RTEMS C 语言用户参考手册

孙永兵 译 email: bradon_syb@hotmail.com

1.1 前言

RTEMS,多进程实时操作系统,是真正的实时操作系统可以为多种应用提供高性能的嵌入式环境。它有如下的特征：

多任务特性

支持同类或不同类（homogeneous and heterogeneous）的多处理器系统

事件驱动，优先级为基础，占先式调度

可选择的单调速率调度（RMS）

多任务间的通讯和同步

优先级继承和优先级置顶

响应中断管理

动态内存分配

很高级别的用户可配置性

1.2 实时系统 RTEMS

实时应用系统是一种非常复杂的计算机应用，它有一系列特性将它与普通的应用区分，它必须可以适应一些苛刻的需求。系统的正确性不仅仅依赖于结果的正确，而且还依赖于结果返回的时间。

有两个重要的概念：

Deadline，任务必须要在 Deadline 到来之前执行结束。

处理同时发生的任务的能力。

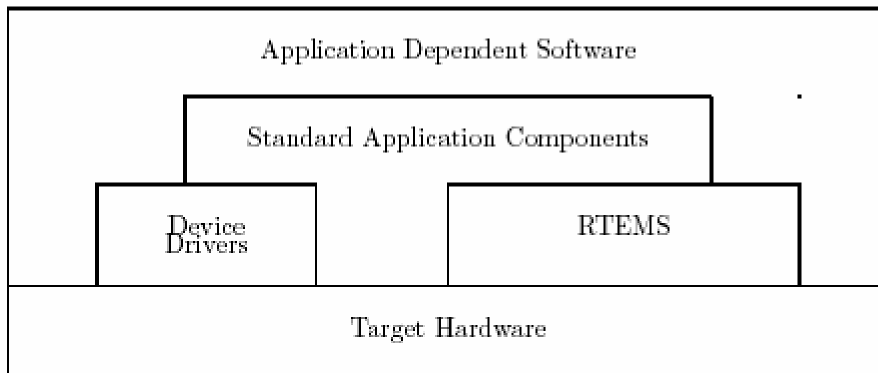
1.3 实时执行

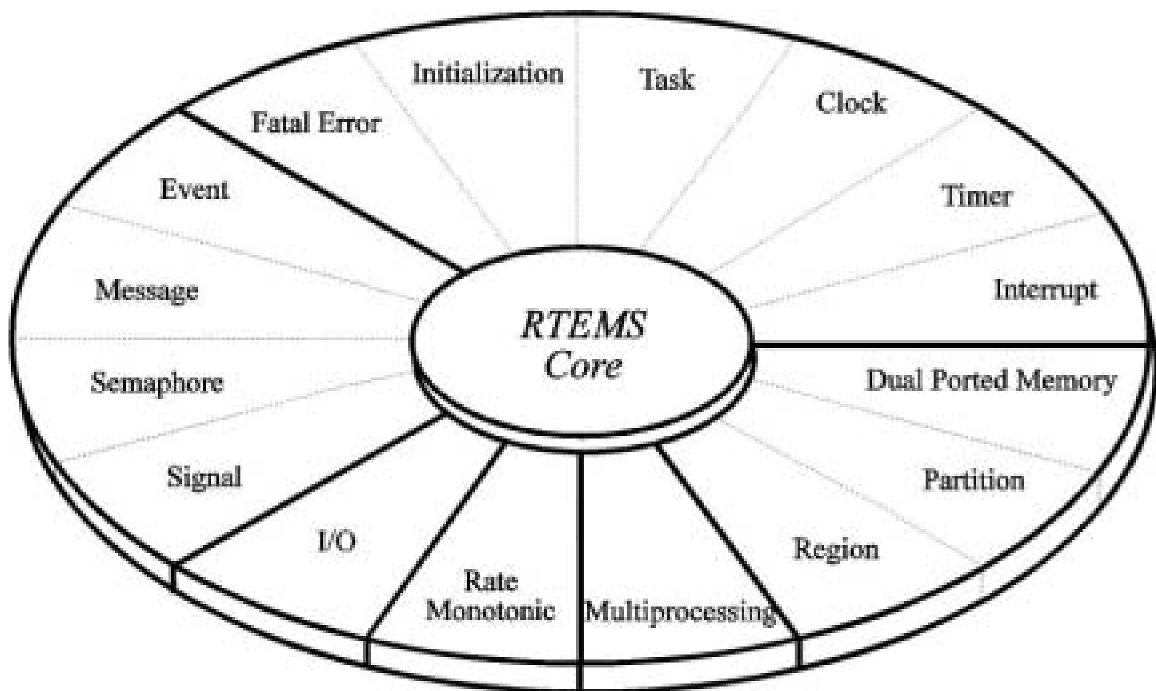
使用 RTEMS 提供的命令，实时应用的开发者不需要去考虑控制，同步多任务和进程。另外，不需要开发，测试，调试和记录日志来管理内存，传送消息或者提供互斥。开发者这样就可以集中精力在应用上了。通过使用标准的软件部分，我们将可以大大减少用于开发的时间和精力。

1.4 RTEMS 的体系结构

RTEMS 的一个重要的设计特征就是在一个实时应用的两层中间提供一个桥。RTEMS 就象一个在应用代码和目标硬件中的缓冲区。RTEMS 的 I/O 接口管理提供一个有效的工具将这些硬件依赖融合在一个系统中，同时为用户的应用提供一种通用的机制。一个设计良好的的实时系统可以从这体系结构中获得好处，通过建立丰富的标准应用库来可以重复使用在不同的实时应用中。

RTEMS 体系结构





1.5 RTEMS 的内部体系结构

RTEMS 共有 17 个管理器,分别是**初始化(initialization),任务(task),**时钟(clock),timer（timer）,信号量（semaphore）,消息（message）,事件（event）,信号（signal）,分区（partition）,区域（region）,双端口内存（dual ported memory）,I/O,致命错误（fatal error），速率单调（rate monotonic）,用户扩展（user extensions）,多处理器（multiprocessing)。其中，**初始化，任务**是必须的，其他的可以根据用户的需要及实际情况进行适当剪裁。

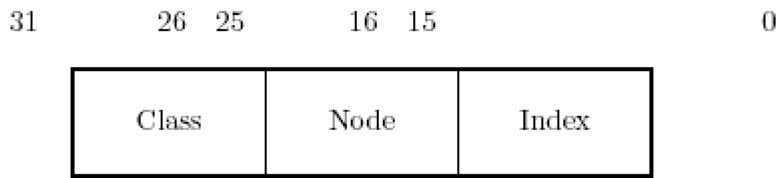
2 关键概念

2.1 介绍

2.2 对象

RTEMS 提供的命令可以允许用户动态的创建，删除，和产生一些预先定义的对象类型。这些类型包括：任务，信息队列，信号量，内存区域，内存分块，时钟，端口和速率单调周期等。面向对象的本意是 RTEMS 鼓励用户创建的模块是可以重用的。

每一个对象，都有一个名字和 ID，名字是用户任意取的，通常是能标是这个对象意义的符号。ID 是 RTEMS 给分配的，在执行中使用。对象名和 ID 都是 32 位无符号数。有三部分组成：



其中，高 6 位为 Class 项，表示该对象的类型，例如是一个任务或者是消息。中间 10 位为 Node 项，指的是该对象所在节点的号，即所在处理器的节点。低 16 位为 Index 项，代表这个对象在他所在的对象类中的索引。这三部分组合起来，使得在一个复杂的多处理器的系统中也可以很快的找到指定的对象。

可以通过特殊的服务，得到每一小部分的值

```
rtems_unsigned32 rtems_get_class( rtems_id );
rtems_unsigned32 rtems_get_node( rtems_id );
rtems_unsigned32 rtems_get_index( rtems_id );
```

而且可以通过 object identification derectives 对给定的一个名字，动态的得到他的 ID. RTEMS 通过这种方式，即使是最复杂的应用中也可以轻易的定位一个对象。

2.3 通信和同步

对一个实时的应用应该具有以下能力：

Data transfer between cooperating tasks 合作任务之间的数据传送
Data transfer between tasks and ISRs 任务和中断服务之间的数据传送
Synchronization of cooperating tasks 合作任务之间的同步
Synchronization of tasks and ISRs 任务和中断之间的同步

有几种 RTEMS 的管理机制可以提供通信和同步。但是不同的机制，可以提供的通信和同步的灵活性级别不同，可以根据用户和应用的需要，选择适当的机制。这几种机制包括：
Semaphore 信号量支持对共享资源的互斥和同步存取。二元信号量可以使用优先级继承法来避免优先级的倒转的问题。
Message Queue 信息队列管理既支持通信又支持同步。
Event 事件机制主要的提供很好的同步机制。
Signal 信号管理支持异步通信，通常在意外处理中使用。

2.4 时间

时间对于开发者来说是一个非常重要的概念。在 RTEMS 中基本的时间单位是一个 tick。RTEMS 可以支持任务延迟，超时，时间片，时间服务的延迟处理，任务的单一速率调度。延迟间隔是由一些 tick 构成的，从当前时间算起，例如，一个任务被要求在 10 个 tick 间隔后执行，那么除非 10 个 tick 结束否则它将不会被执行。所有的间隔都用 rtems_interval 类型来定义。
单一速率调度算法是一个硬实时调度算法。这个算法可以确保即使有暂时过载的情况下也可以确保一系列独立的任务可以在它们的 deadline 之前完成。

仅仅有间隔时间是不足够的，所以 RTEMS 还可以提供真正的时间日期给用户，以满足类似一个任务必须在北京时间午夜前完成这样的需求。

2.5 内存管理

RTEMS 的内存管理可以分成两个部分：动态内存分配和地址转换。动态内存分配应用于那些在执行过程中内存需求有变化的应用中。地址转换是用在那些需要和别的 CPU 共享内存或者有一个智能的 I/O 处理器。RTEMS 提供的管理内存：

_ Region
_ Partition
_ Dual Ported Memory

Partition 是用来管理好维护固定大小的数据池，例如资源控制块。Region 可以支持不同大小的内存块，并且被应用动态的获取和释放。Dual-ported 内存管理管理内部和外部的 dual-ported RAM 地址空间的地址转换。

3 RTEMS 的数据类型

_ rtems_address 数据类型，是用来管理地址的数据类型，它等于"void *" 指针。
_ rtems_asr 返回一个 RTEMS ASR 的类型。
_ rtems_asr_entry 是 RTEMS ASR 的入口地址。
_ rtems_attribute 数据类型，用来管理 RTEMS 的对象属性。它主要用来声明一个对象以创建一个规则来特别指定新对象的特性。
_ rtems_boolean 有 TRUE 和 FALSE 两个值。
_ rtems_context 数据结构，是 CPU 依赖的数据结构用来管理每一个任务上下文（环境）的整数和系统寄存器
_ rtems_context_fp 数据结构，是 CPU 依赖的数据结构用来管理每一个任务上下文（环境）的浮点数和系统寄存器
_ rtems_device_driver 返回一个 RTEMS 设备驱动规则的类型。
_ rtems_device_driver_entry RTEMS 设备驱动规则的入口指向。
_rtems_device_major_number 数据类型，RTEMS 管理的主要设备数。
_ rtems_device_minor_number 数据类型，RTEMS 管理的最小设备数。
_ rtems_double double 数据类型，是硬件的 float 的双倍精确。
_ rtems_event_set 数据类型，通过事件管理器来管理和维护 RTEMS 事件的一个数据类型。
_ rtems_extension 返回 RTEMS 用户扩展规则的类型。
_ rtems_fatal_extension 用户自扩展处理规则的致命错误入口指向。
_ rtems_id 数据类型，用来控制和维护 RTEMS 对象 ID 的数据类型
_ rtems_interrupt_frame 数据结构，定义了出现在用户 ISR 的中断堆栈结构的格式。这个数据结构不能够在所有的端口被定义。
_ rtems_interrupt_level 数据结构，和 rtems_interrupt_disable, rtems_interrupt_enable, 以及 rtems_interrupt_flash 规则一起使用。这个数据类型是 CPU 依赖的，通常和包含中断掩码的进程内容通信。
_ rtems_interval 数据类型，用来管理和维护时间间隔。时间间隔是非负整数，以 tick 来测量时间长度。
_ rtems_isr 返回执行一个 RTEMS ISR 的函数类型。
_ rtems_isr_entry RTEMS ISR 的入口指向地址。它等于指向执行 ISR 的函数入口地址。
_ rtems_mp_packet_classes 枚举类型，特别指出多处理的消息范畴。例如，一个消息的类

是必须被任务管理者执行的。
_ rtems_mode 数据类型，用来管理和动态的维护一个 RTEMS 任务的执行模式。
_ rtems_mpci_entry 返回 RTEMS MPCI 规则的类型。
_ rtems_mpci_get_packet_entry 指向为了 MPCI 执行而获取的包规则的入口地址。
_ rtems_mpci_initialization_entry 初始化一个 MPCI 执行规则的入口地址。
_ rtems_mpci_receive_packet_entry 接收一个 MPCI 执行的包规则的入口地址。
_ rtems_mpci_return_packet_entry 返回一个 MPCI 执行的包规则的入口地址。
_ rtems_mpci_send_packet_entry is the address of the entry point to the send packet routine for an MPCI implementation.
_ rtems_mpci_table is the data structure containing the con_guration information for an MPCI.
_ rtems_option is the data type used to specify which behavioral options the caller desires. It is commonly used with potentially blocking directives to specify whether the caller is willing to block or return immediately with an error indicating that the
resource was not available.
_ rtems_packet_prefix is the data structure that de_nes the _rst bytes in every packet sent between nodes in an RTEMS multiprocessor system. It contains routing information that is expected to be used by the MPCI layer.
_ rtems_signal_set is the data type used to manage and manipulate RTEMS signal sets with the Signal Manager.
_ rtems_signed8 is the data type that corresponds to signed eight bit integers. This data type is de_ned by RTEMS in a manner that ensures it is portable across di_erent target processors.

_ rtems_signed16 is the data type that corresponds to signed sixteen bit integers. This data type is de_ned by RTEMS in a manner that ensures it is portable across di_erent target processors.
_ rtems_signed32 is the data type that corresponds to signed thirty-two bit integers. This data type is de_ned by RTEMS in a manner that ensures it is portable across di_erent target processors.
_ rtems_signed64 is the data type that corresponds to signed sixty-four bit integers. This data type is de_ned by RTEMS in a manner that ensures it is portable across di_erent target processors.
_ rtems_single is the RTEMS data type that corresponds to single precision oating point on the target hardware.
_ rtems_status_codes is the
_ rtems_task is the return type for an RTEMS Task.
_ rtems_task_argument is the data type for the argument passed to each RTEMS task.
_ rtems_task_begin_extension is the entry point for a task beginning execution user extension handler routine.
_ rtems_task_create_extension is the entry point for a task creation execution user extension handler routine.

_ rtems_task_delete_extension is the entry point for a task deletion user extension handler routine.
_ rtems_task_entry is the address of the entry point to an RTEMS ASR. It is equivalent to the entry point of the function implementing the ASR.
_ rtems_task_exitted_extension is the entry point for a task exitted user extension handler routine.
_ rtems_task_priority is the data type used to manage and manipulate task priorities.
_ rtems_task_restart_extension is the entry point for a task restart user extension handler routine.
_ rtems_task_start_extension is the entry point for a task start user extension handler routine.
_ rtems_task_switch_extension is the entry point for a task context switch user extension handler routine.
_ rtems_tcb is the data structure associated with each task in an RTEMS system.
_ rtems_time_of_day is the data structure used to manage and manipulate calendar time in RTEMS.
_ rtems_timer_service_routine is the return type for an RTEMS Timer Service Routine.
_ rtems_timer_service_routine_entry is the address of the entry point to an RTEMS TSR. It is equivalent to the entry point of the function implementing the TSR.

_ rtems_unsigned8 is the data type that corresponds to unsigned eight bit integers. This data type is de_ned by RTEMS in a manner that ensures it is portable across di_erent target processors.
_ rtems_unsigned16 is the data type that corresponds to unsigned sixteen bit integers.
This data type is de_ned by RTEMS in a manner that ensures it is portable across di_erent target processors.
_ rtems_unsigned32 is the data type that corresponds to unsigned thirty-two bit integers. This data type is de_ned by RTEMS in a manner that ensures it is portable across di_erent target processors.
_ rtems_unsigned64 is the data type that corresponds to unsigned sixty-four bit integers. This data type is de_ned by RTEMS in a manner that ensures it is portable across di_erent target processors.
_ rtems_vector_number is the data type used to manage and manipulate interrupt vector numbers.

4.初始化管理

4.1 介绍

RTEMS 初始化和关闭。初始化 RTEMS 包括创建，开始所有配置好的初始化任务，调用所有用户支持的设备驱动器初始化程序。在多处理器中，还要初始化处理器之间的通讯层。RTEMS 提供的命令有 4 个：

_ rtems_initialize_executive 初始化 RTEMS
_ rtems_initialize_executive_early 初始化 RTEMS 但是不开始多任务
_ rtems_initialize_executive_late – 完成初始化开始多任务
_ rtems_shutdown_executive 关闭 RTEMS

4.2 背景

4.2.1 初始化任务

RTEMS 通过初始化任务把初始化控制转给用户的应用。初始化任务不同于其他的应用，在于它们是由用户初始化任务表定义,然后作为初始化序列中的一部分由 RTEMS 自动的创建和开始。由于初始化任务和其他的 RTEMS 任务采用相同的调度算法，所以它们的优先级和模式必须配置成最高，使得他们可以在其他的应用开始之前被执行结束。

一个典型的初始化任务将会创建和开始静态的应用任务。如果仅仅用来初始化的任务将会在结束时自己删除自己,以释放资源给其他任务。其他初始化任务会将自己转换为普通的任务，这种转换将调用改变优先级和执行模式。RTEMS 并不自动删除初始化任务。

4.2.2 系统初始化

系统初始化任务是初始化所有的设备驱动。所以，这个任务有最高的任务优先级来确保没有任务会比他先执行。在单处理器系统中，这个任务初始化结束后将会删除自己。系统初始化任务必须有足够的空间来完成初始化,而且对于多处理器系统必须还要完成任务间通讯接口层。CPU 配置表允许用户修改堆栈数量以获得更多的空间来执行任务。

在多处理器中，在硬件驱动结束后系统初始化任务并不删除自己。它将转换为多处理器服务来初始化多处理器间的通讯接口层，确保多处理器系统的完整性，并且处理所有的远程节点的需求。

4.2.3 空闲任务

空闲任务是最低优先级的任务，仅当没有其他任务执行的时候它才会被执行。它是一个空循环，当有任何任务准备执行的时候，空闲任务将被占先。

4.2.4 初始化任务失败

执行 rtems_initialize_executive 若出现了下面问题将调用 rtems_fatal_error_occurred

如果配置表和 CPU 依赖信息没有提供

在配置表中的应用给出的 RTEMS RAM 工作区地址为空或者没有连结到一个四 byte 的边界。

如果 RTEMS RAM 工作区不够大来初始化和配置系统。
如果中断堆栈太小。
如果多处理器已经配置了,但是多处理器配置表中的节点入口并不在 1 和最大的节点入口范围内。
如果多理器已经配置了，但是多理器间是通讯接口却没有指定。
如果没有用户初始化任务被配置。至少需要有一个初始化任务可以在初始化序列执行结束的时候，让 RTEMS 把控制权交给应用。
如果任何用户初始化任务都不能被成功的创建和开始。

4.3 操作
4.3.1 初始化 RTEMS
在 BSP 初始化结束后，将调用 rtems_initialize_executive，它将执行下列动作：
_ Initializing internal RTEMS variables;初始化变量
_ Allocating system resources;分配系统资源
_ Creating and starting the System Initialization Task;创建和开始系统初始化任务
_ Creating and starting the Idle Task;创建和开始等待任务
_ Creating and starting the user initialization task(s); and.创建和开始用户的任务
_ Initiating multitasking 初始化多任务
注意，必须在调用其他的应用前，调用 rtems_initialize_executive。在初始化中 RTEMS 的许多动作依赖于 Configuration Table and CPU Dependent Information Table。
初始化的最后一个工作是开始多任务。然后优先级最高的和等待中的任务将开始执行。一旦初始化结束后控制权将不会回到 BSP，直到调用 rtems_shutdown_executive。rtems_initialize_executive_early 将在初始化多任务之前返回调用者，而 rtems_initialize_executive_late 来调用初始化多任务。
4.3.2 结束 RTEMS
rtems_shutdown_executive 由一个应用调用，然后将控制权返回给 BSP，当调用了 rtems_initialize_executive 后，BSP 将重新开始运行。

5 任务管理
任务管理提供一系列的命令来创建，删除和管理任务。
_ rtems_task_create –创建一个任务
_ rtems_task_ident – 获得一个任务的 ID
_ rtems_task_start – 开始一个任务
_ rtems_task_restart – 重新开始一个任务
_ rtems_task_delete – 删除一个任务
_ rtems_task_suspend – 挂起一个任务
_ rtems_task_resume – 重新开始一个任务
_ rtems_task_is_suspended – 决定一个任务是否被挂起
_ rtems_task_set_priority – 设置任务优先级
_ rtems_task_mode – 改变当前任务模式
_ rtems_task_get_note – 获得一个任务记事本的入口
_ rtems_task_set_note – 设置任务记事本的入口

_ rtems_task_wake_after – 间隔后唤起任务
_ rtems_task_wake_when –特别指定时唤起
_ rtems_task_variable_add – 关联（增加）每个任务的变量
_ rtems_task_variable_get – 获取每个任务变量的值
_ rtems_task_variable_delete – 删除每个任务变量

5.2 背景
5.2.1 任务定义
一个可分配的单元
进程分配的实体
一个实时多处理器系统的原子
单一的线成，并发的竞争资源
一系列紧密相关的计算，并且可以和其他的计算序列并发的执行
从 RTEMS 的角度来看，任务是最小的不可分的执行线成，以来竞争资源。一个任务可以通过现存的任务控制模块 TCB 来表明。
5.2.2 任务控制模块
TCB 任务控制模块是 RTEMS 自定义的一个数据结构，它包含了和执行一个任务有关的所有信息。在系统初始化的时候，RTEMS 为每一个任务预先留出配置所需要的空间。当创建一个新任务时,TCB 被分配。删除一个任务时，就释放 TCB,返回到 TCB 空闲表中。
TCB 是 RTEMS 的内部数据结构，应用可以通过用户的扩展规则来访问它。TCB 中记录了任务的名字，ID,当前优先级，当前和开始状态，执行模式，记事本地址集，TCB 用户扩展指针，调度控制结构，和阻塞一个任务需要的数据。当任务转换，或是重新开始等，都要用到 TCB.
一个任务的上下文（转换环境）都储存在 TCB 中，当需要转换的时候就要用到 TCB 了。
5.2.3 任务状态
任务有 5 种状态:
执行　　正在 CPU 中调度
就绪　　将要在 CPU 中执行
阻塞　　不能够在 CPU 中执行
睡眠　　创建的任务还不能开始
不存在 不能够创建和删除任务
一个活动的任务可以是开始是四种，否则可以认为是最后一种。一个或更多的任务可以同时被激活。多任务的通讯，同步和竞争系统的资源由系统调用来实现。看起来是并行的，实际上一个时间只能有一个任务占用 CPU，通过 RTEMS 调度算法。调度算法依赖于任务状态和优先级。

5.2.4 任务优先级
一个任务的优先级决定了它的重要性。每个任务都有个优先级.RTEMS 支持 255 个优先级，从 1 到 255。RTEMS 使用 rtems_task_priority 来存储任务的优先级。
优先级标记越小，优先级越高。同一个优先级可以分给多个任务。一个任务的优先级是可以改变的。优先级是用来帮助调度算法来调度任务的。通常，优先级高的任务，更有可能得到处理器的执行时间。

5.2.5 任务模式

一个任务的执行模式和下面 4 个因素相关

_ preemption (优先权)

_ ASR processing (asynchronous signal processing) (异步信号处理)

_ timeslicing (时间片)

_ interrupt level (中断级别)

执行模式可以用来改变 RTEMS 的调度处理,改变任务的执行环境。数据类型 rtems_task_mode 用来管理任务的执行模式。

优先级部分允许一个任务来决定什么时候一个处理应该放弃。如果优先级被禁止了即 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，处理器收回当前任务的使用权，并且交给高优先级的任务。

时间片 timeslicing 元素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 元素就被忽略。

异步信号处理元素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), 那么信号将会在这个任务下一次执行时被处理。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。这个元素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个中断可以使能,RTEMS_INTERRUPT_LEVEL(n) 定义了任务执行使时的中断级别 n。

_ RTEMS_PREEMPT - enable preemption (default)

_ RTEMS_NO_PREEMPT - disable preemption

_ RTEMS_NO_TIMESLICE - disable timeslicing (default)

_ RTEMS_TIMESLICE - enable timeslicing

_ RTEMS_ASR - enable ASR processing (default)

_ RTEMS_NO_ASR - disable ASR processing

_ RTEMS_INTERRUPT_LEVEL(0) - enable all interrupts (default)

_ RTEMS_INTERRUPT_LEVEL(n) - execute at interrupt level n

5.2.6 访问任务声明

所有的 RTEMS 都有一个声明，当任务开始或者重新开始的时候声明将被调用。这个声明通常用来通讯任务间的开始信息，最简单的一个这样的声明是这样的：

```
rtems_task user_task(
rtems_task_argument argument
);
```

5.2.7 浮点考虑

创建一个有 RTEMS_FLOATING_POINT 属性的任务将要求在任务执行的时候，TCB 分配更多的

内存空间给数字协处理器。而这多余的内存对于 RTEMS_NO_FLOATING_POINT 任务缺是完全不需要的。存储和恢复 RTEMS_FLOATING_POINT 的任务也比 RTEMS_NO_FLOATING_POINT 要花更多的时间，因为需要更多的时间用来存储和恢复数字协处理器的运算状态。
因为 RTEMS 的设计是为了嵌入式是应用,所以我们设计的时候避免了不必要的开支在存储和恢复数字协处理器的运算状态上。仅当一个 RTEMS_FLOATING_POINT 任务被分配并且这个任务不是最后一个使用数字协处理器的时候，才存储数字协处理器的状态。如果一个系统里只有一个 RTEMS_FLOATING_POINT 任务，数字协处理器的状态将永远不会被存储和恢复。
虽然一个 RTEMS_FLOATING_POINT 任务的总开销已经最小了，但是一些任务还是要避免使用数字协处理器。通过阻止一个浮点任务在执行时被占先，一个 RTEMS_NO_FLOATING_POINT 可以使用数字协处理器而不必担心 RTEMS_FLOATING_POINT 那样的开销。这种方法同样解决了分配浮点指针区域的问题。如果用户决定采用这种方法，将不能创建任何 RTEMS_FLOATING_POINT 任务。
如果支持的处理器不支持浮点，或者没有一个标准的数字协处理器。那么 RTEMS 将不提供硬件解决浮点运算的方法。这种情况下，所有的任务都被认为是 RTEMS_NO_FLOATING_POINT，而不管用户创建的是什么。一个浮点运算的仿真软件库将被用来做浮点运算。
在一些处理器中，可以动态的禁止浮点运算。如果目标处理器支持这个功能 RTEMS 将可以动态的支持那些被创建为 RTEMS_FLOATING_POINT 的任务。
关于 5307 的浮点计算部分
5307 支持双精度浮点计算，不管运算数是什么，5307 都先将它转换为双精度浮点数（64 位）
5307 有 8 个 64 位浮点数字寄存器（FP0-FP7）
浮点控制积存器（FPCR），意外使能（EE）模式控制（MC）

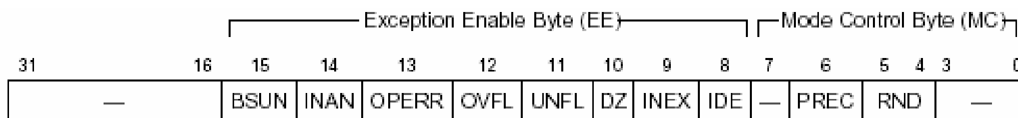


Figure 1-4. Floating-Point Control Register (FPCR)

浮点状态积存器（FPSR），浮点条件代码（FPCC），浮点意外状态（EXC），浮点增长异常（AEXC）

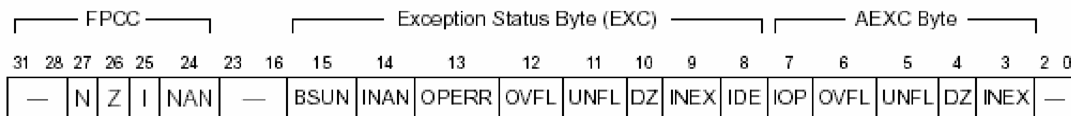


Figure 1-5. Floating-Point Status Register (FPSR)

浮点指令地址积存器（FPIAR）
COLDFIRE 操作数执行管道可以同时执行整数和浮点数。所以，当 PC 指令寄存器响应一个浮点异常陷阱时，将不会指向指令而导致一个意外。
这样 FPU 就可能产生异常陷阱，在 FPU 开始执行之前，FPIAR 就装载了指令寄存器 PC 的地址。在 FPU 出现异常时，陷阱处理器可以使用 FPIAR 中的地址来确定指令并且产生异常。

5.2.8 每个任务的变量（Per task variables）
Per task variables 是用来支持那些全局变量的，这些全局变量对于每一个任务是唯一的。当把一个变量标识为 private 后，任务将可以访问和修改这个变量，这种修改将仅对当前这个任务有效，别的任务的修改也不会影响到这个任务。这些在每一次保存和恢复任务时执行。如果一个任务把一个变量私有了，那么别的任务对这个变量造成的变化也不会影响到该变量

的值。
虽然每一个任务都有自己的变量堆栈，但是他们还是会共享一些静态和全局变量。为了使一个变量变成不可共享的，任务可以用 rtems_task_variable_add 为每一个任务做一个不同的变量拷贝，但是有相同的物理地址。
任务的变量将会增加拥有他们的任务的执行和转化的开销，所以我们要尽可能的最小化任务变量。一个有效的方法是有一个单一的任务变量指针，它指向一个包含任务私有全局数据的动态分配结构。

5.2.9 建立任务属性设置

通常，一个属性设置是通过和所需的成分进行位运算 OR 建立的。有效的属性成分是：
_ RTEMS_NO_FLOATING_POINT - does not use coprocessor (default)不使用协处理器
_ RTEMS_FLOATING_POINT - uses numeric coprocessor 使用数字协处理器
_ RTEMS_LOCAL - local task (default)本地任务
_ RTEMS_GLOBAL - global task 全局任务
属性值被特别设计成互斥的，所以位运算 OR 以及其他的操作是等同的，只要每一个属性仅仅在任务属性成分表中出现一次。缺省的成分表是不需要出现的，虽然一个良好的编程风格最好一一指出。如果想更改缺省配置，需要使用 RTEMS_DEFAULT_ATTRIBUTES。
下面的例子指出了一个本地任务，而且使用数字协处理器的任务属性列表：
attribute_set参数应该是：
RTEMS_FLOATING_POINT or RTEMS_LOCAL | RTEMS_FLOATING_POINT.
因为本地任务是缺省的，所以可以省略不写，但是如果是一个全局任务就要写成：
attribute_set参数应该是RTEMS_GLOBAL | RTEMS_FLOATING_POINT.

5.2.10 建造一个模式或者掩码

通常，一个模式和它相应的掩码是通过和所需的成分进行位运算 OR 建立的。有效的模式常量和相应的每一个模式掩码常量如下：
_ RTEMS_PREEMPT is masked by RTEMS_PREEMPT_MASK and enables preemption
_ RTEMS_NO_PREEMPT is masked by RTEMS_PREEMPT_MASK and disables preemption
_ RTEMS_NO_TIMESLICE is masked by RTEMS_TIMESLICE_MASK and disables timeslicing
_ RTEMS_TIMESLICE is masked by RTEMS_TIMESLICE_MASK and enables timeslicing
_ RTEMS_ASR is masked by RTEMS_ASR_MASK and enables ASR processing
_ RTEMS_NO_ASR is masked by RTEMS_ASR_MASK and disables ASR processing
_ RTEMS_INTERRUPT_LEVEL(0) is masked by RTEMS_INTERRUPT_MASK and enables all interrupts
_ RTEMS_INTERRUPT_LEVEL(n) is masked by RTEMS_INTERRUPT_MASK and sets interrupts level n
属性值被特别设计成互斥的，所以位运算 OR 以及其他的操作是等同的，只要每一个属性仅仅在任务属性成分表中出现一次。缺省的成分表是不需要出现的，虽然一个良好的编程风格最好一一指出。如果想更改缺省配置，需要使用 RTEMS_DEFAULT_MODES 和 RTEMS_ALL_MODE_MASKS。
下面的例子给出了一个任务模式中断级别3和没有优先级：
RTEMS_INTERRUPT_LEVEL(3) | RTEMS_NO_PREEMPT。

5.3操作

5.3.1创建任务

通过分配任务控制模块（TCB）和分配给任务一个用户定义的名字，分配堆栈和浮点计算的

转换区，设置用户定义的初始化优先级，设置用户特别定义的初始化模式，分配一个任务ID，rtems_task_create就这样创建了一个任务。新创建的任务处于休眠状态。

5.3.2获得任务IDs

当一个任务被创建的时候，RTEMS将会分配一个独一无二的ID给它，直到它被删除。任务ID可以通过两种方式获得：

1， 作为调用rtems_task_create的结果，任务的ID将被储存在用户提供的区域里。

2， 我们可以通过调用 rtems_task_ident来获得任务ID。

其它的命令可以通过使用这个任务的ID来维护这个任务。

5.3.3开始和重新开始一个任务

rtems_task_start将一个任务从休眠状态放到就绪状态。这样就允许任务以它自己的优先级来竞争处理器和其它的资源。任何动作，例如挂起和改变优先级，必须在任务开始之前进行，一旦开始了将不能执行任何操作。

通过rtems_task_start指明了用户特别定义的任务开始地址和声明。声明用来传送一些任务的开始信息。RTEMS根据任务的初始化执行模式和开始地址初始化任务的堆栈。新的任务声明用来区分原来的任务调用和后来的任务调用。任务的堆栈和控制区域的改变可以反映它们最初的值。虽然需求请求的相关资源已经清除了，但是一个任务的资源并不会自动返回系统。一个任务不能重新启动，除非先前它已经启动过了。(例如，休眠任务不能自动重新启动)所有重新启动的任务都应该在就绪队列里。

5.3.4挂起和重新开始一个任务

rtems_task_suspend 和 rtems_task_resume是一对命令。前者可以使任务挂起，通过后者，已经挂起的状态可以解除挂起状态。

由于在等待资源或者是时间片消耗完，那么任务就会进入挂起状态，直到用户运行rtems_task_resume命令。如果任务没有被阻塞，rtems_task_resume命令将使得任务从挂起状态变为就绪状态，重新开始和其它任务一起竞争资源和处理器。如果阻塞了，那么将仅仅清除它的挂起状态。

挂起一个已经挂起的任务或者重新一个开始不是挂起状态的任务都会导致错误发生。

rtems_task_is_suspended可以用来判断一个任务是否处在挂起状态。

5.3.5延迟一个当前执行的任务

通过rtems_task_wake_after 命令，使得一个任务睡眠一段时间，这段时间过后，任务重新唤醒。一个任务调用了rtems_task_wake_after并且指定了一个RTEMS_YIELD_PROCESSOR ticks的延迟将使得处理器放弃对它的执行，并且将这个任务重新放回就绪队列，然后处理器将执行就绪队列中的任务。

rtems_task_wake_when 命令，使得一个任务睡眠到一个指定的时间。到指定的时间是，就可以唤醒了。

5.3.6改变任务优先级

rtems_task_set_priority命令可以改变任务的优先级。如果改变任务的优先级后，和原来的优先级一样，那么就不改变原来的。如果不一样则改变。

但是当执行rtems_task_restart命令，就把任务的优先级复位到初始值。

5.3.7改变任务模式

rtems_task_mode 可以用来获得和改变当前任务的执行模式。一个任务的执行模式包括优先级使能，时间片。ASR处理和设置任务的中断级。

同样，rtems_task_restart可以复位任务的初始模式。

5.3.8笔记本区域

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。

Rtems_task_set_note可以使用户设置任务的记事本入口为指定的记事本入口。

Rtems_task_get_note可以使得使用者获得任务的16个记事本中的任意指定的一个。

5.3.9删除任务

通过rtems_task_delete可以把任务删除，同时这个任务相关的资源也被释放。但是，一些动态分配给这个任务的资源不能自动返回给RTEMS,所以，在任务删除前，使用之前先把那些动态分配的释放掉。通常是先把任务restart, 然后再delete.

6 中断管理

6.1 介绍

任何一个实时的操作都必须为响应外部产生的中断而提供一种良好的机制,以来满足那些应用能在deadline之前完成。中断管理就为RTEMS提供了这种机制。中断管理通过改变任务的执行，而提供了一种快速响应机制。任何任务，碰到中断，都必须暂停正在执行的任务，响应中断。处理完中断后再接着执行任务。

中断管理器提供了下面的命令：

_ rtems_interrupt_catch - Establish an ISR

_ rtems_interrupt_disable - Disable Interrupts

_ rtems_interrupt_enable - Enable Interrupts

_ rtems_interrupt_flash - Flash Interrupt

_ rtems_interrupt_is_in_progress - Is an ISR in Progress

6.2 背景

6.2.1 处理一个中断

硬件管理器允许应用和硬件中断向量表相连。当出现一个中断的时候，将把中断向量自动返回给 RTEMS。RTEMS 会为了目标处理器保存和恢复不被普通 C 语言保留的寄存器，RTEMS 也同时调用用户的 ISR。用户的 ISR 是为了响应处理中断，清除中断（如果需要）和设备的特别维护。

rtems_interrupt_catch 命令连接到一个中断向量表的处理。向量采用了rtems_vector_number 类型。

中断服务子程序被认为是遵守这些规则的，而且有一个类似于下列的原形：

```
rtems_isr user_isr(
rtems_vector_number vector
);
```

中断向量是给 RTEMS 用来判断中断源是什么的。然后有一个程序将会用来服务这个中断。例如，一个中断是响应多个端口的串口设备的，那么中断向量就可以鉴别到底应该去服务哪一个端口的串口设备。

用户的 ISR 执行结束后，将把控制权返回给 RTEMS，RTEMS 将可以会恢复在调用 ISR 之前的寄存器状态。中断管理器必须允许更高级别的中断可以中断当前的中断 ISR。因此，中断是可以嵌套的。

6.2.2 RTEMS 的中断级

RTEMS 的中断级别为 256 级,从 0 到 255。级别越低,优先级越高。

6.2.3 禁止中断

当执行关键区的代码时,要屏蔽所有的可屏蔽中断。RTEMS会优化，使得屏蔽所有的可屏蔽中断仅仅持续一个最短的时间。如果，在执行关键区的代码时，产生了不可屏蔽中断（NMI），

因为系统不能保护关键区，就会产生不可预知的结果。所以，在执行这些代码时，最好不要进行系统调用以产生了不可屏蔽中断。

6.3 操作

6.3.1 建立一个 ISR

通过rtems_interrupt_catch可以为系统建立一个ISR。并把用户的ISR的地址写到RTEMS 的向量表中。

6.3.2 ISR 可以调用的系统指令

使用中断管理器来确保 RTEMS 知道什么时候一个命令被 ISR 调用了。ISR 使用系统调用来使它自己和一个任务同步。这个同步包括 ISR 传递给目标任务的消息，事件，信号等。ISR 调用的命令必须是对一个本地对象进行操作。ISR 可以调用的 RTEMS 系统指令包括：

- 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，信号管理
  - message_queue_send, message_queue_urgent
  - event_send
  - signal_send
- 信号量管理
  - semaphore_release
- 双端口管理
  - port_external_to_interal, port_internal_to_external
- I/O 管理
  - io_initialize, io_open, io_close, io_read, io_write, io_control
- 致命错误管理
  - fatal_error_occurred
- 多处理器
  - multiprocessing_announce

7 时钟管理

7.1 介绍

时钟管理提供了跟时间相关的一些系统能力的支持。包括 3 条命令：

1) rtems_clock_set – 设置系统的日期和时间
2) rtems_clock_get – 获取当前系统日期和时间信息
3) rtems_clock_tick – 声明一个系统时钟滴答(tick)

7.2 背景

7.2.1 需要的支持

为了使用时钟管理器，RTEMS 要求硬件必须提供周期性的时钟中断。rtems_clock_tick 命令通常被 timer 中断服务程序调用告诉 RTEMS 系统一个系统时钟中断 tick 出现了。时间由 tick 来计量，一个 tick 必须是微秒的整数倍。Tick 的大小由用户在配置表中声明。

7.2.2 时间和日期的结构

时钟管理器的命令使用下面的数据结构来对本地的时间和日期进行操作:

```
struct rtems_tod_control {
    rtems_unsigned32 year; /* greater than 1987 */
    rtems_unsigned32 month; /* 1 - 12 */
    rtems_unsigned32 day; /* 1 - 31 */
    rtems_unsigned32 hour; /* 0 - 23 */
    rtems_unsigned32 minute; /* 0 - 59 */
    rtems_unsigned32 second; /* 0 - 59 */
    rtems_unsigned32 ticks; /* elapsed between seconds */
};
typedef struct rtems_tod_control rtems_time_of_day;
```

使用 rtems_clock_get 命令时，这个数据结构是我们唯一支持的。一些应用希望能够提供类似于 UNIX 的时间日期，那么 rtems_clock_get 也可以相应的返回以下面的结构为格式的当前时间：

```
typedef struct {
    rtems_unsigned32 seconds; /* seconds since RTEMS epoch*/
    rtems_unsigned32 microseconds; /* since last second */
} rtems_clock_time_value;
```

7.2.3 时间 tick 和时间片

时间片是一个很重要的概念，当系统内的任务是优先级相同的时候，系统将自动采用轮循的方式执行任务，这个时候时间片就决定了每一个任务执行时间的长短。

时间片必须是 tick 的整数倍，由用户在配置表中声明。虽然整个系统必须有一个统一的时间片，但是每一个任务还是可以禁止或者允许时间片。

当系统既允许时间片也允许优先级的时候，rtems_clock_tick 命令通过消耗正在执行的任务的时间剩余计数器来执行时间片。当时间片耗尽，这个任务将被别的优先级相同的任务占先。

7.2.4 延迟

一个睡眠timer允许一个任务延迟一个给定的时间或者延迟到给定的时间，然后再唤醒继续执行。这种类型的timer由rtems_task_wake_after和rtems_task_wake_when命令自动创建，所以，它们没有RTEMS的ID。一旦timer处于活动状态，将不能删除。一个时间上，一个任务有也仅有一个延迟timer。

7.2.5 超时

当rtems_message_queue_receive, rtems_event_receive, rtems_semaphore_obtain或者rtems_region_get_segment命令出现超时的时候，一个特别的timer类型—超时将自动的创建。一个时间上，一个任务有也仅有一个超时timer。一旦超时结束，它将解除超时任务的超时状态代码。

7.3操作

7.3.1 声明一个 tick

RTEMS 提供了 rtems_clock_tick 命令（通过调用用户的实时时钟）ISR 来告诉 RTEMS 一个 tick 已经过去了。一个 tick 必须是微秒的整数倍。Tick 的大小由用户在配置表中声明。系统调用 rtems_clock_tick 命令的频率决定了所有依赖于时间的操作粒度。例如，一秒中调用 rtems_clock_tick 命令 10 次要比一秒中调用 rtems_clock_tick 命令 2 次准确的多。rtems_clock_tick 命令既用来维护系统的日历时间，也用来维护 timer 的动态设置。

7.3.2 设置时间

通过 rtems_clock_set 命令，RTEMS 允许一个任务或者是 ISR 来设置时间。如果日期和时间的设置导致了任何未触发的 timer 超过了他们的 deadline，那么这些 timer 将在执行 rtems_clock_set 命令的时候被触发。

7.3.3 获得一个时间

通过 rtems_clock_get 命令，一个任务或者是 ISR 可以获取当前系统的时间或者相关的时间信息。当前的时间将可以或者以本地的格式或者以类似 UNIX 的格式返回。至于你想获得什么，依赖于你使用的命令：

_ RTEMS_CLOCK_GET_TOD - obtain native style date and time获得本地风格的时间
_ RTEMS_CLOCK_GET_TIME_VALUE - obtain UNIX-style date and time获得UNIX风格的时间
_ RTEMS_CLOCK_GET_TICKS_SINCE_BOOT - obtain number of ticks since RTEMS was initialized获得自RTEMS初始化后的tick数
_ RTEMS_CLOCK_GET_SECONDS_SINCE_EPOCH - obtain number of seconds since RTEMS epoch获得自RTEMS初始化后的秒数
_ RTEMS_CLOCK_GET_TICKS_PER_SECOND - obtain number of clock ticks per second获得每秒的tick数。

但是要注意，必须先通过rtems_clock_set设置, 然后再用rtems_clock_get获取,否则，将会返回出错代码。

8定时器管理

8.1介绍

timer管理提供一些对timer的支持。命令有：

1) rtems_timer_create – 创建一个 timer
2) rtems_timer_ident – 获取一个 timer 的 ID
3) rtems_timer_cancel – 取消一个 timer
4) rtems_timer_delete – 删除一个 timer
5) rtems_timer_fire_after – 在一段时间后激发 timer
6) rtems_timer_fire_when – 在特定的时间激发 timer
7) rtems_timer_reset – timer 复位

8.2背景

8.2.1所需支持

timer管理需要系统定义tick。

8.2.2 Timer

timer是RTEMS的一个对象。用来调度一段特定的程序（timer service routine）在一段特定的时间后(这段时间由timer决定)执行。TSR或者被rtems_clock_tick命令调用，或者被特定的timer服务任务调用（当timer被触发时)。

Timer可以用来执行开门狗程序，用来指出一个应用出了错误。当一个应用执行到重起点的时候timer将被重新设置，这样看门狗就不会触发。如果应用未能运行到那个重起点，那么看门狗将会被触发，然后将执行TSR。

8.2.3 timer服务

timer服务任务是为了响应所有基于任务的TSR。这个任务比其它所有的应用任务优先级都高，所以可以认为是最低级别的中断。

可以把 timer 看成是一种最低优先级的中断。它和中断很象，也有一个 timer 服务子程序。它要比中断更灵活。

Timer 任务将处于阻塞状态，直到一个基于任务的 timer 被触发。这样可以减少\执行这个任务的开销。

8.2.4 timer 服务程序

该程序有如下原形：

```
rtems_timer_service_routine user_routine(
    rtems_id timer_id,
    void *user_data
);
```

timer_id是被触发的timer的ID。user_data是一个指针，指向要被TSR使用的用户定义信息，可以为NULL。

8.3操作

8.3.1创建一个timer

通过 rtems_timer_create 命令可以创建一个 timer。系统还会为这个 timer 分配一个 timer 控制块（TMCB）来管理这个新创建的 timer。新创建的 timer 没有相关联的 timer 服务程序，是不活跃的。

8.3.2 获取一个 timer 的 ID

当一个 timer 被创建了之后,RTEMS 就会给他分配一个唯一的 ID。有两种方法可以获得 timer 的 ID:1.通过 rtems_timer_ident 命令可以获得这个 timer 的 ID,以便在调用其他的 timer 命令中使用。2.作为调用 rtems_timer_create 的结果，ID 将被保存在用户定义的路径里。

8.3.3 初识化一个间隔 timer

rtems_timer_fire_after 和rtems_timer_server_fire_after命令可以初始化一个间隔timer，使得它在一段时间过后触发一个TSR。当间隔逝去后，TSR就被rtems_clock_tick命令触发调用如果是用rtems_timer_fire_after初始化的；或者被timer服务任务调用如果是用rtems_timer_server_fire_after初始化的。

8.3.4 初始化一个 day timer 的 timer

rtems_timer_fire_when 和rtems_timer_server_fire_when命令可以初始化一个间隔timer，使得它在特定的时间触发一个TSR。当间隔逝去后，TSR就被rtems_clock_tick命令触发调用如果是用rtems_timer_fire_when初始化的；或者被timer服务任务调用如果是用

rtems_timer_server_fire_when初始化的。

8.3.5 取消一个 timer

rtems_timer_cancel 指令可以停止一个特定的 timer。一旦 timer 被取消了，timer 服务子程序就不会被触发，直到 timer 被重新初始化。命令 rtems_timer_reset, rtems_timer_fire_after, rtems_timer_fire_when 都可以重新初始化 timer。

8.3.6 复位一个 timer

rtems_timer_reset 命令把一个被 rtems_timer_fire_after 或 rtems_timer_server_fire_after 初始化的计时器恢复到原来的时间间隔。如果 timer 没有被使用过，或者最后一次使用这个 timer 的是 rtems_timer_fire_when 和 rtems_timer_server_fire_when 命令，则报错。

8.3.7 初始化 timer 服务

rtems_timer_initiate_server命令用来分配或者开始一个timer服务任务的执行。应用可以指定堆栈的大小或者服务的属性。timer服务任务是为了响应所有基于任务的TSR。这个任务比其它所有的应用任务优先级都高，所以rtems_timer_initiate_server命令会抢先执行。

8.3.8 删除一个 timer

rtems_timer_delete 命令用来删除一个计时器。如果这个计时器正在运行，并且没有过期，这个计时器就会自动取消。TMCB 也会被回收，返回到 TMCB 空闲表中。除了创建这个 timer 的任务外，timer 也可以被其它任务删除。

9 信号量管理

9.1 介绍

信号量管理应用标准的 Dijkstra 计数信号量来进行同步和互斥操作。主要的命令有：

1) rtems_semaphore_create – 创建一个信号量
2) rtems_semaphore_ident – 获取信号量的 ID
3) rtems_semaphore_delete – 删除一个信号量
4) rtems_semaphore_obtain – 获得信号量
5) rtems_semaphore_release – 释放信号量
6) rtems_semaphore_flush – 解除所有等待该信号量的任务的阻塞

9.2 背景

RTEMS 支持二元信号量和多元信号量。二元信号量可以取 0，1 两个值，多元信号量可以是任何非负的整数。二元信号量是控制单个资源的互斥访问。多元信号量是控制资源集的访问。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 ，应该设置成 3 个值信号量。

同时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,于是就在同步点发出 rtems_semaphore_obtain 指令，A 将发现信号量为 0，则继续等待。任务 B 到达同步点后，发出了一个相应的 rtems_semaphore_release 指令，将信号量加 1，任务 A 收到了该指令,继续执行，达到了同步。

9.2.1 嵌套资源访问

RTEMS 解决死锁问题，RTEMS 通过允许任务以一种嵌套的方式，在持有信号量时可以继续获得同一个信号量。每个 rtems_semaphore_obtain 必须与 rtems_semaphore_release 配套。仅当最外层的 rtems_semaphore_obtain 与 rtems_semaphore_release 匹配的时候，其它任务对信号量的请求才有效。

简单信号量（0，1）不允许嵌套访问，因此可以用来同步。

9.2.2 优先级倒置

高优先级的任务要对共享资源读取时,资源被低优先级任务占用,直到低优先级的任务完成后,高优先级才能读取共享资源的现象。(比如: 低优先级占有了共享资源,但是他却由于中间优先级的任务占用CPU而不能执行,由于不能执行,也就不能释放资源,高优先级的任务得不到资源,也只能处于挂起状态。) 这种现象是要避免的。

9.2.3 优先级继承

它是一种避免优先级倒置的算法。把占用资源的低优先级的任务的优先级增加到**当前**等待这个资源的任务中的最高的优先级。当执行完成后,释放资源,并把优先级恢复成原来。RTEMS 支持这种算法。

9.2.4优先级置顶

把占用资源的低优先级任务提高到等待或将要等待这个资源的所有任务中的最高优先级。这样置顶就避免了继承那样需要在执行的时候多次改变任务的优先级。当执行完成后,释放资源,并把优先级恢复成原来。虽然这个算法可以避免多次改变优先级，但是由于在一个复杂的多任务系统中计算所有任务的最高级很难实现，所以通常采用**优先级继承**算法。

9.2.5 信号量属性设置

_ RTEMS_FIFO - tasks wait by FIFO (default)
_ RTEMS_PRIORITY - tasks wait by priority
_ RTEMS_BINARY_SEMAPHORE - restrict values to 0 and 1 (default)
_ RTEMS_COUNTING_SEMAPHORE - no restriction on values
_ RTEMS_SIMPLE_BINARY_SEMAPHORE - restrict values to 0 and 1, do not allow nested access, allow deletion of locked semaphore.
_ RTEMS_NO_INHERIT_PRIORITY - do not use priority inheritance (default)
_ RTEMS_INHERIT_PRIORITY - use priority inheritance
_ RTEMS_PRIORITY_CEILING - use priority ceiling
_ RTEMS_NO_PRIORITY_CEILING - do not use priority ceiling (default)

_ RTEMS_LOCAL - local task (default)
_ RTEMS_GLOBAL - global task
属性值被特别设计成互斥的，所以位运算OR以及其他的操作是等同的，只要每一个属性仅仅在任务属性成分表中出现一次。缺省的成分表是不需要出现的，虽然一个良好的编程风格最好一一指出。如果要改变缺省设置，需要改变RTEMS_DEFAULT_ATTRIBUTES的缺省属性。
例如设置一个信号量的属性是创建一个本地的信号量，有任务优先级等待队列。那么我们应该这样设置：RTEMS_PRIORITY 或者 RTEMS_LOCAL | RTEMS_PRIORITY.

9.2.6 建立一个SEMAPHORE_OBTAIN

这个属性的特别指rtems_semaphore_obtain命令的
_ RTEMS_WAIT - task will wait for semaphore (default)
_ RTEMS_NO_WAIT - task should not wait

属性值被特别设计成互斥的，所以位运算 OR 以及其他的操作是等同的，只要每一个属性仅仅在任务属性成分表中出现一次。缺省的成分表是不需要出现的，虽然一个良好的编程风格最好一一指出。如果要改变缺省设置，需要改变 RTEMS_DEFAULT_OPTIONS 的缺省属性。

9.3 操作

9.3.1 创建一个信号量

rtems_semaphore_create 命令就可以创建一个二元或是多元信号量。如果一个二元信号量创建的初值为 0，则表示资源已经被占用了，创建这个信号量的任务被看成是这个信号量的拥有者。并且在创建时，等待这个信号量的任务队列的排列顺序也确定了（是按 FIFO 还是按任务优先级）。同时，是否使用优先级继承还是优先级置顶算法也被选择。RTEMS 也会给每个新创建的信号量分配一个信号量控制块（SMCB）。

9.3.2 获取信号量的 ID

当一个信号量被创建了之后，RTEMS 就会给他分配一个唯一的 ID。有两种方法可以获得信号量的 ID：1，通过 rtems_semaphore_ident 命令可以获得这个信号量的 ID,以便在调用其他的信号量命令中使用；2，作为调用 rtems_semaphore_create 命令的结果，ID 将被存放在用户指定的位置。

9.3.3 获取一个信号量

rtems_semaphore_obtain 命令用来获得指定的信号量。算法如下：

```
if semaphore’s count is greater than zero
  then decrement semaphore’s count
  else wait for release of semaphore
return SUCCESSFUL
```

如果不能立即获得信号量,通常有三种处理情况:

- 缺省情况下，任务将会一直等待直到得到信号量。
- 如果属性为RTEMS_NO_WAIT，则立即错误error码，不进行任何等待。
- 设置一个等待时间极限，任务等待的时间超过了这个极限，就返回error码，不再等待。

如果一个任务在等待获得一个信号量,信号量任务等待队列将按照 FIFO 或者优先级来等待。如果采用优先级继承算法，并且当前拥有信号量的任务优先级比等待的低，则把当前的任务优先级增加到等待的任务的优先级。如果在等待过程中，信号量被删除了，所以等待的任务将报错。

9.3.4 释放一个信号量

rtems_semaphore_release 命令来释放指定的信号量。算法如下：

```
if no tasks are waiting on this semaphore
  then increment semaphore’s count
  else assign semaphore to a waiting task
return SUCCESSFUL
```

如果采用了优先级继承或者置顶算法将在信号量释放后，恢复原来的优先级。

9.3.5 删除一个信号量

rtems_semaphore_delete 命令从系统中删除一个信号量。SMCB 将会被回收。所有该信号量等待队列中的任务都被返回一个码表示该信号量被删除了。

10 消息管理

10.1 介绍

消息管理器通过 RTEMS 消息队列来提供通信和同步机制。主要的命令有：

rtems_message_queue_create – 创建一个队列

rtems_message_queue_ident – 获得队列的 ID

rtems_message_queue_delete – 删除一个队列

rtems_message_queue_send – 把消息放在队列的尾部

rtems_message_queue_urgent – 把消息放在队列的首部

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 消 息 的 数 量

rtems_message_queue_flush – 除去消息队列中所有的消息

10.2 背景

10.2.1 消息

消息是用来储存信息的变长缓冲区，用来支持任务间通信的。消息的长度及储存在消息中的信息由用户定义，储存在消息中的信息可以是实际的数据，指针，或者为空。

10.2.2 消息队列

消息队列允许任务和ISR进行通信。

消息队列是有多条消息组成，通常，通常消息队列的发送和接收是按 FIFO 的原则进行的。

你也可以用 rtems_message_queue_urgent 使消息放在队列的头，就变成了 LIFO。

同步通过一个任务等待队列里的一个消息实现。

10.2.3 建立一个消息队列属性

_ RTEMS_FIFO - tasks wait by FIFO (default)
_ RTEMS_PRIORITY - tasks wait by priority
_ RTEMS_LOCAL - local message queue (default)
_ RTEMS_GLOBAL - global message queue

属性值被特别设计成互斥的，所以位运算 OR 以及其他的操作是等同的，只要每一个属性仅仅在任务属性成分表中出现一次。缺省的成分表是不需要出现的，虽然一个良好的编程风格最好一一指出。如果要改变缺省设置，需要改变 RTEMS_DEFAULT_ATTRIBUTES 的缺省属性。

10.2.4 建立一个MESSAGE_QUEUE_RECEIVE命令的属性

RTEMS_WAIT - task will wait for a message (default)
RTEMS_NO_WAIT - task should not wait

属性值被特别设计成互斥的，所以位运算 OR 以及其他的操作是等同的，只要每一个属性仅仅在任务属性成分表中出现一次。缺省的成分表是不需要出现的，虽然一个良好的编程风格最好一一指出。如果要改变缺省设置，需要改变 RTEMS_DEFAULT_OPTIONS 的缺省属性。

10.3 操作

10.3.1 创建一个消息队列

rtems_message_queue_create 命令创建一个消息队列。用户自定义名字，队列的最大长度和最大消息数。用户可以选择 FIFO,还是任务优先级方式作为任务等待队列的管理机制。RTEMS 为消息队列分配一个队列控制块（QCB）来管理消息队列。

10.3.2 获得一个消息队列的 ID

一个消息队列创建后，ETEMS 就会给每个消息队列分配一个唯一的 ID。有两种方法可以获得信号量的 ID：1，通过 rtems_message_queue_ident 命令可以获得这个信号量的 ID,以便在调用其他的信号量命令中使用；2，作为调用 rtems_message_queue_create 命令的结果，ID 将被存放在用户指定的位置。

10.3.3 接收消息

用rtems_message_queue_receive来接收特定消息队列里的消息。如果消息队列里存在消息，就取出消息，复制到调用者的消息缓冲区里，并返回消息的长度。如果消息队列里没有消息，下列3种处理情况：

- 缺省情况下，调用者任务一直等待直到消息队列有新消息。
- 如果属性为RTEMS_NO_WAIT，则不等待，立即返回一个error码。
- 设置一个等待时间极限，如果在这个极限时间内还没有得到消息，就不等待了，返回error码。

如果一个任务要等待一个消息，它就进入消息队列的任务等待队列。在队列中的位置按FIFO或优先权方式决定。

10.3.4 发送一个消息

通过rtems_message_queue_send 和 rtems_message_queue_urgent 来发送一个消息。如果没有任务等待时，这两个命令就会把消息发送到消息队列中。两者的不同就在于，前者把消息放在消息队列的最后方。后者把消息放在消息队列的最前方。

10.3.5 广播一个消息

rtems_message_queue_broadcast 命令发送同一个消息给每一个在任务等待队列里的任务。每个任务的消息缓冲区里都得到了消息，每个任务就被激活。被激活任务的数量返回给调用者。

10.3.6 删除一个队列

rtems_message_queue_delete 命令从系统中删除一个消息队列。它对应的 QCB 及跟它有关的内存缓冲区都被释放了。这个队列中的消息会返回到系统消息缓冲区。所有该消息等待队列中的任务都被返回一个码表示该信号量被删除了。

11 事件管理

11.1 介绍

事件管理为任务间的通信和同步提供了一个高性能的方法。有两个命令：

1) rtems_event_send – 发送一个事件
2) rtems_event_receive – 接收事件

11.2 背景

11.2.1 事件集

任务（或 ISR）通过事件标志（Event Flag）来通知另一个任务一个重要状况的发生。一个任务与 32 个事件标志相联系，一个或几个事件标志组合起来就构成了一个事件集（event set）。数据类型 rtems_event_set 用来管理事件集。

1) 事件提供了同步的机制
2) 事件的目标是任务
3) 一个任务可以同时等待多个事件
4) 事件之间是独立的
5) 事件不保持和传递数据
6) 事件不能够排队（ 也就是说同一个事件在没被接收时发送给同一个任务多次，除了第一个事件，后面的都没有作用，任务只接收第一个事件。）

一个事件集（event set）发送给了一个任务，这个 event set 就是 posted。如果它已经 posted 的了，但是没被任务接收，它就是 pending 的。一个任务的事件条件是否被满足有两种决定算法:

- RTEMS_EVENT_ANY,只要一个要求的事件 posted,就满足了。
- RTEMS_EVENT_ALL, 所有的要求的事件都 posted 才能满足。

11.2.2 建立一个事件集或条件

通过位运算 OR，事件集建立一个期望的事件集。有效的事件集从 RTEMS_EVENT_0 到 RTEMS_EVENT_31。事件被特别设计成互斥的，所以位运算 OR 以及其他的操作是等同的，只要每一个事件仅仅在事件集表中出现一次。

例如，我们要发送事件集包括事件6，15和31。那么发送命令应该是RTEMS_EVENT_6 |

RTEMS_EVENT_15 | RTEMS_EVENT_31.

11.2.3 建立一个事件接收属性

通常，通过位运算 OR，建立一个期望的接收属性。有效的选择有:

_ RTEMS_WAIT - task will wait for event (default)
_ RTEMS_NO_WAIT - task should not wait
_ RTEMS_EVENT_ALL - return after all events (default)
_ RTEMS_EVENT_ANY - return after any events

属性值被特别设计成互斥的，所以位运算 OR 以及其他的操作是等同的，只要每一个属性仅仅在任务属性成分表中出现一次。缺省的成分表是不需要出现的，虽然一个良好的编程风格最好一一指出。如果要改变缺省设置，需要改变 RTEMS_DEFAULT_OPTIONS 的缺省属性。

11.3 操作

11.3.1 发送一个事件集

rtems_event_send命令允许一个任务（ISR）将一个事件集发送给一个目标任务。根据目标任务的状态有两种情况:

A 目标任务正在等待一个事件的发生而处于阻塞状态。

如果这个发送的事件集满足了等待的任务的事件条件，等待任务就变为就绪状态。

如果这个等待的任务的事件条件没被满足，这个事件集是pending的，等待的任务仍处于阻塞状态。

B 目标任务没有等待事件

这个事件集被posted并处于pending状态。

11.3.2 接收一个事件集

rtems_event_receive命令用来使得一个任务接收一个特定的事件集。如果这个任务被接收的事件集所满足，一个successful返回码立即返回。如果任务的事件条件没有被满足，会出现3种情况：

- 缺省条件下，任务将会一直等待直到事件条件满足。
- 如果属性选为 RTEMS_NO_WAIT, 任务不等待，立即返回 error 码。
- 设定一个等待时间极限，如果等待的时间超过了极限，就不等待了，返回一个 error 码。

11.3.3 确定一个 pending 的事件集

通过调用rtems_event_receive命令（给定输入事件条件的值为RTEMS_PENDING_EVENTS）一个任务可以确定一个pending的事件集。Pending事件集将返回给调用任务，而事件集将不会改变。

11.3.4 接收所有的 pending 事件

通过调用rtems_event_receive命令（给定输入事件条件的值为RTEMS_ALL_EVENTS和接收可选属性为RTEMS_NO_WAIT | RTEMS_EVENT_ANY）一个任务可以接收所有当前的pending事件集。Pending的事件将返回给调用者，并且清除事件集。如果没有事件是pending的，那么将返回条件代码RTEMS_UNSATISFIED。

12 信号管理

12.1 介绍

信号管理是用来提供异步通信机制的。命令有：

rtems_signal_catch – 建立一个 ASR

rtems_signal_send – 给一个任务发送信号

12.2 背景

12.2.1 信号管理定义

信号管理器允许任务选择性的定义一个ASR(Asynchronous Signal Routine)异步信号子程序。类似于一个应用的ISR。当处理器中断的时候，应用的执行也会被中断然后ISR开始控制。类似的，当一个信号发送给一个任务时，这个任务的执行就会被ASR“中断”。

任务都有信号标志（signal flag），来通知另一个任务重要状况的发生。每个任务都跟32个信号标志相联系。从RTEMS_SINGNAL_0到RTEMS_SINGNAL_31。一个或多个信号标志组成一个信号集（signal set）。

一个signal set发送给一个任务,它就是posted，但是没有被处理，这个signal set就是pending的。

12.2.2 比较 ASR 和 ISR

虽然ASR跟ISR很相似,但是也有不同,主要表现为:

- ISR 是硬件处理器调度的；ASR 是 RTEMS 调度的。
- ISR 不在任务的转换中执行而且仅仅调用命令的子集。；ASR 在任务的转换中也会执行，而且它有可能调用所有的命令。
- ISR 被调用时，中断向量号作为参数传递；ASR 被调用时，信号集作为参数传
- ASR 有任务的执行模式，并且这个模式可以与当前的任务不同；ISR 不像任务一样执行，所以，就没有任务的执行模式。

12.2.3 建立一个信号集

通过位运算 OR，信号集建立一个期望的事件集。有效的信号集从 RTEMS_EVENT_0 到 RTEMS_EVENT_31。信号被特别设计成互斥的，所以位运算 OR 以及其他的操作是等同的，只要每一个信号仅仅在信号集表中出现一次。

例如，我们要发送信号集包括信号6，15和31。那么发送命令应该是RTEMS_SIGNAL_6 | RTEMS_SIGNAL_15 | RTEMS_SIGNAL_31.

12.2.4 建立一个 ASR 模式

通过位运算 OR，建立一个期望的 ASR 模式。

_ RTEMS_PREEMPT is masked by RTEMS_PREEMPT_MASK and enables preemption

_ RTEMS_NO_PREEMPT is masked by RTEMS_PREEMPT_MASK and disables preemption

_ RTEMS_NO_TIMESLICE is masked by RTEMS_TIMESLICE_MASK and disables timeslicing

_ RTEMS_TIMESLICE is masked by RTEMS_TIMESLICE_MASK and enables timeslicing

_ RTEMS_ASR is masked by RTEMS_ASR_MASK and enables ASR processing

_ RTEMS_NO_ASR is masked by RTEMS_ASR_MASK and disables ASR processing

_ RTEMS_INTERRUPT_LEVEL(0) is masked by RTEMS_INTERRUPT_MASK and enables all

interrupts
_ RTEMS_INTERRUPT_LEVEL(n) is masked by RTEMS_INTERRUPT_MASK and sets interrupts level n
属性值被特别设计成互斥的，所以位运算 OR 以及其他的操作是等同的，只要每一个属性仅仅在任务属性成分表中出现一次。缺省的成分表是不需要出现的，虽然一个良好的编程风格最好一一指出。如果想更改缺省配置，需要使用 DEFAULT_MODES。

12.3 操作

12.3.1 建立一个 ASR

使用rtems_signal_catch命令可以为调用者任务建立一个ASR。ASR的地址和执行模式在这个命令中定义。并且它的执行模式跟任务的模式是可以不同的。例如一个任务可以是有优先级的，但是它的ASR却可以是没有的。除非使用rtems_signal_catch命令来建立一个ASR，否则它的ASR是无效的，没有信号集会发给这个任务。

一个任务可以通过设置rtems_signal_catch命令中的ASR的地址为NULL来使ASR无效，并且舍弃该任务的所有pending信号。一个任务的ASR是无效的，新发给它的信号集都会被丢弃。

任务也可以通过设置任务的执行模式为RTEMS_NO_ASR来使ASR无效。如果一个任务的ASR被设置成禁止，那么所有的信号都会被pending，直到ASR被使能后才执行。

一个任务可以使用的任何命令，ASR 也可以使用。一个任务仅仅允许一个活动的 ASR。因此，每次调用 rtems_signal_catch 命令将覆盖前面的 ASR。
通常，ASR 的执行模式都禁止信号处理，如果 ASR 的信号处理被使能，那么 ASR 必须重入。

12.3.2 发送一个信号集

使用rtems_signal_send命令可以使任务和ISR发送信号给一个目标任务。发送一个信号给任务不会影响任务当前的状态。如果这个任务当前不是执行的任务，那么信号先保持pending状态。当任务被执行时，信号才会被ASR来处理。同ISR, ASR也是可以嵌套的。

信号也不能够排队（ 也就是说同一个信号在没被接收时发送给同一个任务多次，除了第一个信号，后面的都没有作用，任务只接收第一个信号。）

12.3.3 处理一个 ASR

ASR具有一些实现软件中断的能力。软件中断和硬件中断是并行处理的。ASR的执行有下面的原形：

```
rtems_asr user_routine(
    rtems_signal_set signals
);
```

当ASR返回到RTEMS, 被中断任务的执行模式,执行路径恢复到进入ASR之前的情况。

13 分区管理

13.1 介绍

分区管理是提供在固定大小的单元里动态分配内存的管理。命令有：

1) rtems_partition_create – 创建一个分区
2) rtems_partition_ident – 获取分区的 ID
3) rtems_partition_delete – 删除一个分区

4) rtems_partition_get_buffer – 从分区中获取一个缓冲区

5) rtems_partition_return_buffer – 把缓冲区返回到分区中

13.2 背景

13.2.1 分区管理定义

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,并且这些缓冲区可以被动态的分配和回收。

分区是以缓冲区链表形势进行管理的。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 8 字节作为空缓冲区链表。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，改变一个在以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以分配的缓冲区的内容。

13.2.2 建立一个分区属性集

通过位运算 OR，建立一个期望的分区属性集。

RTEMS_LOCAL - local task (default)

RTEMS_GLOBAL - global task

属性值被特别设计成互斥的，所以位运算 OR 以及其他的操作是等同的，只要每一个属性仅仅在任务属性成分表中出现一次。缺省的成分表是不需要出现的，虽然一个良好的编程风格最好一一指出。如果想更改缺省配置，需要使用 RTEMS_DEFAULT_ATTRIBUTES。

13.3 操作

13.3.1 建立一个分区

通过 rtems_partition_create 命令将以用户特定的名字来创建一个分区。同时分区的名字，起始地址，长度，缓冲区的大小都在这个命令中定义。RTEMS 为每个分区分配一个分区控制块（PTCB）,来管理每一个新创建的分区。每个分区中的缓冲区的数量可以通过分区长度和缓冲区的大小来决定。

13.3.2 获得一个分区的 ID

当一个分区创建后，RTEMS 就为它产生一个唯一的分区 ID，直到分区被删除。分区 ID 在调用其他分区命令时会用到，有两种方法可以获得它的 ID：1，通过 rtems_partition_ident 可以获得分区的 ID。2，作为调用 rtems_partition_create 的结果，ID 将被保存在用户定义的路径里。

13.3.3 获得一个缓冲区

通过 rtems_partition_get_buffer 命令可以获取一个缓冲区。如果获取成功，那么就立即返回 successful 码。如果没成功，就返回 unsuccessful

13.3.4 释放一个缓冲区

通过rtems_partition_return_buffer命令可以释放缓冲区，并且这个缓冲区返回到空缓冲区链的尾部。如果要释放的缓冲区之前还没在这个分区中被分配，那么就返回一个error码.

13.3.5 删除一个分区

通过rtems_partition_delete命令可以把一个分区删除。当一个分区被删除了，这个分区的PTCB也要返回到空PTCB表中。一个已经分配了缓冲区的分区是不可以被删除的。如果要删除他，就会返回一个error码。

14 区域管理

14.1 介绍

区域管理是在不同大小的单元里动态分配内存的管理。主要的命令有：

1) rtems_region_create – 创建一个区域
2) rtems_region_ident – 获取区域的ID
3) rtems_region_delete – 删除一个区域
4) rtems_region_extend – 扩展一个区域
5) rtems_region_get_segment – 从区域中获取一段
6) rtems_region_return_segment – 把一段返回到区域中
7) rtems_region_get_segment_size – 获取段的大小

14.2 背景

14.2.1 区域管理定义

一个区域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，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，那么就分配一个512字节的段。

区域采用不同大小内存块的双链表管理。采用"first-fit"算法。从链表的头检查，如果匹配就分配。当一个段返回到区域时，这个空闲的块就跟他邻接的（如果也是空闲的话）块接合，成为一个更大的空闲块。由于各个块是可以自动接合的，所以一个区域的段数也是动态变化的。

14.2.2 建立一个属性集

通常，通过位运算OR，建立一个期望的属性。有效的选择有:

RTEMS_FIFO - tasks wait by FIFO (default)
RTEMS_PRIORITY - tasks wait by priority

属性值被特别设计成互斥的，所以位运算OR以及其他的操作是等同的，只要每一个属性仅仅在任务属性成分表中出现一次。缺省的成分表是不需要出现的，虽然一个良好的编程风格最好一一指出。如果要改变缺省设置，需要改变RTEMS_DEFAULT_ATTRIBUTES的缺省属性。

14.2.3 建立一个选项集

通常，通过位运算OR，建立一个期望的选项。有效的选择有:

RTEMS_WAIT - task will wait for semaphore (default)
RTEMS_NO_WAIT - task should not wait

属性值被特别设计成互斥的，所以位运算OR以及其他的操作是等同的，只要每一个属性仅仅在任务属性成分表中出现一次。缺省的成分表是不需要出现的，虽然一个良好的编程风格最好一一指出。如果要改变缺省设置，需要改变RTEMS_DEFAULT_OPTIONS的缺省属性。

14.3 操作

14.3.1 创建一个区域

通过rtems_region_create命令可以创建一个区域。用户可以选择使用FIFO还是任务优先级算法作为任务等待队列中的任务调度算法。RTEMS为每个区域分配一个区域控制块（RNCB）来管理新建的这个区域。

14.3.2 获得一个区域的 ID

当一个区域创建后，RTEMS 就为它产生一个唯一的区域 ID，直到区域被删除。区域 ID 在调用其他区域命令时会用到，有两种方法可以获得它的 ID：1，通过 rtems_region_ident 可以获得区域的 ID。2，作为调用 rtems_region_create 的结果，ID 将被保存在用户定义的路径里。

14.3.3 区域扩展

通过rtems_region_extend命令可以为一个已经存在的区域添加内存。调用者定义要添加的内存的大小和起始地址。

14.3.4 获取一个段

通过rtems_region_get_segment可以从一个特定的区域中获取一个段。如果这个区域有足够的空闲区域，那么这个段就获取成功。如果没有合适的段被分配，有下列3种可能的情况：

- 缺省情况下，调用者任务将会一直等待直到获得了需要的段。
- 如果定义了RTEMS_NO_WAIT选项，任务将不会做任何等待，立即返回一个error码。
- 定义一个任务等待的极限时间,当等待的时间超过了这个极限,就返回一个error码，不再等待。

在任务等待队列中的任务可以按照 FIFO 或是优先级顺序排列的，这个由用户定义。

14.3.5 释放一个段

通过rtems_region_return_segment命令可以使一个分配的段返回到区域中。这个段跟它两边邻接的段接合成为一个长度更长的空闲段。然后，等待队列中的第一个任务检查现在是否有一个段可以满足要求。如果有，就分配段，并且该任务被取消阻塞状态。如果没有，继续等待。

14.3.6 获取一个段的长度

通过 rtems_region_get_segment_size 命令可以获得一个特定段的大小。

14.3.7 删除一个区域

通过 rtems_region_delete 命令可以把一个区域删除。当一个区域被删除了，这个区域的 RNCB 也要返回到空 RNCB 表中。一个已经分配了段的区域是不可以被删除的。否则，就会返回一个 error 码。

15 双端口内存管理（Dual-Ported Memory Manager）

15.1 介绍

双端口内存管理提供了一个双端口内存区域（DPMA）内外地址转换的机制。提供的命令有：

1) rtems_port_creat – 创建一个端口
2) rtems_port_ident – 获得端口的 ID
3) rtems_port_delete – 删除一个端口
4) rtems_port_external_to_internal – 外部地址转换成内部地址
5) rtems_port_internal_to_external – 内部地址转换成外部地址

15.2 背景

双端口内存区域（DPMA）是被一个特定的处理器拥有的RAM块，并且这个RAM块还可以被系统

中的其他的处理器访问。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：

1. 一种是紧偶合的多处理器系统，双端口内存将在所有的节点之间被共享，并被用来做内部通讯。
2. 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。

15.3 操作

15.3.1 创建一个端口

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 同时也分配一个独一无二的 DPMA ID。

15.3.2 获取端口的 ID

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 ID。端口 ID 在调用其他的端口命令时用到，可以通过 rtems_port_ident 命令来获取。2，作为调用 rtems_port_create 的结果，ID 将被保存在用户定义的路径里。

15.3.3 转换地址

rtems_port_external_to_interna命令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命令可以把一个特定端口的内部地址转换成外部地址。如果试图转换一个 DPMA 之外的地址，那么这个要被转换的地址将被返回。

15.3.4 删除一个 DPMA 端口

通过rtems_port_delete命令可以把一个端口从系统中删除。当一个端口被删除后，它的DPCB返回到DPCB的空闲表中。

16 I/O 管理

16.1 介绍

I/O 管理为设备驱动的访问提供了一种完好定义的机制，并为设备驱动的组织提供了一个结构化的方法。主要的命令为：

1) rtems_io_initialize – 初始化设备驱动
2) rtems_io_register_name – 登记一个设备驱动名字
3) rtems_io_lookup_name – 查找一个设备驱动名字
4) rtems_io_open – 打开一个设备
5) rtems_io_close – 关闭一个设备
6) rtems_io_read – 读设备
7) rtems_io_write – 写设备
8) rtems_io_control – 特殊的设备服务（跟对应的驱动有关）

16.2 背景

16.2.1 设备驱动表

每一个使用 RTEMS 的 I/O 管理的应用都必须在系统配置表中定义设备驱动表的地址。这个表包含了由 RTEMS 在初始化的时候初始化的设备驱动入口点。每一个设备驱动包括下列入口

点：

- Initialization
- Open
- Close
- Read
- Write
- Control

如果一个设备不支持入口点，那么在 Configuration Table 中我们就将它设为 NULL。

应用程序可以同 rtems 的 i/o 管理程序登记或注销驱动程序。这样就可以避免所有驱动程序都要同这个表静态的连接或断开。

Confdef.h中的CONFIGURE_MAXIMUM_DRIVERS是对于一个应用有效的驱动数量。

16.2.2 主要和次要的设备值

每次调用 I/O 管理器的时候，都必须提供主要和次要的设备值作为声明。主要数量是指要请求的设备在设备驱动表中的线索值，它也用来选择一个特别的设备。准确的使用次要值是由驱动特别决定的，它用来区分由同一个驱动控制的一些设备。

数据类型rtems_device_major_number和rtems_device_minor_number用来维护系统主要和次要的设备值。

16.2.3 设备名

I/O 管理器给每一个设备关联一个设备名。并提供用来注册设备名的命令，和利用这个名字来查询主要和次要的设备值。

16.2.4 设备驱动环境

应用的开发者和驱动的开发者必须注意下面的因素：

- 在调用任务切换的时候执行设备驱动,所以如果一个驱动阻塞了,调用的任务也将阻塞。
- 设备驱动程序可以自由的改变任务的执行模式，虽然在它必须恢复任务原有的执行模式。
- 设备驱动程序可以由 ISR 调用。
- I/O 管理器只可以访问本地的设备驱动
- 设备驱动程序调用所有的 RTEMS 命令,包括 I/O 管理命令,既可以对本地也可以对全局。

虽然 RTEMS 为设备驱动提供了一个框架结构，但是并不认为驱动一定要采用这种结构。

16.2.5 运行时驱动注册

BSP 和应用开发者可以选择是否让驱动一开始就静态的和设备驱动表相连，还是动态的在运行时注册。

动态的注册有助于应用，当：

- 对于一个特别的目标板，BSP 和核内的库对于一系列应用是通用的。一个应用将由一个有所有驱动的通用库来建立。应用选择驱动和注册驱动。统一的驱动名查找保护应用。
- 在初始化的时候，当应用在总线上查找时，驱动的名字和范围是变化的。
- 支持 hot swap 总线系统例如 Compact PCI。
- 支持运行十可载入的驱动模块。

16.2..6 设备驱动接口

当一个应用调用 I/O 管理命令的时候，RTEMS 要决定哪一个设备驱动入口点要被调用。RTEMS 调用的设备驱动入口点被假定和下列原型是兼容的：

```
rtems_device_driver io_entry(
```

```
        rtems_device_major_number major,
        rtems_device_minor_number minor,
        void *argument_block
    );
```
推荐不要让驱动程序产生报错代码，因为这样将会和应用的部分相抵触。通常产生报错代码的手段是用状态中最重要的一部分来指出驱动特别的代码。

16.2.7 设备驱动初始化

RTEMS自动的初始化所有的设备驱动，当多任务通过rtems_initialize_executive初始化的时候。RTEMS通过调用每一个驱动的入口点来初始化，入口点有如下参数：

major the major device number for this device driver.

minor zero.

argument_block will point to the Configuration Table.

返回的代码将被RTEMS忽略，如果不能正常初始化，那么将调用fatal error occurred 命令。

16.3 操作

16.3.1 注册和查询名字

命令 rtems_io_register 使得一个特定的设备名字跟这个设备的 major/minor number 相关联。命令 rtems_io_lookup 是用来对于一个给定设备名字,来找到跟它相对应的 major/minor number。(其中，一个设备的 major number 是指定设备的设备驱动表的入口地址的索引，用来选择一个特定的设备驱动。Minor number 是用来区分同一个驱动控制的各个设备。)

16.3.2 访问一个设备驱动

I/O管理可以使得应用程序在标准方式下使用设备驱动。rtems_io_initialize, rtems_io_open，rtems_io_close, rtems_io_read, rtems_io_write, rtems_io_control 都是访问设备驱动的命令。

17 致命错误管理器

17.1 介绍

致命错误管理用来处理所有的致命的和无法挽救的错误。提供的命令：

1) rtems_fatal_error_occurred – 调用致命错误处理器

17.2 背景

当 RTEMS 或者应用发现一个不可挽回的错误，将调用致命错误处理器。致命错误处理器可以由下列 3 种源调用：

- the executive (RTEMS)
- user system code
- user application code

每个状态和用户动态扩展表将有可能会包括致命错误处理器。在静态扩展表中的致命错误处理器可以提供对目标硬件上的 debugger 和 monitor 访问。如果任何用户提供的致命错误处理器被安装了，那么致命错误处理器将调用它们。如果没有安装或者用户的致命错误处理器将控制权都返回给 RTEMS，系统将调用默认的致命错误处理器，那么系统将被标记为失败。

虽然致命错误处理器的具体操作跟处理器的类型有关,但是通常它会屏蔽所有可屏蔽的中断,放入一个错误码到处理器对应的堆栈或寄存器中,然后停机。其他的具体操作,可以在特定的

处理器应用辅助文档中找到。

17.3 操作

17.3.1 声明一个致命错误

当一个致命错误被发现时,rtems_fatal_error_occurred命令就被激活。在调用任何户提供的致命错误处理器或者默认的致命错误处理器之前,rtems_fatal_error_occurred命令将把用户信息存储在variable_Internal_errors_What_happened里。这个结构包括下面3个准确的信息：

- 错误源（API或者执行核）
- 错误是否是内部执行产生
- 错误类型

错误类型依赖于错误源和错误是否是内部执行产生的。如果错误是由于API产生的，那么错误代码将是API的错误码或状态码。RTEMS的API状态码在c/src/exec/rtems/headers/status.h里。对于POSIX API的可以在<errno.h>里找到。

rtems_fatal_error_occurred不仅可以激活RTEMS的致命任务处理器，还可以激活用户提供的致命任务处理器。

如果一个应用要求有更复杂的错误处理，就可以用户自定义相应的处理。自定义的处理器要在RTEMS的配置表中声明。然后，把处理程序的地址写到用户扩展表的致命区域，这样当出现致命错误的时候，rtems_fatal_error_occurred就可以调用用户自定义的处理了。如果没有用户自定义的处理或者用户的致命错误处理器将控制权都返回给RTEMS，系统将调用默认的致命错误处理器。那样将会导致停机。

18 调度

18.1 介绍

调度对于实时应用是非常关键的。

调度是目的就是为了能够更好的使用处理器和分配资源。RTEMS 采用了基于优先级的，占先式的算法，在同优先级时采用轮循。这样的目的就是使在任何时刻，在处理器上运行的任务都是就绪队列里的最高级。

有两种调度机制，都需要任务在就绪队列里是一个链表。一种方法是随机的将任务放到就绪链表中，然后让调度机来扫描所有的任务来决定哪一个任务应该得到处理器。另一种方法是按照调度的标准在将任务插入就绪任务链表的时候就把他们排好序。这样当处理器空闲的时候，在链表中的第一个任务将得到执行。RTEMS 采用了后者。

18.2 调度机制

RTEMS 提供了 4 种机制：

- user-selectable task priority level　　用户可选任务优先级
- task preemption control　　任务占先控制
- task timeslicing control　　任务时间片控制
- manual round-robin selection　　手动轮循选择

虽然这 4 个机制是独立的，但是他们还是有优先顺序的。优先顺序按照上面的顺序排列。

18.2.1 任务优先级和调度

最重要是机制就是应用可以为每一个独立的任务分配一个优先级,并在运行的时候还可以改变它。RTEMS 支持 255 的优先级。当一个任务被放到就绪链表的时候，它将被放到具有相同

优先级的最后一个。这就是说，如果任务的优先级是相同的，它们将按照 FIFO 的原则执行。RTEMS 的调度机将选择就绪队列里的最高级任务，然后在处理器上运行。

18.2.2 占先

用户可以通过操作独立任务的占先模式标志(RTEMS_PREEMPT_MASK)来改变基本的调度算法。如果优先级被禁止了即preemption被设为disabled (RTEMS_NO_PREEMPT), 只要任务处于执行状态，他就不会释放对处理器的控制权，直到执行结束，或被阻塞，或者重新允许占先，这时即使是有一个更高优先级的任务处于就绪状态也不会被执行。注意到占先设置对于任务的调度方法是没有影响的，它仅影响一个已经控制了处理器的任务。

18.2.3 时间片

时间片 timeslicing 元素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同等优先级的任务.如果当前就绪队列里面的任务优先级比这个任务低，那么这个任务将再被分配一个时间片继续执行。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 元素就被忽略。如果一个更高优先级的任务就绪了,那么当前的任务将被放弃，即使它的时间片还没有用完。

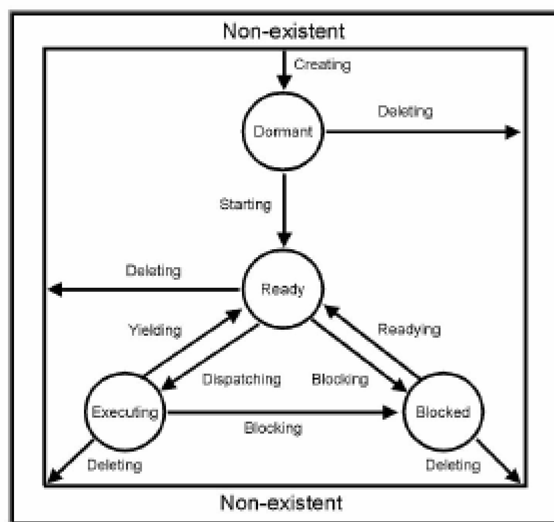
18.2.4 手动轮循

通过rtems_task_wake_after (指定时间间隔为RTEMS_YIELD_PROCESSOR) 命令来调用手动轮循。当前的任务将立刻放弃对处理器的拥有并返回到和它同优先级的队尾，如果没有优先级都比它低，将不放弃处理器。

18.2.5 分配任务

分配机是 RTEMS 用来把处理器分配给一个已经就绪的任务的。这就涉及到一个任务转换的问题。任务转换就是将当前的任务的运行环境储存起来，然后恢复将要执行的任务的环境。因为 RTEMS 的设计是为了嵌入式是应用,所以我们设计的时候避免了不必要的开支在存储和恢复数字协处理器的运算状态上。仅当一个 RTEMS_FLOATING_POINT 任务被分配并且这个任务不是最后一个使用数字协处理器的时候，才存储数字协处理器的状态

18.3 任务状态转换



RTEMS 的任何一个任务都必须处在这 5 中状态中的一种。如果没有创建那么就是不存在，或

者被删除也将成为不存在状态。

一个任务被创建了之后就进入休眠状态，虽然这个任务在系统中是存在的，但是却不会去竞争任何资源。直到执行了 rtems_task_start 命令，这个任务就进入就绪状态。

如果这个任务不能够被调度来运行，那么就将进入阻塞状态。下列命令都将导致一个任务被阻塞：

- 任务调用了rtems_task_suspend挂起命令，将可以挂起自己或者其它任务。
- 运行中的任务执行了rtems_message_queue_receive命令收到一个等操作或者消息队列是空的。
- 运行中的任务执行了rtems_event_receive命令收到一个等操作或者当前的等待事件不能满足需求。
- 运行中的任务执行了rtems_semaphore_obtain命令收到一个等操作或者需求的信号量是无效的。
- 运行中的任务执行了rtems_task_wake_after命令来阻塞任务等待一个给定的时间。如果时间为0，任务将不阻塞而进入就绪状态。
- 运行中的任务执行了rtems_task_wake_when命令将阻塞任务直到需要的时间到来。
- 运行中的任务执行了rtems_region_get_segment收到一个等操作或者没有足够大发内存段来满足任务的需求。
- 运行中的任务执行了rtems_rate_monotonic_period命令而且必须要等待特定的单调速率周期来决定。

一个已经阻塞的任务也可以被挂起，所以任务要进入就绪状态，必须清除阻塞和挂起两个条件。任务就绪并没有开始被处理器操作，如果任务被阻塞，用完时间片或者删除处理器都将放弃当前任务的执行。相同优先级的任务用FIFO策略排队执行。下列命令将导致一个任务进入就绪状态：

运行中的任务执行了rtems_task_resume命令，且这个任务虽然被挂起了但是没有因为等待任何资源而阻塞。

运行中的任务执行了rtems_message_queue_send, rtems_message_queue_
broadcast, 或者 rtems_message_queue_urgent命令,发出一个消息到那个阻塞任务正等待的队列。

运行中的任务执行了rtems_event_send命令发出一个事件条件到那个正等待这个事件条件的阻塞任务那。

运行中的任务执行了rtems_semaphore_release命令，释放了一个信号量到正等待这个信号量的阻塞任务那。

一个执行了rtems_task_wake_after的任务已经等待到了足够长的时间。

一个执行了rtems_task_wake_when的任务已经等待到了足够长的时间。

运行中的任务执行了rtems_region_return_segment释放了一个内存段，从而使得正在等待的任务有了足够大的执行所需内存空间。

一个执行了rtems_rate_monotonic_period的任务已经等待到了它的周期终止。

_ A timeout interval expires for a task which was blocked waiting on a message, event, semaphore, or segment with a timeout speci_ed.

_ A running task issues a directive which deletes a message queue, a semaphore, or a
region on which the blocked task is waiting.

_ A running task issues a rtems_task_restart directive for the blocked task.

_ The running task, with its preemption mode enabled, may be made ready by issuing any of the directives that may unblock a task with a higher priority. This directive may be issued from the running task itself or from an ISR.
A ready task occupies the executing state when it has control of the CPU. A task enters the executing state due to any of the following conditions:
_ The task is the highest priority ready task in the system.
_ The running task blocks and the task is next in the scheduling queue. The task may be of equal priority as in round-robin scheduling or the task may possess the highest priority of the remaining ready tasks.
_ The running task may reenable its preemption mode and a task exists in the ready queue that has a higher priority than the running task.
_ The running task lowers its own priority and another task is of higher priority as a
result.
_ The running task raises the priority of a task above its own and the running task is
in preemption mode.

19 速率单调管理

19.1 介绍

速率单调管理是实现周期性任务执行的设备。它可以支持那些使用速率单调调度算法来保证周期性任务即使是在短暂的超负荷的条件下也可以满足 deadline 的应用。主要的命令：

1) rtems_rate_monotonic_create – 创建一个速率单调周期
2) rtems_rate_monotonic_ident – 获得周期的 ID
3) rtems_rate_monotonic_cancel – 取消一个周期
4) rtems_rate_monotonic_delete – 删除一个周期
5) rtems_rate_monotonic_period – 结束当前的/开始下一个周期
6) rtems_rate_monotonic_get_status – 获取周期的状态信息

19.2 背景

速率单调管理提供了对于周期性任务的管理。这个管理器用来支持那些采用速率单调调度算法（RMS）来确保他们的任务能够满足deadline的开发，即使是在短暂的超负荷的条件下。在硬实时系统中，RMS提供的服务可能被任何周期性的应用所使用。

19.2.1速率单调管理的需求

需要一个时钟tick。

19.2.2速率单调管理定义

一个周期性的任务是以规则的时间间隔来执行的。时间的间隔和它的周期有关。周期性的任务可以由它们是执行时间和时间间隔来区别。周期和执行的时间可以用来确定任务对处理器的使用。通常，即使在最坏的情况下，一个任务的执行时间也应该比它的周期要小。例如一个周期的任务是每100毫秒执行10毫秒。

而一个非周期的任务则不需要那么严格的deadline。例如，一个接收用户在终端输入的任务。

然而，也有一些任务是非周期的但是却要求一个严格deadline和一个最小的间隔到达时间。最小的间隔到达时间指的是任务反复出现的最小周期。例如，用户反复的按了游戏控制杆上的发射键，虽然控制杆上的机械特性保证了连续动作的最小周期，但是子弹必须在用户按下后发射（要满足硬实时）。

19.2.3速率单调算法

对于实时系统，速率单调调度算法是非常重要的，它确保了一系列任务是可以调度的。当一系列任务都可以满足它们的deadline的时候，我们说这些任务是可调度的。速率单调调度算法（RMS）就为一系列任务提供了可调度性分析——在最坏的情况下任务是否是可调度的，而且这个算法强调系统的可预测性。它被证明：

RMS是一个优化的最佳静态优先级算法，用来调度在一个处理器上的独立的，可占先的，周期的任务。

RMS是优化的,意味着如果一系列任务可以被静态优先级算法调度,那么也就可以被RMS调度。RMS根据任务的周期决定任务的优先级，周期越短，优先级越高。当两个任务有同样的优先级时，RMS就不区分两者。但是当同样优先级的任务在等待时，RMS将会调度等待时间最长的那个任务。例如：

Task	Period (in milliseconds)	Priority
1	100	Low
2	50	Medium
3	50	Medium
4	25	High

许多应用的任务既有硬实时，也有软实时的，通常一个硬实时的任务对 deadline 要求是非常严格的。所以我们对于硬实时的任务采用 RMS 算法,即使先暂时放弃对软实时任务的执行。我们将把更高的优先级分配给那些紧急任务，虽然 RMS 也可以给非紧急任务分配优先级，但是那没有必要。在这里，可调度特指那些紧急的任务。

19.2.4 可调度性分析

RMS 允许应用设计者确保任务可以满足 deadline，即使有短暂的过载，而不准确的知道什么时候一个任务将通过申请以被证明的可调度分析规则来执行。

19.2.4.1 假设

RMS 的可调度分析规则由下列假设发展出来：

对于所有任务的请求的硬 deadline 是周期性存在的，并且在请求之间有一个固定的间隔。

每一个任务必须在下一个请求出现之前完成。

任务之间是独立的，一个任务不依赖于别的任务的请求的开始和完成。

每一个任务的执行时间是固定的，中间没有占先和中断。

在系统中任何非周期性的任务都是特别的。这些任务执行的时候取代周期任务，而且没有硬和紧急的deadline。

19.2.4.2 处理器使用规则

处理器的使用规则依赖于每一个任务的执行时间Time(index)和周期Period(index)。计算规则：

```
Utilization = 0
```

```
for index = 1 to maximum_tasks
Utilization = Utilization + (Time(index)/Period(index))
```

为了确保周期性任务即使是在短暂的超负荷的条件下也可以满足deadline,处理器的使用必须遵循下面的规则：

```
Utilization = maximum_tasks * (2^(1/maximum_tasks) - 1)
```

当任务增加的时候上面的公式将会接近Ln(2)大约是0.693。平均处理器的使用极限是0.88。

19.2.4.3 处理器使用规则例子

Task	RMS Priority	Period	Execution Time	Processor Utilization
1	High	100	15	0.15
2	Medium	200	50	0.25
3	Low	300	100	0.33

计算上面的处理器使用：全部的开销是0.73<3 * (2^(1/3) - 1)=0.779？

所以这个应用的可以调度的。

19.2.4.4 第一 deadline 规则

如果任务集不满足处理器使用规则，就是说它已经超过了处理器使用上限，我们仍然可以满足 deadline。这需要我们使用第一 deadline 规则。

对于一系列独立的任务，如果同时开始所有的任务后，每一个任务都能满足它的第一 deadline，那么在开始之后的任何时间内 deadline 都将会被满足。

这个规则的关键就是要求所有的任务在同一时间开始。这看起来是不可能的，但是 RTEMS 却很容易的保证了它。通过开始一个非占先的用户初始化任务，所有的任务不论优先级，都可以在初始化任务删除自己之前被创建或开始。这个技术保证了所有的任务可以在同一时刻开始竞争资源，这一刻就是初始化任务删除自己后。

19.2.4.5 第一 deadline 规则例子

Task	RMS Priority	Period	Execution Time	Processor Utilization
1	High	100	25	0.25
2	Medium	200	50	0.25
3	Low	300	100	0.33

计算上面的处理器使用：全部的开销是 0.83>3 * (2^(1/3) - 1)=0.779？

这样处理器使用规则就不能确保它是可以调度的，不满足 RMS 的算法。我们必须使用第一 deadline 规则。

Deadline Time	Task 1	Task 2	Task 3	Total Execution Time	All Deadlines Net?
100	1	1	1	25 + 50 + 100 = 175	NO
200	2	1	1	50 + 50 + 100 = 200	YES

可以看出，在 time100 的时候，任务 1 和任务 2 已经执行结束了，满足了它们的 deadline。然后剩余 25 给任务 3 执行。到此时，任务 3 还没有执行结束。

在 time200 的时候，任务 1 必须要满足它的第二次 deadline，任务 2 应该满足它的第一次 deadline，已经满足了。到这个时候，任务 1 要用 50，任务 2 也用了 50，那么剩下 100，刚好满足任务 3 的执行，也满足任务 3 的 deadline。所以，在 time200 的时候所有的任务都满足了它们的 deadline，那么使用第一 deadline 规则，这任务集是可以调度的。

19.2.4.6 放宽假设

在大多数实时系统中都很难满足我们所说的 RMS 假设，例如，要求一个任务是执行时间是固

定的。通过每个任务的最坏执行情况，我们可以放宽这种假设。
另一个假设是这些任务的独立的。这就是说，任务不需要去等待或竞争资源。这个假设也可以放宽，通过计算一个任务等待资源的时间。类似的，每一个任务的执行时间也要计算 I/O 性能和 RTEMS 的命令调用。
另外，假设并没有考虑中断执行的开销。这些可以通过计算中断服务程序对处理器的使用来考虑进去。类似的，还需要考虑访问本地内存和其它处理器访问本地内存的延迟。
假设认为非周期性的任务仅仅用来初始化和失败-复苏，这个可以通过把所有的周期性任务都放在紧急任务集里来放宽这种假设。可以使用 RMS 来分析和调度这个任务集。所有的非周期性任务都要放在非紧急任务集里。即使是在短暂的超负荷的条件下，紧急任务集里的任务也会被执行。但是非紧急任务集里的任务将不提供这种确保。

19.2.4.7 更多的参考

19.3 操作

19.3.1 创建一个速率单调周期

rtems_rate_monotonic_create命令来创建一个速率单调周期。RTEMS为每个周期分配一个周期控制块PCB来管理新建的速率单调周期。RTEMS还为这个周期分配一个唯一的名。

19.3.2 维护一个周期

rtems_rate_monotonic_period来建立和保持先前已经创建了的速率单调周期的周期性运行。一旦被初始化后，这个周期就一直执行直到周期过期了或被重新初始化。速率单调周期的状态会产生如下几种情况：

- 如果速率单调周期正在运行，调用者任务就会被阻塞。当这个周期运行完，这个周期将会被重新初始化。
- 如果速率单调周期没有正在运行并且没有过期，它会被初始化，调用者任务立即返回。
- 如果在任务调用rtems_rate_monotonic_period之前，速率单调周期已经过期了，周期将被初始化，调用者任务立即返回一个timeout错误状态

19.3.3 获得一个周期的状态

如果命令rtems_rate_monotonic_period (并指定周期为RTEMS_PERIOD_STATUS) 被激活了一段时间，指定的速率单调周期的当前状态将会返回。状态为：

- RTEMS_SUCCESSFUL - period is running　　周期在运行
- RTEMS_TIMEOUT - period has expired　　周期过期
- RTEMS_NOT_DEFINED - period has never been initiated　　周期从未初始化

获取周期的状态不会改变周期的状态和长度。

19.3.4 取消一个周期

通过 rtems_rate_monotonic_cancel 命令可以停止一个周期。被停止的周期可以通过 rtmes_rate_monotonic_period 命令重新初始化。

19.3.5 删除一个速率单调周期

rtems_rate_monotonic_delete 命令来删除一个速率单调周期。如果这个周期正在运行并且没有过期，这个周期将会自动取消。PCB 也会自动返回到 PCB 空闲表中。除了创建它的任务外，其它的任务也可以删除这个周期。

19.3.6 例子

19.3.7 简单的周期任务

这个例子包括一个单一周期的任务，在初始化之后，以每 100 个 tick 为周期执行。

```
rtems_task Periodic_task(rtems_task_argument arg)
{
    rtems_name name;
    rtems_id period;
    rtems_status_code status;
    name = rtems_build_name( 'P', 'E', 'R', 'D' );
    status = rtems_rate_monotonic_create( name, &period );
    if ( status != RTEMS_STATUS_SUCCESSFUL ) {
    printf( "rtems_monotonic_create failed with status of %d.\n", rc );
    exit( 1 );
    }


    while ( 1 ) {
        if ( rtems_rate_monotonic_period( period, 100 ) == RTEMS_TIMEOUT )
        break;
        /* Perform some periodic actions */
    }

    /* missed period so delete period and SELF */
    status = rtems_rate_monotonic_delete( period );
    if ( status != RTEMS_STATUS_SUCCESSFUL ) {
    printf( "rtems_rate_monotonic_delete failed with status of %d.\n", status );
    exit( 1 );
    }
    status = rtems_task_delete( SELF ); /* should not return */
    printf( "rtems_task_delete returned with status of %d.\n", status );
    exit( 1 );
}
```

这个例子创建了一个 RMS 作为它初始化的一部分。当第一次执行循环的时候，命令 rtems_rate_monotonic_period 将会初始化 100ticks 的周期然后立刻返回。接下来调用 rtems_rate_monotonic_period 将会使任务阻塞 100 个 ticks 周期。如果循环执行的时间超过了 100 个 ticks，那么 rtems_rate_monotonic_period 将返回 RTEMS_TIMEOUT 的状态。如果上面的任务超过了它的 deadline，那么 RMS 和任务自己都将被删除。

19.3.8 多周期的任务

这个例子包括了一个单周期的任务，在初始化之后，每 100ticks 表现两个行为。第一个行为是每 100 个 ticks 的第一个 40tick，第二个行为要在 40 到 70 之间的 ticks。最后的 30 个 tick 这个任务不使用。

```
rtems_task Periodic_task(rtems_task_argument arg)
{
rtems_name name_1, name_2;
rtems_id period_1, period_2;
rtems_status_code status;
name_1 = rtems_build_name( 'P', 'E', 'R', '1' );
name_2 = rtems_build_name( 'P', 'E', 'R', '2' );
(void ) rtems_rate_monotonic_create( name_1, &period_1 );
(void ) rtems_rate_monotonic_create( name_2, &period_2 );
while ( 1 ) {
if ( rtems_rate_monotonic_period( period_1, 100 ) == TIMEOUT )
break;
if ( rtems_rate_monotonic_period( period_2, 40 ) == TIMEOUT )
break;
/*
* Perform first set of actions between clock
* ticks 0 and 39 of every 100 ticks.
*/
if ( rtems_rate_monotonic_period( period_2, 30 ) == TIMEOUT )
break;
/*
* Perform second set of actions between clock 40 and 69
* of every 100 ticks. THEN ...
*
* Check to make sure we didn't miss the period_2 period.
*/
if ( rtems_rate_monotonic_period( period_2, STATUS ) == TIMEOUT )
break;
(void) rtems_rate_monotonic_cancel( period_2 );
}
/* missed period so delete period and SELF */
(void ) rtems_rate_monotonic_delete( period_1 );
(void ) rtems_rate_monotonic_delete( period_2 );
(void ) task_delete( SELF );
}
```

上面的这个任务在它的初始化的时候创建了 2 个 RMS。当第一次执行循环的时候命令，rtems_rate_monotonic_period 将会初始化 100ticks 的周期 1 然后立刻返回。接下来调用 rtems_rate_monotonic_period 并指明周期 1 将会使任务阻塞 100 个 ticks 周期。周期 2 的周期用来控制两个动作的的执行时间会在周期 1 创建的时间后 100 个 ticks 完成。命令

rtems_rate_monotonic_cancel 可以确保当任务在周期 1 被阻塞的时候，周期 2 不会过时。如果没有这个 cancel 命令，每次执行 rtems_rate_monotonic_period(period_2, 40)，除了初始化的第一次外，都将返回一个 RTEMS_TIMEOUT 的状态。如果上面的任务超过了它的 deadline，那么 RMS 和任务自己都将被删除。

21 用户扩展管理

21.1 介绍

RTEMS 的用户扩展管理允许开发应用的人员增强程序执行的功能，这是通过允许他们应用一些扩展程序来实现的，这些程序在临界系统事件发生时被调用。用户扩展程序提供的指令包括：

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

21.2 背景

用户扩展程序通常在下述系统事件发生时被调用：

Task creation　建立任务
Task initiation　初始化任务
Task reinitiation　重新初始化任务
Task deletion　删除任务
Task context switch　任务内容转换
Post task context switch　任务内容转换后
Task begin　任务开始
Task exits　退出任务
Fatal error detection　致命错误检测

扩展调用相当于与系统事件相关的功能函数。

21.2.1 扩展集

一个扩展集定义为在临界系统事件发生时被调用的一系列程序。这些程序一起执行一个特殊的功能，比如性能监测或调试支持。下面的结构体用来通知 RTEMS 包含扩展集的入口点：

```
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;
} rtems_extensions_table;
```

RTEMS 允许用户同时有多个扩展集。首先，一个单独的静态的扩展集可以在应用的用户扩展表中定义，该表是设置表的一部分。该扩展集存在于系统的整个生命期而且不能被删除。这个扩展及非常重要，因为它是一个应用在 RTEMS 的初始化指令失败时能够提供致命错误扩展的唯一方式。静态扩展集是可选的，如果不需要可以将它设置为 NULL。

其次，用户可用rtems_extension_create指令建立动态扩展。这些扩展都是RTEMS的对象，它们有名字、ID，而且能被动态的建立和删除。同静态扩展集相比较，这些扩展只有在初始

化指令完全成功之后才能被建立和装入。动态扩展对于扩展集的封装非常有用，例如，应用可以用一个扩展管理一个特殊的协处理器，执行性能监测，进行堆栈越界检查。每个扩展集的安装都是独立于其他扩展集的。

所有的用户扩展都是可选的，而且 RTEMS 对扩展的名字没有限制。用户扩展的入口点被复制到 RTEMS 的内部结构中，这意味着用户不需要在建立扩展表后保持它，也不需要在一个曾经建立已经无效的表中进行动态的句柄入口点转换。建一个函数位置表能为一些空间紧缺的应用节省空间。

如果没安装交换句柄，扩展交换对内容交换是没有影响的。

21.2.2 TCB 扩展区

RTEMS 为每个扩展集提供一个指向用户定义数据区的指针，该指针用于和各个任务控制块（TCB）的连接。这些指针是 TCB 的扩展，而且能存储一些用户扩展功能需要的附加数据。它还可以让用户扩展使用和每个任务相关的记事本，尽管这有可能引起和这些特定的记事本的应用的冲突。

TCB 扩展是 TCB 的一个指针集的排列。指向这个表的索引可以通过下述指令获得：

```
    index = rtems_get_index(extension_id);
```

该区域的指针数和用户扩展集设置的数目是相同的。这就允许应用通过用户定义信息增强 TCB。例如：一个应用可以在 TCB 的扩展内存区执行存储时间统计的任务。当任务内容交换执行时，TASK_SWITCH 扩展能读出实时时钟，并结合任务换入的时间戳来计算任务换出用了多少时间。

需要的话，在任务建立和开始时用TASK_CREATE或TASK_START扩展分配用于TCB的扩展内存区和设置TCB扩展指针。应用负责管理TCB的扩展内存区。内存可能被TASK_RESTART重新初始化而且可能在任务删除时被TASK_DELETE取消分配。由于TCB扩展缓冲区一般来说是大小固定的，RTEMS分区管理器可以用来管理应用的扩展内存区。应用可以建立一个固定大小的TCB扩展缓冲区，并用分区管理器分配和取消分配命令来获得和释放扩展缓冲区。

21.2.3 扩展

下述章节包含各个扩展的描述，每部分都包含调用相应扩展的函数原型。函数名和参数由用户定义，示例的名字尽量接近用户习惯但并不是标准。

21.2.3.1 建立任务扩展

建立任务扩展直接响应rtems_task_create指令。如果这个扩展在任何静态或动态扩展集中被定义了，并且一个任务正在被建立，该扩展程序将自动被RTEMS系统调用。该扩展的程序原型如下：

```
boolean user_task_create(
    rtems_tcb *current_task,
    rtems_tcb *new_task
);
```

current_task 用于当前任务访问TCB，new_task用于新建立的任务访问TCB。这个扩展调用从rtems_task_create指令开始直到new_task完全初始化之后和在new_task置于TCB链之前。返回一个布尔型值。一般一个任务的建立都要申请分配一些资源，如果没申请到，返回FALSE，建立任务的操作也意味着失败。

21.2.3.2 任务开始扩展

任务开始扩展直接响应rtems_task_start指令。如果这个扩展在任何静态或动态扩展集中被定义了，并且一个任务正在开始，该扩展程序将自动被RTEMS系统调用。该扩展的程序原型如下：

```
rtems_extension user_task_start(
```

```
    rtems_tcb *current_task,
    rtems_tcb *started_task
);
```
current_task 用于当前任务访问TCB，started_task用于正在开始的休眠任务访问TCB。这个扩展调用从rtems_task_start指令开始直到started_task已经就绪准备执行之后和在started_task置于TCB链之前。

21.2.3.3 重新开始任务扩展

重新开始任务扩展直接响应task_restart指令。如果这个扩展在任何静态或动态扩展集中被定义了，并且一个任务正在重新开始，该扩展程序将自动被RTEMS系统调用。该扩展的程序原型如下：
```
rtems_extension user_task_restart(
    rtems_tcb *current_task,
    rtems_tcb *restarted_task
);
```
current_task 用于当前任务访问TCB，restarted_task用于重新开始的任务访问TCB。这个扩展调用从task_restart指令开始直到restarted_task已经就绪准备执行之后和在restarted_task置于TCB链之前。

21.2.3.4 删除任务扩展

删除任务扩展直接响应task_delete指令。如果这个扩展在任何静态或动态扩展集中被定义了，并且一个任务正在被删除，该扩展程序将自动被RTEMS系统调用。该扩展的程序原型如下：
```
rtems_extension user_task_delete(
    rtems_tcb *current_task,
    rtems_tcb *deleted_task
);
```
current_task 用于当前任务访问TCB，deleted_task用于被删除的任务访问TCB。这个扩展调用从task_delete指令开始直到TCB被从TCB链中移出之后和在所有资源包括TCB返还给自由场地之前。如果任务删除它自己，这个扩展将不能调用任何RTEMS指令。

21.2.3.5 任务转换扩展

任务转换扩展响应任务内容的转换。如果这个扩展在任何静态或动态扩展集中被定义了，并且一个任务内容转换正在进行，该扩展程序将自动被RTEMS系统调用。该扩展的程序原型如下：
```
rtems_extension user_task_switch(
    rtems_tcb *current_task,
    rtems_tcb *heir_task
);
```
current_task 用于被转换出的任务访问TCB，heir_task用于被转换入的任务访问TCB。这个扩展调用从RTEMS转换指令开始直到current_task内容已经被保存和在heir_task内容保存之前。这个扩展不能调用任何RTEMS指令。

21.2.3.6 任务执行扩展

任务执行扩展在一个任务开始执行时被调用。它在程序体开始前就启动，并在任务执行过程中一直执行。该扩展的程序原型如下：
```
rtems_extension user_task_begin(
    rtems_tcb *current_task
```

);
current_task用于当前执行的已经开始的任务访问TCB。任务执行（Begin）与任务开始（Start）的区别在于：执行是在任务的整个运行过程中，开始是在任务的运行前。对于大多数扩展，这不是严格区分的。

21.2.3.7 任务退出扩展

任务退出扩展在一个运行的任务由于潜在的或直接的返回状态退出本体时被调用。

23 多处理器控制

23.1 介绍

在多处理器实时操作系统中，新的需求，例如在不同的处理器之间共享数据和全局资源被引入了。这种需求需要一个功能强大的通信机制，允许不同的处理器在需要的时候可以互相通信。另外，可以想象不同处理器的分支影响了整个实时操作系统，并且使得整个系统变得异常复杂。

RTEMS 提供了简单或者复杂的实时多处理性能。执行的时候可以和目标硬件或者紧密结合，或者松散的结合。同时，RTEMS 支持目标处理器和目标板或者是同类的处理器或者是异类的处理器的混合。

RTEMS 的一个重要的设计特性就是超越目标硬件的局限。这个目标可以通过让不同的节点之间的处理器之间看起来的透明的，从而给应用软件一个逻辑上的全局影象。RTEMS 要求用户将一些对象定义为全局对象的例如任务，队列，事件，信号，信号量和内存区块。这样这些全局对象就可以被任何任务访问了，而不管这些任务的实际物理地址。简单的说，RTEMS 可以让用户把全部软件和硬件看成是一个整体。

21 用户扩展管理

21．1 介绍

RTEMS 的用户扩展管理允许开发应用的人员增强程序执行的功能，这是通过允许他们应用一些扩展程序来实现的，这些程序在临界系统事件发生时被调用。用户扩展程序提供的指令包括：

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

21．2 背景

用户扩展程序通常在下述系统事件发生时被调用：

Task creation 建立任务
Task initiation 初始化任务
Task reinitiation 重新初始化任务
Task deletion 删除任务
Task context switch 任务内容转换
Post task context switch 任务内容转换后
Task begin 任务开始

Task exits 退出任务
Fatal error detection 致命错误检测
扩展调用相当于与系统事件相关的功能函数。

21．2．1 扩展集

一个扩展集定义为在临界系统事件发生时被调用的一系列程序。这些程序一起执行一个特殊的功能，比如性能监测或调试支持。下面的结构体用来通知 RTEMS 包含扩展集的入口点：

```
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;
} rtems_extensions_table;
```

RTEMS 允许用户同时有多个扩展集。首先，一个单独的静态的扩展集可以在应用的用户扩展表中定义，该表是设置表的一部分。该扩展集存在于系统的整个生命期而且不能被删除。这个扩展及非常重要，因为它是一个应用在 RTEMS 的初始化指令失败时能够提供致命错误扩展的唯一方式。静态扩展集是可选的，如果不需要可以将它设置为 NULL。

其次，用户可用rtems_extension_create指令建立动态扩展。这些扩展都是RTEMS的对象，它们有名字、ID，而且能被动态的建立和删除。同静态扩展集相比较，这些扩展只有在初始化指令完全成功之后才能被建立和装入。动态扩展对于扩展集的封装非常有用，例如，应用可以用一个扩展管理一个特殊的协处理器，执行性能监测，进行堆栈越界检查。每个扩展集的安装都是独立于其他扩展集的。

所有的用户扩展都是可选的，而且 RTEMS 对扩展的名字没有限制。用户扩展的入口点被复制到 RTEMS 的内部结构中，这意味着用户不需要在建立扩展表后保持它，也不需要在一个曾经建立已经无效的表中进行动态的句柄入口点转换。建一个函数位置表能为一些空间紧缺的应用节省空间。

如果没安装交换句柄，扩展交换对内容交换是没有影响的。

21．2．2 TCB 扩展区

RTEMS 为每个扩展集提供一个指向用户定义数据区的指针，该指针用于和各个任务控制块（TCB）的连接。这些指针是 TCB 的扩展，而且能存储一些用户扩展功能需要的附加数据。它还可以让用户扩展使用和每个任务相关的记事本，尽管这有可能引起和这些特定的记事本的应用的冲突。

TCB 扩展是 TCB 的一个指针集的排列。指向这个表的索引可以通过下述指令获得：

```
    index = rtems_get_index(extension_id);
```

该区域的指针数和用户扩展集设置的数目是相同的。这就允许应用通过用户定义信息增强 TCB。例如：一个应用可以在 TCB 的扩展内存区执行存储时间统计的任务。当任务内容交换执行时，TASK_SWITCH 扩展能读出实时时钟，并结合任务换入的时间戳来计算任务换出用了多少时间。

需要的话，在任务建立和开始时用TASK_CREATE或TASK_START扩展分配用于TCB的扩展内存区和设置TCB扩展指针。应用负责管理TCB的扩展内存区。内存可能被TASK_RESTART重新初始化而且可能在任务删除时被TASK_DELETE取消分配。由于TCB

扩展缓冲区一般来说是大小固定的，RTEMS分区管理器可以用来管理应用的扩展内存区。应用可以建立一个固定大小的TCB扩展缓冲区，并用分区管理器分配和取消分配命令来获得和释放扩展缓冲区。

21．2．3 扩展

下述章节包含各个扩展的描述，每部分都包含调用相应扩展的函数原型。函数名和参数由用户定义，示例的名字尽量接近用户习惯但并不是标准。

21．2．3．1 建立任务扩展

建立任务扩展直接响应rtems_task_create指令。如果这个扩展在任何静态或动态扩展集中被定义了，并且一个任务正在被建立，该扩展程序将自动被RTEMS系统调用。该扩展的程序原型如下：

```
boolean user_task_create(
    rtems_tcb *current_task,
    rtems_tcb *new_task
);
```

current_task 用于当前任务访问TCB，new_task用于新建立的任务访问TCB。这个扩展调用从rtems_task_create指令开始直到new_task完全初始化之后和在new_task置于TCB链之前。

返回一个布尔型值。一般一个任务的建立都要申请分配一些资源，如果没申请到，返回FALSE，建立任务的操作也意味着失败。

21．2．3．2 任务开始扩展

任务开始扩展直接响应rtems_task_start指令。如果这个扩展在任何静态或动态扩展集中被定义了，并且一个任务正在开始，该扩展程序将自动被RTEMS系统调用。该扩展的程序原型如下：

```
rtems_extension user_task_start(
    rtems_tcb *current_task,
    rtems_tcb *started_task
);
```

current_task 用于当前任务访问TCB，started_task用于正在开始的休眠任务访问TCB。这个扩展调用从rtems_task_start指令开始直到started_task已经就绪准备执行之后和在started_task置于TCB链之前。

21．2．3．3 重新开始任务扩展

重新开始任务扩展直接响应task_restart指令。如果这个扩展在任何静态或动态扩展集中被定义了，并且一个任务正在重新开始，该扩展程序将自动被RTEMS系统调用。该扩展的程序原型如下：

```
rtems_extension user_task_restart(
    rtems_tcb *current_task,
    rtems_tcb *restarted_task
);
```

current_task 用于当前任务访问TCB，restarted_task用于重新开始的任务访问TCB。这个扩展调用从task_restart指令开始直到restarted_task已经就绪准备执行之后和在restarted_task置于TCB链之前。

21．2．3．4 删除任务扩展

删除任务扩展直接响应task_delete指令。如果这个扩展在任何静态或动态扩展集中被定义了，并且一个任务正在被删除，该扩展程序将自动被RTEMS系统调用。该扩展的程序原型如下：

```
rtems_extension user_task_delete(
```

```
    rtems_tcb *current_task,
    rtems_tcb *deleted_task
);
```

current_task 用于当前任务访问TCB，deleted_task用于被删除的任务访问TCB。这个扩展调用从task_delete指令开始直到TCB被从TCB链中移出之后和在所有资源包括TCB返还给自由场地之前。如果任务删除它自己，这个扩展将不能调用任何RTEMS指令。

21．2．3．5 任务转换扩展

任务转换扩展响应任务内容的转换。如果这个扩展在任何静态或动态扩展集中被定义了，并且一个任务内容转换正在进行，该扩展程序将自动被RTEMS系统调用。该扩展的程序原型如下：

```
rtems_extension user_task_switch(
    rtems_tcb *current_task,
    rtems_tcb *heir_task
);
```

current_task 用于被转换出的任务访问TCB，heir_task用于被转换入的任务访问TCB。这个扩展调用从RTEMS转换指令开始直到current_task内容已经被保存和在heir_task内容保存之前。这个扩展不能调用任何RTEMS指令。

21．2．3．6 任务执行扩展

任务执行扩展在一个任务开始执行时被调用。它在程序体开始前就启动，并在任务执行过程中一直执行。该扩展的程序原型如下：

```
rtems_extension user_task_begin(
    rtems_tcb *current_task
);
```

current_task用于当前执行的已经开始的任务访问TCB。任务执行（Begin）与任务开始（Start）的区别在于：执行是在任务的整个运行过程中，开始是在任务的运行前。对于大多数扩展，这不是严格区分的。

21．2．3．7 任务退出扩展

任务退出扩展在一个运行的任务由于潜在的或直接的返回状态退出本体时被调用。格式如下：

```
rtems_extension user_task_exitted(
    rtems_tcb *current_task
);
```

current_task用于正在退出的执行任务访问TCB。

尽管任务退出也被认为是致命错误，但是该扩展允许通过重启或删除的方式修复该错误。如果用户不希望修复，致命错误才被报告。如果用户不提供这个扩展，系统将使用缺省的扩展来处理。缺省的句柄调用具有RTEMS_TASK_EXITTED指令状态的fatal_error_occurred指令。

21．2．3．8 致命错误扩展

致命错误扩展与fatal_error_occurred指令相关。如果这个扩展在任何静态或动态扩展集中被定义了，并且fatal_error_occurred指令已经被激活，该扩展程序将自动被RTEMS系统调用。该扩展的程序原型如下：

```
rtems_extension user_fatal_error(
    Internal_errors_Source the_source,
    rtems_boolean is_internal,
    rtems_unsigned32 the_error
);
```

the_error是传递给fatal_error_occurred指令的错误代码，该扩展由fatal_error_occurred指令激

活。
如果定义了该扩展，它就在RTEMS的缺省致命错误激活和处理器停止之前被激活。例如，该扩展可以在致命错误发生时传递控制给调试器。该扩展不能调用任何RTEMS指令。

21．2．4 调用顺序

当一个临界系统事件发生时，用户扩展调用有“向前”和“向后”两种顺序方式。向前指的是静态扩展跟着动态扩展集就像建立时的顺序一样。向后指的是动态扩展集的执行顺序是反向的，动态扩展在静态扩展之后建立。通过这种方式建立的扩展集，扩展是依赖其他扩展建立的。系统事件发生后，向前顺序的扩展包括：
_ Task creation
_ Task initiation
_ Task reinitiation
_ Task deletion
_ Task context switch
_ Post task context switch
_ Task begins to execute
系统事件发生后，向后顺序的扩展包括：
_ Task deletion
_ Fatal error detection
在这些事件中，扩展集的执行是反向的以确保扩展及按正确的依赖关系执行。例如，通过调用静态扩展集last，就知道“系统”的致命错误扩展将是最后的致命错误扩展在执行。

21．3 操作

21．3．1 建立一个扩展集

通过分配一个扩展集控制块（ESCB），rtems_extension_create指令可以创建和安装一个新的扩展集，分配扩展及一个用户定义的名字，分配一个扩展集的ID。新的扩展集立即安装并被下一个系统甚至是一个支持的扩展集调用。

21．3．2 获得扩展集ID

当一个扩展集被创建后，RTEMS系统就给它分配一个唯一的ID直到它被删除。可以通过两种方法来获得这个ID。一是可以通过rtems_extension_create命令，扩展集的ID存储在用户提供的位置。二是可以通过rtems_extension_ident命令来获取它的ID。这个ID可以被其它的命令用于对该扩展集的操作。

21．3．3 删除一个扩展集

通过 rtems_extension_delete 命令可以删除一个扩展集，扩展集控制块也同时返回给 ESCB 空闲表中。扩展及甚至可以被不是该扩展集的创建者删除。任何后来的对该扩展集的名字和 ID 的引用都是非法的。

23 多处理器控制

23.1 介绍

在多处理器实时操作系统中，新的需求，例如在不同的处理器之间共享数据和全局资源被引入了。这种需求需要一个功能强大的通信机制，允许不同的处理器在需要的时候可以互相通信。另外，可以想象不同处理器的分支影响了整个实时操作系统，并且使得整个系统变得异

常复杂。
RTEMS 提供了简单或者复杂的实时多处理性能。执行的时候可以和目标硬件或者紧密结合，或者松散的结合。同时，RTEMS 支持目标处理器和目标板或者是同类的处理器或者是异类的处理器的混合。
RTEMS 的一个重要的设计特性就是超越目标硬件的局限。这个目标可以通过让不同的节点之间的处理器之间看起来的透明的，从而给应用软件一个逻辑上的全局影象。RTEMS 要求用户将一些对象定义为全局对象的例如任务，队列，事件，信号，信号量和内存区块。这样这些全局对象就可以被任何任务访问了，而不管这些任务的实际物理地址。简单的说，RTEMS 可以让用户把全部软件和硬件看成是一个整体。