北京邮电大学

硕士学位论文

基于μC/OSⅡ的嵌入式操作系统关键技术的研究与改进

姓名：王劲松

申请学位级别：硕士

专业：电路与系统

指导教师：孙文生

20080201

基于μC /OS II的嵌入式操作系统关键技术的研究与改进

摘 要

近年来，随着信息技术的飞速发展，嵌入式系统具有了新的应用。原有的前后台式嵌入式软件已经不能满足新的需求。嵌入式RTOS（real time operation system 即实时操作系统）支持多任务，同时使得应用程序的开发变得更加容易，也方便日后的维护和二次开发，同时大大提高了嵌入式系统的稳定性和可靠性。

本文基于μC/OS-II开源操作系统，对RTOS的某些关键技术进行了探索和研究。本文对目前有代表性的四种开源RTOS的关键技术进行了分析。涉及到的技术主要包括：任务管理，任务及中断的同步与通信机制，存储器管理和中断管理。

本文的主要工作在于：

1、详细的阐述了μC/OS-II的任务管理和任务调度算法，并提出对原有支持的最大64个任务数进行了扩展，同时提出动态任务调度的算法；

2、文中对在不同处理器中使用不同的任务堆栈策略进行了分析，包括独立任务堆栈和公用堆栈；

3、针对RTOS的关键技术，搭建了测试平台，对改进后的系统性能进行了测试。

在面对实际问题时必须结合实际，具体问题具体分析，才能拿出真正有效的嵌入式解决方案。首先，影响RTOS的性能的因素很多，系统的某一个性能得到提高，并不意味着系统整体性能的提高。其次，对于不同的应用来说，系统的性能要求是不同的。任何一个嵌入式系统都是一个灵活定制的，可裁剪的系统。

关键词：嵌入式RTOS，μC/OS-II，操作系统性能，关键技术，测试平台

RESEARCH AND IMPROVEMENT ON EMBEDDED RTOS KEY TECHNOLOGY BASED ON μC /OS II

ABSTRACT

With the development of info technology, the embedded system enlarged its application area. The original embedded software, such as foreground and background mode software, has been fallen behind. The embedded RTOS (Real Time Operation System), which can support multiple tasks and makes the application development much easier, brings about the privilege in maintenance and second-time development, and improves the stability and reliability as well.

Based on the μC /OS II open ware operation system, some key technologies of RTOS are covered in this assay. Four typical open RTOS are concerned, and their technologies are compared. Furthermore, the technologies are discussed thoroughly, including task management, the synchronization and communication between interrupt and task, the storage management and interrupt management.

The main work of this assay includes:

1. The task management and task scheduler are thoroughly discussed. Besides, the original maximum task number is extended from 64 to 256. The dynamic task scheduling is brought about to improve the scheduler performance;
2. Different task stack using strategies are discussed in different hardware platform, including independent task stack and global stack;
3. The test platform is made to verify the technologies and improvements.

There is no general solution for all the applications. Firstly, many factors are concerning with the RTOS performance. Improvement of one factor does not always lead to the improvement of the whole system. Secondly, different application means

different performance goal. So every embedded system is a customized system, and solution should be made wisely according to the situation.

Key words: embedded RTOS, μC /OS II, OS performance, key technologies, test platform

独创性（或创新性）声明

本人声明所呈交的论文是本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京邮电大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名：王劲松　　日期：2008.2.21

关于论文使用授权的说明

学位论文作者完全了解北京邮电大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属北京邮电大学。学校有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许学位论文被查阅和借阅；学校可以公布学位论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存、汇编学位论文。（保密的学位论文在解密后遵守此规定）

保密论文注释：本学位论文属于保密在__年解密后适用本授权书。非保密论文注释：本学位论文不属于保密范围，适用本授权书。

本人签名：王劲松　　日期：2008.2.21

导师签名：[illegible]　　日期：2008.2.21

第一章 绪论

1.1 嵌入式软件定义

所谓嵌入式软件（ Embedded Software ），从广义上讲是计算机软件的一种，它也是由程序及其文档组成，也可分成系统软件、支撑软件、应用软件三类。

嵌入式软件与嵌入式系统密不可分。嵌入式系统最初是指用以控制设备的计算机，通常是在设备内部，为了控制设备行为或是嵌入在其它系统中的一种专用软件和硬件。它一旦启动就执行某一特定的程序，中间无需人工干预，直到关机为止。以单片机为例，嵌入式系统要求具有实时响应能力，对应的嵌入式软件是采用前后台系统结构，它是由无限循环（后台）和中断服务程序 ISR（前台）组成的。

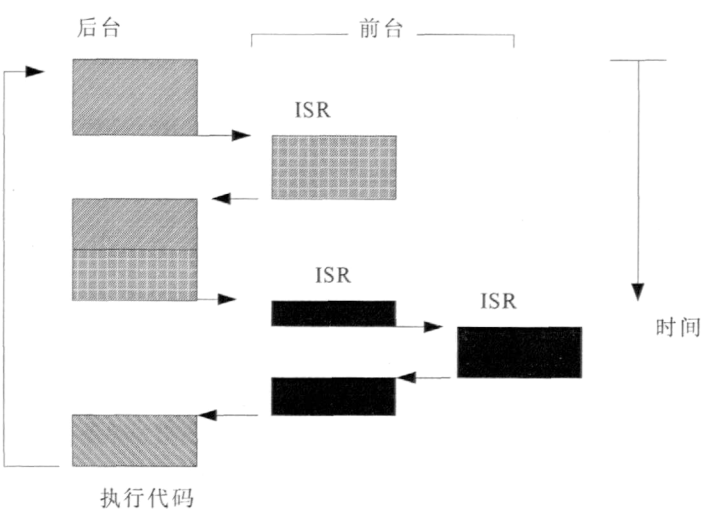

图 1- 1 前后台系统示意图

近年来，随着信息技术的飞速发展，嵌入式系统具有了新的应用。原有的前后台式嵌入式软件已经不能满足新的需求。嵌入式 RTOS（Real Time Operation System, 即实时操作系统）支持多任务，同时使得应用程序的开发变得更加容易，也方便日后的维护和二次开发，同时大大提高了嵌入式系统的稳定性和可靠性。

1.2 RTOS 的定义

根本上讲，实时操作系统 RTOS 是一个程序。它使得多任务得以调度执行，管理资源，并为开发应用代码提供一致的基础。RTOS 最适合于实时，应用特定的嵌入式系统。

在一个 RTOS 上设计的应用代码是相当多样化的，从数字秒表这样的简单应用到飞机导航这样非常复杂的应用。因此，好的 RTOS 是可裁剪的，以满足不同应用的不同需求。例如，在某些应用中，一个 RTOS 只有一个内核。它提供最小逻辑的核心检测软件，调度和资源管理算法。每个 RTOS 有一个内核。另一方面，一个 RTOS 可以是各种模块的组合，包括内核，文件系统，网络协议栈和应用要求的其他部件。

虽然许多 RTOS 可以上下裁剪以满足应用需求，但是其内核上具有的公共部件基本是一致的。大多数 RTOS 内核包括下面的组件：

调度器——包含在每个 RTOS 中，一组算法决定何时执行哪个任务。常见的调度算法包括时间轮换和抢占调度。

对象——是特殊的内核构建，帮助开发者创立实时嵌入式系统的应用。常见的内核对象包括任务，信号灯和消息队列。

服务——是内核在对象上执行的操作或通用的操作，如计时，中断处理和资源管理。

下图列举了这些部件。并非所有的 RTOS 内核与下图都一致。

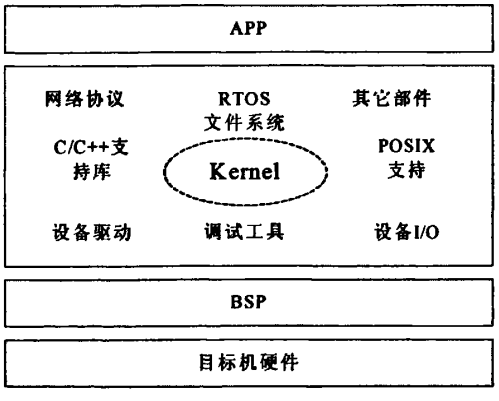

图 1-1 RTOS 的高层视图（含内核和嵌入式操作系统的其他部件）

随着嵌入式 RTOS 迅速的普及与应用，对于它的学习和研究成为了嵌入式系统的一个热点，据统计目前已经问世的各种嵌入式 RTOS 多达几百个，然而这些林林总总的嵌入式 RTOS 在功能和性能上良莠不齐，且相互之间的兼容性较差，基于某嵌入式 RTOS 开发的应用程序无法轻松的移植到另一嵌入式 RTOS。一些由专业的嵌入式软件开发商研发的嵌入式 RTOS 具体功能丰富、扩展性强、性能可靠等特点，同时还提供完善的技术支持，然而要使用这些产品往往需要支付昂贵的费用，甚至有些产品还是非开源的；另外一些由开源社区或个人爱好者开发的嵌入式 RTOS 虽然免费且代码开源，但是由于在可靠性上无法保证且缺乏相关技术支持，这一类的嵌入式 RTOS 通常只能作为研究和学习使用。

在众多的 RTOS 中，我们选择μC/OSII 来作为 RTOS 研究的起点。首先，μC/OSII 是一个开源的 RTOS，且有大量的移植范例可以从网站上获得。其次，μC/OSII 良好的性能可以与许多高端商业软件相媲美，并在世界范围内得到了广泛应用。实际上，μC/OSII 已经通过了非常严格的测试，并且得到了美国航空管理局的认证，可以用在飞行器上，这说明μC/OSII 是稳定可靠的。

1.3 研究内容

本论文的根本出发点和目的是对 RTOS 的关键技术进行研究和扩展，使得其实时性得到提高。

囿于条件限制，作者制定了研究方法，如下：

首先，比较现有 RTOS 的特点，明确 RTOS 的关键性能指标和衡量方法。通过调查，选定μC/OSII 中研究的关键技术和改进的方向。在掌握μC/OSII 构架后，对μC/OSII 涉及的关键技术进行改进，争取提高μC/OSII 的实时性和扩展性。最后针对调试改进的方法，在 Thread-Metric 试验平台上进行性能的测试和对比。

如果没有特别说明，本文研究的μC /OS II 的版本为 v2.6.2。

1.4 论文结构安排

第一章“绪论”给出了嵌入式软件的定义和嵌入式操作系统的基本组成，还

介绍了本论文的研究背景和研究内容。

第二章“RTOS 关键技术概述”，以目前通用的四种开源 RTOS 为例，介绍了 RTOS 的关键技术，并对这些关键技术怎样对 RTOS 的性能产生影响这一问题进行了理论上的探讨。

第三章基于 μC /OS II，对任务管理，同步和通信机制以及中断管理这几个方面进行了讨论。在任务管理技术上，对 μC /OS II 原有的 64 个优先级进行了扩充，使得可调度的优先级达到了 256 个，此外，参照别的开源 RTOS，将 μC /OS II 的静态优先级调度算法改进为支持动态优先级的调度算法。

第四章对于系统在不同处理器下使用不同的堆栈策略进行了讨论。

第五章在 Thread-Metric 试验平台上进行性能的测试，通过分析比较系统的性能数据，对提出的关键技术创新点进行评价。

第六章是对全文的总结和对未来工作的展望。

第二章 RTOS关键技术概述

2.1 嵌入式实时操作系统的关键技术简介

嵌入式实时操作系统（ERTOS）涉及的关键技术有很多，其中内核部分主要包括：任务管理；任务与中断间的同步与通信机制；中断管理；对CPU和存储其的需求等，它们对于RTOS的性能有着重要影响。

任务管理是 ERTOS 的核心和灵魂，决定了 ERTOS 的实时性能。任务管理的目的是让系统运行的多个任务能够无冲突的和谐地共享 CPU 资源。通常涉及到以下技术：动态优先级、时间确定性、基于优先级抢占式调度、时间片轮转调度、多任务调度机制。

任务与中断间的同步与通信是 ERTOS 的重要内容之一，通常通过若干任务和中断服务程序共同完成，任务与任务之间、任务与中断服务程序之间必须协调动作互相配合，这就牵涉到任务与中断间的同步与通信问题。ERTOS 通常是通过信号量（Semaphore）、互斥型信号量（Mutex）、事件标志（Event Flag）和异步信号（Asynchronous Signal）来实现同步，通过消息邮箱（Message Box）、消息队列（Message Query）、管道（Pipe）和共享内存（Shared Memory）来提供通信服务。

中断管理是 ERTOS 内核的重要部分，主要考虑是否支持中断嵌套、中断处理机制、中断延时等内容。大多数的中断处理细节是与体系结构有关的，但 ERTOS 内核中同时会有一些通用的处理中断的机制和接口。ERTOS 一般会允许中断嵌套，也就是说在中断服务期间，ERTOS 可以识别另一个更高级别的中断，并服务于那个更高级别的中断。ERTOS 的实时性能大部分体现在系统对中断请求的响应 和中断服务例程 IRQISR 的处理效率上，因此 ISR 必须尽可能地有效，不能经常或者长时间阻塞中断，是各 ERTOS 的主要目标之一。

另外，ERTOS 的内容还包括 API 特征、CPU 种类和存储容量需求、开发环境和工具等。这些基本内容对于嵌入式操作系统的使用有着重要的影响。

2.2 ERTOS 的比较和分析

2.4.1 任务管理

任务管理是 ERTOS 的核心和灵魂,决定了 ERTOS 的实时性能。任务管理的最终目标是让系统中运行的多个任务按照一定的调度逻辑去无冲突的共享 CPU 资源。任务管理通常涉及到以下技术：动态优先级、时间确定性、基于优先级抢占式调度、时间片轮转调度、多任务调度机制。

ERTOS 函数调用与服务的执行时间应具有可确定性。系统服务的执行时间不依赖于应用程序的多少。基于此特征,系统完成某个确定任务的时间是可预测的。

四种 ERTOS 任务调度机制上的比较，如表 2-1 所示。

操作系统	RT-Linux(实时部分)	μC Linux	μC/OS-II	eCOS	
				位图调度器	多队列调度器
内核抢占	是	否	是	是	是
优先级变化	静态，动态	动态	动态	动态	动态
调度算法	基于优先级抢占式调度；静态采单调调度(RMS)； 动态采用最早期限优先调度(EDF)	分实时进程先来先服务调度；普通进程时间片转调度	基于固定优先级抢占式调度	基于固定优先级抢占式调度	基于优先级抢调度；时间片轮转调度
同优先级调度	有	有	无	无	有
优先级数量	1024（线程）	100	64	32	32
任务数量	无限制	无限制	64	32	无限制
时间的可确定性	是	否	是	是	是

表 2- 2 四种 ERTOS 的调度机制比较

RT-Linux 是在标准 Linux 基础上实现了一个小的基于优先级的抢占式内核，使用双核设计的一种硬实时操作系统 Linux 本身作为一个可抢占的任务在核内运行，优先级最低，随时会被高优先级任务抢占。如果用户觉得有必要，可以自己编写调度算法来替换其中的调度算法。

μC Linux 则在结构上继承了标准 Linux 的多任务实现方式分实时进程和普通进程采用不同的调度策略 即先来先服务(，调度和时间片轮转调度)，仅针对中低档嵌入式 CPU 特点进行改良。当前如果需要实现比较强的实时性效果，减少中断响应时间的唯一办法是使用 RT-Linux 或者 RTAI 所提供的 Linux-in-an-ERTOS 方法。

μC/OS-II 内核是针对实时系统的要求设计实现的，只支持基于固定优先级抢占式调度，相对简单，可以满足较高的实时性要求，用法也简单。μC/OS-II 通过任务就绪表中的 OSRdyGrp 和 OSRdyTbl [0…7] 变量以及任务调度器 OSSched( )，完成对最多 64 个不同优先级任务的调度，而且任何时刻都不可能有相同优先级的任务在就绪表中。

eCos 的调度方法丰富，当前 2.0 版本的 eCos 提供了两种基于优先级的调度器，即位图(Bitmap) 调度器和多级队列(Multi-Level Queue, MLQ) 调度器，用户在进行配置时可以根据实际需要在这两种调度器中选择一个。位图调度器的主要思想是设置若干个不同的线程优先级(系统默认设置 32 个优先级)，任何时刻在每一个优先级只允许运行一个线程。位图调度器所允许的线程个数有严格的限制，如果系统被设置成具有 32 个优先级，那么系统中最多只能有 32 个线程。位图调度器具有简单、高效的特点，但并不支持避免优先级反转的方法、对称多处理器(SMP) 系统的支持等功能。MLQ 调度器是一种支持优先级继承和同优先级的 FIFO 调度策略。MLQ 调度器中相同优先级的线程形成一个 FIFO 队列，这些线程采用时间片轮转策略进行调度。如果内存允许，系统所允许的任务/线程数目是没有限制的。MLQ 调度器调度功能丰富，但需要查找最高优先级这样的大开销操作，执行效率较低。

由于 eCos 采用的是开放式的可配置结构，所以今后还可以再加入别的调度方法。

2.4.2 任务及中断间的同步与通信机制

任务及中断间的同步与通信是 ERTOS 的重要内容之一,通常通过若干任务和中断服务程序共同完成，任务与任务之间、任务与中断服务程序之间必须协调动作互相配合，这就牵涉到任务及中断间的同步与通信问题。ERTOS 通常是通过信号量(Semaphore) 、互斥型信号量(Mutex) 、事件标志(Event Flag) 和异步信号(Asynchronous Signal) 来实现同步，通过消息邮箱(Message Box) 、消息队列(Message Query) 、管道(Pipe) 和共享内存(Shared Memory) 来提供通信服务。

ERTOS 中由于使用了互斥量等，因此常常会面临优先级反转/倒置(Priority Inversion) 问题。优先级反转是一种不确定的延迟形式，任何 ERTOS 都必须要处理这一问题。一般的例子是：一个高优先级任务等待一个互斥量，而这个互斥量正被一个低优先级任务所拥有,如果这时这个低优先级任务被一个或多个中等优先级的任务剥夺了运行，那么就发生了优先级反转，即这个高优先级任务被一个不相关的较低优先级任务阻止了继续执行，实时性难以得到保障。

因此，ERTOS 应尽量避免出现优先级反转，通常可以采用优先级置顶协议(Priority Ceiling Protocol) 和优先级继承协议(Priority Inheritance Protocol) 这两种方法。优先级置顶协议是指所有获得互斥量的任务把它们的优先级提升到一个事先规定好的值。该方法的缺点是：需要事先知道使用这个互斥量任务的最高优先级;如果这个事先规定的值太高，它相当于一个全局锁禁止了所有的调度。

优先级继承协议是指拥有互斥量的任务的优先级被提升到与下一个在等待这个互斥量的最高优先级任务的优先级相等。该技术不需要预先知道准备使用这个互斥量的任务的优先级，当一个更高优先级任务处于等待时，只有拥有互斥量的任务的优先级被提升。该方法减少了对别的任务进行调度的影响，但是它增加了每次同步调用的开销。内核支持优先级继承。

2.4.3 存储器管理

存储管理也是 ERTOS 的重要内容之一，主要包括:内存分配原则、存储保护和内存分配方式。

（1）内存分配原则

①快速性:系统强调对实时性的保证，要求内存分配过程要尽可能地快,通常都采用简单、快速的内存分配方案。

②可靠性:系统强调对可靠性要求，也就是内存分配的请求必须得到满足。

③高效性:系统强调对高效性要求，不仅仅是对系统成本的要求，而且系统本身可配置的内存容量也是很有限的，所以内存分配要尽可能地少浪费。

（2）存储保护

在 ERTOS 中，内存中既有系统程序，又有许多用户程序。为使系统正常运行,避免内存中各程序相互干扰，通常需要对内存中的程序和数据进行保护。存储保护的内容包括：保护系统程序区不被用户有意或无意侵犯；不允许用户程序读写不属于自己地址空间的数据，如系统区地址空间；其他用户程序的地址空间。存储保护通常需要有硬件支持（如 MMU），并由软件配合实现。而事实上在许多 ERTOS 中，内核和用户程序同在相同的内存空间,也就是说没有存储保护。

（3）内存分配方式——静态分配和动态分配

①静态分配:程序要求的内存空间是在目标模块连接装入内存时确定并分配的,并且在程序运行过程中不允许再申请或在内存中移动,即分配工作是在程序运行前一次性完成。采用静态分配方案时,程序在编译时所需要的内存都已分配好，不可避免地使系统失去了灵活性，必须在设计阶段就预先考虑到所有可能的情况，根据需要对内存进行分配，一旦出现没有考虑到的情况,系统就无法处理。当然如果系统对于实时性和可靠性的要求极高（硬实时系统，如 RT-Linux），不允许一点延时或一次分配失败,则必须采用静态分配方案。

②动态分配:程序要求的基本内存空间是在目标模块装入时确定并分配的,但是在程序运行过程中允许申请附加的内存空间或在内存中移动,即分配工作可以在程序运行前及运行过程中逐步完成。采用动态分配方案时,嵌入式系统的程序设计者可以方便地将原来运行于非嵌入式操作系统的程序移植到嵌入式系统中,灵活地调整系统的功能,在系统中各个功能之间作出权衡。因此,大多数实时操作系统提供了动态内存分配接口,例如 malloc( ) 和 free( ) 函数。

(1)RT-Linux

它采用虚拟缓冲技术的页存储机制,每个实时任务以内核线程形式存在,运行在单一的核心地址空间,这样有利于存储保护。RT-Linux 实时内核不直接支持内存分配初始化,将这部分工作交给了 Linux 完成。在实时任务启动之前,Linux 为实时任务动态分配所需要的内存空间,采用 MBUFF 模式(mbuff_alloc, mbuff_free) 在 RT-Linux 与 Linux 之间实现内存共享。因为 RT-Linux 需要系统具有硬实时能力,必须使用静态分配的内存来完成硬实时任务,不可以使用动态内存分配函数 malloc( ) 和 free( ) 。

(2) μC Linux

它是针对没有 MMU 的处理器设计的,不能使用处理器的虚拟内存管理技术,只能采用实存储器管理策略 Real Memory Management) 。系统使用分页内存分配方式,系统在启动时把实际存储器进行分页。系统对于内存的访问是直接的,所有进程中访问的地址都是实际的物理地址。操作系统对内存空间没有保护,多个进程实际上共享一个运行空间。

(3) μC/OS-Ⅱ。它是采用实存储器管理策略,分区内存分配方式,系统在启动时把实际存储器进行分区。μC/OS-Ⅱ为了消除多次动态分配与释放内存所引起的内存碎片的问题,把连续的大块内存按分区来管理,每个分区中都包含整数个大小相同的内存块,但不同分区之间内存块的大小可以不同。利用这种机制,μC/OS-Ⅱ对 malloc( ) 和 free( )进行改造,使得它们可分配和释放固定大小

的内存块，并且使这两个函数的执行时间也固定下来。用户需要动态分配内存时，选择一个适当的分区，按块来分配内存。释放内存时将该块放回它以前所属的分区。μC/OS-Ⅱ也可以使用常规的 malloc( ) 和 free( ) 内存管理函数来增强可移植性，但在使用更严格的场合，应使用系统提供的特殊内存管理。

（4）eCos。它的内存管理有点特殊，但也相对简单，既不分段也不分页，唯一复杂的是可配置部分。eCos 采用一种基于内存池的动态内存分配机制。eCos 的内存管理通过内存池对象来实现，它提供两个类模板，即 Cyg_Mempoolt 和 Cyg_Mempolt2。每个类模板又提供两种可选的内存池。eCos 通过两种内存池类来实现两种内存管理方法：一种是变长的内存池（Variable Size Memory Pool），即内存池根据申请的内存大小进行分配；另一种是定长的内存池（Fixed Size Memory Pool），即内存池以固定大小的块为单位进行分配。变长的内存池使用链表来进行管理，定长的内存池使用位图来进行管理。默认情况下，eCos 使用变长内存方式管理内存，eCos 的 C 库函数 malloc( ) 就是使用变长内存池来实现内存分配的。

2.4.4 中断管理

（1）RT-Linux

RT-Linux 中断分成普通 Linux 中断和实时中断两类。普通中断可无限制地使用内核调用作为实时中断。RT-Linux 处理的第二部分是相当不错的。RT-Linux 采用在 Linux 内核和中断控制硬件之间增加一层仿真软件的方法截取所有的硬件中断实现了一个虚拟中断机制。Linux 中断它发出的中断屏蔽信号和打开中断信号都修改成向 RT-Linux 发送一个信号。如果 RT-Linux 内核收到的中断信号是普通 Linux 中断，那就设置一个标志位；如果是实时中断，就继续向硬件发出中断。Linux 用禁止中断的方法作为同步机制，由于关中断和开中断的混合使用使得中断延时不可确定。但是无论 Linux 处在什么状态，都不会对实时系统的中断响应时间增加任何延迟，导致时间上的不可确定性。Linux 进程的屏蔽中断虽不能禁止实时中断的发生，却可以禁止普通 Linux 中断。Linux 不能中断自

身,只能被 RT-Linux 中断;而 RT-Linux 不仅可以中断 Linux, 而且可以中断自身。如果想在实时处理程序和常规 Linux 驱动程序中处理同一设备 IRQ, 必须为每一个硬中断单独设置 IRQ。

(2) μC Linux

μC Linux 中断处理的核心机制, 沿袭 Linux 中断处理的方法, 将中断处理分为两个部分: “顶半(Top Half) 处理”和“底半(Bottom Half) 处理”。在“顶半处理”中, 必须关中断运行,进行必要的、非常少、非常快的处理, 其他的处理交由“底半处理”处理。“底半处理”中, 执行那些流程复杂的、耗时的且又不十分紧迫的多数工作,并且是开中断运行, 所以运行时, 可以接受中断。在 Linux 系统中, 因为有许多中断的“底半处理”,这些“底半处理”形成一个任务队列,由 Linux 调度管理。在这样的中断机制下,因为“底半处理”队列的调度,所以会引起中断处理的延时。此外,Linux 中的中断可以被系统屏蔽,即使是优先级很高的硬件中断也会因为被系统屏蔽而引起中断响应的延时。

(3) μC/OS-Ⅱ

μC/OS-Ⅱ中断处理, 在四种 ERTOS 中是最简单的。一个中断向量上只能挂一个中断服务子程序 ISR,而且用户代码必须都在 ISR 中完成, ISR 做的事情比较多, 中断延时相对较长。系统提供两个函数 OSIntEnter() 和 OSIntExit() 用来进行中断管理。OSIntEnter() 通知内核即将开始 ISR, 使内核可以跟踪中断嵌套, 最大嵌套深度为 255。在 ISR 的末尾, 使用 OSInt-Exit() 判断中断是否已经脱离了所有的中断嵌套。如果脱离了中断嵌套, 内核函数需要判断是否有更高优先级的任务进入就绪状态, 如果有系统要让更高优先级的任务进入就绪状态。在这种情况下,中断要返回到更高优先级的任务, 而不是被中断了的任务, 因而中断恢复时间要稍长一些。

(4) eCos

eCos 使用了分层式中断处理机制, 这种机制将中断处理分为两部分, 即传统的 ISR 和滞后中断服务程序(Deferred Service Routine, DSR) 。这种机制类似于在 Linux 中将中断处理分为顶部处理和底部处理。这种机制可以在中断允许时运行 DSR,因此在处理较低优先级的中断时, 允许别的潜在的更高优先级

的中断发生和处理。为了使这种中断模型能够有效地工作，ISR 应当快速运行。如果中断引起的服务量少，ISR 可以单独处理这个中断而不需要 DSR。但是，如果中断服务比较复杂，则在 ISR 中只处理必要的工作，这种情况下，ISR 一般仅仅屏蔽中断源，通知 DSR 处理该中断，然后结束。eCos 会在稍后合适的时间执行这个 DSR，在这个时候系统已经允许进行线程调度了。推迟到这个时候运行 DSR，可以使内核使用简单的同步方式。如果中断源是爆发性的，在执行一个被请求的 DSR 之前，可能已经产生了多个中断并且多次调用了这个 ISR，eCos 内核会记录下被调用的 DSR 的次数。在这种情况下，这个 DSR 最终会被调用，其中一个参数告诉它有多少个 ISR 请求执行这个 DSR。

过上述分析可知：RT-Linux 由于采用硬实时解决方案，因而中断延时最短；μC Linux 由于采用软实时方案，中断延时最长；μC/OS-Ⅱ由于采用微内核设计，中断机理简单，中断延时相对较长；eCos 采用可配置微内核设计，中断机理复杂，中断延时相对较短。

2.4.5 小结

这四种源码开放的 ERTOS 在嵌入式系统中已有大量的应用，但它们的特点又不完全相同。通过本文第三部分的分析比较，得出不同 ERTOS 各自适用的领域：

（1）RT-Linux

RT-Linux 最大特点是在成熟的 Linux 基础上实现了一个硬实时内核，充分利用了 Linux 的系统服务，因此适合于需要比较复杂的系统服务，对时间有严格要求的应用领域，航空航天、科学研究、机器人技术以及工业控制和测量。

（2）μC Linux

μC Linux 最大特点在于针对无 MMU 处理器设计，可以利用完全免费且功能强大的 Linux 资源，因此适合开发对时间要求不高的小容量、低成本的各类产品，特别适用于开发与网络应用密切相关的嵌入式设备。

（3）μC/OS-Ⅱ

μC/OS-Ⅱ是一个非常容易学习、结构简单、功能完备和实时性很强的嵌入式操作系统内核，适合于广大的嵌入式系统开发人员和嵌入式 ERTOS 爱好者的

入门学习，以及大专院校教学和科研。μC/OS-II很适合开发那些对系统要求不是很苛刻的，而 RAM和 ROM有限的,各种小型嵌入式系统设备。

（4）eCos

eCos 最大特点是其配置灵活，而且是面向深度嵌入式应用的 很适合于用在一些商业级或工业级对成本敏感的嵌入式,系统如一些消费电子、汽车制造、工业控制等。比较知名的应用有网络彩色激光打印机、Brother HL-2400 CeN Delphi Com-车载信息处理系统 车载信息处理系统（MPU）（Mobile、数字音频播放器、ProdμCtivity Center,MPC） Iomega Hip Zip Ik-endi 指纹识别系统等。

2.3 ERTOS 关键技术对于性能的影响

根据 Evans Data Corporation 的调查显示，开发者最看重的 RTOS 性能中，排在首位的就是实时特性（Real-time Responsiveness)。而系统的实时特性与系统的中断相应和处理延迟，任务上下文切换时间等关系密切。

Evans Data Corporation's December, 2002 Survey revealed the Top 5 RTOS Features most valued by developers:

1. Real-time responsiveness (33.2%)
2. Royalty-free pricing (14.7%)
3. Source code availability (10.6%)
4. Tools integration (IDE) (10.1%)
5. Microprocessor coverage (7.8%)

在 μC /OS II 中，从一个异步事件的发生到系统作出相应的反应这个过程包括：1. 系统外设检测到事件的发生；2. 外设产生中断信号；3. CPU 检测到中断信号并根据中断向量进行跳转；4. 中断服务程序的执行；5. 中断返回。假设这个系统是可剥夺内核，则在中断返回时系统进行任务调度，将进入就绪态的

优先级最高的任务运行。在这个过程中有两条特殊指令：关中断（disable interrupt）和开中断（enable interrupt），可以让微处理器不响应或者相应中断。在实时环境里，关中断时间应该尽量的短。关中断影响中断延迟时间。同时如果关中断时间太长，将会引起中断的丢失。

上述的过程中，有以下几个时间概念来对系统的实时性其进行衡量：

中断延迟

关中断的时间长短是实时内核的重要指标之一。所有实时系统进入临界代码段之前，都必须关中断，这样在临界代码段中，系统不会因为异步事件而发生任务切换，造成多个任务对临界资源进行操作，使得系统发生混乱。在执行完临界代码之后，需要重新开中断。关中断的时间越长，中断延迟就越长。中断延迟由下面的表达式给出。

中断延迟 = 关中断最长时间 + 开始执行中断服务程序第 1 条指令的时间

中断响应

中断响应的定义为：从中断发生到开始执行用户终端服务子程序代码来处理这个中断的时间。中断时间包括开始处理这儿中断前的全部开销。典型的，在实时系统进入临界区代码之前，先要保护现场，即将 CPU 的个寄存器推入堆栈，包括 PC， PSW 等。这段时间被记做中断相应时间。

对于可剥夺型内核，需要调用一个特定的函数。该函数通知内核即将开始中断服务，使得内核可以跟踪中断的嵌套。可剥夺型内核的中断相应时间由下式给出。

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

中断响应是系统在最坏情况下的响应中断时间。某系统 100 次中有 99 次在 50μs 之内相应中断，只有 1 次相应中断的时间是 250μs，只能认为中断响应时间是 250μs。

中断恢复

中断恢复时间的定义为：微处理器被返回到中断了的程序代码所需要的时间，或者返回更高优先级任务的时间。

对于 μC/OS II 这样的可剥夺实时内核来说，中断的恢复要比较复杂一些。典型的，在中断服务子程序的末尾，要调用一个由实时内核提供的函数。这个函数用于判断中断是否脱离了所有的中断嵌套。如果脱离了嵌套，表明可以返回到被中断的任务了。由于中断服务程序 ISR 的执行，内核要判断是否使一个优先级更高的任务进入了就绪态。如果是，则要让这个优先级更高的任务开始运行。在这种情况下，被中断的任务只有重新成为优先级最高的任务而进入就绪态时，才能继续运行。对于可剥夺型内核，中断恢复时间由下式给出：

中断恢复时间 = 判断是否有优先级更高的任务进入了就绪态的时间 + 恢复优先级更高的任务的 CPU 寄存器所需的时间 + 执行中断返回指令的时间

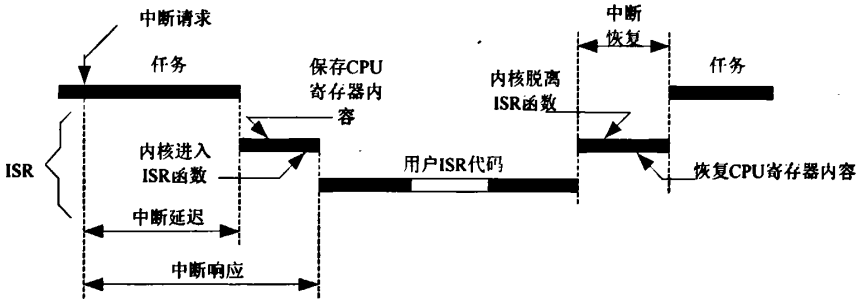

图 2-1 中断延迟，相应和恢复（可剥夺型内核）

从上面的分析可以看出，实时内核的实时性即对于外部事件的反应时间主要取决于操作系统和硬件的中断反应时间（interrupt response）和任务上下文切换时间（context switch requirements）。

此外中断的处理方式也对实时内核的特性有影响。例如如果中断服务子程序的执行时间 T 内 CPU 关中断了，如果系统外部有异步事件发生，则该事件要等到 CPU 重开中断后才有可能得到处理。这样，这段时间内的延迟最多可以达到 T。但是如果在中断子程序中暂开一次中断，就可以使得事件在较短的时间内得到处

理了。像2.2节中 μC Linux 即使用了将中断分解的策略，现对中断进行初步处理，而将处理时间较长的工作留给“下半部”程序进行处理。

2.4 应用的多样性与关键技术的选择

嵌入式操作系统具有嵌入式软件共有的可裁剪、低资源、低功耗等特点；作为实时操作系统除了要满足应用的功能需求以外，更重要的是还要满足应用提出的实时性要求。实时操作系统所遵循的最重要的设计原则是：采用各种算法和策略始终保证系统行为的可预测性。实时操作系统的首要任务是调动一切可利用的资源完成实时控制任务。

上节中我们讨论了嵌入式的各种性能指标。而在实际的应用中，在系统的诸多指标中，不同的应用有着不同的侧重点。高可靠性，实时性，甚至系统的易用性都会对 RTOS 的使用造成很大的影响。

第三章 μC/OS II 系统的任务管理技术和改进

3.1 μC/OS II 系统结构

μC/OS II 的开发者是 Jean J. Labrosse， 他于 1992 年推出了叫做μC/OS 的第一个版本。此后在此基础上又推出了第二版，并称做μC/OS II。这个版本以其精巧，实用而受到了业界的欢迎。μC /OS II 提供了 60 多个系统调用，包括任务时间信号量，互斥型信号量，事件标志组，邮箱，队列和内存管理等功能。

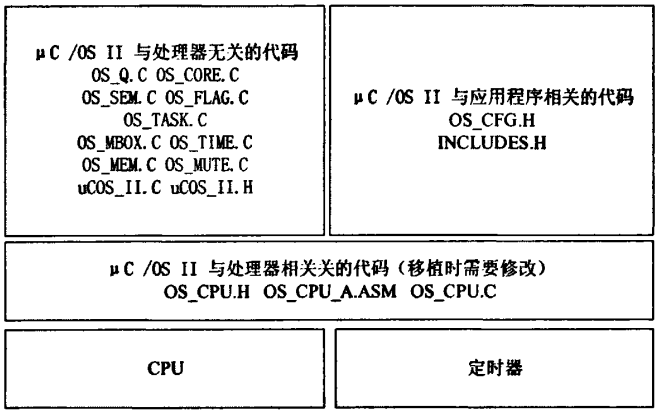

图 3- 1 μC /OS II 的系统结构

μC /OS II 作为一个微内核，他只对计算机的处理器和硬件时钟进行了抽象和封装，而没有提供其他硬件抽象层。之所以这样做，可能是因为μC /OS II 的开发者认为，作为嵌入式系统，在不同的应用中，宿主对象具有差异极大的硬件结构，它们的硬件抽象层只能由硬件供应商或者是目标系统的开发者提供或开发。因此，作为嵌入式操作系统的开发者是没有办法也没有必要提供所有硬件抽象层的。也正是出于这种考虑，使得μC /OS II 具有了较强的可移植性。也就是说，移植μC /OS II 时，其主要工作就是根据具体硬件换一个或者添加一个硬件抽象层。

μC /OS II 是基于优先级的可剥夺型内核，系统中的所有任务都有一个唯一的优先级别，它适合应用在实时性要求较强的场合。

μC /OS II 的另一个特点是它区分用户空间和系统空间，所以它适合应用在比较简单的处理器上。当然，系统和用户共用一个空间会由于用户应用程序与系统服务模块之间联系过于紧密而使系统的安全性变差。但是，由于嵌入式应用的封闭性，从而使系统和用户共用一个空间并不会成为一个很严重的问题，而且在有些时候还会给用户带来某种方便。

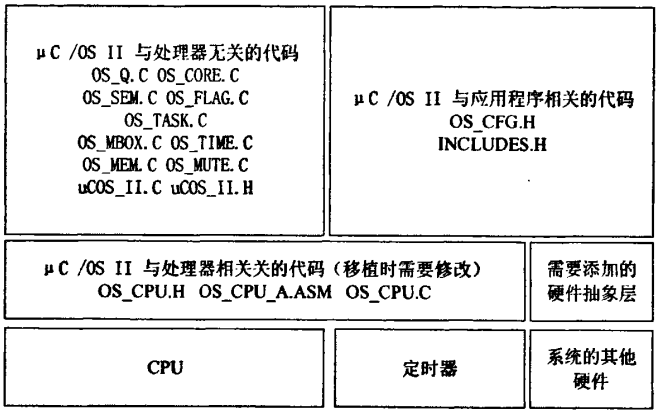

图 3- 2 μC /OS II 需要添加的硬件抽象层

3.2 μC/OS II 的任务管理

3.2.1 任务管理的基本概念

任务

一个任务也叫做一个线程，是一个简单的程序，该程序可以认为 CPU 完全只属于该程序自己。实时应用程序的设计过程包括如何把问题分解为多个任务。每个任务都是整个应用的一部分，都被赋予一定的优先级，都有自己的一套 CPU 寄存器和栈空间（如图 3-3 所示）。

μC /OS II 最多可以管理 64 个任务，建议用户不要使用优先级为 0，1，2，3 以及 OS_LOWEST_PRIO，OS_LOWEST_PRIO-1，OS_LOWEST_PRIO-2，OS_LOWEST_PRIO-3 因为内核可能会用到这些任务。其中 OS_LOWEST_PRIO 是作为定义的常数，在 OS_CFG.H 中用定义常数语句#define constant 定义的，因此可以有多达 63 个用户可使用的任务。如果按照作者的建议，不使用最高和最低的

4 个优先级，则可以有 56 个用户自己的任务。

每个任务赋予不同的优先级。优先级可以为 0 到 OS_LOWEST_PRIO-2。优先级越高，优先级数值越低。μC /OS II 总是进入就绪态优先级最高的任务。目前的μC /OS II 中，任务的优先级编号就是任务编号 ID，优先级号也被一些内核功

能函数调用，如改变优先级函数，以及任务删除函数。

为了使μC /OS II 能管理用户任务，必须在建立任务时，间任务的起始地址与其它参数一一传递给下面的 2 个函数之一：OSTaskCreate()或者 OSTaskCreateExt()。OSTaskCreateExt()是对 OSTaskCreate()的扩展，扩展了一些附加的功能。

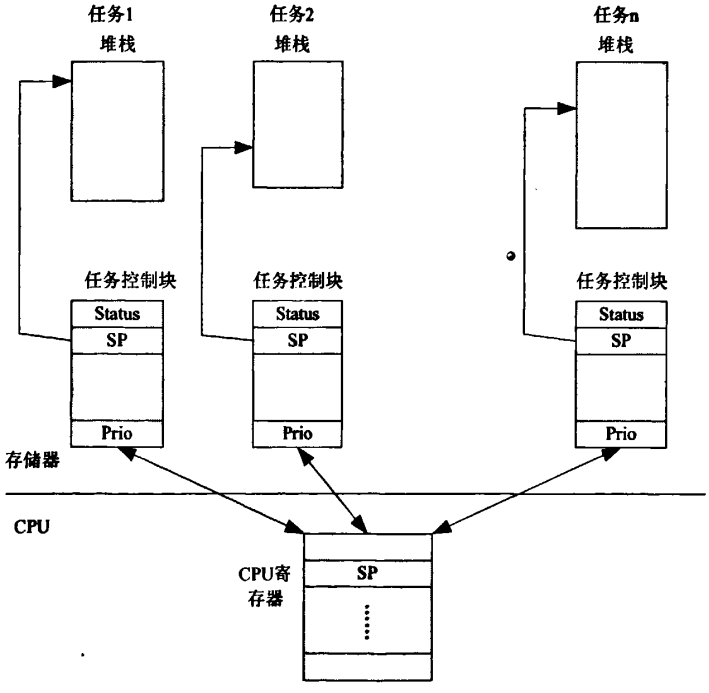

图 3-3μC /OS II 的多任务堆栈

任务状态

μC /OS II 中，任务可以处于以下的状态下，在一定条件下，任务可以在不同的状态中跳转。

睡眠态：指任务驻留在程序空间（ROM 或者 RAM），还没有交给μC /OS II

来管理。

就绪态：任务一旦建立，这个任务就进入了就绪态，准备运行。如果任务已经启动，且一个任务是被另一个任务建立的，而新建立的任务的优先级高于它的任务优先级，则这个刚刚建立的任务将立即得到CPU的使用权。

运行态：调用 OSStart()可以启动多任务。OSStart()函数只能在启动时调用一次，该函数运行用户初始化代码中已经建立的，进入就绪态的优先级最高的任务。优先级最高的任务就这样进入了运行态。任何时刻只能有一个任务处于运行态。就绪的任务只有当所有的优先级高于这个任务的任务进入等待状态或者是被删除了，才能进入运行态。

等待状态：在运行的任务可以通过调用以下2个函数之一，将自身延迟一段时间。这两个函数是OSTimeDly()和OSTimeDlyHMSM()。这个任务于是进入等待状态，一直到函数定义的延迟时间到。

正在运行的任务可能需要等待某一事件的发生，可以通过调用下面函数之一实现：OSFlagPend()，OSSemPend()，OSMuteOend()，OSMboxPend()或者OSQPend()。如果事件并未发生，调用上述函数的任务就将进入等待状态，直到等待的事件发生了。

中断服务态：正在运行的任务是可以被中断的，除非该任务将中断关闭，或者μC /OS II 将中断关闭。被中断的任务于是进入了中断服务程序。响应中断时，正在执行的任务被挂起，中断服务程序控制了CPU的使用权。中断服务程序可能会报告一个或者多个事件的发生，而使得一个或者多个人物进入就绪态。

当所有的任务都在等待事件发生或者延迟结束时，μCOS将执行被称为空闲任务的内部任务，即OSTaskIdle()。

以上所有状态可以用下图表示：

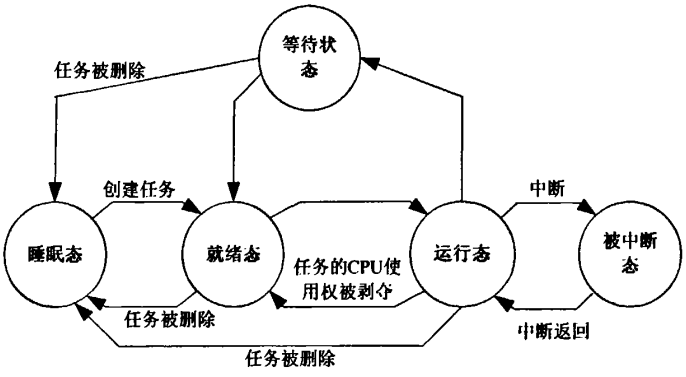

图 3-4 任务的状态

任务控制块

任务控制块是一个数据结构。一旦任务创建，一个任务控制块 OS_TCB 就被赋值。当任务的 CPU 使用权被剥夺时，μC /OS II 用它来保存该任务的状态。当任务重新得到 CPU 使用权时，任务控制块能确保人物从当时被中断的那一点丝毫不差地继续执行。OS_TCB 全部驻留在 RAM 中。读者将会注意到笔者在组织这个数据结构时，考虑到了各成员的逻辑分组。

OS_TCB 数据结构的一些变元是用条件编译语句定义的。如果不需要使用μC /OS II 的全部功能，那么这种定义方法可以使用户缩减μC /OS II 对 RAM 的占有量。

```
typedef strμCt os_tcb {
    OS_STK *OSTCBStkPtr;       //Pointer to current top of stack

#if OS_TASK_CREATE_EXT_EN > 0
    void    *OSTCBExtPtr; //Pointer to user definable data for TCB extension
    OS_STK *OSTCBStkBottom; //Pointer to bottom of stack
    INT32U OSTCBStkSize; // Size of task stack (in number of stack elements)
    INT16U              OSTCBOpt; // Task options as passed by OSTaskCreateExt()
```

```
    INT16U              OSTCBId; //Task ID (0..65535)
#endif


    strμCt os_tcb *OSTCBNext;           //Pointer to next       TCB in the TCB list
    strμCt os_tcb *OSTCBPrev;           //Pointer to previous TCB in the TCB list


#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0) ||
(OS_SEM_EN > 0) || (OS_MUTEX_EN > 0)
    OS_EVENT        *OSTCBEventPtr; //Pointer to event control block
#endif


#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    void *OSTCBMsg; // Message received from OSMboxPost() or OSQPost()
#endif


#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
#if OS_TASK_DEL_EN > 0
    OS_FLAG_NODE   *OSTCBFlagNode;//Pointer to event flag node
#endif
    OS_FLAGS    OSTCBFlagsRdy;    //Event flags that made task ready to run
#endif


    INT16U   OSTCBDly; //Nbr ticks to delay task or, timeout waiting for event
    INT8U    OSTCBStat;    // Task status
    INT8U    OSTCBPrio;   // Task priority (0 == highest, 63 == lowest)
```

```
    INT8U    OSTCBX; //Bit position in group corresponding to task priority (0..7)
    INT8U    OSTCBY; // Index into ready table corresponding to task priority
    INT8U    OSTCBBitX;   //Bit mask to access bit position in ready table
    INT8U    OSTCBBitY; //Bit mask to access bit position in ready group

#if OS_TASK_DEL_EN > 0
    BOOLEAN   OSTCBDelReq;//Indicates whether a task needs to delete itself
#endif
} OS_TCB;
```

各个变量的简单说明如下：

OSTCBStkPtr

指向当前任务堆栈栈顶的指针。μC/OS II 允许每个任务有自己的堆栈，并且每个堆栈的大小可以是任意的。某些商业内核要求其任务堆栈的大小是一致的，除非用户写一个接口函数来改变它。这种不灵活的限制浪费了 RAM 的空间。

OSTCBExtPtr

指向用户定义的任务控制块扩展。用户可以定义自己的任务扩展块，而不必修改μC/OS II 的源代码。该变量只在 OsTaskCreateExt()中使用，使用时需要将 OS_CFG..H 中的 OS_TASK_CREATE_EN 选项设为 1，以允许建立任务函数的扩展。然后就可以将 OSTCBPtr 指向一个包含扩展信息的数据结构了。

OSTCBStkButtom

指向任务堆栈栈底的指针。如果微处理器的堆栈指针是递减的，即栈存储器从高地址向低地址方向分配，则 OSTCBStkButtom 指向任务使用栈空间的最低地址。该变量只在 OsTaskCreateExt()中使用，使用时需要将 OS_CFG..H 中的 OS_TASK_CREATE_EN 选项设为 1，以便使用该项功能。

OSTCBSize

保存栈中可容纳的指针元数目。

OSTCBNext 和 OSTCBPrev

用于任务控制块 OS_TCB 双向链表的前后链接，该链表在时钟节拍函数 OSTimeTick()中使用。OSTimeTick()使用链接各个任务的 OS_TCB 的双向链表，来刷新各任务的任务延迟变量 OSTCBDly。每个任务建立时，OS_TCB 链接到链表中，任务删除时从链表中删除。

OSTCBDly

当需要把任务延迟若干时钟节拍时，或者需要把任务挂起一段时间以等待某事件的发生时，需要用到该变量。这种等待有超时限制。如果该变量为 0，则表示该任务不延时，或者表示等待事件发生的时间没有限制。

OS_TCBStat

任务的状态字。例如当 OS_TCBStat 值为 OS_STAT_READY 时，任务进入就绪态。

OSTCBPrio

任务的优先级。值越小，任务的优先级越高。

OSTCBX，OSTCBY，OSTCBBitX 以及 OSTCBBitY

用于记录一些计算值，这些计算值用来标示任务的优先级或者寻找优先级对应的任务。这些值是在任务建立时算好的，或者是在改变任务优先级时计算的。这些值的算法见下面的程序清单。

```
OSTCBY          = prio >> 3;
```

```
OSTCBBitY      = OSMapTbl[OSTCBY];
OSTCBX         = prio & 0x07;
OSTCBBitX      = OSMapTbl[OSTCBX];
```

应用程序中可以有的最多任务数（OS_MAX_TASKS）已经在文件 OS_CFG.H 中定义了。这个最多任务数也决定了分配给用户程序的任务控制块 OS_TCB 的数目。OS_MAX_TASKS 的数目设置为用户应用程序实际需要的任务数，可减小 RAM 的需求量。所有的任务控制块 OS_TCB 都是放在任务控制快列表数组 OSYCBTbl[]中的。请注意μCOS 分配给系统任务 OS_N_SYS_TASKS 若干个额外的任务控制块，供其内部使用（见文件μCOS_II.H）。目前，一个用于空闲任务，另一个用于统计任务（如果 OS_TASK_STAT 设为 1）。在μC/OS II 初始化时，所有任务控制块 OS_TCB 都被链接成单空任务链表。任务一旦建立，空任务控制块指针 OSTCBFreeList 指向的任务控制块便赋值给了该任务，然后 OSTCBFreeList 的值调整为指向链表的下一个空的任务控制块。一旦任务被删除，任务控制块就还给空任务链表。

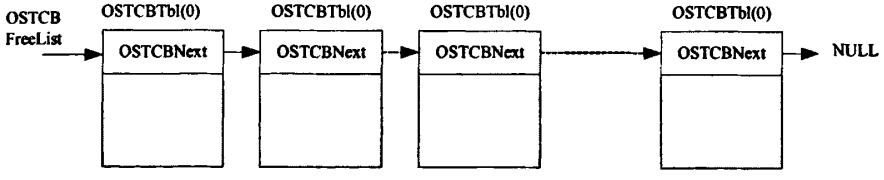

图 3-5 空任务控制块列表(free OS_TCBs)

任务建立时，函数 OS_TCBInit()初始化控制块 OS_TCB。函数 OS_TaskCreate()或函数 OSTaskCreateExt()调用任务控制块初始化函数 OS_TCBInit()。

每个任务被赋予不同的优先级等级，从 0 到最低优先级 OS_LOWEST_PRIO（见文件 OS_CFG.H）。当μC/OS II 初始化时，最低优先级总是被赋予空闲任务 idle_task。

就绪表

每个就绪的任务都可以放在就绪表中，就绪表有两个变量，OSRdyGrp 和 OSRdyTbl[]。在 OSRdyGrp 中，任务按照优先级分组，8 个任务为一组。OS RdyGrp 中的每一位表示 8 组任务中每一组是否有进入就绪态的任务。任务进入就绪态时，就绪表 OSRdyTbl[]中相应的元素的响应位也置为 1。

为了确定下一次该哪个优先级的任务运行了，μC /OS II 总是把最低优先级任务在就绪表中相应字节的相应位置置 1。

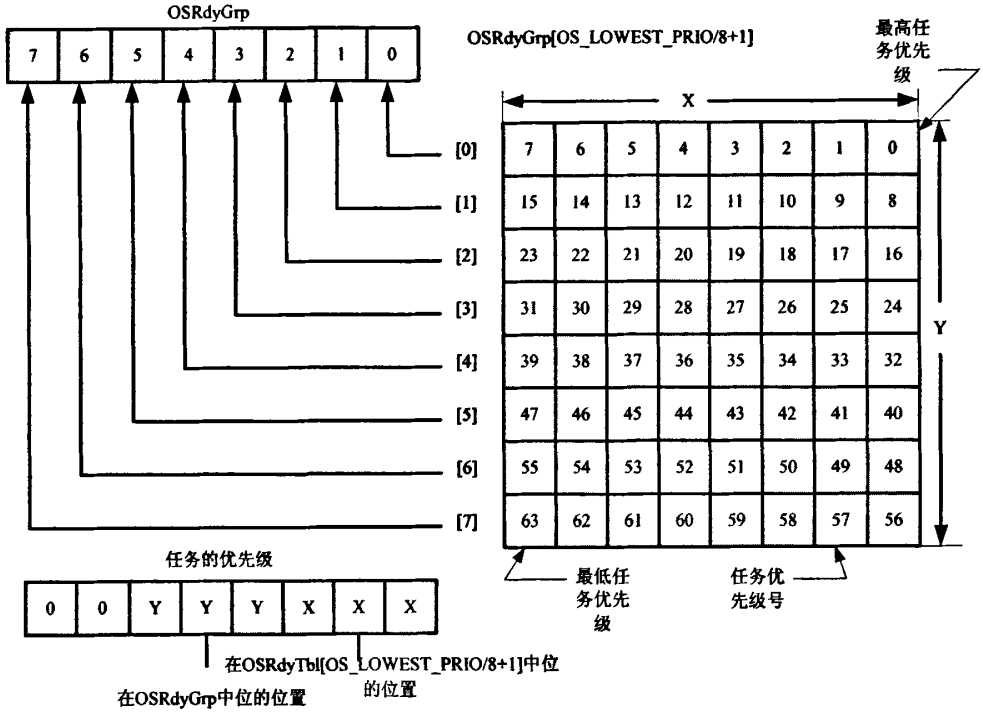

图 3-6 μC /OS II 的任务就绪表

从下图中可以看出任务优先级低 3 位用于确定总就绪表 OSRdyTbl[]中的所在位。接下来的 3 位用于确定是在 OSRdyTbl[]数组的第几个元素。OSMapTbl[]是 ROM 中的屏蔽字，用于限制 OSRdyTbl[]数组的元素下标为 0~7。见下表

下标	位掩码
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

表 3-7 OSMapTbl[]的值

任务调度

μC /OS II 总是进入就绪态中优先级最高的任务。确定哪个任务优先级最高，下面该哪个任务运行了，这一工作是由调度器完成的。任务的调度器是由函数 OSSched()完成的。中断级的调度是由另一个函数 OSIntExt()完成的。这个函数将在以后描述。μC /OS II 任务调度的执行时间是常数，与应用程序建立了多少个任务没有关系。下面的伪代码说明了调度的过程：

```
void  OS_Sched (void)
{
进入临界代码段;
if (所有的中断都已近被处理 & 允许被调度)
{
得到进入就绪态的最高优先级任务;
if (最高优先级任务 != 当前任务)
{
```

```
    进行任务上下文切换;
    }
        }
        退出临界代码段;
    }
```

任务切换

任务切换很简单，有以下 2 步完成：将被挂起的处理器寄存器推入堆栈；然后将较高优先级任务的寄存器值从栈中恢复到寄存器中。在系统中，就绪任务的栈结构总是看起来跟刚刚发生过中断一样，所有处理器的寄存器都保存在栈中。换句话说，系统运行就绪态的任务所要做的一切，只是恢复所有的 CPU 寄存器并运行中断返回指令。为了做任务切换，运行 OS_TASK_SW()，人为模拟了一次中断。多数微处理器由软中断指令或者指令陷阱 TRAP 来实现上述操作。中断服务子程序或陷阱处理 trap handler，也称作事故处理，必须给汇编语言函数 OSCtxSw()提供中断向量。OSCtrSw()除虚 OS_TCBHighRgy 指向即被挂起的任务，还需让当前任务控制块 OSTCBCur 指向即将被挂起的任务。现在只需知道 OS_TASK_SW()挂起了正在运行的任务，而让 CPU 运行更加重要的任务。

OS_Sched()的所有代码都属于临界代码段。在寻找进入就绪态的优先级最高的任务过程中，为防止中断服务程序把一个或者几个任务的就绪位置位，中断是关掉的。为了缩短切换时间,OS_Sched()全部代码都可以用汇编语言编写。为了增加可读性，可移植性及将汇编语言代码最少化，OSSched()都是用 C 语言写的。

调度器确定更重要的任务运行了，就调用 OS_TASK_SW()做任务切换。任务的 context，其实就是 CPU 中的全部寄存器内容，context-swtich 就是任务切换，任务切换代码需恢复该任务在 CPU 使用权被剥夺时保存下来的全部寄存器的值，以便让这个任务能继续运行。

OS_TASK_SW()是宏调用，通常含有微处理器的软中断指令，因为μC/OS II 是靠中断级代码完成的。μC /OS II 需要的就是硬件中断，其行为就像是硬件中断，所以被称为软中断。

4.2.2 μC/OS II 调度任务数目的扩展

本节主要对μC /OS II 支持的最大任务数目进行扩展，从原来的64个扩展到256个。

当系统相应时间很重要时，需要使用抢占式内核。μC /OS II 的实时内核就是抢占式内核，其特点是目前CPU运行的任务永远是进入就绪态的任务中优先级最高的任务，如果在运行过程中有更高优先级的任务进入就绪态，则该任务将取得CPU的使用权。这样，任务的响应时间可以得到最优化。通过仔细设计，能够保证系统处于一个确定的状态。

μC /OS II 设计了64个优先级。由于不支持同优先级的调度，μC /OS II 最多可以运行64个任务。为了增加μC /OS II 内核可以管理任务的数目，需要对μC /OS II 的任务相关的结构和算法进行改进。一个自然而有效的做法是在原有的结构上进行扩展，即将原来6位的优先级定义字节扩展为8位。原有的基于64个任务调度的优先级调度算法分别用3个比特位来定位任务优先级在就绪表重的行和列，即0~2位标识该任务在总就绪表中的列信息，3~5位标识该任务在就绪表中的行信息。因此，存放任务优先级的字节中 8 个比特位只用到了 6 个。而扩充后可以使用闲置的比特位，从而能够区分256个不同任务优先级。扩展后的算法规定任务优先级字节的定义如下图所示。

我们沿用原来的就绪表变量OS RdyGrp和OSRdyTbl[]，仍旧用变量OS RdyGrp来表示优先级在就绪表中所在的行，在OS RdyGrp中，任务按照优先级分组，16个任务为一组。OS RdyGrp的每一位表示某一组中是否有任务进入就绪态，如果有，则相应的位被置1.使用OSRdyTbl[]数组表示优先级在就绪表中的列信息，即存放在每个优先级的任务是否就绪的信息，如果某一位对应的任务处于就绪态，则该位的值置1。变量和OSRdyFrp和OSRdyTbl[]之间的关系如下图所示。（图中OSRdyGrp下标中标注的数字0~15仅表示为清楚起见表示16组任务，并非表示OSRdyGrp中每一位的状态信息，同理，OSRdyTbl[]下表格中的数字0~255也仅表示256个任务，并非实际存放的状态信息）。

源代码在原来代码的基础上，进行了改进。其中把任务放入就绪表的程序代码是：

```
OSRdyGrp                    |= OSRdyTbl[ prio >> 4 ];
OSRdyTbl[ prio >> 4 ]       |=  OSMapTbl[ prio & 0X0F ];
```

其中，char OSMapTbl[] 是 ROM 中的一张静态表，用于限制 OSRdyTbl[]数组下标为 0~16。其值为 OSMapTbl[] = {0X01, 0X02, 0X04, 0X08, 0X10, 0X20, 0X40, 0X80, 0X0100, 0X0200, 0X0400, 0X0800, 0X1000, 0X2000, 0X4000, 0X8000}。

从就绪表中删除优先级为 prio 的任务也比较简单，只要清除 OSRdyGrp 和 OSRdyTbl[]中的对应该任务的位即可。

```
if((OSMapTbl[ prio >> 4] &= ~ OSMapTbl[prio & 0X0F]) == 0)
    OSRdyGrp &= ~ OSMapTbl[ prio >> 4 ];
```

对于 OSRdyGrp 只有当被删除任务所在的任务组中劝阻任务一个都没有进入就绪态时，才将相应位清零。也就是说，OSRdyTbl[prio >>]所有位都为 0 时，OSRdyGrp 的相应位才清 0。

在μC/OS 中，采用基于优先级的可抢占式调度，系统为每一个任务分配一个优先级，调度程序总是选择优先级最高的就绪任务运行。内核运行中将频繁地任务调度，并且任务调度属于系统的临界资源，调度时需要独占 CPU，不允许外部中断和任务切换，所以调度的速度会影响整个系统的相应速度和处理能力。因此，如何快速查找最高优先级就绪任务就成了整个算法的核心问题之一。

μC/OS 的原有算法使用了查表的方法，使得查找优先级的时间为一个常数。这里我们提出的改进算法中，仍然希望这个过程所用的时间仍然是一个常量。这样就可以保证系统的实时性。

原有的 OSUnMapTbl[]表为 16X16 的数组，每个数组元素的值是固定的。当使用该表格时，数组下标转换成为相应的八位二进制，其中高四位对应数组的行，

第四位对应数组的列（假设最右边的一位是第 0 位 bit0），由此可以查表判断一组任务就绪态任务中优先级最高的那个任务所在的位置，再经过计算可以得到最高优先级就绪任务的优先级。

类似于原有的 OSUnMapTbl[]表（优先级判定表格），我们可以建立一个扩展的表格 OSUnMapTbl[]。把改进后的优先级就绪表看做一个类似二维坐标的平面，使 256 个 优先级分布在 4 个象限中，同时使用了两个变量 ox, oy 来确定任务的优先级所在的象限，并由此进一步查找最高优先级的就绪任务。

变量 ox 和 oy 的取值如下图所示，下面的查找最高优先级算法首先从判定优先级所在的象限开始进行。具体代码如下：

```
oy = OSRdyGrp & 0X00FF;
if(oy == 0)
 y = OSUnMapTbl[ OSRdyGrp >> 8] + 8;
else
 y = OSUnMapTbl[ oy ];
ox = OSRdyTbl[y] & 0X00FF;
if(ox == 0)
 x = OSUnMapTbl[   OSRdyTbl[y] >> 8] + 8;
else
 x = OSUnMapTbl[ ox ];
prio = (y << 4 ) + x;
```

例如，如果 OSRdyGrp 的值为 0X0068，则利用首先可以判断 oy! = 0，查优先级判定表格得到 OSUnMapTbl[oy]的值为 3，假设 OSRdyTbl[3]的值为 0X00E4，则可以判定 ox != 0，再查优先级判定表格得到 OSUnMapTbl[ox]的值为 2，最后计算得到任务的优先级 prio 为 50。

4.2.3 任务调度算法概念

如何使任务集内各任务满足各自的时限，使系统得以正常、高效率工作的任务调度算法一直是实时系统领域内研究的焦点。根据其应用领域及追求精简、高

效角度的不同，任务调度算法从简单的合理安排任务循环，发展到基于优先级的速率单调调度(RMS)、截止时间单调调度 (DMS)、最早时限优先(EDF)、最短空闲时间优先(LLF)等算法。任务调度算法的好坏以及执行效率直接关系到嵌入式内核的应用范围及实时性程度。

静态调度

静态调度是在系统开始运行前进行调度的，严格的静态调度在系统运行时无法对任务进行重新调度。 静态调度的目标是把任务分配到各个处理机，并对每一处理机给出所要运行任务的静态运行顺序。 静态调度算法实现简单，调度的额外开销小，在系统超载时可预测性好。 但也具有很大的局限性，例如资源利用率低、受系统支持的优先级个数限制以及灵活性和自适应性差等。下面介绍两种常见的静态调度算法。

（1）速率单调调度

RMS 算法于 1973 年由 C.L.Liu 和 J.Layland 提出的，RMS 算法的核心思想是根据任务的周期来设置任务的优先级，周期越小，其优先级越高，并以单调的顺序对剩余的任务分配优先级。

RMS 做了一系列假设：

1) 所有任务都是周期性的，且周期大于或者等于时限；
2) 所有任务必须在其时限到来前结束；
3) 所有实时任务相互独立且均有恒定的运行时间；
4) CPU 必须总是执行优先级最高且处于就绪态的任务，即须用可剥夺型调度法。

任务按周期由小到大排列为 $T_1 \le T_2 \le ... \le T_n$。EDF 以任务的时限与当前时刻的距离确定任务优先级，距离越近，优先级越高。因此，EDF 总是选择当前最迫切需要完成的任务获得处理器。Liu 和 Layland 的理论证明了下列公式，即用 RMS 调度的独立的周期性任务总能满足其截止时间(deadline)的要求。

定理： 对于由 n 个周期组成的实时任务集，当且仅当：

$$(\frac{C_1}{T_1})+(\frac{C_2}{T_2})+...+(\frac{C_n}{T_n})\leq 1$$

该任务集能够由 EDF 调度。其中 C 为最坏执行时间。

已经证明：对于在 RMS 单处理器上调度独立、可抢占的周期任务，是 RMS 最佳的静态优先级算法。算法的优点是开销小，灵活性好，可调度性测试简单。但在某些情况下，最高执行率的任务并非是最重要的任务。

（2） 截止时间单调调度

DMS 是在速率单调调度的基础上发展起来的，不同的是任务的优先级按截止时间来分配。截止时间短的任务优先级高，截止时间长的任务优先级低。截止时间调度具有与速率单调算法相同的优点，但 DMS 算法放松了对任务的周期必须等于其截止时间的限制。在任务的截止时间小于或等于其周期的情况下，DMS 已被证明是静态最优的调度算法。

动态调度

在嵌入式实时系统中，动态调度依赖于任务的优先级。 优先级可以静态分配或者依据不同的特征参数，如截止时间、空闲时间或关键性（即任务的重要程度或者价值）等进行动态分配。动态调度可以是抢占式的或非抢占式的。 当检查到一事件时，动态抢占式算法立即决定是运行与此事件相关的任务，或继续执行当前的任务；对于动态非抢占式算法，它仅仅知道有另一个任务可以运行，在当前任务结束后，它才在就绪的任务中选择一个来运行。以下介绍的是两个经典的动态调度算法：最早截止时间优先算法和最小松弛时间算法。

（1） 最早截止时间优先算法

抢占式 EDF 调度算法是一个动态优先级驱动的调度算法，其中分配给每个任务的优先级根据它们当前对最终期限的要求而定。当前请求的最终期限最近的任务具有最高的优先级，而请求最终期限最远的任务被分配最低优先级。这个算法

能够保证在出现某个任务的最终期限不能满足之前，不存在处理器的空闲时间。抢占式 EDF 调度算法最大的优势在于，对于任何给定的任务集，只要处理器的利用率不超过 100%，能够保证它的可调度性。EDF 算法在系统的负载相对较低时非常有效，EDF 的缺点在于不能解决过载问题。在系统负载较重时，系统性能急剧下降，引起大量任务错过截止期，甚至可能导致 CPU 时间大量花费在调度上，在这时系统的性能还不如 FIFO(First In First Out)方法。

（2）最小松弛时间算法

LSF 算法优先级的分配基于每个任务的有效松弛时间，一个任务的松弛时间被定义为在其错过截止期之前能够承受的最大时间延迟，这个算法给具有最小松弛时间的任务分配最高优先级，以此来保证紧急任务的优先执行。但是这个算法会导致调度开销没有上界，并且当两个或多个任务具有相似的松弛时间时调度情况会变坏。由于等待任务的松弛时间是严格递减的，其等待执行的缓急程度也随时间越来越紧迫，因此在系统执行过程中，等待任务随时可能会抢占当前执行的任务。LSF 算法造成任务之间的频繁切换现象较为严重，增大了系统开销并限制了 LSF 算法的应用。

4.2.4 针对 μC/OS II 的任务调度算法优化

嵌入式实时系统中资源是非常有限的，所以开销要尽可能小。开销主要包括运行开销和调度开销。运行开销与队列分析和从调度队列中增加、删除任务相关。每个任务在一个调度周期内至少被阻塞和唤醒一次，所以任务调度器在一个周期内不得不对一个任务进行两次选择。RMS 算法根据任务的执行频率设置优先级，有较小的运行开销，但执行频率最高的任务不一定是最重要的。

EDF 算法中则是对整个任务列表的调度开销进行全面比较，选择最高优先级任务进行调度，较小的调度开销，但对多个任务具有同一优先级的情况考虑不足。

通过上述分析，如果能够混合 RMS 和 EDF 的优点，采用 RMS 较小的运行开销和 EDF 较小的调度开销，得到一个优先级分配的优化算法，在多个任务要求调度时选择截止时间最短(即优先级最高) 的任务运行，若任务的优先级相同时则选择运行频率最高的任务运行。

结合源码公开的嵌入式实时操作系统μc/os-II 为例，μc/os-II 可以管理 64 个任务，保留了其中 8 个系统，所以应用程序最多可以有 56 个任务，采用抢占式的优先级调度策略，绝大多数服务的执行时间具有确定性，要求赋予每个任务的优先级必须是不相同的，这样的限制简化了任务管理系统，但也给大型应用带来了限制。如果采用上述改进的优级分配的优化算法，则突破了原系统对任务数量的限制，并去除对每个任务必须有不同优先级的要求。算法改进的关键之处是对优先级相同的任务建立散列表，并在散列表中按照任务执行频率进行降序排列，表中的第一个任务就是在该优先级上执行频率最高的任务，这要求在一个优先级上可以建立多个任务。在对某优先级任务执行调度的时候，首先看该优先级上是否有其他任务，如果有，则找到执行频率最高的进行调度。改进后的任务创建和任务调度的伪码如下 。

建立任务：

```
建立任务控制块;
建立任务堆栈 OSTaskStkInit ()  ;
if (在该优先级上已有任务存在)
    {
        比较任务的执行频率;
        将该 TCB 按降序插入到散列表中  ;
     }
 else
    {
    新任务是优先级散列表上的第一个任务;
    }
```

任务调度：

```
if (在该优先级上有任务存在)
```

```
{
if(任务数大于 1)
        {
    在该优先级散列表中找到执行频率最高的任务进行调度;
    }
else
    {  执行任务调度; }
  }
else
   {返回错误，任务不存在; }
```

作者已经在源代码上进行了改进，使得μC/OS II 使用优级分配的优化算法。比较该算法和μC/OS II 原有的简单的实时优先级抢占算法，原有的算法优点是开销小，灵活性好，可调度性测试简单，而改进后的优先级算法比较复杂，CPU 的一部分资源将被用于调度器上，但是能保证 CPU 较早的运行最重要，最紧急的任务。

第四章 μC/OS-II 系统的任务堆栈的讨论

μC/OS II 内核的另外一个值得改进的地方就是其任务栈管理方法。在 μC/OS II 内核中，各个不同的任务使用独立的堆栈空间，堆栈的大小按每个任务所需要的最大堆栈深度来定义，这种方法可能会造成堆栈空间的浪费。下面讨论如何在 RTOS 中多个任务共用一段连续存储空间作为堆栈。

4.1 μC/OS II 对任务栈的处理方法与缺陷

4.1.1 任务切换需要保存的数据

对于抢先式 RTOS 来说，在任务切换时，应保存当前任务的各种现场数据。现场数据包括局部变量、各个 CPU 寄存器、堆栈指针和程序被中止的任务指针。CPU 寄存器是任何任务代码均会用到的；而局部变量，一般的编译器是将其它安排在堆栈空间中，堆栈指针也是各任务公用，所以也需要保存。对于全局变量，由于一般是在内存中的固定位置，各任务所占用的空间完全独立，不需要保存。

在 X86 环境中，要保存的 CPU 寄存器共 14 个 16 位寄存器；通用寄存器 8 个（AX、BX、CX、DX、SP、BP、SI、BI）、段寄存器 4 个（CS、DS、ES、SS）以及指令指针 IP 和标志寄存器 FR 各 1 个。

4.1.2 C 编译器中变量在堆栈中的位置

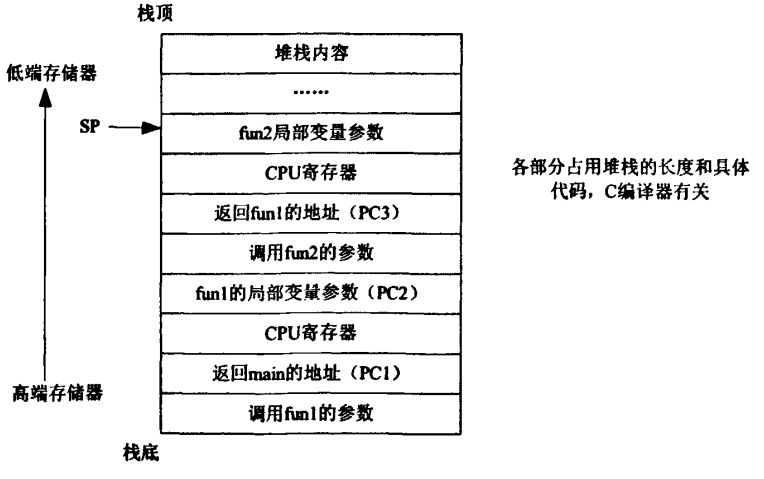

图 4-1 函数调用嵌套时的堆栈情况

假设函数 main（）调用 funl()，而 funl（）调用 fun2（），则在执行 fun2() 中的代码时，堆栈映像如图 4-1 所示（X86 CPU 的情况）。

对于一个存在函数调用嵌套的 C 程序来说，大部分编译器将传递的参数和函数本身的局部变量放在了堆栈中，编译器会自动生成压栈（push）和弹栈（pop）代码，以保存上级函数的运行寄存器。

对于 RTOS 软件，堆栈中的各种数据就是一个任务的作现场。一般 CPU 的堆栈指针 SP 只有一个，在进行任务切换时，必须将挂起任务所使用的堆栈内容保存起来，以便使该任务在下次唤醒时能从原地继续运行。

4.1.3 μC/OS-II 对堆栈的定义和使用

μC/OS-II 为了保存任务堆栈中的数据，对每个任务定义一个数组变量作为堆栈，在任务切换时，将 CPU 堆栈指针 SP 指向该数组中的某个元素，即栈顶。

比如，在其 ex21.c 文件中定义的任务堆栈语句为：

```
OS_STK TaskStartStk[TASK_STK_SIZE]; /*启动任务堆栈*/
OS_STK TaskClkStk[TASK_STK_SIZE]; /*时钟任务堆栈*/
OS_STK TasklStk[TASK_STK_SIZE]; /*任务 1#，任务堆栈*/
```

以上各任务堆栈数组变量在初始化函数 OSTCBInit()中被会给了任 务控制块 OS_TCB 的 OSTCBStkPtr 变量。在任务切换时，μC/OS-II 调用 OSCtxSw 汇编过程（OS_CPU_A.ASM 文件），将 CPU 的 SP 指针指向该变量，从而使每个任务使用独立的任务堆栈。

```
LES BX，DWORD PTR DS：_OSTCBCur
；保存挂起任务的堆栈指针 SP
MOV ES：[BX+2]，SS
MOV ES：[BX+0]，SP
……
LESB X，DWORD PTR DS：_OSTCBHighRdy
```

```
；切换 SP 到要运行任务的堆栈空间
MOV SS，ES：[BX+2]
MOV SP，ES:[BX]
……
```

在代码中，变量 OSTCBHighRdy（OSTCBCur）和堆栈指针变量 OSTCBStkPtr 的数值是同同的，因为 OSTCBStkPtr 是结构 OSTCBHighRdy 的第一个变量。

这种任务栈处理方法的缺点是可能造成空间的浪费。因为一个任务如果堆栈满了，该任务也就无法运行，即使其它任务的堆栈还有空间可用。当然，这种方法的好处是任务栈切换的时间非常短，只需要几条指令。

4.2 对 μC/OS II 任务栈的改进——共用堆栈的处理方法

4.2.1 栈共用连续存储空间

如果多个任务使用同一段连续空间作为堆栈，这样各个堆栈之间就可以互补使用。在前面说过，共用空间的问题在于一个任务运行时不能破坏其它任务的堆栈数据。为简单起见，先看图 3 所示两个任务的情况。

假定任务 1 首次运行时任务栈为空。运行一段时间后任务 2 运行，堆栈空间继续往上生长。这次任务切换不需要修改 CPU 的 SP 数值，但需要记下任务 1 的栈顶位置 SP1（图 3 中）。

在任务 2 运行一段时间后，RTOS 又切换到任务 1 运行。在切换时，不能简单地将 SP 指针修改回 SP1 的数值，因为这样堆栈向上生长时会破坏任务 2 堆栈中的数据。办法是将原来任 1 务堆栈保存的数据移动到靠栈顶的位置，而将任务 2 堆栈数据下移到靠栈底的位置，堆栈指针 SP 实际上不需要修改（图 3 右）。

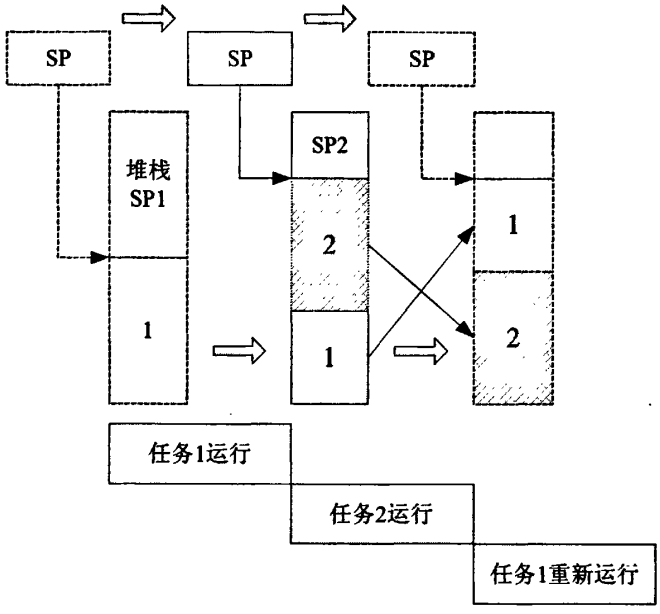

图 4-2 共用栈任务切换时堆栈的移动情况

考虑到更为一般的情况，有 N 个任务，当前运行的任务为 k，下一个运行的任务为 j，在共用任务堆栈时必须做的工作有：

*为每个任务定义栈顶和栈底 2 个堆栈指针；

*在任务切换时，将待运行任务 j 的堆栈内容移动到靠栈顶位置，同时将其堆栈上方的任务堆栈下移，修改被移动推栈的任务堆栈指针。

假设我们定义的任务栈空间和任务的栈指针变量为：

```
void TaskSTK[MAX_STK_LEN];/*任务堆栈空间*/
typedef strμCt TaskSTKPoint{
int TaskID;
int pTopSTK;
int pBottomSTK;
}TASK_STK_POINT;
TASK_STK_POINT pTaskSTK[MAX_TASK_NUM];
/*存放每个任务的栈顶和栈底指针*/
```

任务栈指针数组 pTaskSTK 的元素个数同任务个数。为了堆栈交换，需要另

外一块临时存储空间，其大小可按单个任务栈最大长度定义，用于中转堆栈交换的内容。堆栈内容交换的伪 C 算法可写为：

```
StkEechange(int CurTaskID,int RunTaskID)
{ /*2 个参数为当前运行任务号和下一运行任务号*/
void TempSTK[MAX_PER_STK_LEN]; /*注意该变量长度可小于 TaskSTK*/
L=任务 RunTaskTD 的堆栈长度;
①将 TaskSTK 顶部的 L 字节移动到 TempSTK 中;
②将 RunTaskID 任务的堆栈内容移动到 TaskSTK 顶部;
③将 RunTaskID 堆栈上方（移动前位置）所有内容下移 L 个字节;
④修改 RunTask 堆栈上方(移动前位置)所有任务栈顶和栈底指针(pTaskSTK
变量）;
};
```

该算法的平均时间复杂度可计算如下：

O（T）=SL/2+SL/2+SL×N/2

式中，第一、二项为步骤①和步骤②时间，第三项为步骤③时间；SL 表示每个任堆栈的最大长度（即 MAX_PER_STK_LEN），N 表示任务数。

取 SL 为 64 字节，任务数为 16 个，则数据项平均移动次数为 576。假设每次移动指令时间为 2μs，则一次任务栈移动时间长达约 1ms。所以在使用该方法时，为了执行时间尽量短，编码时应仔细推敲。

从空间上说，共用任务栈比独立任务栈优越。假设独立任务栈方法中每个堆栈空间为 K，任务数为 N，则独立任务栈方式的堆栈总空间为 N×K。在共用任务栈时，考虑各任务互补的情况，TaskSTK 变量不需要定义为 N×K 长度，可能定义为二分之一或者更小就可以了。

另外，这种方法不需要在任务切换时修改 CPU 的 SP 指针。

4.2.2 工作栈和任务堆栈

上节共用任务栈算法的缺点是：任务切换时的堆栈内容交换算法复杂，占用时间长。另外一个折中的方法是设计一个工作堆栈，用于给当前运行的任务使用；在任务切换时，将工作栈内容换出得另外的存储空间，该空间可以动态申请，其

大小按实际需要即可。

这种方法看起来和独立任务栈的方法类似，需要 N+1 块存储空间，其中一块用于工作栈空间。和独立任务堆栈相比，其区别有 2 点：

①SP 指针所指向的空间始终是同一块存储空间，即工作栈；

②每个任务栈的大小不需要按最大空间定义，可以动态按实际大小从内存中分配空间。

对于 8031 这种处理器结构，由于堆栈指针只能指向其内部存储器，大小十分有限。采取这种方法，可将工作栈设在内部 RAM，将任务栈设在外部 RAM，扩展了堆栈空间。

和上一种共用堆栈方法相比，这种方法的交换时间要短，其时间复杂度约为 1.5 倍最大任务栈长度。

4.2.3 小结

独立任务栈的方法适合于存储器充足、任务切换频繁、对任务切换时间要求较 高的场合，一般主要用在 16 位或者 32 位微处理器平台环境。值得注意的是，在某些微处理器中，虽然可使用的数据存储器可以设计得较大，但堆栈所能使用的存 储器却是有限的。比如 8031 系列存储器，堆栈只能使用内部的 128 字节数据存储器，即使系统中有 64K 字节的外部数据存储器，任务栈的总空间也不能超过 128 字节。这种处理器使用共用任务栈结构的 RTOS 就更好一些。

由于共用任务栈系统需要较长的任务切换时间，不适于任务切换频繁的场合，在很多嵌入式系统中，长时间只有几个任务会处于运行状态，其它任务在特定的条件下才会运行。对于 RTOS 的使用者，也可以适当地划分任务，来减小任务切换的时间。

无论使用哪种方法，在存储空间有限时，任务栈的长度应仔细计算。计算的根据是任务中的函数嵌套数、函数局部变量长度。对于共用任务栈，还要考虑同时运行态和挂起态的最大任务数。一些编译器可以生成堆栈溢出检查代码，在调试时可将该编译开关打开，以测试需要的实际堆栈长度。

第五章 实时操作系统的性能测试

RTOS 的应用中，对于其评价可以从很多角度来进行，如体系结构、API 的丰富程度、网络支持、可靠性等。其中，实时性是 RTOS 评价的最重要的指标之一，实时性的优劣是用户选择操作系统的一个重要参考。评价一个操作系统的实时性应该着重考察它的哪些指标，以及如何进行测试，是本章着重讨论的问题。

5.1 Thread-Metric 简介

Thread-Metric 是一个开源且免费的测试套件，同时 Thread-Metric 还提供了 ThreadX 的测试结果供使用者进行比较参考。ThreadX 本身是一个非常优秀的商业化实时内核，在行业里有着许多非常成功的应用，通过与 ThreadX 测试结果的比较，我们可以对自己所测试的 RTOS 有个更加直观的了解。

5.1.1 Thread-Metric 的测试原理

整个 Thread-Metric 测试套件由几个独立的测试项目组成，每个项目分别用于测试实时内核中的某一基本功能（如任务切换、中断处理、信号量处理等等）。测试的基本原理是通过计算一定周期时间长度里内核反复处理某一事务的次数，并将结果通过“printf”函数输出给 PC 终端获取。

Thread-Metric 中的第一个测试项目为“基准测试（Basic Processing Test）”，该测试用于获取一个称之为“校准值（Calibration）”的数据，校准值的大小反映的是测试中所使用的硬件平台的能力，它的引入是为了屏蔽硬件平台对测试结果的影响，因为我们所需要评估的是 RTOS 的性能，而并非整个系统的性能。

除第一个测试项目（基准测试）外，在其它测试项目中，我们将会获取到一个称之为“迭代值（Iteration）”数据，迭代值的就是在一个测试周期长度里内核所处理的这一事务的次数，于是我们使用公式：

$$\text{得分} = \text{迭代值} \div \text{校准值}$$

即可得到实时内核在这一测试项目的得分。

5.1.2 Thread-Metric 的文件结构

Thread-Metric 测试套件全部由 C 语言编写，因此它适用于绝大部分实时内核，使用 Thread-Metric 也需要一个移植过程，不过 Thread-Metric 的移植非常简单，其移植过程只是一些 API 的重映射操作。Thread-Metric 测试套件的源文件的组成如表 4-1 所示：

文件名	功能描述
tm_api.h	API 声明和宏定义常量
tm_basic_processing_test.c	基准测试
tm_cooperative_scheduling_test.c	协同式的任务调度测试
tm_preemptive_scheduling_test.c	抢占式的任务调度测试
tm_interrupt_processing_test.c	中断处理测试
tm_interrupt_preemption_processing_test.c	中断当中的任务抢占处理测试
tm_synchronization_processing_test.c	任务同步处理测试
tm_message_processing_test.c	消息处理测试
tm_memory_allocation_test.c	内存分配测试
tm_porting_layer.c	Thread-Metric 移植相关文件
tm_porting_layer_threadx.c	Thread-Metric 移植于 ThreadX 内核的参考实例

表 4-1 Thread-Metric 中的文件

文件“tm_porting_layer_threadx.c”是 Thread-Metric 提供的一个已经完成的基于 ThreadX 的移植文件，它只是用来帮助我们快速的将 Thread-Metric 移植到其它实时内核，在我们实际利用 Thread-Metric 测试其它 RTOS 时，它是不需要使用的。

5.1.3 Thread-Metric 的使用要求

在使用 Thread-Metric 测试套件时，为了得到一个客观公正的测试结果，我们应当遵循 Thread-Metric 建议的几点要求，

一、测试周期长度应至少大于 30 秒。越大的测试周期长度越有利于消除调用“printf”函数输出测试结果时对于测试结果本身的影响；

二、关闭所有编译器优化选项，不允许将代码缓存在处理器的任何高速 Cache 中运行；

三、在移植 Thread-Metric 的过程中，API 的重映射不能采用宏定义的方式；

四、内核的时钟节拍周期应设置为 10 毫秒；

5.2 Thread-Metric 中的测试项目

当前版本的 Thread-Metric 总共包含 8 个测试项目，这些测试项目基本覆盖了实时操作系统最重要的核心功能。

测试 1：基准测试（Basic Processing Test）

测试 1 的主要目的就是获取硬件平台的性能校准值，校准值越大说明硬件平台的性能越强。在这个测试中将只创建一个运行任务。

测试 2：协同式的任务调度测试（Cooperative Scheduling Test）

该测试中包含 5 个相同优先级的任务，各个任务在在执行过程中会先将自己的计数器加 1，然后通过调有“relinquish”函数主动将 CPU 使用权交给下一个任务。图 4-1 是测试 2 的运行示意图：

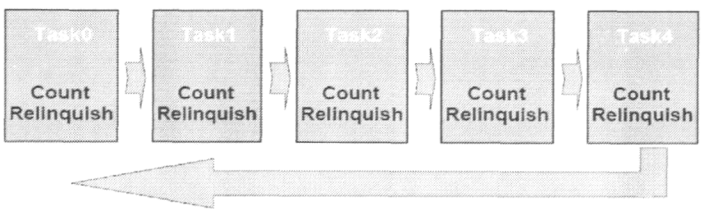

图 4-1　协同式的任务调度测试

测试 3：抢占式的任务调度测试（Preemptive Scheduling Test）

该测试中包含 5 个由高到低不同优先级的任务，各个任务在执行过程中会将自己的计数器加 1。在测试开始时，只有优先级最低的任务处于就绪，其它任务都被挂起。优先级最低的任务先唤醒优先级次低的任务被抢占，这样依次抢占下去后，最高优先级的任务获的 CPU 使用权后又将自己挂起，次高优先级的任务也将自己挂起，到最后优先级最低任务又获得 CPU 使用权，一个新的循环又开始。图 4-2 是测试 3 的运行示意图：

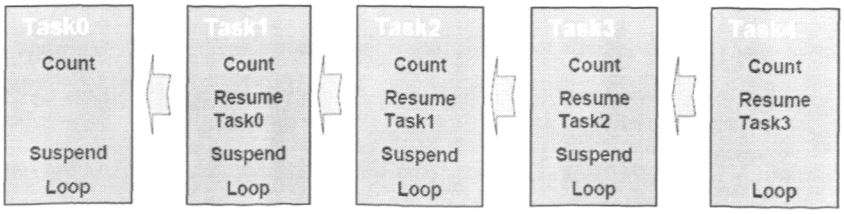

图 4-2　抢占式的任务调度测试

测试 4：中断处理测试（Interrupt Processing Test）

该测试中只包含 1 个任务，该任务通过调用软中断（SWI）指令的方式来连续模拟中断的发生，中断服务程序会释放一个信号量，中断返回后，任务去获取该信号量。获取成功后再次调用软中断。图 4-3 是测试 4 的运行示意图：

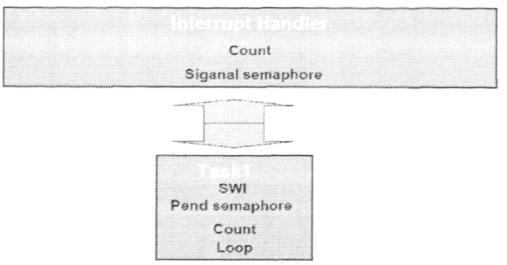

图 4-3　中断处理测试

测试 5：中断当中的任务抢占处理测试（Interrupt Preemption Processing Test）

该测试中包含 2 个优先级不同的任务，低优先级的任务通过调用软中断（SWI）指令的方式来模拟中断，中断服务程序中另外一个高优先级的任务被唤醒，中断返回时发生任务抢占。图 4-4 是测试 5 的运行示意图：

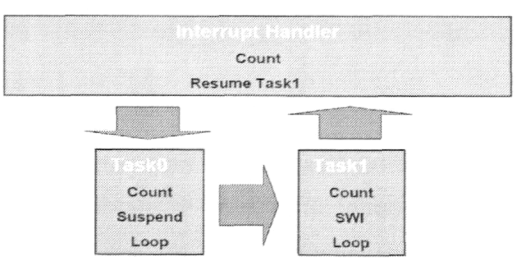

图 4-4　中断当中的任务抢占处理测试

测试 6：消息处理测试（Message Processing Test）

该测试包含 1 个任务，任务先想邮箱中发送一条消息，然后紧接着又再去邮箱中获取，并将获取的消息与发送的做对比，图 4-5 是测试 6 的运行示意图：

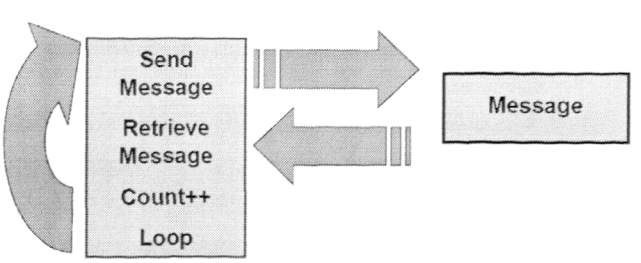

图 4-5　消息处理测试

测试 7：任务同步处理测试（Synchronization Processing Test）

该测试包含 1 个任务，任务通过不断获取和释放信号量的操作来模拟信号量的任务同步功能，图 4-6 是测试 7 的运行示意图：

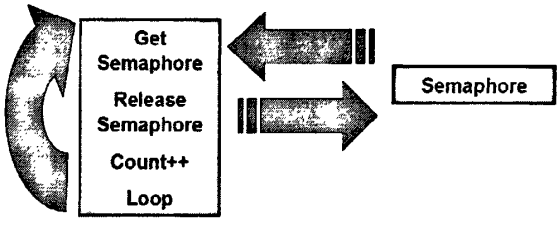

图 4-6　任务同步处理测试

测试 8：内存分配测试（Memory Allocation Test）

该测试包含 1 个任务，任务通过不断获取和释放一个内存块来的测试内核的内存管理功能。图 4-7 是测试 8 的运行示意图：

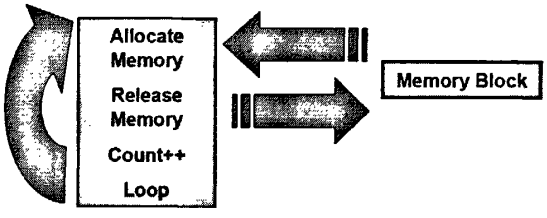

图 4-7　内存分配测试

5.3 测试步骤与方法

在开始使用 Thread-Metric 进行测试之前，以下几个预备条件应当确保已经满足：

◆ 欲测试的实时内核已经成功移植到某硬件平台

◆ 硬件平台的核心处理器支持软中断功能

ARM 处理器具有专门的软中断指令（SWI 指令）

◆ 欲测试的实时内核具有将任务延迟特定时间长度的能力

μC /OS II 具有任务延迟 API（函数 slp_tsk()）

◆ 系统可以使用“printf”函数将测试结果输出

printf 函数的输出在 ARM7TDMI 评估板上可以串口输出到 PC 终端

5.3.1 如何移植 Thread-Metric

Thread-Metric 的移植非常简单，其移植过程就是将文件“tm_porting_layer.c”中已经定义好的 API 进行重影射，对于 μC/OS II 来说，其映射关系如下：

(1) int tm_thread_create()

Thread-Metric 使用此 API 创建一个任务，使用 μC/OS II 的 OSTaskCreate()进行映射。

(2) int tm_thread_resume()

Thread-Metric 使用此 API 恢复一个任务，使用 μC/OS II 的 OSTaskResume()进行映射。

(3) int tm_thread_suspend()

Thread-Metric 使用此 API 挂起一个任务，使用 μC/OS II 的 OSTaskSuspend()进行映射。

(4) void tm_thread_relinquish()

Thread-Metric 使用此 API 让任务主动交出 CPU 使用权给同优先级的其它任务，μC/OS II 不支持同优先级调度，所以没有对应的函数与之映射。

(5) void tm_thread_sleep()

Thread-Metric 使用此 API 让任务延迟，使用 μC/OS II 的 OSTimeDly()进行映射。

(6) int tm_queue_create()

Thread-Metric 使用此 API 创建消息队列，使用 μC/OS II 的 OSQCreate()进行映射。

(7) int tm_queue_send()

Thread-Metric 使用此 API 发送消息，使用 μC/OS II 的 OSQPost()进行映射。

(8) int tm_queue_receive()

Thread-Metric 使用此 API 接收消息，使用 μC/OS II 的 OSQAccept()进行映

射。

(9) int tm_semaphore_create()

Thread-Metric 使用此 API 创建信号量，使用μC /OS II 的 OSSemCreate()进行映射

(10) int tm_semaphore_get()

Thread-Metric 使用此 API 获取信号量，使用μC /OS II 的 OSSemPend()进行映射

(11) int tm_semaphore_put()

Thread-Metric 使用此 API 释放信号量，使用μC /OS II 的 OSSemPost()进行映射

此外，Thread-Metric 中还有几个与内存管理相关的 API，由于μC /OS II 没有相关功能，在这里我们无需对相关 API 进行映射。

5.4 针对μC /OS II 任务调度算法的性能测试

5.4.1 μC /OS II 优先级扩充后的测试结果

表 5-2 列出了此次使用 Thread-Metric 测试μC /OS II 的结果，同时 ThreadX 的结果也放在了表中作为比较。

测试项目	μC /OS II		ThreadX®
测试 1 --基准测试	校准值= 5928		/
	迭代值	得分	得分
测试 3 --协同式的任务调度测试	/	/	140
测试 3 --抢占式的任务调度测试	393026	66.3	55
测试 4 --中断处理测试	494988	83.5	84
测试 5 --中断当中的任务抢占处理测试	262017	44.2	36

测试 6 --消息处理测试	465940	78.6	94
测试 7 --任务同步处理测试	759376	128.1	177
测试 8 --内存分配测试	564345	95.2	159

表 4-2 基于 Thread-Metric 的 μC /OS II 测试

表中所显示的μC /OS II 的测试结果比较令人满意，在有些项目的得分甚至超过了 ThreadX。

在测试 3 中，因为原版的μC /OS II 不支持同优先级的任务调度，所以该项没有计分。

表 5-3 列出了此次使用 Thread-Metric 测试μC /OS II 经过任务扩展后的结果，同时μC /OS II 原版测试的结果也进行了对比。

测试项目	μC /OS II		μC /OS II@
测试 1 --基准测试	校准值= 5928		/
	迭代值	得分	得分
测试 3 --协同式的任务调度测试	/	/	/
测试 3 --抢占式的任务调度测试	350937	59.2	66.3
测试 4 --中断处理测试	427409	72.1	83.5
测试 5 --中断当中的任务抢占处理测试	235342	39.7	44.2
测试 6 --消息处理测试	451121	69.1	78.6
测试 7 --任务同步处理测试	684091	115.4	128.1
测试 8 --内存分配测试	564345	94.5	95.2

表 4-2 基于 Thread-Metric 的 μC /OS II 测试

表中的数据指出对μC /OS II 进行任务扩展后，任务调度的时候数值有所下降（测试 3，4），这是因为在算法上改进后的算法需要处理 4 位的运算，比原

有的 3 位运算的时间有所增加。如果能在汇编层面上对于位运算进行一下优化，应该可以得到更好的结果。测试 6 和测试 7 的数值也有所下降，因为在进行消息处理必须用到的事件就绪表中的数据结构也进行了扩充。

5.4.1 μC/OS II 采用新的调度算法后的结果

在前面的算法分析中，已经指出改进的算法和μC/OS II 原有的简单的实时优先级抢占算法相比，原有的算法优点是开销小，灵活性好，可调度性测试简单，而改进后的优先级算法比较复杂，CPU 的一部分资源将被用于调度器上，但是能保证 CPU 较早的运行最重要，最紧急的任务。所以需要较好的设计测试的数据，才能比较全面的对两种算法的效果进行对比。

Thread-Metric 的测试项目中测试 3：抢占式的任务调度测试（Cooperative Scheduling Test），可以较好的反应调度算法的速度，因此，我们以该项测试作为基础。将测试项中的任务数依此设为 1，3，5，7，9，11，15，20，这 8 组测试中，分别对μC/OS II 原有的算法和改进后的系统进行测试比较。

	1	3	5	7	9	11	15	20
静态算法	2105622	681431	350937	234270	178867	143709	101721	73112
改进后的算法	216527	531183	299503	218645	169057	140410	103834	79376

表 4-4 两种不同任务调度算法的测试结果

对上表进行分析后发现，对于任务数较少的简单应用，原有的静态调度算法的得分较高，当任务数增多后，相比较而言，改进后的算法得到了较高的分数。

第六章 总结

影响 RTOS 的性能的因素很多。系统的某一个性能得到提高，往往并不意味着系统整体性能的提高。一个设计良好的构架对于 RTOS 的性能有更加着重大的影响。囿于某一个小点可能使我们失去对于整个系统整体性能的把握。此外，嵌入式系统本身就是一个必须灵活定制的，可裁剪的系统。对于不同的应用来说，系统的性能要求是不同的。这些都要求我们在面对实际问题时必须结合实际，具体问题具体分析，才能拿出真正有效的优化方案。这也是我学习和探索 RTOS 性能以及关键技术的真正原因。

本文的出发点是讨论 RTOS 关键技术和性能的关系。首先，第一章“绪论”中对嵌入式 RTOS 的基本概念进行了论述。第二章“RTOS 关键技术概述”对当前四种开源 RTOS 的性能和采用的关键技术进行了分析和讨论，明确了 RTOS 关键技术的几个改进点。第三章“μC /OS II 系统的任务管理技术和改进”是论文的重点，基于μC /OS II，对任务管理，同步和通信机制以及中断管理这几个方面进行了讨论。在任务管理技术上，对μC /OS II 原有的 64 个优先级进行了扩充，使得可调度的优先级达到了 256 个，此外，参照别的开源 RTOS，将μC /OS II 的静态优先级调度算法改进为支持动态优先级的调度算法。第四章“μC /OS II 任务堆栈的讨论”对于系统在不同处理器下使用不同的堆栈策略进行了讨论。第五章“实时操作系统的性能测试”提出了系统性能试验的方法，包括试验硬件平台和软件平台的搭建，以及测试的过程设计。

因为资源和时间的关系，在一篇论文内对 RTOS 的关键技术进行逐一讨论是不现实的，所以本文只选取了多任务调度算法和系统的内存分配这两项进行了简单的讨论。希望以后有机会能继续学习和研究。

主要参考文献

[1]. ITRON Committee of the TRON Association. μITRON4.0 Specification. Jun-1999

[2]. Jean J. Labrosse 著,邵贝贝等译, 嵌入式实时操作系统μC/OS-II（第二版）,2003,北京航空航天大学出版社

[3]. Express Logic, Inc., Thread-Metric RTOS Test Suite Guide, Apr-2004

[4]. L. Sha, R. Rajkumar, J. Lehoczky, Priority Inheritance Protocols: An Approach to Real-Time Synchronization, IEEE Transactions on Computers, Vol.39, No.9, 1990:1175~1185.

[5]. Goodenough JB, Sha L. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. ACM Ada Letters, Vol.8, No.7,1988: 20~31

[6]. ARM Limited. ARM Architecture Reference Manual. Jun-2000

[7]. ARM Limited. ARM7TDMI Technical Reference Manual. Apr-2001

[8]. 谢拴勤等, 计算机应用研究：基于 RMS 调度周期、非周期混合任务集的一种新方法,，2006 年

[9]. 武汉理工大学信息工程学院，赵二涛等，中国科技论文在线：嵌入式操作系统内核的设计与实现

[10]. Qing Li 著，王安生 译，嵌入式系统的实时概念，2004 年，北京航空航天大学出版社

致谢

首先我要感谢北邮，在北邮的这七年时间里，我不仅学到了专业知识，而且深刻领悟到认真做事，踏实做人的道理，良好的氛围和严谨的学风造就了一批又一批优秀的北邮学子，能在这里度过自己人生中最重要的七年，我深感荣幸。

然后我更要感谢我的导师孙文生副教授！孙老师平易近人的学者风范，严谨的治学态度，踏实的做事准则，使我受益匪浅。本论文是在孙老师的悉心指导下才得以顺利完成的，从论文的选题、收集资料到论文的撰写都凝聚着导师的心血，在此，特向孙老师表示衷心的感谢！

感谢我的家人，他们的鼓励永远是我前进的动力。

感谢我实验室的同学。本班的李涛同学和李哲同学在我学习和写论文期间给了我很多的启发。三人行，必有我师焉，他们用自己的方式向我诠释了勤奋，刻苦和勇往直前。

感谢我的师兄师姐，他们在我的论文写作期间给了我很多的鼓励和帮助。

最后，感谢各位评委在百忙之中抽出宝贵时间评阅本论文。