设置初始值。

具体操作如下：

1. 检查传入的 name 和 id 的合法性；
2. 如果信号量的属性集中指定了优先级继承或者优先级天花板策略，则判断要创建的信号量是否是基于优先级的二值信号量/简单二值信号量，如果都不是，则返回 RTEMS_NOT_DEFINED；
3. 如果要创建的信号量不是计数信号量并且初始值大于1，则返回 RTEMS_INVALID_NUMBER；
4. 屏蔽调度；
5. 调用 _Semaphore_Allocate 函数分配一个信号量控制块，如果分配失败，表明超出了定制的信号量个数，使能中断，返回 RTEMS_TOO_MANY；_Semaphore_Allocate 函数实际工作是：

```
return (Semaphore_Control *) _Objects_Allocate( &_Semaphore_Information );
```

6. 把传入的 attribute_set 赋给信号量控制块的 attribute_set；
7. 如果要创建的信号量不是计数信号量，也就是二值信号量，则根据它的属性来设置它的核心组件的等待策略，即 FIFO、优先级、优先级继承和优先级天花板策略其中的一种；

8. 然后继续判断要创建的信号量是不是二值信号量，如果不是，则不支持嵌套获取，并且可以由非持有者释放，即把它的属性的 `lock_nesting_behavior` 设为 `CORE_MUTEX_NESTING_BLOCKS`，属性的 `only_owner_release` 设为 `FALSE`；如果不是简单二值信号量，即一般的二值信号量，则支持嵌套，把它的属性的 `lock_nesting_behavior` 设为 `CORE_MUTEX_NESTING_ACQUIRES`，如果它的等待策略是 `FIFO` 或者优先级，则 `only_owner_release` 设为 `FALSE`，否则如果是优先级继承或者是优先级天花板则把 `only_owner_release` 设为 `TRUE`；然后，把传入的参数优先级天花板 `priority_ceiling` 赋给临时变量 `the_mutex_attributes` 的成员变量 `priority_ceiling`，如果传入的 `count` 值为1，则初始的状态 `lock` 为 `CORE_MUTEX_UNLOCKED`，如果为0，则状态为 `CORE_MUTEX_LOCKED`，最后调用 `_CORE_mutex_Initialize` 来初始化它的核心互斥体对象

```
_CORE_mutex_Initialize(  
    &the_semaphore->Core_control.mutex,  
    &the_mutex_attributes,  
    lock  
);
```

9. 如果要创建的是计数信号量，根据创建时指定的属性，设置它的核心组件等待策略，是 `FIFO` 还是优先级，把信号量的成员变量信号量最大值 `maximum_count` 设为 `0xFFFFFFFF`，然后调用 `_CORE_semaphore_Initialize` 初始化它的核心信号量对象

```
_CORE_semaphore_Initialize(  
    &the_semaphore->Core_control.semaphore,  
    &the_semaphore_attributes,  
    count  
)
```

10. 调用 `_Objects_Open` 把这个信号量注册到 `_Semaphore_Information` 中，然后信号量对象的 `id` 保存在入口参数指定的地址里；
11. 使能调度，返回 `RTEMS_SUCCESSFUL`；

信号量


函数 `rtems_semaphore_delete` 用于删除一个信号量。

函数调用信息

```
rtems_status_code rtems_semaphore_delete(  
    rtems_id id          //要删除的信号量的 ID  
)
```

函数功能描述

具体实现如下：

1. 根据信号量 `ID` 调用 `_Semaphore_Get` 获得要删除的信号量的控制块，并获得它的位置；
2. 如果它的位置表示它是本地的对象，则进行如下工作：
 这进一步判断它是否一个计数信号量，是则直接调用 `_CORE_semaphore_Flush` 清空它的等待队列；如果是一个二值信号量，则需要判断当前状态是否已经被锁，如果被锁并且不是简单二值信号量，那么就使能调度，然后返回 `RTEMS_RESOURCE_IN_USE`，也就是说简单二值信号量可以直接删除；如果没有被锁，或者是简单二值信号量，则调用 `_CORE_mutex_Flush` 清空它的等待队列；
3. 调用 `_Objects_Close` 函数把这个对象从 `_Semaphore_Information` 中注销，调用 `_Semaphore_Free` 函数释放这个控制块，使能调度，返回 `RTEMS_SUCCESSFUL`。

□得一个信号量

函数 `rtems_semaphore_obtain` 用于获得一个信号量，如果可以等待，在当前信号量无法获取的情况下，任务会被阻塞起来。

函数调用信息

```
rtems_status_code rtems_semaphore_obtain(
  rtems_id id,           //要获得的信号量 id
  uint32_t option_set,   //等待选项
  rtems_interval timeout //等待的 tick 数(0表示一直等待)
)
```

函数功能描述

具体实现如下：

1. 调用 `_Semaphore_Get_interrupt_disable` 获得这个信号量的控制块，并得到它的位置，这个函数 `_Semaphore_Get` 的区别在于，它调用的不是 `_Objects_Get` 而是 `_Objects_Get_isr_disable`，从这个函数的名字也可以看出，它的作用也是获得一个对象的控制块，只不过会调用 `_ISR_Disable` 函数屏蔽中断；
2. 如果这个对象是本地的，判断这个信号量是计数信号量还是二值信号量，如果不是计数信号量（显然是二值信号量），调用 `_CORE_mutex_Seize` 函数获得这个核心互斥体，具体处理是：如果不想等待，则调用 `_ISR_Enable` 使能中断，然后根据在这个函数中设置的线程的等待返回值 `Wait.return_code` 返回相应的状态；如果想等待，则调用 `_Thread_Disable_dispatch` 函数屏蔽调度，调用 `_ISR_Enable` 使能中断，把调用线程加入线程等待队列，调用 `_Thread_Enable_dispatch` 函数使能调度。
3. 如果是计数信号量，则调用 `_CORE_semaphore_Seize_isr_disable` 获取信号量，具体处理是：如果可以获得信号量，则把信号量计数值减一，使能中断，返回；如果得不到信号量，则又分两种情况处理：（a）如果不想等待，则调用 `_ISR_Enable` 使能中断，然后根据在这个函数中设置的线程的等待返回值 `Wait.return_code` 返回相应的状态；（b）如果想等待，则调用 `_Thread_Disable_dispatch` 函数屏蔽调度，调用 `_ISR_Enable` 使能中断，把调用线程加入线程等待队列，调用 `_Thread_Enable_dispatch` 函数使能调度。

□放一个信号量

函数 `rtems_semaphore_release` 用于释放一个信号量。

函数调用信息

```
rtems_status_code rtems_semaphore_release(
  rtems_id id //释放的信号量的 id
)
```

函数功能描述

具体实现如下：

1. 调用函数 `_Semaphore_Get` 获得这个信号量的控制块和它的位置；
2. 如果它是本地的对象，则进一步判断它是否是计数信号量，如果不是，调用 `_CORE_mutex_Surrender` 释放核心互斥体，然后使能调度，并根据 `_CORE_mutex_Surrender` 返回不同的状态值；
3. 如果是计数信号量，调用 `_CORE_semaphore_Surrender` 释放这个核心信号量，然后使能调度，并根据 `_CORE_semaphore_Surrender` 返回不同的状态值。

清空信号量的等待队列

函数 `rtems_semaphore_flush` 用于清空一个信号量的等待队列，即解决所有等待任务的阻塞。

函数接口信息

```
rtems_status_code rtems_semaphore_flush(
    rtems_id id //被操作的信号量的 ID
)
```

函数功能描述

，具体实现如下：

1. 调用函数 `_Semaphore_Get` 获得这个信号量的控制块和它的位置；
2. 如果它是本地的对象，则进一步判断它是否是计数信号量，如果不是，调用 `_CORE_mutex_Flush` 来清空它的等待队列；
3. 如果是计数信号量，调用 `_CORE_semaphore_Flush` 来清空它的等待队列；
4. 使能调度，返回 `RTEMS_SUCCESSFUL`。

获取信号量的 ID

函数接口信息

```
rtems_status_code rtems_semaphore_ident(
    rtems_name name, //信号量的名字
    uint32_t node, //查找的节点范围
    rtems_id *id //保存 ID 的指针
)
```

函数功能描述

函数 `rtems_semaphore_ident` 就是根据信号量的名字，调用 `_Objects_Name_to_id` 函数，在 `_Semaphore_Information` 上查找出信号量的 ID。

应用举例

下面是使用信号量互斥的采用读者-写者策略的读者-写者问题。主要包括三个部分：初始读者、读者任务、写者任务（源程序附后，需理解）。

初始读者完成的工作包括：

- ❏ 用 `rtems_semaphore_create` 创建信号量 `x`，用于保护各读者任务共享 `readcount` 的互斥，创建信号量 `wsem` 用于读者任务和写者之间的互斥
- ❏ 随机创建一定数目的读者任务和写者任务，并启动这些任务

读者任务的工作包括：

1. 执行 `rtems_semaphore_obtain`，等待至信号量 `x` 可用，即 `readcount` 可以被修改，然后执行 `readcount++` 操作；
2. 判断这是否是第一个读任务，如果是，那么需判断共享数据区是否可用——是否被写任务占用，通过调用 `rtems_semaphore_obtain` 获取信号量 `wsem` 来等待并获得共享数据区的控制权；
3. 如果 `readcount` 不为1说明此读任务执行前已有其他读任务正在运行，即对于共享数据区的操作在读者，所以它可以并行进行读操作；
4. 调用 `rtems_semaphore_release`，释放对信号量 `x` 的独占，这样再有读任务启动时，可以顺利执行；
5. 开始进行读操作，这里执行的具体内容是向屏幕输出读的页码；
6. 读操作结束，该读任务结束，需要在 `readcount` 中-1。执行的步骤同样需要先通过调用 `rtems_semaphore_obtain` 获得对信号量 `x` 的控制权，再将 `readcount--`。这时需判断减完之后的结果是否为0，如果为0说明没有读操作正在执行，则释放掉对信号量 `wsem` 的控制，使得写任务

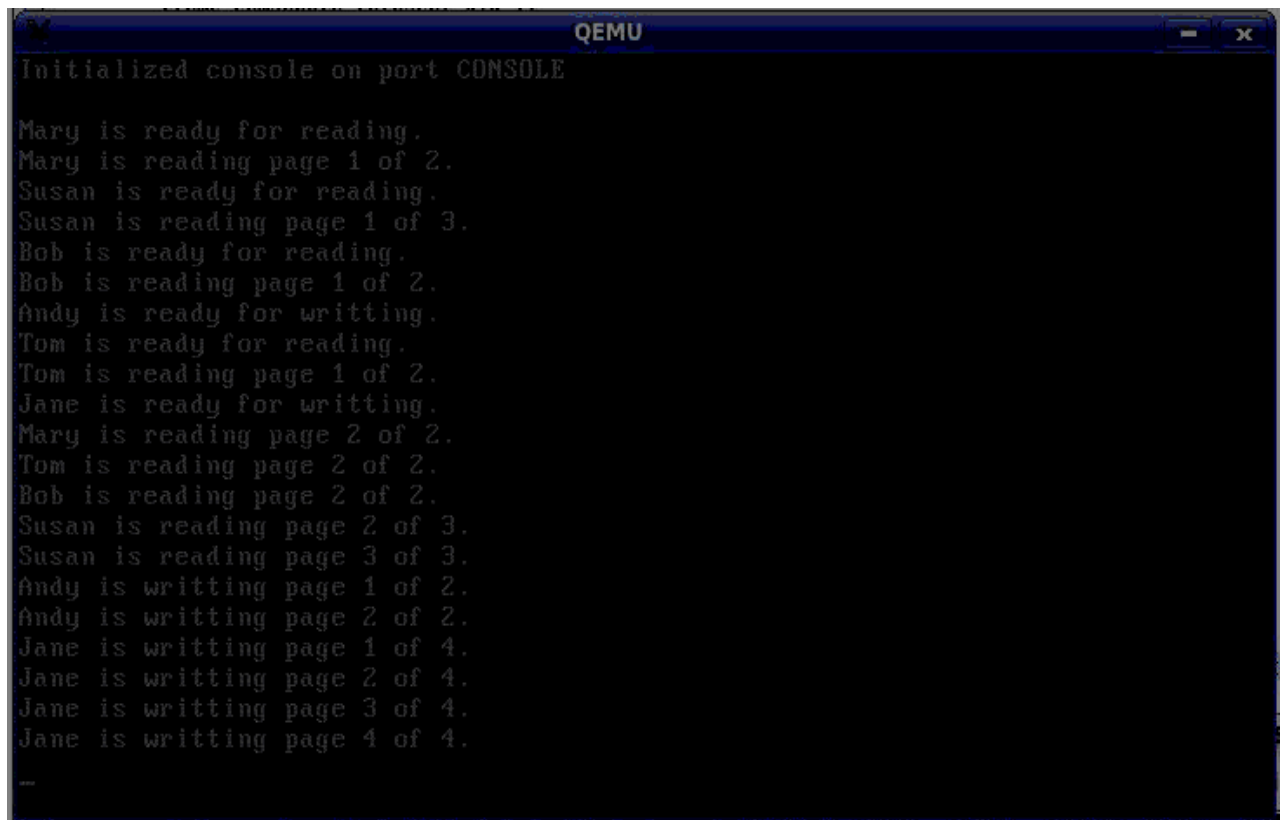
可以得到它；

7. 对 `readcount` 的操作结束，释放对信号量 `x` 的控制。

写者任务的工作包括：

1. 执行 `rtems_semaphore_obtain`，判断信号量 `wsem` 是否可用，如果可用则写任务运行，否则将该写任务放入等待队列；
2. 执行写操作，这里只是用一个循环输出正在写入的信息到屏幕；
3. 写操作完毕，执行 `rtems_semaphore_release`，释放对信号量 `wsem` 的控制权。

运行结果如下图所示：



消息队列

消息队列是操作系统提供了一种任务间缓冲通信的方式，发送消息的操作将消息放入队列，如果消息队列满，这个操作会被阻塞起来，而接收消息的操作则将消息从队列中取出，如果消息队列为空，这个操作也会被阻塞起来。队列中的消息可以按优先级排列，消息的长度也是可以由用户定义的。对操作系统来说，消息就是无特定含义的字节流，而应用程序可以根据确定的格式来使用消息。

核心层 message queue handler 组件

RTEMS 核心层中的 message queue handler 组件主要包括核心消息队列的 `CORE_message_queue_Control` 结构体及相关操作。message queue handler 组件会对线程的执行状态产生影响。

关键数据结构

核心消息队列的主要数据结构：

核心消息队列的控制块--`CORE_message_queue_Control`是核心消息队列管理的核心数据结构，它包含了一个核心消息队列的线程等待队列`Wait_queue`、核心消息队列的属性`Attributes`、最多可存放消息数量

maximum_pending_messages，当前队列中还没有被处理的消息数目number_of_pending_messages、一个消息的最大长度maximum_message_size，等待处理的消息链表Pending_messages，存放消息的内存位置message_buffers，通知函数notify_handler等。

```
typedef struct {
    Thread_queue_Control    Wait_queue;
    CORE_message_queue_Attributes    Attributes;
    uint32_t                maximum_pending_messages;
    uint32_t                number_of_pending_messages;
    uint32_t                maximum_message_size;
    Chain_Control            Pending_messages;
    CORE_message_queue_Buffer    *message_buffers;
    CORE_message_queue_Notify_Handler    notify_handler;
    void                    *notify_argument;
    Chain_Control            Inactive_messages;
}    CORE_message_queue_Control;
```

消息缓冲区

```
typedef struct {
    uint32_t    size; //消息的大小
    uint32_t    buffer[1]; //消息数据的内存区
} CORE_message_queue_Buffer;
```

Pending_messages 消息缓冲区控制块

```
typedef struct {
    Chain_Node            Node; //目标对象
    int                    priority; //消息的优先级
    CORE_message_queue_Buffer    Contents; //消息缓冲区
}    CORE_message_queue_Buffer_control;
```

关键实现函数

核心层对核心消息队列的操作包括：

- | | |
|--|-----------------|
| ❧ _CORE_message_queue_Initialize: | 核心消息队列的初始化 |
| ❧ _CORE_message_queue_Submit: | 向核心消息队列请求发送消息 |
| ❧ _CORE_message_queue_Insert_message: | 向核心消息队列中插入消息 |
| ❧ _CORE_message_queue_Seize: | 从核心消息队列中获取消息 |
| ❧ _CORE_message_queue_Broadcast: | 核心消息队列广播消息 |
| ❧ _CORE_message_queue_Flush: | 清空核心消息队列 |
| ❧ _CORE_message_queue_Flush_waiting_threads: | 清空核心消息队列的线程等待队列 |
| ❧ _CORE_message_queue_Close: | 关闭核心消息队列 |

初始化核心消息口列

函数_CORE_message_queue_Initialize用于在创建一个核心消息队列时，初始化它的控制块结构。

函数调用信息

```
boolean _CORE_message_queue_Initialize(
```

```
CORE_message_queue_Control  *the_message_queue, //初始化的核心消息队列控制块
CORE_message_queue_Attributes *the_message_queue_attributes, //核心消息队列的属性
uint32_t                    maximum_pending_messages, //最多可以存放的消息数
uint32_t                    maximum_message_size //每个消息的最大长度
)
```

函数功能描述

具体实现如下：

- 1. 根据传入的参数初始化核心消息队列控制块的相应成员，并把number_of_pending_messages设为0，通知函数及其参数设为空；
- 2. 计算这个核心消息队列需要的内存大小，并调用_Workspace_Allocate为其分配内存，让核心消息队列的message_buffers成员指向分配的内存。其中所需内存大小的计算方法如下：

```
message_buffering_required = maximum_pending_messages *
                             (allocated_message_size
                              +
                              sizeof(CORE_message_queue_Buffer_control));
```

从上面的代码可以看出，消息体的大小加上消息缓冲控制块的大小，每个消息需要存放消息内容的缓冲区的空间和对应的缓冲区控制块，然后再乘以最大消息个数，就是这个核心消息队列所需要的内存大小，其中 allocated_message_size是指定的每个消息的最大长度向上4字节取整得到的数值。

- 3. 调用 _Chain_Initialize函数来将上面分配的内存初始化成消息的链表：

```
_Chain_Initialize (
    &the_message_queue->Inactive_messages, // Inactive_messages 指向这个链表
    the_message_queue->message_buffers, // 起始的内存位置
    maximum_pending_messages,           // 节点的个数
    allocated_message_size + sizeof( CORE_message_queue_Buffer_control ) //节点大小
);
```

- 4. 调用 _Chain_Initialize_empty函数把Pending_messages初始化为空链；
- 5. 调用 _Thread_queue_Initialize初始化核心消息队列的线程等待队列：

```
_Thread_queue_Initialize(
    &the_message_queue->Wait_queue,
    _CORE_message_queue_Is_priority( the_message_queue_attributes ) ?
        THREAD_QUEUE_DISCIPLINE_PRIORITY : THREAD_QUEUE_DISCIPLINE_FIFO,
    STATES_WAITING_FOR_MESSAGE,
    CORE_MESSAGE_QUEUE_STATUS_TIMEOUT
);
```

可以看出，初始化完成后的核心消息队列中有两个消息的链表，一个是等待处理的消息链表 Pending_messages，它初始化为空；还有一个空闲消息的链表，此链表的每个节点的结构如下：

| |
|----------|
| Node |
| priority |
| size |

| |
|-----------|
| buffer[0] |
| ... |
| buffer[N] |

向核心消息队列提交消息

函数 `_CORE_message_queue_Submit` 用于向核心消息队列发送消息，如果核心消息队列已满，并且可以等待，那么调用线程会被阻塞起来。

函数调用信息

```

CORE_message_queue_Status _CORE_message_queue_Submit(
    CORE_message_queue_Control *the_message_queue, // 要插入的核心消息队列
    void *buffer, // 插入的消息的内容
    size_t size, // 插入的消息的大小
    Objects_Id id, // 核心消息队列的 id
    CORE_message_queue_API_mp_support_callout api_message_queue_mp_support,
    CORE_message_queue_Submit_types submit_type, // 插入的消息的类型,
    boolean wait, // 是否可以等待
    Watchdog_Interval timeout // 等待的 tick 数
)

```

其中 `api_message_queue_mp_support` 是核心消息队列在多处理器上实现时调用的回调函数，`submit_type` 会决定在核心消息队列中插入的位置。

函数功能描述

具体实现如下：

1. 判断要入队的消息的大小是否超过核心消息队列允许的最大消息长度，如果是，则返回；
2. 如果核心消息队列的 `number_of_pending_messages` 为 0，也就是核心消息队列为空，没有 thread 在消息队列上等待，则调用 `_Thread_queue_Dequeue` 从等待的线程队列中取出一个线程，如果等待的队列为空。这个函数的返回值也为空，如果确实有线程等待，则按照等待策略选择一个线程出队，然后调用函数 `_CORE_message_queue_Copy_buffer` 把这个消息的内容拷贝到这个线程的 `Wait.return_argument` 处，这个值在线程进入等待队列时会被赋成线程保存要获取的消息的缓冲区地址，把 `Wait.return_argument_1` 设为消息的大小，把 `Wait.count` 设为提交的类型 `submit_type`；
3. 如果没有线程在等待消息，则判断核心消息队列是否已满，如果未满，则调用 `_CORE_message_queue_Allocate_message_buffer` 函数从 `Inactive_messages` 指向的空消息链表上取下一个节点，并对它的大小、优先级赋值，并调用函数 `_CORE_message_queue_Copy_buffer` 拷贝消息的内容，最后调用 `_CORE_message_queue_Insert_message` 函数把这个消息插入 `Pending_messages` 指向的链表；
4. 如果当前核心消息队列已满，则判断调用线程是否愿意等待，如果不能等待，则返回 `CORE_MESSAGE_QUEUE_STATUS_TOO_MANY`；
5. 如果可以等待，调用 `_ISR_Is_in_progress` 判断是否处于中断服务例程中，如果是则返回 `CORE_MESSAGE_QUEUE_STATUS_UNSATISFIED`，因为在 ISR 中不能被阻塞；
6. 如果不是处于中断服务例程中，把要入队消息保存在调用线程的等待结构体里，在核心消息队列有空间时，可以从这个结构体里取出信息，再次入队，具体的操作包括：屏蔽中断，调用 `_Thread_queue_Enter_critical_section` 函数使核心消息队列的线程等待队列进入线程入队准备状态，然后设置调用线程控制块的成员变量，即把调用线程的线程等待队列 `Wait.queue` 设为这个核心消息队列的线程等待队列，把等待的 `id`（即 `Wait.id`）设为输入参数的 `id` 值，把等待的返回参数 `Wait.return_argument` 设为传入的消息缓冲区地址，等待的选项 `Wait.option` 设为消息的大

小，把Wait.count设为消息的提交类型，然后使能中断，最后调用_Thread_queue_Enqueue把线程入队；

7. 返回CORE_MESSAGE_QUEUE_STATUS_UNSATISFIED_WAIT。

向核心消息队列插入消息

函数_core_message_queue_Insert_message根据提交的类型把消息插入核心消息队列。

函数调用信息

```
void _CORE_message_queue_Insert_message(
    CORE_message_queue_Control *the_message_queue, //插入的核心消息队列
    CORE_message_queue_Buffer_control *the_message, // 需要插入的消息
    CORE_message_queue_Submit_types submit_type // 提交的类型
)
```

函数功能描述

具体实现如下：

1. 把提交的类型submit_type赋给消息的优先级priority；
2. 根据submit_type值，采取不同的入队操作，如果是CORE_MESSAGE_QUEUE_SEND_REQUEST，则把消息插入到队列尾，如果是CORE_MESSAGE_QUEUE_URGENT_REQUEST，则把消息插入队列头，如果是其他值，则遍历消息链表，按优先级插入。这三种情况，都会把消息个数加1，并且如果原来的消息个数为0，则把通知标志变量notify设为真（这些操作，包括入队操作都是在屏蔽中断的情况下完成的）；
3. 如果notify为真并且notify_handler不为空，则调用如下函数：

```
(*the_message_queue->notify_handler)( the_message_queue->notify_argument );
```

这个函数主要是在消息数从0变为1的时候被调用，用来通知等待线程有消息到达。

从核心消息队列取消息

函数_core_message_queue_Seize用于从核心消息队列获取消息。

函数调用信息

```
void _CORE_message_queue_Seize(
    CORE_message_queue_Control *the_message_queue, //要插入的核心消息队列
    Objects_Id id, //核心消息队列的 id
    void *buffer, //保存获取消息的内容
    size_t *size, // 保存获取的消息的大小的地址
    boolean wait, //插入的线程是否可以等待
    Watchdog_Interval timeout //等待的 tick 数
)
```

函数功能描述

具体实现如下：

1. 判断队列中的消息个数是否为0，如果不为0，则把消息的个数减1，然后调用函数_core_message_queue_Get_pending_message从 Pending_messages链表中取出第一个消息，将该消息的大小存入size处，把消息的优先级赋给调用线程的Wait.count，然后把消息的内容拷贝到指定的缓冲区buffer中；然后
调用_Thread_queue_Dequeue从核心消息队列的等待队列中取出线程，如果等待队列为空，则把这个消息的空间交还给Inactive_messages链表，正常返回；

- 如果取出一个线程，表明有线程因为核心消息队列满无法提交消息而被阻塞，那么就把等待线程中的成员变量--等待结构里的关于消息的信息转移拷贝到核心消息队列中，正常返回；
2. 如果当前核心消息队列中没有消息，并且调用线程不愿意等待，则把线程的等待返回值设为 `CORE_MESSAGE_QUEUE_STATUS_UNSATISFIED_NOWAIT`，然后返回；
 3. 如果可以等待，则调用 `_Thread_queue_Enter_critical_section` 使核心消息队列的线程等待队列进入准备处理调用线程入队操作的状态，然后把调用线程的线程等待队列 `Wait.queue` 设为这个核心消息队列的等待队列，把等待的 `id` 设为输入参数的 `id` 值，把传入的消息缓冲区地址保存在等待的返回参数 `Wait.return_argument` 中，把保存消息大小的变量地址保存在 `Wait.return_argument_1` 中，这些值会在核心消息队列可以给这个线程提供消息时用到，然后使能中断，最后调用 `_Thread_queue_Enqueue` 把线程入队。

核心消息队列广播消息

函数 `_CORE_message_queue_Broadcast` 用于向核心消息队列的所有等待线程发送消息，使它们就绪。

函数调用信息

```
CORE_message_queue_Status _CORE_message_queue_Broadcast(
CORE_message_queue_Control *the_message_queue, //要插入的核心消息队列
void *buffer, // 保存获取消息的内容
size_t size, //广播消息的大小
Objects_Id id, //核心消息队列的 id
CORE_message_queue_API_mp_support_callout api_message_queue_mp_support,
uint32_t *count //用于保存收到消息的线程个数
)
```

函数功能描述

具体实现如下：

1. 判断队列中的消息个数是否为0，如果不是0，表明有消息，即线程等待队列中不可能有线程在等待消息，所以将 `count` 赋为0，返回；
2. 如果消息个数为0，则循环调用 `_Thread_queue_Dequeue` 从核心消息队列的线程等待队列中取出线程，直到线程等待队列为空为止，并把传入的消息拷贝给等待线程，更新 `count`，然后返回。

清空核心消息队列

函数 `_CORE_message_queue_Flush` 用于清空核心消息队列中的所有消息，它的入口参数就是要被清空的核心消息队列的控制块的指针。

函数调用信息

```
uint32_t _CORE_message_queue_Flush(
CORE_message_queue_Control *the_message_queue //需要清空的核心消息队列的控制块
)
```

函数功能描述

它先判断队列中的消息个数是否为0，如果是0，则直接返回，如果不是0，则调用 `_CORE_message_queue_Flush_support` 把消息清空，这个函数会把 `Pending_messages` 指向的消息链表挂接到 `Inactive_messages` 指向的链表上，然后把 `Pending_messages` 初始化为空，把队列中的消息个数设为0，返回原来核心消息队列中消息的个数。

清空核心消息队列的线程等待队列

函数 `_CORE_message_queue_Flush_waiting_threads` 用于清空核心消息队列中的线程等待队列。

函数调用信息

```
uint32_t _CORE_message_queue_Flush_waiting_threads(  
CORE_message_queue_Control *the_message_queue//清空等待队列的核心消息队列的控制块  
)
```

函数功能描述

该函数调用_Thread_queue_Flush来清空线程等待队列的。

```
_Thread_queue_Flush(  
    &the_message_queue->Wait_queue,  
    NULL,  
    CORE_MESSAGE_QUEUE_STATUS_UNSATISFIED_NOWAIT  
);
```

关□核心消息□列

函数_CORE_message_queue_Close用于关闭一个核心消息队列。

函数调用信息

```
void _CORE_message_queue_Close(  
    CORE_message_queue_Control *the_message_queue, //需要关闭的核心消息队列  
    Thread_queue_Flush_callout remote_extract_callout,  
    uint32_t status //传给线程的状态  
)
```

函数功能描述

具体实现如下：

1. 调用_Thread_queue_Flush清空核心消息队列中的线程等待队列；
2. 如果消息个数不为0，调用_CORE_message_queue_Flush_support清空核心消息队列；
3. 调用_Workspace_Free释放核心消息队列分配的内存。

系统服务层的 message queue manager 组件

消息队列的系统服务层在核心层的数据结构和函数的基础上进行了封装，提供了创建或者删除消息队列，向消息队列发送消息以及从消息队列获取消息等应用编程接口。

关键数据结构

在系统服务层的消息队列管理结构体是在 CORE_message_queue_Control 基础上又封装了一层而形成的，这便于利用对象管理机制对消息队列统一管理。

```
typedef struct {  
    Objects_Control      Object;  
    rtems_attribute      attribute_set;  
    CORE_message_queue_Control message_queue;  
} Message_queue_Control;
```

维护消息队列的全局变量是：

```
RTEMS_EXTERN Objects_Information _Message_queue_Information;
```

关键实现函数

RTEMS的CLASSIC API为用户提供了以下操作消息队列的API：

- ✎ `rtems_message_queue_create`：创建一个消息队列
- ✎ `rtems_message_queue_ident`：获得消息队列的ID
- ✎ `rtems_message_queue_delete`：删除一个消息队列
- ✎ `rtems_message_queue_send`：把消息放在消息队列的尾部
- ✎ `rtems_message_queue_urgent`：把消息放在消息队列的首部
- ✎ `rtems_message_queue_broadcast`：广播消息
- ✎ `rtems_message_queue_receive`：从队列中接收消息
- ✎ `rtems_message_queue_get_number_pending`：获取消息队列中消息的数量
- ✎ `rtems_message_queue_flush`：除去消息队列中所有的消息

下面对其中的部分关键函数进行分析。

□建消息□列

函数调用信息

```
rtems_status_code rtems_message_queue_create(  
  rtems_name      name,           //用户指定的队列名字  
  uint32_t        count,          //队列中可以存放的最大消息个数  
  uint32_t        max_message_size, //每个消息的最大长度  
  rtems_attribute attribute_set,   //消息队列的属性集  
  Objects_Id      *id             //指向消息队列的 id  
)
```

函数功能描述

函数`rtems_message_queue_create`用于创建一个消息队列。

具体实现如下：

1. 检查参数的合法性；
2. 屏蔽调度，调用`_Message_queue_Allocate`函数分配一个消息队列控制块，将控制块的属性设为输入参数`attribute_set`；
3. 根据 `attribute_set` 指定的属性，确定消息队列的排队策略，调用 `_CORE_message_queue_Initialize`为核心消息队列控制块分配空间并初始化控制块；
4. 调用`_Objects_Open`函数在`_Message_queue_Information`全局变量加入新建的消息队列对象，这样此消息队列就可以使用了；把队列的id保存在输出参数`id`中；
5. 使能调度，返回`RTEMS_SUCCESSFUL`。

□除消息□列

函数`rtems_message_queue_delete`用于删除一个消息队列。

函数□用消息

```
rtems_status_code rtems_message_queue_delete(  
  Objects_Id id //被删除的消息队列的 id  
)
```

函数功能描述

具体实现如下：

1. 调用_Message_queue_Get获得这个消息队列的控制块是否为本地对象；
2. 如果是本地的对象，调用_Objects_Close将_Message_queue_Information中队列对象信息指针置为NULL，这样此对象处于空闲态；
3. 调用_CORE_message_queue_Close关闭核心消息队列，它会释放核心消息队列的内存；
4. 最后调用_Message_queue_Free释放这个消息队列的控制块。

从消息队列上接收消息

函数rtems_message_queue_receive用于从消息队列上接收消息。

函数调用信息

```
rtems_status_code rtems_message_queue_receive(
  Objects_Id      id,      //被删除的消息队列的 id
  void            *buffer,  //存放消息的缓冲区
  size_t          *size,    //保存消息的长度
  uint32_t        option_set, //选项集，包括是否等待等信息
  rtems_interval  timeout   //可以等待的时间
)
```

函数功能描述

具体实现如下：

1. 检查参数的合法性；
2. 调用_Message_queue_Get获得这个消息队列的控制块及其位置；
3. 如果是本地的对象，判断是否可以等待，然后把相应的wait值作为参数调用_CORE_message_queue_Seize函数获取信息；
4. 使能调度，根据该线程的Wait.return_code返回相应的状态值；

其他函数

其他函数实现过程和rtems_message_queue_receive基本相同，只不过它的核心函数不同：

- ❏ 函数rtems_message_queue_send调用_CORE_message_queue_Send；
- ❏ 函数rtems_message_queue_broadcast调用_CORE_message_queue_Broadcast；
- ❏ 函数rtems_message_queue_flush调用_CORE_message_queue_Flush；
- ❏ 函数rtems_message_queue_get_number_pending从它的控制块的核心组件中获取number_of_pending_messages的值；
- ❏ 函数rtems_message_queue_urgent调用_CORE_message_queue_Urgent，而_CORE_message_queue_Urgent就是以类型CORE_MESSAGE_QUEUE_URGENT_REQUEST来调用函数_CORE_message_queue_Submit；

关于消息队列，除了上面的对消息队列操作的函数外，还有一个初始化函数_Message_queue_Manager_initialization用于初始化消息队列的管理器，它的作用是调用_Objects_Initialize_information来初始化全局变量_Message_queue_Information。这个函数会在系统启动时，初始化API的各种对象的管理器时调用。

应用举例

RTEMS源码中的\${RTEMS_ROOT}/testsuites/sptests/sp13中包含了消息队列的测试用例。它主要包括一个初始化任务和3个用来发送和接收消息的任务，由于这个测试用例的执行过程比较复杂，所以我们按照系统执行过程进行分析，并且由于篇幅关系，只分析这个用例的前面的一部分。

初始化任务：

1. 创建并启动三个任务，优先级都为4，并且支持抢占，不支持时间片轮转
2. 调用rtems_message_queue_create创建三个消息队列，队列1和3都采用FIFO策略，并且容量为100条消息，队列2采用优先级策略，并且容量为10
3. 删除初始化任务，使得其他三个任务可以运行

任务1首先运行:

- ☞ 调用函数 `rtems_message_queue_ident` 获取队列1的ID
- ☞ 两次调用 `rtems_message_queue_send` 向队列1发送消息1和消息2
- ☞ 调用 `rtems_task_wake_after` 等5秒

任务2开始运行:

- 连续3次调用 `rtems_message_queue_receive` 从队列1上获取消息
前两次会成功, 因为任务1已经向队列1发送了两条消息, 所以任务又可以获取2条消息, 但是第3次获取消息为什么会使任务2阻塞起来, 任务3开始执行
- 调用 `rtems_message_queue_receive` 从队列2获取消息, 如果没有消息任务可以永久等待
因为队列2上没有消息, 所以任务3也会被阻塞起来, 这时只有等待任务1的5秒到后继续运行:

- ☞ 调用 `rtems_message_queue_send` 向队列1发送消息3
- ☞ 调用 `rtems_task_wake_after` 等5秒

任务1再次向队列1发送消息后, 任务2会就绪, 在任务1开始等待后, 任务2开始运行:

- 调用 `rtems_task_set_priority` 提高自己的优先级
- 调用 `rtems_message_queue_receive` 从队列2上获取消息, 可以永久等待
这个操作会使任务阻塞起来直到等任务1的5秒等待时间到后, 任务1开始运行:

- ☞ 调用 `rtems_message_queue_send` 向队列2发送消息
此时任务2和任务3都在等待队列2上的消息, 并且任务3先于任务2等待, 但队列2是基于优先级策略的, 并且任务2的优先级高于任务3, 所以任务2会获得这个消息, 然后开始运行:

- 调用 `rtems_message_queue_send` 向队列2发送消息
- 然后调用 `rtems_message_queue_receive` 从队列1上接收消息, 等待时间为10秒
由于任务2向队列2发送了消息, 所以任务3会就绪, 但是此前任务1已经就绪, 所以任务1开始运行:

- ☞ 调用 `rtems_message_queue_receive` 在队列1上等待消息, 等待时间为10秒
此时开始运行任务3:

- 输出从队列2收到的消息
- 调用 `rtems_message_queue_broadcast` 向队列1上的线程广播

此时任务1和任务2都在队列1上等待消息, 它们收到消息后都会就绪, 但是任务2的优先级要高于任务1, 所以任务2开始运行

- 输出从队列1上获取的消息, 调用 `rtems_message_queue_receive` 从队列3上获取消息
虽然此时任务1和任务3都处于就绪状态, 但任务3原来就是就绪的, 所以任务3会继续运行:
- 输出广播消息唤醒的线程个数
- 调用 `rtems_message_queue_receive` 从队列3上获取消息

此时轮到任务1运行:

- ☞ 输出从队列1上获取的消息
- ☞ 调用 `rtems_task_delete` 删除任务2
- ☞ 调用 `rtems_message_queue_send` 向队列3发送消息
- ☞ 调用 `rtems_task_wake_after` 等待5秒

任务1向队列3发送的消息会使得任务3就绪, 所以接下来轮到任务3开始运行:

- 输出从队列3收到的消息
- 调用 `rtems_task_delete` 把自己删除

这部分程序运行结果如下:

```
*** TEST 13 ***
TA1 - rtems_message_queue_ident - qid => 1c010001
TA1 - rtems_message_queue_send - BUFFER 1 TO Q 1
TA1 - rtems_message_queue_send - BUFFER 2 TO Q 1
TA1 - rtems_task_wake_after - sleep 5 seconds
TA2 - rtems_message_queue_receive - receive from queue 1 - RTEMS_NO_WAIT
TA2 - buffer received: BUFFER 1 TO Q 1
TA2 - rtems_message_queue_receive - receive from queue 1 - RTEMS_WAIT FOREVER
TA2 - buffer received: BUFFER 2 TO Q 1
TA2 - rtems_message_queue_receive - receive from queue 1 - RTEMS_WAIT FOREVER
TA3 - rtems_message_queue_receive - receive from queue 2 - RTEMS_WAIT FOREVER
TA1 - rtems_message_queue_send - BUFFER 3 TO Q 1
```

```

TA1 - rtems_task_wake_after - sleep 5 seconds
TA2 - buffer received: BUFFER 3 TO Q 1
TA2 - rtems_task_set_priority - make self highest priority task
TA2 - rtems_message_queue_receive - receive from queue 2 - RTEMS_WAIT FOREVER
<pause>
TA1 - rtems_message_queue_send - BUFFER 1 TO Q 2
TA2 - buffer received: BUFFER 1 TO Q 2
TA2 - rtems_message_queue_send - BUFFER 2 TO Q 2
TA2 - rtems_message_queue_receive - receive from queue 1 - 10 second timeout
TA1 - rtems_message_queue_receive - receive from queue 1 - 10 second timeout
TA3 - buffer received: BUFFER 2 TO Q 2
TA3 - rtems_message_queue_broadcast - BUFFER 3 TO Q 1
TA2 - buffer received: BUFFER 3 TO Q 1
TA2 - rtems_message_queue_receive - receive from queue 3 - RTEMS_WAIT FOREVER
TA3 - number of tasks awakened = 02
TA3 - rtems_message_queue_receive - receive from queue 3 - RTEMS_WAIT FOREVER
TA1 - buffer received: BUFFER 3 TO Q 1
TA1 - rtems_task_delete - delete TA2
TA1 - rtems_message_queue_send - BUFFER 1 TO Q 3
TA1 - rtems_task_wake_after - sleep 5 seconds
TA3 - buffer received: BUFFER 1 TO Q 3
TA3 - rtems_task_delete - delete self

```

事件

事件主要用于任务间和任务与中断服务例程之间的同步和通信。由于用bit位表示一个事件，所事件相关的处理比消息队列等要快，但不能传递大的数据量，适合于对某些标记的通知等情况。另外，事件还有一个特点就是，某一个事件没有得到处理前，即使再接收到同一事件，其效果也只等同于发送一次。事件同其他的同步机制相比，更适用于一个任务要与多个任务或者中断服务的情况，任务可以等待一个事件集中的所有事件都发生，也可以等待事件集中的任何一个发生。

一个任务可以设置它关心的事件，其他任务或者中断服务例程可以触发该事件，并通知所有关心这个事件的任务。每个事件只是一个标志位，只有发生和不发生两种状态，不包含其他信息。每个事件之间是相互独立的，这样多个事件就可以构成一个事件集，事件集中的每一位表示一个事件。RTEMS中的事件集rtems_event_set 实际上就是一个32位的无符号整数，它的每一位都被定义成一个事件：

```

#define RTEMS_EVENT_0      0x00000001
#define RTEMS_EVENT_1      0x00000002
#define RTEMS_EVENT_2      0x00000004
....
#define RTEMS_EVENT_30     0x40000000
#define RTEMS_EVENT_31     0x80000000

```

系统服务层的 event manager 组件

在 RTEMS 的事件机制主要通过系统服务层上的 event manager 组件实现。

关键数据结构

事件管理器的状口

RTEMS为了管理事件，定义了如下的事件管理器的状态，全局变量_Event_Sync_state就是用来表示当前状态。

```

typedef enum {

```

```

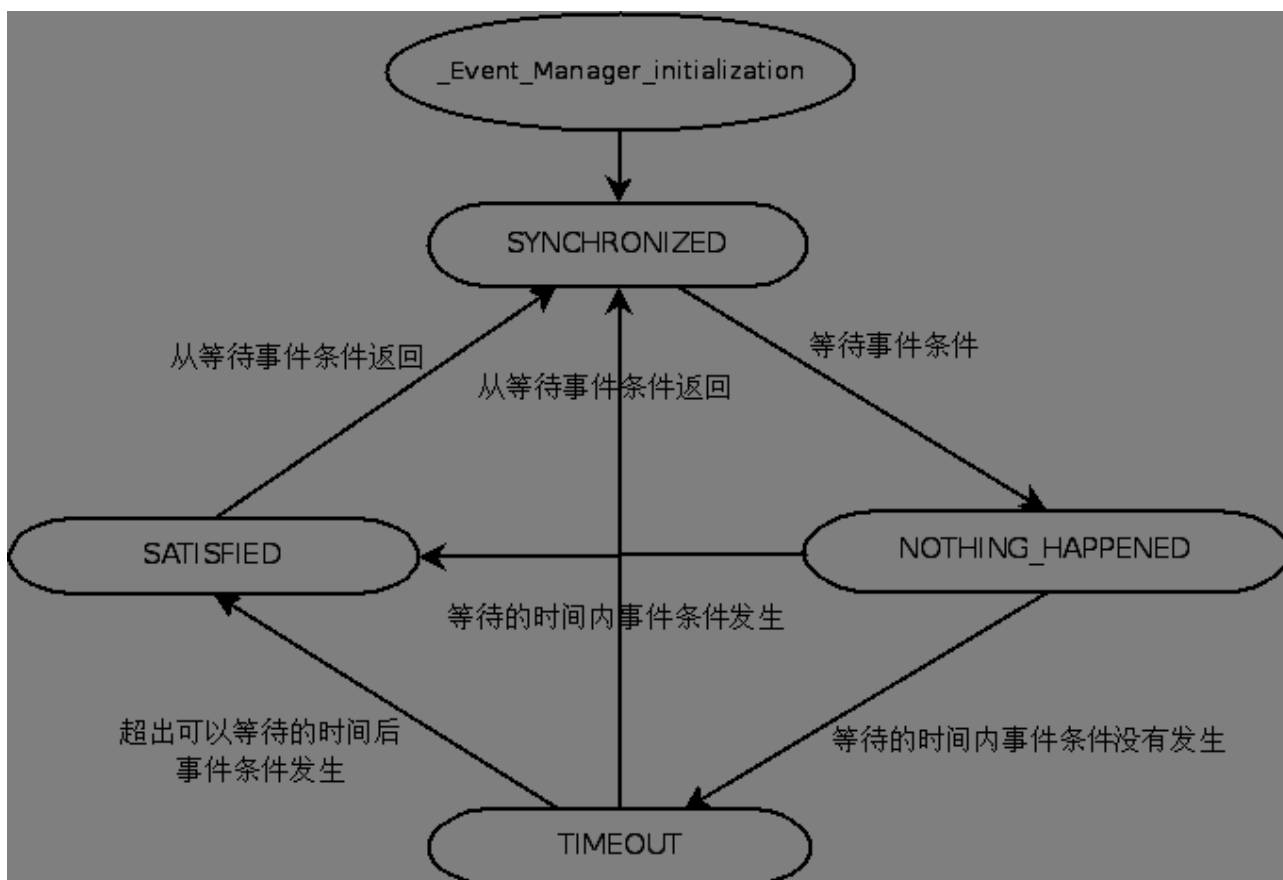
EVENT_SYNC_SYNCHRONIZED,
EVENT_SYNC_NOTHING_HAPPENED,
EVENT_SYNC_TIMEOUT,
EVENT_SYNC_SATISFIED
} Event_Sync_states;

```

各状态的含义：

1. EVENT_SYNC_SYNCHRONIZED：已同步状态，初始化时就是这个状态，在调用函数_Event_Seize时线程被阻塞后重新得到执行时也会变这个状态。
2. EVENT_SYNC_NOTHING_HAPPENED：没有事件发生并且正在等待，在调用函数_Event_Seize获取事件时，如果当前没有事件，并且可以等待，则把_Event_Sync_state设成这个状态
3. EVENT_SYNC_TIMEOUT：超时，如果在指定的时间内需要的事件没有发生，在超时处理函数_Event_Timeout会把事件状态设为EVENT_SYNC_TIMEOUT
4. EVENT_SYNC_SATISFIED：事件条件得到满足，在函数_Event_Surrender中，如果原来正在等待事件，则将事件状态设为EVENT_SYNC_SATISFIED

事件从创建到消亡的整个过程中的状态变化如下图所示：



事件状态转换图

关键实现函数

事件管理初始化

函数调用信息

```
void _Event_Manager_initialization( void )
```

函数功能描述

函数 `_Event_Manager_initialization` 的作用是初始化事件管理，其来龙去脉如下：

```
rtms_initialize_executive_early() --> RTEMS_API_Initialize() -  
-> _Event_Manager_initialization()
```

此函数只是把全局变量 `_Event_Sync_state` 设为 `EVENT_SYNC_SYNCHRONIZED`，这个变量用来表示事件的同步状态。

使线程接收事件

函数 `_Event_Surrender` 用于让线程接收事件，它的入口参数就是后备线程的控制块。

函数调用信息

```
void _Event_Surrender(  
    Thread_Control *the_thread    //接收事件的线程  
)
```

函数功能描述

它的具体实现如下：

1. 获得后备线程的 CLASSIC API 控制块：

```
api = the_thread->API_Extensions[ THREAD_API_RTEMS ];
```

`RTEMS_API_Control` 的定义如下：

```
typedef struct {  
    uint32_t                Notepads[ RTEMS_NUMBER_NOTEPADS ];  
    rtems_event_set         pending_events;  
    rtems_event_set         event_condition;  
    ASR_Information         Signal;  
} RTEMS_API_Control;
```

这个结构包括了记事本、等待处理的事件集、事件条件（期望接收的事件）、异步信号集，这些都是 RTEMS 任务正常运行所需要的内部数据。

2. 从线程的 `Wait.option` 获得等待的选项，即等待期望事件集合中的任何一个事件还是所有事件；
3. 从 API 控制块中取出等待处理的事件集合 `pending_events`，从线程的 `Wait.count` 中取出线程期望的事件 `event_condition`，再把 `pending_events` 和 `event_condition` 相与，得到捕获事件 `seized_events`；
4. 如果 `seized_events` 为空，直接返回，如果不为空，则判断这个线程是否处于 `STATES_WAITING_FOR_EVENT` 状态，如果已经满足事件条件，即捕获的事件 `seized_events` 和期望的事件 `event_condition` 相同，或者等待其中任何一个事件，那么把 `pending_events` 中的捕获的事件清除，把期待的事件 `Wait.count` 清 0，把捕获的事件集存入 `Wait.return_argument`；如果当前线程的定时器仍处于运行状态，即仍在等待事件，那么停止并移除定时器，然后调用 `_Thread_Unblock` 使线程就绪；正常返回；
5. 如果这个线程没有处于等待事件状态或者没有要等的事件，那么判断全局变量 `_Event_Sync_state` 的值，如果是 `EVENT_SYNC_SYNCHRONIZED` 或者 `EVENT_SYNC_SATISFIED` 直接返回；如果是 `EVENT_SYNC_NOTHING_HAPPENED` 或者 `EVENT_SYNC_TIMEOUT`，则进一步判断后备线程是否正在运行，如果不在运行，则返回；如果正在运行，并且已经满足事件条件，即捕获的事件 `seized_events` 和期望的事件 `event_condition` 相同，或者等待其中任何一个事件，那么把 `pending_events` 中的捕获的事件清除，把期待的事件 `Wait.count` 清 0，把捕获的事件集存入 `Wait.return_argument`，把 `_Event_Sync_state` 设为 `EVENT_SYNC_SATISFIED`；
6. 使能中断，然后返回。

□取期望事件

函数_Event_Seize用于让线程获取期望事件。

函数调用信息

```
void _Event_Seize(
    rtems_event_set event_in,      //期望获取的事件集
    rtems_option option_set,      //操作的选项集，比如是否要等待
    rtems_interval ticks,        //等待的 tick 数
    rtems_event_set *event_out    //指向捕获的事件集
)
```

函数功能描述

具体实现如下：

1. 将调用线程的等待返回值Wait.return_code设为RTEMS_SUCCESSFUL；
2. 获得后备线程的CLASSIC API控制块；
3. 从API控制块中取出等待处理的事件集pending_events，再把pending_events和期望的事件集event_in相与，得到捕获事件集seized_events；
4. 如果seized_events不为空，则判断是否满足事件条件，即捕获的事件seized_events和期望的事件event_in相同，或者等待其中任何一个，也就是说期望的事件已经发生了，那么把pending_events中的捕获的事件清除，把捕获的事件集存入event_out里，然后返回；
5. 如果seized_events为空，或者不满足事件条件，则根据选项判断是否可以等待，如果不可以，则把线程的等待返回值设为RTEMS_UNSATISFIED，并把seized_events保存在event_out里，然后返回；
6. 如果可以等待，则把_Event_Sync_state设为EVENT_SYNC_NOTHING_HAPPENED，然后把option_set存入当前执行线程的成员变量Wait.option，把event_in存入当前执行线程的成员变量Wait.count，把event_out保存在当前执行线程的成员变量Wait.return_argument（保存的这些变量会在事件发生时使用）；根据传入的tick值调初始化定时器，定时处理函数指定为_Event_Timeout，并调用_Watchdog_Insert_ticks函数把定时器插入到定时器队列中；
7. 调用_Thread_Set_state把线程设为STATES_WAITING_FOR_EVENT，这个操作会阻塞线程；
8. 接下来的代码是在线程重新得到运行时才会执行的，先恢复原来的事件状态，然后把_Event_Sync_state设为EVENT_SYNC_SYNCHRONIZED，并根据原来的状态做出相应的操作：如果是EVENT_SYNC_TIMEOUT，则把线程的等待返回值Wait.return_code设为RTEMS_TIMEOUT，然后使能中断，并调用_Thread_Unblock使线程解除阻塞；如果是EVENT_SYNC_SATISFIED，则判断watchdog是否还在运行，如果是则停止（调用_Watchdog_Deactivate），使能中断（调用_ISR_Enable）并删除（调用_Watchdog_Remove），然后调用_Thread_Unblock使线程解除阻塞，返回。

超□□理函数

函数_Event_Timeout是在指定的时间内等待的事件没有发生而运行的超时处理函数。

函数调用信息

```
void _Event_Timeout(
    Objects_Id id,                //等待的线程的 ID
    void *ignored
)
```

函数功能描述

具体实现如下：

1. 它首先调用_Thread_Get通过传入的参数id找到线程控制块及其位置
2. 如果这个线程是本地的，判断这个线程是否在等待事件，如果是则把线程等待的事件集count清零，如果_Event_Sync_state不是EVENT_SYNC_SYNCHRONIZED并且这个线程不在运行，相关代码如下：

```

if ( _Event_Sync_state != EVENT_SYNC_SYNCHRONIZED &&
    _Thread_Is_executing( the_thread ) ) {
    if ( _Event_Sync_state != EVENT_SYNC_SATISFIED ) {
        _Event_Sync_state = EVENT_SYNC_TIMEOUT;
    }
    _ISR_Enable( level );
}

```

3. 如果不符合上面的条件，则把线程的返回码return_code设为RTEMS_TIMEOUT，最后调用_Thread_Unblock使线程就绪，参与调度。

口送事件

函数口用信息

```

rtems_status_code rtems_event_send(
    Objects_Id id,          //目标任务的 id
    rtems_event_set event_in //发送的事件集
)

```

函数功能描述

函数rtems_event_send用于向指定的任务发送事件

具体实现如下：

1. 它首先调用_Thread_Get通过传入的参数id找到线程控制块及其位置；
2. 如是这个线程是本地的，则获得它的CLASSIC API控制块，然后把传入的事件集和线程的pending_events相或，也就是把要发送的事件加入到pending_events，然后调用_Event_Surrender函数使后备线程接收事件，最后调用_Thread_Enable_dispatch使能调度，返回RTEMS_SUCCESSFUL。

接收事件

函数口用信息

```

rtems_status_code rtems_event_receive(
    rtems_event_set event_in, //期望发生的事件集
    rtems_option option_set, //选项
    rtems_interval ticks,    //等待的 tick 数，0表示永远等待
    rtems_event_set *event_out //指向捕获的事件集
)

```

函数功能描述

函数rtems_event_receive用于发送事件

具体实现如下：

1. 检查入口参数的合法性；
2. 获取它的CLASSIC API控制块；
3. 如果传入的event_in为空，则把pending_events保存在 event_out里，然后返回；
4. 如果事件条件不为空，则禁止调度，调用_Event_Seize来获取事件，然后使能调度，把这个线程的等待返回值Wait.return_code作为返回值返回。

应用举例

RTEMS 源码中的\${RTEMS_ROOT}/testsuites/sptests/sp26中包含了事件的测试用例。它包括初始任务和子任务1、2三部分，初始化任务就是创建并启动两个子任务。

子任务1的工作包括

1. 调用 `rtems_task_wake_after` 等待3秒后，执行 `rtems_event_send (taskId2, 1)`，向子任务2发送一个事件
2. 调用 `rtems_task_wake_after` 等待3秒后，输 “Event sent”，然后调用 `rtems_task_suspend` 将自己阻塞
3. 醒来后输出 “Back to task 1”，然后再调用 `rtems_task_wake_after` 等待3秒
4. 调用 `rtems_task_suspend` 将自己阻塞

子任务2的工作包括

1. 调用 `rtems_task_wake_after` 等待1秒后，调用 `rtems_event_receive` 接收任务1发送的事件1
2. 收到事件后，判断并输出任务1是否处于阻塞状态
3. 等待4秒后，判断并输出任务1是否处于阻塞状态
4. 调用 `rtems_task_resume` 使任务恢复，判断并输出任务1是否处于阻塞状态
5. 等待4秒后，判断并输出任务1是否处于阻塞状态

这个测试用例中的事件主要用于同步，它的运行输出结果如下：

```
*** TEST 26 ***
subTask2 - Task 1 suspended? - should be 0: 0
subTask1 - Event sent
subTask2 - Task 1 suspended? - should be 1: 1
subTask2 - Task 1 suspended? - should be 0: 0
subTask1 - Back to task 1
subTask2 - Task 1 suspended? - should be 1: 1
*** END OF TEST 26 ***
```

异步信号

异步信号是当某个事件发生时产生的软件中断。当一个异步信号到达时，任务从正常的执行路径上断开，并调用相应的异步信号处理例程（Asynchronous Signal Routine，简称 ASR），然后在恢复以前被断开的执行。异步信号机制用于任务与任务之间、任务与 ISR 之间的异步操作，异步信号被任务（或 ISR）用于通知其他任务某个事件的出现等情况。

系统服务层的 signal manager 组件

在 RTEMS 的事件机制主要通过系统服务层上的 `event manager` 组件实现，它提供了发送事件和接收事件的应用编程接口。

关键数据结构

ASR_Information 异步信号管理信息，RTEMS_API_Control 控制块的 Signal 就是这样一个结构体，它保存了一个任务的异步信号的相关信息。

```
typedef struct {
    boolean      is_enabled;    /* are ASRs enabled currently? */
    rtems_asr_entry handler;    /* address of RTEMS_ASR */
    Modes_Control mode_set;     /* RTEMS_ASR mode */
    rtems_signal_set signals_posted; /* signal set */
    rtems_signal_set signals_pending; /* pending signal set */
    uint32_t     nest_level;    /* nest level of RTEMS_ASR */
} ASR_Information;
```

另外，在头文件 `asr.h` 中还定义了从 `RTEMS_SIGNAL_0` 到 `RTEMS_SIGNAL_31` 共32个异步信号。系统服务

层定义了异步信号集 `rtems_signal_set`，这就是一个32位整数，每一位表示一个信号。

```
typedef uint32_t rtems_signal_set;
```

关键实现函数

初始化 ASR

函数调用信息

```
RTEMS_INLINE_ROUTINE void _ASR_Initialize (  
    ASR_Information *information    //要初始化的 ASR 控制块  
)
```

函数功能描述

函数 `_ASR_Initialize` 用于初始化一个 ASR 的控制块结构 `ASR_Information` 的各个成员变量，部分代码如下所示：

```
information->is_enabled    = TRUE;  
information->handler        = NULL;  
information->mode_set       = RTEMS_DEFAULT_MODES;  
information->signals_posted = 0;  
information->signals_pending = 0;  
information->nest_level     = 0;
```

运行 ASR 口理函数

函数 `RTEMS_tasks_Post_switch_extension` 是任务切换的钩子（hook）扩展函数，它的主要作用就是调用异步信号的处理函数来处理异步信号。

函数口用信息

```
void _RTEMS_tasks_Post_switch_extension(  
    Thread_Control *executing  
)
```

函数功能描述

具体实现如下：

1. 根据输入参数 `executing` 表示的线程得到 RTEMS API 控制块 `api`；
2. 从线程的 RTEMS API 控制块中（`api->Signal`）获得异步信号的管理信息 `asr`；
3. 屏蔽中断，保存 `signals_posted` 的值到 `signal_set`，并把 `asr` 的 `signals_posted` 清0，使能中断；
4. 如果 `signal_set` 为空，表明没有异步信号需要处理，返回；
5. 如果 `signal_set` 不为空，表明有信号需要处理，则把 `asr->nest_level` 加1，调用 `rtems_task_mode` 函数把任务模式设为 `asr->mode_set`，然后调用

```
(*asr->handler)( signal_set );
```

此函数是异步信号服务例程；

6. 处理完异步信号后再把 `asr->nest_level` 减1，调用函数 `rtems_task_mode` 把任务的工作模式恢复到原来的模式。

函数 `RTEMS_tasks_Post_switch_extension` 是结构体 `RTEMS_tasks_API_extensions` 的成员：

```
API_extensions_Control _RTEMS_tasks_API_extensions = {  
    { NULL, NULL },  
    NULL, /* predriver */  
    _RTEMS_tasks_Initialize_user_tasks, /* postdriver */  
    _RTEMS_tasks_Post_switch_extension /* post switch */  
};
```

而在系统初始化时，函数 `_RTEMS_tasks_Manager_initialization` 会调用函数 `_API_extensions_Add` 把 `_RTEMS_tasks_API_extensions` 添加到系统的扩展的链表 `_API_extensions_List` 上。而函数 `_API_extensions_Run_postswitch` 会执行 `_API_extensions_List` 上的所有扩展函数。系统中有两处调用函数 `_API_extensions_Run_postswitch`，一处是在 `_Thread_Dispatch` 函数的最后，也就是一个任务获得执行机会后，在开始执行它原来被切换出去时的代码之前，先要处理它的异步信号；另外一个是在 `_ThreadProcessSignalsFromIrq`，这个函数会在中断返回时检查是否有信号发生，如果有则调用这个函数来处理。

□送一个异步信号

函数调用信息

```
rtems_status_code rtems_signal_send(
    Objects_Id id,           //目标任务的 ID
    rtems_signal_set signal_set //发送的信号集
)
```

函数功能描述

函数 `rtems_signal_send` 用于向一个目标任务发送异步信号

具体实现如下：

1. 检查 `signal_set` 的合法性
2. 调用 `_Thread_Get` 获取目标任务的 ID 及其位置
3. 如果是本地任务，获得这个任务的异步信号控制块 `asr`，判断 `asr->handler` 是否为空，如果是，则使能调度，返回
4. 如果不为空，则判断异步信号是否使能，即 `asr->is_enabled` 是否为真，如果不为真，则调用 `_ASR_Post_signals` 把 `signals_pending` 加到 `asr->signals_pending`
5. 如果异步信号使能，则把 `signals_pending` 加到 `signals_posted` 中，并把这个任务的 `the_thread->do_post_task_switch_extension` 设为 `TRUE`（这个变量置为真，`_API_extensions_Run_postswitch` 才可以执行），然后判断当前是否处于中断处理函数并且目标任就是当前运行的任务，如果是则把 `_ISR_Signals_to_thread_executing` 设为，在中断返回时会检查这个变量的值，如果为真，则会调用 `_ThreadProcessSignalsFromIrq` 处理

注册异步信号□理函数

函数□用信息

```
rtems_status_code rtems_signal_catch(
    rtems_asr_entry asr_handler, //要注册的异步信号的处理函数
    rtems_mode mode_set         //任务模式
)
```

函数功能描述

函数 `rtems_signal_catch` 用于指定异步信号的□理函数。

具体□□如下：

1. □取当前任□的异步信号控制□ `asr`
2. □用 `_Thread_Disable_dispatch` 屏蔽□度
3. 如果指定的 `asr_handler` 不□空，□把 `asr->mode_set` □□ `mode_set`，把 `asr->handler` □□ `asr_handler`
4. 如果指定的 `asr_handler` □空，□□用 `_ASR_Initialize` 重新初始化 `asr`，也就是不再□理异□信号
5. 使能□度，返回

□用□例

RTEMS源码中的\${RTEMS_ROOT}/testsuites/sptests/sp26中包含了事件的测试用例。它包括四部分内容：

初始化任务：创建并启动两个任务，任务2的优先级要高于任务1

任务1：

1. 调用rtems_signal_catch设置异步信号的处理函数为Process_asr
2. 调用rtems_signal_send向它本身发送一个信号RTEMS_SIGNAL_16
3. 判断全局变量Task_2_preempted是否为真，如果是，则输出“TA1 - TA2 correctly preempted me”

任务2：

1. 将全局变量Task_2_preempted设为假
2. 调用rtems_task_suspend将自己阻塞
3. 在得到恢复运行后将Task_2_preempted 设为真，然后再把自己阻塞

ASR□理函数Process_asr：这个函数调用rtems_task_resume使任务2恢复运行。

这个例子中，虽然任务2的优先级更高，但它将自己阻塞，所以任务1得到运行，任务1向自己发送一个信号，这个信号的处理函数会恢复任务2的执行，而任务2的优先级又高于任务1，所以它能够得到执行，把全局变量Task_2_preempted设为真。这里的ASR处理函数并没有区分是哪一个异步信号，如果需要对不同的信号采取不同的行为，可以在ASR的处理函数中使用条件分支结构。

这个例子的运行结果如下：

```
*** TEST 17 ***
TA2 - Suspending self
TA1 - rtems_signal_catch: initializing signal catcher
TA1 - Sending signal to self
TA2 - signal_return preempted correctly
TA1 - TA2 correctly preempted me
TA1 - Got Back!!!
*** END OF TEST 17 ***
```

栅栏

barrier 也称为栅栏，障碍等。其作用是线程运行过程中，设置一个障碍，阻止线程的进一步执行，等满足一定条件（比如被阻止的线程达到一定数目）后，被阻拦的线程才能继续执行。这一功能是 RTEMS 4.8版本的新特性。在 RTEMS 4.8版本中，RTEMS 在系统服务层的 CLASSIC API 类和 POSIX API 类中均提供了组件和相应的服务接口，相关代码分别位于 rtems\cpukit\rtems\src 和 rtems\cpukit\posix\src 路径下。其实这两套接口均通过调用 rtems\cpukit\score\src 下的核心层的 barrier handler 组件来实现其功能。下面将从 RTEMS 系统服务层中的 barrier manager 组件和 RTEMS 核心层的 barrier handler 组件入手，分析 barrier 在 RTEMS 中是如何实现的。

核心层的 barrier handler 组件

关键数据结构

core barrier 控制块

```
typedef struct {
    Thread_queue_Control    Wait_queue; //等待 barrier 释放的线程等待队列
    CORE_barrier_Attributes Attributes; //barrier 的属性
    uint32_t                number_of_waiting_threads; //等待 barrier 的线程个数
} CORE_barrier_Control;
```

core barrier 控制块定义的结构体为 CORE_barrier_Control，是一个核心数据结构，其所包含的成员变量 Wait_queue 属于 Thread_queue_Control 结构体，描述了在 barrier 上等待的线程。而多少个线程等在此 core barrier 控制块上就可以“跨过”此 barrier 继续运行呢？以及 barrier 是自动跨过还是手动跨过？这些 barrier 属性的描述都在成员变量 Attributes，其结构体定义为：

```
typedef struct {
    CORE_barrier_Disciplines discipline; //barrier 的“跨过”属性（手动还是自动）
    uint32_t maximum_count; //满足“跨过” barrier 的等待线程数
} CORE_barrier_Attributes;
```

CORE_barrier_Disciplines 有两种：

```
    CORE_BARRIER_AUTOMATIC_RELEASE, //指定的等待线程个数满足后，等待线程就自动
    跨过此 barrier
    CORE_BARRIER_MANUAL_RELEASE //由应用指定等待线程“跨过”此 barrier
```

关键实现函数

RTEMS CLASSIC API 类和 POSIX API 类里的 barrier 的操作函数仅仅是外壳，它们直接调用 barrier handler 提供的服务函数。barrier handler 提供的服务函数都是在 corebarrier.c、corebarrierrelease.c、corebarrierwait.c、corebarrier.inl 四个文件中。主要包括：

`_CORE_barrier_Initialize`: core barrier 初始化

core barrier 初始化

函数调用信息：

```
void _CORE_barrier_Initialize(
    CORE_barrier_Control *the_barrier,
    CORE_barrier_Attributes *the_barrier_attributes
)
```

输入参数为 core barrier 控制块和 core barrier 的属性。

函数功能描述：

此函数先根据输入参数给 core barrier 控制块的属性赋值，把被 core barrier 阻塞的线程数为设置0，调用函数 Thread_queue_Initialize 初始化用于存放阻塞线程的队列 the_barrier->Wait_queue。这里需要注意的是，线程等待队列的等待方式被设置为先来先服务（THREAD_QUEUE_DISCIPLINE_FIFO），支持过时唤醒。

等待 core barrier

等待 core barrier 的实现函数是 `_CORE_barrier_Wait`。

函数调用信息：

```
void _CORE_barrier_Wait(
    CORE_barrier_Control *the_barrier,
    Objects_Id id,
    boolean wait,
    Watchdog_Interval timeout,
    CORE_barrier_API_mp_support_callout api_barrier_mp_support
)
```

输入参数依次为：指向 barrier 控制块的指针，用于等待此 barrier 的对象 ID，是否允许 wait（1为允许），允许等待的时间（0为永久等待），用于支持 MP（多处理器）操作的 callout 函数。

函数功能描述：

此函数的执行流程如下：

1. 屏蔽中断；

2. 等待在 core barrier 上的线程数加1;
 3. 判断目前 core barrier 上面等待的线程数, 如果此 core barrier 已经达到允许等待线程数的最大值 (即 the_barrier->Attributes.maximum_count), 则如果这个 core barrier 的“跨越”属性又是自动“跨越” (即 CORE_BARRIER_AUTOMATIC_RELEASE), 则发出设置当前执行线程的等待返回值为自动返回 (即 CORE_BARRIER_STATUS_AUTOMATICALLY_RELEASED); 使能中断, 调用 _CORE_barrier_Release 函数“唤醒”在此 core barrier 等待的线程, 然后返回;
 4. 否则, 让当前执行线程进入 wait 状态;
 5. 开启中断;
 6. 线程进入此 core barrier 的线程等待队列;
- 注意: 设定第3步和第4步的顺序, 这样可以避免达到 barrier 上允许最大值的最后一个线程刚等待又被释放, 而带来不必要的开销。

释放 core barrier

释放 core barrier 的实现函数是 _CORE_barrier_Release 函数。

函数调用信息:

```
uint32_t _CORE_barrier_Release(
    CORE_barrier_Control      *the_barrier,
    Objects_Id                id,
    CORE_barrier_API_mp_support_callout api_barrier_mp_support
)
```

输入参数依次为: 指向将要被释放的 core barrier 控制块指针, 对象 ID, 支持 MP 操作的 callout 函数。

函数功能描述:

此函数的执行步骤如下:

1. 将唤醒线程计数置为0;
2. 将线程等待队列中的线程离队 (即唤醒线程), 如果支持多处理器, 则调用此函数传递进来的 callout 函数, 直到线程等待队列为空, 然后记录唤醒的线程个数;
3. 返回被唤醒的线程个数。

删除 core barrier

函数调用信息:

```
#define _CORE_barrier_Flush( _the_barrier, _remote_extract_callout, _status) \
    _Thread_queue_Flush( \
        &((_the_barrier)->Wait_queue), \
        (_remote_extract_callout), \
        (_status) \
    )
```

函数功能描述:

此函数实际上是一个宏, 它直接调用了 _Thread_queue_Flush 函数来删除与此 barrier 相关的线程等待队列中的线程指针, 这样这些等待的线程就会有可能会占用 CPU。

系统服务层的 barrier manager 组件

关键数据结构

barrier 控制块

```
typedef struct {
    Objects_Control    Object;    //barrier 的对象 ID
    rtems_attribute    attribute_set; //barrier 的属性
    CORE_barrier_Control Barrier; //core barrier 控制块
} Barrier_Control;
```

其实 barrier 控制块是对 core barrier 控制块的一个扩展封装，具体的实现都在 core barrier handler 中完成。

管理 barrier 对象的信息表是一个全局变量，定义如下：

```
RTEMS_EXTERN Objects_Information _Barrier_Information
```

关键实现函数

RTEMS CLASSIC API 类中有关 barrier 的管理主要分为行为控制和属性控制两大类。前者主要管理 barrier 的行为；后者主要配置 barrier 的属性。主要的实现函数包括：

- ❧ `_Barrier_Manager_initialization`: barrier manager 初始化
- ❧ `rtems_barrier_create`: 创建 barrier
- ❧ `rtems_barrier_ident`: 获得 barrier 的对象 ID
- ❧ `rtems_barrier_delete`: 删除 barrier
- ❧ `rtems_barrier_wait`: 等待 barrier
- ❧ `rtems_barrier_release`: 释放 barrier

barrier manager 初始化

函数调用信息：

```
void _Barrier_Manager_initialization(
    uint32_t    maximum_barriers
)
```

参数 `maximum_barriers` 表示了 RTEMS 要分配的 barrier 的最大个数

函数功能描述：

这是系统服务层的 barrier manager 组件的初始化函数，此函数直接调用核心层中的 `_Objects_Initialize_information` 函数来新建一个 barrier manager 的对象，统一于 rtems 的 object，该函数的唯一输入为 `maximum_barriers`，定义了最多 barrier 的个数：

其余对 barrier 的操作如 `allocate`、`get`、`free`、`isnull` 和 Rtems 正常的 object 对象完全一致，提供了分配空间、获得对象指针，释放资源，判断是否存在等操作。

创建 barrier

函数调用信息：

```

rtems_status_code rtems_barrier_create(
    rtems_name      name,           //barrier 的 name
    rtems_attribute  attribute_set,  //barrier 的属性
    uint32_t         maximum_waiters, //最大的任务等待个数
    rtems_id         *id            //barrier 的 id, 输出参数
)

```

函数功能描述:

此函数的执行流程如下:

1. 首先要创建的 barrier 的属性需要满足一系列条件, 包括名字合法, id 为 NULL;
2. 然后根据输入参数初始化 barrier 的“跨越”方式属性;
3. 禁止线程调用;
4. 调用 `_Barrier_Allocate` 函数分配一个 barrier, 此函数直接调用 `_Objects_Allocate(&Barrier_Information)` 从空闲对象池中获取一个对象存放 barrier 信息;
5. 调用 `_CORE_barrier_Initialize` 函数初始化 core barrier 和相关的属性;
6. 通过执行 `_Objects_Open` 函数, 使得 barrier 非空闲;
7. 使能线程调用;
8. 返回 id 和函数返回码。

获得 barrier 的对象 ID

函数调用信息:

```

rtems_status_code rtems_barrier_ident(
    rtems_name      name, //barrier 的 name, 输入参数
    rtems_id        *id   //barrier 的 id, 输出参数
);

```

函数功能描述:

此函数根据输入参数 name, 直接调用 `_Objects_Name_to_id` 函数, 这样就可根据 `_Barrier_Information` 信息表中的记录, 找到此 barrier 的 ID。

删除 barrier

函数调用信息:

```

rtems_status_code rtems_barrier_delete(
    rtems_id  id
);

```

此函数的输入参数是 barrier 的 id

函数功能描述:

此函数的执行流程如下:

1. 调用 `_Barrier_Get` 函数 (会进一步调用 `_Objects_Get` 函数, 屏蔽线程调度), 根据 id 获得 barrier 控制块;
2. 执行 `_CORE_barrier_Flush` 函数, 把此 core barrier 相关的等待队列上的线程都释放掉;
3. 执行 `_Objects_Close` 函数和 `_Barrier_Free` 函数完成对 barrier 的彻底清除;
4. 使能线程调度;
5. 正常返回。

等待 barrier

函数调用信息:

```
rtems_status_code rtems_barrier_wait(  
    rtems_id id,           //barrier 的 id  
    rtems_interval timeout //等待超时时间  
);
```

函数功能描述:

1. 调用_Barrier_Get 函数（会进一步调用_Objects_Get 函数，屏蔽线程调度），根据 id 获得 barrier 控制块；
2. 调用_CORE_barrier_Wait 函数，完成具体的 barrier 等待工作；
3. 使能线程调度；
4. 正常返回。

注意：如果指定了 timeout（不为零），则在 timeout 时间后唤醒等待线程。

释放 barrier

函数调用信息:

```
rtems_status_code rtems_barrier_release(  
    rtems_id id,  
    uint32_t *unblocked  
);
```

函数功能描述:

此函数的执行流程如下:

1. 调用_Barrier_Get 函数（会进一步调用_Objects_Get 函数，屏蔽线程调度），根据 id 获得 barrier 控制块；
2. 调用_CORE_barrier_Release 函数，完成具体的 barrier 释放工作；
3. 使能线程调度；
4. 正常返回。

应用举例

RTEMS源码中的\${RTEMS_ROOT}/testsuites/sptests/sp33中包含了对barrier的测试用例。它主要包括两部分内容：初始化任务和等待线程运行的任务Waiter。

Waiter的主要任务包括:

1. 输出等待信息，调用rtems_barrier_wait无超时地等待Barrier
2. 判断SuccessfulCase为真还是DeletedCase为真，输出相应的信息
3. 删除自己

初始化任务包括:

1. 调用rtems_barrier_create创建Barrier，属性为RTEMS_DEFAULT_ATTRIBUTES，即采用手动方式“跨过”Barrier,最大等待任务数为0.
2. 调用rtems_barrier_ident获取ID，看是否和Barrier相同，用来测试这个函数是否可以正常工作
3. 调用rtems_barrier_wait等待Barrier，等待的时间为25秒，因为此时只有一个任务在工作，所以

Barrier肯定不会被释放，在等待25秒后超时返回

4. 调用rtems_barrier_release释放Barrier，如果释放的线程数不为0, 则出错
5. 将successfulCase设为TRUE，DeletedCase设为FALSE
6. 创建并启动指定个数任务，每个任务运行Waiter任务
7. 调用rtems_task_wake_after等待1秒钟，让创建的Waiter任务阻塞在Barrier上
8. 调用rtems_barrier_release释放Barrier，并判断释放的任务数是否和创建的Waiter任务数相等，如果不相等则出错
9. 调用rtems_task_wake_after等待1秒钟，让Waiter任务输出信息
10. 将successfulCase设为FALSE，DeletedCase设为TRUE，重复上面的6-9操作，只不过第8步改为调用rtems_barrier_delete删除Barrier, 这个操作也会使等待在Barrier上的任务就绪。

运行结果如下：

```
*** TEST 33 ***
Create Barrier
Check Barrier ident
Wait on Barrier w/timeout and TIMEOUT

*** Testing Regular Release of Barrier ***
Delay to let Waiters block
Waiter 0 waiting on barrier
Waiter 1 waiting on barrier
Waiter 2 waiting on barrier
Waiter 3 waiting on barrier
Releasing tasks
Delay to let Waiters print a message
Waiter 0 back from barrier
Waiter 1 back from barrier
Waiter 2 back from barrier
Waiter 3 back from barrier

*** Testing Deletion of Barrier ***
Delay to let Waiters block
Waiter 0 waiting on barrier
Waiter 1 waiting on barrier
Waiter 2 waiting on barrier
Waiter 3 waiting on barrier
Delete barrier
Delay to let Waiters print a message
Waiter 0 back from barrier
Waiter 1 back from barrier
Waiter 2 back from barrier
Waiter 3 back from barrier
*** END OF TEST SP33 ***
```

小节

第九章 内存管理

概述

内存管理机制是实时操作系统的重要组成部分。从实时性的角度出发，要求内存分配过程的时间要尽可能的确定和快速。RTEMS 针对不同领域应用的需求差异，对内存管理提供了比较完善的机制。同其它常见的嵌入式操作系统类似，RTEMS 不支持虚拟存储管理，不支持复杂的段页式的保护机制，而采用线性编址方式，即逻辑地址和物理地址一一对应的平面模式。这样虽然没有虚拟存储管理提供的不受限于物理内存大小的地址空间、地址保护等功能，但在内存分配的确定性和实时性等方面有保障。

RTEMS 同时支持静态内存分配和动态内存分配两种管理方式。静态内存分配是指在编译或链接时将应

用所需的内存空间分配好。采用这种分配方案的 RTEMS 内核和应用映像所占内存空间（代码段和数据段等）的大小一般在编译时就能够确定，中断向量表等其它区域所占用的内存空间大小是个定值。这样采用静态的内存分配机制，在编译时就可以确定 RTEMS 所需内存的大小。

而动态内存分配是指系统运行时根据应用需要动态地分配内存。是否支持动态内存分配主要基于三个方面的考虑：实时性、可靠性和成本要求。对于实时性和可靠性要求极高的系统(硬实时系统)，不允许延时或者分配失败，必须采用静态内存分配。然而仅仅采用静态内存分配，使整个系统失去了灵活性，必须在设计阶段就预先知道所需要的内存并对之作出分配。虽然动态内存分配会导致响应和执行时间不确定、内存碎片等问题，但是它的实现机制灵活，给程序实现带来极大的方便，有的应用环境中动态内存分配甚至是必不可少的。当前大多数的嵌入式实时系统是硬实时和软实时的综合体。系统中的一部分任务有严格的时限要求，而另一部分只是要求完成得越快越好。而在这样的系统中，就可以采用动态内存分配来满足部分对可靠性和实时性要求不高的任务。采用动态内存分配的最大好处就是给设计者很大的灵活性，可以方便地将原来运行于非嵌入式操作系统的程序移植到嵌入式系统中。在 RTEMS 系统中，RTEMS 应用的堆（heap）空间等可采取动态内存分配机制，在系统运行时可以根据需要自动调整所需内存的大小。

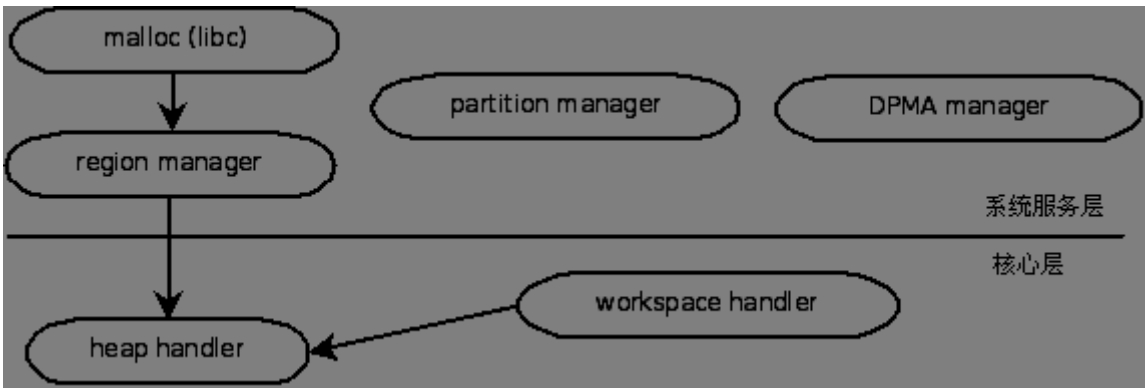
在 RTEMS 中，在 confdefs.h 中完成了大部分静态内存分配的工作。在动态内存分配方面，RTEMS 提供了两种形式的内存管理机制。动态内存管理应用于那些在执行过程中内存需求有变化的实时应用。这两种形式各有不同的特色：

- ❧ 定长分区内存管理：RTEMS 提供的从“定长内存区”（ partition，简称分区）动态内存分配“定长内存块”（ buffer，简称缓冲区）的操作。RTEMS 通过链表方式管理和维护各个缓冲区。下面简称分区管理。
- ❧ 变长区域内内存管理：RTEMS 提供的从“变长内存区”（ region，简称区域）动态内存分配“变长内存块”（ segment，简称段）的操作。RTEMS 通过双向链表方式管理和维护各个段。这种内存管理方式下面简称区域管理。

在地址转换方面， RTEMS 提供了一种双端口内存形式的内存管理机制。地址转换应用于那些需要与其它节点（CPU）或者一个智能的 I/O 处理器等共享内存的应用。它的特点如下：

- ❧ 双端口内存管理：对于多处理器， rtems 则使用“双端（双口）内存管理器”来控制不同处理器对同一片“双端口内存”的访问。“双端口内存”即是提供了可供两个处理器同时访问的内存芯片。这种内存管理的操作对象是“双端口内存区”（简称 DPMA），是一个连续的内存区。操作的机理，是建立内部地址与外部地址的映射，使 DPMA 的属主和非属主都可以访问 DPMA。

在 RTEMS 中，与内存管理相关的组件关系如下图所示：



内存管理相关组件关系图

下面将围绕 RTEMS 核心层的堆管理、系统服务层的分区管理、区域管理和双端口内存管理分别进行深入的分析。

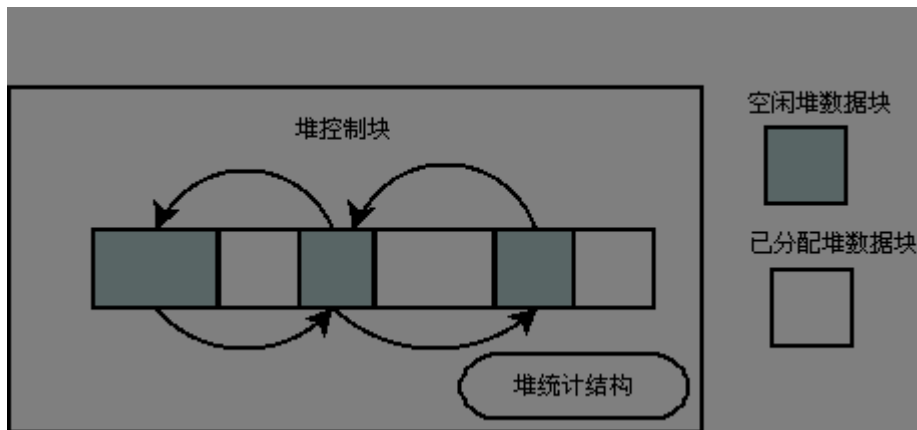
堆管理

RTEMS 的核心层 heap handler 组件是 RTEMS 动态内存管理的基础。heap handler 组件在核心层实现了 Heap_Control 结构体和相关的堆空间分配、堆空间释放操作。下面将就其关键数据结构和函数实现进

行分析。

关键数据结构

在与堆管理相关的关键数据结构包括：堆控制块、堆数据块和堆统计结构，它们之间的关系如下图所示：



堆控制块

```
typedef struct {
    Heap_Block free_list;    //空闲的堆数据块环形链表的头和尾
    uint32_t page_size;     //分配单位和对齐大小
    uint32_t min_block_size; //按 page_size 对齐的最小 block 大小
    void *begin;            //堆的起始地址
    void *end;              //堆的结束地址
    Heap_Block *start;       //堆中第一个合法的堆数据块地址
    Heap_Block *final;       //堆中最后一个合法的堆数据块地址
    Heap_Statistics stats;   //对堆使用的运行时统计
} Heap_Control;
```

该结构用来描述堆，包括堆中可用块的链表，最小块的大小，堆的开始地址，堆中的第一个堆数据块和最后一个堆数据块，堆运行时的统计状态。free_list 代表了空闲的堆数据块环形链表的头和尾，由于是环形链表，所以头尾是同一个链表节点。其中堆数据块这个结构就是把整个堆空间分成若干内存块，初始化的时候整个堆空间只有一个大的可用堆数据块（还有一个只是标志结束的 dummy block），以后每分配一块内存就产生一个堆数据块（block）。堆数据块是通过 Heap_Block 结构来描述的。

堆数据块

```
typedef struct Heap_Block_struct Heap_Block;
struct Heap_Block_struct {
    uint32_t prev_size;    //如果前一堆数据块是空闲的，则表示前一堆数据块的大小
    uint32_t size;        //表示了本块的大小和前一块的状态
    Heap_Block *next;      //指向下一空闲堆数据块的指针
    Heap_Block *prev;      //指向上一空闲堆数据块的指针
};
```

这个结构体定义了它前一个堆数据块 block 的大小—prev_size，而这个 block 自身的大小为 size。此 size 还有一个功能，因为 size 以字节为单位，而且保证四字节对齐，所以最低两位一定是0，这样 RTEMS 就可以利用最低位来表示前一个 block 的状态。这样前一个堆数据块 block 是否空闲 (free)，用 size 的最后一位来标志。next 和 prev 分别指向前一个和后一个 free 的 block，这两个域只有在这个 block 本身是 free 的时候才有效。

堆统计结构

```
typedef struct Heap_Statistics_tag {
    uint32_t instance;           //堆的实例个数
    uint32_t size;               //堆所占空间大小
    uint32_t free_size;          //当前堆管理的空闲空间的大小
    uint32_t min_free_size;      //最小空闲空间大小
    uint32_t free_blocks;        //当前的空闲堆数据块个数
    uint32_t max_free_blocks;    //最大空闲堆数据块的个数
    uint32_t used_blocks;        //当前已被使用的堆数据块的个数
    uint32_t max_search;         //搜索的最大堆数据块的个数
    uint32_t allocs;             //成功完成 alloc 函数调用的次数
    uint32_t searches;           //总的搜索次数
    uint32_t frees;              //成功完成 free 函数调用的次数
    uint32_t resizes;            //成功完成 resize 的次数
} Heap_Statistics;
```

这个结构描述了堆运行时的状态，其中的各个域的含义很直观。在从堆上分配空间的时候会改变它的域的信息。

关键实现函数

下面分析的关键函数涉及：

- _Heap_Initialize: 对堆管理本身的初始化
- Heap_Allocate: 分配堆所管理的数据空间
- _Heap_Extend: 扩展堆管理的数据空间
- _Heap_Resize_block: 改变堆管理的数据空间大小
- _Heap_Free: 释放堆管理的数据空间

初始化堆

堆的初始化是通过 _Heap_Initialize 函数实现的，在 RTEMS 初始化过程中完成了对堆的初始化。对堆管理的初始化的调用有两处。一处是在 RTEMS 初始化的时候，其调用顺序是：

```
bsp_start-> rtems_initialize_executive_early -> _Workspace_Handler_initialization ->
_Heap_Initialize
```

另外一处被 rtems_region_create 函数调用，用于创建区域。

函数调用信息：

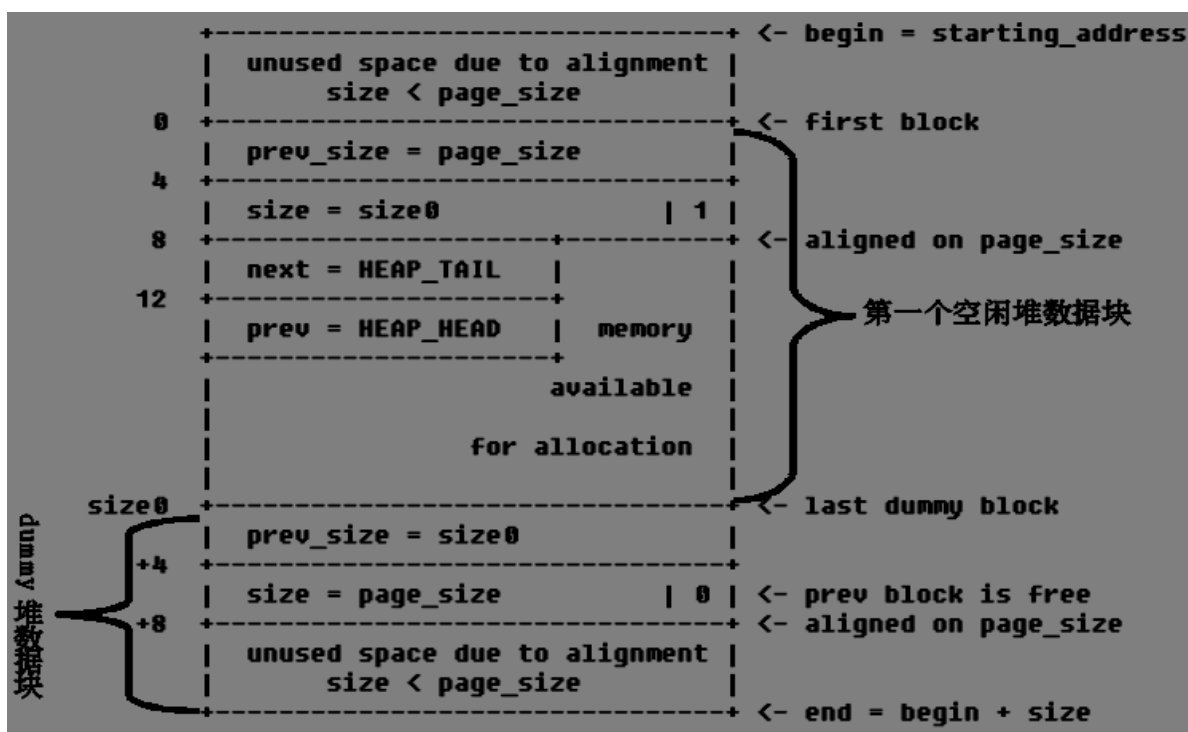
```
uint32_t _Heap_Initialize(
    Heap_Control *the_heap,           //堆控制块的头指针
    void *starting_address,          //堆数据块的起始地址
    uint32_t size,                   //堆的大小
    uint32_t page_size               //可分配的页大小
)
```

函数功能描述:

`_Heap_Initialize` 函数的工作是对空闲内存空间进行初始化, 建立一个大的空闲堆数据块。其大致流程如下:

1. 此函数首先从传入的 `starting_address` 开始 (不包括它自己) 找到第一个页对齐的地址, 然后把这个地址减去8个字节 (这8个字节用来存放第一个堆数据块的 `prev_size` 和 `size` 成员变量, 这也是管理堆数据块必须占用的空间) 作为第一个和唯一一个堆数据块的地址;
2. 再对这个堆数据块的成员变量初始化。其中这个堆数据块的 `size` 这个成员变量表明堆数据块的大小, 其值是通过此函数的参数 `size` (整个堆占用的内存空间) 减去用来页对齐而未使用的空间和 dummy block (dummy block 是用来管理和表示堆空间的结束的位置, 本身没有占用用于分配的内存空间) 的空间来得到。 `size` 成员变量的最低位置1, 表明它的前一个 block 不可用 (已分配了), 它的 `prev` 和 `next` 指向堆的可用块的头和尾, 因为现在只有一个空闲的堆数据块, 所以也就是它自己;
3. 然后, 在第一个 block 结束的位置存入一个 dummy block 作为堆空间的最后一个堆数据块 (`final_block`), dummy block 的 `size` 成员变量的最低位为0, 表明它的前一个 block 可用 ;
4. 最后对堆控制块 (`the_heap`) 的用于统计的成员变量 (`the_heap->states`) 进行初始化。

初始化完成以后堆的内存空间如下图所示:



从上图可以看出, 堆初始化以后只有一个可用的大的堆数据块, 从这个堆数据块的地址+8, 到 `size0` 的位置都是可以分配的空间。

分配堆

`_Heap_Allocate` 函数用于分配堆, 此函数在两个地方被调用。当被 `_Workspace_Allocate` 函数调用时, 用于对 RTEMS 的工作区 (workspace) 分配堆, 其针对的堆控制块是 `_Workspace_Area`。当被 `_Region_Allocate_segment` 调用时, 用于对一个存在的分区分配空间其针对的堆控制块是此分区的成员变量 `Memory`。RTEMS 通过 `_Heap_Allocate` 函数和 `_Heap_Allocate_aligned` 函数完成对内存的分配。

函数调用信息:

```
void *_Heap_Allocate(
    Heap_Control    *the_heap,
    size_t          size
)
```

函数功能描述:

在 Heap_Allocate 函数中, 首先通过 Heap_Calc_block_size 计算需要分配的内存大小。这个操作中, 预留的冗余部分来自两方面, 先是堆栈数据块描述符, 其次是保证内存大小是 page 整数倍的对齐填充除了需要留出来给描述这个堆数据块的成员变量的内存, 还有要调整 align 以保证内存大小是 page 大小的整数倍。所以实际分配的内存会大于函数参数中的 size 变量 (调用者要求的内存大小), 因为在堆数据块里除了可以使用的内存, 还有相关堆数据块的成员变量。

然后对这个堆控制块中的各个空闲堆数据块进行遍历, 来寻找足够大的空闲块。如果堆数据块的 size 大于需要分配的块大小, 就对这个堆数据块调用 _Heap_Block_allocate 函数, 其函数调用信息如下:

```
uint32_t _Heap_Block_allocate(
    Heap_Control* the_heap,
    Heap_Block*   the_block,
    uint32_t      alloc_size
)
```

_Heap_Block_allocate 函数会判断当前的堆数据块在分配掉 block_size 大小后, 剩下的空间是否还能构成一个空闲的堆数据块。如果可以的话就把这个块剩下的空间构造成为一个新的堆数据块, 然后通过调用函数 Heap_Block_replace 把原来的 heap_control 中的空闲堆数据块替换成新的堆数据块; 如果不能构成堆数据块那么就把原来的堆数据块从空闲堆数据块链中直接删除, 并减少 Heap_Statistics 中的空闲堆数据块的统计。

采用 _Heap_Allocate_aligned() 从堆上分配空间与 Heap_Allocate() 的不同之处在于返回的分配空间的开始地址要按照调用时指定的 alignment 对齐。

可见, RTEMS 对于堆的动态分配采用的是首次适配 (first-fit) 的算法。首次适配算法的缺点是会使得内存的前端出现很多小的空闲分区, 如果分配的块大小比较大, 则每次查找时都会扫描这些小的空闲分区。

为了解决这个问题, RTEMS 设计了 min_block_size 这个变量, 一旦小于这个变量, 那些细小的块就被忽略, 也就是不再分割, 这样避免重复的无效果的遍历这些小空闲块。对于块队列的管理, 如果剩余块的大小大于等于 min_block_size, 则保留剩余块, 操作时直接把剩余块替换原来那个块在队列中的位置。而如果剩余块的大小小于 min_block_size, 则分配之后把原来那块删除即可。

下面是分配空间以后的堆的内存布局:


```

Heap_Control    *the_heap,           //堆控制块头指针
void            *starting_address,   //堆扩展的起始地址
size_t          size,                //堆扩展的大小
uint32_t        *amount_extended     //堆扩展后的大小
)

```

函数功能描述:

1. 此函数首先判断堆扩展的起始地址是否紧接着堆的老的结束地址;

(即不满足 `starting_address >= the_heap->begin && starting_address < the_heap->end` 或者 `starting_address != the_heap->end` 的条件)

2. 如果不是, 则出错返回;
3. 如果是, 修改堆控制块的设置, 在堆控制块中的最后堆数据块的成员变量 (`heap->final`) 的后面, 指定一个新堆数据块, 其长度为函数的参数 `size`, 修正堆控制块中的统计成员变量;
4. 释放堆控制块中的老的堆数据块 `final`, 这样可以把老的堆数据块和新的堆数据块合并在一起, 形成一个大的空闲数据块。

需要注意的是, 堆扩展的空间应该是预先保留好的空闲空间, 且紧接在当前堆的结束地址。对堆的扩展主要是针对堆控制块管理的最后一个堆数据块, 扩展这个堆数据块的大小。

改变堆大小

函数调用信息:

```

Heap_Resize_status _Heap_Resize_block(
Heap_Control *the_heap,           //堆控制块指针, 输入参数
void         *starting_address,   //内存块的起始地址, 输入参数
size_t       size,                //要求满足的新的内存块大小, 输入参数
uint32_t     *old_mem_size,        //老的内存块大小, 输出参数
uint32_t     *avail_mem_size      //可用获得的新内存块大小, 输出参数
)

```

函数功能描述:

1. 根据 `the_heap`、`starting_address` 函数参数, 调用 `_Heap_Start_of_block` 函数, 得到对应的堆数据块指针 `the_block`;
2. 根据当前堆数据块指针 `the_block` 得到与它相邻的下一个堆数据块指针 `next_block`;
3. 判断下一个堆数据块是否在堆控制块的地址管理范围判断和当前堆数据块是否在使用;
4. 如果二者有一个不是, 则错误返回;
5. 如果二者都是, 则开始进行堆的空间大小改变;
6. 如果要求的新内存大小大于老的内存块大小, 则进一步判断下一个堆数据块是否空闲;
7. 如果不空闲, 则错误返回; 否则, 进行内存块扩展。扩展的方法是调用 `_Heap_Block_allocate` 函数分配下一个内存块的一部分空间给当前堆数据块使用, 即把当前堆数据块与这部分空间进行合并, 形成新的大的数据块;
8. 减少堆控制块中的使用块个数的统计信息 (因为分配的数据块与当前数据块合并为一个了), 至此, 对要求的新内存大小大于老的内存块大小的情况处理完毕;

9. 如果要求的新内存大小小于老的内存块大小，则要释放当前堆数据块的部分空间，即减少当前堆数据块大小，如果下一个堆数据块是空闲的，则把多余的空间合并到下一个堆数据块中，表示为一个空闲的堆数据块；如果下一个堆数据块不是空闲的，则把多余的空间形成一个独立的空闲堆数据块，至此，对要求的新内存大小小于老的内存块大小的情况处理完毕；
10. 把堆控制块中的改变堆大小的次数的统计信息加1，正常返回。

释放堆

函数调用信息：

```
boolean _Heap_Free(
    Heap_Control    *the_heap,
    void            *starting_address
)
```

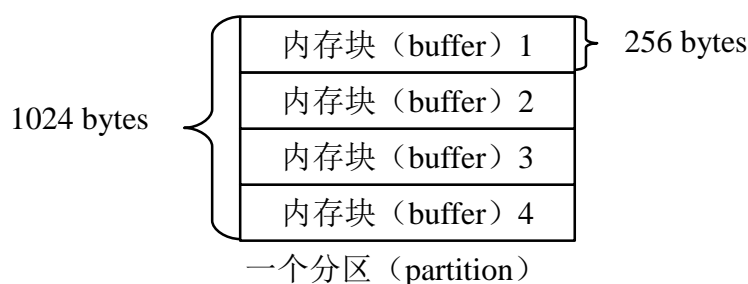
函数功能描述：

其具体处理流程是先判断 `starting_address` 是否合法（是否属于某个 block），如果合法，看看要释放的 block 是否可以与前一/后一块 free 的 block 合并，如果能够就合并，否则，改变相关 block 的属性，把它们链接起来。最后修正一下统计信息。

分区（Partition）管理

RTEMS 初始化后，系统创建的分区数目可在运行时动态增减。

一个分区就是把物理上邻接的内存区域划分成固定大小的缓冲区，RTEMS 通过对这些缓冲区动态的分配和回收来管理分区。各个分区是以缓冲区链表形式进行管理的。可以从空缓冲区链的头部获得缓冲区，回收时缓冲区返回到链的末尾。当缓冲区处于空闲状态时，RTEMS 为每个缓冲区分出两个指针变量作为空缓冲区链表。当缓冲区被分配了，那么整个缓冲区都可以被应用所使用。因此，我们只能修改一个在已分配的缓冲区之内的内存，因为改变一个在已分配的缓冲区之外的内存将会毁坏空闲缓冲区链表或者是一个临近的已分配的缓冲区的内容。例如一个包含4个内存块的1024字节的分区如下所示：



分区管理示例

关键数据结构

分区控制块是管理分区的核心数据结构，描述了管理分区的必要消息，分区属性确定了分区是否属于本地资源还是远程资源，以及等待分区分配的线程是基于 FIFO 排队还是基于优先级排队。

分区控制块

```
typedef struct {
    Objects_Control    Object;
```

```

void          *starting_address;    //物理地址
uint32_t      length;              //分区大小
uint32_t      buffer_size;         //每个缓冲区大小
rtems_attribute attribute_set;      //属性
uint32_t      number_of_used_blocks; //已使用的缓冲区个数
Chain_Control Memory;              //缓冲区链表
}

```

分区属性

```

typedef uint32_t rtems_attribute;
#define RTEMS_DEFAULT_ATTRIBUTES 0x00000000
#define RTEMS_LOCAL 0x00000000      /* 本地资源 */
#define RTEMS_GLOBAL 0x00000002     /* 全局资源 */
#define RTEMS_FIFO 0x00000000       /* 基于 FIFO 处理等待线程*/
#define RTEMS_PRIORITY 0x00000004   /* 基于优先级处理等待线程 */
.....//还有其他属性定义，内存管理部分没用到，不一一列举

```

关键实现函数

对分区的管理操作包括：

- `_Partition_Manager_initialization`: 初始化分区管理器
- `rtems_partition_create`: 创建一个分区
- `rtems_partition_ident`: 得到分区 ID
- `rtems_partition_delete`: 删除指定 ID 的分区
- `rtems_partition_get_buffer`: 从分区中获取一个 buffer
- `rtems_partition_return_buffer`: 释放之前获取的 buffer

初始化分区管理器

分区管理器初始化由函数 `_Partition_Manager_initialization` 完成，在 RTEMS 进行系统初始化时。

函数调用信息：

```

void _Partition_Manager_initialization(
    uint32_t maximum_partitions
)

```

此函数的来龙去脉如下所示：

```

boot_card->rtems_initialize_executive_early->_RTEMS_API_Initialize->_Partition_Manager_initialization

```

函数功能描述：

这个函数通过调用函数 `Objects_Initialize_information` 对分区管理进行初始化。这个工作是内核初始化工作中的一部分。需要注意的是如果应用不使用分区管理器，即不使用动态定长内存分配机制，则此函数为空。

创建分区

创建分区是由 `rtems_partition_create` 函数实现的。此函数创建一个用户命名的内存定长内存区。定长内存区的名字、起始地址、长度和内存大小全部在函数的参数中指定。RTEMS 在分区控制块链表中分配一个空闲的分区控制块供系统管理定长内存区。根据单个缓冲区的大小以及定长内存区大小可以计算出分

区的缓冲区（buffer）数目，创建后将返回定长内存区的 ID 标识。

函数调用信息：

```
rtems_status_code rtems_partition_create(  
    rtems_name      name,      //分区的名字，用户指定  
    void            *starting_address, //分区的开始地址，用户指定  
    uint32_t        length, //分区的总长度，用户指定  
    uint32_t        buffer_size, //定长内存块的大小，用户指定  
    rtems_attribute attribute_set, //分区的属性，用户指定  
    Objects_Id      *id //分区的 id, 返回值  
) Partition_Control;
```

函数功能描述：

1. 检查参数的合法性，如有错则返回错误信息，其中包括看各参数是否为0以及诸如 starting_address 是否对齐，分区大小是否对齐的操作是通过等；
2. 禁止线程切换；
3. 通过 _Partition_Allocate 函数分配分区控制块，按照输入参数给此控制块赋初始值；
4. 通过 _Chain_Initialize 对缓冲区链表进行初始化，值得注意的是，链表中结点的个数为 length / buffer_size；
5. 最后再允许任务切换，并成功返回。

关键的代码段如下所示：

```
{  
    register Partition_Control *the_partition;  
    .....  
  
    _Thread_Disable_dispatch();          /* prevents deletion */  
    .....  
  
    the_partition = _Partition_Allocate();  
  
    the_partition->starting_address      = starting_address;  
    the_partition->length                 = length;  
    the_partition->buffer_size           = buffer_size;  
    the_partition->attribute_set         = attribute_set;  
    the_partition->number_of_used_blocks = 0;  
  
    _Chain_Initialize( &the_partition->Memory, starting_address,  
                      length / buffer_size, buffer_size );  
    .....  
    _Thread_Enable_dispatch();  
    return RTEMS_SUCCESSFUL;  
}
```

需要注意的是： 给定长内存区指定地址时，需要指定一个连续的内存块。定长内存区默认被当作本地内存区处理，也可能是远端任务发出请求。 starting_address 和 buffer_size 参数一定要是4的整数倍。buffer 中特定的8 个字节作为链表指针。 buffer 的数目= 定长内存区大小/buffer 的大小，即 length/buffer_size 。

得到分区 ID

得到分区 ID 是通过 rtems_partition_ident 函数来完成的。当分区被创建的时候， RTEMS 产生一个独特的分区 ID 标识，该 ID 将伴随分区的整个生存周期。

函数调用信息:

```
rtems_status_code rtems_partition_ident(
    rtems_name      name,
    uint32_t        node,
    Objects_Id      *id
)
```

函数功能描述:

输入参数 name 是分区名字, 这样通过调用 _Objects_Name_to_id 函数就可以得到分区的 ID。需要注意的是: 分区的 ID 标识唯一, 名字则可能不唯一。查找时必须指定查找范围, 即 node。只有 rtems_partition_create 函数和 rtems_partition_ident 函数可以访问分区的 ID。

删除分区

rtems_partition_delete 函数用于删除分区。当分区被删除, 对应的分区控制块将会释放。需要注意的是, 一个已经分配了缓冲区 (buffer) 的分区是无法被删除的。所以如果分区中有正在使用的缓冲区, 那么不能释放该定长内存区。

函数调用信息:

```
rtems_status_code rtems_partition_delete(
    Objects_Id id
)
```

函数功能描述:

此函数先对 the_partition->number_of_used_blocks 是否等于0进行判断, 如果不等于0表示有缓冲区在使用, 所以返回。如果所有的缓冲区都没有被使用, 则继续通过调用

```
_Objects_Close( &Partition_Information, &the_partition->Object )
_Partition_Free( the_partition );
```

这两个函数的功能分别是删除这个分区的对象, 和释放分区控制块所占空间, 从而删除分区。

获取缓冲区 (buffer)

任务通过调用 rtems_partition_get_buffer 函数获得一缓冲区 (buffer)。如果有可用缓存, 那么函数立刻返回一个成功码。否则返回错误码, 任务不会因为内存获取失败而阻塞。

函数调用信息:

```
rtems_status_code rtems_partition_get_buffer(
    Objects_Id id,
    void        **buffer
)
```

函数功能描述:

此函数首先调用 _Partition_Get 函数, 从给定 ID 对应的分区中得到缓冲区。它调用 _Partition_Allocate_buffer 函数来分配 buffer, 而 _Partition_Allocate_buffer 函数直接调用了 _Chain_Get 函数来得到 buffer 链中未被使用的第一个缓冲区的地址。函数首先通过:

```
the_partition = _Partition_Get( id, &location );
```

从 id 得到分区控制块。然后使分区控制模块 the_partition 调用:

```
the_buffer = _Partition_Allocate_buffer( the_partition );
```

分配一个缓冲区。如果分配成功, 则把 the_partition->number_of_used_blocks 加一, 并把 the_partition 赋值给输出 buffer。

对于关键的 _Partition_Allocate_buffer, 通过调用 _Chain_Get 实现。而 _Chain_Get 的参数就是分

区控制块中的 Memory。有关_Chain_Get 函数的实现分析，可参看前面有关“对象管理机制”的章节。而 first 作为指向第一个缓冲区的指针，是已经作了初始化的。因此每次上述函数可以直接使用 the_chain->first。

需要注意的是，如果还有缓冲区空闲，则返回一个成功码，已使用缓冲区的个数加1。如果没有缓冲区空闲，则返回错误码，不会扩展分区大小。

释放缓冲区 (buffer)

rtems_partition_return_buffer 函数将缓冲区 (buffer) 释放回空闲链表。如果释放的缓冲区不属于指定的分区，函数返回错误码。

函数调用信息：

```
rtems_status_code rtems_partition_return_buffer(  
    Objects_Id id,  
    void *buffer  
)
```

函数功能描述：

此函数首先调用 _Partition_Get 函数，从给定 ID 对应的分区中得到缓冲区。然后通过 _Partition_Is_buffer_valid 函数判断指定缓冲区是否有效。如果有效则调用 _Partition_Free_buffer 释放 buffer，并将 number_of_used_blocks 减1。_Partition_Free_buffer 函数实际上是直接调用 _Chain_Append 函数来回收缓冲区的。有关_Chain_Append 函数的实现分析，可参看前面有关“对象管理机制”的章节。

□用□例

RTEMS源码中的\${RTEMS_ROOT}/testsuites/sptests/sp15中包含了分区的测试用例。

它主要包括两部分内容：

初始化任务：

1. 创建并启动任务1，任务1的优先级为4，而初始化任务的优先级为1，所以即使任务1就绪了，仍然是初始化任务继续执行。
2. 创建两个分区PT1，PT2，其中，Area_1和Area_2都是定义在system.h中的数组，大小分别为4096和274个字节。这两个分区上的缓冲区大小分别为512字节和128字节

```
puts("INIT - rtems_partition_create - partition 1");  
status = rtems_partition_create(  
    Partition_name[ 1 ],  
    Area_1,  
    4096,  
    512,  
    RTEMS_DEFAULT_ATTRIBUTES,  
    &Partition_id[ 1 ]  
);  
directive_failed( status, "rtems_partition_create of PT1" );  
  
puts("INIT - rtems_partition_create - partition 2");  
status = rtems_partition_create(  
    Partition_name[ 2 ],  
    Area_2,  
    274,  
    128,  
    RTEMS_DEFAULT_ATTRIBUTES,  
    &Partition_id[ 2 ]  
);  
directive_failed( status, "rtems_partition_create of PT2" );
```

3. 删除自己，以使得任务1得以运行。

任务1:

1. 调用函数 `rtems_partition_ident` 获得两个分区的ID
2. 连续两次调用 `rtems_partition_get_buffer` 从分区1上获取缓冲区，并输出这个缓冲区相对于分区起始地址的偏移
3. 连续两次调用 `rtems_partition_get_buffer` 从分区2上获取缓冲区，并输出这个缓冲区相对于分区起始地址的偏移
4. 再调用 `rtems_partition_return_buffer` 把分配的缓冲区分别返回给分区1、2
5. 调用 `rtems_partition_delete` 删除两个分区

```
*** TEST 15 ***
INIT - rtems_partition_create - partition 1
INIT - rtems_partition_create - partition 2
TA1 - rtems_partition_ident - partition 1 id = 2a010001
TA1 - rtems_partition_ident - partition 2 id = 2a010002
TA1 - rtems_partition_get_buffer - buffer 1 from partition 1 - 0x00000000
TA1 - rtems_partition_get_buffer - buffer 2 from partition 1 - 0x00000200
TA1 - rtems_partition_get_buffer - buffer 1 from partition 2 - 0x00000000
TA1 - rtems_partition_get_buffer - buffer 2 from partition 2 - 0x00000080
TA1 - rtems_partition_return_buffer - buffer 1 to partition 1 - 0x00000000
TA1 - rtems_partition_return_buffer - buffer 2 to partition 1 - 0x00000200
TA1 - rtems_partition_return_buffer - buffer 1 to partition 2 - 0x00000000
TA1 - rtems_partition_return_buffer - buffer 2 to partition 2 - 0x00000080
TA1 - rtems_partition_delete - delete partition 1
TA1 - rtems_partition_delete - delete partition 2
*** END OF TEST 15 ***
```

区域 (Region) 管理

区域管理机制比分区管理机制灵活。一个区域 (Region) 就是把一个物理上邻接的内存空间通过用户自定义的边界划分成大小不同的段 (Segment)，这些段可以动态的被分配和回收。每个段的大小是用户定义的页的大小的倍数。页的大小必须是4的倍数。例如，一个区域中的页是256字节，需要的段是350字节，那么就要分配一个512字节的段。

在 RTEMS 初始化时，系统创建的区域数目可在运行时动态增加。而且内存的控制结构 (区域控制块) 在回收已使用的段 (segment) 时，控制块会通过相邻数据块合并的方式来尽量维护大的空闲空间。这使得区域管理的系统开销降低到最小。而当应用程序发现一个区域的内存空间不够时，可以调用 `rtems_region_extend` 函数来扩展该区域的大小。当任务要求从某个区域获取空闲段而未成功时，可以立即返回，也可以采取多种等待策略。等待策略包括优先级等待、FIFO 等待。在 FIFO 等待策略中又可分为有限等待和无限等待。这样可以最大限度地满足不同实时应用的内存管理需求。

在另一方面，区域管理机制对空闲块采用不同大小内存块的双链表管理，并采用简单的首次适配 (“first-fit”) 算法。即从链表的头检查，如果匹配就分配。当一个段返回到区域时，这个空闲的块就跟它邻接的 (如果也是空闲的话) 块结合，成为一个更大的空闲块。由于各个块是可以自动结合的，所以一个区域的段数也是动态变化的。这种算法虽然保证了动态分配内存的速度，适合实时性的要求，但由于算法过于简单，容易导致系统中存在大量的内存碎片，降低内存使用效率和系统性能。

关键数据结构

区域控制块

```
typedef struct {
    Objects_Control      Object;
    Thread_queue_Control Wait_queue;           //等待得到内存空间的线程等待队列
```

```

void                *starting_address;    //开始地址
uint32_t            length;              //物理地址大小
uint32_t            page_size;           //页大小
uint32_t            maximum_segment_size; //最大段大小
rtems_attribute     attribute_set;       //属性
uint32_t            number_of_used_blocks; //已分配了的块数
Heap_Control        Memory;              //内存堆
} Region_Control;

```

区域控制块信息表

管理区域控制块的信息表是一个全局变量，其定义如下：

```
RTEMS_EXTERN Objects_Information _Region_Information;
```

关键实现函数

对区域的操作相关的关键实现函数包括：

- `_Region_Manager_initialization`：初始化区域管理器
- `rtems_region_create`：用给定起始地址、长度、名字等创建一个区域
- `rtems_region_extend`：扩展段长可变的区域
- `rtems_region_ident`：得到与给定区域名对应的 ID
- `rtems_region_get_information`：得到给定区域的信息
- `rtems_region_get_free_information`：得到区域中空闲块的信息
- `rtems_region_delete`：对于没有段在被分配的区域，实施删除操作
- `rtems_region_get_segment`：从给定区域中分配段
- `rtems_region_get_segment_size`：得到段长
- `rtems_region_return_segment`：将段释放回区域
- `rtems_region_resize_segment`：改变给定段的大小到给定值

其中 `rtems_region_ident` 函数与分区（partition）提供的类似函数在实现上基本没有区别，这里就不用进一步分析了。下面分析主要函数的功能和实现。

初始化区域管理器

区域管理器初始化由函数 `_Region_Manager_initialization` 完成，在 RTEMS 进行系统初始化时。
函数调用信息：

```

void _Region_Manager_initialization(
    uint32_t    maximum_regions
)

```

此函数的来龙去脉如下所示：

```

boot_card->rtems_initialize_executive_early->_RTEMS_API_Initialize->_Region_Manager_initialization

```

函数功能描述：

此函数首先初始化并建立 `_API_Mutex_Information` 链表，并分配一个属于 `API_Mutex` 结构体的全局变量 `_RTEMS_Allocator_Mutex`。然后再通过 `_Objects_Initialize_information` 初始化对区域的管理。

创建区域

`rtems_region_create` 函数创建用户命名的区域，是 RTEMS 系统服务层的区域管理器组件的重要函数。此函数可以被基于 RTEMS CLASSIC API 的应用所调用。同时它也是基于 newlib 的 `malloc` 函数的实现基础，`malloc` 函数利用了区域管理器提供的 `rtems_region_create`、`rtems_region_extend` 等函数完成上层应用的内存分配请求。

当实时任务调用 `rtems_region_create` 函数分配内存暂时得不到满足的时候，实时任务可以排队等待。排队规则可以是 FIFO 或者优先级高低次序。RTEMS 为区域分配区域控制块。此外 RTEMS 还创建一个区域的 ID 标识。由于在区域中的一些内存将会预留给 RTEMS 内部使用，所以区域中实际可用的内存空间会小于分配值。**而且 RTEMS 实际开销不是固定值, 例如，包含两个信号量的内存区比只有一个信号量的内存区可用空间要小???**。此外，内存区中的内存会自动合并，区域中内存块的数目也会变化，RTEMS 预留的内存大小也会动态变化。

函数调用信息：

| | |
|--------------------------------|----------------------------------|
| <code>rtems_status_code</code> | <code>rtems_region_create</code> |
| <code>rtems_name</code> | <code>name</code> , |
| <code>void</code> | <code>*starting_address</code> , |
| <code>uint32_t</code> | <code>length</code> , |
| <code>uint32_t</code> | <code>page_size</code> , |
| <code>rtems_attribute</code> | <code>attribute_set</code> , |
| <code>Objects_Id</code> | <code>*id</code> |

函数功能描述：

本函数在连续的内存区域创建区域 `region`，从中可以分配变长的内存段，其大致流程如下所示：

1. 函数首先检查输入 参数的合法性，包括起始地址是否对齐等等；
2. 然后通过 `_Region_Allocate` 函数分配区域控制块，`_Region_Allocate` 函数实际上直接调用了 `_Objects_Allocate` 函数，在 `_Region_Information` 对象链表中分配了一个 `Region object`；
3. 使用 `Heap_Initialize` 方式分配堆，其返回值代表这个区域可分配的最大空闲空间大小；
4. 配置好区域属性，包括 `starting_address`、`length`、`attribute_set`、`number_of_used_blocks`；
5. 随后初始化线程等待队列，并根据输入参数 `attribute_set` 决定按等待线程是安优先级排队还是按 FIFO 排队。
6. 通过 `_Objects_Open` 使得此区域对象可用，且把此区域对象 ID 告诉函数入口参数 ID。

关键代码片断如下：

```
.....
the_region->maximum_segment_size =
    _Heap_Initialize(&the_region->Memory, starting_address, length, page_size);

if ( !the_region->maximum_segment_size ) {
    _Region_Free( the_region );
    _RTEMS_Unlock_allocator();
    return RTEMS_INVALID_SIZE;
}

the_region->starting_address    = starting_address;
the_region->length              = length;
the_region->page_size           = page_size;
the_region->attribute_set       = attribute_set;
the_region->number_of_used_blocks = 0;

_Thread_queue_Initialize(
    &the_region->Wait_queue,
```



```

_Attributes_Is_priority( attribute_set ) ?
    THREAD_QUEUE_DISCIPLINE_PRIORITY : THREAD_QUEUE_DISCIPLINE_FIFO,
STATES_WAITING_FOR_SEGMENT,
RTEMS_TIMEOUT
);
.....

```

删除区域

`rtems_region_delete` 函数删除区域。区域被删除时，区域控制块释放回空闲链表。如果区域中有内存块在使用，那么删除会失败。如果成功删除，在该内存区域上等待的任务将获得表示内存区被删除的返回码。

函数调用信息：

```

rtems_status_code rtems_region_delete(
    Objects_Id id
)

```

函数功能描述：

1. 首先获得 `RTEMS_Allocator_Mutex` 互斥体；
2. 在删除之前，如果定义了 `RTEMS_DEBUG` 宏，这要通过 `_Region_Debug_Walk`（最终调用 `_Heap_Walk` 实现）检查内存区的完整性和正确性。注意，这主要用于调试；
3. 然后判断 `the_region->number_of_used_blocks` 是否等于0，也就是没有分配任何段时，才可以执行删除。如果条件满足，则通过调用 `_Objects_Close` 函数和 `_Region_Free` 函数完成删除操作。最后释放互斥体。注意，这里的 `_Region_Free` 函数实际上是直接调用了 `_Objects_Free(&_Region_Information, &the_region->Object)` 来完成具体的删除工作。

关键代码如下：

```

if ( the_region->number_of_used_blocks == 0 ) {
    _Objects_Close( &_Region_Information, &the_region->Object );
    _Region_Free( the_region );
    RTEMS_Unlock_allocator();
    return RTEMS_SUCCESSFUL;
}

```

需要注意：`rtems_region_delete` 函数不提供多处理器支持。任何知道区域 ID 标识的本地任务都能删除内存区。

获得段（segment）

`rtems_region_get_segment` 函数从指定的区域（region）中取的满足大小的空闲内存块（segment）。如果区域中有足够内存可用，函数返回成功，否则：

- ❏ 调用任务将会永远地等候空闲内存块（缺省情况）
- ❏ 如果指定了 `RTEMS_NO_WAIT` 函数会返回错误码
- ❏ 任务返回前等待指定时间

在 region 中等待队列里等待的任务将会按照 FIFO 或者优先级方式排序。如果内存区被删除，等待任务将会得到错误码。

函数调用信息：

```

rtems_status_code rtems_region_get_segment(
    Objects_Id      id,
    uint32_t        size,
    rtems_option     option_set,
    rtems_interval   timeout,
    void            **segment
)

```

输入参数包括对象 ID，区域的大小，关于等待的参数，等待的时间。输出主要是分配的段指针。

函数功能描述：

这个函数是区域管理的核心。其执行流程如下所示：

1. 首先是一些必要的判断；
2. 判断后，然后调用 `_Region_Allocate_segment` 分配空闲段，`_Region_Allocate_segment` 函数会直接调用 `_Heap_Allocate` 函数分配段（有关 `_Heap_Allocate` 函数的分析，可参见本章的“堆管理分析”的小节）；
3. 如果分配成功，把 `number_of_used_blocks` 加1，成功返回；
4. 如果分配不成功，那么判断是否允许等待，如果不允许等待，则直接返回“不满足”；如果允许等待，则设置任务的等待消息，并把任务加入到该区域控制块的线程等待队列中。

关键代码如下：

```

    if ( the_segment ) {
        the_region->number_of_used_blocks += 1;
        _RTEMS_Unlock_allocator();
        *segment = the_segment;
        return RTEMS_SUCCESSFUL;
    }

    if ( _Options_Is_no_wait( option_set ) ) {
        _RTEMS_Unlock_allocator();
        return RTEMS_UNSATISFIED;
    }

    _Thread_Disable_dispatch();
    _RTEMS_Unlock_allocator();

    executing->Wait.queue      = &the_region->Wait_queue;
    executing->Wait.id         = id;
    executing->Wait.count      = size;
    executing->Wait.return_argument = segment;

    _Thread_queue_Enter_critical_section( &the_region->Wait_queue );
    _Thread_queue_Enqueue( &the_region->Wait_queue, timeout );
    _Thread_Enable_dispatch();

    .....

```

释放段（segment）

`rtems_region_return_segment` 函数将内存块释放回内存区中。释放的内存将会和邻接内存合并形成较大的空闲内存区。释放后，内存区的等待队列的队首任务的需求将会被处理，如果内存大小合适，那么将内存分配给该任务，并解除任务的阻塞；否则任务继续等待。

函数调用信息:

```
rtems_status_code rtems_region_return_segment(  
    Objects_Id id,  
    void *segment  
)
```

函数功能描述:

1. 此函数首先获得对区域操作的互斥体;
2. 然后通过 `_Region_Free_segment` 函数来完成释放段到相应的 `region`, 并且在需要合并的情况下合并空闲段。而 `_Region_Free_segment` 函数最终调用 `_Heap_Free` 函数来释放段所占空间 (有关 `_Heap_Free` 函数的分析, 可参见本章的“堆管理分析”的小节);
3. 把区域的已使用块的数目减1;
4. 最后调用 `_Region_Process_queue` 函数, 释放区域操作的互斥体, 如果有等待分配段的线程, 且能够满足此线程的分配要求, 则调用 `_Region_Allocate_segment` 函数完成分配, 并唤醒等待线程;
5. 正常返回。

扩展段 (segment)

`rtems_region_extend` 函数可以增加内存区域的大小。函数参数指定了新加入内存的大小以及起始地址。

函数调用信息:

```
rtems_status_code rtems_region_extend(  
    Objects_Id id,  
    void *starting_address,  
    uint32_t length  
)
```

函数功能描述:

此函数可以在已经存在的区域上添加内存, 用户可以指定添加内存的大小和起始位置。

操作通过 `_Heap_Extend` 实现。有关 `_Heap_Extend` 函数的分析, 可参见本章的“堆管理分析”的小节。

改变段 (segment) 的大小

搜索 RTEMS 源码, 发现 `rtems_region_resize_segment` 函数只是被 `realloc` 函数调用, 这个函数是为 `newlib` 服务的, 给基于 `newlib` 的应用提供动态调整分配内存大小的功能。我们可用首先看看 `realloc` 函数的功能, 这样可用更好地理解 `rtems_region_resize_segment` 函数的实现 (其实是 `_Heap_Resize_block` 函数的实现)。

`realloc` 函数可以对给定的指针所指的空间进行扩大或者缩小, 无论是扩张或是缩小, 原有内存的内容将保持不变。当然, 对于缩小, 则被缩小的那一部分的内容会丢失且返回指针为原来内存起始地址; 但若扩大, 则 `realloc()` 不一定会返回原先的指针。即 `realloc` 并不保证调整后的内存空间和原来的内存空间在同一起始地址。

函数调用信息:

```
rtems_status_code rtems_region_resize_segment(  
    Objects_Id id,  
    void *segment,  
    size_t size,  
    size_t *old_size
```

)

函数功能描述:

`rtems_region_resize_segment` 函数通过调用 `_Heap_Resize_block` 实现改变段的大小。有关 `_Heap_Resize_block` 函数的分析, 可参见本章的“堆管理分析”的小节。

其它函数

其他关于区域的操作函数, 都建立在 `Heap` 的基础上且相对简单, 这里就不再详细分析。

应用举例

RTEMS源码中的`{RTEMS_ROOT}/testsuites/sptests/sp16`中包含了区域的测试用例。

它主要包括一个初始化任务和5个任务, 由于这个测试用例的执行过程比较复杂, 所以我们按照系统执行过程进行分析, 并且由于篇幅关系, 只分析到任务3被删除。

首先运行初始化任务:

1. 创建并启动任务1、2、3, 它们的优先级都是140, 并且使用`RTEMS_DEFAULT_MODES`, 即可以抢占, 并且禁止时间片轮转。
2. 调用`rtems_region_create`创建4个区域`RN1`, `RN2`, `RN3`, `RN4`, 这4个分区的地址空间都是定义在`system.h`中的数组, 大小都是4096个字节, 页大小是128字节, 除了区域2, 其他区域的属性都是`RTEMS_DEFAULT_ATTRIBUTES`, 表明按照FIFO排队, 区域2上等待的任务按照优先级排队。
3. 删除自己, 以使得其他任务得以运行, 因为初始化任务的优先级最高

由于任务1最先就绪, 所以任务1最先运行:

- 调用函数`rtems_region_ident`获得区域1的ID
 - 调用`rtems_region_get_segment`从区域2上获得100个字节的缓冲区, 可以无限等待, 并输出返回的缓冲区相对地址, 注意这里实际分配的是128字节。
 - 调用`rtems_region_get_segment`从区域3上获得3K个字节的缓冲区, 可以无限等待, 并输出返回的缓冲区相对地址
 - 调用`rtems_region_get_segment`从区域1上获得3080个字节的缓冲区, 不等待, 并输出返回的缓冲区相对地址
 - 调用`rtems_task_wake_after`让出CPU, 把任务1移到它所在的任务链表的最后
- 这时任务2得到运行机会:

- 调用`rtems_region_get_segment`从区域1上获得2048个字节的缓冲区, 可以永久等待, 获取成功后会输出缓冲区的相对地址。

由于此时任务1已经在区域1上分配了3080字节的空间, 所以任务2只能等待, 这时任务3开始运行:

- 调用`rtems_region_get_segment`从区域2上获得3950个字节的缓冲区, 可以永久等待, 获取成功后会输出缓冲区的相对地址

由于此时任务1已经在区域2上分配了128字节, 所以任务3也只能等待, 这时又轮到任务1继续运行 (需要解释为何不够):

- 任务1调用`rtems_region_return_segment`将原来在区域1上分配的空间返回

- 然后调用`rtems_region_get_segment`在区域1上分配3K空间, 可以等待10秒

在任务1返回它的内存空间时, 系统就会把为任务2在区域1上分配内存, 然后使得任务2就绪, 但是由于任务2和任务1的优先级相同, 所以无法抢占, 任务1继续执行, 当它再次在区域1上请求空间时, 就会被阻塞起来, 轮到任务2运行:

- 任务2调用`rtems_region_return_segment`将原来在区域1上分配的空间返回, 这会使任务1获得内存空间并就绪

- 调用`rtems_task_set_priority`提高自己的优先级

- 调用`rtems_region_get_segment`从区域2上获取3950字节空间

由于此前任务3已经在区域2上等待空间, 所以任务2也会被阻塞起来, 任务1开始运行:

- 调用`rtems_region_return_segment`释放原来在区域2上获取的空间

此时任务2和任务3都在区域2上等待, 而且任务3排在任务2前面, 但是由于区域2指定的是优先级策略, 而且任务2已经提高了自己的优先级, 所以任务2会获得这个空间并就绪, 由于它的优先级高, 所以它抢占任务1开始运行:

- 调用 `rtems_region_return_segment` 释放在区域2上获取的空间
 - 调用 `rtems_task_delete` 将自己删除
- 在任务2释放空间时，任务3会在区域2上获得空间变为就绪，但是任务1原来就处于就绪状态，所以任务1继续运行：
- ☒ 调用 `rtems_task_wake_after` 让出CPU，把任务1移到它所在的任务链表的最后
- 这时任务3得到运行机会：
- 调用 `rtems_region_get_segment` 从区域3上获得2048个字节的缓冲区
- 因为此前任务1已经在区域3获取3K空间，所以任务3会被阻塞起来，任务1继续运行：
- ☒ 调用 `rtems_task_delete` 删除任务3
- 这部分程序的运行结果如下：

```

*** TEST 16 ***
TA1 - rtems_region_ident - rnid => 32010002
TA1 - rtems_region_get_segment - wait on 100 byte segment from region 2
TA1 - got segment from region 2 - 0x000000f8
TA1 - rtems_region_get_segment - wait on 3K segment from region 3
TA1 - got segment from region 3 - 0x0000003f8
TA1 - rtems_region_get_segment - get 3080 byte segment from region 1 - NO_WAIT
TA1 - got segment from region 1 - 0x000000378
TA1 - rtems_task_wake_after - yield processor
TA2 - rtems_region_get_segment - wait on 2K segment from region 1
TA3 - rtems_region_get_segment - wait on 3968 byte segment from region 2
<pause>
TA1 - rtems_region_return_segment - return segment to region 1 - 0x000000378
TA1 - rtems_region_get_segment - wait 10 seconds for 3K segment from region 1
TA2 - got segment from region 1 - 0x0000007f8
TA2 - rtems_region_return_segment - return segment to region 1 - 0x0000007f8
TA2 - rtems_task_set_priority - make self highest priority task
TA2 - rtems_region_get_segment - wait on 3968 byte segment
TA1 - got segment from region 1 - 0x0000003f8
TA1 - rtems_region_return_segment - return segment to region 2 - 0x000000f8
TA2 - got segment from region 2 - 0x000000008
TA2 - rtems_region_return_segment - return segment to region 2 - 0x000000008
TA2 - rtems_task_delete - delete self
TA1 - rtems_task_wake_after - yield processor
TA3 - got segment from region 2 - 0x000000008
TA3 - rtems_region_get_segment - wait on 2K segment from region 3
TA1 - rtems_task_delete - delete TA3
.....
*** END OF TEST 16 ***

```

双端口内存 (Dual Ported Memory) 管理

双端口内存区域 (DPMA) 是分布式嵌入式系统中的一个特定处理器拥有的 RAM 块，并且这个 RAM 块还可以被此系统中的其他处理器访问。拥有 RAM 块的处理器通过内部地址访问内存，而其他的处理器通过外部地址来访问。因此，RTEMS 定义了一个特殊的端口作为内部和外部地址映射。有两种系统配置会用到双端口内存管理：

- ☒ 一种是紧耦合的多处理器系统，双端口内存将在所有的节点之间被共享，并被用来做内部通信。
- ☒ 一种是有智能外设控制器的计算机系统，这些控制器将使用 DPMA 作为高性能的数据传送。

关键数据结构

双端口控制块

```

typedef struct {
    Objects_Control Object;

```

```

void          *internal_base; /* base internal address */
void          *external_base; /* base external address */
uint32_t      length;         /* length of dual-ported area */
} Dual_ported_memory_Control;

```

关键实现函数

对双端内存操作相关的关键实现函数包括：

- ✎ `rtems_port_create`: 创建一个端口
- ✎ `rtems_port_ident`: 获取端口的对象 ID
- ✎ `rtems_port_external_to_internal`: 外部地址转换成内部地址
- ✎ `rtems_port_internal_to_external`: 内部地址转换成外部地址
- ✎ `rtems_port_delete`: 删除一个端口

创建一个端口

函数调用信息：

```

rtems_status_code rtems_port_create(
    rtems_name      name,
    void            *internal_start,
    void            *external_start,
    uint32_t        length,
    Objects_Id      *id
)

```

函数功能描述：

通过 `rtems_port_create` 以用户自定义的名字来创建一个 DPMA 的端口。用户定义这个端口的内部外部关系。RTEMS 为每个新创建的 DPMA 端口，从空闲的 DPCB 链表中分配一个端口控制块 DPCB 来管理新建的端口。RTEMS 同时也分配一个唯一的 DPMA 对象 ID。

获取端口的对象 ID

函数调用信息：

```

rtems_status_code rtems_port_ident(
    rtems_name      name,
    Objects_Id      *id
)

```

函数功能描述：

当一个端口被创建的时候，RTEMS 就为每个端口分配一个唯一的对象 ID。端口对象 ID 在调用其他的端口函数时会被用到。这就可以通过 `rtems_port_ident` 函数来获取。此函数实际上是通过直接调用 `_Objects_Name_to_id` 函数来完成根据 name 查找到 id 的。

转换地址

函数调用信息：

```

rtems_status_code rtems_port_external_to_internal(
    Objects_Id id,          //端口 id, 输入参数
    void *external,        //外部地址, 输入参数
    void **internal         //内部地址, 输出参数
)

rtems_status_code rtems_port_internal_to_external(
    Objects_Id id,          //端口 id, 输入参数
    void *internal,         //内部地址, 输入参数
    void **external         //外部地址, 输出参数
)

```

函数功能描述:

`rtems_port_external_to_internal` 函数可以把一个特定端口的外部地址转换成内部地址;
`rtems_port_internal_to_external` 函数可以把一个特定端口的内部地址转换成外部地址。需要主要的是如果试图转换一个 DPMA 管理之外的地址, 那么这个要被转换的地址将直接赋值给输出参数。所有在写应用程序的时候, 为了保证可靠性, 需要对返回结果进行分析。

删除一个端口

函数调用信息:

```

rtems_status_code rtems_port_delete(
    Objects_Id id
)

```

函数功能描述:

通过 `rtems_port_delete` 函数可以把一个端口从系统中删除。此函数通过调用:

```

_Objects_Close( &_Dual_ported_memory_Information, &the_port->Object );
_Dual_ported_memory_Free( the_port );

```

`_Objects_Close` 函数调用后, `_Dual_ported_memory_Free` 函数直接调用了 `_Objects_Free` 函数, 来完成最终的释放工作。当一个端口被删除后, 它的 DPCB 返回到 DPCB 的空闲表中。

小节

第十章 文件系统

概述

在简单的嵌入式系统中, 不需要长期存储数据, 因此不需要嵌入式操作系统的文件管理功能。这也直接导致了在不少简单的嵌入式操作系统中不提供文件系统和文件管理功能。但在相对复杂的环境下, 需要对数据进行长期保存和访问, 这样没有文件系统的功能, 很难有效地支持上述需求。

早期的 RTEMS 没有对文件系统的支持。在目前的 RTEMS 中, 支持 IMFS(In Memory Filesystem, 内存文件系统)和 DOSFS 文件系统。其中 IMFS 是 RTEMS 文件系统的基础它和 Linux 中的 VFS 比较类似。对于 IMFS 而言, 需要管理的所有的文件和目录都在内存中。而其他的文件系统都需要挂载于这个文件系统中。DOSFS 文件系统实际上就是 Microsoft 公司提出的 FAT 文件系统, 在早期的桌面计算机系统中使用相当广泛。随着硬盘容量的迅速增加, FAT 文件系统在当前的桌面领域和服务端领域逐渐被 NTFS、EXT3 等新型文件系统所替代。但在嵌入式系统中, FAT 文件系统由于其简单和通用的特点而被继续使用。

本章首先介绍 RTEMS 的特点, 并分析其文件系统架构, 然后对 RTEMS 文件系统虚拟文件系统设计和相关的数据结构进行分析, 接下来对 IMFS 和 DOSFS 文件系统的关键数据结构和相关函数实现进行分析。

RTEMS 中文件系统的特点

RTEMS提供了POSIX标准的编程接口，使得用户可以通过POSIX编程接口方式对文件系统中的文件，外设及目录信息进行有效的操作。这些特性与传统的 UNIX文件系统的特性类似，并且RTEMS的文件系统的实现可以很方便地被扩展或者修改。 RTEMS提供IMFS以及 FAT12 / 16 / 32文件系统，可在内存中和软盘/硬盘上保存数据并访问数据。对上层应用而言，RTEMS文件系统支持如下功能：

- ☞ 可 mount 的文件系统
- ☞ 层次文件系统目录结构
- ☞ 文件目录操作的POSIX接口
- ☞ 文件目录支持
- ☞ 读，写及执行的权限
- ☞ 用户ID
- ☞ 组ID
- ☞ 访问时间
- ☞ 修改时间
- ☞ 建立时间
- ☞ 到文件及目录的硬连接
- ☞ 到文件及目录的符号连接

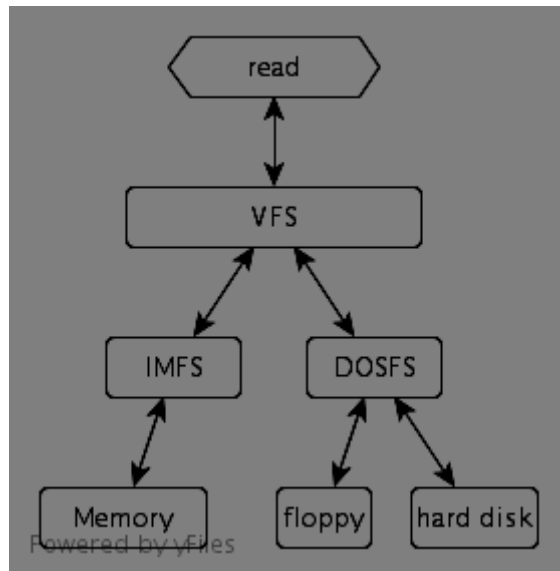
由于 RTEMS文件系统内容繁多复杂，我们从总体架构，以及底层实现函数和上层应用可调用的 API接口入手，分析RTEMS文件管理机制以及IMFS/DOSFS的部分实现细节。

VFS 架构

RTEMS 支持 IMFS 和 DOSFS(即 FAT)文件系统。其中 IMFS(In Memory Filesystem)是 RTEMS 文件系统的基础。对于 IMFS 而言，需要管理的所有的文件和目录都在内存中。而其他的文件系统都需要挂载于这个文件系统中。

当 RTEMS 系统启动时，在进行初始化的过程中，会调用 `rtems_filesystem_initialize` 函数。这个函数完成对 IMFS 的挂载安装，并把 IMFS 作为根文件系统 (`base filesystem`，也称基文件系统)工作。对于 DOSFS 文件系统而言，RTEMS 实现了 DOSFS 文件系统的相关函数提供给接口。而在底层，RTEMS 可以支持 FAT12/16/32多种文件系统格式。

在 RTEMS 中，尽管支持不同的文件系统，但通过虚拟文件系统抽象层，这些不同的文件系统可以使用统一的接口，主要包括访问文件的接口函数表和访问文件系统的接口函数表，给上层提供统一的服务，使得具体文件系统的底层实现可被屏蔽。通过使用统一的接口，使得文件系统的底层实现被屏蔽。为了同时支持不同的文件系统，RTEMS 为每个文件系统设置挂载点 (`mount table entry`)，当一个文件系统被挂载时，它的 `mount table entry` 会被初始化并加入挂载点链表 (`Mount Table Chain`，简称挂载表)；当该文件系统被卸载时，对应的 `mount table entry` 会从 `Mount Table Chain` 中删除。比如 `read` 函数，上层接口由 VFS 文件系统完成。VFS 的 `read` 函数会进一步调用不同的文件系统提供的底层 `read` 实现函数，来完成实际的读文件工作。大致的架构如下：



而 VFS 的详细架构如下图所示：

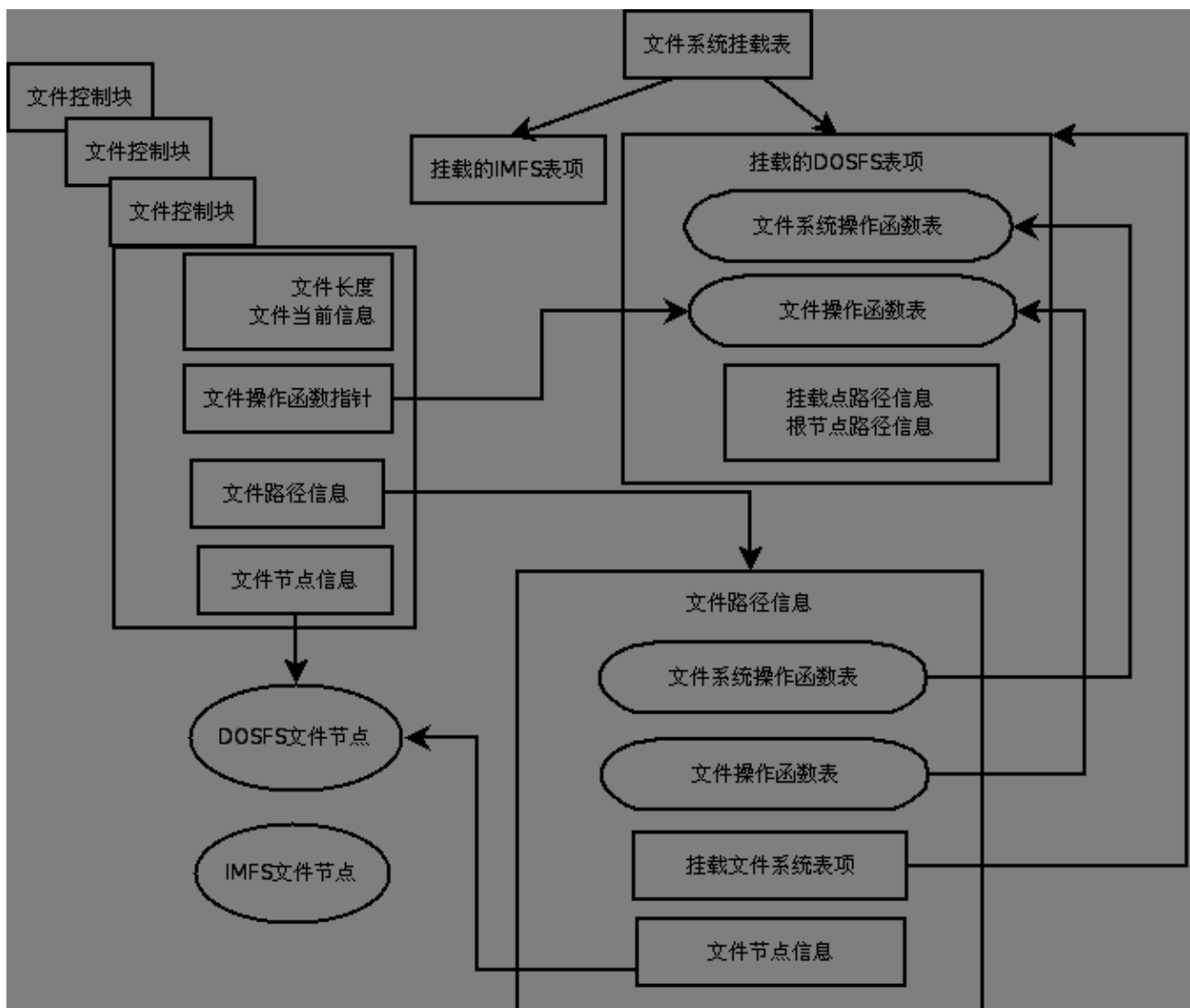


图 RTEMS VFS 详细架构图

对于上述图中每个组成单元的详细说明位于下面的各个小节中。

VFS 的关键数据结构

RTEMS 中文件系统的虚拟抽象层主要定义在 `rtems/cpukit/libcsupport/include/ rtems/libio.h` 中，这个文件中定义了与 VFS 有关的主要数据结构。在 RTEMS 中，与 VFS 相关的关键结构体（数据结构）有：

- ✎ `rtems_libio_tt`: 文件控制块
- ✎ `rtems_filesystem_location_info_tt`: 文件路径信息
- ✎ `rtems_filesystem_file_handlers_r`: 文件操作函数表
- ✎ `fstab_t`: 文件系统描述信息
- ✎ `rtems_filesystem_operations_table`: 文件系统操作函数表
- ✎ `rtems_filesystem_mount_table_t`: 启动时间期间的文件系统挂载表
- ✎ `rtems_filesystem_mount_table_entry_t`: 挂载的文件系统表项

文件控制块

`rtems_libio_tt` 结构体定义了 RTEMS 用于管理打开的文件的文件控制块，其中保存了关于文件的各种控制信息。

```
struct rtems_libio_tt {
    rtems_driver_name_t      *driver;
    off_t                    size;      /* size of file */
    off_t                    offset;    /* current offset into file */
    uint32_t                 flags;
    rtems_filesystem_location_info_t pathinfo;
    rtems_id                 sem;
    uint32_t                 data0;     /* private to "driver" */
    void                    *data1;    /* ... */
    void                    *file_info; /* used by file handlers */
    rtems_filesystem_file_handlers_r *handlers; /* type specific handlers */
};
```

上面的结构包含了与文件相关的一些重要的信息。比如文件的大小--size，当前操作的偏移量--offset。pathinfo包含了文件的路径信息。handler是特定的文件系统所提供的对文件的操作函数指针列表，包括open、close、read、write等常见的文件访问函数。这些函数会在对文件进行操作时被调用，同时，这些操作函数依赖于不同文件系统的具体实现。file_info是操作文件时的文件描述符，它会根据不同类型的文件系统而指向对于的文件节点信息，如IMFS的IMFS_jnode_t类型或FATFS的 fat_file_fd_t类型的节点信息。对pathinfo、file_info和handler的赋值是在对文件执行open操作的时候完成的（loc即文件路径信息变量，iop即文件控制块变量）：

```
.....
iop->handlers    = loc.handlers;
iop->file_info    = loc.node_access;
iop->flags       |= rtems_libio_fcntl_flags( flags );
iop->pathinfo     = loc;
.....
```

在RTEMS中，文件控制块的个数决定了应用可以访问的最大文件数。文件控制块的个数在`confdefs.h`中有定义：

```
#ifdef CONFIGURE_INIT
uint32_t    rtems_libio_number_iops = CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTOR;
#endif
```

在执行`rtems_libio_init`函数的时候，会完成文件控制块数组`rtems_libio_iops`的创建，其个数为`rtems_libio_number_iops`。

文件路径信息

```
struct rtems_filesystem_location_info_tt
{
    void                                *node_access;
    rtems_filesystem_file_handlers_r   *handlers;
    rtems_filesystem_operations_table   *ops;
    rtems_filesystem_mount_table_entry_t *mt_entry;
};
```

正如这个结构体的名称所示，它反应了表示在树状文件系统中的文件路径的所有信息。在实际使用中，此结构体主要用于管理文件的路径信息。

`node_access` 这个变量指向文件系统指定的节点信息，与特定的文件系统有关，用于对文件的路径进行查找访问，在执行文件 `open` 操作的时候，会把 `node_access` 赋值给文件控制块的 `file_info`。在文件系统对文件进行路径访问中 `node_access` 被赋值，对于一个文件系统的根（root）节点来说，这个变量在文件系统初始化的时候被赋值，直到这个文件系统被卸载。

`handlers` 的类型是 `rtems_filesystem_file_handlers_r` 结构体，这是一个封装了许多文件相关的操作函数的集合，如 `rtems_filesystem_open_t`，`rtems_filesystem_close_t` 等，这些都是函数指针，也就是说，`handlers` 会被初始化成一个操作集合，表示对文件的节点信息可以执行哪些操作。在具体操作时，会指定为特定文件系统所支持的对应操作函数集合。

`ops` 的类型是 `rtems_filesystem_operations_table` 结构体，此结构封装了许多文件系统相关的操作集合。和 `handlers` 不同的地方在于，`ops` 包含的是静态操作指针，也就是说，它提供的操作函数与具体的文件节点无关，任何文件都可以调用这里面的任何函数。

`mt_entry` 代表了一个 `mount_table` 的表项，其类型是 `rtems_filesystem_mount_table_entry_tt` 结构体。它很重要，因为它包含了被挂载的文件系统的信息。而文件系统只有被挂载后才能对其中的文件进行访问。

文件操作函数表

`rtems_filesystem_file_handlers_r` 结构体给出了用于处理文件的各个接口函数，实现了虚拟文件系统接口和实际文件系统的连接。其定义位于在 `libio.h` 中，具体描述如下：

```
struct _rtems_filesystem_file_handlers_r {
    rtems_filesystem_open_t      open_h;
    rtems_filesystem_close_t     close_h;
    rtems_filesystem_read_t      read_h;
    rtems_filesystem_write_t     write_h;
    rtems_filesystem_ioctl_t     ioctl_h;
    rtems_filesystem_lseek_t     lseek_h;
    rtems_filesystem_fstat_t     fstat_h;
    rtems_filesystem_fchmod_t    fchmod_h;
    rtems_filesystem_ftruncate_t ftruncate_h;
    rtems_filesystem_fpathconf_t fpathconf_h;
    rtems_filesystem_fsync_t     fsync_h;
    rtems_filesystem_fdatasync_t fdatasync_h;
    rtems_filesystem_fcntl_t     fcntl_h;
    rtems_filesystem_rmnode_t    rmnode_h;
};
```

在这个结构里，给出了实际文件系统中对文件进行各种操作的函数指针，如常见的文件访问函数 `open`、`close`、`read` 和 `write` 等函数。在对具体文件进行 `open` 操作的时候，会根据具体文件系统的类型

来对这些函数指针进行具体的赋值。

文件系统描述信息

```
typedef struct {
    char *dev;
    char *mount_point;
    rtems_filesystem_operations_table *fs_ops;
    rtems_filesystem_options_t mount_options;
    uint16_t report_reasons;
    uint16_t abort_reasons;
} fstab_t;
```

文件系统描述信息结构体定义了 RTEMS 要挂载的具体文件系统需要提供的信息。`dev` 表示具体系统所在的物理设备；`mount_point` 表示文件系统的路径（用字符串表示）；`fs_ops` 表示具体文件系统的底层操作函数表；`report_reasons` 表示进行挂载操作后的结果；`abort_reasons` 表示忽略挂载操作的原因；`mount_options` 是一个简单的枚举类型：

```
typedef enum
{
    RTEMS_FILESYSTEM_READ_ONLY,
    RTEMS_FILESYSTEM_READ_WRITE,
    RTEMS_FILESYSTEM_BAD_OPTIONS
} rtems_filesystem_options_t;
```

在 `rtems/testsuits/samples/fileio` 目录下的例子中，有一个具体使用了此结构体的代码片断（位于此目录的 `init.c` 中），可以把上面的介绍更加具体化的表示出来：

```
.....
fstab_t fs_table[] = {
    {
        "/dev/hda1", "/mnt/hda1",
        &msdos_ops, RTEMS_FILESYSTEM_READ_WRITE,
        FSMOUNT_MNT_OK | FSMOUNT_MNTPNT_CRERR | FSMOUNT_MNT_FAILED,
        0
    },
    .....
}
.....
void fileio_fsmount(void)
{
    rtems_status_code rc;
    .....
    rc = rtems_fsmount( fs_table,
                        sizeof(fs_table)/sizeof(fs_table[0]),
                        NULL);
    .....
}
```

上面定义了一个 `fs_table[]` 数组，应用定义的函数 `fileio_fsmount` 调用了 `rtems_fsmount` 函数，`rtems_fsmount` 函数会进一步调用 `mont` 函数来完成第 `fs_table[]` 数组中描述的具体的 FATFS 文件系统进行挂载操作。

文件系统操作函数表

```
struct _rtems_filesystem_operations_table {
    rtems_filesystem_evalpath_t  evalpath_h;
    rtems_filesystem_evalmake_t  evalformake_h;
    rtems_filesystem_link_t      link_h;
    rtems_filesystem_unlink_t    unlink_h;
    rtems_filesystem_node_type_t node_type_h;
    rtems_filesystem_mknod_t     mknod_h;
```

```

rtems_filesystem_chown_t    chown_h;
rtems_filesystem_freenode_t freenod_h;
rtems_filesystem_mount_t    mount_h;
rtems_filesystem_fsmount_me_t fsmount_me_h;
rtems_filesystem_unmount_t  unmount_h;
rtems_filesystem_fsunmount_me_t fsunmount_me_h;
rtems_filesystem_utime_t    utime_h;
rtems_filesystem_evaluate_link_t eval_link_h;
rtems_filesystem_symlink_t  symlink_h;
rtems_filesystem_readlink_t readlink_h;
};

```

每个具体的文件系统都需要定义一个对应的文件系统操作函数表。此表给出了对实际文件系统中进行各种底层操作的函数指针，如常见的文件访问函数 mount、unmount、mknod 和 freenod 等函数。在对具体文件进行 open 操作的时候，会根据具体文件系统的类型来对这些函数指针进行具体的赋值。

这里需要注意，mount_h 函数和 fsmount_me_h 函数有一定的区别，从 IMFS 和 DOSFS 的实现可以看出，fsmount_me_h 函数完成了具体文件系统的初始化和主要挂载工作，mount_h 函数则完成部分挂载工作。unmount_h 函数和 fsunmount_me_h 函数有一定的区别，fsunmount_me_h 函数完成具体文件系统的主要卸载工作，unmount_h 函数完成部分卸载工作。

evalpath_h 函数和 evalformake_h 函数主要工作是查询路径，但二者有一定的区别。evalpath_h 函数调用信息如下：

```

int evalpath_h(
  const char      *pathname, /* IN */
  int             flags,     /* IN */
  rtems_filesystem_location_info_t *pathloc /* IN/OUT */
)

```

evalpath_h 函数根据传入的文件路径字符串信息 pathname 查询对应的文件路径信息 pathloc 是否合法，如果合法，则要设置文件路径信息 pathloc 相关的文件操作函数表。

evalformake_h 函数调用信息如下：

```

int evalformake(
  const char      *path, /* IN */
  rtems_filesystem_location_info_t *pathloc, /* IN/OUT */
  const char      **name /* OUT */
)

```

evalformake_h 函数除了完成类似 evalpath_h 函数的工作外，还要创建具体文件系统的节点。根据传入的文件路径字符串 path 信息查询文件路径的合法性，如果合法，根据文件路径的最后一个字符串（称为 token）创建新的文件路径信息 pathloc，返回对应的文件名 name。改进!!!! ????

文件系统挂载项代表一个挂载的文件系统。文件系统挂载项的定义如下：

```

struct rtems_filesystem_mount_table_entry_tt {
  Chain_Node      Node;
  rtems_filesystem_location_info_t mt_point_node;
  rtems_filesystem_location_info_t mt_fs_root;
  int             options;
  void            *fs_info;
  rtems_filesystem_limits_and_options_t pathconf_limits_and_options;
  char            *dev;
};

```

其中的 Node 产生一个有序链表，包含了 mount table entry 的节点。Node 用于将不同的文件系统的信息形成链表，在 RTEMS 中，所有被挂载的文件系统通过链表 rtems_filesystem_mount_table_control

进行维护。

mt_point_node 包含了需要访问的挂载点的所有路径信息，在执行挂载文件系统的 mount 函数中，完成对挂载点的路径分析后，会得到一个具体的文件路径信息 loc，并把其中的一些域赋值给 mt_point_node。主要的操作如下(temp_mt_entry 表示要挂载的文件系统表项)：

```
.....
temp_mt_entry->mt_point_node.node_access = loc.node_access;
temp_mt_entry->mt_point_node.handlers = loc.handlers;
temp_mt_entry->mt_point_node.ops = loc.ops;
temp_mt_entry->mt_point_node.mt_entry = loc.mt_entry;
.....
```

mt_fs_root 包含了被挂载的文件系统的根目录的文件路径信息。当进行文件路径解析时，这个被挂载的文件系统的根目录是解析的起始点。fs_info 用来指示特定文件系统的系统级信息，对于 IMFS，它指向 IMFS_fs_info_t 结构的变量，对于 DOSFS，它指向 fat_fs_info_t 变量。dev 指向当前文件系统所在的块设备信息，这样可以访问到具体装载文件系统的块设备。

文件系统挂载表链表

在使用文件系统前，首先需要挂载 (mount) 文件系统，使用完毕后，要卸载 (umount) 文件系统。一个文件系统需要挂载到一个挂载点 (mount point) 上。挂载点必须要具有以下特征：

- ❧ 挂载点必须是一个目录，此目录下可以有文件或者其他目录，而这些文件和子目录在挂载时将被隐藏。
- ❧ 任务对挂载点必须要设置允许“读/写/执行”等命令，否则挂载请求将被拒绝。
- ❧ 只有文件系统才能被挂载到一个特定的挂载点。

❧ 可挂载的文件系统在挂载完成以后，挂载点标识符是该文件系统的根节点的引用。???? 不通

在 RTEMS 启动的时候，在缺省情况下，必须要挂载一个文件系统-IMFS 作为根文件系统，启动时的挂载文件系统链表是一个全局变量，位于 RTEMS 配置文件 condefs.h 中，定义如下：

```
rtems_filesystem_mount_table_t
    *rtems_filesystem_mount_table = &configuration_mount_table;
```

为了方便对文件系统的挂载进行管理，RTEMS 基于其 object handler 组件提供的对象链表机制，采用了一个动态的链表结构用来动态记录和维护多层次文件系统中一个特定文件系统的挂载点的信息。此链表是一个全局变量，位于 rtems/cpukit/libcsupport/src/mount.c 中。其定义如下：

```
Chain_Control rtems_filesystem_mount_table_control;
```

链表的每一项是一个文件系统挂载项，在挂载文件系统 (mount) 操作发生时被加进链表中；在卸载文件系统操作发生时该项被删除出链表。

VFS 的关键实现函数

初始化 rtems 文件系统

当 RTEMS 的基本初始化完成之后，关于文件系统的初始化也就开始了。首先会挂载一个基于内存的文件系统-IMFS 作为根文件系统 (base filesystem, 基础文件系统)，这个 IMFS 的根目录就会作为整个文件系统结构树的逻辑根目录 (/)，后面挂进来的文件系统都是在这个根目录下。在根目录下的 dev 目录很重要，因为大部分的设备驱动程序在此目录下会建立设备文件。文件系统的初始化工作主要是通过调用 rtems_filesystem_initialize() 来完成的。文件系统初始化的来龙去脉如下所示：

```
boot_card->rtems_initialize_executive_early->bsp_pretasking_hook->bsp_libc_init->rtems_li
```

首先 `rtems_libio_init` 函数会完成对文件控制块数组 `rtems_libio_iops` 的空间分配和初始化工作，创建保证对文件控制块数组 `rtems_libio_iops` 互斥访问的二值信号量 `rtems_libio_semaphore`。然后再调用 `rtems_filesystem_initialize` 函数完成文件系统的初始化工作。`rtems_filesystem_initialize` 函数会根据 RTEMS 配置表 (`confdefs.h`) 的信息, 先将一部分内存初始化并挂载为 IMFS, 把它作为根文件系统。这样才能进行进一步的 RTEMS 系统初始化工作, 比如在进行设备驱动程序初始化是, 建立设备文件等。这个函数的流程是这样的:

1. 设置默认的 `umask` 为022
2. 调用 `init_fs_mount_table` 函数来初始化 `rtems_filesystem_mount_table_control` 个链表全局变量
3. 得到初始化第一个 `mount table chain` 的表项并通过 `mount` 函数（在后面与进一步分析）进行挂载，以指示 `mount point` 是 `NULL` 并且被 `mount` 的文件系统是根文件系统
4. 设置根路径和当前路径为 “/”
5. 建立起 `/dev` 这个目录，这样方便后面的驱动程序建立设备文件

调用 `mount` 函数的时候, `mount point` 这个参数传入的是 `NULL`, 这个标志指示着当前挂载的文件系统是第一个文件系统。

根据文件系统描述信息创建挂载点并挂载文件系统

`rtems_fsmount` 函数完成了根据文件系统描述信息创建挂载点并挂载文件系统的功能。

函数调用信息:

```
int rtems_fsmount
(
  const fstab_t *fstab_ptr,
  int fstab_count,
  int *fail_idx
)
```

`fstab_ptr` 指向属于 `fstab_t` 结构体的数组, 此数组描述的要挂载的文件系统的信息。`fstab_count` 表示了此数组的大小。`fail_idx` 表示挂载出错的数组索引。

函数功能描述:

`rtems_fsmount` 函数的执行流程如下:

1. 调用 `rtems_fsmount_create_mountpoint` 函数, 根据 `fstab_ptr->mount_point` 指向的路径信息, 获得或创建与此路径对应的文件目录节点信息;
2. 调用 `mount` 函数, 对属于 `fstab_ptr` 指向的数组中的每一个元素进行挂载文件系统的工作。

`rtems_fsmount_create_mountpoint` 函数主要完成根据路径信息, 调用 `IMFS_get_token` 函数, 判断文件目录节点信息是否存在, 如果存在, 则正常返回, 如果不存在, 则调用 `mknod` 函数创建对路径信息应的文件目录节点。

注意: 有关 `mount` 函数和 `mknod` 函数的分析参见接下来的内容。

挂载单个文件系统

在 RTEMS 中, 挂载一个文件系统是通过 `mount` 函数实现的。一个文件系统只有被挂载后, 才能被应用访问。RTEMS 为所有的文件系统提供统一的 `mount` 接口函数, 给上层应用调用, 此函数实现在 `/cpukit/libcsupport/src/mount.c` 中, 这说明这个函数是 `newlib` 需要访问的。

函数调用信息:

```
int mount(
  rtems_filesystem_mount_table_entry_t **mt_entry,
```

```

    rtems_filesystem_operations_table    *fs_ops,
    rtems_filesystem_options_t          options,
    char                                 *device,
    char                                 *mount_point
)

```

函数功能描述:

mount 函数的执行流程如下:

1. 首先设置局部变量, 并检查挂载是否合法: 输入参数是否合法, 当前的挂载点是否已经挂载了文件系统, 当前的文件系统是否已经被挂载, 以及挂载的权限问题等。当不满足条件时, 进入出错处理;
2. 如果挂载合法, 给该文件系统分配一个 mount table entry, 设置这个表项的属性;
3. 检查 mount point 的合法性, mount point 应该是一个目录, 有读写执行的权限, 首先通过调用 rtems_filesystem_evaluate_path 获得绝对路径, 然后来检查这个路径的合法性;
4. 如果文件系统提供了挂载函数 fsmount_me_h, 则调用该函数进行挂载。否则进入出错处理。
5. 如果挂载成功, 则将该文件系统相应的 mount table entry 加入到 Mount Table 链中。

在 mount 函数中, 检查挂载的合法性的工作比较多, 需要确保挂载点必须是一个允许读/写/执行操作的目录, 且一个文件系统只能被挂载在一个挂载点上。另外在挂载点通过检查后, 会将 rtems_filesystem_location_info_t 类型的临时变量 loc 中描述的文件路径信息拷贝到临时创建的挂载表链表的表项中。需要注意的是, 在 loc 中保存的信息不应该在该文件系统被卸载之前被释放, 因为它可能要用来遍历文件系统目录树。真正的挂载工作是在 fsmount_me_h 中进行的, 而它是输入参数 fs_ops 中的一个函数, 此函数的实现与具体的文件系统相关, 完成具体文件系统的挂载工作。在后面的部分将分析 IMFS 和 DOSFS 文件系统的挂载, 将会进一步分析。

卸载单个文件系统

unmount 函数的功能是卸载一个文件系统。和 mount 函数一样, 在 RTEMS 上层, 不同的文件系统使用统一的接口, 但在下层调用不同文件系统提供的函数。unmount 函数的定义在 /cpukit/libcsupport/src/unmount.c 中, 这说明这个函数是 newlib 需要访问的。

函数调用信息:

```
int unmount( const char *path )
```

函数的输入参数 path 是要卸载的文件系统的路径。

函数功能描述:

这个函数的实现过程大致如下所示:

1. 设置局部变量;
2. 调用 rtems_filesystem_evaluate_path 函数, 根据 path 得到文件路径信息 loc;
3. 根据 loc 获取要卸载的文件系统的挂载点, 即 loc.mt_entry, 并根据挂载点获得要卸载文件系统的根目录文件路径信息, 挂载点文件路径信息;
4. 检查卸载是否合法, 包括该文件系统下是否还挂载了文件系统, 当前目录是否在该文件系统中, 该文件系统中是否有文件还被打开等;
5. 如果卸载合法, 且如果包含该要卸载的具体文件系统的挂载点的 unmount_h 函数存在, 则调用该函数, 清除该挂载点下的挂载信息;
6. 如果卸载合法, 且如果包含要卸载的文件系统的 fsunmount_me_h 函数存在, 则调用该函数, 释放该文件系统占用的资源;
7. 将要卸载的文件系统的 mount table entry 从 Mount Table Chain 中删除;
8. 释放该文件系统的 mount table entry 占用的资源。

卸载一个文件系统需要两个函数的支持, 一个清除挂载点的文件系统信息的 unmount_h, 另一个是要卸载的文件系统并释放节点内存的 fsunmount_me_h。这两个函数由特定的文件系统提供。在后面的部分将分析 IMFS 和 DOSFS 文件系统的挂载, 还会分析这些函数。

打开文件

open 函数用于打开一个文件。所有的文件系统使用统一的 open 接口，但对于不同的文件系统，它的底层实现是不同的。open 函数定义在/cpukit/libcsupport/src/open.c 中, 这说明这个函数是 newlib 需要访问的。

函数调用信息:

```
int open( const char *pathname, int flags, ...)
```

pathname 是要打开的文件的路径。flags 是相关的选项，表示是否要创建一个文件等。

函数功能描述:

open 函数的实现过程是这样的:

1. 给要打开的文件分配文件控制块;
2. 调用 rtems_filesystem_evaluate_path 函数判断要打开的文件是否存在，如果文件不存在而且没有打开创建文件的选项，则出错处理;
3. 如果 flag 中不存在且打开创建文件的选项，则调用 mknod 函数创建该文件的节点;
4. 如果该文件存在，则调用具体文件系统提供的 rtems_filesystem_file_handlers_r 结构体的成员变量 open_h 函数指针打开文件;
5. 填写相应的文件控制块信息。

从上面的过程可以知道，打开一个文件的真正操作是由具体文件系统提供的 open_h 函数提供的。

关闭文件

close 函数的作用是关闭一个文件，这个函数定义在/cpukit/libcsupport/src/close.c 中, 说明这个函数是 newlib 需要访问的。

函数调用信息:

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

fd 是指文件描述符，即文件控制块数组 rtems_libio_iops 的索引下标

函数功能描述:

函数的输入是要关闭的文件的描述符。函数的实现是这样的:

1. 检查合法性，包括文件是否存在，文件是否还被打开;
2. 调用此文件对应的具体文件系统提供的 rtems_filesystem_file_handlers_r 结构体的成员变量 close_h 函数指针关闭文件;
3. 释放文件路径信息所占空间;
4. 释放文件的控制块。

需要注意的是，关闭文件的操作是由特定的文件系统提供的 close_h 实现的。

创建文件系统对象

RTEMS 文件系统的 mknod 函数定义在/cpukit/libcsupport/src/mknod.c 中, 此函数并不是 POSIX 1003.1b 标准中定义的函数，但它被许多 UNIX 系统和符合 POSIX 标准的操作系统所支持。它主要用于创建文件系统对象。

函数调用信息:

```
int mknod(  
    const char *pathname,  
    mode_t      mode,  
    dev_t       dev
```

```
)
```

函数功能描述:

1. 根据文件路径 `pathname`, 调用宏 `rtems_filesystem_get_start_loc`, 判断文件路径是绝对路径还是相对路径, 并获得对应的文件路径信息变量;
2. 调用文件路径信息变量的成员变量 `ops->evalformake_h` 函数指针对文件路径进行分析;
3. 调用文件路径信息变量的成员变量 `ops->mknod_h` 函数指针完成对文件节点的创建;
4. 调用 `rtems_filesystem_freenode` 宏, 释放文件路径信息变量 `temp_loc`。

`rtems_filesystem_freenode` 宏实际上是对文件路径信息变量的成员变量 `ops->freenod_h` 函数指针的封装, 具体实现在 `ops->freenod_h` 函数指针所指向的函数中。

创建文件

RTEMS 文件系统的 `create` 函数定义在 `/cpukit/libcsupport/src/creat.c` 中, 这说明这个函数是 `newlib` 需要访问的。

函数调用信息:

```
int creat (const char *path, mode_t mode)
```

函数功能描述:

从代码中可以知道, `creat` 函数是对带特定调用参数的 `open` 函数的封装。`open` 函数具体完成文件的创建。当在 `open` 中创建一个文件时, 需要调用 `mknod` 函数, 和前面的函数类似, `mknod` 函数是还是靠调用具体文件系统提供的 `mknod_h` 函数指针来实现的。

读文件

`read` 函数定义在 `/cpukit/libcsupport/src/read.c` 中, 这说明这个函数是 `newlib` 需要访问的。

函数调用信息:

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

函数的参数 `fd` 是要读文件的描述符, `buffer` 是读出数据的缓冲区, `count` 是读出数据的大小。返回值是实际读出的数据大小。

函数功能描述:

此函数首先获取文件的控制块信息, 接着进行必要的检查, 然后调用具体文件系统提供的 `read_h` 函数指针进行读操作, 这个函数会返回实际读的字节数 `rc`, 最后将文件控制块信息中的文件的偏移量 `offset` 加上 `rc`。`read` 函数的实际实现是靠具体文件系统提供的 `read_h` 函数来完成。

写文件

`write` 函数定义在 `/cpukit/libcsupport/src/write.c` 中, 统一介绍。

函数调用信息:

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

函数的参数 `fd` 是要写文件的描述符, `buffer` 是写出数据的缓冲区, `count` 是写出数据的大小。返回值是实际写出的数据大小。

函数功能描述:

此函数首先获取文件的控制块信息, 接着进行必要的检查, 然后调用具体文件系统提供的 `write_h` 函数指针进行写操作, 这个函数会返回实际写的字节数 `rc`, 最后将文件控制块信息中的文件的偏移量 `offset` 加上 `rc`。因此, `write` 操作的实现依赖于具体文件系统的 `write_h`。

IMFS 文件系统

IMFS 是一个符合 POSIX 标准的内存文件系统，文件在内存中按块存储，按照类似于 UNIX 文件系统中的索引节点方式来进行管理。如果系统掉电，则保存在内存中的文件信息将会丢失。一个内存文件系统从系统的内存资源获得文件和目录的资源信息。在卸载文件系统的时候，支持此系统的内存资源被释放。释放这些资源使用递归方式，子文件/目录都删除后，父目录才删除。

关键数据结构

在 `imfs.h` 中定义了内存文件系统的相关结构：

IMFS 文件类型

IMFS 的文件类型有如下几种：

- IMFS DIRECTORY: IMFS 目录
- IMFS MEMORY FILE: IMFS 内存文件
- IMFS HARD LINK: IMFS 硬链接
- IMFS SYM LINK: IMFS 符号链接
- IMFS DEVICE : IMFS 设备文件

这些不同类型的文件如下图所示：

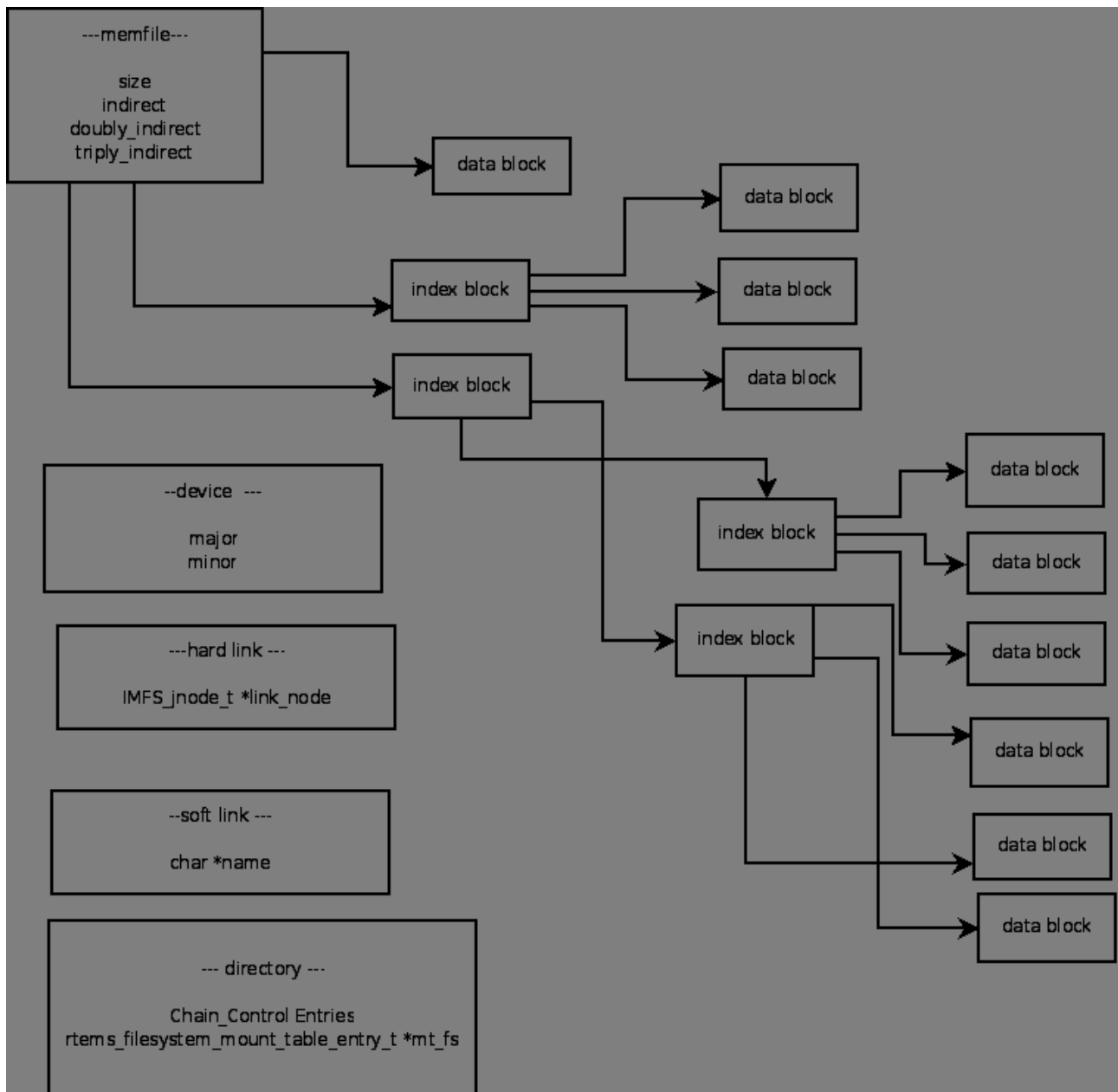


图 IMFS 中的五种文件类型

内存文件结构体和操作函数列表

内存文件结构体定义如下：

```
typedef struct {
    off_t      size;           //文件大小(bytes)
    block_ptr  indirect;       //128个数据块指针数组
    block_ptr  doubly_indirect; //128个块
    block_ptr  triply_indirect; // 128个doubly块
} IMFS_memfile_t;           //内存文件类型
```

`memfile` 类型的文件是典型的基于索引节点函数的文件，通过直接索引和二级/三级间接索引的数据块来表示。一个数据块的缺省大小为：

```
#define IMFS_MEMFILE_DEFAULT_BYTES_PER_BLOCK 128
```

一个数据块中可以保存的数据块指针个数为：

```
#define IMFS_MEMFILE_BLOCK_SLOTS \
    (IMFS_MEMFILE_BYTES_PER_BLOCK / sizeof(void *))
```

在32bit CPU 的计算机系统中，sizeof (void *)的值为4。所以数据块指针个数为32。这样如果数据块大小为128Byte，则 memfile 类型可以表示的最大文件大小为：

```
(32*32*32+32*32+32)*128=4329472 Bytes
```

与内存文件结构体对应的操作函数列表如下：

```
rtems_filesystem_file_handlers_r IMFS_memfile_handlers = {
    memfile_open,
    memfile_close,
    memfile_read,
    memfile_write,
    memfile_ioctl,
    memfile_lseek,
    IMFS_stat,
    IMFS_fchmod,
    memfile_ftruncate,
    NULL,          /* fpathconf */
    IMFS_fdatasync, /* fsync */
    IMFS_fdatasync,
    IMFS_fcntl,
    memfile_rmnod
};
```

线性文件结构体和操作函数列表

线性文件结构体定义如下：

```
typedef struct {
    off_t      size;           // 文件大小
    block_p    direct;        // 指向文件镜像的指针
} IMFS_linearfile_t;         //线性文件类型
```

这种文件相对简单，只是指向一个连续线性空间的数据块。由于在 RTEMS 源码中没有发现与线性文件结构体对应的操作函数列表，所以我们认为 RTEMS 目前没有实现 IMFS 线性文件类型。

目录/设备/链接文件结构体和操作函数列表

目录/设备/链文件结构体分别定义如下：

```
struct IMFS_jnode_tt;        //IMFS文件节点结构，用于管理目录信息
typedef struct IMFS_jnode_tt IMFS_jnode_t;

typedef struct {
    Chain_Control      Entries; //目录项
    rtems_filesystem_mount_table_entry_t *mt_fs; //文件系统挂载表项
} IMFS_directory_t;         //目录文件类型

typedef struct {
    rtems_device_major_number major; //主设备号
    rtems_device_minor_number minor; //从设备号
} IMFS_device_t;            //设备文件类型
```

```
typedef struct {
    IMFS_jnode_t *link_node;    //IMFS文件节点
} IMFS_link_t;                //链接文件类型
```

目录文件比较特殊，它维护了一个链表，链表中的节点是位于此目录下的文件，同时还保存了它对应的挂载文件系统表项指针，这样有助于完成与 IMFS 文件系统相关的文件路径检索等操作。一个目录项的结构定义如下：

```
struct dirent {
    long    d_ino;
    off_t   d_off;        //此目录项在目录项链表中的偏移量
    unsigned short d_reclen; //dirent 结构体的大小
    unsigned short d_namlen; //目录项指定的文件名的大小
    char     d_name[NAME_MAX + 1]; //目录项指定的文件名
};
```

这个定义不在 rtems 软件包中，而是在 newlib 中的 newlib-1.15.0 /newlib/libc/sys/rtems/sys/dirent.h 文件中。

与目录文件结构体对应的操作函数列表如下：

```
rtems_filesystem_file_handlers_r IMFS_directory_handlers = {
    imfs_dir_open,
    imfs_dir_close,
    imfs_dir_read,
    NULL,          /* write */
    NULL,          /* ioctl */
    imfs_dir_lseek,
    imfs_dir_fstat,
    IMFS_fchmod,
    NULL,          /* ftruncate */
    NULL,          /* fpathconf */
    NULL,          /* fsync */
    IMFS_fdatasync,
    IMFS_fcntl,
    imfs_dir_rmdir,
};
```

设备文件仅仅维护了设备（device）的主节点号和从节点号，通过这两个信息，可以确定一个特定的设备。这种表示与 UNIX 操作系统中的设备文件的表示很类似。与设备文件结构体对应的操作函数列表如下：

```
rtems_filesystem_file_handlers_r IMFS_device_handlers = {
    device_open,
    device_close,
    device_read,
    device_write,
    device_ioctl,
    device_lseek,
    IMFS_stat,
    IMFS_fchmod,
    NULL, /* ftruncate */
    NULL, /* fpathconf */
    NULL, /* fsync */
    NULL, /* fdatasync */
    NULL, /* fcntl */
    IMFS_rmdir,
};
```

链接文件保存了一个 IMFS 的 node 指针，指向实际的文件。这与 UNIX 操作系统的硬链接的机制也是相似的。与链接文件结构体对应的操作函数列表如下：

```
rtems_filesystem_file_handlers_r IMFS_link_handlers = {
  NULL, /* open */
  NULL, /* close */
  NULL, /* read */
  NULL, /* write */
  NULL, /* ioctl */
  NULL, /* lseek */
  IMFS_stat, /* stat */
  NULL, /* fchmod */
  NULL, /* ftruncate */
  NULL, /* fpathconf */
  NULL, /* fsync */
  NULL, /* fdatsync */
  NULL, /* fcntl */
  IMFS_rmnod
};
```

IMFS 节点结构体

IMFS 节点结构体为 IMFS_jnode_tt，它为每个实例化的文件、设备和目录维护一个节点结构，用于管理用户权、访问权、访问时间、修改时间和创建时间。这些节点结构提供了文件数据、设备选择或者目录的管理，这些性质由符合 POSIX 标准的文件和目录函数提供。下面是其结构体描述：

```
struct IMFS_jnode_tt {
  Chain_Node Node; /* 节点链表 */
  IMFS_jnode_t *Parent; /* 指向父节点的指针 */
  char name[NAME_MAX+1]; /* 节点的相对路径名 */
  mode_t st_mode; /* 标准Unix文件访问模式 */
  nlink_t st_nlink; /* 对本文件有连接的个数，直到此值比1小才能删除jnode以及其相关资源 */

  ino_t st_ino; /* 节点标识数 */
  uid_t st_uid; /* 文件拥有者的用户ID */
  gid_t st_gid; /* 文件拥有者的组ID */
  time_t st_atime; /* 上次访问时间 */
  time_t st_mtime; /* 上次修改时间 */
  time_t st_ctime; /* 上次状态改变时间 */
  IMFS_jnode_types_t type; /* 目录类型 */
  IMFS_typs_union info; /* 记录与文件类型有关的信息 */
}
```

第一个成员变量 Node 很有意思，如果此节点有父节点，则这里的 Node 代表的是此节点的兄弟节点形成的节点链表，这些兄弟节点指向同一个父节点，即成员变量 Parent。

IMFS 文件系统管理函数表

有关 IMFS 文件系统的管理函数记录在属于 rtems_filesystem_operations_table 结构体的 IMFS_ops 所表示的函数列表中。此函数列表记录了针对 IMFS 文件系统的相关管理函数指针。实际上一个属于 rtems_filesystem_operations_table 结构体的特定函数指针列表对应 RTEMS 配置支持的一种文件系统。rtems_filesystem_operations_table 结构体描述的函数指针列表如下所示，在后面将详细说明主要的函数功能和实现。

```
rtems_filesystem_operations_table IMFS_ops = {
```

```

    IMFS_eval_path,
    IMFS_evaluate_for_make,
    IMFS_link,
    IMFS_unlink,
    IMFS_node_type,
    IMFS_mknod,
    IMFS_rmnod,
    IMFS_chown,
    IMFS_freenodinfo,
    IMFS_mount,
    IMFS_initialize,
    IMFS_unmount,
    IMFS_fsunmount,
    IMFS_utime,
    IMFS_evaluate_link,
    IMFS_symlink,
    IMFS_readlink
};

```

关键实现函数

IMFS 文件系统中的实现函数很多，这里只列出有代表性的一些函数进行分析，主要涉及文件系统的初始化、挂载、卸载、打开文件、关闭文件、创建文件、读写文件等。

IMFS 文件系统初始化和挂载

IMFS 文件系统初始化由 **IMFS_initialize** 函数实现，此函数实际上通过调用 **IMFS_initialize_support** 函数来完成具体的初始化过程。此函数的来龙去脉如下：

```
mount->fsmount_me_h->IMFS_initialize->IMFS_initialize_support
```

函数调用信息：

```

int IMFS_initialize_support(
    rtems_filesystem_mount_table_entry_t *temp_mt_entry    //挂载文件系统表项
    rtems_filesystem_operations_table    *op_table,        //文件系统操作函数表
    rtems_filesystem_file_handlers_r    *memfile_handlers  //内存文件处理
    rtems_filesystem_file_handlers_r    *directory_handlers //目录处理
)

```

函数功能描述

IMFS_initialize_support 函数的流程如下所示：

1. 检查每块的内存文件比特数；
2. 创建基于 jnode 结构体类型的根节点；
3. 用 IMFS_directory_handlers 作为根节点的文件处理函数列表；
4. 用 IMFS_ops 作为根节点的文件系统处理函数列表；
5. 创建 IMFS 的 fs_info 文件系统信息
6. 进一步设置根节点的访问类型和引用个数；
7. IMFS 初始化完毕。

设置 IMFS 文件系统资源

设置 IMFS 文件系统资源是通过 IMFS_mount 函数实现的。

函数调用信息:

```
int IMFS_mount( rtems_filesystem_mount_table_entry_t *mt_entry)
```

函数功能描述:

```
{ IMFS_jnode_t *node;
  node = mt_entry->mt_point_node.node_access; //取本条目指向的节点
  /* 检查此结点是否是指向目录的节点, 如果不是则报错 */
  if ( node->type != IMFS_DIRECTORY )
    rtems_set_errno_and_return_minus_one( ENOTDIR );
  /* 设置mt_fs指针指向文件系统的挂载表项*/
  node->info.directory.mt_fs = mt_entry;
  return 0;
}
```

清除 IMFS 文件系统资源

清除 IMFS 文件系统资源是通过 IMFS_unmount 函数实现的, 此函数完成了卸载文件系统的一部分工作。函数检查卸载的 IMFS 文件系统表项的文件路径信息的合法性, 并把文件路径信息的挂载表项, 即 node->info.directory.mt_fs 清空。执行完此函数后, 并没有完全卸载 IMFS 文件系统, 还要进一步调用 IMFS_fsunmount 函数才能达到目的。

函数调用信息:

```
int IMFS_unmount( rtems_filesystem_mount_table_entry_t *mt_entry)
```

函数功能描述:

```
{
  IMFS_jnode_t *node;
  node = mt_entry->mt_point_node.node_access;
  /*同样, 检查此结点是否是指向目录的节点, 如果不是则报错*/
  if ( node->type != IMFS_DIRECTORY )
    rtems_set_errno_and_return_minus_one( ENOTDIR );
  /*检查是否已经有目录挂载在本节点上了, 没有则报错*/
  if ( node->info.directory.mt_fs == NULL )
    rtems_set_errno_and_return_minus_one( EINVAL );
  /* 设置本节点的mt_fs指针为null, 表示没有文件系统挂载在此上了*/
  node->info.directory.mt_fs = NULL;
  return 0;
}
```

卸载 IMFS 文件系统

IMFS_fsunmount 函数实现了将挂载文件系统的根节点开始遍历文件树并释放为节点分配的所有内存工作。

函数调用信息:

```
int IMFS_fsunmount( rtems_filesystem_mount_table_entry_t *temp_mt_entry )
```

输入参数是文件系统挂载表项 temp_mt_entry

函数功能描述:

此函数主要根据文件系统挂载表项的信息，遍历要卸载文件系统的根目录文件路径信息，释放占用的相关内存。其主要流程如下所示：

1. 根据文件系统挂载表项 temp_mt_entry 获得根目录文件路径信息 loc 和 IMFS 根目录的节点信息 jnode;
2. 把 IMFS 根目录的节点信息设置为 NULL;
3. 执行一个循环直到 jnode==NULL: 调整文件路径信息 loc, 设置为卸载后的文件路径信息, 如果对应的 IMFS 文件不是目录, 要执行 IMFS_unlink 函数来释放资源, 并设置 jnode 为 jnode->Parent; 如果是目录, 把 jnode 设置为此目录第一个 IMFS 子节点。

打开 IMFS memfile 文件

对于 memfile 文件的打开操作是通过 memfile_open 函数来实现的。 memfile_open 函数中的参数指针 iop 的结构体是 rtems_libio_t, 此结构体是专为打开的文件而设置的文件结构, 即文件控制块, 其中包括文件的大小、偏移量、可写/可读标志、文件位置, 线程, 数据, 文件信息, 文件处理器等量。打开文件就是对于本文件创建一个节点 jnode, 维护好这个节点以便于后面对文件的操作。

函数调用信息:

```
int memfile_open( rtems_libio_t *iop, const char *pathname, uint32_t flag, uint32_t mode)
```

函数功能描述:

需要注意的是, 在执行 memfile_open 函数前, 对应的文件已经创建了。此函数被上层的 open 函数调用。 memfile_open 函数的流程如下所示:

1. 如果是线性文件且访问属性是write|append且count, 则对线性文件进行写复制;
2. 如果是一般文件且访问属性为append, 则调整此文件的offset;
3. 设置此文件的大小。

关闭 IMFS memfile 文件

对于 memfile 文件的关闭操作是通过 memfile_close 函数来实现的。

函数调用信息:

```
int memfile_close( rtems_libio_t *iop){
```

函数功能描述:

其流程如下所示:

```
{
    IMFS_jnode_t  *the_jnode;
    the_jnode = iop->file_info;  //首先用一个节点指向本文件
    if (iop->flags & LIBIO_FLAGS_APPEND)
        iop->offset = the_jnode->info.file.size;

    memfile_check_rmnod( the_jnode );
    return 0;
}
```

可以看到, 本函数也是首先用一个节点指向本文件, 然后调用函数memfile_check_rmnod, 我们来看看memfile_check_rmnod函数做了哪些事情。

检查并清除 IMFS memfile 文件

函数调用信息:

```
int memfile_check_rmnod( IMFS_jnode_t *the_jnode ) {
```

函数功能描述:

此函数检查memfile文件是否能够释放, 如果能够就完成对memfile文件的彻底释放。

其流程如下:

```
/*如果要关闭一个文件, 那么这个文件不能处于打开状态而且指向本文件的指针数少于1(也就是0) */
if ( !rtems_libio_is_file_open( the_jnode ) && (the_jnode->st_nlink < 1) ) {
    /*如果当前节点就是本节点*/
    if ( rtems_filesystem_current.node_access == the_jnode )
        rtems_filesystem_current.node_access = NULL;
    /*释放相应的内存空间, 释放节点jnode*/
    if (the_jnode->type != IMFS_LINEAR_FILE)
        IMFS_memfile_remove( the_jnode );
    free( the_jnode );
}
return 0;
}
```

具体的释放 memfile 文件所占资源是通过函数 IMFS_memfile_remove 来完成的, 下面, 我们看看其实现。

释放 IMFS memfile 文件所占内存资源

函数调用信息:

```
int IMFS_memfile_remove(
    IMFS_jnode_t *the_jnode
)
```

函数功能描述:

此函数完成了对memfile所占资源的释放。在具体实现上做得比较保守。它实际上遍历并释放memfile所有占用的所有数据块。具体的数据块释放工作是由_free_blocks_in_table函数完成的。最终调用free函数完成了对内存块的释放。

删除 IMFS memfile 文件

函数调用信息:

```
int memfile_rmnod(
    rtems_filesystem_location_info_t *pathloc /* IN */
)
```

函数功能描述:

此函数完成对IMFS memfile文件的删除工作。它的工作流程如下:

1. 根据输入参数pathloc得到IMFS的jnode;
2. 如果此jnode有父节点, 则把jnode从兄弟节点链表中去除, 把jnode指向的父节点指针Parent设置为NULL;
3. 把此jnode的引用计数st_nlink减一;
4. 调用memfile_check_rmnod函数, 看此jnode的引用计数是否为0, 如果是, 则进一步清除此节点。

创建 IMFS memfile 文件

函数调用信息:

```
int IMFS_mknod(
    const char          *token,      /* IN */
    mode_t              mode,        /* IN */
    dev_t               dev,         /* IN */
    rtems_filesystem_location_info_t *pathloc /* IN/OUT */
)
```

函数功能描述:

此函数的实现很简单, 首先根据参数 mode 设置文件的类型, 然后就调用 IMFS_create_node 函数完成与文件路径对应的 jnode。IMFS_create_node 函数函数的分析如下。

创建 IMFS 文件节点

函数调用信息:

```
IMFS_jnode_t *IMFS_create_node(
    rtems_filesystem_location_info_t *parent_loc,
    IMFS_jnode_types_t               type,
    char                             *name,
    mode_t                           mode,
    IMFS_types_union                  *info
)
```

函数功能描述:

IMFS_create_node 函数的流程如下所示:

1. 分配一个 jnode 的内存空间;
2. 初始化 jnode 的部分内容: st_link, type, name;
3. 设置 jnode 的 mode 和访问权限控制、用户 id 和组 id 信息 (属于 POSIX 的内容);
4. 设置 jnode 的时间信息;
5. 如果文件属于目录类型, 初始化目录链表; 如果文件属于硬链接 (hard link) 类型, 则设置 node->info.hard_link.link_node 为对应的节点信息; 如果文件属于符号链接 (sym link) 类型, 则设置 node->info.sym_link.name 为对应的文件名; 如果文件属于设备文件, 则设置主设备号和从设备号; 如果文件属于线性文件或内存文件, 则初始化文件大小为0, 各个 block 指针也为0;
6. 如果文件有父目录节点, 则把文件自身的 node 挂接到父目录节点的目录项列表中;
7. 增加此文件对应的挂载的 IMFS 文件系统的节点引用计数 (在 IMFS_fs_info_t 中的 ino_count), 并赋值给 jnode 结构中的 st_ino。

读 IMFS memfile 文件

如果是读 memfile, 则具体的处理函数是 memfile_read。

函数调用信息:

```
ssize_t memfile_read ( rtems_libio_t *iop, void *buffer, size_t count)
```

函数功能描述:

它的流程很简单。可以看出, 本函数主要根据文件描述符 iop 找到对应的节点 jnode, 然后传入参数 jnode, 文件偏移量, 缓冲以及要读出的数量, 调用 IMFS_memfile_read 函数来完成具体的读文件操作。

IMFS_memfile_read函数的调用信息如下:

```
MEMFILE_STATIC ssize_t IMFS_memfile_read(
    IMFS_jnode_t    *the_jnode, //参数1: 与本文件关联的节点
    off_t           start,      //参数2: 文件中开始读的地方
    unsigned char   *destination, //参数3: 目标地址
    unsigned int     length     //参数4: 读出数据长度
)
```

那么我们来看看IMFS_memfile_read函数是怎样实现的。本函数从jnode指向的内存文件读出数据到定义的目的数据缓冲地点。如果偏移量大于文件长度则报错。如果参数中读出的数据长度length超过了参数中的偏移量start到文件尾的长度,那么读出前者,后面的数据不读。

IMFS_memfile_read函数主要分以下几个步骤:

1. 检查jnode是关联一个内存文件
2. 检查读文件的目的地正确
3. 如果是线性文件,则在进一步调整读出数据长度(如果太长就剪短到至文件尾的长度),从文件的内存块中读出数据;
4. 如果是内存文件,则在进一步调整读出数据长度((如果太长就剪短到至文件尾的长度),把读出数据分为三个阶段:
 - ❧ 起始数据: 读出处可能是一个block的后面一部分,一直独到一个块的块头
 - ❧ 中间数据: 都是从块头读到块尾
 - ❧ 结尾数据: 结束读出处可能是一个block的前面一部分,需要单独处理
5. 更新文件访问时间。

读文件需要注意的地方是根据要读的文件偏移量确定文件内容所在的数据块,确定读文件的起始偏移和结束偏移所在块的位置。

写 IMFS memfile 文件

如果是写memfile类型的文件,这具体的处理函数是 **memfile_write**,此函数将特定数据(存储在参数buffer中)写到jnode所指的内存中。

函数调用信息:

```
ssize_t memfile_write(
    rtems_libio_t *iop,
    const void     *buffer, //buffer 是待写数据
    size_t         count //count 是数据大小
)
```

函数功能描述:

可以看到,根据文件描述符iop找到对应的节点jnode后,调用IMFS_memfile_write(),参数为jnode,文件偏移量,缓冲以及要写的数量。函数IMFS_memfile_write与函数IMFS_memfile_read在处理流程上基本类似,区别有两点,一是内存拷贝的方向是相反的,另外一个如果是当前文件的数据块不够存放数据了,则会分配新的数据块来保存数据。

打开 IMFS 目录文件

函数调用信息:

```
int imfs_dir_open(
    rtems_libio_t *iop,
    const char *pathname,
    uint32_t flag,
    uint32_t mode
)
```

函数功能描述:

此函数相对简单, 首先根据文件控制块iop得到IMFS节点jnode, 判断jnode的类型是否是 IMFS_DIRECTORY, 如果不是, 则报错; 如果是, 则把文件控制块iop的offset成员变量设置为0, 即指向了目录包含的第一个目录项。

关闭 IMFS 目录文件

函数调用信息:

```
int imfs_dir_close(
    rtems_libio_t *iop
)
```

函数功能描述:

此函数被VFS层的close函数调用, 主要的工作都被VFS层的close函数完成了, 这里只是返回一个0, 表示成功。

删除 IMFS 目录文件

函数调用信息:

```
int imfs_dir_rmnod(
    rtems_filesystem_location_info_t *pathloc /* IN */
)
```

函数功能描述:

此函数完成对IMFS目录文件的删除工作。其大致流程如下:

1. 根据输入参数pathloc得到IMFS的jnode;
2. 检查此jnode没有子节点, 不是文件系统根节点, 不是加载点, 如果是, 则报错返回;
3. 如果不是, 则进一步处理: 如果此jnode有父节点, 则把jnode从兄弟节点链表中去除, 把jnode指向的父节点指针Parent设置为NULL;
4. 把此jnode的引用计数st_nlink减一;
5. 调用free函数释放jnode。

读 IMFS 目录文件

函数调用信息:

```
ssize_t imfs_dir_read(
    rtems_libio_t *iop,
    void *buffer,
    size_t count
)
```

iop 代表目录文件描述符, buffer 是用来函数目录中的项的信息的内存块, count 是 buffer 的大小。

函数功能描述:

此函数根据文件描述符iop的成员变量offset确定目录项列表中的第一个目录项, 再根据参数count确定要读取的最后一个目录项。然后读取目录文件中的各个目录项信息, 保存到buffer中。这里需要注意对offset和count的处理, 使其能够对应到正确的第一个目录项和最后一个目录项。

打开 IMFS device 文件

函数调用信息:

```
int device_open(
    rtems_libio_t *iop,
    const char *pathname,
    uint32_t flag,
    uint32_t mode
)
```

```
)
```

函数功能描述:

此函数在open函数和rtems_io_open函数之间架起一个桥梁。其主要的工作如下所示:

1. 根据文件描述符iop获得IMFS的jnode信息;
2. 设置要传递给 rtems_io_open函数的参数args;
3. 调用 rtems_io_open函数完成对具体设备的open操作。

关闭 IMFS device 文件

函数调用信息:

```
int device_close(  
    rtems_libio_t *iop  
)
```

函数功能描述:

此函数在close函数和rtems_io_close函数之间架起一个桥梁。其主要的工作如下所示:

1. 根据文件描述符iop获得IMFS的jnode信息;
2. 设置要传递给 rtems_io_close函数的参数args;
3. 调用 rtems_io_close函数完成对具体设备的close操作。

读 IMFS device 文件

函数调用信息:

```
ssize_t device_read(  
    rtems_libio_t *iop,  
    void          *buffer,  
    size_t        count  
)
```

buffer 是要读出内容的内存块, count 是要读出的大小

函数功能描述:

此函数在read函数和rtems_io_read函数之间架起一个桥梁。其主要的工作如下所示:

1. 根据文件描述符iop获得IMFS的jnode信息;
2. 设置要传递给 rtems_io_read函数的参数args;
3. 调用 rtems_io_read函数完成对具体设备的read操作。

写 IMFS device 文件

函数调用信息:

```
ssize_t device_write(  
    rtems_libio_t *iop,  
    void          *buffer,  
    size_t        count  
)
```

buffer 是要写入内容的内存块, count 是要写入的大小

函数功能描述:

此函数在write函数和rtems_io_write函数之间架起一个桥梁。其主要的工作如下所示:

1. 根据文件描述符iop获得IMFS的jnode信息;
2. 设置要传递给 rtems_io_write函数的参数args;
3. 调用 rtems_io_write函数完成对具体设备的write操作。

IMFS device 文件的 ioctl 操作

函数调用信息:

```
int device_ioctl(  

```

```

rtems_libio_t *iop,
uint32_t      command,
void          *buffer
)

```

command 是要传递给设备的命令编码，buffer 用于读出或写入信息。

函数功能描述：

此函数在ioctl函数和rtems_io_ioctl函数之间架起一个桥梁。其主要的工作如下所示：

4. 根据文件描述符iop获得IMFS的jnode信息；
5. 设置要传递给 rtems_io_ioctl函数的参数args；
6. 调用 rtems_io_ioctl函数完成对具体设备的ioctl操作。

分析 IMFS 文件路径

函数调用信息：

```

int IMFS_eval_path(
const char          *pathname, /* IN */
int                 flags,     /* IN */
rtems_filesystem_location_info_t *pathloc /* IN/OUT */
)

```

path 是要分析的文件路径字符串，flags 是文件访问参数，pathloc 是文件路径信息

函数功能描述：

此函数主要完成对IMFS中的文件路径的分析，判断路径的合法性。其大致处理流程如下：

1. 根据参数pathloc得到IMFS的节点node；
2. 在一个while循环中，调用IMFS_get_token函数把pathname分解为一个一个的token（token用“/”作为分割符），并解析出pathname中的一个一个目录文件，直到最后一个；在解析目录中，如果发现了一个文件系统的加载点，则调用此文件系统的evalpath_h函数进一步进行解析；调用IMFS_find_match_in_dir函数分解掉一个pathname中的token（即进入pathname中的下一级目录），直到最后一个token；
3. 如果node是一个挂载点，则调用挂载的文件系统的evalpath_h函数，参数为根节点对应的文件路径信息；如果不是，则调用IMFS_Set_handlers函数，根据文件路径信息loc的类型，设置loc的文件处理函数列表。

????!!!! 需要进一步分析

为创建节点而分析 IMFS 文件路径

IMFS_evaluate_for_make函数与IMFS_eval_path函数的区别在于，IMFS_evaluate_for_make 函数需要分析出的最后的节点是一个还没有创建的节点。

函数调用信息：

```

int IMFS_evaluate_for_make(
const char          *path, /* IN */
rtems_filesystem_location_info_t *pathloc, /* IN/OUT */
const char          **name /* OUT */
)

```

path 是要分析的文件路径字符串，pathloc 是文件路径信息，name 是

函数功能描述：

1. 根据参数pathloc得到IMFS的节点node；
2. 在一个while循环中，调用IMFS_get_token函数把pathname分解为一个一个的token（token用“/”作为分割符），并解析出pathname中的一个一个目录文件，调用IMFS_find_match_in_dir函数分解掉一个pathname中的token（即进入pathname中的下一级目录），直到最后一个，且最后一个node并不存在；
3. 根据pathname的分解状态，给name赋值；
4. 调用IMFS_Set_handlers函数，根据文件路径信息loc的类型，设置loc的文件处理函数列表。

????!!!! 需要进一步分析

应用举例

见本书带的例子中，位于 `samples/imfs` 目录下的文件，这是一个简单的创建 IMFS 文件，然后写 IMFS 文件，最后读 IMFS 文件内容的例子。此例子包含了一个初始化任务来完成上述工作。首先需要注意 RTEMS 配置中与 IMFS 文件相关的设定：

```
//采用 IMFS 文件系统作为根文件系统
#define CONFIGURE_USE_IMFS_AS_BASE_FILESYSTEM
//设置最大文件描述符为4
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTOR 4
```

如果不配置 `CONFIGURE_USE_IMFS_AS_BASE_FILESYSTEM`，那么缺省情况下将采用 miniIMFS 文件系统作为根文件系统，这样不支持对文件的 `mount`, `link` 系操作。如果不设置最大文件描述符个数为4，则由于缺省的最大文件描述符个数为3（被标准 IO 输入/输出/错误输出占用），导致没有空闲的最大文件描述符可用。

接下来的工作就比较简单了，大致流程是：

1. 用 `creat` 函数创建 IMFS `memfile` 类型的文件；
2. 用 `write` 函数对 `memfile` 类型的文件写字符串；
3. 用 `close` 函数关闭文件；
4. 用 `open` 函数（带参数 `O_RDONLY`，表示只读）打开文件用于读操作；
5. 用 `read` 函数读出 `memfile` 类型的文件中保存的写的字符串，并显示字符串内容；
6. 用 `close` 函数关闭文件；
7. 用 `exit` 函数正常退出。

此程序的运行结果如下所示：

```
BUFSIZ = 1024
Wrote 243
Wrote 243
Wrote 243
Wrote 243
Wrote 243
Wrote 243
Wrote 243
Wrote 243
Wrote 243
Wrote 243
Wrote 243
Wrote 243
Read=60, total=60 : Happy days are here again. Happy days are here again.1Happ
Read=60, total=120 : days are here again.2Happy days are here again.3Happy days
.....
```

DOSFS 文件系统

在 RTEMS 中，DOSFS 文件系统就是指的 FAT 12/16/32文件系统。FAT 是 File Allocation Table 的缩写，最早用于 MS-DOS 和 Windows9x 操作系统下的文件管理系统，包括 FAT12、FAT16和 FAT32（以下统称 FATxx）。在一个磁盘卷上建立 FATxx 文件系统，即将其格式化成 FATxx 格式。FATxx 将磁盘卷分成4个部分：

- boot 记录
- 文件分配表（FAT）
- 根目录
- 数据区

大致布局如下所示：

要重画

| | | | | | |
|-------------|------------------|-------|-------------------|-------------|-----------------------------|
| Boot Sector | Reserved Sectors | FAT 1 | FAT 2 (Duplicate) | Root Folder | Other Folders and All Files |
|-------------|------------------|-------|-------------------|-------------|-----------------------------|

其中 boot 记录区记录着在系统里定义该磁盘卷和该卷其他三个部分的信息，如果该卷是 bootable 的，也包含启动程序。Boot 记录区一定在磁盘卷的第一个扇区（FAT32是前3个扇区）。文件分配表是一张表；FATxx 系统分配磁盘资源的最小单位是簇（cluster），分配空间时，FATxx 以簇为单位分配空间，每个簇大小一样。文件分配表给每个分配簇建立表项，表项内容是邻接的下一个簇的地址。也就是说，簇是按链表组织的。文件分配表区紧跟着 boot 记录区，存在两个相同的文件分配表（FAT1和 FAT2），FAT2平时不使用，用于当 FAT1出现错误或损坏的情况下恢复数据。根目录区记录着该磁盘卷的根目录信息，紧跟着文件分配表区；该区后是数据区，被划分成簇，记录数据并由文件分配表管理。

关键数据结构

RTEMS 的 DOSFS 文件系统代码主要在 retms/cpukit/libfs/src/dosfs 目录下。此目录下有5个头文件，分别是：

- dosfs.h：定义了 DOSFS 文件系统面向上层应用的接口函数。
- msdos.h：定义了 DOSFS 文件系统管理的常量和接口函数。
- fat.h：定义了 FATxx 文件系统和块设备交互的接口函数。
- fat_file.h 和 Fat_fat_operation.h：提供面向 FATxx 文件操作的接口函数，该层也是 msdos.h 和 fat.h 的中间层。其中 fat_file.h 提供常用的文件访问接口函数，Fat_fat_operation.h 则提供给文件分配空间的接口函数。

根据 FAT12、FAT16和 FAT32的不同特点（簇大小不同，FAT 表大小不同等），RTEMS 在 fat.h 里定义了一系列常量来说明，并定义了如下结构：

- fat_vol_s：描述 FAT 系统所在的磁盘卷，包含 boot 记录区（boot 扇区）和 BPB 区
- fat_cache_s：定义 FAT 文件系统的缓存
- fat_fs_info_s：定义 FAT 文件系统的系统级结构和数据
- fat_auxiliary_s：如果查找时所要的数据不在文件占有的簇中的第一个簇的话，需要知道簇号（cluster number）和簇内偏移量（offset），这些信息保存在这个结构里

DOSFS 文件系统信息描述

```
typedef struct msdos_fs_info_s
{
    fat_fs_info_t          fat;          //卷描述符
    rtems_filesystem_file_handlers_r *directory_handlers; //目录操作函数列表
    rtems_filesystem_file_handlers_r *file_handlers;    //文件操作函数列表
    rtems_id                vol_sema;    //用于卷访问的一个信号量
    uint8_t                 *cl_buf;     //一个占位符
};
```

msdos_fs_info_t 结构中给出了 DOSFS 文件系统的信息，其中 fat_fs_info_t 中保存了在 fat 层面上的文件系统的信息，directory_handlers 中保存了 DOSFS 文件系统中对目录进行操作的函数指针列表，而 file_handlers 中保存了 DOSFS 文件系统中对普通文件进行操作的函数指针列表。

另外，在 msdos.h 中还给出了以下的声明：

```
extern rtems_filesystem_file_handlers_r msdos_dir_handlers;
extern rtems_filesystem_file_handlers_r msdos_file_handlers;
```

msdos_dir_handlers 是 DOSFS 文件系统中用于处理目录的函数指针列表，而 msdos_file_handlers 是用于处理普通文件的函数的指针列表。在 DOSFS 文件系统初始化的时候，msdos_fs_info_t 结构中的 directory_handlers 和 file_handlers 就分别指向这两个结构。

DOSFS 目录文件管理函数表

DOSFS 目录文件管理函数表是一个全局变量 msdos_dir_handlers。msdos_dir_handlers 的定义在 msdos_handlers_dir.c 中给出，如下：

```
rtems_filesystem_file_handlers_r msdos_dir_handlers = {
    msdos_dir_open,
    msdos_dir_close,
    msdos_dir_read,
    NULL,           /* msdos_dir_write */
    NULL,           /* msdos_dir_ioctl */
    msdos_dir_lseek,
    msdos_dir_stat,
    NULL,
    NULL,           /* msdos_dir_ftruncate */
    NULL,
    msdos_dir_sync,
    msdos_dir_sync,
    NULL,           /* msdos_dir_fcntl */
    msdos_dir_rmdir
};
```

对比 rtems_filesystem_file_handlers_r 的结构可以看出，msdos_dir_open、msdos_dir_close、msdos_dir_read 函数分别实现了实际文件系统中目录文件的 open、close 和 read 操作。

DOSFS 一般文件管理函数表

DOSFS 一般文件管理函数表是一个全局变量 msdos_file_handlers。msdos_file_handlers 的定义在 msdos_handlers_file.c 中给出，如下：

```
rtems_filesystem_file_handlers_r msdos_file_handlers = {
    msdos_file_open,
    msdos_file_close,
    msdos_file_read,
    msdos_file_write,
    msdos_file_ioctl,
    msdos_file_lseek,
    msdos_file_stat,
    NULL,
    msdos_file_ftruncate,
    NULL,
    msdos_file_sync,
    msdos_file_datasync,
    NULL,           /* msdos_file_fcntl */
    msdos_file_rmdir
};
```

对比 rtems_filesystem_file_handlers_r 的结构可以看出，msdos_file_open、msdos_file_close、msdos_file_read 和 msdos_file_write 函数分别实现了实际文件系统中普通文件的 open、close、read 和

write 操作。

DOSFS 文件系统管理函数表

另外，在 `rtems/cpukit/libfs/src/dosfs/msdos_init.c` 中给出了如下声明：

```
rtems_filesystem_operations_table msdos_ops = {
    msdos_eval_path,
    msdos_eval4make,
#ifdef 0
    NULL,                /* msdos_link */
#else
    msdos_file_link,      /* msdos_link (pseudo-functionality) */
#endif
    msdos_file_rmnod,
    msdos_node_type,
    msdos_mknod,
    NULL,                /* msdos_chown */
    msdos_free_node_info,
    NULL,
    msdos_initialize,
    NULL,
    msdos_shut_down,      /* msdos_shut_down */
    NULL,                /* msdos_utime */
    NULL,
    NULL,
    NULL
};
```

`msdos_ops` 是文件操作的另一个重要的结构，对比 `rtems_filesystem_operations_table` 的结构可以看出其中定义了文件系统操作的实际函数指针，如 `msdos_eval_path` 是 `rtems_filesystem_operations_table` 中成员 `evalpath_h` 对应的解析路径函数，在虚拟文件系统的接口 `rtems_filesystem_evaluate_path` 函数中得到实际调用。`msdos_initialize` 是成员 `fsmount_me_h` 对应的文件系统初始化函数，在虚拟文件系统接口 `mount` 函数中得到调用。`msdos_shut_down` 函数是成员 `fsunmount_me_h` 对应的文件系统卸载函数，在虚拟文件系统接口 `umount` 函数中得到调用。

综上所述，DOSFS 文件系统主要通过利用 `fat` 中的数据结构和各个操作函数实现 `msdos_dir_handlers`、`msdos_file_handlers`、`msdos_ops` 列表中的函数，实现了 `msdos` 实际文件系统的各个操作，并通过各个数据结构将操作函数的指针传给虚拟文件系统，从而实现 RTEMS 中虚拟文件系统的正常工作。

fat 相关数据结构

DOSFS 文件系统使用了 `fat`（包括 `fat12`、`fat16` 和 `fat32`）的支持，实现了虚拟文件系统所需的各个函数，实现了和虚拟文件系统的交互。`rtems/cpukit/libfs/src/dosfs/fat.h` 中定义了与 `fat` 相关信息的常数和数据结构。

`fat_vol_t` 结构体为 `fat` 卷的描述结构，其描述如下：

```
typedef struct fat_vol_s
{
    uint16_t    bps;                /* bytes per sector */
    uint8_t     sec_log2;           /* log2 of bps */
    uint8_t     sec_mul;            /* log2 of 512bts sectors number per sector */
};
```

```

uint8_t      spc;           /* sectors per cluster */
uint8_t      spc_log2;     /* log2 of spc */
uint16_t     bpc;          /* bytes per cluster */
uint8_t      bpc_log2;     /* log2 of bytes per cluster */
uint8_t      fats;         /* number of FATs */
uint8_t      type;         /* FAT type */
uint32_t     mask;
uint32_t     eoc_val;
uint16_t     fat_loc;      /* FAT start */
uint32_t     fat_length;   /* sectors per FAT */
uint32_t     rdir_loc;     /* root directory start */
uint16_t     rdir_entrs;   /* files per root directory */
uint32_t     rdir_secs;    /* sectors per root directory */
uint32_t     rdir_size;    /* root directory size in bytes */
uint32_t     tot_secs;     /* total count of sectors */
uint32_t     data_fsec;    /* first data sector */
uint32_t     data_cls;     /* count of data clusters */
uint32_t     rdir_cl;      /* first cluster of the root directory */
uint16_t     info_sec;     /* FSInfo Sector Structure location */
uint32_t     free_cls;     /* last known free clusters count */
uint32_t     next_cl;      /* next free cluster number */
uint8_t      mirror;       /* mirroring enable/disable */
uint32_t     afat_loc;     /* active FAT location */
uint8_t      afat;         /* the number of active FAT */
dev_t        dev;          /* device ID */
disk_device  *dd;          /* disk device (see libblock) */
void         *private_data; /* reserved */
} fat_vol_t;

```

其中的成员是使用 fat 文件系统的卷中的相关信息，如 bps 是每个扇区中的字节数，spc 是每个簇中的扇区数，等等。这些信息在文件操作函数中也经常用到。

fat_fs_info_t 结构体中则是关于 fat 文件系统的信息，描述如下：

```

typedef struct fat_fs_info_s
{
    fat_vol_t      vol;           /* volume descriptor */
    Chain_Control  *vhash;        /* "vhash" of fat-file descriptors */
    Chain_Control  *rhash;        /* "rhash" of fat-file descriptors */
    char           *uino;         /* array of unique ino numbers */
    uint32_t       index;
    uint32_t       uino_pool_size; /* size */
    uint32_t       uino_base;
    fat_cache_t    c;             /* cache */
    uint8_t        *sec_buf; /* just placeholder for anything */
} fat_fs_info_t;

```

其中 vhash 和 rhash 是为了使用哈希方法而设置的成员，vhash 是保存合法的文件描述符（已经打开的文件）的哈希表，而 rhash 是指保存“remove-but-still-open”文件的哈希表。“remove-but-still-open”文件是指执行了 msdos_file_open 函数（这表明打开了该文件），然后执行了 msdos_file_rmnod 函数或 msdos_dir_rmnod 函数（这表明删除了此文件），但没有执行 fat_file_close 函数（这表明还没有关闭文件，即相关资源还没有释放），的文件。二者都是在 fat_init_volume_info 函数中被分配空间。

fat_file_fd_t 是 fat 文件的文件描述结构，相关内容在 rtems/cpukit/libfs/src/dosfs/fat_file.h 中。

具体描述如下：

```
typedef struct fat_file_fd_s
{
    Chain_Node    link;        /*
                                * fat-file descriptors organized into hash;
                                * collision lists are handled via link
                                * field
                                */
    uint32_t      links_num; /*
                                * the number of fat_file_open call on
                                * this fat-file
                                */
    uint32_t      ino;         /* inode, file serial number :)))) */
    fat_file_type_t fat_file_type;
    uint32_t      size_limit;
    uint32_t      fat_file_size; /* length */
    uint32_t      info_cln;
    uint32_t      cln;
    uint16_t      info_ofs;
    unsigned char first_char;
    uint8_t       flags;
    fat_file_map_t map;
    time_t        mtime;
} fat_file_fd_t;
```

其中如 links_num 记录了此文件打开了几次，即文件的引用数，在文件处理中需要考虑，如在关闭文件时如果大于1则不用真正关闭。 fat_file_type 表明文件是 fat12还是 fat16还是 fat32类型。

fat.h 还提供了一系列 FATxx 系统和磁盘在物理层面打交道的接口，包括簇与扇区对应关系的接口：

```
fat_cluster_num_to_sector_num
fat_cluster_num_to_sector512_num
```

FATxx 系统和它的缓存（fat cache）打交道的接口：

```
fat_buf_access
fat_buf_release
fat_buf_mark_modified
```

文件系统在物理层面上的 io 是以块（block）为基本单位的，在逻辑层面上的 io 是以记录（record）为单位的；一个块包含多个记录，使文件系统的读写速度得以保证。fat.h 提供块级别的 io 操作函数：

```
_fat_block_read
_fat_block_write
```

因为是实现 FATxx 文件系统，所以 fat.h 也提供簇级别的 io 操作函数（一次读写整个簇）

```
fat_cluster_read
fat_cluster_write
```

以下接口涉及到 inode 的概念。文件系统每个文件与目录（特殊的文件）由唯一的 inode 来描述，inode 保存着文件的相关信息。涉及 inode 的接口有：

```
fat_get_unique_ino
fat_ino_is_unique
```

```
fat_free_unique_ino
```

其他接口还有：

```
fat_shutdown_drive  
fat_fat32_update_fsinfo_sector
```

以上接口的实现在 fat.c 里，这里就不详述了。现在来看看高一层的文件管理接口。首先是如何给文件分配空间。Fat_fat_operation.h 为该类操作提供了如下接口：

```
fat_get_fat_cluster  
fat_set_fat_cluster  
fat_scan_fat_for_free_clusters  
fat_free_fat_clusters_chain
```

以上接口在 Fat_fat_operation.c 里实现。fat_file.h 提供了文件访问操作。由于文件的创建和存储用到了 hash 表，所以要有一个将簇号和簇内偏移量映射到文件分配表项的 hash 函数：

```
* fat_construct_key --  
*   Construct key for hash access: convert (cluster num, offset) to  
*   (sector512 num, new offset) and then construct key as  
*   key = (sector512 num) << 4 | (new offset)  
*  
* PARAMETERS:  
*   cl      - 簇号  
*   ofs     - 簇内偏移量  
*   mt_entry - 挂载表项  
*  
* RETURNS:  
*   constructed key  
*/  
static inline uint32_t  
fat_construct_key(  
    rtems_filesystem_mount_table_entry_t *mt_entry,  
    uint32_t                               cl,  
    uint32_t                               ofs)  
{  
    return ( ((fat_cluster_num_to_sector512_num(mt_entry, cl) +  
              (ofs >> FAT_SECTOR512_BITS)) << 4) +  
              ((ofs >> 5) & (FAT_DIRENTRIES_PER_SEC512 - 1)) );  
}
```

以下是主要的文件访问操作：

```
fat_file_open  
fat_file_close  
fat_file_read  
fat_file_write
```

这些接口隐藏了 fat-file 的真实结构，使得 fat-file 看起来就像线性文件一样。

其他接口函数还有：

```
fat_file_reopen
fat_file_extend
fat_file_truncate
fat_file_datasync
fat_file_ioctl
fat_file_size
fat_file_mark_removed
```

RTEMS 的 DOSFS 文件系统面向用户的接口和相关的声明都在 `msdos.h` 里。

关键实现函数

DOSFS 文件系统实现函数很多，这里只列出有代表性的一些函数进行分析，主要涉及文件系统的初始化、挂载、卸载、打开文件、关闭文件、创建文件、读写文件等。

挂载 DOSFS 文件系统

挂载 DOSFS 文件系统工作是在 `fsmount_me_h` 中进行的，它由 `mount` 函数的输入参数 `fs_ops` 指定，并被 `mount` 函数调用。为了分析 DOSFS 文件系统的挂载，还需要分析该函数。RTEMS 实现 DOSFS 文件系统的挂载过程可以归结如下：

```
mount->fsmount_me_h->msdos_initialize->msdos_initialize_support->fatXX
```

通过源代码中 `/cpukit/libfs/src/dosfs/msdos_init.c` 中下面的定义可以发现 DOSFS 文件系统的 `fsmount_me_h` 的实现。将 `msdos_ops` 的定义和前面相关数据结构中 `rtems_filesystem_operations_table` 的结构进行对应可以发现，DOSFS 文件系统的 `fs_mount_me_h` 是用 `msdos_initialize` 这个函数来实现的。所以，下面来具体分析一下 `msdos_initialize` 这个函数。

`msdos_initialize` 函数和上面的结构定义在同一个文件中。它的代码十分简单，仅仅是调用 `msdos_initialize_support` 辅助函数来完成具体的实现。`msdos_initialize_support` 这个辅助函数定义在 `/cpukit/libfs/src/dosfs/msdos_initsupp.c` 中。其参数描述如下：

```
int  msdos_initialize_support(
    rtems_filesystem_mount_table_entry_t *temp_mt_entry,
    rtems_filesystem_operations_table    *op_table,
    rtems_filesystem_file_handlers_r     *file_handlers,
    rtems_filesystem_file_handlers_r     *directory_handlers
)
```

其中，`temp_mt_entry` 是要挂载的文件系统的 mount table entry。`op_table` 是访问文件系统的操作函数列表。`file_handlers` 和 `directory_handlers` 是访问文件系统的文件和目录的函数列表。从 `msdos_initialize_support` 被 `msdos_initialize` 调用时的输入参数可以知道，这些参数被设定为了 `msdos` 相关的处理函数。下面分析一下 `msdos_initialize_support` 函数。这个函数的执行过程是这样的：

1. 调用 `fat_init_volume_info` 获取文件系统的卷信息。
2. 根据输入参数设定 DOSFS 文件信息中的文件和目录处理函数
3. 调用 `fat_file_open` 打开文件系统的根目录
4. 根据上面的信息设定根目录的 FAT 文件描述信息
5. 初始化信号量 `vol_sema`，设定 mount table entry 中的根目录信息

从上面的分析可以知道，最终挂载文件系统的实现是通过最底层的 `fatXX` 函数来实现的。这些函数定义在 `/cpukit/libfs/src/dosfs/fat.c` 和 `fat_file.c` 中。`fatXX` 函数的实现比较繁琐，因为它是根据 FAT12/16/32 规范来实现的，所以这里就不做具体分析了。当挂载成功后，该文件系统对应的 mount table entry 会加入 Mount Table Chain 当中。

卸载 DOSFS 文件系统

和前面的 mount 函数一样，通过 msdos_ops 可以知道，对于 DOSFS 文件系统，它的 fsmount_me_h 的实现是 msdos_shut_down。

msdos_shut_down 函数定义在 /cpukit/libfs/src/dosfs/msdos_fsmount.c 中，其实现是先是调用了 fat_file_close 来关闭根目录。然后调用 fat_shutdown_drive 来释放资源，卸载文件系统。而这些函数都是根据 FAT 规范来实现的。

所以从上面的分析可以知道，最终的实现是使用底层的 fat**函数。umount 的过程可以归结如下：

umount 接口→文件系统 fs_mount_h→msdos_shut_down→fat**函数。

打开 DOSFS 文件

对于 DOSFS 文件系统，open 分为 dir_open 和 file_open。和文件操作相关的函数定义在 msdos_file_handlers 中：

```
rtems_filesystem_file_handlers_r msdos_file_handlers = {
    msdos_file_open,
    msdos_file_close,
    msdos_file_read,
    msdos_file_write,
    msdos_file_ioctl,
    msdos_file_lseek,
    msdos_file_stat,
    NULL,
    msdos_file_ftruncate,
    NULL,
    msdos_file_sync,
    msdos_file_datasync,
    NULL, /* msdos_file_fcntl */
    msdos_file_rmnod
};
```

从上面的定义可以知道，对于 DOSFS 文件系统，它的打开文件的处理函数是 msdos_file_open，因此需要分析一下这个函数。msdos_file_open 函数定义在 /cpukit/libfs/dosfs/msdos_file.c 中。

函数调用信息：

```
int  msdos_file_open(rtems_libio_t *iop, const char *pathname,
                    uint32_t  flag, uint32_t  mode)
```

此函数先获得要打开的文件系统信息和要打开的文件的描述符 fat_fd。然后将要打开的文件的描述符作为输入，调用 fat_file_reopen 函数实现打开文件的操作。此外，为了实现同步互斥，还需要实现对文件系统的信号量的获取和释放。

fat_file_reopen 函数会将前面提到的文件的引用计数 link_num 加一。

从上面的分析可以知道，和前面的函数类似，最终的实现是依赖于底层的 fat**函数。因此，对于 DOSFS 文件系统的 open 操作的过程可以归结如下：

open 接口→文件系统 open_h→msdos_file_open→fat**函数

msdos_dir_open 的函数定义如下：

```
msdos_dir_open(rtems_libio_t *iop, const char *pathname, uint32_t  flag,
               uint32_t  mode)
```

其实现是：

1. 设置局部变量 fs_info=iop->pathinfo.mt_entry->fs_info 和 fat_fd= iop->file_info;
2. 获得和卷相关的那个信号量，保证访问的互斥性；

3. 接着通过文件描述符 `fat_fd` 调用函数 `fat_file_reopen` 打开那个目录，其实只是把 `fat_fd` 里面的 `link` 计数器加1；
4. 设置文件控制块的偏移量为0；
5. 释放信号量；
6. 正常返回。

关闭 DOSFS 文件

对于 DOSFS 文件系统，与 `open` 函数相对应，`close` 函数也有两个，分为 `dir_close` 和 `file_close`。通过 `msdos_file_handlers` 的定义可以知道，关闭文件是由 `msdos_file_close` 来实现的。这个函数和前面的 `msdos_file_open` 定义在同一个文件中：

函数调用信息：

```
int msdos_file_close(rtems_libio_t *iop)
```

函数的输入是文件的控制块。

这个函数的实现流程如下：

- (1) 获取要关闭的文件的描述符和所在文件系统的信息。
- (2) 获取操作的信号量 `vol_sema`, 保证访问的互斥性
- (3) 判断文件是否被删除，如果没有，需要修改文件的大小，时间等数据。
- (4) 调用 `fat_file_close` 关闭文件
- (5) 释放信号量。

此函数与前面的 `msdos_file_open` 函数一样，最终的操作是由底层的 `fat**` 函数来完成的。对于 `fat_file_close` 函数，它会将文件的引用技术 `link_num` 减一，如果 `link_num` 变为0，才会将该文件关闭，释放相应的资源。

从上面的分析可以知道，对于 DOSFS 文件系统，`close` 操作的过程归结如下：

`close` 接口->文件系统 `open_h`->`msdos_file_close`->`fat_file_close`

先看 `msdos_dir_close` 函数。

函数调用信息：

```
msdos_dir_close(rtems_libio_t *iop)
```

这个函数的实现流程如下：

1. 设置局部变量，从传入的 `iop` 中提出信息
2. 获取操作的信号量 `vol_sema`, 保证访问的互斥性
3. 调用 `fat_file_close` 函数来关闭文件
4. 释放信号量
5. 正常结束

在 `msdos_file_close` 函数中，流程也如同 `msdos_dir_close` 中一样，唯一不同的地方在于，如果文件没有被标记为 `removed`，那么，需要调用若干函数（`msdos_set_first_cluster_num`、`msdos_set_file_size`、`msdos_set_dir_wrt_time_and_date`）来同步 `size`、`first cluster number`、`write file time/date` 等内容。

创建 DOSFS 文件

在一个 Parent Directory 下面建立一个新的目录项的过程比较复杂。这需要在内存中建立起新创建节点的目录项的内容，并将其修改成合适的值，以反映这个新节点的类型、大小、时间等信息。这些内容是放在 `new_node` 变量中的。如果新创建节点是一个目录，那么还需要创建这个目录的“.”和“..”目录的目录项，它们将构成这个新创建节点最初的内容。这些内容是放在 `dot_dotdot` 变量中的。还需要从

Parent Directory 里面分配出一个新的目录项所需空间（这个指的是在磁盘上的空间），并且把 new_node 里面的相应内容写到磁盘空间上去。

当在 open 中创建一个文件时，需要调用 mknod 函数，和前面的函数类似，mknod 函数是靠调用文件系统提供的 mknod_h 来实现的。对于 DOSFS 文件系统，这个函数是 msdos_mknod，而这个函数又会调用 msdos_creat_node 来创建文件的节点。这个函数定义在/cpukit/libfs/src/msdos_create.c 中。

函数调用信息：

```
int  msdos_creat_node(  
    rtems_filesystem_location_info_t  *parent_loc,  
    msdos_node_type_t                  type,  
    char                               *name,  
    mode_t                             mode,  
    const fat_file_fd_t                 *link_fd  
)
```

这个函数的执行过程是这样的：初始化文件结构，分配空间，并将文件结构写入磁盘，如果是创建目录，还需要创建.和..这个两个文件。

在创建文件的过程中，调用了底层的 fat_block_read, fat_file_write 等函数。所以，DOSFS 文件系统的 create 操作可以归结为：

create 接口->open->mknod->文件系统mknod_h->msdos_mknod->msdos_creat_node->fat**函数。

读 DOSFS 文件

read 的实现是靠文件系统提供的 read_h 函数。对于 DOSFS 文件系统，这个函数是 msdos_file_read。函数的定义如下：

```
ssize_t  msdos_file_read(rtems_libio_t *iop, void *buffer, size_t count)
```

此函数的输入是前面 read 函数中获得的文件控制块，缓冲区，和大小。msdos_file_read 函数的实现如下所示：

1. 获取要读的文件所在的文件系统的信息和该文件对应的 FAT 文件信息。
2. 获取进行读操作的信号量 vol_sema
3. 调用 fat_file_read 进行读操作
4. 释放信号量

对于 DOSFS 读操作最终是靠底层的 fat_file_read 来实现的。fat_file_read 是根据 FAT 规范来实现的，它对磁盘进行读操作，并将结果写入 buffer 中。因此，DOSFS 文件系统的 read 操作可以归结如下：

read 接口->文件系统 read_h->msdos_file_read->fat_file_read

写 DOSFS 文件

write 操作的实现依赖于文件系统的 write_h。对于 DOSFS 文件系统，它相应的函数是 msdos_file_write。msdos_file_write 函数定义如下：

```
ssize_t  msdos_file_write(rtems_libio_t *iop, const void *buffer, size_t count)
```

函数的输入是文件的控制块信息，写缓冲和大小。

函数的执行过程如下所示：

1. 获取要写的文件所在文件系统的信息和文件对应的 FAT 文件信息。
2. 获取进行写操作的信号量 vol_sema
3. 调用 fat_file_write 进行文件的写操作
4. 修改写后文件的大小。
5. 释放信号量

对于 DOSFS 文件系统，写操作最终是靠底层的 fat_file_write 来实现的。这个函数按照 FAT 规范对磁盘上的文件进行写操作，并返回实际写的字节数。因此，DOSFS 的 write 操作可以归结如下：

write 接口->文件系统 write_h->msdos_file_write->fat_file_write

fat_file_open 函数

函数调用信息:

```
int
fat_file_open(
    rtems_filesystem_mount_table_entry_t *mt_entry,
    uint32_t                               cln,
    uint32_t                               ofs,
    fat_file_fd_t                          **fat_fd
)
```

函数功能描述:

在此函数中, 首先调用 `fat_construct_key` 根据传入的参数计算出哈希表的键值。首先在 `mt_entry` 的 `fs_info` 的 `vhash` 文件哈希表中查找, 如果查到, 表明文件已经打开, 只需要将文件的引用数增1, 然后将其文件描述在 `fat_fd` 中返回即可。否则, 在 `rhash` 中查找。如果没有查找到, 则在将新的文件描述结构插入 `vhash` 中时, 令两个键值域都为计算出的键值; 否则, 插入 `vhash` 的新的文件描述结构第一个键值域是计算出的键值, 而第二个键值域为唯一的 `ino` 值。

fat_file_reopen 函数

函数调用信息:

```
int
fat_file_reopen(fat_file_fd_t *fat_fd)
```

函数功能描述:

此函数处理打开一个已经被打开的文件的情况, 只需将其文件描述结构的引用数增1即可。

fat_file_close 函数

函数调用信息:

```
int
fat_file_close(
    rtems_filesystem_mount_table_entry_t *mt_entry,
    fat_file_fd_t                       *fat_fd
)
```

函数功能描述:

函数查看文件的引用数, 如果大于1, 则只是将引用数减1即可退出。否则, 如果文件是删除了的 `fat` 文件则将所有被占有的簇释放, 并从 `rhash` 中删除文件描述结构, 如果文件不是删除了的 `fat` 文件, 则根据其第二个键值域是否为唯一的 `ino` 值进行处理。如果是, 则将引用数设为0后即可退出, 将文件描述结构留在哈希表中; 如果不是, 则将文件描述结构占用的内存释放。

我们继续看 `fat_file_close` 函数是怎么关闭文件和目录的:

1. 设置局部变量
2. 如果这个 `fat_fd` 正在被多个线程同时打开, 那么就将计数器减1然后返回
3. 否则, 产生一个 `hash key`
4. 根据文件控制块的 `flags` 判断文件是否被删除了
5. 如果是被删除了, 则利用 `file_truncate` 函数来 `free` 所有的 `clusters`, 从 `rhash` 表中删除节点, 如果这个 `ino` 是唯一的, 那么在 `fat` 文件系统释放此节点, 释放基于 `fat_file_fd_t` 结构体的

- fat_fd; 如果 ino 不是位于的
6. 如果没有被删除, 则进一步判断释放 ino 是唯一的, 如果是, 则把计数器 fat_fd->links_num 清零, 如果不是, 则从 vhash 中删除节点, 释放基于 fat_file_fd_t 结构体的 fat_fd;
 7. 最后调用 fat_buf_release 函数把与此文件系统相关的修改过缓冲 buffer 写回磁盘。

fat_file_read 函数

函数调用信息:

```
ssize_t
fat_file_read(
    rtems_filesystem_mount_table_entry_t *mt_entry,
    fat_file_fd_t                        *fat_fd,
    uint32_t                             start,
    uint32_t                             count,
    uint8_t                              *buf
)
```

函数功能描述:

函数从文件的 start 位置开始, 读出 count 个字节到缓冲 buf 中, 这个函数隐藏了 fat 文件的 FAT 结构, 使之表现得就像线性文件一样。FAT 结构中, 根据块号码索引的 FAT 条目包含文件的下一个块的块号码, 一直到最后一块, 它的 FAT 条目的值为文件结束值。

根据这个结构特点, 在对参数进行检查之后, 函数就调用了 fat_cluster_num_to_sector_num、_fat_block_read、fat_file_lseek 等辅助函数对文件进行读操作, 其具体实现比较琐碎, 这里不再分析。

下面我们再来看 fat_file_read 是怎么工作的, 由于流程比较复杂, 下面直接在代码上进行分析。

```
{
//设置局部变量
    int          rc = RC_OK;
    .....
//现 check 要读的字节数, 如果是0就返回
    if (count == 0)
        return cmplt;
//如果开始读的位置大于这个文件的大小, 那么返回 EOF
    if ( start >= fat_fd->fat_file_size )
        return FAT_EOF;
//如果需要读的字节数大于文件大小或者开始的地方不能满足 count 的需要, 那么就缩小 count
    if ((count > fat_fd->fat_file_size) ||
        (start > fat_fd->fat_file_size - count))
        count = fat_fd->fat_file_size - start;
//如果是根目录并且是 FAT12或者 FAT16格式, 那么根据 cluster 取得 sector 并且根据 start 来算出起始的 sector 位置,
//最后调用 _fat_block_read 来读取数据并返回
    if ((FAT_FD_OF_ROOT_DIR(fat_fd)) &&
        (fs_info->vol.type & (FAT_FAT12 | FAT_FAT16)))
    {
        sec = fat_cluster_num_to_sector_num(mt_entry, fat_fd->cln);
        sec += (start >> fs_info->vol.sec_log2);
        byte = start & (fs_info->vol.bps - 1);
```

```

        ret = _fat_block_read(mt_entry, sec, byte, count, buf);
        if ( ret < 0 )
            return -1;

        return ret;
    }
//否则
//算出起始的 cluster
    cl_start = start >> fs_info->vol.bpc_log2;
    save_ofs = ofs = start & (fs_info->vol.bpc - 1);
//调用 fat_file_lseek 来定位
    rc = fat_file_lseek(mt_entry, fat_fd, cl_start, &cur_cln);
    if (rc != RC_OK)
        return rc;
//用 while 循环来读取数据
    while (count > 0)
    {
//取 count 和大小中小的
        c = MIN(count, (fs_info->vol.bpc - ofs));
//算出 sector
        sec = fat_cluster_num_to_sector_num(mt_entry, cur_cln);
        sec += (ofs >> fs_info->vol.sec_log2);
        byte = ofs & (fs_info->vol.bps - 1);
//调用 _fat_block_read 函数来读取数据
        ret = _fat_block_read(mt_entry, sec, byte, c, buf + cmpltd);
        if ( ret < 0 )
            return -1;
//累加
        count -= c;
        cmpltd += c;
        save_cln = cur_cln;
//重新计算 cluster
        rc = fat_get_fat_cluster(mt_entry, cur_cln, &cur_cln);
        if ( rc != RC_OK )
            return rc;

        ofs = 0;
    }
    fat_fd->map.file_cln = cl_start +
                        ((save_ofs + cmpltd - 1) >> fs_info->vol.bpc_log2);
    fat_fd->map.disk_cln = save_cln;
}

```

fat_file_write 函数

函数调用信息:

```

ssize_t
fat_file_write(
    rtems_filesystem_mount_table_entry_t *mt_entry,
    fat_file_fd_t                        *fat_fd,

```

```

uint32_t          start,
uint32_t          count,
const uint8_t     *buf
)

```

函数功能描述:

函数从 `start` 开始, 将 `count` 字节的数据写到文件中。这个函数的作用和 `fat_file_read` 的作用一样, 隐藏了 fat 文件的 FAT 结构, 使之表现得就像线性文件一样。类似地, 在对参数进行检查 (可能需要使用 `fat_file_extend` 函数对文件进行扩展) 之后, 函数就调用了 `fat_cluster_num_to_sector_num`、`_fat_block_write`、`fat_file_lseek` 等辅助函数对文件进行写操作。

在 `fat_file_write` 函数中, 整个流程和 `fat_file_read` 是一样的, 唯一的不同只是把 `_fat_block_read` 换成了 `_fat_block_write`

应用举例

见本书带的例子中, `samples/ramdisk-dosfs` 目录下的文件, 这是一个简单的在 `ramdisk` 上创建并挂载 DOSFS 文件系统, 然后执行创建 DOSFS 目录和文件, 读写 DOS 文件等工作的例子。此例子包含了一个初始化任务来完成上述工作。首先需要注意 RTEMS 配置中与 DOSFS 文件相关的头文件和设定。相关的头文件的定义如下:

```

#include <rtems/dosfs.h> //包含了 DOSFS 相关的定义
#include <rtems/fsmount.h> //包含了挂载文件系统相关的定义

```

相关配置的设定如下:

```

//采用 IMFS 文件系统作为根文件系统, DOSFS 是挂载在 IMFS 的根目录下
#define CONFIGURE_USE_IMFS_AS_BASE_FILESYSTEM
//设置最大文件描述符为20
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTOR 20

```

完成头文件定义和配置设定后, 就需要定义 DOSFS 文件挂载点:

```

fstab_t fs_table[] = {
{ "/dev/ramdisk0", "/mnt/ramdisk0",
  &msdos_ops, RTEMS_FILESYSTEM_READ_WRITE,
  FSMOUNT_MNT_OK | FSMOUNT_MNTPNT_CRERR | FSMOUNT_MNT_FAILED,
  0
}};

```

这里指定了一个 DOSFS 文件挂载点位于 `/mnt/ramdisk0` 目录上, 此文件系统建立在 `ramdisk` 块设备上。有关 `ramdisk` 的相关分析 “I/O 系统” 一章的 “`ramdisk` 去的实例分析” 一节。完成完上述定义后, 就可以进行相关文件操作了。下面是此例子的大致执行流程:

1. 调用 `msdos_format` 函数格式化 `ramdisk` 块设备;
2. 调用 `rtems_fsmount` 函数把保存在 `ramdisk` 块设备上的 DOSFS 文件系统挂载到 `/mnt/ramdisk0` 目录上, 具有读写权限;
3. 接下来就是调用 `mkdir` 函数创建目录, 调用 `fopen` 函数创建文件, 调用 `fprintf` 函数写文件, 调用 `fclose` 函数关闭文件, 调用 `stat` 函数统计文件, 调用 `fread` 函数读文件, 调用 `opendir` 函数打开目录, 调用 `readdir` 函数读取目录内容;
4. 用 `exit` 函数正常退出。

此例子程序的运行结果如下所示:

```
fsmount: mounting of "/dev/ramdisk0" to "/mnt/ramdisk0" succeeded

create directory /mnt/ramdisk0/test
Write a text to /mnt/ramdisk0/test/test1.txt

Now get the size of /mnt/ramdisk0/test/test1.txt, size: 27

read /mnt/ramdisk0/test/test1.txt:
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Now we create another file /mnt/ramdisk0/test2.txt

Get the size of directory /mnt/ramdisk0/test, size: 512

Now we list /mnt/ramdisk0:
TEST
TEST2.TXT
Now we list /mnt/ramdisk0/test:
.
..
TEST1.TXT

EXECUTIVE SHUTDOWN! Any key to reboot...
```

小节

第十一章 I/O 系统

I/O 基本概念

在计算机系统中，有大量的输入输出设备，其种类繁多，差异大。而且随着技术的发展，新设备也不断地出现。因此，如何管理好这些设备，使资源得以合理的利用，是操作系统的一个主要功能，此功能由操作系统的主要组成部分—I/O 系统完成。I/O 系统要能使这些设备能够运转起来，包括给它们发送控制命令、捕获中断、错误处理等；同时它还需要在这些设备与系统的其他部分之间，提供一个简单、易用的接口，并尽可能地使这个接口适用于所有的设备，即实现与设备的无关性。

I/O 控制器

对于 I/O 硬件（简称外设），操作系统所关心的并不是硬件自身的设计、制造和维护，而是如何来对它进行控制，即该硬件所接受的控制命令、所完成的功能，以及所返回的出错报告。所以在设计操作系统的 I/O 功能的时候，需要从这个角度来理解硬件。CPU 通过 I/O 总线连接 I/O 控制器，从而与 I/O 设备交互。在 PC 机中，I/O 控制器的形式通常是印刷电路卡，可以插入到主板的扩充槽中，而在嵌入式系统中，它是一个 SoC 芯片的一部分或者是利用总线扩展的芯片。对于一个 I/O 设备，操作系统主要就是与其 I/O 控制器打交道。

I/O 控制器在物理上包含三个层次：I/O 端口、I/O 接口和设备控制器。每个连接到 I/O 总线上的设备都有自己的 I/O 地址集，即所谓的 I/O 端口。不同的体系结构对 I/O 端口的地址空间有不同的定义，在 IBM PC 体系结构中，I/O 地址空间一共提供了65536个8位的 I/O 端口。在执行 I/O 端口读写指令时，CPU 使用地址总线选择所请求的 I/O 端口，使用数据总线在 CPU 寄存器和端口之间传送数据。I/O 端口也可以被映射到物理地址空间，如在 ARMM68K 等体系结构中，I/O 地址空间映射为一段特定地址范围的内存空间。这样就将 CPU 和 I/O 设备之间的通信转化为 CPU 和 RAM 之间的通信方式来处理。这样的体系极大的方便了程序员，所以在现代操作系统中得到了广泛的应用。

I/O 接口是处于一组 I/O 端口和对应的设备控制器之间的一种硬件电路。它将 I/O 端口中的值转换成设备所需要的命令和数据；并且检测设备的状态变化，及时将信息传递到 I/O 端口的状态寄存器，供操作

系统使用（发出中断请求）。I/O 接口包括键盘接口、图形接口、磁盘接口、总线鼠标、网络接口、括并口、串口、通用串行总线、PCMCIA 接口和 SCSI 接口等。

I/O 控制器并不是所有 I/O 设备所必须的，只有少数复杂的设备才需要。它负责解释从 I/O 接口接收到的高级命令，并将其以适当的方式发送到 I/O 设备；并且对 I/O 设备发送的消息进行解释并修改 I/O 端口的状态寄存器。典型的设备控制器就是磁盘控制器，它将 CPU 发送过来的读写数据指令转换成底层的磁盘操作。

I/O 设备的控制方式

操作系统对 I/O 设备的控制方式主要与三种：程序循环检测方式(Programmed I/O)、中断驱动方式(Interrupt-driven I/O)、直接内存访问方式(DMA, Direct Memory Access)。

对于程序循环检测方式而言，其控制方式体现在执行过程中通过不断地检测 I/O 设备的当前状态，来控制 I/O 操作。具体而言，在进行 I/O 操作之前，要循环地检测设备是否就绪；在 I/O 操作进行之中，要循环地检测设备是否已完成；在 I/O 操作完成之后，还要把输入的数据保存到内存（输入操作）。从硬件的角度来说，控制 I/O 的所有工作均由 CPU 来完成。所以此方式也称为繁忙等待方式（busy waiting）或轮询方式（polling）。其缺点是在进行 I/O 操作时，一直占用 CPU 时间。

中断驱动方式的基本思路是用户任务通过系统调用函数来发起 I/O 操作。执行系统调用后会阻塞该任务，调度其他的任务使用 CPU。在 I/O 操作完成时，设备向 CPU 发出中断，然后在中断服务例程中做进一步的处理。在中断驱动方式下，数据的每次读写还是通过 CPU 来完成。但是当 I/O 设备在进行数据处理时，CPU 不必等待，可以继续执行其他的任务。

在中断驱动方式下，每次的数据读写还是通过 CPU 来完成。如果每次处理的数据量少（如1个字节），则会导致中断的次数很多，而中断需要额外的系统开销，因此也会浪费一些 CPU 时间。例如：假设打印机的打印速度为100字符/秒，在循环检测方式下，当一个字符被写入打印机的数据寄存器中后，CPU 需要等待10毫秒才能写入下一个字符。而在中断驱动方式下，这10毫秒中的大部分用于运行其他任务，少部分用于系统开销。但如果中断次数较多，浪费的 CPU 时间也不少。

要使用直接内存访问（Direct Memory Access, DMA）的控制方式，首先在硬件上要有一个 DMA 控制器。该控制器可集成在设备控制器中，也可集成在主板上。DMA 控制器可以直接去访问系统总线，它能代替 CPU 指挥 I/O 设备与内存之间的数据传送。DMA 控制器包含了一些寄存器，可被 CPU 来读或写。包括：一个内存地址寄存器、一个字节计数器，以及一个或多个控制寄存器（指明了 I/O 设备的端口地址、数据传送方向、传送单位，以及每一次传送的字节数）。

如果使用直接内存访问（Direct Memory Access, DMA）的控制方式，大致的处理过程如下：首先 CPU 对 DMA 控制器进行编程，告诉它应把什么数据传送到内存的什么地方。并向 I/O 控制器发出命令，让它去外设中读出所需的数据块，保存到内部缓冲区中，并验证数据的正确性；DMA 控制器通过总线向 I/O 控制器发出读操作的信号，并把要写入的内存地址打在总线上；I/O 控制器取出一个字节，按该地址写入内存；I/O 控制器向 DMA 控制器发一个确认信号，DMA 控制器把内存地址加1，把字节计数器减1。若计数器的值大于0，转回重复读取；当处理完毕后（计数器的值等于0），DMA 控制器向 CPU 发出一个中断，告诉它数据传输已完成。

I/O 设备的接口形式

类 unix 操作系统都是基于文件概念来管理 I/O 设备，RTEMS 也采用了这种思路，即把一个具体的设备抽象为一个文件，用访问文件的方式来访问设备。文件是以字符序列构成的信息载体。同理，I/O 设备也可以当作一个文件来处理，这样就衍生出了一个词汇——设备文件。操作系统可以使用访问磁盘上普通文件的方式来访问 I/O 设备，极大的简化了操作。RTEMS 中使用主设备号（major）和次设备号（minor）结合来表示一个设备文件。

I/O 设备接口层次

根据其基本特性，设备文件可分为块和字符等许多类。目前在 RTEMS 中支持的 I/O 设备包括：键盘、鼠标、软盘、硬盘、显示器、串口、时钟、网卡、non-volatile memory、Digital to Analog Converters (DACs)等。操作系统中的 I/O 系统的设计水平决定了设备管理的效率。RTEMS 的 I/O 系统的基本思想是采用分层的结构，把各种设备管理软件组织成一系列的层次。底层与硬件特性相关，它把硬

件和较高层的软件隔离开来；而较高层软件则向用户提供一个友好、清晰、统一的接口。从嵌入式软件的 I/O 处理来看，一般可分为四个层次。其大致结构如下所示：

嵌入式软件的 I/O 处理层次

对于中断服务例程而言，当应用认为需要 I/O 服务时，会调用相应的系统调用函数，该函数又调用相应的设备驱动程序，驱动程序在启动 I/O 操作后被阻塞，直到 I/O 操作完成后，将产生一个中断，然后中断服务例程将接管 CPU，并唤醒被阻塞的驱动程序。中断服务例程可以采用有限的任务间通信的方式，如信号量的 V 原语，唤醒操作等。需要注意的是，中断服务例程不能被阻塞。中断服务例程在进行中断处理时需要执行不少 CPU 指令，如任务上下文的保存和恢复，这些都需要一定的系统开销。

设备驱动程序与具体的设备类型相关，是用来控制 I/O 设备运行的程序。一般由设备生产商提供。每一个 I/O 设备都需要相应的设备驱动程序，而每一个设备驱动程序一般只能处理一种设备类型。因为对于不同的设备来说，它的设备控制器中的寄存器数目各不相同，而且控制命令的类型也不相同。为实现设备独立性，操作系统把各种类型的设备划分为块设备、字符设备等，并为每一类定义了一个标准接口，所有设备驱动程序都必须支持其中之一。这个接口供上层的操作系统中间层使用。设备驱动程序的任务是接收来自于上层的、与设备无关的操作系统中间层的抽象请求，并执行这个请求。例如：抽象的读、写操作、设备初始化操作等。设备驱动程序的通用执行过程如下所示：

1. 检查输入参数是否有效，若无效，返回出错报告；若有效，把抽象参数转换为控制设备所需的具体参数；
2. 检查设备当前是否空闲，若正忙，则加入等待队列，稍后处理；若正闲，则检查硬件的状态是否可以开始运行（可能需要打开设备等初始化工作）；
3. 设备驱动程序向设备控制器发出一连串的命令，即把这些命令写入到控制器的寄存器当中。每发出一条命令后，可能还需要检查一下控制器是否已收到该命令并准备接收下一条命令；
4. 在发出所有的控制命令后，设备驱动程序可以有两种处理流程：
 - ❧ I/O 操作不能马上完成，有延迟，那么驱动程序就阻塞自己，直到操作完成，发生中断，然后在中断服务例程中把它唤醒；
 - ❧ 如果 I/O 操作无须延迟即可完成，那么驱动程序就不必阻塞自己。
5. I/O 操作完成后，驱动程序检查出错情况，若一切正常，返回一些状态信息给调用者，并且可能还要把输入的数据上传给上一层的操作系统中间层。

与设备无关的 I/O 中间层是操作系统内核的一部分，它的基本任务是实现所有设备都需要的一些通用的 I/O 功能，并向应用层的 I/O 处理提供一个统一的接口。**RTEMS 的 I/O 中间层包括块设备缓冲管理层、通用块设备管理层和字符设备 TERMOS 层等。**实现的主要功能：

- ❧ 与设备驱动程序的统一接口
- ❧ 提供与设备无关的数据块大小
- ❧ 缓冲技术
- ❧ 出错报告
- ❧ 独占设备的分配和释放

从普遍性考虑，块设备和字符设备是使用最广泛的。在下面的小节中，我们将基于块设备（磁盘）和字符设备（串口）分别展开讨论。

□□□管理

文件系统一般实现在块设备（例如磁盘）上。这也很自然地联想到 I/O 系统中的块设备管理层主要是给文件系统层服务的。文件系统只需把文件中的字节流转化为一个一个逻辑上的数据块，文件系统不

需要关心自身要放在物理介质的哪一个块上。而块设备要完成对这些数据块的缓冲工作（提高 I/O 效率），逻辑并完成数据块映射到物理介质上的磁盘数据块的转换工作，以及与具体的块设备驱动程序（如 IDE 硬盘驱动程序）打交道的工作。

□体□构

???? 分析块设备管理的层次结构，!!!! 参考” I/O 处理软件层次” 小节的写法。

虽然块设备驱动程序可以一次传输一个单独的数据块，频繁的 I/O 操作会降低系统的效率，因为每次 I/O 操作的耗时都是相当可观的。因此在可能的情况下，内核会把几个块合并，尽量在一次 I/O 操作里完成，这样可以节省很多时间。而且还通过缓冲技术来减少不必要的 I/O 操作。这样极大地提高了对块设备的访问效率。

RTEMS 中，\${RTEMS_ROOT}/cpukit/libblock/blkdev.c 为上层的块设备驱动，通过这一层将实际的 I/O 请求发送到对应的块设备，由其驱动程序来处理。

□□□□冲管理

在 RTEMS 中，块设备是以块为单位进行读取和写入。但 RTEMS 的文件系统对块设备的操作，并非是通过直接访问设备来完成的，而是利用了缓冲的机制。通过在内存中开辟专门的缓冲区，来完成对块设备的读写。读，是先通过 buffer 管理器去读取设备的数据，并将数据读出到内存 buffer 中，然后，通过 buffer 管理器，将数据传递给文件系统，再到用户程序。写，用户程序的数据先写到内存中的 buffer 区，然后通过 buffer 管理器，将数据写入块设备中。这就是整个块设备读写操作的流程。通过这种缓冲机制，文件系统无需直接操作块设备，只需对内存的 buffer 操作，底层与块设备的交互，则通过缓冲管理器和设备驱动来完成。

关键数据结构

缓冲区控制块

```
typedef struct bdbuf_buffer {
    Chain_Node link; /* Link in the lru, mod or free chains */
    struct bdbuf_avl_node {
        signed char cache; /* Cache */
        struct bdbuf_buffer* left; /* Left Child */
        struct bdbuf_buffer* right; /* Right Child */
        signed char bal; /* The balance of the sub-tree */
    } avl;
    dev_t dev; /* device number */
    blkdev_bnum block; /* block number on the device */
    unsigned char *buffer; /* Pointer to the buffer memory area */
    rtems_status_code status; /* Last I/O operation completion status */
    int error; /* If status != RTEMS_SUCCESSFUL, this field contains
                errno value which can be used by user later */
    boolean modified:1; /* =1 if buffer was modified */
    boolean in_progress:1; /* =1 if exchange with disk is in progress;
                            need to wait on semaphore */
    boolean actual:1; /* Buffer contains actual data */
}
```

```

int          use_count; /* Usage counter; incremented when somebody use
                        this buffer; decremented when buffer released
                        without modification or when buffer is flushed
                        by swapout task */

rtems_bdbufpool_id pool; /* Identifier of buffer pool to which this buffer
                        belongs */

CORE_mutex_Control transfer_sema;

                        /* Transfer operation semaphore */
} bdbuf_buffer;

```

在 RTEMS 中，缓冲区根据块的大小划分在不同的缓冲池。不同的缓冲池，维护不同块大小的缓冲区。而在某缓冲池中的缓冲区，是按照 AVL 树的方式进行构建的，以提高缓冲区的查找和释放等操作的速度。每一个缓冲区，将由缓冲区描述符来描述。描述符，记录了缓冲区在 AVL 树种的信息，并保留了左右子树的指针，以维护 AVL 树的结构。同时，描述符记录了缓冲区对应的块设备，以及在该设备中对应的块号。此外，还要记录该缓冲区是否被修改过，缓冲区的起始地址，缓冲区所在的缓冲池的 id 等信息。这些信息，将由缓冲区管理器维护。

缓冲池

```

typedef struct bdbuf_pool{

    bdbuf_buffer *tree;          /* Buffer descriptor lookup AVL tree root */
    Chain_Control free;          /* Free buffers list */
    Chain_Control lru;           /* Last recently used list */
    int          blksize;        /* The size of the blocks (in bytes) */
    int          nblks;          /* Number of blocks in this pool */
    rtems_id     bufget_sema;    /* Buffer obtain counting semaphore */
    void          *mallocd_bufs; /* Pointer to the malloc'd buffer memory,
                                or NULL, if buffer memory provided in
                                buffer configuration */
    bdbuf_buffer *bdbufs;        /* Pointer to table of buffer descriptors
                                allocated for this buffer pool. */
} bdbuf_pool;

```

上面的结构是缓冲池的结构。RTEMS 中，每一缓冲池对应着不同大小块对应的块设备。缓冲池，维护着该池所有的缓冲区，这个结构是一个 AVL 树。*tree 就是指向 AVL 树根节点的指针。同时，pool 维护着 free 和 lru 的链表，记录的是空闲缓冲区和最近使用的缓冲区。而 bdbufs 指针，就是该池所有的 buffers 数组。另外，pool 还需要维护该池对应的块大小，块的总数目，以及缓冲池的内存起始地址。所以，RTEMS 对具有不同块大小的设备分配不同的缓冲池中的缓冲区予以使用。所以，RTEMS 中可能会维护着可能好几个缓冲池，以对应不同块大小的块设备。

缓冲上下文

```

struct bdbuf_context {

    bdbuf_pool    *pool;          /* Table of buffer pools */

```

```

int            npools;        /* Number of entries in pool table */
Chain_Control  mod;          /* Modified buffers list */
rtems_id      flush_sema;    /* Buffer flush semaphore; counting
                               semaphore; incremented when buffer
                               flushed to the disk; decremented when
                               buffer modified */

rtems_id      swapout_task; /* Swapout task ID */
};

extern struct bdbuf_context rtems_bdbuf_ctx;

```

所有的缓冲池将由数据结构缓冲上下文来管理。这个结构维护着系统中所有的缓冲池，及池对应的一些性质。可以看到，这个结构包含了缓冲池的数组，说明它维护的是整个系统的缓冲区数据。同时，context 还维护了所有修改过的 buffer 链表 mod，在同步缓冲和磁盘时，缓冲管理器将根据 mod 信息数据重新写入设备。rtems_bdbuf_ctx 就是全局的缓冲上下文对象，用于管理系统的缓冲池和缓冲区。

```

typedef struct rtems_bdbuf_config {
    int        size;          /* Size of block */
    int        num;           /* Number of blocks of appropriate size */
    unsigned char *mem_area;

                               /* Pointer to the blocks location or NULL, in this
                               case memory for blocks will be allocated by
                               Buffering Layer with the help of RTEMS partition
                               manager */
} rtems_bdbuf_config;

extern rtems_bdbuf_config rtems_bdbuf_configuration[];

```

rtems_bdbuf_cmfis 结构用于管理缓冲池创建。这个对象包含了缓冲池对应的块大小，对应的块数目，以及缓冲池的起始内存地址等信息。rtems_bdbuf_configuration[] 是一个全局的数组，维护了所有缓冲池创建时需要的信息。在进行缓冲池初始化时，这个数据将作为参数传入，系统将根据参数中指定的 block 大小和数目，进行内存空间的分配，以及初始化相应缓冲池中的缓冲区结构。

关键实现函数

下面，我们将看看具体实现缓冲机制的一些函数。但由于涉及的函数比较多，我们将重点介绍几个关键函数的实现。比如 initial, read 和 write 等关键操作。Bdbuf 的初始化，需要在内存中申请对应的空间。

初始化 Bdbuf 缓冲池

此函数的来龙去脉如下：

```

驱动程序初始化 -> ata_initialize      ->
                  -> rtems_disk_io_initialize ->
                      rtems_disk_io_initialize->rtems_bdbuf_init

```

可以看出 bdbuf 初始化是在块设备初始化过程中调用的。通过对全局变量 disk_io_initialized 的控制，

此函数只需被执行一次。

函数调用信息:

```
rtems_status_code rtems_bdbuf_init(rtems_bdbuf_config *conf_table, int size)
```

输入参数为 Bdbuf 缓冲池的初始化配置结构 `rtems_bdbuf_config` 和缓冲池大小。

函数执行流程:

1. 通过 `calloc` 函数进行内存的分配, 获取相应的空间;
2. 将缓冲区链表初始化为空;
3. 根据全局的 `rtems_bdbuf_config` 对象, 对所有的缓冲池进行初始化, 这个初始化是由函数 `bdbuf_initialize_pool()` 来完成的。`bdbuf_initialize_pool()` 是内部函数, 作用是初始化一个 buffer 池。基本上整个函数的流程就是给 buffer 池的数据结构 `bdbuf_pool` 中的各个成员函数分配空间, 初始化各个值, 并且对 buffer 池中的各个块进行初始化, 并加入到 `free` 队列中。最后创建一个读取 buffer 块时使用的互斥锁。
4. 创建一个信号量, 用于保证缓冲区的原语操作;
5. 系统将会创建和运行一个 task, 这个任务是 `task_swapout_task`, 它完成的工作是将初始化数据写入到块设备。

由□□□□取□□□缓冲区

函数调用信息:

```
rtems_status_code rtems_bdbuf_get(dev_t device, blkdev_bnum block, bdbuf_buffer **bd)
```

输入参数为块设备描述符 `device`, 起始块号 `block` 和缓冲区指针 `bd`。

函数执行流程:

1. 先要从设备列表中, 获取设备对应的设备结构, 根据设备结构对应的数据, 设置块的起始位置;
2. 然后, 通过函数 `find_or_assign_buffer()` 获得缓冲区。`find_or_assign_buffer()` 函数将根据传入的设备和设备块起始号码来查找当前的 `modified` 和 `lru` 链表中是否有该块对应的 buffer, 如果有, 则返回对应的 buffer 描述符或者是 buffer 的起始位置。如果没有对应块的 buffer 在 `modified` 和 `lru` 链表中, 则会向系统申请缓冲区, 如果当前的缓冲没有可用的缓冲区, 这个操作原语将会被阻塞, 直到有空闲的 buffer 为止, 然后返回获得的 buffer 的地址或者描述符。如果是已经 `cached` 的 buffer, 设备将不允许进行对 buffer 的写操作。

□□□的□操作

函数调用信息:

```
rtems_status_code rtems_bdbuf_read( dev_t device, blkdev_bnum block, bdbuf_buffer **bd)
```

输入参数为块设备描述符 `device`, 起始块号 `block` 和缓冲区指针 `bd`。

函数功能描述:

与 `rtems_bdbuf_get` 函数类似, 先判断请求设备的合法性。然后根据请求设备的参数, 获取对应缓冲区, 如果该块对应的缓冲区已经存在则不用新分配, 否则需要从内存中分配空闲的缓冲区。如果没有空闲的缓冲区可以用, 则会阻塞。区别的一点, 是 `rtems_bdbuf_read` 函数将具体操作从设备往缓冲区的写过程。可见, 在执行设备输入操作时, 在进行了类似 `rtems_bdbuf_get` 函数的操作之后, 将会把数据通过 `ioctl` 的原语操作, 将数据写入缓冲。REQ 请求中包含了缓冲的指针, 所以设备将会把数据写入到

该缓冲指向的 buffer 中。而该指向的 buffer，可以是已经存在的该块的缓冲，也可以是刚申请的缓冲。这里的一个关键函数是 `ioctl()`。这个函数是控制设备读写操作的。

□□□的写操作

缓冲区的写操作将由 `rtems` 上层的函数通过 `memcpy` 来进行写入。而缓冲区到磁盘的同步，则是通过缓冲区管理器来完成的。缓冲区管理器，会根据修改的缓冲区，与设备达到同步。而具体的写入磁盘操作，将由 `bdbuf_swapout_task` 函数来完成。

函数调用信息：

```
static rtems_task bdbuf_swapout_task(rtems_task_argument unused)
```

函数功能描述：

函数执行流程：

1. 首先声明回写操作需要的数据结构和信息，在后面的执行回写操作时，这些信息都会被用到；
2. 然后初始化请求的相关参数，以进行对磁盘设备的回写操作，这里的设置参数包括请求类型、完成状态、请求的数目、传递给处理函数的参数等信息；
3. 接下来获取对应的缓冲区，如果 buffer 是上次循环的 buffer 则直接进行下面的处理，否则，将会重新获得对应的 buffer，这个 buffer 的获得是从 `modified` 态的 buffer 中进行获取的，获取之后，会将对应 buffer 的修改态置为 `false`，表示将把这个 buffer 的数据写回设备，缓冲的数据将是和 `disk` 设备同步的；
4. 然后完成对设备的回写操作；
5. 最后将对执行完 `swap` 操作后的缓冲区进行处理，将其放置到适当的链表中。如果其 `modified` 位是 `true`，则会将其添加到 `modified` 链表，否则，将被添加到 `lru` 链表，以备其他设备或者后续的使用。

释放 buffer 块

函数调用描述

```
rtems_status_code rtems_bdbuf_release(bdbuf_buffer *bd_buf)
static rtems_status_code bdbuf_release(bdbuf_buffer *bd_buf)
rtems_status_code rtems_bdbuf_release_modified(bdbuf_buffer *bd_buf)
```

输入参数为需要释放的 buffer

函数功能描述

`rtems_bdbuf_release()` 函数就是 `bdbuf_release` 的封装，屏蔽了抢占式调度之后就调用 `bdbuf_release()` 函数来执行释放的操作。`bdbuf_release()` 函数首先判断将 buffer 块的 `use_count` 减1，然后判断是否减到0，没有则直接结束。否则判断块是否被修改过，若被修改过，则将该块加到被写入磁盘的列表中，并让做 `flush` 操作的线程恢复工作，否则直接加到 `LRU` 列表中。`rtems_bdbuf_release_modified()` 和 `rtems_bdbuf_release()` 函数类似，不过这个假定要 `release()` 的块是被修改过的，和 `rtems_bdbuf_release()` 相比也就是强制将 buffer 块修改位置1，并且原来没有修改过的话，调用 `bdbuf_initialize_transfer_sema()` 给这个 buffer 块初始化传输信号互斥锁。然后调用 `bdbuf_release()` 函数。

同步缓冲区

函数调用信息

```
rtems_status_code rtems_bdbuf_sync(bdbuf_buffer *bd_buf)
rtems_status_code rtems_bdbuf_syncdev(dev_t dev)
```

函数功能描述

`rtems_bdbuf_sync()` 这个函数和 `rtems_bdbuf_release_modified()` 也很类似，这个函数的作用是强制将一块 buffer 块的内容写回磁盘，实现目标和办法和 `rtems_bdbuf_release_modified()` 完全一样，唯一的不同是这个函数等到写入完成后才返回，这个等待是通过那个传输信号互斥锁来实现的。而和 `rtems_bdbuf_sync()` 强制将一块 buffer 块的内容写回磁盘不同，`rtems_bdbuf_syncdev()` 这个函数是强制将一个设备的所有 buffer 写回磁盘。整个实现的思路是，不断调用 `avl_search_for_sync()` 来在 AVL 树中找被修改过的 buffer 块，写回到磁盘中。

AVL 树相关的函数

在 `bdbuf.c` 中自己实现了 AVL 树数据结构的相关操作，相关的函数为：

- `avl_search` 函数：搜索指定设备指定块号的 buffer 块
- `avl_search_for_sync` 函数：搜索指定设备中第一个被修改过的 buffer 块
- `avl_insert` 函数：插入一个 buffer 块
- `avl_remove` 函数：移除一个 buffer 块

由于 AVL 树的实现都是很经典的数据结构问题，这里不再进行分析。

总的来说，缓冲机制在 RTEMS 中，是通过缓冲区描述符和缓冲池来管理的。缓冲区描述符记录的是缓冲区对应的块和设备，而缓冲池，则记录了池中所有的缓冲区数据，以及缓冲区对应块大小等信息。最后，系统用缓冲区上下文这一数据结构来实现对缓冲池和缓冲区的管理。因为不同的缓冲池维护的是不同块大小的缓冲区，所以需要缓冲区上下文结构来维护所有的缓冲区池。而缓冲区，就是文件系统和块设备之间的通信和数据交换的桥梁。

缓冲区管理

`$(RTEMS_ROOT)/cpukit/sapi/` 中包含初始化和中止 I/O 服务的代码。

`$(RTEMS_ROOT)/cpukit/libblock/` 中包含对使用硬盘、软盘、光驱类块设备的支持，包含基本的通用 I/O 代码，对磁盘缓存的支持，以及 RAMdisk（把 RAM 仿真成 disk 的设备）的驱动程序。

关键实现函数

初始化 I/O 管理

函数调用描述：

```
rtems_status_code rtems_io_initialize(
    rtems_device_major_number major,
    rtems_device_minor_number minor,
    void                    *argument
)
```

输入参数为设备的主从设备号和附加参数

函数功能描述：

函数执行流程：

1. 首先分配足够的内存空间供设备驱动使用；
2. 然后完成设备驱动的静态注册，如果需要动态增加或动态删除设备驱动则需要调用

rtems_io_register_driver 和 rtems_io_unregister_driver 来实现;

3. 最后分配足够的内存空间供设备描述符使用。

□□□的通用□□程序

RTEMS 中包含一个块设备的通用驱动程序，其实现位于\${RTEMS_ROOT}/cpukit/libblock/blkdev.c

关键数据结构

I/O 请求数据结构

```
typedef struct blkdev_request {
    blkdev_request_op req;      /* Block device operation (read or write) */
    blkdev_request_cb req_done; /* Callback function */
    void            *done_arg; /* Argument to be passed to callback function*/
    rtems_status_code status; /* Last I/O operation completion status */
    int             error;    /* If status != RTEMS_SUCCESSFUL, this field
                               * contains error code
                               */
    blkdev_bnum     start;    /* Start block number */
    uint32_t        count;    /* Number of blocks to be exchanged */
    uint32_t        bufnum;   /* Number of buffers provided */
    blkdev_sg_buffer bufs[0]; /* List of scatter/gather buffers */
} blkdev_request;
```

该数据结构描述了一个读写操作的请求细节。包含操作类型 req（读或写）、回调函数 req_done 以及回调函数函数参数*done_arg、请求完成状态 status、错误码 error、起始块号 start、请求交换的块总数 count、数据交换缓冲区 bufs[0]以及交换缓冲区的个数 bufnum。

关键实现函数

通用□□□打开操作

函数调用信息:

```
rtems_device_driver rtems_blkdev_generic_open(
    rtems_device_major_number major,
    rtems_device_minor_number minor,
    void                    * arg)
```

输入参数为设备的主从设备号和附加参数

函数功能描述:

函数执行流程:

1. 依据设备的主从设备号组成设备号;

2. 依据设备号搜索设备列表，得到设备描述符；
3. 对指定设备的引用计数器增一。

通用□□□关□操作

函数调用信息：

```
rtems_device_driver rtems_blkdev_generic_close(  
    rtems_device_major_number major,  
    rtems_device_minor_number minor,  
    void                    * arg)
```

输入参数为设备的主从设备号和附加参数

函数功能描述：

函数执行流程：

1. 依据设备的主从设备号组成设备号；
2. 依据设备号搜索设备列表，得到设备描述符；
3. 对指定设备的引用计数器减一。

通用□□□□操作

函数调用信息：

```
rtems_device_driver rtems_blkdev_generic_read(  
    rtems_device_major_number major,  
    rtems_device_minor_number minor,  
    void                    * arg)
```

输入参数为设备的主从设备号和附加参数

函数功能描述：

函数执行流程：

1. 依据设备的主从设备号组成设备号，依据设备号搜索设备列表，得到设备描述符；
2. 从设备描述符中获取设备信息；
3. 对于每一个块，首先将其读出缓冲区，然后将其拷贝到目的地址。

通用□□□□写操作

函数调用信息：

```
rtems_device_driver rtems_blkdev_generic_write(  
    rtems_device_major_number major,  
    rtems_device_minor_number minor,  
    void                    * arg)
```

输入参数为设备的主从设备号和附加参数

函数功能描述：

函数执行流程：

1. 依据设备的主从设备号组成设备号，依据设备号搜索设备列表，得到设备描述符；
2. 从设备描述符中获取设备信息；
3. 对于每一个块，首先将其写入缓冲区，然后将其拷贝到目的地址。

通用 ioctl 控制操作

函数调用信息:

```
rtems_device_driver rtems_blkdev_generic_ioctl(
    rtems_device_major_number major,
    rtems_device_minor_number minor,
    void * arg)
```

输入参数为设备的主从设备号和附加参数

函数功能描述:

依据不同的请求, 该函数可以执行的操作包括获取磁盘设备的块大小、磁盘大小以及同步磁盘设备和对应的缓冲区 (由于使用了缓冲机制), 还可以进行 I/O 的读写操作。

通用底口操作

物理磁盘在系统中被识别为一个个的逻辑设备, 对物理磁盘的操作将通过对逻辑磁盘的操作来完成, 物理磁盘与逻辑磁盘的联系是必需的, RTEMS 中这部分的实现在

`${RTEMS_ROOT}/cpukit/libblock/diskdevs.c`

关键数据结构

磁盘控制口

```
typedef struct disk_device {
    dev_t      dev;           // 设备描述符 (major+minor)
    struct disk_device *phys_dev; // 物理设备 ID
    char       *name;        // 设备名
    int        uses;         // 引用计数器, 非零时不允许删除此设备
    int        start;        // 物理设备上的起始块号,
                                // 如果是逻辑磁盘则可能不是0
    int        size;         // 总块数, 表明逻辑磁盘的大小
    int        block_size;   // 块大小, 以字节表示, 必须是2^n 大小
    int        block_size_log2; // 块大小, 以字节数的二进制位数表示,
                                // 即 log2
    rtems_bdpool_id pool;    // 分配给该设备的缓冲池
    block_device_ioctl ioctl; // 指向该设备的请求处理模块
} disk_device;
```

每一个物理或逻辑磁盘都包含这个数据结构, 每一个磁盘设备的主设备号下面都维护如下一个数据结构, 其中有一个包含所有从设备 `disk_device` 数据结构的链表。

```
struct disk_device_table {
    disk_device **minor; // 用 minor 号索引的磁盘设备表
```

```
int size;          /// 表中的项的数目
};
```

关键实现函数

指定磁口口口

函数调用信息

```
static inline disk_device *get_disk_entry(dev_t dev)
```

输入参数为设备号 dev

函数功能描述:

该函数由其主设备号 Major 和次设备号 Minor (dev_t dev 实际上是这两个设备号的并置), 首先以 Major 为下标在数组中 disktab 找到 disktab[Major] 这一项, 然后在 disktab[Major].minor 数组中以 Minor 为下标查找 disktab[Major].minor[Minor], 即可找到这个设备的控制结构 (disk_device_table *).

搜索磁口口口

函数调用信息

```
disk_device *rtems_disk_lookup(dev_t dev)
```

输入参数为设备号 dev

函数功能描述:

该函数查找并索引一个设备, 也就是说, 在找到了需要的设备的情况下, 要将它的引用计数加一。

磁口口口的口建

函数调用信息

```
static disk_device * create_disk_entry(dev_t dev)
```

输入参数为设备号 dev

函数功能描述:

该函数依据主设备号 Major 和次设备号 Minor 在二级数组结构里面合适的位置创建一个磁盘设备控制块。适当的时候需要重新分配空间供设备使用。

磁口口口控制口的口建

函数调用信息

```
static rtems_status_code create_disk(dev_t dev, char *name, disk_device **diskdev)
```

输入参数为设备号 dev, 设备名 name, 设备控制块 diskdev。

函数功能描述：

首先查找是不是已经存在了这个设备号的磁盘，如果已经存在，则出错。否则，为这个设备分配空间以保存设备名称字符串，然后调用 `create_disk_entry` 函数创建相应的 disk 控制结构，`dd` 指向这个控制结构，最后把 `dd->dev` 设置成这个设备的设备号 `dev`，`dd->name` 设置成这个设备的名称。

□建一个物理磁□

函数调用信息

```
rtems_status_code rtems_disk_create_phys(dev_t dev, int block_size, int disk_size,
                                         block_device_ioctl handler,
                                         char *name)
```

输入参数为设备号 `dev`，磁盘块大小 `block_size`，磁盘大小 `disk_size`，设备控制函数 `handler`，设备名 `name`。

函数功能描述：

这个函数就是首先检查 `block_size` 是不是确实是2的整数次方，并且计算出 `block_size_log2`，然后判断 `block_size` 一定要大于等于512。调用 `rtems_bdbuf_find_pool` 函数为这个设备找一个合适的缓存池（也就是说，这个缓存池的块大小不能小于设备的块大小，在所有满足条件的缓存池中，找一个块大小最小的），调用 `create_disk` 函数为这个设备创建一个 disk 控制块（`dd` 指向这个控制块）并将其纳入到二级数组索引的管理之下，再把 `dd->phys_dev` 指向 `dd` 本身，初始化 `dd` 的引用计数、`start`、`size`、`block_size`、`block_size_log2` 等等，其中最重要的是初始化 `dd->iocctl` 为 `handler`，这是这个物理盘的驱动函数，以及将 `dd->pool` 设置为刚才找到的缓存池编号。

□建一个□□磁□

函数调用信息

```
rtems_status_code rtems_disk_create_log(dev_t dev, dev_t phys, int start, int size, char
*name)
```

输入参数为设备号 `dev`，物理磁盘设备号 `phys`，物理磁盘起始块 `start`，逻辑磁盘大小 `size`，设备名 `name`。

函数功能描述：

创建一个逻辑盘的操作与创建物理盘大体相似，只是需要提供它所属的物理盘设备号以及 `start` 参数，而不用提供其块大小、驱动函数。

RAMdisk □□□例分析

RAM disk 是 RTEMS 系统在内存开辟空间的虚拟磁盘。RAM disk 同样是模拟实际物理磁盘的行为和结构，给文件系统提供一个透明的虚拟磁盘接口。所以，在 RAM disk 中，实际上同样也利用了 buffer 缓冲，即读和写实际上还通过 buffer 缓冲区。因为文件系统把 RAM Disk 当作真的物理磁盘，所以不会提供特别的读写接口来完成操作，而是仍然跟普通物理磁盘一样，通过缓冲区管理器来进行。我们看看具体的实现细节。

关键数据结构

disk 描述符

```
struct ramdisk {
    int    block_size; /* RAM disk block size */
    int    block_num;  /* Number of blocks on this RAM disk */
    void   *area; /* RAM disk memory area */
    rtems_boolean initialized; /* RAM disk is initialized */
    rtems_boolean    malloced; /* != 0, if memory allocated by malloc
                                for this RAM disk */
};

static struct ramdisk *ramdisk;
static int nramdisks;
```

这个描述符，包含了 ram disk 对应的属性。如 disk 的块大小，块数目，是否已经进行了初始化，是否已经通过已经获得内存空间的标志位等信息。系统维护着一个 ram disk 的列表，如上面代码中的 ramdisk 指针变量，同时还有一个维护 ram disk 数目的 int 类型变量 nramdisks，维护当前系统存在的 ram disk 的数目。

Disk 配置结构

```
typedef struct rtems_ramdisk_config {
    int    block_size; /* RAM disk block size */
    int    block_num;  /* Number of blocks on this RAM disk */
    void   *location; /* RAM disk permanent location (out of RTEMS controlled
                        memory), or NULL if RAM disk memory should be allocated
                        dynamically */
} rtems_ramdisk_config;

extern rtems_ramdisk_config rtems_ramdisk_configuration[];
extern size_t rtems_ramdisk_configuration_size;
```

对应每个具体的 ram disk，系统同时对应着一个 disk 配置结构，包含块大小、块数目以及 disk 的起始位置。这个配置结构，将在初始化每个对应的 ram disk 对象时用到。

关键实现函数

初始化 Ram Disk

函数调用信息：

```
rtems_device_driver ramdisk_initialize(
    rtems_device_major_number major,
    rtems_device_minor_number minor,
    void *arg)
```

输入参数为设备的主从设备号和附加参数

函数执行流程：

1. 首先获取全局的 ramdisk 配置表；
2. 然后申请对应的 ram disk 的内存空间。这里分配空间由 ram_disk 决定，该参数在应用程序中指定 rtems_ramdisk_configuration_size。而 ram disk 与 bdbuf 的关系也体现在这里，因为这里的初始化工作，会调用 rtems 的系统函数 rtems_disk_io_initialize() 来进行 I/O 的初始化，rtems_disk_io_initialize() 函数中，将为 ram disk 设备初始化缓冲区，从而将 Ramdisk 和缓冲区联系起来，以后的文件系统对 ram disk 的读写操作都将是直接对缓冲区的操作。这就是 ram disk 与 buffer 的联系起点。
3. 接下来，将针对每一个 ram disk 进行具体的初始化工作；
4. 首先是由文件系统创建新设备 rtems_filesystem_make_dev_t(major, i)，根据主设备号和次设备号来创建 device。所以，使用通用的 POSIX/RTEMSAPI 接口创建设备在文件系统看来是一样的，只是 ram disk 内部的操作不一样而已。
5. 同时将为每一个 ram disk 申请内存空间，根据指定的块大小和块数目申请内存，并且根据申请的结果来设置每一个 ram disk 描述符的 malloced 域。
6. 接下来完成对应 ram disk 扇区的创建和初始化。同时，为 disk 的 I/O 操作指定了 ioctl 函数 handler。这样，以后的操作将由指定的 ioctl 函数来进行。

IOCTL 设备控制函数

函数调用描述：

```
static int ramdisk_ioctl(dev_t dev, uint32_t req, void *argp)
```

输入参数为设备描述符和请求参数

函数功能描述：

这个函数完成 ioctl 方法，通过判断请求的类型，从而调用相应的处理函数。这里设备的获得是通过主从设备号来获取的。读和写操作都是需要判断 ram disk 是否已经进行过初始化，否则不能进行读写。这里，io handler 处理的主要有两类请求，一个是 BLKDEV_REQ_READ，表示对设备的块读操作，另一个是对块的写操作 BLKDEV_REQ_WRITE。而 io handler 则通过调用 ram disk 设备驱动函数 read 和 write 来进行与设备的数据交互，这实际上也是普通意义上的磁盘所进行的操作。

RAM disk 口操作

函数调用描述：

```
static int ramdisk_read(struct ramdisk *rd, blkdev_request *req)
```

输入参数为指定 RAM disk 设备以及请求参数

函数功能描述：

ram disk 的读操作非常简单，由于是对内存的直接操作。所以，这里直接进行内存的复制就可以了。将对应的 ram disk 扇区的数据直接拷贝到对应的 buffer 缓冲区，然后发出设备读结束的信号，这样文件系统就可以从 buffer 中读数据了。

RAM disk 写操作

函数调用描述:

```
static int ramdisk_write(struct ramdisk *rd, blkdev_request *req)
```

输入参数为指定 RAM disk 设备以及请求参数

函数功能描述:

ram disk 的写操作同读操作类似, 由于是面对内存直接操作, 所以只需在 ram disk 和缓冲区之间直接进行拷贝就可以了。

实际上, 由于 RAM disk 模拟的是一个实际的磁盘, 因此 RAMDISK 文件系统提供的访问接口和普通磁盘驱动一样接口, 所以 RAM disk 的数据读写, 仍然是通过缓冲机制进行, 即文件系统对 RAM disk 的读写, 依然是通过 buffer, 所以 RAM disk 与 Bdbuf 的关系, 就和磁盘等块设备与 buffer 的关系一样, 只是数据的操作变成了内存块之间的直接复制拷贝而已。

□用□例

见本书带的例子中, 位于 samples/ramdisk-dosfs 目录下的文件, 这是一个简单的在 ramdisk 上创建并挂载 DOSFS 文件系统, 然后执行创建 DOSFS 目录和文件, 读写 DOS 文件等工作的例子。首先需要注意 RTEMS 配置中与 ramdisk 相关的头文件和设定。相关的头文件的定义如下:

```
#include <rtems/ramdisk.h> //ramdisk 相关定义
#include <rtems/bdbuf.h>  //块设备缓冲管理相关定义
```

相关配置的设定如下:

```
#define CONFIGURE_APPLICATION_NEEDS_LIBBLOCK //需要块设备管理
//设置 ramdisk 的数据块大小和块的个数
rtems_ramdisk_config rtems_ramdisk_configuration[] = {
{
512, 200, (void *) 0
}
};
//只有一个 ramdisk
size_t rtems_ramdisk_configuration_size = 1;
//采用定制的驱动程序表
#define CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE
rtems_driver_address_table Device_drivers[] = {
.....
//指定必须的 ramdisk 块设备驱动程序
RAMDISK_DRIVER_TABLE_ENTRY
};
//当前驱动程序的个数
#define CONFIGURE_NUMBER_OF_DRIVERS \
((sizeof(Device_drivers) / sizeof(rtems_driver_address_table)))
//设定最大驱动程序个数
#define CONFIGURE_MAXIMUM_DRIVERS 10
```

然后完成与 DOSFS 相关的头文件定义和配置。关于此例子程序的执行流程和运行结果的具体分析可参考“文件系统”一章的“DOSFS 文件系统”一节的“应用实例”小节。

当将 ramdisk 格式化成 DOSFS 文件系统, 然后调用 rtems_fsmount 挂载这个文件系统到/mnt/ramdisk0

目录上后，就可以在 ramdisk 块设备进行正常的 DOSFS 文件读写等操作了。

字符设备管理

字符（char）设备是一个以字节流方式访问的设备。由字符设备驱动程序实现对字符设备的访问。大多数字符设备都是一个只能顺序访问的数据通道，例如字符终端（console）、串口和并口等等。高速图形接口设备也被归于字符设备一类。在 RTEMS 中字符设备支持轮询和中断两种访问方式。在所有的字符设备中，字符终端设备一直是一个具有代表性的部分，许多其它的设备驱动程序的设计都受其影响。

设备结构

???? 要分析字符管理的层次结构，!!!! termios 设备包括 RS232串口，slip，ppp 等

TERMIOS 设备

TERMIOS 是 RTEMS 中字符设备驱动程序的一个底层框架，它实现了轮询和中断两种访问字符 I/O 端口的方
式，以及访问物理端口的多种底层操作，使得其他字符设备驱动程序的设计大大简化。它的实现在
\${RTEMS_ROOT}/cpukit/libcsupport/termios.c 和
\${RTEMS_ROOT}/cpukit/libcsupport/termiosinitialize.c。

关键数据结构

字符设备描述块

```
struct rtems_termios_tty {
    /*
     * Linked-list of active TERMIOS devices
     */
    struct rtems_termios_tty *forw;
    struct rtems_termios_tty *back;

    /*
     * How many times has this device been opened
     */
    int refcount;

    /*
     * This device
     */
    rtems_device_major_number major;
    rtems_device_major_number minor;

    /*
     * Mutual-exclusion semaphores
     */
    rtems_id isem;
    rtems_id osem;

    /*
     * The canonical (cooked) character buffer
     */
    char *cbuf;
    int ccount;
}
```

```

int            cindex;

/*
 * Keep track of cursor (printhead) position
 */
int            column;
int            read_start_column;

/*
 * The ioctl settings
 */
struct termios      termios;
rtems_interval      vtimeTicks;

/*
 * Raw input character buffer
 */
struct rtems_termios_rawbuf rawInBuf;
uint32_t           rawInBufSemaphoreOptions;
rtems_interval      rawInBufSemaphoreTimeout;
rtems_interval      rawInBufSemaphoreFirstTimeout;
unsigned int        rawInBufDropped;    /* Statistics */

/*
 * Raw output character buffer
 */
struct rtems_termios_rawbuf rawOutBuf;
int  t_dqlen; /* count of characters dequeued from device */
enum {rob_idle, rob_busy, rob_wait } rawOutBufState;

/*
 * Callbacks to device-specific routines
 */
rtems_termios_callbacks device;
volatile unsigned int  flow_ctrl;
unsigned int           lowwater,highwater;

/*
 * I/O task IDs (for task-driven drivers)
 */
rtems_id              rxTaskId;
rtems_id              txTaskId;
/*
 * line discipline related stuff
 */
int t_line; /* id of line discipline */
void *t_sc; /* hook for discipline-specific data structure */
/*
 * Wakeup callback variables
 */
struct ttywakeup tty_snd;
struct ttywakeup tty_rcv;
int              tty_rcvwakeup;

```

```
};
```

`rtems_termios_tty` 为 TERMIOS 中最重要的数据结构，它完整地描述了一个字符设备。`forw` 和 `back` 说明该设备在 TTY 设备列表中的位置，设备的引用计数器 `refcount`，设备主从设备号 `major` 和 `minor`，输入信号量 `isem` 和输出信号量 `osem`，标准的字符缓冲区 `cbuf`，光标位置 `column` 和 `read_start_column`，输入缓冲 `rawInBuf` 和输出缓冲 `rawOutBuf`，`TERMIOS_TASK_DRIVEN` 模式下输入和输出任务的标记 `rxTaskId` 和 `txTaskId` 等等。

TERMIOS 数据结构

```
struct termios {
    tcflag_t c_iflag;        /* 输入选项标志 */
    tcflag_t c_oflag;        /* 输出选项标志 */
    tcflag_t c_cflag;        /* 控制选项标志 */
    tcflag_t c_lflag;        /* 本地选项标志 */
    cc_t c_line;             /* line discipline */
    cc_t c_cc[NCCS];         /* 控制字符 */
};
```

关键实现函数

初始化 TEEMIOS

函数调用信息

```
void rtems_termios_initialize (void)
```

函数功能描述

为系统的 TTY 设备链表创建一个互斥信号量 `rtems_termios_ttyMutex`。

打开一个 TERMIOS □□

函数调用信息

```
rtems_status_code rtems_termios_open (
    rtems_device_major_number    major,
    rtems_device_minor_number    minor,
    void                         *arg,
    const rtems_termios_callbacks *callbacks
)
```

输入参数为 TTY 设备的主从设备号 `major` 和 `minor`，附加参数的地址 `arg`，以及回调函数的地址 `callbacks`。

函数功能描述：

执行步骤如下：

1. 首先获得互斥信号量 `rtems_termios_ttyMutex`;
2. 利用主从设备号在系统 TTY 设备列表中搜索对应设备，如果没有对应设备则创建该设备，创建一个字符设备的主要工作就是填写 `rtems_termios_tty` 数据结构，如果是 `TERMIOS_TASK_DRIVEN` 模式则要为接受和发送任务创建两个互斥信号量;
3. 如果是第一次打开该设备，则在 `TERMIOS_TASK_DRIVEN` 模式下运行发送 `rtems_termios_txd daemon` 和接收 `rtems_termios_rxd daemon` 两个后台任务。
4. 释放互斥信号量 `rtems_termios_ttyMutex`。

关闭一个 TTY 设备

函数调用信息

```
rtems_status_code rtems_termios_close (void *arg)
```

输入参数为附加参数的地址 arg。

函数功能描述:

执行步骤如下:

1. 首先获得互斥信号量 rtems_termios_ttyMutex;
2. 该设备的引用计数器减一, 若计数器非零则结束处理, 否则关闭该设备;
3. 在有特殊关闭处理函数时执行该函数, 否则仅仅调用 drainOutput 清空输出缓冲区;
4. 在 TTY_TASK_DRIVEN 模式下, 结束发送和接受两个后台任务;
5. 将该设备从 TTY 设备列表中删除;
6. 删除该设备相关的信号量, 释放缓冲区。

从一个 TTY 设备中取字

函数调用信息

```
rtems_status_code rtems_termios_read (void *arg)
```

输入参数为附加参数的地址 arg。

函数功能描述:

执行步骤如下:

1. 首先获得该设备的输入信号量 tty->isem;
2. 在有特殊输入处理函数时执行该函数, 否则执行默认处理;
3. 在 TTY_POLLING 模式下, 调用 fillBufferPoll 将数据读入输入缓冲区, 否则调用 fillBufferQueue 完成数据的读出;
4. 完成输入后, 释放输入信号量 tty->isem。

向一个 TTY 设备中写入字

函数调用信息

```
rtems_status_code rtems_termios_write (void *arg)
```

输入参数为附加参数的地址 arg。

函数功能描述:

执行步骤如下:

1. 首先获得该设备的输出信号量 tty->osem;
2. 在有特殊输出处理函数时执行该函数, 否则执行默认处理;
3. 在含有 OPOST 标志时, 调用 oproc 处理输出, 否则直接调用 rtems_termios_puts 输出数据;
4. 完成输出后, 释放输出信号量 tty->osem。

TTY 设备的 ioctl

函数调用信息

```
rtems_status_code rtems_termios_ioctl (void *arg)
```

输入参数为附加参数的地址 arg。

函数功能描述:

执行步骤如下：

1. 首先获得该设备的输出信号量 `tty->osem`;
2. 依据 `cmd` 分别执行，包含读取和写入设备属性、清空输出队列、设定输入输出唤醒函数、等等，同时还可以通过特殊的 `ioctl` 模块来处理特殊的 `cmd`;
3. 完成 `ioctl` 处理后，释放输出信号量 `tty->osem`。

从 TERMios 口取字的后台程序

函数调用信息

```
static rtems_task rtems_termios_rxdaemon(rtems_task_argument argument)
```

输入参数为 RTEMS 任务参数 `argument`。

函数功能描述：

执行步骤如下：

1. 收到 `TERMios_RX_TERMINATE_EVENT` 事件则终止该任务，否则处理输入；
2. 以轮训的方式每字节读取直到遇到 `EOF` 结束符，并调用 `rtems_termios_enqueue_raw_characters` 将字节读出。

向 TERMios 口写字的后台程序

函数调用信息

```
static rtems_task rtems_termios_txdaemon(rtems_task_argument argument)
```

输入参数为 RTEMS 任务参数 `argument`。

函数功能描述：

执行步骤如下：

1. 收到 `TERMios_TX_TERMINATE_EVENT` 事件则终止该任务，否则处理输出；
2. 调用 `rtems_termios_refill_transmitter` 将数据发送至设备。

设置流控制类型

函数调用信息

```
static void termios_set_flowctrl(struct rtems_termios_tty *tty)
```

输入参数为一个字符设备的描述块 `tty`

函数功能描述：

依据字符设备的描述块中 `flow_ctrl` 信息和 `termios` 结构中的各个选项标志重新设定 `flow_ctrl`。

向设备发送字节

函数调用信息

```
void rtems_termios_puts (  
    const void *_buf, int len, struct rtems_termios_tty *tty)
```

输入参数为字符设备的描述块 `tty`，以及待输出数据 `buf` 和数据长度 `len`。

函数功能描述：

执行步骤如下：

1. 如果设备设定为 `POLL` 模式，则直接使用设定好的 `write` 函数，返回；
2. 将数据拷贝到描述块中定义的缓冲区，这个过程是中断允许的，目的是为了减小延迟；
3. 从描述块中定义的缓冲区将数据按字节顺序输出，这个过程是中断禁止的。

将设备输入读出输入缓冲区

函数调用信息

```
static rtems_status_code fillBufferQueue (struct rtems_termios_tty *tty)
```

输入参数为字符设备的描述块 tty

函数功能描述

。。。。。没看明白

串口□□□□

TTY 设备是典型的串口设备，我们将以 I386平台下的 TTY 驱动程序为例给大家分析一下 RTEMS 中的串口驱动实现。RTEMS 中仅仅支持两个串口 COM1和 COM2，在 console 使用 VGA 模式的情况下，两个串口都可以使用，但是如果 console 使用串口的话，将有一个串口被占用，不能自由使用。TTY 和 console 的驱动程序都是在 TERMIOS 的框架上完成的。

关键数据结构

为两个串口分别实现了两个中断描述数据结构：

```
/* Interrupt structure for tty1 */
static rtems_irq_connect_data tty1_isr_data =
{
    BSP_UART_COM1_IRQ,
    BSP_uart_termios_isr_com1,
    0,
    isr_on,
    isr_off,
    isr_is_on};

/* Interrupt structure for tty2 */
static rtems_irq_connect_data tty2_isr_data =
{
    BSP_UART_COM2_IRQ,
    BSP_uart_termios_isr_com2,
    0,
    isr_on,
    isr_off,
    isr_is_on};
```

其具体信息可参考中断处理一章

关键实现函数

串口初始化

函数调用信息

```
rtems_device_driver tty1_initialize(rtems_device_major_number major,
                                   rtems_device_minor_number minor,
                                   void *arg)
rtems_device_driver tty2_initialize(rtems_device_major_number major,
                                   rtems_device_minor_number minor,
                                   void *arg)
```

输入参数为串口设备的主从设备号 major 和 minor，附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 首先判断该串口是否被指定为终端口，如果不是的话继续执行初始化，否则直接退出；
2. 执行 TERMIOS 的初始化

3. 设定串口参数，包括波特率、数据位、奇偶校验、停止位和数据流控制类型；
4. 绑定中断服务例程；
5. 注册设备名称与主从设备号匹配。

打开串口□□□

函数调用信息

```
rtems_device_driver tty1_open(rtems_device_major_number major,
                             rtems_device_minor_number minor,
                             void *arg)
rtems_device_driver tty2_open(rtems_device_major_number major,
                             rtems_device_minor_number minor,
                             void *arg)
```

输入参数为串口设备的主从设备号 major 和 minor，附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 根据系统设定是中断模式（TERMIOS_IRQ_DRIVEN）还是任务模式（TERMIOS_TASK_DRIVEN），分别设置回调函数接口；
2. 调用 rtems_termios_open 打开一个字符设备；
3. 设定串口的系统参数；
4. 使能中断。

关□串□□□□

函数调用信息

```
rtems_device_driver tty_close(rtems_device_major_number major,
                              rtems_device_minor_number minor,
                              void *arg)
```

输入参数为串口设备的主从设备号 major 和 minor，附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 直接调用 rtems_termios_close 关闭字符设备。

从串□□□□□数据

函数调用信息

```
rtems_device_driver tty_read(rtems_device_major_number major,
                             rtems_device_minor_number minor,
                             void *arg)
```

输入参数为串口设备的主从设备号 major 和 minor，附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 直接调用 rtems_termios_read 读取数据。

向串□□□□□写数据

函数调用信息

```
rtems_device_driver tty_write(rtems_device_major_number major,
                             rtems_device_minor_number minor,
                             void * arg)
```

输入参数为串口设备的主从设备号 major 和 minor，附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 直接调用 rtems_termios_write 写出数据。

串口 ioctl

函数调用信息

```
static rtems_device_driver tty_control( int port, void *arg )
```

输入参数为串口设备号（COM1或者 COM2），附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 直接调用 rtems_termios_ioctl 模块处理各种请求。

接下来简述一下 I386平台下的 console 驱动，其大部分的实现都与串口驱动的实现类似。我们只向大家介绍使用串口时的处理流程，VGA 模式的处理读者有兴趣的话可以参考

`${RTEMS_ROOT}/c/src/lib/libbsp/i386/pc386/console` 目录。

关键数据结构

为 console 实现的中断描述数据结构：

```
static rtems_irq_connect_data console_isr_data = {BSP_KEYBOARD,
                                                  keyboard_interrupt_wrapper,
                                                  0,
                                                  isr_on,
                                                  isr_off,
                                                  isr_is_on};
```

其具体信息可参考中断处理一章

关键实现函数

console 初始化

函数调用信息

```
rtems_device_driver console_initialize(rtems_device_major_number major,
                                       rtems_device_minor_number minor,
                                       void *arg)
```

输入参数为设备的主从设备号 major 和 minor，附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 执行 TERMIOS 的初始化；
2. 调用 uart 串口驱动设定串口参数，包括波特率、数据位、奇偶校验、停止位和数据流控制类型；
3. 绑定中断服务例程；
4. 注册设备名称与主从设备号匹配。

打开 console

函数调用信息

```
rtems_device_driver console_open(rtems_device_major_number major,
                                rtems_device_minor_number minor,
                                void *arg)
```

输入参数为设备的主从设备号 major 和 minor，附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 调用 rtems_termios_open 打开设备；
2. 调用 uart 串口驱动设定串口的系统参数；
3. 调用 uart 串口驱动使能中断。

关 console

函数调用信息

```
rtems_device_driver console_close(rtems_device_major_number major,
                                  rtems_device_minor_number minor,
                                  void *arg)
```

输入参数为设备的主从设备号 major 和 minor，附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 直接调用 rtems_termios_close 关闭字符设备。

从 console 口数据

函数调用信息

```
rtems_device_driver console_read(rtems_device_major_number major,
                                  rtems_device_minor_number minor,
                                  void *arg)
```

输入参数为设备的主从设备号 major 和 minor，附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 直接调用 rtems_termios_read 读取数据。

向 console 写数据

函数调用信息

```
rtems_device_driver console_write(rtems_device_major_number major,
                                   rtems_device_minor_number minor,
                                   void *arg)
```

输入参数为设备的主从设备号 major 和 minor，附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 直接调用 rtems_termios_write 写出数据。

console □□ ioctl □

函数调用信息

```
rtems_device_driver console_control(rtems_device_major_number major,
                                     rtems_device_minor_number minor,
                                     void * arg)
```

输入参数为设备的主从设备号 major 和 minor，附加参数的地址 arg。

函数功能描述：

执行步骤如下：

1. 直接调用 rtems_termios_ioctl 模块处理各种请求。

Uart 串口□□□

关键数据结构

串口参数配置

```
struct uart_data
{
    int ioMode;
    int hwFlow;
    unsigned int ier;
    unsigned long baud;
    unsigned long databits;
    unsigned long parity;
    unsigned long stopbits;
};
```

其中包括了基本的串口参数，例如波特率、数据位、停止位、流控制模式等等。

关键实现函数

□串口

函数调用信息

```
static inline unsigned char uread(int uart, unsigned int reg)
```

输入参数为串口编号 uart，寄存器编号 reg。

函数功能描述：

直接调用 inport_byte 从串口读出一个字节。

写串口

函数调用信息

```
static inline void uwrite(int uart, int reg, unsigned int val)
```

输入参数为串口编号 uart，寄存器编号 reg，以及待写的值 val。

函数功能描述：

直接调用 `outport_byte` 向串口写入一个字节。

串口初始化

函数调用信息

```
void BSP_uart_init
(
    int uart,
    unsigned long baud,
    unsigned long databits,
    unsigned long parity,
    unsigned long stopbits,
    int hwFlow
)
```

输入参数为标准的串口配置信息

函数功能描述

按照 `uart.h` 中定义的标准向串口写入控制字符、控制位信息以把串口初始化成要求的状态（包括默认的参数，和通过该函数参数传递进来的参数）。

口置串口参数

函数调用信息

```
void BSP_uart_set_attributes
(
    int uart,
    unsigned long baud,
    unsigned long databits,
    unsigned long parity,
    unsigned long stopbits
)
```

输入参数为待设定的串口配置信息

函数功能描述

其实设置串口参数的过程就是使用新的参数再次调用 `BSP_uart_init` 初始化串口。这里我们注意到，在设置参数之前，先把 `MCR`（状态寄存器）、`IER`（中断势能寄存器）保留起来，然后完成初始化操作，最后再把保留起来的 `MCR`、`IER` 写回。也就是虽然初始化设备，但是却不把 `MCR`、`IER` 回归初始值。

设定串口中断类型

函数调用信息

```
void BSP_uart_intr_ctrl(int uart, int cmd)
```

输入参数为串口编号 `uart` 和命令 `cmd`

函数功能描述：

依据 cmd 命令，设定中断的类型，并写入串口的中断使能寄存器中。

串口 POLL 模式状态检测

函数调用信息

```
int BSP_uart_polled_status(int uart)
```

输入参数为串口编号 uart

函数功能描述

这个函数用来返回 POLL 模式下的状态：-1 表示错误，0 表示收到的字符不合法，1 代表收到的字符合法，2 表示 break 发生。首先，它从 LSR 寄存器中读取状态字 val，val & BI，常量 BI 表示 break interrupt 的存在，如果与的结果为真，那么就返回常量“status_break”。如果(val & (DR | OE | FE))为真，其中，DR 表示数据已经准备好，OE 表示 overrun error，FE 表示 framing error。那么说明没有错误，返回 1，否则返回 0，代表字符不合法。这里的没有错误意味这 RTEMS 忽略了 OE 错误和 FE 错误。最后，如果以上的条件都不满足，那么就应该返回-1，代表错误。

串口在 POLL 模式下的写操作

函数调用信息

```
void BSP_uart_polled_write(int uart, int val)
```

输入参数为串口编号 uart 和待写入的字节

函数功能描述

首先这个函数阻塞直到满足条件：(val1=uread(uart, LSR)) & THRE，也就是直到传送寄存器可用（为空）时，再往串口写入一个参数指定的字节。之后再次阻塞直到满足条件((val1=uread(uart, LSR)) & THRE)。也就是等待这个字节成功写入。这个操作保证串口能够成功的写入数据。不执行这样的等待操作有可能导致写丢。

串口在 POLL 模式下的读操作

函数调用信息

```
int BSP_uart_polled_read(int uart)
```

输入参数为串口编号 uart

函数功能描述

首先这个函数阻塞直到满足条件：uread(uart, LSR) & DR，也就是数据已经准备好。数据准备好之后，从串口读取一个字节直接返回。

串口的一般口操作

函数调用信息

```
int BSP_uart_termios_read_com1(int uart)
```

```
int BSP_uart_termios_read_com2(int uart)
```

输入参数为串口编号 `uart`

函数功能描述

首先用一个 `while` 循环判断是否越界和是否数据准备完毕，如果数据准备完毕并且不越界，则持续读字节。如果写越界的话则把越界部分通过函数 `rtems_termios_enqueue_raw_characters` 写入到 `raw_enqueue` 中。最后打开接收中断。

串口的一般写操作

函数调用信息

```
int BSP_uart_termios_write_com1(int minor, const char *buf, int len)
int BSP_uart_termios_write_com2(int minor, const char *buf, int len)
```

输入参数为待写入数据 `buf` 和数据长度 `len`

函数功能描述

首先是合法性判断，判断传入的 `len` 是否大于 0。之后判断串口是否空闲，或者如果现在串口处于停止状态，返回。利用 `uwrite` 写入数据。最后，判断一下串口的 `tx` 是否已经激活，如果没有被激活，激活它并且允许中断。

□用□例——使用串口作□ console 的□入口□出

我们可以使用 QEMU 的虚拟串口来实现 console 的串口输入输出方式。在解压了 `rtems` 代码后，在配置编译整个 `rtems` 以前，需要修改 `rtems/c/src/lib/libbsp/i386/pc386/configure.ac` 中的第21行

```
RTEMS_BSPOPTS_SET([USE_COM1_AS_CONSOLE], [0], [0])
```

为

```
RTEMS_BSPOPTS_SET([USE_COM1_AS_CONSOLE], [0], [1])
```

然后删除 `rtems/c/src/lib/libbsp/i386/pc386/configure`，并在 `rtems/c/src/lib/libbsp/i386/pc386/` 目录下通过运行如下命令重新生成 `rtems/c/src/lib/libbsp/i386/pc386/configure`

```
autoconf
```

最后再重新配置和编译 `rtems`，得到缺省通过 `pc 386` 的 `COM1` 进行输入输出的 `rtems`

在调试的时候通过如下命令（主要增加了 `-serial stdio` 这个 `qemu` 参数）就可以在命令行看到串口输入比过来进行输出了。

```
qemu -fda XXX.img -serial stdio
```

小□

第十二章 裁减定制与移植

RTEMS 有很好的可移植性与可剪裁性，可移植主要是指能将 RTEMS 移植到新的硬件上，而剪裁与定

制指的是根据硬件以及应用程序的需要，对 RTEMS 库进行配置，使最后生成的可执行代码满足用户的需要。本章将介绍 RTEMS 的移植与裁剪定制。

实时系统移植需要程序员编写板级支持包 BSP 和处理器支持包 CSP。BSP 和 CSP 可以看成介于硬件和操作系统间的中间层，应该说是属于操作系统的一部分，主要目的是为了支持操作系统，使之能够更好的运行于硬件主板。RTEMS 的 BSP 和 CSP 使用了层次化设计方法，保证了 RTEMS 以及应用程序能访问对应的硬件平台。这些软件包括硬件初始化代码，设备驱动程序，用户扩展，多处理器接口等。当然，最小的 BSP 和 CSP 可以只包括处理器复位，初始化，线程切换，中断处理等。

本章将通过分析实际代码，介绍 RTEMS 的 BSP 和 CSP 的设计思路。为了描述方便，除非特殊说明，下面的章节将 BSP 和 CSP 都统称为 BSP。

12.1 嵌入式系 BSP 结构

对于熟悉运行与 PC 机上的 windows 或 linux 系统的朋友而言，常认为 BSP 就是设备驱动程序。但是这个认识是片面的。其原因在于 PC 机均采用统一的 x86 体系架构，这样通用操作系统（windows, linux..）不需要做任何修改就可以在 x86 上正常运行。这样，Window 和 Linux 的驱动程序员将会更着重于外设的访问而不是对处理器本身的控制上。

而对嵌入式系统来说情况则完全不同，目前市场上多种结构的嵌入式处理其并存(ARM, MIPS, PowerPC, Sparc....),为了性能的需要，外围设备也会有不同的选择和定义。一个嵌入式操作系统针对不同的 CPU，会有不同的 BSP，即使同一种 CPU，由于片上外设的一点差别（如外部扩展 SRAM 的大小，类型改变），BSP 相应的部分也不一样。所以根据硬件设计编写和修改 BSP，保证系统正常的运行是非常重要的。从 PC 驱动过渡到嵌入式系统驱动的程序员还应该注意一点就是，由于 PC 的硬件结构比较固定，所以在实际开发驱动程序的时候，通常没有必要真正直接对硬件进行操作，例如 Window 程序员会使用到 HAL 的 API，但是对于嵌入式程序员，经常需要对形形色色的片上外设以及板上外设进行访问，使用的硬件访问方式和总线接口也形形色色常常需要花费大量时间阅读芯片文档才能写出正确、高效的 BSP。

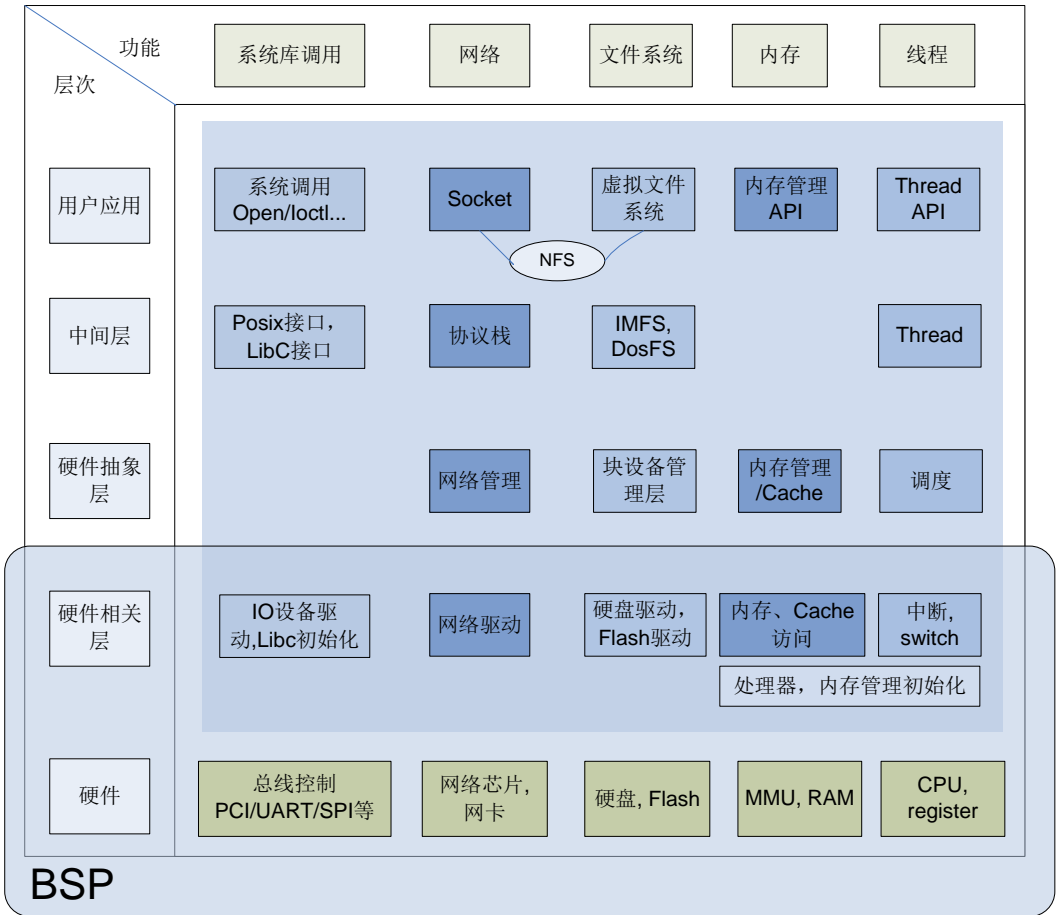


图12-1 RTEMS 内核层次结构图

图12-1 是 RTEMS 内核的层次结构图。从图上可以看出，BSP 的主要功能是操作硬件，它包括：最底层的 CPU 以及内存控制器初始化部分；中断以及线程切换部分；

内存与 Cache 访问以及控制部分；
外设以及 I/O 驱动部分；
libc 库函数初始化部分等。

在 RTEMS 中，BSP/CSP 的相关代码主要存放在 `cpukit/score/cpu/`，`c/src/lib/libcpu/`，`c/src/lib/libbsp/`和 `c/src/libchip/`下面介绍这几个目录的作用。

(1) `cpukit/score/cpu/` 目录。RTEMS 4.7以后版本中，与处理器 CPU 的 CSP 相关目录是 `cpukit/score/cpu/`，例如 `cpukit/score/cpu/i386`，`cpukit/score/cpu/mips`，`cpukit/score/cpu/arm`。`score` 目录下面和 CPU 相关代码主要功能是定义中断处理函数和线程上下文切换函数等，而且需要注意的是，`score` 目录下面的 CPU 存放的是处理器通用代码，这些代码是按照处理器类型划分，并不是按照处理器划分，例如 ARM 的处理器可能会有 ST，Atmel，Motorola，NXP 等公司的产品，但是 `score` 中只有一个 `score/cpu/arm/`目录。`cpukit/score/cpu/`目录下面代码的介绍将在12.11移植到新的处理器一节中进行。

(2) `c/src/lib/libcpu/` 目录。CPU 支持文件目录 `c/src/lib/libcpu/`。这里放了和 CPU 紧密相关的代码和驱动（例如 Soc 上集成的串口，I2C 总线等）。这个目录下面的文件夹分类则是根据具体处理器的型号来划分的。早期的代码没有这个目录，后来由于支持的硬件体系越来越多样化，而且集成度也越来越高，就单独设置了这个目录。例如同样一款 Atmel 的 9200 处理器，可能会应用到不同的硬件体系上，有的硬件板需要多个串口，有的需要扩展多个网口，但是不同的硬件板，和 `at91rm9200` 相关的处理(例如时钟，中断等)还是一样的。总的来说，这个目录主要是用于存放系统级处理器 SoC 的片上集成外设的驱动程序。

(3) `c/src/lib/libbsp` 目录。`c/src/lib/libbsp` 存放的是与嵌入式主板，外围设备相关的驱动程序。

(4) `c/src/libchip/` 目录。存放各种独立外设芯片驱动（例如网口，串口，RTC 等）。

下面首先介绍 `c/src/lib/libbsp/`目录，该目录结构如下：

- `c/src/lib/libbsp/shared`
- `c/src/lib/libbsp/CPU/shared`
- `c/src/lib/libbsp/CPU/<BSP>`

通常有下面的<BSP>：

- `console`: 控制台（串口）
- `clock`: 时钟管理
- `timer`: 定时器管理。
- `rtc`: 硬件 RTC。
- `nvmem`: EEPROM 或者 Flash 管理。
- `network`: Ethernet 驱动。
- `shmsupp`: 多处理系统中共享内存 MPCIE 层
- `include`: 头文件
- `wrapup`: 打包文件，现在很少用了

设计一个新 BSP 要做的：

- 为新 BSP 创建 `Makefile.am`
- 将一个接近的 `.cfg` 拷贝为 `BSP.cfg`
- 修改 `BSP.cfg` 中的 `RTEMS_CPU`，`RTEMS_CPU_MODEL`，`RTEMS_BSP_FAMILY`，`RTEMS_BSP`，`CPU_FLAGS`，`START_BASE`，变量以及 `make` 规则。
- 如果目标硬件的 CPU 类型在当前 `score` 中不支持，需要创建 `score/cpu/<cpu_family>`目录以及相应文件，这里 `cpu_family` 指的是处理器类型，例如 `i386`，`ARM` 等
- 创建 `c/src/lib/libcpu/<cpu_model>`目录以及相应文件，这里 `cpu_model` 指的是处理器型号，例如 ARM 中的 `atmel9200`或者是 `lpc2210`等。
- 创建 `c/src/lib/libbsp/<bsp_name>`目录以及相应文件。这里 `<bsp_name>`指的是 BSP 运行的硬件电路板的编号/名称
- 修改 `score/cpu/`，`c/src/lib/libcpu/`以及 `c/src/lib/libbsp/`中的 `m4`脚本，让 `autoconf/automake` 能解析新添目录。

上面介绍了 RTEMS BSP 的基础知识，下面会分步介绍如何进行 RTEMS BSP 开发。

12.2 Linkcmds

产生可执行文件的最后一步是链接目标文件和库文件。RTEMS 缺省将生成 `elf` 格式的 `image`，程序员可以根据自己的需要在实际应用中通过 `bootloader` 加载 `elf` 格式的可执行文件，或者使用二进制

转换工具 objcopy 转换成适合在 ROM 中存储的格式。由于嵌入式系统硬件上的多样性（不同硬件系统的内存起始地址，大小不同），各个 BSP 需要单独的链接脚本。

可执行代码包含多个程序字段（section），gnu 的 ld 命令 ld 软件的作用是把各种目标文件（.o 文件）和库文件（.a 文件）链接在一起，并定位数据和函数地址，并将其放到定义好的字段中，最终生成可执行程序。

对于嵌入式系统来说，存储器是稀缺资源，为了程序能顺利执行，必须有两种内存：

- **RAM**：随机存储器，用于存储程序运行时的临时数据（栈，堆等）
- **ROM**：只读存储器，掉电时数据不会丢失例如 ROM, PROM, EEPROM, Flash 等。

为了嵌入式系统能够顺利运行，嵌入式程序员必须将合理的分配寄存器资源。通常来说，目标

二进制文件至少包括下面的程序段组成



图12-2 Image 组成

- 代码(.text)段: 程序的可执行代码，资源文件（例如字库），而且它不应该被修改。该区段可放在只读存储器 ROM 中。例如图20-2中，地址0x00000000到0x01FFFFFF的空间可以用于存放.text
- 未初始化内存段（.bss）：用于存放程序的未初始化的全局临时变量。虽然 C 标准并没有要求.bss 字段中的内容正确初始化，但是一般来说 BSP 的开发者应该把.bss 字段中的堆清零。
- 初始化的数据区段（.data）：容纳已经初始化的全局变量（有初始值的）。有时候，变量的初始值存放在 ROM 中。由 bootloader 或者初始化程序将对应的值拷贝到内存.data 段。

使用 objdump 加上-h 选项可以查看 elf 格式镜像文件的各个 section 内容，下面是 rtl22xx BSP 下的 hello.exe 各个 section 段的情况。

```
hello.exe:  file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
 0 .base           00000100 40000000 40000000 00000074 2**0
   ALLOC
 1 .text           00009314 81000000 81000000 00000080 2**2
   CONTENTS, ALLOC, LOAD, CODE
 2 .init           00000014 81009314 81009314 00009394 2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 3 .fini           00000010 81009328 81009328 000093a8 2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 4 .data           00000c6c 81009338 81009338 000093b8 2**2
   CONTENTS, ALLOC, LOAD, DATA
 5 .eh_frame       00000004 81009fa4 81009fa4 0000a024 2**2
   CONTENTS, ALLOC, LOAD, READONLY, DATA
 6 .rodata         00000744 81009fa8 81009fa8 0000a028 2**2
   CONTENTS, ALLOC, LOAD, READONLY, DATA
 7 .bss            00000e40 8100a700 8100a700 0000a76c 2**5
   ALLOC
 8 .comment        00000f66 00000000 00000000 0000a76c 2**0
   CONTENTS, READONLY
 9 .debug_aranges  00001b10 00000000 00000000 0000b6d8 2**3
   CONTENTS, READONLY, DEBUGGING
.....
```


对象文件中的每一个段都有名字和大小。大多数的段还有一个相连的数据块，也就是"section contents"。一个被标记为可加载（LOAD）的段，意味着在输出文件运行时，contents 可以被加载到内存中。没有 contents 的段也可以被加载。而既不是可加载的又不是可分配的（ALLOC）段，通常包含了某些调试信息。

以上面的 hello.exe 为例

| Idx | Name | Size | VMA | LMA | File off | Algn |
|-----------------------------|-------|----------|----------|----------|----------|------|
| 1 | .text | 00009314 | 81000000 | 81000000 | 00000080 | 2**2 |
| CONTENTS, ALLOC, LOAD, CODE | | | | | | |

段.text 的大小为0x9314，其加载地址和运行地址都是0x81000000，表示程序需要加载到内存0x81000000的地方运行。

每个可加载或可分配的输出段（output section）都有2个地址。第一个是虚拟存储地址 VMA（virtual memory address），这是在输出文件执行时该段所使用的地址。第二个是加载存储地址 LMA（load memory address），这是该段被加载时的地址。在大多数情况下，这两个地址是相同的，例如上面的 hello.exe。但是当 .data 段加载在 ROM 中，后来在程序开始执行时又拷贝到 RAM 中（在基于 ROM 的系统中，这种技术经常用在初始化全局变量中）。在这种基于 ROM 的系统情况下，这时，ROM 地址是 LMA，而内存地址是 VMA。

每个目标文件也有一个符号列表，这就是符号表(symbol table)。一个符号可以是"已定义"（defined）或"未定义"（undefined）的。每个符号有名字，并且每个定义了符号还有地址。在编译一个 C/C++ 程序成目标文件时，每个定义的函数，全局变量，静态变量，都可以有一个"已定义"的符号。输入文件中引用的每个没有定义的函数和全局变量则变成"未定义"的符号。

使用 nm 可以查看对象文件中的 symbol，objdump 并使用"-t"选项也提供类似的功能。通常，如果使用标准的 RTEMS Makefile 文件，那么在最后的可执行文件目录中将会有记录链接过程的文件.num 文件和.map 文件。.num 是对应可执行文件 hello.exe 中所有符号的符号表。以 rtems 源代码的 sample 目录中的 hello 为例。

下面是使用 rtl22xx BSP 编译的 hello.num 的部分：

```
81000000 A _sdram_base
81000000 T _start
81000000 T _text_start
810000b8 T Reset_Addr
810000d8 T Undefined_Handler
810000dc T SWI_Handler
810000e0 T Prefetch_Handler
810000e4 T Abort_Handler
810000e8 T IRQ_Handler
810000ec T FIQ_Handler
810001b4 T Init
810001cc T _Barrier_Manager_initialization
810001d0 T _Dual_ported_memory_Manager_initialization
810001d4 T _Event_Manager_initialization
810001d8 T _Message_queue_Manager_initialization
810001dc T _Partition_Manager_initialization
810001e0 T _Region_Manager_initialization
810001e4 T _Signal_Manager_initialization
```

这里

```
81000000 A _sdram_base
```

这行说明了符号_sdram_base 在产生可执行文件 hello.exe 中加载地址为0x81000000。81000000表示16进制地址，"A"表示符号值是绝对值(Absolute)，此值在多次 link 过程中不会被修改。再看第二行

```
81000000 T _start
```

这行表示符号_start 被链接到0x81000000。"T"表示_start 在.text 段。同时，我们也可以看到，同样是0x81000000地址，被多个符号用到。这并不表示多个符号被加载到这个地址，而是因为_sdram_base 的作用类是指针变量，这个变量在 link 脚本中定义，用于定义 link 过程的基地址。

RTEMS 的 BSP 链接脚本 linkcmd 是一个文本文件，其内容是链接器命令语言。每个命令是一个关键字，可能还带着参数，又或者是对一个符号的赋值。程序员可以使用分号来隔开命令，脚本中的空格则会被 ld 忽略。

在链接脚本中可以使用注释，就像在 C 中，定界符是"/*"和"*/"，和 C 中一样，注释会被 ld 解释为空格。

12.3 连接器常用命令：

(1) 设置入口点

格式：ENTRY(symbol)

设置 symbol 的值为执行程序的入口点。

ld 有多种方法设置执行程序的入口点，确定程序入口点的顺序如下：

- ld 命令的-e 选项指定的值
- Entry(symbol)指定的值
- .text 段的起始地址
- 入口点为0

RTEMS 中的入口地址通常是_start，所以 linkcmd 中通常要有：

```
ENTRY(_start)
```

用于指定入口地址，这样_start 函数将会被 ld 定向到.text 的起始地址。

输出格式。通常使用 OUTPUT_FORMAT 和 OUTPUT_ARCH 指定输出的格式，例如 ARM 系统中，一般使用：

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")
OUTPUT_ARCH(arm)
```

来指定输出格式。这里 OUTPUT_FORMAT 用来指定输出文件的格式，OUTPUT_ARCH 用来指定输出的目标机器体系结构。

MEMORY 指令，该命令描述了目标系统各个内存块的起始地址和大小。嵌入式系统中，各种处理器的内存空间各不相同，而且同一个处理器，可能因为片选信号联结方式不同，或者存储芯片容量差别导致可用存储空间不同。MEMORY 指令的格式如下：

```
MEMORY
{
    name [(attr)]:ORIGIN = origin,LENGTH = len
    ...
}
```

例如：

```
MEMORY
{
    sdram : ORIGIN = 0x20100000, LENGTH = 15M
    sram : ORIGIN = 0x00200000, LENGTH = 16K
}
```

上面的指令表示目标系统有两种内存，一个是 SDRAM，起始位置是0x20100000 SDRAM 的大小是15M，另一种是静态内存，起始地址是0x00200000，内存块的大小是16K。

变量定义，在 linkcmd 中可以定义变量，使用这些变量，可以更方便的定义各个段。例如：

```
_sdram_base = DEFINED(_sdram_base) ? _sdram_base : 0x20100000;
```

这里定义了_sdram_base，如果这个变量没有在其他地方定义的话，就定义成0x20100000，又比如可以使用下面的命令定义 ARM 中的 IRQ 栈的大小：

```
_irq_stack_size = DEFINED(_irq_stack_size) ? _irq_stack_size : 0x1000;
```

SECTIONS 命令

SECTIONS 告诉 ld 如何把输入文件的各个段映射到输出文件的各个段中。

在一个链接描述文件中只能有一个 SECTIONS 命令

在 SECTIONS 命令中可以使用的命令有四种：

- ENTRY 指令定义入口点
- 赋值
- 定义输出段
- 覆盖(Overlay)描述

12.4 linkcmd 例分析

链接脚本的作用是描述输入的代码/数据段如何映射到最后的 image 中。下面是 RTEMS 的在线文档中提供的 linkcmd 样例：

```
/*
 * 输出格式定义为 coff-m68k
 */

OUTPUT_FORMAT(coff-m68k)
```

```

/*
 * 目标板内存大小
 *
 * NOTE: 该值会被传递给 ld 的参数覆盖
 */

RamSize = DEFINED(RamSize) ? RamSize : 4M;

/*
 * 应用程序堆空间。应用程序中 malloc 分配的内存来自堆空间。
 * 对于一个应用，定义尽可能小的堆空间通常来说比较麻烦，
 * 但是如果应用程序使用的内存在初始化的时候就从堆中分配了
 * 需要的所有内存，那么定义堆的大小就会比较简单。*
 * NOTE 1: 该值会被传递给 ld 的参数覆盖。
 *
 * NOTE 2: TCP/IP 协议栈会需要额外的堆空间
 *
 * NOTE 3: GNAT/RTEMS run-time 会需要额外的堆空间
 */
/*定义堆空间大小为0x10000*/
HeapSize = DEFINED(HeapSize) ? HeapSize : 0x10000;

/*
 * 设置 BSP 初始化阶段的堆栈空间大小
 * NOTE: 可以通过向 ld 传递参数覆盖该值。
 */

StackSize = DEFINED(StackSize) ? StackSize : 0x1000;

/*
 * RAM 和 ROM.的大小和起始地址。
 *
 * 地址和大小必须根据实际硬件情况（例如根据片选信号等）来定
 */

MEMORY {
    ram : ORIGIN = 0x10000000, LENGTH = 4M
    rom : ORIGIN = 0x01000000, LENGTH = 4M
}

/*
 * 以太网 SRAM 的起始地址
 */

ETHERNET_ADDRESS =
    DEFINED(ETHERNET_ADDRESS) ? ETHERNET_ADDRESS : 0xDEAD12;

/*
 * 下面定义了各个段在镜像中的地址和排列顺序
 * 同时也定义了一些应用程序中可以使用的变量
 *
 * NOTE: 有些变量名称是由一个或者两个下划线开始，这主要是为了方便 C/C++
 * 代码访问这些变量，在一些目标代码中，LD 会自动在 C 可以访问的全局变量前面
 * 加上一个下划线。
 */

SECTIONS {

    /*
     * 使 RomBase 变量可以被 C 语言访问
     */

    _RamSize = RamSize;
    __RamSize = RamSize;

    /*
     * Boot PROM - 设置 RomBase 变量为 ROM 的起始位置

```

```

*/

rom : {
    _RomBase = .;
    __RomBase = .;
} >rom

/*
 * 动态 RAM – 设置变量 RamBase 为 RAM 的起始地址.
 */

ram : {
    _RamBase = .;
    __RamBase = .;
} >ram

/*
 * text (code) goes into ROM
 */

.text : {
    /*
     * 为每个目标文件(.o)创建符号表
     */

    CREATE_OBJECT_SYMBOLS

    /*
     * 以下是目标文件(.o)的.text 区
     */

    *(.text)

    . = ALIGN (16);    /* 16-字节对齐*/

    /*
     * C++ 构造和析构 、
     *
     * NOTE: "[.....]"的使用可参考 CROSSGCC 邮件链表 FAQ.
     */

    __CTOR_LIST__ = .;
    [.....]
    __DTOR_END__ = .;

    /*
     * .text 段的结束
     */

    etext = .;
    _etext = .;
} >rom

/*
 * 异常处理帧
 */

.eh_frame : {
    . = ALIGN (16);
    *(.eh_frame)
} >ram

/*
 * GCC 异常
 */

.gcc_exc : {
    . = ALIGN (16);
    *(.gcc_exc)
}

```

```

} >ram

/*
 * 双口内存区的定义，应用程序可以通过访问_m340来访问双口内存起始地址
 */

dpram : {
    m340 = .;
    _m340 = .;
    . += (8 * 1024);
} >ram

/*
 * 初始化.data 段
 */

.data : {
    copy_start = .;
    *(.data)

    . = ALIGN (16);
    _edata = .;
    copy_end = .;
} >ram

/*
 * BSS 段
 */

.bss : {
    /*
     * M68K 的中断向量
     * (256个中断，每个4字节).
     */

    M68Kvec = .;
    _M68Kvec = .;
    . += (256 * 4);

    /*
     * 需要初始化（清零）的内存地址起始.
     */

    clear_start = .;

    /*
     * 所有目标文件(.o)中的 bss 数据都放在这里
     */

    *(.bss)

    *(COMMON)

    . = ALIGN (16);
    _end = .;

    /*
     * 应用程序堆空间起始位置
     */

    _HeapStart = .;
    __HeapStart = .;
    . += HeapSize;

    /*
     * 应用程序栈空间起始地址，对于 M68K，该空间必须在堆空间后面。
     * 因为 M68K 堆栈是向下增长的
     */

```

```

. += StackSize;
. = ALIGN (16);
stack_init = .;

clear_end = .;

/*
 * RTEMS 可执行文件 Workspace 的起始地址。RTEMS
 * 为 tasks, stacks, semaphores 等分配内存都在这个空间
 */

_WorkspaceBase = .;
__WorkspaceBase = .;
} >ram
}

```

链接与初始化过程

对于通用操作系统，例如 Linux，在启动的时候，需要将 Linux 的 Kernel 整个解压拷贝到内存中才能执行。对于一些 SOC 芯片例如飞利浦(NXP)的 lpc21xx 系列以及 atmel 的 SAM 系列 ARM 处理器，片上的 ROM 和 SRAM 都很有限，而且也无法扩展片外的 ROM 和 SRAM，也就不能使用 bootloader 加载 elf 格式的 image。在这种情况下，image 需要对 image 做特殊处理。.data 区域由于在程序运行过程中，数据会变化，所以必须存在在内存中。为了解决这个问题，产生 Image 的时候，将 .data 段中的数据也存放在 ROM 段中，在系统启动的时候，将片上的 SRAM 划分成 .bss 和 .data 后，将 ROM 区域中的 .data 数据拷贝到 .data 区中（如图12-3所示）。

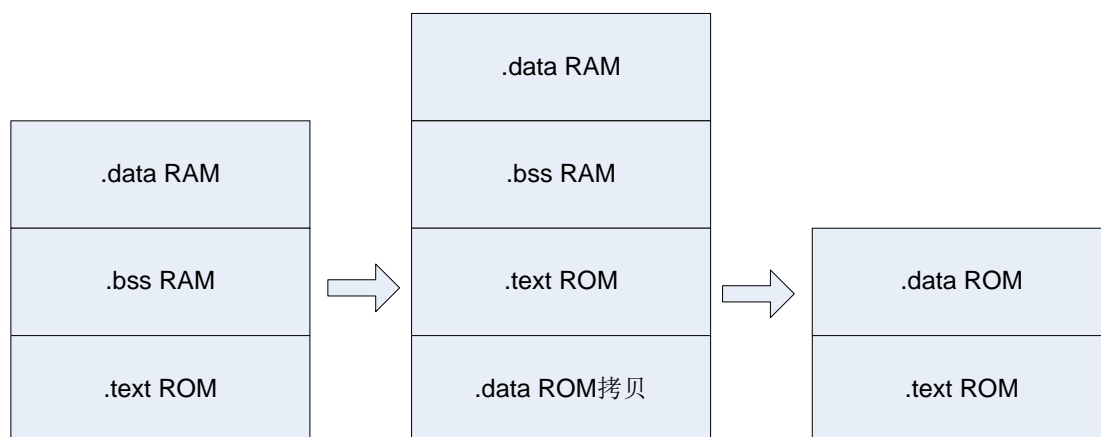


图 12-3 image 生成过程

在这种情况下，产生 image 的步骤如下：

- 1) 按照 BSP 中的 Linkcmd 脚本生成可执行 image
- 2) 建立一个 .data 段的拷贝，将这个拷贝放在 .text 段后面，这一步在链接的时候完成，下面是 M68K 的一个生成 Image 的一个例子，该例子来自 \$RTEMS_ROOT/make/custom/gen68340.cfg

```

# make a PROM image using objcopy
m68k-rtems-objcopy \
--adjust-section-vma .data= \
`m68k-rtems-objdump --section-headers \
$(basename $@).exe \
| awk '[...]' \
$(basename $@).exe

```

这里，使用 awk 提取了 .text 的结束位置，然后将 .data 拷贝到 .text 结束位置。

- 3) 在系统启动的时候，需要启动初始化代码将 ROM 中 .data 拷贝到片上 SRAM 中的预留区域。

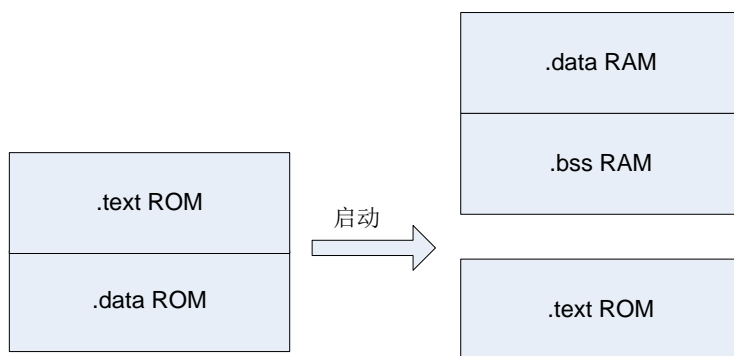


图 12-4 启动过程

12.5 其它 BSP 相关文件

其它 BSP 相关文件包括：bsp_specs, README, bsp.h, coverhd.h, 重载 sbrk 的 C 语言代码源文件。下面依次进行介绍。

(1) bsp_specs 文件，该文件是 gcc 编译器用到的编译配置文件。该文件中，开发人员可以指定 BSP 使用到的 gcc 相关库文件和初始化二进制文件(start.o, crt0.o 等)。下面是一个 bsp_spec 的例子：

```
%rename cpp old_cpp
%rename lib old_lib
%rename endfile old_endfile
%rename startfile old_startfile
%rename link old_link

*cpp:
%(old_cpp) % {qrtems: -D__embedded__} -Asystem(embedded)

*lib:
% { !qrtems: %(old_lib) } % {qrtems: --start-group \
% { !qrtems_debug: -lqrtems } % {qrtems_debug: -lqrtems_g } \
-lc -lgcc --end-group ecrt0.o %s \
% { !qno linkcmds: -T linkcmds %s } }

*startfile:
% { !qrtems: %(old_startfile) } % {qrtems: ecrt0.o %s \
% { !qrtems_debug: startsim.o %s } \
% {qrtems_debug: startsim_g.o %s } }

*link:
% { !qrtems: %(old_link) } % {qrtems: -Qy -dp -Bstatic \
-T linkcmds %s -e _start -u __vectors }
```

上面的例子中，首先将编译过程中使用到的配置变量备份为 old_xxx，例如 old_cpp，这样，在后面使用这些变量的时候，就可以在使用原始值的同时，也使用一些 RTEMS 专用的编译参数。

然后是*cpp 定义，在这部分中，首先包含了原来的 old_cpp，然后，如果定义了 qrtems，那么就定义预处理变量__embedded__（使用-D 参数）。

在*lib 的定义部分，保证了当为 RTEMS 编译的时候，BSP 的 linkcmds、gcc 支持库、EABI 等会被正确的使用。

在*startfile 的定义中，将 startsim.o 链接到最后的可执行文件中，而不是链接 crt0.o。同时，需要将 EABI 的 ecrt0.o 链接到可执行文件中。

*link 定义中，指定了需要向 ld 传递的参数。

(2) BSP 中 README 文件是开发者为 BSP 写作的说明文件，times 文件是该 BSP 在执行 time 测试的时候得到的结果。

(3) bsp.h 文件定义了实际硬件的参数，例如时钟频率，板上的硬件资源配置等。有时候，也将 linkcmd 中定义的一些变量原型放到该文件中。

(4) coverhd.h 和 times 一样，也是描述函数调用的额外的延时。这里的延时参数都是根据时间测试用例得到的结果。例如

```
#define CALLING_OVERHEAD_IO_OPEN 4
```

表示如果调用 open() 函数打开 IO 设备的过程中，向 open() 函数通过压栈传递参数，调用跳转指令，以及最后函数返回的时间的总和。

```
/*open 指令调用的代价是多少? */
fd = open("/dev/tty", O_RDONLY      );
.....
```

需要注意的是，这里的函数的执行时间本身是不包括在额外时间代价中的，也就是说，这里定义的时间是函数调用的额外开销，函数执行时间本身不包括在内。

(5) 重载 `sbrk`。有些 BSP 需要能动态的扩展堆空间，而缺省的 `sbrk` 函数是一个空函数，这就需要定义 BSP 专门的 `sbrk`。读者可以参考 PowerPC BSP 中 `skbrk` 的写法。同样，`libbsp/share` 目录下面还有其他的一些 BSP 公用函数例如 `bsp_cleanup()`, `set_vector()` 等。BSP 的开发人员也可以根据自己的需要重写这些函数。

12.6 系口初始化

设计 BSP 的第一步是设计处理器的复位/初始化代码。系统复位或者重启时，将首先执行系统初始化代码，这部分代码初始化硬件设备，它是所有 BSP 以及应用代码的基础，也是 BSP 中最接近硬件核心的代码之一。初始化代码负责初始化硬件平台的各个器件。虽然在不同的硬件平台上，初始化代码不同，但是这些代码的逻辑功能却是大同小异。一般来说，初始化被分成两部分完成初始化

- ☐ 调用 `rtems_initialize_executive_early` 初始化硬件以及内核组件
- ☐ `rtems_initialize_executive_late` 执行应用程序任务创建

处理器复位时，初始化代码就会执行。所有的设备必须在 RTEMS 初始化前进行必要的设置，这些初始化工作使用 BSP 的初始化模块完成。

由于 RTEMS 将会使用处理器的中断向量表，这些向量表必须在使用前初始化。由于中断是在 RTEMS 调用 `rtems_initialize_executive_early` 函数时启用的，所以必须在此以前正确的设置中断向量表以确保系统正确工作。此后 RTEMS 的初始化过程中，会使用 `rtems_interrupt_catch` 函数修改向量表。在一些处理器上面，初始化代码将会在初始化时设置自己的中断向量表，这样就不需要手工初始化。

在编写初始化代码的时候必须注意：

- 1) 初始化完成前不能调用 RTEMS 系统函数以及标准 C 库函数
- 2) 如果处理器支持多重优先级，那么必须让处理器运行在最高优先级上（对于 ARM 来说应该在 supervisory 态执行）
- 3) 必须为 `rtems_initialize_executive` 函数分配栈空间并且初始化栈指针
- 4) 必须初始化处理器的中断向量表
- 5) 必须屏蔽所有中断
- 6) 如果处理器支持多个中断堆栈，那么必须为每个栈分配空间并且初始化栈顶指针

`rtems_initialize_executive` 函数不返回到初始化代码，只是启动最高优先级的初始化任务，进行初始化本地和全局的应用。一般来说该任务将会创建应用所需要的所有任务。

本章将介绍 BSP 的初始化代码写法以及需要注意的事项。本章将包含 BSP 初始化中以下知识要点：

- ☐ 全局变量:
- ☐ 板初始化
- ☐ `boot_card()` 函数
- ☐ BSP 初始化函数 `bsp_start()`
- ☐ `main()` 函数
- ☐ RTEMS Pretasking 钩子
- ☐ RTEMS Predriver 钩子
- ☐ 驱动程序初始化
- ☐ Postdriver 钩子
- ☐ 中断向量表

- (1) 全局变量:

在一些系统中，会有全局变量需要初始化，这些变量的初始化在 BSP 目录的 `startup/bspstart.c` 中。这些变量为不同的 BSP 提供初始化支持。下面是需要初始化的全局变量列表：

- `BSP_Configuration` RTEMS 系统配置表
- `Cpu_table` RTEMS CPU 相关信息表.
- `bsp_isr_level` 系统启动时中断等级。当应用结束，返回 BSP 时，系统中断等级将恢复到这个设定的值。

下面是初始化的例子：

```
/*
 * 初始化 workspace 的起始地址
 */
```



```

BSP_Configuration.work_space_start = (void *)_WorkspaceBase;

/*
 * 初始化 CPU table
 */
Cpu_table.pretasking_hook = bsp_pretasking_hook; /* pretask 钩子初始化 libc 等 */
Cpu_table.postdriver_hook = bsp_postdriver_hook;
Cpu_table.interrupt_stack_size = 4096;
{
    extern void _BSP_Thread_Idle_body(void);
    Cpu_table.idle_task = _BSP_Thread_Idle_body;
}
Cpu_table.interrupt_vector_table = (m68k_isr *)0;

```

上面的例子中，初始化程序根据硬件设备情况初始化了 BSP_Configuration 和 Cpu_table 中对应的字段。

(2) 板初始化

BSP 目录中 start 目录下的汇编代码是整个系统中将最先执行的代码，它负责初始化处理器和嵌入式主板，以方便后续 BSP 代码执行，他完成下面的工作：

- 初始化堆栈
- 将.bss 段清零
- 屏蔽外部中断
- 将.data 段的数据从 ROM 拷贝到 RAM

汇编文件中完成的工作应该越少越好，尽可能将其他工作交给 C 完成。汇编的初始化完成后，他应该调用 boot_card()。初始化代码启动地址一般命名为 start，进行链接的时候需要将初始化汇编代码链接到.text 段的顶端（偏移为0）。这需要链接脚本和编译器协同完成。

下面是 csb337（at91rm9200）start.S 代码中的片断。

```

.text
.globl _start
_start:
/*
 * 初始化 CPSR
 */
mov    r0, #(PSR_MODE_SVC | PSR_I | PSR_F)
msr    cpsr, r0

/* 清零 bss 段 */
ldr    r1, =_bss_end_
ldr    r0, =_bss_start_

_bss_init:
mov    r2, #0
cmp    r0, r1
strlot r2, [r0], #4
blo    _bss_init    /* 循环执行，直到.bss 初始化结束 */

/* 初始化堆栈 */
/* 进入 IRQ 模式，初始化 IRQ stack 指针 */
mov    r0, #(PSR_MODE_IRQ | PSR_I | PSR_F)
msr    cpsr, r0
ldr    r1, =_irq_stack_size
ldr    sp, =_irq_stack
add    sp, sp, r1

/* 进入 FIQ 并且设置 FIQ 堆栈指针 */
mov    r0, #(PSR_MODE_FIQ | PSR_I | PSR_F)    /* 屏蔽中断 */
msr    cpsr, r0
ldr    r1, =_fiq_stack_size
ldr    sp, =_fiq_stack
add    sp, sp, r1

/* 进入 ABT 并且设置 ABT 堆栈指针 */
mov    r0, #(PSR_MODE_ABT | PSR_I | PSR_F)    /* 屏蔽中断 */

```

```

msr    cpsr, r0
ldr    r1, =_abt_stack_size
ldr    sp, =_abt_stack
add    sp, sp, r1

/*进入 UNDEF 并且设置 UNDEF 堆栈指针*/
mov    r0, #(PSR_MODE_UNDEF | PSR_I | PSR_F) /*屏蔽中断*/
msr    cpsr, r0
ldr    r1, =_undef_stack_size
ldr    sp, =_undef_stack
add    sp, sp, r1

/* 设置 SVC 堆栈指针, 并保持在此 SVC 模式进行后面的操作*/
mov    r0, #(PSR_MODE_SVC | PSR_I | PSR_F) /*屏蔽中断*/
msr    cpsr, r0
ldr    r1, =_svc_stack_size
ldr    sp, =_svc_stack
add    sp, sp, r1
sub    sp, sp, #0x64
.....

/*
 * 初始化向量表
 */
mov    r0, #0
adr    r1, vector_block
ldmia  r1!, {r2-r9}
stmia  r0!, {r2-r9}
ldmia  r1!, {r2-r9}
stmia  r0!, {r2-r9}

/* 执行 bootcard.c 中的 BSP 初始化 C 函数 */
bl     boot_card
.....

```

前面 linkcmd 章节中介绍过 image 的生成以及 .data 段的拷贝。上面的 start.S 并没有做这一块的工作。

(3) boot_card() 函数

汇编函数 start() 将会调用 boot_card() 函数，该函数是第一个 C 语言函数，通常，BSP 使用一个公用的 boot_card() 函数，该函数定义在 c/src/lib/libbsp/shared/bootcard.c 中。boot_card() 函数流程如下：

- 初始化 CPU 配置表中的字段为缺省值。
- 将应用的配置表(变量 Configuration) 拷贝到 BSP 配置表(变量 BSP_Configuration) 中并且修改 BSP_RTEMS_Configuration, BSP_Configuration 等配置表字段。
- 调用 BSP 相关的例程 bsp_start(),
- 调用 rtems_initialize_executive_early() 初始化系统, C 函数库, 设备驱动等, 此时并没有配置多任务, 也没有开发中断。
- 然后调用文件中的 main() 函数, main() 函数通常不会返回, 除非调用了 rtems_shutdown_executive()。
- 函数返回, 表示应用完成, 调用 BSP 相关的 bsp_cleanup() 关闭系统。

需要说明: 系统和环境必须在 main() 调用前完成初始化。

(4) BSP 初始化函数 bsp_start()

该函数是系统启动后执行的第一个 BSP 相关的 C 语言函数, 该例程实现硬件系统的初始化例如总线控制器寄存器初始化等。对应的代码一般在 c/src/lib/libbsp/CPU/BSP/startup/bspstart.c 中(有时候函数名称为 bsp_start_default), 这里 CPU 和 BSP 是开发者使用的 BSP 和处理器名称。该函数会进一步初始化 CPU 配置表并且设置和具体硬件 BSP 相关的字段。其中包含了增加 RTEMS 最大对象数目; 同时会将加载下面的初始化钩子函数:

- BSP Pretasking Hook
- BSP Predriver Hook
- BSP Postdriver Hook

bsp_start() 函数所作的最重要的工作是决定 RTEMS 在内存中的起始位置。RTEMS 所有的对象和任务堆栈都将从这个位置开始。RTEMS 工作空间和堆空间不同, 很多 BSP 将 RTEMS 放到内存空间的顶部(高地址处)。函数执行完后, 返回 boot_card()。

(5) main() 函数

该函数是 C 的入口函数, 代码如下。

```
int c_rtems_main(int argc, char **argv)
{
    if ((argc > 0) && argv && argv[0])
        rtems_prognam = argv[0];
    else
        rtems_prognam = "RTEMS";
    rtems_initialize_executive_late( bsp_isr_level );
    return 0;
}
```

在新发布的4.7.99.2 Beta 版本代码中，`c_rtems_main()`的功能已经合并到 `boot_card()`种，所以以后的代码可能就没有 `c_rtems_main()`了。`main()`的共享版本在文件 `c/src/lib/libbsp/shared/main.c` 中。

(6) RTEMS Pretasking 钩子

RTEMS CPU 配置表中的 `pretasking_hook` 字段允许用户自定义一些初始化函数。该函数的调用将会在 RTEMS API 初始化后，中断和多任务启动前的这样一个时间点。调用此函数的时候还没有任何任务(task)启动，所以钩子函数名称叫做 `pretasking_hook`。

一般来说，Pretasking 钩子是可选的，但是大多数 RTEMS BSP 提供了 `pretasking` 钩子。该例程通常被称作 `bsp_pretasking_hook`，位于下面的文件中：

`c/src/lib/libbsp/CPU/BSP/startup/bspstart.c`

`bsp_pretasking_hook()`例程一般用于初始化调试等级，RTEMS C 库。这些初始化过程一般会使用 RTEMS API。

`bsp_pretasking_hook()` 中的函数 `bsp_libc_init()` 如下所示：

```
void bsp_libc_init(
    void *heap_start,
    uint32_t heap_size,
    int use_sbrk
)
```

它传递了 C 库所需的堆空间大小和起始地址。`use_sbrk` 用于定义堆空间的增长长度。这只有在用户使用完所有的堆空间后才会增长。

(7) RTEMS Predriver 钩子

RTEMS CPU 配置表中的 `predriver_hook` 字段是用户定义的初始化函数地址。该初始化函数在设备驱动和 MPCPI 驱动初始化前完成。发生此调用的时候 RTEMS 的初始化已经结束，但是中断和多任务还没有启动。该字段可以为 NULL。

(8) 驱动程序初始化

初始化函数将会初始化设备驱动表中的所有设备驱动函数。驱动初始化的次序和他们在驱动表中的位置有关。驱动地址表是 RTEMS 配置表中的另外一个表，他定义了驱动入口函数（例如：`initialization`, `open`, `close`, `read`, `write` 和 `control`）

RTEMS 初始化将会依次调用每个驱动的初始化函数。此时需要设备的主设备号和从设备号。主设备号表明设备类型，从设备号用来区别同一类型中的不同设备。

(9) Postdriver 钩子

RTEMS CPU 配置表中的 `postdriver_hook` 字段指向用户定义的例程。该例程在驱动程序和 MPCPI 初始化后初始化。此时中断和多任务还未打开。该函数是可选的。一般来说在 `postdriver` 钩子中主要会检查驱动是否成功加载。

(10) 中断向量表

当 CPU 接受到外部设备或者总线发出的中断信号是，将根据中断向量表对应的条目进行跳转。跳转前，CPU 会进行上下文转换等工作。对于一些处理器中断的入口地址是固定的，此时 BSP 的工作比较少。

如果向量表条目可以动态改变，那么 RTEMS 需要手动进行分配。例如 ARM 处理中，中断向量表初始化如下：

```
VectorInit:
MOV R0, #0
ADR R1, Vector_Init_Block
LDMIA R1!, {R2, r3} /* Copy the Vectors (8 words) */
STMIA R0!, {r2, r3}
LDMIA R1!, {R2, r3} /* Copy the Vectors (8 words) */
STMIA R0!, {r2, r3}
LDMIA R1!, {R2, r3} /* Copy the Vectors (8 words) */
STMIA R0!, {r2, r3}
LDMIA R1!, {R2, r3} /* Copy the Vectors (8 words) */
STMIA R0!, {r2, r3}
```

```

LDMIA R1!, {R2, r3} /* Copy the .long'ed addresses (8 words) */
STMIA R0!, {r2, r3}
LDMIA R1!, {R2, r3} /* Copy the .long'ed addresses (8 words) */
STMIA R0!, {r2, r3}
LDMIA R1!, {R2, r3} /* Copy the .long'ed addresses (8 words) */
STMIA R0!, {r2, r3}
LDMIA R1!, {R2, r3} /* Copy the .long'ed addresses (8 words) */
STMIA R0!, {r2, r3}
B init2

```

通常，各个厂商的 ARM 处理器还有单独的中断控制器用于控制各种中断源（串口，SPI 等），这样在 BSP 代码中还需要对中断控制器初始化。

12.7 程序体系结构

设备驱动为输入 / 输出管理器提供了访问特定的外围设备并且进行管理控制的接口（对于不包含 RTEMS API 的应用，则可以通过驱动直接访问驱动接口）。RTEMS 的输入 / 输出管理器又为应用程序访问驱动程序提供了统一的逻辑接口。板支持包一般包含了访问目标硬件的驱动程序，常见的驱动包括串口驱动与并口驱动、实时时钟、定时器（看门狗）、IDE 接口、以太网，可擦写存储器(flash)等。

RTEMS 使用了统一的接口对硬件设备进行操作，RTEMS BSP 中设备驱动通常也使用了文件访问的接口（如 open, close, ioctl 等）。但是，需要说明的是，在 RTEMS 中，对于设备驱动的书写比 Linux 中更为灵活，由于 RTEMS 可以脱离文件系统独立运作，所以，也支持非文件系统接口模式的驱动程序。本节中介绍的驱动程序，将是文件系统访问接口模式的驱动程序为主。下面将以串口驱动程序（控制台驱动）为例子介绍 RTEMS 中驱动程序开发的基本技术。

（1）控制台驱动

RTEMS 中控制台驱动使用了 POSIX Termios 标准。控制台主要用于数据输入以及交互信息的 I/O。上层软件可以使用标准的 C 库对控制台进行操作。控制台有两种操作模式：

- （1） 控制台方式：基本的 telnet 方式进行交互，包含了特殊字符的处理
- （2） 数据 I/O Raw 模式：允许发送二进制数据

普通的串口设备驱动的接口如下所示：

```

typedef struct _console_fns {
    boolean (*deviceProbe)(int minor);
    int (*deviceFirstOpen)(int major, int minor, void *arg);
    int (*deviceLastClose)(int major, int minor, void *arg);
    int (*deviceRead)(int minor);
    int (*deviceWrite)(int minor, const char *buf, int len);
    void (*deviceInitialize)(int minor);
    void (*deviceWritePolled)(int minor, char cChar);
    int (*deviceSetAttributes)(int minor, const struct termios *t);
    int deviceOutputUsesInterrupts;
} console_fns;

```

驱动程序在整个操作系统中的位置可以参考下图20-4：

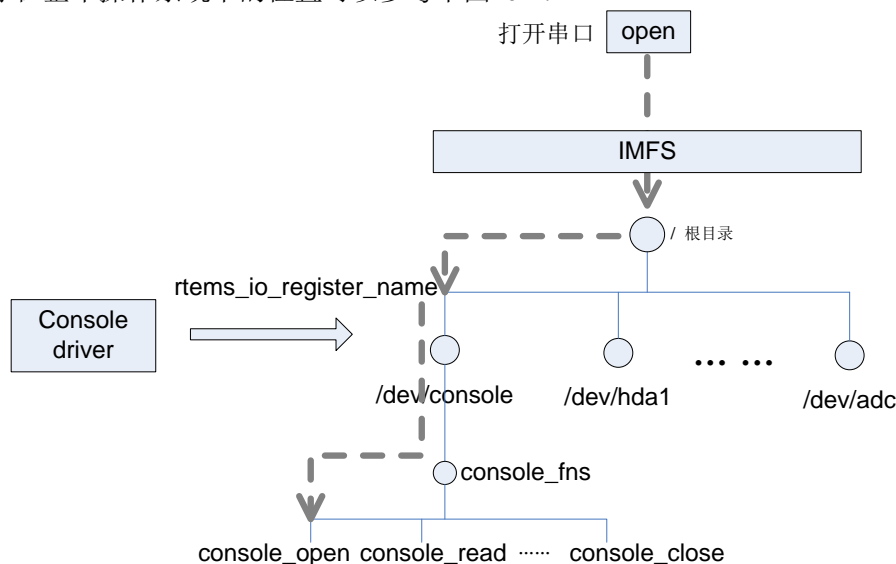


图15-5 Driver 层次结构

从图12-5可以看到，驱动程序使用 rtems_io_register_name() 函数在内存文件系统中创建文件节点

/dev/console，以后访问串口的时候就可以通过 IMFS 文件系统接口对串口操作。

对于串口终端，有两种模式：

- 1) 轮询模式
- 2) 中断模式

在轮询模式下，处理器阻塞的等待接受/发送的字符。对处理器的利用率不高。但是轮询模式对于在出现异常时进行错误输出很有帮助。这也是使用起来最方便的模式。如果是要对系统进行调试的话，这种模式是很合适的。

中断模式下，处理器不会在串口上阻塞，数据将会使用缓存区在中断处理和应用层之间传输。

RTEMS 中一般会使用双缓冲技术，一个缓存用于存放接受的字符串，一个用来处理输出字符串。当 UART 上接收到字符的时候，将会产生中断，中断处理程序将接收到的字符放入输入 FIFO 缓冲区中。当应用程序要获取缓存内容的时候，每次将读取缓存器首部信息。当系统需要通过串口向外部输出信息的时候，一个或者多个输出字符将会放到 FIFO 的输出缓存区的尾部。当字符放入后，产生 UART 中断，中断处理程序将传输缓冲区中所有的字符。

串口驱动程序的具体设计可以参考十一章中 termios 驱动的写作，本节就不特殊说明了。

(2) 时钟滴哒 (tick) 设备驱动

对于实时的应用，时钟驱动无疑是非常重要的。而且时钟驱动程序和串口驱动程序不同，时钟驱动是非文件系统模式的驱动程序。意味着时钟驱动程序中不再会有 open, close 这样的接口。大多数的 RTEMS 应用都会包括时钟驱动，该驱动会周期型的调用 rtems_clock_tick 函数。如果系统使用了时间片复用、定时器、日期管理、单调周期管理以及超时管理这些功能，就必须包含时钟驱动。

系统时钟 tick 通常使用计时器或者实时时钟 RTC 提供的中断来实现。当使用硬件定时器来提供 tick，那么定时器的的工作模式通常在驱动中设置为连续操作模式。该模式下，驱动将定期初始化计数器，然后自动进行倒计时。这样不但能减少处理器处理 ISR 的开销，还能提高时钟的精度。计数器的初始值通常是基于 RTEMS 配置表中的 microseconds_per_tick 字段为基础。此外，也可以将计数器初始化为一个固定的时间值（例如一微秒），然后让 ISR 在 microseconds_per_tick 到达时调用 rtems_clock_tick。不过这种方式的缺陷是如果时钟中断周期不是 microseconds_per_tick 的整数倍，就会产生误差。

需要说明，RTEMS 中时钟 tick 精度直接影响 RTEMS 中各种和时间相关操作的精度。但是同时，如果时钟的精度太高，系统执行时钟 ISR 的时间会比较多，系统开销也会增加。关于时钟驱动程序的讨论，也可以参考12.9时钟中断一节。

12.8 中断口理

在嵌入式系统中，包括两种中断，一种是硬件中断。指的是由硬件向处理器发送的异步信号，该信号请求处理器对硬件进行处理（例如传输数据等），软件中断则指的是使用软件方式模拟硬件中断的一种技术。下面，我们将主要介绍硬件中断的处理技术。硬件中断是让 CPU 能更有效相应 I/O 请求的一种技术。不同处理器都会有自己独特的中断处理流程。同时，每种处理器都提供了对中断处理和控制的的手段。在 BSP 的中断处理代码中，通常需要改变当前处理器的状态，并且在中断处理完成后，回到中断前的状态。在将 RTEMS 移植到新的系统中，需要对中断的机制有足够的了解。在 BSP 中，需要提供中断处理程序以及控制中断的 API。BSP 中其他设备，将使用这些 API 控制和使用系统中断。中断相关的代码在 BSP 代码中的 irq 目录中或者是 score/cpu/目录下对应处理器目录 cpu_asm.S 中。

以 ARM 处理器为例，对于中断处理程序，需要完成的工作包括：

- 1) 转到中断或者异常 CPU 对应的状态
- 2) 保存处理器状态寄存器，例如 ARM 处理器，需要保存 CPSR
- 3) 屏蔽其他中断
- 4) 将中断堆栈指针，程序计数器 PC，状态寄存器，链接寄存器 LR 等压栈
- 5) 转到中断向量对应的地址

IRQ 中断对应的中断处理函数入口是 _ISR_Handler()，_ISR_Handler() 函数是一个处理器家族公用的，属于 CSP 范畴。_ISR_Handler() 将保存 R0-R3, R12, R14(LR) 等通用寄存器，处理完寄存器备份以后，又会调用 BSP 定义的 ExecuteITHandler()。ExecuteITHandler() 函数的作用通常是对各种 ARM 类型处理器自己的中断控制器进行特殊处理，例如读取中断向量信息，来判断是哪个中断源产生的中断。

其他需要为操作系统提供中断控制的函数包括：

- ☐ BSP_rtems_irq_mngt_init() 初始化处理器的中断控制器
- ☐ isValidInterrupt() 判断中断是否来自有效的中断源

- ☐ BSP_install_rtems_irq_handler() 安装中断向量
- ☐ BSP_remove_rtems_irq_handler() 删除中断向量

12.9 □□中断

对于实时系统来说，获取和处理时钟信息重要性不言而喻。同时，对于一些对时间有高度要求的实时控制系统来说，更是需要获取高精度的时钟信息。所有这些时间信息的获取都源于时钟驱动程序。时钟驱动程序在 BSP 目录的 clock 目录中，下面将根据实例来介绍时钟驱动程序的写作。

RTEMS 为了方便程序员开发时钟驱动程序，采用了类似 C++ 模版的技术。时钟驱动中最常用的代码都放在了 libbsp/share/clockdrv_shell.c 中。然后，和真实硬件相关的代码则采用了宏定义的技术进行抽象。

在 clockdrv_shell.c 中，为操作系统提供了一个全局变量：

```
volatile rtems_unsigned32 Clock_driver_ticks;
```

该变量描述了自从系统启动以后经过的时钟 tick 数量。

同时另一个变量：

```
volatile uint32_t Clock_driver_ticks;
```

描述了上一次时钟中断以后经历的 tick 数量。

时钟的初始化，中断处理，时钟的控制在 clockdrv_shell.c 中都有实现。具体到每个 BSP 中，则需要实现下面的宏定义：

(1) clock_driver_support_initialize_hardware()

该宏函数需要实现时钟芯片/寄存器的初始化，例如清除已有的中断信息，设置时钟中断优先级，设置中断的周期长度等。

(2) clock_driver_support_at_tick

该宏定义代码需要实现每次时钟中断到来的时候，需要做的工作。主要的工作是清除中断寄存器信息，为下一次中断做准备。

(3) clock_driver_support_shutdown_hardware

用于关闭时钟中断。

(4) clock_isr_on/ clock_isr_off

用于开启/关闭时钟中断。

(5) clock_isr_is_on

判断中断是否打开。

(6) clock_driver_support_install_isr

用于安装时钟中断

例如下面的 csb337 代码：

```
/* Replace the first value with the clock's interrupt name. */
rtems_irq_connect_data clock_isr_data = { AT91RM9200_INT_SYSIRQ,
    (rtems_irq_hdl)Clock_isr,
    clock_isr_on,
    clock_isr_off,
    clock_isr_is_on,
    3, /* unused for ARM cpus */
    0 }; /* unused for ARM cpus */

#define Clock_driver_support_install_isr( _new, _old ) \
    BSP_install_rtems_irq_handler(&clock_isr_data)
```

代码为 clock_isr_data 赋值，并使用 irq 函数 BSP_install_rtems_irq_handler 安装了时钟中断。

12.10 移植到新的□处理器家族

对于嵌入式开发人员，有时候需要将操作系统移植到其他处理器。为了能正确地移植 RTEMS，需要程序员熟悉新 CPU 的体系结构，熟悉 automake 和 autoconf M4 脚本，并且能熟练进行远程调试。此外，在移植过程中，有一些代码需要使用汇编语言，所以也需要基于 gas 的汇编程序设计的能力。限于篇幅，本节将介绍移植的一些要点。

和处理器相关的代码放在目录 cpukit/score/cpu/下，目前 RTEMS 支持的处理器包括：

- * Motorola ColdFire

- * Motorola MC68xxx
- * Motorola MC683xx
- * Intel ix86 (i386, i486, Pentium and above)
- * ARM
- * ADI Blackfin
- * NOIS2
- * MIPS
- * PowerPC 4xx, 5xx, 6xx, 7xx, 8xx, and 84xx
- * SPARC
- * Hitachi H8/300
- * Hitachi SH
- * OpenCores OR32
- * Texas Instruments C3x/C4x

此外，目前也有一些处理器的支持正在进行中，例如 AVR/AVR32，Z80/Rabbit 等。此外，RTEMS 也提供使用 UNIX/Linux 系统 API 服务的模拟器 UNIX。

需要说明的是，cpukit/score/cpu/目录下面存放的代码和 libbsp 以及 libcpu 目录下面的代码不同，cpukit/score/cpu/下面的源代码是同类型处理器所公用的软件代码。例如 MC9328MXL 和 AT91RM9200 可以看成不同的处理器，但是他们都是 ARM9 家族的处理器，其中断屏蔽等代码是可以复用的。这些是可复用的，和处理器内核关系最紧密的代码就放到了 cpukit/score/cpu/目录下面。这种代码组织和设计结构能很好的满足嵌入式系统硬件种类多元化的特点。例如常见的 ARM 处理器的型号有上百种，但是这些处理器的内核都属于 ARM 公司，其基本的寄存器以及相关操作模式都是一样的。这样把与处理器内核相关的代码放在 cpukit/score/cpu/ 目录下面，具体不同型号的处理器，其 CSP 放在 libcpu 目录下面，和硬件板相关的代码就放在 libbsp 目录下面，可以满足不同层次的代码复用要求。

开始移植前，程序员需要了解 cpukit/score/cpu/内代码的作用。该目录下面的 no_cpu 是一个很好的起点。no_cpu 目录中的代码可以看成开发人员需要完成的代码模板。下面将介绍移植到新处理器时需要完成的工作：

1、 automake/autoconf 文件

创建自己移植代码的第一步就是创建代码目录，拷贝代码模块到对应目录，然后就是修改 automake/autoconf 等 M4 脚本文件。首先是 cpu 根目录下面的 Makefile.am，将你自己的处理器例如 avr32 加到 automake 搜索目录 DIST_SUBDIRS 中。然后在处理器的目录中修改 Makefile.am 和 Preinstall.am。前者定义需要编译的源代码文件以及目标文件，后者定义在编译过程中，那些头文件需要安装到系统 include 目录中。

2、 处理器 cpu_table 初始化

这里的处理器 cpu_table 定义如下：

```
typedef struct {
    /* RTEMS API 初始化时的 pretasking 钩子函数*/
    void    (*pretasking_hook)( void );
    /*设备驱动程序初始化时的 predriver 钩子函数*/
    void    (*predriver_hook)( void );
    /*设备驱动程序初始化结束时的 postdriver 钩子函数*/
    void    (*postdriver_hook)( void );
    /*空闲线程指针*/
    void    (*idle_task)( void );
    /*是否需要需要清零 workspace 内存空间*/
    boolean  do_zero_of_workspace;
    /*空闲线程的栈空间，缺省被定义为 MINIMUM_STACK_SIZE，也就是说在大多数处理器上是4K*/
    uint32_t idle_task_stack_size;
    /*中断使用的堆栈空间，缺省被定义为 MINIMUM_STACK_SIZE 4K*/
    uint32_t interrupt_stack_size;
    /*mpci 服务器线程的栈空间大小*/
    uint32_t extra_mpci_receive_server_stack;
    /*为用户线程分配运行栈时的钩子函数，必须和 stack_free_hook 一起配套使用*/
    void *   (*stack_allocate_hook)( uint32_t );
    /*为用户线程回收运行栈时的钩子函数，必须和 stack_allocate_hook 一起使用*/
    void    (*stack_free_hook)( void* );
} rtems_cpu_table;
```

上面的字段中，如果用户不初始化，RTEMS 会赋一个缺省值。所以初始化函数：

```
void _CPU_Initialize(
    rtems_cpu_table *cpu_table,
```

```
void (*thread_dispatch) /* ignored on this CPU */
)
```

在很多处理器的 CSP 中都是空函数。有些字段也可以在 BSP 的代码中初始化。这里需要说明的是 thread_dispatch 函数指针指向的函数是 ISR 结束后用于实现上下文切换的函数。这主要是用在某些编译环境下面不能在汇编语言中调用 C 函数，这个时候就需要在 _CPU_Initialize 函数中为 _CPU_Thread_dispatch_pointer 赋初始值：

```
_CPU_Thread_dispatch_pointer = thread_dispatch;
```

3、中断处理

RTEMS 是一个支持多级硬件中断的实时系统，目前支持的最大中断等级为256级。高优先级中断可以屏蔽低优先级中断。但是实际上很多处理器都不支持多级中断或者支持的中断级别比较少，例如 i386, ARM 等处理器。对于这种处理器，设置/获取中断优先级的操作实际上意义并不大。和中断等级相关的需要实现的函数包括：

- _CPU_ISR_Get_level 获取当前中断等级
- _CPU_ISR_Set_level(new_level) 设置中断优先级。
- _CPU_ISR_Disable(_isr_cookie) 屏蔽比 _isr_cookie 等级低的中断
- _CPU_ISR_Enable(_isr_cookie) 使能比 _isr_cookie 等级低的中断
- _CPU_ISR_Flash(_isr_cookie) 暂时打开然后又关闭中断。这个函数主要是用于长时间关闭中断的函数中，暂时打开中断可以对外部中断先响应，然后进行后面的处理。

RTEMS 对于中断堆栈管理也提供和很好的支持。目前有的处理器提供中断堆栈管理提供硬件支持，有的处理器则依赖于操作系统为中断提供支持。对于需要 RTEMS 分配中断堆栈的处理器，必须在 cpu.h 中将 CPU_HAS_SOFTWARE_INTERRUPT_STACK 和 CPU_ALLOCATE_INTERRUPT_STACK 两个宏定义为 TRUE。

中断安装是 CSP 代码必须处理的一个函数对于大多数支持中断向量的处理器，需要实现 _CPU_ISR_install_raw_handler/_CPU_ISR_install_vector 函数。主要目的是设置对应的中断控制器。

4、线程切换处理

线程切换是 RTEMS 移植过程中必须使用汇编语言实现的一个功能。通常实现都在 cpu_asm.S 中。其作用是在线程切换中换出的时候将状态寄存器以及通用寄存器的值存储到内存堆栈中，并且能将这些值在线程切换换入的时候将这些值读回寄存器。_CPU_Context_switch 和 _CPU_Context_restore 分别处理换出和换入的操作。对于一些有浮点寄存器的线程切换，可以定义浮点任务切换，专门对浮点寄存器进行处理。

5、空闲线程

空闲线程和统用操作系统中的空闲任务相同，主要是一个 while(1)循环，处理器通常不用做专门的处理。当然，程序员也可以根据自己的需要，让空闲线程做一些额外的工作，例如设置准备进入节电模式等。

6、优先级位图操作

在第四章的优先级管理中，提到了 RTEMS 的调度算法是一个基于多优先级的多级 FIFO 算法。这样，选择下一个要执行的任务就只需从优先级最高的 FIFO 就绪队列的队首取任务，而阻塞一个任务就只用将其放到 FIFO 的队尾。这样，也就不需要对任务队列进行遍历了。这样在大多数情况下面该算法的复杂度都是 O(1)，但是如果出现图12-6的情况，当换出的任务因为同步的原因进入了等待队列，而当前优先级的等待队列中没有其他就绪任务。这样，RTEMS 就需要在下一个优先级队列中搜索就绪任务，如果，还是没有，就需要再下一个优先级。

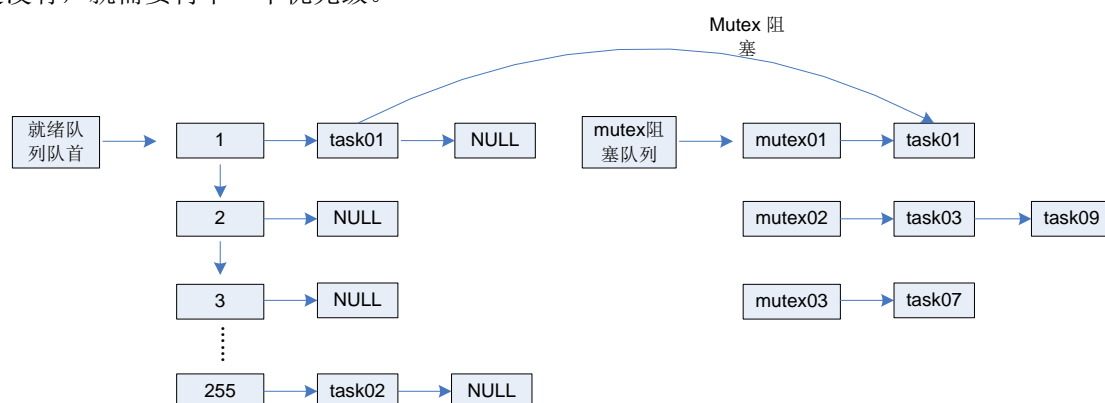


图12-6 寻找下一个就绪任务

这样，调度器就需要遍历整个就绪队列寻找下一个就绪任务，为了解决这个问题，RTEMS 使用了位

图操作来降低时延。数组_Priority_Bit_map 用于存放优先级比特位。这样寻找最高优先级的就绪任务就变成了寻找数组_Priority_Bit_map 中第一个不为零的比特。由于不同处理器对位操作的汇编语言定义不太一样，这样，RTEMS 给程序员一个重写寻找第一个不为零比特的机会。如果程序员觉得 C 语言写作的代码效率不高，可以使用汇编重新写一个宏_CPU_Bitfield_Find_first_bit。同样，程序员也可以写其他的优先级位图操作，例如设置位图的_CPU_Priority_Mask 以及将查找到的比特位映射成优先级队列的_CPU_Priority_bits_index。

7、其他功能

在 cpu.h 中还有一些和处理器相关的功能定义：

- CPU_STRUCTURE_ALIGNMENT 可以定义为32比特对齐，或者是8比特64比特等。
- CPU_ALIGNMENT 处理器对齐模式，通常设置为处理器支持的最大数据类型长度，CPU_HEAP_ALIGNMENT 和 CPU_PARTITION_ALIGNMENT 的值都被定义成该值。
- _CPU_Fatal_halt 用于定义处理器出错时的操作，通常可以输出出错信息，当前寄存器信息等。
- CPU_BIG_ENDIAN/CPU_LITTLE_ENDIAN 处理器大端小端，需要根据处理器内存操作方式进行定义。通常，就算是处理器同时支持大端与小端，也最好能只定义一个

12.11 RTEMS 剪裁与配置概要

一般来说，程序员在进行 RTEMS 应用程序开发的时候，使用 RTEMS 缺省的配置选项就能让 RTEMS 应用运行起来了。但是对于某些对内存要求比较苛刻的应用以及一些特殊应用，需要对应用程序进行手工或者自动配置。RTEMS 的配置的方法主要有下面几种：

- (1) 修改系统配置文件 confdefs.h 中宏定义的值。通常可以在应用程序中重新定义这些宏的值达到重新配置的目的。需要配置的信息包括：
 - 每个时钟 tick 的实际时间值
 - RTEMS 上能创建的对象数目
 - 应用初始化任务，和应用的设备驱动等等
- (2) 更改应用程序中需要的 Manager 数量。例如如果应用程序不使用 message-queue 组件，可以通过修改应用程序中的配置文件进行对于配置。
- (3) 程序员的自定义修改。这种方法一般只有在对系统比较熟悉，并且很清楚自己的应用程序需求时才会使用，这种方法有可能导致应用无法运行。

这些信息存放在 RTEMS 中对应的数据结构中。本章将介绍手工配置的要点，也会介绍如何进行简单的自动配置。

12.12 系口的串口配置

RTEMS提供 confdefs.h C语言头文件，该头文件包含了RTEMS系统配置使用的主要宏定义。该文件中包含了大量的宏定义，这些宏用来创建系统的配置表。confdefs.h中，已经定义了RTEMS系统配置的初始值，程序员可以定义需要设定的配置参数，这样就可以不用手动设置每一个配置表了。下面的例子中，进行了如下配置：

```
/*
 * system.h 用于重新定义 confdefs.h中的全局配置变量
 */
#include <rtems.h>

/* functions */

rtems_task Init(
    rtems_task_argument argument
);

/* 配置信息 */

#include <bsp.h> /* 驱动程序需要的头文件 */

#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
/*需要加载串口控制台驱动*/
```

```
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
/*需要加载时钟驱动*/
#define CONFIGURE_MICROSECONDS_PER_TICK 1000 /* 1 millisecond */
/*时钟单位为1ms*/
#define CONFIGURE_TICKS_PER_TIMESLICE 50 /* 50 milliseconds */
/*时间片50*/
#define CONFIGURE_MAXIMUM_TASKS 4
/*应用最大任务数量为4*/
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
/*将使用confdefs.h中任务初始化表的初始信息*/
```

这个系统将会使用任务 **Init** 为启动任务。同时配置成有一个控制台设备驱动 (为标准的输入 / 输出) 和一个时钟设备驱动的系统。

为了使confdefs.h中的配置生效, 一定要定义CONFIGURE_INIT常量,该常量在包含confdefs.h的应用程序中定义。

程序员应该注意, 系统配置的默认值一般都是尽可能的小。缺省情况下, 不会为应用分配太多资源。confdefs.h 文件确保至少有一个用户任务 (线程), 并且至少有一个初始化任务配置表。

confdefs.h 文件中将估算RTEMS 运行所需的内存。如果提供的信息不精确, 那么这个估算也会出现偏差。RTEMS通常会因为下面的原因为系统分配过多的内存:

- 所有的任务/ 线程被假定是浮点任务。

相反地,如果出现下面的情况, 则可能出现内存不够:

- 任务/ 线程堆栈空间不够
- 没有考虑到消息使用的内存
- 设备驱动的需求没有考虑进去
- 网络协议栈的内存需求没有考虑
- 附加函数库的内存需求没有考虑进去

当然, 如果程序员在confdefs.h中提供的信息准确, 那么confdefs.h的估算会非常精确。

RTEMS可以自动生成下面缺省配置:

- 函数库配置
- 基本系统信息
- 设备驱动表
- 标准API初始表

下列将介绍配置文件中的常量。

a) 函数库配置

和RTEMS相关的函数库包括libio, libc, 文件系统库等。以下主要介绍confdefs.h中文件系统I/O库相关的常量:

CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS: 可以同时打开的文件数目。I/O库中, 每打开一个文件就需要为其分配一个互斥变量, 所以该配置会改变系统的内存需求。

CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS的默认值为3, 也就是支持标准的输入, 输出和错误输出三个文件描述符。

CONFIGURE_TERMIOS_DISABLED: 程序中是否使用POSIX的termios功能, 缺省为支持。

CONFIGURE_NUMBER_OF_TERMIOS_PORTS 终端数目, 缺省为1。

CONFIGURE_HAS_OWN_MOUNT_TABLE: 是否使用用户自定义得文件系统安装表。缺省情况下用户是没有 (也不需要) 自己的文件系统安装表的, 此时, RTEMS就会定义的 `rtems_filesystem_mount_table_t` 类型的变量 `rtems_filesystem_mount_table`, 该变量将加载IMFS/MiniIMFS为缺省文件系统。

CONFIGURE_USE_IMFS_AS_BASE_FILESYSTEM 使用 IMFS 为缺省文件系统。这是默认选项, 否则可以通过 `#define CONFIGURE_USE_MINIIMFS_AS_BASE_FILESYSTEM` 来使用 miniIMFS 为基本文件系统。

STACK_CHECKER_ON 是否对堆栈越界进行动态检查, 缺省为不检查。

b) 基本系统信息

confdefs.h中定义的系统参数如下:

- **CONFIGURE_HAS_OWN_CONFIGURATION_TABLE** 系统是否使用自己定义的配置表

- `CONFIGURE_INTERRUPT_STACK_MEMORY` 中断堆栈大小，缺省大小为 `RTEMS_MINIMUM_STACK_SIZE`，这个值在大多数系统中是4K。
- `CONFIGURE_EXECUTIVE_RAM_WORK_AREA` RTEMS基准地址，缺省为NULL，表示由BSP判断。
- `CONFIGURE_MICROSECONDS_PER_TICK` 每个 tick 实际的微秒数。
- `CONFIGURE_TICKS_PER_TIMESLICE` 每个时间片中的tick数目，缺省为50。
- `CONFIGURE_MEMORY_OVERHEAD` 为系统追加的内存数目，默认价值是 0
- `CONFIGURE_EXTRA_TASK_STACKS` 为系统追加的堆栈数目，默认价值是 0，如果系统使用的堆栈大于 `confdefs.h` 计算出的堆栈数目，那么就可以增大该值。

c) 设备驱动表

自动生成设备驱动表格需要的配置参数。当然这些配置选项只适用于标准设备驱动。

`CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE`: 如果定义了自己的非标准设备驱动表，就需要打开此开关。一般情况下，程序员可以选择使用RTEMS的驱动设备表，此时会生成下面的全局变量。

```
rtems_driver_address_table Device_drivers[] = {
#ifdef CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
    CONSOLE_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
    CLOCK_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER
    RTC_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER
    DEVNULL_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_IDE_DRIVER
    IDE_CONTROLLER_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_ATA_DRIVER
    ATA_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_NULL_DRIVER
    NULL_DRIVER_TABLE_ENTRY
#elif !defined(CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER) && \
    !defined(CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER) && \
    !defined(CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER) && \
    !defined(CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER) && \
    !defined(CONFIGURE_APPLICATION_NEEDS_IDE_DRIVER) && \
    !defined(CONFIGURE_APPLICATION_NEEDS_ATA_DRIVER)
    NULL_DRIVER_TABLE_ENTRY
#endif
};
```

上面的初始化中，`CONFIGURE_MAXIMUM_DRIVERS`表示驱动的最大数目，缺省为10。

`CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER` 是否使用控制台驱动，如果定义，那么该设备驱动负责提供标准的输入和输出使用文件 `"/dev/console"`。缺省状况该值不被定义。

`CONFIGURE_APPLICATION_NEEDS_TIMER_DRIVER` 是否需要时钟驱动，缺省被屏蔽。

`CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER` 是否适用桩驱动，该驱动实际上就是UNIX/Linux中的/dev/null设备，缺省是不需要的。

可以看到，缺省的系统驱动表也是静态配置的。程序开发人员在系统定义阶段定义需要的驱动，然后增加对应的预处理定义，然后系统会根据配置进行各个驱动程序的初始化。当然，用户也可以根据自己系统实际状况定义自己的驱动设备表，例如下面的system.h文件：

```
/*system.h, 用户配置文件，必须在末尾包含#include <confdefs.h> */
.....
#define CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE
rtems_driver_address_table Device_drivers[] = {
    CONSOLE_DRIVER_TABLE_ENTRY,
    CLOCK_DRIVER_TABLE_ENTRY,
    TTY1_DRIVER_TABLE_ENTRY,
    TTY2_DRIVER_TABLE_ENTRY,
    {NULL, NULL, NULL, NULL, NULL, NULL}
};
.....
```

```
#include <confdefs.h>
```

上面的例子中，系统的设备表一共有五个，包括console，clock，两个TTY设备，以及一个空表项。

d) 标准API初始表

confdefs.h 能够自动产生初始化任务表Initialization_tasks，下面是控制表自动创建的参数。

CONFIGURE RTEMS_INIT_TASKS_TABLE 如果用户愿使用标准RTEMS API初始化任务表。任务一般会通过其他任务进行初始化，所以这个字段缺省为未定义。

CONFIGURE_HAS_OWN_INIT_TASK_TABLE有自定义的初始表，缺省没有。

CONFIGURE_INIT_TASK_NAME 初始任务名称，缺省值为rtems_build_name。(‘U’，‘T’，‘I’)

CONFIGURE_INIT_TASK_STACK_SIZE 任务栈大小，缺省为 RTEMS_MINIMUM_STACK_SIZE。

CONFIGURE_INIT_TASK_PRIORITY 初始任务的优先级，缺省为1。

CONFIGURE_INIT_TASK_ATTRIBUTES 缺省的初始化任务属性，缺省为 RTEMS_DEFAULT_ATTRIBUTES。

CONFIGURE_INIT_TASK_ENTRY_POINT 初始任务，缺省为Init。

CONFIGURE_INIT_TASK_INITIAL_MODES 初始任务执行模式，初始为RTEMS_NO_PREEMPT。

CONFIGURE_INIT_TASK_ARGUMENTS 初始任务参数，初始值是0。

12.13 RTEMS 配置表

在 RTEMS 中通过各种数据结构表格来配置操作系统。常用的配置表包括：

- a) RTEMS 系统配置表
- b) RTEMS API 配置表
- c) 处理器信息表
- d) 驱动地址表格
- e) 用户扩展表
- f) 多处理器配置表

下面分别介绍这些表的使用。

(1) RTEMS系统配置表

RTEMS 配置表用于对应用进行剪裁。例如配置最大任务数目等。配置表的地址将传递给 rtems_initialize_executive 函数，配置表定义如下：

```
typedef struct{
    void                *work_space_start;
    rtems_unsigned32    work_space_size;
    rtems_unsigned32    maximum_extensions;
    rtems_unsigned32    microseconds_per_tick;
    rtems_unsigned32    ticks_per_timeslice;
    rtems_unsigned32    maximum_devices;
    rtems_unsigned32    maximum_drivers;
    rtems_unsigned32    number_of_device_drivers;
    rtems_driver_address_table *Device_driver_table;
    rtems_unsigned32    number_of_initial_extensions;
    rtems_extensions_table *User_extension_table;
    rtems_multiprocessing_table *User_multiprocessing_table;
    rtems_api_configuration_table *RTEMS_api_configuration;
    posix_api_configuration_table *POSIX_api_configuration;
}rtems_configuration_table;
```

表项内容可以根据名称判断，该表在 confdefs.h 中被初始化为：

```
rtems_configuration_table Configuration = {
    CONFIGURE_EXECUTIVE_RAM_WORK_AREA,
    CONFIGURE_EXECUTIVE_RAM_SIZE,
    CONFIGURE_MAXIMUM_USER_EXTENSIONS + CONFIGURE_NEWLIB_EXTENSION +
        CONFIGURE_STACK_CHECKER_EXTENSION,
    CONFIGURE_MICROSECONDS_PER_TICK,
    CONFIGURE_TICKS_PER_TIMESLICE,
    CONFIGURE_MAXIMUM_DRIVERS,
    CONFIGURE_NUMBER_OF_DRIVERS,          /* driver的数目 */
    Device_drivers,                        /* 指向driver表的指针 */
    CONFIGURE_NUMBER_OF_INITIAL_EXTENSIONS, /* 初始化扩展的数目 */
    CONFIGURE_INITIAL_EXTENSION_TABLE,     /* 指向初始化扩展的指针 */
}
```

```

CONFIGURE_MULTIPROCESSING_TABLE,      /* 指向多处理器表的指针 */
&Configuration_RTEMS_API,             /* 指向RTEMS API 配置表的指针 */
#ifdef RTEMS_POSIX_API
&Configuration_POSIX_API,             /* 指向POSIX API配置表的 */
#else
NULL,                                  /* 指向POSIX API配置表的指针设置为空 */
#endif
#ifdef RTEMS_ITRON_API
&Configuration_ITRON_API              /* 指向ITRON API配置表的指针 */
#else
NULL                                  /*指向ITRON API配置表的指针设置为空*/
#endif
};
#endif

```

在实际的 BSP 中，一般需要将 work_space_start 和 work_space_size 等信息重新初始化。

(2) RTEMS API 配置表

RTEMS API配置表为RTMES中使用到的API进行配置。其数据结构定义如下:

```

typedef struct{
    uint32_t      maximum_tasks;
    /*最大任务数目，由CONFIGURE_MAXIMUM_TASKS定义，缺省为10*/
    uint32_t      maximum_timers;
    /*最大timer数目，由CONFIGURE_MAXIMUM_TIMERS定义，缺省为0，表示没有限制，下同*/
    uint32_t      maximum_semaphores;
    /*最大semaphore数目，由CONFIGURE_MAXIMUM_SEMAPHORES定义，缺省为0*/
    uint32_t      maximum_message_queues;
    /*最大message queue数目，由CONFIGURE_MAXIMUM_MESSAGE_QUEUES 定义，缺省为0*/
    uint32_t      maximum_partitions;
    /*最大partition数目，由CONFIGURE_MAXIMUM_PARTITIONS定义，缺省为0*/
    uint32_t      maximum_regions;
    /*最大region数目，由CONFIGURE_MAXIMUM_REGIONS定义，缺省为0*/
    uint32_t      maximum_ports;
    /*最大port数目，可同时使用的双口存储器存储区*/
    uint32_t      maximum_periods;
    /*最大period数目，用于单调周期调度*/
    uint32_t      maximum_barriers;
    /*最大barrier数目，由CONFIGURE_MAXIMUM_BARRIERS定义*/
    uint32_t      number_of_initialization_tasks;
    /*初始化任务数量，缺省为0*/
    rtems_initialization_tasks_table *User_initialization_tasks_table;
    /*任务初始化表头地址*/
}rtems_api_configuration_table;

```

其中需要说明的是初始化API主要指RTEMS,ITRON,POSIX等API，对于RTEMS应用来说，一般需要至少一个API，如果用户只希望使用RTEMS的API，那么可以使用系统缺省配置。系统初始化任务配置表 rtems_initialization_tasks_table 的信息包括任务名称，任务堆栈大小，优先级，入口函数，是否会被抢占调度等信息。其定义如下：

```

typedef struct {
    rtems_name      name;
    rtems_unsigned32 stack_size;
    rtems_task_priority initial_priority;
    rtems_attribute attribute_set;
    rtems_task_entry entry_point;
    rtems_mode      mode_set;
    rtems_task_argument argument;
} rtems_initialization_tasks_table;

```

下面是一个实际配置的例子：

下面是设置的实例：

```

rtems_initialization_tasks_table
Initialization_tasks[2] = {
    { INIT_1_NAME,
      1024,
      1,
      DEFAULT_ATTRIBUTES,
      Init_1,
      DEFAULT_MODES,

```

```

    1
},
{ INIT_2_NAME,
  1024,
  250,
  FLOATING_POINT,
  Init_2,
  NO_PREEMPT,
  2
}
};

```

(3) 处理器信息表

处理器信息表用来描述处理器相关的信息。其中的信息是由 BSP 提供的。confdefs.h 无法自动产生该表，需要手工配置。

(4) 驱动地址表格

设备驱动表格是用向输入 / 输出管理器提供驱动信息。表格中包含了驱动需要的入口条目。其定义如下：

```

typedef struct {
    rtems_device_driver_entry initialization;
    rtems_device_driver_entry open;
    rtems_device_driver_entry close;
    rtems_device_driver_entry read;
    rtems_device_driver_entry write;
    rtems_device_driver_entry control;
} rtems_driver_address_table;

```

下面是设置的实例：

```

rtems_driver_address_table Driver_table[2] = {
    { tty_initialize, tty_open, tty_close, /* major = 0 */
      tty_read,    tty_write, tty_control
    },
    { lp_initialize, lp_open,  lp_close, /* major = 1 */
      NULL,        lp_write, lp_control
    }
};

```

(5) 用户扩展表

用户扩展多用来对系统进行监控，此外也可以处理其他系统事件。例如线程切换时，用户扩展可以为非标准数字协处理器提供上下文切换服务。此时，任务的建立和删除扩展被用来分配与删除任务协处理器上下文，任务初始化扩展用来初始化协处理器上下文区域，任务切换扩展用来处理任务切换。用户扩展表格是用来告知 RTEMS 有哪些可选择的，由用户提供的静态扩展集。该表格为每个可能的扩展建立一个入口。对应事件发生时，会调用对应的扩展函数，扩展的定义如下：

```

typedef struct {
    rtems_task_create_extension    thread_create;
    rtems_task_start_extension    thread_start;
    rtems_task_restart_extension  thread_restart;
    rtems_task_delete_extension   thread_delete;
    rtems_task_switch_extension   thread_switch;
    rtems_task_begin_extension    thread_begin;
    rtems_task_exitted_extension  thread_exitted;
    rtems_fatal_extension         fatal;
} User_extensions_Table;

```

下面是设置的实例：

```

rtems_extensions_table User_extensions = {
    task_create_extension,
    NULL,
    NULL,
    task_delete_extension,
    task_switch_extension,
    NULL,
    NULL,
    fatal_extension
};

```

(6) 多处理器配置表

当系统为多处理器系统时，多处理器配置表包含了多处理器的配置信息。多处理器配置表的地址应该被放在表 User_multiprocessing_table 中。

使用confdefs.h 配置多处理RTEMS 应用，必须定义 CONFIGURE_MP_APPLICATION。
多处理器格局表格的格式在下列的 C语言结构中被定义:

```
typedef struct{
    uint32_t node;
    uint32_t maximum_nodes;
    uint32_t maximum_global_objects;
    uint32_t maximum_proxies;
    rtems_mpci_table*User_mpci_table;
}rtems_multiprocessing_table;
```

当用 confdefs.h 配置应用，通信接口表的缺省名称为MPCI_table。
该表格的格式在下列的 C语言结构中被定义:

```
typedef struct {
    uint32_t      default_timeout; /* in ticks */
    uint32_t      maximum_packet_size;
    rtems_mpci_initialization_entry initialization;
    rtems_mpci_get_packet_entry   get_packet;
    rtems_mpci_return_packet_entry return_packet;
    rtems_mpci_send_entry         send;
    rtems_mpci_receive_entry      receive;
} rtems_mpci_table;
```

12.14 内存需求

由于内存是嵌入式系统的稀缺资源。RTEMS允许暂时不用的管理器释放占用内存资源。这就允许开发者能灵活的剪裁RTEMS以适应小内存系统的需要。但是，必须注意，RTEMS的内存需求是和应用相关的。为了保证程序运行时有足够的内存，程序员应该有效的估算各个模块的内存使用。系统中下面的API可以被剪裁:

- 信号
- 可变内存区
- 双口内存
- 事件
- 多处理器处理
- 固定内存区
- 定时器
- 信号量
- 消息
- 单调周期

RTEMS被设计成可以链接的函数库。由于程序的模块化设计，RTEMS允许链接时不链接未使用的库函数，从而减少目标代码的大小。

(1) 内存空间基本要求

RTEMS系统必须为应用提供内存空间，虽然RTEMS的数据段必须存放在RAM中，但是代码段可以选择存放在ROM中或者RAM中。此外也必须为系统分配RAM工作空间(workspace)，工作空间的大小可以根据对应的处理器文档计算。

下面是i386的内存需求:

数据(.data)段 833字节

内核: 全 39.592字节

内核: 精简: 22.660字节 (只包括初始化, 中断和异常管理)

代码需要的存储空间

表12-1 各个组件内存使用一览表

| 组件 | 包含后占用存储空间 | 最小使用空间 |
|----------------|-----------|--------|
| Core | 16,948 | 必须包含 |
| Initialization | 916 | 必须包含 |
| Task | 3,436 | 必须包含 |
| Interrupt | 52 | 必须包含 |
| Clock | 296 | 必须包含 |

| | | |
|--------------------|-------|------|
| Timer | 1,084 | 144 |
| Semaphore | 1,500 | 136 |
| Message | 1,596 | 224 |
| Event | 1,036 | 44 |
| Signal | 396 | 44 |
| Partition | 1,052 | 104 |
| Region | 1,392 | 124 |
| Dual Ported Memory | 664 | 104 |
| I/O | 676 | 00 |
| Fatal Error | 20 | 必须包含 |
| Rate Monotonic | 1,132 | 136 |
| Multiprocessing | 6,840 | 228 |

从上表可以看出来，有些组件是必须包含的，对于Event，Signal这样的组件，则可以根据实际的情况进行剪裁，使用空桩模块代替原来的模块，这些空桩模块大概需要占用一百字节左右的空间。

(2) RTEMS系统工作空间内存需求

RTEMS中系统工作空间（workspace）是为系统预留的内存区，主要包含RTEMS数据结构，例如系统配置信息，API配置信息等。此外，还包含任务栈和浮点上下文等信息。

RTEMS可以通过confdefs.h中的配置自动计算RTEMS需要的工作空间大小。confdefs将所有的任务都看成浮点任务，为其分配尽可能小的栈空间。此外还计算RTEMS内核自身的内存消耗。下列的定义可以使弥补计算的不足：

☐ CONFIGURE_MEMORY_OVERHEAD 定义内存的额外开销，以 KB 为单位

☐ CONFIGURE_EXTRA_TASK_STACKS 任务堆栈额外开销

RTEMS数据存储的起始地址必须是四字节对齐，否则会引起异常。

下面是i386体系的workspace的计算方法。

表12-2 RTEMS Workspace 内存空间计算表

| 说明 | 乘法倍率 | 需要内存数量 |
|------------------------|---------------------|--------|
| Maximum_tasks | * 372 = | |
| Maximum_timers | * 68 = | |
| Maximum_semaphores | * 124 = | |
| Maximum_message_queues | * 148 = | |
| Maximum_regions | * 144 = | |
| Maximum_partitions | * 56 = | |
| Maximum_ports | * 36 = | |
| Maximum_periods | * 36 = | |
| Maximum_extensions | * 64 = | |
| Floating Point Tasks | * 108 = | |
| Task Stacks | 由应用程序定 ¹ | |
| 单处理器系统内存需求 | | |
| Maximum_nodes | * 48 = | |
| Maximum_global_objects | * 20 = | |
| Maximum_proxies | * 124 = | |
| 多处理器系统内存需求 | | |
| 固定需要内存大小 | 6,768 | |
| 单处理器系统内存需求 | | |
| 多处理器系统内存需求 | | |

说明1：上表中任务栈空间需求(Task Stacks)一般不同系统给出定义不同，例如 ARM 系统中，缺省的任务堆栈空间为4K，实际上，应用程序也可根据实际应用的复杂程度来修改栈空间的需求。下面的例子将假设栈空间大小为4K。

内存的需求计算方法如下所示：

可以看到，每增加一个任务，需要多372个字节，有了这些计算公式，系统使用的内存可以量化到字

节为单位。如果你的应用需要10个任务，5个 semaphore，那么需要的 workspace 空间大小就是 $10 \times 372B + 5 \times 124B + 10 \times 4KB = 44.2KB$ 。

- 1) 需要说明，在实际的应用中，实际需要的内存可能会超过通过使用表计算出来的内存容量，主要是上表中没有考虑下面的因素：
- 2) BSP 中内存的需求。不同 BSP 对内存的需求是不一样的，有以太网以及 USB 等外设的系统的 BSP，其内存需求通常大于只有 UART 的系统。
- 3) 扩展库。通常，应用程序不单单包括内核，都会使用各种函数库，最常见的例如 libc, 以及 libnetworking 等，这些库通常都会占用一些内存。
- 4) 文件系统。无论是 IMFS 还是用户自定义的文件系统，都会占用数量可观的内存。

通常，考虑了上面所有的因素，通常就能正确地估计真实的内存需求。不过对系统最精确的估算往往来自于 objdump。Objdump 通常能准确地反映系统对静态分配内存的需要。加上计算表对动态内存的估计，通常能让开发人员对于系统内存需求有一个基本的概念。

对象的分配可以使用两种模式。默认模式是规定对象数目的上限。对象在初始化时分配好需要的内存，系统运行过程中系统对象的数目不能超过配置的数目。

第二种模式允许系统动态增加对象的数目，这通过配置 rtems_resource_unlimited 实现。例如：
#define CONFIGURE_MAXIMUM_TASKS rtems_resource_unlimited(5)

rtems_resource_unlimited 定义的值是对象表的大小，当系统需要动态为新对象分配空间，每次分配固定大小的空间，空间的大小和 rtems_resource_unlimited 成正比。对象使用完毕后，内核将这些内存释放。

一般来说每个版本的 RTEMS 对内存的需求将会是不同的。而且对于不同的处理器，不同的编译器，内存的需求都可能发生变化。为了避免问题，在如下情况下应该重新计算内存需求：

1. 修改了配置参数，
2. 任务或中断栈需求改变，
3. 任务的浮点属性被改变，
4. RTEMS 版本变化
5. 目标处理器被改变

如果不能为 RTEMS 提供足够的工作空间将会产生异常。

12.15 应用程序配置示例

最后来看看在应用程序层面如何配置 RTEMS。

通常，编译好的 RTEMS 库有近 20 兆，但是最后生成的可执行文件可以控制在几十 K 字节，这主要归功于 RTEMS 的编译系统。通常来说，使用缺省的配置选项就能得到一个足够小的目标二进制代码。如果用户希望能进一步剪裁，就需要修改 Makefile.am 来配置 RTEMS 库函数的连接方式。以 samples 目录里面的 hello 为例，其 Makefile.am 如下：

```
##
## $Id: Makefile.am,v 1.24 2007/05/11 19:45:50 joel Exp $
##

MANAGERS = io semaphore

rtems_tests_PROGRAMS = hello.exe
hello_exe_SOURCES = init.c system.h

dist_rtems_tests_DATA = hello.scn
dist_rtems_tests_DATA += hello.doc

include $(RTEMS_ROOT)/make/custom/@RTEMS_BSP@.cfg
include $(top_srcdir)/../automake/compile.am
include $(top_srcdir)/../automake/leaf.am

hello_exe_LDADD = $(MANAGERS_NOT_WANTED:%=$(PROJECT_LIB)/no-%.rel)

LINK_OBJS = $(hello_exe_OBJECTS) $(hello_exe_LDADD)
LINK_LIBS = $(hello_exe_LDLIBS)

hello.exe$(EXEEXT): $(hello_exe_OBJECTS) $(hello_exe_DEPENDENCIES)
    @rm -f hello.exe$(EXEEXT)
    $(make-exe)

include $(top_srcdir)/../automake/local.am
```

这里需要注意的是：

```
MANAGERS = io semaphore
```

和

```
hello_exe_LDADD = $(MANAGERS_NOT_WANTED:%=$(PROJECT_LIB)/no-%.rel)
```

前者定义了hello.exe将诸多管理器中的io和semaphore链接到最后的可执行代码中，而

hello_exe_LDADD的定义则说明库函数是如何链接的。下面我们看看实际的编译链接过程(以i386 BSP为例)：

```
i386-rtems4.8-gcc -B../pc586/lib/ -specs bsp_specs -qrtems -mtune=pentium -O2 -g -Wl,-Ttext,0x00100000 -o  
hello.elf init.o ../pc586/lib/no-dpmem.rel ../pc586/lib/no-event.rel ../pc586/lib/no-  
msg.rel ../pc586/lib/no-mp.rel ../pc586/lib/no-part.rel ../pc586/lib/no-  
signal.rel ../pc586/lib/no-timer.rel ../pc586/lib/no-rtmmon.rel
```

可以看到在生成hello.elf的过程中，链接了no-dpmem.rel，no-event.rel，no-msg.rel等库而不是dpmem.rel，event.rel，这些库实际上是optman目录下面的库，在介绍optman目录的时候介绍过，该目录下面定义的一些空函数。如果hello.exe不需要使用dpmem，event等管理器的话，可以通过修改Makefile.am来控制链接库的数量。在编译成功后，可以通过编译目录下面的hello.num文件来确认是否链接到空函数。

常用的可配置管理器包括io，event，message，rate_monotonic，semaphore，timer，如果希望全部包含，可以使用

```
MANAGERS = all
```

本章介绍了RTEMS的BSP开发的基本知识以及移植与剪裁的基本知识。由于篇幅原因，本章并没有对每个知识点都进行深入的介绍。读者可以参考各个BSP的移植代码已经内核代码获取更深刻的理解，此外RTEMS的在线文档也可以提供大量更进一步的介绍。当然熟悉这些知识最好的方法莫过于为让RTEMS跑在您自己的硬件上。