国防科工委“十五”规划教材. 信息与通信技术

嵌入式实时操作系统及应用开发

罗　蕾　主编

北京航空航天大学出版社

北京理工大学出版社　西北工业大学出版社
哈尔滨工业大学出版社　哈尔滨工程大学出版社

内容简介

本书以嵌入式软件的核心——嵌入式实时操作系统为重点，以应用为目的，全面介绍嵌入式系统。它使读者既能对嵌入式系统及开发有一个全景的把握，又能深入理解和使用嵌入式实时操作系统。同时，为了加强理论与实践的结合，还专为该教材配套了实验系统。实验系统包括嵌入式实时操作系统和集成开发工具，提供了丰富的实验用例。

本书共分 9 章。第 1～3 章介绍嵌入式软硬件系统，包括基本概念、组成、特点、分类和发展趋势等；第 4～8 章重点介绍嵌入式实时内核，包括任务管理与调度、同步互斥与通信、中断和时间管理、内存管理和 I/O 管理；第 9 章介绍嵌入式系统软件的开发，包括开发模式、任务划分方法等。

本书可以作为高等学校有关嵌入式系统方面教学的本科生或研究生的教材，也适合于从事嵌入式系统研发的人员参考。

图书在版编目(CIP)数据

嵌入式实时操作系统及应用开发/罗蕾主编. —北京：北京航空航天大学出版社，2005. 1
ISBN 7-81077-580-4

Ⅰ. 嵌… Ⅱ. 罗… Ⅲ. 实时操作系统
Ⅳ. TP316. 2

中国版本图书馆 CIP 数据核字(2004)第 105631 号

嵌入式实时操作系统及应用开发

罗 蕾 主编
责任编辑 刘晓明
北京航空航天大学出版社出版发行
北京市海淀区学院路 37 号(100083)
发行部电话：(010)82317024 传真：(010)82328026
http://www.buaapress.com.cn E-mail：bhpress@263.net
涿州市新华印刷有限公司印装 各地书店经销
*
开本：787×960 1/16
印张：20.75 字数：465 千字
2005 年 1 月第 1 版 2005 年 1 月第 1 次印刷
印数：5 000 册
ISBN 7-81077-580-4 定价：34.00 元(含光盘)

国防科工委“十五”规划教材编委会

（按姓氏笔画排序）

前　言

计算机是20世纪人类社会最伟大的发明之一。从1946年第一台计算机ENIAC在美国宾夕法尼亚大学的诞生到现在，计算机经历了两个发展阶段：大型计算机和个人计算机阶段。伴随着21世纪的曙光，计算机技术正进入下一个充满机遇的阶段——“后PC时代”或“无处不在的计算机”阶段。无处不在的计算机是指计算机彼此互联，而且计算机与使用者的比率达到100∶1以上。无处不在的计算机包括通用计算机和嵌入式计算机，在100∶1的比例中，95％以上都是嵌入式计算机。

嵌入式计算和控制技术的发展可追溯到20世纪60年代初，航天工业从那时起已开始使用实时控制计算机进行航天飞行器的测控和导航。随着半导体和微处理器技术的发展，到了20世纪80年代，大量4位、8位和16位微处理器开始应用于各种设备中，如飞行器、舰船、汽车、电话、电视机、微波炉、照相机甚至玩具，嵌入式实时操作系统开始应用。进入后PC时代，4位、8位和16位微处理器逐步让位于32位嵌入式芯片，嵌入式计算机的应用更加广泛。它将广泛应用于消费电子、有线和移动通信、汽车、医疗、工业控制、航空航天及国防等领域的民用和军用电子设备。形式多样的嵌入式计算机正努力把Internet连接到人们生活的各个角落，其消费量将以亿计，逐步形成一个充满商机的巨大产业。

随着嵌入式应用复杂度的提高，嵌入式软件的规模也发生了指数型增长。软件的实现从某种意义上说决定了产品的功能，已成为新产品成功与否的关键因素，是未来市场竞争力的重要体现。由于嵌入式系统具备硬件平台多样性和应用个性化的特点，因此嵌入式软件呈现出一种高度细分的市场格局，国外产品进入也很难垄断整个市场，这为我国的软件产业提供了一个难得的发展机遇。

目前市场急需专业化的嵌入式软件人才，为适应这种需求，国内高校大多已开设嵌入式软件的各种课程。电子科技大学自1990年以来就开始从事嵌入式软件的研究、开发和教学工作，承担并完成了国家863、国防预研和电子发展基金等多项与嵌入式软件相关的重点课题，开发出具有自主知识产权的嵌入式实时操作系统CRTOS及开发工具，其成果获得了多项部级科技进步奖。从1999年开始进

行产业化工作以来，已走出了一条学、研、产相结合的道路，并于 2000 年推出了中国第一套完整的嵌入式实时软件开发平台“道系统(DeltaSystem)”。

本教材就是在此基础上，以嵌入式软件的核心——嵌入式实时操作系统为重点，以应用为目的，全面介绍嵌入式系统。它使读者既能对嵌入式系统及开发有一个全景的把握，又能深入理解和使用嵌入式实时操作系统。同时，为了加强理论与实践的结合，还专为该教材配套了实验系统。实验系统包括嵌入式实时操作系统和集成开发工具，提供了丰富的实验和手册。读者利用 PC 机就可以自己动手搭建嵌入式系统的开发平台，熟悉应用开发，更好地学习和理解嵌入式系统的基础知识：从纯软件到硬/软件结合、从“纸”上谈兵(编程序)到“板”上谈兵、从“懂”怎么做到“会”做以及从讲/听到讲/听/做。

教材的内容

教材共分 9 章，参与教材编写的主要人员有李允(负责第 5，7，8 章)、陈丽蓉(负责第 4，6，9 章)、雷航(负责第 1，9 章部分工作)、罗蕾(负责第 1，2，3 章)，全书由罗蕾统稿。

教材包括嵌入式系统基础、嵌入式实时操作系统及应用开发三方面的内容。具体内容如下：

第 1 章　嵌入式系统导论。讲述什么是嵌入式系统、嵌入式系统的发展历程、嵌入式系统的特点、嵌入式系统的分类和应用领域、嵌入式实时系统的实时性和可靠性及嵌入式系统的发展趋势。

第 2 章　嵌入式硬件系统。讲述嵌入式硬件系统的组成、嵌入式微处理器的特点、主流的嵌入式微处理器 ARM 系列和 MIPS 系列、AMBA 和 PCI 总线、嵌入式系统存储器结构。

第 3 章　嵌入式软件系统。讲述嵌入式软件的特点和分类、嵌入式软件的体系结构、运行流程，嵌入式操作系统结构、组成、功能、特点和发展趋势，嵌入式软件开发工具的分类、交叉开发环境，嵌入式软件实现阶段的开发过程及开发工具的发展趋势。

第 4 章　嵌入式实时内核基础。讲述嵌入式实时内核的基础知识，主要包括嵌入式实时内核的关键设计问题、主要功能和主要性能指标，为读者深入学习实时内核打下良好的基础。

第 5 章　任务管理与调度。讲述什么是任务和任务的分类、主要特性及内

容，任务管理机制，嵌入式实时系统常见的几种调度算法，优先级反转及解决方法，多处理器调度。

第 6 章　同步、互斥与通信。讲述任务间、任务与中断处理程序间常见的同步、互斥与通信机制（信号量、邮箱、消息队列、事件和异步信号）。

第 7 章　中断和时间管理。讲述中断分类、中断处理过程及中断管理机制等，硬件时钟设备。（实时时钟 RTC 和定时器/计数器）及与操作系统的关系，时间管理机制。

第 8 章　存储管理和 I/O 管理。讲述嵌入式实时系统对存储管理的需求以及存储管理的三种机制：固定大小存储区管理、可变大小存储区管理及保护机制、嵌入式系统 I/O 管理的特点及机制。

第 9 章　嵌入式系统软件的开发。讲述嵌入式系统的开发模式、任务划分的方法、嵌入式系统的设计工具以及调度算法时限可满足性分析等。

配套实验系统的内容

配套了嵌入式实时软件开发平台“道系统”：

- 嵌入式实时操作系统 DeltaOS　嵌入式实时内核 DeltaCORE、嵌入式 TCP/IP DeltaNET、嵌入式文件系统 DeltaFILE。
- 嵌入式集成交叉开发环境 LambdaTOOL　集成开发环境 LambdaIDE、交叉编译器 LambdaGCC、交叉调试器 LambdaGDB、目标监控器 LambdaTRA 等。

共配套 19 个实验：

- 嵌入式交叉开发环境建立实验　1 个；
- 嵌入式实时内核实验　10 个；
- 嵌入式 TCP/IP 实验　5 个；
- 嵌入式文件系统实验　3 个。

嵌入式实时软件开发平台、实验和手册放在本教材所附的 CD 中。

致　谢

本书除得到国防科工委重点教材建设的支持外，还得到电子科技大学研究生院课程建设及电子科技大学教务处优秀教材建设的支持。编者对此深表感谢。

感谢电子科技大学嵌入式实时系统研究室的各位老师和研究生的支持。感

谢北京科银京成公司的帮助和支持。感谢北航出版社编辑和出版人员的辛勤工作。

感谢家人和朋友们的支持。

由于作者知识所限，书中不足之处，恳请各位专家和读者赐正。您可通过E-mail：lluo@uestc.edu.cn 或 lrchen@uestc.edu.cn 与我们联系。

编　者

2004 年 1 月

于电子科技大学

总　　序

国防科技工业是国家战略性产业，是国防现代化的重要工业和技术基础，也是国民经济发展和科学技术现代化的重要推动力量。半个多世纪以来，在党中央、国务院的正确领导和亲切关怀下，国防科技工业广大干部职工在知识的传承、科技的攀登与时代的洗礼中，取得了举世瞩目的辉煌成就；研制、生产了大量武器装备，满足了我军由单一陆军，发展成为包括空军、海军、第二炮兵和其他技术兵种在内的合成军队的需要，特别是在尖端技术方面，成功地掌握了原子弹、氢弹、洲际导弹、人造卫星和核潜艇技术，使我军拥有了一批克敌制胜的高技术武器装备，使我国成为世界上少数几个独立掌握核技术和外层空间技术的国家之一。国防科技工业沿着独立自主、自力更生的发展道路，建立了专业门类基本齐全，科研、试验、生产手段基本配套的国防科技工业体系，奠定了进行国防现代化建设最重要的物质基础；掌握了大量新技术、新工艺，研制了许多新设备、新材料，以“两弹一星”、“神舟”号载人航天为代表的国防尖端技术，大大提高了国家的科技水平和竞争力，使中国在世界高科技领域占有了一席之地。十一届三中全会以来，伴随着改革开放的伟大实践，国防科技工业适时地实行战略转移，大量军工技术转向民用，为发展国民经济作出了重要贡献。

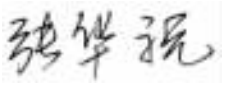
国防科技工业是知识密集型产业，国防科技工业发展中的一切问题归根到底都是人才问题。50 多年来，国防科技工业培养和造就了一支以“两弹一星”元勋为代表的优秀的科技人才队伍，他们具有强烈的爱国主义思想和艰苦奋斗、无私奉献的精神，勇挑重担，敢于攻关，为攀登国防科技高峰进行了创造性劳动，成为推动我国科技进步的重要力量。面向新世纪的机遇与挑战，高等院校在培养国防科技人才，传播国防科技新知识、新思想，攻克国防基础科研和高技术研究难题当中，具有不可替代的作用。国防科工委高度重视，积极探

索，锐意改革，大力推进国防科技教育特别是高等教育事业的发展。

高等院校国防特色专业教材及专著是国防科技人才培养当中重要的知识载体和教学工具，但受种种客观因素的影响，现有的教材与专著整体上已落后于当今国防科技的发展水平，不适应国防现代化的形势要求，对国防科技高层次人才的培养造成了相当不利的影响。为尽快改变这种状况，建立起质量上乘、品种齐全、特点突出、适应当代国防科技发展的国防特色专业教材体系，国防科工委全额资助编写、出版 200 种国防特色专业重点教材和专著。为保证教材及专著的质量，在广泛动员全国相关专业领域的专家、学者竞投编著工作的基础上，以陈懋章、王泽山、陈一坚院士为代表的 100 多位专家、学者，对经各单位精选的近 550 种教材和专著进行了严格的评审，评选出近 200 种教材和学术专著，覆盖航空宇航科学与技术、控制科学与工程、仪器科学与技术、信息与通信技术、电子科学与技术、力学、材料科学与工程、机械工程、电气工程、兵器科学与技术、船舶与海洋工程、动力机械及工程热物理、光学工程、化学工程与技术、核科学与技术等学科领域。一批长期从事国防特色学科教学和科研工作的两院院士、资深专家和一线教师成为编著者，他们分别来自清华大学、北京航空航天大学、北京理工大学、华北工学院、沈阳航空工业学院、哈尔滨工业大学、哈尔滨工程大学、上海交通大学、南京航空航天大学、南京理工大学、苏州大学、华东船舶工业学院、东华理工学院、电子科技大学、西南交通大学、西北工业大学、西安交通大学等，具有较为广泛的代表性。在全面振兴国防科技工业的伟大事业中，国防特色专业重点教材和专著的出版，将为国防科技创新人才的培养起到积极的促进作用。

党的十六大提出，进入 21 世纪，我国进入了全面建设小康社会、加快推进社会主义现代化的新的发展阶段。全面建设小康社会的宏伟目标，对国防科技工业发展提出了新的更高的要求。推动经济与社会发展，提升国防实力，需要造就宏大的人才队伍，而教育是奠基的柱石。全面振兴国防科技工业必须始终把发展作为第一要务，落实科教兴国和人才强国战略，推动国防科技工业

走新型工业化道路，加快国防科技工业科技创新步伐。国防科技工业为有志青年展示才华，实现志向，提供了缤纷的舞台，希望广大青年学子刻苦学习科学文化知识，树立正确的世界观、人生观、价值观，努力担当起振兴国防科技工业、振兴中华的历史重任，创造出无愧于祖国和人民的业绩。祖国的未来无限美好，国防科技工业的明天将再创辉煌。



目　　录

第 1 章　嵌入式系统导论

1.1　嵌入式系统概述

计算机是 20 世纪人类社会最伟大的发明之一，也是 20 世纪科学技术发展的三大主题之一。从 1946 年第一台计算机 ENIAC(Electronic Numerical Integrator And Computer)在美国宾夕法尼亚大学的诞生到现在，计算机的发展经历了三大阶段：

- 第一阶段　大型机阶段，始于 20 世纪 50 年代，IBM，Burroughs 和 Honeywell 等公司率先研制出大型机。
- 第二阶段　个人计算机阶段，始于 20 世纪 70 年代。
- 第三阶段　进入 21 世纪，计算机技术正进入充满机遇的阶段，即“后 PC 时代”或“无处不在的计算机”阶段。

施乐公司 Palo Alto 研究中心主任 Mark Weiser 认为：“从长远来看，PC 机和计算机工作站将衰落，因为计算机变得无处不在，例如在墙里，在手腕上，在手写电脑中(像手写纸一样)等，随用随取，伸手可及。”

目前全世界的计算机科学家正在形成一种共识：计算机不会成为科幻电影中的那种贪婪的怪物，而是将变得小巧玲珑、无处不在。它们藏身在任何地方，又消失在所有地方，功能强大，却又无影无踪。人们将这种思想命名为“无处不在的计算机”。

无处不在的计算机是指计算机彼此互联(如图 1－1 所示)，而且计算机与使用者的比率达到和超过 100∶1 的阶段。无处不在的计算机包括通用计算机和嵌入式计算机系统，在 100∶1 的比例中，95 %以上都是嵌入式计算机系统，并非通用计算机。

图 1－1　无处不在的计算机彼此互联

嵌入式计算机系统在应用数量上远远超过了各种通用计算机。一台通用计算机的外部设备中就包含了 5～l0 个嵌入式微处理器，例如键盘、鼠标、软驱、硬盘、显示卡、显示器、Modem、网卡、声卡、打印机和扫描仪等。

通用计算机(如表 1－1 所列)是具有通用计算平台和标准部件的“看得见”的计算机，如 PC 机、服务器、大型计算机等。其硬件一般包括主机、存储设备(软盘、硬盘、光驱等)及标准的计算机外部设备等，如显示设备(CRT，LCD 显示器等)、输入设备(键盘、鼠标等)和联网设备等。图 1－2 所示是通用计算机的典型硬件组成。通用计算机既可作为开发平台，又可作为运行平台，且应用程序可按用户需要随时改变，即重新编制。

表 1－1　通用计算机与嵌入式系统对比

特　征	通用计算机	嵌入式系统
形式和类型	“看得见”的计算机； 按其体系结构、运算速度和结构规模等因素分为大、中、小型机和微机	“看不见”的计算机； 形式多样，应用领域广泛，一般按应用分类
组成	通用处理器、标准总线和外设； 软件和硬件相对独立	面向应用的嵌入式微处理器，总线和外部接口多集成在处理器内部； 软件与硬件是紧密集成在一起的
开发方式	开发平台和运行平台都是通用计算机	采用交叉开发方式，开发平台一般是通用计算机，运行平台是嵌入式系统
二次开发性	应用程序可重新编制	一般不能再编程
发展目标	变为功能电脑，普遍进入社会	变为专用电脑，实现“普及计算”(pervasive computing)

图 1－2　通用计算机硬件组成

嵌入式计算机系统即“看不见”的计算机，一般只是运行平台，不能独立作为开发平台。它们不能被用户编程，有一些专用的 I/O 设备，对用户的接口是应用专用的。不严格地说：嵌入式计算机系统是任意包含可编程计算机的设备，但这种设备不是作为通用计算机而设计的(“Any sort of device which includes a programmable computer but itself is not intended to be a general－purpose computer”)，比如 PC 可以用于搭建嵌入式计算机系统，但 PC 不能称为嵌入式计算机系统。

通常将嵌入式计算机系统简称为嵌入式系统。

嵌入式系统已渗透到日常生活的各个方面，不同的应用其形式和名称各异，因此目前没有

一个统一定义。除了上述定义外，常用的定义归纳如下：

① 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、体积和功耗严格要求的专用计算机系统。

② IEEE(国际电气和电子工程师协会)定义是“Device used to control，monitor，or assist the operation of equipment，machinery or plants”。

③ 嵌入式系统是将先进的计算机技术、半导体技术和电子技术与各个行业的具体应用相结合后的产物。这一点就决定了它必然是一个技术密集、资金密集、高度分散、不断创新的知识集成系统。

嵌入式系统一般由嵌入式硬件和软件组成，且软件与硬件是紧密集成在一起的。硬件以嵌入式微处理器为核心，集成存储器和系统专用的输入/输出设备；软件包括初始化代码及驱动、嵌入式操作系统和应用程序等，这些软件有机地结合在一起，形成系统特定的一体化软件。

1.1.1　嵌入式系统的发展历程

嵌入式计算机系统出现于 20 世纪 60 年代。40 多年来随着计算机技术、电子信息技术等的发展，嵌入式计算机的各项技术也蓬勃发展，市场迅猛扩大，嵌入式计算机已深入到生产和生活的每个角落。

1. 嵌入式系统的出现和兴起(1960—1970 年)

第一代电子管计算机(1946—1957 年)是像 ENIAC 那样占地 170 m^2、质量达 30 t、耗电 140 kW 的“庞然大物”，无法满足嵌入式计算所提出的体积小、质量轻、耗电少、可靠性高及实时性强等一系列要求。20 世纪 60 年代以晶体管、磁芯存储为基础的计算机开始用于航空等军用领域。第一台机载专用数字计算机是奥托内蒂克斯公司为美国海军舰载轰炸机“民团团员”号研制的多功能数字分析器(verdan)。它由几个体积相当大的黑盒子组成，中央处理装置处理所有主要电子系统传来的信号，开始有了数据总线的雏形。同时，嵌入式计算机开始应用于工业控制。1962 年美国一个乙烯厂实现了工业装置中的第一个直接数字控制(DDC)。

嵌入式系统的兴起是在 1965—1970 年。当时计算机已开始采用集成电路，也就是通常所说的第三代计算机。在军事和航空航天领域的需求推动下，计算机的硬、软件技术达到了可以把人送上月球再返回地面的可靠性要求。

第一次使用机载数字计算机控制的是 1965 年发射的 Gemini3 号；第一次通过容错来提高可靠性的是 1968 年的阿波罗 4 号、土星 5 号。阿波罗中的嵌入式计算机系统提供人机交互功能，通过该系统与人的紧密结合来引导飞行。在这一时期，计算机系统结构取得了许多重大发展，出现了并行、先行控制，以及流水线、操作系统等新技术和影响广泛的 IBM360 系列机。

1963 年 DEC 公司推出的第一台商用小型机由 PDP8 发展成 PDP11 系列。它的单总线结构、高速通用寄存器、强有力的中断系统和交叉存取技术，很好地适应了工业控制系统实时嵌入式应用的需求，成为工业生产集中控制的主力军。在军用领域中，为了可靠和满足体积、质

量的严格要求，还需为各个武器系统设计五花八门的、专用的嵌入式计算机系统。

2. 嵌入式系统开始走向繁荣，软件和硬件日臻完善(1971—1989 年)

(1) 微处理器问世

嵌入式系统的大发展是在微处理器问世之后。1971 年 11 月，Intel 公司成功地把算术运算器和控制器电路集成在一起，推出了世界上第一片微处理器 Intel 4004。它本来是专为袖珍计算器而设计的，由于体积小、质量轻、价格低廉和成功的设计促使 Intel 公司把它进一步通用化，推出了 4 位的 4040 和 8 位的 8008。

1973—1977 年各厂家推出了许多 8 位的微处理器，包括 Intel 公司的 8080/8085，Motorola 公司的 6800/6802，Zilog 公司的 Z80 和 Rockwell 公司的 6502 等。微处理器不仅用来组成微型计算机，而且用来制造仪器仪表、医疗设备、机器人和家用电器等嵌入式系统。据统计，兼容 8085 微处理器的出货量超过了 7 亿片，这些芯片大部分是用于嵌入式工业控制。这时，人们再也不必为设计一台专用机而研制专用的电路、专用的运算器了，只需以微处理器为基础进行设计。

微处理器的广泛应用形成了一个广阔的嵌入式应用市场，计算机厂家除了要继续以整机方式向用户提供工业控制计算机系统外，还开始大量地以插件方式向用户提供 OEM 产品，再由用户根据自己的需要构成专用的工业控制微型计算机，嵌入到自己的系统设备中。为了灵活兼容，形成了标准化、模块化的单板机系列。流行的单板计算机有 Intel 公司的 iSBC 系列、Zilog 公司的 MCB 等。这样，人们就不必从选择芯片开始来设计一台专用的嵌入式计算机了，只要选择一套适合自己应用的 CPU 板、存储器板和各式 I/O 插件板，就可以组建一台专用计算机。用户和厂家都希望从不同的厂家选购最适合的 OEM 产品，插入外购或自制的机箱中就形成新的系统，即希望插件是互相兼容的，这就导致了工业控制微机系统总线的诞生。1976 年 Intel 公司推出了 Multibus，1983 年将其扩展为带宽达 40 MB/s 的 Multibus Ⅱ；1978 年Prolog 公司设计的简单的 STD 总线广泛用于小型嵌入式系统；1981 年 Motorola 公司推出的 VME_Bus 则与 Multibus Ⅱ 瓜分高端市场。

(2) 单片机、DSP 出现

随着微电子工艺水平的提高，集成电路设计制造商开始把嵌入式应用所需要的微处理器、I/O 接口、A/D 转换、D/A 转换、串行接口以及 RAM 和 ROM 全部集成到一个 VLSI，制造出面向 I/O 设计的微控制器，就是人们俗称的单片机。

最早的单片机 Intel 8048 出现在 1976 年。20 世纪 80 年代初 Intel 公司在它的基础上开发出了著名的 8051，Motorola 公司推出 68HC05，Zilog 公司则转向专门生产 Z80 单片机。这些含有 8 位微处理器、RAM、ROM、几个 8 位并口、几个全双工串口及 2 个 16 位定时器的单片机，迅速地渗入到消费电子、医用电子、智能控制、通信、仪器仪表和交通运输等各种领域。可以根据各种不同的应用要求不断改进工艺，提高运行速度，降低功耗；同时，把不同的外设接口配置到芯片内，衍生出几十个品种、各种各样的型号，可以说是“总有一款适合你”。如果 8 位处理器处理速度太慢，还有 16 位的单片机。

此外，为了高速实时地处理数字信号，1982 年世界上诞生了首枚 DSP 芯片。DSP 是在模拟信号变换成数字信号以后进行高速实时处理的专用处理器，其处理速度比当时最快的 CPU 还快 10～50 倍。20 世纪 80 年代后期，第三代 DSP 芯片问世，运算速度进一步提高，其应用范围逐步扩大到通信、控制及计算机等领域。

(3) 软件技术的进步使嵌入式系统日臻完善

20 世纪 80 年代后，嵌入式计算机的大发展还要归功于软件技术的进步。最初的嵌入式计算机都是非常专用的，软件也是专门用汇编甚至机器语言编制的。在微处理器出现的初期，为了保障嵌入式软件的时间及空间效率，软件也只能用汇编语言编写。这样，嵌入式系统的开发只能由计算机专业人员用原始的工具来完成，其效率低、周期长。由于微电子技术的进步，对软件时空效率的要求不再那么苛刻了，嵌入式计算机的软件开始使用 PL/M，C 等高级语言。在军用领域，为改变各种武器系统使用五花八门的专用语言和软件使系统费用不断增加的状况，美国推行三军通用的 Ada 语言，开发可重用的通用软件，提高软件生产效率。

嵌入式系统大多是实时系统。对于复杂的嵌入式系统来说，除了需要高级语言开发工具外，还需要嵌入式实时操作系统的支持。20 世纪 70 年代的小型计算机为了适应实时应用领域的需求，由计算机生产厂家为自己的机器配置了实时操作系统(RTOS)。20 世纪 80 年代初开始出现一批软件公司，推出商品化的嵌入式实时操作系统和各种开发工具，像 Ready System公司的 VRTX 和 XRAY，Integrated System Incorporation (ISI) 公司的 pSOS 和 PRISM＋，WindRiver 公司的 VxWorks 和 Tornado，QNX 公司的 QNX 等。这些操作系统具有强实时、可剪裁、可配置、可扩充和可移植的特点，支持主流的嵌入式微处理器。在开发工具方面，他们提供不同种类的面向软、硬件开发的工具，如硬件仿真器、源码级的交叉调试器等。商用嵌入式实时操作系统和开发工具的出现和推广应用，使嵌入式系统的开发从作坊式向分工协作规模化的方向发展，促使嵌入式应用扩展到更广阔的领域。

3. 嵌入式系统应用走向纵深(1990 年至今)

进入 20 世纪 90 年代，在分布控制、柔性制造、数字化通信和数字化家电等巨大需求的牵引下，嵌入式系统的硬件、软件技术进一步加速发展，应用领域进一步扩大。今天，手机、数码相机、VCD、数字电视、路由器和交换机等都是嵌入式系统。大多数豪华轿车每辆拥有约50 个嵌入式微处理器。它们控制着从发动机火花塞、传动轴，直到避免由于关门时产生的压力而使司机耳朵胀痛的控制系统等众多部件。在一架先进的飞机上可能有十几台嵌入式系统、上百个单片机。如果没有机载计算机，F—16 战斗机就不能成为可靠的空中武器平台。当飞行员通过传统的控制系统控制飞机时，机载计算机能使飞机在保持空中飞行的前提下尽可能满足飞行员的各种要求。最新的波音 777 宽体客机上约有 1 000 个微处理器。在不久的将来你会在你的家里发现几十到上百个嵌入式系统在为你服务。

4 位、8 位、16 位微处理器芯片已逐步让位于 32 位嵌入式微处理器芯片，如图 1－3 所示。面向不同应用领域(application specific)、功能强大、集成度高、种类繁多、价格低廉和低功耗

的 32 位芯片已大量应用于各种各样的军用和民用设备。DSP 向高速、高精度和低功耗发展。DSP 与通用嵌入式微处理器集成已成为现实，并已大量应用于嵌入式系统，如手机、IP 电话等。就通信设备而言，32 位芯片使得开发高度智能化的通信设备成为可能，例如数字移动电话、宽带网交换机、路由器和卫星系统等。在工业控制领域，嵌入式 PC 大量应用于嵌入式系统中；PC104，CPCI(Compact PCI)总线因其成本低、兼容性好也已广泛应用。

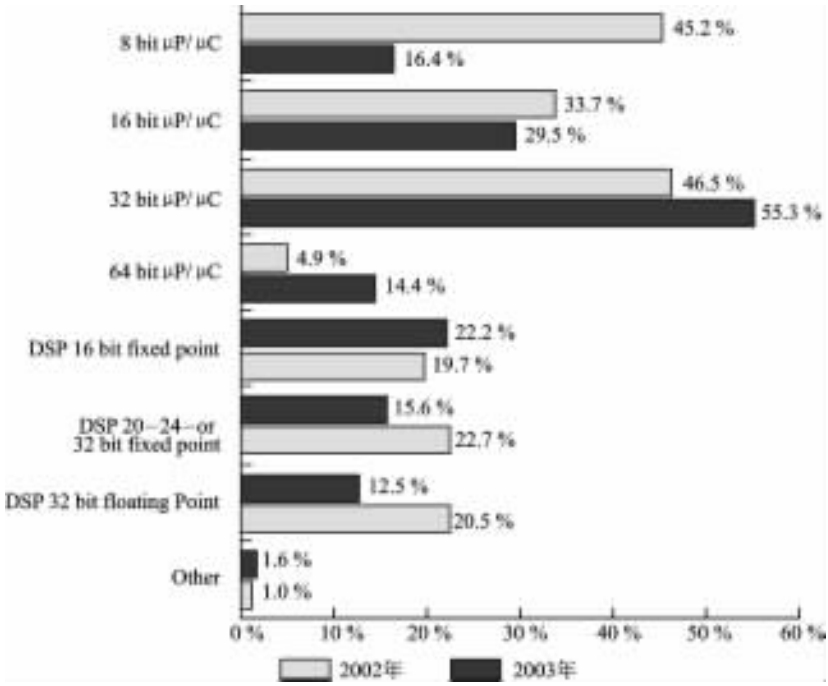


图 1－3　采用微处理器设计的趋势

随着微处理器性能的提高，嵌入式软件的规模也发生指数型增长。32 位芯片将能够执行由上百万行 C 代码构成的复杂程序，使得嵌入式应用具备高度复杂和智能化的功能。软件的实现从某种意义上说决定了产品的功能，已成为新产品成功与否的关键因素，如图 1－4 所示。

为此，嵌入式系统已大量采用嵌入式操作系统。嵌入式操作系统的功能不断扩大和丰富，由 20 世纪 80 年代只有内核，发展为包括内核、网络、文件、图形接口、嵌入式 JAVA、嵌入式 CORBA 及分布式处理等丰富功能的集合。嵌入式操作系统在嵌入式软件中的作用越来越大，所占的比例逐步提高，从最初的 10 % 左右，到 90 年代初期的 30 %左右，然后到 90 年代中期的 60 %左右，再到 90 年代后期的 80 %左右，如图 1－5 所示。

此外，嵌入式开发工具更加丰富，其集成度和易用性不断提高。目前，不同厂商已开发出不同类型的嵌入式开发工具，可以覆盖嵌入式软件开发过程的各个阶段，提高嵌入式软件开发效率。

图 1－4　嵌入式软件的地位

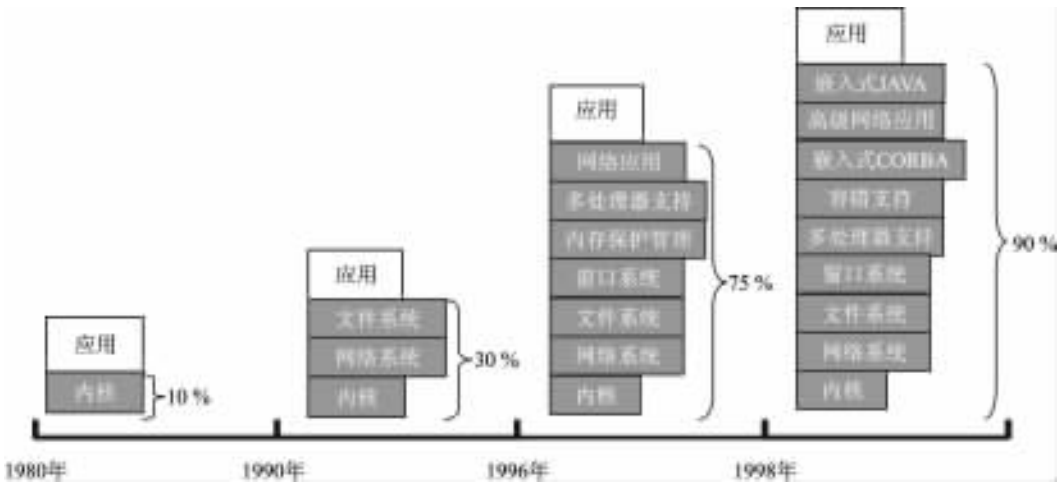


图 1－5　嵌入式操作系统的发展

1.1.2　嵌入式系统的特点

嵌入式系统同通用计算机相比具有以下特点。

1. 嵌入式系统通常是形式多样的面向特定应用的软硬件综合体

嵌入式系统一般用于特定的任务，其硬件和软件都必须高效率地设计，量体裁衣，去除冗余；而通用计算机则是一个通用的计算平台。

嵌入式微处理器与通用微处理器的最大不同就是每种嵌入式微处理器大多专用于某种或几种特定的应用,工作在为特定用户群设计的系统中。它通常都具有低功耗、体积小和集成度高等特点,能够把通用微处理器中许多由板卡完成的功能集成在芯片内部,从而有利于嵌入式系统设计趋于小型化,移动能力大大增强,与网络的耦合也越来越紧密。

嵌入式软件是应用程序和操作系统两种软件的一体化程序。对于通用计算机系统,操作系统等系统软件和应用软件之间界限分明。换句话说,在统一配置的操作系统环境下,应用程序是独立的运行软件,可以分别装入执行。但是,在嵌入式系统中,这一界限并不明显。这是因为应用系统配置差别较大,所需操作系统繁简不一,I/O 操作也不标准,这部分驱动软件常常由系统设计者完成。这就要求采用不同配置的操作系统和应用程序,链接装配成统一运行的软件系统;也就是说,应在系统总体设计目标指导下将它们综合加以考虑、设计与实现。

2. 嵌入式系统得到多种处理器类型和体系结构的支持

通用计算机采用少数的处理器类型和体系结构,而且处理器掌握在少数大公司手里;而嵌入式系统可采用多种类型的处理器和处理器体系结构。在嵌入式微处理器产业链上,IP 设计、面向应用的特定嵌入式微处理器的设计以及芯片的制造已形成巨大的产业,大家分工协作,形成多赢模式。目前有上千种嵌入式微处理器和几十种嵌入式微处理器体系结构可以选择,主流的体系有 ARM,MIPS,PowerPC,X86 和 SH 等。

3. 嵌入式系统通常极其关注成本

嵌入式系统通常需要注意的成本是系统的成本,特别是量大的消费类数字化产品,其成本是产品竞争的关键因素之一。

嵌入式系统的系统成本包括:

- 一次性的开发成本　NRE(Non-Recurring Engineering)成本;
- 每个产品的成本　硬件 BOM(Bill Of Material)、外壳包装成本和软件版税等。

批量产品的总体成本=NRE 成本+每个产品的成本×产品总量

每个产品的最后成本=总体成本/产品总量=NRE 成本/产品总量+每个产品的成本

4. 嵌入式系统有实时性和可靠性的要求

嵌入式系统有实时性的要求表现在两个方面:一方面大多数实时系统都是嵌入式系统;另一方面嵌入式系统多数有实时性的要求,且软件一般是固化运行或直接加载在内存中运行的,具有快速启动的特点。嵌入式系统对实时的强度要求各不一样,可分为硬实时系统和软实时系统。

嵌入式系统一般要求具有出错处理和自动复位功能,特别是对于一些在极端环境下运行的嵌入式系统而言,其可靠性设计尤其重要。在大多数嵌入式系统中一般都包括一些硬件和软件机制来保证系统的可靠性。比如硬件的看门狗定时器,它在软件失去控制后使系统重新开始正常运行。软件的可靠性机制包括内存保护和重启机制等。

5. 嵌入式系统使用的操作系统的特性

嵌入式系统使用的操作系统一般是适应多种类型处理器、可剪裁、轻量型、实时可靠和可固化的嵌入式操作系统。

由于嵌入式系统应用的特点，嵌入式操作系统像嵌入式微处理器一样，也是多姿多彩的。大多数商业嵌入式操作系统可同时支持不同种类的嵌入式微处理器；可根据应用的情况进行剪裁、配置；与通用计算机操作系统相比其规模小，所需的资源有限，如内核规模在几十 KB 左右；一般包括一个实时内核，调度算法一般采用基于优先级的可抢占的调度算法，同时目前一些操作系统还提供了 HA(High Available)机制；能与应用软件一起固化在 Flash 中直接运行，或全部加载到 RAM 中运行。

6. 嵌入式系统开发需要专门工具和特殊方法

多数嵌入式系统开发意味着软件与硬件的并行设计和开发，其开发过程一般分为几个阶段：产品定义、软件与硬件的设计与实现、软件与硬件集成、产品测试与发布、维护与升级。

由于嵌入式系统资源有限，一般不具备自主开发能力，产品发布后用户通常也不能对其中的软件进行修改，必须有一套专门的开发环境。该开发环境提供专门的开发工具(包括设计、编译、调试、测试等工具)，采用交叉开发的方式进行。

交叉开发环境由宿主机和目标机组成，如图 1－6 所示。宿主机一般采用通用计算机系统，是主要的开发环境，开发工具的大部分工作由它完成；目标机就是嵌入式系统，是所开发应用的执行环境，并配合宿主机的开发工作。

图 1－6　交叉开发环境

1.1.3　嵌入式系统的分类

嵌入式系统可按嵌入式处理器的位数、应用、实时性和软件结构等原则进行分类。

1. 按嵌入式处理器的位数来分类

按嵌入式处理器的位数来分，嵌入式系统可分为 4 位、8 位、16 位、32 位和 64 位。目前已大量应用的是 4 位、8 位、16 位嵌入式系统，32 位嵌入式系统正成为主流发展趋势，高度复杂的、高速的嵌入式系统已开始采用 64 位嵌入式处理器。

2. 按应用来分类

目前的嵌入式处理器一般是面向特定领域应用而设计的，集成 CPU Core、面向特定领域应用的基本 I/O 和总线等，因此，嵌入式系统可分为信息家电类、移动终端类、通信类、汽车电子类和工业控制类等。

3. 按实时性来分类

嵌入式系统可分为嵌入式实时系统和嵌入式非实时系统。

根据实时性的强弱，可将嵌入式实时系统进一步分为硬实时、软实时系统。

(1) 硬实时系统

该系统对系统响应时间有严格的要求，如果系统响应时间不能满足，就要引起系统崩溃或致命的错误。如飞机的飞控系统，如果不能及时控制飞机的飞行，就可能造成致命的后果。

(2) 软实时系统

该系统对系统响应时间有要求，但是如果系统响应时间不能满足，不会导致系统出现致命的错误或崩溃。如一台喷墨打印机平均处理周期从 2 ms 延长到 6 ms，其后果不过是打印速度从 3 页/min 下降到 1 页/min。

4. 按嵌入式软件结构来分类

从嵌入式系统的设计角度来看，嵌入式软件结构可以分为循环轮询系统、前后台系统、单处理器多任务系统以及多处理器多任务系统等几大类。

(1) 循环轮询系统

最简单的软件结构是循环轮询(polling loop)，程序依次检查系统的每一个输入条件，一旦条件成立就进行相应的处理，如图 1-7 和图 1-8 所示。

图 1-7 循环轮询系统运行方式

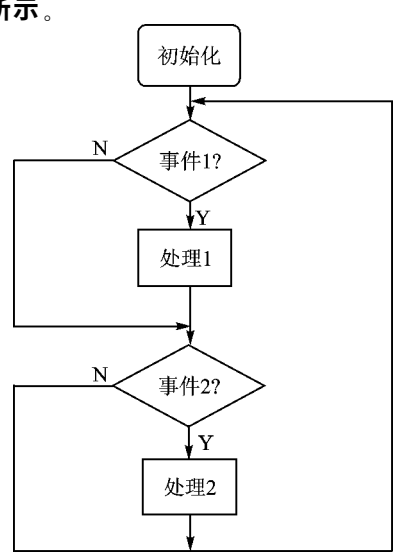


图 1-8 循环轮询系统流程

循环轮询系统通常的软件结构如下：

```
initialize()
while(true)  {
    if (condition_1) action_1();
    if (condition_2) action_2();
    ……
    if (condition_n) acition_n();
}
```

以下是该结构的优缺点。

优点：

- 对于简单的系统而言，便于编程和理解；
- 没有中断的机制，程序运行良好，不会出现随机的问题。

缺点：

- 应用领域有限（由于不可确定性）；
- 对于有大量 I/O 服务的应用不容易实现；
- 如果程序规模大，则不便于调试。

由此看来，循环轮询系统适合于慢速和非常快速的简单系统。

（2）前后台系统

前后台（foreground/background）系统又叫中断驱动系统。后台是一个一直在运行的系统，前台是由一些中断处理过程组成的。当有一个前台事件（外部事件）发生时，引起中断，中断后台运行，进行前台处理，处理完成后又回到后台（通常后台又称为主程序）。

这种系统的一个极端情况是，后台只是一个简单的循环，不做任何事情，所有其他工作都是由中断处理程序完成的。但大多数情况是中断只处理那些需要快速响应的事件，并且把 I/O 设备的数据放到内存的缓冲区中，再向后台发信号，其他的工作由后台来完成，比如对这些数据进行运算、存储、显示和打印等处理。

这种系统的软件运行方式如图 1－9 所示，程序流程如图 1－10 所示。

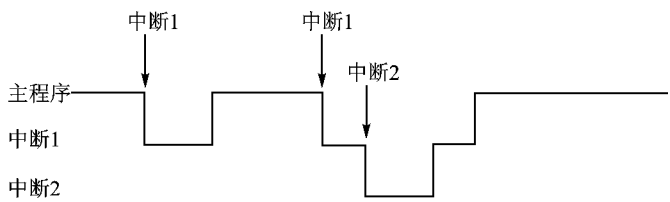


图 1－9　前后台系统运行方式

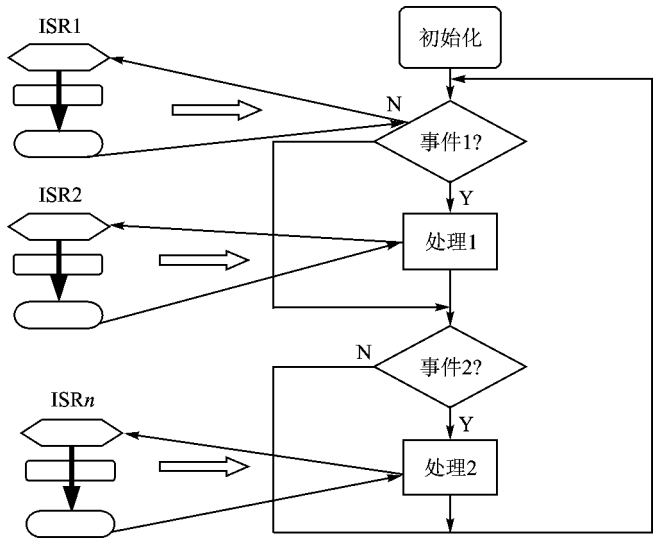


图 1－10　前后台系统程序流程

这种系统需要考虑的是中断的现场保护和恢复、中断嵌套、中断处理过程与主程序的协调(共享资源)问题。系统的性能主要由中断延迟时间(interrupt latency time)、响应时间(response time)和恢复时间(recovery time)来刻画,如图 1－11 所示。

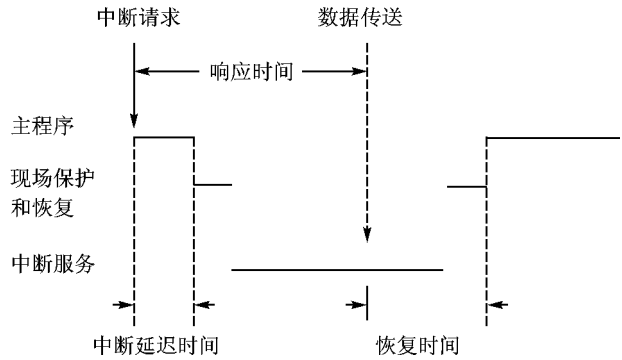


图 1－11　前后台系统的系统性能

前后台系统应用领域广泛,主要应用在一些小型的嵌入式系统中,以下是其主要优缺点。

优点:

- 可并发处理不同的异步事件,设计简单;
- 中断处理程序有多个,主程序一个;
- 无须学习 OS 相关的知识。

缺点：

- 对于复杂的系统而言，其主程序设计复杂，系统复杂度提高，可靠性降低；
- 实时性只能通过中断来保证，如果采用中断＋主程序的方式来处理事件，其实时性难以保证；
- 中断处理程序与主程序间的共享互斥问题需应用自身解决。

(3) 单处理器多任务系统

对于一个较复杂的嵌入式系统来说，当采用中断处理程序加一个后台主程序这种软件结构难以实时、准确、可靠地完成时，或存在一些互不相关的过程需要在一个计算机中同时处理时，就需要采用多任务(multitasking)系统。它对于降低系统的复杂性，保证系统的实时性和可维护性是必不可少的。

嵌入式多任务系统的实现必须有嵌入式多任务操作系统的支持。操作系统主要完成任务切换，任务调度，通信、同步、互斥，实时时钟管理和中断管理等。

多任务系统实际上是由多个任务、多个中断处理过程和嵌入式操作系统组成的有机整体，如图 1－12 所示。在多任务系统中每个任务是顺序执行的，并行性通过操作系统来完成，任务间以及任务与中断处理程序间的通信、同步和互斥也需要操作系统的支持。

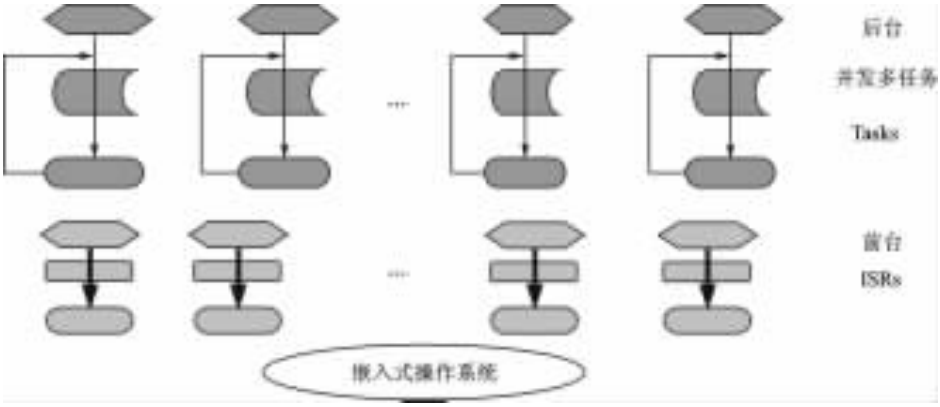


图 1－12　多任务系统程序流程

单处理器多任务系统目前已广泛应用于 32 位嵌入式系统。以下是其主要特点和优缺点。

特点：

- 多个顺序执行的任务并行运行；
- 宏观上看，所有的任务同时运行，每个任务运行在自己独立的 CPU 上；
- 实际上，不同的任务是共享同一个 CPU 和其他硬件的，因此需要嵌入式操作系统来对这些共享的设备和数据进行管理；
- 每个任务一般被编制成无限循环的程序，等待特定的输入，执行相应的处理；
- 这种程序模型将系统分成相对简单的、相互合作的模块。

优点：

- 将复杂的系统分解成相对独立的多个任务，达到分而治之的目的，从而降低系统的复杂性；
- 保证系统的实时性；
- 系统的模块化好，可维护性高。

缺点：

- 需要采用一些新的软件设计方法；
- 需要对每一个共享资源进行互斥；
- 导致任务间的竞争；
- 需要使用嵌入式操作系统，增加系统的开销。

(4) 多处理器多任务系统

当有些工作用单处理器来处理难以完成时，就需要增加另外的处理器。这就是多处理器系统的由来。在单处理器系统中，多个任务在宏观上看是并发的，但在微观上看实际是顺序执行的。在多处理器系统中，多个任务可以分别在不同的处理器上执行，宏观上看是并发的，微观上看也是并发的；前者称为伪并发性，后者称为真并发性。

多处理器系统或称并行处理器系统可分为单指令多数据流(SIMD)系统和多指令多数据流(MIMD)系统，嵌入式系统大都是 MIMD 系统。MIMD 系统又可分为紧耦合系统(tightly-coupled system)和松耦合系统(loosely-coupled system)两种。紧耦合系统是多个处理器通过共享内存空间来交换信息的系统，而在松耦合系统中多个处理器是通过通信线路来连接和交换信息的。

1.2 嵌入式系统的应用领域

嵌入式系统广泛地应用于消费电子、通信、汽车、国防、航空航天、工业控制、仪表和办公自动化等领域。嵌入式系统市场广阔，现在带来的工业年产值已超过了 10 000 亿美元。形式多样的嵌入式系统正努力把 Internet 连接到人们生活的各个角落，其消费量将以亿计，嵌入式系统应用正逐步形成一个充满商机的巨大产业。据预测，未来 10 年将有 90 %以上的微处理器和 65 %以上的软件应用于各种嵌入式系统中。

1. 消费电子领域

随着技术的发展，消费电子产品正向数字化和网络化方向发展。嵌入式计算机技术与各种电子技术紧密结合，渗入到各种消费电子产品中，涌现出各种新型的消费电子产品，提高了消费电子产品的性能和功能，使其简单易用，价格低廉，维护简便。

高清晰度数字电视将代替传统的模拟电视，数码相机将代替传统的胶片相机，固定电话今后会被 IP 电话所替代，各种家用电器(电视机、冰箱、微波炉、电话等)将通过家庭通信控制中

心与 Internet 连接，实现远程控制、信息交互、网上娱乐、远程医疗和远程教育等，从而转变为智能网络家电。

从手机的发展来看，随着移动通信技术的发展，移动通信系统将逐渐由提供话音为主的服务发展为以提供数据为主的服务。随着通信网络传输速率的提高，包括多媒体、彩色动画和移动商务等在内的新的无线应用也将逐渐涌现出来，使得以提供话音为主的传统手机逐渐发展成为融合了 PDA、电子商务和娱乐等特性的智能手机，如图 1－13 所示。

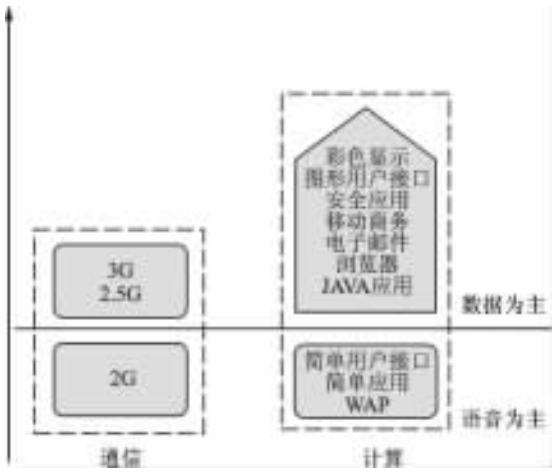


图 1－13　手机的发展

从全球市场来看，IDC 预测 2005 年手机的供货数量将达到 67 200 万部的规模。从国内市场来看，全国移动电话普及率已超过 20 %，与国外发达国家与地区超过 50 %的移动电话普及率相比，我国移动通信用户数量增加的潜力十分巨大。

2. 通信领域

通信领域大量应用嵌入式系统，主要包括程控交换机、路由器、IP 交换机和传输设备等。据预测，由于互联的需要，特别是宽带网络的发展，将会出现各种网络设备，如 xDSL Modem/Router 等，其数量将远远高于传统的网络设备。它们基于 32 位的嵌入式系统，价格低廉，将为企业和家庭提供更为廉价、方便、多样的网络方案。就宽带上网的网络设备 ADSL Router 而言，国外现在每月需要 60 万套的数量。

3. 工业控制、汽车电子、医疗仪器等领域

随着工业、汽车、医疗卫生等各部门对智能控制需求的不断增长，需要对设备进行智能化、数字化改造，这为嵌入式系统提供了很大的市场。

在工业控制领域，嵌入式系统主要应用在各种智能测量仪表、数控装置、可编程控制器、控制机、分布式控制系统、现场总线仪表及控制系统、工业机器人、机电一体化设备等系统中。

就汽车电子系统而言，目前大多数高档轿车每辆拥有约 50 个嵌入式微处理器。如

BMW 7系列轿车，平均安装有 63 个嵌入式微处理器，车前的大灯和车后的尾灯都是用微处理器控制的。智能化的侧视镜与光学传输系统相连接，可以指向车的下面。驾车人在倒车时，可以从车内看清车下的情况。车内的音响与传感器和控制器相连，可以根据具体情况自动调节音响的音量，使得输出电平适量地超出车内环境噪声的电平。如果出现事故，安全气囊的传感器除了给气囊自动充气以防止撞伤外，还同时向 GPS 服务站登记报案，并且还将有关数据传送给手机，让手机自动向警察报警，以便交通警察及时知道在哪里出了事故。

据预测，21 世纪初美国接入 Internet 的汽车将有 1 亿辆。IC Insights 报道 2001 年车载计算系统的市场规模是 30 亿美元，2004 年可达到 46 亿美元。

在医疗仪器领域，嵌入式系统主要应用在各种医疗电子仪器中，如 X 光机、超声诊断仪、计算机断层成像系统、心脏起搏器、监护仪、辅助诊断系统和专家系统等。

4. 国防、航空航天领域

嵌入式系统最早应用在军事和航空航天领域，目前主要应用在各种武器控制系统（火炮控制、导弹控制、智能炸弹制导引爆装置），坦克、舰艇、轰炸机等陆海空军用电子装备，雷达、电子对抗军事通信装备，各种野战指挥作战专用设备等系统中。

1.3 嵌入式系统的实时性与可靠性

嵌入式实时系统的典型应用领域是工业控制和军用武器装备的控制。它通常是强实时系统，强调的是实时性和可靠性。本节主要说明嵌入式实时系统（以下也称为实时系统）中可靠性和实时性的若干基本概念。

1.3.1 嵌入式系统的可靠性

1. 嵌入式系统硬件的可靠性

无论是嵌入式系统的硬件还是通用计算机系统的硬件，所采用的可靠性技术基本上是相同的，只是当嵌入式系统用于一些高可靠性要求的环境时，在硬件的设计上有一些特殊的要求。

硬件系统的可靠性技术包括可靠性保证技术和可靠性评价技术两个方面。

（1）硬件可靠性的一般原理

- 硬件可靠性保证技术；
- 硬件可靠性评价技术。

（2）高可靠性硬件系统的特殊设计

- 热设计；
- 抗振动及抗冲击设计；
- 电磁兼容性设计；

- 防电磁泄漏设计；
- 冗余容错设计；
- 检错及纠错。

2. 嵌入式系统软件的可靠性

现代软件可靠性的研究如硬件系统一样，主要包含两个方面：软件可靠性保证技术和软件可靠性评价技术。它们分别针对以下两个核心问题：

① 如何开发可靠的软件；

② 如何检验软件是否满足可靠性要求。

这里首先对影响软件可靠性的主要因素作一个简要描述。

(1) 影响软件可靠性的主要因素

由于软件的生产过程都是在人的控制或干预下进行的，故单纯从技术的角度来看，影响软件可靠性的主要因素有以下方面：

- 软件规模　软件规模的大小对软件可靠性的影响是很显然的。一个仅有几条或几十条指令的软件，便没有必要讨论其可靠性。根据 Halstead 理论，软件中的初始错误数是与程序容量成正比的。
- 软件内部结构　内部结构越复杂，软件复杂度越高，内含的错误就越多，其可靠性就越低。可采用 McCabe 度量方法对软件复杂度进行度量。
- 软件运行环境　同一软件在不同的运行环境下发生错误的可能性是不同的，即可靠性不同。比如，假设某一软件的数据输入域为 I，并设 $I_1 \mathrm{Y} I_2 = I$ 且 $I_1 \mathrm{I} I_2 = \phi$，软件的输出(结果)为输入的函数 $f(I)$，则一种可能的运行情况如图 1-14 所示。因此，对某一运行环境，函数 $f(I)$ 输出正确结果的概率正比于输入数据落在域 I_1 内的概率。

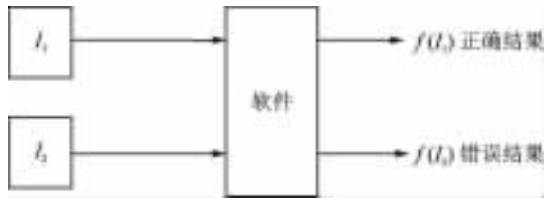


图 1-14　不同输入域下软件的运行情况

软件的一组可能输入构成输入空间的一个点 P，这些 P 点的全体为输入空间 S。在 S 上，定义一个概率密度函数 $f(P)$。

设 G 是 S 的一个子点集，则输入点 P 落在子点集 G 内的概率为 $P = \int G f(P) \mathrm{d}P$。其中，$\{S, f(P)\}$ 称为软件的“运行剖面”。

软件使用方必须明确软件的运行剖面。这里既包括软件在正常情况下的输入，也

包括可能的非正常输入。软件的需求包括在正常输入下应该得到的正常输出，也包括在可能的非正常输入下应该有的正确反应，例如空间卫星上用的计算机软件。

- 软件可靠性设计技术　包括能改善软件可靠性的一切技术，如失效模式与效应分析(FMEA)、故障树分析(FTA)等。
- 软件测试　软件测试是提高软件可靠性的一种十分有效的途径，测试过程也是错误排除过程。
- 软件开发方法　软件工程旨在提高开发效率和软件质量。软件工程的一个重要内容是软件开发方法，合理且先进的软件开发方法有助于保证软件的质量，因而对软件可靠性的实现至关重要。
- 软件可靠性管理　包括软件生产-维护过程中每一阶段的系统化、规范化、组织化和现代化。
- 软件开发人员的能力和经验　软件人员的能力越强，经验越丰富，软件开发过程产生的错误就越少。
- 实时性　对于实时系统，系统的正确性不仅取决于计算结果的正确性，还取决于产生结果的时间。

实际上，嵌入式软件与通用计算机系统软件的可靠性问题，除了实时性要求外，在很大程度上是相同的。

(2) 嵌入式软件可靠性保证技术

从保证软件可靠性的角度，通用软件所采用的措施如下：

- 进行充分详细的需求分析；
- 采用合理的软件设计方法；
- 设计容错软件；
- 有效的可靠性管理；
- 可靠性测试和可靠性评估等。

这些措施都可用于保证嵌入式软件的可靠性。但针对嵌入式软件的特殊性，应特别注意以下几个方面的可靠性措施。

1) 采用行业标准的嵌入式操作系统

目前各行业都在定义嵌入式操作系统的行业标准，选择本行业的嵌入式操作系统，将有利于软件的升级换代，提高可靠性、可重用性和可移植性。应用系统在操作系统之上开发，与系统硬件可以没有直接关系，硬件产品的升级换代对应用系统不会产生直接影响。

2) 可靠性测试

软件的可靠性测试是提高软件可靠性的重要手段。嵌入式软件的测试方法与通用计算机软件测试有许多共同之处。软件测试的方法很多，一般分为白盒测试和黑盒测试。在测试的进程上，一般采用的策略是渐进式测试，即单元测试(模块测试或任务测试)、联合测试(集成测

试)、确认测试和系统测试等几个阶段。软件可靠性测试的三个关键环节如下：

① 根据用户使用软件的方式，构成软件运行剖面，生成测试案例；

② 开发软件可靠性测试的环境，使被测软件能在该环境中得以测试；

③ 对测试结果进行分析，并作出软件可靠性的预计。

与通用计算机软件测试相比，嵌入式软件的测试也有其特殊性。由于其硬件平台的特定性和与专用外部设备的连接，使得嵌入式软件在相应的目标机系统未交付之前不能真正运行，造成动态测试、覆盖分析等方法的使用受到一定的限制。除此以外，嵌入式软件的输入/输出涉及计算机系统专用的端口、外部设备以及各种不同的信号形式，再加上实时性对输入/输出时序特性的要求，使嵌入式软件测试输入与结果的获得更为困难。它们不能通过简单的方式进行描述和控制，给测试数据注入、测试序列执行带来极大的不便。因此，嵌入式软件的测试较为困难。

3) 容错设计

为了提高一些关键嵌入式软件的可靠性，采用容错设计是目前的一项重要手段。现有的各种容错软件技术归纳起来主要有以下两方面：

- 恢复块技术　使用“自动后向错误恢复”的错误处理技术。其方法是使用主程序和若干个候选程序以及接收测试程序。当一个完整功能的程序段执行结束之后，进行验收测试；如果该程序段的执行结果没有通过验收，则系统执行其候选版本来进行错误恢复。
- N 版本程序设计　依据相同的规范要求，独立地设计 $N(N>2)$ 个功能等价的程序。这些程序称为版本。N 个版本同时运行，最终结果为 N 个版本共同表决的结果。

显然，容错软件设计是以牺牲系统的时间和空间为代价来换取可靠性的，因此在时间和空间资源非常有限的嵌入式系统中，该方法的使用受到一定限制。

4) 实时性设计

实时系统的重要特征是时间性，即实时性，而系统的实时性主要由系统中的软件(包括系统软件和应用软件)来保证。因此在实时软件的可靠性设计中，必须将实时性一并考虑，以满足系统响应时间的要求。这一问题将在 9.3 节作更详细的说明。

(3) 嵌入式实时软件可靠性评价技术

随着人们对软件可靠性工程研究的深入，软件可靠性评价技术成为了一项不可缺少的重要研究内容。因为离开了准确的评价方法，可靠性保证措施的有效性将难以判断。目前在软件可靠性评价技术中，主要有两种评价方法：

- 基于软件可靠性测试的方法　软件可靠性测试的目的是对软件需求规范中所规定的软件可靠性目标作出定量的回答，即为了达到或验证用户对软件的可靠性要求进行的测试。这种测试是在软件确认阶段进行，并且往往是在用户参与的情况下实施。同时，在测试过程中对软件不进行故障剔除。根据测试结果(收集的数据)给出可靠性的定量估计值，以便从可靠性角度判断是否接收该软件。这一方法的典型代表是 Nelson

可靠性验收模型。

- 基于软件可靠性建模的方法　在软件测试阶段，对测试过程中收集到的软件可靠性（故障）数据进行建模，以估计软件可靠性的实际水平，从而从可靠性角度判断何时停止软件测试，交付用户验收。在软件测试阶段被发现的软件错误不断被剔除，软件可靠性呈增长趋势。因此，这一方法称为软件可靠性增长建模。迄今为止，这一建模方式是软件可靠性建模的主要内容。

目前软件可靠性评价技术的研究大都针对通用计算机软件，而嵌入式实时软件的相关研究还有待进一步深入。虽然两者在评价方法上有相当一部分是相同的，但实时软件的可靠性评价还有自身的特殊性。嵌入式实时软件与通用计算机软件的最重要的差异是对时间资源的管理不同。

1）时间要求

如果嵌入式实时软件的执行超过了规定的截止时间，其造成的后果（损失）可能与由于软件代码故障造成的后果（损失）是相同的。

因此，对嵌入式实时软件的时限可满足性进行评价，与对代码故障所进行的可靠性评价具有同等重要的意义。

一个任务 A_i 在时刻 t_s 由于事件 e_i 到达系统被激活，到时刻 t_f，A_i 执行完成，而规定的完成时间为 D_i，如果 $t_f-t_s>D_i$，则称任务 A_i 发生了一次超时故障。

将任务不能满足规定的时限要求定义为超时故障，从而将其纳入实时软件可靠性评价的范畴。

2）时间基准

对于嵌入式软件，时间基准问题可分两种情况来讨论。

第一种情况　系统一旦启动后，软件一刻不停地运行，比如卫星或某种全天候的监控系统内的软件。在这种情况下，日历时间与软件的执行时间是一致的。

第二种情况　软件运行时间完全取决于系统的启动情况，比如战斗机执行一次飞行任务。在这种情况下，软件执行时间与日历时间有很大的差异。假设一架战斗机的使用寿命为15年，但其软件运行时间（即战斗机实际飞行时间）却要小得多，比如小于 2 500 h，显然这时用日历时间作为时间基准是没有意义的。

上述两种情况是指在软件实际投入运行后的情况。在软件的测试阶段，为便于测试和故障数据收集等活动，可以采用日历时间作为时间基准；但在软件投入运行后，应以执行时间为基准，为此，需要将日历时间转换为执行时间。

1.3.2　嵌入式系统的实时性

实时系统是对外来事件在限定时间内能作出反应的系统。从揭示“时间”的重要性的角度出发，实时系统可以描述如下：

在实时系统中，时间是一种重要的系统资源，对外部事件的响应和任务的执行都必须在限定的时间内完成。在多机系统中，还必须在限定的时间内完成消息的发送和接收。实时系统中，输出结果的正确性不仅取决于计算所形成的逻辑结果，还要取决于结果产生的时间。

除了系统的可靠性以外，嵌入式实时系统的使用者关心的另一个重要因素是系统对外部事件响应的实时性，确切地讲是系统能否在规定的时间内对外部的某一个事件予以响应。假设某一事件发生在时刻 T_0，期望系统的最迟响应时间为 T_{DL}，如果外部事件采用中断方式通知系统，则系统从中断响应开始，之后进行任务调度，使处理该事件的任务投入运行（即占有处理器资源），到该任务执行完成产生相应的输出结果，整个处理时间必须小于 $T_{DL}-T_0$。

为了满足上述系统时间的要求，系统设计者必须仔细分析影响系统响应时间的所有因素。对于实时多任务系统，通常都会有很多事件同时产生。一个任务描述一个事件，整个系统的并发机制是通过让很多任务并行运行而实现的。因此在系统内部，以满足系统响应时间为目标，必须保证每个任务都能够在规定的时间（deadline——截止时间）内完成。

1. 影响系统响应时间的主要因素

在实时系统中，系统对外部的响应有两种基本模式：

① 系统响应过程仅需要由中断处理来完成，此时，系统响应时间即为中断响应时间。

② 系统响应过程由两部分构成，即“中断响应＋任务执行”。这种模式通常用于处理一些较为复杂的工作。

假设系统中有两个外部事件，即事件 1 和事件 2，系统对这两个事件有不同的响应模式，其响应过程如图 1－15 所示。

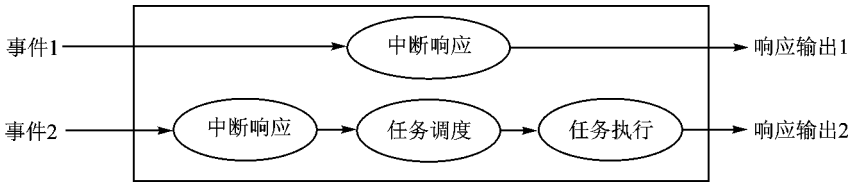


图 1－15　系统对外部事件的响应过程

显然，仅由中断处理过程完成系统响应是一个比较简单的过程。下面对影响系统响应时间的主要因素的分析针对的是“中断响应＋任务执行”这种模式。

（1）任务调度算法

在多任务环境中，任务调度的作用是以某种调度算法，在调度时刻从多个任务中选择其中一个任务投入运行。对于不同的应用，实时性要求是不同的，比如有的系统要求满足尽可能多的任务的时限要求，而有的系统则希望优先满足几个最紧急任务的时限要求。仅对这两种情况，优先级的确定方式就相差甚远。因此存在若干种调度算法，不同调度算法对任务优先级的确定方式不同。显然，不同的调度算法将使系统有不同的响应时间，或者

说影响系统响应时间。

(2) 任务执行时间

任务执行时间是影响系统响应时间的一个重要因素。在通常的分析中,仅假设任务 A_i 的执行时间为一个确定的值 C_i;但在实际情况下,C_i 常常是不确定的。比如,一个任务的工作流程如图 1-16 所示。

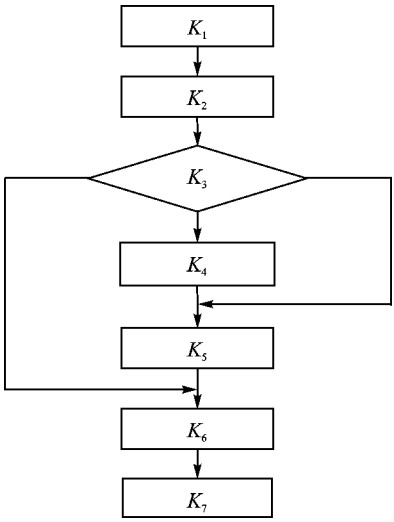


图 1-16 任务的工作流程

假设该任务的输入数据域 D_f 可以划分为三个明确的子域 D_{f1},D_{f2} 和 D_{f3},且 $D_{f1} \mathrm{I} D_{f2} \mathrm{I} D_{f3} = \phi$。对于任意一个输入数据 d,如果 $d \subset D_{f1}$,则任务执行流程是 $K_1 \rightarrow K_2 \rightarrow K_3 \rightarrow K_6 \rightarrow K_7$,所需要的时间为 C_1;如果 $d \subset D_{f2}$,则任务执行流程为 $K_1 \rightarrow K_2 \rightarrow K_3 \rightarrow K_5 \rightarrow K_6 \rightarrow K_7$,所需时间为 C_2;如果 $d \subset D_{f3}$,则任务执行流程为 $K_1 \rightarrow K_2 \rightarrow K_3 \rightarrow K_4 \rightarrow K_5 \rightarrow K_6 \rightarrow K_7$,所需时间为 C_3。显然,$C_1 < C_2 < C_3$。所以仅仅假设任务执行时间为一个确定的值是不够的,还需要考虑最大执行时间和平均执行时间。

由以上的分析可以看出,任务执行时间常常是可变的,在分析最坏情况下的系统响应时间时,可以采用任务的最大执行时间。比如在上例中,C_3 是任务的最大执行时间。

在实际情况中,发生超时故障的概率会低于采用最大执行时间时所得到的超时故障发生率。一般地,可采用任务平均执行时间来分析任务超时故障的平均发生率。对于选定的具有代表性的 n 个输入数据,分别得到 n 个执行时间 $C_1, C_2, \cdots, C_n$,则任务平均执行时间为 $C_{avg} = \sum C_i / n (i = 1, 2, \cdots, n)$。由于不同输入数据的使用率是不同的,因此在计算任务平均执行时间时,应以输入数据的使用率作为 C_i 的权值系数。

(3) 事件发生的频率

在实时系统中,从事件发生的时间上看,分为周期性事件(可采用时钟定时)和非周期性事件。事件发生的频率越低,系统响应时间越容易得到满足;反之,事件发生的频率越高,系统响应时间越难得到满足。

在进行时间特性分析时,对于周期性事件(比如在巡检系统中,以时钟方式周期性地启动任务),分析系统响应时间相对较容易;但对于非周期性事件的时间特性分析则比较复杂。

(4) 任务数量

任务数量是影响任务能否满足时限要求的另一因素。随着系统中任务数量的增加,处理器时间利用率逐渐提高。由于处理器时间资源是有限的,显然当处理器时间利用率等于 1 时,再增加任何一个任务,都至少会使系统中有一个任务的截止时间得不到满足。

(5) 中断响应时间

前面说明了任务数量、事件发生的频率、任务执行时间和任务调度算法对系统响应时间的影响,所有这些都是与任务相关的参数。而系统的其他时间开销也将影响任务时限的可满足性,中断响应时间便是其中的一种。

如采用抢占调度,在一个任务的运行过程中,由于某一事件的发生而产生中断,使当前运行任务暂时中止,系统进行中断处理。中断处理完成后,系统根据当前运行任务和中断所对应的任务的优先级高低,决定是继续运行当前任务还是切换到新任务执行。

由于中断延缓了任务的执行,因此在进行任务实时性能分析时,可将中断响应时间加到任务执行时间中。一般来说,有

中断响应时间=中断延迟时间+中断处理时间

(6) 任务响应时间

任务响应时间是指当外部事件到达系统至系统转入对该事件作相应处理(对应该事件的任务开始执行)的这段时间。首先假设实时系统采用的是抢占调度,这种算法的一个重要特征是,正在运行的任务一般不会去动用关中断来封锁中断机制;只有一种情况例外,即它在进行现场环境的调整时。它要动用关中断来封锁中断机制的目的是,防止新产生的中断搅乱原有的现场调整过程。因此在考虑中断响应的情况下,有

任务响应时间=中断响应时间+任务调度时间

(7) 资源共享

由于多个任务的存在,系统中的资源共享使得任务之间存在制约关系。这主要表现为,一个任务可能需要等待其他任务释放某种资源才能运行。资源的使用是互斥的,通常采用信号量方式来实现各任务之间的互斥。从时间上看,需等待资源的任务的运行被推延了一段时间,但是被推延的时间长度是未知的。比如,任务 A_1 和 A_2 共享一个资源,由信号量 S 实现互斥。当任务 A_1 试图使用该共享资源而对信号量 S 进行操作(如 P 操作)时,发现资源已被占用,A_1 进入等待状态。由于 A_1 只知道资源被占用,但无法知道它将被占用多长时间,因此 A_1 需要等待多长时间是不确定的。故系统资源共享也将影响系统响应时间。

(8) 任务间的通信

任务之间采用什么样的通信机制对系统响应时间也有影响。比如传统的生产者-消费者问题中的信号量机制,其通信效率就比较低;而采用共享存储器方式则有较高的通信效率。

前面指出了影响系统响应时间的主要因素。其中,系统中任务的划分方法(关系到任务数量和任务执行时间)、任务中所使用的算法(关系到任务执行时间)和任务调度算法起着最为重要的作用,而且都直接与应用设计相关;而中断响应时间、任务切换时间、资源共享及任务间通信机制等则主要与操作系统有关,一旦硬件平台和操作系统选定之后,这些因素也就确定了,其相关的时间参数为定值。因此,从系统的角度,为了保证系统响应时间要求,应用设计者应主要考虑任务的划分方法、任务中所使用的算法和任务调度算法(操作系统一般提供了若干种

任务调度算法,选择何种算法是与具体应用有关的)。

2. 实时性保证与评价

(1) 通过合理设计来保证实时性

系统实时性保证涉及硬件平台、系统软件(如嵌入式实时操作系统)和应用软件等多方面因素。在硬件平台和系统软件选定之后,实时性主要取决于应用软件的设计。任务的划分方法、任务中所使用的算法和任务调度算法的选择对实时性能的影响十分重要。

(2) 通过测试和建模来评价实时性

对于较为复杂的系统,采用实时多任务的方式进行设计。在系统的设计完成之后,应该对系统的实时性能进行评估和确认。对于实时多任务系统,当系统需要完成某一项工作时,需要启动(执行)一系列任务,因此如果能确保这一系列任务在规定的截止时间内完成,则能够保证系统的响应时间。因此在进行实时性能评价时,需要确认每一个任务的完成时间,并最后确认系统响应时间。

评价方法主要有两种:

① 通过模拟测试(即通过仿真的方式模拟多任务的运行),测试各任务是否能够在规定的时间内完成以及在最坏情况下任务的完成时间,并最终测试系统响应时间。

② 根据影响系统实时性的参数以及这些参数之间的关系建立性能评价模型。比如根据系统的中断响应时间、任务切换时间、资源共享、任务间通信方式、任务数量、任务执行时间和任务调度算法等,建立起性能评价模型。

在上述两种实时性能评价方法中,前者需要建立相应的仿真运行环境(包括软件环境和硬件环境),模拟多任务的运行。有些时候这种环境的建立比较困难,特别是嵌入式实时系统往往需要有相应的激励设备来仿真各种外部设备。后者则是一项理论性很强的工作,当考虑到了系统中的各个方面时,其评价模型可能非常复杂。所以,评价模型通常用于系统实时性的初期评估,忽略一些次要因素(参数)来建立评价模型,评价在最坏情况下,系统对响应时间的满足性。

1.4 嵌入式系统的发展趋势

以信息家电、移动终端、汽车电子和网络设备等为代表的互联网时代的嵌入式系统,不仅为嵌入式市场展现了美好前景,注入了新的生命,同时也对嵌入式系统技术提出新的挑战。这主要包括:支持日趋增长的功能密度、灵活的网络联接、轻便的移动应用、多媒体的信息处理、低功耗、人机界面友好互动、支持二次开发和动态升级等。

面对这些需求,嵌入式系统的主要发展趋势是有以下方面。

1. 形成行业的标准——行业性嵌入式软硬件平台

嵌入式系统是以应用为中心的系统,不会像 PC 一样只有一种平台;但它会吸取 PC 的成功经验,形成不同行业的标准。统一的行业标准具有开放、设计技术共享、软硬件重用、构件兼

容、维护方便和合作生产等特点，是增强行业性产品竞争能力的有效手段。

在工业控制等领域，嵌入式 PC 已成为一种标准的软硬件平台。它在硬件上兼容 PC，以 ISA，CPCI 为标准总线，并扩展 DOC(Disk On Chip)，DOM(Disk On Module)和 Flash 等多种存储方式。其软件以 BIOS 为基础，可运行多种嵌入式操作系统。

近几年，一些地区和国家的若干行业协会纷纷制定嵌入式系统标准，如欧共体汽车产业联盟规定以 OSEK(OSEK 的名称来自于德文"车内电子设备的开放系统的接口")标准作为开发汽车嵌入式系统的公用平台和应用编程接口。航空电子工程协会 AEEC(Airlines Electronics Engineering Committee)制定了航空电子的嵌入式实时操作系统应用编程接口 ARINC 653。我国数字电视产业联盟也在制定本行业的开放式软件标准，以提高中国数字电视产品的竞争能力。看来，走行业开放系统道路、建立行业性的嵌入式软硬件开发平台，是加快嵌入式系统发展的捷径之一。根据应用的不同要求，今后不同行业会定义其嵌入式操作系统、嵌入式支撑软件和嵌入式硬件平台等行业标准。

2. SOC 将成为应用主流

由于嵌入式系统应用领域的广泛性、不断增加的复杂性、联网的需要以及半导体技术的发展，IC 厂家可根据应用的需要，开发出面向应用领域的、高度集成的、以 32 位嵌入式微处理器为核心的 SOC(System On Chip)，以便嵌入式系统的开发。32 位嵌入式系统将成为主流。

随着 EDA 的推广、VLSI 设计的普及化及半导体工艺的迅速发展，在一个硅片上实现一个更为复杂的系统的时代已来临，这就是 SOC。除 8 位/16 位处理器核外，各种 32 位 RISC 通用处理器内核将作为 SOC 设计公司的标准库，和许多其他嵌入式系统外设一样，成为 VLSI 设计中的标准器件，用标准的 VHDL 等语言描述并存储在器件库中。用户只需定义出其整个应用系统，仿真通过后就可以将设计图交给半导体工厂制作样品。这样除个别无法集成的器件以外，整个嵌入式系统大部分均可集成到一块或几块芯片中去，应用系统电路板将变得很简洁。这对于减小体积和功耗、提高可靠性非常有利。

3. 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

为了满足应用的需求，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。

目前，国外商品化的嵌入式实时操作系统已进入我国市场的有 WindRiver，Microsoft，QNX 和 Nucleus 等公司的产品。我国自主开发的嵌入式系统软件产品有北京科银京成(CoreTek)公司的嵌入式软件开发平台"道系统"(DeltaSystem)和中国科学院推出的 Hopen 嵌入式操作系统等。其中"道系统"不仅包括 DeltaOS 嵌入式实时操作系统，而且还包括 LambdaTool 系列开发工具套件、GammaRay 系列测试工具及各种嵌入式应用组件等。

嵌入式操作系统将在现有的基础上，不断采用先进的操作系统技术，结合嵌入式系统的需求，向可适应不同嵌入式硬件平台并具有可移植、可伸缩、功能强大、可配置、良好的实时性、可

靠性和高可用等方向发展。

嵌入式开发工具将向支持多种硬件平台、覆盖嵌入式软件开发过程各个阶段、高效和高度集成的工具集等方向发展。

4. 嵌入式系统联网成为必然趋势

为适应嵌入式分布处理结构和应用上网需求，面向 21 世纪的嵌入式系统要求配备一种或多种标准的网络通信接口。针对外部联网要求，嵌入式系统必须配有通信接口，需要 TCP/IP 协议簇软件支持；由于家用电器相互关联（如防盗报警、灯光能源控制、影视设备和信息终端交换信息）及工业现场仪器的协调工作等要求，新一代嵌入式系统还需具备 IEEE 1394，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驱动软件；为了支持网络交互的应用，还需内置 XML 浏览器和 Web Server。

5. 嵌入式系统向新的嵌入式计算模型方向发展

- 支持自然的人机交互和互动的、图形化的、多媒体的嵌入式人机界面。操作简便、直观，无须学习，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。
- 可编程的嵌入式系统。嵌入式系统可支持二次开发，如采用嵌入式 JAVA 技术可动态加载和升级软件，增强嵌入式系统功能。
- 支持分布式计算。与其他嵌入式系统和通用计算机系统互联，构成分布式计算环境。

思考题

1.1 什么是嵌入式系统？嵌入式系统与通用计算机系统的异同是什么？

1.2 嵌入式系统的特点是什么？

1.3 按实时性来分，嵌入式系统可分为几类？它们的特点是什么？

1.4 按软件结构来分，嵌入式系统可分为几类？它们的优缺点是什么？分别适用于哪些系统？

1.5 前后台系统的组成和运行模式如何？需要考虑的主要因素有哪些？主要性能指标是什么？

1.6 单处理器多任务系统由哪些部分组成？其运行方式如何？

1.7 嵌入式系统的主要应用领域有哪些？

1.8 影响嵌入式软件可靠性的主要因素有哪些？如何保证嵌入式软件的可靠性？

1.9 影响系统响应时间的主要因素有哪些？

1.10 描述嵌入式系统的发展历程和发展趋势。

第 2 章　嵌入式硬件系统

本章介绍嵌入式硬件系统的组成、嵌入式微处理器的特点、主流的嵌入式微处理器 ARM 系列和 MIPS 系列、AMBA 和 PCI 总线、嵌入式系统存储器结构及输入/输出接口和设备。

2.1　基本组成

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。

1. 嵌入式微处理器

每个嵌入式系统至少包含一个嵌入式微处理器。嵌入式微处理器体系结构可采用冯·诺依曼(Von Neumann)结构或哈佛(Harvard)结构；指令系统可采用精简指令集系统 RISC(Reduced Instruction Set Computer)或复杂指令集系统 CISC(Complex Instruction Set Computer)。它们之间的对比见表 2－1。

表 2－1　CISC 和 RISC 的对比表

类　别	CISC	RISC
价格	由硬件完成部分软件功能，硬件复杂性增加，芯片成本高	由软件完成部分硬件功能，软件复杂性增加，芯片成本低
性能	减少代码尺寸，增加指令的执行周期数	使用流水线降低指令的执行周期数，增加代码尺寸
指令集	大量的混杂型指令集，有简单快速的指令，也有复杂的多周期指令，符合 HLL(High Level Language)	简单的单周期指令，在汇编指令方面有相应的 CISC 微代码指令
高级语言支持	硬件完成	软件完成
寻址模式	复杂的寻址模式，支持内存到内存寻址	简单的寻址模式，仅允许 Load 和 Store 指令存取内存，其他所有的操作都基于寄存器到寄存器
控制单元	微码	直接执行
寄存器数目	寄存器较少	寄存器较多

冯·诺依曼体系结构如图 2－1 所示，哈佛体系结构如图 2－2 所示。

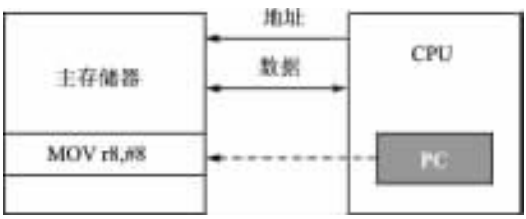


图 2－1　冯·诺依曼体系结构

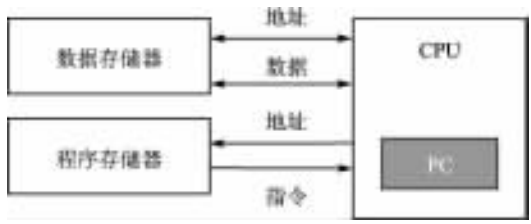


图 2－2　哈佛体系结构

嵌入式微处理器有许多不同的体系，即使在同一体系中也可能具有不同的时钟速度和总线数据宽度，集成不同的外部接口和设备。据不完全统计，目前全世界嵌入式微处理器的品种总量已经超过千种，有几十种嵌入式微处理器体系。主流的体系有 ARM，MIPS，PowerPC，X86 和 SH 等。嵌入式微处理器的选择是由具体的应用所决定的。

2. 总　线

嵌入式系统的总线一般集成在嵌入式微处理器中。从微处理器的角度来看，总线可分为片外总线（如 PCI，ISA 等）和片内总线（如 AMBA，AVALON，OCP 和 WISHBONE 等）。选择总线和选择嵌入式微处理器密切相关，总线的种类随不同的微处理器的结构而不同。

3. 存储器

嵌入式系统的存储器包括主存和外存（又称为辅存）。

大多数嵌入式系统的代码和数据都存储在处理器可直接访问的存储空间即主存中。系统上电后，主存中的代码直接运行。主存储器的特点是速度快，一般采用 ROM，EPROM，Nor Flash，SRAM 和 DRAM 等存储器件。

目前有些嵌入式系统除了主存外，还有外存。外存是处理器不能直接访问的存储器，用来存放各种信息，相对主存而言具有速度慢、价格低和容量大的特点。在嵌入式系统中一般不采用硬盘而采用电子盘作外存。电子盘的主要种类有 DOC（Disk On Chip），NandFlash，CompactFlash，SmartMedia，Memory Stick，MultiMediaCard 和 SD（Secure Digital）卡等。

4. 输入/输出接口和设备

嵌入式系统的大多数输入/输出接口和部分设备已经集成在嵌入式微处理器中。输入/输出接口主要有中断控制器、DMA、串行和并行接口等；设备主要有定时器（timers）、计数器（counters）、看门狗（watchdog timers）、RTC、UARTs、PWM（Pulse Width Modulator）、AD/DA、显示器、键盘和网络等。

2.2　嵌入式微处理器

经过近 20 年的发展，嵌入式微处理器的集成度、主频、位数都得到了提高，如表 2－2 所列。

表 2－2　嵌入式微处理器的发展

项　目	年　代			
	20 世纪 80 年代中后期	20 世纪 90 年代初期	20 世纪 90 年代中后期	21 世纪初期
制作工艺单位：μm	1～0.8	0.8～0.5	0.5～0.35	0.25～0.13
主频/MHz	＜33	＜100	＜200	＜600
晶体管个数	＞500×10³	＞2×10⁶	＞5×10⁶	＞22×10⁶
位数/bit	8/16	8/16/32	8/16/32	8/16/32/64

嵌入式微处理器种类繁多，按位数可分为 4 位、8 位、16 位、32 位和 64 位。16 位以下的嵌入式微处理器一般称为微控制器(MCU)，32 位以上的称为处理器。

按用途来分，嵌入式微处理器可分为嵌入式 DSP 和通用的嵌入式微处理器两种。

① 嵌入式 DSP　专用于数字信号处理，采用哈佛结构，程序和数据分开存储，采用一系列措施保证数字信号的处理速度，如对 FFT(快速傅里叶变换)的专门优化。

② 通用的嵌入式微处理器　一般是集成了通用微处理器的核、总线、外围接口和设备的 SOC 芯片，有些还将 DSP 作为协处理器集成。

本节只对通用的嵌入式微处理器(简称嵌入式微处理器)作详细介绍。

2.2.1　嵌入式微处理器的特点

嵌入式微处理器的基础是通用微处理器。它与通用微处理器相比具有体积小、集成度高(higher integration)、质量轻、成本低、可靠性高、功耗低、工作温度范围宽和抗电磁干扰等特点。

1. 集成度高

用于桌面和服务器的微处理器芯片内部通常只包括 CPU 核心、Cache、MMU 和总线接口等部分，附加的功能如外部接口、系统总线、外部总线和外部设备独立在其他芯片和电路内。

嵌入式微处理器除了集成 CPU 核心、Cache、MMU 和总线等部分外，还集成了各种外部接口和设备，如中断控制器、DMA、定时器和 UART 等。这是符合嵌入式系统的低成本和低功耗要求的。一块单一的、集成了大多数需要的功能块的芯片价格更低，功耗也更低。

嵌入式微处理器是面向应用的，其片内所包含组件的数目和种类是由它的市场定位决定的。在最普通的情况下，嵌入式微处理器包括：

- 片内存储器；
- 外部存储器的控制器、外设接口（串口、并口）；
- LCD 控制器（面向终端类应用的嵌入式微处理器具有）；
- 中断控制器、DMA 控制器和协处理器；
- 定时器，A/D 和 D/A 转换器；
- 多媒体加速器（当高级图形功能需要时具有）；
- 总线；
- 其他标准接口或外设。

集成外围逻辑芯片目前有两种方式：

① 单芯片（single chip）方式　如图 2－3 所示，是用于终端类应用的 Samsung 公司的 44B0X 芯片的内部结构。

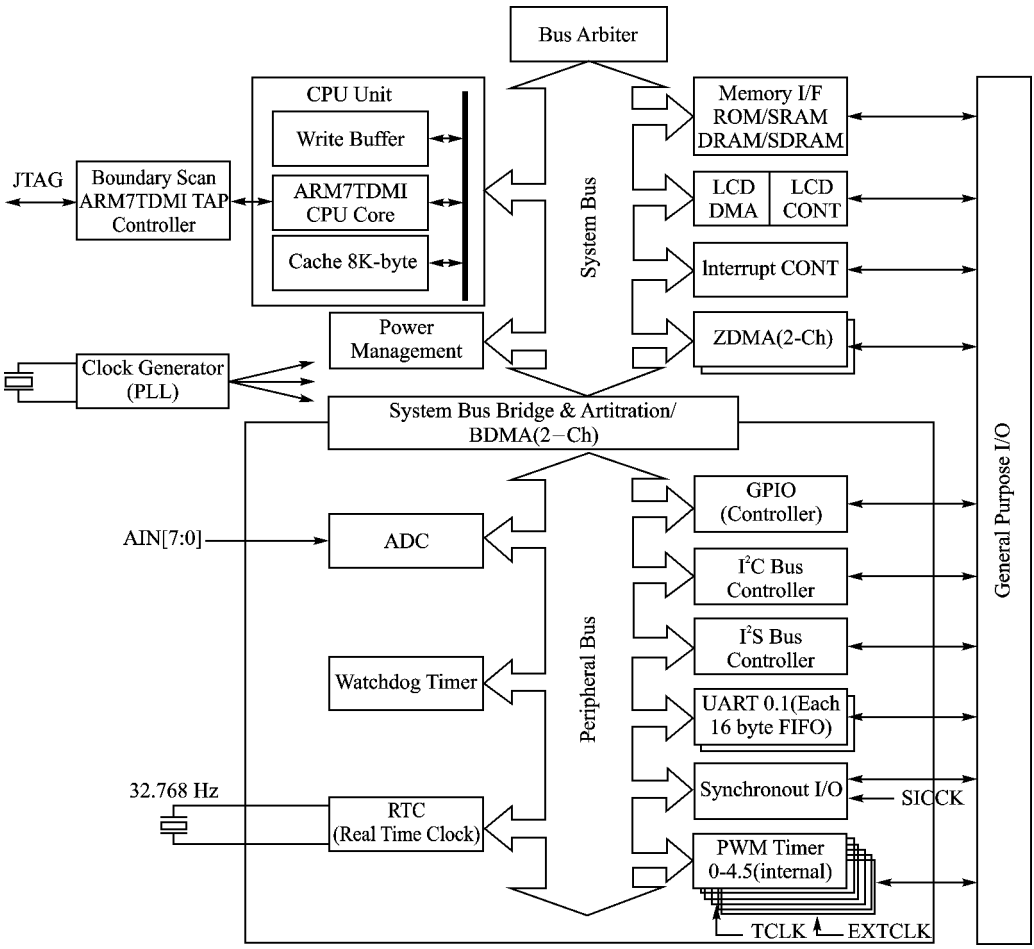


图 2－3　44B0X 芯片的内部结构

② **芯片组**(chip set)**方式**　由微处理器主芯片和一些从芯片组成。图 2－4 为两芯片组的手持 PC 方案，主芯片提供计算和基本外围设备的控制功能，从芯片加入了新的接口(LCD 控制器、红外线接口和触摸屏功能块等)。

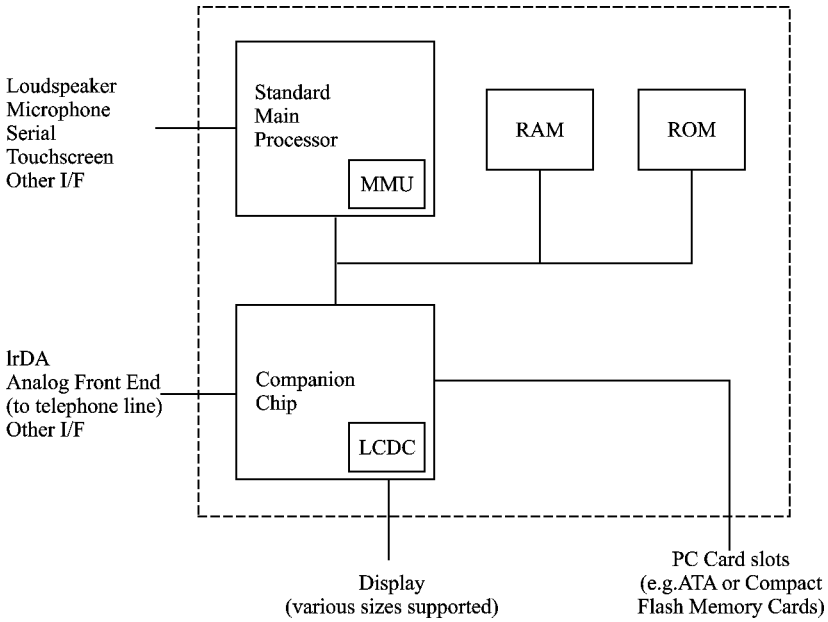


图 2－4　两芯片组的手持 PC 方案

2. 体系结构

(1) **算术格式**

由于低成本和低功耗的限制，大多数的嵌入式微处理器的算术格式(arithmetic format)使用定点算法(fixed－point arithmetic)，即数值被表示为整数或在－1.0～＋1.0 之间的分数。这样的芯片比数值表示为尾数和指数的浮点版本的芯片便宜。

当嵌入式系统中需要使用浮点运算时，可采用软件模拟的方式实现浮点运算，只不过这样要占用更多的处理器时间。

(2) **功能单元**

大多数的嵌入式微处理器包括不止一个的功能单元(functional units)，典型的是包含一个 ALU、移位器和 MAC。处理器通常用一条指令完成乘法操作。

(3) **流水线**

大多数嵌入式微处理器采用单周期执行指令，这样可能导致比较长的流水线(pipeline)；如果产生分支，则要多花几个周期来处理。在通用处理器中，采用分支预测(branch prediction)来减少分支的时间，这种技术非常有效；但是由于大多数嵌入式处理器工作在有实时性要求的系统中，所以没有采用分支预测技术。

此外，为更好地满足并行性，在超标量(superscalar)体系结构通用处理器中经常采用动态调度机制(dynamic scheduling)；但是在嵌入式微处理器中并不采用这种机制，而是采用其他机制提高系统的并行性，具体见第 4 条性能中的说明。

3. 指令集

为满足应用领域的需要，嵌入式微处理器的指令集(instruction set)一般要针对特定领域的应用进行剪裁和扩充。

因为目前很多应用系统需要类似于 DSP 的数字处理功能，故许多嵌入式微处理器扩展了特定领域的指令，如 DSP 指令集。这些指令主要有以下几种：

- **乘加(MAC)操作**　它在一个周期中执行了一次乘法运算和一次加法运算。这种顺序的运算在 DSP 算法中非常常见，比如点积、卷积、相关性和积分。为了能在一个周期内执行这些操作，处理器需集成一个 MAC 单元或添加一个推进路径，使乘法器返回的结果能用于加法器。
- **SIMD 类操作**　允许使用一条指令进行多个并行数据流的计算。
- **零开销的循环指令**　采用硬件方式减少了循环的开销。仅使用两条指令实现一个循环，一条是循环的开始并提供循环次数；另一条是循环体。
- **多媒体加速指令**　像素处理、多边形和 3D 操作等指令。

4. 性　能

嵌入式微处理器的性能(performance)根据应用系统的不同，可分为三类。

(1) **低　端**

一般低端(低价，低性能)嵌入式微处理器的性能最多达到 50 MIPS，应用在对性能要求不高，但对价格和功耗有严格要求的应用系统中。

(2) **中　档**

中档的嵌入式微处理器可达到较好的性能(如 150 MIPS 以上)，采用增加时钟频率、加深流水深度、增加 Cache 及一些额外的功能块的方法来提高性能，并保持低功耗。

(3) **高　端**

高端嵌入式微处理器用于高强度计算，使用不同的方法来达到更高的并行度。

- **单指令执行乘法操作**　通过加入额外的功能单元和扩展指令集，使许多操作能在一个单一的周期内并行执行。
- **每个周期执行多条指令**　桌面和服务器的超标量处理器都支持单周期多条指令执行。在嵌入式领域通常使用 VLIW(Very Large Instruction Word)来实现，这样只需较少的硬件，总体价格会更低些。例如 TI 公司的 TMS320C6201 芯片，通过使用 VLIW 方法，能在每个周期同时执行 8 条独立的 32 位指令。
- **使用多处理器**　采用多处理器的方式满足应用系统的更高要求。一些嵌入式微处理器采用特殊的硬件支持多处理器。如 TI 公司的 OMAP730 包括了三个处理器核，

即 ARM9，ARM7 和 DSP。

5. 功耗和管理

在嵌入式系统中功耗和管理(power consumption and management)是很重要的问题，须仔细考虑。大多数嵌入式系统有功耗的限制(特别是电池供电的系统)，它们不支持使用风扇和其他冷却设备。

嵌入式微处理器采用不同的技术来降低功耗。

- 降低工作电压　1.8 V，1.2 V 甚至更低，而且这个数值一直在下降。
- 提供不同的时钟频率　通过软件设置不同的时钟分频。
- 关闭暂时不使用的功能块　如果某功能块在一个周期内不使用，就可以完全关闭，以节约能量。
- 提供功耗管理机制　具有功耗管理的处理器可以处于如下模式之一：
  - 运行模式(running mode)　处理器处于全速运行状态下；
  - 待命模式(standby mode)　处理器不执行指令，所有存储的信息是可用的，处理器能在几个周期内返回运行模式；
  - 时钟关闭模式(clock－off mode)　时钟完全停止，要退出这个模式，系统需要重新启动。

影响功耗的其他因素还有总线(特别是总线转换器，可以采用特殊的技术使它的功耗最低)和存储器的大小(如果使用 DRAM，则需要不断地刷新)。为了使功耗最低，总线和存储器要保持在应用系统可接受的最小规模。

6. 价　格

价格(cost)是设计嵌入式系统关心的一个问题，特别对于那些量大的系统，价格至关重要。为降低价格，需要在嵌入式微处理器的设计中考虑不同的折衷方案。

处理器的价格受如下因素影响：

- 处理器的特点　功能块的数目、总线类型等。
- 片上存储器的容量大小。
- 芯片的引脚数和封装形式　如塑料四边引出扁平封装 PQFP(Plastic Quad Flat Package)通常就比球栅阵列封装 BGAP(Ball Grid Array Package)便宜。
- 芯片大小(die size)　取决于制造的工艺水平。
- 代码密度(code density)　代码存储器的大小将影响价格，不同种类的处理器有不同的代码密度。CISC 芯片代码密度高，但它的结构复杂，其额外的控制逻辑单元使价格变得很高；RISC 芯片拥有简单的结构，但它的代码密度低，因为其指令集简单；VLIW 芯片代码密度最低，因为它的指令字倾向于采用多字节。此外，还有以下不同的策略用来解决这个问题。
  - 指令长度固定的短指令　如 Hitachi SuperH 的结构，其所有指令都为 16 位。
  - ARM 通过 Thumb 扩展提供一种指令压缩的方法　Thumb 指令集是 32 位 ARM

指令集的一个子集，Thumb 指令集是以 16 位的宽度压缩产生的。在执行以前，通过芯片上的逻辑块，它们被解压为 32 位等同的指令。

- 使用不同的指令宽度　比如 Infineon Carmel 使用 24 位、48 位和 144 位的指令 CLIW(Configurable Long Instruction Word)。当代码大小是主要关心的问题时，它使用 24 位的指令，这对操作数有一些限制。当程序的大部分需要并行执行时，将采用 48 位或 144 位的指令。

2.2.2　主流的嵌入式微处理器

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 系列等。属于这些系列的嵌入式微处理器产品很多，有上千种以上。

1. ARM 系列

ARM(Advanced RISC Machine)公司是一家专门从事芯片 IP 设计与授权业务的英国公司，其产品有 ARM 内核以及各类外围接口。ARM 内核是一种 32 位 RISC 微处理器，具有功耗低、性价比高和代码密度高等三大特色。

目前，70 %的移动电话、大量的游戏机、手持 PC 和机顶盒等都已采用了 ARM 处理器，许多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。

目前 ARM 处理器主要有 5 大系列：ARM7，ARM9，ARM9E，ARM10 和 SecurCore；此外还有与 Intel 公司合作实现的 StrongARM 和 XScale 处理器，性能从 30～1 200 MIPS 不等。

表 2－3 列出了 ARM 各个系列产品的信息。

表 2－3　ARM 各个系列产品信息

系列类别	相应产品	性能特点
ARM7 系列	ARM7TDMI，ARM7TDMI－S，ARM720T，ARM7EJ	三级流水。 性能：0.9 MIPS/MHz
ARM9 系列	ARM9TDMI，ARM920T，ARM922T，ARM940T	五级流水，硬核(hard IP core)。 性能：1.1 MIPS/MHz
ARM9E 系列	ARM966E－S，ARM946E－S，ARM926EJ－S	五级流水，软核(soft IP core)，支持浮点操作。 性能：1.1 MIPS/MHz
ARM10 系列	ARM1022E，ARM1020E	6 级流水，硬核，支持高性能浮点操作，双 64 位总线接口，内部 64 位数据通路。 性能：1.25 MIPS/MHz

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，同时具有以下特点：

- 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；
- 自动递增和自动递减的寻址模式，以优化程序中的循环；
- 同时执行 Load 和 Store 多条指令，以增加数据吞吐量；
- 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。

(1) 数据类型

- 字节(byte)型数据　数据宽度为 8 bits；
- 半字(halfword)数据类型　数据宽度为 16 bits，存取时必须以 2 字节对齐的方式。
- 字(word)数据类型　数据宽度为 32 bits，存取时必须以 4 字节对齐的方式。

(2) 运行模式

ARM 处理器有 7 种运行模式，见表 2－4。大多数应用程序在 User 模式下执行；当特定的异常出现时，进入相应的 6 种异常模式之一。每种模式都有某些附加的寄存器保存相应的状态，以避免异常出现时 User 模式的状态不可靠。除 User 模式外，其他模式都被称为特权模式，可以存取系统中的任何资源。User 模式下程序不能访问有些受保护的资源，也不能直接改变 CPU 的模式，而只能通过异常的形式来改变 CPU 的当前运行模式。软件可以控制 CPU 模式的转变；异常和外部中断也可以引起模式的改变。

表 2－4　ARM 处理器的 7 种运行模式

处理器模式	说　明
用户模式(User)	正常程序执行模式，用于应用程序
异常模式(FIQ)	快速中断处理，用于支持高速数据传送或通道处理
异常模式(IRQ)	用于一般中断处理
异常模式(Supervisor)	特权模式，用于操作系统
异常模式(Abort)	存储器保护异常处理
异常模式(Undefined)	未定义指令异常处理
系统模式(System)	运行特权操作系统任务(ARM V4 以上版本)

(3) 寄存器结构

程序员可见的 ARM 的寄存器共 37 个，如表 2－5 所列。

表 2-5 ARM 的寄存器

User	System	Supervisor	Abort	Undefined	IRQ	FIQ
R0～R7	R0～R7	R0～R7	R0～R7	R0～R7	R0～R7	R0～R7
R8～R12	R8～R12	R8～R12	R8～R12	R8～R12	R8～R12	R8_fiq～R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR _abt	SPSR _und	SPSR _irq	SPSR _fiq

1) R0～R15

ARM 中的通用寄存器是 R0～R15(R15 也是 PC)。它们可以被分为以下三类：

① 没有对应影子寄存器的寄存器 R0～R7。对于所有的模式，R0～R7 所对应的物理寄存器都是相同的。这 8 个寄存器是真正意义上的通用寄存器，ARM 体系结构中对它们没有作任何特殊的假设，它们的功能都是等同的。在中断或者异常处理程序中一般都需要对这几个寄存器进行保存。

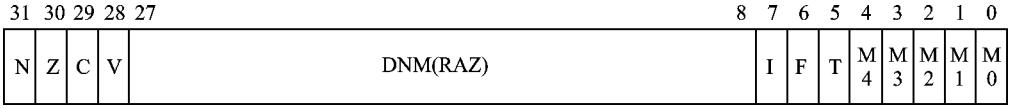
② 有对应影子寄存器的寄存器 R8～R14。程序访问的物理寄存器取决于当前的处理器模式。若要访问特定的物理寄存器而不依赖于当前的处理器模式，则要使用规定的名字。

R8～R12 有两组物理寄存器：一组是 FIQ 模式；另一组是除 FIQ 以外的其他模式。R13～R14 有 6 个组的物理寄存器，1 组用于用户模式和系统模式，其他 5 组分别用于 5 种异常模式。R13(也被称为 SP 指针)被用做栈指针，通常在系统初始化时需要对所有模式下的 SP 指针赋值；当 CPU 在不同的模式时栈指针会被自动切换成相应模式下的值。R14 有两个用途：一是在调用子程序时用于保存调用返回地址；二是在发生异常时用于保存异常返回地址。

③ 程序计数器 R15(或者 PC)。

2) CPSR

CPSR(当前程序状态寄存器)在所有的模式下都是可以读/写的。它主要包含条件标志、中断使能标志、当前处理器的模式、其他的一些状态和控制标志。



CPSR 的格式如下：

- 条件标志包括 N，Z，C，V。
  - N——Negative，负标志；
  - Z——Zero，零标志；

- C——Carry，进位标志；
- V——Overflow，溢出标志。

- 中断标志包括 I，F。
  - I—— 置 1 表示禁止 IRQ 中断的响应；置 0 表示允许 CPU 响应 IRQ 中断。
  - F——置 1 表示禁止 FIQ 中断的响应；置 0 表示允许 CPU 响应 FIQ 中断。
- ARM/Thumb 控制标志：T。
  - 置 0 表示执行 32 bits 的 ARM 指令；
  - 置 1 表示执行 16 bits 的 Thumb 指令 。
- 模式控制位 M0～M4，见表 2－6。

表 2－6　模式控制位及可用寄存器

M[4:0]	模　式	可用寄存器
0b10000	User	PC，R14～R0，CPSR
0b10001	FIQ	PC，R14_fiq～R8_fiq，R7～R0，CPSR，SPSR_fiq
0b10010	IRQ	PC，R14_irq，R13_irq，R12～R0，CPSR，SPSR_irq
0b10011	Supervisor	PC，R14_svc，R13_svc，R12～R0，CPSR，SPSR_svc
0b10111	Abort	PC，R14_abt，R14_abt，R12～R0，CPSR，SPSR_abt
0b11011	Undefined	PC，R14_und，R13_und，R12～R0，CPSR，SPSR_und
0b11111	System	PC，R14～R0，CPSR(ARM architecture v4 and above)

(4) 指令集

一个 CPU 的指令集是硬件和软件之间的一个重要的分水岭。根据分层的思想，指令集向上要有力地支持编译器，向下要方便硬件的设计实现。ARM 是典型的 RISC 体系，根据 RISC 的设计思想，其指令集的设计应该尽可能地简单；和 CISC 体系相比，它可以通过一系列简单的指令来实现复杂指令的功能。

ARM 的指令集包括 6 种典型的指令：

① 分支指令　如 B，BL 等；

② 数据处理指令　如 ADD，SUB，AND 等；

③ 状态寄存器转移指令　如 MRS，MSR 等；

④ Load-Store 数据移动指令　如 LDR 等；

⑤ 协处理器指令　如 LDC，STC 等；

⑥ 异常处理指令　如 SWI 等。

ARM 指令集是一个非常优秀的指令集。它有以下特点：

- 所有 ARM 指令都是 32 位定长，在内存中以 4 字节边界保存(地址最后两位为 0)，这

样方便译码电路和流水线的实现。当然，ARM 内核一般也支持另外一种 16 位的指令集 Thumb。Thumb 指令集可以看作是 ARM 指令集的一种压缩形式，它在处理器中仍然要扩展为标准的 32 位 ARM 指令来运行。用户采用 16 位 Thumb 指令集最大的好处就是可以获得更高的代码密度和降低功耗。

- Load－Store 架构。ARM 指令集属于 RISC 体系。RISC 体系的特征就是：一般指令只能把内部寄存器和立即数作为操作数，只有 Load－Store 类型的数据移动指令才可以访问内存，在内存和寄存器之间转移数据。
- 由于硬件上有桶形(barrel)移位器，所以 ARM 可以在一条指令中用一个指令周期完成一个移位操作和一个 ALU(算术逻辑)操作。
- 所有指令都可以条件执行，这是由其指令格式决定的，如下所示：

任何指令的高 4 位都是条件指示位，根据 CPSR 中的 N，Z，C，V 决定该指令是否执行。这样可以方便高级语言的编译器设计，很容易实现分支和循环。

- 有功能很强的一次加载和存储(Load－Store)多个寄存器的指令：LDM 和 STM。这样，当发生过程调用或中断处理时，只用一条指令就能把当前多个寄存器的内容保护到内存堆栈中。

(5) 异　常

异常是由内部或者外部原因引起的。当异常发生时 CPU 自动到指定的向量地址读取指令或地址并且执行。

对于 X86 CPU，当有异常发生时 CPU 是到指定的向量地址读取要执行的程序的地址，跳转到相应的地址并执行；而对于 ARM CPU，当有异常发生时 CPU 是到向量地址的地方读取指令并执行，也就是 ARM 的向量地址处存放的是一条指令(一般是一条跳转指令)。

ARM 将引起异常的类型分为 7 种，如表 2－7 所列。

表 2－7　ARM 的异常类型

异常类型	模　式	优先级	一般向量地址	高向量地址
Reset	Supervisor	1	0x00000000	0xFFFF0000
Undefined Instruction	Undefined	6	0x00000004	0xFFFF0004
Software Interrupt	Supervisor	6	0x00000008	0xFFFF0008
Prefetch Abort	Abort	5	0x0000000C	0xFFFF000C
Data Abort	Abort	2	0x00000010	0xFFFF0010
IRQ(interrupt)	IRQ	4	0x00000018	0xFFFF0018
FIQ(fast interrupt)	FIQ	3	0x0000001C	0xFFFF001C

当异常出现时，异常模式下的 R14 **和** SPSR **用于保存状态，即**

```
R14_<exception_mode> =return link
SPSR_<exception_mode> =CPSR
CPSR[4:0]=exception_mode number
CPSR[5]=0                          /* 在 ARM 状态执行 */
if <exception_mode> ==Reset or FIQ then
    CPSR[6]=1                      /* 禁止快速中断 */
    CPSR[7]=1                      /* 禁止正常中断 */
    PC=exception vector address
```

当处理异常返回时，将 SPSR **的值传送到** CPSR **中，将** R14 **的值传送到** PC **中。**

1）Reset **异常**

当 Reset **被触发时，**CPU **进入** Supervisor Mode **并且禁止** FIQ **和** IRQ**。具体操作如下：**

```
R14_svc= unpredictable value
SPSR_svc= unpredictable value
CPSR[4:0]=0b10011                  /* 进入 Supervisor 模式 */
CPSR[5]=0                          /* 在 ARM 状态执行 */
CPSR[6]=1                          /* 禁止快速中断 */
CPSR[7]=1                          /* 禁止正常中断 */
if high vectors configured then
    PC=0XFFFF0000
else
    PC=0X00000000
```

2）Undefined Instruction **异常**

有两种情形会触发 Undefined Instruction **异常：**

① CPU **执行一条协处理器指令时，未等到任何协处理器的应答；**

② CPU **执行一条未被定义的指令时，也会触发该异常。**

一旦发生该异常时，CPU **将完成如下的动作：**

```
R14_und= address of next instruction after the undefined instruction
SPSR_und= CPSR
CPSR[4:0]=0b11011                  /* 进入 Undefined 模式 */
CPSR[5]=0                          /* 在 ARM 状态执行 */
/*CPSR[6]不变*/
CPSR[7]=1                          /* 禁止正常中断 */
if high vectors configured then
    PC=0XFFFF0004
```

```
else
    PC=0X00000004
```

3) Software Interrupt **异常**

软中断是执行 SWI **指令时触发的,该异常主要用于进入** OS **的系统调用。在触发该异常后** CPU **完成如下的动作:**

```
R14_svc= address of next instruction after SWI
SPSR_svc= CPSR
CPSR[4:0]=0b10011              /* 进入 Supervisor 模式 */
CPSR[5]=0                      /* 在 ARM 状态执行 */
/* CPSR[6]不变 */
CPSR[7]=1                      /* 禁止正常中断 */
if high vectors configured then
    PC=0XFFFF0008
else
    PC=0X00000008
```

4) Prefetch Abort **异常**

如果 CPU **在读取指令时发生读内存错误且该指令还要被执行,则触发该异常;如果只是在读取指令时发生了内存错误而该指令又未被执行,则不会触发该异常。在** ARMv5 **和以上的版本中,**BKPT **指令(断点指令)也会触发该异常。**

```
R14_abt= address of aborted instruction + 4
SPSR_abt= CPSR
CPSR[4:0]=0b10111              /* 进入 Abort 模式 */
CPSR[5]=0                      /* 在 ARM 状态执行 */
/* CPSR[6]不变 */
CPSR[7]=1                      /* 禁止正常中断 */
if high vectors configured then
    PC=0XFFFF000C
else
    PC=0X0000000C
```

5) Data Abort **异常**

当 CPU **在读/写数据时,如果发生错误,则触发该异常。**

```
R14_abt= address of aborted instruction + 8
SPSR_abt= CPSR
CPSR[4:0]=0b10111              /* 进入 Abort 模式 */
```

```
CPSR[5]=0                    /* 在 ARM 状态执行 */
/* CPSR[6]不变 */
CPSR[7]=1                    /* 禁止正常中断 */
if high vectors configured then
    PC=0XFFFF00010
else
    PC=0X000000010
```

6）IRQ

当外部 IRQ 输入请求发生时(IRQ 中断已经被使能)，触发该异常。触发该异常后，IRQ 被禁止，但是 FIQ 的状态未被改变。常用的中断返回方式为 SUBS PC，R14，#4。

```
R14_irq= address of next instruction to be executed + 4
SPSR_irq= CPSR
CPSR[4:0]=0b10010            /* 进入 IRQ 模式 */
CPSR[5]=0                    /* 在 ARM 状态执行 */
/* CPSR[6]不变 */
CPSR[7]=1                    /* 禁止正常中断 */
if high vectors configured then
    PC=0XFFFF0018
else
    PC=0X00000018
```

7）FIQ

当外部 FIQ 输入请求发生时(FIQ 中断已经被使能)，触发该异常。FIQ 通常被用于快速传输数据，触发该异常后，FIQ 和 IRQ 都被禁止。常用的中断返回方式为 SUBS PC，R14，#4。

```
R14_fiq= address of next instruction to be executed + 4
SPSR_fiq= CPSR
CPSR[4:0]=0b10001            /* 进入 FIQ 模式 */
CPSR[5]=0                    /* 在 ARM 状态执行 */
CPSR[6]=1                    /* 禁止快速中断 */
CPSR[7]=1                    /* 禁止正常中断 */
if high vectors configured then
    PC=0XFFFF001C
else
    PC=0X0000001C
```

(6) 内存和 I/O 地址

ARM **的寻址空间是线性地址空间，最大为** 2^{32} B=4 GB。ARM **支持大端和小端的内存**

数据方式，可以通过硬件的方式（没有提供软件的方式）设置端模式。大小端的数据存放格式分别如图 2－5 和图 2－6 所示。

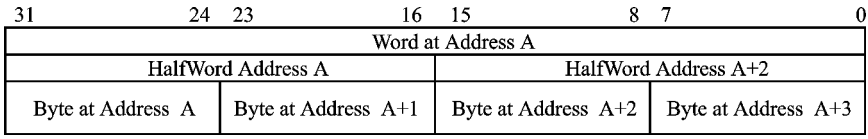


图 2－5　大端的数据存放格式

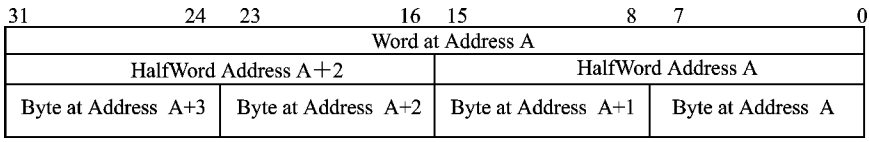


图 2－6　小端的数据存放格式

I/O 端口的编址方法即地址安排方式有两种。

1）存储器映射编址

I/O 端口的地址与内存地址统一编址，即 I/O 单元与内存单元在同一地址空间。

其优点是可采用丰富的内存操作指令访问 I/O 单元，无需单独的 I/O 地址译码电路，无需专用的 I/O 指令；缺点是外设占用内存空间，不易区分 I/O 程序。比如 ARM 的 I/O 端口与内存单元就是统一编址的。通常将 I/O 所映射的存储系统标识为非高速缓存（uncachable）和非缓冲（unbufferable）的。

2）I/O 映射编址

I/O 端口与内存单元分开编址，即 I/O 单元与内存单元都有自己独立的地址空间。

其优点是 I/O 单元不占用内存空间，易区分 I/O 程序；缺点是 I/O 操作指令仅有单一的传送指令，I/O 接口需有地址译码电路。例如：Intel 80X86 系列处理器的 I/O 端口与内存单元分开编址，I/O 端口有自己独立的地址空间，其大小为 64 KB。

2. MIPS 系列

MIPS 是世界上很流行的一种 RISC 处理器。MIPS（Microprocessor without Interlocked Piped Stages）的意思是“无互锁流水级的微处理器”，其机制是尽量利用软件办法避免流水线中的数据相关问题。

MIPS 这个名字其实也是 MIPS 科技公司的名字。这是一家设计制造高性能、高档次及嵌入式 32 位和 64 位处理器的厂商，在 RISC 处理器领域占有重要地位。MIPS 处理器是由斯坦福（Stanford）大学 Hennessy 教授领导的研究小组研制出来的。1984 年 MIPS 计算机公司成立，1986 年推出 R2000 处理器，1988 年推出 R3000 处理器，1991 年推出第一款 64 位商用微处理器 R4000。1992 年，SGI 公司收购了 MIPS 计算机公司之后，该公司又陆续推出 R8000（于 1994 年）、R10000（于 1996 年）和 R12000（于 1997 年）等型号的处理器。1998 年，MIPS 脱

离 SGI 公司成为 MIPS 技术公司之后，其战略发生变化，把重点放在了嵌入式系统上。1999 年，MIPS 公司发布 MIPS 32 和 MIPS 64 架构标准，为未来 MIPS 处理器的开发奠定了基础。新的架构集成了所有原来的 MIPS 指令集，并且增加了许多更强大的功能。MIPS 公司陆续开发了高性能、低功耗的 32 位处理器内核(core)MIPS 32 4Kc 与高性能的 64 位处理器内核 MIPS 64 5Kc。2000 年，MIPS 公司发布了针对 MIPS 32 4Kc 的新版本以及未来的 64 位 MIPS 64 20Kc 处理器内核。

和 ARM 公司一样，MIPS 公司本身并不从事芯片的生产活动(只进行设计)，不过其他公司如果要生产该芯片，则必须得到 MIPS 公司的许可。

下面将重点介绍 MIPS 32 和 MIPS 64 体系结构。

(1) MIPS 体系的组成

1) MIPS 指令集体系 ISA

MIPS 指令集体系 ISA(MIPS Instruction Set Architecture)从最早的 MIPS I ISA 开始发展，到 MIPS V ISA，再到现在的 MIPS 32 和 MIPS 64 结构，其所有版本都是与前一个版本兼容的。在 MIPS III 的 ISA 中，增加了 64 位整数和 64 位地址；在 MIPS IV 和 MIPS V 的 ISA 中，增加了浮点数的操作等。

MIPS 32 和 MIPS 64 体系是为满足高性能、成本敏感的需求而设计的。MIPS 32 体系是基于 MIPS II ISA 的，并从 MIPS III，MIPS IV 和 MIPS V 中选择一些指令以增强数据及代码的有效操作。MIPS 64 体系是基于 MIPS V ISA 并与 MIPS 32 体系兼容的。MIPS 32 和 MIPS 64 体系都将特权环境(privileged environment)引入体系，以满足操作系统或其他核心软件的需要。同时，MIPS 32 和 MIPS 64 体系支持 ASE(MIPS Application Specific Extensions)，UDI(User Defined Instructions)和客户协处理器，以满足特定领域的应用需求。MIPS 32和 MIPS 64 体系的关系如图 2－7 所示。

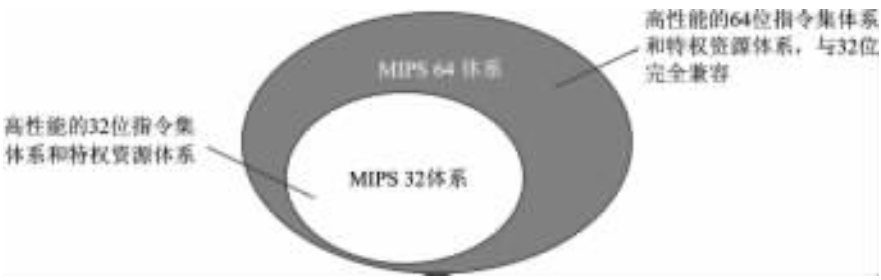


图 2－7　MIPS 32 和 MIPS 64

2) MIPS 特权资源体系 PRA

MIPS 32 和 MIPS 64 的 PRA(MIPS Privileged Resource Architecture)定义了一组指令，其中大多数指令只能在特权模式下使用。有些指令在非特权模式下也是可见的，如虚拟内存

布局。PRA 提供了管理处理器资源所必需的机制，如虚存、Cache、异常和用户的上下文等。

3) MIPS 特定应用扩展 ASE

MIPS 32 和 MIPS 64 体系支持可选的特定应用的扩展 ASE(MIPS Application Specific Extensions)。ASE 是对基本体系的扩展，不承担体系中指令的实现，是在 ISA 和 PRA 基础上完成特定领域应用的需要。

4) MIPS 用户定义指令集 UDI

除了支持 ASE 外，MIPS 32 和 MIPS 64 体系还提供专门的指令，即用户定义指令集 UDI (MIPS User Defined Instructions)。这些指令的功能是在具体实现时定义的。

(2) 数据类型

MIPS 处理器定义了下列数据格式：

- 位 Bit (b)；
- 字节 Byte (8 bits,B)；
- 半字 Halfword (16 bits,H)；
- 字 Word (32 bits,W)；
- 双字 Doubleword (64 bits,D)。

浮点处理器定义了下列数据格式：

- 32 位的单精度浮点数(. fmt type S)；
- 32 位的单精度浮点单对数(. fmt type PS)；
- 64 位双精度浮点数(. fmt type D)；
- 32 位字固定数(. fmt type W)；
- 64 位长固定数(. fmt type L)。

(3) 协处理器

MIPS 体系定义了 4 个协处理器 CP0，CP1，CP2 和 CP3。

- CP0 是在 CPU 芯片内的，支持虚存、管理异常和处理核心态与用户态的切换，控制 Cache 系统，提供诊断控制和错误恢复机制，通常被称为系统控制协处理器 SSC(System Control Coprocessor)；
- CP1 保留给 FPU；
- CP2 可用于专门的实现；
- CP3 保留给 MIPS 64 版本 1 和所有体系版本 2 的 FPU。

(4) CPU 寄存器

MIPS 32 体系定义了下列 CPU 寄存器。

- 32 个 32 位通用寄存器 GPRs(General Purpose Registers)：r0～r31；
- 一对专门的寄存器 HI 和 LO，用于存放整数乘、除和乘加运算的结果；
- 程序计数器 PC。

2.3　总　线

微处理器需要与一定数量的部件和外围设备连接；如果将各部件和每一种外围设备都分别用一组线路与 CPU 直接连接，那么连线将会错综复杂，甚至难以实现。为了简化硬件电路设计和简化系统结构，常用一组线路，配置以适当的接口电路，将 CPU 与各部件和外围设备连接，这组共用的连接线路被称为总线。

总线是指一组进行互连和传输信息（指令、数据和地址）的信号线，是连接系统各个部件之间的桥梁。采用总线结构便于部件和设备的扩充，尤其是制定了统一的总线标准后更容易使不同设备间实现互连。

嵌入式系统的总线一般分为片内总线和片外总线。片内总线就是嵌入式微处理器内的 CPU 与片内其他部件连接的总线；片外总线集成在嵌入式微处理器内或外接芯片扩展上，用于连接外部设备，如图 2－8 所示。下面重点介绍 ARM 的片内总线 AMBA 和近几年已在嵌入式系统广泛应用的片外总线 PCI 总线。

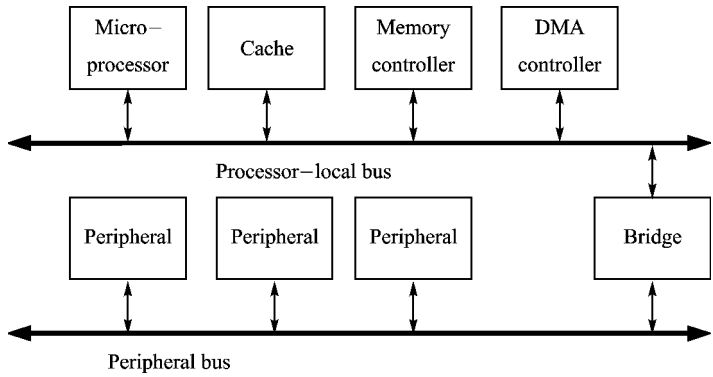


图 2－8　嵌入式系统总线

2.3.1　AMBA 总线

AMBA（Advanced Microcontroller Bus Architecture）是 ARM 公司研发的一种总线规范，目前为 3.0 版本。在 AMBA 总线规范中，定义了 3 种总线：

① AHB（Advanced High-performance Bus）　用于高性能系统模块的连接，支持突发模式数据传输和事务分割；可以有效地连接处理器、片上和片外存储器，支持流水线操作。

② ASB（Advanced System Bus）　也用于高性能系统模块的连接，支持突发模式数据传输。这是较老的系统总线格式，后来由 AHB 总线替代。

③ APB（Advanced Peripheral Bus）　用于较低性能外设的简单连接，一般是接在 AHB

或 ASB 系统总线上的第二级总线。

一个典型的基于 AMBA 总线的系统框图如图 2－9 所示。

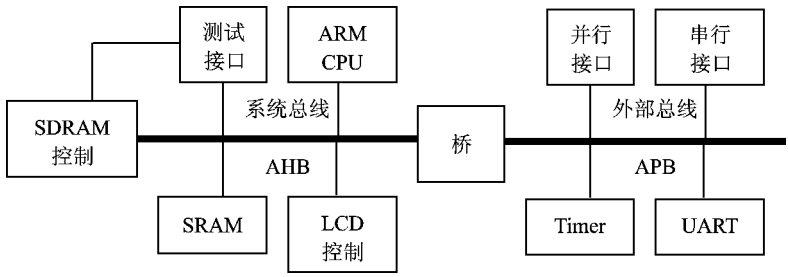


图 2－9　基于 AMBA 总线的典型系统

1. AHB

(1) AHB 总线的组成

AHB 总线主要由主单元、从单元、仲裁器和译码器组成。

1) AHB 主单元

总线的主单元可以初始化读或写。只有主单元可在任何时刻使用总线。AHB 可以有一个或多个主单元。主单元可以是 RISC 处理器、DSP 以及 DMA 控制器，以启动和控制总线操作。

2) AHB 从单元

从单元可以响应(并非启动)读或写总线操作。总线的从单元可以在给定的地址范围内对读/写操作进行相应的反应。从单元向主单元发出成功、失败信号或等待各种反馈信号。从单元通常是其复杂程度不足以成为主单元的固定功能块，例如外存接口、总线桥接口以及任何内存都可以是从单元，系统的其他外设也包含在 AHB 的从单元中。

3) AHB 仲裁器

总线仲裁器用来确定控制总线的是哪个主单元，以保证在任何时候只有一个主单元可以启动数据传输。一般来说仲裁协议都是固定好的，例如最高优先级方法或平等方法。可根据实际的情况选择适当的仲裁协议。

4) AHB 译码器

总线译码器用于传输的译码工作，提供传输过程中从单元的片选信号。

(2) AHB 总线工作过程

图 2－10 是一个典型的 AHB 总线工作过程。它包括以下两个阶段：

① 地址传送阶段(address phase)　它将只持续一个时钟周期。在 HCLK 的上升沿数据有效。所有的从单元都在这个上升沿来采样地址信息。

② 数据传送阶段(data phase)　它需要一个或几个时钟周期。可通过 HREADY 信号来

延长数据传输时间；当 HREADY 信号为低电平时，就在数据传输中加入等待周期，直到 HREADY 信号为高电平才表示这次传输阶段结束。

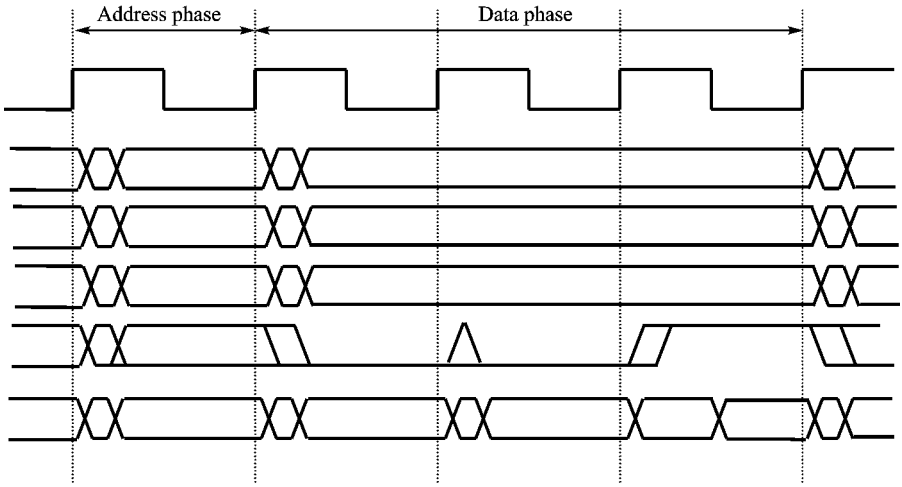


图 2－10　AHB 总线工作过程

上面的地址传送阶段和数据传送阶段是相互分开的；但是在实际应用中，地址传送阶段发生在前一次传输的数据传送阶段。地址传送阶段和数据传送阶段之间的重合对于总线流水线操作以及提高系统的性能很有好处。

AHB 信号的具体说明请参照 AMBA 的规格书。

(3) 传输类型

主单元使用 HTRANS[1:0]来表示主单元在传输时所处的状态，有下列 4 种类型：

① IDLE 状态　表示当前主单元不会进行数据传输，从单元可以忽略本次传输。

② BUSY 状态　允许主单元在猝发传输中插入一个 IDLE 的等待状态，表示主单元将会继续进行猝发传输，但是下一个传输不会马上发生。如果一个主单元处于 BUSY 状态，则地址和控制信号必须是猝发传输中的下一个传输信息。

③ NONSEQ 状态　表示主单元要传输单个数据或连续数据中的第一个数据，主单元传输的控制信号和上次传输的没有什么联系。

④ SEQ 状态　表示主单元传输的控制信息和数据与上次传输的有联系，控制信息和上次的相同，而地址等于上次传送的地址加固定值。

从单元通过使用 HREADY 信号在传输中插入适当的等待周期；如果这次传输从单元返回的是 HREADY 为高电平，并且返回一个 OKAY 信号，那么就表示这次传输成功完成。

从单元使用 HRESP[1:0]来表示从单元在传输中所处的状态，有下列 4 种类型：

① OKEY 状态　表示传输正常，如果 HREADY 为高电平，则传输顺利完成；

② ERROR 状态　表示传输错误，通过 ERROR 状态让主单元认识到传输无法成功完成；

③ RETRY 状态　表示传输无法立即完成，请求主单元继续传输；

④ SPLIT 状态　同样表示传输无法立即完成，请求主单元继续传输。

2. APB

APB 主要由 APB 桥和 APB 从单元(slave)组成。APB 桥是 APB 中惟一的主单元，是 AHB/ASB 的从单元。

APB 桥的接口信号如图 2－11 所示。APB 桥将系统总线 AHB/ASB 和 APB 连接起来，并执行下列功能：

- 锁存地址并保持其有效，直到数据传送完成；
- 译码地址并产生一个外部片选信号，在每次传送时只有一个片选信号(PSELx)有效；
- 写传送(write transfer)时驱动数据到 APB；
- 读传送(read transfer)时驱动数据到系统总线 AHB/ASB；
- 传送时产生定时触发信号 PENABLE。

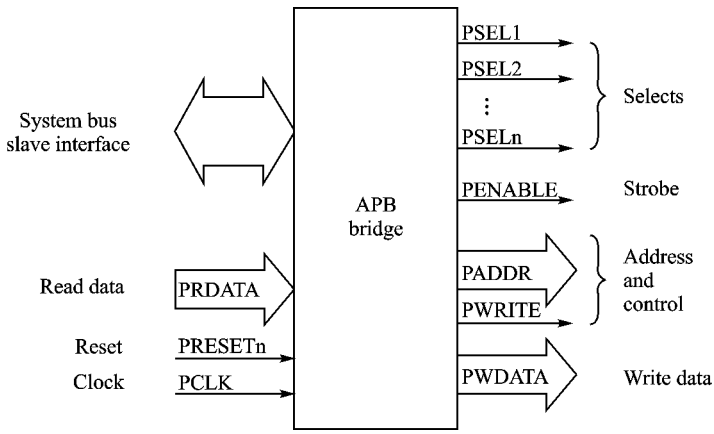


图 2－11　APB 桥的接口信号

APB 从单元具有简单灵活的接口。接口的具体实现是依赖于特定设计的，有许多不同的可能。APB 从单元的接口信号如图 2－12 所示。APB 从单元的接口是非常灵活的，当写传送时数据锁定在下列点：

- 当 PSELx 为高电平时，在每个 PCLK 的上升沿，在 PENABLE 的上升沿；
- 片选信号 PSELx 和地址信号 PADDR 可合并起来决定需要操作的寄存器。

读传送时，当 PWRITE 为低电平并且 PSELx 和 PENABLE 同时为高电平时，数据传送到数据总线上。PADDR 用于确定读哪个寄存器。

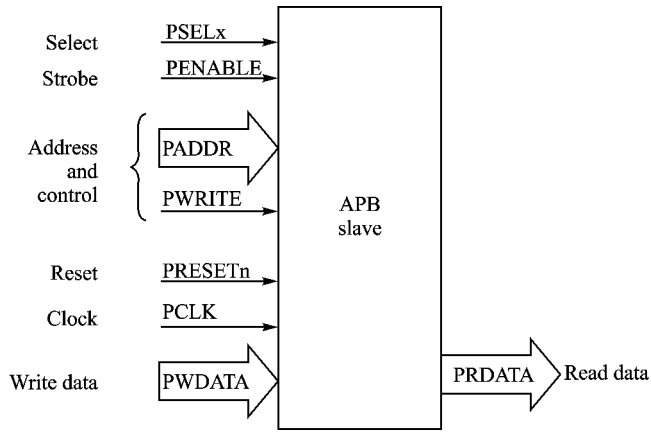


图 2-12　APB 从单元的接口信号

2.3.2　PCI 总线

随着嵌入式系统的发展，嵌入式系统已开始逐步采用微机系统普遍采用的 PCI(Peripheral Component Interconnect)总线，以便于系统的扩展。

1991 年 Intel 公司联合世界上多家公司成立的 PCISIG(Peripheral Component Interconnect Special Interest Group)协会，是国际上微型机界的行业协会。它致力于促进 PCI 总线工业标准的发展。PCI 总线规范先后经历了 1.0 版、2.0 版和 1995 年的 2.1 版。PCI 总线是地址、数据多路复用的高性能 32 位和 64 位总线，是微处理器与外围控制部件、外围附加板之间的互连机构。它规定了互连的协议、电气、机械及配置空间规范，以保证全系统的自动配置；在电气方面还专门定义了 5 V 和 3.3 V 信号、环境，特别是 2.1 版本定义了 64 位总线扩展以及 66 MHz总线时钟的技术规范。

从数据宽度上看，PCI 总线有 32 位 、64 位之分；从总线速度上分，有 33 MHz，66 MHz 两种。目前流行的是 32 bit @33 MHz，而 64 位系统正在普及中。改良的 PCI 系统 PCI-X，最高可以达到 64 bit @ 133 MHz 的数据传输速率。

1. PCI 总线概述

与 ISA 总线不同，PCI 总线的地址总线与数据总线是分时复用的，支持即插即用(plug and play) 、中断共享等功能。分时复用的好处是一方面可以节省接插件的引脚数，另一方面便于实现突发数据传输。

数据传输时，由一个 PCI 设备做发起者(主控、Initiator 或 Master)，而另一个 PCI 设备做目标(从设备、Target 或 Slave)。总线上所有时序的产生与控制都由 Master 来发起。PCI 总线在同一时刻只能供一对设备完成传输。这就要求有一个仲裁机构来决定谁有权拿到总线的

主控权。

32 位 PCI 系统的引脚按功能来分有以下几类。

(1) 系统控制

- CLK　PCI 时钟,上升沿有效;
- RST　Reset 信号。

(2) 传输控制

- FRAME #　标志传输开始与结束;
- IRDY#　Master 可以传输数据的标志;
- DEVSEL #　当 Slave 发现自己被寻址时设置低电平应答;
- TRDY#　Slave 可以传输数据的标志;
- STOP #　Slave 主动结束传输数据;
- IDSEL　在即插即用系统启动时用于选中板卡的信号。

(3) 地址与数据总线

- AD[31 : :0]　地址/数据分时复用总线;
- C/ BE #[3 : :0]　命令/字节使能信号;
- PAR　奇偶校验信号。

(4) 仲裁信号

- REQ #　Master 用来请求总线使用权;
- GNT #　仲裁机构允许 Master 得到总线使用权。

(5) 错误报告

- PERR #　数据奇偶校验错;
- SERR #　系统奇偶校验错。

当 PCI 总线进行读操作时如图 2 - 13 所示。发起者先置 REQ #,当得到仲裁器的许可时(GNT #),将 FRAME #置低电平,并在 AD 总线上放置 Slave 地址,同时 C/ BE #放置命令信号,说明接下来的传输类型。PCI 总线上的所有设备都需对此地址译码,被选中的设备置 DEVSEL #以声明自己被选中。然后当 IRDY# 与 TRDY# 都置低时,传输数据。Master 在数据传输结束前,将 FRAME #置高以表明只剩最后一组数据要传输,并在传完数据后放开 IRDY#以释放总线控制权。由此可见,PCI 总线的传输是高效的,发出一组地址后,理想状态下可以连续发数据,峰值速率为 132 MB/s。

2. CPCI 总线

为了将 PCI 总线规范用在工业控制计算机系统上,1995 年 11 月 PCI 工业计算机制造者联合会(PICMIG)颁布了 CompactPCI(以后简称 CPCI)规范 1.0 版,以后相继推出了 PCI - PCI Bridge 规范、Computer Telephony TDM 规范和 User - Defined I/O Pin Assignment 规范,1997 年推出了 CPCI 2.0 规范。

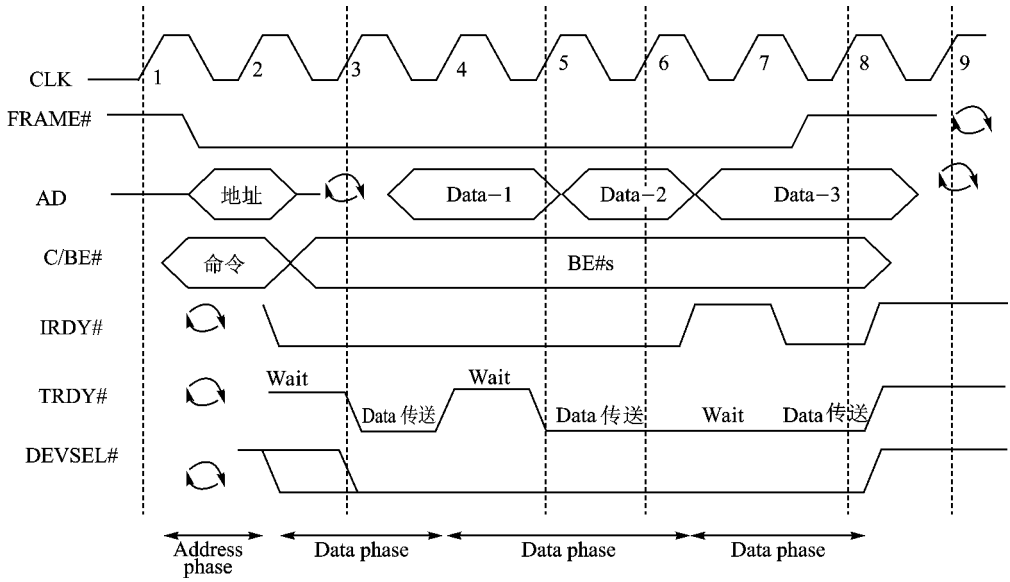


图 2-13　PCI 总线读操作

简言之，CPCI **总线规范**＝PCI **总线的电气规范**＋ **标准针孔连接器**（IEC—1076—4—101）＋**欧洲卡规范**（IEC 297/IEEE 1011.1）。

目前在嵌入式 PC、**工控计算机及高端的嵌入式系统中已经开始大量采用** CPCI **接口**，CPCI**将逐步替代** VME **和** Multibus **总线**。CPCI **总线工控机之所以被业界所青睐，是因为其既具有** PCI **总线的高性能，又具有欧洲卡结构的高可靠性，是符合国际标准的真正工业型计算机，适合在可靠性要求较高的工业和军事设备上应用**。

CPCI **总线工控机定义了两种板卡尺寸，即** 3U（100 mm×160 mm）**和** 6U（233 mm×160 mm），**主要具有以下特点**：

- **标准欧洲卡尺寸，符合** IEEE 1101.1 **结构标准**；
- **气密性、高密度** 2 mm **针孔连接器，符合** IEC—1076 **国际标准**；
- **板卡垂直于地面安装，利于散热降温**；
- **板卡正向安装，反向拔出，四面固定**；
- **良好的抗振动和抗冲击特性**；
- **金属前面板，便于安装、固定和指示**；
- **现场** I/O **信号由板后通过针孔连接器引出，弹性连接，抗腐蚀、抗振动性能好**；
- **标准金属机箱**（铝），EMC，ESD **性能好，抗干扰能力强**；
- **支持热插拔和热切换**（hotswap）；
- **多处理器和多操作系统支持**；

- 无源母板，标准插槽，可扩展性和伸缩性能好；
- 支持混合总线系统。

2.4 存储器

2.4.1 存储器结构

目前较为复杂的嵌入式系统的存储结构如图 2－14 所示。

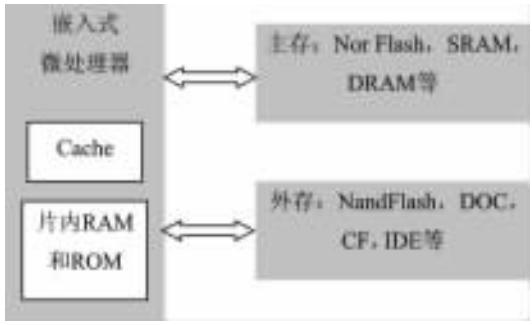


图 2－14 嵌入式系统的存储结构

嵌入式系统的存储器分三种：高速缓存 Cache、主存（片内和片外）和外存。

1. 高速缓存 Cache

高速缓冲存储器中存放的是当前使用得最多的程序代码和数据，即主存中部分内容的副本。在嵌入式系统中 Cache 全部都集成在嵌入式微处理器内，可分为数据 Cache、指令 Cache 或混合 Cache，不同的处理器其 Cache 的大小不一样。一般中高档的嵌入式微处理器才内置 Cache。

2. 主 存

主存是处理器能直接访问的存储器，用来存放系统和用户的程序及数据。嵌入式系统的主存可位于处理器内和处理器外。片内存储器存储容量小、速度快；片外存储器容量大。

可以做主存的存储器如下：

- ROM 类 Nor Flash，EPROM，E^2PROM，PROM 等；
- RAM 类 SRAM，DRAM，SDRAM 等。

3. 外 存

外存是处理器不能直接访问的存储器，用来存放用户的各种信息，容量大，存取速度相对主存而言要慢得多，但它可用来长期保存用户信息。

在嵌入式系统中常用的外存有：NandFlash，DOC(Disk On Chip)，CF(Compact Flash)，SD(Secure Digital)和 MMC(Multi Media Card)等。

2.4.2　电子盘

电子盘采用半导体芯片来存储数据，具有体积小、功耗低和极强的抗震性等特点。因此在嵌入式系统中普遍采用各种电子盘作为外存。常用的电子盘有：NandFlash，DOC，DOM，CF，SM，MS，MMC 和 SD 等。

1. NandFlash

NandFlash 是 Flash Memory 的一种。NandFlash 可独立成为外存，也可组成其他各种类型的电子盘如 USB 盘、CF、SD 和 MMC 存储卡等。

Flash Memory 的中文称为快闪存储器或快速擦写存储器。Flash Memory 由 Toshiba 公司于 1980 年申请专利，并在 1984 年的国际半导体学术会议上首先发表。目前在 Flash Memory 技术上主要发展了两种非易失性内存：一种叫 NOR(逻辑“或”)，另一种叫 NAND(逻辑“与”)。前者是 Intel 公司于 1988 年发明的，后者是 Toshiba 公司于 1999 年创造的。

NandFlash 强调降低每位的成本和更高的性能，并且像磁盘一样可以通过接口轻松升级。目前有能力大规模生产 NandFlash 的只有少数厂家：Samsung，Toshiba，Sandisk 和 Fujitsu 等，主要厂商是 Samsung 和 Toshiba 两家。

NorFlash 具有随机存储速度快、电压低、功耗低和稳定性高等特点，主要用于主存。NandFlash 具有容量大、回写速度快和芯片面积小等特点，主要用于外存。表 2-8 是 NOR 和 NAND 两种 Flash 的对比。

表 2-8　NOR 和 NAND 对比

类　别	NOR	NAND
写入/擦除一个块的操作时间	1～5 s	2～4 ms
读性能	1 200～1 500 KB	600～800 KB
写性能	＜80 KB	200～400 KB
接口/总线	SRAM 接口/独立的地址数据总线	8 位地址/数据/控制总线，I/O 接口复杂
读取模式	随机读取	串行地存取数据
成本	较高	较低，单元尺寸约为 NOR 的一半，生产过程简单，同样大小的芯片可以做更大的容量

续表 2-8

类　别	NOR	NAND
容量及应用场合	1～64 MB,主要用于存储代码	8 MB～1 GB,主要用于存储数据
擦写次数(耐用性)	约 10 万次	约 100 万次
位交换(bit 位反转)	少	较多,关键性数据需要错误探测/错误更正(EDC/ECC)算法
坏块处理	无,因为坏块故障率少	随机分布,无法修正

2. DOC

Disk On Chip,简称 DOC,是采用 NandFlash 芯片作为基本存储单元,外加一些控制芯片,通过特殊的软硬件来操作的一种模块化、系列化的电子存储装置。它采用了 TureFFS 硬盘仿真技术对 Flash 进行管理,可以把 Flash 模拟成为硬盘,使用方便且容量可以达到 288 MB。也正是因为采用了 TureFFS 技术对数据在 Flash 中的读/写操作进行管理,大大提高了 DOC 写操作的次数,远远超过了普通 Flash 的写寿命,提高了 Flash 的可靠性。

DOC 封装形式主要有三种:单芯片封装、塑胶外壳封装和贴片式封装。

3. DOM

DOM 是一种采用 IDE 接口的电子盘,可以直接插在 IDE 接口上,像硬盘一样方便地使用,而体积却比硬盘小很多。它最大的好处就是可以在没有 DOC 插槽或 CF 插槽的板卡上,非常方便地使用电子盘来替代硬盘、软盘。

4. CF

CF 的诞生比较早,由最大的 Flash Memory 卡厂商之一美国的 Sandisk 公司于 1994 年首次推出。CF 的大小仅为 43 mm×36 mm×3.3 mm,体积只有 PCMCIA 卡的 1/4,看起来就像是 PCMCIA 卡的缩小版。

CF 提供了完整的 PCMCIA-ATA 功能,而且通过 ATA/ATAPI-4 兼容 TrueIDE。与 68 针接口的 PCMCIA 卡不同,同样遵从 ATA 协议的 CF 的接口只有 50 针。

5. SM

SM(Smart Media)是由 Toshiba 公司 Toshiba America Electronic Components(TAEC)在 1995 年 11 月发布的 Flash Memory 存储卡,Samsung 公司在 1996 年购买了生产和销售许可。这两家公司成为主要的 SM 厂商。

最开始时 SM 被称为 SSFDC,即 Solid State Floppy Disk Card,1996 年 6 月改名为 SM,并成为 Toshiba 公司的注册商标。

SM 采用了 NAND 型 Flash Memory,因此体积做得很小,仅为 45 mm×37 mm×0.76 mm,

非常地薄，质量仅有 1.8 g。在接口方面，SM 采用了 22 针的接口，人们在卡上看到的是扁平的金手指。

为了节省成本，SM 存储卡上只有 Flash Memory 模块和接口，并没有包括控制芯片，所以使用 SM 的设备必须自己装置控制机构。

6. MS

1997 年 7 月 Sony 公司宣布开发 MS(Memory Stick)。MS 被很多人称为口香糖存储卡，因为其尺寸确实像一条香口胶：50 mm×21.5 mm×2.8 mm，质量为 4 g。MemoryStick 包括了控制器，采用 10 针接口，数据总线为串行的，最高频率可达 20 MHz，电压为 2.7～3.6 V，电流平均为 45 mA。

7. MMC

MMC 即 Multi Media Card，由 Siemens 公司(现在称为 Infineon 公司)和 Sandisk 公司于 1997 年推出。1998 年 1 月 14 家公司联合成立了 MMC 协会(简称 MMCA)，现有成员超过 84 个，其中包括了 Nokia，Ericsson，Motorola 公司等手机巨头。

MMC 号称是目前世界上最小的 Flash Memory 存储卡，尺寸只有 32 mm×24 mm×1.4 mm；虽然比 SM 厚，但整体体积却比 SM 小，而且也比 SM 轻，质量只有 1.5 g。

和 CF 一样，MMC 也是把存储单元和控制器一同做到了卡上，智能的控制器使得 MMC 保证了兼容性和灵活性。

MMC 被设计作为一种低成本的数据平台和通信介质，接口设计非常简单，只有 7 针，接口成本低于 0.5 美元。

8. SD

SD 卡由 Panasonic，Toshiba 和 SanDisk 公司联合推出，1999 年 8 月被首次发布。2000 年 2 月 1 日成立了 SD 协会(简称 SDA)，成员公司超过 90 个，其中包括 Hewlett－Packard，IBM，Microsoft，Motorola，NEC，Samsung Electronics，Toyota Motor 公司等巨头。SDA 是开放式的，缴纳 1 500 美元就可以成为一般会员，缴纳 4000 美元可以成为执行会员。SD 存储卡的详细规范并没有公开，只有 SDA 会员或签订了保密协议才能获得。

SD 卡数据传送和物理规范由 MMC 发展而来，其大小和 MMC 差不多，尺寸为 32 mm×24 mm×2.1 mm。其长宽和 MMC 一样，只是厚了 0.7 mm，以容纳更大容量的存储单元。

SD 卡与 MMC 卡保持着向上兼容，也就是说，MMC 可以被新的 SD 设备存取，其兼容性取决于应用软件，但 SD 卡却不可以被 MMC 设备存取。SD 接口除了保留 MMC 的 7 针外，还在两边多加了 2 针作为数据线。

表 2－9 为 MMC，SM，MS，SD 和 CF 的对比。

表 2-9　MMC,SM,MS,SD 和 CF 的对比

类　别	MMC	SM	MS	SD	CF
体积/mm^3	1 075	1 298	3 010	1 612	5 108 7 740
厚度/mm	1.4	0.7	2.8	2.1	3.5 5
质量/g	1.5	1.8	3.0	2.0	—
最大容量	64 MB	128 MB	128 MB	128 MB	1.0 GB
电压/电流	2.7 V/27 mA	3.3 V/40 mA	2.7 V/35 mA	2.7 V	5 V,3.3 V
接口/pin	7	22	10	9	50

注:表中的数据仅供参考。

2.5　输入/输出接口和设备

嵌入式系统是面向应用的,不同的应用所需的接口和外设不同。在嵌入式系统中通常大多数接口和部分外设已经集成到嵌入式处理器上,如 Timer,RTC,UART,GPIO,I^2C,USB,IrDA,MII,A/D,D/A,LCD Controller,DMA 控制器和中断控制器等。

这些接口和设备的工作原理与微机系统中的很类似,由于篇幅所限这里就不一一叙述,读者可参考其他教材和资料。

思考题

2.1　嵌入式硬件系统由哪些部分组成?

2.2　嵌入式微处理器的分类、特点是什么?主流的嵌入式处理器有哪些?

2.3　ARM 有几种运行模式?哪些具有特权?如何改变处理器的模式?运行模式和寄存器的关系如何?什么是影子寄存器?

2.4　ARM 有几种异常?其异常处理方式和 X86 有什么不同?

2.5　ARM 和 X86 采用哪种 I/O 编址方式?请比较其特点。

2.6　AMBA 属于哪种总线?请以一款 ARM 芯片为例分析 AMBA 的作用。

2.7　请对比 NorFlash 和 NandFlash,并指出其在嵌入式系统中的作用是什么?

2.8　请分别填出在大端数据存放格式和小端数据存放格式(见图 2-15)下,下列变量在内存中的存放情况(该机器的字长为 32 位)。

变量 A:word A=0xf6 73 4b cd,在内存中的起始地址为 0xb3 20 45 00;

变量 B:half word B=218,**在内存中的起始地址为** 0xdd dd dd d0。

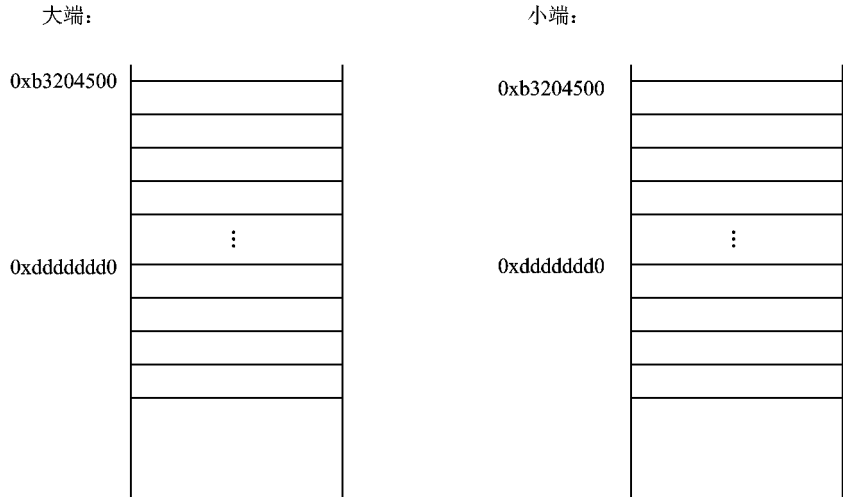


图 2－15　数据存放格式

第 3 章　嵌入式软件系统

3.1　嵌入式软件系统概述

嵌入式软件是计算机软件的一种，具有软件的一般特性，同时还具有其特殊性。下面首先回顾一下软件的一般特性。

软件(software)是计算机系统中与硬件(hardware)相互依存的另一部分。它包括程序(program)、相关数据(data)及其说明文档(document)。其中：

- 程序是按照事先设计的功能和性能要求执行的指令序列；
- 数据是程序能正常操纵信息的数据结构；
- 文档是与程序开发维护和使用有关的各种图文资料。

软件同传统的工业产品相比，有其独特的性质。

① 软件是一种逻辑实体，具有抽象性。这个特点使它与其他工程对象有着明显的差异。人们可以把它记录在纸、内存、磁盘或光盘上，但却无法看到软件本身的形态，必须通过观察、分析、思考和判断，才能了解它的功能及特性。

② 软件没有明显的制造过程。一旦软件研制开发成功，就可以大量拷贝同一内容的副本。所以对软件的质量控制，必须着重在软件开发方面下功夫。

③ 软件在使用过程中，没有磨损、老化的问题。软件在生存周期后期不会因为磨损而老化，但会为了适应硬件、环境以及需求的变化而被修改，而这些修改又不可避免地引入错误，导致软件失效率升高，从而使得软件退化。当修改的成本变得难以被人接受时，软件就被抛弃。

④ 软件对硬件和环境有着不同程度的依赖性。这导致了软件移植的问题。

⑤ 软件的开发至今尚未完全摆脱手工作坊式的开发方式，生产效率低。

⑥ 软件是复杂的，而且以后会更加复杂。软件是人类有史以来生产的复杂度最高的工业产品。软件涉及人类社会的各行各业、方方面面，软件开发常常涉及其他领域的专门知识，因而对软件工程师提出了很高的要求。

⑦ 软件的成本相当昂贵。软件开发需要投入大量高强度的脑力劳动，成本非常高，风险也大。现在软件的开销已大大超过了硬件的开销。

⑧ 软件工作牵涉到很多社会因素。许多软件的开发和运行涉及机构、体制和管理方式等问题，还会涉及到人们的观念和心理。这些人为的因素常常成为软件开发的困难所在，直接影响到项目的成败。

嵌入式软件具有以下特点。

(1) 规模小,开发难度大

嵌入式软件的规模一般比较小,多数在几 MB 以内;但开发的难度大,需要开发的软件可能包括板级初始化程序、驱动程序、应用程序和测试程序等。嵌入式软件一般都要涉及到低层软件的开发,应用软件的开发也是直接基于操作系统的。这需要开发人员具有扎实的软、硬件基础,能灵活运用不同的开发手段和工具,具有较丰富的开发经验。

(2) 快速启动,直接运行

嵌入式软件需快速启动,上电后在几十秒内就会进入正常工作状态。为此,多数嵌入式软件事先已被固化在 NorFlash 等快速启动的主存中,上电后直接启动运行;或从 NorFlash 调入到内存(可能需要解压)后直接运行;或被存储在电子盘中,上电后快速调入到 RAM 中运行。

(3) 实时性和可靠性要求高

大多数嵌入式系统都是实时系统,有实时性和可靠性的要求。这两方面除了与嵌入式系统的硬件(如嵌入式微处理器的速度、访问存储器的速度和总线等)有关外,还与嵌入式系统的软件密切相关。

嵌入式实时软件对外部事件作出反应的时间必须要快,在某些情况下还需要是确定的、可重复实现的,不管当时系统内部状态如何,都是可预测的(predictable)。

嵌入式实时软件需要有处理异步并发事件的能力。在实际环境中,嵌入式实时系统处理的外部事件不是单一的,这些事件往往同时出现,而且发生的时刻也是随机的,即异步的。

嵌入式实时软件需要有出错处理和自动复位功能,应采用特殊的容错、出错处理措施,在运行出错或死机时能自动恢复先前的运行状态。

(4) 程序一体化

嵌入式软件是应用程序和操作系统两种软件的一体化程序。

(5) 两个平台

嵌入式软件的开发平台和运行平台各不相同,如图 3-1 所示。

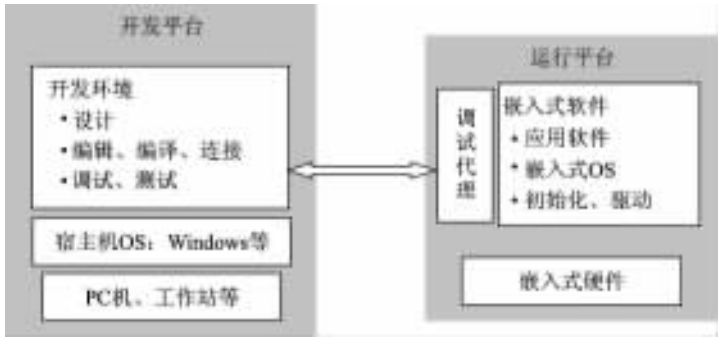


图 3-1　嵌入式软件的开发平台和运行平台

3.1.1 嵌入式软件分类

按通常的软件分类，嵌入式软件可分为系统软件、支撑软件和应用软件 3 大类。

- 系统软件　控制、管理计算机系统资源的软件，如嵌入式操作系统、嵌入式中间件(CORBA，JAVA)等。
- 支撑软件　辅助软件开发的工具软件，如系统分析设计工具、仿真开发工具、交叉开发工具、测试工具、配置管理工具和维护工具等。
- 应用软件　是面向特定应用领域的软件，如手机软件、路由器软件、交换机软件和飞行控制软件等。这里的应用软件除包括操作系统之上的应用外，还包括低层的软件，如板级初始化程序、驱动程序等。

按运行平台分类，嵌入式软件可分为运行在开发平台(如 PC 机的 Windows)上的软件和运行在目标平台上的软件。

- 运行在开发平台上的软件　设计、开发及测试工具等。
- 运行在目标平台即嵌入式系统上的软件　嵌入式操作系统、应用程序、低层软件及部分开发工具代理。

按嵌入式软件结构分类，嵌入式软件可分为循环轮询系统、前后台系统、单处理器多任务系统和多处理器多任务系统等几大类。具体见 1.1.3 节的描述。

3.1.2 嵌入式软件体系结构

嵌入式软件的体系结构如图 3－2 所示，包括驱动层、操作系统层、中间件层和应用层。

1. 驱动层

驱动层是直接与硬件打交道的一层，它对操作系统和应用提供所需驱动的支持。该层主要包括三种类型的程序：

- 板级初始化程序　这些程序在嵌入式系统上电后，初始化系统的硬件环境，包括嵌入式微处理器、存储器、中断控制器、DMA 和定时器等的初始化。
- 与系统软件相关的驱动　这类驱动是操作系统和中间件等系统软件所需的驱动程序，它们的开发要按照系统软件的要求进行。目前操作系统内核所需的硬件支持一般都已集成在嵌入式微处理器中了，因此操作系统厂商提供的内核驱动一般不用修改。开发人员主要需要编写的相关驱动程序有网络、键盘、显示和外存等的驱动程序。
- 与应用软件相关的驱动　与应用软件相关的驱动不一定需要与操作系统连接，这些驱动的设计和开发由应用决定。

2. 操作系统层

操作系统层包括嵌入式内核、嵌入式 TCP/IP 网络系统、嵌入式文件系统、嵌入式 GUI 系统和电源管理等部分。其中嵌入式内核是基础和必备的部分，其他部分要根据嵌入式系统的

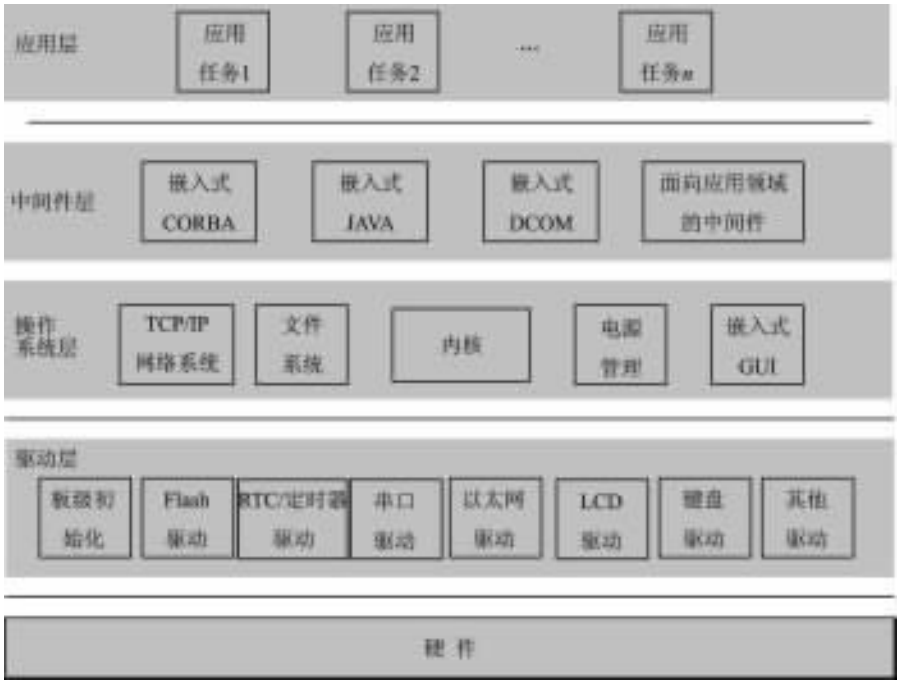


图 3－2　嵌入式软件体系结构

需要来确定。

3. 中间件层

目前在一些复杂的嵌入式系统中也开始采用中间件技术，主要包括嵌入式 CORBA、嵌入式 JAVA、嵌入式 DCOM 和面向应用领域的中间件软件，如基于嵌入式 CORBA 的应用于软件无线电台的应用中间件 SCA(Software Core Architecture)等。

4. 应用层

应用层软件主要由多个相对独立的应用任务组成，每个应用任务完成特定的工作，如 I/O 任务、计算的任务和通信任务等，由操作系统调度各个任务的运行。

3.1.3　嵌入式软件运行流程

基于多任务操作系统的嵌入式软件的主要运行流程如图 3－3 所示。该运行流程主要分为 5 个阶段。

1. 上电复位、板级初始化阶段

嵌入式系统上电复位后完成板级初始化工作。板级初始化程序具有完全的硬件特性，一般采用汇编语言实现。不同的嵌入式系统，板级初始化时要完成的工作具有一定的特殊性，但

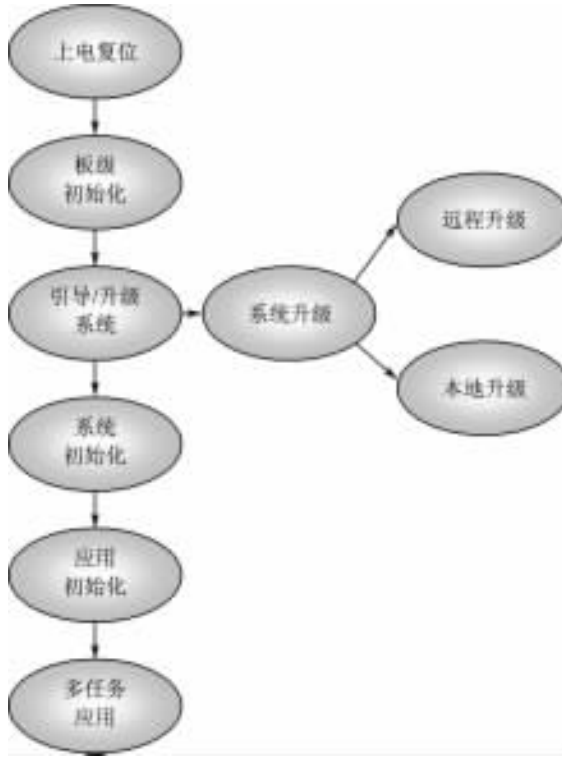


图 3-3　嵌入式软件执行的主要流程

以下工作一般是必须完成的。

- CPU 中堆栈指针寄存器的初始化；
- BSS 段(Block Storage Space，表示未被初始化的数据)的初始化；
- CPU 芯片级的初始化，中断控制器、内存等的初始化。

2. 系统引导/升级阶段

根据需要分别进入软件系统引导阶段或系统升级阶段。软件可通过测试通信端口数据或判断特定开关的方式分别进入不同阶段。

(1) 系统引导阶段

系统引导有以下几种情况：

- 将系统软件从 NorFlash 中读取出来加载到 RAM 中运行。这种方式可以解决成本及 Flash 速度比 RAM 慢的问题。软件可压缩存储在 Flash 中。
- 将软件从外存(如 DOC，CF 卡和 NandFlash 等)中读取出来加载到 RAM 中运行。这种方式的成本更低。
- 不需将软件引导到 RAM 中，而是让其直接在 NorFlash 上运行，进入系统初始化阶段。

(2) 系统升级阶段

进入系统升级阶段后,系统可通过网络进行远程升级或通过串口等进行本地升级。远程升级一般支持 FTP,HTTP 等方式;本地升级可通过 Console 口,使用超级终端或特定的升级软件进行。

3. 系统初始化阶段

在该阶段进行操作系统等系统软件各功能部分所必需的初始化工作,如根据系统配置初始化数据空间、初始化系统所需的接口和外设等。系统初始化阶段需要按特定顺序进行,如首先完成内核的初始化,然后完成网络、文件系统等的初始化,最后完成中间件等的初始化工作。

4. 应用初始化阶段

在该阶段进行应用任务的创建,信号量、消息队列等的创建和与应用相关的其他初始化工作。

5. 多任务应用阶段

各种初始化工作完成后,系统进入多任务状态,操作系统按照已确定的算法进行任务的调度,各应用任务分别完成特定的功能。

3.2　嵌入式操作系统

嵌入式操作系统就是应用于嵌入式系统的操作系统,其产品出现于 20 世纪 80 年代初。经过 20 多年的发展,到目前为止国际市场上已出现了几十种嵌入式操作系统。

近 10 年来,嵌入式操作系统得到飞速的发展,从支持 8 位微处理器到 16 位、32 位甚至 64 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 网络系统和窗口图形系统等。同时,嵌入式操作系统的品种也在不断地变化:早期嵌入式系统应用领域有限,嵌入式操作系统品种比较少,一般没有考虑特定应用领域的需求;随着嵌入式系统应用领域的扩展,目前嵌入式操作系统的市场在不断细分,出现了针对不同领域的产品,这些产品按领域的要求和标准提供特定的功能。

从应用的角度来看,嵌入式操作系统可以分为:

- 面向航空电子的嵌入式操作系统;
- 面向智能手机的嵌入式操作系统,如 SymbianOS,PalmOS,Smartphone2003,Embedded Linux 等;
- 面向数字电视的嵌入式操作系统;
- 面向通信设备的嵌入式操作系统;
- 面向汽车电子的嵌入式操作系统;
- 面向工业控制的嵌入式操作系统。

从实时性的角度来看，嵌入式操作系统可以分为：

- 嵌入式实时操作系统　具有强实时特点，如 VxWorks，QNX，Nuclear，OSE，DeltaOS 和各种 ItronOS 等；
- 非实时嵌入式操作系统　一般只具有弱实时特点，如 Win CE、版本众多的嵌入式 Linux 和 PalmOS 等。

3.2.1　体系结构

操作系统是硬件与应用之间的一层软件，负责管理整个系统，同时将硬件细节与应用隔离开来，为应用提供一个更容易理解和进行程序设计的接口。体系结构是操作系统的基础，它定义了硬件与软件的界限、内核与操作系统其他组件（文件、网络、GUI 等）的组织关系以及系统与应用的接口。操作系统的体系结构是确保系统的性能、可靠性、灵活性、可移植性和可扩展性的关键，就好比房子的梁架，只有梁架搭牢固了才提得上房子的质量，再做一些锦上添花的工作才有意义。

目前操作系统的体系结构可分为：单块结构、层次结构和客户/服务器（微内核）结构。

1. 单块结构

单块结构的操作系统由许多模块组成，这些模块之间可以相互调用（如图 3－4 所示）。在这样的操作系统中通常有两种工作模式，即系统模式和用户模式。这两种模式有不同的执行权限和不同的执行空间。在用户模式下，系统空间受到保护，并且有些操作是受限制的，例如 I/O操作和一些特殊指令。而在系统模式下可以访问用户空间，可以执行任何操作。运行在用户模式的应用程序可以通过系统调用进入系统模式，完成操作后再返回用户模式。

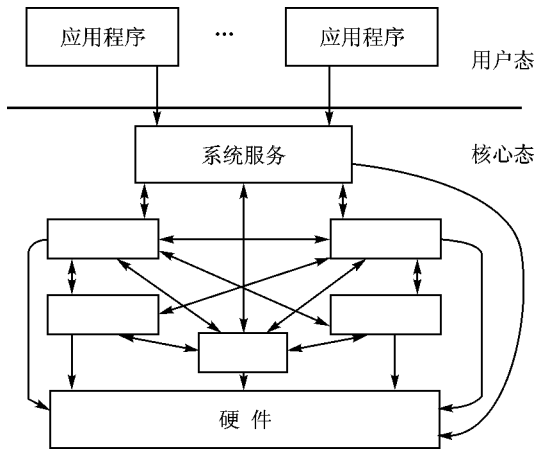


图 3－4　单块结构的操作系统

这种结构的操作系统很难调试和维护。如果一个模块被改变，则对其他模块的影响可能是很大的；如果一个模块被修正，则可能会在其他模块中出现其他的 BUG。其包含的模块越多，以及模块间的相互连接越多，软件就会因为多重连接变得更加混乱。

2. 层次结构

层次结构操作系统提供了“环”管理机制，即第 n 环的程序无权修改第 $n-1$ 环的数据，从而提高了操作系统的安全性。例如在著名的 OSI 层次结构中，不能跳过任何一层，因此可以很容易地在不影响其他层的情况下替换其中的一层。

但是不管什么样的 OS 技术，出于性能方面的考虑，各层之间不会像 OSI 那样正交。一个系统调用可以直接到达每一层，在 RTOS 中甚至可以直接到达硬件。

请看图 3－5 所展示的 MS－DOS 的结构，它给出了一个有代表性的良好组织的 OS 结构。然而重画该图（见图 3－6），就能够理解事物是怎样被真正地组织起来的：它确实是一个分层结构，但是一个应用可以直接访问 BIOS 甚至是硬件。

嵌入式操作系统多数是以这种方式设计的。

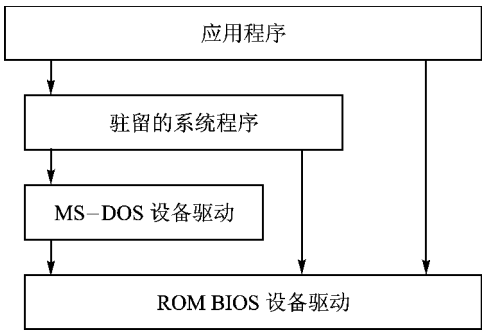


图 3－5　MS－DOS 系统结构一

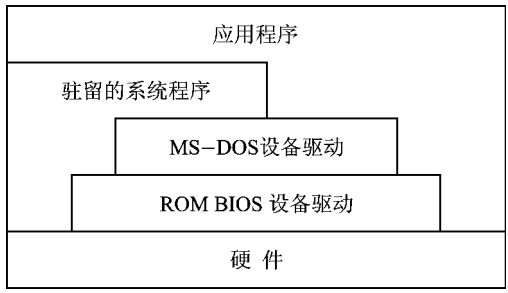


图 3－6　MS－DOS 系统结构二

3. 客户/服务器（微内核）结构

客户/服务器结构（或称微内核结构）的操作系统如图 3－7 所示，只有一个很小的内核——微内核，以完成任务管理、任务调度和通信等基本功能，而把许多其他功能作为服务器实现为系统任务或进程，运行于用户模式，不再像一个完整的操作系统那样仍然将它们作为微内核的一部分。用户任务作为客户，它们通过系统调用发出请求，服务器响应请求，微内核仅完成它们之间的通信、同步和任务调度等基本功能。

这种体系结构进一步提高了操作系统的模块化程度，使其结构更清晰，使系统更加易于调试、扩充和剪裁（扩充和剪裁相当于服务器的增加或删减）；同时由于和目标硬件相关的部分被放到微内核的底层部分和驱动程序中实现，因此很容易实现不同硬件平台之间的移植。另外，每个服务器在用户态运行，有自己的存储空间，一个服务器出错不会影响到整个内核，从而增

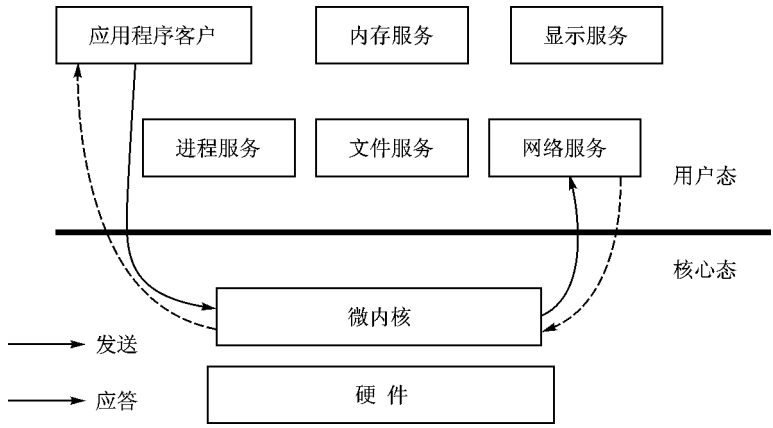


图 3-7 客户/服务器结构的操作系统

强了系统的健壮性。

总的来说,客户/服务器(微内核)结构的操作系统可以有如下好处:

- 统一的界面——对内核和用户层服务都一样;
- 可扩展性——能够加入新的服务;
- 可剪裁与可配置性——能够减去或修改服务;
- 可移植性——与硬件相关的特定部分很小,更容易被移植;
- 可靠性——更小的内核使得它的实现更为可靠;
- 分布式系统支持——某些功能块可以位于不同的机器上;
- 面向对象——以面向对象的方式实现和使用操作系统。

客户/服务器(微内核)结构的缺点也是非常明显的,主要体现在性能方面。虽然微内核能最小化因关中断或其他原因造成的延迟,但是通信和上下文切换的开销将大大增加。如果实施了存储保护,那么服务器任务必须被保护;每当一个服务被请求时,系统必须从应用的存储空间切换到服务器的存储空间;当任务之间实施了保护时,从一个任务切换到另一个任务的时间也会增加。

目前嵌入式实时操作系统主要采用分层和模块化相结合的结构或微内核结构。分层和模块化结合的结构将操作系统分为硬件无关层、硬件抽象层和硬件相关层,每层再划分功能模块。这样移植工作便集中在硬件相关层,与其余两层无关;功能的伸缩则集中在模块上,从而确保其具有良好的可移植性和可伸缩性。而采用微内核结构,则可利用其可伸缩的特点适应硬件的发展,便于扩展。

举 例

(1) DeltaCORE 的体系结构

嵌入式实时内核 DeltaCORE 的设计采用层次+模块的结构。它从上到下有 3 个层次:

① 最上层是应用，通过系统调用（应用编程接口）使用内核的服务；

② 中间是实时内核，为实时应用提供任务管理，同步、通信与互斥机制以及中断和内存管理等服务；

③ 硬件抽象层向上对内核提供简化抽象的硬件操作，向下操作具体的目标硬件，便于DeltaCORE在多种硬件平台上移植。

DeltaCORE层次＋模块结构如图3－8所示。

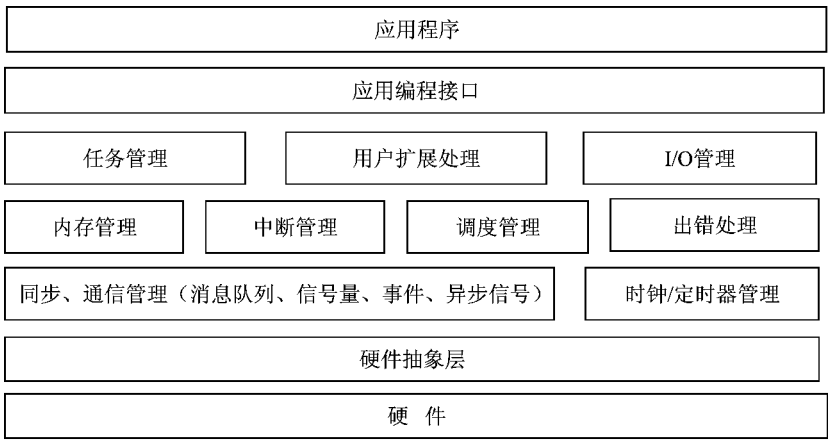


图3－8　DeltaCORE层次＋模块结构

（2）QNX 4.25的体系结构

嵌入式实时操作系统QNX 4.25采用客户/服务器结构，由一个微内核和可选的协作进程组成。微内核只实现了核心的服务，如调度和派遣、第一级中断处理和IPC（进程间通信）的路由。

内核的附加功能在协作进程中实现。它们作为服务器进程，响应客户进程（例如应用进程）的请求。服务器进程的实例包括文件系统管理器、进程管理器、设备管理器和网络管理器等。在386以上的处理器中，内核运行在特权级0；管理器和设备驱动运行在特权级1和2（为了执行I/O操作）；应用进程运行在特权级3，因此只能够执行处理器的通用指令。比起内核、设备驱动和应用都运行在特权级0的情况，这种特权保护机制使得系统更加健壮。

QNX系统结构如图3－9所示。

QNX 4.25提供了进程间通信（IPC）的基本机制。消息传递服务是基于客户/服务器模型的。客户（例如一个应用进程）发送一个消息给一个服务器（例如设备管理器），后者给予回答。QNX 4.25的许多API调用都使用了消息通信机制。例如，当一个应用进程需要打开一个文件时，该系统调用被翻译成一个发送给文件系统管理器的消息。文件管理器在通过其设备驱动程序访问磁盘之后，回答以一个文件句柄。

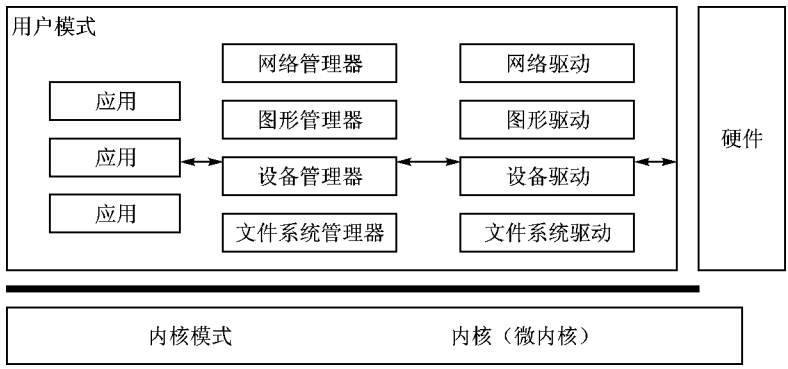


图 3-9　QNX 系统结构

QNX 4.25 的网络管理器是对操作系统消息通信能力的一个强大的扩展。网络管理器通过向远程机器传播消息，加强了 QNX 消息传递 IPC 的功能，所有这些对应用来说都是完全透明的。从应用的观点来看，另一个进程位于同一个机器上与位于一个远程机器上没什么两样。当消息被发送给一个远程机器上的进程时，微内核将消息传递给网络管理器（而不是给目的进程），后者通过网络把消息传输给它的目的进程。

3.2.2　功能及特点

嵌入式操作系统一般由内核、嵌入式 TCP/IP 网络系统和嵌入式文件系统等组成。

1. 内　核

内核是嵌入式操作系统的基础，也是必备的部分。它提供任务管理，内存管理，通信、同步与互斥机制，中断管理，时间管理及任务扩展等功能。内核还提供特定的应用编程接口，但目前没有统一的标准。

（1）任务管理

任务的管理是内核的核心部分，具有任务调度、创建任务、删除任务、挂起任务、解挂任务和设置任务优先级等功能。

通用计算机的操作系统（以下简称通用操作系统）追求的是最大的吞吐率。为了达到最佳整体性能，其调度原则是公平，采用 Round-Robin 或可变优先级调度算法，调度时机主要以时间片为主驱动。而嵌入式操作系统多采用基于静态优先级的可抢占的调度，任务优先级是在运行前通过某种策略静态分配好的，一旦有优先级更高的任务就绪，就马上进行调度。

（2）内存管理

嵌入式操作系统的内存管理比较简单，通常不采用虚拟存储管理，而采用静态内存分配和动态内存分配（固定大小内存分配和可变大小内存分配）相结合的管理方式。有些内核利用 MMU 机制提供内存保护功能。

通用操作系统广泛使用了虚拟内存的技术，为用户提供一个功能强大的虚存管理机制。由于虚存机制引起的缺页、调页现象会给系统带来不确定性且需要比较多的资源，因此嵌入式系统很少或有限地使用虚存技术，一般采用固定分区和堆的动态内存分配方式。这种方式的优点是系统具有较好的可预测性，开销小。

(3) 通信、同步和互斥机制

这些机制提供任务间以及任务与中断处理程序间的通信、同步和互斥功能，一般包括信号量、消息、事件、管道、异步信号和共享内存等功能。

与通用操作系统不同的是，嵌入式操作系统需要解决在这些机制的使用中出现的优先级反转问题。

(4) 中断管理

中断管理一般具有以下功能：

- 安装中断服务程序；
- 中断发生时，对中断现场进行保存，并且转到相应的服务程序上执行；
- 中断退出前，对中断现场进行恢复；
- 中断栈切换；
- 中断退出时的任务调度。

为方便中断处理程序的开发，中断管理负责管理中断控制器，负责中断现场保护和恢复，用户的中断处理程序只需处理与特定中断相关的部分，并按一般函数的格式编写中断处理程序。

为防止堆栈的溢出，提高系统的可靠性，专门设置中断栈；一旦进入中断就切换到中断栈，退出中断时再进行堆栈的切换。

通用操作系统的中断处理比较复杂，需要采用专门的开发包开发中断处理程序，而嵌入式系统的中断处理无需专门的开发包。

(5) 时间管理

时间管理提供高精度、应用可设置的系统时钟。该时钟是嵌入式系统的时基，可设置为10 ms以下。时间管理还提供日历时间，负责与时间相关的任务管理工作，如任务对资源有限等待的计时、时间片轮转调度等，提供软定时器的管理功能等。

通用操作系统的系统时钟的精度由操作系统确定，应用不可调，且一般是几十个 ms。

(6) 任务扩展功能

嵌入式系统的应用领域非常广，任何一个嵌入式操作系统都不可能面面俱到，提供完善的功能。任务扩展功能就是在内核中设置一些 Hook 的调用点，在这些调用点上内核调用应用设置的、应用自己编写的扩展处理程序，以扩展内核的有关功能。

Hook 调用点有任务创建、任务切换、任务删除和出错处理等。

2. 嵌入式 TCP/IP 网络系统

TCP/IP 协议已经广泛地应用于嵌入式系统中，嵌入式 TCP/IP 网络系统提供符合 TCP/IP 协议标准的协议栈，提供 Socket 编程接口，如图 3－10 所示。

图 3－10 TCP/IP 协议

嵌入式 TCP/IP 网络系统具有以下特点。

（1）可剪裁

由于嵌入式应用的要求千差万别，各种嵌入式应用对网络系统的要求也不尽相同，且嵌入式系统对产品的成本、功耗比较敏感，因此嵌入式 TCP/IP 网络系统必须提供可剪裁的机制，能根据嵌入式系统的功能的要求选择所需的协议，对完整的 TCP/IP 协议簇进行剪裁，以满足用户的需要。

（2）采用“零拷贝”技术提高实时性

由于 TCP/IP 协议的层次特性，每个协议层次都有自己的数据格式。发送数据时，各个协议层从自己的上层协议接收数据，加上本层的控制信息后再交给自己的下层协议，这个过程叫做打包。其中的控制信息只有其他主机上的相同协议层才能正确解释。接收数据时，各个协议层从自己的下层协议接收数据，取出本层的控制信息后再把剩下的部分数据交给上层的协议，这个过程叫做拆包。

用户数据在从本地主机传输到远地主机的过程中，需要不断地打包和拆包。如果各协议层之间均采用数据拷贝进行数据传递，则将大大增加系统开销，从而降低系统性能。在嵌入式 TCP/IP 网络系统中，普遍采用“零拷贝”(zero copy)技术以解决该问题。所谓“零拷贝”技术，是指 TCP/IP 协议栈没有用于各层间数据传递的缓冲区，协议栈各层间传递的都是数据指针，只有当数据最终要被驱动程序发送出去或是被应用程序取走时，才进行真正的数据搬移。

（3）可扩展

由于嵌入式系统的多样性，使得网络接口多样化、网络应用多样化，这就要求 TCP/IP 网络系统提供便于扩展的网络接口，方便不同驱动程序和网络应用的开发；并且根据实际的需要，可以添加新的协议模块到 TCP/IP 网络系统中，实现对新的网络协议的支持。

（4）采用静态分配技术

如果在网络发送或接收的过程中，某一次传送的数据超过了在一个物理网络上能够传输的最大数据量(MTU)，则处理该数据的任务往往会阻塞等待，直到上层重新调整需要处理的数据量的大小，它才会继续执行下去。

嵌入式 TCP/IP 网络系统通常采用静态分配技术，在网络初始化时就静态分配通信缓冲区，设置了专门的发送和接收缓冲（其大小一般小于或等于物理网络上的 MTU 值），从而确保了每次发送或接收时处理的数据不会超过 MTU 值，也就避免了数据处理任务的阻塞等待。

3. 嵌入式文件系统

通用操作系统的文件系统通常具有以下功能：

- 提供用户对文件操作的命令；
- 提供用户共享文件的机制；
- 管理文件的存储介质；
- 提供文件的存取控制机制，保障文件及文件系统的安全性；
- 提供文件及文件系统的备份和恢复功能；
- 提供对文件的加密和解密功能。

嵌入式文件系统相比之下较为简单，主要具有文件的存储、检索和更新等功能，一般不提供保护和加密等安全机制。它以系统调用和命令方式提供对文件的各种操作，主要有：

- 设置、修改对文件和目录的存取权限；
- 提供建立、修改、改变和删除目录等服务；
- 提供创建、打开、读、写、关闭和撤消文件等服务。

此外，嵌入式文件系统还具有以下特点。

（1）兼容性

嵌入式文件系统通常支持几种标准的文件物理结构，如 FAT16，FAT32 等。

（2）实时文件系统

除支持标准的文件物理结构外，为提高实时性，有些嵌入式文件系统还支持自定义的实时文件系统，这些文件系统一般采用连续文件的方式存储文件。目前实时文件系统还没有形成国际标准。

（3）可剪裁、可配置

可根据嵌入式系统的要求选择所需的文件物理结构，如只选择 FAT16；可选择所需的存储介质，配置可同时打开的最大文件数等。

（4）支持多种存储设备

嵌入式系统的外存形式多样，嵌入式文件系统需方便地挂接不同存储设备的驱动程序，具备灵活的设备管理能力。同时根据不同外存的特点，嵌入式文件系统还需考虑其性能、寿命等因素，发挥不同外存的优势，提高存储设备的可靠性和使用寿命。

3.2.3　发展趋势

嵌入式操作系统今后的主要发展趋势有以下 4 点。

1. 形成行业的标准

目前一些行业已经开始定义其相关的嵌入式操作系统行业标准，如汽车电子的 OSEK/VDX、航空电子的 ARINC 653 等。根据应用的不同要求，今后不同行业都会定义其嵌入式操作系统的行业标准。

2. 向高可用和高可靠方向发展

采用可靠性保证措施和保证技术开发出稳定可靠的操作系统，如按照 DO—178B 标准开发操作系统，并通过其测试。

在一些高可用(high available)、高可靠(high reliability)的嵌入式操作系统中，利用 MMU 技术实现操作系统与应用程序的隔离，以及应用程序与应用程序之间的隔离，以防止应用程序破坏操作系统的代码和数据。对于应用程序来讲，也可以防止别的应用程序对自己的非法入侵，避免破坏应用程序自身的运行。

3. 适应不同的嵌入式硬件平台

嵌入式操作系统的体系结构采用分层和模块化结构或微内核结构。分层和模块化结构将操作系统分为硬件无关层、硬件抽象层和硬件相关层，每层再划分功能模块，这样移植工作便集中在硬件相关层，与其余两层无关；而功能的伸缩则集中在模块上，从而确保了系统具有良好的可移植性和可伸缩性。采用微内核结构，利用其可伸缩的特点适应硬件的发展，便于扩展。

4. 功能丰富

嵌入式操作系统的功能越来越丰富，不仅能提供一些基本的功能，如内核、网络、GUI、文件系统和电源管理等，而且还会具有很多新的功能。

同时，嵌入式操作系统需具有可剪裁、可配置的特点。只有量体裁衣，去除冗余，才能更好地发挥嵌入式硬件的效率，降低成本，提高竞争力。

3.3 嵌入式软件开发工具

嵌入式软件开发工具是嵌入式支撑软件的核心，它的集成度和可用性将直接关系到嵌入式系统的开发效率。嵌入式软件开发工具包括系统分析设计工具、仿真开发工具、交叉开发工具、测试工具、配置管理工具和维护工具等。本节重点讲述开发工具的分类、交叉开发工具的基本概念及交叉调试技术等。

3.3.1 嵌入式软件开发工具的分类

从开发步骤来看，嵌入式软件的开发和一般软件开发一样，主要分为以下几个阶段(如图 3－11 所示)。

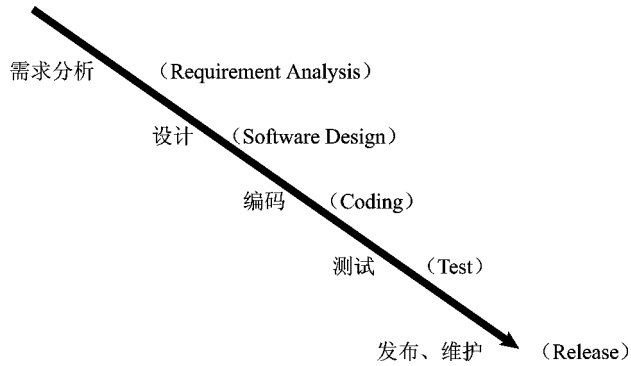


图 3－11　嵌入式软件开发阶段

根据不同的阶段，嵌入式软件开发工具可以分为：

- **需求分析工具**（Requirement Analysis Tools）；
- **软件设计工具**（Software Design Tools）；
- **编码、调试工具**（Coding Tools）；
- **测试工具**（Testing Tools）。

国内外主要的嵌入式软件开发工具产品如图 3－12 **所示。一个完整的嵌入式软件工具集是覆盖嵌入式软件开发过程各个阶段的工具集合，并且以集成开发环境**（IDE）**的方式提交。**

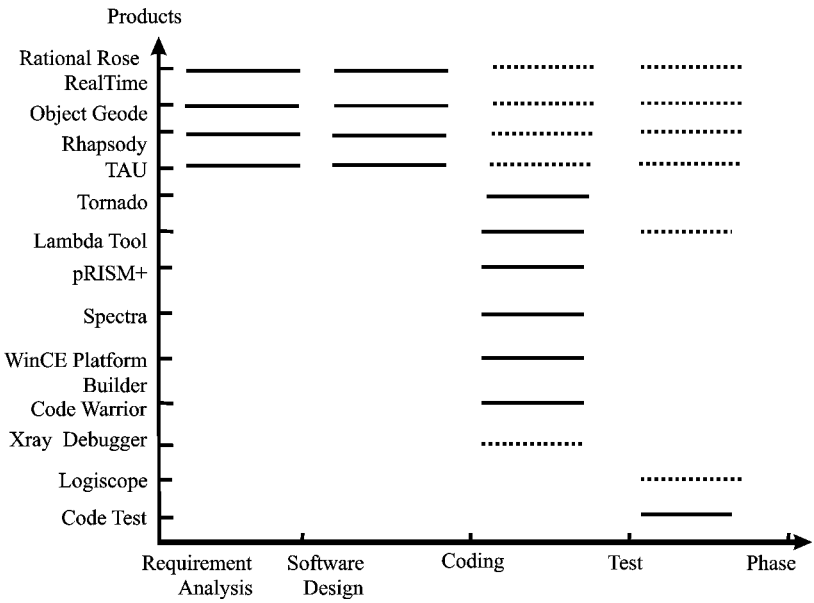


图 3－12　嵌入式软件开发工具一览

目前大多数厂商提供的工具集中在实现(编码、编译和调试)工具上,这些厂商主要包括芯片厂商、嵌入式操作系统厂商及工具厂商。在设计和测试方面的工具较少,由于这些工具与嵌入式软件的开发管理和质量保证密切相关,因此随着嵌入式软件复杂度的提高和嵌入式系统上市时间的缩短,嵌入式系统开发者非常需要设计、测试方面的工具,以提高其开发的水平,降低项目的风险。

另外,嵌入式软件的开发可以分为以下几种:

- 编写简单的板级测试软件,主要是辅助硬件的调试;
- 开发基本的驱动程序;
- 开发特定嵌入式操作系统的驱动程序(有时称为板级支持包 BSP);
- 开发嵌入式系统软件,如嵌入式操作系统、中间件等;
- 开发应用软件。

从以上嵌入式软件开发的分类来看,嵌入式软件开发工具可以分为:

- 与嵌入式 OS 相关的开发工具,用于开发基于嵌入式 OS 的应用和部分驱动程序等;
- 与嵌入式 OS 无关的开发工具,用于开发基本的驱动程序、辅助硬件调试的软件和系统软件等。

3.3.2 嵌入式软件的交叉开发环境

嵌入式软件的开发通常采用交叉开发环境。它是指用于嵌入式软件开发的所有工具软件的集合,一般包括文本编辑器、交叉编译器、交叉调试器、仿真器和下载器等工具。从开发方式来看,交叉开发环境由宿主机和目标机组成。宿主机与目标机之间在物理连接的基础上建立起逻辑连接。

1. 宿主机

宿主机是用于开发嵌入式系统的计算机。从硬件配置来讲,它们一般为通用的 PC 机、工作站,其上的软件配置很丰富,除了功能强大的桌面操作系统外,还具备各种开发工具,为编辑、编译、链接、调试、测试及固化嵌入式应用软件提供全过程的支持。

2. 目标机

目标机即所开发的嵌入式系统,是嵌入式软件的运行环境。目标机一般都是裸机,没有任何软件资源。目标机的嵌入式操作系统是用于支撑嵌入式应用的,不是用于开发的环境平台。在开发过程中,目标机端须接收和执行宿主机发出的各种命令,如设置断点、读内存和写内存等,将结果返回给宿主机,配合宿主机各方面的工作。

断点主要功能是使程序在某个指定的地方停下来,以便观察各种有用的信息,如寄存器的值、某个变量的值和某个内存单元的值等。断点可分为硬件断点和软件断点两种:

- 硬件断点　需要使用 CPU 硬件支持来实现的断点被称为硬件断点。硬件断点主要用于调试 ROM/Flash 中的程序和监控程序对变量的访问。

- **软件断点**　调试器通过将指令替换为断点中断指令(如 X86 处理器中的 INT 3)来实现的断点被称为软断点。这种断点只用于在 RAM 中运行的程序代码。

3. 物理连接和逻辑连接

物理连接是指宿主机与目标机上的一定物理端口通过物理线路连接在一起。其连接方式主要有三种:串口、以太口和 OCD(On Chip Debug)方式(如 JTAG,BDM)等。物理连接是逻辑连接的基础。

逻辑连接指宿主机与目标机间按某种通信协议建立起来的通信连接,目前逐步形成了一些通信协议的标准。

要顺利地建立起交叉开发环境,需要正确设置这两种连接,缺一不可。在物理连接上,要注意使硬件线路正确连接,且硬件设备完好,能正常工作,连接线路的质量要好。逻辑连接在于正确配置宿主机和目标机的物理端口的参数,并且与实际的物理连接一致。

3.3.3　嵌入式软件实现阶段的开发过程

设计完成后,嵌入式软件的开发进入实现阶段(编码、编译连接和调试阶段)。在这个阶段的开发可分为三个步骤:生成、调试和固化运行。

软件的生成主要是在宿主机上进行。开发人员利用各种工具完成对应用程序的编辑、交叉编译和链接工作,生成可供调试或固化的目标程序。

调试是通过交叉调试器完成软件的调试工作。调试完成后还需进行必要的测试工作,测试完成后进入到最后的固化运行阶段。

固化运行是先用一定的工具将应用程序固化到目标机上,然后启动目标机,在没有任何工具干预的情况下应用程序能自动地启动运行。

1. 嵌入式软件的生成

如图 3-13 所示,嵌入式软件的生成又可以分为三个阶段:源代码程序的编写;将源程序交叉编译成各个目标模块;将所有目标模块及相关的库文件一起链接成可供下载调试或固化的目标程序。

这一过程看似与普通计算机软件的开发过程一样,但其中有本质的区别,关键就在于交叉编译器和交叉链接器上。

交叉编译器的主要功能是把在宿主机上编写的高级语言程序编译成可以运行在目标机上的代码,即在宿主机上能够编译生成另一种 CPU(嵌入式微处理器)上的二进制程序。不同目标机的处理器所对应的编译器不尽相同。为了提高编译质量,硬件厂商针对自己开发的处理器的特性定制编译器,既提供对高级编程语言的支持,又能够很好地对目标代码进行优化。

嵌入式软件的运行方式主要有两种:调试方式和固化方式。不同方式下程序代码或数据在目标机内存中的定位有所不同。宿主机上提供一定的工具或手段对目标程序的运行方式和

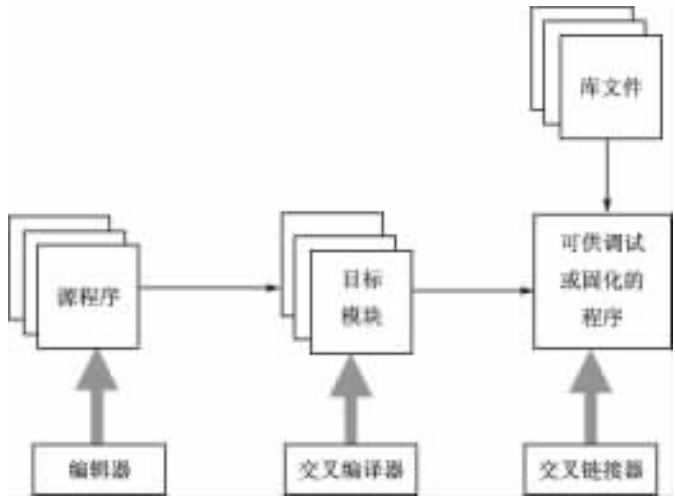


图 3-13　嵌入式软件的生成阶段

内存定位进行选择和配置，链接器再根据这些配置信息将目标模块和库文件中的模块链接成目标程序，因此称这样的链接器为交叉链接器。同目标模块相链接的运行库也是与嵌入式应用相关的。

2. 嵌入式软件的调试

在开发嵌入式软件时，交叉调试是必不可少的一步。嵌入式软件的特点决定了其调试具有如下特点：

- 一般情况下调试器(debugger)和被调试程序(debugged)运行在不同的计算机上。调试器主要运行在宿主机上，而被调试程序运行在目标机上。
- 调试器通过某种通信方式与目标机建立联系。通信方式可以是串口、并口、网络、JTAG 或者专用的通信方式。
- 一般在目标机上有调试器的某种代理(agent)，这种代理能配合调试器一起完成对目标机上运行的程序的调试。这种代理可以是某种软件，也可以是某种支持调试的硬件等。
- 目标机也可以是一种虚拟机。在这种情形下，似乎调试器和被调试程序运行在同一台计算机上。但是调试方式的本质没有变化，即被调试程序都是被下载到了目标机，调试并不是直接通过宿主机操作系统的调试支持来完成的，而是通过虚拟机代理的方式来完成的。

因此，交叉调试可以被这样定义：交叉调试器是指调试程序和被调试程序运行在不同机器上的调试器；调试器通过某种方式能控制目标机上被调试程序的运行方式，并且通过调试器能查看和修改目标机上的内存、寄存器以及被调试程序中的变量等。

交叉调试与非交叉调试的比较如表 3－1 所列。

表 3－1　交叉调试与非交叉调试的比较

交叉调试	非交叉调试
调试器和被调试程序运行在不同的计算机上	调试器和被调试程序运行在同一台计算机上
可独立运行，无需操作系统支持	需要操作系统的支持
被调试程序的装载由调试器完成	被调试程序的装载由专门的 Loader 程序完成
需要通过外部通信的方式来控制被调试程序	不需要通过外部通信的方式来控制被调试程序
可以直接调试不同指令集的程序	只能直接调试相同指令集的程序

交叉调试的方式即调试器控制被调试程序运行的方式有很多种，一般可以分为：

- Crash & Burn 方式；
- Rom Monitor 方式；
- Rom Emulator 方式；
- ICE(In Circuit Emulator)方式；
- OCD(On Chip Debugging)方式。

目前常用的方式是 Rom Monitor 方式和 OCD 方式。

(1) Crash & Burn 方式

最初的调试方式被称为“Crash and Burn”。利用该方式开发嵌入式程序的过程如下：

① 编写代码；

② 反复地检查代码，直到编译通过；

③ 将程序固化(即 Burn)到目标机上的非易失性存储器(如 E^2PROM，Flash 等)中；

④ 观察程序是否正常工作；

⑤ 如果程序不能正常工作(即 Crash)，则反复检查代码，查找问题的根源；

⑥ 改写代码；

⑦ 重复③～⑥，直到程序正常工作。

显然，这种调试方式对于开发人员而言是非常辛苦的，并且开发效率很低。如果比较幸运，或许还能从目标机打印一些有用的提示信息(例如从监视器、LCD 或串口等输出信息)；档次低一点的目标机就只能通过 LED 灯或者示波器等辅助设备进行调试。因此在调试程序时，不得不经常地“Burn”—“Crash”—“Burn”—“Crash”…。这种调试方式的难度是可想而知的。

(2) Rom Monitor 方式

如图 3－14 所示，在 ROM Monitor 调试方式下，调试环境由三部分构成，即宿主机端的调试器、目标机端的监控器(监控程序，ROM Monitor)以及二者间的连接(包括物理连接和逻辑连接)。

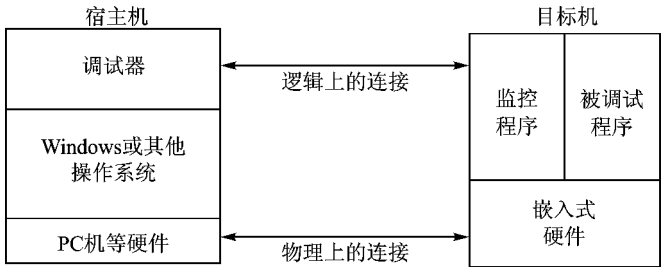


图 3-14　嵌入式程序的 ROM Monitor 调试方式

调试器一般都支持源码级调试，高级的还支持任务级调试。

ROM Monitor 是运行在目标机上的一段程序。它负责监控目标机上被调试程序的运行，通常和宿主机端的调试器一起完成对应用程序的调试。ROM Monitor 预先被固化到目标机的 ROM 空间中，在目标机复位后首先执行的就是 ROM Monitor 程序。它对目标机进行一些必要的初始化，然后初始化自己的程序空间，最后就等待宿主机端的命令。ROM Monitor 能配合调试器完成被调试程序的下载、目标机内存和寄存器的读/写、设置断点以及单步执行被调试程序等功能。一些高级的 ROM Monitor 能配合完成代码分析(code profiling)、系统分析(system profiling)、ROM 空间的写操作以及设置各种非常复杂的断点等功能。

利用 ROM Monitor 方式作为调试手段时，开发应用程序的步骤如下：

① 启动目标机，监控器掌握对目标机的控制，等待与调试器建立连接。

② 调试器启动，与监控器建立起通信连接。

③ 调试器将应用程序下载到目标机上的 RAM 空间中。

④开发人员使用调试器进行调试，发出各种调试命令，监控器解释并执行这些命令；监控器通过目标机上的各种异常来获取对目标机的控制，将命令执行结果回传给调试器。

⑤ 如果程序有问题，则开发人员在调试器的帮助下定位错误；修改之后再重新编译链接并下载程序，开始新的调试，如此反复直至程序正确运行为止。

用 ROM Monitor 的方式明显提高了调试程序的效率，减小了调试的难度，缩短了产品开发的周期，有效地降低了开发成本。

ROM Monitor 调试方式的最大好处就是简单、方便。它还可以支持许多高级的调试功能，可扩展性强，成本低廉，基本上不需要专门的调试硬件支持。

但是 ROM Monitor 也具有较多缺点：

- 开发 ROM Monitor 的难度比较大；
- 当 ROM Monitor 占用 CPU 时，应用程序不响应外部的中断，因此不便于调试有时间特性的程序；
- 当目标机的 CPU 不支持硬件断点(hardware breakpoint)时，ROM Monitor 无法调试

ROM 程序和设置数据断点(data breakpoint,即能够监视对数据的读/写的断点);

- ROM Monitor 要占用目标机一定数量的资源,如 CPU 资源、RAM 资源和通信设备(如串口、网卡等)资源;
- 在调试时 ROM Monitor 已经为应用程序建立了运行环境,因此在一定程度上造成应用程序的最终运行环境和调试环境的差异,如程序初始化部分的代码、内存空间分配等都与最终运行环境不同。

虽然 ROM Monitor 有如此之多的缺点,但 ROM Monitor 仍然是一种应用相当广泛的调试方式,几乎所有的交叉调试器都支持这种方式。

(3) ROM Emulator 方式

ROM Emulator 从一定程度上讲,可以被认为是一种用于替代目标机上的 ROM 芯片的设备,即 ROM 仿真器。利用这种设备,目标机可以没有 ROM 芯片,但目标机的 CPU 可以读取 ROM Emulator 设备上的 ROM 芯片的内容,因为 ROM Emulator 设备上的 ROM 芯片的地址可以实时地映射到目标机的 ROM 地址空间,从而仿真(emulation)目标机的 ROM。

实质上 ROM Emulator 是一种不完全的调试方式。通常 ROM Emulator 设备只是为目标机提供 ROM 芯片,并在目标机和宿主机间建立一条高速的通信通道,因此它经常和前面两种调试方式结合起来形成一种完备的调试方式。ROM Emulator 的典型应用就是和 ROM Monitor 的调试方式相结合。

这种调试方式的最大优点就是目标机可以没有 ROM 芯片,却可以使用 ROM Emulator 提供的 ROM 空间,并且不需要用别的工具来写 ROM。其缺点是目标机必须能支持外部 ROM 存储空间,并且由于其通常要和 ROM Monitor 配合使用,因此它也拥有 ROM Monitor 的所有缺点。现在大多数目标板都提供 ROM 芯片并且支持在板上直接对 ROM 空间进行快速的写操作,所以这种 ROM Emulator 属于被淘汰的调试方式。

(4) ICE 方式

ICE(In Circuit Emulator)即在线仿真器,是一种用于替代目标机上的 CPU 的设备。ICE 的 CPU 是一种特殊的 CPU(被称为"bond-out")。它可以执行目标机 CPU 的指令,但它比一般的 CPU 有更多的引出线,能够将内部的信号输出到被控制的目标机。ICE 上的内存也可以被映射到用户的程序空间,这样即使在目标机不存在的情形下,也可以进行代码的调试。

在连接 ICE 和目标机时,一般是将目标机的 CPU 取下,而将 ICE 的 CPU 引出线接到目标机的 CPU 插槽中。在用 ICE 进行调试时,在宿主机端运行的调试器通过 ICE 来控制目标机上运行的程序。

采用 ICE 方式,可以完成如下的特殊调试功能:

- 同时支持软件断点和硬件断点的设置;
- 设置各种复杂的断点和触发器;
- 实时跟踪目标程序的运行,并可实现选择性的跟踪;

- 支持“Time Stamp”；
- 允许用户设置“Timer”；
- 提供“Shadow RAM”，能在不中断被调试程序运行的情况下查看内存和变量，即非干扰调试查询。

ICE 的调试方式特别适用于调试实时应用系统、设备驱动程序以及对硬件进行功能和性能的测试。利用 ICE 可进行一些实时性能分析，精确地测定程序运行时间（精确到每条指令执行的时间）。ICE 的主要缺点就是太昂贵，一般价格都在几千美元，功能更强的要几万美元。这显然阻碍了团队的整体开发，因为不可能给每位开发人员都配备一套 ICE。所以，现在 ICE 一般都是应用在普通的调试工具解决不了问题的情况下或者在做严格的实时性能分析的时候。

（5）OCD 方式

OCD（On Chip Debugging）是 CPU 芯片提供的一种调试功能（片上调试），可以被认为是一种廉价的 ICE 功能。OCD 的价格只有 ICE 的 20 %，但却提供了 ICE 80 %的功能。

最初的 OCD 是一种仿 ROM Monitor 的结构，是将 ROM Monitor 的功能以微码（Microcode）的形式体现，其中比较典型的是 Motorola 的 CPU 32 系列的处理器。后来的 OCD 彻底摒弃了这种 ROM Monitor 的结构，而采用了两级模式的思路，即将 CPU 的模式分为正常模式和调试模式。

正常模式是指除调试模式外 CPU 的所有模式。在调试模式下 CPU 不再从内存读取指令，而是从调试端口读取指令，通过调试端口可以控制 CPU 进入和退出调试模式。这样在宿主机端的调试器就可以直接向目标机发送要执行的指令，通过这种形式调试器可以读/写目标机的内存和各种寄存器，控制目标程序的运行以及完成各种复杂的调试功能。OCD 的调试结构如图 3－15 所示。

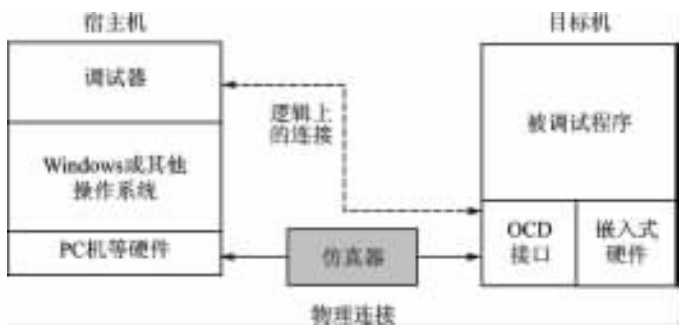


图 3－15　OCD 调试方式

OCD 方式的主要优点是：不占用目标机的资源，调试环境和最终的程序运行环境基本一致，支持软硬件断点，跟踪（trace）、精确计量程序的执行时间，具有时序分析等功能。

OCD 方式的主要缺点是：调试的实时性不如 ICE 强，不支持非干扰调试查询，CPU 必须具有 OCD 功能。

现在比较常用的 OCD 的实现有 BDM(Backgroud Debugging Mode)，JTAG(Joint Test Access Group)和 OnCE(On Chip Emulation，实质上是 BDM 和 JTAG 的一种融合方式)等。其中 JTAG 是主流的 OCD 方式，现在 ARM，MIPS 和 PowerPC 等都采用不同种类的增强 JTAG 方式。

1) 边界扫描技术

边界扫描技术(JTAG)全称是标准测试访问接口与边界扫描结构(standard test access port and boundary scan architecture)，已被 IEEE 1149.1 标准所采纳，是面向用户的测试接口。这个用户接口一般由 4 个引脚组成：测试数据输入(TDI)、测试数据输出(TDO)、测试时钟(TCK)和测试模式选择引脚(TMS)。有的接口还加了一个异步测试复位引脚(TRST)。其体系结构如图 3－16 所示。

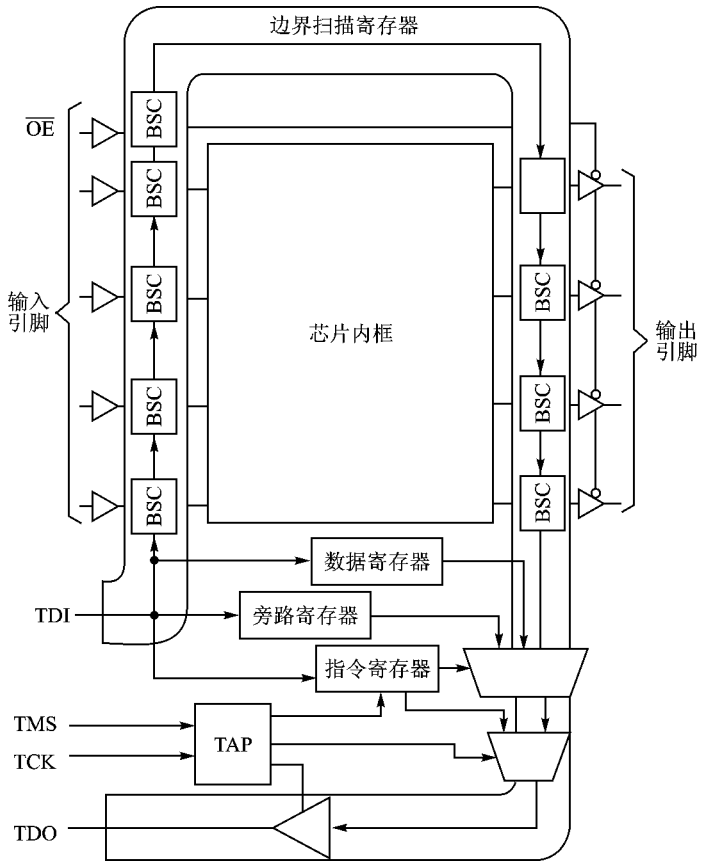
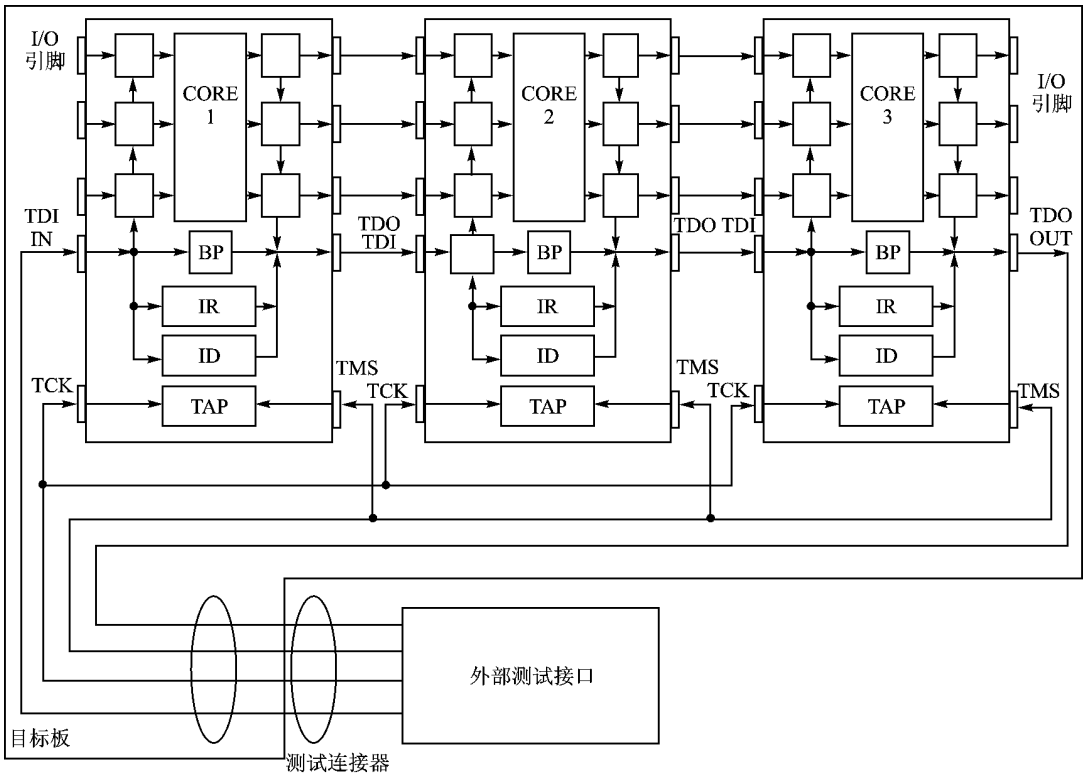


图 3－16　边界扫描体系结构

所谓边界扫描就是将芯片内部几乎所有的引脚通过边界扫描单元(BSC)串接起来,从JTAG的TDI引入,TDO引出。芯片内的边界扫描链由许多的BSC组成,通过这些扫描单元,可以实现许多在线仿真器的功能。根据IEEE 1149.1的规定,芯片内的片上调试逻辑通常包括一个测试访问接口控制器TAP。它包括一个16状态的有限状态机以及测试指令寄存器、数据寄存器、旁路寄存器和芯片标识寄存器等。在正常模式下,这些测试单元(BSC)是不可见的。一旦进入调试模式,调试指令和数据从TDI进入,沿着测试链通过测试单元送到芯片的各个引脚和测试寄存器中,通过不同的测试指令来完成不同的测试功能。包括用于测试外部电气连接和外围芯片功能的外部模式以及用于芯片内部功能测试(对芯片生产商)的内部模式,还可以访问并修改CPU寄存器和存储器,设置软件断点,单步执行,下载程序等。其优点如下:

- 可以通过边界扫描操作测试整个板的电气连接,特别为表面贴元件提供方便;
- 可以对各个引脚信号进行采样,并可强制引脚输出用以测试外围芯片;
- 可以软件下载、执行、调试和控制,为复杂的实时跟踪调试提供路径;
- 可以进行多内核和多处理器的板级和芯片级的调试,通过串接(如图3-17所示),为



BP—旁路寄存器; ID—(可选)标识寄存器; IR—指令寄存器; TAP—测试访问接口;
TCK—测试时钟; TDI—测试数据输入; TDO—测试数据输出; TMS—测试-模式选择

图3-17 对多内核和多处理器的调试

芯片制造商提供芯片生产、测试的途径。

虽然 JTAG 调试不占用系统资源，能够调试没有外部总线的芯片，代价也非常小，但是由于 JTAG 是通过串口依次传递数据，速度比较慢，只能进行软件断点级别的调试，自身还不能完成实时跟踪和多种事件触发等复杂调试功能。因此便有了几种功能更为完善的增强版本。

2）ARM 芯片的实时调试方案

ARM 公司的内核芯片采用 E－TRACE 实时调试方案片上调试模式。它实际上是 JTAG 的升级版本，通过增强的辅助片上调试硬件来完成实时调试，解决了许多传统调试器难以解决的问题。它的实时调试方案通过 3 种途径解决，即

① EmbeddedICE 硬逻辑；

② 实时监控；

③ 实时跟踪。

EmbeddedICE 逻辑单元存在于 ARM7TDMI，ARM9TDMI，ARM9E 和 ARM10 内核中。它在 JTAG 口的基础上，增加了硬件断点寄存器、比较器，通过断点寄存器的值可以进行硬件断点的设置，不仅对地址还可以对数据、控制总线的信号进行复杂的触发控制设定，而不是单单在指令级别进行中断（如软中断），从而满足对特定事件的中断响应，极大地增加了灵活性，同时可以在 ROM 中设置断点和观察点，极大地方便调试。其示意如图 3－18 所示。

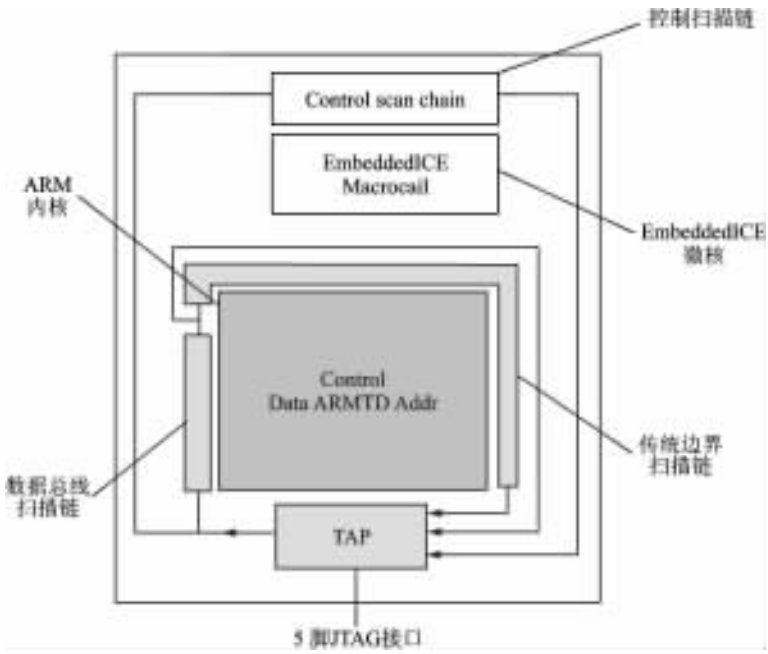


图 3－18　ARM 的 EmbeddedICE 模式

实时监控则是进一步在 ARM9E 和 ARM10 中的改进。它改变 EmbeddedICE 在触发中断后进入调试模式状态而停止内核运行的弊端，进入一段非常小的中断监控程序中，得到所需要的信息后迅速把控制权转让给先前的任务。在监控程序内处理器完全可以再接收外界的中断和其他触发事件，而不是停止运行。这种方式综合了 JTAG 和 E－Trace 的优点，可以增加以下 2 个好处：

① 在不禁止中断的前提下调试中断时正在运行的任务；

② 不用停止处理器的运行就可以读/写和修改存储器(对于机电设备非常重要)。

ARM 的实时跟踪解决方案由 3 部分组成，即

① 嵌入跟踪微核；

② 跟踪分析仪；

③ 跟踪调试软件。

通过这 3 种工具可以实现完全的实时跟踪。跟踪微核存在于芯片中，它可以不停止 CPU 的运行而实时监视芯片总线的信息，并把设定触发范围内的所有信息在 CPU 运行的同时通过压缩的方式送到外部的跟踪分析仪器里。分析跟踪仪器从芯片外部通过跟踪口(另外一个不同于 JTAG 的接口)收取信息。因为是压缩的数据，所以分析仪不需要采用与跟踪微核实时跟踪相同的速度。这大大降低了分析仪的成本，并增加了存储的容量。而 PC 端的跟踪软件则把来自分析仪的数据重新组织起来，从而重现处理器的历史状态和数据、程序流程；同时还可以把执行代码与源代码链接起来，使调试者快速理解跟踪数据。ARM 的这种方式通过芯片内部的实时跟踪硬件加上低成本的分析仪器，解决了传统在线仿真器和逻辑分析仪的诸多弊端。其示意图如图 3－19 所示。

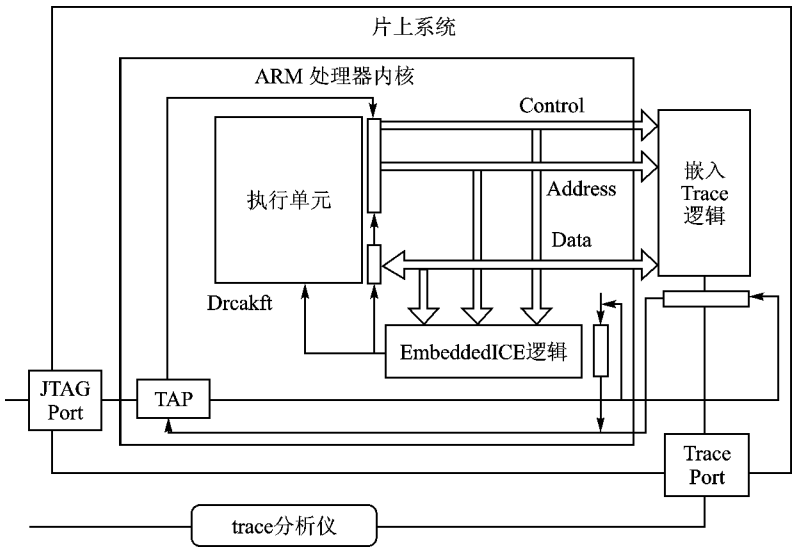


图 3－19　ARM 的 Trace 调试模式

3）Nexus 标准

自从 JTAG IEEE 1149.1 标准出来后，越来越多的高端嵌入式芯片生产商开始采用这个标准。但是 JTAG 标准只能提供一种静态的调试方式，如处理器的启动和停止、软件断点、单步执行和修改寄存器，而不能提供处理器实时运行时的信息。于是各个厂家在自己的芯片上，把原有 IEEE 1149.1 的基本功能进行了加强和扩展，如前面提到的 E－TRACE、背景调试模式 BDM 和片上仿真 OnCE 等，在处理器不停止运行的前提下，进行实时的调试。由于这些增强的 JTAG 版本之间各有差异，而且即使同一厂家的不同产品之间也存在着不同，所以一些芯片厂商和调试工具开发公司于 1998 年成立了 Nexus 5001 论坛，以期提出一个在 JTAG 之上的嵌入式处理器调试的统一标准。

Nexus 将调试开发分成四级，从第一级开始，每级的复杂度都在增加，并且上级功能覆盖下一级。第一级使用 JTAG 的简单静态调试；第二级支持编程跟踪和实时多任务的跟踪，并允许用户用 I/O 引脚作为多路复用辅助调试口；第三级包括处理器运行时的数据写入跟踪和存储器的读/写跟踪；第四级增加了存储替换并触发复杂的硬件断点。从第二级开始，Nexus 规定了可变的辅助口。辅助口使用 3～16 个数据引脚，用来帮助其他仿真器和分析仪之类的辅助调试工具。其示意图如图 3－20 所示。通过 Nexus 标准可以解决以下问题：

- 调试内部总线没有引出的处理器，如含有片内存储器的芯片；
- 传统在线仿真器无法实现的高速调试；
- 深度流水线和有片上 Cache 的芯片，能够探测具体哪条指令被取指和最终执行；
- 可以稳定地进行多内核处理器的调试。

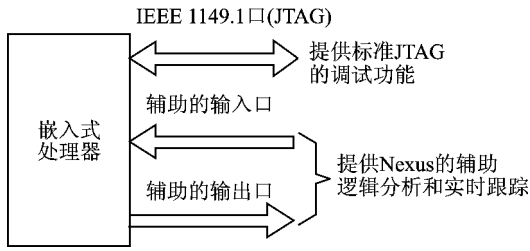


图 3－20　Nexus 的接口标准

3. 嵌入式软件的固化运行

当调试完成之后，程序代码需要被完全烧入到目标板的非易失性存储器中，并且在真实的硬件环境上运行，这个过程叫做固化。

在如前所述的 ROM Monitor 调试方式下，目标机上驻留的监控器导致了调试环境与真实目标环境（或称为固化环境）的区别。分析调试环境与固化环境之间的区别是解决固化问题的关键所在。两者的详细区别如表 3－2 所列。

表 3-2　调试环境与固化环境的区别

阶　段	调试环境	固化环境
编译	目标文件需要调试信息	目标文件不需要调试信息
链接	应用系统目标代码不需要 Boot 模块，此模块已由目标板上的监控器程序实现	应用系统目标代码必须以 Boot 模块作为入口模块
定位	程序的所有代码段、数据段都依次被定位到调试空间的 RAM 中	程序的各逻辑段按照其不同的属性分别定位到非易失性存储空间(ROM)或 RAM 中
下载	宿主机上的调试器读入被调试文件，并将其下载到目标机上的调试空间中，目标机掉电后所有信息全部丢失	在宿主机上利用固化工具将可固化的应用程序写入目标机的非易失性存储器中，目标机掉电后信息不丢失
运行	被调试程序在目标监控器的控制下运行，并与后者共享某些资源，如 CPU 资源、RAM 资源以及通信设备(如串口、网口等)资源	程序在真实的目标硬件环境上运行

可见，固化的代码和 RAM 中(调试)的代码有以下两个主要差别。

(1) 代码定位不同

在去掉调试监控模块和加入 Boot 模块的情况下，重新设置固化代码的定位控制文件，编译链接工具将按照各代码模块的属性把它们定位到相应的目标板存储空间中去。

在嵌入式系统中，一般使用两种存储器：一种是可读/写的 RAM；另一种是非易失性存储器如 ROM，Flash Memory 等。这两种存储器同时被映射到系统的寻址空间中，一般 RAM 被映射到地址空间的低端，而 ROM，Flash Memory 等则被映射到地址空间的高端。

在调试方式下，全部应用代码和数据都定位在 RAM 中，代码在 RAM 中运行；在固化方式下，代码和数据是存储在非易失性存储器中的，系统启动时要先将数据搬移到 RAM 中，而程序代码可在 ROM，Flash Memory 中运行。

(2) 初始化部分不同

固化程序要创建 Boot 模块，此模块被连接作为整个应用系统代码的入口模块。当应用程序在真实的目标环境下运行时将首先执行该程序，完成对 CPU 环境的初始化。在嵌入式环境中，Boot 模块一般包含以下几个功能：

- 初始化芯片的引脚，即按照系统的最终配置定义处理器芯片引脚功能；
- 初始化一些系统外部控制寄存器，如 WatchDog、DMA、时钟计数器和中断控制器等；
- 初始化基本的输入/输出设备，一般为串口、并口等；
- 初始化 MMU，包括片选控制寄存器等；
- 执行数据拷贝，将一些存储在非易失性存储空间的数据拷贝到真实的运行空间中去。

完成了上述准备工作，就可以利用编译链接工具生成可固化的应用程序，再用固化工具将它固化到目标机的 ROM，Flash Memory 等非易失性存储器上。当用户启动目标机时，该应用程序就会被自动装入运行。

3.3.4　嵌入式软件开发工具的发展趋势

嵌入式软件开发环境起初主要由专门开发工具的公司提供，这些公司根据不同操作系统和不同处理器版本进行专门定制，如美国 Microtec 公司的交叉开发工具套件曾经被 VRTX，pSOS 等定制采用。随着用户对开发工具套件的需求大增，一些著名的操作系统供应商投入巨资发展本系列操作系统产品的开发工具套件，如 Wind River 公司的 Tornado，ISI 公司（目前已被 Wind River 兼并）的 pRISM＋，Microtec 公司的 Spectra 以及 MS 公司的 VC＋＋嵌入式 Toolkit 等。它们有的使用方便，调试功能强大；有的采用先进的 CORBA 工具总线技术支持版本控制、团队开发；有的还具有一些辅助软件工程方面的工具。

在国际上，嵌入式软件开发环境的另一支研发队伍是 GNU。他们在因特网上提供免费的相关研究和开发成果，如针对特定处理器的 GCC（本地编译器）和 CGCC（交叉编译器）。尽管补丁和 Bug 较多，测试工作也还存在一些问题，但其不失为自主开发嵌入式软件开发环境的重要资源。一些公司已在 GNU 软件的基础上，经过集成、优化和测试，推出更加成熟、稳定的商业化的嵌入式软件开发环境，如 Cygnus 公司（目前已被 RedHat 兼并）推出的商业化产品 GNUPro，WindRiver 公司的 Tornado 和北京科银京成公司的 LambdaTool 等。

随着嵌入式系统的发展，嵌入式软件开发环境越来越重要，它直接影响嵌入式软件的开发效率和质量。下面介绍其发展趋势。

1. 向着开放的、集成化的方向发展

以客户/服务器的系统结构为基础，具有运行系统的无关性、连接的无关性、开放的软件接口（与嵌入式实时操作系统、开发工具和目标环境的接口）和环境的一致性等特点。

为了缩短开发时间和控制开发成本，嵌入式软件开发环境需要最大限度地承担重复性的工作，以便让开发人员有更多的精力去进行富于创造性、提高产品竞争力的工作。因此，需要集成各种类型和功能强大的工具，构成统一的集成开发环境。

2. 具有系统设计、可视化建模、仿真和验证功能

开发人员可通过功能强大的、可视化的软件开发工具对所开发的项目进行描述，建立整套系统的模型，并进行系统功能的模拟仿真和性能的分析验证，在设计阶段就能规避项目开发的很多风险，保证进度和质量。

3. 自动生成代码和文档

开发工具可根据系统模型生成 C/C＋＋/JAVA 语言的源代码，提供完善的、标准化的软件说明文档。这样就可有效节省 30 %～70 %的开发工作量，提高软件质量，提高软件团队的工程化能力和管理水平。

4. 具有更高的灵活性

嵌入式应用需求的个性化、多样化提升了嵌入式软件开发平台的灵活性，同时也对现有的技术和产品提出了更苛刻的要求。为此，嵌入式系统开发商需要拥有极其灵活的产品架构和开发工具，配备适应于特定行业的工具、操作系统和中间件。嵌入式软件开发平台是否具有很强的灵活性以适应产品的不断复杂化，将直接影响到客户的满意度和产品的市场竞争力。

思考题

3.1 嵌入式软件的种类和特点是什么？

3.2 嵌入式软件的体系结构包括哪些部分？每部分的作用是什么？

3.3 嵌入式软件的运行流程一般分几个阶段？每个阶段完成的主要工作是什么？

3.4 嵌入式操作系统与通用计算机操作系统的区别是什么？其发展趋势是什么？请分析一种面向行业的嵌入式操作系统标准。

3.5 嵌入式软件开发工具的分类如何？什么是交叉开发环境？

3.6 什么是交叉调试？交叉调试的种类有哪些？ROM Monitor 和 OCD 的主要优缺点是什么？

3.7 嵌入式软件固化运行和调试运行环境有什么不同？固化时需要注意哪些方面的问题？

3.8 嵌入式软件开发工具的发展趋势是什么？

第 4 章　嵌入式实时内核基础

本章介绍嵌入式实时内核的基础知识，并带来以下知识要点，以便为读者深入展开后面的学习打下基础。

- 嵌入式实时内核的关键设计问题；
- 嵌入式实时内核的主要功能；
- 嵌入式实时内核的重要性能指标。

4.1　嵌入式实时内核的关键设计问题

通过前面几章的学习，大家对嵌入式操作系统已经有了比较多的认识。相对于通用的、大型的计算机操作系统来说，嵌入式操作系统更为精巧，但这并不是说嵌入式操作系统是由简单地剪裁通用操作系统而来的。嵌入式应用领域对它提出了很多特别的要求，尤其对于嵌入式实时内核，在设计时通常需要考虑以下要求：

- 实时性；
- 可移植性；
- 可剪裁、可配置性；
- 可靠性；
- 应用编程接口。

下面分别对这些要求进行详细阐述。

4.1.1　实时性

实时性是实时内核最重要的特性之一。实时系统的正确性不仅依赖于系统计算的逻辑结果，还依赖于产生这些结果的时间。从整体上考虑，一个系统的实时性能与硬件、操作系统及应用程序三方面都有关系，提高硬件能力可以在一定程度上提高实时性；但是当硬件条件确定之后，嵌入式系统的性能主要是由操作系统来决定的，其中实时内核起着关键的作用。

为了更好地理解实时性，首先要了解一些相关的概念。

(1) 确定性

实时性通常是与确定性紧密相关的。所谓实时性是指实时内核应该保证系统尽可能快地对外部事件产生响应，而确定性是指系统对外部事件响应的最坏时间是可以预知的。对于实时内核，高的实时性和好的确定性是缺一不可的。一个系统是确定的，就意味着它在固定的、

预先确定的时间间隔内操作。

(2) 响应性

一个与确定性有关但又有区别的特性是响应性。确定性关心的是系统在识别一个外部事件(通常以中断的形式到达)之前有多长的延迟;而响应性关心的是在识别外部事件后,系统要花多长时间来服务该事件。比如对于中断来说,系统的响应性包括如下方面:

- 中断处理初始化以及开始执行中断服务例程(ISR)所需的时间。在一个多任务的实时应用中,ISR 可以在独立的上下文环境中执行,也可以在被中断任务的上下文环境中执行。如果在独立的上下文环境中执行,则需要完成上下文切换操作,那么总的延迟将会比 ISR 在当前任务的上下文中执行时更长。
- 执行 ISR 所需的时间。ISR 的功能是由实际应用定义的,因此这段时间取决于应用,另外还与硬件平台的性能有关。
- 中断嵌套的影响。如果一个 ISR 可以被其他中断打断,那么该中断对应的服务会被延迟,也会影响系统对特定事件的响应性。

(3) 响应时间

确定性和响应性结合在一起构成了系统对外部事件的响应时间。对于多任务实时应用来说,可以分为两种情况:中断响应时间和任务响应时间(4.3.10 节会有专门描述)。响应时间对于实时系统来说是关键的,对强实时内核来说,其响应时间应该在 μs 级。

在进一步分析影响实时性的主要因素之前,提出关于内核实时性的几个重要原则:

- 支持多任务。为了降低任务切换延迟,许多实时内核的实现都使用轻量级任务(即线程)。线程是轻量级的,因为它们携带的信息比进程要少。这意味着一个线程的控制块比进程的控制块要小,存储被抢占线程的控制块和恢复下一个执行线程的控制块所带来的开销就被降低了。因此在嵌入式实时系统中,多采用单进程多线程(任务)调度来提高实时性。
- 支持抢占式多任务。
- 支持任务的优先级调度。
- 任务的优先级可以继承。
- 支持可预测的任务同步机制。
- 实时内核的运行时间(如中断延迟、任务切换延迟等)可知并可以预测。
- 系统调用的确定性。

系统调用的确定性是指系统调用的执行时间即使在最不利的情形下也是可预测的。一个时间确定的系统调用,它的执行时间往往不是惟一的值,而是在一个范围内,其道理很容易想像。系统调用在各种情况下都只有惟一的执行时间是比较极端的情况。系统调用运行时间可以预测还有一层含义,即不论系统负载如何,系统调用的最大执行时间可以确定。

上述这些原则意味着在满负载的情况下,系统的最大响应时间可以获得。但是需要注意

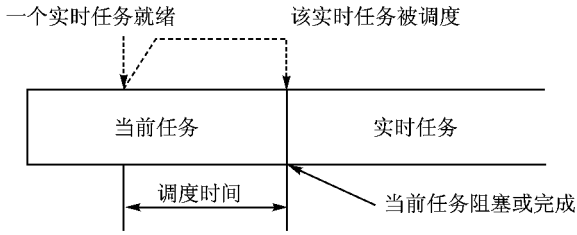
的是这些条件仅仅是必要而非充分的。

下面对影响实时性的主要因素进行详细分析。

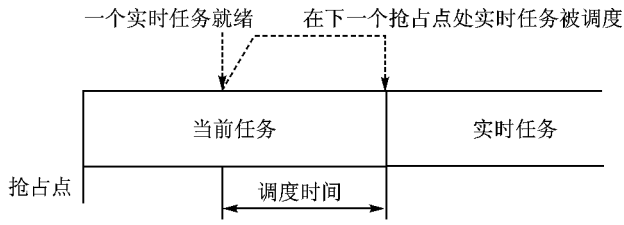
1. 调度算法

对于单处理器上的多任务实时内核，调度算法对响应时间有很大影响。在设计一个实时内核的调度器的时候，公平和最小化平均响应时间不是重要的。重要的是所有的硬实时任务要在它们的最后期限之前完成（或开始），以及使尽可能多的软实时任务也在它们的最后期限之前完成（或开始）。因此，实时内核被设计为尽可能地响应实时任务；当一个实时任务的最后期限接近时，它能够被迅速地调度。

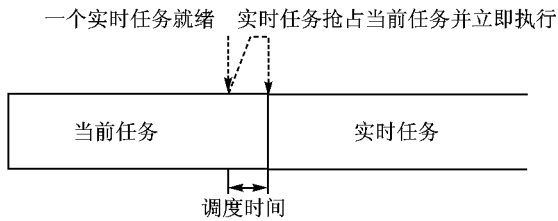
图 4－1 展示了几种不同的调度情况。



(a) 优先级驱动的非抢占式调度



(b) 在抢占点的优先级驱动的抢占式调度



(c) 立即抢占调度

图 4－1　一个实时任务的调度

- 情况(a):在一个非抢占式的调度器中,可以使用优先级调度,给予实时任务较高的优先级。在这种情况下,一旦当前任务阻塞或运行完成,一个已就绪的优先级更高的实时任务将会被调度。如果一个慢速的、低优先级的任务在这个关键的时间执行,就可能延迟优先级更高的实时任务,因而这个方法对实时内核来说是不可接受的。
- 情况(b):在规定的时间间隔处发生抢占点。当一个抢占点发生时,如果有一个更高优先级的任务在等待,当前运行的任务就被抢占。在这种情况下造成的延迟根据抢占点之间的时间间隔而定,可能在几个 ms 的数量级上。
- 情况(c):这最后一个方法可以满足要求更高的实时应用,被称为立即抢占。在这种情况下,内核几乎立即响应一个实时任务,除非系统处于一个临界代码锁定段中。一个实时任务的调度延迟可以被降低到更少。

在大多数实时内核中,为了能够在突发状态时迅速作出反应,大都采用"抢占式优先级任务调度"的机制,也就是实时内核有权主动终止当前任务的执行,将执行权交给新就绪的高优先级任务,并且是立即抢占的。目前比较流行的思路是采用基于优先级的可抢占调度作为主要的调度方式,配合同优先级时间片轮转调度作为可选择的调度方式,兼顾同优先级任务,使它们具有平等的运行权利。基于优先级的调度方式是指 CPU 总是让处于就绪态的、优先级最高的任务先运行。

实时任务就绪的原因可能是:

- 系统发生中断,中断处理过程中使实时任务就绪;
- 当前运行任务调用操作系统功能,使实时任务就绪。

假设实时任务就绪的原因是发生了中断,通过下面的实例,说明在非抢占式调度和抢占式调度的情况下,系统的具体运行过程。

(1) 非抢占式调度

非抢占式调度要求每个任务主动放弃 CPU 的使用权。假设在任务运行过程中可以响应中断,并且中断服务使一个高优先级任务由其他状态变为就绪态;但中断服务以后,CPU 使用权还是归还给原来被中断了的那个任务,直到该任务主动放弃 CPU 的使用权,新就绪的高优先级的任务才能获得 CPU 的使用权。

图 4-2 表示了非抢占式调度时系统的运行情况。

① 低优先级任务在运行过程中,一个外部中断到达;

② 如果允许中断,则 CPU 进入中断服务程序;

③ 中断处理过程中使一个高优先级任务就绪;

④ 中断服务完成且中断嵌套层数为 0 时,CPU 归还给原先被中断的低优先级任务;

⑤ 低优先级任务继续运行;

⑥ 低优先级任务运行完成或因其他原因被阻塞,释放 CPU,内核进行任务调度,切换到在中断处理过程中就绪的高优先级任务;

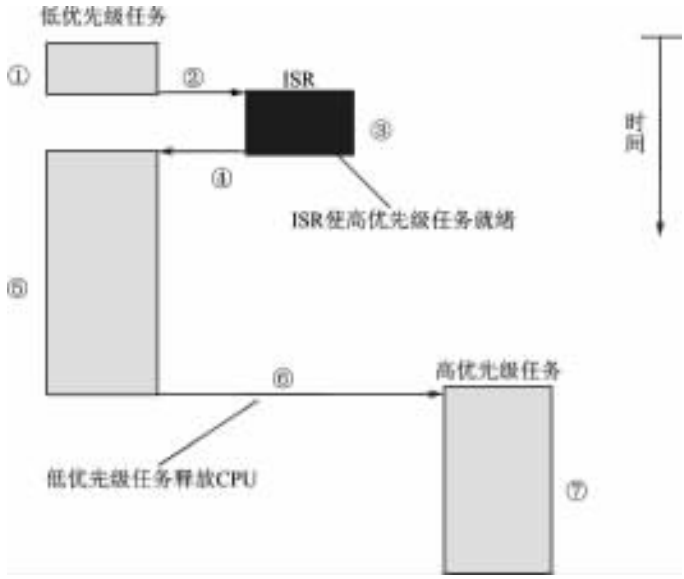


图 4－2　非抢占式调度

⑦ 高优先级任务运行。

因为无法确定已经就绪的最高优先级任务何时能获得 CPU 的使用权，故非抢占式调度的任务响应时间是不确定的。这完全取决于应用程序。

（2）抢占式调度

在抢占式调度的情况下，一旦更高优先级的任务就绪，当前任务的 CPU 使用权就会被尽快剥夺，以使更高优先级的任务能够尽快得到 CPU。如果是中断服务程序使一个高优先级任务就绪，那么在中断完成后，高优先级任务开始运行，如图 4－3 所示。

① 低优先级任务在运行过程中，一个外部中断到达；

② 如果允许中断，则 CPU 进入中断服务程序；

③ 中断处理使得一个更高优先级的任务就绪；

④ 当 ISR 完成且中断嵌套层数为 0 时，内核进行任务重新调度，让新就绪的高优先级任务获得 CPU；

⑤ 高优先级任务运行；

⑥ 高优先级任务运行完成，或者因为其他原因被阻塞，内核进行任务重新调度；

⑦ 低优先级任务获得 CPU，从被中断的代码处继续往下运行。

可见，抢占式调度使得任务响应时间得以优化。

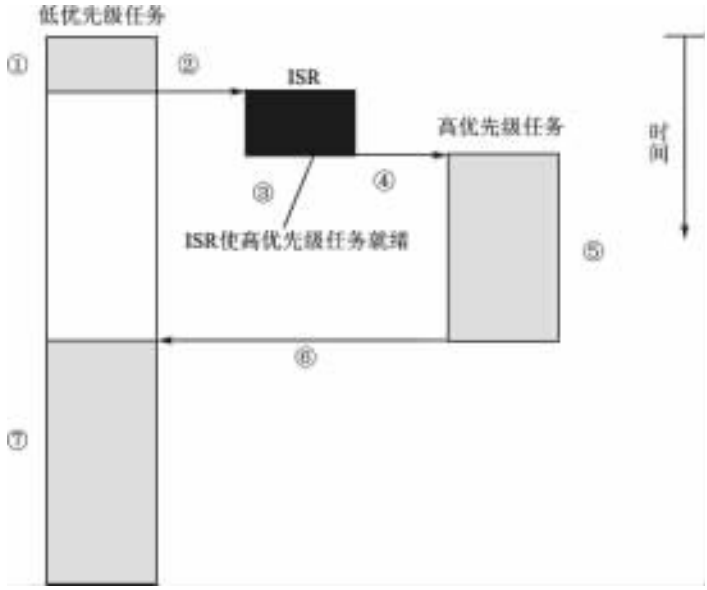


图 4-3　抢占式调度

2. 可抢占内核

可抢占内核(preemptable kernel)与可抢占调度(preemptive scheduling)是不同的概念。内核可抢占与不可抢占,体现在任务在使用内核提供的系统调用过程中被中断打断的不同处理上。

- 可抢占内核　即使正在执行的是内核服务函数,也能响应中断,并且中断服务程序退出时能进行任务重新调度。如果有优先级更高的任务就绪,就立即让高优先级任务运行,不要求回到被中断的任务,将未完成的系统调用执行完。
- 不可抢占内核　不可抢占内核有两种情况:一是内核服务函数不能被中断,二是能被中断但是不能进行任务重新调度。在第一种情况下,系统在执行内核服务函数时处于关中断状态,不能响应外部可屏蔽中断,这样就会在一定程度上延迟中断响应时间。在第二种情况下,系统在执行内核服务函数时可以响应中断,不会延迟中断响应时间,但是在中断退出时不进行任务重新调度,即使在中断服务程序执行过程中有更高优先级的任务就绪,也必须回到被中断的任务,将未完成的内核函数执行完后,才能让高优先级任务执行。

假定内核采用抢占式调度,通过图 4-4 和图 4-5,可以理解可抢占内核和不可抢占内核的区别(假定中断服务程序使更高优先级的任务就绪)。

图 4-4 中:

① 低优先级任务调用内核服务;

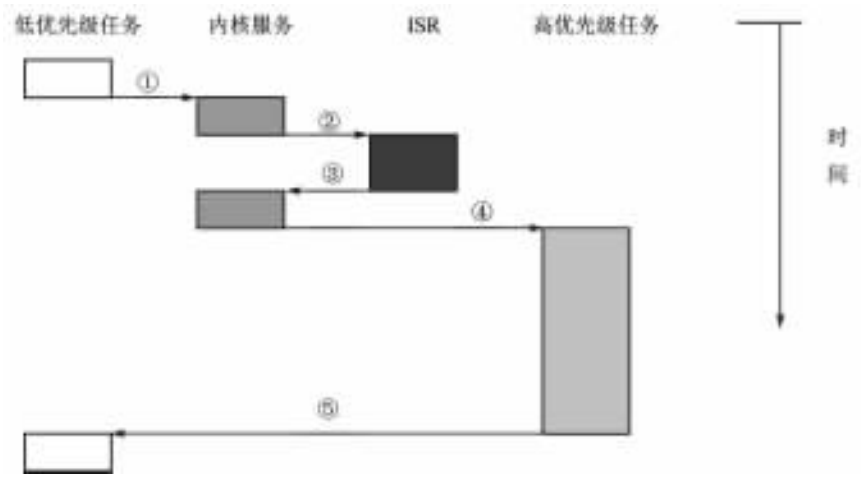


图 4－4　不可抢占内核

② 内核服务过程中，系统发生中断，在允许中断的情况下，进入中断服务程序(ISR)；

③ 中断服务程序完成后，回到内核服务中；

④ 内核服务完成，进行任务调度，切换到新就绪的高优先级任务；

⑤ 高优先级任务运行完成或者因为其他原因阻塞，内核调度低优先级任务，低优先级任务恢复执行。

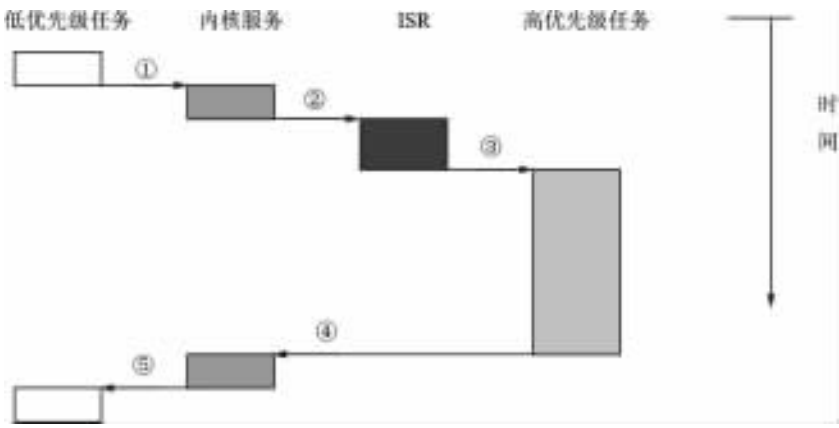


图 4－5　可抢占内核

图 4－5 中：

① 低优先级任务调用内核服务；

② 内核服务过程中系统发生中断，在允许中断的情况下，进入中断服务程序(ISR)；

③ 中断服务程序完成后，内核调度新就绪的高优先级任务运行；

④ 高优先级任务运行完成或者因为其他原因阻塞，系统回到先前被低优先级任务调用的、尚未完成的内核服务中；

⑤ 内核服务完成，返回到低优先级任务中，低优先级任务继续执行。

通过图 4－4 也可看出，采用抢占式调度方式的内核不一定是可抢占内核。把内核设计成为一个可抢占内核，将进一步提高它的响应性。

3. 内核的关中断时间

内核的关中断时间由内核服务函数对临界资源的操作而引入。为了保护临界资源不被破坏，在临界区中需要暂时屏蔽中断。内核服务函数对临界区的操作可能不连续，即临界区之间有非临界区的操作。对于内核中的这种服务函数，可以合理地设置一些可抢占区域或可抢占点（开放中断的地方），以减少系统的关中断时间。不同的内核在中断响应时间上的差异主要来自于内核的最大关中断时间，所以，通过这些处理可以让内核具有优秀的及时响应中断的能力。

4. 数据结构

为了保证内核各种功能的执行时间的确定性，可在算法和数据结构方面进行细致的考虑，比如采用优先级位图算法、双向链表数据结构和差分时间链等。

（1）优先级位图算法

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找出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。采用优先级位图算法可以做到调度时间与就绪任务数目无关。

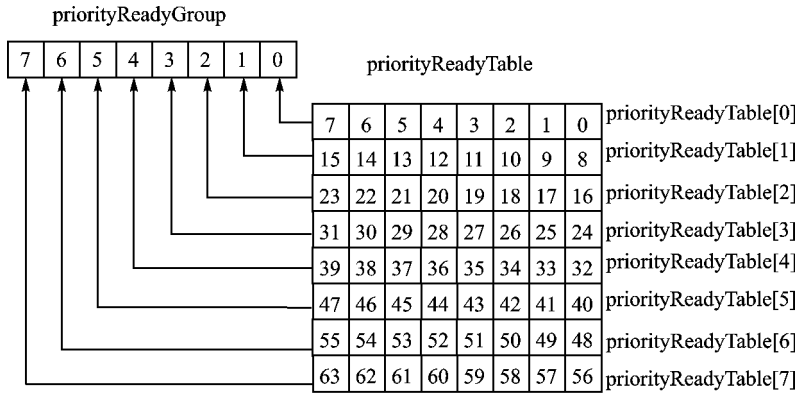
图 4－6(a)，(b)是优先级位图算法中关键数据结构（优先级就绪表和优先级判定表）的示意图。

关于优先级位图算法的原理在本书第 5 章中有具体描述。

（2）双向链表

内核的实现中会涉及到对链表的操作，例如在链表中某一位置插入一项或从链表中某一位置删除一项等。对于单向链表，从头到尾每个表项（组成链表的元素）只有后继指针，没有前驱指针，插入表项或者删除表项的执行时间取决于链表的长度和插入或者删除的位置。当表项的位置随着链表不断被操作（插入或者删除）而不断变化时，这类操作的时间无法预测。在内核实现中采用双向链表结构可以解决这个问题。由于双向链表记录了表项的前驱和后继，从而可以大大提高这类操作执行时间的确定性。

如图 4－7 所示，内核使用各种控制块来管理各种内核对象，比如任务控制块 TCB（Task Control Block）、消息队列控制块 QCB（message Queue Control Block）和信号量控制块 SCB（Semaphore Control Block）等。图 4－7 中的 *X*CB 即代表其中的一种。具体实现时，双向链表的形式是多样的。



(a) 优先级就绪表

```
char priorityDecisionTable[] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x00-0x0F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x10-0x1F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x20-0x2F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x30-0x3F */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x40-0x4F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x50-0x5F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x60-0x6F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x70-0x7F */
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x80-0x8F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0x90-0x9F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0xA0-0xAF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0xB0-0xBF */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0xC0-0xCF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0xD0-0xDF */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  /* 0xE0-0xEF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0   /* 0xF0-0xFF */
}
```

(b) 优先级判定表

图 4－6　优先级位图算法中关键数据结构

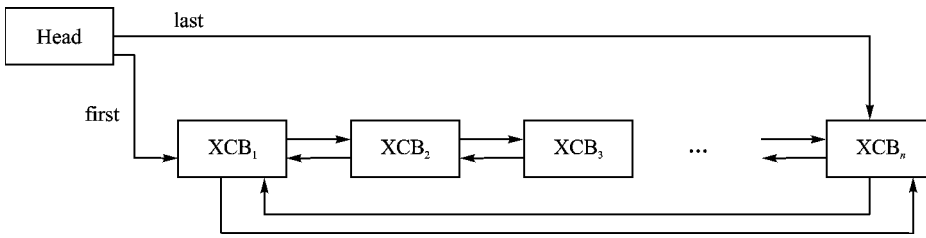


图 4－7　双向链表示例

(3) 差分时间链

在嵌入式实时内核的实现中,需要进行各种与时间有关的操作,比如将任务进行延时;按时间片进行同优先级任务的轮转调度;设置定时器以触发定时的处理等。在这些操作中会使用到时间链这种数据结构,而采用差分时间链可以提高相关运算的效率。如图 4-8 所示的定时队列,每个节点处的数字代表需要延时的"时基"数。"时基"是对系统时钟进行计数的基本单位。每次将一个定时节点插入到该队列中时都按照差分算法加入,这样每当一个时基到达时,只需要对队列头节点中的数字进行减 1 操作,而不需要对每个节点都这样做。

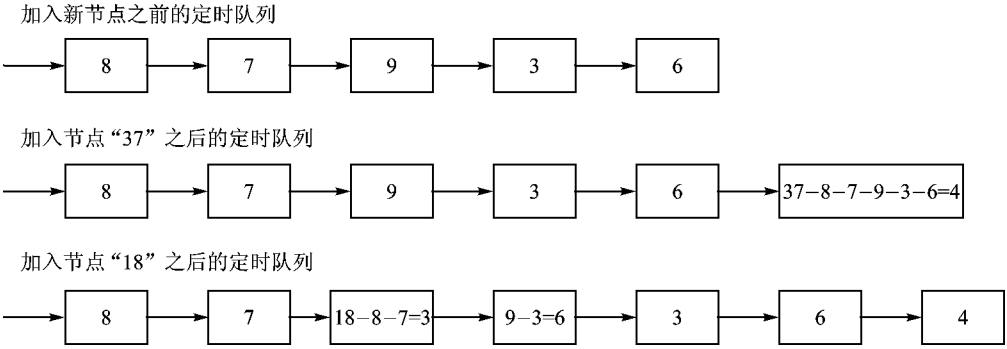


图 4-8 差分时间链

5. 存储管理机制

在嵌入式实时内核的实现中,通常不采用虚拟内存管理机制,以提高系统的实时性。

虚拟内存管理中的缺页调度时间取决于需要调入的页面在外围存储介质中的物理位置(比如某一柱面、磁道、扇区),造成执行时间上的无法预测性。一个严格的实时嵌入式系统要求实时性、内存锁定和代码紧凑,虚拟内存管理往往不能满足这些要求。尤其对于一个强实时的嵌入式系统,虚拟内存管理机制对时间确定性是不利的,而且还有移植上的困难(在不同的嵌入式目标硬件平台上实现虚存管理需要一定硬件机制的支持,比如微处理器的内存管理单元 MMU。不是所有的微处理器都具备 MMU 功能)。

从实践上,大多数的嵌入式系统一方面不具备大容量的外部存储器以支持虚拟存储;另一方面,嵌入式系统是确定的,不像通用计算机那样总是希望运行更多的应用程序,对于既定的应用来说,它在运行过程中所需的动态数据空间也是可预测的。因而在嵌入式系统中的内存容量一般都是经过特别计算并适合应用需求的。

因此,实时内核的存储管理通常采用无虚拟内存交换的管理方式。比如在 32 位的线性内存模式下,通过堆和分区两种内存分配技术分别提供可变大小和固定大小的分配方式,并且在堆分配中采用巧妙的算法有效避免内存碎片的产生。一般来说,嵌入式实时内核的存储管理没有碎片收集的功能。

6. 资源的有限时间等待

在多任务实时应用中，任务没能获得需要的资源就会被阻塞。如果该资源并不是任务继续运行必备的，则任务可选择有限等待该资源；在等待一段时间后如果还没获得该资源，则内核可以唤醒该任务，确保它余下的工作不被耽误。这就是资源的有限时间等待方式。采用该方式能有效地避免死锁，并提高系统的实时性。

7. 优先级反转问题的解决

抢占式的内核设计虽然可以降低任务重新调度的延迟时间，但会产生优先级反转失控的问题：低优先级的任务 L 占用临界资源但无执行权，高优先级任务 H 拥有执行权但又必须等待该临界资源，这样就导致低优先级的任务反而优先执行。这时如果系统中不断有中等优先级任务出现，系统的任务调度将变得不稳定并且难以预测。

凡是涉及到资源竞争的地方都会有优先级反转问题。考虑到互斥信号量这种特殊的资源通常被作为得到其他资源之前必须获得的钥匙，所以内核可以只针对互斥信号量的优先级反转问题提出解决方法，其他资源竞争中存在的优先级反转问题可以间接地通过互斥信号量加以解决。优先级继承和优先级天花板两种方法可以解决优先级反转问题。

优先级继承是当占有互斥信号量的任务的优先级低于请求获得该信号量的任务的优先级时，占有互斥信号量的任务的优先级被抬升到请求互斥信号量的任务的优先级。当任务释放完它所占有的全部互斥信号量时，其优先级才恢复到它在进行任何优先级继承操作前的优先级。

优先级天花板是将申请某资源的任务的优先级抬升到可能访问该资源的所有任务中最高优先级任务的优先级。这个优先级称为该资源的优先级天花板。互斥信号量的优先级天花板在它被创建时指定。

关于优先级继承和优先级天花板的详细说明请参看本书后续的章节。

8. 中断处理

内核对中断的处理，可通过以下设计使系统具有良好的实时性。

(1) 允许中断嵌套

在处理某一级中断的过程中，允许更高优先级的中断打断它。中断嵌套让更紧急的中断优先得到服务。

通过这项功能，系统设计者可以指示外部中断的优先级，从而确保高优先级的中断能及时处理。利用外部中断控制器来设置中断的优先级，在中断处理程序的启动过程中，设置中断控制器(比如 8259)的中断屏蔽寄存器的相应位，使得较低优先级的中断不能被响应；在离开中断处理程序时，恢复屏蔽位。

(2) 简短的中断服务程序

基于两方面的原因，中断服务程序应该尽量简短。

一方面，中断处理过程中屏蔽了同级和较低级别的中断，对这些中断的处理要等当前的中

断服务程序执行完后才能开始。如果发生中断嵌套致使当前中断服务程序被打断,则被打断的中断服务程序要等所有高优先级中断的服务程序执行完后才能继续执行。

另一方面,在多任务应用中,对中断的处理并不需要全都由中断服务程序完成,通常采用中断服务程序和任务配合的方式来处理导致中断的外部事件。这时,中断服务程序只需作必要的处理,如接收外部设备产生的数据或信号、清除中断位等,更进一步的操作放到与之相关的任务中完成。中断服务程序通过内核提供的同步和通信机制与任务协调工作。

9. 浮点数的优化处理

在进行浮点数处理的时候,往往要使用到数字协处理器(如果有的话)。内核执行过程中,数字协处理器的内容仅在浮点任务被调度,并且该任务不是最近使用协处理器的任务时才会被更新。如果系统只有一个浮点任务,数字协处理器的内容就可以不作保存和恢复。这样的优化处理避免了不必要的协处理器上下文切换,有效地提高了效率。

4.1.2 可移植性

随着硬件平台的迅猛发展和嵌入式实时系统应用范围的不断扩大,支持多平台已经是嵌入式实时操作系统发展的必然趋势。可移植性好的操作系统在支持多平台方面具有开发周期缩短、代码可重用度高和维护工作量小等显著优点,所以追求良好的可移植性是设计实时内核时需要重点考虑的目标之一。

嵌入式软件的移植工作分为异种处理器平台之间的移植和同种处理器平台之间的移植两种。一方面,硬件平台多样化为嵌入式实时系统提供了多种选择,用户希望选用的操作系统能运行在多种嵌入式微处理器上;另一方面,多数嵌入式硬件系统是由微处理器及其外围系统电路组成的,具有相同微处理器的硬件系统可能会有多种不同的外围电路(如串行通信控制器、定时器、显示控制器和模拟/数字转换器等)。

对实时内核而言,在不同平台处理器下的移植工作集中在任务切换、中断控制设备和时间设备的驱动上。对于同类处理器平台、不同型号及不同外围电路间的移植工作则更加方便,主要集中在对芯片级控制寄存器的操作上。

影响内核可移植性的因素主要有以下三方面。

1. 编程语言

可移植性与编程语言有很大关系。高级语言实现的代码比用汇编语言实现相应功能的代码具有更好的移植性。此外,汇编语言实现的代码具有更高的执行效率和更紧凑的代码空间。设计内核时,要立足系统整体性能和效率的考虑,不要片面强调某一方面而对另一方面弃之不顾。

在嵌入式软件中汇编语言的使用是必不可少的。对一些反复运行的代码,使用高效、简捷的汇编能大大减少程序的运行时间。汇编语言作为一种低级语言可以很方便地完成硬件的控制操作,这是汇编语言在与硬件联系紧密的嵌入式软件中的一个优点。然而汇编语言是高度

不可移植的，尽可能少地使用汇编语言，而改用可移植性好的高级语言（如 C 语言）进行开发，可以有效地提高软件的可移植性。现在，编译技术已经得到了很大的发展，高级语言编译后产生的可执行代码与汇编产生的代码在性能和代码量上都相差不大。现在的高级语言编译器都提供灵活、高效的选项，以适应开发人员特殊的编程和调试需求。

2. 体系结构

合理的体系结构是良好移植性的基础。可以将内核设计为三层结构：硬件无关层、硬件抽象层和硬件相关层。内核的移植工作集中在硬件相关层，与其余两层无关，从而确保其具有良好的可移植性。

3. 代码实现的技巧

代码实现时可以使用一些技巧来提高可移植性，例如把不可移植的代码和汇编代码通过宏定义和函数的形式，分类集中在某几个特定的文件之中。程序中对不可移植代码的使用转换成对函数和宏定义的使用，在以后的移植过程中，既有利于迅速地对要修改的代码进行定位，又可以方便地进行修改，从而大大提高移植的效率。

举　例

为方便移植，考虑到编译工具间的差异，可采用下列的数据类型重定义。

```
typedef     signed char              T_BYTE;
typedef     unsigned char            T_UBYTE;
typedef     signed short             T_HWORD;
typedef     unsigned short           T_UHWORD;
typedef     signed int               T_WORD;
typedef     unsigned int             T_UWORD;
typedef      char                    T_CHAR;
typedef     signed long long         T_DWORD;
typedef     unsigned long long       T_UDWORD;
typedef     double                   T_DOUBLE;
typedef     float                    T_FLOAT;
typedef     unsigned int             T_BOOL;
typedef     volatile signed char     T_VBYTE;
typedef     volatile unsigned char   T_VUBYTE;
typedef     volatile signed short    T_VHWORD;
typedef     volatile unsigned short  T_VUHWORD;
typedef     volatile signed int      T_VWORD;
typedef     volatile unsigned int    T_VUWORD;
typedef     volatile signed long long T_VDWORD;
```

```
typedef    volatile unsigned long long    T_VUDWORD;
typedef    volatile double                T_VDOUBLE;
typedef    volatile float                 T_VFLOAT;
typedef    volatile unsigned int          T_VBOOL。
```

4.1.3 可剪裁、可配置性

嵌入式操作系统的开发者需要完成系统功能的全集，但是其使用者往往因为资源的限制，只要求获得其功能的子集，剪裁掉不需要的部分。另外，为了满足多种嵌入式实时系统的要求，嵌入式操作系统还应该能让用户可以方便地配置它。因此，为了能满足不同复杂程度的应用需求，嵌入式操作系统应该具有良好的可剪裁、可配置性。

可配置与可剪裁是联系紧密，但又有区别的两种特性。可剪裁性表示系统在增加、卸装功能模块（包括操作系统组件、组件中的模块和接口库）时仅需要做少量的修改或者根本不用修改。可剪裁性要求系统中各功能模块之间尽量不存在耦合关系。可配置性针对系统中未被卸装的模块，根据应用在数量、机制、工作空间和堆栈等方面的不同需求，决定系统的规模、功能以及内存分配等。例如，用户可以配置系统的任务数目、调度算法（是否采用可抢占调度、是否采用时间片循环轮转调度）和任务堆栈的大小等。

一个最小的多任务嵌入式软件可以只需要如下的内容：

- 一段用做引导的程序；
- 一个具备任务管理和定时功能的最基本内核；
- 一个初始任务。

内核的剪裁性取决于模块之间的耦合程度。耦合程度越小的系统，可剪裁的力度越大。在设计内核时，按照功能对系统模块作比较细致的划分，抽象出一部分公共函数作为实现其他功能的基础，这部分内容不可被剪裁。其他功能模块之间比较独立，具有良好的可剪裁性。但是，考虑到内核本身的意义所在，一些功能单从技术上看可以被剪裁，但还是必须保留在内核中，这些功能包括任务管理和定时功能。因此，剪裁也是有限的，有规则的。

用户可以通过配置已经选择了的模块，进一步调整系统的功能和规模。比如用户根据操作系统提供的一组配置参数实现组件或组件中模块的剪裁，以及对系统对象数（最大任务数、最大定时器数、最大信号量数、最大消息队列数、最大内存块数和使用的设备数等）、各种性质的内存空间（任务堆栈、系统堆栈和中断堆栈等）、调度算法和要使用哪些 API 等的配置。用户可以利用系统配置表为不同的应用配置系统，系统初始化时将根据配置表来初始化系统，并进行相关的操作，如加入新设备的驱动程序、加入扩展处理等。当然，为了减轻用户配置系统的负担，这些配置项目都应该提供缺省值，用户只需在配置文件中加入要重新设置或添加的项目即可。

另外，由于嵌入式实时系统应用的多样性，要求实时内核为应用提供多种服务，甚至为一

些现在没有，但将来可能需要的应用服务提供扩展支持，即要求内核具有良好的功能可伸缩性。关于内核功能扩展的问题将在 4.2.9 节中介绍。

4.1.4　可靠性

可靠性对于实时系统来说通常比非实时系统更为重要。在一个非实时系统中的一个瞬间错误可以通过简单地重新启动系统来解决；在多处理器的非实时系统中一个处理器的失败可能导致服务的降级，直到失败的处理器被修复或替换。但是一个实时系统是以实时的方式响应和控制事件的，性能的丢失或降级可能会带来灾难性的后果，从金钱损失到主设备的破坏，甚至会丧失人的生命。

为保证应用系统运行的可靠性，嵌入式实时内核可以提供诸多机制供用户使用，包括异步信号、定时器、优先级继承、优先级天花板、异常（或出错）处理、用户扩展和内存保护等。

嵌入式实时系统一般不具备通用的人机接口，运行时人不能干预其操作，因此系统的异常（或出错）处理能力是其可靠性很关键的因素之一。嵌入式软件通常被写入只读存储介质中，系统一上电就开始运行它，不允许软件临时从盘上装入，因此要求内核能提供一定的异常（或出错）处理机制。

异常处理为用户提供了处理应用程序造成的非正常情况的机制；而对于内核运行时出现的错误，异常处理可以记录错误的来源，判断错误的性质，并消除错误，使系统继续正常运行，或终止运行以避免带来更大损失。然而，由于应用环境千差万别，内核不可能为每一种错误都提供专用的异常处理，而只能够提供上述标准的异常处理。为了使用户能够针对自己应用的实际情况设计适应系统需要的异常处理，内核可以提供标准异常处理的扩展接口。

4.1.5　应用编程接口

每一个操作系统提供的系统调用（应用编程接口 API）的功能和种类都不同。当然，对于一个操作系统来说，它提供的系统调用越多，功能越强，也就越能对应用程序的开发提供高效的支持，同时也会减少应用程序的维护工作量；相反，一个操作系统的系统调用越少，越单一，则应用程序就因为要做更多的工作而变得更复杂，引入错误的可能性就越高。为了适应不断复杂的应用程序开发需求，操作系统设计的系统调用就越来越多，越来越复杂，功能也越来越强大。嵌入式操作系统的应用领域非常广，简单的可以应用在调制解调器这类设备上，复杂的可以应用在卫星地面通信接收站中。这就决定了嵌入式操作系统所提供的系统调用的数量和功能是因应用不同而不同的，它具有可剪裁性和可配置性。

另外，API 也影响着应用参与系统控制的深浅程度。在通用计算机系统中，用户一般不能控制操作系统的调度功能；然而在嵌入式实时系统中，允许用户精确控制任务的优先级却是很基本的。用户应该能够区分硬实时任务和软实时任务，并指定在不同的“优先级带”中的任务拥有什么样的权限，以及在每个级别内任务的相对优先级等。通过实时内核提供的 API，开发

者可以在应用中使用实时内核提供的各种功能，达到对系统硬件资源和软件资源的合理、充分的使用。

基于下面的原因，提供面向行业的接口标准是嵌入式实时操作系统的一个发展趋势。

- 尽管可剪裁性是嵌入式操作系统的一个非常重要的特性，但是任何一个嵌入式操作系统都不可能从具有各种完善功能、代码达几百 KB 的操作系统，剪裁到只具有实时调度和信号量操作的几 KB 的代码。所以嵌入式操作系统只能面向实际的被嵌入系统的具体要求，确定并提供最有效的系统调用。根据行业的要求确定 API 的定义，既方便行业应用的可移植性，又能减少操作系统的代码量。
- 目前嵌入式实时操作系统的 API 没有形成统一的标准，各个厂商都定义了各自的一套 API。虽然有些实时操作系统提供了与 POSIX(a Portable Operating System Interface based on UNIX)接口标准兼容的 API 调用，但也仅仅是实现了 POSIX 的一个子集，并且还同时提供一套非 POSIX 的系统调用集。POSIX 本身是以类 UNIX 为基础开发的，它实际上代表了 UNIX 类型的操作系统用户接口的国际标准集，以使应用程序可以从一个系统移植到另一个系统。针对实时领域的需要，专门定义了 POSIX 1003.1c 和 POSIX 1003.1d 接口标准。由于 POSIX 所涉及的范围十分广泛，内容繁多，嵌入式实时操作系统为保证精简，不可能提供对 POSIX 所有定义的支持。直接套用 POSIX 标准会导致系统过于庞大，违背嵌入式实时操作系统小巧的特征。
- 通常，嵌入式系统的开发人员不一定是从事计算机行业的专业人员，嵌入式系统硬件和软件仅仅是他们实现自己特定行业应用的工具。而嵌入式领域又是计算机专业与其他各种专业相结合的“边缘”领域，这就涉及到作为嵌入式系统硬件和软件开发商应该如何理解各种行业，如何提供适合行业需求的软、硬件产品的问题。为了推广市场，使嵌入式软、硬件产品迅速应用到这些行业中，开发商可以提供非常友好的界面，使最终产品开发者易于理解和使用这些产品。对于嵌入式操作系统厂商而言，可以按照不同的行业标准或面向不同行业的需求开发特定的 API，将下层通用 API 封装起来并进行必要的扩展。对于行业而言，则需要制定一些标准，以便操作系统开发商能够遵循这些标准，提供面向行业的版本，既为行业提供了多种选择的机会，也有利于保护用户在上层软件上的投资。

为了应对日趋激烈的国际市场竞争态势，设计技术共享和软件重用、构件兼容、维护方便和合作生产是增强行业性产品竞争能力的有效手段。近几年，一些地区和国家的若干行业协会纷纷制定嵌入式产品标准，特别是软件应用编程接口规范，比如航空电子的 ARINC 653、汽车电子的 OSEK 等。我国数字电视产业联盟也在制定本行业的开放式软件标准，提高中国数字电视产品的竞争能力。看来，走行业开放系统道路是加快嵌入式软件技术发展的途径之一。

举　例

航空电子应用软件标准接口规范 ARINC 653 **于** 1997 **年** 1 **月** 1 **日发布。它阐述了模块化综合航空电子设备** IMA(Integrated Modular Avionics)**和传统** ARINC 700 **系列航空电子设备中使用的应用软件的基线操作环境** BOE(Baseline Operating Environment)**。它定义了一个通用的** APEX(APplication/EXecutive)**接口,该接口位于航空计算机操作系统和应用软件之间;它还定义了系统为应用软件提供的一个功能集合,利用这个功能集合应用软件可以控制系统的调度、通信和内部状态信息。**

APEX **接口定义分为若干个阶段,其首要目的是提供一个位于应用软件和** IMA **内的** OS **之间的通用接口,确定提供给应用软件的最小功能集。最终的** APEX **接口定义可以为各种范围的应用提供基本功能,应用的范围从飞行关键应用,到为专门应用(比如数据库管理和大量数据存储与恢复)而开发的全功能** OS **接口。**

图 4－9 **显示了** OS,APEX **接口和应用软件的关系。**

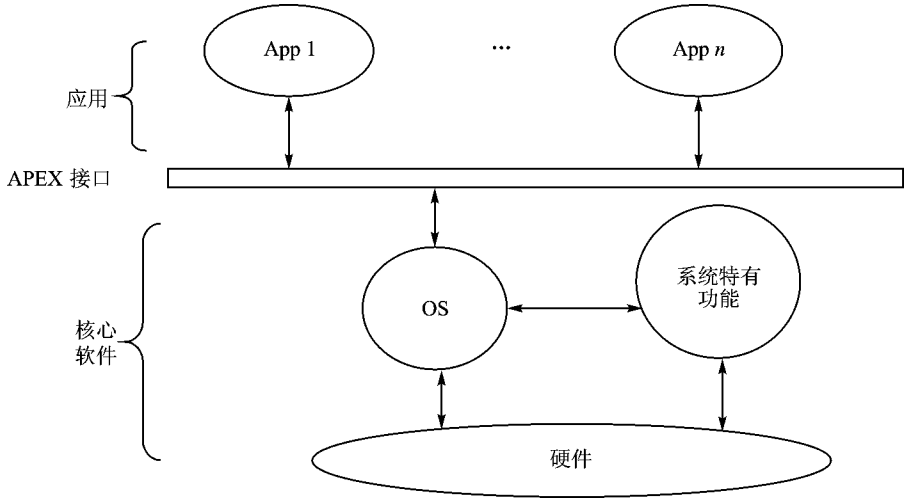


图 4－9　核心模块软件结构

为了保证 APEX **接口可以为业界带来最大益处,确立了如下准则:**

- APEX **接口提供满足** IMA **需求的最少数量的服务,简化应用软件开发者的使用,并且帮助他们生产出可靠的产品。**
- APEX **接口是可扩展的,以适应将来系统的增强特性。**OS **实际上会受到将来的扩充的影响,所以** APEX **接口应该能够容易扩展,同时保持与以前版本软件的兼容。**
- APEX **接口应满足** Ada 83,Ada 95 **和** CIFO **的公共的实时要求。根据其认证和确认关键级别的不同,这些不同的模型应使用** APEX **接口提供的下层服务。**
- APEX **接口消除了应用软件和实际的处理器结构之间的关系。因此,硬件的改变对应**

用软件是透明的。APEX 接口允许应用软件访问执行体的服务，但是隔离应用软件对体系结构的依赖。

- APEX 接口规范是与语言无关的，它必须支持使用不同高级语言，或者支持使用同一语言但是用不同编译器编写的应用软件。满足了这点要求，在选择编译器、开发工具和开发平台时会得到很好的灵活性。

APEX 的预期特性如下：

- 可移植性　APEX 接口使软件移植很简便，它通过去除语言对硬件的依赖性达到这个目标。
- 可重用性　使用 APEX 接口可以为 IMA 系统生产可重用的应用代码。当一个组件被重用时，APEX 接口可以减少定制工作的工作量。
- 模块独立性　开发应用软件时，APEX 接口可以提供模块化的好处。通过去除软、硬件的依赖性，可以减少对应用软件的影响。
- 关键级的集成性　APEX 接口支持将不同关键级别的应用软件很好地集成在一起。

4.2　嵌入式实时内核的主要功能

从某种意义上说，操作系统是计算机的一个扩展，它赋予了计算机更多的功能；从另一个角度看，操作系统也是计算机系统的资源管理者。

一般来讲，嵌入式实时操作系统内核主要有如下功能：

- 任务管理；
- 中断管理；
- 时间管理；
- 对共享资源的互斥管理；
- 任务间的同步与通信管理；
- 内存管理；
- I/O 管理；
- 出错处理；
- 用户扩展管理；
- 电源管理。

下面将对嵌入式实时内核的这些功能进行简要介绍，以便为后续章节的展开描述提供铺垫。

4.2.1　任务管理

在多任务系统中，任务是被调度执行和竞争资源的基本实体单元，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实现。那么由谁来对这些任务实施有效的管理，使得它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。

任务管理主要包括创建任务、删除任务、改变任务状态（或改变任务运行轨迹，如启动与重新启动任务、挂起与恢复任务、改变优先级和使任务睡眠等）和查询任务状态（如优先级、属性）等功能，其核心是任务调度。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。

实时内核一般采用基于优先级的可抢占式调度，同优先级的任务还可以采用时间片轮转调度（轮转调度是使任务按照一个固定的时间间隔轮流运行；时间片调度是根据每个任务的实际情况在不同数量的时间片内运行。有的内核要求实行轮转的各任务的时间片是一样的，而有的内核可以给不同的任务提供不一样的时间片）。有的内核还提供单调速率调度 RMS（Rate Monotonic Scheduling）。基于优先级调度的实时内核可以提供改变任务优先级的调用，应用程序可以在运行中改变任务的优先级；另外在某些提供防止优先级反转策略的内核中，根据所选用的策略对任务优先级实施动态改变，但是除此之外内核一般不支持其他的动态优先级调度算法，即自动地检测当前系统中各任务的状态，并根据一定的调度策略自动改变任务的优先级。

不同的内核最多支持的应用任务数目、任务的状态数不一样，最多支持的优先级数也不一样。

4.2.2　中断管理

中断本身是一种异步机制，中断服务程序（ISR）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。内核对中断提供的管理功能使得这种协同机制能够实现。一般来讲，内核的中断管理有如下功能：

- 安装指定中断的服务程序，使得一个硬件中断和一段例程相关联。当中断发生时，系统作中断现场的保存和恢复，并且转到相应的 ISR 中执行。ISR 负责处理中断，清除中断标记以使中断能够再次发生（与具体的中断控制器相关），以及进行设备操作（如果有）。ISR 运行时可以使用当前被中断任务的堆栈，也可以使用专门的中断堆栈。实时内核一般提供专门的堆栈来处理中断，防止可能的任务堆栈溢出。
- 为某些设备提供缺省的中断处理程序。
- 将内核内部支持的各级中断映射到目标处理器的各级中断上。
- 在 ISR 中可以使用内核提供的功能调用，以便完成与任务的通信。在 ISR 中使用系统调用是有一定条件限制的，内核必须提供某种机制来避免在 ISR 中由于这些调用可能造成的意想不到的后果。

● 提供屏蔽与使能中断的系统调用，使应用能够根据需要在运行时关闭和打开中断。

对中断的处理包括中断前导、用户中断程序和中断退出三个部分。各部分的功能如下：

① 中断前导　完成现场保护，调用系统的中断处理程序，最后调用用户的中断程序；

② 用户中断程序　完成用户自己的中断处理；

③ 中断退出　恢复现场并进行任务调度方面的相关处理。

第①与第③部分可以由内核接管，对应用透明，用户的 ISR 不用关注现场保护等工作，简化了 ISR 的编写。

内核根据需要支持中断嵌套。发生嵌套时，任务调度会延迟到最外层中断处理结束后才发生。

4.2.3　时间管理

内核的时间管理功能为应用系统的实时响应提供支持，保证其实时性、正确性，以提高整个嵌入式系统的实时工作能力。与时间相关的管理包括时钟管理和定时器管理。

时钟管理部分的功能可以包括：

● 维护系统时基，执行与系统时基相关的处理或操作，比如任务延时、资源限时等待等；

● 设置和取得时间信息，包括以系统时基为单位的时间和日历时间。

定时器机制允许任何函数与一个特定的时间延迟相联系。设置并启动定时器后，定时器开始计时，一旦指定的延时结束，该函数立即被调用。具体来讲，定时器管理功能可以包括：

● 创建、删除定时器；

● 设置定时器的触发时间；

● 定时器重新开始计时；

● 终止定时器计时。

利用定时器功能可以为系统提供多个“软定时闹钟”，以满足对多种不同的定时事件进行处理的需求。定时的事件可以是单次触发的，也可以是周期性触发的。

4.2.4　对共享资源的互斥管理

与共享资源打交道时，实现资源互斥访问的方法很多，不同之处在于互斥的范围和程度。这些方法包括关中断、使用测试并置位指令、禁止任务切换和使用信号量。

1. 关中断

处理共享数据时保证互斥，最简便快捷的办法是关中断和开中断。内核在处理内部变量和数据结构时就是使用的这种手段，即使不是全部，也是大部分。这也是在中断服务子程序中处理共享变量或共享数据结构的惟一方法。内核可以提供两个调用（宏），允许用户在应用程序中关中断和开中断。

从互斥的粒度来讲，禁止中断是最强有力的互斥机制，这种上锁保证了对 CPU 的独占访

问。这种方法涉及到中断级互斥，也就是说，在互斥期间，即使外部事件引发相应的中断，系统也不会切换到相应的中断服务程序(ISR)。所以在上锁期间，它可能会造成系统对外部事件反应迟钝。对于多数实时系统而言，这使得系统的实时性得不到保证，影响着系统的中断响应时间，因而不适合作为一种通用的互斥方法。在任何情况下，关中断都要尽量短。如果使用不恰当，将会明显增加中断延迟。一般来说，当改变或复制某几个变量的值时，可以采用这种方法来实现互斥。

2. 使用测试并置位指令

如果不使用实时内核提供的机制，当两个任务共享一个资源时，可以采用如下方法：先测试某一全局变量，如果该变量为 0，则允许该任务访问共享资源。为防止另一个任务也要使用该资源，只要简单地将全局变量设置为 1 即可，这通常称为测试并置位。但这有个前提：测试和置位是微处理器的一条不会被中断的指令，或者在做该操作时关中断，以后再开中断。有的微处理器有硬件的测试和置位指令。

3. 禁止任务切换

另一种比关中断稍弱的互斥机制是禁止任务抢占，即不允许其他任务抢占当前任务的执行。禁止抢占提供了一种较小限制性的互斥，在这种情形下，ISR 仍然能够执行。因为此时虽然任务切换被禁止了，但中断还可以是开着的。如果这时中断来了，中断服务程序还是会在这一临界区内执行。中断服务程序结束时，即使有优先级高的任务进入就绪态，内核还是返回到原来被中断了的任务。临界区执行完后开调度时，内核才看是否有优先级更高的任务被中断服务程序激活而进入就绪态；如果有，则作任务切换。虽然禁止任务切换(即对抢占上锁，或关调度/开调度)也是一种有效的互斥方法，但也应该尽量避免这一类的操作，因为内核最主要的功能就是作任务的调度与协调。

另外，这种方法仍然可能造成系统的实时性得不到充分保证。这是因为在上锁的任务离开临界区解锁之前，处于就绪态的更高优先级的任务也不能够执行，尽管这个高优先级的任务可能根本没有涉及到临界区操作。当然，这种互斥方法非常简单。使用这种方法时，同样需要控制禁止抢占的时间尽可能短。一种更好的机制是信号量。

4. 使用信号量

信号量是提供任务间通信、同步和互斥的最优选择，也是同步和互斥的主要手段。内核可以提供专门优化了的互斥信号量，以解决信号量机制内在的互斥、优先级反转、删除安全和递归等情况。

信号量提供比禁止中断或禁止抢占更为精确的互斥粒度，与这两种方法相比，信号量将互斥仅仅限于与之联系的资源的访问。通过对共享资源上锁，实现高效的互斥访问。

信号量常被用过了头，处理简单的共享变量也使用信号量是多余的。请求和释放信号量的过程是要花一定的时间的，有时这种额外的负荷是不必要的。用户可能只需要通过关中断、开中断来处理简单的共享变量，以提高效率。例如两个任务共享一个 32 位的整数变量，一个

任务给这个变量加1,另一个任务给这个变量清0。如果这两种操作对微处理器来说都只花极短的时间,就不用使用信号量来满足互斥条件了。每个任务只需在这个操作前关中断,之后再开中断就可以了。然而,如果这个变量是浮点数,而相应的微处理器又没有硬件的浮点协处理器,浮点运算的时间相当长,关中断时间长了会影响延迟时间,那么这种情况下就有必要使用信号量了。

各种互斥机制比较如表4-1所列。

表4-1 各种互斥机制比较

比较项目	关中断	使用测试并置位指令	禁止任务切换	使用信号量
锁定范围	互斥粒度最强,锁定所有外部可屏蔽中断,凡是以中断形式到达的外部事件以及与之相关联的任务或处理过程均得不到执行	所有使用该指令访问共享资源的代码	所有的任务	只影响竞争共享资源的任务
对系统响应时间的影响	如果关中断的时间较长,则对系统的响应性能有很大影响	较小	如果禁止切换的时间过长,则影响系统的响应性能	对系统响应性能有一定影响,可能导致优先级反转
实现时的系统开销	小	小	小	较大
注意事项	关中断时间要尽量短	不是所有的处理器都具备这种指令,影响可移植性	关调度的时间要尽量短	需采用一定的策略解决优先级反转问题

在上述几种互斥机制中,关中断、禁止任务切换以及信号量都是内核可以提供给用户的功能。如何合理地使用这些机制保护临界区(访问共享资源的代码段),即采用何种手段保护临界区是根据共享资源的访问源决定的。下面给出一些原则。

(1) 访问源都是任务

在这种情况下通常采用信号量保证互斥。创建初值为1的互斥信号量,任务访问共享资源之前都先申请信号量,退出时再释放。最先试图进入临界区的任务会成功获得信号量进入临界区,其他任务在此期间申请信号量都会被阻塞,待访问共享资源的任务完成访问退出临界区时再释放信号量,内核会完成相应的任务调度,选择合适的任务进入临界区继续执行。

在有些实时内核中,互斥信号量可以嵌套获得,即该互斥信号量即使处于锁定状态,它的拥有者如果再次申请该信号量仍然可以成功获得,当然其他任务申请处于锁定状态的信号量仍然会阻塞。

除了信号量,还可以使用关调度的方法,强行禁止在访问临界区期间发生任务切换。当然

要注意关调度可能会影响那些不需要访问共享资源的任务。

如果用户对系统有较好的了解和把握，采用变通的方法，使用其他的机制（比如事件或者消息队列）也可以在一定程度上满足需要，当然前提是一定要清楚采用这些方法会怎样影响系统的行为。

（2）访问源都是中断服务程序

因为 ISR 是不同于任务的独立体，不能使用信号量等方法，ISR 是不会阻塞在信号量上的，所以如果访问共享资源的都是中断服务程序，则只能使用关中断的方法。

（3）访问源可能是任务也可能是 ISR

如果无法区分一次具体的访问到底是来自任务还是来自 ISR，只能使用关中断的方法。不过可以尝试通过重新规划应用程序来改变这种情况。

4.2.5　同步与通信管理

任务与任务之间、中断服务程序与任务之间的同步和通信机制可以有：

- 信号量；
- 消息队列；
- 事件；
- 异步信号；
- 管道。

1. 信号量

在多任务内核中普遍将信号量用于任务之间、任务与中断服务程序之间的同步以及任务间的互斥。

- 控制共享资源的使用权（满足互斥条件）；
- 标志某事件的发生；
- 同步任务之间、任务与中断之间的行为。

信号量是一把钥匙，与之相关的代码要运行下去，得先拿到这把钥匙。包括前面提到的用于互斥的信号量在内，内核一般可以提供 3 种类型的信号量用于解决不同的问题。

① 用于解决互斥问题的互斥信号量；

② 用于解决同步问题的二值信号量；

③ 用于解决资源计数问题的计数信号量。

互斥信号量是比较特殊的，在其使用过程中可能发生优先级反转问题。内核可以提供优先级继承和优先级天花板两种可选技术，以解决这个问题。

对信号量的基本操作有：

- 创建与删除信号量；
- 申请与释放信号量。

2. 消息队列

消息队列机制提供任务之间、任务与中断服务程序之间的通信功能。

一个消息就是一个可变长度的缓冲,在这个缓冲中存放信息以完成通信。消息的长度及其中存放的内容是用户定义的,可以是数据,也可以是指针,或者为空(在这种情况下消息的发送和接收是用来进行同步的,不进行数据传输)。消息队列允许在任务和中断间传递消息。

对消息及消息队列的基本操作可以有:

- 创建与删除消息队列;
- 发送普通消息;
- 发送紧急消息;
- 广播消息;
- 接收消息;
- 取得消息队列上的未决消息数目;
- 清空消息队列;
- 取得消息队列标识符。

按照消息的紧急程度,可以将它们分为普通消息和紧急消息。紧急消息无论何时发送,都将被优先接收。

接收消息时,任务可以选择是否等待。如果等待还可以选择是按照先进先出(FIFO)方式等待,则还是按照任务优先级高低顺序等待。

3. 事　件

一个事件标志被用来通知其他任务或中断服务程序出现了一个预先定义的事件。事件机制用于任务之间、任务与中断服务程序之间的同步。

事件机制提供了复杂同步的功能。一个任务可以与多个任务或中断服务程序同步,一个或多个事件构成一个事件集。事件的特点有:

- 事件之间是相互独立的;
- 事件不提供数据传输功能;
- 任务可以同时等待多个任务;
- 对事件的等待方式有“与”和“或”两种;
- 事件无队列,即多次向任务发送同一事件,在事件标志未被接收处理之前,其效果等同于只发送一次。

对事件的基本操作可以有:

- 发送事件;
- 接收事件。

4. 异步信号

异步信号机制用于任务之间、任务与中断服务程序之间的异步操作。异步信号被任务(或

中断服务程序)用来通知其他任务某个事件的出现。

异步信号机制允许任务定义一个异步信号例程(ASR)。ASR 与 ISR(中断服务程序)的一个重要区别是:一个 ASR 对应于一个任务,而 ISR 对应于一组任务。当系统运行时,若外部中断出现且未被系统屏蔽,则任务的执行将被中断,系统转而执行与该中断相关的服务例程;同样,当一个异步信号被发送给某任务,该任务将中止其自身代码的运行,转而运行与该异步信号相关的例程。因此,异步信号机制也可以称作软中断机制。给任务发送异步信号对接收任务的当前执行状态没有任何影响。

一个或多个异步信号标志组成一个异步信号集。发送信号集是指发往目标任务的一个或多个异步信号的组合。待处理信号集是指发送给具有有效 ASR 的目标任务并等待处理的异步信号组合。

由于异步信号的接收任务在被调度、完成上下文切换后才处理异步信号,执行异步信号例程,所以异步信号提供了任务切换后进行扩展操作的能力。

对异步信号的基本操作可以有:

- 安装异步信号处理例程;
- 发送异步信号。

5. 管　道

有的实时内核提供管道通信功能。管道使用操作系统的 I/O 系统,它是一种由驱动程序管理的虚拟 I/O 设备,与其底层的消息队列相联系,可提供与消息队列互换的功能。

被创建的管道是命名的 I/O 设备。一旦管道被创建,任务就能够像使用标准 I/O 设备那样来打开、读/写管道,也可以设置其属性。如果任务试图向一个已满的管道执行写操作,则该任务将等待(被阻塞)。如果任务试图向一个空的管道执行读操作,则该任务也会被阻塞,直到有消息到达。像对消息队列的操作一样,ISR 能够向管道写数据,但是不能从管道读数据。

管道具备文件描述符的特征,因而能够提供消息队列所不具备的一个重要功能,即可以使用 select 函数,允许任务等待一个 I/O 设备集合之一的数据可用。图 4-10 示意了一个任务通过管道等待来自三个任务的一个消息。

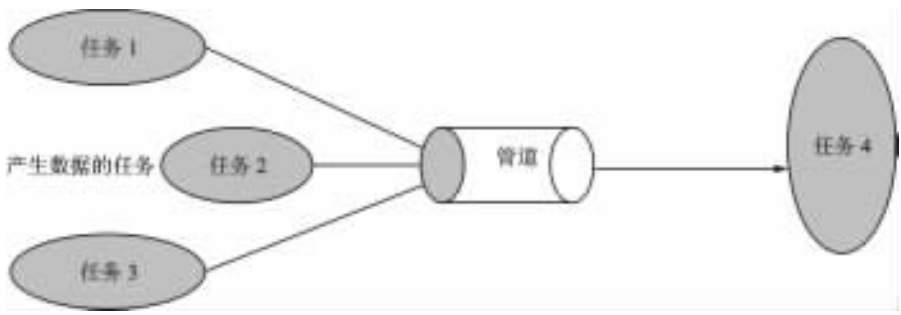


图 4-10　一个任务通过管道等待来自三个任务之一的消息

管道也能实现任务之间的客户/服务器通信模式。

4.2.6 内存管理

在 4.1 节讨论内核的关键设计问题时,已阐述了嵌入式实时内核所采用的内存管理机制是比较简单的。具体说来,嵌入式实时内核的内存管理一般是提供固定大小和可变大小两种动态内存分配机制。

固定大小的动态内存块分配是由分区管理功能提供的。一个分区是一个物理地址连续的内存区域,该内存区域可以被划分为多个大小固定(但是可配置)的缓冲区,应用申请和释放空间都对缓冲区进行。

可变大小的动态内存块分配是由堆管理功能提供的,堆也是一个物理地址连续的内存区域。从堆中申请的内存块大小虽然是可变的,但是应用每次得到的实际内存块大小是最小分配单位(比如页)的整数倍,最小分配单位可以在创建堆的时候由应用指定。另外,内核在堆分配的实现算法中应注意有效地避免内存碎片的产生。

在应用中可以创建多个分区和堆,以实现对多个内存区域的管理。

对分区的操作主要有:

- 创建分区;
- 删除分区;
- 从分区中申请内存块;
- 将内存块释放回分区中。

对堆的操作主要有:

- 创建堆;
- 删除堆;
- 从堆中申请内存块;
- 将内存块释放回堆中。

下面讨论关于内存保护的问题。

为提高可靠性,有些实时内核提供了内存保护的机制,其实现需要用到嵌入式微处理器的 MMU(Memory Management Unit)功能。MMU 提供了一种用于实现程序之间相互隔离、保护的硬件机制。操作系统通常利用 MMU 来实现一定的内存保护机制,实现操作系统与应用程序的隔离,以及应用程序与应用程序之间的隔离。这样可以防止应用程序破坏操作系统的代码和数据,防止应用程序对硬件的直接访问;对于应用程序来讲,也可以防止别的应用程序对自己的非法入侵,从而破坏自身的运行。这种关系如图 4-11 所示。

采用 MMU 便于在应用开发阶段发现更多的潜在问题,也便于问题的定位。

在采用内存保护机制后,应用程序如果要通信就只能通过操作系统提供的通信服务,如信号量、消息队列、事件、异步信号和管道等,而不能直接访问彼此的地址空间。

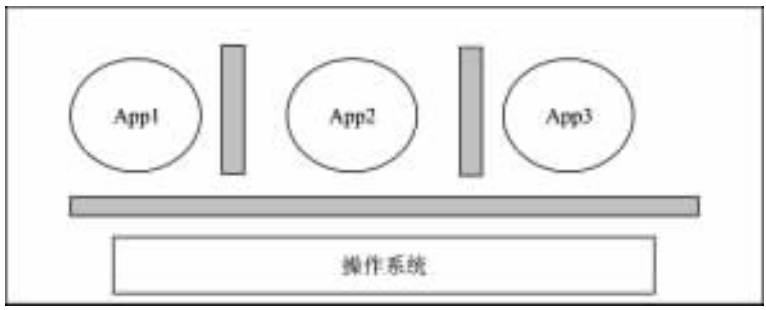


图 4-11　内存保护模式示意图

在一些低端的应用中出于成本的考虑，CPU 不具备 MMU 功能且运算速度较慢，存储空间有限，并且应用也很简单，系统软件的代码量受到严格限制（比如在有的嵌入式系统中，操作系统只占据几 KB 的空间）；另外，即使系统崩溃也不致造成重大损失。这样的应用对内存保护方面的要求非常弱，因此不需要内核提供内存保护的机制。

在一些高端的和安全关键的高可靠应用中，允许投入更多的硬件、软件成本，CPU 的速度非常快并且具备 MMU 功能，内存保护功能对时间性能造成的影响也可忽略，因而要求内核提供内存保护的机制。

综上所述，并不是所有的嵌入式实时操作系统都提供了内存保护功能，比如 WindRiver 公司的 VxWorks 在其 AE（Advanced Edition，2001 年推出的）版本才全面支持 MMU。

4.2.7　I/O 管理

很多嵌入式实时内核都提供了简单的输入/输出（I/O）管理，提供了结构化访问设备的方式。使用 I/O 管理的应用可以在系统配置表中定义设备驱动程序表，其中包括了设备驱动程序的入口地址。每一个设备驱动程序包括 Initialization（初始化）、Open（打开）、Close（关闭）、Read（读）、Write（写）和 Control（控制）等操作。

4.2.8　出错处理

嵌入式系统中出现的错误可以分为一般性错误和严重性错误。

一般性错误不会对系统直接造成致命的影响，其后果可以在一定范围内得到纠正，对这类错误的处理由内核和应用协作完成。比如，系统调用因为各种原因没能正确返回时，会返回一个表示错误类型的出错代码给调用者，调用者可以根据出错代码作一定的纠正。其实质是内核向应用提出错误，实际的处理由应用完成。

严重性错误会对系统造成致命的影响，其后果不可恢复。实时内核一般会提供这类错误的处理程序，使得系统在出现严重性错误后能处于一个确定的状态。在系统开发过程中还可

借助调试器、分析仪之类的软件，在系统出错时将控制权交给它们作进一步的处理，为用户分析出错原因提供一些信息。

另外，内核提供的任务重启动、用户扩展管理、任务超时处理（比如在单调速率调度中周期任务的超时处理）以及其他的异常处理都是出错处理的有效手段。

4.2.9 用户扩展管理

用户扩展管理向应用程序提供了灵活处理扩展的机制，在无须或者不可能更改内核代码的情况下，用户编写自己的扩展例程，内核在调用点调用这些例程。通过用户扩展程序入口（hook）和系统配置表可以实现内核在功能和规模上的可伸缩性。

比如，有时应用需要在任务上下文切换时增加相应的处理，又不需要涉及内核的修改。内核可以提供任务切换的钩子函数，允许当任务上下文切换时调用附加的函数。存储任务上下文的数据结构（比如任务控制块 TCB）可以被扩展，用户安装的钩子函数将在内核上下文中被调用执行。

内核可以提供的扩展点（即能够调用用户扩展程序的系统事件）包括（但不限于）以下几点：

- 任务创建时；
- 任务启动时；
- 任务重新启动时；
- 任务删除时；
- 任务上下文切换时；
- 任务第一次投入运行之前；
- 任务退出时；
- 系统错误时。

内核可以将扩展程序组织成扩展程序集，每个扩展程序集都包括内核提供的所有可扩展事件的入口。扩展程序集也是一种内核对象，拥有自己的标识符，可以被创建和删除。一个应用中可以有多个扩展集。

4.2.10 电源管理

对使用电池供电的嵌入式设备来说，电源消耗是一个值得关注的问题。即便不是使用电池供电的设备，也有必要尽可能地减少其能源消耗。因而在有些嵌入式系统应用领域中除了硬件，还对软件的电源管理功能提出了较高的要求。嵌入式系统的电源管理有如下特点：

- 首先硬件（包括 CPU 和外部设备）需具备一定的功耗特性，比如提供多种不同功耗的运行模式；
- 嵌入式应用系统本身有特定的电源管理需求；
- 嵌入式实时内核根据既定硬件平台的功耗特性，结合应用系统的电源管理需求，提供

相关的电源管理机制和应用编程接口；

- 开发人员利用内核的电源管理机制和应用编程接口，以及支持不同功耗运行模式的硬件驱动程序，实现最终应用系统的电源管理需求。

本节将主要从以下方面说明与内核电源管理相关的知识。

- 嵌入式硬件平台的功耗特性；
- 嵌入式实时内核提供的电源管理功能。

1. 嵌入式硬件平台的功耗特性

为了满足嵌入式应用的低功耗需求，CPU 和外围设备大都考虑了低功耗特性，并提供了可编程控制的多种功耗工作模式。对于硬件设备，如果提供了多种功耗工作模式，并能够通过软件编程的方法来实现工作模式之间的切换，就称该硬件设备为可编程功耗管理的设备。可编程功耗管理设备的目的在于提供功耗可以变化的工作模式。就 CPU 来说，为达到低功耗目的，提供了多种功耗管理机制，如：

- 允许停止 CPU 时钟；
- 能够工作于多种时钟频率；
- 能够工作于多种电压；
- CPU 中的模块能够被单独停止工作。

比如在 DragonBall 系列芯片 MC68VZ328 中，为适应 PDA 和智能电话等移动设备的需要，就采用了如下的低功耗处理措施：采用静态的 HCMOS 技术；具有低功耗的停止特性；各模块可以被单独停止工作；低功耗控制模式；可工作于 DC 到 33 MHz 的时钟频率；可工作于 2.7～3.3 V 的工作电压。

对于外部设备，也大都提供了多种功耗工作模式，如：

- 睡眠模式，能够维持设备的基本功能；
- 设备内部时钟保持运行状态的掉电模式，设备不能提供正常的功能；
- 设备内部时钟停止运行的掉电模式，设备不能提供正常的功能。

比如在 RTL8019 网卡中，就提供了三种级别的功耗模式：睡眠模式、网卡内部时钟保持运行状态的低功耗模式和网卡内部时钟停止运行的低功耗模式。

可编程功耗管理的设备是对系统进行电源管理的基础，但是要真正达到良好的功耗管理效果，不仅需要定义和实现好内核的电源管理功能，还要根据应用的不同特点，将硬件的功耗特性与内核的电源管理很好地结合起来，设计实现适合应用需要的电源管理方案。

2. 嵌入式实时内核提供的电源管理功能

从总体上说，内核的电源管理模块具有以下两大方面的功能：一是提供基本的电源管理机制，二是提供给应用系统操作电源管理的应用编程接口(API)。

(1) 提供电源管理机制

电源管理模块负责从系统角度，根据用户的配置，完成电源管理最基本的初始化工作(包

括电源管理需要的控制结构的初始化、电源操作模式的初始化等),按照电源操作模式的转换规则(又叫功耗模式转换规则)完成系统内电源操作模式的转换;在进入、退出相应的电源操作模式前,执行系统和应用控制的电源可控设备的操作。电源管理机制从系统角度为编写具有电源管理功能的应用提供了基础,但这部分内容对用户透明,而只有电源管理模块提供的电源管理 API,才是用户在编写具有电源管理功能的应用时能直接使用的资源。

(2) 提供电源管理 API

电源管理 API 包括:对电源操作模式的设置;电源可控设备的操作方式的设置;为了提供灵活性,允许用户对电源管理作一些配置、电源管理方面信息的查询。电源管理方面的信息主要包含系统当前工作的功耗模式、电源可控设备的状态和电池容量等信息,为系统进行功耗管理和根据当前可用电量进行自适应决策提供服务。

下面将具体说明嵌入式内核的一种较为通用的电源管理机制。在该机制中将系统的功耗模式分为四种:常规模式、空闲模式、休眠模式和睡眠模式。其中,常规模式的功耗最高,空闲模式和休眠模式的功耗次之,睡眠模式的功耗最低。

常规模式:为通常的工作模式,系统的大部分操作都在此模式下进行。在该模式下,CPU Core(指执行任何与计算相关的操作都需要上电的硬件内容,包括 CPU 时钟、Cache、系统总线和系统定时器)和所有的外部设备都处于上电状态,系统的功耗最大,性能也最好。

空闲模式:在该模式下,CPU Core 被关闭,而大多数外部设备仍然处于活动状态。该状态表示外部设备需要处于活动状态而 CPU 不需要处于活动状态时的一种低功耗模式。例如,终端设备的 LCD 的静态显示(即它的显示内容在显示过程中不需要发生变化)就属于这种情况。在该模式下没有活动的应用任务,所有应用任务都处于挂起或停止状态;而外部设备仍然处于活动状态,以便接收内部或外部的事件。

休眠模式:在该模式下,CPU 中的大多数模块和大多数外部设备(如 LCD 和 LCD 控制器)处于掉电状态,但是 CPU Core 仍然处于运行状态。在该模式下,仍然会处理一些对应用来说是无效的而且不需要切换功耗模式的外部事件,比如用户操作了触摸屏上无效的输入区域。

睡眠模式:为功耗最低的模式,只有系统实时时钟处于活动状态,而 CPU Core 和所有的外部设备都处于掉电模式。在该模式下,只有外部中断能够唤醒系统,并使系统首先进入休眠模式,然后再进入常规模式。

各种功耗模式之间的转换关系如图 4-12 所示。

- 上电后,系统工作于常规模式。如果有任务处于活动状态或是有外部事件发生,则系统将保持该模式;否则,系统将从常规模式切换到空闲模式。
- 在空闲模式下,可以设置并启动一个定时器,记录系统持续处于空闲模式的时间。如果在定时器到期之前有外部事件发生,则系统将回到常规模式,并取消定时器;如果在定时器到期之前一直没有外部事件发生,则系统将在定时器到期后切换到休眠模式。

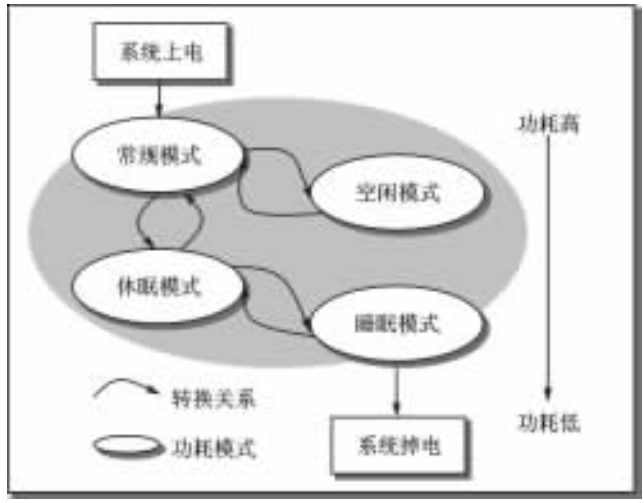


图 4-12　功耗模式之间的转换关系

由于在空闲模式下 CPU Core 处于关闭状态，因而系统需要短暂地回到常规模式进行超时处理，再切换到休眠模式。这是因为长时间没有外部事件意味着系统的功耗应该进一步降低。

- 在休眠模式下，也可以设置并启动一个定时器，记录系统持续处于休眠模式的时间。如果在定时器到期之前有外部事件发生，系统将回到常规模式，并取消定时器；如果出现的是对应用无效的外部事件，系统将保持在休眠模式；如果在定时器到期之前一直没有有效的外部事件发生，则系统会在定时器到期后进入睡眠模式，以便更进一步地降低功耗。休眠模式下 CPU Core 是处于运行状态的，因而可以直接进行切换。
- 在睡眠模式下出现外部中断时，系统将首先回到休眠模式，然后再回到常规模式。

由于降低功耗主要是通过关掉一些暂时不用的 CPU 模块或外部设备来实现的，并且高功耗模式到低功耗模式的切换还需要保存一些必要的数据，因此，工作模式的功耗越低，要恢复到常规模式的时间也越长。

4.3　嵌入式实时内核的重要性能指标

4.3.1　概　述

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，分为时间性能指标和存

储开销。

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；
- 中断响应时间；
- 中断恢复时间；
- 内核最大关中断时间；
- 任务上下文切换时间；
- 任务响应时间；
- 系统调用的执行时间。

上述几项中，内核最大关中断时间和任务上下文切换时间是评价内核实时性能最重要的两个技术指标。

在这些时间性能指标中，有些是可以直接测试得到的，比如中断恢复时间；而有些则必须通过对测试数据进行分析和处理得到，比如任务上下文切换时间。在后面对每个指标的描述中，将给出一些具体的测试或计算的方法。

一般来说，要测量某个物理量，所用工具或方法的精度要比被测对象的精度要求高出至少一个数量级。对于嵌入式强实时内核，其响应时间都在 μs 级（个别系统调用除外），那么测试方法的精度至少应该在 0.1 μs 级。

如果要对不同实时内核的时间性能进行比较，必须考虑测试得到的数据是否具有可比性。一般来说，应考虑如下两个因素：

① 测试的具体硬件环境，比如 CPU 的速度、访问存储器的速度、RAM 空间的大小、Cache 的大小以及是否使能 Cache 等，各项指标应该是在同一个物理设备上测试得到；

② 在比较系统调用的执行时间时，应确定其功能是否完全对等。

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，在实时内核的设计实现和应用开发中，除了上述各项时间性能指标，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他操作系统的明显区别之一。

4.3.2 中断时序图

在实时内核的各项时间性能指标中，有很多都是与中断有关的。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。为了更好地理解这些指标，本节将给出与中断有关

的一些时序图。

中断是一种硬件机制，用于通知 CPU 有个异步事件发生了。中断一旦被识别，CPU 保存部分（或全部）上下文（context）（包括各个寄存器的值），跳转到专门的中断服务程序（ISR）中执行。中断使得 CPU 可以在事件发生时才予以处理，而不必连续不断地查询是否有事件发生。

微处理器一般允许中断嵌套，也就是说在中断服务期间如果打开中断，则微处理器可以识别另一个更重要的中断，并服务于那个更重要的中断，如图 4－13 所示。

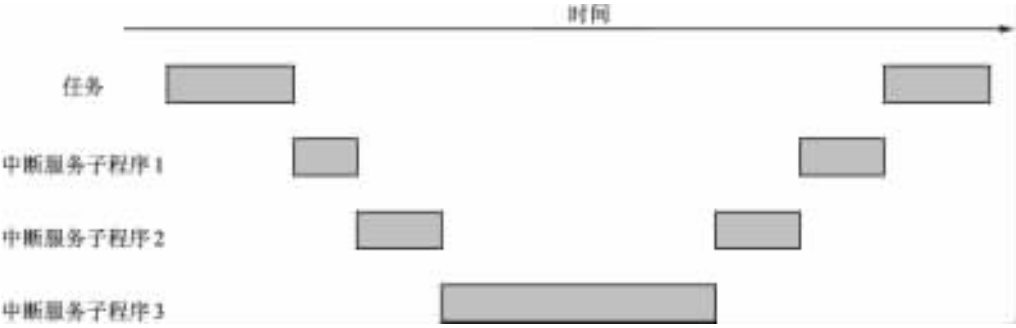


图 4－13　中断嵌套

下面将给出前后台系统、非抢占式调度内核和抢占式调度内核的中断时序图，其中抢占式调度内核的中断时序图是重点；为了加深理解，给出了前后台系统、非抢占式调度内核的中断时序图进行对比说明。在这些中断时序图中，为了简化和突出描述的重点，暂不考虑中断嵌套的情况。

从这些时序图中可以看出，中断服务程序完成事件处理后，完成以下工作：

- 在前后台系统中，程序回到后台程序；
- 对非抢占式调度内核而言，程序回到被中断了的任务；
- 对抢占式调度内核而言，让处于就绪态的优先级最高的任务运行。这个任务有可能是先前被中断打断的任务，也有可能是另一个在 ISR 执行过程中新就绪的任务。

（1）前后台系统的中断时序图

该中断时序图如图 4－14 所示。

（2）非抢占式调度内核的中断时序图

该中断时序图如图 4－15 所示。

（3）抢占式调度内核的中断时序图

对于采用基于优先级抢占式调度内核的系统，中断返回时可能出现两种不同的情况，如图 4－16中的 A 和 B 所示。在后一种情况下，中断恢复过程的时间要长一些，因为内核要做任务切换。

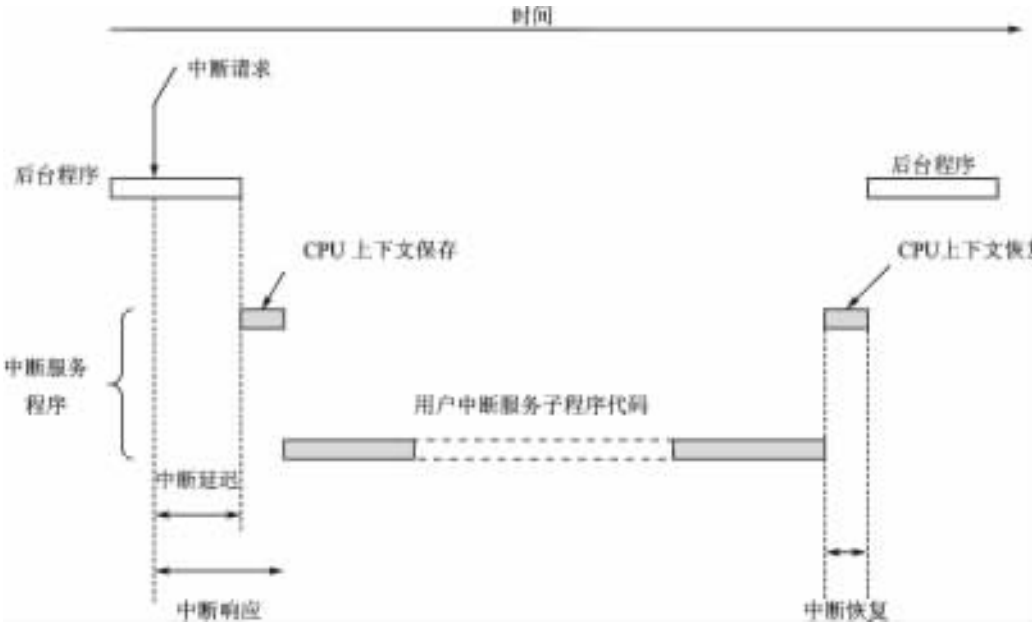


图 4－14　前后台系统的中断时序图

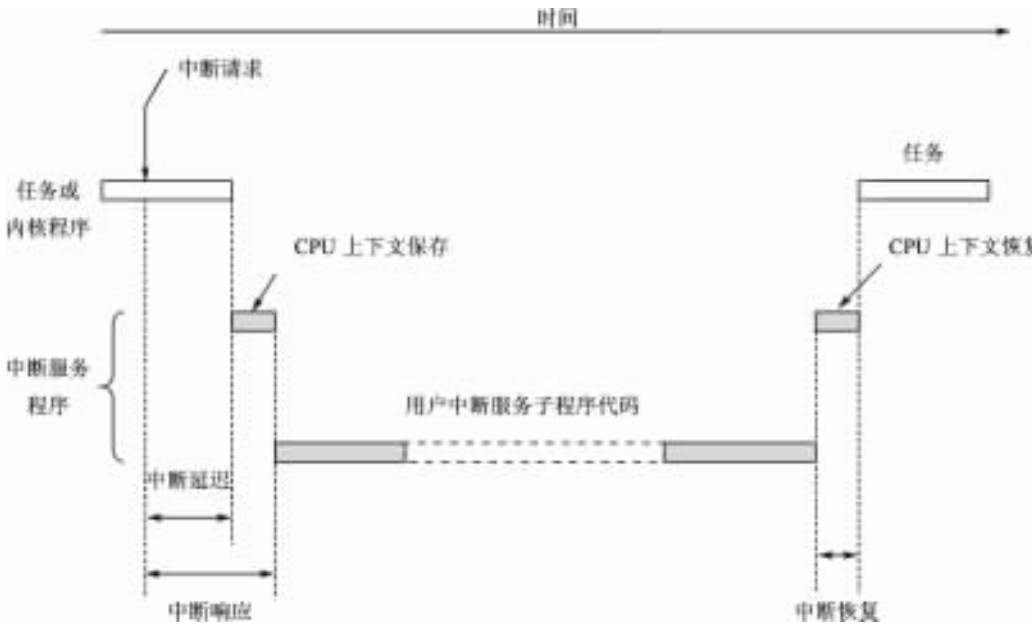


图 4－15　非抢占式调度内核的中断时序图

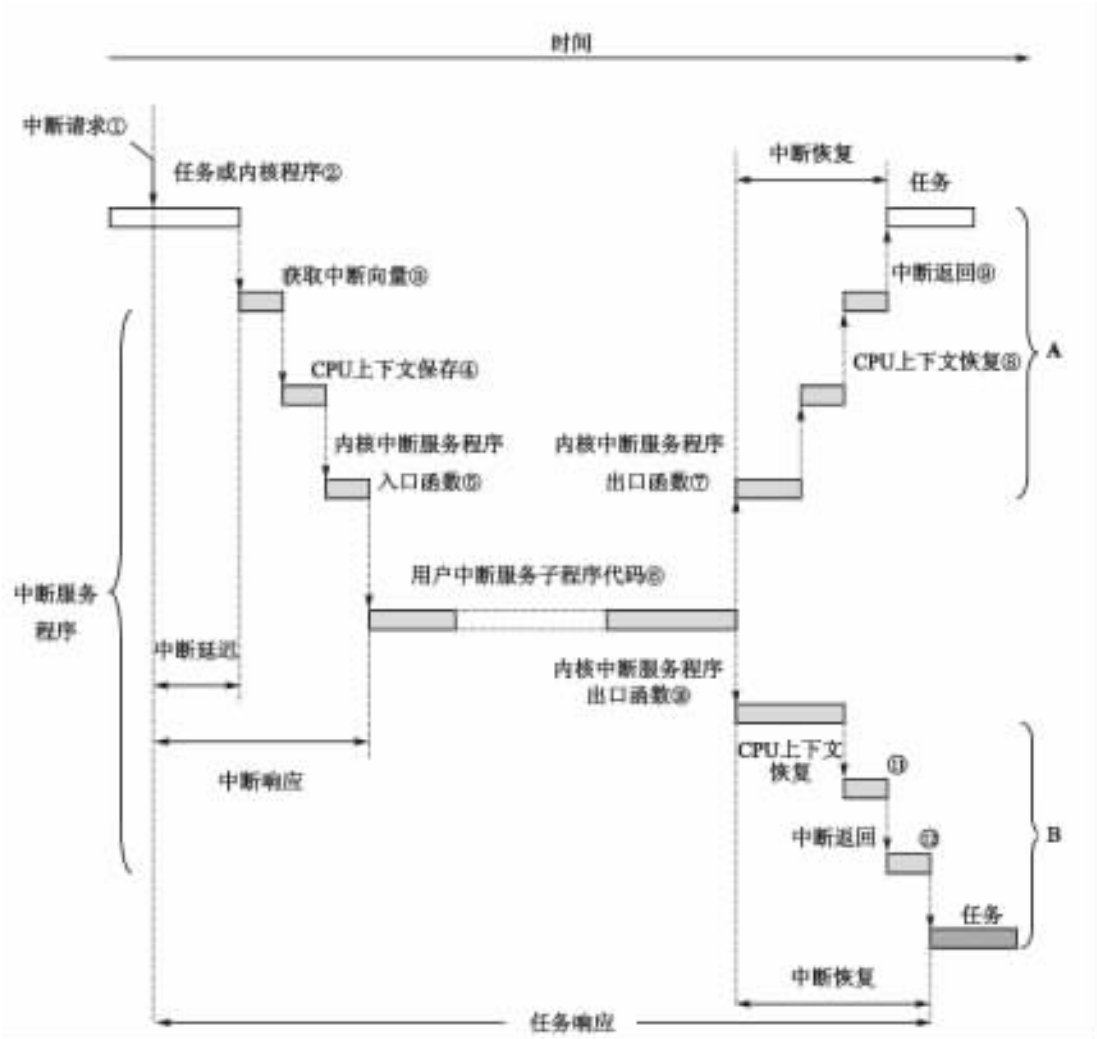


图 4－16　抢占式调度内核的中断时序图

中断服务程序的示意代码如下列程序清单所示：

① **保存** CPU **寄存器；**

② **调用内核中断服务程序入口函数；**

③ **执行用户代码做中断服务；**

④ **调用内核中断服务程序出口函数；**

⑤ **恢复** CPU **寄存器；**

⑥ **执行中断返回指令。**

下面对时序图 4－16 加以详细说明。

① 中断来到了但还不能被 CPU 识别，也许是因为 CPU 还没执行完当前指令，或者是因为中断被内核或用户应用程序关闭了。

② CPU 执行完当前指令并且中断打开，该中断被响应。

③ CPU 在其中断响应周期中获取中断向量，跳转到中断服务程序。

④ 中断服务程序保存 CPU 上下文（比如寄存器的内容）。

⑤ 中断服务程序调用内核中断服务程序入口函数，通知内核系统已进入中断处理中了，该入口函数将中断嵌套层数加 1，如果内核提供了专门的中断堆栈，则进行中断堆栈换入。

⑥ 用户的中断服务代码被执行，可以真正开始服务于申请中断的设备了，其功能完全取决于应用本身。用户中断服务中做的事要尽可能地少，把大部分的工作留给协同的任务去做。中断服务程序通知某任务去做事的方法是调用内核提供的同步或通信的系统调用，这可能导致接收消息的任务就绪。

⑦ 用户中断服务完成后，调用内核中断服务程序出口函数，通知内核系统退出了此次中断，将中断嵌套层数减 1。当嵌套层数减到 0 时，包括嵌套的中断在内的所有中断就都完成了；如果内核提供了专门的中断堆栈，则进行中断堆栈换出。此时内核中断服务程序出口函数还有一个功能就是执行调度程序，当然其前提是系统允许调度。调度程序要判断是否需要做任务切换，这时可能出现两种不同的情况。一是先前被中断的任务仍然是系统中优先级最高的就绪任务，系统将返回到被中断的任务继续执行（情况 A），不需要做任务切换⑦。二是先前被中断的任务已经不是系统中优先级最高的就绪任务了，内核将调度执行另一个任务（情况 B），需要做任务切换⑩）。出现这种情况的原因有多种，有可能是该中断服务程序（或其他任何一个嵌套的中断服务程序）使得另一个优先级更高的任务进入了就绪态，也有可能是先前被中断任务的状态被中断服务程序改变了，比如使用内核调用将该任务挂起。从上面的时序图中也可明显看出，在第二种情况下内核中断服务程序出口函数的执行时间要长一些。

⑧ 在不进行任务切换的情况下，CPU 上下文只是简单地恢复。

⑨ 执行中断返回指令。

⑩ 如第⑦项所述，用户中断服务完成后，调用内核中断服务程序出口函数；由于需要做任务切换，该函数的执行时间较长。

⑪ 如果要进行任务切换，新任务的上下文内容要恢复到 CPU 中。

⑫ 执行中断返回指令。

注意：

- 如果用户服务程序执行得非常快，并且不使用内核系统调用完成与任务的同步或通信，那么在它执行期间可以不允许中断嵌套，也不需要调用入口/出口函数通知内核系统进入/退出了中断服务。任务和这个中断服务程序通信的惟一方法是通过全局变量，在这种情况下时序图要简单得多。

- 图 4－16 只表示了没有中断嵌套的情况。如果存在中断嵌套，则该次中断服务完成后会回到被嵌套的、尚未完成的中断服务中。
- 如果在中断服务完成时系统不允许调度，那么还是会回到先前被中断的任务，而不管是否有新的任务就绪。关于这一点要非常小心，如果应用禁止调度，但在中断服务里面又使得先前被中断的任务挂起，那么系统将会出错。
- 在图 4－16 中假设所使用的内核本身是不可抢占的。如果系统采用的是可抢占内核，则其时序图将更为复杂一些，在此不再赘述。

4.3.3　中断延迟时间

中断延迟时间是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。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。中断延迟时间可以由下面的表达式表示，即

中断延迟时间＝最大关中断时间＋中断嵌套的时间＋
硬件开始处理中断到开始执行 ISR 第一条指令之间的时间

“硬件开始处理中断到开始执行 ISR 第一条指令之间的时间”由硬件决定。

“中断嵌套的时间”与具体的应用有关，不同的应用可能同时发生的最大嵌套层数不同，每个 ISR 的执行时间不同。这段时间可能是不确定的。

由于中断是外部异步事件，故不能确定何时会发生中断，并且不能确定发生中断时的系统状态是处于开中断的状态，还是处于关中断的状态。即便是在关中断状态下，也不能确定已经关闭了多长时间，以及还将关闭多长时间，因此在确定中断延迟时间时，要使用最坏情况下的关中断时间，即最大关中断时间。

“最大关中断时间”取决于两方面的因素。首先，内核关中断时间，内核在执行一些临界区的代码时采取了关中断；其次，应用关中断时间，即在应用程序中也可以关中断。关中断的最长时间应该是这两种关中断时间的最大值，即

最大关中断时间＝max［max（内核关中断时间），max（应用关中断时间）］

内核最大关中断时间将在 4.3.4 节详细描述。

测试方法

注意：

- 本节及后面各节所描述的测试方法针对的是抢占式调度内核的情况。
- 有关时间测试的方法很多，这里描述的只是其中一种；后面给出的关于其他内核时间性能指标的测试方法也是如此。

- 给出的测试方法主要是基于硬件定时器的，设置和获取定时器计数值的操作本身要消耗额外的时间，因此在得到测试时间后，应该减去这些额外的时间。另外还要注意定时器的数据长度和可能的计数溢出问题。

如果暂不考虑内核的关中断时间(它的测试比较麻烦，在 4.3.4 节将详细描述)，则中断延迟时间可以使用如下方法测试得到(以 X86 平台为例)。

使用一个外部定时器(8253 的某个通道)定时向 CPU 发送中断，中断服务程序一开始就立即读这个定时器的计数值。对定时器进行设置，使得每次中断都使其计数值复位到零，因此每次读得的值乘以计数脉冲周期就是中断延迟时间。如果定时器被设置成减法计数方式，那么用设置值(val1)减去读取值(val2)，再乘以计数脉冲周期就得到了中断延迟时间。

例如，PC 提供的时钟晶振频率为 1.193 18 MHz，则定时器计数周期为 1/1.193 18 MHz，即 0.84 μs。如果设置定时器通道的工作方式为减 2 计数，那么计算时间的公式为

$$(val1-val2)/2\times 0.84\ \mu s$$

在考察中断延迟时间时，还要考虑到中断嵌套的影响。由于中断嵌套导致的延迟是与具体应用有关的，具有不确定的因素，故这里给出一个在考虑中断嵌套时测试中断延迟时间的例子，以便和没有中断嵌套的情况进行对比，让读者了解两者的区别。实际测试结果也表明，是否存在中断嵌套对这个时间指标的影响是很大的。

比如在上述测试过程中就可以区分两种情况。第一种情况，不考虑实时时钟的影响，即关掉实时时钟，并且假定测试用的定时器的中断级别在其他的外部硬件中断中是最高的，因此在其中断服务过程中不会有中断嵌套存在。第二种情况，考虑实时时钟的影响，更能够反映实时应用的情况。因为在实时应用中实时时钟是不可少的，其中断优先级在所有的外部硬件中断中也是最高的，因此在定时器中断服务过程中可能产生实时时钟中断，发生中断嵌套。

考虑实时时钟的中断延迟时间的测试方法如下：

① 初始化中断控制器 8259，使其工作在级联方式。

② 将定时器 8253 的通道 0 和通道 1 都初始化为中断方式，通道 0 连接在主 8259 上，其中断优先级是最高的；通道 1 连接在从 8259 上，其中断优先级较低，模拟其他的外部硬件，产生被测的中断信号。

③ 通道 1 的中断产生后，在对应的中断服务程序的第一条指令就去读它的计数值，之后计算中断延迟时间的方法同前所述。

4.3.4 内核最大关中断时间

如何使内核的最大关中断时间尽量地减小，是设计内核时在实时性方面需要仔细考虑的问题。比如，假设在一开始进入内核系统调用时就关闭中断，试图采取这种粗放的方法来进行互斥，以达到对内核服务中的关键(临界)代码的保护，那么就会造成比较长的关中断时间。其实通过对内核代码的仔细分析，可以发现其中存在一些非关键(临界)的代码段；在这些地方开

中断，增加内核代码中的可抢占点，就可以大大缩短关中断时间。

测试方法

内核最大关中断时间可以通过测试各个系统调用的关中断时间得到。

某些实时内核在实现时，统一采用了一对“关/开”中断的函数或宏调用，因而可以在这对函数或宏调用中嵌入指令，使得系统调用在进入第一层关中断操作时，读取一次定时器的值；在退出最后一层开中断操作时，再次读取定时器的值。两次的值相减，再乘以计数脉冲周期，就得到了关中断时间。

为了获得内核最大关中断时间，需要对所有函数调用的关中断时间进行测试，包括各种可能的情况，然后取其中的最大值。

4.3.5　中断响应时间

在这里，从内核的角度出发，给出“中断响应时间”的一种定义：中断响应时间是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。

应注意中断延迟时间与中断响应时间的区别：前者指到中断服务程序的第一条指令，而后者指到用户的中断服务程序的第一条指令。

根据 4.3.2 节给出的中断时序图，对于前后台系统和采用非抢占式调度内核的系统，保存 CPU 上下文（主要是其内部寄存器的内容）以后立即执行用户的中断服务子程序代码，其中断响应时间由下面的表达式给出，即

中断响应时间＝中断延迟＋保存 CPU 内部寄存器的时间

对于采用抢占式调度内核的系统，处理中断时先要做一些处理，确保中断返回前调度程序能正常工作，即要先调用一个特定的函数（前面提到的内核中断服务程序入口函数）。该函数通知内核即将进行中断服务，使得内核可以跟踪中断的嵌套，以便在解除中断嵌套后进行重调度。内核一般把该入口函数和相应的出口函数提供给用户，用户可选择是否要在 ISR 中使用这两个函数。抢占式调度内核的中断响应时间由下面的表达式给出，即

中断响应时间（抢占式调度）＝中断延迟＋保存 CPU 内部寄存器的时间＋
内核中断服务程序入口函数的执行时间

与中断延迟时间一样，中断响应时间应该是系统在最坏情况下的响应中断时间。例如某系统 100 次中有 99 次在 50 μs 之内响应中断，只有 1 次响应中断的时间是 250 μs，则只能认为该系统的中断响应时间是 250 μs。

测试方法

中断响应时间与“中断延迟时间”的测试方法相同。

4.3.6 中断恢复时间

中断恢复时间是用户中断服务程序结束后回到被中断代码之间的时间。对于采用抢占式调度内核的系统中发生任务切换的情况,中断恢复时间还指用户中断服务程序结束后到开始执行新任务代码之间的时间。

在前后台系统和采用非抢占式调度内核的系统中,中断恢复时间很简单,只包括恢复CPU上下文(主要是其内部寄存器的内容)的时间和执行中断返回指令的时间,由下面的表达式给出(在没有中断嵌套的情况下),即

中断恢复时间=恢复CPU内部寄存器的时间+执行中断返回指令的时间

对于采用抢占式调度内核的系统,中断的恢复要复杂一些。通常在用户的中断服务子程序的末尾要调用内核中断服务程序出口函数,用户可选择是否要在ISR中使用这个函数,但是必须与入口函数的使用相匹配。该出口函数判断是否脱离了所有的中断嵌套,如果脱离了嵌套,则内核要判断是返回到原来被中断的任务,还是进入另外一个优先级最高的就绪任务。在这两种情况下的中断恢复时间都可由下面的表达式给出,即

中断恢复时间(抢占式调度)=内核中断服务程序出口函数执行时间+
恢复即将运行任务的CPU内部寄存器的时间+执行中断返回指令的时间

测试方法

下面给出在抢占式调度系统中测试中断恢复时间的方法。

① 中断结束后不发生任务切换的情况　使一个任务在一个预知的位置被中断打断,在相应的中断服务程序退出之前(最后一条指令处)开始计时,被打断的任务一旦获得运行就立即获取计时时间,这一时间扣除计时本身消耗的时间后,便是中断恢复时间。

② 中断结束后发生任务切换的情况　创建两个任务A和B,B的优先级较高,创建后启动任务A但不启动任务B(即B尚不具有竞争CPU的权利)。任务A运行,并被定时器中断打断。在定时器中断服务程序中启动任务B,并在中断退出之前(最后一条指令处)开始计时。系统切换到任务B运行,任务B的第一条指令就立即去获取计时时间,这一时间扣除计时本身消耗的时间后,便是中断恢复时间。

4.3.7 非屏蔽中断

前面讨论的各种时间性能指标都是针对可屏蔽中断而言的。然而有时中断服务必须来得尽可能快,在这种情况下内核引起的延迟将变得不可忍受。这时可以使用非屏蔽中断(NMI),绝大多数微处理器都有非屏蔽中断功能。通常非屏蔽中断留做紧急处理用,或用于时间要求最苛刻的中断服务。根据下列表达式即可确定非屏蔽中断的中断延迟时间、中断响应时间和中断恢复时间。

中断延迟时间＝指令执行时间中最长的那个时间＋开始做非屏蔽中断服务的时间
中断响应时间＝中断延迟时间＋保存 CPU 寄存器的时间
中断恢复时间＝恢复 CPU 寄存器的时间＋执行中断返回指令的时间

在非屏蔽中断服务程序中不能使用内核提供的服务，非屏蔽中断是关不掉的，因而不能在非屏蔽中断中处理临界区代码。然而向非屏蔽中断传递参数或从中获取参数还是可以进行的。参数传递必须使用全局变量，全局变量的长度必须是一次能够完成读或写操作的，不能是两个以上分离的字节，需要两次以上的读或写操作。

4.3.8　中断处理时间

用户的中断处理是由应用决定的，并不是内核的组成部分，但在这里单独提出来的目的是让读者对中断处理时间的要求有个清晰的认识。

虽然中断服务的处理时间应该尽可能地短，但是对它并没有绝对的限制，根据应用的情况，中断服务需要多长时间就应该给它多长时间。在大多数情况下，中断服务程序应识别中断来源，从产生中断的设备取得数据或状态，并通知真正处理该事件的那个任务。

当然应该考虑到，通知一个任务所花的时间是否比处理这个事件本身所花的时间还多。在中断服务中通知一个任务做事件处理可以采用内核提供的各种同步或通信机制，比如信号量、消息队列等，这是需要一定时间的。如果事件处理所花的时间短于给任务发通知的时间，就应该考虑在中断服务程序中做事件处理，并在这期间打开中断，以允许优先级更高的中断进入并优先得到服务。

4.3.9　任务上下文切换时间

在多任务系统中，任务上下文切换是指 CPU 的控制权由运行任务转移到另外一个就绪任务时所发生的事件，当前运行任务转为就绪（或者等待、删除）状态，另一个被选定的就绪任务成为当前运行任务。任务上下文切换时间包括保存当前运行任务上下文的时间、选择下一个任务的调度时间以及将要运行任务的上下文的恢复时间。任务切换是在实时系统中频繁发生的动作，其时间的快慢直接影响到整个实时系统的性能。

从图 4－17 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。

任务上下文（context，即当前的状态）是指 CPU 寄存器中的全部内容。这些内容保存在任务的当前状态保存区，比如任务自己的堆栈或任务控制块（TCB）之中。入栈工作完成以后，内核就把下一个将要运行任务的当前状态从该任务的保存区重新装入到 CPU 的寄存器中，并开始下一个任务的运行。任务切换过程增加了应用程序的额外负荷。CPU 内部的寄存器越多，额外负荷就越重。因此，做任务切换所需要的时间与 CPU 有多少寄存器有关，实时内

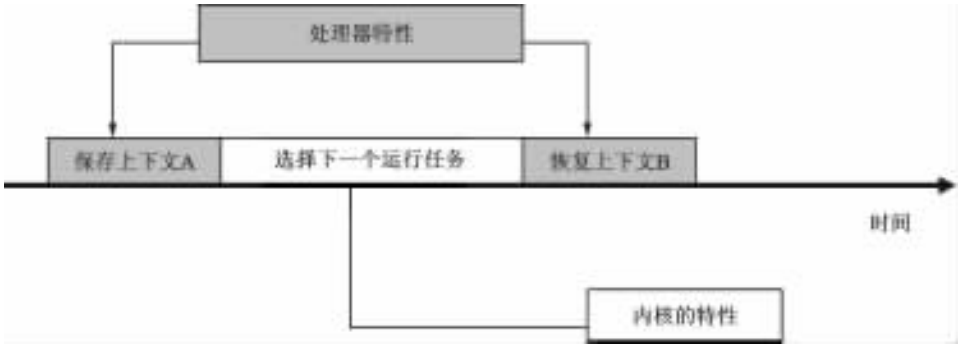


图 4-17　任务上下文切换时序图

核的性能不应该以每秒钟能做多少次任务切换来评价。

在带有浮点协处理器的系统中，任务上下文切换的时候还要对浮点协处理器的内容进行保存和恢复，这也是相当耗时的。在可行的情况下，内核可以采取优化策略，并不是每次做任务切换的时候都保存和恢复浮点协处理器的内容。

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。强实时内核要求调度过程所花费的时间是确定的，即不能随系统中就绪任务的数目而变化。这与具体实现调度算法时所采用的数据结构有关，比如在基于优先级调度的内核中，采用优先级位图的数据结构来保证此选择过程的时间确定性。

测试方法

对于任务之间的切换时间，可以通过两次测试间接地得出。

（1）第一次测试

第一次测试称为“交替悬置/恢复任务”（ping suspend/resume tasks）的时间测试，如图 4-18所示。测试案例设计为用一个低优先级的任务来恢复一个被悬置的高优先级的任务，然后这个高优先级的任务又立即使自己悬置，如此周而复始。

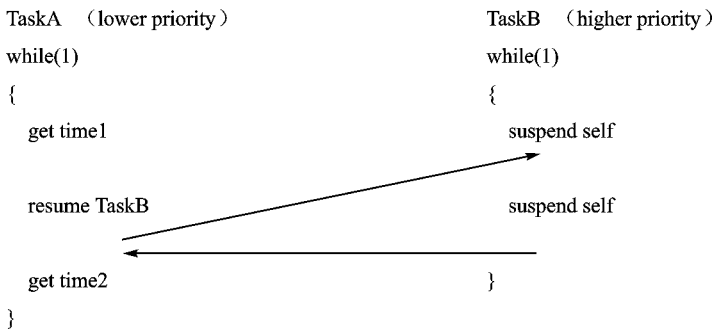


图 4-18　交替悬置/恢复任务的时间测试

用一个程序周期中获得的 time2 减去 time1 就可以得到交替悬置/恢复任务时间，包括了两次任务切换时间，以及任务悬置(suspend)和恢复(resume)各一次的时间。

(2) 第二次测试

第二次测试称为“悬置/恢复任务”(suspend/resume task)的时间测试，如图 4-19 所示。测试案例设计为用一个高优先级的任务反复悬置和恢复一个低优先级的任务。

```
TaskA  (lower priority)
while(1)
{
}
```

```
TaskB  (higher priority)
while(1)
{
 get time1
 suspend TaskA
 resume TaskA
 get time2
}
```

图 4-19　悬置/恢复任务的时间测试

用一个程序周期中获得的 time2 减去 time1 就可以得到一次任务悬置和一次任务恢复的时间，不包括任务切换的时间。

(3) 计算任务上下文切换时间

计算任务上下文切换时间的公式为

$$\text{任务上下文切换时间} = (\text{交替悬置/恢复任务时间} - \text{悬置/恢复任务时间})/2$$

4.3.10　任务响应时间

任务响应时间是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。任务响应时间又称调度延迟。

在实时系统中，任务在等待某些外部事件来激活它们。当一个中断发生时，如果该中断对应着一个比当前运行任务优先级更高的任务，即该中断的服务程序使这个高优先级任务就绪，则当前运行的任务必须迅速终止，使这个高优先级的(实时)任务进入。任务响应时间就是指这一过程的时间，因此这个性能指标很重要。

内核的调度算法是决定调度延迟的主要因素。在基于优先级的抢占式调度的内核中，调度延迟是比较小的。因为这种内核是即时抢占的，一旦系统或任务状态发生了变化，有任务抢占的要求时，内核的重调度过程就会被调用。有的内核并不是随时都能发生抢占的，它们有一定的调度时刻，其调度延迟就较前一种内核长。

调度延迟还受到另外一个因素的影响，就是禁止任务切换即关调度。关调度是一种互斥手段。在关调度的情况下，即使中断服务程序使得一个优先级更高的任务就绪，中断返回时也不能

切换到这个高优先级任务运行。一般来讲,内核程序中是不会关调度的,关调度可以是内核提供给应用的一个系统调用,因此在应用中要注意慎重使用它,以免给系统带来过长的关调度时间。

图 4-20 展示了从中断发生到所对应的任务开始运行的过程,可见任务响应时间受到很多因素的影响,最长的响应时间就是这些潜在的不同延迟的最大值的总和。

图 4-20 从中断到对应任务的运行

测试方法

在一个实时系统中,设计两个任务 Task1 和 Task2,Task2 的优先级高于 Task1 的优先级。Task2 由于等待一个信号量而被阻塞,这时发生了一个中断激活了 Task2(即释放了该任务等待的信号量),从发生中断到中断服务程序完成并切换到 Task2 的时间即是任务 Task2 的响应时间。考虑到中断嵌套的影响,测试同样也可分为关闭实时时钟和打开实时时钟两种情况。将定时器的通道 0(其中断优先级在所有外部中断中是最高的)设置为实时时钟,用通道 1(中断优先级较低)代替外部硬件。系统开始运行后首先创建 Task1,创建一个初始值为 0 的信号量 S,将定时器的通道 1 设置为"加计数"方式,每当计数满后产生中断。该测试用例的运行流程如图 4-21 所示。

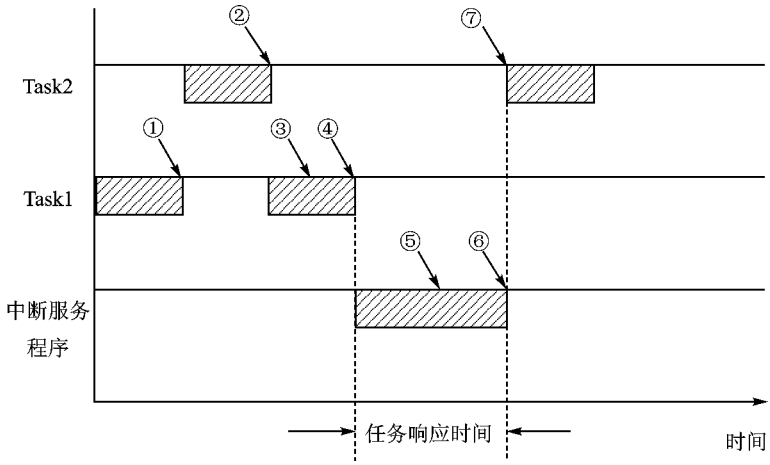


图 4-21 任务响应时间的测试

图 4－21 中：

① Task1 创建并启动 Task2，由于 Task2 的优先级比 Task1 的高，系统切换到 Task2；

② Task2 申请信号量 S，由于该信号量的值为 0，Task2 挂起，系统切换到 Task1；

③ Task1 设置一个标志，标明 Task2 已经在等待信号量 S 了，以便控制定时器通道 1 的中断服务程序的行为；

④ 定时器的通道 1 产生中断，系统进入相应的中断服务程序；

⑤ 中断服务程序判断出标志已经设置后，释放信号量 S；

⑥ 中断服务完成后发生任务切换，系统切换到 Task2；

⑦ Task2 一运行就立即读定时器通道 1 的计数值。

将读到的计数值乘以计数脉冲周期就可以得到 Task2 的任务响应时间了。

4.3.11　系统调用的执行时间

系统调用的执行时间也是评价一个实时内核性能优劣的指标。然而，由于调用时的情境和参数的差别，每一个系统调用的每次执行，可能会经历不同的路径，其执行时间都不是一个定值，而是在一定范围内波动。对某个系统调用而言，人们关心的是它的最大执行时间，所以在测试系统调用执行时间时，应该根据可能的使用情况设计不同的测试用例，获取其最大值。根据实时内核的确定性要求，在最坏情况下，系统调用的最大执行时间是可预测的。

对于实时内核提供的系统调用来说，它们的执行还可能导致发生任务切换，因此在测试这些系统调用的执行时间时，还要考虑发生任务切换的情况，比如表 4－2 所列的一些情况。

表 4－2　系统调用和任务切换

系统调用	调用情境	调用结果
启动任务	被启动任务的优先级低于当前任务的优先级，或不满足任务切换的条件（比如系统处于关调度状态等）	不发生任务切换
	被启动任务的优先级高于当前任务的优先级，且满足任务切换条件（比如系统处于开调度状态等）	发生任务切换
挂起任务	任务挂起自己	发生任务切换
	挂起其他任务	不发生任务切换
解挂任务	被解挂的任务就绪（即不等待其他资源）且该任务满足抢占当前任务的条件	发生任务切换
	被解挂的任务就绪（即不等待其他资源）但该任务不满足抢占当前任务的条件（比如优先级较低），或者被解挂的任务还要等待其他资源	不发生任务切换

续表 4－2

系统调用	调用情境	调用结果
删除任务	任务删除自己	发生任务切换
	删除其他任务	不发生任务切换
设置任务优先级	改变当前运行任务优先级，导致其他就绪任务优先级比当前运行任务高，且满足切换条件	发生任务切换
	改变其他任务优先级，导致其他就绪任务优先级比当前运行任务高，且满足切换条件	发生任务切换
	改变当前运行任务优先级，但不影响它与其他就绪任务的优先级高低关系，或不满足切换条件	不发生任务切换
	改变其他任务优先级，但不影响当前运行任务与其他就绪任务的优先级高低关系，或不满足切换条件	不发生任务切换
申请信号量	申请失败，且等待	发生任务切换
	申请成功	不发生任务切换
	申请失败，但不等待	不发生任务切换
释放信号量	有等待任务接收了信号量，且满足切换条件	发生任务切换
	没有等待任务	不发生任务切换
	有等待任务接收了信号量，但不满足切换条件	不发生任务切换
发送消息	有等待任务接收消息后就绪，且满足切换条件	发生任务切换
	有等待任务接收消息，但不满足切换条件	不发生任务切换
	没有等待任务	不发生任务切换
接收消息	接收失败，且等待	发生任务切换
	接收成功	不发生任务切换
	接收失败，但不等待	不发生任务切换
发送事件	有等待任务在接收到事件后就绪，且满足切换条件	发生任务切换
	有等待任务，但它在接收到事件后不满足切换条件	不发生任务切换
	没有等待任务	不发生任务切换
接收事件	接收失败，且等待	发生任务切换
	接收成功	不发生任务切换
	接收失败，但不等待	不发生任务切换

测试方法

① 初始化定时器的某个通道，将其作为测试时钟，设置适宜的工作方式和计数初值；

② 在系统调用之前读出定时器的当前值（设为 val1）；

③ 在系统调用结束后再次读出定时器的当前值（设为 val2）；

④ 计算系统调用的时间，用 val1 减去 val2，再乘以计数脉冲周期，即可得到系统调用的执行时间。

对于在系统调用执行过程中发生任务切换的情况，测试值 val2 应该是在任务切换后进行取值，如下例所示：

```
Task1()
{
……
get   val1;          /* 取 val1 的值 */
内核 API();          /* 某内核系统调用，其执行导致系统发生任务切换，切换到 Task2 运行 */
……
}
Task2()
{
 ……
 get   val2();       /* 取 val2 的值 */
……
}
```

在实时多任务应用中，快速的同步与通信是很重要的。信号量、消息队列等是常用的同步与通信机制，它们执行效率的高低会影响到整个系统的性能。下面具体来看几个与之相关的时间的测试。

1. 交替信号量的时间测试

交替信号量时间是指在两个不同优先级的任务之间交替释放和获取信号量的时间。具体的测试方法如图 4－22 所示。

用 time2 减去 time1 就得到了在两个任务之间交替信号量的时间，它包括了两次任务切换的时间。

2. 取得/释放信号量的时间测试

测试在同一个任务中反复取得和释放信号量的时间。它与交替信号量时间的差别在于不包含任务切换的时间。具体的测试方法如图 4－23 所示。

用 time2 减去 time1 就得到了取得和释放信号量的时间。

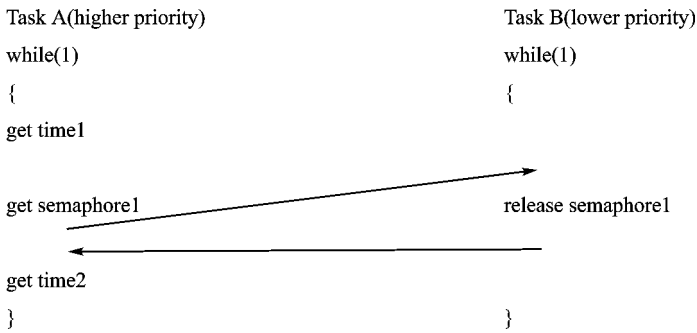


图 4-22　交替信号量的时间测试

```
Task A
while(1)
{
get time1
get semaphore1
release semaphore1
get time2
}
```

图 4-23　取得/释放信号量的时间测试

3. 交替消息队列传输时间

交替消息队列传输时间主要测试在两个任务之间通过消息队列进行通信的时间。具体的测试方法如图 4-24 所示。

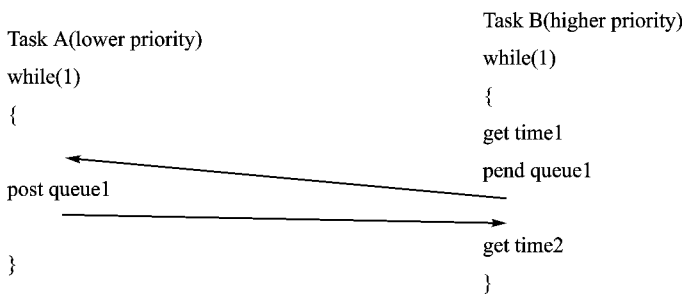


图 4-24　交替消息队列传输时间测试

用 time2 减去 time1 就可以得到交替消息队列的传输时间，其中包含了两次任务切换的时间。

4.3.12　有关时间确定性的测试

一个实时内核是否满足确定性的要求，也是可以通过测试得出结论的。下面是一些有关实时内核确定性测试的问题。

- 强实时内核，其各项时间性能指标应该不受系统负载、系统当前状态的影响。可以在内核所支持的最大任务数范围内，构造任意多个任务；在不同的时刻，这些任务可能处于不同的状态（就绪、阻塞、挂起或运行），测试前面所述的各项重要的时间性能指标（比如任务上下文切换时间），以及相关系统调用（比如创建任务）的执行时间等。
- 在测试同步或通信机制的系统调用时，构造多个任务阻塞在同一个同步或通信对象上的情况，测试获得和释放一个同步或通信对象的时间是否与阻塞在这个对象上的任务数目有关。

4.3.13　嵌入式实时内核的存储开销

对于前/后台系统，它对存储器容量的需求仅仅取决于应用程序；而使用多任务内核时的情况则很不一样，内核本身需要额外的代码空间和数据空间。

内核的代码大小取决于多种因素，包括内核的特性，从几 KB 到上百 KB 都是可能的。

内核的数据空间通常又被称为内核的工作区，其大小包括固定的 RAM 需求和可变（可配置）的 RAM 需求，取决于如下因素：

- 内核的系统变量需求（固定）。
- 内核对象的控制结构及相关空间需求（可配置）。

  在多任务应用中要创建任务、消息队列、事件、堆、分区、信号量、软定时器和用户扩展等各种内核对象，其数目根据应用的需要是可变的。这部分空间需求有两种：一是每个内核对象的控制结构（比如任务控制块、消息队列控制块、事件控制块等）所占用的空间；二是某些对象本身的存储空间，这些对象包括消息队列、堆和分区等。
- 堆栈的空间需求，包括任务堆栈（可配置）、中断堆栈（可配置）。

  在多任务应用中，每个任务都是独立运行的，内核给每个任务提供了单独的堆栈空间。应用开发者决定分配给每个任务的堆栈空间时，应尽可能使之接近实际需求量（但这实际上是一件很困难的事）。决定堆栈空间的大小，不仅需要计算任务本身的需求（局部变量、函数调用等），如果允许中断嵌套，则还需要计算最多中断嵌套层数（保存寄存器、中断服务程序中的局部变量等），根据不同的目标微处理器和内核的类型，任务堆栈和中断堆栈是可以分开的。中断堆栈专门用于处理中断级代码，这样做可以使得每个任务需要的堆栈空间大大减少。在有些实时内核的实现中，每个任务所需的堆栈空间大小可以分别定义，而有些内核则要求每个任务的堆栈空间大小相同。

思考题

4.1　嵌入式实时内核设计的关键问题包括哪些方面？

4.2　在嵌入式实时内核的设计实现中，可以采取哪些方法来提高其实时性？

4.3　可抢占式调度和可抢占的操作系统内核是否是同一个概念？为什么？哪一个对系统的实时响应性能影响更大？

4.4　本章在描述差分时间链时，说明了向链中新加入一个与时间有关的节点时，链表的变化情况。

① 当从差分时间链中删除一个节点时，链表会发生什么样的变化？请用图示说明几种可能的情况。

② 设计一种数据结构来表示差分时间链，并用一种高级语言编写程序来实现对差分时间链的插入和删除操作。

4.5　由于嵌入式硬件平台的多样性，嵌入式软件的移植工作主要体现在哪些方面？对实时操作系统内核而言主要包括哪些内容？

4.6　请分析 1～2 款嵌入式实时内核的系统调用，列举它们在任务管理、时间管理、内存管理、中断管理、同步和互斥、通信和消息传递等方面提供的功能。

4.7　对共享资源进行互斥保护，什么情况下最适宜用关中断的方法？什么情况下需要使用互斥信号量？请对这两种方法作一个详细的对比。

4.8　为了在一个嵌入式系统中实现有效的电源管理，硬件、操作系统及应用这三个层面分别应该起到什么样的作用？

4.9　为了尽量减少屏蔽中断对系统响应性能的影响，在操作系统内核实现和应用程序设计中可以采取哪些措施？

4.10　任务响应时间受到哪些因素的影响？在最坏的情况下，它可能包括哪些部分？

4.11　采用专门的堆栈来处理中断，有哪些好处？

第 5 章　任务管理与调度

5.1　概　述

在日常生活中，人们通常都愿意采用把一个比较大的工作划分为一些小的任务，然后再对这些任务进行处理的方法，来分散工作的复杂度和难度。这种方法在嵌入式实时系统中被称为多任务(multitasking)的处理方式，使系统能以可预见的方式对外部的并发事件进行及时处理。嵌入式实时系统采用多任务处理方式有如下好处：

- 相对于前后台软件结构而言，多任务软件结构的每个任务规模较小，每个任务更容易编码和调试，其质量也更容易得到保证。
- 不少应用本身就是由多个任务构成的，如一个应用可能需要进行以下任务的处理，即计算、从网络获取数据和刷新显示屏幕。在这种情况下，采用多任务的处理方式是应用问题的一个非常自然的解决方式。
- 任务之间具有较高的独立性，耦合性小，通过增加新的任务就能方便地扩充系统功能。
- 实时性强，保证紧急事件得到优先处理成为可能。

为此，嵌入式实时操作系统中的内核应该提供多任务的管理机制。

在嵌入式实时系统中，任务(task)通常为进程(process)和线程(thread)的统称，并把任务作为调度的基本单位进行阐述。

进程的概念最初是由 Multics 的设计者在 20 世纪 60 年代提出来的。在进程的定义中，主要包括以下内容：

- 一个正在执行的程序；
- 计算机中正在运行的程序的一个实例；
- 可以分配给处理器，并由处理器执行的一个实体；
- 由一个顺序的执行线程、一个当前状态和一组相关的系统资源所刻画的活动单元。

进程由代码、数据、堆栈和进程控制块 PCB(Process Control Block)构成。其中，进程控制块包含了操作系统用来控制进程所需要的信息，如进程状态、CPU 寄存器、调度信息、内存管理信息和 I/O 状态信息等。

在早期的进程概念中，包含了以下两个方面的内容：

- 资源　进程是资源分配的基本单位，一个进程包括一个保存进程映像的虚拟地址空间、主存、I/O 设备和文件等资源。

● 调度执行　进程作为操作系统的调度实体，是调度的基本单位。

随着操作系统的发展，进程所包含的两个方面的内容逐渐被分开，由操作系统进行独立处理。把调度执行的单位称为轻量级进程(lightweight process)或线程，把资源分配的单位称为进程。线程是进程内部一个相对独立的控制流，由线程上下文和需要执行的一段程序指令构成。

在进程中，所有线程共享该进程的状态和资源，可以访问相同的数据。如果一个线程改变了存储器中的一个数据项，则同一进程中的其他线程能够看到变化后的结果；一个线程按照读操作的方式打开一个文件后，同一进程中的其他线程也能够从该文件中进行读操作。因此，使用线程具有如下主要优势：

- 在一个已有进程中创建一个新线程比创建一个全新的进程所需的时间开销少。
- 终止一个线程比终止一个进程所花费的时间少。
- 线程切换比进程切换所花费的时间少。
- 使同一进程内部不同线程之间的通信效率得到显著提高。在大多数操作系统中，不同进程之间的通信需要内核的干预，而同一进程内部不同线程之间则可直接通信。

引入线程的概念后，可把进程和线程的使用分为以下几种模型：单进程/单线程模型(如MS－DOS)、单进程/多线程模型(如JAVA虚拟机)、多进程/单线程模型(如传统的UNIX)和多进程/多线程模型(如Windows NT，Solaris，Mach等)。这些模型的示意图如图5－1所示。在单进程/单线程模型中，整个系统只有一个进程、一个线程；在单进程/多线程模型中，整个系统有一个进程、多个线程；在多进程/单线程模型中，整个系统有多个进程，每个进程只有一个线程；在多进程/多线程模型中，系统有多个进程，每个进程又可包含多个线程。

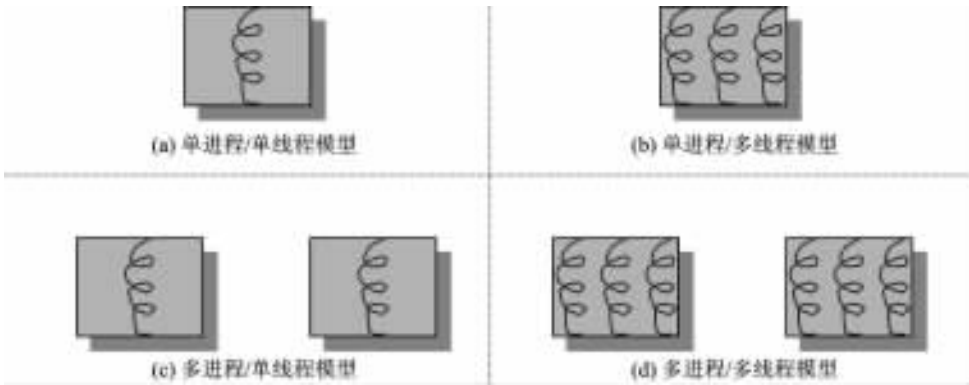


图5－1　进程和线程的使用模型

大多数实时内核都把整个应用当作一个没有定义的进程来对待，应用则被划分为多个任务的形式来进行处理，即单进程/多线程模型，或简单地称为任务模型。也有一些嵌入式实时

操作系统采用了多进程/多线程模型，即系统中包含多个进程，每个进程又对应包含多个线程。多进程/多线程模型适合于处理复杂的应用，任务模型则适用于实时性要求较高的、相对简单的应用。

任务管理(task management)是实时内核的主要工作，完成任务创建、任务删除、任务调度和改变任务优先级等工作。同通用操作系统相比，嵌入式实时操作系统的任务管理更具有危险性。通常，在实时系统中，任务被创建的时候就应该无延迟地获得所需要的内存，并且为了避免交换带来的访问延迟，只能使用主存储设备。另外，在系统运行的时候改变任务的优先级会影响整个系统的运行特性。

多任务系统中，在同一时刻通常会有多个任务处于活动状态。操作系统就需要对资源进行管理，在任务间实现资源(CPU，内存等)的共享。CPU 作为一种非常重要的资源，通过操作系统来确定如何在任务之间共享 CPU 就被称为调度(scheduling)。对于调度算法的实现，实时操作系统也与通用操作系统存在差异。实时操作系统和通用操作系统都使用相同的基本调度原理，由于性能需求上的不同导致这些算法在应用方面的差异。通用操作系统通常都期望获得最大的平均吞吐量，并防止系统出现饥饿和死锁状态。但这对嵌入式实时操作系统还远远不够。嵌入式实时操作系统则以确定性为目标，并具有内存资源占用少和低功耗的要求。

5.2　任　务

5.2.1　任务的定义及其主要特性

任务是一个具有独立功能的无限循环的程序段的一次运行活动，是实时内核调度的单位，具有动态性、并行性和异步独立性等特性。

- 动态性　任务状态是不断变化的。任务状态一般分为就绪态、运行态和等待态。在多任务系统中，任务的状态将随着系统的需要不断进行变化。
- 并行性　系统中同时存在多个任务，这些任务在宏观上是同时运行的。
- 异步独立性　每个任务各自按相互独立的不可预知的速度运行，走走停停。

5.2.2　任务的内容

任务主要包含以下内容：

- 代码，即一段可执行的程序。
- 数据，即程序所需要的相关数据(变量、工作空间、缓冲区等)。
- 堆栈。
- 程序执行的上下文环境。

任务所包含的程序通常为一个具有无限循环的程序，如图 5－2 所示。任务与程序是两个

不同的概念，它们之间的区别主要体现在以下几个方面：

- 任务能真实地描述工作内容的并发性，而程序不能。
- 程序是任务的组成部分，除程序外，任务还包括数据、堆栈及其上下文环境等内容。
- 程序是静态的，任务是动态的。
- 任务有生命周期，有诞生，有消亡，是短暂的；而程序是相对长久的。
- 一个程序可对应多个任务，反之亦然。
- 任务具有创建其他任务的功能，而程序没有。

上下文环境(context)包括了实时内核管理任务以及处理器执行任务所需要的所有信息，如任务优先级、任务的状态等实时内核所需要的信息，以及处理器的各种寄存器的内容。任务的上下文环境通过任务控制块 TCB(Task Control Block)来体现。

```
/* ioTask implements data obtaining and handling continuously */
void ioTask(void)
{
      int data;

     initial();
      /* The following sentences get data and  handle data continuously */
      while(TRUE)
      {
            data = getData();
            handleData(data);
      }
}
```

图 5-2　任务示例程序

5.2.3　任务分类

在嵌入式实时应用中，通常都包含有多个任务。比如：

- 网页浏览器　当浏览器在下载一幅图片时，同时还可以进行动画和声音的播放，而用户还可以在同时进行页面的滚动浏览；也可以在下载一个新的页面的同时，进行一个已下载页面的打印输出。
- 文字处理　文字处理器在进行后台文件打印等其他处理工作的同时，还可以处理与用户的交互内容。

图 5-3 和图 5-4 为多任务的系统模型示意图。

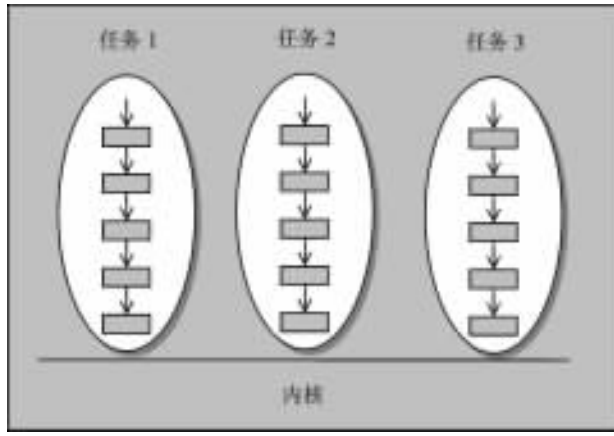


图 5-3　多任务系统示意图一

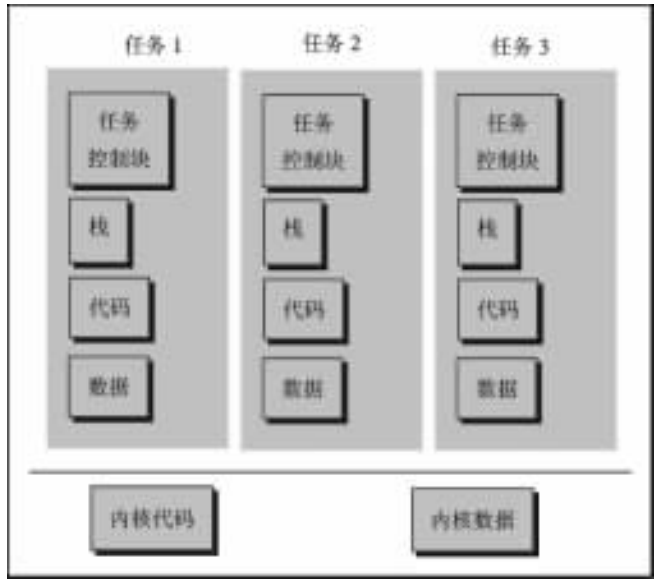


图 5-4　多任务系统示意图二

按照到达情况的可预测性，任务可以划分为周期任务(periodic task)和非周期任务两种类型；按照重要程度，又可分为关键任务(critical task)和非关键任务(noncritical task)两种类型。

实时系统中有不少任务需要重复并周期性地执行。比如飞行器，可能需要每隔 100 ms 获得一次关于飞行器的速度、高度和姿态数据，控制传感器获取这些数据就需要通过周期任务来进行。周期任务每隔一个固定的时间间隔就会执行一次。与此相反，非周期任务执行的间隔

时间则为不确定的，如移动通信设备中的通信任务。该任务只有在需要进行通信的情况下才会得到执行。非周期任务还分为有最小到达间隔时间限制的非周期任务(sporadic task)和没有到达时间限制的非周期任务(aperiodic task)两种情形。

关键任务为需要得到及时执行的任务，否则将出现灾难性的后果，如飞行器中用于处理生命支持系统和稳定性控制系统的任务。相对来说，非关键任务如果没有得到及时执行，则不会产生严重后果。

5.2.4 任务参数

任务的特性可以通过优先级(priority)、周期(period)、计算时间(computation time)、就绪时间(ready time)和截止时间(deadline)等参数来进行描述。

任务的优先级表示任务对应工作内容在处理上的优先程度。优先级越高，表明任务越需要得到优先处理。如飞行器中处理稳定性控制的任务，就需要具有较高的优先级，一旦执行条件得到满足，应及时得到执行。任务的优先级分为静态优先级和动态优先级。静态优先级表示任务的优先级被确定后，在系统运行过程中将不再发生变化；动态优先级则意味着在系统的运行过程中，任务的优先级是可以动态变化的。

周期是周期任务所具有的参数，表示任务周期性执行的间隔时间。

任务的计算时间是指任务在特定硬件环境下被完整执行所需要的时间，也被称为是任务的执行时间(execution time)。由于任务每次执行的软件环境的差异性，导致任务在各次具体执行过程中的计算时间各有不同，因此通常用最坏情况下的执行时间(worst case time)或是需要的最长执行时间来表示，也可用统计时间(statistical time)来表示。

任务的就绪时间表示任务具备了在处理器上被执行所需要的条件时的时间。

任务的截止时间意味着任务需要在该时间到来之前被执行完成。截止时间可以通过绝对截止时间(absolute deadline)和相对截止时间(relative time)两种方式来表示，相对截止时间为任务的绝对截止时间减去任务的就绪时间。对于具体的任务来说，其截止时间可以分为强截止时间(hard deadline)和弱截止时间(soft deadline)两种情况。具有强截止时间的任务即为关键任务，如果截止时间不能得到满足，就会出现严重的后果。拥有关键任务的实时系统又被称为强实时(hard real－time)系统，否则称为弱实时(soft real－time)系统。

5.3 任务管理

5.3.1 任务状态与变迁

在多任务系统中，任务要参与资源竞争，只有在所需资源都得到满足的情况下才能得到执行。因此，任务拥有的资源情况是不断变化的，导致任务状态也表现出不断变化的特性。不同

的实时内核实现方式对任务状态的定义不尽相同，但是都可以概括为以下三种基本的状态：

- 等待(waiting)　任务在等待某个事件的发生；
- 就绪(ready)　任务等待获得处理器资源；
- 执行(running)　任务获得处理器资源，所包含的代码内容正在被执行。

在单处理器系统中，任何时候都只有一个任务在 CPU 中执行；如果没有任何事情可做，就运行空闲任务执行空操作。任何一个可以执行的任务都必须处于就绪状态，实时内核的调度程序从任务的就绪队列中选择下一个需要执行的任务。处于就绪状态的任务拥有除 CPU 以外的其他所有需要的资源。除执行和就绪状态外，任务还可能处于等待状态。如任务在需要等待 I/O 设备或其他任务提供的数据，而数据又还没有到达该任务的情况下，就处于等待状态。

在一定条件下，任务会在不同的状态之间进行转换，称为任务状态的变迁(task state transition)。任务状态的变迁情况如图 5－5 所示。对于处于就绪状态的任务，获得 CPU 后，就处于运行状态。处于运行状态的任务如果被高优先级任务所抢占，任务又会回到就绪状态；处于运行状态的任务如果需要等待资源，任务会被切换到等待状态。对处于等待状态的任务，如果需要的资源得到满足，就会转换为就绪状态，等待被调度执行。

图 5－5　任务状态变迁

图 5－6 为三个任务进行状态转换的过程。图中包含三个任务和一个调度程序。调度程序用来确定下一个需要投入运行的任务，因此调度程序本身也需要占用一定的处理时间。

在时刻 0，任务 1 开始运行，处于运行态；在时刻 8，任务 1 结束运行，处于就绪态，实时内核的调度程序开始运行；在时刻 10，调度程序停止运行，任务 2 开始运行；在时刻 16，任务 2 停止运行，处于等待态，调度程序开始运行；在时刻 18，调度程序停止运行，任务 3 开始运行；在时刻 28，任务 3 停止运行，处于就绪态，调度程序开始运行；在时刻 30，任务 1 又开始运行；在时刻 40，任务 1 停止运行，处于就绪态，调度程序开始运行；在时刻 42，调度程序停止运行，任务 3 开始运行。

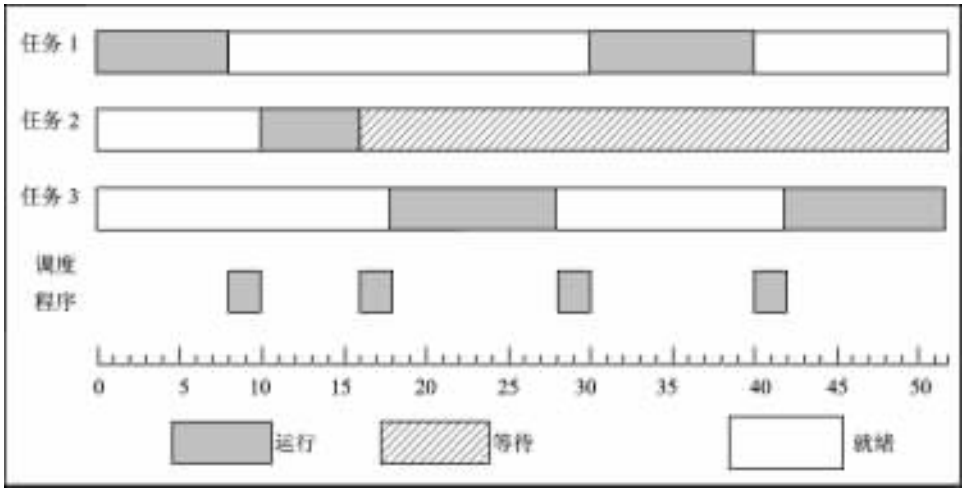


图 5-6　任务状态随时间变化的示意图

5.3.2　任务控制块

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。不同实时内核的任务控制块所包含的信息通常都不太一样，但大都包括任务的名字、任务执行的起始地址、任务的状态、任务的优先级、任务的上下文（寄存器、堆栈指针和 PC 等）和任务的队列指针等内容，如图 5-7 所示。

task name
task ID
task status
task priority
task context（registers and flags of CPU）
…

图 5-7　任务控制块示意图

为节约内存，实时内核所需支持的任务数量通常需要进行预先配置，然后在实时内核初始化的过程中，按照配置的任务数量初始化任务控制块，一个任务对应一个初始的任务控制块，形成一个空闲任务控制块链。在任务创建时，实时内核从空闲任务控制块链中为任务分配一个任务控制块。随后对任务的操作，都是基于对应的任务控制块来进行的。当任务被删除后，对应的任务控制块又会被实时内核回收到空闲任务控制块链。

任务控制块的内容可以通过实时内核提供的系统调用进行修改，也可能随着系统运行过程中内部或者外部事件的发生而发生变化。

任务的上下文为运行任务的 CPU 的上下文，通常为所有的寄存器和状态寄存器。

5.3.3　任务切换

任务切换(context switching)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。当发生任务切换时,当前正在运行的任务的上下文就需要通过该任务的任务控制块保存起来,并把需要投入运行的任务的上下文从对应的任务控制块中恢复出来。

在图 5－6 中,在时刻 8 即发生了任务切换,任务 1 的上下文需要保存到任务 1 的任务控制块中去。经过调度程序的处理,在时刻 10 任务 2 投入运行,需要把任务 2 的任务控制块中关于上下文的内容恢复到 CPU 的寄存器中。

任务切换的示意图如图 5－8 所示。图 5－8 中,任务 1 执行一段时间后,由于某种原因,需要进行任务切换,进入实时内核的调度程序。调度程序首先把当前的上下文内容保存到任务 1 的任务控制块 TCB1 中,然后又把任务 2 的上下文从 TCB2 中恢复到 CPU 寄存器中,随后任务 2 得到执行。任务 2 执行一段时间后,由于某种原因,需要进行任务切换,进入实时内核的调度程序。调度程序首先把当前的上下文内容保存到任务 2 的任务控制块 TCB2 中,然后又把任务 1 的上下文从 TCB1 中恢复到 CPU 寄存器,随后任务 1 得到执行。

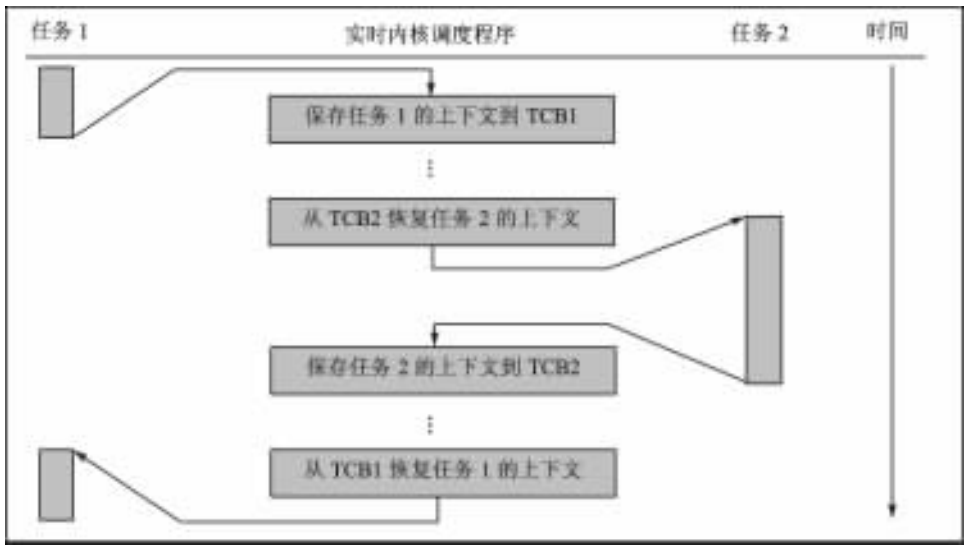


图 5－8　任务切换示意图

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是等待状态,需要投入运行的任务则由就绪状态变为运行状态。通常,任务切换具有如下基本步骤:

① 保存处理器上下文环境。

② 更新当前处于运行状态任务的任务控制块的内容,如把任务的状态由运行状态改变为就绪或是等待状态。

③ 把任务的任务控制块移到相应的队列(就绪队列或是等待队列)。

④ 选择另一个任务进行执行。实时内核通过调度程序按照一定的策略来选取需要投入运行的任务。

⑤ 改变需要投入运行的任务的任务控制块的内容,把任务的状态变为运行状态。

⑥ 根据任务控制块,恢复需要投入运行的任务的上下文环境。

任务切换可以在实时内核从当前正在运行的任务中获得控制权的任何时刻发生。导致控制权交给实时内核的事件通常包括如下内容:

- 中断、自陷。如当 I/O 中断发生的时候,如果 I/O 活动是一个或多个任务正在等待的事件,则实时内核将把相应的处于等待状态的任务转换为就绪状态;同时,实时内核还将确定是否继续执行当前处于运行状态的任务,或是用高优先级的就绪任务抢占该任务。自陷是由于执行任务中当前指令所引起的,将导致实时内核处理相应的错误或异常事件,并根据事件类型,确定是否进行任务的切换。
- 运行任务因缺乏资源而被阻塞。如当任务执行过程中进行 I/O 操作时(如打开文件),如果此前该文件已被其他任务打开,将导致当前任务处于等待状态,而不能继续执行。
- 采用时间片轮转调度时,实时内核将在时钟中断处理程序中确定当前正在运行的任务的执行时间是否已经超过了设定的时间片;如果超过了时间片,则实时内核将停止当前任务的运行,把当前任务的状态变为就绪状态,并把另一个任务投入运行。
- 一个高优先级任务处于就绪时,如果采用基于优先级的抢占式调度算法,将导致当前任务停止运行,使更高优先级的任务处于运行状态。

5.3.4 任务队列

任务队列通过任务控制块实现对系统中所有任务的管理。图 5-9 为一种比较简单的管

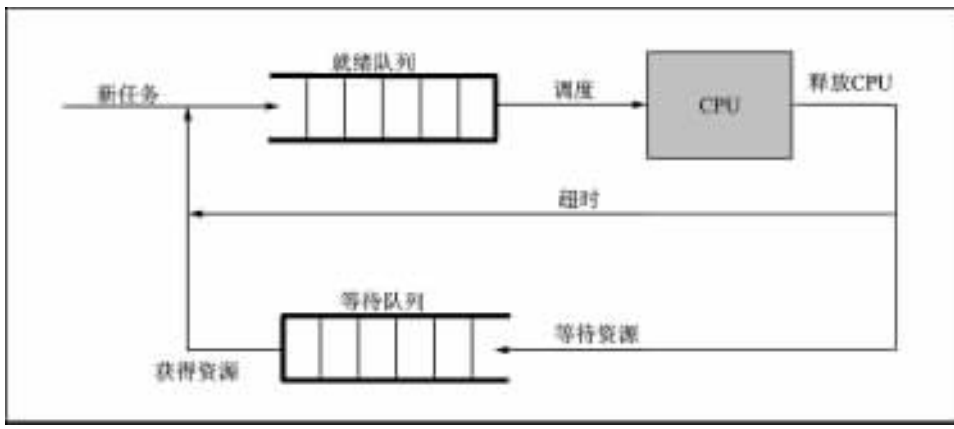
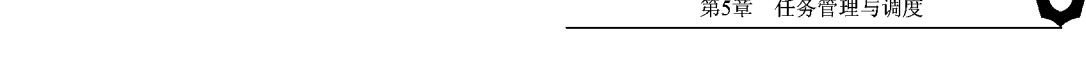


图 5-9 单就绪队列和单等待队列



理方式，把任务组织为就绪队列和等待队列两个队列。如果任务拥有除 CPU 以外的其他所需资源，则把任务放置到就绪队列；当处于运行状态的任务因为需要的资源得不到满足时，就会变为等待态，任务对应的任务控制块会被组织到等待队列中。当操作系统需要把一个新的任务投入运行时，就会按照一定的策略从任务控制块组成的就绪队列中选择一个任务投入运行。如果任务等待的资源得到满足，则任务对应的任务控制块就会从等待队列转换到就绪队列。

队列由任务控制块构成，队列的示意图如图 5－10 所示。

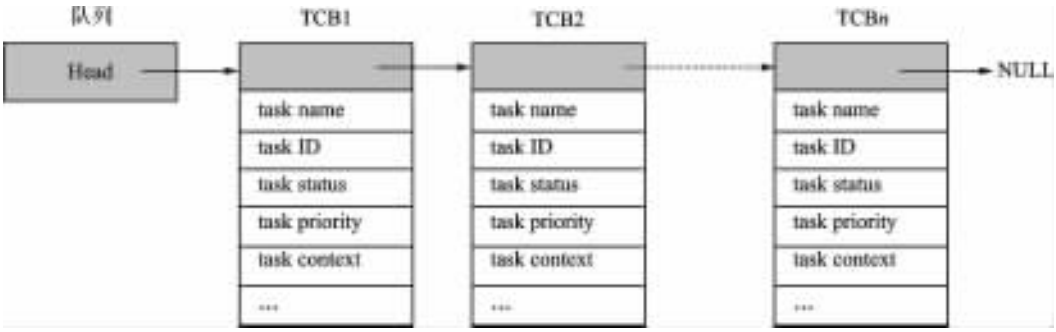


图 5－10　任务队列示意图

对于单等待队列，资源对应的事件发生时，实时内核需要扫描整个等待队列，搜索等待该资源的任务，并按照一定的策略选取任务，把任务的任务控制块放置到就绪队列。如果系统的资源和任务比较多，则采用单等待队列时，搜索等待该资源的任务所需要的时间就比较长，会影响整个系统的实时性。为此，可采用一种多等待队列的处理方式，如图 5－11 所示。在多等

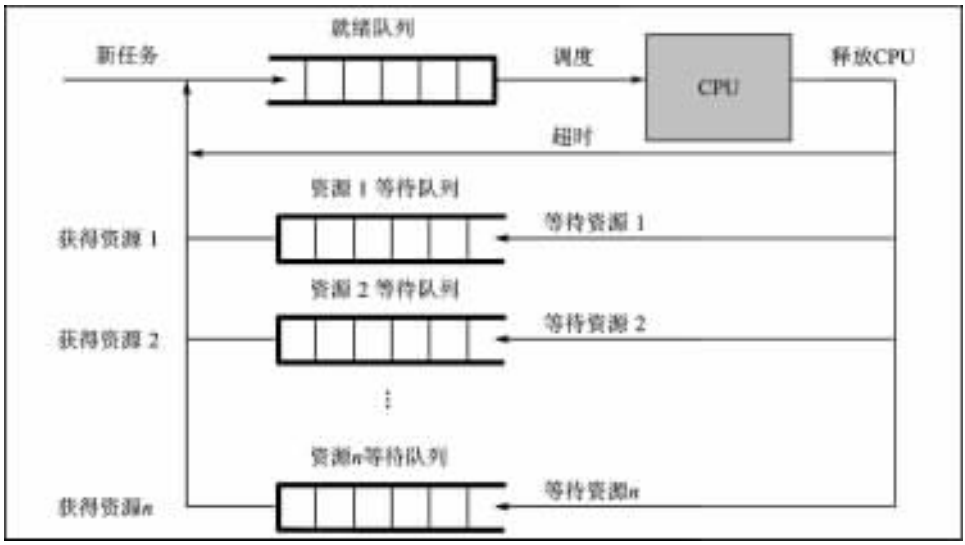


图 5－11　单就绪队列和多等待队列

待队列中，资源对应的事件发生时，能够在较短的时间内确立等待该资源的任务等待队列。

对于就绪任务，如果采用类似图 5－11 所示队列方式进行管理，则在基于优先级的调度处理中，要获得当前具有最高优先级的就绪任务，通常可以采用以下方式进行处理。

- 任务就绪时，把就绪任务的任务控制块放在就绪队列的末尾。在这种情况下，调度程序需要从就绪队列的头部到尾部进行一次遍历，才能获得就绪队列中具有最高优先级的任务。
- 就绪队列按照优先级从高到低的顺序排列。新的就绪任务到达时，需要插入到就绪队列的合适位置，确保就绪队列保持优先级从高到低排列的顺序性。

在上述两种处理方式中，所花费的时间与任务数量有密切的关系，具有不确定性。为提高实时内核的确定性，可采用一种被称为优先级位图的就绪任务处理算法。

假定每个任务的优先级都不同，下面以 64 个优先级为例来说明优先级位图算法。

(1) 设置两个变量

```
char priorityReadyGroup;
char priorityReadyTable[8];
```

priorityReadyGroup 与 priorityReadyTable 之间的关系如图 5－12 所示。priorityReadyTable 的每个数组元素对应 64(0～63：0 对应最高优先级，63 对应最低优先级)个优先级中的 8 个优先级。如 priorityReadyTable[5]对应的优先级为 40～47，如果对应优先级存在就绪任务，则相应的二进制位为 1，否则为 0。若 priorityReadyTable[5]的第 0 位为 1，则当前存在一个优先级为 40 的就绪任务；若 priorityReadyTable[5]的第 0 位为 0，则当前不存在一个优先级为 40 的就绪任务。

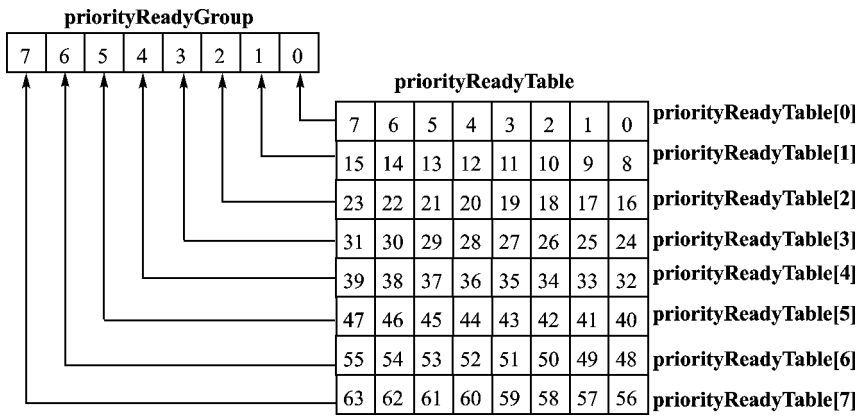


图 5－12　priorityReadyGroup 与 priorityReadyTable 之间的关系

priorityReadyGroup 中的每个二进制位与 priorityReadyTable 的一个数组元素相对应。如 priorityReadyGroup 中的第 0 位对应 priorityReadyTable[0]，priorityReadyGroup 中的第 1

位对应 priorityReadyTable[1]。priorityReadyGroup 中每个二进制位的值意味着对应 priorityReadyTable 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 priorityReadyTable 数组元素中对应的优先级有就绪任务；二进制位的值为 0，表示对应 priorityReadyTable 数组元素中对应的优先级没有就绪任务。如，若 priorityReadyGroup 的第 0 位为 1，表示 priorityReadyTable[0]中对应优先级存在就绪任务；否则，当前不存在优先级为 0～7 的就绪任务。

(2) 任务优先级与 priorityReadyGroup 和 priorityReadyTable 的关系

任务优先级与 priorityReadyGroup 和 priorityReadyTable 的关系如图 5－13 所示。当优先级数为 64 时，优先级与 6 个二进制位相对应。在表示优先级的 6 个二进制位中，高三位为优先级在 priorityReadyGroup 的二进制位位置，与 priorityReadyTable 的数组元素下标相对应；低三位表示对应 priorityReadyTable 的数组元素的值。如对于优先级 35，二进制表示为 00100011，优先级高三位为 100，在 priorityReadyGroup 中位序 4 为 1，对应 priorityReadyTable[4]；优先级低三位为 011，对应 priorityReadyTable[4]中的位序 3，即优先级 35。

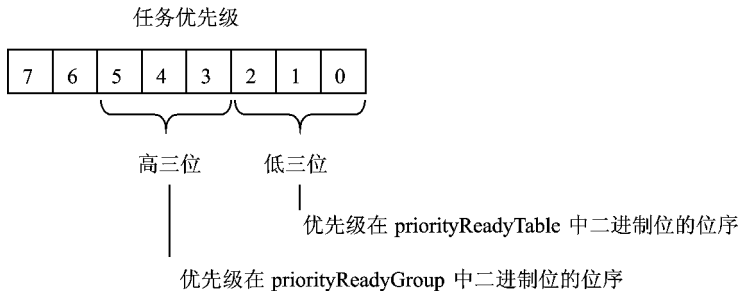


图 5－13　任务优先级与 taskReadyGroup 与 taskReadyTable 的关系

(3) 设置任务优先级到 priorityReadyGroup 的优先级映射表

```
char priorityMapTable[8];
```

优先级映射表 priorityMapTable 的内容如表 5－1 所列。priorityMapTable 的数组元素的下标与任务优先级的高三位相对应。priorityMapTable 的数组元素对应的二进制值中，位为 1 的位表示 priorityReadyGroup 的对应位也为 1。

表 5－1　优先级映射表

下　标	二进制值	下　标	二进制值
0	00000001	4	00010000
1	00000010	5	00100000
2	00000100	6	01000000
3	00001000	7	10000000

(4) **设置优先级判定表**

char priorityDecisionTable[256];

优先级判定表 priorityDecisionTable **的内容为**:

```
char priorityDecisionTable[] = {
    0,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0x00-0x0F */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0x10-0x1F */
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0x20-0x2F */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0x30-0x3F */
    6,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0x40-0x4F */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0x50-0x5F */
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0x60-0x6F */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0x70-0x7F */
    7,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0x80-0x8F */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0x90-0x9F */
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0xA0-0xAF */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0xB0-0xBF */
    6,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0xC0-0xCF */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0xD0-0xDF */
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,        /* 0xE0-0xEF */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0         /* 0xF0-0xFF */
}
```

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

(5) **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 priorityReadyGroup **和** priorityReadyTable **中的表示,转换过程可用** C **语言语句描述为**:

```
priorityReadyGroup |= priorityMapTable[priority >> 3];
priorityReadyTable[priority >> 3] |= priorityMapTable[priority & 0x07];
```

如果需要进入就绪态的任务的优先级为 50(**二进制为** 00110010),**右移三位后为** 00000110(**十进制为** 6),priorityMapTable[6]**的值为** 01000000,**则该值与** priorityReadyGroup **进行二进制"或"运算后,**priorityReadyGroup **的第** 6 **位为** 1。priority & 0x07 **用来获取优先级**

的低三位，优先级为 50 时，低三位为 010（十进制为 2），由于 priorityMapTable[2]的值为 00000100，该值与 priorityReadyTable[6]进行二进制“或”运算后，priorityReadyTable[6]的第 2 位为 1。priorityReadyTable[6]的第 2 位正好与优先级 50 相对应。

(6) 任务退出就绪态

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

```
if((priorityReadyTable[priority >> 3] &= ~prirotiyMapTable[priority & 0x07]) == 0)
    priorityReadyGroup &= ~priorityMapTable[priority >> 3];
```

首先根据任务优先级的低三位，从优先级映射表 priorityMapTable 中获取二进制位的掩码，并按位取反后与 priorityReadyTable 的对应优先级组进行按位“与”运算，把优先级在 priorityReadyTable 中的表示清除掉（相应二进制位置 0）。并且，如果按位“与”运算后的结果为 0，则表示 priorityReadyTable 的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，应把该优先级组所对应的在 priorityReadyGroup 中的二进制位清除。如，若需要把优先级为 50 的任务从就绪态退出，则需要把 priorityReadyTable[6]的第 2 位清 0；此时，如果 priorityReadyTable[6]的所有二进制位都为 0，则 priorityReadyGroup 的第 6 位也应清 0。

(7) 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：

```
high3Bit = priorityDecisionTable[priorityReadyGroup];
low3Bit = priorityDecisionTable[priorityReadyTable[high3Bit]];
priority = (high3Bit << 3) + low3Bit;
```

high3Bit 为就绪任务最高优先级的高三位，low3Bit 为就绪任务最高优先级的低三位，priority 为就绪任务的最高优先级。如果当前 priorityReadyGroup 的二进制值为 00010010（十六进制为 0x12），以 0x12 为下标，从优先级判定表 priorityDecisionTable 中获得就绪任务最高优先级的高三位为 1（二进制为 001），假定当前 priorityReadyTable[1]的二进制值为 00001010（十六进制为 0x0A），则从优先级判定表 priorityDecisionTable 中获得就绪任务最高优先级的低三位为 1（二进制为 001）。据此，得出当前就绪任务的最高优先级为 9。

(8) 根据优先级获取相应的任务

假定任务按优先级进行组织，以优先级为下标，即可得到相应任务的任务控制块。

5.3.5　任务管理机制

任务管理用来实现对任务状态的直接控制和访问。实时内核的任务管理是通过系统调用

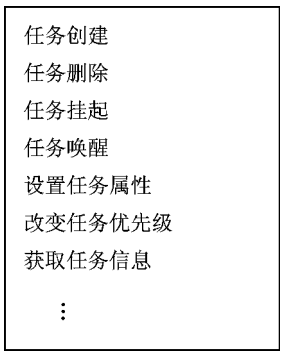


图 5－14　任务管理功能

来体现的，这些系统调用主要包括任务创建、任务删除、任务挂起、任务唤醒和设置任务属性等内容，如图 5－14 所示。

创建任务的过程即为分配任务控制块的过程。在创建任务时，通常需要确定任务的名字和任务的优先级等内容，并为任务确立所能使用的堆栈区域。任务创建成功后，通常会为用户返回一个标识该任务的 ID，以实现对任务的引用管理；任务删除是把任务从系统中去掉，释放对应的任务控制块；任务挂起是把任务变为等待状态，可通过任务唤醒操作把任务转换为就绪状态；设置任务属性可以用来设置任务的抢占、时间片等特性，以确定是否允许任务在执行过程中被抢占或是对同优先级任务采用时间片轮转方式运行等；改变任务优先级用来根据需要改变任务的当前优先级；获取任务信息是获得任务的当前优先级、任务的属性、任务的名字、任务的上下文和任务的状态等内容，便于用户进行决策。

1. 任务创建

任务创建为任务分配和初始化相关的数据结构。任务创建时通常需要使用如下信息：

- 任务的名字；
- 任务的初始优先级；
- 任务堆栈；
- 任务属性；
- 任务对应的函数入口地址；
- 任务对应函数的参数；
- 任务删除时的回调函数。

任务名字由实时内核的用户在创建任务时指定，也可以由系统自动分配。如果用户在创建任务时为任务指定了名字，则实时内核通常应将名字字符串进行复制保存，便于用户在创建任务后可以释放名字字符串的空间。

由于不同任务运行时需要的堆栈空间的大小不同，由实时内核进行任务堆栈的分配就不能适应应用任务的多样性需求。因此，通常由用户指定任务运行过程中需要使用的堆栈空间。确定任务到底需要多少堆栈空间是一件比较困难的事情。为了避免堆栈溢出所导致的任务栈被破坏的情况，同时又能充分利用嵌入式系统的有限资源，大都需要进行一个反复修正的过程：

① 在最开始的时候，根据应用的类型，为任务分配一个比预期估计更大的堆栈空间；

② 使用堆栈检测函数（通常由实时内核提供），定期监控任务对堆栈的使用情况，并据此对任务堆栈的大小进行调整。

任务可以包含多种属性，通常包括任务是否可被抢占、是否采用时间片轮转调度方式调

度、是否响应异步信号、任务中开放的中断级别和是否使用数字协处理器等内容。如果任务需要进行浮点运算，则在创建任务时实时内核应为任务分配浮点堆栈空间，以在任务切换时保存或是恢复数字协处理器的上下文内容。如果任务具有使用数字协处理器的属性，则任务可以进行以下操作：

- 执行浮点操作；
- 调用具有浮点类型返回值的函数；
- 调用使用了浮点类型数据作为参数的函数。

任务对应函数的入口地址用来表示所创建任务起始执行的入口。

通过在任务创建时提供的任务删除时的回调函数，任务可以在该任务被删除时对其所用资源进行回收或是删除。该资源是实时内核不可知的、应用特定的资源。

实时内核创建任务成功后，通常返回任务的标识（ID），实时内核的用户可以通过创建任务时获得的 ID 进行任务相关的其他操作。

任务创建通常需要完成以下工作：

① 获得任务控制块 TCB；

② 根据实时内核用户提供的信息初始化 TCB；

③ 为任务分配一个可以惟一标识任务的 ID；

④ 使任务处于就绪状态，把任务放置到就绪队列；

⑤ 进行任务调度处理。

2. 任务删除

实时内核根据任务创建时获得的 ID 删除指定的任务。

由于任务使用了各种各样的资源，因此，在删除一个任务时，需要释放该任务所拥有的资源。释放任务所拥有的资源通常由实时内核和任务共同完成。实时内核通常只释放那些由实时内核为任务分配的资源，如任务名字和 TCB 等内容所占用的空间。对于那些由任务自己分配的资源则通常由任务自身进行释放，如任务的堆栈空间以及其他一些任务申请的资源（如信号量、timer、文件系统资源、I/O 设备和使用 malloc 等函数动态获得的内存空间等）。

可以使用任务删除回调函数来进行任务删除时资源释放的处理。

任务删除通常需要进行以下工作：

① 根据指定的 ID，获得对应任务的 TCB；

② 把任务的 TCB 从队列中取出来，挂入空闲 TCB 队列；

③ 释放任务所占用的资源。

3. 任务挂起

任务挂起根据任务的 ID，把指定任务挂起，直到通过唤醒任务对任务进行解挂。

通过任务挂起，一个任务可以把自己挂起。当任务把自己挂起后，会引起任务的调度，实时内核将选取另外一个合适的任务进行执行。

任务被挂起后，该任务将处于等待状态。

任务挂起通常需要进行以下工作：

① 根据指定的 ID，获得对应任务的 TCB；

② 把任务的状态变为等待状态，并把 TCB 放置到等待队列；

③ 如果任务自己挂起自己，则进行任务调度。

4. 任务唤醒

任务唤醒根据任务 ID 解挂指定的任务。如果任务还在等待其他资源，则任务解挂后仍然处于等待状态；否则，解挂后的任务将处于就绪状态。

解挂任务通常需要进行以下工作：

① 根据指定的 ID，获得对应任务的 TCB。

② 如果任务在等待其他资源，任务将仍然处于等待状态；否则，把任务的状态变为就绪状态，并把 TCB 放置到就绪队列。

③ 进行任务调度。

5. 任务睡眠

任务睡眠使当前任务睡眠一段指定的时间，时间到后，任务又重新回到就绪状态。

任务睡眠通常需要进行以下工作：

① 修改任务状态，把任务状态变为等待状态；

② 把任务 TCB 放置到时间等待链；

③ 进行任务调度。

6. 关于任务扩展

实时内核通常还提供任务扩展的功能，以便于应用能够向系统中添加一些关于任务的附加操作，为应用提供在系统运行的关键点上进行干预的手段。通过任务扩展，可把应用提供的函数挂接到系统中去，使得在创建任务、任务上下文发生切换或是任务被删除的时候，这些被挂接的函数能够得到执行。

任务扩展的时机通常包含以下情况：

- 任务创建时；
- 任务删除时；
- 任务上下文切换时。

任务扩展功能可通过任务扩展表或是单独应用编程接口的方式来实现。任务扩展表的实现方式如图 5－15 所示，单独应用编程接口方式如图 5－16 所示。

在任务扩展表处理方式中，任务扩展表用来存放实现任务扩展处理的例程，实时内核通过查找任务扩展表来获取扩展处理的入口函数。通过创建任务扩展表，把任务扩展例程添加到系统中去，通过删除任务扩展表则可把任务扩展例程删除掉。在图 5－16 所示的处理方式中，为任务创建、任务删除和任务上下文切换分别提供了添加和删除任务扩展处理例程。

```
typedef void (*extensionRoutine)(void);
typedef struct
{
        extensionRoutine extensionOfTaskCreate;
        extensionRoutine extensionOfTaskDelete;
        extensionRoutine extensionOfTaskSwitch;
}taskExtensionTable;

statusCode extensionCreate(taskExtensionTable *extensionTable ， int *ID);
statusCode extensionDelete(int extensionTableID);
```

图 5－15　任务扩展表处理示意图

API	描　　述
extensionRoutineOfTaskCreateAdd();	/* 为任务创建时提供扩展处理例程 */
extensionRoutineOfTaskCreateDelete();	/* 删除为任务创建提供的扩展处理例程 */
extensionRoutineOfTaskDeleteAdd();	/* 为任务删除时提供扩展处理例程 */
extensionRoutineOfTaskDeleteDelete();	/* 删除为任务删除提供的扩展处理例程 */
extensionRoutineOfTaskSwitchAdd();	/* 为任务上下文切换时提供扩展处理例程 */
extensionRoutineOfTaskSwitchDelete();	/* 删除为任务上下文切换提供的扩展处理例程 */

图 5－16　通过单独的 API 实现任务扩展

7. 任务变量

某些例程可能同时被多个任务调用，每个调用任务又期望例程中的全局或是静态变量提供不同的值，比如多个任务都希望使用一个私有的内存区域，而该内存区域又是通过全局变量的方式来提供的情况。为此，实时内核可采用任务变量的处理方式。任务变量位于任务的上下文，属于任务 TCB 的内容。这样，在发生任务切换时，任务变量对应的全局或是静态变量的内容也需要进行切换。发生任务切换时，如果当前任务拥有任务变量，则需要把任务变量对应全局或是把静态变量的内容保存到该任务的任务变量之中。如果即将投入运行的任务使用了任务变量，则需要把任务变量的内容恢复到对应的全局或是静态变量中去。

通过任务变量，多个任务可以把同一个全局或是静态变量作为任务私有的变量来使用。其他任务对该变量的修改不会影响这个变量在调用任务中的值，即该变量虽然是一个全局或是静态变量，但是在单个任务中可以像私有变量一样使用。

图 5－17 为任务切换时所发生的任务变量切换示意图。TCB1 为当前正在运行的任务的

任务控制块，TCB2 为需要执行的任务的任务控制块。pTaskVar 用来表示任务的任务变量。

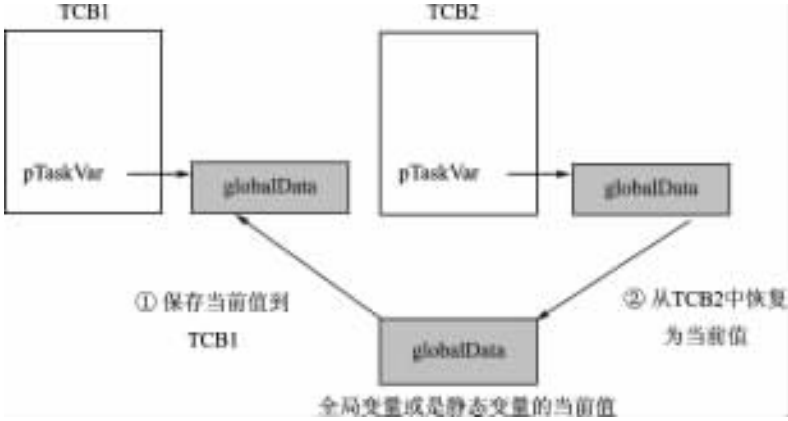


图 5-17　任务变量的切换

实时内核通常提供以下关于任务变量的操作：

- 向指定的任务中添加任务变量；
- 删除指定任务的任务变量；
- 获得指定任务的任务变量；
- 获得指定任务当前拥有的任务变量的数量；
- 获得指定任务的所有任务变量。

5.4　任务调度

5.4.1　概　述

在计算机发展的初期，需要使用计算机时，通常都要集中在计算机所在的地方，人为地以作业(job)的方式把工作内容一件一件地提交给计算机进行处理，也就不存在调度的概念。随后，出现了计算机的批处理(batch processing)方式，计算机把作业按照先来先服务的方式进行处理，体现了一种非常简单的调度概念。只有在后来出现的多道程序处理(multiprogramming)方式下，调度才变得复杂和重要起来。

在多任务的实时操作系统中，调度是一个非常重要的功能，用来确定多任务环境下任务执行的顺序和在获得 CPU 资源后能够执行的时间长度。

操作系统通过一个调度程序(scheduler)来实现调度功能。调度程序以函数的形式存在，用来实现操作系统的调度算法。调度程序本身并不是一个任务，而是一个函数调用，可在内核的各个部分进行调用。调用调度程序的具体位置又被称为是一个调度点(scheduling point)。

调度点通常处于以下位置：

- 中断服务程序的结束位置；
- 任务因等待资源而处于等待状态；
- 任务处于就绪状态时等。

调度本身需要一定的系统开销(overhead)，需要花费时间来计算下一个可被执行的任务。因此，竭力使用最优调度方案往往并不是一个明智的办法。高级的调度程序通常具有不可预见性，需要花费更多的时间和资源，并且，其复杂性也增加了应用编程人员的使用难度。简单是实时内核所强调的主要特点，实用的实时内核在实现时大都采用了简单的调度算法，以确保任务的实时约束特性和可预见性是可以管理的。复杂的、高级的调度算法则通常用于研究领域。

实时内核的主要职责就是要确保所有的任务都能够满足任务的时间约束特性要求。时间约束特性来源于任务的不同需求(如截止时间、QoS 等)，且同一个任务在不同时候也可能具有不同的时间约束特性。比如，机器人中用来控制行动的任务在障碍环境下行走所需要考虑的约束特性，就比行走在开放环境下要多得多。因此，能够同时适应所有情况的调度算法是不存在的。尽管现在已经提出和采用了很多调度算法，但仍然需要继续进行调度算法的研究，以更好地满足多样性需求的需要。从理论上来说，最优调度只有在能够完全获知所有任务在处理、同步和通信方面的需求，以及硬件的处理和时间特性的基础上才能实现。但在实际的应用中，则很难实现这一点，特别是当这些需要获知的信息处于动态变化的情况下。即使在这些需要的信息都是可以预见的情况下，常用的调度问题仍然是一个 NP(Nondeterministic Polynomial time)难题，调度的复杂性将随调度需要考虑的任务和约束特性的数量呈现出指数增长。因此，调度算法不能很好地适应系统负载和硬件资源不断增长的系统。当然，这并不意味着调度算法不能解决只有少量、定义好的任务应用需求。

调度程序是影响系统性能(如吞吐率、延迟时间等)的重要部分。在设计调度程序时，通常需要综合考虑如下因素：

- CPU 的使用率(CPU utilization)；
- 输入/输出设备的吞吐率；
- 响应时间(responsive time)；
- 公平性；
- 截止时间。

这些因素之间具有一定的冲突性。比如可通过让更多的任务处于就绪状态来提高 CPU 的使用率，但这显然会降低系统的响应时间。因此，调度程序的设计需要优先考虑最重要的需求，然后在各种因素之间进行折衷处理。

可以把一个调度算法(scheduling algorithms)描述为是在一个特定时刻用来确定将要运行的任务的一组规则。从 1973 年 Liu 和 Layland 开始关于实时调度算法的研究工作以来(1973 年，Liu 和 Layland 发表了一篇名为“Scheduling Algorithms for Multiprogramming in a

Hard Real-Time Environment"的论文),相继出现了很多调度算法和方法。对于调度方法,可以划分为以下三个主要的行为:

- 脱机配置;
- 运行时调度;
- 先验性分析。

脱机配置产生运行时调度所需要的静态信息;运行时调度在系统运行的时候根据不同的事件在各个计算之间进行切换处理;先验性分析根据静态配置信息和调度算法在运行时的行为,分析确定所有的时间需求是否得到满足。各种调度算法之间的差异在于它们在上述三种行为之间的侧重点。先验性分析通过测试来确定调度方法的可行性,比如所有任务在运行时的时间约束特性是否都能得到满足。

对于大量的实时调度方法而言,存在着以下几类主要的划分方法:

- 离线(off - line)和在线(on - line)调度;
- 抢占(preemptive)和非抢占(non - preemptive)调度;
- 静态(static)和动态(dynamic)调度;
- 最佳(optimal)和试探性(heuristic)调度。

根据获得调度信息的时机,调度算法分为离线调度和在线调度两类。对于离线调度算法,运行过程中使用的调度信息在系统运行之前就确定了,如时间驱动的调度(clock - driven scheduling)。离线调度算法具有确定性,但缺乏灵活性,适用于那些特性能够预先确定,且不容易发生变化的应用。在线调度算法的调度信息则在系统运行过程中动态获得,如优先级驱动的调度(如 EDF,RMS 等)。在线调度算法在形成最佳调度决策上具有较大的灵活性。

根据任务在运行过程中能否被打断的处理情况,调度算法分为抢占式调度和非抢占式调度两类。在抢占式调度算法中,正在运行的任务可能被其他任务所打断。在非抢占式调度算法中,一旦任务开始运行,该任务只有在运行完成而主动放弃 CPU 资源,或是因为等待其他资源被阻塞的情况下才会停止运行。实时内核大都采用了抢占式调度算法,使关键任务能够打断非关键任务的执行,确保关键任务的截止时间能够得到满足。相对来说,抢占式调度算法要更复杂些,且需要更多的资源,并可能在使用不当的情况下会造成低优先级任务出现长时间得不到执行的情况。非抢占式调度算法常用于那些任务需要按照预先确定的顺序执行,且只有当任务主动放弃 CPU 资源后,其他任务才能得到执行的情况。

根据任务优先级的确定时机,调度算法分为静态调度和动态调度两类。在静态调度算法中,所有任务的优先级在设计时就确定下来了,且在运行过程中不会发生变化(如 RMS)。在动态调度算法中,任务的优先级则在运行过程中确定,并可能不断地发生变化(如 EDF)。静态调度算法适用于能够完全把握系统中所有任务及其时间约束(如截止时间、运行时间、优先顺序和运行过程中的到达时间)特性的情况。静态调度比较简单,但缺乏灵活性,不利于系统扩展;动态调度有足够的灵活性来处理变化的系统情况,但需要消耗更多的系统资源。

CPU 的使用率(utilization)表示对于给定的一组任务,这些任务所使用的整个 CPU 资源的比率。对于一个给定的调度算法,其 CPU 使用率存在着一个理论上的上限,如 EDF 算法的最大 CPU 使用率为 1。

可调度性(schedulability)表示对于给定的一组任务,如果所有任务都能满足截止时间的要求,则这些任务就是可调度的。对于在线调度算法,当一个新的任务需要加入到系统中时,应进行可调度性分析;如果加入任务后会导致某些任务不能满足调度性,则该任务就不能添加到系统中去。可调度性的程度可以通过所有任务截止时间都能得到满足的情况下的最大 CPU 使用率来进行衡量。

5.4.2　基于优先级的可抢占调度

在基于优先级的可抢占调度方式中,如果出现具有更高优先级的任务处于就绪状态时,则当前任务将停止运行,把 CPU 的控制权交给具有更高优先级的任务,使更高优先级的任务得到执行。因此,实时内核需要确保 CPU 总是被具有最高优先级的就绪任务所控制。这意味着当一个具有比当前正在运行任务的优先级更高的任务处于就绪状态的时候,实时内核应及时进行任务切换,保存当前正在运行任务的上下文,切换到具有更高优先级的任务的上下文。

图 5-18 为多个任务在基于优先级的可抢占调度方式下的运行情况。任务 1 被具有更高优先级的任务 2 所抢占,然后任务 2 又被任务 3 抢占。当任务 3 完成运行后,任务 2 继续执行。当任务 2 完成运行后,任务 1 才又得以继续执行。

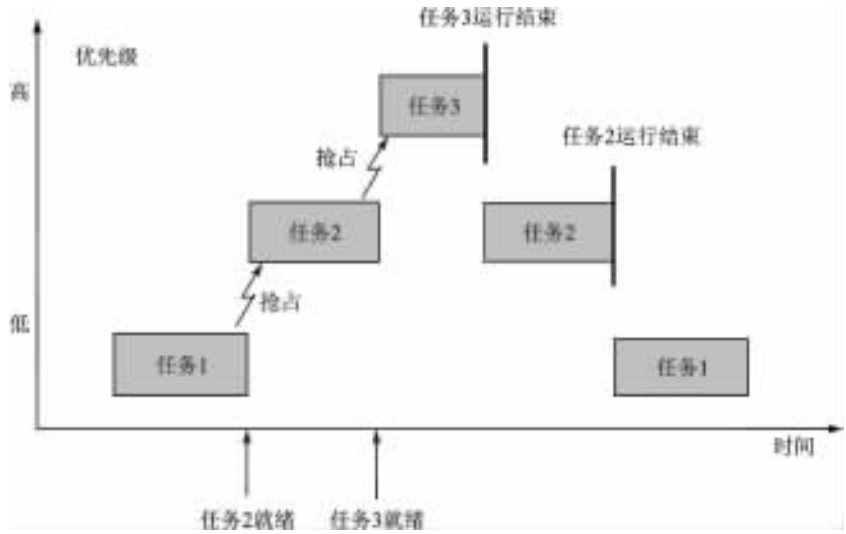


图 5-18　在可抢占调度方式下的任务运行情况

5.4.3 时间片轮转调度

时间片轮转调度(round－robin scheduling)算法是指当有两个或多个就绪任务具有相同的优先级,且它们是就绪任务中优先级最高的任务时,任务调度程序按照这组任务就绪的先后次序调度第一个任务,让第一个任务运行一段时间;然后又调度第二个任务,让第二个任务又运行一段时间;依次类推,到该组最后一个任务也得以运行一段时间后,接下来又让第一个任务运行。这里,任务运行的这段时间称为时间片(time slicing)。

在时间片轮转调度方式中,当任务运行完一个时间片后,该任务即使还没有停止运行,也必须释放处理器让下一个与它相同优先级的任务运行,使实时系统中优先级相同的任务具有平等的运行权利。释放处理器的任务被排到同优先级就绪任务链的链尾,等待再次运行。

采用时间片轮转调度算法时,任务的时间片大小要适当选择。时间片大小的选择会影响系统的性能和效率。

- 时间片太大,时间片轮转调度就没有意义;
- 时间片太小,任务切换过于频繁,处理器开销大,真正用于运行应用程序的时间将会减小。

另外,不同的实时内核在实现时间片轮转调度算法上可能有一些差异:

- 有的内核允许同优先级的各个任务有不一致的时间片;
- 有的内核要求相同优先级的任务具有一致的时间片。

图 5－19 为多个任务在时间片轮转调度方式下的运行情况示意图。任务 1 和任务 2 具有相同的优先级,按照时间片轮转的方式轮流执行。当高优先级任务 3 就绪后,正在执行的任务 2 被抢占,高优先级任务 3 得到执行。当任务 3 完成运行后,任务 2 才重新在未完成的时间片内继续执行。随后任务 1 和任务 2 又按照时间片轮转的方式执行。

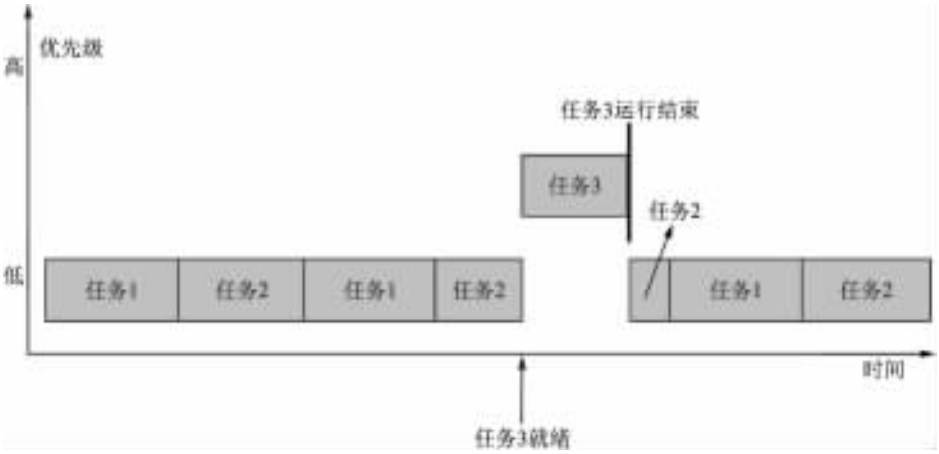


图 5－19 在时间片轮转调度方式下的任务运行情况

5.4.4　静态调度

在静态调度算法中，任务的优先级需要在系统运行前进行确定。通常来说，静态调度中确立任务优先级的主要依据有以下几点。

① 执行时间。以执行时间为依据的调度算法为：

- 最短执行时间优先(smallest execution time first)；
- 最长执行时间优先(largest execution time first)。

② 周期。以周期为依据的调度算法为：

- 短周期任务优先(smallest period first)；
- 长周期任务优先(largest period first)。

③ 任务的 CPU 使用率。任务的 CPU 使用率为任务计算时间与任务周期的比值。以任务的 CPU 使用率为依据的调度算法为：

- 最小 CPU 使用率优先(smallest task utilization first)；
- 最大 CPU 使用率优先(largest task utilization first)。

④ 紧急程度。根据任务的紧急程度，以人为的方式进行优先级的静态安排。

1. 人为安排优先级

人为安排优先级是嵌入式实时软件开发中使用得非常多的一种方法。该方法以系统分析、设计人员对系统需求的理解为基础，确定出系统中各个任务之间的相对优先情况，并据此确定出各个任务的优先级。

例 5-1　在一个用来实现网页浏览的应用中，可把整个应用划分为网络数据接收dataRecv、网页数据处理 dataHandling 和网页显示 webDisplay 等三个任务。三个任务之间的工作流程如图 5-20 所示。其中，由于 dataRecv 需要及时接收从网络传过来的数据，其优先级应最高；webDisplay 需要及时处理用户的输入(如停止获取当前页面或是请求新的页面等)，其优先级应比 dataHandling 高。据此，可人为地把 dataRecv，dataHandling 和 webDisplay 的优先级分别安排为 5，7 和 6(设定数字越大，任务的优先级越低)。

2. 比率单调调度算法

1973 年，Liu 和 Layland 在 ACM 上发表了题为“Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”的论文，该论文奠定了实时系统所有现代调度算法的理论基础。比率单调调度算法 RMS(Rate-Monotonic Scheduling algorithm)即在该论文中被提出来。RMS 是在基于以下假设的基础上进行分析的。

① 所有任务都是周期任务；

② 任务的相对截止时间等于任务的周期；

③ 任务在每个周期内的计算时间都相等，保持为一个常量；

④ 任务之间不进行通信，也不需要同步；

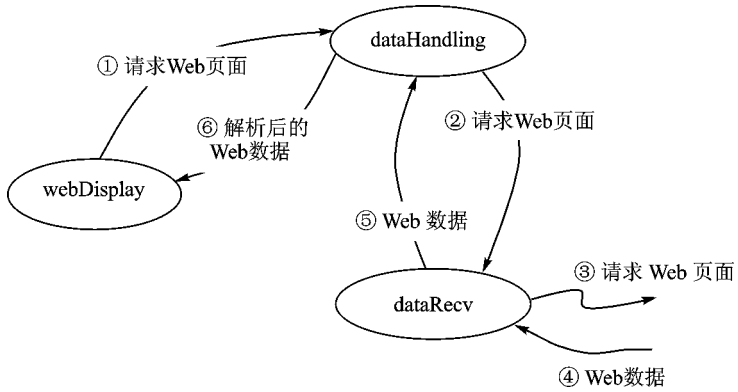


图 5－20　网页浏览应用各任务之间的关系

⑤ 任务可以在计算的任何位置被抢占，不存在临界区。

尽管 Liu 和 Layland 把该调度算法定位于单处理器的实时任务调度，但实验证明，这些结果也同样适用于分布式系统。

RMS 是一个静态的固定优先级调度算法，任务的优先级与任务的周期表现为单调函数关系，任务周期越短，任务的优先级越高；任务周期越长，任务的优先级越低。RMS 是静态调度中的最优调度算法，即如果一组任务能够被任何静态调度算法所调度，则这些任务在 RMS 下也是可调度的。

任务的可调度性可以通过计算任务的 CPU 使用率，然后把得到的 CPU 使用率同一个可调度的 CPU 使用率上限进行比较来获得。这个可调度的 CPU 使用率上限被称为可调度上限(schedulable bound)。可调度上限表示给定任务在特定调度算法下能够满足截止时间要求的最坏情况下的最大 CPU 使用率。可调度上限的最大值为 100 %，与调度算法密切相关。对于一组任务，如果任务的 CPU 使用率小于或等于可调度上限，则这组任务是可被调度的；如果任务的 CPU 使用率大于可调度上限，就不能保证这组任务是可被调度的，任务的调度性需要进一步分析。

在 RMS 中，CPU 使用率的可调度范围为

$$\sum_{i=1}^{n} \frac{C}{T} \leqslant n(2^{\frac{1}{n}} - 1)$$

上述条件是一个充分条件，但不是一个必要条件。如果任务的 CPU 使用率满足该条件，则任务是可调度的；但如果不满足该条件，也有可能被 RMS 所调度。

RMS 的可调度的 CPU 使用率上限如表 5－2 所列。

表 5－2　RMS 可调度的 CPU 使用率上限

任务数量	可调度的 CPU 使用率上限	任务数量	可调度的 CPU 使用率上限
1	1	5	0.743
2	0.828	6	0.735
3	0.780	⋮	⋮
4	0.757	∞	ln 2

例 5－2　满足条件的任务的执行情况。任务的计算时间、周期时间、CPU 使用率及其 RMS 要求的可调度上限如表 5－3 所列。任务的执行情况如图 5－21 所示。

表 5－3　任务的计算时间、周期时间、CPU 使用率及其 RMS 要求的可调度上限(例 5－2)

任　务	计算时间	周　期	CPU 使用率	CPU 使用率之和	可调度上限
1	1	5	0.20	0.20	1.00
2	5	20	0.25	0.45	0.83
3	8	50	0.16	0.61	0.78
4	12	100	0.12	0.73	0.76

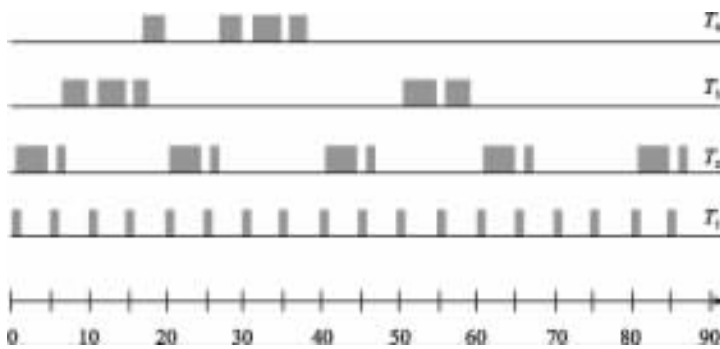


图 5－21　任务的执行情况(例 5－2)

例 5－3　不满足条件，但能够被调度的任务的执行情况。任务的计算时间、周期时间、CPU 使用率及其 RMS 要求的可调度上限如表 5－4 所列。任务的执行情况如图 5－22 所示。

表 5-4　任务的计算时间、周期时间、CPU 使用率及其 RMS 要求的可调度上限(例 5-3)

任　务	计算时间	周　期	CPU 使用率	CPU 使用率之和	可调度上限
1	1	5	0.20	0.20	1.00
2	5	20	0.25	0.45	0.83
3	10	50	0.20	0.65	0.78
4	20	100	0.20	0.85	0.76

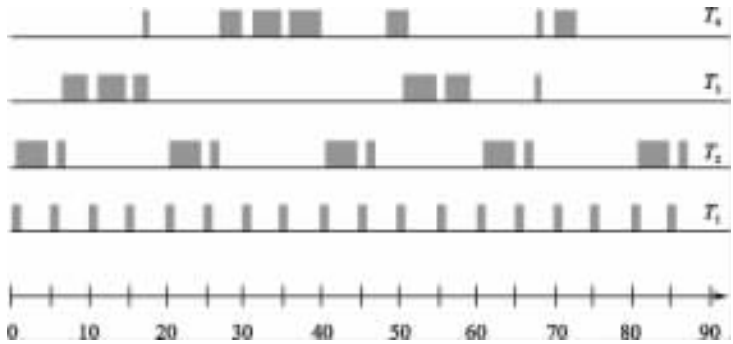


图 5-22　任务的执行情况(例 5-3)

在任务比较多的情况下,RMS 的可调度的 CPU 使用率上限为 ln 2。如此低的 CPU 使用率对大多数的系统来说都是不可接受的。对于 RMS 来说,即使 CPU 的使用率小于 100 %,但如果大于了算法给定的可调度范围,仍有可能不能满足截止时间的要求。

另外,RMS 的不足之处还在于它假定任务是相互独立的、周期性的,且任务能够在任何计算点被抢占。在实时系统中,任务之间通常都需要进行通信和同步。Rajkumar 在博士论文“Synchronization in Real-Time Systems: A Priority Inheritance Approach”中,基于 RMS 提出了可以处理任务同步的调度算法。该算法解决了单处理器上任务之间的同步问题。Rajkumar 还解决了任务之间存在互斥执行的问题,允许任务存在临界区;在执行临界区时,任务不能被其他任务所抢占。为了支持非周期任务的处理,比较简单的办法是以先用后台方式执行非周期任务,或是把就绪的非周期任务组织成一个队列,然后用周期性地查询该队列是否有非周期任务的方式来执行。后台执行非周期任务的方式效率比较低,非周期任务只有在所有周期任务都没有执行的情况下才会得到调度。周期性的查询方式相对来说性能要好一点,但有可能出现刚查询完成,就有非周期任务就绪的情况,导致该任务等待一个轮询周期才能得到调度执行的现象。

5.4.5　动态调度

对于静态调度，任务的优先级不会发生变化。在动态调度中，任务的优先级可根据需要进行改变，也可能随着时间按照一定的策略自动发生变化。

1. 截止时间优先调度算法

RMS 调度算法的 CPU 使用率比较低，在任务比较多的情况下，可调度上限为 68 %。Liu 和 Layland 又提出了一种采用动态调度的、具有更高 CPU 使用率的调度算法——截止时间优先调度算法 EDF(Earliest Deadline First)。在 EDF 中，任务的优先级根据任务的截止时间来确定。任务的绝对截止时间越近，任务的优先级越高；任务的绝对截止时间越远，任务的优先级越低。当有新的任务处于就绪状态时，任务的优先级就有可能需要进行调整。同 RMS 一样，Liu 和 Layland 对 EDF 算法的分析也是在一系列假设的基础上进行的。在 Liu 和 Layland 的分析中，EDF 不要求任务为周期任务，其他假设条件与 RMS 相同。

EDF 算法是最优的单处理器动态调度算法，其可调度上限为 100 %。在 EDF 调度算法下，对于给定的一组任务，任务可调度的充分必要条件为

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant 1$$

对于给定的一组任务，如果 EDF 不能满足其调度性要求，则没有其他调度算法能够满足这组任务的调度性要求。

同基于固定优先级的静态调度相比，采用基于动态优先级调度的 EDF 算法的显著优点在于 EDF 的可调度上限为 100 %，使 CPU 的计算能力能够被充分利用起来。EDF 也存在不足之处，在实时系统中不容易实现，并且，同 RMS 相比，EDF 具有更大的调度开销，需要在系统运行的过程中动态地计算确定任务的优先级。另外，在系统出现临时过载的情况下，EDF 算法不能确定哪个任务的截止时间会得不到满足。为此，Liu 和 Layland 提出了一种混合的调度算法，即大多数任务都采用 RMS 进行调度，只有少量任务采用 EDF 调度算法。尽管混合的调度算法不能达到 100 %的 CPU 使用率，但确实综合了 EDF 和 RMS 调度算法的优点，使整个调度比较容易实现。

例 5-4　任务及其到达时间、计算时间与绝对截止时间如表 5-5 所列。

表 5-5　任务及其到达时间、计算时间与绝对截止时间

任　务	到达时间	计算时间	绝对截止时间
T_1	0	10	30
T_2	4	3	10
T_3	5	10	25

当任务 T_1 到达的时候，由于 T_1 是系统中等待运行的惟一一个任务，因此 T_1 得到立即执行。任务 T_2 在时间 4 到达，由于任务 T_2 的截止时间(10)小于任务 T_1 的截止时间(30)，因此任务 T_2 的优先级比 T_1 高，并抢占任务 T_1 得到执行。任务 T_3 在时间 5 到达，由于 T_3 的截止时间比 T_2 的截止时间大，因此 T_3 的优先级比 T_2 低，需要等到 T_2 执行完成后，才能得到执行。当 T_2 执行完成后(时间 7)，T_3 开始执行(T_3 的优先级比 T_1 高)。T_3 运行到时间 17，T_1 才能继续执行，直到运行结束。

2. 最短空闲时间优先调度算法

在最短空闲时间优先调度算法(least - laxity - first scheduling)中，任务的优先级根据任务的空闲时间进行动态分配。任务的空闲时间越短，任务的优先级越高；任务的空闲时间越长，任务的优先级越低。任务的空闲时间可通过下式来表示，即

任务的空闲时间＝任务的绝对截止时间－当前时间－任务的剩余执行时间

5.4.6 静态调度与动态调度之间的比较

动态调度的出现是为了确保低优先级任务也能被调度。这种公平性对于所有任务都同等重要的系统比较合适，对于需要绝对可预测性的系统一般不使用动态调度。这些系统中，在出现临时过载的情况下，要求调度算法能够选择最紧急的任务执行，而放弃那些不太紧急的任务。而动态调度的优先级只反映了任务的时间特性，没有把任务的紧急程度体现到优先级中去。

动态调度的调度代价通常都比静态调度高，这主要是由于在每一个调度点都需要对任务的优先级进行重新计算，而静态调度中任务的优先级则始终保持不变，不需要进行计算。

5.5 优先级反转

5.5.1 概 述

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，就被称为优先级反转(priority inversion)。

两个任务都试图访问共享数据的情况即为出现优先级反转最通常的情况。为了保证一致性，这种访问应该是顺序进行的。如果高优先级任务首先访问共享资源，则会保持共享资源访问的合适的任务优先级顺序；但如果是低优先级任务首先获得共享资源的访问，然后高优先级

任务请求获取对共享资源的访问，则高优先级任务将被阻塞，直到低优先级任务完成对共享资源的访问。

阻塞是优先级反转的一种形式，它使得高优先级任务必须等待低优先级任务的处理。如果阻塞的时间过长，即使在 CPU 使用率比较低的情况下，也可能出现截止时间得不到满足的情况。为了在系统中维持一个比较高的可调度性，需要通过一定的协议来使发生任务阻塞的情况降到比较低的程度。

通常的同步互斥机制为信号量(semaphore)、锁(lock)和 Ada 中的 Rendezvous(汇合)等。为保护共享资源的一致性，或是确保非抢占资源在使用上的合适顺序，使用这些方法是非常必需的，但应确保使用这些方法后系统的时间需求能够得到满足。事实上，直接应用这些同步互斥机制将导致系统中出现不定时间长度的优先级反转和比较低的任务可调度性情况。

例 5-5　假设 T_1，T_2，T_3 为优先级顺序降低的三个任务，T_1 具有最高优先级。假定 T_1 和 T_3 通过信号量 S 共享一个数据结构，并且在时刻 t_1，任务 T_3 获得信号量 S，开始执行临界区代码。在 T_3 执行临界区代码的过程中，高优先级任务 T_1 就绪，抢占任务 T_3 而获得 CPU 资源，并在随后试图使用共享数据，但该共享数据已被 T_1 通过信号量 S 加锁。在这种情况下，会期望具有最高优先级的任务 T_1 被阻塞的时间不超过任务 T_3 执行完整个临界区的时间。但事实上，这种阻塞时间的长度是无法预知的。这主要是由于任务 T_3 还可能被具有中等优先级的任务 T_2 所阻塞，使得 T_1 也需要等待 T_2 和其他中等优先级的任务释放 CPU 资源。

任务 T_1 的阻塞时间长度不定，可能会很长。如果任务在临界区内不允许被抢占，则这种情况可得到部分解决。但由于形成了不必要的阻塞，使得这种方案只适合于非常短的临界区。比如，一旦一个低优先级任务进入了一个比较长的临界区，不会访问该临界区的高优先级任务将会被完全不必要地阻塞。

关于优先级反转的问题，Lampson 在 1980 年发表的题为“Experiences with Processes and Monitors in Mesa”的论文中被首先讨论，建议临界区执行在比可能使用该临界区的所有任务的优先级更高的优先级上。

解决优先级反转现象的常用协议为：

- 优先级继承协议(priority inheritance protocol)；
- 优先级天花板协议(priority ceiling protocol)。

5.5.2　优先级继承协议

优先级继承协议的基本思想是：当一个任务阻塞了一个或多个高优先级任务时，该任务将不使用其原来的优先级，而使用被该任务所阻塞的所有任务的最高优先级作为其执行临界区的优先级。当该任务退出临界区时，又恢复到其最初的优先级。

例 5-6　考虑例 5-5 中的情况。如果任务 T_1 被 T_3 阻塞，优先级继承协议要求任务 T_3 以任务 T_1 的优先级执行临界区。这样，任务 T_3 在执行临界区的时候，原来比 T_3 具有更高优

先级的任务 T_2 就不能抢占 T_3 了。当任务 T_3 退出临界区时，又恢复到其原来的低优先级，使任务 T_1 又成为最高优先级的任务。这样，任务 T_1 会抢占任务 T_3 而继续获得 CPU 资源，而不会出现例 5－5 中 T_1 会无限期被任务 T_2 所阻塞的情况。

优先级继承协议的定义如下：

① 如果任务 T 为具有最高优先级的就绪任务，则任务 T 将获得 CPU 资源。在任务 T 进入临界区前，任务 T 需要首先请求获得该临界区的信号量 S。

- 如果信号量 S 已经被加锁，则任务 T 的请求会被拒绝。在这种情况下，任务 T 被称为是被拥有信号量 S 的任务所阻塞；
- 如果信号量 S 未被加锁，则任务将获得信号量 S 而进入临界区。当任务 T 退出临界区时，使用临界区过程中所加锁的信号量将被解锁。此时，如果有其他任务因为请求信号量 S 而被阻塞，则其中具有最高优先级的任务将被激活，处于就绪状态。

② 任务 T 将保持其被分配的原有优先级不变，除非任务 T 进入了临界区并阻塞了更高优先级的任务。如果由于任务 T 进入临界区而阻塞了更高优先级的任务，则任务 T 将继承被任务 T 阻塞的所有任务的最高优先级，直到任务 T 退出临界区。当任务 T 退出临界区时，任务 T 将恢复到进入临界区前的原有优先级。

③ 优先级继承具有传递性。比如，假设 T_1，T_2，T_3 为优先级顺序降低的三个任务，如果任务 T_3 阻塞了任务 T_2，此前任务 T_2 又阻塞了任务 T_1，则任务 T_3 将通过任务 T_2 继承任务 T_1 的优先级。

在优先级继承协议中，高优先级任务在两种情况下可能被低优先级任务所阻塞。

① 直接阻塞　如果高优先级任务试图获得一个已经被加锁的信号量，则该任务将被阻塞，这种阻塞即为直接阻塞。直接阻塞用来确保临界资源使用的一致性能够得到满足。

② 间接阻塞　由于低优先级任务继承了高优先级任务的优先级，使得中等优先级的任务被原来分配的低优先级任务阻塞，这种阻塞即为间接阻塞。这种阻塞也是必需的，用来避免高优先级任务被中等优先级任务间接抢占。

优先级继承协议具有如下特性：

- 只有在高优先级任务与低优先级任务共享临界资源，且低优先级任务已经进入临界区后，高优先级任务才可能被低优先级任务所阻塞；
- 高优先级任务被低优先级任务阻塞的最长时间为高优先级任务中可能被所有低优先级任务阻塞的具有最长执行时间的临界区的执行时间；
- 如果有 m 个信号量可能阻塞任务 T，则任务 T 最多被阻塞 m 次。

根据上述特性可知，对于一个任务来说，采用优先级继承协议，系统运行前就能够确定该任务的最大阻塞时间。但优先级继承协议存在以下两个方面的问题：

① 优先级继承协议本身不能避免死锁的发生；

② 在优先级继承协议中，任务的阻塞时间虽然是有界的，但由于可能出现阻塞链，使得任

务的阻塞时间可能会很长。

例 5-7　假定在时间 t_1，任务 T_2 获得信号量 S_2，进入临界区。在时间 t_2，任务 T_2 又试图获得信号量 S_1，但一个高优先级任务 T_1 在这个时候就绪，抢占任务 T_2 并获得信号量 S_1；接下来任务 T_1 又试图获得信号量 S_2。这样就出现了死锁现象。

这种死锁可以通过规定按顺序访问信号量的方式得到解决。

例 5-8　假定任务 T_1 需要顺序获得信号量 S_1 和 S_2，任务 T_3 在 S_1 控制的临界区中被 T_2 抢占，然后 T_2 进入 S_2 控制的临界区。这个时候，任务 T_1 被激活而获得 CPU 资源，发现信号量 S_1 和 S_2 都分别被低优先级任务 T_2 和 T_3 加锁，T_1 将被阻塞两个临界区，需要先等待任务 T_3 释放信号量 S_1，然后等待任务 T_2 释放信号量 S_2，这样就形成了关于任务 T_1 的阻塞链。

5.5.3　优先级天花板协议

使用优先级天花板协议的目的在于解决优先级继承协议中存在的死锁和阻塞链问题。

优先级天花板指控制访问临界资源的信号量的优先级天花板。信号量的优先级天花板为所有使用该信号量的任务的最高优先级。在优先级天花板协议中，如果任务获得信号量，则在任务执行临界区的过程中，任务的优先级将被抬升到所获得信号量的优先级天花板。

在优先级天花板协议中，主要包含如下处理内容：

① 对于控制临界区的信号量，设置信号量的优先级天花板为可能申请该信号量的所有任务中具有最高优先级任务的优先级。

② 如果任务成功获得信号量，任务的优先级将被抬升为信号量的优先级天花板；任务执行完临界区，释放信号量后，则其优先级恢复到最初的优先级。

③ 如果任务不能获得所申请的信号量，则任务将被阻塞。

例 5-9　假设系统中存在 T_1，T_2，T_3 三个优先级顺序降低的任务（优先级分别为 p_1，p_2，p_3）。假定 T_1 和 T_3 通过信号量 S 共享一个临界资源。根据优先级天花板协议，信号量 S 的优先级天花板为 p_1。假定在时刻 t_1，T_3 获得信号量 S，按照优先级天花板协议，T_3 的优先级将被抬升为信号量 S 的优先级天花板 p_1，直到 T_3 退出临界区。这样，T_3 在执行临界区的过程中，T_1 和 T_2 都不能抢占 T_3，确保 T_3 能尽快完成临界区的执行，并释放信号量 S，退出临界区。当 T_3 退出临界区后，T_3 的优先级又回落为 p_3。此时，如果在 T_3 执行临界区的过程中，任务 T_1 或 T_2 已经就绪，则 T_1 或 T_2 将抢占 T_3 的执行。

优先级继承协议和优先级天花板协议都能解决优先级反转问题，但在处理效率和对程序运行流程的影响程度上有所不同。

1. 关于执行效率的比较

优先级继承协议可能多次改变占有某临界资源的任务的优先级，而优先级天花板协议只需改变一次。从这个角度看，优先级天花板协议的效率高，因为若干次改变占有资源的任务的优先级会引入更多的额外开销，导致任务执行临界区的时间增加。例 5-10 对该情况进行了

说明。

例 5-10 假设系统中有七个任务，按优先级从高到低分别为 $T_1, T_2, T_3, T_4, T_5, T_6, T_7$，使用由信号量 S 控制的临界资源。图 5-23 为采用优先级继承协议的任务执行情况，图5-24为采用优先级天花板协议的任务执行情况。图中，对于一个任务来说，单横线处于低端位置时，表示对应任务被阻塞，或是被高优先级任务所抢占。单横线处于高端位置时，表示任务正在执行。如果没有单横线，则表示任务还未被初始化，或是任务已经执行完成。阴影部分表示任务正在执行临界区。

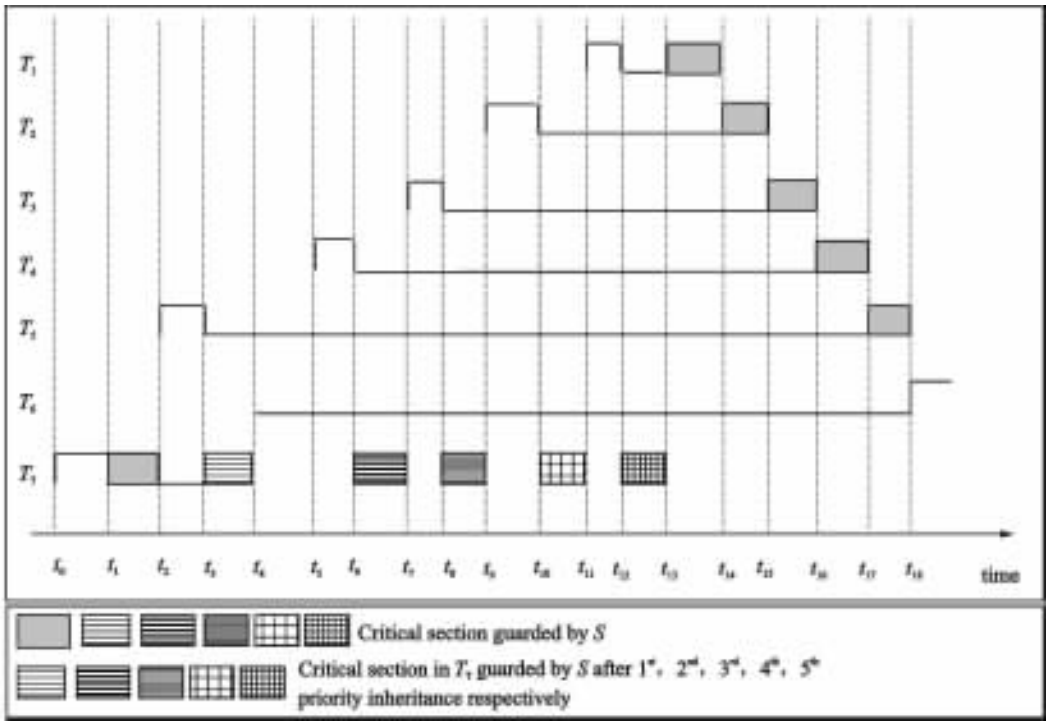


图 5-23 基于优先级继承协议的任务执行情况

在图 5-23 中，任务的执行情况如下：

① t_0 时刻，任务 T_7 开始运行，并于时刻 t_1 获得信号量 S，进入临界区；

② t_2 时刻，任务 T_5 就绪，任务 T_5 抢占任务 T_7 运行；

③ t_3 时刻，任务 T_5 申请信号量 S 失败，任务 T_7 继承任务 T_5 的优先级继续运行；

④ t_4 时刻，任务 T_6 就绪，但因其优先级低于任务 T_7 而无法运行；

⑤ t_5 时刻，任务 T_4 就绪，任务 T_4 抢占任务 T_7 运行；

⑥ t_6 时刻，任务 T_4 申请信号量 S 失败，任务 T_7 继承任务 T_4 的优先级继续运行；

⑦ t_7 时刻，任务 T_3 就绪，任务 T_3 抢占任务 T_7 运行；

⑧ t_8 时刻，任务 T_3 申请信号量 S 失败，任务 T_7 继承任务 T_3 的优先级继续运行；

⑨ t_9 时刻，任务 T_2 就绪，任务 T_2 抢占任务 T_7 运行；

⑩ t_{10} 时刻，任务 T_2 申请信号量 S 失败，任务 T_7 继承任务 T_2 的优先级继续运行；

⑪ t_{11} 时刻，任务 T_1 就绪，任务 T_1 抢占任务 T_7 运行；

⑫ t_{12} 时刻，任务 T_1 申请信号量 S 失败，任务 T_7 继承任务 T_1 的优先级继续运行；

⑬ t_{13} 时刻，任务 T_7 释放信号量 S，退出临界区，优先级降低为初始的优先级，任务 T_1 获得信号量 S 并运行；

⑭ t_{14} 时刻，任务 T_1 释放信号量 S 并运行完毕，任务 T_2 获得信号量 S 并运行；

⑮ t_{15} 时刻，任务 T_2 释放信号量 S 并运行完毕，任务 T_3 获得信号量 S 并运行；

⑯ t_{16} 时刻，任务 T_3 释放资源 S 并运行完毕，任务 T_4 获得信号量 S 并运行；

⑰ t_{17} 时刻，任务 T_4 释放资源 S 并运行完毕，任务 T_5 获得信号量 S 并运行；

⑱ t_{18} 时刻，任务 T_5 释放资源 S 并运行完毕，任务 T_6 运行。

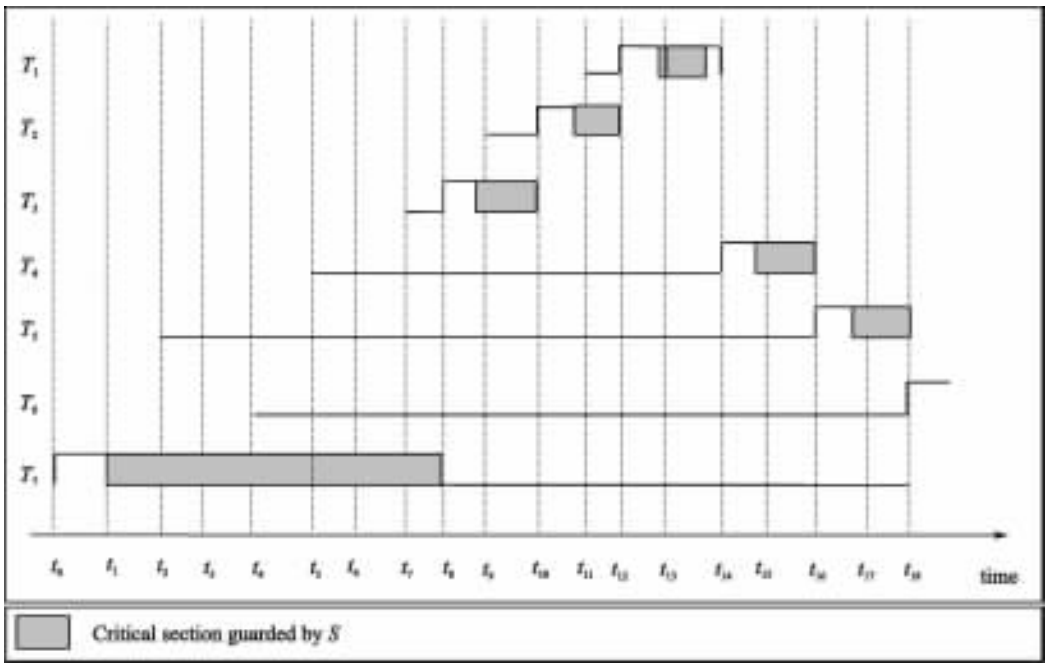


图 5－24　基于优先级天花板协议的任务执行情况

在图 5－24 中，任务的执行情况如下：

① t_0 时刻，任务 T_7 开始运行，并于时刻 t_1 获得信号量 S，进入临界区，任务 T_7 的优先级被抬升到信号量 S 的优先级天花板（为任务 T_1 的优先级）。

② t_2 时刻，任务 T_5 就绪，但由于其优先级低于任务 T_7 当前的优先级而无法运行 。

③ t_4 时刻，任务 T_6 就绪，但由于其优先级低于任务 T_7 当前的优先级而无法运行。

④ t_5 时刻，任务 T_4 就绪，但由于其优先级低于任务 T_7 当前的优先级而无法运行。

⑤ t_7 时刻，任务 T_3 就绪，但由于其优先级低于任务 T_7 当前的优先级而无法运行。

⑥ t_8 时刻，任务 T_7 释放信号量 S，退出临界区，其优先级回复到原来的值，当前具有最高优先级的就绪任务 T_3 运行；随后，任务 T_3 获得信号量 S，任务 T_3 的优先级被抬升到信号量 S 的优先级天花板。

⑦ t_9 时刻，任务 T_2 就绪，但由于其优先级低于任务 T_3 当前的优先级而无法运行。

⑧ t_{10} 时刻，任务 T_3 释放信号量 S 并运行完成，其优先级回复到原来的值，当前具有最高优先级的就绪任务 T_2 运行；随后，任务 T_2 获得信号量 S，任务 T_2 的优先级被抬升到信号量 S 的优先级天花板。

⑨ t_{11} 时刻，任务 T_1 就绪，但由于其优先级与任务 T_2 当前的优先级相等而没有被调度运行。

⑩ t_{12} 时刻，任务 T_2 释放信号量 S 并运行完成，其优先级回复到原来的值，当前优先级最高的就绪任务 T_1 运行；然后，任务 T_1 获得信号量 S，并于 t_{14} 时刻任务 T_1 运行完毕，当前优先级最高的就绪任务 T_4 开始运行；最后，任务 T_4 获得信号量 S，任务 T_4 的优先级被抬升到信号量 S 的优先级天花板。

⑪ t_{16} 时刻，任务 T_4 释放信号量 S 并运行完成，其优先级回复到原来的值，当前优先级最高的就绪任务 T_5 开始运行；随后，任务 T_5 获得信号量 S，其优先级被抬升到信号量 S 的优先级天花板。

⑫ t_{18} 时刻，任务 T_5 释放信号量 S 并运行完成，优先级回复到原来的值，当前优先级最高的就绪任务 T_6 开始运行。

2. 对程序运行过程影响程度的比较

优先级天花板协议的特点是一旦任务获得某临界资源，其优先级就被抬升到可能的最高程度，而不管此后在它使用该资源的时间内是否真的有高优先级任务申请该资源。这样就有可能影响某些中间优先级任务的完成时间。但在优先级继承协议中，只有当高优先级任务申请已被低优先级任务占有的临界资源这一事实发生时，才抬升低优先级任务的优先级，因此优先级继承协议对任务执行流程的影响相对要较小。这可以从例 5－11 进行说明。

例 5－11 假设系统中存在四个任务，按优先级从高到低分别为 T_1，T_2，T_3，T_4；任务 T_1 和 T_4 共享由信号量 S 控制的临界资源。假定在某次运行过程中，T_4 首先运行并获得信号量 S。根据优先级天花板协议，T_4 的优先级被抬升到 T_1 的水平，直到它释放信号量 S，其优先级才会回复到原先的水平。假定 T_4 的优先级被抬升的时间长度为 t，如果在时间段 t 内，T_2，T_3 就绪，而 T_1 未就绪，则 T_2，T_3 的执行被延迟。如果在 T_1 和 T_4 间存在的中等优先级任务越多，则造成这种延迟的几率就越大。

为了解决例 5－11 中出现的中等优先级任务的处理被延迟的情况，可以采用另外一种形式的优先级天花板协议，并称这种优先级天花板协议为基于优先级继承的优先级天花板协议。

① 信号量的优先级天花板为使用该信号量的任务的最高优先级。只有在任务 T 的优先级高于所有当前被其他任务阻塞(如果任务获得一个信号量，则称该信号量被任务阻塞)的信号量的优先级天花板时，任务 T 才能进入临界区。

② 若任务 T 拥有所有处于就绪状态的任务的最高优先级，获得了 CPU 资源，并设信号量 S^* 为当前被其他任务所阻塞的所有信号量中具有最高优先级天花板的信号量；若任务 T 的优先级不高于信号量 S^* 的优先级天花板，则任务 T 将不能进入临界区而被阻塞。在这种情况下，称任务 T 被拥有信号量 S^* 的任务阻塞在信号量 S^* 上；否则，任务 T 将进入临界区。

③ 通常情况下，任务 T 将使用被分配的优先级进行执行，除非该任务在临界区的执行过程中阻塞了其他高优先级任务。如果任务 T 阻塞了高优先级任务，则任务 T 将继承被任务 T 所阻塞的具有最高优先级任务的优先级。当任务 T 退出临界区时，将恢复到其进入临界区时所拥有的优先级，并且，优先级继承具有传递性。

④ 如果任务 T 不试图进入临界区，则任务 T 可以抢占低优先级的任务执行。

在基于优先级继承的优先级天花板协议中，任务 T 进入临界区并不会立即抬升 T 的优先级，只有在另一高优先级任务抢占任务 T，并在高优先级任务试图进入临界区的时候，如果高优先级任务的优先级不高于当前被阻塞信号量的优先级天花板，高优先级任务被阻塞，任务 T 的优先级才会由于继承高优先级任务的优先级而得到抬升。这样，中等优先级任务处理的延迟情况就能得到较大改善。为了便于区别，把前面提到的优先级天花板协议称为简单优先级天花板协议。

同简单优先级天花板协议相比，基于优先级继承的优先级天花板协议能够改善中等优先级任务的延迟处理情况，但也存在优先级继承协议中任务执行临界区的过程中可能导致任务的优先级需要进行多次改变的情况，不如简单优先级天花板协议中对任务优先级的一次性抬升简单。

例 5－12　假设系统中有三个任务 T_1，T_2，T_3，并拥有三个临界区，分别通过信号量 S_1，S_2 和 S_3 来控制。每个任务的处理顺序如下：

$$T_1=\{\cdots,P(S_1),\cdots,V(S_1),\cdots\}$$

$$T_2=\{\cdots,P(S_2),\cdots,P(S_3),\cdots,V(S_3),\cdots,V(S_2),\cdots\}$$

$$T_3=\{\cdots,P(S_3),\cdots,P(S_1),\cdots,V(S_2),\cdots,V(S_3),\cdots\}$$

由于任务 T_2 的优先级高于 T_3，因此，信号量 S_2 和 S_3 的优先级天花板相同，为任务 T_2 的优先级。事件发生的先后顺序如图 5－25 所示。

图 5－25 中：

① 在时刻 t_0，任务 T_3 被初始化并开始执行，然后加锁信号量 S_3。

② 在时刻 t_1，任务 T_2 被初始化，并抢占任务 T_3。

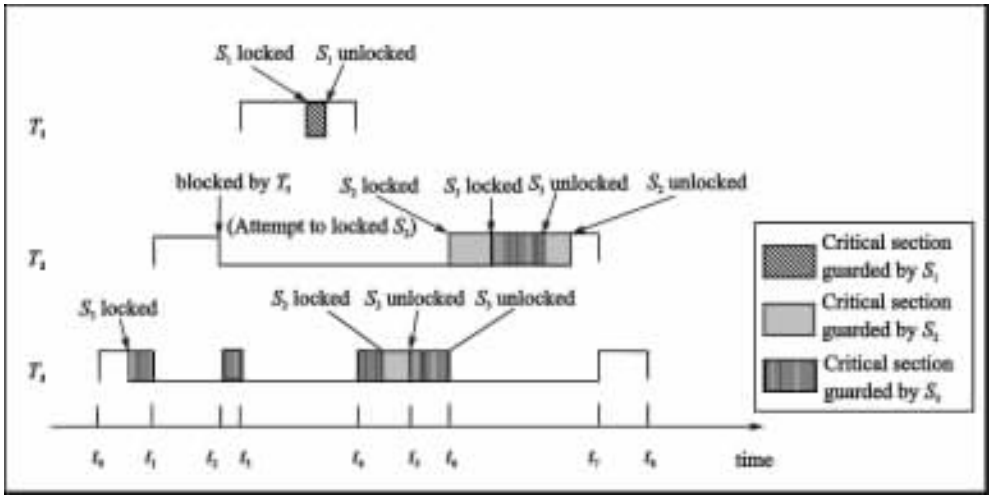


图 5-25　基于优先级继承的优先级天花板协议的事件发生的先后顺序

③ 在时刻 t_2，任务 T_2 通过 $P(S_2)$试图进入临界区。由于任务 T_2 的优先级不高于当前被加锁信号量 S_3 的优先级天花板，因此，任务 T_2 不能进入 S_2 控制的临界区，并被阻塞。任务 T_3 继承任务 T_2 的优先级，然后继续执行。这样，任务 T_2 不能进入 S_2 控制的临界区，并被挂起，就避免了死锁的发生。

④ 在时刻 t_3，任务 T_1 被初始化，并抢占任务 T_3；然后，任务 T_1 试图加锁信号量 S_1。由于任务 T_1 的优先级高于被锁住的信号量 S_3 的优先级，任务 T_1 将被允许加锁信号量 S_1，并在临界区中执行。

⑤ 在时刻 t_4，任务 T_1 退出临界区，完成任务的执行。任务 T_3 继续执行，然后加锁信号量 S_2。

⑥ 在时刻 t_5，任务 T_3 释放信号量 S_2。

⑦ 在时刻 t_6，任务 T_3 释放信号量 S_3，其优先级恢复到被分配的优先级，并唤醒任务 T_2，使得任务 T_2 成为具有高优先级的任务，抢占任务 T_3 继续执行，随后加锁信号量 S_3；然后，任务 T_2 加锁信号量 S_2，执行嵌套临界区，释放信号量 S_2；最后，任务 T_2 释放信号量 S_3，并开始执行非临界区。

⑧ 在时刻 t_7，任务 T_2 完成执行，任务 T_3 开始执行。

⑨ 在时刻 t_8，任务 T_3 完成执行。

例 5-12 中，任务 T_1 由于其优先级比信号量 S_2 和 S_3 的优先级天花板都高，不会被阻塞。在$[t_2, t_3]$和$[t_4, t_6]$，任务 T_2 被低优先级任务 T_3 阻塞。事实上，出现这些阻塞时间段是由于任务 T_3 加锁信号量 S_3 的需要。但尽管如此，任务 T_2 被低优先级任务 T_3 阻塞的时间仍然不会超过低优先级任务的一个临界区执行时间的长度。这反映了基于优先级继承的优先级天花板协议的一个重要特性：任务阻塞的最大时间间隔为低优先级任务在单个临界区内的执行时

间。另外，简单优先级天花板协议也具有该特性，例如在图 5－24 中，任务 T_5 在$[t_2, t_8]$时间段和任务 T_8 在$[t_4, t_8]$时间段的阻塞情况。

例 5－13　考虑例 5－9 会出现阻塞链的情况。假定任务 T_1 需要顺序使用信号量 S_1 和 S_2，且任务 T_2 需要使用信号量 S_2，任务 T_3 需要使用信号量 S_1。因此，信号量 S_1 和 S_2 的优先级天花板相同，为任务 T_1 的优先级，设为 p_1。同例 5－9 一样，在时刻 t_0，任务 T_3 加锁信号量 S_1。在时刻 t_1，任务 T_2 被初始化，并抢占任务 T_3。在时刻 t_2，任务 T_2 试图加锁信号量 S_2，但其优先级不高于被阻塞信号量 S_1 的优先级天花板 p_1，使得加锁信号量 S_2 失败而被阻塞。任务 T_3 继承任务 T_2 的优先级，继续执行。在时刻 t_3，任务 T_3 仍然在临界区中，任务 T_1 被初始化，此时只有信号量 S_1 被加锁。在时刻 t_4，任务 T_1 在试图拥有已被任务 T_3 加锁的信号量 S_1 时被阻塞。因此，任务 T_3 继承任务 T_1 的优先级。在时刻 t_5，任务 T_3 退出临界区，恢复到其被分配的优先级继续执行，然后被任务 T_1 抢占，任务 T_1 继续执行。

例 5－13 中，在$[t_4, t_5]$时间段，任务 T_1 被任务 T_3 阻塞。该时间段对应着任务 T_3 中被信号量 S_1 控制的临界区，而且任务 T_2 在$[t_2, t_3]$和$[t_4, t_5]$被阻塞也是由于 T_3 中被信号量 S_1 控制的同一个临界区的原因。

例 5－14　假定任务 T_1 有如下执行序列：$\{\cdots, P(S_1), \cdots, V(S_1), \cdots, P(S_2), \cdots, V(S_2), \cdots\}$；任务 T_2 具有如下执行序列：$\{\cdots, P(S_3), \cdots, V(S_3), \cdots\}$；任务 T_3 具有如下执行序列：$\{\cdots, P(S_3), \cdots, P(S_2), \cdots, V(S_2), \cdots, V(S_3), \cdots\}$。其中，信号量 S_1 和 S_2 的优先级天花板为 p_1，信号量 S_3 的优先级天花板为 p_2。

图 5－26 中为事件的执行序列。

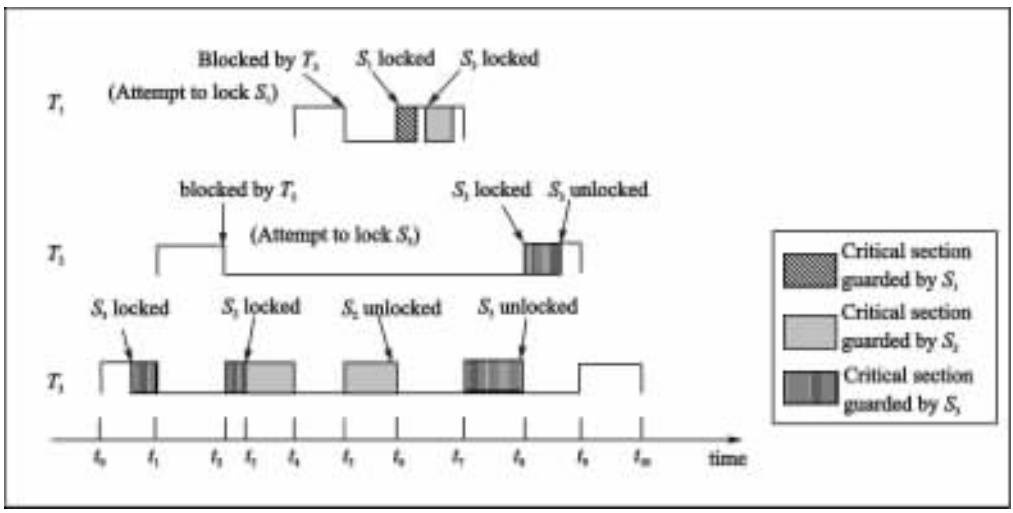


图 5－26　基于优先级继承的优无级天花板协议的事件执行序列

事件的执行序列描述如下：

① 在时刻 t_0，任务 T_3 开始执行，然后加锁信号量 S_3。

② 在时刻 t_1，任务 T_2 被初始化，抢占任务 T_3，然后开始执行。

③ 在时刻 t_2，任务 T_2 通过 $P(S_2)$试图进入临界区。由于任务 T_2 的优先级不高于被加锁信号量 S_3 的优先级天花板，因此，任务 T_2 不能进入 S_2 控制的临界区，并被挂起。任务 T_3 继承任务 T_2 的优先级 p_2，然后继续执行。

④ 在时刻 t_3，任务 T_3 进入信号量 S_2 控制的临界区。任务 T_3 被允许进入临界区，是由于当前没有信号量被其他任务所阻塞。

⑤ 在时刻 t_4，任务 T_3 仍然在临界区执行，但具有最高优先级的任务 T_1 被初始化。任务 T_1 抢占任务 T_3，并继续执行。

⑥ 在时刻 t_5，任务 T_1 试图进入信号量 S_1 控制的临界区。该信号量未被其他任务加锁。但由于任务 T_1 的优先级不高于被加锁信号量 S_2 的优先级天花板，使得任务 T_1 被 T_3 阻塞在信号量 S_2 上。这种阻塞是一种新的阻塞形式，不同于前面提到的直接和间接阻塞形式。然后任务 T_3 按照从任务 T_1 继承的优先级 p_1 继续执行。

⑦ 在时刻 t_6，任务 T_3 退出信号量 S_2 控制的临界区，其优先级恢复到从任务 T_2 继承来的优先级 p_2。此时，由于任务 T_1 的优先级高于当前被阻塞的信号量 S_3 的优先级天花板，使得任务 T_1 被激活，并抢占任务 T_3 得到执行。然后任务 T_1 相继进入和退出信号量 S_1 和 S_2 控制的临界区。

⑧ 在时刻 t_7，任务 T_1 完成执行，任务 T_3 按照继承的优先级 p_2 在信号量 S_3 控制的临界区中执行。

⑨ 在时刻 t_8，任务 T_3 退出信号量 S_3 控制的临界区，并恢复到其原有优先级。然后任务 T_2 被激活，抢占任务 T_3，并执行信号量 S_3 控制的临界区，之后退出。

⑩ 在时刻 t_9，任务 T_2 完成执行，任务 T_3 开始执行。

基于优先级继承的优先级天花板协议具有如下特性：

① 只有在任务 T 的优先级不高于当前所有被其他低优先级任务所获得的信号量的最高优先级天花板的情况下，任务 T 才会被其他低优先级任务所阻塞；

② 优先级天花板协议能够避免死锁；

③ 若任务 T_i 的优先级比 T_j 的优先级高，则 T_i 被任务 T_j 阻塞的最长时间为任务 T_j 中能够阻塞任务 T_i 的具有最长执行时间的临界区的执行时间长度。

优先级继承协议中存在直接阻塞和间接阻塞两种阻塞形式，简单优先级天花板协议中也存在间接阻塞的形式(如例 5－10 的图 5－24 中，任务 T_5 在$[t_2, t_8]$时间段和任务 T_8 在$[t_4, t_8]$时间段的阻塞情况)。基于优先级继承的优先级天花板协议则存在另一种阻塞形式，可称该种阻塞形式为天花板阻塞(如例 5－14 中，在时刻 t_5，任务 T_1 所发生的阻塞)。天花板阻塞是必需的，用来防止死锁和阻塞链的发生。尽管如此，优先级天花板协议仍极大地改善了最坏阻塞

的情况。使用优先级继承协议，任务 T 被阻塞的最长时间为可能阻塞任务 T 的所有临界区的执行时间之和。与此相反，在优先级天花板协议下，任务 T 被阻塞的最长时间为一个最长临界区的执行时间。

另外，优先级天花板协议还能够解决优先级继承协议中存在的死锁和阻塞链问题。

5.6　多处理器调度

5.6.1　概　述

多处理系统(multiprocessing system)指包含有超过一个 CPU 的系统。使用多处理系统主要有三个方面的原因：

① 多处理系统能够提高整个系统的性能，即增加系统的吞吐量；

② 多处理系统能够增加系统的可获得性和可靠性；

③ 使用多处理系统能够降低系统的成本。

多处理系统可分为松耦合(loosely coupled)和紧耦合(tightly coupled)两种体系结构模型。松耦合多处理系统就像一个网络一样，也称为分布式系统，其中的每一个计算机模块都包含一个处理器、本地内存、I/O 接口和通道切换等内容。计算机模块之间通过消息传送系统进行连接，这些消息传送系统就像一个仲裁(arbiter)机构，接收计算机模块发送的消息，然后通过调度系统确定处理该消息的计算机模块。紧耦合多处理系统中各处理器之间共享主存储器，多用于对处理速度要求较高的系统。通常，紧耦合系统趋向于用做并行系统(解决同一问题)，而松耦合系统趋向于作为分布式系统(解决多个不相关的问题)。

用于多处理系统的操作系统与用于单处理系统的操作系统之间没有大的差别，差异性主要在于多处理系统中所有处理器必须要同时工作。用于多处理系统的操作系统应具备系统重构功能，以进行优美降级(graceful degradation)，并考虑如何有效提高资源的使用率。

为使多处理系统中的多个处理器都能够同时工作，操作系统应该进行任务分配，把任务分配到多处理系统中的各个处理器上。在实际应用中，任务在处理器上的最佳分配是一个 NP 问题。通常的做法是先进行任务分配，然后检查调度的可行性。如果调度是不可行的，就需要对分配方案进行修订，以确保调度的可行性。

5.6.2　使用率平衡算法

使用率平衡算法(utilization - balancing algorithm)试图平衡处理器的使用率，把任务一个一个地分配到具有最低处理器使用率的处理器上。算法描述如图 5 - 27 所示。

该算法考虑了容错的需要，能够在多个处理器上同时运行同一个任务的多个拷贝。

```
for 每个任务T_i,循环执行
    分配任务T_i 的拷贝到T_i 个具有最低使用率的处理器;
    根据分配情况，更新处理器的使用率。
end for
```

图 5-27　使用率平衡算法描述

5.6.3　基于 RMS 的任务分配算法

基于 RMS 的任务分配算法也是基于处理器使用率并结合使用 RMS 的分配算法。参与调度的任务需要满足 RMS 中的假设条件，且认为多处理系统中各处理器都相同，任务除了使用 CPU 外，不需要使用其他资源。

基于 RMS 的任务分配算法对 CPU 使用率进行分级，如果任务的 CPU 使用率属于某个级别的范围，且该任务同该级别范围中已分配任务满足 RMS 的可调度性要求，则把该任务分配到该级别对应的处理器上。如果把 CPU 使用率分为 M 级，则每级对应的 CPU 使用率范围为

$$\left(2^{1/(j+1)}-1,2^{1/j}-1\right] \qquad (j<M)$$

$$\left(0,2^{1/j}\right] \qquad (j<M)$$

例 5-15　假定存在四个级别，$M=4$。表 5-6 即为各个级别及其所对应的 CPU 使用率范围。

表 5-6　CPU 使用率范围

级　别	CPU 使用率范围	级　别	CPU 使用率范围
C_1	(0.41,1]	C_3	(0.19,0.26]
C_2	(0.26,0.41]	C_4	(0.00,0.19]

考虑表 5-7 中的周期任务。

表 5-7　周期任务

项　目	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
执行时间	5	7	3	1	10	16	1	3	9	17	21
周期	10	21	22	24	30	40	50	55	70	90	95
CPU 使用率	0.50	0.33	0.14	0.04	0.33	0.40	0.02	0.05	0.13	0.19	0.22
级别	C_1	C_2	C_4	C_4	C_2	C_2	C_4	C_4	C_4	C_4	C_3

处理器的数量大于或等于级别数，可先认为每个级别对应了一个不同的处理器，并假定 p_i 与 c_i 对应($1 \leqslant i \leqslant 4$)。根据表中的情况，$T_1$ 分配给 p_1，T_2 分配给 p_2，T_3 分配给 p_4。$T_4 \in c_4$，且$\{T_3, T_4\}$满足在同一个处理器上的RMS可调度性要求，因此，T_4 也分配给 p_4。$T_5 \in c_2$，且$\{T_2, T_5\}$满足在同一个处理器上的 RMS 可调度性要求，因此，T_5 也分配给 p_2。$T_6 \in c_2$，但$\{T_2, T_5, T_6\}$不满足在同一个处理器上的 RMS 可调度性要求，需要增加一个处理器，称为 p_5，并把 T_6 分配到 p_5 上。$T_7 \in c_4$，且$\{T_3, T_4, T_7\}$满足在同一个处理器上的 RMS 可调度性要求，因此，T_7 也分配给 p_4。同理，T_8，T_9，T_{10}也可分配到 p_4 上。T_{11}分配到 p_3 上。任务在处理器上的分配情况如表 5－8 所列。

表 5－8　任务在处理器上的分配情况

处理器	分配在处理器上的任务
p_1	T_1
p_2	T_2, T_3
p_3	T_{11}
p_4	$T_3, T_4, T_7, T_8, T_9, T_{10}$
p_5	T_6

根据这种分配，在每个处理器上运行的任务都能够满足 RMS 的可调度性要求。

5.6.4　基于 EDF 的首次匹配算法

假定多处理系统中的处理器都是相同的，需要分配到各处理器上的任务为独立、可抢占的周期任务，任务的相对截止时间与周期相等，且任务除了需要 CPU 外，不使用其他资源。

对 EDF 算法，只要分配到单个处理器上的任务的 CPU 使用率不大于 1，这些任务就能满足 EDF 算法的可调度性要求。因此，算法可以简化为考虑任务分配后，各个处理器上的所有任务的 CPU 使用率不超过 1。另外，还期望使用处理器的数量尽可能地少。基于 EDF 的首次匹配算法即是为解决这些问题而提出的。

例 5－16　考虑表 5－9 中的任务集。

表 5－9　任务集

项　目	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
执行时间	5	7	3	1	10	16	1	3	9	17	21
周期	10	21	22	24	30	40	50	55	70	90	95
CPU 使用率	0.50	0.33	0.14	0.04	0.33	0.40	0.02	0.05	0.13	0.19	0.22

把任务按照 CPU 使用率进行降序排列，为 $L=(T_1, T_6, T_2, T_5, T_{11}, T_{10}, T_3, T_9, T_8, T_4, T_7)$。任务的分配情况如表 5－10 所列。在 $U=(u_1, u_2, u_3, \cdots)$中，$u_i$ 表示分配给处理器 p_i 的 CPU 资源使用率。本例中 $U=(1.00, 0.90, 0.45)$，表示需要三个处理器 p_1, p_2, p_3 来运行所有的 11 个任务，p_1 上各任务的 CPU 使用率为 1.00，p_2 上各任务的 CPU 使用率为 0.90，p_3 上各任务的 CPU 使用率为 0.45。

表 5-10　任务分配情况

算法步骤	任　务	CPU 使用率	分配到的处理器	分配后的使用率情况
1	T_1	0.5	p_1	(0.50)
2	T_6	0.40	p_1	(0.90)
3	T_2	0.33	p_2	(0.90,0.33)
4	T_5	0.33	p_2	(0.90,0.66)
5	T_{11}	0.22	p_2	(0.90,0.88)
6	T_{10}	0.18	p_3	(0.90,0.88,0.18)
7	T_3	0.14	p_3	(0.90,0.88,0.32)
8	T_9	0.13	p_3	(0.90,0.88,0.45)
9	T_8	0.06	p_1	(0.96,0.88,0.45)
10	T_4	0.04	p_1	(1.00,0.88,0.45)
11	T_7	0.02	p_2	(1.00,0.90,0.45)

思考题

5.1　请说明什么叫任务，任务有哪些主要特性，主要包含哪些内容，并说明任务、进程与线程三个概念之间的区别。

5.2　请描述一个实际的系统，该系统包含多个任务，多个任务协同实现系统功能。

5.3　请说明任务主要包含哪些参数，并对参数的含义进行解释。

5.4　请说明任务主要包含哪些状态，并就状态之间的变迁情况进行描述。

5.5　以 Windows 或 Linux 为例，收集资料，分析描述操作系统的进程/线程模型，并基于所选择的操作系统，选择合适的简单应用，利用操作系统提供的关于进程/线程的编程接口，编写完整的、能反映多任务运行情况的示例程序。

5.6　请说明什么叫任务切换，任务切换通常在什么时候进行以及任务切换的主要工作内容。

5.7　请说明任务调度有哪些分类方法，并说明每种分类下的主要调度方法。

5.8　请解释什么叫 RMS 和 EDF 调度方法，并分别说明 CPU 使用率的可调度范围。

5.9　基于 RMS 调度算法，为表 5-11 中的任务分配优先级(假定数字越大，优先级越低)。如果所有任务的运行时间均为 6 ms，请问这些任务是否是可调度的？请用图示和文字描述的方式对任务的运行情况进行详细说明。

表 5-11　任务及周期

任　务	周期/ms	任　务	周期/ms
T_1	25	T_4	150
T_2	60	T_5	75
T_3	50	T_6	50

5.10　给定一组任务，如表 5-12 所列。

表 5-12　任务、执行时间及周期

任　务	执行时间	周　期
T_1	1	5
T_2	1	10
T_3	2	20
T_4	10	50
T_5	7	100

请分别基于 RMS 和 EDF 调度算法，用图示和文字描述的方式对任务的运行情况进行详细说明，并比较两种调度算法下任务的上下文切换情况。

5.11　构建一组周期任务（包括任务到达时间、执行时间和周期），使得这些任务采用 EDF 调度算法是可调度的，但在 RMS 调度算法下是不可调度的。请用图示和文字描述的方式对任务的运行情况进行详细说明。

5.12　给定一组优先级顺序升高的任务：任务 a、任务 b、任务 c 和任务 d。其中，任务 a 的执行序列为 EQQE，到达时间为 0；任务 b 的执行序列为 EVQE，到达时间为 2；任务 c 的执行序列为 EVE，到达时间为 3；任务 d 的执行序列为 EQVE，到达时间为 4。在执行序列中，Q 和 V 为共享资源，E 表示执行。另外，四个任务的周期均为 25。请分别基于优先级继承协议、简单优先级天花板协议和基于优先级继承的优先级天花板协议，用图示和文字说明的方式对任务的运行情况进行详细说明，并给出每个任务的最大阻塞时间。

5.13　什么叫优先级反转？解决优先级反转有哪些主要方法？分别就这些方法进行描述。

5.14　举例说明如何实现多处理器调度。

5.15　以一种开源的嵌入式操作系统为例，就该操作系统所采用的任务管理（包括任务状态及变迁、任务的数据结构、实现任务管理的应用编程接口等内容）和任务调度策略进行详细分析，并写出分析报告。

第 6 章　同步、互斥与通信

本章主要讲解嵌入式操作系统内核提供的各种同步、互斥与通信机制，包括信号量、消息队列、事件和异步信号等。对于每一种机制，均从主要数据结构、功能及有关的资源配置等方面进行阐述。在信号量部分，对信号量的分类、与互斥信号量有关的优先级反转、嵌套资源访问和删除安全等问题进行了说明；在讲解异步信号时，将它与中断机制和事件机制进行了详细的对比。希望通过本章的学习，读者能够了解同步、互斥与通信机制的基本原理，以便在做具体应用开发时能正确、合理地使用它们。

6.1　概　述

嵌入式实时多任务应用是由多个任务、多个中断处理过程以及嵌入式实时操作系统组成的有机整体。嵌入式实时操作系统为应用提供系统管理、底层的支持、协调任务和中断处理程序等功能。

以下是任务之间的关系：

① 相互独立　仅仅是竞争 CPU 资源。

这里的“相互独立”是指任务间除了竞争 CPU 外，再无其他关联。随后谈到的几种任务间关系并不影响任务本身的独立性，即它们都是独立运行的单元，是资源竞争和系统调度的实体。

② 竞争除 CPU 外的其他资源(互斥)　任务必须能对共享资源进行互斥访问。

共享资源是多任务系统中主要关心的问题。在大多数情况下，有些资源在某一时刻仅能被某一任务使用，并且在使用过程中不能被其他任务中断。这些资源主要包括特定的外设、共享内存以及 CPU。当 CPU 禁止并发操作时，那些包含了使用 CPU 之外的共享资源的代码就不能同时被多个任务调用执行。这样的代码就称为“临界区域”。如果两个任务同时进入同一临界区域，将会导致意想不到的错误。

③ 同步　协调彼此运行的步调。

④ 通信　彼此间传递数据或信息，以协同完成某项工作。

通信可以是在任务与任务之间，也可以是在中断服务程序(ISR)与任务之间。

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多个任务与多个中断处理程序(ISR)共同完成。它们之间必须协调动作，互相配合，甚至需要交换信息，即进行通信。具体说来，任务能以以下方式与中断处理程序或其他任务进行同步或通信。

- **单向同步或通信**　一个任务与另一个任务或者一个 ISR 同步或通信。如图 6－1 和图 6－2所示，比如一个任务做 I/O 操作，然后等待回应信号。当 I/O 操作完成时，中断服务程序(或另外一个任务)发出信号，该任务得到信号后继续往下执行。
- **双向同步或通信**　两个任务相互同步或通信。如图 6－3 所示，两个任务用两个不同的标志同步它们的行为。双向同步不可能在任务与 ISR 之间进行，因为 ISR 不可能等待一个标志。

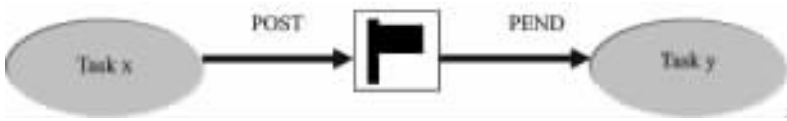


图 6－1　任务与任务之间的同步(单向)

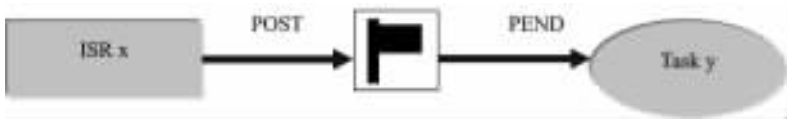


图 6－2　任务与中断之间的同步(单向)

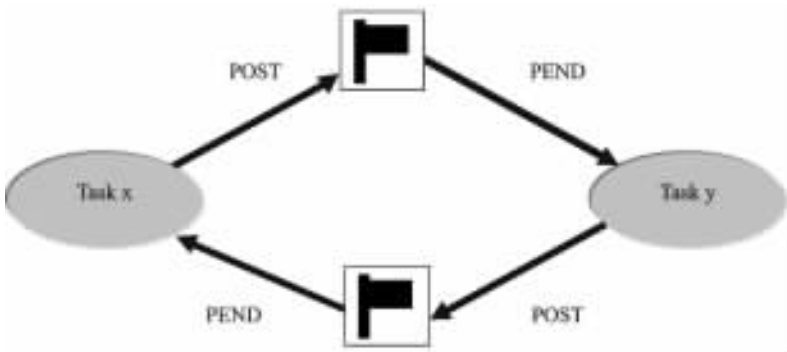


图 6－3　任务与任务之间的同步(双向)

说明：图中的旗帜标志着某一事件的发生信号。

由此也可以看出，在一个嵌入式多任务系统中，任务间的耦合程度是不一样的：如果两个任务需要进行大量的通信，则它们的耦合程度较高，相应的系统开销较大；如果两个任务之间不存在通信需求，其间的同步关系很弱甚至不需要同步或互斥，则耦合程度较低，系统开销较小。研究任务间耦合程度的高低对于合理地设计应用系统和划分任务有很重要的作用。

嵌入式实时内核一般都提供一套丰富的同步、互斥与通信的机制，在单处理器平台上主要包括：

- 信号量(semaphore),用于基本的互斥与同步;
- 事件(组)(event group),用于同步;
- 异步信号(asynchronous signal),用于同步;
- 邮箱(mailbox)、消息队列(message queue)或管道(pipe),用于消息通信。

上述机制是本章讲述的重点。另外,以下一些机制也可用于同步与通信(在单处理器或多处理器系统中)。

- 全局变量　在一个嵌入式系统中,操作系统、任务及中断服务程序的代码和数据是链接在一起的,通过全局变量可以完成一定的同步与通信功能。
- 共享内存　在单处理器或多处理器的嵌入式系统中,如果实施了内存保护,任务之间无法直接访问彼此的地址空间时,可以使用共享内存机制来完成同步与通信。
- 对于分布在不同嵌入式系统上的任务还可以通过 Sockets 和远程过程调用 RPC,实现任务间透明的网络通信与同步。

6.2　信号量

6.2.1　信号量的种类及用途

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。根据用途,信号量一般分为三种:用于解决互斥问题的互斥信号量、用于解决同步问题的二值信号量和用于解决资源计数问题的计数信号量。其中互斥信号量比较特殊,可能会引起优先级反转问题。

信号量可以让一定数量的任务同时访问共享资源。只有当所有希望使用某个资源的任务都使用同一个信号量时,信号量才能起到保护资源的作用,如果某个任务不使用该信号量而直接访问资源,那么信号量就不能保护资源。

通常,信号量被设置成只允许最多一个任务在给定的时间内访问某个资源,这就是所谓的以二值模式使用信号量。在这种模式下,信号量的计数不是 0 就是 1,这在互斥或同步共享数据的访问时是很有用的。

用信号量保护的代码区有时称为“临界区”。当用做互斥时,信号量被初始化成 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”的任务将成功获得信号量,而所有其他的任务就必须等待。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。

当二值信号量用于同步时,被初始化成 0,表示同步的事件尚未发生。一个任务申请信号量以等待该同步事件的发生。另一个任务到达同步点时,释放信号量(将其值设置为 1)表示同步事件已发生,以唤醒等待的任务。

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，因此信号量被初始化为 n（非负整数）。n 为该种共享资源的数目。

嵌入式实时内核将信号量细分为上述三类，是因为根据其用途，在具体实现这些信号量的时候可以做专门的处理，以提高执行的效率和可靠性。

6.2.2　互斥信号量

对一个共享资源进行互斥访问的方法包括禁止中断、禁止任务切换以及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。禁止中断完全将 CPU 与外部世界隔断开来，如果对共享资源的访问需要较多的机器指令，则可能增加中断延迟。禁止任务切换没有禁止中断激进，但它不加选择地阻止所有其他任务的执行，包括与该资源完全无关的高优先级任务。一个更精确的方法是使用信号量，因为该方法只影响实际竞争该资源的任务，可以说它控制的精度更高，影响的范围更小（对这些互斥机制的详细说明和比较请参看本书第 4 章中的有关内容）。确实，很多多任务操作系统都提供了互斥信号量——一种为解决互斥中的固有问题（优先级反转、删除安全和对资源的递归访问等）的特殊信号量，或简称为互斥体。

互斥信号量机制非常普遍地应用在嵌入式操作系统和各种可重入的库中，因此即便不直接使用该机制，应用仍然可能正在使用一个互斥体。例如，堆管理例程诸如 malloc()，free() 和 realloc()等一般都是被一个互斥体保护的，因此多任务能够并发地访问（共享的）堆资源。

然而，尽管普遍存在（或可能正因为这样），互斥体是所有机制中最危险的，因为它可能导致优先级反转。关于优先级反转及解决优先级反转的两种方法（优先级继承和优先级天花板），请参见第 5 章的详细叙述。

1. 嵌套（递归）资源访问

一般来说，当已经占有一个互斥信号量的任务企图获得同一个互斥信号量时，死锁就会发生。由于信号量已经分配给了该任务，它还没有被释放，但同时此任务又在等待该信号量被释放，因此该任务将永远不会再执行。

有些实时内核允许对互斥信号量的嵌套访问。嵌套访问是指某个任务再次申请自身已占用的互斥信号量时，该信号量的嵌套层数增加 1。互斥信号量的嵌套层数反映了它对应的资源被同一个任务嵌套访问的次数。释放该互斥信号量时，仅当释放到最后一层时，才真正释放该信号量的使用权限，即其他申请此信号量的任务才能获得它。每个获取信号量的调用必须与释放信号量的调用相匹配。当最外层的获取信号量的调用与释放信号量的调用匹配时，该信号量才允许被其他任务访问。

用于同步的信号量不支持嵌套访问，任务如果对同步信号量使用上述操作会被永久阻塞，阻塞条件永远不会解除。

2. 删除安全

另一个与互斥有关的问题是任务删除。在一个受信号量保护的临界区，经常需要保护在临界区执行的任务不会被意外地删除。删除一个在临界区执行的任务可能引起意想不到的后果，造成保护资源的信号量不可用，可能导致资源处于破坏状态，也就导致了其他所有要访问该资源的任务无法得到满足。

为了避免任务在临界区执行时不被意外删除，可以提供“任务保护”和“解除任务保护”这样一对原语作为一种解决方法；同时，为互斥信号量提供“删除安全”选项。在创建信号量的时候使用这个选项，当应用每次获取信号量时隐含地使能“任务保护”功能，当每次释放信号量时隐含地使能“解除任务保护”功能，从而使任务在得到信号量的同时能够得到不会被删除的保护。

6.2.3 二值信号量

二值信号量主要用于任务与任务之间、任务与中断服务程序之间的同步，如图 6－1，6－2 和 6－3 所示。这里具体以两个任务之间的双向同步为例进行说明。

用于同步的二值信号量 sem1 和 sem2 均被初始化为 0，表示同步事件尚未产生。假定这两个任务是运行在基于优先级的抢占式调度内核之上的，并且 Task2 的优先级高于 Task1 的优先级。如果在某一时刻 Task1 和 Task2 均处于就绪状态，那么之后系统的运行过程如图 6－4所示。首先 Task2 申请信号量 sem1 以同步 Task1，Task1 做完一些工作后将 sem1 置

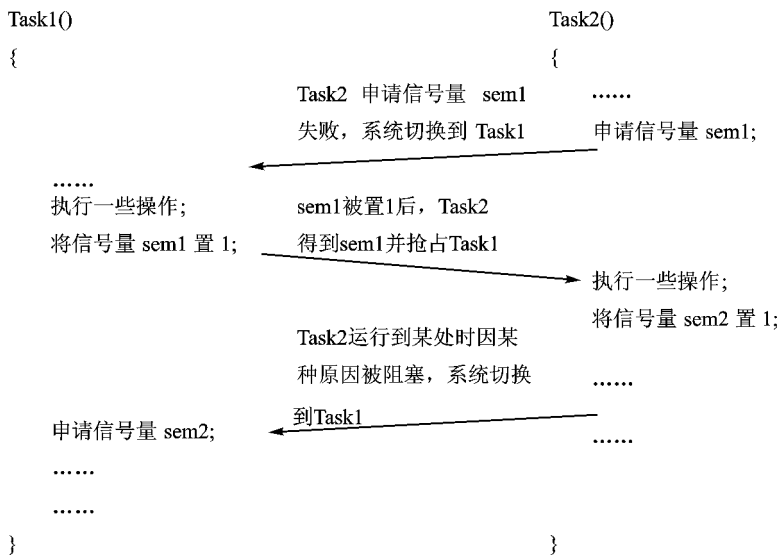


图 6－4 任务间的双向同步

1(即向 Task2 发送信号),然后申请获取信号量 sem2(即等待 Task2 完成处理后的回应)。同样的,Task2 完成处理后将 sem2 置 1(即向 Task1 发送信号)。这样,两个任务就可以实现双向同步。

6.2.4　计数信号量

计数信号量用于控制系统中共享资源的多个实例的使用。比如在下面的例子中,对有界缓冲(见图 6-5)使用的控制,就综合利用了计数信号量和互斥信号量的功能。

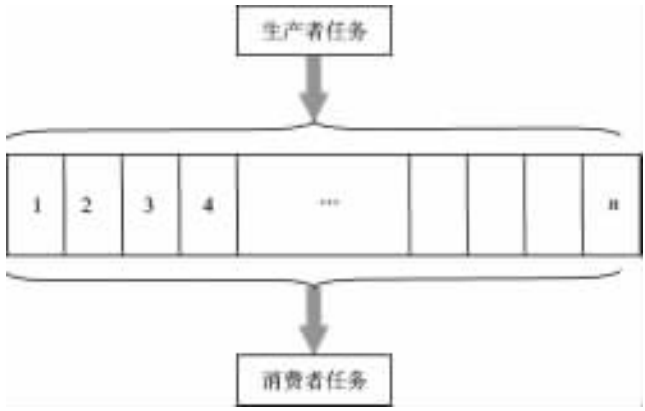


图 6-5　有界缓冲问题

用一个计数信号量 full 表示已被填充了的数据项数目,用一个计数信号量 empty 表示空闲数据项数目,它们的取值范围为 0～n,初始值分别为 0 和 n。由于有界缓冲区是共享资源,故需要用一个互斥信号量 mutex 来控制生产者任务和消费者任务对它的访问,其初始值为 1。下面是生产者任务和消费者任务的代码示意。

1. 生产者任务

```
do {
…
产生一个数据项
…
申请 empty
申请 mutex
…
将新生成的数据项添加到缓冲中
…
释放 mutex
释放 full
```

} while (1);

2. 消费者任务

do {
申请 full
申请 mutex
…
从缓冲中移出一个数据项的内容
…
释放 mutex
释放 empty
…
消费新获得的数据项内容
…
} while (1);

6.2.5 信号量机制的主要数据结构

内核使用信号量控制块来管理所有创建的信号量,在系统运行时动态地分配和回收信号量控制块。

不同的内核,其信号量控制块的具体实现不同。下面所描述的只是其中的一种。

对于前面所描述的三种类型的信号量(互斥信号量、二值信号量和计数信号量),可以采用相同的控制块结构,也可以采用不同的控制块结构。这样以便做优化处理。下面展现了二值信号量和计数信号量两种不同的控制块结构,互斥信号量的控制块结构与二值信号量的相同。

1. 二值信号量控制块结构 Binary_Semaphore_Control_Block

wait_queue	任务等待队列;
attributes	二值信号量属性;
lock	是否被占有;
nest_count	嵌套层数;
holder	拥有者。

其中的 attributes 成员又具备下面的结构:

lock_nesting_behavior	试图嵌套获得时的规则;
wait_discipline	任务等待信号量的方式;
priority_ceiling	优先级天花板值。

其中 lock_nesting_behavior 用于控制任务在试图嵌套获得二值信号量时的系统行为。如前所述,由于只有互斥信号量允许被嵌套访问,因此任务试图对其他一般的二值信号量进行此

操作时将会出错或被永久阻塞。

2. 计数信号量控制结构 Counting_Semaphore_Control_Block

wait_queue	任务等待队列；
attributes	计数信号量属性；
count	当前计数值。

其中的 attributes 成员又具备如下的结构：

maximum_count	最大计数值；
wait_discipline	任务等待信号量的方式。

6.2.6　信号量机制的主要功能

信号量机制提供相应的应用编程接口(系统调用)，一般有如下主要功能：

- 创建信号量；
- 获取(申请)信号量；
- 释放信号量；
- 删除信号量；
- 获取有关信号量的各种信息。

1. 创建信号量

根据应用传递的参数创建一个信号量，参数一般包括信号量的名字、属性和初始值。信号量的属性包括其类型、任务等待信号量的方式(即排列的顺序)以及解决优先级反转的策略。

信号量的类型如下：

- 互斥信号量(MUTEX_SEMAPHORE)；
- 计数信号量(COUNTING_SEMAPHORE)；
- 二值信号量(BINARY_SEMAPHORE)。

任务等待信号量的方式如下：

- 任务按先进先出(FIFO)顺序等待；
- 任务按优先级(PRIORITY)等待。

多个任务可能会因为试图获取同一个信号量而被挂起。如果信号量支持 FIFO 挂起，则任务以它们被挂起的顺序恢复执行；否则(信号量支持优先级挂起方式)，任务以优先级别的高低顺序为恢复执行的顺序。

优先级反转问题的解决方法(只适用于互斥信号量)如下：

- 使用优先级继承算法(INHERIT_PRIORITY)；
- 使用优先级天花板算法(PRIORITY_CEILING)。

对于任务按优先级方式排队等待的互斥信号量，可使用优先级继承或优先级天花板算法。如果采用优先级天花板算法，则必须给出所有可能获得此信号量的任务中优先级最高的任务

的优先级。

创建信号量时,内核从空闲信号量控制块链中分配一个信号量控制块,并初始化信号量属性。创建成功时,内核为此信号量分配惟一的并且有效的 ID 号,并将它返回给应用。如果已创建的信号量数量已经达到用户配置的最大数量,就返回错误。

2. 获取(申请)信号量

试图获得应用指定的信号量。这个功能的流程可简单描述如下:

if　　信号量的值大于 0;

then　将信号量的值减 1;

else　根据接收信号量的选项,将任务放到等待队列中,或是直接返回。

当所申请的信号量不能被立即获得时,可以有以下几种选择:

- 永远等待;
- 不等待,立即返回,并返回一个错误状态码;
- 指定等待时限。

获取信号量的功能可以被任务调用,也可以被中断服务程序调用。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。

为了对应用的实时确定性有所保障,任务可选择有限时间等待。有限时间等待可以有效地避免死锁的发生,这对于嵌入式实时应用来说是很重要的。

如果任务等待获得信号量,则它将被按 FIFO 或优先级顺序放置在等待队列中。如果任务等待一个使用优先级继承算法的互斥信号量,且它的优先级高于当前正占有此信号量的任务的优先级,那么占有信号量的任务将继承这个被阻塞的任务的优先级。

如果任务成功地获得一个采用优先级天花板算法的互斥信号量,它的优先级又低于优先级天花板,那么它的优先级将被抬升至天花板。

3. 释放信号量

释放一个应用指定的信号量。这个功能的流程可简单描述如下:

if　　没有任务等待这个信号量;

then　信号量的值加 1;

else　将信号量分配给一个等待任务(将相应的任务移出等待队列,使其就绪)。

如果使用了优先级继承或优先级天花板算法,那么执行该功能(系统调用)的任务的优先级将恢复到原来的高度。

4. 删除信号量

从系统中删除应用指定的一个信号量。任何知道此信号量 ID(或名字)的代码都可删除这个信号量,即删除信号量的不一定是创建信号量的任务。如果有任务正在等待获得该信号量,则执行此功能(系统调用)将使所有等待这个信号量的任务回到就绪队列中,且返回一个状态码指示该信号量已被删除。

此功能(系统调用)执行成功后,内核将信号量控制块返还给系统(即该控制块成为空闲的)。

5. 获取有关信号量的各种信息

一般地,内核还可以提供获取信号量相关信息的调用,比如获得信号量 ID 或名字。一个信号量被创建后,系统将一个惟一的 ID 分配给它,直到它被删除。获得信号量的 ID 可通过两种途径:

① 在执行创建信号量的系统调用时,信号量的 ID 可以被返回到一个应用指定的变量中;

② 执行获取信号量 ID 的系统调用。

其他的信号量管理系统调用一般都使用 ID 访问信号量。

使用获取信号量名字的系统调用,内核可以把名字字符串复制到应用指定的空间中。

应用还可以获取活动信号量的列表、每个信号量的其他细节信息,包括当前值、挂起类型、等待的任务数量以及第一个等待的任务等。

6.2.7　与信号量有关的资源配置问题

从理论上讲,一个应用可以拥有的信号量数目没有限制,只受制于存储信号量 ID 号的变量类型。比如,对于使用长度为 32 位的变量来存储 ID 号的嵌入式实时内核,最多可以提供 $2^{32}=4\ 294\ 967\ 294$ 个信号量。但在实际配置时需要注意:

- 每个信号量需要一个控制块,该控制块是需要一定的内存空间的;
- 每个应用实际需要使用的信号量数目是有限的。

不同的嵌入式实时内核具体实现的信号量控制块结构是不一样的,所以在配置应用的存储空间的时候,需要计算这些控制结构所占用的内存。嵌入式实时操作系统供应商可以告诉用户单个信号量(以及其他操作系统对象)的控制结构所需的空间,提供对存储空间综合需求的计算公式,或在开发工具中提供根据应用的配置自动计算空间需求的功能。

对信号量数目的配置可以通过开发环境提供的配置工具完成,开发者也可以手工修改配置文件。在操作系统(内核)的初始化过程中,初始化程序根据配置文件中的内容对信号量控制块等数据结构进行初始化。

6.3　邮箱和消息队列

6.3.1　任务间的通信方式

任务间的通信方式可以有直接通信和间接通信两种。

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 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。

本节所描述的邮箱和消息队列都属于间接通信方式,嵌入式实时操作系统也大多提供这些机制。

6.3.2 消息、邮箱和消息队列概述

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应用完成。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。

一些操作系统内核把消息机制进一步分为邮箱机制和队列机制。

邮箱仅能存放单条消息,它提供了一种低开销的机制来传送信息。每个邮箱可以保存一条大小为若干个字节的消息。发送消息方请求将消息内容放进邮箱中,接收消息方从邮箱中将消息取出。

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。消息可以放在队列的前端，也可以放在队列的后端（与 6.3.4 节讲述的普通消息和紧急消息有关）。消息队列中消息的数量可由用户自己定义。

消息机制可支持定长与可变长度两种模式的消息，消息模式在队列被创建时定义。可变长度的消息队列需要对队列中的每一条消息增加额外的存储开销。

由于邮箱机制较之消息队列更为简单，故在后面将只详细说明与消息队列有关的问题。

6.3.3　消息队列机制的主要数据结构

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，接收者从缓冲区中取出消息。消息的发送或接收可以有两种方法：一是将数据从发送任务的空间完全拷贝到接收任务的空间中；二是只传递指向数据存储空间的指针。第一种方法的弊端是效率较低，用于复制消息的时间与消息的大小有关。为了提高系统的性能，嵌入式实时内核在实现消息队列相关功能时可以不采取复制的方式进行数据传递，而采用传递指针等方法，但一定要注意确保该指针的有效性。比如在实施内存保护的情况下，这种传指针的方式就有问题。选择哪种消息传递方法将影响到消息缓冲区的结构。

不同的内核，其消息队列控制块、消息缓冲区的具体结构不同，下面所描述的只是其中一种，这里的消息缓冲区中存放的是指向消息的指针。

1. 消息队列控制块结构 Message_Queue_Control_Block

wait_queue	任务等待队列；
max_pending_count	最大未决消息数；
number_of_pending	当前未决消息数；
max_message_size	最大消息大小；
wait_discipline	任务等待消息的方式；
queue_start	消息指针数组首地址；
queue_in	消息指针数组写指针；
queue_out	消息指针数组读指针；
queue_end	消息指针数组尾地址。

2. 消息队列缓冲区结构 Message_Queue_Buffer

该结构如图 6－6 所示。

消息指针数组的每个元素是一个指向消息块的指针。该数组其实是一个环形缓冲区，如图 6－7所示。

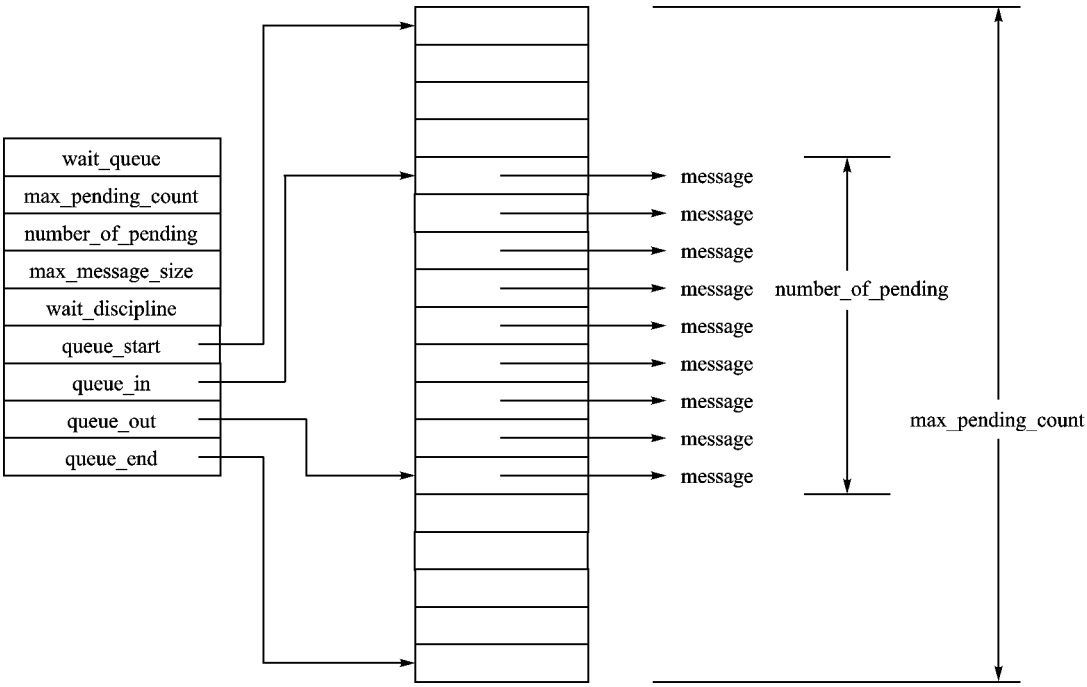


图 6－6　消息队列缓冲区结构 a

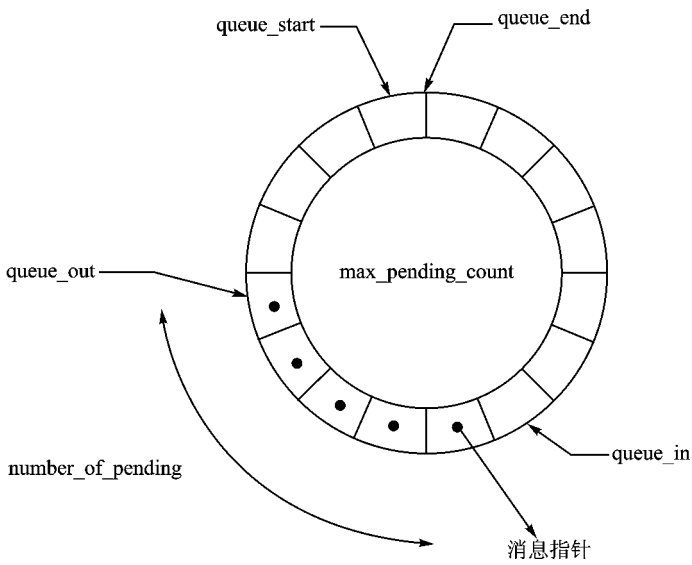


图 6－7　消息队列缓冲区结构 b

6.3.4　消息队列机制的主要功能

消息队列机制一般提供如下的主要功能和相应的应用编程接口(系统调用):

- 创建消息队列;
- 发送普通消息;
- 发送紧急消息;
- 发送广播消息;
- 接收消息;
- 删除消息队列;
- 获取有关消息队列的各种信息。

随着任务(或 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,消息队列里没有存放一条消息);
- 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。

1. 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,调用者可以指定消息的最大长度和每个消息队列中最多的消息数。调用者还可以指定消息队列的属性,比如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一般有以下两种,只能选其一。

① 任务按先进先出(FIFO)方式等待;

② 任务按优先级(PRIORITY)方式等待。

如果消息队列支持 FIFO 挂起,则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,则任务以优先级别的高低顺序为恢复执行的顺序。

创建消息队列后,系统为它分配惟一的 ID。

2. 发送消息

应用可以使用发送消息的调用将消息发送到消息队列。根据紧急程度的不同,消息通常可分为普通消息与紧急消息。如果有任务正在等待消息(即消息队列为空),则普通消息发送和紧急消息发送的执行效果是一样的。任务从等待队列中移到就绪队列中,消息被拷贝到任务提供的缓冲区中(或者由接收任务得到指向消息的指针)。

如果没有任务等待,则发送普通消息的调用将消息放在队列尾,而发送紧急消息的调用将消息放在队列头。

如果发送消息时队列已被填满,则不同的操作系统可能采取不同的处理办法:一是挂起试图向已满的消息队列中发送消息的任务;二是简单地丢弃该条消息并向调用者返回错误信息。由于中断服务程序是不能被阻塞的,因此第一种处理方法不适用于中断服务程序。

队列中的消息可以被广播。该功能类似于发送操作,不同的是所有试图从队列中接收消息的任务都将获得广播消息,这些任务获得的是相同的消息。该功能拷贝消息到各任务的消息缓冲中(或者让所有的等待任务得到指向消息的指针),并唤醒所有的等待任务。

3. 接收消息

从调用者指定的消息队列接收消息。如果此时消息队列中有消息,则将其中的第一条消息拷贝到调用者的缓冲区(或者将第一条消息指针传递给调用者),并从消息队列中删除它。如果此时消息队列中没有消息,则可能出现以下几种情况:

- 永远等待消息的到达;
- 等待消息且指定等待时限;
- 不等待,强制立即返回。

如果选择等待,则等待消息的任务按 FIFO 或优先级高低顺序排列在等待队列中。这取决于消息队列的属性,如前所述。当选择等待接收消息时,还可以进一步选择永远等待或限时等待。在限时等待的情况下,如果等待的时间超时,则系统将返回一个错误信息,而调用者可以继续执行,这样可以防止死锁。但是中断服务程序接收消息时必须选择不等待,因为中断服务程序是不能被阻塞的。

如果消息队列被应用删除,则所有等待该消息队列的任务都被返回一个错误信息,并回复到就绪状态。

4. 删除消息队列

从系统中删除调用者指定的消息队列,释放消息队列控制块及消息队列缓冲区。所有知道这个消息队列的惟一标识(比如 ID 号)的代码都可以删除它。消息队列被删除后,所有等待从这个消息队列接收消息的任务都回到就绪队列,并得到一个错误信息,表明消息队列已被删除。消息队列被删除以后,任何对这个消息队列的使用都是错误的。

5. 获取有关消息队列的各种信息

创建消息队列后,系统通常为它分配惟一的 ID,因此在创建成功后应用可直接获得其 ID。另外,操作系统也可提供专门的调用以获取消息队列 ID 号。对消息队列的使用一般通过其 ID 进行。

操作系统可提供专门的调用以获取消息队列的名字,将名字字符串复制到调用者指定的空间中。

有关消息队列的其他信息还可以包括活动消息队列的列表、队列中的消息数、消息队列属

性以及等待队列中的第一个任务等。不同的操作系统提供的功能不尽相同，提供的功能越多，应用编程越方便。

6.3.5　与消息队列有关的资源配置问题

从理论上讲，对于一个应用可以拥有的消息队列数目可以没有限制，只受制于存储消息队列 ID 号的变量类型。比如，对于使用长度为 32 位的变量来存储 ID 号的实时内核，最多可以提供 $2^{32}=4\ 294\ 967\ 294$ 个消息队列。但在实际配置时需要注意：

- 每个消息队列需要一个控制块，该控制块是需要一定的存储空间的；
- 每个消息队列缓冲区中有若干条消息，每条消息具备一定的大小；
- 每个应用实际需要使用的消息队列数目是有限的。

不同的嵌入式实时操作系统具体实现的消息队列控制块及缓冲区的结构是不一样的，所以在配置应用的存储空间的时候，需要计算这些控制结构所占用的内存。嵌入式实时操作系统供应商提供对存储空间综合需求的计算公式，或在开发工具中提供根据应用的配置自动计算空间需求的功能。

对于消息队列的数目、每个队列中最多可以容纳的消息数以及每条消息的最大长度，可以通过开发环境提供的配置工具完成配置，开发者也可以手工修改配置文件。在操作系统（内核）的初始化过程中，初始化程序根据配置文件中的内容对相关数据结构进行初始化。

6.4　事　件

6.4.1　事件机制概述

在嵌入式实时内核中，事件是指一种表明预先定义的系统事件已经发生的机制。事件机制用于任务与任务之间、任务与 ISR 之间的同步。其主要的特点是可实现一对多的同步。

一个事件就是一个标志，不具备其他信息。一个或多个事件构成一个事件集。事件集可以用一个指定长度的变量（比如一个 32 位的无符号整型变量。不同的操作系统其具体实现不一样）来表示，而每个事件由在事件集变量中的某一位来代表。

事件及事件集有以下特点：

① 事件间相互独立。

在事件集中，事件之间是按位“或”的关系，彼此互不相关。比如对于一个由 32 位变量表示的事件集，有如下的表达式：

$$\text{EVENT\_3} \mid \text{EVENT\_15} \mid \text{EVENT\_22} \mid \text{EVENT\_31} \qquad (6-1)$$

这是一个包括事件 3，15，22，31 的事件集，这些事件分别对应该 32 位事件集变量的第 3，

15,22 和 31 位(bit)。EVENT_3 是一个宏,其值为“0x00000008”,其他以此类推,比如 EVENT_15 为 0x00008000,EVENT_22 为 0x00400000,EVENT_31 为 0x80000000 等,而表示所有事件标记都置位的事件集则可用宏 ALL_EVENTS(其值为 0xFFFFFFFF)定义。

由于事件之间相互独立,因此按位“或”与按位“加”的操作是相同的。因此式(6-1)等同于下面这个表达式:

$$\text{EVENT\_3}+\text{EVENT\_15}+\text{EVENT\_22}+\text{EVENT\_31} \tag{6-2}$$

② 事件仅用于同步,不提供数据传输功能。

③ 事件无队列,即多次发送同一事件,在未经过任何处理的情况下,其效果等同于只发送一次。

能用于同步的机制有很多,那么提供事件机制的意义何在呢?当某任务要与多个任务或中断服务同步时,就需要使用事件机制。若任务需要与一组事件中的任意一个发生同步,则可称为独立型同步(逻辑“或”关系)。任务也可以等待若干事件都发生时才同步,称为关联型同步(逻辑“与”关系),如图 6-8 所示。

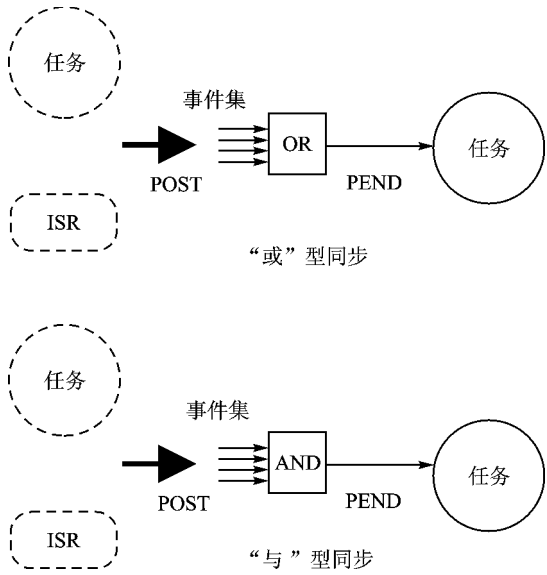


图 6-8 “或”同步和“与”同步

可以用多个事件的组合发信号给多个任务,如图 6-9 所示。事件标志(集)可以由任务或中断服务程序发出;当接收任务所需的事件(集)发生时,该任务继续执行。

以下术语是在理解事件机制时需要掌握的。

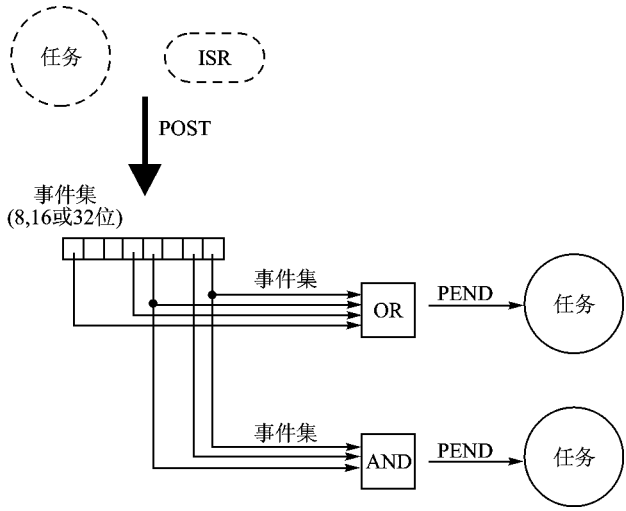


图 6-9　事件标志

(1) 发送事件集

指在一次发送过程中发往接收者(比如任务)的一个或多个事件的组合。

(2) 待处理事件集

指已被发送到一个接收者但还没有被接收(即正在等待处理)的所有事件的集合。

(3) 事件条件

指事件接收者在一次接收过程中期待接收的一个或多个事件的集合。因此每个接收者可以包含两个变量,分别用于存放待处理事件集和事件条件。对应"或"同步和"与"同步,对于事件条件的满足由两种算法决定。其一是待处理事件集只要包括事件条件中的任一事件即可满足要求;其二是待处理事件集必须包括事件条件中的全部事件方可满足要求。

6.4.2　事件机制的主要数据结构

在某些嵌入式实时内核的实现中,事件集是可以被应用动态地创建和删除的。在这种情况下,内核使用事件集控制块来管理所有创建的事件集,在系统运行时动态地分配和回收事件集控制块。但是在某些嵌入式实时内核中,已经缺省为每个任务赋予了一个事件集,因此不需要应用再去创建它们,事件集的相关参数已经成为了任务控制块(TCB)的一部分。在这种情况下,内核也不提供删除事件集的功能。

不同的内核,其事件机制的具体实现不同。下面所描述的是其中一种事件集控制块及相关数据结构的情况。

事件集控制块结构 Event_Set_Control_Block

attribute	事件集的属性;
event_set	当前事件集;

eventset_condition_queue_and　事件集“与”等待队列；

eventset_condition_queue_or　事件集“或”等待队列。

事件集的属性可以指出任务在该事件集上排队的方式是按先进先出(FIFO)还是按优先级高低(PRIORITY)的顺序。

当前事件集指示已经被置位并且未被任务接收的事件标志位。

事件集“与”和“或”等待队列的组织方式相同，它们都是数组，其长度等于事件集的位数(比如对应 32 位的事件集，该数组就有 32 个元素)，其中的每一个元素都对应着一个标志位的等待队列。

对某事件标志位有等待要求的任务被加入该标志位的等待队列中，等待多个标志位的任务将被分别加入其等待的所有标志位的等待队列中，这样从任意一个其等待的标志位都可以访问到该任务。

对每个等待事件集的任务，内核为其生成一个“任务事件集等待控制块”。其结构如下：

任务事件集等待控制块结构 **Event_Set_Task_Waited_Buddy**

task　等待任务的控制块指针；

event_set　任务当前等待的事件集；

flag_node_array　任务等待标志节点数组。

同样地，任务等待标志节点数组的长度等于事件集的位数，其中每个节点元素对应一个等待的事件标志。

任务等待的所有标志位的“等待节点”均被加入到相应标志位的等待队列中。

为了表明上述各种数据结构的关系，以一个实例说明它们之间的关联情况。假设在某系统中有 Task 1，Task 2，Task 3 和 Task 4 四个任务，其优先级由高到低，它们均以“与”条件等待在同一个事件集对象上。其中 Task 1 等待的标志为 0，3；Task 2 等待的标志为 0；Task 3 等待的标志为 2，3；Task 4 等待的标志为 2，31，则该事件集控制块中的“与”等待队列、任务事件集等待控制块的组织情况如图 6－10 所示。

任务事件集等待控制块中，“T”为任务控制块指针；“E”为任务当前等待的事件集；然后是 32 个等待节点，每个节点分别对应一个事件标志。对某标志有等待要求的所有任务的相应节点组成一个双向链表。

在本例中，Task 1 加入了 0，3 标志位的等待队列；如果按照优先级高低顺序排队，它是 0，3 标志位等待队列的第一个节点。

Task 2 加入 0 标志位的等待队列，并且位于 Task 1 之后。

Task 3 加入 2，3 标志位的等待队列，它是 2 标志位等待队列的第一个节点，但在 3 标志位等待队列上位于 Task 1 之后。

Task 4 加入 2，31 标志位的等待队列，它在 2 标志位等待队列上位于 Task 3 之后，但在 31 标志位等待队列上它是第一个节点。

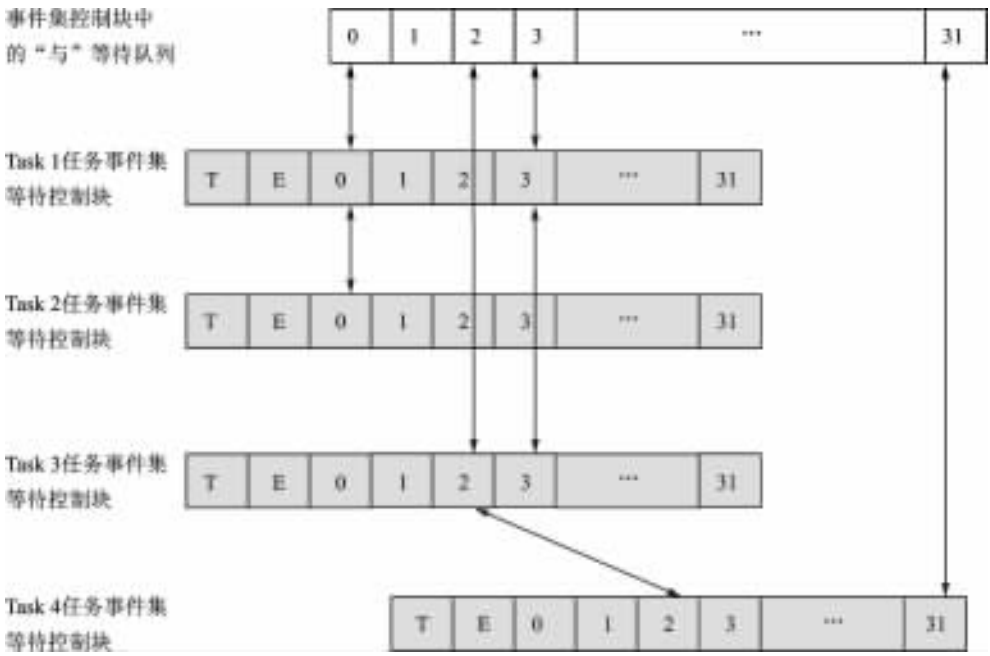


图 6-10　事件集任务等待队列的组织方式

这样，从事件集控制块开始，可以访问到任何一个等待任务，并可以进行相关的插入和删除操作。

6.4.3　事件机制的主要功能

事件机制一般提供如下主要功能和相应的应用编程接口（系统调用）：

- 创建事件集；
- 删除事件集；
- 发送事件（集）；
- 接收事件（集）；
- 获取有关事件集的各种信息。

1. 创建事件集和删除事件集

创建事件集时，内核将申请一个空闲的事件集控制块，并根据调用者指定的各项参数设置被创建事件集的属性，初始化事件集控制块中的域（当前事件集、“与”/“或”等待队列）。创建事件集的调用还将系统分配的事件集 ID 号返回给调用者，以后对该事件集的各项操作都可通过 ID 号进行。

删除事件集时，内核将被删除事件集的控制块回收到空闲控制块链中，所有正在等待接收

被删除事件集的任务会被恢复。

2. 发送事件(集)

调用者(任务或中断)构造一个事件(集),将其发往接收者(比如目标任务)。可能会出现以下几种情况之一:

- 目标任务正在等待的事件条件得到满足,任务就绪;
- 目标任务正在等待的事件条件没有得到满足,该事件(集)被按"或"操作,保存到目标任务的待处理事件集中,目标任务继续等待;
- 目标任务未等待事件(集),该事件(集)被按"或"操作,保存到目标任务的待处理事件集中。

3. 接收事件(集)

在接收事件(集)时可以有如下选项,每一类只能选择其一。

- 接收事件(集)时可等待(WAIT);
- 接收事件(集)时不等待(NO_WAIT);
- 待处理事件集必须包含事件条件中的全部事件方可满足要求(EVENT_ALL),即按照"与"条件接收事件;
- 待处理事件集只要包含事件条件中的任一事件即可满足要求(EVENT_ANY),即按照"或"条件接收事件。

该系统调用根据接收选项(EVENT_ALL 或 EVENT_ANY)检测待处理事件集是否满足所指定的事件条件,如果满足,则接收事件(集)的操作成功返回;否则,根据接收选项(NO_WAIT 或 WAIT)确定是否进入等待状态,从而可能出现如下几种情况:

- 接收者永远等待,直到事件条件被满足后成功返回。
- 接收者根据指定的时限等待,之后可能出现两种情况,其一是当系统中的其他任务(或中断处理程序)发送了该任务正在等待的事件(集)且满足接收事件条件后,该任务重又恢复到就绪状态;其二是指定的时限已到,但等待任务仍然未接收到满足事件条件的事件(集),任务恢复到就绪状态,但得到系统返回的一个出错信息。
- 接收者立即返回,得到系统返回的一个出错信息。

4. 接收(清空)待处理事件集

内核可以提供这样的功能,以方便应用。接收事件(集)时,接收任何已置位的事件标志且不等待,这样可以将待处理事件集返回到输出参数中,并将待处理事件集清空。

5. 获取待处理事件集

与接收(清空)待处理事件集的功能类似,在接收事件(集)时,可以将待处理事件集返回到输出参数中,但是保持待处理事件集不变。

6. 获取有关事件集的各种信息

这些信息可以包括活动事件集的列表、事件集的名称和 ID 等。不同的嵌入式实时内核提

供的功能不尽相同;提供的功能越多,应用编程越方便。

6.4.4　与事件机制有关的资源配置问题

对于可以自由创建事件集的系统来说,从理论上讲,一个应用可以拥有的事件集的数量没有限制,只受制于存储事件集 ID 号的变量类型。比如,对于使用长度为 32 位的变量来存储 ID 号的实时内核,最多可以提供 4 294 967 294 个事件集。

对于为每个任务配置一个事件集的系统来说,一个应用可以拥有的事件集的数量与实际的任务数相同。

不同的嵌入式实时内核具体实现的事件集控制块或其他相关数据结构是不一样的,所以在配置应用的存储空间的时候,需要计算这些控制结构所占用的内存。通常嵌入式操作系统供应商提供对内存空间综合需求的计算公式,或在开发工具中提供根据应用的配置自动计算空间需求的功能。

除了可以通过开发环境提供的配置工具完成配置外,开发者也可以手工修改配置文件。在内核初始化的过程中,初始化程序根据配置文件中的内容对相关数据结构进行初始化。

6.5　异步信号

6.5.1　异步信号机制概述

异步信号机制用于任务与任务之间、任务与 ISR 之间的异步操作,异步信号被任务(或 ISR)用来通知其他任务某个事件的出现。

异步信号标志可以依附于任务,比如在嵌入式实时内核 DeltaCORE 的实现中,每个任务可以有 32 个异步信号标志,用一个 32 - bit 无符号整型变量记录这 32 个异步信号,有效的异步信号从 SIGNAL_0 ~ SIGNAL_31。SIGNAL_0 和 SIGNAL_31 都是宏,其值分别为“0x00000000”和“0x80000000”,其他以此类推。

一个或多个异步信号标志组成了一个异步信号集。发送异步信号集指发往目标任务的一个或多个异步信号的组合;待处理异步信号集指发送给具有有效 ASR(异步信号服务例程)的目标任务并等待处理的异步信号的组合。

异步信号集中各个异步信号之间是相互独立,互不相交的,因此对异步信号标志的“加”操作与“或”操作是相同的。例如,要发送由 SIGNAL_6,SIGNAL_15 和 SIGNAL_31 构成的异步信号集,发送异步信号调用的参数可以写为 SIGNAL_6 | SIGNAL_15 | SIGNAL_31 或 SIGNAL_6 + SIGNAL_15 + SIGNAL_31。

需要处理异步信号的任务由两部分组成,一个是与异步信号无关的任务主体;另一个是 ASR。

异步信号管理允许任务定义一个异步信号服务例程(ASR)。一个 ASR 对应于一个任务,而 ISR 则与任务没有对应关系,它们是全局的。在系统运行过程中,当外部中断出现且未被系统屏蔽时,任务的执行将被中断,系统转而执行与该中断相关的服务例程。同样,当向任务发送一个异步信号时,如果该任务正在运行,则将中止其自身代码的运行,转而运行与该异步信号相关的服务例程;或者当该任务被激活时,在投入运行前执行 ASR。因此,异步信号机制也可以称为软中断机制,异步信号又被称为软中断信号。发送异步信号给任务,对接收任务的当前执行状态没有任何影响。

异步信号可以被使能,也可以被禁止。被使能的异步信号在发送的时候才会执行对应的处理程序。

6.5.2 异步信号机制与中断机制的比较

异步信号机制与中断机制非常相似,但各自又有特点,并且具有其内在的许多根本性的不同。以下是异步信号机制与中断机制的比较。

1. 相同点

(1) 具有中断性

对中断的处理和对异步信号的处理都要先暂时地中断当前任务的运行。

(2) 有相应的服务程序

根据中断向量,有一段与中断信号对应的服务程序,称为 ISR(Interrupt Service Routine)。

根据异步信号的编号,有一段与之对应的服务程序,称为 ASR(Asynchronious Service Routine)。

(3) 可以屏蔽响应

外部硬件中断可以通过相应的寄存器操作被屏蔽。

任务也可屏蔽对异步信号的响应。

2. 不同点

(1) 实质不同

虽然异步信号机制又称为软中断机制,但这只是强调异步信号也具有中断性,它们有不同的实质。中断由硬件或者特定的指令产生,不受任务调度的控制。异步信号由系统调用(使用发送异步信号功能)产生,受到任务调度的控制。

(2) 处理时机(或响应时间)不同

中断触发后,硬件根据中断向量找到相应的服务程序执行。为了缩短中断响应时间,通常采用中断服务程序与任务相结合的方式来处理中断。中断服务程序仅完成必要的处理,在退出中断服务程序之前会进行重调度,所以中断结束后开始运行的任务不一定是先前被中断的任务。

异步信号通过发送异步信号的系统调用触发,但是系统不一定马上开始对它的处理。

ASR 可以看成是任务切换后所做的任务扩展(post switch extension)。如果发送异步信号的是中断服务程序,则接收异步信号的任务可以是当前任务,也可以是另一个任务。由于异步信号的发送不会导致任务状态的变化(因为任务不会去“等待”一个异步信号),因而系统在中断退出后返回到先前被中断的任务,并开始执行该任务的 ASR(如果中断向它发送了异步信号);如果发送异步信号的是任务,则 ASR 要等到接收任务被调度,完成上下文切换后才能开始执行,之后再执行任务自身的代码。任务也可以给自己发送异步信号,在这种情况下,其 ASR 将马上执行。

(3) 执行的环境不同

一般地,ISR 在独立的上下文中运行,操作系统为之提供专门的堆栈空间。ASR 在相关任务的上下文中运行,所以 ASR 也是任务的一个组成部分。

6.5.3　异步信号机制与事件机制的比较

同样是标志着某个事件的发生,事件机制的使用是同步的,而异步信号机制是异步的。意即,对一个任务来说,什么时候会接收到事件是已知的,因为接收事件的功能是它自己在运行过程中调用的。但是,任务不能够预知在什么时候会收到一个异步信号,并且一旦被其他任务或中断服务程序发送了异步信号,在允许响应的情况下,它会中断正在运行的代码转去执行异步信号处理程序。

6.5.4　异步信号机制的主要数据结构

不同的内核,其异步信号机制的具体实现不同。如前所述,某些嵌入式实时内核为每个任务赋予了一个异步信号集,因此不需要应用再去创建它们,异步信号集的相关参数已经成为了任务控制块(TCB)的一部分。在这种情况下,内核也不提供删除异步信号集的功能。

不管怎样,内核都必须定义相关的数据结构。下面描述了一种异步信号控制结构的情况。

异步信号控制结构 **Asynchronous_Signal_Control_Block**

enabled	是否使能对异步信号的响应;
handler	处理例程;
attribute_set	ASR 的执行属性;
signals_posted	使能响应时,已发送但尚未处理的信号;
signals_pending	屏蔽响应时,已发送但尚未处理的信号;
nest_level	ASR 中异步信号的嵌套层数。

ASR 有自己独立的执行属性,由于 ASR 可以看成是任务切换后的一个扩展,是任务的一个组成部分,因此其属性集可以与任务的属性集相同。例如,它可以是下列几组相互独立的属性元素值相“或”的组合。

(1) 是否允许任务在执行 ASR 过程中被抢占

ASR 是在任务的环境(虽然 ASR 有自己独立的执行属性)中执行的,如果在执行 ASR 的过程中有一个优先级更高的任务就绪,则执行 ASR 的任务能否被抢占由下面的属性决定。

- PREEMPT ——允许任务在执行 ASR 时被更高优先级的任务抢占;
- NO_PREEMPT ——不允许任务在执行 ASR 时被更高优先级的任务抢占。

(2) 是否允许时间片切换

在执行 ASR 的过程中,如果其所属任务的时间片用完,并且还有相同优先级的任务处于就绪状态,这时能否发生任务切换由下面的属性决定。

- TIMESLICE ——若在执行 ASR 过程中时间片用完,则允许任务切换;
- NO_TIMESLICE ——若在执行 ASR 过程中时间片用完,则不允许任务切换,直到 ASR 运行完才切换。

(3) 是否支持 ASR 嵌套

在执行 ASR 过程中可能会有新的异步信号到达,比如 ASR 给当前任务发送异步信号,或者在 ASR 执行过程中响应了中断而中断服务程序又给当前任务发送了异步信号。如果支持 ASR 嵌套处理,则可以认为新的异步信号的优先级高于当前正被处理的异步信号,前者可以先得到处理而中断当前 ASR 的执行。在这种情况下,必须考虑异步信号处理例程的可重入性。如果禁止 ASR 嵌套,则新发送给任务的异步信号将被放入待处理异步信号集中,等当前 ASR 完成后再处理。

- ASR ——允许 ASR 嵌套处理;
- NO_ASR ——不允许 ASR 嵌套处理。

(4) 是否允许在执行 ASR 过程中响应中断

- INTERRUPT_LEVEL(0) ——打开中断;
- INTERRUPT_LEVEL(1) ——关闭中断。

如果在 ASR 的执行过程中不响应外部中断,则可能会影响系统的响应能力和实时性,尤其是 ASR 比较复杂、耗时较多时。

在打开中断的情况下,有些内核还允许定义具体开放的中断级别。这与内核所采用的控制方法有关。通过设置 CPU 标志寄存器中的中断屏蔽位只能实现对开/关中断的控制,而通过设置目标平台上的中断控制器可以达到对具体中断级别的控制。

ASR 的执行属性由上述各项属性元素相“或”而构成。各属性元素是独立的、互不相交的,因此“加”操作与“或”操作是相同的。例如在建立 ASR(安装异步信号处理例程)时,让它不允许被抢占、时间片切换、嵌套处理和打开中断,则属性值应写为

NO_PREEMPT|NO_TIMESLICE|NO_ASR|INTERRUPT_LEVEL(0)

或　　NO_PREEMPT+NO_TIMESLICE+NO_ASR+INTERRUPT_LEVEL(0)

这里需要注意 ASR 的执行属性和任务属性的关系。虽然 ASR 的属性元素与任务的相同，但一个具体的 ASR 的属性可以与它所对应的任务的属性不同。例如，任务允许被抢占，而任务的 ASR 可以不允许被抢占。

6.5.5　异步信号机制的主要功能

异步信号机制提供的主要功能和相应的应用编程接口（系统调用）包括：

- 安装异步信号处理例程；
- 发送异步信号到任务。

1. 安装异步信号处理例程

操作系统提供此功能调用为任务安装一个异步信号处理例程（ASR），仅当任务已建立了 ASR，才允许向该任务发送异步信号，否则发送的异步信号无效。当任务的 ASR 无效时，发送到任务的异步信号将被丢弃。调用者需指定 ASR 的入口地址和执行属性。

异步信号处理例程一般为如下的形式：

```
void  handler(signal_set)
 {
  switch(signal_set)
  {
    CASE SIGNAL_1:
    动作 1;
    break;
    CASE SIGNAL_2:
    动作 2;
    break;
    ……
  }
 }
```

其中的 signal_set 参数为任务接收到的异步信号集。

2. 发送异步信号到任务

任务或 ISR 可以调用该功能发送异步信号到目标任务，发送者指定目标任务与要发送的异步信号（集）。发送异步信号给任务对接收任务的执行状态没有任何影响。在目标任务已经安装了异步信号处理例程的情况下，如果目标任务是其他任务，那么发送给它的异步信号就会等下一次该任务占有处理器时再由相应的 ASR 处理，任务获得处理器后，将首先执行 ASR。如果当前运行的任务发送异步信号给自己或收到来自中断的异步信号，则在允许 ASR 处理的前提下，它的 ASR 会立即执行。

与事件相同，发送给一个任务的相同异步信号不排队。也就是说，多次向一个任务发送同样的异步信号（中间没有对该异步信号进行过任何处理），与发送一次的效果是一样的。

思考题

6.1　嵌入式操作系统内核提供的同步、互斥与通信机制主要有哪些？请以一款具体的嵌入式操作系统为例，分析它所提供的同步、互斥与通信机制。

6.2　在创建互斥信号量、二值信号量和计数信号量的时候有哪些异同点？

6.3　什么是“删除安全”问题？在什么情况下需要对任务实施删除安全保护？

6.4　请利用一款具体的嵌入式操作系统提供的同步与互斥机制，解决本章所提到的有界缓冲问题。

6.5　任务等待消息的方式有哪几种？当任务试图接收消息时，什么情况下系统可能发生任务重新调度？

6.6　请利用一款具体的嵌入式操作系统提供的通信机制，实现如图 6－11 所示的任务之间的全双工通信。

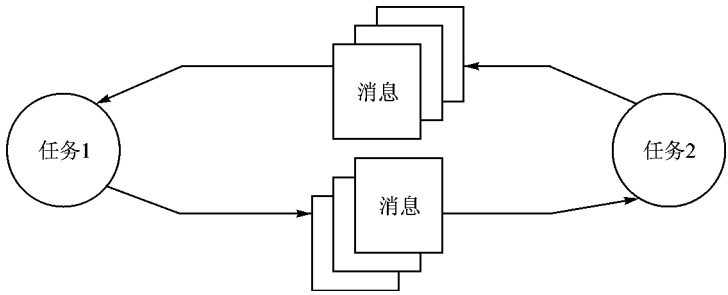


图 6－11　任务之间的全双工通信

6.7　请利用事件机制提供的“与”和“或”同步功能，分别设计一个应用的例子，并说明与其他同步机制相比，这两种同步方式给多任务应用设计带来的便利之处。

6.8　请列表说明异步信号机制与中断的异同点。

6.9　怎样利用操作系统提供的同步和通信机制达到任务与中断的协同工作？请使用一款具体的嵌入式操作系统编写多任务应用程序，并实现任务与中断之间的单向同步。

第 7 章　中断和时间管理

7.1　中断管理

7.1.1　概　述

实时系统通常都需要处理来自外部环境的事件，如按键、视频输出的同步以及通信设备的数据收发等。这些事件都要求能够在特定的时间范围内得到及时处理。比如，当键盘上的键被按下后，系统应该尽快把相应的字符显示出来，否则会给用户造成系统没有响应的情形。中断即为解决该类问题的有效机制。

从发展过程来看，中断(interrupt)最初被用来替换 I/O 操作的轮询处理方式，以提高 I/O 处理的效率。随后，中断又包含了自陷(trap，也称为内部中断或是软件中断)的功能。后来，中断的概念得到进一步扩大，被定义为导致程序正常执行流程发生改变的事件(不包括程序的分支情况)。可把概念被扩大的中断称为广义中断。在实际应用中，广义的中断通常被分为中断、自陷和异常(exception)等类别。

- 中断　由于 CPU 外部的原因而改变程序执行流程的过程，属于异步事件，又称为硬件中断。自陷和异常则为同步事件。
- 自陷　表示通过处理器所拥有的软件指令，可预期地使处理器正在执行的程序流程发生变化，以执行特定的程序，如 Motorola 68000 系列中的 Trap 指令和 ARM 中的 SWI 指令和 Intel 80x86 中的 INT 指令。自陷是显式的事件，需要无条件地执行。
- 异常　CPU 自动产生的自陷，以处理异常事件，如被 0 除、执行非法指令和内存保护故障等。异常没有对应的处理器指令；当异常事件发生时，处理器也需要无条件地挂起当前运行的程序，执行特定的处理程序。

使用中断的目的在于提高系统效率，使得系统在进行一些 I/O 操作时，CPU 仍然能够继续执行正常的程序流程。而在轮询系统(polling system)中，CPU 需要持续不断地探测 I/O 设备，以获取 I/O 操作是否已经完成的信息。

在中断驱动的系统中，CPU 继续其正常执行情况；当 I/O 设备需要服务时，I/O 设备会通过中断的方式来通知 CPU。轮询系统则由于在设备未准备好时需要不断对设备进行轮询而耗费 CPU 的指令周期。这可以通过串口的情况来进行说明。假定执行通过串口获取字符的通信程序，当有字符到达时，会产生一个中断，启动通信程序对得到的字符进行缓存，然后退出

中断。与此同时,由于处理串口中断的时间非常短,另外的程序可以在没有显著性能降低的情况下得到执行。如果用轮询方式来处理上述情况,为了查看是否有字符到达,CPU 就需要持续不断地轮询串口芯片。在这种方式下,即使字符的到达率非常低,CPU 也需要花费所有的时间来获得输入字符,而没有时间来进行其他程序的处理。当然,在没有中断机制的情况下,也可以通过后台处理的方式来获得字符。但这可能需要其他程序每隔几个 ms 就查询串口芯片一次,以确保数据不会丢失。这使得这些程序的实现比较困难和繁琐。

对于实时系统来说,中断通常都是必不可少的机制,以确保具有时间关键特性的功能部分能够得到及时执行。实时内核大都提供了管理中断的机制,该机制方便了中断处理程序的开发,提高了中断处理的可靠性,并使中断处理程序与任务有机地结合起来。

7.1.2 中断的分类

根据硬件中断是否可以被屏蔽,中断分为可屏蔽中断和不可屏蔽中断。由于中断的发生是异步的,故程序的正常执行流程随时有可能被中断服务程序打断。如果程序正在进行某些重要运算,则中断服务程序的插入将有可能改变某些寄存器的数据,造成程序的运行发生错误。因此,在程序某段代码的运行过程中就可能需要屏蔽中断,通过设置屏蔽标志对中断暂时不作响应。能够被屏蔽掉的中断称为可屏蔽中断。对于可屏蔽中断,外部设备的中断请求信号一般需要先通过 CPU 外部的中断控制器,再与 CPU 相应的引脚相连。可编程中断控制器可以通过软件进行控制,以禁止或是允许中断。而另一类中断是在任何时候都不可屏蔽的,称为不可屏蔽中断。一个比较典型的例子是掉电中断,当发生掉电时,无论程序正在进行什么样的运算,它都肯定无法正常运行下去。这种情况下,急需进行的是一些掉电保护的操作。对这类中断,应随时进行响应。

从中断源来说,中断又可分为硬件中断和软件中断。硬件中断是由于 CPU 外部的设备所产生的中断,属于异步事件,可能在程序执行的任何位置发生,发生中断的时间通常是不确定的。软件中断也称为同步中断或是自陷,通过处理器的软件指令来实现,产生中断的时机是预知的,可根据需要在程序中进行设定。软件中断的处理程序以同步的方式进行执行,其处理方式同硬件中断处理程序类似。软件中断是一种非常重要的机制,系统可通过该机制在用户模式执行特权模式下的操作。软件中断也是软件调试的一个重要手段,如 Intel 80x86 中的 INT 3,使指令进行单步执行,调试器可以用它来形成观察点,并查看随程序执行而动态变化的事件情况。

从中断信号的产生来看,根据中断触发的方式,中断分为边缘触发中断和电平触发中断。在边缘触发方式中,中断线从低变到高或是从高变到低时,中断信号就被发送出去,并只有在下一次的从低变到高或是从高变到低时才会再度触发中断。由于该事件发生的时间非常短,有可能出现中断控制器丢失中断的情况,并且,如果有多个设备连接到同一个中断线,即使只有一个设备产生了中断信号,也必须调用中断线对应的所有中断服务程序来进行匹配,否则会

出现中断的软件丢失的情况。对于电平触发方式，在硬件中断线的电平发生变化时产生中断信号，并且中断信号的有效性将持续保持下去，直到中断信号被清除。这种方式能够减少中断信号传送丢失的情况，且能通过更有效的方式来服务中断，每个为该中断服务后的 ISR 都要向外围设备进行确认，然后取消该设备对中断线的操作。因此，当中断线的最后一个设备都得到中断服务后，中断线的电平就会发生变化，不用对连接到同一个硬件中断线的所有中断服务程序进行尝试。

根据中断服务程序的调用方式，可把中断分为向量中断、直接中断和间接中断。向量中断是通过中断向量来调用中断服务程序的方式。除向量中断外，还存在中断服务程序调用的直接中断和间接中断方式。在直接中断调用方式中，中断对应的中断服务程序的入口地址是一个固定值；当中断发生的时候，程序执行流程将直接跳转到中断服务程序的入口地址，执行中断服务程序。对于间接中断，中断服务程序的入口地址由寄存器提供。

目前，系统大都采用向量中断的处理方式。中断硬件设备的硬件中断线（也称为中断请求 IRQ）被中断控制器汇集成中断向量（interrupt vector），每个中断向量对应一个中断服务程序 ISR（Interrupt Service Routine），用来存放中断服务程序的入口地址或是中断服务程序的第一条指令。系统中通常包含多个中断向量，存放这些中断向量对应中断服务程序入口地址的内存区域被称为中断向量表。

在 Intel 80X86 处理器中，中断向量表包含 256 个入口，每个中断向量需要 4 个字节（存放中断服务程序的首址）。ARM 的中断向量表开始于内存地址 0x00000000 或是 0xFFFF0000 处。也有些处理器采用了非向量的中断处理方式，当中断发生的时候，相应的控制就直接传送到一个处理程序，由该处理程序完成中断相关的操作。

大多数 CPU 核心有两种中断输入：

① 提供给可屏蔽中断（如 80X86 的 INTR）；

② 提供给非屏蔽性中断（如 80X86 的 NMI）。

外围设备使用可屏蔽中断输入，对于大多数系统来说，仅有一个可屏蔽中断的输入不足以应付丰富的中断资源，如计数器、DMAs 和通信端口。由于可屏蔽的中断源很多，并需要对其管理，如区分是哪个中断源发出的中断信号、哪个中断源最优先及怎样处理多级中断嵌套等问题。为此，可使用中断控制器，以对多个可屏蔽中断源进行管理，使 CPU 核心能和更多的中断资源相联系。

中断控制器能够对中断进行排队，避免中断信号的丢失，对不同的中断进行优先级配置，使高优先级中断能够中断低优先级中断，满足系统中具有更高时间约束特性功能的需要。在基于 X86 的架构中，8259 是一个非常通用的中断控制器芯片，称为 PIC（Programmable Interrupt Controller）。每个 PIC 只能够处理 8 个中断；为支持更多数量的中断，需要组织成菊花链（daisy chain）的方式，把一个 PIC 的输出连接到另一个 PIC 的输入上，如图 7－1 所示。

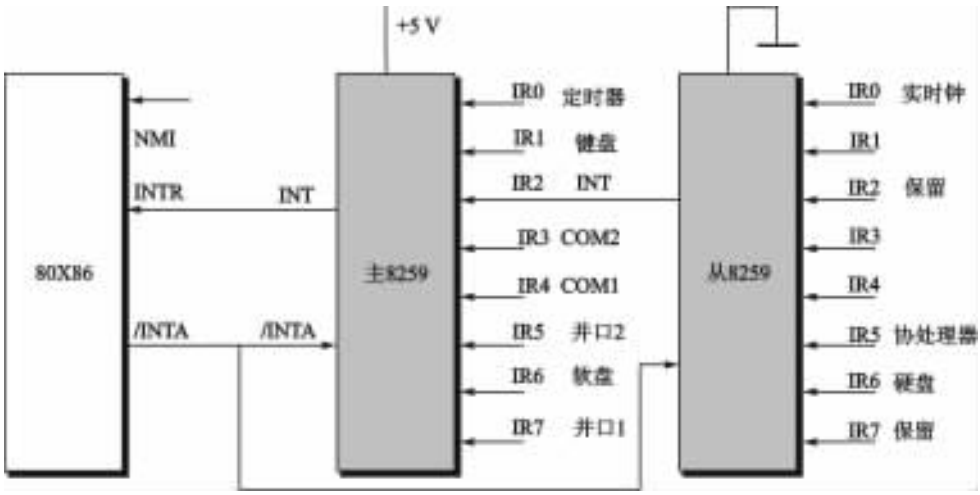


图 7-1　可编程中断控制器 8259A

7.1.3　中断处理的过程

中断处理的全过程分为中断检测、中断响应和中断处理三个阶段。

中断检测在每条指令结束时进行，检测是否有中断请求或是否满足异常条件。为满足中断处理的需要，在指令周期中使用了中断周期，如图 7-2 所示。在中断周期中，处理器检查是否有中断发生，即是否出现中断信号。如果没有中断信号，则处理器继续运行，并通过取指周期取当前程序的下一条指令；如果有中断信号，将进入中断响应，对中断进行处理。

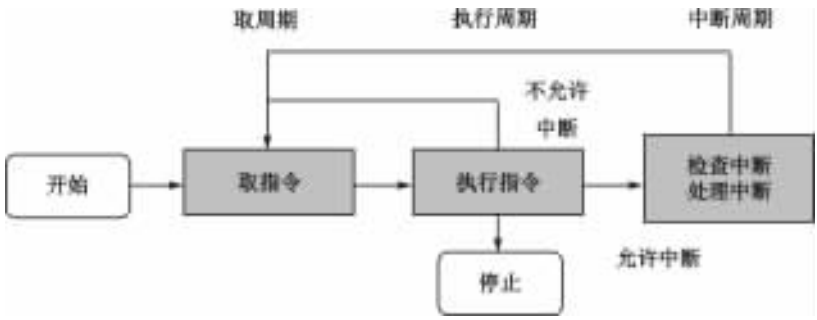


图 7-2　中断和指令周期

中断响应是由处理器内部硬件完成的中断序列，而不是由程序执行的。在 Intel 80X86 中，中断响应过程的操作如下：

- 对可屏蔽中断，从 8259 中断控制器芯片读取中断向量号；

- 将标志寄存器 EFLAG,CS 和 IP 压栈;
- 对于硬件中断,复位标志寄存器中的 IF 和 TF 位,禁止可屏蔽外部中断和单步异常;
- 根据中断向量号,查找中断向量表,根据中断服务程序的首址转移到中断服务程序执行。

中断处理即执行中断服务程序。中断服务程序用来处理自陷、异常或是中断。尽管导致自陷、异常和中断的事件不同,但大都具有相同的中断服务程序结构。中断服务程序的主要内容如下:

- 保存中断服务程序将要使用的所有寄存器的内容,以便于在退出中断服务程序之前进行恢复;
- 如果中断向量被多个设备所共享,为了确定产生该中断信号的设备,需要轮询这些设备的中断状态寄存器;
- 获取中断相关的其他信息;
- 对中断进行具体的处理;
- 恢复保存的上下文;
- 执行中断返回指令,使 CPU 的控制返回到被中断的程序继续执行。

上面描述的是对一个中断进行处理的情况。如果对一个中断的处理还没有完成,又发生了另外一个中断,则称系统中发生了多个中断。对于多个中断,通常可以采用以下两种方法进行处理。

① 在处理一个中断的时候,禁止再发生中断。这种处理方法又称为非嵌套的中断处理方式。在非嵌套的中断处理方式中,处理中断的时候,将屏蔽所有其他的中断请求。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,并由处理器检查是否还有中断。非嵌套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,使高优先级中断不能得到及时的处理,甚至导致中断丢失。只有在当前中断服务程序退出后,其他中断的中断服务程序才可能得到执行。

非嵌套中断处理的情况如图 7-3 所示。

② 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。这种处理方法又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比当前中断优先级低或是与当前中断优先级相同的中断,在完成必要的上下文保存后即使能中断。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,然后调用高优先级中断的服务程序。当高优先级中断的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。

嵌套中断处理的情况如图 7-4 所示。

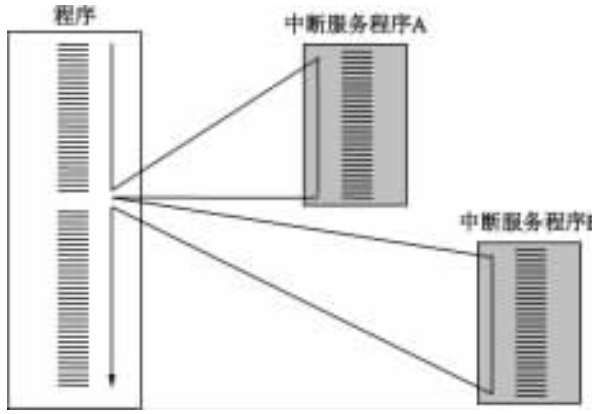


图 7-3　非嵌套中断顺序处理

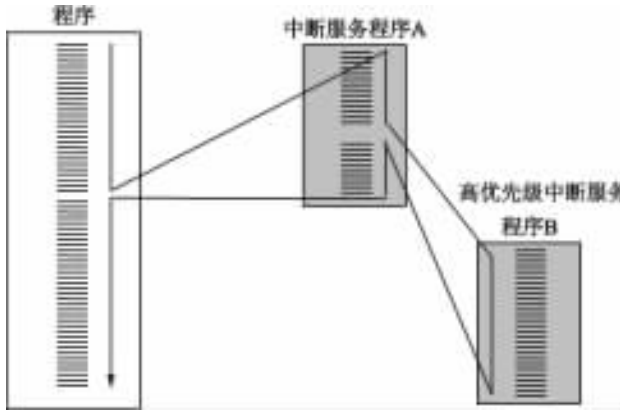


图 7-4　嵌套中断处理

7.1.4　实时内核的中断管理

在实时多任务系统中,中断服务程序通常包括三个方面的内容:

- 中断前导　保存中断现场,进入中断处理;
- 用户中断服务程序　完成对中断的具体处理;
- 中断后续　恢复中断现场,退出中断处理。

在实时内核中,中断前导和中断后续通常由内核的中断接管程序来实现。硬件中断发生后,中断接管程序获得控制权,先由中断接管程序进行处理,然后才将控制权交给相应的用户中断服务程序。用户中断服务程序执行完成后,又回到中断接管程序。基于中断接管程序的

中断处理情况如图 7－5 所示。

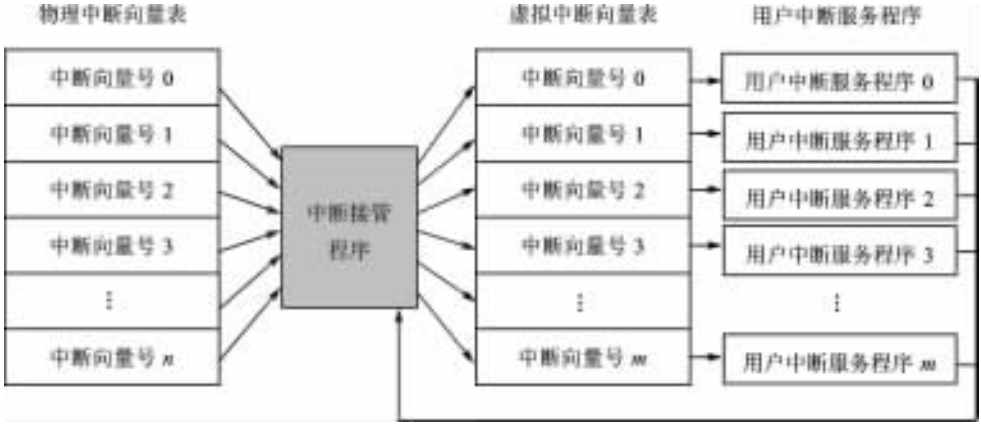


图 7－5　基于中断接管程序的中断管理

中断接管程序负责中断处理的前导和后续部分的内容。用户中断服务程序被组织为一个表，称为虚拟中断向量表。

中断处理前导用来保存必要的寄存器，并根据情况在中断栈或是任务栈中设置堆栈的起始位置，然后调用用户中断服务程序。中断处理后续则用来实现中断返回前需要处理的工作，主要包括恢复寄存器和堆栈，并从中断服务程序返回到被中断的程序。另外，如果需要在用户中断服务程序中使用关于浮点运算的操作，中断前导和中断后续中还需要分别对浮点上下文进行保存和恢复。

如果中断处理导致系统中有比被中断任务具有更高优先级的就绪任务出现时，需要把高优先级任务放入就绪队列中，把被中断的任务从执行状态转变为就绪状态，并在完成用户中断服务程序后，在中断接管程序的中断后续处理中激活重调度程序，使高优先级任务能在中断处理工作完成后得到调度执行。

在允许中断嵌套的情况下，在执行中断服务程序的过程中，如果出现高优先级的中断，当前中断服务程序的执行将被打断，以执行高优先级中断的中断服务程序。当高优先级中断的处理完成后，被打断的中断服务程序才又得到继续执行。发生中断嵌套时，如果需要进行任务调度，任务的调度将延迟到最外层中断处理结束时才能发生。虽然可能在最外层中断被继续处理之前就存在高优先级任务就绪，但中断管理只是把就绪任务放置到就绪队列，不会发生任务调度；只有当最外层中断处理结束时才会进行任务调度，以确保中断能够及时地完成处理工作。

通常，中断服务程序使用被中断任务的任务栈空间。但在允许中断嵌套处理的情况下，如果中断嵌套层次过多，中断服务程序所占用的任务的栈空间可能比较大，将导致任务栈溢出。为此，可使用专门的中断栈来满足中断服务程序的需要，降低任务栈空间使用的不确定性。可

在系统中开辟一个单独的中断栈，为所有中断服务程序所共享。中断栈必须拥有足够的空间，即使在最坏中断嵌套的情况下，中断栈也不能溢出。如果实时内核没有提供单独的中断栈，就需要为任务栈留出足够的空间，不但要考虑通常的函数嵌套调用，还需要满足中断嵌套的需要。使用单独的中断栈还能有效降低整个系统对栈空间的需求，否则需要为每个任务栈都预留处理中断的栈空间。

中断栈需要在系统初始化的时候进行设置。图 7－6 为中断向量表起始位置为 0 的情况下的内存布局情况，其中栈的增长方向为高地址端到低地址端。对于图 7－6(a)，中断栈的溢出可能导致中断向量表被破坏；对于图 7－6(b)，中断栈不会影响中断向量表。

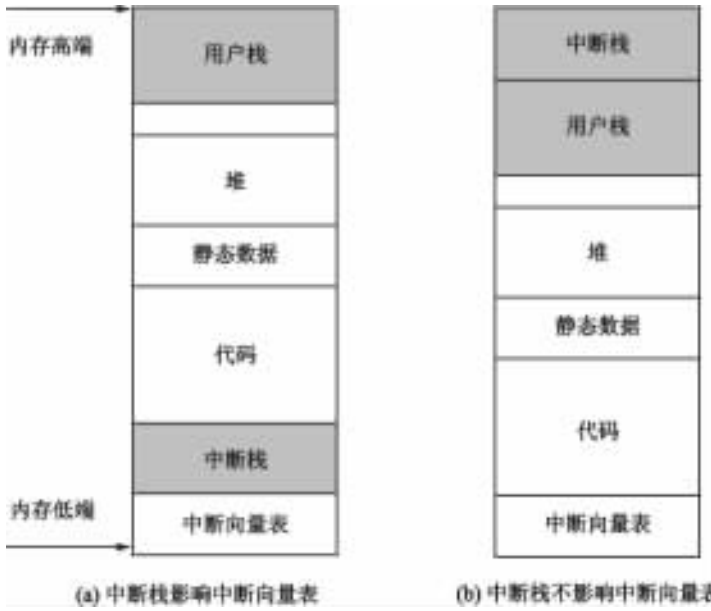


图 7－6　中断栈在内存中的布局情况

基于中断接管机制的中断管理方式对中断的处理存在着一定的延迟，不能满足某些关键事件的处理或是系统故障的响应。对这类事件应该进行最高优先级的零延迟处理。因此，实时内核还提供对高优先级中断的预留机制，这些中断的处理由用户中断服务程序独立完成，不经过中断接管程序的处理。

另外，实时内核通常还提供如下中断管理功能：

- 挂接中断服务程序　把一个函数(用户中断服务程序)同一个虚拟中断向量表中的中断向量联系在一起。当中断向量对应中断发生的时候，被挂接的用户中断服务程序就会被调用执行。
- 获得中断服务程序入口地址　根据中断向量，获得挂接在该中断向量上的中断服务程

序的入口地址。

- 获取中断嵌套层次　在允许中断嵌套的处理中，获取当前的中断嵌套层次信息。
- 开中断　使能中断。
- 关中断　屏蔽中断。

7.1.5　用户中断服务程序

当中断线上发生中断的时候，对应中断向量中注册的中断服务程序就会被调用执行。中断服务程序的注册即以中断号为索引，把处理中断的函数的地址放置到中断向量的地址表中。中断服务程序的启动完全由 CPU 来负责，不需要操作系统的处理。

如果处理器或实时内核允许中断嵌套(interrupt nesting)，则中断服务程序将可能被另外的中断服务程序所抢占。中断嵌套将使代码更加复杂，要求中断服务程序是可重入的。

许多系统允许不同的外围设备使用相同的硬件中断，相应的中断服务程序在执行过程中应能识别是哪个设备产生了中断。可通过两种方式来实现这个功能。

① 查看共享该中断的各个设备的状态寄存器；

② 执行注册在该中断号上的所有中断服务程序。

在多个中断服务程序使用同一个中断号的情况下，内核可以把属于自己的中断服务程序注册在硬件中断上，该内核中断服务程序对应用程序注册的所有中断服务程序进行逐一激活。这意味着应用程序注册的中断服务程序将在硬件中断服务程序执行后，在所有任务执行之前得到执行。在实时内核中，通常只允许一个中断号使用一个中断服务程序，否则其他中断服务程序的时间确定性就得不到保障。

由于中断服务程序中通常都对中断进行了屏蔽，要求中断服务程序应该尽可能比较短，保证其他中断和系统中的任务能够得到及时处理。为此，中断服务程序通常都只处理一些必要的操作，其他操作则通过任务的方式来进行。通常，中断服务程序只是进行与外围设备相关的数据的读/写操作，并在需要的情况下向外围设备发送确认信息，然后唤醒另外的任务进行进一步的处理。比如，用于网卡的中断服务程序大都只是传送或是接受原始的包数据，对数据的解释则由另外的任务来进行。用来配合中断服务程序的另外的任务通常被称为 DSR(Deferred Service Routine)。由 ISR 和 DSR 组合的中断处理方式如图 7－7 所示。

在图 7－7 中，dataReceiveISR 为中断服务程序，用来接收数据，但不进行数据处理。处理数据的工作由一个名为 dsrTask 的任务来进行。因此，dataReceiveISR 接收到数据后，通过一个信号唤醒处理该数据的任务 dsrTask，由 dsrTask 进行数据处理工作。

在中断服务程序中可以使用实时内核提供的应用编程接口，但一般只能使用不会导致调用程序可能出现阻塞情况的编程接口，如可以进行挂起任务、唤醒任务和发送消息等操作，但不要使用分配内存、获得信号量等可能导致中断服务程序的执行流程被阻塞的操作。这主要是由于对中断的处理不受任务调度程序的控制，并优先于任务的处理。如果中断出现被阻塞

```
/* Uses to handle data from dataReceiveISR */
dsrTask()
{
        while(1)
        {
                wait_for_signal_from_isr();
                process_data_of_ISR();
        }
}
/* Uses to receive data by interrupt */
dataReceiveISR()
{
        …
        get_data_from _device();
        send_signal_to_wakeup_dsrTask();
        …
}
```

图 7－7　ISR 与 DSR 相结合的中断处理方式

的情况，将导致中断不能被及时处理，其余工作也就无法按时继续进行，将严重影响整个系统的确定性。

内存分配和内存释放过程中通常都要使用信号量，以实现对维护内存使用情况的全局数据结构的保护。因此，中断服务程序不能进行这类操作，也不能使用包含了这些操作的编程接口。这通常也意味着中断服务程序不能使用关于对象创建和删除方面（如任务创建与任务删除）的操作。

在实时系统中，完整的中断服务程序通常由实时内核和用户共同提供。实时内核实现关于中断服务程序的公共部分的内容，如保存寄存器和恢复寄存器等；中断服务程序中由用户提供的内容通常被称为是用户中断服务程序，实现对特定中断内容的处理。因此，如果要在用户中断服务程序中使用关于浮点处理方面的内容，就需要清楚实时内核是否对浮点上下文进行了保护。由于大多数中断的处理都不涉及浮点内容，故实时内核的中断管理中一般没有对浮点上下文进行处理。如果用户中断服务程序需要使用浮点操作，就需要在用户中断服务程序中实现对浮点上下文的保护与恢复，以确保任务的浮点上下文不会被破坏。

中断服务程序还需要与系统中的任务进行通信。从中断服务程序到任务的通信机制主要包括以下内容：

● 共享内存　中断服务程序与任务共享变量、缓冲区，实现中断服务程序与任务之间的

通信。

- 信号量　中断服务程序可以释放任务正在等待的信号量。
- 消息队列　中断服务程序可以把消息发送给正在等待该消息的任务。由于中断服务程序不能被阻塞，如果消息队列已满，则应该丢弃所发送的消息，而不能等待消息队列有空间来存放该消息。
- 管道　中断服务程序可以把消息写到任务可以进行消息读取的管道中。由于中断服务程序不能被阻塞，如果管道已满，则应该丢弃准备写入的消息。
- 异步信号　中断服务程序可以向任务发送异步信号，使任务对应的异步信号处理程序能够得到执行。

7.2　时间管理

在实时系统中，时钟具有非常重要的作用。通过时钟，应用和内核能够查询当前时间、定时地完成各项工作、报警、有限等待和睡眠等，是处理具有时间约束特性应用必不可少的内容。因此，实时内核都需要提供对时钟进行管理的机制。

时间管理一般具有以下功能：

- 维持日历时间；
- 任务有限等待的计时；
- 软定时器的定时管理；
- 维持系统时间片轮转调度。

7.2.1　硬件时钟设备

大多数嵌入式系统有两种时钟源，分别为实时时钟 RTC(Real Time Clock)和定时器/计数器。

实时时钟一般是靠电池供电的，即使系统断电，也可以维持日期和时间。图 7－8 所示为 Samsung公司 44B0X(ARM7)芯片中的 RTC 部分。

图 7－8 所示的 RTC 具有以下特点：

- 在系统没有上电的情况下，可由后备电池供电。
- 可以通过 ARM 的 STRB/LDRB 操作获取 RTC 以二进制编码的十进制数据格式，向 CPU 提供 8 bits 数据。数据包含秒、分、小时、日、月和年等内容。
- 使用一个外部的 32.768 kHz 晶振。
- 包括一个闰年产生器。
- 提供报警中断或是从掉电模式中唤醒的报警功能。
- 能够避免 2000 年问题(即千年虫问题)。

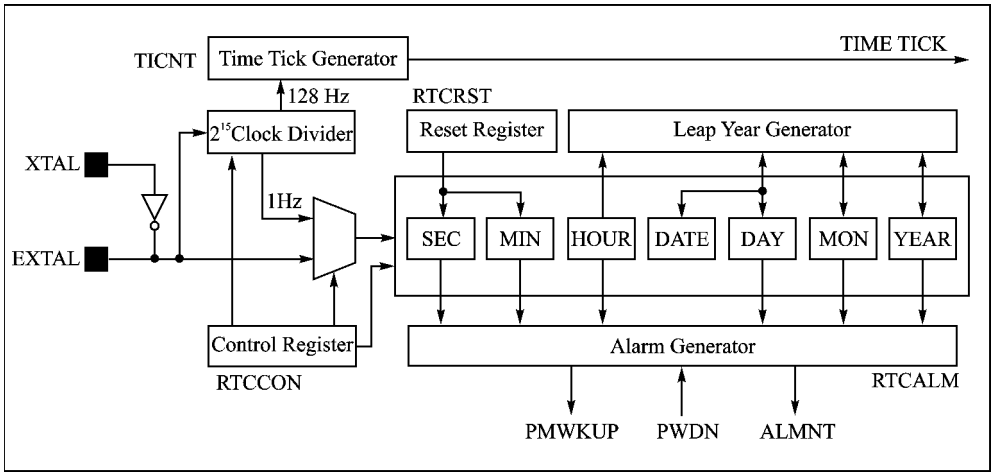


图 7-8　实时时钟硬件结构示例图

- 独立的电源引脚。
- 能够为实时内核的系统时钟提供 ms 级的时间中断。
- 能够进行循环复位。

由于实时时钟独立于操作系统，所以也被称为硬件时钟。它为整个系统提供一个计时标准。

另外，嵌入式微处理器通常还集成了多个定时器/计数器。实时内核需要一个定时器作为系统时钟(或称 OS 时钟)，并由实时内核控制系统时钟工作。一般情况下，系统时钟的最小粒度是由应用和操作系统的特点决定的。

在不同的操作系统中，实时时钟和系统时钟之间的关系是不同的。实时时钟和系统时钟之间的关系通常也被称作操作系统的时钟运作机制。一般来说，实时时钟是系统时钟的时间基准，实时内核通过读取实时时钟来初始化系统时钟，此后二者保持同步运行，共同维系系统时间。所以系统时钟并不是本质意义上的时钟，只有当系统运行起来以后才有效，并且由实时内核完全控制。

从硬件的角度来看，定时器(timer)和计数器(counter)的概念是可以互换的，其差别主要体现在硬件在特定应用中的使用情况。

图 7-9 为一个简单的定时器/计数器。该定时器包含一个可装入的 8 位计数寄存器，一个时钟输入信号和一个输出脉冲。通过软件可以把一个位于 0x00～0xFF 之间的初始数据转入到计数寄存器。随后的每一个时钟输入信号都会导致该值被增加。当 8 位计数器溢出时，就产生输出脉冲。输出脉冲可以用来触发处理器上的一个中断，或是在处理器能够读取的地方设置一个二进制位。输出脉冲是操作系统时钟的硬件基础，是因为输出脉冲将送到中断控

制器上，产生中断信号，触发时钟中断，由时钟中断服务程序维持操作系统时钟的正常工作。为了重启定时器，软件需要重新装入一个相同或不同的初始数据到计数寄存器。

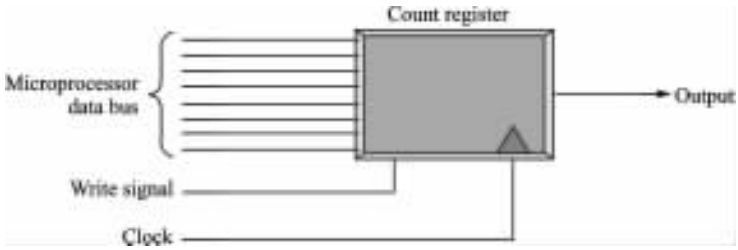


图 7-9　一个简单的定时器/计数器

如果计数器是增量计数器，将从初始装入的数据开始，递增计数到 0xFF；减量计数器则从初始装入的数据开始，递减计数到 0x0。

在一个典型的计数器中，当初始数据被装入后，可以使用一定的方式来启动计数器。该方式通常是在一个控制寄存器中设置一个二进制位。另外，一个实际的计数器也需要为处理器提供一种通过数据总线读取计数寄存器当前值的方式。

如果希望定时器能够自动重新装入初始数据，就需要一个锁存寄存器，以保存处理器所写入的计数数据。当处理器向锁存寄存器写入数据时，计数寄存器也被写入了该数据。定时器溢出时，定时器产生输出脉冲，然后自动把锁存寄存器中的数据重新装入到计数寄存器。这样，由于锁存寄存器仍然拥有处理器写入的数据，计数器将从同样的初始数据重新开始进行计数。这样的定时器能够产生与时钟具有相同精度的规则性输出。输出脉冲产生的周期性中断可以用于实时内核需要的 tick，或是为 UART 提供一个波特率时钟，或是驱动需要规则脉冲的设备。

7.2.2　时间管理

实时内核的时间管理以系统时钟为基础，系统时钟一般定义为整数或长整数，提供给应用程序所有与时间有关的服务。系统时钟是由定时器/计数器产生的输出脉冲触发中断而产生的。输出脉冲的周期叫做一个“时钟滴答”，也称为时标、tick。

tick 为系统的相对时间单位，也被称为系统的时基，来源于定时器的周期性中断。一次中断表示一个 tick。一个 tick 与具体时间的对应关系可在初始化定时器时设定，也就是说，tick 所对应的具体时间长度是可以调整的。一般来说，实时内核都提供相应的调整机制，应用可以根据特定情况改变 tick 对应的时间长度。如可以使系统 5 ms 产生一个 tick，也可以是 10 ms 产生一个 tick。tick 的大小决定了整个系统的时间粒度。

定时器的初始化工作主要包含以下内容：

- 初始化定时器相关的寄存器；

- 设置 tick 的间隔时间，使定时器每隔一个确定的时间产生一个时钟中断；
- 挂接系统时钟中断处理程序。

通常来说，实时内核提供以下主要与时间相关的管理：

- 维持相对时间(时间单位为 tick)和日历时间；
- 任务有限等待的计时；
- 定时功能；
- 时间片轮转调度的计时。

这些管理功能是通过 tick 处理程序来实现的，如图 7－10 所示。定时器发生中断后，执行系统时钟中断处理程序，并在中断处理程序中调用 tick 处理程序，实现系统中与时间和定时相关的操作。tick 处理程序作为实时内核的一部分，与具体的定时器/计数器硬件无关，由系统时钟中断处理程序调用，使实时内核具有对不同定时器/计数器硬件的适应性。

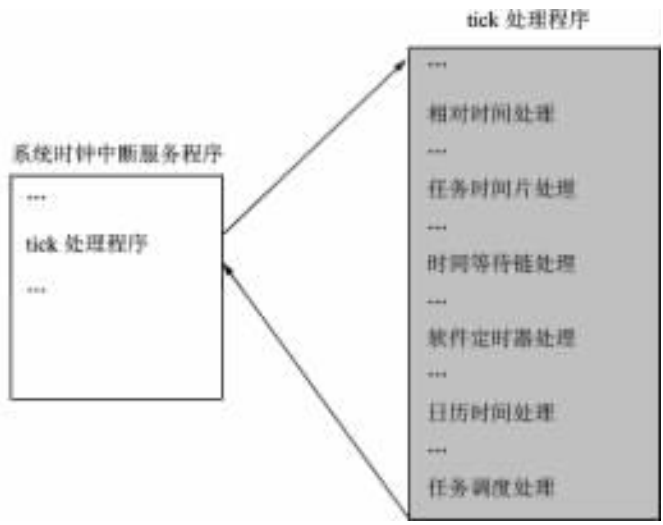


图 7－10　tick 处理程序

相对时间即系统时间，是指相对于系统启动以来的时间，以 tick 为单位，每发生一个 tick，对系统的相对时间进行一次加 1 操作。实时内核根据 tick 对应的时间长度，可以把相对时间转换为以 s 或是 ms 为单位的其他时间格式，并可根据实时时钟获得日历时间。

如果对任务设置了时间片处理方式，则需要在 tick 处理程序中对当前正在运行的任务的已执行时间进行更新，使任务的已执行时间数值加 1。执行加 1 操作后，如果任务的已执行时间同任务的时间片相等，则表示任务使用完一个时间片的执行时间，需要结束当前任务的执行，设置调度标志，把当前任务放置到就绪链。

时间等待链用来存放需要延迟处理的对象，产生 tick 后，需要对时间等待链中的对象的

剩余等待时间值进行处理。对于时间等待的对象，通常都被组织为差分链表的方式进行管理，以有效降低时间等待对象的管理开销。在时间差分链中，每个表项所包含的计时值并非当前时刻到表项激活时刻的绝对计数，而是该表项和先于它的所有表项的计数值之和。对于图 7－11所示的差分链，在当前时刻，A 对象需要等待 3 个时间单位就应被激活；B 对象需要等待 5(3＋2)个时间单位就应被激活；C 对象需要等待 10(3＋2＋5)个时间单位就应被激活；D 对象需要等待 14(3＋2＋5＋4)个时间单位就应被激活。在当前时刻，如果有一个等待 7 个时间单位的对象 E 需要插入到队列中，由于 7－3－2＝2，而 7－3－2－5＝－3，因此 E 对象需要插入到差分链中介于对象 B 和对象 C 之间的位置，如图 7－12 所示。

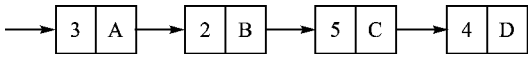


图 7－11　差分时间链一

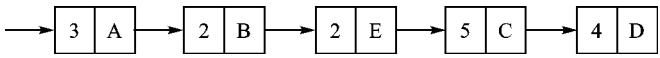


图 7－12　差分时间链二

对于差分时间链，系统每接收到一个 tick，就修订链首对象的时间值。如果链表对象的时间单位为 tick，则每发生一个 tick，链首对象的时间值就减 1；当减到 0 时，链首对象就被激活，并从差分时间链中取下来，下一个对象又成为链首对象。

为实现定时功能，实时内核需要提供软件定时器管理功能，应用程序可根据需要创建、使用软件定时器。软件定时器在创建时，由用户提供定时值；当软件定时器的定时值减法计数为 0 时，触发定时器服务例程。用户可在此例程中完成自己需要的操作。因此，在 tick 处理程序中需要对软件定时器的定时值进行减 1 操作，并在定时值为 0 时，触发挂接在该定时器上的服务例程。

软件定时器可用于实现“看门狗”(watchdog)。在应用的某个地方进行软件定时器的停止计时操作，确保定时器在系统正常运行的情况下不会到期，即不会触发定时器服务例程；如果某个时候系统进入了定时器服务例程，就表示使用停止计时操作的地方没有执行到，系统出现了错误。

如果需要进行任务的重调度，则 tick 处理程序还需要调用调度程序进行任务调度处理，使需要执行的下一个任务获得对 CPU 的控制。

在时间方面，内核通常提供以下功能：

- 设置系统时间　使应用能够设置当前系统的日期和时间。日历时间的数据结构示意情况如图 7－13 所示。
- 获得系统时间　以日历时间、系统启动以来所经历的 tick 数等形式获得当前的系统时间。

```
typedef struct
{
        unsigned32 year;          /*  year*/
        unsigned32 month;         /*  month，1-12  */
        unsigned32 day;           /*  day，1-31  */
        unsigned32 hour;          /*  hour，0-23  */
        unsigned32 minute;        /*  minute，0-59  */
        unsigned32 second;        /*  second，0-59  */
        unsigned32 ticks;         /* elapsed ticks between secs  */
}  TOD;                           /* Time Of Day */
```

图 7－13　日历时间数据结构示意

- **维护系统时基并处理定时事件**　通过时钟中断，维持系统日志时间、任务延迟时间、超时、单调速率周期和实现时间片等内容。

在定时方面，内核通常提供以下功能：

- **创建软件定时器**　分配一个定时器数据结构，创建一个软件定时器，并为这个定时器分配用户指定的名字。新创建的定时器没有被激活，且没有相应的定时器服务例程。软件定时器创建成功后，将为该定时器分配一个 ID 标识。图 7－14 为软件定时器数据结构的示意内容。图中，class 表示所创建定时器的触发时间类型，可以是相对时间触发，也可以是绝对时间触发；state 表示定时器的当前状态，可以是活动状态、非活动状态或是中间状态（如正在进行计时链表的插入操作）；interval 表示触发时间间隔；timeRemain 表示剩余的触发时间；startTime 表示自系统启动以来所经历的时间；handler 表示定时器需要触发的服务例程。*usrData 表示需要触发的服务例程的参数；type 表示定时器的触发类型，可以是单次触发、多次触发或是周期性触发；repeatCount 表示多次触发时重复触发的次数；repeatRemain 表示多次触发情况下的剩余触发次数。
- **启动软件定时器**　使定时器在给定的时间过去后，触发定时器服务例程。对于软件定时器，通常还可以指定是单次触发还是周期触发。在单次触发中，只触发执行一次挂接的定时服务例程；周期触发则可以在每次触发服务例程后，经过相同的时间间隔又会触发挂接在该定时器上的服务例程。
- **使软件定时器停止计时**　使指定的软件定时器停止工作。因此，对应的定时器服务例程不再被触发，除非定时器被重新激活。
- **复位软件定时器**　把定时器的定时值恢复到原来设定的值。
- **删除软件定时器**　用来删除一个软件定时器。如果定时器还在工作，则其自动停止。

该定时器对应的数据结构被返回给系统。

```
typedef struct
{
    timer_class                 class;
    timer_state                 state;
    timer_time                  interval;
    timer_time                  timeRemain;
    timer_time                  startTime;
    timer_service_routine_entry handler;
    void                        *usrData;
    attribute                   type;
    unsigned32                  repeatCount;
    unsigned32                  repeatRemain;
} timerInformation;
```

图 7-14　软件定时器数据结构示意

思考题

7.1　请阐述中断的概念，并说明中断与自陷、异常之间在概念上有哪些联系与区别。

7.2　请说明中断是如何分类的。

7.3　请描述中断处理的基本过程。

7.4　以嵌入式应用中的具体硬件为例，收集相关资料，通过图示和文字描述的方式详细描述中断硬件的体系结构，并在不考虑操作系统的情况下，描述相应的中断服务程序实现方式。

7.5　以一种开源的嵌入式操作系统为例，就该操作系统所采用的中断管理方式进行详细分析，并写出分析报告。

7.6　请分别描述什么叫实时时钟和定时器/计数器。

7.7　请说明在系统时钟中断服务程序中，主要完成哪些工作。

7.8　以嵌入式应用中的具体硬件为例，收集相关资料，通过图示和文字描述的方式详细描述时间管理硬件的体系结构。

7.9　以一种开源的嵌入式操作系统为例，就该操作系统所采用的时间管理方式进行详细分析，并写出分析报告。

第 8 章　内存管理和 I/O 管理

8.1　内存管理

8.1.1　概　述

不同实时内核所采用的内存管理方式不同，有的简单，有的复杂。实时内核所采用的内存管理方式与应用领域及硬件环境密切相关。在强实时应用领域，内存管理方法就比较简单，甚至不提供内存管理功能。一些实时性要求不高，可靠性要求比较高，且系统比较复杂的应用在内存管理上就相对复杂些，可能需要实现对操作系统或是任务的保护。

通常，嵌入式实时操作系统在内存管理方面需要考虑如下因素：

- 快速而确定的内存管理　最快速和最确定的内存管理方式是根本不使用内存管理。这意味着编程人员可以把整个可以获得的物理内存区域作为一个连续的内存块，并按照自己的需要进行自由使用。这种方法只适用于那些小型的嵌入式系统，系统中的任务比较少，且数量固定。通常的操作系统都至少具有基本的内存管理方法，即提供内存分配与释放的系统调用。
- 不使用虚拟存储技术　虚拟存储技术为用户提供一种不受物理存储器结构和容量限制的存储管理技术，是桌面操作系统为在所有任务中使用有限物理内存的通常方法。每个任务从内存中获得一定数量的页面，并且当前不访问的页面将被置换出去，为需要页面的其他任务腾出空间。这种置换是一种具有不确定性的操作，置换需要访问的磁盘等后援存储器。而磁盘控制器为优化平均吞吐率，通常会进行具有不确定性的缓存操作；当任务需要使用当前被置换出去的页面中的代码和数据时，将不得不从磁盘中获取页面，而在内存中另外的页面又可能不得不需要先被置换出去。因此，在嵌入式实时操作系统中一般不使用虚拟存储技术，以避免页面置换所带来的开销。
- 内存保护　大多数传统的嵌入式操作系统依赖于平面内存模式，应用程序和系统程序能够对整个内存空间进行访问。平面内存模式比较简单，易于管理，性能也比较好，但通常只适合于程序简单、代码量小和实时性要求比较高的领域。在应用比较复杂、程序量比较大的情况下，为了保证整个系统的可靠性，就需要对内存进行保护，防止应用程序破坏操作系统或是其他应用程序的代码和数据。内存保护包含两个方面的内容。一方面的内容是防止地址越界，每个应用程序都有自己独立的地址空间；当应用程序

要访问某个内存单元时，由硬件检查该地址是否在限定的地址空间之内，只有在限定地址空间之内的内存单元访问才是合法的，否则需要进行地址越界处理。内存保护另一个方面的内容是防止操作越权，对于允许多个应用程序共享的存储区域，每个应用程序都有自己的访问权限；如果一个应用程序对共享区域的访问违反了权限规定，则进行操作越权处理。

8.1.2　内存管理机制

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。在静态分配方式中，系统在启动前，所有的任务都获得了所需要的所有内存，运行过程中将不会有新的内存请求。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。为此，大多数系统都使用内存的动态分配方式。

动态内存的传统管理机制为堆(heap)，应用通过分配(malloc)与释放(free)操作来使用内存。在使用一段时间后，堆会带来碎片(fragmentation)，内存被逐渐划分为位于已被使用区域之间的越来越小的空闲区域。在申请使用内存区域时，如果需要使用的内存区域的大小超过了最大可获得的分片大小，则操作系统内核可能会使任务停止运行，并等待以获得需要的内存。有的操作系统内核提供了垃圾回收(garbage collection)功能，对内存堆进行重新排列，把碎片组织成为大的连续可用内存空间。垃圾回收看上去是解决内存碎片的有效办法，但该方法可能在一个随机的时间使任务停止运行，并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。在实时系统中，比较好的办法是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进行回收。

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。应用根据需要从固定大小存储区或者可变大小存储区中获得一块内存空间，用完后将该内存空间释放回相应的存储区。

1. 固定大小存储区管理

在固定大小存储区管理中，可供使用的一段连续的内存空间被称为是一个分区。分区由大小固定的内存块构成，且分区的大小是内存块大小的整数倍。一个大小为 512 字节的分区，内存块为 128 字节的分区，如图 8－1 所示。分区可采用的数据结构如图 8－2 所示。

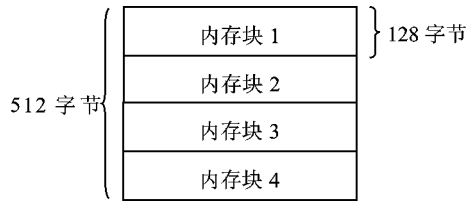


图 8－1　固定大小存储区示意图

```
typedef struct
{
        PartitionID         ID;                     /*分区的 ID */
        PartitionName       Name;                   /* 分区的名字 */
        void                *starting_address;      /* 分区的起始地址 */
         int                length;                 /* 分区的长度 */
        int                 buffer_size;            /*内存块的大小 */
         PartitionAttribute attribute;              /* 属性 */
         int                number_of_used_blocks;  /* 剩余内存块数 */
         MemoryChain        memory;                 /* 内存块链 */
} Partition;
```

图 8-2　分区的数据结构

图 8-2 **中,**ID **表示分区的标识;***starting_address **表示分区的起始地址;**length **表示分区的存储单元的数量;**buffer_size **表示分区中每个内存块的大小;**attribute **表示分区的属性;**number_of_used_blocks **表示分区中已使用内存块的数量;**memory **为一个指针,指向分区中由空闲内存块组成的双向空闲内存块链表的头节点。空闲内存块链表的示意图如图** 8-3 **所示。**

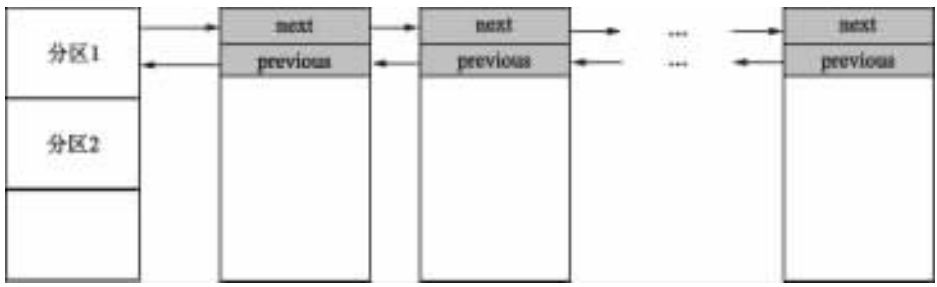


图 8-3　由分区管理的空闲内存块链表示意图

分区的操作主要包含以下内容:

- **创建分区;**
- **删除分区;**
- **从分区得到内存块;**
- **把内存块释放到分区;**
- **获取分区** ID;
- **获取当前创建的分区的数量;**

- 获取当前所有分区的 ID；
- 获取分区信息。

创建分区时，根据分区和内存块的大小，把分区组织为一个空闲内存块构成的双向链表。如果创建分区成功，将返回分区的 ID，用于关于分区的其他操作。

删除分区用来释放分区所占据的存储空间。

当需要使用分区中的内存块时，可以根据分区的 ID，从分区的空闲内存块链表中，按照空闲内存块链表的顺序进行内存块的分配。如果分区中没有空闲内存块，调用者不会被阻塞，而是获得一个空指针，以确保申请内存调用的时间确定性。

如果不再使用从分区中获得的内存块，则需要把该内存块释放给分区，将该内存块挂在空闲内存块链的链尾。

在固定大小存储区的管理方式中，如果内存块处于空闲状态，将使用内存块中的几个字节作为控制结构，用来存放用于双向链接的前向指针和后向指针。在使用内存块时，内存块中原有的控制信息不再有效，其中的所有存储空间都可以被使用。因此，内存块的控制结构所占用的内存空间不会影响该内存块实际可用的大小，即固定大小存储区管理的系统开销对用户的影响为零。另外，由于内存块的大小固定，所以不存在碎片的问题。

2. 可变大小存储区管理

可变大小存储区管理为基于堆的管理方式。堆为一段连续的、大小可配置的内存空间，用来提供可变内存块的分配。可变内存块称为段，最小分配单位称为页，即段的大小是页的大小的整数倍。如果申请段的大小不是页的倍数，则实时内核将会对段的大小进行调整，调整为页的倍数。如从页大小为 256 字节的堆中分配一个大小为 350 字节的段，实时内核实际分配的段大小为 512 字节。

堆的数据结构示意图如图 8－4 所示。图中，waitQueue 表示任务等待队列，如果任务从堆中申请段不能得到满足，将被阻塞在堆的等待队列上；*starting_address 表示堆在内存中的起始地址；length 表示堆的大小；page_size 表示页的大小；maximum_segment_size 表示堆中当前最大可用段的大小；attribute 表示堆的属性；number_of_used_blocks 表示已分配使用的内存块的数量；memory 表示空闲段链表。

堆的属性用来控制任务等待段的方式。任务等待段主要有以下方式：

- 任务按先进先出顺序等待；
- 任务按优先级顺序等待。

可变大小存储区中的空闲段通过双向链表链接起来，形成一个空闲段链。在创建堆时，只有一个空闲段，其大小为整个存储区的大小减去控制结构的内存开销。从存储区中分配段时，可依据首次适应算法，查看空闲链中是否存在合适的段。当把段释放回存储区时，该段将被挂在空闲段链的链尾；如果空闲链中有与该段相邻的段，则将其合并成一个更大的空闲段。该方法由于对申请的内存的大小作了一些限制，所以避免了内存碎片的产生。

```
typedef struct {
    HeapID              ID;                 /* 堆的 ID */
    HeadName            name;               /* 堆的名字 */
    TaskQueue           waitQueue;          /* 等待队列 */
    void                *starting_address;  /* 内存空间起始地址*/
    int                 length;             /* 内存空间长度（字节）*/
    int                 page_size;          /* 页长度（字节）*/
    int                 maximum_segment_size; /* 最大可用段大小*/
    RegionAttribute     attribute;          /* 堆的属性 */
    int                 number_of_used_blocks; /* 分配的块数 */
    HeapMemoryChain     memory;             /* 堆头控制结构 */
}Heap;
```

图 8-4　堆的数据结构

空闲段链的示意图如图 8-5 所示。图 8-5 中，在段的控制块中设置了一个标志位，表示段被使用的情况。标志位为 1 表示该段正被使用；标志位为 0 表示该段空闲。

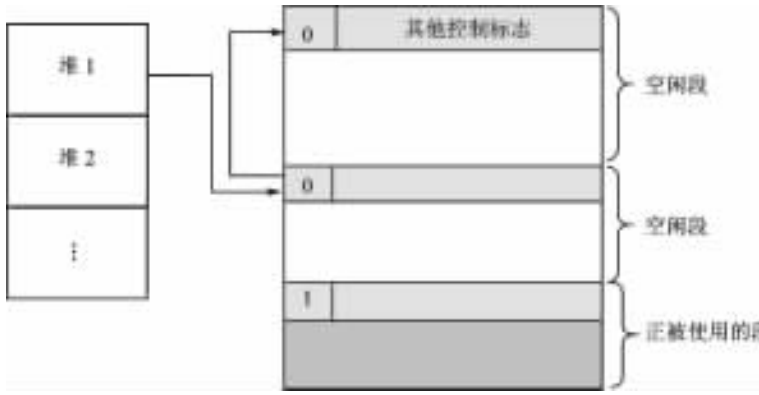


图 8-5　堆的空闲段链

在固定大小存储区管理方式中，只有在空闲状态下，内存块才拥有控制信息。在可变大小存储区管理方式中，无论段空闲或是正在被使用，段的控制结构都始终存在。

堆的操作主要包含以下内容：

- 创建堆；
- 从堆中得到内存块；
- 释放内存块到堆中；
- 扩展堆；

- 获得已分配内存块的实际可用空间大小；
- 删除堆；
- 获得堆的 ID；
- 获得在堆上等待的任务数量；
- 获得等待任务的 ID 列表；
- 获得堆的数量；
- 获得堆列表；
- 获得堆信息。

创建堆操作用来根据指定的名字、起始地址、堆大小、页大小和属性创建一个具有可变大小存储区管理的堆，并在创建成功的情况下，返回堆的 ID，用于关于堆的其他操作。

堆创建成功后，可以从堆中获取指定大小的内存块。如果当前堆中没有满足要求的内存块可供使用，则调用者可以选择以下处理行为：

- 不等待；
- 有限时间等待；
- 永久等待。

如果调用者选择不等待处理方式，则在没有可供使用的内存块时，执行流程将立即返回调用程序。调用者也可以选择有限时间等待或是永久等待的处理方式。

在向堆中返回内存块时，将把内存块同相邻的空闲内存块进行合并，形成一个更大的空闲内存块。此时，如果有任务等待获得内存块，则内核会检查是否能满足等待队列中第一个任务的请求；如果可以，则把分配内存块给该等待任务。这个过程会不断重复，直到等待队列上的任务的请求不能被满足。

在堆不能满足应用需要的情况下，还可以对堆的大小进行扩展。但通常要求扩展的存储空间应该与原有的堆相邻接，这样扩展后的堆仍然是一段连续的存储空间。

删除堆用来释放堆所占据的存储空间，但要求堆里面没有正处于被使用状态的内存块。

3. 内存保护

内存保护可通过硬件提供的 MMU(Memory Management Unit)来实现。目前，大多数处理器都集成了 MMU。这种在处理器内部实现 MMU 的方式，能够大幅度降低那些通过在处理器外部添加 MMU 模块的处理方式所存在的内存访问延迟。MMU 现在大都被设计作为处理器内部指令执行流水线的一部分，使得使用 MMU 不会降低系统性能；相反，如果系统软件不使用 MMU，还会导致处理器的性能降低。在某些情况下，不使能 MMU，跳过处理器的相应流水线，可能导致处理器的性能降低 80 %。

早期的嵌入式操作系统大都没有采用 MMU，一方面是出于对硬件成本的考虑，另一方面是出于实时性的考虑。嵌入式系统发展到现在，硬件成本越来越低，MMU 所带来的成本因素基本上可以不用考虑。另外，原来的嵌入式 CPU 的速度较慢，采用 MMU 通常会造成对时间性

能的不满足，而现在 CPU 的速度也越来越快，并且采用新技术后，已经将 MMU 所带来的时间代价降低到比较低的程度。因此，嵌入式 CPU 具有 MMU 的功能已经是一种必然的趋势。

由于采用 MMU 后对应用的运行模式甚至开发模式都会有一些影响，所以大量嵌入式操作系统都没有使用 MMU。但是，对于安全性、可靠性要求高的应用来讲，如果不采用 MMU，则几乎不可能达到应用的要求。如果没有 MMU 的功能，将无法防止程序的无意破坏，无法截获各种非法的访问异常，当然更不可能防止应用程序的蓄意破坏了。采用 MMU 后，能够在应用开发阶段便于发现更多的潜在问题，并且也便于问题的定位。因为未采用 MMU 时，内存模式一般都是平面模式，也就是应用可以任意访问任何内存区域、任何硬件设备，这样当程序中出现非法访问时，开发人员是无从知晓的，即使发现了问题，也非常难以定位。因此，使用 MMU 的好处是不言而喻的。

MMU 通常具有如下功能：

- 内存映射；
- 检查逻辑地址是否在限定的地址范围内，防止页面地址越界；
- 检查对内存页面的访问是否违背特权信息，防止越权操作内存页面；
- 在必要的时候（页面地址越界或是页面操作越权）产生异常。

内存映射把应用程序使用的地址集合（逻辑地址）翻译为实际的物理内存地址（物理地址），如图 8－6 所示。

图 8－6　通过 MMU 进行内存映射

应用程序需要通过内存来存储以下内容：

- 指令代码（二进制机器指令）。
- 静态分配的数据（如静态变量、全局变量）。
- 具有后进先出（last in，first out）处理方式的栈或动态分配的数据（如动态变量和返回地址）。
- 堆。用来存储数据，并可被编程人员分配和释放。

在 MMU 的处理方式下，上述属于应用程序的四个部分的空间被划分为大小相等的页面（page）。大多数处理器的典型页面大小为 4 KB，有些处理器也可能使用大于 4 KB 的页面，但页面大小总是 2 的幂，以对发生在 MMU 中的地址映射行为流水线化。当页放置到物理内存时，页面将放置到页框架（page frame）中。页框架是物理内存的一部分，具有与页面同样的大小，且开始地址为页面大小的整数倍。MMU 包含着能够把逻辑地址映射为物理地址的表，称为页表（page table）。基于页表的内存映射过程如图 8－7 所示。操作系统能够在需要的时候

对这种映射关系进行改变，如应用程序对内存的需求发生变化或是当应用程序被添加或删除的时候。在应用程序中的任务发生上下文切换时，操作系统也可能需要对映射关系进行改变。

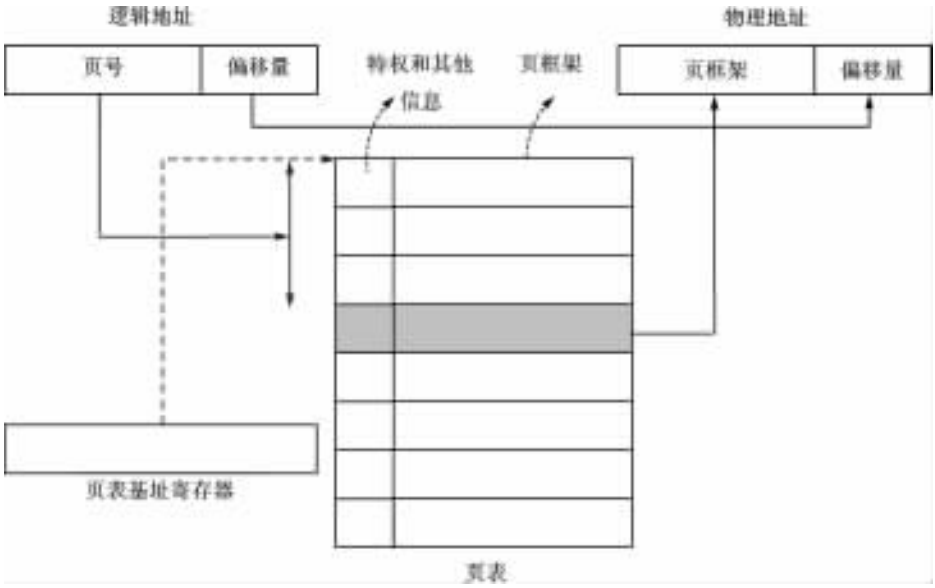


图 8-7　基于页表的内存映射过程

例如，一个系统具有 32 位的地址空间和 4 KB 的页面，32 位的地址空间由 20 位的页号和 12 位的页内偏移量构成。MMU 将检查 20 位的页号，并为该页面提供（根据 MMU 表进行映射）页框架地址。结合映射得到的页框架地址和 12 位的页内偏移量，即可得到实际的物理地址。具有流水线处理能力的处理器可为 MMU 映射分配一个流水线阶段，并用另一个阶段访问实际的内存。

每个内存页还具有一些特权和状态信息。MMU 提供二进制位来标识每个页面的特权或状态信息。这些二进制位用来确定页面中的内容是否满足以下条件：

- 可被处理器指令所使用（执行特权）；
- 可写（写特权）；
- 可读（读特权）；
- 已被回写（脏位）；
- 当前在物理内存中（有效位）。

另外，在操作系统的支持下，MMU 还提供虚拟存储功能，即在任务所需要的内存空间超过能够从系统中获得的物理内存空间的情况下，也能够得到正常运行。当需要的页面被添加到逻辑地址空间时，任务对内存页面的合法访问将自动访问到物理内存。如果该页面当前不在物理内存中时，将导致页面故障异常，然后操作系统负责从后援存储器（如硬盘或是 Flash

存储设备)中获取需要的页面,并从产生页面故障的机器指令处重新执行。

在实际应用中,MMU 通常具有如下不同功能程度的使用方式:

① 0 级,内存的平面使用模式　该模式中,没有使用 MMU,应用程序和系统程序能够对整个内存空间进行访问。采用该模式的系统比较简单,性能也比较好,适合于程序简单、代码量小和实时性要求比较高的领域。大多数传统的嵌入式操作系统都采用该模式。

② 1 级,用来处理具有 MMU 和内存缓存的嵌入式处理器　由于大多数传统的嵌入式操作系统依赖于平面内存模式,为简化系统,1 级模式通常只是打开 MMU,并通过创建一个域(domain,为内存保护的基本单位,每个域对应一个页表)的方式来使用内存,并对每次内存访问执行一些必要的地址转换操作。事实上,该模式仍然只是拥有 MMU 打开特性的平面内存模式。

③ 2 级,内存保护模式　该模式下,MMU 被打开,且创建了静态的域(应用程序的逻辑地址同应用程序在物理内存中的物理地址之间的映射关系在系统运行前就已经确定),以保护应用和操作系统在指针试图访问其他程序的地址空间时不会被非法操作。在该级别的内存保护模式下,通常使用消息传送机制实现数据在被 MMU 保护起来的各个域之间的移动。

④ 3 级,虚拟内存使用模式　在该模式下,通过操作系统使用 CPU 提供的内存映射机制,内存页被动态地分配、释放或是重新分配。从内存映射到基于磁盘的虚拟内存页的过程是透明的。

在 MMU 的上述四种使用模式中,0 级模式为大多数传统嵌入式实时操作系统的使用模式,同 1 级模式一样,都是内存的平面使用模式,不能实现内存的保护功能。2 级模式是目前大多数嵌入式实时操作系统所采用的内存管理模式,既能实现内存保护功能,又能通过静态域的使用方式保证系统的实时特性。3 级模式适合于应用比较复杂、程序量比较大,并且不要求实时性的应用领域。

图 8-8 为一种基于静态域的 MMU 使用方式,也称为是一一对应的使用方式。在这种一一对应的使用方式中,应用程序的逻辑地址同应用程序在物理内存中的物理地址相同,简化了内存管理的实现方式。

因此,在嵌入式实时操作系统中,MMU 通常被用来进行内存保护,实现操作系统与应用程序的隔离,以及应用程序与应用程序之间的隔离,这样可以防止应用程序破坏操作系统的代码、数据以及应用程序对硬件的直接访问。对于应用程序来讲,也可以防止别的应用程序对自己的非法入侵,而破坏应用程序自身的运行。

在内存保护方面,MMU 提供了以下措施:

- 防止地址越界　通过限长寄存器检查逻辑地址,确保应用程序只能访问逻辑地址空间所对应的、限定的物理地址空间。MMU 将在逻辑地址超越限长寄存器所限定的范围时产生异常。
- 防止操作越权　根据内存页面的特权信息控制应用程序对内存页面的访问,如果对内

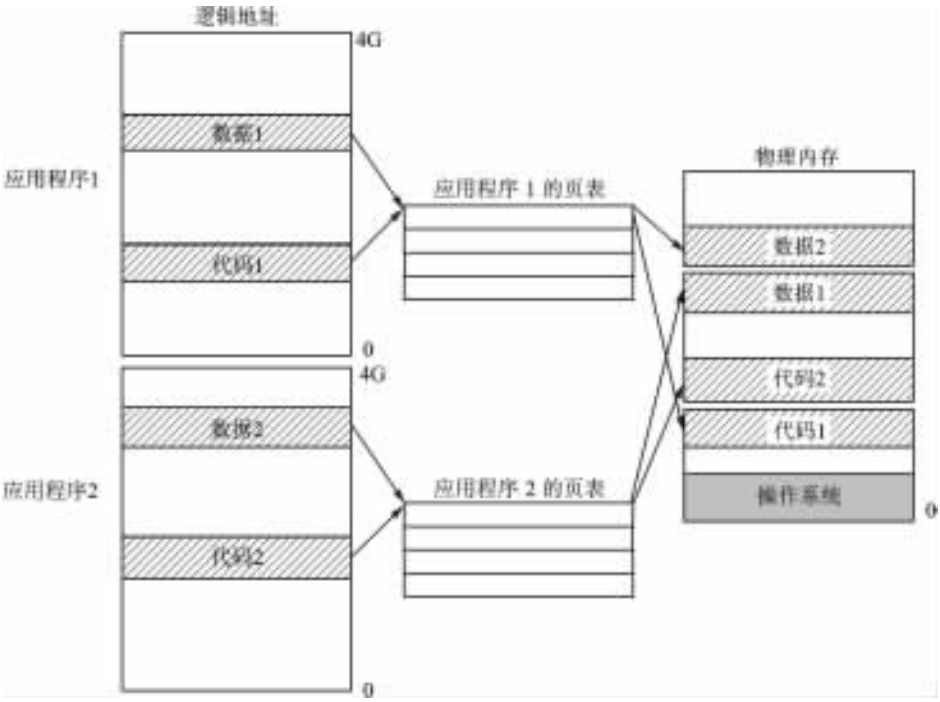


图 8－8　基于静态域的 MMU 使用方式示意图

存页面的访问违背了内存页面的特权信息，则 MMU 将产生异常。

简单的 MMU 保护模式如图 8－9 所示。在这种模式中，应用程序之间要通信就只能通过操作系统提供的通信服务，如信号量、管道、消息和共享内存等，而不能直接访问彼此的地址空间。另外，MMU 通常还提供权限等级，不同的权限等级对硬件访问的权限不一样。操作系统

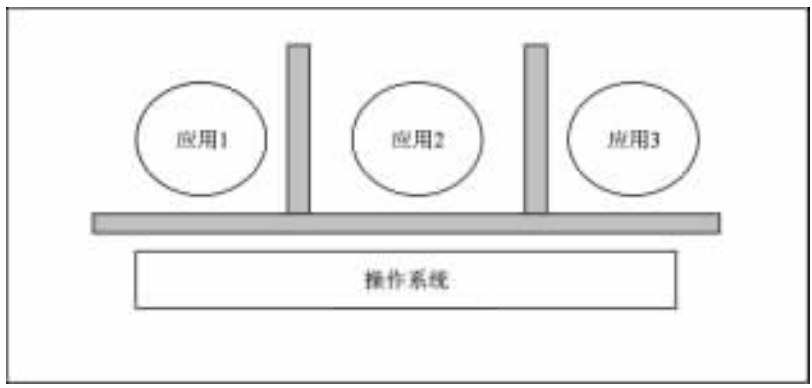


图 8－9　简单的 MMU 保护模式

一般运行在核心态，具有所有的特权；而应用则一般运行在用户态，具有一般权限，以防止应用程序的故意破坏。

图 8－10 为基于域的一种内存管理方式。一个域可以包含多个任务、内存页、系统对象（信号量、队列等）和各种共享区域等，代表了一个执行环境，是一个隔离的空间，与进程的概念类似。可以认为域是资源分配的单位，任务则是内核调度的单位。在基于域的管理方式中，域就是一个独立的空间，共享库、内核等部分的代码和数据空间直接映射到域空间中，在每个域中都是独立的。

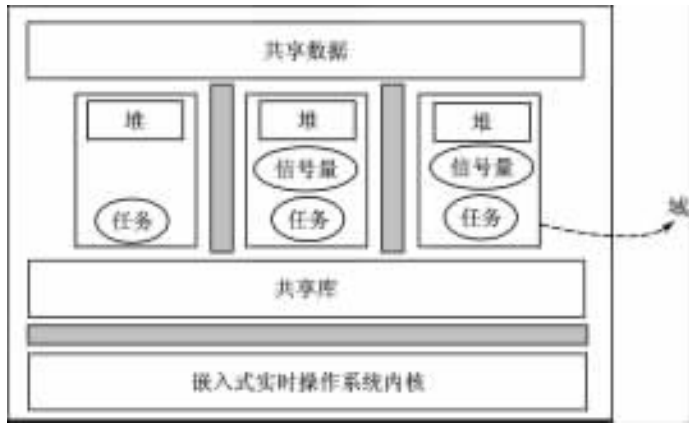


图 8－10　基于域的内存管理方式

图 8－10 所示的方法可实现如下功能：

- 应用和应用之间、应用和嵌入式实时操作系统内核之间通过域的方式进行了相互隔离，互不侵犯。
- 共享库（如文件系统、网络协议、图形用户接口等）采用非受限访问方式，以提高应用的运行速度。但可能由于没有对共享库采用任何的保护机制，容易造成系统崩溃的现象。如果要实现全面的保护，共享库也应该通过域的方式进行管理，并按照客户/服务器的方式提供服务。
- 嵌入式实时操作系统内核可通过 TRAP（自陷）的机制提供服务。
- 堆是域私有的，进一步提高了安全性。

8.2　I/O 管理

在通用计算机的操作系统中，I/O 管理采用层次结构的思想（如四个层次的结构：中断处理程序、设备驱动程序、与设备无关的操作系统软件和用户层软件），较低层的软件要具有使较高层的软件独立于硬件的特性，较高层软件则要向用户提供一个友好、清晰、规范的界面。在

I/O 管理的层次结构中，主要通过设备独立的 I/O 系统和设备驱动程序来共同完成 I/O 操作。设备驱动程序通过一组例程来提供比较低级的 I/O 功能，比如把字节序列输入或输出到面向字符的设备中。高级协议（如面向字符设备的通信协议）则由与设备无关的 I/O 系统来实现。用户的 I/O 请求则主要是由设备驱动程序获得控制之前的 I/O 系统来进行处理。

I/O 管理的分层处理方式在实现驱动程序以及确保不同设备具有相近的行为特性方面具有比较明显的优势，但也存在不足之处。在这种处理方式中，对于驱动程序的编写人员来说，实现一些 I/O 系统中没有提供的通信协议将是一件困难的事情。而在嵌入式系统中，由于提高设备吞吐率或是设备本身不适于采用标准的 I/O 处理模式时，经常需要另行编写设备通信协议，而不采用系统提供的标准通信协议。

在实时内核的 I/O 系统中，用户 I/O 请求在到达设备驱动程序之前，通常都只进行非常少量的处理。事实上，实时内核的 I/O 系统的作用就像一个转换表，把用户对 I/O 的请求转换到相应的驱动程序例程。这样，驱动程序就能够获得最原始的用户 I/O 请求，并对设备进行操作。

另外，为满足标准设备处理的需要，I/O 系统通常也提供一些高级的例程库，便于实现设备的标准通信协议。这样，I/O 系统既便于实现能够满足大多数设备要求的、标准的驱动程序，也能在需要的时候，方便地实现非标准的设备驱动程序，以满足实时性或是其他的特殊需要。

8.2.1　I/O 管理的功能

I/O 管理在整个系统中的体系结构如图 8－11 所示。

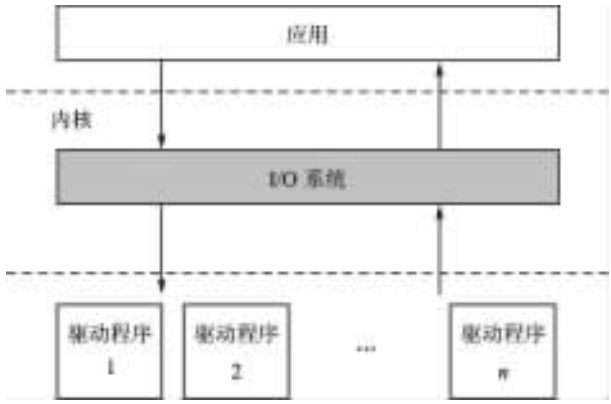


图 8－11　I/O 管理在系统中的体系结构

I/O 系统主要提供以下功能：

- 管理设备驱动程序；
- 实现设备命名；

● 向用户提供统一的调用。

管理设备驱动程序通过驱动程序地址表来实现。驱动程序地址表中存放了设备驱动程序的入口地址,可以通过此表实现设备驱动的动态安装与卸载的管理。驱动程序地址表通常可以通过双向循环链表或是数组的方式来实现。

应用层可以通过设备名来使用设备,I/O 系统和驱动程序内部则采用主/次设备号来操作设备,即用主设备号区别不同的驱动程序。由于一个驱动程序可能管理多个同类设备,所以驱动程序内部再用次设备号区别不同设备。比如,一个串行通信设备的驱动程序可以用来处理多个独立串行通道,这些通道之间只存在参数方面的差异(如设备地址)。为此,I/O 系统还需要提供将设备名映射到主/次设备号的方法。可以通过设备名表来实现设备名到主/次设备号的映射方法。有了设备名,通过设备名表就能得到设备的主/次设备号。

另外,应用层如果每次都通过设备名使用设备并不方便。为此,I/O 系统可采用文件描述符的机制简化这一过程。用户打开设备后获得该设备的文件描述符,以后对设备的操作都通过这个文件描述符来进行。

由此,I/O 系统可采用主/次设备号、设备名表和文件描述符等三方面的内容共同完成设备命名,即设备识别。

为便于 I/O 系统使用的统一性,通常都要求 I/O 系统提供统一的调用接口。I/O 系统的统一接口主要实现以下几种对设备的操作:① 设备初始化;② 打开设备;③ 关闭设备;④ 读设备;⑤ 写设备;⑥ 设备控制。

8.2.2 I/O 系统的实现考虑

I/O 系统主要通过文件描述符表、设备名表和驱动程序地址表实现 I/O 的管理功能。I/O 系统内部文件描述符、设备名和主设备号之间的层次关系如图 8-12 所示。

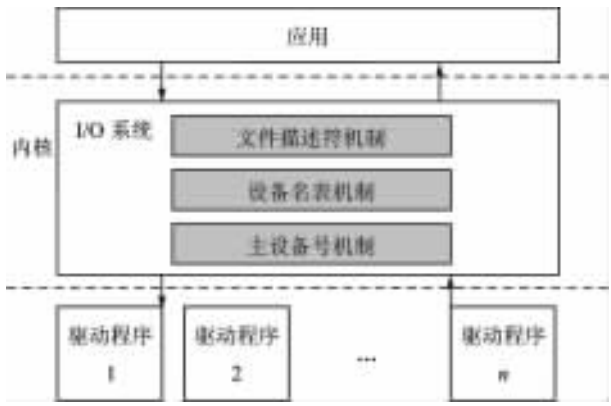


图 8-12 I/O 系统内部主要机制之间的层次关系

1. 主设备号

I/O 系统内部最下层是主设备号机制，通过主设备号来区分不同的驱动程序。I/O 系统内部不使用次设备号，次设备号仅在驱动程序内部有用。

主设备号机制的实现主要体现在驱动程序地址表数据结构上，实现了 I/O 系统对驱动程序的统一管理。主设备号是访问驱动程序地址表的索引，通过主设备号访问驱动程序地址表，可以获得驱动程序提供的关于设备的六个标准操作的函数调用的入口地址。

若驱动程序地址表组织为数组的形式，则需要在 I/O 系统初始化时为驱动程序地址表分配存储空间，以容纳各个设备的驱动程序的入口地址。若驱动程序地址表组织为双向循环链表的形式，则在安装设备的时候动态分配容纳该设备驱动程序入口地址的存储空间。

在进行驱动程序的安装或是卸载时，需要对驱动程序地址表中的相应内容进行修订。

驱动程序地址表的数据结构及其示意图分别如图 8－13 和图 8－14 所示。

```
#define NUMBER_OF_DRIVERS 16    /* 驱动程序的最大数量 */
struct DriverAddressTable
{
    DeviceDriverEntry    init;      /* 驱动程序的xxx_init函数指针 */
    DeviceDriverEntry    open;      /* 驱动程序的xxx_open函数指针 */
    DeviceDriverEntry    close;     /* 驱动程序的xxx_close函数指针 */
    DeviceDriverEntry    read;      /* 驱动程序的xxx_read函数指针 */
    DeviceDriverEntry    write;     /* 驱动程序的xxx_write函数指针 */
    DeviceDriverEntry    control;   /* 驱动程序的xxx_control函数指针 */
} DriverAddressTable[NUMBER_OF_DRIVERS];
```

图 8－13　驱动程序地址表的数据结构

2. 设备名表

设备名表中含有设备名、主设备号和次设备号等内容。设备名表完成设备名与主/次设备号之间的转换，所有要使用的设备（不论当前是打开还是关闭）都要在设备名表中注册上述内容，每个设备对应设备名表中的一项内容。设备名表的数据结构及其示意图如图 8－15 和图 8－16所示。

对于一个具体设备来说，其在设备名表中的内容在安装驱动程序的时候进行注册。在卸载驱动程序时，该驱动管理的设备在设备名表中的注册信息会被清除掉。如果有多个同类型设备需要使用相同的驱动程序，则需要使用相同的主设备号、不同的设备名和不同的次设备号在设备名表中进行注册。对于次设备号的作用，由驱动程序根据具体情况进行处理。

设备名表中含有主设备号，通过设备名查找设备名表就可以得到其主设备号，用这个主设备号就可以访问驱动程序地址表，得到驱动程序信息。

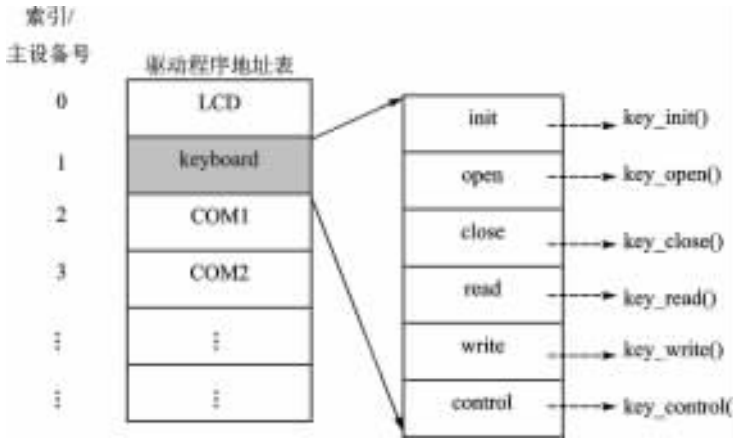


图 8－14　驱动程序地址表示意图

```
/* 设备的最大数量 */
#define NUMBER_OF_DEVICES 16

struct DeviceNameTable
{
    char   DeviceName[16];      /* 设备名 */
    int    major;               /* 主设备号 */
    int    minor;               /* 次设备号 */
} DeviceNameTable[NUMBER_OF_DEVICES];
```

图 8－15　设备名表数据结构

设备名	主设备号	次设备号
LCD	0	0
keyboard	1	0
COM1	2	0
COM2	2	1
⋮	⋮	⋮
⋮	⋮	⋮

图 8－16　设备名表示意图

至此，通过设备名表和主设备号机制已经可以实现用设备名对设备驱动程序的访问。但通过设备名调用驱动还不是非常方便，为此，还可以在设备名表和主设备号机制的基础上使用文件描述符的处理方式。

3. 文件描述符

文件描述符机制需要使用文件描述符表。在文件描述符机制中，用户打开设备后为设备分配一个文件描述符，以后对设备的操作都通过这个文件描述符进行。在用户看来，文件描述符就是设备的句柄，通过文件描述符就能完成与设备相关的所有操作，不再通过设备名。由此，可把对设备的操作同对文件的操作统一起来。

文件描述符表维护着当前打开的设备的信息，其数据结构及其示意图分别如图 8－17 和图 8－18 所示。其中，包含了实现临界资源互斥访问的机制。

```
/* 文件的最大数量 */
#define NUMBER_OF_FILES 16
typedef struct
{
        /* 指向对应的DNT表项的指针 */
        struct DeviceNameTable      *pDNT;
        int size;           /* 文件尺寸 */
        int offset;         /* 从打开至今累计读/写字符数 */
        int flag;           /* 标志位 */
        int sem;            /* 信号量信息 */
} FileDescriptorTable NUMBER_OF_FILES];
```

图 8－17　文件描述符表示意图

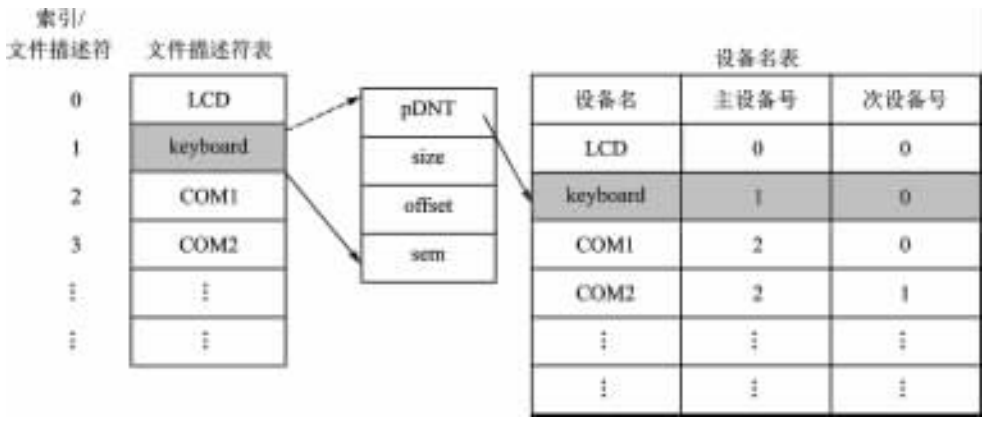


图 8－18　文件描述符表示意图

文件描述符实际上是访问文件描述符表的索引，文件描述符表的每一项才是真正的文件描述符结构。用户第一次打开设备后获得该设备的文件描述符，以后对设备的操作都通过文件描述符进行。

用文件描述符访问文件描述符表得到文件描述符结构，文件描述符结构中的域 pDNT 是指向设备名表中的对应表项，再访问设备名表就可以得到主设备号，用主设备号访问驱动程序地址表就可以调用设备驱动程序提供的调用。

思考题

8.1　请说明内存主要存放哪些内容，实时系统在进行内存管理时通常需要考虑哪些因素。

8.2　以嵌入式应用中的具体硬件为例，收集相关资料，通过图示和文字描述的方式详细描述 MMU 的处理情况。

8.3　以一种开源的嵌入式操作系统为例，就该操作系统所采用的内存管理方式进行详细分析，并写出分析报告。

8.4　请以一个具体的设备为例，基于主设备号、设备名表和文件描述符的 I/O 管理机制，对 I/O 设备管理的具体处理过程进行描述。

8.5　以一种开源的嵌入式操作系统为例，就该操作系统所采用的 I/O 管理方式进行详细分析，并写出分析报告。

第 9 章　嵌入式系统软件的开发

本章首先介绍嵌入式系统的开发模式，重点讨论如何选择处理器及硬件开发平台、操作系统和开发环境的问题。这些都对后续的嵌入式系统软件开发有很大影响。然后，以一个机器人控制器为例，具体讲解两种分析设计嵌入式实时软件尤其是多任务软件的方法——DARTS 和 UML，重点回答如何划分任务以及定义任务间接口的问题，将面向对象的概念引入到嵌入式软件开发过程中，并阐述如何将这两种方法有机地结合起来。对于多数实时系统开发人员关心的系统性能问题，9.3 节详细分析主要的影响因素，以便为进一步开展相关研究打下基础。

9.1　嵌入式系统开发模式

9.1.1　嵌入式系统开发模式概述

在开发普通应用系统（如数据库管理系统、办公自动化系统）时，程序员只需要考虑软件系统的功能设计，而硬件部分直接根据软件需求购买即可。但这种相对简便的开发方式对嵌入式系统难以实用。

嵌入式系统开发的最大特点就是需要软硬件综合开发，这有两方面的原因：

一方面，任何一个嵌入式产品都是软硬件的结合体。在台式机（如 PC）上，可以认为构建的一个文件管理系统就是一个产品，因为它可见。但在嵌入式系统领域，用户看到的是一个个硬件产品，起核心控制作用的嵌入式软件是不可见的。

另一方面，一旦嵌入式产品研制完成，软件就已固化在硬件环境中，用户不能对该软件系统进行任意修改（预留的扩展接口除外）。也就是说，嵌入式软件是针对相应的嵌入式硬件开发的，是专用的。

例如，日常使用的自动洗衣机就可被看成是一个典型的嵌入式产品。对用户而言，它是一台洗衣机，其核心是用户不可见的自动控制系统，其任意部件是不可能被用户修改的。

上述特点决定了嵌入式应用的开发方式不同于传统的软件工程方法，其开发过程如图 9－1所示。

不难发现，嵌入式系统开发与通用系统的软件开发有很大差别。图 9－1 中的各个术语叙述如下：

● 系统定义　该阶段采用的方法与通用软件工程中的含义一致，是系统生存周期中最简

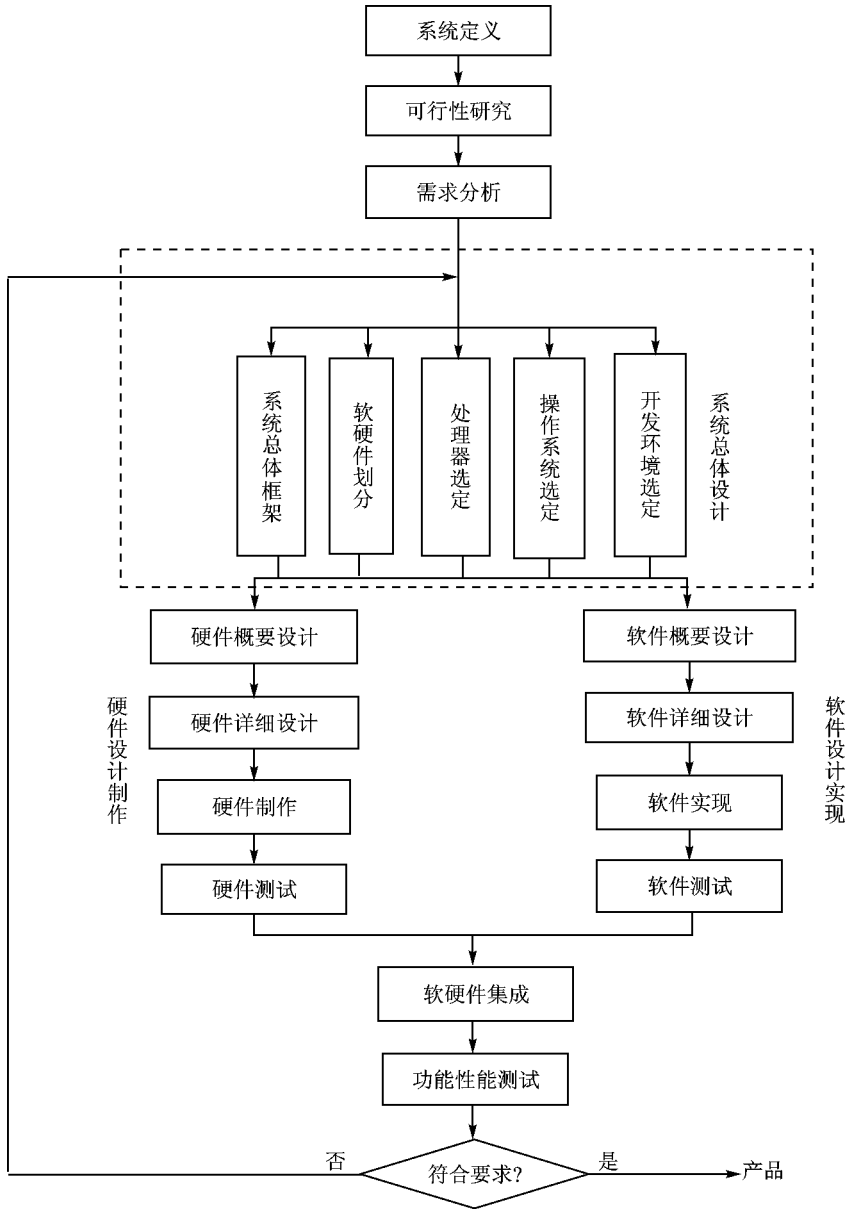


图 9－1　嵌入式系统开发过程

短的阶段，所提供的报告通常比较粗略。

- 可行性研究　用于确定是否存在行之有效的方法来解决上一阶段定义的关键问题，从而决定该系统有无研制的必要。
- 需求分析　在解决必须做什么时，除确定功能需求外，还需要确定性能需求、环境需求等。需求分析的结果是提交需求分析报告（系统规格说明），包括系统功能模块图。到本阶段为止，嵌入式系统开发与通用系统软件开发的区别还不大。
- 系统总体设计　这是整个嵌入式系统的总体设计。它需要确定嵌入式系统的总体构架，从功能实现上对软硬件进行划分；在此基础上，选择嵌入式系统硬件实现的核心——处理器，同时根据系统的复杂程度确定是否使用嵌入式操作系统，以及选用哪种操作系统；此外，还需要选择系统的开发环境。

  本阶段应该提供系统总体设计报告，推荐一个基本的软硬件配置方案（包括系统中各模块间的接口关系）。在此阶段之后，系统设计就分成软件和硬件两个方面分别进行。软硬件划分是一项很精细的工作，常常需要通过反复比较和权衡利弊才能最后选定，且划分结果对软件的设计与实现影响较大。比如系统中拥有的语音图像压缩、解压功能，可以用软件实现（软解压），也可以用硬件实现（硬解压），两种方案在成本、性能等方面差别极大。

  本阶段是整个设计过程中最基本的一环，也是最重要的阶段。它决定了此后软硬件的设计走向以及系统测试的方式和环境。
- 硬件概要设计　在此阶段，针对系统总体对硬件部分的描述，进一步确定各功能模块的详细特性（如存储器大小、LCD 分辨率等）、模块间关联的详细定义以及所选择的总线电路等。本阶段必须提交一份详细的硬件功能框图，包括每个模块以及它们的输入/输出。
- 硬件详细设计　选定实现硬件功能的各个具体的器件（包括型号、规格、封装等），设计相应的周边电路，得到符合系统需求和硬件概要设计的电路原理图，进一步生成实际的 PCB 图。
- 硬件制作　根据详细设计所得的 PCB 图加工出印制电路板（PCB），再焊接/装配必要的芯片和元器件（如 CPU，LCD 等），形成目标硬件；然后调试修改该目标硬件，直到基本上无错误发生。
- 硬件测试　对制作出的硬件系统进行功能、性能等方面的测试和修改，使其达到硬件设计目标。
- 软件设计实现　这部分的开发过程与硬件设计制作并行、交互进行，所完成的任务是软件概要设计、详细设计、实现和测试。
- 软硬件集成　将测试完成的软件系统装入制作好的硬件系统中，进行系统综合测试，验证系统功能是否能够正确无误地实现；最后将正确的软件固化在目标硬件中。本阶段的工作是整个开发过程中最复杂、最费时的，特别需要相应的辅助工具支持。

● 功能性能测试　测试最终完成的系统功能和性能是否满足需求。若不能满足，在最坏的情况下，可能需要回到设计初始阶段（系统总体设计）重新进行。

需要注意的是：到了后面的阶段，不应对硬件设计作过多修改，主要应该通过修改软件功能来改善或调整系统功能。

在本节后面，将分别对前面提出的处理器及硬件开发平台的选定、操作系统选定和开发环境选定等问题作进一步的阐述。

软硬件协同开发

在前面已经介绍了嵌入式系统开发的特点，即软硬件一体化。但是，目前通常的开发方法是分离了软件开发与硬件开发的，如图 9－2 所示。

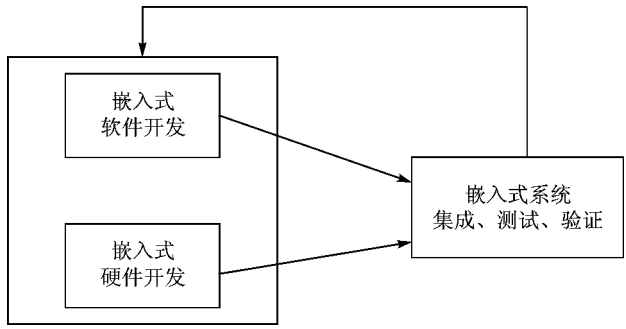


图 9－2　通常的嵌入式系统开发

这种开发过程可描述如下：

① 需求分析；

② 系统总体设计，包括总体框架、软硬件划分、处理器选型、操作系统选型及开发环境选型等；

③ 软硬件分别设计、开发、调试和测试；

④ 系统集成，即软硬件集成；

⑤ 集成测试；

⑥ 若系统正确，则结束开发，否则继续进行；

⑦ 若出现集成错误，则需要对软硬件分别验证和修改；

⑧ 返回④，继续进行集成和测试。

虽然在系统设计的初始阶段考虑了软硬件的接口问题，但是由于软硬件分别开发，各自部分的修改和缺陷是很容易导致系统集成出现错误的。由于设计方法的限制，这些错误不但难以定位，而且更重要的是对它们的修改往往会涉及整个软件结构或硬件配置的改动。

为避免上述问题，一种新的开发方法应运而生——软硬件协同开发，如图 9－3 所示。

这种开发过程可归纳为：

① 需求分析；

② 软硬件协同设计；

③ 软硬件实现；

④ 软硬件协同测试和验证。

这种方法的特点在于协同设计(Co－design)、协同测试(Co－test)和协同验证(Co－verification)。它充分考虑了软硬件的关系，并在设计的每个层次上给予测试验证，使得尽早发现和解决问题，避免灾难性错误的出现。

此外，由于减少了系统集成这一重大环节，所以整个系统的开发效率大大提高，可以更好地适应嵌入式应用系统开发的需求。

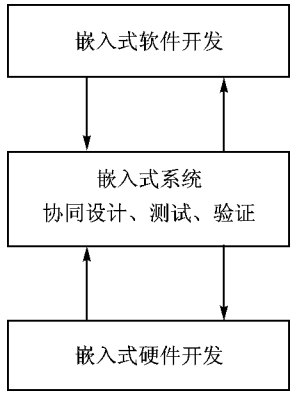


图 9－3　软硬件协同开发

9.1.2　处理器及硬件开发平台的选定

对于嵌入式应用系统或产品的开发，处理器及硬件平台的选定是非常重要的。一个好的嵌入式系统需要事先完善的硬件规划才能达到所需要的功能条件。嵌入式系统开发人员必须要从众多的嵌入式微处理器中选择一种最适当的产品作为嵌入式系统的控制核心，才能够兼具低成本和高性能的优势。

选择时应该面向应用，以处理器为主，结合考虑硬件平台的情况。主要考虑的因素有应用的类型、I/O 接口、主频、功耗、所支持的存储器类型、总线、价格、封装、产品的生命力和厂家的实力、技术支持以及第三方软件的支持。

1. 应用的类型及 I/O 接口

嵌入式系统广泛地应用于消费电子、通信、汽车、国防、航空航天、工业控制、仪表和办公自动化等领域。不同的领域所需要的处理器类型是不同的。

当今嵌入式微处理器发展的一个重要趋势是面向不同行业应用的特点进行开发，并且多采用 SOC 技术开发，也是为了应对迅速变化的行业需求，更快地推出多样化的微处理器，满足甚至引导市场的需要。比如应用在 2.5G，3G，结合了 PDA，Internet 和多媒体数据处理及其他相关功能的智能型移动电话，这些嵌入式系统需要强大的嵌入式微处理器来满足多媒体、数据通信和移动商务等应用需求，因此通常集成了 ARM9 CPU Core、Cache、MMU、DSP、部分 RAM、主存储器接口、外存接口、电源管理、LCD 控制器、多媒体编解码和 USB 等丰富的 I/O 接口。

2. 主频及功耗

处理器的主频越高，运算速度越快，越能满足复杂计算或数据处理的需要；但是主频越高的处理器功耗也相对较高，而且价格昂贵。所以还是应该根据应用的需要，选择运算速度够快且性价比高、满足功耗要求的处理器产品。

3. 对不同类型存储器的支持

嵌入式设备中的存储器种类很多，分为高速缓存 Cache、主存（片内和片外）和外存。其中，Cache 内置于处理器内，主存的类型可以有 Nor Flash，EPROM，E^2 PROM，PROM，SRAM，DRAM 和 SDRAM 等，而外存的类型可以有 NandFlash，DOC，CF，SD 和 MMC 等。

不同的嵌入式应用决定了系统对存储器的需求，比如像数码相机这一类的产品，就要求支持大容量的外部存储器，因而所选用的处理器要能支持这一类的设备；又比如某些计算量大的系统，需要处理器的运算速度够快，此时对处理器内置的 Cache 及主存的容量要求就比较高。

一般嵌入式微处理器都集成了主存储器和外存储器控制器，以便能有效地连接不同种类的存储器；如果没有相应的接口，还可通过一些扩展接口与系统连接。所以在选择嵌入式微处理器时，应根据应用需要考虑其所具备和支持的存储器类型及容量。

4. 封　装

所谓封装是指安装半导体集成电路芯片用的外壳。它不仅起着安放、固定、密封、保护芯片和增强导热性能的作用，而且还是沟通芯片内部世界与外部电路的桥梁。芯片上的接点用导线连接到封装外壳的引脚上，而这些引脚又通过印刷电路板上的导线与其他器件建立连接。因此，封装对 CPU 和其他大规模集成电路都起着重要的作用。新一代 CPU 的出现常常伴随着新的封装形式的使用。

芯片的封装技术已经历了好几代的变迁，从 DIP，QFP，PGA，BGA 到 CSP 再到 MCM，技术指标一代比一代先进，包括芯片面积与封装面积之比越来越接近于 1、使用频率越来越高、耐温性能越来越好、引脚数增多、引脚间距减小、质量减小、可靠性提高和使用更加方便等。

选择封装形式主要应考虑芯片面积与封装面积之比、装配面积、装配成功率和引脚数目等因素。嵌入式系统设备通常要求体积小的 CPU，目前用得较多的还是塑料四边引线扁平封装 PQFP（Plastic Quad Flat Package）和球栅阵列封装 BGA（Ball Grid Array）等形式。BGA 封装比 PQFP 先进，而 PQFP 通常比 BGA 便宜。

更先进的封装形式有芯片尺寸封装 CSP（Chip Size Package 或 Chip Scale Package）、多芯片模式 MCM（Multiple Chip Model）、系统级芯片 SOC（System On Chip）和电脑级芯片 PCOC（PC On Chip）。相信随着 CPU 和其他 VLSI 电路的不断进步，封装形式也将有相应的发展，而封装形式的进步又将反过来促进芯片技术向前发展。

5. 产品生命力和厂家的实力、技术支持及第三方软件的支持

当今在嵌入式领域，芯片设计制造、嵌入式操作系统/开发工具供应、应用解决方案供应及嵌入式设备开发制造等各个环节构成一个产业链，成功不单是某个企业或厂商的事情，而应该多方分工合作，形成多赢的创业与发展模式。

嵌入式领域的应用需求日新月异，所以每一款产品的生命力是有限的，开发商应该为其产品的多样化及升级换代做好准备。

因此，在选择处理器或硬件平台时，应考虑产品本身的生命力和厂家的实力、技术支持及第三方软件的支持情况。以 ARM 嵌入式微处理器为例，它赢得了众多知名芯片厂商如 Samsung，Intel，TI 等公司的支持，开发出了从低端到高端的各个系列的、支持不同应用领域的产品。对其提供支持的第三方软件及软件开发商也是相当多的。第三方软件比如嵌入式操作系统 Nucleus＋，μCOS，pSOS，VxWorks，RTXC，OSE，EPOC，Linux，Windows CE，Palm OS，JavaOS 以及国内的 DeltaOS 等；软件开发商包括 ARM，Cygnus，Green Hills，ISI，Metaware，Microsoft，Microware，Wind River 及 CoreTek 等公司。这一切使得嵌入式应用产品开发的选择余地更大了。如果选择在这样的平台上进行开发，除了能开发多样化的产品外，还有一个明显的好处就是能够有一个长期的技术积累，保护软件（包括操作系统、中间件及应用）开发商在人力资源上的投资。

如果嵌入式微处理器厂家的实力很强，包括其技术核心竞争力、产品制造规模（与成本有关）、合作伙伴、商业模式和销售渠道等，那么对于中小型的嵌入式软件及应用产品开发商来说是有裨益的；另外，及时有力的技术支持也对产品的快速上市和获得成功提供了保障。

6. 硬件开发平台的选择

选定处理器之后，硬件平台的选择或设计就相对容易得多，包括对内存及外围存储器件、输入/输出接口及设备等几项主要内容。为加快开发速度，赢得市场，特定应用的软硬件一般是同步开发的。为方便开发，一般都会在软件开发的早期阶段选择硬件评估板。该硬件评估板与所设计的嵌入式系统接近，至少嵌入式微处理器和部分外围接口一致。硬件开发就可以此为基础，在该评估板上建立开发环境后，软件实现就可以在该环境下进行。这样，硬件和软件的开发就可以做到很大程度的并行。

9.1.3　操作系统选定

1. 选择嵌入式 OS 的必要性

是否所有的嵌入式系统都必须在最优秀的嵌入式 OS 支持下才能工作得最好？不是。有很多简单的嵌入式产品，根本无需复杂的嵌入式 OS 的支持，比如循环轮询系统。但是对于复杂的应用，可采用基于嵌入式 OS 的多任务软件结构，能够将复杂的问题分而治之，提高开发效率。嵌入式 OS 使得用户可以更快地开发产品。它要求一些额外的开销，但是随着技术的进步，这种开销正在减少。

在进行嵌入式应用开发的时候，市场上可能已经有众多的嵌入式 OS 可供选择。但是，在实际应用中不同的应用对嵌入式 OS 的需求也有所不同。如有的实时应用强调时间上具备严格的要求，而功能上却只需要简单的处理；有的应用则要求系统具有较为复杂的功能，而对实时性要求并不苛刻。所以不能简单地凭借哪一种嵌入式 OS 的功能强大就作出判定，应该确定选择嵌入式 OS 的具体用途和目标是什么。仔细思考下面描述的问题，也许会有所帮助。

- 是自建还是购买,或是使用开源软件;
- 对嵌入式操作系统的功能、性能要求;
- 与硬件平台和开发工具的关系;
- 产品所属的应用领域是否有行业的标准或限制;
- 技术支持;
- 版税或服务费的问题。

下面就对这些因素作具体阐述。面对这么多的问题,要决定对应用来说哪条或哪些准则是真正重要的。

2. 自建、购买或使用开源软件

一些嵌入式工程师喜欢从零开始开发应用系统,这一点并非十分可取。如果决定要使用嵌入式 OS 进行应用开发,则在大多数情况下,利用一种现成的嵌入式 OS 是较好的选择,这也是重用他人的工作成果以达到目标的第一步。

但也不是在所有的情况下都是这样。在某些时候,自建一个嵌入式操作系统也许更为恰当。比如在性能绝对至关重要的场合,自建一种嵌入式 OS 虽然代价较高,但可以换取性能的明显提升。另外,某些特定的工业(比如医疗设备、安全系统等)对软件有着特定的规则或要求,现成的操作系统满足不了这些要求,这时也只能选择自建一种嵌入式操作系统。

有些供应商会为用户提供其产品的源代码,而有的却仅仅提供目标代码。使用没有源代码的嵌入式操作系统可能会令人不安,其实这两种方式都能开发出优秀的产品。如果对源代码做了重大改动(这有些违背购买现成操作系统的初衷),则需要足够的测试,验证没有引入新的问题。

开源软件的使用已逐渐成为一种潮流,毕竟在这里可以获取很多的资源,其商业模式也成为吸引最终产品厂商的亮点。但是开源并不意味着一切都是免费。如果想让产品尽快上市并且有更好的延续性和生命力,还需要获得有力的技术支持和服务。

3. 对嵌入式操作系统的功能、性能要求

(1) 基本功能要求

所选择的嵌入式操作系统应该在功能上满足应用开发的需要,所以最基本的应考虑核心功能的完整度,比如:该操作系统需要动态地创建和删除任务吗?一个任务能同时等待多个事件吗?任务有多少优先级?调度算法、任务同步机制的需求是什么?应该提供哪些系统函数调用?

很难预料在整个应用的设计过程中需要操作系统的哪些服务。除核心功能外,还要考虑网络功能、软件组件和设备驱动程序等。一般来说,商用操作系统的很多特性可以实现用户想要的大多数功能。

(2) 网络功能

在“无所不在的计算机”时代,更多的嵌入式系统需要与各类网络连接,因此需要具有网络

功能，配备一种或多种标准的网络通信接口。

针对外部联网要求，嵌入式系统需要 TCP/IP 协议簇软件支持。

针对内部联网要求，新一代嵌入式系统还需具备 IEEE 1394，USB，CAN，Bluetooth 或 IrDA通信接口。IEEE 1394 提供快速串行通信的能力。在此作个比较：一般的 UART 串行通信达到的速率是 115 kbit/s，USB 可达到几十 Mbit/s，而 IEEE 1394 则可达到 400 Mbit/s 乃至上 Gbit/s 的通信能力。所以它用来作为多媒体数据传输最为适合，可以用在多媒体播放、剪辑或者是实时压缩传输等应用中。而 CAN 提供了汽车和工业应用中越来越需要的片上串行通信能力，在工业自动化和马达控制领域获得了广泛的应用，它还是汽车动力总成网络的行业标准。为支持这些接口设备，也需要提供相应的通信组网协议软件和驱动软件。

为了支持网络交互的应用，还需内置 XML 浏览器和 Web Server。

(3) 软件组件和设备驱动程序

在互联的基础上，嵌入式系统需要通过各种标准增强互操作性，应用开发者要依赖于他人开发的组件，包括通信协议、服务、中间件或者其他组件(如 TCP/IP，HTTP，Ftp，Telnet，SNMP，CORBA，JAVA 和图形等)。同样，在设计中用到现成的板卡或 IC 时，还要确定是否可以得到设备驱动程序。

根据操作系统供应商提供这些组件或驱动程序的不同方式，它们既可作为操作系统的一部分，也可作为选配组件。另外，这些内容也可以从第三方供应商那里获得。在选择嵌入式操作系统时，要弄清楚该操作系统中具体集成了哪些组件。如果操作系统供应商为相关联的设备驱动程序提供了良好的架构及范例，用户还可以利用它们写出自己的周边设备驱动程序，以便和操作系统的设备管理功能结合起来。

(4) 内核要求的存储器大小

嵌入式操作系统可以装入到很小的内存空间中。尽管如此，当供应商给出一个内核要求的最小存储器大小时，很重要的一点是了解这个最小要求的内核中包含了些什么。最小的内核往往支持很少的特性(功能)，而典型的配置可能产生大得多的内核。如果用户的设计非常在意 RAM 或 ROM 的大小，一定要澄清这个问题。有时供应商会提供一份详细的列表，说明创建包含不同服务的内核分别需要多大的 RAM 和 ROM。

(5) 时间性能

对实时性有较高要求的项目来说，要了解嵌入式操作系统的各项指标以及它们对应用系统的影响，然而这并非很容易。当比较供应商提供的 benchmark(测试基准)时，用户要明白以下问题：其测试内容是什么？使用的是什么样的测试平台？硬件的配置怎样？微处理器的时钟频率是多少？使用的是什么存储系统？存储器访问使用了几个等待周期？只有弄清楚这些问题才能对不同的操作系统性能作公平的对比。

(6) 其　他

如果需要文件系统和数据库，则其设计必须是有效率且可以预测的，包括占有空间的程

度、数据删除时的回收性等，这对寸土寸金的存储空间来说是相当重要的。此外，可能还需关注如下问题：

① 图形界面效率及函数库的完整性　绘图的函数及算法必须具有很高的效率，另外还可以通过硬件的方式达到更高效率的运算。

② 电源管理机制　特别是手持设备的电源管理，要尽可能省电，除了硬件配置之外，软件的算法、系统的架构及驱动程序的设计都会影响到省电机制。

4. 与硬件平台和开发工具的关系

操作系统的选择与硬件平台和开发工具都有关系。

由于在选择操作系统之前已经选定了处理器及相关的硬件平台，所以，应考虑即将选择的操作系统是否支持所选择的硬件平台；如果不支持，则需要开展移植的工作。移植工作包括以下两大方面的内容：

① 不同类型嵌入式微处理器之间的移植　这涉及到操作系统中与 CPU 紧密相关的、通过汇编语言实现的功能部分，如任务上下文切换、时钟和中断等。最主要的，由于处理器指令集的变化，需要采用适宜的编译器来产生新的软件库。

② 同类型微处理器但不同类型的硬件板之间的移植　这涉及到对硬件接口及设备驱动程序的移植或改造。

在选定嵌入式操作系统之前还应该确认供应商支持哪一些工具，换句话说，就是确认工具与操作系统之间能否协调匹配。有些操作系统供应商提供给用户集成开发环境，该环境是与其操作系统配套的，可能具有如下好处：

① 提供应用工程创建和管理的功能，支持构建基于特定操作系统的应用框架，并且可以把操作系统提供的应用编程接口(API)嵌入到应用框架中。

② 提供特定操作系统的剪裁与配置功能，便于应用开发。

③ 由于掌握自己所开发的操作系统的内部机制，因此供应商可以提供一些辅助开发工具，如给调试器赋予一些高级调试功能(任务级调试、观测并修改操作系统对象的状态和行为)，更便于查找错误，而这些错误(比如死锁、忘了释放信号量等)用某些第三方的调试器难以发现。

④ 提供与该操作系统配套的应用逻辑分析工具、覆盖测试工具等。比如应用逻辑分析工具，它在不打断被测应用程序运行流程的基础上，对程序运行中的相关信息进行采集和分析，然后通过真实地再现程序运行的逻辑流程和分析程序运行数据，来帮助用户优化系统设计和解决出现的一些问题。具体来说，通过这种工具可以做到以下几点：

- 真实地再现程序运行流程；
- 发现系统中的死锁以及由于软件原因造成的死机；
- 发现系统中的内存泄漏；
- 帮助用户了解系统中任务的层次划分、任务的切换频度，指导对任务的合理划分；

- 指导用户进行关键路径的设计与验证；
- 显示堆栈的使用情况，指导用户合理分配任务的堆栈空间；
- 统计出任意时间段内的 CPU 使用率，指导用户合理分配 CPU；
- 显示中断发生的情况，指导用户合理地设计中断服务程序。

通过这些功能，应用开发者可以更有效地对系统进行调整，使应用程序的效率更高、健壮性更强。但是这类工具的实现可能需要与特定的操作系统结合，因为要获取某些系统信息，需要对操作系统代码进行插装，或要求操作系统提供额外的信息查询函数。

通过逻辑分析工具观测到的嵌入式应用软件运行的逻辑流程如图 9－4 所示，系统堆栈使用率分析如图 9－5 所示。

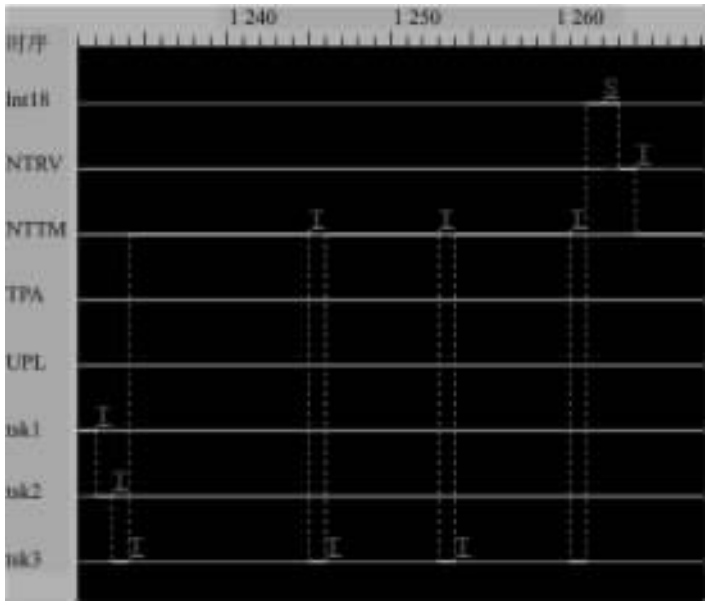


图 9－4　嵌入式应用软件运行的逻辑流程

5. 行业标准

要考虑嵌入式操作系统是否支持不同的行业标准，例如汽车电子、航空电子和数字电视等行业的标准。即使大多数开发者不需要 POSIX，也可以将其作为一个考虑因素。在一些安全关键的领域，嵌入式操作系统应该支持该领域所要求的安全标准，如 Do178B，并获得相关的认证。有些嵌入式操作系统供应商已经开始认证他们的产品。

另外，应了解该嵌入式操作系统在哪个行业表现最为出色；或者说，在既定的行业范围内，目前市场份额最大、成功率较高的是哪一款或哪一些操作系统，以及它们是否为特殊的应用领域做过优化。

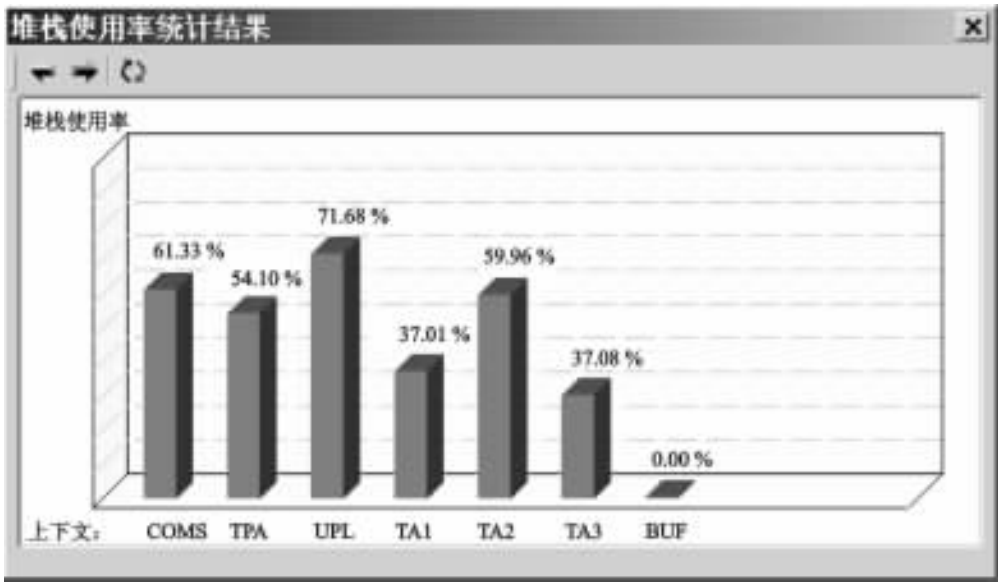


图 9-5　系统堆栈使用率分析

6. 技术支持

购买了嵌入式操作系统之后，还需要技术支持。现在是强调服务的时代，不过技术支持和服务一般都是需要一定费用的。OS 供应商提供多种支持渠道，其中包括电话、电子邮件、网站信息或现场支持；但是，要确认这些支持能持续多久。如果对嵌入式 OS 完全是新手，供应商的培训就是非常重要的。

另外，与硬件产品一样，也要考虑软件的生命力。对于嵌入式操作系统，用户也应该将眼光扩展到其公司的整个产品线，了解该产品在将来是否仍会有所发展，以及它与整个产品线的兼容性如何。

7. 版税或服务费

购买嵌入式操作系统需要考虑费用的问题，其供应商可能采取如下方式中的一种：

① 一次性收取一大笔费用，包括产品开发期间的软件使用费、服务费以及产品上市后每台设备的版税。

② 为其产品发放许可证，以便设备开发者开发最终应用产品。当应用产品上市后，按已销售的数量收取版税。

③ 对于开发阶段的收费，有的供应商是一次性买断，而有的则是以每个开发者、每种平台、每种产品和每个地点为单位收取费用。

用户在考虑能够接受的模式时，不仅要考虑总体价格的多少，还要结合资金周转情况、产品进度风险、市场风险、核心技术拥有程度以及自身的商业模式等因素，综合权衡利弊。

9.1.4　开发环境选定

“工欲善其事，必先利其器”。像任何其他工作一样，好的工具能提高工作的效率。除了操作系统外，开发嵌入式系统软件还有一个关键就是可用的工具包。

工具的选择与几个方面有很大的关系，比如它是否支持所选择的硬件平台；是否与选用的操作系统匹配；支持什么样的编程语言；生成什么格式的目标文件；是否有利于开发人员使用，提供项目管理功能以提高开发效率等。

到现在还有许多嵌入式系统的编译器还在使用命令行方式，所以开发人员必须要具备编写 makefile 文件的能力。所谓 makefile 就是可以将一个项目里的源代码、目标代码及库文件等，通过 makefile 文件里的指示（其中包括许多编译选项和参数、文件搜索路径及输出路径等），全部编译并链接出最后的可执行程序。而现在更多的集成开发环境以图形界面的形式提供灵活、方便的编译设置功能，减轻了开发者书写 makefile 文件的负担，尤其是可帮助初学者跨越书写此类文件的障碍，而把精力放在应用开发上。

1. 对硬件平台的支持

嵌入式软件开发的最大特点就是交叉开发，体现在交叉编译、交叉链接和交叉调试等方面。因此，工具一定要能支持用户事先已经选定的硬件平台，能够生成基于特定微处理器指令集的目标程序，并能对其进行调试。

目前嵌入式微处理器发展迅速，为了能充分发挥微处理器的特性，并对代码进行优化，选择配套的编译器是非常重要的。一般微处理器供应商提供了针对其产品的开发工具，这些工具在发挥微处理器特性上是很有优势的。

对调试器而言，如本书第 3 章所述，嵌入式软件的调试有多种方式。基于调试功能要求和成本的考虑，不妨采用一些能发挥处理器本身调试特性的、价格低廉但满足功能要求的调试工具。比如 ARM 系列的微处理器内部都含有 EmbeddedICE 及调试逻辑单元，就可以利用其芯片提供的调试功能（读/写内存、读/写 CPU 的寄存器、单步执行和实时执行、硬件断点及触发功能设定等），采用片上调试 OCD(On Chip Debug)工具，“用 20 %的价格完成 80 %的工作”。

除了基本的编译调试功能外，有些工具还具备性能分析和优化控制、仿真等功能，可以缩短开发进程，提高程序性能及增加可重用性。

2. 所使用的编程语言

选择开发工具时还应考虑它支持的编程语言。嵌入式软件开发中使用的编程语言主要有汇编语言、C/C++、JAVA 等。

3. 与嵌入式操作系统的关系

开发工具与嵌入式操作系统的关系已经在 9.1.3 节第 4 条中进行了描述。

9.2 实时软件分析设计方法

9.2.1 实时软件的分析设计要求

实时系统软件是指运行在实时系统硬件平台上的具有实时性的程序。由于计算机速度的加快,实时系统设计人员越来越倾向于使用软件代替硬件实现需要的功能,以利于缩短开发时间,增加自动处理能力。于是,实时软件越来越复杂,代码成千上万行。这使得如何高效、可靠地设计和开发实时系统软件成为一个重要的问题。

与其他任何基于计算机的系统一样,一个实时系统必须将硬件、软件和数据元素等集成起来,以恰当地实现一组功能和性能需求,系统设计人员必须对系统元素分配功能和性能。实时系统的问题在于如何恰当地分配性能。实时性能常常与功能一样重要,但却很难有把握地作出与性能相关的分配决策。例如:一个处理算法能满足严格的时间约束吗?还是说应该建造特殊的硬件来完成这个工作?一个购买来的操作系统能够满足进行高效的中断处理、多任务和通信的需求吗?还是说应该使用自定义的程序?与推荐的软件配对的特定硬件能够满足性能要求吗?所有这些以及其他许多问题都需要由实时系统设计人员来回答。

Everett 定义了实时软件开发不同于其他软件工程的三个特征:

① 实时系统的设计是受资源约束的。时间是实时系统的首要资源,关键是要在指定数目的 CPU 周期内完成一个定义好的任务;除此以外,其他系统资源如内存大小等,在实现系统目标时都有可能和时间进行折衷。

② 实时系统是紧凑而复杂的。尽管一个复杂的实时系统可能包含上百万行的代码,但软件中有关时间标准的代码一般只占很小一部分。这一小部分代码是最为复杂的(从算法的角度来说)。

③ 实时系统的运行常常不需要用户的参与。因此,实时软件必须能检测到导致故障的问题原因,并在对数据和控制环境造成破坏前改正这些问题。

满足系统响应时间是实时系统设计的一项重要任务。这里,先从一个实际的例子入手,建立实时性的具体概念。在该实例中,忽略了一些细节,只是希望读者能够对满足系统实时性有一个感性认识。

假定有一个移动作战系统,该系统具有对敌方目标探测、攻击和接收上级指挥部下达的命令等功能,还具有油料告警、弹药告警、自身定位(即确定所处的经纬度)等功能。其中,自身的定位是一个相对独立的子系统。这里,先就定位系统的工作过程进行描述,通过了解实时定位系统的基本工作原理,认识系统的实时性设计要求。

实时定位系统作为一个相对独立的子系统,它的功能是:在移动作战系统经过一段时间的运动之后,确定自身的地理位置(经度和纬度)。为此,定位系统需要有若干传感器,如温度计、

高度计、里程计、加速度计、角速度计和 GPS，另外还有一个陀螺仪（定位所必需的）。由这些传感器对温度、高度、里程、加速度和角速度等信息进行采样，采样的结果参数作为计算经纬度的相关数学模型的输入参数，并由这些数学模型计算出经度和纬度。

假设要求的系统响应时间为 2 500 ms，即从启动该定位系统开始，到输出结果的时间不超过 2 500 ms。

该系统的工作流程如图 9－6 所示。

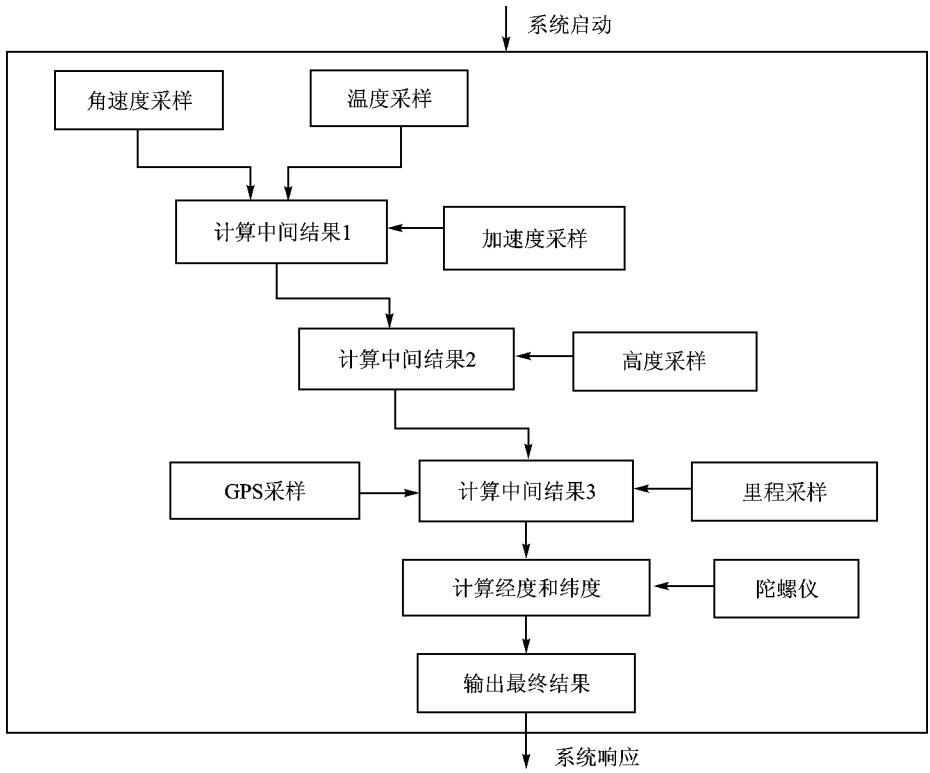


图 9－6　定位系统工作流程

说明：

- 定位系统启动后，首先进行角速度采样和温度采样，然后将采样结果送到一个计算中间结果 1 的程序；
- 中间结果 1 计算程序接收角速度和温度参数，结合加速度计的采样结果，计算出中间结果 1，并将该结果送往计算中间结果 2 的程序；
- 中间结果 2 计算程序接收中间结果 1，再结合高度采样参数，计算出中间结果 2，将该结果送往计算中间结果 3 的程序；

- 中间结果 3 计算程序接收中间结果 2，再结合 GPS 采样参数和里程采样参数，计算出中间结果 3，并将该结果送往经度和纬度计算程序；
- 经纬度计算程序接收中间结果 3，并结合陀螺仪参数计算出最终结果，即经度和纬度。

该系统的实时性能要求为：从系统启动开始，到系统输出最终结果，整个过程应该在 2 500 ms 以内完成。根据该实时性能要求以及传感器性能和计算模型的算法复杂性，分配采样时间和任务计算时间如下：

角速度采样时间　10 ms；
温度采样时间　200 ms；
加速度采样时间　20 ms；
中间结果 1 计算时间　500 ms；
高度采样时间　10 ms；
中间结果 2 计算时间　600 ms；
GPS 采样时间　30 ms；
里程采样时间　20 ms；
中间结果 3 计算时间　400 ms；
采样陀螺仪参数时间　15 ms；
最终结果计算时间　650 ms。

根据上述时间分配，由五个程序分别完成以下功能：

- 程序 1　角速度采样和温度采样；
- 程序 2　加速度采样并接收程序 1 的采样数据，计算中间结果 1；
- 程序 3　高度采样并接收程序 2 的中间结果 1，计算出中间结果 2；
- 程序 4　同时启动 GPS 采样和里程采样并接收中间结果 2，计算出中间结果 3；
- 程序 5　接收陀螺仪参数和中间结果 3，计算出经度和纬度。

系统工作过程按程序转换流程的方式，如图 9－7 所示。

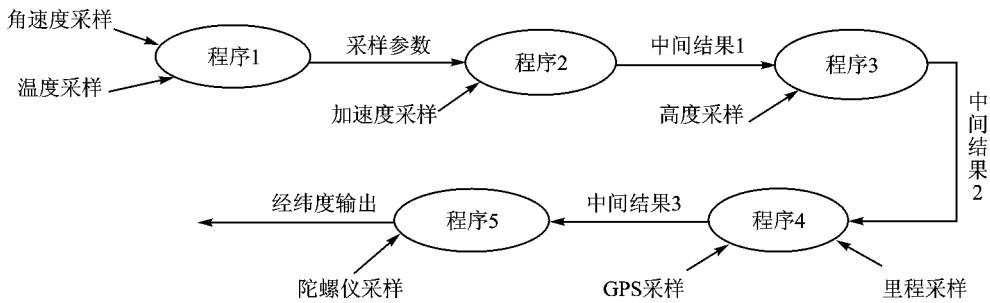


图 9－7　程序转换流程

根据前面分配的采样时间和计算时间，得出上述程序的完成时间和系统响应时间为：

- 程序 1 可同时启动角速度采样和温度采样，在 200 ms 内完成；
- 程序 2 在 520 ms 内完成；
- 程序 3 在 610 ms 内完成；
- 程序 4 在 430 ms 内完成；
- 程序 5 在 665 ms 内完成。

系统响应时间为 2 425 ms，满足要求。

以上分析可以概括为：采样的实时性、计算的实时性和子系统实时性。

从以上的分析可以看出，用户往往提供最原始的实时限制，如系统响应时间等；对关键的步骤，也可能会提出一些要求（比如计算某个中间结果或采样的时间）。而用户具体会细化实时要求到哪个程度，取决于用户的水平以及实际实时系统的需求。

从软件工程的角度来讲，嵌入式实时多任务应用软件也有一定的生存周期。这一生存周期过程如下：

- 需求分析与详细说明　分析用户需求，说明系统能满足这些需求的程度。
- 系统设计　任务分解，将系统分解成任务（并发进程），定义任务间接口关系。
- 任务设计　按模块方式设计每个任务，并定义出模块间接口。
- 模块构筑　完成每个模块的详细设计、编码和单元测试。
- 任务与系统集成　逐个模块连接、测试以构成任务，逐个任务连接和测试形成最终系统。
- 系统测试　测试整个系统或主要子系统，以验证功能指标的实现；为具有更强的客观性，系统测试最好由一个独立的测试小组执行。

可见同其他通用软件相比，嵌入式多任务应用软件的开发有其独特之处：在系统设计阶段，着眼于将系统划分为多个并发的任务，而非多个模块；要定义任务间的接口关系，而非模块间接口。模块的划分以及模块间接口定义则放在了任务设计阶段。嵌入式多任务应用软件的开发要比其他非多任务（或多进程/多线程）软件的开发多一个层次。由此，需要引入新的设计方法，这里就以 DARTS(a Design Algorithm for Real Time Systems)设计方法为例作较为详细的介绍。DARTS 是结构化分析/结构化设计的扩展，它给出了划分任务的方法以及定义任务间接口的机制。从这个意义上说，它吸取了并行处理中的经验。和其他设计方法一样，DARTS 也倾向于重复。

目前，在嵌入式实时系统软件设计中使用实时 UML 已成为趋势。本节还将以 Rose RealTime为例介绍基于 UML 的嵌入式软件设计过程，并通过实例将它与 DARTS 方法结合起来，体现两者互补的优势。

9.2.2 DARTS 分析设计方法

下面将以一个机器人控制器为例介绍开发过程。该机器人控制器控制六个转轴，并与数字 I/O 传感器交互作用。转轴和 I/O 由程序控制。该程序由控制面板操作启动执行，控制面板包括若干按键和一个程序选择开关，如图 9－8 所示。

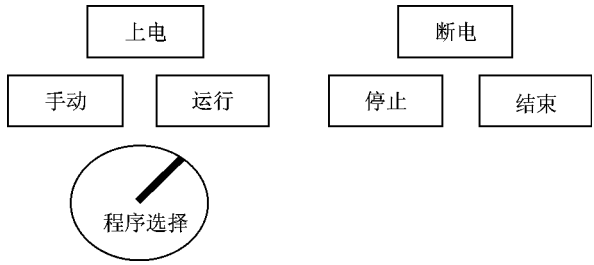


图 9－8　控制面板

按下“上电”键，系统进入了上电状态。成功地完成了上电后，系统进入了手动状态。这时，操作者可以通过程序选择开关选择程序，即旋转开关，使它指向期望的程序号。按下“运行”键，则选定的程序开始运行，系统转为运行态。程序运行中如果按下“停止”键，则程序被挂起，这时系统进入挂起态。之后，操作者可以按下“运行”键，使程序恢复执行；也可按下“结束”键，结束程序。按下“结束”键后，系统进入终止态。当程序最终终止执行时，系统返回手动状态。

1. 需求分析与说明

需求说明过程给出系统功能需求（功能、输入、输出）、外部接口需求（如用户界面）、性能以及诸如文件/数据库安全等其他要求。

实时系统中，常用状态变迁图描述系统。为此设计状态变迁图，在设计阶段可能需要进一步细化它，以便包括用户无须知道的系统内部状态。

此外，应清楚地说明操作员与系统间的所有交互作用。至此，形成一本操作手册脚本是有意义的，它为用户提供了使用该系统的操作步骤。为使系统说明更为清楚，还需要将状态变迁图与操作手册脚本相关联起来。如图 9－9 所示，引起状态变迁的用户操作已在图上标明。

2. 系统设计

系统设计说明该系统如何被分解成多个任务，如何定义任务间的关系。

下面介绍任务划分方法——DARTS 设计方法。

（1）数据流分析

在系统需求分析的基础上，以数据流图作为分析工具，从系统的功能需求开始分析系统中的数据流，确定主要功能。扩展数据流图，并分解到足够的深度，识别出主要的子系统和每个

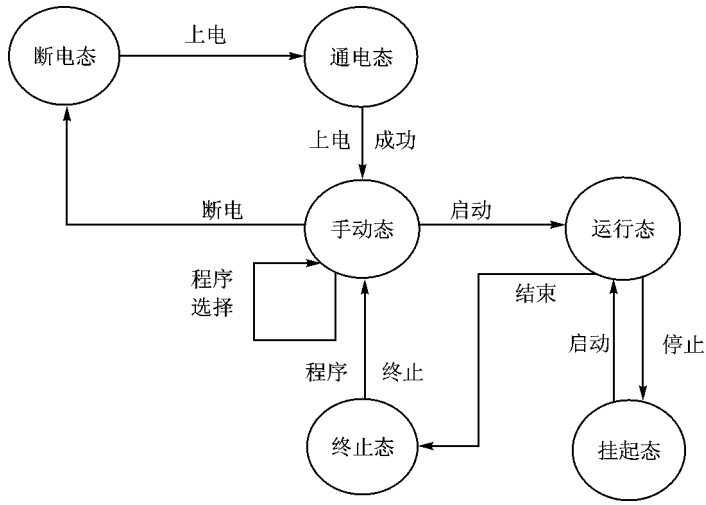


图 9-9　状态变迁图

子系统的主要成分。

每个数据流图都包含变换圈，表示系统完成的功能，箭头表示变换间的数据流动，数据存储区表示数据的存储场所。数据字典定义了数据流和数据存储区所包含的数据项。图 9-10 所示为机器人控制器的数据流图。

读入控制面板的输入并进行确认。每次按下按键，输入就被 RPI(读面板输入)读入，并转换成系统内部格式。然后，控制面板输入被传送给 VPI(面板输入有效性检查)。因为输入的有效性与系统当前的状态有关，所以要核对控制器状态转移表。为使该例子简单，假设用户的无效输入被忽略。

经过确认的控制面板输入被传送到 PPI(处理面板输入)，在那儿对其进行处理，然后传送给相应的变换，或是 IPS(解释程序各语句)，或是 OAD(输出动作轴数据)。此外，PPI(处理面板输入)还将控制面板输出数据(对应于控制面板状态灯)送到 OTP(输出到面板)。

当程序选择开关上的设置改变时，新的开关设置值被传到 PPI(处理面板输入)，它将所选程序代号更新。按下“运行”键时，通过查状态转移表，有效地控制面板输入产生一个“运行开始”信号，PPI(处理面板输入)将“运行开始”的请求传给 IPS(解释程序各语句)，然后 IPS 开始解释程序。IPS 直接执行算术和逻辑语句，而动作语句和 I/O 语句需要进一步处理。动作命令被传送到 PMC(处理动作命令)，PMC 对数据进行一些数学变换，然后将一个动作块传送给 OAD(输出动作轴数据)。OAD 把数据转换成 AC(轴控制器)所需要的格式，并将一个轴块传送给 AC。

按下“停止”键时，OAD(输出动作轴数据)不再向 AC(轴控制器)输送动作轴块；再按下“运行”键，则又恢复传送。当与一个动作轴块相对应的轴动作完成后，AC 向 RA(接收确认)

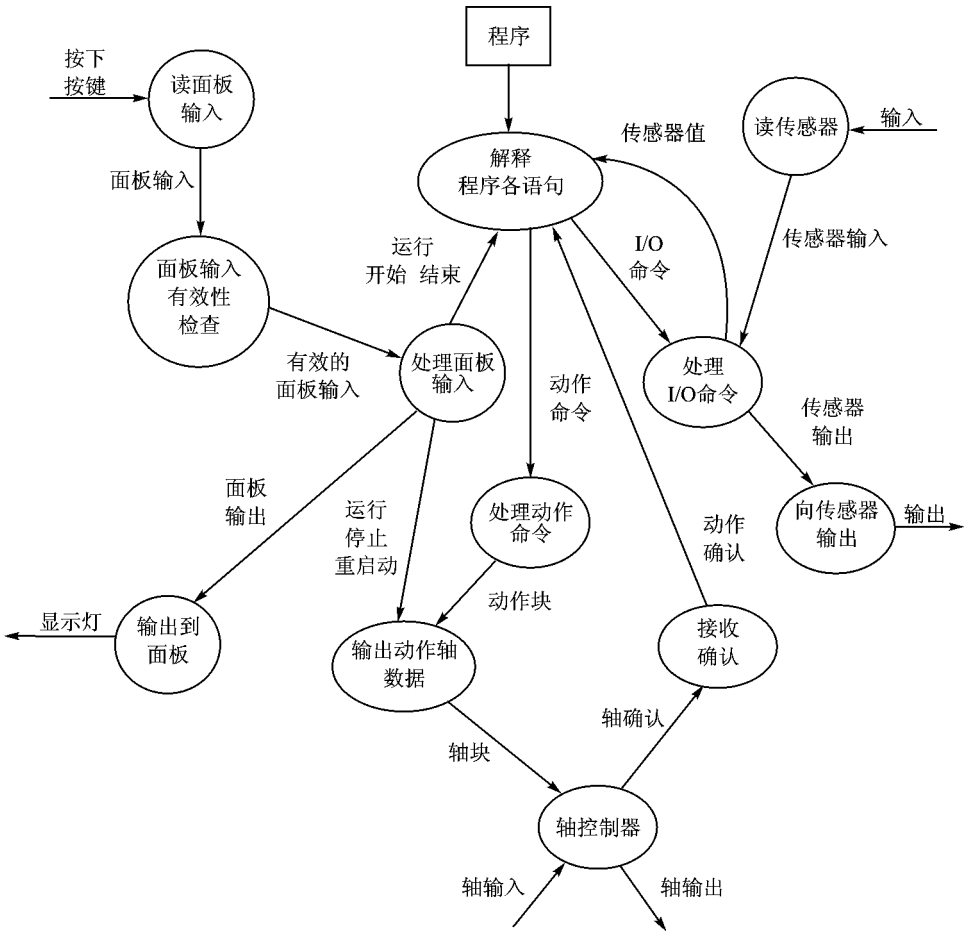


图 9-10　机器人控制器数据流图

发送一个轴确认，处理这个确认后，产生动作确认送到 IPS(解释程序各语句)。

如果是传感器 I/O 语句，IPS 就发送一条 I/O 命令给 PIOC(处理 I/O 命令)，PIOC 接收从 RS(读传感器)传来的传感器输入数据，并且将传感器输出数据送至 OTS(向传感器输出)。

(2) 划分任务

识别出系统的所有功能和它们之间的数据流后，下一步要做的是识别出并行性的功能。因此，DARTS 方法的下一步涉及怎样在数据流图上确定出并发的任务。

在将一个软件系统分解成并行任务时，主要需考虑的是系统内功能的异步性。分析数据流图中的变换，确定哪些变换可以并行，而哪些变换在本质上是顺序的。通过这种方法，划分出任务，一个变换对应一个任务，或者一个任务包括几个变换。

现在，重画数据流图，反映出各任务及任务间的接口。在重画时，用方框将逻辑上形成一个任务的变换（一个或一组）框起来，则每个方框就代表一个任务。

一个变换是应该成为一个独立的任务，还是应该和其他变换一起组成一个任务，决定的原则如下：

1）I/O 依赖性

I/O 依赖性如图 9－11 所示。如果变换依赖于 I/O，那么它运行的速度常常受限于与它互操作的 I/O 设备的速度。在这种情况下，该变换应成为一个独立的任务。

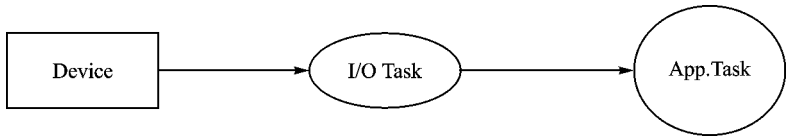


图 9－11　I/O 依赖性

- 在系统中创建多个与 I/O 设备数目相当的 I/O 任务；
- I/O 任务只实现与设备相关的代码；
- I/O 任务的执行只受限于 I/O 设备的速度，而不是处理器；
- 在任务中分离设备相关性。

2）功能的时间关键性

功能的时间关键性如图 9－12 所示。

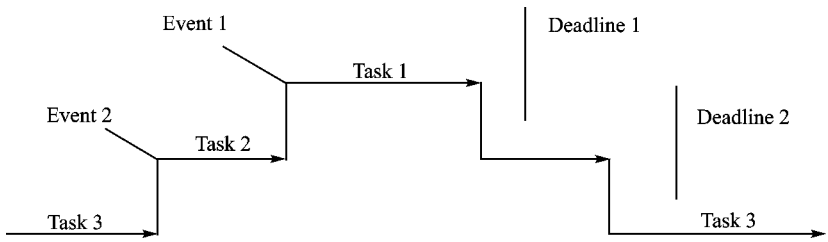


图 9－12　功能的时间关键性

- 将有时间关键性（deadline）的功能分离出来，组成独立运行的任务；
- 赋予这些任务高的优先级，以满足对时间的需要。

3）计算需求

计算需求如图 9－13 所示。

- 计算量大的功能占用 CPU 的时间多，把计算功能捆绑成任务，以消耗 CPU 的剩余时间；
- 赋予计算任务较低优先级，能被高优先级的任务抢占，从而保持高优先级的任务是轻量级的；

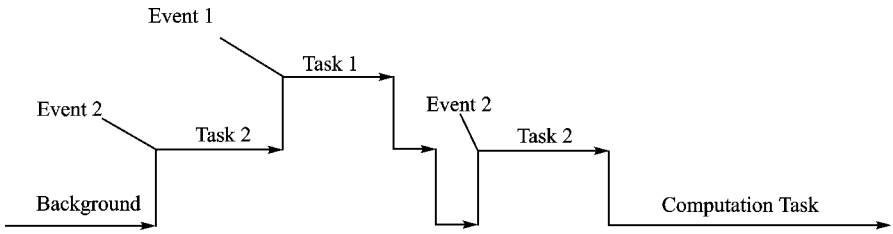


图 9-13　计算需求

- 多个计算任务可安排成同等优先级，按时间片循环轮转。

4）功能内聚

完成的功能紧密相关的变换可以组成一个任务，因为这些功能间的数据通信较多，把它们作为一个个独立的任务会增加系统的开销；反之，把每个变换都作为同一任务中一个独立的模块，不仅保证了模块级的功能内聚，而且保证了任务级的功能内聚，如图 9－14 所示。

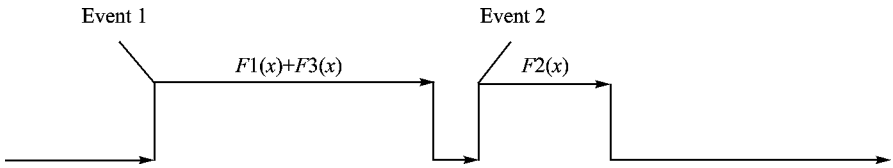


图 9-14　功能内聚

5）时间内聚

- 将在同一时间内完成的各功能组成功能组（即形成一个任务），即使这些功能是不相关的，也如此处理；
- 功能组的各功能是由相同的外部事件驱动的（如时钟等），这样每次任务接收到一个事件，它们都可以同时执行；
- 这样有利于减少系统的开销，因为减少了系统任务调度和切换的次数。

虽然时间内聚在结构化设计中并不被认为是一个好的模块分解原则，但在任务级是可以被接受的。每个功能都作为一个独立的模块来实现，从而达到了模块级的功能内聚；这些模块组合在一起，又达到了任务级的时间内聚。时间内聚如图 9－15 所示。

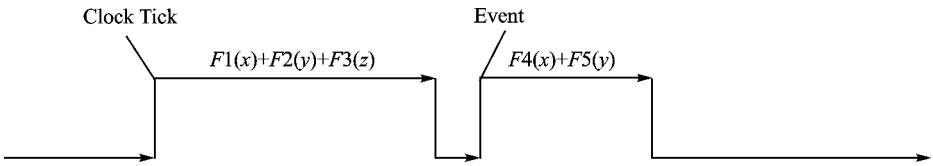


图 9-15　时间内聚

6）周期执行

- 一个需要周期执行的变换可以作为一个独立的任务，按一定的时间间隔被激活；
- 将在相同周期内执行的各功能组成一个任务；
- 频率高的任务赋予高优先级。

周期执行如图 9－16 所示。

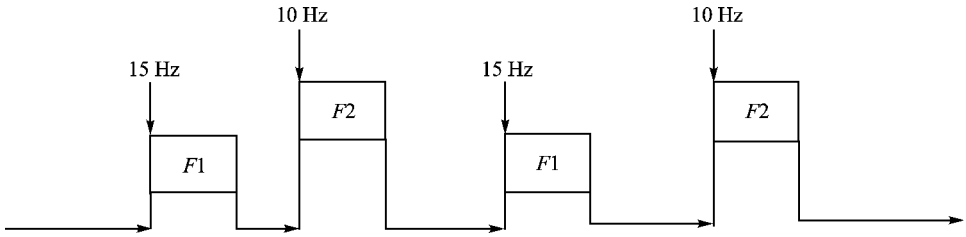


图 9－16　周期执行

（3）定义任务接口

现在，来考虑任务间的接口。在数据流图中，接口以数据流和数据存储区的形式存在。下一步进行的是格式化任务间接口。

在 DARTS 中，定义了两类任务接口模块来处理接口问题，即任务间通信模块 TCM（Task Communication Module）和任务同步互斥模块 TSM（Task Synchronization Module）。在某些实现中，这些模块对调用它们的任务来说就是操作系统的系统调用。

1）任务间通信模块

TCM 处理任务间的所有通信情况。一般情况下，TCM 包含一个数据结构，并定义对该结构的访问过程。

从原理上讲，TCM 总是运行在调用它的任务中，因而 TCM 有可能在两个任务中并发执行，所以访问过程必须提供必要的同步和互斥条件来确保数据的一致性和正确性。

TCM 利用了操作系统提供的同步原语，因此，TCM 的实现依系统的不同而不同，但它们的功能在概念上是相似的。

DARTS 支持两类不同的 TCM：消息通信模块和信息隐藏模块。

① 消息通信模块　消息通信由一个称为消息通信模块（MCM）的 TCM 进行处理。MCM 支持松耦合和紧耦合两类消息通信。

在松耦合的消息通信中，消息队列包括互斥信号量，用于互斥。事件同步用来在队列满时挂起生产者，在队列为空时挂起消费者。访问例程用来发送和接收消息，以及获取和释放消息块。此外，每个消息队列都限定了最大长度。

在紧耦合的消息通信中，队列中只含有一个元素，消息的发送和接收通过两个方向各只有一个元素的消息队列来实现，一个方向用于发送消息，另一个方向用于应答。

另外，在松耦合的通信中，任务等待一条消息，或等待一个应答到达若干消息队列中的任一个。当一条消息或一个应答到达时，任务被激活，这是通过使每一消息队列都和一个事件相连来实现的。向一个空的队列加入一条消息，会产生一个事件发生的信号，从而任务被激活。

DARTS 支持消息通信机制。松耦合和紧耦合消息通信的图形表示如图 9－17 所示。

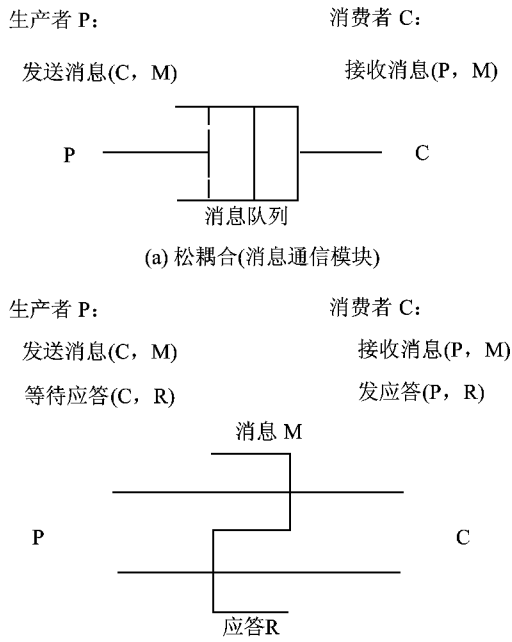


图 9－17　消息通信

② 信息隐藏模块　系统中可能有一些资源如查询数据、数据池和数据存储区等可以被两个或更多的任务共享使用，或是只读，或是可读、可写。一个称为信息隐藏模块（IHM）的 TCM 用来实现这一目的。IHM 定义数据存储区和对它进行访问的访问过程。

图 9－18 用图形表示了 DARTS 中的一个 IHM。数据存储区用方框表示，访问过程在概

图 9－18　信息隐藏模块

念上是在任务 A 和任务 B 中执行，箭头指示了任务和数据存储区之间的数据流动。

2）任务同步模块

当任务之间并不需要传送真正的信息时，事件用来实现同步的目的。目标任务等待一个事件或几个事件的发生，源任务发送事件信号激活目标任务。任务同步如图 9－19 所示。

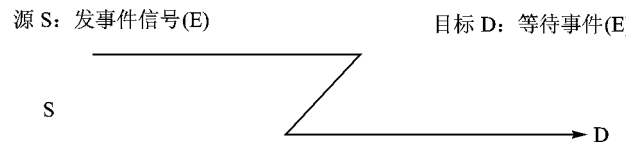


图 9－19　任务同步

3. 实例说明

根据上述原则，图 9－20 中用方框框出了各变换或变换组，各方框内的变换在逻辑上构成一个任务。

任务划分的首要原则是：直接和 I/O 设备打交道的各功能都应成为独立的任务，因为它的运行速度受制于与它互操作的 I/O 设备的速度。而 RPI（读面板输入）从控制面板接收输入，故 RPI 变换应作为一个独立的任务——CPIH（控制面板输入处理器）。类似地，OTP（输出到面板）变换也应作为一个独立的任务——CPOH（控制面板输出处理器）。

根据时间内聚的原则，VPI（面板输入有效性检查）变换和 PPI（处理面板输入）变换组成一个任务——CPP（控制面板处理器）。这样，控制面板输入在经过确认后立即被处理。

IPS（解释程序语句）变换、PMC（处理动作命令）变换和 PIOC（处理 I/O 命令）变换组成一个任务——命令解释器。这几个变换在功能上紧密相关，所以根据功能内聚的原则，把它们组合在一起。它们组成的任务在逻辑上可与 CPP 任务并行运行。按照时间内聚的原则，OAD（输出动作轴数据）变换和 RA（接收确认）变换组成一个任务——动作轴管理器（AM）。OAD 每向 AC（轴控制器）输送一个动作轴块，RA 在 OAD 输送下一块之前，必须等待一个确认。因此，让这两个任务并行执行是不可取的。

此外，这两个变换的执行速度受限于轴的速度，故只能让这两个变换组成 AM 任务。AC（轴控制器）被作为一个独立的时间关键性任务。因为它与轴频繁地互操作，所以，该任务在一个单独的处理器上运行。从程序解释器来的传感器 I/O 请求由两个任务处理。只要有输出需求，OTS（向控制面板输出）变换就被激活，所以把 OTS 变换作为一个单独的依赖于 I/O 的任务——传感器输出（SO）。而 RS（读传感器）变换周期性地扫描输入传感器，故把它作为一个单独的周期任务——传感器输入（SI）。

划分好任务后，下一步要做的是定义任务间接口，如图 9－21 所示。

CPIH 将控制面板输入排成一个消息队列，以供 CPP 接收，因此，这两个任务间的接口是一个消息队列。类似地，CPP 将控制面板输出排成一个队列供 CPOH 接收，用一个 MCM 来

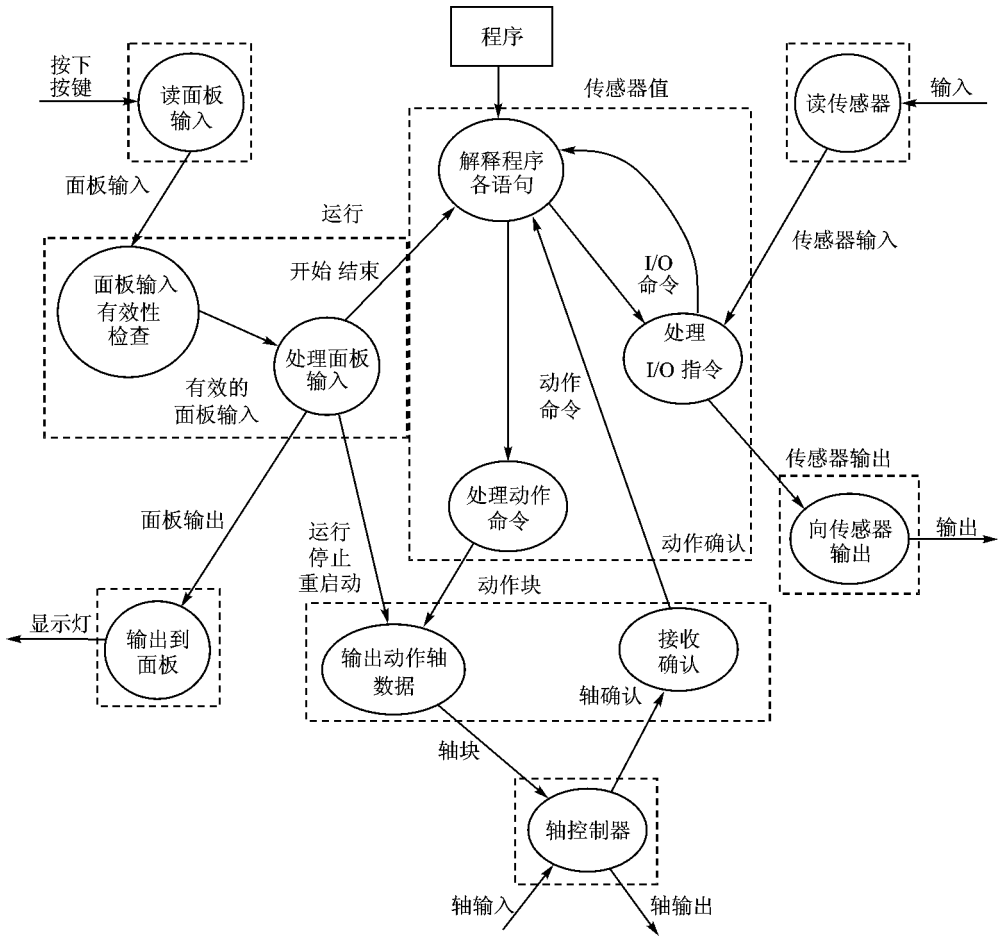


图 9－20　机器人控制器的任务划分

处理消息队列。

CPP 向命令解释器发送一条“开始运行程序”的消息，确定出要执行的程序。解释器产生一些动作块，并把它们放在动作块队列中。因为一些动作块意味着一个长的移动，而其他却是短移动，所以，解释器和 AM（动作轴管理者）之间的队列作为缓冲区使用。

当解释器读到的是非动作语句（如传感器 I/O 命令）时，解释器执行该语句之前必须等待，直到轴动作到达目标点。解释器等待一个 AM 传来的动作确认信号，表示所有的轴块已经被执行。解释器也等待一个“结束”事件信号，指示程序应该被终止。这些条件中只要有一个成立，解释器就被唤醒。

解释器的主过程完成任务同步，它处理解释器的所有同步条件。其工作过程是：解释器等

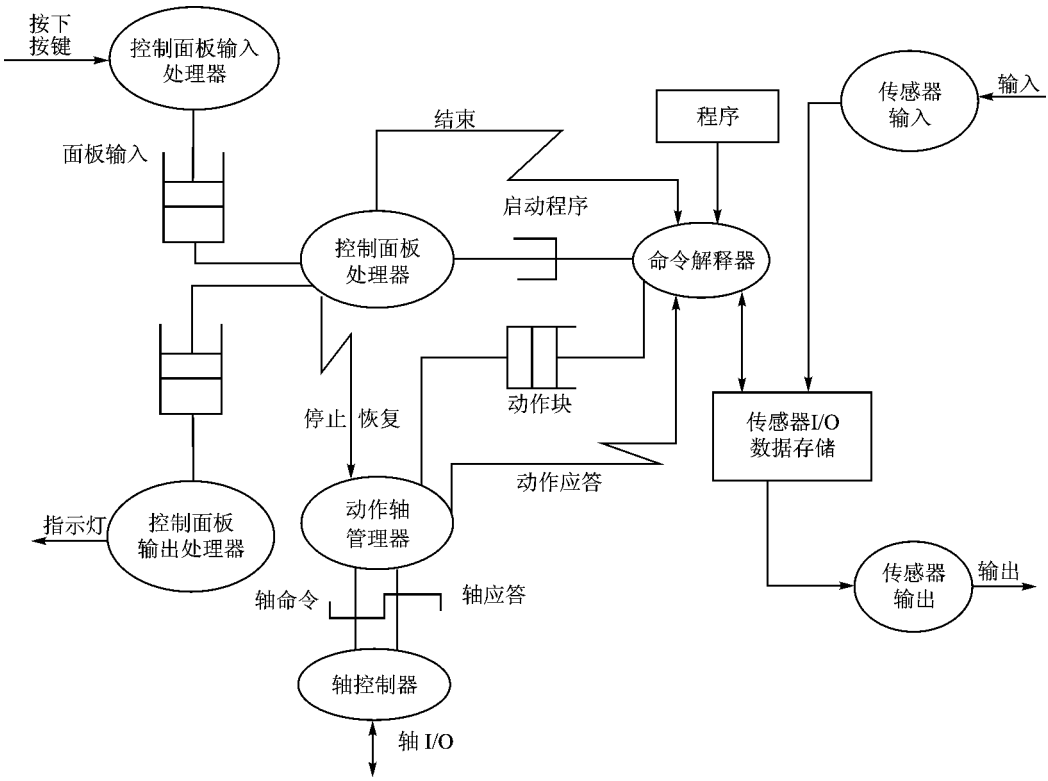


图 9－21　机器人控制器的任务结构

待 CPP 传来的“开始运行程序”的消息，在解释程序期间，解释器周期性地检查是否有“结束”信号。当解释过程被挂起时，解释器等待“结束”事件或“动作确认”事件。

AM(动作轴管理者)任务从它的消息队列中接收动作块和 CPP 传来的“停止”信号或者“恢复”信号。AM 任务的主过程完成任务同步，处理 AM 的所有同步条件。每次 AM 等待解释器的动作块时，AM 测试是否有“停止”事件信号，如果有，就等待“恢复”事件信号。如果停止的条件不成立或是得到“恢复”信号，AM 就向 AC(轴控制器)发送动作轴块并等待块完成的轴确认。AM 与 AC 间的通信属于紧耦合通信，用一个 MCM(消息通信模块)提供紧耦合通信机制。

传感器 I/O 数据存储区(SIODS)用来存储传感器 I/O 数据的当前值。如果解释器处理一条输出命令，解释器将更新 SIODS，并向 SO 任务(传感器输出任务)发信号，通知它有一个输出。SI 任务(传感器输入任务)周期性地扫描输入传感器，当传感器发生变化时，更新 SIODS。如果解释器处理一条输入命令，则 SI 任务就读取 SIODS 中传感器的当前值。因为对 SIODS 的访问是由三个任务产生的，所以对 SIODS 的访问必须通过访问过程来实现同步。

因此，由 SIODS 和访问过程共同构成一个 IHM(信息隐藏模块)。

4. 任务设计

任务间接口定义好之后，下一步要做的是建立每个任务的结构。一个任务代表一个程序序列。画出每个任务的数据流图，然后使用结构化设计法，从数据流图导出任务的模块结构图，并且定义出各模块的接口，如图 9－22 所示。

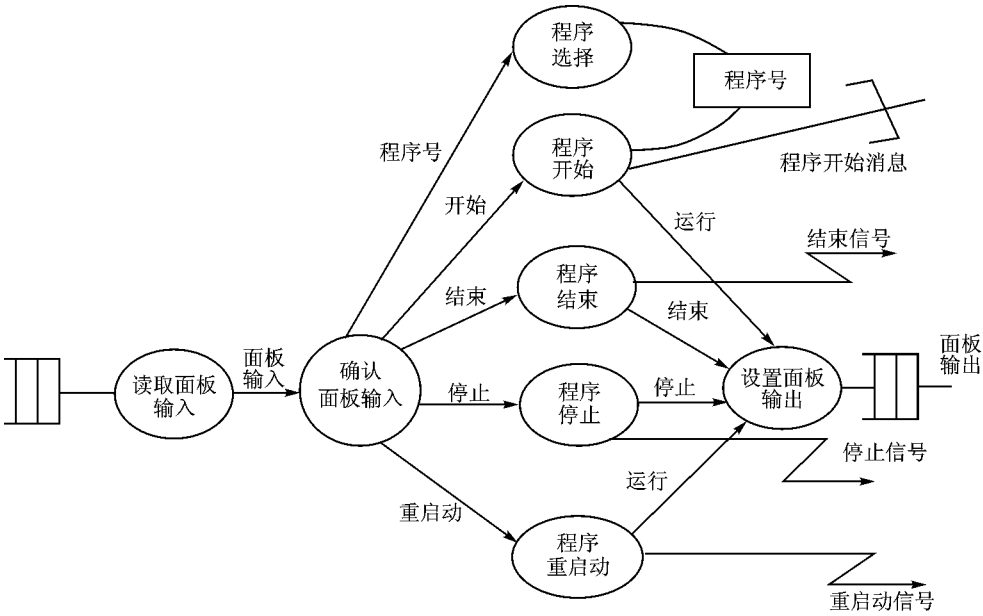


图 9－22　控制面板处理器任务数据流

为了说明上面的方法，来看一个特殊的任务——CPP 任务。如图 9－20 所示，CPP 任务由 VPI(面板输入有效性检查)变换和 PPI(处理面板输入)变换组成。因此，CPP 任务的数据流图(见图 9－22)是这两个变换的扩展。

在图 9－22 中，GPI(读取面板输入)从控制面板输入消息队列中接收输入消息，输入传送到 VPI(确认面板输入)，由 VPI 检查在当前系统状态下，输入是否有效。

假设输入有效，则输入被送到相应的变换，完成相应的动作。例如，SP(停止程序运行)变换发送一个“停止”事件信号，熄灭控制面板“运行”状态灯，点亮“停止”灯。控制面板输出被传送到 PPO 变换(设置面板输出)，由 PPO 变换将控制面板输出消息队列，以供 CPOH 使用。

CPP 的模块结构如图 9－23 所示。其主要过程(也称为 CPP)是一个控制模块，调用 GPI 读取消息。如果没有消息，则在消息到达之前，任务被挂起；当接收到输入消息时，VPI 被调用。VPI 以控制面板输入作为调用时的输入参数，返回为“有效”或“无效”的状态。

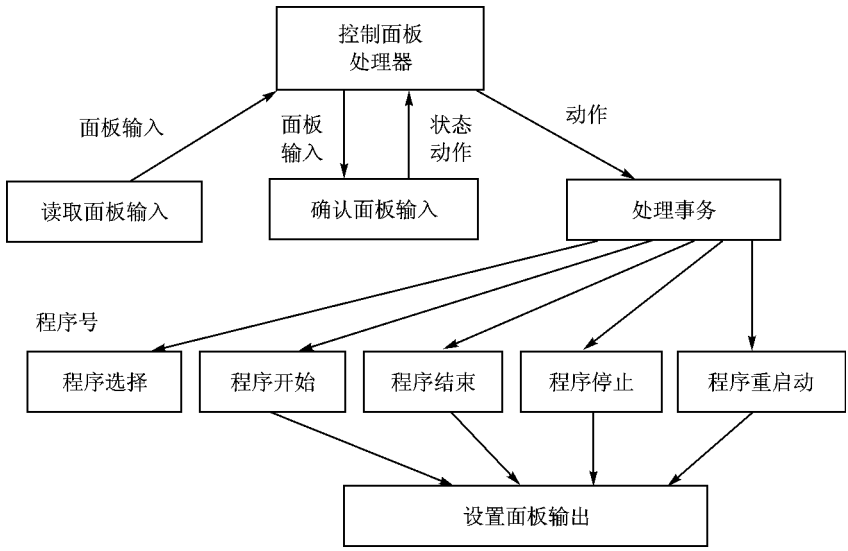


图 9－23　控制面板处理器任务的模块结构

5. 模块构筑

系统和任务设计完成后，进行每个模块的详细设计；详细设计完成后才可开始编码。但是在单元测试前不必编完模块全部程序，可以分阶段编码和测试。尽管如此，该模块的详细设计也应该先完成，这样可保证设计工作一气呵成，避免系统以非结构化方式形成。

6. 任务与系统集成

模块逐个连接、测试以构成任务，任务被逐个连接和测试形成最终系统。

系统可分两步集成：在宿主机上模拟集成（软集成）；在目标机上集成。

9.2.3　基于 UML 的分析设计方法

使用 UML 来设计和验证软件是软件工程发展的一大趋势，现在已经得到业界的普遍接受。它是由 Grady Booch（Booch 方法）、Ivar Jacobson（OOSE 方法）和 James Rumbaugh（OMT 方法）三者共同创建的一种建模语言，并且结合了许多其他经过实践检验的面向对象设计方法。UML 在 1997 年得到 OMG（对象管理组织）的采纳而成为国际标准，其发展历程如图 9－24 所示。

正如程序员普遍使用高级语言替代低层的汇编语言一样，软件开发工具会发展到更高的阶段，这就是使用可视化的开发工具直接在模型和设计层面进行开发和验证。实时系统具有特殊的复杂性，如果采用这种方法，则会获得更高的开发和维护效率，也提高了设计级的可重用性。

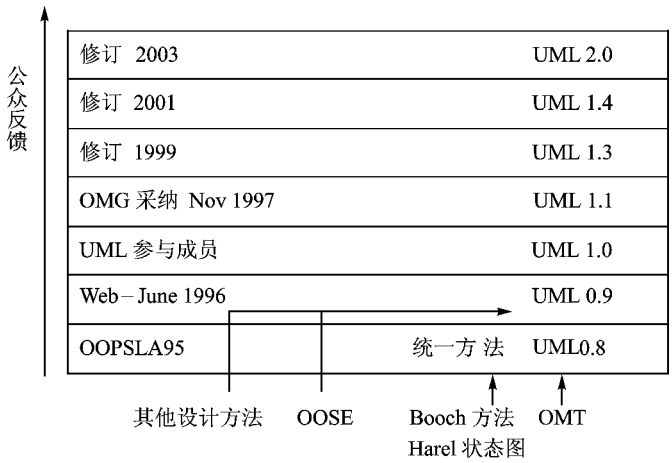


图 9－24　UML 的发展历程

UML 有八种图形表达法，分别表示以下的内容：

- 需求　用例图；
- 静态结构　类图——对象类型及它们之间的关系；
- 对象行为　状态机图——对象的生命周期；
- 对象间行为　活动图、序列图和协作图——对象间的控制流，用于捕捉系统级的行为；
- 物理实现结构　组件图和部署图——软件模块及其在硬件节点上的部署。

虽然以上表示法在表达大多数软件建模的时候已经完全够用了，但是随着 UML 应用的推广和深入，人们发现，要使用 UML 来建模实时系统仍然缺乏一些必要的设施。比如，用例图集中于功能需求，而在实时系统中却往往强调一些非功能需求，如端到端延迟、最大响应时间、可用性、可靠性以及可维护性等，现在还没有标准的方式可以将这些实时系统中很重要的特性与用例图结合到一起。鉴于此情况，OMG 在实时任务组中成立了专门负责实时 UML 的部门，负责制定 UML 实时规范。但到现在为止，还没有提出正式采纳的 UML for RealTime 规范(但是已经有了 RFP)，因此各大公司也极力推荐自己的方案，希望能够影响最终规范的形成。

一般来说，通用 UML 在实时系统中的应用至少存在以下问题：

- 顺序图作为表示对象交互的图形，非常适合于表示事件驱动的行为，但是顺序图同样没有表示时间信息的标准方式；
- 与顺序图相比，活动图可以很好地表示并发关系，但是仍然没有表示时间信息的标准方式；
- 状态图非常适合于描述实时系统，但是除了时间事件外，状态图与时间没有直接关系；
- 在实时系统中非常重要的调度方法在 UML 中没有相应的机制可以表示；

- 实现系统时的部署图没有复杂到足够捕捉实时系统的各方面，也没有标准的方式可以描述硬件和软件的特殊性（调度规则等）。

因此，各公司都提出了自己的解决方案，一般来说都是基于 UML 规范作必要的扩展，以支持实时表达和设计。实践表明，将具体的开发工具（如 Rose RealTime）和传统的分析设计方法（如 DARTS）结合起来是比较有效的，因为工具并没有指明具体如何划分任务，而 DARTS正好在这方面是长期工程经验的积累，具有很强的实用性。反过来，DARTS 并没有工具可以帮助建模和验证，因此总是处在较低的层面，始终不能成为实时软件开发在建模和验证层面的有力辅助。将两者结合，可以说是 DARTS 的一个新发展，也可以说是 UML 渗入嵌入式实时开发的一个成功案例。

1. Rational Rose RealTime 简介

Rose RealTime 是 Rational 公司开发的实时系统开发工具。它不但支持一般的实时系统，也支持嵌入式实时系统。

Rose RealTime 的实时 UML 与 UML 标准的关系如图 9－25 所示。

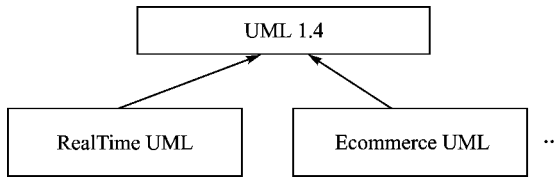


图 9－25　实时 UML 与 UML 规范的关系

Rose RealTime 并没有引入新的软件开发过程，但是却可以结合 Rational Unify Process（RUP）使用。Rose RealTime 本质上是一个强大的设计表达和模型验证/调试工具。与前述的 DARTS 方法不同，它只提供设计思路和框架，在这种框架下开发者可以容易地表达自己的设计思想并可以进行模型调试和验证。

与 DARTS 方法更为不同的是，在实时软件开发的整个过程中（从需求分析到设计、实现、测试和最终的发布），都可以使用 Rose RealTime 来进行表达和控制，开发者处于较高的抽象层面，着重关心的是状态图和协作图中的通信关系。

Rose RealTime 的主要特征可以概括如下：

- 使用 UML 的标准元素和图形来创建 UML 模型（所有 UML 工具应支持的最基本功能）；
- 将创建的 UML 模型转换为完整的代码实现（在 RealTime 中，因为集成了模型级的调试，应该说比传统 Rose 中的代码产生功能更为有用）；
- 使用可视化观察工具在模型级上执行、测试和调试模型（这是针对实时系统非常有用的功能，也是 RealTime 版本的精华之一）；

- 为开发队伍提供变更管理系统的功能(Rational 系列产品一直都很强调开发过程的管理)。

RealTime 版本既然对 UML 概念作了扩展,一定有一些超出 UML 规范的概念。下面介绍其中最关键的部分,同时用 RealTime 实际环境中的图形来展示相关概念。

① 封装体(capsule) 是一种特殊的类(class),除了普通类的特征(属性、接口等)外,还增加了一些语法机制,用于建模封装体间的通信关系以及使用状态图来建模事件触发行为。

- 封装体提供对轻量级和重量级并发对象的支持;
- 得益于封装体基于消息的本质以及高度的封装性,封装体可以容易地分布到不同的物理控制线程上,并且不用对封装体作任何改变(这使得模型和代码以及最后的部署得到统一);
- 封装体之间的通信通过它们各自的端口(port)对象进行。

为什么要引入封装体的概念呢?除了以上的原因外,最根本的原因还是受到软件开发的需求所驱动。由于封装体不需要知道除了自己的接口之外的任何环境信息,这是高内聚性和低耦合性的一种体现,而正是这种独立性使得封装体比一般的对象有更好的分布性、可重用性和鲁棒性。这些性质正是软件开发人员所极力追求的。在实时系统中引入这个概念,对提高软件生产效率同样有好处。

至此,细心的读者可能已经发现,封装体既然是最后部署到单个线程或者进程(或任务)中的,那么封装体就等价于拥有可执行线程(进程、任务)的主动对象了吗?事实的确如此,如果再联想到 DARTS 方法中的任务划分,可以看出,分析出封装体实际上就是分析出任务。这里完全可以将 DARTS 方法和 Rose RealTime 的表示法结合起来,两者是互补的,分别处于不同的层次。DARTS 方法给出了实时软件开发各阶段的具体方法和原则,而 Rose RealTime 用模型将设计和编码、测试、发布、管理等一系列的动作统一起来,但是并没有给出每一步如何做(这有赖于软件开发的规范和工程经验的积累,在这方面 RUP 是比较好的指导,但是对于实时软件开发的核心问题——如何划分任务,RUP 还是有一定距离)。

② 协议(protocols) 封装体之间交换的消息形成一个消息集合,把这个消息集合叫做协议。消息是从接收者和发送者两个角度来定义的,因此,有两个不同的视角来看待协议,称这些视角为协议角色(protocol roles)。

在封装体之间发送的消息包含了信号名(标识消息)、可选的优先级(表示本消息对比同一线程内其他未处理消息的相对重要性)以及可选的应用数据。

③ 端口(port) 端口是封装体用来发送和接收消息的对象。端口是属于封装体实例(capsule instance)的,它随着封装体实例的创建/销毁而被创建/销毁。

为了指定端口可以发送和接收哪些消息,端口必须成为某个协议角色。协议角色实际上就是从某个端口可以发送和接收哪些消息的一个规定。

④ 状态图 状态图使用 UML 规范中定义的符号进行表示。它表达了封装体的内

部行为。

⑤ 封装体结构图　封装体结构图是 Rose RealTime 引入的新概念。它基于协作图，用于指明封装体的界面（端口）和内部组成。

除以上概念外，Rose RealTime 引入了可执行模型（executable model）的概念。它在高层次的设计层面上执行模型，使得不但在软件开发的早期阶段，而且在后续整个开发过程中都可以更好地发现、分析和修改设计中存在的问题。这无疑将在很大程度上提高软件开发的可控性和软件产品质量。另外，在实时系统中，随着系统规模增加，任务之间的交互变得越来越复杂和不清晰，这时通过纯粹的文档跟踪和代码级跟踪都不容易抓住问题的关键，通过 Rose RealTime 的可执行模型却可以从设计的角度确定问题的所在，并且在开发和修改软件设计的过程中不断地执行软件模型，始终保持模型可执行。这样就可以既不失去“森林”视图（整体设计），也不会丢失对“树木”视图（可执行代码）的跟踪。

下面对使用 Rose RealTime 的开发过程作一概述，然后通过一个具体的开发实例来说明使用 Rose RealTime 建模、执行和验证实时软件的设计。

2. Rose RealTime 的开发过程

Rose RealTime 支持迭代的面向对象开发过程，该迭代过程源自 RUP。它创建可执行模型的过程可以概括如下：

① 使用用例建模元素并且使用用例图来对问题进行半形式化的描述。

② 创建封装体、协议、类，使用类图、封装体图和封装体状态图来描述模型的结构和行为。在封装体状态图和类操作中加入实现代码。

③ 使用协作图和序列图捕捉各种用例图的内部行为，并在运行时用 Rose RealTime 的执行和调试工具验证模型行为。

④ 一旦设计文档定下来，就使用类和协议的状态图来捕捉抽象设计，以便别人也能理解你的设计。

⑤ 使用组件图指定模型的编译时配置（诸如平台、OS 和编译器版本等）。

⑥ 使用部署图来表示怎样部署和执行组件。部署图也可以用来表示目标系统的物理结构。

开发实例：机器人控制器

① 使用用例图进行需求分析，如图 9－26 所示。

Rose RealTime 用例实景图如图 9－27 所示。

② 使用 DARTS 方法进行任务划分（Rose RealTime 并没有如何划定线程、进程或者任务的具体方法，这时使用 DARTS 方法是比较好的选择；当然也可以用其他分析方法），将每个任务作为一个封装体，并且分析出类。不管用什么方法，都是从需求中分析出类和封装体的。封装体和类的最大区别在于封装体是拥有单独线程（进程、任务）的，是针对并发操作设置的概念。在设置封装体的时候，需要设置一个 RobotController 封装体，用来充当主任务，包装所有

的相关任务。利用 DARTS 方法分析出的封装体如图 9－28 和图 9－29 所示。

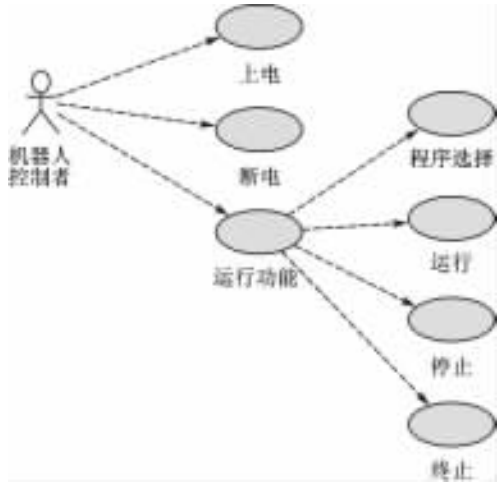


图 9－26　用例图

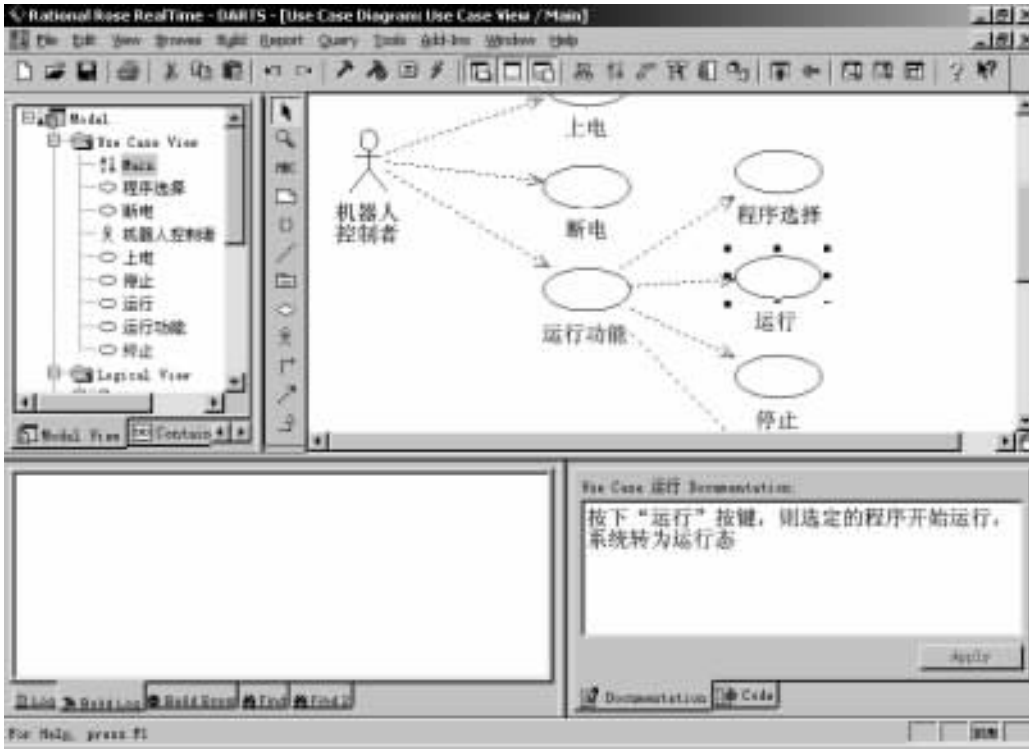


图 9－27　Rose RealTime 用例实景图

图 9－28　封装体图

图 9－29　封装体图开发实景

③ 用封装体结构图和序列图来描述封装体的结构和行为。这样做的目的有两个：一是帮助开发者确认封装体的责任，以及封装体之间如何交互以实现用例中定义的行为；二是帮助开发者识别出用于封装体间通信需要用到的端口和协议。具体操作方法是：打开 RobotController 的结构图，将相关的所有封装体聚集进来，如图 9－30 所示。

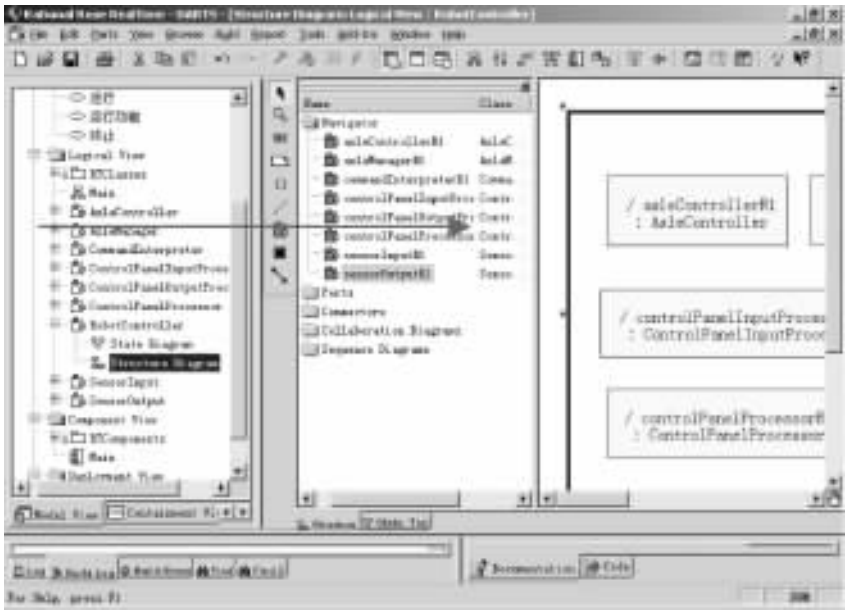


图 9－30　将封装体聚集到主封装体内

在逻辑视图中的 Query 菜单中选择 Filter Relationships，选择所有关系，单击 OK 按钮，就可以看到逻辑视图的 Main 视图已经显示出了这个聚集关系，如图 9－31 所示。

下面创建序列图、协议、端口和连接器(connector)。

在机器人控制主任务中将任务间的通信用序列图表示出来(对于复杂的交互，如果一个图表示不清晰，可以用多个图分别说明，每个形成一个交互流(flow))。图 9－32 表示了其中的一种交互情况。

④ 定义协议。首先在逻辑图的 Main 图中创建一个协议，然后给该协议添加信号集。

控制面板处理器和命令解释器之间的协议如图 9－33 所示(协议就是两个对象之间通信的接口集)。

需要说明的是，协议是可以随着模型的细化而逐步细化的。

⑤ 创建端口和连接器。打开控制面板处理器的结构图，将 CPP_PROTOCOL 拖到其边界上。对命令解释器同样处理，然后打开 RobotController 的结构图，就可以看到两者都多了一个端口。用连接器将两个端口连接起来，如图 9－34 所示。

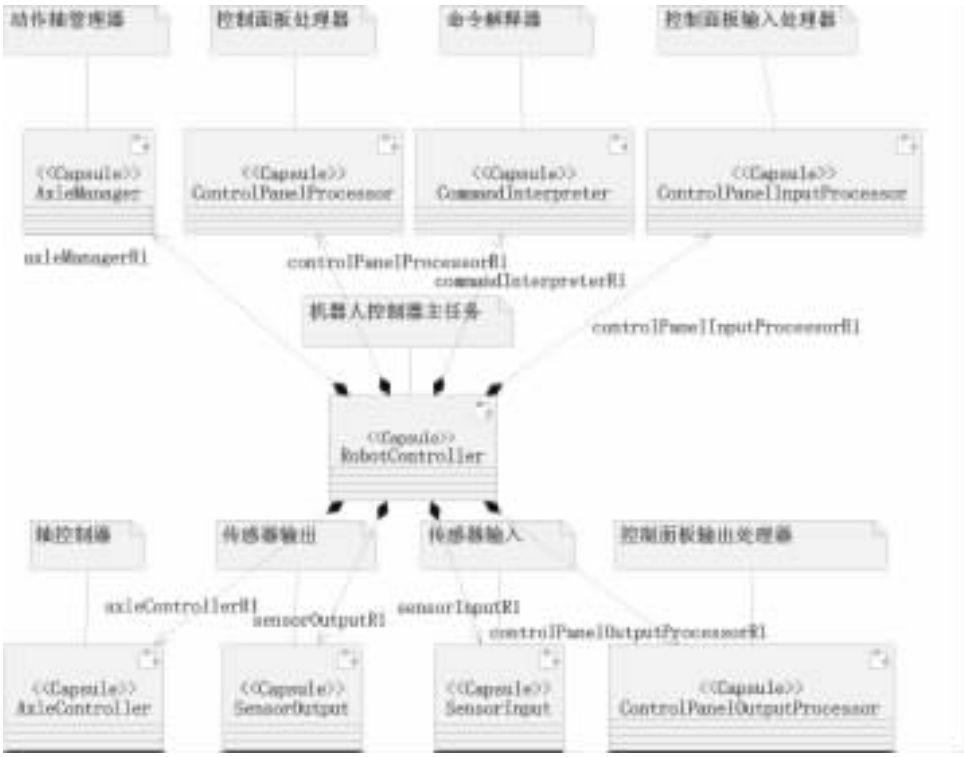


图 9－31　逻辑视图的 Main 视图

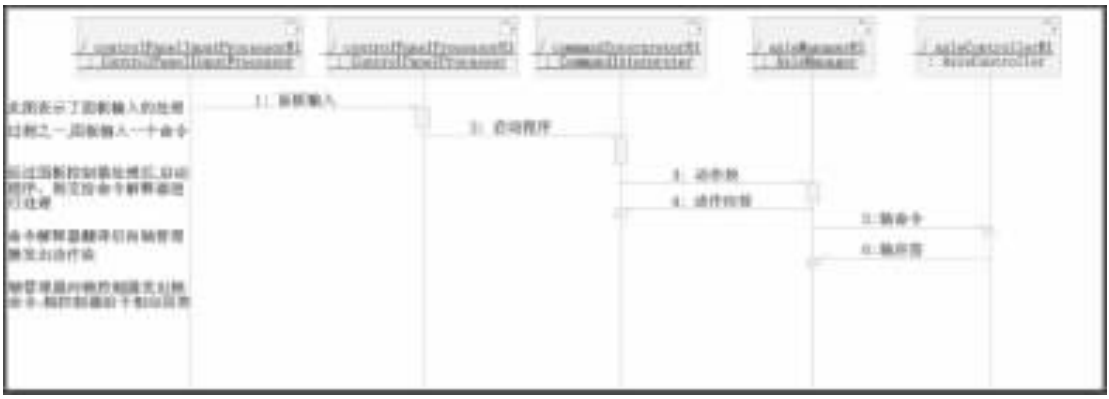


图 9－32　交互序列图

依以上步骤建立所有协议和端口、连接器，就可以形成一个完整的软件模型（原型）。

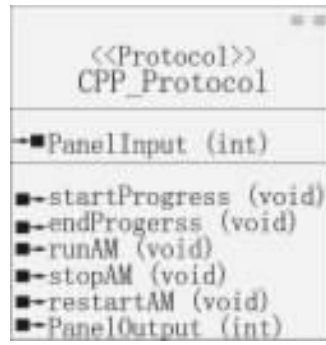


图 9－33　协议

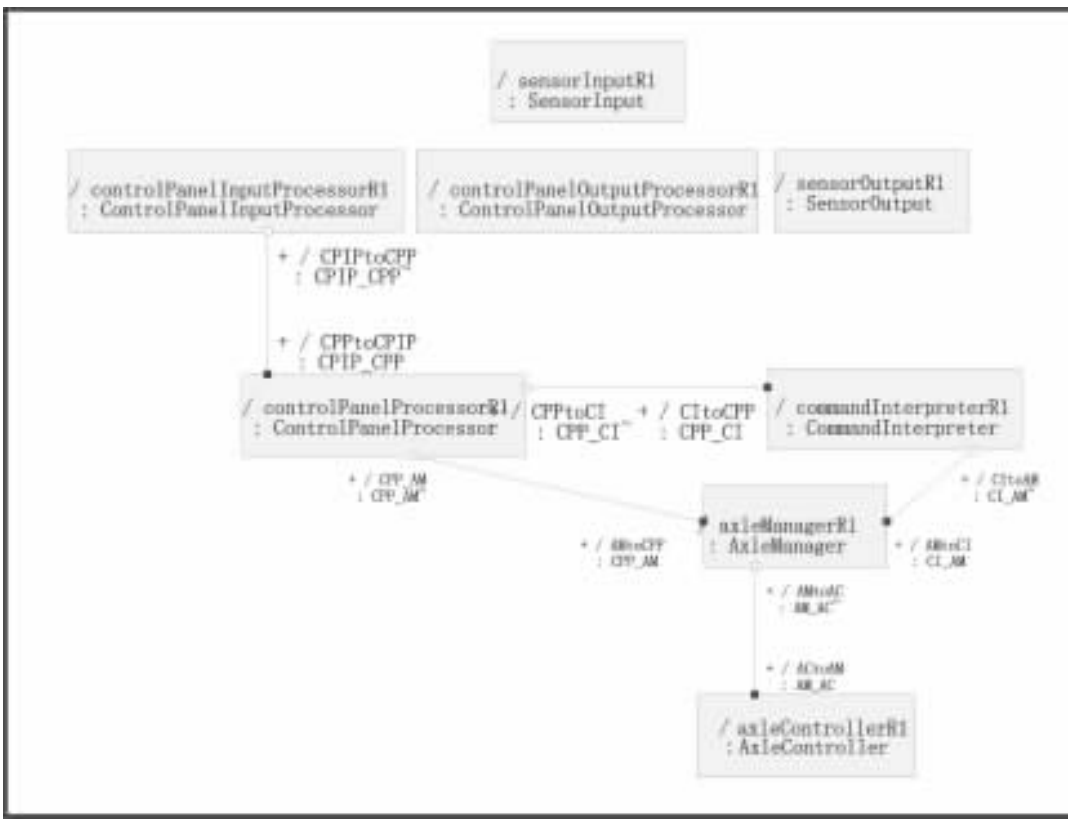


图 9－34　模型的一部分(建立了部分协议和端口)

为了简化例子，取 CPIP_CPP **协议、**CPP_CI **协议、**CI_AM **协议、**CPP_AM **协议和** AM_AC **协议一共五个协议来说明如何进行下一步的模型执行。建立好的模型逻辑如图** 9－35 **所示。**

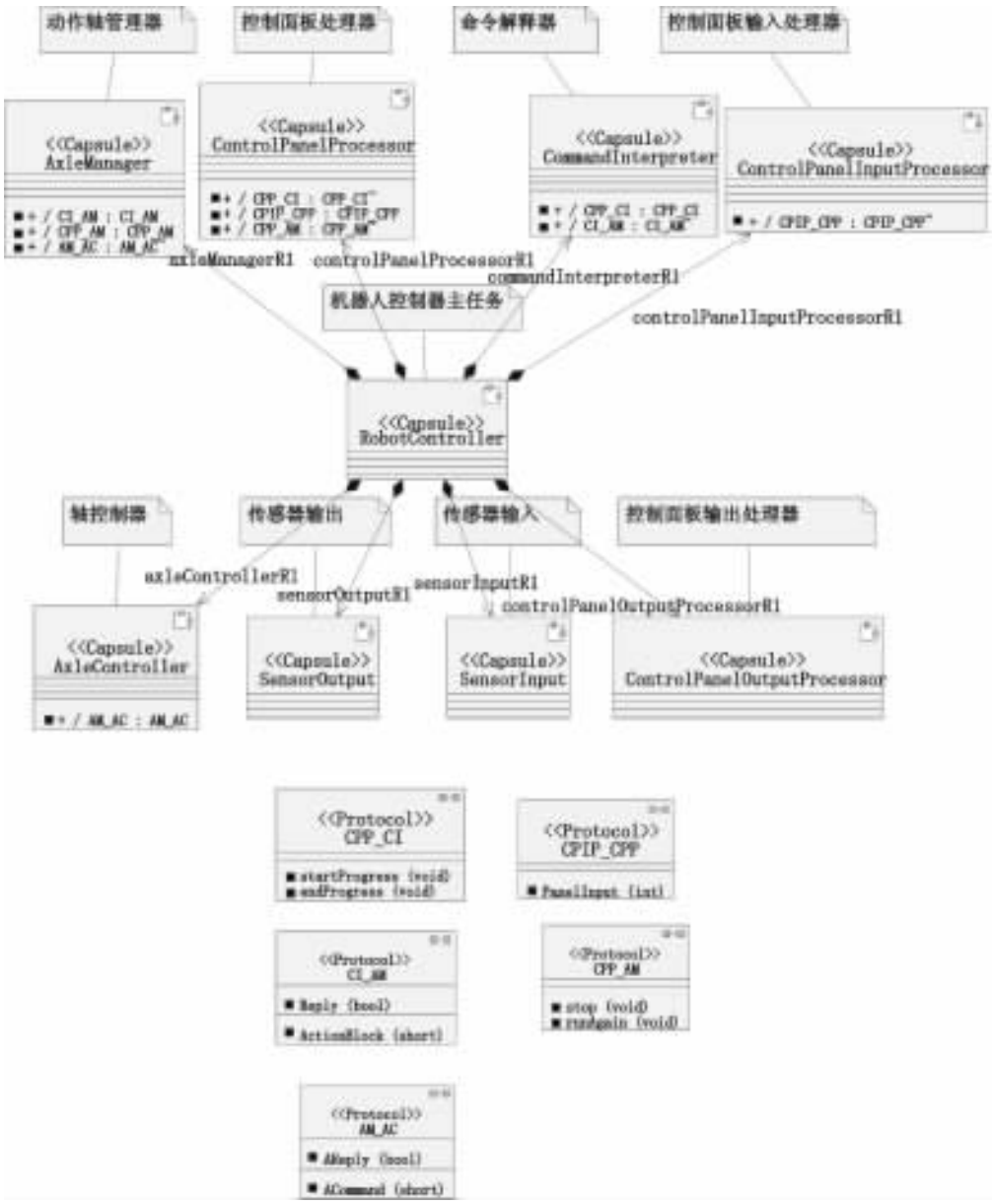


图 9－35　建立好的模型逻辑

通过分别指明协议和端口，封装体之间的交互关系就比较明确了。

以上原型可以让开发者评估主要风险，给客户尽早的反馈以及进行测试和持续集成。以上模型是可执行的，虽然模型并未完整完成，但是已经有了足够的信息来初步验证模型和设计方法，从而在设计阶段修正许多错误。充分利用 Rose RealTime 的最重要一步就是尽量利用好它迭代式的开发方法。下面就创建相应的组件并部署、执行。

⑥ 创建组件。在组件视图中新建一个组件，并将其命名为 Robot。打开组件配置规则，选择目标平台和编译工具链，选择了“NT 40T. X86 - VisualC++-6.0”的组合。在 Reference 选项卡中从左边拉入所有相关封装体、类、协议等，表示本组件要用到的各种元素。C++ Executable选项卡中的 TopCapsule 项选择 RobotController，单击 OK 按钮，完成组件的建立。建立好的组件如图 9-36 所示。

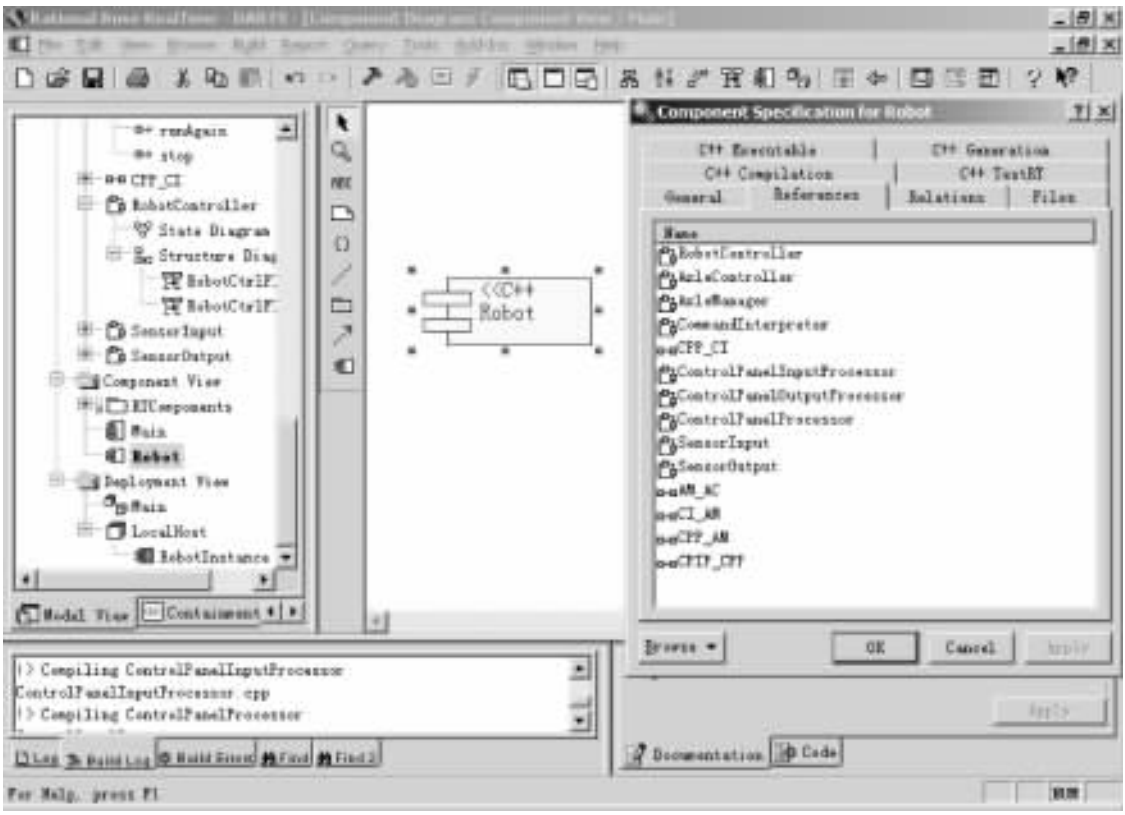


图 9-36　建立好的组件

⑦ 部署组件。在部署视图中，建立一个 Processor 节点，取名为 LocalHost，然后将上一步建立的组件拖到该节点上，立即就会出现一个 RobotInstance 节点(表示已经部署好)。将该节点设置为活动(set active)，然后编译该模型，完成后如图 9-37 所示。

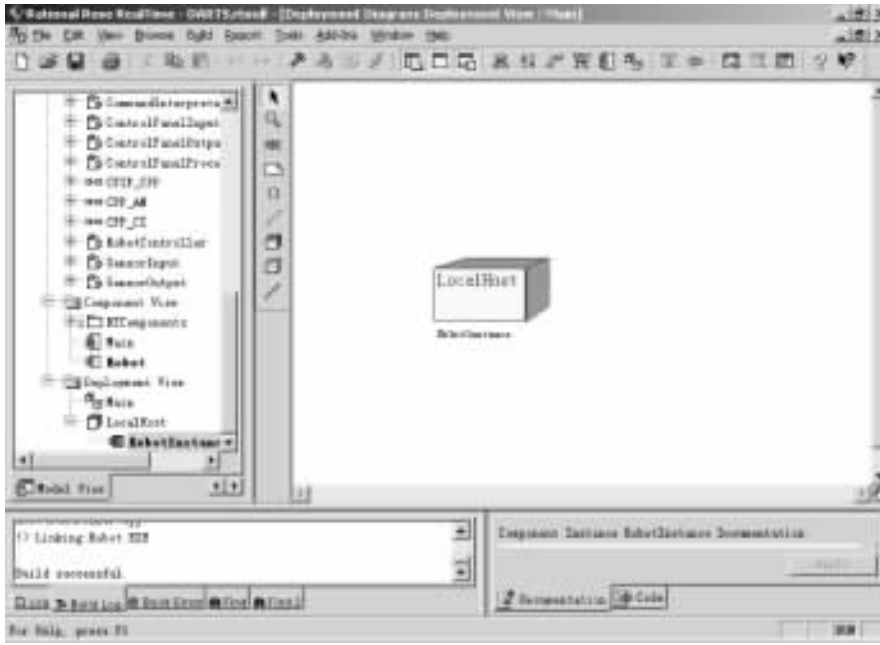


图 9－37　部署组件

⑧ **执行模型。直接运行** RobotInstance，**可看到如图** 9－38 **的界面。**

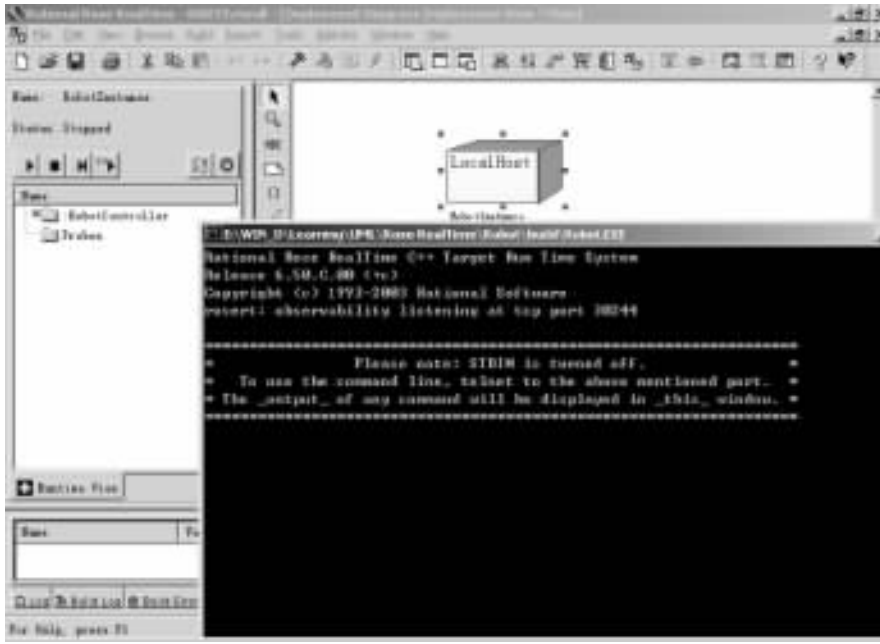


图 9－38　执行模型

单击开始图标即可执行模型，因为没有加入任何具体代码，也没有指定各封装体的内部状态机，所以看不到什么输出。但是根据模型所产生的所有代码已经放到了 Robot\src 目录下，如图 9－39 所示。

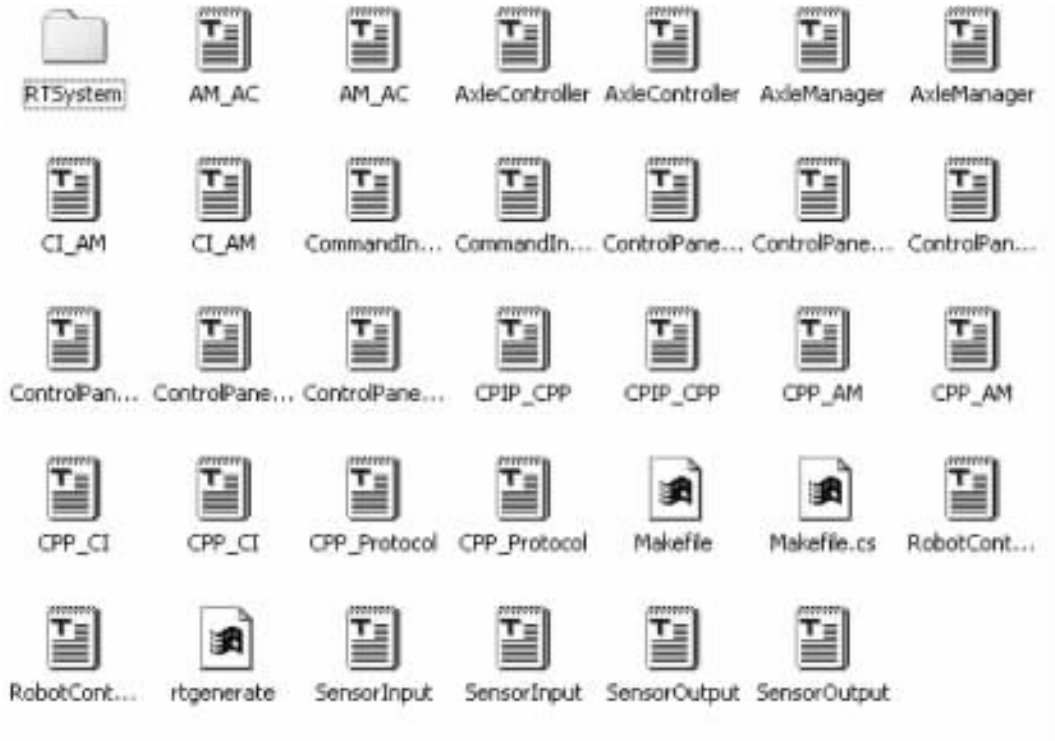


图 9－39　产生的代码

模型执行时如图 9－40 所示。

在继续进行之前，对前面做的工作做一总结如下：

- 首先建立了机器人控制系统的用例图；
- 然后使用序列图和结构图描述了封装体之间的通信；
- 创建了协议，协议描述了封装体之间通信的一套信号；
- 编译并执行了模型。

以上步骤的关键是确定以下各项：

- 系统中关键组件的名字（在模型视图中可以看到）；
- 每个组件的主要职责（在文档框中描述）；
- 组件之间通信的接口（在结构图和序列图中描述）。

有了以上基础，就可以增量式地向模型添加内容，并且在添加的过程中可以不断执行和验

证。首先给封装体添加行为。

⑨ 给封装体添加行为。前面所创建的封装体是没有行为的，因此也不能完成什么具体的功能。添加行为实际上就是创建状态图，包括图中的状态、变迁、触发器和动作代码。到底一个封装体具有哪些行为呢？这可以通过封装体的责任描述和由该封装体加入的序列图来提取，因为这两个来源表明了封装体接收和发送什么消息。

下面来分析面板输入控制处理。它开始处于等待输入态，当收到输入时（按下按钮时），转换到接收命令状态，在接收命令状态内将队列中的命令数量加 1；收到命令处理面板的确认后，面板输入控制进行队列满判定，如果满，则暂停接受控制命令，否则直接返回等待用户命令的状态，如图 9－41所示。

在变迁上定义相应的触发器（触发器是引起状态变化的事件），然后填入变迁代码，则变迁的箭头就会变成实心。

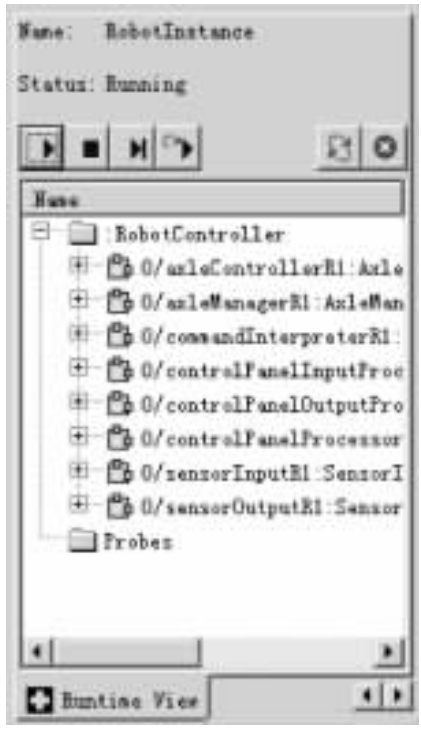


图 9－40　模型执行时

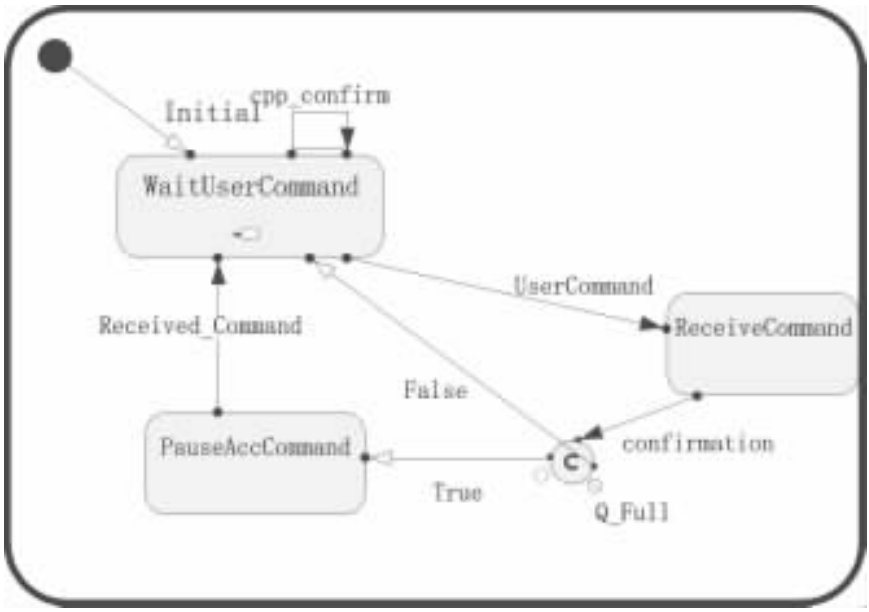


图 9－41　面板输入控制处理

实际上，这里应该定义专门的输入队列对象来保存输入命令。为了使例子尽量简化，假设最简单的情况，即用 Rose RealTime 的时间触发器来模拟用户输入，每 2 s 产生一个控制请求（控制请求用数字来表示）。一旦 CPIP 收到控制请求，就发出 CommandInput 通知 CPP，CPP 取出请求进行处理，然后用 confirmation 通知 CPIP（因为没有专门的队列，假设 CPP 对 CPIP 有一个确认过程）。

在每个变迁和状态中要求填入动作代码，这些代码最初可以是简单的原型级的代码，只要可以编译通过，逻辑正确即可，这时集中精力于验证整个模型的体系结构上；而随着每次迭代，这些代码可以逐步细化到真实的、产品级的代码，同时也可以不断在模型中进行编译和执行。

设置好所有有关 CPIP 和 CPP 之间交互的触发器和动作代码后，来看一下编译执行的结果。首先是可以通过状态监视器监视封装体的内部状态变化，如图 9－42 所示。

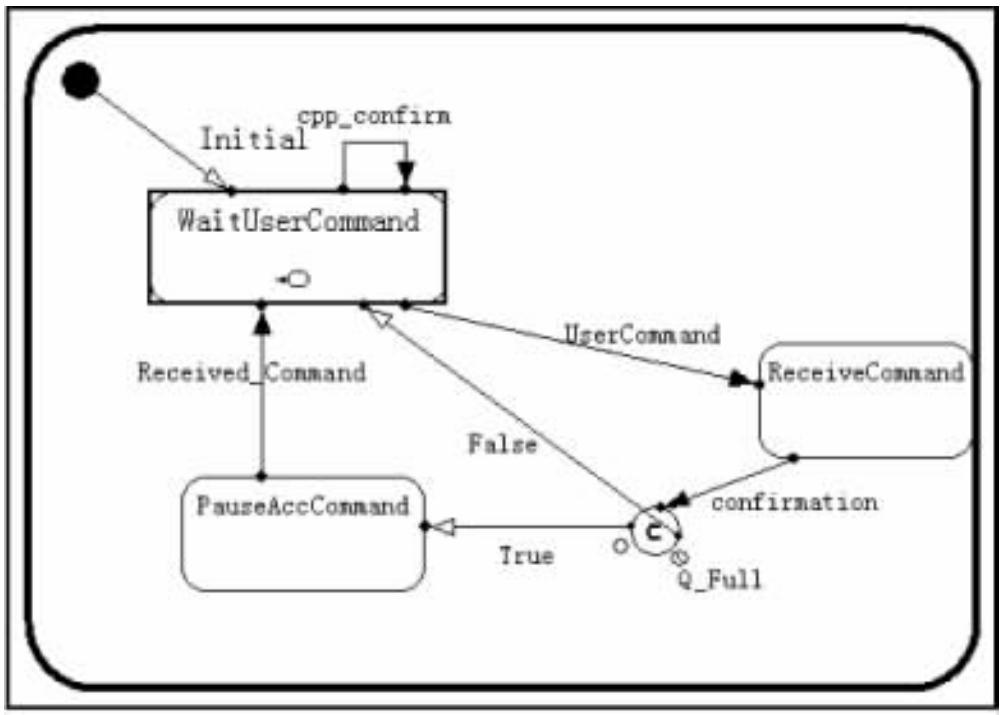


图 9－42　通过状态监视器监视封装体的内部状态变化

还可以用跟踪工具跟踪封装体之间的信息，以及利用其他监控工具来监视运行时的变量值等。下面是一个样例，如图 9－43 所示。

例子因为经过了大量简化，离完全实现整个模型还有一定距离，主要是让读者对利用 Rose RealTime 如何进行建模和运行、调试有一个整体的概念。

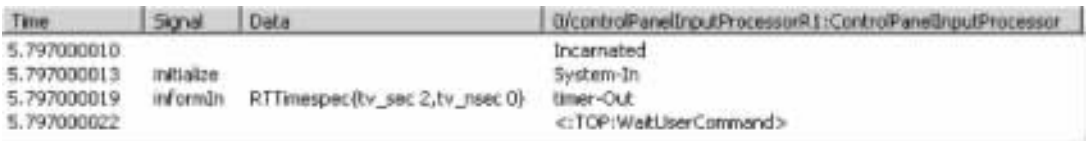


图 9－43　样例图

3. 小　结

DARTS 方法的不足：如果遇到大规模的实时软件，DARTS 对任务的划分方法虽然有效，但是使用起来却不那么方便，只有采取分层的办法，层层利用 DARTS 方法细化。

可以看到，使用 Rose RealTime 似乎使得原来看起来直接的控制流、状态变迁等表达更为复杂了，实际上正是通过建立模型，才迫使软件设计者可以有步骤、有规律、有把握地一步步推进整个软件建造过程，而且可以采用逐步细化的办法（配合可不断验证的设计模型）来推进软件设计和实现。

DARTS 方法可以很自然地结合到 Rose RealTime 中，使用 Rose RealTime 作为建模和执行、验证工具和集成开发环境；而如何划分任务、如何定义任务间接口可以利用 DARTS 的分析方法进行。可以利用 Rose RealTime 来表达设计、验证设计，协助迭代开发；用 DARTS 来做重要的分析工作，两者可以结合得很好。

当然，Rose RealTime 也可以结合其他任务划分方法使用，这取决于开发者的习惯。所以 Rose RealTime 并不是要淘汰以前的办法，而是通过“模型–代码同步”的方法让实时软件开发者在控制软件开发过程方面更加有力。

9.3　影响系统性能主要因素的分析

前面较详细地论述了 DARTS 的设计方法，并集中在多任务的划分上。无论采用哪一种设计方法，在进行嵌入式实时系统设计时都要将保证系统响应时间要求作为考虑的重要因素。将设计过程（包括多任务的划分、数据流程、控制流程、调度方法和资源分配等）、与任务时限要求相关的时间参数以及可靠性初期评估进行形式化描述。其次，根据实时任务的特征，将影响实时性能的因素分为任务属性和系统属性。任务属性包括任务周期（对非周期任务，有最大到达率和平均到达率）、任务执行时间和任务数量。系统属性包括系统响应时间、任务调度算法、资源共享情况、总线结构、通信协议和输入/输出的时间约束条件等。利用上述的软件特征参数，对实时多任务软件的实时性进行评估。

如果将任务时限要求（用 DL 表示）表达为系统中与时限相关的参数的函数，即 $DL(X)=F(x_1,x_2,x_3,\cdots,x_n)$，则围绕满足系统的时限要求，将整个设计过程用图 9－44 来表示。

下面对影响系统响应时间的几个重要因素进行较详细的描述。

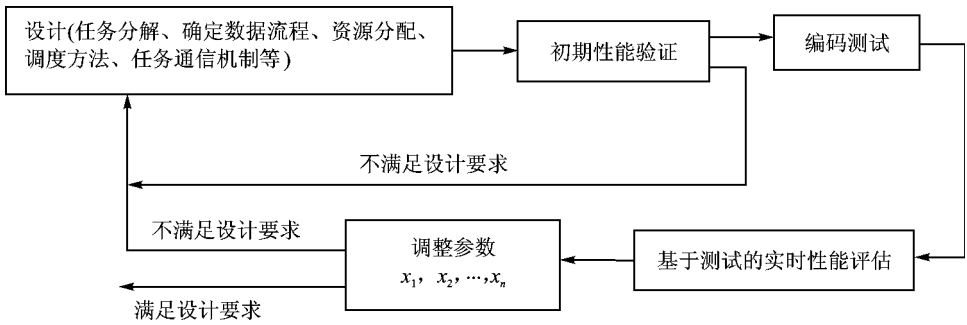


图 9-44　围绕时限要求的设计过程

在第 5 章,对操作系统的任务调度算法作了详细的说明。这里,在任务级对使用不同调度算法时任务的时限可满足性作进一步分析。

假设一个由 m 个任务组成的任务集合为$\{\tau_1,\tau_2,\tau_3,\cdots,\tau_m\}$;

各任务周期分别为$\{T_1,T_2,T_3,\cdots,T_m\}$;

各任务计算时间分别为$\{C_1,C_2,C_3,\cdots,C_m\}$。

按照上述假设,从时刻 0 到时刻 t,任务 i 总计到达的次数为$[t/T_i]$,需要的运行时间为$[t/T_i]C_i$,则对于 m 个任务,从时刻 0 到时刻 t,处理器用于任务执行的时间为

$$[t/T_1]C_1+[t/T_2]C_2+\cdots+[t/T_m]C_m$$

定义任务集的利用率(utilization)为

$$U=\{[t/T_1]C_1+[t/T_2]C_2+\cdots+[t/T_m]C_m\}/t=$$

$$C_1/T_1+C_2/T_2+\cdots+C_m/T_m=\sum_{i=1}^{m}C_i/T_i$$

根据 Liu 和 J. Layland 提出的调度理论,如果 $U>1$,则采用任何调度算法,都至少有一个任务的截止时间得不到满足。当采用某种调度算法调度一个任务集时,如果该任务集的所有任务都能够满足截止时间要求,就称该任务集在该调度算法下是可调度的。同时,Liu 和 J. Layland还给出了该利用率的一个最小上界,即

$$U^*=n(2^{1/n}-1)$$

其中,n 为任务数量,并证明了以下定理:

对于给定的 n 个任务,如果$U=\sum_{i=1}^{m}C_i/T_i\leqslant U^*=n(2^{1/n}-1)$,则这 n 个任务是可调度的。但该定理仅仅是判断任务可调度性的充分条件,而不是充分必要条件。

例 9-1　假设有两个任务$\{\tau_1,\tau_2\}$,其中 $C_1=1,C_2=2,T_1=2,T_2=5$。安排 τ_1 的优先级高于 τ_2 的优先级,假设两个任务都从时刻 $t=0$ 开始,则任务的执行过程如图 9-45 所示。

很容易验证,两个任务的截止时间都能够得到满足。

图 9－45　多任务执行过程一

例 9－2　假设有两个任务$\{\tau_1,\tau_2\}$,其中,$C_1=1$,$C_2=2$,$T_1=2$,$T_2=5$。安排τ_1的优先级低于τ_2的优先级,假设两个任务都从时刻$t=0$开始,则任务的执行过程如图 9－46 所示。

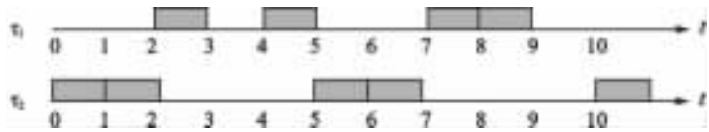


图 9－46　多任务执行过程二

可以看出,任务 1 的第一次执行就不能满足截止时间要求。因此,对同样的任务集的不同优先级安排,可调度性是不同的。

例 9－3　假设有两个任务$\{\tau_1,\tau_2\}$,其中$C_1=1$,$C_2=2.1$,$T_1=2$,$T_2=5$。

可以计算出　　$U=C_1/T_1+C_2/T_2=0.92$

安排τ_1的优先级高于τ_2的优先级,则任务的执行过程如图 9－47 所示。

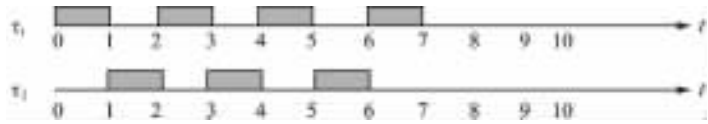


图 9－47　多任务执行过程三

可以分析出,任务 2 的第一次执行,在它的截止时间($T_2=5$)之前不能完成。

如果安排τ_1的优先级低于τ_2的优先级,则任务的执行过程如图 9－48 所示。

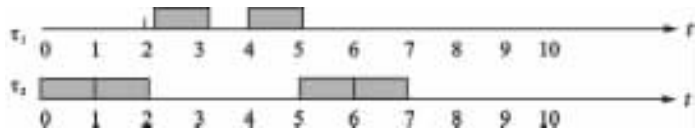


图 9－48　多任务执行过程四

可以看出,任务 1 的第一次执行就不能满足截止时间要求。因此,即使$U<1$,采用静态优先级调度算法,也不能保证任务集是可调度的。

从上面的例子可以得出以下结论:

- 任务集的可调度性取决于任务优先级的安排;

● 即使任务集的利用率 $U<1$，采用静态优先级调度，任务集仍然可能是不可调度的。

在静态优先级调度时，需要解决的主要问题是：

● 怎样安排任务优先级，哪一种安排是最好的？

● 怎样评价任务优先级安排方法？

● 怎样比较不同的任务优先级安排方法？

下面就上述问题作进一步讨论。

9.3.1 静态优先级调度性能分析

这里将静态优先级调度算法分为简单优先级安排和优化优先级安排两种情况。

(1) 简单优先级安排

简单优先级安排分为以下几种情况。

① 根据计算时间 C_i 来安排：

● 最短计算时间优先；

● 最长计算时间优先。

② 根据任务周期 T_i 来安排：

● 最短周期优先；

● 最长周期优先。

③ 根据任务利用率 C_i/T_i 来安排：

● 最低任务利用率任务优先；

● 最高任务利用率任务优先。

④ 随机安排。

(2) 优化优先级安排

某种调度算法 DDSF 被称为是优化方法，是指这样一种情况：如果一个任务集能够被一种静态优先级方法所调度，该任务集也能够被 DDSF 所调度，则称 DDSF 是一种优化调度算法。

Liu 和 J. Layland 证明了比率单调调度算法 RMS(Rate - Monotonic Scheduling method)是一种优化算法。RMS 算法的基本原理是：给周期最短的任务安排最高优先级，即短周期任务优先。下面就这一调度算法作进一步讨论。

1. RMS 调度算法

先来分析只有两个任务的情况。首先定义任务调度中的几个关键时间。

定义 1 任务的关键时间点——在该时间点上，任务的调度请求具有最长响应时间。

比如在例 9-1 中，任务 τ_2 的关键时间点是 0，10，20，…。这里要特别注意的是，时刻 5 不是关键时间点，因为在该时刻，任务的请求被立即响应。

定义 2 任务的关键时间域——关键时间点与响应结束之间的时间间隔。

比如在例 9－1 中，任务 τ_2 的关键时间域是[0，4]，[10，14]，[20，24]，…。要研究任务集在最坏情况下的可调度性，需要考虑任务集中所有任务的关键时间域。

定理 1　任何任务的关键时间点，出现在当该任务的请求与比之优先级更高的所有任务的请求同时产生时(证明略)。

按照定理 1，需要考虑在时刻 0 同时启动任务集中的所有任务。

假定 $T_1 < T_2$，仅需要考虑两种方法。

方法 1　安排 $\tau_1 = (C_1, T_1)$ 有更高的优先级，任务的运行情况如图 9－49 所示。

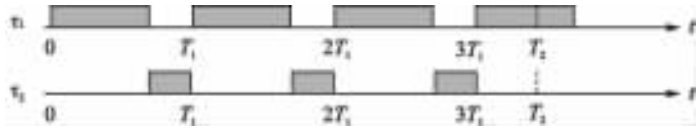


图 9－49　任务 1 优先级高的运行情况

用方法 1，任务集能够被调度的充分必要条件是

$$[T_2/T_1]C_1 + C_2 \leqslant [T_2/T_1]T_1 \tag{9-1}$$

方法 2　安排 $\tau_2 = (C_2, T_2)$ 有更高的优先级，任务的运行情况如图 9－50 所示。

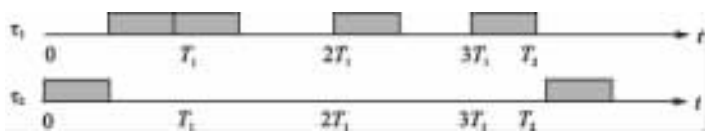


图 9－50　任务 2 优先级高的运行情况

任务集能够被调度的充分必要条件是

$$C_1 + C_2 < T_1 \tag{9-2}$$

从前面的例子可知，式(9－1)成立并不意味着式(9－2)成立。当用方法 1 时任务集可调度，但用方法 2 时该任务集可能是不可调度的。由此可以说明方法 2 不是优化的。

再考察式(9－2)成立时，式(9－1)是否成立。

将$[T_2/T_1]$乘以式(9－2)，则有

$$[T_2/\ T_1]C_1 + [T_2/\ T_1]C_2 \leqslant [T_2/\ T_1]T_1$$

因为

$$[T_2/\ T_1]C_1 + C_2 \leqslant [T_2/\ T_1]C_1 + [T_2/\ T_1]C_2$$

所以有

$$[T_2/\ T_1]C_1 + C_2 \leqslant [T_2/\ T_1]T_1$$

即式(9－2)成立时，式(9－1)成立。

由此可得出，如果用方法 2 能够调度一个任务集，则方法 1 也能调度该任务集，因此方法 1 是优化的。

前面讨论的任务集仅有两个任务，下面分析多个任务的情况。

定理 2 如果一种优先级安排使得某些任务集是可调度的，则这些任务集在 RMS 调度算法上一定是可调度的。

定理 2 说明了 RMS 算法在静态优先级安排中是优化算法。

例 9-4 证明对于任意的 δ，当 $\delta > 0$ 时，任务集 $\{(C_1, T_1), (C_2, T_2), (C_3, T_3)\} = \{(1, 2), (\delta, 2), (1, 3)\}$ 采用任何静态优先级安排都是不可调度的。

证明：根据定理 2 可知，只需要证明如果用 RMS 调度算法时任务集是不可调度的，即可说明任何静态优先级安排下任务集都是不可调度的。

按照定理 1，只考虑时间范围 $[0, 3]$，假设所有任务都在时刻 $t = 0$ 启动，使用 RMS 调度算法，τ_3 在该时间范围内可获得的执行时间为

$$T_3 - (2C_1 + C_2) = 3 - (2 + \delta) = 1 - \delta < C_3$$

因此 τ_3 的第一次执行就会超过它的截止时间，所以任务集是不可调度的。图 9-51 是例 9-4 中任务集的执行过程。

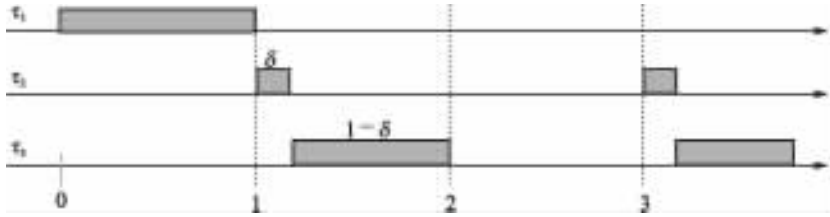


图 9-51 例 9-4 的任务集运行过程

例 9-5 证明当 $C > 0$ 时，采用静态优先级安排方法，任务集 $\{(C_1, T_1), (C_2, T_2), (C_3, T_3)\} = \{(C, 3C), (2C, 6C), (4C, 12C)\}$ 是可调度的。这里给出证明的思路，按照定理 1，只考虑时间范围 $[0, 12C]$。假设所有任务都在时刻 $t = 0$ 时启动。用 RMS 调度算法，可以证明在时间范围 $[0, 12C]$ 内，任何任务的每次执行都能在它的截止时间内完成。详细证明过程读者可自行完善。图 9-52 是例 9-5 中任务集的运行过程。

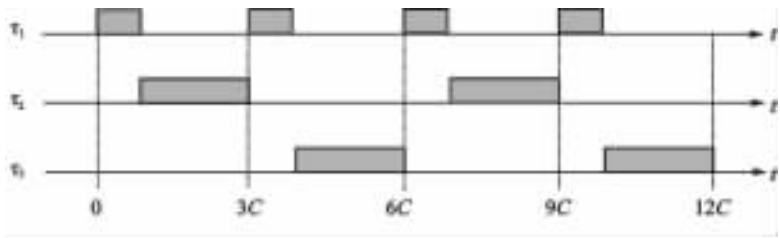


图 9-52 例 9-5 的任务集运行过程

下面，就采用静态优先级安排时的最坏情况进行分析。

2. 最坏情况利用率分析

(1) 全利用率任务集与 WCAU

如果一个任务集中的任何任务增加任何计算时间，都会导致至少有一个任务的执行不能满足截止时间要求，则称这个任务集是全利用率任务集。

例 9-6　任务集{(1，2) (1，2)}的 $U=100\ \%$，显然它是一个全利用率任务集。

任务集{(1，2)，(1，3)}的 $U=1/2+1/3=0.833$，虽然它的 $U<100\ \%$，但它仍然是一个全利用率任务集。

定义：最坏情况利用率 WCAU(Worst Case Achievable Utilization)等于可调度任务集的利用率的最小上界。

对于给定的调度方法，如果一个任务集的利用率低于最坏情况利用率，则该任务集一定是可调度的，与任务的周期、计算时间等因素无关。这一结论在实际应用中是非常有用的。

(2) 由 WCAU 导出的定理

定理 3　对于由 2 个任务构成的任务集，$\mathrm{WCAU_{RMS}}=2(2^{1/2}-1)\approx 83\ \%$。

定理 4　对于由 m 个任务构成的任务集，如果 $1\leqslant i,j\leqslant m$，$T_i/T_j<2$，则 $\mathrm{WCAU_{RMS}}=m(2^{1/m}-1)$。

实际上，可以有一个更一般性的定理。

定理 5　对于由 m 个任务构成的任务集，$\mathrm{WCAU_{RMS}}=m(2^{1/m}-1)$。

对于 RMS，有一个很严格的检验方法来保证任务的截止时间。

规则 1：如果 $U>100\ \%$，则任务集是不可调度的。

规则 2：如果 $U\leqslant m(2^{1/m}-1)$，则任务集是可调度的。

规则 3：还可以采用的方法是，使用 RMS 并假设所有任务在时刻 $t=0$ 同时启动，检查每个任务的可调度性。

下面推导 $\mathrm{WCAU_{RMS}}$ 的极值，首先证明它是一个 m 的递增函数，即

$$\frac{\mathrm{d}U}{\mathrm{d}m}=(2^{1/m}-1)+m\left(\frac{-1}{m^2}2^{1/m}\ln 2\right)=\left(\frac{2^{1/m}}{m}\right)(m-m2^m-\ln 2)\leqslant 0$$

由此可以导出它的极值，即

$$\lim_{m\to\infty}\mathrm{WCAU_{RMS}}=\lim_{m\to\infty}m(2^{1/m}-1)\lim_{m\to\infty}\frac{2^{1/m}-1}{1/m}=$$

$$\lim_{m\to\infty}\frac{(2^{1/m}-1)^t}{(1/m)^t}=\lim_{m\to\infty}\frac{2^{1/m}\ln 2\left(\frac{1}{m^2}\right)}{\frac{1}{m^2}}=\ln 2=0.69$$

根据上述推导过程和结论，有一个更简单的检验任务是否满足截止时间的方法。

规则 1：如果 $U>100\ \%$，则任务集是不可调度的。

规则 2a：**如果** $U \leqslant 0.69$，**则任务集是可调度的**。

规则 2b：**如果** $U \leqslant m(2^{1/m}-1)$，**则任务集是可调度的**。

规则 3：**使用** RMS **并假设所有任务在时刻** $t=0$ **同时启动，检查每个任务的可调度性**。

下面针对周期任务，介绍如何检查每个任务的可调度性。

考虑任务集 $S=\{A_1, A_2, \cdots, A_n\}$，**用** $\lfloor t/T_j \rfloor$ **来表示任务** A_j **从时刻** 0 **到某一时刻** t **所到达的次数，则** $C_j \times \lfloor t/T_j \rfloor$ **就为多次完成任务** A_j **所需要的累计** CPU **时间**。

简言之，对于任务 $A_1, A_2, \cdots, A_i$，**从** 0 **到时刻** t **所需要的累计** CPU **时间为**

$$W_i(t) = \sum_{j=1}^{i} C_j \lfloor t/T_j \rfloor$$

按照定义，这时处理器的利用率为

$$L_i(t) = W_i(t)/t$$

推论 1 假设任务 $A_1, A_2, \cdots, A_i$ **同时到达，由于** A_i **的优先级最低，因此** A_i **的完成时间一定在** $A_1, \cdots, A_{i-1}$ **的完成时间之后。在** A_i **完成之前，处理器没有空闲时间，如果存在某个时间点** t，**有** $t \leqslant T_i$ **且** $t \geqslant W_i(t)$，**则** A_i **一定能在截止时间内完成**。

推论 2 如果 A_i **能在截止时间内完成，则必定存在某个时间点** t，**使得** $L_i(t)=W_i(t)/t \leqslant 1$。

推论 3 对于所有的 $A_i(1 \leqslant i \leqslant n)$，**如果** $L_i(t) \leqslant 1$ **都成立，则任务集** $S=\{A_1, A_2, \cdots, A_n\}$ **都是可调度的**。

根据推论 1～3，**可以得出以下定理**：

定理 6 对于给定的任务集 $S=\{A_1, A_2, \cdots, A_n\}$，**采用比率单调调度算法，有**

① **当且仅当** $L_i \leqslant 1$ **时，任务** A_i **在任何时候都是可调度的**。

② **当且仅当** $\forall (i)(1 \leqslant i \leqslant n), L_i(t) \leqslant 1$ **时，任务集** S **是可调度的**。

定理 7 如果任务 A_i **与比它优先级更高的所有任务同时到达，则** A_i **的响应时间最长。如果在这种情况下，**A_i **的截止时间能够得到满足，那么在任何时候** A_i **的截止时间都能得到满足**。

定理 7 **的前提条件指出了任务响应时间的最坏情况，其结论的正确性是很显然的**。

定理 8 对任务集 $S=\{A_1, A_2, \cdots, A_n\}$，**设** $T_1 \leqslant T_2 \leqslant \cdots \leqslant T_n$，**采用比率单调调度算法。如果** n **个任务同时到达并且** A_n **的完成时刻不超过它的截止时间** D_n，**则一定存在一个** t **值，使** $W_n(t)=t$。

证明：

式 $W_n(t) = \sum_{i=1}^{n} C_i \lfloor t/T_i \rfloor$，**表示** n **个任务从到达时刻起到** t **这段时间所需要的累计** CPU **时间。因为** A_n **最后完成并且完成时刻不超过它的** D_n，**则一定存在一个** t，**使** $W_n(t) \leqslant t \leqslant D_n$。**当然也就可以找到这个** t **值，它将使** $W_n(t) = t$。

证毕。

根据定理 6～8，**可以通过计算每个任务的完成时刻，来判断它的可调度性**。

假设有任务集 $S=\{A_1,A_2,A_3,A_4,A_5,A_6\}$，任务参数如表 9-1 所列。

表 9-1　任务参数表

任　务	周　期	执行时间	截止时间
A_1	80	20	80
A_2	120	50	120
A_3	250	45	250
A_4	300	75	300
A_5	400	58	400
A_6	600	85	600

很容易判断 A_1 和 A_2 是可调度的。

计算 A_3 的完成时间：

$$t_0 = C_1 + C_2 + C_3 = 115$$
$$t_1 = C_1[t_0/T_1] + C_2[t_0/T_2] + C_3[t_0/T_3] = 135$$
$$t_2 = C_1[t_1/T_1] + C_2[t_1/T_2] + C_3[t_1/T_3] = 185$$
$$t_3 = C_1[t_2/T_1] + C_2[t_2/T_2] + C_3[t_2/T_3] = 205$$
$$t_4 = C_1[t_3/T_1] + C_2[t_3/T_2] + C_3[t_3/T_3] = 205$$

A_3 的完成时刻为 $205 \leqslant D_3 = 250$，因此，A_3 是可调度的。

计算 A_4 的完成时间：

$$t_0 = C_1 + C_2 + C_3 + C_4 = 190$$
$$t_1 = C_1[t_0/T_1] + C_2[t_0/T_2] + C_3[t_0/T_3] + C_4[t_0/T_4] = 280$$
$$t_2 = C_1[t_1/T_1] + C_2[t_1/T_2] + C_3[t_1/T_3] + C_4[t_1/T_4] = 395$$

因为 $t_0 \neq t_1 \neq t_2$，但 $t_2 = 395 > D_4 = 300$，所以 A_4 是不可调度的。

用同样的方法可以检查出 A_5 和 A_6 也是不可调度的。

前面介绍的内容都是针对静态优先级安排的，并得出了在实际应用中进行任务规划和设计时，检验任务截止时间的可满足性的一些非常有用的定理和规则。下面介绍动态优先级安排的情况下，任务调度的相关原理。

9.3.2　动态优先级调度性能分析

动态优先级安排是指在任务运行时，其优先级可以改变。根据任务和系统当时的情况（比如任务等待时间的长短、任务距离截止时间的长度和任务的资源要求等），来确定任务的优先级。这类算法可以有：

- 先来先服务（FCFS）；

- **后来先服务**（LCFS）；
- **最早截止时间优先** EDF(Earliest Deadline First)；
- **最晚截止时间优先** LDF(Latest Deadline First)；
- **未完成工作量最少优先**(minimum unfinished work first)；
- **未完成工作量最大优先**(maximum unfinished work first)。

在上述调度算法中，EDF 是动态优先级安排中较为常用的一种。它也是一种优化的调度算法。该算法的核心思想是：在进行调度时，计算当前的任务中，哪一个任务距离截止时间最近，则该任务的优先级最高，让该任务投入运行。在此，对 EDF 算法的特性作进一步地讨论。

在例 9－3 中，对于两个任务$\{\tau_1,\tau_2\}$，让 $C_1=1, C_2=2.1, T_1=2, T_2=5$。$U=C_1/T_1+C_2/T_2=0.92$。采用 EDF 调度算法，则任务运行过程如图 9－53 所示。

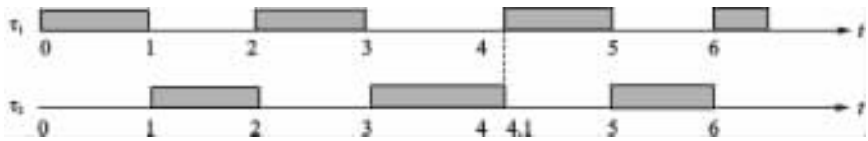


图 9－53　按 EDF 算法调度例 3 中的任务集

可以看出，对同样的任务集，采用 RMS 是不可调度的；但采用 EDF 后，该任务集成为可调度任务集。

定理 9　如果采用 EDF 算法，且仍然有任务不能满足它的截止时间要求，则在该任务的截止时间来到之前，处理器没有空闲时间。

该定理是比较容易理解的。因为有任务发生超时（即不能满足截止时间要求），那么在发生超时的时刻，一定还有任务未执行完成。假设发生超时的任务是 T_i，而其他任务均未超时，则按最早截止时间优先调度原则，在 T_i 的截止时间之前，T_i 一定会成为优先级最高的任务而被调度，因此，CPU 不会有空闲时间。

定理 10　EDF 算法在最坏情况下的处理器利用率为 100 %，即 $\mathrm{WCAU_{EDF}}=100\ \%$。

证明：

假设上述定理不成立。以图 9－54 为例，图中的时刻 T 表示某个任务在该时间点超过截止时间。

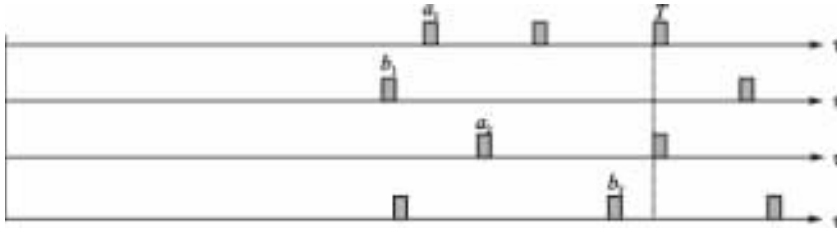


图 9－54　任务超时时间点

让 $a_1, a_2, \cdots$ 是截止时间为 T 的任务的请求时间，$b_1, b_2, \cdots$ 是截止时间在 T 之后的任务的请求时间。

情形 1：没有任何在 $b_1, b_2, \cdots$ 的计算请求能够在 T 之前完成，则在时间范围 $[0, T]$ 内，总的计算时间为

$$\left[\frac{T}{T_1}\right]C_1 + \left[\frac{T}{T_2}\right]C_2 + L + \left[\frac{T}{T_m}\right]C_m$$

因为在 $[0, T]$ 内 CPU 没有空闲时间，并且有任务超过其截止时间，因此有

$$\left[\frac{T}{T_1}\right]C_1 + \left[\frac{T}{T_2}\right]C_2 + L + \left[\frac{T}{T_m}\right]C_m > T$$

这意味着

$$(T/\ T_1)\ C_1 + (T/\ T_2)\ C_2 + L + (T/\ T_m)\ C_m > T$$

即 $U > 1$，这显然是不可能的。

情形 2：假设有一些在时刻 $b_1, b_2, \cdots$ 的计算请求能够在 T 之前执行完成，则有

① 这些计算一定能够在时刻 T' 之前完成，且 $T' < T$；

② 所有在 T' 之前提出的请求必须在 T' 之前完成；

③ 在时间 $[T', T]$ 内的请求所需的计算时间大于 $T - T'$。

即有

$$\left[\frac{T - T'}{T_1}\right]C_1 + \left[\frac{T - T'}{T_2}\right]C_2 + L + \left[\frac{T - T'}{T_m}\right]C_m \geqslant D$$

则

$$\left[\frac{T - T'}{T_1}\right]C_1 + \left[\frac{T - T'}{T_2}\right]C_2 + L + \left[\frac{T - T_t}{T_m}\right]C_m \geqslant D > T - T'$$

这意味着 $U > 1$，这显然是不可能的。

证毕。

由定理 10 可以得到如下推论：

对任务集 $S = \{A_1, A_2, \cdots, A_n\}$ 采用最早截止时间优先调度算法，当且仅当 $(C_1/T_1) + (C_2/T_2) + \cdots + (C_n/T_n) \leqslant 1$，所有任务都是可调度的。

证明：

① 设 $(C_1/T_1) + (C_2/T_2) + \cdots + (C_n/T_n) \leqslant 1$ 成立。

让任务周期的最小公倍数为 M_{LCM}，考察时间范围 $[0, M_{\mathrm{LCM}}]$，所有任务占有的累计处理器时间为

$$\frac{M_{\mathrm{LCM}}}{T_1}C_1 + \frac{M_{\mathrm{LCM}}}{T_2}C_2 + \cdots + \frac{M_{\mathrm{LCM}}}{T_n}C_n = \sum_{i=1}^{n}\frac{M_{\mathrm{LCM}}}{T_i}C_i$$

如果到时刻 M_{LCM} 有任务发生超时，即在时刻 M_{LCM} 处有任务未执行完成，则

$$\sum_{i=1}^{n}\frac{M_{\mathrm{LCM}}}{T_i}C_i = M_{\mathrm{LCM}}\sum_{i=1}^{n}\frac{C_i}{T_i} > M_{\mathrm{LCM}}$$

从而

$$\sum_{i=1}^{n}\frac{C_i}{T_i}>1$$

这与假设 $\sum_{i=1}^{n}\frac{C_i}{T_i}\leqslant 1$ 相矛盾，所以几个任务都是可调度的。

② 假设 n 个任务都是可调度的。

在时间范围$[0,M_{\mathrm{LCM}}]$内，如果所有任务都不发生超时，这意味着到时刻 M_{LCM} 处，所有任务都已完成，则

$$\sum_{i=1}^{n}\frac{M_{\mathrm{LCM}}}{T_i}C_i=M_{\mathrm{LCM}}\sum_{i=1}^{n}\frac{C_i}{T_i}>M_{\mathrm{LCM}}$$

于是

$$\sum_{i=1}^{n}\frac{C_i}{T_i}\leqslant 1$$

证毕。

在调度时刻，可能距截止时间最近的任务有多个，此时，这几个任务的优先级相同。当 $\sum_{i=1}^{n}\frac{C_i}{T_i}\leqslant 1$ 时，对优先级相同的多个任务，任意优先调度哪个任务，都不会发生超时；但当 $\sum_{i=1}^{n}\frac{C_i}{T_i}>1$ 时，对相同优先级任务的调度方式不同，发生超时的任务就会不同。对上述情况，可分别采用短周期任务优先和短计算任务优先这两种方式进行调度。

先来看一个例子。假设有任务集 $S=\{A_1,A_2,A_3\}$，其参数为 $T_1=3,C_1=1.5,T_2=4,C_2=1.6,T_3=6,C_3=1$。

因为 $\sum_{i=1}^{n}\frac{C_i}{T_i}>1$，所以有任务发生超时。

如果采用短周期任务优先，则任务的执行过程如图 9-55 所示。

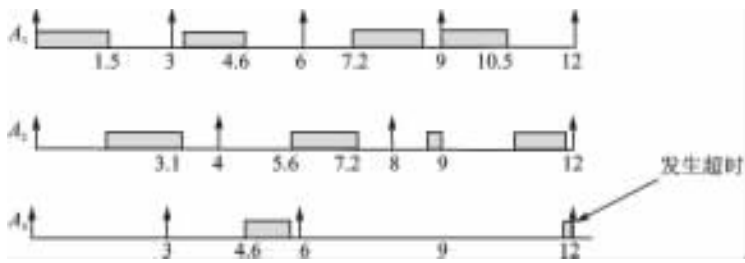


图 9-55　短周期任务优先

如果采用短计算任务优先，则任务的执行过程如图 9-56 所示。

在 $t=3.1$ 处，A_1 和 A_3 距截止时间最短且相同。采用短周期任务优先，则优先调度 A_1。

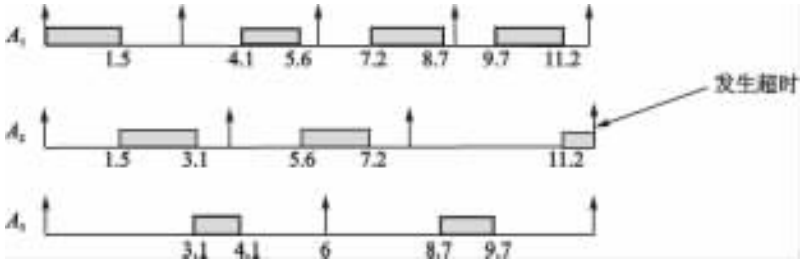


图 9－56　短计算任务优先

在时刻 $t=9$ 处，由于 A_1 的到达，产生一次调度；此时三个任务的截止时间相同，则依次调度 A_1，A_2，A_3，结果使 A_3 仅运行 0.2 个时间单位后发生超时。

采用短计算任务优先，在 $t=9.7$ 处，A_3 完成运行，此时 A_1 和 A_2 的截止时间相同，优先调度 A_1，使 A_2 在运行 0.8 个时间单位后发生超时。

由于假设 $D_i=T_i$，在调度时间，若有 k 个任务的截止时间相同，则在截止时间处，周期一定是这 k 个任务的周期的公倍数。

当 $\sum_{i=1}^{n}\frac{C_i}{T_i}>1$ 时，需要检查任务超时的情况，有如下定理：

定理 11　如果 $\sum_{i=1}^{n}[t/T_i]C_i\leqslant[t/T_k]T_k$，则在时刻 t 之前，任务 A_k 不会发生超时。

证明：

在时刻 t，分为以下两种情况：

① 如果 t/T_k 能整除，则意味着 t 恰在 A_k 的周期点，此时 $[t/T_k]T_k=t$。当 $\sum_{i=1}^{n}[t/T_i]C_i\leqslant[t/T_k]T_k=t$ 时，由于假设 $D_k=T_k$，意味着在时刻 t 之前所有任务均在 D_k 之前完成，所以 A_k 不会发生超时。

② 如果 t/T_k 不能整除，则 $[t/T_k]T_k\leqslant t$，并且 $[t/T_k]T_k$ 表示在时刻 t 之前，A_k 最靠近 t 的那个周期点；当 $\sum_{i=1}^{n}[t/T_i]C_i\leqslant[t/T_k]T_k$，意味着所有任务均在 A_k 的周期点（该周期点在 t 之前）之前完成，所以 A_k 不会发生超时。

证毕。

当 $\sum_{i=1}^{n}\frac{C_i}{T_i}>1$ 时，检查每一个截止时间点。由于 $D_i=T_i$，因此从 A_1 开始检查时间点 T_1，$2T_1,3T_1,\cdots,M_{\mathrm{LCM}}=\mathrm{LCM}(T_1,T_2,\cdots,T_n)$。在某一个 kT_1 处，若 $\sum_{i=1}^{n}\left[\frac{kT_1}{T_i}\right]C_i>kT_i$，则意味着有任务在时间点 $t=kT_1$ 处发生超时；再检查时间点 $T_2,2T_2,3T_2,\cdots,M_{\mathrm{LCM}}$，以此类推，直到检查最后一个任务的 $T_n,2T_n,\cdots,M_{\mathrm{LCM}}$。

如果第 i 个任务在时间点 jT_i 处有 $\sum_{i=1}^{n}\left[\frac{jT_i}{T_i}\right]>jT_i$，则记 $t_{ij}=\{jT_i\}$，于是有

$$t_{1j}=\{jT_1\},t_{2j}=\{jT_2\},\cdots,t_{nj}=\{jT_n\}$$

对任意一个时间点 $jT_i\in t_{ij}$，如果不存在一个 $VT_u(VT_u\in t_{uv}\parallel u=1,2,\cdots,n\wedge u\neq i)$，使 $jT_i=VT_u$，则在时间点 jT_i，任务 A_i 将发生超时。

如果存在两个时间点 jT_i 和 VT_u，有 $jT_i=VT_u$，则意味着该时间点为 T_i 和 T_u 的公倍数，且 A_i 和 A_u 具有相同的截止时间，发生超时的情况有三种可能：

① A_i 和 A_u 都发生超时；

② A_i 发生超时；

③ A_u 发生超时。

为了判断上述三种情况，需确定采用什么算法决定其优先级。

(1) 短周期任务优先

让 $T_{\min}=\min(T_i,T_u)$，$T_{\max}=\max(T_i,T_u)$，如果 $\sum_{i=1}^{n}\left[\frac{jT_i}{T_k}\right]C_k-\left[\frac{jT_i}{T_{\max}}\right]C_{\max}\leqslant 1$，则 $A_{\max}$ 在时间点 jT_i 处发生超时，$A_{\min}$ 能满足截止时间要求。

如果 $\sum_{i=1}^{n}\left[\frac{jT_i}{T_k}\right]C_k-\left[\frac{jT_i}{T_{\max}}\right]C_{\max}>1$，则意味着优先调度 $T_{\min}$ 时，$A_{\min}$ 仍将发生超时，所以 A_i 和 A_u 都会发生超时。

(2) 短计算任务优先

让 $C_{\min}=\min(C_i,C_u)$，$C_{\max}=\max(C_i,C_u)$，则优先调度 $A_{\min}$，其后的过程与(1)相同。

若有多个任务(其数目大于 2)的截止时间相同，且在某时间点有任务发生超时，则分析方法与上述方法相同。

下面对前面两类调度算法作简单的小结。

① 静态优先级安排：当 $\mathrm{WCAU_{RMS}}=0.69$ 时，则 RMS 是优化的；

② 动态优先级安排：当 $\mathrm{WCAU_{EDF}}=1.00$ 时，则 EDF 是优化的。

上述结论对于构建一个实际的实时系统是非常有用的。

9.3.3 任务计算时间的确定

任务计算时间即是任务完成一次执行的时间。在系统设计时(实际编码之前)，为满足系统响应时间，通常应规定一个任务必须在什么时间内完成，这涉及到中断响应时间、所选用的调度算法(关系到任务优先级)以及该任务占有处理器资源后的一次执行。当中断响应时间和调度算法确定后，则任务计算时间主要取决于该任务本身所使用的计算方法和程序量(代码行)。由于还未进行编码，还不能精确估计任务执行时间，但可以进行模拟并粗略估计。如果一个任务执行时间太长(如计算方法过于复杂、需要完成的功能太多等)而无法满足截止时间

要求，则需要简化计算方法或者重新进行任务划分。

在编码完成后，可以通过任务的实际运行来测试任务执行时间。但是，由于多任务系统中，系统的工作过程是一个动态的过程，同一个任务在不同时间的执行情况可能是不同的，比如某一个任务 T_i，其最坏情况下的执行时间涉及到该任务的最长执行路径；所需资源已经被其他任务占有；有多个优先级更高的任务已经在任务就绪队列上等情况，因此必须测试在最坏情况下的执行时间。

9.3.4　系统开销

上述的例子中没有考虑系统中的时间开销。在一个实际的系统中，中断响应时间（开关中断、中断嵌套和中断服务程序执行时间等）、系统响应时间（最坏响应时间等）、任务切换时间（开关调度两种不同情况）、任务优先级反转和任务间通信等诸多因素都将影响任务的时限可满足性。下面将对这些情况以及它们对任务实现的影响进行论述。

（1）中断响应与任务切换

假设有一个任务 A_1 正处于运行之中，当运行到时刻 t_1 时，有一个优先级更高的任务 A_2 所对应的事件产生了中断，则系统的响应过程如图 9－57 所示。

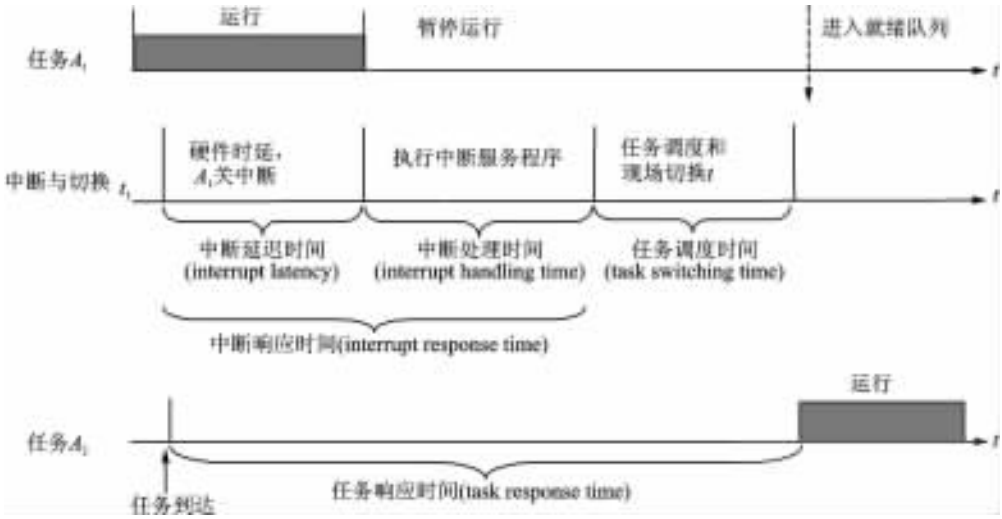


图 9－57　任务响应时间分析

任务响应时间＝中断延迟时间＋中断处理时间＋任务调度时间＝
中断响应时间＋任务调度时间

假设一个任务原有的运行时间为 C，可令“C＋任务响应时间”作为实际的任务运行时间来进行任务的时限分析。

(2) 其他系统时间开销

假设所有任务都以中断方式到达系统,此时可有两种情况:第一,通过硬件来保证仅使那些优先级高于当前运行任务的优先级的任务所对应的中断才能到达系统;第二,中断的到达与当前运行任务的优先级无关,即当前任务可能被任何优先级别的任务所中断。对前一种情况,一旦有中断发生,就会产生任务抢占。对后一种情况,如果中断对应的任务的优先级高于当前运行任务的优先级,才会产生任务抢占;否则系统在进行中断处理以后仍返回到被中断的任务执行。显然,第一种情况要简单一些,并且系统时间的开销更小。因此,这里只分析第二种情况。

除了系统用于中断的时间开销外,还有任务调度所需的时间。因此,引入下列时间参数:

- C_{bd} 中断且发生了任务剥夺(即抢占)所需的时间,包括:中断响应;执行任务调度程序并确定下一个投入运行的任务;保存被中断任务的状态;从就绪队列中取出新任务并投入运行。
- C_{nbd} 中断但未发生任务剥夺(即抢占)所需的时间,包括:中断响应;执行任务调度程序并确定下一个投入运行的任务;返回到被中断任务。
- C_{zw} 任务正常完成并进行新一轮调度所需的时间,包括:将已完成任务挂到任务休眠队列;执行任务调度程序;从就绪队列中取出新任务执行。

影响任务运行时间的另一个因素是由于资源共享而引起的任务互斥。在这种情况下,一个任务可能因得不到资源而被挂起,直到该资源被占有它的任务所释放;并且,由于资源共享还可能造成如 5.5 节所述的优先级反转。

假设共有 u 个临界资源 $Z_1, Z_2, \cdots, Z_u$,对应 u 个信号 $S_1, S_2, \cdots, S_u$,每个临界区操作的时间长度分别为 $L_1, L_2, \cdots, L_u$。每当有一个任务进入临界区就禁止对其再作任务调度,直到该任务退出临界区,再开调度。

当一个任务进入临界区而关调度时,显然只会对比它优先级高的任务产生影响,并且在一个任务的临界区操作期间,若有多个更高优先级的任务同时到达,也只会对优先级最高的那个任务有直接影响,因为即使不关调度,也只会调度优先级最高的那个任务。由于是抢占调度方式,一个高优先级任务的一次到达至多被低优先级任务的关调度阻塞一次。

由于不同任务的临界区操作的时间不同,并且高优先级任务的到达时刻是否正好处于运行任务的临界区中是随机的,作为最坏情况分析,可假定每次的阻塞时间 $C^* = \max(L_1, L_2, \cdots, L_u)$。显然多个任务可能共享同一个临界资源,也还可能一个任务与其他的任务共享不同的临界资源。比如,假设有任务集$\{A_1, A_2, \cdots, A_{15}\}$,各个任务优先级的安排为从 $A_1 \sim A_{15}$ 由高到低。其资源共享情况如图 9-58 所示。

S_1,S_2和 S_3都被多个任务所共享,并且 A_3既和 A_1,A_5共享 S_1,又和 A_8,A_{10}共享 S_3。因此,A_3既可能因试图使用 S_1所对应的资源被 A_5所阻塞,也可能因试图使用 S_3对应的资源被 A_8或 A_{10}所阻塞。如前所述,一个任务的一次运行至多因关调度被阻塞一次,而优先级最低的

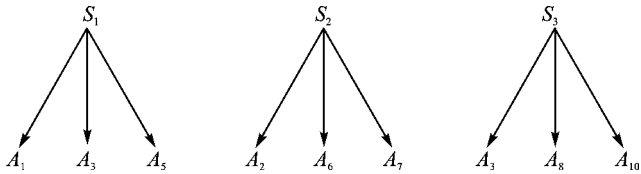


图 9－58　资源共享情况

任务则不会因关调度而被阻塞。

为了反映一个任务 A_j 是否会因关调度而被阻塞，可构造如下集合：

$$B_j=\{A_k \parallel \text{具有共享资源的所有 } A_k(1<k\leqslant n)\text{且 } k>j\}$$

例如，根据图 9－58 所反映的资源共享情况，可得

$$B_1=\{A_2,A_3,A_5,A_6,A_7,A_8,A_{10}\}$$
$$B_2=\{A_3,A_5,A_6,A_7,A_8,A_{10}\}$$
$$B_3=\{A_5,A_6,A_7,A_8,A_{10}\}$$
$$B_4=\{A_5,A_6,A_7,A_8,A_{10}\}$$
$$B_5=\{A_6,A_7,A_8,A_{10}\}$$
$$B_6=\{A_7,A_8,A_{10}\}$$
$$B_7=\{A_8,A_{10}\}$$
$$B_8=\{A_{10}\}$$
$$B_9=\{A_{10}\}$$
$$B_{10}=B_{11}=B_{12}=B_{13}=B_{14}=B_{15}=\phi$$

在得到集合 B_j 后，让

$$Q_j=\begin{cases}1 & (B_j\neq\phi)\\ 0 & (B_j=\phi)\end{cases}$$

当 $B_j=\phi$ 时，意味着 A_j 不会因关调度而被阻塞；只有当 $B_j\neq\phi$，A_j 才会因关调度而被阻塞。如果 A_j 会被阻塞（即 $Q_j=1$），则在最坏情况下，A_j 的每一次运行都被阻塞一次。因此到时刻 t，A_j 至多被阻塞 $\lfloor t/T_j\rfloor$ 次，被阻塞的时间为 $C^*\lfloor t/T_j\rfloor$。

在对实际系统进行时间性能分析时，应考虑 C_{bd}，C_{nbd}，C_{zw} 等时间开销，比如可以将任务计算时间 C_j 加上 C_{bd} 和 C_{zw}，即 $(C_j+C_{\text{bd}}+C_{\text{zw}})$ 作为任务的计算时间进行分析。按照 RMS 算法，从时刻 0 到时刻 t，所要求的累计 CPU 时间为

$$W_i(t)=\sum_{j=1}^{i}C_j\lfloor t/T_j\rfloor$$

考虑系统其他时间开销，则从时刻 $0\sim t$，所要求的累计 CPU 时间为

$$W_i(t)=\sum_{j=1}^{i}(C_j+C_{\text{bd}}+C_{\text{zw}})\lfloor t/T_j\rfloor+\sum_{k=i+1}^{n}C_{\text{nbd}}\left\lceil\frac{t}{T_k}\right\rceil$$

该表达式中，还未考虑因采用关调度方式来防止可能出现的优先级反转而产生的时间开销。

需要说明的是，一般情况下 C^*，C_{bd}，C_{nbd} 等时间开销与任务计算时间 C 相比，可能要小得多。在这种情况下，如果系统的实时性能要求不是非常高，为了简化分析算法，也可以不考虑 C^*，C_{bd}，C_{nbd} 等时间。

思考题

9.1 嵌入式系统开发模式有什么特点？为什么？

9.2 在选择嵌入式微处理器时应该考虑哪些方面的因素？

9.3 请以一款具体的嵌入式操作系统及配套开发工具为例，结合本章分析的要素或者使用该操作系统和工具的经验，说明它们在支持嵌入式应用软件开发上的优缺点。

9.4 在 DARTS 方法中定义的任务划分原则中，使用哪些原则划分的任务需设置较高的优先级？为什么？

9.5 DARTS 方法中定义的任务间接口类型有哪些？请以使用过的一款嵌入式操作系统为例，说明怎样将 DARTS 提出的任务接口类型映射到该操作系统的相关机制上。

9.6 使用 UML 方法来分析设计实时软件有哪些好处？怎样将它与 DARTS 方法结合起来？

附录　CD－ROM 内容

为了加强理论与实践的结合，作者专为“嵌入式实时操作系统及应用开发”教材配套了实验系统。该实验系统包括嵌入式实时操作系统和集成开发工具，提供了丰富的实验用例。读者利用 PC 机就可以自己动手搭建嵌入式系统的开发平台，熟悉应用开发，更好地学习和理解嵌入式系统的基础知识。

实验系统由下列内容构成。

- 嵌入式实时软件开发平台“道系统”；
- 实验用例程序；
- 实验指导书；
- 嵌入式硬件平台或仿真平台。

下面是对这些内容的简要说明。

1. 嵌入式实时软件开发平台“道系统”

嵌入式实时软件开发平台“道系统”包括以下内容：

- 嵌入式实时操作系统 DeltaOS　嵌入式实时内核 DeltaCORE、嵌入式 TCP/IP DeltaNET 和嵌入式文件系统 DeltaFILE。
- 嵌入式集成交叉开发环境 LambdaTOOL　集成开发环境 LambdaIDE、交叉编译器 LambdaGCC、交叉调试器 LambdaGDB 和目标监控器 LambdaTRA 等。

2. 实验用例程序

配套该实验系统的用例有如下内容：

- 嵌入式软件开发环境建立实验　指导嵌入式软件开发环境建立的详细过程，让读者迅速建立起交叉开发或仿真开发的概念。
- 嵌入式实时内核实验　任务管理、同步互斥与通信、中断管理、内存管理和时钟管理等。
- 嵌入式 TCP/IP 实验　TCP 客户/服务器、UDP 客户/服务器和 PING 应用等。
- 嵌入式文件系统实验　文件系统的初始化和启动、文件和目录的管理以及文件系统 API 的使用等。

3. 实验指导书

实验指导书是对配套实验的详细说明，对每个实验均从以下几方面进行描述。

- 实验目的；
- 实验原理；
- 实验程序结构；

● 实验步骤及结果。

4. 嵌入式硬件平台及仿真平台

在该实验系统的交叉开发环境或仿真开发环境中，宿主机均使用 PC 机，目标机可以使用嵌入式仿真 PC 平台或两种主流的 32 位嵌入式硬件平台，读者可根据需要选择。

(1) 嵌入式仿真 PC 平台

使用仿真 PC 平台，嵌入式软件的开发在同一台 PC 机上进行。这对于学习嵌入式操作系统是很方便的。

(2) ARM7 S3C4510B 嵌入式硬件平台

这是一种低端的嵌入式硬件平台，具备基本的外围接口；除了前面描述的嵌入式操作系统系列实验外，读者还可以在此平台上学习嵌入式微处理器 ARM 的编程，并开发一些硬件接口实验。

(3) ARM7 S3C44B0X 嵌入式硬件平台

该款硬件平台具备丰富的外围接口及设备，除了嵌入式操作系统实验、ARM 编程及硬件接口实验外，读者还可以在此平台上学习更加复杂的设备驱动程序开发，完成更加复杂和生动的实验内容。

5. CD－ROM 内容说明

在本书附随的 CD－ROM 中，提供了如下内容：

● 嵌入式实时软件开发平台“道系统”安装程序；

● 实验用例程序；

● 实验指导书。

注 1：

对于上述内容，在 CD－ROM 中只提供针对嵌入式仿真 PC 平台的版本，读者可在此基础上进行嵌入式软件的仿真开发。读者如果需要针对 ARM7 S3C4510B 或 ARM7 S3C44B0X 的版本以及相关的硬件平台进行交叉开发，请通过下列 E－mail 地址与作者联系。

E－mail 地址	lluo@uestc.edu.cn lrchen@uestc.edu.cn	请在邮件标题中注明： *嵌入式实验系统咨询*

注 2：

实验系统的内容会不断更新，因此请读者留意以下网址发布的信息，以便获取最新的内容。

http://www.coretek.com.cn:8000

注 3：

读者如果对教材及实验系统有任何疑问，均可访问上述地址并留言，或给作者发送 E－mail。

参考文献

1 [美]米奇欧·卡库. 远景——二十一世纪的科技演变. 徐建译. 海口：海南出版社，2000

2 [美]Wayne Wolf. 嵌入式计算系统设计原理. 孙玉芳，梁彬，罗保国，等译. 北京：机械工业出版社，2002

3 金惠华. 蓬勃发展的嵌入式计算机结构. 中国计算机世界，2000

4 Arnold Berger. 嵌入式系统设计. 吕骏译. 北京：电子工业出版社，2002

5 Gupta R K. Embedded Processors Project Report for ICS 212. March 2000

6 David Seal. ARM Architecture Reference Manual. 2nd ed. Addison, Wesley, 2000

7 ARM Limited. AMBA Specification. Rev 2.0. 1999

8 高鹏，陈咏恩. AMBA 总线及应用. 半导体技术，2002，27(9)

9 刘鑫，周金莲. CompactPCI 总线工控机技术的现状与应用. 电子技术应用，2002(7)

10 MIPS Technologies Inc. MIPS32 Architecture for Programmers. June 2003

11 刘国安. PCI 总线技术. 仪表技术，2003(3)

12 刘乙成，周祖成，陈尚松. SOC 片上总线技术的研究. 半导体技术，2003，28(2)

13 李剑，赵鹏程，汤建彬. 32 位 ARM 嵌入式处理器的调试技术. 电子技术应用，2003(3)

14 熊光泽，罗蕾. 32 位微处理器嵌入式实时软件开发与调试技术. 计算机应用，1995，15(4)

15 Mercer C W. An Introduction to Real-Time Operating Systems: Scheduling Theory. School of Computer Science, Carnegie Mellon University, November 1992

16 Herman Bruyninckx, Leuven K U. Real-Time and Embedded Guide. Mechanical Engineering, Leuven, Belgium, 2001

17 Krishma C M, Shin K G. Real-Time Systems. Beijing: Tsinghua University Press, 2001

18 Lampson B W, Redell D D. Experiences with Processes and Monitors in Mesa. Commun. ACM, Feb. 1980, 23(2):105～117

19 Lehoczky J P, Sha L, Strosnider J K. Enhanced Aperiodic Responsiveness in a Hard Real-Time Environment. In Proceedings of 8th IEEE Real-Time Systems Symposium, December 1987. 261～270

20 Liu C L, Layland J W. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. Journal of the ACM, 1973

21 Lui Sha, Ragunathan Rajkumar, Lehoczky J P. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers, September 1990, 39(9)

22 Rajkumar R. Task Synchronization in Real-Time Systems: [PhD Thesis]. Carnegie Mellon University, August 1989

23 Rajkumar R. Synchronization in Real-Time Systems: A Priority Inheritance Approach. Kluwer Academic Publishers, 1991

24 Ramamritham K, Stankovic J A. Scheduling Algorithms and Operating Systems Support for Real-Time. Systems Proceedings of the IEEE, January 1994, 82(1)

25 Sprunt B, Sha L, Lehoczky J P. Aperiodic Task Scheduling for Hard Real-Time Systems. The Journal of Real-Time Systems, 1989, (1): 27～60

26 Strosnider J K, Lehoczky J P, Lui Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real - Time Environments. IEEE Trans. On Computers, Jan. 1995, 44(1): 73～91

27 Tindell K, Hansson H. Real Time Systems by Fixed Priority Scheduling: [Technical Report]. Departament of Computer Systems, Uppsala University, 1997

28 **李允，熊光泽，程红蓉. 普及计算终端设备的电源管理技术研究. 电子科技大学学报**，2001，30(5)：497～502

29 [**美**] Labrosse J J. **嵌入式实时操作系统** μC/OS - II. **第** 2 **版. 邵贝贝，宫辉，蒋俊峰，等译. 北京：北京航空航天大学出版社**，2003

30 William Stallings. **操作系统：精髓与设计原理. 第** 3 **版(英文影印版). 北京：清华大学出版社**，1998

31 **眷新燕. 实时操作系统测试技术的研究与实现：[硕士论文]. 成都：电子科技大学**，1998

32 **程红蓉. 一种实时、嵌入式操作系统内核** DeltaCORE **的设计与实现：[硕士论文]. 成都：电子科技大学**，2001

33 Luolei, Zhu Minyuan. Partitioning Based Operating System: A Formal Model. ACM SIG Operating Systems Review, July 2003

34 Luolei, Zhu Minyuan . A Formal Semantic Definition of DEVIL. ACM Sigplan Notices, April 2003

35 Zhu Minyuan, Luolei, Xiong Guangze. A Provably Correct Operating System: DeltaCore. ACM SIG Operating Systems Review, January 2001, 35(1)

36 Zhu Minyuan, Luolei, Xiong Guangze. The Minimal Model of Operating Systems. ACM SIG Operating Systems Review, July 2001(35)

37 **熊光泽，罗蕾. 嵌入式软件技术的现状与发展动向. 计算机应用**，2000，20(7)

38 **罗蕾，熊光泽. 实时多任务应用最坏情况设计的研究. 电子科技大学学报**，1997(2)

39 **雷航，罗蕾，熊光泽.** Rate - Monotonic **调度方式下的超时故障分析. 微型计算机**，1996，16(2)

40 **熊光泽，罗蕾，喻梅. 嵌入式系统开放式交叉开发环境的研究. 计算机与数字工程**，1995，23(6)

41 **熊光泽，罗蕾. 嵌入式实时操作系统配置技术的研究. 微处理机**，1992(1)

42 **孔祥营，柏桂枝. 嵌入式实时操作系统** VxWorks **及其开发环境** Tornado. **北京：中国电力出版社**，2001

43 Sloss A N. Interrupt Handling. http://www.13thmonkey.org/documentatio/misc/HAI.pdf

44 Martin Carlsson, Jakob Engblom, Andreas Ermedahl, et al. Worst - Case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real - Time Operating System. In Proc. Workshop on Real-Time Tools, August 2002

45 Wade Waler, Cragon H G. Interrupt Processing in Concurrent Processors. IEEE Computer, 1995, 28(6): 36～46

46 **钟汉如，王创生. 嵌入式** Linux **的中断处理与实时调度的实现机制. 计算机工程**，2002，28(10)

47 **屠征，谢康林. 嵌入式实时操作系统中对时钟中断服务程序的改进. 计算机工程**，2003，29(6)

48 Stefan Savage, Hideyuki Tokuda. Real Time - Mach Timers: Exporting Time to the User. USENIX MACH Symposium, 1993. 111～118

49 Miller F W. Simple Memory Protection for Embedded Operating System Kernels. Proceedings of the FREENIX

Track: 2002 USENIX Annual Technical Conference. Monterey, California, USA, June 2002. 299～308

50 **李江，常葆林. 嵌入式操作系统中的** I/O **驱动软件. 计算机工程**，2000 (6)：90～102

51 Herman Bruyninckx. Real－Time and Embedded Guide. http://people. mech. kuleuven. ac. be/～bruyninc/rthowth

52 **桑楠. 嵌入式系统原理及应用开发技术. 北京：北京航空航天大学出版社**，2002

53 Liu C L, Layland J W. Scheduling Algorithms for Multiprogramming in a Hard Real－Time Environment. Journal of ACM, 1973, 20(1): 46～61

54 Wei Kuanshih, Jane W S. Liu C L. Modified Rate－Monotonic Algorithm for Scheduling Periodic Jobs with Deferred Deadlines. IEEE Transactions on Software Engineering, 1993, 19(12): 1171～1179

55 Harbour M G, Klein M H, Lehoczky J P. Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems. IEEE Transactions on Software. Engineering, 1994, 20(1): 13～28

56 Hong Jiawei, Tan Xiaonan, Don Towsley. A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real－Time System. IEEE Trans. On Computer, 1989, 38(12): 1736～1744

57 Goel A L, Kazu Okumoto. Time－Dependent Error－Detection Rate Model for Software Reliability and Other Performance Measures. IEEE Trans. On Reliability, August 1979, 28(3): 206～211

58 **陆廷孝，郑鹏洲，何国伟，等. 可靠性设计与分析. 北京：国防工业出版社**，1995

59 **阮镰，刘斌，陈雪松. 软件可靠性测试及其测试环境. 测控技术**，2000，19(2)

60 **刘杰. 软件可靠性评测及其应用探讨**. http://www. scstlab. com. cn/techdoc/liuj. htm

61 **郑竑宇，宋寅卯. 运用** UML **分析设计占先式实时内核. 单片机与嵌入式系统应用**，2003(9)

62 [**美**]Grady Booch, James Rumbaugh, Ivar Jacobson. UML **用户指南. 邵维忠，张文娟，孟祥文，等译. 北京：机械工业出版社**，2001

63 [**美**]Grady Booch, James Rumbaugh, Ivar Jacobson. UML **参考手册. 姚淑珍，唐发根，等译. 北京：机械工业出版社**，2001

64 [**美**] Martin Fowler, Kendall Scott. UML **精粹：标准对象建模语言简明指南. 第** 2 **版. 徐家福译. 北京：清华大学出版社**，2002

65 [**美**] Bruce Powel Douglass. UML **实时系统开发. 第** 2 **版(英文影印版). 北京：科学出版社**，2003

66 [**美**] Mellor S J, Balcer M J. Executable UML **技术内幕. 英文影印版. 北京：科学出版社**，2003

67 [**美**] Hassan Gomaa. **并发与实时系统软件设计. 姜昊，周靖，译. 北京：清华大学出版社**，2003

68 UML 2.0 Infrastructure Final Adopted Specifcation. http://www. omg. org, 2003

69 [**美**] IBM Rational **公司**. Rational Rose RealTime **用户手册**，2002

70 de Jong G. A UML－Based Design Methodology for Real－Time and Embedded Systems. Proceedings of Date 2002

71 Martin G, Lavagno L, Guerin J L. Embedded UML: A Merger of Real-Time UML and Co-Design. Proceedings of Data 2001

72 **赖明志，尤晋元. 使用时间化自动机形式化带有时间扩展的** UML **状态图. 计算机应用**，2003(8)

73 **唐英，李志蜀. 使用** UML **分析设计嵌入式系统. 计算机应用研究**，2002

74 **俞晓箴.** UML **的实时应用开发环境. 湖北教育学院学报**，2002，19(2)

75 **全国量和单位标准化技术委员会**. GB 3100～3102－93 **量和单位**. **北京**：**中国标准出版社**，1994

76 OMG. Unified Modeling Language Specification. Version 1.5，2003

77 OMG. UML Profile for Schedulability，Performance，and Time Specification. Version 1.0，2003

78 OMG. UML 2.0 Infrastructure Specification. 2003

79 Bran Selic. A UML Profile for Modeling Complex Real－Time Architectures. http://www.rational.com，2001

80 Selic B，Rumbaugh J. Using UML for Modeling Complex Real-Time Systems. http://www.objectime.com/otl/technical/，Mar. 1998

81 Mellor S J，Wolfe J R. Model－Based Embedded Systems Development with ^{XT}UML. http://www.projtech.com，2002