2006 届研究生硕士学位论文　　学校代码：10269

学　　号：51041500002

华东师范大学

基于对象的嵌入式实时操作系统
—MKRTOS 的实现和移植

院　　系：　软件学院

专　　业：　软件工程

研究方向：　嵌入式系统

指导教师：　曾振柄　教授

硕士研究生：　李文豪

2006 年 4 月完成

摘　要

嵌入式实时系统在人们的日常生活和工作中扮演着日益重要的角色，其开发方法和实现技术也一直是研究的热点。面向对象的程序设计方法在非实时领域已经取得了巨大的成功，但是在嵌入式实时操作系统领域的应用仍然比较少。嵌入式实时操作系统一般都是采用微内核结构，系统功能集中在微内核中实现。但是随着微内核中提供的服务不断增加，其规模也随之扩大，因此如何保证系统的实时性、可伸缩性和可移植性成为了开发者需要关注的问题。

在上述的研究背景下，本文论述了一款基于对象的超微内核嵌入式实时操作系统 MKRTOS（Micro-Kernel Real-Time Operation System）的设计实现以及在 ARM 处理器平台上的移植。本文的研究内容主要分为 4 个部分：

1） 在系统的设计和实现过程中运用了基于对象的方法，在比较面向对象与面向过程的开发方法的基础上，指出了使用面向对象的开发方法设计嵌入式实时操作系统的诸多优点，并且对 MKRTOS 中的面向对象的特性进行了简单的分析。

2） 由于微内核结构渐渐地不能再满足嵌入式实时操作系统的性能要求。本文提出了一种全新的内核设计思想：即在微内核的基础上增加了超微内核层。超微内核是对微内核的精简，它只包含一些最重要、最基本的系统功能，具有更少的任务上下文和切换时间，从而保证操作系统的性能要求。

3） 在对 μC/OS-II 的任务管理方案和任务调度策略进行研究的基础上，提出了 MKRTOS 任务管理的实现方案，同时根据 MKRTOS 中任务的特点，结合了 RMS 和 Round-Robin 这两种调度算法，设计了 MKRTOS 的任务调度策略，并且详细地阐述了它们的实现细节。

4） ARM系列处理器是32位嵌入式系统应用的主流，在众多领域都有广泛的应用，因此我们选择ARM7处理器平台来移植MKRTOS。论文中介绍了ARM7处理器平台——LPC2200实验开发板的基本特点和性能，并且阐述了将MKRTOS移植到该实验平台上的细节。

【关键词】嵌入式实时操作系统，优先级反转，单调执行率调度算法，时间片轮转调度算法，内核移植

Abstract

Embedded real-time operation system has been acting as a more and more important role in our industry and social life. With requirements growing rapidly, the developing methodology for real-time systems gradually becomes the focus of researchers. Object-oriented analysis and design methodology has got huge popularity in non-real-time system domains; but there is still little application in developing real-time systems. Commonly, embedded real-time operation systems adopt micro-kernel architecture, the system functions are collected in the kernels. With the increasing of the functions, the kernel becomes more and more complicated, so it makes the performances of the system become worse and worse.

Based on the background stated above, the thesis describes the design , implementation and porting of an Object-Oriented embedded real-time operation system with nano-kernel on ARM processor which called MKRTOS. The main contents of the thesis can be devided into 4 parts:

1) Comparing the structure based developing methodology with Object-oriented methodology, then the thesis points out the advantages of developing RTOS using Object-oriented developing methodology and analyses the OO characters of MKRTOS.
2) Because the micro-kernel architecture cannot satisfy the performances of embedded real-time operation systems gradually, the thesis raises a new idea to design the kernel, which is increasing a nano-kernel layer under the micro-kernel layer. The nano-kernel is much simplier than micro-kernel, it just has some very important and basic functions and it has less task contexts and switching time.
3) Based on the researches of task mangement and task scheduling strategy in μC/OS-II, we implement the basic functions of task management and task scheduling strategy in MKRTOS which based on RMS arithmetic and Round-Robin scheduling arithmetic.
4) The series of ARM processors are very popular in 32 bits embedded systems's applications. It has got huge popularity in many fields, so we deceide to port MKRTOS on platform with ARM7 processor. In thesis, we introduce the characters of the target platform-LPC2200 and describle how to port MKRTOS on it in detail.

【Key Words】 embedded real-time operation system, priority inversion, task scheduling, Rate Monotonic Scheduling, Round-Robin Scheduling, kernel porting

学位论文独创性声明

本人所呈交的学位论文是我在导师的指导下进行的研究工作及取得的研究成果。据我所知，除文中已经注明引用的内容外，本论文不包含其他个人已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中作了明确说明并表示谢意。

作者签名：李文豪　　日期：2006.5.11

学位论文授权使用声明

本人完全了解华东师范大学有关保留、使用学位论文的规定，学校有权保留学位论文并向国家主管部门或其指定机构送交论文的电子版和纸质版。有权将学位论文用于非赢利目的的少量复制并允许论文进入学校图书馆被查阅。有权将学位论文的内容编入有关数据库进行检索。有权将学位论文的标题和摘要汇编出版。保密的学位论文在解密后适用本规定。

学位论文作者签名：李文豪　　导师签名：

日期：2006.5.11　　日期：2006.5.29

第 1 章 绪论

本章主要对嵌入式系统的定义、特点、分类、发展以及嵌入式操作系统的基本概念进行了介绍，并且在对国内外嵌入式操作系统领域的研究现状进行了分析之后，阐述了本文的研究意义，说明了设计一款基于对象的超微内核嵌入式实时操作系统的可行性和必要性。

1.1 嵌入式系统概述

1.1.1 嵌入式系统的定义

随着计算机技术的发展，嵌入式系统(Embedded Systems)已成为计算机领域的一个重要组成部分，并被广泛应用于信息电器、移动计算机设备、网络设备和工控仿真等领域。嵌入式系统的相关技术也成为近年IT行业的热点。

目前，关于嵌入式系统，有一种普遍流行的定义，[1] 即："嵌入式系统是以应用为中心、以计算机技术为基础、软件硬件可裁剪、适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。" 但是上述定义过于模糊，没能将嵌入式系统与其他事物（传统的单片机系统和传统的计算机系统等）区别开来，更没能深入嵌入式系统的实质。

从嵌入式系统的名称来分析，"嵌入式系统"中的"系统"实际上指的是一个"子系统"，这意味着在这个"子系统"外面还有更大更复杂的大系统和大设备，我们称其为宿主系统。"嵌入式系统"是嵌在那个宿主系统中的部件。同时，"嵌入式系统"作为一个系统，应当不依赖于外部其他系统就能独立运行。由此，我们可以这样定义嵌入式操作系统：嵌入式系统描述的是一种智能系统，它嵌入在其他以非传统计算机形态出现的外部系统中，并且可以不依赖外部系统而独立运行，智能系统是这些产品的控制、运算或信息处理的中心[2]。

1.1.2 嵌入式系统的特点

嵌入式系统的主要特点有：[3]

- 嵌入式系统通常是面向特定应用的。嵌入式系统的专用性很强，其中的软件

系统和硬件的结合非常紧密，一般需要针对硬件进行系统的移植。同时针对不同的任务，往往需要对系统进行较大更改，程序的编译下载要和系统相结合，这种修改和通用软件的“升级”是完全不同的概念。

- 系统精简。嵌入式系统一般没有系统软件和应用软件的明显区分，不要求其功能设计及实现上过于复杂，这样一方面利于控制系统成本，同时也利于实现系统安全。
- 为了提高执行速度和系统可靠性，嵌入式系统中的软件一般都固化在存储器芯片或单片机本身中，而不是存贮于磁盘等载体中。
- 嵌入式软件开发走向标准化。为了合理地调度多任务、利用系统资源、系统函数以及专家库函数接口，用户必须自行选配RTOS (Real Time Operating System)开发平台，这样才能保证程序执行的实时性、可靠性，并减少开发时间，保障软件质量。
- 嵌入式系统本身不具备自举开发能力，即使设计完成以后用户通常也是不能对其中的程序功能进行修改的，必须有一套开发工具和环境才能进行开发。开发时往往有主机和目标机的概念，主机用于程序的开发，目标机作为最后的执行机，开发时需要交替结合进行。
- 多数嵌入式应用要求操作系统具有实时处理能力。在多任务嵌入式系统中，对重要性各不相同的任务进行合理调度是保证每个任务能够及时执行的关键，单纯的通过提高处理器速度是低效的，甚至是无法实现的，必须有一个高效的多任务实时操作系统来支撑。

1.1.3 嵌入式系统的分类

1) 按应用目标分：[1][3]

a) 系统控制型嵌入式系统：智能机器人、水电控制中心等等都包含了以可靠的控制为主要目标的嵌入式系统。

b) 信息处理型嵌入式系统：计算机网络、通讯、多媒体等等应用则包含了以信息的高效处理为主要目标的嵌入式系统。虽然单纯数学运算的应用也存在，但似乎传统计算机更适合这种应用。

2) 按嵌入深度分：嵌入深度的界定在于离最外层和最内层系统的距离，越外层的系统离“最终用户”越近，嵌得越浅；越内层的系统离“最终用户”越远，嵌得越深。

a) 片级嵌入式系统：既是指嵌在某块芯片上的子系统，如 ARM 核就是片级

的嵌入式系统，它嵌在各种集成微处理器中，片级嵌入式系统涵盖了集成电路的设计方法，也被称为片上系统(System on Chip，SoC)。

b) 板级嵌入式系统：如以 MCU 为核心，配以硬盘等多种外设的产品，其中的 MCU 与外设构成的系统就是板级嵌入式系统，这是常见的嵌入式系统。

c) 系统级嵌入式系统：如神舟号上的生命保障系统就是系统级的嵌入式系统。

1.1.4 嵌入式系统的发展

嵌入式系统已经渗透到各个领域：工业自动化、消费类电子产品、数据通信、电信、仪器操作、卫生保健等。随着社会的日益信息化，计算机和网络已经全面渗透到日常生活的每一个角落。以信息家电为代表的互联网时代嵌入式产品，不仅为嵌入式市场展现了美好前景，注入了新的生命力，同时也对嵌入式系统技术，特别是软件技术提出新的挑战。

未来嵌入式系统的发展趋势：[5]

1） 嵌入式应用软件的开发需要强大的开发工具和操作系统的支持。

随着因特网技术的成熟、带宽的提高，网上提供的信息内容日趋丰富、应用项目多种多样，像电话手机、电话座机及电冰箱、微波炉等嵌入式电子设备的功能不再单一，电气结构也更为复杂。为了满足应用功能的升级，设计师们一方面采用更强大的嵌入式处理器如32位、64位RISC芯片或信号处理器DSP增强处理能力；同时还采用实时多任务编程技术和交叉开发工具技术来控制功能复杂性，简化应用程序设计、保障软件质量和缩短开发周期。

2） 网络互联成为必然趋势。

为适应嵌入式分布处理结构和应用上网的需求，面向21世纪的嵌入式系统要求配备标准的一种或多种网络通信接口。针对外部联网要求，嵌入设备必需配有通信接口，相应需要TCP/IP协议簇软件支持；由于家用电器相互关联及实验现场仪器的协调工作等要求，新一代嵌入式设备还需具备IEEE1394，USB，CAN，Bluetooth或IrDA通信接口，同时也需要提供相应的通信组网协议软件和物理层驱动软件。为了支持应用软件的特定编程模式，如Web或无线Web编程模式，还需要相应的浏览器，如HTML等。

3） 支持小型电子设备实现小尺寸、微功耗和低成本。

为满足这种特性，要求嵌入式产品设计者相应降低处理器的性能，限制内存容量和复用接口芯片。这就相应提高了对嵌入式软件设计技术的要求。如选用最佳的编程模型和不断改进算法，优化编译器性能。因此，既要求软件人员有丰富

经验，更需要发展先进嵌入式软件技术，如Java, Web和WAP等。

4） 提供精巧的多媒体人机界面。

嵌入式设备之所以为亿万用户乐于接受，重要原因之一是它们与使用者之间的亲和力，自然的人机交互界面，如司机操纵高度自动化的汽车主要还是通过习惯的方向盘、脚踏板和操纵杆。人们与信息终端交互要求以GUI屏幕为中心的多媒体界面。手写文字输入、语音拨号上网、收发电子邮件以及彩色图形、图像已取得初步成效。

1.2 嵌入式操作系统概述

在最初的嵌入式系统中并没有嵌入式操作系统的概念，因为完成简单功能的嵌入式系统一般不需要操作系统，如以前许多MCS51系列单片机组成的小系统就只是利用软件实现简单的控制环路。但是随着所谓后PC时代的来临，嵌入式系统设计日趋复杂，嵌入式操作系统就必不可少了。一般而言，嵌入式操作系统不同于一般意义的计算机操作系统，它有占用空间小、执行效率高、实时性、方便进行个性化定制、软件要求固化和存储等特点[4]：

1） 小巧高效：

受嵌入式系统所能够提供的资源限制，嵌入式操作系统在高效地完成预定功能的同时必须做到体积较小。

2） 实时性：

大多数嵌入式系统工作在具有实时性要求的环境中，不同任务要求系统有不同的响应时间，这就要求嵌入式操作系统应该能对重要性不同的各个任务进行统筹兼顾的合理调度保证满足每个任务的需求。

3）可裁减性：

由于嵌入式系统需要迎合各种各样的需求，所以一个好的嵌入式操作系统也必须能够根据应用的要求进行相应的裁减，去掉多余的部分，添加需要的部分，这是一个好的嵌入式操作系统的重要特征。

从上世纪八十年代开始，开始出现各种各样的商业用嵌入式操作系统，这些操作系统大部分都是为专有系统而开发，从而形成了现在多种形式的商用嵌入式操作系统并存的局面。现在市面上流行的嵌入式操作系统主要有：windows CE、VxWorks、QNX、EEOS、HOPEN OS和μclinux等等。

1.3 国内外研究现状分析

在嵌入式实时操作系统领域，国外无论是在技术水平还是在市场占有率上都远远超过我国。从 1981 年开始，国外在嵌入式实时操作系统上的研究已经超过 20 年，而且已经有一批著名的产品：Vxwork，QNX，Lynx 等。

国内在这几年中也已经着手开发自主知识产权的 RTOS，以及在开放源码的 Linux 基础上发展自己的嵌入式 Linux 版本。目前主要产品有：中科院系统的 "女娲"，英文 是"Hopen"；北京科银京成(原电子科大)的 δ OS（原名是 CRTOS）；中科院红旗 Linux；深圳蓝点 Linux。但是这些产品的市场占有率都较低。

纵观这些操作系统，基本上都是用 C 语言开发的，而用 C++开发的操作系统几乎是没有的。虽然标准 C++的面向对象的特性较之 C 语言有很多的优势，但是并非它所有的特性都适用于嵌入式实时操作系统，反而它的某些特性还会对系统的性能产生负面的影响，这也阻碍了标准 C++在实时嵌入式系统中的应用。1996 年，一群日本的芯片厂商联合起来定义了一个 C++语言和库的子集，使得 C++适合嵌入式软件的开发。嵌入式 C++省略了很多不限制下层语言可表达性的任何可以省略的东西，包括：多重继承性、虚拟基类、运行时类型识别和异常处理等昂贵的特性，以及一些最新的添加特性，比如：模板、命名空间、新的类型转换等。所剩下的是一个 C++的简单版本，仍然是面向对象的并且是 C 的一个超集，但是它具有明显更少的运行开销和更小的运行库。很多商业的 C++编译器已经专门支持嵌入式 C++标准。面向对象封装带来的松耦合，加上嵌入式 C++的出现，使得它成为分布式可重构、可伸缩嵌入式系统的首选技术。

1.4 论文的研究意义

从 1981 年 Ready System 发展了世界上第 1 个商业嵌入式实时内核(VRTX32)开始，嵌入式实时操作系统就以飞快的速度不断的发展着。如今，嵌入式实时操作系统在嵌入式系统设计中已经占据了主导的地位。而且，嵌入式应用本身极具多样性，所以不太可能出现 1 个垄断的产品，因此我们希望可以通过自己设计开发一个嵌入式实时操作系统，对该领域的技术以及嵌入式实时操作系统的开发方法有更深入的理解和认识。

对实时系统而言，面向过程的开发方法已经渐渐地不能满足其设计需要。这是因为面向过程的开发方法没有考虑系统模块和现实世界事物之间的映射，从而导致许多涉及系统与外界交互的关键数据或操作以紧密耦合的方式分布在多

个模块中，使得软件的灵活性和一致性都较差；而面向对象的开发方法正好弥补了上述的不足之处，并且以其良好的抽象性、封装性、灵活性和重用性在理论上和工程实践中都成为较成功的软件开发方法。然而，使用纯面向对象方法开发实时系统特别是硬实时系统在设计和实现阶段同样面临困难。首先，面向对象软件系统的正确性取决于对象内部的操作和对象间交互的动态语义，而对象交互的语义复杂且不如面向过程模型那样直观，这导致实时系统的面向对象模型难于理解和验证；其次，面向对象程序设计语言的许多特性都对系统的总体性能有负面的影响，例如：动态绑定和垃圾回收机制会降低系统的可预测性。因此可以利用面向对象语言的某一些特性，诸如，抽象性和封装性等，开发一款基于对象的嵌入式实时操作系统，从而弥补面向过程的开发方法的不足，而又不影响系统的总体性能。

嵌入式实时操作系统的内核一般都是采用微内核的结构，系统功能集中在微内核中实现。但是随着微内核中提供的服务不断增加，微内核的规模也随之扩大，导致系统的实时性、可伸缩性和可移植性都变得难以保障。因此，我们提出了超微内核的设计思想：在微内核的基础上增加了超微内核层，通过精简微内核的功能，将最基本的系统功能和高实时性的任务以及与处理器密切相关的数据结构等放到超微内核中实现，来保证系统的性能。

我们选择在 ARM 体系架构上进行移植是因为 ARM 系列处理器是 32 位嵌入式系统应用的主流，ARM 处理器在计算机专业、电子技术专业以及其他的一些领域都有广泛的应用。通过在 ARM 体系结构上进行移植，可以使我们加深对 ARM 体系结构的了解，为以后进行其他操作系统在 ARM 处理器上的移植或是开展其他与 ARM 有关的工作提供帮助。

1.5 本章小结

本章首先介绍了嵌入式系统，对嵌入式系统的定义，特点，分类和发展进行了简单的描述；然后介绍了嵌入式操作系统的相关概念，指出了嵌入式操作系统是嵌入式系统的核心；接着，本章阐述了论文的研究意义，说明了要开发基于对象的嵌入式实时操作系统的原因和可行性，以及论文的主要内容；最后论述了国内外在嵌入式实时操作系统领域的研究现状。

第 2 章 面向对象程序设计方法

本章首先介绍了面向对象的主要思想，并且在比较面向对象和面向过程的开发方法的基础上，分析了使用面向对象的开发方法开发嵌入式实时操作系统的优势所在；然后简单的介绍了面向对象编程应该遵循的几点基本要求；最后对 MKRTOS 中面向对象的特性进行了分析。

2.1 面向对象的主要思想

面向对象是一种新兴的程序设计方法，或者说是一种新的程序设计规范(paradigm)，其基本思想是使用对象、类、继承、封装、消息等基本概念来进行程序设计。从现实世界中客观存在的事物（即对象）出发来构造软件系统，并且在系统构造中尽可能运用人类的自然思维方式。面向对象涉及的一些基本概念如下所示：[6]

- 对象的基本概念

对象是系统中用来描述客观事物的一个实体，它是构成系统的一个基本单位。一个对象由一组属性和对这组属性进行操作的一组服务组成。

- 类的基本概念

类是具有相同属性和服务的一组对象的集合，它为属于该类的所有对象提供了统一的抽象描述，其内部包括属性和服务两个主要部分。在面向对象的编程语言中，类是一个独立的程序单位，它应该有一个类名并包括属性说明和服务说明两个主要部分。

- 消息

消息就是向对象发出的服务请求，它应该包含下述信息：提供服务的对象标识、服务标识、输入信息和回答信息。服务通常被称为方法或函数。

2.2 面向对象方法与面向过程方法的比较

本节所说的两种程序设计方法是指：面向过程的程序设计和面向对象的程序设计。

面向过程的程序设计将问题分解为一系列顺序执行的过程，这种描述方式特别符合计算机本身的构造和运算模式，适用于处理计算性较强的操作。面向过程的程序设计方法比较适合于单任务的操作系统（如 MS-DOS）。

但是，客观世界的事物往往是复杂的、并发的，很难用一串连续的过程加以

描述，因此产生了面向对象的程序设计方法。面向对象设计方法最基本的思想就是把客观世界看成一个个相对独立而又相互联系的实体，即对象。每个对象都有自己的状态和行为，能够完成一定功能。在面向对象的程序设计中（以C++为例），状态称为属性，对应C++类的数据成员；行为称为方法，对应C++类的成员函数。面向对象的思想就是把方法和属性结合起来，把它们看成一个有机整体。面向对象的程序设计方法特别适合于多任务系统[9]。

表 2-1 面向对象与面向过程的编程语义的比较

传统方法	面向对象方法
数据结构＋算法＝程序设计	以对象为中心组织数据与操作
数据	对象的属性
操作	对象的服务
类型与变量	类与对象实例
函数过程调用	消息传递
类型与子类型	一般类与特殊类，继承
构造类型	整体－部分结构（聚合）

表 2-1展示了面向对象的编程语义与传统面向过程的编程语义的比较。可见，编程语义的提升，使得解决问题的思想观念也得到了提升：从数据与操作紧密结合的对象出发，认识问题域，并以对象作为构成系统的基本单位。

使用面向对象的方法构建软件系统具有以下优点：[6][12]

- 充分运用人类日常的思维方法，使用人类在日常的逻辑思维中经常采用的思想方法与原则，例如抽象、分类、继承、聚合、封装和关联等等。这使得软件开发者能更有效地思考问题，并以其他人也能看得懂的方式把自己对问题的理解和认识表达出来。
- 用类和对象作为系统的基本构成单位，对象对应问题域中的事物，对象的属性和服务反映了事物的静态特征和动态特征，对象之间的继承关系、聚合关系、消息和关联如实地表达了问题域中事物之间实际存在的各种关系。因此，无论是上述的这些构成系统的部分，还是通过这些部分之间的关系而体现的系统结构，都可以直接映射问题域。
- 使用对象封装的特性，把易变的数据结构和部分功能封装在对象内并加以隐藏，既保证了对象行为的可靠性，又可以避免对它们进行修改时影响到其他的对象。有利于软件维护，对需求的变化有较强的适应性。
- 把对象的属性和操作捆绑在一起，提高了对象（作为模块）的内聚性，减少了与其他对象的耦合，方便对象的复用。在继承结构中，特殊类对一般类的继承，本身就是对一般类的属性和操作的复用。

2.3 基于对象编程的基本要求

用 C++开发面向对象系统的基本原则如下：[9]

1）高度的凝聚性：基类和类的继承必须具有高度的凝聚性。所谓凝聚性是指类应具有单一的用途，如果某个类具有多个用途，应将其分解为多个专用的类；

2）松散的耦合：所谓耦合是指系统的各组成部分之间的相互依赖性。像方法和类这样的要素，设计时应尽可能互相独立。如果某个类过于依赖其他类来完成其工作，那么就必然要牺牲可重用性和可移植性；

3）数据隐藏：类的数据和实现细节对于别的类来说应该是不可见的。因此，在设计 C++类时，应尽量避免在一个类中有大量公用数据元素；避免类的实现依赖于全局变量；避免类中含有大量友元（其他特定的类能够直接访问的成员函数）；

4）尽可能重复使用现有代码；

5）设计类的时候保持类接口的稳定性：类的接口也就是类的 public 部分，对派生类来说，还包括基类的 protect 部分。类接口定义了该类怎样同外部世界交流，保持其稳定性是实现兼容性的必须条件。

2.4 MKRTOS 中面向对象特性的分析

MKRTOS 是使用 C++编写的嵌入式实时操作系统，在设计开发的过程中都是按照 2.3 节中的要求进行的，本节中将主要讨论 MKRTOS 的封装性的特点。

封装把发生关联的所有属性和方法捆绑在一起，把它们看成一个有机整体，从而使对象内部有了明确的范围和清楚的外部边界。封装是面向对象理论中十分重要的概念，因为封装实现了数据隐藏，保护了对象的数据不被外界随意改变，并且封装使对象成了相对独立的功能模块，提高了程序的结构化和模块化的程度。如果需要改变某个对象，只要它对外的接口保持不变，系统的其它部分就可以不受影响。

在程序中加入了封装这一特性之后，我们需要考虑两个问题：首先，程序的布局成本是否会因此而有所增加；其次，程序是否达到了封装的要求。下面我们会分别来讨论这两个问题。

MKRTOS 是嵌入式实时操作系统，系统运行的硬件环境有很苛刻的要求，因此我们要尽量保证程序代码本身不会为系统带来太大的额外负担。而事实证明，增加了封装性之后，布局成本并没有比不具备这一特性的程序（使用 C 语言等面向过程的编程语言开发的程序）有所增加。下面我们就用 C++的 class 和 C 语言

的 struct 进行比较：[10][12]

- 首先，对于 data members 而言，他们直接内含在每一个 class object 或者是 struct 之中，这对于 C++和 C 来说都是相同的；
- 其次，对于 member functions 而言，虽然他们包含在 class 的声明之内，但是却不出现在 object 之中，每一个 non-inline member function 只会诞生一个函数实体，至于每一个“拥有零个或一个定义”的 inline function 则会在其每一个使用者（模块）身上产生一个函数实体。

因此可以发现增加了封装性之后，并没有造成任何空间或是执行效率上的不良反映。而 C++在布局和存取时间上主要的额外负担是由虚函数和多重继承引起的，因此出于系统执行效率上的考虑，在我们的程序中并没有使用虚函数和多重继承。

加入了封装性之后，还需要考虑我们的程序是否达到了封装性的要求。封装性有两个方面的内容：[8] 对象的封装性和类的封装性（即信息隐藏）。

- 对象封装性是指对象的属性与服务结合为一体，即所有对该对象的服务请求都是施加于该对象的。在程序中可以支持“对象.方法”的访问方式，如：我们可以使用 OS.OSInit()来初始化内核或是使用 OS.OS_Sched()来启动任务调度器，其中 OS 是类 Ckernel 的一个对象实例，Osinit()和 OS_Sched()分别是类 Ckernel 的方法。
- 类的封装性是指把内部的属性和服务隐藏起来，只有公共的服务对外是可见的，它决定了该类对象的封装性，以及对象能够对外提供的服务。在 C++中可以通过 private，protected 和 public 来实现对类的不同程度的信息隐藏。比如在程序中，我们使用 private 关键字来修饰 OSTCBPrioTblReadyCXH[MAXPRIO]，OSTCBPrioTblWaitingCXH[MAXPRIO]和 OSTCBPrioTblWaitingTimerCXH[MAXPRIO]等私有的变量，要存取这些变量只能通过在类中定义的公共函数 OSSetOSTCB，OSGetOSTCB 和 OSGetCurOSTCB 等，而直接使用“对象.变量名”的方法是无法访问这些需要保护的对象的。

由此可以看出我们的程序达到了封装性的要求。所以 MKRTOS 是支持对象的封装性的。

2.5 本章小结

本章首先论述了面向对象技术的主要思想，说明了什么是面向对象技术；然后比较了面向对象程序设计方法与面向过程的程序设计方法的不同之处；最后分析了 MKRTOS 中面向对象的特性，并且对 MKRTOS 中的封装性进行了详细的分析。

第 3 章 MKRTOS 内核设计

本章主要介绍了 MKRTOS 的内核设计方案和实现。在 MKRTOS 的内核设计过程中引入了超微内核的思想，即是在传统的微内核的基础上增加了超微内核层。本章首先简单地介绍了操作系统内核的发展史；然后，提出了 MKRTOS 的内核设计方案，并分层次地描述了 MKRTOS 的内核结构；最后，给出了 MKRTOS 内核的部分实现代码。

3.1 操作系统的发展概述

3.1.1 宏内核结构

对于宏内核结构来说，整个操作系统就是一个整体，其中包括了进程管理，内存管理，中断处理，设备驱动，文件系统等等。传统的宏内核操作系统既对用户程序提供服务功能，同时又作为管理者管理着整个系统。它的优点和缺点都是非常明显的。由于全部功能集中在一起，系统花在内核功能切换上的开销就非常的小（例如文件系统到I/O驱动系统上的切换），提供给用户程序的反应就很快。同时，因为全部功能集中在一起，作为软件学上的最大忌讳，各个功能之间的耦合度就很高，导致了内核难以修改和增加新的功能[13]。

3.1.2 微内核结构

为了改进宏内核的缺点，因此就提出了微内核的结构。微内核的观点是在微内核中只完成最基本的服务功能，其他的管理功能（包括内存管理，文件系统等等）都放在微内核之外，交给一个或多个特权服务器进程处理。在微内核之外的服务进程都处于同一级别（既所谓的水平分层），它们之间通过向微内核传递消息来实现交互。典型的微内核结构如图 3-1 所示[13]：

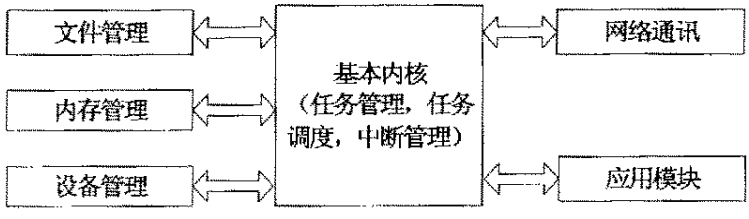

图 3-1 微内核结构图

微内核以静态的内核调用表作为其通讯的标准，实现模块之间的调用。使用微内核结构的优点有：

1） 充分的模块化，模块可以独立更换而不影响其它模块，方便让第三方的开发者设计新的模块；

2） 未被使用的模块功能不必运行，因而能大幅度减少系统的内存需求；

3） 具有很高的可移植性，几乎所有与处理器相关代码都包含在微内核中，理论上讲只需要对微内核部分进行移植修改即可；

微内核结构也有缺点，系统划分的粒度越精细，必然会造成系统整体性能的下降，这对微内核体系结构来说也是如此。相对于传统的宏内核操作系统，微内核的模块化程度得到很大的提高，但同时也导致了模块间通信代价的提高，服务器进程以及用户进程必须用进程间通信的方式来得到微内核的服务，同时需要对应的进程上下文切换。大多数嵌入式操作系统都是采用微内核结构，比如QNX, Vxworks, μC/OS-II等。

3.1.3 超微内核结构

严格的来说超微内核并不能算是一种操作系统的体系结构，因为使用到超微内核的操作系统无一不是基于微内核结构的，因此，将它理解为微内核结构的扩展更为合适。在我们的系统中，也是在微内核结构上增加了超微内核层。

提出超微内核这个概念是因为随着嵌入式实时应用领域的应用需求不断的提高，微内核结构已经渐渐的变得不再精简，实时性和可移植性都受到了极大的挑战。超微内核是对微内核进一步精简，提供了最小的操作系统的功能集。它既具有汇编的高效性，又具有模块化的特点。超微内核主要用于处理任务调度、中断管理器、定时器以及处理一些用于系统管理的数据结构。它具有更少的任务上下文，切换时间更快。

在微内核结构中，硬件中断可能会被当作消息处理。微内核可以识别中断但不能处理中断，它为与该中断相关联的用户级进程产生一条消息，并且将其放在消息队列中等待下一个时钟周期到来才能做相应的处理；而在超微内核中，具有高实时性要求的外部中断会跳过内核，直接调用任务指定的中断服务程序。这样就去除了调度延迟和内核延迟的开销，提高了系统的实时性能。

3.2 MKRTOS 内核的设计方案

3.2.1 操作系统的总体结构

实时操作系统内核的实现大多采用微内核的体系结构。整个操作系统是由提供一些基本服务机制的微内核加上一些服务进程构成，系统的各个系统调用和服务都是由内核发消息到不同的服务进程，服务进程执行相应的操作，然后以消息的方式返回内核[16]。

MKRTOS 系统的总体结构图如图 3-2 所示。该操作系统的总体结构采用基于对象的微内核设计，同时增加了超微内核层，以进一步地提高实时性[14][15]。

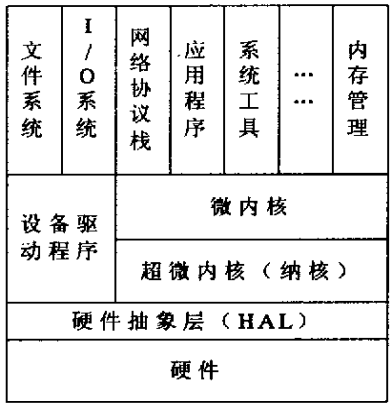

图 3-2 超微内核系统的总体结构

超微内核层主要实现了调度器，中断管理器和定时器等功能，具有更少的任务上下文，切换时间更快，同时为微内核层提供服务。下面详细介绍各层的设计。

3.2.2 超微内核层

超微内核层由许多缩减了上下文的纳核层任务（PROCESS）组成，在 40MHZ 下可以实现少于 1 微秒的任务上下文切换。纳核任务被称为 PROCESS，提供不同的通信和同步原语。每个 PROCESS 以汇编程序开始和结束，而且能够调用 C 或 C++函数和使能中断。一般情况下，可以在该层写有关低级的设备驱动程序和时间关键性代码。由于低开销和 Round-Robin 调度，对于数据流驱动的应用或者编写快速的驱动程序来说，超微内核是非常理想的。在所有的超微内核层的 PROCESS 中，有一个 PROCESS 是专门为微内核任务调度提供服务的，处理微内核任务的调度。

超微内核层的 PROCESS 有独特的优点，结合了任务的易用性和 ISR 的执行速度。除了可以通信和同步之外，超微内核层的 PROCESS 也可以等待和调度（在 ISR 是不可行的）。在多处理器系统中，超微内核层的 PROCESS 扮演了重要的角色，因为如果没有超微内核层的 PROCESS，开发者只能选择在 ISR 层或是任务层编程，但是前者因为处于临界区而需要关中断，后者会增加系统的反应时间。在多处理器系统中，在任务层编程会产生很坏的影响。因为处理器之间的通信必须在高优先级的中断进行处理，如果没有尽快的处理，它可能使远程处理器（中断源）延迟。超微内核的 PROCESS 调度可以采用 Round-Robin 调度算法或是基于优先级的调度算法。

3.2.3 微内核层

MKRTOS 的微内核层是面向对象的 C++语言层。该层是完全可以抢占式的，优先级驱动，而且每个任务都有自己的完全的上下文。它为建立应用程序提供了高层的框架。而大多数实时内核仅仅提供了唯一的 ISR 层和 C 任务层。一般来说，超微内核层的服务时间少于微内核服务的 10-20 倍。这不仅是因为交换的寄存器数量的不同，而且主要是因为在微内核中比在超微内核中有更多的语义上下文[16]。MKRTOS 的微内核层需要调用超微内核层提供的服务，并且向上层提供服务。

3.3 MKRTOS 的超微内核的实现

CKernel 中实现了超微内核中最主要的一些功能，如：初始化变量，任务的创建和删除，任务调度，开中断和关中断等等服务。[16][17][22]以下是部分接口描述：

```
class CKernel
{
    public:                                    //提供的公有数据
        INT32U OSTaskCtr;                      //已经创建任务的个数
        BOOLEAN OSRunning;                     //操作系统是否运行的标志
        INT32U OSCtxSwCtx;                     //任务上下文切换次数
        INT8U OSPrioCur;                       //当前任务的优先级
        INT8U OSPrioHeighest;                  //最高任务优先级
        INT32U OSTaskMaxID;          //最大任务的 ID 值+1,即空闲的//ID 号
        OS_STK *OSCurTCB;
```

```
private:                                    //提供的私有数据
    OS_TCB_CXH *OSTCBPrioTblReadyCXH[MAXPRIO];  //就绪态数组链表
    OS_TCB_CXH *OSTCBPrioTblWaitingCXH[MAXPRIO]; //等待态数组链表
    OS_TCB_CXH *OSTCBPrioTblWaitingTimerCXH[MAXPRIO];
                                                //等待时间数组链表
public:                                     //提供的公有方法
    void OS_Sched();                        //任务调度器
    void OSInit();              //OS 的初始化：初始化 OS 所有的变
                                //量和数据结构，建立空闲任务和统计任务
    void OSStart();             //找出优先级最高的任务控制块，
                                //然后调用高优先级就
                                //绪任务启动函数 OSStartHighRdy()
    INT8U OSInsertOSRdyTCB(INT8U     prio, OS_STK    *ptos,
                           OS_STK    *pbos, INT16U    id,
                           INT32U    stk_size, void      *pext,
                           INT16U    opt);
                                //插入 TCB 到就绪任务表中
    OS_TCB_CXH * OSSetOSTCB(INT8U prio,INT16U id,INT8U type,INT8U
                        newprio);
                //改变 TCB 的内容,对哪个队列进行操作由 type 决定
    OS_TCB_CXH * OSGetOSTCB(INT8U prio,INT16U id,INT8U type,OS_TCB
                *ostcb);
                //从就绪\等待\TIMER 任务列表中,得到 TCB 的内容,
    OS_TCB_CXH *OSGetCurOSTCB(INT8U prio,INT8U type,OS_TCB *ostcb);
                //从就绪\等待\TIMER 任务列表中,得到当前 TCB 的内容
    void OSEndCurTask();
                //结束当前任务,并释放内存
    INT8U OSFindHighestRdy(INT8U type);
                //从就绪\等待\TIMER 任务队列中查找最高优先级
    void OS_ENTER_CRITICAL();                   //关中断
    void OS_EXIT_CRITICAL();                    //开中断
};
```

超微内核 PROCESS 不能被其他的 PROCESS 抢占，但是可以被中断服务程序打断，也可以抢占微内核层的任务。超微内核不仅要处理自身的任务（PROCESS）

调度，而且要为微内核层的任务调度提供服务。MKRTOS 在超微内核中实现了一些最最基本的功能，诸如上下文切换，开关中断，任务调度，操作就绪表（插入、删除和查找）和事件等待列表等等功能，并且作为服务提供给上层使用；处于微内核中的任务，诸如任务管理，内存管理，定时器管理都可以调用超微内核提供的相应的服务。如图 3-3 所示：

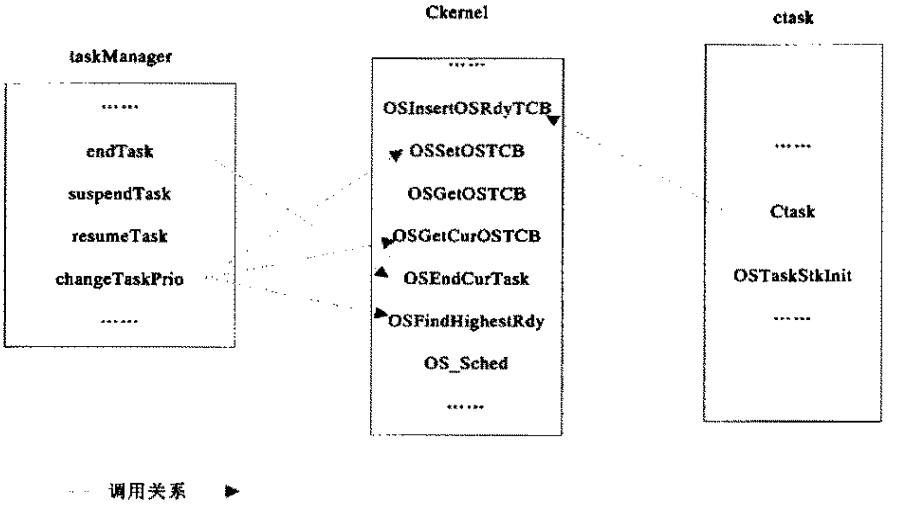

图 3-3 MKRTOS中的服务调用图

3.4 本章小结

本章首先介绍了操作系统的发展历程，并且简单的介绍了操作系统发展过程中 3 中不同的内核结构——宏内核结构，微内核结构和超微内核结构；然后提出了一种超微内核体系结构的嵌入式实时内核的设计方案，并且对该结构中的每一个层次分别进行了介绍；最后给出了 MKRTOS 的超微内核的实现代码，并且进行了分析。

第 4 章 MKRTOS 任务管理的设计与实现

任务可以分为周期任务、间歇任务和非周期任务三类。周期任务是按照固定的频率被激活(请求处理器)的任务，该频率就是任务的周期。间歇任务是相邻两次激活间隔的时间不少于某个特定值的任务。非周期任务是激活时间完全没有规律的任务。

对于非周期任务而言，硬实时的时间约束很难得到保障，而系统设计者和用户更关心任务请求的平均响应时间或截止期错过率等与服务质量有关的参考数值。当任务集合既包含周期任务又包含非周期任务时，调度的目标通常是在保证周期任务得到满足的前提下最大限度地提高非周期任务的服务质量。

本章主要可以分为 3 个部分：首先，简单地介绍了 μC/OS-Ⅱ的任务管理和任务调度，因为 MKRTOS 中任务管理的实现参考了 μC/OS-Ⅱ的任务管理机制；然后，介绍了 MKRTOS 的任务管理的基本功能，指出了它与 μC/OS-Ⅱ中任务管理的区别；最后，根据 MKRTOS 任务管理的特点，结合多种实时调度算法，设计并实现了 MKRTOS 的任务调度策略。

4.1 μC/OS-Ⅱ的任务管理和调度

μC/OS-Ⅱ可以管理最多 64 个任务，其中优先级最高的 2 个任务和优先级最低的 2 个任务保留给系统使用，用户一共可以使用 56 个任务。任务的优先级越高，反映优先级的值就越低。因为在 μC/OS-Ⅱ中一个优先级只能分配给一个任务使用，所以可以用优先级来标识任务。在 μC/OS-Ⅱ的任务管理中，最重要的数据结构是任务控制块 OS_TCB。通过这个数据结构，内核可以得知和控制所有任务的状态。[17]

```
typedef struct os_tcb {
    OS_STK        *OSTCBStkPtr;      //当前堆栈指针
    void          *OSTCBExtPtr;      //指向用户定义数据的指针
    OS_STK        *OSTCBStkBottom;   //栈底指针
    INT32U         OSTCBStkSize;     //堆栈大小
    INT16U         OSTCBOpt;         //任务选项
    INT16U        OSTCBId;           //任务 ID
    struct os_tcb *OSTCBNext;        //指向 TCB 链中下一个 TCB 块的指针
    struct os_tcb *OSTCBPrev;        //指向 TCB 链中前一个 TCB 块的指针
    ......
```

```
    INT16U          OSTCBDly;       //延时时间
    INT8U           OSTCBStat;      //任务状态
    INT8U           OSTCBPrio;      //任务优先级
    INT8U           OSTCBX;         //根据任务优先级指出任务组
    INT8U           OSTCBY;         //根据任务优先级指出就绪表
    INT8U           OSTCBBitX;      //就绪表的位掩码
    INT8U           OSTCBBitY;      //就绪组的位掩码
    ......
} OS_TCB;
```

一旦任务建立了，任务控制块 OS_TCB 将被赋值。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，任务控制块确保任务从当时被中断的那一点继续执行。OS_TCB 全部驻留在 RAM 中。在对任务进行创建，删除，挂起以及恢复等等的操作过程中都会使用到数据结构 OS_TCB。

在进行任务调度时，作为实时操作系统，μC/OS-II采用的是可剥夺型实时多任务内核。可剥夺型的实时内核在任何时刻都会运行已经就绪的最高优先级的任务。μC/OS-II中最多可以支持64个任务，分别对应优先级0～63，其中0为最高优先级。

调度工作的内容可以分为两部分：最高优先级任务的寻找以及任务的切换。

- 寻找最高优先级任务：在μC/OS-II中，最高优先级任务的寻找是通过建立就绪任务表来实现的，每个任务的就绪态标志都放入就绪表中，当发生任务切换时，内核会在就绪表中查找已经就绪的最高优先级的任务进行切换。与就绪表有关的数据结构和变量有：prio（8位）、OSRdyGrp（8位）、OSRdyTbl[]（8位）、OSMapTbl[]和OSUnMapTbl[]等。μC/OS-II的就绪表是一个8×8的二维表，它将64个优先级分成8组，第一行（组）从左到右依次是优先级0到优先级7，依次类推；OSRdyGrp一共有8位，分别对应就绪表的8个组（第0行到第7行），当第n组中的第m个任务就绪时，则OSRdyGrp的第n位置1，同时就绪表OSRdyTbl[n]的第m位置1；OSMapTbl[]是在ROM中的（在文件OS_CORE.C中定义）屏蔽字，用于限制OSRdyTbl[]数组的元素下标在0到7之间；因为就绪表是8×8的，所以prio[2:0]可以用来表示任务的优先级在就绪表中的列，prio[5:3] 可以用来表示任务的优先级在就绪表中所处的行；OSUnMapTbl[]是优先级判定表（在OS_CORE.C中定义），可以通过它来找到就绪表中优先级最高的就绪任务。可以对OSRdyGrp和OSRdyTbl进行操作来改变就绪表。
  - ✧ 使任务进入就绪表的代码如下所示：

```
OSRdyGrp               |= OSMapTbl[prio >> 3];//决定任务所在组
OSRdyTbl[prio >> 3]  |= OSMapTbl[prio & 0x07];
```

✧ 从就绪表中删除一个任务的代码如下所示：

```
if ((OSRdyTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0)
    OSRdyGrp &= ~OSMapTbl[prio >> 3];
```

✧ 查找就绪表中优先级最高任务的代码如下所示：

```
y    = OSUnMapTbl[OSRdyGrp];
x    = OSUnMapTbl[OSRdyTbl[y]];
prio = (y << 3) + x;
```

使用位图的方式来查找就绪任务可以加快查找的时间，优化性能。

➢ μC/OS-Ⅱ中任务的调度是由函数OSSched()完成的。函数的结构如下：

```
void OSSched(void){
    关中断;
    if(不是中断嵌套并且系统可以被调度){
        确定优先级最高的任务(即赋值给OSTCBHighRdy );
        if(最高级的任务不是当前的任务){
            调用OS_TASK_SW();
        }
    }
    开中断;
}
```

任务调度模块首先用变量OSTCBHighRdy记录就绪任务表中当前最高优先级任务的TCB地址，然后调用OS_TASK_SW()函数来进行任务切换。

4.2 MKRTOS 的任务管理

4.2.1 任务控制块

任务控制块是一个数据结构。当任务的 CPU 使用权被剥夺时，MKRTOS 用它来保存该任务的状态。当任务重新得到 CPU 使用权时，任务控制块能确保任务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。任务建立的时候，OS_TCB 就被初始化了。

在 MKRTOS 中，每一个任务都有一个任务控制块，任务控制块的数据结构如

下所示：

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;     //指向栈顶的指针
    void            *OSTCBExtPtr;     //指向用户定义数据的指针
    OS_STK          *OSTCBStkBottom;
    INT32U           OSTCBStkSize;
    INT16U           OSTCBOpt;
    INT16U           OSTCBId;
    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;
    INT16U           OSTCBDly;
    INT8U            OSTCBStat;
    INT8U            OSTCBPrio;
} OS_TCB;
```

- OSTCBStkPtr 是指向当前任务栈顶的指针。MKRTOS 允许每个任务有自己的栈，尤为重要的是，每个任务的栈的容量可以是任意的。有些商业内核要求所有任务栈的容量都一样，除非用户写一个复杂的接口函数来改变之。这种限制浪费了 RAM，当各任务需要的栈空间不同时，也得按任务中预期栈容量需求最多的来分配栈空间。
- OSTCBExtPtr 指向用户定义的任务控制块扩展。用户可以扩展任务控制块而不必修改内核的源代码。
- OSTCBStkBottom 是指向任务栈底的指针。如果微处理器的栈指针是递减的，即栈存储器从高地址向低地址方向分配，则 OSTCBStkBottom 指向任务使用的栈空间的最低地址。类似地，如果微处理器的栈是从低地址向高地址递增型的，则 OSTCBStkBottom 指向任务可以使用的栈空间的最高地址。
- OSTCBStkSize 中存有的是栈中可容纳的指针数目而不是用字节（Byte）表示的栈容量总数。也就是说，如果栈中可以保存 1,000 个入口地址，每个地址宽度是 32 位的，则实际栈容量是 4,000 字节。同样是 1,000 个入口地址，如果每个地址宽度是 16 位的，则总栈容量只有 2,000 字节。
- OSTCBId 用于存储任务的识别码。
- OSTCBNext 和 OSTCBPrev 用于任务控制块 OS_TCB 的双重链接，每个任务的任务控制块 OS_TCB 在任务建立的时候被链接到链表中，在任务删除的时候从链表中被删除。双重连接的链表使得任一成员都能被快速插入或删除。
- OSTCBDly 当需要把任务延时若干时钟节拍时要用到这个变量，或者需要把任

务挂起一段时间以等待某事件的发生，这种等待是有超时限制的。在这种情况下，这个变量保存的是任务允许等待事件发生的最多时钟节拍数。如果这个变量为 0，表示任务不延时，或者表示等待事件发生的时间没有限制。

- OSTCBPrio 是任务优先级。OSTCBPrio 越小，任务的优先级越高。

在创建任务的同时，任务控制块将会被赋值，在任务的整个生命周期中，任务控制块即代表任务，在就绪表中存放的也是指向任务控制的指针。任务控制块的赋值如下所示：

```
OS_TCB *pTMPTCB = new OS_TCB;
if (pTMPTCB != NULL) {
    pTMPTCB->OSTCBStkPtr = ptos;
    pTMPTCB->OSTCBExtPtr = NULL;
    pTMPTCB->OSTCBStkBottom = NULL;
    pTMPTCB->OSTCBStkSize = 0;
    pTMPTCB->OSTCBOpt = 0;
    pTMPTCB->OSTCBId  = i;//id;//OSTaskMaxID ++;
    pTMPTCB->OSTCBNext = NULL;
    pTMPTCB->OSTCBPrev = NULL;
    pTMPTCB->OSTCBDly = 0;
    pTMPTCB->OSTCBPrio = prio;
}
```

4.2.2 任务的建立

在 MKRTOS 中用户可以通过传递任务地址和其它参数到 OSTaskCreate()中来创建任务。任务可以在多任务调度开始前建立，也可以在其它任务的执行过程中被建立。在开始多任务调度(即调用 OSStart())前，用户必须建立至少一个任务，并且任务不能由中断服务程序(ISR)来建立。

在 MKRTOS 中，任务由类 CTask 表示。创建一个任务就是创建一个类，一个任务的信息全部被封装在该任务的类中。在创建一个任务时，cTask 会自动调用它的构造函数来对该任务进行初始化，其中 task 是任务代码的指针，pdata 是当任务开始执行时传递给任务的参数的指针，ptos 是分配给任务的堆栈的栈顶指针，prio 是分配给任务的优先级，Cycle 是周期任务的周期，taskid 是任务的标识号，stk_size 是堆栈的大小，opt 是可选项。在建立任务的时候，我们需要知道处理器的堆栈是从上往下递增还是从下往上递增的，因为我们需要把栈顶的指针传递给任务的构造函数。在创建任务的时候我们需要注意该任务是否是周

期任务，对于周期任务，我们需要在创建任务的时候设置 Cycle 为非负的整数；而对于非周期任务，我们通过将 Cycle 的值设置为-1 来表示该任务是非周期任务。类 Ctask 的定义如下所示：

```
class CTask
{
    public:
        CTask(void (*task)(),void *pdata,OS_STK *ptos,INT8U prio,
              INT8U  Cycle,INT16U  taskid,INT32U  stk_size,INT16U
              opt);
    private:
        OS_STK *OSTaskStkInit (void (*task)(), void *pdata, OS_STK
            *ptos, INT16U opt);
};
```

在创建任务时，每个任务都会被分配自己的堆栈空间。堆栈必须声明为 OS_STK 类型，并且由连续的内存空间组成。下列代码用来建立任务的堆栈。

```
typedef INT32U  OS_STK; /* 堆栈是 32 位宽度*/
OS_STK * CTask::OSTaskStkInit (void (*task)(), void *pdata, OS_STK
*ptos, INT16U opt)
{
    OS_STK *stk;
    opt= opt;     /* 'opt'  没有使用。作用是避免编译器警告     */
    stk=ptos;      /* 获取堆栈指针*/

    /* 建立任务环境，ADS1.2 使用满递减堆栈          */
    *stk = (OS_STK) task;                    /*  pc  */
    *--stk = (OS_STK) task;                  /*  lr  */
    *--stk = 0;                              /*  r12  */
    *--stk = 0;                              /*  r11  */
    *--stk = 0;                              /*  r10  */
    *--stk = 0;                              /*  r9   */
    *--stk = 0;                              /*  r8   */
    *--stk = 0;                              /*  r7   */
    *--stk = 0;                              /*  r6   */
    *--stk = 0;                              /*  r5   */
```

```
    *--stk = 0;                                    /*  r4    */
    *--stk = 0;                                    /*  r3    */
    *--stk = 0;                                    /*  r2    */
    *--stk = 0;                                    /*  r1    */
    *--stk = (unsigned int) pdata;      /*  r0,第一个参数使用 R0 传递    */
    *--stk = (USER_USING_MODE|0x00); /*  spsr，允许 IRQ, FIQ 中断    */
    *--stk = 0;                                /*  关中断计数器 OsEnterSum;    */
    return (stk);
}
```

4.2.3 任务的删除

当一个任务结束了它的生命周期之后，它将会被内核从就绪队列中删除。在 taskManager.h 中定义了 endTask（）函数可以用来删除任务，程序如下所示：

```
void taskManager::endTask()
{
    OS.OSEndCurTask();
}
```

函数 endTask()会调用类 Ckernel 中的 OSEndCurTask()函数来完成删除任务的工作，实际上 endTask 只是简单的调用了 OSEndCurTask，而将任务从就绪表中删除等工作都是在 OSEndCurTask 中完成的。程序如下所示：

```
void CKernel::OSEndCurTask()
{
    OS_TCB_CXH *pTMP;
    INT8U i = MAXPRIO;
    /*从就绪表中将任务删除*/
    pTMP = OSTCBPrioTblReadyCXH[OSPrioCur];
    if (pTMP == NULL) {
        return ;
    }
    if (pTMP->sum == 1) {
        delete
        OSTCBPrioTblReadyCXH[OSPrioCur]->pOSTCBFirst->OSTCBNext;
        OSTCBPrioTblReadyCXH[OSPrioCur]->pOSTCBFirst = NULL;
        OSTCBPrioTblReadyCXH[OSPrioCur]->pOSTCBLast = NULL;
```

```
            pTMP->sum --;
        } else {
            OSTCBPrioTblReadyCXH[OSPrioCur]->pOSTCBFirst=
                OSTCBPrioTblReadyCXH[OSPrioCur]->pOSTCBFirst->OSTCBNext;
            delete
                OSTCBPrioTblReadyCXH[OSPrioCur]->pOSTCBFirst->OSTCBPrev;
            OSTCBPrioTblReadyCXH[OSPrioCur]->pOSTCBFirst->OSTCBPrev=
                NULL;
            pTMP->sum --;
        }
        delete pTMP;
        /*找到就绪表中优先级最高的就绪任务，准备做任务切换*/
        i = OSFindHighestRdy(0);
        if (i < MAXPRIO) {
            OSPrioCur=i;
            OSStartHighRdy(OSTCBPrioTblReadyCXH[OSPrioCur]->pOSTCBFirst->
            OSTCBStkPtr);
        }
}
```

4.2.4 任务的优先级

MKRTOS 中的优先级机制有两个特点：第一，没有设置优先级的上限；第二，允许几个不同的任务被赋予同一个优先级。MKRTOS 的优先级列表如图 4-1 所示：

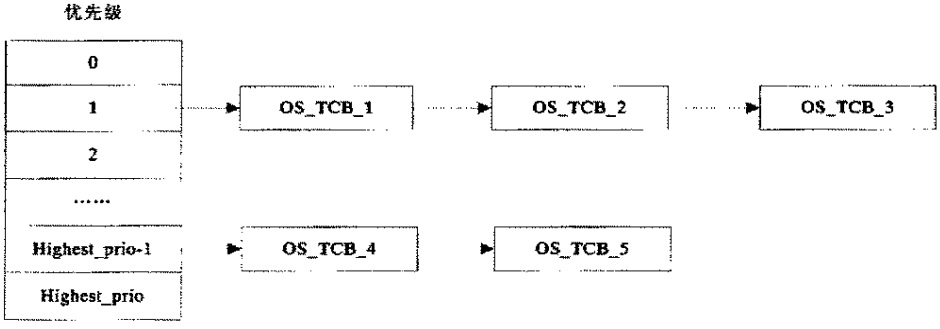

图 4-1 优先级列表示意图

图 4-1 中的 OS_TCB_1、OS_TCB_2、OS_TCB_3 分别是任务 1，任务 2 和任务 3

的任务控制块，如图 4-1 中所示，任务 1、任务 2 和任务 3 具有相同的优先级（优先级 1），任务 4 和任务 5 也具有相同的优先级（优先级 highest_prio-1）。MKRTOS 中使用数据结构 os_tcb_cxh 来表示优先级表中的每一个优先级项。

```
typedef struct os_tcb_cxh {
    INT8U  sum;
    OS_TCB *pOSTCBFirst;
    OS_TCB *pOSTCBLast;
} OS_TCB_CXH;
OS_TCB_CXH *OSTCBPrioTblReadyCXH[MAXPRIO];
```

其中，sum 用来表示有多少个任务具有同一个优先级；在优先级表中，具有相同优先级任务的 os_tcb 被连在一起形成一个链表，pOSTCBFirst 指向链表中的第一个任务的 os_tcb，pOSTCBLast 指向链表中最后一个任务的 os_tcb。

- 下列代码可以将一个任务的 os_tcb 按照任务的优先级加入任务就绪表：

```
if (OSTCBPrioTblReadyCXH[prio]->pOSTCBLast == NULL) {
    OSTCBPrioTblReadyCXH[prio]->pOSTCBLast = pTMPTCB;
    OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst = pTMPTCB;
}
else {/*如果该优先级下已经有任务存在，则将新的任务的 os_tcb 加入到链表的尾端*/
    OSTCBPrioTblReadyCXH[prio]->pOSTCBLast->OSTCBNext = pTMPTCB;
    pTMPTCB->OSTCBPrev = OSTCBPrioTblReadyCXH[prio]->pOSTCBLast;
    OSTCBPrioTblReadyCXH[prio]->pOSTCBLast = pTMPTCB;
}
```

- 根据优先级删除就绪表中的任务：先根据任务的优先级找到任务所在的优先级链表，再在链表中找到要删除的任务

```
pTMP = OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst;
while (pTMP != NULL ) {
   if (pTMP->OSTCBId == id) {
      if (OSTCBPrioTblReadyCXH[prio]->sum == 1) {/*如果该优先级下
            只有一个任务*/
         OSTCBPrioTblReadyCXH[prio]->pOSTCBLast  = NULL;
         OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst = NULL;
      }
      else if(OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst == pTMP){
```

```
        /*如果该优先级链表的第一个任务就是要删除的任务*/
          OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst= pTMP->OSTCBNext;
          OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst->OSTCBPrev = NULL;
        }
        else if (OSTCBPrioTblReadyCXH[prio]->pOSTCBLast == pTMP) {
        /*如果该优先级链表的最后一个任务是要删除的任务*/
          OSTCBPrioTblReadyCXH[prio]->pOSTCBLast = pTMP->OSTCBPrev;
          OSTCBPrioTblReadyCXH[prio]->pOSTCBLast->OSTCBNext = NULL;
        }
        else {
          pTMP->OSTCBPrev->OSTCBNext = pTMP->OSTCBNext;
          pTMP->OSTCBNext->OSTCBPrev = pTMP->OSTCBPrev;
        }
        OSTCBPrioTblReadyCXH[prio]->sum --;
        delete pTMP;
        pTMP = NULL;
        bTMP = TRUE;
      }
      else {//继续按照优先级在就绪表中查找要删数的任务
          pTMP = pTMP->OSTCBNext;
      }
    }
```

➢ 查找就绪表中优先级最高的任务

```
while (i < MAXPRIO) {
            if (OSTCBPrioTblReadyCXH[i]->pOSTCBFirst == NULL) {
                i ++;
            } else {
                return i;
            }
        }
        return MAXPRIO;
}
```

4.2.5 改变任务的优先级

4.2.5.1 优先级反转

每个任务都有优先级。任务越重要，赋予的优先级应越高。在应用程序的执行过程中，如果任务的优先级保持不变，则称之为静态优先级。在静态优先级系统中，任务以及它们的时间约束在程序编译时是已知的。反之，如果在应用程序执行过程中，任务的优先级是可变的，则称之为动态优先级。

在优先级驱动的系统中，如果在运行过程中，任务的优先级是固定不变的，那么很可能会产生优先级反转的问题。

假设有3个任务，它们的优先级从高到低依次为：任务1、任务2和任务3。任务1和任务2处于挂起状态，等待某一事件的发生，任务3正在运行。在某一时刻，任务3申请并得到了共享资源的信号量，并开始使用共享资源。由于任务1优先级高，它等待的事件到来之后剥夺了任务3的CPU使用权，任务1开始运行。运行过程中任务1也要使用任务3正在使用着的资源，由于该资源的信号量还被任务3占用着，因此任务1只能进入挂起状态（图4-2中t2处），等待任务3释放该信号量。任务3得以继续运行。由于任务2的优先级高于任务3，当任务2等待的事件发生后，任务2剥夺了任务3的CPU的使用权并开始运行。处理它该处理的事件，直到处理完之后将CPU控制权还给任务3。任务3接着运行，直到释放那个共享资源的信号量。然后任务1才能够得到信号量继续运行。在这种情况下，任务1优先级实际上降到了任务3的优先级水平。由于任务2剥夺任务3的CPU使用权，使任务1的状况更加恶化，任务2使任务1增加了额外的延迟时间。在这种情况下，任务1和任务2的优先级发生了反转。优先级反转的情况如图4-2所示。[23]

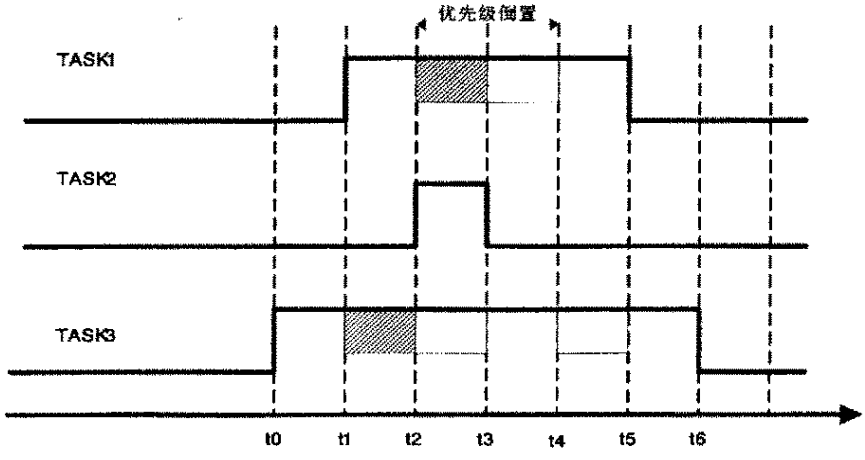

图 4-2 优先级反转示意图

优先级反转问题的解决方法大致有2种：一种是优先级继承(inheritance)策略；另一种是优先级极限(ceilings)策略。

优先级继承的基本思想是：当高优先级任务在等待低优先级的任务占有的信号量时，让低优先级任务继承高优先级任务的优先级，即把低优先级任务的优先级提高到高优先级任务的优先级；当低优先级任务释放了高优先级任务等待的信号量后，立即把它恢复到原来的优先级。

当高优先级任务task1想要进入临界区时，由于低优先级任务task3占有这个临界资源的信号量，导致task1被阻塞，因此系统把task3的优先级升到task1的优先级级别，此时即使task2处于就绪状态也不能够被调度执行。当task3释放task1需要的信号量后，系统立即把task3的优先权降到原来的级别，来保证task1和task2的正常执行。使用优先级继承策略的任务运行情况如图4-3所示。目前，有许多RTOS是采用这种方法来防止优先级反转的，如大家比较熟悉的业界有名的WindRiver公司的VXWORKS。

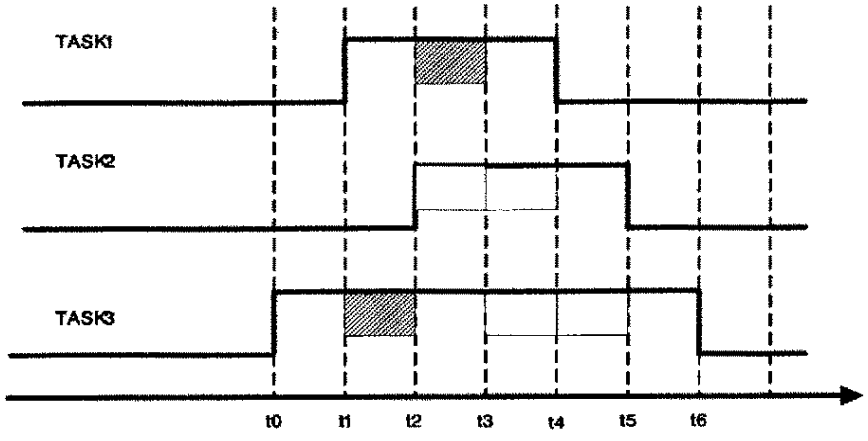

图 4-3 优先级继承示意图

在优先权极限方案中，系统把每一个临界资源与1个极限优先级相联系。这个极限优先权等于系统此时最高优先权加1。当1个任务进入临界区时，系统便把该临界区的极限优先级传递给这个任务，使这个任务成为申请该临界资源的所有任务中优先级最高的任务；当这个任务退出临界区后，系统立即把它的优先级恢复正常，从而保证系统不会出现优先权反转的情况。如上例中，当task3进入临界区时，立即把它的优先级升高到极限优先级，保证task3此时能尽快退出临界区，进而释放其占有的信号量。当高优先级任务task1执行的时候就不会出现等待低优先级任务task3释放信号量而被阻塞的情况，从而保证不会出现优先级反转。

4.2.5.2 改变任务的优先级

为了避免优先级反转的现象，在 MKRTOS 中我们预留了一个函数接口 changeTaskPrio()，通过这个函数可以动态的改变任务的优先级。函数 changeTaskPrio()在任务管理类——taskManager 中定义。主要代码如下所示：

```
void taskManager::changeTaskPrio(INT8U newprio,INT16U taskid)
{
    ......
    if(newprio>MAXPRIO)                                          ①
          return;
    tmpoldprio = OS.OSPrioCur;                                   ②
    pTMPostcb = new OS_TCB;
    ret1 = OS.OSGetCurOSTCB(tmpoldprio,0,pTMPostcb);             ③
    if (ret1 == 0) {
        retid = pTMPostcb->OSTCBId;                              ④
        ret2  = OS.OSSetOSTCB(tmpoldprio,retid,0,newprio);       ⑤
        if (ret2 ==0) {
            Heighprio = OS.OSFindHighestRdy(0);                  ⑥
            OS.OSPrioHeighest = Heighprio;
            if (newprio < OS.OSPrioHeighest) {
                OS.OSPrioHeighest = newprio;
                OS.OSPrioCur = newprio;
                OS.OS_Sched();
            }
        }
    ......
}
```

函数 changeTaskPrio 用来改变正在运行的任务的优先级。传递给函数 changeTaskPrio 的参数一共有 2 个，分别是任务想要改变的新优先级 newprio 和任务的 id 标识号。在 changeTaskPrio 中，首先会检查想要分配给任务的新优先级是否在规定的范围内，如果新的优先级超出了规定的范围，程序将会立刻返回（①）。判断完毕之后，程序会从当前正在运行的任务的 os_tcb 中读出任务的优先级，然后根据该优先级，通过 OS.OSGetCurOSTCB 将任务的 os_tcb 读取到 pTMPostcb 中（③），并且从中得到当前任务的 id。得到当前的任务的优先级之后，程序就可以通过调用 OS.OSSetOSTCB 将新的优先级更新到任务的 os_tcb 中

了（⑤）。在OS.OSSetOSTCB中主要做三个工作：1）将拥有原先优先级的任务从就绪表中删除；2）更新任务的优先级；3）将更新优先级后的任务重新插入就绪表中。最后程序从更新了的就绪表中查找优先级最高的就绪任务，准备做任务切换（⑥）。

4.3 MKRTOS的任务调度

一个操作系统的实时性与它的调度策略密切相关。在操作系统领域，对调度算法的研究始终是一个热点，调度算法主要分为两大主要的类别：实时调度算法和非实时调度算法。顾名思义，实时调度算法的目标就是实时性。所谓实时性，并不是指响应速度非常快，而是指有保证，就是在最坏情况下，响应时间也不会超过事先规定的一个常数。而实时系统又可以分成“硬实时系统”和“软实时系统”，所谓的“硬实时系统”是指那些在系统整个运行时间，所有的事件都必须在规定时间内响应的系统；相对的，另外一类要求大部分时间内事件在规定时间内被响应，可以容忍偶尔几次违规的实时系统，则称为“软实时系统”。

由于实时系统总是用在一些非常特定的环境中，因此也没有什么通用的“硬实时”操作系统，在设计调度算法的时候，我们主要还是考虑通用的调度算法，并尽量使之满足软实时的要求。

各种实时操作系统的实时调度算法可以分为如下三种：基于优先级的调度算法、基于CPU使用比例的共享式的调度算法、以及基于时间的进程调度算法。

基于优先级的调度算法给每个进程分配一个优先级，在每次进程调度时，调度器总是调度那个具有最高优先级的任务来执行。根据不同的优先级分配方法，基于优先级的调度算法可以分为静态和动态两种类型。

4.3.1 静态优先级调度算法

静态调度算法给那些系统中得到运行的所有进程都静态地分配一个优先级。静态优先级的分配原则多种多样，其中的一种很典型的静态优先级调度算法是单调执行率算法RMS(Rate Monotonic Scheduling)。

在单调执行率算法中，任务执行的次数越是频繁，对应的优先级越是高。RMS做了一系列假设：

- 所有任务都是周期性的；
- 任务间不需要同步，没有共享资源，没有任务间数据交换等问题；
- CPU必须总是执行那个优先级最高且处于就绪态的任务。换句话说，要使用

可剥夺型调度法。

假设系统种存在3个任务R1，R2和R3；他们的执行时间分别是1，3，2；他们的周期分别是4，12，6；根据RMS可知他们的优先级分别是1，3，2。他们的执行情况如图4-4所示：[20][21]

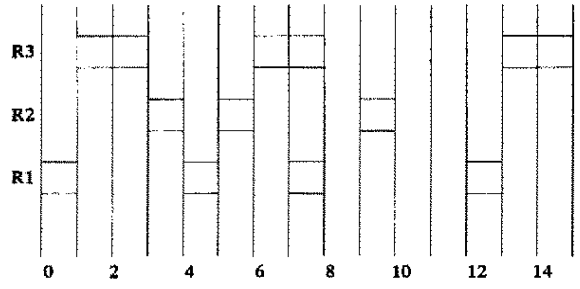

图 4-4 RMS调度算法示意图

由图4-4所示，在时间4时，R1的执行周期到达，所以R2被打断，R1得以执行；R1执行完后，又切换到R2继续执行；在时间6时，R3的执行周期到达，所以R2再次被打断，R3开始执行；依次类推。

包含n个任务的简单任务集合，可以使用RMS算法可调度的充分条件是：

$$\sum_{i} \frac{E_i}{T_i} \le n(2^{1/n} - 1) \qquad (1)$$

这里E_i是任务i最长执行时间，T_i是任务i的执行周期，E_i/T_i是任务i所需的CPU时间（即CPU利用率），n是系统中的任务数。对于无穷多个任务，$n(2^{1/n}-1)$的极限值是0.693。

RMS的缺点是CPU的使用率偏低；而RMS的优点是实现起来比较方便，因为进程的优先级是静态的，可以事先设定，在任务执行的过程中不会改变。

4.3.2 动态优先级调度算法

在动态优先级调度算法中，执行时间最靠近的任务会具有最高的优先级，因此任务的优先级会随着时间的改变而改变。动态优先级算法中一种非常典型的算法是最早截止时间调度算法 EDF。[20][21]

现在假设系统中有 3 个任务 R1，R2 和 R3，他们的执行时间分别是 1，2，1；他们的周期分别是 3，5，4；EDF 的执行情况如图 4-5 所示：

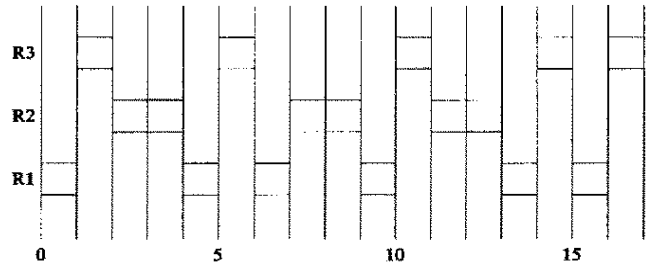

图 4-5 EDF调度算法示意图

在执行时间 4 时，因为 3 个任务的第一轮执行都结束了，因此 R1 的下一个执行时限为 6，R2 的下一个执行时限为 10，R3 的下一个执行时限是 8 ，所以此时 R1 的优先级最高，任务切换到 R1 执行；在时间 6 时，因为 R1 和 R3 的第二轮执行都结束了，因此 R1 的下一个执行时限为 9，R2 的下一个执行时限为 10，R3 的下一个执行时限是 12，所以此时还是 R1 的优先级最高，任务切换到 R1 执行，而不是 R2；到时间 7 时，R1，R2 和 R3 的执行下一个执行期限分别是 12，10 和 12，因此 R2 的优先级最高，R2 得到执行权限。以此类推。

EDF 的优点是它的 CPU 利用率几乎可以达到百分之一百；而 EDF 的缺点是，在每次调度程序启动时，都要重新计算优先级，因此在调度上负担不小，而且实现上很复杂。

简单周期任务集使用 EDF 调度算法的充分必要条件是：CPU 利用率<=1。

4.3.3 Round-Robin 调度算法

Round-Robin 算法的设计出发点是使得各个进程公平地获得在 CPU 上运行的机会。Round-Robin 算法中，所有就绪的进程组成一个链表；在被调入运行的时候，每个进程被分配一定的时间点数，称为一个时间片。在当前进程耗完这个时间片的时候，就将其从进程列表的头上取下，并插入进程队列的尾部；如果进程因为其他原因不得不放弃 CPU 时，则直接将其从进程队列上取下，并插入到相应的阻塞队列上去。无论上述种情况中的那一种，进程队列中的下一个进程被调入执行。

Round-Robin 算法的优点是算法简单，容易实现。但是，其缺点也非常明显，就是无法表现进程与进程之间的相对重要性，因此即使非常重要的进程在就绪后也必须等待所有排在其前面的进程运行完毕后才可能运行，因而事件的响应时间可能相当长。[24]

4.3.4 MKRTOS 的任务调度的实现

无论是 Round-Robin 算法还是优先级算法一般都不会单独地使用，在 MKRTOS 中使用 RMS 和时间片轮转调度相结合的调度算法。

4.3.4.1 RMS 任务调度策略的应用

因为 RMS 是基于优先级的任务调度策略，而且只是对周期任务适用，它根据任务的周期来制定任务的优先级，因此在创建任务的时候需要对周期任务和非周期任务分开处理（在 4.2.2 节中已经简单的介绍过），我们根据创建任务时变量 Cycle 的输入值来区分两种任务，创建任务的代码如下所示：

```
CTask a((void) f,(void *)0,(OS_STK *)ptos1,(INT8U) 0,
        (INT8U) 3,                  /* 任务 a 是周期任务，它的周期是 3 */
        (INT16U) 0,(INT32U) 1,(INT16U) 0);
CTask b((void) g,(void *)0,(OS_STK *)ptos2,
        (INT8U) 1,                  /*设定任务优先级是 1*/
        (INT8U) -1,                 /* 任务 b 是非周期任务 */
        (INT16U) 0,(INT32U) 1,(INT16U) 0);
```

根据 Cycle 的值的不同，我们在创建任务时会赋予他们不同的优先级，如下所示：

```
if(Cycle!=-1)
        pTMPTCB->OSTCBPrio = Cycle+HIGHPRIO;
else
        pTMPTCB->OSTCBPrio = prio;
```

对于周期任务，我们将任务的周期作为任务的优先级，根据 RMS 的定义，任务的周期越短，则他们的优先级越高；在 MKRTOS 中，优先级的值越是小，则它对应的优先级级别越是高，所以使用周期任务的周期作为任务的优先级是完全可行的。非周期任务的优先级由用户自己设定，因为可能会出现优先级很高的突发性非周期任务，所以我们预留了一些优先级（Cycle+HIGHPRIO）给这些任务使用。

RMS 的调度策略的实现比较简单，因为它是基于优先级的静态的调度策略，在任务切换时只需要查找就绪表，找到优先级最高的就绪任务进行切换就可以了。RMS 调度算法的实现如下所示：

```
void CKernel::OS_Sched()
{
    INT8U curPrio;
```

```
        OSCtxSwCtx++;                          //任务上下文切换计数器
        curPrio = OSPrioCur;
        OSPrioCur = OSPrioHeighest;
        if (OSPrioCur < curPrio || OS.OSRRSched){
            OS_TASK_SW(OSTCBPrioTblReadyCXH[OSPrioHeighest]->pOSTCBF
            irst->OSTCBStkPtr,OSCurTCB);
        }
    }
```

4.3.4.2 时间片轮转调度算法的应用

在 MKRTOS 中，同一个优先级可以分配给多个任务共同拥有，因此单是使用 RMS 很难保证相同优先级任务之间调度的实时性，因此我们引入了时间片轮转调度算法对相同优先级的多个任务进行调度。

首先，我们在每一个任务的 OS_TCB 中增加了 TimeSlice 变量，用来记录任务分配到的时间片，每一个任务在建立的时候都会被赋予一个 TIMESLICE 值（系统中定义），当该任务运行时，每经过一个系统时间，该值都会被减 1，一直到 TimeSlice 的值为 0 时，该任务会被同等优先级的其他任务所代替（如果没有高优先级任务打断的话）。

然后，我们在内核中增加函数 OS_RRSched()，用来判断是否需要进行同等优先级任务之间的切换，该函数的代码如下所示：

```
    void CKernel::OS_RRSched(INT8U prio)
    {
       OS_TCB *pTMPTCB = new OS_TCB;
       pTMPTCB=OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst->OSTCBNext;
       OS.OSRRSched=0; /* OSRRSched 为 1 表示要进行任务切换，
                          为 0 表示不需要 */
       /* 如果时间片用完，则找到同等优先级的下一个任务的 TCB */
       if(OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst->TimeSlice==0)
       {
          if(pTMPTCB!=NULL)
          {
                OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst->TimeSlice=
                TIMESLICE;              //重新设定任务的时间片

             /* 将时间片用完的任务放置到队列的尾端 */
```

```
                OSTCBPrioTblReadyCXH[prio]->pOSTCBLast->OSTCBNext=O
                STCBPrioTblReadyCXH[prio]->pOSTCBFirst;
                OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst->OSTCBNext=
                NULL;
            /* 设置新的任务队列的头部 */
                OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst=pTMPTCB;
            /* 需要进行任务切换 */
                OS.OSRRSched=1;
        }
        else
        {
                OSTCBPrioTblReadyCXH[prio]->pOSTCBFirst->TimeSlice=
                TIMESLICE;
        }
    }
}
```

当一个任务的时间片用完之后，程序会判断在同一优先级级别的就绪队列中是否有其他正在等待的任务，如果有的话就将当前任务放到就绪队列的尾段，并重新设置该任务的 TimeSlice 的值；然后将就绪队列的头指针指向下一个就绪的任务；最后设置 OS.OSRRSched=1，用来告诉调度器 OS_Sched()需要进行同一优先级级别的任务间的切换。如果就绪队列中没有其他的任务了，则只是简单的重置 TimeSlice 的值，而且设置 OS.OSRRSched=0，不进行任务切换。TimeSlice 的值保存在任务的 OS_TCB 中，因此就算任务被高优先级的任务打断，TimeSlice 的值也可以保证不会被改变。

4.4 本章小结

本章主要介绍了 MKRTOS 的任务管理和任务调度的实现机制。首先，本章对 μC/OS-II 的任务管理和任务调度进行了分析；随后介绍了 MKRTOS 的任务管理机制，并着重分析介绍了 MKRTOS 的任务管理的核心部分的实现方法，其中包括任务控制块，任务的建立函数，任务的删除函数，任务的优先级分配机制和改变任务的优先级的函数等；最后本章介绍了 MKRTOS 的任务调度策略，指出了 MKRTOS 使用的任务调度算法是结合 RMS 和 Round-Robin 的调度算法，并给出了主要的实现代码。

第 5 章 硬件开发平台简介

本章对 MKRTOS 将要移植的目标平台 LPC2200 和 ARM7TDMI-S 核的相关技术和知识进行了简单的介绍，主要分析了 LPC2200 实验开发平台上与移植相关的一些细节。

5.1 LPC2000 系列系统硬件结构介绍

嵌入式系统硬件平台包括 CPU、外围的控制电路、只读存储器、可读写存储器和外围设备。本项目使用的是 EasyARM2200 教学实验平台。该平台是一款功能强大的 32 位 ARM 单片机开发版，采用了 PHILIPS 公司的 ARM7TDMI-S 核，总线开放的单片机 LPC2200 具有 JTAG 调试等功能。版上提供了一些键盘，LED 和 RS232 等常用功能部件，并具有 IDE 硬件接口，CF 存储卡接口，以太网接口和 MODEM 接口等等，并且设计有外设 PACK，极大地方便了用户在 32 位 ARM 嵌入式系统领域进行开发试验。图 5-1 为 LPC2000 系列系统硬件结构图。LPC2200 系列的 ARM 芯片有 16KB 的片内 SRAM，有 4Mbit 的片外 SRAM 和 16Mbit 的片外 FLASH，可以方便用户样机开发，完全可以满足使用 C++开发的内存需求。

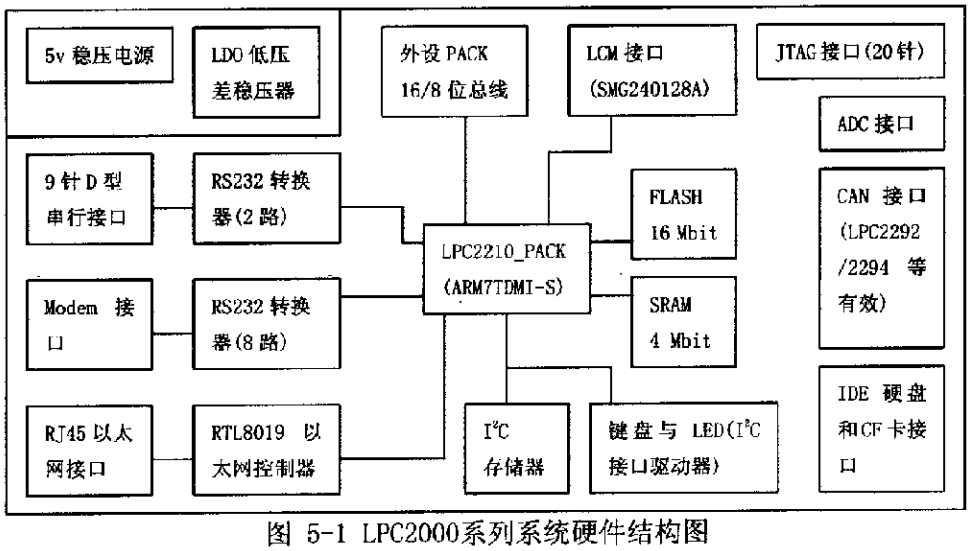

图 5-1 LPC2000系列系统硬件结构图

5.1.1 存储器寻址

5.1.1.1 存储器映射

LPC2200 的存储器可以分为：片内 FLASH 程序存储器（128K 或 256K）、片内

LPC2200 的存储器可以分为：片内 FLASH 程序存储器（128K 或 256K）、片内

静态 RAM（16K，可以用于代码和数据的存储）和片外存储器。[27][28]

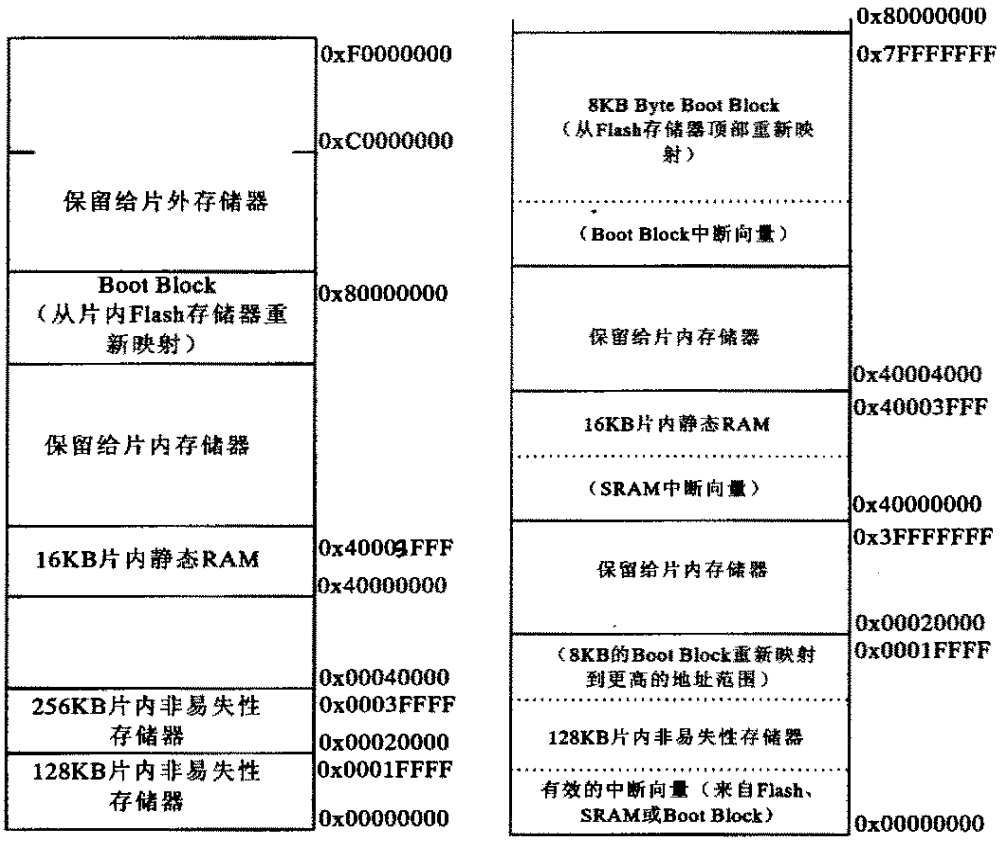

图 5-2 存储器映射图

图 5-2 中左图为系统复位后，从用户的角度看到的整个地址空间的映射。中断向量支持地址的重新映射。如果试图访问一个保留地址或未分配区域的地址，将会产生预取指令中止或数据中止异常。

存储器映射的基本概念是：每个存储器组在存储器映射过程中都有一个“物理上的”位置，它是一个地址范围，该范围内可写入程序代码。每一个存储器空间的容量都永远固定在同一个位置，这样就不需要将代码设计成在不同地址范围内运行。由于 ARM7 处理器上的中断向量的位置（地址为 0x00000000～0x0000001C），Boot Block 和 SRAM 空间的一部分需要通过重新映射来实现在不同操作模式下对中断的使用，这些模式包括：Boot 装载程序模式，用户 Flash 模式，用户 RAM 模式和用户外部模式。中断的重新映射通过存储器映射控制特性来实现。

为了与将来器件相兼容，整个 Boot Block 都被映射到片内存储器空间的顶端。以 128K 的片内 Flash 为例，重新映射后，从地址 0x00000000 开始的连续的 32 个字的空间范围用来放置中断向量表，从 0x0001E000 到 0x0001FFFF 的 8KB 的地址空间用来放置重新映射后的 Boot Block。在这种方式下，使用较大或较

小的Flash模块都不需要改变Boot Block的位置或改变其中的中断向量的映射。除了中断向量以外的存储器空间都保持固定的位置。（在图5-2中，右图所示为重新映射后的低端存储器空间）

存储器重新映射的部分允许在不同模式下处理中断，它包括中断向量区（32字节）和额外的32字节。重新映射的代码位置与地址0x00000000～0x0000003F重叠。一个位于FLASH存储器中的典型用户程序可以将整个FIQ处理程序放置在地址0x0000001C，而不需要考虑存储器的边界。包含在SRAM、外部存储器和Boot Block中的向量必须包含跳转到实际中断处理程序的分支或者其他可以跳转到中断处理程序的转移指令。使用存储器的重新映射主要有3个原因：

- 使Flash存储器中的FIQ处理程序不必考虑由于重新映射所导致的存储器边界问题；
- 用来处理代码空间中段边界仲裁的SRAM和Boot Block向量的使用大大减少；
- 为超过单字转移指令范围的跳转提供空间来保存常量。

重新映射的存储器组，包括Boot Block和中断向量，除了重新映射的地址外，仍然继续出现在它们最初的位置。

5.1.1.2 存储器映射控制

存储器映射控制用于改变从地址0x00000000开始的中断向量的映射，这就允许了运行在不同存储器空间中的代码对中断进行控制。存储器映射控制是通过存储器映射控制寄存器（MEMMAP－0xE01FC040）来实现的，可以使用MEMMAP的低2位来选择从Flash Boot Block、用户Flash或RAM中读取ARM中断向量。

存储器映射控制从处理ARM异常必须的4个数据源中选择一个使用（32字节的异常向量表加上32字节额外空间），如图5-3所示。[27]

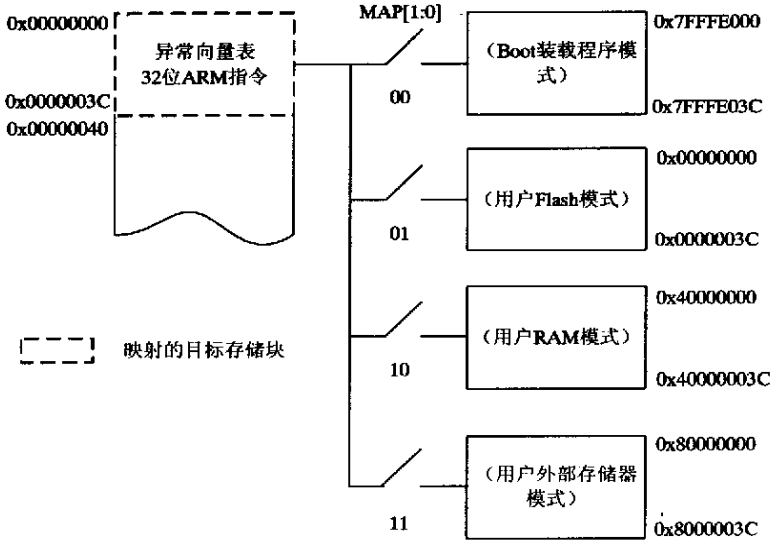

图 5-3 存储器映射控制示意图

每当产生一个软件中断请求时，ARM 内核就从 0x00000008 处取出 32 位数据，根据 MAP[1:0]值的不同，系统会使用不同的映射模式。

- 当 MEMMAP[1:0]＝11 时，即用户外部存储器模式，对地址空间 0x00000008 的操作实际上是对 0x80000008 单元进行操作；
- 当 MEMMAP[1:0]＝10 时，即用户 RAM 模式，对地址空间 0x00000008 的操作实际上是对 0x40000008 单元进行操作；
- 当 MEMMAP[1:0]＝01 时，即用户 Flash 模式，中断向量不需要重新映射，因为它位于 Flash 中；
- 当 MEMMAP[1:0]＝00 时，即 Boot 装载程序模式，对地址空间 0x00000008 的操作实际上是对 0x7FFFE008 单元进行操作(Boot Block 在片内 Flash 存储器中重新映射)；

为了使系统能够实现基本的工作，必须在进入 main 函数之前，对系统进行一些基本的初始化工作，这些工作由 TargetResetInit 函数完成，他们包括存储器的映射和时钟设置等等。使用 LPC2200 模版建立工程时，编译器会根据用户选择的 target 项来预定义__DEBUG、__OUT_CHIP 或__IN_CHIP 三个宏中的一个，不同的 target 对应着不同的工程配置，这样当配置改变时就不用改变代码了。下列代码设置存储器的映射方式。

```
#ifdef __DEBUG
    MEMMAP = 0x3;                    //remap
#endif
#ifdef __OUT_CHIP
    MEMMAP = 0x3;                    //remap
#endif
#ifdef __IN_CHIP
    MEMMAP = 0x1;                    //remap
#endif
```

在芯片复位的时候，MEMMAP＝0，Boot 装载程序将会被启动，Boot 装载程序会检查 P0.14 端口的状态和用户的异常向量表，判断是进入 ISP 状态还是启动用户程序，若启动用户程序，则自动设置 MEMMAP＝1（片内 Flash 启动）或 MEMMAP＝3（片外存储器启动）。若用户程序需要随时更改异常向量表，可以将异常向量表（64 字节）复制到片内 RAM 上（0x40000000），然后设置 MEMMAP＝2 进行重新映射，0x40000000 地址上的向量表就可以更改了。复制向量表的程序如下所示：

```
uint8    i;
```

```
uint32   *cp1,*cp2;
extern void Reset(void);
cp1 = (unit32 *) Reset;
cp2 =  uint32 *  0x40000000;
for(i=0;i<16;i++)
{
    *cp2++ = *cp1++;
}
MEMMAP = 2;
```

当使用片内 RAM 进行调试时，需要设置 MEMMAP=2，使保存在 0x40000000 地址上的异常向量表映射到 0x00000000 地址上。

5.1.1.3 启动代码相关部分

ARM芯片复位后，系统进入特权模式、ARM状态，PC寄存器的值是0x00000000，所以必须要保证用户的向量表代码定位在 0x00000000 处，或者映射到 0x00000000 处（例如向量表代码在 0x80000000 处，通过存储器映射，访问 0x00000000 就是访问 0x80000000）。向量表的定义如下列程序段所示：

```
Reset
        LDR     PC, ResetAddr
        LDR     PC, UndefinedAddr
        LDR     PC, SWI_Addr
        LDR     PC, PrefetchAddr
        LDR     PC, DataAbortAddr
        DCD     0xb9205f80
        LDR     PC, [PC, #-0xff0]
        LDR     PC, FIQ_Addr
ResetAddr       DCD     ResetInit
UndefinedAddr   DCD     Undefined
SWI_Addr        DCD     SoftwareInterrupt
PrefetchAddr    DCD     PrefetchAbort
DataAbortAddr   DCD     DataAbort
Nouse           DCD     0
IRQ_Addr        DCD     0
FIQ_Addr        DCD     FIQ_Handler
```

向量从上到下依次为复位、未定义指令异常、软件中断、预取中止，预取数

据中止、保留的异常位置、IRQ 和 FIQ。使用 LDR 指令而不是 B 指令跳转的原因有两个：

- LDR 指令可以全地址范围跳转，而 B 指令不行；
- 芯片具有 ReMap 功能。当向量表位于 RAM 中时，用 B 指令不能跳转到正确的位置。

LPC2200 的工程模版使用了 ADS 的分散加载机制，只要编写相应的分散加载描述文件，即可以将代码段、数据段分别定位到指定地址上。在 LPC2200 的工程模版中，使用片外 FLASH 启动程序的分散加载描述文件如下所示：

```
ROM_LOAD 0x80000000
{
    ROM_EXEC 0x80000000
    {
        Startup.o (vectors, +First)
        * (+RO)
    }
    ......
}
```

ROM_LOAD 是加载区的名称，0x80000000 表示加载区的起始地址，既是存放程序代码的位置，也可以在后面继续添加加载区的空间大小，如“ROM_LOAD 0x80000000 0x20000”。ROM_EXEC 描述了执行区的地址，放在第一块定义，其起始地址、空间大小必须要和加载区的起始地址、空间大小相一致。Startup.o (vectors, +First)表示从起始地址开始放置向量表，其中 Startup.o 为 Startup.s 的目标文件，接着放置其他代码（即“* (+RO)”）。这样就可以将向量表定义到 0x80000000 处。

复位初始化程序 ResetInit 调用 InitStack 子程序来初始化各个模式下的堆栈，调用 TargetResetInit()函数来初始化与目标系统相关的设置，最后调用 ADS 提供的_main，初始化运行时库，并进入用户的 main()函数。

```
ResetInit
        BL      InitStack
        BL      TargetResetInit
        B       __main
```

5.1.2 复位

LPC2200 有两个复位源：RESET 引脚和看门狗复位。图 5-4 所示为 LPC2200

复位处理流程图。[27][28]

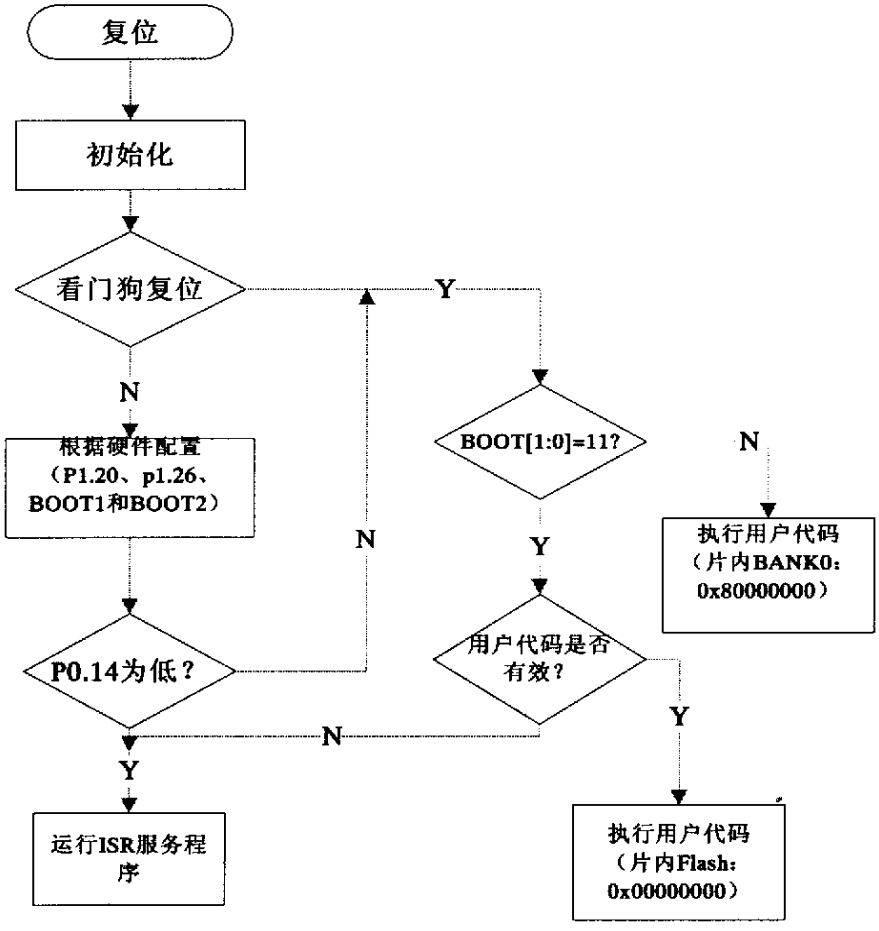

图 5-4 LPC2200复位处理流程图

当复位后执行引导装载程序时，片内引导装载程序将对 p0.14 进行判断。引脚 p0.14 有两个功能，他们分别是 UART1 数据载波检测输入端和外部中断 1 输入。当 $\overline{RESET}$ 为低时，P0.14 的低电平将强制片内引导装载程序复位后控制器件的操作，即进入 ISP 状态。

用户代码是否有效是指：只有当向量表中所有的数据 32 位累加和为 0 时（机器码累加），用户的程序才能脱机运行。可以通过定义向量表中的保留字的值，使得向量表中所有数据 32 位累加和为 0。LPC2200 启动代码的向量表及指令机器码如下列程序清单所示。

```
Reset
[0xe59ff018]          LDR     PC, ResetAddr
[0xe59ff018]          LDR     PC, UndefinedAddr
[0xe59ff018]          LDR     PC, SWI_Addr
```

```
[0xe59ff018]          LDR     PC, PrefetchAddr
[0xe59ff018]          LDR     PC, DataAbortAddr
[0xb9205f80]          DCD     0xb9205f80
[0xe51ffff0]          LDR     PC, [PC, #-0xff0]
[0xe59ff018]          LDR     PC, FIQ_Addr
```

向量表中所有数据 32 位累加和是：

0xe59ff018 + 0xe59ff018 + 0xe59ff018 + 0xe59ff018 + 0xe59ff018 + 0xb9205f80+0xe51ffff0+0xe59ff018=0x00000000

向量表中的保留字的值的得出方法是：

~(0xe59ff018 + 0xe59ff018 + 0xe59ff018 + 0xe59ff018 + 0xe59ff018 + 0xe51ffff0+0xe59ff018)+1=0xb9205f80

5.1.3 向量中断控制器

向量中断控制器（Vectored Interrupt Controller，VIC）具有 32 个中断请求输入，VIC 会将这 32 个中断输入（其中使用了 19 个中断输入，还有 13 个暂时没有使用）分配为 FIQ、向量 IRQ 或非向量 IRQ：[27][28]

- 快速中断请求（FIQ）具有最高优先级。如果分配给 FIQ 的请求多于一个，VIC 将中断请求相“或”后，向 ARM 处理器产生 FIQ 信号。当只有一个中断被分配位 FIQ 时，FIQ 服务程序可以立即启动对中断的处理就可以了；如果被分配给 FIQ 的中断多于一个时，FIQ 中断服务程序需要读取 VICFIQStatus 的内容来识别产生中断请求的 FIQ 中断源是哪一个。建议只将一个中断分配给 FIQ，因为多个 FIQ 中断源会增加中断服务程序的延迟。
- 向量 IRQ 中断具有中等优先级。一共有 16 个向量 IRQ 中断，分别对应着 16 个优先级，其中 slot0 具有最高的优先级，而 slot15 具有最低的优先级；每一个 slot 都可以被分配到 32 个中断请求中的任何一个。IRQ 中断优先级的作用是当产生多个 IRQ 中断时，VIC 会将最高优先级请求的 IRQ 服务程序的地址写入 VIC 的向量地址寄存器 VICVectAddr 中。
- 非向量 IRQ 中断的优先级最低，如果分配给非向量 IRQ 的中断多于一个，默认的中断服务程序要从 VIC 中读出 VICIRQStatus 的内容来决定响应哪一个中断源。

每一个向量 IRQ 中断都有一个对应的中断处理程序地址，但是所有的非向量 IRQ 中断都共享同一个默认的地址。如果有一个向量 IRQ 中断发出请求，VIC 会提供最高优先级 IRQ 服务程序的地址；如果发出请求的是非向量的 IRQ 中断，则提供默认的服务程序地址（所有的非向量 IRQ 共用同一个默认服务程序）。IRQ

中断入口程序可以通过读取 VIC 的向量地址寄存器 VICVectAddr 来取得该地址，然后跳转到相应的地址执行中断服务程序就可以了。

图 5-5 所示为使用 VIC 的 IRQ 中断过程。

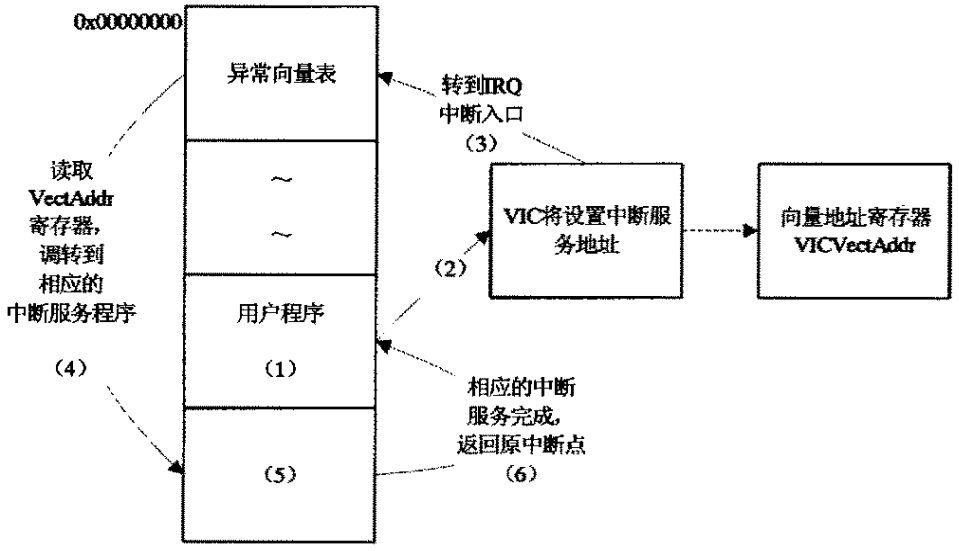

图 5-5 使用VIC的IRQ中断过程

使用 VIC 的 IRQ 中断可以分成 6 步：

（1） 用户程序初始化 VIC，是能够响应中断，然后运行用户程序；

（2） 当有 IRQ 中断产生的时候，VIC 将会根据中断源设置 VICVectAddr 为相应的中断服务程序的地址；

（3） 切换处理器的工作模式为 IRQ 模式，并跳转到 IRQ 中断入口 0x00000018 处；

（4） 异常向量表中 0x00000018 处使用指令“LDR PC, [PC, #-0xFF0]”，该指令将读出 VICVectAddr 中的内容，然后放入 PC 指针，使程序跳转到相应的中断服务程序；

（5） 在中断服务程序中进行相应的中断处理，清楚中断标志；

（6） 完成中断服务程序中的操作，返回被中断处，并切换处理器的工作模式。

5.2 ARM7TDMI－S 简介

EasyARM2200教学实验平台的CPU内核采用了PHILIPS公司的ARM7TDMI－S核。TDMI的基本含义为：T：支持16位压缩指令集Thumb；D：支持片上Debug；M：内

嵌硬件乘法器；I：嵌入式ICE,支持片上断点和调试点。[26]

图 5-6 是 ARM7TDMI 核的结构框图，ARM7TDMI-S 是 ARM7TDMI 的可综合版本，除非芯片生产厂商对 ARM7TDMI-S 进行裁剪，否则在逻辑上 ARM7TDMI-S 和 ARM7TDMI 没有太大的区别，两者的编程模式是一致的。

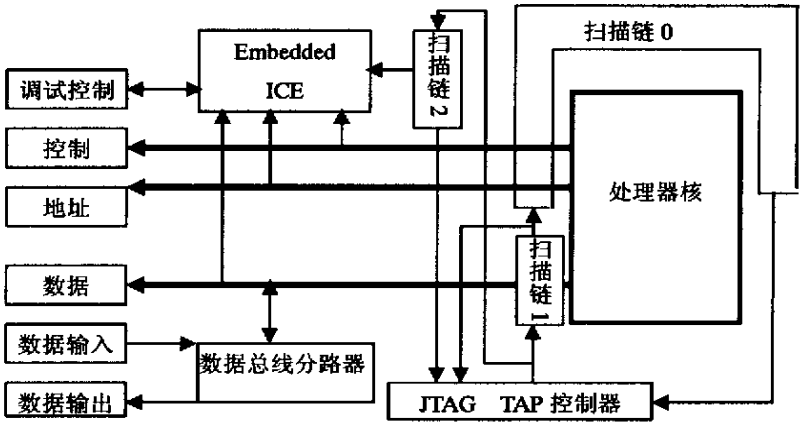

图 5-6 ARM7TDMI核的结构框图

5.2.1 指令集

32位的ARM指令集由13种基本的指令类型组成，可分为如下几类：

1) 4类分支指令用于控制程序的执行流程、指令的特权等级以及在ARM代码与Thumb代码之间进行切换。

2) 3类数据处理指令用于操作片上的ALU、桶型移位器、乘法器以完成在31个32位的通用寄存器之间的高速数据处理。

3) 3类加载/存储指令用于控制在存储器和寄存器之间的数据传输。一类为方便寻址进行了优化；第二类用于快速的上下文切换；第三类用于数据交换。

4) 3类协处理器指令用于控制外部的协处理器。几乎所有的32位ARM指令都可以条件执行。

5.2.2 三级流水线

ARM7TDMI处理器使用3级流水线来增加处理器处理指令的速度。使用流水线可以使几个操作同时进行，并行处理和存储器系统连续操作，能提供0.9MIPS/MHz的指令执行速度。

ARM7TDMI的流水线分为3个阶段：

(1) 取指：从存储器中取出指令；

(2) 译码：对指令进行译码；

(3) 执行：从寄存器中读出指令，进行相应的操作，并将结果写回寄存器。

5.2.3 操作模式

ARM7TDMI内核支持7种操作模式，

1)用户(User)模式：正常的程序执行状态。

2)FIQ模式：用于支持特殊的数据传送与通道处理 。

3)IRQ模式：用于通用的中断处理。

4)特权模式：一种用于操作系统的保护模式。

5)中止模式：当数据或指令预取中止时进入该模式。

6)系统模式：一种用于操作系统的特权用户模式。

7)未定义模式：当执行了未定义指令时进入该模式。

可用软件中断控制操作模式的切换，同时外部的中断和异常处理也会导致操作模式的切换；绝大多数的用户应用程序运行在用户模式；当系统响应中断或异常、或访问受保护的系统资源时，处理器会进入特权模式(除用户模式以外的所有模式)。

5.2.4 寄存器

ARM7TDMI 处理器内部有 37 个用户可见的寄存器。其中包括 31 个通用寄存器和 6 个状态寄存器。除了快速中断模式，每个处理器模式都共用寄存器 R0～R12，快速中断模式有自己的 R8～R12 寄存器；每一个异常模式都有自己的栈指针寄存器（R13）、连接寄存器（R14）和备份程序状态寄存器。

5.2.5 异常

只要正常的程序流被暂时中止，处理器就进入异常模式，例如响应一个来自外设的中断。在处理异常之前，ARM7TDMI内核保存当前的处理器状态，这样当处理程序结束以后可以恢复执行原来的程序。

5.2.5.1 进入异常

处理异常时，ARM7TDMI内核会进行下面的处理：

- 在适当的LR中保存一条指令的地址；（产生数据中止的装载或保存指令位PC+8）

- 将CPSR复制到适当的SPSR；
- 根据异常将CPSR模式强制设为某一值；
- 强制PC从相关的异常向量出取指令。

ARM7TDMI内核在发生中断时会置位中断禁止标志，这样可以防止不受控制的异常嵌套。

5.2.5.2 退出异常

在退出异常处理程序时：

- 将 LR 中的值减去偏移量后移入 PC；
- 将 SPSR 的值复制回 CPSR；
- 清零在入口置位的中断禁止标志。

表 5-1 中断向量表

中断向量地址	异常中断类型	异常中断模式	进入时 I/F 状态		优先级
0x0	复位	特权模式	禁止	禁止	1
0x4	未定义指令	未定义指令中止模式	I	F	6
0x8	软件中断（SWI）	特权模式	禁止	F	6
0x0C	指令预取中止	中止模式	I	F	5
0x10	数据访问中止	中止模式	I	F	2
0x14	保留	未使用	—	—	未使用
0x18	外部中断请求	外部中断模式	禁止	F	4
0x1C	快速中断请求	快速中断模式	禁止	禁止	3

5.3 本章小结

本章介绍了与 ARM7 处理器平台相关的技术。首先，本章中对 LPC2000 系列系统硬件结构进行介绍，其中主要对与移植有关的相关部分做了详细的介绍；然后介绍了 ARM7TDMI-S 核的指令集，流水线，操作模式，寄存器组和异常等。

第 6 章 移植 MKRTOS 到 ARM7

根据第 5 章所介绍的 LPC2200 实验平台的体系结构的特点，本章具体说明了将 MKRTOS 移植到 LPC2200 实验平台上所需要改变或是增加的与处理器相关的数据结构，函数以及相应的文件。

6.1 移植规则

6.1.1 移植前的准备工作

要移植一个操作系统到一个特定的 CPU 体系结构上并不是一件很容易的事情，要成功的移植一个操作系统，必须对以下各项有详细的了解[27]：

① 要了解目标体系结构；

② 要了解操作系统的原理；

③ 要了解使用的编译器；

④ 要了解准备移植的操作系统；

⑤ 要了解所使用的芯片；

在前面的章节中，我们已经分别对上述的一些内容进行了介绍，如：要移植的目标系统的体系结构和处理器核以及 MKRTOS 操作系统的内核。

6.1.2 处理器的选择

在熟悉操作系统内核的基础上，我们还需要对处理器的相关技术有所了解。我们选择的处理器需要满足下面的一些条件：

- ✧ 处理器支持中断，并且能够产生定时中断，中断对于嵌入式实时操作系统是非常重要的，如果处理器不能支持中断，则很难保证系统的实时性；
- ✧ 处理器支持能够容纳一定量数据（可能是几千字节）的硬件堆栈，在 MKRTOS 中要为每一个任务分配堆栈，在发生中断时，用户模式和系统模式也使用不同的堆栈；
- ✧ 处理器有将堆栈指针和其他 CPU 寄存器读出和存储器到堆栈或内存中的指令。

我们使用的 ARM7TDMI 处理器核满足上面的所有需求。

6.1.3 编译器的选择

目前，针对 ARM 处理器核的 C/C++语言编译器有很多，如 SDT、ADS、IAR、TASKING 和 GCC 等。据了解，目前在国内外最流行的是 SDT、ADS 和 GCC。前两者均是 ARM 公司自己开发的，ADS 为 SDT 的升级版，以后 ARM 公司都不会再支持 SDT，因此不选用 SDT。GCC 虽然使用广泛，很多开发套件使用它作为编译器，与 ADS 比较其编译效率低，这对充分发挥芯片的性能很不利，所以最终使用 ADS 编译程序和调试。

6.1.4 任务模式的取舍

ARM7 处理器核具有用户、系统、特权、中止、未定义、中断和快速中断 7 种模式，其中除了用户模式外，其他均为特权模式。关于 ARM7 处理器的详细情况在前面的章节中已经有所介绍，在 7 种模式中，特权模式、中止模式、未定义指令中止模式、外部中断模式和快速中断模式与相应的异常相联系，用户任务一般不能使用这些模式。而系统模式除了是特权模式以外，其他与用户模式是一样的（用户模式和系统模式使用完全相同的寄存器组）。因此可供用户使用的模式是用户模式和系统模式。为了尽量减少任务代码错误对整个程序的影响，缺省的任务模式定为用户模式，系统模式为可选模式，同时提供接口使任务可以在这两种模式间切换。

6.1.5 支持的指令集

带 T 变量的 ARM7 处理器核具有两种指令集：标准的 32 位 ARM 指令集和 16 位的 Thumb 指令集，这两种指令集有不同的应用范围。为了最大限度地支持芯片的特性，任务应当可以使用任意地一个指令集并可以自由切换，不同地任务应当可以使用不同的指令集。

6.2 开发工具

如前所述，移植我们的操作系统需要一个 C++编译器，并且是针对我们使用的 CPU，我们选择的开发工具是 CodeWarrior for ADS 集成开发环境来完成系统的开发、移植和测试的，并且使用 AxD 对程序进行调试[27]。

CodeWarrior for ADS 集成开发环境主要提供了下面一些功能：

- ✧ 按照工程项目的方式来组织源代码文件、库文件以及其他文件；
- ✧ 设置各种生成选项，以生成不同配置的映像文件；
- ✧ 一个源代码编辑器和一个源代码浏览器；
- ✧ 在文本文件中进行字符串搜索和替换；
- ✧ 文本文件比较功能等。

ARM 提供的可执行的映像文件的模版包括 3 个生成目标：

- ✧ Debug：使用本生成目标生成的映像文件中包含了所有的调试信息，用于在开发过程中使用；
- ✧ Release：使用本生成目标生成的映像文件中不包含调试信息，用于生成实际发行的软件版本；
- ✧ DebugRel：使用本生成目标生成的映像文件中包含了基本的调试信息。

6.3 INCLUDES.H 文件

includes.h 是一个头文件，它在所有.CPP 文件的第一行被包含。如下所示：

```
#include "includes.h"
```

includes.h 使得用户项目中的每个.CPP 文件不用分别去考虑它实际上需要哪些头文件。使用 includes.h 的唯一缺点是它可能会包含一些实际不相关的头文件。这意味着每个文件的编译时间可能会增加。但由于它增强了代码的可移植性，所以我们还是决定使用这一方法。

```
#include "Os_cpu.h"
#include "task.h"
#include "kernel.h"
#include "timer.h"
#include "taskManager.h"
#include "MKEROS.H"
#include "lpc2210.h"
//……
```

6.4 定义与处理器相关的常量，宏和类型

6.4.1 与编译器相关的数据类型

不同的处理器有不同的字长，要保证操作系统的移植成功，就需要定义一系列的数据结构，使得这些数据结构具有可移植性。对于 C/C++语言的 short、int、long 和 float 等等数据类型，他们是与处理器的类型有关的，隐含着不可移植性。试想如果要将一个系统移植到不支持上述数据类型的处理器上，那么可能需要修改每一个源文件中用到上述数据类型的语句，那个工作量是非常可怕的，因此我们根据 ADS 编译器的特性重新定义了在程序中使用到的数据结构，如下所示：

```
typedef unsigned char  uint8;      /* 无符号 8 位整型变量 */
typedef signed   char  int8;       /* 有符号 8 位整型变量 */
typedef unsigned short uint16;     /* 无符号 16 位整型变量 */
typedef signed   short int16;      /* 有符号 16 位整型变量 */
typedef unsigned int   uint32;     /* 无符号 32 位整型变量*/
typedef signed   int   int32;      /* 有符号 32 位整型变量  */
typedef float          fp32;       /* 单精度浮点数（32 位长度）*/
typedef double         fp64;       /* 双精度浮点数（64 位长度）*/
typedef unsigned char  BOOLEAN;    /* 布尔变量*/
......
typedef INT32U         OS_STK;     /* 堆栈是 32 位宽度*/
```

这样的数据结构即可以满足移植性，又非常的直观。这些数据结构在文件 includes.h 中定义。

6.4.2 定义软中断 SWI

软件中断 SWI 用于产生 SWI 异常中断，ARM 通过软中断指令实现在用户模式下对操作系统中特权模式的程序的调用。

用户任务可以使用两种处理器的模式：用户模式和系统模式，这两种模式对于系统的资源有不同的访问控制权限。为了使底层接口函数与处理器状态无关，同时在任务调用相应的函数时不需要知道函数的位置，因此在移植的时候使用软中断指令 SWI 作为底层的接口，使用不同的功能号区分不同的函数。用软中断作为操作系统的底层接口就需要在 C 语言中使用 SWI 指令。

在 ADS 中，用关键字_swi 来声明一个软中断，调用关键字_swi 声明的函数就相当于在调用这个函数的地方插入了一条 SWI 指令，并且可以为其指定功能号。同时，软中断的函数可以有参数和返回值。程序中的软中断定义如下所示：

```
__swi(0x00) void OS_TASK_SW(OS_STK *ptos0,OS_STK *ptos1);
   /*任务级任务切换函数 */
__swi(0x01) void OSStartHighRdy(OS_STK *ptos);
   /*运行优先级最高的任务*/
__swi(0x02) void OS_ENTER_CRITICAL(void); /*关中断*/
__swi(0x03) void OS_EXIT_CRITICAL(void);   /*开中断*/
__swi(0x80) void ChangeToSYSMode(void); /*任务切换到系统模式 */
__swi(0x81) void ChangeToUSRMode(void); /*任务切换到用户模式*/
```

软中断在 kernel.h 中声明。

6.4.3 设置堆栈增长方向

在 ARM 处理器中支持两种形式的堆栈增长方式，我们使用结构常量 OS_STK_GROWTH 来指定堆栈的生长方式：

- 当 OS_STK_GROWTH=0，则堆栈从下往上增长（递增堆栈）；
- 当 OS_STK_GROWTH=1，则堆栈从上往下增长（递减堆栈）；

ADS 的 C 语言编译器只支持堆栈从上往下增长，而且必须是满递减堆栈，所以我们将 OS_STK_GROWTH 设置为 1。代码如下所示：

```
#define OS_STK_GROWTH    1        /*堆栈是从上往下长的*/
```

该结构在 includes.h 中定义。

6.5 与处理器相关的汇编语言文件

6.5.1 软中断 SWI 的汇编接口

通常 SWI 异常中断的处理程序分为两级：第一级 SWI 异常处理程序为汇编程序，用于确定 SWI 指令中的 24 位的立即数；第二级 SWI 异常中断处理程序具体实现 SWI 的各个功能，可以是用汇编程序实现，也可以是 C 语言或 C++语言实现。在后面的章节中会分别对这两个级别的具体实现进行介绍。

本节中将会介绍用汇编语言实现的第一级软中断异常处理程序，即确定 SWI

指令的 24 位立即数。程序代码如下所示：[27][29]

```
SoftwareInterrupt
        LDR     SP, StackSvc            ; 重新设置堆栈指针
        STMFD   SP!, {R0-R3, LR}        ; LR 指向软中断的后一条指令
        MRS     R3, SPSR
        STMFD   SP!, {R3}               ; 将 SPSR 的值压入栈中
        MOV     R10,R0                  ; 新任务的堆栈指针
        MOV     R11,R1
    ; R1 指向参数存储位置，R1 必须保存，而且在其他的过程中不能改动
        MOV     R1, SP
        TST     R3, #T_bit              ; 中断前是否是 Thumb 状态
        LDRNEH  R0, [LR,#-2]            ; 是：将软中断指令读取到 R0 中
        BICNE   R0, R0, #0xff00         ; 取得 Thumb 状态 SWI 号
        LDREQ   R0, [LR,#-4]            ; 否：将软中断指令读取到 R0 中
        BICEQ   R0, R0, #0xFF000000     ; 取得 arm 状态 SWI 号
        ; r0 = SWI 号，R1 指向参数存储位置
        CMP     R0, #1
        LDRLO   PC, =OSIntCtxSw
        LDREQ   PC, =__OSStartHighRdy
        BL      SWI_Exception
        ;恢复 CPU 的寄存器
        MOV     SP,R1
        LDMFD   SP!, {R0}
        MSR     CPSR_cxsf,R0
        LDMFD   SP!, {R0-R3, PC}^
```

软中断的功能号包含在 SWI 指令中，程序通过读取该指令的相应位段获得。由于 ARM 处理器具有两个指令集，两个指令集的指令长度不一样，SWI 指令的功能号的位段也就不同，所以程序要先判断在进入软中断之前处理器是在什么指令集状态，并且根据不同的指令集使用不同的指令读取 SWI 指令并取得其中的功能号。然后程序用功能号与 1 作比较，当功能号为 0x00 时，就跳转到任务切换函数 OS_TASK_SW()处。当功能号等于 0x01 时，跳转到第一次任务切换处，执行函数_OSStartHighRdy()。因为这两个函数都需要明确的堆栈结构，而 C 语言不能满足这种需求，所以他们都只能用汇编语言来实现。

其他的功能则可以用出高级语言编写的处理函数处理，这些函数可以有两个

参数，第一个就是功能号，存在于 R0 中，第二个保存参数和返回值的指针，也就是堆栈中存储用户函数 R0～R3 的位置，实际上就是当前堆栈指针的值，它保存在 R1 中。

6.5.2 任务级的切换函数

函数OS_TASK_SW()是任务级的上下文切换函数，在任务被阻塞时主动请求CPU调度执行。因为发生切换的函数运行在用户模式下，不能调用系统模式下的代码，而上下文切换又需要通过内核来实现，所以任务级的函数切换一般是通过软件中断来实现的。

由于此时任务切换是在非异常模式下进行的，因此需要区别与中断级别的任务切换。函数OS_TASK_SW的处理流程可以分为：1）保存处理器寄存器（除了在软中断中已经保存过的寄存器）；2）将当前任务的堆栈指针保存到当前任务的任务控制块中；3）得到当前任务的任务控制块；4）得到就绪的最高优先级任务的任务控制块；5）从新任务的任务控制块中得到堆栈指针；6）将所有处理器寄存器从新任务的堆栈中恢复出来；7）执行中断返回指令。

6.5.3 中断级的切换函数

函数OSIntCtxSw()是中断级的任务切换函数，在时钟中断ISR(中断服务例程)中发现有高优先级任务等待的时钟信号到来，需要在中断退出后不返回被中断的任务，而是直接调度就绪的高优先级任务。尽管原理与任务级切换基本上相同，但是由于进入中断时已经保存过了被中断任务的CPU现场，因此不再进行类似的操作，只需要根据函数的嵌套情况做相应的调整。实现OSIntCtxSw的方法有两种：

1) 调整堆栈指针。根据所用的编译器对于函数嵌套的处理，通过精确计算得出所要调整的SP的位置，使得进入中断时所采用的保存现场的工作可以被重用。这种方法的好处是直接在函数嵌套内部发生任务切换，使得高优先级的任务能够最快地被调度执行，但是这个办法和具体的编译器以及编译参数的设置相关，需要较多的技巧。这也是我们使用的方法。

2) 设置需要切换的标志位。在ISR里面不发生切换而是先设置一个需要切换的标志，退出函数嵌套后，再根据标志位来判断是否需要进行中断级的任务切换。这种方法的好处是不需要考虑编译器的因素，也不用做计算，但是实际响应不是最快的。

由 6.5.1 节中的程序可知，调用 OSIntCtxSw 时的堆栈结构如图 6-1 所示：

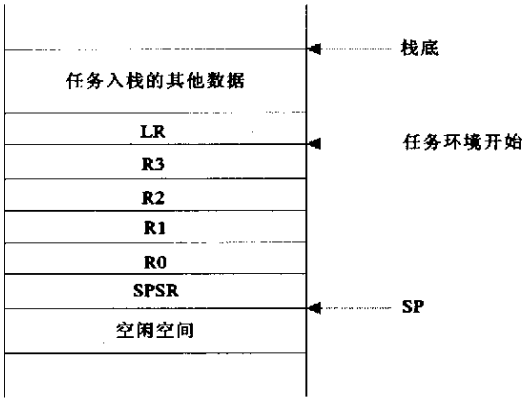

图 6-1 调用OSIntCtxSw时的堆栈结构

因为任务级的切换和中断级的切换大致原理相同，只是中断级的切换过程中不需要再次保存 CPU 寄存器， OSIntCtxSw()的代码如下所示：

```
OSIntCtxSw
        ;下面为保存任务环境
        ADD     SP,SP,#24
        LDR     R2, [SP, #-4]               ;获取 PC
        ;R3 保存了当前状态字
        MSR     CPSR_c, #(NoInt | SYS32Mode);切换到系统模式
        MOV     R12,SP
        MOV     SP, R11
        ADD     SP,SP,#68
        STMFD   SP!, {LR}
        STMFD   SP!, {LR}
        STMFD   SP!, {R0-R12}
        STMFD   SP!, {R3,R0}
        B       OSIntCtxS
__OSStartHighRdy
        ;管理模式下的堆栈指针回送到栈底，下次可以重复利用
        ADD     SP,R1,#24     ;manager mode point get to the bottom
        LDR     LR,[SP,#-4]   ;LR recover in the manager mode
OSIntCtxS
        MSR     CPSR_c, #(NoInt | SVC32Mode) ;进入管理模式
        MOV     SP,R10
        LDMFD   SP!,{R4,R5}                 ;获取新任务堆栈指针
```

```
        MSR     SPSR_cxsf,R5
        LDMFD   SP!,{R0-R12,LR,PC}^
```

需要注意的是使用系统模式返回任务。这是因为任务可能处于系统模式，也可能处于用户模式；可能使用 ARM 指令集，也可能使用 Thumb 指令集，只有系统模式的 SPSR 保存任务的 CPSR，然后使用指令 LDMFD SP!,{R0-R12,LR,PC}^返回时才能正确的切换 CPU 的模式和状态。

6.5.4 运行优先级最高的就绪任务函数

函数 OSStartHighRdy 用于在启动多任务前运行优先级最高的任务。由于此时系统中没有任务正在运行，所以不需要保存当前任务的寄存器到任务堆栈中。__OSStartHighRdy 后的代码为运行优先级最高的就绪任务的代码。

6.5.5 中断及时钟节拍

IRQ是受到操作系统管理的中断，由于各种ARM芯片的中断系统不一样各个用户的目标版也不一样，因此中断及时钟节拍是要进一步移植的代码。程序中定义了一个宏，这个宏可以适用于任何的基于ARM7的芯片，代码如下所示：

```
        MACRO
$IRQ_Label HANDLER $IRQ_Exception_Function


        EXPORT  $IRQ_Label                          ; 输出的标号
        IMPORT  $IRQ_Exception_Function             ; 引用的外部标号
$IRQ_Label
        SUB     LR, LR, #4                          ; 计算返回地址
        STMFD   SP!, {R0-R3, LR}                ; 保存任务环境
        MRS     R3, SPSR                            ; 保存状态
       ; 保存用户状态的R3,SP,LR,注意不能回写
       ; 如果回写的是用户的SP，则后面要调整SP
       STMFD   SP, {R3, SP, LR}^
        LDR     R2,  =OSIntNesting                  ; OSIntNesting++
        LDRB    R1, [R2]
        ADD     R1, R1, #1
        STRB    R1, [R2]
        SUB     SP, SP, #4*3
        MSR     CPSR_c, #(NoInt | SYS32Mode)        ; 切换到系统模式
```

```
        CMP     R1, #1
        LDREQ   SP, =StackUsr
        BL      $IRQ_Exception_Function          ; 调用c语言的中断处理
程序
        MSR     CPSR_c, #(NoInt | SYS32Mode)     ; 切换到系统模式
        LDR     R2, =OsEnterSum                  ; OsEnterSum,使
OSIntExit退出时中断关闭
        MOV     R1, #1
        STR     R1, [R2]
        BL      OSIntExit
        LDR     R2, =OsEnterSum                  ; 因为中断服务程序要
退出，所以OsEnterSum=0
        MOV     R1, #0
        STR     R1, [R2]
        MSR     CPSR_c, #(NoInt | IRQ32Mode)     ; 切换回irq模式
        LDMFD   SP, {R3, SP, LR}^                ; 恢复用户状态的
R3,SP,LR,注意不能回写
        ; 如果回写的是用户的SP，所以后面要调整SP
        LDR     R0, =OSTCBHighRdy
        LDR     R0, [R0]
        LDR     R1, =OSTCBCur
        LDR     R1, [R1]
        CMP     R0, R1
        ADD     SP, SP, #4*3
        MSR     SPSR_cxsf, R3
        LDMEQFD SP!, {R0-R3, PC}^                ; 不进行任务切换
        LDR     PC, =OSIntCtxSw                  ; 进行任务切换
    MEND
```

编写好该宏之后就可以编写中断服务程序的 C 语言部分了，如下所示：

```
 void Timer0_Exception(void)
 {
    T0IR = 0x01;
    VICVectAddr = 0;              // 通知中断控制器中断结束
```

```
    OSTimeTick();
}
```

编写好中断程序代码之后，还要把程序与相关的中断源挂接，使芯片在产生相应的中断之后会调用相应的处理程序，因此必须做两个工作：

- 增加汇编接口的支持：在文件 IRQ.S 中增加接口代码（如：Timer0_Handler　HANDLER Timer0_Exception）
- 初始化向量中断控制器：

```
VICVectAddr0 = (uint32)Timer0_Handler;
VICVectCntl0 = (0x20 | 0x04);
VICIntEnable = 1 << 4;
```

6.6 与处理器相关的 C/C++语言文件

6.6.1 初始化任务堆栈函数

在程序中使用 OSTaskStkInit()来初始化任务底堆栈，在编写该函数之前应该先确定任务的堆栈结构，因为任务的堆栈结构是与 CPU 的体系结构、编译器有密切的联系的。任务的堆栈结构如图 6-2 所示：[27][29]

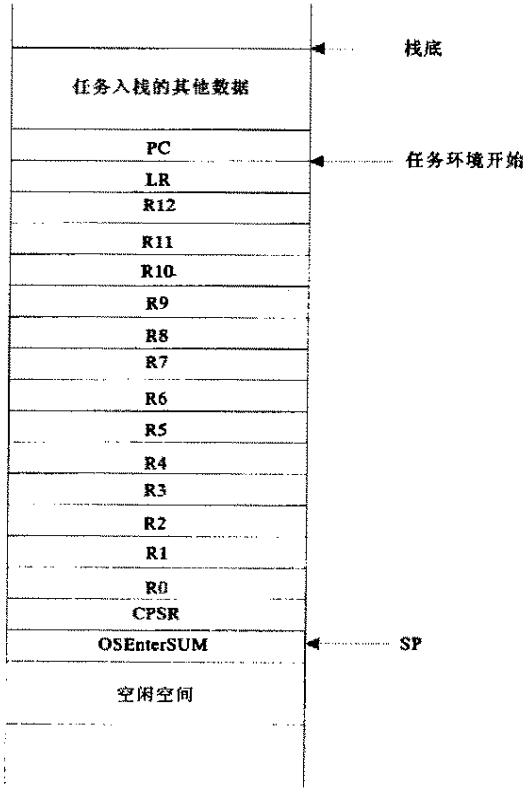

图 6-2 任务堆栈结构图

函数 OSTaskStkInit 在 task.cpp 中定义，因为在每一个任务初始化的时候都会为他们分配堆栈空间，所以我们将 OSTaskStkInit 作为 task 类的私有方法，定义在类 class 中。

```
class CTask
{
    ......
    private:
        OS_STK *OSTaskStkInit (void (*task)(), void *pdata, OS_STK
            *ptos, INT16U opt);
};
```

OSTaskStkInit 具体的代码已经在前面的章节进行了介绍。

我们在堆栈中定义了一个全局变量 OSEnterSUM，它不是 CPU 的寄存器，主要是用来保存关中断的次数，从而关中断和开中断可以嵌套使用。在调用 OS_ENTER_CRITICAL() 时，OSEnterSUM 的值增加，同时关中断。在调用 OS_EXIT_CRITICAL()时，OSEnterSUM 的值减一，并且只有在其为 0 的时候才允许中断。将 OSEnterSUM 保存在任务堆栈中是因为我们为每一个任务都配备了独立的 OSEnterSUM，在任务切换的时候这个任务保存和恢复各自的 OSEnterSUM，这样各个任务开关中断的状态可以不同，任务不必过分考虑关中断对别的任务的影响。

6.6.2 软中断 SWI 的 C++语言接口

我们已经知道软中断的中断处理程序可以分为 2 级，我们的程序中会使用 C++语言实现 SWI 异常处理程序的第 2 级中断处理程序。

软中断的 C++语言的处理函数 SWI_Exception(int SWI_Num, int *Regs)在 kernel.h 中定义，函数有两个参数：SWI_Num 为软中断的功能号，在定义软中断时分配，每一个软中断都有唯一的功能号（__swi(0x02) void OS_ENTER_CRITICAL(void)中 0x02 为该软中断的功能号），它可以从第一级的软中断处理程序中得到；Regs 为指向堆栈中的保存寄存器的值的位置，如果第一级的 SWI 异常中断处理程序将其栈指针作为第二个参数传递给 C 程序类型的第 2 级中断处理程序，就可以实现在两级中断处理程序之间传递参数，可以通过下面的形式交流数据：

value_in_reg_0=reg[0]和 reg[0]=updated_value_0

软件中断的 C 语言处理函数的代码如下所示：

```
extern "C" void SWI_Exception(int SWI_Num, int *Regs)
{
  switch(SWI_Num)
    {
        OS_TCB   *ptcb;
        INT8U sum;
        case 0x00: break;
        case 0x01: break;
        case 0x02:
          /* 关中断函数 OS_ENTER_CRITICAL()*/
           __asm
           {
               MRS      R0, SPSR
               ORR      R0, R0, #NoInt
               MSR      SPSR_c, R0
           }
           OsEnterSum++;
           break;
        case 0x03:
          /* 开中断函数 OS_EXIT_CRITICAL() */
           if (--OsEnterSum == 0)
           {
               __asm
               {
                   MRS      R0, SPSR
                   BIC      R0, R0, #NoInt
                   MSR      SPSR_c, R0
               }
           }
           break;
        case 0x80:                          /* 任务切换到系统模式 */
           __asm
           {
               MRS      R0, SPSR
```

```
            BIC     R0, R0, #0x1f
            ORR     R0, R0, #SYS32Mode
            MSR     SPSR_c, R0
        }
        break;
    case 0x81:                          /* 任务切换到用户模式 */
        __asm
        {
            MRS     R0, SPSR
            BIC     R0, R0, #0x1f
            ORR     R0, R0, #USR32Mode
            MSR     SPSR_c, R0
        }
        break;
    case 0x82:
/* 任务是 ARM 代码 Regs[0]表示任务优先级，Regs[1]表示任务 id*/
        if (Regs[0] < MAXPRIO)
        {
         /*获得任务的 TCB*/
            ptcb=
              OS.OSTCBPrioTblReadyCXH[Regs[0]]->pOSTCBFirst;
            sum=OS.OSTCBPrioTblReadyCXH[Regs[0]]->sum;
            while(sum>=0)
            {
            if(ptcb->OSTCBId!=Regs[1])
                ptcb=ptcb->OSTCBNext;
            sum--;
            }
            /*改变 CPSR 中的 T 位*/
            if (ptcb != NULL)
            {
                ptcb -> OSTCBStkPtr[1] &= ~(1 << 5);
            }
        }
```

```
            break;
        case 0x83:                              /* 任务是 THUMB 代码 */
            if (Regs[0] < MAXPRIO)
            {
             /*获得任务的 TCB*/
                ptcb=
                  OS.OSTCBPrioTblReadyCXH[Regs[0]]->pOSTCBFirst;
                sum=OS.OSTCBPrioTblReadyCXH[Regs[0]]->sum;
                while(sum>=0)
                {
                if(ptcb->OSTCBId!=Regs[1])
                    ptcb=ptcb->OSTCBNext;
                sum--;
                }
                /*改变 CPSR 中的 T 位*/
                if (ptcb != NULL)
                {
                    ptcb -> OSTCBStkPtr[1] |= (1 << 5);
                }
            }
            break;
        default:
            break;
    }
}
```

（注意：要在汇编程序中调用 C++程序时，要在 C++程序中使用关键字 extern “C”。）

6.2.2.1 软中断 0x00 和软中断 0x01

任务切换函数 OS_TASK_SW（0x00）和启动任务函数 OSStartHighRdy（0x01）在 Os_cpu_a.s 中实现：

6.2.2.2 软中断 0x02 和软中断 0x03

这两个软中断分别代表关中断和开中断。在所有的实时内核中访问代码的临界段时都需要关中断，并且在访问完毕后重新允许中断，以防止临界段代码被多任务或中断服务例程破坏。因为中断禁止时间将影响到用户的系统对实时事件的

响应能力，所以一般都用汇编语言实现。通常每个处理器都会提供一定的指令来禁止/允许中断，有些编译器能够允许用户在 C++源代码中插入汇编语言声明，ADS 就允许在 C++语言中嵌入汇编语言。为了隐藏编译器厂商提供的具体实现方法，我们定义了两个宏来禁止和允许中断：OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()。有两种方法来实现这两个宏，我们使用的是第一种方法：

- 执行这两个宏的第一个也是最简单的方法是在 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 中直接调用处理器指令来关闭和允许中断。但是，这个方法存在不足之处。如果用户在禁止中断的情况下调用系统函数，那么在返回的时候，中断可能会变成是允许了！严格意义上的关闭中断应该是执行 OS_ENTER_CRITICAL() 后中断始终是关闭的，因此方法 1 不满足要求。但是它的最大优点就是简单，执行速度快（只有一条指令）。如果在任务中并不在意调用函数返回后是否被中断，可以采用该方法。
- 执行 OS_ENTER_CRITICAL() 的第二个方法是先将中断禁止状态保存到堆栈中，然后禁止中断。而执行 OS_EXIT_CRITICAL() 的时候只是从堆栈中恢复中断状态。如果用户在中断禁止的时候调用系统服务，其实用户是在延长应用程序的中断响应时间。用户的应用程序还可以用 OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL() 来保护代码的临界段，中断状态不会被改变。但是使用这种方法有可能会导致应用程序的崩溃，发生这种情况的原因是当用户调用任务挂起和延时等服务时，任务被挂起直到时间期满，而中断是禁止的，使得用户不能获得节拍中断。一个通用的办法是用户应该在中断允许的情况下调用上述的系统服务。

开关中断的代码是与处理器有关的代码，是需要移植的代码。在 ARM 处理器核关中断和开中断时是通过改变程序状态寄存器 CPSR 中的相应的控制位 CPSR[7] 来实现的，由于使用了软中断，程序状态寄存器 CPSR 保存到程序状态保存寄存器 SPSR 中，软件中断退出时会将 SPSR 恢复到 CPSR 中，因此只需要改变 SPSR 中的相应的控制位就可以了。

6.6.2.3 软中断 0x80 和软中断 0x81

根据 ARM 核心的特点核移植的目标，增加了两个处理器模式转换函数，ChangeToSYSMode() 和 ChangeToUSRMode()，分别是软中断 0x80 和 0x81。他们都是通过软件中断指令 SWI 转换到系统模式，通过软件中断服务程序实现。

函数 ChangeToSYSMode 将当前任务转换到系统模式，函数 ChangeToUSRMode 将当前任务转换到用户模式，他们可以在任务情况下使用。这两个函数改变 SPSR 中相应位置的值，而 SPSR 会在退出软中断处理函数时被复制到 CPSR，任务的处理器模式就改变了。

6.6.2.4 软中断 0x82 和软中断 0x83

ARM 拥有两种指令集：ARM 指令集和 THUMB 指令集，但是任务建立的时候默认的是一种指令集，如果任务的第一条指令不属于默认的指令集，那么程序将会出错，因此，我们增加了两个函数 TaskIsARM()和 TaskIsTHUMB()用于改变任务建立时的默认指令集，这两个函数分别用软中断 0x82 和 0x83 实现。

TaskIsARM 用于声明指定的任务的第一个指令是 ARM 指令集中的指令，TaskIsTHUMB 用于声明指定的任务的第一个指令是 THUMB 指令集中的指令，他们需要两个参数，分别是要改变的任务的优先级和任务的 ID（因为在系统中同一个优先级可以有多个任务），而且这两个函数应该在相应的任务建立后但还没有运行的时候调用。这样的话，如果在低优先级的任务中创建高优先级的任务时可能会出现问题，因为可能在低优先级的任务调用这两个函数之前就发生了任务切换。此时可以有 3 种解决方案：使高优先级的任务使用默认的指令集；在建立任务的时候使任务不要进入就绪态，建立任务后调用相应的服务使任务进入就绪态；建立任务时禁止任务切换，调用 TaskIsARM()和 TaskIsTHUMB()后再允许进行任务切换。

6.7 内核的运行方式

为了提高执行速度和系统可靠性，嵌入式系统中的软件一般都固化在存储器芯片或FLASH中。嵌入式操作系统的内核有两种可选的运行方式：可以在Flash上直接运行，也可以加载到内存中运行。

Flash运行方式：把内核的可执行映象烧写到Flash上，系统启动时从Flash的某个地址开始逐句执行。这种方法实际上是很多嵌入式系统采用的方法。这种做法可以减少内存需要。内核加载方式：把内核的压缩文件存放在Flash上，系统启动时读取压缩文件在内存里解压，然后开始执行，这种方式相对复杂一些，但是运行速度可能更快(RAM的存取速率要比Flash高)。这是标准Linux系统采用的启动方式。使用这种方式，我们需要一个BootLoader。

我们使用的是Flash运行方式。因为在LPC2200的开发版上没有片内的Flash，所以我们将内核固化在片外的Flash上。[30] 首先，我们需要设置编译连接的地址（使代码地址从0x80000000开始）；然后需要编写分散加载描述文件，以便于将内核的各个部分加载到Flash上不同的位置，代码如下所示：

```
ROM_LOAD 0x80000000{
    ROM_EXEC 0x80000000{
        Startup.o (vectors, +First)
```

```
        * (+RO)
    }
    IRAM 0x40000000{
        Startup.o (+RW,+ZI)
    }
.....
}
```

最后使用 JTAG 接口下载程序到片外 Flash 即可，使用 JTAG 仿真器时需要设置 Flash 的起始地址为 0x80000000。

6.8 本章小结

根据前章介绍的 ARM7 处理器平台的特点，本章在 LPC2200 实验开发平台上对 MKRTOS 进行了移植。本章首先对 MKRTOS 的开发工具——ADS 进行了简单的介绍；随后分析了移植 MKRTOS 到 ARM7 处理器平台上需要改动和编写的代码，包括汇编语言的代码和 C++语言的代码；最后说明了 MKRTOS 内核的运行方式。

第 7 章 总结和展望

7.1 研究工作的总结

本文讨论了一种基于面向对象的嵌入式实时操作系统的设计实现与移植。操作系统是复杂的系统软件，涉及到众多的软、硬件方面的内容。我们对现有的多个开源的嵌入式操作系统内核以及 ARM7 处理器平台进行研究和分析，并且对宏内核，微内核和超微内核的体系结构进行了比较，提出了基于超微内核结构的具有面向对象特性的嵌入式实时操作系统的一些设计原则，并且使用面向对象的C++程序开发语言对嵌入式实时操作系统MKRTOS进行了实现和基于ARM7的移植。本文主要的创新工作表现在以下的方面：

1） 对嵌入式操作系统内核和超微内核结构做出深入的理论分析，设计了一款具有超微内核结构的嵌入式实时系统；

2） 使用用面向对象的 C++语言实现了一款嵌入式实时操作系统 MKRTOS；

3） 由于使用了面向对象的程序设计语言进行系统的开发，MKRTOS 具有其他一些嵌入式实时操作系统所没有的面向对象的特性，诸如封装性和重用性等；

4） 根据 MKRTOS 的优先级和任务调度等方面的特点，在系统中采用了 RMS 和 Round-Robin 调度算法相结合的调度算法；

5） 在 ARM7 处理器平台上对 MKRTOS 进行了移植，初步实现了一个能够在 ARM7 处理器平台上正常运行的操作系统。

7.2 进一步研究工作的展望

虽然此次设计基本达到了预先设定的目标，实现了嵌入式实时操作系统的最基本的功能，系统也可以在目标平台上正确的运行。但是由于项目的开发周期比较短，MKRTOS 的很多功能都没有能够实现，诸如：文件系统，设备管理，图形界面，网络等方面都需要进行进一步的完善，而且系统的总体性能也需要进一步的优化与提高。

通过这次的开发过程，使我们对嵌入式实时操作系统的结构， ARM7 处理器的结构，以及对操作系统移植相关知识和技术都有了更深一层的了解和认识。但是由于我们的经验和能力有限，所做的研究和开发工作都存在许多需要改进和提高的地方，因此希望各位老师和专家能够多提宝贵的建议。同时也希望我们所做的工作能够对大家今后的研究有所帮助。

附录　攻读硕士学位期间参加的科研项目和发表的论文

发表论文

基于 generalized dominator 的 BDD 分解方法的分析，电脑知识与技术，2006.3，已发表

程序切片计数浅析，电脑知识与技术，2006.3，已发表

参考文献

【1】Wayne Wolf 著，孙玉芳等 译：嵌入式计算系统设计原理，机械工业出版社，2002

【2】龚炳铮：发展嵌入式计算机及其产业的思考，电子技术应用，2000(10)：4～6

【3】金西，黄汪：嵌入式 Linux 技术及其应用，计算机应用，2002(07)：4～6

【4】吕京建，肖海桥：面向21世纪的嵌入式系统，Semiconductor Technology 2001(26)

【5】徐伦峰，熊光泽：实时操作系统及其发展过程，计算机应用，2001(06)

【6】Eugene Olafsen：MFC Visual C++6 编程技术内幕(英文版)，北京：机械工业出版社，2001

【7】Ramchandra Garge：Xu Leasun 译，面向对象程序设计，2003.03.27

【8】http://www.yesky.com/，C++箴言：将数据成员声明 private 2005.08.02

【9】http://www.ynni.edu.cn/mzxy/jiaoxueyuan/jiaoxu/bcjq_cb.htm， C++ Builder 的编程模式

【10】Stanley B. Lippman：Inside the C++ Object Model，Addison Wesley，1996

【11】 Grady Booth：面向对象项目的解决方案，北京：机械工业出版社，2003

【12】Scott W Ambler， The Application Developer's Guide to Object Orientation and the UML(Second Edition)，Cambridge University Press，2003

【13】Frank Buschmann，Regine Meunier，Hans Rohnert，Peter Sommerlad and Michael Stal：A System of Pattern—Pattern-Oriented Software Architecture，1996，37～40

【14】Eric Verhulst:The Rationale for Distributed Semantics as a Topology Independent Embedded Systems Design Methodology and its Implementation in the Virtuoso RTOS . Boston，Design Automation for Embedded Systems. 2002(06),277～294

【15】JEAN-PHILIPPE BABAU: Object Oriented Design for Real-Time Systems-Response to C. E. Pereira's Contribution. Boston，The International Journal of Time-Critical Computing Systems. 2000(18),95～99

【16】Alan C. Bomberger，A. Peri Frantz，William S. Frantz，et al. The

KeyKOS Nanokernel Architecture. In: Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, USENIX Association, April 1992. 95～112

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

【18】张益农，黄文玲：嵌入式操作系统中硬件抽象层的描述，内蒙古工业大学学报，2002.21(4)，293～297

【19】Bjarne Stroustrup，裘宗燕 译：C++程序设计语言，北京：机械工业出版社，2002

【20】王志平，熊光泽：实时调度算法研究，电子科技大学学报，2000.29(2)，205～208

【21】杨科峰，邵时：嵌入式实时系统调度策略，计算机应用研究，2001(8)，31～33

【22】John Lions 著，尤晋元译：莱昂氏 UNIX 源代码分析，北京：机械工业出版社 2000

【23】Jane W.S.Liu：REAL-TIME SYSTEMS，北京：高等教育出版社，2002

【24】胡继阳等著：嵌入式系统导论，北京：中国铁道出版社，2005

【25】毛德操，胡希明：Linux 内核源代码情景分析，杭州：浙江大学出版社，2001

【26】杜春雷编著：ARM 体系结构与编程，北京：清华大学出版社，2003

【27】周立功：ARM 嵌入式系统基础教程，北京：北京航空航天大学出版社，2005

【28】周立功：ARM 嵌入式系统实验教程，北京：北京航空航天大学出版社，2004

【29】王田苗：嵌入式系统设计与实例开发——基于 ARM 微处理器与 μC/OS-II 实时操作系统，北京：清华大学出版社，2002

【30】毛德操，胡希明：嵌入式系统采用公开源代码和 StrongARM/Xscale，浙江大学出版社，2003

【31】王京起，黄健，沈中杰编著：嵌入式可配置实时操作系统 eCos 技术及实现机制，北京：电子工业出版社，2005

致　谢

经过两年的努力学习，在老师、同学、家人和朋友的帮助和支持下，我的硕士学位论文如期完成了。这两年的学习生活使我受益匪浅。

首先，我要感谢我的导师曾振柄教授和琚小明老师，他们治学态度严谨，为人谦和，有着非常渊博的专业知识。在论文的选题过程、书写过程和审稿过程中都给予我很多的建议和帮助，使我能够顺利地完成论文的写作。

我要感谢同一课题小组的成员陈晓华、田令平和吴景峰同学的大力帮助和支持。在写论文的这半年时间里，我们一起生活，一起学习，一起探讨问题。他们给了我许多启发性的见解，提供了大量有益的资料，开拓了我的视野，丰富了我的知识。虽然只是短短的半年时间，但是我们之间已经建立了非常深厚的友谊。

我要感谢刘韬，邹晓和其他帮助过我的同学，在我心情不佳的时候给我带来快乐，帮助我一起克服困难。

我还要感谢我的家人，他们始终鼓励和支持着我，是我永远坚强的后盾。

最后，祝尊敬的曾振柄教授、琚小明老师和所有的评审老师身体健康、万事如意，祝所有的同学事业有成，前途无量。