Y 953914

双内核实时操作系统的设计和实现

The Design and Implementation of a dual-kernel real-time operating system

院　　系：	计算机科学与工程
专　　业：	计算机系统结构
攻 读 学 位：	理学硕士
作　　者：	王轶凡
指 导 教 师：	涂时亮　教授
指导小组成员：	涂时亮　教授
	陈章龙　教授

图片目录

图 1：FRTOS 的体系结构.....7
图 2：FRTOS 应用系统的内存布局.....8
图 3：FRTOS 的任务切换.....9
图 4：任务状态转换图.....11
图 5：基于优先级的可剥夺的调度.....12
图 6：FRTOS 的多级队列.....12
图 7：时间片调度.....14
图 8：共享内存的通信方法.....16
图 9：获取一个互斥量.....18
图 10：释放一个互斥量.....19
图 11：优先级反转.....21
图 12：优先级继承.....21
图 13：先进先出队列和就绪队列.....23
图 14：邮箱的使用.....25
图 15：使用邮箱进行双向通信.....26
图 16：用邮箱实现客户机-服务器模.....26
图 17：信号的激发和信号处理函数.....27
图 18：中断向量表（传统）.....28
图 19：中断服务程序的运行（传统）.....28
图 20：控制内核代码与 ISR 的并发数据访问（传统）.....29
图 21：中断向量表（FRTOS）.....30
图 22：中断服务程序和可延迟中断服务程序的执行（FRTOS）.....30
图 23：控制内核代码与 ISR/DSR 的并发数据访问（FRTOS）.....31
图 24：抽象文件体统.....33
图 25：设备驱动程序.....34
图 26：FRTOS2 的体系结构.....36

表目录

表 1：典型的嵌入式系统应用 4
表 2：任务控制块的主要成员 10
表 3：启用和禁用剥夺的 API 15
表 4：任务控制相关的 API 15
表 5：禁用中断相关的 API 17
表 6：与互斥量相关的 API 19
表 7：与信号量相关的 API 24
表 8：与邮箱相关的 API 25
表 9：与信号机制相关的 API 27
表 10：一些典型的编译时期宏选项 35

摘要

本文讲述了一个双内核的实时操作系统 FRTOS2 的设计与实现，文章主要包括两部分。第一部分讲述了 FRTOS（一个实时操作系统）的体系结构、主要功能和相应实现，包括：任务管理、互斥量、信号量、邮箱、抽象文件系统、中断处理等等。第二部分讲述了如何把 FRTOS 变成一个双内核的实时操作系统:FRTOS2。这是通过模拟中断控制器，使 uCLinux 作为优先级最低的一个任务运行在 FRTOS 上来实现的。

关键字：实时 操作系统 双内核

中图分类号：TP316.2

Abstract

This thesis introduces the design and implementation of a dual-kernel real-time operating system: FRTOS2. It includes two parts. The first part describes the architecture, major features and implementation of FRTOS(a real-time operating system), including task management, mutex, semaphore, mailbox, abstract file system, interrupt handling and so on. The second part describes how to improve FRTOS to a dual-kernel real-time operating system (FRTOS2). This is achieved by simulating the interrupt controller and porting uCLinux kernel on FRTOS (the uCLinux kernel running as a task with lowest priority).

Key words: real-time, operating system, dual-kernel

1 简介

1.1 实时系统的基本概念

在实时系统（real-time system）中所有操作的执行时间必须小于它允许的最大延时，既要在他的最后期限（deadline）之前完成。实时系统通常分为硬实时系统和软实时系统。

一个硬实时系统的正确性不仅依赖于逻辑上的正确性，而且依赖于产生这个结果所需要的时间。如果任务超过了系统的最后极限，即使结果正确，也会有产非常严重的后果。一个典型的硬实时系统的例子就是汽车的刹车反死锁系统（anti-lock brakes system）。当司机踩下刹车时，反死锁系统必须在最后期限（汽车轮胎死锁）前完成，否则就会有车毁人亡的危险。

一个软实时系统的正确性也是不仅依赖于逻辑上的正确性，而且依赖于产生这个结果所需要的时间。不同点是，如果任务超过了最后期限，结果的有用性会大大降低，但不会产生严重的后果。一个典型的软实时系统的例子就是 IP 网络的路由器。在负载很重的情况下，路由器可以随意的丢弃一些数据包。

实时系统不一定要非常的快，因为即使慢，也可以做到实时（只要满足系统的时间限制就可以）。与此相反，非实时系统没有任何时间约束。虽然这些系统可能需要，甚至必须运行的很快，但是不存在任何最后期限。

1.2 嵌入式系统的基本概念

嵌入式系统是一个计算机控制系统，其中计算机完全被封装在它所控制的设备中，用户完全意识不到计算机的存在，下面是一些典型的嵌入式系统应用的例子：

汽车类	发动机控制、刹车反死锁
通讯类	交换机、路由器
办公自动化	传真机、复印机
计算机外设	打印机、扫描仪、调制解调器
民用消费品	微波炉、洗碗机、洗衣机

表 1：典型的嵌入式系统应用

实际上嵌入式系统与实时系统没有必然的联系：嵌入式系统可以是非实时的（比如手机、PDA、游戏机）；实时系统也不一定都是嵌入式的（比如很多的实时工业控制系统使用的是 PC）。

1.3 嵌入式实时操作系统的概念

嵌入式实时操作系统（在后文中将被简称为实时操作系统）是一个控制计算机化的设备的软件系统，它与传统意义上的通用操作系统既有相似之处，也有很多不同之处。

相似之处在于，它们都有类似于进程、进程间通信、设备驱动程序等概念，实现理论也基本相同。

不同之处在于，实时操作系统通常都是运行在嵌入式系统中，它们没有足够的计算资源。此外，实时操作系统一定要保证系统的实时响应速度。这两个不同之处影响着实时操作系统的设计与实现的各个方面，从而使它们看起来与通用操作系统很不一样。

1.4 文章结构

文章主要包括两部分。第一部分讲述了 FRTOS（一个实时操作系统）的体系结构、主要功能和相应实现，包括：任务管理、互斥量、信号量、邮箱、抽象文

件系统、中断处理等等。第二部分，讲述了如何把 FRTOS 变成一个双内核的实时操作系统:FRTOS2。这是通过模拟中断控制器，使 uCLinux 作为优先级最低的一个任务运行在 FRTOS 上来实现的。

2 FRTOS 的设计和实现

2.1 系统简介

FRTOS 像大多数的现代实时操作系统一样是基于两个基本的概念：多任务（multitasking）和任务间通信（intertask communication）。在多任务环境中，一个实时应用可以被分解成多个独立的任务，每个任务可以有自己的执行上下文和系统资源。多个任务间通过“任务间通信”进行同步和交换数据，从而协调它们的执行。FRTOS 任务间通信包括互斥量（mutex）、信号量（semaphore）、信号处理机制（signal）、邮箱（mailbox）。

另一个在实时操作系统中非常重要的概念是中断处理，它使系统得以响应外界事件。在 FRTOS 中，中断服务程序（interrupt service routine）运行在一个单独的上下文中，不会与普通的任务相互打扰。为了达到快速响应中断的目的，FRTOS 将中断服务程序分为了两部分：中断服务程序（interrupt service routine）（非常紧迫的部分）和可延迟中断服务程序（deferred interrupt service routine）（不是非常急迫的部分）。中断服务程序会在中断发生时立即执行，用于读取设备寄存器中的数据或执行一些时间非常紧要的操作；可延迟中断服务程序会在一个稍后合适的时间执行。这样可以极大的缩短系统的中断延迟（interrupt latency）。

此外，FRTOS 还提供了对设备驱动程序的接口，为 FRTOS 的新设备撰写驱动程序是非常方便的。

最后，通过 FRTOS 的虚拟文件系统可以把各种各样的文件系统挂接到内核上；它的 IO 系统为所有的设备、文件、套接字提外供统一方便的接口。FRTOS 的体系结构如图 1 所示：

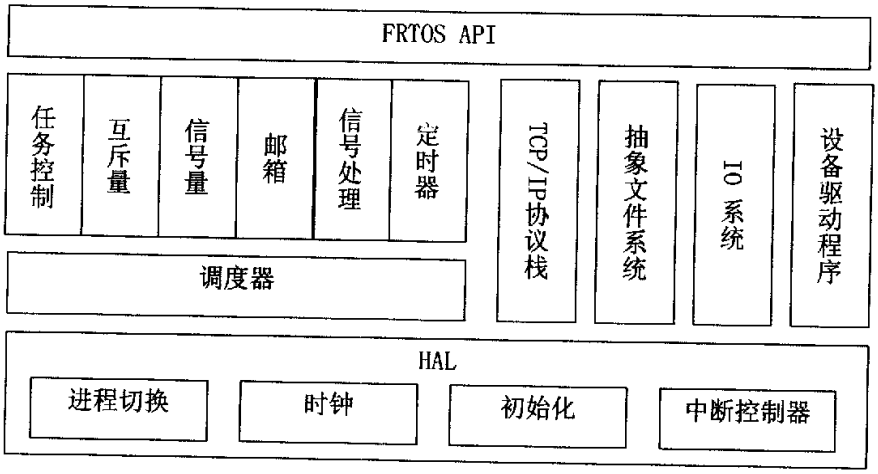


图 1：FRTOS 的体系结构

2.2 FRTOS 的任务

在编写一个大的实时应用时，将应用分解成一个个独立而又相互交互的执行单元是非常重要的。这个执行单元在 FRTOS 中可以被建模为一个任务。在 FRTOS 中，任务和任务之间共享所有的资源，但每个任务又都有自己的执行上下文。

2.2.1 任务的地址空间

由于 FRTOS 的目标平台是没有内存管理单元(MMU: memory management unit)的处理器，所以每个任务没有自己独立的地址空间，它们都和内核共享一个单独的平面地址空间。这是 FRTOS 的任务与通用操作系统中的进程最大的不同之处。

在通用操作系统中，由于有内存管理单元的存在，每个进程会有自己独立的地址空间。而在 FRTOS 中，用户的应用程序是与 FRTOS 的内核编译（compile）、链接（link）在一起的，它们会作为一个映像文件（image）被加载到内存中。一个典型的 FRTOS 应用系统在内存中的布局如图 2 所示：

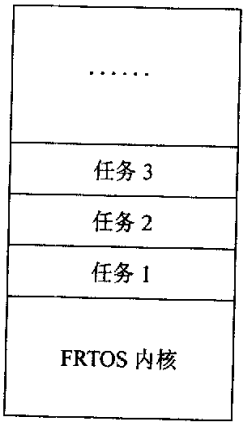


图 2：FRTOS 应用系统的内存布局

如果所有任务都和内核共享一个地址空间，就会带来以下两个问题：

（1）一个任务因为一个严重错误而崩溃（crash），将会导致整个系统的崩溃。

（2）任何一个任务都可以访问其他任务的数据。

对于一个通用操作系统来说，这会是非常严重的系统缺陷，但对于嵌入式实时应用，这些都不重要。因为在嵌入式实时系统中，每一个任务的功能都是预先设计好的，任何一个任务的崩溃都会导致整个系统的失效，所以并不需要错误隔离的机制。此外，在嵌入式实时系统中，所有的任务都是一起被设计和实现的，数据保护是没有必要的。相反，共享一个地址空间，会使任务间交换数据非常的方便。

2.2.2 任务的上下文（context）

如前文所述，每个任务有一个单独的执行上下文，这个上下文提供了任务执行所需要的所有信息。在一个实时应用运行中，CPU 在多个任务之间不停的切换，使他们的运行相互交叉，造成了有多个执行线索的假象，但是实际上每一时刻只能有一个任务在 CPU 上运行。这种任务的切换，对程序员完全透明的：FRTOS 会

在每次任务切换的时候，将前一个任务的执行上下文存储起来，然后读取下一个任务的上下文，这样一个任务不会感觉到自己的执行被其他的任务或者操作系统内核打断过。

一个任务的上下文，主要包含以下几项内容：

CPU 所有寄存器的值（任务的执行线索存贮在 PC 寄存器中）

一个用于局部变量和函数调用的堆栈

标准输入、标准输出、标准错误的重订向

一个延时计数器

一个分片（slice）计数器

内核所需要的控制信息

信号处理的控制信息

……

当一个任务需要被剥夺执行时，所有 CPU 的寄存器将会被保存在这个任务的堆栈上；当一个任务恢复执行时，这些在栈中被保存的寄存器的值将会被恢复。如前所述，每个任务有自己的栈，它会在任务创建的同时被创建。如图 3 所示：

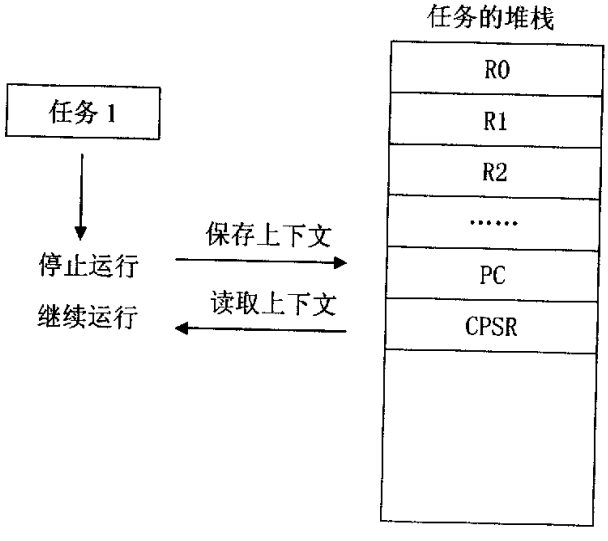


图 3：FRTOS 的任务切换

而其它的运行上下文信息，都存贮在任务控制块（TCB）中，一个任务控制块的主要成员变量如表 2 所示：

名称	说明
栈指针	指向当前任务的堆栈
指向前一个 TCB 的指针	用于实现 TCB 的链表
指向后一个 TCB 的指针	用于实现 TCB 的链表
延时定时器	用于记录任务的延时状态
时间片定时器	用于记录任务的时间片
信号机制控制结构	用于实现信号机制
实际优先级	在优先级继承中，保存任务实际的优先级
当前优先级	任务的当前优先级
队列指针	指向任务当前所在队列的指针，可能是就绪队列，也可能是某个等待队列
任务状态	任务当前的状态
调度器锁的计数器	用于实现调度器加锁的机制
打开文件列表	指向这个任务所有打开的文件

表 2：任务控制块的主要成员

2.2.3 任务的状态转换

FRTOS 维护着每个任务的状态，如前所述，这一信息也同样保存在任务控制块中。一些系统调用或一些系统事件导致任务从一个状态转换到另一个状态。当任务被创建时，它处于就绪（ready）状态；当任务等待一些系统事件时，会处于阻塞（block）状态；当一个任务在 CPU 上运行时，它会处于运行（running）状态。任务的状态转换如图 4 所示：

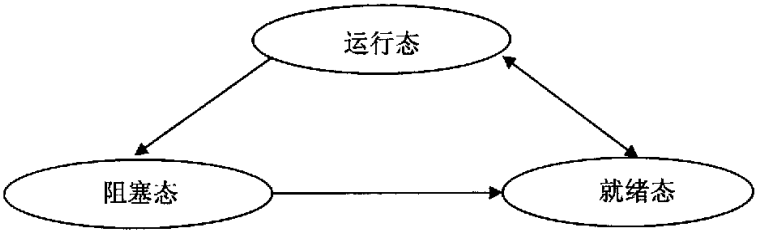


图 4：任务状态转换图

2.2.4 任务的调度

在多任务的环境中，需要一种算法来决定把 CPU 分配给哪一个处于就绪态的任务。FRTOS 的调度算法是基于优先级(priority-based)的可剥夺(preemptive)的调度。FRTOS 支持 256 个不同的优先级，允许多个进程有相同的优先级。程序员可以指定在多个相同优先级的任务之间，是使用先进先出(First-In-First-Out)的调度方法还是时间片轮转(time-slice round-robin)的调度方法。

如前所示，在实时应用中可预测性是非常重要的，而使用时间片轮转，会严重降低任务执行的可预测性，所以 FRTOS 在同优先级的任务中，缺省使用先到先服务的策略。此外，FRTOS 的优先级是动态的，用户可以使用系统调用在系统运行过程中任意修改任务的优先级。

2.2.4.1 基于优先级的可剥夺的调度

所谓“可剥夺”的调度是指只要有一个更高优先级的任务处于就绪状态，就中断当前正在执行的任务，把它的执行上下文保存起来，然后读取新任务的上下文，并把 CPU 分配给它。即 CPU 总是保证优先级最高的就绪任务得到运行。例如在图 5 中：

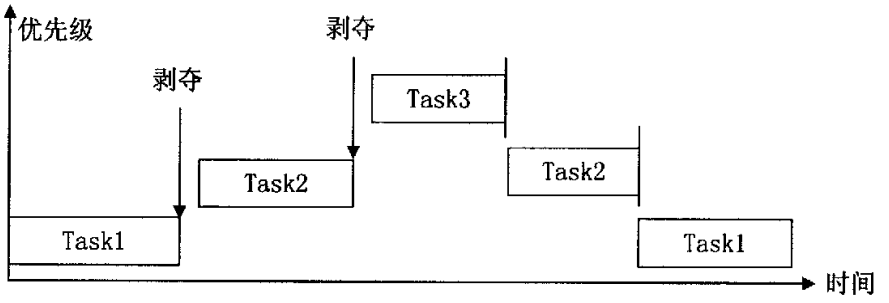


图 5：基于优先级的可剥夺的调度

一开始 Task1 正在运行，但当一个优先级更高的任务 Task2 就绪时，Task1 将立即被剥夺，CPU 被 Task2 抢占；同理，当优相继更高的任务 Task3 就绪时，Task2 将立即被剥夺。当 Task3 运行结束后，由于 Task2 是系统中优先级最高的任务，所以它将获得 CPU；同理，当 Task2 运行结束后，Task1 将获得 CPU。

为了实现这种调度方法，FRTOS 采用了一个多级队列的数据结构来实现就绪队列。如图 6 所示：

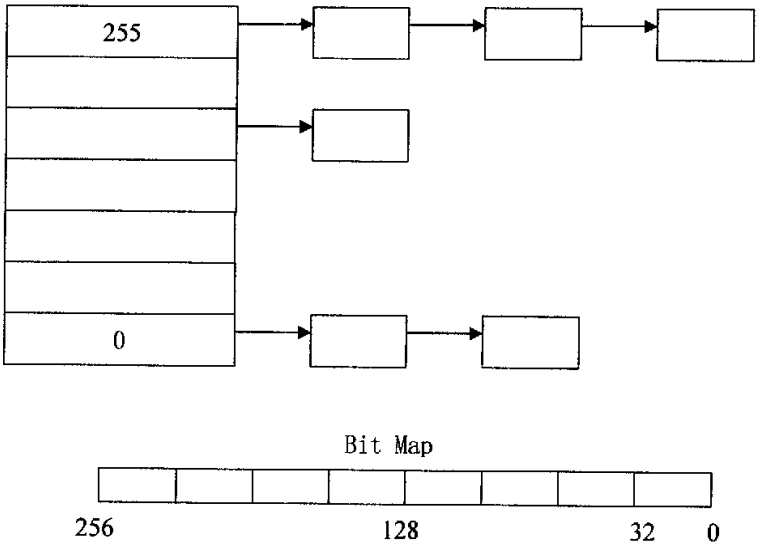


图 6：FRTOS 的多级队列

所有就绪任务的 TCB 都会被保存在就绪队列中。FRTOS 最多支持 256 个优先级（支持多少个优先级可以在编译时设定，但最大值是 256），所以这个多级队列最多有 256 个队列。当一个任务进入就绪状态时，内核会根据它的优先级将它的 TCB 链接到相应的队列中。当一个任务离开就绪状态时，内核会将它的 TCB 从这个多级队列中删除。如果一个就绪任务的优先级发生了变化，内核也会在这个多级队列中调整这个任务的 TCB。

每当内核做调度决策时，它都要从就绪队列中挑出一个优先级最高的任务来运行。内核会维护一个长度为 256 的位图，每一位用于表示有没有就绪任务处于这一优先级。使用这个结构再结合多级队列，内核就可以很快地挑出优先级最高的就绪任务。

2.2.4.2 时间片轮转调度算法

如果多个任务处于相同的优先级，缺省的先入先出算法有一些潜在的问题。如果一个任务要占用 CPU 很长时间，其他处于同一优先级的任务将没有任何机会得到运行。

为了解决这个问题，可以使用时间片轮转的算法。所谓时间片轮转算法是使多个优先级相等的就绪任务公平的共享 CPU。每个优先级相等的任务都轮流运行一个小的时间间隔，这个时间间隔被称为时间片。通过一个编译时期指定的宏可以决定是否启用时间片轮转，以及时间片的大小。

一个时间片是指一个任务在这一周期中所允许运行的时间；在此之后它要将 CPU 让给就绪队列中优先级相等的任务。每个任务执行一段相同的时间间隔，CPU 在这些任务之间不停的轮转。任何一个任务要获得第二次执行机会，要等所有同优先级的任务执行过一次。

如果启用时间片调度，FRTOS 会在每个时钟中断（Tick）发生的时候，减少这个任务的时间片的大小。当这个数值减为 0 时，任务的时间片会被更新为初始时间片大小，并被放到就绪队列中同一优先级的尾部。一个刚刚创建的任务也会被放到同一优先级的就绪队列的尾部，同样，任务的时间片会被更新为初始时间

片的大小。如果一个任务在它运行的时间片中被阻塞或被剥夺，它的时间片的大小将会被保存在 TCB 中，并在它恢复为就绪状态时继续使用。

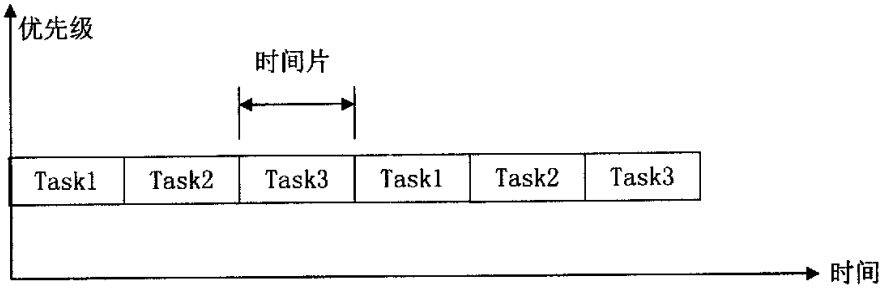


图 7：时间片调度

2.2.4.3 启用和禁用剥夺

在 FRTOS 中，用户可以在任何一个任务中显式的启用和禁用 FRTOS 剥夺的功能。如果一个任务禁用了操作系统剥夺的功能，那么它不会被任何高优先级的任务剥夺。但是，如果这个任务被阻塞或被挂起（suspend），FRTOS 会在就绪队列中选取一个优先级最高的任务继续执行。此时剥夺是否启用，取决于被选中的任务的设置。当原先的任务恢复运行后，剥夺将被重新禁用。

有一点非常重要，禁用剥夺只是阻止任务切换，对中断服务程序（ISR）和可延迟中断服务程序(DSR)没有任何影响，即任务还是可能被中断服务程序(ISR)和可延迟中断服务程序（DSR）剥夺。启用和禁用剥夺可以达到互斥的目的，但为了达到系统实时性的要求，要确保禁用剥夺的时间尽量短。

FRTOS 在每个任务的 TCB 中，添加了一个成员变量用于表示调度器是否被加锁，它的初始值为 0。每当用户调用 scheduler_lock 时，它的值即被增 1；当用户调用 scheduler_unlock 时，它的值则被减 1。如果这个成员变量的值为 0，表示启用剥夺，否则表示禁用剥夺。

系统调用名称	说明
scheduler_lock	启用剥夺
scheduler_unlock	禁用剥夺

表 3：启用和禁用剥夺的 API

2.3 任务控制

FRTOS 提供任务控制相关的 API，可以用于创建任务、挂起/恢复任务、延时等功能，如表 4 所示：

系统调用名称	说明
task_create	创建一个新任务，需要提供这个任务的入口函数指针、优先级和堆栈大小
task_suspend	将一个指定的任务挂起
task_resume	恢复一个被挂起的任务
task_set_pri	为一个任务指定优先级
task_delay	将一个任务延迟若干个 Tick

表 4：任务控制相关的 API

2.4 任务间通信

FRTOS 提供了很多功能强大的任务间通信的手段，它们包括：

共享内存，用于简单的共享数据

互斥量（Mutex），用于任务间的互斥

信号量（Semaphore），用于任务的同步和互斥

邮箱（Message queue），用于任务间的交换数据

信号机制（Signal），用于处理异常和异步事件

2.4.1 共享数据结构

任务之间最简单的通信方式就是访问共享的数据结构。FRTOS 所有的任务共享一个线性的地址空间，在它们之间共享数据结构是非常容易的。全局变量、线性缓冲区、环形缓冲区、链表、指针等都可以在不同的任务上下文中直接引用。

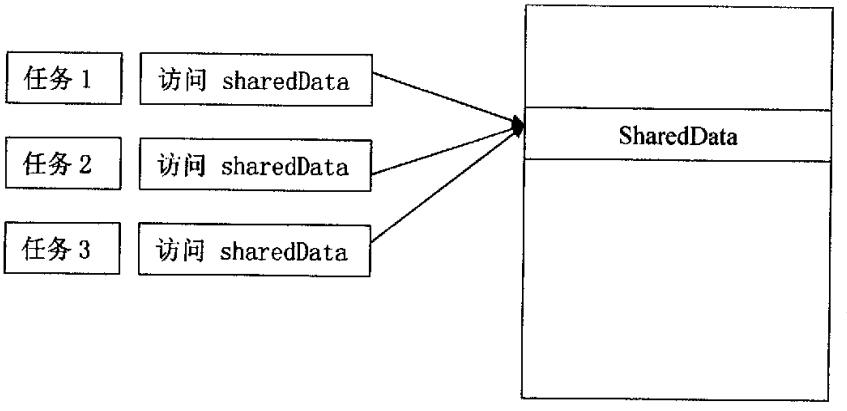


图 8：共享内存的通信方法

2.4.2 互斥

虽然一个共享的存储空间简化了数据交换，但是需要一个加锁的方法来防止竞争（contention）。有很多方法可以达到独占一个资源的目的。如：禁用中断、禁用剥夺和使用互斥量对资源进行加锁。

2.4.2.1 禁用中断

禁用中断是达到互斥最强大的手段。在 FRTOS 中，由于采用了可延迟中断服务程序，所谓禁用中断实质上是指禁用可延迟中断服务程序 （DSR） 。使用这种加锁手段可以确保当前的任务独占 CPU。

虽然这种方法解决了任务与 DSR 互斥的问题，但对于大多数的实时系统而言，尽量不要把禁用中断作为一种通用的实现互斥的手段。因为在这段时间内，系统无法响应外部事件。如果一个实时应用需要快速响应外部事件，那么这种中断延迟是不可以接受的。尽管如此，如果需要和 DSR 达到互斥（即任务要和 DSR 访问同一个数据结构），禁用中断是唯一的方法。在这种情况下，要确保中断被禁用的时间足够短。

系统调用名称	说明
dsr_lock	禁止 DSR 运行
dsr_unlock	允许 DSR 运行

表 5：禁用中断相关的 API

2.4.2.2 禁用剥夺

禁用剥夺提供了一种相比之下更为宽松的互斥方式。没有任何一个任务可以剥夺当前的任务，但是 DSR/ISR 是可以运行的。

这种方法也会影响系统的实时响应。即使一个高优先级的任务不会访问临界区，在禁用剥夺的任务离开临界区之前，它也不会得到运行。所以在使用时，要确保禁用剥夺的时间足够短。

2.4.3 互斥量

FRTOS 的互斥量是经过高度优化的，非常快速的一种任务间通讯手段。它主要是用来实现任务间的互斥和同步。

（1）互斥可以确保任务有序的访问共享资源。它提供了比禁用中断和禁用剥夺更为细粒度的互斥技术。

（2）同步可以协调一个任务的执行和它需要响应的外部事件。

一个互斥量可以被看作一个标记，它有俩个值：有效（满）和无效（空）。

当一个任务试图获取一个互斥量时，其结果取决于这个互斥量是有效的还是无效的。如果互斥量是有效的，那么互斥量将变为无效，当前任务将继续运行。如果互斥量是无效的，那么当前的任务将被放入一个阻塞队列，直到互斥量变为有效。

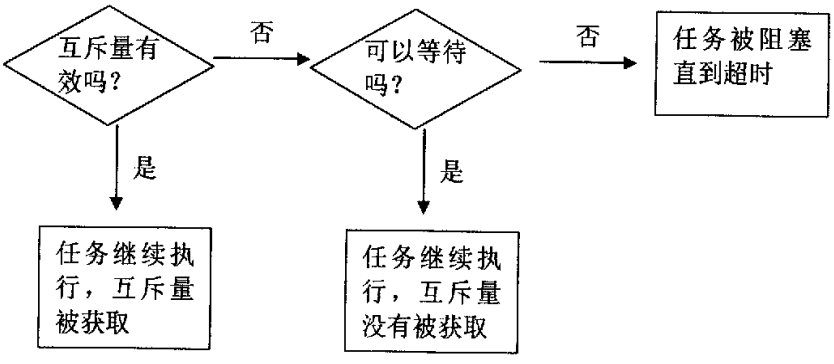


图 9：获取一个互斥量

当一个任务释放一个互斥量时，其结果也是取决于这个互斥量是有效的还是无效的。如果互斥量是有效的，那么这个操作将没有任何作用。如果互斥量是无效的，并且没有任务等待去获取这个互斥量，那么互斥量的值将变为有效。如果

互斥量的值是无效的，并且至少有一个任务在等待它，那么这个互斥量的阻塞队列中的第一个任务将会变为就绪状态，互斥量仍然保持无效。与互斥量有关的 API 如表 6 所示：

系统调用名称	说明
mutex_create	创建一个互斥量
mutex_lock	获取一个互斥量，如果无效则阻塞当前任务，或因超时而返回一个错误
mutex_try_lock	试图获取一个互斥量，如果互斥量无效则返回一个错误，不阻塞当前任务
mutex_unlock	释放一个互斥量

表 6：与互斥量相关的 API

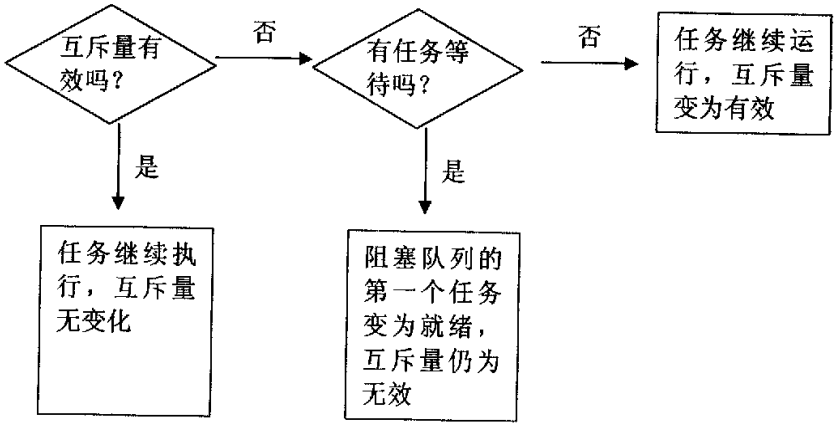


图 10：释放一个互斥量

2.4.3.1 使用互斥量进行互斥

互斥量可以非常有效的对共享资源进行加锁。与禁用中断和禁用剥夺不同的是，互斥量把互斥的范围局限在与这个任务相关的任务中。要使用这种技术，就要创建一个互斥量去保护这个共享资源。互斥量的初始值是有效。当一个任务要去访问这个共享资源，它首先要获取这个互斥量。但这个任务得到互斥量之后，所有试图访问这个共享资源的任务都会被阻塞在这个互斥量上。当得到互斥量的任务结束了共享资源的使用后，就会释放这个互斥量，从而允许另一个任务访问这个资源。

2.4.3.2 使用互斥量进行同步

当一个互斥量用于同步时，它可以用来表示一个任务要等待的条件或事件。在初始时，互斥量是无效的。一个任务或是 DSR 可以用释放这个互斥量来表示事件或是条件的发生。等待的任务会一直阻塞，直到事件发生，互斥量变为有效为止。

从这里可以看出，用互斥量实现同步和互斥时的不同做法。在实现互斥时，互斥量的初始值是有效的，每个需要访问共享资源的任务先要获取互斥量，然后再释放互斥量。而在实现同步时，互斥量的初始值是无效的，一个任务在互斥量上阻塞，等待着另一个任务或是 DSR 来释放这个互斥量。

2.4.3.3 优先级反转

所谓优先级反转是指一个高优先级的任务被一个低优先级的任务无限制的阻塞。考虑图 11 的情景：Task1、Task2、Task3 的优先级分别为低、中、高。Task1 为了访问共享资源而获取了相应的互斥量 M1。由于 Task3 的优先级更高，所以它将剥夺 Task1。当 Task3 获得 CPU 后，它也要访问那个共享资源，但是它会阻塞在 M1 上。这是没有问题的，因为资源是不可剥夺的，并且在 Task1 完成了共享资源的操作之后，Task3 就可以继续运行。但是，Task2 是可以剥夺 Task1 的（Task2 比 Task1 的优先级高），并且 Task1 在阻塞时还持有着一个独占资源。

在这种情况下，Task2 与 Task3 没有共享任何资源，Task3 的优先级高，却被 Task2 无限期的阻塞。这种情形就被称作优先级反转。

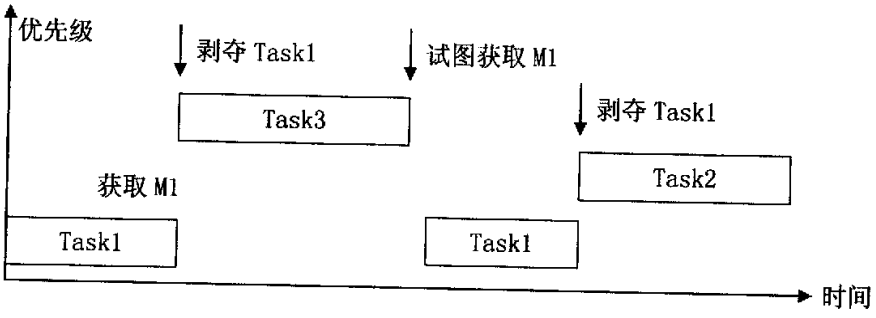


图 11：优先级反转

FRTOS 在编译时提供一个宏，可以用于启用或禁用一个种叫做优先级继承的算法。优先级继承算法可以保证任何一个持有资源的任务都运行在等待这个资源的优先级最高的任务的优先级上。当一个任务的优先级被提升后，它将一直保持这个优先级，直到这个互斥量被释放。因此，优先级继承的算法可以让低优先级的任务在获得资源后获得一个高的优先级，从而使它不会被一个中等优先级的任务剥夺。在图 12 中，在 Task3 被阻塞时，Task1 的优先级将被提升到同 Task3 一样高，因而不会被 Task2 剥夺。这就解决了优先级反转的问题。

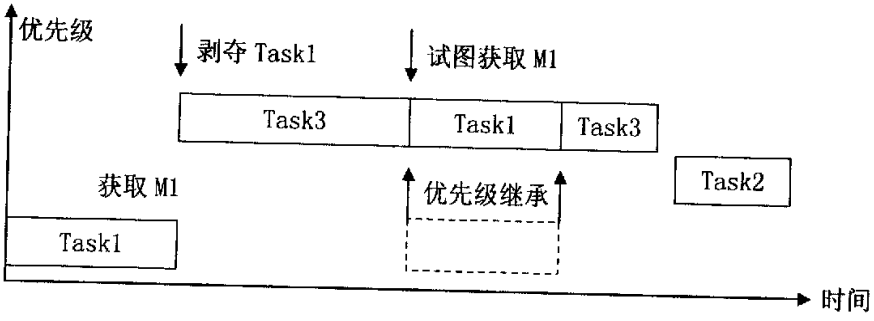


图 12：优先级继承

采用优先级继承会给系统带来一些额外的开销，所以 FRTOS 还提供了一种轻量级的协议，叫做优先级置顶（priority-ceiling）。在采用这种方法时，任何一个任务在获得一个互斥量后，其优先级都会被提升到一个预定义的值（这个值在创建互斥量时指定）。

FRTOS 在实现优先级继承或优先级置顶时，利用了每个任务 TCB 中有的两个表示优先级的成员，即任务的实际优先级和任务的当前优先级，调度器会根据任务的当前优先级来进行调度。

假设有两个任务 T1 和 T2，T1 拥有了一个互斥量，且 T2 的优先级比 T1 高，现在 T2 也要访问这个互斥量，这时 T2 将阻塞在这个互斥量上。如果使用优先级继承，由于 T2 的优先级比 T1 高，则 T1 的当前优先级就会继承 T2 的优先级，一直保持到 T1 释放这个互斥量，T1 释放这个互斥量后，会用自己的实际优先级字段中的值来恢复当前优先级的值。

2.4.3.4 递归获取互斥量

互斥量可以被递归的获取。这意味着一个任务在获得了一个互斥量后，在最终释放之前它可以成功的获得多次这个互斥量。当多个函数互相调用时，如果所有函数都需要互斥的访问共享资源，那么这种可递归的获取方法就非常有用了。

FRTOS 是用记录每个互斥量的所有者来实现这个功能的。每个互斥量的数据结构中都有一个字段来记录这个互斥量的拥有者。当获取或者试图获取一个互斥量时，FRTOS 都会先检查这个字段，如果该字段标识的拥有者就是该任务自己，那么就直接返回获取成功，如果不是才进行真正的获取互斥量的操作。

2.4.3.5 超时

用户可以为获取互斥量的操作指定一个超时。如果获取互斥量的操作需要阻塞，那么等待互斥量的操作将被局限在这个时间段中。如果在这个时间段中，互斥量还是无效的，那么获取操作就会因失败而返回。

FRTOS 在实现超时，是在每次时钟中断时，检查每个任务 TCB 中的一个表示超时的字段，首先将这个字段减 1，再检查是否为 0。如果为 0 就表示超时，这时该任务获取互斥量的操作将失败。

2.4.3.6 等待队列

FRTOS 有一个特性，可以指定互斥量等待队列的类型。用户可以在两种队列中选择：先进先出队列和优先级队列。见图 13。

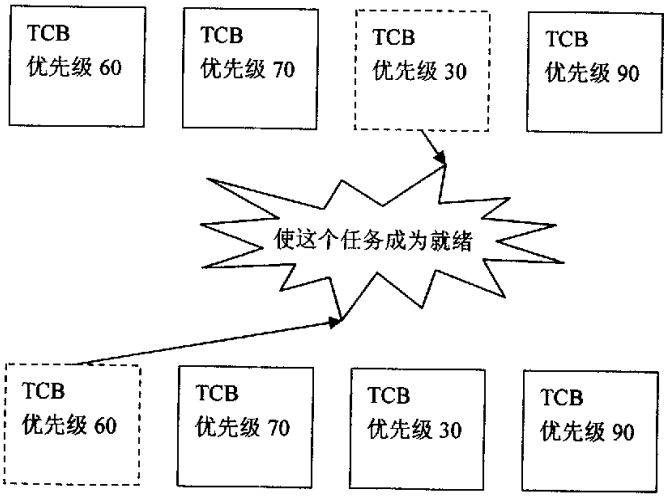


图 13：先进先出队列和就绪队列

优先级队列可以保证系统的实时性，并可以与基于优先级的调度方法保持一致。但是，相比之下，它的开销更大一些（与先入先出的队列相比）。

2.4.4 信号量

信号量也是实现任务同步和互斥的另一种方法。信号量和互斥量很相似，不同之处在于信号量有一个计数器，可以跟踪它被释放的次数。每当一个信号量被释放，它的计数器就会被增 1；每当一个信号量被获取，它的计数器则减 1。当计数器的数值达到 0 时，一个试图获取信号量的任务将被阻塞。当一个信号量被释放时，在它上面阻塞的一个任务将被变为就绪状态；当没有任务在这个信号量上阻塞时，信号量的计数器便会增 1。也就是说，如果一个信号量被释放 2 次，那么就可以不阻塞的被释放 2 次。

系统调用名称	说明
semaphore_create	创建一个信号量，可以指定一个初始值
semaphore_signal	释放一个信号量
semaphore_wait	获取一个信号量，如果无效则阻塞当前任务，或因超时而返回一个错误
semaphore_try_wait	试图获取一个信号量，如果无效则不阻塞当前任务
semaphore_peek	获取当前信号量的值

表 7：与信号量相关的 API

信号量在保护一个资源的多个拷贝时非常有用。例如，要访问 5 个磁带驱动器，可以用一个初始值为 5 的信号量来协调；要实现一个容量为 256 的环形缓冲区，可以使用一个初始值为 266 的信号量

信号量与互斥量一样，也可以指定超时和等待队列的类型。

2.4.5 邮箱

现代的实时应用系统大都由一组相互独立而又互相合作的任务组成。虽然互斥量和信号量提供了一种非常高效的同步和互斥的手段，但是任务间还是需要一

种高级通信机制。FRTOS 是靠邮箱来实现这一功能的。

系统调用名称	说明
mailbox_create	创建一个邮箱
mailbox_put	往邮箱里放入一个消息，如果邮箱已满，则阻塞该任务，如果阻塞超时就会返回一个错误
mailbox_get	从邮箱中读出一个消息，如果邮箱为空，则阻塞该任务，如果阻塞超时就会返回一个错误
mailbox_try_get	试图往邮箱里放入一个消息，如果邮箱已满，返回一个错误，不阻塞该任务
mailbox_try_put	从邮箱中读出一个消息，如果邮箱为空，返回一个错误，不阻塞该任务

表 8：与邮箱相关的 API

FRTOS 的邮箱可以存放很多消息，一个任务或是 DSR 可以向邮箱中发送一条消息，另一个任务可以从邮箱中取出一条消息。

当一个任务或是 DSR 向邮箱中发送一条消息时，如果没有任务阻塞在这个邮箱中，这个消息就会被放在邮箱中；如果有任务因为等待消息而阻塞在这个邮箱上，这个消息将被直接发送到等待队列的第一个任务上（如互斥量和信号量一样，等待队列可以有优先级队列和先入先出队列两种）。

当一个任务从邮箱中取出消息时，如果邮箱中不为空，则邮箱中的第一个消息将被返回；如果邮箱中为空，取消息的任务将被加到邮箱的等待队列中。与互斥量信号量一样，这个等待操作也可以指定超时。

图 14：邮箱的使用

多个任务可以使用同一个邮箱来发送或接收消息。一般来说，两个任务间的双向通讯需要使用两个邮箱。

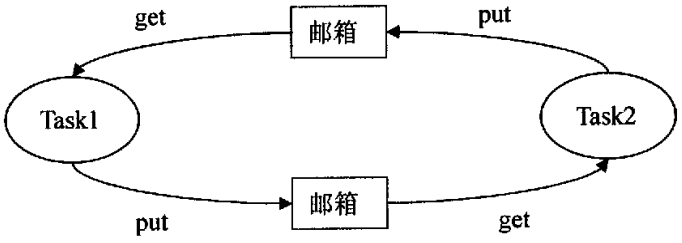


图 15：使用邮箱进行双向通信

实时系统经常被组织成一种客户机-服务器模型。在这种模型中，服务器任务接收来自客户机任务的请求，完成一些服务，并返回给客户机任务一个应答。这种模型可以通过邮箱来实现（如图 16 所示）：

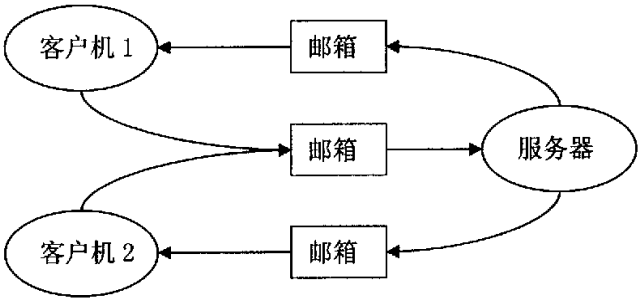


图 16：用邮箱实现客户机-服务器模

服务器任务为了从每一个客户机任务接受请求，需要创建一个邮箱。每一个客户机任务为了接受从服务器放送过来的应答需要创建一个邮箱。每一个请求包

含一个标记位 TID，可以标识消息的来源。一个服务器任务主要由一个循环构成包括：从它的邮箱中取出消息、处理消息和给相应的任务发送应答。

2.4.6 信号机制

FRTOS 还提供了一种软件上的信号机制。信号机制可以异步的改变一个任务的工作流。任何一个任务或是 DSR 都可以为某个任务激发一个信号。收到信号的任务在它下次得到 CPU 时，会立即中止它的当前执行，并且执行一个由该任务指定的信号处理函数（即一个回调函数）。这个信号处理函数执行在任务的上下文中，并且使用该任务的堆栈。

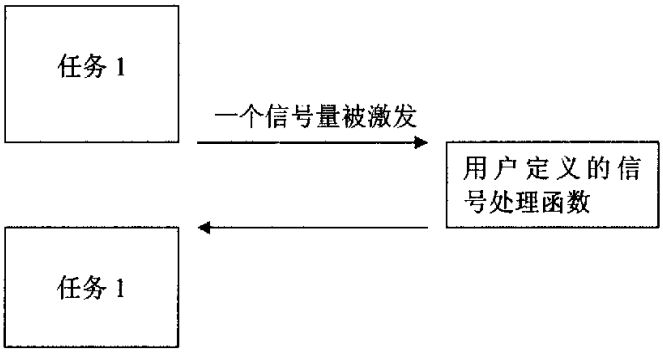


图 17：信号的激发和信号处理函数

与作为一个通用的任务间通信手段相比，信号机制更适合作为一个错误和异常处理的手段。信号处理函数像 DSR 和 ISR 一样，不能调用任何会导致当前任务阻塞的 API。

系统调用名称	说明
signal_register	注册一个信号处理函数
signal_kill	激发一个信号量

表 9：与信号机制相关的 API

2.5 中断处理

硬件中断的处理在实时系统中非常重要，这是系统响应外部事件的唯一方法。为了加速中断的响应速度，FRTOS 采用了两种方法：

（1）中断处理函数（ISR、DSR）在一个单独的上下文中执行

（2）中断服务被分为两个过程 ISR 和 DSR

在传统的中断处理中，所有的处理工作都由一个中断处理函数（ISR：interrupt service routine）完成。操作系统在内存中维护一张中断向量表（interrupt vector table），为每个中断入口存储一个函数指针（如图 18 所示）。

1	ISR1
2	ISR2
3	ISR3
4	ISR4
5	ISR5

图 18：中断向量表（传统）

每当有一个中断发生时，内核会在中断向量表中查找相应的中断函数服务程序（即一个函数指针），并执行这个函数调用。

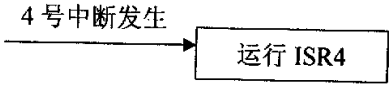


图 19：中断服务程序的运行（传统）

在大多数情况下，ISR 需要访问一些全局数据结构，如释放一个信号量（用于处理环形缓冲区）。所以为了防止发生竞争条件（race condition）和数据不

一致（data inconsistency），在内核需要访问全局数据结构时（即指进入临界区时），必须阻止 ISR 的运行。这是靠在临界区中屏蔽硬件中断来实现的。

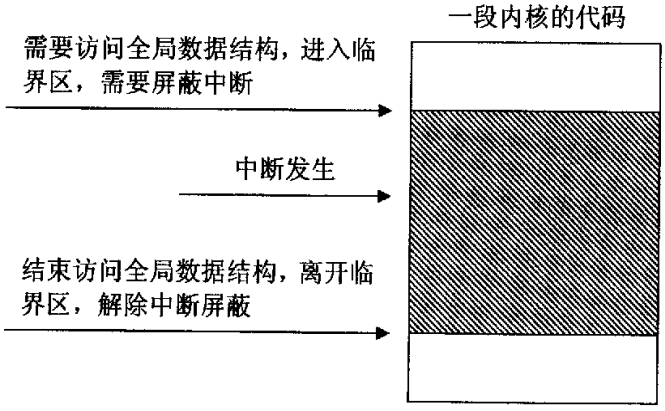


图 20：控制内核代码与 ISR 的并发数据访问（传统）

如图 20 所示，在内核代码访问全局数据结构（即这个数据结构在 ISR 中也会被访问）时，必须屏蔽中断。在这期间，即使有中断发生，内核也不会响应。只有等待内核完成对全局数据结构的访问，离开临界区，这个中断才会被响应。如果在内核屏蔽中断期间，发生了两个同一类型的中断，那么第一个中断将会被丢失。这是因为中断的硬件机制并不提供队列机制，只能有一个中断信号保存在硬件寄存器中。这是非常严重的，丢失中断在实时系统中是不可接受的。

为了解决这个问题，FRTOS 引入了可延迟中断服务程序（DSR: deferred interrupt service routine）的概念，从而提供了一种完全不同的中断处理机制。FRTOS 将传统的中断处理程序分为两部分：中断服务程序（缩写为 ISR）和可延迟中断服务程序（缩写为 DSR）。

其中 ISR 会在中断发生时立即执行，它只会执行一些非常重要的操作，如将硬件寄存器的值写到系统缓冲区中。在 ISR 中不允许调用任何 FRTOS 的 API，因此它不会访问任何内核的数据结构，所以 ISR 可以随意中断内核的执行。而中断

服务程序的另一部分 DSR，则会在内核认为安全的时候执行。中断处理中所有需要访问全局数据结构、调用内核 API 的代码都应该放在这里。

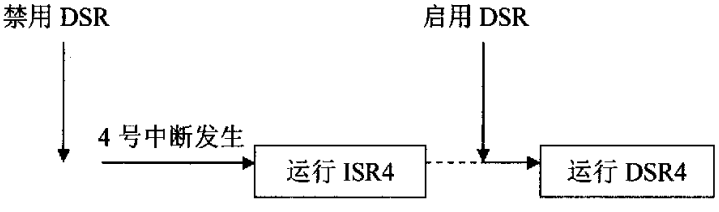
为了实现这一机制，用户要为每个中断向量提供两个可供调用的函数，如图 21 所示：

1	ISR1	DSR1
2	ISR2	DSR2
3	ISR3	DSR3
4	ISR4	DSR4
5	ISR5	DSR5

图 21：中断向量表（FRTOS）

在发生中断时，如果 DSR 没有被屏蔽，那么 ISR 和 DSR 会被依次执行，如图 22（a）所示。如果在中断发生时，DSR 被禁用，那么只有 ISR 会被执行，只有等到内核解除对 DSR 的禁用，这个中断的 DSR 才会被执行。

（a）中断发生时，DSR 没有被禁用



（b）中断发生时，DSR 被禁用

图 22：中断服务程序和可延迟中断服务程序的执行（FRTOS）

当内核需要访问全局数据结构时，与传统的禁用中断不同，FRTOS 只需要禁用 DSR 的运行。在这一时间段中，即使有中断发生，由于 ISR 不会访问任何内核数据结构、内核并没有屏蔽中断，所以这个 ISR 可以被立即被执行。而相应的 DSR 则会被放到一个队列中（一个存放函数指针的队列），在内核离开临界区时，这些 DSRs 会被依次执行（如图 23 所示）。

实际上，可以认为 FRTOS 中存在一个优先级最高的任务，他的任务就是执行 DSR 队列中的函数，当内核进入临界区时，这个任务会被挂起（suspend）。在系统中，只要 FRTOS 没有禁用 DSR，这个任务就会不停的从 DSR 队列中取出函数指针，并执行它们。

由于 FRTOS 的 ISR 会立即执行，相应的 DSR 又拥有一个队列，所有任何中断信号都不会被丢失。由于内核不会屏蔽中断，中断延迟也会大大降低。

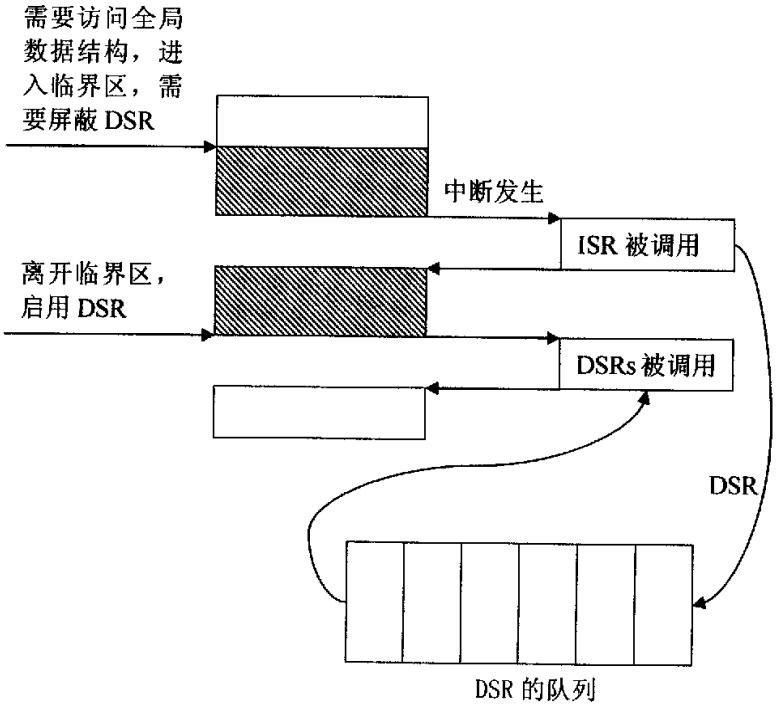


图 23：控制内核代码与 ISR/DSR 的并发数据访问（FRTOS）

2.6 抽象文件系统（abstract file system）

从严格的意义上说，文件系统并不属于内核的一部分，所以 FRTOS 并没有提供文件系统。但是它提供一个叫做“抽象文件系统”的框架，允许把各种文件系统、设备驱动程序、套接字(socket)，都挂接在一起，使用同一套 API(read, write, open, close...）来访问。例如，对于 read（）操作，既可以用它读取 flash 上的文件，也可以用它来在串口上读取数据。

这很像面向对象系统中的多态性，内核只是规定了一些文件访问的接口，并没有实现真正的逻辑。每个想要与 FRTOS 交互的文件系统都要提供这些访问接口、添加一个粘合层。

如果是使用面向对象语言，则可以使用抽象基类、纯虚函数、重载等概念，方便地实现这种可扩展的框架。但是，FRTOS 是用 C——一种非面向对象语言来实现的，这就需要用 C 提供的语言机制来模拟抽象基类、纯虚函数、重载等概念。即使用非面向对象语言来模拟一种面向对象语言。

FRTOS 使用了一个函数指针跳转表来模拟面向对象语言中的虚拟函数表。每一个被打开的文件都有一个指针指向“文件操作函数指针表”，确保用户所发出的 read()、write()被分发到正确的函数入口。

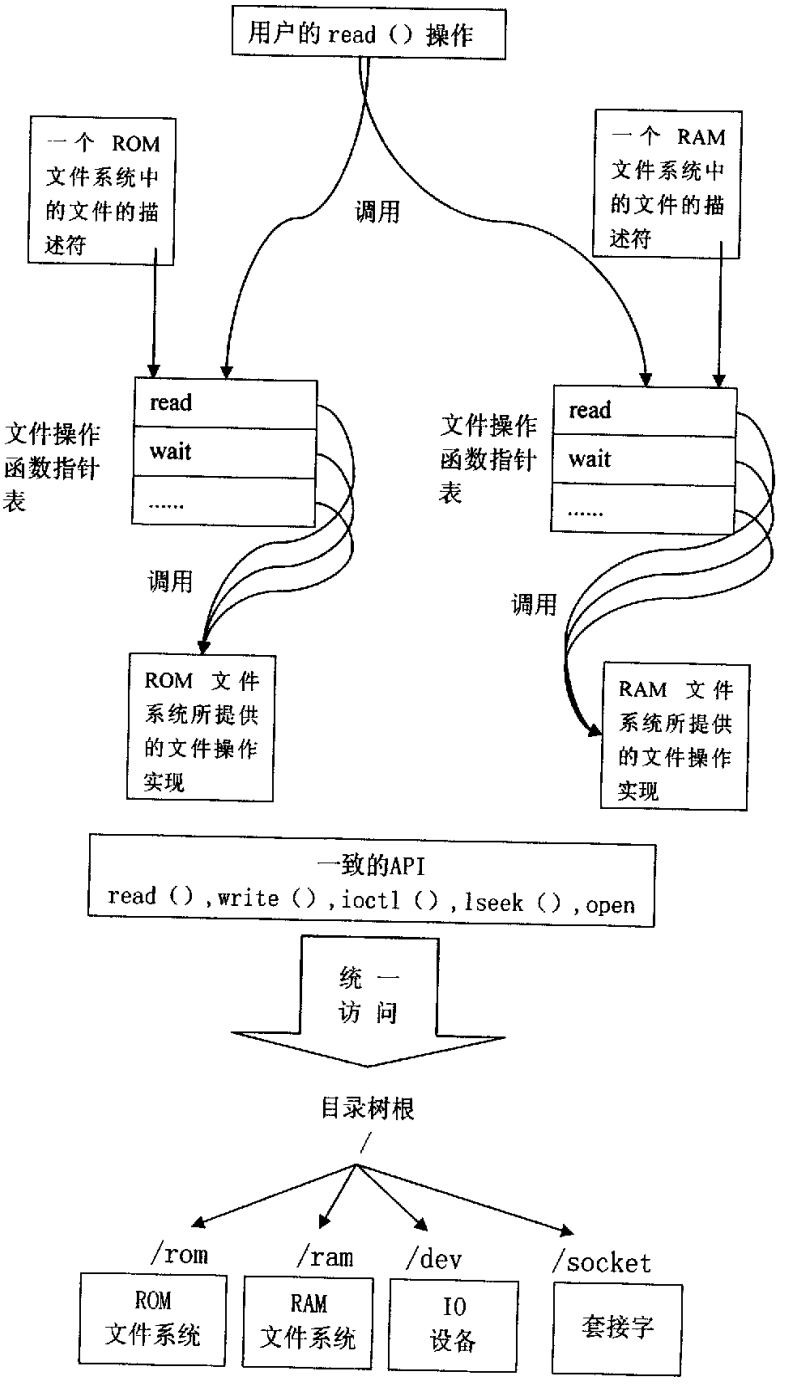

用户的 read（）操作
一个 ROM 文件系统中的文件的描述符
调用
一个 RAM 文件系统中的文件的描述符
文件操作函数指针表
read
wait
.....
文件操作函数指针表
read
wait
.....
调用
调用
ROM 文件系统所提供的文件操作实现
RAM 文件系统所提供的文件操作实现
一致的API
read（），write（），ioctl（），lseek（），open
统一访问
目录树根
/
/rom
/ram
/dev
/socket
ROM 文件系统
RAM 文件系统
IO 设备
套接字

图 24：抽象文件体统

2.7 设备驱动程序

FRTOS 是一个非常灵活的框架，为它撰写应用程序是非常容易的。用户只需要实现 read()、writer()、ioctl()等操作函数，和中断服务程序、可延迟中断服务程序，并且把它挂接到文件系统（在/dev 目录下）中,就可以通过操作文件的 API 像访问文件一样访问这个设备，非常方便。

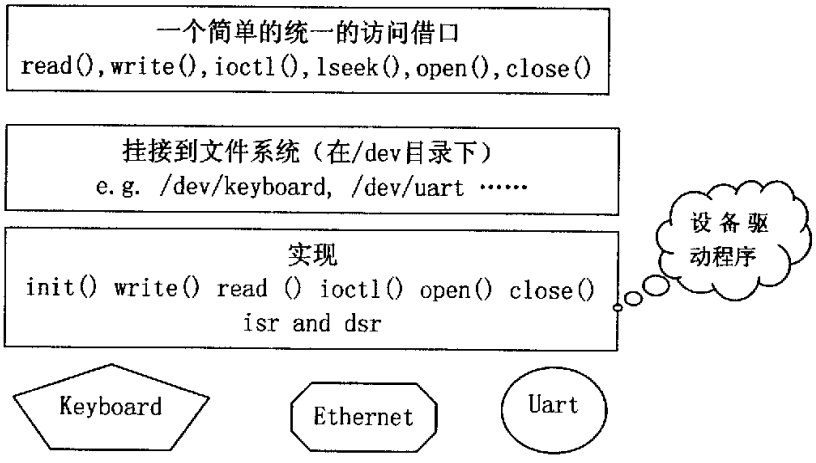


图 25：设备驱动程序

2.8 硬件抽象层

为了方便的把 FRTOS 移植到不同的硬件平台上，在内核中有一个硬件抽象层是非常重要的。这个硬件抽象层封装了所有跟硬件相关的部分，内核中其它部分都是与硬件无关的。如果内核需要访问系统的硬件，它会调用硬件抽象层，不会引用任何跟硬件相关的代码。当需要把 FRTOS 移植到一个新的硬件平台时，只需要重写这个硬件抽象层即可。

FRTOS 的硬件抽象层主要包括五个部分：任务切换、时钟、系统初始化、中断控制器和错误诊断。

2.9 编译时期配置

在嵌入式系统中量体裁衣、按需编译是非常重要的。FRTOS 内核中有很多参数需要用户根据具体情况来进行配置，比如，是否启用时间片调度、时间片的大小、是否启用优先级继承等等。

有多种方法可以实现内核的可配置性，其中编译时期的宏无疑是最有效的。它可以保证不需要的代码不会被编译到内核中，从而减小了内核的体积；这种方法也不会产生任何运行时期的开销。FRTOS 提供了很多编译时期的宏，主要有如下表所示。

宏的名称	说明
MAX_PRIORITY	最大可支持的优先级个数，最大为 255
TOTAL_STACK_SIZE	为所有的任务分配的栈的总大小
CPU_TICK	CPU 时钟中断的频率，单位为毫秒
ENABLE_SLICE	是否启用时间片调度，该值为 1 则启用
SLICE_SIZE	在启用时间片调度的情况下，设定时间片的大小，单位为 CPU_TICK
ANTI_PRIORITY_INVERSION	控制优先级反转的策略，为 0 是什么都不作，为 1 是使用优先级继承，为 2 是使用优先级置顶

表 10：一些典型的编译时期宏选项

3 FRTOS2 的设计与实现

FRTOS2 是一个双内核的实时操作系统。它基于前面所讲述的 FRTOS 内核，并且将 uCLinux（Linux 的一个不需要 MMU 的变种）内核移植到 FRTOS 上——作为 FRTOS 优先级最低的一个任务来运行。在这个系统里，一个硬实时操作系统与一个传统的分时操作系统在同一个单片机上执行。

FRTOS2 主要是靠模拟中断控制器来实现这一功能的。FRTOS2 为 uCLinux 模拟了一个中断控制器，并且重写了 uCLinux 的硬件抽象层中有关中断的部分。所有的硬件中断都由 FRTOS 接管，有选择的转发给 uCLinux，这样就可以保证整个系统的中断延迟都很低。

为了让 FRTOS 和 uCLinux 进行通信，系统还提供了一个 FIFO(first in first out 先入先出)的队列。

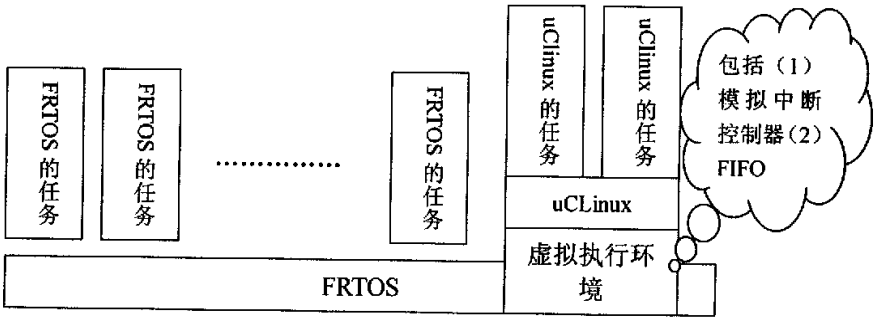


图 26：FRTOS2 的体系结构

3.1 动机

Unix/Linux 最初的设计是一个分时（time-sharing）系统。很多现代的 Unix

实现保留了这个特性,它们竭尽全力提高系统的“平均”性能。显然，这根本不能满足硬实时系统的需要。

Linux/Unix 的很多设计/实现元素决定了上述这个事实：比如分时的调度器、低粒度时钟、内核的不可剥夺性、禁用中断等等。

很多 Unix(尤其是 Linux)调度器的目的是在多个进程间公平的分配处理器,在响应时间和吞吐量之间寻找平衡。每个进程的优先级会在运行中根据很多因素来改变，比如说所占用的 CPU 的时间、IO 的强度等等。例如在 Linux/Unix 的分时调度中，如果一个进程的时间片没有用完就被打断，那么这个进程的优先级就会自动地升高，相反如果这个进程一直运行完它的整个时间片，那么这个进程的优先级就会降低。这是为了更加有利于交互式程序的运行，这些程序大部分的时间会阻塞在 I/O 上，如文本编辑器。

此外传统 Linux/Unix 的时钟粒度过大，例如 sleep()是以秒为单位计算的；再比如，为了不让系统的负载过重，时钟中断的频率很低。

更为重要的 Linux/Unix 大量的把屏蔽中断作为实现互斥的主要手段，使得中断延迟非常长，并且是不确定的。在这一期间任何外界的中断都不会被响应，甚至会造成中断信号的丢失。此外内核还在很多地方禁用了剥夺，使得处于就绪态的高优先级任务无法运行，造成了严重的优先级反转问题。

总之，Linux/Unix 不是一个实时操作系统，在上面开发硬实时应用程序是不可能的。

但是，Linux/Unix 也有很多的优点，比如稳定、功能强大、标准化的应用程序接口等等。此外，我们也应该看到很多实时系统中包含着非实时的部分，比如在一个传统的工业控制程序中，它就包括实时部分（从传感器接收数据，并且发出控制命令）和非实时的部分（把传感器数据显示在 LCD 上，并通过网络发送出去）。

在这个实时部分中，FRTOS 无疑会发挥它的优点：精准的实时调度、极短的中断延时和高效的进程间通信等等。

在非实时部分中，无疑 Linux/Unix 可以发挥它的优点，例如，在 Linux/Unix

上有各种各样的设备的驱动程序、完整的 TCP/IP 协议栈、程序员熟悉的编成接口等等。

毫无疑问，这种实时和非实时混合存在的应用是非常多的，如果能想一个办法来发挥 FRTOS 和 Linux/Unix 两方面的优点，即在不损失实时性的情况下提高程序员的工作效率，无疑是很有意义的。

3.2 双内核实时操作系统

我们需要发挥 FRTOS 和 Linux/Unix 两方面的优势，编写一个允许实时应用和非实时应用共存的系统，这就是 FRTOS2。

FRTOS2 是将 uCLinux 移植到 FRTOS 上，将 uCLinux 作为 FRTOS 上优先级最低的一个任务来运行（uCLinux 是 Linux 的一个不需要 MMU 的变种）。只要有一个 FRTOS 上的实时任务在运行，整个 uCLinux 的内核就会被剥夺。

所有的中断都是由 FRTOS 接管的，所以不会影响系统的实时性。如果没有任务在 FRTOS 上运行，那么 uCLinux 的内核就会占据 CPU，并开始调度它自己的非实时任务。这就是所谓的双内核实时操作系统。

3.3 虚拟中断控制器

为了使 uCLinux 无缝的在 FRTOS 上运行，FRTOS2 提供了一个虚拟中断控制器，并重写了 uCLinux 硬件抽象层中与中断控制器相关的部分。所有的中断首先由 FRTOS 接收，然后有选择的通过虚拟中断控制器转发给 uCLinux。在 uCLinux 看来，一个由虚拟中断控制器发出的中断与一个真实的硬件中断是没有区别的；虚拟中断控制器与真正的中断控制器硬件也是没有区别的。

如果 uCLinux 的内核需要屏蔽中断，那么它只会屏蔽虚拟中断控制器的中断，对真正的中断控制器没有影响，所以整个系统还会响应外界中断（由 FRTOS 来处理这个中断），从而保证了这个系统的实时性。

假设硬件系统中有两个设备 D1 和 D2，用户必须在编译时指定哪个操作系统

控制哪个设备，比如 D1 分配给了 FRTOS，D2 分配给了 uCLinux。那么在 FRTOS 中就会有设备 D1 的驱动程序，而 uCLinux 中会有设备 D2 的驱动程序。如果设备 D1 的中断发生，FRTOS 会直接响应，用户可以通过 D1 的设备驱动程序直接与设备进行交互。如果设备 D2 的中断发生，FRTOS 会发现这个中断属于 uCLinux，虚拟中断控制器会发出相应的中断，用户就可以通过 uCLinux 上设备 D2 的驱动程序与设备进行交互。由于 D1 是 FRTOS 的设备，在 uCLinux 看来这个设备的中断从来没有发生过，因为这个中断会被 FRTOS 处理，而不会通过虚拟中断控制器转发过来。

就这样，FRTOS 通过一种既简单又有效的方式将 uCLinux 移植到了 FRTOS 上。

3.4 系统初始化与入口点

在 FRTOS2 中，由于 FRTOS 会先于 uCLinux 启动。所以，大部分系统初始化工作都是由 FRTOS 来完成的（不包括 uCLinux 所使用的设备）。uCLinux 内核中初始化动态内存、时钟、中断控制器等代码是不会被执行到的，但是 uCLinux 会运行自己使用到的设备的初始化代码，FRTOS 不会负责这些设备的初始化。

在 FRTOS 上会有一个任务用来运行 uCLinux 的内核，这个任务与其它任务没有任何的区别，也拥有自己的 TCB，并在运行、就绪状态之间不停的转换。这个任务的入口点，就是 uCLinux 复位中断的入口点。这个任务（uCLinux 的内核）开始运行时，uCLinux 上的应用程序不会觉察到内核是在一个模拟的环境下运行，它们会觉得内核就是运行在一台真实的计算机上。任何时刻，一旦有一个 FRTOS 的任务就绪，整个 uCLinux 的内核将会被剥夺。

3.5 虚拟时钟中断

时钟是一个特殊的设备，因为它需要被 FRTOS 和 uCLinux 两个内核共享。所以 FRTOS 既要有自己的时钟中断处理程序，又要通过虚拟中断控制器将这个中断转发给 uCLinux。但是，FRTOS 与 uCLinux 时钟中断的频率是不一样的。由于 FRTOS 要保证较高的实时性，所以时钟中断的频率较高，而 uCLinux 为了减小系统的负

载，其时钟中断的频率较低。FRTOS 可以在积攒了一定数目（这个数目可以在编译时指定）的时钟中断后，再通过虚拟中断控制器给 uCLinux 转发时钟中断，从而实现两个内核的时钟中断频率不一样。

3.6 两个内核间的通信手段

在前文所述的工业控制应用中，FRTOS 负责监视和控制设备，uCLinux 负责显示和发送数据。毫无疑问，必须有一种机制使两个内核可以进行通信。FRTOS2 是靠一个全双功（duplex）的先入先出的队列来实现，类似于现代操作系统中管道（pipeline）这种进程间通信手段。

在每个操作系统上，这个队列都是作为一个字符型的设备出现。这两个设备实际上架起了一座通信的桥梁，用户可以通过 read()和 write()等操作，从这个队列中发送或者接受数据，从而完成两个内核间的通信。

4 总结

本文详细地讲述了 FRTOS 的设计与实现：不仅介绍了系统的体系结构、任务的概念、任务调度、任务相关的操作、任务间通信、中断处理等等；还着重写出了优先级继承、可延迟中断服务程序、抽象文件系统等高级特性的实现方法。此为，本文还介绍了如何通过模拟中断控制器来实现双内核操作系统，以便发挥 FRTOS 和 uCLinux 两方面的优势。

无论是 FRTOS 还是 FRTOS2，对我来讲都只是一种尝试。它们只能算是一个原型系统，离一个有工业强度的产品还差得很远。在今后的工作中，我们还需要对他们不断的测试、除错，使它们便稳定。此外，还要考虑把它们移植到有 MMU 的平台上，使 FRTOS 拥有虚拟内存。

参考文献

[1] Jane W. S. Liu. Real-Time Systems[M]. 美国：Prentice Hall，2000

[2] Brian W. Kernighan, Dennis Ritchie. The C Programming Language[M]. 美国：Prentice Hall，1988

[3] Bjarne Stroustrup. The C++ Programming Language: Special Edition[M]. 美国：Addison-Wesley，2000

[4] Jean J. Labrosse. MicroC/OS -II Real-Time Kernel, Second Edition[M]. 美国：CMP Books，2002

[5] Daniel Plerre Bovet, Marco Cesati. Understanding the Linux Kernel[M]. 美国：O'Reilly Media，2005

[6] Wind River Systems, Inc. VxWorks 5.5 Programmer's Guide[M/OL]. www.windriver.com，2002

[7] Michael Barabanov. A Linux-based Real-Time Operating System[D]. 美国：New Mexico Institute of Mining and Technology，1997

致谢

首先要感谢涂老师和陈老师对我的关心和帮助，他们为我提供了非常好的试验条件和硬件设备，并且耐心的解答我的问题；更为重要的是他们为我提供了一个非常宽松的学习环境，使得我可以潜心研究感兴趣的东西。感谢韩国 LG 公司给我们提供的科研经费，使我有机会把自己的想法变成现实。感谢在我三年研究生生活中所有给予过我帮助的人。

论文独创性声明

本论文是我个人在导师指导下进行的研究工作及取得的研究成果。论文中除了特别加以标注和致谢的地方外，不包含其他人或其它机构已经发表或撰写过的研究成果。其他同志对本研究的启发和所做的贡献均已在论文中作了明确的声明并表示了谢意。

作者签名：王振凤　　日期：2006.6.20

论文使用授权声明

本人完全了解复旦大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。保密的论文在解密后遵守此规定。

作者签名：王振凤　　导师签名：徐时文　　日期：2006.6.20