分类号_____ 密级 _____
UDC _____

学 位 论 文

嵌入式实时操作系统内核 AcoolOS 的设计与实现

作者姓名： 郝建义

指导教师： 李晶皎 教授

东北大学信息科学与工程学院

申请学位级别： 硕士 学科类别： 工学

学科专业名称： 计算机系统结构

论文提交日期： 2007 年 12 月 论文答辩日期：2008 年 1 月

学位授予日期： 答辩委员会主席：高福祥

评阅人： 刘浪涛 王剑

东 北 大 学

2007 年 12 月

A Thesis for the Degree of Master in Computer Architecture

Design and Implementation of an Embedded Real-time OS Core AcoolOS

by Hao Jianyi

Supervisor: Professor Li Jingjiao

Northeastern University
December 2007

独创性声明

本人声明所呈交的学位论文是在导师的指导下完成的。论文中取得的研究成果除加以标注和致谢的地方外，不包含其他人已经发表或撰写过的研究成果，也不包括本人为获得其他学位而使用过的材料。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诚挚的谢意。

学位论文作者签名：郝建义

签字日期：2008.1.5

学位论文版权使用授权书

本学位论文作者和指导教师完全了解东北大学有关保留、使用学位论文的规定：即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，允许论文被查阅和借阅。本人同意东北大学可以将学位论文的全部或部分内容编入有关数据库进行检索、交流。

（如作者和导师同意网上交流，请在下方签名：否则视为不同意）

学位论文作者签名：郝建义　　导师签名：李晓欣

签字日期：2008.1.5　　签字日期：2008.1.6

嵌入式实时操作系统内核 AcoolOS 的设计与实现

摘 要

在嵌入式领域，随着微处理器性能的提高，系统复杂度越来越高，实时操作系统RTOS(Real Time Operating System)正得到越来越广泛的应用。但是，目前市场上应用广泛的多是一些付费的 RTOS，即使可以在网络上下载一些免费的操作系统，但其又存在系统服务功能过于简单，应用层接口函数较少，稳定性得不到保证等方面的不足。为此，本课题研究并实现了一个实时操作系统内核－AcoolOS(a cool operating system)。

本课题实现的实时操作系统内核－AcoolOS，它支持多任务，采用基于优先级的可抢占式调度，对相同优先级的任务可采用时间片轮转调度或按任务就绪的先后顺序来调度；对于中断的处理分为两种形式：LISR(Low-Level Interrupt Service Routine)与HISR(High-Level Interrupt Service Routine)，分别用于处理那些需要及时做出响应的事务和可以稍后处理的事务；对于任务间的通信提供了消息队列的机制；对于任务间的同步提供了互斥信号量机制；对于资源的共享提供了计数信号量机制。另外，它还提供了内存管理功能和时间管理功能，为用户提供了齐全的应用接口。

在结构上，AcoolOS 采用模块化的结构，用户可根据需要方便地进行裁剪；在功能上，AcoolOS 为用户提供了任务管理、中断管理、信号量、时间管理、内存管理、消息队列等常用的接口函数。在实时性上，由于采用优先级位图算法，可保证任务调度的时间确定性。

AcoolOS 在基于 ARM 的处理器上实现，并基于此操作系统进行了实际项目开发。实践证明，此操作系统运行良好，有很好的可靠性、实时性，可满足大部分实时系统开发的需要。

关键词：嵌入式；实时；操作系统

Design and Implementation of an Embedded Real-time OS Core AcoolOS

Abstract

In the embedded field, with the enhancement of microprocessor performance, systemes are becoming more and more complex and the RTOS (Real Time Operating System) is used more and more widely. But on the market, most RTOS used are not free, even some free RTOS can be get in the net, but there are too many defects. For example, they provide too simple system services, too little application interface functions, and the reliability can't get guarantee. So the subject is to develop a realtime operating system core, namely AcoolOS(a cool operating system).

The RTOS core AcoolOS, it is a real-time, preemptive, multitasking kernel designed for time-critical embedded applications. For the tasks which have the same priority, it schedules across time slice or the order that they become ready; for the interrupt, it provides two service routines: LISR (Low-Level Interrupt Service Routine) and HISR (High-Level Interrupt Service Routine), LISR is used for quick response to interrupt and HISR is for affairs which can be deal with later; for the communication of tasks, it provides information queue mechanism, for the synchronization of tasks, it provides a mutex semaphore, for the resource sharing, it provides counting semaphores, besides these, it provides memory and time management mechanism. These application interfaces are enough for users.

AcoolOS adopts a kind of modular construction, it is convenient for users to tailor. In function, AcoolOS provides interface functions which are in common use such as task management, interrupt management, semaphore, time management, memory management and information queue and so on. The Bit-Priority arithmetic which is used in the AcoolOS makes sure that the time for the task's schedule is a constant.

AcoolOS runs on ARM microprocessor well, and it has been applied in the practical project, and the effect is very well, and it has high reliability and hard real time.

Keywords: embedded; real-time; OS

目录

第一章 引言

嵌入式实时操作系统是能在确定时间内执行其功能，并对外部的异步事件做出响应的计算机系统。它将一个实时应用作为一系列独立的任务来运行，每个任务有各自的线程和系统资源。另外，它对于硬件中断的处理必须限制在一定的时间内[1]。

实时操作系统是事件驱动(event-driven)的，能对来自外界的作用和信号在限定的时间范围内做出响应。它强调的是实时性、可靠性和灵活性，与实时应用软件相结合成为有机的整体，起着核心作用；由它来管理和协调各项工作，为应用软件提供良好的运行软件环境及开发环境。在多任务实时系统中，必然由实时操作系统来对实时任务进行管理。

随着微处理器性能的提高，嵌入式系统的复杂度越来越高，嵌入式软件的规模也发生指数型增长，为提高嵌入式产品的研发效率，缩短研发时间，降低嵌入式软件的复杂度，促进软件开发的分工协作，嵌入式系统采用操作系统已成为一种趋势或必然，而嵌入式系统大多数是实时系统，这也就要求采用的操作系统支持实时性，即嵌入式实时操作系统。

1.1 课题背景

实时操作系统(RTOS)的研究是从六十年代开始的，而在 1981 年 Ready System 推出了世界上第一个商业嵌入式实时内核(VRTX32)，此后嵌入式实时系统得到了飞速发展，从支持八位微处理器(CPU)到十六位，再到三十二位甚至六十四位微处理器；从支持单一品种的微处理器芯片到支持多品种的微处理器芯片；从只有实时内核到除了内核外还提供其他功能模块，如文件系统，TCP/IP 网络系统等等[2]。

RTOS 在嵌入式系统设计中发挥着越来越重要的作用，然而对大多数的嵌入式开发人员来说，RTOS 到底是如何工作的并不清楚，这也妨碍了他们开发出更加高效的嵌入式应用程序。下面介绍一个 RTOS 即 AcoolOS 的具体开发与实现的过程，希望对广大嵌入式开发人员有所帮助。

1.1.1 RTOS 技术现状

进入二十世纪九十年代后，RTOS 在嵌入式系统设计中的主导地位已经确定，越来越多的工程师使用 RTOS，RTOS 的技术发展有以下一些变化：

(1) 因为新的处理器越来越多，RTOS 自身结构的设计更易于移植，以便在短时间内支持更多种微处理器。

(2) 开放源码之风已波及 RTOS 厂家。数量相当多的 RTOS 厂家出售 RTOS 时，就附加了源程序代码并含生产版税。

(3) 电信设备、控制系统要求的高可靠性，对 RTOS 提出了新的要求。瑞典 Enea 公

司的 OSE 和 WindRiver 新推出的 Vxwork AE 对支持 HA(高可用性)和热切换等特点都下了一番功夫。

(4) 由于 Linux 的开源免费的特性，但由于其是分时的操作系统，不具有硬实时性，因此一些用户对其进行改造，具有了实时的特性。在这方面比较成功的有 RTLinux、RTAI 等[3,4,5]。

(5) 近几年国外发展了一种基于微内核思想设计的精巧的嵌入式微内核，即实时超微内核，超微内核是一种非常紧凑的基本内核代码层，为嵌入式应用提供了可抢占、快而确定的实时服务。在它的基础可以灵活地构造各种类型的、与现成系统兼容的和可伸缩的嵌入式实时操作系统，因此能满足应用代码的可重用性和可伸缩性的要求[2]。

1.1.2 RTOS 技术发展趋势

(1) RTOS 的标准化研究

目前国内外的 RTOS 种类繁多，各具特色，但这给应用开发者带来了困难，由于接口的不统一，当选择不同的 RTOS 时，首先是代码的重用性的问题，另外，开发者还要重新学习新的 RTOS 的应用。

(2) 多处理器结构 RTOS、分布式实时操作系统和实时网络的研究[6]

单处理器的计算机系统已不能很好地满足某些实时应用的需要，开发支持多处理器结构的 RTOS 已成为发展方向，分布式实时操作系统的研究还未完全成熟，特别是在网络实时性和多处理器之间的任务调度算法上还需要进一步的研究[7,8]。

(3) 集成的开放式实时系统开发环境的研究

开发实时应用系统，只有 RTOS 是不够的，需要集编辑、编译、调试、模拟仿真等功能为一体的集成开发环境的支持，开发环境的研究还包括网络上多主机间协作开发与调试应用技术的研究、RTOS 与环境的无缝连接技术等[9,10]。

1.2 课题研究的内容与意义

目前国内外的嵌入式 RTOS 开发厂商有数十家，提供了上百个 RTOS，它们各具特色，提供的功能也起来越丰富，它们有的以二进制代码的形式向用户提供，而有的则提供源代码，但这些商用操作系统无一例外的以版费等形式向用户索取高额费用。而对一些小型软件企业仅仅需要一些功能简单的、可靠的、费用较低的甚至免费的小型实时操作系统，虽然网络可以下载一些免费的非商用的操作系统，但由于其可靠性、维护性等方面的原因令一些厂家不敢恭维，因此自己开发一种嵌入式实时操作系统成为一种选择。本课题主要研究嵌入式操作系统内核的设计与实现，内容包括任务的实时调度、任务间通信、定时器管理、中断处理、内存管理等。将其应用于嵌入式系统的开发中，其意义主要体现在以下几个方面：

(1) 操作系统的应用将缩短嵌入式软件开发周期，提前产品的上市时间，从而提高产品的竞争力；

(2) 针对特定的系统，可以对 RTOS 进行适当的裁减与配置，提高软件运行的效率；

(3) 避免了购买商业 RTOS 的版费，降低了产品的研发费用。

1.3 论文组织结构

本文的章节安排如下：

第一章：分析了当前嵌入式实时操作系统的现状，介绍了本课题研究的内容和意义；

第二章：介绍了开发操作系统 AcoolOS 的硬件平台与软件开发工具；

第三章：详细介绍了操作系统 AcoolOS 各个模块的设计与实现；

第四章：详细介绍了操作系统 AcoolOS 提供给用户的各个接口函数；

第五章：对操作系统 AcoolOS 进行了应用测试，并进行了性能分析；

第六章：总结了开发的操作系统 AcoolOS 的特点及不足。

第二章 AcoolOS 开发平台的构建

操作系统的开发平台包含硬件平台和软件开发工具，在实现操作系统以前，首先需要确定硬件开发平台。

2.1 S3C2440A硬件平台

嵌入式系统的硬件是以嵌入式微处理器为核心，主要由嵌入式微处理器、总线、存储器以及输入/输出接口和设备组成。其中又以微处理器和操作系统的实现紧密相关，这是因为不同的微处理器有不同的体系结构和指令集系统。

目前主流的嵌入式微处理器系列主要有 ARM 系列、MIPS 系列、PowerPC 系列、Super H 系列和 X86 系列，其中以 ARM 系列最为流行，目前已经稳居世界 32/64 位处理器首位，是高端嵌入式系统开发的工业标准和首选[11]。

ARM 嵌入式处理器是一种高性能的 32 位 RISC 芯片，具有功耗低、性价比高和代码密度高等三大特色。它由英国 ARM 公司设计，世界上几乎所有的主要半导体厂商都生产基于 ARM 体系结构的通用芯片，或在其专用芯片中嵌入 ARM 的相关技术。目前，70%的移动电话、大量的游戏机、手持 PC 和机顶盒都已采用了 ARM 处理器，许多的一流的芯片厂商都是 ARM 的授权用户(Licensee)，如 Intel、Samsung、TI、Freescale、ST 等公司，ARM 已成为业界公认的嵌入式微处理器标准。

作为一种 RISC 体系结构的微处理器，ARM 处理器具有 RISC 体系结构的典型特征[12]，同时具有以下特点：

(1) 在每条数据处理指令当中，都控制算术逻辑单元 ALU 和移位器，以使 ALU 和移位器获得最大的利用率；

(2) 自动递增和自动递减的寻址模式，以优化程序中的循环；

(3) 同时执行 Load 和 Store 多条指令，以增加数据吞吐量；

(4) 所有指令都可以条件执行，以增大执行吞吐量。

这些是对基本 RISC 体系结构的增强，使得 ARM 处理器可以在高性能、小代码尺寸、低功耗和小芯片面积之间获得好的平衡。

目前 ARM 的处理器主要有五大系列：ARM7、ARM9、ARM9E、ARM10 和 SecurCore 等，此外还有 StrongARM 和 XScale 等，本课题以目前国内应用较广的三星公司的 ARM9 芯片——S3C2440A 为基础构造实验平台。

S3C2440A 以 ARM920T 为它的 CPU 核心，ARM920T 采用哈佛体系结构，支持五级流水线，可达到 1.1MIPS/MHz 的处理性能，ARM920T 集成了包括 JTAG-ICE、专门 UART 调试通道(DBGU)及实时追踪的一系列的调试功能。这些功能使得开发、调试所有的应用特别是受实时性限制的应用成为可能，当然包括操作系统的开发调试。

2.2 RealView软件开发工具

ARM 软件的开发工具根据功能的不同，分别有编译软件、汇编软件、链接软件、调试软件、嵌入式实时操作系统、函数库、评估板、JTAG 仿真器、在线仿真器等，目前世界上约有四十多家公司提供以上不同类别的产品。用户选用 ARM 处理器开发嵌入式系统时，选择合适的开发工具可以加快开发进度，节省开发成本。因此一套含有编辑软件、编译软件、汇编软件、链接软件、调试软件、工程管理及函数库的集成开发环境(IDE)一般来说是必不可少的。

目前市场上比较流行的 ARM 开发工具有 ARM ADS，IAR EWARM 和 ARM RealView 等，它们均有不同的特点，在市场上占有一定的份额。

ARM ADS 的英文全称为 ARM Developer Suite，是 ARM 公司推出的一代 ARM 集成开发工具，用来取代 ARM 公司以前推出的开发工具 ARM SDT，目前 ARM ADS 的最新版本为 1.2。ADS 包括了四个模块分别是：SIMULATOR；C 编译器；实时调试器；应用函数库。ADS 的编译器调试器较 SDT 都有了非常大的改观， ADS1.2 提供完整的 WINDOWS 界面开发环境。C 编译器效率极高，支持 C 以及 C++，使工程师可以很方便的使用 C 语言进行开发。提供软件模拟仿真功能，使没有 Emulators 的学习者也能够熟悉 ARM 的指令系统。配合 FFT-ICE 使用，ADS1.2 提供强大的实时调试跟踪功能，片内运行情况尽在掌握。ADS1.2 需要硬件支持才能发挥强大功能。目前支持的硬件调试器有 Multi-ICE 以及兼容 Multi-ICE 的调试工具如 FFT-ICE。而简易下载电缆不能支持 ADS1.2。但目前 ARM 公司已停止了对 ADS 的更新，转而支持另外一种新的开发工具 RealView，这也是我们要采用的开发环境，后面将详细介绍。

IAR EWARM 的英文全称为 Embedded Workbench for ARM，是 IAR Systems 公司为 ARM 微处理器开发的一个集成开发环境。比较其他的 ARM 开发环境，IAR EWARM 具有入门容易、使用方便和代码紧凑等特点。IAR Systems 公司目前推出的最新版本是 IAR Embedded Workbench for ARM version 4.30。IAR EWARM 的主要特点如下：

(1) 高度优化的 IAR ARM C/C++ Compiler；
(2) IAR ARM Assembler；
(3) 一个通用的 IAR XLINK Linker；
(4) IAR XAR 和 XLIB 建库程序和 IAR DLIB C/C++运行库；
(5) 功能强大的编辑器；
(6) 项目管理器；
(7) 命令行实用程序；
(8) IAR C-SPY 调试器(先进的高级语言调试器)。

但它并不是由 ARM 公司支持的开发工具，在编译效率、代码密度方面不如 RealView 做的更好一些。

下面介绍本课题所应用的开发工具 RealView，它是由 ARM 公司力推的新一代 ARM

开发工具套件，主要由以下部分组成：

(1) 编译器、调试器：RVDS(RealView Developer Suite)；

(2) JTAG 仿真器：RVI(RealView ICE)Multi_ICE；

(3) 硬件跟踪器：RVT(RealView Trace)Multi_Trace。

与其它 ARM 开发工具相比，得到 ARM 公司支持的 RealView，有以下特性：

(1) 进一步改进基于 ARM 处理器的应用代码密度。该开发套件采用了一个新的可选 microlib C 库(ISO 标准 C 库的一个子集)，并将其面积缩至最小以满足微控制器应用的需求。Microlib C 库可将运行时的库代码尺寸缩小 92%；

(2) 支持所有的 ARM 系列核，并与众多第三方实时操作系统及工具商合作简化开发流程；

(3) RealView 开发工具可通过大幅缩小代码镜像，增加新的源代码和调试分析工具，帮助开发者更好地利用他们的硬件资源，并更有效地对应用进行验证；

(4) 可以实现代码的最大优化，实现代码密度的自动提升等功能，因此不需要软件开发人员花费过多的时间去优化语言代码。

第三章 AcoolOS 内核的设计与实现

嵌入式实时操作系统与其它操作系统相比，最大的特点体现在实时性方面，实时内核的设计均以任务的快速响应为原则，使最高优先级的任务在最短的确定的时间内得到响应[8,13]。下面以各个功能模块的实现为线索来介绍内核的设计。

3.1 AcoolOS体系结构

本课题研究的目的是实现一个嵌入式实时操作系统内核，实现任务的实时调度，便于程序的开发。操作系统开发将采用比较流行的模块化结构[14,15]，这样便于内核的裁剪与移植。

对于一个操作系统内核，最基本的功能是任务管理。然而对于一个实用的操作系统内核，仅仅一个任务管理的功能是不够的，还要实现如下的功能：

(1) 任务管理功能：根据任务的优先级及时间片，实现任务在就绪、挂起、运行、休眠、中断五个状态间的转换与控制，使操作系统具有较高的实时性；

(2) 时间管理功能：使应用层的任务可以获取精确的时间，进行时间有关方面的处理，如任务延时一段时间等；

(3) 任务间的通信功能：使不同的任务间可以相互传递信息；

(4) 信号量的处理：解决对临界区资源和共享资源访问的问题；

(5) 内存管理功能：实现内存的动态分配与释放；

(6) 中断处理功能：实现对外部事件的快速处理。

根据系统所要实现的功能，我们将构造任务管理、中断处理、通信、信号量管理、时间管理、内存管理六大模块，由它们构成的 AcoolOS 操作系统内核体系结构如图 3.1 所示。

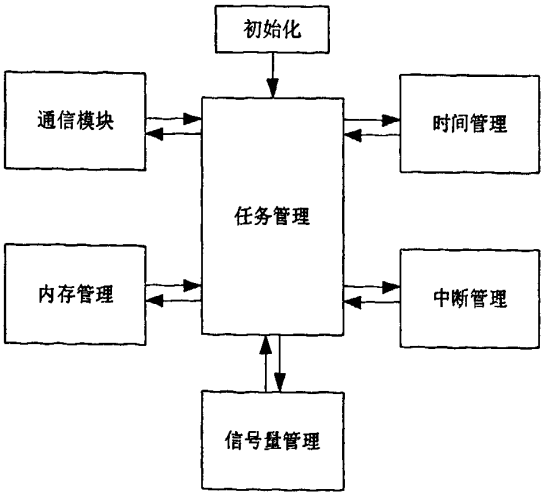

图 3.1 AcoolOS 系统内核体系结构

Fig. 3.1 The architecture of AcoolOS core

3.2 任务管理模块

在多任务系统中，任务是被调度执行和竞争资源的基本单元。实时内核最基本的功能是任务管理，或提供对多线程的支持，这也是为什么在一个多任务的应用中要使用嵌入式实时内核的最基本的原因。在当前的嵌入式应用中，特别是对复杂的应用而言，一般采用多任务的软件结构来实现。那么由谁来对这些任务实施有效的管理，使得他们能够协调地工作呢？内核的任务管理功能将满足这个需求。

任务管理的主要功能包括创建任务、删除任务、改变任务状态(如启动与重启任务，挂起与恢复任务，改变任务优先级等)和查询任务状态(如优先级、属性等)，其核心是任务调度[16]。任务调度策略是否适合嵌入式应用的特定要求，对于应用的实时性能至关重要。下面将着重介绍任务调度。

在操作系统中对任务进行调度有两种基本的调度方式：轮转调度和基于优先级的抢占式调度，前者无法处理任务间可能存在的优先级差别，后者可以保证高优先级的任务得到及时响应。但后者也同时带来了新的问题：当系统中存在几个相同优先级的任务时，就有可能使某一任务独占处理器，直至完成。为此在该方案中采用了基于优先级的时间片轮转调度方式，即在不同优先级的任务之间以抢占方式进行调度，在相同优先级的任务之间采用时间片进行轮转调度[17,18]。

3.2.1 数据结构

任务管理是通过对任务控制块 TCB 的操作来实现的，任务控制块是包含任务相关信息的数据结构，包含了任务执行过程中所需要的所有信息[19]。任务控制块的内容可以随着系统运行过程中内部或外部事件的发生而发生变化。

任务控制块的结构具体定义如下：

```
typedef struct AP_TASK{
        void                *task_stk_ptr;
        struct  AP_TASK     *rdy_pre;
        struct  AP_TASK     *rdy_nxt;
        UINT8                task_pri;
        struct AP_TASK      *sup_dif_pre;
        struct AP_TASK      *sup_dif_nxt;
        struct AP_TASK      *sup_sam_pre;
        struct AP_TASK      *sup_sam_nxt;
        UINT32                    sup_time;
        UINT32                    timer_slice;
        UINT32                    timer_slice_cur;
        UINT8               task_sta;
        UINT8               preempt;
}ap_task;
```

task_stk_ptr：指向为任务分配的内存空间的首地址；

rdy_pre 与 rdy_nxt：用于链接就绪任务链表中的任务控制块，它是一个双向循环链表；

task_pri：表示任务的优先级，范围是 0～255；

sup_dif_pre、sup_dif_nxt、sup_sam_pre 与 sup_sam_nxt：用于链接挂起的任务，它是一个双向十字链表结构，具体结构将在后面详细介绍；

sup_time：指示任务将被挂起的时间；

timer_slice：用于指示任务总的运行时间片；

timer_slice_cur：用于指示任务当前还剩下的运行时间片；

task_sta：用于表示任务的状态，如就绪、运行、挂起等；

preempt：用于表示任务是否具有可抢占性。如其为 0，则表示当任务在运行时不可被其它任务抢占，即使其拥有更高的优先级。

关于任务控制块中的各项的具体应用将在后面各章节中详细介绍。

3.2.2 基于优先级的抢占式调度

操作系统具有抢占性，任务调度的依据就是按照优先级从高到低的顺序进行调度，当具有低优先级的任务正在运行时，有更高优先级的任务准备就绪执行，则高优先级的任务将抢占低优先级的任务运行[20,21,22,23]。下面介绍基于优先级的抢占式调度的具体实现：

(1) 寻找最高优先级的任务

采用优先级调度，在进行任务重新调度时，要从所有就绪任务中选取优先级最高的那个任务。为了确保调度时间的确定性，这一查找过程所耗费的时间不能因为就绪任务数目的变化而无法预测，为此我们采用优先级位图算法。

(a) 设置两个变量

```
UINT32      g_grp_pri_bit;
UINT8       g_sub_pri_bit[32];
```

g_grp_pri_bit 与 g_sub_pri_bit 的关系如图 3.2 所示。g_sub_pri_bit 的每个数组元素对应 256(0～255：0 对应最高优先级，255 对应最低优先级)个优先级中的 8 个优先级，如 g_sub_pri_bit[2]对应的优先级为 16～23，如果对应的优先级存在就绪任务，则相应的二进制位为 1，否则为 0。若 g_sub_pri_bit[2]的第 0 位为 1，则当前存在一个优先级为 16 的就绪任务，若其为 0，则不存在一个优先级为 16 的就绪任务。

g_grp_pri_bit 中的每个二进制位与 g_sub_pri_bit 的一个数组元素相对应。如 g_grp_pri_bit 中的第 2 位对应 g_sub_pri_bit[2]。g_grp_pri_bit 中的每个二进制的值意味着对应 g_sub_pri_bit 数组元素中对应的优先级是否有就绪任务；二进制位的值为 1，表示对应 g_sub_pri_bit 数组元素中对应的优先级有就绪任务，二进制的值为 0，表示对应 g_sub_pri_bit 数组元素中对应的优先级没有就绪任务。

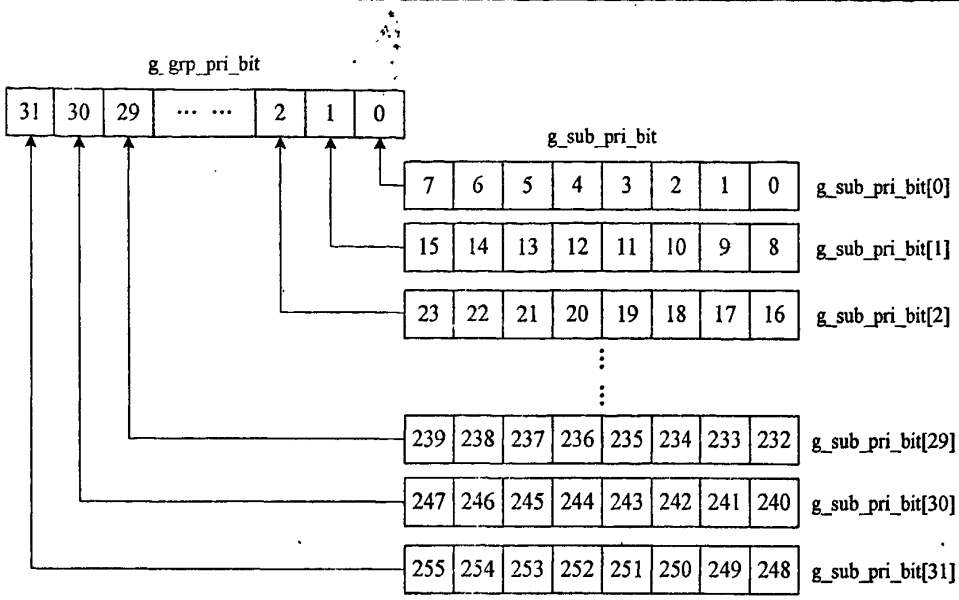

图 3.2 g_grp_pri_bit 与 g_sub_pri_bit 的关系

Fig. 3.2 The relation of g_grp_pri_bit and g_sub_pri_bit

(b) 任务优先级与 g_grp_pri_bit 和 g_sub_pri_bit 的关系

任务优先级与 g_grp_pri_bit 和 g_sub_pri_bit 的关系如图 3.3 所示。优先级数为 256，优先级与 8 个二进制位相对应。在表示优先级的 8 个二进制位中，高 5 位为优先级在 g_grp_pri_bit 的二进制位位置，与 g_sub_pri_bit 的数组元素相对应；低 3 位表示对应 g_sub_pri_bit 的数组元素的值。如对于优先级 16，二进制位表示位 0001 0000，优先级高 5 位为 00010，在 g_grp_pri_bit 中的位序为 2，对应 g_sub_pri_bit[2]；优先级低 3 位为 000，对应 g_sub_pri_bit[2]中的位序 0，即优先级 16。

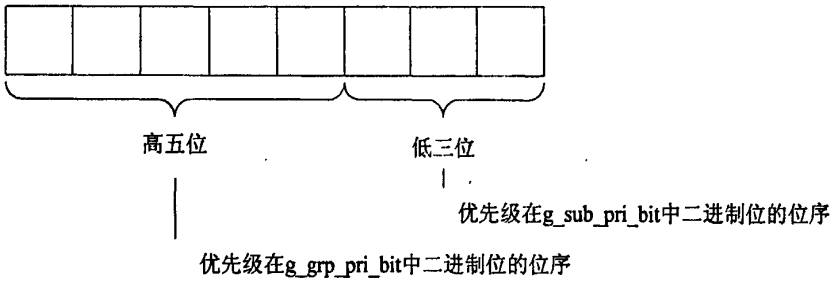

图 3.3 任务优先级与 g_grp_pri_bit 和 g_sub_pri_bit

Fig. 3.3 The relation of Task Priority with g_grp_pri_bit and g_sub_pri_bit

(c) 设置优先级判定表 UINT8　const g_low_set_bit[256]，其定义为：

```
UINT8   const g_low_set_bit[256] = {
                0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0,
                2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0,
                3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0,
                2, 0, 1, 0, 6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, 1, 0,
                2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0,
                3, 0, 1, 0, 2, 0, 1, 0, 7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0,
                2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
                5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0,
                2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 6, 0, 1, 0, 2, 0, 1, 0,
```

```
3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0,
2, 0, 1, 0, 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0};
```

优先级判定表以 g_grp_pri_bit 和 g_sub_pri_bit 数组元素的值为索引，获取该值(g_grp_pri_bit 和 g_sub_pri_bit 数组元素的值的范围为 0x00~0xFF)对应二进制表中 1 出现的最低二进制位的序号(0~7)。如 g_grp_pri_bit 的值为 00010010(0x12)，以 0x12 为下标，对应 g_low_set_bit[256]的数组元素的值为 1，则表示 g_grp_pri_bit 对应二进制表示中 1 出现的最低二进制位的序号为 1。

(d) 任务进入就绪态

任务进入就绪态，需要把任务的优先级转换为在 g_grp_pri_bit 和 g_sub_pri_bit 中的表示，转换过程可用 C 语言描述为：

```
g_grp_pri_bit      |=  1 << (priority >> 3);
g_sub_pri_bit[priority >> 3]  |=  1 << (priority & 0x07);
```

(e) 任务退出就绪态

任务退出就绪态，需要对 g_grp_pri_bit 和 g_sub_pri_bit 进行处理，清除任务优先级在 g_grp_pri_bit 和 g_sub_pri_bit 中的体现，处理过程可用 C 语言语句描述为：

```
if(!(g_sub_pri_bit[priority >> 3] &= (~ (1 << (priority & 0x07)))))
        g_grp_pri_bit &= (~(1 << (priority & 0x07)));
```

首先根据任务优先级的高五位，将其作为索引值，取出数组 g_sub_pri_bit 中的值；根据任务优先级的低三位将相应的“1”清零，如果结果为 0，则表示对应优先级组中所有优先级都不存在对应的就绪任务；在这种情况下，应把该优先级组在 g_grp_pri_bit 中对应的二进制位清除。

(f) 获取进入就绪态的最高优先级

获取进入就绪态的最高优先级是通过优先级判定表来实现的，处理过程用 C 语言语句描述为：

```
if ((1 << (priority >> 3)) & 0xFFul)
    index =  0;
else if ((1 << (priority >> 3)) & 0xFF00ul)
        index =  8;
else if ((1 << (priority >> 3))& 0xFF0000ul)
        index =  16;
else
        index =  24;
temp = g_low_set_bit[((1 << (priority >> 3)) >> index) & 0xFFul];
index = index + temp;
temp = g_sub_pri_bit[index];
pri_highest = (index << 3) + g_low_set_bit[temp];
```

首先寻找 g_grp_pri_bit 中 1 的最低位的序号，然后以此序号为索引值查找数组 g_sub_pri_bit 中相应值的最低位的 1，则其表示的优先级为最高优先级。

(2) 任务切换

将查找到的就绪的最高优先级的任务替换当前正在运行的任务运行。关于任务切换的具体实现将在任务切换一节中详细阐述。

3.2.3 基于时间片的调度

基于时间片的调度，是对相同优先级的任务分别分配一段时间片(不同任务间可有不同的时间片)，当时间片用完后再转到下一个任务，轮流执行直到这个优先级的所有任务全部执行完毕，然后再转到下一个优先级[24,25,26,27,]。下面介绍基于时间片调度的具体实现：

(1) 数据结构

AP_TASK *g_task_timer_slice，用来指示当前正在运行的以时间片调度的任务，如果当前运行的任务不是以时间片调度的，则该指针为一空值。

UINT32 g_com_timer_slice，用来指示当前运行的任务的剩余时间片，即当前任务再运行 g_com_timer_slice 个系统时钟，任务将切换到下一个相同优先级的任务运行。

(2) 当任务被调度执行时

初始化 g_task_timer_slice 与 g_com_timer_slice，并切换任务运行的上下文环境。使 g_task_timer_slice 指向被调度的任务，g_task_timer_slice 为任务将要运行的时间片，C 语言实现如下(task 指向当前正在运行的任务)：

```
g_task_timer_slice = task;
g_com_timer_slice = task->timer_slice_cur;
```

(3) 当任务在运行过程中，发生中断或被更高优先级的任务抢占运行时

在进行上下文环境保护时，还要将任务剩余的时间片保存，并清空 g_task_timer_slice 指针与 g_com_timer_slice 值，C 语言实现如下(task 指向当前被中断或抢占运行的任务)：

```
task-> timer_slice_cur = g_com_timer_slice;
g_task_timer_slice = NULL;
g_com_timer_slice = 0;
```

(4) 当任务的时间片用完时

将任务切换到下一个相同优先级的任务，并复位任务控制块中的 g_timer_slice_cur 值，使其值为任务运行的时间片，以备下次任务被调度运行时所用。C 语言实现如下(task 指向时间片用完的任务)：

```
task->timer_slice_cur = task->timer_slice;
```

由以上在 AcoolOS 中基于优先级和基于时间片的任务调度实现过程，可知任务调度在如下几种情况下发生：

(1) 任务状态转换的时刻，即任务终止、任务睡眠；

(2) 为任务就绪/运行队列增加新的任务时；

(3) 当前运行的任务的时间片用完时；

(4) 内核中断处理完毕时[28]。

3.2.4 任务切换

任务切换是指保存当前任务的上下文，并恢复需要执行任务的上下文的过程。在这里任务的上下文为运行任务的 CPU 的上下文，即所有的寄存器和状态寄存器[29,30]。任务切换用来改变任务的状态，它在以下情况下发生：

(1) 系统进行任务调度时，即选择下一个可运行的任务；

(2) 中断发生时，即有外部事件需要及时相应，此时系统需要保护正在运行的任务的上下文环境，并调度中断服务程序[31,32]。

任务切换将导致任务状态发生变化，当前正在运行的任务将由运行状态变为就绪或等待状态，需要投入运行的任务则由就绪状态变为运行状态。任务切换的具体步骤如下：

(1) 保存处理器上下文环境

具体的上下文环境与采用的处理器有关，且此处代码调度频繁，我们采用汇编语言将寄存器值存入任务栈中(如图 3.4)，并将栈顶值 SP 存入任务控制块中保存。

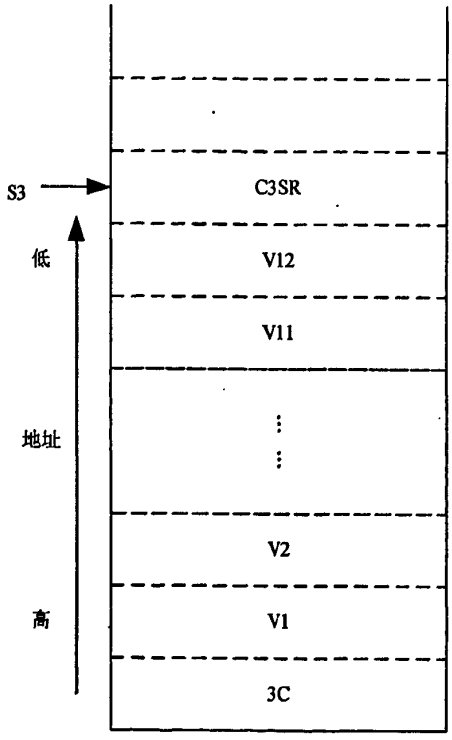

图 3.4 上下文环境在栈中的保存

Fig. 3.4 The storage of context in stack

(2) 更新当前处于运行状态任务的任务控制块的状态位 task_sta，由运行状态改变为就绪(需要响应更高级的任务)或是等待状态(因缺乏资源而阻塞)，并从就绪任务队列 g_task_pri_list 中删除。

(3) 当任务需要被挂起一段固定时间时，需要加入挂起队列 g_task_sup_list 中。

g_task_sup_list 是一个双向循环十字链表(如图 3.5)，在水平方向上，任务按需要挂起的时间，从短到长用任务控制块中的 sup_dif_pre 和 sup_dif_nxt 连接起来，且后一个任务的挂起时间以前一个任务唤醒的时刻为计时基点。例如，假设在某一时刻链表 g_task_sup_list 中只有任务 A，其挂起的时间为 8，此时任务 B 被挂起，时间为 10，则应把任务 B 放在任务 A 的后面，且其挂起时间 sup_time 应设为(10－8)，即为 2，这是因为只有当任务 A 被唤醒时，任务 B 的挂起时间 sup_time 才开始被计时；在竖直方向上，任务按被挂起的先后顺序用任务控制块中的 sup_sam_pre 和 sup_sam_nxt 连接起来，

但需要挂起的时间相等，即它们需要在同一时刻被唤醒。

这样，当链表 g_task_sup_list 不空时，在每一个 Tick 中断中都要将第一个任务的挂起时间减 1，即将任务控制块中的 sup_time 的值减 1，当其值为 0 时，则将任务唤醒。

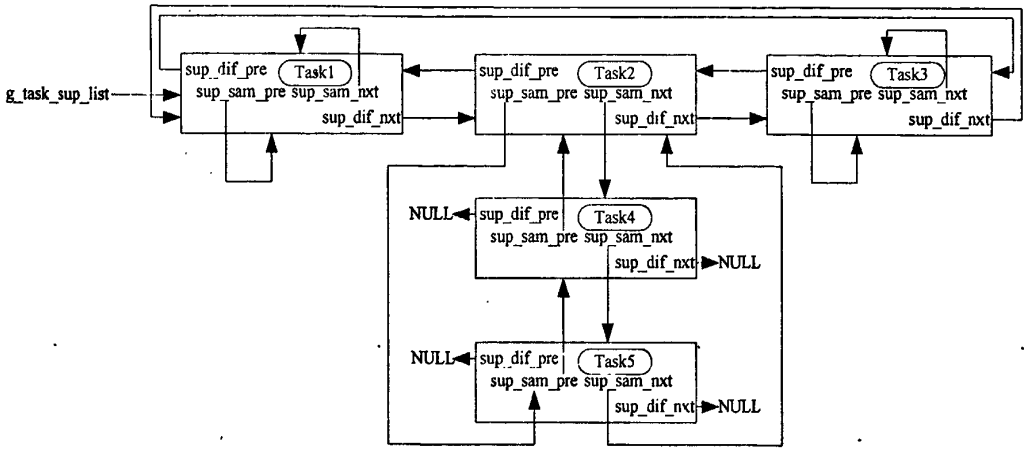

图 3.5 挂起任务链表

Fig. 3.5 The link list of suspended tasks

(4) 选择另一个任务运行。实时内核通过调度程序按着基于优先级的时间片轮转调度方式选取需要投入运行的任务。

(5) 如果选择运行的任务在链表 g_task_sup_list 中，则将其从链表中删除，并将 g_task_sup_list 指向下一个挂起的任务。

(6) 更新需要投入运行的任务的状态位。将其设为运行态，并将其加入到就绪/运行任务链表 g_task_pri_list 中，其中 g_task_pri_list 是一个指针数组(如图 3.6)，共有 256 项，每一项指向一个以数组索引为优先级的就绪任务链表，链表是以任务控制块中的 rdy_pre 和 rdy_nxt 为指针连接而成的双向循环链表。例如，g_task_pri_list[1]指向由优先级为 1 的所有就绪任务组成的链表。

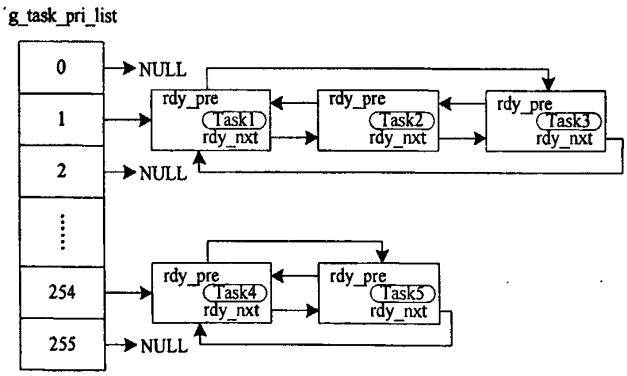

图 3.6 就绪任务链表数组

Fig. 3.6 The link list array of ready tasks

(7) 根据任务控制块，恢复需要投入运行的任务的上下文环境，即恢复在任务栈中保存的所有寄存器和状态寄存器的值。

3.3 中断管理模块

中断是一种硬件机制，用于通知 CPU 有某个异步事件发生了，中断一旦被识别，CPU 保存部分(或全部)上下文环境，即部分或全部寄存器的值，跳转到专门的子程序，称为中断服务程序(ISR)。中断服务子程序用于事件处理，处理完成后，程序进行调度，选择最高优先级的任务继续运行[33]。

中断本身是一种异步机制，中断服务程序不需要内核的调度就可以执行，但是在嵌入式实时应用中，要求 ISR 和其他应用任务之间协同工作，以快速、合理地响应外部事件，并完成后续的处理过程。与普通的任务相比，中断有以下不同：

(1) 中断本身没有自己的堆栈空间，中断的执行占用被中断的任务的空间或设置专门的中断堆栈。

(2) 中断本身不能被其它任务或自己挂起，除非有更高级的中断发生且系统支持嵌套中断，否则中断会连续的执行直到完毕。

(3) 中断的执行是对外部事件的响应，不需要内核的调度且执行的时间及次数等具有不确定性。

3.3.1 中断的分类

对于多个中断可以采用两种处理方式：非嵌套的中断和嵌套的中断处理方式。

(1) 非嵌套的中断

在处理一个中断的时候，禁止再发生中断，这种处理方法称为非嵌套的中断处理方式。在非嵌套的中断方式中，处理中断的时候，将屏蔽所有其它的中断请求。在这种情况下，新的中断将被挂起；当处理器再次允许中断时，再由处理器进行检查。因此，如果程序执行过程中发生了中断，在执行中断服务程序的时候将禁止中断；中断服务程序执行完成后，恢复正常执行流程被中断的程序之前再使能中断，并检查处理器是否还有中断。非嵌套的中断处理方式使中断能够按发生顺序进行处理。这种处理方式的缺点是没有考虑优先级，使高优先级的中断不能得到及时的处理，只有在当前中断服务退出后，其它的中断程序才可能等到执行。

(2) 嵌套的中断

定义中断优先级，允许高优先级的中断打断低优先级中断的处理过程，这种处理方法称为嵌套的中断处理方式。在嵌套的中断处理方式中，中断被划分为多个优先级，中断服务程序只屏蔽那些低的或相同优先级的任务，在完成必要的上下文保存后即可使能中断。高优先级中断请求到达的时候，需要对当前中断服务程序的状态进行保存，然后调用高优先级的中断的服务程序，当高优先级中断服务程序执行完成后，再恢复先前的中断服务程序继续执行。

相比较两种处理方式，嵌套的中断处理方式对高优先级的中断响应更快，但嵌套的中断要求的堆栈空间要相应的增长，且调试时对中断的跟踪更加困难，因此最终决定采用非嵌套的中断处理方式。

3.3.2 中断的处理

中断是对外部和内部事件提供立即响应服务的机制，也是操作系统实时性的重要体现。为了加快中断的响应并尽量缩短中断的处理时间，我们将中断服务分为低级中断服务(LISR)与高级中断服务(HISR)。在 LISR 中做尽量少的工作，以便尽快地响应其它的中断，将大部分可稍后处理的工作交给 HISR 完成。HISR 类似于任务，有自己的堆栈空间，但它的优先级比一切任务的优先级高。

(1) 低级中断服务 LISR

LISR 与普通中断服务程序一样，没有自己的堆栈，我们给其分配专门的空间来处理中断。

(a) 数据结构

```
void  (*g_lisr_fun[MAX_VECTOR_INT])(void);
```

函数指针数组，用来表明每个中断向量对应的函数，数组项的索引值与中断向量对应。MAX_VECTOR_INT 表示中断向量的数目，在应用的芯片 S3C2440A 上有共有 31 个中断向量，故其值为 31。

(b) 注册中断

在应用中断之前，先要注册中断，即将中断函数的地址放入函数指针数组中，使中断向量与中断函数联系起来。C 语言实现如下：

g_lisr_fun[Vector] = lisr_fun，其中 Vector 表示中断向量，lisr_fun 表示中断函数的地址。

(c) LISR 中断的处理

当中断发生时，将调用相应的中断服务程序，LISR 的主要功能如下：

- 中断前导：关闭中断，保存中断现场；
- 用户中断服务程序：完成对中断的具体处理；
- 中断后续：打开中断，恢复任务调度。

其中中断前导与中断后续属于中断管理程序，需要由操作系统来实现。

中断前导，即当中断发生时，首先要关闭中断，防止嵌套中断的发生。然后保留被打断任务的上下文环境，并设置栈的指针指向分配给中断的专用空间，然后调用中断处理程序(*g_lisr_fun[Vector])()，其中 Vector 为中断向量。

中断后续，即处理完中断后，打开中断，然后执行调度程序，恢复任务的运行。此时由于中断是非嵌套的且中断程序已执行完毕，故对于中断处理完后的现场不用保存，当再次发生中断时，只需将中断专用的堆栈分配给中断程序即可。

(2) 高级中断服务程序 HISR

相对于 LISR 用来处理中断中少量的、必要的工作，HISR 用来处理中断中其它的后续工作。HISR 由 LISR 激活，由于在 HISR 中可以响应其它的中断，所以 HISR 有其自己的任务控制块和堆栈空间，用来保存上下文环境，且其可以调用系统的大多数服务例

程。

(a) 数据结构

```
typedef struct AP_HISR{
    UINT32              *hisr_stk_ptr;
    UINT32              *hisr_stk_bak;
    structAP_HISR       *hisr_nxt;
    void                (*hisr_fun)(void);
    UINT8               act_count;
    UINT8               hisr_pri;
}AP_HISR;
```

hisr_stk_ptr：HISR 的栈顶指针值，其值随着 HISR 的执行而更改；

hisr_stk_bak：为 HISR 分配的栈空间的起始地址，其值在每一个 HISR 的生命期间保持不变；

hisr_nxt：用于在链表中指向下一个被激活的相同优先级的 HISR；

hisr_fun：指向 HISR 的执行函数；

act_count：HISR 被激活的次数。当被 LISR 激活时，其值加 1，当 HISR 函数执行完毕时，其值减 1；

hisr_pri：HISR 的优先级，范围为 0～2，共三个优先级，其中 0 表示的优先级最高，1 表示的优先级最低。

(b) HISR 调度方式

对于 HISR 的调度我们采用基于优先级的调度方式，对于具有相同优先级的任务，我们采用先来先服务的原则。由于 HISR 只有三个优先级，比任务调度中的 256 个优先级要少的多，故我们寻找最高优先级 HISR 采用简化的方式。

首先设置两变量：AP_HISR　　*g_hisr_head[3], *g_hisr_tail[3];

其中 g_hisr_head 与 g_hisr_tail 分别指向被激活的 3 个不同优先级的 HISR 组成的单向链表的头指针与尾指针，数组的索引值与其所指向的 HISR 的优先级相对应，0 表示的优先级最高，2 表示的优先级最低。当对应的优先级没有被激活的 HISR 时，则对应的指针值为空，具体结构如图 3.7 所示。

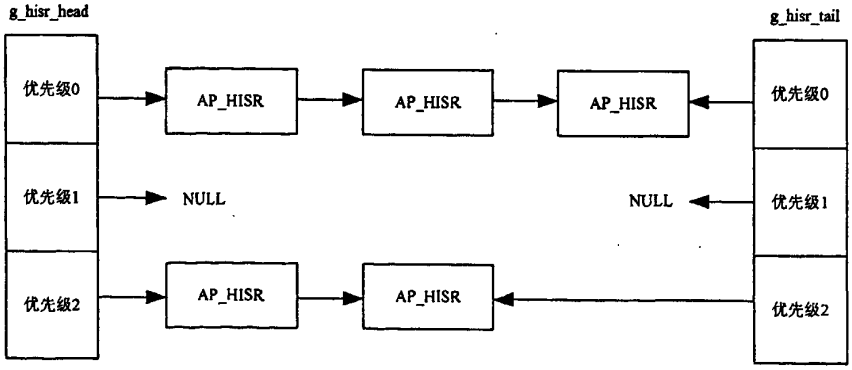

图 3.7 激活 HISR 链表数组

Fig. 3.7 The link list array of activated HISR

当激活 HISR 时，如果此 HISR 已被激活，则只是简单的将 HISR 控制块中的计数

值 act_count 加 1，否则需要将 HISR 加入由 g_hisr_head 与 g_hisr_tail 指向的 HISR 链表的链尾，并将计数值 act_count 设置为 1，表示目前只被激活一次。C 语言实现如下：

```
if(Hisr->act_count == 0) {
    if(g_hisr_head[Hisr->hisr_pri] == NULL) {
        g_hisr_head[Hisr->hisr_pri] = Hisr;
        g_hisr_tail[Hisr->hisr_pri] = Hisr;
    }else{
    (g_hisr_tail[Hisr->hisr_pri])->hisr_nxt = Hisr;
    g_hisr_tail[Hisr->hisr_pri] = Hisr;
    }
}
```

当 HISR 被调度时，由于 HISR 的优先级高于一切任务，所以在调度时，我们首先判断是否有被激活的 HISR，如有被激活的 HISR，则优先调度 HISR 执行。在被激活的 HISR 中，又有不同的优先级，我们首先判断 g_hisr_head[0]是否有被激活的 HISR，如有，则根据相应 HISR 控制块中的栈顶指针 hisr_stk_ptr 调用相应的 HISR 函数 hisr_fun；如果 g_hisr_head[0]没有被激活的 HISR，则再依次判断 g_hisr_head[1]与 g_hisr_head[2]，这样就能保证 HISR 按优先级的顺序被调度。

HISR 被中断时，在 HISR 执行的过程中，当发生中断时，则需要保存中断现场，将所有寄存器和状态寄存器的值存入任务的栈中，并将栈顶指针的值存入 HISR 控制块的 hisr_stk_ptr 中。

当 HISR 执行完毕时，将控制块中的计数值 act_count 减 1，并检查其值是否为 0，如不为 0，则继续执行 HISR 函数 hisr_fun，直到 act_count 为 0，然后将栈顶指针 hisr_stk_ptr 的值更新为为 HISR 分配的空间的起始地址 hisr_stk_bak，并将此 HISR 的控制块从链表中删除，C 语言实现如下：

```
do{
    (*(Hisr->hisr_fun))();
}while(--(Hisr->act_count));
Hisr->hisr_stk_ptr = Hisr->hisr_stk_bak;
g_hisr_head[Hisr->hisr_pri] = Hisr->hisr_nxt;
Hisr->hisr_nxt = NULL;
```

3.4 时间管理模块

在AcoolOS中，由时间管理模块提供所有的计时服务。计时的基本单位是滴答(tick)，其与单个硬件定时器中断相对应，每个tick的时间片可进行编程确定，时间片越短，时钟精度越高，但系统的额外开销也就越大，在此将其设为2ms。

3.4.1 时间管理模块的功能

时间管理在 AcoolOS 中主要提供以下功能：

(1) 提供系统时钟；

(2) 为任务的睡眠、挂起等提供计时服务；

(3) 为以时间片调度的任务提供计时服务；

(4) 为周期性调度的函数提供计时服务。

对于功能(2)与功能(3)已在任务调度章节中详细介绍，下面将要着重介绍功能(1)和功能(4)的实现。

3.4.2 系统时钟

系统时钟是由 AcoolOS 内部维护的一个计数器，其值可由用户访问，每经过一个 tick 时间片，计数器的值加 1。

系统时钟的实现比较简单，设置一个全局变量 SystemClock，用来记录计数值。当发生定时器中断时，将计数值 SystemClock 加 1，用户访问系统时钟的值即可通过访问 SystemClock 来实现。

3.4.3 周期性调度的时钟

许多实时应用需要进行周期性地处理，为此需要提供一个定时器，每当其超时时，则调用相应的函数进行处理。周期性调度的时钟基准是 tick 中断，采用倒数计数机制实现。tick 中断的时长由 CPU 时钟分频决定。系统只提供中断服务程序，在该服务程序中对定时器的计数值进行倒数自减处理。内核设置了一个双向链表对定时器进行管理，内核根据 tick 中断的到来，对定时器链表进行扫描，对所有计数值进行自减，并判断是否为 0，如果为 0，则对该定时器所属的函数进行相应的周期性调度。

(1) 定时器控制块的结构

```
typedef  struct  AP_TIMER{
         UINT32                time_init;
         UINT32                time_loop;
         UINT32                time_cur;
         void                  (*timer_fun)(void);
         struct  AP_TIMER  *tim_pre;
         struct  AP_TIMER      *tim_nxt;
}AP_TIMER;
```

time_init：定时器的初始值，即当打开定时器后，第一次调用相应处理函数的时间段，以 tick 为单位；

time_loop：定时器的周期值，即每次调用处理函数的时钟周期(第一次调用除外)，当其值为 0 时，表示只调用一次处理函数；

time_cur：定时器当前值，即还剩余多少时间将调度处理函数；

timer_fun：周期性调度的函数的地址；

tim_pre：用于定时器链表中连接前一个定时器；

tim_nxt：用于定时器链表中连接下一个定时器。

(2) 定时器链表

为方便定时器的管理，将所有建立的定时器连接起来，构造一个双向链表，并定义一个头指针指向它：AP_TIMER　　*g_com_app_timer(如图 3.8)。

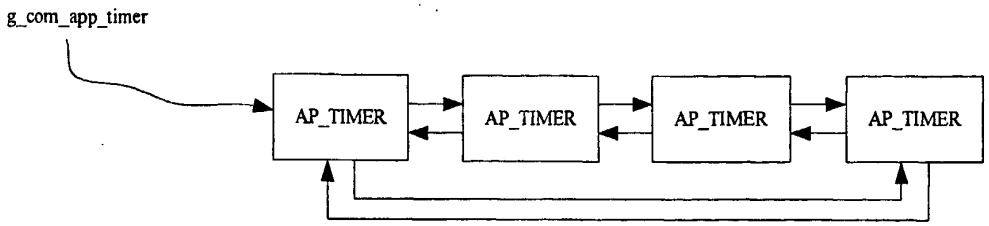

图 3.8 定时器链表

Fig. 3.8 The link list of created timer

(3) 定时器的建立

定时器的建立主要是初始化定时器控制块的各个参数，设定定时器的初始值 time_init，并将 tim_pre 与 tim_nxt 设置为空，表示此定时器还未放入定时器链表中。

(4) 定时器的启动

定时器的启动是将定时器控制块放入 g_com_app_timer 指向的链表中，并初始化控制块中定时器当前值 time_cur 设为初始值 time_init。C 语言实现如下(Timer 指向将要启动的定时器控制块的地址)：

```
if(Timer->tim_pre == NULL && Timer->tim_nxt == NULL){
    if(g_com_app_timer == NULL) {
    g_com_app_timer = Timer;
        Timer->tim_nxt = Timer;
        Timer->tim_pre = Timer;
    }else{
        Timer->tim_pre = g_com_app_timer->tim_pre;
        (g_com_app_timer->tim_pre)->tim_nxt = Timer;
        Timer->tim_nxt = g_com_app_timer;
        g_com_app_timer->tim_pre = Timer;
    }
    Timer->time_cur = Timer->timer_init;
}
```

(5) 定时器的运行

当定时器启动后，内核以一定 tick 中断的时长为步长对 time_cur 进行自减，当 time_cur 为 0 时，表示设定的时钟周期到，开始执行相应的处理函数。

(6) 定时器处理函数的执行

对于定时器，并没有为其分配独享的堆栈空间。AcoolOS 内核是通过激活一个 HISR，在 HISR 中执行处理函数的，处理函数的执行占用的是 HISR 堆栈空间，所以编写的处理函数一般都要求占用较小的空间，且能较快的执行，不能将自己挂起。

(7) 定时器的终止

当不想再继续周期性处理时，需要终止定时器，这时只要将定时器控制块从定时器链表中删除就可以了，这样在 tick 中将不再维护计时值 time_cur，也不会再调用相应的周期处理函数。C 语言实现如下：

```
if(Timer->tim_pre != NULL && Timer->tim_nxt != NULL){
    if(Timer->tim_pre == Timer) {
        g_com_app_timer = NULL;
    }else{
```

```
            if(g_com_app_timer == Timer)
            g_com_app_timer = Timer->tim_nxt;
            Timer->tim_pre->tim_nxt = Timer->tim_nxt;
        Timer->tim_nxt->tim_pre = Timer->tim_pre;
        }
    Timer->tim_pre = NULL;
    Timer->tim_nxt = NULL;
    }
```

(8) 定时器周期值的重新设定

当需要更改定时器的周期值时，只需要更改定时器控制块中参数 time_loop 即可，如果此时定时器正在运行，则更改的值将在下一次周期调度时生效。

3.5 通信模块

在嵌入式多任务应用程序中，一项工作的完成往往要通过多个任务，或者多个任务与中断处理程序(ISR)共同完成。它们之间必须协调动作，互相配合，甚至需要交换数据或信息，即进行通信[34,35]。

3.5.1 通信方式

任务间的通信方式可以有直接通信和间接通信两种[19]：

(1) 直接通信

直接通信是指在通信过程中双方必须明确地知道(命名)彼此。采用类似下面的通信原语：

Send(P，message)——发送一个消息到任务 P；

Receive(Q，message)——从任务 Q 接收一个消息。

在通信双方之间可以说存在一个链接，该链接具有如下特性：

- 一个链接仅与一对相互通信的任务相联系；
- 每对任务之间仅存在一个链接；
- 链接可以是单向的，也可以是双向的。

(2) 间接通信

在间接通信方式中，通信的双方不需要指出消息的来源或去向，即发送者不指出消息将发送给谁，而接收者也不指出从谁那儿接收消息。消息发送到邮箱，从邮箱中接收消息，采用类似下面的通信原语：

Send(A，message)——发送一个消息给邮箱 A；

receive(A，message)——从邮箱 A 中接收一个消息。

每个邮箱有一个惟一标识(比如 ID 号)。

间接通信方式中通信链接的特性如下：

- 只有当任务共享一个公共邮箱时链接才建立；
- 一个链接可以与多个任务相联系；
- 每对任务可以使用几个通信链接；
- 链接可以是单向或双向的。

3.5.2 通信机制

在实时系统中，通常采用邮箱与消息队列机制来传递消息[36]，由于消息队列机制较邮箱机制功能更强大，故在此系统中只是实现了消息队列机制。

消息是内存空间中一段长度可变的缓冲区，其长度和内容均可以由用户定义，其内容可以是实际的数据、数据块的指针或空。消息机制在任务和任务之间、任务和中断服务程序之间提供消息传送(通信)机制。

对消息内容的解释由用户完成。从操作系统观点看，消息没有定义的格式，所有的消息都是字节流，没有特定的含义。从应用的观点看，根据应用定义的消息格式，消息被解释成特定的含义。最简化的情况是，应用对消息格式也不做定义，只把消息当成一个标志，这时消息机制用于实现同步，任务可以在一个空消息队列上等待其他任务发出的消息，以实现两个任务间的同步。

消息队列可存放若干消息，提供了一种任务间缓冲通信的方法。发送消息的请求将消息放入队列，而接收消息的请求则将消息从队列中取出，消息队列中消息的数量及每个消息的长度可由用户自己定义。

3.5.3 消息队列机制的设计与实现

(1) 消息队列机制采用的主要数据结构

AcoolOS 内核使用消息队列控制块来管理所有创建的消息队列，在系统运行时动态地分配和回收消息队列控制块。

每个消息队列都有对应的消息缓冲区，以存放发送到该队列的消息，而接收者从缓冲区中取出消息。当队列满时，发送消息的任务可以被挂起，直到队列中有接收消息的空间；当队列空时，接收消息的任务可以被挂起，直到队列中有新的消息到达。

消息队列控制块的结构定义：

```
typedef  struct   AP_QUEUE{
         UINT32                  *que_add_start;
         UINT32                      *que_add_end;
         UINT32                  *que_read;
         UINT32                      *que_write;
         UINT32                      mesg_size;
         UINT32                      mesg_num;
         UINT32                      mesg_room;
         struct  QUE_SUP_BLK     *que_wait_list;
         UINT8                       fifo_pri;
}AP_QUEUE;
```

que_add_start：消息队列的起始地址；

que_add_end：消息队列的结束地址；

que_read：用于从消息队列中读取消息，指向队列中的第一个可被读取的消息；

que_write：用于向消息队列中传送消息，指向队列中用来存放消息的第一个空的地址空间；

mesg_size：每个消息占用空间的大小，以字(四字节)为单位；

mesg_num：消息队列中当前存放的消息的数目；

mesg_room：消息队列中还可以存放多少条消息；

que_wait_list：用于指向因为队列中没有消息被读取或没有空间存放消息而挂起的任务组成的链表；

fifo_pri：用于表示当任务需要被挂起时，是按“先进先出”的顺序，还是按“优先级”的顺序挂起在任务链表中。

为了便于消息队列的管理，将因为发送消息或接收消息而挂起的任务连接起来，每个链表结点的定义如下：

```
typedef  struct   QUE_SUP_BLK{
    AP_TASK                    *sup_task;
    struct     QUE_SUP_BLK          *sup_pre;
    structQUE_SUP_BLK          *sup_nxt;
    UINT32                          *mesg_area;
    AP_BOOL                    send_rev_success;
}QUE_SUP_BLK;
```

sup_task：指向挂起的任务控制块；

sup_pre：指向链表中的前一个结点；

sup_nxt：指向链表中的后一个结点；

mesg_area：指向将要发送的消息或用来存放接收到的消息的地址；

send_rev_success：指示任务是否成功发送或接收消息。

(2) 消息队列机制的实现

消息队列提供如下主要功能：

- 创建消息队列
- 发送消息
- 接收消息

随着任务(或中断服务程序(ISR))不断地向(从)消息队列发送(接收)消息，消息队列的状态不断转换，可以有如下几种状态：

- 消息队列为空(此时消息队列中没有存放一条消息)
- 消息队列为空且有任务等待接收消息
- 消息队列中有消息，但未满
- 消息队列满
- 消息队列满，且有任务等待向它发送消息。

(a) 创建消息队列

消息队列可以被应用动态地创建和删除。对于一个应用可以拥有的消息队列数从理论上讲没有限制，但是每个消息队列都需要一个控制块，所以受到实际存储空间的限制。

创建一个消息队列时，调用者需要指定消息的最大长度，以及每个消息队列中最多的消息数，并指明任务等待消息时的排队方式。

多个任务可能在同一个消息队列中被挂起，任务等待消息的方式可以选用以下两种

方式中的一种：

- 任务按先进先出(FIFO)方式等待
- 任务按优先级(Priority)方式等待

如果消息队列选择 FIFO 挂起，任务以它们被挂起的时间先后顺序恢复执行；如果消息队列支持优先级挂起方式，任务以优先级的高低顺序为恢复执行的顺序。

消息队列最根本的部分是一个循环缓冲区，用两个指针 que_read 与 que_write 分别指向读取消息与存放消息的地址，如图 3.9 所示(阴影部分表示队列中存放的消息)。

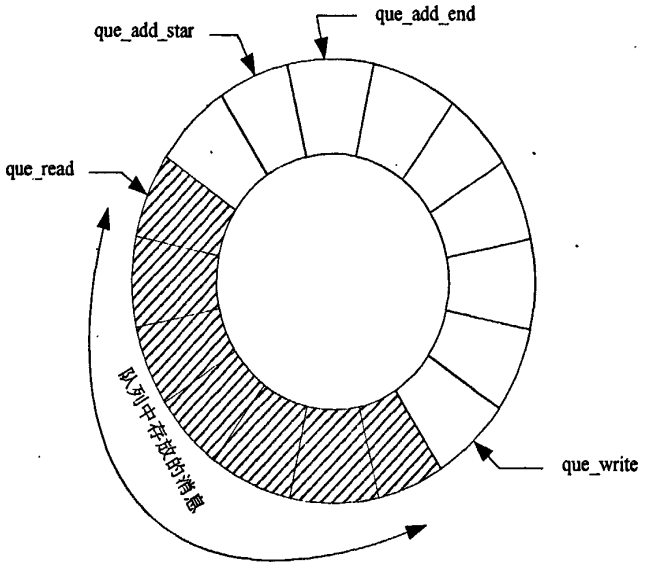

图 3.9 消息队列结构

Fig. 3.9 The architecture of information queue

建立消息队列就是要为消息队列分配一个用来存放消息的缓冲区，并初始化队列控制块中的各个参数变量。

(b) 发送消息

当发送消息的时候，要根据消息队列中存放消息的不同状态做出不同的选择：

当消息队列已满，且任务不允许挂起时，则立即返回，返回值为 Fail；如果任务允许挂起，则根据消息队列控制块中参数项 fifo_pri 的值，按“先进先出”的顺序或“优先级”顺序将发送消息的任务挂起。

当消息队列为空且有等待读取消息的任务时，则将消息传送给挂起的任务并将其唤醒。

当消息队列未满且没有等待读取消息的任务时，将要发送的消息存入消息队列中。

发送消息的流程图如图 3.10 所示。

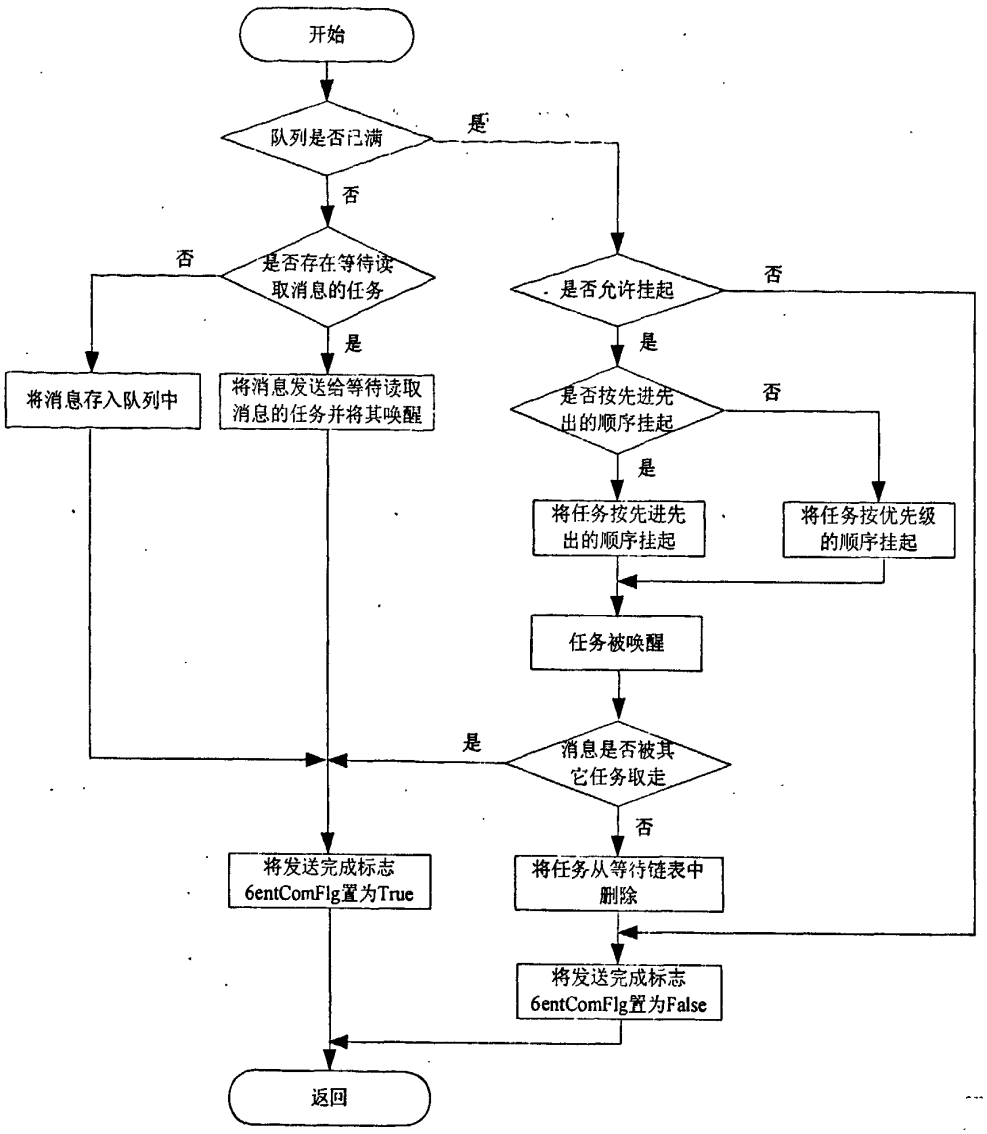

图 3.10 发送消息流程图

Fig. 3.10 The flowchart of sending information

(c) 接收消息

接收消息即从队列中读取消息，要根据消息队列中存放消息的不同状态做出不同的选择：

当消息队列不空且无等待发送消息的任务时，从消息队列中读取消息并返回 Success。

当消息队列不空但有等待发送消息的任务时，从消息队列中读取消息，并将等待发送消息的任务唤醒，返回 Success。

当消息队列为空且不允许任务挂起时，返回 Fail；

当消息队列为空但允许任务挂起时，根据消息队列控制块中参数项 fifo_pri 的值，按“先进先出”的顺序或“优先级”顺序将接收消息的任务挂起。

接收消息的流程图如图 3.11 所示。

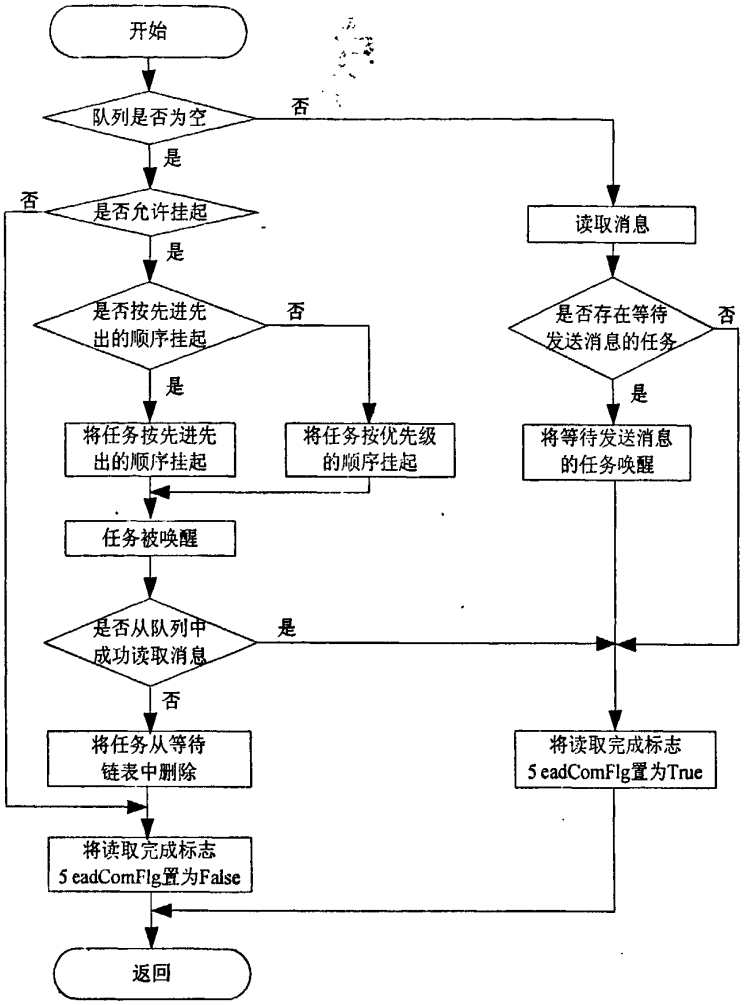

图 3.11 接收消息流程图

Fig. 3.11 The flowchart of receiving information

3.6 信号量模块

信号量用于实现任务与任务之间、任务与中断处理程序之间的同步与互斥。

3.6.1 信号量的分类

根据用途，可将信号量分为两种：用于解决互斥问题的互斥信号量和用于解决共享资源问题的计数信号量[3T,38]。

用信号量保护的代码区被称为“临界区”，当用作互斥时，信号量的初始值设为 1，表明目前没有任务进入“临界区”，但最多只有一个任务可以进入“临界区”。因此第一个试图进入“临界区”任务将成功获得信号量，而所有其它的任务就必须等待。但目前在“临界区”中的任务离开“临界区”时，将释放信号量并允许第一个正在等待的任务进入“临界区”，这种二值的信号量称为互斥信号量[39]。

计数模式的信号量最常用的情况就是控制对多个共享资源的使用，这样的一个信号量允许多个任务同时访问同一种资源的多个实例，因此信号量的初始值设为 n(非负整数)，n 为该共享资源可被同时访问的数目。

3.6.2 互斥信号量的设计与实现

对一个共享资源进行互斥访问的方法包括禁止中断、禁止任务切换和用信号量锁定资源等，前两个方法相对简单，但是有较大的缺陷。禁止中断将完全占有 CPU，可能增加中断延迟，影响系统的实时性。禁止任务切换则是不加选择地阻止所有其它任务的执行，包括与该资源完全无关的更高优先级的任务。使用信号量是最好的选择，因为它只影响实际竞争该资源的任务，因此它的精度更高，影响范围更小。

(1) 优先级反转问题

使用互斥信号量，首先要解决优先级反转的问题。

理想情况下，当高优先级任务处于就绪状态后，高优先级任务就能够立即抢占低优先级任务而得到执行。但在有多个任务需要使用共享资源的情况下，可能会出现高优先级任务被低优先级任务阻塞，并等待低优先级任务执行的现象。高优先级的任务需要等待低优先级任务释放资源，而低优先级任务又正在等待中等优先级任务的现象，称为优先级反转(priority inversion)。

本系统不是通过更改任务的优先级来解决此问题的。当高优先级的任务不能获得临界系统资源时，高优先级的任务并没有挂起，所以比它优先级低的任务都不能运行，系统会把拥有资源的低优先级任务临时提升为当前任务运行，相当于临时把优先级提升为最高，当低优先级的任务释放临界资源后，高优先级的任务会马上运行。

(2) 创建互斥信号量

互斥信号量是通过互斥信号量控制块来实现的，其类型定义如下：

```
typedef    struct   CRITICAL_PROTECT{
    AP_TASK            *protected_task;
    UINT32             *wait_flag;
}CRITICAL_PROTECT;
```

protected_task：指向占用临界区资源的任务，初始为空；

wait_flag：表明是否有其它任务在等待占用此临界区资源—1：有 0：无。

(3) 获取临界区资源

获取临界区资源时，如果有其它任务占用临界区资源，则要将 CPU 控制权交给它，让其先执行，直到释放临界区资源，以避免出现优先级反转的问题。然后该任务占用临界区资源，即设置互斥信号量控制块中的成员值，使 protected_task 指向占用临界区资源的任务控制块。获取临界区资源的程序流程图如图 3.12 所示。

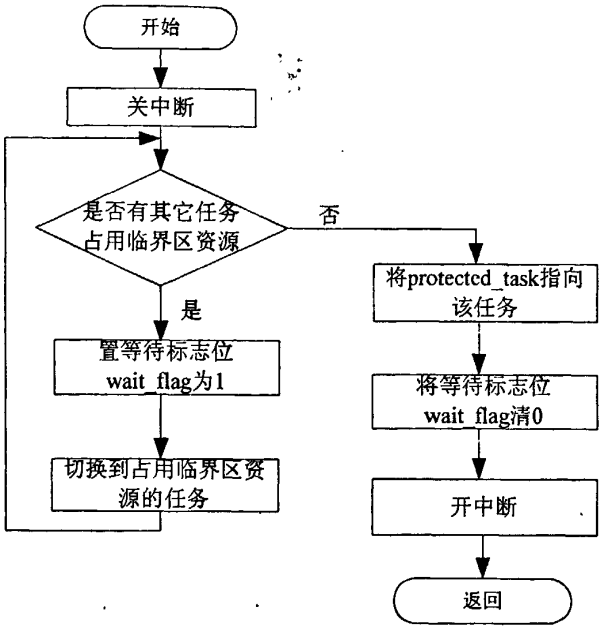

图 3.12 获取临界资源流程图

Fig. 3.12 The flowchart of accessing critical resource

(4) 释放临界区资源

释放临界区资源时，如果有其它任务在等待访问该临界区，则将运行环境切换到该任务。程序流程如图 3.13 所示。

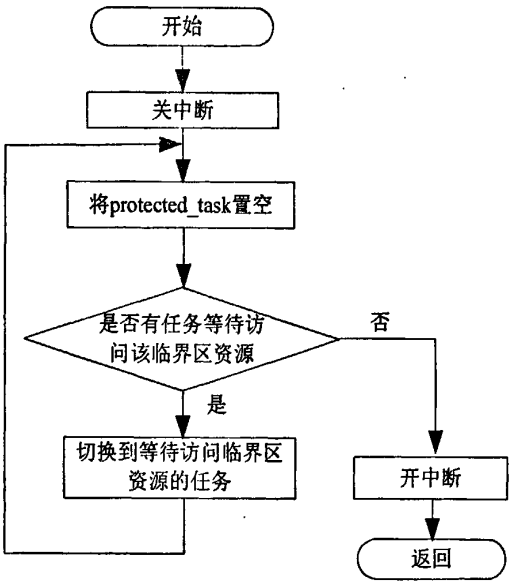

图 3.13 释放临界资源流程图

Fig. 3.13 The flowchart of releasing critical resource

3.6.3 计数信号量的设计与实现

计数信号量用于控制系统中共享资源的使用。当某一资源的所有实例均被分配出去以后，若还有任务要求访问此资源，则信号量机制将阻止其对资源的访问，以免发生死锁等问题。

(1) 创建计数信号量

计数信号量的管理是通过计数信号量控制块来实现的，其定义如下：

```
typedef  struct  AP_SEMAPHORE{
    struct  SEM_SUP_BLK   *sem_wait_list;
UINT32                    sem_wait_cnt;
UINT32                    sem_lft_cnt;
AP_SUSPEND_TYPE           fifo_pri;
}AP_SEMAPHORE;
```

sem_wait_list：用于指向因等待访问资源而挂起的任务链表，关于链表的定义在后面介绍；

sem_wait_cnt：表示因等待访问资源而被挂起的任务的数目；

sem_lft_cnt：表示当前可被访问的资源的数量；

fifo_pri：用于表示当任务需要被挂起时，是按“先进先出”的顺序，还是按“优先级”的顺序挂起在任务链表中。

为了便于计数信号量的管理，将因等待访问资源而挂起的任务连接起来，组成双向循环链表，其链表结点的定义如下：

```
typedef  struct  SEM_SUP_BLK{
         struct     SEM_SUP_BLK          *sup_pre;
         struct     SEM_SUP_BLK          *sup_nxt;
         AP_TASK                      *sup_tsk;
         AP_BOOL                       get_sem_success;
}SEM_SUP_BLK;
```

sup_pre：指向链表中的前一个结点；

sup_nxt：指向链表中的后一个结点；

sup_tsk：指向挂起的任务；

get_sem_success：用于表示任务是否成功分配到共享的资源。

创建计数信号量要给信号量赋初值，并清空等待信号量的任务链表。

(2) 获取信号量

申请信号量即可以在任务中进行，也可以在中断服务程序中进行。在中断服务程序中申请信号量必须选择“不等待”，因为中断服务程序不能被阻塞。

当获取信号量的时候，要根据资源分配的不同状况做出不同的选择：

当有未被分配的资源时，将可用资源的数值减 1，返回 Success。

当资源全部被其它任务占用时，如果任务不允许挂起，则立即返回，返回值为 Fail；如果任务允许挂起，则根据消息队列控制块中参数项 fifo_pri 的值，按“先进先出”的顺序或“优先级”顺序将发送消息的任务挂起。

获取信号量的程序流程图如图 3.14 所示。

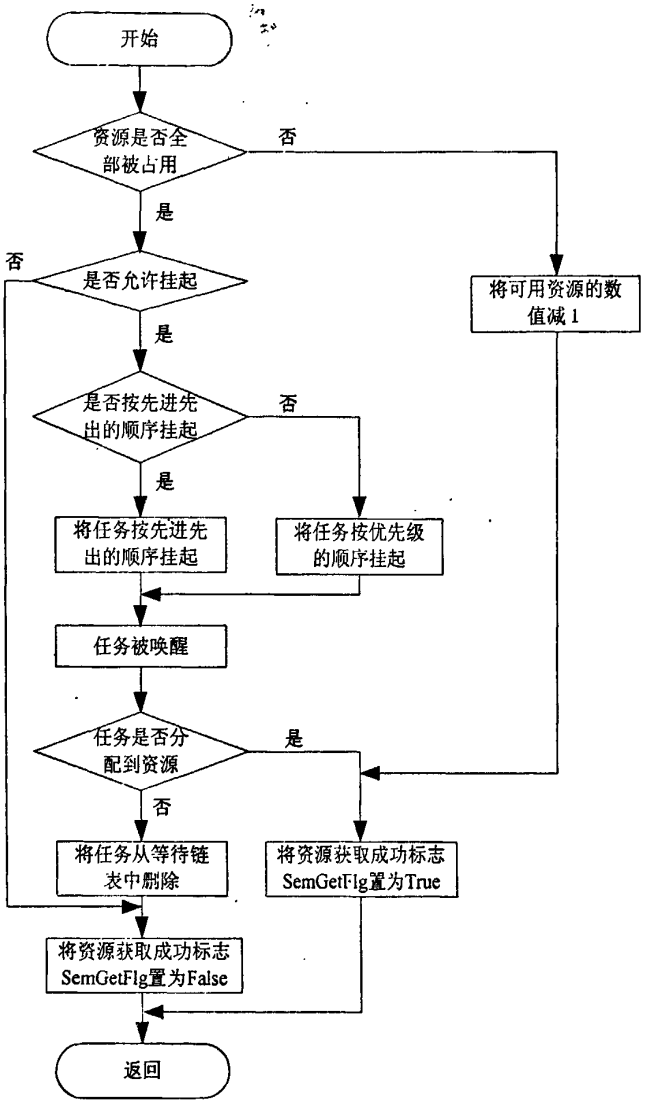

图 3.14 获取信号量流程图

Fig. 3.14 The flowchart of getting semaphore

(3) 释放信号量

释放一个应用指定的信号量时，如果还有其它任务在等待该信号量，则将其唤醒并为其分配信号量，程序流程如图 3.15 所示。

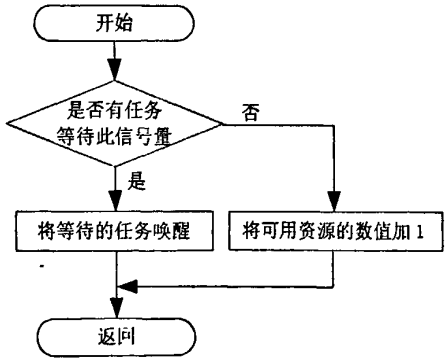

图 3.15 释放信号量流程图

Fig. 3.15 The flowchart of releasing semaphore

3.7 内存管理模块

3.7.1 内存管理概述

作为嵌入式系统的重要组成部分，内存管理必须满足以下特性[41,42]：

(1) 实时性

从实时性的角度出发，要求内存分配过程要尽可能地快。因此，在嵌入式系统中，不可能采用通用操作系统的一些复杂而完善的内存分配策略，一般没有段页式的虚存管理机制；而是采用简单、快速的内存分配方案，其分配方案也因程序对实时性的要求而异。

(2) 可靠性

嵌入式系统应用的环境千变万化，在有些特定情况下，对系统的可靠性要求极高，内存分配的请求必须得到满足，如果分配失败则可能会带来灾难性的后果。

(3) 高效性

内存分配要尽可能地减少浪费。不可能为了保证满足所有的内存分配请求而将内存配置得很大。一方面，嵌入式系统对成本的要求使得内存在其中只是一种很有限的资源；另一方面，即使不考虑成本的因素，系统硬件环境有限的空间和有限的板面积决定了可配置的内存容量是很有限的。

针对以上三个约束条件，一般的嵌入式系统中最基本的内存管理方案有两种—静态分配和动态分配[42,43]。

在强实时系统中，为减少内存分配在时间上可能带来的不确定性，可采用静态分配的内存管理方式。静态分配是指在编译或链接时将程序所需的内存空间分配好。采用这种分配方案的程序段，其大小一般在编译时就能够确定，对于这种方式，不需要操作系统进行专门的内存管理操作，但系统使用内存的效率比较低下，只适合于那些强实时、应用比较简单和任务数量可以静态确定的系统。

动态内存分配管理机制为堆，应用通过分配与释放操作来使用内存。在使用一段时间后，堆会带来碎片的问题，内存被逐渐划分为位于已被使用区域之间的越来越小的区域。对此，有的操作系统提供了垃圾回收机制，对内存堆进行重新排列，把碎片组织成为大的连续可用的内存空间。但该方法可能在一个随机的时间使任务停止运行，且垃圾回收的时间长短也不确定，使得该方法不适合于处理实时应用。因此，在实时系统中应提供灵活的内存分配机制，避免内存碎片的出现，而不是在出现内存碎片时再回收。

固定大小存储区管理和可变大小存储区管理为动态内存的常用管理方法。固定大小存储区和可变大小存储区都是指定边界的一块地址连续的内存空间，其中固定大小存储区管理实现固定大小内存块的分配，可变大小存储区管理实现可变大小内存块的分配[44,45]。

本系统采用了固定大小存储区管理和可变大小存储区管理两种动态内存分配方法，可满足需要。

3.7.2 固定大小存储区管理的设计与实现

固定大小的存储区管理中，可供使用的一段连续的内存空间被称为是一个内存池，内存池中大小固定的存储区称作分区，内存池的位置、内存池的大小、内存池中每个分区的大小都是由应用决定的，存储区管理将实现内存池中分区的分配与回收。

(1) 数据结构

分区内存池的管理是通过分区内存池控制块来实现的，其定义如下：

```
typedef  struct  AP_PAR_MEM_POOL{
    UINT64                          *par_mem_pool_add_start;
    UINT32                          par_mem_pool_size;
    UINT32                          par_mem_size;
    UINT32                          par_mem_available;
        UINT32                          par_mem_allocated;
        struct PAR_MEM_BLK              *par_mem_ available_list;
        UINT32                          sup_task_num;
        struct  PAR_MEM_SUP_TASK        *sup_task_list;
        AP_SUSPEND_TYPE                 fifo_pri;
} AP_PAR_MEM_POOL;
```

par_mem_pool_add_start：内存池的首地址；

par_mem_pool_size：内存池的大小；

par_mem_size：分区的大小；

par_mem_available：分区内存池中未分配的分区的数量；

par_mem_allocated：分区内存池中已分配的分区的数量；

par_mem_ available_list：内存池中未分配的分区链表，具体结构将在后面介绍；

sup_task_num：因为内存池中没有空闲分区而挂起的任务数量；

sup_task_list:因为内存池中没有空闲分区而挂起的任务链表,具体结构将后面介绍；

fifo_pri：用于表示当任务需要被挂起时，是按“先进先出”的顺序，还是按“优先级”的顺序挂起在任务链表中。

为了便于内存池中分区的管理，将所有的空闲分区用链表链接起来，每个分区用分区控制块表示，其结构定义如下：

```
typedef struct PAR_HEADER_STRUCT{
        StructPAR_HEADER_STRUCT  *pm_next_available;
        AP_PAR_MEM_POOL                  *par_partition_pool;
} PAR_HEADER;
```

pm_next_available：指向链表中下一个未分配的分区；

par_partition_pool：指向分区所属的内存池。

如果内存池中的分区全部分配出去，此时又有新的任务申请分配内存，且任务允许被挂起，则内存池管理模块将任务挂起，并将所有因此挂起的任务链接成为一个链表，当有空闲分区时，则按一定的顺序将任务唤醒。链表中任务结点的结构定义如下：

```
typedef  struct  PAR_MEM_SUP_TASK{
        AP_TASK                         *sup_task;
        AP_PAR_MEM_POOL                 *par_mem_pool;
        UINT32                          is_allocate_success;
        void                            *add_allocate;
```

```
        struct  PAR_MEM_SUP_TASK      *sup_pre;
        struct  PAR_MEM_SUP_TASK      *sup_nxt;
} PAR_MEM_SUP_TASK;
```

sup_task：指向被挂起的任务控制块；

par_mem_pool：指向申请内存分区的内存池；

is_allocate_success：表示申请内存分区分配是否成功；

add_allocate：指向分配的内存分区的首地址，当申请失败时，值为空；

sup_pre：指向任务链表的前一个结点；

sup_nxt：指向任务链表的后一个结点。

(2) 创建内存池

创建内存池，根据内存池和分区的大小，将内存池组织成一个由空闲的分区组成的单向链表，并初始化内存池控制块的各个参数，其中，内存池的首地址，内存池的大小，分区的大小均由用户输入。内存池的结构如图 3.16 所示。

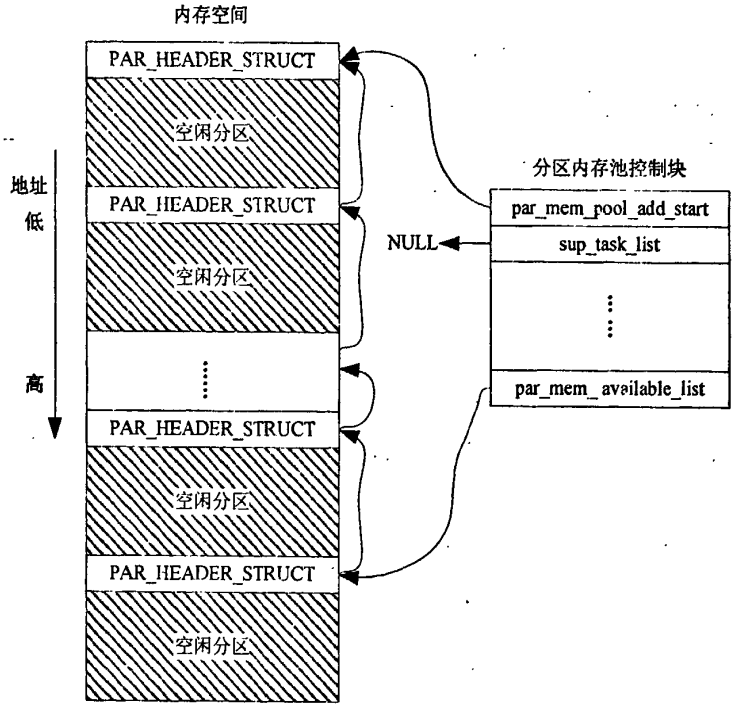

图 3.16 内存池结构图

Fig. 3.16 The architecture of memory pool

(3) 分配分区

当需要使用内存池中的分区时，从空闲分区链表中，按照空闲分区链表的顺序进行分区的分配，当没有可分配的空闲分区时，且调用者允许挂起，则将任务控制块插入 sup_task_lis 指向的任务链表中。具体的程序流程如图 3.7 所示。

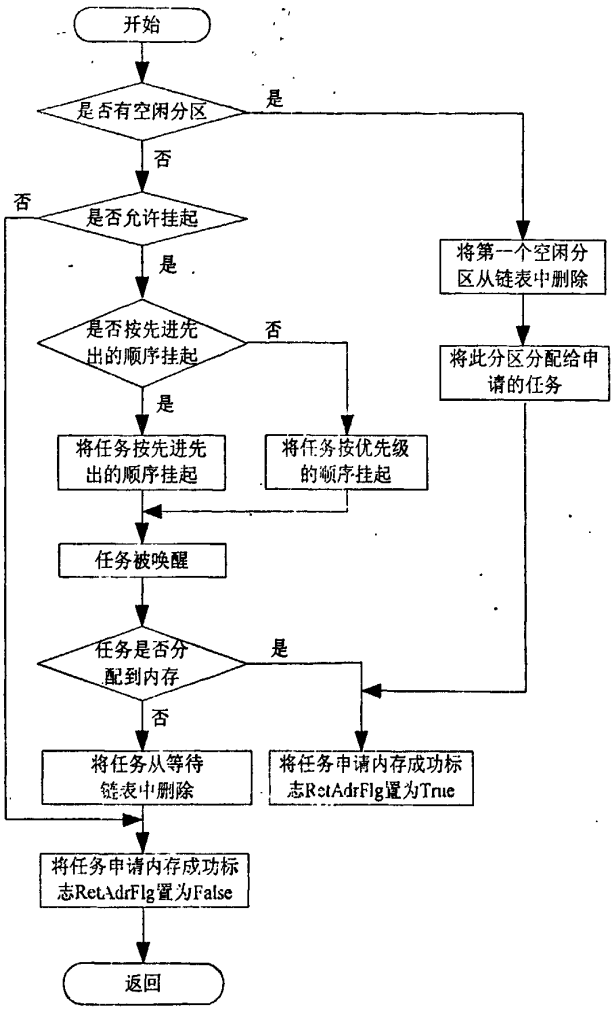

图 3.17 分配分区流程图

Fig. 3.17 The flowchart of distributing subarea

(4) 回收分区

回收分区时，先要查看是否有任务在等待分配分区。如果有，则直接将分区分配给他；如果没有，则将分区插入空闲分区链表，具体的程序流程如图 3.18 所示。

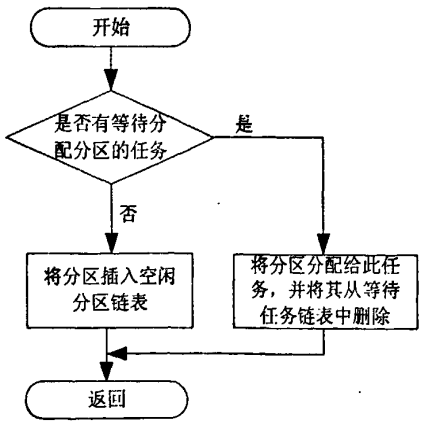

图 3.18 回收分区流程图

Fig. 3.18 The flowchart of reclaiming subarea

3.7.3 可变大小存储区管理的设计与实现

可变大小存储区管理也为基于内存池的管理方式。内存池为一段连续的、大小可配置的内存空间，用来提供可变内存块的分配，可变内存块称为分区。这里的内存池与分区的概念与固定大小存储区管理中的概念不同，这里的分区大小是可变的，而不是大小固定的，它们在内存池中的组织结构也不一样。内存池容量为用户指定的字节数，内存池的位置也根据具体应用来决定。可变大小存储区的管理提供可变长度内存的分配和重分配的服务[4S]。

(1) 数据结构

可变大小存储区的管理是通过内存池控制块来实现的，内存池控制块的结构如下：

```
typedef  struct  AP_DYN_MEM_POOL{
        UINT64                          *dyn_mem_pool_add_start;
        UINT32                          dyn_mem_pool_size;
        UINT32                          dyn_mem_pool_free_size;
        structDYN_MEM_BLK          *dyn_mem_blk_list;
        structDYN_MEM_BLK          *dyn_mem_blk_search;
        UINT32                          sup_task_num;
        struct  DYN_MEM_SUP_TASK        *sup_task_list;
        AP_SUSPEND_TYPE                 fifo_pri;
}AP_DYN_MEM_POOL;
```

dyn_mem_pool_add_start：内存池的起始地址；

dyn_mem_pool_size：内存池的大小；

dyn_mem_pool_free_size：内存池的空闲空间的大小；

dyn_mem_blk_list：指向内存池中的第一个分区，具体结构将在后面介绍；

dyn_mem_blk_search：用于查找内存池中的空闲分区，具体结构将在后面介绍；

sup_task_num：因内存池的空间不足而挂起的任务的数目；

sup_task_list：指向挂起的任务链表，具体结构将在后面介绍；

fifo_pri：用于表示当任务需要被挂起时，是按“先进先出”的顺序，还是按“优先级”的顺序挂起在任务链表中。

为了便于内存池的管理，将所有的分区用双向循环链表链接起来，每个分区用分区控制块表示，其结构定义如下：

```
typedef  struct  DYN_MEM_BLK{
        struct      DYN_MEM_BLK     *dyn_mem_blk_pre;
        struct      DYN_MEM_BLK         *dyn_mem_blk_nxt;
        AP_DYN_MEM_POOL                 *dyn_mem_pool_point;
        AP_BOOL                     free_flag;
}DYN_MEM_BLK;
```

dyn_mem_blk_pre：指向前一个分区控制块；

dyn_mem_blk_nxt：指向后一个分区控制块；

dyn_mem_pool_point：指向分区所属的内存池；

free_flag：表明此分区是否是空闲分区。

如果内存池中的分区全部分配出去，此时又有新的任务申请分配分区，且任务允许

被挂起，则系统将任务挂起，并将所有因此挂起的任务链接成为一个链表。当有空闲分区时，则按一定的顺序将任务唤醒。链表中任务结点的结构定义如下：

```
typedef  struct  DYN_MEM_SUP_TASK{
         AP_TASK                          *sup_task;
         AP_DYN_MEM_POOL                  *dyn_mem_pool_point;
         UINT32                           request_size;
         UINT32                           is_allocate_success;
         void                             *add_allocate;
         struct   DYN_MEM_SUP_TASK        *sup_pre;
         struct   DYN_MEM_SUP_TASK        *sup_nxt;
}DYN_MEM_SUP_TASK;
```

sup_task：指向挂起的任务；

dyn_mem_pool_point：指向申请内存空间的内存池；

request_size：申请空间的大小；

is_allocate_success：表明分区是否分配成功；

add_allocate：分配的分区的首地址；

sup_pre：指向前一个挂起的任务；

sup_nxt：指向后一个挂起的任务。

(2) 创建内存池

可变大小存储区中的分区通过双向链表链接起来，形成一个分区链。在创建内存池时，所有的空间均未分配，理论上应建立一个空闲分区，但为了便于双向循环链表的操作，我们将建立一个空闲分区和一个已分配分区，分配分区大小为 0，位于内存池的末尾处，仅仅用于将分区连接成为双向循环链表；空闲分区的大小为整个存储区的大小减去控制块的内存开销，位于内存池的起始处。整个内存池的内存空间结构如图 3.19 所示。

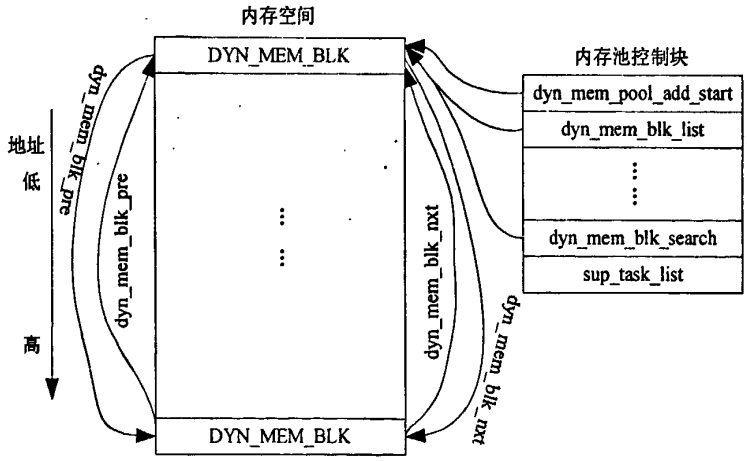

图 3.19 内存池结构图

Fig. 3.19 The architecture of memory pool

(3) 分配分区

分配分区即按分区在链表中的次序，依序遍历内存池中的分区，如果存在一个分区，它是未分配的且它的空间大于或等于所申请的内存大小，则将此分区分配给它。当分区还有剩余空间且

剩余空间足够容下一个分区控制块时，则将剩余空间重新建立一个空闲分区；当剩余空间小于分区控制块的大小时，则将整个空间分配给它，这样做是为了避免碎片的产生。如果所有的分区均不满足条件，且任务允许被挂起，则将任务挂起。具体的程序流程如图 3.20 所示。

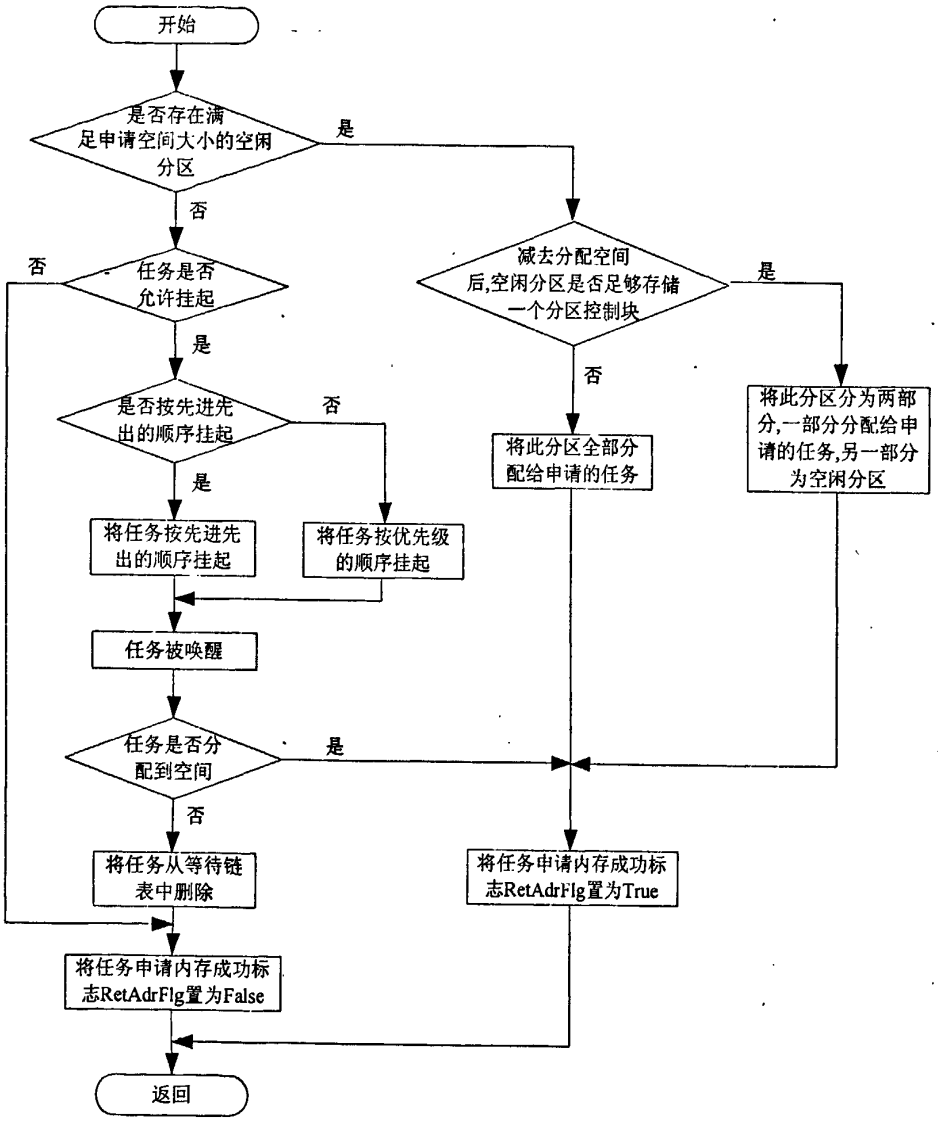

图 3.20 分配分区流程图

Fig. 3.20 The flowchart of distributing subarea

(4) 回收分区

当把分区释放回内存池中时，在双向链表中，如果它相邻的分区为空闲分区时，则将它们合并成为一个更大的分区，这样可以减少碎片的产生。具体的程序流程如图 3.21 所示。

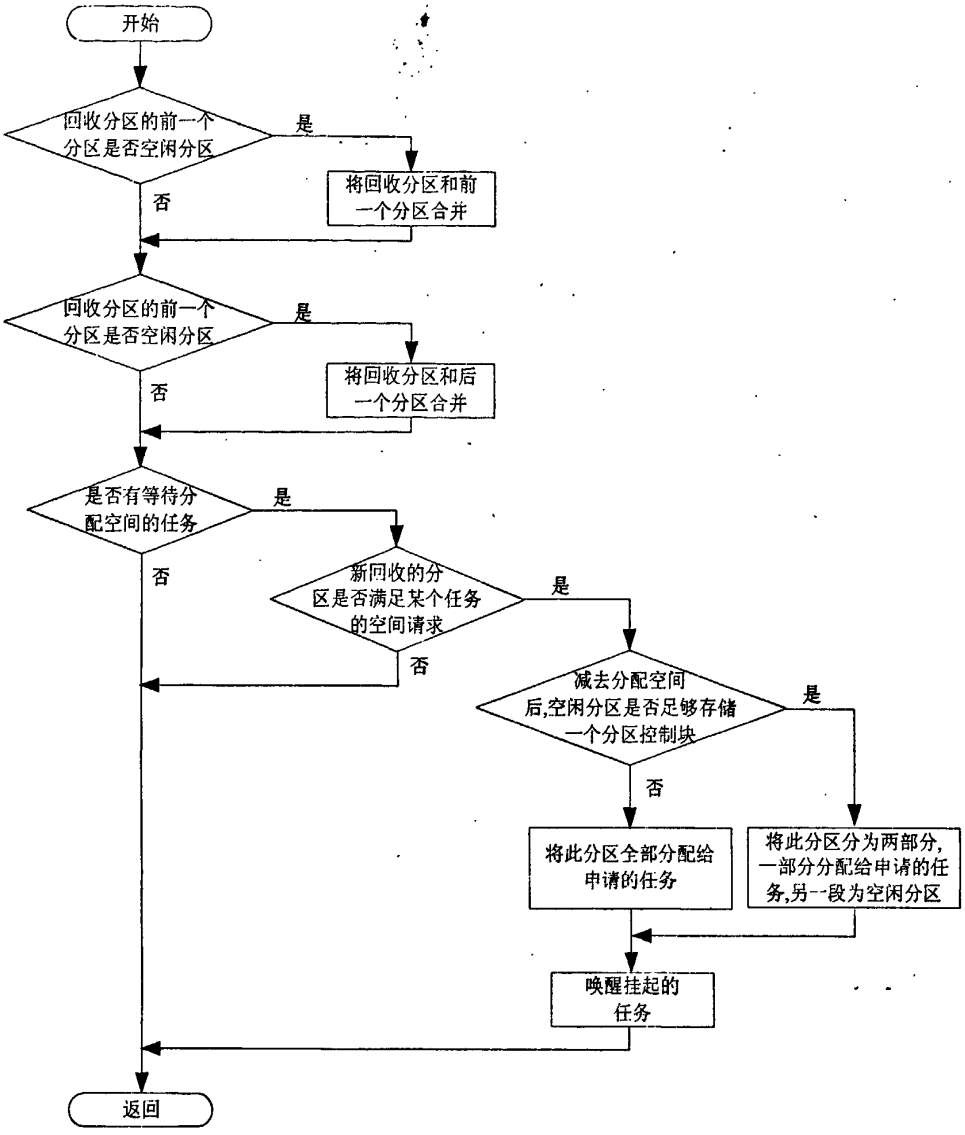

图 3.21 回收分区流程图

Fig. 3.21 The flowchart of reclaiming subarea

3.8 初始化模块

在操作系统启动运行之前，要做一些初始化工作，这包括两部分：硬件的初始化与软件的初始化。

硬件的初始化包括时钟的设置，存储空间的设置，处理器模式的设置等，这些都是与底层的硬件紧密相关的，在此不多做介绍，下面着重介绍软件的初始化。

从处理器看来，操作系统也是一个运行的程序，它也有运行的上下文环境，有自己的变量等，软件的初始化包括以下几部分：

(1) 系统空间的初始化

操作系统的运行也需要占用空间，称其为系统空间，所有的中断函数在运行时都没有自己的堆栈空间，它们占用的是系统空间，为此专门分配一段空间用作系统空间，考虑到中断函数不能太大的缘故(否则影响系统的实时性)，为系统分配的空间大小为 1024

bytes。

(2) 系统变量的初始化

系统应用的全局变量均需要初始化，包括当前任务指针 g_task_cur，挂起的任务链表的指针 g_task_sup_list，系统时钟 g_time 及系统初始化标志 g_os_init 等。

(3) 建立空闲任务

此系统没有电源管理模块，当系统长时间没有要处理的任务时，系统并不能自动进行省电模式，停止 CPU 指令的运行，为此为系统建立了一个空闲任务，它的优先级最低，故当没有其它任务运行时，则系统调度空闲任务运行，在空闲任务中什么都不做，只是个无限循环的空指令而已，等待有更高级的任务的运行。

(4) 建立 tick 中断的延迟调用函数 HISR

在每个 tick 中断中都要检查是否有时间片函数(周期性调度的函数)，是否有更高优先级的任务需要调度，是否有挂起一段时间的任务需要被唤醒等。如果这些检查都放在 tick 中断中处理，必定会延长中断时间函数的运行，影响系统的实时性，为此专门建立了 tick 中断的延迟函数 tick_hisr_fun，用来处理这些检查。

第四章 AcoolOS 应用层 API 接口函数

操作系统内核最终是以C库的形式提供给用户，用户可以选择所需要的部分链入到应用程序当中。AcoolOS提供给用户的主要应用接口函数如下。

4.1 系统初始化函数

(1) void AP_os_init(void)

操作系统的初始化函数，用于初始化操作系统中的一些系统变量，包括中断函数列表，当前任务指针等。

(2) void AP_os_start(void)

启动操作系统即打开 tick 中断，开始任务的调度。在此这前，操作系统是不进行任务调度的。

4.2 任务管理函数

(1) AP_status AP_task_create(AP_TASK *task_ptr, void (*task_fun)(void), void * stk_add, UINT32 stk_size, UINT8 task_pri, UINT32 timer_slice, AP_BOOL Start, AP_BOOL preempt)

建立新的任务。

参数：

task_ptr：指向新建立的任务控制块；

task_fun：指向新建立的任务的函数；

stk_add：指向为任务分配的内存空间的起始地址；

stk_size：为任务分配的空间的大小(以字(4 字节)为单位)；

task_pri：任务的优先级(0~254)；

timer_slice：任务的时间片，表示任务被周期性调度的时钟周期，如对相同优先级的任务不进行周期性调度，则此值为 0；

Start：布尔值。为 AP_TRUE 时，表示新建立的任务处于就绪状态，可以被调度；为 AP_FALSE 时，表示新建立的任务处于挂起状态，需经“唤醒”后，方可被调度；

preempt：布尔值。为 AP_TRUE 时，表示任务具有可被抢占性；为 AP_FALSE 时，表示任务不具有可被抢占性；当其为 AP_FALSE 时，timer_slice 必须为 0。

返回值：

RE_OK：表示任务创建成功；

RE_PARA_ERR：表示参数有错误。

(2) AP_status Ap_task_suspend(AP_TASK *task)

将任务挂起。

参数：

task：指向被挂起的任务控制块。

返回值：

RE_OK：表示任务被成功挂起；

RE_PARA_ERR：表示参数有错误。

(3) AP_sleep(UINT32 tick)

将任务休眠 tick 个系统时钟。

(4) AP_status AP_task_resume(AP_TASK *task)

将挂起的任务“唤醒”。

参数：

task：指向将被唤醒的任务控制块。

返回值：

RE_OK：表示任务被成功唤醒；

RE_PARA_ERR：表示参数有错误；

RE_FAIL：表示“唤醒”任务失败。

(5) AP_status AP_task_chage_timer_slice(AP_TASK *task, UINT32 timer_slice)

改变任务的周期性调度的时间片。

参数：

task：指向被操作的任务控制块；

timer_slice：重新设置的任务周期性调度的时长。

返回值：

RE_OK：表示操作成功；

RE_PARA_ERR：表示参数有错误；

RE_FAIL：表示操作失败。

4.3 中断管理函数

4.3.1 HISR 函数

(1) void AP_hisr_create(AP_HISR *hisr_ptr, void(*hisr_fun)(void), void * stk_add, UINT32 stk_size, UINT8 hisr_pri);

建立新的 HISR。

参数：

hisr_ptr：指向 HISR 的控制块；

hisr_fun：指向 HISR 的函数；

stk_add：指向为 HISR 分配的任务空间的起始地址；

stk_size：为 HISR 分配的任务空间的大小；

hisr_pri：HISR 的任务优先级(0、1、2 三个优先级)。

(2) void AP_ hisr_activate (AP_HISR *Hisr)

激活 HISR。

参数：

Hisr：指向操作的 HISR 的控制块。

4.3.2 LISR 函数

(1) void AP_lisr_register(UINT32 Vector, void (*lisr_fun)(void))

注册 LISR 中断函数。

参数：

Vector：指向要注册的中断向量；

lisr_fun：指向 LISR 调用的中断函数。

4.4 时钟函数

(1) void AP_timer_create(AP_TIMER *Timer, void (*timer_fun)(void), UINT32 timer_init, UINT32 timer_loop, AP_EN_DISABLE is_enable)

用于建立时钟函数的控制块，此控制块用于周期性调度的函数。

参数：

Timer：时钟函数控制块；

timer_fun：周期性调度的函数；

timer_init：时钟初始化值，即函数第一次被调用的时间；

timer_loop：函数被调度的周期；

is_enable：布尔值。用于判断是否立即启动此函数的周期性调度。当为 AP_ENABLE 时，表示立即启动此时钟；当为 AP_DISABLE 时，表示不是立即启动，要以后通过调用 AP_timer_control 来启动时钟。

(2) void AP_timer_control(AP_TIMER *Timer, AP_EN_DISABLE is_enable)

用于时钟控制块时钟的开启与关闭，即是否进行函数的周期性调度。

参数：

Timer：用于指向时钟函数控制块；

IsEnable：布尔值，用于启动/停止时钟。

(3) void AP_timer_loop_set(AP_TIMER *Timer, UINT32 timer_loop)

用于重新设置时钟控制块的时钟周期。

参数:

Timer: 指向将被操作的时钟控制块;

timer_loop: 重新设置的时钟周期。

4.5 消息队列函数

(1) AP_status AP_queue_create(AP_QUEUE * Queue, void *que_add, UINT32 que_size, UINT32 mesg_size, AP_SUSPEND_TYPE fifo_pri)

用于建立消息队列。

参数:

Queue: 用于指向将要操作的任务队列的控制块;

que_add: 用于指向消息队列的起始地址;

que_size: 用于表示消息队列的长度(以字(4 字节)为单位);

mesg_size: 用于表示消息队列中每个消息的长度(以字(4 字节)为单位);

fifo_pri: 用于指示消息在队列中是按“先进先出”或“任务优先级”排列。

返回值:

RE_OK: 表示成功创建消息队列;

RE_PARA_ERR: 表示参数有错误;

RE_FAIL: 表示创建消息队列失败。

(2) AP_status AP_queue_send(AP_QUEUE * Queue, void *mesg_area, UINT32 Suspend)

用于发送消息。

参数:

Queue: 用于指向将要操作的任务队列控制块;

mesg_area: 用于指向将被发送的消息的地址空间;

Suspend: 用于指示发送消息的任务是否可以被挂起(当消息队列已满时)或挂起的最大时间(0: 不允许被挂起; 0XFFFFFFFF: 无时间限制被挂起, 1~0XFFFFFFFE 被挂起的最大时间)

返回值:

RE_OK: 表示消息发送成功;

RE_PARA_ERR: 表示参数有错误;

RE_FAIL: 表示消息发送失败。

(3) AP_status AP_queue_receive(AP_QUEUE *Queue, void *mesg_area, UINT32 Suspend)

用于接收消息。

参数:

Queue: 用于指向将要操作的任务队列的控制块;

MesgArea：用于存放接收到的消息；

Suspend：用于指示接收消息的任务是否可以被挂起(当消息队列为空时)或挂起的最大时间。

返回值：

RE_OK：表示消息接收成功；

RE_PARA_ERR：表示参数有错误；

RE_FAIL：表示消息接收失败。

4.6 信号量函数

4.6.1 互斥信号量函数

(1) void AP_pro_create(CRITICAL_PROTECT * protect，AP_SUSPEND_TYPE fifo_pri)

用于建立互斥信号量。

参数：

protect：用于指向将要操作的互斥信号量控制块；

fifo_pri：用于指示当互斥信号量被占用时，等待申请信号量的任务是按“先进先出”或“任务优先级”的顺序挂起。

(2) AP_status AP_pro_obtain(CRITICAL_PROTECT * protect，UINT32 Suspend)

用于申请互斥信号量。

参数：

protect：用于指向将要操作的互斥信号量控制块；

Suspend：用于指示等待获取信号量的任务是否可以被挂起(当互斥信号量已被占用时)或挂起的最大时间。

返回值：

RE_OK：表示申请信号量成功；

RE_PARA_ERR：表示参数有错误；

RE_FAIL：表示申请信号量失败。

(3) void AP_pro_release(CRITICAL_PROTECT * protect)

用于释放互斥信号量。

参数：

protect：指向将被操作的互斥信号量。

4.6.2 计数信号量函数

(1) void AP_sem_create(AP_SEMAPHORE *Semaphore, UINT32 init_count, AP_SUSPEND_TYPE fifo_pri)

用于建立计数信号量。

参数：

Semaphore：用于指向将要操作的计数信号量控制块；

init_count：指明计数信号量的初始值；

fifo_pri：用于指示当没有空闲资源时，等待分配资源的任务是按“先进先出”或“任务优先级”挂起。

(2) AP_status AP_sem_obtain(AP_SEMAPHORE *Semaphore,UINT32 Suspend)

用于获取信号量。

参数：

Semaphore：用于指向将要操作的计数信号量控制块；

Suspend：用于指示等待获取信号量的任务是否可以被挂起(当计数信号量为 0 时)或挂起的最大时间。

返回值：

RE_OK：表示成功申请信号量；

RE_PARA_ERR：表示参数有错误；

RE_FAIL：表示申请信号量失败。

(3) void AP_sem_release(AP_SEMAPHORE *Semaphore)

用于释放任务所占用的信号量。

参数：

Semaphore：指向将被操作的计数信号量控制块。

4.7 内存管理函数

4.7.1 固定大小内存管理函数

(1) AP_status AP_par_mem_pool_create(AP_PAR_MEM_POOL
*par_mem_pool_con_blk, void * pool_add, UINT32 pool_size,
AP_SUSPEND_TYPE fifo_pri)

建立固定大小内存分配的内存池。

参数：

par_mem_pool_con_blk：指向建立的内存池控制块；

pool_add：内存池的首地址；

pool_size：内存池的大小，以字(四字节)为单位；

fifo_pri：先进先出标志，用于指示当内存空间不足时，等待分配内存的任务是按“先进先出”还是按“优先级”的顺序挂起。

返回值：

RE_OK：表示成功建立内存池；

RE_PARA_ERR：表示参数有错误；

RE_FAIL：表示建立内存池失败。

(2) AP_status AP_par_mem_allocate(AP_PAR_MEM_POOL par_mem_pool_con_blk,
void ** return_add, UINT32 size, UINT32 suspend)

从内存分配池中动态分配内存。

参数：

par_mem_pool_con_blk：指向申请分配内存的内存池控制块；

return_add：分配的内存的首地址(四字节对齐)；

size：申请分配的内存的大小，以字(四字节)为单位；

suspend：表示是否允许任务被挂起(当内存池的空间不足时)。当为 AP_UNSUSPEND 时，表示不允许挂起；当为 AP_SUSPEND 时，表示允许挂起，直到内存池有足够的空间；当为 1~0xFFFFFFFE 时，表示允许挂起的最长时间，当时间超出时，即使内存池没有足够的空间，任务也将被唤醒。

返回值：

RE_OK：表示成功分配内存；

RE_PARA_ERR：表示参数有错误；

RE_FAIL：表示内存分配失败。

(3) AP_status AP_par_mem_deallocate(void * memory)

释放动态分配的内存。

参数：

memory：将要释放的内存的首地址。

返回值：

RE_OK：表示成功释放内存；

RE_PARA_ERR：表示参数有错误；

RE_FAIL：表示内存释放失败。

4.7.2 可变大小内存管理函数

(1) AP_status AP_dyn_mem_pool_create(AP_DYN_MEM_POOL *dyn_mem_pool_con_blk, void * pool_add, UINT32 pool_size, AP_SUSPEND_TYPE fifo_pri)

建立可变大小分配的内存池。

参数：

dyn_mem_pool_con_blk：指向建立的内存池控制块；

pool_add：内存池的首地址；

pool_size：内存池的大小，以字(四字节)为单位；

fifo_pri：先进先出标志，用于指示当内存空间不足时，等待分配内存的任务是按“先进先出”还是按“优先级”的顺序挂起。

返回值：

RE_OK：表示成功建立内存池；

RE_PARA_ERR：表示参数有错误；

RE_FAIL：表示建立内存池失败。

(2) AP_status AP_dyn_mem_allocate(AP_DYN_MEM_POOL dyn_mem_pool_con_blk, void ** return_add, UINT32 size, UINT32 suspend)

从内存池中动态分配内存。

参数：

dyn_mem_pool_con_blk：指向申请分配内存的内存池控制块。

return_add：分配的内存的首地址(四字节对齐)；

size： 申请分配的内存的大小，以字(四字节)为单位；

suspend：表示是否允许任务被挂起(当内存池的空间不足时)。当为 AP_UNSUSPEND 时，表示不允许挂起；当为 AP_SUSPEND 时，表示允许挂起，直到内存池有足够的空间；当为 1~0xFFFFFFFE 时，表示允许挂起的最长时间，当时间超出时，即使内存池没有足够的空间，任务也将被唤醒。

返回值：

RE_OK：表示成功分配内存；

RE_PARA_ERR：表示参数有错误；

RE_FAIL：表示内存分配失败。

(3) AP_status AP_dyn_mem_deallocate(void * memory)

释放动态分配的内存

参数：

memory：将要释放的内存的首地址；

返回值：

RE_OK：表示成功释放内存；

RE_PARA_ERR：表示参数有错误；

RE_FAIL：表示内存释放失败。

第五章 AcoolOS 的测试与性能分析

在操作系统的开发过程中，保证操作系统的正确性至关重要，为此在每个模块的开发过程中都进行了必要的单元测试，保证了模块的正确性，但操作系统整体运行即各个模块共同工作时的情况又如何呢？为此基于操作系统 AcoolOS 进行了项目开发，并分析了 AcoolOS 的性能与特点。

5.1 AcoolOS的应用测试

为了保证操作系统的开发效率，尽快将操作系统实用化，并保证操作系统得到较全面的验证，选择开发的项目应具有如下特点：

(1) 项目应使用到操作系统 AcoolOS 的所有功能模块，使操作系统得到全面的验证；

(2) 项目中程序流程尽量简单清晰，便于查找操作系统开发中残留的 Bug。

为此选择了 Locust 软件升级工程，它具有上述两特点。

5.1.1Locust 工程简介

Locust 软件升级工程用于软件代码的升级，即通过 USB 口与 PC 机相连，将 PC 机编译生成的目标代码写到 Nor Flash 中特定的地址中，实现代码的升级，其中用到操作系统的任务管理、中断管理、时间管理、通信、互斥锁、内存管理等所有的模块，并涉及到 USB 设备与 Nor Flash 驱动的知识。

(1) 硬件环境

使用开发操作系统的 ARM9 S3C2440A 作为目标处理器的应用系统，并对其 USB 设备控制器外围电路进行了连接，配备了一块 8M Nor Flash，具体的硬件清单如下：

- S3C2440A 开发板，并配有 USB 设备接口与 8M Nor Flash；
- USB 线一条；
- ARM 并口 Multi-ICE 仿真器一套；
- PC 机一台，使用 Windows XP 操作系统。

(2) 软件架构

软件由两部分组成：PC 端软件与设备端软件。PC 端软件由应用层与 USB 主机驱动层组成，负责与设备端通信。设备端软件由 USB 设备驱动层、Flash 驱动层和操作系统 AcoolOS 层组成，负责与 PC 机通信和 Flash 的读写。其中为了保证 PC 与设备端的正确通信，在通信两端又加了一层自定义的通信协议：USB-Bulk-Protocol，软件结构图如图 5.1 所示。

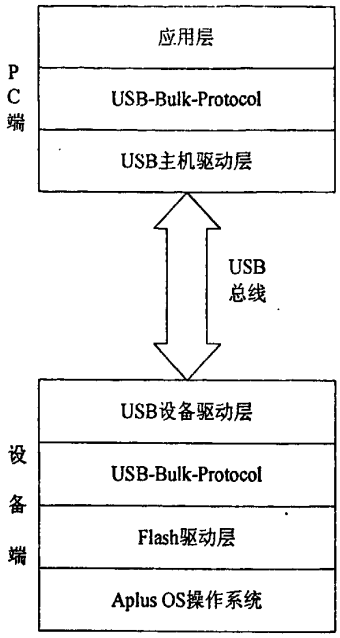

图 5.1 软件结构图

Fig. 5.1 Software's Architecture

5.1.2 测试

将设备端程序代码通过 Multi-ICE 写入 Flash 中，然后在 PC 端安装 USB 主机驱动程序，再通过 USB 数据线将 PC 机与设备相连。启动 PC 端应用程序(应用程序运行界面截图如图 5.2 所示)，然后选择将被写入的代码，依次单击“启动系统”、“软件升级”，此时如果连接正确，则数据代码将被写入设备端的 Flash 中。然后可以单击“升级校验”，来检验数据的正确性。此时如果“信息提示”显示：数据校验正确，说明代码被正确的写入设备，即操作系统 AcoolOS 运行正确；如果显示：数据校验错误，则说明代码没有被正确写入。在实验中，显示数据校验正确，即操作系统 AcoolOS 运行正确。

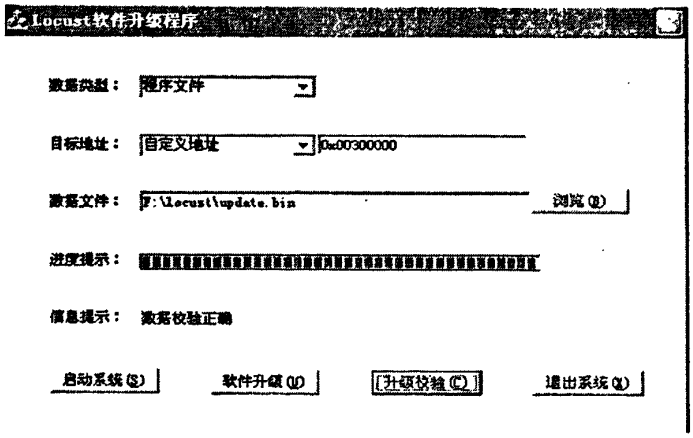

图 5.2 应用程序运行界面

Fig. 5.2 The application program running

5.2 性能分析

实时内核在实时系统中起着重要的作用，其性能的好坏将直接影响到整个系统的性

能。各种量化的性能指标对评价一个嵌入式实时内核提供了客观的依据，这些性能指标可以分为两大类：时间性能指标和存储开销[47,48]。

5.2.1 时间性能指标

嵌入式实时内核的时间性能指标主要包括：

中断延迟时间：指从中断发生到系统获知中断，并且开始执行中断服务程序所需要的最大滞后时间。

中断响应时间：指从中断发生到开始执行用户中断服务程序的第一条指令之间的时间。

中断恢复时间：指用户中断服务程序结束后回到被中断代码之间的时间，对于抢占式调度内核，则指到开始执行新任务代码之间的时间。

任务上下文切换时间：指 CPU 的控制权由运行任务转移到另一个就绪任务时所发生的时间，包括保存当前运行任务上下文的时间、选择下一个任务调度的时间及将要运行任务的上下文的恢复时间。

任务响应时间：指从任务对应的中断产生到该任务真正开始运行这一过程所花费的时间，任务响应时间叫调度延迟。

上述几项中，内核中断响应时间和任务上下文切换时间是评价内核实时性能最常用的且最重要的两个技术指标[49,50]，下面着重介绍这两项。

(1) 中断响应时间

在实时内核的各项时间性能指标中，有很多是与中断有关的，这也不奇怪，嵌入式实时系统很多都是中断驱动的系统，多任务实时应用也主要是由实时内核，多个应用任务和多个中断处理程序构成的相互协作的有机整体[51]。为了更好的理解这些性能指标，下面给出内核的中断时序图，如图 5.3 所示。

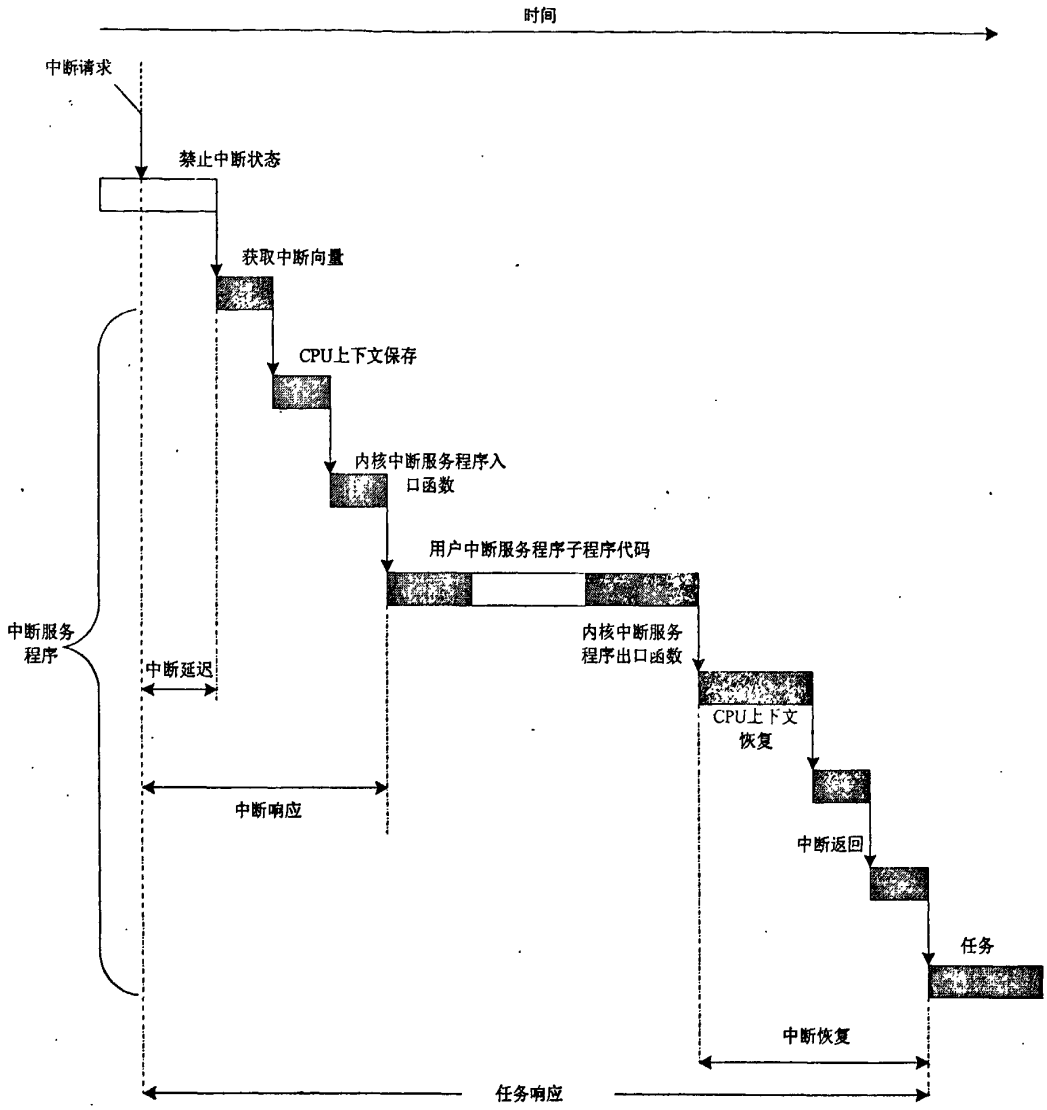

图 5.3 内核的中断时序图

Fig. 5.3 The timing diagram of core interrupt

由图不难看出，中断响应时间可由下面的表达式给出：

中断响应时间＝中断延迟＋获取中断向量的时间＋CPU 上下文保存的时间＋内核中断服务程序入口函数的执行时间

其中的中断延迟时间受到系统关中断时间的影响。实时系统在进入临界区代码段之前要关中断，执行完临界代码之后再开中断。另外本系统是不支持嵌套中断的，每次执行用户中断服务程序时都是要关中断的。关中断的时间越长，中断延迟就越长，并且可能引起中断丢失。中断延迟对中断响应时间的测量是非常重要的，但中断延迟的时间又是与用户的中断服务程序紧密相关的，不易测量，而其它时间都是确定的，查看它们的反汇编代码共有 54 条指令，CPU 的运行频率是 400M，则时间为 48×1/400M＝0.12μs，故

中断响应时间 ＝ 中断延迟 ＋ 0.12μs

在理想的情况下，中断延迟为 0，故中断响应的时间最短为 0.12μs，而 μC/OS-II 在工作

在 44.236MHz 的处理器 LPC2104 上运行时，测得的响应时间为 1.23μs[52]。

(2) 任务上下文切换时间

任务切换是在实时系统中频繁发生的动作，其运行的快慢将直接影响到整个实时系统的性能。由图 5.4 可以看出，任务上下文切换时间主要由三大部分组成，其中保存和恢复上下文的时间主要取决于任务上下文的定义和处理器的速度，不同种类的处理器，任务上下文的定义不同，其内容有多有少。

除开硬件的因素，任务上下文切换的时间与调度(即选择下一个运行任务)的过程有关，与具体实现调度算法时采用的数据结构有关。在本系统中，基于优先级的抢占式调度采用优先级位图的数据结构，保证了选择过程的时间的确定性，即时间不会随系统中就绪任务数目的多少而变化。

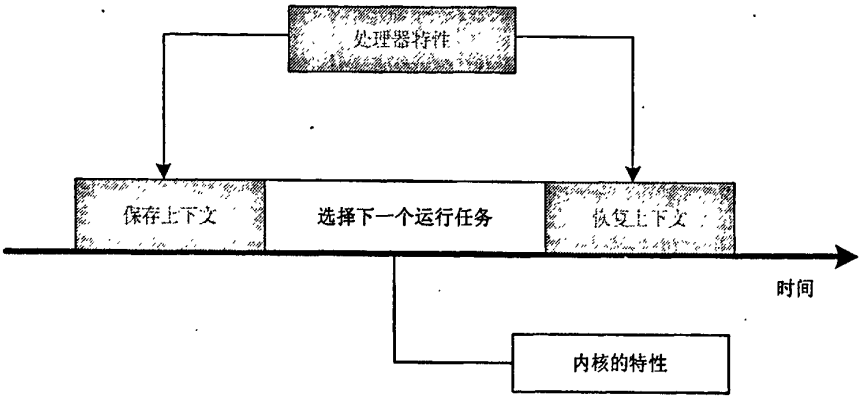

图 5.4 任务上下文切换时序图

Fig. 5.4 The timing diagram of task switching

由上所述，在一个确定的系统中，任务上下文切换的时间是确定的，和用户的应用程序无关，对其值的测量可以利用硬件计数器测量，方法如下：

建立两个任务 A 与 B，其中 A 的优先级高于 B。在任务 A 中，设定计数器的计数周期为 T，并初始化计数值为 0，此时计数器开始计数，然后任务 A 将自己挂起，此时将唤醒任务 B。在任务 B 中读取计数器的值，然后唤醒任务 A，这时任务 B 将被挂起，如此周而复始，程序如下：

```
Task A(higher priority)
while(1)
{
  Settimer();
  Suspend self();
}
TaskB(lower priority)
while(1)
{
    Gettimer();
    Resume TaskA();
}
```

经过多次测量，在 AcoolOS 中任务切换的平均时间为：2.43μs。

5.2.2 存储开销

在嵌入式应用中，系统存储空间的大小也是很重要的问题。即使目前内存以及非易失性存储器的价格在不断下降，但是对于批量很大的嵌入式设备，基于成本和功耗的考虑，其存储器的配置一般都不大。而在这有限的空间内不仅要装载嵌入式实时操作系统，还要装载用户程序。因此，在实时内核的设计和应用开发中，除了上述的时间性能指标，还应关注嵌入式实时内核的存储开销，这也是嵌入式实时操作系统与其它普通操作系统的明显区别之一[53]。

操作系统在没有建立任何任务的情况下，通过生成的映象文件可知，内核的大小是12.03KB，其内核数据结构占用了 0.96KB 的空间，所占用的空间较小，而对于 Nucleus Plus 操作系统，这两项数据分别是 20-40KB 和 2-4KB[54]。

第六章 结束语

实时操作系统 RTOS 正越来越得到计算机嵌入式应用人员的重视，应用也越来越广泛。这是因为 RTOS 可以更合理、更有效地利用 CPU 的资源，简化应用软件的设计，缩短系统开发时间，更好地保证系统的实时性和可靠性。因此，设计一种实用、可靠的 RTOS 内核成为本课题研究的内容。

作者根据国内外 RTOS 技术发展状况，借鉴 μC/OS-II，Nucleus Plus 等开放源码的 RTOS 的设计，并根据实际嵌入式系统开发的需要，开发实现了一套实时操作系统内核 AcoolOS。这套内核为用户提供了任务管理、中断管理、任务间通信管理、信号量管理、时间管理和内存管理等常用的系统服务，并在任务调度方面采用优先级位图算法，在中断方面采用 LISR 与 HISR 的机制，减少了任务的响应时间，提高实时性，可满足大部分的嵌入式实时开发的需要。

在开发过程中，我更加深入理解了 RTOS 工作的基本原理，对系统开发过程及方法有了更深刻的认知。在 AcoolOS 实现后，又对其进行了测试，证明其具有实时性高、体积小、内存分配灵活、易于裁剪等优点。

但是，由于时间和水平的限制，系统做得还不够完善。AcoolOS 的可移植性较差，目前仅提供对 ARM 处理器的支持；AcoolOS 没有提供信号处理的实现，另外 AcoolOS 仅仅是一个内核，做为一个完善的操作系统，还需要增加文件系统，网络通信协议等模块的实现。这些不足的地方需要在以后的开发中得到改善。

同时希望整个设计过程中所用到的原理和方法，对研究嵌入式实时系统的人员能够有所帮助。

参考文献

1. 王京起, 黄健, 沈中杰. 嵌入式可配置实时操作系统 eCos 技术及实现机制[M], 北京: 电子工业出版社, 2005, 1-2.
2. 唐寅. 实时操作系统应用开发指南[M], 北京: 中国电力出版社 2002, 13-21.
3. Warren Webb. Embedded Linux nears real time[J], Electrical Design News, 2004, 49(19): 55-56, 58, 60, 62.
4. Warren Webb. Embedded Linux nears real time[J], Electrical Design News, 2004, 49(19): 55-56, 58, 60, 62.
5. 李晶皎, 王爱侠, 张广渊. ColdFire 系列 32 位微处理器与嵌入式 Linux 应用[M], 北京: 北京航空航天大学出版社, 2005, 12-13.
6. John A. Stankovic, R. Rajkumar. Real-Time Operating Systems[J], Real-Time Systems, 2004, 28(2-3): 237-253.
7. Walter Cedeno, Phillip A. Laplante. An Overview of Real-time Operating Systems[J], Journal of the Association for Laboratory Automation. 2007, 12(1): 40-45.
8. Ali Fuat Alkaya, Haluk Rahmi Topcuoglu. A task scheduling algorithm for arbitrarily-connected processors with awareness of link contention[J], Cluster Computing, 2006, 9(4): 417-431.
9. V.Olive, S.Martin, A.Vareille. OS for embedded systems: state of the art and prospects[J], Microelectronic Engineerin, 2000, 54(1-2): 113-121.
10. Colin Holland. An embeddable RTOS: Michael Haunreiter and Uwe Baumgarten of Miray Software introduce the pnOS real-time operating system which is especially suitable for mobile embeddedsystems[J], Embedded Systems Europe, 2005, 9(67): 27-29.
11. 田泽. ARM7 μClinux 开发实验与实践[M], 北京: 北京航空航天大学出版社 2006, 1-3.
12. 张炯. 嵌入式实时操作系统的多线程计算一基于 ThreadX® 和 ARM®[M], 北京: 北京航空航天大学出版, 2005, 5-13.
13. Lui Sha, Tarek Abdelzaher, Karl-Erik årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, Aloysius K. Mok. Real Time Scheduling Theory: A Historical Perspective[J], Real-Time Systems, 2004, 28(2-3): 101-155.
14. Giorgio Buttazzo. Real-Time Operating Systems: Problems and Novel Solutions[J], Formal Techniques in Real-Time and Fault-Tolerant Systems: 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9-12, 2002. Proceedings, 2002, 2469: 37-51.
15. JEAN-PHILIPPE BABAU. Object Oriented Design for Real-Time Systems—Response to C E.Pereira's Contribution[J]. The International Journal of Time-Critical Computing Systems. 2000, 18 : 95-99 .
16. Yi Youngmin, Kim Dohyung, Ha Soonhoi. Fast and Time-Accurate Cosimulation with os Scheduler Modeling[J], Design Automation for Embedded Systems, 2003, 8(2-3): 211-228

17. 闫茂德, 贺昱曜, 陈金平, 许化龙. 嵌入式实时操作系统内核的设计与实现[J], 长安大学学报(自然科学版)2004, 24(3): 95-100.
18. 魏振华, 洪炳熔, 乔永强, 蔡则苏, 彭俊杰. 嵌入式实时操作系统 Nucleus 中线程控制部件的实现方法[J], 计算机应用研究, 2003, 20(04): 97-99.
19. 罗蕾. 嵌入式实时操作系统及应用开发[M], 北京: 北京航空航天大学出版社 2005, 107-114, 146-153, 184-210.
20. Yi-Chang Chiu, Hong Zheng. Real-time mobilization decisions for multi-priority emergency response resources and evacuation groups: Model formulation and solution[J], Transportation Research Part E: Logistics and Transportation Review, 2007, 43(6): 710-736.
21. Xibo Wang, Benhai Zhou, Ge Yu, Qian Li. Homology priority task scheduling in μC/OS-II real-time kernel[J], Wuhan University Journal of Natural Sciences, 2007, 12(5): 946-950.
22. Sanjoy K. Baruah. Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks[J], Real-Time Systems, 2003, 24(1): 93-128.
23. Sanjoy K. Baruah, Nathan Wayne Fisher. The partitioned dynamic-priority scheduling of sporadic task systems[J], Real-Time Systems, 2007, 36(3): 199-226.
24. Mikhail Y. Kovalyov, C.T. Ng, T.C. Edwin Cheng. Fixed interval scheduling: Models, applications, computational complexity and algorithms[J], European Journal of Operational Research, 2007, 178(2): 331-342.
25. V. A. Kostenko. The Problem of Schedule Construction in the Joint Design of Hardware and Software[J], Programming and Computer Software, 2002, 28(3): 162-173.
26. Weirong Wang, Aloysius K. Mok, Gerhard Fohler. Pre-Scheduling[J], Real-Time Systems, 2005,30(1-2): 83-103.
27. Zvika Brakerski, Aviv Nisgav, Boaz Patt-Shamir. General Perfectly Periodic Scheduling[J], Algorithmica, 2006, 45(2): 183-208.
28. Marcelo Götz, Achim Rettberg, Carlos Eduardo Pereira, A Run-Time Partitioning Algorithm for RTOS on Reconfigurable Hardware[J], Embedded and Ubiquitous Computing 2005, 3824: 469-478.
29. 杨立身, 王中海. 嵌入式实时操作系统任务调度算法改进[J], 微型电脑应用, 2007, 23(9): 44-46.
30. 张腾. 嵌入式实时操作系统内核的研究与实现(D), 北京邮电大学, 2001.
31. Ching-Hsien Hsu, Shih-Chang Chen, Chao-Yang Lan. Scheduling contention-free irregular redistributions in parallelizing compilers[J], The Journal of Supercomputing, 2007, 40(3): 229-247.
32. Zhengyu Ou, Ge Yu, Yaxin Yu, Shanshan Wu, Xiaochun Yang, Qingxu Deng. Tick Scheduling: A Deadline Based Optimal Task Scheduling Approach for Real-Time Data Stream Systems[J], Advances in Web-Age Information Management, 2005, 3739: 725-730.
33. Qiming Teng, Xiangqun Chen, Xia Zhao. On Generalizing Interrupt Handling into a Flexible

Binding Model for Kernel Components[J], Embedded Software and Systems, 2005, 3605: 423-429.

34. K. H. (Kane) Kim. A Non-Blocking Buffer Mechanism for Real-Time Event Message Communication[J], Real-Time Systems, 2006, 32(3): 197-211.
35. K. H. (Kane) Kim. A Non-Blocking Buffer Mechanism for Real-Time Event Message Communication, Real-Time Systems, 2006, 32(3): 197-211.
36. Arthur Segard, Francois Verdier. SOC and RTOS: Managing IPs and Tasks Communications[J], Field Programmable Logic and Application, 2004, 3203: 710-718.
37. Dong, Wei. The design and implementation of a real-time embedded system kernel and signal module for the MC68HC12 micro-controller[D], Dalhousie University (Canada), 2004.
38. Philip N. Klein, Robert H. B. Netzer, Hsueh-I Lu. Detecting Race Conditions in Parallel Programs that Use Semaphores[J], Algorithmica, 2007, 35(4): 321-345.
39. David Scholefield. Proving properties of real-time semaphores[J], Science of Computer Programmin, 1995, 24(2): 159-181.
40. Sivarama P. Dandamudi. Guide to Assembly Language Programming in Linux[M], US: Springer, 2005, 403-421.
41. Thomas E. Hart, Paul E. McKenney, Angela Demke Brown and Jonathan Walpole. Performance of memory reclamation for lockless synchronization[J], Journal of Parallel and Distributed Computing, 2007, 67(12): 1270-1285.
42. A.Crespo, I.Ripoll, M.Masmano. Dynamic Memory Management for Embedded Real-Time Systems[J], From Model-Driven Design to Resource Management for Distributed Embedded Systems, 2006, 225: 195-204.
43. Shlomi Dolev, Reuven Yagel. Memory Management for Self-stabilizing Operating Systems[J], Self-Stabilizing Systems, 2005, 3764: 113-127.
44. Hiser, Jason D. Effective algorithms for partitioned memory hierarchies in embedded systems[D], University of Virginia, 2005.
45. Heine, David L. Static memory leak detection[D], Stanford University, 2005.
46. Jaeheung Yeo, Heon Y. Yeom, Taesoon Park. An Asynchronous Protocol for Release Consistent Distributed Shared Memory Systems[J], The Journal of Supercomputing, 2003, 24(1): 25-41.
47. Changjun Wang, Yugeng Xi. Performance analysis of active schedules in identical parallel machine[J], Journal of Control Theory and Applications, 2007, 5(3): 239-243.
48. Andrew J. Kornecki, Janusz Zalewski. Experimental evaluation of software development tools for safety-critical real-time systems[J], Innovations in Systems and Software Engineering, 2005, 1(2):176-188.
49. Friedhelm Stappert, Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs[J], Journal of Systems Architecture, 2000, 46(4): 339-355.
50. Kemal Koker. Embedded RTOS: Performance Analysis With High Precision Counters[J], Autonomous Robots and Agents, 2007, 76:171-179.

51. Lucia Lo Bello and Kanghee Kim. Overrun handling approaches for overload-prone soft real-time systems[J], Advances in Engineering Software, 2007, 38(11-12): 780-794.
52. 方安平, 肖强. μC/OS-II 的实时性能分析[J], 单片机与嵌入式系统应用, 2005, 8: 12-14.
53. Enrico Bini, Giorgio C. Buttazzo. Measuring the Performance of Schedulability Tests[J], Real-Time Systems, 2005, 30(1-2): 129-154.
54. 郑泽胜. 嵌入式系统以及实时软件开发, http://www.pocketix.com 嵌入式 Linux 中文社区

致谢

值此论文结束之际，我首先感谢我的导师李晶皎教授。本文是在她的悉心指导下才得以顺利完成的。在攻读硕士学位期间，李老师不仅在学业上给予我很多指导和帮助，还在生活上给予我无微不至的关怀和帮助。李老师渊博的知识、严谨的研究治学态度、活跃的学术思想和敏锐的洞察力使我受益非浅；李老师崇高的师德、正直的为人、乐于助人的品德及精益求精的工作态度时时激励着我，是我人生道路中的楷模。这两年半的珍贵时光将成为我人生中最重要的阶段并将使我受益终生。以此，我对李老师致以最崇高的敬意和最诚挚的谢意。

衷心感谢王爱侠老师和闫爱云老师。王老师和闫老师在我融入实验室大家庭的过程中，起了不可替代的作用。在项目和论文上，王老师和闫老师也给予了很多宝贵的意见，每次碰到问题，她们都能耐心而亲切的解答，在生活上王老师和闫老师的照料更是让我非常感激。

衷心感谢实验室的兄弟姐妹。两年左右的实验室生活是迄今为止最令我难忘的日子。很庆幸自己的青春时光能在这样一个环境中度过。这两年充满了太多的欢笑，一张张笑脸会永远刻在我脑中。

衷心感谢寝室的兄弟。感谢他们在我困难焦虑的时候给予的安慰和帮助。

感谢我的爸爸妈妈。繁忙的实验室工作使得我总是难以尽足孝道，而他们却从无怨言。再次向老爸老妈说声谢谢。

最后，向那些因为我忙于实验室工作而忽略了联络的朋友们表示歉意。