国外计算机科学经典教材

Mc Graw Hill Education

Embedded Systems

Architecture, Programming and Design

嵌入式系统

——体系结构、编程与设计

(印度) Raj Kamal 编著
陈曙晖 等译
王续进 审校

清华大学出版社

Embedded Systems

Architecture, Programming and Design

本书是嵌入式系统的基础教程，主要面向初级系统设计人员。书中详细介绍了嵌入式系统最基本的软件和硬件要素，以及将代码嵌入到系统中的一些软件技术和接口技术。

本书主要特色

- 全面介绍了嵌入式系统的编程原理、OS、RTOS函数和进程间同步
- 单处理器和多处理器系统的程序建模和软件设计实践
- 同时涵盖两种实时操作系统——mC/OS-II和VxWorks
- 提供了关于消费电子产品、通信、汽车电子产品和片上安全事务系统的案例研究，同时阐述了RTOS编程原理
- 每章包括插图、示例、关键词及其定义、问题回顾和实践练习
- Java 2 Micro版本的使用，针对手持设备的嵌入式C++编程，嵌入式软件的C语言编程

ISBN 7-302-09624-4

9 787302 096245 >

定价：69.00元

Mc Graw Hill Education

http://www.mheducation.com

国外计算机科学经典教材

嵌入式系统

——体系结构、编程与设计

(印度) Raj Kamal 编著

陈曙晖 等译

王继进 审校

清华大学出版社

北 京

Raj Kamal
Embedded Systems:Architecture, Programming and Design
EISBN: 0-07-049470-3

图书在版编目(CIP)数据

嵌入式系统——体系结构、编程与设计/(印度)卡莫尔(Kamal,R.)编著；陈曙晖等译. —北京：清华大学出版社，2005.5

书名原文：Embedded Systems：Architecture, Programming and Design

(国外计算机科学经典教材)

ISBN 7-302-09624-4

Ⅰ. 嵌… Ⅱ. ①卡… ②陈… Ⅲ. 微型计算机—系统设计—教材 Ⅳ. TP360.21

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
社 总 机：010-62770175　　**客户服务**：010-62776969
组稿编辑：曹　康
文稿编辑：崔　伟
封面设计：康　博
版式设计：康　博
印 刷 者：北京市昌平环球印刷厂
装 订 者：北京鑫海金澳胶印有限公司
发 行 者：新华书店总店北京发行所
开　　本：185×260　**印张**：36.75　**字数**：941千字
版　　次：2005年5月第1版　2005年5月第1次印刷
书　　号：ISBN 7-302-09624-4/TP·6678
印　　数：1～4000
定　　价：69.00元

出 版 说 明

近年来，我国高等学校的计算机学科教育进行了较大的改革，急需一批门类齐全、具有国际水平的计算机经典教材，以适应当前的教学需要。引进国外经典教材，可以了解并吸收国际先进的教学思想和教学方法，使我国的计算机学科教育能够与国际接轨，从而培育更多具有国际水准的计算机专业人才，增强我国信息产业的核心竞争力。Pearson、Thomson、McGraw-Hill、Springer、John Wiley 等出版集团都是全球最有影响的图书出版机构，它们在高等教育领域也都有着不凡的表现，为全世界的高等学校计算机教学提供了大量的优秀教材。为了满足我国高等学校计算机学科的教学需要，我社计划从这些知名的国外出版集团引进计算机学科经典教材。

为了保证引进版教材的质量，我们在全国范围内组织并成立了“清华大学计算机外版教材编审委员会”（以下简称“编委会”），旨在对引进教材进行审定、对教材翻译质量进行评审。“编委会”成员皆为全国各类重点院校教学与科研第一线的知名教授，其中许多教授为各校相关院、系的院长或系主任。“编委会”一致认为，引进版教材要能够满足国内各高校计算机教学与国际接轨的需要，要有特色风格，有创新性、先进性、示范性和一定的前瞻性，是真正的经典教材。为了保证外版教材的翻译质量，我们聘请了高校计算机相关专业教学与科研第一线的教师及相关领域的专家担纲译者，其中许多译者为海外留学回国人员。为了尽可能地保留与发扬教材原著的精华，在经过翻译和编辑加工之后，由“编委会”成员对文稿进行审定，以最大程度地弥补和修正在前面一系列加工过程中对教材造成的误差和瑕疵。

由于时间紧迫和能力所限，本套外版教材在出版过程中还可能存在一些不足和遗憾，欢迎广大师生批评指正。同时，也欢迎读者朋友积极向我们推荐各类优秀的国外计算机教材，共同为我国高等学校的计算机教育事业贡献力量。

清华大学出版社

国外计算机科学经典教材

编审委员会

译 者 序

嵌入式系统是专用于应用或者产品，并且基于计算机的一种系统。我们每天面对的很多设备中都使用了嵌入式系统，例如：微波炉、洗衣机、电视机、汽车、数码相机，等等。近年来，嵌入式系统的发展呈现出几个特点。首先，在硬件领域，SOC 和 SOPC 技术发展迅速，两大 FPGA 厂商 Xilinx 和 Altera 都推出了自己内含 CPU 及其外围电路的 FPGA，与此相应的 IP 包技术也发展迅速；其次，嵌入式软件的开发工具和操作系统日趋完善，Wind River 公司的 VxWorks 统领了嵌入式操作系统和开发平台的半壁江山，而免费的嵌入式 Linux 系统也得到了广泛应用；另外，应用领域不断扩展，尤其是消费电子领域的扩张有力地推进了嵌入式系统的发展。

随着各个应用领域对智能设备的需求迅速增长，嵌入式系统的开发也逐渐成为软硬件领域的研究热点，从而促使了近年来嵌入式系统的研究深度和广度不断增加，这使得更多的开发人员渴望了解嵌入式系统的原理和开发过程。我们本着为嵌入式系统的系统设计人员提供更有价值、更加通俗易懂的参考书的目的，认真地翻译了本书。

可以毫不夸张地说，这是嵌入式系统的经典书籍，在阅读本书的英文版时，我们多次拍案叫绝，尤其是被作者在硬件、软件和应用领域的广博知识所折服。本书作者 Raj Kamal 教授在无线电子、微处理器、微控制器、计算机组成和体系结构以及嵌入式系统和计算机网络等领域有 30 多年的理论和实验教学经验。与其他书籍相比，本书面向的是嵌入式系统开发的初中级设计人员，是一本深入浅出的嵌入式系统教程。我想这也是作者的初衷。本书的特点在于，它不仅全面介绍了嵌入式系统的基础知识，而且通过大量详尽的示例帮助读者理解理论知识。当然，这些真实案例对系统设计人员来说具有非常好的参考价值。如本书第 11 章中的“巧克力自动售卖机”、“网络传输”、“汽车自适应行驶系统”、“智能卡”等完整案例，对于未接触过嵌入式系统开发，但想要尽快熟悉开发过程的设计人员提供了很好的示范和模板。更难得的是，本书并没有拘泥于介绍传统的嵌入式系统原理和设计方法，嵌入式系统领域的新兴技术和产品同样得到关注。

本书由国防科技大学计算机学院陈曙晖、高树静、朱丹和许艳蕊翻译完成，全书由陈曙晖统稿。

虽然我们在翻译过程中力求通俗准确，但限于译者的水平，难免存在错误和不足之处，恳请读者批评指正。

前　言

儿童玩耍的智能视频游戏和商店里的巧克力自动售卖机都需要用到嵌入式系统，年轻人从父母那里借来看电影的智能卡(smart card)也需要嵌入式系统；家庭主妇所使用的许多兼容 Internet 的智能化家庭用品(如微波炉、视听系统等)都需要嵌入式系统；驾驶员需要嵌入式系统实现汽车的自动巡航控制。各个单位和机构需要嵌入式系统用于网络系统和产品。嵌入式系统的应用不胜枚举。

有 3 本著作不仅对我产生了巨大的影响，并且还加强了我对嵌入式系统的深入理解，引发了我对基于微处理器和微控制器的嵌入式系统的浓厚兴趣。首先是由 J.B Peatman 编写，MaGraw-Hill 出版公司在 1998 年出版的 *Design with Microcontrollers and Microcomputers*。其次是由 Kenneth Hintz 和 Daniel Tabak 编写，MaGraw-Hill 出版公司在 1992 年出版的 *Microcontrollers Architechture, Implementation and Programming: HD44795, MC68HC11, MCS-51, 80960CA*。第三本是由 Daniel Tabak 编写，MaGraw-Hill 出版公司在 1995 年出版的 *Advanced Microprocessors*。

过去，嵌入式系统是采用微处理器(如 8085)设计的。其应用比较简单，例如温度监控系统、使用 ADC 和 DAC 的数据采集系统、使用适当接口的音乐系统、使用步进电动机接口的简单机器人系统。现在，这些系统有时候甚至不被视为嵌入式系统。

从 20 世纪 80 年代初开始，小型嵌入式系统使用通用电器公司(General Instrument Corporation)生产的微控制器，使用该公司 70 年代末的微控制器 PIC 16xxx、Motorola 微控制器 68HC05/08 以及 8031 系列的 Intel 微控制器。应用广泛的小型嵌入式系统还包括遥控电视、手表、洗衣机、烤箱、计算器、数字日记和视频游戏。80 年代末，Intel 8051/52、Motorola 68HC11/12、Intel 80196 和 80960 系列微控制器的出现，使嵌入式系统硬件的使用更加多元化。

在最近几年，出现了将低级和高级处理硬件单元和专用处理器嵌入到一个芯片中的技术，出现了多处理器系统的嵌入式系统、单芯片 VLSI(称为片上系统)，这些系统具有智能化功能，且高度复杂。一个简单的示例就是智能卡，以及最近出现的典型复杂系统示例——嵌入式系统智能照相机，该系统由 Princeton 大学嵌入式系统小组开发，并由 Wayne Wolf 和他的团队于 2002 年 9 月在 IEEE Computer Society 出版的 *IEEE Micro* 上发布。

根据时间和理解的不同，嵌入式系统的定义也不同。嵌入式系统可以被定义为：将嵌入了软件的计算机硬件作为其重要组成部分的系统。嵌入式系统是专用于各类实际应用或者产品且基于计算机的一种系统。它解决了系统中各种任务响应时间受限的问题。一个嵌入式系统可以是一个独立的系统，也可以是较大系统的一部分。它的软件通常嵌入在 ROM(只读存储器)中。因此它不像计算机一样需要辅助存储器。

本书兼顾了工科研究生、初级读者和热心学者的需要，他们今天是处于萌芽期的嵌入式系统工程师，明天就有可能成为这些系统的主设计师。此外，本书也适用于对嵌入式软件和实时编程项目感兴趣的年轻软件工程师。本书旨在解释设计高性能、响应时间受限制的复杂嵌入式系统所需要的概念，可作为高等院校师生的教材，或者工程师的参考书。

希望本书的读者首先学习嵌入式系统体系结构、在系统开发过程中要使用的基本硬件和软件元素、编程模型和软件工程实践，然后学习将代码嵌入到系统中的软件技术。还希望读者能够设计出充分利用可用系统资源(处理器、存储器、端口、设备和电能)的系统。本书的写作宗旨也正在于此。

本书都包括哪些主题呢？包括设计嵌入式系统的新技术和工具，具体包括哪些简单的和复杂的技术和工具？本书将帮助读者很容易地理解这些主题。其次，学生必须在他们选择的领域里开发出有用的项目。例如：网络，通信，汽车电子、数据采集和存储、服务、处理及保护信息，智能机器人，实时控制和跟踪系统，生物医学系统，声音、图像和视频的实时处理、过滤、压缩和加密系统。

本书各章主题如下：

第 1 章

本章将详细介绍嵌入式系统的基础内容。嵌入式系统硬件包括处理器、存储器设备、I/O 设备和基本硬件单元——电源、时钟和复位电路、访问外设的 I/O 端口和其他片上和片外单元。物理设备有 UART、调制解调器、收发器、时钟计数器、小键盘、键盘、LED 显示单元、LCD 显示单元、DAC 和 ADC 以及脉冲拨号电路等。该章将介绍这些硬件单元、嵌入式软件、最新的嵌入式系统和 RTOS，还将提供一些应用示例。

第 2 章

本章将通过处理器和存储器的组织结构介绍嵌入式系统体系结构。读者将学习嵌入式系统中提供处理功能的处理器结构单元，还将学习到存储器设备。第 2 章中还将介绍一种给定嵌入式系统处理器和存储器的选择方法，并解释将存储器块和段分配给数据结构所依赖的基础知识、存储器映射概念和 DMA 概念，以及存储器、设备、IO 设备和处理器如何进行接口。

第 3 章

本章将描述各种设备(并口和串口设备、时钟设备，同步、等价同步和异步通信设备)以及连接这些设备的重要总线，还将描述设备端口的复杂接口特性。该章还将描述 I^2C、CAN、USB、高级串行高速总线、ISA、PCI、PCIX、高级并行高速总线。

第 4 章

本章将集中讨论设备驱动程序。它们是嵌入式系统中重要的服务例程。此外还将阐述设备驱动程序和网络函数。设备驱动程序是结合示例描述的。对中断服务和处理器机制的理解是嵌入式系统设计者应该具有的基本知识。第 4 章将集中讨论这些问题，详细描述中断延迟和最终期限的概念。这个概念对于嵌入式系统的实时编程很有用。

第 5 章

本章将描述用嵌入式 C/C++/Java 语言进行嵌入式系统编程的编程和源代码管理工具。还将介绍的重要概念有：循环中多函数调用的使用；函数指针、函数队列的使用；中断服务例程(还有设备驱动程序)的排队；数据结构：队列、堆栈和链表。第 5 章还将介绍使用 C++和 Java 语言的面向对象编程概念。在嵌入式系统中，存储器优化是很重要的。那么如何进行优化呢？本章将解答所有这些问题。

第 6 章

本章将描述单处理器和多处理器系统软件开发过程中的程序建模概念，还将描述数据流和控制数据流图的使用；实时编程过程中的程序模型和 FSM 及 Petri 网的使用。本章还将回答一些重要的问题：如何对微处理器建模，以及如何调度和同步处理器指令的执行。

第 7 章

本章将介绍嵌入式系统开发过程中的软件工程实践和方法，描述线性序列模型、RAD(Rapid Development，快速开发)模型和其他重要模型，包括基于组件的(面向对象的)软件开发过程模型。本章还将介绍软件需求分析、设计、实现、测试、调试和验证策略，并讲述一种重要的设计语言——UML 语言。

第 8 章

本章将要介绍实时编程最重要的内容：进程间通信。首先将介绍进程、任务和线程的概念，其次将描述信号量的使用。第 8 章还将详细介绍信号、互斥、消息队列、邮箱、管道、虚拟(逻辑)插槽和远程调用的概念。

第 9 章

本章将阐述 RTOS 的概念，首先介绍 OS 结构和内核功能，然后介绍进程、存储器、设备、文件和 IP 子系统管理功能，最后介绍 RTOS 对多任务的调度管理。在此基础上，进一步解释周期、循环、抢占式、时间片，以及其他调度模型如何实时调度多任务，同时还将描述 RTOS IEEE 标准。本章重点介绍对多进程之间进行同步的 15 点策略。

第 10 章

本章将结合示例，详细介绍两种最重要的 RTOS 工具——μC/OS-II 和 VxWorks。

第 11 章

本章将描述关于 RTOS 编程的 4 个案例研究。分别是巧克力自动售卖机系统、TCP/IP 网络系统、汽车中的自适应巡航控制系统和智能卡。

第 12 章

本章将描述硬件和软件设计以及集成方法和工具。第 12 章将解释嵌入式系统开发过程行为计划，还将描述目标系统、仿真器、ICE 的使用；用于将最终代码下载到 ROM 中的设备编程器的使用；代码生成工具(汇编器、编译器、加载器和链接器)、模拟器、原型开发工具和 IDE 的使用，以外还将介绍硬件测试工具。

附录

附录将简要介绍 CISC 和 RISC 处理器体系结构的要点、寻址方式和指令集、嵌入式高性能处理器、ARM7、ARM9、ARM11 和 IBM PowerPC 750，还将概述微处理器体系结构。在图像、视频和收敛技术产品的系统中需要使用 DSP。媒体(数据、声音和视频)处理器是用于实时视频、流网络和数据网络的新型处理器。附录也将对它们进行介绍。我们还将简要介绍用于连接嵌入式系统中分散的设备和设备硬件单元的串行和并行总线。嵌入式系统主题将在本科生和研究生课程中介绍。为了指导课程设计人员和教师，附录还给出了这些课程中建议使用的单元。

本书还给出了上百种参考书、网站和期刊杂志。这些将帮助读者对嵌入式系统相关主题进行进一步的研究。

本书 7 个最突出的特点是：

(1) 结构合理、内容系统、主题安排逻辑性强。

(2) 对嵌入式系统编程概念、OS、RTOS 函数和进程间同步进行了详细介绍。

(3) 特别提到了在单芯片或者多芯片系统的软件开发过程中程序的建模，以及软件工程实践的使用。

(4) 详细介绍了端口、设备、用于设备互连的总线和设备驱动程序。

(5) 在消费类电子、通信和汽车电子以及安全事务片上系统中 RTOS 编程的创新案例研究。

(6) 同时介绍了两种 RTOS——μC/OS-II 和 VxWorks，主要集中在 RTOS 函数应用方面。

(7) 表达明晰，突出强调示例，并具有良好设计的图片和表格、关键字及其定义，每章最后还包括了问题回顾和实践。

本书尽可能地给出正确的信息和示例代码以及工具。然而，错误在所难免。敬请广大读者批评指正。

欢迎教师、学者、软件工程师和系统工程师提出宝贵意见，以进一步提高本书的质量。请将建议和问题发往我的电子信箱中，邮件地址为 professional@rajkamal.org，也可以访问网站 http://www.rajkamal.org。

Raj Kamal

目　录

第 1 章　嵌入式系统简介 …… 1
1.1　嵌入式系统 …… 2
1.1.1　系统 …… 2
1.1.2　嵌入式系统 …… 3
1.1.3　嵌入式系统的分类 …… 4
1.1.4　嵌入式系统设计者需要具备的技能 …… 5
1.2　系统中的处理器 …… 6
1.2.1　系统中的处理器 …… 6
1.2.2　微处理器 …… 7
1.2.3　微控制器 …… 8
1.2.4　复杂系统的嵌入式处理器 …… 10
1.2.5　数字信号处理器 …… 11
1.2.6　嵌入式系统的专用系统处理器 …… 11
1.2.7　使用通用处理器的多处理器系统 …… 12
1.3　其他硬件单元 …… 13
1.3.1　电源和低功耗管理 …… 13
1.3.2　时钟振荡电路和时钟单元 …… 15
1.3.3　系统需要的各种计时和计数功能的实时时钟和定时器 …… 16
1.3.4　复位电路、加电复位和 Watchdog 定时器复位 …… 16
1.3.5　存储器 …… 17
1.3.6　输入、输出和 I/O 端口，IO 总线和 IO 接口 …… 18
1.3.7　中断处理器 …… 19
1.3.8　DAC(使用 PWM)和 ADC …… 19
1.3.9　LCD 和 LED 显示 …… 20
1.3.10　小键盘/键盘 …… 21
1.3.11　脉冲拨号电路、调制解调器和收发器 …… 21
1.3.12　GPIB(IEEE 488)连接 …… 21
1.3.13　嵌入式系统硬件的连接和接口总线及单元 …… 22
1.3.14　案例中所需要的硬件单元 …… 22
1.4　嵌入系统软件 …… 24
1.4.1　产品的最终机器可实现软件 …… 24
1.4.2　用机器码编写软件 …… 25
1.4.3　用特定于处理器的汇编语言编写软件 …… 26
1.4.4　用高级语言编写软件 …… 27
1.4.5　使用操作系统的设备驱动程序和设备管理软件 …… 29

1.4.6 多任务调度和使用 RTOS 设备的软件设计……30
1.4.7 设计嵌入式系统的软件工具……30
1.4.8 示例中需要的软件工具……32
1.4.9 软件设计模型……32
1.5 示例嵌入式系统……33
1.6 嵌入式片上系统(SOC)和内部 VLSI 电路……35
1.6.1 用于便携式电话的 SoC 示例……35
1.6.2 ASIP……36
1.6.3 IP 核……36
1.6.4 嵌入 GPP……37
1.6.5 具有一个或者多个处理器的 FPGA 核……37
1.6.6 示例 SoC 中的组成部分——智能卡……38

第 2 章 处理器和存储器组织……45
2.1 处理器中的结构单元……46
2.2 嵌入式系统的处理器选择……55
2.3 存储器设备……58
2.3.1 ROM：使用方法、形式和变种……59
2.3.2 RAM 设备……60
2.4 嵌入式系统的存储器选择……61
2.5 程序段和块的存储器分配及系统的存储器映射……64
2.5.1 各种存储器段中的函数、过程、数据和堆栈……64
2.5.2 不同数据结构和数据集合元素的存储器块……66
2.5.3 存储器映射……71
2.5.4 内部设备和 I/O 设备在映射中的地址……77
2.6 直接存储器访问……79
2.7 处理器、存储器和 I/O 设备的接口……80

第 3 章 设备网络的设备和总线……87
3.1 I/O 设备……88
3.1.1 I/O 设备的类型和示例……88
3.1.2 串行设备的同步、准同步和异步通信……90
3.1.3 内部串行通信设备的示例……93
3.1.4 并口设备……95
3.1.5 设备端口的复杂接口特性……97
3.2 定时器和计数设备……98
3.3 互连的多个设备之间通过 I^2C、CAN 和高级 I/O 总线进行串行通信……102
3.3.1 I^2C 总线……102
3.3.2 CAN 总线……103
3.3.3 USB 总线……105
3.3.4 先进的串行高速总线……105

3.4 多个互连 I/O 设备之间通过 ISA、PCI、PCI-X 和高级总线进行的计算机或者主机系统并行通信 …… 106
3.4.1 ISA 总线 …… 106
3.4.2 PCI 和 PCI/X 总线 …… 107
3.4.3 高级并行高速总线 …… 108

第 4 章 设备驱动程序和中断服务机制 …… 115
4.1 设备驱动程序 …… 117
4.1.1 不使用 ISR 的设备服务 …… 117
4.1.2 设备驱动程序 ISR …… 118
4.1.3 设备驱动程序 …… 120
4.1.4 作为设备驱动和网络函数的 Linux 内幕 …… 121
4.1.5 编写系统中的物理设备驱动 ISR …… 122
4.1.6 虚拟设备 …… 123
4.2 系统中的并口设备驱动程序 …… 124
4.3 系统中的串口设备驱动程序 …… 130
4.4 内部可编程定时设备的设备驱动程序 …… 134
4.5 中断服务(处理)机制 …… 135
4.5.1 硬件和软件相关的中断源 …… 135
4.5.2 软件错误相关的硬件中断 …… 135
4.5.3 软件指令相关的中断源 …… 137
4.5.4 内部设备相关的硬件中断 …… 138
4.5.5 中断向量 …… 138
4.5.6 根据可屏蔽和不可屏蔽的中断分类 …… 138
4.5.7 所有可屏蔽中断源的激活(未屏蔽)和禁用(屏蔽) …… 139
4.5.8 中断挂起寄存器或者状态寄存器 …… 139
4.6 上下文和上下文切换周期、最终期限和中断延迟 …… 139
4.6.1 上下文、延迟和最终期限 …… 139
4.6.2 从上下文保存的角度对处理器中断服务机制的分类 …… 143
4.6.3 使用 DMA 通道帮助缩短中断延迟周期 …… 144
4.6.4 满足服务最终期限的优先级分配 …… 144
4.6.5 硬件优先级的软件覆盖 …… 145

第 5 章 编程概念及 C 与 C++的嵌入式编程 …… 150
5.1 用汇编语言和高级语言 C 进行软件编程 …… 151
5.2 C 程序中的元素：头文件、源文件以及预处理指令 …… 152
5.2.1 用于包含文件的 include 指令 …… 152
5.2.2 源文件 …… 154
5.2.3 配置文件 …… 154
5.2.4 预处理指令 …… 154
5.3 程序元素：宏与函数 …… 154

5.4 程序元素：数据类型、数据结构、修饰符、语句、循环和指针 …… 156
5.4.1 数据类型 …… 156
5.4.2 使用数据结构：队列、堆栈、链表和树 …… 156
5.4.3 修饰符 …… 158
5.4.4 条件语句、循环语句以及无限循环语句 …… 159
5.4.5 指针和 NULL 指针 …… 161
5.4.6 函数调用 …… 163
5.4.7 主程序中按照循环顺序进行的多函数调用 …… 164
5.4.8 函数指针、函数队列和中断服务例程队列 …… 166
5.5 队列 …… 167
5.5.1 队列 …… 167
5.5.2 实现网络协议的队列 …… 170
5.5.3 发生中断时函数的排列 …… 172
5.5.4 网络中进行流控制的 FIPO 队列 …… 174
5.6 堆栈 …… 175
5.7 链表与有序链表 …… 178
5.7.1 链表 …… 178
5.7.2 活动设备驱动器(软件时钟)的链表 …… 186
5.7.3 就绪链表中的任务链表 …… 187
5.8 C++嵌入式编程 …… 189
5.8.1 面向对象的编程 …… 189
5.8.2 C++的嵌入式编程 …… 189
5.9 用 Java 进行嵌入式编程 …… 191
5.9.1 什么时候用 Java 编程 …… 191
5.9.2 Java 的缺点 …… 191
5.10 C 程序编译器与交叉编译器 …… 193
5.11 嵌入式 C/C++的源代码工程管理工具 …… 194
5.12 存储器需求的优化 …… 194

第 6 章 单处理器和多处理器系统软件开发过程中的程序建模概念 …… 201
6.1 软件实现之前对软件分析过程的建模 …… 202
6.1.1 数据流图在程序分析中的用法 …… 202
6.1.2 用于程序分析的控制数据流图的用法 …… 204
6.2 用于事件控制或者响应时间受到约束的实时程序的编程模型 …… 205
6.2.1 有限状态机模型 …… 205
6.2.2 Petri 网模型 …… 209
6.3 多处理器系统的建模 …… 215
6.3.1 多处理器系统中的问题 …… 216
6.3.2 模型 …… 217
6.3.3 同步数据流图模型 …… 218

6.3.4 同构的同步数据流图模型 …… 219
6.3.5 无环优先扩展图模型 …… 220
6.3.6 定时的 Petri 网和扩展预测/转换网模型 …… 221
6.3.7 多线程图系统模型 …… 222
6.3.8 图和 Petri 网在多处理器系统中的应用 …… 223

第 7 章 嵌入式软件开发过程中的软件工程实践 …… 233
7.1 软件的算法复杂度 …… 234
7.2 软件开发生命周期及其模型 …… 236
7.2.1 软件开发过程中的线性顺序模型(瀑布模型或者生命周期模型) …… 236
7.2.2 RAD 模型 …… 237
7.2.3 增量模型 …… 237
7.2.4 并发模型 …… 238
7.2.5 基于组件(面向对象)的软件开发过程模型 …… 238
7.2.6 基于第四代工具的软件开发过程模型 …… 238
7.2.7 基于面向对象和基于第四代工具的方法 …… 239
7.3 软件分析 …… 239
7.4 软件设计 …… 241
7.5 软件实现 …… 242
7.6 软件测试、确认以及调试 …… 243
7.6.1 测试、验证以及确认 …… 243
7.6.2 调试 …… 246
7.7 软件开发过程中的实时程序设计问题 …… 248
7.7.1 在需求和规范的分析中存在的问题 …… 248
7.7.2 设计和实现中存在的问题 …… 249
7.7.3 系统集成中的问题 …… 249
7.7.4 测试中的问题 …… 249
7.8 软件项目管理 …… 250
7.8.1 项目管理 …… 250
7.8.2 项目测度 …… 251
7.9 软件维护 …… 253
7.10 统一建模语言(UML) …… 255

第 8 章 进程间通信与进程、任务和线程的同步 …… 265
8.1 应用程序中的多个进程 …… 266
8.1.1 进程 …… 266
8.1.2 任务 …… 266
8.1.3 线程 …… 268
8.1.4 通过函数、ISR 和任务的特征进行区分 …… 269
8.2 多任务和多例程的数据共享问题 …… 270
8.2.1 数据共享问题及其解决方案 …… 270

8.2.2 对任务或者任务的临界段使用信号量 …… 271
8.2.3 优先级倒置问题和死锁情况 …… 279
8.3 进程间通信 …… 280
8.3.1 信号 …… 281
8.3.2 信号量标识或者互斥体用作资源键(用于进程的资源加锁和解锁) …… 281
8.3.3 消息队列 …… 282
8.3.4 邮箱 …… 284
8.3.5 管道 …… 285
8.3.6 虚拟(逻辑)套接字 …… 287
8.3.7 远程过程调用(RPC) …… 287

第 9 章 实时操作系统 …… 292
9.1 操作系统服务 …… 293
9.1.1 目标 …… 293
9.1.2 结构 …… 294
9.1.3 内核 …… 295
9.1.4 进程管理 …… 296
9.1.5 存储器管理 …… 297
9.1.6 设备管理 …… 298
9.1.7 文件系统的组织和实现 …… 300
9.2 I/O 子系统 …… 302
9.3 网络操作系统 …… 302
9.4 实时操作系统与嵌入式操作系统 …… 304
9.4.1 实时操作系统 …… 304
9.4.2 在嵌入式系统中何时需要 RTOS …… 304
9.4.3 RTOS 的多任务调度管理 …… 306
9.4.4 实时系统中通过 RTOS 进行的多任务调度 …… 308
9.5 RTOS 环境中的中断例程：RTOS 的中断源调用处理 …… 309
9.5.1 通过中断源直接调用 ISR …… 310
9.5.2 通过中断源以及调度任务的暂时挂起，直接调用 RTOS …… 311
9.5.3 通过中断源以及 RTOS 对任务和 ISR 的调度，直接调用 RTOS …… 311
9.6 RTOS 任务调度模型，作为性能测度的中断延迟和任务响应时间 …… 311
9.6.1 使用就绪任务循环队列的协作轮转调度 …… 312
9.6.2 使用按照优先级约束排序列表的就绪任务的协作调度 …… 313
9.6.3 时间分片的循环调度(速率单调的协作调度) …… 315
9.6.4 调度程序控制的抢占式调度模型策略 …… 316
9.6.5 抢占式调度程序提供的临界段服务 …… 318
9.6.6 任务的固定(静态)实时调度 …… 320
9.6.7 调度算法中的优先级分配 …… 320
9.6.8 使用概率定时 Petri 网(随机)和多线程图(MTG)的高级调度算法 …… 321

9.7 周期、零散以及非周期任务的调度模型的性能测度 …… 321
9.7.1 使用 CPU 负载作为性能尺度 …… 321
9.7.2 零散任务模型 …… 322
9.8 为 RTOS 的标准化和任务内部通信函数采用的 IEEE 标准 POSIX 1003.1B …… 322
9.9 抢占式调度程序的基本操作及其在处理器上预期耗费的时间 …… 324
9.10 用于进程间、ISR 间、OS 函数间和任务之间同步及资源管理的 15 条策略 …… 325
9.11 嵌入式 LINUX 的内部组织：设备驱动程序和嵌入式系统的 LINUX 内核 …… 326
9.12 操作系统的安全问题 …… 329
9.13 移动式操作系统 …… 330

第 10 章 实时操作系统编程工具：MicroC/OS-II 和 VxWorks …… 334
10.1 测试稳定且调试合格的实时操作系统的必要性 …… 335
10.2 μC/OS-II …… 337
10.2.1 RTOS 系统级函数 …… 338
10.2.2 任务服务函数及其使用范例 …… 341
10.2.3 时间延迟函数 …… 347
10.2.4 函数相关的存储器分配 …… 349
10.2.5 信号量相关函数 …… 351
10.2.6 邮箱相关函数 …… 363
10.2.7 队列相关函数 …… 372
10.3 VxWorks …… 381
10.3.1 基本特性 …… 382
10.3.2 系统库头文件中的任务管理库 …… 384
10.3.3 VxWorks 系统函数和系统任务 …… 388
10.3.4 进程(任务)间通信函数 …… 391

第 11 章 RTOS 编程案例研究 …… 405
11.1 使用 MUCOS RTOS 对巧克力自动售卖机编码 …… 406
11.1.1 案例定义、多任务及其函数 …… 406
11.1.2 创建任务、函数和 IPC …… 408
11.1.3 编码步骤示例 …… 411
11.2 使用 RTOS VxWorks 将应用层字节流发送到 TCP/IP 网络上 …… 420
11.2.1 案例定义、多任务及其函数 …… 421
11.2.2 创建任务、函数和 IPC …… 424
11.2.3 编码步骤示例 …… 425
11.3 汽车自适应巡航控制系统的嵌入式系统 …… 443
11.4 智能卡中的嵌入式系统 …… 456
11.4.1 嵌入式硬件 …… 457
11.4.2 嵌入式软件 …… 458

第 12 章 嵌入式系统中的软硬件协同设计 …… 472
12.1 嵌入式系统项目管理 …… 473
12.2 系统开发过程中嵌入式系统设计和协同设计问题 …… 474
12.2.1 嵌入式系统开发过程的目标 …… 474
12.2.2 行动计划 …… 474
12.2.3 完整的规范和系统需求 …… 475
12.2.4 原理设计 …… 476
12.2.5 软硬件布局设计 …… 477
12.2.6 详细设计 …… 478
12.2.7 实现工具 …… 478
12.2.8 测试 …… 479
12.3 嵌入式系统开发阶段中的设计周期 …… 479
12.4 目标系统或其仿真器和内置电路仿真器(ICE) …… 480
12.4.1 使用目标系统 …… 480
12.4.2 仿真器和 ICE …… 481
12.4.3 用于将最终代码下载到 ROM 中的设备编程器 …… 483
12.5 嵌入式系统开发中的软件工具 …… 483
12.5.1 代码生成工具(汇编器、编译器、加载器和链接器) …… 483
12.5.2 模拟器 …… 484
12.5.3 嵌入式系统的原型开发、测试和调试工具 …… 486
12.5.4 集成开发环境 …… 487
12.5.5 存储器、处理器敏感程序和设备驱动程序 …… 489
12.5.6 动态链接库 …… 489
12.6 示波器和逻辑分析仪在系统硬件测试中的使用 …… 490
12.6.1 逻辑探测器或者 LED 测试 …… 490
12.6.2 示波器 …… 490
12.6.3 逻辑分析仪 …… 491
12.6.4 位率测量仪 …… 491
12.6.5 用于 ROM 调试的系统监控代码 …… 492
12.7 嵌入式系统设计中的问题 …… 492
12.7.1 选择合适的平台 …… 492
12.7.2 嵌入式系统处理器的选择 …… 493
12.7.3 需要考虑的因素和必需的特性 …… 494
12.7.4 软硬件权衡 …… 495
12.7.5 性能建模 …… 495
12.7.6 性能加速器 …… 496
12.7.7 嵌入式系统中 OS 的移植问题 …… 496

附录 A CISC 和 RISC 处理器体系结 构和指令集示例 …… 504
A.1 CISC 和 RISC 处理器的指令及其处理 …… 504

A.1.1 指令和数据的格式 …… 504
A.1.2 寻址模式 …… 505
A.1.3 指令集 …… 506
A.1.4 CISC 和 RISC 体系结构 …… 507
A.2 指令集示例—— ARM7 …… 508
A.3 ARM 处理器的汇编语言程序示例 …… 510

附录 B 嵌入式系统高性能处理器 …… 511
B.1 ARM 处理器示例 …… 511
B.2 高性能处理器示例 …… 514
B.3 加速器 …… 515

附录 C 嵌入式系统 8/16/32 位微处理器及其体系结构概述 …… 517
C.1 Intel、Motorola 和 PIC 系列微控制器的体系结构概述 …… 517
C.2 Motorola 系列 CISC 和 RISC 的新一代微控制器 …… 519

附录 D 嵌入式数字信号处理器 …… 522
D.1 数字信号处理器的体系结构 …… 522
D.2 DSP 处理器和传统处理器的比较 …… 524
D.3 定点运算和浮点运算的比较 …… 525
D.4 嵌入式系统的 DSP …… 525
D.4.1 TMS320C2000™ 平台 …… 526
D.4.2 TMS320C5000™ 平台 …… 526
D.4.3 TMS320C6000™ 平台 …… 526
D.4.4 DSP 的 TMS320C24x 和 C28X 代产品 …… 527
D.4.5 TMS320C54x 和 TMS320C55x 代 DSP …… 528
D.4.6 TMS320C62x、64x 和 C67x 代 DSP …… 529
D.4.7 RISC 环境下的 OMAP5910 嵌入式处理器 DSP …… 530
D.4.8 基于 SoC 的解决方案 Texas DSP TMS320DM310 …… 530

附录 E 嵌入式系统应用的新型处理器 …… 532

附录 F 串行和并行总线 …… 537
F.1 新的串行总线标准(USB 2.0，IEEE 1394) …… 537
F.2 新的并行总线标准(Compact PCI、PCI-X) …… 537

附录 G 嵌入式系统中的设备 …… 539
G.1 各种形式的 ROM 设备 …… 539
G.2 ROM 设备编程器 …… 540
G.2.1 二进制映像 …… 540
G.2.2 Motorola S-Record 格式 …… 540
G.2.3 Intel Hex-File 格式 …… 541
G.2.4 设备编程器的编程方法 …… 542

G.3 RAM 设备 …… 544
G.3.1 静态和动态 RAM …… 544
G.3.2 EDO RAM …… 544
G.3.3 SDRAM …… 545
G.3.4 RDRAM …… 545
G.3.5 参数化的分布式 RAM …… 545
G.3.6 参数化的块 RAM …… 545
G.4 微控制器中的并口 …… 545
G.5 串行通信设备 …… 546
G.5.1 Motorola 68HC11 中的 SPI 和 SCI …… 546
G.5.2 微控制器中的串行通信设备 …… 547
G.6 微处理器中的定时器 …… 548
G.7 各种处理器系列中的中断源及其控制 …… 550
G.8 80x86 处理器的中断 …… 552
G.9 68HC11 中的中断 …… 552
G.9.1 中断服务 …… 552
G.9.2 中断源 …… 553

附录 H 嵌入式系统体系结构、编程和设计中的重要内容 …… 554
H.1 推荐使用的教学大纲 …… 554
H.2 CDAC 嵌入式系统课程教学大纲涉及的内容 …… 556

第 1 章　嵌入式系统简介

本章学习内容

在本章中，我们将学习下面的内容：

1.1 节 嵌入式系统

(1) 系统(system)以及嵌入式系统(embedded system)的定义。

(2) 嵌入式系统的一种类型。

(3) 设计嵌入式系统所需要的技能。

1.2 节 嵌入式系统的处理单元

(1) 嵌入式系统的处理器。在嵌入式系统中，处理器是一个很重要的单元。微控制器(microcontroller)是一种包含处理器、存储器以及其他许多硬件单元的集成芯片，它们共同组成了嵌入式系统的微计算机部分。嵌入式处理器(embedded processor)是一种具有某些特性的处理器，这些特性使之能够被嵌入到系统中。数字信号处理器(digital signal processor，DSP)是一种用于处理数字信号的处理器(例如，滤波、回音消除、噪声消除、压缩以及加密)。

(2) 在小型、中型、大型嵌入式系统中普遍使用的微处理器、微控制器以及 DSP。

(3) 最近出现的一种新兴技术，将专用系统处理器(application-specific system processor，ASSP)合并到嵌入式系统中。

(4) 多处理器系统。

1.3 节 其他硬件单元

(1) 嵌入式系统的电源及其对控制功耗的需求。

(2) 嵌入式系统的时钟振荡器电路以及时钟单元。在它的作用下，处理器运行并处理指令。

(3) 实时时钟(real time clock，RTC)、定时器以及系统需要的各种定时功能。

(4) 复位电路(reset circuit)以及 watchdog 定时器。

(5) 系统存储器(system memory)(在第 2 章的第 2 部分，我们将学习有关系统存储器的详细内容)。

(6) 系统输入/输出(IO)端口，串行通用异步收发器(UART)以及其他端口，如多路器、信号分离器和接口总线(这些将在第 3 章中详细介绍)。

(7) 中断处理器(第 4 章的后一部分将介绍有关的详细内容)。

(8) 接口单元——使用 PWM(Pulse Width Modulation，脉宽调制)的 DAC(Digital to Analog Converter，数模转换器)、ADC(Analog to Digital Converter，模数转换器)、LED 和 LCD 显示单元、小键盘及键盘、脉冲拨号电路、调制解调器以及收发器。

(9) 示例嵌入式系统所需要的硬件。

1.4 节 用于开发系统的嵌入式软件(embedded software)的语言的不同层次(在第 5 章中将详细介绍高级编程的有关知识、C 和 C++的语言结构以及如何用它们编写嵌入式软件)。

(1) 使用操作系统(OS)和实时操作系统(real time operating system，RTOS)的系统设备驱动程序、设备管理和多任务处理(RTOS 将在第 9 章和第 10 章中详细介绍，在第 11 章中将介绍使用 RTOS 的案例)。

(2) 系统设计中使用的软件工具。

(3) 本书 6 个案例中需要的软件工具。

(4) 软件设计的编程模型(软件设计模型将在第 6 章中详细介绍)。

1.5 节 每种类型的嵌入式系统应用示例

1.6 节 在 VLSI 芯片上设计嵌入式系统。

(1) 嵌入式 SoC(System on Chip，片上系统)和 ASIC(Application Specific Integrated Circuit，专用集成电路)以及它们的应用示例。在 ASIC 芯片中使用(i)专用指令处理器(Application Specific Instruction Processor，ASIP)，(ii)知识产权(Intellectual Property，IP)核，(iii)具有单个或多个处理器单元的现场可编程门阵列(Field Programmable Gate Arrays，FPGA)核。

(2) 智能卡(smart card)是片上(SoC)嵌入式系统单元的一个例子。

1.1 嵌入式系统

1.1.1 系统

系统是一种根据固定的计划、程序或者规则进行工作、组织或者执行一项或多项任务的方式。系统也是一种安排方法，其所有单元按照一定的计划或者程序装配在一起，并共同工作。下面让我们来看两个示例。

手表是一个时间显示系统(time-display system)。其零件包括硬件、表针和电池，以及漂亮的表盘、底盘和表链。这些零件组织在一起显示每一秒的实时时间，并且每一秒都连续更新时间显示。每过一秒钟，其系统程序就使用三个表针来更新显示。它遵守一系列的规则。其中一些规则如下所述：(i)所有的表针都顺时针转动。(ii)细长的表针每过一秒做一次旋转运动，1 分钟后回到原位。(iii)长表针每过一分钟做一次旋转运动，1 小时后回到原位。(iv)短表针每过一小时做一次旋转运动，12 小时后回到原位。(v)每天经过 12 小时后，所有的表针都回归原位。

洗衣机是一个自动洗涤系统(automatic clothes-washing system)。重要的硬件零件包括状态显示板、用于用户定义的开关和转盘、用于旋转的电动机、电源以及控制单元、内部水量传感器、进水螺线管阀门以及出水螺线管阀门。这些零件装配在一起，按照用户给定的程序自动洗涤。系统程序就是清洗放置在桶中的脏衣服，按照事先编程的步骤和阶段进行旋转。它也遵守一系列的规则。其中一些规则如下所述：(i)严格按照下面的步骤执行。第一步：按照一个事先编程的周期旋转电动机，清洗衣物。第二步：将脏水排出后，再注入新水，如果系统没有设置为节水模式，则可以进行第二次注水。第三步：将水完全排出后，按照事先编程的周期快速旋转电动机，通过将水从衣物中甩出，拧干衣物。第四步：通过闪烁显示，提示用户当前洗涤结束。报警一分钟发出信号，表示洗涤结束。(ii)在每一个步骤中都显示系统当前所处的处理阶段。(iii)当发生中断时，只执行程序剩下的部分，再次启动时从过程被中断的地方重新开始。一般

不会出现从第一步开始的重复执行，除非用户加进了另外一些衣服重置了系统和程序。

1.1.2 嵌入式系统

计算机是一个具有下列或者更多组成部分的系统。

(1) 微处理器

(2) 大型存储器，包括如下两种：

(a) 主存储器(半导体存储器——RAM、ROM 以及可快速访问的高速缓存)

(b) 辅助存储器(硬盘、磁盘和磁带中的磁性存储器，以及 CD-ROM 中的光存储器)

(3) 输入单元，例如键盘、鼠标、数字转换器、扫描仪等等。

(4) 输出单元，例如显示器、打印机等等。

(5) 网络单元，例如以太网卡、前端基于处理器的驱动程序等等。

(6) I/O 单元，例如调制解调器、附有调制解调器的传真机等等。

嵌入式系统是一种将嵌入了软件的计算机硬件作为其最重要的一部分的系统。它是一种专用于某个应用或者产品的基于计算机的系统。它可以是一个独立的系统，也可以是更大系统的一部分。由于其软件通常嵌入在 ROM(只读存储器)中，因此并不像计算机一样需要辅助存储器。一个嵌入式系统有三个主要组成部分：

(1) 硬件。图 1-1 给出了嵌入式系统硬件中的组成单元。

(2) 主应用软件。应用软件可以并发地执行任务序列或者多任务。

(3) 实时操作系统(RTOS)用来管理应用软件，并提供一种机制，使得处理器在一次进程调度时运行一个进程，并在各个进程(任务)之间进行上下文切换。RTOS 定义了系统工作的方式。它将对资源的访问组织成为系统的任务序列。它按照计划控制延迟(latency)并满足最后期限，从而调度任务的执行(延迟指的是运行一项任务的各段代码之间，以及任务发生所需要的等待周期)。它在执行应用软件的过程中制定规则。小型嵌入式系统可能不需要 RTOS。

嵌入式系统的软件设计受三个条件的限制：(i)可用的系统存储器，(ii)处理器速度，(iii)当以等待事件、运行、停止和唤醒的周期连续运行系统时，对功耗的限制。

有许多书都给出了嵌入式系统的定义。下面给出的是其他文献中对嵌入式系统的定义：

Computers as Components – Principles of Embedded Computing System Design 一书的作者 Wayne Wolf 认为："什么是嵌入式计算系统？如果不严格地定义，它是任何一个包含可编程计算机的设备，但是它本身却不是一个通用计算机"，"使用微处理器构造的传真机或者时钟就是一种嵌入式计算系统"。

Embedded Microcontrollers 一书的作者 Todd D. Morton 认为："嵌入式系统是一种电子系统，它包含微处理器或者微控制器，但是我们不认为它们是计算机——计算机隐藏或者嵌入在系统中。"

Embedded Software Primer 一书的作者 Davie E. Simon 认为："人们使用嵌入式系统这个术语，指的是隐藏在任一产品中的一个计算机系统。"

An Introduction to the Design of Small Scale Embedded System with examples from PIC, 80C51 and 68HC05/08 Microcontrollers 一书的作者 Tim Wilmshurst 认为：(1)"嵌入式系统是这样一个

系统，它的首要功能并不是计算，而是受嵌入其中的计算机控制的一个系统。“嵌入”暗示了它存在于整个系统中，从外部观察不到，形成了更大整体的一个完整部分。”(2)“嵌入式系统是一种基于微控制器、软件驱动的可靠实时控制系统，以自治的、人工的或者网络方式进行交互，对各种物理变量进行操作，存在于各种环境中，在竞争激烈的市场上出售”。

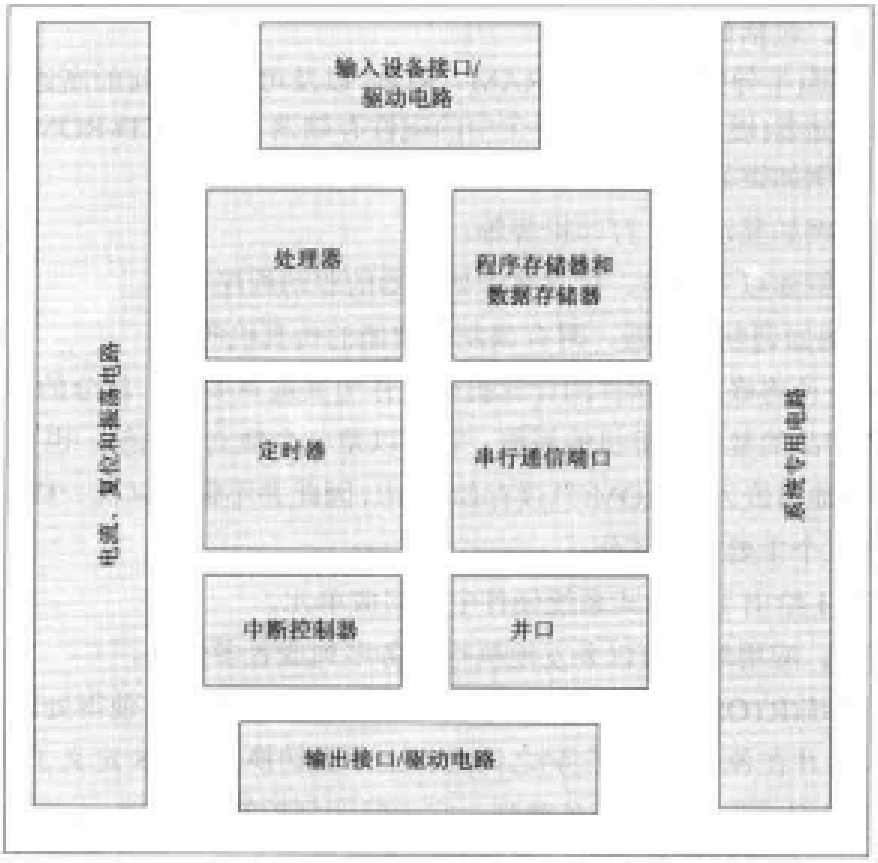


图 1-1 嵌入式系统硬件的组成

1.1.3 嵌入式系统的分类

我们可以将嵌入式系统分成以下三种类型(在第 1.5 节中我们将给出每一种类型的示例)：

(1) 小型嵌入式系统

这些系统是采用一个 8 位或者 16 位的微控制器设计的；硬件和软件复杂度很小，需要进行板级设计。它们甚至可以是电池驱动的。当为这些系统开发嵌入式软件时，主要的编程工具是所使用的微控制器或者处理器专用的编辑器、汇编器(assembler)和交叉汇编器。通常利用 C 语言来开发这些系统。C 程序被编译为汇编程序，然后将可执行代码存放到系统存储器的适当位置上。为了满足系统连续运行时的功耗限制，软件必须放置在存储器中。

(2) 中型嵌入式系统

这些系统是采用一个 16 位或者 32 位的微控制器、DSP 或者精简指令集计算机(RISC)设计的；硬件和软件复杂度都比较大。对于复杂的软件设计，可以使用如下的编程工具：RTOS、源代码设计工具、模拟器、调试器和集成开发环境(IDE)。软件工具还提供了硬件复杂性的解决方法。汇编器作为编程工具来说用处不大。这些系统还可以运用已有的 ASSP 和 IP(后面将会介绍)来完成各种功能，例如，总线接口、加密、解密、离散余弦变换和逆变换、TCP/IP 协

议栈和网络连接功能(ASSP 和 IP 可能还必须用系统软件进行适当的配置，才能集成到系统总线上)。

(3) 复杂嵌入式系统

复杂嵌入式系统的软件和硬件都非常复杂，需要可升级的处理器或者可配置的处理器和可编程逻辑阵列。它们用于边缘应用，在这些应用中，需要硬件和软件协同设计，并且都集成到最终的系统中；然而，它们却受到硬件单元所提供的处理速度的限制。为了节约时间并提高运行速度，可以在硬件中实现一定的软件功能，例如加密和解密算法、离散余弦变换和逆变换算法、TCP/IP 协议栈和网络驱动程序功能。系统中某些硬件资源的功能也可以用软件来实现。这些系统的开发工具要么十分昂贵，要么根本就不存在。有时候，必须为这些系统开发编译器或者可重定目标的编译器(可重定目标的编译器，就是一种可以根据系统中给定的目标配置进行配置的编译器)。

1.1.4 嵌入式系统设计者需要具备的技能

嵌入式系统设计者必须使用已有的工具，并在给定的规范、费用和时间框架内开发产品(在第 7 章和 12 章中将介绍嵌入式系统设计方面的问题。第 1.5 节也有介绍)。

(1) 小型嵌入式系统设计者需要具备的技能

前文介绍的作者 Tim Wilmshurst(参见 1.1.2 节)说过，开发小型嵌入式系统的个人或者团队需要具备下列技能："完全理解微控制器，具备计算机体系结构、数字电子设计、软件工程、数据通信、控制工程、电动机和致动器、传感器和测量、模拟电子设计、IC 设计和生产的基本知识。"特定的情况下需要特定的技能。例如，当设计控制系统时，需要具备控制工程的知识；设计系统接口时，需要具备模拟电子设计的知识。本书将介绍下列主题的基础部分，供那些已经具备了微控制器和微处理器知识的设计者使用。(i)计算机体系结构和组织。(ii)存储器。(iii)存储器分配。(iv)存储器接口。(v)向 PROM 或者 ROM 中刻录(移植所用的术语)可执行机器码(参见 2.3.1 节)。(vi)译码器和信号分离器的使用。(vii)直接存储器访问。(viii)端口。(ix)用汇编语言实现设备驱动程序。(x)简单和复杂的总线。(xi)定时器。(xii)中断服务机制。(xiii)C 编程元素。(xiv)存储器优化。(xv)硬件和微控制器的选择。(xvi)ICE(电路内部仿真器)、交叉汇编器和测试设备的使用。(xvii)使用测试向量调试软件和硬件 bug。其他领域的基础知识，如数据通信、控制工程、电动机和致动器、传感器和测量、模拟电子设计、IC 设计和生产，可以从现有的其他教科书中找到。

对小型嵌入式系统感兴趣的设计者没有必要全面了解中断延迟和最后期限以及它们的处理方式，RTOS 编程工具将在第 9 章和第 10 章中介绍，程序设计模型将在第 6 章介绍。

(2) 中型嵌入式系统设计者需要具备的技能

对于设计中型的嵌入式系统来说，C 语言编程和 RTOS 编程以及程序建模是必须具备的技能。下面的知识对于设计者来说是很关键的。(i)任务以及 RTOS 对它们的调度。(ii)协同式和抢先式调度。(iii)Intel 处理器通信功能。(iv)共享数据的使用，以及对临界区和可重入函数的编程。(v)信号量、邮箱、队列、插槽以及管道的使用。(vi)中断延迟和满足任务最后期限的处理。(vii)

各种 RTOS 函数的使用。(viii)物理和虚拟设备驱动程序的使用。(第 8 章和第 10 章将详细介绍这 8 项技能以及一些示例，在第 11 章中将结合案例来学习它们的用法。)对于将要使用的特定微控制器，设计者必须使用具有应用编程接口(API)的 RTOS 编程工具。各种功能的处理方法，例如存储器分配、定时器、设备驱动程序和中断处理机制，已经作为 RTOS 的 API 存在了。设计者只需要知道硬件组织和这些 API 的使用即可。这样微控制器或者处理器就为设计者提供了一个小的系统元素，设计者只需要具备很少的知识就足够了。

(3) 复杂嵌入式系统设计者需要具备的技能

设计团队的成员必须协同设计，并解决硬件和软件设计中的高层次复杂问题。嵌入式系统硬件工程师应该具备硬件单元的全面技能，还要具备 C 语言、RTOS 和其他编程工具的基本知识。软件工程师应该具备硬件的基础知识，全面掌握 C 语言、RTOS 和其他编程工具的知识。最终的设计方案就通过系统集成来获得。

1.2 系统中的处理器

处理器(processor)是嵌入式系统的核心。对于嵌入式系统设计者来说，微处理器和微控制器的知识是必备的。在下面的内容中，我们假设读者对于微处理器和微控制器已经有了全面的了解(读者可以参考标准教材或者本书后面的“参考文献”中列出的参考书，深入了解在嵌入式系统设计中所集成的微处理器、微控制器和 DSP)。

1.2.1 系统中的处理器

处理器具有两个基本单元：程序流控制单元(CU)和执行单元(EU)。CU 中包含了一个取指单元，用于从存储器中取指令。EU 中含有执行指令的电路，用于数据转移操作以及数据从一种形式到另外一种形式的转换操作。EU 包含算术逻辑单元(ALU)，还包含执行程序控制任务指令的电路，例如挂起、中断或者跳转到其他指令集。它还可以执行调用指令或者跳转到另外一个程序并进行函数调用。

处理器运行取指和执行周期。在处理器的指令集中定义的指令按照它们从存储器中取回的顺序执行。处理器一般是 IC 芯片的形式，它也可以是 ASIC 或者 SoC 中的一个核。核是 VLSI 芯片上功能电路的一部分。

嵌入式处理器芯片或者核可以是下列之一：

(1) 通用处理器(GPP)：

a. 微处理器(参见 1.2.2 节)。

b. 微控制器(参见 1.2.3 节)。

c. 嵌入式处理器(参见 1.2.4 节)。

d. 数字信号处理器(参见 1.2.5 节)。

e. 媒体处理器(参见附录 E 的 E.1 节)。

(2) 作为附加处理器的专用系统处理器(ASSP)(参见 1.2.6 节)。

(3) 使用通用处理器(GPP)以及专用指令处理器(ASIP)的多处理器系统(参见 1.2.7 节)。

(4) 嵌入到一个专用集成电路(ASIC)中或者一个大规模集成电路(VLSI)中的 GPP 核或者 ASIP 核，或者 VLSI(ASIC)芯片中集成了处理器单元的 FPGA 核。

对于系统设计者，选择处理器时，有以下几个要点需要考虑：

(1) 指令集。

(2) 单个算术或者逻辑操作中操作数的最大位宽(8、16 或者 32 位)。

(3) 以 MHz 表示的时钟频率和百万指令/秒(MIPS)表示的处理速度(参见附录 B，了解以公制 Dhyrystone 表示的处理性能)。

(4) 处理器对用于满足最后期限的复杂算法的解决能力。

注意：

由于下列原因，需要使用通用处理器：(i)快速系统开发，用预定义的通用指令集中已知可用的指令处理，可以加快系统开发速度。(ii)如果印制板和 I/O 接口是根据 GPP 设计的，可以通过只修改印制板上 ROM 中的嵌入式软件，应用于一个新的系统。(iii)现有的编译器有助于用高级语言开发嵌入式软件。(iv)经过测试和调试过的处理器专用 API，以及先前为其他应用所设计的代码，能够加快一个新系统的开发速度。

1.2.2 微处理器

CPU 是一个集中取指和处理一组通用指令的单元。处理器指令集(参见 2.4 节)包含数据转移操作、ALU 操作、堆栈操作、输入和输出(I/O)操作以及程序控制、排序和管理操作。通用指令集(参见附录 A 中的 A.1 节)通常是特定的 CPU 专用的。任何一个 CPU 必须具备下列基本的功能单元。

(1) 一个控制单元，用于取指和控制一个给定命令或指令的顺序执行，并与系统其余部分进行通信。

(2) 一个 ALU 单元，用于对字节或者字的算术和逻辑操作。它可以立即处理 8、16、32 或者 64 位的字。

微处理器是一个 VLSI 芯片，芯片中有一个 CPU，还可以有其他附加的单元(例如，高速缓存、浮点处理算术单元、流水线和超标量单元)，这样可以加快指令的处理速度(参见 2.1 节)。

早期微处理器的取指和执行周期受到约为 1MHz 的时钟频率的限制。现在处理器的频率已经达到了 2GHz(Intel 在 2001 年 8 月 25 日发布了一款 2GHz 的处理器。这也是 IBM PC 面世 20 周年的日子。Intel 在 2003 年 4 月 14 日发布了 3 GHz 的奔腾 4 处理器)。自 2002 年初，几个非常复杂的嵌入式系统(例如，Gbps 收发器和加密引擎)中都集成了 GHz 的处理器(Gbps 的意思是每秒吉(G)比特。收发器的意思是，具有适当处理和控制功能(例如总线冲突)的发送和接收电路)。

早期微处理器的一个例子是 Intel 8085。它是一个 8 位的处理器。另外一个是 Intel 8086 或者 8088，是 16 位的处理器。Intel 80x86(也称为 x86)处理器是 8086 的 32 位后继处理器(这里，x 的意思是将 8086 扩展 32 位)。80x86 系列中 32 位处理器的例子是 Intel 80386 和 80486。大多数 IBM PC 使用 80x86 系列的处理器，PC 中集成的完成特定任务的嵌入式系统(例如图形加速器、磁盘控制器、网络接口卡)也使用这些处理器。

新一代 32 位和 64 位微处理器的例子是 Intel 公司著名的奔腾(Pentium)系列。它们具有超标量体系结构(参见 2.1 节)，还拥有强大的 ALU 和浮点处理单元(FLPU)(参见表 2.1)。奔腾 III 处理器在嵌入式系统中以 1GHz 的时钟频率运行的一个例子是“加密引擎”。它可以以 0.464 Gbps 的速度给数据加密。

表 1-1 列出了嵌入式系统中使用的一些重要微处理器。这些微处理器属于下列流派的系列：

表 1-1 嵌入式系统中使用的一些重要微处理器

流　派	微处理器系列	厂　商	CISC、RISC 或者兼有两种特征
流派 1	68HCxxx	Motorola	CISC
流派 2	(a) 80x86	Intel	CISC
	(b) i 860	Intel	带 RISC 的 CISC
流派 3	SPARC	Sun	RISC
流派 4	(a) PowerPC 601, 604	IBM	RISC
	(b) MPC 620	Motorola	

流派 1 和 2 的处理器具有复杂指令集计算机(CISC)体系结构(参见 A.1)。流派 3 和 4 的微处理器具有精简指令集计算机(RISC)体系结构(参见 A.1.4 节)。RISC 处理器处理指令的速度较快，每一条指令都是在一个时钟周期内处理的。此外，除了上面所提到过的改进的性能外，处理指令集中一条指令的速度也有了很大提高。Thumb 指令集是一种新的工业标准，降低了 RISC 处理器的代码密度(嵌入式系统中处理器的体系结构特征概念，CISC 和 RISC 处理器和处理器指令集，将在后面的附录 A 和 B 中介绍)。当系统需要执行比较复杂的计算(例如语音处理系统)时，可以使用 RISC。

系统设计者如何选择微处理器呢？这个问题将在 2.2 节中介绍。

注意：

当大型的嵌入式软件位于芯片的外部存储器时，要使用微处理器。当系统需要执行比较复杂的计算时，要使用 RISC 核微处理器。

1.2.3 微控制器

正如微处理器是计算系统中一个最基本的部分一样，微控制器是控制或者通信电路中一个最基本的组成部分。微控制器是一个单芯片 VLSI 单元(也称为“微计算机”)，这个单元虽然计算能力有限，却具有改进了的输入输出功能以及一些片上功能单元(关于各种功能单元的介绍参见 1.3 节)。微控制器尤其适用于具有片上程序存储器和设备的用于实时控制应用的嵌入式系统。

图 1-2 给出了微控制器中具有的功能电路(实线框中的部分)。图中还给出了给定微控制器系统的特定版本的专用单元(虚线框中的部分)。最近的一些微控制器也具有较高的计算和超标量处理能力(在 2.1 节中将介绍超标量体系结构的意义)。附录 C 给出了这些系列中具有代表性的微控制器的功能比较。

图 1-2　嵌入式系统中的微控制器芯片和核中的各种功能电路(实线框中的部分)，以及微控制器系统的特定版本的专用单元(虚线框中的部分)

嵌入式系统中应用的重要微控制器芯片通常属于表 1-2 中给出的 5 个系列流派。

表 1-2　嵌入式系统中使用的重要微控制器[1]

流　派	微控制器系列	厂　商	CISC、RISC 或者兼有两种特征
流派 1	68HC11xx 、 HC12xx 、 HC16xx	Motorola	CISC
流派 2	8051、80251	Intel	CISC
流派 3	80x86[2]	Intel	CISC
流派 4	PIC 16F84 或者 16C76、16F876 和 PIC18	Microchip	CISC
流派 5[3]	对 ARM9、ARM7 的改进	ARM、Texas 等	具有 RISC 核的 CISC 结构

1. 此外还有一些流行的微控制器。(i)Hitachi H8x 系列和 SuperH 7xxx。(ii)三菱 740、7700、M16C 和 M32C 系列。(iii)美国国家半导体的 COP8 和 CR16/16C。(iv)东芝 TLCS 900S。(v)基于低电压电池系统的得州仪器 MSP 430。(vi)三星 SAM8。(vii)Ziglog Z80 和 eZ80。

2. 80x86 微控制器版本(具有代表性的 80188 8 位处理器或者 80386 16 位处理器)，每一种都具有 64kB 的存储器，3 个定时器和 2 个 DMA 通道。

3. 参见 1.2.4 节和 B.1 节。

图 1-3 给出了在小型、中型和大型嵌入式系统中普遍使用的微控制器。在 C.1 节中(参见表 C-1 到 C-3)将介绍系统设计者在选择微控制器作为一个处理单元之前必须考虑的特征。

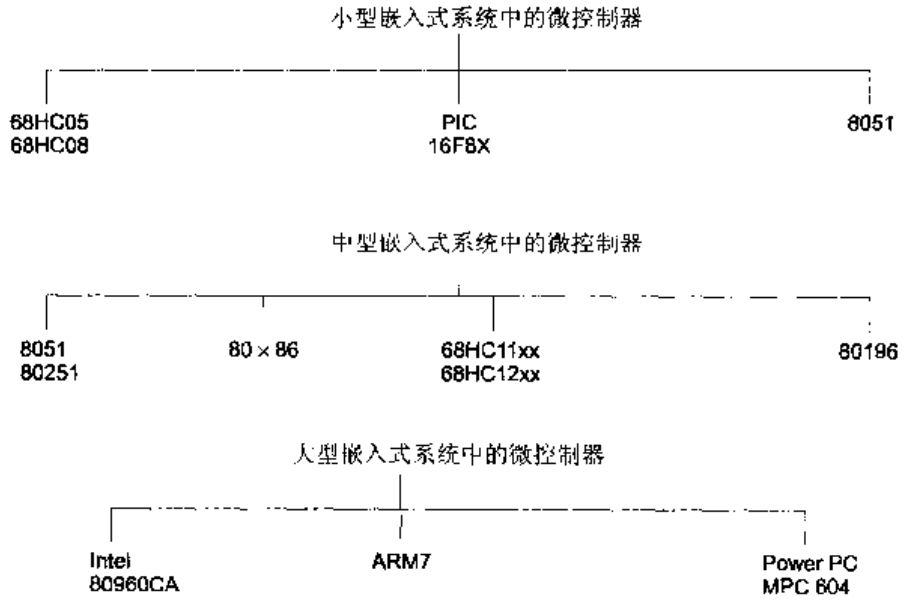


图 1-3 在小型、中型和大型嵌入式系统中普遍使用的微控制器

注意：

当小型的嵌入式软件或者嵌入式软件的一部分位于芯片的外部存储器时，以及当需要一些片上功能单元(例如中断处理、端口、定时器、ADC 和 PWM)时，要使用微控制器。

1.2.4 复杂系统的嵌入式处理器

对于快速、精确、复杂的计算以及复杂的实时应用，上面所提到的微控制器和微处理器是不够的。一个电子作战系统，例如，需要跟踪雷达的事先预警与控制系统(AWACS)，就是一个复杂的实时系统。需要特殊微处理器和微控制器，通常称为嵌入式处理器。当一个微控制器或者微处理器专门设计为具有下列功能时，使用术语嵌入式处理器比微控制器或者微处理器更恰当。

(1) 快速上下文切换，因此降低了复杂实时应用中的任务延迟(参见 4.6 节)。

(2) 自动 ALU 操作，没有共享数据问题。当一个位数较多的操作数放置在 2 个或者 4 个寄存器中时，不完全的 ALU(非自动的)就会导致数据共享问题(参见 2.1 节)。

(3) 提供嵌入式软件进行更加快速、精确和复杂计算的 RISC 核。

实时图像处理和空气动力学计算是 2 个需要高速、精确和大量计算以及快速上下文切换的例子。嵌入式系统中重要的嵌入式处理器芯片属于下列两个系列流派：

- 流派 1：ARM 系列 ARM7*和 ARM9*
- 流派 2：Intel 系列 i960
- 流派 3：AMD 系列 29050

附录 B 介绍了 ARM 系列处理器。它们既有单芯片 CPU 版本，又有可以嵌入到 VLSI 芯片

或者嵌入式系统的 SoC 解决方案中的文件版本(参见 1.6 节)。

Intel 系列 i960 微控制器也称为嵌入式处理器，因为它们拥有包括 CISC 和 RISC 在内的必需的特征(参见 A.1.4 节)。在一个版本中，还有一个 4 通道的 DMA 控制器(参见 2.6 节)。80960 包含一个 8 通道、248 向量的可编程中断控制器。

注意：

当既需要快速处理，又需要上下文快速切换和自动 ALU 操作时，应使用嵌入式处理器。

1.2.5 数字信号处理器

正如微处理器是计算系统的一个最基本单元一样，数字信号处理器(DSP)在需要进行信号处理应用的嵌入式系统中也是一个基本单元。举例来说，这些应用包括图像处理、多媒体、音频、视频、HDTV、DSP 调制解调器和无线电通信处理系统。DSP 还可以用于快速识别图像模式或者 DNA 序列。附录 D 详细介绍了嵌入式系统 DSP。

DSP 作为 GPP 是一个单芯片 VLSI 单元。它具有微处理器的计算能力，也具有一个乘法累加器(MAC)单元。现在，一般的 DSP 都具有一个 16×32 的 MAC 单元。

DSP 提供了快速的、离散时间的、信号处理指令。它具有超长指令字(VLIW)处理能力；能够快速处理单指令多数据(SIMD)指令；能够快速处理离散余弦变换(DCT)和反离散余弦变换(IDCT)。对于信号分析、编码、过滤、噪声消除、回音消除、压缩和解压缩等算法的快速执行来说，DCT 和 IDCT 是必备的。

嵌入式系统中使用的重要的 DSP 可分为表 1-3 列出的三个流派。

表 1-3 嵌入式系统中使用的重要的 DSP

流　派	DSP 系列	厂　商
流派 1	TMS320Cxx[1]	Texas
流派 2	SHARC	Analog Device
流派 3	5600xx	Motorola

1. 例如，定点 DSP TMS320C62XX 的时钟速度为 200MHz。详细描述参见 D.4 节。

注意：

当需要快速处理信号处理功能时，可使用 DSP。

1.2.6 嵌入式系统的专用系统处理器

最近出现了一种新的嵌入式系统。这种系统在其设计中另外集成了专用系统处理器 ASSP 芯片或核。这样的系统最近已经投入使用。

假设有一个实时视频处理嵌入式系统。嵌入式系统中需要进行实时处理的功能包括数字电视、高清电视解码器、视频转接器、DVD((Digital Video Disc，数字视频光盘)播放器、网络电话、视频会议和其他一些系统。这种处理需要一个视频压缩和解压缩系统，这种系统需要符合 MPEG2 或者 MPEG4 标准(MPEG 是 Motionpicture Expert Group 的缩写，即活动图像专家组)。MPEG2 或者 MPEG4 的信号压缩是在存储和传送之前完成的；解压缩是在取回或者接收这些信号之前完成的。对于 MPEG 压缩算法，如果运行了一个 GPP 嵌入式软件，则需要单独的 DSP

以获得实时处理。专用于这些特定任务的 ASSP 本身就能够提供一种快速解决方法。需要对 ASSP 进行配置，使其与嵌入式系统中的其他单元进行接口。

假设有一个嵌入式系统，该系统使用一个特定协议，并通过特定的总线结构将系统单元和另一个系统连接起来。再假设需要进行适当的加密和解密(对输出流加密，可以保护消息或者设计不被未知的外部实体获得)。对于这些任务，除了嵌入软件之外，还有必要嵌入一些 RTOS 特征(参见 1.4.6 节)。如果只有软件用于完成上述的任务，比起使用专用处理硬件来说，可能需要更长的时间。ASSP 芯片提供了这样一种解决方案。例如，一个 ASSP 芯片出自 i2Chip(http://www.i2Chip.com)，其中包含一个 TCP、UDP、IP、ARP 和以太网 10/100 MAC(介质访问控制)硬件逻辑。i2Chip 的 W3100A 芯片是一种纯硬件 Internet 连接解决方案。处理网络任务必备的 TCP/IP 堆栈处理软件现在已经可以作为硬件解决方案使用，这样做比使用系统 GPP 的软件解决方案要快 5 倍。它还是一种缺少 RTOS 特征的解决方案。在嵌入式系统中使用与这个 ASSP 芯片接口的同类型微控制器，可以加入以太网连接(对于术语 TCP、UDP、IP、ARP、以太网 10/100 和 MAC，可以参考关于计算机网络方面的参考书。建议您参考由 Rajkamal 编写，由 McGraw-Hill 在 2002 年出版的 *Internet and Web Technologies* 一书，了解这些协议中每一个位的含义)。

另外一个 ASSP 例子是“串口-以太网转换器(IIM7100)”。通过一个硬件协议堆栈来实现实时数据处理。这里不需要对应用程序或者固件进行修改，就能够提供低成本且最优的 RTOS 解决方案。

注意：

ASSP 是一种附加处理单元，用来代替嵌入式软件运行专用任务。

1.2.7 使用通用处理器的多处理器系统

在一个嵌入式系统中，可能需要多个处理器在严格的时间期限内快速执行一个算法。例如，在实时视频处理中，一秒钟内需要执行多次的 MAC 操作，这超出了一个 DSP 单元的处理能力。那么嵌入式系统就必须集成两个或者多个同步运行的处理器。

在一个便携式电话中，必须执行多项任务：(a)语音信号压缩和编码；(b)拨号；(c)调制和发送；(d)解调和接收；(e)信号解码和解压缩；(f)小键盘接口和显示接口处理；(g)基于短消息服务(SMS)协议的通信；(h)SMS 消息显示。对于所有的这些任务，一个处理器是不够的。需要多个处理器同步执行。

考虑一个视频会议系统。在这个系统中，使用了一种四分之一通用媒介格式(Quarter-CIF)。图像像素只有 144×176，而不是电视上的视频图像 525×625 像素。即便这样，图像样本也必须以每秒 144×176×30=760320 像素的速度采样，并且在将其传送到一个无线电通信或者虚拟专用网络(Virtual Private Network，VPN)之前，必须进行压缩处理(注意：在实时显示和动画中，帧速度应该为 25 或者 30 帧/秒；在视频会议中，帧速度应该在 15~10 帧/秒之间)。基于单个 DSP 的嵌入式系统不足以得到实时图像。实时视频处理和多媒体应用在很多情况下都需要在嵌入式系统中有多个微处理器单元(在附录 E 中将介绍一种媒体处理器，是一种代替多处理器进行实时图像处理的解决方案)。

注意：

当单个处理器不能满足同时执行多项不同任务的要求时，需要使用多处理器。为了得到最佳性能，所有处理器的操作都是同步的。

1.3 其他硬件单元

1.3.1 电源和低功耗管理

多数系统本身都有电源。这种供电方式具有一种特定的操作范围或者电压范围。嵌入式系统中各个单元的操作范围都属于以下 4 种之一：

(i) 5.0V±0.25V

(ii) 3.3V±0.3V

(iii) 2.0V±0.2V

(iv) 1.5V±0.2V

此外，嵌入式系统的微控制器中的闪存(数码相机使用的一种存储器)或者电可擦除可编程只读存储器(EEPROM)，以及 RS232 串行接口，需要提供 12V±0.2V 的电压(目前闪存只需要提供 5V 或者更低的电压就能工作)。

电压在嵌入式系统芯片中是按照如下方式提供的。电流和线路取决于处理器内部提供的管脚加上相关电路和芯片中的管脚的数量。管脚是成对的，由输入电压和地线组成。当连接电源线的时候，必须注意以下几点：

(1) 一个处理器可能有两个以上 V_{DD} 和 V_{SS} 管脚。这样能够把电能传输到所有的区域，减小了各个区域之间的相互干扰。在系统处理器以及其他单元中，每一对 V_{DD} 和 V_{SS} 管脚都应有一个独立的射频频率干扰旁路电容，这些旁路电容应尽可能靠近 V_{DD} 和 V_{SS} 管脚。

(2) 应该分别给(a)外部 I/O 驱动端口(b)定时器(c)时钟以及复位电路单独供电。时钟和复位电路(参见 1.3.2 节和 1.3.3 节)应该进行特殊设计，以避免任何的射频频率干扰。I/O 设备的功耗可能比处理器的其他内部单元的功耗要大。定时器即使在等待状态下，消耗的功率也是恒定的。因此，这三种电路应该分别单独供电。ADC 需要无噪声电压输入。

(3) 每一对 V_{DD} 和 V_{SS} 管脚、模拟地、模拟参考和模拟输入电压线、ADC 单元的数字地和系统中的其他模拟单元，都应该有单独的电源连接。

某些系统本身不含有供电系统，它们使用外部电源或者使用充电泵来供电：(1)网络接口卡(NIC)和图形加速器都是本身没有供电系统，而使用 PC 电源的嵌入式系统。(2)充电泵有一个串联的二极管，后面跟随一个充电电容。二极管从外部信号获得正向偏压；例如，当计算机使用鼠标时，来自于 RTS 信号。充电泵从一个非电源线取得电能(9 管脚的 COM 端口有一个称为发送请求(RTS)的信号。它是一个低电平有效的信号。大多数时间它处于非有效的逻辑“1”状态(约为 5V)。当鼠标处于空闲状态时，鼠标中的充电泵使用它存储电荷。当使用鼠标时，充电泵消耗电能。整流电路从这个电容中获得输入，提供需要的电能。当非接触智能卡插入到读卡器中后，卡中的充电泵使用来自于主机的无线电能量(参见 1.6.6 节)。

低电压系统是使用 LVCMOS(低电压 CMOS)门和 LVTTL(低电压 TTL)建立的。使用 3.3V、2.5V、1.8V 和 1.5V 电压系统以及 IO(输入-输出)接口，而不使用传统的 5V 系统，能够大大降低功耗，可以方便地在下列情况中使用：

(a) 在便携式或者手提式设备中，例如便携式电话(CMOS 电路使用 3.3V 电压，功耗降低了近一半，约为$(3.3/5)^2$。需要对电池重新充电的时间间隔也将缩短 2 倍)。(b)在一个几何空间较小的系统中，低电压系统处理器和 IO 电路产生的热量较少，可以被封装到一个较小的空间中。

门传播延迟和操作电压通常是成反比的。因此，5V 系统处理器和单元也可在大多数系统中使用。

嵌入式系统可能需要连续运行，因此系统的设计就受到运行时低功耗的限制。系统在运行、等待和空闲状态下的总体功耗也应该受到限制。嵌入式系统处理器中任何时刻所需要的电流取决于处理器的状态和模式。下面是处理器在 6 种状态下的功耗：

(i) 当只有处理器运行时：也就是处理器正在执行指令时，电流为 50 mA。

(ii) 当处理器以及外部存储器和芯片运行时：也就是取指和执行同时进行时，电流为 75mA。

(iii) 当只有处理器处于停止状态时：也就是取指和执行都停止，并且处理器的所有结构单元的时钟都禁止时，电流为 15mA。

(iv) 当处理器以及外部存储器和芯片处于停止状态时：也就是取指和执行都停止，并且所有系统单元的时钟都禁止时，电流为 15mA。

(v) 当只有处理器处于等待状态时，也就是取指和执行都停止，但是处理器的结构单元的时钟没有禁止时，例如定时器，电流为 5mA。

(vi) 当处理器、外部存储器和芯片处于等待状态时。等待状态意味着取指和执行都停止了，但是处理器结构单元，以及外部 IO 单元和动态 RAM 刷新的时钟还没有停止，此时电流为 10mA。

嵌入式系统必须从加电开始连续执行任务，还有可能一直处于加电状态，因此，节电在执行过程中是很重要的。嵌入式系统中使用的微控制器必须提供 Wait 和 Stop 指令，能够在低电压模式下运行。一种做法是在软件中集成 Wait 和 Stop 指令。另外一种做法是在空闲状态下选择低电压模式，从而在最低电压下运行系统。还有一种做法是在特殊软件部分运行的时候(例如定时器和 IO 单元)，禁止处理器的某些不必要的结构单元(例如高速缓存)运行，并将它们处于断开连接状态。在 CMOS 电路中，功耗只发生在输入端发生变化的时候。因此，不必要的脉冲干扰和频繁的输入变化会增加功耗。VLSI 电路设计具有独特的方法来避免功耗增加。通过消除所有可以消除的脉冲干扰来设计电路，从而消除频繁的输入变化。

注意：

1. 当处理器接收到 Stop 指令后，它会进入停止状态。停止状态也会在下列情况下发生：(1)禁止处理器的时钟输入时。(2)禁止外部时钟电路工作时。(3)处理器在自动关电模式下工作时。当处于停止状态时，处理器与总线就断开了。(总线处于三态状态。)停止状态可以转换为运行状态。向运行状态的转换可能是因为用户中断，也可能是因为周期性发生的唤醒中断。

2. 处理器在(i)收到 Wait 指令，该指令减慢或者禁止一些包括 ALU 在内的处理器单元的时钟输入，或者(ii)当外部时钟电路停止工作时，会进入等待状态。当(i)发生了一个中断，或者(ii)收到复位信号后，处理器会从等待状态转换为运行状态。

3. 通常情况下，时钟频率每下降 100KHz，功耗就降低 2.5mW。因此将时钟频率从 8000KHz 降低到 100KHz，功耗会降低约 200mW，也就相当于时钟不工作。(注意，总的功耗(需要的能量)可能不会降低。这是因为由于时钟频率的降低，计算所需要的时间可能会增加，所需要的总能量等于每秒的功耗乘以时间。)25mW 的功耗通常是运行定时器以及很少的其他单元所需

要的剩余功耗。通过使时钟低频运转，或者让处理器处于断电模式，有如下几个优点：(i)减少发热。(ii)由于门的功耗降低，也降低了射频干扰(辐射的 RF(射频)功耗取决于门中的 RF 电流，由于发热减少后，漏极和通道之间的“开启”状态阻抗增加了，这个电流也就减小了)。

近来，出现了一种新的技术，将时钟管理电路与振荡电路一起使用。这种技术用于复杂的嵌入式片上系统(SoC)中。通过将时钟倍频电路和时钟分频电路(2 分频)相结合，可以产生 2~16 个同步时钟。此外，总线上引入的时钟信号可以在提供给快速操作电路之前，首先进行分频，然后再进行倍频。这样就能够降低门之间的功耗。对时钟管理器电路进行配置，使之能够在实时运行过程中，智能化地为处于控制范围内的各部分的电路提供适当频率的时钟(注意：必须使用一种复杂的技术：锁相环(phased delay locked loops)。当使用常见的计数器逻辑电路时，在门上就会有连续变化的延迟(例如，10ns±2ns)。单独使用计数器不能够设计出同步时钟)。

注意：

每个系统中都必须有一个内部电压或者一个充电泵，这是最基本的。嵌入式系统必须从加电开始到断电结束，一直连续执行任务，甚至总是处于“加电”状态。通过使用“Wait”和“Stop”指令进行实时编程，并在不需要的时候将某些单元禁用，可以在程序运行时节约电能。当需要控制功耗时，也可以降低时钟频率来操作。然而所有的任务必须在设定的最后期限之内完成，并且所有需要全速处理的任务必须快速处理(关于“最后期限”的概念，参见 4.6 节)。对于嵌入式系统软件，在其设计阶段的性能分析必须包括程序执行过程中和等待过程中的功耗分析。良好的设计必须使功耗和程序的快速、高效执行之间的冲突得到最优化处理。

1.3.2 时钟振荡电路和时钟单元

除了电源之外，时钟就是系统中又一个最重要的单元了。处理器需要有一个时钟振荡(clock oscillator)电路。时钟控制着 CPU、系统定时器和 CPU 机器周期的各种时钟控制需求。机器周期用于：

(i) 从存储器中取回代码和数据，然后在处理器上对它们进行译码并运行。

(ii) 将结果传回到存储器中。

时钟控制着执行一条指令的时间。时钟电路使用一个石英(处理器外部)或者陶瓷谐振器(与处理器内部相关)或者一个与处理器连接的外部振荡器 IC。(a)对于电路中的温漂，石英谐振器的时钟频率稳定性最高。石英与适当的电阻并联，并且在共振的两个管脚都串联一对电容，产生的频率等于石英频率或者是石英频率的两倍。此外，石英要尽可能地与处理器的两个管脚接近。(b)如果处理器有内部陶瓷谐振器的话，可以不使用外部石英，并能够产生合理但是稳定性不高的频率(陶瓷谐振器的温漂通常是每个月大约 10 分钟，石英的温漂通常为每个月 1 分钟或者 5 分钟)。(c)外部基于 IC 的时钟振荡器的功耗明显比内部处理器谐振器的高。然而，它的驱动能力较强，当需要同时驱动嵌入式系统中不同的电路时比较有用。例如，一个多处理器系统需要一个具有较强驱动能力的时钟电路，能够同时驱动所有的处理器。

注意：

处理单元需要有一个高度稳定的振荡器，处理器的时钟输出信号为所有系统单元提供同步时钟。

1.3.3 系统需要的各种计时和计数功能的实时时钟和定时器

将定时器电路进行适当的配置后，就产生了系统时钟(system-clock)，也称为实时时钟(RTC)。RTC 被调度程序使用，也可用于实时编程。RTC 是这样设计的：假设一个处理器每 0.5us 产生一个时钟输出。当系统定时器通过软件指令进行配置，在处理器给出 200 个时钟输入后会发生超时，则每秒有 10000 个中断(嘀哒)。RTC 的嘀哒频率是 10KHz，它每 100us 中断一次。RTC 还用于获得软件控制的延迟和超时。

系统中可能需要多个使用系统时钟(RTC)的定时器，来满足各种时钟和计数需求。定时器和计数器的描述参见 3.2 节。

注意：

为了完成任务调度和实时编程，需要系统时钟(RTC)。系统时钟还驱动着满足系统中各种计时和计数需求的定时器。

1.3.4 复位电路、加电复位和 Watchdog 定时器复位

复位意味着处理器从起始地址开始执行指令。这个起始地址是处理器程序计数器(或者是 x86 系列处理器中的指令指针和代码段寄存器)加电时的默认设置。处理器复位之后，从存储器的这个地址开始取程序指令。在某些存储器中(例如 68HC11 和 HC12)有两个起始地址。一个是作为加电复位向量，另外一个是作为执行 Reset 指令后或者发生超时(例如来自于 watchdog 定时器)之后的复位向量。

复位电路激活固定的周期数(几个时钟周期)后处于无效状态。处理器电路保持复位管脚处于有效状态，然后使之处于无效状态，使程序从默认的起始地址开始执行。如果复位管脚或者内部复位信号与系统中其他的单元(例如 I/O 接口或者串行接口)相连接，它会被处理器再一次激活；它成为一个输出管脚，用于驱动系统中其他单元处于复位状态。在处理器动作之后使复位信号无效(deactivation)，程序会从起始地址开始执行。

通过下列方式可激活复位电路：

(1) 外部复位电路，在加电时激活，接通系统的复位或者测试低电压(例如当系统需要 5V 电压，而实际电压小于 4.5V 时)。这个电路的输出端连接到处理器的复位管脚。这个电路可以是一个简单的 RC 电路、一个外部 IC 电路或者是一个定制的 IC。例如，MAX 6314 和摩托罗拉的 MC 34064。

(2) (a)软件指令，或者(b)watchdog 定时器的时钟输出(或者 68H11 和 68H12 系列中称为 COP 的内部信号)，或者(c)时钟监视器测试出一个由于出错而导致的低于阈值的频率。

watchdog 定时器是一个定时设备，会在事先定义超时之后将系统复位。这个时间通常是配置的，watchdog 定时器在加电后的前几个时钟周期中被激活。其应用很广泛。在许多嵌入式系统中，通过 watchdog 定时器进行复位是最基本的，因为如果产生了错误或者程序中断之后，它会帮助恢复系统。重新启动后，系统可以正常运行。大多数的微控制器都有片上 watchdog 定时器。

考虑一个控制温度的系统。假设当程序开始执行时，传感器输入端工作正常。然而，在达到想要的温度之前，传感器电路产生了某种错误。如果系统没有被复位，控制器将继续传送电流，而不停止。考虑另外一个例子：一个控制机器人的系统。假设机器人胳膊中的接口发动机控制电路在运行时产生了错误。在这种情况下，如果没有 watchdog 定时器，机器人的胳膊会

继续运动。如果不停止，机器人会自己折断自己的胳膊！

注意：

与系统相关的一个重要电路是其复位电路。复位并在加电后运行的程序可以是下列程序之一：(i)从开始就执行的系统程序。(ii)系统引导程序。(iii)系统初始化程序。在控制应用程序中，watchdog 定时器复位是一个非常有用的特征。

1.3.5 存储器

在一个系统中，有各种类型的存储器。图 1-4 给出了系统中出现的存储器的各种形式。它们是：(i)在一个微控制器中作为寄存器(存储临时数据和堆栈)的 256 或者 512 字节的内部 RAM。(ii)存储大约 4~16kB 程序(微控制器)的内部 ROM/PROM/EPROM。(iii)外部 RAM，存储临时数据和堆栈(在大多数系统中)。(iv)内部高速缓存(某些微处理器)。(v)EEPROM 或者闪存(在许多将处理过程的结果(例如周期性的系统状态和数码相机的图像、歌曲，或者经过适当压缩的语音)保存到非易失性存储器的系统中)。(vi)外部 ROM 或者 PROM，存储嵌入式软件(在几乎所有基于微控制器的系统中)。(vii)端口的 RAM 内存缓冲区。(viii)高速缓存(在超标量微处理器中)。(关于这些内容的进一步了解，请参见 2.1 节和 2.3 节。)

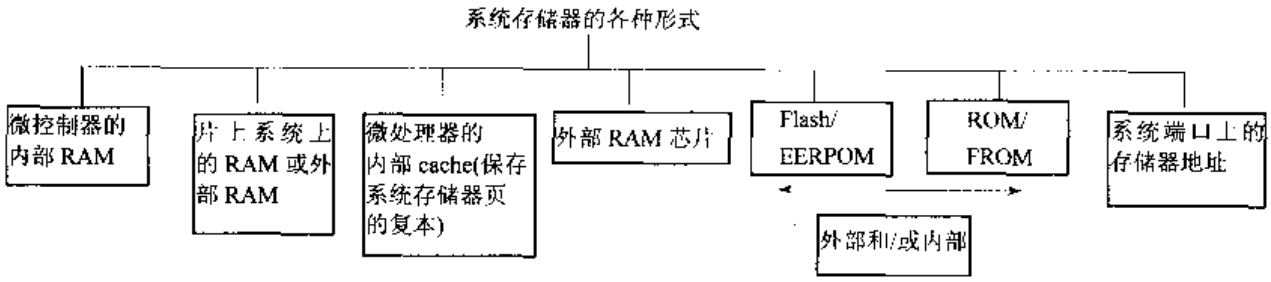


图 1-4 系统中各种形式的存储器

表 1-4 列出了嵌入式系统中存储器的功能。ROM、PROM 或者 EPROM 中嵌入了系统专用的嵌入式软件。

表 1-4 嵌入式系统中存储器的功能

所需要的存储器	功 能
ROM 或 EPROM	用来存储应用程序，处理器从中取指令代码。存储用来进行系统引导和初始化的代码、初始的输入数据和字符串。存储 RTOS 的代码。存储指向不同服务例程的指针(地址)
RAM(内部 RAM 或外部 RAM)和用于缓冲区的 RAM	在程序运行时存储变量并存储堆栈。例如，在语音或图像应用中用来存储输入和输出缓冲区
EEPROM 或闪存	存储非易失性的处理结果
高速缓存(cache)	在外部存储器之前存储指令和数据的备份，在快速处理时用来存储临时结果

注意：

系统可以将下列程序数据写入到(放置到)微控制器的内部 ROM 或 PROM 中，也可以放置在外部 ROM 或者 PROM 中：引导程序、初始化数据、初始屏幕显示字符串或者系统的初始状态、各种任务程序、ISR 和内核。系统中的 RAM，用来保存临时数据，以及程序运行时所需要的堆栈和缓冲区。系统还具有存放非易失性结果的闪存。

1.3.6 输入、输出和 I/O 端口，IO 总线和 IO 接口

系统通过输入端口从物理设备(例如，按键、传感器和变频电路)获得输入。系统中的控制电路从传感器和变频电路获得输入。信号接收器或者网卡从通信系统获得输入(通信系统可以是一个传真机、调制解调器，或者广播服务)。网络信号也可以在端口上接收。下面来看一下巧克力自动售卖机中的系统。它从一个端口获得输入，这个端口收集儿童投入的硬币。这里假设只有儿童可以使用这台奇妙的机器。再来看一下移动电话中的系统。用户通过按键直接或者间接(通过从存储器中重新调用号码)输入移动电话号码。键盘通过输入端口与系统连接。处理器根据每一个输入端口的存储器缓冲地址，也称为端口地址，来识别按键。正如一个字节或字的存储单元是通过地址来标识的一样，每一个输入端口也是用地址标识的。系统通过对端口地址的读操作来获得输入。

系统还具有输出端口，通过这些端口可以向外部发送输出字节。输出结果可以输出到 LED(发光二极管)或者 LCD(液晶显示)面板上。例如，计算器或者移动电话系统向 LCD 显示屏发送输出数字或者 SMS 信息。系统可以向打印机发送输出数据。输出结果可以发送给通信系统或者网络。控制系统向报警器、致动器、熔炉或者锅炉发送输出数据。机器人向其各种电动机发送输出数据。每一个输出端口都是通过本身的存储器缓冲地址(称为端口地址)标识的。系统通过一个写操作向端口地址发送输出数据。

通用端口可以同时进行输入和输出(I/O)操作。例如，移动电话系统通过一个无线通信通道既可以发送输出数据，又可以接收数据。每一个 I/O 端口也是通过地址标识的，在这个地址上可以进行读写操作。

关于端口的详细内容，请参见 3.1 节。有两种类型的 I/O 端口：并行端口和串行端口。系统可以从串行输入端口中获得串行比特流，通过串行输出端口发送串行比特流。例如，通过串行端口，系统可以使用调制解调器获得或者发送信号。串行接口还可以帮助进行远距离通信和互连。串行端口可以是一个串行 UART 端口、一个串行同步接口或者其他串行接口端口(UART 是指通用异步收发)。

一个系统端口可以从多个通道获得输入，或者向多个通道发送输出数据。信号分离器从各个通道获得输入，并选择一个通道的输入发送给系统。多路器从系统获得输出数据，并发送给另外一个系统。

一个系统可能要与其他几个设备和系统连接。对于网络互连系统，有不同类型的总线可供使用。例如，I^2C、CAN、USB、ISA、EISA 和 PCI(关于总线的详细信息，请参见 3.3、3.4 节和附录 F)。

注意：

系统通过并行或者串行端口与外部物理设备和系统相连。信号分离器和多路器通过一个通用路径可以与多个通道进行信号通信。系统经常通过I/O总线与其他设备和系统进行网络互连：例如，I^2C、CAN、USB、ISA、EISA 和 PCI 总线。

1.3.7 中断处理器

一个系统可能有许多设备，系统处理器必须为每个设备运行一个适当的中断服务例程(ISR)，从而控制和处理每一个设备的请求。每一个系统中都必须有一个中断处理机制，处理来自于系统中各个过程的中断：例如，传送来自键盘或者打印机的数据(关于系统的中断及其控制(处理)机制的详细信息，请参见第 4 章)。关于中断以及通过编程来处理中断的要点如下：

(1) 一个处理器中可以有多个或者多组中断源(参见 4.5 节)。中断可能是一个硬件信号，表示一个事件的发生(例如，实时时钟连续更新一个特定存储器地址上的数值；这个数值的变换就是导致中断产生的一个事件)。中断也可以通过定时器、处理器程序的中断指令或者处理过程中的错误引发。这个错误可能是由于一个非法的取码操作、让 0 作为除数或者 ALU 操作过程中的上溢或者下溢而导致的。中断也可以通过一个软件定时器产生。软件中断可能在一个异常条件下产生，这个条件是在程序运行的过程中形成的。

(2) 系统应该为这些中断源和中断服务划分优先级，并按照这个优先级来响应中断(参见 4.6.5 节)。

(3) 某些中断源是不能被屏蔽的，也不能被禁用。这些中断源被定义为具有最高的优先级。

(4) 为了在中断发生后完成特定的任务，处理器的当前程序必须转换到相应的服务例程。例如，如果按下了一个键，ISR 就读取这个键的值，并将该值存储到处理器存储器地址中。如果按下了一组键，例如移动电话，则 ISR 读取键值，并调用一个任务来拨号。

(5) 在微控制器中有一个片上可编程的单元，完成中断处理机制。

(6) 应用程序或者调度程序要调度并控制特定应用程序中的中断例程的运行情况。

调度程序赋予 ISR 的优先级通常要高于应用程序的任务。

注意：

系统必须有一个中断处理机制，在发生物理设备、系统和软件中断时执行 ISR。

1.3.8 DAC(使用 PWM)和 ADC

假设系统需要给出一个控制电路的模拟输出来进行自动操作。模拟输出可以输出到一个直流电动机或者熔炉的供电系统中。微处理器中的脉宽调制器(PWM)的操作如下：使脉宽与需要的模拟输出成比例。对于一个 8 位的 DAC 操作，PWM 输入的范围为 00000000~11111111。PWM 单元的输出到达一个外部积分器，然后产生需要的模拟输出。

假设一个积分器电路(微处理器外部的)在脉宽为整个脉宽周期的 50%时，输出 1.024V 的电压，在脉宽为整个脉宽周期的 100%时，输出 2.047V 电压。当通过使 PWM 输出控制寄存器中的数值减半，使脉宽为 25%时，积分器的输出将变为 0.512V。

现在，假设积分器是在双电压(正负)下操作的。再假设当积分器电路给出 1.023V 的输出时，脉宽为整个脉宽周期的 100%；当产生−1.024V 的输出时，脉宽为整个脉宽周期的 0%。当通过使 PWM 输出控制寄存器中的数值减半，而使得脉宽为 25%时，积分器的输出将变为 0.512V；

当脉宽为 50%时，积分器的输出将变为 0V。

从这些信息中可以发现，根据一个范围在 00000000~11111111 之间的 PWM 寄存器位可以得到转换后的位，该公式的推导将作为课后练习留给读者。

系统微控制器中的 ADC 可以有许多应用，例如数据采集系统(DAS)、模拟控制系统和声音数字化系统。假设一个系统需要从传感器或者变频电路中读取一个模拟输入。如果用系统中的 ADC 单元转换为位，在将这些位进行处理后，也可以产生一个输出。这样就通过结合使用 ADC 和 DAC，产生了一个自动操作控制。

转换后的位可以发送给用于显示数字的端口。这些位还可以传送给存储器地址、串行端口或者并行端口。

处理器可以处理转换后的位，并产生一个调制脉宽编码(PCM)输出。PCM 信号用来将声音信号数字化。

关于 ADC 的要点如下：

(1) 在 ADC 中，需要有单模拟参考电压源，或者双模拟参考电压源。它要么只设置模拟输入上限，要么同时设置下限和上限。对于单参考电压源，下限设置为 0V(隐含意思指地)。当模拟输入等于下限值时，ADC 产生输出的所有位为 0；当模拟输入等于上限值，ADC 产生输出的所有位为 1。例如，假设一个 ADC 的上限或者参考电压值设置为 2.255V。设下限参考电压值为 0.255V。两个限制值的差为 2V。因此，分辨率为(2/256)V。如果 8 位的 ADC 模拟输入为 0.255V，转换后的 8 位是 00000000。当输入为(0.255V+1.000V)=1.255V 时，输出的位是 10000000。当模拟输入为(0.255V+0.50V)时，输出是 01000000(从这些信息中，可以推导出一个公式，根据一个给定的模拟输入 v(V)获得转换后的数据，该公式的推导作为练习留给读者)。

(2) 根据需要转换的分辨率，ADC 可以是 8 位、10 位、12 位或者 16 位。

(3) 转换开始信号(STC)或输入将转换初始化为 8 位。在系统中，由一条指令或者一个定时器触发 STC。

(4) 存在一个转换结束(EOC)信号。在系统中，由一个寄存器中的标志位表明转换的结束，并产生一个中断。

(5) 存在一个转换结束的时间限制。

(6) 采样和保持(S/H)单元用来在固定的时间之内对输入进行采样，并保持采样，一直到转换结束。

嵌入式系统微控制器中的 ADC 单元可以有多通道。它可以连续从与不同模拟源内部相连的各个管脚获得输入。

注意：

对于自动控制和信号处理应用，系统必须为数模转换(DAC)和模数转换(ADC)单元提供必要的接口电路和软件。DAC 操作是在微控制器中的 PWM 单元与外部积分器芯片的结合下完成的。在语音处理、使用仪器、数据采集系统和自动控制系统中需要 ADC 操作。

1.3.9 LCD 和 LED 显示

系统需要一个接口电路和软件，来显示状态或者一行信息、多行显示或者闪烁显示。LCD 屏幕可以显示多行字符，还可以显示一个小图像或者图标(称为象形图)。移动电话系统的最近创新是将显示屏变为蓝色，表示有电话打进来。第三代系统电话既有图像显示又有图形显

示。LCD 耗电很少。它是通过一个电源或者电池(计算器上的太阳能面板)供电的。LCD 是一个二极管，在 3~4V 之间、频率为 50~60Hz、电流小于约 50μA 的电压脉冲下吸收或者发光。在不发光的时候，晶体前后平面的极性是相同的，发光时，极性是相反的。在这里，极性意味着逻辑“1”或者逻辑“0”。在矩阵显示的情况下，通常使用 LSI(小规模集成电路)显示控制器。

为了显示系统的运行状态，需要使用 LED，当系统处于运行状态时，LED 点亮。闪烁的 LED 可以表示一项特定的任务即将结束或者正在运行。它还可以表示正在等待信息的状态。LED 可以发出黄色、绿色、红色光(或者远距离控制器中的红外线)，工作电压为 1.6~2.0V。LED 需要 5~12mA(在闪烁显示状态下电流较小)的电流，比 LCD 亮得多。因此，在闪烁显示和只显示有限的几个数字时，可以在系统中使用 LED。

注意：

为了完成显示和通信功能，系统中使用 LCD 矩阵显示和 LED 阵列。系统必须提供必要的接口电路和软件，供 LCD 显示控制器和 LED 接口端口的输出使用。

1.3.10 小键盘/键盘

小键盘或者键盘是获得用户输入的重要设备。系统必须提供必要的接口和按键反跳电路以及软件，使系统能够从一组按键或者键盘(或者小键盘)获得输入。小键盘最多可以有 32 个按键。键盘可以有 104 个或者更多的按键。键盘或者小键盘可以与处理器串行接口，或者直接通过串口或者并口，或者通过控制器与处理器并行接口。

注意：

可以将小键盘或者键盘连接到系统上，获得输入。系统必须提供必要的接口电路和软件，以便从按键或者通过控制器直接获得输入。

1.3.11 脉冲拨号电路、调制解调器和收发器

对于通过电话线、无线或者网络进行连接的用户，系统提供了必要的接口电路。它还为通过电话线进行的脉冲拨号、用于传真机的调制解调器互连、Internet 包路由、发送和连接 WGA(Wireless gateway，无线网关)或者蜂窝系统提供了软件。收发器是一个既可以发送又可以接收字节流的电路。

注意：

通信系统中使用了脉冲拨号电路、调制解调器或者收发器。系统必须直接或者通过控制器为拨号电路、调制解调器和收发器提供必要的接口电路和软件。

1.3.12 GPIB(IEEE 488)连接

一个系统可能需要与另外一个仪器或者系统链接。IEEE 488 GPIB(General purpose Interface Bus，通用接口总线)连接是一个标准的总线，这个总线最初是由 HP(惠普)开发的，用来连接测量系统和仪器系统。仪器系统中的嵌入式系统使用这个接口标准。

1.3.13 嵌入式系统硬件的连接和接口总线及单元

嵌入式系统硬件中的总线和单元需要连接和接口。一种做法就是集成一个粘合逻辑电路(glue logic circuit)(不使用单独的门、缓冲区和译码器，而是使用粘合逻辑电路)。粘合电路用来完成电路之间以及所有芯片和主芯片(处理器和存储器)之间的逻辑行为。嵌入式系统的粘合逻辑电路可能是一个处理器与外部存储器的粘合电路，使得适当的芯片选择信号——依照系统存储器——能够映射每一个存储器芯片(参见 2.5 节)。粘合逻辑电路还包含一个将并行和串行接口与外设相连的电路(关于端口的详细介绍，参见第 3 章)。粘合电路极大地简化了整个嵌入式系统电路。使用粘合电路的一个例子是连接处理器、存储器以及与 LCD 显示矩阵和小键盘接口的端口。

编写和配置下面给出的粘合电路之一。(i)PAL(Programmable Array Logic，可编程逻辑阵列)。(ii)GAL(Generic Array Logic，通用逻辑阵列)。(iii)PLD(Programmable Logic Device，可编程逻辑设备)。(iv)CPLD(Combined PLD，组合 PLD)。(v)FPGA。这些设备可以通过一个称为设备编程器(device programmer)的系统来配置和编程。

PAL 具有一个与或逻辑阵列。PAL 只实现组合逻辑电路。GAL 是另外一种逻辑阵列，比 PAL 先进，提供时序电路的实现。因此它也提供锁存器、计数器和寄存器电路的实现。PLD 是另外一种可编程逻辑设备。ROM 也是一种 PLD。CPLD 是一种集成了 PLD 的组合电路。它是一种实现数模混合功能的逻辑设备。CPLD 还可以帮助实现 PLC(Programmable Logic Controller，可编程逻辑控制器)的控制功能和设计。FPGA 有一个宏单元，是门和触发器的组合。一个阵列有多个宏单元。阵列中或者宏单元之间的连接可以用一个设备编程器熔断。

注意：

通过配置或者编程 PAL、GAL、CPLD 或者 FPGA 连接而设计的粘合电路为总线提供了接口，也为单芯片上嵌入式系统硬件的所有单元提供了接口。

1.3.14 案例中所需要的硬件单元

图 1-5 给出了嵌入式系统中必需的单元。这里选择了 6 个例子来说明具有不同复杂性的系统。分别是：

- 巧克力自动售卖机(对其进行编程的案例研究将在 11.1 节给出)
- 数据采集系统
- 机器人
- 移动电话
- 汽车间距保持不变的自适应巡航控制(ACC)系统(对其进行编程的案例研究将在 11.3 节给出)
- 声音处理器和存储系统(输入、压缩、存储、解压缩、录制和重新播放)

记住，RTC、定时器、空闲状态、低电压状态、Watchdog 定时器和串行 IO 端口、UART 端口和粘合逻辑电路几乎在所有的应用中都需要，因此没有在表中列出。在这里给出的数值只参考了典型系统。

表 1-5　6 个嵌入式系统示例需要的硬件以及典型数值

所需硬件	巧克力自动售卖机[1]	数据采集系统	机器人	移动电话	汽车间距保持不变的自适应巡航控制系统[2]	声音处理器
处理器	微控制器	微控制器	微控制器	片上多处理器系统	微处理器	微处理器+DSP
处理器内部总线宽度(位)	8	8	8	32	32	32
CISC 或 RISC 处理器体系结构	CISC	CISC	CISC	RISC	RISC	RISC
高速缓存和 MMU	No	No	No	Yes	No	Yes
PROM 或 ROM 存储器	4kB	8kB	8kB	1MB	64kB	1MB
EEPROM+闪存	No	512B	256B	32kB	4kB	4MB[4]
RAM 中断处理器	片上 256B	片上 256B	片上 256B	SoC 上 1MB[3]	片外 4kB	片外 1MB[5]
输入-输出端口	多个端口——硬币分类输入端口、交货端口和显示端口	传感器和致动器多端口	电动机和角度编码器多端口	小键盘和显示端口	开关按钮和显示端口	语音重新播放输出端口输入端口
收发器	No	No	No	用于连接移动电话服务	用于跟踪雷达	No
GPIB 接口	No	Yes	No	No	No	No
信号或者事件的实时检测(事件发生时捕获并比较事件)	No	Yes	No	Yes	Yes	Yes

(续表)

所需硬件	巧克力自动售卖机[1]	数据采集系统	机器人	移动电话	汽车间距保持不变的自适应巡航控制系统[2]	声音处理器
用于 DAC 的脉宽调制	No	Yes	Yes	Yes	Yes	Yes
模数转换(位)	No	Yes	Yes	Yes	Yes	Yes
调制解调	No	No	No	Yes	No	No
数字信号处理指令	No	No	No	Yes	No	Yes
非线性控制器指令	No	No	No	No	Yes	No

1. 参见 11.1 的案例研究。

2. 参见 11.2 节的案例研究。间距保持不变的意思是，汽车之间的距离很稳定。雷达和收发器组用来测量与前面汽车的距离。EEPROM 用来存储自适应算法参数。

3. 额外 RAM 是由于对正在处理的声音输入的缓冲区存储器 RAM 的需要。

4. 额外 RAM 是由于对正在处理的声音输入和输出的缓冲区存储器 RAM 的需要。

5. 语音信号的缓冲存储器。

6. 用来存储声音

注意：

多种应用的嵌入式系统可能需要不同的硬件处理平台。然而，通过选择嵌入式软件，相同的硬件平台也可以用于完全不同的应用或者同一系统新的升级版本。

1.4 嵌入系统软件

软件是最重要的部分，是嵌入式系统的核心。

1.4.1 产品的最终机器可实现软件

嵌入式系统处理器和系统需要专用于系统给定应用的软件。系统的处理器处理指令代码和数据。在设计的最后阶段，这些处理过的指令代码和数据被放置到存储器中，用于执行任务。最后阶段软件也称为 ROM 映像。为什么这样说呢？正如图像是像素独特的排列方式一样，嵌入式软件也是对指令和数据独特的排列方式。

每一条代码或者数据只以位或者字节的形式出现。根据要执行的任务，系统需要每一个 ROM 地址上的字节。因此，机器可实现软件文件类似于每一个系统存储器地址上的地址和字节表。这个表必须作为一个目标硬件的 ROM 映像存在。图 1-5 给出了系统存储器中的 ROM 映像。这个映像包括引导程序、堆栈地址指针、程序计数器地址指针、应用任务、ISR(参见 4.12 节)、RTOS、输入数据和向量地址(详细介绍参见 2.5 节)。

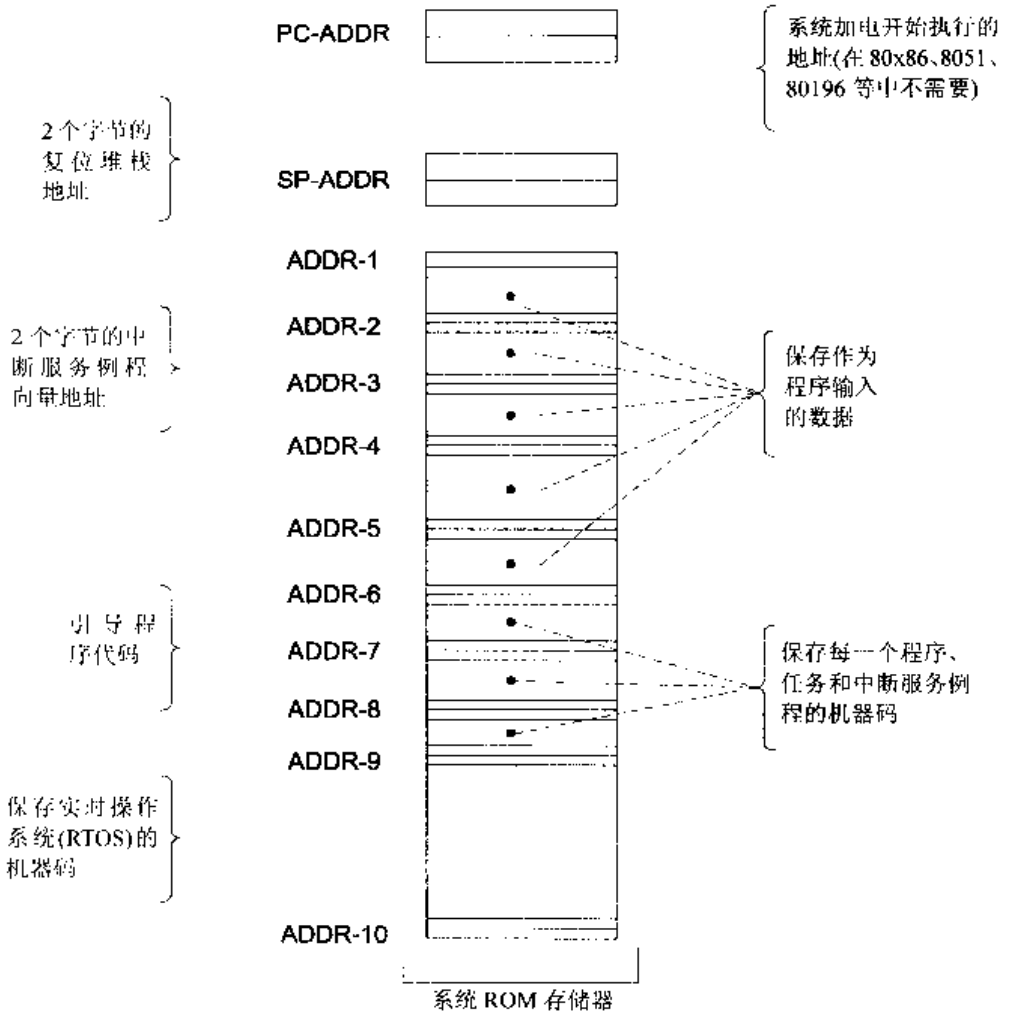


图 1-5　系统 ROM 存储器中嵌入的软件、RTOS、数据和向量地址

注意：

最终阶段软件也称为 ROM 映像。产品的最终机器可将软件嵌入在 ROM(或者 PROM)中，就像画框中的图像。为了创建 ROM 映像，必须定义每一个地址上的字节。通过修改这个映像，相同的硬件平台可以进行不同的操作，可以用于完全不同的应用或者同一个系统新的升级版本。

1.4.2　用机器码编写软件

在以这种形式进行编码的过程中，程序员定义地址以及每一个地址上相应的字节或者位。在配置某个物理设备或者子系统时，使用基于机器码的编码方式。例如，在收发器中，通过放入某种机器码或者位可以对其进行配置，使之以特定的 Mb/s 或者 Gbps，并使用一个特定的总线协议和网络协议进行信息传输。另外一个示例是使用某种代码来配置一个控制寄存器。在处理一个特定代码段的过程中，可以将寄存器配置为启用或者禁用其内部高速缓存。然而，只能够在特定的情形下用机器码进行编码：这是很浪费时间的，因为程序员必须首先熟悉处理器指令集，另外还要记住这些指令以及其机器码。

1.4.3 用特定于处理器的汇编语言编写软件

当程序员完全理解处理器以及其指令集后，就可以用汇编语言(assembly language)编写一个程序或者特定的一小部分程序。在 A.2 节将会给出用 ARM 处理器指令集编写的一个示例汇编语言程序。

用汇编语言编写代码对于已经学习过微处理器或者微控制器课程的设计者来说是很容易的。编码对于配置物理设备来说非常有用，例如端口、行显示接口、ADC 和 DAC 以及从一个缓冲区读取或者发送数据。这些代码还可以是设备驱动程序代码。(参见 4.1 节)。它们能够运用特定处理器或者设备的特性，并提供一种优化的编码方式。如果嵌入式系统设计团队不知道如何编写设备驱动代码，或者使用特定处理器特性激活代码的代码，会导致巨大的损失。嵌入式芯片厂商不仅对 API 收费，还会对公司生产的每一个系统收取知识产权费用。

但是如果用汇编语言编写所有的代码，会很浪费时间。完全用汇编语言编码只适用于少数简单的、小规模系统，例如玩具、巧克力自动售卖机、机器人或者数据采集系统。

图 1-6 给出了将汇编程序转换为机器码文件，最后得到一个 ROM 映像文件的过程。

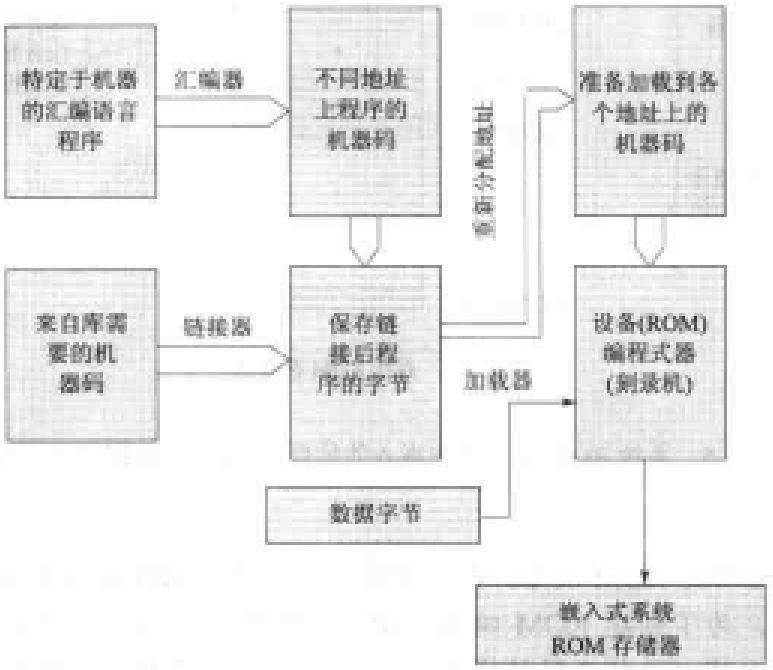


图 1-6 将汇编程序转换为机器码文件，最后得到一个 ROM 映像文件的过程

(1) 汇编器(assembler)经过一种称为汇编(assembling)的步骤，将汇编软件翻译为机器码。

(2) 下一个步骤称为链接(linking)，链接器(linker)将这些代码与其他必要的汇编代码链接在一起。由于有多组代码需要链接在一起，形成最后的二进制文件，因此链接是很有必要的。例如，如果汇编程序中有一个对延迟任务的引用，就可以由标准代码来完成这个任务。延迟代码必须与汇编代码相链接。延迟代码从某个地址开始是连续的。汇编软件代码从某个地址开始也是连续的。两段代码都必须处于不同的地址上，这些地址还必须是系统中的可用地址。链接器将这两者链接在一起。链接后要在机器上运行的二进制文件通常称为可执行文件或者简称“.exe”文件。链接之后，在将代码实际放入存储器中之前，必须重新分配代码序列的排放方式。

(3) 在下一个步骤中，当发现给定的立即数是一个物理 RAM 地址时，加载器(loader)执行

重新分配(reallocating)代码的任务。加载器是操作系统的一部分，读取.exe 文件之后将代码放置到存储器中。这个步骤是必要的，因为可用的存储地址可能不是从 0x0000 开始的，在运行过程中，二进制代码必须装载到不同的地址上。加载器找到适当的起始地址。在计算机中，可以使用加载器将准备运行的程序装载到 RAM 的一部分中。

(4) 系统设计过程的最后一步是将代码定位(locating)为 ROM 映像，并将其永久存放到 ROM 中实际可用的地址中。嵌入式系统不像计算机一样，有一个单独的程序可以跟踪运行过程中不同时间的可用地址。设计者必须定义用于加载的可用地址，并创建用于永久定位代码的文件。定位器(locator)程序将重新分配链接过的文件，并且以静态格式创建用于永久定位代码的文件。这种格式可以是 Intel Hex 文件格式或者 Motorola 的 S-record 格式(详细内容请参见附录 G)。

注意：

定位器在固定的地址上定位 I/O 任务和硬件设备驱动程序。这是因为对于一个给定的系统，这些代码的端口地址是不变的。

(5) 最后，或者是(i)在一个称为设备编程器的实验室系统，将 ROM 映像文件作为输入，并最终将这个映像烧写(burn)到 PROM 或者 EPROM 中，或者是(ii)在一家工厂里，将映像文件做成一个嵌入式系统 ROM 的掩膜(将代码放置到 PROM 或者 EPROM 的过程也称为烧写)。根据映像创建的掩膜使 ROM 成为 IC 芯片的形式。

注意：

为了配置某种特定物理设备或者类似于收发器的子系统，可以直接使用机器码。对于物理设备驱动程序代码或者使用处理器特定特性激活代码的代码，可以使用“特定处理器”的汇编语言。然后可以在 3 个步骤中，分别使用“汇编器”、“链接器”和“定位器”创建一个文件。文件具有标准格式的 ROM 映像。设备编程器最终将这个映像烧写到 PROM 或者 EPROM 中。从映像创建的掩膜使 ROM 成为 IC 芯片的形式。

1.4.4 用高级语言编写软件

在多数情况下，将所有的代码都用汇编语言编写是很浪费时间的。因此软件可以用高级语言开发，例如 C、C++或者 Java。多数情况下，优先考虑使用 C 语言(关于每种语言优点的介绍请参见 5.1 节和 5.9 节，关于 C 语言源代码编程工具的使用请参见 5.11 节)。仅仅为了编写代码的目的，很少需要了解汇编语言指令，程序员完全没有必要知道指令的机器码。程序员只需要知道硬件组织。例如，思考下面的问题：

计算 127、29 和 40 的和，并打印出平方根。

可以用于所有处理器的 C 语言程序示例如下：

```
(i)   #include<stdio.h>
(ii)  #include <math.h>
(iii) void main(void){
(iv)  int i1, i2, i3, a; float result;
(v)   i1=127; i2=29; i3=40; a=i1+i2+i3; result=sqrt(a);
(vi)  printf (result);}
```

很显然，求平方根的代码需要编写许多行，只能由熟练的汇编语言程序员完成。将这个程序使用高级语言编写比起使用汇编语言编写来说简单得多。C 程序有一个特征，当使用某种特

定处理器特性，并编写特定部分的代码时(例如端口设备驱动程序)，可以加入汇编指令。图 1-7 展示了一个典型的嵌入式 C 软件中的不同编程层(第 3~5 章有详细介绍)。这些层是：(i)处理器指令。(ii)主函数。(iii)中断服务例程。(iv)多任务，比如说从 1~N。(v)操作系统核心和调度程序。(vi)标准库函数，协议函数和任务分配函数。

图 1-7 嵌入式软件中的各种程序层

图 1-8 给出了将一个 C 程序转换为 ROM 映像文件的过程。编译器(compiler)产生目标代码。编译器根据处理器指令和其他说明对代码进行汇编。作为编译的最后一个步骤，嵌入式系统的 C 编译器必须使用代码优化器(code-optimizer)。优化器在链接之前，对代码进行优化。在编译之后，链接器(linker)将目标代码与其他必要的代码链接在一起。例如，链接器将某些函数代码包含进来，例如 printf 和 sqrt 代码。设备管理和驱动程序代码(设备控制代码)也是在这个阶段链接的：例如，打印机设备管理和驱动程序代码。链接之后，创建 ROM 映像文件的其他步骤与图 1-6 中所示的步骤相同。

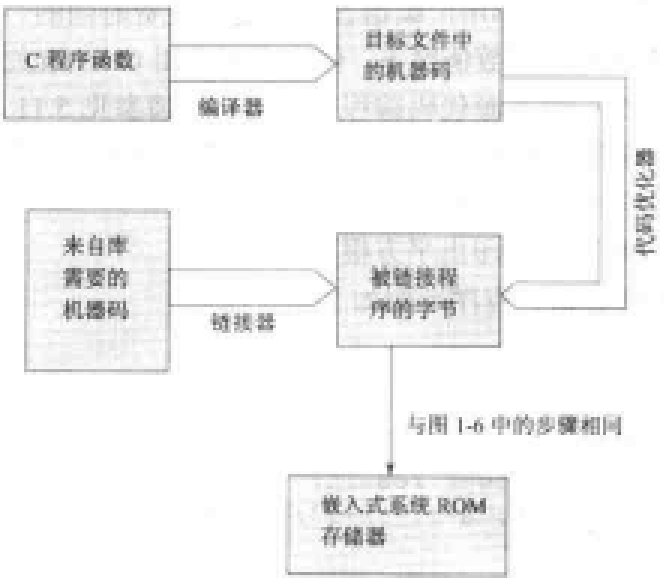


图 1-8 将 C 程序转换为 ROM 映像文件的过程

注意：

对于大多数系统来说，软件代码都是用 C 语言开发的。C 程序有多个层：处理器指令、主函数、任务函数和库函数、中断服务例程和操作系统核心(调度程序)。编译器产生目标文件。使用链接器和定位器，为目标硬件产生 ROM 映像文件。C++和 Java 是另外两种用来进行软件编程的语言。

1.4.5 使用操作系统的设备驱动程序和设备管理软件

在一个嵌入式系统中，有许多物理设备(physical device)，例如，键盘、显示器、磁盘、并口和网卡。

一种创新概念是在编程过程中使用虚拟设备(virtual device)。虚拟设备的例子可以是一个文件(用于读写字节流或者字流)或者是一个管道(缓冲字节流)。术语虚拟设备来自于一个类推，正如键盘在处理器进行读操作时提供处理器输入一样，文件也能提供处理器输入。处理器通过写操作向打印机输出数据。类似地，处理器也向文件中输出数据。通常，嵌入式系统需要实现多个功能，并且必须控制多个物理和虚拟设备。

一个用来控制、处理、读写操作的设备可以被认为具有三个部分：(i)控制寄存器或者字，用于存储数据位，通过设备驱动程序设置或者重置控制设备的行为。(ii)状态寄存器或者字，用于提供标志(位)，表示设备的状态。(iii)控制设备行为的设备机制。在一个设备中可能有输入数据缓冲区和输出数据缓冲区。设备的行为可以是将输入数据送到缓冲区中，或者从缓冲区中输出数据(控制寄存器、输入数据缓冲区、输出数据缓冲区和状态寄存器共同组成了设备的硬件部分)。

设备驱动程序(device driver)是控制、接收和发送出入设备的字节流或者字节流的软件。对于物理设备而言，驱动程序使用处于设置和重置状态下的硬件状态标志位和控制寄存器位。对于虚拟设备而言，设备驱动程序使用处于设置和重置状态下的状态和控制字以及位。

驱动程序控制着三种功能：(i)通过将控制寄存器或者字设置为特定的位，激活的初始化过程。(ii)在产生中断或者设置状态寄存器中的状态标志位时，调用 ISR，并运行(驱动)ISR(也称为中断处理例程)。(iii)中断服务完成后重置状态标志位。可以设计一个驱动程序用于异步操作(多个任务顺次多次使用)或者同步操作(任务同时使用)。这是因为当发生中断时，设备被激活，设备驱动例程为这个过程服务。

使用操作系统(OS)功能，所编写的设备驱动程序代码能够使得潜在的设备尽量隐藏。API 独立地定义硬件。这样就使得当系统中的设备硬件改变时，驱动程序仍然可用。

设备驱动程序在特定地址上访问并口或者串口、键盘、鼠标、磁盘、网络、显示器、文件、管道和插槽。OS 也可以为系统端口地址以及设备硬件访问机制(读、保存、写)提供设备驱动程序代码。

设备管理软件模块提供了测试设备是否存在的代码，以及初始化这些设备和测试已经存在设备的代码。模块也可以包含为各种设备在完全不同的地址上分配和登记端口(实际上可能是一个寄存器或者存储器)的软件，其中包括测试这些端口地址之间的冲突(如果存在的话)的代码。它确保了任何设备在任何一个瞬间只执行一项任务。它还考虑了虚拟设备的地址可能是已经被一个定位器(来自于 PROM)重新分配过了的情况(真正的物理或者硬件设备具有预先定义的固定地址(这些地址不能够被定位器重新分配))。

OS 还提供了管理与嵌入式系统相关设备的执行模块。潜在的原则是，在某一个瞬间，只

有一个物理设备能够接受设备输入或者向设备输出。OS 还提供并管理类似于管道和插槽的虚拟设备(参见 8.3 节)。

注意:

在设计嵌入式软件时，需要考虑两种类型的设备：物理的和虚拟的。物理设备包括小键盘、打印机或者显示矩阵。虚拟设备可以是一个文件(读写字节流或者字流)或者管道(缓存字节流)。系统中需要有设备驱动程序和设备管理软件。操作系统具有设备驱动和设备管理功能的模块。

1.4.6 多任务调度和使用 RTOS 设备的软件设计

通常，嵌入式系统都设计成在控制多个设备的同时，执行多个函数的调度。因此嵌入式系统程序就设计为类似于一个多任务系统程序(关于任务(函数)和任务状态的定义请参见 8.1 节)。

在一个多任务 OS 中，每个过程(任务)都有一个独特的存储器分配，并且一项任务对于特定的工作都具有一个或者多个函数或者过程。一个任务可以与其他任务共享存储器(数据)。一个处理器可以单独地或者并行地处理多个任务。OS 软件包含过程(任务、ISR 和设备驱动程序)的调度特征。OS 或者 RTOS 都具有一个操作系统内核(关于操作系统内核功能的详细介绍，参见 9.2 节)。操作系统内核的重要功能是安排任务从就绪状态向一个运行状态的转换时间。操作系统内核可以阻止一个任务，而让较高优先级的任务进入运行状态(这称为抢先调度)。操作系统内核调整在同一瞬间都处于就绪状态的任务对处理器的使用，确保只有一个任务能够进入运行状态。这是因为系统中只有一个处理器。操作系统内核将一个任务调度或者派遣到不同于当前状态的另外一个状态下(对于多处理器系统，还必须调度和同步各个处理器)。操作系统内核控制着内部进程(任务)通信和对变量、队列和管道的共享。

RTOS 函数非常复杂。第 9 章和第 11 章将介绍嵌入式微处理器中的 RTOS 函数。在一个嵌入式系统中，RTOS 必须是可裁剪的。可裁剪的(Scalable)OS 是一种将存储器优化、使其只具有与最终系统软件相关的部分特征的 OS。

现在已经出现了许多流行的并可以使用的 RTOS。第 9 章和第 10 章将对这些 RTOS 进行介绍。使用这些 RTOS 的案例研究将在第 11 章介绍。

注意:

嵌入式软件通常需要执行多种行为，并控制多个设备以及其 ISR。因此多任务软件是最基本的。为了调度多任务，通常使用流行的、现在已经可用的，并且具有函数(例如设备驱动程序)的 RTOS 内核。

1.4.7 设计嵌入式系统的软件工具

表 1-6 列出了汇编语言编程、高级语言编程、RTOS、调试的软件工具的应用，以及系统集成工具。

表 1-6 设计嵌入式系统的软件模块和工具

软 件 工 具	应 用
编辑器	使用键盘编写 C 代码或者汇编代码。允许输入、增加、删除、插入、附加前面编写过的代码行或者文件，在特定位置上合并记录和文件。创建源文件来存储编辑过的文件。它还有一个适当的名称(由程序员提供)
解释器	依次将每个表达式(逐行)翻译为机器可执行代码
编译器	使用完整的代码集。它可能还包含来自于库例程的代码、函数和表达式。它创建称为目标文件的文件
汇编器	将汇编代码翻译为二进制操作码(指令)，也就是翻译成称为二进制文件的可执行文件。它还产生可以打印的列表文件。列表文件具有地址、源代码(汇编语言代码)和十六进制的目标代码。文件具有汇编语言程序实际运行时调整的地址
交叉汇编器	将一个处理器的目标代码或者可执行代码转换为另外一个处理器的代码，或者相反。交叉汇编器将目标处理器的汇编代码汇编为系统开发中使用的 PC 中的处理器汇编代码。随后，它会提供目标处理器的目标代码。这些代码将是在最后的开发系统中实际需要的代码
模拟器	模拟嵌入式系统电路中的所有功能，包括附加的存储器和外设。它独立于特定的目标系统。它还会模拟当代码在特定的目标处理器上运行时将会执行的过程
源代码设计软件	用于源代码的理解、导航和浏览、编辑、调试、配置(禁用或者启用 C++特性)和编译
RTOS	参见第 9 章和第 10 章
软件示波器	用于动态跟踪任一程序变量的变化。它跟踪任一参数中的变化。演示执行的各种操作(任务、线程、服务例程)序列。它还记录整个时间历史
跟踪示波器	用于帮助跟踪在 X 轴时间上的模块和任务中的变化。行为列表还创建了各种任务需要的时间粒度和期望时间
集成开发环境	包含模拟器、编辑器、编译器、汇编器、RTOS、调试器、软件示波器、跟踪器、仿真器、逻辑分析器、EPROM、EEPROM 应用代码的刻录，用于系统的集成开发
原型[1]	用于模拟和源代码设计，包括编译和调试，以及在浏览器上总结开发阶段中最终目标系统的完整状态
定位器[2]	使用交叉汇编器输出和存储器分配映射，产生定位器输出。这是嵌入式系统软件设计过程的最后一步

1. 一个例子是来自于 WindRiver®的 Tornado Prototyper，它用于具有一组工具的集成交叉开发环境。
2. 关于定位器的描述参见 2.5 节。定位程序的输出是 Intel 十六进制文件或者 Motorola S-record 格式。

注意：

汇编器是用汇编语言编写的。对于高级语言编程，当设计复杂系统的时候，可能会需要一种特殊的源代码设计工具。在大多数的嵌入式系统中都需要 RTOS，它作为一个系统，在大多数情况下都需要调度多任务，来满足其最后期限；驱动几个物理和虚拟设备并处理许多 ISR。需要使用调试工具(例如软件示波器和跟踪示波器)进行调试。这是系统开发的一个很重要的步骤。系统软件和硬件的集成开发需要一种复杂的工具，也就是集成开发环境或者原型开发工具。

1.4.8 示例中需要的软件工具

表 1-7 列出了设计示例系统所需要的各种工具。

表 1-7 示例系统所需要的软件工具

软件工具	巧克力自动售卖机	数据采集系统	机器人	移动电话	汽车间距保持不变的自适应巡航控制系统	声音处理器
编辑器	是	是	是	NR	NR	NR
解释器	是	NR	是	NR	NR	NR
编译器	NR	是	否	是	是	是
汇编器	是	是	是	否	否	否
交叉汇编器	NR	是	否	否	否	否
定位器	是	是	是	是	是	是
模拟器	NR	是	是	是	是	是
源代码设计软件	NR	NR	NR	是	是	是
RTOS	MR	MR	MR	是	是	是
软件示波器	NR	NR	NR	是	是	是
跟踪示波器	NR	NR	NR	是	是	是
集成开发环境	NR	是	是	是	是	是
原型[1]	NR	否	否	是	是	是

1. 一个例子是用于具有一组工具的集成交叉开发环境的 Tornado 原型 WindRivera。

注意：NR 的意思是不需要。MR 的意思是在特定的复杂系统中可能会需要，但是不强制需要。关于定位器的描述，参见 2.5 节。

注意：

在大多数嵌入式系统中都需要 RTOS，因为系统必须调度多任务，驱动多个物理和虚拟设备，并处理大量 ISR。用于中型和复杂应用的嵌入式系统可能需要多个复杂的软件工具。

1.4.9 软件设计模型

在复杂系统或者多处理器系统中，在嵌入式系统软件及其 RTOS 的设计过程中，需要使用不同的模型。这些模型包括：(i)有限状态机(FSM)。(ii)Petri 网模型。(iii)控制和数据流图。(iv)基于 UML 模型的行为图。对于多处理器系统，还需要下列模型：(i)同步数据流(SDF)图，(ii)时间 Petri 网和扩展预测/转换网。(iii)多线程图(MTG)系统。这些模型将在第 6 章中介绍。

1.5　示例嵌入式系统

1.5.1　每种嵌入式系统的应用示例

嵌入式系统具有非常广泛的应用。其中几个应用领域是电信、智能卡、导弹和卫星、计算机网络、数字消费类电子、自动化。图 1-9 给出了嵌入式系统在这些领域中的应用。

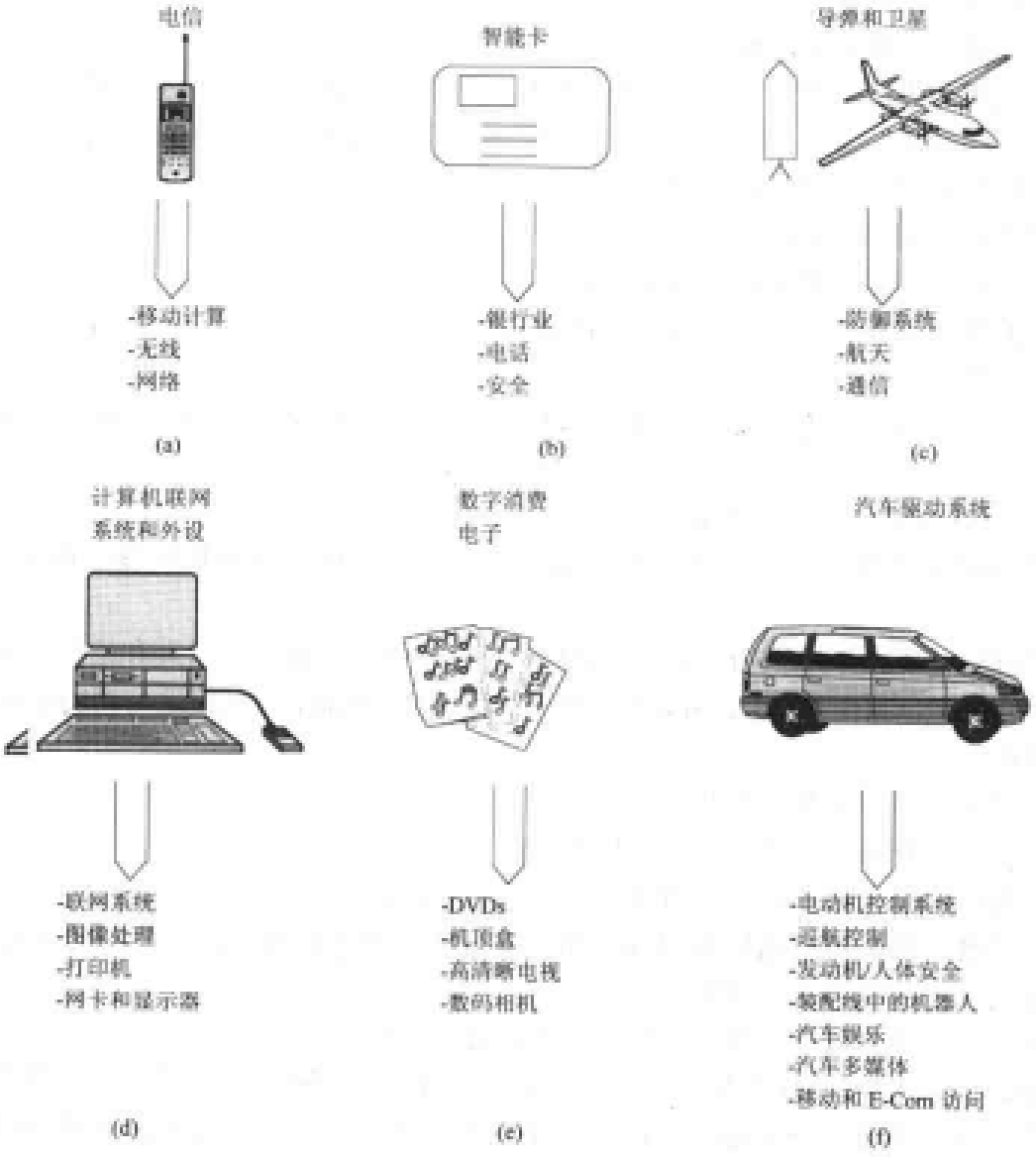


图 1-9　嵌入式系统在各个领域中的应用

小型嵌入式系统应用的几个示例如下所示：

- 巧克力自动售卖机
- 用于机器人系统的步行发动机控制器
- 洗涤和烹饪系统
- 用于显示电压、电流、电阻和频率的基于微控制器的单显示或者多显示数字板仪表
- 键盘控制器

- 串口卡
- 计算机鼠标
- CD 驱动和硬盘驱动控制器
- 计算机外设控制器，例如 CRT 显示控制器、键盘控制器、DRAM 控制器、DMA 控制器、打印机控制器、激光打印机控制器、LAN 控制器、磁盘驱动控制器
- 传真、影印、打印机或者扫描仪
- 数字日记
- 远程电视(控制器)
- 带有存储器、显示和其他复杂特性的电话
- 电动机控制系统，例如 d.c.电动机、机器人和 CNC 机的速度和位置的准确控制；自动应用，例如闭环发动机控制、动态行驶性能控制、防锁制动系统监测器
- 电子数据采集和监督控制系统
- 电子器械，例如工业过程控制器
- 电子智能重量显示系统和工业湿度记录器及控制器
- 信号波形、电表或者水表读取数字存储系统
- 光谱分析器
- 生物医学系统，例如 ECG LCD 显示及记录器。血细胞记录器及分析器、病人检测系统

中型嵌入式系统的示例如下所示：

- 计算机网络系统，如路由器、服务器、交换机、网桥、网络集线器及网关的前端处理器
- 对于网络设备，有大量应用系统：(i)分布式网络中的智能操作、管理和维护路由器(IOAMR)。(ii)邮件客户卡，用来保存电子邮件及个人地址，并智能地与调制解调器或服务器相连
- 娱乐系统，例如可视游戏和音乐系统
- 银行系统，例如银行 ATM 和信用卡交易
- 信号跟踪系统，例如自动信号跟踪器和目标跟踪器
- 通信系统，例如移动通信 SIM 卡，数字寻呼机，便携式电话，电缆电视终端，具有或者没有图形加速器的传真机
- 图像过滤、图像处理、模式识别机、语音处理和视频处理
- 将袖珍 PC 或者 PDA(个人数字助手)连接到汽车驾驶员移动电话或者无线接收器上的系统。系统连接到 Internet 或者电子邮件远程服务器上，或者连接到 ASP(应用服务供给器)上的远程计算机。这个系统形成了移动电子商务(mobile e-commerce)和移动计算的中心。
- 手持设备中使用构架缓冲区的个人信息管理器
- 瘦客户端(thin client)(瘦客户端提供了具有远程引导功能的无磁盘节点)。瘦客户端的应用程序从数个节点访问数据中心；在 Internet 实验室中，通过远程服务器访问 Internet 专线。
- 使用 ARM7/i386 多处理器和 32MB 闪存的嵌入式防火墙/路由器。负载平衡和以太网接口是它的另外两个重要功能。这些接口支持 PPP、TCP/IP 和 UDP 协议。
- DNA 序列和模式存储卡，以及 DNA 模式识别器

复杂嵌入式系统的示例如下所示：

- 用于无线 LAN 以及聚合技术设备的嵌入式系统
- 用于实时视频和音频或者多媒体处理系统的嵌入式系统
- 使用高速(大于 400MHz)、极高速(10Gbps)和大带宽的嵌入式接口和网络系统：路由器、LAN、交换机和网关、SAN(存储区域网络)、WAN(广域网)、视频、交互式视频和宽带 IPv6(Internet 协议第 6 版)Internet 及其他产品
- 安全产品及高速网络安全。G 比特速率的加密产品
- 用于正在开发的太空救生艇(NASA 的 X-38 计划)的嵌入式复杂系统。这个系统用于将来与 ISS(国际太空站)一起使用的救援救生艇上。在紧急情况时，它会将宇航员和其他人员从 ISS 带回到地球。只需要按一下按钮，这个救生艇就会从 ISS 发射，能够承受所有的气候/大气条件，并满足精确的时间限制。它还是一个容错系统。

1.6 嵌入式片上系统(SOC)和内部 VLSI 电路

近来，嵌入式系统正在被设计到单个的硅片上，称为片上系统(System on chip，SoC)。SoC 是嵌入式系统的一种新的设计创新。嵌入式处理器是 SoC VLSI 电路的一部分。SoC 可以与下列组件一起嵌入：多处理器、存储器、多标准源解决方案，所谓的 IP(知识产权)核以及其他逻辑和模拟单元。SoC 中还可以嵌入网络协议。还可以嵌入一个加密功能单元。它可以嵌入用于信号处理的离散余弦变换。还可以嵌入 FPGA(现场可编程门阵列)核(参见 1.6.5 节)。

对于有些应用，GPP(微控制器、微处理器或者 DSP)核可能是不够的。对于安全应用、killer 应用、智能卡、视频游戏、掌上电脑、便携式电话、移动 Internet、手持嵌入式系统，Gbps 收发器、每秒 G 比特的 LAN 系统和卫星或者导弹系统，我们在 VLSI 设计电路中需要特殊的处理单元，能够具有处理器的功能。这些特殊单元称为专用指令处理器(ASIP)。对于一个应用，芯片中可能同时需要可配置处理器(称为 FPGA 及 ASIP 处理器)和不可配置处理器(DSP、微处理器或者微控制器)。一种使用多 ASIP 的 killer 应用示例是高清晰电视信号处理(高清晰的意思是，信号处理的目标是无噪声、消除回音的转换，并在电视机显示屏上获得高清平面图像(1920×1020 像素)。便携式电话是另外一种 killer 应用(killer 应用指的是一种对于数百万用户都有用的应用)。

最近，已经设计出了具有 DNA 芯片功能的嵌入式 SoC。考虑一个具有大量门阵列的 FPGA。现在，使用 VLSI 设计技术，我们可以配置这些阵列，使其处理 SoC 上的特定任务。这样就产生了一个可以看作是 DNA 芯片的 SoC。每一组阵列都具有特定的、独特的 DNA 复杂结构。这些结构以及处理器都嵌入到 DNA 芯片中。

1.6.1 用于便携式电话的 SoC 示例

图 1-10 给出了一个集成了两个内部 ASIC、两个内部处理器(ASIP)、共享存储器和公共总线上的外设接口的 SoC。除了处理器、存储器、数字电路以及用于特殊应用的嵌入式软件之外，SoC 还可以包含模拟电路。这种 ASIC 嵌入式 SoC 的一种应用示例就是便携式电话。对其中的一个 ASIP 进行配置，用来处理编码和译码，另外一个用于音频压缩。一个 ASIC 用来拨号、调制、解调、接口键盘和多线 LCD 矩阵显示、存储数据输入以及从存储器中取回数据。ASIC

是使用 VLSI 设计工具设计的，设计中嵌入了 GPP 或者 ASIP 和模拟电路。该设计是使用电子设计辅助(EDA)工具完成的(设计 ASIC 数字电路时，需要使用“硬件描述语言(HDL)”)。

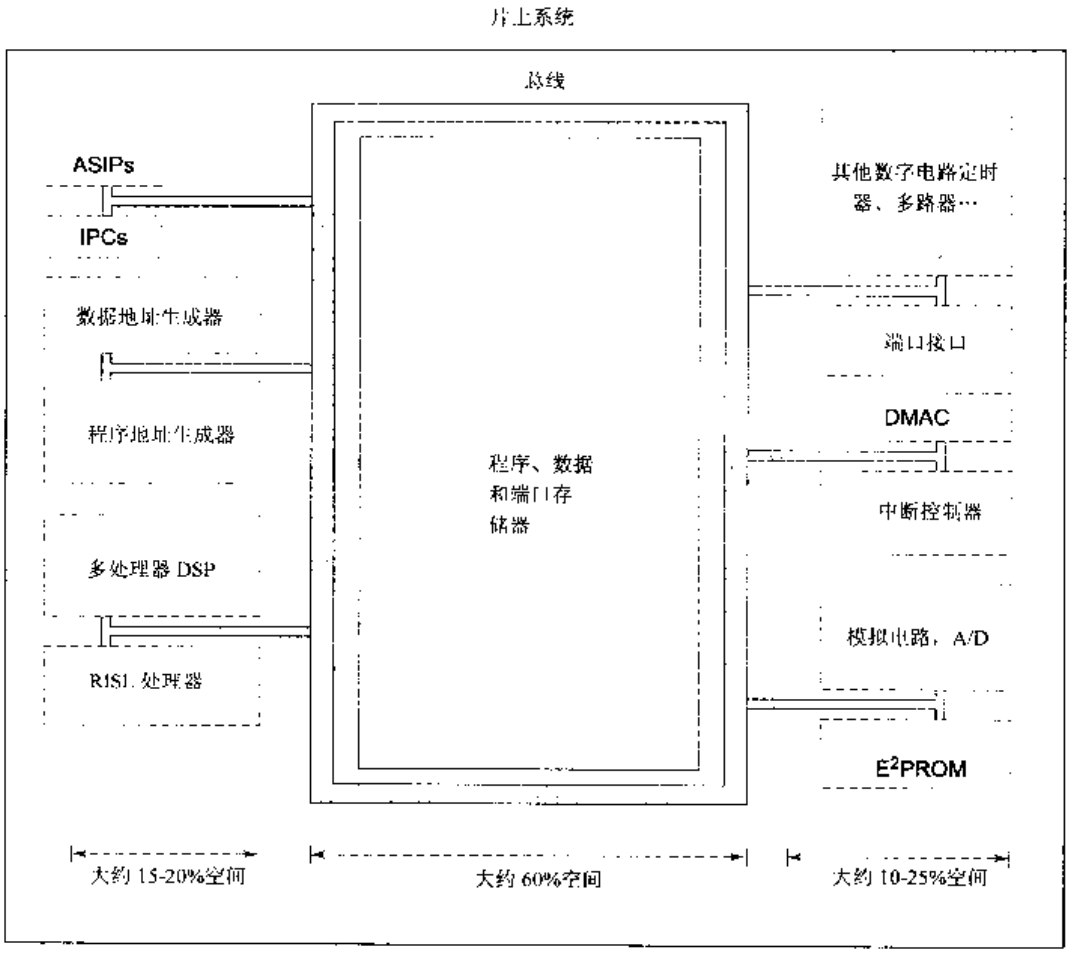


图 1-10　SoC 嵌入式系统及其公共总线，具有两个内部 ASIC、两个内部处理器、一些共享存储器以及外设接口

1.6.2　ASIP

使用 VLSI 工具可以设计处理器。专用系统处理器(ASIP)是一种不使用 GPP(标准的 CISC 或者 RISC 微处理器、微控制器或者信号处理器)的处理器。芯片上的处理器集成了一组 CISC 或者 RISC 指令集。这种特定的处理器可以有针对某个应用的可配置指令集。ASIP 也是可配置的。使用适当的工具，可以将 ASIP 设计并配置为下列示例功能所需要的指令：DSP 功能、控制信号处理器功能、自适应滤波功能和通信协议实现功能。在一个 VLSI 芯片上，特殊系统中的嵌入式 ASIP 可以是 ASIC 或者 SoC 中的一个单元。

1.6.3　IP 核

在 VLSI 芯片上，还有一些高级组件。这些组件具有很高的门级电路复杂性，这些电路比计数器、寄存器、多路器、浮点计算单元和 ALU 更复杂。一种标准的通过配置 FPGA 核或者 VLSI 芯片核来集成高级组件的解决方法，可能已经作为知识产权(IP)存在了。高级组件的 IP 门级实现设计的版权，是属于设计者或者设计公司的。使用每一个芯片都要付费。嵌入式系统

中可以集成 IP。

- IP 可以提供变换、加密算法或者解密算法的硬件可实现设计。
- IP 可以提供一个信号的自适应滤波设计。
- IP 可以提供实现超文本传输协议(HTTP)或者文件传输协议(FTP)的完整设计，用来在 Internet 上传送 web 页或者文件。
- 可以设计用于 PCI 或者 USB 总线控制器的 IP(参见 3.3 和 3.4 节)。

1.6.4 嵌入 GPP

通用处理器(GPP)可以嵌入到一个 VLSI 芯片上。最近，已经有了这样的 GPP 示例，即 ARM 7 和 ARM 9，可以嵌入到 VLSI 芯片中。这两款 GPP 是由 ARM 公司开发的，并由德州仪器(Texas Instrument)做了改进(请访问 http://www.ti.com./sc/docs/asic/mocules/arm7.htm 和 arm9.htm)。ARM 处理器既有作为 CPU 芯片的 VILSI 体系结构，也有用于集成到 VLSI 或者 SoC 中的 VILSI 体系结构(ARM 的指令集特征参见 A.1 节)。ARM 提供了 CISC 特征，但内核是 RISC 体系结构。ARM 嵌入电路的一个应用是 ICE(参见 12.3.2 节)。ICE 用来调试嵌入式系统。ARM 9 的应用有电视机机顶盒、有线调制解调器，以及无线设备，例如移动电话。

ARM 9(参见 2.2.5 节)具有一个单周期 16×32 的乘法累加单元。其操作频率为 200MHz。使用的是 0.15μs 的 GS30 CMOS 工艺。具有 5 级流水线。它是一种 RISC 体系结构。当设计一个具有多处理器体系结构的 ASIC 时，可以将其与 DSP 集成在一起(参见 6.3 节)。例如，可以将其与 DSP TMS320C55x 集成在一起。ARM 7 比 ARM 9 的处理能力差一些，但是也相当流行。其工作频率是 80MHz。使用的是 0.18μs 的 GS20 CMOS 工艺。现在已经有大量使用 ARM 7 的嵌入式系统出现。最新的一个应用是将操作系统 Linux 2.2 核和设备驱动程序集成到 ASIC 中。

1.6.5 具有一个或者多个处理器的 FPGA 核

一种新的创新技术是具有一个或者多个处理器的现场可编程门阵列(FPGA)核。例如，Xilinx Virtex-II Pro FPGA XC2VP125。另外一个示例是由 Xilinx 在 2003 年 4 月 14 日发布的 90 纳米 Spartan-3 FPGA(FPGA 在一个 VLSI 芯片上包含了大量的可编程门。在每一个 FPGA 单元中都有一个门，称为“宏单元”。每一个单元具有多个输入和输出端。所有的单元互连起来，像一个阵列(矩阵)。每一个连接都可以使用 FPGA 编程工具熔断(分离))。感兴趣的读者可以参考“Xcell Journel”2002 和“Xcell Journel”2003。

考虑下面的算法：SIMD 指令、傅立叶变换及其逆变换、DFT 或者拉普拉斯(Laplace)变换及其逆变换、压缩和解压缩、加密和解密、特定模式识别(识别一个信号、指纹或者 DNA 序列)。我们可以将一个算法配置(熔和)到 FPGA 的逻辑门中。它给出了一个处理单元的硬连线实现。它是专用于满足嵌入式系统需要的。可以将嵌入式软件的一个算法在 PFGA 的一个区域中实现，将另外一个算法在另一个区域中实现。

最新的 SoC 设计示例是 XC2VP125 系统。在 FPGA 核中总共有 125 136 个逻辑单元，有 4 个 IBM PowerPC。近来已经用于开发嵌入了可编程逻辑的嵌入式系统。例如已经报道了具有加密引擎的数据安全机制，数据频率为 1.5Gb/s。另外一些集成了逻辑阵列的嵌入式系统是 DSP 使能、实时视频处理系统和用于公用电话交换网(PSTN)的线性回音消除器以及包交换网(包是一个信息单元或者数据流，它可以在同时开放的多个可选择路径中选择一个可编程路径传输)。

1.6.6 示例 SoC 中的组成部分——智能卡

图 1-11 给出了一个非接触式智能卡 SoC 嵌入式系统的硬件组成部分。它的组成部分包括(11.4 节将描述一个嵌入式软件设计的案例研究):

- ASIP(专用指令处理器)
- 存储临时变量和堆栈的 RAM
- 应用程序代码以及用来调度任务的 RTOS 代码
- 存储用户数据、用户地址、用户标识码、卡号和过期日期的 EEPROM
- 定时器和中断控制器
- 约为 16MHz 的载波频率生成电路和振幅键控(ASK)调节器。对于位“1”，ASK 调节器给出多于正常 10%的载波脉冲，对于位“0”，给出小于正常 10%的载波脉冲。负载调制子载波是该频率的 1/16，使用二进位移相键控(BPSK)来调节位 1 和位 0。
- I/O 接口电路
- 向发送天线和系统电路传送电能的充电泵。充电泵从卡的天线接收 RF(射频)上获取电荷并存储(充电泵是一个很简单的电路，包含一个二极管和一个高数值、基于铁电材料的电容)。

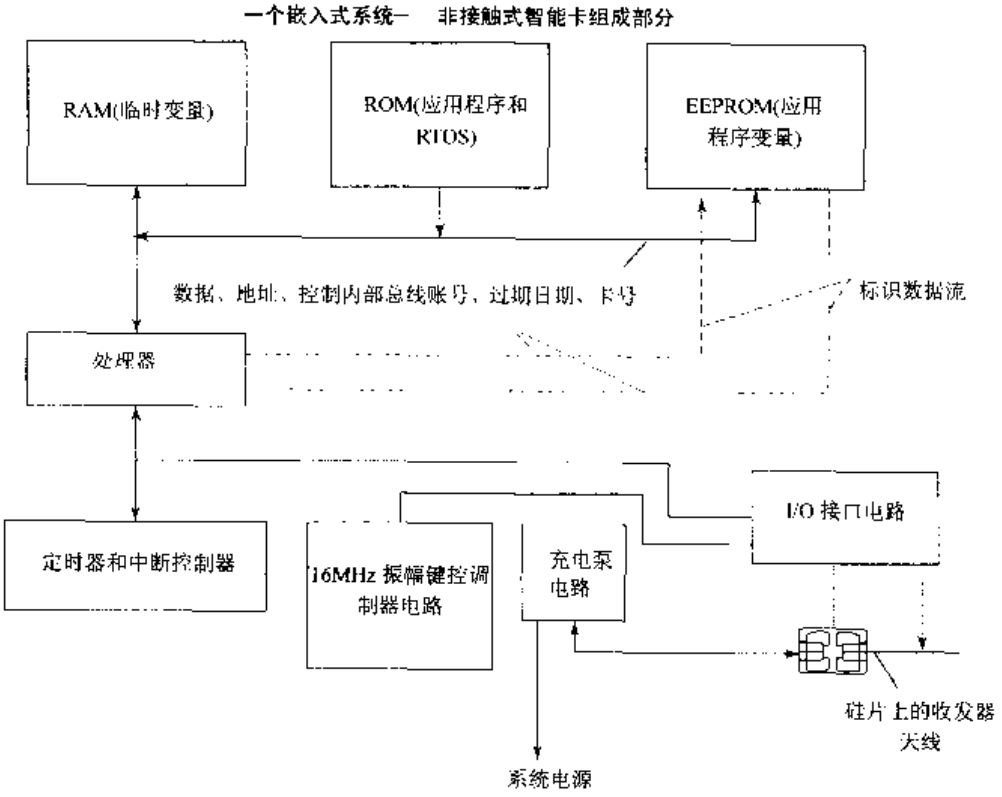


图 1-11 非接触式智能卡中的嵌入式系统硬件组成部分

本章小结

- 嵌入式系统是一种包含多种硬件和软件组成部分的复杂系统，并且它的设计比起 PC 以及在 PC 上运行的程序来说要复杂许多倍。

- 嵌入式系统处理器可以是从多个系列的处理器、微控制器、嵌入式处理器和数字信号处理器(DSP)中选择的一个通用处理器。此外，也可以设计一个专用指令处理器(ASIP)，用于 VLSI 芯片上的同一个特定应用。ASSP 可以附加地用于嵌入式软件某部分的快速硬件实现。复杂的嵌入式系统也可以使用多处理器单元。
- 嵌入式系统在 ROM 中嵌入(放置)了一个软件映像。映像通常包含下列部分：(i)引导程序。(ii)初始化数据。(iii)初始屏幕显示或者系统状态的字符串。(iv)系统执行的多任务程序。(v)RTOS 内核。
- 对于一个给定的硬件和软件，嵌入式系统需要一个电源和控制并优化功耗的单元，以满足给定硬件和软件总的能量需求。在某些嵌入式系统中，充电泵提供了一个无电源系统。
- 嵌入式系统需要时钟和复位电路。时钟管理器是最近出现的新技术。
- 嵌入式系统需要许多接口：输入输出(IO)端口、串行 UART 以及其他通过与外设接口接收输入以及发送输出的端口、显示单元、小键盘或者键盘。
- 嵌入式系统还可能需要总线控制器，用于将其总线与其他系统进行网络互连。
- 嵌入式系统需要定时器和 watchdog 定时器，用于系统时钟以及实时程序调度和控制。
- 嵌入式系统需要中断控制单元。
- 嵌入式系统可能需要 ADC 来获得来自一个或者多个源的模拟输入。它需要使用 PWM 的 DAC 向电动机、扬声器、语音系统等发送模拟输出。
- 嵌入式系统可能需要 LED 或者 LCD 显示单元、键盘和小键盘、脉冲拨号电路、调制解调器、发送器、多路器和信号分离器。
- 嵌入式软件通常是使用高级语言 C、C++或者 Java 编写的，需要加入、启用或者禁用某些编程特征。C 和 C++语言还帮助汇编语言代码的合并。
- 嵌入式系统通常需要一个用于实时编程和调度的实时操作系统、设备驱动程序、设备管理和多任务调度。
- 在嵌入式系统的设计阶段需要大量的软件工具(参见表 1-7)。
- 除了 ASIP 和 GPP 核之外，VLSI 芯片中还可以嵌入专用 IP。片上系统是新近的嵌入式系统概念，例如移动电话。非接触智能卡就是这样的一个应用，它的详细组成部分已经在图 1-11 中给出。

关键词及其定义

- 系统(system)：系统是一种按照固定的计划、程序或者规则进行工作、组织或者执行一项或者多项任务的方式。
- 嵌入式系统(embedded system)：是一个将计算机(其中嵌入了应用软件和 RTOS 的硬件)作为它的一个组成部分的复杂系统。嵌入式系统是专用于某个应用或者产品的基于计算机的系统。
- 处理器(processor)：处理器在给定一个命令(指令)之后，实现了一个或者多个进程。
- 进程(process)：一个具有独立内存分配，并且对于一个特定的工作具有一个或者多个函数的程序、任务或者线程(一个进程可以与其他任务共享内存数据)。一个处理器可以分别或者并发执行多个进程。

- 微控制器(microcontroller)：具有处理器的单元。根据应用的需求，提供存储器、定时器、watchdog 定时器、中断控制器、ADC 或者 PEM 等等。
- GPP(general-purpose processor，通用处理器)：一个来自几个系列的处理器、微控制器、嵌入式处理器和数字信号处理器(DSP)，具有通用指令集和立即可以使用的编译器，能够用高级语言编程。
- ASSP(Application Specific System Processor，专用系统处理器)：特定任务的处理器单元，例如图像压缩，它通过总线与主处理器一起集成在嵌入式系统中。
- ASIP(application specific Instruction porcessor，专用指令处理器)：专用于 VLSI 芯片上特定应用的处理器。
- FPGA：是芯片上的现场可编程门阵列。芯片中有大量的阵列，每一个元素都有可熔断的链接。阵列的每一个元素包含许多 XOR、AND、OR、多路器、信号分离器和三态门。通过对可熔断链接进行适当的编程，可以在芯片上构造一个复杂的数字电路设计。
- 寄存器(register)：与处理器相关，临时存储来自存储器以及在指令处理过程中执行单元的变量值。
- 时钟(clock)：由一个振荡电路产生的固定频率脉冲，控制处理过程中的所有操作以及系统中的所有定时参考。频率依赖于处理器电路的需求。如果一个处理器需要 100MHz 的时钟，则它的最小指令执行时间是这个值的倒数，也就是 10ns。
- 复位(reset)：是一个处理器状态，在这个状态中处理器寄存器获得初始值，从这个状态开始执行初始程序。这个程序也通常在加电时运行。
- 复位电路(reset circuit)：一个强制进入复位状态的电路，从加电的一个短周期内激活。当复位被激活后，处理器产生一个复位信号，给其他需要复位的系统单元。
- 存储器(memory)：它存储了所有的程序、输入和输出数据。处理器从存储器中取指令运行，并在每一条指令运行完后将结果传送给它。
- ROM：一个只读存储器，放置了下列数据—— 嵌入式软件、初始数据、字符串，以及操作系统或者 RTOS。
- RAM：是一个随机访问读写存储器，处理器用来存储可变的程序和数据，断电后存储器中的数据将消失。
- 高速缓存：处理器执行单元的快速读写片上单元。它存储了一页指令或者数据的副本。它提前从 ROM 或者 RAM 中取来这些指令或者数据，这样处理器就不用等待来自于外部总线的指令和数据。
- 定时器(timer)：一个提供系统时钟和实时操作、调度时间的单元。
- watchdog 定时器(watchdog timer)：当程序粘连超过一定时间后，该定时器定时结束，并将处理器复位。
- 中断控制器(interrupt controller)：控制来自于中断源的中断引起的处理器操作的单元。
- ADC：一个按照需要，根据参考电压将处于+和-极之间的模拟输入转换为 8 位、10 位或者 12 位的数字信号。
- PWM(Pulse Width Modulator，脉宽调制器)：给出脉宽与需要的模拟输出成比例的脉冲的脉宽调制器。结合了 PWM 输出后，就完成了 DAC 操作。
- DAC：将数字信号(8 位、10 位或者 12 位)转换为与参考电源成比例的模拟信号的转换器。

- 输入输出(IO)端口(Input Output port)：系统从这些端口获得输入或者发送输出。通过这些端口，将键盘或者 LCD 单元与系统连接起来。
- UART：通用异步收发器。
- LED：发光二极管，发出电压在 1.6~2V 之间、电流为 8~15mA 之间的红色、绿色、黄色或者红外线光的二极管。多段和多线 LED 单元用来高亮显示数字、字符、图表和短信息。
- LCD：液晶显示，在 3~4V、50~60Hz 的操作电压脉冲、电流最大为 50mA 的条件下吸收或者发出光。多段和多线 LCD 单元用来高亮显示数字、字符、图表和短信息，功耗很低。
- 调制解调器(modem)：将送出的比特调制为电话线中通常使用的脉冲，将输入的脉冲解调为输入信息的比特。
- 多路器(multiplexer)：一个数字电路，其输入来自于多个通道。一次只将一个通道的信息发送出去。输出端的通道地址与输入端的通道地址相同。
- 信号分离器(demultiplexer)：一个瞬间只向多个通道中的一个输出数字信号的数字电路。连接的通道地址与该电路的输入端的通道地址相同。
- 编译器(compiler)：一个按照处理器描述、从高级语言产生机器码的程序。这些机器码称为目标码。
- 汇编器(assembler)：将汇编语言软件翻译为放置在一个称为.exe(可执行)文件中的机器码。
- 链接器(linker)：一个将编译过的代码与其他代码链接起来，并为加载器或者定位器提供输入的程序。
- 加载器(loader)：是一个重新分配用于加载到系统 RAM 程序中的物理存储器地址的程序。重新分配是有必要的，因为在计算机处理过程中的给定的瞬间，可用的存储器地址可能并不是从 0x0000 开始的。加载器是计算机中 OS 的一部分。
- 定位器(locator)：在ROM存储器的实际地址中重新分配应用程序链接过的文件和RTOS代码的程序。它创建一个标准格式的文件。这个文件称为 ROM 映像。
- 设备编程器(device programmer)：它从定位器产生的一个文件中获得输入，并烧断可熔断链接，经数据和代码实际存储到 ROM 中。
- 掩膜和 ROM 掩膜(mask and ROM mask)：在工厂中为了制作芯片而创建的。ROM 掩膜是从 ROM 映像文件中产生的。
- 物理设备(physical device)：连接到系统端口的设备，例如打印机或者键盘。
- 虚拟设备(virtual device)：一个通过编程用来打开和关闭、读取和写的文件或者管道，例如连接和断接一个物理设备以及用作输入和输出的程序。
- 管道(pipe)：一种数据结构(或者虚拟设备)，发送来自于一个数据源(例如程序结构)的字节流，并将数据传递给数据目的端(例如打印机)。
- 文件(file)：一种数据结构(或者虚拟设备)，将记录(字符或者字)发送给数据目的端(例如程序结构)，并存储来自于数据源(例如程序结构)的数据。计算机中的文件还可以存储在硬盘中。

- 设备驱动程序(device driver)：中断服务例程软件，在外围设备(或者虚拟设备)的控制寄存器(或者字)编程之后运行，使设备获得输入或者输出。它在发生以设备为目的或者由设备发生的中断时执行。
- 设备管理器(device manager)：管理多设备和驱动程序的软件。
- 多任务调度(multitasking)：由调度程序指导，处理不同任务的代码。
- 内核(kernel)：一个具有存储器分配和解除分配、任务调度、内部进程通信、通过使用信号、异常(错误)处理信号、信号量、队列、邮箱、管道和插槽(参见 8.3 节)有效管理共享存储器的访问、I/O 管理、中断控制(处理器)、设备驱动程序和设备管理功能的程序。
- 实时操作系统(real-time operating system)：用于实时编程和调度、进程和存储器管理、设备驱动程序、设备管理和多任务调度的操作系统软件。
- VLSI 芯片(VLSI chip)：制作在硅片上的最多为 100 万晶体管的大规模集成电路。
- 片上系统(System on Chip)：VLSI 芯片上的一个系统，具有所有需要的数字电路和模拟电路，例如移动电话。

问题回顾

(1) 分别对系统和嵌入式系统给出定义。

(2) 下列器件的基本结构单元是什么？请一一加以列举。(a)微处理器(b)嵌入式处理器(c)微控制器(d)DSP(e)ASIP(f)ASIP

(3) DSP 与通用处理器(GPP)的主要区别是什么？参见 1.2.5 节和 D.2 节。

(4) (a)只有定点算术单元的处理器和(b)一个具有附加的浮点算术处理单元的处理器的优缺点各是什么？

(5) 媒体处理器是一个新的技术创新(参见 E.1 节)。参见 1.2.5 节、D.2 节和 E.1 节。媒体处理器与 DSP 的区别是什么？

(6) 解释收敛技术嵌入式系统(例如具有邮件客户、Internet 连接和图像框架下载的移动电话)中媒体处理器的使用。

(7) 比较下列每一种示例系列芯片(或者核)的特征：微处理器、微控制器、RISC 处理器、数字信号处理器、ASSP、视频处理器核媒体处理器。参见 1.2 节和附录 A~E。

(8) 为什么新一代的系统能够在低电压下(<2V)操作处理器和 IO(最大为 3.3V)？

(9) 系统中能量和电能管理的技术是什么？

(10) 在执行某些特定指令时使处理器降频运行，在执行其他指令时使处理器全速运行，这样做的好处是什么？

(11) 下面一些技术或指令的优点是什么？(a)Stop 指令(b)Wait 指令(c)处理器空闲模式操作(d)禁止使用高速缓存的指令(e)在嵌入式系统中启用具有多路线和多块的指令。

(12) 充电泵的意思是什么？在不使用电源的情况下，嵌入式系统中的充电泵是如何提供电源的？

(13) “实时”和“实时时钟”的意思是什么？

(14) 处理器复位和系统复位的任务是什么？

(15) 解释监视时间结束后 watchdog 定时器和复位的需要。

(16) 嵌入式系统中 RAM 的作用是什么？

(17) 为什么对于嵌入式系统的设备，我们需要多种行为和多种控制任务？用一个嵌入式系统示例——远程彩色电视机——来加以解释。

(18) 何时需要多任务调度 OS?

(19) 何时需要 RTOS?

(20) 为什么嵌入式系统 RTOS 应该是可伸缩的？

(21) 解释术语 IP 核、FPGA、CPLD、PLA 和 PAL。

(22) 片上系统(SoC)的意思是什么？嵌入式系统的定义应该根据片上系统做哪些改动？

(23) 用 FPGA 设计嵌入式系统的优点是什么？

(24) 用 ASIC 设计嵌入式系统的优点是什么？

(25) 用 ASIP 设计嵌入式系统的优点是什么？

(26) 实时视频处理需要严格实时限制的复杂嵌入式系统。为什么？请加以解释。

(27) 为什么处理器系统总是需要“中断处理器(中断控制器)”？

(28) 链接器所起的作用是什么？

(29) 为什么在计算机系统中使用加载器，在嵌入式系统中使用定位器？

(30) 为什么在嵌入式系统中程序存在于 ROM 中？

(31) 定义 ROM 映像，并解释一个示例系统中 ROM 映像的每一部分。

(32) 何时会使用 ROM 中的压缩程序和数据？给出 5 个将压缩程序和数据放置在 ROM 映像中的嵌入式系统示例。

(33) 何时会使用 SRAM 和 DRAM？请解释思的答案。

(34) 下列术语的意思是什么：物理设备，虚拟设备，即插即用设备，总线自驱动设备，设备管理和专用设备处理器。

实践练习

(35) 后文从“参考文献”部分中给出的书中寻找嵌入式系统的定义，并将定义和参考书列成表格，定义放在第一列，参考书放在第二列。

(36) 将嵌入式系统分类为小型、中型和复杂系统。现在，再将嵌入式系统重新分类为具有和没有实时(反映时间受限)要求的系统，并对每一类给出 10 个示例。

(37) 汽车巡航系统将被设计到一个项目中。设计团队中硬件和软件工程师都需要具备哪些技能？

(38) 给定一个数值，x=1.7320508075688。首先由浮点算术处理单元计算它的平方。现在 x 被一个 16 位的整数定点算术处理单元计算它的平方。这两个结果有什么不同(注意：定点单元将只计算 17320 乘以 17320，然后用 10000 除以结果两次)？

(39) 设计一个具有四列的表，两个嵌入式系统示例分别在每一行的第 2 列和第 3 列。第 1 列：下列所需要的处理器类型：微处理器，微控制器，嵌入式处理器，数字信号处理器，ASSP，视频处理器和媒体处理器。在第 4 列中给出理由。

(40) 为什么 CMOS IO 电路的功耗在使用 3.3V 的操作电压时，比使用 5V 的操作电压要降低近一半(约为$(3.3/5)^2$)？

(41) 当操作电压从 5V 降低到 1.8V 时，微处理器 CMOS 电路的功耗会降低多少？

(42) 参见 1.3.5 节以及 G.1~G.3 节。列出各种类型的存储器以及它们在下列系统中的应用：

机器人，电子智能重量显示系统，EGC LCD 显示及记录器，路由器，数码相机，语音处理，智能卡，嵌入式防火墙/路由器，具有冲突控制和超时传输控制(jabber)的收发系统(冲突的意思是当网络中其他系统都不使用网络时的接收和发送。超时传输控制控制的意思是控制网络中的连续的随机数据流，这些数据流最终会阻塞网络)。

(43) 以表格的形式列出在问题(42)中所提到的每一个系统所需要的硬件单元。

(44) 给出嵌入式系统的两个示例，它们需要一个或者多个下列的单元：(a)DAC(使用 PWM) (b)ADC (c)LCD 显示 (d)LED 显示 (e)小键盘 (f)脉冲拨号电路 (g)调制解调器 (h)收发器 (i)GPIB(IEEE 488)链接。

(45) ADC 一定是 10 位的 ADC 吗？它的参考电压为 V_{ref-}=0.0V，V_{ref+}=1.023V，当输入为(a)－0.512V；(b)+0.512V；(c)+2.047V 时，ADC 的输出分别是什么？在下列三种情况下，ADC 的输出分别是什么：(i)Vref-=0.512V，Vref＋=1.023；V (ii)Vref-=1.024V，Vref＋=2.047V；(iii)Vref-=－1.024V，V_{ref+}=+2.047V。

(46) 参见 1.4、1.5、5.8 和 5.9 节。以表格的形式列出使用下列编码语言的优点和缺点：(a)最终机器可实现的语言；(b)ALP(汇编语言编程)；(c)C；(d)C++；(e)Java。

(47) 列出问题(42)中所提到的每一个嵌入式系统所需要的软件工具。

(48) 论证嵌入式系统中设备驱动程序的重要性。参见 1.4.5 节和 4.1.3 节。

(49) 设计嵌入式系统的工作可能要比它的处理器和硬件单元的工作高数千倍。请说明这句话的含义。

(50) 将 FPGA(现场可编程门阵列)和单个或者多个处理器单元集成到芯片上的 FPSLIC(现场可编程系统级集成电路)是最新的技术创新。它们如何帮助设计用于实时视频处理的复杂嵌入式系统？

第 2 章　处理器和存储器组织

本章前所学内容

前面一章介绍了下列内容：

(1) 嵌入式系统是一个基于计算机的专用系统，或者是大型应用系统中的一部分。它的处理器可以是具有 CISC 或者 RISC 体系结构的微处理器，也可以是一个微控制器或者数字信号处理器(DSP)。

(2) 嵌入式系统的应用领域很广，具有系统专用的软件和多个硬件单元。

(3) 处理器和存储器是系统中的两个重要组成部分。处理器在各种存储器单元(ROM、RAM 和高速缓存)的支持下来运行软件。

(4) 系统软件嵌入在 ROM 中，称为 ROM 映像。这个映像中包括引导程序、初始化数据、初始屏幕显示或者系统状态的字符串、系统执行的多任务程序以及 RTOS 内核。

(5) ROM 中存储的 ROM 映像包含最终设计的代码，RAM 中存储的是程序执行过程中变量和堆栈的临时值。高速缓存提前存储来自于外部存储器的指令和数据副本，并在快速处理过程中临时存储结果。

系统硬件设计者的一个很重要的问题是，如何选择合适的微处理器、微控制器或者 DSP，使它们在适当存储器设备的协助下表现出最优的性能。设计者还必须指定(i)如何组织所选择的处理器和存储器；(ii)如何设计一个适当的接口电路，将处理器、存储器、I/O 设备和系统中的其他单元连接起来。因此，系统设计者必须完全掌握基本结构单元、存储器、存储器分配映像、总线信号和序列，以及处理器中所有这些单元的运行速度。本章将介绍这些内容。

系统中处理器和存储器的组织提供了获得良好性能的基本平台。设计的目标是，在目标嵌入式系统存储器的最佳使用和最小功耗的条件下，取得最好的处理器性能。

本章将学内容

本章中我们将学习下列内容：

(1) 处理器的结构单元

a. 内部总线以及将处理器与系统存储器、I/O 设备和所有其他系统单元连接起来的外部总线

b. 通过快速的程序执行来改进处理器计算性能的超标量、进程、流水线和高速缓存单元

c. 使用 ALU 处理整数数据的算术与逻辑操作，以及使用处理单元进行浮点数的浮点运算

d. 某些嵌入式处理器中的原子操作处理单元，用来解决多任务操作过程中的共享数据问题

e. 处理器寄存器、寄存器窗口和寄存器文件

(2) 各种类型的存储器设备及其作用
(3) 用来保存数据结构和数据集合元素的存储器块
(4) 存储器映射的概念
(5) 设备寄存器和 I/O 设备地址
(6) 通过设计一个定位程序，将 ROM 映像作为单个存储器映射来定位
(7) 能够使系统设备直接访问系统存储器的直接存储器访问特征
(8) 存储器和 I/O 设备与处理器的接口电路

2.1 处理器中的结构单元

表 2-1 中列出了通用存储器中的结构单元以及每一个单元的功能。图 2-1 给出了将一个处理器的 25 个结构单元进行连接的示意图。

表 2-1 处理器中的结构单元及功能

结 构 单 元	功 能
MAR(存储器地址寄存器)	它保存将要从外部存储器取来的字节或者字的地址。处理器在开始一个取周期之前，先把指令或者数据的地址发送给 MAR
MDR(存储器数据寄存器)	它保存从(或者要发送到)外部存储器或者 I/O 地址取来的字节或者字
内部总线	将处理器的所有结构单元内部相连。它的宽度可以是 8、16、32 或者 64 位
地址总线	这是一个外部总线，将地址从 MAR 发送到存储器、IO 设备以及系统的其他单元
数据总线	是一个在读写操作过程中从一个地址中取来(或者发送到)指令或者数据字节的外部总线
控制总线	在处理器和存储器(或者设备)之间传送控制信号的外部总线
BIU(总线接口单元)	它是外部总线与处理器内部单元之间的接口单元
IR(指令寄存器)	它连续地将指令码(操作码)发送给处理器的执行单元
ID(指令译码器)	它翻译 IR 接收到的指令操作码，并将它发送给处理器 CU
CU(控制单元)	它控制着处理过程所需的所有总线行为和单元功能
ARS(应用寄存器组)	是一组在处理用户应用程序的指令过程中使用的片上寄存器。寄存器窗口包含寄存器的一个子集，每一个子集包含一个软件例程的静态变量。寄存器文件是与单元(例如 ALU 或者 FLPU)相结合的文件
ALU(算术逻辑单元)	根据 IR 中出现的当前指令，执行算术或者逻辑指令的单元
PC(程序计数器)	它产生了通过 MAR 从存储器获取地址的指令周期。当有规律地连续取指时，它的值自动增加。在 80x86 处理器中称为指令指针
SRS(系统寄存器组)	在处理监督系统程序的指令过程中使用的一组寄存器
SP(堆栈指针)	它是一个地址指针，与存储器的栈顶对应

(续表)

结 构 单 元	功　　能
IQ(指令队列)	它是一个指令队列，作用是使得当一条指令执行完后，IR 没有必要等待下一条指令
PFCU(预取控制单元)	这是一个控制将数据提前从存储器中取到 I-缓存或者 D-缓存中的单元。当处理器的执行单元需要指令和数据时，将指令和数据传送过去。处理器没有必要在执行指令之前才取数据
I-缓存(指令缓存)	它连续地将指令以 FIFO 的模式保存，就像一个指令队列。它使得处理器能够通过使用 PFCU 高速执行指令；处理器访问外部系统存储器的速度相对要慢很多
BT 缓存（分支目标缓存)	当遇到分支指令时，例如 jump、loop 或者 call，它协助准备好要执行的下一个指令集。它的取单元能够预见 I-缓存中的分支指令
D-缓存(数据缓存)	它保存从外部存储器预取来的数据。数据缓存通常将键(地址)和数值(字)共同保存在一个位置上。如果进行适当的配置，它还可以保存写穿透数据。写穿透数据意味着从执行单元获得执行结果，并将这个结果从缓存传回到相应的外部存储器地址中
MMU(存储管理单元)	它是管理存储器[2]的单元，负责执行过程所需要的指令或者数据(参见下面的注意 2)
FLPU(浮点处理单元)	一个从 ALU 分离出来的单元，用于浮点处理，这是在微处理器或者 DSP 中快速执行数学函数的基本单元
FRS(浮点寄存器组)	专门用来以标准格式保存浮点数的寄存器组，FLPU 使用它来保存数据和堆栈
高级处理单元	这些单元用于多级流水处理、多路超标量处理，使得处理速度达到每个时钟周期能执行多条指令[3]。还有一个 MAC[4] 单元，用于计算过程中系数的相乘和累加
AOU(原子操作单元)	它使得当一个用户(编译器)指令分割成几个叫做原子操作的处理器指令后，在发生过程中断之前结束该指令的操作。这样就避免了各个例程和任务之间发生共享数据的问题

1. 当处理器并行运行多任务时，这实际上帮助完成了从一个任务到另外一个任务的快速切换。(参见 4.6 节了解上下文切换以及它的必要性。)

2. MMU(存储器管理单元)管理 RAM 存储器中的页，以及这些页在外部和内部高速缓存中的副本。管理必须做到当指令执行时，产生的页和高速缓存失败(失效)要尽量少。

3. 由于高级处理单元的作用，指令周期时间比时钟周期要短得多。

4. 在 DSP 中总是需要 MAC 单元 (参见文中所述)。

5. 高性能处理器中的单元用斜体显示。

几乎所有的处理器都具有图 2-1 中实线框中的结构单元。高性能处理器在图 2-1 中用虚线框中的结构单元表示(通常，性能是通过 MIPS(百万指令/秒)或者 MFLOPS(百万浮点指令操作/秒)或者 Dhrystone/s 来度量的。参见附录 B)。

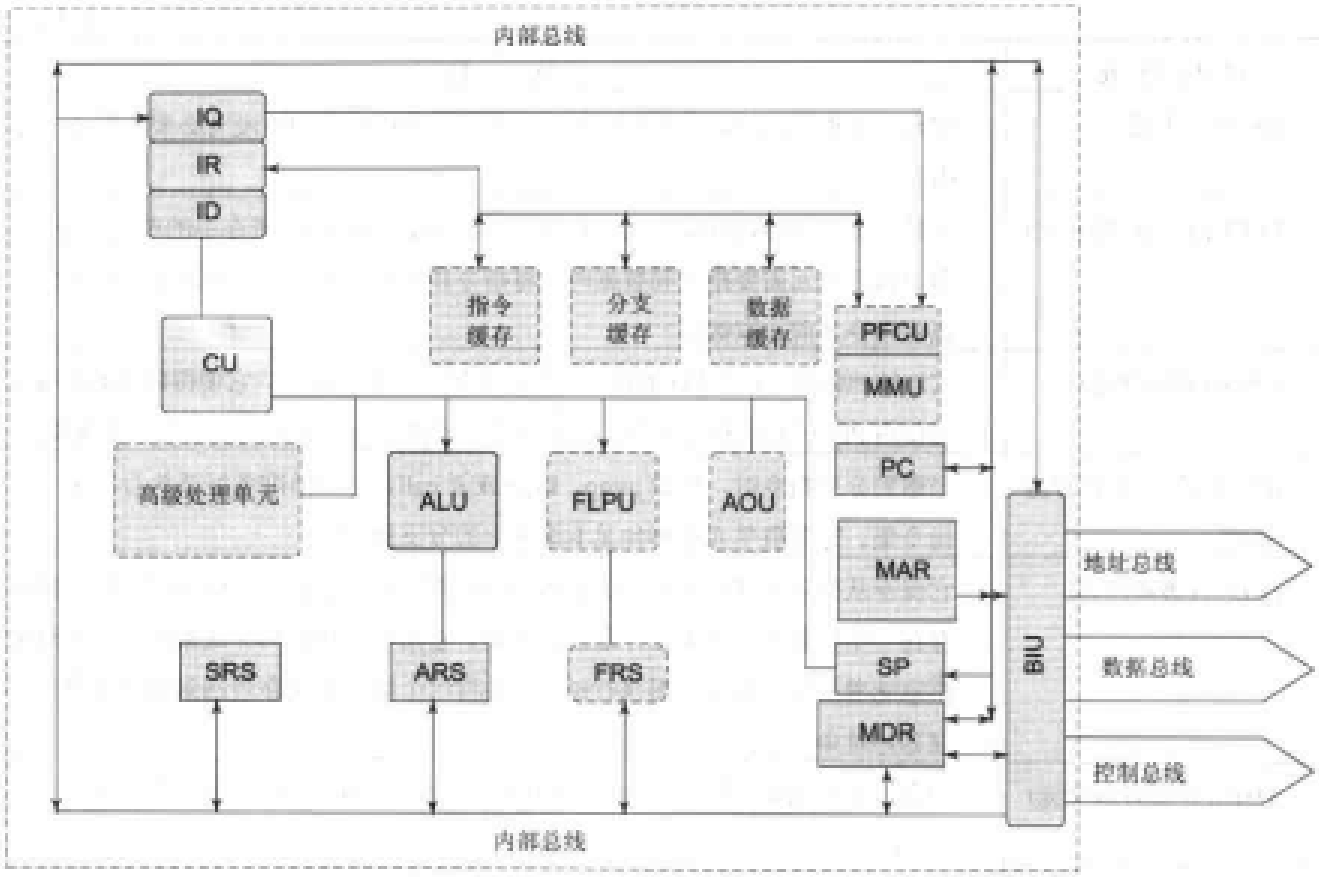


图 2-1 嵌入式系统处理器中的结构单元示意图(虚线框中的单元只在高性能处理器中才有)

高级处理单元包括指令流水线单元，它通过在多个级中处理指令从而提高性能，此外还包括超标量处理单元，它通过并行执行一个或者多个指令来提高性能(参见由 Daniel Tabak 编著，McGraw-Hill 在 1995 年出版的 *Advanced Microprocessors*)。

预取单元通过提前取指令和取数据来提高性能。高速缓存和 MMU 通过向处理器执行单元快速提供指令和数据来提高性能。

RISC 体系结构通过多种方式提高性能：在单个时钟周期内执行指令(指令的硬连线实现)；使用多个寄存器组、窗口和文件；由于减少了寻址方式而降低数据对外部存储器访问的依赖性。

使用浮点单元和 FRS 比使用处理整数的 ALU 来处理数学函数的速度更快。

DSP 中的 MAC(乘累加)单元能够快速地将两个操作数相乘，并在一个地址上将结果累加。它能够快速计算表达式，例如下面的求和表达式。$y_n=\Sigma(a_i\times x_{n-i})$，其中 n=0，1，2，…，N-1。并且 i、n 和 N 是整数，a 是一个系数，x 是一个独立的变量或者输入值，y 是结果变量或者输出元素。

流水线和超标量单元是如何提高性能的呢？让我们来看一下示例 2-1。

示例 2-1 流水线和超标量执行

步骤 1

假设处理器指令周期是 0.02ms(频率为 50MHz)，并且处理器在一个时钟周期内执行一条指令。如果不考虑其他高级处理单元，处理器的性能预期为 50MIPS。

步骤 2

假设处理器中有一个三级流水线，如 ARM7 中的流水线。我们暂时忽略分支的影响(称为分支代价)。三条指令将在一个时钟周期内处理完。不考虑超标量处理单元，只考虑流水线，处理器的最大预期性能为 150MIPS。

步骤 3

假设有一个两路的超标量。先忽略未调整数据的影响(称为数据依赖性代价)。当具有三级流水线、两路超标量单元时，6 条指令将在一个时钟周期内执行完。现在最高的性能是处理器周期的 6 倍，也就是 300MIPS。

(1) 在这个示例中用到了一个术语——分支代价(branch penalty)。如果在一个多级流水线中碰到一条分支指令，则在流水线前面几级中执行的指令就是冗余的。当循环结束或者从例程返回后，这些指令必须重新执行。重新执行这些指令所需要的时间就称为分支代价。

(2) 在这个示例中使用的另外一个术语是数据依赖性代价(data dependency penalty)。假设在一个超标量操作中，有两个指令分别在两路中执行。并且一条指令依赖于另外一条指令的数据结果。这就是所谓的不对准。因此，在将两条指令分别放置在不同的路中之前，它们是不对准的。此时，一条指令必须等待，直到另外一条指令执行完后才能执行。等待的时间就是数据依赖性代价。

下面介绍的是处理器结构的基本特征。每一个系统设计者在选择处理器和设计硬件的过程中，都应该考虑这些特征。

(1) 指令周期时间(instruction cycle time)：这是处理器执行一条指令所需要的时间，对于工作频率约为 12MHz 的 8051 处理器来说，这个值约为 1μs；对于 PowerPC MPC 604 处理器，这个值为 1.6ns。系统设计者将指令周期时间作为处理器速度与应用程序匹配的指示器。对于需要快速处理的应用程序，PowerPC MPC 604 处理器比较适合；对于那些处理速度较慢也能满足要求的应用程序，可以选择 8051、68HC11 或者 80196 处理器。

(2) 内部总线宽度(internal bus width)：ALU 从总线上获得输入。ALU(在一次数学或者逻辑操作中)的一个操作数的位数等于总线宽度。32 位的总线能够满足在一个周期中对 32 位的操作数进行数学操作的要求。对于信号处理和控制系统指令，32 位总线是必要的。当总线宽度为 32 位时，其读写的整数宽度就为 32 位，且执行速度比总线宽度为 8 时的速度快大约 4 倍。PowerPC MPC 604 处理器和奔腾处理器的内部总线宽度为 64 位。

(3) CISC 或者 RISC 体系结构：CISC 或者 RISC 体系结构会影响系统设计。CISC 具有用较少的寄存器处理复杂指令和复杂数据集的能力。RISC 执行较简单的指令，每条指令只占用一个周期(参见附录 A，了解 CISC 和 RISC 体系结构，以及作为示例的 RISC ARM 7 和其指令集)。具有 RISC 实现的 CISC 的意思是：对于大多数指令而言，都具有在单个指令周期中执行的硬连线实现(示例有 ARM、80960 和奔腾处理器)。

(4) 程序计数器(PC)及其复位值：PC 的位数确定了处理器可以访问的物理内存的最大空间。复位值告诉设计者，在运行系统复位和加电的程序应该存放在什么地方，处理器将从此开始执行(最初的指令指针和代码段寄存器位确定了 80x86 处理器中最初的程序存储器地址)。

(5) 堆栈指针(stack pointer)及其初始复位值：堆栈指针中必须保存存储在堆栈中字的地址。

这些地址必须在系统分配给堆栈使用的地址范围之内。软件设计者定义初始复位值，并由此设置初始化程序中过程的堆栈指针。硬件设计者在两种处理器之间进行选择，一种使用外部存储器作为堆栈，另外一种使用内部堆栈。

(6) 流水线(pipeline)和超标量(superscaler)单元：许多情况下都需要较高的处理器性能。例如，实时信号处理。流水线和超标量操作现在已经成为必要的功能(参见示例 2-1)。硬件设计者必须根据需要的 MIPS 或者 MFLOPS 性能来选择处理器(对于需要很高性能的应用，需要使用多处理器系统(参见 6.3 节))。

(7) 片上存储器作为 RAM 和/或者寄存器文件、窗口、高速缓存和 ROM：这些是系统中存储器组织的组成部分。软件设计者必须通过执行适当的指令来启用处理器中的高速缓存，使得在运行程序中某一部分时获得更高的性能；而同时又禁止程序中其余部分使用高速缓存，以降低功耗并将系统的能量需求最小化。硬件设计者应该选择一个具有多路缓存单元的处理器，这样就能够只激活一个高速缓存，其中包含执行指令的一个子集时所需要的数据。这样还能降低功耗。

(8) 外部中断(external interrupt)：在处理器中有几个管脚，外部电路可以通过这些管脚发送中断信号。在 4.1 节中将介绍外部中断的使用。

(9) 中断控制器(interrupt controller)：处理器可以包含一个内部中断处理器，对服务程序优先级进行编程，并分配向量地址。内部中断处理器在大多数应用程序中都有很大的帮助。

(10) 位操作指令(bit manipulation instruction)：这些指令有助于简化端口和存储器地址中数据的处理。

(11) 浮点处理器(float point processor)：包含 FLPU 和 FRS 单元的处理器能够快速执行浮点运算。它们能够提高处理器的计算能力，对于信号处理和复杂的控制应用来说，浮点处理器是必需的。

(12) 多通道直接存储器访问(direct memory access，DMA)控制器：当有多个 I/O 设备，并且一个 I/O 设备需要快速访问一个多字节数据集时，系统存储器片上 DMA 控制器会有很大的帮助。第 2.6 节将对其进行介绍。

(13) 片上 MMU：当使用高速缓存的时候需要这个单元。

表 2-2 列出了 CISC 系列微控制器和微处理器的 19 个特征。表 2-3 列出了 ARM、Intel 和 IBM 的 RISC 处理器的 19 个特征。这两个表给出了这些特征在嵌入式系统硬件各种系列处理器中的有无情况。

表 2-2 4 种 CISC 微控制器和微处理器的特征

性 能 指 标	Intel 8051 和 Intel 8751	Motorola M68HC11 E2	Intel 80196KC	Intel 奔腾
处理器指令周期(以 μs 为单位)(通常情况下)	1	0.5	0.5	0.01[1]
内部总线宽度(bit)	8	8	16	64
CISC 或者 RISC 体系结构	CISC	CISC	CISC	具有 RISC 特征的 CISC[2]

(续表)

性 能 指 标	Intel 8051 和 Intel 8751	Motorola M68HC11 E2	Intel 80196KC	Intel 奔腾
程序计数器位和复位值	16 (0x0000)	16 [(0xFFFE)]	16 (0x2080)	32[1] (0xFFFF FFFF)
堆栈指针位和处理器定义的初始复位值	8 (0x07)	16	16	32[3]
原子操作单元	无	无	无	无
流水线和超标量体系结构	无	无	无	有
片上RAM和/或寄存器文件字节	128&128 RAM	512 RAM	256&232	……
指令缓存	无	无	无	8KB[4]
数据缓存	无	无	无	8 KB[4]
程序存储器 EPROM/EEPROM	4k	8k	8k	无
程序存储器容量(字节)	64k[5]	64k	64k	4GB
数据/堆栈存储器容量(字节)	64k[5]	64k	64k	4GB
主存 Harvard 或者 Princeton 体系结构	Harvard	Princeton	Princeton	Princeton
外部中断	2	2	2	1[6]
位操作指令	有	有	有	有
浮点处理器	无	无	无	有
中断控制器	无	无	无	有
DMA 控制器通道	无	无	1(PTS)[7]	4
片上 MMU	无	无	无	有

1. 在一个典型的奔腾 100MHz 版本中，这是最大的时间(最新的奔腾 4 操作频率为 3GHz)
2. 类似于 RISC 简单指令的单时钟周期硬连线实现
3. 堆栈指针 ESP 32 位以及堆栈段 ES 16 位指向存储器中的物理堆栈地址。它等于 ES * 0x10000+ ESP。
4. 这是一个标准版本。
5. 程序和数据存储器空间在 Intel 8051 系列成员中是分开的。这种情况在其他处理器中也很常见。
6. 使用一个 INTR 管脚和一个外部可编程中断控制器，最多可以处理 256 个中断。
7. PTS 意味着有一个外设事务处理服务器，提供类似于 DMA 的特征。

存储器地址是十六进制的。应该记住，0x10000 的意思是十六进制的存储器地址 10000。0x100FF 的意思是十六进制的存储器地址 100FF。这是 C 语言的传统表示方法，用于将十进制数和十六进制数区分开来。

表 2-3　3 种 RISC 处理器系列的特征

性 能 指 标	ARM7[1]	Intel 80960CA	PowerPC MPC 604
处理器指令周期(以 ms 为单位)(通常情况下)	0.012[1]	0.03	0.0016
内部总线宽度(位)	32	32	64
CISC 或者 RISC 体系结构	兼有[2]	兼有[2]	RISC
程序计数器位	32	32[3]	32[4]
堆栈指针位	32	32[5]	32[6]
原子操作单元	无	有	无
超标量体系结构	有	有	有
片上 RAM 和/或寄存器文件字节	16 KB	1536B RAM	32 KB[7]
指令缓存	16 KB[8]	1 KB	16 KB
数据缓存	16 KB[8]	无	16 KB
程序存储器 EPROM/EEPROM	—	——	…
程序存储器容量(字节)	4GB	4GB	4GB
数据/堆栈存储器容量(字节)	4GB	4GB	4GB
主存 Harvard 或者 Princeton 体系结构	Princeton[9]	Princeton	Princeton
外部中断	32	8	2
位操作指令	有	有	有
浮点处理器	有	有[10]	有
中断控制器	有	有	无
DMA 控制器通道	无	4	无
片上 MMU	有	有[10]	有

1. ARM9 处理器的操作频率为 200MHz，具有 16 × 32 的 MAC 单元。

2. RISC 和 CISC 兼有的意思是指令集提供了两种类型的指令，但其内部的核是基于 RISC 的，以提供快速的单时钟周期处理。

3. 表示一个 32 位的指令指针代替了程序计数器。其中还有一个寄存器用来存储分支和链接指令之后的下一条指令的地址。

4. PowerPC 中具有一个 CR(计数寄存器)，代替了程序计数器。

5. 表示在 80960 中具有 3 个 32 位的指针，

6. 在 PowerPC 中，LR(链接寄存器)代替了堆栈指针。

7. 32 KB 表示在同一个高速缓存中有 16 KB 的指令和 16 KB 数据。在 PowePC 601 中有 4 种方法来设置高速缓存中每一行的 32 个字节。

8. 指令和数据缓存是很常见的(例如，在 StrongARM 中使用 16 KB、32 路指令缓存，其中有 32 字节的块用来存储数据，还有 16 KB、32 路数据缓存，其中有 32 字节的指令块(Intel))。

9. 其后续版本 ARM 9 处理器具有 Harvard 结构。

10. 只在 80969 MC 版本中。

寄存器构成了处理器中常见的内部总线。根据 ALU 是否一次执行一个 32 位、16 位或者 8 位的操作，寄存器可以是 32、16 或者 8 位的。

硬件设计者还必须记住下述处理器特有的特征。

(1) 当用汇编语言编程，或者设计编译器或定位器时，必须记住数据在处理器中应该以 big-endian 的模式存储。较低的字节存储在较高的地址中。例如，Motorola 处理器。在处理器中，数据还可以以 little-endian 的模式存储。较低的字节存储在较低的地址中。例如，Intel 处理器。处理器还可以在初始程序阶段配置为 big-endian 或者 little-endian 模式。例如，ARM 处理器。

(2) 在某些高性能处理器中，当考虑处理器性能时，必须记住外部数据总线可以以一种称为 burst 的模式访问存储器的数据。例如，32 位的处理器 80960 使用 burst 模式总线。访问 27960 存储器就是通过这种 burst 模式进行的。处理器向地址总线发送一个地址后，在 4 个连续的时钟周期内，从存储器中传出 4 个字。存储器在每 4 次连续访问字后给出一次总线地址。例如，如果 80960 给出一个地址，0x00000000。在 burst 模式下，在 4 个连续的处理器周期内，访问地址 0x00000000、0x00000001、0x00000002 和 0x00000003 中的字节。取指令及整数和浮点数的速度就可以加快。

(3) 当用汇编语言编程，或者组织某些处理器的主存时，必须记住其主存组织可能是 Harvard 结构，而不是 Princeton 结构。程序存储器和数据存储器具有独立的地址，并且指令和数据存储器区域访问的地址也是独立的。访问数据流时需要具有 Harvard 结构的处理器。例如 (i)单指令多数据类型指令；(ii)DSP 指令。

例如，考虑下列“有限脉冲响应(FIR)滤波器”中表达式的 DSP 计算。第 n 个滤波的输出序列，$y_n=\Sigma(a_i\times x_{n-i})$是对 i=0，1，2，3…N-1 求和。在这里 i、n 和 N 是整数。如果 N=10，则对于每一个数值 y，首先计算出 10 个系数其中之一的 a_i 和 10 个输入序列其中之一 x 的乘积，然后计算出这些乘积的总和。n 个数值中所有 10 个数值总的计算需要 100 次乘法和 100 次加法。从独立的存储器地址空间存储或者访问系数的速度很快，因为使用的是独立的总线。

80x86 处理器和 ARM7 的主存具有 Princeton 结构。8051 系列微控制器具有 Harvard 结构。ARM9 处理器具有 Harvard 结构。DSP 处理器都具有 Harvard 结构。现在，大多数高速缓存都是按照 Harvard 结构组织的(指令缓存和数据缓存)。

(4) 处理器可以具有独立的 I/O 存储地址空间，并具有独立的输入-输出和存储器 load-store 指令。当处理器不需要直接对 I/O 数据执行数学和逻辑操作，以及当只需要考虑有限的几个外部 I/O 单元时，就可以采取这种方法。这在计算机中是必需的。Intel 80x86 系列处理器通过单独的指令处理和访问 I/O 单元和 I/O 设备。这些处理器访问输入和输出端口时还具有单独的地址空间。它简化了与处理器相连的 I/O 单元接口电路。当针对主存中的数据进行的数学、逻辑和位操作指令也同样适用于 I/O 数据时，处理器应该有一个共同的 I/O 和主存地址空间，读写这些数据的指令也是通用的。因此，几乎所有的微控制器对 I/O 处理都没有单独的指令。Motorola 处理器也没有专门用于 I/O 处理的单独指令。

(5) 编写代码的时候必须记住，嵌入式系统必须处理多任务，并且必须在某些任务之间共享某些变量。当处理多任务的时候，必须使用原子操作(关于共享数据的问题参见 8.2.1 节)。某

些嵌入式处理器，例如 80960，提供了原子操作的硬件实现。此外，如果在处理器中没有提供原子操作，就必须使用特殊的编程技巧(例如，使用类似于信号量的 IPC。参见 8.2.2 节)。

将实时时钟(或者系统时钟)所产生的嘀哒个数存放在一个 32 位的变量 RTC_Count 中。每当从时钟传来一个中断时，RTC_Count 的值就增加 1。指令 RTC_Count++的意思是，有一个主存地址分配给了 RTC_Count。16 位或者 8 位处理器的指令可以按照如下的步骤实现。

(1) 步骤 1：向地址总线发送 2 次或者 4 次地址。这是因为处理器必须 2 次或者 4 次使用数据总线才能取回 RTC_Count。

(2) 步骤 2：通过 ALU 实现数值增加。这个过程需要占用 2 个或者 4 个周期，因为 ALU 适用于 16 位或者 8 位的操作数，而指令是对一个 32 位的数据进行递增操作。

(3) 步骤 3：将增值后的 RTC_Count 发送回原地址。

(4) 在 80x86 中有一个单机器指令，用来增加一个主存地址中的数值。编译器产生了这条指令。如果产生了一个中断，这条指令必须先执行完，然后处理器允许从当前程序切换到服务例程。如果指令没有完全执行完，就不可以进行切换。步骤 1~3 必须全部完成。如果在执行完上述 3 个步骤及其指令后产生了一个中断，称这个操作为原子操作。这是因为该指令已经完整地实现了。

(5) 在 8051 中没有单条指令可以完成上述操作。必须执行 4 条机器指令将 RTC_Count 传送到 4 个 4 位的寄存器中。然后在下一条指令中，这些寄存器中的字被接下来执行的寄存器增值指令增值。在第 3 步中这些寄存器中的字节传送回 RTC_Count 地址中。假设一条机器指令执行结束，但是增加后的值还没有传回到 RTC_Count 地址中，如果此时发生一个中断，这个操作是非原子的。其他任何使用寄存器中 RTC_Count 的程序都只能获得旧的值。在处理器没有 AOU(参见表 2-1)的情况下，在原子操作的粒度(处理器执行层次)上，指令不能执行完。此外，当返回到这个程序之后，如果 RTC_Count 已经被另外一个函数或者例程增加了值，处理器将使用旧的(没有改变的)数值来完成剩下的指令(这是因为早先保存的寄存器数值在从中断服务例程返回的时候，又被重新取出)。如果忽略了另外一个函数对这个数值的改变，计数值是错误的。

考虑将两个主存地址(m1)和(m2)中的数值相加，并将结果存储到地址(m3)中。如果在使用 ALU 执行这条指令的过程中发生了一个系统单元中断，并且如果中断单元服务例程也使用这些主存地址，则从 ALU 中计算出来的结果将是错误的，因为被中断的过程和中断过程共享了这部分主存。原子操作能够确保对一个共享存储地址的特定操作能够在被修改之前首先完成，这种修改有可能是由于嵌入式系统另外一个单元的系统总线访问造成的。另外一个示例单元是 DAM 控制器(参见 2.6 节)或者协处理器。在 80960 中有一个 AOU 单元。(i)考虑其原子 ADD 指令，atadd(m1),(m2),(m3)表示在另外一个单元进行系统总线访问改变(m1)或者(m2)的数值之前，先完成(m3)←(m1)+(m2)的操作。(ii)考虑其原子 Modify 指令，atmod(m1),(m2),(m3)表示在系统总线访问改变(m1)或者(m2)或者(m3)的数值之前，先完成(m3)←((m3)ADD(m2)) OR ((m1)ADD(m2))的操作。

(6) 当涉及到处理器的时候必须记住，嵌入式系统需要降低功耗。当在特殊的时间间隔之

内没有使用，处理器的各种结构单元中必须具有自动关机(auto-shut down)特征。处理器还必须具有可编程性，这样能够在处理普通指令的时候将处理器中的高级处理单元禁止，在处理对时间有严格要求的指令集时再启用这种单元。在实时声音视频处理单元中需要使用对时间有严格要求的指令集。PowerPC 中具有上述两种特征，并且还具有特殊的高速缓存设计，以降低功耗(因此处理器就能够在低功耗的条件下具有较高的计算能力。这可能也是处理器名称中第一个单词的意义)。最近的 StrongARM 系列处理器具有复杂的低功耗特征。

注意：

(1) 能够在高速时钟下工作的处理器每秒处理的指令数较多。

(2) 当具有下列单元时，处理器的计算性能较高：(a)流水线和超标量体系结构，(b)预取缓存单元、高速缓存、寄存器文件和 MMU，(c)RISC 体系结构。

(3) 具有寄存器窗口的处理器在多任务系统中能够提供快速的上下文切换。

(4) 低功耗嵌入式系统要求处理器的各个单元具有自动关机特性，并且当处理器要执行一个没有执行时间限制的函数或者指令时，能够通过编程来禁用这些单元。还要求具有 Stop、Sleep 和 Wait 指令。此外可能还需要特殊的高速缓存设计。

(5) 具有 burst 模式的处理器能够快速访问外部存储器，读写速度都很快。

(6) 对于单指令多数据类型指令或者 DSP 指令访问数据流的时候，需要一个具有 Harvard 主存结构的处理器。独立的数据总线能够确保同时访问指令和数据。

(7) 当适用于主存数据的算术、逻辑和位操作指令同样要适用于 IO 操作的时候，必须使用具有主存映射输入输出的处理器。

(8) 具有原子操作单元的处理器提供了设计嵌入式软件时所遇到的共享数据问题的硬件解决方法，当在多任务中共享变量的时候，还需要其他特殊的编程技巧。

2.2 嵌入式系统的处理器选择

通过分析四种具有代表性的情况，就可以了解处理器的选择过程。首先要建立一个类似于表 2-4 所示的设计表。然后选择所需要的结构单元，并且处理器性能满足要求的处理器。

(1) 情况 1：系统要求处理器指令周期时间约为 1μs，并且片上设备和存储器就足够了。示例有巧克力自动售卖机、56 Kb /s 的调制解调器、机器人、类似于 ECG 录音机、天气记录器或者多点温度和压力记录器的数据采集系统和实时机器人控制器。

(2) 情况 2：系统要求处理器指令周期时间为 10～40ns，不仅需要片上设备的存储器，还需要多媒体处理器性能。例如，2Mb/s 的路由器、图像处理、声音数据采集、声音压缩、视频解压缩、具有行稳定性的适应性巡航控制系统和网关。

(3) 情况 3：系统要求处理器指令周期时间为 5～15ns，并且需要较高的 MIPS 或者 MFLOPS 性能。例如，多端口 100Mb/s 网络收发器、快速 100Mb/s 交换机、路由器、多通道快速加密和解密系统。

(4) 情况 4：系统要求处理器指令周期时间小于 1ns，需要非常高的处理器性能，并需要使用浮点和 MAC 单元。例如，声音处理器、视频处理、实时视频或者音频处理和移动电话系统。

表 2-4　四种示例系统所需要的处理器性能

需要的处理器性能	情况 1： 巧克力自动售卖机，数据采集系统，实时机器人控制	情况 2： 声音数据采集，声音数据压缩，视频压缩，具有行稳定性的适应性巡航控制系统，网关	情况 3： 多端口网络收发器，快速交换机，路由器，多通道快速加密和解密	情况 4： 声音处理器，视频处理和移动电话系统
需要的处理器	微控制器	微处理器	多处理器系统	基于微处理器＋DSP 的多处理器系统
处理器指令周期(μs)(通常情况下)	0.5~1	0.01~0.04	0.005~0.01	0.001~0.005
处理器性能	低	中到高	高	很高
内部总线宽度(位)	8	32	32	64
CISC 或者 RISC 体系结构	CISC	CISC 或者 RISC	RISC	RISC
程序计数器和堆栈上的堆栈	16	32	32	32
外部或者内部存储器上的堆栈	外部	外部或者内部	内部	内部
片上原子操作单元	-	-	有[1]	-
流水线和超标量体系结构	无	有	有	有
是否需要片外 RAM	不，片内就足够了	是	是	是
由于快速上下文切换所需要的片上寄存器窗口	无	有	有	有
片上中断处理器	有	有	有	有
指令和数据高速缓存以及 MMU	无	有	有	有
片上存储器 EPROM	是的，片上就足够了	不，片上的 EPROM 不够	不，片上的 EPROM 不够	不，片上的 EPROM 不够
外部中断	1~16	1	128~256	16
位处理器指令	使用	使用很频繁	使用很频繁	使用很频繁

(续表)

需要的处理器性能	情况 1： 巧克力自动售卖机，数据采集系统，实时机器人控制	情况 2： 声音数据采集，声音数据压缩，视频压缩，具有行稳定性的适应性巡航控制系统，网关	情况 3： 多端口网络收发器，快速交换机，路由器，多通道快速加密和解密	情况 4： 声音处理器，视频处理和移动电话系统
浮点处理器	无	有	无	有
需要 Harvard 主存结构的数据流	无	通常不需要	也许需要	总是需要
DMA 控制器通道	无	无	有	也许有
示例处理器系列	8051，68HC11 或者 12 或者 16，80196，PIC16F84	80x86，80860，80960	ARM7，PowerPC	ARM9，TMS 系列 DSP，PowerPC

1. 当多端口和多通道操作需要数据共享的时候需要片上原子操作单元。

示例 2-2 实时机器人控制系统案例研究

(1) 机器人系统电动机需要以高于 50~100ms 的速度提供信号。因此，当使用一个指令周期时间约为 1μs 的处理器时，有足够的时间发出信号和对机器人的多个电动机进行实时控制。

(2) 处理器的速度不需要很高，所需要的性能也大大低于 1MIPS。因此不需要高速缓存以及类似于流水线和超标量处理的高级处理单元。

(3) 4 线圈步进电动机只需要 4 位的输入，直流电动机需要 1 位脉冲宽度调制的输出。因此，8 位的处理器就足够了。

(4) 需要经常对 I/O 端口进行访问和位操作。因此 CISC 体系结构就足够了。

(5) 程序可以放置在 4 Kb 或者 8 Kb 的内部片上 ROM 中，程序中所需要的堆栈很小，可以保存在 256 或者 512 字节的片上 RAM 中。因此需要微控制器，不需要浮点单元。

适用于上述案例的微控制器有 8051、68HC11、68HC12、68HC16 或者 80196。微控制器 68HC12 或者 68HC16 是最佳选择，因为它们的端口较多(68HC12 指令周期和时钟周期时间＝0.125ms，共有 12 个端口。因此具有 6 个或者更多个自由度、6 个或者更多个电动机的机器人可以通过这些端口直接驱动。处理器中有 STOP 和 WAIT 指令，当机器人处于休息状态的时候节约电能)。

示例 2-3 声音数据压缩系统案例研究

(1) 声音信号是脉冲码调制的。产生的速度为 64 Kb /s。当处理器使用高级处理单元和高速缓存的时候，使用适当的算法可以在 0.01~0.04μs(100~25MHz)的指令周期之内处理这些数据的压缩。

(2) 假设处理器指令周期为 0.02μs(50MHz)。具有三级流水线和两路超标量体系结构，最大性能为 300MIPS。(参见示例 2-1 理解 MIPS 的计算。)这个性能不仅能够满足声音压缩的需要，也能够满足图像压缩的需要。

(3) 不需要频繁地访问和执行复杂的指令。CISC 或者 RISC 结构都可以使用。

(4) 程序不能存储到片上的 4 Kb 或者 8 Kb 的内部 ROM 中，程序中需要的堆栈比较大。需要较大的 ROM 和 RAM，还需要使用高速缓存。

(5) 不需要进行浮点操作，因为多数位操作指令都是在压缩过程中处理的。

适用于上述案例的处理器有 80x86，80860 和 80960。

示例 2-4 快速网络切换系统的案例研究

(1) 网络中的快速切换需要传送 100MHz 的脉冲。假设每一次切换和收发操作需要执行 10 条指令，指令周期时间应该是 0.001 倍的脉冲宽度。对于 GHz 的传输速度，需要多处理器系统。

(2) 假设处理器的指令周期时间为 0.01μs(100MHz)。对于五级流水线以及两路超标量体系结构，最高性能应该是 1000MIPS(参见示例 2-1 理解 MIPS 的计算)。对于高于 100MHz 的切换频率，需要多处理器系统。

(3) 为了满足单周期指令处理，处理器应该具有 RISC 体系结构。

(4) ROM、RAM 和高速缓存都需要。

(5) 不需要进行浮点操作，因为多数位操作都是用来进行输入和输出的。

适用于上述案例的处理器有 PowerPC 和 ARM7。

示例 2-5 实时视频处理

(1) 实时视频处理需要对图像进行快速压缩。有必要使用 DSP。有几个实时功能必须处理：例如，图像的缩放和旋转、阴影校正、颜色和色调、图像锐化和过滤功能。在这种情况下，需要具有 DSP、并且具有处理性能最好的多处理器系统。

(2) 所有在 2.1 节中列出的高级处理单元都需要。

适用于多处理器系统的处理器有：与 TMS 系列 DSP 集成在一起的 ARM9，或者使用具有多处理器 Virtex-II Pro 的 FPGA 的 ASIC 解决方案(参见 1.6.5 节)。

注意：

不同的系统需要具有不同处理速度的处理器性能，并且需要不同的处理器特性。硬件设计者应该记住这些问题，选择性能最佳的处理器。

2.3 存储器设备

在 1.3.5 节中已经介绍了系统中存储器的使用。我们已经学习了嵌入式系统存储器的功能(参见表 1-4)。所研究的 6 个示例(参见表 1-5)指出了这样一个事实：系统不同，需要的存储器也不同。一个简单的信用借贷交易卡可能只需要 2 Kb 的存储器，而用于安全交易的智能卡，当嵌入了一个用于密码功能的 Java 程序时，可能需要 32 Kb (常用值)的存储器。复杂的嵌入式系统可能需要更大的存储器，达到 MB 的量级。

从存储设备的所有可寻址地址中，可以访问到数据字节、字、双字或者四字。对所有地址的访问过程是相同的，对于与存储器地址无关的读写操作来说，访问时间是相等的。这种模式

叫做随机访问模式，与串行访问模式区分。下面的各个小节将解释并研究关于存储器设备的一些重要技术细节。各种存储器设备的描述都是从嵌入式硬件或者软件设计者的角度进行的。

2.3.1 ROM：使用方法、形式和变种

ROM 的不可写性是其最重要的特性，这种特性适用于将代码和数据嵌入到系统中。ROM 是一个不严格使用的术语。它可以指代硬件设计者掩膜的 ROM、PROM、OTP-ROM、EPROM 和 EEPROM。严格来讲，ROM 指的是根据程序员的 ROM 映像文件(参见 1.4.2 节)而在厂家制作的掩膜 ROM。嵌入了软件或者应用逻辑电路的 ROM 有以下三种形式：掩膜 ROM、PROM 和 EPROM。当 ROM 需要在运行时编程并存储处理结果时，可以用作 EEPROM 或者闪存储器设备。

(i) 掩膜 ROM

掩膜 ROM(masked ROM)是由一个电路构造的，这个电路具有 r 个输入(A_0~A_{r-1})和 8 个输出(D_0~D_{r-1})(最常见的是在一个地址上存储一个字节)。掩膜 ROM 的电路是 2^r 个组合电路中的一个。正确的掩膜能够给每一个组合电路正确的输出。某些链接被熔断，而另外一些被掩膜的则不熔断(组合电路是由一些逻辑门构成的，当输入逻辑状态一定时，能够产生惟一的输出逻辑状态。对于 r 个输入和 8 个输出，这个电路具有惟一的真值表。只要输入发生变化，电路的输出也会发生变化)。

嵌入式软件设计者(在经过彻底地测试和调试后)提供给厂商一个文件，文件中含有输入地址各种组合的真值表。(定位器(参见 2.5 节)创建这个表。)厂商准备好编程掩膜，并在生产线上对 ROM 编程。编程后的 ROM 返回到系统工厂(正常情况下，一次掩膜的费用大约 10 万卢比(约 2 千美元)。一般来说，接下来系统厂商将下订单，厂商生产线将会接收最小为 1000 片的订单。对于大量使用的嵌入式系统厂商来说，ROM 的使用是一种很经济的解决方案)。使用掩膜 ROM 的嵌入式系统厂商在每次使用 EPROM 或者 PROM 制作系统 ROM 的时候，不必使用设备编程器(ROM 烧制器)。掩膜 ROM 永远都不会在系统厂商的层次进行烧写。

(ii) EPROM、E^2PROM 和 OTP ROM

特殊版本的嵌入式系统 ROM 可以由设计者或者厂商借助于设备编程器(device programmer)进行编程(对于编程器的工作情况请参见 G.2 节)。一种版本是 EPROM。它是一种紫外线可擦除的(Erasable)、设备编程器可编程的只读存储器(Programmable Read Only Memory)设备。擦除设备的意思是将每一个 ROM 地址上的单元阵列重新写为 1。另外一个版本是 E^2PROM(EEPROM)。它是一种电可擦除可编程只读存储器(Electrically Erasable and Programmable Read Only Memory)设备。EPROM 和 EEPROM 的示例有：4 KB 的 EPROM 2732，32 KB 的 EEPROM 28F256 和 512×16 位的 EEPROM 28F001。

在应用程序运行时，EEPROM 的擦除是通过在写脉冲有效过程中，将所有的 8 位数据总线位都发送为 1，此时的电压为高电平(+5V 或者 12V)。当将 EPROM 用作 EEPROM 时，需要使用 12V 电压。当擦除过程是在系统运行时发生时，需要使用 5V 的电压。通过一条写指令将 1、0 字节发送出去，就可以在程序运行过程中完成 EEPROM 的编程工作。必须在写一个字节之前先擦除。系统中的处理器可以完成擦除和写操作，与写 RAM 相同。如果是这样，那么 EEPROM 和 RAM 的区别是什么？区别在于，在 RAM 中，读和写的时钟周期是相同的。而对于 EEPROM，写周期必须比 RAM 的写周期长，并且必须将要擦除的字节写上 0xFF。此外，当对 EEPROM 进行读写的时候，必须提供一个 V_{pp} 电压信号。写 EEPROM 的时间是 100 万个

时钟脉冲。对于 RAM 来说没有限制，如果不首先写 1，实际上可能会需要无数的写操作。

闪存是一种最近出现的 5V 形式的 EEPROM，一个扇区的字节可以在一个瞬间(与单时钟周期比较是一个非常短的时间)擦除(最近甚至出现了 3V 的闪存)。一个扇区的大小可以是 256B~16 KB。它比 EPROM 优越的地方是：可以同时擦除许多字节，节省了每次写周期前的擦除周期时间。缺点是一旦一个扇区被擦除，必须逐个字节地写进去，其时间很长。闪存的一个新版本是引导区闪存(boot block flash)。一个扇区被预留出来，只在第一次开机的时候存储一次。开机后将不允许被擦除。换句话说，它有一个 OTP 扇区，可以用来存储 ROM 映像，与 ROM 一样。

PROM(一种 OTP ROM)是另外一种形式。PROM 一旦写入就不能被擦除。

(iii) ROM 和 EPROM 的使用

图 1-5 给出了一个 ROM 中所嵌入的程序和数据：各种任务的程序代码、中断服务例程、操作系统内核、初始化(引导程序和数据)以及其他标准数据、表格或者常量字符串。

ROM 不仅可以存储程序和数据，还可以用来获得预编程的逻辑输出，对于给定的输入序列产生输出序列。输入是由处理器发出一个地址信号完成的，输出是在处理器读周期中获得的。)假设有 8 个输入(r=8)。ROM 的真值表将有 256 个组合。可以对 8×8 的 ROM 预编程，为每一种组合产生 256 组的 8 位输出。预编程的逻辑输出应用示例如下：

(1) ROM 用来保存特定语言专用的数据，用来描述与打印机中每一个字符相关的字体。

(2) ROM 用来保存显示的图像数据。从这些数据可以产生一个象形图。在显示电路中使用 ROM。它保存与象形图的像素相关联的完整位图的字节。输入端的连续变化就会产生完整的象形图。

在 CISC 体系结构中有一个 ROM，用来作为微编程单元的控制 ROM(参见 A.1.4 节)。它为每一个处理器指令存储了一组微指令。一个序列中存储了一组微指令，它可以指定每一个指令执行过程中各种取指和执行单元的信号。

(iv) EEPROM、闪存和 OTP 的使用

EEPROM 在经过超过 1 百万次的擦除后仍然可以使用。它可以在运行时自行擦除。闪存可以承受 1 万次的反复擦除和运行时编程。PROM 只能被设备编程器或者系统首次运行时写一次。

EEPROM 存储设备应用的三个示例，如下所述。(i)存储机器的当前日期和时间。(ii)存储端口状况。(iii)存储汽车的驾驶、故障和失灵历史，以供后来分析。

闪存应用的三个示例如下所述。(i)存储数码相机的照片。(ii)在录音机中存储声音的压缩形式(从电话机中提取预先录制的信息)。(iii)在移动电话中存储信息。

OPT ROM 存储设备应用的三个示例如下所述。(i)智能卡识别号码和用户的个人信息。(ii)存储引导程序和初始化数据，例如显示图像或者字母组合的象形图。(iii)ATM 卡、信用卡或者身份证。一旦各种详细信息在银行中写入并交给帐户持有人之后，就不可能对卡中的嵌入式 PROM 进行任何改动。正如纸张一旦写上或者打印上信息之后会永久保持一样，PROM 也是一样的。

2.3.2 RAM 设备

系统设计者认为 RAM 设备有 8 种形式。这些形式包括 SRAM、DRAM、NVRAM、EDO RAM、SDRAM、RDRAM、参数化分布式 RAM 和参数化块型 RAM。这些形式将会在附录 G(参见 G.2)中作详细介绍。

RAM 的使用

正如在 1.3.5 节中已经介绍过的一样，RAM 可用来存储程序运行时的变量，还可用来存储堆栈。此外还可存储输入和输出缓冲，例如声音和图像。当 ROM 映像在嵌入式系统中是以可压缩的形式存储时，它还可以保存应用程序和数据，在系统要实际运行之前，将 ROM 映像解压缩。

SRAM 的应用最为普遍。

DRAM 主要应用于计算机或者高存储密度的系统中。

(1) 当操作频率超过 100MHz 时，EDO RAM 在系统中使用时与设备通过总线相连，两次取操作之间需要一个 0 等待状态，读或者写操作都是在单周期内完成的。

(2) SDRAM 同步读操作，当前一个字被取走的时候，将下一个字准备好。当总线允许处理器的取操作频率超过 1GHz 的时候，可以使用 SDRAM。

(3) RDRAM 是以 burst 方式访问的，一次取操作可以连续取 4 个字，因此整个系统的性能可以超过 1GHz。

(4) 在参数化的分布式 RAM 中，RAM 分部在各个系统子单元中。IO 缓冲区和收发器子单元可以分别有单独的一段 RAM，系统堆栈可以使用另外一段 RAM。在被处理器访问之前，分部式 RAM 提供给了子单元的存储器缓冲。这种设备比起通过总线访问存储器设备的系统来说，能够帮助提高 IO 设备的输入速度。

(5) 当一个特定的 RAM 块只能由一个子单元——例如 MAC 单元——使用时，要使用参数化的块 RAM。当总线访问的速度比起子单元的处理速度来说较慢时，要使用参数化的块 RAM。

注意：

各种类型、大小不一的存储器设备可以满足各种需求。(1)掩膜 ROM、EPROM 或者闪存存储嵌入式软件(ROM 映像)。掩膜 ROM 要进行散装生产。(2)EPROM 或者 EEPROM 用于测试和设计阶段。(3)EEPROM(5V)用来保存系统程序运行时的结果。它是逐个字节擦除的，在系统运行时写入。它对于存储可变数据是很有用的，例如运行时系统状态、时间、日期和电话号码。(4)闪存在经过完全的扇区擦除之后，在程序运行时逐个保存字节。(5)因此，当需要存储处理过的图像、声音、数据集合或者系统配置数据时，闪存是很有用的，当需要的时候，可以对数据进行更新。例如在闪存中，在单个的指令周期之内，可以将一个新的图像(在压缩和处理之后)存储进去，并将旧的图像擦除。(6)引导区闪存具有一个 OPT 扇区，也用来保存引导程序和初始化数据或者永久的系统配置数据。它用来将 ROM 映像或者 ROM 映像的一部分保存到 OPT 扇区中，同时其他的扇区作为一般的闪存保存数据。(7)RAM 在系统中主要用作 SRAM。(8)复杂的系统使用 RAM 的形式有 DRAM、EDO RAM、SDRAM 或者 RDRAM。(9)当 I/O 设备和子单元需要存储器缓冲区，以及当另外一个系统需要快速写 I/O 设备和子单元时，使用参数化的分布式 RAM。(10)像 MAC 这样的子单元在高速操作时，需要使用参数化的块 RAM。

2.4 嵌入式系统的存储器选择

当软件设计者编写好程序，并且 ROM 映像已经准备好之后，系统的硬件设计者所面临的问题就是应该使用哪些类型的存储器设备，每一种设备的大小为多少。首先要建立一个类似于

表 2-5 的设计表。选择具有所需要特性和大小的存储器设备。通过下面的示例，就可以知道这些问题是如何解决的。只有当编码完成，并给出适当的描述后，才能确定实际需要的存储器。ROM 映像大小和各个段、数据集合和数据结构的 RAM 分配可以由软件设计者提供。然而，下面的案例研究给出了一种事先估计所需要的存储器类型和大小的方法(需要记住可以使用的存储器设备等级。例如 1 KB、4 KB、16 KB、32 KB、64 KB、128 KB、256 KB、512 KB 和 1MB。因此当需要 92 KB 的存储器时，可以选择 128 KB 的设备)。

表 2-5 4 个示例系统所需要的存储器设备

需要的存储器	案例 1：巧克力自动售卖机，实时机器人控制系统	案例 2：数据采集系统	案例 3：多端口网络收发器，快速交换机，路由器，多通道快速加密和解密系统	案例 4：音频处理器，视频处理和移动电话系统	案例 5：数码相机或者录音系统
使用的处理器	微控制器	微控制器	微处理器系统	基于微处理器＋DSP 的多处理器系统	微处理器
内部 ROM 或者 EPROM	4~8 KB	—	—	—	—
内部 EEPROM	256B~512B	256B~512B	—	—	—
内部 RAM	256B~512B	256B~512B	—	—	—
ROM 或者 EPROM 设备	不需要	不需要	64 KB	64 KB	64 KB
EEPROM 或者闪存[1]设备	0~126 KB	64 KB	512 KB	32 KB	256 KB ~16MB
RAM 设备	不需要	4 KB ~8 KB	64KB	1MB	1MB
参数化的分布式 RAM	不需要	不需要		IO 缓冲区需要，每个通道 4 KB	
参数化的块 RAM	不需要	不需要		MAC 单元、拨号 IO 单元需要	—

注释：

1. 具有引导区的闪存可以用来将引导程序中受保护的部分存储到其 OTP 扇区中。

示例 2-6a/b 自动洗衣机和巧克力自动售卖机系统的案例研究

(1) 先来看自动洗衣机系统。(a)EEPROM 的第 1 个字节用来存储状态(洗衣、漂洗周期 1、漂洗周期 2 和甩干)，这些状态在洗衣机运行的某个瞬间完成。EEPROM 的第 2 个字节用来存储当前阶段已经运行的时间。第 3 个字节用来存储用户设置的按钮状态。因此微控制器中 128 个字节的 EEPROM 就足够了。(b)嵌入式软件可以写入到微控制器 4 KB 的 ROM 中。(c)只有几

个变量和堆栈需要使用 RAM。128B 的内部 RAM 就足够了。(d)因此，当使用微控制器的时候，不需要外部存储器设备。

(2) 再来考虑巧克力自动售卖机系统。(a)需要 EEPROM 字节来存储时间和日期。还需要 EEPROM 字节来存储机器状态和现金收取通道、硬币找零通道中每一种硬币的数量。因此微控制器中 128 字节的 EEPROM 就足够了。(b)嵌入式软件可以保存到微控制器中 4KB 的 ROM 中。只有几个变量和堆栈需要 RAM。128B 的内部 RAM 就足够了。(c)因此，当使用微控制器的时候，不需要外部存储器设备。

(3) 再来考虑机器人系统。(a)需要 EEPROM 字节存储每一个自由度剩下的状态。因此微控制器中 512 字节的 EEPROM 就足够了。(b)嵌入式软件可以保存到微控制器中 8KB 的 ROM 中。(c)只有变量需要存储在 RAM 中，只需要一个堆栈用来存储子例程调用的返回地址。512 字节的内部 RAM 就足够了。因此，当使用微控制器的时候，不需要外部存储器设备。

示例 2-7a/b　用于 16 参数通道和 ECG 波形的数据采集系统的案例研究

(1) 先来考虑第一个数据采集系统。假设有 16 个通道，在每一个通道中，每分钟存储 4 个字节的数据。(a)有一些字节需要保存在闪存中。假设结果需要在闪存中保存一天，然后打印出来或者传送到计算机中。因此一天需要 92KB。此时选用 128KB 就足够了。(b)嵌入式软件可以保存到微控制器中 8KB 的 ROM 中。(c)只有变量需要存储在 RAM 中，只需要一个堆栈用来存储子例程调用的返回地址。512 字节的内部 RAM 就足够了。(d)为了将 ADC 结果以适当的形式保存，需要进行中间计算。还需要考虑单元转换功能，大小约为 4KB~8KB 的 RAM 是必要的。(e)因此，系统需要一个具有 8KB EPROM 和 512 字节的 RAM 的微控制器，还需要 128KB 的外部闪存(或者 5V 的 EEPROM)和 4KB~8KB 的外部 RAM。

(2) 再来看另外一个数据采集系统，这个系统用来存储记录 ECG 波形。设每一个波形需要记录 256 个点。对于 256 个病人记录，需要 64KB 的闪存。

示例 2-8　多通道快速加密和解密收发器系统

(1) 考虑一个多通道的系统。每一个通道都有加密的输入。这些输入是需要发送给其他系统的中继数据。

(2) 需要 EEPROM 来配置端口并存储其状态。假设有 16 个通道。512KB 的 EEPROM 就足够了，每个通道 16 个字节。

(3) 加密和解密算法可以存储在 64KB 的 ROM 中。

(4) 在高速缓存处理算法之前，需要多通道数据缓冲区。因此需要 1MB 的 RAM。

(5) 每一个通道需要 4KB 的 IO 缓冲区。如果每个通道都使用参数化的分布式 RAM，系统性能将会提高。

(6) 因此系统将需要如下的存储器系统，64KB 的 ROM、512KB 的 EEPROM、1MB 的 RAM 和每个通道 4KB 的参数化分布式 RAM。

示例 2-9　移动电话系统的案例研究

(1) 由于处理音频的压缩和解压缩、加密和解密算法、DSP 处理算法的需要，ROM 映像会比较庞大。假设大小为 1MB。如果 ROM 映像是以压缩的形式存储的，引导程序还要首先运行一个解压程序。解压程序和数据首先保存到 RAM 中，应用程序会从这里开始运行。这些系统中的 ROM 显然比较大。可以按照压缩参数来缩减 ROM 的需求。

(2) 还需要较大的 RAM。RAM 可以是 1MB 的，用来存储解压的程序和数据，以及用作数据缓冲区。

(3) 用来保存打入的重要电话号码的电话号码存储器可以是 16KB 的 EEPROM。使用 EEPROM 的原因是当数据发生变化的时候，需要逐个字节进行变化。可以用一个 16KB 的闪存来记录信息。

(4) 在 MAC 子单元或者其他的子单元中使用参数化的块 RAM 会提高系统的性能。

(5) 因此系统将需要下列的存储器设备，1MB 的 ROM、16KB 的 EEPROM、16KB 的闪存、1MB 的 RAM 和某些子单元中的块 RAM。

示例 2-10　数码相机和录音机的案例研究

(1) 假设是一个低分辨率的黑白数码相机系统。需要记录 gif 的压缩格式(图像格式)。(a)假设一个图像具有 144×176 像素的 Quarter-CIF(四分之一通用媒介格式)。那么每个图像就需要存储 25344 个像素。假设将图像以系数 8 压缩，则每一个图像就需要占用 3KB 的空间。对于 64 个图像，需要 0.2M 的闪存。因此 256KB 的闪存就足够了。(b)一个 64 数字黑白相机系统将需要下列的存储器设备，64KB 的 ROM、256KB 的闪存和 1MB 的 RAM。

(2) 再来看一个录音系统。假设声音信号是 8 位脉冲码调制的，需要 64Kb/s(平均频率是 8kHz)。假设声音数据以系数 8 进行压缩，每秒需要 1KB 的闪存。对于每小时的录音，需要 4MB 的闪存。

(3) 由于必须执行声音压缩和解压算法，ROM 映像将非常庞大。它可以保存在 1MB 的空间中。使用压缩技术，一个 64KB 的 ROM 就足以存储 ROM 映像。

(4) 存储解压程序需要很大的 RAM。估计为 1MB。

(5) 1 小时的录音系统估计需要下列的存储器设备，64KB 的 ROM、4MB 的闪存和 1MB 的 RAM。

注意：

简单的系统，例如巧克力自动售卖机或者机器人系统，不需要外部存储器设备。设计者可以选择微控制器，因为微控制器上有系统需要的片上存储器。数据采集系统需要 EEPROM 或者闪存。移动电话系统需要大于 1MB 的 RAM 设备和大于 32KB 的 EEPROM 或者闪存设备。图像、声音或者视频录制系统需要很大的闪存存储器。

2.5　程序段和块的存储器分配及系统的存储器映射

2.5.1　各种存储器段中的函数、过程、数据和堆栈

程序例程和过程可以有不同的段。例如一个程序可以分段，每一段存储在不同的存储器块中。指针(pointer)地址指向存储着一个程序段的开始处，可以使用偏移量(offset)数值来获取段中的存储器地址。也可以将数据分段，每一段存储在不同的块中。与此类似，字符串也可以分段。图 2-2(a)给出了在软件设计过程中可能会需要的不同段的类型。一个段可以分为固定大小的部分，称为页(page)。处理器(例如 80x86)可以包含段寄存器和偏移量寄存器(参见图 2-2(b))。

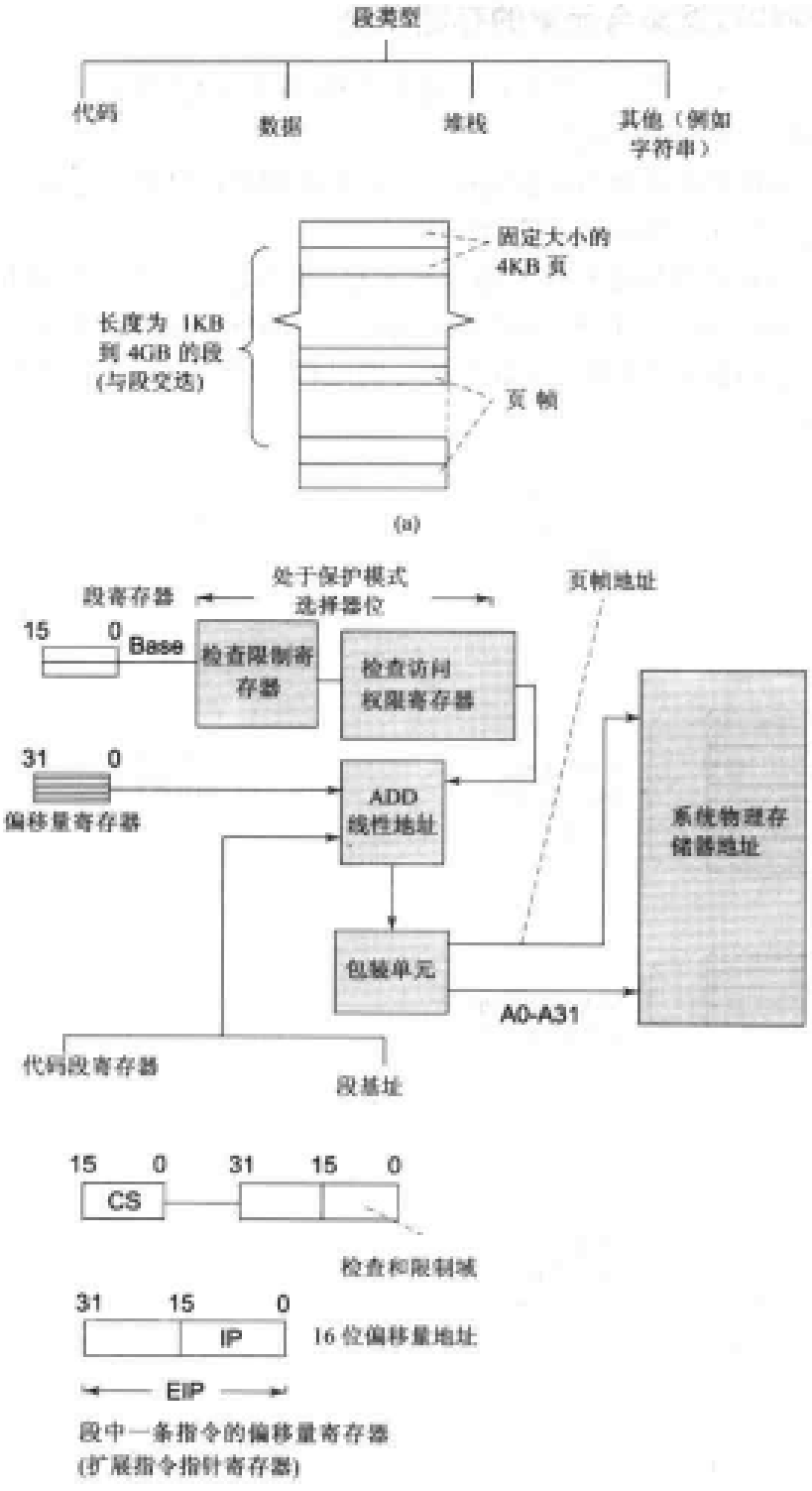


图 2-2　(a)示例程序中的段类型和页；(b)80x86 中寄存器、CS 和 EIP 使用的示例，用来取回或者写物理存储器地址

注意：

对于函数和过程(任务)，存储器中可以有不同的段。这包括用于数据和用于堆栈的段。每一个段都具有开始存储器地址和结束存储器地址。每一个段都具有指针地址和偏移量地址。可以使用偏移量从一个段中取回代码或者数据字。

2.5.2 不同数据结构和数据集合元素的存储器块

软件设计方法就是在程序中使用数据集合和数据结构(参见 5.4.2 节)。因此，对于系统软件设计者来说，理解下面的内容是很重要的。

存储器中可以有不同的数据集合和数据结构。下面是系统处理过程中通常使用的数据结构和数据集合，它们存储在系统中不同的存储器块中。

A. 称为堆栈(stack)的数据结构是程序中的特殊元素。堆栈表示一个分配的存储器块，处理器总是以 LIFO(Last In First Out，后进先出)的方式读取其中的每一个数据元素。在过程中可以产生各种堆栈结构。图 2-3 给出了在嵌入式软件执行的过程中所产生的各种堆栈结构。

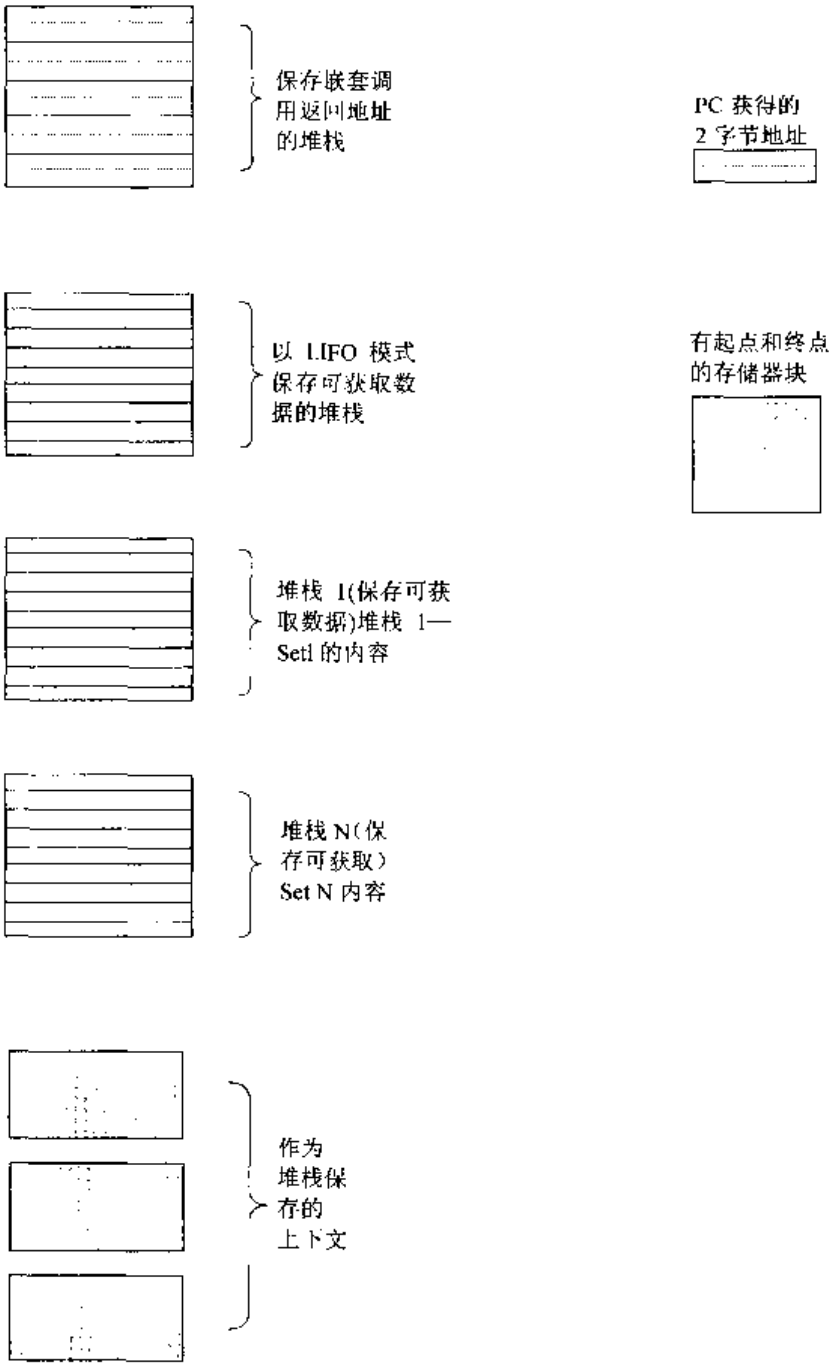


图 2-3 存储器块中的不同堆栈结构的示例。每一个堆栈指针都指向堆栈的栈顶，处理器可以从这里进行读写操作。堆栈中的数据总是以 LIFO 的模式取回的

(1) 在一个例程的运行过程中，可以调用另外一个例程。为了在被调用的例程执行完后，处理器只返回到发出调用的那个例程，返回的指令地址必须存储到堆栈中。这也可以是嵌套的。意思是说一个例程调用另外一个例程，从被调用的例程总是返回到发出调用的例程。因此，在存储器中就要分配一块存储器地址给堆栈，能够保存嵌套调用的返回地址。

(2) 开始的时候，在 RAM 中应该有一个输入数据被保存为一个堆栈，以便在后来的 LIFO 模式中取回。应用程序可以产生运行时的堆栈结构。在不同的存储器块中可以有多数据堆栈(multiple data stack)，每一个堆栈都具有单独的指针地址(5.6 节中将介绍软件设计者如何使用堆栈数据结构)。

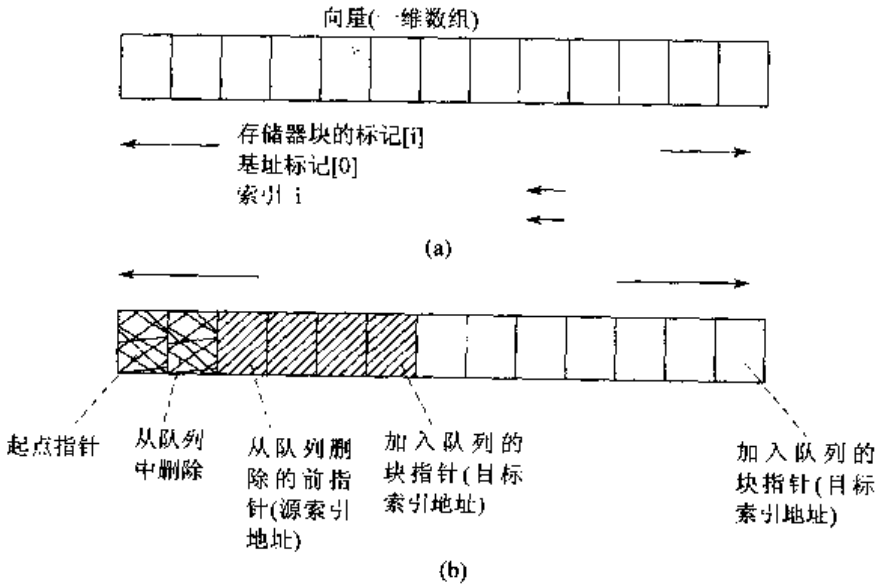
(3) 多任务或者多线程软件设计(参见 8.1 节)中的每一个任务或者线程都应该有自己的堆栈，其中存储着它们的上下文(context)。当处理器切换到另外一个任务或者线程的时候，需要保存上下文。上下文中包含着程序计数器的返回地址，以便在切换回到任务的时候取回。在存储器中，不同的存储器块可以有不同的上下文有多堆栈(multiple stack)，每一个存储器块都具有单独的指针地址。应用程序和监督程序(OS)在单独的存储器块中有单独的堆栈。

每一个处理器至少具有一个堆栈指针，可以使其指向指令堆栈，并协助例程的调用。在 8051、68HC11 和 80196 中都有一个堆栈指针。一些高级的处理器具有多个堆栈指针。在 80960 中有 4 个堆栈指针，分别是 RIP、SP、FP 和 PFP。MC8010 提供了两个堆栈指针，USP(User Stack Pointer，用户堆栈指针)和 SPP(Supervisory Stack Pointer，监督堆栈指针)。MC68040 提供了 4 个堆栈指针，USP(User Stack Pointer，用户堆栈指针)、SPP(Supervisory Stack Pointer，监督堆栈指针)、存储器堆栈帧指针和指令堆栈指针。(块也可以称为帧。)当处理器只有一个堆栈指针的时候，OS 会分配存储器地址，用作多指令和多数据堆栈的指针。

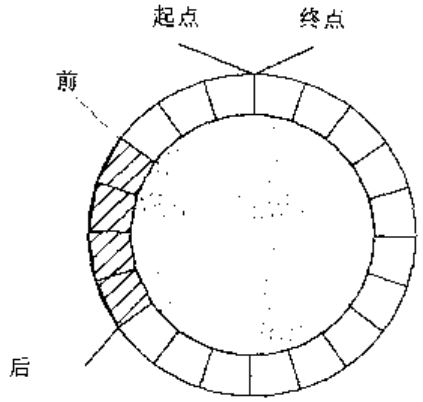
注意：

堆栈是存储器中的特殊数据结构。它具有一个指针地址，总是指向堆栈的栈顶。这个指针地址称为堆栈指针。处理器至少具有一个堆栈指针。堆栈中的数值是以 LIFO 的模式从存储器中取回的，而数据行、表格或者队列中的数据是以 FIFO(First In First Out，先进先出)的模式访问的。每一个过程都应该有单独的栈顶指针，并且在分配的存储器中有一个单独的块。由于嵌入式系统中有多个过程，因此有多个堆栈。

B. 数组(array)数据结构是一个重要的编程元素。让我们来看一个具有 30 名学生、每个学生的编号为 1~30 的班级的测试结果。令 i 代替名单编号作为索引。令名单编号 1 的测试标志存储在一个可伸缩的整型变量 M[0]中。令 M[0]、M[1]，…，M[28]和 M[29] 分别作为名单编号 1，2，…，29，30 的标志向量。有一个指针指向第一个可伸缩的变量 M[0]。它可以保存在一个称为索引指针的寄存器中。这样索引寄存器就可以通过一个循环中的指令从 0 增加到 29，这个指令指向前一个名单编号学生的标志。再来看另外一个例子，表达式 $y_n=\Sigma(a_i\times x_{n-i})$的系数为 a_i。这些系数存储在一个数组中。输入值 x_i 和输出值 y_k 也分别存储在另外两个数组中。在这里，i、j 和 k 都是从 - N 到 N - 1 变化的整数，N 是上限。图 2-4 给出了存储器块中的一个数组，指针和索引共同指向它的元素。



循环队列,但当队尾超过终点时，队尾等于队首



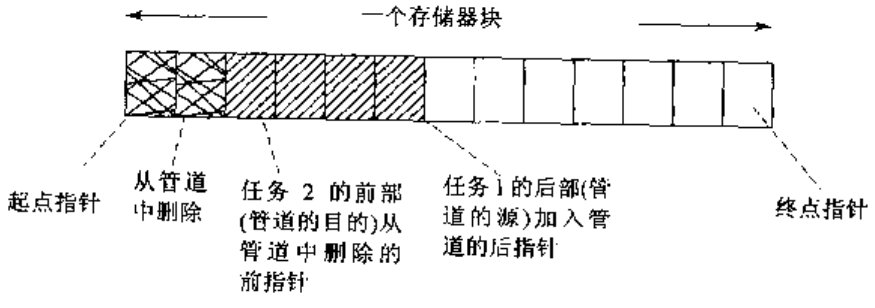


图 2-4 (a)存储器块中的一个数组，具有一个基指针，第一个元素的索引为 0。可以通过定义指针和索引从某个元素中取回数据。(b)存储器块中的一个队列，具有两个指针，分别指向其第一个和最后一个元素。数据字从队列中是以 FIFO 的模式取回的。(c)存储器块中的一个循环队列，具有两个指针分别指向其第一个和最后一个元素。(d)管道的存储器块，第一个和最后一个元素分别指向不同的任务

注意：

一维数组是存储器中的特殊数据结构。它具有一个指针地址，总是指向数组的第一个元素。根据第一个元素指针和这个元素的索引，可以产生一个地址，处理器可以使用这个地址访问一个数组元素。索引是一个从 0 开始的整数。可以从分配给数组的存储器块中的某个元素地址中取回数据。还可以用一个处理器寄存器来保存索引，另外一个寄存器保存数组基指针。

C. 称为队列(queue)的数据结构是另外一个重要的编程元素(队列是一个分配的存储器块，其中的数据元素总是以 FIFO 的模式取回)。使用队列数据结构，可以将数据通过网络传送到一个文件中或者打印机中。总共需要两个指针。一个用于指向存储器块的一个地址，元素可以从这里插入(写操作)。每次增加元素后这个指针都应该增加，称为队尾指针。另外一个指令用于指向存储器块的一个地址，元素可以从这里删除(读操作)。每次删除元素后这个指针都应该增加。两个指针开始的时候都指向块中的起始存储器地址，称为队首指针。向队列中插入数值通常要比从队列中删除数值快。例如，在打印机的队列中，系统插入数值的速度要比打印的速度快。两个指针的差值是队列的当前长度。4.5 节将介绍嵌入式软件设计过程中队列的使用。由于存在尾指针超过存储器块末地址所设定的限制的可能性，当指针超过了存储器块的末边界时，通常会产生一个异常(一种出错指示)。此时继续增加元素会占用其他的存储器块。图 2-4(b)给出了具有两个指针的存储器块，分别用来插入和删除元素。

注意：

队列是一种分配了存储器块(缓冲区)的数据结构，其中的数据元素总是以 FIFO 的模式取回的。它有两个指针，一个指向首，一个指向尾。任何的删除操作都是从有地址开始的，任何插入操作都是在尾地址进行的。当指针超过了块的末边界时，必须产生一个异常(一种出错指示)，然后采取适当的处理。

D. 当指针到达一个界限时就会返回到其起始值的队列，称为循环队列(循环队列(circular queue)是分配给队列的一个有边界限制的存储器块，其指针永远都不会超过设定的限制)。循环队列中的数据元素仍然是以 FIFO 的模式取回的，并且当指针超过了存储器块的界限时不会产生异常。图 2-4(c)给出了一个具有循环队列的存储器块，两个指针分别用于插入和删除操作。

注意：

循环队列是一种两个指针都不会超过存储器块(缓冲区)的队列，当指针超过了限制后，会复位到初始值。

E. 当执行插入操作一端的标识与执行删除操作一端的目的标识不同时，通常称这样的队列为管道(pipe)。管道是分配给一个队列的普通存储器块，这个队列中有两个实体以某种方式相连。例如，一个实体用来向队列中写(插入)元素，另外一个实体用来从队列中读(删除)元素。一个管道通常连接两个任务。图 2-4(d)给出了具有管道的存储器块。

注意：

管道是一个清楚地定义了源和目的实体的队列。有时候程序员也称管道为队列，反之亦然。

F. 表(table)是一个二维数组(矩阵)，是分配了存储器块的一个重要数据集合。表总是有一个基指针。它指向表的第 1 列第 1 行的第 1 个元素。存在两个索引，一个是列索引，一个是行

索引。图 2-5(a)给出了一个具有表指针的存储器模块。与数组相同，任何元素都可以通过三个地址(表基址、列索引和行索引)取回。当用表中的数值代替指针的时候，指令中使用的数值称为位移(displacement)。位移可以是一行或者一列。

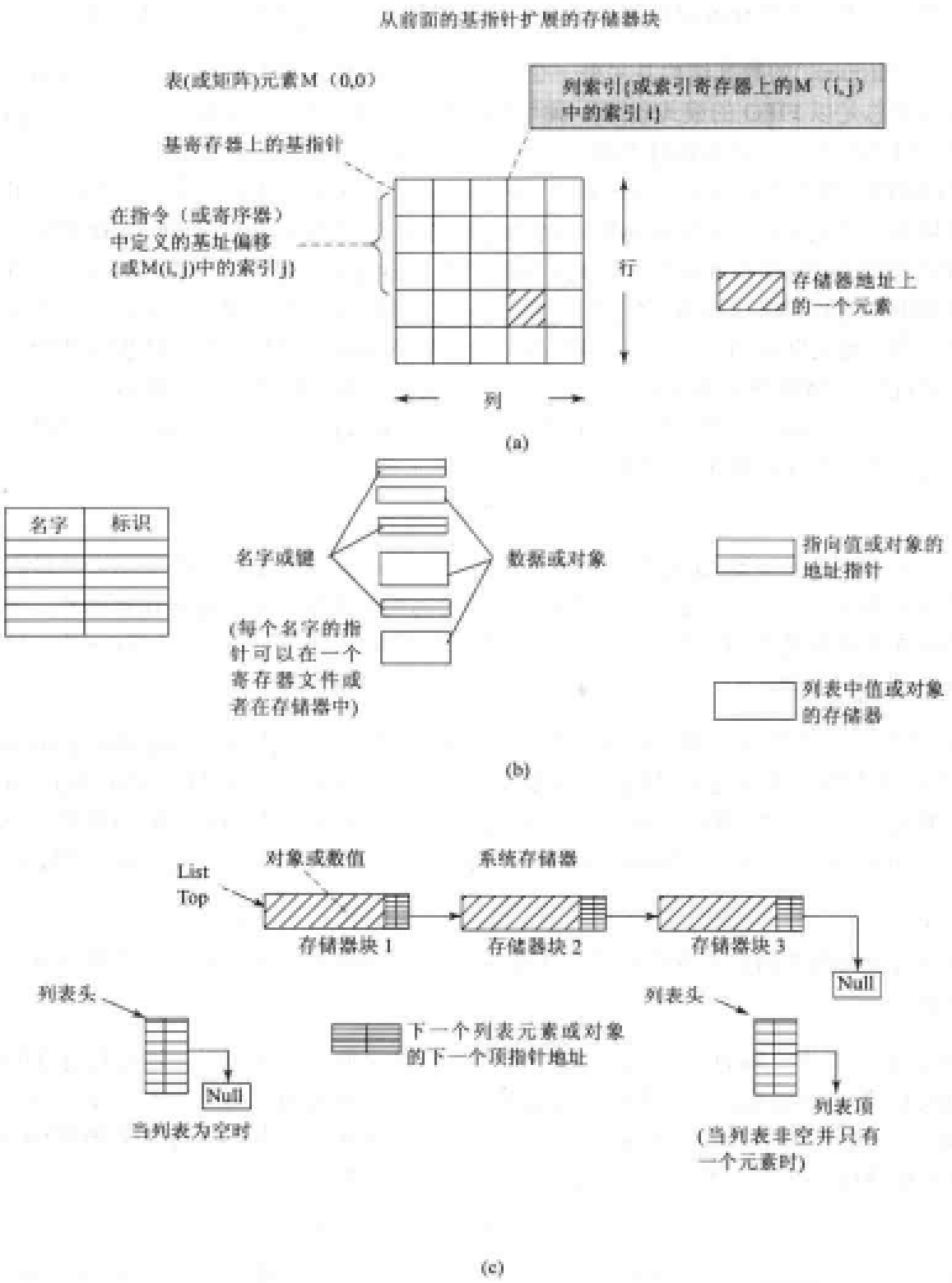


图 2-5 (a)具有表指针的存储器块；(b)具有哈希表或者查询表指针的存储器块；(c)具有链表指针的存储器块

注意：

表是一个分配了存储器块的数据集合。通过三个地址(表基址、列索引和行索引)可以取回表中的任意一个元素。

G. 哈希表(hash table)是一个键/值数据集合。哈希表的每一列都有一个键或者名字。相应的数值或者对象在第 2 列。键可以存在于不连续的存储器地址中。查表的过程就像创建散列。如果表的第 1 列作为键(指向数值的指针)，第 2 列作为数值，我们称这样的表为查询表。图 2-5(b)给出了一个具有哈希表指针的存储器块。

注意：

哈希表是一个分配了存储器块的数据集合，通常用作查询表。正如索引标识了数组元素一样，一个哈希键也标识了一个哈希元素。

H. 链表(list)是一个具有多个存储器块的数据结构，每一个元素都有一个存储器块。链表具有顶(头)指针，指向链表开始处的存储器地址。存储器中的每一个链表元素也保存了指向下一个元素的指针。最后一个元素不指向任何地方。链表用来保存存储器中不连续存储的对象。图 2-5(c)给出了具有链表指针的存储器块。

注意：

链表是每一个元素都保存了指向链表中下一个元素的指针的数据结构。每个元素都分配了一个存储器块。链表头指针指向它的第一个元素，最后一个元素不指向任何位置。

示例 2-11

一个应用程序的函数设计如下所述。其数据集合和数据结构的存储器块分配随后给出。

(1) 对系统或者其他函数的 10 次递归调用。需要分配一个堆栈数据结构，用来保存返回指令地址，大小为 40 字节(假设程序计数器地址是 4 字节)。

(2) 在每次调用的时候，下列内容也保存到堆栈中。(i) 4 个指针(每个地址为 4 个字节)。(ii)4 个整数(每个整数为 4 个字节)和(iii)4 个浮点数字(每个为 4 个字节)。需要分配一个堆栈数据结构，用来保存函数参数，大小为 4×4+4×4+4×4=48 字节。

(3) 三个数组，通过表达式 $y_n = \sum(a_i x_{n-i})$ 计算滤波后的输出序列(参见 2.1 节)。需要分配三个数组结构，每个大小为 20×8=160 字节(假设每一个元素都是 8 字节的双精度浮点数(64 位的 IEEE 754 格式))。

2.5.3 存储器映射

图 2-6(a)给出了系统中具有 Princeton 体系结构时所需要的存储器区域。图 2-6(b)给出了系统中具有 Harvard 体系结构时所需要的存储器区域。

(1) 在 Princeton 体系结构的情况下，程序中的向量和指针、变量、程序段和保存数据和堆栈的存储器块都具有不同的地址。

(2) 在 Harvard 体系结构的情况下，程序段和保存数据和堆栈的存储器块具有独立的地址空间。控制信号和读写指令也是分开的(关于段和块，请参见 2.5.1 节和 2.5.2 节)。

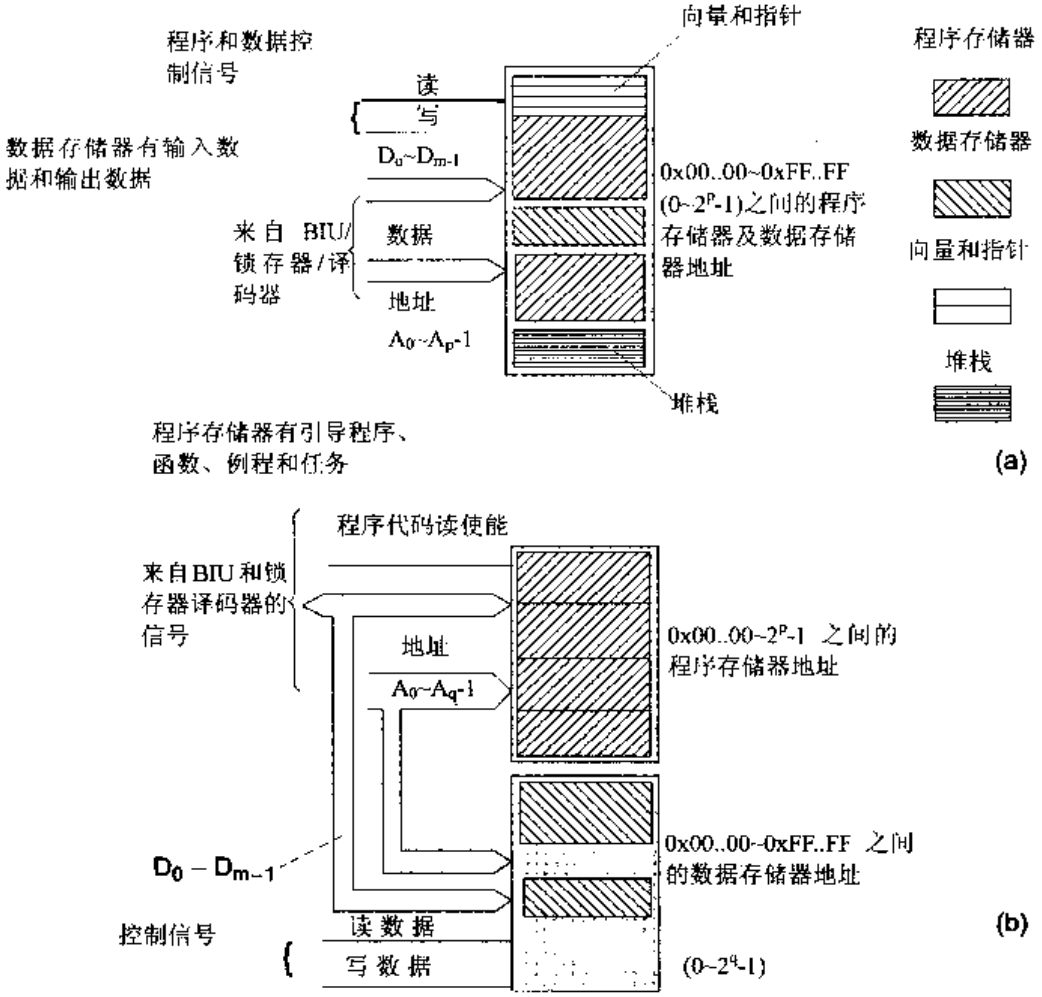


图 2-6 (a)Princeton 体系结构的存储器映射(b) Harvard 体系结构的存储器映射

图 2-7 给出了 80960 的外部存储器设备，以及与 80960 中的端口地址和存储器地址共存的存储器块。系统存储器分配映射(system memory allocation-map)不仅是存储器块中可用地址以及 IO 设备可以使用的段和地址的反映，也是系统硬件中存储器和 IO 设备描述的反映。它反映了不同单元中的各种存储器的实际存在性，如 EPROM、PROM、ROM、EEPROM、闪存、SRAM(静态 RAM)、DRAM(动态 RAM)和 IO 设备。它反映了定位器对程序、数据和 IO 操作的存储器分配情况。它说明了这些地址上的存储器块和端口(设备)。图 2-8(a)和(b)分别给出了 68HC11(具有存储器映射的 IO 体系结构)的存储器和 I/O 设备存储器的分配映射，以及 80x86 PC(具有 IP 映射的 IO 体系结构)的分配映射。

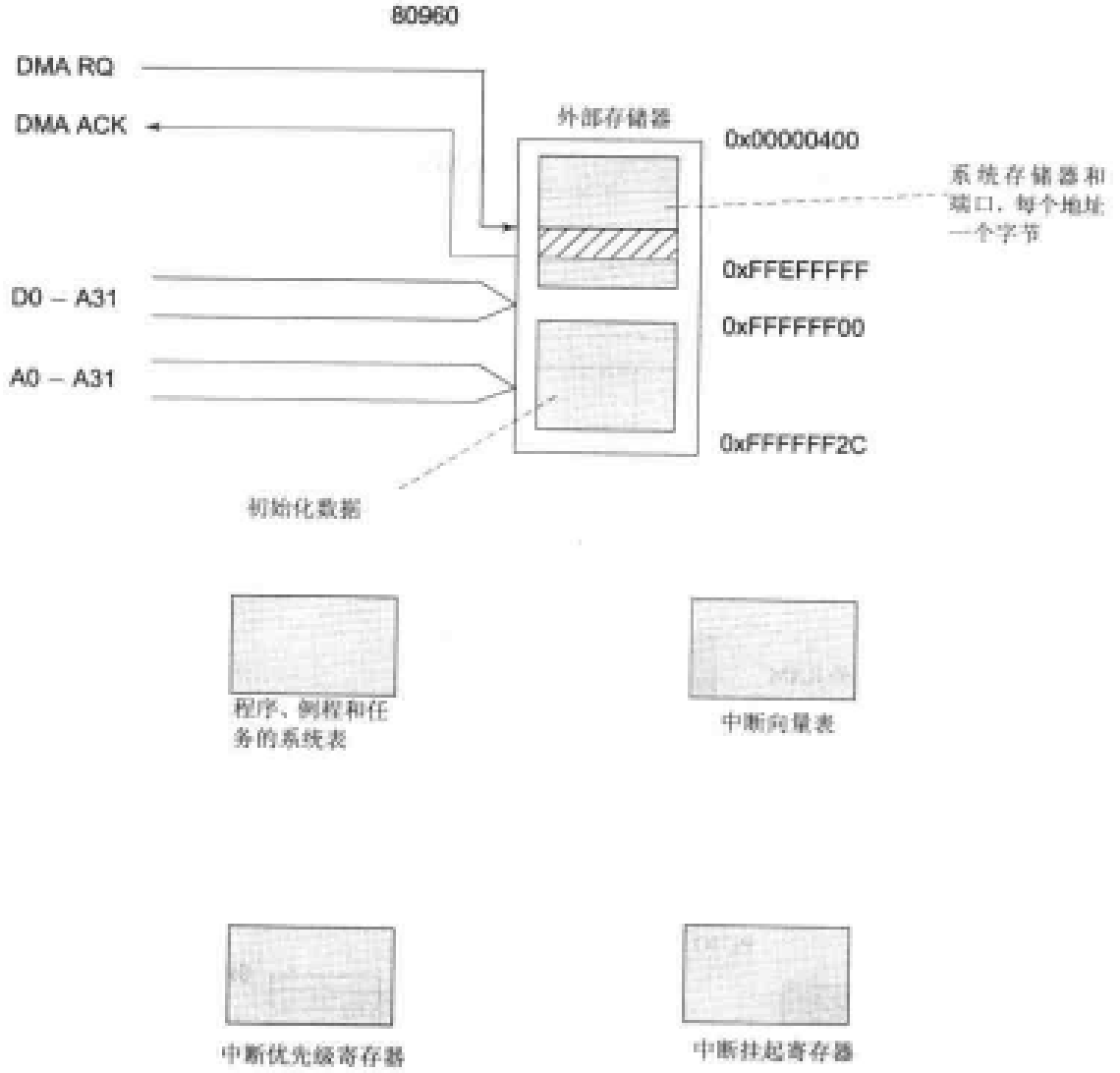


图 2-7　80960 的外部存储器设备，以及与端口地址共存的存储器块(还给出了总线信号)

图 2-9(a)~(d)给出了 4 个存储器分配映射的示例。系统 I/O 设备映射(system I/O device map)可以单独设计。它不仅反映了 I/O 设备的实际存在性，还反映了各种设备寄存器和端口数据的可用地址(示例设备是定时器。I/O 设备是系统的外设单元)。

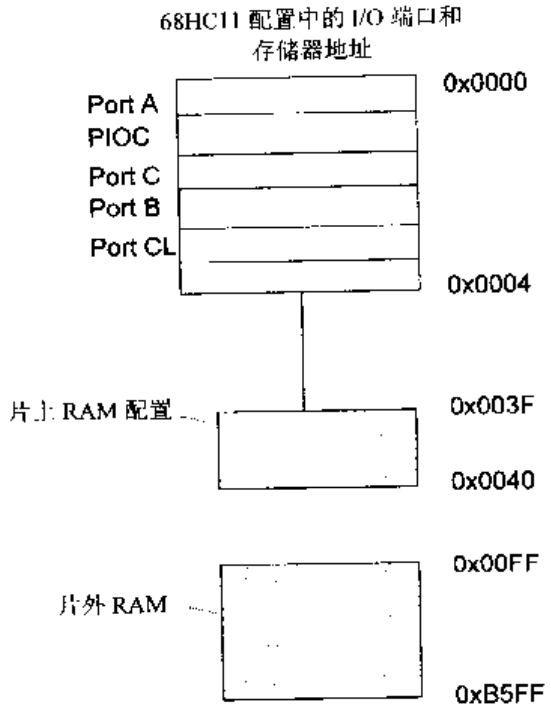
下面是对图 2-9 中 4 个示例系统的定位器的存储器分配映射的介绍。

示例 2-12

嵌入式系统示例(借贷卡)的存储器映射需要一个 2KB 的存储器。它还需要一个 256 字节的 RAM，主要用于堆栈。还需要 512 字节的 EEPROM，用来存储贷方或者借方的余额，以及卡的前一次交易记录。因此这个系统的存储器定位器或者链接脚本程序可以如下定义存储器映射。

```
1. Memory
2. {ram : ORIGIN = 0x10000, LENGTH = 256
3. eeprom : ORIGIN = 0x20000, LENGTH = 512
4. rom : ORIGIN = 0x00000, LENGTH = 2K
```

5. }



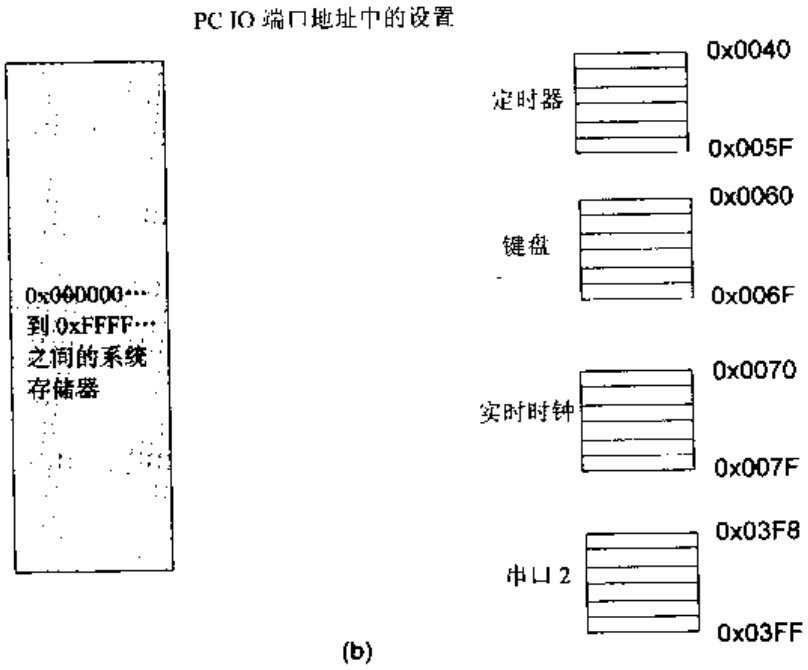


图 2-8 (a)68HC11 的 IO 端口、存储器和设备地址空间 (b)基于 80x86PC 的设备地址

图 2-9　(a)-(d) 4 个示例系统中的 4 个存储器分配映射及其定位器

示例 2-13

嵌入式系统示例(具有加密软件和解密处理的 Java 嵌入式卡)的存储器映射需要一个 32KB 的 ROM 存储器和一个 4KB 的 RAM。它还需要 512 字节的 EEPROM 来存储借方和贷方的余额，以及密码关键字和卡的前一次交易记录。因此这个系统的存储器定位器或者链接脚本程序可以如下定义存储器映射。

```
1. Memory
2. { ram : ORIGIN = 0x10000, LENGTH = 4K
3. eeprom : ORIGIN = 0x20000, LENGTH = 512
4. rom : ORIGIN = 0x00000, LENGTH = 32K
5. }
```

从上述示例中还可以做出如下重要的判断。不管可用存储器的长度是否非常小，在初始的

ROM、RAM 和 EEPROM 之间存在着存储器空间间隔。这个间隔是由硬件设计的设计特征造成的，这样可以在将来不改变存储器和处理器之间的接口译码电路的情况下，对这些存储器进行扩展(参见 2.7 节)。此外，其软件程序必须做一些小的改动。这些变化只是长度上的。原因是当不存在间隔的时候，起点也要发生变化。这个特征确保了将来程序代码大小和数据大小发生任何变化都没有必要改变定位器代码。定位器也必须有一个特征，使之不必重新定位专用端口的地址，这些专用端口专用于一个特殊的 I/O 任务或者专用于设备驱动程序的读写操作。在 G.2 节中介绍了用于设备程序的定位器输出记录的格式。

嵌入式系统设计的最后一个步骤是，在 ROM 映像中定位引导(复位)程序和数据、初始化数据和标准数据、表或者常量字符串设备驱动程序和数据、各种任务的代码、中断服务例程和操作系统内核(引导程序包含着只有在系统复位时才会执行的指令。引导数据应用的示例是用于堆栈指针的初始化。初始化数据可能用来定义初始状态和系统参数。常量字符串可以用来初始化屏幕显示)。ROM 中有一个映像段。在映像段中有初始化数据、常量字符串和启动代码，这些代码在系统开机的时候由影像段复制程序复制到 RAM 中。当执行开机代码程序的时候，在 RAM 中就产生了一个从 ROM 复制来的影像段。RAM 还保持了数据(立即数和输出数据)和堆栈。如果系统映射的 ROM 映像比较长，ROM 中存放的是压缩的程序格式。这是因为解压程序和压缩的映像比非压缩的 ROM 映射需要的存储空间小。开机代码的任务是产生解压缩后的程序代码，并存储到 RAM 中。处理器从 RAM 中取回这些代码，并顺序执行。

处理器可以为初始化开机记录提前定义存储器位置。例如在 80960 中，一个记录包含 12 个字。它们存储在从 0xFFFFFF00~0xFFFFFF 的 ROM 地址中(参见图 2-7)。

示例 2-14

在存储器分配映射中由许多节(section)。参考一下它在定位器中的描述。嵌入式系统示例 2-12 中的借贷卡存储器可以做如下的描述。

```
 1. SECTIONS
 2. {/* Stack Top Location for 256 B RAM*/
 3. _TopOfStack = 0x10100;
 4. /* Bottom of Heap */
 5. _BottomOfHeap = 0x10080;
 6. text rom :
 7. {/* Debit-credit card program instructions are at the text file named here*/
 8. * (——.txt)
 9. }
10. data ram :
11. {
12. /* Shadow Segment for 16 byte of Initialised Data at RAM for a copy from ROM from */
13. _DataStart = 0x10000;
14. /* Debit-credit card shadow segment data at the data file named here*/
15. * (——.data)
16. _DataEnd = 0x1000F;
17. }
18. /* Command for copy into the RAM */
19. > rom
20. bss :
21. {
22. /* Base Segment for 32 byte of Program Variables Data at RAM */
```

```
23. _bssStart = 0x10010;
24. /* Debit-credit card base segment data at the base segment data file named here*/
25. * (——.bss)
26. _bssEnd = 0x1002F;
27. }
28. }
```

示例 2-15

参考一下示例 2-13 中所描述的嵌入式系统——Java 嵌入式卡的存储器分配映射的段。

```
 1. ECTIONS
 2. {/* Stack Top Location in 4 KB RAM*/
 3. _TopOfStack = 0x11000;
 4. /* Bottom of Heap */
 5. _BottomOfHeap = 0x10900;
 6. text rom :
 7. {/* Encrypting Java Card program instructions are at the text file named
    here*/
 8. * (——.txt)
 9. }
10. data ram :
11. {
12. /* Shadow Segment for 256 bytes of Initialised Data at RAM for a copy from */
13. _DataStart = 0x10000;
14. /* The card shadow segment data at the data file named here*/
15. * (——.data)
16. _DataEnd = 0x100FF;
17. }
18. /* Command for copy into the RAM */
19. > rom
20. bss :
21. {
22. /* Base Segment for 2 KB Program Variables Data at RAM */
23. _bssStart = 0x10100;
24. /* Java card base segment data at the base segment data file named here*/
25. * (——.bss)
26. _bssEnd = 0x108FF;
27. }
28. }
```

注意：

在对指针、向量、数据集合和数据结构进行适当的地址分配后，设计出包括设备 I/O 地址的存储器映射。根据这个映射，很容易设计出定位器。设计者必须记住，如果主存是 Harvard 结构，程序存储器映射应该是单独的。例如 8051 通过单独的指令集(输入/输出指令)从存储器中读取数据。

2.5.4 内部设备和 I/O 设备在映射中的地址

所有的 I/O 端口和设备都有地址。它们根据系统处理器和系统硬件的配置分配给各个设备。下面的示例阐明了这种情况。下面考虑具有三个不同处理器的系统。

(1) 先来看一个具有 80960 嵌入式处理器的系统。图 2-7 说明，设备的地址处于分配给系统处理器地址和过程、例程的系统表、中断向量表的存储器地址之间。

(2) 考虑另外一个具有 68HC11 微控制器的系统。图 2-8(a)说明，设备地址在 RAM 中，并且与存储器地址截然不同。

(3) 考虑另外一个具有 80x86 微处理器的系统。图 2-8(b)的左边给出了存储器地址。图的右边给出了 IBM PC 系统分配给定时器、键盘、实时时钟和串口(称为 COM2)的端口地址。这个图说明，设备地址可以与存储器地址相同。此外这还依赖于系统硬件配置。

80960 和 68HC11 将 I/O 设备地址看作是存储器地址的一部分(参见图 2-7 和图 2-8(b))。图 2-8(a)给出了一种 68HC11 配置寄存器位的特殊情况。图中的配置表明，端口 A、I/O 控制寄存器 PIOC、端口 C、B 和端口控制寄存器的地址在 0x0000~0x0004 之间。片上 RAM 的地址被配置为 0x003F~0x0040 之间。端口地址和片上 RAM 可以通过 68HC11 中的配置寄存器位来配置。例如，可以对上述地址进行配置，并分配在 0x0100~0x1040 之间。另一方面，80516 和 8019 微控制器的内部设备地址是事先分配的，它们是不可配置的。

在 80x86 中，I/O 设备地址不是存储器地址的一部分(参见图 2-8(b))。

第 3.1 节和第 3.2 节将详细介绍 I/O 和其他的设备。设备地址是提供给驱动程序(driver)处理使用的(参见第 4 章)。一个设备具有一个地址，这个地址通常是根据系统硬件分配的，也可能是处理器分配的。这些地址分配给如下的设备：

(1) 设备数据寄存器(device data register)或者 RAM 缓冲区。

(2) 设备控制寄存器(device control register)。它保存控制位，也可以保存配置位。

(3) 设备状态寄存器(device status register)。它保存标志位作为设备的状态。一个标志位可以反映服务请求，并表示发生了设备中断。

每一个设备，也就是每一个设备寄存器必须在存储器映射中分配地址。有一点非常重要，在大多数情况下，每个 IO 设备地址空间都是由系统硬件确定的。定位器或者加载器不能够将它们重新分配给另外的地址空间。另外一点需要记住的就是，根据设备不同，在一个设备地址上可以有一个或者多个设备寄存器。物理或者虚拟设备可以通过配置与接收输入或者发送输出连接或者断开。设备地址也可以像一个文件，使之只读或者只写，或者可读可写。

示例 2-16 给出了一个 I/O 设备(串行设备(也称为 UART 设备))地址的详细介绍。

示例 2-16

串行设备(serial line device)具有如下的设备寄存器地址(参见第 3.1 节的图 3-1(b)了解关于 UART 信号和串行 I/O 格式的详细内容)。在使用 80x86 处理器的系统中，这些地址是由其 UART 端口接口电路硬件配置决定的。在 PC 中处于 COM1，地址是从 0x2F8~0x2FE。

(1) (A)两个 I/O 数据缓冲寄存器(一个用来接收，另外一个用来发送)具有一个公共的地址——0x2F8。假设地址 0x2FB 上的控制位为 0，(i)在从这个地址读取的过程中，处理器访问设备的 RBR(Receiver Data Buffer Register，接收数据缓冲寄存器)；(ii)在向这个地址写的过程中，处理器访问设备的 TRH(Transmitter Holding Register，发送保持寄存器)。RBR 和 TRH 的地址都是 0x2F8。(B)假设地址 0x2FB 的控制位为 1，除数锁存器(devisor latch)的两个字节的数据具有不同的地址——0x2F8(LSB)和 0x2F9(MSB)。除数锁存器保持了一个 16 位的数值，用来对系统时钟进行分频。这样就选择了串行发送的速率(然而当编写设备驱动程序的时候，要记住 2FB 上另外一个寄存器(控制寄存器)的一个位将 0x2F8 从 IO 寄存器(RBR 或者 TRH)改变为除数锁

存寄存器的低字节。参见第 4.3 节的示例)。

(2) 在写操作的过程中，设备的三个控制寄存器分别具有不同的地址——0x2FA、0x2FB 和 0x2FC。这些寄存器如下所述。(i)IER(Interrupt Enabling Register，中断使能寄存器)。它控制设备中断。(ii)LCR(Line Control Register，线路控制寄存器)。它定义了有多少位以及这些位在线路上的情况。(iii)MCR(Modem Control Register，调制解调器控制寄存器)。它定义了调制解调器如何进行“握手”和通信。

(3) 在读操作的过程中，设备的三个状态寄存器分别具有不同的地址——0x2FA、0x2FD 和 0x2FE。这些寄存器如下所述。(i)0x2FA 上的 IIR(Interrupt Identification Register，中断识别寄存器)。它具有标志位。标志位在设备中断的时候置位，当对相应的设备中断服务和系统复位的时候复位。(ii)0x2FD 上的 LCR。它定义了有多少位以及这些位在线路上的情况。(iii)0x2FE 上的 MCR。它定义了调制解调器如何进行握手和通信。

注意：

每一个 I/O 设备都具有不同的地址。每一个设备都有三个寄存器组：数据(缓冲)寄存器、控制寄存器和状态寄存器。在一个设备地址上可以有一个或者多个设备寄存器。设备的地址取决于系统处理器和系统硬件配置。多数的处理器采用相同的指令处理存储器设备和其他设备。设计者必须记住，80x86 处理器使用不同的指令集(输入－输出指令集)处理这些设备。

2.6 直接存储器访问

I/O 设备需要将其他系统的数据传送到系统中的存储器地址上。系统可能还需要将要传送到其他系统的数据传送到 I/O 设备上。当多字节数据或者一个数据块需要在两个系统之间、在 CPU 不干预(除了传送的开始和结束之外)的情况下进行传送时，需要直接存储器访问(Direct Memory Access，DMA)。

在 DMA 操作中通常支持三种模式。(a)一次传送一个字节，然后放弃系统总线。(b)一次进行 burst 传送，然后放弃系统总线。一次 burst 传送可能是几个 KB。(c)进行批传送，并在传送完后放弃系统总线。

DMA 传送是在 DMAC(DMA 控制器)的协助下进行的。由于使用 DMAC 只需要很少的处理器干预，I/O 设备和系统存储器之间的数据传送效率很高。在传送的过程中，系统地址和数据总线对于处理器来说是不可用的，而对于与 DMAC 相连的 IO 设备是可用的。例如，在计算机中，硬盘和系统存储器之间的数据传送使用的是 DMAC 通道。除了处理器之外，其他的设备也可以通过直接访问来获得系统存储器的使用权。图 2-10 给出了一个 DMAC。它还介绍了处理器、存储器、DMAC 和数据传送 I/O 设备之间的总线和控制信号。

DMAC 可以为多个通道提供存储器访问。对于每一个通道的编程，有单独的寄存器组。在多通道 DMAC 的情况下，会有单独的中断信号。一个多通道 DMAC 提供来自系统存储器和两个(或者多个 IO)设备的 DMA 操作。

80x86 处理器没有片上 DMAC 单元。8051 系列成员 83C152JA(及其姊妹版本 JB、JC 和 JD 微控制器)有两个片上 DMA 通道。80196KC 具有支持 DMA 功能的 PTS(Peripheral Transaction Server，外设事务服务器)。PTS 只支持单字节和块传送模式，而不支持 burst 传送模式。MC68340 微控制器具有两个片上 DMA 通道。80960CA 具有 4 个片上 DMA 通道，还提供了一种称为交

互传输的方式。

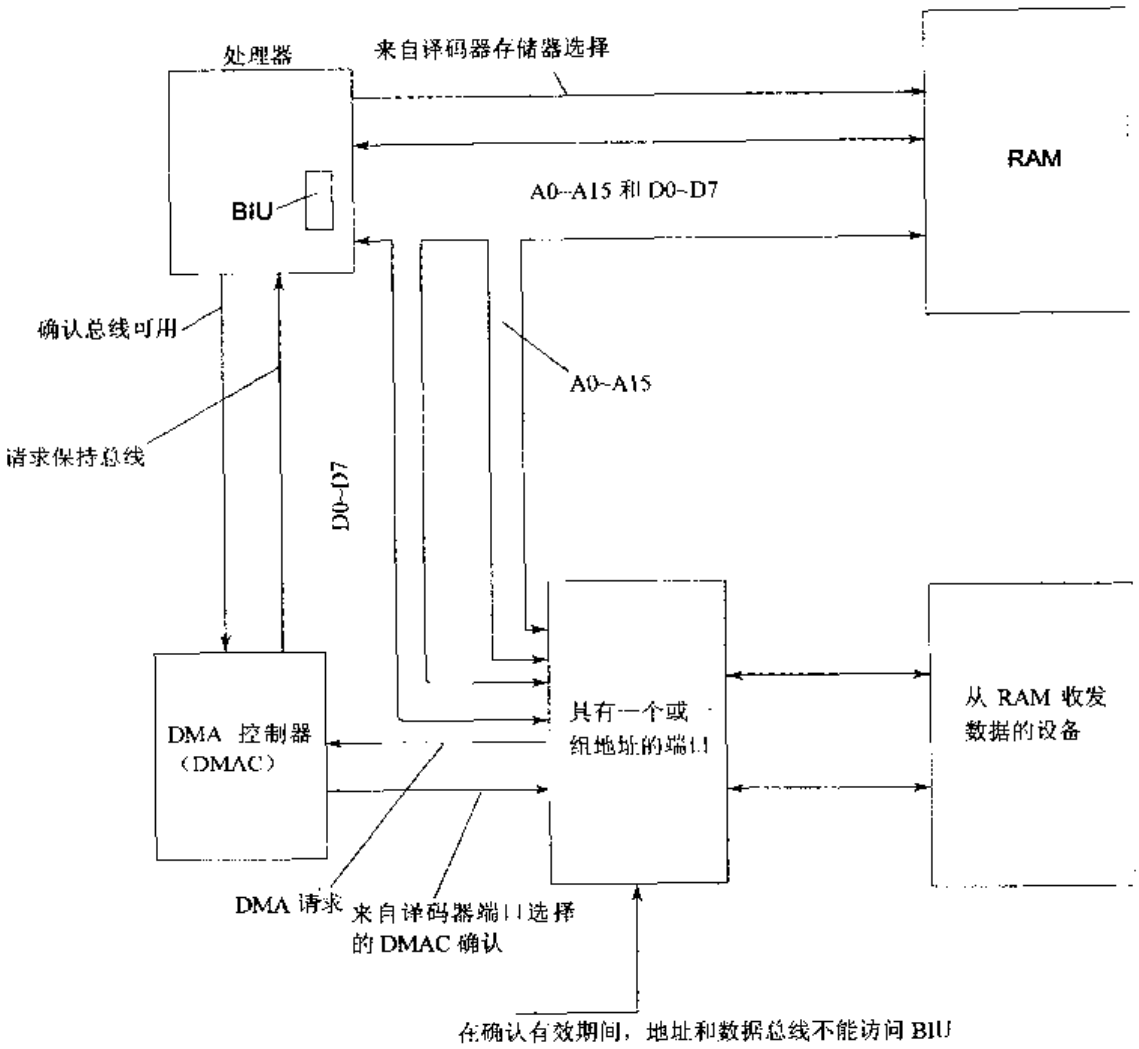


图 2-10　DMA 控制器以及其间的总线和控制信号

注意：

片上或者单独的 DMAC 帮助完成存储器和 I/O 设备之间的直接字节传送。设计者可以在复杂系统中使用 DMAC，使得通过单独处理外设的输入和输出过程，提高系统的性能。

2.7　处理器、存储器和 I/O 设备的接口

第 1.3.13 节介绍了使用胶合电路进行处理器、存储器设备和 IO 设备之间的总线接口。互连是通过数据和地址总线以及控制信号进行的。对总线信号时序图的理解是适当地设计接口电路和 GAL 或者 FPGA 中的熔断(烧制)的基本要求。图 2-11(a)和(b)分别给出了 8051 和 68HC11 中存储器设备和端口的接口电路。8051 微控制器使用一个附加的信号 $\overline{\text{PEN}}$ (由程序存储器的读操作产生的程序存储使能)，这是因为主存为 Harvard 体系结构。

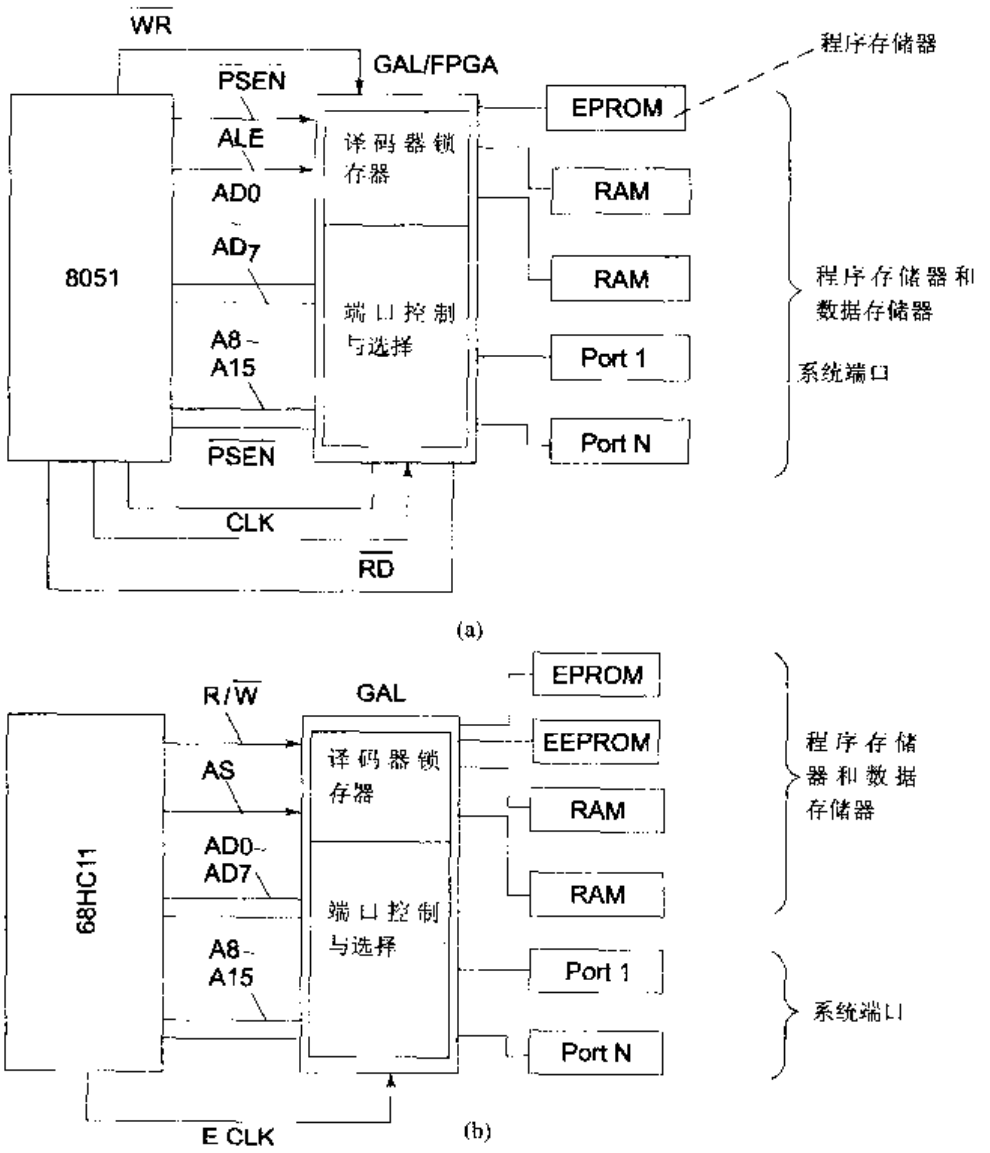


图 2-11 (a)使用 GAL 或者 FPGA 的 8051 中的存储器设备和端口的接口电路；(b)使用 GAL 或者 FPGA 的 68HC11 中的存储器设备和端口的接口电路

(1) 存储器中地址和数据总线的多路选择信号分离输出：分时复用技术(TDM)的意思是，在不同的时间段中，有不同的信号集合(通道)。到达(或者来自)存储器的地址和数据总线可以使用时间分割信号分离技术。在一个时间段中是地址信号，而在另外一个时间段中是数据总线信号。在 8051 中有一个称为地址锁存使能(Address Latch Enable，ALE)的控制信号。在 68HC11 中，控制信号是地址选通脉冲(Address Strobe，AS)。80196 中控制信号是地址有效(ADV)信号。ALE、AS 或者 ADV 对输入到设备的地址和数据总线进行信号分离。

(2) Harvard 结构的信号分离。在 8051 中，程序和数据存储器公用的地址总线信号分离是由 $\overline{\text{PSEN}}$ 完成的。当这个信号有效时(低)，到达程序存储器设备的地址信号有效。当 $\overline{\text{PD}}$ 有效时，到达数据存储器的地址信号有效。在 8051 中，信号分离电路和译码电路使用 $\overline{\text{PSEN}}$ 和 ALE。

(3) 译码器：每一个与处理器相接的存储器设备芯片或者端口都有一个单独的片选信号(CS)。译码器是一个电路，当输入端有正确的地址总线位时，产生相应的 CS 信号给每一个设备(存储器和端口)。

注意：

接口电路包含译码器和信号分离器，并按照控制信号和总线信号的时序图进行设计。这个电路连接了所有的单元、处理器、存储器设备和 IO 设备。它是系统中使用的胶合电路的一部分，是在 GAL 或者 FPGA 中设计的。

本章小结

- 嵌入式系统硬件设计者必须选择一个适当的处理器和适当的存储器集合，并设计处理器、存储器和 IO 设备之间适当的接口电路。这些都是在充分考虑了各种可用的处理器、结构单元和体系结构、存储器类型、大小和速度、总线信号和时序图的基础上完成的。
- 通过一个总线相连的处理器的结构单元有存储器地址和数据寄存器、系统和算术单元寄存器、控制单元、指令译码器、指令寄存器和算术逻辑单元。处理器中的寄存器(寄存器组)是非常重要的，对于处理过程中的各种功能都很有用。
- 高级处理器具有下列的附加结构单元：预取控制单元、指令排队单元、指令和数据高速缓存、分支转移、浮点寄存器和浮点算术单元。在高性能系统中，通常运用处理器中的流水线、超标量特征和高速缓存。MIPS、MFLOPS 或者每秒的 Dhrystone 定义了计算性能。设计目标是在最低的代价和最低的功耗和总体能量需求的条件下，提供最优的计算性能。
- 对于算术和逻辑指令、数据传输指令、I/O 指令和程序流控制指令，有一个指令集和各种寻址模式。CISC 和 RISC 处理器中的指令和数据格式以及寻址模式是不同的。每一个处理器只支持一个指令集。
- CISC 处理器的指令支持多种寻址模式。编程特征得到改善。对于多种类型的数据结构，程序员可以使用多种类型的指令。对于一条指令，有多种执行周期。
- RISC 处理器的指令只支持几种寻址模式，处理器性能较高，并实现单指令单执行周期。高级处理器中的 RISC 体系结构、超标量处理、流水线和高速缓存单元改进了处理器的性能，提供了快速的程序执行速度。
- 当各个函数或者任务共享一个变量的时候，就发生了共享数据问题。原子操作解决了共享数据问题。某些处理器中具有一个原子操作控制单元。
- 处理器的选择可以通过设计表来完成。
- 系统需要不同类型和大小的 ROM 和 RAM。ROM 的形式有掩膜 ROM、PROM、EPROM、EEPROM、闪存和引导区闪存。关于存储器的基本详细情况是可用地址、读写操作速度和存储器访问模式。
- 存储器具有提供给程序段、数据、堆栈和数组地址的地址块。存储器还保持了向量地址(指针)、引导程序指令集、任务指令集、中断服务例程和函数。在存储器种还有堆栈。设计者设计存储器分配映射(在使用 80x86 的情况下还要设计 IO 分配映射)。存储器的大小应该足够保存所有的代码、数据、数据集合和数据结构以及堆栈。
- 存储器访问速度应该与处理器结构和访问模式相匹配。
- 通过提供对 I/O 设备和外设的直接访问，使用 DMA 控制器可以提高系统的性能。
- 根据具有 I/O 设备地址的存储器映射，可以设计一个定位器，来定位链接的目标代码文件，并产生 ROM 映像。

- 存储器选择可以通过设计表来完成。
- 每一个 I/O 设备都具有不同的地址空间。每一个 I/O 设备还具有不同的设备寄存器组——数据寄存器、控制寄存器和状态寄存器。在一个设备地址上，可以有一个或者多个寄存器。设备地址是根据系统硬件确定的。
- 总线信号提供了处理器、存储器和设备之间的接口。接口电路充分考虑了处理器的时序图。电路使用处理器控制、地址和数据总线信号，并考虑了总线信号的时序图。基于 PAL、GAL 或者 FPGA 的电路提供了锁存器、译码器、多路器、信号分离器和其他需要的接口电路的单芯片解决方案。

关键词及其定义

- 内部总线(internal bus)：在处理器的各种内部结构单元之间并行传送信号的一组路径。在 64 位的处理器中，它的大小是 64 位。
- 存储器地址寄存器(memory address register)：为一个存储器单元保存地址的寄存器，通过使用总线接口单元，将这个存储器单元发送总线上。
- 存储器数据寄存器(memory data register)：保持传送给或者来自于一个存储器单元的数据的寄存器。
- 总线接口单元(bus interface unit)：将内部总线和外部总线进行互连的单元，以获取控制、地址和数据位。
- 系统寄存器(system register)：处理器寄存器。
- 算术单元寄存器(arithmetic unit register)：保存 ALU 使用的输入输出操作数和标志的寄存器。
- ALU：执行算术和逻辑操作指令的单元。
- 控制单元(control unit)：在一条指令的执行过程中，对所有的处理操作进行控制和排序。
- 指令寄存器(instruction register)：保持当前执行的指令的寄存器。
- 指令译码器(instruction decoder)：翻译操作码的电路，并根据译码结果指导控制单元。
- 预取控制单元(prefetch control unit)：从存储器单元中提前取指令和数据的单元。
- 指令排队单元(instruction queuing unit)：保持指令队列的单元，并将这些指令放置到高速缓存中。
- 指令缓存(instruction cache)：连续保存被基于超标量和流水线的并行处理预取过的指令的高速缓存。
- 数据缓存(data cache)：将数据以可寻址的模式存储的高速缓存。
- 分支转移缓存(branch transfer cache)：提前保持分支到将要被执行的程序中的下一个指令集的缓存。
- 共享数据问题(share data problem)：当一个变量被各个任务共享的时候，如果对该变量进行处理的任务还没有完成，该变量被另外一个任务指令修改，就会发生数据共享问题。
- 原子操作(atomic operation)：在代码的临界区中，它允许对一个共享变量进行操作的指令完成操作。
- 超标量处理器(superscalar processor)：具有同时并行地预取、译码和执行多条指令的能力的处理器。

- 流水线(pipelining)：在超标量处理器中也有流水线。它的意思是，它的 ALU 电路分成了 n 个子阶段。如果在某个瞬间它的第 p 条指令正在进行最后阶段的处理，那么在第一阶段中，正在进行对第(p+h)条指令的处理。在一个处理器中可以有多条流水线并行地处理。
- 存储管理单元(memory management unit)：管理存储器的预取、分页和分段的单元。
- 累加器(accumulator)：一个给 ALU 提供输入、并将来自 ALU 的结果操作数累加的寄存器。
- 程序计数器(program counter)：保持在总线的预取周期之后将要执行的当前指令地址的处理器寄存器。
- 堆栈指针(stack pointer)：保持着定义可用存储器地址的寄存器，处理器可以通过堆栈操作将寄存器和变量压入堆栈中，并从堆栈中将它们弹出。
- 堆栈(stack)：是一个存储器块，保持着 LIFO 数据传送模式下压入的数值。
- 队列(queue)：是一个存储器块，保持着 FIFO 数据传送模式下入列的数值。
- 链表(list)：是一种每一个对象都有一个指向下一个对象的指针的数据结构。第一个对象由链表头指向，最后一个对象不指向任何地方。
- 哈希表(hash table)：是一种关键字存储在表中的第 1 列、相关的数值存储在第 2 列的表。
- 索引寄存器(index register)：保持着数组、队列、表或者列表中一个变量的存储器地址的寄存器。
- 段寄存器(segment register)：指向程序代码、数据集合、字符串或者堆栈的段起始点的寄存器。
- 特殊功能寄存器(special function register)：8051 中用于累加器、数据指针、定时器控制、定时器模式、串行缓冲串行控制、断电控制、端口等特殊功能的寄存器。
- 操作码(opcode)：指令的第一个字节，用于处理器的指令译码。它定义了将要对操作数进行的操作或者处理。
- CISC：复杂指令集计算机，这种计算机有一个特征就是指令集庞大，允许对指令中的源和目的操作数进行多种模式的寻址。硬件执行指令的时钟周期数不同。它取决于指令中使用的寻址模式。
- RISC：精简指令集计算机，这种计算机有一个特征就是指令集精简，允许对指令中的源和目的操作数进行有限制的寻址模式。硬件执行每条指令的时间是一个时钟周期。
- 指令集(instruction set)：一个特定处理器的指令集合。
- ARM7 和 ARM9：来自于 ARM 公司和德州仪器，用于 SoC 的两种系列的 RISC 处理器，还具有单芯片和用于嵌入到 VLSI 芯片中的文件两种模式。ARM7 的主存是 Princeton 结构，ARM9 的主存是 Harvard 结构。
- PROM 或者 OTP：一种存储器类型，只能被设备编程器进行一次编程。OTP 是一次性可编程(One Time Programmable)存储器。
- EPROM：一种存储器类型，能够被紫外线进行多次擦除，并由设备编程器进行多次编程。
- EEPROM：一种存储器类型，每一个字节能够被多次擦除，然后由程序中的指令或者设备编程器进行编程。

- 闪存：一种存储器类型，每一个扇区能够被多次擦除，然后由程序中的写指令或者设备编程器进行编程。
- 引导区闪存(boot block flash)：是一种其中的几个扇区与 OTP 设备相同的闪存，能够存储引导程序和初始化数据。
- 存储器映射(memory map)：是一个存储器地址映射表，这个映射反映了处理器各种用途的可用存储器地址。存储器映射定义了系统的 ROM 和 RAM 地址。
- 设备(device)：物理或者虚拟单元，具有一个寄存器组——数据寄存器、控制寄存器和状态寄存器，处理器对它们进行与存储器相同的寻址。
- 设备地址(device address)：处理器用来访问设备寄存器组的设备地址。在每一个地址上可以有多个设备寄存器。
- 设备寄存器(device register)：设备中用于字节、数据字、标志或者控制位的寄存器。多个设备寄存器可以有一个公有地址。
- 时序图(timing diagram)：反映了外部总线信号相对于处理器时钟脉冲的时间间隔。
- 接口电路(interface circuit)：一种由锁存器、译码器、多路器和信号分离器组成的电路。
- DMA：内部或者外部控制器进行的直接存储器访问。DMA 操作协助外设和系统设备直接获取对相同存储器的访问，而不需要由处理器控制存储器块中的字节传送。

问题回顾

(1) 大多数处理器中的通用结构单元有哪些？

(2) 数码相机系统、实时视频处理系统、音频压缩系统、声音压缩系统和视频游戏的处理器中的特殊结构单元各是什么？

(3) 指令、数据和分支转移的高速缓存各有什么优点？

(4) 多路高速缓存单元能够使得只有包含执行一个指令子集所必要的数据的高速缓存单元的一部分被激活，这样做的优点是什么？列出具有多路高速缓存的 4 个处理器示例。

(5) 在何种情况下，系统中的处理器需要 MAC 单元？

(6) 解释三级流水线、超标量处理、分支、数据依赖代价。

(7) Harvard 结构的优点有哪些？与 Princeton 存储器结构相比，Harvard 存储器结构访问程序存储器中的堆栈和数据表为什么会比较简单？

(8) 说明处理器的三种性能优点：MIPS、MFLOPS 和 Dhrystone 每秒。

(9) 为什么应该将程序划分为函数(例程或者模块)，并且将每一个函数放置在不同的存储器块或者段中？

(10) 为什么应该将数据划分为数据类型和数据结构，并且将其中的每一个放置在不同的存储器块或者段中？说明下列的数据结构在存储器中如何存储：堆栈、向量、数组、循环队列、链表和查询表。

(11) 设备寄存器和设备地址的意思是什么？

(12) 引导区闪存与闪存的区别是什么？闪存和 EEPROM 以及闪存 EEPROM 的区别是什么？在嵌入式系统中，何时使用掩膜 ROM 保存 ROM 映像？何时使用闪存保存 ROM 映像？

(13) 参见 B.1 节。ARM7、ARM9、ARM11 和 StrongArm 的区别是什么？分别在什么时候使用 ARM7、ARM9 和 ARM11？

(14) 68HC12 和 68HC16 的区别是什么？什么时候会使用 68HC12？什么时候会使用 68HC16？

(15) 存储器映射如何帮助定位器的设计？Intel 和 Motorola 的 ROM 映像记录的格式各是什么？

(16) 下列术语的意思是什么？原子操作、burst 模式、PowerPC 特殊节电模式、密钥、片上 DMAC 和时间分割多路技术。

实践练习

(17) 一个 3×3 的矩阵与另外一个 3×2 的矩阵相乘。如果从一个寄存器到另外一个寄存器的数据传送需要 2ns，加法需要 20ns，乘法需要 50ns，执行时间为多少？假设这些时间在具有 MAC 单元的 DSP 中是相同的，MAC 将起到什么作用？

(18) 一个数组具有 10 个整数，每个整数都是 32 位的。令每个整数等于其在数组中的索引乘以 1024。令存储器中的基址为 0x4800。对于第 0 个、第 4 个、第 9 个元素，在(a)big-endian 模式和(b)little-endian 模式下将如何存储？

(19) 我们可以假设嵌入式系统的存储器也是一个设备。列出这样说的理由(提示：利用访问控制寄存器和虚拟文件和 RAM 磁盘设备的概念)。

(20) 在快速收发器中，参数化的块 RAM 和参数化的分布式 RAM 的优点有哪些？

(21) 现在高性能嵌入式系统或者使用 RISC 处理器，或者使用具有 RISC 核、代码最优化 CISC 指令集的处理器。为什么？

(22) 存储器中的循环队列具有 100 个字符，每个字符为 32 位。包括两个队列指针在内，总共需要多少存储器空间？

(23) 估计容量为 500 张图像的数码相机的存储容量需求，分辨率分别为(a)1024×768 像素，(b)640×480 像素，(c)320×240 像素(d)160×120 像素，每一个图像都以压缩的 jpg 格式存储。假设每一个像素颜色是由 24 位定义的。

(24) 数码相机系统、实时视频处理、音频压缩和视频游戏系统的处理器中的特殊结构单元是什么？

(25) 在存储器和 I/O 设备接口中，译码器的作用是什么？画出 4 个示例电路。

第 3 章　设备网络的设备和总线

本章前所学内容

下面将简要概述前两章已经学过的内容：

(1) 嵌入式系统的硬件包括处理器、存储器设备、输入/输出(I/O)设备和基本硬件单元，例如电源、时钟电路和复位电路。

(2) I/O 设备包含用来访问外设和其他片上或片外单元的 I/O 端口：例如 UART、调制解调器、收发器、时钟计数器、小键盘、键盘、LED 显示单元、LCD 单元、DAC、ADC 和脉冲拨号电路。

(3) 处理器高速缓存、流水线、超标量和其他高级处理单元是高计算性能处理器中的单元，高速计算过程中的功耗控制是由适当的处理器指令管理的。

- 我们还学习了处理器、存储器设备和 I/O 设备的组织结构；为了得到最佳的系统性能，如何选择适当的处理器和存储器设备；各种类型——根据大小和访问速度的不同——的存储器设备(ROM 和 RAM)；存储器和 I/O 设备中的地址和它们在系统中的寄存器；系统总线与存储器、I/O 设备的接口，使用 DMA，通过使能 I/O 单元对系统存储器的直接访问，来提高系统的性能。

我们是否可以想像没有视频输出、鼠标、键盘输入和磁盘存储的计算机？不能。因此对于嵌入式系统也是这样，这些设备都是必不可少的。系统 I/O 设备和时钟设备在任何一个嵌入式系统中都起着非常重要的作用。设备可能是内部的，可能是通过一个端口与系统相连并进行信息交互，每一个端口都分配了一个类似于存储器地址的端口地址。高级网络设备(例如收发器和加密解密设备)的操作速度都达到 MHz 或者 GHz。分布式设备通过系统中复杂的 I/O 总线进行网络互连。我们以汽车为例来分析。汽车上的多数设备都分部在不同的位置上。这些设备通过一条称为控制区域网络(CAN)总线的总线互连。

因此，设计嵌入式系统的硬件工程师必须清楚地理解新的复杂设备、接口电路的特征和它们的操作速度，以及进行设备网络互连的总线。

本章将学内容

在本章中我们将学习下面的内容：

(1) 设备、并行、串行输入/输出和 I/O 端口；

(2) 同步和异步串行设备及其示例，分别是高级数据链路控制(HDLC)和 UART；

(3) 微控制器的内部串行通信设备；

(4) 并口的特征；

(5) 用于高速 I/O、高速收发器和实时声音和视频 I/O 的复杂接口特征；

(6) 时钟和计数设备，将软件定时器作为虚拟时钟设备的概念；

(7) 多个分布式 IC 之间的嵌入式集成电路通信(I^2C)总线；

(8) CAN 总线作为汽车中分布式设备之间的控制网络；

(9) 用于嵌入式系统主机与分布式串行设备(如键盘、打印机、扫描仪和 ISDN 系统)之间的快速串行收发的通用串行总线(USB)；

(10) 主机或者系统和基于 PC 的设备、系统或者卡之间的 IBM 标准体系结构(ISA)和外设组件互连(PCI)/PCI-X(PCI 扩展)接口总线，如 PC 和网络接口卡(NCI)之间的 PCI 总线。

3.1 I/O 设备

3.1.1 I/O 设备的类型和示例

I/O 设备可以分为以下几种类型：(i)同步串行输入；(ii)同步串行输出；(iii)异步串行 UART 输入；(iv)异步串行 UART 输出；(v)并行一位输入；(vi)并行一位输出；(vii)并口输入；(viii)并口输出。有些设备可以同时具有输入和输出的功能，例如调制解调器。表 3-1 给出了 I/O 设备的分类，还给出了每一种类型的示例。

表 3-1 各种类型 I/O 设备的示例

I/O 设备类型	示 例
串行输入	音频输入、视频输入、拨号音、网络输入、收发器输入、扫描仪、远程输入和串行 I/O 总线输入
串行输出	音频输出、视频输出、拨号、网络输出、远程 TV 控制、收发器输出、多处理器通信和串行 I/O 总线输出
串行 UART 输入	串行线路上的小键盘、鼠标、键盘、调制解调器和字符输入
串行 UART 输出	串行线路上的调制解调器、打印机和字符输出
并口一位输入	(i)轮子旋转的完成；(ii)获取锅炉当前的压力；(iii)超过电子称托盘的最大重量上限；(iv)在机器人的胳膊能触及的地方或者附近到胳膊根部磁片的存在；(v)将液体装到容器中固定的水平面
并口一位输出	(i)DAC 的 PWM 输出，控制着液体水平面、温度、压力、速度、旋转轴的角度位置、一个对象的位移或者直流电动机控制；(ii)输出到外部电路的脉冲
并口输入	(i)来自液体水平面检测传感器、温度传感器、压力传感器、速度传感器或者直流电动机转速(rpm) 传感器的 ADC 输入；(ii)旋转轴角度位置位或者一个对象的线性位移的译码器输入
并口输出	(i)移动电话中的多道 LCD 显示阵列单元，用来在屏幕上显示电话号码、时间、信息、字符输出、位图、电子邮件或者网页；(ii)打印机或者机器人步进电动机线圈控制输出位

图 3-1(a)给出了输入串口、输出串口、双向半双工串口和双向全双工串口信号。半双工的意思是，在某一个瞬间，通信的方向只能有一个。例如，电话通信。在一条电话线上，我们只能以半双工方式通话。全双工的意思是，在某一个瞬间，可以在两个方向同时进行通信。

全双工通信模式的一个示例是调制解调器和计算机之间的通信。图 3-1(a)还以时间函数的形式给出了串行位格式和线路状态。图 3-1(b)给出了端口的握手信号和串行线设备上的 UART 串行位(参见示例 2-16)。此外还给出了当发送或者接收一个字符(字节)时，在 10~11 个周期之间作为时间的函数的逻辑状态。一个位周期也就是 8T=波特率的倒数，即 UART 的位变化频率。UART 位包括一个起始位，8 个字符位，一个可选的可编程位(P 位)和一个停止位。

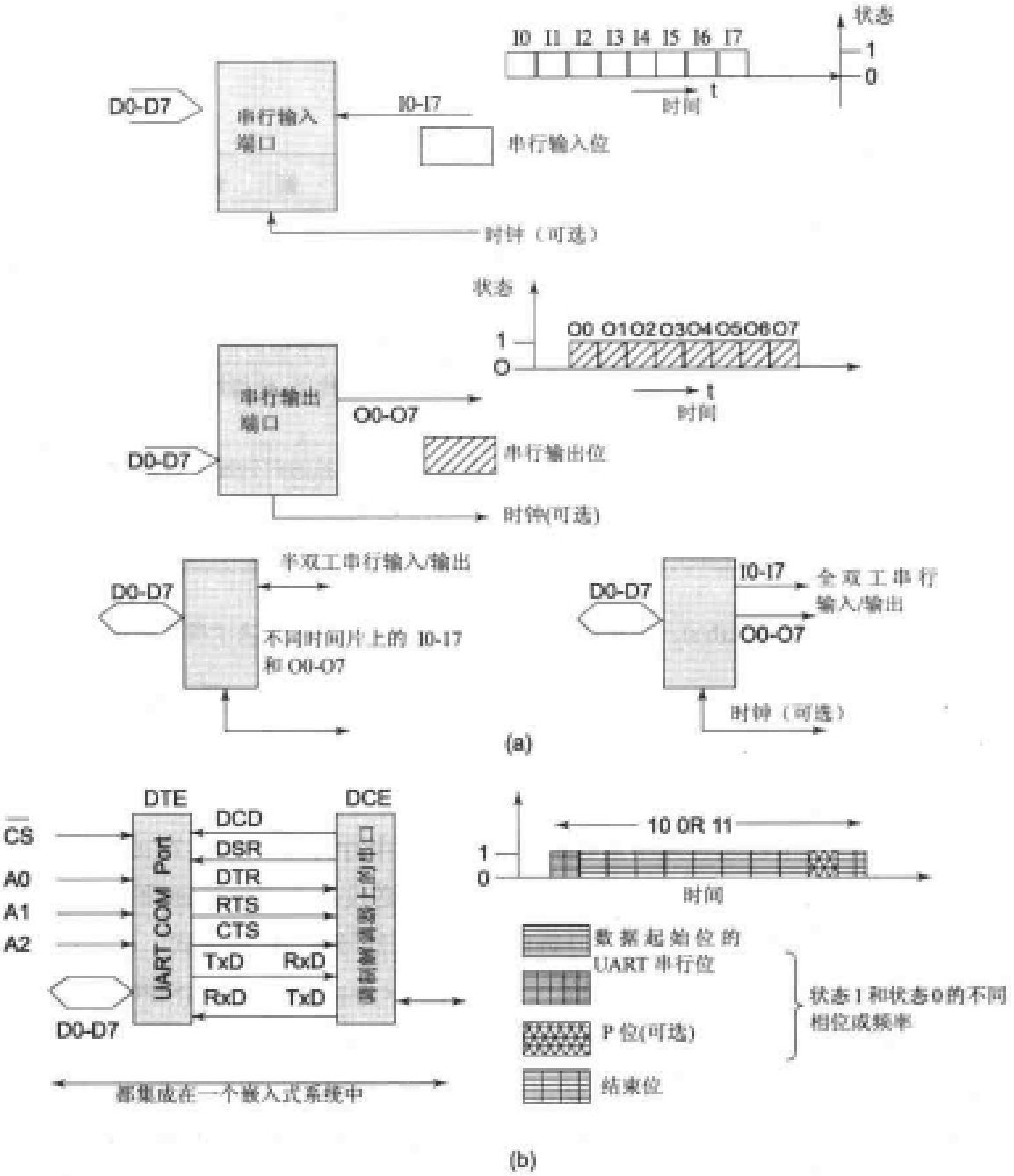


图 3-1　(a)输入串口、输出串口、双向半双工串口和双向全双工串口；(b)UART 串口使用的握手信号

注意：

串行设备(UART)通信的格式是 10 位或者 11 位。通信可以是全双工的，两个方向同时通信；也可以是半双工的，一个瞬间只有一个方向通信。半双工模式是一种很重要的通信模式。

3.1.2 串行设备的同步、准同步和异步通信

当以统一相位差在恒定时间间隔内接收或者发送一个字节(字符)或者一个帧(一组字节)的数据时，这种通信称为同步(synchronous)通信。数据帧的位是在一个固定的最大时间间隔内发送的。准同步通信是同步通信的一种特例，这种通信方式的最大时间间隔可以改变。

同步串行通信的一个例子是通过 LAN 发送数据帧。数据帧在进行通信时，每个帧之间的时间间隔保持恒定。另外一个例子是多处理器系统中内部处理器之间的通信。表 3-2 给出了一个同步端口设备的位。下面将介绍同步通信的两个特征。

表 3-2 同步端口设备位

编　号	端口上的位	强　制	解　释
1	同步编码位、二进制同步编码位或者帧起始和结束信号位	可选的	在数据位之前，有几个位(每个位之间都用时间间隔 ΔT 分开)作为用于帧同步或者信号的同步编码。在每个帧传输之后，代码位可能会翻转。在某些协议中也会使用起始和结束标志位
2	数据位	必须有	m 帧位或者 8 位传送，使得每一位在线上的时间都是 $m.\Delta T^1$
3	时钟位	通常是可选的	或者是在一个单独的时钟线上，或者是在同一条线上，时钟信息也通过适当的编码或者调制，嵌入到数据位中

1. ΔT 的倒数是每秒传输的位数(b/s)。注意：m 可能是一个很大的数。它取决于所使用的协议。

(1) 字节(或者帧)保持一个恒定的相位差。这就意味着它们是同步的，也就是处于同步状态。不允许在任意的时间间隔内发送字节或者帧；这种模式在通信过程中不提供握手机制。发送设备是主设备，接收设备是从设备。

(2) 必须总是存在一个以某种频率运行的时钟，用于串行传送所有字节(或者帧)的位。对于串行数据接收设备来说，时钟并不总是隐含存在的。发送设备通常将时钟频率信息在数据的串行通信中发送。图 3-2 给出了发送具有时钟信息的串行信号的 10 种模式。(i)数据位和时钟有两条单独的线，分别使用并行输入串行输出(PISO)和串行输入并行输出(SIPO)，进行发送和接收。(ii)数据位和时钟只有一条公用的线，可以通过调制将时钟信号编码为数据流。(iii)具有前缀和后缀附加同步和信号位。有五种方法可以将时钟信息编码为串行数据流。(a)频率调制(Frequency Modulation，FM)。(b)中频调制(Mid Frequency Modulation，MFM)。(c)曼彻斯特编码。(d)四分之一振幅调制(QAM)。(e)双相位编码。串行接收设备将信息的串行位进行分离，并与时钟同步。

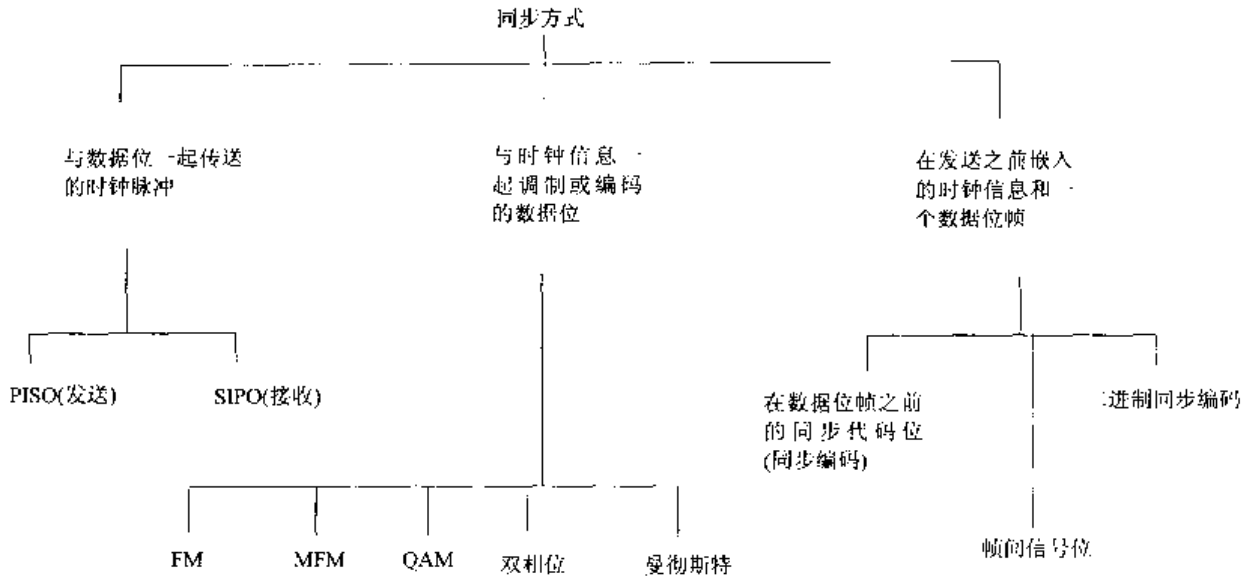


图 3-2　具有时钟信息的同步信号从主设备传送到从设备的 7 种方式

当在可变时间间隔内接收或者发送一个字节(字符)或者一个帧(一组字节)的数据时，这种通信称为异步(asynchronous)通信。异步通信的一个示例是电话，通话是在可变的时间间隔内发生的。线路上的声音数据是异步发送的。异步数据的另外一个示例是键盘和计算机之间的交互字符。UART 设备(DTE)和调制解调器(DCE)之间的通信也是异步通信。DTE 是“数据终端设备”的标准。DCE 是“数据通信设备”的标准。RS232C 是 DCE 和 DTE 之间接口信号的标准。

下面将给出异步通信的两个特征。

(1) 字节(或者帧)不需要保持一个恒定的相位差，也就是不是同步的。字节或者帧可以在可变时间间隔内发送。这种方式对于串行发送端口和串行接收端口之间的握手也很有帮助。

(2) 尽管时钟必须以某种频率运行，以便串行地传送单个字节(或者帧)的位，对于异步接收设备来说，这总是隐含的。在异步通信中，发送设备并不将串行比特流和时钟频率信息一起传送(既不是单独地，也不是通过使用调制进行编码)。因此，接收设备时钟不能保持与发送时钟相同的频率和相位差。

当一个设备用串行通信帧发送数据时，情况就不会像表 3-2 中描述的或者像 UART 设备(参见图 3-1)一样简单。这种情况很复杂，必须作为一个发送和接收设备都遵守的通信协议。下面来看一个示例。

当在网络上通过物理设备进行数据通信时，通常使用的是同步串行通信方式。HDLC(High Level Data Link Control，高级数据链路控制)是一个数据链接网的国际标准协议。它用来将数据从一个点链接到另外一个点或者是多点之间的数据链接。这个协议用于无线电通信和计算机网络中。它是一个面向位的协议。总的位数不一定是一个字节的整数倍或者是一个 32 位的整数。通信方式是全双工的。

表 3-3 给出了 HDLC 协议中的同步网络端口设备位。读者可以参考标准手册来获得关于 HDLC 及其域位的详细信息，例如由 Fred Halsall 编著、Pearson Education 在 1996 年出版的 *Data Communication,Computer Networks and Open Systems*。

表 3-3 基于 HDLC 协议的同步网络设备的位的格式

编 号	端口的位	必选或者可选	解 释
1	帧的起始和结束信号标志位	必须	起始和结束的标志位是(01111110)
2	目标设备的地址位	必须	标准格式是 8 位的，扩展格式是 16 位的
3a	控制位 情况 1：信息帧	对于情况 1、2、3 都是必须的	在标准格式中，第一位为 0，接下来是 3 位的 N(S)，下一位[1]是 P/F，最后 3 位是 N(R)；在扩展格式[&]中是 N(R)和 N(S)各 7 位
3b	控制位 情况 2：监控帧	-	在标准格式中，前两位为(10)，接下来两位[2]是 RR 或者 RNR 或者 REJ 或者 SREJ，下一位是 P/F，最后三位是 N(R)；在扩展格式[3]中是 N(R)和 N(S)各 7 位
3c	控制位 情况 3：未标序帧	-	前两位为(11)，接下来两位是[4]M，下一位是 P/F，最后三位是 M 的保留位。在扩展格式中，M 位后的 8 位是无意义的
4	数据位	必须	m 帧位传输，每一位在线上的时间都是 ΔT，每一帧在线上的时间都是 m. ΔT[5]。
5	FCS(Frame Check Sequence，帧校验序列)	必须	标准格式是 16 位的，扩展格式是 32 位的
6	帧结束标志位	必须的	结束的标志位也是(01111110)

1. [$]P/F = 1，P 的意思是，一个主站(命令设备)正在轮询检测从站(接收设备)。P/F = 1，F 的意思是，接收设备没有数据要发送。通常这是在最后一个帧中发出的。

2. RR、RNR、REJ 和 SREJ 传达的分别是“接收设备就绪”、“接收设备忙”、“拒绝”和“有选择地拒绝”四种信息。REJ 或 SREJ 是一种负的响应(NACK)。只有当拒绝接收帧的时候才会发出 NACK。“拒绝”的意思是接收设备接收到的帧是乱序的；这个帧被拒绝接收，从被 REJ 拒绝接收的帧开始，所有帧都要重复发送。“有选择地拒绝”的意思是，接收设备接收到的帧是乱序的；这个帧被拒绝接收，要对这个帧进行有选择的重新发送。

3. [&]N(R)和 N(S)表示接收的(先前)和正在发送的(当前)帧序列号。在标准格式和扩展格式中，分别是以 8 或者 128 为模。

4. ^M 5 位是发送设备给出的命令(或者响应)。命令的示例有复位、断开或者设置一个定义的模式类型。响应的示例有，来自于接收设备的表示断开模式接收、帧拒绝、命令拒绝的信息，以及表示一个未标序响应。

5. [@]当传送五个 1 的数据时，要增加一个附加的 0 作为填充位。这样能够避免接收设备将所接收到的数据错误地解释为标志位(01111110)。

6. 位是按照其发送或者接收的顺序给定的。

除了 HDLC 以外，通信系统还可以使用另外一种设备，用于同步或者异步地从一个设备端口传送信息。X.25、帧延迟、ATM、DSL 和 ADSL 是用于无线电通信和计算机网络中的联网物理设备的其他协议。以太网和令牌环是 LAN 网中使用的协议。应用协议的示例有 HTTP、

HTTPS、SMTP、POP3、ESMTP、TELNET、FTP、DNS、IMAP4 和 Bootp。嵌入式 Internet 设备使用应用协议和 Web 协议。WAP 是一种在无线网络中使用的协议。嵌入式系统设计者在嵌入式网络设备——例如桥和路由器——中使用这些协议。有兴趣的读者可以参考关于数据通信和计算机网络方面的标准参考书，了解关于这些协议的详细信息。

注意：

同步、准同步和异步是设备通信的三种方式。时钟信息显式地或隐含地以同步方式通信。接收设备时钟连续地与发送设备时钟保持恒定的相位差。HDLC 是计算机网络和无线电通信设备使用的重要的数据链路协议。UART 通信是异步的。

3.1.3 内部串行通信设备的示例

大多数的微控制器中都有内部串行通信设备。表 3-4 给出了几种微控制器中的片上串行设备的特征。

表 3-4 微控制器中具有片上串口(设备)的处理器

特　征	Intel 8051 和 Intel 8751	Motorola M68HC11E2	Intel 80196
同步串口(半双工或者全双工)	半双工	全双工	半双工
异步 UART 端口(半双工或者全双工)	全双工	全双工	全双工
UART 中每个字节的 10 位和 11 位的可编程性	Yes	Yes	Yes
同步和 UART 串口单独的非多路选择的端口引脚	NO	有(单独的 4 引脚)	NO
由软件或者硬件定义，将同步串口作为主设备或者从设备	软件	硬件和软件	软件
通过软件编程 P 位，将 UART 串口定义为主设备或者从设备	Yes	Yes	Yes
同步串口寄存器	SCON、SBUF 和 TL-TH 0-1	SPCR、SPSR 和 SPDR	SPCON、SPSTAT、BAUD_RATE 和 SBUF
UART 串口寄存器	SCON、SBUF 和 TL-TH 0-1(8052 中的定时器 2)	BAUD、SCC1、SCC2、SCSR、SCIRDR 和 SCITDR	SPCON；SPSTAT；BAUD_RATE 和 SBUF
使用内部定时器或者使用单独的可编程波特率发生器	定时器	单独的	既有单独的也有定时器

1. Intel 80960 和 PowerPC 604 没有内部串口。

2. 68HC12 提供 SIP(Serial Peripheral Interface，串行外设接口)通信设备，工作频率为 4Mb/s。在 68HC11 中，SPI 设备的工作频率可以达到 2Mb/s。68HC12 提供了两个 SCI(Serial Communication Interface，串行通信接口)通信设备，可以以两种不同的时钟频率工作。标准波特率可以设置到 38.4 Kb/s。在 68HC11 中，只有一个 SCI，标准波特率也只能设置到 9.6 Kb/s。

(1) 在 8051 中有一个片上通用硬件设备，类似于 USART。USART 的意思是“通用同步异步收发器”。在 8051 中这种设备称为 SI(Serial Interface，串行接口)。其特征如下：SCON(Serial Control Register，串行控制寄存器)是一个特殊功能寄存器。它保存 SI 的控制位和状态标志位，同时设置通信模式。SFR(Special Function Register，特殊功能寄存器)、SBUF(Serial Buffer，串行缓冲区)是一个串行缓冲区，在同步和 UART 通信中，都可以通过指令进行写或者读操作。不能对串行输出数据的时间间隔 ΔT 中的下降沿和上升沿的发生情况进行编程。SI 是双缓冲。双缓冲的意思是，在 SBUF 被读取的同时，一个中间寄存器保持所接收的最后一个帧的数据位。当读取 SBUF 的时候，中间寄存器传送 SBUF 的数值。当不读取 SBUF 时，中间寄存器接收到最后一个帧的数值后，就会发生过载异常。SI 具有以下两种操作方式：

i. 半双工同步操作模式，称为模式 0。当 8051 使用 12MHz 的晶振并与处理器相连时，时钟位之间的间隔是 1μs。

ii. 全双工异步串行通信，称为模式 1、2 或者 3。在模式 2 和 3 中，使用定时器，波特率根据定时器位的不同编程而不同。使用称为 PCON 的 SFR 上 SMOD 位，当使用模式 2 时，波特率只能被编程为两种频率。在 8051 中分别是晶振频率的 1/64 或者 1/32。当使用 11 位数据的传送格式时，T8 和 R8 提供第 10 位，用于异常检查，或者表示发送给从(接收)设备的数据类型是一个命令还是数据。

(2) 在 68HC11 中，有两个单独的硬件设备，分别用于串行和并行通信。这两个设备分别是 SPI(Serial Peripheral Interface，串行外设接口)和 SCI(Serial Communication Interface，串行通信接口)。

i. SPI 具有用于串行通信的全双工特性。SPI 的一个特性是可以对时钟的频率进行编程，因此也就可以在 68HC11 使用 8MHz 的晶振时，数据位的串行输出间隔最低为 0.5μs。还可以对 SPI 进行编程，定义有串行输入或者输出数据时，数据位之间间隔的下降沿和上升沿发生的情况。还可以定义从主设备输出到从设备的开路漏极和推拉输出电路，和定义通过硬件和软件将设备选择为主设备或者从设备。硬件方法是将主 SPI 设备的从设备选择管脚连接到“1”，而将相应的从设备管脚连接到“0”。

ii. UART 异步(SCI)波特率是相同的，不能单独为串行输入和输出线编程。对于 SCI 收发器，通信方式是全双工的。通过三个频率位和两个预分频器位，可以从 32 个支持的波特率中选择一个。SCI 接收设备的唤醒特征是可编程的。如果 RWU(Receiver wakeup Unavailable，收发设备唤醒不可用)(SCC2 的第一位，串行通信控制寄存器 2)被置位，唤醒特征被启用；如果 RWU 被复位，则唤醒特征被禁止。如果 RWU 被置位，从设备的收发器不能够被后面的帧打断。当编程的控制位是用于 11 位的格式时，T8 和 R8 提供内部处理器(主从)UART 通信。

(3) 在 Intel 80196 中，具有一个片上通用硬件设备，称为 SI。它类似于 UART。其特征如下：

在两次加载 14 位的 BAUD_RATE 寄存器后，可以对频率寄存器进行编程。不能对传送输出数据的时间间隔 dT 中的下降沿和上升沿的发生情况进行编程。80196 中串行接收器的缓冲区增加了一倍，同 8051 一样。

i. 只能进行半双工同步串行通信。位频率是由定时器 T2 时钟或者波特率寄存器确定的。位之间的最小间隔是 1.33μs(禁止将频率寄存器编程为全 0)。

ii. USART 电路(一种异步串行通信接口)还具有全双工 UART 模式。波特率可以通过定时器 T2 时钟或者 BAUD_RATE 寄存器进行编程(禁止将寄存器编程为全 0。)还有一个 PEN(奇偶使能)位(SPCON 的第 3 位)。将 UART 中 11 位格式的 P 位作为奇偶校验位，T8 和 R8 作为内部设备命令和数据。

注意：

微控制器中具有内部设备，用于同步和异步 UART 发送和接收。

3.1.4 并口设备

图 3-3(a)给出了三个设备各自的并行输入端口、输出端口和一个双向端口。图中还给出了具有处理器和系统总线的设备接口电路。可以从键盘输入控制器中输入数据 I0 到 I7。O0 到 O7 是输出到 LCD 显示输出控制器的串行输出位。Br_i 和 Br_0 分别是输入端口和输出端口的缓冲区。每一个端口都通过一个端口地址译码器与地址总线信号 A_i 和 A_j 相连。当处理器是 80x86 时，因为其具有 I/O 映射的 I/O，因此还有 $\overline{\text{IORD}}$ 和 $\overline{\text{IOWR}}$，分用于端口读和写的附加控制信号。它们与处理器中的存储器读写信号 $\overline{\text{RD}}$ 和 $\overline{\text{WR}}$ 相同，此时处理器中具有存储器映射的 I/O 组织。

图 3-3(b)给出了握手信号。外部输入发送设备发送一个查询请求，系统 I/O 设备发送应答信号(PORT READY)。当系统 I/O 设备发送缓冲区满(BUFFER FULL)信号时，外部输出接收设备发送应答信号(ACKNOWLEDGE)。当转换缓冲区空(可以进行下一次写操作)或者当接收缓冲区满(可以进行读操作)时，就给处理器发送一个中断请求(INTERRUPT REQUEST)信号。

当与一个设备端口进行接口时，要考虑下列特性。

(1) 一个端口设备可能会有多字节数据输入缓冲区和数据输出缓冲区。假设有一个 8 字节的缓冲区。假设一个设备(像 80196 微控制器中的设备)可以产生三个中断，一个是接收到一个字节时，一个是接收到第 4 个字节时，一个是当缓冲区满时。在一个字节或者一个字的缓冲区的情况下，对这些中断服务的最后期限就会增加到 8 倍。

(2) 一个端口可以有一个 DDR(数据方向寄存器，Data Direction Register)(像 68H11 微控制器中的端口)。这是一个很大的优点，因为现在端口的每一个位都是可编程的。它可以被设置为输入或者输出。DDR 可以对端口位进行编程。

(3) 端口 LSTTL 驱动能力和端口负载能力都是重要的特性。一个端口可能是一个 OD(漏极开路)端口。其驱动能力为 0。如果所给的端口具有 OD 门，就要将每一个门连接上适当的上拉电阻或者晶体管，使得门具有一定的驱动能力。

(4) 如果所给的门是一个准双向(quasi bi-directional)(像 80196 中的端口)端口，则在一个或者多个时钟周期内，对于一个或者多个 LSTTL 门，这个端口具有有限的驱动能力。当这个设

备端口要与一个以上的 LSTTL 相连时，每一个门都要连接合适的上拉电路。

(5) 端口管脚上会有多种或者可选的功能，例如 80196 的输入端口管脚。P2 的每一个管脚都可以用作 8 个模拟输入的多通道模拟输入设备。另外一个例子是 8051 系列的两个端口，简写为 P0 和 P2。当内部存储器空间不够，需要内部多路选择的总线用于外部程序时，这些端口位还可以作为输出使用。8051 中 P3 的每一个管脚都有多种用途。它们在串行通信过程中，可以用作定时器/计数器信号、中断信号，以及外部存储器的 $\overline{RD}$ 和 $\overline{WR}$ 控制信号。

68HC11 的端口 B 和 C 都具有 8 位，端口位有多种用途。一种用途是分别输出内部地址和数据总线。

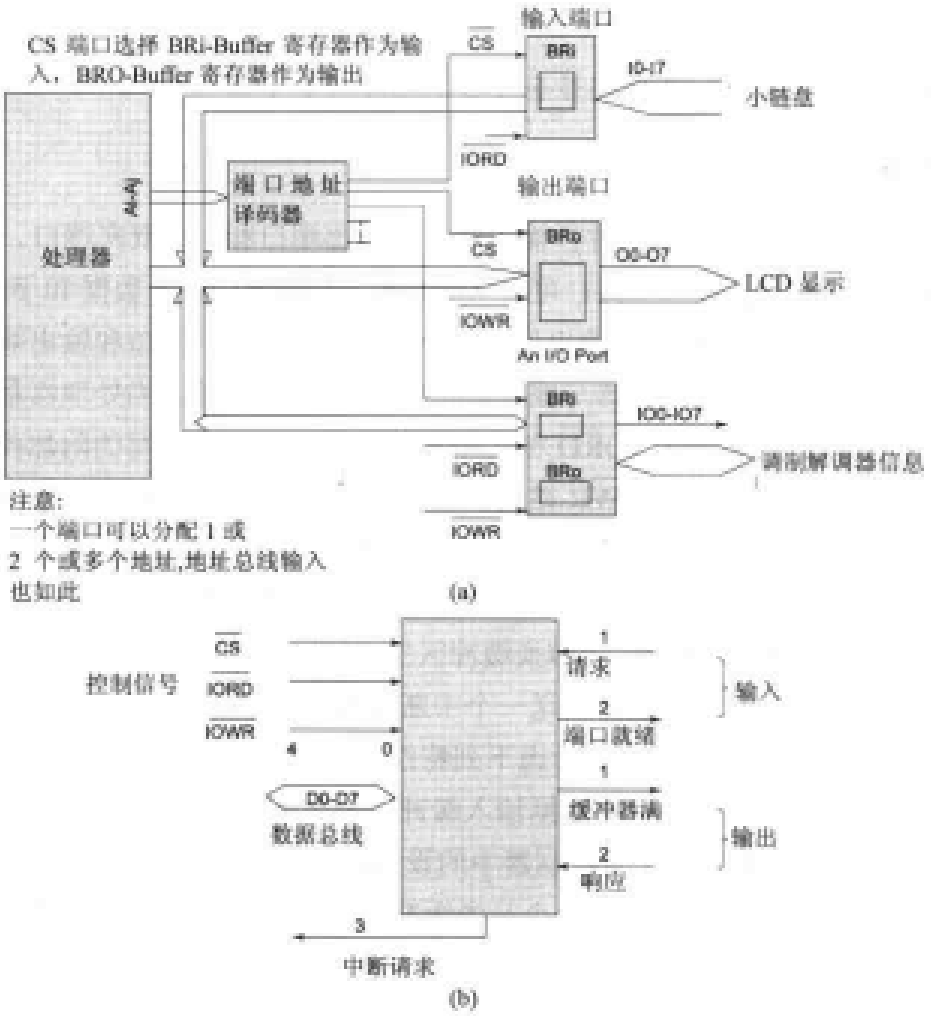


图 3-3 (a)连接设备的并行输入端口、输出端口和双向端口；(b)I/O 端口使用的握手信号

(6) 一个端口应该有用于连接多个系统或者单元的多路选择输出的预留管脚。

(7) 一个端口应该有用于连接多个系统或者单元的多路选择输入的预留管脚。

注意：

并口设备有几点需要注意，当与并口进行接口时，必须将这些事项考虑在内。

3.1.5 设备端口的复杂接口特性

一个端口设备可能不会像步进电动机或者串行 UART 设备一样简单。现在，复杂嵌入式系统具有非常复杂的 I/O 设备。例如，快速 I/O 的 I/O 设备、快速串行化和反串行化、快速收发器和实时视频处理 I/O。下面是复杂的接口特征。

(1) 令操作电压的逻辑状态 1＝5V(TTL 或者 CMOS)。施密特触发器电路具有一个特性，当发生从 0 到 1 的转换时，如果电压值超过了 5V 的 2/3，则发生到 1 的转换。类似地，当发生从 1 到 0 的转换时，如果电压值低于 5V 的 1/3，则发生到 0 的转换。因此，当设备的输入端为双电压时，施密特触发器电路消除的噪声最大为 5V 的 2/3＝3.3V。端口内置施密特触发器电路的一个最大优点就是对信号的噪声消除作用。否则，端口设备输入将需要一个基于施密特触发器的噪声消除芯片。这样的设备对于中继系统的收发器相当有用。在远距离通信时使用施密特触发器电路。

(2) 当一个端口设备处于等待指令的状态时，可以对设备的门进行电源管理。最近，已经开发出了一种在端口中使用的称为 DataGate(来自于 Xilinx)的新技术。DataGate 是一个用于电源管理的可编程 ON/OFF 开关；DataGate 可以通过在不使用时减少输入端不必要的翻转来降低功耗。端口设备中类似于 DataGate 的内置电路的最大优点是，当端口设备高速操作时，能够降低功耗。这种设备对于与通用总线相连的系统来说是相当有用的，因为这种情况下需要对不必要的输入切换进行控制。例如，在一个总线接口单元中，只有当输入数据发送给电路时，输入信号才有必要激活。随着系统中总线接口数目的增加，对禁止输入信号不必要的切换需求也随之增加。

(3) 在早期，端口接口是开路漏极 CMOS、TTL 或者 RS232C。(i)现在，系统的操作电压可能需要低于 5V(参见 1.3.1 节)。在使用 1.5V I/O 的端口设备中，可以使用 LVTTL(低电压 TTL)和 LVCMOS(Low Voltage CMOS，低电压 CMOS)门。(ii)现在，系统可能需要使用先进的 I/O 标准接口进行操作。示例有 HSTL(High Speed Transreceiver logic，高速收发器逻辑)和 SSTL(stub-series Terminated Logic，线脚系列终端逻辑)标准。HSTL 用于高速操作。当总线需要与相对较大的线脚相互独立时，需要使用 SSTL。

(4) 当一个设备与其他的设备进行网络互连时，该设备与系统总线以及 I/O 总线相连。I/O 操作过程中的设备阻抗和总线阻抗需要匹配。否则就会发生线反射。最近产生的新技术能够使得这些匹配自动完成。例如，可以使用一种称为 XCITE(Xilinx 控制阻抗技术)的新兴技术。用于自动阻抗匹配内置设备的最大的优点是，当电阻被设备中的数字自动控制和匹配的阻抗代替时，就不会产生线反射，也就不会发生位丢失或者总线故障。

(5) 一个 I/O 设备可能会包含多个 G 比特(622 Mb/s~3.125G b/s)收发器(MGT)。在这个频率下需要特殊的支持电路。Rocker I/O™ 串行 3.125G b/s 收发器就是这种电路的一个示例，能够提供这种频率的支持电路。

(6) I/O 设备可以集成一个 SerDes(serialization and De-serialization，串行化和反串行化)子单元。在字节保存在“发送保持缓冲区”中并串行发送的设备中，SerDes 是一个标准的子单元，一旦接收数据后，进行反串行化并存入到“接收缓冲区”中。一旦配置了 SerDes 后，串行化和反序列化就可以在不使用处理器指令的情况下自动完成。SerDes 的最大优点是，它提高了操作速度。当串行化的时候，I/O 设备可以集成一个 DAA 或者 McBSP 子单元(参见 D.4.4 和 D.4.5)。

(7) 最近，已经制定了多个用于 I/O 设备的 I/O 标准。在某些嵌入式系统中可能会需要支

持多个 I/O 标准。一种称为灵活选择 I/O 的技术支持超过 20 种的单端模式和差动模式的 I/O 信号标准。支持多种标准的设备优点显而易见。

(8) I/O 设备可以集成一个数字物理编码子层(Physical Coding Sub layer，PCS)。那么模拟视频和音频信号可以在子层中进行脉冲代码调制(PCM)。PCS 子层直接提供了来自于设备自身的模拟输入编码。然后这些编码被保存在设备数据缓冲区中。端口设备中内置 PCS 的优点是不需要进行外部 PCM 编码。此外这些操作是在后台进行的，速度也很快。它提高了用于多媒体输入设备的系统性能。

(9) I/O 设备可以集成一个模拟物理介质接入(PMA)单元，用于连接声音、音乐、视频和图形的直接输入和输出。内置 PMA 的最大优点是，设备与物理介质直接相连。在实时处理视频和音频输入的设备中，需要 PMA。

注意：

现在，I/O 设备具有复杂的特征。施密特触发器电路用于噪声消除。具有低电压门的设备和通过禁止输入端的不必要切换进行电源管理的设备，用于复杂的应用中。动态控制的阻抗匹配是一种新兴技术，当与设备进行接口时，能够消除线反射。设备中的 SerDes 子单元能够对输入和输出进行串行化和反串行化。一个端口可以包含 PCS 和 PMA 子单元，用于音频和视频 I/O 设备的模拟输入。

3.2 定时器和计数设备

我们是否能认为像电视遥控器或者洗衣机这样简单的系统没有一个定时设备？不能。对于嵌入式系统也同样如此。定时器设备是相当复杂的。它具有几个状态(参见表 3-5)。

表 3-5 定时器中的状态

编　号	状　态
1	复位状态
2	初始加载(空闲)状态
3	当前状态
4	溢出状态
5	过载状态
6	运行(激活)或者停止(阻塞)状态
7	结束状态
8	复位使能/禁止状态
9	加载使能/禁止状态
10	自动重装使能/禁止状态
11	服务例程使能/禁止状态

表 3-6 列出了一种定时器设备的 12 种应用。表中还解释了每一种应用的含义。

表 3-6 定时器设备的使用

编 号	应用和解释
1	实时时钟数(系统心跳)。(实时时钟是这样一个时钟，一旦系统开启之后，就不会停止，并且不能被复位，并且其计数数值不能被重装。实时时钟永远不停地运行，并永远不能返回！)
2	延迟一定时间后，初始化一个事件。延迟是作为一个计数数值加载进去的。
3	当比较设定时间和计数数值之后，初始化一个时间(两个时间或者一个时间链)。设定时间加载在一个比较寄存器中(它类似于设定闹钟)
4	获取一个事件发生时的定时器计数值。时间信息(事件的实例)保存在采集寄存器中
5	找出两个事件之间的时间间隔。每一个事件发生时获取时间，然后计算出两个时间之间的间隔
6	当使用 RTOS 时，等待用于设定时间的来自一个队列、邮箱或者信号量信息。
7	Watchdog 定时器。当超过定义的时间后，系统复位
8	用于线路或者网络中串行通信的波特率或者位频率控制。定时器超时中断定义每一个波特的时间 δT 或者每一位的 ΔT
9	当使用定时器时的输入脉冲计数，定时器是由非周期性的输入驱动的，而不是由时钟输入驱动的。如果代替时钟输入，对于每一个要计数的实例，将输入发送给定时器，则定时器就作为计数器工作
10	各种任务的调度。软件定时器中断链和 RTOS 使用这些中断来调度任务(关于任务的定义参见 8.1.2 节)
11	各种任务的时间片管理。当经过设定的时间延迟之后，RTOS 从一个正在运行的任务切换到下一个任务。那么每一个任务都可以在预先定义的时间段内运行
12	分时复用(Time division multiplexing，TDM)。定时器设备用来从几个通道中多路选择输入。每一个通道输入都分配了一个专用的并且固定的时间段获得 TDM 输出。例如，多个电话呼入是输入，TDM 设备产生 TDM 输出，用于将这个输出发送到光纤中

系统中至少有一个硬件定时器设备，用作系统时钟。硬件定时器从来自处理器的时钟输出信号获得输入，并且每当经过设定在硬件定时器中的 numTicks 之后，激活系统时钟(系统中断发生之前所需要的系统时钟嘀哒数＝numTicks)。

图 3-4 给出了硬件定时器控制位(和信号)和状态标志位。控制位作为硬件信号，与控制寄存器中的位对应。控制位(或者信号)可以分为 9 种类型，分别是：(i)定时器使能(激活定时器)。(ii)定时器启动(每当时钟输入时开始计数)。(iii)在下一个时钟输入到来时停止定时器(停止计数)。(iv)预伸缩位(划分来自处理器的时钟输出频率)。(v)向上计数使能(通过每次时钟输入时增加计数值来使能向上计数)。(vi)向下计数使能(每当有时钟输入就减小计数值)。(vii)加载使能(使寄存器种的一个数值加载到定时器中)。(viii)定时器中断使能(当定时结束(溢出)并且计数值＝0 时使能中断服务)。(ix)定时器中断使能(当定时器溢出(计数值＝0)时启用中断服务)。

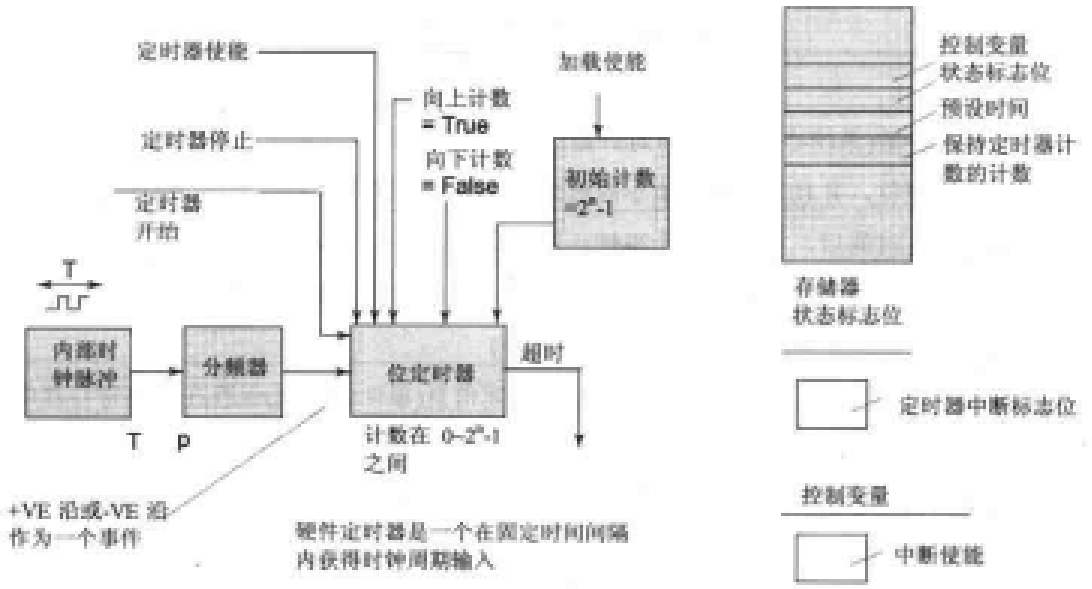


图 3-4 硬件定时器设备中的寄存器或者存储器的信号、时钟输入、控制位和状态标志位

状态标志位反映的是硬件定时器相应的硬件信号超时的情况。当定时器的所有位(计数值)都为 0 时，这个标志位和信号被置位。

表 3-7 列出了表 3-6 中定时器使用的 10 种形式。软件定时器(Software timer，SWT)是一个新的概念，是一种虚拟定时设备。

表 3-7 定时器的 10 种形式

编　号	类　型
1	硬件内部定时器
2	软件定时器(SWT)
3	用户软件控制硬件定时器
4	RTOS 控制硬件定时器。RTOS 可以定义为系统中的硬件定时器，每秒嘀哒一次
5	具有周期性时钟溢出事件的定时器(溢出状态后自动重装)。定时器可以是可编程的，每当超时后自动重装
6	One shot 定时器(溢出和终止状态后不再重装)。它触发用于激活它的事件输入，从空闲状态开始运行。它还用来增强两个状态或者时间之间的时间延迟
7	向上定时器。每当从时钟传来一个计数输入时，定时器的计数值增加
8	向下定时器。每当从时钟传来一个计数输入时，定时器的计数值减小
9	具有溢出标志的定时器，当中断服务例程开始运行时，自动复位
10	具有溢出标志的定时器，不能自动复位

系统时钟或者任何其他的硬件定时设备周期性地产生一个中断或者一个中断链。这个周期也就是计数值置位的周期。现在，中断作为 SWT 的一个时钟输入。这个输入对于被激活 SWT 列表中的所有 SWT 是公有的。列表中任意数量的 SWT 都可以被激活。每一个 SWT 在超时(计数值为 0)后都将设置一个状态标志。图 3-5 给出了 SWT 中的控制位和状态位。每次应用时，

都要将 SWT 的控制位置位。这些控制位总共有上述的 9 种。SWT 中没有硬件输入或者输出。当 SWT 的计数值到达 0 后，就会将一个标志位置位。表 3-8 列出了 SWT 的所有变量，包括控制位和状态标志位。因此 SWT 具有类似于硬件定时器或者计数器中的控制变量和标志位。

SWT 的行为与硬件定时器的行为是可比拟的。系统中的硬件定时器数量是有限制的(1、2、3 或者 4)，然而 SWT 的数量是没有限制的。某些处理器(微控制器)甚至定义了 2 个或者 4 个 SWT 中断向量地址。

表 3-8　软件定时器中的控制位和状态变量

编　　号	32、16、8 或者 1 位变量
1	复位数值 32/16/8
2	初始加载数值(NumTicks)32/16/8
3	计数数值(当前数值)32/16/8
4	最大数值 32/16/8
5	最小数值 32/16/8
6	定时器运行使能位
7	定时器中断使能位
8	定时器复位使能位
9	定时器加载使能位
10	定时器重装(结束状态后)使能位
11	溢出标志

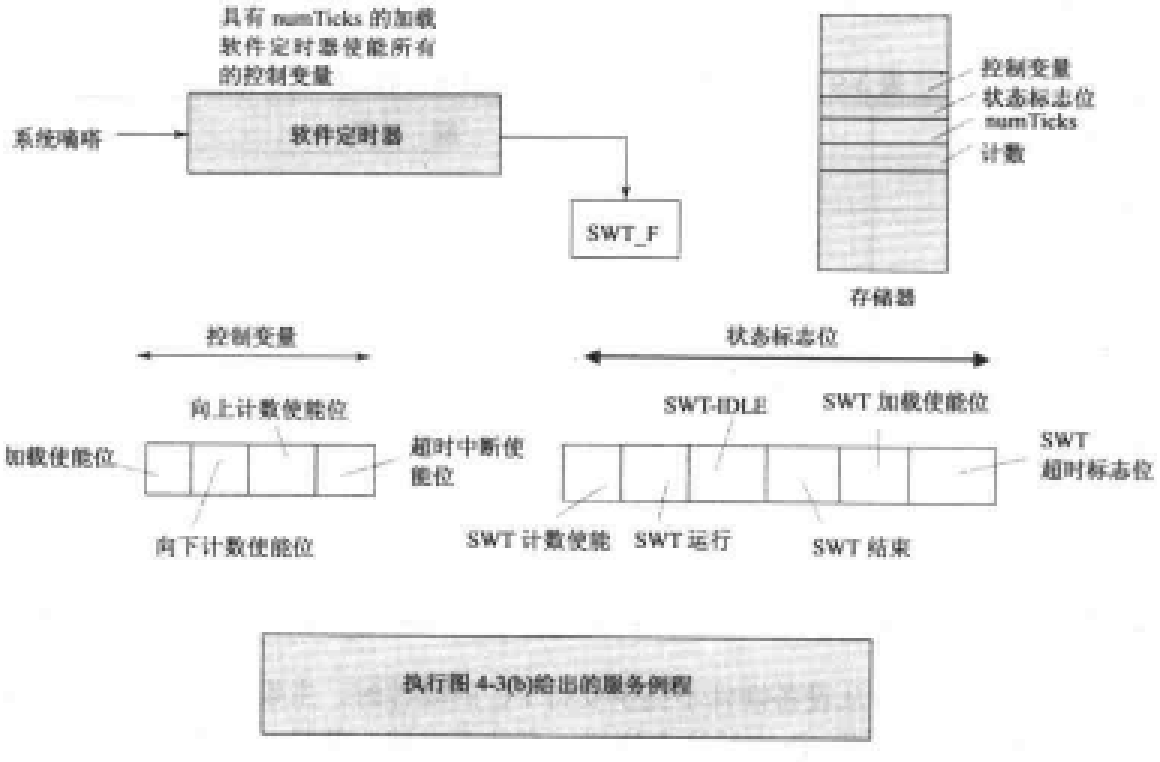


图 3-5　软件定时器的控制位、状态标志和变量

注意：

嵌入式系统中的许多应用都需要定时设备。(i)系统中的硬件定时器数量是有限的。一个系统至少有一个硬件定时器。系统时钟就是由硬件定时器配置得到的。一个微控制器可以有 2、3 或者 4 个硬件定时器。一个硬件定时器是根据来自处理器的内部时钟的输入运行的，并产生系统时钟。使用系统时钟或者内部时钟，可以驱动当前的硬件定时器。这些定时器可以通过设备驱动程序进行编程。(ii)软件定时器是一种软件，根据定时器输出中断或者根据实时时钟中断增加或者减小计数变量的数值(计数值)。软件定时器用作虚拟定时设备。每一个定时器设备中都有几个控制位和一个超时状态标志。当定时器的输入是计数输入，而不是时钟脉冲时，定时器设备就可以作为计数设备工作。

3.3 互连的多个设备之间通过 I^2C、CAN 和高级 I/O 总线进行串行通信

3.3.1 I^2C 总线

假设工厂的几个工艺过程中有多个设备电路，一些用于测量温度，一些用于测量压力。这些 IC 如何通过一个通用总线进行网络互连？I^2C 总线已经成为这些设备互连的标准总线(总共有三个标准：工业 100Kb/s I^2C、100Kb/s SM I^2C 和 400Kb/s I^2C)。

I^2C 总线使用两条线来传送信号——一条线用于时钟信号，一条线用于双向数据信号。I^2C 总线有一个协议。图 3-6(a)给出了使用 I^2C 总线传送一个字节的过程中的信号，图 3-6(b)给出了 I^2C 总线位的格式。

主设备的数据帧具有表 3-9 所示的域。

表 3-9 基于同步 HDLC 协议的网络设备的位格式

域 和 长 度	解 释
第一个域：1 位	起始位，类似于 UART
第二个域：7 位	称为地址域。定义从设备地址，正是主设备发送数据帧(或者多个字节)的对象
第三个域：1 个控制位	定义了正在进行读周期还是写周期
第四个域：1 个控制位	定义了当前数据是否是一个应答信号
第五个域：8 位	用作 IC 设备数据字节
第六个域：1 位	一个 NACK 位(负响应)。如果该位有效，则数据传送结束后不需要从设备给出应答，否则从设备要给出应答
第七个域：1 位	一个结束位，类似于 UART

这种总线的缺点是，当从设备硬件不提供支持 I^2C 的硬件时，主设备需要花费时间来分析通过 I^2C 总线传输的位。有些 IC 支持这个协议，有些不支持。并且，主设备中还有开集极驱动程序。因此每一条线上必须有一个 2.2K 的上拉电阻。

注意：

I^2C 总线是一个 IC 间互连的串行总线。它与 UART 类似，也具有一个起始位和一个结束位。共具有七个域：起始域、7 位地址、定义读或者写、定义一个字节作为响应字节、数据字节、NACK 和结束域。

3.3.2　CAN 总线

汽车中分散着多个设备。它们进行网络互连，并且通过一个网络总线进行控制。“CAN”总线是分布式网络的一个标准总线。它主要应用在汽车电子中(关于这个标准以及嵌入式系统在汽车电子中的应用，请参见 11.3 节)。它具有一个双面串行线。它以最高为 1Mb/s 的速率在瞬间接收或者发送一个位。它通过一个双绞线与每一个节点相连。这个双绞线的最长工作长度为 40m。

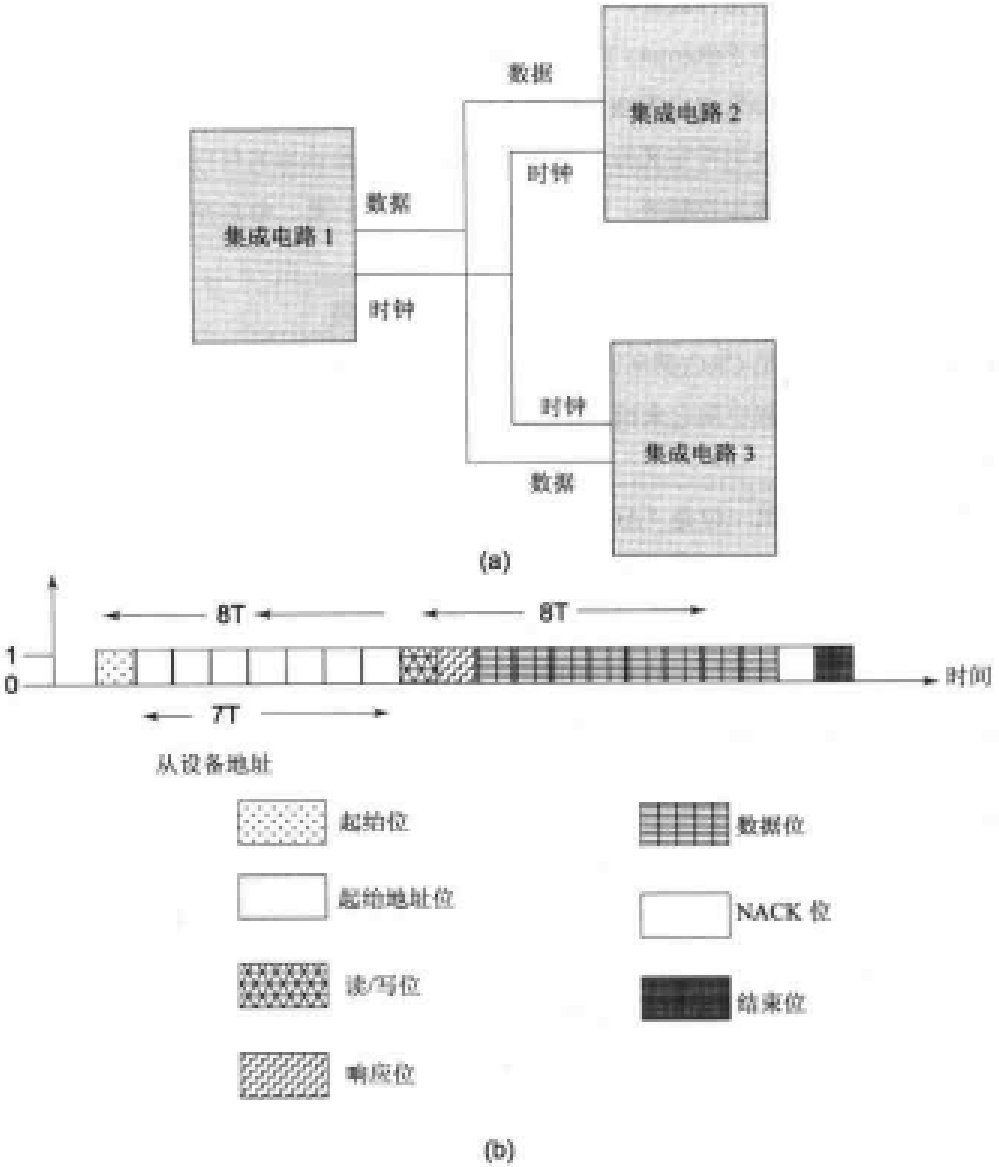


图 3-6　(a)当使用 I^2C 总线时字节传输过程中的信号；(b)I^2C 总线的位格式

(1) CAN 串行线被电阻上拉到逻辑 1，电压为+4.5V ~ +12V。在空闲状态时，线处于逻辑 1 状态，也称为“高阻状态”。

(2) 每一个节点在输入管脚和 CAN 串行线之间都有一个缓冲门。在任一个瞬间，当发现线被下拉到了逻辑 0 时，节点从线上获得输入。逻辑 0 状态被称为“隐性状态”。

(3) 每一个节点在输出管脚和串行线之间都有一个电流驱动电路。节点通过其驱动电路将线下拉为 0，并瞬间向线上发送输出。NPN 晶体管是电流驱动管，其发射极与地线相连，集电极与线相连。通过使用驱动电路(包含一个与 NPN 的基极相连的缓冲反向器)，节点可以将线下拉为 0，否则线处于逻辑 1 的空闲状态。这样就使得其他节点能够发现输入。

(4) 节点以数据帧的形式发送数据。数据帧总是以 1 开始，以 7 个 0 结束。在两个数据帧之间，最少有三个域。表 3-10 给出了 CAN 帧中每一个域的详细描述。

表 3-10　CAN 帧中的每一个域

域及其长度	解　　释
第一个域：12 位	称为仲裁位，包含打包的 11 位目标地址和 RTR 位。RTR 表示的是远程发送请求(Remote Transmission Request。当该位为 1 时，表示这个包是目标地址；如果该位为 0(支配状态)，表示这个包是一个对设备的数据请求。该设备是由标识符定义的。这个设备的地址就是该域的目标地址
第二个域：6 位	称为控制域。第一位是标识符扩展。第二位总是 1。最后 4 位是数据长度的编码
第三个域：0~64 位	长度依赖于控制域中的数据长度编码
第四个域(如果当前数据域没有数据，则是第三个域)：16 位	是 CRC(循环冗余校验，Cyclic Redundancy Check)字。接收器节点在传送过程中用它来检查错误(如果有的话)
第五个域：2 位	第一位是“ACK 间隙”。在这个间隙中，当接收器检测到接收错误后，发送器发送该位为 1，接收器回发一个 0。发送器在 ACK 间隙中发送 0 后，通常重新发送数据帧。第二位是“ACK 定界符”位。它表示 ACK 域的结束。如果在特定的时间间隙中，发送节点没有收到任何数据帧的响应，就发送该位
第六个域：7 位	帧结束标志，全 0

(5) CAN 总线通常连接到线和节点上的主机之间的 CAN 控制器上。它给出输入，并从主机节点的物理和数据链路层之间获得输出。CAN 控制器具有 BIU(包括缓冲区和驱动器的总线接口单元)、一个协议控制器、状态及控制寄存器、接收缓冲区和信息对象。这些单元通过主机接口电路连接主机节点。

(6) 还有一个称为 CSMA/AMP(具有信息优先级仲裁的载波侦听多路访问)的仲裁方法。节点发现有支配位之后，就停止发送数据，因为支配位标志着另外一个节点正在发送数据。

注意：

CAN 是一种用来链接集中控制网络的串行总线。它主要用于汽车中。它具有多个域，例如总线仲裁位、地址和数据长度的控制位、数据位、CRC 检查位、响应位和结束位。

3.3.3 USB 总线

通用串行总线(USB，Universal Serial Bus)是一种主机系统与几个互连外设之间的总线。它同时提供主机和串行设备之间的高速(最高为 12Mb/s)和低速(最高为 1.5Mb/s)串行传送和接收，串行设备有扫描仪、键盘、打印机、ISDN 等等。有两个标准：USB1.1(一个低速 1.5Mb/s 3m 通道长度以及一个高速 12Mb/s 25m 通道长度)和 USB2.0(高速 480Mb/s 25m 通道长度)。USB 协议具有它的特征，一个设备可以被连接、配置和使用、复位、重新配置和使用、与其他设备共享带宽、断开(同时其他设备处于工作状态)、重新连接。在一个瞬间，主机调度连接的各个设备之间的带宽共享。一个设备可以是总线驱动的，也可以是自驱动的。此外，主机中还有一个用于 USB 端口的电源管理软件。

主机使用 USB 端口驱动软件和主机控制器连接设备或者节点。主机计算机或者系统有一个主机控制器，该控制器连接到一个根集线器上。该集线器又连接到其他的节点或者集线器上。一个树形的拓扑结构构成如下。在第一层，根集线器与集线器和节点相连。第一层的一个集线器连接到第二层的集线器和节点上，依此类推。在最后一层上只有节点。根集线器和同一层上的每一个集线器与下一层形成一个星形的拓扑结构。

USB 总线电路有 4 根线，一条是+5V 线，两条是双绞线，一条是地线。在每一端都有终止阻抗。电磁干扰(EMI)屏蔽电路用于 1.5Mb/s 的设备。

串行信号是不归 0 的(NRZI)，时钟是通过在每一个包之前插入同步编码(SYNC)域而编码的(参见表 3-2)。接收器连续同步它的位恢复时钟。数据传输有四种类型：(a)受控的数据传输。(b)块数据传输。(c)中断驱动的数据传输。(d)准同步传输。

USB 是一种轮询总线。主机控制器有规律地查询设备是否存在，这些设备是由软件调度的。它发送一个令牌包。令牌包括的域有类型、方向、USB 设备地址和设备终点编号。设备通过一个握手包进行握手，标志着发送的成功或者失败。数据包中的 CRC 域允许进行错误检测。

USB 支持三种类型的管道。(a)无 USB 定义协议的“流”。在连接已经建立并且数据流开始发送时使用。(b)用于提供访问的“默认控制”。(c)用于设备控制功能的“信息”。主机使用数据带宽、传输服务类型和缓冲大小来配置各个管道。

注意：

USB 是一个系统互连的串行总线。它将一个设备从网络上连接或者断开。它使用一个根集线器。包含设备的节点可以组织成树形结构。它主要用于将扫描仪之类的 IO 设备连接到计算机系统中。

3.3.4 先进的串行高速总线

嵌入式系统可能会需要连接多个每秒 Gb(G b/s)的收发器(发送和接收)串行接口。示例产品有无线 LAN、Gb 以太网、SONET(OC-48、OC-192、OC-768)。下面是一些新的先进数据协议示例。

(1) 用于 125MHz 性能的 IEEE 802.3-2000(1 Gb/s 带宽 Gb 的以太网 MAC)。

(2) 用于 156.25MHz 双向性能的 IEE P802.3oe 草案 4.1(10Gpbs 以太网 MAC)。

(3) 用于 4 通道、每个通道 3.125G b/s 收发性能的 IEE P802.3oe 草案 4.1(12.5Gb/s 以太网 MAC)。

(4) XAUI(10Gb 连接单元)。

(5) XSBI(10Gigabit 串行总线交换)。

(6) SONET OC-48。

(7) SONET OC-192。

(8) SONET OC-768。

(9) ATM OC-12/46/192。

附录E 的第 E.1 节给出了一个关于串行总线设备标准的表，并总结了串行总线标准的特征。

3.4 多个互连 I/O 设备之间通过 ISA、PCI、PCI-X 和高级总线进行的计算机或者主机系统并行通信

在 PC 或者嵌入式系统中，我们需要一个内部互连总线，连接多个基于 PC 的 I/O 卡、系统和设备。这个总线应该与连接处理器和存储器的系统总线分开。系统总线和互连总线的操作速度必须有所不同。示例设备有显示监视器、打印机、字符设备、网络子系统、视频卡、modem 卡、硬盘控制器、瘦客户端、数字视频采集卡、流显示、10/100 Base T 卡、使用 DEC 21040 PCI 以太网 LAN 控制器的卡。这些设备中的每一个都有特定的功能，都可以包含一个处理器和软件。每一个设备都有特定的存储器地址范围、特定的中断向量(预分配的或者自动配置的)和设备 I/O 端口地址。适当规格和协议的总线将这些设备与主机系统或者计算机进行接口。两个较老的用于主机和设备之间进行通信的总线是 ISA 和 EISA(扩展 ISA)。新的互连总线是 PCI 和 PCI/X(它的一个变种是 cPCI)。下面的小节中将介绍 ISA 和 PCI 总线。

3.4.1 ISA 总线

ISA 总线(IBM 标志体系结构中使用)只连接到一个具有 8086、80186 或者 80286 处理器的卡(或者嵌入式设备)上，在这些卡中考虑了处理器寻址和 IBM PC 体系结构寻址限制和中断相连地址分配等情况。不存在地理寻址。

我们可以对使用 ISA 总线的 IBM PC 对存储器访问的限制进行如下的解释。ISA 总线存储器访问可以在两个范围内：640KB~1MB 和 15MB~16MB。前者还覆盖了视频板和 BIOS 使用的范围(提示：Linux OS 不直接支持通过第二个范围访问设备)。

我们可以对使用这个总线的设备的 I/O 端口地址限制进行如下的解释。8086 到 80286 处理器具有 I/O 映射的 I/O，而不是存储器映射的 I/O。通过提供给 64KB I/O 地址的 I/O 指令，IBM PC 配置忽略了地址线 A_{10}~A_{15}，不对它们进行译码。因此只有 1024 个 I/O 端口地址。对于一个设备，只能使用寻址范围为 000~3FF 的十六进制寻址方式。因此 A_{10}~A_{15} 位是非实质的。下面是工业标准体系结构(ISA)中的地址分配。

(1) DMA 芯片 8237 的地址分配是 000~00F。其他设备的地址如下。

(2) 可编程中断控制器 8255 的地址分配是 020~021。定时器 8253 的地址是 040~043。

(3) 并口可编程并行接口的地址是 060~063。

(4) 地址 080~083、0A0~0AF、0C0~0CF、0E0~0EF 分配给主板上的构件。

(5) 外设的保留地址是 220~24F、278~27F、2F0~2F7、3C0~3CF 和 3E0~3F0。

(6) IBM COM 端口的地址是 2F8~2FF 和 3F8~3FF。

(7) 硬盘和软驱的地址分别是 320~32F 和 3F0~3F7。

(8) 原型卡，例如 ADC 卡，只能使用 300~31F 之间的 32 个地址。

(9) 同步通信的地址是 380~389 和 3A0~3A9。

(10) 同步数据链路控制(SDLC)的地址是 380~38C。

(11) 显示监控器端口地址是 380~38F(单色)和 3D0~3DF(彩色和图形)。

在 IBM PC 80x86 系列中，中断向量是有限的。中断服务函数并不是在软件层上共享的：例如 SWT 中断。最初的 ISA 不允许这种情况发生。

EISA 总线是 ISA 的一个 32 位数据和地址线版本，并且支持 ISA 设备(系统使用这个总线用于 I/O)。EISA 设备驱动程序首先检查主机或者系统中是否有 EISA 总线。它支持中断函数、SCI(串行通信接口，Serial Communication Interface)控制器和以太网设备之间的共享。Unix 和 Linux 支持 EISA 总线驱动的卡和设备。

注意：

ISA 和 EISA 总线与 IBM 体系结构是兼容的。它们用于将 IO 地址下的设备和中心向量连接，称为 IBM PC 体系结构。EISA 是 ISA 的 32 位扩展版本。它还支持软件中断函数和以太网设备。

3.4.2 PCI 和 PCI/X 总线

最近，在计算机系统中使用最多的、与基于 PC 的设备进行接口所用的总线是 PCI。PCI 的吞吐量比 EISA 更高。它几乎是独立于平台的，这一点与 ISA 不同，ISA 依赖于 IBM 平台、IBM PC 中断向量和 I/O 及存储器分配。PCI 的时钟频率与系统时钟最接近。PCI 提供了三种同步并行接口。它的两个版本是 32/33 MHz 和 64/66 MHz。最近出现了一个新的版本，称为 PCI-X 64/100MHz。

最近还出现了 PCI 的两个高速版本，分别是 PCI Super V2.3 264/528Mb/sV 3.3V(在一个 64 位的总线上)、132/264(在一个 32 位的总线上)和用于 800Mb/s 64 位总线 3.3V 的 PCI-X Super V1.01a。

PCI 总线的 32 位数据总线可以扩展为 64 位。并且，其 32 位地址也可以扩展为 64 位。它的协议描述了计算机的不同构件之间的交互作用(规范是 2.1 版本的)。它的同步/异步吞吐量最高可达到 132/528MB/s(33Mx 4/66M x 8 字节/s)。它对 3.3V~5V 的信号进行操作。例如，PCI 卡可以具有一个 16MB 的闪存，以及一个用于 LAN 的路由器网关。

PCI 驱动器可以自动访问硬件，也可以通过程序员分配的地址访问。PCI 用于分配新地址的自动检测接口系统特征对于编码设备驱动程序来说是很重要的。因此 PCI 总线简化了系统外设的增加和删除(连接和断接)。一个生产厂家注册一个全球编号。例如 68HC11 和 80386 是注册的全球号码。PCI 设备中的一个 16 位寄存器标识了它的编号，使得设备自动识别。另外一个 16 位的寄存器标识了一个设备 ID 编号。这两个编号允许设备通过它的主机进行自动识别。每一个设备都可以使用具有 FIFO 缓冲区的 FIFO 控制器，来获得最高的吞吐量。

设备总共可以通过三个识别码说明其地址空间。(i)I/O 端口。(ii)存储单元。(iii)具有 4 字节惟一 ID 的总管 256B 的配置寄存器。每一个 PCI 设备具有 256 字节的地址空间分配，主机可以通过这个地址空间来访问设备。PCI 总线独有的特征是它的地址空间配置。中断是由惟一分配的中断类型(一个编号)来处理的(关于 IBM PC 中类型的意义，请参见 4.5.3 节)。第 60 号配置

寄存器保存了一个字节，用于定义这个惟一编号的中断类型。图 3-7 给出了一个 PCI 设备中的 64 字节标准配置寄存器。下面是图中使用的缩写。

VID：厂商 ID。DID：设备 ID。RID：修订 ID。CR：常用寄存器。CC：系列编码。SR：状态寄存器。CL：高速缓存线。LT：等待时间定时器。BIST：Base Input Tick。HT：头类型。BA：基地址。CBCISB：卡的基 CIS 指针。SS：子系统。ExpROM：扩充 ROM。MIN_GNT：最少保证时间。MAX_GNT：最长保证时间。

VID、DID、RID、CR、CC、SR 和 HT 是必须配置的，其余是可选的。

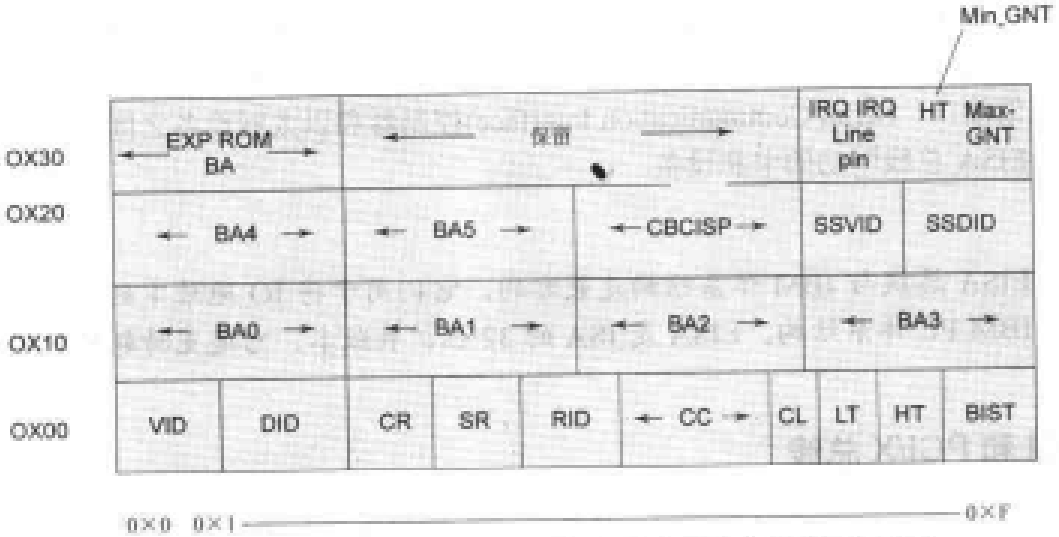


图 3-7 一个 PCI 设备中的 64 字节标志独立于设备的配置寄存器

一个 PCI 控制器在同一时间只能访问一个设备。因此，主机中的所有设备可以共享 I/O 端口地址和存储单元，但是不能共享配置寄存器。这就意味着，一个设备不能修改其他配置寄存器，但是能够访问其他设备资源、共享其他设备的工作或者协助其他设备。如果有必要，PCI 驱动程序可以改变默认的开机配置。

设备可以在启动时进行初始化。这样能够帮助避免发生地址冲突。启动时，一个 PCI 设备禁止中断，并关闭除了配置寄存器空间之外的地址空间。具有设备的 PCI BIOS 完成配置，然后存储器和地址空间自动映射到主机中的地址空间。

注意：

PCI 和 PCI/X 总线的使用是独立于 IBM 体系结构的。PCI/X 是 PCI 的一个扩展，并支持 64/100MHz 的转换。最近又出现了 PCI 总线体系结构的新版本。

3.4.3 高级并行高速总线

许多无线电通信、计算机和具有嵌入式处理器的产品都需要并行总线，用于系统 I/O。PCI 并行同步/异步总线的三个版本提供了系统同步并行接口。这三个版本可能不具有需要系统 I/O、路由器、交换机和网关、SAN(存储区域网)、WAN(广域网)和其他设备的足够高速和超高速以及大带宽。它们不能满足源同步接口需要。在下述情况中，带宽是以指数增长的：音频、图形、视频、交互式视频和宽带 IPv6 Internet。嵌入式系统可能会需要使用 Gb 并行同步接口连接 I/O。下面是一些先进的总线标准协议和最近开发的几个协议。

(1) GMII(Gigabit 以太网 MAC 交换接口)。

(2) XGMI(10G bit 以太网 MAC 交换接口)。

(3) CSIX-1 6.6 G b/s 32 位 HSTL，性能为 200MHz。

(4) RapidIO™ 互连规范 v1.1、8G b/s，性能为 500Mb/s 或者使用 8 位 LCDS(低压数据总线)的 250MHz 双向寄存器性能。

附录 E 中的第 E.2 节给出了并行总线设备标准的表，并总结了现有的并行总线标准。

注意：

PCI 并行总线对于分布式设备来说是很重要的。最近的高速复杂系统都使用新的复杂总线。

本章小结

I/O 设备和定时设备在任何一个系统中都是最基本的。

- 嵌入式系统通过端口与外部设备，例如小键盘、多线显示单元、打印机或者调制解调器连接。设备通过并行或者串行 I/O 端口连接和访问系统处理器。设备端口可以是半双工或者全双工的。每一个端口都有一个分配的端口地址。处理器访问存储器映射 I/O 中的地址，就像访问一个存储器地址一样。译码器以地址总线信号作为输入，并产生片选信号 CS，用于端口地址选择。当处理器初始化一个端口的读或者写周期时，这个信号被激活。在将数据位保存到端口缓冲区或者接收来自于端口缓冲区的数据位之前，这个设备可以使用握手信号。
- 接收器根据同步串行输入和输出的发射器时钟相位接收数据位。
- 数据位在接收器的接收独立于 UART(异步串行输入输出端口)的时钟相位。UART(在微控制器中)通常以 10 位或者 11 位的格式发送字节。HDLC 是一个用于设备之间的同步通信数据连接网络的标准协议。嵌入式网络设备使用一个给定应用推荐的网络协议。
- 微控制器中还有专用端口。这些端口在特定的情况下提供特定的功能。微控制器中还存在着具有处理器的片上设备。
- 定时器设备在系统中具有广泛的应用。系统中至少要有一个硬件定时器。软件定时器是一个虚拟定时设备。系统中可以有多个软件定时器。定时器本质上是一个在规则时间间隔内获得计数输入(嘀哒)的计数器。具有处理器(微控制器)单元的内部可编程定时设备在许多应用中都可以使用，可以产生软件定时器使用的嘀哒中断。
- I^2C 总线用于内部集成电路通信的多个 IC 之间。
- 在汽车电子的集中控制网络中使用 CAN 总线。
- USB(通用串行总线)是一个系统和设备——例如扫描仪、键盘、打印机和鼠标——之间的串行总线通信的标准。有一个根集线器。所有的节点都具有树形结构。
- 如果有一个 I/O 卡连接到 PC 或者嵌入式系统上，则需要使用 ISA 或者 PCI 总线。这些总线使得主机能够与其他设备进行通信，例如网络接口卡(NIC)。最近许多系统都使用 PCI 或者 PCI/X 总线。
- 复杂嵌入式系统的设备和总线具有很高的复杂度。

关键词及其定义

- 并口(parallel port)：同时能够进行多个位的读和写操作的端口。
- 串口(serial port)：同一时间只能进行一位的读和写操作的端口，传送信息的每一位都由固定的时间间隔开来。
- 输入缓冲区(input buffer)：输入设备向缓冲区中写入数据，处理器随后读取这个数据。
- 输出缓冲区(output buffer)：在处理器的写操作之后，输出设备重缓冲区中接收数据。
- 握手信号(handshaking signal)：在将数据保存到端口缓冲区或者从端口缓冲区中接收数据之前所出现的信号。
- 设备(device)：通过端口连接到处理单元的单元。根据其接口电路，它具有固定的预分配端口地址(设备地址)。
- 控制寄存器(control register)：保存控制设备行为的位的寄存器。只能对其进行写操作。
- 状态寄存器(status register)：保存反映端口缓冲区当前状态的位的寄存器。只能对其进行读操作。设备服务时，这些位可以或者不可以自动复位。
- I/O 端口(I/O Port)：在一个瞬间进行输入或者输出操作的端口。握手输入和握手输出端口也称为 I/O 端口。例如，小键盘连接到一个 I/O 端口上。
- 半双工(half duplex)：一个具有一条公用 I/O 线的串口。例如电话线。在一个瞬间，信息只向一个方向传送。
- 全双工(full duplex)：一个具有两条 I/O 线的串口。例如，调制解调器连接到计算机的 COM 端口。在 9 针或者 25 针的连接器上有两条线：TxD 和 RxD。在一个瞬间，信息可以向两个方向传送。
- 设备译码器(device decoder)：将数据总线信号作为输入，产生片选信号 CS，用于端口地址选择。
- UART：串行数据的标准异步串行输入输出端口。UART(微控制器中的)通常以 10 位或者 11 位的格式发送数据。10 位格式是开始有一个起始位，后面跟随 8 位信息(字符)，最后是一个结束位。11 位格式是在结束位之前还有一个特殊位。
- TxD：用于 UART 串行数据传送的线。当使用 RS232C COM 端口时，0 和 1 信号的电压水平遵循 RS232C，在微控制器中遵循 TTL 电压水平。
- RxD：用于 UART 串行数据接收的线(0 和 1 信号的电压值与 TxD 相同)。
- COM 端口(COM Port)：是计算机上的一个端口，可以连接鼠标、调制解调器或者串行打印机。
- RS232C 端口(RS232CPort)：是一个 UART 发送和接收数据标准，TxD 和 RxD 的电压水平是相反的(0 是+12V，1 是−12V)，握手信号 CTS、RTS、DTR、DCD 和 RI 遵循 TTL 的电压水平(RS232C 标准用于 COM 端口)。
- 协议(protocol)：一种在网络上传送信息的方式，通过使用软件，增加附加的位，例如起始位、头、源和目的地址、错误控制位和结束位。在通过网络传送信息之前，每一个层或者子层都要使用各自的协议。
- HDLC(High Level Data Link)：高级数据链路控制协议，用于主设备和从设备按照标准的定义进行同步通信。它是一个面向位的协议。
- 实时(real time)：一直增加，没有停止或者复位的时间。

- 实时时钟(Real Time Clock，RTC)：一个永不停止地在固定时间间隔内产生中断的时钟。RTC 中断驱动系统中的其他定时器，例如 SWT。
- 系统时钟(system clock)：与处理器时钟成比例的时钟，总是增加，不能停止或者复位，在预先设定的时间间隔内产生中断。
- 硬件定时器(hardware timer)：在系统中以硬件形式存在的定时器，从内部处理器或者系统时钟获得输入。设备驱动程序对它进行编程，就像对其他物理设备编程一样。
- 定时器溢出(timer overflow)或者超时(Time-Out)：计数输入的数量超过了设定数值的状态，当到达该状态后，产生一个中断。
- 定时器结束(timer finish)：定时器到达预先设定的计数数值并停止之后的状态。此时会产生一个中断。
- 定时器复位(timer reset)：定时器的所有位为 0 或者 1 的状态。在定时器被编程为连续运行的情况下，溢出后都会产生一个复位。
- 定时器重装(timer reload)：定时器的所有位为 0 或者 1 的状态。在定时器被编程为自动重装的情况下，结束后都会产生一次重装，并重新开始运行。
- 软件定时器(software timer)：根据定时器输出或者实时时钟中断运行，增加或者减小计数数值的软件。软件定时器也可以在计数数值溢出或者计数变量到达结束数值(到达预先定义的数值)后产生中断。
- 计数器(counter)：从事件的发生(可能是在不规律的时间间隔内)获得计数输入的单元。
- 自激计数器(free running counter)：加电时启动、由内部时钟(系统时钟)驱动、不能被停止和复位的计数器。
- 事件(event)：当前条件的改变。
- 事件标志(event Flag)：一个布尔变量，当为真时表示事件的发生。
- 延迟(delay)：在某个预先定义的周期内，一个行为被阻塞。
- Watchdog 定时器(watchdog timer)：设备中的一个重要定时设备，在预先定义的定时事件到后将系统复位。在复位后的最初几个时钟周期内，这个时间可以预先定义。
- 漏极开路输出(open drain output)：源和漏之间内部断开的门。优点是上拉了电路电压和电流，在与它进行接口的时候需要这种状态。当使用输出的时候需要一个外部上拉电路。
- 准双向端口(quasi bi-directional port)：具有双重优势，当与端口进行接口时使用上拉电路满足电压和电流的需要，在短时间内不使用上拉电路就足以驱动 LSTTL 电路。
- 分时复用技术(time division multiplexing)：是一种选通方式，在不同的时间端内，可以选择不同通道的信息。
- 信号分离技术(demultiplexing)：一种分离多路选择输入并将信息传递给多个通道的方式。
- 片上端口和设备(on-chip port and device)：与处理单元共同存在的端口和设备，例如在微控制器中。
- 同步通信(synchronous communication)：在驱动发送设备和接收设备的时钟之间存在固定的相位差的一种通信方式。在字节帧发送之间的最大时间间隔是固定的。
- 准同步通信(isosynchronous communication)：帧之间的相位差不固定，而每个帧中的相位差固定的一种通信方式。驱动发送设备和接收设备的时钟没有分离。只有帧字节传

送之间的最大时间间隔没有预先定义，也就是说是可变的。用于在 LAN 或者两个处理器之间进行通信。

- 异步通信(asynchronous communication)：相位差不固定、驱动发送设备和接收设备的时钟没有分离的一种通信方式。帧字节传送之间的最大时间间隔没有预先定义，并且是无限制的。
- PISO(Parallel Input and Serial Output，并行输入和串行输出)：用于并行输入和串行输出的移位寄存器。它用于以同步方式接收串行数据位。
- SIPO(Serial Input and Parallel Output，串行输入和并行输出)：用于串行并行输入和并行输出的移位寄存器。它用于以同步方式发送串行数据位。
- FSK(Frequency Shifted Keying，频移键控)调制：0 和 1 的逻辑状态的频率是不同的。例如，电话线的 0 频率是 1050Hz，1 频率是 1250Hz。它允许将电话线用于串行位发送和接收。
- PSK(Phase Shifted Keying，相移键控调制)调制：在高频信号中，0 和 1 的逻辑状态的频率是不同的。PSK 调制允许将电话线用于串行位发送和接收。
- QPSK(Quadrature Phase Shifted Keying，4 相相移键控)：它允许将电话线用于串行位的双频发送和接收。它允许 56 Kb/s 的调制解调器表现出等价于 112 Kb/s 的性能。
- 主从式通信(Master Slave Communication)：是两个处理器之间的一种通信方式，一个处理器在接收到来自于寻址的从处理器的应答信号后，驱动数据的发送过程。
- I^2C 总线：遵循一种通信协议的标准总线，在多个 IC 之间使用。它允许一个系统和连接到该总线上的多个兼容 IC 之间进行数据交换。
- PCI 总线：一种用作“外设组件互连”的总线。
- CAN 总线：在汽车电子的控制域网络中使用的一种标准总线。
- ISA 总线：基于“IBM 标准体系结构”总线的标准总线。
- USB 总线：用于快速串行发送和接收的标准总线。
- PCI/X 总线：用作“PCI 扩展”总线的标准总线。

问题回顾

(1) 将 I/O 端口和设备与存储器设备一样进行地址映射的处理器的优点是什么？

(2) 比较使用串行和并口/设备进行数据传输的优点和缺点。

(3) 举例说明串行设备进行串行通信的三种模式：“异步”、“准同步”和“同步”。

(4) 下列通信方式(a)UART；(b)HDLC；(c)CAN 如何表示数据帧的开始和结束？

(5) (a)8051 和(b)68HC11 中的内部串行通信设备分别是什么？比较每一个设备的工作模式(参见 3.1.3 节和 G.5 节)。

(6) 一个端口设备可以具有多字节数据输入缓冲区和数据输出缓冲区。它们的优点有哪些？

(7) 参见 D.4 节中的表。DSP 中的 DAA 和 McBSP 的优点有哪些？

(8) 在嵌入式实时控制器中，68HC16 和 683xx 微控制器中的 TPU(定时器处理单元)起到什么作用(参见附录 C 中的 C.2 节)？

(9) 串行设备互连总线的意义是什么？并行设备互连总线的意义是什么？

(10) 解释 I^2C 总线的每一个控制位的作用。

(11) 即插即用设备的意义是什么？下面给出的练习中列出的支持即插即用设备的总线协议是什么？

(12) 热插拔的意义是什么？下面给出的练习中列出的支持热插拔的总线协议是什么？

(13) 什么是定时器？计数器如何实现(a)定时功能；(b)预先初始化的事件产生；(c)时间捕获功能？

(14) 在一个嵌入式系统中，为什么必须至少有一个定时设备？

实践练习

(15) 在嵌入式系统中，下列设备特征有什么作用？(a)施密特触发器输入；(b)低电压 3.3V IO；(c)动态控制的阻抗匹配；(d)PCS 子单元；(e)PMA 子单元；(f)SerDes。对每一个设备给出一个应用实例。

(16) 用于点对点互连的 PPP 协议具有 8 个起始标志位、8 个地址位、8 个协议说明位、可变数量的数据位、16 位 CRC 和 8 个结束标志位。一个 PPP 帧的最大数据量可以是 10264 位。在一个 PPP 帧中，最多可以传送多少个字节？载荷(帧)中的过量的最小百分比是什么？

(17) 参见 D.6 节。列出自激定时器、有规律中断的定时器和脉冲累加计数器(PACT)的应用。从 PACT 中如何获得 PWM 输出？从 PWM 设备中如何获得 DAC 输出？

(18) 一个 16 位的计数器从 12MHz 的内部时钟获得输入。有一个预引比例电路，预引比例因子为 16。该定时器产生的溢出中断的时间间隔是多少？必须为该中断服务的时间间隔是多少？

(19) 软件定时器(SWT)的意义是什么？在实时时钟下调度多任务时，SWT 所起的作用是什么？假设有三个 SWT，分别编程为根据练习 18 中的定时器的溢出中断 1024、2048 和 4096 次中断溢出后，超时。每一个 SWT 的超时中断频率为多少？

(20) 拒绝响应的优点和缺点各是什么？

(21) 一种新一代的汽车具有大约 100 个嵌入式系统。CAN 总线中的总线仲裁位、地址和数据长度控制位、数据位、CRC 检查位、响应位和结束位如何帮助汽车嵌入式系统的互连设备进行通信？

(22) USB 协议如何提供给设备连接、配置、复位、重新配置、与其他设备的带宽共享和设备断开(而其他设备处于操作状态)和重新连接服务？

(23) 参见 3.3 节和 F.1 节。设计一个表格，比较最大操作速度和总线长度，给出使用下面每一个串行设备的两个示例：(a)UART；(b)单线，CAN；(c)工业 I^2C；(d)SM I^2C 总线；(e)68 系列 Motorola 微控制器的 SPI；(f)容错 CAN；(g)标准串口；(h)MicroWire；(i)I^2C；(j)高速 CAN；(k)IEEE 1284；(l)高速 I^2C；(m)USB 1.1 低速通道和高速通道；(n)SCSI 并行；(o)高速 SCSI；(p)Ultra SCSI-3；(q)FireWire/IEEE 1394；(r)高速 USB2.0。

(24) 参见 3.4 节和 F.2 节，并通过互联网搜索相关资料。设计一个表格，比较最大操作速度和总线长度，并给出使用下列并行设备的两个实例：(a)ISA；(b)EISA；(c)PCI；(d)PCI-X；(e)COMPACT PCI；(f)GMII(Gb 以太网 MAC 交换接口)；(g)XGMI(10 Gb 以太网 MAC 交换接口)；(h)CSIX-1，6.6G b/s 32 位 HSTL，性能为 100MHz；(i)RapidIO™ 互连规范 v1.1，8G b/s，性能为 500Mb/s，或者使用 8 位 LVDS(低压数据总线)的 250MHz 双向寄存性能。

(25) 通过 Internet 搜索相关资料，并设计一个表格给出下列最近出现的串行总线的特征。

(a)IEEE 802.3-2000(1 G b/s 带宽 Gigabit 以太网 MAC)，性能为 125MHz；(b)IEEE P802.3oe 草案 4.1(10 Gb 以太网 MAC)用于 156.25MHz 双向性能；(c)IEEE P802.3oe 草案 4.1(12.5 Gb 以太网 MAC)用于四个通道，每个通道具有 3.125Gb 的收发性能；(d)XAUI(10 Gb 连接单元)；(e)XSBI(10 Gigabit 串行总线交换)；(f)SONET OC-48、OC-192 和 OC-768；(g)ATM OC-12/46/192。

第 4 章　设备驱动程序和中断服务机制

本章前所学内容

(1) 嵌入式系统硬件包括处理器、存储器设备(ROM 和 RAM)内部时钟和 I/O 端口设备以及基本硬件单元：电源、时钟和复位电路。

(2) 系统中的处理器、存储器和设备的组织结构。

(3) 存储器映射的 I/O 处理器访问内部设备、I/O 端口上的设备、外设以及使用类似于存储器地址的端口地址访问其他片外设备。

(4) 每一个设备都具有三组寄存器的地址：数据寄存器(或者缓冲区)、控制器寄存器和状态寄存器。处理器用这些地址访问每一个寄存器。我们还学习了关于下列重要设备和分布式设备互连总线的知识。

(5) 同步和异步串行设备：例如，HDLC 和 UART 设备(串行线设备)。微控制器中的内部串行通信设备。具有复杂接口特征、用于高速 I/O 和高速收发功能的设备。用于实时声音和视频 I/O 的设备。定时和计数设备。作为虚拟定时设备的软件定时器。其他设备，例如小键盘、键盘、LED 显示单元、LCD 显示单元、DAC 和 ADC 和脉冲拨号电路。

(6) 用于分布式设备互连的串行总线：用于内部 IC 通信的多个分布式 IC 之间的 I^2C 总线、汽车中的分布式设备控制网络中的 CAN 总线(Control Area Network bus，控制域网络总线)、用于嵌入式系统和分布式设备——例如键盘、打印机、扫描仪和 ISDN 系统——之间的高速串行发送和接收的 USB(通用串行总线)。

(7) 主机(或者系统)和基于 PC 的设备(或者系统、卡)——例如 NIC(网络接口卡)——之间的并行总线—ISA(工业标准体系结构)和 PCI(外设组件互连)/PCI-X(PCI 扩展)接口总线。

考虑巧克力自动售货机中的一个设备。设备连接在一个输入端口上，用于收集投入的硬币。系统如何唤醒、激活和收集硬币？系统通过一个硬件或者软件信号中断唤醒和激活。中断后系统运行一个服务例程收集硬币。这个例程就是端口的设备驱动例程。端口设备如何唤醒？一种方式是通过“编程的 I/O 转换”，也称为“等待传送”或者“忙等待传送”，设备持续等待硬币投入，当发现有硬币投入后被激活并运行服务例程。在标准的方法中，设备应该在发现硬币投入事件发生后，由一个中断激活。

考虑一个数码相机系统。它具有一个图像输入设备。(否则就只能是一个玩具！)系统如何唤醒和激活，设备如何获取图像？同样，系统根据来自设备的一个硬件或者软件信号中断唤醒和激活。中断后，系统运行一个服务例程。这个例程读取设备帧缓冲区(称为帧抓取)，处理并压缩，然后将图像数据保存到一个闪存中。服务例程是用于图像输入设备的设备驱动程序。输入帧抓取设备如何唤醒？也是在发现有图像后，由一个中断唤醒。

上面给出的两个示例明显地说明，中断和中断服务设备驱动程序(ISR)在嵌入式系统中起着非常重要的作用。任何系统都会有设备，因此也就需要设备驱动程序。嵌入式软件设计者必须

使用用于设备(i)配置(初始化)，(ii)激活(也称为打开或者连接)，(iii)用 ISR 驱动，以及(iv)复位(也称为去活、关闭或者断开)的代码。需要设计系统中每一个设备功能以及设备的每一个中断的代码(提示：(i)当 ISR 为设备中断服务时通常称为设备驱动程序。(ii)设备驱动程序是存储器和处理器敏感的程序)。

下面来看一个视频系统的示例。当系统运行时，两个设备驱动程序 ISR 也运行。一个驱动程序用于声音设备，另外一个用于图像设备。假设 ISR 和其他系统软件是这样设计的：两个设备驱动程序不保持同步。因此设备不能满足每一个系统设备服务的最终期限设置。就像一个人总是能够到达车站，但是却赶不上车！下一组图像和声音信号将会丢失。因此，系统软件设计者的一个关键问题是如何为多个设备中断设计适当的 ISR，使得所有的设备中断调用都能够在每一个中断的最终期限之内得到响应。设计应该获得最优的延迟，并为每一个任务设置适当的最终期限。

每一个 ISR 服务的延迟是什么？什么是最终期限？当需要切换到另外一个 ISR 上下文时，先前运行的例程上下文如何以及在多少时间内保存？当返回到先前的 ISR 时，保存的上下文如何恢复？设计者必须清楚地理解 I/O 设备和定时设备、控制和状态位、中断发生后的动作次序和速度、延迟和最终期限的问题。

本章将学内容

在本章中我们将学习下面的内容：

(1) 使用任何系统中的中断服务例程的基本知识，以及通过简单的示例学习系统中断服务机制的工作情况。

(2) 为设备中断服务的设备驱动程序，以及外设单元(设备)的驱动。

(3) 设备初始化和端口访问。

(4) Linux 内幕和设备驱动程序的使用。

(5) 并口和串行 UART 的设备初始化和设备驱动程序编写的示例。

(6) 设备中断和各种可能的软件和硬件中断源列表。

(7) 系统中的软件指令相关和运行时间相关中断。

(8) 中断系统和单个设备中断使能和禁止、中断向量、中断等待寄存器和状态寄存器、不可屏蔽、可屏蔽、只有在复位后几个时钟周期内设置为不可屏蔽时才是不可屏蔽的。

(9) 中断发生后的上下文和上下文切换。

(10) 中断延迟和中断服务最终期限。

(11) 80197、80960 和 ARM7 中采用的快速上下文切换方法。

(12) 根据中断服务“保存”或者“不保存”除了程序计数器以外的上下文而进行的处理器分类。

(13) 使用 DMA 通道帮助中断源缩短中断延迟周期。

(14) 多中断源之间的软件和硬件优先级的设置。

(15) 同时有多个中断源请求中断服务情况下的响应方法。

4.1　设备驱动程序

4.1.1　不使用 ISR 的设备服务

示例 4-1 给出了不使用 ISR 的设备服务。该示例说明了使用中断服务机制的优点，并解释了设备服务机制的各个要点。

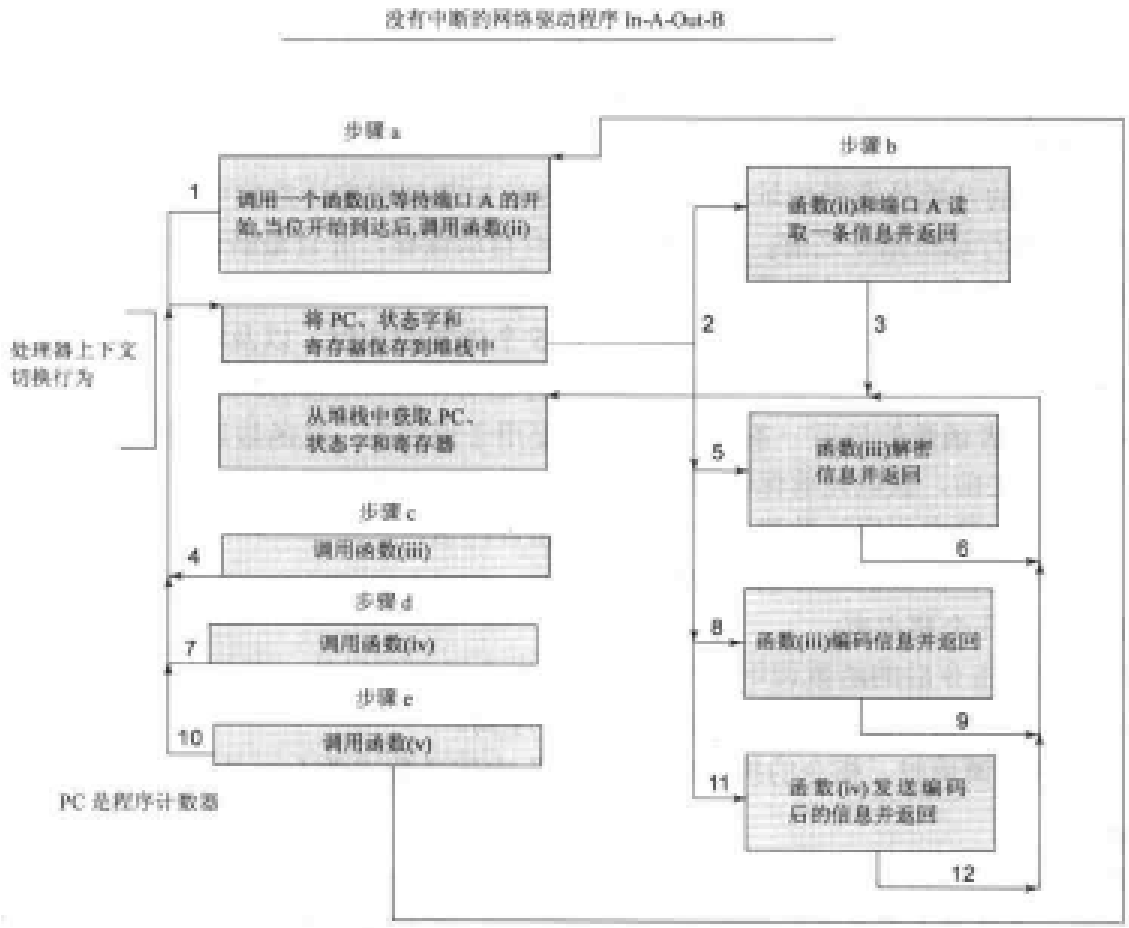


图 4-1　示例网络驱动程序 In_A_Out_B 中步骤 a-e 对 5 个函数的调用，图中还给出了每一个被调用函数的调用和返回过程。箭头上的编号表示程序的运行顺序

示例 4-1

考虑一个 64 kpbs 的网络。使用以 11 位格式传送数据的 UART，网络每秒钟最多传送 5818 个字符。在 172μs 之前，接收器会接收到另外一个字符，假设所有接收的字符是连续的，中间没有时间间隔。设端口 A 连接到 PC 中的一个以太网口，端口 B 是 PC 的调制解调器输入，将字符发送到电话线上。设 In_A_Out_B 是从端口 A 接收字符，并向端口 B 发送输出字符的例程。在不使用中断服务机制的情况下，In_A_Out_B 调度下列从 a~e 的步骤，执行函数(i)~(v)，因此确保了调制解调器从卡上连续获得字符。

步骤 a：函数(i)：检查端口 A 是否有字符。如果没有，则等待。

步骤 b：函数(ii)：读端口 A。

步骤 c: 函数(iii): 解密信息。

步骤 d: 函数(iv): 对信息进行编码。

步骤 e: 函数(v): 将编码后的字符传送给端口 B。

假设将必须切换到 In_A_Out_B 周期开始之前的最小时间延迟定义为 150μs。这样就不需要连续执行函数(i)来检查端口 A 是否有字符，可以先调用一个延迟函数执行 150μs，然后执行函数(i)。在这 150μs 之间可以执行其他的代码。如果字符不是连续接收的，步骤 a 中的等待周期将很长。当前方法最大的缺点是等待周期的处理器时间浪费。

在从 a~e 的每一个步骤中，当有一个对函数(方法)的调用时，函数的执行主要有三个步骤：

(1) 将当前的程序计数器地址保存到堆栈中，当不调用函数时，处理器将从该地址开始执行。

(2) 调整堆栈指针的值。

(3) 从函数的开始处获得地址(指针)，将该地址加载到程序计数器中，然后执行被调用的函数指令。在执行这些指令之前，如果处理器没有自动保存，并且最新调用的函数需要，还要保存当前程序的状态字、寄存器和其他程序上下文。

图 4-1 给出了 In_A_Out_B 在步骤 a~e 中对 5 个函数的调用，以及每一个被调用的函数在调用和返回时的执行过程。箭头上的编号表示程序运行(流程)过程的顺序。

任何例程或者函数的最后一条指令(动作)总是用于返回的。从函数返回的步骤如下所述。

(1) 在返回之前，获取先前保存的状态字、寄存器和其他上下文。

(2) 从堆栈中获取程序计数器，并调整堆栈指针的值。

(3) 执行调用该函数的程序剩下的部分。

需要记住的几个要点有：

i. 执行某条指令后的函数调用是一个从当前指令序列到另外一个指令序列的有计划的(用户编程)转移；直到返回之前，都应该执行该指令序列。

ii. 发生函数调用时，指令的执行类似于“C”中的函数或者 Java 中的方法。

注意：

一种服务(输入、输出或者任何其他动作)方法是“编程的 IO 转移”，也称为“忙等待转移”。当等待周期占用了整个程序执行周期的很大一部分时，等待过程中浪费处理器时间是这种方法的最大缺点。

4.1.2 设备驱动程序 ISR

设备驱动程序的一个示例是处理端口输入的驱动程序。其工作情况如下所述。

(i) 将设备缓冲区满标志(在状态寄存器中)复位，准备进行下一次读操作(记住：标志是一个布尔变量寄存器位，当置位时表示一种需求。这种需求就是执行中断服务例程(Interrupt Service Routine，ISR)。当相应的 ISR 开始执行后，标志必须被复位。如果一个中断源的标志用来表示中断的发生，则每一个中断源的标志必须是不同的)。

(ii) 通过将缓冲区腾空并将字节保存到存储器中，或者根据系统需要使用接收到的字节，读取输入缓冲区。

记住，在驱动之前，还必须对设备进行配置。在这一点上，设备驱动例程与其他的 ISR 不同。

示例 4-2 给出了使用 ISR 的设备服务。它解释了使用系统中断机制的优点，还解释了在所

有设备服务机制中都包含的各个要点。

示例 4-2

考虑如何使用 In_A_Out_B 从(i)~(v)周期的中断机制。使用中断服务机制能够节约函数(i)中的处理器等待时间。当字符并不是连续到达端口 A 时，也能够节约时间。在节约的这段时间中，系统的处理器空闲，可以执行其他的例程或者代码。

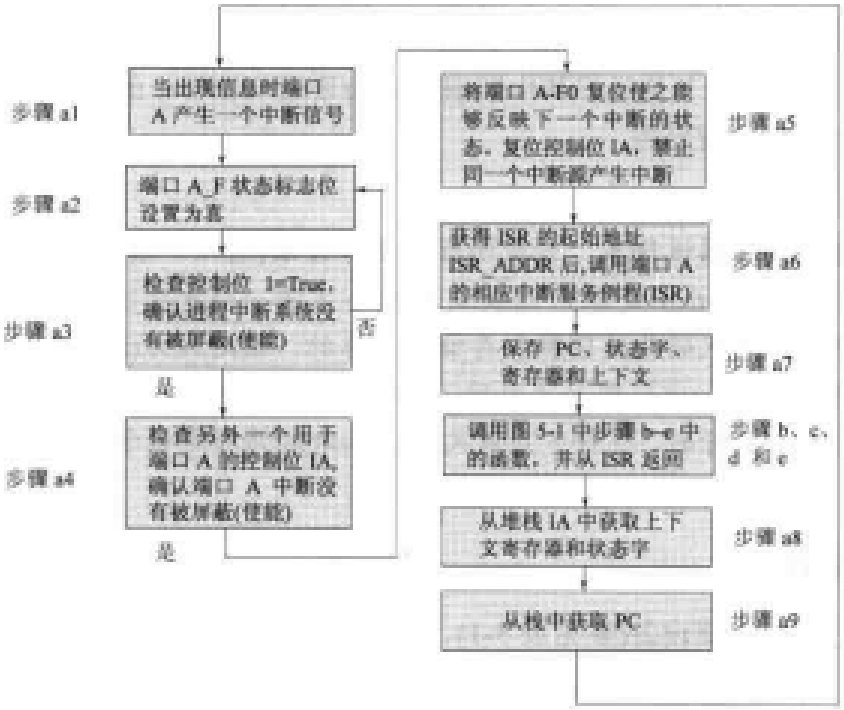


图 4-2　示例网络 In_A_Out_B 的中断调用步骤

图 4-2 给出了上一个示例中网络中断服务调用步骤 a1~a6，并在步骤 a7~a9 中展示了前一个示例中步骤 b~e 的驱动 ISR。

当使用中断服务机制时，需要执行下列步骤。

(i) 当信息到达时，端口 A 产生一个中断信号(步骤 a1)。

(ii) 当一个字符到达端口 A 时，状态表示标志 PORT_A_F 被置位(真)，表示需要激活这个中断源(一个内部或者外部硬件信号)(步骤 a2)。

(iii) 当在步骤 a3 和 a4 将 PORT_A_F 置位，任何其他正在进行的函数或者程序就会发生中断，相应的服务例程执行函数(ii)~(v)。当一个 ISR 调用初始化后，步骤 a5~a9 运行。如果步骤 a3 发现控制位 I 已经屏蔽(禁止)了各种设备中断的中断服务机制，则中断服务被拒绝(这个控制位也称为一级屏蔽位)。如果步骤 a4 发现一个控制位 IA 已经屏蔽了端口 A 的这个中断服务，则端口 A 的中断服务被拒绝(这个控制位也称为二级屏蔽位)。

(iv) ISR 的最后一条指令也类似于一个函数。它的作用是返回。从例程返回的步骤如下所述。

(v) 从例程返回之前，要恢复状态字、寄存器和其他上下文。

(vi) 如果中断被源硬件设备中断自动禁止或者在步骤 a5 中由用户指令禁止设备产生进一步的中断，则应该分别自动地或者由用户指令激活进一步的中断。将控制位 IA 置位，准备下一次的端口 A 读操作(步骤 a8)。

(vii) 获取保存的程序计数器，并调整堆栈指针(步骤 a9)。

(viii) 继续执行先前被中断程序的剩余部分。

以下几点需要记住。

(i) 执行某条指令后的中断调用不是有计划的(源激活的)从当前指令序列到另外一个指令序列的转移，并在返回之前一直执行转移到的指令序列。源激活示例(事件)并不是预知的。中断源可以是任何设备中断、软件异常或者错误。

(ii) 在中断调用时，指令并不是像 C 函数或者 Java 方法一样完全连续执行。它们的执行就像系统的中断机制一样。例如，从 ISR“返回”在某些重要的方面是不同的。中断机制可能是 ISR 开始执行后会自动禁止其他的设备中断服务。如果这些被禁止的设备中断服务是服务调用之前启用的，则会自动重新启用。

注意：

实际的设备要比处理器慢得多。因此实际的服务方法(输入、输出或者任何其他动作)都是中断驱动的 IO 传送。指令并不是像 C 函数或者 Java 方法一样完全连续执行。它们按照系统的中断机制执行。驱动程序代码是处理器和存储器敏感的。这是因为(i)当设备地址改变后，也应该对程序进行修改，并且(ii)当处理器改变后，中断服务机制也改变了。当等待周期占用了程序整个执行周期的很大一部分时，这种方式最大的优点是能够避免由于等待而导致的处理器时间浪费。

4.1.3 设备驱动程序

一个系统具有多个物理设备(参见表 3-1)。一个设备可以有多个功能。每一个设备功能都需要一个驱动程序。定时器设备执行定时功能，还执行计数功能。它还执行延迟功能和周期性的系统调用。(参见 3.2 节)收发设备既具有接收功能，又具有发送功能。它不应该仅仅是一个转发器。它还有可能进行超长控制和冲突控制。(超长控制意味着当发生系统失效时，禁止不必要的连续字节流。冲突控制意味着在发送之前必须检查网络总线是否可用。)声音数据传真调制解调器设备具有对声音、传真和数据的接收和发送功能。需要一个公用驱动程序或者针对每一个设备功能的单独驱动程序。为什么设备驱动程序是系统中的重要例程？下述几点针对这个问题进行了解释。

(1) 驱动程序提供了应用程序和实际设备之间的一个软件层(接口)：当运行一个应用程序时，需要使用系统的设备。例如，一个用于邮件发送的应用程序产生了一个字节流。当按照各个层的协议——例如 TCP/IP——对信息流进行压缩后，需要将它们通过网络驱动卡发送出去。为了使用网络接口卡(设备)，网络驱动例程将提供应用程序和网络之间的软件层。

(2) 驱动程序帮助更加方便地使用设备：驱动例程的编写方式通常是使其能够被应用程序开发人员作为一个黑盒使用。一个任务或者函数的简单命令就可以驱动设备。一旦有了驱动程序后，应用程序开发人员不需要知道关于设备使用的机制、地址、寄存器、位和标志等情况。例如，考虑系统时钟被设置为每 100is 嘀哒一次(每秒 10000 次)的情况。用户程序只需要调用函数 OS_Ticks(10000)即可。这个函数的用户没有必要知道该函数将执行哪一个定时器设备。地址是什么，驱动程序将使用哪一个？哪一个设备寄存器将存储数值 10000？哪些控制位将被置位或者复位？当 OS_Ticks(10000)运行时，只需要中断设备。执行的驱动程序以 10000 作为输入，对实时时钟进行配置，使得系统时钟每 100is 嘀哒一次，每 100μs 时连续产生系统时钟中断，使得每一秒产生 10000 次嘀哒。

(3) 通常，有一个公用的方法来使用设备驱动程序：所有对设备的访问都不是直接进行的，而是首先将设备中断，然后执行需要的驱动例程。

(4) 通常，在使用驱动程序之前，需要打开(注册或者连接)设备(或者设备函数模块)：使用用户函数或者 OS 函数，首先初始化设备，并通过置位和复位设备控制寄存器中的控制位对设备进行配置，并启用中断服务机制。这个过程也称为打开、注册或者连接设备(或者设备函数模块)。使用用户函数或者 OS 函数，可以在另一个过程中将设备(或者设备函数模块)关闭、撤销注册或者断开。执行这个过程之后，在设备重新打开(重新注册或者重新连接)之前，设备驱动程序是不可访问的。

端口、小键盘、显示器、定时器和网络设备的驱动程序(参见 3.1 节~3.4 节)在系统中是应用最普遍的，多数用于声音和视频系统的媒体设备中都需要 PCS 和 PMA 驱动程序。由软件程序员编写设备的每一个功能的代码变得不太现实。对于普遍使用的设备，设计者通常依赖于经过彻底测试和调试的操作系统中所提供的驱动程序。(关于μCOSII 和 VxWorks 操作系统，请参见第 10 章。)Linux 操作系统是经过彻底测试和调试的操作系统，在全世界都普遍使用。此外，它还具有大量公开的驱动程序。(公开意味着没有所有权，任何人都可以使用。)既然有免费的比萨饼为什么还要做饭呢！如果正在设计的嵌入式系统具有 Linux 中现存驱动程序的设备，设计者可以选择 Linux(参见 http://www.linuxdoc.org)。

注意：

设备驱动程序在大多数嵌入式系统中起着重要作用，因为它们提供了应用程序和设备之间的软件层。驱动程序控制着系统中除了存储器设备和处理器之外几乎所有的设备。Linux 驱动程序被普遍使用，因为它们是经过彻底测试和调试的操作系统，并且是公开的。

4.1.4 作为设备驱动和网络函数的 Linux 内幕

Linux 具有内部函数，称为内幕。内幕的存在用于设备驱动程序和网络信息函数。嵌入式系统的 Linux 驱动程序的示例如下所示。表 4-1 列出了这些驱动程序，并给出了每一个的使用情况。

表 4-1 有用的 Linux 设备驱动程序

驱动程序类型	解　　释
字符	字符设备的驱动程序。字符设备是处理字符(字节)流的设备
块	块设备的驱动程序。块设备是处理一个块或者数据块的一部分的设备。例如，一次处理 1kB 的数据(注意：Unix 块驱动程序并不支持在读写过程中使用块的一部分)
网络	网络设备的驱动程序。网络设备是一个使用协议——例如 tty 或者 PPP 或者 SLIP——处理网络接口设备(卡或者适配器)的设备
输入	标准输入设备驱动程序。输入设备是处理来自于设备——例如键盘——的输入的设备
媒体	声音和视频输入设备驱动程序。示例有视频帧抓取设备、文字电视广播设备、无线电广播设备(实际上是声音、音乐或者语音流)
视频	标准视频输出设备驱动程序。视频设备是处理从一个系统到另外一个系统的帧缓冲区，就像字符设备或者 UDP 网络包发送设备一样
声音	标准音频设备驱动程序。声音设备是处理标准格式音频的设备
系统	平台专用的驱动程序。最近，Linux 中还出现了系统处理器专用的驱动程序。例如，专用于基于 ARM 处理器系统的驱动程序

Linux 中有套接字、套接字缓冲处理、防火墙、网络协议(例如 NFS、IP、IPv6 和以太网)和桥的内部函数。它们存在于网络目录下。它们既作为驱动程序单元工作，也可以形成操作系统网络管理功能的一部分(读者可以参考标准教科书，了解关于 UDP、PPP 和 SLIP 的意义，以及套接字函数、防火墙和网络协议。例如，参考由 Tata McGraw-Hill 2000 年编写的 *Internet and Web Technologies*，了解 PPP、SLIP、TCP、IP、以太网和其他协议)。

注意：

对于大多数普通物理和虚拟设备，Linux 操作系统具有内幕和大量可用的设备驱动程序，并且还具有用于网络套接字和协议的函数。

4.1.5 编写系统中的物理设备驱动 ISR

在 1.4.6 节中定义并解释了设备驱动程序的功能。在 2.4.4 节(示例 2-16)通过一个示例解释了 PC 串行设备(UART 设备)的寄存器和地址。编写驱动程序软件并不简单！在编写设备驱动程序之前，必须清楚以下几点。

(1) 一个设备具有三组设备寄存器，分别是数据寄存器或者缓冲区、控制寄存器和状态寄存器。

(2) 在一个设备地址上可以有不止一个设备寄存器。

(3) 通过置位控制寄存器位来初始化(配置、注册、连接)设备。

(4) 通过将控制寄存器位复位来关闭(复位、撤销注册、端口)设备。

(5) 控制寄存器位控制着设备的所有动作。控制位甚至还控制着在一个瞬间哪一个地址对应哪一个数据寄存器。例如，当在某个瞬间将 DLAB 控制位置位后，0x2F8 就对应着约数锁存器的低字节(参见示例 2-16)。

(6) 状态寄存器位反映了某个瞬间设备的状态，当执行设备驱动程序后改变。状态寄存器中的一个状态标志反映了设备的当前状态。例如，从 TRH 缓冲区寄存器中结束数据传送和从下一次传送获取新的数据之间的瞬间。发送设备空标志反映了这个瞬间(参见示例 2-16)。

(7) 系统通过设置一个状态标志(软件调用)或者通过一个信号进行硬件调用来初始化一个执行 ISR 的调用(ISR(也称为中断控制例程)执行，条件是(i)它被使能(系统中没有屏蔽)并且(ii)中断系统本身是使能的)。当编写设备控制和配置以及驱动代码时，必须具有下面的信息。

(8) 每一个寄存器的地址。物理设备硬件及其接口电路固定了物理设备地址，它们通常不能重新分配。设备称为这些地址的所有者。例如，IBM PC硬件的设备地址如下：定时器地址是0x0040~5F；小键盘地址是0x00600~6FD；实时时钟(系统时钟)地址是0x0070~7F；串行COM端口2的地址是0x02F8~2F；串行COM端口1的地址是0x03F8~3FF。注意：(i)在同一个地址上可能有输入缓冲寄存器和输出缓冲寄存器。这是因为在设备写和读指令执行时，控制总线上有可能发出不同的信号。当执行时，物理设备可以选择适当的寄存器。例如，在8051中有一个寄存器SBUF。这既是输出传送缓冲区的地址，也是输入缓冲区的地址。(ii)在同一个地址上可能有多个寄存器。在示例2-16中列出了下列情况。RBR(Receiver Data Buffer Register，接收数据缓冲寄存器)和TRH(Transmitter Holding Register，发送保持寄存器)在PC COM2串行设备的同一个地址(0x2F8)上。这个地址对于约数锁存器的低字节来说也是公用的，这个字节用来预先设置设备的波特率。当设置设备波特率时，将一个控制器设置为1，来写这个字节；在执行设备的“读”和“写”指令时，将该控制位设置为0，分别将这个地址用作RBR和TRH。

(9) 控制寄存器每一位的作用。

(10) 状态寄存器中每一个状态标志的作用。哪一个状态位被置位，反映了哪一个设备中断，是对哪一个 ISR 的调用。

(11) 状态标志和控制位是否在同一个地址上。在读指令执行过程中，处理器从这个地址读取状态。在写指令执行过程中，处理器向这个地址写控制位。

(12) 控制位和标志是否在同一个寄存器中。

(13) 设备中断是置位的状态标志是否在执行 ISR 时或者 ISR 需要将它复位时自动复位。

(14) 在返回被中断的过程之前，控制位是否需要改变、复位或者置位。

(15) 数据缓冲区、控制寄存器和状态寄存器的驱动程序需要执行的动作列表。

注意：

设备驱动 ISR 是使用设备地址和三组设备寄存器-数据寄存器或者缓冲区、控制寄存器和状态寄存器——来设计的。设备是由控制位配置和控制的。驱动程序在状态标志改变时初始化和执行。需要一个数据缓冲区、控制寄存器和状态寄存器要执行的动作列表，并在编写驱动程序代码之前准备好。

4.1.6 虚拟设备

在 1.4.5 节中定义了虚拟设备，并给出了虚拟设备的两个示例——文件和管道。除了系统中的物理设备之外，嵌入式系统中还使用了虚拟设备的驱动程序。物理设备驱动程序和虚拟设备驱动程序是可以类比的。与虚拟设备一样，物理设备驱动程序也可以具有设备打开、读、写和关闭的函数。下面来看一个文件设备和一个物理设备的类比。(i)正如文件需要打开，使能读和写操作一样，需要发送给设备一个中断调用，来初始化和配置该设备(打开、注册或者连接)。这可以通过适当地设置控制位来实现。(ii)正如需要发送给文件一个读调用一样，当需要读取设备的输入缓冲时，也要发送给设备另外一个中断调用。(iii)正如需要发送给文件一个写调用一样，当需要写设备的输出缓冲时，也要发送给设备另外一个中断调用。(iv)正如需要发送给文件一个关闭调用一样，为了准备进一步的读和写操作，需要发送给设备另外一个中断调用，禁止该设备(关闭或者撤销注册或者断开)。

在编程过程中，虚拟(软件)中断设备驱动程序的概念是很重要的。示例如下：

(1) 存储器块可以具有类似于 IO 设备缓冲区的数据缓冲区，并且可以由字符驱动程序和块驱动程序访问。当设备可以访问一个字符或者一个字符块时，分别称设备为字符设备和块设备(参见表 4-1 的第一行和第二行)。

(2) 物理设备收发器(具有输入/输出块缓冲区)或者转发器对于虚拟设备来说是等效的，称为环回设备。它通过块驱动设备驱动程序保存分配的存储器块，并从存储器中返回数据块。

(3) 存储器中有限缓冲区设备可以比作打印机缓冲区。一个例程(驱动程序)将数据流发送出去，由另外一个例程(驱动程序)读取。有限缓冲区设备是一个虚拟设备，通常称为管道设备。

(4) 程序可以保存在一组称为 RAM 磁盘的存储器块中，类似于文件系统保存在硬盘中。它是一个文件设备。RAM 磁盘是一个包含多个内部文件设备的设备。

注意：

虚拟设备是系统软件设计的一个创新概念。这些设备的驱动程序也与物理设备驱动程序一

样编写。重要的设备有字符设备、块设备、环回设备、文件设备、管道、套接字和 RAM 磁盘。设备配置等价于创建一个文件。中断时的设备激活等价于打开一个文件。设备复位等价于关闭文件。设备断开等价于释放分配给文件数据的存储空间。

4.2 系统中的并口设备驱动程序

驱动一个并口需要三个模块。一个模块通过设置控制寄存器中适当的位来初始化设备。当状态寄存器中的一个状态标志置位时，第二个模块调用中断。第三个模块是中断服务(设备驱动)程序。

下面是两个解释设备驱动程序(服务例程)行为的示例。选择的微控制器 68HC11(或者 68HC12)具有 I/O 端口的存储器映射。当发生中断时，调用设备驱动程序中断服务(处理)例程，用于输入或者输出服务，下面是驱动程序提供的服务。

(1) 驱动程序例程 portA_ISR_Input 的输入端口服务部分从输入端口 A 获取数据，就像从存储器中读数据一样，并产生一个队列，将读取的数值加入到其中。这个服务例程需要两个参数——队尾指针和队列长度(参见图 2-4(b))。示例 4-3 解释了驱动程序。

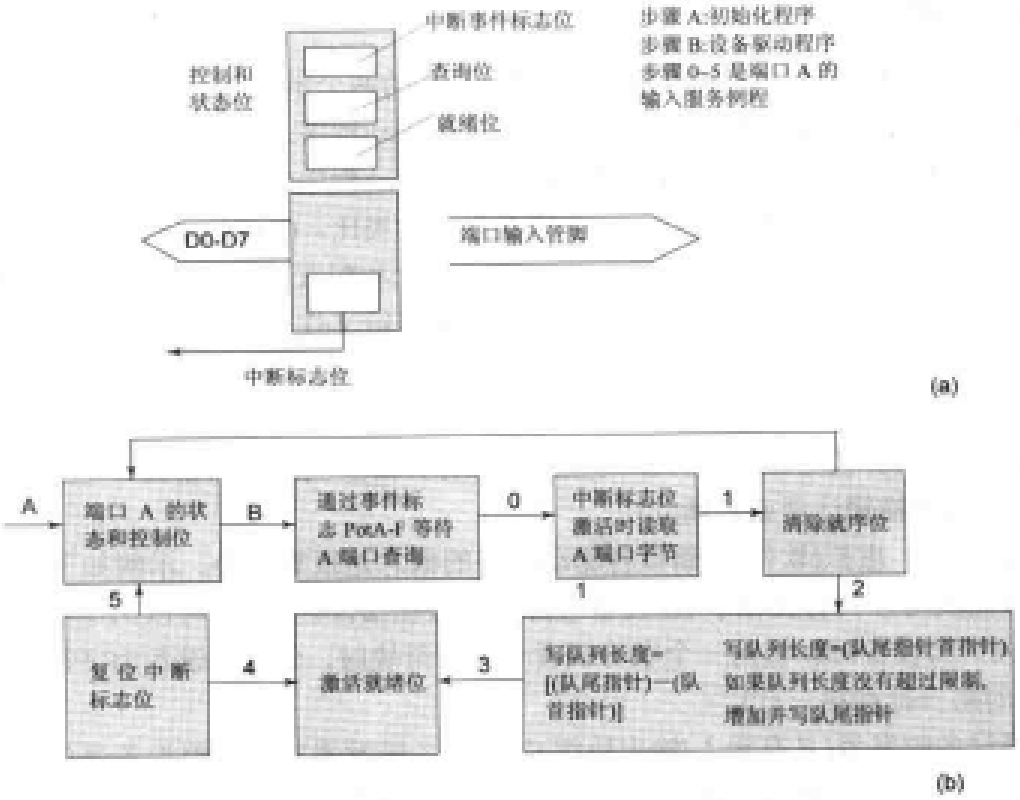


图 4-3 (a)系统调用中使用的控制和状态位以及与数据总线接口的端口管脚；(b)步骤 A 是初始化，步骤 B 是对驱动程序的系统调用，步骤 0–5 是 Port_ISR_Input 驱动程序。驱动程序从端口读取一个字节，并将它放置到建立在存储器中的队列中

(2) 第二个驱动例程 PortC_ISR 的输入端口服务部分从队列中读取(删除)发送给端口的字节，并将它写到输出端口 C 中，就像写存储器一样。这个服务例程也需要两个参数——队列头指针和队列长度。当队列满时，称为产生了异常。示例 4-4 解释了这个驱动程序。

示例 4-3

考虑单芯片 68HC11 的端口 A。这个端口没有相关的握手信号。图 4-3(a)给出了系统调用过程中使用的控制和状态位。图中还列出了与数据总线接口的端口管脚。图 4-3(b)给出了用于设备初始化的步骤 A 和处理器对驱动程序调用的步骤 B。还展示了驱动程序 portA_ISR_Input 例程的步骤 0~5。驱动程序从一个端口读取一个字节，并将它放置到建立在存储器中的队列中。设备驱动程序中端口 A 输入服务例程 portA_ISR_Input 的各个步骤用于从端口读取一个字节到队列中，这个队列建立在存储器中，用于端口的连续输入。在进行端口 A 的设备驱动程序调用之前，必须执行两个动作。(i)如下定义端口 A 的地址。当 68HC11 寄存器的存储器页地址是 0x1000 时，定义为# define port A 0x1000 /*。端口存储器地址为 0xp000，处理器复位时将页地址定义为 p。将最多为 4 位的半字节定义为 0001。(ii)处理器可操作系统将事件(状态)标志 portA_F 设置为“1”，使其能够用于端口 A 输入的设备驱动程序调用。在驱动程序 portA_ISR_Input 调用之前，设备初始化和系统调用步骤如下所述。

(i) 等待，直到 portA_F 等于“1”(图 4-3 中的步骤 A)。

(ii) 如果 portA_F 等于“1”，系统调用 portA_ISR_Input(图 4-3 中的步骤 B)。

驱动程序 portA_ISR_Input 服务例程编程将执行下列的操作。

(i) 读取端口 A。

(ii) 如果 portA_Queuelength 小于 portA_MaxQueueSize，将端口输入字节保存到指针 *portA_Queuetail 中。(a)*portA_Queuetail 是指向一个存储器地址的队尾指针，端口 A 输入的字节保存在这个队列中。(b)portA_Queuelength 是端口的当前队列长度。(c)portA_MaxQueueSize 是定义的端口 A 的最大队列长度。

(iii) 如果 portA_Queuelength 不等于 portA_MaxQueueSize，递增*portA_Queuetail 到达下一个地址。现在指针指向的是下一个例程调用将要放置字节的地方。如果相等，调用一个端口 A 的错误任务例程。

(iv) 如果可操作系统没有复位到可以对事件(对驱动程序的调用)进行响应的状态，则必须将 portA_F 复位。

在这里，portA_Queuelength 等于保存来自端口 A 的字节的队列长度。portA_MaxQueueSize 等于队列的最大可能长度。*portA_Queuetail 等于指向存储器地址的队尾指针。portA_F 是步骤 B 中使用的标志位。

示例 4-4

现在考虑另外一个在单芯片模式中读取端口 C 的示例。68HC11 端口 C 使用握手信号。图 4-4(a)给出了端口 C 的握手信号。图 4-4(b)给出了驱动程序调用的控制和状态位。图 4-4(c)给出了作为输入端口的端口 C 以及其与数据总线的接口。图 4-4(d)给出了作为输出端口的端口 C。图 4-4(e)中的步骤 A 用于初始化例程，步骤 B 用于驱动程序的系统调用，步骤 C 用于驱动程序 PortC_ISR 的执行。驱动程序从端口读取字节，并加入到队列中。队列是建立在存储器中的，用于端口的连续输入。一个外部设备激活 STRA 管脚。外部设备请求通过 STRA 将其字节传送给端口 C。当把 STRA 管脚设置为“0”将其激活后，如果控制寄存器中的 STAI(STRA 中断屏蔽位)没有置位(STAI 不为“1”)，则端口 C 发出一个响应。STRB 管脚是用于端口 C 向外部设备发送就绪状态(响应)的硬件信号。当 STAI 被编程为“0”时，只要 STRB 管脚发送了响应信号，外部设备就将字节发送到缓冲区中。只要外部设备完成了将字节发送到端口 C 的动作，就

将 STAF 置位。STAF 在一个状态寄存器中。STAF 是中断标志，当外部设备完成了将字节发送给端口 C 的动作后，该标志置位。

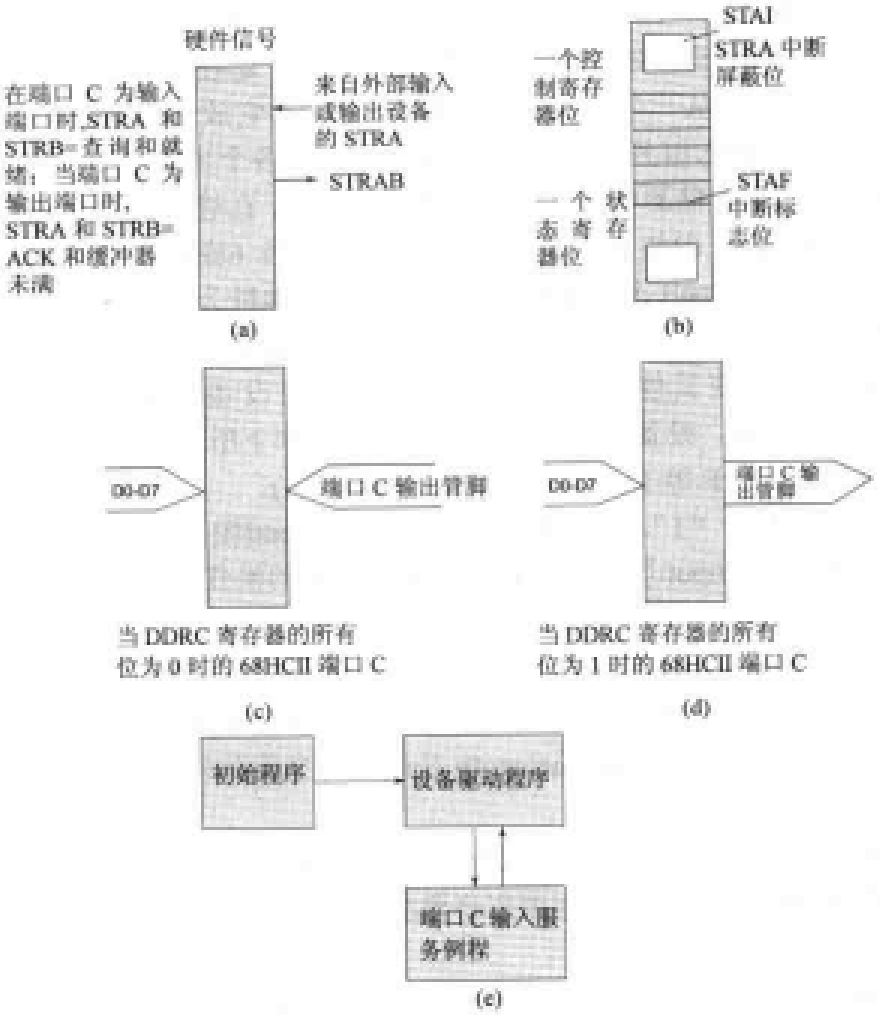


图 4-4 (a)与端口的握手信号；(b)系统调用中的控制和状态位；(c)作为输入端口的端口 C 和它与数据总线的接口；(d)作为输出端口的端口 C；(e)步骤 A 用于初始化，步骤 B 用于对驱动程序的系统调用，步骤 C 是驱动程序 PortC_ISR 的执行

当 68HC11 中页地址配置为 0xp000 时，端口存储器地址为 0xp003(p 是一个最多为 4 位的半字节)。在调用端口 C 的设备驱动程序之前，设备初始化程序需要做三件事情。(i)如下定义端口 C 地址。# define PortC 0x1003 /*p 位是 0x0001 */。(ii)将 DDRC 的所有 8 位都复位为 0，将端口 C 设置为输入并口。DDRC 是处于存储器地址 0xp007 上端口 C 的数据方向寄存器。(iii)在进行可操作系统调用时，STAI 变为“0”，启用设备中断。STAI 是 PIOC(端口 I/O 控制寄存器)的第 6 位。其地址存储器为 0xp002。

驱动程序系统调用的操作如下。

(1) 如果 STAI 设置为“1”，则读取 STAF(STAF 是 PIOC 的第 7 位。PIOC 还提供了状态位。它是控制及状态位)。

(2) 如果 STAF 设置为“1”，则调用 PortC_ISR(端口 C 服务例程)，否则等待。

(3) 没有必要对 STAF 进行软件复位，因为调用 PortC_ISR 时，68HC11 会自动用硬件将其复位。

驱动程序例程 PortC_ISR 的动作如下。

(4) 如果 quasi_bidir 位为真，则将 0xFF 写到端口 C。

(5) 读取端口 C。

(6) 如果 portC_Queuelength 小于 portC_MaxQueueSize，则将端口位保存到由 *portC_Queuetail 定义的地址上。上述缩写的解释如下。(a)*portC_Queuetail 是指向存储器地址的一个指针，来自端口 C 的字节保存到队列中。(b)portC_Queuelength 是端口的当前队列长度。(c)portC_MaxQueueSize 是为端口 C 字节定义的最大队列长度。

(7) 如果 portC_Queuelength 不等于 portC_MaxQueueSize，增加*portC_Queuetail 的值，使它指向下一个地址。现在这个指针指向的是下一次调用将要存放字节的地方。当两者相等时，调用端口 C 的异常(错误例程)。

示例 4-5~4-7 是 C 程序。这些示例解释了预处理 C 程序对设备的初始化、调用驱动程序 ISR 和系统调用函数和驱动程序 ISR。程序中以注释行的形式给出必要的解释。示例 4-5 是设备初始化、设备驱动程序调用和设备驱动程序 ISR 的 C 语言实现。它使用在线汇编(参见 5.1 节)。将 68HC11 的端口 A 的地址配置为 0x1000。PIOC 的第 7 位——STAF——的地址为 0x1001。二级端口 A 中断屏蔽是 STAI，是 PIOC 的第 6 位。

示例 4-5

```
/************** Definitions in Pre- Processor ***********************/
# define false 0
# define true 1
# define enable_Maskable_Intr ( )
# define disable_Maskable_Intr ( )
# define enable_PortA_Intr ( )
# define disable_PortA_Intr ( )
# define PortA_ISR_Input (unsigned char* )
# define volatile portA (volatile unsigned char *) 0x1000 /* Let portA be at
address 0x1000*/
# define volatile PIOC (volatile unsigned char *) 0x1001 /* Let Port IO Control
register PIOC beat address 0x1001*/
# define STAF equ %10000000 /* Let Port A interrupt flag, STAF be the 7th bit */
# define STAI equ %01000000 /* Let Port A interrupt Mask, STAI be the 6th bit */
/*****************************************************************/
void enable_Maskable_Intr ( ) {
asm {
/* Assembly Instruction for I bit setting at the CCR to enable all maskable
    Interrupts at primarylevel */Bit
   STI;
   };
};
/
*****************************************************************/
void disable_Maskable_Intr ( ) {
asm {
/* Assembly Instruction for I bit setting at the CCR to enable all maskable
    Interrupts at primarylevel */Bit
   CLI;
   };
};
```

```
/
*************************************************************************/
void enable_PortA_Intr ( ) {
asm {
/* Assembly Instruction for Bit Set (true) of portA_F in PIOC*/
   bset PIOC, STAI;
   };
};
/
*************************************************************************/
void disable_PortA_Intr ( ) {
asm {
/* Assembly Instruction for Bit Reset (false) of portA_F in PIOC*/
   bclr PIOC, STAI;
   };
};
/************ Port A Interrupt Service Routine **********************/
void portA_ISR_Input (*portAdata) {
disable_PortA_Intr ( ); /* Disable another interrupt from port A*/
/* Insert Code for reading Port A bits*/
portAdata = &PortA;
}
/* Main Program with wait instruction for interrupt flag setting and then calling
Port ISR */
void main (void) {
int n=0; int nMax = 100; /*define an index for a char array */
unsigned char [ ] portAinput; /* Let portAinput is an arrary to hold the data
from port A*/
unsigned char *portAdata;
/* Wait till an interrupt occurs and sets STAF*/
while (STAF != 1) { };
/* Execute interrupt service routine for portA*/
if(STAF == 1) {
   portA_ISR_Input (&portAdata);};
portAinput [n] = portAdata;/* Write port A input array element from the returned data*/
n++; /* Prepare for next byte from port A*/
if (n >nMax) println ("Error! Port A input bytes exceeded the Defined Array size")
else
enable_PortA_Intr ( ); /* Prepare for another interrupt from port A*/
}
```

示例 4-6 给出了同一个示例编码但不使用在线汇编的高级代码。

示例 4-6

```
/***************Definitions in Pre- Processor**********************/
typedef unsigned char int8bit;
# define int8bit boolean
# define false 0
# define true 1
# define volatile boolean IntrEnable
# define enable_Maskable_Intr ( ) {
```

```
IntrEnable = true; IntrDisable = false;
};
# define volatile boolean IntrDisable
# define disable_Maskable_Intr ( ) {
IntrEnable = false; IntrDisable = true;
};
# define volatile boolean IntrPortAEnable
#define enable_PortA_Intr ( )
# define volatile boolean IntrPortADisable
#define disable_PortA_Intr ( )
boolean STAF /* Let Port A interrupt flag is variable STAF */
boolean STAI /* Let Port A interrupt Mask, STAI be the 6th bit */
void enable_PortA_Intr ( ) {
IntrPortAEnable = true; IntrPortADisable = false; STAI = true;
};
void disable_PortA_Intr ( ) {
IntrPortAEnable = false; IntrPortADisable = true; STAI = false;
};
/*************Port A Interrupt Service Routine ***********************/
void portA_ISR_Input ( ) {
disable_PortA_Intr ( ); /* Disable another interrupt from port A*/
/* Insert same Code for reading Port A bits as in example 4.3 */
.
.
.
}
/* Main Program with wait instruction for interrupt flag setting and then calling
Prot ISR  */
void main (void) {
/* Insert same Code for reading Port A bits as in example 4.5*/
.
.
.
.
}
```

示例 4-7 给出了示例 4-6 中 In_A_Out_B 布局的 C 程序示例。

示例 4-7

```
typedef unsigned char int8bit;
# define int8bit boolean
# define false 0
# define true 1
/* Same preprocessor declaration for Port A as in Example 4.5 */
/* Insert preprocessor declaration for Port B similar to that for the port A  */
/* Note: Port B of 68HC11 is not interrupt driven */
# define volatile portB (volatile unsigned char *) 0x1004 /* Let portB be
ataddress 0x1004*/
/*********************************************************************/
/*Main Program this wait instruction for interrupt flag setting and then calling
Prot ISR */
```

```
void main (void) {
/* The Declarations of all variables, Pointers, functions here and also
initializations here*/
unsigned char *portAdata;
unsigned char *portBdata;
void portA_ISR_Input (unsigned char *portAdata);
void enable_Port A_Intr ();
/* Insert code here similar to the main function in Example 4.5 in loop to continue
  port A read while (true) */
while (true) {
/* Codes that repeatedly execute*/
/*The code ofr a function for availability check of a character at port A*/
/*Wait till an interrupt occurs and sets STAF*/
while (STAF != 1) { };
/* The code for a function for reading a character at port A*/
if (STAF == 1) {
portA_ISR_Input (&portAdata);};
enable_PortA_Intr ( ); /* Prepare for another interrupt from port A*/
}
/* The code for a function for deciphering the character read at port A*/
/* …. Insert Here the codes for deciphering portAdata */
.
.
.
/* …. Insert Here codes for encrypting fefore sending portAdata to portB */
.
.
.
/* Function for retransmit output ot PortB */
portB = &portAdata;

   };
}
```

注意：

我们从上述的示例中学习了如何编写设备驱动程序代码。并口设备驱动程序可以用特定处理器的汇编语音或者 C 语言编写，在 C 语言中，当需要改变或者使用设备寄存器时，加入在线汇编的汇编代码。

4.3 系统中的串口设备驱动程序

示例 2-16 介绍了一个串行 UART 设备。所有的 PC 都使用这种设备。示例 4-8 给出了使用 80x86 处理器的 IBMPC 中的设备驱动程序，以及具有 FIFO 输入和输出缓冲区的 UART 8250 或者新一代 UART 设备的驱动程序。UART 驱动程序在不同系统的可移植性是必需的。关于这个问题有一个 IEEE 标准，称为 POSIX(Portable Operating System Interface，可移植操作系统界面)标准。下面的驱动程序示例使用了 POSIX 定义的函数。

示例 4-8

```
1. /************** Definitions in Pre- Processor ***************************/
# define false 0
# define true 1
2. #define baud_l data
#define baud_h interrupt_enable
#define SERIALOUT 3
#define SERIALIN 4
#define I_CHAR_IN (1<<0)
#define I_TRANS_EMPTY (1<<1)
#define F_BAUD_LATCH (1<<7)
#define F_NORMAL (0<<7)
#define F_BREAK (1<<6)
#define F_NO_BREAK (0<<6)
#define F_PARITY_NONE (0<<3)
#define F_STOP1 (0<<2)
#define O_LOOP (1<<4) /* Loopback test */
#define O_OUT1 (1<<2) /* Port COM1 unused */
#define O_OUT2 (1<<3) /* Port COM2 Serial Interrupts used /
#define O_RTS (1<<1) / * RTS output signal Used*/
#define O_DTR (1<<0) / * DTR output signal Used*/
#define S_TBE (1<<5) /* Transmitter buffer empty */
#define S_RxRDY (1<<0) /* We have got a character */
#define SPEED 48 /* 2400 baud */
#define F_DATA8 (3)
Device Configuration Definition Codes
3. /* Device Initialization Codes Initialize 8250. Refer Example 2.16. We have
to define Output
port and input port addresses */
COM2 registers:
2F8 Txbuffer (THR) DLAB=0 (Write) (DLAB is at LCR control register)
2F8 Rxbuffer (RBR) DLAB=0 (Read) (Read Write Addresses are same for THR and TBR)
2F8 Divisor Latch LSB DLAB=1 (Define Baud Rate Divisor Latch when control bit
DLAB =1)
2F9 MSB DLAB=1
2F9 Interrupt enable register (IER) ( When DLAB = 0)
2FA Interrupt identification register (IIR)
2FB Line control register (LCR)
2FC Modem control register (MCR)
2FD Line status register (LSR)
2FE Modem status register
Device Activation Codes
4. /* Device Initialization Codes - Set Control Bits for Interrupt Mask and for
Allocate Interrupt Vector Addresses for the Device Interrupts. We can define
Mask serial lines to interrupt IRQ3 or IRQ4. Set interrupt vector and set
interrupt mask: COM2 interrupt is at IRQ3 in Interrupt controller of 80x86. Recall
Interrupt vector is a memory address where the driver ISR is located. 80x86 can
be programmed to run in two modes. Recall that number of real mode
processor interrupts = 8 and number of protected mode processor interrupts =
32. */
/* Codes for Real Mode of Processor */
```

```
setvect (0xB, serial_interrupt); /*0xB is Serial Interrupt Vector at 80x86 in
Real Mode /
5. /* Define function outportb (q, p) by Assembly Language instructions, move
variable p in 80x86 AL register and send AL to the output at address q */
Outportb (q, p) means:
mov al, p
out q, al
outportb (0x21, inportb (0x21) & 0xE7);
outportb (0x20, 0x20); /* End of Intterrupt processing =EOI*/
6. enable ( ); /* Enable Interrupts */
7. /* Declare Eight Character data structure for data at the device registers *
struct sio {
char data; /* Data register RBR THR*/
char interrupt_enable; /* Interrup enable register IER*/
char interrupt_id; /* Interrupt identification register IIR*/
char format; /* Line control register LCR */
char out_control; /* Modem control register MCR */
char status; /* Line status register LSR */
char i_status; /* Modem status register MSR /
char temp; / * Dummy character for non existent 0x2FF address */
};
8. /* Define Memory address*/
#define COM ( (struct sio near *) 0x2F8)
9. /* Set Serial line Interrupts Set */
10. outportb ( (int) & COM- > interrupt_enable,I_CHAR_IN | I_TRANS_EMPTY);
11. /* Set line parameters: ( DLAB=1)
outportb ((int)&COM->format,F_BAUD_LATCH|F_NO_BREAK|
F_PARITY_NONE|F_STOP1| F_DATA8);
12. /* Set Baud Rate */
outportb ((int)&COM->baud_l,SPEED&0xff);
outportb ((int) & COM->baud_h,SPEED >>8);
13./********************* Initialize Normal Serial Data Output
************************/
outportb ((int) &COM->format,
F_NORMAL|F_NO_BREAK|F_PARITY_NONE|F_STOP1|F DATA8);
outportb ((int) &COM-> out_control,O_OUT1|O_OUT2|O_RTS|O_DTR);
14. /* Reset the Serial Line Device */
(void ) inportb ((int) &COM->data);
(void ) inportb ((int) &COM -> interrupt_enable);
(void ) inportb ((int) &COM -> interrupt_id);
(void ) inportb ((int) &COM -> status);
15. /* Enable Interrupt Controller as initialization of device is finished by
above steps. */
outportb (0x20, 0x20);
enable( );
16. /* For Mailbox concept refer Section 5.7.3 */
/* Transfer character to mailbox circular */
int GetSerial ( )
{ int x;
disable ( );
x = GetBuff(&MailBox [SERIALIN]); /* Read char from serial port */
```

```
enable ( );
return (x);
}
Device Driver Codes
17. /* Transfer a character to Serial Line Device by writing to serial port at
0x2F8 */
int transferring;
int PutSerial (char c) /* Write char to serial port */
{
disable ( );
if (Going)
{
if ( (PutBuff (&MailBox [SERIALOUT], (int) c)) == -1)
{enable ( );return (-1);}
else { enable ( ); return (1);} }
else {
transferring=1;
outportb ((int)&COM->data,c); /* Start I/O */ }
enable ( );
return (1);
}
/
*************************************************************************/
void interrupt serial_interrupt ( )
{
int int_status;
disable( );
int_status = inportb ((int) &COM->status);
inportb ((int) &COM->interrupt_enable);
inportb ((int) &COM->interrupt_id);
if ( (int_status & S_RxRDY) != 0)
{PutBuff (&MailBox[SERIALIN],inportb ((int)&COM->data) & 0x7f); }/* Receive
char */
if ((int_status & S_TBE ) != 0)
{ if ( MailBox[SERIALOUT].Head == MailBox[SERIALOUT].Tail) transferring= 0;
else outportb((int)&COM->data,
(char)GetBuff (&MailBox[SERIALOUT]));
}
outportb( 0x20, 0x20);
enable ( );
} /* End of the Driver ISR Codes */
/
*************************************************************************/
```

注意：

我们通过具有 UART 8250 设备的 PC 80x86 处理器系统示例学习了串口设备驱动程序代码的编写。

4.4 内部可编程定时设备的设备驱动程序

在 3.2 节中曾经提到以下几点：(i)通常在需要定时器的系统中至少有一个硬件定时器作为内部设备。(ii)使用硬件定时器的超时(嘀哒)，可以驱动任意多个软件定时器。基于 80x86、80960 和 PowerPC 的嵌入式系统需要片外硬件定时器，因为这些处理器芯片中没有片上定时器。8051、68HC11 和 80196 微控制器系列(参见附录 C)有片上定时器。

编写定时器设备驱动程序需要理解定时器控制寄存器和状态寄存器中的每一位。定时器设备驱动程序编写的一个重要步骤，是对当前的一个或者两个控制寄存器和状态寄存器中每一位的编程。程序员还必须考虑以下情况。(i)设备可能有屏蔽位。将屏蔽位置位后，中断被禁止；复位后，中断使能。其行为与使能位正好相反。程序员还必须记住，某些中断不能被禁止(不能屏蔽，即可屏蔽中断)。这些使能或者屏蔽位是二级位。还有一个整个中断系统使能位，类似于所有可屏蔽中断源的主关键字(一级位)。驱动程序还必须设置这一位。

步骤 I

将 RTC 的计数输入个数 numTicks 写入到保存定时器复位数值的寄存器中。

步骤 II

将状态寄存器中的定时器状态标志位设为“复位”状态(当读状态标志位后，设备自动将该标志位复位的情况)。

步骤 III

写控制寄存器中的每一位。将控制寄存器中的一级和二级使能位写为真，将其他位按照使用情况写入。要使设备能够工作，必须写设备的使能位。对当前模式寄存器中每一位的定义也是必需的。驱动程序要对当前的一个或者两个控制寄存器、状态寄存器和输出比较寄存器中的每一位进行编程。自激计数器用作时钟设备。68HC11 中的设备编程过程如下：

示例 4-9

(i) 步骤 a：定义保存 FRC 的一个实例的输出比较寄存器，设置 OC 标志位以及 OC 中断的发生。

(ii) 步骤 b：如果设备没有自动复位，则读取状态寄存器中的标志位后必须将其复位。当前的标志位可能有：FRC 溢出状态标志位、OC 标志位、ICAP_F 标志位、RTC 标志位和 SWT 标志位。在读取状态寄存器后，需要将这些位复位。

(iii) 步骤 c：定义控制寄存器位。在这里，必须定义当前的每一位。这些位有：计数输入时钟的预引比例因子位、溢出中断使能位、RTC 中断使能位、OC 中断使能位、OC 使能位、OC 输出级位、ICAP 使能位、ICAP 输入沿位、ICAP 输入位、ICAP 中断使能位、SWT 使能(位)、SWT 中断使能位。

(iv) 步骤 d：如果一级中断使能位已经被禁止，要将其激活。

4.5 中断服务(处理)机制

4.5.1 硬件和软件相关的中断源

硬件中断是与特定处理器相关的。 不仅设备中断需要 ISR，软件相关的中断也需要。软件中断的一个示例是处理器产生的“异常(exception)”中断。参见示例 4-10。

示例 4-10

假设在一个程序的执行过程中出现了一条除数为 0 的指令。当发生这种情况时，必须执行相应的 ISR。这个 ISR 应该在屏幕上显示“在……出现了除数为 0 的情况”，并终止当前程序的执行。

处理器当前正在执行的用户程序不知道何时 ALU(算术逻辑单元)会发出这样的内部错误标志(一个硬件信号)。利用中断机制，执行特定的中断服务例程，对除数为 0 的错误信号进行处理。产生这样的信息后，程序执行完当前指令后就会被中断，然后将标志位复位，并执行 ISR。

硬件中断源可以是内部的，也可以是外部的。每一个中断源(如果没有被屏蔽)都需要暂时从当前执行程序跳转移到相应的 ISR。在不同的版本和系列中，内部中断源和设备是不同的。表 4-2 列出了来自多个中断源的 5 类硬件和软件中断。在一个给定的系统中，并不是表中列出的所有中断源类型都存在或者启用。此外，系统中可能还会有其他特殊类型的中断源。

4.5.2 软件错误相关的硬件中断

软件错误可能是由非法的或者没有实现的操作码造成的。例如，在 68HC11 中有一个非法指令码陷阱(illegal opcode trap)。发生这种错误后就会产生指向一个向量地址的中断。在 80196 中，没有实现的操作码错误会产生指向一个向量地址的中断。在 80x86 中有软件相关的中断。例如，除数为 0(类型 0)和溢出(类型 2)(关于类型的定义请参见 4.5.3 节)。这两种类型的中断是由处理器中的 ALU 硬件产生的。每一个处理器都有一个特定的指令集。该处理器是专为这个指令集设计的。非法代码(软件中的指令)是一条指令，该指令不与指令集中的任何指令相关。当取回了这样一条指令后，在某些处理器中就会产生一个中断。错误相关的中断也称为软件陷阱(或者软件异常)(注意：在 8086 和 80x86 处理器中，标志位“陷阱”是一个不同类型的中断源。在这种情况下，陷阱具有一个完全不同的意义。它意味着异常的运行条件，导致了当前运行例程的陷入)。软件异常是一个运行异常条件，如果在运行时发生(置位)，就会跳转到另外一个异常例程，此例程是在该条件发生时调用。例如，队列 (参见图 2-4(b))满是一个运行条件异常。该异常会导致程序转跳到称为异常的例程，该例程将启动适当的行为。异常是处理运行错误的重要例程。遇到异常时的行为称为抛出异常。

表 4-2　5 个系列 28 种类型的中断源

软件错误相关的中断源(异常或者 SW 陷阱)	软件代码相关的中断源	内部硬件设备中断源	具有内部向量地址产生的外部硬件设备	提供 ISR 地址、向量地址或者类型的外部硬件
1. 溢出	11. 断点指令(INTn、INT0 和 80x86 中的类型 3、68HC11 中的 swi)	21. 并口和 UART 串行接收端口(68HC11 中的噪声、超载、帧错误、IDLE、RDRF)	25. 在最初几个时钟周期中无可屏蔽、可声明的管脚(68HC11 中的 XIRQ)	28. INTR(8086 和 80x86 中)
2. 除数为 0	12. 调试陷阱标志位(8086 中的 TF)	22. 同步接收字节完成	26. 不可屏蔽管脚(8086 和 80x86 中的 NMI)	
3. 下溢	13. 异常指令(例如 ARM7 中进入管理模式的指令)	23. UART 串行发送端口发送结束、TDRF 空字节同步发送结束	27. 可屏蔽管脚(68HC11 中的 IRQ 和 8051 中的 INTO)	
4. 非法指令码	14. RTC 驱动软件定时器	24. ADC 转换开始、转换结束		
5. 程序员[1]定义的异常[2](信号)	15. 输入驱动软件定时器			
6. 任务阻塞	16. 脉冲累加器溢出			
7. 任务结束	17. 实时时钟超时			
8. 事件相关	18. watohdog 定时器复位			
9. 信号量任务和释放	19. 定时器超时后溢出			
10. 用作状态标志位来定义软件中断源的布尔变量	20. 具有输出比较寄存器或者定时器输入捕获的定时器比较			

(1) 括号中是特定处理器的实例。

1. 信号有时用作异常(例如在 VxWorks RTOS 中。参见 10.3 节)。信号或者异常是由于某些条件满足、得到某些结果或者程序运行时的输出所产生的一个中断。

2. 有两种类型的异常。一种是由处理器内部产生的。例如，80x86 中的除数为 0。第二种是用户定义的。

注意：

(1) 软件错误相关的硬件中断需要对某些错误——例如除数为 0 或者非法操作码——进行响应。

(2) 异常对于处理运行时错误是必需的。“异常”类似于 ISR。当产生中断时执行 ISR，并为中断服务。软件运行并发生抛出异常)异常条件时执行异常，该异常为所遇到的异常条件服务(处理)。

4.5.3 软件指令相关的中断源

有些指令是用于中断的。它们与下面的调用指令是不同的。在 68HC11 中有一条引起软件中断的指令 SWI。在 80x86 中有一条单字节指令 INT0。它产生类型 0 中断，该中断相关的向量地址是 0x00000H。INT0 产生了 8086 和 80x86 在除数为 0 时由硬件产生的类型 0 中断。考虑另外一个 8086 和 80x86 单字节指令 TYPE3(相关的向量地址是 0x00004*3=0x0000CH)。它产生了一个类型 3 中断，称为断点中断。该指令类似于 PAUSE 指令。PAUSE 是一种临时的停止。它使得用户在这期间执行一些内部操作，并在断点处理完成后返回到指令的执行。考虑 8086 和 80x86 中的另外一个双字节指令 INT n，n 代表的是类型，是第二个字节。该指令的意思是“产生类型 n 中断”，并从向量地址 0x00004*n 处获得 ISR 地址。当 n=1 时，代表了 8086 和 80x86 中的单步执行。还有一个称为陷阱的状态标志，由 TF 表示。这个标志分别处于 8086 和 80x86 中的 FLAG 寄存器和 EFLAG 寄存器中。这意味着当 TF 在执行完每一条指令被自动置位时(写为“1”)，处理器行为导致类型 1 中断的重复产生。处理器每次从向量地址 0x00004(与类型 1 中断地址相同)中取 ISR 地址。INT 1 软件指令也会产生类型 1 中断。该指令与 TF 标志置位时指令结束所产生的指令相同。在 80196 中有一条用于调试软件的指令，称为陷阱。陷阱指令执行后，直到下一条指令为止，所有的中断源都不能中断和服务。

有时候需要并行地运行几个例程，如同需要并行地运行多个设备和中断源的 ISR 一样。此时可以使用一个布尔变量，而不是状态标志位，当该变量置位时，产生了一个请求运行软件中断服务例程(software interrupt service routine，SWISR)的请求信号。可以用一组布尔变量代替状态寄存器，或者用于中断挂起寄存器，发出多个运行 SWISR 请求。在 80x86 中使用指令 INT n_{type} 运行 SWISR。通过定义适当的 n_{type} 数值，使之在标志位置位的情况下执行。另外一组布尔变量用作中断控制位(屏蔽(禁止)位)。另外一种中断机制可以这样编程。正如 CUP 在每一条指令执行完后要观察查看寄存器或者中断挂起寄存器一样，为了提供一种中断机制，软件中断(software interrupt，SWINT)调度程序(或者前台程序)必须周期性地检查上述变量集，使多个 SWISR 并发执行。

注意：

系统也可以有复杂的软件中断源。复杂意味着不与类似于 80x86 中的 INT n_{type} 指令相关，而是与一组指令(SWISR)相关。示例应用有任务调度和任务时间片(参见 9.4.4 节)。软件定时器可以在程序执行时由一个事件中断。其对来自于一个中断源的事件计数，然后产生中断。在任务阻塞、任务结束、系统信号捕获和释放时可以产生中断。

注意：

在嵌入式系统中使用软件指令相关或者软件定义条件相关的软件中断。它们对于设计类似于错误处理 ISR 和软件定时器驱动的 ISR 是必需的。这些中断也称为异常或者信号。除了设备驱动程序之外，系统的操作系统还必须提供中断服务机制。

4.5.4 内部设备相关的硬件中断

表 4-2 的第三列给出了与内部设备相关的一般硬件中断。

4.5.5 中断向量

(1) 系统具有类似于片上定时器和片上 A/D 转换器的内部设备。最常使用的是内部设备(中断源或者中断源组)，它们自动产生每一个设备的中断向量地址 ISR_VECTADDR。这些地址对于具有该内部设备的微控制器或者处理器来说是特有的。当发现一个中断源将一个状态标志位置位后，处理器从 ISR_VECTADDR 中的内容获得 ISR 地址。系统软件设计者必须通过定位器或者设备编程器将每一个 ISR_ADDR 的字节存放到 ISR_VECTADDR 中。ISR_ADDR 是设备驱动 ISR 的起始地址。实际上，在 ROM 映像中，存在着一个系统中各个中断源的中断向量表。这个表协助每一个内部设备的多中断源服务。ISR_VECTADDR 的每一行给出提供列中相应 ISR_ADDR 的字节(向量表在存储器中的位置取决于处理器。在 68HC11 中，它位于较高的处理器地址 0xFFC0~0xFFFB 中。在 80x86 处理器中，它位于存储器的最低地址 0x00000~0x003FF 中。在 ARM7 中，其起始地址是最低的存储器地址 0x0000)。外部设备也可以通过数据总线将 ISR_VECTADDR 发送给处理器。

(2) 80x86 处理器的机制是使用向量地址。这种机制是，在从外部搜索 ISR_VECTADDR 的情况下，只需要一个短的地址类型编号，称为 ISR_type(ISR_type 与 0x00004 相乘，得到字节的地址。根据这些字节产生地址 ISR_VECTADDR。从这个地址处取 ISR_ADDR)。

(3) 一个设备甚至可以有多个中断源。此时设备不必发送 ISR_VECTADDR。

(4) 一个中断源组具有相同的 ISR_VECTADDR。例如在 8051 中，TI 和 RI 是相同组中的两个中断源，具有相同的 ISR_VECTADDR(当发送串行缓冲寄存器结束串行发送时，产生 TI 中断，当缓冲区从串行接收器中接收到一个字节后产生 RI 中断。当一个中断源组的向量地址或者 ISR 地址相同时，ISR_ADDR 中的 ISR 首先必须识别中断源。识别是通过标志位或者其他信息进行的)。

注意：

中断向量表是中断服务机制的一个重要部分，与特定的处理器相关。它为处理器提供了一个中断源或者一组中断源的 ISR 地址。

4.5.6 根据可屏蔽和不可屏蔽的中断分类

在一个系统中有三种类型的中断源。(i)不可屏蔽的。例如，PC 中的 RAM 奇偶错误和类似于除数为 0 的错误中断。这些中断必须处理。(ii)可屏蔽的。这些中断可以被临时禁止，使具有较高优先级的函数继续执行而不被打断。(ii)一个中断源只有在复位后的几个时钟周期内定义为不可屏蔽的才是不可屏蔽的。某些处理器——例如 68HC11 包含这种类型的中断。例如，68HC11 中的外部中断管脚 XIRQ 中断。只有在 68HC11 复位后的几个时钟周期内定义为不可屏蔽的，

XIRQ 中断才是不可屏蔽的。

4.5.7 所有可屏蔽中断源的激活(未屏蔽)和禁用(屏蔽)

在设备中可以有控制位。可以有一个称为一级位的位激活或者禁用可屏蔽中断的全部中断系统。一组位可以激活或者禁用中断系统中特定的中断源或者中断源组。通过执行用户软件中的适当指令，写一级或者二级使能位激活(或者屏蔽)所有或者部分的可屏蔽中断源。

4.5.8 中断挂起寄存器或者状态寄存器

对先前来自于一个中断源的中断的识别是通过下列方法之一进行的：(i)中断挂起寄存器(IPR)中的处理器挂起标志位(布尔变量)，这个变量是由中断源置位(由硬件置位)，并当随后转移到相应的 ISR 且开始相关的中断源服务时，由内部硬件立即自动复位。(ii)状态寄存器中的局部级标志位，该寄存器可以为一个或者多个中断源或者一组中断源保持一个或者多个状态标志位。IPR 和状态寄存器有以下的不同之处。状态寄存器是只读的。(i)状态寄存器位(标识标志位)是只读的，并且是在读的过程中清除的(自动)。IPR 位或者由相应的 ISR 服务清除(自动复位)，或者由复位的写指令复位。(ii)IPR 位可以由写指令置位，也可以由一个中断的发生置位。状态寄存器位只能由中断源硬件复位。(iii)IPR 位可以与来自一组中断源的挂起中断相关，但是标识标志位(位)与多个中断中的每一个中断源是分离的。

中断标志位的特性如下所述。在多中断源的情况下，每一个中断源的发生标识都必须有一个单独的标志位。标志位在中断发生时被置位，并且(i)它存在于处理器的内部硬件电路中或者在 IPR 中或者在状态寄存器中。(ii)它可以被指令读取，但是只能被中断源硬件写。(iii)它在读的过程中复位(被去活)。这是为了使该标志位能够反映来自同一个中断源的下一个中断的发生，在某些硬件设计中提供的自动复位特性。(iv)如果立即置位，并不一定意味着它会随后被识别和处理。当存在与该标志位相关中断的屏蔽位时，除非屏蔽(或者使能)位随后做了修改，否则即使是标志位被置位，也会被处理器忽略。这样就有可能禁止一个不应发生的中断，也就禁止了向相应 ISR 的跳转。

4.6 上下文和上下文切换周期、最终期限和中断延迟

4.6.1 上下文、延迟和最终期限

如果需要进行多任务编程，系统软件设计者必须注意下面的细节。

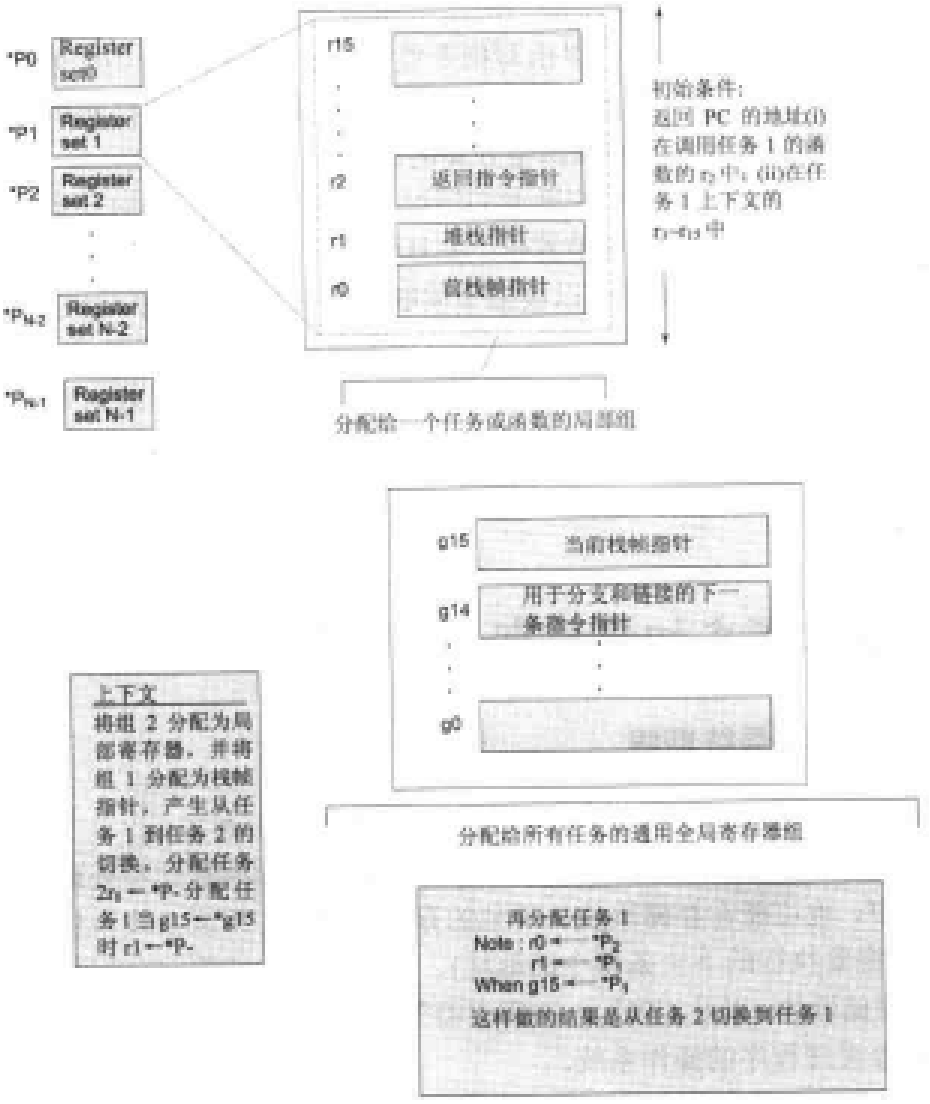
(1) 一个程序的上下文包括程序计数器以及程序状态字、堆栈指针和处理器寄存器。上下文可能在寄存器组中，也可能在存储堆栈的单独的存储器块中。组或者块中保持着关于程序计数器(反映了该任务将要执行的下一条指令的地址)、分配给一个特殊前台程序或者将要被中断过程的 ISR、ID 的存储器块，以及 CPU 状态(寄存器等)(参见 8.1 节)。上下文的组成是什么呢？这依赖于处理器或者管理程序的操作系统。

(2) 上下文切换意味着保存被中断例程或者任务的上下文，加载被调用例程或者任务的新的上下文。上下文切换所需要的时间必须在一个周期之内，称为中断延迟周期——T_{lat}。在图 4-1 中给出了保存和获取步骤的实现；图 4-2 中的函数或者 ISR 调用和返回步骤 a6、a7、a8 和 a9 都比较快，这是因为在某些处理器中存在特殊硬件。关于某些处理器中的快速上下文切换示例

在下面给出。

80960 嵌入式处理器程序分配了一组局部寄存器(*p0、*p1……)在调用指令执行时使用。每一个寄存器组保存着一个例程或者前台程序的各种局部变量。当执行调用程序时，就产生了一个内部堆栈帧，并且将组的先前例程或者前台程序的分配解除(内部堆栈的意思是堆栈不在外部 RAM 中，而是与处理器内部相关)。在解除分配之前，组的所有局部寄存器保存到堆栈帧中。将另外一个局部寄存器组分配给被调用程序。返回时，堆栈帧返回分配的局部寄存器组中的变量，并将堆栈帧解除分配。保存和获取的速度很快。这是因为，首先，没有将外部 RAM 用作堆栈；其次，当从一个例程向另外一个例程进行跳转的时候，只有指向堆栈帧中上下文的指针数值发生了变化或者进行了交换。

图 4-5 解释了 80196 和 80960 处理器中分配和重新分配的过程。这个图还给出了一个从任务 1 到任务 2 的上下文切换示例。只有引用从一个寄存器窗口(组)交换到另外一个寄存器组(或者堆栈帧)。一旦程序员将不同函数、ISR 和任务的上下文预先加载到寄存器组中后，上下文切换只需要几个时钟周期(关于任务的定义请参见 8.1 节)。



注意:代码执行时，r1 改变但 r15 不变，使得当获取时，保存的变量可以被压入堆栈和弹出堆栈

图 4-5　在基于 80196 和 80960 的系统中使用寄存器组和堆栈帧的分配和再分配进行上下文切换

在 ARM7 处理器中，发生 ISR 调用的上下文切换如下所述。(i)将中断屏蔽(使能)位置位。将低优先级的中断禁止。(ii)保存程序计数器(PC)。(iii)将当前程序状态寄存器(Current Program Status Register，CPSR)复制到一个已保存的程序状态寄存器(Saved Program Status Register，SPSR)中，CPSR 中保存了新的状态(中断源数据或者信息)。(iv)PC 从向量表中获得中断源的新数值。ISR 返回到前一上下文的上下文切换过程如下所述。(i)返回程序计数器(PC)。(ii)将对应的 SPSR 复制回 CPSR 中。(iii)将中断屏蔽(禁止)标志位复位(启用先前禁止的低优先级中断)。

(3) 对于每一个中断源，都应该有一个 ISR 指令保持挂起的最长周期。这个周期定义了服务必须完成的最终期限(参见 4.4.1 节中所描述的视频处理器系统的示例。参见 1.3.1 节在嵌入式系统中为了节约功耗所规定的最终期限信息)。

中断延迟周期 T_{la} 是下列周期的和。

i. 所占用的时间包括响应和初始化 ISR 指令的时间。这包括保存或者切换上下文(包括程序计数器和寄存器)加上恢复上下文的时间。例如在 ARM7 处理器中，这个周期等于 2 个时钟周期，加上结束正在执行指令的 0~20 个时钟周期，再加上异常数据的 0~3 个时钟周期。(其中的差别在于中断有可能正好是一条指令结束前或者一条指令开始后发生。在 ARM7 中，执行时间最长的指令需要 20 个时钟周期。在特定的情况下，异常数据(将 CPSR 复制到 SPSR 中)有可能需要，也可能不需要。如果需要的话则需要 3 个时钟周期)。因此在 ARM7 处理器中，最小时间是 4 个时钟周期，最长是 27 个时钟周期。因此计算 T_{la} 的时候，需要考虑最长的 27 个时钟周期。

ii. 这个时间包括为优先级高于当前中断源的中断进行服务的时间。

iii. 根据程序员的编程方式，这个时间可以包括进去，也可以不考虑，也就是：在任何源代码或者函数中，禁止中断服务的最长时间(这段代码有可能包含着访问临界区的指令代码)。

图 4-6(a)给出了一个中断的最终期限、延迟、ISR 服务的周期。

考虑对于串行同步或者 UART 通信端口，当缓冲区的空间无法保存更多的字节，并且在端口的下一个中断到来之前，必须对当前的中断进行处理的情况。如果没有遵循 ISR 尽量短的规则，那么当前正在执行的 ISR 就有可能在最终期限 T_{max} 之前不能返回。在示例 4-1 中，In_A_OUT_B 传送字节的时间间隔为 170μs，内部网络中断服务的 T_{max} 为 170μs。如果给定的嵌入式系统中的中断延迟是 T_{lat}，那么开始执行 ISR 的最长时间间隔将是 T_{max} 减去 T_{lat}。令 T_{ISR} 作为对 In_A_OUT_B 中的指令进行服务的时间。T_{ISR} 应该小于 T_{max} 减去 T_{lat}。

中断源的中断延迟和最终期限帮助进行各个例程和函数的时间分配。图 4-6 按照服务优先级的顺序列出了 5 个示例中断：RTI、RI、STATUS_I、FIFI_4thEntry 和 FIFO-Full。在示例 4-11 和 4-12 对这些中断进行了介绍。

示例 4-11

假设一个串口从网络中读取字节，当缓冲区满时将产生一个中断。设当前有一个 8 字节(第 0~7)的 FIFO。在处理器 80196 中，当缓冲区在最终期限内没有被读取，就从接收数据缓冲区传送一个字节到 FIFO。假设有三个中断源，：缓冲区收到一个字节、FIFO 半满、FIFO 满。将这个三个中断源分别命名为 RI、FIFI_4thEntry 和 FIFO-Full。对这些中断源服务的最终期限 T_{max} 是频率的倒数。使用 SWT，必须检查网络的状态，以获得端口的状态。将这个中断源命名为 STATUS_I。

表 4-3 的第 2 列列出了网络以每秒 1000 个字节的速度读取的情况。假设三个中断源的 T_{lat} =340μs。并假设对中断源 RI、FIFI_4thEntry 和 FIFO-Full 进行服务的 T_{ISR} 分别为 20μs、90μs 和 170μs。假设有一个中断源 RTI(real time Clock Interrupt，实时时钟中断)，显然该中断源具有较高的优先级。那么对 RTI 进行处理的剩余时间为 $T_{max}-T_{lat}-T_{ISR}$(第 3 列)。

示例 4-12

设 RTI 的中断延迟 $T_{lat}^{'}$ 为 120μs。当没有定时器超时时，ISR_TRI 需要 $T_{ISR}^{''}$ =200μs 的时间将计数输入给 4 个 SWT；当一个定时器超时时，这个时间为 $T_{ISR}^{'''}$ =300μs(参见示例 3-5)。我们必须选择两个时间中较大的一个。那么优先级较高的 STATUS_I 中断的 ISR 指令获得服务的最小时间 $(T_{ISR}^{'v})_{max}$ 是多少？假设 STATUS_I 中断源的 $T_{lat}^{''}$ =80μs。那么 $(T_{ISR}^{'v})_{max}=T_{max}-T_{lat}-T_{ISR}-T_{lat}^{'}-T_{lat}^{''}$(第 4 列)。

表 4-3　中断源的参数表

具有两条较高优先级中断的中断源最终期限	T_{max} (μs)频率为 1000B/s	对 RTI 进行服务的时间(μs)	对查看网络状态的 ISR 指令进行服务的时间(ms)
RI	1	640	160
FIFI_4thEntry	5	4570	4970
FIFO_Full 最大中断延迟	9	8490	7990

多中断源软件设计的一个规则是使得 ISR 尽量短。为什么？因为这样能够对挂起中断进行服务，将可以推迟的函数推迟到后面执行。如果不遵循这个规则，就不能保证某些特殊中断源在最终期限(最大允许的挂起时间)内被服务。随后在示例 4-6 中将解释对 ISR 函数进行排队的使用。这样就能够缩短 ISR 的执行周期，进而满足最终期限的要求。

注意：

程序在运行过程中的每一个瞬间都有一个上下文。上下文反映了 CPU 状态(程序计数器、堆栈指针、寄存器和程序状态(应该被另外一个例程修改的变量))。当调用另外一个 ISR、任务或者例程时，在进行上下文切换之前，必须保存当前上下文。某些处理器通过为上下文提供内部堆栈帧或者局部寄存器组，加快了上下文切换的速度。快速的上下文切换减小了中断延迟，并且能够满足每一个任务服务最终期限的要求。如果系统中使用的处理器没有内部堆栈帧，操作系统会提供存储器块作为堆栈帧。

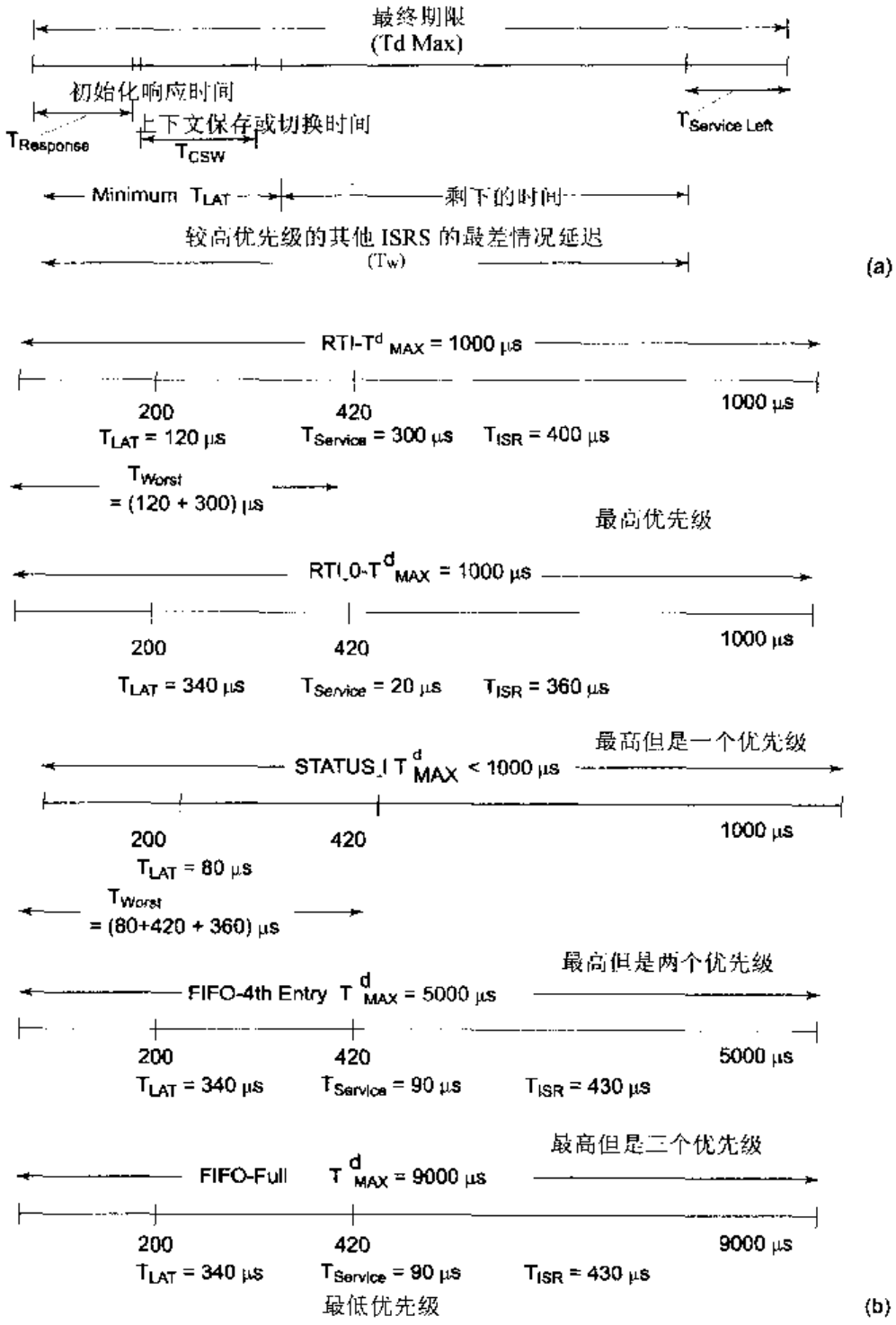


图 4-6 (a) 一个中断的最终期限、延迟、服务和 ISR 执行周期；(b)按照服务优先级的顺序排列的 5 个示例中断：RTI、RI、STATUS_I、FIFI_4thEntry 和 FIFO-Full。

4.6.2 从上下文保存的角度对处理器中断服务机制的分类

8051 的中断服务机制是，当发生了中断服务之后，处理器将处理器寄存器 PCH 和 PCL 压入到存储器堆栈中。8051 系列处理器不保存程序的上下文(除了必须保存的程序计数器之外)，上下文的保存只能通过特殊的指令集进行。例如使用 Push 指令。这样能够加速 ISR 的起始和返回速度，但是是有代价的。如果上下文在剩下的 ISR 指令执行过程中的服务或者函数调用时

被修改，上下文切换是程序员的任务。

68HC11 的中断服务机制是，每当发生一个中断服务，就将处理器寄存器保存到堆栈中。保存的顺序是 PCL、PCH、IYL、IYH、IXL、IXH、ACCA、ACCB 和 CCR。因此在用户程序中没有保存上下文的指令的情况下，68JC11 自动保存程序的处理器上下文(不是程序变量上下文)。因为上下文切换需要占用处理器时间，ISR 的开始和返回速度比较慢，但是其最大的优点是，处理器寄存器上下文切换不是程序员的任务，并且上下文不会在服务或者函数调用时被修改。

80960 的上下文切换过程是很强大的(参见图 4-5)。ARM7 还提供了快速的上下文切换。

4.6.3 使用 DMA 通道帮助缩短中断延迟周期

小中断延迟服务的一个优点是当存在多个 IO 中断源时，可以使用 DMA 通道。这样中断服务例程从开始到结束的周期会很短，因为初始化 DMA 进行中断服务的 ISR 只需要对 DMA 的命令、数据计数、存储器块地址和 I/O 总线起始地址寄存器进行编程(参见 2.6 节)。

注意：

使用 DMA 通道进行 IO 服务，代替处理器中断驱动的 ISR 是一种高效的方法。这是因为 DMA 传输发生在系统总线空闲的时间内。

4.6.4 满足服务最终期限的优先级分配

需要对级编号和优先级顺序进行分配。这一点可以通过下面两种情况理解。(a)假设总共有 7 个中断源：watchdog 定时器、定时器 1 输入捕获、定时器 2 输入捕获、定时器 1 输出比较、定时器 2 输出比较、串口输入和 A/D 转换。A/D 转换的优先级应该最低，因为定时器和串口中断应该比 A/D 的服务快。Watchdog 定时器的优先级应该最高。因此，在编程之后，应该对程序重新初始化和运行。此外，内部定时器需要比外部串口更快的服务。在较高优先级中断源接连激活的情况下，中断调用应该是嵌套的。从任何一个 ISR 中断返回后，进入到较低优先级的 ISR。当有多个中断源时，每一个中断源(或者中断源组)的发生都可以从状态寄存器和/或 IPR 中的一个或者多个位中识别。情况 1：当存在来自多个设备的多个中断源时，处理器给每一个中断源(包括陷阱或者异常)或者中断源组分配一个预先假定的优先权(或者级或者类型)。假设编号 p_{hw} 代表中断源(或者组)硬件预先假定的优先级。假设编号为 0、1、2、……、k、……、m-1。$p_{hw}=0$ 的优先级最低；$p_{hw}=1$ 次之；……；$p_{hw}=m-1$ 的优先级最高。为什么硬件分配预先假定的优先级？在一组指令的执行过程中，多个中断会在同时发生，所有的或者只有几个中断被激活服务。使用相关 ISR 中断源的服务只能够按照一定的优先级顺序执行(只有一个处理器)。

处理器硬件可以为 7 个中断源分别分配 $p_{hw}=6$、5、4、3、2、1、0。以 80x86 系列处理器为例。考虑其 6 个中断源：除数为 0、单步、NMI(来自 RAM 奇偶错误等的不可屏蔽中断)、断点、溢出和打印屏幕。这些中断可以分别假设为 $p_{hw}=0$、–1、–2、–3、–4 和–5。处理器将最高的优先级分配给除数为 0。这是因为它是出现在用户软件中的异常条件。单步执行的优先级次之，将该中断源激活是因为当进行软件调试时，需要在每一条指令执行完后产生一个断点。NMI 的优先级再次之，这是因为需要对外部存储器读错误立即做出反映。打印屏幕的优先级最低。

ARM7 提供了两种类型的中断源：IRQ(interrupt request，中断请求)和 FIQ(fast interrupt request，快速中断请求)。

在这些挂起的中断中，哪一个中断应该被首先服务？轮询的方式可以解决这个问题。8086具有一个“向量式优先级轮询方法”。一种处理器中断机制可以内部提供向量的编号ISR_VECTADDR。向量式优先级方法意味着中断机制分配ISR_VECTADDR和p_{hw}。在每一个指令周期的最后(或者从一个ISR返回时)都要调用激活的和挂起中断源中优先级最高的中断源。80x86中的向量式优先级为n_{type}.n_{type}=0的优先级最高，n_{type}=0xff(=255)的优先级最低。

80196是另外一个中断控制的系统示例。每一个中断源都有不同的标志位。但是对中断源进行了分组。80196将所有可屏蔽中断分为8组。每一组都有一个公用的一级使能位。在ISR的服务期间，有8个处理器二级使能位。在一个ISR运行期间可以向另外一个优先级较高的ISR跳转。

4.6.5 硬件优先级的软件覆盖

可以通过几个示例理解这种情况。8051和80960内部中断机制如下。哪一个中断源或者中断源组的优先级较高？这首先是由在用户软件中已经被分配的较高优先级的ISR确定的。如果用户分配的优先级是相等的，则在最高优先级的ISR中，在处理器内部硬件已经预先分配的ISR优先级最高。例如，在8051中有中断寄存器，在IP寄存器中具有针对8051的5个中断源的5个优先级位(还有5个中断使能位。它们是处理器的ISR服务期间的二级使能位)。当IP的一个优先级位被置位后，相应的中断源获得一个较高的优先级；如果该位被复位，则该中断源获得较低的优先级。8051首先通过按照IP寄存器中的位，在高优先级中断的轮询进行选择。IP寄存器的概念不是必须的，因为已经存在了二级屏蔽位。在68HC11和80196中没有IP寄存器。80960的管理模式指令和非用户指令改善了p_{sw}，p_{sw}可以由5位进行定义。

注意：

当存在多个设备驱动程序和硬件、软件中断时，必须对每一个中断源或者中断源组的优先级进行分配。这个优先级可以是硬件定义的，也可以通过软件分配。利用这些方式分配的优先级覆盖了硬件优先级。系统中使用的操作系统为每一个例程或者任务分配了软件优先级。

本章小结

- 系统的软件设计者使用物理设备驱动程序、虚拟设备驱动程序和软件指令ISR、软件定义的条件和错误条件。
- 在任何一个系统中都必须使用中断服务例程(ISR)。
- 每一个系统都有一个中断服务机制。关于中断服务机制已经通过使用简单的实例进行了解释。
- 设备驱动程序例程为设备中断服务，并驱动一个外设单元(设备)。设备是通过使用其控制寄存器中的控制位进行配置和初始化的。发生中断时，设备驱动程序按照状态寄存器中的标志位执行。
- Linux具有大量的驱动程序。
- 通过示例学习了68HC11中的并口、串行UART和内部时钟设备的设备初始化和设备驱动程序编程。
- 在系统软件设计过程中使用虚拟设备，例如字符设备、块设备或者文件设备、RAM磁盘、套接字、管道、环回设备。按照与物理设备相似的方式对它们进行操作。

- 有一些设备中断以及其他中断源的驱动程序必须由软件设计者编写。文中还给出了各种可能的软件和硬件中断源列表。在系统中，软件条件、运行时相关的条件或者运行时错误相关的中断是很重要的。
- 系统的中断服务机制所提供了单个设备中断使能和禁止、中断向量和向量表、中断挂起寄存器和状态寄存器、不可屏蔽的和可屏蔽的中断。
- 每一个运行时程序在一个瞬间都有一个上下文。上下文的意思是 CPU 状态(程序计数器、堆栈寄存器、寄存器和程序状态(不能被另外一个例程修改的变量))。在调用另外一个 ISR、任务或者例程的时候必须保存上下文。在处理器切换到另外一个上下文之前，必须保存当前的上下文。某些处理器——例如 ARM 系列的处理器和 80960 处理器——提供快速上下文切换。它们具有用于上下文的局部寄存器的内部堆栈帧或者组。快速上下文切换缩短了中断延迟，能够满足每一个任务的最终期限。如果系统中使用的处理器不提供内部堆栈帧，操作系统程序提供用作堆栈帧的存储器块(分配存储器块)。
- 软件设计时要注意，要使中断延迟尽量短。这样就很容易满足每一个中断服务的最终期限。DMA 通道的使用能够帮助减小 IO 中断源的中断服务延迟周期。
- 可能会有多个中断源同时请求服务。在软件设计中，在考虑硬件优先级的基础上为多个中断源分配软件优先级，是最基本的要求。

关键词及其定义

- 设备初始化代码(device initialization code)：对设备控制寄存器进行编程的代码。
- 设备驱动程序代码(device driver code)：在设备地址上执行读和写操作的代码，并读取设备状态和初始化中断。
- 设备连接(加入)(device attaching/adding))：配置设备，并使能设备驱动程序。
- 设备断开(移除)(device detaching/removing))：由系统禁止设备驱动程序。
- 设备打开(device opening)：将设备控制位复位，准备驱动程序的使用。
- 设备关闭(device closing)：将设备控制位复位，下次要使用该设备只能够重新打开。
- Linux：一种源代码开放的操作系统。它具有大量的设备驱动程序和网络管理函数。
- Linux 设备驱动程序(Linux Device Driver)：从 Linux 源代码中获得的设备驱动程序。
- 中断(interrupt)：在产生中断信号后，CPU 会通过调用中断服务例程(ISR)初始化进一步的行为，或者 CPU 会继续执行当前的过程(任务)。
- 中断服务例程(Interrupt Service Routine, ISR)：中断产生后，首先将必要的参数保存到堆栈中，使得从例程的最后一条指令返回时能够获取同样的参数，然后执行该程序。
- 硬件定时器(hardware timer)：在系统中以硬件形式存在的定时器，从处理器的内部时钟程序系统时钟获得输入。设备驱动程序对其进行编程，像对其他物理设备编程一样。
- 中断机制(interrupt mechanism)：设备和端口中断驱动服务的机制。这种机制节约了处理器的等待时间，因为这样能够使得处理器处理多个设备和虚拟设备。中断机制还设置各个中断的优先级，并进行设备服务的激活和禁止。
- 前台程序(foreground program)：当没有对中断调用进行服务时运行的程序称为前台程序。
- 中断使能位(interrupt enable Bit)：如果设置为真，则使能中断源的中断。

- 中断屏蔽位(interrupt mask Bit)：当将该位复位(=false)，则对中断服务初始化的请求做出响应，否则不响应。
- 中断标志(interrupt flag)：一个布尔变量寄存器位，将该位置位表示需要执行一个中断服务例程(ISR)。当相应的 ISR 开始执行后，该位复位。
- 不可屏蔽中断(Non-Maskable Interrupt)：不能被禁止的中断源，用于必须进行服务的情况。
- 可屏蔽中断(maskable interrupt)：可以被禁止和屏蔽的中断源。
- 一级使能位(primary level enable bit)：使能或者禁止所有可屏蔽中断源的任何服务的位。在执行临界区代码时帮助禁止任何可屏蔽中断源的服务。
- 二级屏蔽位(secondary level mask bit)：它禁止单个中断源或者中断源组的中断服务。
- 中断向量(interrupt vector)：提供相应 ISR 地址的存储器地址。系统中具有由硬件为每一个内部中断设备的中断源分配的特殊向量地址。
- 中断向量表(interrupt vector table)：存储器中的中断向量表。该表能够帮助每一个内部设备的多中断源的服务。表中的每一行都是针对一个中断向量地址的，其中保存了提供相应中断服务例程地址的字节。
- 中断挂起寄存器(interrupt pending register)：展示来自各种正在挂起服务(也就是运行相应的 ISR)的设备中断源或者中断源组。它是可读可写的。当一个中断服务开始后，相应的位自动复位。用户指令也可以将寄存器中的位复位。
- 状态寄存器(status register)：是一个设备的只读寄存器，当产生一个中断时，将一个标志位置位。用户指令也可以将其中的位复位。如果一个设备具有几个中断源，状态寄存器就有几个标志位，每一个中断源对应一个位。当被处理器指令读取之后，标志位复位。
- 上下文(Context)：程序计数器、堆栈指针以及程序状态字和用于前台程序或者 ISR 或者任务的处理器寄存器。还可以包括分配给程序或者例程的存储器块地址。可以包括 ID。
- 上下文切换(context switching)：保存被中断例程(或者函数)的上下文，恢复或者加载被调用例程的新的上下文。上下文切换所需要的时间包括在中断延迟周期中。
- 堆栈帧(stack frame)：保存着一个程序或者 ISR 的上下文的一组寄存器或者一个存储器块。
- 中断延迟(interrupt latency)：服务请求发出后(中断源状态标志位置位)等待服务的周期。
- 最差情况延迟(worse-case latency)：最差情况下的中断延迟周期最长。
- 最终期限(deadline)：对一个中断的服务必须开始的周期。
- 软件中断(software interrupt)：由指令、软件定时器、错误条件陷阱或者非法操作码产生的中断。
- 异常(exception)：可能由程序员定义的一个条件的发生，程序员还定义 ISR 用于为这个条件服务。错误条件是由异常处理的。
- 信号(signal)：异常有时候也称为信号。处理器也可以用信号产生一个异常。例如，80x86 中当发生除数为 0 的情况后。
- 硬件中断(hardware interrupt)：系统的设备或者端口中断。
- 硬件分配的优先级(hardware assigned priority)：由处理器本身分配的优先级，用于当多个中断需要中断服务时，对一个中断源进行服务。

- 软件分配的优先级(software assigned priority)：中断源或者中断源组的优先级。是在称为中断优先级寄存器的寄存器中定义的。当同时发生多个中断时，软件分配的优先级覆盖硬件分配的优先级。
- 轮询(polling)：一种中断服务方式，在一条指令执行完或者 ISR 结束时，由处理器搜索挂起的中断，并为最高优先级的中断进行服务。

问题回顾

(1) I/O 设备的忙等待传输方式的优缺点是什么？

(2) 中断驱动的数据传输的优缺点是什么？

(3) 基于 DMA 或者外设事务服务器(80960)的数据传输与中断驱动的数据传输相比的优点有哪些？

(4) 如何使用中断源的向量地址？

(5) 在微控制器中，对于给定的内部外设，中断向量地址在中断机制中是预先设定的。如何将向量地址分配给异常和用户定义的中断？

(6) 每一个系列处理器的中断机制都是不同的。试解释设备驱动程序是处理器敏感的程序。

(7) 如何初始化和配置一个设备？以 PC COM 端口的串行驱动程序为例。

(8) 存储器中一个文件的行为如何作为一个设备来处理？

(9) RAM 磁盘的优点是什么？

(10) 为套接字、套接字缓冲区处理、防火墙、网络协议(例如 NFS、IP、IPv6 和以太网)和网桥做一个 Linux 内部网目录函数列表。为什么这些设备驱动程序分配在 Linux 操作系统的不同网络管理函数目录中？

(11) 给出上下文、中断延迟和中断服务最终期限的定义。

(12) 为什么嵌入式处理器中的上下文切换比使用堆栈指针将指针和变量保存在堆栈中要快？以 80960 为例说明。

(13) 在 ARM7 中，上下文切换是如何处理的？

(14) DMA 通过提供对 IO 的直接访问而减小了处理器的负担。在多任务系统中，DMA 如何通过缩短中断服务延迟来加快任务的执行？

(15) 抛出异常的意义是什么？函数(例程)执行过程中的异常条件是如何处理的？

(16) 参见 G.7、G.8 和 G.9 节。80x86 微处理器和 68HC11 微控制器中的设备驱动程序和 ISR 的区别是什么？

(17) 如何为系统中的多个设备驱动程序分配服务优先级？如何为定时器设备和 ADC 设备分配优先级？

(18) 在一个中断处理机制中，硬件分配的优先级的用处是什么？

(19) 在一个中断处理机制中，软件分配的优先级的用处是什么？

(20) 断点中断对于调试嵌入式软件的意义是什么？

(21) POSIX 函数的意义是什么？

实践练习

(22) 如何编写设备驱动程序？列出编写设备驱动程序的步骤。

(23) 通过互联网查找资料，设计一个表格，展示嵌入式 Linux OS 和 Red Hat eCOS(嵌入式可配置操作系统)中驱动程序模块的特征。

(24) 通过示例解释字符设备、块设备和可配置为字符设备的块设备。UART 是一个字符设备。为什么这样说？

(25) 给出软件相关的中断示例。在 8086 中，哪些中断在发生软件错误时产生？

第 5 章　编程概念及 C 与 C++的嵌入式编程

本章前所学内容

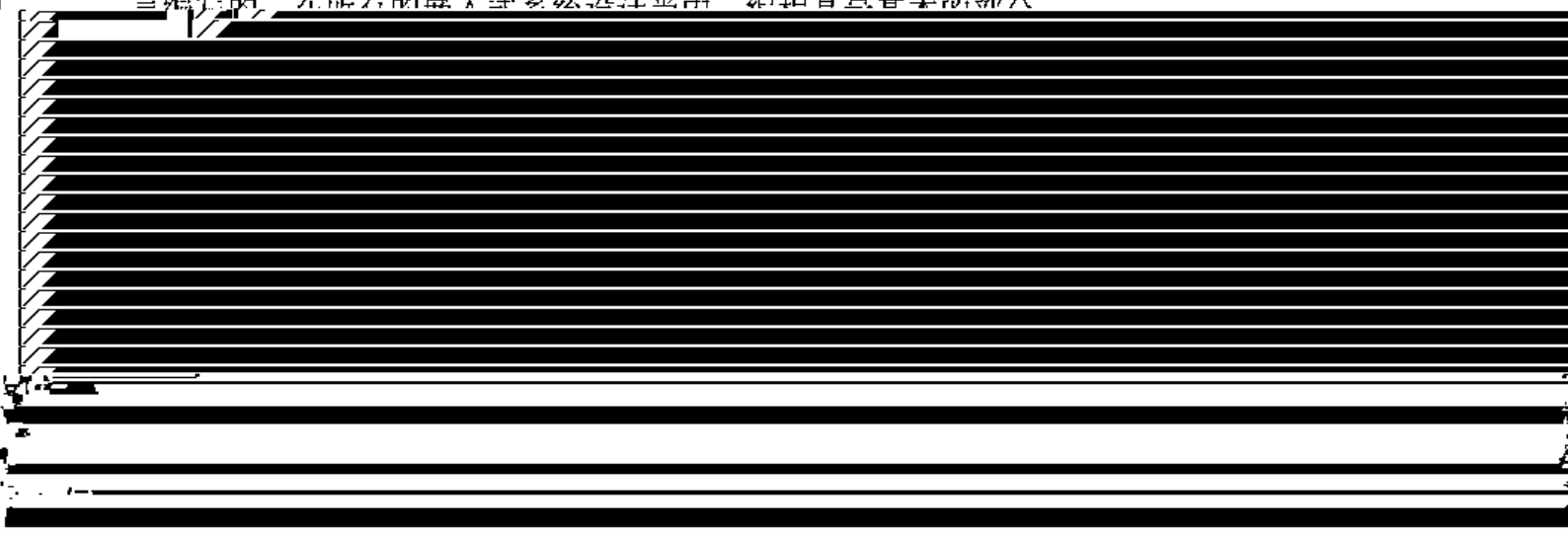
我们将前面章节中已经介绍的知识概括为如下几点：

(1) 系统硬件由处理器、存储设备(ROM 和 RAM)与内部构件、I/O 端口、物理内部设备与外部设备、定时设备与基本的硬件单元——电源、时钟电路与复位电路组成。

(2) 系统中有用于连接物理设备端口的总线和接口电路。

(3) 系统中为设备驱动程序配备了处理器和存储敏感软件，它们作为中断服务机制，用于处理来自设备、虚拟设备驱动的中断以及软件中断、异常和错误。

除了上述的各种软件以外，在系统中硬件的简单或者复杂应用也需要软件。除了一些特定的处理器和存储器敏感指令的程序代码可能用汇编语言编写以外，其他的代码都是采用高级语言编写的。在所有的嵌入式系统设计当中，编程是最基本的部分。



本章的目的是对以下几点进行解释：(i)详细的编程概念、程序元素和数据结构。(ii)面向对象编程方法的使用。(iii)源代码管理工具的使用，以及(iv)存储器优化方法的使用。最终通过对下面所列出的概念和实践进行解释来达到目的。

(1) 用汇编语言进行编程与用高级语言进行编程之间的比较，以及用 C 语言对嵌入式系统编程的特征。

(2) 程序元素：预处理器指令与头文件，为某个应用提供程序的 include 文件与源文件。

(3) 程序元素：宏和函数，以及它们在 C 程序中的使用。

(4) 程序元素：数据类型、数据结构、修饰符、条件语句和循环。

(5) 程序元素：指针、函数调用、多种函数、函数指针、函数队列以及服务例程队列。

(6) 重要的数据结构：数组、队列、堆栈、链表以及树结构。

(7) 队列在网络通信或者客户机-服务器通信中所扮演的重要角色。

(8) 使用“队列”的编程细节，队列是在程序当中使用的一种重要的数据结构。

(15) 操作系统函数进行多任务处理的就绪任务列表。

(16) 面向对象的编程概念、C/C++语言的嵌入式编程以及用 C++和 Java 编程的优点与缺点。

(17) 编译器、交叉编译器，以及源代码组织工具的使用。

(18) 满足嵌入式软件需求而优化存储器所需要的步骤。

设计嵌入式系统程序所采用的程序建模的概念和软件工程实践，将在第 6 章和第 7 章中进行阐述。进程的定义与进程的同步、RTOS(实时系统)与采用 RTOS 进行 RTOS 编程将在第 8~11 章中进行讲述。

5.1 用汇编语言和高级语言 C 进行软件编程

采用汇编语言来为应用程序进行编码具有如下优点：

(1) 能够对处理器的内部设备进行精确地控制，并且能够通过其指令和寻址方式来完全利用处理器的特定特征。

(2) 机器代码是压缩的。这是因为不存在用于声明条件、规则和数据类型的代码。因此系统需要的存储器可以更小。额外的存储器需求并不依赖于程序员对数据类型的选择以及对规则的声明。它也不需要特定的编译器和特定的库函数。

(3) 设备驱动程序的代码只需要几条汇编指令就可以了。例如，考虑一个小的嵌入式系统，微波炉、全自动洗衣机或者巧克力自动售卖机中的定时设备。它们的汇编代码都是压缩的、精确的，可以方便地进行编写。

对复杂的系统用 C、C++或者 Java 来开发源文件会比较方便，因为对于这样一些系统，高级语言具有如下所示的优点：

(1) 使得复杂系统的开发周期短，这是由于函数(过程)、标准库函数、模块编程方法以及自上而下设计方法的采用。应用程序需要进行构建以确保软件能够基于所提倡的软件工程原则进行开发。

a. 让我们来回顾一下示例 4-8 中的 UART 串行线设备驱动。由于该设备在很多系统当中都要使用，所以直接使用该函数会导致重复编码的冗余。只要在需要的时候改变某些(为变量)传递的参数，并将这些参数用于设备的另一个实例当中即可。

b. 当需要用到另一个数值(参数)的平方根时，求平方根的代码段应该重写吗？标准库函数 square root ( )为程序员节省了编写代码的时间。类似的函数还有 delay( )、wait( ) 以及 sleep( )。

c. 模块化编程方法中的结构块是可重用的软件构件。我们来考虑根据 IC(集成电路)进行类推。IC 是将几个电路集成到一个电路中，类似地，一个结构也可以调用多个函数和库函数。但是应该对模块进行认真的测试。模块必须有一个正确的目标和适当的数据输入和输出。它应该只有一个调用过程，并且存在一个返回点。除了目标数据之外，它不应该影响任何其他的数据(数据封装)。还必须返回(报告)执行过程中所遇到的发生错误的条件。

d. 自底向上的设计(Bottom up design)是这样一种设计方法，编程首先是从特定的子模块和各个行为集合开始。对于特定行为集合的模块，软件定时器 RTCSWT::run 就是一个很好的示例。用于延时、计数、寻找时间间隔以及其他很多应用的程序都可以先编写，然后再设计最终的程序。采用这种方式设计程序的方法是首先为基本的函数模块编写代码，然后使用这些代码来搭建一个更大的模块。

e. 自顶向下的设计(Top-Down design)是另一种编程方法，即首先设计主程序，然后再设计其中的模块与子模块，最后才是函数。

(2) 数据类型声明使得编程更加轻松。例如，存在四种类型的整数：整数类型 int、无符号整数类型 unsigned int、短整数类型 short 以及长整数类型 long。当只单纯处理数值的时候，我们就将变量声明为无符号整数类型。例如，numTick(时钟在定时结束以前的嘀哒次数)就只能是无符号整数类型 unsigned int。在进行算术运算的时候，我们就需要有符号整数类型 int(32 位)。整数也可以被声明为短整数类型 short(16 位)和长整数类型 long(64 位)。要操作字符文本和字符串所需要的另外一种数据类型是字符类型 char。每一种数据类型都是所要使用的、所要操作的、所要表示的方法以及允许进行的操作的抽象。

(3) 类型检查使得程序更不容易出错。例如，有四种类型检查不允许对字符数据类型进行减法、乘法以及除法。而且，它把+用作连接符。例如，micro+controller 就应该得到 microcontroller，在这里 micro 是一个字符数组，而 controller 是另一个字符数组。

(4) 控制结构(例如，while、do-while、break 以及 for)与条件语句(例如，if、if-else、else-if 以及 switch-case)能够使得程序流路径设计任务变得简单。

(5) 非处理器特定代码具有可移植性。因此，当硬件发生改变的时候，只有设备驱动与设备管理模块、初始化与定位模块以及初始启动记录数据需要修改。

作为高级语言，C 语言还具有如下所示的优点：

(1) 它是处于低级(汇编)语言与高级语言之间的一种语言。在其中插入汇编语言代码称为内联汇编(in-line assembly)。因此，通过内联汇编直接对硬件进行控制也是可行的。程序的复杂部分可以用高级语言编写。示例 4-5 给出了在 C 中使用内联汇编代码来实现端口 A 的驱动程序。

注意：

用高级语言编程可以缩短程序的开发周期，使用模块化编程方法，让我们能够遵循所提倡的软件工程思想。它采用“自底向上”与“自顶向下”的方法方便了程序的开发。嵌入式系统程序员之所以长期都偏好于使用 C 有以下几个原因：(i)嵌入式汇编代码可以采用内联汇编的特征。(ii)在 C 的编译器中有嵌入式系统直接可用的模块，以及可以直接与系统程序员代码接口的库代码。

5.2 C 程序中的元素：头文件、源文件以及预处理指令

C 程序中的元素，头文件、源文件以及预处理指令如下所示：

5.2.1 用于包含文件的 include 指令

任何 C 程序首先都要包含那些准备使用的头文件和源文件。谁都不会在想要牛奶的时候才在家里养一头牛！示例 11-2 给出了一个通过网络驱动卡采用 TCP/IP 协议发送字节流的例子。它的程序是以示例 4-5 和示例 11-2 的代码作为开头的。每一个包含文件的目的都在用*号括起来的注释部分当中给出。

示例 5-1

```
# include "vxWorks.h" /* Include VxWorks functions*/
# include "semLib.h" /* Include Semaphore functions Library */
# include "taskLib.h" /* Include multitasking functions Library */
# include "msgQLib.h" /* Include Message Queue functions Library */
# include "ioLib.h" /* Include File-Device Input-Output functions Library */
# include "sysLib.c" /* Include system library for system functions */
# include "netDrvConfig.txt" /* Include a text file that provides the 'Network
Driver Configuration'. It provides the frame format protocol (SLIP or PPP or
Ethernet) description, card description/make, address at the system, IP address
(s) of the node (s) that drive the card for transmitting or receiving from
the network. */
# include "prctlHandlers.c" /* Include file for the codes for handling and actions
as per the protocols used for driving streams to the network. */
```

Include 是一个用于包含某个文件内容(代码或者数据)的预处理指令。下面给出了可以被包含的文件。所有的文件和指定的头文件都必须在需要的时候再包含进来。

(i) 包含代码文件：这些文件是已经存在的代码文件。例如，# include "prctlHandlers. c"。

(ii) 包含常量数据文件：这些文件是代码文件，可以有扩展名.const。

(iii) 包含字符串数据文件：这些文件是包含字符串的文件，可以带扩展名.string、.str 或者.txt。例如，示例 5-1 当中的# include "netDrvConfig.txt"。

(iv) 包含初始数据文件：回顾一下第 2.4.3 节以及示例 2-13~2-15。这些文件用于嵌入式系统掩膜只读存储器的初始或默认数据。启动程序会被复制到 RAM 当中，可以具有扩展名".init"。另外，RAM 的数据文件扩展名为".data"。

(v) 包含基本变量文件：这些文件是存储在 RAM 中的全局或者局部静态变量文件，因为它们不具有初始(默认的)值。所谓静态的意思是变量只有一个普通的变量地址实例，并且有一个静态存储空间分配。由于只存在一个实时时钟，所以该变量的地址只有一个实例(请参考第 5.4.3 节中的案例(iv))。这些基本变量都被存储在以.bss 为扩展名的文件当中。

(vi) 包含头文件：这是一个预处理指令，目的是要包含一组源文件的内容(代码或者数据)。它们都是某个特定模块的文件。头文件的扩展名为.h。例如，在程序当中要使用字符串就需要字符串操作函数。只要在程序当中包含了一个名为 string.h 的头文件就可以了。如果在程序当中要用到算术表达式就需要使用算术函数：square root(求平方根)、sin(正弦函数)、cos(余弦函数)、tan(正切函数)、atan(余切函数)等等。只要包含了名为“math.h”的头文件就可以达到这一目的。预处理指令就应该为# include <string.h>与# include<math.h>。

还可以为汇编代码、I/O 操作(conio.h)、操作系统函数以及实时操作系统函数包含头文件。示例 5-1 中的# include"vxWorks.h"是给编译器的指示，让它包含 VxWorks RTOS 函数。

注意：

某些编译器提供 conio.h 来取代 stdio.h。这是因为嵌入式系统通常都不需要用于打开、关闭以及读写文件的函数。所以如果包含了头文件 stdio.h 就会使得代码量过大。

包含头文件、文本文件、数据文件或者常量文件之间有什么差别呢？考虑一下包含文件 netDrvConfig.txt 与 math.h。(i)头文件是通过了良好测试和调试的模块。(ii)头文件提供了访问标准库的接口。(iii)头文件可以包含多个文本文件或者 C 文件。(iv)文本文件是对包含特定信息的文本的描述。

5.2.2 源文件

源文件是为实现应用软件功能而编写的程序文件。源文件需要编译。源文件当中也有应用程序的预处理指令，并且还有开始处理的第一个函数。该函数称为主函数。它的代码从 void main( )开始。由主函数来调用其他函数。源文件就像前面的示例 4-3~4-5 当中的文件那样保存代码。

5.2.3 配置文件

配置文件是用于描述系统配置的文件。让我们来回顾一下示例 4-8 当中的第 3 行代码。设备配置代码可以放在一个保存基本变量的文件当中，然后在需要的时候再将该文件包含进来。如果这些代码在文件 serialLine_cfg.h 中，那么预处理指令就应该为# include "serialLine_cfg.h"。考虑另一个例子# include"os_cfg.h"，它将会包含 os_cfg 头文件。

5.2.4 预处理指令

预处理指令以一个井号(#)开头。这些命令是为了给编译器传达下面的指示。

(1) 预处理全局变量：# define volatile Boolean IntrEnable 是示例 4-6 中的一条预处理指令，它的意思是在处理之前先考虑一个布尔数据类型的全局变量 IntrEnable，并且该变量是不稳定的("不稳定"是告诉编译器，在对代码进行压缩和优化的时候，不要将该变量考虑在内)。IntrDisable、IntrPortAEnable、IntrPortADisable、STAF 以及 STAI 是示例 4-5 当中其他的全局变量。

(2) 预处理常量：#define false 0 是示例 4-3 当中的一个预处理指令。它表示在处理之前将 false 假设为 0 的一条指示。指令 define 表示在程序当中分配指针数值。考虑一下#define PortA(volatile unsigned char*)0x1000 以及#define PIOC (volatile unsigned char*)0x1001(请参阅第 4.2 节)。0x1000 与 0x1001 分别为 PortA 与 PIOC 的固定地址。它们都是为这些 68HC11 寄存器定义的常量。字符串是常量，例如那些在移动系统的屏幕上用于初始展示的字符串。例如，# define welcome "Welcome To ABC Telecom"。

注意：

预处理常量、变量以及包含配置文件、文本文件、头文件以及库函数都在 C 程序中使用到了。

5.3 程序元素：宏与函数

表 5-1 列出了这些元素，并给出了它们的用法。

表 5-1 程序元素各种指令集的用法

程序元素	用法	在开始之前将上下文存储到堆栈中，并且在返回时重新取回	嵌套的可行性
宏	执行一小段命名的代码集合	No	不可嵌套
函数	调用程序通过它的参数传递数值来执行一组命名代码。当数据对象没有被声明为 void 时，可以将它返回。它具有上下文保存和获取开销	Yes	可以嵌套，可以调用另一个函数，也可以被中断
主函数	进行函数、数据类型、类型定义的声明以及(i)执行一组命名代码，调用一组函数，以及发生中断时调用 ISR，或者(ii)启动操作系统内核	No	不可嵌套
再入函数	请参考第 5.4.6 节	Yes	只可以调用另一个再入函数
中断服务例程或者设备驱动	声明函数、数据类型、类型定义并执行一段命名代码。必须要简短，以便其他中断源也能在最终期限内服务。必须是一个再入例程，或者必须有解决共享数据问题的方法	Yes	可以调用更高优先级的中断
任务	请参考第 8.2 节。必须是一个再入例程，或者必须能解决共享数据问题	Yes	不可嵌套
递归函数	一个调用自身的函数。它必须也是一个再入例程。大多数情况下，由于存储器的约束，总是避免使用递归函数。(在每一次递归调用以后堆栈都会有所增长，并且会阻碍存储器空间的获取)	Yes	可以嵌套

预处理宏：宏是在程序当中定义的命名代码段。它与函数有所不同，因为一旦宏以某个名字进行了定义，编译器就将相关的代码放到宏的名称出现的每一个地方。enable_Maskable_Intr()与 disable_Maskable_Intr()都是示例 4-5 当中出现的宏(其中的括弧是可选的。写出这对括弧可以增强可读性，因为这样可以将宏与常量相区分)。只要名称 enable_Maskable_Intr 出现了，编译器就会将它涉及到的代码段放置到名称出现的地方。宏，所谓的测试宏或者测试向量，也都是为调试系统而设计和使用的(请参考第 7.6.3 节)。

宏与函数究竟有什么不同呢？函数的代码只需要编译一次。当调用该函数的时候，处理器会保存上下文，当调用返回的时候就会恢复上下文。而且，函数可以不返回任何结果(声明为 void 类型的情况)，或者返回一个布尔值、一个整数、任意一个简单类型或者引用类型的数据(简

单类型类似于一个整数或者字符。引用类型类似于一个数组或者结构)。enable_PortA_Intr( )与disable_PortA_Intr( )是在示例 4-5 当中调用的函数(这里的括号不是可选的)。宏只能是简短的代码。这是因为如果用某个函数调用来取代宏，那么(函数调用和返回时要进行的上下文保存以及其他一些行为)需要时间开销$T_{overheads}$，它与在函数中执行简短的代码所需要花费的时间T_{exec}是处于同一数量级的。当$T_{overheads} \ll T_{exec}$时，我们使用函数，当$T_{overheads} \sim= T_{exec}$或者$T_{overheads} > T_{exec}$时，我们使用宏。

注意：

宏与函数都在 C 程序当中使用到了。当代码只需要编译一次的时候，我们使用函数。但是，一旦调用了函数，处理器就必须保存上下文，并且在返回时必须恢复上下文。而且，函数可以不返回结果(声明为 void 类型的情况下)或者返回一个布尔类型的数值、整数、原语或者引用类型的数据。当需要将短小的功能代码段插入到很多地方或者函数当中时就应该使用宏。

5.4 程序元素：数据类型、数据结构、修饰符、语句、循环和指针

5.4.1 数据类型

数据命名后，就会在存储器中分配地址。很多地址分配都取决于数据类型。C 允许使用下面的简单数据类型。用于表示字符的 char 类型(8 位)、byte 类型(8 位)、无符号短整型 unsigned short(16 位)、短整数类型 short(16 位)、无符号整数类型 unsigned int(32 位)、整数类型 int(32 位)、长双精度类型 long double(64 位)、浮点类型 float(32 位)、双精度类型 double(64 位)(有些编译器不将“byte”作为一种数据类型进行定义。就像示例 4-6 当中的第 2 行一样，typedef 用于在 C 程序当中创建一个布尔类型的变量)。

一般都会使用适合于硬件的数据类型。例如，一个 16 位的定时器可以只有无符号短整型数据类型，它的范围可以只从 0~65535。也可以使用 typedef。看了下面所示的例子就可以明白了。编译器版本可以不处理无符号字节类型的声明。那么可以将“无符号字符类型(unsigned character)当成一种数据类型使用。可以对其进行如下声明：

```
typedef unsigned character portAdata
#define Pbyte portAdata Pbyte =0xF1
```

5.4.2 使用数据结构：队列、堆栈、链表和树

学生在一学期内所学的不同课程的分数保存在一个适当的表中。成绩表必须以一种有组织的方式来显示分数。当存在大量数据的时候，必须对它们正确地组织。数据结构是用于组织大量数据的一种方法；在几个指针和/或者索引以及函数的帮助下，数据元素可以被识别和访问(读者可以参考用 C 与 C++语言实现的数据结构算法的教材。例如，Brooks/Cole Thomson Learning 于 2001 年出版的由 Adam Drozdek 编著的 *Data Structures and Algorithms in C++*)。

数据结构是所有程序的重要组成元素。第 2.5 节定义和描述了几种重要的数据结构，堆栈、一维数组、队列、环形队列、管道、表(二维数组)、查找表、散列表以及链表。图 2-3~2-5 给出了不同的数据结构，并介绍了如何以一种有组织的方式将它们保存到存储器块当中。任何数据结构都是不可恢复的。

第 5.5、5.6、5.7 节分别对三种数据结构进行了详细的阐述：队列、堆栈以及链表。表 5-2 给出了使用方法，还给出了使用队列、堆栈、链表以及树结构的示例。

表 5-2　各种数据结构在程序中的使用

数 据 结 构	定义和适用范围	示　例
队列	它是由一系列元素组成的一种数据结构，其中第一个元素等待操作。操作只能以先进先出(FIFO)的方式进行。它应该在某个元素不会直接被索引或者指针访问，而只能按照先进先出的顺序访问时使用。元素只能被插入到一系列等待被操作的元素末尾。存在两个指针，一个用于在操作完成以后进行删除，另一个用于插入。这两个指针都会在操作完成以后加 1	(1)打印缓冲。每个字符都要以先进先出的方式打印 (2)网络上的帧(每一帧也都有一个字节流队列)每个字节都必须按照先进先出的方式发送 (3)在某个序列中的图像帧。(它们必须按照先进先出的方式处理)
堆栈	它是由一系列元素组成的一种数据结构。它的最后一个元素等待处理。任何操作都只能以后进先出(LIFO)的方式处理。它应该在元素不会直接被索引或者指针访问，而只能按照后进先出的方式处理时使用。元素只能被压入(插入)到一系列等待处理的元素顶上。操作完成以后，不仅是弹出(删除)操作，而且压入(插入)操作都只使用一个指针。在每一次操作之后指针都加 1 或者减 1。这取决于是插入还是删除	(1)发生中断或者调用函数时变量的压入。(2)从堆栈当中得到一个已经压入的数据
数组(一维向量)	它是由一系列元素组成的数据结构，其中每个元素都只能用一个标识名或者索引访问。它的元素在使用和操作时都非常容易。它应该在数据结构的每个元素为了方便操作而都采用索引以不同的标识来访问时使用。索引从 0 开始，并且是递增整数	$ts = 12 * s(1)$：年薪 ts 是第一个月薪水的 12 倍。 Marks_weight[4]=marks_weight[0]：索引为 4 的课程成绩的权重与索引为 0 的课程成绩的权重相同
多维数组	它是由一系列元素组成的一种数据结构，其中每一个元素都由另一子系列的元素组成。每个元素都以标识名和两个或者多个索引来访问。它应该在数据结构的每一个元素为了方便操作而都采用索引以不同的标识来访问时使用。数组的维数等于需要用来识别数组元素的索引数目。索引从 0 开始，并且是递增整数	处理矩阵或者张量。考虑某个图像帧当中的一个像素。再考虑大小为 144x176 的一个图像帧中的四分之一 CIF 格式图像像素。(回顾一下第 1.2.7 节)像素[108,88]表示垂直列的第 108 个像素点，水平行的第 88 个像素点。[1]请参考下面的注释

(续表)

数据结构	定义和适用范围	示　例
链表	每个元素都有一个指针指向它的下一个元素。只有第一个元素是可识别的，并且由顶头的指针指示(头指针)。其他的元素都是不可识别的，因此不能够直接访问。一个元素可以通过第一个元素，然后再顺序通过所有后继的元素来读取，读取并删除，增加一个相邻的元素或者用另一个元素取代	每一个任务都有指针指向下一个任务。另一个例子就是一个指向子菜单的菜单
树	有一个根元素。有两个或者多个分支，每一个分支都有一个子元素。每个子元素都有两个或者多个子元素。最后一个没有子元素。只有根元素是可标识的，它由树顶的指针标识(头指针)。其他的元素都是不可标识的，因此都不能够直接访问。通过根元素，然后继续通过所有的子元素，元素就可以读取或者读取并删除、添加另一个子元素，或者被另一个子元素取代。树将数据元素作为分支。最后一个子元素，也被称为结点是没有后继子元素的。二叉树就是每个元素最多只有两个子元素(分支)的树	它的一个例子就是目录。有很多文件夹，每一个文件夹都有很多其他的文件夹，而最后都是文件

1. 像素[0,0]表示位于左上角的像素点。像素[144，176]表示位于右下角的像素点。像素[10,108,88]是三维数组表格中的一个像素数据元素。它表示第 10 帧的同一位置(108 x 88)处的像素点。

5.4.3 修饰符

修饰符的行为如下所示：

(i) 在函数块之外，如果某个变量在程序中被初始化，加了修饰符 auto 或者不加修饰符表示使用定位器为该变量分配 ROM。如果没有进行初始化，表示用定位器给它分配 RAM。

(ii) 如果在函数块内，在程序中对某个变量进行了初始化，修饰符 auto 或者没有加修饰符表示定位器为该变量分配 ROM。定位器不进行 RAM 的分配。

(iii) 修饰符 unsigned 用于短整型、整型或者长整型，分别指示只允许使用 16 位、32 位或者 64 位的整数。

(iv) 修饰符 static 在函数块内声明。静态声明指示编译器，变量应该在函数块外也能够访问到，并且应该为它准备一个保留的存储空间。这样，当要切换到另一个任务时，不必将它保存到一个堆栈中。当同时有几个任务在协同执行时，使用声明 static 非常有用。考虑一个声明的例子，private: static void interrupt ISR_RTI ( );。这里的 static 声明指示编译器 ISR_RTI ( )的函数代码仅限于为函数 ISR_RTI ( )分配的存储块当中。而 private 声明则表示，在其他的所有对象当中不存在该方法的其他实例。这样就不将它保存到堆栈当中。如果在程序当中进行了初始化，定位器为其分配 ROM。如果没有在程序当中进行初始化，定位器为其分配 RAM。

(v) 修饰符 static 在函数块之外声明。那么在被声明的类之外或者在被声明的模块之外，是不可用的。通过定位器为函数代码分配 ROM。

(vi) 修饰符 const 在函数块之外声明。它必须被程序初始化。例如，#define const Welcome_Message "There is a mail for you"。通过定位器分配 ROM。

(vii) 修饰符 register 在函数块之内声明。它必须被程序初始化。例如，register CX。CPU 寄存器是在需要的时候临时分配的，没有分配 ROM 和 RAM。

(viii) 修饰符 interrupt 指示编译器，一旦进入函数代码，就保存所有的处理器寄存器，并且在从函数返回的时候恢复它们(该修饰符以下划线作为前缀，例如某个编译器中的_interrupt)。

(ix) 修饰符 extern。它指示编译器在非当前使用的模块中寻找数据类型声明或者函数。

(x) 函数块之外的修饰符 volatile 用于警告编译器某个事件会改变它的数值，或者它的改变表示发生了一个事件，如中断事件、硬件事件或者内部任务通信事件。例如，示例 4-6 中的这样一个声明：volatile Boolean IntrEnable;。如果初始值为真，那么通过一个服务例程，在服务开始的时候，它的值会变为假。编译器不会对 volatile 变量进行优化。给变量赋值 c=0。后来它又被赋值为 c=1。在进行代码优化的时候，编译器会忽略语句 c=0，而执行 c=1。但是，如果 c 是一个事件变量，那么它不应该被优化。如果在 ISR 的执行期间使用了中断使能变量来禁止所有的中断，那么 IntrEnable=0 会在服务例程的开头。IntrEnable=1 在从 ISR 返回之前执行。这就能在系统中重新激活中断了。将 IntrEnable 声明为 volatile 指示编译器不要在同样的函数中优化两个赋值语句。这里，定位器没有分配 ROM 和 RAM。

(xi) 修饰符 volatile static 在函数块之内声明。例如：(a)volatile static boolean RTIEnable = true; (b)volatile static boolean RTISWTEnable; (c)volatile static boolean RTCSWT_F;。静态声明是用于指示编译器，变量应该在函数块之外也能够访问，并且应该为其保留一个存储空间；volatile 指示在事件可以修改的时候不要进行优化。当上下文切换到另一个任务时，它不会将上下文压到堆栈当中。当有几个任务在协同执行的时候，声明为 static 非常有用。由于进行了 volatile 声明，所以编译器是不会进行优化的。这里定位器没有分配 ROM 和 RAM。

5.4.4 条件语句、循环语句以及无限循环语句

条件语句在程序中会多次使用。如果某个定义的条件能够被满足，那么会执行紧跟在该条件语句之后的大括号内的语句(或者不带大括号的语句)，否则程序会转到下一条语句或者转到另一组语句中执行。有时一组语句会在某个循环当中重复执行。以数组为例，索引发生改变，同一组语句会为数组当中的其他元素重复执行。

在编程过程中，无限循环都是不希望发生的。为什么呢？程序永远都不会结束，永远都不会退出，并在循环之后继续执行代码。无限循环是嵌入式系统编程的一个特征。看下面的例子就明白了。(i)切断电话如何？电话中的系统软件必须总是保持在寻找铃声的等待循环当中。从循环当中退出会使得系统硬件冗余。(ii)回顾一下示例 4-1 中为 In_A_out_B 编写的互联程序。端口 A 可以在任意时刻给出输入。系统必须等到端口 B 输出的时候才能够执行代码，然后回来接收并等待下一次输入。无限循环的硬件等价于一个系统时钟(实时时钟)，或者一个正在运行的空闲计数器。

示例 5-2 给出了一个 C 程序设计，其中，程序从 main( )函数开始执行。其间有对函数进行的调用以及对中断的处理。它必须回到起始处。系统主程序永远都不能出现停止状态。因此，main( )是一个从开始到结尾反复执行的无限循环。

示例 5-2

```
# define false 0
# define true 1
/*************************************************************************/
void main (void) {
/* The Declarations here and initialization here */
.
.
/* Infinite while loop follows. Since the condition set for the while loop is
always true, the statements within the curly braces continue to execute */
while (true) {
/* Codes that repeatedly execute */
.
.
}
/*************************************************************************/
```

假如函数 main 中没有等待循环，而只是简单地将控制传递给实时操作系统(RTOS)。考虑一个多任务的程序。操作系统可以创建一个任务。操作系统可以将一个任务插入到链表当中，也可以从链表中删除一个任务。让操作系统内核优先调度各种链表任务的运行。这样每个任务都会有代码在无限循环当中。(请参考第 9 章与第 10 章理解这里提到的各种术语)。示例 5-3 展示了任务当中的无限循环。

多个无限循环如何共存呢？其中的代码等待一个信号、事件或者一组传递给操作系统内核的消息。它被操作系统内核检测到以后会传递另一个消息，并为该任务生成另一个信号，取代前面正在运行的任务。假设事件正在设置一个标志，只要将该标志传递给一个正在等待的任务，那么该标志就会触发该任务的执行。指令 if (flag1) {...}; 表示当 flag1 为真的时候执行该任务函数。

示例 5-3

```
# define false 0
# define true 1
/*************************************************************************/
void main (void) {
/* Call RTOS run here */
rtos.run ( );
/* Infinite while loops follows in each task. So never there is return from the
RTOS. */
}
/*************************************************************************/
void task1 (....) {
/* Declarations */
```

```
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event*/
if (flag1) {....;}; flag1 =0;
/* Codes that execute for message to the kernel */
message1 ( );
}
}
/*************************************************************************/
void task2 (...) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event*/
if (flag2) {.......;}; flag2 =0;
/* Codes that execute for message to the kernel */
message2 ( );
};
}
/*************************************************************************/
.
.
.
/*************************************************************************/
void taskN (...) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event*/
if (flagN) {....;}; flagN =0;
/* Codes that execute for message to the kernel */
message2 ( );
};
}
/*************************************************************************/
```

5.4.5 指针和 NULL 指针

如果按照一定的原则正确地使用指针，它会是一个强有力的工具。下面给出了使用指针的

实例。假设每个字节都按一个存储地址存储。

(1) 假设系统中的端口 A 有一个用于存储一个字节的缓冲寄存器。使用指针的程序可以采用下面的形式对端口 A 处的字节进行声明：unsigned byte *PortA(或者 Pbyte *portA)。*号表示返回操作数所指定地址的变量的值。该声明的意思是 PortA 有一个指针和一个无符号字节。编译器将会为该字节保留一个存储器地址。考虑 unsigned short *timer1。指针 timer1 将会指向两个字节，编译器将为 timer1 的内容保留两个存储器地址。

(2) 考虑声明 void*portAdata。void 的意思是没有为 portAdata 定义数据类型。，编译器会为 *portAdata 分配存储器地址，而不进行任何的类型检查。

(3) 正如示例 4-5 中一样，可以给指针赋予一个常量的固定地址。回顾两个预处理指令# define portA (volatile unsigned byte*) 0x1000 与# define PIOC (volatile unsigned byte*) 0x1001。另外，函数中的地址也可以进行如下的赋值，volatile unsigned byte * portA = (unsigned byte *) 0x1000 和 volatile unsigned byte *PIOC= (unsigned byte *) 0x1001。指令 portA++;将使得指针 portA 指向下一个地址，即 PIOC。

(4) 考虑一下 unsigned byte portAdata 与 unsigned byte *portA=&portAdata。第一条语句指示编译器为 portAdata 分配一个存储器地址，因为每一个地址都有一个字节的空间。&返回操作数的内存地址。该声明表示 portA 指向的八位(一个字节)整数被 portAdata 地址处的字节所取代。表达式的右边对该地址所包含的字节求值，左边将该字节放置到所指的地址处。既然右边的变量 portAdata 不是一个声明过的指针，&符号将指向它的地址，以便右边的指针能够得到该地址处的内容(注意：程序语句中的=符号表示“被取代”)。

(5) 考虑两个句子，unsigned short *timer1;与 timer1++;。第二条语句在 timer1 的地址上加了 0x0002。为什么呢？timer1++表示指向下一个地址，无符号短整数类型的声明为 timer1 分配了两个地址(timer1++; 、timer1+=1; 或者 timer1=timer1+1; 执行相同的操作)。因此，下一个地址是在 timer1 原来地址的基础上增加 0x0002。如果声明为 unsigned int(以 32 位的 timer 为例)，第二条语句就会在地址上增加 0x0004。如果是一个字符数组，那么索引会以步长为 1 递增，指向前一个元素的指针实际上增加了 1。这样，如果是整数数组，那么地址会以步长为 4 递增。数组数据类型从来不会在标识符之前放置*号，但是它会在标识符后面的一对方括号中放置一个索引。考虑这样一个声明 unsigned char portAMessageString[80]。端口 A 的消息是一个字符串，即由 80 个字符组成的数组。PortAMessageString 本身就是指向一个地址的前面不带*的指针(注意：数组因此被称为引用数据类型)。但是，*portAMessageString 将指向该字符串中所有的 80 个字符。portAMessageString[20]将指向该字符串中的第 20 个元素(字符)。假设有一个可以立即启动的 RTCSWT(实时时钟中断触发软件定时器)定时器的链表。该链表的头用指针 *RTCSWT_List.top 指示。在这里，RTCSWT_List.top 是指向存储一系列 RTCSWT 的存储器头部的指针。考虑语句 RTCSWT_List.top++;，它在一个循环当中递增该指针。它不会指向链表中另一个对象(另一个 RTCSWT)的头部，而会指向某些取决于为 RTCSWT_List 中某项所分配的存储地址。假设 ListNow 为指向链表头部元素的存储器块中的一个指针。语句* RTCSWT_List.ListNow=*RTCSWT_List.top;将进行下面的操作。在这里，RTCSWT_List 指针被 RTCSWT List-top 指针取代，现在指向下一个链表元素(对象)(注意：使指针指向下一个链表对

象的语句 RTCSWT_List.top++只能在 RTCSWT_List 的元素被放置到一个数组中时使用。这是因为数组类似于在存储器的链表中连续放置元素。回顾一下表 5-2)。

(6) NULL 指针声明如下：# define NULL (void*) 0x0000(除了 0x0000 以外，我们可以赋给它任意地址，只要该地址没有正在某个给定的硬件中使用)。NULL 指针是非常有用的。考虑这样一条语句：while (*RTCSWT_List.ListNow->state!=NULL) {numRunning++;。在当前正在运行的软件定时器链表中某个指向 ListNow 的指针不为空时，只需要执行大括号中的语句组。NULL 指针的重要用途之一就是在链表中使用。它表示指向链表末尾的最后一个元素，或者空的堆栈、队列、链表。

5.4.6 函数调用

表 5-1 给出了 C 程序中各种指令集的作用。其中有一些函数和一个专门用于启动程序执行的函数 void main (void);。下面给出在程序中使用函数的步骤。

(1) 声明一个函数：就像每个变量必须声明一样，每个函数也必须声明。考虑这样一个例子。按照如下的方式声明一个函数：int run(int indexRTCSWT, unsigned int maxLength, unsigned int numTicks, SWT_Type swtType, SWT_Action swtAction, biilean loadEnable);。这里 int 定义了返回的数据类型。run 是函数名，在括号当中有参数。每一个参数的数据类型也进行了声明。修饰符定义从函数所返回的元素(变量或者对象)的数据类型。这里数据类型指定为整数类型(指定返回元素类型的修饰符也可以是 static、volatile、interrupt 或 extern)。

(2) 定义函数中的语句：就像每一个变量都必须给定内容或者数值一样，每一个函数也必须有自己的语句。考虑函数 run 的语句。在大括号中给出了如下所示的内容：int RTCSWT:: run(in int indexRTCSWT, unsigned int maxLength, unsigned int numTicks, SWT_Type swtType, SWT_Action swtAction, boolean loadEnable) {...};。函数中的最后一条语句是用于返回的，也可以返回一个元素。

(3) 调用函数：考虑这样一个例子：if (delay_f = = true && SWTDelayEnable = = true) ISR_Dlay ( );。一旦满足条件就会进行一次调用。该调用可以发生多次，并且可以反复进行。每进行一次调用，括号内给出的参数值就会传递给函数中的语句使用。

i. 传递数值(元素)

将数值复制到函数的参数当中。当函数按照这种方式执行时，调用程序时不会改变变量的值。函数只能使用通过参数复制到它的变量中的数值。考虑这样一条语句， run (int indexRTCSWT, unsigned int maxLength, unsigned int numTicks, SWT_Type swtType, SWT_Action swtAction, boolean loadEnable){...};。在调用代码的过程中，函数 run 的参数 indexRTCSWT、maxLength、numTicks、swtType、swtAction 以及 loadEnable 在程序中的初始数值将保持不变。优点是当从函数返回的时候可以给出同样的数值。通过数值所传递的参数被暂时保存在一个堆栈当中，当从函数返回的时候再取回。

ii. 再入函数

再入函数可以被几个任务和例程同时(在同一时刻)使用。这是因为其所有的参数值都可以从堆栈当中得到。当函数能满足下面的三个条件时，就可以称之为再入函数。

(1) 只要某个调用函数调用了该函数，那么所有的参数都传递数值，所有参数都不是指针(地址)。在上面的函数 run 中，所有参数都不是指针。

(2) 如果操作不是原子的，那么该函数不应该对函数外面声明的变量，或者中断服务所使用的变量，或者按照传引用的方式(而不是传值方式)作为参数传递给函数的全局变量进行操作(当对另一个程序进行调用的时候，这样的一个或者多个非局部变量的值都不保存到堆栈当中)。

回顾一下 2.1 节来理解原子操作。下面对它进行更深入的分析。假设在服务器端(软件)，有一个用于对客户端(软件)所需服务数目进行计数的 32 位变量 count。除了将 count 声明为所有客户端共享的全局变量以外没有别的选择。到服务器端的连接上的每一个客户端都会发送调用来增加 count。当满足下面的条件时，用汇编代码来实现的对该存储单元进行的加 1 操作就是非原子操作：(i)处理器是 8 位的；并且(ii)服务器端的编译器设计是这样的，它不考虑四条用于在该 8 位处理器上对 32 位变量 count 进行加 1 操作的指令之间发生中断的可能性。如果 count 正在进行加 1 操作期间发生中断，服务器会产生一个错误的数值。

(3) 该函数不会调用其他的本身不可再入的函数。假设 RTI_Count 是一个全局声明。考虑这样一个 ISR，ISR_RTI。假设存在这样一条“RTI_Count++;”指令，其中 RTI_Count 是一个用于对实时时钟中断进行计数的变量。这里 ISR_RTI 不是一个可再入的例程，因为在一个给定的处理器硬件当中，第二个条件不可能满足。由于在除了 ISR_RTI 以外的其他例程或者函数当中都不可能存在修改 RTI_Count 的操作，所以程序员不会采取任何措施来避免在 RTI_Count 的地址处发生共享数据的问题。但是如果存在另一个修改 RTI_Count 的操作，共享数据的问题就会发生(要寻找解决方法请参考第 8.2.1 节)。

iii. 传递引用

如果传递给函数的参数数值是通过指针来实现的，那么函数就可以改变该数值。一旦从该函数返回，新的数值在调用程序或者该函数所调用的另一个函数中就可用了。(以下情况都不会将数值保存到堆栈当中(a)函数的参数通过指针进行传递，或者(b)在函数中作为全局变量进行操作，或者(c)通过一个在函数块之外声明的变量进行操作。)

5.4.7 主程序中按照循环顺序进行的多函数调用

进行多函数调用最常用的方法之一，就是在主程序的无限循环中按照循环顺序进行调用。回顾示例 4-1 中的 64kbps 网络问题。我们为该问题的无限循环设计示例 5-3 中所给出的 C 代码。示例 5-4 介绍了如何在主程序中定义多程序调用，使它们按照循环顺序执行。图 5-1 给出了这里采用的模型。

示例 5-4

```
typedef unsigned char int8bit;
# define int8bit boolean
# define false 0
# define true 1
void main (void) {
/* The Declarations of all variables, pointers, functions here and also
    initializations here */
```

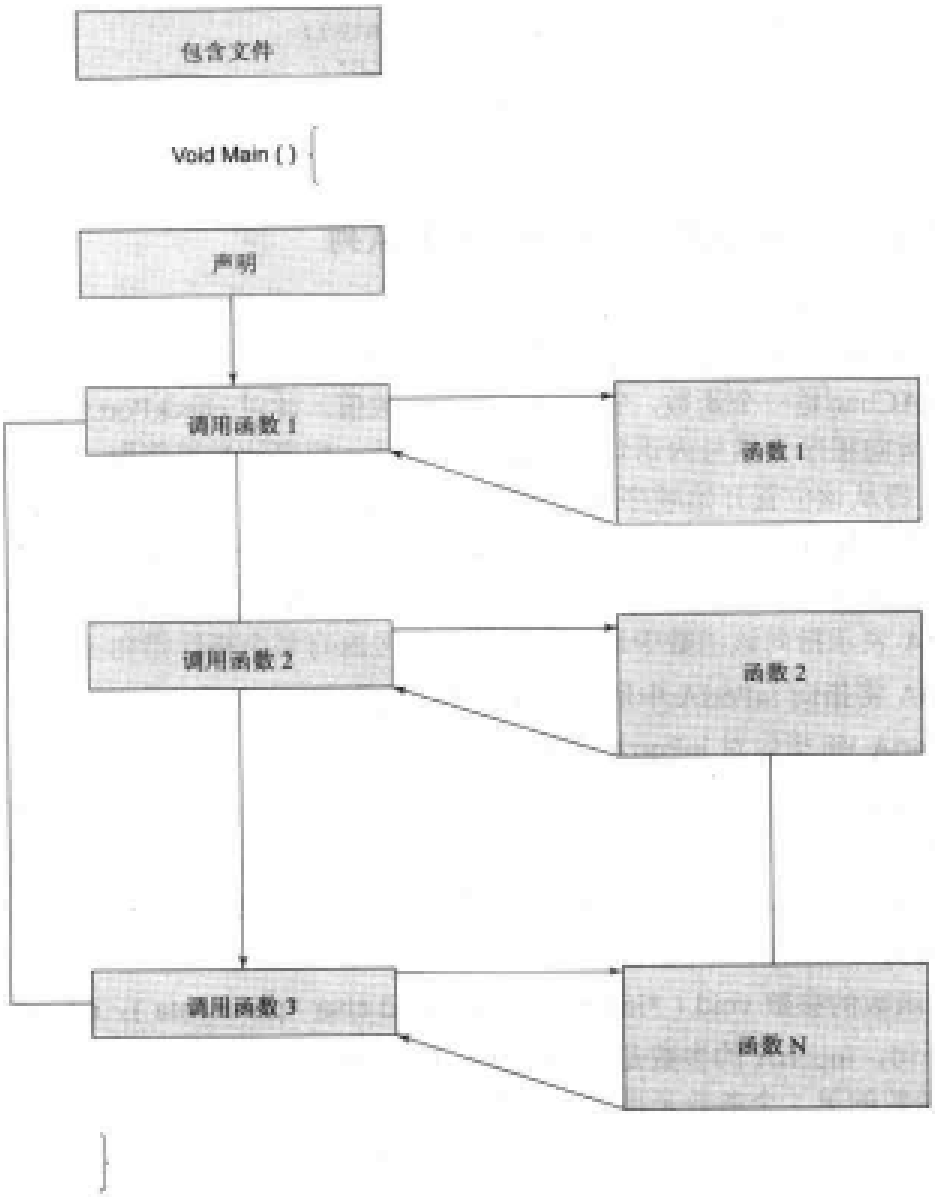


图 5-1　主程序中多程序调用的编程模型

```
unsigned char *portAdata;
boolean charAFlag;
boolean checkPortAChar ( );
void inPortA (unsigned char *);
void decipherPortAData (unsigned char *);
void encryptPortAData (unsigned char *);
void outPortB (unsigned char *);
.
while (true) {
/* Codes that repeatedly execute */
/* Function for availability check of a character at port A*/
 while (charAFlag != true) checkPortAChar ( );
/* Function for reading PortA character*/
 inPortA (unsigned char *portAdata);
/* Function for deciphering */
 decipherPortAData (unsigned char *portAdata);
```

```
/* Function for encoding */
 encryptPortAData (unsigned char *portAdata);
/* Function for retransmit output to PortB*/
 outPort B (unsigned char *portAdata);
 };
}
```

5.4.8 函数指针、函数队列和中断服务例程队列

不要将*号放置到函数名前面，但是在程序的括号内有参数，并且有发生函数调用时要执行的语句。这些语句在(被调用程序的)大括号中。考虑示例 5-4 中的声明 boolean checkPortAChar ();，checkPortAChar 是一个函数，它返回一个布尔数值。此时 checkPortAChar 前面没有*，它本身就是一个指向程序大括号内语句起始地址的指针。程序计数器将取 checkPortAChar 的地址，并且 CPU 将从该位置开始顺序执行它的程序语句。

现在，在该函数前面加上*号。“*checkPortAChar”将指向存储器中所有已经编译过的，大括号内详细说明的语句。考虑本例中的一个声明 void inPortA (unsigned char *);。

(1) inPortA 表示指向该函数中语句的指针。括号内有某个指针指向一个无符号字符。

(2) *inPortA 将指向 inPortA 中所有已经编译了的语句。

(3) (* inPortA)将指向对 inPortA 中语句的调用。

(4) 语句 void create (void (*inPortA)(unsigned char *), void *portAStack, unsigned char port Apriority);是什么意思?

a. 首先，修饰符 void 意味着 create 函数不会返回任何结果。

b. create 是另一个函数。

c. 考虑该函数的参数 void (*inPortA)(unsigned char *portAdata)。(* inPortA)表示调用 inportA 中的语句，inportA 的参数是“unsigned char *portAdata”。

d. create 函数的第二个参数是指向存储器中 portA 的堆栈的指针。

e. create 函数的第三个参数是定义 portA 优先级的一个字节。

由此应该得到一个重要的结论，即是要记住，后面跟着(* functionName) (functionArguments)的返回数据类型说明(例如 void)是使用 functionArguments 来对 function Name 中的语句进行调用的，并在返回的时候返回指定的数据对象。这样，我们就可以使用函数指针来实现函数的调用了。

当存在多个 ISR 时，较高优先级的中断服务例程先执行，最低优先级的最后执行(请参考第 4.6.4 节)。在最终期限内，较高优先级中断中的函数调用和语句可能会阻塞较低优先级 ISR 的执行。如何解决低优先级例程的最终期限问题呢？一种解决方案是在例程中使用程序指针，并为它们构造一个队列。这样，该函数就可以在稍后阶段执行(请参考第 5.5.3 节)。

注意：

在很多情况下都需要指针，例如，端口的位操作以及读、写操作。软件设计师们必须深入学习指针的用法。一个创新的概念是函数队列和 ISR 所建立的函数指针队列的使用。它明显缩短了 ISR 的延迟时间。因此每个设备 ISR 都能够在其规定的时限之内完成。

5.5　队列

5.5.1　队列

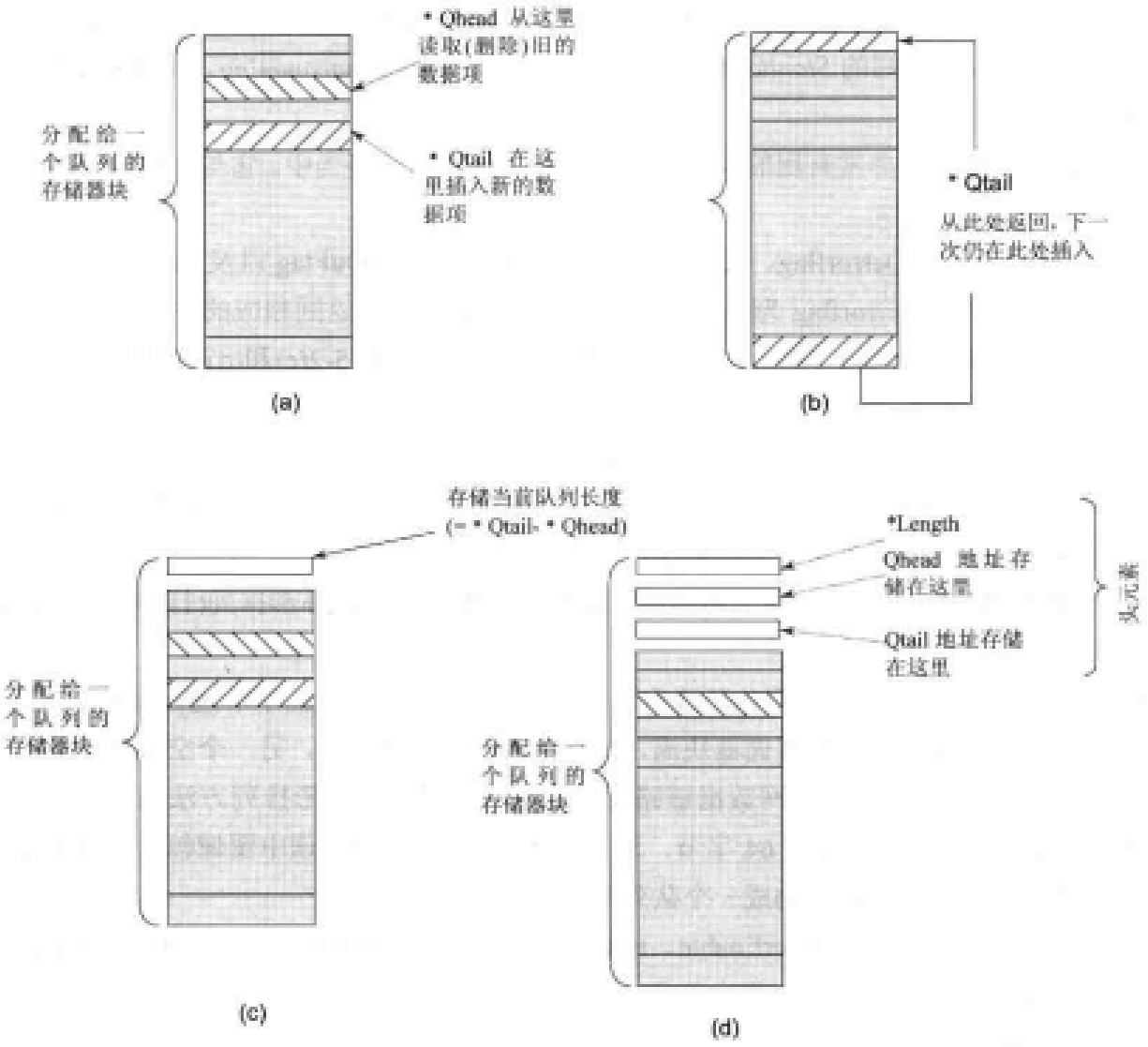


图 5-2　(a)一个队列；(b)一个循环队列；(c)一个将自身长度作为起始元素的队列；(d)一个将自身长度、源地址和目标地址作为起始元素的队列

图 5-2(a)和(b)分别给出了一个队列和一个循环队列。下面我们来实现一个由等待操作的一般类型的元素所组成的队列。假定队列中的每个元素都按照数组的形式存储在存储器中。那么队列和数组之间的差别是什么呢？差别在于元素的访问和读取。在队列中，元素的访问和读取是采用 FIFO(先入先出)模式进行。而在数组中，元素是通过其索引进行访问和读取的。队列可以被假想为元素的链表，在这个链表中，元素像在 FIFO 链表中那样访问和读取，在 FIFO 链表的中间进行写(插入)操作是不可行的，只能在最后进行。

以数组实现的队列还可以通过一个称为头(前端)的指针进行访问，通过一个称为尾(后端)的指针进行插入。

在一个具有小型存储器的系统中，队列的长度可以限制为256个元素。必要时，在存储器较大的系统中，可以在编程例子中使用短整型或者整型代替无符号字节类型。队列的大小也可以从256扩展到最大限度(在编译器没有将byte定义为数据类型的情况下就使用字符类型)。

示例5-5给出了C++类QueueElArray的代码，并假定队列最多能容纳65535个元素。(要想进行深入研究，读者可以参考标准的C++教材。比如Brooks/Cole Thomson Learning(2001)出版的由Paul S. Wang编写的*Standard C++ with object oriented programming*。第5.8节介绍了面向对象的编程概念)。

嵌入式网络系统中非常有用的设计概念都用到了代码的编写当中。它与一般的应用程序编程的不同之处在于：

(1) 有5个标识，Qerrorflag、headerFlag、trailingFlag、CirQuFlag以及PolyQuFlag。

a. 当有错误时，Qerrorflag为真。错误服务函数将根据错误返回相应的字符串。

b. 在队列元素之前存在头字节时，headerFlag为真(参考图5-2(c)和(d)中所给出的例子)。服务函数插入并返回头字节。

c. 队列元素之后存在尾字节时，trailingFlag为真。服务函数插入并返回尾字节(例如，在队列末尾放置一个check-num，接收端将它用于错误检查。表3-2、3-3、3-9以及3-10中数据后面的域也是在队列中使用尾字节的例子)。

d. 如果CirQuFlag为真，那么服务函数将对队列中字节进行环形排列(打印缓冲区的数据是循环排队方法的一个例子)。

e. 当PolyQuFlag为真时，服务函数将对队列中不同块(或帧)内的字节进行排列。多边形排列的意思是当某个保存队列的存储器块满，并且尾指针到达块尾时，另一个空块就开始插入元素。可以存在多个队列块(以太网数据链路层的数据就是采用多边形排列方法的一个例子。数据分成不同的帧(块)，每帧最小64字节，最大1518字节。视频处理中图像帧的传送是使用多边形排列的另一个例子。每帧构成一个队列块)。

(2) 有4个布尔变量，headerEnable、trailingEnable、CirQuEnable以及PolyQuEnable。

a. 如果headerEnable为真，那么前几个字节用于队列的头。

b. 如果trailingEnable为真，那么后几个字节用于队列元素的尾字节。

c. 如果CirQuEnable为真，那么只允许环形排列。

d 如果PolyQuEnable为真，那么当某个队列满时，字节只允许存储到下一个块中。

(3) 有4个无符号的短整型变量，headerNumByte、trailingNumByte、CirQuNumByte以及PolyQuBlockNum。

a. headerNumByte等于队列头的字节数。

b. trailingNumByte等于队列尾的字节数。

c. CirQuNumByte等于环形排列允许的数目。

d. PolyQuBlockNum等于多边形队列(队列块)允许的块数。

示例 5-5

```
# define false 0
# define true 1
typedef unsigned char int8bit;
# define int8bit boolean
/*Declare a constant of Assumed Size of the queue = 65536. */
static const AssumedQSize = 65536;
/* ..... Insert Codes that the type of a queue element, QElType */;
class QueueElArray {
private:
 /* Define three numbers, qhead, qtail and qsize of the queue. The qsize means
 the number of elements in a queue. The qhead means the first element. The qtail
 means the last element. Any new inserted element will be at the qtail. Any deleted
 element will be from the qhead*/
 unsigned short qhead, qtail, qsize; unsigned short qfull;
 boolean headerEnable, trailingEnable, CirQuEnable, PolyQuEnable;
 unsigned short headerNumByte, trailingNumByte, CirQuNumByte, PolyQuBlockNum;
 boolean headerFlag, trailingFlag, CirQuFlag, PolyQuFlag;
 void incCirc (int & item); /* inserting and deleting the element at the
 increasing indices circularly */
 QueueElArray (const QueueElArray & Qelement); /* Prevent calling a queue element
 using Qelement*/
 /*Define Queue Error Handling variable and function */
 boolean volatile Qerrorflag;
 static void interrupt ISR_Qerror (volatile boolean Qerrorflag, unsigned short
 [ ] );};
public :
 /* A constructor for the QueueElArray */
 QueueElArray (QElType * QelementsArray, unsigned short maxSize = AssumedQSize);
 /* The function delete in C++ is equivalent to function free in C. A destructor
 for the QueueElArray */
 ~QueueElArray (delete [ ] QelementsArray;};
 /* An operator for the QueueElArray*/
 const QueueElArray & operator = (const QueueElArray & Qelement);
 boolean isQNotEmpty ( ) const {return (qsize > 0);};
 void Qempty ( );
 void QElinsert (const QElType & item); /* A function for inserting an element
 at tail*/
 boolean isQNotFull ( ) const {return (qsize < qfull);};
 QElType QElReturn ( ); /* A function for returning an element from head*/
}; /* End of class Queue */
/***************** Constructor for Queue ***********************************/
 QueueElArray (QElType * QelementsArray, unsigned short maxSize) {qfull =
 maxSize; Qerrorflag =
false;
Qempty ( );/* Construct Empty Queue */
QelementsArray = new QElType [maxSize];
/* Now handle the errors */
If (QelementsArray = = NULL) {Qerrorflag = true; ISR_Qerror (Qerrorflag, "Error!
Queue Space
Not Available");
```

```
}

/
*************************************************************************/
void QueueElArray :: Qempty ( ) {qhead =1; Qtail = 0; qsize = 0;}
/*************************************************************************/
void QueueElArray :: QElinsert (const QElType & item) {
if (isQNotFull ( )) {
qsize ++;
incCirc (qtail);
QelementsArray [qtail] = item; Qerrorflag = false;}
else {Qerrorflag = true; ISR_Qerror (Qerrorflag, "Error! Queue Found Full");};
} /* End of insertion into the QueueElArray */
/
*************************************************************************/
QElType QueueElArray :: QElReturn ( ) {
QElType = temp;
if (isQNotEmpty ( )) {temp = QelementsArray [qhead]; qsize --; incCirc (qhead);
Qerrorflag =
false;
return (Qelement);}
else {Qerrorflag = true; ISR_Qerror (Qerrorflag, "Error! Queue Found Empty);};
/*
} /* End of a deletion from the QueueElArray */
/
*************************************************************************/
void QueueElArray :: incCirc (unsigned byte & item) {
if (++item = = qfull) {item = 0;};
} /* End of incCirc and Next Queue Element pointer back to start*/
/
*************************************************************************/
/*
Place here codes for ISR_Qerror
*/
```

5.5.2 实现网络协议的队列

网络应用需要专门的队列格式。图 5-2(c)给出了一个将当前大小(长度)作为其头元素的队列。图 5-2(d)给出了一个将自身大小、源地址和目标地址作为其头元素的队列。

在网络中，比特按顺序进行传送，并在另一端按顺序接收。为了以块、帧或包来分割这些比特，就有了头字节(包和块的不同之处在于，包能够按照不同的路径(路或管道)到达目的地。一个块可以有多个帧。帧总是按照同样的路径(管道)到达目的端口。块中的帧可能针对不同的端口，但是目的地址是一样的)。队列元素的头也要遵从协议。在队列尾部追加字节也可以有相应协议。尾部可以有 CRC(Cyclic Redundancy Check 循环冗余检查)字节。图 5-3(a)展示了队列的一个管道。图 5-3(b)给出了套接字之间的队列。图 5-3(c)给出了网络包中的三个队列。表 5-3 给出了不同的情况介绍了每种情况下如何修改示例 5-5 中的代码。

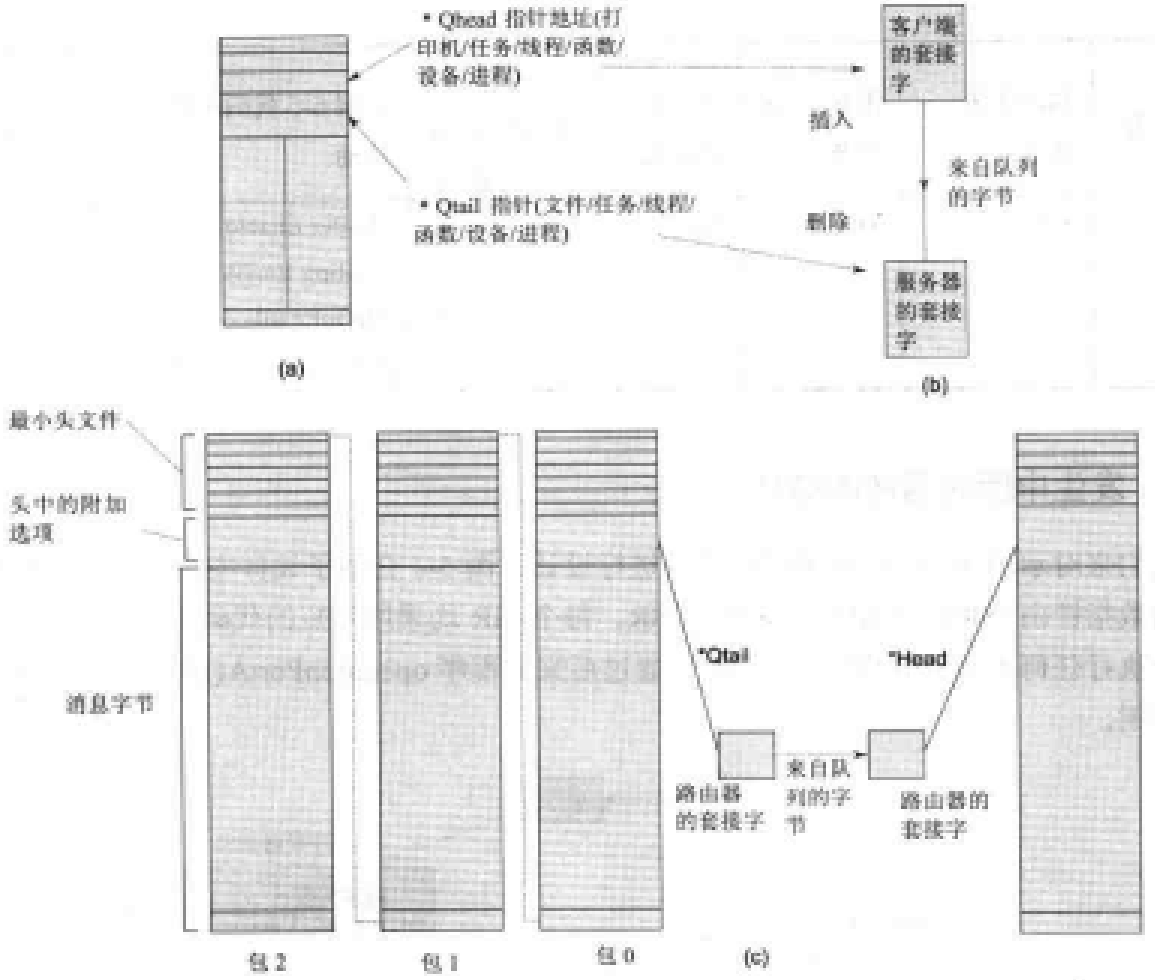


图 5-3 (a)队列的一个管道；(b)两个套接字之间的队列；(c)网络包中的三个队列

表 5-3 使用队列实现网络协议

序 号	附加了头字节	附加了尾字节	队列大小为常数	当一个队列满时形成另一个队列	示例 5-5 代码中的改动	应用示例
1.	Yes, 长度为两个字节	No	Yes	N.	Header Enable = 1; trailing Enable = 0; CirQuEnable = 1; PolyQuEnable = 0;	套接字上字符串的传输
2.	Yes, 长度为源地址加上目标地址	No	Yes	Yes	Header Enable = 1; Trailing Enable = 0; CirQuEnable = 0; PolyQuEnable = 1;	网络层协议 UDP
3.	Yes,长度为源地址加上目标地址	Yes, 使用 CRC 字节	Yes	No	Header Enable = 1; Trailing Enable = 1; CirQuEnable = 1; PolyQuEnable = 0;	以太网数据链路层

(续表)

序　号	附加了头字节	附加了尾字节	队列大小为常数	当一个队列满时形成另一个队列	示例 5-5 代码中的改动	应用示例
4.	Yes	Yes	Yes	Yes	Header Enable = 1; Trailing Enable = 1; CirQuEnable = 0; PolyQuEnable = 1;	通过路由器传送

5.5.3 发生中断时函数的排列

我们来对示例 5-4 中给出的代码重新进行设计。图 5-4 介绍了设计中所使用的编程模型。多个函数指针由 ISR 排列并且设备驱动 ISR。每个 ISR 均采用简短的代码集进行设计。在 ISR 中不会执行任何不必要的代码。它们可以通过后来的程序 operationPortA()执行，该程序由 main 函数调用。

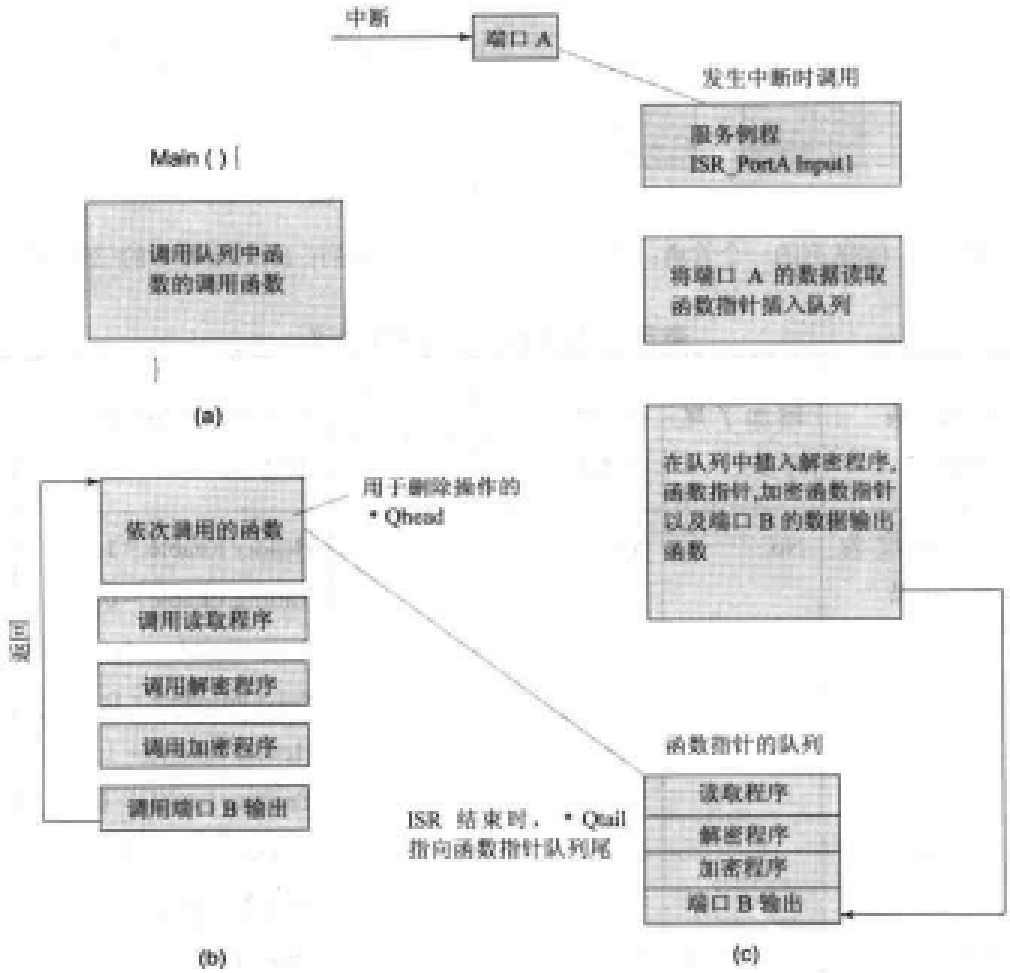


图 5-4　通过中断服务例程进行多个函数指针排列的程序模型。这些程序在后来被 operationPortA( )执行。(a)main( )函数；(b)程序 operationPortAFunctionQueues( )；(c)通过 ISR 建立函数指针的队列

示例 5-6 介绍了怎样在中断服务队列中不执行调用函数。只是通过 ISR 将这些函数的指针放入队列中。这是一个很重要的特征，ISR 现在只包含少量代码。现在，假定有多个源。每个源都有一个其服务能够满足的较短期限，使得其服务不会被错过。从某个服务队列返回时，操作程序 perationPortA( )从队列中得到了函数指针并执行所指的函数。

示例 5-6

```
/*
From Example 4.5 Insert here all preprocessor directives, commands and functions
except the main and portA_ISR_Input ( ) functions.
  */
  void main (void) {
/* The Declarations of all variables, pointers, functions here and also
initializations here */
  .
  .
while (true) { operationPortAFunctionQueues ( ); /*Call Functions from a Queue
in cyclic (Round Robin) Mode*/
  };
  }
  /*************************************************************************/
  void operationPortAFunctionQueues ( );
  unsigned char *portAdata;
  boolean checkPortAChar ( );
  void inPortA (unsigned char *);
  void decipherPortAData (unsigned char *);
  void encryptPortAData (unsigned char *);
  void outPortB (unsigned char *);
  void checkPortAChar ( );
  QueueElArray In_A_Out_B = new QueueElArray (QElType * QelementsArray, 65536);
  portAIF = false; portAIEnable = true;
  /* Codes that repeatedly execute */
  while (portAFlag != true) checkPortAChar (In_A_Out_B, STAF);
  In_A_Out_B.QElReturn ( );
  In_A_Out_B.QElReturn ( );
  In_A_Out_B.QElReturn ( );
  In_A_Out_B.QElReturn ( );
  };
  void checkPortAChar (QueueElArray In_A_Out_B, volatile boolean portAIF) {
  while (portAIF != true) { }; /*Wait till the occurrence of Port A Interrupt
*/
/* Call ISR_PortAInputI, an Interrupt Service Routine on a real time clock
interrupt. */
  void interrupt ISR_PortAInputI (QueueElArray In_A_Out_B);
  }
  /*************************************************************************/
  void interrupt ISR_PortAInputI (QueueElArray In_A_Out_B) {
  disable_PortA_Intr ( ); /* Disable another interrupt from port A*/
void inPortA (unsigned char *portAdata); /* Function for retransmit output to
```

```
Port B*/
void decipherPortAData (unsigned char *portAdata); /* Function for deciphering */
void encryptPortAData (unsigned char *portAdata); /* Function for encrypting */
void outPort B (unsigned char *portAdata); /* Function for Sending Output to
Port B*/
  /* Insert the function pointers into the queue */
  In_A_Out_B.QElinsert (const inPortA & *portAdata);
  In_A_Out_B.QElinsert (const decipherPortAData & *portAdata);
  In_A_Out_B.QElinsert (const encryptPortAData & *portAdata);
  In_A_Out_B.QElinsert (const outPort B & *portAdata);
  enable_PortA_Intr ( ); /* Enable another interrupt from port A*/
  }
  /*************************************************************************/
```

5.5.4 网络中进行流控制的 FIPO 队列

通常使用的一种网络传输协议为 Go back to N。它通常在点对点的网络当中使用。接收器应答一般出现在连续但不规则的时间段。字节从网络驱动器(传送器)传送，并排队等待达到限定数目或者发生超时(time-out)(这两者当中较早发生的那一个)。

在很多网络协议当中对“字节”、“包”或者“帧”所进行的流控制都是通过考虑来自接收实体的应答来完成的。

(1) 如果在限定的数目以内或者超时事件发生之前都没有接收到应答信号，就会对队列中的字节全部重新传送。

(2) 如果序列中任一字节有应答信号，就会重新传送第 N 个序列中的仍然没有应答的字节(因此这种协议叫做 Go back to N，也叫做滑动窗口协议。窗口的意思是在接收者从缓冲当中接收应答期间排队等待的片断或者帧)。

这里至少需要三个指针，一个用于头(*QHEAD)，一个用于尾(*QTAIL)，还有一个用于 tempfront (*QACK)。前两个指针在每个队列当中都是相同的。第三个指针定义了一个当前接收到应答的点。应答表明一个字节已经被插入(放置)到队列尾部。

对队列进行插入是在队列尾部(*QTAIL)进行的。在头和尾(*QTAIL)之间存在一个预定义的限定差。在队列尾部(*QTAIL)进行插入也存在一个预定义的时间间隔。在 tempfront(*QACK)与队列头(*QHEAD)之间也存在一个最大限制的差值。

该设计有一个必需的特征。接收一个字节的应答或者其后继应答所需要的延迟，以及传送一个字节所需要的延迟都是可变的。接收方并不对每一个字节都作应答。只有在一个连续的预定义时间段之后才会有应答。该设计可以称为 FIPO(First In provisionally Out)。图 5-5 给出了一个在网络中计算应答的 FPIO 队列，还在下面给出了三个实例的指针地址，即在传送开始时、产生应答时以及应答之后。注意，在为第 N 个序列的接收方接收到 QACK 之后，QACK 所指向的第 N 个序列与 QTAIL 当作时间函数所指向的等待序列之间的窗口，被表示为滚动的 (请参考图 5-5 当中从左到右所发生的变化)。

(1) 在传送开始的时候，头(*QHEAD)、尾(*QTAIL)以及 tempfront(*QACK)都是相等的。

(2) 当出现一个应答信号时，头(*QHEAD)就会重置，而等于 tempfront(*QACK)。

(3) 传送再一次从 tempfront 开始。

(4) 在从 tempfront(*QACK)开始传送与这个时刻之后存在一个受限的最大时差，如果 tempfront(*QACK)与头(*QHEAD)不相等，那么头(*QHEAD)就会重置，而再一次等于

tempfront(*QACK)。这表明在该限定的时间差之后，头(*QHEAD)将被强制为与tempfront(*QACK)相等。这是因为接收方在规定的时间间隔内没有产生应答。

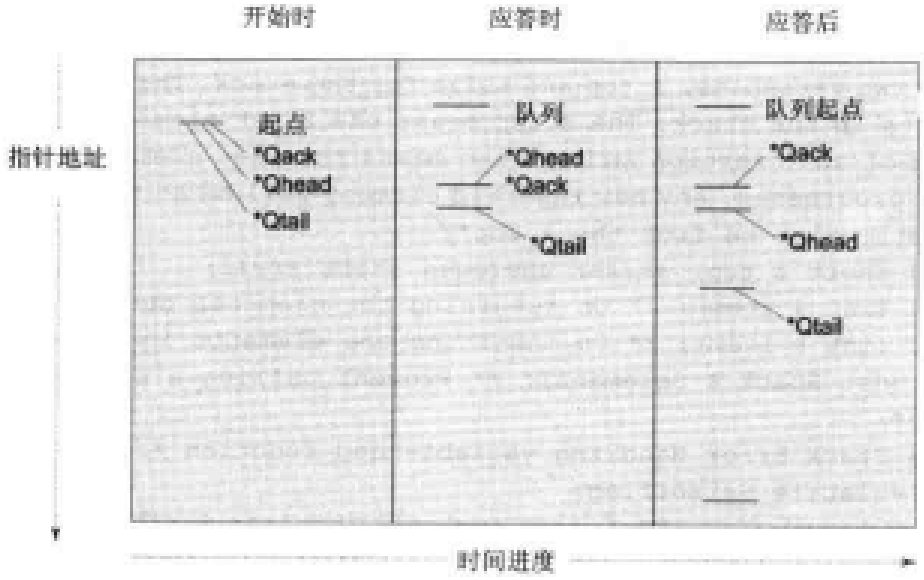


图 5-5　用于计算网络中应答信号的采用 Go back to N 协议来实现传送流控制的 FIPO 队列。注意这三个实例中三个指针的地址：在传送开始时、产生应答时以及应答之后

注意：

队列与管道都是在嵌入因网络系统中所使用的数据结构。网络处议堆栈是通过对头字节与尾字节使用专门的格因所创建的队列来实现的。创建函数指针的队列可用于解决低优先级的 ISR 超过时限的问题。

5.6　堆栈

下面我们来实现一个用于保存一般类型元素的堆栈。每个元素都被插入堆栈的最后。那么堆栈与队列之间存在什么样的差别呢？其差别就在于：在堆栈当中，元素的插入是在堆栈的栈顶指针处进行，元素只能从栈顶返回；而队列中的插入操作是在尾部进行，而从头部返回。堆栈中的访问和读取模式为 LIFO(后进先出)，然而在队列当中却是 FIFO(先进先出)。

在存储器容量小的系统当中，堆栈可以被限制为 256 个元素。如果有一个大的存储系统，

无符号字节类型 unsigned byte 就可以用 short 或者 int 类型来取代。这样就扩大了堆栈的容量(如果编译器没有将字节类型 byte 定义为一种数据类型，就使用字符类型 char)。而且，这样我们就可以有更多的元素。此处限定堆栈只能容纳 65536 个元素。示例 5-7 给出了一个最多有 65536 个元素的类的 C++代码。

示例 5-7

```
# define false 0
# define true 1
typedef unsigned char int8bit;
# define int8bit boolean
/*Declare a constant of Assumed Size of the stack = 65536. */
static const AssumedS_Size = 65536;
/* Insert Codes that the type of a stack element, SElType */
class Stack {
private:
/* Define two variables, s_top and ssize for the stack. The size means the number
of elements in the stack. The s_top means the first element pointer before the
beginning of an insertion and it also means the last element as insertion into
the stack progresses. Any new inserted element will be at the s_top. Any deleted
element will also be from the s_top*/
 unsigned short s_top, ssize; unsigned short sfull;
 void dec (int & item); /* On returning the elements decrease the indices */
 void inc (int & item); /* On inserting the elements increase the indices */
 Stack (const Stack & Selement); /* Prevent calling a stacked item using
 Selement*/
 /*Define Stack Error Handling variable and function */
 boolean volatile Serrorflag;
 static void interrupt ISR_Serror (volatile boolean Serrorflag, unsigned short
 [ ] );};
public :
 /* A constructor for the Stack */
 Stack (SElType * SelementsArray, unsigned short maxSize = AssumedS_Size);
 /* The function delete is in C++ equivalent of free in C. A destructor for the
 Stack */
 ~Stack {delete [ ] SelementsArray;};
 /* An operator for the Stack*/
 const Stack & operator = (const Stack & Selement);
 boolean isSNotEmpty ( ) const {return (ssize > 0);};
 void Sempty ( );
 void SElinsert (const SElType & item); /* A function for inserting an element
 at s_top*/
 boolean isSNotFull ( ) const {return (ssize < sfull);};};
 SElType SElReturn ( ); /* A function for returning an element from head*/
}; /* End of class Stack */
/*************** Constructor for Stack ***********************************/
Stack (SElType * SelementsArray, unsigned short maxSize) {
sfull = maxSize; Serrorflag = false;
SEmpty ( );/* Construct Empty Stack */
```

```
SelementsArray = new SElType [maxSize];
/* Handle Errors */
If (SelementsArray = = NULL) {Serrorflag = true; ISR_Serror (Serrorflag, "Error!
Stack Space Not
Available");
}
/
*************************************************************************/
void Stack :: Sempty ( ) { s_top = 0; ssize = 0; }
/*************************************************************************/
void Stack :: SElinsert (const SElType & item) {
if (isSNotFull ( )) {
ssize ++;
inc (s_top);
SelementsArray [s_top] = item; Serrorflag = false;}
else {Serrorflag = true; ISR_Serror (Serrorflag, "Error! Stack Found Full" );};
} /* End of insertion into the Stack */
/*************************************************************************/
SEltype Stack :: SElReturn ( ) {
SElType = temp;
if (isSNotEmpty ( )) {temp = SelementsArray [s_top]; ssize —; dec (s_top);
Serrorflag = false;
return (Selement); }
else {Serrorflag = true; ISR_Serror (Serrorflag, "Error! Stack Found Empty);};
} /* End of a deletion from the Stack */
/
*************************************************************************/
void Stack :: inc (unsigned byte & item) {(++item;
if (item < sfull) {Serrorflag = false;};
if (item > = sfull) {Serrorflag = true; ISR_Serror (Serrorflag, "Error! Stack
Found Out Of Bound"
);};
} /* End of increment of top of the stack*/
/*************************************************************************/
void Stack :: dec (unsigned byte & item) { item —;
if (item < 0) {Serrorflag = true; ISR_Serror (Serrorflag, "Error! Stack Found
Out Of Bound" );}
else Serrorflag = false;
} /* End of dec and Next Stack Element pointer back to s_top*/
/
*************************************************************************/
/*
Place here codes for ISR_Serror
*************************************************************************/
```

注意：

堆栈是嵌入式系统中使用的以 LIFO 方式进行访问的数据结构。

5.7 链表与有序链表

5.7.1 链表

考虑一个由一般类型的元素(对象)所组成的链表。链表与数组之间的差别如下。(i)在数组当中，存储器的分配是给每一个元素赋予一个索引。每个元素都是从第 0 个元素地址开始存储一片连续的存储器地址中。数组中每个元素(或者对象)的数值通过两个数值，即第 0 个元素的指针与索引，进行读取、置换以及写操作。如果数组是由简单数据类型构成的，那么其中的每一个元素通常都占用相同的存储空间。(ii)在链表当中，每个元素都必须包含一个数据项和一个指针 LIST_NEXT。这是因为每个元素都存储在由前导元素所指的不同的存储器地址处。LIST_NEXT 指向链表中的下一个元素。在链表末尾的元素中，LIST_NEXT 指向 NULL。每个元素所占用的存储空间也有所不同。头部的链表元素的地址是指针 LIST_TOP。只有通过使用 LIST_TOP，然后为前导元素遍历所有的 LIST_NEXT 数值才能对元素进行删除，置换或者将它插入到两个元素之间。

链表可以是有序链表。如果要进行新的插入或者删除，在进行创建的时候，所有的元素都会根据有序链表中的优先权参数重新进行排列。该参数可以是一个 unsigned byte、short、int 以及字符 ASCII 码(字母顺序)。优先权参数与数据项和 LIST_NEXT 一起被存储在每一个链表元素当中。有序链表中每一个元素都是按照赋予它的数据项优先级顺序进行排列的。图 5-6(a)~(c)分别给出了有序链表中数据项的排列，在第 1 项后做插入时进行重排以及对它的第 1 项进行删除。

链表与队列之间的差别如下：队列也可以被称为一个只能采用 FIFO 方式进行读取和访问的链表。链表中元素的插入可以发生在链表中任何地方，这可以通过遍历前导元素中的 LIST_NEXT 指针来选择。插入总是发生在队列的尾部。元素也可以从链表中的任何地方进行读取和删除。而在队列当中却总是发生在头部。

我们将存储空间小的系统中的链表元素限制为 256 个。当对它插入一个元素的时候也对该链表进行排序。在存储空间更大的系统中，我们用 short 或者 int 来替代 unsigned byte 以扩大链表的长度。

示例 5-8 给出了一个最大可容纳 256 个元素的 OrderedList 类的代码。有序链表的优点是使用函数 LelSearch( )来搜索一个更高优先级的数据项需要花费的时间更少。这是因为链表项在 ListTop 附近。对这一点可以解释如下。

我们将 LIST_NOW 设为一个链表元素的起始地址。当要对最后一项指针进行搜索时，就可以使用 for 循环。ListNow 从头部(LIST_TOP)到尾部(LIST_NEXT=NULL)是不同的。因此搜索只会在对第 1 项进行的情况下花费更少的时间。

与函数 LEldeleteLast ( )相比，使用函数 LeldeleteFirst( )对第 1 项进行删除可以节省时间。为什么呢？在对最后一项指针进行搜索的情况下，ListNow 从头到尾是不同的，这时就应该采用 for 循环，在对第 1 项指针进行搜索的情况下，删除所花费的时间就更少(图 5-6(c))。在示例 5-8 当中创建对象链表时，用到了 C++对象的多语素性质。C++对象作为链表元素进行存储。LelType 取决于构造函数怎样构建链表。它可以是字符，可以是函数指针，也可以是数组指针。构造函数声明了 LElType。搜索函数与插入函数被声明为“虚(virtual)”函数。这就引起动态运

行时的绑定。编译器使用与类 OrderList 创建的对象相关的函数，否则就使用该类创建的函数。换句话说，在进行编译和链接的时候，对象只从 ROM 当中创建函数的副本，而在运行的时候，准备执行的函数就会被复制到 RAM 当中(声明 virtual 的惟一缺点就是当进行虚函数调用的时候，需要进行额外的存储器指针查找)。

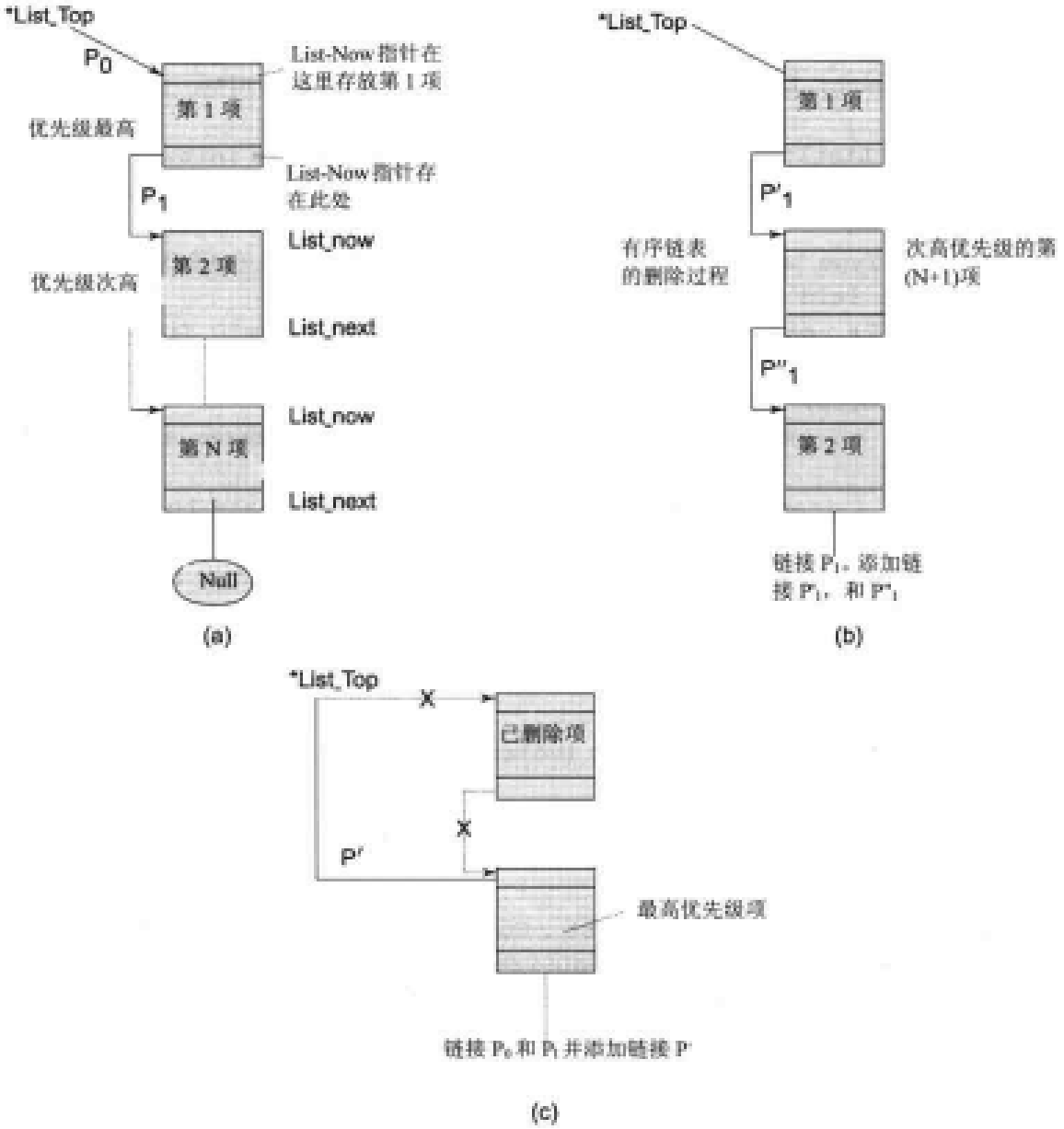


图 5-6　(a)有序链表中数据项的排列；(b)在链表的第 1 项之后进行插入；(c)对链表中的第 1 项进行删除

示例 5-8

```
# define false 0
# define true 1
typedef unsigned char int8bit;
# define int8bit boolean
/*Declare a constant of Assumed Size of the ordered list = 256. */
static const AssumedLSize = 256;
/* ..... Insert Codes that the type of an ordered list element, LElType
*/;
```

```
class OrderedList {
protected:
/* Define three numbers, lhead, ltail and lsize of the ordered list. The lsize
means that the number of elements in an ordered list. The lhead means the first
element. The ltail means last element. Any new inserted element will at the ltail.
Any deleted element will be from the lhead*/
 unsigned byte lfull; lsize, priorityOld =255;
 struc ListItem {
 LElType Lelement; ListItem *pNext; unsigned byte priority, unsigned byte
 itemID,
/* Constructor ListItem defines the arguments and default initialises the items
in the list as 0, pointer to next ListItem as NULL, item id =0, priority assignment
= 255 maximum*/
 ListItem (LElType Lel = 0, ListItem *pTemp = NULL, unsigned byte prnew = 255,
 unsigned byte itId =0)
: Lelement (Lel), pNext (pTemp), priority (prnew), itemID (itId) { }
 };
 ListItem *ListTop; ListItem *ListNow;
 void inc (int & item); /* inserting the list item increases an index when the
 list grows */
 void dec (int & item); /* deleting the list item decrease the index when the
 list reduces */
 OrderedList (OrderedList & anElement); /* Prevent calling a list item using
 anElement*/
 Void deleteAll ( );
/*Define Ordered listErrorHandling variable and function */
 boolean volatile Lerrorflag;
 static void interrupt ISR_Lerror (volatile boolean Lerrorflag, unsigned char
 [ ]);};
/* ———————————————— ————————————————————
————————————*/
public :
/* A constructor for the OrderedList . Constructed is definition of the Lelement
type as a pointer, maxSize is AssumedLSize; List top is as new ListItem constructor
and List current position now accord-ing to the previous List top. */
 OrderedList (LElType Lelement, unsigned byte maxSize = AssumedLSize) : ListTop
 (new ListItem), ListNow (ListTop) { };
/* Destructor calls the function deleteAll for freeing the memory used by an
ordered list. Declare virtual because there are the virtual declared functions
in this class*/
  virtual ~OrderedList ( ){deleteAll ( );};
/* The operators for the Ordered List Note: Before const we may use the inline
modifier so that the compiler inserts the actual codes at all the places where
we use these operators. This reduces time and stack overheads in the function
call and return. But this is at the cost more ROM codes. */
   const OrderedList & operator = (OrderedList & anElement);
   const OrderedList & operator ++ ( ) {
   if (ListNow != NULL) ListNow = ListNow -> pNext;
   return *this;}
   const LElType & operator ( ) ( ) const {
```

```
    if (ListNow != NULL) return (ListNow -> LElement);
    else return (ListTop -> LElement); };
    boolean int OrderedList & operator ! ( ) const {return (ListNow != NULL);};
   /* Empty the list */
    void Lempty ( );
    /* A function for getting to the list top */
    void top ( ) {ListNow = ListTop;};
    /* A function for assigning List first Element */
    void firstElement ( ){if ListTop -> pNext ! = NULL) {
    ListNow = ListTop -> pNext; };
   /*Test List items not than the full list*/
  boolean isLNotEmpty ( ) const {return ((ListTop -> pNext) != NULL || (lsize
  > 0));};
  virtual boolean LElSearch (boolean present, const LElType & item); /* Refer
  text for its use */
  virtual boolean LElprioritySearch (boolean present, const unsigned byte &
  priority); /* Refer
   text for its use */
  virtual boolean LElitemIDSearch (boolean present, const unsigned byte & itemID);
  /* Refer text for its use */
  /* This function if for searching the priority of the List items viz a viz the
  item to be inserted. The inserted item is after the higher priority item. */
  virtual boolean LElprioritySearchThenInsert (boolean present, const unsigned
  byte & prior-ity);
  virtual LElinsert (const LElType & item, unsigned byte & priority); /* Refer
  text for its use */
  virtual LElinsertLast (const LElType & item, unsigned byte &priority) /* Refer
  text for its use */
  virtual LElinsertPrev (const LElType & item, unsigned byte &priority); /* Refer
  text for its use */
  virtual LElinsertFirst (const LElType & item, unsigned byte &priority); /* Refer
  text for its use
*/
/*Test List items not than the full list*/
boolean isLNotFull ( ) const {return (lsize < lfull);};
/* Check in a ListItem */
 boolean checkInList (const LElType & item);
/* Delete a ListItem from anywhere. Return false if not successful else true
*/
 boolean LEldelete (const LElType & item);
/* Delete the last element */
void LEldeleteLast ( );
}; /* End of class OrderedList */
/********** Constructor for Ordered List ************************/
OrderedList (LElType *Lelement, unsigned byte maxSize = AssumedLSize) : ListTop
(new ListItem), ListNow (ListTop) {
lfull = maxSize; Lerrorflag = false;
/*Construct Empty Ordered List */
Lempty ( );
/* Handle Errors */
```

```
If ((ListTop == NULL) || (lfull == 0)) {Lerrorflag = true; ISR_Lerror (Lerrorflag,
"Error! Ordered List Space Not Available");
}
/*************************************************************************/
void OrderedList :: Lempty ( ) {
ListTop -> pNext = NULL; ListTop -> itemID = 0; ListTop -> priority = 255, lsize
= 0;
}
/*************************************************************************/
boolean LElType OrderedList :: checkInList (const LElType & item) {
ListItem *pTemp;
if (isLNotEmpty ( )) {
for (pTemp = ListTop -> pNext; pTemp != NULL; pTemp = pTemp -> pNext ) {
if (pTemp -> Lelement = = item) {return true;} else return false;};};
return false;};
}
/*************************************************************************/
boolean LElType OrderedList :: LElSearch (boolean present, const LElType & item)
{
ListItem *pTemp;
if (isLNotEmpty ( )) {Lerrorflag = false;
 if (present) {
 for (pTemp = ListTop -> pNext; pTemp != NULL; pTemp = pTemp -> pNext ) {
 if (pTemp -> Lelement = = item) {ListNow = pTemp; return true;}};};
 else {
  for (pTemp = ListTop -> pNext; pTemp -> pNext != NULL; pTemp = pTemp -> pNext)
  {
 if (pTemp -> Lelement = = item) {ListNow = pTemp; return true;}};}
 }else {Lerrorflag = true; ISR_Lerror (Lerrorflag, "Error! Ordered List Found
Empty);};
return false;};
}
/*************************************************************************/
boolean LElType OrderedList :: LElprioritySearch (boolean present, const
unsigned byte & priority) {
ListItem *pTemp;
if (isLNotEmpty ( )) {
 if (present) {Lerrorflag = false;
 for (pTemp = ListTop -> pNext; pTemp != NULL; pTemp = pTemp -> pNext) {
 priority = pTemp -> priority; return true;};
 }
 else {
  for (pTemp = ListTop -> pNext; pTemp -> pNext != NULL; pTemp = pTemp -> pNext)
 {
 priority = pTemp -> priority; return true;};}
else {Lerrorflag = true; ISR_Lerror (Lerrorflag, "Error! Ordered list Found
Empty);}; return false;
};
}
/*************************************************************************/
```

```
boolean LElType OrderedList :: LElitemIDSearch (boolean present, const unsigned
byte &
itemID) {
ListItem *pTemp;
if (isLNotEmpty ( )) {
 if (present) { Lerrorflag = false;
 for (pTemp = ListTop -> pNext; pTemp = NULL; pTemp = pTemp -> pNext) {
 itemID = pTemp -> itemID; return true;};}
 else {
 for (pTemp = ListTop -> pNext; pTemp -> pNext != NULL; pTemp = pTemp -> pNext )
 {
 itemID = pTemp -> itemID; return true;};}
else {Lerrorflag = true; ISR_Lerror (Lerrorflag, "Error! Ordered list Found
Empty);}; return false;
}
/*************************************************************************/
void OrderedList :: LElinsert (const LElType & item, unsigned byte priority)
{
unsigned byte prnow;
if (isLNotFull ( )) {
if (! isLNotEmpty ( )) {LElinsertFirst (const LElType & item, unsigned byte
priority); return;};
Lerrorflag = false;
LElpriority Search (present = true, & prnow); /* Get priority of Last List Item
*/
if (priority < = prnew) { LElinsertLast (const LElType & item, unsigned byte
priority); return;}
else{LElinsertPrev (const LElType & item, unsigned byte priority); return;}
};} else {Lerrorflag = true; ISR_Lerror (Lerrorflag, "Error! OrderedList Found
Full"); return;};
}
/
*************************************************************************/
void OrderedList :: LElinsertLast (const LElType & item, unsigned byte priority,
unsigned
priority) {
/* Find item id in case needed for the new last List Item in the last using the
function call
LElitemIDSearch (true, & itemID); */
ListItem *pLast;
ListItem *pTemp;
lsize++;
firstElement ( ); pLast = ListNow;
for (pLast; pLast != NULL; pLast = pLast -> pNext) {
 ListNow = pLast;};
ListNow -> pNext = pTemp;
pTemp = new (item, ListNow -> pNext, priority, itemID); pTemp -> pNext= NULL;
}
/* End of insertion at the last into the Ordered List */
/
```

```
*************************************************************************/
void OrderedList :: LElinsertPrev (const LElType & item, unsigned byte priority,
unsigned
priority) {
/* When needed find item id for the new last List Item in the last using the
function call */
LElitemIDSearch (false, & itemID); */
ListItem *pbefore;
ListItem *pTemp;
lsize++;
/* Retrieve the pointer of last but one list item */
firstElement ( ); pbefore = ListNow;
for (pbefore; pbefore -> pNext != NULL; pbefore = pLast -> pNext ) {
ListNow = pbefore;};
ListNow -> pNext = pTemp;
pTemp = new (item, ListNow -> pNext, priority, itemID);
pTemp -> pNext= pbefore;
pbefore -> pNext = NULL; pbefore -> itemID = itemID + 1;
}
/* End of insertion at the last but one into the OrderedList */
/*************************************************************************/
void OrderedList :: LElinsertFirst (const LElType & item, unsigned byte priority)
{
ListItem *pTemp;
itemID = 1; lsize = 1;
ListTop-> pNext = pTemp;
pTemp = new (item, ListTop -> pNext, priority, itemID);
ListNow = pTemp;
pTemp -> pNext= NULL;
}
/* End of insertion at the first element into the OrderedList when a list is
empty*/
/*************************************************************************/
boolean OrderedList :: LEldelete (const LElType & item) {
ListItem *pTemp;
/*Find the ListNow of the previous ListItem to present one where the item is
found, then delete */
if (LElSearch (false, const LElType & item) ) {
pTemp = ListNow -> pNext;
ListNow -> pNext = pTemp -> pNext;
delete pTemp; lsize-; return true;};
return false;
}
/*************************************************************************/
void OrderedList :: void deleteAll ( ) {
/* This function frees the memory space of all the pointers in a list */
ListItem *pTemp;
ListItem *pdel;
pTemp = ListTop -> pNext;
while (pTemp ! = NULL) {
```

```
pdel = pTemp -> pNext;
delete pTemp;
pTemp = pdel;};
delete ListTop;
}
/*************************************************************************/
void OrderedList :: LEldeleteLast ( ) {
ListItem *pdel;
ListItem *pbefore;
if (isLNotEmpty ( )) {return;};
firstElement ( ); pbefore = ListNow;
/* Find the last but one List Item position pointer ListNow */
for (pbefore; pbefore -> pNext != NULL; pbefore = pbefore -> pNext ) {ListNow
= pbefore; };
pdel = pbefore -> pNext; delete (pdel);
/* Define ListNow next pointer as NULL*/
ListNow -> pNext = NULL;
lsize--; }
}
/* End of deletion at the last into the OrderedList and also freeing the memory
*/
/*************************************************************************/
void OrderedList :: LEldeleteFirst ( ) {
ListItem *pdel;
ListItem *plater;
if (!isLNotEmpty ( )) {return;};
/* Find the pointer ListNow for the List Item at first position */
firstElement ( );
pdel = ListNow;
plater = pdel -> pNext;
ListTop -> pNext = plater; delete (pdel);
lsize--; }
}
/* End of deletion at the first into the OrderedList and also freeing the memory
for that element.
*/
/
*************************************************************************/
/* Place here codes for
boolean OrderedList : : LElprioritySearchAndInsert (boolean present, const
unsigned byte &
priority);
*/
/*************************************************************************/
/* Insert here codes for ISR_Lerror
*/
/*************************************************************************/
```

在示例 5-8 当中，注释中没有解释的函数解释如下：

- 当数据项存在并且布尔变量 present 的值为真时，函数 LElSearch 为 ListItem 重新获得一个新的指针 ListNow。如果搜索成功，函数就返回 true。当布尔变量 present 值为假

时，找到一个链表项之后，函数就会为前一个链表项而不是当前链表项获取指针ListNow。如果该项的搜索成功，函数就返回 true。

- 如果在调用 LElprioritySearch 的时候为 present 传递的值为真，函数 LElprioritySearch 就会为 ListNow 当前所指向的链表项重新获得一个优先级。当搜索成功的时候，函数返回 true，不成功时返回 false。如果在调用 LElprioritySearch 的时候为 present 传递的值为假，那么函数 LElprioritySearch 会为 ListNow 的前一链表项重新获得一个优先级。如果对前导链表项所进行的搜索成功，函数就返回 true。
- 当为布尔变量 present 传递的值为真时，函数 LElitemIDSearch 为 ListNow 当前所指向的链表项重新获取一个 itemID。如果搜索成功，函数返回 true。当为布尔变量 present 传递的值为假时，函数 LElitemIDSearch 就会为 ListNow 的前一链表项重新获取 itemID。如果该项的搜索成功，函数就返回 true。
- LElinsertLast、LElinsertPrev、LElinsertFirst 这三个函数分别插入一个函数中作为最后的链表项、倒数第二个链表项以及第一个链表项。
- 函数 LElinsert 测试链表没有满并且测试链表不为空。如果为空，就将插入的项作为第一项，如果没有满也不为空，就将插入的项作为最后一项。如果插入项的优先级较低，就将其作为倒数第 2 个链表项。如果插入项的优先级比当前项高，就将其插入到最后。LElinsert 是智能的！它将插入项的优先级与最后一项进行比较，然后决定是应该将其插入到最后，还是应该将其插入作为倒数第 2 项。
- 函数 LelpriorietySearchAndInsert 在需要对一个未排序的链表进行排序或者重排序时调用。

5.7.2 活动设备驱动器(软件时钟)的链表

一个系统当中可以有许多设备，因此就有许多设备驱动程序。考虑一下软件定时器的定时设备驱动程序(RTCSWT)。插入函数用在活动的 RTCSWT 中来生成链表。函数可以用 RTCSWT_List(this)来表示。所有开始运行(从实时时钟中断得到输入数)的定时器都被插入到链表中。然后在每一次发生实时时钟中断的时候，所有插入到链表中的定时器都会得到输入数。一旦到达完成状态，发生上溢时链表中的加计数器，或者发生超时时的减计数器都会从链表中删除掉。完成状态在发生上溢或者非周期(单触发)定时器发生超时的时候产生。被激活的正在运行的每一个 RTCSWT 都立刻被存储到该有序链表当中。排序是根据到达完成状态所需进行的计数多少来进行的。达到完成状态所需的计数越小优先级就越高。如果某个 RTCSWT 是单触发的减计数器，并且需要记下的滴答数越少，那么它就会完成得越早。在接合点处余下的计数更少的计数器更接近于该链表的头部。

图 5-7(a)与(b)给出了一个含有 5 个初始 RTCSWT 元素的链表的编程模型(初始 RTCSWT 的意思是已经定义了的 RTCSWT 对象)。初始的 RTCSWT 可以立即激活(运行)或者不激活(完成)。每个元素立即存储下面的 5 个程序变量：(i)优先级；(ii)项标识号；(iii)状态(激活或者不激活)；(iv)C，到达完成状态所需进行的计数；(v)指向下一个元素的指针，*pNext。图 5-7(a)给出了活动的有序链表 RTCSWT_list 中的 3 个软件定时器。图 5-7(b)给出了在活动的有序链表当中不存在软件定时器的情况。因此指向链表中元素的每一个指针都指向 Null。

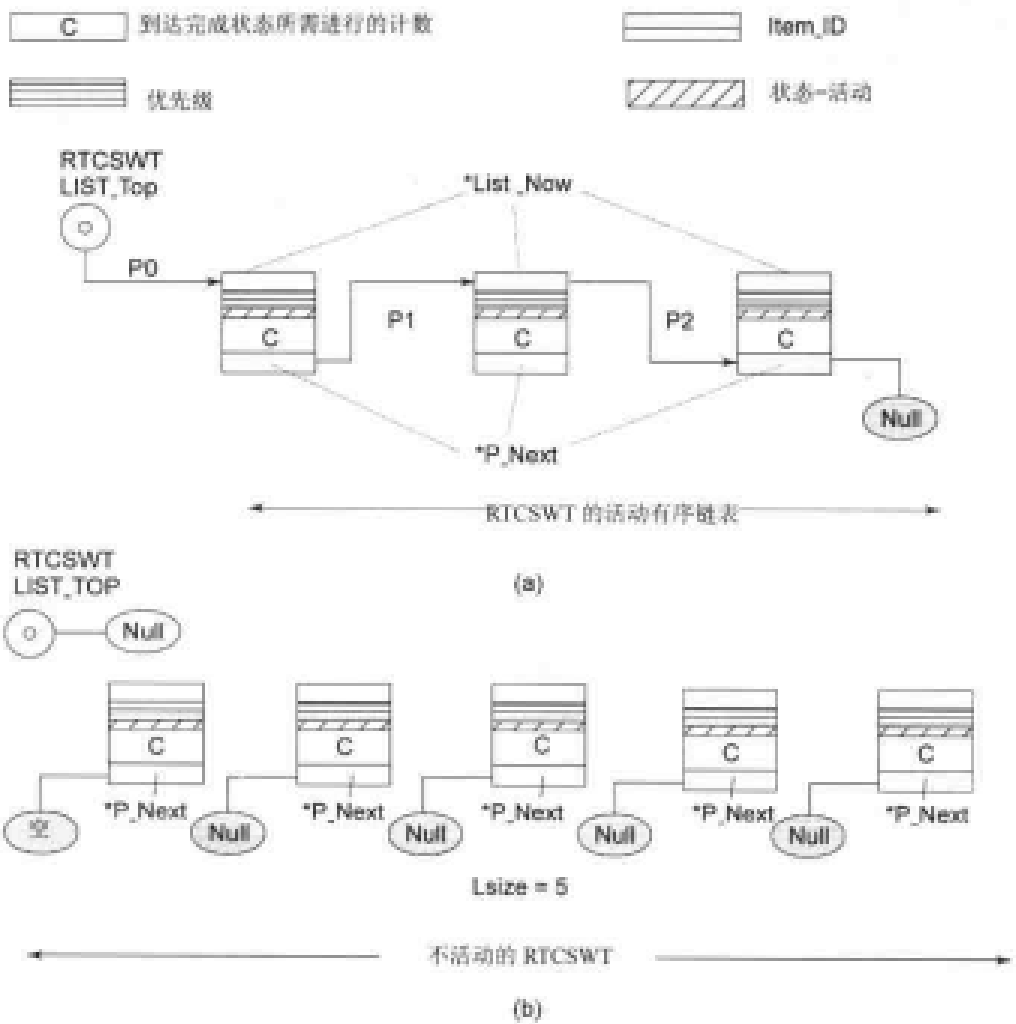


图 5-7　5 个 RTCSWT 中的活动链表 TCSWT_List 当中存在 3 个软件定时器的程序模型

5.7.3　就绪链表中的任务链表

多任务的使用，取代了函数指针队列(示例 5-6)(任务的定义请参考第 8.1 节)。每个函数都被设计为一个由多任务操作系统软件(OS)所管理的任务。任务具有可再入性。另外，它还是解决共享数据问题的方法(再入函数的定义和使用请参考第 5.4.6 节中的第(ii)点)。在就绪任务的有序链表当中完成创建或者删除是由 OS 按照下列方法进行的：

(1) 创建一个由初始(就绪)状态任务所组成的有序链表。这些任务是那些已经被标识好了的，要由 CPU 按照一定顺序执行的任务。

(2) 当 OS 收到一条消息，要将某个任务初始化为就绪状态时，处于空闲状态的任务就会插入到初始(就绪)任务链表当中。

(3) 加入到链表中的每个任务都会在 OS 的指导(调度)下执行。

(4) 已经完成(完成了代码执行)的任务会在 OS 的指导下从链表任务中删除。

多任务操作的程序模型如下。假设有 4 个 ISR。每一个 ISR 都有最少必需的代码，等待执行的函数就是任务。假设存在 6 个由 OS 管理的任务。每个元素立即存储下面的 5 个程序变量：(i)优先级；(ii)任务标识；(iii)状态(=运行、就绪或者空闲)；(iv)就绪时任务必须要执行的代码；(v)指向下一个链表元素的指针，*pNext。图 5-8 给出了这 6 个任务和 4 个 ISR 的程序模型。其

中有 3 个任务在初始任务链表当中，任务要么处于运行状态要么处于就绪状态。执行期间的 3 个步骤如下。步骤 A：ISR 发送一个事件消息(或者标志)通知需要的任务初始化为就绪状态。步骤 B：OS 将该任务插入到已经初始化了的就绪任务链表当中，并监督其执行。OS 为链表中任务的执行采用了策略。一种策略是循环(轮转)。步骤 C：当任务的所有代码都执行完毕，并且任务的状态等于空闲时，OS 就会从已初始化的链表中删除该任务。

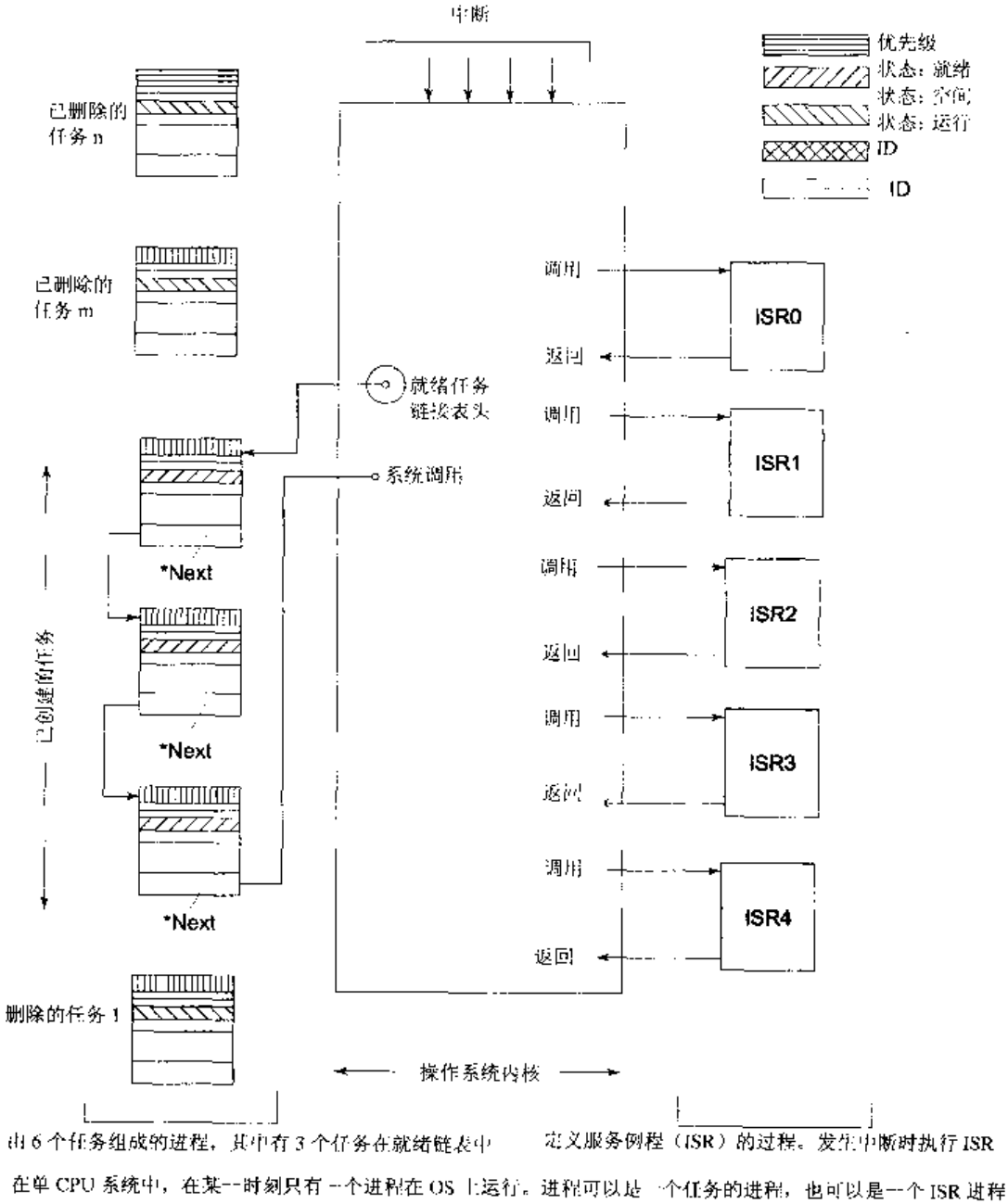


图 5-8　多任务操作的程序模型中有 6 个任务和 4 个 ISR。初始任务链表中的 3 个任务状态 = 就绪或者运行

注意：

链表与有序链表都是在嵌入式 C/C++中广泛使用的数据结构。例如，由活动的软件定时器所组成的链表，活动任务的链表以及 ISR 的链表(包括相关的设备驱动程序)。

5.8 C++嵌入式编程

5.8.1 面向对象的编程

当需要重复利用程序或者很多应用中已定义的常用对象或者对象集合时，可以使用面向对象的语言。当要构造一个大的程序时，面向对象的语言可以提供很多优点。数据封装、可重用软件组件的设计以及继承性都是 OOP(面向对象编程)的优点。

面向对象的语言提供了定义对象和无需修改定义就可以操作对象的方法。它还提供了封装数据和方法的特性。对象可以使用如下所示的特点进行描述：

(1) 一个标识符(引用保存其状态和行为的存储器块)。

(2) 一个状态(它的数据、性质、域以及属性)。

(3) 一个行为(方法或者可以对该对象的状态进行操作的方法)。

在 FORTRAN、COBOL、Pascal 以及 C 等基于过程的语言当中，大的程序都被划分为更简单的函数块和语句。在 Smalltalk、C++或者 Java 这些面向对象的语言当中，一般都会首先创建逻辑组(也就是我们所说的类)。每一组都会定义数据以及使用该数据的方法。这些组会构成一个应用程序。每个组都为数据设置了内部用户级的域以及在这些域中处理数据的方法。然后每个组都可以通过复制这个组，并对其进行功能化来创建很多对象。每个对象都可以通过与其他对象交互来处理用户的数据。语言通过对一组具有相似属性和公共行为的对象进行定义来形成类。类创建对象。对象是类的实例。

5.8.2 C++的嵌入式编程

(1) 使用 C++编程的优点是什么

C++是一种面向对象的编程(OOP)语言，它同时也支持用 C 编写的面向过程的代码。使用 C++编写的程序代码不仅具有 C 和内联汇编的优点，还同时具有面向对象编程的优点。C++嵌入式编程的概念如下所示：

i. 类将所有的成员函数与正在创建的对象捆绑在一起。该对象将会给它没有声明为静态的变量赋予默认值，并且分配存储空间。假设每个从实时时钟接受计数输入的软件定时器都是一个对象。现在考虑一个 C++类 RTCSWT 的代码。所有的软件定时器对象都可以被创建为 RTCSWT 的实例。

ii. 类也可以派生于(继承)其他的类。从父类 RTCSWT 创建一个子类就可以创建 RTCSWT 的一种新应用。

iii. 方法(C 函数)在被继承的类中可以具有相同的名称。这就叫做方法重载(method overloading)。在被继承的类中，方法不仅可以具有相同数量和类型的参数，还可以具有相同的名称，这就叫做方法覆盖(method overriding)。这是在大型程序中非常有用的两个重要特征。

iv. C++中的操作符与方法一样也可以被重载。回顾如下所示的语句和示例 5-8 中的表达式。操作符++与！都被重载来执行一组操作(通常操作符++用于之后加 1 操作和之前加 1 操作。操作符！被用于非操作)。

```
const OrderedList & operator ++( ) {if (ListNow != NULL) ListNow = ListNow ->
pNext; return * this;}
boolean int OrderedList &operator ! ( ) const {return (ListNow != NULL);};
```

(除了+操作符以外，Java 不支持操作符重载。+操作符不仅用于求和，也用于字符串连接。)

在 C 中存在可以将所有的成员函数都绑定在一起的结构 struct。但是 C++的类具有对象的特征。它可以被扩展，并且子类可以从父类中派生得到。许多子类都可以从一个通用类中派生。这个特征就称为多态性。类可以被声明为公有的或者私有的。如果类被声明为私有的，那么对数据和方法的访问都会受到限制。而结构 struct 不具有这些特征。

(2) C++的缺点是什么

程序代码会变得很长，尤其是在使用了标准 C++的某些特征以后。这些特征的例子如下所示：

a. 模板。

b. 多继承(从很多父类中派生得到一个类)。

c. 异常处理。

d. 虚基类。

e. 用于 IO 流的类(两个库函数分别为 cin(用于字符输入)和 cout(用于字符输出)。I/O 流类库提供了字符(字节)的输入输出流。它支持管道、套接字以及文件管理特征。要了解内部任务通信的用法请参考 8.3 节)。

嵌入式 C++程序可以使用经过优化的代码来消除这些缺点吗？

如果采用如下所示的方法来使用 OOP 语言，嵌入式系统代码可以得到优化。

(a) 尽可能多地将类声明为私有的。这有助于优化已经生成的代码。

(b) 用字符类型、整型以及布尔类型来代替对象(引用数据类型)用作参数，尽可能地使用局部变量。

(c) 将指向对象的指针改为 NULL 就可以恢复已经使用过的存储器。

嵌入式系统的专用编译器可以禁用 C++中提供的特定特征。嵌入式 C++是 C++的一个可以提供对上述特征进行选择的版本，以便于减少时间开销和运行时需要的库。库函数的解可以使用，并且可以直接移植到 C 中。嵌入式 C++编译器中的 IO 流库函数也是可再入的。所以，在嵌入式系统中使用嵌入式 C++编译器或者专门的编译器可以使得 C++(与 C 相比)是一种更加强大的编程语言。

GNU C/C++编译器(也叫做 gcc)在嵌入式软件开发的 C++环境中有广阔的应用空间。嵌入式 C++是一种新的编程工具，它带有一个可以提供小的运行时库的编译器。通过有选择地分解配置特征，比如模板、多继承、虚基类等，可以满足运行时对 RAM 的需要。如果运行时开销很小并且运行时库很少，那么就可以找到解决办法了。有选择地删除(可配置)的特征可以是模板、运行时类型标识、多继承、异常处理、虚基类、IO 流以及基类(基类的例子如 GUI(图形用户界面)。GUI 的例子有按钮、复选框(checkbox)按钮以及单选框按钮(radio)等)。

由 Diab Data 提供的 Diab 编译器就是一种嵌入式系统 C++编译器(除了 gcc 以外)。该编译器提供了代码的目标(嵌入式系统处理器)指定优化。(第 5.12 节)运行时分析工具检查期望的运行时错误，并给出一个能够可视交互的配置文件。

注意：

嵌入式 C++是一个通过提供面向对象编程(OOP)特征使得大的程序开发更加简单的一个 C++版本，这就将状态与行为进行了绑定，并通过某个类的一个实例进行定义。我们以一种最小化存储器需求的方式来使用对象。嵌入式系统程序员使用 C++是因为 OOP 具有以下特征，软件重用、可扩展性、多态性、函数覆盖与重载，以及 C 代码与内联汇编代码的可移植性。

C++也提供操作符的重载。编译器 gcc 广泛应用于嵌入式 C++的代码编译当中。Diab 编译器具有两个特定的特征：(i)针对处理器进行专门的代码优化，以及(ii)用于查找预计的运行时错误的运行时分析工具。

5.9 用 Java 进行嵌入式编程

5.9.1 什么时候用 Java 编程

对于嵌入式编程来说，Java 具有如下所示的优点：

(1) Java 是一种完全的 OOP 语言。

(2) Java 对于创建多线程具有内置支持(线程的定义以及它在某些方面与任务的相似性请参考第 8.1 节)。它避免了处理任务时所需的基于操作系统进行的调度(第 1.5.6 节)。

(3) Java 是在 Web 应用程序中使用最多的语言，并允许不同类型的机器在 Web 上进行通信。

(4) 网络上有一个可以加快程序开发的巨大的类库。

(5) 网络上所驻留的编译代码具有平台无关性，这是因为 Java 产生的是字节代码。它们是在机器上的一个 JVM(Java 虚拟机)上执行的(嵌入式系统中的虚拟机被存储在 ROM 当中)。对于所使用的处理器来说，平台无关性提供了可移植性。

(6) Java 支持许指针操作指令。所以在不发生内存泄漏和与存储器相关的错误时，它是很健壮的。例如，当试图在一个有界数组的末尾写入数据时，就会发生内存泄漏。

(7) 如果方法中的局部变量多于 3 个或者 4 个，那么生成的 Java 字节代码需要一个更大的存储空间。

(8) Java 具有平台无关性，可以在具有 RISC(只具有少量寻址模式的指令执行)的机器上运行(RISC 中的几种寻址模式请参阅表 A-1-1)。

5.9.2 Java 的缺点

嵌入式 Java 系统至少需要 512KB 的 ROM 和 512KB 的 RAM，因为它需要首先安装 JVM 并运行应用程序。

使用 J2ME(Java 2 Micro 版本)、Java Card 或者 EmbeddedJava 有助于将智能卡等常见应用程序代码精简到 8KB。这是怎样做到的呢？下面给出了该方法。

(1) 只使用核心类。用于基本运行时环境的类形成了 VM 内部格式，只有新的 Java 类不采用内部格式。

(2) 为配置运行时环境作准备。例如，删除异常处理类、用户定义的类装载器、文件类、AWT 类、同步线程、线程组、多维数组以及长整型与浮点数据类型。其他的配置例子还包括在需要连接的时候增加指定的类、数据图的输入、输出以及流。

(3) 在运行多线程的时候创建一个对象。

(4) 重用对象而不是使用大量的对象。

(5) 只要可行就使用标量数据类型。

智能卡(第 11.4 节)是一个带有存储器和 CPU 或者集成 VLSI 电路的电子电路。它像 ATM 卡一样被封装。对于智能卡来说，它用到了 Java 卡技术(请参考 http://www.java.sun.com/products/javacard)。运行时环境的内部格式主要在 Java 卡技术的几个类中可用。所用的 Java 类

有连接、数据图的输入输出与流、安全与加密。

上面所描述的都是嵌入式系统中 Java 应用程序的优点和缺点。JavaCard、EmbeddedJava 以及 J2ME(Java 2 Micro 版本)是 Java 能够产生精简代码的三个版本。

考虑智能卡这样一个嵌入式系统。它是使用 JavaCard 的一个简单应用程序。在字节代码中 Java 具有平台无关性的优点是一个方面。智能卡与远程服务器相连接。卡会以加密的形式存储用户账户超支和远程服务器信息的用户细节。在识别和认证以后，它进行解码，并将用户的需求与服务器通信。为复杂应用程序编写的代码在服务器上运行。只有在用于连接、数据图的输入输出与流、安全与加密的 Java 类中才存在受限制的运行时环境。

(关于 EmbeddedJava，请参考 http://www.sun.java.com/embeddedjava。它提供了一个嵌入式运行时环境和一个独立的系统。每个方法、类以及运行库都是可选的)。

J2ME 提供了一个优化的运行时环境。J2ME 只能为核心类的代码所用，而不能使用包。这些代码被存储在嵌入式系统的 ROM 当中。为两种可选的配置，连接设备配置(CDA)与连接限制设备配置(CLDC)作准备。CDC 从 net、secruity、io、reflect、security.cert、text、text.resources、util、jar 以及 zip 这些包中继承了几个类。CLDC 在 java.lang 中没有提供 applet、awt、bean、math、net、rmi、security 以及 sql 和文本包。在 CLDC 中存在一个分离的 javax.microedition.io 包。PDA 就用 CDC 或者 CLDC 配置。

J2ME 中具有可缩放的 OS 特征。有一种新的虚拟机 KVM 可以代替 JVM。当使用 KVM 的时候，系统需要 64kB 的运行时环境而不是 512kB。KVM 所具有的特征如下所示：

(1) 可以选用如下数据类型：(i)多维数组；(ii)64 位长整数类型；以及(iii)浮点类型。

(2) 错误可以使用从用于处理异常的 java I/O 包中继承的错误处理类处理。

(3) 使用分组的 API 来取代 JINI。JINI 是便携式的。但是在嵌入式系统中，ROM 已经装载了该应用程序，并且用户却没有改变它。

(4) 不对类进行验证。KVM 假设类已经验证过了。

(5) 不能终结(finalization)对象。废物回收器不必浪费时间改变对象以期望终结。

(6) 类装载器对于用户程序来说是不可用的。KVM 提供了该装载器。

(7) 线程组是不可用的。

(8) 不能使用 java.lang.reflection。因此，没有接口进行对象的串行化、调试以及分析。

J2ME 不要求必须配置 JVM 来限制类。该配置可以通过配置文件(Profiler)类得以加强。例如，MIDP(移动信息设备配置文件)。配置文件定义了 Java 对某个设备系列所提供的支持。配置文件是应用程序与配置之间的一层。例如，MIDP 就位于 CLDC 与应用程序之间。在设备与配置之间存在一个 OS，它是专门为了设备需求而提供的。

移动信息设备具有如下组件。

(1) 一个触摸屏或者一个键区。

(2) 最小不低于 96 x 54 像素的彩色或者单色显示。

(3) 无线网络。

(4) 最少不低于 32kB RAM、8kB EEPROM 或者闪存的数据存储器，以及 128kB 的只读存储器 ROM。

(5) 在 PDA、移动电话以及寻呼机中使用的 MIDP。

MIDP 类描述正在显示的文本。它描述网络的连接情况。例如， HTTP(Internet 超文本传输协议)。它为存储于 EEPROM 或者闪存中的小型数据库提供支持。它调度应用程序并支持定

时器(回顾一下 RTCSWT)。

RMI 配置文件就是用在分布式环境中的一个配置文件示例。

注意:

Java 对象通过 Java 类的实例来绑定状态与行为。EmbeddedJava 是 Java 的一个版本，它通过在 Java 当中提供完全的面向对象编程的特征使得大型程序开发更加简单。JVM 被配置来最小化系统中的存储器需求和运行开销。嵌入式系统程序员在为 IO 流、网络以及安全所编写的大量类中使用 Java。Java 程序具有在受限的情况下运行的能力。JavaCard 是一种用于智能卡中的技术，它是基于 Java 的。

5.10 C 程序编译器与交叉编译器

5.10.1 编译、可执行以及定位器文件

需要两个编译器。一个编译器用于开发、设计、测试以及调试的主计算机。另一个编译器是一个交叉编译器。交叉编译器在主机上运行，但是它为目标系统(嵌入式系统的处理器)开发机器代码。有一种叫做 GNU C/C++编译器的流行免费软件和为 68HC11 设计的免费 ASHM 汇编器。GNU 编译器与主编译器和交叉编译器一样都是可配置的。它支持 80x86、Windows 95/NT、80x86 Red Hat Linux 以及其他几种平台。它支持 80x86、68HC11、80960、PowerPC 以及其他几个目标系统处理器。

编译器生成目标文件。对于单独为主机进行的编译来说，编译器可以是 turbo C、turbo C++ 或者 Borland C 和 Borland C++。目标系统指定的或者商用的多选择交叉编译器都可以使用。对于大多数的嵌入式系统微处理器和微控制器来说这些都是可用的。瑞典的 IAR 系统为多种目标提供了交叉编译器。目标可以是 PIC 系列、8051 系列、68HC11 系列或者 80196 系列的。这些编译器可以在进行配置的时候从嵌入式系统指定的交叉编译器切换回到 ANSI C 标准编译器。PCM 是 PIC 微控制器系列，PIC 16F84、16C76 或者 16F876 的另一种交叉编译器。

主机也运行了一个提供完整开发环境的交叉编译器。这意味着目标系统可以在主机上仿真和模拟该应用系统。

注意:

图 1-6 介绍了将一个 C 程序转化为机器可实现的软件的过程。在嵌入式系统设计的最后一步中，字节必须在编译之后放置到 ROM 当中。C 程序有助于实现这一目标。当需要可执行文件的时候，对程序进行编译就会产生目标文件，它包含了具有绝对地址的源代码。可执行文件是设备程序用来将初始数据、常量、向量、表、字符串以及源代码放置(存储)到 ROM 中的文件。定位文件具有为代码、数据、初始数据等进行存储器分配的最终信息。然后定位器(locator)使用分配映射文件，生成分配地址中的源代码(第 G.2 节描述了这些文件输出纪录的(Intel 与 Motorola)两种格式。设备程序员(第 G.2 节)选用其中一种作为输入格式)。ROM 具有以下几个部分：(i)用于引导(重启)程序的机器(可执行)代码。(ii)在执行期间用于向 RAM 中进行复制的掩膜 ROM 中的初始化(默认)数据。(iii)应用程序和中断服务例程的代码。(iv)执行代码所需要的系统配置数据(例如，端口地址)。(v)标准数据、向量或者表。(vi)进行设备管理和设备驱动的机器代码。

注意：

正确使用编译器和交叉编译器是所有嵌入式软件开发中的关键。

5.11 嵌入式 C/C++的源代码工程管理工具

源代码工程管理工具对于源代码的开发、编译以及交叉编译有很大的帮助。用于嵌入式C/C++代码的工程管理、测试以及调试的工具都可以通过商业途径获得。

工具的特征包括导航和浏览的功能，可进行编辑、调试、配置(禁止和启用 C++的特征)和编译。SNiFF+是一种用于C和C++的工具。它是 WindRiver 系统的一部分。它的一个版本 SNiFF+PRO 不仅具有调试模块，而且还有完整的 SNiFF+代码。该工具的主要特征如下所示：

(1) 它搜索并列出类的定义、符号和层次，以及类的继承树(符号包括类成员。树是一种数据结构。数据结构树有一个根。从根开始产生分支，从分支产生更多的分支。在最后的分支上有叶(终结点))。

(2) 它搜索并列出符号的从属物以及已经定义的符号、变量、函数(方法)和其他符号。

(3) 它管理、启用和禁止虚函数的实现(虚函数的使用请参考第 5.7.1 节。这些是用于动态运行时绑定的)。

(4) 它能找出代码变动对源代码所造成的影响。

(5) 它搜索并列出所包含的头文件的从属物和层次。

(6) 能够在实现与符号声明之间导航。

(7) 能够在被覆盖和覆盖方法之间导航(覆盖方法是子类当中与父类具有相同名称、相同个数、相同类型参数的方法。被覆盖方法是在子类中被重新定义的父类中的方法)。

(8) 它浏览关于类的实例(对象创建)的信息。

(9) 它浏览成员之间变量的封装情况，浏览成员的公有、私有以及受保护的可视性。

(10) 它浏览对象成员的关系。

(11) 它自动删除易于出错和从未使用过的任务。

(12) 它提供自动的查找和替换操作。

注意：

复杂应用程序的嵌入式软件设计对嵌入式系统软件的程序编码、分析、测试以及嵌入式系统调试采用了一种源代码管理工具。

5.12 存储器需求的优化

在不影响代码性能的情况下，代码被压缩并存储在较小的存储空间中就称为存储器优化。它还减少了 CPU 周期的总数，因此，减少了总的功耗需求。下面所示的方法用于优化系统中存储器的使用。

(1) 如果某个变量的数值总是处于 0~255 之间，就将其声明为无符号字节类型。当使用数据结构的时候，将队列、链表以及堆栈的最大长度限制为 256。字节算术运算花费的时间比整数运算花费的时间少。

遵循这样一条规则，即在可能的情况下用无符号字节类型代替短整型，用短整型代替整型，以优化系统中 RAM 和 ROM 的使用。应尽可能地避免使用“长”整型和“双”精度类型的浮点数值。

(2) 在可能使用更简单代码的情况下应避免使用库函数。库函数是通用的函数。通用函数在某些情况下需要更多的存储空间。

遵循这样一条规则，通用函数的代码比较简单时，如果它占用的存储空间更大，应避免使用库函数。

(3) 在软件设计者完全知道目标处理器指令集的情况下，必须使用汇编代码。这也有利于存储空间的有效利用。由于需要为控制和状态寄存器使用位 set-reset 指令，用汇编语言编写的设备驱动程序特别有利于提高效率。只需要几条用于使用设备 I/O 端口地址、控制和状态寄存器的汇编指令。其最佳的使用价值是由给定应用程序的可用特征带来的。汇编编码也有助于对原子操作进行编码。为了能够快速访问某个经常使用的变量，可以在 C 程序中使用修饰符寄存器。如果经常使用 portA 的数据，就可以这样使用，register unsigned byte portAdata。修饰符寄存器指示编译器将存放到该处理器的通用寄存器当中。

应遵循的规则是，如果指令集非常清晰易懂，对配置设备控制寄存器、端口地址以及位操作等简单函数应该使用汇编代码。对加 1 和加法等原子操作使用汇编代码。对经常使用的变量使用修饰符 register。

(4) 调用函数会使得上下文保存到一个存储器堆栈当中，函数返回时再取回上下文。这涉及到时间，会增加最坏情况下的中断延迟。有一个 inline 修饰符。当使用 inline 修饰符时，编译器会在所有用到这些操作符的地方插入实际的代码。这样可以减少在函数调用和返回过程中的时间开销和堆栈开销。但是，这是以代码需要更多的 ROM 为代价的。如果使用，它会增加程序的大小，但可以得到更快的速度。使用修饰符指示编译器用(在花括号中的)函数的代码来代替调用该函数。

应该遵循的规则是，如果 ROM 在系统中可用，那么应该对函数中经常使用的代码组或者操作符重载函数使用 inline 修饰符。空的 ROM 存储器是未使用的资源。为什么不使用它，从而通过减少经常性地保存和取回程序上下文来降低最坏情况下的中断延迟呢？

(5) 只要不发生共享数据问题，全局变量的使用就可以得到优化。它们与用于传递数值的参数不同。好的函数应该是没有参数需要传递的函数。在进行中断服务调用和其他函数调用的情况下，传递的数值保存在堆栈当中。除了可以不必重复进行声明以外，全局变量的使用还可以减少最坏情况下的中断延迟，以及函数调用和返回时所需要的时间和堆栈开销。但是这是以用于消除共享数据问题的代码为代价的。如果某个变量声明为静态变量，那么处理器访问需要的指令比从堆栈访问的情况下要少。

应该遵循的规则是，如果遇到了共享数据问题就使用全局变量，如果经常需要在堆栈中保存就使用静态变量。

(6) 尽可能地将两个函数进行组合。例如，在示例 5-8 中的函数 LElSearch (boolean present, const LElType & item)就是一个组合函数。它是由用于查找指向某个链表项的指针的搜索函数和前一链表项的指针搜索函数组合而成。如果 present 为假，前一链表项的指针就重新取回含有该项的那一个。

应该遵循的规则是，尽可能将代码类似的两个函数组合在一起使用。

(7) 回顾一下第 5.7.2 节和 5.7.3 节中由正在运行的定时器所组成的链表和由已经初始化了

的任务组成的链表。还使用了在定时器运行时改变计数输入，和在定时器处于空闲状态时不改变计数输入的所有定时器和条件语句。然而需要更多的条件语句，不只是一次，而是在每次实时时钟中断时都重复调用。所需的 RAM 存储器将会更多。因此，创建一个由运行中的计数器组成的链表是一种更有效的方法。首先，将任务存放到一个已经初始化了的任务链表(图 5-8)当中，将减少与操作系统之间发生的频繁交互，避免了将上下文保存到堆栈和从中取回的开销以及时间开销。优化堆栈的 RAM 使用。这是通过减少与操作系统间交互的任务数而实现的。一个函数调用另一个函数，这个函数再调用第三个函数，依此类推，这就是嵌套调用。减少嵌套调用的个数，最好在一个函数当中调用一个以上的函数。这样可以优化堆栈的使用。

应该遵循的规则是，减少函数调用和嵌套调用的使用频度，这样分别可以减少时间和堆栈所需要的 RAM 存储空间。

(8) 尽可能用指向函数的指针来代替 switch 语句。这样可以为在沿着某条链执行条件测试时决定执行哪一组语句更节省处理器时间。

(9) 如果执行之后不再需要某一组语句，那么可以使用删除函数。

应该遵循的规则是，如果要释放某组语句所使用的 RAM 空间，可以使用删除函数或者析构函数。

(10) 使用 C++的时候，应该对编译器进行配置，使之不支持多继承、模板、异常处理、新的样式转换、虚基类以及命名空间。

应该遵循的规则是，对于 C++，使用类不能多继承，不使用模板，使用运行时标识和可抛出的异常的类。

注意：

嵌入式软件设计者必须使用各种标准方法来优化系统中的存储器需求。

本章小结

- 使用汇编语言进行编程可以对处理器内部设备进行精确控制和完全使用其指令集中的特定特征和寻址模式。
- 用高级语言进行编程可以缩短复杂系统的开发周期以及方便地修改系统硬件。
- C 语言对内联汇编(用汇编语言编写的代码段)提供支持可兼获上面两种好处。
- 在满足下列条件时，C++就能提供 C 和面向对象编程的所有优点，适合于嵌入式系统：(i)尽可能将更多的类声明为私有。这有助于优化生成的代码。(ii)使用字符类型、整数类型以及布尔类型(标量数据类型)作为参数来取代对象(参考数据类型)并尽可能地使用局部变量。(iii)通过把指向某个对象的引用改为 NULL，来恢复曾经使用过的存储器。(iv)有选择地分解配置 C++的某个特征使得运行时开销更小，使用的运行时库更少。(v)有选择地删除模板、运行时的类型标识、多继承、异常处理、虚基类、IO 流以及基类等特征。
- Java 提供了可用的扩展类库、模块性、机动性、可移植性以及平台无关性等好处。
- C 程序使用各种指令元素、预处理指令、宏以及常量，包括文件、头文件和函数。基本的 C 程序元素有数据类型、数据结构、修饰符、条件语句、循环、函数调用、多函数、函数队列、服务例程队列。

- 无限循环是嵌入式系统当中广泛使用的特征，因为它总是使某个任务或者系统在被调用时运行。
- C 程序通过引用函数、指针、空指针以及函数指针来传递变量的数值。
- 队列是程序当中使用的一个重要数据结构。与函数相关的队列数据结构有“构建”一个队列，向队列中“插入”一个元素，从中删除一个元素以及队列的“析构”。队列是一种先进先出的数据结构。流中的字节队列在网络通信或者客户端-服务器通信中也扮演重要的角色。一种采用 FIPO 策略进行流控制的新式队列数据结构经常应用于嵌入式网络系统当中。被删除的元素是先进来的，并且之前预备取出的字节。当接收它时，它重新从应答队列中插入到该队列中。
- 对发生中断时指向函数的指针进行排队，然后再从该队列调用函数是一种比较好的方法(编程模型)，因为它提供了执行时间比较短的中断服务例程。
- 在发生中断或者进行函数调用的情况下，使用“堆栈”来保留数据是经常发生的。与堆栈相关的函数有“构造”一个堆栈，向堆栈压入一个元素，从堆栈中弹出一个元素以及堆栈的“析构”。
- 与“链表”和按优先级排列的“有序链表”相关的函数有“构造”一个链表，向链表中“插入”一个元素，从中查找一个元素，从中删除一个元素以及链表的“析构”。由实时时钟中断驱动的软件定时器所组成的链表是它应用的一个例子。另一个例子是调度多个任务的就绪任务所组成的链表。
- 一旦源代码就绪，编译器和交叉编译器使用使能定位器来提示设备程序员将机器代码存储到系统 ROM 当中。
- 源代码管理工具有助于对用高级语言编写的代码进行调试和性能分析。
- 嵌入式系统程序应该根据存储器需求进行优化。要得到优化的程序需要进行很多步骤。除了可以减少必需的存储器大小以外，它还可以减少总的 CPU 周期数，因此可以减少总的能量需求。

关键词及其定义

- 高级语言(high-level language)：这是一种比汇编语言更容易编写代码的编程语言。它还可以缩短复杂系统的开发周期，并便于系统硬件的修改。
- 开发周期(Development Cycle)：编写代码，测试并调试的一个周期。结束编程之前需要数个周期才能将源代码存储到嵌入式系统的 ROM 当中。
- 内联汇编(in-line assembly)：高级语言中的一个用汇编语言编写的代码段，它可以带来处理器指定指令和寻址模式的好处。
- 面向对象编程(object oriented programming)：它是一种编程方法，操作是面对对象进行的，而不是对数据类型、数据结构、变量以及函数单独进行的。
- 类(class)：包含很多成员——变量、函数等一组命名代码可以从它那里创建对象。在面向对象编程当中，操作是通过将信息传递给对象来实施的。每个类都是一个带有标识名、状态和行为说明的逻辑组。

- 标量数据类型(scalar data type)：字符类型、整数类型、无符号整数类型、浮点数类型、长整数类型以及双精度类型都称为标量数据类型。与数组不同的是，数据由一个元素组成。
- 私有(private)：一个变量只属于一个特定的类，在该类之外是不可用的。
- 引用数据类型(reference data type)：数组和字符串都是引用数据类型。
- 局部变量(local variable)：在某个函数中定义的，在其他函数当中不可修改的变量。
- NULL：当指针指向 NULL 时，意味着没有指向存储器中的任何单元。由某个元素、对象或者数据结构占用的存储单元可通过将指向它的指针指向 NULL 来释放。
- 运行时开销(runtime overhead)：数据和堆栈所使用的 RAM 称为运行时开销。
- 运行时库(Runtime Library)：运行的时候动态链接的库函数。运行时进行链接会增加运行时的开销，并引发内存溢出的错误。
- 模板(template)：使用以后可以创建新类的一组类。
- 多继承(Multiple Inheritance)：从多个类中继承成员函数的子类(派生类)。
- 异常处理(exception handling)：一种在处理某个异常条件发生时调用函数进行处理的方法。例如，缓冲区不能再存储更多的字节。程序员考虑异常发生的条件，并在该异常发生时提供函数，并实施调用。
- 虚基类(Virtual Base Classes)：C++中所提供的一种特殊类型的类。
- IO 流(IO stream)：它是通过从源地址到目的地址发送字节或者字符创建的一个存储器缓冲，因此目的地址像一个接收器，按照它们发送的顺序接收。IO 流对象对文件、指针、队列或者网络设备进行写操作。
- 基类(foundation class)：用于 GUI(图形用户接口，例如，按钮、复选框、菜单等)的类。
- 类库(class libraries)：用于加密和安全等应用中的类可以经过仔细地调试和测试在需要的时候使用。类库的使用可以缩短程序的开发周期。
- 模块性(modularity)：如果一组代码在多个应用程序中都可以使用，那么我们将这组代码当作一个模块。
- 机动性(robustness)：在没有发生堆栈上溢和内存溢出等错误的情况下，如果程序可以正常运行，那么该程序是机动的。避免指针操作指令，如果后来不再使用而释放存储器，使用异常可以使得代码具有机动性。
- 可移植(portable)：可以适当地改变配置，将代码移植到另一个程序当中。
- 平台无关性(platform independence)：代码可以移植到不同的机器和操作系统当中。
- 预处理指令(preprocessor directives)：主程序定义全局变量、全局宏(代码部分)、新的数据类型以及全局常量之前的程序语句和编译器指令。
- 包含文件(include file)：编译之前，与用户源代码一起被编译器包含进来的文件。
- 头文件(header file)：为用户包含代码(主要是标准函数)的文件。例如，文件“math.h”包含了数学函数的代码。
- 数据类型(data type)：变量数据的类型，例如，整数类型。
- 数据结构(data structure)：可以用一个公共名称引用的多元素结构。
- 函数队列(function queue)：由指向等待执行的函数的指针组成的队列。
- 无限循环(indefinite loop)：一个不能从程序中退出的循环，除非发生中断或者它所使用的某个参数发生改变。

- 传数值(passing the value)：某个数值从一个函数传递到另一个函数，但是从所调用的函数返回之后，同样的数值被重新赋给原始的函数。在进行传递以前，参数的数值被存储到堆栈当中，从函数返回的时候再取回。
- 传引用(passing the reference)：当把处理传递给另一个调用函数时，参数值的地址从一个函数传递给被调用的函数。当进行操作时，被调用的函数参数就变得可以修改了。在返回调用函数之后，该函数可以取回不同的参数值。通过引用传递时，参数的数值并不保存在堆栈当中。
- 队列(queue)：元素可以顺序插入，并按照先进先出(FIFO)的模式取出的数据结构。它需要两个指针，一个指向队列尾(后面)进行插入，另一个指向队列头(前面)进行删除(读取并指向下一个元素)。
- 堆栈(stack)：元素可以被压入以保存到某个存储块中，可以采用后进先出(LIFO)的模式取出的数据结构。它需要一个指针指向栈顶以进行插入和删除(读取并指向下一个元素)。
- 链表(list)：元素可以顺序插入和取出，而不必采用先进先出模式或者后进先出模式的数据结构。每一个元素都有一个指针指向该链表中的下一个元素。最后一个元素指向Null。还有一个顶(头)指针指向它的第一个元素。
- 有序链表(ordered list)：一个按优先级排序的链表，很容易对它进行删除操作(读取，然后将指针指向下一个元素)。它是从顶开始顺序进行的。
- 源代码管理工具(Source code engineering tool)：一种用于管理源代码的强大工具，也可以对使用高级语言编写的代码进行调试和性能分析。
- 存储器的优化(Optimisation of Memory)：改变某些步骤以减少对存储器的需求，得到精简的代码。它可以减少总共所需的存储器大小。也可以减少 CPU 周期总数，从而可以减少总的能量需求。

问题回顾

(1) 为一个给定的嵌入式软件选择适当的编程语言，什么是关键？

(2) C 的哪些特征使得它能够成为嵌入式系统中使用率最高的高级语言？

(3) 在 Java 中，什么特征使得它能够在很多与网络相关的应用程序中成为嵌入式系统最有用的高级语言？

(4) 在使用 C++进行编程的时候，多态性的优点是什么？

(5) 为什么要将一个程序分为头文件、配置文件、模块和函数？

(6) 设计一个表格列出程序自顶向下设计和自底向上设计的特征。

(7) 解释下面这些声明的重要性：静态、可变的，嵌入式 C 中的中断。

(8) 在 C 程序中如何使用并且何时使用(a)# define；(b)typedef；(c)空指针；(d)传引用；(e)递归函数

(9) 使用免费软件 GNU C/C++编译器的优点是什么？

(10) 为什么需要交叉编译器？

(11) 为什么要在嵌入式系统中使用无限循环？

(12) 在嵌入式系统软件中使用再入函数的优点是什么？

(13) 在主函数中按照循环顺序使用多函数调用有什么优点？

(14) 建立ISR队列的优点是什么？

(15) 使用为后面的处理建立函数队列的短ISR有什么优点？

(16) 队列应该如何用于网络？

实践练习

(17) 为什么在使用模板、多继承(从多个父类产生一个子类)、异常处理、虚基类以及IO流的时候，C++的特征会使得代码冗长？用表格列出原因。

(18) 用C语言中包含内联汇编代码的形式编写一个COM串口的设备驱动程序。

(19) 使用最多的预处理指令有哪些？每个指令举出4个例子。

(20) 宏与函数有什么不同？使用代码示例进行解释。

(21) 编写一个C程序利用循环使10个整数只与奇数索引的数值累加相加。每个整数都是32位。然后展开循环，重新编写C代码。比较两种情况的代码长度。

(22) 要处理某个视频帧中的一组图像。什么数据结构最适合在以适当的格式压缩之前存储输入？

(23) 将两个函数组合起来是如何减少存储器需求的？请用4个例子进行解释。

(24) 请参考第3章的习题16。它给出了PPP的格式。请编写一个C程序来传递一个封装了4096个数据位的PPP格式数据帧。这些位要按照以big-endian格式存储在存储器中的32位整数的顺序传递。

(25) 给出两个嵌入式软件中使用了下列数据结构的编程示例：(a)数组；(b)队列；(c)堆栈；(d)链表；(e)有序链表；(f)二进制树。

第6章 单处理器和多处理器系统软件开发过程中的程序建模概念

本章前所学内容

我们来对前面章节中所学的与软件开发过程相关的如下主题进行概述。

(1) 嵌入式系统软件的概念，它是所有嵌入式系统设计中最关键的部分。

(2) 代码生成或者源代码管理工具、编译器、交叉编译器以及生成最终代码以形成 ROM 映像的定位器。

(3) 面向过程和面向对象编程语言的概念，设备驱动程序的编码以及使用修饰符、条件语句和循环、指针、函数调用、多函数、函数指针、函数队列以及服务例程等 C/C++编程元素的应用程序。

(4) 使用汇编语言为特定处理器敏感或者存储器敏感的指令编写的程序。这些编码通常用 C 语言内联汇编。

(5) 不同数据结构(如数组、队列、堆栈、链表和树)，不同类型数据的使用，以及使用数据结构进行编程的例子。

(6) 程序中优化系统存储器需求的方法。

本章将学内容

工程师所采取的标准设计就是在将要找到问题的解决办法时使用一个模型。建模能够为程序所用吗？本章的首要目标是学会在软件实现之前进行程序建模的重要概念。下面对在软件实现之前进行程序建模的重要概念进行了解释：

(1) 进行程序设计和分析的数据流图。

(2) 进行程序设计和分析的受控数据流图。

大多数时候，嵌入式系统都需要实时编程。实时编程是对响应、处理延迟以及处理时限受到约束的处理器或者指令集进行编程。实时软件设计的复杂问题将在下一章中阐述了软件开发过程模型以及分析、设计、实现和测试阶段中的活动之后进行描述。实时操作系统和它们的工作原理将在后面章节中深入探讨。本章的第二个目标是学会如下所示的为实时程序进行程序建模的重要概念。

(1) 用于程序模型和实时程序分析的有限状态机。

(2) 用于程序模型和实时程序分析的 Petri 网。

如果使用两个或者两个以上的处理器，当系统同时运行多个进程时，(i)程序函数；(第 5.3 节)；(ii)任务(第 8.1.2 节)；(iii)单指令多数据流(SIMD)指令；(iv)多指令多数据流(MIMD)指令；(v)DSP 指令的超长指令字(VLIW)都可以高速完成(指令字的意思是操作码加上其操作数)。很多复杂的嵌入式系统都是多处理器系统，这使得进程的延迟短，因此可以满足时限。本章的第 3 个目标是揭示如下所示的用于多处理器系统建模中的概念。

(1) SDF 图

(2) 展开 SDF 图之后的 HSDF 表示

(3) APEG

(4) 定时 Petri 网

(5) 扩展的预测/转换网

(6) 多线程图(Multi Thread Graph，MTG)系统

(7) 在程序流的过程中，模型在多个处理器上进行划分、负载平衡、调度、同步以及再同步中的应用。

UML 是一种功能强大的建模语言。除了这些建模过程之外，理解软件工程方法和实践也是必不可少的。还应该理解对于一种能够对多种类型的进程、活动以及设计和开发过程的方法进行建模的统一语言的需求。这些内容将在下一章中进行阐述。

6.1 软件实现之前对软件分析过程的建模

软件分析是(i)系统需求的一种描述；(ii)在进行软件设计之前准备的一个基本框架；(iii)一组需求的定义，设计好的软件可以用其进行验证。在进行软件分析的时候，数据流图(缩写为 DFG)和控制流图(缩写为 CDFG)用于对软件的数据路径和程序流进行建模。

6.1.1 数据流图在程序分析中的用法

数据流的意思是程序流和所有的程序执行步骤都仅由数据决定。软件设计者预先确定数据输入，并设计编程步骤来产生数据输出。例如，一个用于确定各门课程平均分的程序将把分数作为数据输入，将分数的平均值作为数据输出。该程序执行一个函数来产生适当的输出。数据流图模型适用于对这个求均值的程序进行建模。

程序中的数据流是怎样的呢？输入的数据在数据流之后就变成了输出。一个称为数据流图(DFG)的图将对这一过程进行图形化表示。DFG 没有任何条件，因此该程序有一个数据入口点和一个数据输出点。程序在执行的时候，程序流只存在一条独立的路径。

每个圆圈表示 DFG 中的每个过程。有一个箭头指向该圆圈表示数据输入(或者输入集合)，从该圆圈出发的箭头表示一个数据输出(或者输出集合)。

如果每个输入只有一组数值，且对于给定的输入只有一组输出的数值，那么该 DFG 也称为 ADFG(Acylic Data Flow Graph，无环数据流图)。无环数据输入的例子如下：(i)一个事件；(ii)某个设备中设置的一个状态标识；(iii)前一个过程的每个输出条件作为输入。

示例 6-1 给出了一个 DSP 算法的 DFG。

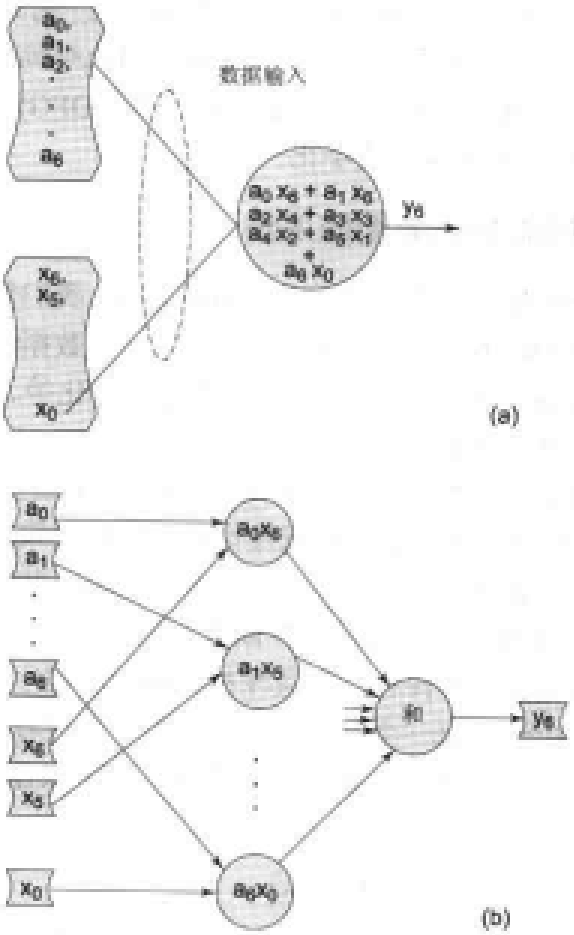


图 6-1　(a)第 6 个 FIR 序列过程的 DFG；(b)FIR 滤波器的同一序列的一组进程的 DFG

示例 6-1

图 6-1 介绍了将如下表达式作为一个“有限脉冲响应(FIR)滤波器”的输出序列 y_6 而得到的 DFG(回顾一下第 2.1 节的第 n 个滤波输出序列 $y_n = \sum(a_i x_{n-i})$，其中 $i = 0, 1, 2, \cdots, N\text{-}1$)。图 6-1(a)展示了第 6 个 FIR 序列过程的 DFG，图 6-1(b)展示了同一组序列过程的 DFG。下面是计算 $y_6 = a_0x_6 + a_1x_5 + a_2x_4 + a_3x_3 + a_4x_2 + a_5x_1$ 时应该注意的要点：

(1) 计算 y_6 的圈所表示的过程有一个输入点。

(2) y_6 有一个输出点。

(3) 每个系数和每个滤波器输入都只有一个存储器地址和变量。x 的 6 个输入都只有一个数值，且每个系数 a 都只有一个数值(因此这里的 DFG 也是 ADFG)。

这种获得输入和进行求和的顺序并不重要的。

在示例 6-1 中，你一定注意到处理 y_6 的过程并不复杂。数据流图模型有助于简单的代码设计。简单的代码设计可以定义为程序通常被划分为 DFG 的设计。DFG 对具有一条独立路径的基本程序元素进行建模。它提供一个没有控制条件，从而程序流只有一条单路径的系统单元。单元给出了程序的上下文，并根据复杂性帮助分析程序。复杂的程序与简单的程序相比，具有的 DFG 过程更少(想了解软件复杂度的衡量方法，请参考第 7.2 节)。

注意：

如果使用数据流图，软件实现会变得非常简单，因为在 DFG 模型中，由圆圈所表示的每

一个过程或者过程组都只有一个数据输入点和一个数据输出点。它用圆圈来表示每个过程的代码，用输入箭头表示数据输入，输出箭头表示数据输出，因此使编程任务得到了极大的简化。如果在 DFG 中输入的赋值是固定的，那么也称为无环 DFG(ADFG)。编程复杂性可以通过利用尽可能多的 DFG 和尽可能多的 ADFG 来最小化。

6.1.2 用于程序分析的控制数据流图的用法

以医生开处方推荐需要的药品以及服药的时间间隔为例，我们假设体温和听诊器检查的结果是输入。如果诊断过程用 DFG(每个数据输入都会产生一次数据输出，并且只存在一条程序路径)表示，那么就没有考虑到其他的情况，如发烧情况的变化等。该诊断过程应该考虑这些条件，然后每种情况下都必须推荐相应的处方。

控制流表示只有程序决定所有的程序执行步骤以及程序的流程。软件设计者都会编程预先决定这些步骤。如何设计一个过程来对进行数据操作时所作的决策以及进入程序的数据流进行协调控制呢？一个过程中可以有用于控制输入或者输出的语句，其中还可以有循环语句或者条件语句(回顾第 5.4.4 节)。经过每个控制条件的控制数据流之后，输入的数据会产生数据输出。CDFG 是一个图，它图形化地表示条件和沿着条件相关路径的程序流。CDFG 图还对过程中事件的影响进行了表示，并给出针对每一个特定的事件，哪一个过程被激活。这里，一个可变数值超出某个上界，低于某个下界或者位于某个范围之内，也可以作为激活某一过程的事件。

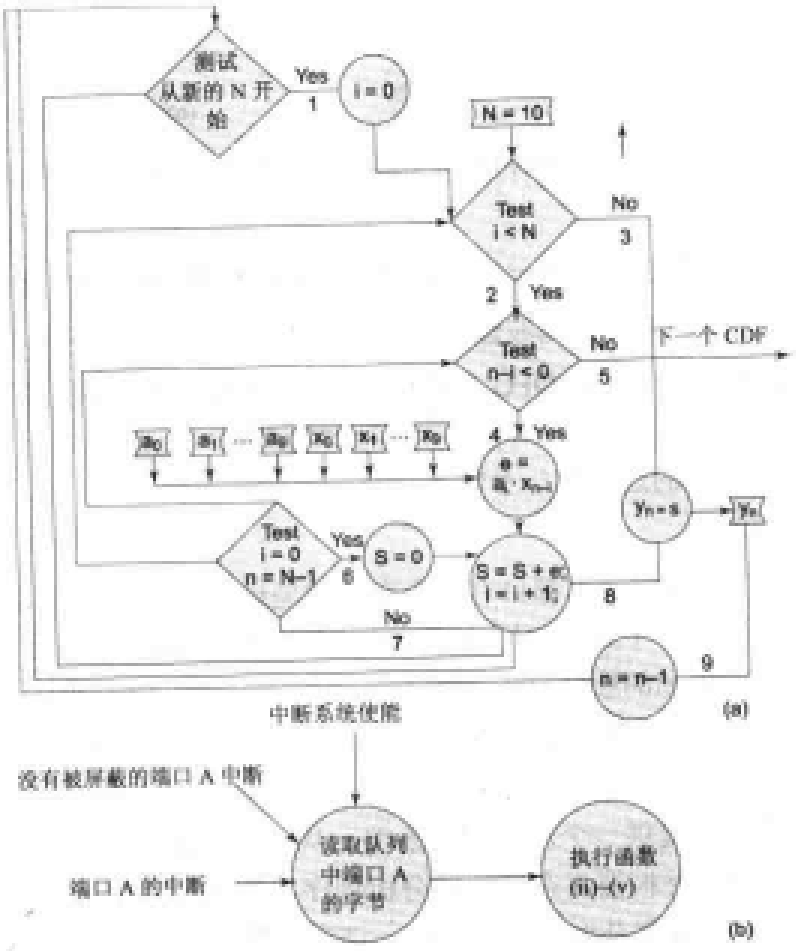


图 6-2 具有 10 个输入和 10 个系数的 FIR 滤波器的 CDFG 中，由测试框所表示的数据输入与控制输入(决策)结点；(b)示例 4-1 与示例 4-2 的 In_A_Out_B 程序中所标记的控制输入条件(条件标记在弧的起始处而不是在方框中)

考虑下面给出的例子。在 CDFG 中每个圆圈都表示一个过程。指向圆圈的有向箭头表示数据输入(或者一组输入)，从圆圈出发的有向箭头表示数据输出(或者一组输出)。方框(对角线是水平的或者垂直的正方形或者长方形)表示一个条件，示例参见图 6-2(a)。另外，条件也可以标记(或者表示)在有向弧或者箭头的起始处，示例参见图 6-2(b)。从方框出发的有向箭头或者做了标记的起始条件决定了条件为真时应该采取的动作。

示例 6-2

图 6-2(a)给出了通过指定测试条件的方框控制的输入(决策)结点，CDFG 对一个带有 10 个输入和 10 个系数的 FIR 滤波器进行数据输入(回顾一下示例 6-1 中第 n 个滤波后的输出序列 $y_n = \sum(a_i x_{n-i})$; 中各个术语的意思，其中求和是对 i=0, 1, 2, …, 9 进行的)。下面是计算 y_n 时应该注意的要点。用圆圈表示计算 y_n 的过程有一个输入点。

(1) y_6 有一个输出点。每个系数和每个滤波器输入只有一个存储器地址和变量。i、n、s 和 e 这些变量在程序流中可以取多个值。

(2) 获得输入的顺序和求和的顺序有关。图 6-2(b)介绍了示例 4-1 和示例 4-2 的 In_A_Out_B 程序如何控制输出的过程。这里，条件是在弧的起始处，而不是在方框中标记的。

在示例 6-2 中，请注意在计算 y_n 的过程中，复杂性增加了。CDFG 模型有助于理解所有的条件和决定程序应该存在的路径数。它还向我们展示了软件必须针对从决策点出发的每一条路径进行测试，并根据复杂性帮助分析程序(7.2 节)。

注意：

如果对条件、CDFG 中表示结点处控制决策的决策点，以及在决策之后从结点处顺序遍历的程序路径使用规格说明，软件实现就会变得简单。

6.2 用于事件控制或者响应时间受到约束的实时程序的编程模型

6.2.1 有限状态机模型

以洗衣机为例，在洗衣服的过程中，可以定义 3 个状态——“漂洗”、“清洗”以及“甩干”。在通电话的过程中，可以定义如下 5 五个状态，空闲状态、接收到响铃、拨号、连接以及交换消息(由于我们假设只考虑电话服务，所以没有提及到非功能状态)。同样，过程(函数、中断服务例程(ISR)、任务或者线程)的状态也是可以定义的。

什么时候可以将过程建模为状态呢？通常，过程都存在一些输入，它们可以将该过程的状态转变为新的状态，并产生输出，而这些输出也可以是下一个状态的输入。现在假设在模型中，过程(程序流)的运行可以被认为是机器产生状态的过程。程序流可以简单地通过内部状态转换(从一个状态到另一个状态)进行建模。

建模，或者说表示状态和内部状态转换的步骤如下所示。

(1) 到新状态的转换是从某个事件(输入)的前一状态开始进行的。该事件可以设置某个参数的数值或者某段代码执行的结果。转换也可以由中断标识驱动(某个标识被设置以后)、由信号量驱动或者由中断源服务需求驱动。

(2) 状态由某个标识条件、一组正在执行的代码或者某些参数的一组数值进行标识。

(3) 状态可以从另一个状态接收多个令牌(输入、消息、标识中断或者信号量)。令牌在这里可以看作一个一般的术语，表示一个输入或者事件输入。事件输入的特征是异步的(从来都不知道某个事件会在何时发生)。当对某个标识进行设置或者重置时就可能引发一个事件输入。事件输入可能发生，当(i)发出信号量或者获得信号量时；(ii)出现产生资源、信号以及数据项的迹象时；(iii)一组代码执行完毕(信号量和信号的意思请分别参考第 8.1.2 节和 8.3.1 节)。

(4) 一个状态可以产生多个令牌(输出、消息、中断标识或者信号量)。

用圆圈表示状态，用有向弧(或者箭头)表示从一个状态到另一个状态的程序流。当要将某个过程建模为有限状态机时，设计者会为每个状态进行如下的指定：

(1) 能够使其到达该状态的有限集合的输入数值。

(2) 该状态的有限行为(例如计算)。

(3) 有限集合的输出与它们可能的数值(或者令牌、事件标识或者状态标识)，以及为该状态给出输出的输出函数。

(4) 使得每个状态转变为下一个状态的状态转换函数。

注意：

如果使用 FSM，那么实时系统软件实现就会变得非常简单。编程任务精简为如下所示。(i)为每个状态转换函数和输出函数编码；(ii)在进行实时编程的时候，应该了解该过程在每个状态过转换函数中以及每个状态之间所花费的时间周期。

考虑某个定时器处于运行状态的例子。计数输入是时钟输入。被改变的计数值是输出。输出函数是对计数值的增加。当到达预定数目的计数输入时，状态转换函数是发生上溢时的超时。

为了便于理解 FSM 模型表示，图 6-3(a)给出了定时器的状态。定时器具有 4 个有限状态："空闲"、"起始"、"运行"以及"完成"。

(1) "空闲"状态：一旦接收到输入，它就启动状态转换，时钟周期计数(定时器完成之后的时钟周期数)和向"起始状态"的转换就开始进行。

(2) "运行"状态：每来一个时钟输入，计数值就减 1。

(3) "完成"状态：到完成状态的程序流。当计数值为 0 时到达该状态。

图 6-3(b)介绍了如何对一个 C 函数进行状态建模。该函数具有 4 个有限状态。"空闲"、"调用"、"执行"以及"返回"。

(1) 从"空闲"状态到"调用"状态的转换发生在程序中出现函数调用时(请参考第 5.4.6 节)。来自该状态的输出保存在堆栈中。

(2) 在"执行"状态时，指令执行，且程序计数器随着程序流发生改变。

(3) 当指令到达返回指令时，就会发生向"返回"状态的转换。

(4) 从堆栈中取回保存的状态和数值之后，就开始向“空闲”状态转换。

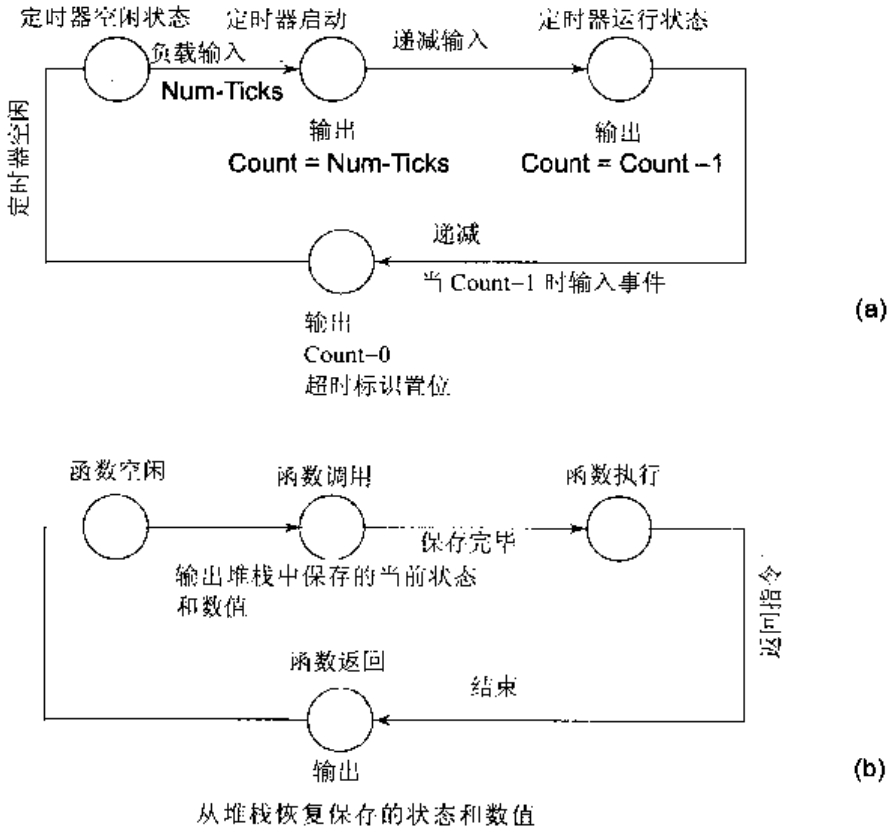


图 6-3 (a)使用 FSM 的一个定时器的状态；(b)使用 FSM 的一个 C 函数的状态

图 6-4(a)介绍了如何像 FSM 中的状态那样，对 ISR 的状态进行建模。ISR 具有 5 个有限状态：“空闲”、“调用”、“执行”、“阻塞”以及“返回”。

(1) 当发生中断或者某个状态标识被置位时，就开始从“空闲”状态向“调用”状态的转换。来自新状态的输出由如下两部分组成(i)将上下文保存到某个堆栈中的行为；(ii)重置状态标识。

(2) 指令在“执行”状态中执行，且程序计数器随着程序流发生改变。

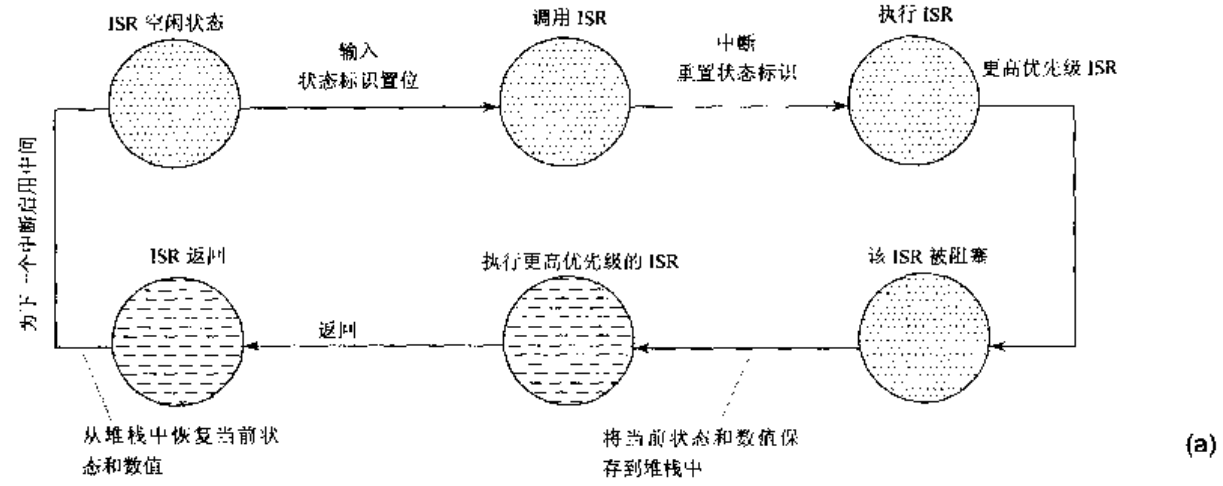
(3) 当出现优先级更高的中断时，事件就会触发到“阻塞”状态的转换。

(4) 当被阻塞的 ISR 指令再次开始执行，并到达返回指令的时候，程序就到达“返回”状态。

(5) 从堆栈中恢复上下文之后，程序流进入空闲状态。

图 6-4(b)介绍了如何将一个任务的状态建模为 FSM(要理解任务的概念，请参考第 8.1 节)。任务具有 5 个有限状态——“空闲”状态、“就绪”状态、“运行”状态、“阻塞”状态以及“完成”状态。对于一个状态的输出来说，它将成为下一个状态的输入，从调度程序来的令牌是就绪标识和阻塞标识。而发送给调度程序的标识是运行标识、阻塞标识以及完成标识。

(1) 当 RTOS 发送一个令牌(消息)来调度该任务的时候，该任务开始从“空闲”状态向“就绪”状态转换。该状态的输出是将调度程序的上下文保存到调度程序的堆栈当中。



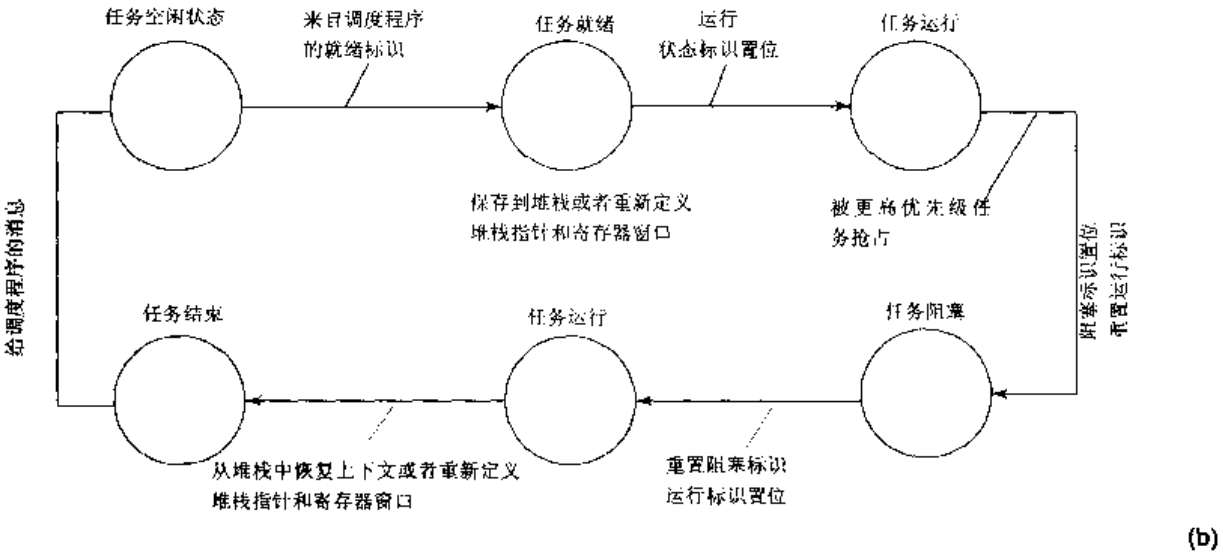


图 6-4　(a)ISR 在 FSM 中的状态；(b)多任务调度中某个任务在 FSM 中的状态

(2) “运行”状态使得指令执行，程序计数器随着程序流不断变化。

(3) 当调度程序抢占了某个任务时，程序流进入“阻塞”状态。它向该任务发送一个令牌(消息)。该状态的输出是将任务的上下文保存到堆栈当中。

(4) 当从调度程序接收到一个令牌，且从堆栈恢复上下文之后，程序流再一次进入“运行”状态。

(5) 当指令到达结束阶段时，程序流进入“完成”状态。输出一条给调度程序的消息。

(6) 当任务处于有限等待循环状态时，程序流进入“就绪”状态。

(7) 当该任务向调度程序发送了一个消息，且该任务从就绪列表中删除后，程序流进入“空闲”状态。

注意：

FSM 模型在某个时刻只适合于一个过程，适合于从一个状态到下一个状态的顺序流，适合

于该程序的受控流。FSM 有助于简化对洗衣、定时器、函数、一组嵌套调用的函数、ISR 或者任务等一系列过程的状态进行建模。

当 FSM 模型用圆圈和有向弧进行图形化表示的时候，在遇到一个具有大量状态的复杂过程时，问题就变得复杂了。如果要使用模型来设计软件，那么可以设计一个状态表，用它的行来表示每一个状态。而每一行分下面的列：

(1) 提供状态名或者标识名；
(2) 发生某些事件时，该状态中的行为；
(3) 引起状态转换函数执行的事件(令牌)；
(4) 从该状态的输出函数产生的输出；
(5) 下一状态；
(6) 期望在事件发生之后，到新状态的转换结束所需要的时间间隔。

使用这些行就可以像下面这样简单地编码了。

```
while (presentState) {action ( ); if (event = .....; token = .... ) {output
= ......; stateTransitionFunction ( ); };}
```

这里 presentState 是一个布尔变量，只要当前状态继续，其值就为真，一旦发生了到下一个状态的转换，其值就会变成假。action()是在该状态中执行的一个函数。如果发生了某个事件，并接收到令牌(例如，定时器中的时钟输入)，状态转换函数 stateTransitionFunction 就执行，且使得 presentState = false，通过设置 nextState(一个布尔变量)=true 进行到下一个状态的转换。

注意：

如果使用 FSM 模型，那么在编写代码的时候用状态表来表示就变得得心应手了。

6.2.2 Petri 网模型

1. Petri 网

Petri 网已经被认为是对实时嵌入式系统进行建模的强有力的工具之一。Petri 网是由 C.A. Petri 开发的一种图形建模工具(请参考 Tadao Murata on Petri Nets Properties, Analysis and Applications, Proceedings of IEEE, 77, 541-580, 1989)。在为控制电路、计算和通信设备设计算法的过程中，已经用到了基于 Petri 网的建模。下面的 4 个例子、图 6-5 以及图 6-6 都清晰地说明了 Petri 网模型的用法。

(1) 与 FSM 模型中的状态相对应的是，在每一个网中都有 Petri 网模型中的结点位置和转换。计算机网络是多种结点的互联：计算机、服务器、打印机、集线器、交换机以及路由器。Petri 网互联有两种类型的结点：结点位置和结点转换。

(2) 在一个图形化表示的 Petri 网当中，圆圈表示结点位置。结点转换用矩形表示(图 6-5 与图 6-6)。

(3) 有一个位置称为标记位置。它表示了一个类似于 FSM 中的初始状态或者图 6-5 与图 6-6 中“空闲”位置的初始标记。

(4) 有向弧(箭头)位于结点位置与结点转换之间，或者位于结点转换到下一个结点位置之间。

(5) Petri 网是使用有向图表示的。这些图都是双向的(双向的意思是具有两个方向。注意图 6-3 与图 6-5 之间或者图 6-4 与图 6-6 之间的差别)。

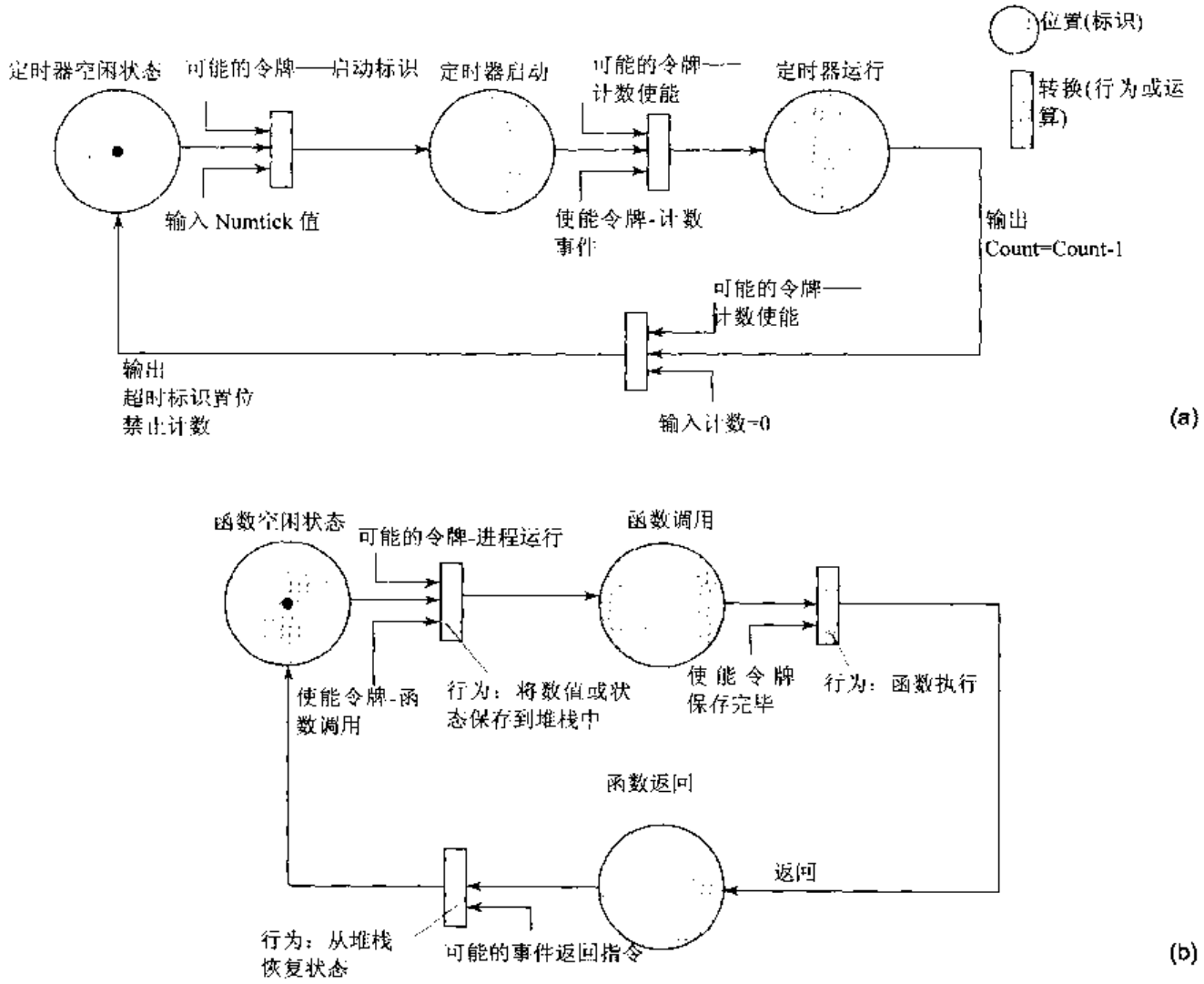


图 6-5 (a)一个定时器的 Petri 网模型；(b) 一个函数的 Petri 网模型(注意每一个转换处都只有一条有向弧)

图 6-5(a)给出了一个定时器的 Petri 网模型中的结点位置和结点转换。该定时器初始时处于“空闲”结点位置。存在一个令牌，即通过触发转换可以启动该定时器的启动标识。当接收到一个使能令牌(时钟周期数，numTick)的时候，转换就会被触发(触发的意思是离开某个结点位置，重新开始，它导致某种行为、计算或者程序流)。当某个转换被触发时，定时器进入“启动”结点位置。(a)有一种令牌，即计数使能(count-enable)标识，它通过触发向“运行”结点位置的转换来运行定时器。(b)还有一种使能令牌，即通过触发向“运行”结点位置的转换来运行定时器的计数事件(count-event)，减少了 count 的数值以后，输入就为 count。它使能一个转换。然后，一旦接收到这种令牌，即计数使能和使能令牌，并且 count=0，转换就被触发到“空闲”位置。

图 6-5(b)展示了某个函数的 Petri 网模型中的结点位置和转换。函数被调用之前，它处于“空闲”结点位置。有一种可能的令牌，即过程运行。有一种使能令牌 call，它通过触发一个从“空闲”位置开始的转换来启动该函数。在转换处，发生的行为是将处理器的状态保存到堆栈当中。此后，流程转向函数调用的位置。从这个位置开始，处理器发出保存完毕(save-over)令牌之后，转换被触发。程序计数器根据程序流发生改变。产生下一个位置的输出令牌。转换被触发之后，该函数进入“返回”结点位置。有一种可能的令牌，即返回指令。一旦触发了某个转换，就会恢复保存的状态。然后，程序流转向“空闲”位置。

图 6-6(a)给出了某个 ISR 的 Petri 网模型结点位置和结点转换。发生中断之前，该 ISR 处于“空闲”结点位置。存在一种可能的令牌，即中断使能。存在一种使能令牌，即中断标识，它通过从该“空闲”结点位置触发一个转换来启动 ISR。在转换位置处进行的行为是将处理器状态保存到堆栈当中。此后，程序流转向 ISR 调用结点位置。在来自处理器的使能 ISR 的 save-over 令牌到达之后，有一个转换从该位置处触发。在转换位置处，进行的行为是按照每条 ISR 指令进行计算。程序计数器随着程序流发生改变。然后为下一个位置产生输出令牌。ISR 转向“运行”位置。可能的令牌 p 可以是触发某个转换的输入。当另一个 ISR 被使能的时候，p 是一条 ISR 阻塞消息。在使能令牌，即“使能更高优先级的 ISR”之后，转换被触发。所进行的行为是保存上下文和执行更高优先级的 ISR。而正在运行的 ISR 发生阻塞。来自于正在运行的更高优先级 ISR 的“返回事件”使能令牌触发转换之后，ISR 再一次转向“运行”位置。在返回指令之后有一个使能令牌。一旦触发了某个转换，所导致的行为就是上下文切换。然后程序流转向 ISR 原来的“空闲”位置。

图 6-6(b)给出了某个任务的 Petri 网模型位置和转换(第 8.1.2 节)。在发生中断之前，任务处于“空闲”结点位置。存在一个可能的使能令牌，即任务就绪标识。在进行转换的时候，所进行的行为是将处理器状态(上下文)保存到堆栈当中。此后，程序流转向任务“就绪”位置。在可能的使能令牌(事件)，即运行之后，转换在该位置处触发。如果在某个任务执行期间出现了错误，就会产生输出令牌，即错误标识。任务转向“运行”位置。转换时所进行的行为是按照每条任务指令进行计算。程序计数器随着程序流发生改变，并产生下一个位置的输出令牌。下一个位置可能是 IPC 消息队列(参考第 8.3.3 节)。该任务转向“运行”位置。可能的使能令牌是 p。当另一个任务从 RTOS 位置处使能的时候，p 是任务的阻塞消息。导致的行为是保存上下文以及阻塞正在运行的更高优先级任务的执行。正在运行的任务发生阻塞。任务转向“阻塞”任务位置。一旦出现某个来自正在运行的较高优先级任务的“返回事件”，转换会被来自 RTOS 的使能令牌触发。此后，RTOS 会再一次将该任务转向某个转换。在进行转换时，会执行该任务余下的代码，该任务转向“运行”位置。在触发转换的返回指令之后会产生一个使能令牌 r，所进行的行为是上下文切换。然后，程序流转向该任务原来的“空闲”位置。

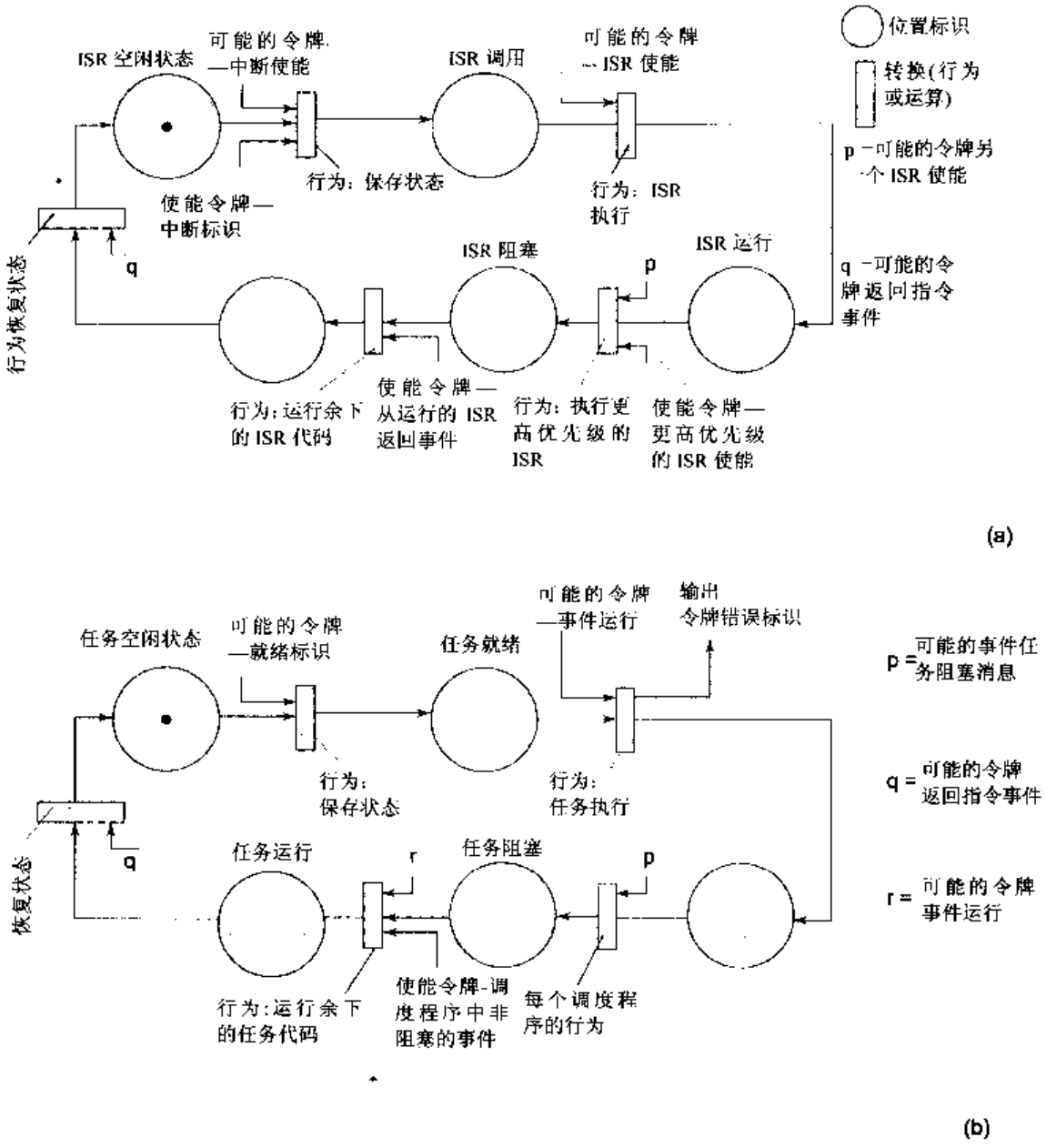


图 6-6　(a)一个 ISR 的 Petri 网模型；(b)一个任务的 Petri 网模型

注意：

有向弧展示了从转换到位置或者从位置到转换的程序流。这些行为(例如，计算)在转换处发生(在 FSM 中行为发生在状态中)。位置表示 Petri 网中某个特定的结点。从该结点开始到转换所指向的位置进行的转换只有在该转换从该(这些)位置处收集到(接收到)了所有的使能令牌之后才能够触发。存在指向该转换并使能该转换的可能的令牌。这样，转换不需要只从一个位置处接收触发它的令牌。触发就表示这些行为(例如，计算)。输出令牌从该转换处产生，它可以指向多个结点位置。

对 ISR、任务以及调度程序的编程具有很大帮助的两个模型是 FSM 和 Petri 网。FSM 在状态数目有限的时候使用，而 Petri 网在状态数目不明确的情况下使用。在对一个 RTOS 的调度程序编写代码的时候，Petri 网比 FSM 有用得多。

注意：

如果可以从几个状态中的任意一个状态转向另一个状态，那么使用 Petri 网可以极大地简化实时系统软件实现。要明确知道哪一个状态会转向哪一个编程是不可能的。编程任务可以简化为如下两个部分。(i)在每一个转换处为每个行为(函数或者运算集)编码；(ii)知道行为发生的时间和每个结点转换函数的处理过程以及每个触发的转换之间的时间间隔。

2. Petri 网的一个特例 FSM

当每个节点转换处从一个位置和一条输出弧只存在一条有向输入弧的时候，就得到了有限状态机的等效。有限状态机可以认为是 Petri 网的一个子集。早期的时候，洗衣机被假想为具有以下三个状态的有限状态机：洗涤、清洗和甩干。当将洗衣机建模为 Petri 网的时候，认为洗涤状态是由结点转换、洗涤、结点位置以及洗涤结束组成的。当该转换被触发的时候，所进行的行为是洗涤。有一条有向输入弧指向洗涤结束位置。从洗涤结束到下一个结点转换——清洗，只存在一条有向弧。

来自 FSM 的过程流中的状态可以转向后续的几个可能状态中的任意一个。这取决于该状态的输出函数和状态转换函数。可以存在从一个状态出发的多个有向图。但是从一个有限状态到某个状态只能有一个有向图。可以存在多个结点转换与 Petri 网中的结点位置相关联。在某个结点转换处，首先寻找可能的令牌(信号量或者中断使能位、中断标识设置)，然后再寻找使能令牌。只有在接收到使能令牌之后，转换才会被触发。

3. 实时编程中 Petri 表的用法

在一个进程或者一个多任务调度程序当中可以存在多个 Petri 网(进程与任务的定义请参考第 8.1 节)。Petri 网模型可以同时考虑多个进程。从结点位置到结点转换以及从结点转换到结点位置可能存在动态转换。可以有彩色的 Petri 网。彩色的 Petri 网的意思是在该转换中存在多种类型的令牌。

如果存在大量的转换和结点位置，大量的彩色 Petri 网和动态(时间不可预知)转换，那么图形描述就会变得非常复杂。在存在多个 ISR，且多个任务共享同一资源的 RTOS(实时操作系统)等复杂的系统当中，表格表示法比图形表示法更有效(资源是指 CPU、变量、存储器、文件、IO 设备)。

在进行实时编程的时候，Petri 表中行的使用为简化代码编写提供了强有力的建模技巧。在 Petri 表中，每一行都定义了下面的列来表示每一对结点位置和结点转换。

(1) 提供结点位置名，用与之相关的命名结点转换(输入或者指令集的执行)来标识条件。

(2) 结点位置处的可能事件(令牌)。

(3) 触发转换的事件(使能令牌)。

(4) 输出(令牌)。

(5) 转换处的输出函数和行为。

(6) 下一个结点位置。

(7) 期望完成转换的时间间隔。

使用 Petri 表的优点可以从如下所示的编程方法中得以理解。考虑为从当前结点位置出发到下一结点位置的进程流建立的 Petri 表。该例中代码的编写是基于 Petri 表进行的，如表 6-1 所示。对这里考虑的进程流作如下的假设。

表 6-1 从当前结点位置出发到下一结点位置的进程流的 Petri 表

结点位置	可能的令牌	结点转换	使能令牌	输出令牌	输出行为	下一结点位置	允许的时间
Atpresent Node-place	flagE1、inputE1、semaphoreE1 和 eventE1 flagE2、inputE2、semaphoreE2 和 eventE2	Transition1	flagE1, inputE1, semaphoreE1, eventE1	来自函数 output1()的 error1	Output-Actions1()	在下一个结点位置 1	t1"
Atpresent Node-place	同上	Transition2	flagE2, inputE2, semaphoreE2, eventE2	来自函数 output2()的 error2	Output-Actions1()	下一结点位置 2	t1"

(1) 假设存在能够触发 transition1 的可能令牌。它们是 flagE1、 inputE1、 semaphoreE1，以及 eventE1。

(2) 假设存在能够触发 transition2 的可能的令牌。它们是 flagE2、 inputE2、 semaphoreE2，以及 eventE2。

(3) 假设当 transition 1 被触发的时候，布尔变量 transition1 等于真，当 transition 2 被触发的时候，布尔变量 *transition2* 等于真。

(4) 当 transition1 被触发的时候，假设所进行的操作是函数 outputActions1 ( )所进行的运算。当 transition2 被触发的时候，假设所进行的操作是像函数 outputActions2 ( )那样的运算。

(5) 当转换 1 被触发的时候，假设输出令牌与函数 output1 ( )相同。令牌是 error1。当 transition1 被触发的时候，假设输出令牌与函数 output1 ( )相同。令牌是 error2。

(6) 假设某个 Petri 网的当前结点位置用布尔变量 atpresentNodePlace 表示，只要进程处于当前位置，它就等于真。只要 atpresentNodePlace 不等于假值，该进程就处于该位置。

(7) 假设在某个结点位置处花费的时间为 T。T 是 t' = timeatpresentNodePlace。假设 t"1 是允许当前结点位置结束的最大运行时间。假设 t"1 = aTimeInterval。只要 t' < t"1，进程就保持在当前结点位置(不会发生超时)。

(8) 假设直到 transition1 或者 transition2 中有一个被触发，变量 atpresentNodePlace 才会取值为假。

(9) 假设转换是到下一个结点位置 1。布尔变量 atpresentNodePlace1 表示下一个结点位置 1，并且在 transition1 转到下一位置之后取值为真。假设转换是转向下一个结点位置 2。布尔变量 atnextNodePlace2 表示下一结点位置 2，在 transition2 转到下一个位置以后它取值为真。

在根据该表编写代码时，第 1 个 while 循环将用于在第 1 列的某个位置处等待，直到第 3 列的转换被触发或者发生超时。第 2 个 while 循环直到第 2 列的可能令牌是来自某个结点位置的输入。一旦接收到第 4 列的输入令牌，转换就会被触发。如果输出令牌的条件语句与第 5 列

的相同，且输出行为与第 6 列的相同，那么使用第 1 个。下一结点位置与第 7 列相同。在经过第 8 列中规定的时间间隔之后会发生超时。

示例 6-3

```
While (atpresentNodePlace && (timeatpresentNodePlace < aTimeInterval)) {
/* A set of possible tokens is at a place as per column 2 of table. Wait for
a possible token / possibleToken = (flagE1 == true || inputE1 == .input1 ||
semaphoreE1 - = taken || eventE1 = true || flagE2 == .true || inputE2 == .input2
|| semaphoreE2 - = taken || eventE2 = true);
while (possibleToken) {/* Possible Tokens Expected */
/* Wait for a set of the enabling tokens atPresentNodePlace as per column 4 row
1 or row 2*/
/* Transition 1 fires as per enabling token set in row 1 column 4. */
transition1 = (flagE1 == true) & & (inputE1 == input1) &&(semaphoreE1 == taken)
&& ( eventE1 = = true);
/* Transition 2 fires as per enabling token set in row 2 column 4. */
transition2 = ( flagE2 == true) & & (inputE2 == input2) &&(semaphoreE2 == taken)
&& ( eventE2 = = true);
}; /* End of all codes for the present NodePlace possible tokens */
/* If transition1 = true then transition 1 output tokens as per row 1 column
5, actions as per row 1 column 6 and flow to the next node place as per row 1
column 7. */
if ((atpresentNodePlace) && transition1)) {
output1 ( ); OutputActions1 ( ); atnextNodePlace1 = true; atpresentNodePlace
= false;};
/* If transition2 = true, transition 2 output tokens as per row 2 column 5, actions
as per row 2
column 6 and flow to the next node place as per row 2 column 7. */
if ((atpresentNodePlace) && transition2)) {
output2 ( ); OutputActions2 ( ); atnextNodePlace2 = true; atpresentNodePlace
= false;};
/* End of all codes for two sets of input tokens in two row of the table. These
enabled the output of new tokens and output functions and a transition to another
NodePlace */
}; /* End of while loop for present place. Now at the next place*/
```

注意：

在使用 Petri 网模型的时候，对于编写代码来说，“Petri 表”的表格表示法是非常有用的。

6.3 多处理器系统的建模

多处理器嵌入式系统在设计复杂的嵌入式系统时有所应用(参考第 1.2.7 节)。

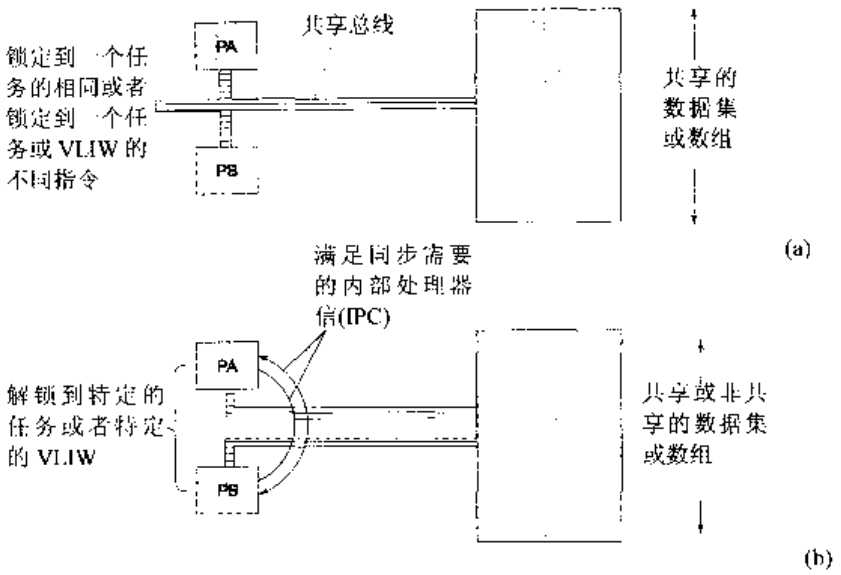


图 6-7 (a)在处理多任务的时候共享相同地址空间的紧密耦合处理器；(b)不仅具有数据集和数组的共享地址空间，而且在网络中具有分离自治的地址空间的松散耦合处理器

首先，在一个系统当中，两个处理器 PA 与 PB 如何与存储器接口？情况 1：处理器通过公共总线共享相同的地址空间，称为处理器之间的紧密耦合。情况 2：处理器不仅共享数据集和数组，而且还具有不同的自治地址空间(像在网络中一样)，称为松散耦合。图 6-7(a)与(b)给出了这两种情况。情况 3：处理器以一种总线体系结构，例如，三维网络、环、带或者树来共享存储器，而不是通过在紧密耦合的不同处理器之间共享总线来共享存储器。那么，当前的处理器如何进行处理呢？

(1) 目前进行处理的一种方法是对每个任务进行调度，使它在不同的处理器上执行，通过某些内部处理器的通信机制来同步任务。

(2) 第 2 种方法是，如果一条 SIMD、MIMD 或者 VLIW 指令具有不同的数据(例如，示例 6-2 中的系数不同)，每个任务针对不同的数据在不同的处理器(紧密耦合的处理)上运行。这与在 TMS320C6 上执行 VLIW 相似。它使用了两组相同的 4 个单元，VLIW 指令字可以处于 4~32 字节之间。如果编译器进行调度使得处理器将不同的指令元素分派到不同的单元并行执行，那么它具有指令集并行性。注意：编译器对 VLIW 进行静态调度。静态调度(第 6.3.6 节)就是编译器进行编译，使得代码在每次调度所决定的不同处理器或者处理单元上运行，即使某个处理器在等待其他的处理器完成调度处理，这种调度在程序运行的过程中也会始终保持静态。

(3) 还有一种可选的方法是任务指令在同样的处理器上执行，但是同一个任务的不同指令可以在不同的处理器(松散耦合的)上执行。编译器调度在不同的处理器之间调度该任务的不同指令。

注意：

多处理器系统的存储器可以是紧密耦合的或者松散耦合的。它也可以组织为网、环、带或者树结构。有几种在系统中进行调度和同步执行指令的方法，如 SIMD、MIMD 以及 VLIW。

6.3.1 多处理器系统中的问题

如何在多处理器系统中进行编程？问题在于如何将该程序划分为不同处理器之间的任务

或者指令集，然后针对给定的处理器时间和资源如何调度指令和数据才能得到优化的性能。应该对在处理器上运行一个任务进行调度吗？那么，假设一个处理器比另一个处理器先完成任务，性能代价是什么(第 6.3.6 节)？如果所获得的处理器时间还有空闲，性能代价会更高。如果一个任务需要向另一个任务发送一条消息，并且另一个处理器要等待直到接收到这条消息为止，那么性能代价又是什么？下面列出了在多处理器系统当中对指令的处理过程进行建模所存在的问题：

(1) 进程、指令集和指令的划分。

(2) 针对每个进程中的 SIMD、MIMD 以及 VLIW 指令进行的调度和为每个处理器调度它们。

(3) 对每个处理器上的进程进行的并发处理。

(4) 处理器的每个超标量单元和流水线上的并发处理(要了解流水线调度和超标量处理请参考第 2.1 节)。

(5) 编译器的静态调度，类似于超标量处理器中的调度。

(6) 硬件调度，例如，硬件(处理器和存储器)的静态调度是否可行(如果不影响系统的性能，这样会更简单，并且它的使用还要依赖于指令的类型)。

(7) 静态调度问题(例如，什么时候性能不会受到影响，什么时候处理行为是可预测的和同步的)。

(8) 同步问题，同步的意思是使用内部处理器或者进程通信，使得多处理器系统中任意处理器中的运算按照某种确定的顺序触发。

(9) 动态调度问题(例如，什么时候性能会受到影响，什么时候存在中断，什么时候对任务的服务是异步的。什么时候会有一个抢占式调度也与此相关，因为它也是异步的)。

注意：

在一个多处理器系统当中，当进行并发处理和指令(如 SIMD、MIMD 以及 VLIW)的调度时，有很多问题都需要检验。

6.3.2 模型

这里给出了有助于编程和调度系统的 4 个模型。在理解这些模型之前，应该先理解这些模型中所使用的有向图。这些有向图的表示类似于 DFG。(第 6.1 节)图中的结点(圆圈)称为参与者(actor)。参与者进行运算。参与者本身还可以表示完整的 DFG。结点之间的边(带一个箭头表示方向的弧线)表示来自某个结点的输出值队列和传给另一个结点的输入值队列。边将数值从一个参与者传递给另一个参与者。

假设 X 和 Y 是两个曾经被触发(启动)的指令的集合。在进行运算的时候，它们不需要来自任何源的输入。假设 X 产生输出数值(令牌/数据)a、b、c。假设 Y 接收到输出数值(令牌/数据)a、c、I 和 j，其中 j 有一个延迟。Y 的输入个数不一定要等于 X 的输出个数。Y 可以得到额外的输入，不一定全都要来自于 X 的输出。此处，这些运算和数据是通过从 X 到 Y 的有向数据流图进行建模的。输出和输入的数目标记在弧的起点处和终点处。参与者等效于 Petri 网模型中的转换，边等效于其中的位置。图 6-8(a)给出了 X 与 Y 之间的有向图中的参与者和弧。它还展示了来自 X 的一组输出(a、b 和 c)和到 Y 的输入(a、c、I 和 j)。图 6-8(b)用 Petri 网(第 6.2.2 节)展示了与之相似的模型。

注意：

多处理器系统的运算和触发实例都可以建模。建模可以简化处理器的编程、调度和同步。

6.3.3 同步数据流图模型

同步数据流图(SDFG)模型如下所示(请参考 E. A. Lee 与 D. G. Messerschmitt 共同编著的 *Static scheduling of synchronous data flow*，IEEE Transactions on Computers, Feb. 1987)。假设弧表示物理存储器中的缓冲。弧中可以包含一个或者多个带有一定延迟的初始令牌。直到该结点接收到令牌以后，令牌才能触发结点处的运算。这样，初始的令牌也可以表示由 SDFG 的边上的点所展示的延迟。(图 6-8(a))例如，i 和 j 是图 6-8(a)和(b)中结点 Y 的初始令牌，其中，i 有延迟。弧上的点表示 SDFG 模型中的初始令牌。如果存在多个初始令牌，那么初始令牌的数目会在点上标出(编译器随后将按照 SDFG 静态调度。考虑令牌和初始令牌的数目，编译器在每个结点处调度执行(触发运算和创建另一个输出令牌的集合))。

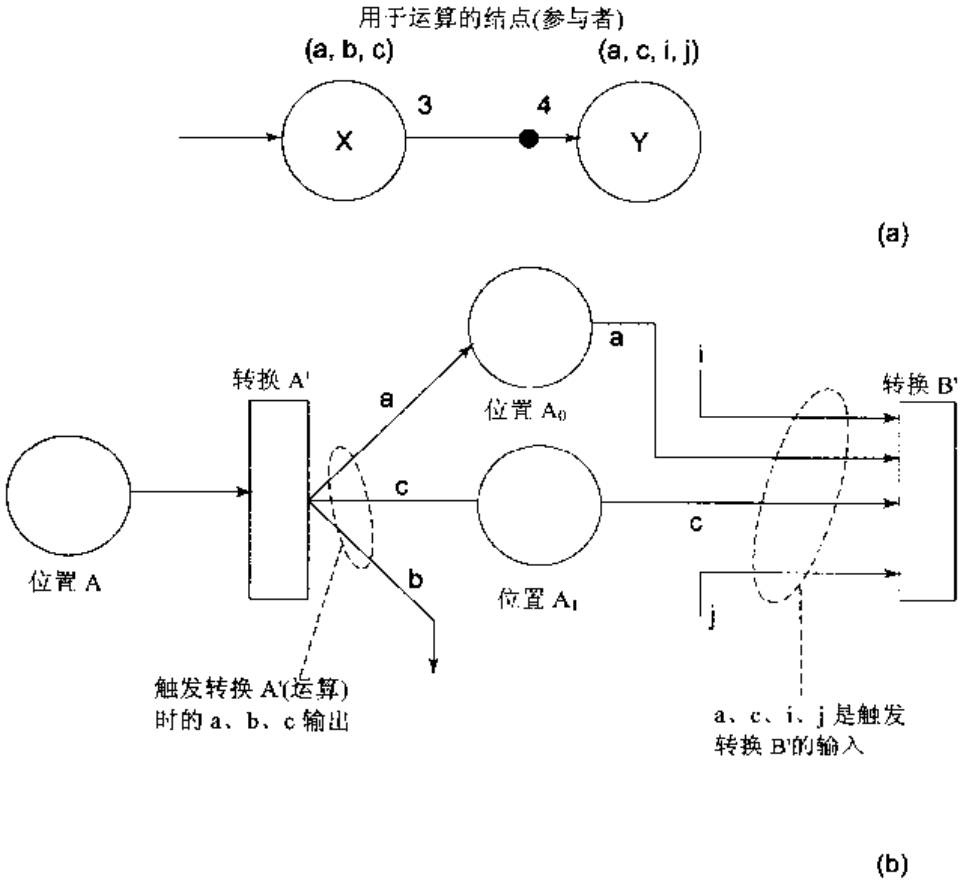


图 6-8 (a)X 与 Y 之间有向图中的参与者和弧。还展示了输出 a、b、c 以及输入 a、c、i、j。i 带有延迟(用点标示)；(b)使用转换(参与者)和位置(边)Petri 网对比模型

注意：

SDFG 模型类似于 DFG，但是它不仅要对输入和输出的数目建模，还要对延迟进行建模。

6.3.4　同构的同步数据流图模型

如果在输入处只有一个令牌，且在输出处也只有一个令牌，那么 SDFG 就是一个近似的 SDFG(HSDFG)。例如，如果来自结点 X_1 的输出是 a，而到 Y(另一个运算集)的输入也是 a。因此，SDFG 可以展开为 HSDFG。一个 SDF 图可以展开为一个或者多个 HSDFG。两个结点可以通过 HSDF 图中的两条或者多条边进行连接。HSDF 图自然会比 SDFG 具有更多的结点或者边，因为每个结点处只允许一个令牌。图 6-9(a)、(b)和(c)给出了 SDFG 相应地展开为一个带有 3 个结点和 2 条边的 HSDFG 图，并在删除延迟后得到了相关的 APEG。

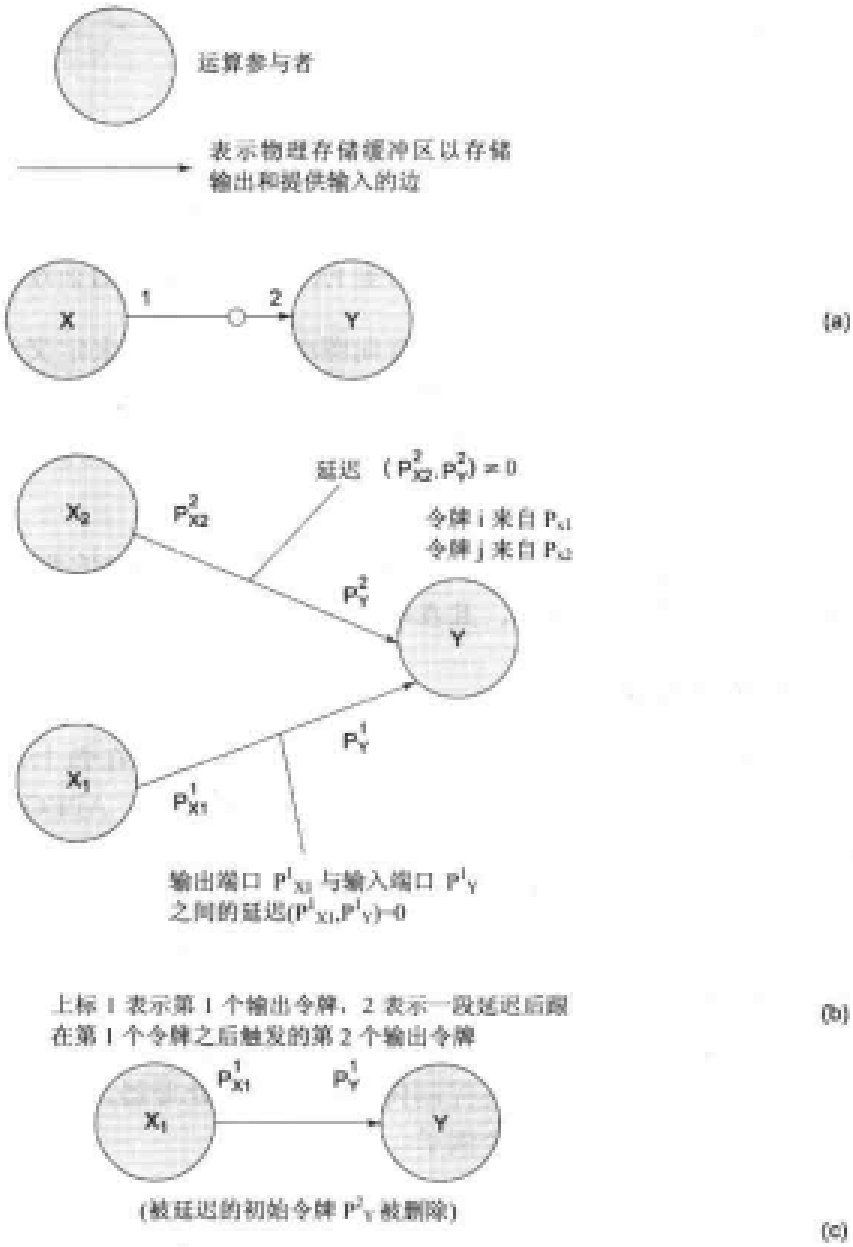


图 6-9　(a)用 SDFG 表示的运算的建模，2 上的点和标记表示结点 Y 处延迟的两个数字输入令牌；(b)展开 SDFG 后得到的 HSDFG 表示；(c)删除了延迟边之后的 HSDF 的 APEG 表示

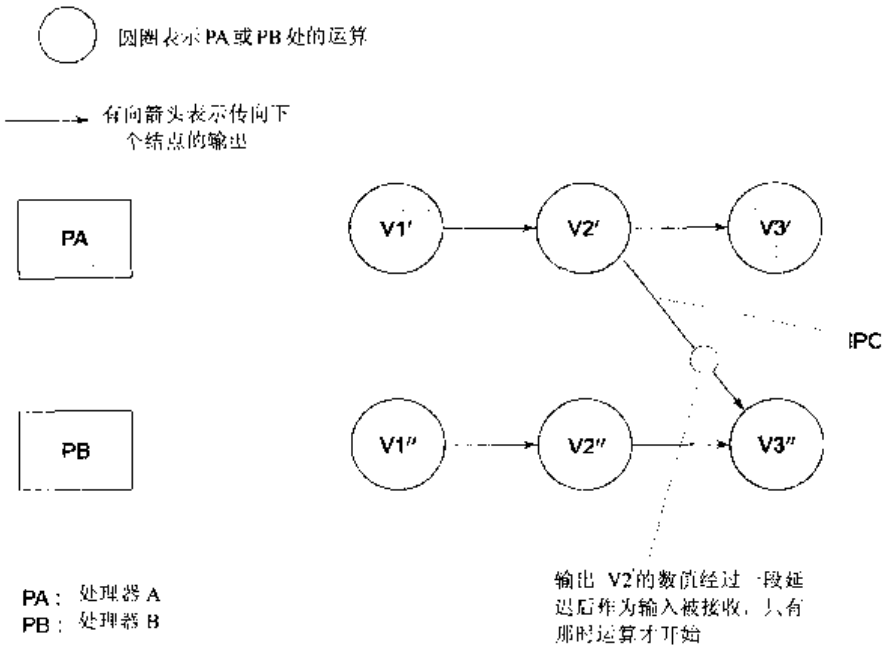


图 6-10　带有一个 APEG 和一个带有从 PB 到 PA 的 IPC 的 HSDFG 的双处理器系统

假设有一个在结点处触发的运算序列。用有向图中的优先顺序来定义运算顺序，按照该顺序放置第 1 个结点，第 2 个结点，然后一个接一个地放置。在处理器组中的一个处理器上的序列可以在弧上被延迟。来自于另一个处理器的输入(初始令牌)也可以被延迟。

注意：

HSDFG 模型与 SDFG 相似，但是具有这样的特征，沿着一条边(弧或者箭头)延迟的只有一个令牌，因为在输入处只存在一个令牌，且在输出处也只存在一个令牌。

6.3.5　无环优先扩展图模型

无环优先(acrylic precedence)是指有向图中结点的优先，使得在弧上没有延迟。从 HSDF 中去掉初始令牌(延迟)就可以得到一个无环优先扩展图(APEG)。APEG 的重要性是什么？APEG 不仅沿着弧有与来自前一个结点的输出相同的起始输入，而且没有延迟令牌。因此，沿着弧执行是平滑的，不需要内部处理器通信时间。基于 APEG 的算法最容易调度，这使得算法中的优先级约束和以前的相同。图 6-9(c)给出了一个相关的 APEG，它是一个没有延迟的图。它是从 HSDFG 或者 SDFG 中派生来的。

IPC 图和任务级并发进程都可以用 APEG 和 HSDFG 来建模。在建模为 APEG 的处理器上运行的线程可以通过阻塞自己或者休眠将控制传递给另一个处理器，但是沿着该 APEG 的序列和处理流始终保持完整。下面给出了一个例子。

示例 6-4

假设 V'_1、V'_2 以及 V'_3 是分配给处理器 PA 的运算结点。假设 V''_1、V''_2 以及 V''_3 是分配给与 PA 并行工作的处理器 PB 的运算结点。如果算法(或者运算集)V''_3 只有在接收到来自 V'_2 的一条消息(令牌)以后才能继续，那么就需要 IPC。假设 IPC 发生在 V''_3 与 V'_2 之间。这样通过 IPC 同步了 PA 和 PB 的进程。图 6-10 给出了带有从 PB 到 PA 的 IPC 的一个 APEG 和一个 HSDFG。

如果有一个无限长的数据序列，基于 SDFG 的建模以及随后的展开为 HSDF 图都有帮助。

例如，一个应用到快速傅立叶变换的运算或者声音数据编码中的 HSDFG。HSDF 图还可以有效地实现 IPC(Inter Processor Communication 内部处理器通信)图的建模。

(算法的细节请参考 Sundararajan Sriram 和 Shuvra S. Bhattacharya 编著的 *Embedded MultiprocessorsScheduling and Synchronization* 一节。这两个作者都是加利福利亚大学伯克利分校著名教授 Edward A. Lee 的学生。)

注意：

在一个 APEG 或者 APEG 链中的任意一个段，执行过程都是没有延迟的。这就是 APEG 模型复杂问题首先建模为 SDFG，然后将 SDFG 展开为 HSDFG，将 HSDFG 分离为 APEG。处理是按照 APEG 之间的优先级约束进行的。基于 APEG 的算法最容易调度，但是 APEG 之间算法中的优先级约束和以前一样。

6.3.6　定时的 Petri 网和扩展预测/转换网模型

回顾第 6.2 节中 Petri 网模型的应用，有两个模型可以用来开发先进的调度算法："定时 Petri 网"和概率定时 Petri 网(也称为随机 Petri 网)。这里也使用了彩色令牌。彩色令牌表示一个具有两种或者多种颜色的令牌。当令牌表示一个标识(一个异步事件)的时候，我们说它具有一种颜色，当它表示一个数值的时候，我们说它具有另一种颜色。当它表示一个禁止令牌的时候，它又具有另外一种颜色。如果禁止令牌出现在某个转换处(Petri 网模型)，该转换不能触发。该令牌什么时候改变颜色，转换就什么时候触发(回顾第 6.2 节，Petri 网转换展示了一组没有任何输入的情况下继续进行的运算。在转换处所进行的行为和运算与 DFG 或者 APEG 中相似)。

在"定时 Petri 网"模型当中，结点位置和结点转换处的时间(延迟)也像示例 6-5 和图 6-11 那样标记在图上。当该延迟不是常数而是由概率分布函数定义的时候，该模型被称为"定时扩展预测/转换网(tEPr/T)"(时间的概率分布函数可以如下理解。延迟在某个限制之内可以有 50%的可能性，延迟在某个限制之内可以有 70%的可能性，等等。预测表示该延迟或者分布函数是可预测的)。

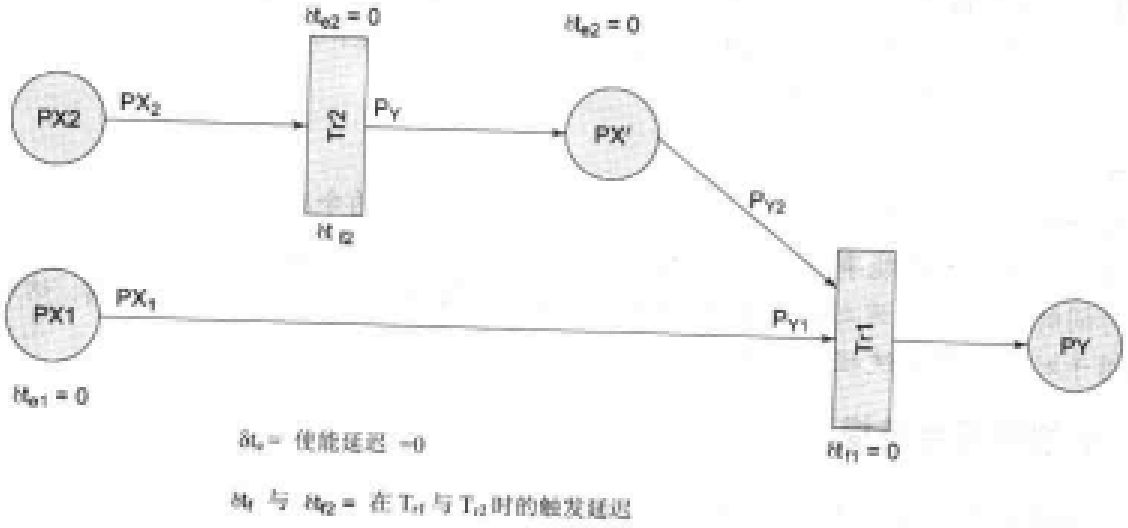


图 6-11　一个定时扩展预测/转换网(tEPr/T)

具有相同数量、相同容量和相同做法的同一种食品在两个微波炉上的烹饪周期是否相同？

不相同。附近的气流不允许存在烹饪周期相同的概率。相似地，没有两个相同的电子门恰好具有相等的传输延迟，进程和处理器的行为也不会相同。运行在两个处理器上的两个并行进程的执行时间的不确定性服从概率分布。因此，我们要使用定时扩展预测/转换网(tEPr/T)(请参考示例 6-5)。

定时扩展的意思是指结点位置处的时间(触发时间)和被触发的运算时间也都加入到模型当中，就象 SDFG 中在点处所表示的延迟一样。扩展的预测转换的意思是转换具有一定程度的不确定性的可预测性，并且使用概率函数表示。一个转换连接多个结点位置，并且可以向多个结点位置进行输出(想要了解细节可以参考 Tadao Murata 的经典著作 *Petri Nets, Properties, Analysis and Applications*)。

示例 6-5

图 6-11 给出了一个定时扩展预测/转换网(tEPr/T)。有一个从两个结点位置 PX'和 PX1 获取令牌的结点转换 Tr1。一旦从两个位置接收到彩色令牌，就会触发运算。还有另一个从结点位置 PX2 获取令牌的结点转换 Tr2。一旦接收到所有的彩色令牌，就可以像 DFG 那样触发运算。在转换 Tr1 与 Tr2 之后，这个新的结点位置是 PY。这样，每个触发可以分别在一个延迟(也就是说 δt_{e1} 和 δt_{e2})之后进行。假设 δt_{e1} 和 δt_{e2} 分别表示 Tr1 与 Tr2 的运算时间，称为触发时间。如果($\delta t_{e2} + \delta t_{r2}$) < ($\delta t_{e1}$)成立，那么都必须使得在持续时间= δt_{r1} 之后可以获得令牌。在 δt_{e1} 之后，后续的转换只能够从结点位置 PY 处触发。使用定时 Petri 网或者 Petri 表，建模可以和使用 SDF 一样进行。所有的 4 个延迟在 Petri 网表示当中都是固定的。在 tEPr/T 模型当中，每个延迟都有一个概率分布。

回顾一下第 6.2.1 节中所描述的 FSM 模型。它表示一个 CDFG 方法(第 6.1.2 节)。在这两个模型当中，步骤(程序路径)都是确定的。FSM 通过状态转换函数确定下一个状态。CDFG 确定发自决策点的下一条路径。考虑一个循环调度程序。FSM 模型和 CDFG 模型都是可用的。APEG 可以取代 CDFG 方法。

当存在非无环(异步)数据输入，且步骤还没有确定的时候，SDFG 模型和 tEPr/T 模型表示 DFG。除非在 SDFG 当中存在一个初始令牌，该图都不会在下一个结点处继续进行运算。除非定时事件发生，转换不会在定时 Petri 网或者 tEPr/T 当中继续(不会触发)。当调用一个多处理器系统的时候，这两者都可以应用。

注意：

彩色令牌是指一个可以作为是输入、事件或者禁止事件的令牌。其颜色可以发生改变。例如，从禁止事件变为使能事件。定时 Petri 网是定义了转换的触发时间以及已经被触发的转换的运算时间的 Petri 网。定时扩展预测/转换网是时间按照概率函数分布的 Petri 网。这两种类型的网都用于对多处理系统程序流建模。

6.3.7 多线程图系统模型

我们假设某个任务设计是一个线程(第 8.1.3 节)。图 6-12 给出了一个 MTG 系统模型。多线程模型是其软件由带某些控制和调度机制的多线程组成的系统。(第 9.4.2 节)程序的 MTG 模型有两层，一层是 CDFG，另一层是定时 Petri 网。下面给出了使用 MTG 进行建模(要了解相关细节，读者可以参考 Filip Thoen 与 Francky Catthor 编写的 *Modeling, Verification an Exploration*

of Task-Level Concurrency in Real Time Embedded Systems)。

(1) 用 FSM 表示的不同线程模型之间的任务级并发。图 6-12(a)给出了一个利用 FSM 建模作为线程的任务(回顾一下 FSM 正好是 Petri 网的子类，其中转换只能从一个结点位置接收输入，并且只能向一个指定的结点位置发送输出)。

(2) 存在对不同处理器上的线程进行并发处理，以及通过定时 Petri 网(作为 SDFG 的候选)建模的处理器级并发模型。图 6-12(b)给出了两个处理器 PA 与 PB 上的定时 Petri 网。

MTG 的特征如下所示：

a. MTG 提供 RTOS 级优化的静态和动态调度。

b. 时序分析和验证方法分开进行。

c. MTG 实体之间的暂时分隔的计算为多处理器系统给出了性能矩阵(性能的定义可以参考第 6.3.6 节)。

注意：

多线程图模型利用两层进行建模的，一层是 CDFG，另一层是定时 Petri 网，分别针对任务级和处理器级的并发处理。

6.3.8　图和 Petri 网在多处理器系统中的应用

A. 先划分后调度

回顾一下图 6-10。当多处理器并行的时候，如何对程序进行划分，以发生下面的情况。

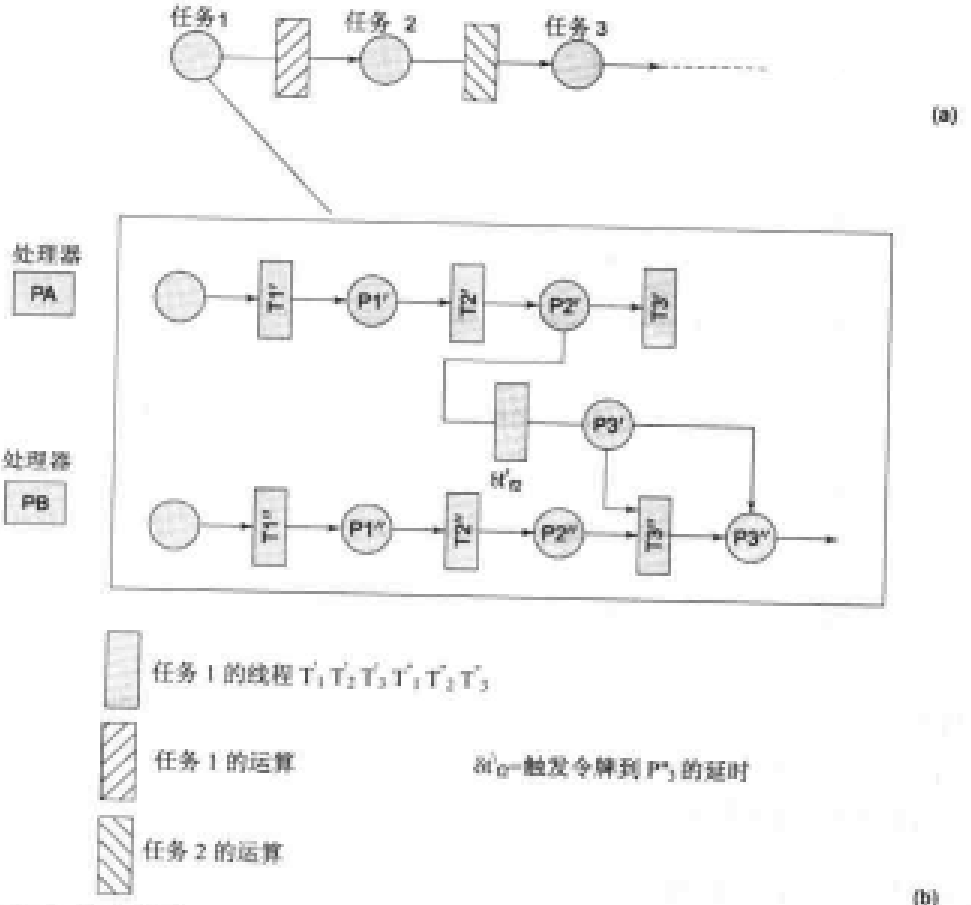


图 6-12　一个 MTG 系统模型。(a)用 FSM 建模的线程(任务)级并发处理(Petri 网的一个子类)；(b)使用定时 Petri 网对于任务 1 进行建模的处理器级并发处理

(1) 最少量的 IPC，使得 IPC 延迟(等待时间)的总时间最短。

(2) 达到负载平衡。每个处理器通过共享处理负载都有最短的等待时间。

(3) 性能代价最小。性能代价的意思是(a)对令牌进行运算所需要的执行时间以及边沿延迟(通信时间)，(b)在被某个结点(转换)触发之前的运算时间，(c)上下文切换时间。

在每个结点处进行运算，使得优先级约束不变(保持完整)。这样，程序图划分为函数、任务或者线程。有三种策略可以调度程序运行：

(1) a.每个任务或者函数在一个指定的处理器上执行。(b)每个任务或者函数在不同时期不同处理器上执行。(c)四个不同任务的指令在两个处理器上进行划分。(d)四个不同任务的指令在不同时期不同处理器上进行划分和调度的指令(图 6-13(a)到(d)给出了这四个划分和调度策略)。

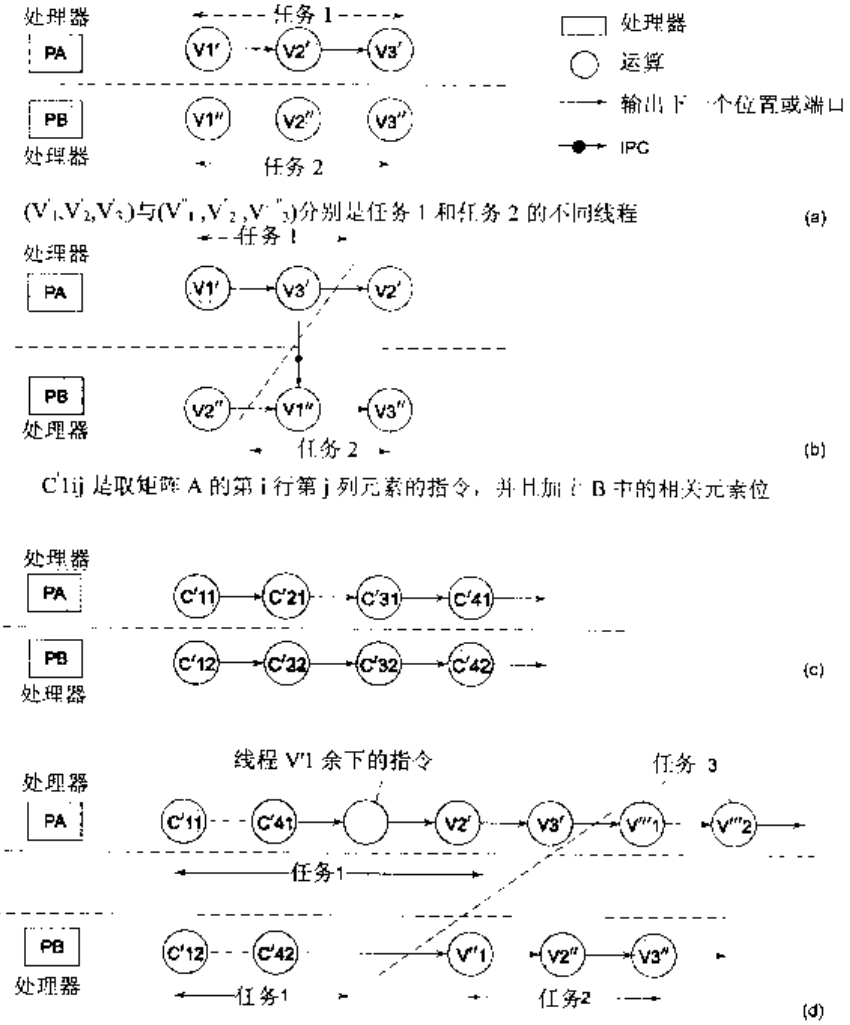


图 6-13 (a)每个任务或者函数在指定处理器上执行。(b)每个任务或者函数在不同时期在不同的处理器上执行。(c)四个不同任务的指令在两个处理器上进行划分。(d)四个不同任务的指令在不同的时期在两个不同处理器上进行划分和调度的指令

(2) 数据的每个集合在 VLIW 指令中划分，并在执行同一程序的不同处理器上执行。考虑一个矩阵加法的处理。每一行都可以加到一个不同的处理器上，这时每行的数据都可以在处理

器之间进行划分。当处理 DSP-VLIW 的时候就会进行这样的数据划分。

(3) 组合划分不但可以在任务(或者函数)级进行，还可以在数据级进行。不同的函数本身可以在不同的处理器上并发执行，但是在宏级或者原子级，指令通过数据划分来执行。Lee 的多维 SDF 模型或者 Printz 的几何划分模型都可以在图中使用(要想了解这些模型，可以参考前面提到的由 Sundararajan Sriram 与 Shuvra S. Bhattacharya 合著的 *Embedded Multiprocessors Scheduling and Synchronization* 一书)。

注意：

结点的划分和调度可以按照下面几种方法进行：(a)每个任务或者函数在一个分配的处理器上执行。(b)每个任务或者函数在不同时期在不同的处理器上执行。(c)四个不同任务的指令在两个处理器上进行划分。(d)四个不同任务的指令在不同的时期在不同的处理器上进行划分和调度。(e)SIMD、MIMD 以及 VLIW 的数据划分。

B. 通过装载平衡实现性能最小化的方法

接下来，我们利用图 6-14 中使用的 HSDFG 来对所有的过程进行建模。如下所示为划分图的步骤：

(1) 步骤 I：图 6-15(a)对该步骤进行了介绍。图划分算法将 APEG 用作 DFG。它对结点进行分配。通过对嵌套控制结构进行标识，就得到一棵图划分树。每个嵌套控制结构都给出了一个微线程。每个结点按层次组织为子图的一棵树。APEG 是这棵树的叶。考虑使用某个调度程序进行多任务调度。我们可以假设每个任务、ISR 以及它们的调度程序都是宏线程。

(2) 步骤 II：图 6-15(b)介绍了这一步。第二步是将每一个 APEG 进一步划分为微线程。在这里，每个结点都被赋给一个不同的微线程。微线程对相继进行组合直到：(i)没有引发内部宏线程通信循环(叶结点之间的边上的内部分支循环通信)，(ii)它减少了性能开销。

将程序中的平均通信时间和上下文切换时间加在一起，得出每次划分的性能开销，然后得出划分程序总的性能开销。

每个结点都有一组运算，在进行任何性能开销计算的时候，每个结点处的执行时间之和都是静态的。性能矩阵的元素等于某个线程中一条指令的性能开销。矩阵中的列与指令相关，行与线程相关。

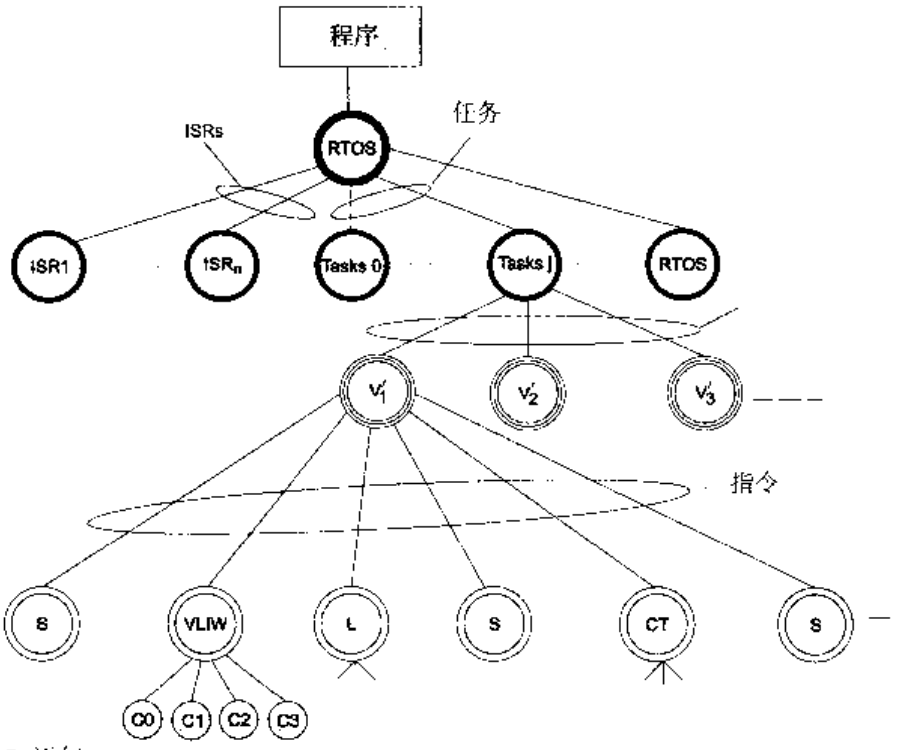
减小总的性能开销和静态开销之比，正是划分为多个进程和宏线程的目的所在。

注意：

划分是按照使总的性能开销最小的原则进行的。

C. 进行并发处理的 4 种调度和同步策略

图 6-14 中(a)~(d)给出了 4 种调度和同步策略：(a)完全静态；(b)自定时调度；(c)准静态；(d)完全动态。



S=语句
L=循环
VLIW=超长指令字
CT=条件测试
C0…Cn=操作该集合中不同数据的指令线程

= 微线程
= 单重、双重和三重圆圈分别表示任务的微线程、宏线程和线程

图 6-14　通过 HSDFG 对处理器上运行的所有进程进行划分的模型，以及每个结点都作为子图的一棵树，并且都有一个嵌套的控制结构(如宏线程或任务)的层次结构

进行每个结点处的运算时，优先级约束保持不变(保持完整)，就叫做静态调度。静态调度在编译的时候进行。处理器被分配给每个结点运算、内部结点通信以及对结点排序，使得它们能够在系统中分配的处理器上进行处理。

假设某个结点上的运算提前完成了，处理器就会保持空闲。然后，会有一种方式代替静态方式，通过这种方式可以进行动态运行时间调度和同步。它在运行时确定在哪里，在哪一个处理器之后，运行哪一个结点来执行运算。因此，完全静态方法的一种变形是，在编译时确定处理器之间进行通信的顺序，但是只有预定的事务才在运行时确定。假设有一个编译时调度，优先级首先是 V，其次是 V'，最后是 V"。在运行的时候，调度程序确定 V 应该在哪一个处理器上运行。V 运算完以后，调度程序在运行时调度 V'在同一个处理器或者另一个处理器上运行。

图 6-15　通过首先标识微线程，然后对它进行组合的方法来最小化性能开销的步骤 I 与步骤 II

完全静态方法的另一种变形是自定时调度和同步。它在 V、V'以及 V"的执行时间发生改变时使用(发生变化是由于在运算的过程中遇到的条件和控制语句)。这里，假设 V 与 V"被调度到处理器 PA 上运行，而 V'被调度到处理器 PB 上运行。这是根据编译时静态调度确定的。但是在运行时，PB 等待，直到收到来自处理器 PA 的一个运行时通信(即 V 已经完全触发)为止。处理器 A 等待直到 V'完全触发为止。在 A 与 B 之间存在一个自定时。图 6-16(b)给出了这个过程。

在运行的时候有流控制。图 6-16(d)给出了这种完全动态的调度和同步方法。对于形成一个网络的分布式处理器来说，动态调度是必需的。动态调度的一种变形是在运行时给事物分配处理器。另一种变形是准静态。

图 6-16(c)给出了这种调度和同步的准静态方法。这种方法在数据依赖于参与者的触发时间时很有用。只有概率分配和排序是在编译时进行的。由于参与者在触发期间所使用的数据的性质，它们可以在运行时被覆盖。

D. 再同步

当存在太多 IPC 的时候，整个性能开销都会增加。这种开销可以通过适当的再同步来减少。示例 6-6 对再同步机制进行了解释。

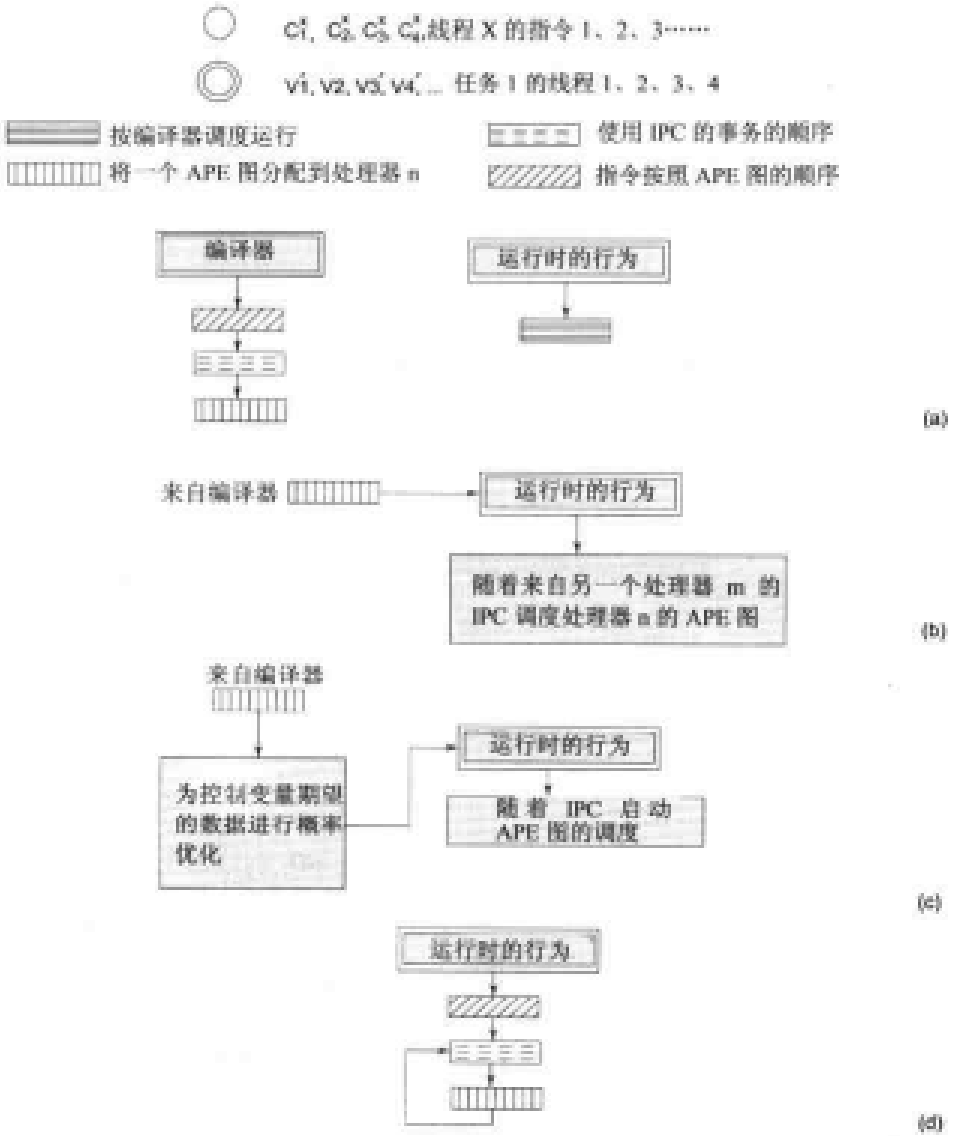


图 6-16 (a)完全静态的调度和同步；(b)自定时调度和同步；(c)准静态调度和同步；(d)完全动态的调度和同步

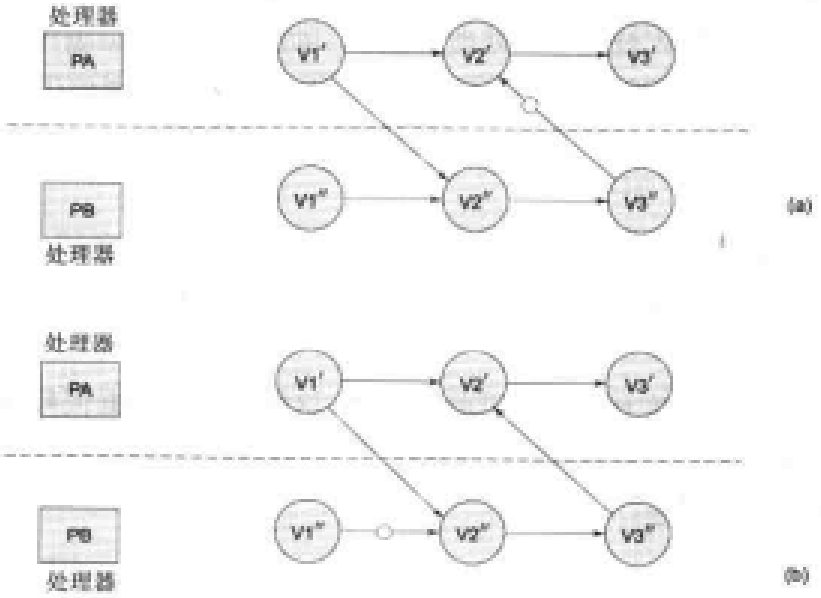


图 6-17 (a)结点的同步；(b)处理器 PA 和 PB 的再同步

示例 6-6

考虑图 6-17 中给出的调度机制。假设在处理器 PA 上，优先顺序约束是 V'_1 开始执行，其次是 V'_2 执行，最后是 V'_3 执行。在另一台姊妹处理器 PB 上，优先顺序约束为 V''_1 开始执行，其次是 V''_2 执行，最后是 V''_3 执行。这里假设从 V''_3 到 V'_2 存在 IPC(在图 6-17(a)当中，IPC 用点表示延迟)。直到有一个来自于 V''_3 的通信时，处理器 PA 才能触发 V'_2。但是假设可以进行再同步，使得 V'_2 被 V''_2 或者 V''_1 的输出触发，这样可以缩短等待时间。通过使用适当的数学理论或者分析，也可能删除冗余的 IPC。在估计了 IPC 的同步边延迟在延迟位置移动时如何减少性能开销之后可以进行再同步(图 6-17(b)给出了该点的移动和通过使用没有延迟的边来取代 PA 与 PB 之间打了点的边所进行的再同步)。

注意：

四种调度和同步策略是(a)完全静态；(b)自定时调度；(c)准静态以及；(d)完全动态。

本章小结

- 软件工程师在软件的开发过程中必须采用利用某种模型的标准设计方法。通过数据流图(DFG)的使用，软件实现得到了极大地简化，因为只需为每个用圆圈表示的过程编写代码，该过程使用有向箭头表示数据输入和数据输出。
- 在一个 DFG 模型当中，有一个数据入口点和一个数据出口点，还有使用圆圈表示的一个或者多个处理器。当对某个输入的赋值在 DFG 中固定时，也称为无环 DFG(ADFG)。编程复杂度可以通过使用尽可能多的 DFG 和 ADFG 对程序建模来实现最小化。
- 程序建模的另一个重要概念是用于程序设计和分析的受控数据流图(CDFG)。CDFG 表示结点处的控制决策和决策之后，从该结点遍历的程序路径(DFG)。
- 实时程序的程序模型是通过有限状态机模型或者 Petri 网进行建模的。
- FSM 模型适用于每次一个进程，从一个状态到下一个状态的顺序流，以及程序控制流的情况。利用 FSM 很容易对实时系统的进程中的状态进行建模。
- 基于 Petri 网的建模已经应用于嵌入式系统、控制电路和运算以及通信操作的设计算法当中。人们发现 Petri 网是实时嵌入式系统强有力的工具之一。Petri 网模型中有结点位置和转换，它们取代了 FSM 模型中的状态。Petri 网是两类结点的互连：结点位置和结点转换。
- 当 Petri 网大到不能处理的时候，Petri 表是一个强有力的编程模型。
- 多处理器系统使用两个或者多个处理器来更快地执行(i)程序函数；(ii)任务；(iii)单指令多数据流指令；(iv)多指令流多数据流指令；(v)超长指令字。DSP 指令中的 VLIW 可以高速完成。多处理器系统的建模使用 SDFG 和 HSDFG 来表示(SDFG 还可以展开，使得沿着它的边延迟的只有一个令牌)。另外，还可以使用一个不存在延迟的 APEG 表示。
- 定时 Petri 网用来对多处理器系统进行建模。它在网上给出了边上的转换触发时间和被触发了的运算的时间。
- 扩展的预测/转换网是带有用概率分布函数表示的延迟的定时 Petri 网。
- 多线程图(MTG)系统模型在用 FSM 建模的不同线程模型之间存在任务级并发和用定时 Petri 网建模的处理器级并发的情况下使用。

- 模型用于在多处理器上的程序流中进行划分、负载平衡、调度、同步以及再同步。这样可以使总的性能开销(处理延迟)最小。

关键词及其定义

- 模型：它是一种表示，使用它可以使得问题、过程、设计或者分析变得容易理解，建模之后可以简化问题。
- 数据流图(data flow graph，缩写为 DFG)：每个过程的代码用一个圆圈表示，过程的数据输入用输入箭头表示，数据输出用输出箭头表示。
- 无环 DFG(Acrylic DFG，缩写为 ADFG)：某个输入的赋值是固定的一种 DFG 模型。
- 受控数据流图(controlled data flow graph，缩写为 CDFG)：通过表示结点处的受控决策和决策之后从该结点处遍历的程序路径(DFG)来建模。
- 有限状态机(Finite State Machine)：存在有限模型的状态。在一组给定的输入集合之后，状态根据状态转换函数进行改变。
- 状态转换函数(State transition function)：将程序从一个状态转换到另一个状态的过程或者代码状态。
- Petri 网(Petri-net)：由 C.A. Petri 给出的一个模型，其中存在两种结点的互连：结点位置和结点转换。用矩形对转换时的参与者(在这里运算被触发)建模。用于触发转换的令牌来自结点位置。令牌是一个输入、标识或者异步并发事件。
- Petri 表(Petri table)：用于表示 Petri 网的表。当 Petri 网变得大到不能处理的时候，Petri 表是一种强大的实时编程模型。
- 多处理器系统(multiprocessor system)：它是一个这样的系统，使用两个或者多个处理器来更快地执行(i)程序函数；(ii)任务；(iii)单指令多数据流指令；(iv)多指令流多数据流指令；(v)超长指令字。
- SDFG：它是一种还要显示输入延迟的 DFG。圆圈(结点)表示参与者，运算像在 DFG 中的 Petri 网转换或者结点处那样执行。边(弧或者箭头)上用点表示延迟，并标记输入和输出的数目。
- HSDFG：SDFG 可以展开的表示方法，使得只有一个令牌，它的延迟沿着它的边。
- APEG：没有延迟的 SDFG。
- 调度(scheduling)：不同处理器上的不同结点或者子图的分配。
- 定时 Petri 网(Timed Petri-Net)：在边上展示了转换触发时间和触发了的运算的时间的 Petri 网。
- 扩展的预测/转换网(Extended Predicate/Transition Net)：具有用概率分布函数表示的延迟的定时 Petri 网。
- 多线程图(MTG)系统模型：这样一个模型，其中存在用 FSM 建模的不同线程模型之间的任务级并发和用定时 Petri 网建模的处理器级并发。
- 划分(partitioning)：将图划分为几个部分，每一部分在所采用的每个调度策略分配的处理器上运行。
- 负载平衡(load balancing)：微线程和指令的划分和调度，使得每个处理器都共享多处理器系统当中负载的处理。

- 宏线程(Macro Thread)：将很多过程和指令组合到一起得到的一个线程或者线程的一个部分。
- 微线程(Micro thread)：某个线程当中，参与者(结点、转换或者 DFG 结点)能够表示的最小的一组指令或者控制嵌套结构。微线程到宏线程的组合用来最小化性能开销。
- 性能开销(Performance cost)：在某个结点、子图或者微线程处执行所花费的时间。
- 总的性能开销(Total Performance Cost)：所有性能开销的总和。如果处理上的负载是平衡的，那么总的性能开销会最小。
- 再同步(resynchronisation)：通过适当的数学分析重复进行同步，减少 IPC 的数目，由此减少多处理器系统中处理器等待 IPC 导致的延迟。

问题回顾

(1) 为什么程序复杂性会随着 DFG 的减少和决策结点数目的增加而增加？

(2) 分别用一个例子解释 APEG、彩色令牌、SDFG、HSDFG 以及再同步。

(3) 为什么要将 SDFG 展开成尽可能多的 HSDFG，又要将 HSDFG 展开成尽可能多的 APEG？

(4) 并发处理如何有助于 VLIW 指令高速执行？

(5) 如何在两个处理器上调度一条 SIMD 指令？

(6) 如何在两个处理器上调度一条 MIMD 指令？

(7) 如何在两个处理器上调度一条 VLIW 指令？

(8) Petri 表怎样有助于 Petri 网？请用两个问题示例进行解释。

(9) 在多处理器系统当中使用完全动态的调度和完全静态的调度是什么意思？

(10) 使用负载平衡是什么意思？通过组合划分怎样达到负载平衡？

实践练习

(11) 回顾表 3-3 中的 HDLC 网络协议格式。为程序画一个数据流图模型，该程序传送帧起始位、信息帧控制位、数据位、FCS 位和帧结束位。

(12) 如何将练习 11 中所画的 DFG 修改为受控数据流图(CDFG)，CDFG 为位填充提供决策结点(当给数据传送了五个 1 的时候，要额外地填充一个 0)。

(13) 如何将练习 11 中所画的 DFG 修改为受控数据流图(CDFG)，CDFG 提供了决策结点以如下所示的三种情况跟随行 3a、3b 以及 3c 中的三条路径之一：(a)每当出现第 1 位的时候；(b)如果第 2 位=0 且第 1 位=1；(c)如果第 2 位=1。

(14) 画一个 CDFG 来合并循环起始处和循环结尾处的决策结点以限制附录 D 中等式 D.3 的总和项达到 n=10。该等式为在 IIR 滤波器中使用的 $y_n = \Sigma a_i.x.(n-i) + \Sigma b_i.y.(n-i)$。

(15) 现在，根据尽可能多的 DFG 进行建模，通过将 CDFG 转换为 DFG 来使用尽可能多的 ADFG，且在该程序当中不使用循环来最小化练习 14 的结果 CDFG 中的编程复杂性。

(16) 为一个巧克力自动售卖机程序画一个 FSM 模型(第 11.1 节)。机器只允许一种类型的硬币，Rs.1，每次只能出售一块巧克力，一块巧克力价值 Rs.8。

(17) 为练习 16 中的问题画一个 Petri 网模型。

(18) 回顾一下示例 5-6 中在网络上对控制流使用的 FIPO 队列。用彩色令牌将程序建模为 Petri 网。

(19) 用 Petri 网模型将示例 8-1(第 8 章)中的 P 和 V 信号量的程序建模为互斥函数。

(20) 画出下列情况的多处理器系统：(a)存储器紧密耦合；(b)松散耦合；(c)交织耦合；(d)环状耦合；(e)带状耦合；(f)树状耦合。

(21) 给出两个可以使用概率定时 Petri 网的实例系统。

(22) 你如何解决下面的问题："如何将一个程序划分为不同处理器上的任务或者指令的集合？指令和数据如何利用获得的处理器时间和资源进行调度，使得性能优化？"

(23) 多线程图模型使用两层进行建模，一层是 CDFG，另一层是分别为任务级和处理器级并发进程建立的定时 Petri 网。在处理图像输入和图像压缩算法的过程当中，您将如何使用它？

(24) 假设将 4 个进程调度到两个处理器上运行。程序按照这样一种方式进行划分，每个进程在每个处理器上运行 10000ns。上下文切换/微秒的最小值是多少？

(25) 假设在练习 24 当中每 100ns 出现一个内部进程通信。假设平均通信时间=5ns，被划分的程序总的性能开销是多少？

第7章　嵌入式软件开发过程中的软件工程实践

本章前所学内容

我们来对目前为止已经学习过的关于开发嵌入式软件的以下几点进行概述。

(1) 面向过程的C语言与面向对象编程的C++和Java语言在大多数嵌入式系统程序设计中使用。设备驱动程序和应用软件代码的编写大多数都是使用C/C++程序设计元素来完成的，如：修饰符、条件语句与循环、指针、函数调用、多函数、函数指针、函数队列以及服务例程队列。

(2) 在编写代码时使用了不同类型的数据和不同的数据结构，例如数组、队列、堆栈、链表和树。

(3) 嵌入式编程使得优化系统存储器需求的方法也得以应用。

(4) 与生成最终代码一样，嵌入式系统软件转换为ROM映像是通过使用源代码设计工具、代码管理工具、编译器、交叉编译器、链接器以及定位程序实现的。

(5) 在软件实现过程中，程序员将程序建模为DFG和CDFG模型。

(6) 在软件实现过程中，程序员通过建立FSM和Petri网的程序流模型来对实时系统建模。

(7) 通过使用SDFG、HDSFG、APEG图、定时Petri网、扩展的预测/转换网以及多线程图(MTG)系统等程序设计模型，程序员对多处理器系统的软件实现进行建模。

非常明显，除了需要管理大量的数据类型、结构和算法(函数)之外，嵌入式软件开发还具有很高的复杂度。

也可以看到，嵌入式系统程序与通常的计算机程序相比具有以下几个不同之处。(i)在嵌入式系统中存在几个功能差异很大的组件。物理设备驱动例程和虚拟设备驱动例程、应用软件和操作系统的功能都具有很大的差异，不但要对它们进行适当地设计，而且要进行系统集成。(ii)各种软件组件都具有响应时间约束，且通常都必须满足严格的时限。(iii)所有的软件组件还必须优化使用存储器。(iv)每个软件组件的执行速度都必须是最优的，以使得在满足了所有时限的情况下开销达到最小化。(v)软件的复杂度必须在一定的范围内，必须经过仔细地测试和调试，因为它的代码将会永久性地嵌入到系统的ROM中。因此，软件算法的开发过程必须使用标准的软件工程实践。

除了复杂度最小的系统之外，在其他任何系统中，不遵循软件工程的方法和实践，匆忙地直接进行编程都是不可取的。软件工程专家进行的研究已经表明，一般来说，设计者50%的时间将用于计划、分析和设计，40%的时间用于测试、验证以及调试，10~15%的时间用于编写代码。

很多标准教材有助于深入理解软件工程的原理、方法以及实践。软件工程包括：

(i) 开发和使用技巧以及 CASE(Computer Aided Software Engineering，计算机辅助软件工程)工具，使得我们能够对系统进行正确地分析和说明；

(ii) 在软件实现中使用一套合适的源代码设计工具；

(iii) 进行测试和验证以及确认，以开发出高质量的软件。

应该学习深入开发这些技术的标准教材(例如，*SoftwareEngineering – A Practitioner's Approach*，它是由这方面的国际权威 Roger S. Pressman 编著的，是 McGraw-Hill 于 2001 年出版的第 5 版 20 周年纪念版。也见“参考书目”)。

本章将学内容

因此，本章的第一个目标是学会软件工程的概念，并解答下列问题：

(1) 什么是算法的复杂度？

(2) 什么是软件开发过程中的阶段和模型？

(3) 什么是已开发软件的实现、测试、调试以及验证过程？

嵌入式系统中需要实时程序。本章还将解答下列问题。

(4) 什么是实时程序设计和软件开发过程问题？

当软件开发完毕，其设计和代码完成以后，还必须对其进行维护。本章的下一个目标是解决以下问题：

(5) 如何管理项目？什么是软件管理实践和在计划阶段估计工作量的项目测度？

(6) 如何维护软件？

统一建模语言(UML)是在开发、设计或程序设计过程中使用的一种建模工具。本章的下一个目标是要学会：

(7) UML 的基本组成元素。

(8) UML 图形。

(9) UML 在软件建模中的应用。

7.1 软件的算法复杂度

一个原则是，在开发过程中，复杂的系统需要花更多的精力。软件复杂度可以通过如下几点来理解：

(1) 复杂度可以用来预测(a)关于可靠性的关键信息；(b)通过源代码的自动分析或者过程设计信息的软件系统的可维护性。

(2) 在进行软件开发的过程中，复杂度的测度也提供了数字反馈，以通过控制项目的设计活动来帮助开发软件。

(3) 复杂度提供了关于模块的详细信息，有助于将注意力集中到测试和维护过程中具有潜在不稳定性的地方。

(4) 面向对象(OO)设计的复杂度是依据结构特征进行衡量的。它是通过检测 OO 设计的类之间如何相互关联得到的。

如何衡量软件算法的复杂度？至少存在 18 种不同种类的软件复杂度。最常用的方法之一是由 McCabe 提出的(请参考由 Roger S. Pressman 编著，McGraw-Hill 于 2001 年出版的 *Software Engineering—a Practitioners Approach* 第 5 版，20 周年纪念版)。它被称为“圈复杂度(Cyclomatic complexity)”。当使用流程图中的术语来说明时，独立路径是这样一条路径，它至少移动过一条在该路径被定义之前未曾遍历过的边(弧或者有向弧)。独立路径是由于至少一组新的处理语句或者一个新的条件而被遍历的路径。图 6-2(a)给出了 8 条路径。其中有 4 条是独立路径。圈复杂度 C(g)等于独立遍历的路径数目。C(g)运算给出了要寻找的路径数目，在这个例子中是 4(除了那条期望用于推动程序流的路径)。示例 7-1 给出了 3 个计算公式，并对图 6-2 中 CDFG 的 C(g)进行分析。

示例 7-1 3 种 C (g)计算公式如下。

(1) C (g) = E – N_d +2，其中，E 等于图中边的数目，N_d 表示决策点的数目。E = 6(每个结点有 3 个入边和出边。4 个结点共有 12 条边。但是每条边都是两个结点之间共享的。所以边的数目取为 6)。结点的数目 N_d = 4。根据这个公式可以计算出圈复杂度为 4。

(2) C (g) = P + 1，其中 P 表示 CDFG 中所包含的预测判断结点的数目。在图中，预测结点的数目=3。根据这个标准，圈复杂度计算出来还是 4(某个结点处存在两条可预测路径，一条在测试结果为真(yes)的时候选择，另一条在测试结果为假(no)的时候选择，以此定义了预测结点)。图 6-2(a)给出了 P=3(3 条可预测路径)。在测试起始处的测试结点只是一个决策结点，而不是一个预测结点。y_n 的计算在输入处出现新的 n 值时开始进行。

(3) C (g)等于流图中的区域数。存在 4 个区域。第 1 个区域用于计算 e。第 2 个区域用于计算 s。第 3 个用于计算 y_n。第 4 个用于测试流起始处的条件。根据这个标准，圈复杂度计算出来仍然是 4。

假设程序具有逻辑复杂度。在一个算法的基本集合中，独立路径的数目等于确保所有的语句都至少执行过一次的测试的最大数目。图 6-2(a)给出了 4 个测试条件和 4 条独立路径。因此，例 6-2 当中的圈复杂度可以取为 4。

McCabe 的圈复杂度提供了最大模块的量化说明。从大量的程序设计项目中，可以发现圈复杂度 10 是模块大小的实际上界。当圈复杂度超出了这个上界时，要充分地测试一个模块就会变得困难了。

注意：

软件复杂度是一种重要的尺度。它给出了最大算法规模和可靠性的量化说明。它对必须要测试的路径数目进行衡量。它还是衡量维护软件困难程度的尺度。圈复杂度是由 McCabe 提出的量化尺度。它给出了只要存在软件复杂度，就应该进行测试的独立路径的数目。

7.2 软件开发生命周期及其模型

表 7-1 列出了软件开发过程中 3 个阶段的活动。

表 7-1 软件开发中的阶段

阶 段	活 动
系统需求的定义和分析	(1)使系统需求的说明书完全清晰是定义阶段的目标。(2)说明书应该包含(i)需要对数据进行的处理，(ii)必需的函数和任务以及所期望的性能，(iii)期望实现的系统行为，(iv)设计的约束，(v)期望得到的产品的生命周期，(vi)人机交互，(vii)最终开发出的系统的确认标准(viii)交付的时间安排
开发	软件设计、代码编写和测试是开发阶段中的主要活动。在软件设计阶段，系统的概念性设计主要包括(i)结构化数据，(ii)实现函数，(iii)实现接口，(iv)它们所使用的算法和语言，(v)要使用的测试方法。概念性设计带来了应用软件的代码开发。测试和确认是所有软件开发过程的关键活动
支持	嵌入式软件应该不需要什么支持。支持阶段的活动是纠正所发现的错误和通过增加额外的由于改变环境或者软件的重新组织而必需的函数来增强软件

对生命周期进行管理有 7 个可以进行选择的重要模型：(i)线性顺序生命周期模型；(ii)RAD(rapid development，快速开发)模型；(iii)增量开发模型；(iv)并发式开发模型；(v)基于组件的模型；(vi)基于第四代生成工具的开发模型(4GL 模型)；(vii)面向对象的开发模型。

7.2.1 软件开发过程中的线性顺序模型(瀑布模型或者生命周期模型)

图 7-1 表示了该模型。开发过程中的每个阶段如表 7-2 中所示。线性顺序模型是软件工程方法进行软件开发过程的一个经典模型。软件开发要经过 4 个顺序的增量阶段。它的路径像一个瀑布，因此，也称为瀑布模型，即发现错误和故障时，可能会多次返回到起始处。该模型从开始一直进行到结束，因此也称之为生命周期模型。在生命周期中，开发周期重复进行这 4 个阶段，直到最终软件被确认(完成所有的测试，证明符合所有的需求说明)为止。

表 7-2 线性顺序模型的软件开发过程的阶段

阶 段	活 动	模型的缺点
第一阶段	对系统需求进行建模和分析，第一个阶段和第四个阶段更易于模块化	当所有的团队成员都完成了某个阶段的工作之后，相互依赖的活动可能会发生阻塞
第二阶段	数据结构、软件体系结构、接口和算法的属性设计	
第三阶段	通过转换设计概念来创建代码(软件实现)	
第四阶段	测试算法的内部逻辑、测试外部函数和发现错误与故障	

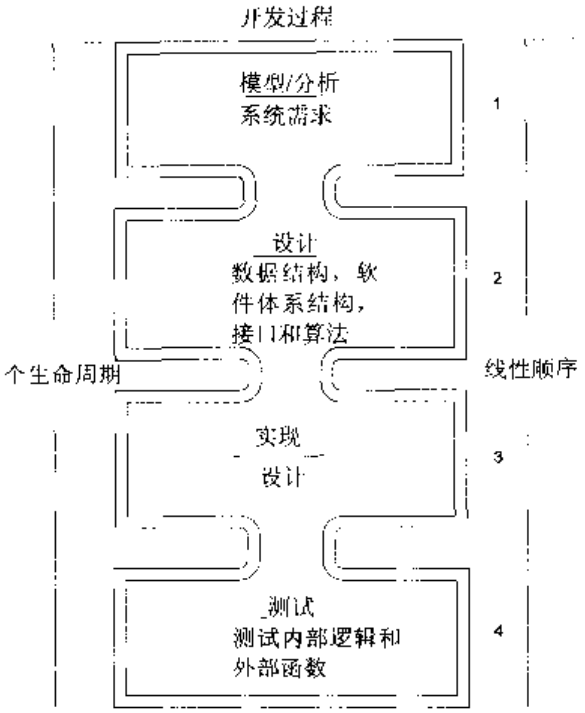


图 7-1　软件开发过程中拇性顺序模型(瀑布模型或生命周期模型)

7.2.2　RAD 模型

高速开发模型是针对开发小组分别进行每个软件组件的开发，而对线性顺序模型的改进。在开发过程中，RAD 的各个阶段如表 7-3 所示。

表 7-3　RAD 模型在软件开发过程中的阶段

阶　段	活　动	模型的缺点
第一阶段	业务功能信息流的业务建模	要求每个成员进行快速开发，在存在大量软件组件的情况下，需要更多的小组成员，缺乏易于获得的可重用组件，这样会延迟开发周期，在快速开发过程中可能会出现错误
第二阶段	数据结构和对象组的特征和属性，数据结构、软件体系结构、接口以及算法属性的数据建模设计	
第三阶段	创建增加、删除、修改以及获得数据对象的描述的过程建模	
第四阶段	通过重用软件组件进行的代码创建	
第五阶段	由于重用以前测试过的软件组件，所以测试阶段很短。将算法的内部逻辑、外部函数的测试以及寻找错误相结合	

7.2.3　增量模型

采用增量的方式交付软件需要多个阶段。第一个阶段设计的软件要先交付，下一阶段设计的软件随后交付。然后再交付接下来的改进版本。不断改进软件的增量过程如表 7-4 所示。

表 7-4 使用对线性顺序模型进行改进得到的增量模型进行软件开发的各个阶段

阶 段	活 动	模型的缺点
第一阶段	分析、设计、编码和测试与线性顺序模型相同	特别需要新的越来越好的带有附加功能的软件，并且需要预测未来的需求
第二阶段	当第一个阶段处于设计阶段时，第二个阶段开始分析下一个增量软件	
第三阶段	当第一个阶段处于编码阶段，第二个阶段处于设计阶段时，第三个阶段开始分析第三个增量软件	
第四阶段	当第一个阶段处于测试阶段，第二个阶段处于编码阶段，第三个阶段处于设计阶段时，第四个阶段开始分析第四个增量软件	

7.2.4 并发模型

并发模型使得软件开发过程中构成产品的每个元素的所有阶段都并发进行。例如，考虑一下过程分析阶段的活动。其每一个元素可以处于(i)“开发中”；(ii)“等待变化”(在第二个和接下来的周期里)；(iii)“修改中”；(iv)“结束”。客户/服务器过程体系结构非常适合于软件开发过程中的并行开发。

7.2.5 基于组件(面向对象)的软件开发过程模型

基于组件的面向对象软件开发过程模型具有表 7-5 中所示的阶段。

表 7-5 基于组件的面向对象软件开发过程

阶 段	活 动	模型的缺点
第一阶段	标识可以在软件开发中使用的组件	需要组件的接口强大，如果得不到需求数量的可重用组件就会使得开发减速
第二阶段	从软件组件资源库中选择可用的类(单逻辑耦合的组)	
第三阶段	对可获得的组件、通过重新组织可以重用的组件以及不可获得的组件进行分类	
第四阶段	重新设计组件和创建不可获得的组件	
第五阶段	用组件来构造软件，并对其进行测试	
第六阶段	重复进行构造直到最终的软件确认	

在进行开发的过程中，开发组成员可以遵循并发工程的方法。成员并发地设计组件。例如，一个小组可以设计设备驱动程序，另一个小组可以设计错误处理例程，而还有一个小组可以设计应用软件。

请参考 7.10 节，了解面向对象的设计方法和表示设计的建模语言。

7.2.6 基于第四代工具的软件开发过程模型

我们可以使用能够根据较高级的设计规范生成代码的软件工具。这些工具被称为第四代工具。例如用于自动报表生成、自动高级图形生成、创建数据库查询以及在创建网站时用于自动

HTML 代码生成的工具。另外，还可以使用嵌入式系统的 RTOS 工具。

第四代工具可以加快开发过程，也可以提供经过测试和调试的解决方案。其缺点是需要进行详细地分析、设计和确认。

7.2.7 基于面向对象和基于第四代工具的方法

目前，软件开发过程结合了面向对象的方法和第四代工具。该方法还必须针对嵌入式系统进行完善。

不断改进的模型，其过程生命周期是迭代的，直到进行验证、确认和交付(或者安装到系统的 ROM)为止。

注意：

软件开发过程中的活动分为三个阶段：系统需求的定义和分析，设计开发，编写代码和支持。软件开发过程可以用一个称为生命周期的时间段来表示。从最后一个阶段开始，该周期可以反复从第一个阶段重新开始，直到完成软件的验证和确认。改进是通过反复迭代来实现的。有几种正在使用的软件开发过程模型。这些模型中有 7 个已经讨论过了。

7.3 软件分析

软件分析中的要点如下：

(1) 需求分析是软件开发过程中的第一步(请参考表 7-2)。首先写出语句，然后从这些概要语句中获得具体的技术规范。

(2) 任何问题都可以建模为三个域：信息域、功能域以及行为域。

(3) 一个问题可以被划分为几个部分，并通过描述问题本质的恰当符号来表示。

(4) 很多时候都不可能在早期就完全详细地说明问题。

(5) 原型的开发和原型说明通常能够提供帮助。

(6) 即使进行最好的分析和建模，问题也可能发生改变。因此，开发小组与客户之间的交流是非常关键的，应该在设计过程开始之前进行；两者对问题、域、需求、软件以及系统规范有相同的理解。

分析可以建模为三层结构。(i)数据及其内容和信息形成内层；(ii)下一层是 DFG、CDFG、实体关系图以及状态转换图；(iii)数据、控制、过程名、符号以及规范(描述)形成最高层。在嵌入式网络系统中也可以有协议的规范和描述。

实现规范——消息如何在不同的对象之间传递——也可以通过分析进行描述。在定义需求的时候，可以在分析阶段看到实现。

表 7-6 列出了这三层中使用不同模型和获得规范的过程中，用于分析软件需求的活动。其中还给出了每个阶段的例子。

表 7-6 模型和规范的需求分析

建 模	活 动	例 子
数据	定义数据类型和对象，了解属性，分析数据之间的关系，收集需求规范，建模实体关系	将网络数据建模为队列块，在每层发送错误检查域使用的协议类型，收发缓冲区
算法(函数)	(i)信息流的 DFG，(ii)程序流的 CDFG，(iii)实时函数和它们的属性，抢占策略以及临界区域	用于实现下列功能的算法(i)通过 TCP/IP 网络中的传输层和网络层的应用数据流；(ii)用于建立网络连接，传送和连接终端的控制算法；(iii)自动导航控制当中的实时功能
“行为”	状态和状态转换图	在等待硬币时，硬币被投入(输入事件)时，调用过程时(内部机器行为)以及巧克力交付(输出)时，巧克力自动售卖机的行为。还可以参见图 6-3 到图 6-6 中的例子
数据内容规范	数据内容命名、输入源(文件)、输出源、符号、含义以及规范	用于视频处理的帧、图像和声音格式名、输入系统名、压缩格式规范、压缩文件的名称和规范
控制规范	用于分析常规实时功能的控制规范	机器人的电动机控制规范，用于指定什么样的控制功能是实时的
过程规范	通过制定事件列表输入，事件输出及每个事件上活动过程的列表来分析过程规范	导航控制过程规范，说明如何测量、分析以及使用导航速度来控制节流阀(见 11.3 节)

图 7-2 给出了在嵌入式软件开发过程中的软件分析的活动。

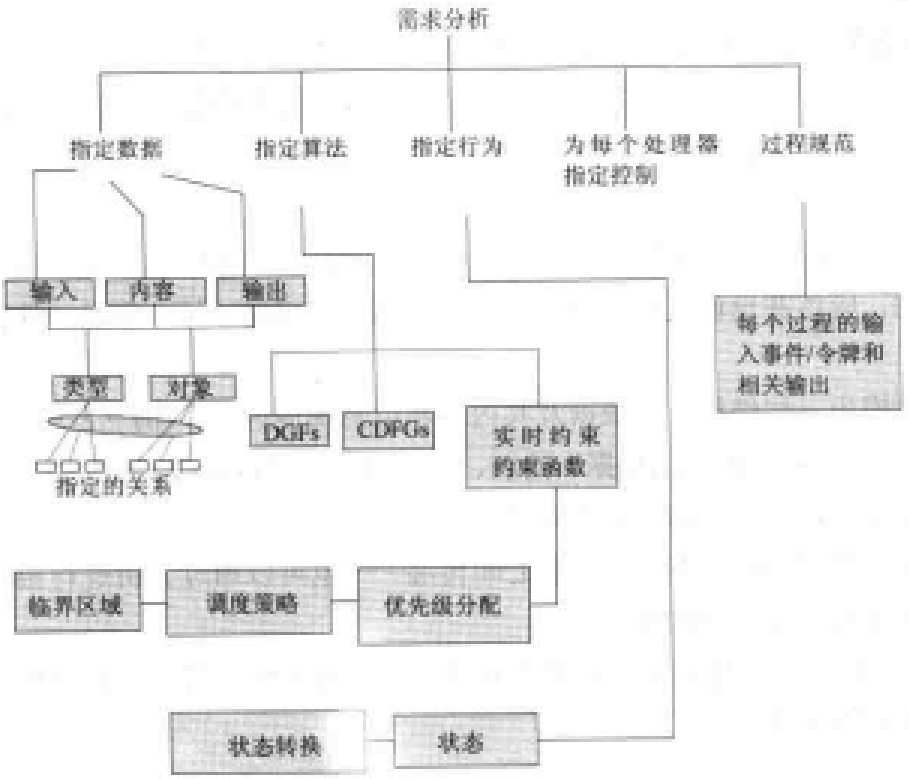


图 7-2 嵌入式软件开发过程中的软件分析活动

注意：

需求分析结束之后可以得到数据元素和结构、算法、行为、数据内容以及过程的规范。其后紧接着的是设计阶段。

7.4 软件设计

要了解开发阶段和开发过程生命周期模型，请读者参考表 7-2。在软件需求分析和规范完成之后，下一个阶段是进行设计。设计是软件开发的蓝图。它给出了要构建的系统的工程表述。软件设计包括 4 个层次。

(1) 第一层是体系结构的设计。在这一层里，将进行系统体系结构的设计。问题在于应该如何组织不同的元素——数据结构、数据库、算法、控制函数、状态转换函数、过程、数据和程序流。

(2) 第二层是进行数据设计。该阶段的问题如下：什么是最适合于给定问题的数据结构和数据库的设计？将数据组织成树形结构是否合适？在数据中，什么是组件的设计(例如，视频信息有两个组件，图像和声音)？

(3) 第三层是接口的设计。该阶段的重要问题如下：什么是集成组件的接口？什么是系统集成的设计？用于从数据对象、数据结构和数据库处得到输入以及传递输出的接口如何设计？接收输入和传递输出的端口结构是什么？

(4) 第四层是组件级设计。这一阶段的问题如下：什么是每个组件的设计？在嵌入式系统的设计中存在一个额外的需求，即每个组件都应该针对存储器的使用和开销进行优化。

在设计过程中所使用的概念有哪些呢？它们如下所示。

(1) 抽象。首先要对每个问题的组件进行抽象。例如，机器人系统问题的抽象可以根据手臂和电动机的控制进行。

(2) 在进行设计之前应该充分地理解软件的体系结构属性。

(3) 待开发系统的额外功能属性应该从设计当中充分加以理解。

(4) 在进行设计的过程中，应该对早期开发的相关系列的系统加以考虑。

(5) 使用模块设计概念。系统设计是将软件分解为将要实现的模块。模块应该在后来还能够组合(耦合或者集成)起来。有效的模块设计应该确保有效的函数独立性、内聚性以及耦合性。

(6) 应该对模块有清晰的理解，应该保持其持续性。例如，在对一个链表进行模块设计的过程中(示例 5-8)，设计模块被插入到某个链表元素之前或者之后。

(7) 对于每个模块来说，适当的保护策略也是必需的。模块不允许改变或者修改另一个模块的功能。例如，禁止某个设备驱动程序修改另一个驱动程序的配置。

(8) 根据软件需求映射为各种表示形式。例如，在程序流当中，同一条路径上的 DFG 可以映射为一个实体。在设计过程中使用转换和事务映射设计过程。例如，某个图像是某个系统的输入数据；它可以有不同数量的像素，每个像素可以具有不同的颜色。该系统不能对每一个像素及其颜色单独进行处理。图像的转换映射是通过适当的压缩和存储算法实现的。事务映射用于定义图像的顺序。

(9) 用户界面设计是设计的重要组成部分。用户的界面是按照每个用户的需求、环境和系统功能的分析进行设计的。例如，在巧克力自动售卖机系统中，用户界面是一个 LCD 矩阵显

示屏。它不仅可以显示欢迎信息，还可以针对每种巧克力指定需要投入到机器中的硬币数。界面设计必须经过软件客户的确认。例如，客户必须在界面设计进入实现阶段之前确认消息语言和要显示的消息。

(10) 改进。每个组件和模块设计都需要反复改进，直到它们最适合软件小组的实现为止。

软件设计过程需要使用体系结构描述语言(ADL)。它可以表示如下的项：(a)控制层次结构；(b)结构划分；(c)数据结构和层次结构；(d)软件过程。

图 7-3(a)给出了在进行嵌入式软件开发过程中所进行的软件设计活动。

注意：

软件设计阶段包含了从抽象层到详细设计层的设计活动。假定软件设计由 4 层组成：体系结构设计、数据设计、接口设计以及组件级设计。在设计中，需要设计者和实现者之间不断交流从而改进设计。

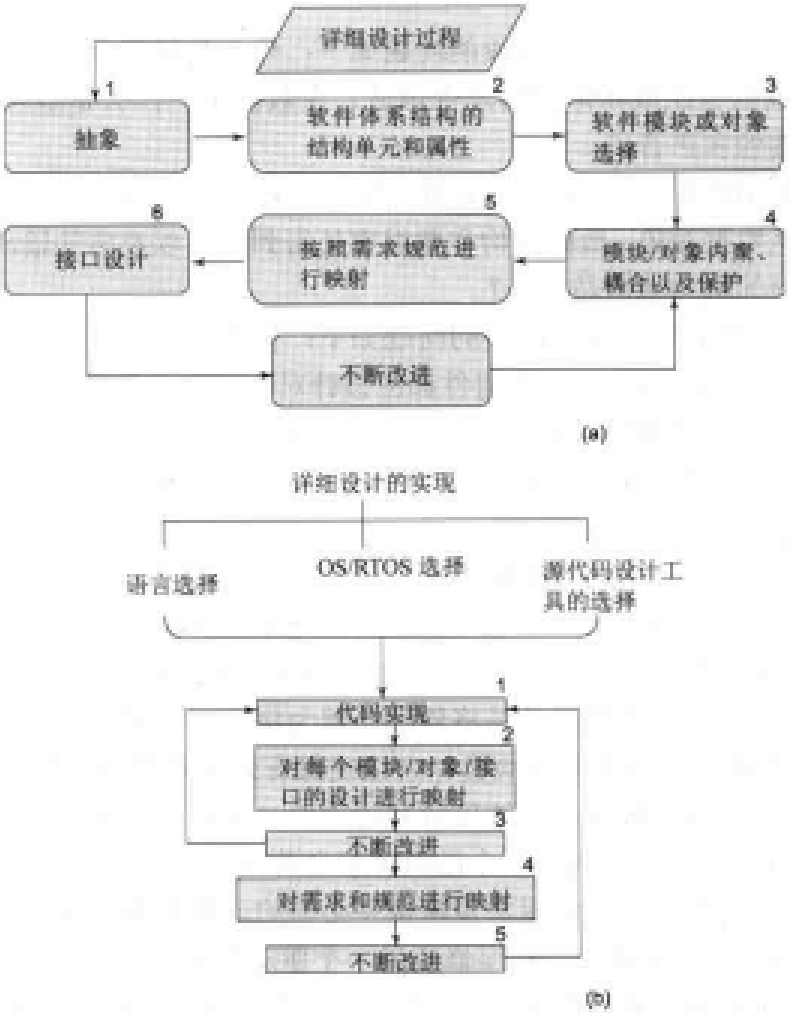


图 7-3 (a)在进行嵌入式软件开发过程中所进行的软件设计活动；(b)通过使用详细设计来完成实现

7.5 软件实现

读者可以参考表 7-2 了解开发阶段和开发过程生命周期模型。成功地完成了对需求和规范的软件分析之后，下一个阶段便是软件实现。在将软件设计进行翻译之后，实现就是现实世界

的实现。图 7-3(b)给出了使用详细设计实现方式完成的实现必须根据开发小组和客户之间达成共识的软件需求和规范进行。函数(代码)按照设计来实现信息(事件)的处理。首先分别实现每个组件的每个模块。然后再将它们进行耦合和接口。

软件是使用源代码设计工具实现的(细节请参考第 5.11 节)。对于软件实现，需要遵守特定的标准和过程。下面是实现的原则。

(1) 使用软件小组擅长和熟练的语言。例如，在实现中，如果某个小组没有 Java 编程经验，但是具有 C++技术和经验，他们应该使用 C++。好的软件必须在预定的时间内满足指定的需求，这一点是非常重要的。

(2) 使用最合适的、开发小组最擅长和熟练的操作系统或者 RTOS(在简单的实现中，小规模的嵌入式系统可能不需要 RTOS。但同样要求保证代码量最小，并且不会影响功能)。

(3) 应该保持程序简单。每个函数都应该执行一个单独的任务，并且具有简单的设计。在控制语句之前必须添加注释。这有助于开发小组的所有成员理解程序，并有助于后面进行的程序调试。循环应该是简单的、嵌套的或者连接而成的。在任何程序中都应该避免非结构化的循环。一个循环与另一个循环相交织的情况不应该发生。

(4) 程序实现应该根据程序结构设计规范进行。

(5) 令程序一般化，而不要受到特定输入或者特定事件的束缚。这样可以使得重用更加容易。

(6) 可能需要进行重新设计，以解决软件的实现困难。

注意：

每个组件及其接口系统的集成都是在设计阶段之后实现的。程序实现是用语言实现的，可以使用源代码设计工具。它应该根据程序结构设计规范进行。在实现的过程中应该保持程序的简单性。

7.6 软件测试、确认以及调试

7.6.1 测试、验证以及确认

请参考表 7-2 了解开发的各个阶段和开发生命周期模型。分析阶段之后的阶段是对设计和实现进行测试。测试是开发过程中的一个重要组成部分。嵌入式软件需要认真测试。这是因为软件会永久性地安装。

测试的目的是找出错误，并确认所实现的软件是根据规范和需求设计的。确认的意思是必须表明软件符合规范，并且在所有指定的事件、条件和时刻，软件必须展示预期的行为。对于一个良好的软件测试来说，最重要的特征是什么？如果在测试开始 20%的阶段里发现的错误越多，那么存在尚未发现的错误的可能性就越大。这样找出还没有发现的错误的可能性就越高。当找出一个在前面所有的测试当中都没有发现的错误时，这个测试就称为是成功的。测试的一个重要原则也是软件应该完全满足客户的需求。什么样的测试是最有效的测试呢？最有效的测试是第三方所进行的测试。

验证和确认具有不同的含义。验证指的是确保指定函数能够正确实现而进行的活动。确认指的是确保已经创建的软件满足分析阶段所达成共识的需求，并确保它的质量。

测试必须早在测试阶段开始之前进行计划。Pareto 原则是，在测试的过程中未揭露的错误中有 80%的错误属于程序中 20%的错误。独立组件应该在集成到系统中之前进行测试。

表 7-7 列出了 5 种测试方法。

表 7-7 测试方法

方 法	活 动	何时使用	何时不需要或者不可用
白盒测试	白盒(也称为玻璃盒)测试的意思是通过在独立的路径上测试程序流来对每个过程的细节进行检验。测试是这样进行的(i)所有的决策点(第 7.2 节)至少被遍历一次，它确保所有的条件都要测试，(ii)所有的循环都从开始执行到结束；(iii)所有的逻辑路径都要仔细地测试。测试是通过测试输入进行的。测试输入集的设计是至关重要的，它们是针对特定的循环和条件组进行的。测试是针对控制结构、条件、数据流以及循环(for 循环、while-do、do-until)进行的	白盒测试在测试的早期还没有重用前面开发的组件时使用。需要它是由于软件本身的不足所致。逻辑错误和不正确的假设经常都会发生，它们与程序路径执行的可能性成反比。可以假设某条逻辑路径不会被访问，不会发生任何情况使得这条路径被访问。但这最终是不会发生的	当测试和路径的数目太大时，白盒测试就不实际了。它应该在系统中关键的逻辑路径有限的情况下使用，应该充分地检验程序逻辑
黑盒测试	黑盒测试(行为测试)必须对白盒测试进行补充。它揭露的错误与白盒测试所发现的错误不同。测试是通过选择最合适的输入条件集合和产生测试功能有效性的事件进行的。 黑盒测试是对行为和性能的测试。黑盒测试用于：观察系统响应是否对特定的输入和在这些输入和事件处的系统故障敏感；当数值超出系统的集合范围时，系统如何反映；是否存在导致故障的数据对象、输入或者事件的特定组合；以更快的速率到达的数据输入是否会修改系统性能。 黑盒测试只看结点的图形集合之间关系。它是针对边界数值的分析，而不是针对整个范围内的输入。每个结点集合都要被检验，对功能和关系进行描述	当重点是测试功能需求，且存在紧密的交付调度时，开发小组就应当设计快速且可靠的黑盒测试	开发的早期。能够有足够的时间仔细进行白盒测试时

(续表)

方　法	活　动	何时使用	何时不需要或者不可用
特定环境的测试	特定环境测试的例子如下所示。(i)测试用户接口和 GUI：嵌入式系统的 LCD 矩阵显示和小键盘分别为 GUI 和用户接口。测试是要找出每条来自键盘的命令的影响和消息的正确显示。(ii)测试客户服务器结构：针对来自每个客户的不同输入的服务器行为都要测试。检查对客户端需求的响应是否与规范中一致。(iii)测试帮助程序和文档：系统为用户提供了帮助。这些都需要测试。文档也需要测试以确定写入文档的功能是否工作正常，甚至于是否存在	当系统设计完成，且有一个特定的环境用于测试它的行为时。例如，移动电话系统。它需要通过拨号进行测试。机器人系统需要通过指定机器人的手臂到达特定的位置进行测试	早期阶段和不完整的系统中
比较测试	几个可用版本并行运行，对功能和行为进行比较	例如，一个汽车制动系统的可靠性是至关重要的。应该测试新的软件与以前的版本相比是否具有更高的可靠性	无法得到比较系统。例如，嵌入式系统软件已经准备好，但是硬件还处于开发阶段
正交测试	测试由数值的所有组合完成	输入有限，且每个组合都可以测试	当输入的组合数目很大时

如何确定实现的软件是否可测？可测性的标准如下：(i)系统应该可以分解，使得每个系统组件都变得简单，因而更易于测试。(ii)必须针对每个输入对不同的输出进行观察，使得所有不正确的输出都易于识别，且内部错误能够自动消除。(iii)测试设计小组必须充分地理解所要设计的软件。(v)设计中的变化必须通知所有必要的组件。(iv)必须管理好技术文档，对于详细的测试更是如此。

什么时候测试能够更加有效？如果系统工作正常，那么它的测试可以更加有效。例如，如果在进行测试的时候没有一个机器人的电动机运转正常，那么就需要更加仔细的测试；当在所有其他的功能都正常的情况下只有一个电动机不运转，测试应该只针对驱动该电动机的模块进行。测试变得容易，因此会更加有效。

嵌入式软件需要的质量最高，因此需要一种更加系统化的方法进行测试。测试所采取的策略是什么呢？

(i) 每个模块或者单元的测试和组件的测试。

(ii) 烟雾测试从开始到结束对所有的系统进行测试，软件小组可以在每个阶段对整个项目

进行有规律地访问。一旦某个组件就绪，就开始对其进行测试，一旦一组组件就绪，就按照周期性的时间间隔对它们进行集成，并进行周期性测试。

(iii) 确认测试就是确认软件。在一系列显示需求一致的黑盒测试之后，软件就得到了确认。确认测试的目的如下所示：(a)测试性能。在进行测试时，每个需要的功能都要展示它具有与需求相一致的特征。(b)从需求规范中找出偏差，准备一个缺陷列表，然后创建缺陷列表。(c)测试安全性。当存在错误条件时，可以进行这一测试。例如，当某个队列超出它的大小时，软件必须执行“异常”处理例程(请参考示例 5-5 中的 ISR_Qerror 和第 4.5.2 节中关于异常的定义)。有两种类型的确认测试：(a)Alfa 测试，该测试在开发端进行。(b)Beta 测试，它在嵌入式系统的用户端进行。

(iv) 恢复测试用于决定当软件和硬件发生故障时，系统如何恢复。

(v) 系统测试的意思是集成系统测试，要么自顶向下，自底向上或者进行退化测试(测试所有测试实例的子集)。

测试一个系统存在几种含义。例如，单元测试、集成测试、设计测试以及需求测试。

嵌入式系统测试分如下两部分进行：(i)在软件开发过程中进行测试，这只在主机上进行，(ii)与目标系统一起进行测试。后面这一部分将在第 12 章中进行阐述。

与软件复杂度一样，软件测试也有度量标准。它使用(i)数据元素或者对象；(ii)数据或者对象之间的关系；(iii)状态；(iv)转换；(v)许多功能点。测试测度的计算可以通过如下所示的例子加以理解。

示例 7-2

(1) 假设(a)k_1 等于出现在第 n 个软件模块中的不同操作符的数目。(b)k_2 等于出现在该模块中的不同操作数的数目。(c)K_1 等于 k_1 个操作符在模块中出现的次数，K_2 等于 k_2 个操作数在模块中出现的次数。(d)r 等于操作符与操作数在程序中出现次数的比率，S 等于软件的大小。

(2) 这里，$c = r * k_1 /2$ 定义了程序设计的代价系数。这样定义的逻辑如下所示。某个操作符在模块中出现的次数越多，测试代价就越高。如果操作数出现的比率越高，进行测试所需要的代价就越高。之所以要被 2 除是因为操作是在两个操作数之间进行的。

(3) 针对某个模块 $t_e(n)$来说，测试代价的测度$= c*S$。该定义的逻辑是：模块的规模越大，总的测试代价就越高。

(4) 软件测试的代价为 $T_E = \Sigma t_e(n)$，其中，当软件中有 N 个模块时，n=1, 2,…, N。

7.6.2 调试

测试是检查故障和错误的活动，而调试是为某个故障找出发生的原因。调试表示一旦发现了某个错误(或者错误集)，就必须找到问题发生源和产生的原因，并且出现在软件中的错误必须矫正(确定)。在嵌入式软件当中，错误的出现是不可忍受的。考虑为航天飞船设计的一个复杂的嵌入式系统(第 1.5 节)。一旦系统中存在任何未能发现的错误，其代价是可以想象的。

事实上，有的软件工程师能够很容易调试系统，而另一些却不能。调试可以认为是一门艺术。调试的标准指导原则如下所示。

(1) 使用适当的策略。可用的策略如下：(i)在早期进行测试和调试。(ii)使用二分法对程序进行调试和定位，也称为原因消除法。其意思是首先针对期望找到的问题源将模块分成两半。然后，再将它分成另外的两半。例如，在调试一个 TV 故障的时候，首先划分为两个部分——视频和音频。如果存在音频问题，那么将音频部分电路再分成两半——检波器电路和扩音器电路，继续进行调试直到确定问题源为止。(iii)使用回溯的方法。问题源可以沿着程序流路径回溯找到。(iv)使用强力(brute force)方法。强力方法的一个例子是通过白盒测试找出问题源(表 7-7)。

(2) 进行一个有效的假设，使得在调试过程中通过追踪问题源来找出将要被检验的新值。

(3) 开发可以重复使用的测试模块。

(4) 检验先前记录的测试结果。

表 7-8 给出了各种嵌入式系统在软件开发过程中以及之后进行调试的技术。

表 7-8 软件开发过程阶段的调试技术

方 法	活 动	何时使用	何时不需要或者不可行
使用断点	程序流在某个特定点中断，通过一个设计好的输入集合来对该阶段的输出进行观察或者测试。这些点称为断点。在软件实现的过程中添加断点。在最终的验证和最终的确认之后删除断点	在开发阶段，在使用白盒测试时，在早期测试阶段，在没有重用先前开发的组件时	当断点的增加会无限延迟软件执行时
使用测试宏	专门为测试编写的宏的结果对于嵌入式软件的调试有很大的帮助(要理解 C 语言中的宏请参考第 5.3 节)	当测试函数也被添加到软件开发过程中以确保有效调试时	测试宏的增加会使得代码量超出可用的存储空间时
使用针对样例输入的输出文件	为调试阶段设计一组样例输入文件，观察系统产生的输出	软件开发阶段和进行烟雾测试时	不可能创建像事件这样的输入时
使用指令集模拟器	参考第 12.3 节	当测试汇编代码，找出指令集吞吐量，测试设备驱动程序代码，检验在特定硬件上的可移植性时	在不是 CPU 所指定的软件代码中
使用实验工具	参考第 12.5 节	将它嵌入到硬件中后，在目标系统上对软件性能进行测试时	在软件开发过程中

警告：

故障修复不应该导致额外的故障产生。

图 7-4 给出了嵌入式软件开发过程中的软件测试、调试、验证和确认活动。

图 7-4 嵌入式软件开发过程中软件的测试、调试、验证以及确认活动

注意：

测试阶段在实现阶段后面。测试方法有“白盒测试”、“黑盒测试”、特定需求的测试、比较测试以及通过正交数据输入进行的测试。测试验证在整个生命周期之后开发出的软件符合需求和达成共识的规范。需求规范的确认在向客户进行交付之前的最后一个阶段完成。如果确认失败，该周期必须根据后来的改进模型反复迭代。在调试的过程中，使用断点、测试宏、测试向量、模拟器以及实验工具追踪观察到的错误源。

7.7 软件开发过程中的实时程序设计问题

实时系统是具有额外的程序设计需求，且在其中的一个或者多个函数中必须满足响应时间(时限)约束的系统(关于实时程序模型请参见第 6.2 节)。因此，实时程序设计存在下面一些额外的问题。

7.7.1 在需求和规范的分析中存在的问题

(i) 事件在本质上是异步的，什么是事件的最大延迟和时间相关？当出现实时事件的时候，什么是程序流控制规范？

(ii) 每个软件功能(或者组件)可接受的最大响应时间(输入集与输出集之间的时间间隔)的规范是什么？

(iii) 什么是中断服务例程(ISR)可接受的最大延迟的规范，什么是它们的服务时限？

(iv) 什么是软件组件(任务)和中断优先级的规范？如何向那些具有更短时限的中断分配优先级，哪些是可以在后面阶段等待执行的例程？

(v) IPC 规范(请参考第 8 章)。

(vi) 不能满足时限的风险是什么，如果未能满足时限，补救的方法是什么？

(vii) 可靠性与可接受的容错性(请参考第 1.5 节中航天飞船的例子)。

7.7.2 设计和实现中存在的问题

(i) 处理器、存储器及其大小和硬件的选择(请参考第 1.3.15 节、第 2.2 节以及 2.4 节)。

(ii) 单处理器与多处理器系统的选择。

(iii) 选择能够使系统中存储器和功耗得到优化，软硬件设计得到优化的软件设计。

(iv) 软件语言 C、C++或者 Java 的选择(请参考第 5.1、5.8 和 5.9 节)。

(v) 使用 RTOS(实时操作系统)和在需要的时候设计它们所作的决策(请参考第 9 章)。

(vi) 是让软件开发小组设计 RTOS 还是使用已经测试和调试好了的 RTOS(请参考第 10 章)。

(vii) 测试方法、调试断点以及宏(测试向量)的选择(表 7-7 与 7-8)。

(viii) 系统 ROM 中未压缩的或者压缩的软件和输入数据的选择(第 2.4 节)。

(ix) 决定是否使用高速缓存，如果要使用，在哪些部分和数据能管理功耗。

(x) 软件中停止状态和等待状态的使用，以管理功耗。

7.7.3 系统集成中的问题

(i) 使用适当接口的统一系统的集成和开发。

(ii) 先对接口进行验证，然后对集成系统进行验证。

7.7.4 测试中的问题

由于输入、事件以及中断是时间相关和异步的，所以，与非实时软件相比，实时系统软件难于测试。系统不仅需要白盒测试、黑盒测试以及烟雾测试，而且需要所有测试用于异步事件的处理。实时系统测试、调试以及验证可以提供真实的挑战。下面给出了存在的问题：

(i) 测试实时数据是否会偶尔在某个状态中处理正确，而在其他的系统中处理失败。

(ii) 测试策略的选择：(a)通过对每个任务单独进行测试来完成任务的测试，使得不仅能够检测出逻辑错误和功能错误，而且还能够检测出满足时序约束和系统行为中的错误。(b)通过模拟实时系统的行为和研究出现外部事件时的行为进行的行为测试。

(iii) 按照单独地、成对地以及成组地测试事件得到的效果。

(iv) 硬件和软件集成的系统测试。

(v) 测试适当的中断优先级分配。

(vi) 针对每个 ISR 的每个中断源和中断源组所进行的测试。

(vii) 对任务和 ISR 的吞吐量的测试。

(viii) 在某个关键时刻到来的大量中断如何影响系统行为和性能？

(ix) 通过时序分析，仔细地对响应时间约束和是否满足所有 ISR 的时限、所有可能的事件集中的任务和功能以及它们之间的事件间隔进行测试而实现的时序的测试。

(x) 针对任务、以及 ISR 的并行和并发处理所进行的测试。

(xi) IPC 的测试(第 8 章)。

(xii) 存储器上溢和堆栈上溢的测试。

(xiii) 针对共享数据的临界区域的程序流、优先级倒置以及死锁问题进行的测试(第 8.2 节)。

(xiv) 软件在硬件中的可移植性的测试。

(xv) 在响应时间约束之内的功耗优化的测试。

(xvi) 容错和故障排除的测试。

注意：

当对算法和 ISR 存在软件响应时间约束的时候，实时系统的开发过程会更加复杂。要对时序分析、IPC、异步事件以及它们在系统处于每个状态时的集合进行测试。还要针对快速发生的事件进行测试。测试必须仔细，对容错和故障排除的测试也是必不可少的。

7.8 软件项目管理

7.8.1 项目管理

在本书中，Pressman 为软件项目的管理定义了 4 个 P：人(people)、产品(product)、过程(process)、项目(project)。表 7-9 列出了这 4 个 P，并给出了它们的功能(在这里，人实际上代表一组软件工程师或者系统开发工程师)。软件项目管理的原则如下：(a)人的组织。(b)在开发产品时，正确理解分析、设计、实现以及测试。(c)对每个阶段中过程的监控。(d)对项目进行组织和管理以确保它的成功。

表 7-9 软件项目管理的 4 要素

要　素	任　务	需要避免的问题
人，高级管理人员	负责对所有的通信和组织活动进行推动、创建环境、组织、协调以及管理。监控、追踪项目及其进度	(i)偏离这样的基本原则，即在任何项目当中最重要的是人。(ii)不平衡的方法和不必要的控制
项目技术管理员或者团队领导	(i)为过程选择语言、工具以及软件开发过程生命周期模型。(ii)调整和重新编写已有的软件规范、设计、组件以及现有的过程。(iii)在规定的时间内找出能够将最初的开发理解应用到最终产品中去的新方法。(iv)为每个活动和控制的开始、持续阶段、预定日期和结束维护一张活动图。(v)激励和鼓励实现人员。(vi)在执行组长指示和在过程及其开发进度表限定的时间内工作时，让实现人员在给定产品的范围内产生好的思想，找出新的方法	对实现人员的理解缺乏正确评价以及不协调的开发

(续表)

要　素	任　务	需要避免的问题
实现人员	使用建模、源代码设计、测试、模拟和调试工具实现软件(和硬件)开发过程	不遵循已达成共识的设计方案，实现人员之间缺乏协调性
软件(或者嵌入式系统)客户	指定产品及其质量需求，商议费用	干预开发过程，在达成一致意见之后又更改产品说明书
最终用户	在建议的范围内使用该产品	不按照说明书使用产品。例如，在巧克力自动售卖机中插入不同国家的硬币
产品(嵌入式系统)	对需求、需要的功能以及产品的实时行为具有正确的理解是最基本的。这可以通过在客户与管理之间进行交流来实现。产品的范围、内容和目标都必须正确定义，期望达到的性能必须达成共识	缺少正确的产品说明书
过程	产品开发过程必须分成多个问题、组件和模块(还有数据结构和对象)，以完成为每个活动在活动图中所定义的活动。如何对该过程进行划分才能使它能够适应问题和人的需求？在开发生命周期过程中所采用的过程必须根据人和需要的产品进行选择。选择哪一个模型？第 7.2 节中 7 个给定模型中的任何一个、它们的一些变种或者它们的改版	错误划分和适应错误模型
项目	软件项目管理的目标是对项目进行组织以确保项目的成功。成功可以通过以下几点来确保：(i)认真计划和评估每个成员对产品的每个活动所花费的代价和时间(第 7.7.2 节)；(ii)清晰地定义项目测试点和质量检查点；(iii)设置有效的监控和控制机制；(iv)协调通信问题及其解决方案。项目的成功是通过如下几点实现的：(i)系统地工作以理解该问题；(ii)设置现实的活动图、清晰的对象以及合理的期望；(iii)通过跟踪人、产品和过程活动来维持正确的开发要素和继续进行跟踪；(iv)在适当的时候进行正确决策；(v)从每个成员前面的失败和成功中吸取经验教训	不正确的计划，不正确的费用评估，对成功与否不够关注，让人忙于一些与项目无关的活动

7.8.2 项目测度

有没有一种在项目开始的时候就能够测量项目大小的量级和所需要代价的方法呢？软件的项目测度有(i)KLOC(代码的千行数)；(ii)每 KLOC 对应的文档页数；(iii)每 KLOC 中出现的错误数；(iv)每 KLOC 中的缺陷数；(v)每页文档的开销。它们对以下几项进行评估：(a)每 LOC 的开销；(b)每个人每月出现的错误；(c)每人每月编写的 LOC。

示例 7-3

考虑汽车里的一个嵌入式系统。假设下面给出的是系统的外部接口设备和 C 程序。假设要开发一个系统来监控(i)表，(ii)引擎速度，(iii)行驶速度，(iv)里程表，(v)燃料控制，(vi)报警器，(vii)照明，(viii)自动变速箱，(ix)安全警告控制(为以后进行维修和维护而记录汽车的使用状况和机械修补的故障数据)和一些其他的活动。表 7-10 给出了项目中常见的 LOC 评估。表 7-11 给出了书写缺陷计划、生产率和费用估计的格式。

表 7-10 估计使用 C 语言编写代码的功能

按代码行(LOC)*计算的功能大小			
	最小	一般	最大
表	828	1254	1680
安全警告控制	120	168	217
引擎速度	320	399	479
行驶速度	435	557	680
里程表	958	999	1040
唤醒/休眠	200	252	304
燃料控制	360	376	392
看门狗	160	188	217
报警器	4640	4781	4922
冷却温度	435	537	640
照明控制	1611	1925	2240
自动变速箱	174	207	240
总计	10241	11643	13051

注意：

(1)*LOC 包括如下几项。

注释	100%
includes( )	100%
	!
不以半列结束的行	100%
空白行	0%
像宏这样的特殊语句	100%

注意：

因为这些评估，项目测度不起作用。为什么呢？因为它提前使用了研究估计，且开发小组一点都不理解当前项目中 C 算法的汇编代码部分所需要的代价的实质。

表 7-11　书写缺陷计划、生产率和费用估计的格式

LOC 计划的行缺陷率的功能大小			
	最小	一般	最大
第一次交付后认真回顾后得到的 bug 争论			
LOC 中行生产率计划中的功能大小			
	最小	一般	最大
每小时的 LOC			
估计负载和价格			
每天的装载时间			
每天计划的价格			

如何估算 LOC 呢？有一种面向功能的测度，称为 FP(function point，功能点)，对于项目中所有的 FP 来说称为 FFP(Full Function Point，全功能点)。功能点的计算是通过计数实现的，然后针对表 7-10 中的 3 列，将得到的计数值相应地乘以三个权重因子(简单、一般和复杂)。下面的功能点用于找出第 i 个计数值，然后计算总和。在括号中给出了 F(i)的用于与计数值相乘的三个权重因子。这些因子是根据大量项目中得出的经验而总结出来的，并且已经适用于项目测度了。

(i) 外部接口数(5、7、10)

(ii) 用户输入数(3、4、6)

(iii) 用户输出数(4、5、7)

(iv) 用户询问数(3、4、6)

(v) 文件数(7、10、15)

FP = 计数总和 * [0.65 + 0.01 * [F (i)]

每个 FP 的 LOC 取决于编程语言。对于 C 代码来说，每个 FP 的 LOC 等于 128。对于汇编代码，等于 320，对于 C++，等于 64。它应用于生产率的功能测度中，用来生成如表 7-10 这样的表格。

在进行数据通信、临界性能、代码重用、主文件在线更新和相关参数计数的时候，F 必须将复杂度调整数值进行合并。

注意：

根据 Pressman 的意思，软件项目管理是由人、产品、过程以及项目组成的。对于计划阶段程序代价的估计来说，项目测度是可用的。这种测度使用功能点和 FFP(Full Function Point，全功能点)来估计每人每月的 KLOC(代码的千行数)和生产力。

7.9　软件维护

软件开发过程的生命周期以交付而告终，但是其本身的生命却并不会就此结束。维护包含

到交付(或者对于嵌入式系统来说是写入到 ROM 中)之后在对象或者模块中进行的更改、删除和添加。维护需求与软件复杂度成反比，因为简单的系统几乎不需要维护。表 7-12 给出了维护系统时所需的维护类型。

表 7-12 软件维护的类型

类 型	活 动	何 时 使 用	何时不需要或者不可行
预防性维护	可以对系统进行周期性的检查和维护	硬件组件受到磨损，例如，一个打印设备系统，因此需要修改其接口	难于指定必须的预防性行为时
矫正性维护	矫正在系统使用领域的特定条件下所发现的偏差	客户的产品说明书不完整和客户理解不正确都会导致矫正性维护的需求	系统工作令人满意时
适应性矫正	让软件适应新的条件(裁缝为一个减肥者进行衬衫的适应性维护)	机器人必须适应关于其手臂和操作范围的变化。巧克力自动售卖机必须适应国家现在使用的新型硬币	复杂系统可能更加困难，或者与设计一个新软件相比适应的代价更大时
增强或者完善维护	开发小组按计划交付系统，然后发现另外一种工作效率更高的设计	开发者进行新的设计开发并使用新的工具，从而交付更好的系统	完善的过程永远不会结束，系统必须具有容错性能
系统重新设计	只交付一个永远都不能满足创新团队需求的系统。需要对先前开发的对象和组件重新进行设计	重新设计是另一种维护系统的方法	复杂系统很难重新设计

什么时候软件或者系统可以称为成熟了，不再需要进一步的维护了？IEEE 给出了一个称为 SMI(Software Maturity Index，软件成熟度指标)的标准，该标准回答了这个问题——IEEE 962.1 标准。当 SMI 达到 1.0 时，此软件就成熟了，不再需要进一步的维护。让软件分别进行 C_c、C_r、C_n 更改、删除和新建对象或者模块。假设 N_c 是当前发布的总的对象或者模块。C_c/N_c、C_r/N_c 和 C_n/N_c 是反映每个单元对象(或者模块)的变化、删除和添加程度的比例 r1、r2 和 r3。因此，SMI=(1-r1- r2- r3)。

注意：

应该对嵌入式系统进行管理，使之在确认和交付之后所需要的维护最小。但是，由于不正确的客户说明书必须矫正、不断发生变化的条件、适应的需求以及改进系统的需求或者系统的进一步完善，软件维护是必需的。在进行维护的过程中，复杂度较大的软件需要的代价更大。称为 SMI 的 IEEE 962.1 标准是用来衡量是否需要维护的。当 SMI 到达 1.0 时，该软件就成熟了，不再需要进一步的维护。

7.10 统一建模语言(UML)

第 5.8.1 节对下面的内容进行了阐述：

(1) 面向对象的语言在下列情况下使用(i)当需要重用程序中，或者很多应用程序之间普遍使用的对象或者对象集时，(ii)当有进行抽象的需求时，(iii)当能够通过层次化定义对象来创建新对象时。在对象内可以进行数据封装。

(2) 对象用其标识名(保存它的状态和行为的引用)、状态(它的数据、性质、域以及属性)和行为(操作、方法或者操作对象状态的方法)进行刻画。

(3) 对象是从类的实例创建得来的。定义逻辑上相关的组可以创建一个类。类对状态和行为进行定义。它的状态和行为具有内部用户级的域。并定义了处理域的方法。

(4) 然后类可以通过对组进行复制和使它具有功能来创建很多对象。每个对象都有其功能。每个对象都可以与其他的对象相互作用，按照定义的行为来处理状态。

(5) 一组类就给出了一个应用程序。

面向对象的设计也可以如上面一样进行。

(1) 当需要将已定义的软件组件作为对象或者对象组(可重用的组件)重用时，就需要进行面向对象的设计。新的组件可以从已有的软件中抽象出来。新的组件和对象的设计是通过对象继承和多态实现的。在设计好的组件或者对象中存在信息封装。

(2) 设计好的组件对象也是用它的标识名(保存它的状态和行为的引用)、状态(它对数据、性质、域以及属性的设计)和行为(方法或者可以对该设计的状态进行操作的方法)进行刻画。

(3) 新对象的设计是从一个设计好的类的实例创建的。

(4) 一个设计好的类可以通过对组进行复制和使它具有功能来创建很多的组件对象(设计)。每个设计都是一个功能设计。每个对象的设计都可以和其他的设计接口以处理已定义行为的状态。

(5) 那么一组类就为系统给出了完整的软件设计。

是否能为一般的系统找到一种统一的(通用的)建模语言，使得对系统进行面向对象的分析和设计是可行的，并且能够通过模型对系统进行抽象？UML 在很多设计或者过程中都普遍应用。我们可以使用相似的图形组来完成对下面的建模：(i)软件可视化；(ii)数据设计；(iii)算法设计；(iv)软件设计；(v)软件说明书；(vi)软件开发过程；(vii)工业过程。

UML 是一种建模语言。图 7-5(a)~(f)给出了 6 个 UML 基本元素的表示：类、包、原型、对象、匿名对象以及状态。概念设计建模可以采用 UML 方法。概念设计可以使用“用例图”、“对象图”、“顺序图”、“状态图”、“类图”以及“活动图”。

表 7-13 给出了它的元素。表 7-14 给出了 UML 图：“类图”、“状态图”、“顺序图”、“协作图”以及“对象图”。

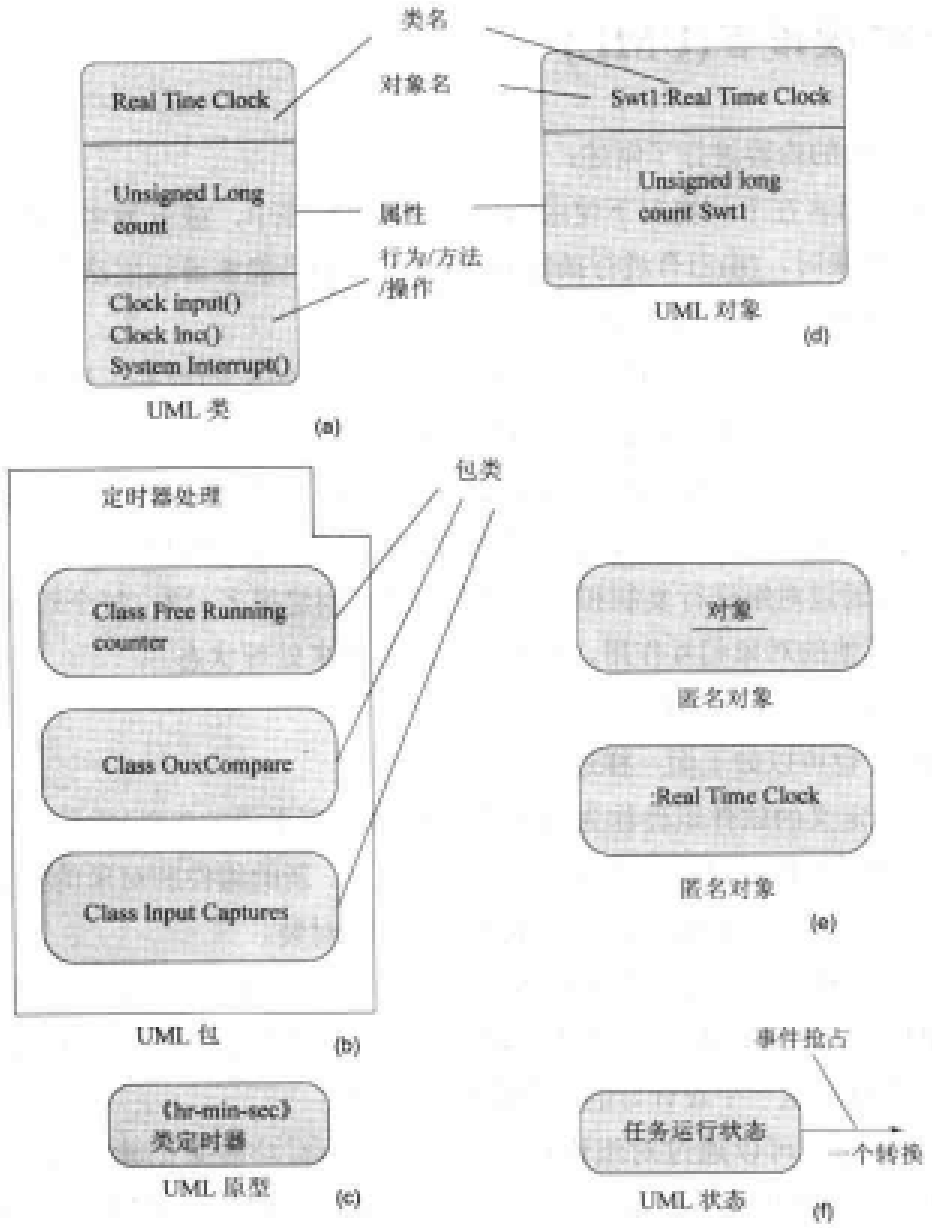


图 7-5 UML 基本元素的表示(a)类(活跃类、抽象类或者不活跃的类)；(b)包；(c)原型；(d)对象；(e)匿名对象；(f)状态

表 7-13 UML 基本元素

建 模 图	建模对象及内容	表 示 示 例
类	类对状态、属性以及行为进行定义。类也可以称为是活跃的或者抽象的。活跃类表示一个定义了状态、行为和属性，并且将其实例作为对象的线程类。一般地，当一个或者多个状态、操作或者行为还未完全定义，还处于抽象阶段或者当不是用它来创建对象而是用它的继承(扩展)创建对象时，类是抽象的	被分为类名、标识名、属性和操作的矩形框(活跃类的识别是通过在类的标识名之前添加前缀 active 实现的，请参考图 7-5(a))
包	将类和对象打包的集合	被分为类名、标识名、属性和操作的矩形框(图 7-5(b))

(续表)

建 模 图	建模对象及内容	表 示 示 例
原型	反复使用的元素中未打包的集合	后面跟着类标识名，且在两对开始和结束符中有原型标识名的矩形框。例如，<<SerialLine Driver >>(图 7-5(c))
对象	通过从类中复制状态、属性和行为来形成一个功能实体的类的实例	后面跟着分号和类标识名的对象标识名的矩形框(图 7-5(e))
匿名对象	没有标识名的对象	在分号和类标识名前没有对象标识名的矩形框(图 7-5(d))
状态	一个状态	用状态名作为标识名的圆角矩形框(图 7-5(f))

表 7-14 UML 图

建 模 图	建模对象及内容	表 示 示 例
类图	类图了类与类的对象之间的关系，层次关联以及类和对象之间的对象交互	矩形框表示类，末端为空心三角形的箭头表示类的层次。类可以用线连接。线上起点和终点处的数字表示一个类的对象与多少个其他类的对象相关联(图 7-6(a))
状态图	状态表示了一个结构通过转换实现的起始、终止、中间关联的模型，展示了具有关联转换的事件标签(或者条件)	实心圆点表示起点(请参考 Petri 网中做标记的地方)，箭头表示转换。箭头上的标签表示触发该转换的条件或者事件。圆圈内带一个实心点表示终点(图 7-6(b))
对象图	对象图定义了系统的静态配置。也给出了对象之间的关系	请参考图 7-6(c)
顺序图[1]	顺序图表示了对象之间的交互。顺序图也说明了状态的顺序	圆角矩形框表示对象标识名，矩形框表示状态，用箭头连接类。垂直箭头向下指示时间的推进(图 7-7(a)和(b))
协作图	协作图表示了状态或者对象交互的并发顺序	水平轴或者垂直轴朝右或者朝下表示时间的推进，并行的序列集表示并发。条件或者事件可以在箭头上标记(图 7-7(c))

1. 图 7-7 给出了 UML 的顺序图。图 7-7(a)表示了状态之间进行交互的顺序，图 7-7(b)给出了顺序图(例如，巧克力自动售卖机的状态顺序)，图 7-7(c)给出了协作图(并发多处理)。

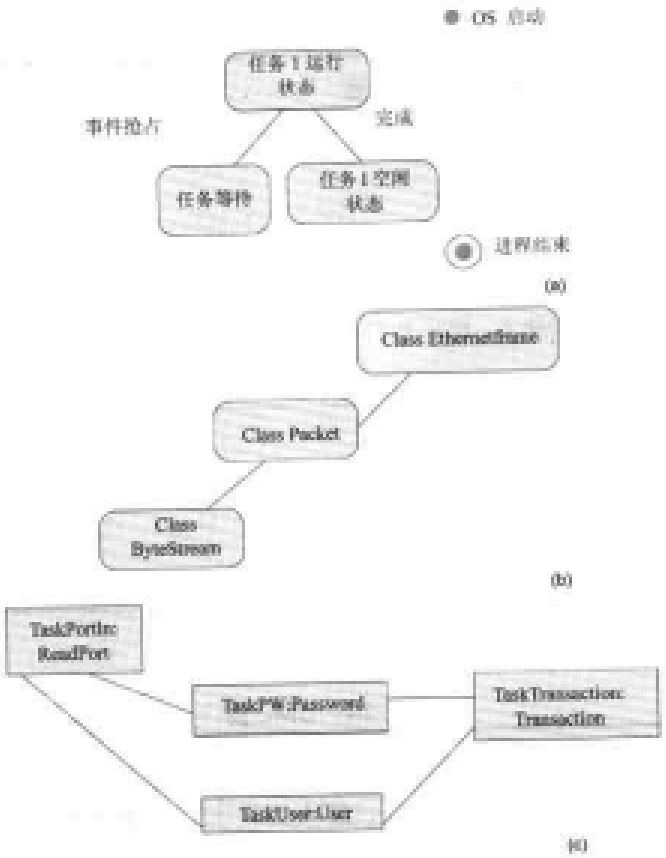


图 7-6 UML 图(a)状态图；(b)类图；(c)对象图

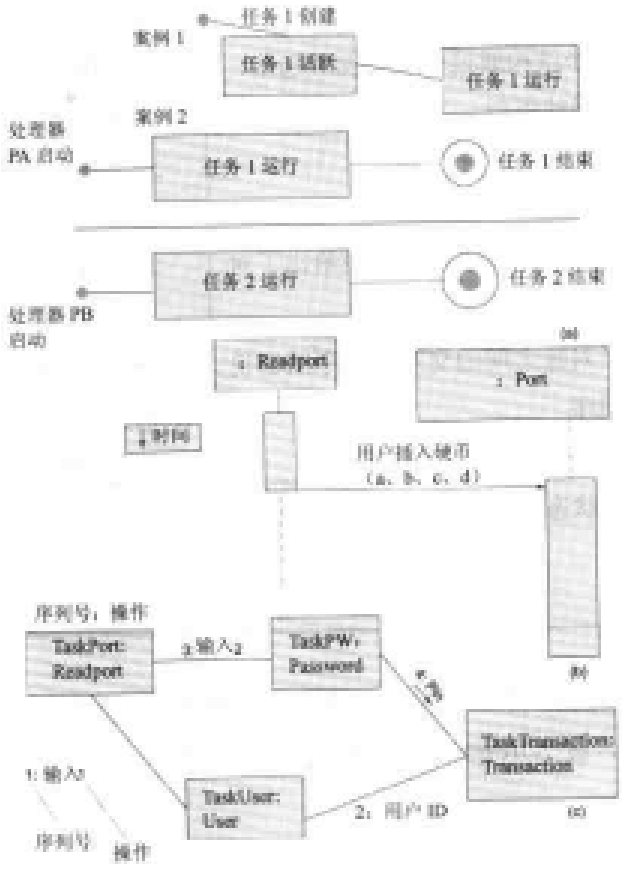


图 7-7 UML 图(a)状态之间的交互顺序；(b)顺序图(例如，巧克力自动售卖机的状态序列)；(c)协作图(例如，多处理并发处理系统)

示例 7-4

示例 5-5 给出了 C++程序代码。图 7-8 给出了其 UML 模型图。

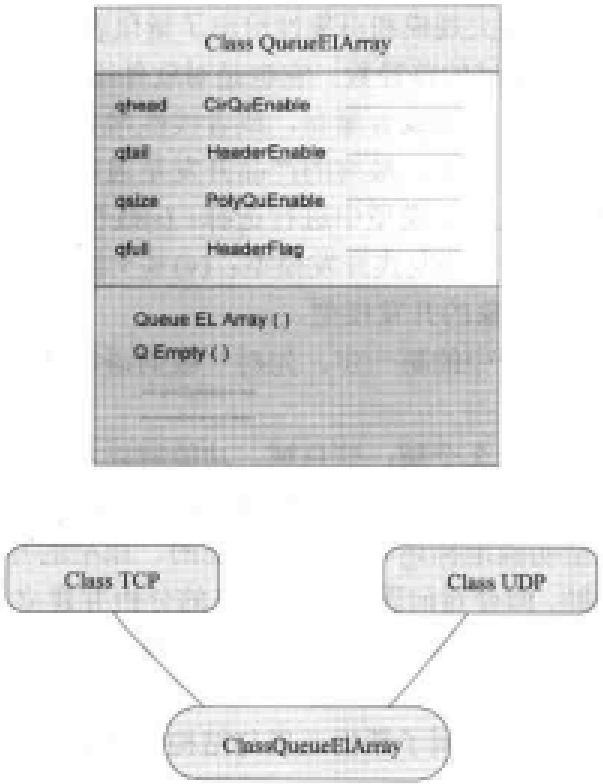


图 7-8　示例 5-5 中队列的 C++程序的 UML 模型图

可以查阅标准教材深入学习基于 UML 的表示方法和建模技巧(面向对象设计中有一本标准教材是 Addison Wesley 出版社于 1999 年出版的由 Grady Booch、James Rambaugh 和 Ivar Jacobson 合著的 *The Unified Modeling Language User Guide*)。

注意：

UML 允许：(1)顺序图也可以使用 Statechart 子层，或者用 Statechart 语言创建的模型。Statechart 是一种用于实现活动图、FSM 状态和状态转换、并发、同步、时序以及行为层次的语言。首先准备消息顺序图，然后，Statechart 从消息顺序图表示活动图。例如，Statechart 可以对两个 FSM 的两个并发活动进行建模。

(2)SpecCharts 是另一种用于描述规范和图的语言。它的模型很容易提供异常处理(自陷和中断)例程的实现。

注意：

UML 是一种用于下列过程的功能强大的建模语言，(i)软件可视化；(ii)数据设计；(iii)算法设计；(iv)软件设计；(v)软件说明书；(vi)软件开发过程。UML 的基本元素有类、包、原型、对象、匿名对象以及状态。UML 建模是通过类图、状态图、顺序图、协作图以及对象图实现的。

本章小结

- 在软件或者系统的开发过程中，必须遵守软件工程实践。
- 软件的复杂度对算法的最大规模和可靠性给出了量化表示。一种测量复杂度的方法称为圈复杂度。它等于需要测试的路径数。它也是对软件维护过程中困难程度的一种测度。
- 软件开发过程可以用生命周期或者瀑布，或者线性增长模型来表示。分析、设计、实现和维护是该模型的 4 个阶段。最终的产品开发是通过反复改进得到的。
- 过程生命周期模型可以从 7 个模型中进行选择。(i)线性顺序生命周期模型；(ii)RAD 模型；(iii)增量开发模型；(iv)并发式开发模型；(v)基于组件的模型；(vi)基于第四代工具的开发模型；(vii)面向对象的开发模型。
- 需求分析是软件开发过程中的第一步。先对一般的语句进行分析，然后从中找出具体的技术性规范。
- 所有的问题都可以建模为 3 个域：信息域、功能域以及行为域。解决问题的一种重要方法是首先将它分为几个部分，然后用描述问题关键的适当符号来表示它们。
- 通常，在早期就完全明确地描述问题是不可能的。原型的开发和原型说明书通常对问题的理解都很有帮助。即使对问题进行了最好的分析并建立了最好的模型也可以更改。因此，开发小组与客户之间的交流是很关键的，这样才能使得设计过程一开始就对需求有清晰的理解。
- 对需求进行分析之后，就得到了数据元素与结构、算法、行为、数据内容以及过程规范的说明书。
- 设计的第一步是
  (a) 抽象。每个问题组件都要首先进行抽象。例如，机器人系统问题抽象可以依据手臂和电动机的控制进行。
  (b) 软件体系结构的结构属性设计。
  (c) 额外的功能属性设计。
  (d) 相关系统的设计
- 使用模块化的设计概念。将系统设计分解为要实现的模块。模块应该具有允许它们后来进行组合(耦合或者集成)的性质。有效的模块化设计应该确保(i)功能的独立性，(ii)内聚性以及(iii)耦合性。
- 对于对象和组件的集成来说，接口设计也是至关重要的。
- 软件设计可以假设由以下 4 层组成：体系结构设计、数据设计、接口设计以及组件级设计。在设计中需要通过设计者和实现者之间的有效交流来不断地改进。
- 实现程序应该简单。函数应该一般化，不要受到某个输入条件或者事件的约束。这样可以使重用更加简单。程序应该按照程序结构设计规范加以实现。
- 测试的目标是找出错误，确认所实现的软件符合规范和需求。确认的意思是软件必须符合规范，且必须显示出与所有事件、条件和时间定义的行为相同。
- 重要的测试策略有白盒测试、黑盒(行为)测试、特定环境下的测试以及比较测试。
- 调试是通过使用断点、宏、来自标准输入的样例输出、指令模拟器以及实验工具完成的。

- 实时系统的开发过程存在一些复杂的问题。它对算法和 ISR 都有软件响应时间的约束。必须针对时序分析、IPC、异步事件以及系统每个状态的设置进行约束。还必须针对快速发生的事件进行测试。测试必须仔细，对于容错和故障恢复的测试也是至关重要的。
- 软件项目管理问题就是对人、产品、过程以及项目的组织管理。对于计划程序代价的阶段性估计来说，项目测度是可用的。一种项目测度是使用功能点和 FFP(Full Function Point)来估计每人每月的 KLOC(Kilo lines of code，代码千行数)和生产率。
- 维护的意思是在交付(如果是嵌入式系统，则是将软件安装到 ROM 中)之后，对对象或者模块进行更改、删除以及添加。维护需求与软件的复杂度成反比，因为较简单的系统几乎不需要维护。
- 嵌入式系统应该尽可能少地进行交付后的维护。不正确的客户说明书需要进行矫正性维护。条件发生变化需要适应性维护。系统改进或者系统完善需要完善性维护。预防性维护需要的是完善性维护。复杂度越高的软件需要的维护代价越高。
- IEEE 962.1 标准，也称为 SMI(Software Maturity Index，软件成熟度指标)是用来测量维护需求的。当 SMI 达到 1.0 时，该软件就成熟了，不再需要进一步的维护。
- UML 是软件开发过程中的一种功能强大的建模语言。它的基本元素是类、包、原型、对象、匿名对象以及状态。使用 UML 进行建模是通过开发类图、状态图、对象图、顺序图以及协作图来实现的。

关键词及其定义

- 软件工程实践(software engineering practices)：在软件开发的过程中使用软件工程方法的一种实践。
- 软件复杂度(software complexity)：算法最大规模与它的可靠性以及复杂度测度的量化表示。
- 圈复杂度(cylomatic complexity)：通过找出程序在执行期间能够访问的独立路径的数目进行软件复杂度测量。
- 软件开发过程(software development process)：开发软件的过程，通常通过分析、设计、实现以及测试来实现。
- 生命周期模型(瀑布模型或者线性增量模型)：一个用于在开发阶段的软件生命过程中对分析、设计、实现以及维护阶段进行描述的模型。之后还要对它循环改进直到最终的验证和确认。
- 线性顺序生命周期模型(linear sequential lifecycle Model)：一个用于开发软件的，分为如下 4 个阶段的线性顺序模型：分析、设计、实现以及测试。
- RAD(快速开发)模型：一个用于软件快速开发的模型。
- 并发式开发模型(concurrent development Model)：在开发的过程中同时并行地开发多个过程。
- 基于组件的模型(component based model)：将软件划分(分解)为组件，还要通过修改来使用可重用组件的一种方法。
- 第四代工具(Fourth generation tools)：用于开发的标准工具和包。例如，Java 和 CASE 工具。

- CASE 工具(CASE tool)：计算机辅助软件工程工具。
- 对象(object)：一种具有标识名、状态和行为的实体，它是某个类的一个实例。
- 类(class)：一个对数据、域、属性、状态、方法以及行为进行定义的逻辑组。
- 面向对象的开发模型(Object Oriented Development Model)：一个开发软件的过程，它首先将软件划分为组件，也可以通过修改来使用可重用组件。这样，每个组件由可重用的对象组成。重用一个对象是通过直接使用、继承、改变状态和行为来实现的。
- 软件分析(software analysis)：对需要的对象、组件、数据、数据结构、数据库、函数进行分析以及对行为、控制和过程规范的分析。
- 软件设计(software design)：在实现之前，从抽象阶段到最终细节的设计软件的过程。
- 软件抽象(software abstraction)：根据需求和规范进行的软件可视化和开发。
- 问题抽象(problem abstraction)：首先要对每个组件进行抽象。例如，机器人系统的问题抽象可以根据手臂和电动机的控制来进行。
- 需求分析(requirements analysis)：它是软件开发过程的第一步。首先，对一般的语句进行分析，然后从中找出具体的技术性规范。最后，所有软件组件、数据、控制和过程所定义的需求就确定了。
- 建模域(modeling domain)：任何问题都可以建模为由 3 个域组成：信息域、功能域以及行为域。
- 原型开发(prototype development)：开发一个原型，原型具有与最终系统相似的功能。例如，首先开发一个小实验规模的机器人原型，然后再开发工业级的机器人。
- 模块化设计(modular design)：将系统设计分解为要实现的模块。模块应该具有允许它们后来进行组合(耦合或者集成)的性质。有效的模块化设计应该确保(i)功能的独立性，(ii)内聚性以及(iii)耦合性。
- 软件设计层(software design layers)：体系结构设计、数据设计、接口设计和组件级设计。
- 体系结构设计(architecture design)：对结构单元和它们的行为及关系进行描述。
- 数据设计(data design)：对输入和输出的数据元素、结构和数据库进行设计。
- 接口设计(interface design)：为了集成对象和组件进行设计。
- 软件实现(software implementation)：采用 C、C++和 Java 等语言，使用代码设计工具对软件进行程序设计。
- 软件测试(software testing)：找出错误，直到所开发的软件通过验证和确认。
- 软件确认(software validation)：确认的意思是软件必须符合规范，必须使其行为与所有事件、条件和时刻所定义的行为相同。且所有的组件必须满足约束(存储器、能量需求和功能、例程以及驱动程序的响应时间)。
- 白盒测试(white box testing)：测试每条程序路径、每个循环、输入范围、事件、快速发生的事件以及约束(存储器、能量需求和功能、例程以及驱动程序的响应时间)。
- 黑盒测试(black box testing)：只测试行为和所有的功能。
- 特定环境的测试(specific environment testing)：针对特定的环境和组件进行测试。例如，小键盘、显示器、驱动程序以及 ISR。
- 比较测试(comparison testing)：与前面得到的结果相比较。
- 调试(debugging)：通过使用断点、宏、来自标准输入的样例输出、指令模拟器以及实验工具来跟踪错误源。

- 实时软件(real time software)：对其元素和组件具有响应时间约束的软件。
- 异步事件(asynchronous event)：在不相关的时间间隔内发生的事件。
- 软件维护(software maintenance)：维护包含在交付之后，对对象或者模块进行更改、删除以及添加。
- 软件开发人员(software people)：一组软件工程师或者系统开发工程师。
- 项目管理(project management)：通过开发人员(小组)对活动的每个阶段进行监控和协调来对产品的开发过程进行项目管理。
- 软件规划(software planning)：估计项目的代价，为开发过程找到一种适当的策略。
- 程序代价(program effort)：在程序设计的时候，人所付出的劳动量，它是通过项目测度进行衡量的。
- 功能点(function points)：它是通过对很多外部接口、用户输入、用户输出、询问以及文件等开发参数进行总的计数而得到的一个参数。每个功能点都会根据软件工程领域研究者以前得出的结果确定一个适当的权重。功能点针对三种类型——简单、一般以及复杂(最大)进行计算，以找出组成全功能点的 KLOC。
- 全功能点(full function points)：FFP 的意思是一个简单、一般以及复杂程序的功能点数目。
- KLOC：代码的千行数。
- SMI(Software Maturity Index，软件成熟度指标)：维护需求的一种测度。
- UML：一种功能强大的建模语言，它广泛应用于软件开发过程，尤其是用在设计中。

问题回顾

(1) 为什么在(i)线性顺序生命周期模型；(ii)软件生命周期模型以及(iii)软件开发的瀑布模型之间存在共性？

(2) 在嵌入式软件开发过程中使用 RAD 的优点和缺点是什么？

(3) 增量开发模型与并发式开发模型存在什么区别？

(4) 基于组件的模型和面向对象的开发模型之间存在什么相似之处？

(5) 使用基于第四代生成工具的开发模型(4GL 模型)如何简化开发过程？

(6) 行为规范的意思是什么？

(7) 过程的抽象在 OOD(Objected Oriented Design，面向对象的设计)中有什么作用？

(8) 过程规范是什么意思？

(9) 根据过程规范的详细设计，当程序的维护简单时，怎样使用程序设计模型、程序设计语言、RTOS 以及代码管理工具来实现软件？

(10) 在进行白盒测试、黑盒测试、正交测试以及比较测试时，需要进行什么活动？在一个实例系统中为每种测试列出这些活动。

(11) 抽象测试如何转化为详细设计？用移动电话中的嵌入式软件进行说明。

(12) 什么是调试的回溯方法？

(13) 为什么嵌入式系统需要最高质量的软件？

(14) 用一个例子来解释软件测试的测度？

(15) 何时必须进行嵌入式系统的适应性维护？

(16) 何时必须进行矫正性维护？

(17) 何时使用完善性维护，何时使用预防性维护？

(18) 叙述 Pareto 原则。

(19) 宏、测试向量、断点和样例输出文件集合怎样协助软件调试？程序中的宏对调试和系统测试有什么帮助？

(20) 在实时系统中进行测试需要什么？请列出来。

(21) 为什么实时系统的开发过程更复杂？

(22) IEEE 有一个称为 SMI(Software Maturity Index，软件成熟度指标)的标准，IEEE 962.1 标准。请解释该标准。

(23) 在 UML 当中如何表示一个匿名对象？

(24) UML 有什么特性？

实践练习

(25) 为练习 11 计算软件的 McCabe“圈复杂度”，然后再对第 6 章的练习(12)进行计算。

(26) 使用表 7-1，并参考第 11.1 节中的案例研究，描述巧克力自动售卖机的开发过程的 3 个阶段。

(27) 需要为数据、算法、行为、数据内容、控制以及过程书写规范。为第 11.3 节中自动巡航控制系统案例列出规范。

(28) 为什么嵌入式系统的处理过程必须要有可接受的最大延迟，而不是最小延迟？

(29) 在软件开发过程中，何时使用强力测试方法？

(30) 为什么实时系统的测试应该更加严格？给出一个实时系统的例子，其中很多超过时限的范围可以在一个可容忍的限度之内，但是吞吐量中的可变延迟是不可接受的。

(31) 为什么快速发生的事件在实时视频处理或者音频处理系统中是必需的？

(32) 容错系统和故障恢复系统是什么意思？

(33) 如何用第 7.8 节中的 4P 来描述示例 7-3 中定义的项目？

(34) 对每个项目测度举一个例子来解释(i)KLOC 代码的千行数)；(ii)每 KLOC 的文档页数；(iii)每 KLOC 中出现的错误；(v)每页文档的开销；(vi)每人每月出现的错误数；(vii)每人每月的 LOC。

(35) 用示例解释实现 PPP 通信的对象的标识名、状态和行为(请参考第 3.6 节中的练习(16))。

(36) 为第 11.1 节中的案例画出类图、状态图、顺序图、协作图以及对象图。

第 8 章　进程间通信与进程、任务和线程的同步

本章前所学内容

在前面的章节中已经对如下所示的重要问题进行了解释：

(1) 嵌入式软件可能非常复杂；在多处理器系统编程和实时系统软件开发中存在复杂的问题。

(2) 很多时候，程序具有大量的物理设备驱动程序和虚拟设备驱动程序、函数、进程(ISR、任务以及线程)，还具有几个必须在多个或单个处理器上并发处理的程序对象。

(3) 在嵌入式系统软件中，存在响应时间约束、任务优先级、功能存在巨大差异的组件的延时和时限等问题。

本章将学内容

下列程序实体：程序、进程、任务和线程的含义和概念是什么？它们之间有什么区别？嵌入式软件编程人员为什么要使用函数，他们在何时使用函数？他们为什么，且何时使用进程(任务或者线程)？

本章的第一个目的是通过理解下列内容来回答上述问题：

(1) 下列概念及其含义：(i)进程、任务和线程以及它们的状态，(ii)进程控制块，任务控制块以及线程控制块。系统中上下文、上下文切换、多处理、多任务以及多线程的概念。

(2) 函数、ISR 以及任务之间的区别，以理解它们在程序运行过程中更详细的细节。

一个进程产生的数据输出是如何传递给另一个进程的呢？换言之，两进程间的内部进程通信(IPC)是如何进行的呢？在进程的临界段数据(共享变量)被另一个在第一个进程结束以后开始执行的，具有更高优先级的进程操作和修改之前，如何保护它们？

请回顾图 6-9(a)、6-10 和 6-17，结点(运算集)等待某个 IPC 令牌激活计算。再回顾 Petri 网中某个转换等待的输入令牌。一个被阻塞(在等候)的或者是下一个进程(或任务)在从另一个进程接收到 IPC(在某个事件、通信或者令牌)之后是如何开始运行的呢？换句话说，在多处理和多任务系统中，进程和任务是如何同步的？

本章的第二个目标通过完成下面的任务来理解这些问题：

(1) 理解多个任务和例程之间必须共享数据的问题。当一个进程修改了某个变量而其他进程还没有完成时，必须找到合理的解决办法。

(2) 在解决共享数据问题和运行临界段代码的过程中，学会信号量的概念(如标志、互斥或者计数信号量)。

(3) 在使用有限缓冲时，用 P 或者 V 信号量函数解决经典的生产者和消费者问题。

(4) 在使用信号量时，解决优先级倒置问题和死锁现象。

(5) (i)使用中断关闭-打开机制，或者(ii)使用信号量进行内部进程通信的方法来解决共享数据问题。

(6) 在 RTOS 中为进程间通信编写详细的函数，来实现多个任务(进程)和调度程序(RTOS)之间的调度和同步。

(7) 使用诸如旋转锁等概念。

(8) 对于错误处理函数和执行时间最短的 IPC，使用信号(异常)。

(9) 使用信号量、队列、信箱、管道、套接字和远程过程调用来进行调度和同步。

RTOS 的任务调度将在第 9 章中进行阐述。本章的目的在于解释 IPC 的概念，RTOS 中为 IPC 编写的函数以及进程、任务和线程之间的同步。为了完全理解同步和通过 IPC 进行的任务并发处理，将分别在后面的第 10 章和第 11 章中进行阐述。

8.1 应用程序中的多个进程

8.1.1 进程

我们首先来理解进程(process)的含义和基本概念。进程定义了一个顺序执行的程序及其状态。进程在运行期间的状态是通过其状态(运行、阻塞或者完成)、控制块——称为进程控制块(PCB)或者进程结构——它的数据、对象以及资源等来表示的。进程将受操作系统(内核)调度而运行，操作系统将根据进程的请求(系统调用)来提供 CPU 的控制权。进程通过执行指令来运行，其状态的连续改变由最重要的参数——PCB 的程序计数器来监控。

进程是可以使用 DFG、CDFG(见 6.1 节)、FSM 或者 Petri 网(见 6.2 节)建模的运算单元，它按照使之能够在 CPU 上执行的调度机制，和使之能够使用系统存储器和其他系统资源的资源管理机制，受操作系统中某些进程的控制。根据 Gary Nutt(Gary Nutt 编著的 *Operating Systems A modern Perspective* 一书，第 2 版，由 Addison Wesley 出版社于 2000 年出版)的描述，程序可以定义为“当在某一数据集上执行时，定义进程行为的程序语句所组成的静态实体”。应用程序可以定义为由进程和不同状态下的进程行为所组成的程序。

注意：

进程是一个在操作系统调度内核控制下，运行在 CPU 上的运算单元。它在存储器上有一种进程结构，称为进程控制块。

8.1.2 任务

应用程序也可以定义为由任务和不同状态下的任务行为所组成的程序，这些任务和任务行为受使之能够在 CPU 上运行的系统软件(称为操作系统软件)的调度机制进程，以及使得任务能够使用系统存储器的系统软件的资源管理机制进程控制。

针对某个应用的嵌入式软件可以运行大量的任务，并且每个任务都需要一个 CPU 控制。

假定系统中只存在一个 CPU。

(1) 每个任务都是独立的，其在 OS 中被调度程序调度时，取得 CPU 的控制。调度程序控制并运行任务。一个任务不能调用另一个任务(这一点与 C(或 C++)函数不同，C(或 C++)函数是可以被另一个函数调用的)。一个任务是一个独立的进程。OS 只能阻塞某个正在运行的任务，并让另一个任务获得 CPU 的访问权，以运行服务代码。

(2) 每个任务都有一个 ID 号，就像每一个函数都有一个名称一样。ID 号，也就是“任务 ID”，如果其取值在 0~255 之间，它就是一个字节。ID 也是任务的索引。

(3) 在某一时刻，每个任务针对下列各项都有独立的(和其他任务不同的)数值：(i)程序计数器(如果允许任务访问 CPU，程序计数器的值就是任务的存储器起始地址)；(ii)虚拟堆栈的指针(调度程序允许任务访问 CPU 时，虚拟堆栈的指针用来保存参数的存储器地址)。这两个值是任务的上下文。只有在这个时候，CPU 的控制权才切换到其他的进程或者任务。在将控制权由程序切换回 CPU 时，或者操作系统对其状态进行解阻塞，使其再次进入运行状态时，必须获取上下文以运行该任务。

(4) 每个任务都通过一个 TCB 标识。TCB 是一个保存当前时刻程序计数器信息(指示该任务所执行的下一条指令的地址)、存储器映射、信号(消息)发送表、信号屏蔽、任务 ID、CPU 状态(寄存器，程序计数器和 CPU 栈指针)和内核栈(为执行系统调用等)的存储器块(注意：TCB 和过程控制块 PCB 类似)。

(5) 每个任务都可以有一个优先级参数。如果该优先级在 0~255 之间，那么它可以用一个字节表示(通常，值越高，该任务的优先级别就越高)。

(6) 每一个任务都有一个上下文(定义可参考 4.6 节)。它是一个记录，用于反映 OS 阻塞某个任务，并将另一个任务初始化为运行状态之前的 CPU 状态。因此，在任务运行的过程中，上下文是不断更新的，并且在切换到另一个任务之前，上下文会被保存。每个任务都还有一个初始的上下文，context_init。context_init 是任务的初始参数。这些参数包括：(i)指向启动函数的指针；函数的运行必须从该地址开始；(ii)指向上下文数据结构的指针，该结构包括处理器寄存器和状态标识；(iii)任务上下文还可能包含一个指向新任务对象(函数)的指针；(iv)还可能包含一个指向先前某个任务对象(函数)的堆栈的指针。每次调度程序阻塞某个任务而运行另一个任务时，上下文的切换行为必须进行。

(7) 每个任务可以编写成无限的事件等待循环。事件循环是一个持续等待某个事件发生的循环。在启动事件发生时，该循环从自己的第一条指令开始执行。然后是服务代码(或者为另一个任务的事件设定某个标识)的执行。最后，任务返回到启动事件，等待循环。

(8) 在任意时刻，每个任务都处于下列某一个状态：

a. 空闲状态：是指在任务初始化(准备执行)之前的事件等待循环状态。处于空闲状态的任务等待一个(或者多个)事件的发生。任务在执行完所有应当执行的代码之后，返回到空闲状态。

b. 就绪状态：已经退出事件等待循环，任务控制权已经被 OS 剥夺，并且使任务从起点(或者任务被阻塞的地方)开始进入准备执行服务代码的状态所必需的事件已经发生。

c. 运行状态：执行服务代码。

d. 阻塞(等待)状态：在将需要用到的参数保存到该任务的上下文之后，服务代码的执行被暂时挂起。例如，当某任务等待某个键盘或者文件输入时，该任务就是挂起的。然后调度程序会将其置于阻塞状态。

(9) 任务执行完毕(运行状态终结)之后，也就是说，所有的服务代码执行完毕之后，就返回到空闲状态或者就绪状态。

(10) 每个任务要么必须是一个可再入例程，要么必须有办法解决共享数据问题(可回顾第5.4.4节(ii)中阐述过的可再入函数)。

注意：

任务是在调度内核控制下，在 CPU 上运行的运算或者行为的集合。它也具有一个保存在存储器中的进程结构，称为任务控制块。它具有一个惟一的 ID 号。在系统中，任务具有如下的状态：空闲、就绪、运行、阻塞和完成。如果任务有无限等待循环——嵌入式系统设计中的一个重要特征，那么在完成该状态之后，会再进入到就绪状态。多任务操作是通过不同任务之间的上下文切换实现的。

8.1.3 线程

多处理操作系统运行多个进程。一个进程可能包含一个或者多个线程，线程定义了调度程序调度 CPU 和其他系统资源的最小单位。线程是一个进程或者进程中的子进程。线程具有自己的程序计数器、堆栈指针与堆栈、用于线程调度程序进行调度的优先级参数，以及在进行上下文切换时，装入处理器寄存器中的变量。在内核中，线程具有自己的信号屏蔽。在没有被屏蔽的时候，激活线程并且使之运行。当被屏蔽时，线程被放入处于挂起状态的线程队列中。一个进程的不同线程可以共享该进程的通用结构。多线程可以共享该进程的数据。

进程结构由表示存储器映射、文件描述和目录的数据组成。线程不需要具有这些数据。因此进程可以认为是重量级的进程和核心级受控实体。进程可具有虚拟存储器图、文件描述符、用户 ID 等进程结构。线程可认为是轻量级的进程和进程级受控实体(注意：该结构依赖于操作系统)。

如何将任务和线程区分开来？线程是在 Java 或者 Unix 中使用的一个概念。线程可以是进程中的子进程，或者应用程序中的进程。要调度多个进程，有一个形成线程组和线程库的概念。一个任务就是一个进程，操作系统进行多任务处理；任务是核心级受控实体，而线程则是一个进程级受控实体。任务在大多数方面和线程类似。线程不能调用另一个线程运行。任务也不能直接调用另一个任务运行。它们都需要适当的调度程序。多线程需要线程调度程序。多任务需要任务调度程序。在一个给定的操作系统当中，任务组和任务库的存在与否是不确定的。

注意：

线程是 Java 和 Unix 中的一个概念，在应用程序中，它是一个轻量级的子进程或者进程。

受操作系统内核控制。在存储器中，线程具有一个进程结构，称为线程堆栈。线程具有一个惟一的 ID。在系统中，线程具有下列状态：启动、运行、阻塞和完成。

8.1.4 通过函数、ISR 和任务的特征进行区分

为什么嵌入式软件程序员要使用函数、任务和线程？分别都在什么时候使用？当存在多个设备、函数、ISR 以及程序对象时，嵌入式软件可以建模为由多个任务组成，每个任务都由内核调度程序调度，并且使用 IPC 进行同步。线程在基于 Linux 或者 Unix 的嵌入式应用程序中使用。函数是进程、任务或者 ISR 的子单元。函数没有类似于 PCB 和 TCB 这样的结构，它只有一个堆栈，没有与内核中的任务调度程序或者线程调度程序相对应的调度程序。表 8-1 总结了函数、ISR 和任务的特征。

表 8-1 函数、ISR 以及任务的特征

函　数	ISR	任　务
1.函数是在任何例程中用于根据每次传递给它的参数来执行特定行为集合的实体。它可以从进程或任务中调用。函数是进程、任务或者 ISR 的子单元	所有中断源调用都是独立的。调用可以来自硬件或者软件。在执行某个 ISR 时，CPU 允许另一个具有更高优先级的 ISR 运行。它要么是 ISR 进程指令，要么是控制 CPU 上 ISR 调度的 RTOS。它依赖于内核如何管理 ISR(第 9.5 节)	任务是一个独立的进程，它的调用来自系统(RTOS)。RTOS 可以允许另一个优先级更高的任务执行。只有 RTOS(内核)才能控制 CPU 上的任务调度
2.每个函数都有一个程序计数器和在调用另一个函数之前必须用于保存上下文的堆栈	每个 ISR 都有一个堆栈，它用于保存程序计数器的当前值和在执行另一个具有更高优先级的ISR之前必须保存的其他数值。在执行不同的 ISR 时，堆栈不需要位于不同的存储器块中	每个任务都有不同的任务堆栈，它用于保存当任务从运行状态进入阻塞状态时必须保存的上下文(程序计数器的当前值和任务控制块中的其他数值(包括 ID))。每个任务在不同的存储块中都有一个进程结构(TCB)
3.函数可以调用另一个函数，并且可以嵌套另一个函数的调用。函数间存在不受调度程序或者操作系统控制的直接同步 (图 8-1(b))	根据给定的 OS 内核特性，对于硬件资源调用的响应和同步存在三种选择(参见下一章的图 9-4(a)、(b)和(c))	如果没有内核的帮助，任务之间就不存在同步。RTOS 内核一次只调用一个任务。任务何时运行和何时阻塞都完全处于 RTOS 的控制之下。RTOS 内核的调度有两种通用的方法：(i)协同调度；(ii)抢占式调度[参见 9.6 节]

图 8-1(a)给出了程序中函数嵌套调用的典型特征。图 8-1(b)给出了在进行嵌套调用时，程序计数器在不同时候的赋值。

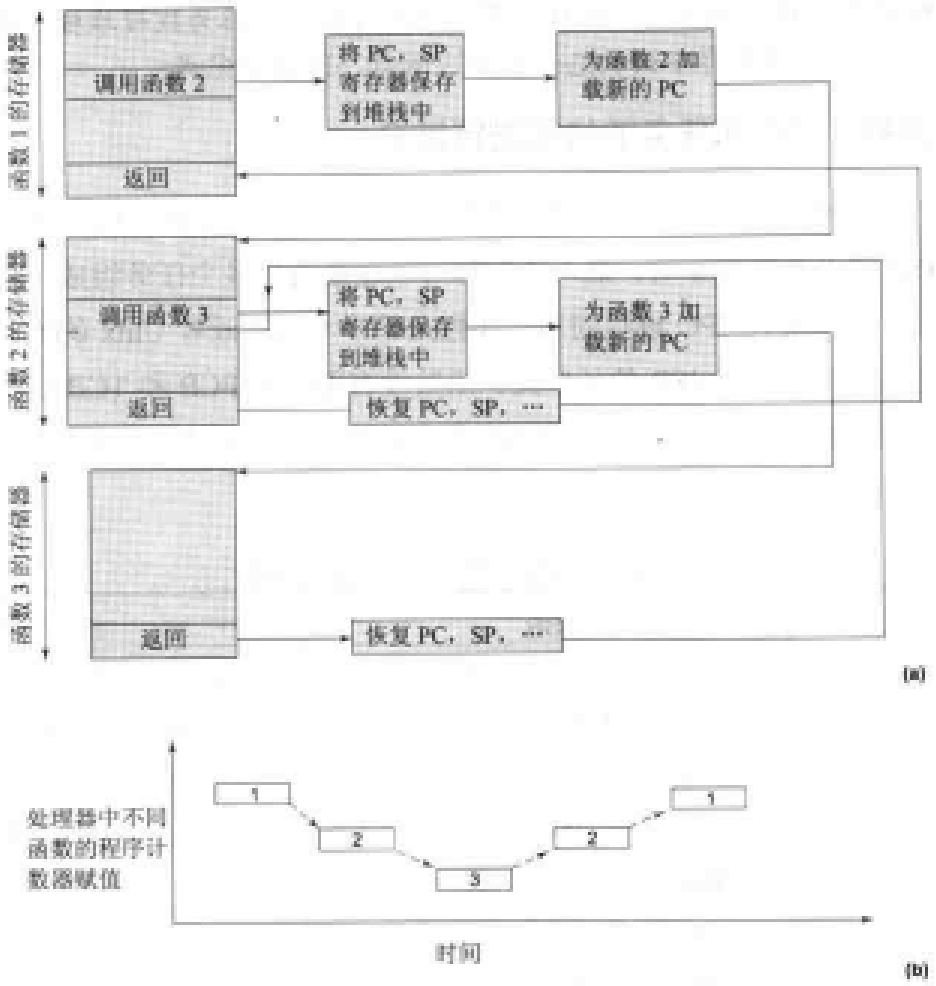


图 8-1 (a)程序中进行函数调用时的行为；(b)进行嵌套调用时，处理器中不同函数的程序计数器赋值

注意：

函数是在任意例程中都使用的一个实体，它根据每次传递给它的参数执行指定的指令集。函数可以从进程或者任务中进行调用，也可以存在于发生中断时执行的 ISR 当中。任务在被调度或者通过系统接收到 IPC 消息的时候执行。

8.2 多任务和多例程的数据共享问题

8.2.1 数据共享问题及其解决方案

数据共享问题可以解释如下。假定多个函数(或者 ISR、任务)共享一个变量。我们假定在某一时刻存在对该变量的数值的操作，并且在对其施加操作的过程中，仅有部分操作完成，还有一部分没有完成。假定此时产生了一个中断。如果此时还存在另一个函数也共享同样的变量，且前面的操作已经完成，该变量的数值可能与预期的不同。未完成的操作部分可以是下列情况。假设变量是 32 位的，而处理器是 8 位的。为了使用该处理器的 8 位 ALU，“C”编译器不得不将操作组装成四个 8 位的 ALU 操作。那么，假定在使用该编译器时，32 位的操作是非原子的。因此，当该函数与另一个函数共享该变量时，所调用的 ISR 或者另一个函数都能够改变该变量。在返回的时候，该变量的新数值将从堆栈装载到四个寄存器中。未完成的操作将按照寄

存器中新的数值执行。

考虑另一个例子，C 中的 printf 函数。在处理器所使用的某些 C 编译器中，可以对该函数进行非原子的处理。

再考虑另一个例子。假定有一个中断服务例程正在运行。假设存在一个在条件测试语句中使用的变量。且在另一个中断发生之前，使用该测试语句的某个 while 循环已经部分执行了。从中断服务例程返回之后，该变量为先前被中断的 ISR 返回了一个新的数值。这里，条件测试将展示一个不同的条件。

2.1.8 节描述了用于解决共享数据问题的原子操作。我们需要原子操作，因为(i)中断可以发生在指令周期结束的时候，但不能发生在高级指令周期结束的时候；(ii)DMA 操作可以在机器周期结束的时候进行，编译器或者程序不可以考虑这些原子级的细节。5.4.4 节的(ii)阐述了可再入函数的使用，它是解决共享数据问题的一种方法。

下面给出了一些措施，如果一起使用，就可以消除程序中由共享数据问题所导致的错误了。

(1) 对于从中断返回的变量，请在声明中使用修饰符 volatile。该声明将警告编译器，这些变量是可修改的，因为 ISR 没有考虑到该变量也被另一个正在调用的函数共享。

(2) 在中断之前需要完全执行的部分中对原子指令使用可再入函数。这个部分称为临界段。

(3) 将共享的变量放入循环队列中。需要使用该变量数值的函数总是从队列的前端将其删除(取走)。而另一些插入(写入)该变量数值的函数总是从队列的末端进行操作。在这里，如果存在大量的将变量数值存入或者取出队列的函数，而队列的大小不够用，就会出现问题。

(4) 在临界段开始执行之前关闭中断，在其完成之时打开中断。这是一种非常有用但却比较过激的选择。任何中断，即使是比当前临界函数的优先级更高，都必须关闭。采取这项措施的难点在于增加了中断的延迟周期，并且有可能会超过某个中断服务的时限。作为另一种关闭中断的措施，下面一节将介绍使用信号量来解决共享数据问题。软件设计者不应当采取这种过激的措施，在所有的临界段中关闭中断(注意：在用于汽车的操作系统中，进入任何临界段之前都关闭中断，以避免一切由于不正确使用信号量所产生的无意识行为。参见 11.3 节)。

上面这些措施——信号量的使用、禁止任务切换——一定能够完全消除多任务、多 ISR、多共享变量所导致的共享数据问题。每一个措施在解决问题方面都有其固有的好处。软件设计者必须根据解决问题的需要合理使用不同的措施。

注意：

当另一个更高优先级的任务完成了操作，并且修改了某个数据和变量之后，就会在系统中发生共享数据问题。解决方法有关闭中断机制、使用信号量以及使用可再入函数。

8.2.2 对任务或者任务的临界段使用信号量

(i) 单信号量

假定有两列使用同一条轨道的火车。当第一列火车 A 准备在轨道上出发时，火车 A 的信号(通知)被设置(为真或者获取)，另一列火车的信号 B 被复位(假或者释放)。类似地，考虑使用某个信号量，当用作事件标识的时候，它是一个二进制的布尔变量；或者说它是信号变量或者通报变量(当某个信号量的数值为 0 时，假定它已经被获取；当其数值为 1 时，假定还没有任务获取它，并且已经被释放，这样的信号量称为二元信号量)。

当执行一个临界段时，任务 A 会通知 OS 获取该信号量(引起注意)。操作系统返回值为获取(接受)的信号量。然后，任务 A 执行临界段的代码。先前收到通知的 OS 不会将该信号量返回给另一个任务 B。

图 8-2(a)给出了在任务 A 和任务 B 之间使用的信号量。图中介绍了在五个不同时刻 T0、T1、T2、T3 以及 T4 的五个顺序动作。图 8-2(b)以时间函数的形式展示了处于运行态的任务的时间图。它标记了 T0、T1、T2、T3 以及 T4 时刻的五个顺序动作。

RTOS 像下面那样使用信号量来进行多任务处理。假定 I、J、K、L 以及 M 是任务(参见 9.6.5 节对任务或者任务临界段信号量的使用)。

(1) 信号量的用途之一就是让任务在许多已经初始化、并且状态为“就绪”或者“运行”的任务中运行。因此信号量的使用，允许共享共用的资源 CPU。比如，当某个任务 K 要开始运行时，它会获取一个信号量。OS 阻塞任务 I、J、L 以及 M。这些任务将等待 K 释放信号量。

(2) 信号量的另一个用途在于可以作为任务之间传递的消息(或者事件标识)：通过 OS 的任务间通信。比如，任务 M 从某个端口接收消息字节，而另一个任务 I 在解码消息后，需要再次传送这些收到的数值。任务 M 需要任务间通信。当 I 完成了再次传送的时候，它会释放先前从 OS 中获取的信号量，然后 OS 通知任务 M，使其能够从该端口获取另一组字节。

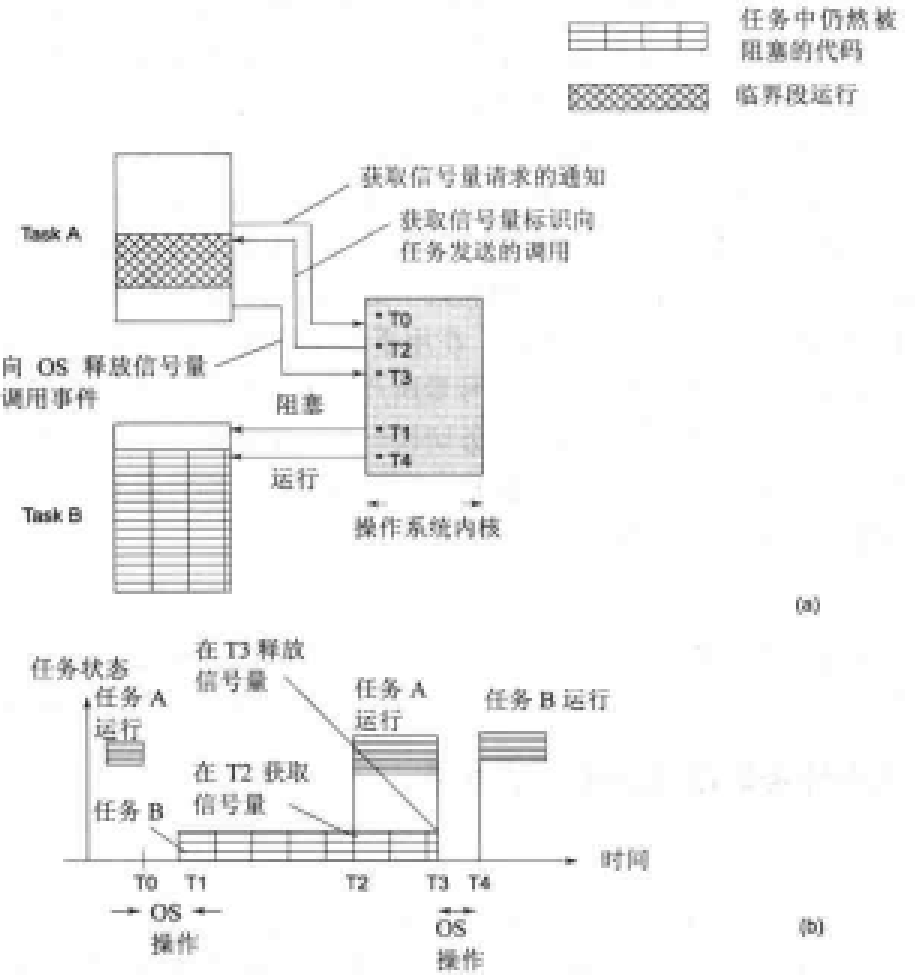


图 8-2 (a)使用任务间的信号量，A 和 B 给出了在 T0、T1、T2、T3 以及 T4 这五个不同时刻的五个顺序动作。 (b)作为时间函数在运行状态的任务时间图。它标记了五个不同时刻 T0、T1、T2、T3 以及 T4 的顺序动作，并给出了操作系统在任务 A 和 B 之间使用的信号量

注意：

信号量提供了让某个任务等待另一个任务完成的机制。它是一种同步并发处理操作的方法。当信号量被某个任务获取时，该任务就有权访问所需的资源了；当信号量被任务“释放”，该资源就被解锁。信号量可以用作事件标识或者资源键。资源键允许使用诸如 CPU、存储器、其他函数或者临界段代码这一类资源。

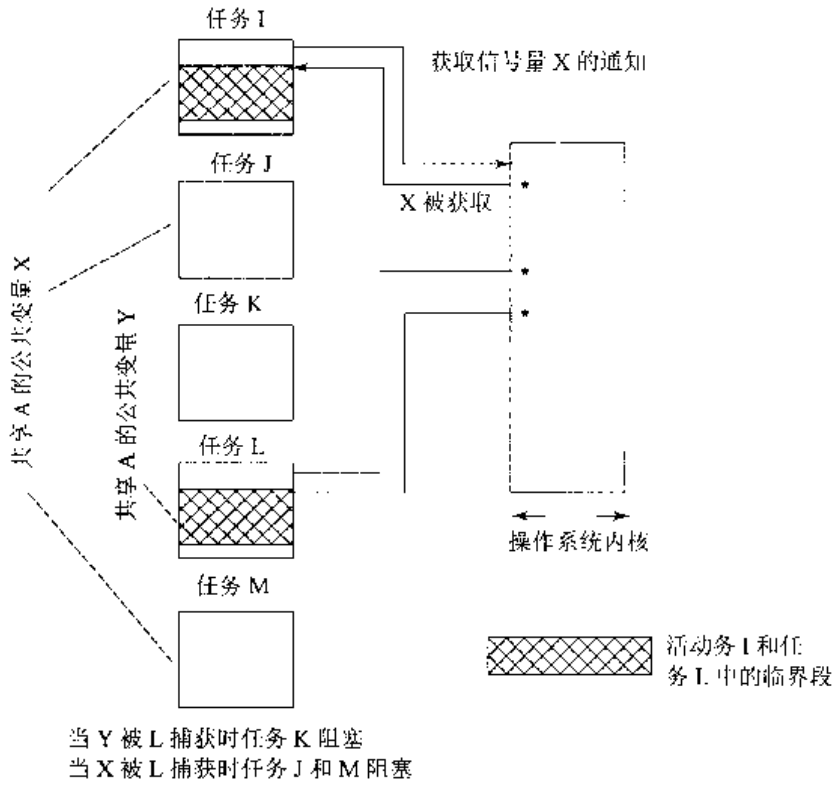


图 8-3　两个信号量 x 和 y 在任务 I~M 间的使用

(ii) 多信号量

假设有五列火车 i、j、k、l 以及 m，它们根据不同的目的地，使用两条轨道 p 和 q。首先假定只有一个信号。假设火车 i、j 及 m 共享轨道 p，火车 k 与 l 共享轨道 q。当第一列火车 i 将要在某条轨道上启动时，发给 i 的信号是旗子向下(释放)。其他火车 j、k、l 以及 m 接到通知，但并不释放。为什么所有火车 j、k、l 以及 m 都应该接到通知？不。两条轨道应该有两个单独的信号。当火车 i 移动时，火车 j 和 m 应当注意，当 l 移动时 k 应当注意。同样，考虑使用两个信号量，x 和 y。当执行临界段时，任务 I 通知 OS 捕获该信号量 x。操作系统向任务 J 和 M 返回信号量现在已经被获取的信息，任务 l 执行临界段的代码。由于 OS 已经从 I 接到关于信号量 x 已经被获取的通知，所以它不再捕获，也不会对任务 J 和 M 释放任何信号量。但是 OS 会对任务 K 与任务 L 之一返回另一个信号量 y。图 8-3 展示了任务 I 和 M 间两个信号量 x 和 y 的使用。

注意：

多个信号量可以在不同的任务组之间使用，不同的信号量组可以在不同的任务组之间共享。

(iii) 互斥体

互斥体使得两个或多个进程能够互相排斥地访问资源(CPU)。同一变量 sem_m 在不同的进程间共享。假设进程 1 和进程 2 共享 sem_m，且其初值为 1。

(1) 在 sem_m 减小、等于 0 之后，进程 1 开始执行，并获得对 CPU 的专有访问。

(2) 在 sem_m 增加、等于 1 之后，进程 1 结束；这时，进程 2 就可以获得对 CPU 的专有访问。

(3) 在 sem_m 减小、等于 0 之后，进程 2 开始执行，并获得对 CPU 的专有访问。

(4) 在 sem_m 减小、等于 1 时，进程 2 结束；这时，进程 1 就可以获得对 CPU 的专有访问。

sem_m 就像一个资源键。无论哪个进程的 sem_m 先减小到 0，哪个进程就能获得资源访问权，并禁止共享该键的其他进程访问。

注意：

互斥体是在某一时刻对两个任务赋予资源互斥访问权限的信号量。

(iv) P 和 V 信号量函数

信号量也可以是一个整型变量，除了初始化之外，该变量只能通过两个标准的原子操作 P 和 V 进行访问[P(用于等待操作)来源于荷兰语 Proberen，它的意思是“检验”。V(用于信号传送操作)来源于 Verhogen，它的意思是“递增”]。

(i) P 信号量函数：需要某种资源，如果该资源不可用则等待。

(ii) V 信号量函数：信号(信息)传递到操作系统，这时资源就释放给其他用户使用。

在使用标准 Posix 1003.1b——一个 IEEE 标准——的时候，RTOS 中使用了一种有效的同步机制——P 和 V 信号量(参考 9.8 节)。这两个函数对一个互斥信号量变量 sem_m 进行操作(示例 8-1 与示例 8-2)，它们的定义如下：

考虑 P 信号量。它是函数 P(&sem_1)，在进程中调用时，该函数使用信号量 sem_1 进行下列操作。

```
1. /* Decrease the semaphore variable*/
   sem_1 = sem_1 -1;
2. /* If sem_1 is less than 0, send a message to OS by calling a function
   waitCallToOS. Control ofthe process transfers to OS, because less than 0 means
   that some other process has already executed P function on sem_1. Whenever
   there is return for the OS, it will be to step 1. */if (sem_1 < 0){waitCallToOS
   (sem_1);}.
   Consider V semaphore. It is a function, V(&sem_2) which, when called in a
   process, does the following operations using a semaphore, sem_2.
3. /* Tncrease the semaphore variable*/
   sem_2 = sem_2 + 1;
4. /* If sem_2 is less or equal to 0, send a message to OS by calling a function
   signalCallToOS.Control of the process transfers to OS, because < or = 0 means
   that some other process is already executed P function on sem_2. Whenever
   there is return for the OS, it will be to step 3.*/
   if (sem_2 < = 0){signalCallToOS (sem_2);}
```

示例 8-1

使用带有互斥属性的 P、V 信号量函数

假设 sem_1 和 sem_2 是同一个变量 sem_m。当信号量函数 P 和 V 在两个进程——任务 1 和任务 2 中使用时，后一个函数作为互斥体，如下所示。

进程 1(任务 1)

```
while (true) {
/* Codes before a critical region*/
.
.
.
/* Enter Process 1 Critical region codes*/
P (&sem_m);
/* The following codes will execute only when sem_m is not less than 0. */
.
.
.
/* Exit Process 1 critical region codes */
V (&sem_m);
/* Continue Process 1 if sem_m is not equal to 0 or not less than 0. It means
that no process is executing at present. */
.
.
.
};
```

进程 2(任务 2)

```
while (true) {
/* Codes before a critical region*/
.
.
.
/* Enter Process 2 Critical region codes*/
P (&sem_m);
/* The following codes will execute only when sem_m is not less than 0. */
.
.
.
/* Exit Process 2 critical region codes */
V (&sem_m);
/* Continue Process 2 if sem_m is not equal to 0 or not less than 0. It means
that no process is executing at present. */
.
.
.
};
```

进程 1 和进程 2 共享了同一个变量 sem_m。其作用在于使两个进程能够对资源(CPU)进行相互排斥地访问。函数 P 运行之后，要么运行进程 1，要么运行进程 2。同样，函数 V 运行之

后，要么运行进程 1，要么运行进程 2。

图 8-4(a)给出了 P 和 V 信号量在任务、ISR 以及调度程序中的使用。当任务获取了信号量 P 时，如果之前 sem_m＝“真”(＝1)，则它将变为“假”(＝0)，因为 sem_m 不小于 0，任务继续执行。当任务执行 V 时，如果之前 sem_m 为“假”，则它将置为“真”，且任务继续运行，否则任务停滞并等候另一个任务的执行。图 8-4(b)给出了在使用 P 和 V 信号量时，进程或者函数的程序计数器的赋值。

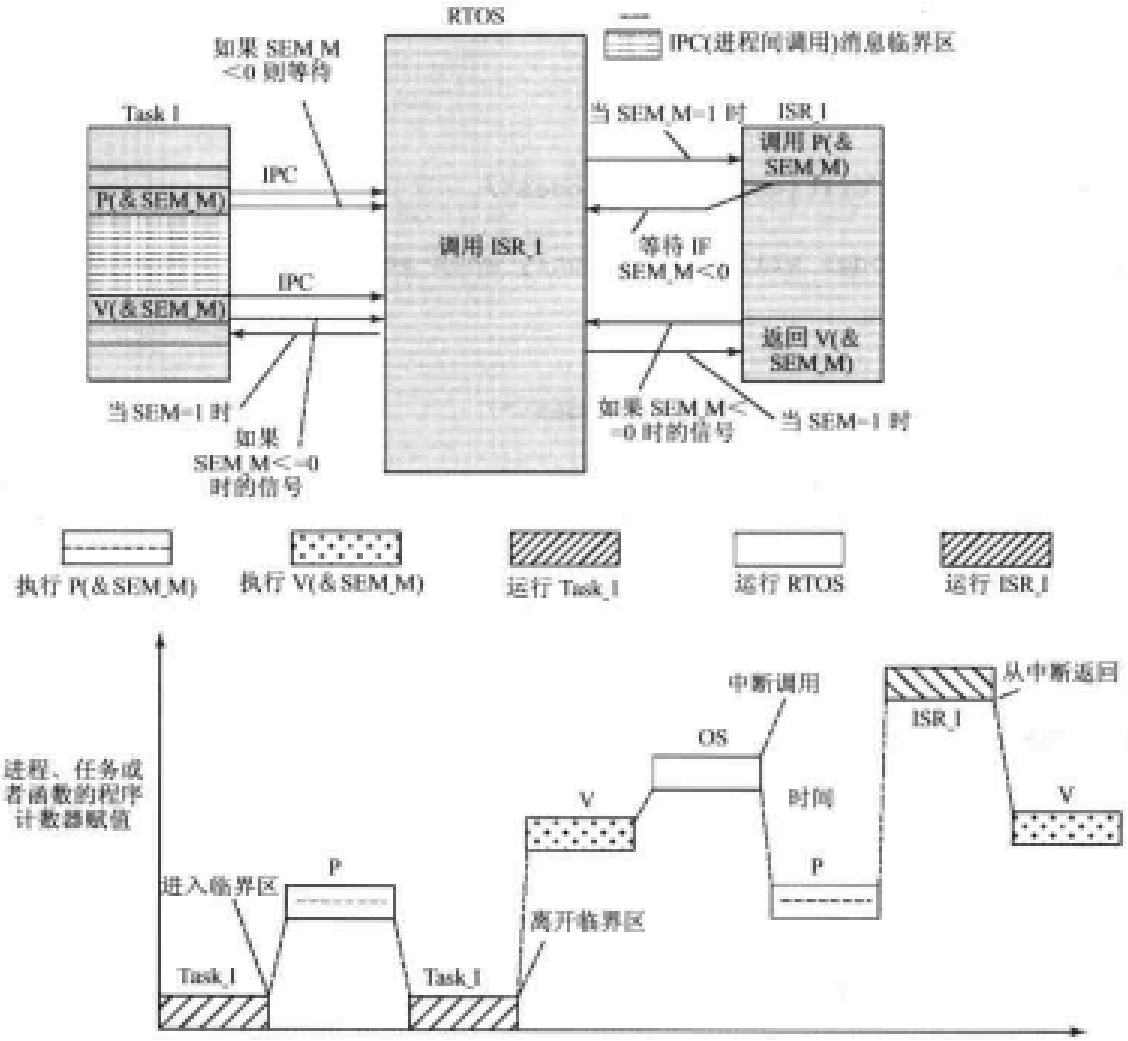


图 8-4 (a)在任务、ISR 和调度程序中使用 P 和 V 信号量；(b)使用 P 和 V 信号量时，进程或者函数的程序计数器赋值

注意：

P 和 V 信号量函数在被 IEEE 接受的 IPC 标准 POSIX 1003.1b 中，它可以用作一个互斥体；也可用作计数信号量操作。

(v) 计数信号量

计数信号量是无符号的整数。计数信号量的数值不仅控制任务代码的阻塞或者运行，而且控制与该任务共享数值的伴随任务代码的阻塞或者运行。计数信号量计算自己被获取的次数。当被某个任务获取时，它会加 1，而当被某个(同一个或者另一个)任务释放时，它会减 1。某一

时刻的值反映了它被获取的次数和被释放的次数之差。信号量使得其中一个任务等待执行代码，或者等待某个资源，直到令牌积累到必需的数目。使用计数信号量的一个例子是巧克力自动售卖机。只有收集到足够多的硬币时，它才会交付巧克力(参见 11.1 节中的案例研究)。当只有一个投币口，且当机器只接受一种类型的硬币时，直到投入足够数目的硬币，才会执行另一个等待任务。当存在多种硬币时，还需要事件的数目，且交付巧克力的任务将保持在阻塞状态，直到获得足够的硬币。

注意：
计数信号量是一个能够“获取”或者“释放”多次的信号量。

(vi) 带有计数信号量特性的 P 和 V 信号量函数

假定一个任务产生输出供另一个任务使用。一个任务可以有一个单独的计数信号量。
考虑三个例子：
(i) 某任务将字节传送到 I/O 流以填充流中可用的位置；
(ii) 某进程将一个 I/O 流“写”入打印缓冲区；
(iii) 生产巧克力的任务正在执行。
在例(i)中，另一个任务从填满的位置处读取 I/O 流字节，并创建空位置。
在例(ii)中，在进行缓冲区读取之后，从 I/O 流的打印缓冲区开始打印。打印之后，缓冲区中就创建了更多的空位置。
在例(iii)中，消费者正在消费生产的巧克力，且创建更多的空位置(用来存放巧克力)。
任务互锁操作问题通常称为生产者-消费者问题。如果在流缓冲区中没有空位置，那么任务不能向 I/O 流进行传送。如果在打印缓冲区没有空位置，则任务不能从存储器向打印缓冲区写入。如果在消费者端没有空位置，则生产者不能继续进行生产。
生产者-消费者问题程序是一个经典的同步问题，也称为有限缓冲问题程序。这里，一个或多个生产者(任务进程或者线程进程)创建数据输出，然后这些数据由一个或多个消费者(任务或者进程)处理。来自生产者的数据输出通过使用共享存储器和计数信号量(消息队列或者邮箱)的 IPC，传递给消费者处理(见 8.8 节)。
假设存在两个进程(任务 3 和任务 4)。如示例 8-2 所示，函数 P 和函数 V 操作两个共享的计数信号量，sem_c1 和 sem_c2。

示例 8-2
将函数 P 和函数 V 用作计数信号量。
假定两个进程使用了 P 信号量函数和 V 信号量函数以及两个任务——任务 3 和任务 4。假定 sem_c1 和 sem_c2 是两个计数信号量变量，且分别表示由进程 3 创建的填充位置数和进程 4 所创建的空位置数。函数 P 和函数 V 进行如下操作。

进程 3(任务 3)

```
while (true) {
/* Codes before a producer region*/
.
.
.
```

```
/* Enter Process 3 Producing region codes*/
P (&sem_c2);
/* The following codes will execute only when sem_c2 is not less than 0. */
.
.
.
/* Exit Process 3 producing region codes */
V (&sem_c1);
/* Continue Process 3 if sem_c1 is not equal to 0 or not less than 0. */
.
.
.
};
```

进程 4(任务 4)

```
while (true) {
/* Codes before a consumer region*/
.
.
.
/* Enter Process 4 Consuming region codes*/
P (&sem_c1);
/* The following codes will execute only when sem_m is not less than 0. */
.
.
.
/* Exit Process 4 consuming region codes */
V (&sem_c2);
/* Continue Process 4 if sem_m is not equal to 0 or not less than 0. It means
that no process is executing at present. */
.
.
.
};
```

两个信号量，sem_c1 和 sem_c2，由进程 3 和进程 4 共享。当进程 3 执行时，它首先减少进程 4 的空位置数。当进程 3 完成了生产之后，它将增加进程 3 中填充位置的数目。当进程 4 进行消费时，它首先减少进程 3 中填充位置的数目。当进程 4 完成消费时，它增加进程 4 中的空位置数目。要么执行了函数 P 之后，进程 3 产生输出，要么执行了函数 P 之后，进程 4 消费(使用)输入。也可以是，在执行函数 V 之后，进程 3 产生输出，或者执行函数 V 之后，进程 4 消费输入。

注意：

P 和 V 信号量函数可用于计数信号量操作，而且可以用在程序中解决有限缓冲区等类似问题。这些都能用在解决经典的生产者——消费者问题的程序中。

(vii) 共享数据问题的消除

信号量的使用并没有完全消除共享数据问题。在所有临界段中，软件设计人员可能无法选

择过激的方法，通过使用信号量来关闭所有临界段中的中断。当使用信号量时，RTOS 关闭中断。另外，还可以使用任务切换标识[8.2.3 节]，以避免使用信号量时可能产生的下列问题。

(1) 两个信号量的共享造成死锁问题(参考 8.2.3 节)。

(2) 假定所获取的信号量永不释放？那么应当有一些超时机制，超过时限之后，将产生错误信息或者采取适当的措施。超时和看门狗定时器操作有一定程度的相似性。一旦超时，看门狗定时器就会将处理器复位。这里，在超时之后，OS 报告错误并运行错误处理函数。如果没有超时机制，那么 ISR 在最坏情况下的延迟可能会超过时限。

(3) 信号量未被捕获，另一个任务使用了共享变量。

(4) 当一列火车接收到一条来自错误轨道的信号时，会发生什么？当使用多个信号量时，如果非计划中的任务捕获了该信号量，就会出现问题。

(5) 可能存在优先级倒置问题(参考 8.2.3 节)。

8.2.3 优先级倒置问题和死锁情况

假设任务的优先级具有某种顺序，使得任务 I 的优先级最高，任务 J 次之，任务 K 最低。假定只有任务 I 与任务 K 共享数据，而 J 不与 K 共享数据。再假设任务 I 和任务 K 单独共享一个信号量，J 无共享。为什么只有几个任务共享一个信号量？为什么不能所有的任务共享一个信号量？原因在于，如果某个任务捕获信号量时将所有任务都锁定，就会使得最坏情况下的延迟变得很大，且可能超出时限。只有当共享资源的任务所耗费的时间相关时，最坏情况下的延迟才会变小。现在考虑下面的情况。

在某时刻 t_0，假定任务 K 捕获了一个信号量，它不会阻塞任务 J 和任务 I。这是因为只有任务 I 和任务 K 共享了该信号量数据，而 J 并没有共享。考虑 K 和 I 之间选择性共享所产生的问题。在下一个时刻 t_1，假设在发生中断时任务 K 首先就绪。现在，假定在下一个时刻 t_2，发生中断时，任务 I 就绪。在该时刻，K 处于临界段中。因此，由于 K 处于临界段，任务 I 此时无法启动。那么，如果在下一个时刻 t_3，某个行为(事件)引起一个未阻塞的、优先级比 K 高的任务 J 运行。在时刻 t_3 之后，正在运行的任务 J 并不允许最高级别的任务 I 运行，因为 K 没有运行，这样 K 就不能释放其与 I 共享的信号量。而且，任务 J 的设计可能造成即使任务 K 释放了信号量，也不会让 I 运行(J 总是像它处于临界段中那样运行代码)。这样，就好像 J 具有比 I 更高的优先级。这是因为当让 J 运行时，K 进入临界段并获取了信号量之后，K 并没有共享 I 的优先级信息——任务 I 的优先级要高于 J。如果 K 等待 I，但是 J 不等待，且在 K 尚未完成临界段代码的时候，J 运行，那么另一个优先级更高的任务 I 的优先级信息也应当被 J 暂时继承。但这种情况不会发生，因为给定的 RTOS 在这种情形下，并不提供暂时的优先级继承。

上述情形也称为优先级倒置问题。RTOS 必须为优先级倒置问题提供一个解决方案。一些 RTOS 提供了这些情况下的优先级继承，因此在使用的时候就不会出现优先级继承问题。有关使用互斥体进行资源共享的细节可以参考 8.2.2(iii)节。互斥体应当是一个相互排斥的布尔标识，可以阻止临界段产生中断，通过这种方式可以消除优先级倒置问题。一些 RTOS 自动提供了互斥体，所以不会出现优先级倒置问题。互斥体的使用也类似于 8.2.2 节(i)中所定义的、在其他一些 RTOS 中使用的信号量，但信号量并没有解决优先级倒置问题。

考虑另一个问题。假定出现下列的情况。

(1) 假设任务的优先级关系是：任务 H 优先级最高，任务 I 次之，任务 J 最低。

(2) 假设有两个信号量，SemFlag1 和 SemFlag2。这是因为任务 I 和任务 H 只有一个通过 SemFlag1 共享的资源，而任务 I 和任务 J 通过两个信号量 SemFlag1 和 SemFlag2 共享了两个资源。

(3) 假设在t_0时刻，J 发生中断，首先捕获两个信号量 SemFlag1 和 SemFlag2 并运行。

假定在下一个时刻t_1，由于这时任务 H 具有更高的优先级，它进行中断，任务 I 和 J 在其后获得信号量 SemFlag1，并因此阻塞了 I 和 J。在t_0到t_1的时间间隔内，任务 J 的运行过程中，SemFlag1 被释放，SemFlag2 并没有释放。但后者并不会造成影响，因为任务 I 和 J 并不共享 SemFlag2。在某个时刻t_2，如果 H 释放了 SemFlag1，假设任务 I 捕获了它。即便如此，它还是不能运行，因为还要等待任务 J 释放 SemFlag2。下个时刻t_3，任务 J 等待 H 或者 I 释放 SemFlag1，因为需要该信号量进入其临界段。在时刻t_3之后，任务 I 和 J 都无法运行。这是一个在 I 和 J 之间建立的循环依赖关系。

上面的情形也称为死锁。当 H 产生中断时，任务 J 在退出运行态之前，应当已经被放置于队列的前端，使得在后来，它应当首先获得 SemFlag1，任务 I 处于队列中下一个位置以捕获同一个标识，这样，死锁就不会发生(参考 8.3.3 节的消息排队)。

使用互斥体解决了某些 RTOS 的死锁问题。它的使用可能与 6.2.2 节(i)和(ii)中定义的其他 RTOS 中的信号量类似，那时，互斥体并不能解决死锁情形。

注意：

在使用信号量时，某些情况下会出现优先级倒置和死锁(循环依赖)问题。某些 RTOS 提供了该问题的解决方法，即使用信号量来确保这些情形不会在多任务操作的并发处理中出现。

8.3 进程间通信

IPC(inter process communication，进程间通信)是指进程(调度程序、任务或 ISR)通过设置或者重置标识或值产生一些信息，或者产生输出，以使得另一个进程注意或者使用它(回顾图 6-9(a)、图 6-10 以及图 6-17 中的等待结点(运算集合)及 6.2.2 节的等待转换)。多处理器系统中的 IPC 用来产生关于在一个处理器上完成的运算集合的信息，并让其他处理器注意到它。

能通过调度程序发送用于另一个任务处理的输出数据(一条长度已知、带有标题或者不带标题的消息)吗？一种方法是使用全局变量。使用全局变量会带来两个问题。一个是共享数据问题(第 8.2 节)。另一个问题是全局变量无法阻止(封装)消息被其他任务访问。因此，可以使用下列 IPC：

(i) 信号

(ii) 共享一个公共缓冲区的任务之间的任务间通信中使用的信号量(如标识、互斥体)或者计数信号量

(iii) 队列、管道或者邮箱

(iv) 套接字

(v) 用于分布式处理的远程过程调用(remote procedure call，RPC)

IPC 的一个简单例子是运行 print 函数的任务。调度程序应当让其他的任务共享该任务。打印任务能够通过使用 IPC 而在多个任务间共享。当打印机可用时，将产生来自于打印任务的 IPC，且调度程序会注意到它。其他的任务则通过调度程序注意到这一点。另一个任务产生 IPC 来获取该打印任务的访问。

考虑另一个例子。它是一个用于对输出进行多行显示的任务，且另一个任务用于在最后一行显示当前时间。当多行显示任务完成了倒数第二行的显示时，将产生一个来自于显示任务的

IPC，且调度程序将注意到它。另一个任务——持续更新时间——能够为当前时间捕获并产生一个 IPC 输出。

在客户机-服务器网络中，也会产生对 IPC 的需求，从而产生对进程间通信的需求。

RTOS 方便了 IPC 的使用：信号、事件、计数器、互斥体信号量、邮箱、队列、管道以及套接字。RTOS 也方便了 RPC。上述 IPC 将在后面的 8.3.1~8.3.6 小节中进行阐述。

注意：

进程间通信(Inter process communication，IPC)是指进程(调度程序、任务或者 ISR)通过设置或者重新设置标识或数值产生一些信息，或者产生输出，以使得其他的进程注意到它，或者在操作系统的控制下使用它。

8.3.1 信号

进行消息传递的一种方法是使用“信号”。“信号”提供了最短的消息。信号是进程对于 IPC 的一个 1 位输出。使用它的一个优点在于它与其他的信号量不同，占用 CPU 的时间最短。信号是一个标识，在进行同步的 IPC 函数中使用。信号是在发生硬件中断时标识在寄存器中的软件等效。参考第 3 章中介绍的软件定时器。除非被信号屏蔽，信号允许信号处理函数执行，就像硬件中断允许 ISR 执行一样。

信号与信号量标识不同。信号量标识只能当作事件标识使用，让另一个任务进程阻塞，或者针对给定的代码段锁定特定任务进程的资源。信号是另一个中断服务进程为信号处理函数共享和使用的标识。某个进程发送的信号强制另一个进程发生中断，并且如果信号没有被屏蔽则捕获该信号(使用没有被禁止)。由于信号可能会打乱正常的调度和优先级继承，它只能由优先级非常高的进程处理。它还可能引起可再入问题。

信号的一个很重要的用途是处理异常(异常是在特定运行时条件下执行的进程)。信号在任务运行期间报告一个错误(称为“异常”)，然后让调度程序初始化一个错误处理进程或者函数。错误处理任务可以处理其他任务的不同错误注册。这种处理是通过使用 ISR 处理函数完成的。

参考 Gary Nutt 编著的 *Operating Systems - A modern Perspective*(Addison Wesley 出版社，2000 年)。Unix 和 Linux 操作系统使用了各种各样的信号，对不同的事件使用了 31 种不同类型的信号。对于 VxWorks 的信号，请参考 10.3.4 节中的(1)。

注意：

用于消息传递和同步进程的最简单的 IPC 是对“信号”的使用。“信号”提供了最短的消息。信号用于初始化异常和错误处理进程。

8.3.2 信号量标识或者互斥体用作资源键(用于进程的资源加锁和解锁)

信号量[见 8.2(i)和(ii)节]介绍了信号量标识(令牌)的使用。作为事件标识，信号量便于在任务间通信。任务间通信是用于正在运行的任务 N 或者 ISR 发生事件时，通知(通过调度程序)某个等待的任务 M 进入运行状态。在一个地方产生的信号量令牌在另一个地方是可用的。在运行位置进入临界段之前，标识 Sem_NTakenFlag 变为真。

8.2.2 节(iii)和(iv)介绍的信号量被用作互斥体(彼此专用)来访问代码集合(或者线程、进程)。互斥体使得优先权倒置问题[见 8.2.3 节]在某些 RTOS 中没有解决，但在其他 RTOS 中已经解决了。

使用互斥体的处理器在任务的临界段加锁。假设某任务中存在一个临界段。m1_mutex 是由函数创建的用于捕获互斥数值的互斥体。可以编写两个C函数——进入该段前的taskCriticalSec_mutex_lock(&m1_mutex)和退出该段前的 taskCriticalSec_mutex_unlock(&m1_mutex)。另外，信号量P和信号量V与互斥体一起使用(参见6.2.2节(iv))。

将二元信号量用作标识或者整型变量，互斥体提供了一种资源键，并且有利于任务与调度程序之间的通信。键用于获取对某种资源(一段代码、临界段代码或者函数)的访问。它解决了共享数据问题或者共享资源问题。如果对某种资源的访问被阻塞，它就按照资源加锁机制工作。在其他任务已经捕获了信号量期间，任务的阻塞周期可以通过定义超时数值加以限制。

以类似的方式使用键，在超出某个预先设定的时间间隔之后，也可以阻塞费时的ISR。该ISR捕获某个信号量，并在超时之后释放，以使其他的ISR运行。这提高了其他ISR的ISR响应时间。

假设与某个ISR应当占用的时间相比，临界段占用的时间过长。预先设定捕获信号量的最大时限会有所帮助。在其他的加锁情况下，通过使用互斥体或者P和V信号量，在任务切换到另一个任务或者ISR时可以锁定某种资源。

在下列情况下，优先级高的资源不应该通过阻塞正在运行的任务来锁定其他的进程。假定某个任务正在运行，并且只剩下少量时间可用于其完成操作。与阻塞该任务并进行上下文切换的时间相比，剩下的运行时间更短。在某些调度程序中，就有了旋转锁(spin lock)的概念。

在上面描述的情况中，旋转锁是一个非常有用的工具(参考 Bil Lewis 和 Daniel J.Berg 共同编著的 *Multithreaded Programming with Java*，Sun Microsystems Inc 出版，2000年)。为某个任务锁定处理器的调度程序等待正在运行的任务发生阻塞，第一次等待时间间隔 t，接着等待(t-δt)，然后等待(t-2δt)，以此类推。当时间间隔减少到0时，请求锁定该处理器的任务,以便解锁当前运行的任务，并阻塞其进一步运行。这时，请求就得到了许可。旋转锁不会使某个运行的任务立即锁定，而是在最终阻塞任务前，先逐步尝试缩短测试时间。

一个信号量可以被捕获多次吗？可以，对于计数信号量来说是可以的(参见 8.2 节中的(v)和(vi))。计数信号量就像是从某个任务发送到另一个任务的事件计数消息。它的意思是被捕获的信号量在被再次捕获之前不需要释放。它是一个无符号的整数或者字节。在生产者-消费者问题[8.2.(vi)节]中，计数信号量便于多进程间的通信。

注意：

RTOS 为信号量的创建和使用提供了 IPC 函数，信号量被用作事件标识、互斥体、资源键(用于进程的资源加锁和解锁)和计数信号量。

8.3.3 消息队列

对于在进程间进行消息通信所用的队列、管道和邮箱，某些 RTOS 并不区分或者只作细微区分，但是其他的 RTOS 认为队列的使用有所不同。

严格地讲，消息队列是一个具有下列特征的 IPC。

(1) 单个进程(任务)或者一组任务(进程)可以使用队列。

(2) 对队列的读(删除)操作采用 FIFO 模式。通常，队列是环形队列，读操作就像缓冲区中

的一个字节环。读队列(从队列中删除)的任务可能具有较低的优先级，以至于执行被延迟。这样的任务按照队列的写操作序列(插入)来读队列和进行相关的动作。

打印机任务是使用消息队列的一个例子。下列是常用的 RTOS 特征。

(1) 在使用调度程序中用于消息队列的函数之前，每个消息队列可能都需要进行初始化。

(2) 可以为多种类或者多目的地的消息提供多个队列。每个队列都可以有一个 ID。

(3) 每个队列要么具有用户可定义的大小，要么具有由调度程序指定的固定的预定义大小。

(4) 当 RTOS 的调用插入队列中时，插入的字节数按照指针所指数据的类型确定。例如，对于一个指向整型或者浮点型变量的指针来说，每次调用将插入 4 个字节。如果该指针指向的是由 8 个整数组成的数组，那么应当在队列中插入 32 个字节。

(5) 当某个队列满的时候，可能需要错误处理和用户代码来阻塞任务(不可能出现自阻塞)。

图 8-5(a)给出了 RTOS 中的三个存储器块。一块用于初始化函数。另外两块用于队列删除和不同的函数。图 8-5(b)给出了一个具有消息和标题的消息队列块。标题用于存放队列的 ID、消息类型(例如，整型、浮点类型、8 个整数组成的数组类型)、队列的最大长度、当前长度以及两个分别指向队列的头和尾存储器位置的指针*Qfront 和*Qback。

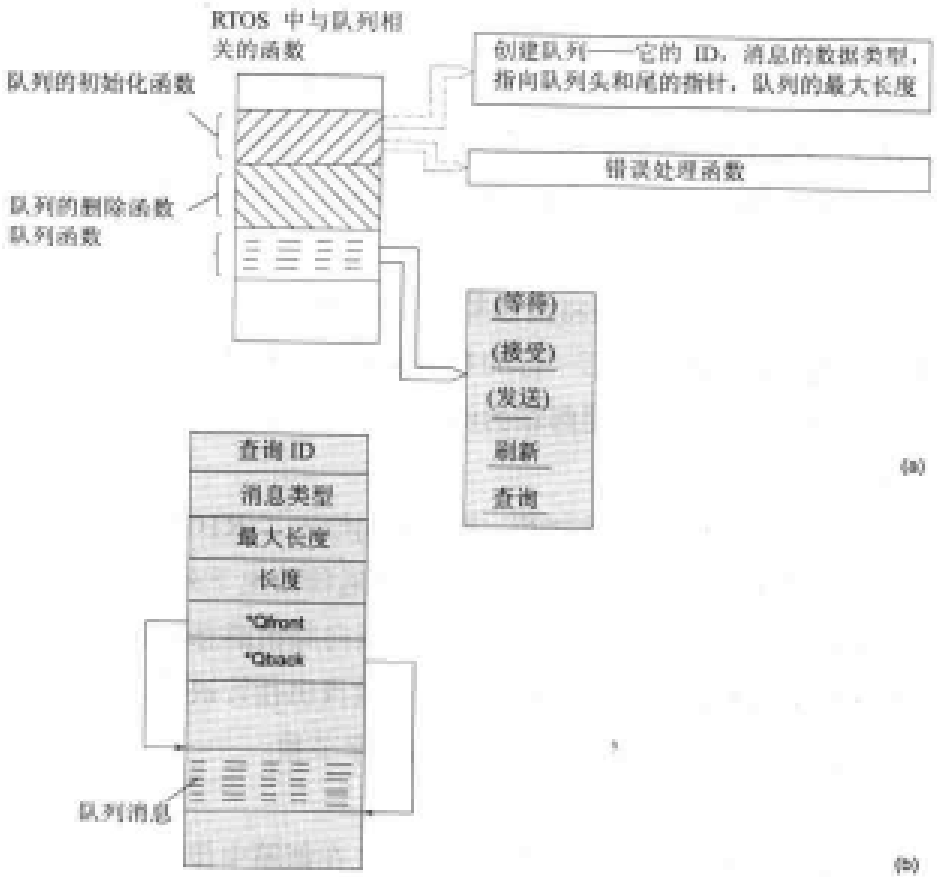


图 8-5　(a)RTOS 的三个存储器块—— 一个用于初始化函数，其他两个用于队列插入、删除和其他函数；(b)队列消息块。

用于队列服务的 RTOS 函数如下：

(1) RTOS_QCreate，用于创建队列并初始化队列消息块和内容，使得指针*Qfront 和*Qback 分别指向队列的头和尾。

(2) RTOS_QWrite(Post)将按照队列末尾的*Qback 指针来向存储器(邮寄)发送消息。

(3) RTOS_QWait(Pend)等待队列中的队列消息，并在接收到该消息时读入。

(4) RTOS_QAccept 检测当前队列的头指针存在与否之后(不等待)，读入该指针。

(5) 由于使用之后不再需要，从前到后读入队列，并删除队列块的 RTOS_QFlush。

(6) 在进行读入且该消息以后不再需要(使用)时，RTOS_QQuery 只查询队列消息块。

(7) RTOS_QQostFront 按照队列的头指针*Qfront 发送(紧急)消息。参见 5.5.4 节。该函数在下列情况中使用。在网络中，如果某个消息没有被接受，它就会返回。然后进行 FIPO 操作(5.5.4 节)。

注意：

RTOS 为消息队列的创建和使用提供了 IPC 函数，队列按照 FIFO 或者 LIFO(对于优先级消息)模式来使用队列。

8.3.4 邮箱

消息邮箱是一个仅用于单目标任务的 IPC 消息块。邮箱消息也可以包含标题，以识别消息类型的说明。源(邮件发送者)是将消息指针发送到已创建(初始化)邮箱的任务(刚开始，邮箱在进行发送之前包含 NULL 指针)。目的地是 RTOS_BoxWait(Pend)函数等待邮箱消息，并且当接收到消息时，读取该消息。

移动电话 LCD 的多行显示任务是将消息邮箱用作 IPC 的一个例子。在邮箱中，当来自时钟进程的时间消息到达时，时间在最后一行的右角显示。当来自另一个任务的消息要显示电话号码时，在中间一行显示。另一个使用邮箱的例子是用于错误处理任务的邮箱，它处理其他任务的不同错误注册。

图 8-6(a)给出了不同 RTOS 的三种邮箱。图 8-6(b)给出了 RTOS 中为邮箱提供的初始化功能和其他功能。在使用邮箱时， RTOS 可以对任务间通信给出下列规定：

(1) 在进行 RTOS 调用时，任务只向邮箱中放入指向邮箱消息块的指针。

(2) 向调度程序所创建的邮箱发送的消息字节的数目由指针决定。当某个 RTOS 调用要向邮箱进行插入时，每次调用所插入的字节是所指向的字节数。

(3) RTOS 可以提供三种邮箱中的一种(图 8-6(a))。

(4) 规定将多个无限的消息装入一个邮箱，并使之在邮箱中排队。

(5) 每个邮箱中可以写入一条信息，且只有当前一条消息被读入之后才能接收下一条消息。规定每个邮箱只能装入一条消息的邮箱，在装入一条消息之后就满箱了。

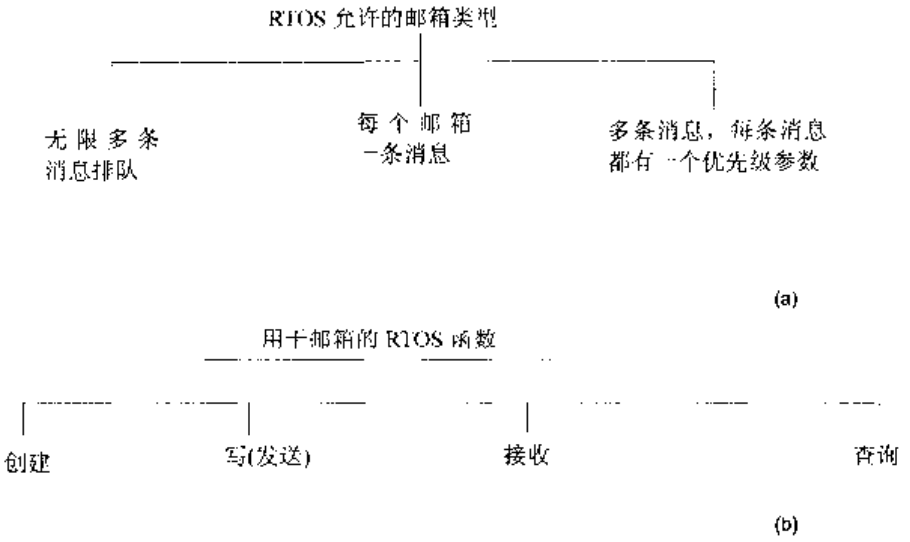


图 8-6 (a)不同 RTOS 中的邮箱类型；(b)RTOS 中邮箱的初始化功能及其他功能

当规定一个邮箱装入多条消息时，队列可以假想为这种邮箱的一个特例。要么假设存在这样一个队列，从该队列进行读(删除)操作按照 FIFO 进行，要么假设存在多条消息，其中每条消息都有一个优先级参数。那么读(删除)操作只能基于优先级进行。即使消息按照不同的优先级顺序接受，读操作也是按照给定的优先级参数进行的。这样的邮箱就像有序的链表一样，链表中的写操作(插入)能够在中间进行，但是读取(删除)操作仅能在头部进行。

邮箱服务中用到的 RTOS 函数如下。

(1) RTOS_BoxCreate 建立一个邮箱并用 NULL 指针初始化该邮箱的内容。

(2) RTOS_BoxWrite(或者 Post)向邮箱发送一条消息。

(3) RTOS_BoxWait(Pend)等待一条邮箱消息，接收到时读取该消息。

(4) RTOS_BoxAccept 函数在检测是否存在消息之后(不等待)读取当前消息指针。读取之后如果不再需要则删除该邮箱。

(5) RTOS_BoxQuery 函数在读取消息且不再需要或者不再用到时，只查询邮箱。

注意：

RTOS 为邮箱的创建和使用提供了 IPC 函数，邮箱的创建和使用就像消息指针或者指向目标任务的消息一样。

8.3.5 管道

RTOS 的管道功能类似于消息-队列功能。惟一的差别在于无限的或者有限的管道尺寸，命名的或者未命名的读取字节流的任务，以及命名的或者未命名的写入字节流的任务。

在严格的意义上来说，消息管道是给定的内部连接的两个任务或者两组任务之间的 IPC 队列。管道的读操作和写操作就像使用 C 命令 fwrite With a file name 进行写操作，以及 fread With a file name 进行读操作一样。管道也类似于 Java 的 PipeInputOutputStreams。它们是 Java 为输入输出流定义的类。

队列中规定可以有多条消息，管道也可以有多条消息。这些消息是两个特定任务(或任务集)之间的消息。管道可以被假想为这样一个队列的特例，从该队列中进行读操作(删除)只能按照 FIFO 进行，并且只有专门命名的任务组能够通过调度程序进行插入和删除。

(1) 任务集中的一个任务能够通过调度程序在尾指针地址*pBACK 处对管道进行写操作。

(2) 任务集中的一个任务能够通过调度程序从管道的头指针地址*pFRONT 处进行读取。

(3) 在管道中，每条消息的字节数目可以是不固定的，且有初始的指针分别指向消息的头和尾，且尾指针地址可以是无限制的。因此，管道可以是无限的，并且在头指针和尾指针之间可以有可变数目的字节。

客户机服务器网络是需要使用管道进行消息传递和任务间通信的一个例子。从某个端口接收输入的客户端任务是与读取字节流的管道相连的。该端口将发送给服务器任务的字节流写入管道中。

下面是 RTOS 常用的特征。

(1) 每条消息的管道都需要在使用调度程序的消息管道函数之前进行初始化。

(2) 可以为消息的多个目的地提供多连接的管道。因此一个服务器上可以连接许多客户。

(3) 当某个 RTOS 调用要向管道中进行插入的时候，字节应该插入到指针指向的地址处。

(4) RTOS[比如 VxWorks]中的管道可以在象文件设备这样的任务虚拟设备中使用。

图 8-7(a)展示了 RTOS 三种类型的函数。一种函数用于初始化，建立一个管道，定义管道 ID、长度、最大长度(在某些 RTOS 中不定义)并初始化两个指针的值，即分别用于管道消息目标(头)的*pFRONT 和用于消息源(尾)的存储器地址的*pBACK。第二种类型的函数是用于管道连接和定义源 ID 及目标 ID 的函数。第三种是用于错误处理的函数。图 8-7(b)给出了消息块中的管道消息，*pFRONT 指向管道头，*pBACK 指向管道尾。

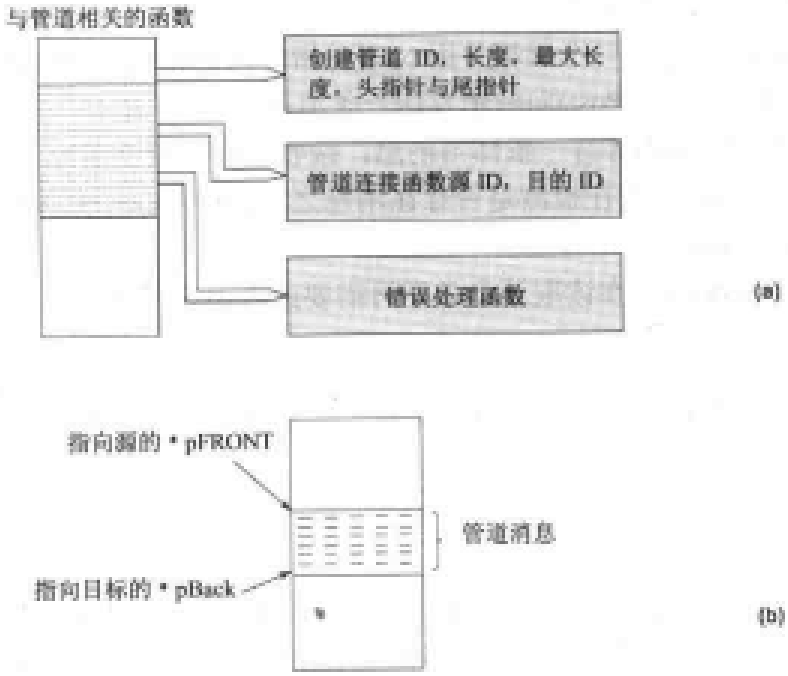


图 8-7 (a)RTOS 的三种函数(初始化函数，管道连接和 ID 函数，错误处理函数)；
(b)在消息块中展示管道消息，*pFRONT 指向管道头，*pBACK 指向管道尾

注意：

RTOS 在源任务和目标任务间建立和使用管道作为消息队列提供了 IPC 函数。IPC 管道用作管道或者文件设备。这意味着管道可以像文件一样打开和关闭。它就像是可以与系统连接或与之断连的设备。

8.3.6 虚拟(逻辑)套接字

两个物理连接器的套接字将两个系统连接起来。例如，计算机的 COM 端口连接着一个调制解调器。类似地，两个任务组之间的两个任务(或者任务的某段)也通过套接字相连(它们都是虚拟(逻辑)的，而不是物理的)。

虚拟(逻辑)套接字在任务间通信中的使用类似于虚拟套接字在浏览器与网站之间的 Internet 连接中的使用。端口 80 是在服务 http 网站的 Web 服务器上约定使用的数字。用于 SMTP 邮件服务的端口是 25。用于 POP3 邮件服务的是 110。每一个服务器或者客户机都有一个 IP 地址。IP 地址和端口数指定了 Internet 上的套接字。

假定使用调度程序，让一个套接字连接源任务组 I 和目标任务组 J 之间的字节流。假设在任务组 I 中存在四个段或者任务，即任务组 I 中的 a、b、c 以及 d。在任务组 J 中有两个段 x 和 y。假设套接字用来从(任务组 I，段 c)发送字节流到(任务组 J，段 x)。这里，源(客户机套接字)端的套接字是由(I，c)的套接字和(J，x)服务器端的套接字指定的。

套接字是在指定的集合(地址)处的两个指定段之间的管道。每一个套接字具有任务的地址(与网络或者 IP 地址相似)和一个段(与进程或者端口相似)号。段(或者是端口，或者是任务)和任务组(地址)可以在同一台计算机上或者同一个网络中。

必须有套接字处的消息进行互连必须遵循的特定协议。(I，c)和(J，x)。段(端口)的规范建立基于协议的链接。任务(地址)数目的规范建立了物理链接。图 8-8 给出了 RTOS 上客户机任务组和服务器任务组之间初始化后的套接字。套接字的应用实例如下。套接字必须在用于互连之前建立。

(1) 套接字的一个应用是嵌入式系统分布式环境中的任务进行互连。比如，网络互连。

(2) TCP/IP 套接字是 Internet 另一个常见的应用。套接字另一个实例应用是任务在移动 Internet 连接上接收按照 TCP/IP 协议传送的字节流。

(3) 还有另一个应用如下。考虑一个具有输入字节流的移动电话，该字节流是通过蜂窝服务接收 SMS 消息或者电子邮件的一个管道输入。

(4) 诸如网络中 Berkeley 套接字这样的应用(它是客户机-服务器网络的一个标准。RTOS 可以提供 POSIX Berkeley 套接字)。

(5) 一个实例应用是任务使用 NFS 协议向计算机或者网络上的某个文件进行写入(网络文件系统协议)。

(6) 另一个套接字的应用是在嵌入式系统的任务或者任务组的一段与一个独立的重量级进程中任务组的另一段之间的连接。调度程序带有套接字连接程序，程序代码指定了源和目标段以及任务组。

注意：

套接字是用于从(任务组 I，段 c)向(任务组 J，段 x)发送字节流的 IPC。套接字是一个客户端服务器，或者说是点到点的 IPC。现在，源套接字(客户端套接字)由(I，c)上的套接字指定，服务器上的套接字由(J，x)指定。套接字有着多种应用。Internet 套接字用于两个端口间的虚拟连接：两个端口均有各自的 IP 地址。RTOS 提供了 IPC 函数用来创建、建立、使用和断连套接字。

8.3.7 远程过程调用(RPC)

在嵌入式系统的分布式环境中存在着远程过程调用。当某个任务在系统 1 中，另一个任务

在系统 2 时，RPC 提供了任务间通信。两个系统都按照点到点的通信模式工作，且不使用客户机服务器模式。点到点的每个系统都能够建立 RPC(客户机进行本地或者远程调用，在客户机-服务器调用中，服务器的响应是本地的或者远程的)。

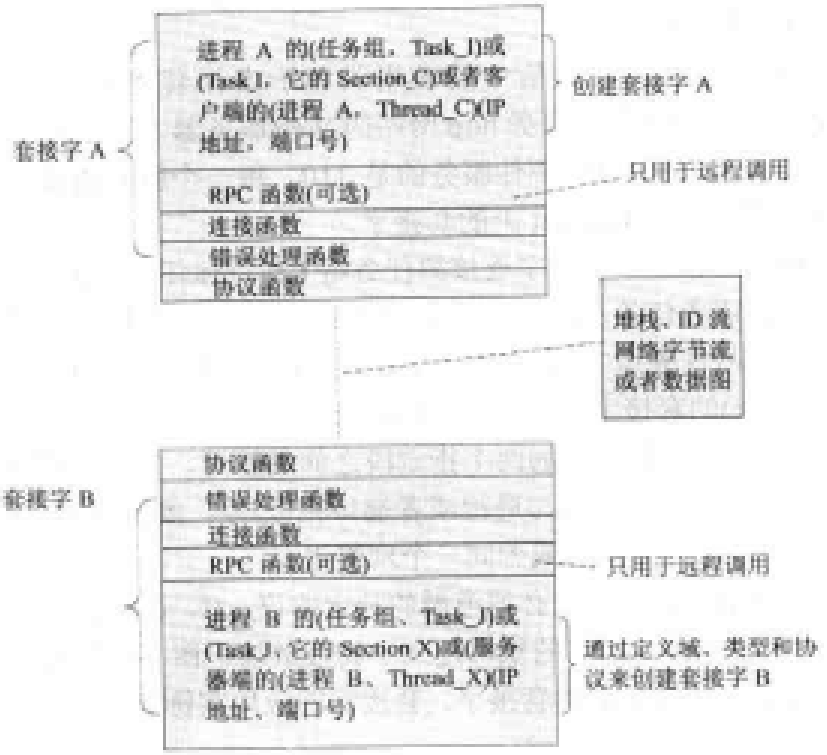


图 8-8 RTOS 中客户端任务组和服务器任务组之间经过初始化的虚拟套接字

本章小结

- 进程是在操作系统调度内核控制下在 CPU 上进行处理的一个运算单元。
- 任务是操作系统调度内核控制下运行在 CPU 上的一个运算单元或者代码组，动作或者函数。
- 每个任务在被 OS 的调度程序调度时都是一个取得 CPU 控制的独立进程。一个任务不能够调用其他的任务。每个任务都通过其 TCB(存储器块)识别，TCB 保存着程序计数器、存储器映射、信号(消息)分派表、信号屏蔽、任务 ID、CPU 状态(寄存器等)以及一个核心堆栈(用于执行系统调用等)的信息。任务总是处于四种状态中的一种：空闲、就绪、阻塞和运行，这些状态是通过调度程序来控制的(如果某个任务没有无限等待循环，将有五种状态——空闲、就绪、阻塞、运行和完成)。
- 进程可以由多个线程组成，线程定义了调度程序调度 CPU 和其他系统资源的最小单元。Unix 和 Java 使用线程。线程是轻量级的进程。
- 单 CPU 系统每次运行一个进程(或进程的一个线程)。对于多任务或者多线程系统来说，调度程序是至关重要的。
- 通常同样的数据会在两个不同的任务(或进程)中使用，如果另一个任务在没有完成对这些数据的操作之前发生中断，就会出现共享数据问题。关闭中断直到第一个任务的操作完成，然后再打开中断，这是一种解决途径。信号量(标识、互斥体或者计数信号量)的使用是另一种解决共享数据问题并执行临界段代码的有效方法。

- 缓冲区是用于存储输出源和输入接收器之间(比如，任务之间、文件之间、计算机和打印机之间、物理设备和网络之间)的消息队列、消息链表，或者字节流的一个存储器块。它必须介于两个界限之间，不能是无限制的或者无限的。比如，打印机缓冲区无法接受来自计算机的无限输出。有界的缓冲区问题是同步的发送源与接收器之间存在的问题。如果消费者不消费，生产者就无法继续生产。消费者也不能一直消费，除非生产者一直生产。使用了有界缓冲区之后，P 信号量和 V 信号量函数解决了传统的生产者-消费者问题。
- 当使用信号量时，某些情况下可能出现优先级倒置问题和死锁情况。操作系统应当注意到这一点并采取相应的措施来避免。
- RTOS 能够处理多个任务间的进程通信。
- RTOS 提供下列 IPC：信号、信号量、队列、邮箱、管道、套接字以及远程进程调用(RPC)。
- 可以为 RTOS 中的单条消息、多条消息或者一个消息指针提供邮箱。
- 管道是连接两个任务的消息队列或者消息流。
- 套接字可以使得网络或者客户机服务器像任务间通信一样进行通信。RPC 用于分布式任务。
- 当前，已经有了 RTOS 和 IPC 函数的标准。

关键词及其定义

- 进程(process)：具有独立程序计数器值和单独堆栈的代码。单 CPU 系统一次只能运行一个进程(或进程的一个线程)。进程是一个概念(抽象)。它定义了一个序列执行(运行)程序及其状态。在进程运行过程中，状态用它的状态(运行、阻塞或者完成)、控制块、调用进程控制块(PCB)或者进程结构，即数据、对象和资源表示。
- 任务(task)：任务用于指定动作的服务，也可以与用于中断执行的代码相应。当任务被 OS 中的调度程序调度时，它是一个独立的掌握 CPU 控制权的进程。每个任务都有一个 TCB。
- 任务控制块(Task Control Block，TCB)：一个用于保存程序计数器、存储器映射、信号(消息)分派表、信号屏蔽、任务 ID、CPU 状态(寄存器等)以及核心堆栈(用于执行系统调用等)信息的存储器块。
- 任务状态(task state)：任务的状态在调度程序的指导下改变。某个实例任务可能处于下列四种状态中的一种，空闲、准备、阻塞和运行，并由调度程序控制。
- 线程(thread)：调度程序调度 CPU 和其他系统资源的最小单元。一个进程可能由多个线程组成。一个线程有独立的进程控制块，与任务控制块类似，且线程也在调度程序的控制下执行代码。
- 同步(synchronisation)：它可以使每段代码、任务和 ISR 运行，并按照某种调度策略依次获得 CPU 访问权，使得在任意时刻的操作都是可预见的。
- 进程间通信(inter process communication)：来自一个任务(或进程)的输出通过调度程序和信号、异常、信号量、队列、邮箱、管道、套接字以及远程进程调用(RPC)传送到另一个任务。

- 共享数据问题(Shared Data Problem)：如果某变量被两个不同的进程(任务)使用，且另一个任务在中断时没有完成对该数据的操作，就会产生共享数据问题。
- 信号量(semaphore)：一个用于观察某个操作以阻止另一个任务或进程运行的特殊变量(或函数)。
- 互斥体(mutex)：一个用于观察某个操作以阻止其他任务或进程进一步运行并同时让另一个任务进一步执行的特殊变量。互斥体有助于在多任务操作调度程序的控制下相互排斥地运行两个任务。
- 计数信号量(counting semaphore)：可以对它进行加 1 和减 1 操作，它不是布尔类型的信号量。
- 缓冲区(buffer)：一个用于输出源和输入接收器(比如任务、文件、计算机和打印机、物理设备和网络)之间的消息队列、消息列表或字节流的存储器块。
- P 和 V 信号量(P and V semaphore)：用 IEEE 标准定义的信号量函数，用于互斥体或计数信号量，或在使用有界缓冲区时用于解决常见的生产者-消费者问题。
- 优先级倒置(priority inversion)：低优先级任务无意间没有释放高优先级任务的进程所造成的问题。操作系统应当采取相应措施避免这一现象。
- 死锁情形(deadlock situation)：等待某个信号量的任务，其信号量的释放来自于某个任务，而另一个任务则等待另一个信号量的释放以运行。这两个中的任何一个都无法继续运行。操作系统应当采取相应措施避免这一现象。
- 消息队列(Message Queue)：任务发送多个 FIFO 或者优先级消息到队列中，另一个任务将给队列消息作为输入。
- 邮箱(mailbox)：来自于某个任务的消息或者消息指针，这些消息或者指针指出另一个任务的地址。
- 管道(pipe)：一个任务用它来发送消息，另一个任务将这些消息作为输入。管道像文件一样也是一个虚拟设备。
- 套接字(socket)：它在客户机-服务器或者点到点环境下，提供了使用任务间协议的逻辑链接。
- 远程过程调用(Remote Procedure Call)：一种用于连接两个远程方法的方法。它先使用某个协议来连接进程。用于分布式任务的情况中。

问题回顾

(1) 如何使用 IPC(进程间通信)将某个进程产生的数据输出传送到另一个进程？

(2) 任务的 TCB 的参数是什么？为何每个任务应当有不同的 TCB？

(3) 任务的状态是什么？谁是控制(调度)任务从一个状态变化到另一个状态的实体？

(4) 定义任务的临界段。

(5) 在进程结束前，另一个优先级更高的进程如何操作和修改该进程临界段的数据(共享变量)？

(6) 计数信号量的使用为何不同于互斥体？如何使用计数信号量？

(7) 给出一个在多进程(多任务)执行过程中出现死锁情况的例子。

(8) 在进程的临界段运行过程中关闭中断的优点和缺点有哪些？

(9) 解释多任务 OS 和多任务调度程序的含义。

(10) 在抢占式调度程序中，每个进程或任务都有一个无尽(无限)循环。如何将资源控制从一个任务传递给另一个任务？

(11) 异常是什么？在发生异常时运行的错误处理任务又是什么？

(12) ISR、任务、线程和进程中的函数各有什么不同？

(13) 列出 P 和 V 信号量的特性，它们如何用作资源键、计数信号量以及互斥体？

(14) 哪些情况导致优先级倒置问题？OS 如何通过优先级继承机制解决该问题？

(15) 管道的含义是什么？管道和队列的区别在哪里？

(16) 旋转锁的含义是什么？指出在何种情况下使用旋转锁机制将有助于锁定控制权向高优先级任务的传递。

(17) 什么是邮箱？在 IPC 中，邮箱是如何传递消息的？

(18) 何时套接字对 IPC 有用？举出四个例子。何时使用 RPC？举出两个例子。

(19) 进程、任务和线程的相似性有哪些？它们的区别又在哪里？

实践练习

(20) 设计一个表来区分使用调度程序何时会出现进程与任务及线程的并发处理。

(21) 将信号用作 IPC 有哪些优点？举出允许使用信号的情况。

(22) 列出五种使用 P 和 V 互斥信号量来解决有界缓冲区问题的情况。

(23) 在 100 秒的时间间隔内，每 10 秒就有 64kB 的字节段以 512kbp 的速度到达。是否需要输入缓冲区？如果需要，需要多少？如果需要，请写出使用 P 和 V 信号量的缓冲区的程序。

(24) 了解 IEEE 承认的标准 POSIX 1003.1b 的详细情况。

(25) 可以使用不同的 IPC 吗？请给出选择，如何根据信号、信号量、队列或邮箱选择 IPC？

第 9 章　实时操作系统

本章前所学内容

下面的内容已经在前面的章节中讲到了：

(1) 硬件单元(处理器、存储器、总线、接口电路、端口、物理设备和虚拟设备、定时器和实时时钟驱动的软定时器、设备网络总线以及它们的接口)(第 1~3 章)。

(2) 关于驱动程序的细节、设备驱动的中断处理例程的编写以及中断处理的细节(第 4 章)。

(3) 面向过程和面向对象的程序设计概念、采用 C/C++进行的设备驱动程序与应用软件的代码编写、数据类型和结构的使用(第 5 章)。

(4) 在单处理器和多处理器系统的软件实现过程中的程序建模概念(第 6 章)。

(5) 嵌入式软件开发过程中的软件工程实践(第 7 章)。

(6) 进程间通信及其在同步并发处理进程、任务和线程中的应用(第 8 章)。

在前面的章节中还阐述了以下的要点，它们使得嵌入式系统中的软件设计颇具挑战性：

(1) 存在大量的中断源，其中包括物理的和虚拟的。只要有中断产生，都由中断服务例程(Interrupt service routine，ISR)负责处理。通过使用 ISR，系统的中断机制保证处理器不仅对前台程序进行处理，而且还对中断进行处理。每个 ISR 在中断源产生中断之后到它开始在处理器上服务(运行)之前，都有一个中断延迟周期。ISR 可能会有时限，在这个时限之内 ISR 必须提供服务，并且结束对中断的处理。因此，系统软件只给少数的 ISR 赋予较高的优先级，并为它们的处理设置优先级顺序。系统必须处理事件控制函数，且函数有响应时间约束(请回顾 4.6 节)。

(2) 对程序中的事件控制函数或者具有响应时间约束的函数进行建模，以及对实时函数的调度进行建模都是通过有限状态机或者 Petri 网实现的。存在进程间通信(发生某个事件、产生中断、某个函数或者一组指令执行结束时进程的输入)，进程间通信之后进程开始在多处理器系统中运行(请回顾 6.2 节)。

(3) 多处理器系统的建模和两组运算之间延时(令牌)的建模(请回顾 6.3 节)。

(4) 两个进程之间的进程间通信(请回顾 6.3 节)。

(5) 实时系统软件开发过程中包含的特殊问题(分析、设计、实现以及测试)(请回顾 7.7 节)。

(6) IPC 的信号量(如标识、互斥体和计数信号量)、解决数据共享问题以及在实时处理过程中，运行临界段代码的概念(请回顾 8.2 节)。

(7) 为了达到最短的周期延迟，而将事件、信号和错误处理函数的异常用于 IPC 当中。

(8) 为了进行调度和同步，对信号量、队列、邮箱、管道、套接字以及远程过程调用(IPC)的使用(请回顾 8.3 节)。

显然，嵌入式软件开发的复杂性非常高；在实时系统软件开发过程中存在着复杂的问题，还存在一些功能差异很大的组件。(i)我们需要一个能够及时提供多种服务，例如资源、存储器、进程、设备、文件、IO 子系统和网络子系统的管理的操作系统。(ii)同样，我们需要 OS 是一个实时操作系统(real-time operating system，RTOS)，能够为调度和同步、响应时间约束、任务优先级、延时和时限以及进程间通信等问题提供解决方法。

当在一个复杂应用中存在对多个设备的处理和服务，并因此要对具有实时约束的多个任务

进行处理和服务时，RTOS 是必不可少的。

除了其他的基本 OS 服务以外，RTOS 两个必需的服务是进程间通信和调度。已经看到，在 RTOS 中，存在为在 RTOS 的多个任务(进程)与调度程序之间进行调度和同步的进程间通信机制设计的详细函数(请回顾 8.3 节)。

本章将学内容

本章的目的是详尽地阐述当存在多个任务和 ISR 时，OS 和 RTOS 的服务，以及任务状态调度。

为了达到这一目的，有必要理解以下几点：

(1) (i)内核服务、操作系统(OS)以及实时操作系统(RTOS)，理解 OS 的目标、结构、进程、存储器和设备管理、文件的组织与管理、IO 子系统和网络操作系统。(ii)按照调度多任务的三种基本策略—— 循环、抢占和时间分片策略进行的调度服务。

(2) 对实时/嵌入式操作系统、实时任务模型和性能测度的需求。

(3) 在 RTOS 环境中，通过中断服务例程实现的中断处理。

(4) RTOS 的(i)任务调度和与优先级有序的任务列表、时间分片以及抢占式任务调度协作的中断延迟。(ii)资源管理。

(5) 在优先级调度案例中临界段的处理。

(6) 固定的实时调度。

(7) 调度程序对周期性的、零散的以及非周期性任务的调度。

(8) 调度算法中的优先级分配。

(9) 使用概率定时 Petri 网(随机的)和多线程图(Multi Thread Graph，MTG)的高级调度算法。

(10) RTOS、IPC 函数以及抢占式调度程序的基本行为列表的标准化，且提供多种行为的相对时序，以允许对使用 RTOS 开发的应用中的处理时序进行优化。

(11) 为进程间同步编写代码时必须注意的一些要点(ISR、函数、任务、调度程序函数)。

(12) OS 安全问题。

(13) 嵌入式 LINUX 的内核。

(14) 移动 OS。

9.1 操作系统服务

系统程序员可以立即使用在给定的 OS 中提供的 OS 函数，而不必再为这些服务(函数)编写代码。

9.1.1 目标

OS 的目标是“完美并正确”，以达到下面几点：

(1) 根据调度和分配简化共享资源(资源指的是处理器、存储器、I/O、设备、虚拟设备、系统定时器、软件定时器、小键盘、显示器、打印机以及其他同类资源。进程(任务或线程)向 OS 请求这些资源。正在处理的任务或者线程都不能使用任何资源，直到某一给定时刻 OS 给它们分配了资源)。

(2) 在给定的系统硬件条件下，通过系统软件简化应用软件的实现。

(3) 提供恰当的上下文切换机制，在一个(或者多个)CPU 上以最佳方式调度进程。

(4) 系统的性能最优化，使不同的进程(或者任务或者线程)在保护下最有效地共享资源，而不会导致安全问题。违反安全要求的例子有：在没有系统调用的情况下，任务获得了对其他任务数据的非法直接访问权、堆栈区溢出到存储器中以及存储器中 PCB 被覆盖(第 8.1.1 节)。

(5) 为进程(任务或者线程)、存储器、设备和 I/O 提供管理功能，以及其他功能。

(6) 对 I/O、设备、文件和类文件设备提供管理和组织功能。

(7) 对网络协议和网络提供易于操作的接口和管理功能。

(8) 提供应用程序在不同硬件配置上的可移植性。

(9) 为不同网络上的应用程序提供互操作性。

(10) 提供公共接口集，以便能够通过标准的开放系统集成多种设备和应用软件。

注意：

OS 的目标是完美性、正确性、可移植性、互操作性，并能够给系统提供公共接口集，在管理进程时，能够提供有序的访问和控制。

9.1.2 结构

系统应该具有表 9-1 中的结构。

表 9-1 系统的分层模型

自顶向下的结构层次	行　为
应用软件	应用软件在使用接口和系统软件的给定系统硬件上执行
应用编程接口(API)	提供应用软件和系统软件之间的接口(输入输出)，使应用软件能够运行在使用指定系统软件的处理器上
OS 提供的系统软件以外的系统软件	OS 可能不具备某些功能，例如，特定网络或特定设备(如多媒体设备)驱动程序。这一层提供除了 OS 服务功能之外的系统软件服务
OS 接口	OS 和上层软件之间的接口(用于输入输出)
OS	内核管态服务(表 9-2)、文件管理和其他用户态的处理服务
硬件 OS 接口	使函数在系统给定的硬件(处理器、存储器、端口和设备)上执行的接口
硬件	处理器、存储器、总线、接口电路、端口、物理设备、定时器、设备网络总线

当使用 OS 时，系统中的处理器在两种状态下运行：

(1) 用户态：允许执行用户进程，但只能使用 OS 的部分功能和指令。

(2) 管态：OS 在保护模式下运行特权函数和指令，并且 OS(更确切地说是内核)只访问一个硬件资源[内核(kernel)的意思是核心(nucleus)]。

注意：

OS 的结构由内核和内核外围的其他服务功能组成。

系统软件包括内核。

9.1.3 内核

OS 是应用软件和系统硬件之间的中间层。OS 包括以下结构单元中的部分或全部。

(1) 内核。

(2) 文件管理(如果它不是给定 OS 中内核的一部分)以及其他所有内核未提供的，但却必需的功能。

内核是所有 OS 中的基本结构单元。它可以定义为 OS 中一个运行于管态的安全单元，而 OS 的其他部分和应用软件都运行于用户态。表 9-2 列出了内核中的功能(服务)，它们都是根据 OS 的设计而定的。

表 9-2 OS 中内核服务

功 能	行 为
进程管理：从创建到删除	允许进程的创建、激活、运行、阻塞、再运行、释放和删除，并在 PCB(Process Control Block，进程控制块)中保持进程的结构。(第 9.1.4 节)
进程管理：处理资源请求	进程通过调用(也就是我们所说的系统调用)或者发送消息来发出处理资源请求。(第 9.1.4 节)
存储器管理：分配和释放	存储器的分配、释放和管理。它还对任务能够访问的区域加以限制。可以进行动态存储器分配。(第 9.1.5 节)
进程管理：调度	进程调度。例如，循环调度方式或优先级调度方式。(9.4 节)
进程管理	通过将数据以消息的方式从一个任务发送到另一个任务来实现进程同步
进程间通信 (任务、ISR、OS 函数间的通信)服务	OS 通过使用 IPC 信号、异常(错误)处理信号、信号量、队列、邮箱、管道和套接字来有效地管理共享存储器访问。(8.3 节)
I/O 管理	字符或块 I/O 管理。例如，确保并口或串口在同一时刻只被一个任务访问。(9.2 节)
中断控制(通过处理 ISR)机制	参见 9.5 节
设备管理	物理设备管理指的是它在同一时刻只被一个任务或者进程访问。而且还提供管道和套接字等虚拟设备的管理，参见 9.1.6 节。设备管理程序的组件有(i)设备驱动程序和设备 ISR(设备中断处理程序)；(ii)设备的资源管理程序
设备驱动程序	简化多种物理设备和虚拟设备的使用，如键盘、显示系统、磁盘、并口、网络接口卡、网络设备以及虚拟设备

注意：

在 OS 控制的进程模型中，进程也表示多任务模型中的任务和多线程模型中的线程(请参考 8.1 节)。

注意：

内核具有对进程、资源、ISR、设备驱动程序、IO 子系统以及网络子系统进行管理的功能。

9.1.4 进程管理

1. 进程的创建

初始的进程是一个在处理器重启时执行存储器指令，然后调用 OS 的进程。处理器开始执行随后创建的所有进程。创建指的是为创建的进程定义地址空间(存储器块)，并为进程定义资源。进程可以以继承的方式创建。

进程管理程序在创建进程时分配 PCB(或者如果任务代表进程时，分配 TCB)，并对它进行管理。其他的 OS 单元可以在必要的时候查询进程的 PCB。回顾一下 8.1 节讲到的 PCB 和 TCB。在这里将再次解释它的结构。它是进程管理程序使用的进程描述符。PCB 或者 TCB 描述了以下内容：

(i) 上下文(在执行最后一条指令和处理器切换到其他进程瞬间的处理器状态字、程序计数器以及其他 CPU 寄存器的状态)。

(ii) 进程堆栈指针。

(iii) 当前状态(是被创建、激活还是创建子进程？在运行？发生阻塞？)。

(iv) 分配的地址和当前正在使用的地址。

(v) 存在进程层次结构的情况下，父进程的指针。

(vi) 指向子进程(层次结构中较低级的进程)列表的指针。

(vii) 指向只能使用(消费)一次的资源列表的指针。例如，输入数据、存储器缓冲、管道、邮箱信息或者信号量(可能存在这些资源的生产者和消费者)。

(viii) 指向可多次使用的资源类型列表的指针(资源类型的一个例子是存储块，另一个例子是 IO 端口)。每种资源类型都有对这些类型资源的计数，例如，存储块的数目或者 IO 端口的数目。

(ix) 指向消息队列的指针。消息队列被认为是只能使用一次的资源的特例。这是因为 OS 产生的消息也要排队等待进程的处理。

(x) 指向访问权限描述信息块的指针。访问权限描述信息块用来描述是全局地共享一组资源，还是与另一进程共享一组资源。

(xi) 进程管理程序用来识别身份的 ID。

2. 已创建进程的管理

进程管理程序能够进行进程的创建、激活、运行、阻塞、再运行、释放以及删除。进程管理程序便于执行下列操作：在多进程(或者任务、线程)系统中执行每一个进程，使得进程状态可以切换。进程的顺序经过以下这些状态："创建"、"就绪或者激活"、"产生"(创建且激活)、"运行"、"阻塞"、"再运行"、"完成"以及"完成"之后的"就绪"(当进程中存在无限循环时)。最后，释放或者删除(在长进程中，阻塞和再运行可以发生很多次)。

回顾一下我们在 8.1 节中学过的进程、任务和线程的定义。进程(任务或者线程)可以看作是只有在 OS 的控制下，才能顺序运行的单元，且每个进程都具有独立的控制块(处理器在某一时刻的描述符)(回顾 8.1 节讲过的 PCB 和 TCB)。负责控制进程执行的实体是 OS 的"进程管理程序"单元。

进程管理程序执行进程对资源或者 OS 服务的请求，然后，同意这一请求，让进程共享资源。正在运行的进程可以通过表 9-3 中解释的两种方法发出请求。

表 9-3　运行中进程对资源或 OS 服务的请求

请 求 方 法	解　　释
消息	运行于用户态的进程产生并发出一个消息，使 OS 让被请求的资源(例如，来自某个设备或者队列的输入)使用或者运行一个 OS 服务函数(例如，重新定义系统时钟速率或者定义进程需要再次执行的延迟时间)
系统调用	OS 中的函数调用。首先发送一条指令，使处理器发生自陷，并且切换到管态。这时 OS 可以像执行库函数一样执行函数(5.1 节)。一旦完成了调用函数的指令，处理器就从管态切换到用户态，并让正在调用的进程继续运行

进程管理程序实现以下功能(i)使进程能够顺序执行或者在需要资源时发生阻塞，并使其在资源可用时继续运行。(ii)为进行资源管理(包括 CPU 上的进程调度)实现了与资源管理程序的逻辑链接。(iii)限制某些资源只在某些进程间共享。(iv)按照系统的资源分配机制分配资源。(v)管理系统中的进程和资源。

注意：

进程管理程序创建进程，为每一个进程分配一个 PCB，管理资源的访问，促使进程状态的切换。PCB 为处于某一状态的进程定义了进程结构。

9.1.5　存储器管理

1. 存储器分配

当进程创建的时候，存储器管理程序通过映射进程地址空间来为进程分配存储器地址(存储器块)(参考 8.1 节)。

2. 初始分配后的存储管理

OS 的存储器管理程序必须是安全的、完善的、受到良好保护的。除了存储器泄漏和堆栈上溢以外，一定不能有任何的错误发生。存储器泄漏指的是试图向没有分配进程或数据结构的存储块中写入数据。堆栈上溢指的是在没有提供额外堆栈空间的情况下，超出分配给堆栈的存储块。

存储器管理程序管理(i)进程对存储器地址空间的使用。(ii)共享存储空间的特殊机制。(iii)用于限制给定存储空间共享的特殊机制。(iv)通过使用存储器(缓存、基本的或者扩展的二级磁存储体和光存储体)层次结构来优化访存周期。记住访存周期按以下顺序逐渐增大：缓存、基本和扩展的磁存储体、光存储体。表 9-4 给出了系统的存储器管理策略。

表 9-4 系统的存储体管理策略

管理策略	解释
固定块分配	存储器地址空间被划分为块。小地址空间的进程获得的块较少，大地址空间的进程获得的较多
动态块分配	存储器地址空间被划分为块。小地址空间的进程获得的块较少，大地址空间的进程获得的较多。之后，存储器管理程序根据进程所处的运算阶段动态地从未使用的存储块描述表中分配大小可变的块(64 或者 256 字节)
动态页分配	存储器中将大小固定的块称为页。存储器管理程序用页描述信息表来动态分配页
动态数据存储器分配	存储器管理程序给链表的节点、队列以及堆栈等不同的数据结构动态分配存储地址
动态地址浮动(relocation)	将相对地址与浮动寄存器相加之后，管理程序将初始约束为相对地址的地址空间进行动态分配。此后，存储器管理程序只需要动态改变浮动寄存器的内容。它考虑使用一个有限定义的寄存器，使得浮动地址位于可用地址的界限之内。这也称为运行时的动态地址绑定
多处理器的存储器分配	参考 6.3 节中的图 6-7。管理程序采用了这样一种分配策略，使存储器可以在两个或多个处理器间紧密耦合地共享，也可以松散耦合地共享，或者进行多段分配

注意：

存储器管理程序为进程分配存储空间，并使用适当的保护机制来进行管理。存储器分配可以是静态的，也可以是动态的。这样可以优化存储器的需求，提高存储器利用率。

9.1.6 设备管理

回顾 4.2 节。系统中存在大量设备驱动程序的 ISR，每一个设备或者设备功能都有一个单独的与其硬件对应的驱动程序。设备管理程序是管理这些驱动程序的软件。回顾 1.4.5 节，OS 的设备管理程序提供并执行用于管理设备和它们的驱动程序 ISR 的模块。

(1) 它不仅管理物理设备，而且还管理虚拟设备，如使用通用策略来管理管道和套接字。

(2) 设备管理对于三种类型的设备驱动程序有三种标准的方法：(i)通过轮询来自每个设备的服务需求实现程控 I/O。(ii)来自设备驱动程序 ISR 的中断。(iii)设备用来访问存储器的 DMA 操作。最通用的方法是使用设备驱动程序的 ISR。

(3) 设备管理程序具有表 9-5 中给出的功能。

表 9-5 设备管理程序的功能

功能	行为
设备检测和添加	提供代码检测各个设备的当前状态，然后对其初始化或者测试，并配置它们，使之可以被 OS 函数使用
设备删除	提供禁用设备资源的代码
设备分配和注册	在不同的地址处给各个设备分配并注册端口(可以是寄存器或存储器)地址，也提供检测它们之间冲突的代码

(续表)

功　能	行　为
删除和注销	对地址明显不同的设备除去和注销端口(可以是寄存器或存储器)地址，并且如果对仍然连接着(注册)的设备再定位地址，还要包含检测现有地址间冲突的代码
限制单个设备	限制某一时刻，设备只能有一个进程(任务)访问
设备共享	允许一个设备被一组进程共享，但是在某一时刻只能被一个进程(任务)访问
设备缓冲区管理	设备硬件可能只有一个单字节缓冲区、双缓冲区或者 8 字节缓冲区。设备缓冲区管理程序使用一个存储器管理程序来缓冲来自发送数据和消息的设备的 I/O 数据流，以使得运算速率比从该设备接收数据的速率快。(运算的一个例子是对输入数据进行译码，如果译码速率低于接收速率，缓冲区很快就会填满)，也可以使用多缓冲区和生产-消费者类型的有限缓冲区。(8.2.(vi)节和示例 8-2)
设备队列，环形队列或者队列块的管理	来自设备的设备 IO 数据流可以被组织为队列、环形队列或队列块的形式 (参见 5.5 节)
设备驱动程序	Unix 设备驱动程序的组件有(i)设备 ISR；(ii)设备初始化代码(用于配置设备控制寄存器的代码)；(iii)系统初始化代码，它仅在系统重启后(一步一步地)执行。当设备驱动程序是 OS 的一部分时，设备管理程序在采用适当的策略后可以有效地操作它们，并优化性能。管理程序在应用程序进程、驱动程序和设备控制程序之间进行协调。进程通过中断向驱动程序发送请求，驱动程序通过执行 ISR 提供行为。设备管理程序轮询设备请求，并按照它们的优先级顺序产生动作。设备管理程序管理 IO 中断(请求)队列。它创建适当的内核接口和 API，并激活设备控制寄存器的特定行为(通过 API 和内核接口激活设备控制程序)
设备访问管理	(i)顺序访问；(ii)随机访问；(iii)半随机访问；(iv)串行通信，这可以通过 UART 或者 USB 实现(参见第 3 章)设备管理程序提供必要的接口

表 9-6 给出了设备的 OS 命令函数集。

表 9-6　用于设备管理的 OS 命令函数集

命　令	行　为
create 与 open	create 用来创建设备，open 用来创建(如果之前没有创建)、配置和初始化设备
write	写入设备缓冲区或发送设备的输出数据
read	读入设备缓冲区或从设备中读取输入数据
ioctl	以特定的功能和特定的参数配置特定的设备。例如，status = ioctl(fd, FIOBAUDRATE, 19200)。fd 是设备描述符(设备打开时返回的一个整数)；FIOBAUDRATE 是使用 19200 作为参数的函数。这条语句设置设备以 19200 波特率进行工作
close 和 delete	close 用于从系统中注销设备，delete 用于关闭(如果之前没有关闭)和删除设备

1. 存在两种类型的设备：字符设备和块设备(参考表 4.1 中的定义)。

2. 对于硬件设备，设备 ISR 也可以称为系统 ISR 或者系统中断处理程序。

3. ioctl 是一个带有 3 个参数的功能强大的函数。下面列出了它的使用范例。(i)访问指定分区的信息。(ii)定义命令和设备寄存器的控制函数。(iii)IO 通道控制。(iv)为设备控制选择命令数字(例如，1 表示读，2 表示写)。(v)通过给定波特率或者其他参数来控制网络设备。如何使用呢？应该按照函数定义，使用第 2 个参数，受控设备将根据第 1 个参数选择。定义函数所需的数值是第 3 个参数。

设备驱动程序ISR调用了多个OS函数。例如：调用intlock( )关闭设备中断系统，调用intUnlock( )打开设备中断。调用 intConnect( )将某个 C 函数和一个中断向量连接(让设备 ISR 的中断向量地址指向指定的 C 函数)。函数 intContex( )判断当某个 ISR 在执行时，是否有中断被调用。

UNIX OS 使设备和文件以尽可能相似的方式实现。设备有 open( )、close( )、read( )、write( )函数，这些函数类似于文件的 open、close、read、write 函数(9.1.7 节)。BSD(Berkley Sockets for Device，设备的 Berkley 套接字)UNIX 中的 API 和核心接口有 open、close、read 以及 write。核内命令如下：(i)select，它用于第一次检查读操作或者写操作能否成功。(ii)ioctl，用于传送特定的驱动信息给设备驱动程序。(例如表 9-6 中的波特率)。(iii)stop，停止设备的输出。(iv)strategy，允许块读写或字符读写。

注意：

设备管理程序初始化、控制和驱动系统中的物理设备和虚拟设备。设备可分为两类：字符设备和块设备。设备驱动函数与文件函数 open、read、lseek、write 以及 close 类似。

9.1.7 文件系统的组织和实现

文件是磁盘、光盘或者系统存储器上的一个命名实体。文件包含数据、字符和文本，也可以是它们的组合。每一个 OS 对文件可能有不同的抽象：(i)文件可以是磁盘上的一个命名实体，它是一个结构化的记录，可以在系统中随机访问。(ii)与在磁盘上的文件类似，文件可以是 RAM 上的一个有着特定结构的记录，也可以称为“RAM 盘”，或者简单地称为“文件”(虚拟设备)。(iii)文件可以是位或者字节的非结构化的记录。(iv)文件设备可以是用于进程间通信的类管道设备。

含有整型数据的文件是否应该和含有字节型数据的文件相区分呢？含有字节型数据的文件是否应该和含有字符型数据的文件相区分呢？由于设备和文件管理方法的不同，开发一个标准的接口集是必需的。只有这样，系统才是可移植的。IEEE 中有一个标准叫做 POSIX 的接口集。POSIX 指的是使用多线程编程的可移植操作系统接口标准(Portable Operating System Interface)。I 之后增加了 X，是因为这一接口和 UNIX 下的接口很相似。它是根据 AT&T UNIX 的 System V 接口的定义而制定的。POSIX 定义了以下函数，如表 9-7 所示：open、close、read、write、lseek 以及 fcntl。lseek 函数用来在字节流中移动指针。fcntl 函数用于文件控制。POSIX 标准下的文件操作与在线性字节序列上的操作类似。

表 9-7 POSIX 文件系统中的命令函数集

POSIX 中的命令	行　为
open	创建文件的函数
write	写文件
read	读文件
lseek(链表查找) 或者设置文件指针	为下一次的读写操作将指针设置于文件中的适当位置
close	关闭文件

1. UNIX 系统中的文件设备是块设备。Linux 也允许像字符设备一样使用块设备。这是因为，Linux 在字符型设备和块设备之间有一个附加的接口。换句话说，在 Linux 中，字符设备和块设备的内核接口是相同的，而在 UNIX 中不是。

2. RAM 上按层次结构组织的文件，通常称做 RAM 盘。RAM 存储器类似于磁盘，对它的访问也类似于磁盘(回顾磁盘的访问；先是根目录，然后是子目录，再然后是文件夹和子文件夹)。

3. Unix 有一个具有非结构化硬件接口的结构化文件系统。Linux 支持不同标准的文件系统。

Windows NT 认为文件是用于顺序保存字节记录的命名实体。OS 分别使用命令函数 CreateFile、ReadFile、WriteFile、SetFilePointer 以及 CloseHandle 来创建文件、读文件、写文件、将文件指针从当前位置设置到一个新位置。

UNIX 中的文件有 open( )、close( )、read( )以及 write( )函数，类似于设备的 open、close、read 和 write 函数。BSD UNIX 接口与 UNIX 接口有细微的差别。

- 块文件系统：它的应用程序产生记录，并保存到存储器中。首先，它们被结构化为一种适当的格式，然后被转换为块流。文件指针(记录)指向从文件开始到文件结尾的一个块。
- 字节流文件系统：它的应用程序产生记录流。这些流保存在存储器中。首先，它们被结构化为一种适当的格式，然后被转换为字节流。在大小为 N 字节的文件中，文件指针(字节索引)指向从索引=0~N-1 之间的一个字节。

就像每个进程有一个处理器描述符(PCB)一样，每一个文件系统都有一个称为文件描述符的数据结构，如表 9-8 所示(对每一个文件管理程序，该结构会有所不同)。对一个文件来说，文件描述符，或者称为 fd，是在打开文件时返回的一个整数。直到文件关闭前，fd 都一直可用。

表 9-8　常见文件系统中文件描述符的数据结构

文件描述符	意　义
identity(文件名)	应用程序中用于识别文件的名字
creator(创建者)或者 owner (所有者)	进程或程序的创建者
state(状态)	状态可以是“关闭”、“存档(保存)”、“打开正在执行的文件”或者“打开用于追加的文件”
lock(锁)与 protection field (保护域)	O_RDWR 文件以可读写方式打开，O_RDONLY 文件以只读方式打开，O_WRONLY 文件以只写方式打开
file info(文件信息)	当前大小、何时创建、何时最后一次修改、何时最后一次被访问
sharing permission(共享许可)	是否可以被共享执行或者读写
count(计数)	引用该文件的目录数
storing media detail(存储媒质细节)	每一次访问可传送的块

注意：

文件管理程序可以创建、打开、读取、查找记录、写入、关闭文件。每一个文件有一个文件描述符。

9.2 I/O 子系统

I/O 端口是 OS 设备管理系统的子系统。与许多设备进行通信的驱动程序将会用到这些端口。I/O 指令依赖于硬件平台。在不同的 OS 中，I/O 系统也有所不同。表 9-9 给出了一个典型 IO 系统的子系统。

表 9-9 OS 中常见 I/O 系统的 I/O 子系统

子系统的层次	行为和子系统之间的层
应用程序	一个应用系程序有一个 I/O 系统，应用程序和 I/O 基本函数——被缓冲的 I/O 或用于读写功能的 I/O 库函数之间可以存在子层
IO 基本功能	这些是独立于设备的 OS 功能。基本的 I/O 功能与 I/O 设备驱动函数——用于读写操作的文件系统函数之间可以存在子层
IO 设备驱动程序功能	这些是设备依赖 OS 功能。I/O 基本功能和 I/O 设备驱动功能之间可能存在一个子层(IO 基本功能是缓冲 I/O 或者文件(块)读写功能)。驱动程序可以和一系列的库函数接口。例如，与用于串行通信的库函数接口
设备硬件、端口或者 IO 接口卡	串行设备或者网络

分别存在同步类型的函数和异步类型的函数。例如，在 POSIX 中，存在如下的异步函数：aio_read( )和 aio_write( )用于 I/O 系统中的异步读操作和写操作；aio_list( )用来初始化某个最大的异步 I/O 端口请求的链表；aio_error( )、aio_cancel( )以及 aio_suspend( )分别用于异步 IO 错误状态恢复，取消 I/O 操作和挂起 I/O 操作。挂起持续到下一个端口设备中断或超时为止；aio_return( )返回完成的操作的状态。

注意：
I/O 子系统是 OS 服务的一个重要组成部分。例如，UART 访问和并口访问。

9.3 网络操作系统

参考关于计算机网络、Internet 和 Web 技术的标准教材，查阅下面所用术语的含义。网络指的是 LAN 中数据帧的交换或者不同主设备(接口)之间包的交换。数据交换可以按照客户机-服务器方式进行，也可以按照点到点的方式进行。TCP/IP、Telnet 和 FTP 是重要的协议，使用这些协议，字节流格式化为帧或者包之后，就可以放置到网络上了。网络主设备间的通信不同于字符设备或者块设备之间的通信。11.2 节中阐述的一个案例研究阐明了两者之间的基本区别以及复杂性。它解释了字节流在网络中从应用程序到网络协议堆栈的转换。对于网络接口(设备)，Linux 或者 Unix 都具有类似于 I/O 子系统的网络子系统。

Webopedia(网络百科全书)中网络操作系统(Network Operating System，NOS)的定义如下：(http://www.webopedia.com/TERM/N/operation_system.htm)

“具有将计算机和设备连接到局域网(local area network，LAN)中的特殊功能的操作系统”。“NOS”的另一个定义是“具有协议堆栈(例如 TCP/IP 或者 Ethernet 802.3)功能和网络设备驱动程序的操作系统”。

NOS 提供了远程登录、文件传输和通过网络设备进行互连的功能。NOS 的内核处理地址解析问题、包收集、身份认证以及向适当的网络设备分派数据的问题。这些功能都是独立处理的，因此对网络上的系统(称作主机)来说，NOS 是必不可少的。

BSD 4.3 就是一个“NOS”。它的内核在 UNIX 的基础上增加了对网络计算的支持。它有套接字接口(参考图 8-8)。在网络上，进程可以通过套接字地址来引用(回顾文件设备。每一个通信系统端口都有独立的套接字地址。套接字地址指向特定的主机，就像 fd 指向一个文件一样)。服务器和路由器都是具有特定用途的 NOS。

Unix、Mac OS、Linux 和 OS/2 都具有网络功能。Linux 网络还提供了防火墙服务和 Web 服务。OS/2 支持 TCP/IP、点到点网络、防火墙以及传统的 LAN 协议。Windows 提供了拨号网络、远程访问和连接共享。NOS 和普通的 OS 有什么不同吗？基本的 OS 有内置的网络功能，但是网络操作系统通常都有通过增加网络特征来增强基本 OS 功能的软件。

下面给出了一些 NOS 的例子：

(1) 3+Share

(2) Arcnet 系统

(3) Artisoft LANtastic

(4) AT&T StarGroup (非 LAN 管理程序)

(5) Banyan VINES

(6) DEC *Pathworks* 和 PCSA

(7) IBM PC LAN

(8) IBM DOS LAN Requester Version 1.30 或者早期版本

(9) Microsoft LAN Manager 及其全兼容网络

(10) Microsoft MS-Net 及其全兼容网络

(11) Net/One PC

(12) Novell NetWare 基本服务器，用于文件和打印机共享以及基于 Web 的网络功能

(13) PC-NFS

(14) TCS 10Net 或 DCA 10Net

注意：

网络操作系统是一个具有将计算机和设备连接到局域网(LAN)中的特殊功能的操作系统，它是一个具有协议堆栈(例如 TCP/IP 或 Ethernet802.3)功能和网络设备驱动程序的操作系统。

9.4 实时操作系统与嵌入式操作系统

9.4.1 实时操作系统

实时操作系统(缩写为 RTOS)软件具有表 9-10 中给出的操作系统结构单元。

表 9-10 RTOS 服务

功 能	活 动
基本 OS 功能	进程管理、资源管理、存储器管理、设备管理、IO 设备子系统、网络设备和子系统管理
RTOS 主要功能	实时任务调度、中断延迟控制(4.6 节)和定时器与系统时钟的使用
时间管理	时间分配和回收以有效地满足时间约束
可预测性	可预测的系统时间行为、可预测的任务同步
优先级管理	优先级分配和优先级继承
IPC 同步	通过 IPC 实现的进程同步
时间分片	时间分片的进程执行
硬实时可操作性和软实时可操作性	硬实时和软实时操作(硬实时指的是每一个任务调度都有严格的时间限制。软实时指的是仅仅为任务操作定义了优先级和任务操作顺序)

5.7.3 节阐述了 RTOS 调度程序对就绪任务列表的使用。9.4.3 小节讲述了三种从这个链表中调度多个任务的方法。

注意:

RTOS 是为响应时间受控和事件受控的进程设计的 OS。RTOS 适用于嵌入式系统，因为嵌入式系统存在实时编程的问题需要解决。

9.4.2 在嵌入式系统中何时需要 RTOS

8.1.2 节介绍了 RTOS 功能的任务，9.4.1 节介绍了 RTOS 功能的职责。嵌入式系统总是需要 RTOS 吗？答案是否定的，并不总是需要。大量小规模嵌入式系统的软件不使用 RTOS，这些功能包含在应用软件中。

回顾 9.4.1 节中 RTOS 的功能列表。在小规模的嵌入式系统中使用以下方法(不需要 RTOS)。

(1) 不必使用 RTOS 存储器的分配和回收函数，C 函数：melloc( )和 free( )分别用于存储器的分配和释放(在 C++函数中使用 delete( )释放存储器)。

(2) 不必使用 RTOS 函数来对存储器的不同地址进行访问，使用 C++类就可以提供数据封装特性，并限制存储器地址的访问。

(3) 不必使用 RTOS 内核来调度任务，可以使用函数队列。编程可以类似进行，代码设计能够最简单。函数队列应用于顺序使用所有可用资源的情况下。回顾前面的示例 5-4 和 5-6(参

考 5.4.6(iv))，示例 5-4 中，主函数执行了一个“函数队列调度”算法。示例 5-6 中有一个“发生中断时的函数排列”。后来，主函数为队列函数调用了这一队列。

(4) 软件也可以直接处理进程(任务)间通信(8.3 节)，而不需要依靠 RTOS。软件代码通过有效管理共享存储器的访问，进行同步和从一个函数向另一个函数发送数据。

(5) 使用用户设计的设备管理程序和设备驱动函数，在没有 RTOS 的情况下，IO 管理也可以进行(第 4 章中的示例)。

(6) 不需要使用 RTOS 来进行文件管理，可以使用标准的“C”函数：fopen( )、fread( )、fwrite( )和 fclose( )完成。

(7) 在小型系统中，键盘、显示器、并口、网络、管道和套接字的驱动程序都很容易编写。

然而，在多任务环境中，RTOS 是必不可少的，因为以下功能为所有的 OS 内核提供了完备、正确、保护和安全的特征。

(1) 存储器分配和回收以及限制堆栈和其他临界存储器块的存储器地址。存储器的动态分配也是可行的。

(2) 在多任务的情况下，任务的有效调度、运行以及阻塞(9.6 节)。

(3) 处理来自中断的硬件源调用的通用且有效的方法(下面的 9.5.1 小节)。

(4) 回顾图 5-8，图中给出了多任务编程模型中的六个任务和四个 ISR，其中的三个任务处于已经初始化了的(就绪)任务列表中。

(5) 任务间(进程间)通信及其同步(8.3 节)。

(6) RTOS 的使用简化了设备、文件、邮箱、管道以及套接字的 IO 管理。

(7) CPU 的多种状态、内部和外部的物理设备或者虚拟设备的有效管理。假设在某个应用程序中同时需要进行以下操作：(i)物理设备定时器、UART 和键盘已经产生中断，服务例程要执行。(ii)文件被认为是虚拟设备，文件打开时，其指针必须指向第一个记录。(iii)物理定时器设置它的控制寄存器。(iv)另一个定时器从系统时钟得到了计数输入。(v)虚拟设备——文件获得了用于写入其中的输入。(vi)发生超时的时候，定时器状态的改变，这将产生对其服务的需求。(vii)在所有所需记录被传送完后，文件的状态发生改变。(viii)定时器在超时后执行一个系统例程。(ix)文件需要执行函数 close( )。RTOS 使用了一种有效且通用的方法来处理这些需求，解决了这些问题。

(8) 回顾 8.2.2 节。文中描述了任务对信号量的使用，以及任务或函数对临界段的使用(临界段指的是不能阻塞的代码或者资源。当与其他例程或者任务共享数据或者资源时会发生临界情况)。RTOS 为这种情况提供了有效的处理方法。

注意：

在小规模的嵌入式系统中，RTOS 并不是必要的。只有当多个进程、ISR 和设备的调度非常重要时，RTOS 才是必需的。RTOS 必须监视响应时间受控的进程和事件受控的进程。

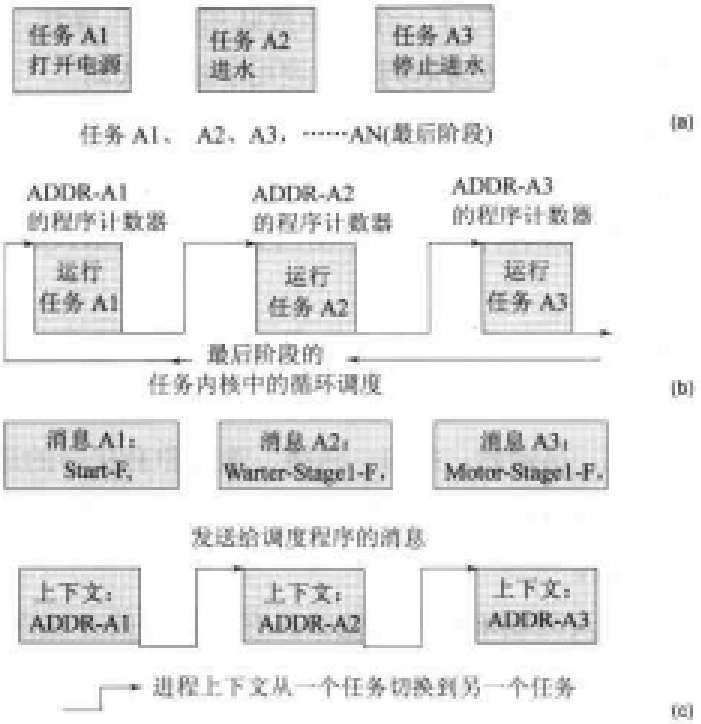


图 9-1 (a)文中示例的嵌入式软件划分的任务组 A1 到 AN 中的前三个任务；(b)循环调度； (c)不同时刻来自调度程序和任务程序上下文的消息

9.4.3 RTOS 的多任务调度管理

RTOS 按照预先制定的调度策略执行多个任务的代码。首先，通过一个简单的例子来考虑由循环 scheduler( )函数实现的循环调度。考虑一个嵌入式系统——自动洗衣机。有许多任务，而且每一个任务都是独立的。任务集 A1 到 AN 中的前三个任务是 A1、A2 以及 A3。图 9-1(a)给出了系统嵌入式软件中多进程的前三个任务。调度程序首先启动任务 A1，并使之进入等待循环，之后调度程序等待任务 A1 发出消息 A1。

(1) 任务 A1：如果洗衣机门关闭，电源按钮已经按下，并且允许启动系统时，该任务重启系统并打开电源。任务 1 等待循环结束，循环在检测到以下两个事件后结束：(i)机门关闭；(ii)用户按下了电源按钮。最后，任务 A1 设置 Start_F 标识，这是消息 A1，它通知调度程序开始执行任务 A2。

(2) 任务 A2：调度程序等待消息 A1 来设置 start_F，如果这一事件发生，任务 A2 就开始执行。一个信号位被设置，通知给水箱充水，并不停地检测水位，当水位足够高时，任务 A2 设置 water_stage1_F 标识，这是消息 A2，它通知调度程序开始执行任务 A3。

(3) 任务 A3：调度程序等待消息 A2。如果这一事件发生，任务 A3 开始执行。一个信号位被设置，停止入水；另一信号位被设置，以启动水箱转动。然后设置 motor_stage1_F 标识，这是消息 A3，它通知调度程序开始执行下一个任务。

图 9-2(b)给出了这个例子中的调度程序。这是一个循环调度程序，称为循环模式调度程序。图 9-2(c)给出了不同时刻任务程序的上下文。任务 A1 的上下文有一个指向 A1 的指针 ADDR_A1。任务 A2 上下文有一个指向 A2 的指针 ADDR_A2。任务 A3 上下文有一个指向 A2 的指针 ADDR_A3。

9.6.1 和 9.6.2 节将详细讲述循环调度。

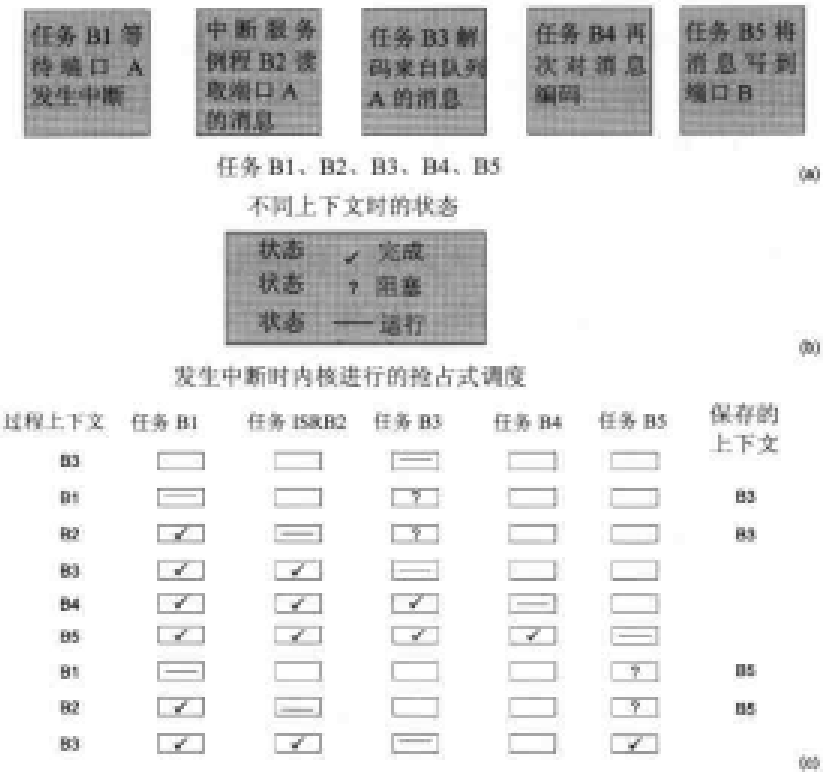


图 9-2　(a)前五个任务 B1~B5；(b)表示抢占式调度状态的符号；(c)前一时刻的任务程序上下文

现在，我们通过另一个简单的例子来考虑抢占式调度程序。假设有一段编码后的消息流到达端口 A。消息流将被解码，并在重新编码后传送到端口 B(回顾示例 4-1)。图 9-2(a)给出了应用程序中的多个进程。5 个进程在 5 个任务中执行，B1、B2、 B3、B4 以及 B5。优先级的顺序如下所示：

(1) 任务 B1：检查端口 A 上的消息

(2) 任务 B2：读端口 A

(3) 任务 B3：解码消息

(4) 任务 B4：编码消息

(5) 任务 B5：向端口传送编码后的消息

图 9-2(b)给出了图 9-2(c)中的内核抢占式调度程序中使用的符号。这是一个基于优先级的抢占式调度程序。高优先级任务从低优先级任务接收控制权。高优先级任务在阻塞低优先级任务后，切换到运行状态。上下文在抢占过程中被保存。图 9-2(c)介绍了如下内容：

(1) 在第 1 个时刻(第 1 行)，上下文是 B3，B3 正在运行。

(2) 在第 2 个时刻(第 2 行)，因为上下文 B3 在端口 A 发生中断时被保存，并且 B1 拥有最高的优先级，所以上下文切换为 B1。现在任务 B1 处于运行状态，而任务 B3 处于阻塞状态。上下文 B3 进入任务 B3 的堆栈中。

(3) 在第 3 个时刻(第 3 行)，在发生中断处的上下文切换到 B2，这一事件仅仅在任务 B1 结束后发生。任务 B1 处于完成状态，任务 B2 处于运行状态，B3 仍处于阻塞状态。上下文 B3 仍在任务 B3 的堆栈中。

(4) 在第 4 个时刻(第 4 行)，上下文 B3 被恢复，上下文切换为 B3。优先级高于任务 B3 的任务 B1 和 B2 都已执行完毕，处于完成状态，因此，任务 B3 从阻塞状态变为运行状态。

(5) 在第 5 个时刻(第 5 行)，上下文切换到 B4，优先级高于任务 B4 的任务 B1、B2 和 B3 都已执行完毕，处于完成状态。现在，B4 处于运行状态。

(6) 在第 6 个时刻(第 6 行)，上下文切换到 B5，优先级高于任务 B4 的任务 B1、B2、B3 以及 B4 都已执行完毕，处于完成状态。现在，B5 处于运行状态。

(7) 在第 7 个时刻(第 7 行)，因为 B5 在端口 A 发生中断时被保存，且任务 B1 具有最高的优先级，所以上下文切换为 B1。现在，任务 B1 处于运行状态，任务 B5 被阻塞。上下文 B5 进入任务 B5 的堆栈中。

(8) 在第 8 个时刻(第 8 行)，上下文在中断处切换为 B2，这一事件仅仅在任务 B1 结束后发生。任务 B1 处于完成状态，任务 B2 处于运行状态，B5 仍处于阻塞状态。上下文 B5 仍在任务 B5 的堆栈中。

(9) 在最后一个时刻(最后一行)，上下文为 B3，任务 B3 在运行中。任务 B1、B2 和 B5 都处于完成状态。没有任何任务上下文。

9.6.4 节详细介绍了抢占式调度。

9.4.4 实时系统中通过 RTOS 进行的多任务调度

RTOS 使系统能够实时调度各种任务。实时系统能够在有限的时间内，且在确定的时刻响应事件。通过一个简单的例子，我们可以理解具有时间约束的任务的调度程序。

假设每 10ms 有一段经过加密的消息流到达嵌入式系统的端口 A，然后，这一段消息被读取并解码，在对每条解码后的消息编码后，重新传送到端口。多进程包含 5 个任务：C1、C2、C3、C4 以及 C5，如下所示：

(1) 任务 C1：每 10ms 检查端口 A 上的消息。

(2) 任务 C2：读 A 端口，将消息放置到消息队列中。

(3) 任务 C3：从消息队列中解码消息。

(4) 任务 C4：从队列中编码消息。

(5) 任务 C5：将经过编码的消息从队列发送到端口 B。

图 9-3(a)给出了 5 个被调度的任务，C1~C5。图 9-3(b)给出了 5 个上下文，它们分别在 5 个调度时间，0~10ms、10~13ms、13~15ms、15~17ms 和 17~20ms 中的 5 个上下文。RTOS 初始化任务 C1~C5。每 1ms 都有一个 RTC 实时时钟中断。只要定时器触发一个事件，RTOS 就调度任务 C1，使其从阻塞状态变为运行状态。如果已知每 10ms 有一个字节到达端口 A，那么让触发器 RTCSWT 每 10ms 触发一个事件。任务 C1 在 10ms 后完成，之后 C2 开始运行。

图 9-3(b)给出了不同时间片的实时调度的状态、保存的上下文和处理器上下文。

(1) 在第 1 个时刻(第 1 行)，上下文是 C1，任务 C1 正在运行。

(2) 在第 2 个时刻(第 2 行)，在第 10ms 后，RTOS 切换上下文到 C2。任务 C1 结束，C2

运行。C1 完成时，没有在堆栈中保存任何内容。

(3) 在第 3 个时刻(第 3 行)，RTOS 在任务 C1 开始后第 13ms 的定时器中断处切换上下文到 C3。C1 已经完成，C2 被阻塞，C3 正在运行，上下文 C2、C3 分别在任务 C2、C3 的堆栈中。

(4) 在第 4 个时刻(第 4 行)，RTOS 在任务 C1 开始后第 15ms 的定时器中断处切换上下文到 C4。C1 已经完成，C2、C3 被阻塞，C4 正在运行，上下文 C2、C3 分别在任务 C2、C3 的堆栈中。

(5) 在第 5 个时刻(第 5 行)，RTOS 在任务 C1 开始后第 17ms 的定时器中断处切换上下文到 C5。C1 已经完成，C2、C3、C4 被阻塞，C5 正在运行，上下文 C2、C3、C4 分别在任务 C2、C3、C4 的堆栈中。

(6) RTOS 在第 20ms 的定时器中断处切换上下文到 C1。由于任务 C5 已经在第 20ms 前完成，只有 C2、C3、C4 在堆栈中。按照调度原则，C1 运行。

9.6.3 节详细介绍了循环调度程序中的时间分片调度(为每个任务分配特定的时间片运行)。

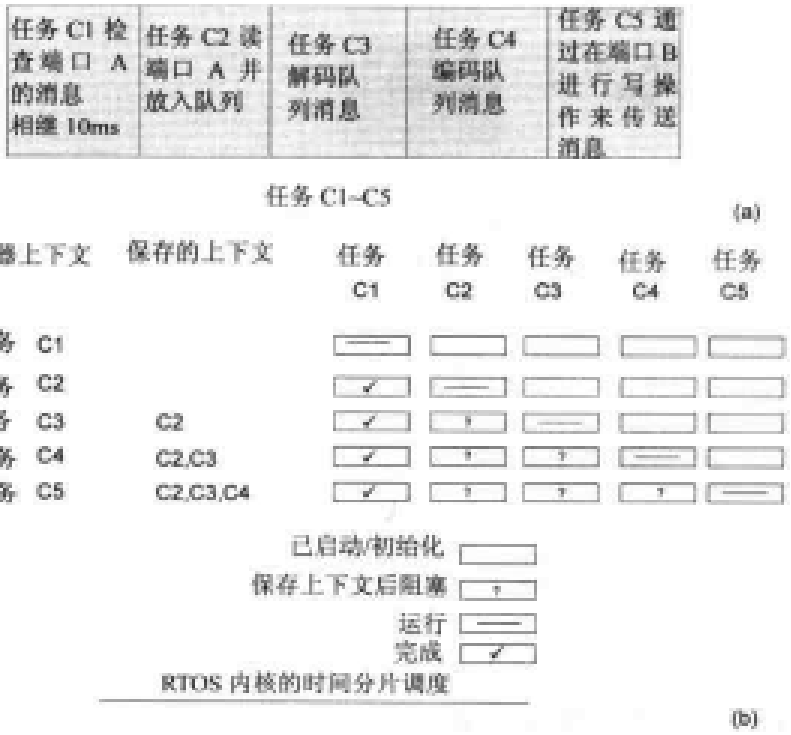


图 9-3　(a)任务 C1–C5；(b) C1–C5 的实时调度程序中 5 个时刻的任务程序上下文

注意：

在处理多任务的时候，RTOS 的调度管理可以采用循环(轮转)调度模式、抢占式调度模式或者时间分片调度模式。

9.5　RTOS 环境中的中断例程：RTOS 的中断源调用处理

RTOS 有 3 个可以选择的系统，以响应中断硬件源的调用(回顾表 8-1 的第 3 行)。图 9-4(a)、

(b)、(c)给出了这 3 个系统。下面的章节将阐述 3 种 RTOS 中的 3 种用于响应中断硬件源调用的系统。

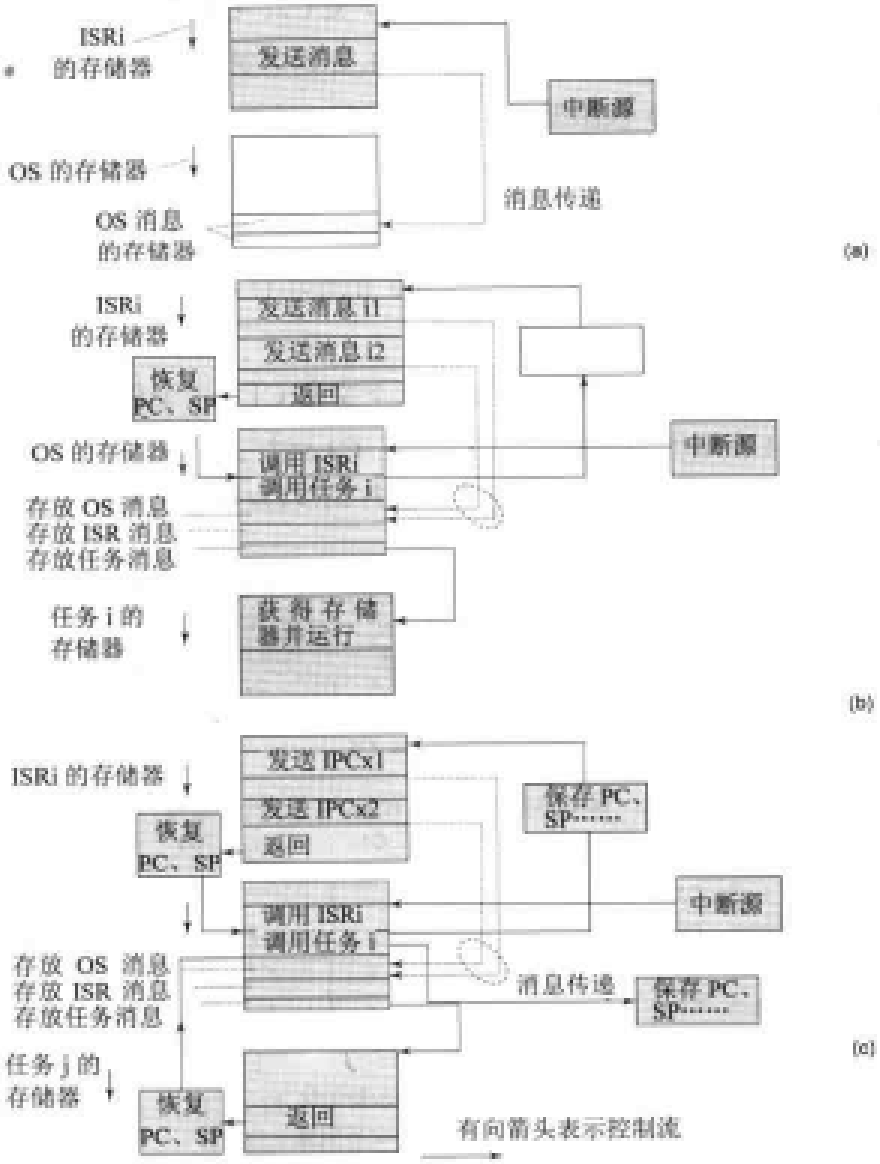


图 9-4 (a)~(c)分别给出了在 3 种 RTOS 中响应中断硬件源调用的 3 个系统

9.5.1 通过中断源直接调用 ISR

硬件源直接调用 ISR，ISR 仅向操作系统发送消息，如图 9-4(a)所示。一旦发生中断，CPU 中运行的进程中止，该中断源所对应的 ISR 就开始执行。RTOS 仅从 ISR 发送信息至邮箱或消息队列中，通知系统哪个 ISR 已经获得 CPU 的控制，ISR 继续执行中断服务指令。存在两个进程：ISR 和 RTOS，它们分别位于两个存储器块中。第 i 个中断源调用第 i 个 ISR(ISR_i)。例程给操作系统发送消息，消息存储在为操作系统消息分配的存储器中。ISR 结束后，操作系统返回被中断的进程或重新调度进程。RTOS 的行为取决于邮箱中的消息。

9.5.2 通过中断源以及调度任务的暂时挂起，直接调用 RTOS

发生中断时，RTOS 侦听硬件源调用，并在保存了处理器状态(或者内容)之后，初始化相应的 ISR。在执行过程中，ISR 向邮箱或者队列发送一个或者多个输出和消息，如图 9-4(b)所示。在三个存储器块中分别有一个例程：第 i 个 ISR、RTOS 以及第 i 个任务。第 i 个中断源导致操作系统调用第 i 个 ISR，它将现场保存在堆栈中后执行 ISR_i。ISR 向 RTOS 发送消息，以初始化第 i 个任务，并在保存现场后返回，这些消息存储在为 RTOS 消息分配的存储器中。此时 RTOS 负责将第 i 个任务初始化到就绪状态，使其之后运行中断服务所需的指令。该 ISR 必须短小，并且只是简单地发送消息，无论任务何时被调度，都要执行余下的代码。RTOS 仅对任务(进程)进行调度，而且仅在任务间切换上下文。ISR 仅在任务暂时挂起时执行。

9.5.3 通过中断源以及 RTOS 对任务和 ISR 的调度，直接调用 RTOS

RTOS 截获并执行从 ISR 返回的任务，无需对任务进行初始化，如图 9-4(c)所示。三个存储器块中分别保存的是：第 i 个 ISR、RTOS 以及第 j 个任务。第 i 个中断源导致 RTOS 调用第 i 个 ISR 并锁住第 j 个任务，RTOS 切换上下文后执行 ISR_i。例程不向 RTOS 发送初始化第 i 个任务的消息，它仅对任务所需参数发送进程间通信(例如设备状态寄存器和输入寄存器的 x1 和 x2 内容)。参数保存在为 RTOS 输入分配的存储器中，此时 RTOS 调用(初始化)返回，恢复上下文并为第 j 个或其他任务所需指令切换上下文。ISR 无需短小，它只是在进程间通信时产生并存储输入参数(设备状态寄存器和输入寄存器)。无论何时，只有任务执行指令。RTOS 对任务(进程)和 ISR 进行调度并且在它们中切换上下文。

注意：

RTOS 采用下面三种方法之一来处理中断源调用：(i)ISR 服务直接通知或者仅通知 RTOS。(ii)内核侦听调用并调用相应的 ISR 和任务。RTOS 内核仅在任务(进程)间调度，ISR 仅在任务被 RTOS 暂时挂起时运行。(iii)当 RTOS 内核切换回到该任务被调用时的上下文时，内核通过上下文切换调用 ISR 来获取参数，然后执行中断源对应的任务。RTOS 不仅对任务(进程)进行调度也对 ISR 进行调度。

9.6 RTOS 任务调度模型，作为性能测度的中断延迟和任务响应时间

调度程序可以采用三种通用的模型策略：

(1) 控制流策略：完全控制输入、输出的顺序。控制步骤是确定的。最坏情况下的中断延迟也进行了定义。调度程序保证在状态(或位置和转换)在编写代码的时候预先确定了。协作调度程序采用了这种策略。调度程序程序可以通过 FSM(有限状态机)建模，也可以用 Petri 网的一个等效子类模拟 FSM 建立，Petri 网中的转换只从一个结点位置到另一个指定的结点位置 (参

考 6.2 节 FSM 和 Petri 网模型)。

(2) 数据流策略：中断的产生是不确定的。同样，任务控制步骤也是不确定的。网络即是一个例子。包的到达不确定。当客户请求服务时，其优先级是未知的。对于这种情况，最坏中断延迟没有定义。输入、输出的顺序是不可行的。抢占式调度程序采用数据流策略。因为从一个或者多个结点位置到另一个或者多个结点位置存在转换，所以调度程序程序可以采用 Petri 网建模。来自转换处的输出函数产生的数据，决定了程序中的下一个序列(参考 6.1.1 节数据流模型)。

(3) 控制数据流策略：任务设计以及调度程序可以提供超时延迟特征。用确定时间的 Petri 网来代替FSM进行建模是可行的。因为最大延时是确定的，所以最坏延时也是确定的[参考6.1.2节的控制数据流模型]。

下面列出了 RTOS 调度程序使用的调度模型。

a. 循环队列中就绪任务的循环协作调度。它和函数队列调度紧密相关。

b. 优先约束的协作调度。

c. 时间分片的循环协作调度。

d. 抢占式调度。

e. 固定时间调度。

f. 周期、零散的和非周期任务的调度。

g. 使用“最早时限优先”(Earliest Deadline First，EDF)的动态实时调度。

h. 使用概率定时 Petri 网(随机)(6.2.2 节)或多线程图的高级调度算法。这些算法适用于多处理器以及复杂的分布式系统。

9.6.1 使用就绪任务循环队列的协作轮转调度

图 8-1 给出了多个函数调用的编程模型。同样，还有一种调度程序编程模型。图 9-5(a)给出了一个调度程序，在该调度程序中，RTOS 在协作循环模型的调度程序中插入顺序执行的就绪任务列表。每当 CPU 开始执行下一个进程时，程序计数器就会改变。图 9-5(b)给出了切换上下文时，程序计数器是如何发生变化的。调度程序切换上下文，使得不同任务能够顺序执行，调度程序在循环队列中逐个从列表中调用这些任务。

协作的意思是每个就绪任务共同协作，使正在运行的任务完成。从就绪到完成期间，每个任务都无堵塞。轮转是指每个就绪任务仅仅从循环队列中顺序执行。服务按照发生中断时任务的初始化顺序进行。在轮转模式下，每个任务的执行优先级都相同。任务优先级参数按照其在队列中的位置进行设置。

对于每个任务来说，最坏延时都是相同的。即 t_{cycle}，它是就绪任务的循环队列的循环周期。队列越长，t_{cycle} 越大。如果某个任务正在运行，所有其他的就绪任务都必须等待。对于第 i 个任务，假设事件进入列表时，事件检测时间是 dt_i，从一个任务到另一个任务的切换时间为 st_i，

任务执行时间是 et_i。那么如果在就绪列表中有 n 个任务，调度的最坏延时是：

$$T_{worst} = \{(dt_i + st_i + et_i)_1 + (dt_i + st_i + et_i)_2 + \cdots + (dt_i + st_i + et_i)_{n-1} + (dt_i + st_i + et_i)_n\} + t_{ISR}$$

其中，t_{ISR} 是所有 ISR 的执行时间之和。记住，T_{worst} 总是小于时限的，对于列表中的每个任务，用 T_d 表示其时限(参考 4.6 节)。

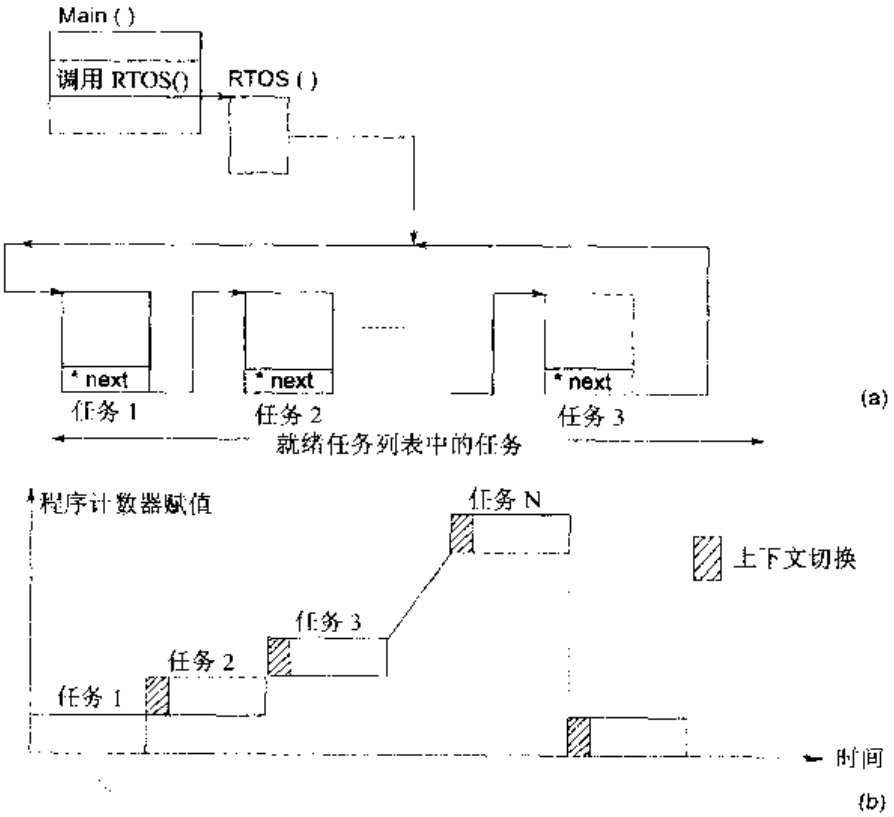


图 9-5 (a) RTOS 在协作循环模型的调度程序中插入顺序执行的就绪任务列表的调度程序；(b)调度程序从循环链表中逐个调用任务的过程中，不同时刻的程序计数器赋值(切换)

9.6.2 使用按照优先级约束排序列表的就绪任务的协作调度

图 9-6(a)给出了在第一层(右上方)上执行的 ISR 的基于优先级的协作调度，和在第二层上(左下方)执行的基于优先级排列的有序列表中的就绪任务。图 9-6(b)给出了调度程序调用 ISR 和在有序列表中的相关任务的过程中，不同时刻的程序计数器切换。任务列表的排序是由 RTOS 使用优先级参数 *taskPriority* 来完成的(参考 5.7.3 节了解关于含有 *m* 个已初始化的任务的有序列表的使用方法)。

RTOS 调度程序首先只运行有序列表中的第一个任务，t_{cycle} 等于列表中第一个任务的执行周期。当第一个任务执行完后，将其从列表中删除，则下一个任务就成为第一个任务。如果有序列表中的任务按照循环顺序执行，就称为基于优先级的循环协作调度程序。构成有序列表的插入和删除操作只在每个周期的开始时进行。

在第一层中，ISR 中有一小段必须立即执行的代码(请参考 9.5.1 节)。ISR 根据其被赋予的优先级运行于第一层(图中右上方)。它为要进行初始化的任务发送一个标识或者令牌和优先级参数。这就是要插入到就绪任务列表中的任务。此时发生协作调度，且每一个就绪任务进行

协作，以便让正在运行中的任务完成。从开始到结束，没有一个任务会发生阻塞。然而在循环调度的情况下，循环只是在来自于按优先级排序的列表中依次运行的就绪任务当中进行的。顺序是根据中断源和任务的优先级确定的。

假设 P_m 为执行时间最长的任务的优先级。那么在最坏的情况下，最高优先级任务和最低优先级任务的延迟分别为：

$$\left\{ (dt_i + st_i + et_i)_{P_m} + t_{OSR} \right\}$$

和

$$\left\{ (dt_i + st_i + et_i)_{P_1} + (dt_i + st_i + et_i)_{P_2} + \cdots + (dt_i + st_i + et_i)_{P_{m-1}} + (dt_i + st_i + et_i)_{P_m} + t_{OSR} \right\}$$

此处，P_1、P_2、…、P_{m-1} 以及 P_m 为有序列表中任务的优先级，且 $P_1 > P_2 > \cdots > P_m$。对于每一个任务和中断源来说，在这样的调度程序下，要满足 T_{worst} 小于 T_{best} 的要求是很容易的，但不保证一定能满足这种要求。程序员给执行时间 T_d 最短的任务分配最高的优先级。

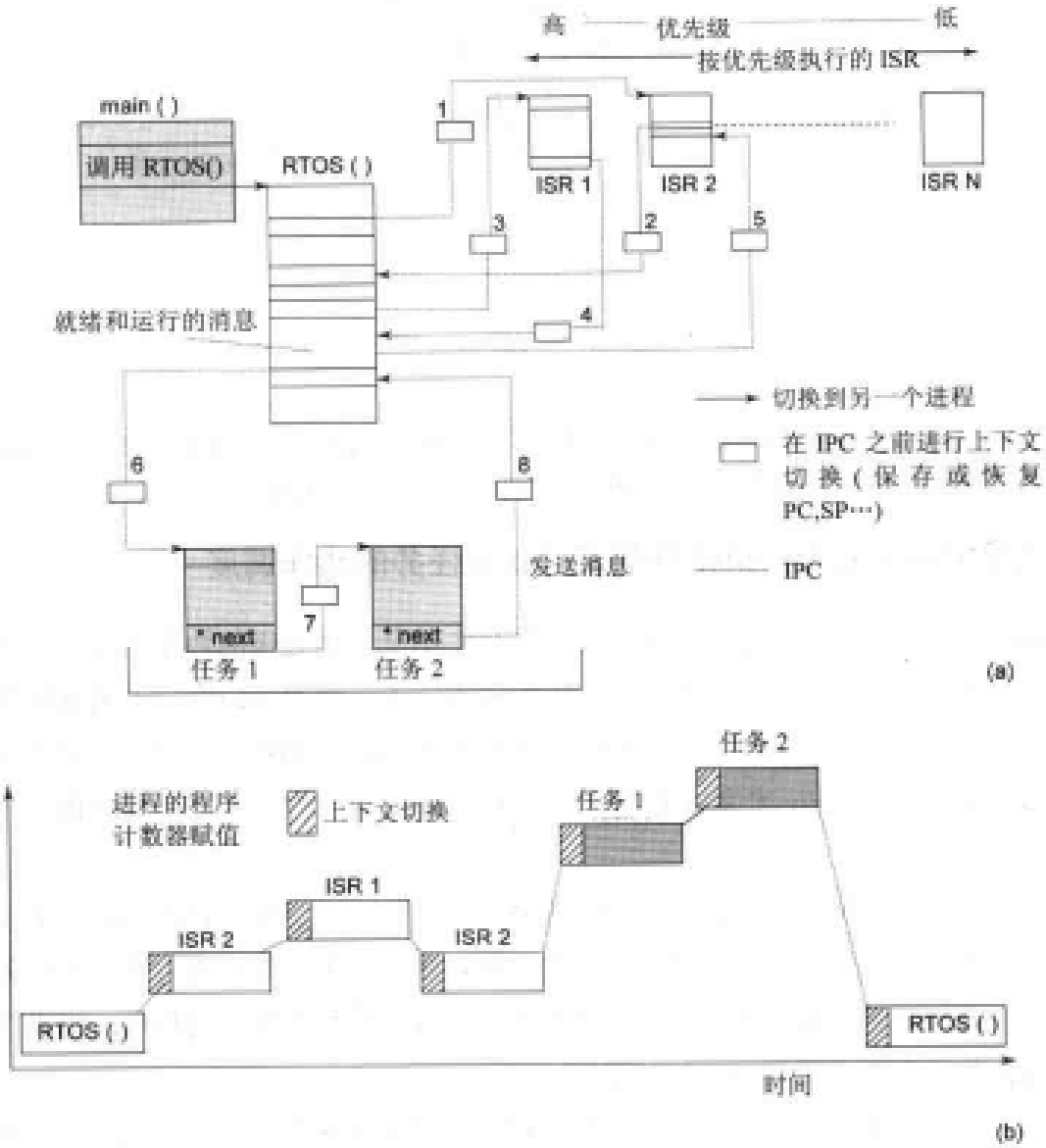


图 9-6 (a)在第一层上执行的 ISR 基于协同优先级调度(右上方)和在第二层上执行的有序列表中基于优先级的有序任务(左下方)；(b)调度程序调用 ISR 和相应任务的过程中，程序计数器在不同时刻的赋值

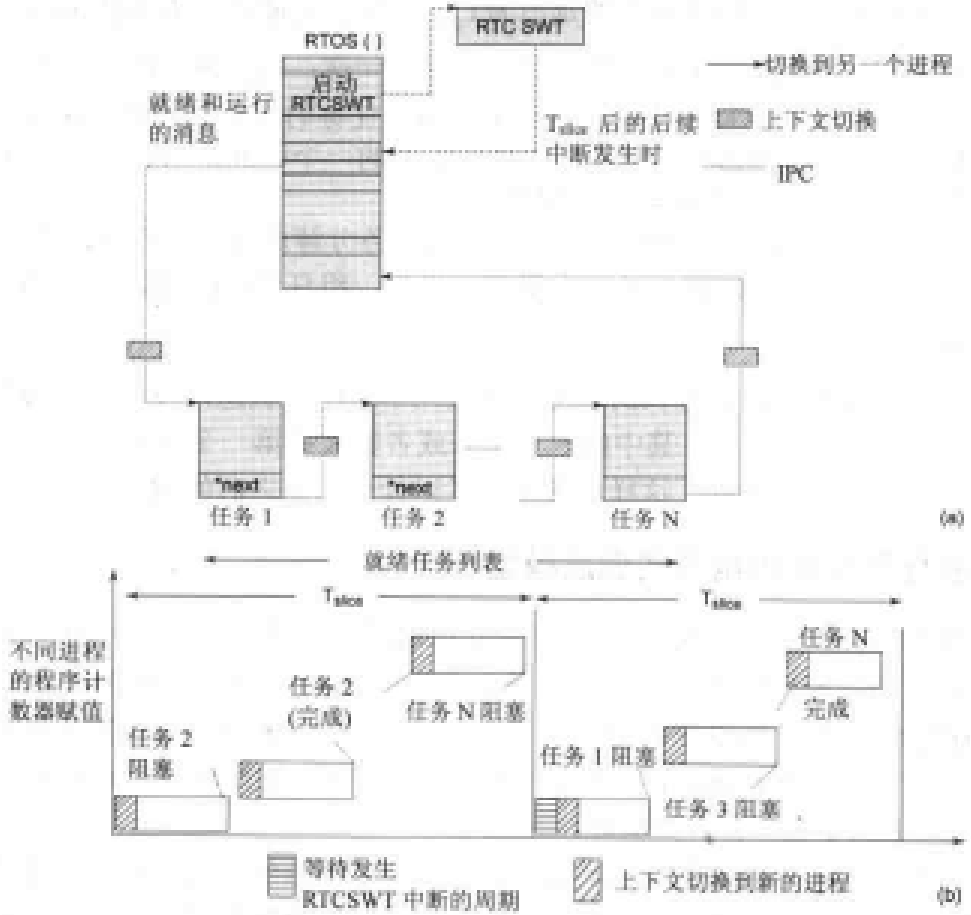


图 9-7　(a)协同时间分片任务调度的编程模型；(b)在两个重要时刻，调度程序调用任务时，程序计数器的赋值。每个周期任务占用时间=t_{slice}，且 $et_2<et_N<et_1$

9.6.3　时间分片的循环调度(速率单调的协作调度)

当第 p 个任务的执行时间 et_p 比较长时，最低优先级任务的最坏情况下的延迟会超过它的时限。为了解决这个问题，最好是 RTOS 为每一个任务的结束定义一个时间片。如果某个任务超过了时限还没有结束，就阻塞它，并且只在后续的周期(在最低优先级任务执行完毕，且新的周期从被分配为最高优先级的任务开始之后)中完成余下的代码。使用时间分片的循环调度非常简单，没有对队列或者列表中进行的插入和删除操作。图 9-7(a)给出了协作时间分片调度的编程模型。图 9-7(b)给出了当调度程序在两个连续时间片调用任务时进行上下文切换的程序计数器。这里假定每个任务占用的时间满足关系 Task2<TaskN<Task1。给每个任务分配一个最大时间间隔为 t_{slice}/N，t_{slice} 是 RTCSWD 定时器(RTOS 中)发生中断并初始化一个新的周期所需要的时间。

在这种模式下，RTOS 在时间片 t_{slice} 的一个周期内完成所有的就绪任务。周期完成速率 $(t_{slice})^{-1}$ 是一个常数(对速率单调的调度程序来说)。如果总共有 N 个任务，假设 T_{worst} =所有任务的最长执行时间之和，那么当 $t_{slice} \geqslant T_{worst}$ 时，T_{worst} =

$$\left\{ (dt_i + st_i + et_i)_1 + (dt_i + st_i + et_i)_2 + \cdots + (dt_i + st_i + et_i)_{N-1} + (dt_i + st_i + et_i)_N \right\} + t_{ISR}$$

如果 t_{slice} 等于每个任务最长执行时间之和，那么每个任务只执行一次，并在一个周期中结束执行。如果一个任务在可能占用的最长时间之内执行完毕，那么在两个周期之间就有一段

延迟。对每个任务来说，最坏情况下的延迟为 t_{slice}。任务可能需要周期性地执行。任务需要重复执行的时间为 t_{slice} 的整数倍。如果每个任务只执行一次，t_{slice} 也应该小于所有任务执行周期的最大分解因子。在时间分片协作调度过程中，每个任务的响应时间都很容易估算。考虑第 k 个任务。任务会在任务周期加上从任务 1 到任务(k-1)的一个时间片所花费的最长时间之内做出响应。列表最后的第 m 个任务的响应时间最长。

另一种模型策略是把一个执行时间特别长的任务分解成两个、四个或者更多任务，这组任务(或奇数个任务)在一个时间片 t'_{slice} 内执行，另外一组任务(或者偶数个任务)在另外一个时间片 t''_{slice} 内执行。

另一种可行的策略是把耗时长的任务分解成几个顺序的状态，或者是 FSM、Petri 网络内的几个结点位置和转换。此时，其中的一个状态或者转换在第一个循环中运行，下一个状态在第二个循环中运行，依此类推。这样，就缩短了运行于状态之后的剩余任务的响应时间。

9.6.4 调度程序控制的抢占式调度模型策略

协作调度程序(在 9.6.1 到 9.6.3 节中已经介绍过)的作用是使每个就绪任务相互协作，从而使正在运行的任务完成。假设存在 N 个任务 task 1 到 task N，并且为中断服务赋予的优先级顺序为 1(最高)到 N(最低)。假设在一个时间片协作调度中，中断发生在周期刚刚开始的时候，这就意味着，从开始到结束，task 1 一直没有机会运行，即直到循环到达 task N 结束或者定义的周期 t_{slice} 结束，task 1 也不会得到服务。

协作时间片调度程序在设计上比较简单，当需要顺序地使用嵌入系统资源或者没有一个任务的时限比 t_{slice} 或者 t_{cycle} 小时特别有用。然而，这个协作调度程序的一个弊端在于，由于低优先级任务的长时间执行，使得高优先级任务至少要等到它执行结束后才能运行。此外，如果有一个循环协作调度程序没有预先定义 t_{slice}，那么还存在一个弊端，因为它会一直延迟到所有其他列表或队列中的任务都结束(参考 9.6.1 和 9.6.2 节的内容)。一个高优先级的任务能不能通过中断低优先级任务，从而实现抢占运行？可以回顾 4.6.4 节的内容。硬件轮询，以决定在指令的末尾处比当前优先级高的 ISR 是否需要服务，如果需要，那么高优先级的 ISR 运行。同样，通过向当前任务发送一个消息，RTOS 抢占调度程序可以在一条指令的末尾中断正在运行的任务，使得优先级更高的 ISR 控制 CPU。

假设任务的优先级满足：task_1>task_2>task_3>task_4>…>task_N。图 9-8(a)给出了 N 个任务的抢占式调度，同时图中它也介绍了进程从一个任务转换到 RTOS，以及从 RTOS 转换到一个任务时的上下文转换情况。当优先级顺序满足 task_1>task_2>task_3 时，图 9-8(b)给出了调度程序抢占调用 task_2 时程序计数器的赋值情况。

每个任务都有一个从开始(在空闲状态之后)到结束的无限循环(参考图下方的三个小方框：task_1、task_2、task_N)。task_1 的最后一条指令指向下一个地址，*next。在无限循环的情况下，*next 指向 task_1 开始的位置(参考 5.4.5 节的内容)。由于此处指向返回到 RTOS 的地址，从而使 RTOS 初始化，然后执行就绪任务列表中的下一个任务，因此它看起来不像是一个协作式调度程序。在抢占式调度程序中，task_2 的运行过程中存在一个抢占执行任务的 RTOS 消息。注意图 9-8(a)中标记的(1)、(2)、(3)。它们的含义将在下文中解释。

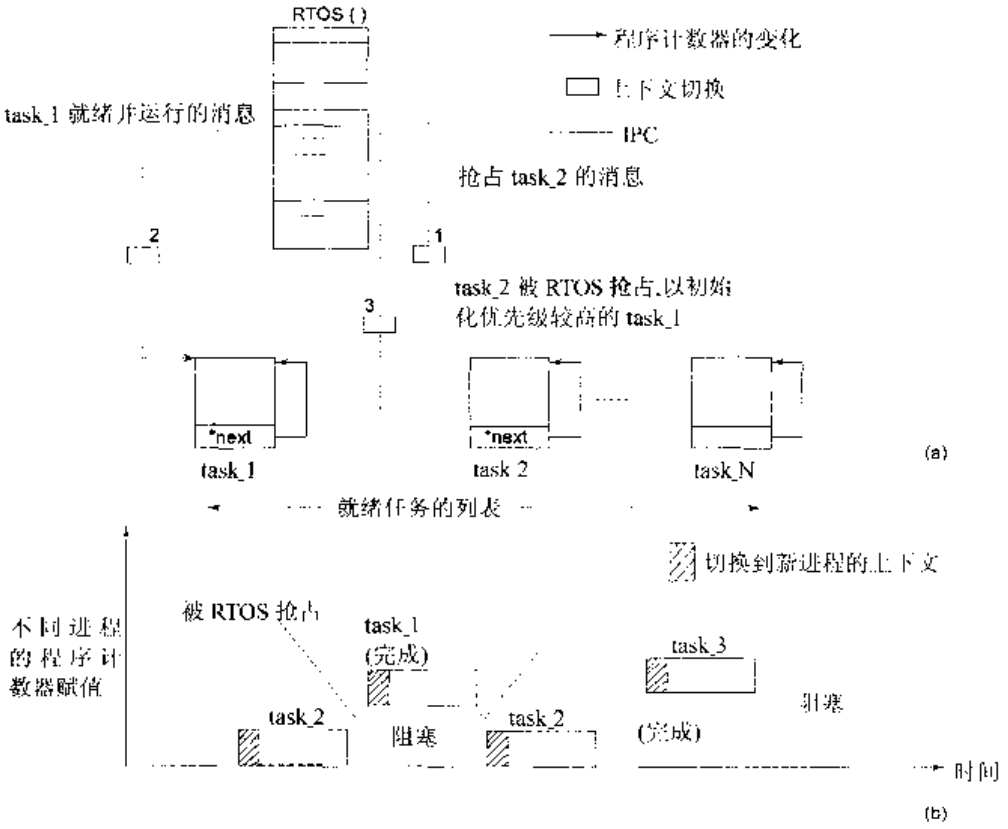


图 9-8 (a)任务的抢占式调度(运行中的任务被抢占并阻塞，以保证另一个优先级更高的任务执行)注意序列编号 2 和 3。其含义可以参考正文；(b)在调度程序调用抢占 task_2 时程序计数器的赋值。任务的优先级关系为 task_1>task_2>task_3

(1) 较高优先级的 task_1 初始化如下。task_2 发生阻塞，并且向 RTOS 发送一个消息。

(2) 现在 RTOS 向 task_1 发送一个消息，使其进入就绪状态然后运行。

(3) 当 task_1 结束时，向 RTOS 发送一个消息，task_1 的上下文恢复到其开始运行时的状态。然后，RTOS 使 task_2 进入就绪状态，task_2 开始运行。当 task_2 结束时，将会向 RTOS 发送一个消息，task_2 的上下文恢复到其开始运行时的状态。继而，RTOS 向 task_3 发送一个消息，使得 task_3 进入就绪状态，task_3 开始运行。

在无限循环中，每一个任务都设计得像是处在任务就绪位置和任务运行位置之间的一个独立的程序(关于无限循环的概念，请参考示例 5-2 和 5-3)。与函数或者协作调度程序中任务不同之处在于，在这种模式下，任务并不返回到调度程序。在循环内部，操作和转换是由事件、标识或者令牌决定的。任务设计使得任务可以移植到另外一个类似模块的抢占式调度程序中。

在设计任务代码时，使用超时间隔的一个好处是可以估计最坏情况下的延迟。在最坏情况下，任何一个延迟都等于 t_{ISR} 与所有其他更高优先级任务的时间间隔之和。另一个好处是，RTOS 可以报告错误并进行处理。在需要的情况下，超时提供了一种方法，使得在必要时，RTOS 甚至可以运行优先级最低的那个任务。请参考图 9-9 种的模型，这是一个抢占式调度的 Petri 网络。

当抢占事件发生时，必须进行任务切换，此时调度程序搜索优先级最高的任务，只有这个任务能被调度程序转换到运行位置。当 taskSwitchFlag 标识被发送到优先级最高的那个任务，而不是先前的任务时，就会发生转换(参考 9.5.3 节))。那么应该怎样缩短上下文转换的时间间隔呢？由于在函数调用中，静态变量驻留在 RAM 中，而不是保存在堆栈中，所以可以通过变量的静态声明缩短上下文转换的时间间隔(参考 5.4.3 节中的静态声明)。在这种情况下，发生调用时，程序计数器和很少几个必须保存的寄存器被保存，因此任务转换不会导致额外的堆栈保存开销。

引起一个事件(令牌)，即抢占事件发生，使得任务从运行位置转换到就绪位置的条件如下：

(i) 当产生中断时，就会发生抢占事件，在返回到这个中断之前，ISR 会向 RTOS 发出一个服务调用。当向 RTOS 发出这个调用时，令牌 preemptionEvent 会被设置。然后任务向 readyTaskPlace 位置转换，并且只在调度程序(通过发送 taskSwitchFlag)请求时才运行。

(ii) 每个 RTOS 都使用 RTCSWT 作为系统时钟。当 RTOS 中产生 RTCSWT(Real Time Clock driven Software Timer)中断时，就会产生一个抢占事件。当产生这个事件时，RTOS 就会控制处理器，检查是应该让当前正在执行的任务继续，还是抢占它，以便执行优先级更高的任务。当标识转换到后者时，这个事件使得另外一个优先级更高的任务准备运行。

(iii) 当发生对 RTOS 的调用，以进入临界段，或者向 RTOS 发送任务消息(输出)时，如果另一个优先级更高的任务此时需要服务(获得 CPU 的控制权)，此时，抢占事件就会发生(目前抢占操作还没有进入临界段)。

9.6.5 抢占式调度程序提供的临界段服务

对于临界段的服务来说，在某些情况下，当发生了超出更高优先级任务的服务需求时，抢占式调度程序不应该抢占优先级更低的任务。阻塞不应该在任何优先级更低的任务的临界段发生。这是为了避免发生共享数据问题(参考 8.2 节的内容)。

在进入临界段之前，运行中的任务接收从调度程序发送来的一个信号量，然后在退出临界段时把这个信号量释放掉。图 9-9 给出了一个 Petri 网，它对在运行期间具有临界段的任务建模，并设计代码，图中用圆圈表示位置，用矩形框表示转换。下面给出了位置和转换。

(i) RTOS 通过执行函数 task create ( )来初始化从空闲到就绪的转换。就目前情况而言，它是通过执行函数 task_J_create ( )来完成这项操作的。从空闲状态开始的任务转换的激发过程为：首先，RTOS 发送两个令牌，RTOS_CREATE 事件和 taskJSwitchFlag，转换的输出令牌是 taskSwitchFlag=true(请参考图左上方的转换)，每个任务开始时都处于空闲状态，发送到 RTOS 的令牌是 taskSwitchFlag=reset。

(ii) 考虑 task_J_Idle 这个位置，此时它在就绪任务中具有最高的优先级。当 RTOS 创建了 task_J 之后，位置 task_J_Idle 就转换到就绪状态(到 readyTaskPlace)，即 task_J_Ready 位置。

(iii) 当 task J 结束后，它就不再需要被 RTOS 控制了，这时 RTOS 向其发送 RTOS_DELETE 事件(令牌)，然后它返回到 task_J_Idle 位置，其相关标识 taskJSwitchFlag 也被重新设置(请参考图左下方的转换)。

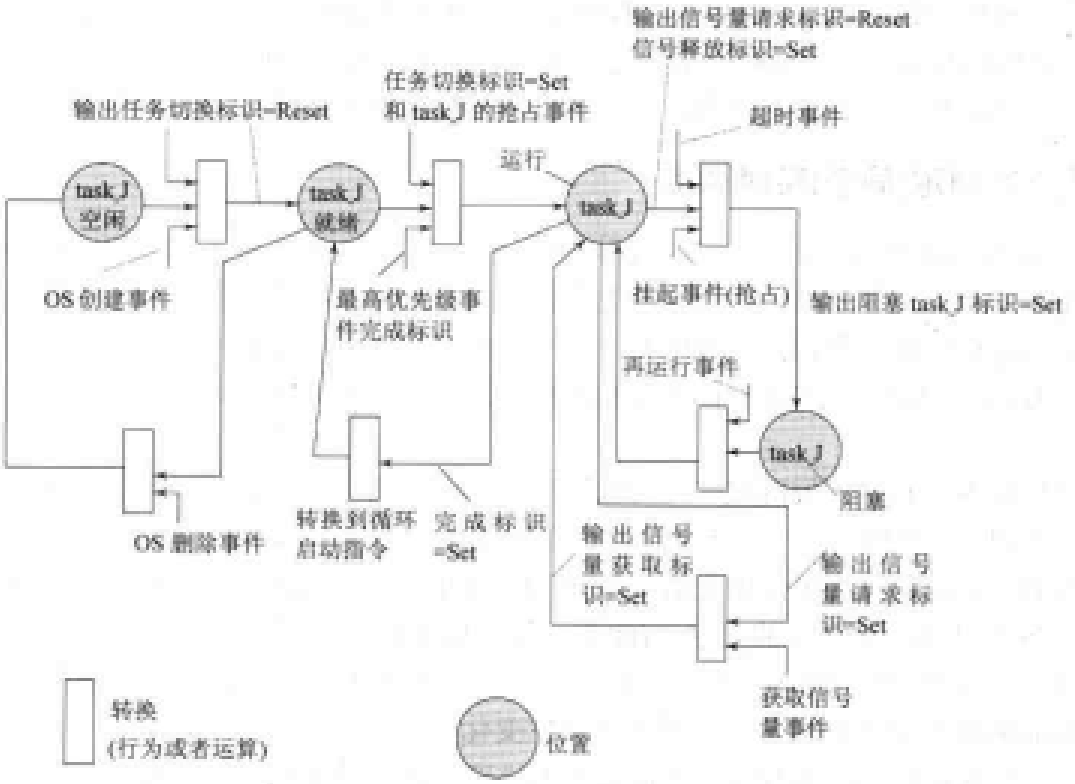


图 9-9　带抢占式调度程序和临界段的任务的 Petri 网，在临界段中，使用一个信号量，并在 criticalSectionOver 事件发生时释放

(iv) 在 task_J_Ready 这个位置，调度程序会考虑优先级参数，如果当前任务优先级刚好最高，则调度程序发送两个令牌，taskJSwitchFlag=true 和 highPriorityEvent=false，以便转换到 task J 的运行位置，即 task_J_Running。同时调度程序也会重置并发送令牌任务切换标识给所有其他优先级较低的任务，这是因为系统在任何时刻都只有一个 CPU 工作。

(v) 当任务结束标识被设置时，从 task_J_Running 位置到 task_J_Ready 位置的转换被触发。

(vi) 转换 task J 的代码在 task_J_Running 位置处被执行(参考图最右上方的转换)。

(vii) 当 RTOS 发送令牌 suspendEvent 时，抢占操作的转换在 runningTaskPlace 处被触发。另一个有用的令牌(当前是 time_out_event)也会触发这个转换。两种情况下都有效的是信号释放标识，不管什么时候，它一定会被设置。当执行完 task J 临界段的代码时，信号释放标识被设置。一旦被触发，下一个位置就是 task_J_Blocked。有两种情况可以引起中断：一个是抢占操作，当 runningTaskPlace 处的调用请求 RTOS 中止运行时，会发生 suspendEvent，此时就会执行抢占操作；另一种情况是与运行任务位置相关的 SWT 的超时。

(viii) 当存在 resumeEvent(RTOS 发出的一个令牌)时，就会发生到 task_J_Running 位置的转换操作(请参考右侧中部的转换，它处于图中三个转换之间)。

(ix) 当 RTOS 发送令牌 take_Semaphore_Event 请求 taskJ 接收信号时，在 *task_J_Running* 位置，会触发另一个转换，使 taskJ 回到 task_J_Running 位置(RTOS 会设置信号请求标识 take_Semaphore_Event，同时它把信号释放标识重新复位，并使 task J 运行而不会被中断)。

(x) 可以没有临界段，也可以有一个或者几个临界段。在临界段的执行过程中，RTOS 使信号释放标识复位，同时设置接收信号事件令牌。

如果没有临界段信号或者不对其进行人工干预，此时还能行得通吗？答案是肯定的，一种

策略就是禁用和使用抢占操作。禁用抢占操作只是禁止任务转换标识的变化，或者在使用共享数据时，禁止它们传送到任务就绪位置，之后就允许任务转换标识的改变和传送。但需要保证的是，所有的 ISR 都是再入式的函数。另一个策略是使用资源锁定信号 mutex(参考 8.2.2(iii))。

9.6.6 任务的固定(静态)实时调度

9.6.3 节叙述的调度方法被称为“固定实时调度”。假设有 m 个任务和 m 个 RTCSWT，则调度程序能够为每个任务分配一个固定的进度表。在相应的定时器发生超时的时候，任务从就绪位置转换到运行位置。我们假定 RTOS 为每个任务定义了硬实时进度。

9.6.1、9.6.2、9.6.4 以及 9.6.5 节介绍的调度方法也是固定时间的调度方法，称为软实时调度方法。

当进度是静态并且确定的时候，我们就认为调度程序使用了固定时间的调度方法。当过程在系统的单 CPU 上调度时，工作环境也不会发生变化。由于最坏情况下所有中断和任务的延迟可以预先确定，因此进度表是确定的，而 RTOS 调度程序也因此能够定时调度每个任务，使得所有的任务都可以满足时限的要求。(此时，每个任务的最坏情况下的延迟都小于它的服务时限)“无时限超出”的好处只有在工作环境确定的情况下才能得以体现。任务的编码使得执行时间在不同的输入和不同的条件下都不会发生变化。

在一个固定时间的调度程序中，先前定义的进度仍然是静态的。固定进度可以通过下面三种方法进行定义：

(1) 模拟退火方法。不同的进度都是可以固定的，而性能也可以模拟。现在，通过改变 RTCSWT 定时器设置(使用对应的 OS 函数)，直到模拟结果表明所有的任务都满足了时限的要求，或者无法满足时限要求的数量太小而不会影响系统性能，此时任务的进度是逐渐增加的。

(2) 启发式方法。此时，推理或者过去的经验都有助于定义和确定进度。

(3) 动态程式模型。具体来说，特定的运行程序首先会为每个任务确定进度，然后 RTCSWT 从这个程序的输出中加载定时器的设置。

如果调度程序不能固定进度，这就是一个不确定情况。例如，在网络中期望从另一个系统得到某个任务的消息，然而接收这个消息的最小和最大时间间隔却不知道；同样，当期望从另一个系统得到一个任务的输入时，而当输入被接收时最小和最大时间间隔也不知道。

下面是动态调度模型：在运行过程中，当一条消息或者错误消息被接收时，软件设计使优先级可以被重新调度，并且使固定的时间可以被重新定义。

9.6.7 调度算法中的优先级分配

最好的策略是基于 EDF(Earliest Deadline First，最早时限优先)优先的策略。与最早产生中断，且时限最早到期的中断源相应的任务的优先级被设为最高。

对不同的任务和 EDF 来说，在 CPU 负载可变的情况下，优先级是怎样分配的呢？一种方法如下。

假设 t_1 是任务 1 第一次需要抢占的时刻，t_2 为下一个时刻。则具有时间间隔最短 $(t_2\text{-}t_1)$ 的任务被插入到任务优先级列表的最顶端，同时它被分配具有最高的优先级。列表是根据 $(t_2\text{-}t_1)$ 动态调整顺序的。

首先，在高级调度算法中有确定的或者静态的优先级分配，也就是说，一开始是等速率调

度(RMS)，然后调度程序动态分配，再重新确定超时延迟，然后调整每个 EDF 的优先级。由于任务分布不均匀，以及分布式或多处理器环境等因素，所以需要动态分配。

9.6.8 使用概率定时 Petri 网(随机)和多线程图(MTG)的高级调度算法

在并发任务系统、多处理器或者分布式系统(请回顾 6.3.4 节的内容)中，超时延迟必然是随机的，系统也是随机的。当运行中的任务不确定时，输入和输出的令牌和操作也只是随机的(不确定的)，上述的调度模型策略(从 9.6.1 到 9.6.7 节)也就无用了。此时调度必须是动态的和随机的。在一个不确定环境中，调度程序必须使用随机调度策略。在高级调度算法中，一种方法是使用随机定时 Petri 网模型(6.3.4 节)(关于模型的具体细节，请参考 Tadao Murata 编著的 *Petri Nets Properties, Analysis and Applicatoins* 一书。

MTG 是两层模型，它集成了随机定时 Petri 网和受控数据流(CDF)图(关于 CDF 图，请参考 6.1.2 节的内容)。通过集成，使系统性能达到最优化。之前曾经发现它适合于具有并发任务的实时嵌入系统的建模。MTG 也可以用于并发任务调度程序的验证和性能评估，同时，它对集成的实时数字信号处理系统也同样适用。

注意：

(1) 下面是 RTOS 对任务的调度和处理模型：(i)在循环队列中对就绪任务进行循环协作调度，(ii)具有优先级约束的协作调度，(iii)具有时间片的循环协作调度，(iv)抢占式调度。(2)下列调度的周期是固定的：(a)固定时间的调度，(b)周期的、零散的和非周期任务的调度，(c)使用 EDF 的动态实时调度。(3) 使用随机定时 Petri 网或者多线程图的高级调度算法，适用于多处理器系统和复杂的分布式系统。

9.7 周期、零散以及非周期任务的调度模型的性能测度

已经为测量性能提出了不同的模型。下面给出了三种性能测度：

(i) 中断延迟总和与执行时间总和之比

(ii) CPU 负载

(iii) 最坏情况的执行时间与平均执行时间的比较

在不同的任务模型中，“中断延迟”可以用来评估性能。不同任务模型的延迟在 9.6 节中已有介绍。CPU 负载是察看性能的另一种方式，在 9.7.1 节中有详细的介绍。最坏情况下一个零散任务的性能计算将在 9.7.2 节中阐述。此外，还有许多可选的模型用于性能评估，详细内容请参考 Jane W. S. Liu 编著的 *Real Time Systems* 一书，由 Pearson Education 出版社在 2000 年出版。

9.7.1 使用 CPU 负载作为性能尺度

每个任务都给 CPU 一个负载，它等于任务执行时间除以任务周期(任务周期指的是分配给该任务的一段时间)。在示例 4-1 的 In_AOut_B 内部网络中，接收端口 A 在 172 μs 内期望得到另外一个字符，任务周期就是 172 μs 。如果任务执行时间也等于 172 μs，则该任务的 CPU 负载就等于 1(100%)。当一个字符被接收时，任务的执行时间一定小于 172 μs，因此 CPU 的最大负载等于 1(<100%)。

下面介绍多任务情况下的 CPU 负载或者系统负载的评估。假设有 m 个任务，对多任务来

说，所有任务和 ISR 的 CPU 负载之和应该小于 1。任务的超时和定时期限定义减少优先级较高任务的 CPU 负载，以便使优先级较低的任务也可以在时限之前运行。当 CPU 负载之和等于 0.1(10%)时，这意味着什么呢？这意味着 CPU 没有被充分使用，它在 90%的时间内都处于等待状态。因为执行时间和任务周期都有变化，因此 CPU 负载也会相应变化。

当任务只需要运行一次时，那么在应用程序中，它就是非周期性的(一次性的)。需要以固定周期运行的任务的调度可以是周期性的，这时 CPU 负载非常接近于 1。下面是一个周期性任务的例子。在预定周期的端口上可能有一些输入量，这些输入前后连续，没有任何时间间隔。

如果任务不能按照固定的周期调度，那么其调度是零散的。例如，如果期望任务在可变的时间间隔内接收输入，那么这个任务的进程就是零散的。一个例子就是网络中从路由器发送的数据包。可变的时间间隔必须小于定义的界限。

(i) 对于非周期性的任务，只需要执行一次抢占操作。

(ii) 对于周期性的任务，需要在固定的周期之后执行抢占操作，而且必须在下一个抢占操作之前运行这个任务。

(iii) 对于零散的任务，在它出现的最小时钟周期之后，需要对它进行检查是否需要抢占操作。通常来说，对零散任务，软件设计人员采用的策略是保持 CPU 负载在(0.7 ± 0.25)之间。

9.7.2 零散任务模型

考虑下面的参数

T_{total} ＝零散任务出现的周期总长度

e ＝总的任务执行时间

T_{av} ＝两个零散任务之间的平均周期

T_{min} ＝两个零散任务之间的最小周期

定义最坏情况下执行时间性能尺度 p。在一个模型中，任务在最坏情况下的 p 可以这样计算

$$p=p_{worst}=(e*T_{total}/T_{av})/(e*T_{total}/T_{min})$$

这是因为零散任务出现的平均速率等于(T_{total}/T_{av})，它出现的最小速率等于(T_{total}/T_{min})

注意：

有许多模型都可以用来定义性能测度。在 RTOS 进行调度管理时，三种性能测度为：(i)与执行时间相关的中断延迟；(ii)CPU 负载；(iii)最坏情况下的执行时间。

9.8 为 RTOS 的标准化和任务内部通信函数采用的 IEEE 标准 POSIX 1003.1B

进程的编码方法不是一成不变的。例如，(i)可以使用某种错误报告机制，也可以不使用；(ii)在进入临界段之前，与调度程序的消息通信、信号的使用以及时间标识都可以通过不同的途径来处理。对互斥来说，可以使用布尔变量，使用 P 和 V 信号量，或者使用一个符号位。

标准化要求是很重要的，没有它，就很难得到使用接口的能力以及接口的可移植性。IEEE

规范给出了一个标准 POSIX。1996 年后，标准的 RTOS 接口是 POSIX 1003.1b，而在此之前是 POSIX 1003.4。

POSIX 1003.16 定义的规范有：

(i) 应用程序和系统实现的可移植性

(ii) 定义了 RTOS 服务接口、子例程、环境变量、标题和结构

(iii) 确保可移植性、死锁恢复、错误报告以及错误恢复

(iv) 使用 POSIX 定时器和时钟接口的具有更高分辨率的定时器设置和读取设置

(v) 使用 POSIX 存储器管理共享存储器

(vi) 优先级调度和基于抢占式调度的优先级

(vii) 创建和访问实时文件以及它们的确定性能

(viii) 对进程锁定存储器块

(ix) I/O 同步

(x) 使用 POSIX 的异步 I/O

(xi) 定义具有性能矩阵的一个实时系统的性能

(xii) 定义上下文切换时间，中断延迟(出现中断和执行第一条指令之间的时间间隔)，过程调度延迟(中断发生到返回进程之间的时间间隔)

(xiii) RTOS 的特定语言服务

(xiv) 使用 POSIX 信号、信号量和消息队列的同步和异步的进程间通信

支持这个规范的 RTOS 也支持使用 IPC 的消息发送、流以及资源控制，具体如下：

(1) 使用 P、V 信号量提供有效同步，P 信号量用于访问一个资源，V 用于释放该资源。

(2) 对进程间通信提供支持，例如，使用管道。

(3) 向 RTOS 通知所有同步事件，同时也通知所有使用确定性发送排列事件的同步事件。

(4) 支持管道锁定(参考 8.3.2 节的内容)。

(5) 支持优先级继承，以解决优先级倒置的问题(参考 8.2.3 节的内容)。

POSIX 1003.1b/1c 同时也包含线程的定义以及线程机制，相关细节可以参考 8.1.3 节的内容。像任务一样，线程也是一个单元，它包含一套指令，RTOS 只能把资源调度并分配到该单元。只有在 RTOS 识别内部线程时，RTOS 才能调度多线程(或者任务)。轻量级进程指的是那些上下文切换时间较少，且 RTOS 拥有全部存储器管理信息的进程，而重量级进程需要有单独的存储器管理和资源。

为了使用 POSIX 1003.1b 标准接口，VxWorks(在 10.3 节中有定义)还有一些附加的函数，对上文提到的 14 种接口的绝大多数，VxWorks 都提供了 POSIX 库以便使用那些接口。然而，在提供的 VxWorks 版本中，可能并不支持 POSIX 存储器映射功能，也不支持开放、受保护、无链接、非异步同步的共享存储器功能，以及文件和数据的同步通信。

注意：

POSIX1003.b 定义了 IPC 和其他的标准接口。在实时程序设计和多任务、多处理系统中应该遵循这些标准。

9.9 抢占式调度程序的基本操作及其在处理器上预期耗费的时间

表 9-11 给出了抢占式 RTOS 的基本操作列表。预期耗费的时间一般取决于特定的嵌入式系统的处理器以及存储器的访问时间。然而，为了提供抢占式调度程序进行基本操作所耗费的时间的相对量值，它定义了一个新的参数。根据一个假定的比例参数 S，它定义了 RTOS 的一项操作所耗费的时间。S 强调了在典型的 RTOS 中进行不同操作的执行时间的相对量值。

假设最简单的指令所耗费的时间为 t_{min}。最小时间是当信号量 P 和 V 被赋予某个初始值，true 或 false 时。S 以 T_s 为单位进行定义。T_s 和 t_{min} 取决于特定的嵌入式系统的处理器以及存储器的访问时间。例如，对于一个基于 80960 的目标处理器，t_{min} 为 0.6μs，并假设 $T_s=(4\mu s \pm 0.6\mu s)$。假设 $t_{exec}=t_{min}+S\cdot T_s$。该等式将 S 定义为以基本时间单位 T_s 为单位，超过 t_{min} 的部分时间。

如果同一个 RTOS 在不同的处理器上运行，则 S 将保持不变。S 取值为与调度程序操作执行时间的相对量值最接近的正整数。

在 t_{min}=0.6μs 的 80960 处理器上，S 的意义如下：

(1) $S=1$ 表示 t_{exec} 在 2.8~5.2μs 之间。

(2) $S=2$ 表示 t_{exec} 在 7.4~9.8μs 之间。

(3) $S=4$ 表示 t_{exec} 在 14.2~19.0μs 之间。

(4) $S=5$ 表示 $t_{exec}=20.6\mu s \pm 3.0\mu s$。

(5) $S=10$ 表示 $40.6\mu s \pm 6.0\mu s$。

(6) $S=15$ 表示 $60.6\mu s \pm 9.0\mu s$。

表 9-11 抢占式 RTOS 的基本操作及其基于比例参数 S 的执行时间

操　作	S	操　作	S	操　作	S	操　作	S
上下文切换	2	任务挂起	1	信号量释放/获取	1	消息队列删除	10
任务启动	12	任务恢复	1	信号量可用时获取信号量标志、互斥体以及计数信号量	1	队列接收可用消息	2
任务创建并运行或删除	28	任务锁定或任务解锁	tmin	无可用信号量时获取信号量标志、互斥体、计数信号量	1	队列接收不可用消息	1
任务删除	10	存储器分配	2	信号量标志、互斥体以及计数信号量的创建或删除	6	消息队列发送任务挂起	5
任务切换标志		存储器释放	4	当任务在队列中时释放信号量标志、互斥体、计数信号量	3	消息队列发送任务未挂起	2

(续表)

操　作	S	操　作	S	操　作	S	操　作	S
任务创建或删除	18	网络字节发送	t_{min}	互斥体刷新	1	消息发送队列已满	1
		信号量标志或计数信号量刷新	4	当队列中无任务时释放信号量标志、互斥体以及计数信号量	1	消息队列创建	105

注意：

当使用 RTOS 进行程序设计的时候，抢占式调度中的基本操作和 IPC 功能变得非常重要。不同操作的相对速度可以帮助程序员优化响应时间，提高系统的性能。

9.10 用于进程间、ISR 间、OS 函数间和任务之间同步及资源管理的 15 条策略

单 CPU 的嵌入式系统在某一时刻只能运行一个进程。在任一时刻，该进程可能是 ISR、调度程序或者任务(9.5 节)。RTOS 采用了 9.5.1、9.5.2 以及 9.5.3 小节中介绍的三种策略之一(参考图 9-4 中关于同步 ISR、调度程序以及任务的叙述)。在代码的设计过程中，对于进程间同步，必须考虑下面十五个要点。

(1) 采用适当的优先级分配策略。

(2) ISR 只能用来为 RTOS 写(发送)消息和为任务写(发送)参数。ISR 的指令都不应该阻塞任何任务。只有 RTOS 必须根据用于任务交互的变量、信号量、队列、邮箱和管道[8.3 小节]来启动相应的操作，而且 RTOS 只能控制任务的状态。变量和任务切换标志必须总是处于 RTOS 控制之下。ISR 的指令不能读取和改变任务和 RTOS 的任务切换标志，或者发送给任务和 RTOS 的消息。

(3) ISR 代码的编写应该仿照没有共享数据问题的可重入函数进行。中断服务程序应该简短，并且在其执行过程中不应该等待 RTOS 和任务的操作。

(4) 任务不能调用其他任务，因为每一个任务都必须在 RTOS 的控制之下。试图进行这样的操作应该产生一个错误。

(5) 任务只能使用 RTOS 的调用获取和发送信息。

(6) 如果可能的话，执行临界段代码时，只应该避免任务切换标志的改变(这是通过使用信号量实现的)，而不是禁止中断。因此，只应该禁止 RTOS 的抢占操作。禁用抢占可能要比禁止中断好一些。然而，这两种情况都会增加最差情况下的中断延迟(回顾 4.6 节)。

(7) 采用互斥信号量进行资源锁定可能比禁用抢占或者中断要好(参考 8.2.1(v)的互斥信号量)。使用互斥信号量函数 P 和 V(8.2.1(vi))是锁定资源的一种方法。

(8) 任务应该只在临界段单独执行的一个较短的周期内获取信号量。在更长时间里禁止其他任务的运行，将会增加所有中断在最差情况下的中断延迟。

(9) 多任务处理可以改善相对响应时间，但是需要对任务的数量加以限制。它减少了上下文切换的时间间隔和堆栈的需求。否则，就会失去响应时间的优势，且任务所需的堆栈将占用更大的存储器。

(10) 记住，RTOS 用于创建任务的 create( )函数所需要花费的 CPU 时间，比向队列执行读写操作要长，使用信号量花费的时间最少(表 9-3)。所以，在满足需要的情况下，应该尽可能地使用信号量。获取信号量需要耗费的 CPU 时间最少。对于最紧急的进程间通信应使用信号(例如，通过产生异常来报告错误)(8.3.1 节)。

(11) 只在启动的时候创建任务，避免在后面创建和删除任务。删除任务的惟一优点在于可以获取额外的存储空间。假设一个任务被 RTOS 的 delete( )函数删除。现在的情形可能是任务正在 RTOS 中等待一个信号量(让其他的任务完成临界段)或者等待队列消息以获得一个指针，这个指针指向一条发送给已删除任务的消息。长时间的阻塞、致命的循环或者死锁就会发生。RTOS 可能并不提供对这类情况的保护。

(12) 信号量、队列和消息之间不应该共享变量，每一个只应该在一组任务间共享并与其他任务隔离。

(13) 对于内部函数官使用 CPU 空闲时间。CPU 经常不运行任何任务。所有的任务都在等待抢占(从就绪位置转换到运行位置)。这时，CPU 可能进行 RTOS 的以下操作：读取内部队列，管理存储器，查找空的存储器块，删除或者分派任务，执行内部的和 IPC 函数。

(14) 如果存储器分配和释放由任务完成，则 RTOS 的函数数量可以减少。这降低了中断延迟周期，因为无论何时 RTOS 抢占了一个任务，由 RTOS 来执行这些函数都会耗费大量的时间。

(15) 使用可配置的或者分层的 RTOS，这样的系统可以只能放置调度程序需要的函数，而将其他的放置在外部。这是因为时间都耗费在上下文切换以及保存和获取 RTOS 函数的指针上了，如存储器分配，进程间通信，从而增加了中断延迟周期。如果分层 RTOS 的存储器管理函数、文件系统函数、进程间通信(例如：管道、信号、套接字以及 RPC)处于调度程序外部，则中断延迟可以达到最小。分层 RTOS 的意思是 RTOS 只对调度程序分派有限的函数，而其他的函数则在需要的时候进行动态链接和绑定。

注意：
这十五条策略是为嵌入式系统的实时程序设计提出的。

9.11 嵌入式 LINUX 的内部组织：设备驱动程序和嵌入式系统的 LINUX 内核

下面的描述中假定读者已经具备了 Unix 和 Linux 操作系统的知识(操作系统的名称 Linux 是根据 Linux 操作系统之父 Linus Torvalds 命名的)。这些系统的详细信息可以查阅参考文献中列出的标准参考书。用户应用程序(任务)为实现下列功能在内核中进行系统调用或者消息传递(表 9-3)。

(i) 进程管理功能。

(ii) 存储器管理功能。(例如，分配、释放、指针、创建或删除任务)。

(iii) 文件系统功能。

(iv) 网络系统功能。

(v) 系统(计算机)的任何外围设备的设备控制功能。

下面是三类重要的设备：

(1) 一种是字符设备。并口就是一个例子。液晶显示阵列、串口、键盘、鼠标都属于这类设备。字符的访问是逐个字节进行的，类似于访问打印设备。和 I/O 设备一样，系统调用函数也可以打开、关闭、写入和读取。

(2) 另一种是块设备。它就像一个文件系统(磁盘)。Linux 允许块设备像字符设备一样逐个字节地读取和写入，或者成组读取和写入。部分块是可以访问的(在 Unix 系统中只能成块进行读写)。

(3) 类似于以太网、FTP、TCP、IP、UDP 的传输协议。这些设备称为网络设备(尽管 FTP 和 Telnet 像字符设备一样，也归入网络设备)。网络设备允许通过传输协议交换数据包。Linux 为套接字和来自网络设备的基于协议的传送提供操作系统功能(套接字为网络发送器和网络接收器提供逻辑连接)。Linux 内核将设备映射成文件系统的一个结点。网络设备的驱动程序提供地址解析协议(ARP)和逆向地址解析协议(RARP)。

嵌入式系统的设备控制函数非常重要。设备驱动函数控制了移动电话或电视遥控器的键盘。这些函数控制了移动电话或视频游戏的液晶显示阵列。因此，设备驱动程序在嵌入式系统中非常重要，尤其是在与网络相关的嵌入式系统中。回顾 4.1.4 节，表 4-1 描述了 Linux 的驱动器。下面将对 Linux 的重要方面进行阐述。

早期，OS 将设备分为字符设备和块设备。网络子系统被加入到操作系统中，以控制网络设备和子系统。(9.3 节)尽管每个设备最后是按照字符或者块流来执行，在这些复杂设备的内部，软件的执行是不同的。两个重要的问题是：

(1)为什么需要定义另一类设备和设备驱动器？

(2)是否有可能在操作系统中定义除了块设备之外的设备类，如文件？

第一个问题的答案如下。回顾 3.3.3 节，USB 设备的特点在于它可以被连接、配置和使用、重启动、重新装配和使用、与其他设备共享带宽以及在其他设备运行的过程中进行断连和连接操作。文件系统不具备这个特征。

Linux 给出了第二个问题的答案。Linux 操作系统简化了设备分类和设备驱动器模块的定义。一个模块可以是 USB 类的设备或者 SCSI(Small Computer System Interface，小型计算机系统接口)。在 Linux 中，可以获取越来越多的设备及其模块。Linux 代码的最大优点在于其开放式源代码。

Linux 内核可以通过注册插入一个模块，也可以通过注销删除它(注册意味着当轮到它的时候就预定。注销意味着这个模块将被忽略。这和任务的产生和删除相似)。MIcroC/OS-II(10.2 节)仅仅是注册和注销任务。VxWorks(10.3 节)则不仅对任务进行注册和注销，而且还对中断服务例程(ISR)进行注册和注销。Linux 内核也支持设备驱动模块的注册和注销。因此，Linux 内核也允许对设备驱动器和模块进行调度。所以，不同的任务和程序可以通过其在内核中注册的驱动器同步地或顺序地向设备发送字节。而且，Linux 内核通过注册和注销机制强制使用顺序访问，而且只能访问特定的存储器。这些特征使得 Linux 操作系统在嵌入式系统中特别重要。

Unix 假定不同的设备类型对应不同的任务。Linux 假定每个设备类型是一个模块。因此，Linux 可以降低存储器的使用，并提高速度，这是嵌入式系统最希望获得的特点。Linux 中的每一个模块执行一个驱动程序或者其中的一个函数，所以作为设备类型的分类是相同的。两个

8 位的数字被用来表示总线代号、设备代号和函数代号。

每一个设备必须有一个单独的设备驱动程序。每一个设备驱动程序都是一个可以通过注册植入到内核、通过注销删除的模块。不同的程序是否可以同步地或顺序地发送字节到设备？内核怎么强制使用顺序访问并指定存储器地址？设备是只接收 ASCII 码还是二进制数？设备有端口地址、地址、存储器空间需求、中断矢量、控制寄存器和状态寄存器。内核必须提供注册这些特征的方法。既然每一个用户有不同的需求，操作系统内核不能对用户使用的内核制定硬性规则。随后，内核可以使用设备管理程序执行存储器的缩放和映射。内核一定不能为设备强制规定任何东西。它接收设备的配置和需要。无论何时设备允许同步，内核必须执行设备驱动模块以提供支持。每一个设备都有一个内核必须支持的安全模块。单独使用内核底层来执行安全功能是非常危险的。

Linux 操作系统支持注册驱动程序的配置和函数，也支持使用表 9-12 中列出的函数进行注销(参考 9.5.3 小节的实时操作系统，该系统不仅支持任务的调度，而且还支持 ISR 的调度。Linux 调度任务和设备驱动的 ISR，而且还调度所有不同的设备模块)。

表 9-12 Linux 中设备驱动模块的注册与注销函数以及相关的函数

函　数	操　作
insmod	向 Linux 内核中插入模块
rmmod	从内核中删除一个模块
cleanup	核心级的 void 函数，当发生一次来自模块执行的 rmmod 函数调用时它就执行
register_capability	核心级的注册函数
unregister_capability	核心级的注销函数
register_symtab	符号表函数支持，它是另一种声明函数和静态变量的方法

NULL 参数将创建一个空的符号表。这解决了命名空间冲突的问题。问题来源于使用了与内核或者其他模块中相同的名字。Linux 支持模块的初始化、错误处理、禁止进行未授权的端口访问、使用计数、根级安全与清理。

如果包含了头文件 Linux/time.h，则内核可以支持定时函数的调度。延迟函数在 Linux/delay.h 头文件中。通过包含 Linux/tqueue.h 头文件，可以提供对任务队列的支持。任务队列提供的支持有定时器、磁盘、即时响应和调度程序。

在采用了嵌入式系统的计算机系统中使用 Linux 设备驱动程序来处理不同的外部设备和网络接口是非常有价值的。一个例子就是内置的调制调解器或者 PCI 卡(3.4 节)。(PCI 是个人计算机的总线系统。更老的则采用 ISA 总线系统)。当采用其他操作系统或实时操作系统时，设备驱动程序的设计可以避开 Linux 操作系统。但是，软件设计者必须清楚地知道 Linux 设备驱动程序中的函数。这样通过使用类似的代码设计策略，可以使编程更加简单，从而不必重复设计周期。

操作系统及其设备驱动程序是否可以和处理器整合起来？答案是肯定的。例如，一体化软件设备公司提供的 Linux 设备驱动程序将 Linux 内核 2.2 移植到了 ARM7 处理器上。

注意：

Linux 越来越多地被应用于嵌入式系统中。这不仅仅是因为它是一个共享软件，还因为它

强大的设备驱动程序特性、运行时内核代码的可扩展性以及设备驱动模块的注册与注销功能，从而像进程一样为它们的调度提供了方便。这些特点在复杂的嵌入式系统都是必不可少的。

9.12 操作系统的安全问题

当医生必须为多个病人配药时，避免病人的药物出现混杂就非常必要了(否则就会晚上吃了早晨的药片或者胃痛的吃了感冒药)。当操作系统必须管理多个进程和它们对资源的访问时，保护存储器和资源免于遭受任何向 PCB 或者资源的非法写入，或者避免对它们的访问发生混淆是必不可少的。操作系统的安全问题是一个至关重要的问题。

每一个进程都要确定它是否具有对某个系统资源的专有控制权，或者是否与其他进程相互隔离，或者是否共享了某组进程中公用的资源。例如，文件或者文件的存储器块将拥有对处理器的专有控制权，所有进程对空闲的存储器空间都具有访问权。操作系统然后配置何时将资源从进程中隔离出来以及何时将资源与一组已经定义的进程共享。

操作系统也应该具备在需要的时候改变这些配置的灵活性，以满足所有进程的需要。例如，进程需要在某一时刻拥有对 32 个存储器块的控制权，则操作系统将对系统进行相应的配置。随后，更多的进程被创建了，这就需要重新进行设定。

操作系统应该提供保护机制并执行系统管理员定义的安全规则。例如，系统管理员可以定义注册用户和授权用户(与它们的进程)对资源的使用权。

如果应用程序改变操作系统的配置会产生什么后果？操作系统需要一个自身的保护机制。程序员可以发现保护机制的漏洞并获取非法访问权。这样，对于每一个操作系统的设计者来说，保护机制的执行和资源的安全策略是一个极具挑战性的任务。网络环境加大了该问题的复杂性。

表 9-13 给出了实现重要的安全功能的不同操作。

表 9-13 重要的安全功能

功　能	行　为
受控的资源共享	通过用户进程来控制对资源和参数的读写。例如，某些资源只能让某个进程写，而某些资源只能让某一组进程读
限制机制	限制只能由某一组进程共享的机制
安全策略	授予对 OS、系统和信息的访问权的规则。例如，通信系统有一个点到点通信的策略(在数据包传递之前建立连接)
身份验证机制	用户的外部认证机制，和除非用户已经注册，且系统管理员(软件)已经授权，否则，禁止应用程序运行的机制。进程的内部认证，该进程不能类似于(模仿)其他进程。如果用户公布了密码或者其他认证方法，则用户身份验证将变得困难
授权机制	用户或进程允许按照安全策略使用系统资源
加密	用于改变信息，使得在没有正确密钥解码的情况下不可被其他用户或者进程使用的工具

注意：

操作系统的安全问题是一个重要的问题。使用 OS 的安全和保护机制对内存和资源加以保

护，避免出现对 PCB 或者资源进行任何未授权的写入，或者避免混淆对它们的访问是势在必行的。

9.13 移动式操作系统

一个崭新的时代即将来临。这就是掌上电脑和便携式 PC 的时代。移动式操作系统(OS)的意思是移动计算或者嵌入式系统(膝上电脑、手提式或无线设备)的 OS。Windows CE、Palm OS，Pocket 和 Mac OS X Jaguar(用来连接 VPN(Virtual Private Network，虚拟私有网络))都是移动式操作系统。Windows XP 与 Linux 也提供移动支持。MotionBlackBerry 双向寻呼机、iPaqPDA 和摩托罗拉 StarTac 电话都使用了移动操作系统。

Windows CE(用于 C 嵌入式软件的 Windows 系统)需要的空间很小，只要 400kB。其优点在于它对 Windows API 提供了支持。它为 Word 和 Excel 文件的下载提供了可行性，也使得网络浏览可行。它是一个 32 位可升级的实时操作系统，具有 256 个可定义的进程优先级。在 2003 年 4 月 9 日，微软宣布手提式 PC 等设备的嵌入式系统设计人员可以通过添加额外的功能，来使用和修改该操作系统。在源代码共享的费用许可项目下，出售修改后的 Windows CE 不需要缴纳额外的使用费。

Palm OS 在工业上有大量特殊的应用。Palm OS5 被期望具有 PocketPC 的特点。

当 TravRoute 使用它的移动式 OS 时，PocketPC 可以提供听音乐或者使用 GPS(Geographic Positioning System，全球定位系统)进行详细导航等服务。PocketPC 的 MicrosoftMoney 使用移动式 OS 来提供对 Word 和 Excel 文件的支持功能。PocketPC OS 还提供基于 PC 的个人视频记录应用程序接口。

基于 Linux 的 PDA 有 SharpZarus。还有基于“SprintPCS 视频”的 T-Mobile PocketPC 电话(带便携式 PC 的电话)。“Stringer”是微软为电话开发的一个新的移动式 OS。

注意：

移动式 OS 是一个用于移动系统、手持式设备、PDA、电话、膝上电脑和便携式电脑的 OS。PocketPC、Windows CE 以及 Palm OS 都是流行的移动式操作系统。移动系统的嵌入式系统直接利用移动式 OS，或者对其进行了适当的改变或添加。

本章小结

- 内核是任何操作系统的一个基本单元，它包含了存储器分配与释放、禁止非法访问存储器、任务调度、进程间通信、输入/输出管理、中断处理机制、设备驱动和管理的功能。操作系统也控制输入/输出和网络子系统。操作系统的内核还包含了文件管理功能和网络子系统的功能(这些功能在某些操作系统中也可能从操作系统的内核中分离出来了)。
- 实时操作系统中具有进行实时任务调度和中断延迟控制的函数。实时操作系统使用定时器和系统时钟。它包括时间分配与释放的函数，以在任务给定的定时约束下达到最佳利用。它有任务的同步机制，即在系统中使用 IPC、可预测的定时以及的同步来对任务进行同步。

- 调度多个任务的基本策略有按照中断发生的顺序进行循环、按照优先级进行循环、抢占式以及时间分片。循环调度的意思是对就绪任务列表中的任务按照顺序进行调度。这样，一个任务执行了，下一个最优先的任务接着执行。抢占式调度的意思是具有较高优先级的任务被调度程序强制阻塞，并让另一个较高优先级的任务运行。时间分片的意思是给每个任务分配一个时间片，过了这个时间片之后，该任务就被阻塞，直到下一个循环轮到它的时候再继续运行。
- 实时操作系统包含处理中断的功能。对于实时操作系统来说，有三种方法可以对硬件资源的中断调用进行响应。
- 实时操作系统具有优先级调度方案中处理临界段的功能。实时操作系统也可以提供固定的实时调度。
- 在使用实时操作系统进行应用程序开发时，抢占式调度程序中各种操作的相对时间可以帮助我们对进程的定时器进行优化。
- 最近，已经有了标准化(如 POSIX 1003.b)的实时操作系统和进程间通信函数。
- Linux 有很多设备驱动函数。Linux 假定每个设备函数就是一个模块。这样，Linux 就可以在降低对存储器使用的情况下获取更高的速度，这也是嵌入式系统所希望获得的特性。Linux 中的每一个函数执行一个驱动程序或者其中的一个函数，因此，作为设备类型都具有相同的分类方式。两个 8 位的数字被用来表示总线代号、设备代号和函数代号。
- Linux 操作系统包括对设备模块进行注册和注销的内部函数。Linux 操作系统的调度程序不仅完成进程同步的任务，而且方便模块、设备、中断服务例程和任务的同步。
- 流行的实时操作系统 MicroC/OS-II 的内核仅仅对任务进行注册和注销。另一个流行的实时操作系统 VxWorks 的内核在注册和注销任务的同时，也对中断服务例程进行注册和注销。Linux 内核还提供对一类驱动程序的设备驱动模块进行调度的功能。
- 在一个系统中，操作系统的安全问题非常重要，保护存储器和资源或进程控制块不被非法写入都是非常必要的。
- 移动操作系统应用在可移动的设备上，如无线设备、电话、个人数字助理(PDA)、掌上电脑和便携式个人计算机。

关键词及其定义

- 内核(kernel)：内核是任何操作系统的一个基本单元，它包括用于进程、存储器和任务调度，进程间通信、设备管理、输入/输出和中断的函数，某些操作系统也可能包含文件系统和网络子系统。
- 操作系统(Operating System)：一个包括内核函数、文件管理函数和其他函数的系统。
- 实时操作系统(Real Time Operating System)：具有实时任务调度、中断延迟控制、同步任务和进程间通信、可预定计时和对系统进行同步的操作系统。
- 轮转式或者循环式调度(Round Robin or Cyclic scheduling)：一种调度算法，按照顺序对就绪任务列表中的任务进行调度。
- 抢占式调度(Preempting scheduling)：一种调度算法，一个较高优先级的任务被调度程序阻塞，以让另一个较高优先级的任务运行。

- 时间片调度(Time slicing scheduling)：一种调度算法，在这种算法中，每个任务都会被分配一个时间片，在这个时间片之后，该任务被阻塞直到下一个循环轮到它的时候再继续执行。
- 协作式调度(Cooperative Scheduling)：一个任务等待直到另一个任务运行完毕。
- 临界段运行(Critical section run)：尽管还有更高优先级的任务在等待，使用信号量可以使调度程序允许临界段继续运行。
- 固定实时调度(Fixed real time scheduling)：一种调度策略，按照这种策略每一个任务的运行时间是固定的。
- Linux OS：一个功能强大的操作系统，具有类似于 UNIX 的特征和一些特殊的设备驱动调度的特征，以及存储器管理的特征。
- 注册设备(Registering a Device)：使得设备在实时操作系统中可用，像任务一样进行调度。
- 注销(Deregistering)：使得设备在实时操作系统中不能像任务一样进行调度。
- 保护机制(Protection Mechanism)：在操作系统中防止对资源非法访问的机制。
- 移动式 OS：用于移动的手持式设备、无线设备、掌上个人电脑、便携式个人电脑、个人数字助理和电话的操作系统。

问题回顾

(1) 操作系统的目标是什么？

(2) 列出应用层和硬件层之间的层。

(3) 为什么操作系统的函数提供两种模式，用户模式和管理模式？

(4) 列出内核函数。内核外部的函数是什么样子的？

(5) 说明进程描述符和进程控制块。任务控制块和进程控制块之间有什么相似之处？

(6) 对系统资源访问时，什么时候使用消息，什么时候使用系统调用？

(7) 进程或任务的创建和管理是内核中最重要的功能。为什么？

(8) 有一种策略是只在启动时创建任务，并避免在后面对任务进行创建和删除的操作。为什么要采用这个策略？

(9) 存储器分配和管理是内核中最重要的功能。为什么？

(10) 列出静态和动态块分配方式的优缺点。

(11) 内核控制对系统资源、CPU、存储器、输入输出子系统和设备的访问。为什么需要内核？

(12) 在嵌入式系统的操作系统中设备管理有什么重要性？

(13) 给出一个输入输出子系统的例子。

(14) 定义一个网络操作系统。网络操作系统和传统的操作系统之间有什么区别？

(15) 操作系统的可协作性和便携性有什么作用？

(16) 针对周期任务、非周期任务和零散任务，如何选择调度策略？

(17) Linux 设备驱动程序有什么独特的性质？

(18) 块文件系统是什么？

(19) Linux 可以像对任务一样为设备驱动模块提供注册和注销功能。为什么这个特征对于嵌入式系统非常有用处？

(20) 在实时操作系统中操作系统的单元是什么？

(21) 什么时候采用协作式调度，什么时候采用抢占式调度？

(22) 比较以下调度策略：实时调度、轮转模式和轮转调度。

(23) 实时操作系统的时间片调度有什么优点？

(24) 举出三种实时操作系统在多任务条件下处理中断服务例程的方法。

(25) 抢占事件是怎么发生的？

(26) 实时操作系统的性能是通过中断延迟，平均响应时间和错过的时限来度量的。说明每个度量标准的重要性。

(27) 为什么必须估计最差情况下的延迟？

(28) 举例说明模拟退火方法的应用。

(29) 操作系统的安全策略应该是什么？

(30) 操作系统的保护机制是什么？

(31) 操作系统的安全问题是需要重点考虑的事项。考虑到操作系统的安全和保护机制，保护存储器和资源防止非法写入进程控制块或混淆相互之间的访问权限已经势在必行。说明每一个需要考虑的事项。

(32) 分层实时操作系统是什么意思？

(33) 对于电话和便携式个人电脑的嵌入式系统来说，为什么移动式操作系统变得非常受欢迎？

(34) 任务的优先级分配是怎么实现的？在动态程序设计中，优先级分配算法是怎么使用的？

(35) 列出同步任务和中断服务例程的最佳策略。

(36) 什么是动态调度？

实践练习

(37) 给出调度程序不能确定进度且情况不确定的两个例子。

(38) 怎样评价多任务系统处理零散事件时的 CPU 负载？

(39) 什么情况下采用 CPU 负载作为实时操作系统的性能测度？

(40) 为图 9-9 中的 petri 网设计 petri 表，然后用该表来编写 C 代码。

第 10 章　实时操作系统编程工具：MicroC/OS-II 和 VxWorks

本章前所学内容

在第 8 章，我们学习了有关 RTOS 实时系统编程服务的知识，主要有如下几个要点：

(1) 进程是 OS(操作系统)调度的计算单元。进程以请求的方式，即通过系统调用或者消息传递，向 OS 申请使用 CPU、存储器、I/O 子系统、设备和网络子系统等资源。

(2) 进程在多任务模型的处理过程中也有“任务”的含义，同样，在多线程模型的处理过程中有“线程”的含义。这两种模式下的进程都是由 OS 控制的。

(3) 任务的并发处理过程中，进程间的通信通过信号、信号量、队列、邮箱、管道、套接字和 RPC 实现同步。

在第 9 章中，我们已经学习了有关 RTOS 实时系统编程服务的知识，主要有如下几个要点：

(4) 系统结构包括应用程序软件、应用程序编程接口(Application programming Interface，API)、系统软件(不包括 OS 中提供的)、OS 接口、OS、硬件 OS 接口(hardware-OS interface)以及硬件。

(5) OS 的基本功能(服务)是对进程从创建到删除的全过程的管理、处理资源请求、存储器管理(分配和释放)、进程调度、处理并管理进程间通信(任务、ISR 或者 OS 函数之间的通信)、I/O 子系统管理、管理中断控制(通过处理 ISR)机制、设备和设备驱动程序的管理，以及在 Linux 中处理每一类设备和核心设备驱动程序的设备驱动程序模块。

(6) 中断和任务调度的处理通过 RTOS 来完成。RTOS 有些基本的 OS 函数，以及用于实时任务调度和中断延时控制(参见 4.6 节)的函数；它使用定时器和系统时钟，能在给定的时间限制内获得最佳时间使用效率的时间分配和释放，系统的可预测定时行为和可预测任务同步，优先级分配和优先级继承，带有 IPC 的任务同步，进程执行的时间分片，以及硬实时和软实时操作。

(7) 调度多任务的基本策略是循环(协作)、抢占式调度和时间分片。

(8) POSIX 标准。

(9) 不同 RTOS 函数的相对定时。

(10) 在为进程(ISR、函数、任务和调度程序函数)间同步进行编码的过程中应该注意的要点。

任何嵌入式软件的目标，以及 RTOS 的目标，都应该是完美无缺的。读者肯定也已经意识到，使用高级语言进行嵌入式系统的实时编程相当复杂。本章的目的是介绍两个流行的 RTOS，在这两个系统中，大大简化了编程的复杂程度，也明显缩短了设计嵌入式系统的代码所需要的时间。

本章将学内容

本章我们将要学习如下有关内容：

(1) 用在复杂实时应用程序中的嵌入式系统需要实现 RTOS 功能和进程间通信(IPC)。在多进程处理应用程序所依赖的多任务多线程系统中，为什么需要有一个测试稳定且调试合格的 RTOS？

(2) 通过 20 个示例(示例 10-1 至 10-20)，详细阐述什么是 RTOS MicroC/OS-II(也称作 μC/OS-II，或者 MUCOS，或者 UCOS)。本章还将详细解释每一个给定的 MUCOS 函数传递的参数和返回值。学习 MUCOS 中函数的使用，对于每个读者来说都是很重要的。即使以后读者会使用其他的 RTOS，但同样也会用到这部分知识。所有的这些内容对于理解高级复杂的嵌入式 RTOS 都将有很大的帮助。

(3) Wind River®系统中的 VxWorks 也是用在复杂嵌入式系统中的一种 RTOS。VxWorks 有许多功能强大的特性。我们将通过 7 个示例(示例 10-21 至 10-27)来学习 VxWorks。10.2 节中涉及的 VxWorks 信号量、邮箱和队列之间的区分，相对于 MUDOS 中的信号量、邮箱和队列的区分而言，将会更加清楚明了。

10.1 测试稳定且调试合格的实时操作系统的必要性

回顾一下 9.4.2 节的知识，除了设计小规模嵌入式系统之外，对于一些较大较复杂的的嵌入式系统的设计和开发，RTOS 的代码都是必需的。在一个复杂嵌入式系统的设计过程中，设计人员需要有如下一些软件组件的代码，这些代码要求测试完全合格并且没有错误。

- 嵌入式 C 或者嵌入式 C++中的多任务函数(参见第 5 章)。
- 采用系统(硬件)时钟的实时时钟软件定时器(Real-time clock-based software timers，RTCSWT)。
- 用于协作调度程序及其测试的软件(9.6.1 节、9.6.2 节或 9.6.3 节)。测试的内容是看它是否可以使 CPU 的负载总和维持在 0.7 左右，如果情况比较坏，但是否可以保证中断延迟、响应时间和最后期限在允许的范围内(参见 9.6.7 节)。
- 如果协作调度程序没有起作用，并且导致进度超过了任务最后期限，那么我们就应该采取备选方案，开发一个抢占式调度程序，即要么采用单调速率调度(Rate Monotonic Scheduling，RMS)——需要最高速率服务的任务必须以最高优先级进行分配，要么采用最早时限优先(earliest deadline first，EDF)调度优先级策略。我们可以利用 Petri 表来进行软件设计。抢占式调度程序中的每一项任务必须有其优先级，它包含在被分配的任务各自的控制模块中(8.1.2 节)。抢占式调度程序为具有较高优先级的任务提供更短的响应时间。
- 设备驱动和设备管理器。

- 使用任务切换标志进行进程间通信的函数、信号量处理函数，以及用于信号、队列、邮箱、管道和套接字的函数。
- 网络函数。
- 错误处理函数和异常处理函数。
- 测试和系统调试软件。

总之，在一个复杂嵌入式系统的设计中需要大量的编码工作，当开发一个产品时，需要能在一个合理的时间内完成这些工作。一个就绪可用的 RTOS 程序包具有这样的优点：在那些提前测试和调试过的 RTOS 函数，以及错误和异常处理函数已经经过众多用户测试使用后，RTOS 程序包可以方便地获得测试结果。

当设计一个任务要求苛刻的实时应用程序时，如果缺乏适当的错误处理能力，或者缺乏适当的 RTOS，或者缺乏测试和调试工具，那就可能导致数据的丢失，甚至还可能损害硬件。因此，对于开发者，一个就绪可用的、测试稳定和调试合格的 RTOS 不仅可以大大简化编码过程，而且对于快速形成产品也大有裨益。其目标是能在代码植入硬件之前，通过测试和仿真，建立强大无错的软件。

图 10-1(a)给出了可用于选择 RTOS 的常见选项。图 10-1(b)给出了 RTOS 中需要的基本函数。

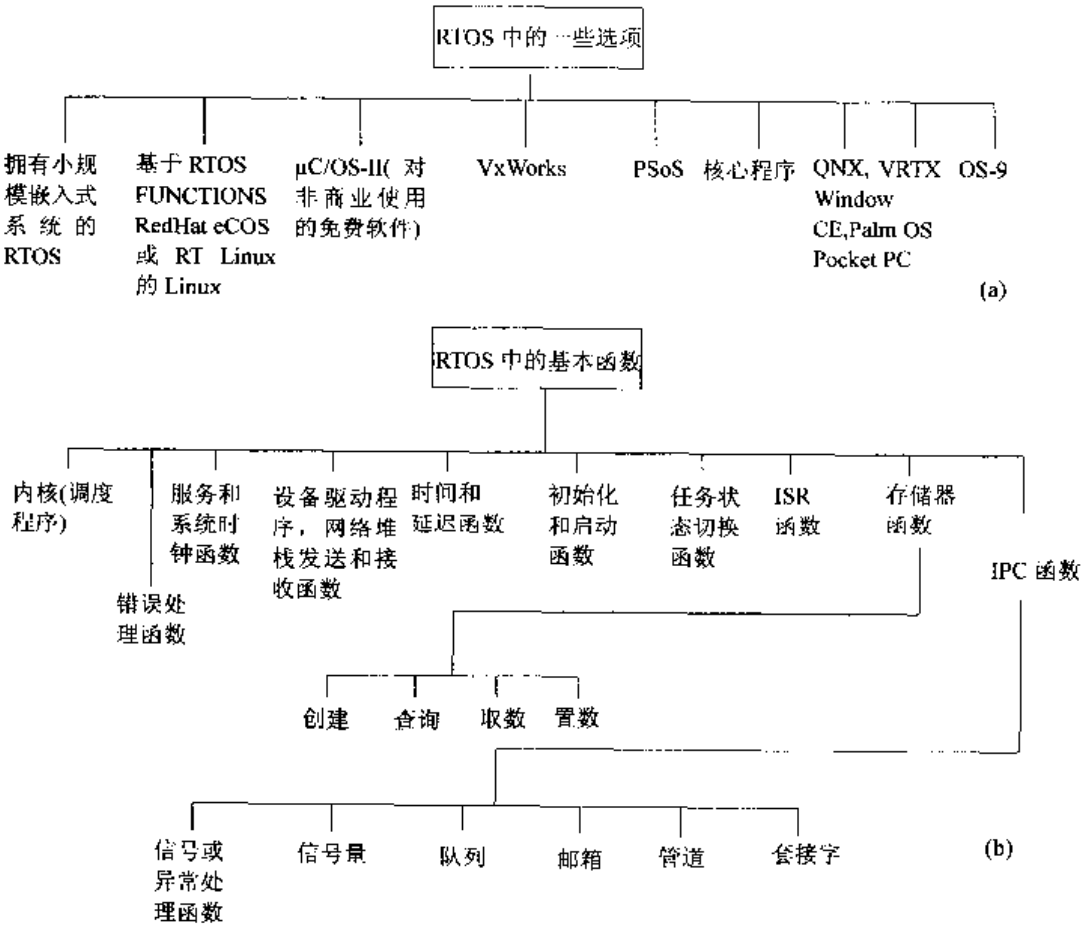


图 10-1 (a)选择 RTOS 时可用的常见选项；(b) RTOS 内核需要的基本函数

10.2 μC/OS-II

μC/OS-II 就是一个满足以上要求的 RTOS。如果作为非商业用途，RTOS μC/OS-II 是免费的。它由 Jean J. Labrosse 在 1992 年设计完成。其名称 μC/OS-II 来源于术语 Micro-Controller Operating System(微控制器操作系统)。它通常也称为 MUCOS 或者 UCOS。使用该 RTOS 还有另外一个优点，就是设计者可以很方便地将其整理成文档以及书籍(参见本书最后给出的参考文献)。

MUCOS 的代码是用 C 语言编写的，它集成了几个特定的 CPU 模块。其代码涉及许多在嵌入式系统的设计中被广泛使用的处理器。

代码中有两种类型的源文件。这两种文件的主头文件都包含“#include”预处理命令。本文使用的主头文件就是“includes.h”(参见 5.2.1 节)。

(1) 处理器相关的源文件：两个主要的头文件是：(i)os_cpu.h，处理器定义头文件。(ii)os_cfg.h，内核生成配置文件。另外，两个 C 文件用作 ISR(中断服务例程)和 RTOS 定时器，具体是 os_tick.c 和处理器的 C 代码 os_cpu_c.c。任务切换函数的汇编代码在 os_cpu_a.s12 中(68HC12 微控制器)。

(2) 处理器无关的源文件：两个文件，MUCOS 头文件(包含在主头文件中)和 C 文件，分别是 ucos_ii.h 和 ucos_ii.c。用于 RTOS 核心、定时器和任务的文件是 os_core.c、os_time.c 和 os_task.c。存储器分区、信号量、队列和邮箱的代码分别在 os_mem.c、os_sem.c、os_q.c 和 os_mbox.c 中。

i. 当函数或变量的前缀有 OS 或者 OS_(OS 后紧接下划线)时，表示它是 MUCOS 中的函数或变量。例如，OSTaskCreate()是一个创建任务的 MUCOS 函数。OS_NO_ERR 是一个 MUCOS 变量，它在函数没有出错的情况下返回 true。OS_MAX_TASKS 是一个常量，表示用户应用程序中任务的最大数目。(用户在预处理器定义中定义这一常量)。

ii. MUCOS 是可扩展的 OS。只有必需的 OS 函数才会成为应用程序代码中的一部分，这样就减少了对存储器的需求。任务服务、进程间通信等函数必须在配置文件中预先定义，这些配置文件包含在用户代码中(参见示例 10-7 的步骤 1 和步骤 2，可以清楚地理解配置设定代码)。

iii. 对于多任务处理，MUCOS 使用抢占式调度程序(参见 9.6.4 节)。

iv. MUCOS 有系统级函数。这些函数用于 RTOS 初始化和启动、RTC 节拍(中断)初始化以及 ISR 入口和出口函数(表 10-1)。对于临界段，MUCOS 包含中断禁用和启用函数，这些函数分别在进入和离开临界段时执行。

v. MUCOS 有任务服务函数(例如，任务创建、运行、挂起和恢复执行等函数)(表 10-2)。

vi. MUCOS 有任务延迟函数(表 10-3)。

vii. MUCOS 有在存储器中创建和划分块的存储器分配函数，这些函数首先得到一个块，然后进驻到这个块中，并在块中调试的时候进行查询(表 10-4)。

viii. MUCOS 有进程间通信(IPC)函数。这些函数分别在表 10-5、表 10-6 和表 10-7 中。MUCOS 的 IPC 使用信号量、队列和邮箱(参见 8.3 节)。

ix. MUCOS 有信号量函数，这些函数可以像事件标志、资源获取键或者计数信号量那样使用(见 8.3.2 节)。表 10-5 列出了这些函数。

x. MUCOS 有邮箱函数。MUCOS 中每个邮箱都有一个消息指针(参见 8.3.4 节)。由于 MUCOS 只发送指针到邮箱中，所以发送的消息数量可以是任意的。表 10-6 列出了这些函数。

xi. MUCOS 有队列函数。MUCOS 允许每个队列有一个消息指针数组，消息在队列中采用 FIFO(先进先出)方法进行检索(参见 8.3.3 节)。由于 MUCOS 只将消息地址发送到队列中，所以，队列元素中的消息数量可以是任意的。表 10-7 列出了这些函数。

接下来的 7 个小节、10.2.1~10.2.7 将分别介绍了以上提到的 MUCOS 函数。对于 7 个表中的每个函数，表 10-1~10-7 都给出了 MUCOS 函数返回值的详细信息，以及使用值或者引用传递给 MUCOS 函数的参数(变元)的详细信息。

10.2.1 RTOS 系统级函数

回顾一下示例 5-3 中的 rtos_run( )函数，我们首先在 MUCOS 中调用初始化函数，然后在调用任务创建函数创建任务之后，调用启动函数。这些函数分别是 OSInit 和 OSStart。

回顾图 9-7，我们首先初始化系统定时器 RTC 节拍(和中断)。MUCOS RTOS 带有在进入 ISR 和退出 ISR 时应该被执行的系统函数。回顾 9.6.5 节，MUCOS RTOS 带有在进入任务的临界段和退出临界段时应该被执行的系统函数(参见 8.2 节)。表 10-1 列出了这些系统级函数。

表 10-1 RTOS 的任务初始化、启动和 ISR 函数

函 数 原 型	何时调用 OS 函数
void OSInit(void)	开始阶段，OSStart()之前调用该函数
void OSStart(void)	OSInit()和任务创建函数之后调用该函数
void OSTickInit(void)	在第一个任务函数执行一次来初始化系统定时器节拍(RTC 中断)的时候调用该函数
void OSIntExit(void)	在 OSIntExit 从 ISR 中返回之前必须调用的 ISR 代码刚刚开始执行之后，调用该函数。#(NTER 和 EXIT 函数形成一对)
void OSIntExit(void)	OSIntEnter()在 ISR 代码刚刚开始执行之后被调用，OSIntExit 在刚刚从 ISR 中返回之前被调用，在这两次调用之后，该函数才被调用。(ENTER 和 EXIT 函数形成一对)*
OS_ENTER_CRITICAL	禁止中断的宏(参见 8.2.2(vi))
OS_EXIT_CRITICAL	启用中断的宏(ENTER 和 EXIT 函数在临界段形成一对)(参见 8.2.2(vi))

注意：

1. 该表中的函数不传递参数，函数返回值为空。

2. #有一个全局变量 OSIntNesting。它在进入调用之后加 1。(虽然可以直接增加该变量的值，但是我们不应该这样做，而应该让它在进入 ISR 时自动加 1)。

3. *全局变量 OSIntNesting 在退出调用的时候减 1。(虽然可以直接减少该变量的值，但是我们不应该这样做，而应该让它在进入 ISR 时自动减 1)。

(1) 在 RTOS 函数开始调用之前初始化操作系统

函数 void OSInit(void)用于初始化系统。该函数在调用任何 OS 内核函数之前被强制调用。该函数不返回参数。

下面是该函数的一个使用范例：

示例 10-1

```
1. /* Start executing the codes */void main (void) {
2. /* Initiate MUCOS RTOS to let us use the OS kernel functions */OSInit ( );
3. /* Create (Define Identity, stack size and other TCB parameters for the tasks
   using RTOS Functions in Table 10.1 */
   .
   .
4. /* Create semaphore, queue and mailboxes, etc. */
   .
   .
   }.
```

(2) 开始使用 RTOS 多任务函数并运行任务

函数 void OSStart(void)用于启动已初始化的系统和已创建的任务。该函数在进行多任务 OS 内核操作时被强制调用。

该函数不返回参数。

下面是该函数的一个使用范例：

示例 10-2

```
1. /* Start executing the codes from Main*/void main (void) {
2. OSInit ( );
3. /* Create tasks and inter-process communication variables by defining their
   identity, stack size and other TCB parameters. */
   .
   .
4. /* Start MUCOS RTOS to let us use RTOS control and run the created tasks and
   inter-process communication. */
OSStart ( );
   /* An infinite while-loop follows in each task. So there is no return from
   the RTOS. */
   }/ * End of the Main function.
```

(3) 启动 RTOS 系统时钟

函数 void OSTickInit(void)用作系统节拍和中断的定时初始化，每个 OS_TICK_PER_SEC 在配置 MUCOS 期间被预先定义。该函数在将要调用定时器函数时，强制用作多任务 OS 内核操作。

该函数返回空值，不传递参数。

下面示例 10-7 的步骤 2~8 给出了该函数的使用方法。

(4) 在 ISR 启动时向 RTOS 内核发送消息，以获得控制权

OSIntEnter(void)函数在启动 ISR 执行时调用。调用时，该函数向 RTOS 内核发送消息，以获得控制权。该函数被强制调用，使多任务 OS 内核能控制多个有嵌套关系的 ISR，在多个具

有不同优先级的中断同时发生时，会出现 ISR 的嵌套。

该函数不返回参数。

下面时该函数的一个使用范例。

示例 10-3

```
1. /* Start executing the codes of an ISR*/
   ISR_A ( ) {
2. /* sending message to RTOS kernel for taking control of ISR_N from nested
   ISRs loop. Increment OSIntNesting, a global variable */OSIntEnter ( );
3. /* Codes for servicing of the ISR by calling a task. */
   .
   .
```

(5) 从 ISR 中返回时，向 RTOS 内核发送消息以撤销控制权

函数 void OSIntExit(void)在刚刚从 ISR 的执行过程中返回之前被调用。该函数调用时向 RTOS 内核发送消息，从嵌套循环中撤销控制权。它被强制调用，使 OS 内核从各个 ISR 的嵌套循环中撤销 ISR。

该函数不返回参数。

下面是该函数的一个使用范例：

示例 10-4

```
1. to 3. As in Example 10.3
   .
   .
4. /* Sending message to RTOS kernel for quitting the control of ISR_A from the
   nested loop. Decrement OSIntNesting, a global variable */
   OSIntExit ( );
   }/ * End of the ISR function.
```

(6) 在启动临界段时，向 RTOS 内核发送消息以获得控制权

宏函数 OS_ENTER_CRITICAL 在 ISR 开始执行时被调用。它在调用时向 RTOS 内核发送消息以禁止中断。它被强制调用，使 OS 内核注意到它，从而禁止系统中断。

该函数不返回参数。

下面使该函数的一个使用范例：

示例 10-5

```
1. /* Start executing the codes of an ISR*/task_A ( ) {
2. /* Codes for servicing of the task. */
   .
   .
3. /* Sending a message to RTOS kernel and disabling the interrupts.*/
   OS_ENTER_CRITICAL;
4. /* Run critical section codes as follows. */
   .
   .
5. /* Codes for Exiting the service*/
   }
```

(7) 在从临界段返回时，向 RTOS 内核发送消息以撤销控制权

宏函数 OS_EXIT_CRITICAL 在刚刚从 ISR 的执行过程中返回之前被调用。该函数调用时向 RTOS 内核发送消息，从临界段中撤销控制权。它被强制调用，使 OS 内核退出临界段，并启用系统中断。

该函数不返回参数。

下面是该函数的一个使用范例：

示例 10-6

```
1. to 4. As in Example 10.5
   .
   .
5. /* Sending a message to RTOS kernel for quitting the control of critical section
    and enabling the interrupts. */
    OS_EXIT_CRITICAL;
    }/ * End of the ISR function.
```

10.2.2 任务服务函数及其使用范例

表 10-2 给出了任务中会用到的 MUCOS 服务函数。服务函数是指创建、挂起和恢复任务的函数，以及时间设定和时间检索(获取)函数。这些函数的调用要结合预处理器命令和变量赋值，以及原型指派，如示例 10-7 中的步骤 1 和步骤 2。这两个步骤的代码保存在配置文件中，并且这些配置文件在编译前被包含到源代码内。这几个步骤在函数被调用之前，对 MUCOS 进行配置(示例 10-7 将给出这几个步骤)。

在下面的示例中，我们会看到在每个任务函数中都有一个无限循环，这是优先任务调度中一种特有的任务编码方式。CPU 如何控制从无限循环中返回到 MUCOS 中继续执行呢？换句话说，就是上下文切换在 OS 中是如何实现的？然后 OS 如何将任务切换到较高优先级的任务执行？只要有以下一种情况发生，CPU 控制权就会返回到 MUCOS(或者返回到任何一个其他的任务调度程序)。

(1) 发生任何包含定时器节拍中断的中断事件。参见示例 10-7 的步骤 8(步骤 2 中每 1 秒钟进行一次时间设定)。

(2) 通过调用 OSTaskSuspend，挂起目前正在运行的任务，如示例 10-8 中的步骤 12。

(3) 调用任何 OS 函数，比如说表 10-3 中的时间延迟函数，或者表 10-5 中的信号挂起函数。然后调度程序切换上下文，抢占控制权，并通过激活任务切换，将控制权传递给具有最高优先级的任务。

(4) 由于抢占的较高优先级任务中也存在一个无限循环，CPU 的控制又如何从这个抢占控制权的任务中返回呢？显然必须通过一种适当的编码进行返回控制。使用范例参见示例 10-8 中步骤 12 的代码。在这里，FistTask 的优先级是 4(用户任务可用的最高优先级)。在执行步骤 12 时，该任务将自己从循环中挂起。

表 10-2 任务的服务和系统时钟函数

函 数 原 型	返回的参数	传递的参数	何时调用 OS 函数
unsigned byte OSTaskCreate(void(*task)(void *taskPointer),void*pmdata, OS_STK *task StackPointer, unsigned byte taskPriority)	RA	PA	在运行任务之前必须调用该函数
unsigned byte OSTaskSuspend(unsigned byte taskPriority)	RB	PB	需要阻塞任务执行时调用该函数
unsignedbyte OSTaskResume(unsigned byte taskPriority)	RC	PC	恢复执行被阻塞的任务时调用该函数
void OSTimeSet (unsigned int counts)	无	PD	当通过 counts 值来设定系统时间时调用该函数
unsigned int OSTimeGet(void)	RE	无	当读取系统时间时，获得目前的 counts 值

unsigned int 是指 32 位无符号整数。第 2 列和第 3 列中使用的缩写在本文中有解释。

(1) 创建任务

函数 OSTaskCreate (void (*task)(void*taskPointer)，void*pmdata，OS_STK*task StackPointer, unsigned byte taskPriority)解释如下。

抢占式调度程序抢占调度具有较高优先级的任务。所以，每个用户任务都会标上一个优先级，这个优先级必须设定在 4 和 OS_MAX_TASKS + 3 之间(或者 4 和 OS_LOWEST_PRIORITY － 4 之间)。如果用户任务的最大数目 OS_MAX_TASKS 是 8，那么其优先级必须设定在 4 和 11 之间。对于优先级在 4~11 之间的 8 个用户任务，OS_LOWEST_PRIO 必须设定为 15，这是因为 MUCOS 会将 priority = 14 赋给优先级最低的任务。然后，priority = 0/1/2/3/12/14 的优先级将被 MUCOS 内部使用。OS_LOWEST_PRIO 和 OS_MAX_TASKS 是预处理器代码中用户定义的常量，这些预处理器代码在为用户应用程序配置 MUCOS 时需要用到。如果用户实际上只创建了 4 个任务，那么就应该避免定义 20 个任务，因为 OS_MAX_TASKS 越大，意味着分配给任务的存储空间就越大，这是没有必要的资源浪费。

传递 PA 的任务参数：

(i) *taskPointer 是指向正在创建的任务代码的指针。

(ii) *pmdata 是指向传递给任务的可选消息数据引用的指针。如果引用为空，就将其指定为 NULL。

(iii) *TaskStackPointer 是正在创建的任务中指向任务代码的指针。

(iv) TaskPriority 是任务优先级，当 OS_MAX_TASKS=10 时，它必须在 4~13 之间取值。

返回 RA：任何任务的最低优先级 OS_PRIO_LOWEST 都是 16。对于应用程序的编程，指定的优先级必须在 4~13 之间。函数 OSTaskCreate( )的返回值包含 3 种情况：(i)如果任务创建成功，则返回 OS_NO_ERR。(ii)如果传递的优先级的值已经存在，则返回 OS_RPIO_EXIST。(iii)如果传递的优先级的值比 OS_PRIO_LOWEST 更大，则返回 OS_PRIO_INVALID。(iv)如果

没有足够的存储区块用于任务控制，则返回 OS_NO_MORE_TCB。

这里给出了一个函数的使用范例，是为一个处于连接状态的任务 Task1Connect 创建任务进程。OSTaskCreate(Task1_Connect, void (*) 0, (void*) *Task1_ConnectStack[100], 6)。

以下是作为参数传递的任务参数：

(i) Task1_Connect，任务创建时指向 Task1_Connect 的代码指针。

(ii) 指向传递给任务的可选消息数据引用的指针是 NULL。

(iii) *Task1_ConnectStack 是指向 Task1_Connect 堆栈的指针，系统为该指针分配 100 个地址的存储器空间。

(iv) TaskPriority 是任务优先级，一般指定为 6，这个值是除了可以分配的两个最高的优先级值以外，可以指定的最高任务优先级值。

范例中的这个函数可能产生两个错误参数。如果任务 Task1Connect 创建成功，则 OS_NO_ERR 为 true。如果优先级为 4 的任务已经创建，并仍在运行，则产生错误参数 OS_PRIO_EXIST 。如果传递的优先级参数的值比 OS_LOWEST_PRIO 更大，则产生错误参数 PS_PRIO_INVALID。如果 Task1Connect 缺少可用的任务控制块，则 OS_NO_MORE_TCB 为 false(TCB 的定义在 8.1.1 节中给出)。

示例 10-7

```
1. /* Preprocessor MUCOS configuring commands to define OS tasks service and
timing functions as enabled and their constants*/
#define OS_MAX_TASKS 8 /* Let maximum number of tasks in user application be
8. */
#define OS_LOWEST_PRIO 15 /* Let lowest priority task in the OS be 15. */
#define OS_TASK_CREATE_EN 1/* Enable inclusion of OSTaskCreate ( ) function */
#define OS_TASK_DEL_EN 1/* Enable inclusion of OSTaskDel ( ) function */
#define OS_TASK_SUSPEND_EN 1/* Enable inclusion of OSTaskSuspend ( ) function
*/
#define OS_TASK_RESUME_EN 1/* Enable inclusion of OSTaskResume ( ) function */
.
.
/* End of preprocessor commands */
2. /* Specify all user prototype of the task functions to be scheduled by MUCOS
*/
/ * Remember: Static means permanent memory allocation */
static void FirstTask (void *taskPointer);
static void Task1_Connect (void *taskpointer);
static OS_STK FirstTaskStack [FirstTask_StackSize];
static OS_STK Task1_ConnectStack [Task1_Connect_StackSize];
/* Define public variable of the task service and timing functions */
#define OS_TASK_IDLE_STK_SIZE 100 /* Let memory allocation for an idle state
task stack size be
100*/
#define OS_TICKS_PER_SEC 1000 / * Let the number of ticks be 1000 per second.
An RTCSWT will interrupt and thus tick every 1 ms to update counts. */
#define FirstTask_Priority 4 /* Define first task in main priority */
#define FirstTask_StackSize 100 /* Define first task in main stack size */
#define Task1_Connect_Priority 6 /* Define Task1_Connect priority */
#define Task1_Connect_StackSize 100 /* Define Task1_Connect stack size */
```

```
.
.
.
3. /* The codes of the application starts from Main*/
void main (void) {
4. /* Initiate MUCOS RTOS to let us use the OS kernel functions */
OSInit ( );
5. /* Create first task that must execute once before any other. Task creates
by defining its identity as FirstTask, stack size and other TCB parameters. */
OSTaskCreate (FirstTask, void (*) 0, (void *) &FirstTaskStack [FirstTask_StackSize],
FirstTask_Priority);
/* Create other main tasks and inter-process communication variables if these
must also ex-ecute at least once after the FirstTask. */
.
.
6. /* Start MUCOS RTOS to let us use RTOS control and run the created tasks */
OSStart ( );
/* Infinite while-loop is there in each task. So there is no return from the
RTOS function OSStart
( ). */
}/ *** End of the Main function ***/
/* The codes of the application first task that creates in Main*/
7. static void FirstTask (void *taskPointer) {
8. /* Start Timer Ticks for using timer ticks later. */
OSTickInit ( ); /* Function for initiating RTCSWT that ticks at the configured
time in the MUCOS configuration preprocessor commands in Step 1 */
9. /* Create a Task as per Step 2 defined by task identity, Task1_Connect, stack
size and other TCB parameters. */
OSTaskCreate (Task1_Connect, void (*) 0, (void *) &Task1_ConnectStack
[Task1_Connect_StackSize], Task1_Connect_Priority)
10. /* Create other tasks and inter-process communication variables. */
.
.
11. while (1) /* Infinite loop of FirstTask */
.
.
12.}; /* End infinite loop */
13.} /End of FirstTask Codes. */
```

The first main task is the only task created in Step 5 of *main*. The first task calls a function for the timer initiation (step 8). This is required when the RTOS timer functions are needed in a particu-lar application. A task function *Task1_Connect* codes are created above in Step 9. All other tasks that are created are the private tasks using OSTaskCreate function within that first main task function].

```
14. /* The codes for the Task1_Connect*/
static void Task1_Connect (*taskPointer) {15. /* Initial assignments of the
variables and pre-infinite loop statements that execute once only*/
.
.
.
16. /* Start an infinite while-loop. */
while (1) {
```

```
17. /* Codes for Task1_Connect*/
.
.
18.}; /* End of while loop*/
19.}/ * End of the Task1_Connect function */
```

(2) 挂起(阻塞)任务[C]

函数 unsigned byte OSTaskSuspend(unsigned byte taskPriority)

传递 PB 的任务参数： taskPriority 指将挂起的任务，或者正在被挂起的任务的优先级。它的值在 4~13 之间。

返回 RB：如果阻塞任务成功，则函数 OSTaskSuspend ( )返回错误参数 OS_NO_ERR。如果任务优先级的值超过 16，即常量 OS_PRIO_LOWEST，则返回 OS_PRIO_INVALID。如果已经传递的优先级的值不存在，则返回 OS_TASK_ SUSPEND_PRIO。如果试图挂起一个非法空闲任务，则返回 OS_TASK_SUSPEND_IDLE。

以下是阻塞任务的一个范例，被阻塞任务是 Task1_Connect，其优先级 priority = Task1_Connect_Priority。阻塞函数：OSTaskSuspend(Task1_Connect_Priority)。作为参数传递的任务参数是 4。我们可以回顾一下，先前在示例 10-7 中给 Task1_Connect_Proirity 指定的值就是 4。该函数返回的错误参数如下：

(i) 如果阻塞任务成功，则 OS_NO_ERR 为 ture。

(ii) 由于 4 是一个有效的优先级，它没有超过 OS_PRIO_LOWEST 的值，则 OS_PRIO_INVALID 为 false。

(iii) OS_PRIO_LOWEST = 16。

(iv) 如果已经传递的优先级的值不存在，则 OS_TASK_SUSPEND_PRIO 为 false。

(v) 如果试图挂起一个非空闲的任务，则 OS_TASK_SUSPEND_IDLE 为 false。

示例 10-8

```
1 to 11. /* Steps Codes as in Example 10.7 */
12. /* Suspend FirstTask, as it was for initiating the timer ticks (interrupts),
creating the user application tasks, and was to be run only once */
OSTaskSuspend (FirstTask_Priority); /*Suspend First Task and control of the RTOS
passes to other
tasks waiting execution*/
} /* End of while loop */
) /* End of FirstTask Codes */
```

(3) 恢复执行(取消阻塞)任务

函数 unsigned byte OSTaskResume(unsigned byte taskPriority) 恢复执行挂起的任务。

传递 PC 的任务参数：taskPriority 指将恢复执行的任务的优先级。它的值在 4~13 之间。

返回 RC：如果取消阻塞任务成功，则函数 OSTaskResume( )返回 OS_NO_ERR。如果传递的优先级值大于 16，即常量 OS_PRIO_LOWEST，则返回 OS_PRIO_INVALID。如果传递的优先级值不存在，则返回 OS_TASK_RESUME_PRIO。如果试图恢复执行没有挂起(阻塞)的任务，则返回 OS_TASK_NOT_SUSPENDED。

以下给出了取消阻塞任务的使用范例，任务是 Task1_Connect，其优先级 priority = Task1_Connect_Priority 。使用的函数为 OSTaskResume(Task1_Connect_Priority) 。由于

Task1_Connect_Priority = 4，因而作为参数传递的任务参数是 4。以下是恢复执行函数返回的错误参数。

(i) 如果取消阻塞任务成功，并且优先级为 4 的达到了运行状态，则 OS_NO_ERR 为 true。

(ii) 由于 4 是一个有效的优先级值，而且不大于 OS_PRIO_LOWEST，所以 OS_PRIO_INVALID = false。

(iii) OS_PRIO_LOWEST = 16。

(iv) 如果已经传递的优先级值不存在，则 OS_TASK_RESUME_PRIO 为 false。

(v) 如果试图恢复执行没有被挂起的任务，则 OS_TASK_NOT_SUSPENDED 为 false。

示例 10-9

```
1. to 12. /* Steps as per Example 10.8 codes for Task1_Connect Function. */
.
.
13. /* Other codes, for example, time delay for 1 second */
.
.
14. /* Resume Task1_Connect. Control. */
OSTaskResume (Task1_Connect_Priority);
15.}; /* End of while loop*/
16.}/ * End of the Task1_Connect function.
```

(4) 设定系统时钟

函数 void OSTimeSet(unsigned int count) 返回空值。传递参数 PD 在下面作为参数给出。

PD：该参数传递 32 位的整数 count(为当前时间设定节拍数目，该数在每个系统 RTC 之后减 1)。

下面给出了函数 OSTimeSet(to preset the time)的一个使用范例。函数 OSTimeSet(0)的当前参数 count 为 0，该函数执行结果就是设置 presetTime = 0。注意：由于一些其他依赖于定时器的函数可能会出错，建议在任务函数中不使用 OSTimeSet 函数。继续使用 OS 定时器作为自由运行的计数器。因为某一时刻的时间可以使用 get 函数(示例 10-11)读取得到，并且其他任意时刻的时间可以通过对该时刻的时间加上一个值来定义。示例 10-10 在 FirstTask 中使用了 set 函数。

示例 10-10

```
void FirstTask (*taskPointer) {
1. to 8. /* The codes up to OSTickInit ( ) in Example 8.7*/
9. /* Set the timer number of ticks to 0 */
presetTime =0;
OSTimeSet (presetTime);
10. /* Other codes of FirstTask */
.
.
}/ * End of the FirstTask function.
```

(5) 获取系统时钟

函数 unsigned int OSTimeGet(void)以无符号整数类型返回当前节拍数值。作为参数传递的参数为空。

RE：返回32位整数，即当前系统RTC中节拍的数值。

示例10-11

```
1. /* The codes as for a task function */
.
.
currentTime = OSTimeGet ( );
2. /* Other codes of the task after determining current time */
.
.
}/ * End of the Task function.
```

10.2.3 时间延迟函数

表10-3中的每一行给出了任务中用到的MUCOS时间延迟函数。

表10-3 任务的时间延迟函数

函数原型	返回的参数	传递的参数	何时调用OS函数
void OSTimeDly (unsigned short delay Count)	无	PF	当某个任务需要被延迟，计数输入等于delayCount-1时调用该函数。(RTCSWT::delay()在任务代码中执行该延迟函数)#
unsigned byte OSTimeDlyResume (unsigned byte_task Priority)	RG	PG	当某个优先级为taskPriority的任务在预设延迟之前恢复执行时调用该函数。这里的预设延迟通过由delayCount或者(hr、mn和ms)定义的数值来实现，但是现在正处于阻塞状态。
void OSTimeDly HMSM(unsigned byte hr, unsigned byte mn, unsigned byte sec, unsigned short ms)	RH	PH	当需要延迟并阻塞任务hr小时、mn分钟、sec秒或者ms毫秒*时调用该函数

表中第2列和第3列的缩写在文中有解释。

注意：1.#任务不能使用函数OSTimeDly来实现大于65535个节拍计数输入的延迟。2.*如果delay(in hr, mn, sec和ms)被设为大于65535个系统节拍计数输入，则在后面任务不能通过OSTimeDlyResume( )恢复执行。

(1) 通过定义由节拍数确定的时间延迟进行延迟操作

函数void OSTimeDly(unsigned short delayCount)将任务延迟delayCount-1个系统RTC的节拍。该函数不返回参数。

传递PF的任务参数：16位整数delayCount，延迟任务直到系统时钟计数输入(节拍)等于(delayCount - 1)+count为止，这里的count是指在系统RTC中节拍的当前数值。如果需要更多延迟，就通过在任务中多次调用该函数来实现。

以下是该函数的一个使用范例，如果系统RTC时钟每1毫秒嘀哒1次，OSTimeDly(1000)

将任务至少延迟 1000 毫秒。

示例 10-12

```
1. to 18. /* Steps as per Example 8.7 codes for Task1_Connect Function. */
.
.
19. /* Time delay for 1 second = period of 1000 system ticks if system tick is
set at every 1 ms.*/
OSTimeDly (1000);
20. /* Resume Task1_Connect by a function defined in next subsection and execute
other codes within the loop. */
.
.
21. }; /* End of while loop*/
22. }/ * End of the Task1_Connect function.
```

(2) 恢复执行被延迟的任务

函数 unsigned byte OSTimeDlyResume(unsigned byte_taskPriority)恢复执行先前被延迟的任务，这里不区分延迟参数是按照 delayCount 节拍还是按照小时、分钟和秒。注意：如果定义的延迟大于 65535 个 RTC 节拍，那么 OSTimeDlyResume 将不会恢复执行被延迟的任务。

返回 RG：返回如下错误参数

(i) 如果在任务延迟成功之后恢复执行，则 OS_NO_ERR 为 true。

(ii) 如果任务在之前没有被创建，则 OS_TASK_NOT_EXIST 为 true。

(iii) 如果任务没有被延迟，则 OS_TIME_NOT_DLY 为 true。

(iv) 如果传递的任务优先级参数 taskPriority 大于 OS_PRIO_LOWEST(16)，则 OS_PRIO_INVALID。

传递 PG 的任务参数：taskPriority 是任务的优先级。确切地说，在恢复执行之前延迟了任务。

以下是 OSTimeDlyResume(Task_CharCheckPriority)的使用范例。该函数恢复执行被延迟的任务，OS 通过优先级 Task_CharCheckPriority 来识别该任务。

示例 10-13

```
1. to 19. /* Steps as per Example 10.12 codes for Task1_Connect Function. */
.
.
20. /* Time delay for 1 second = period of 1000 system ticks if system tick is
set at every 1 ms.*/ OSTimeDly (1000);
21. /* Other codes */
.
.
22. /* Resume Task1_Connect Control and execute other codes within the loop.
*/
OSTimeDlyResume (Task1_Connect_Priority);
.
.
23. }; /* End of while loop*/
24. }/ * End of the Task1_Connect function.
```

(3) 通过以小时、分钟、秒和毫秒作为单位定义的时间延迟来实现任务延迟

函数 void OSTimeDlyHMSM(unsigned short hr, unsigned short mn, unsigned short sec, unsigned short mils)可以延迟某个任务最多 65535 个节拍，延迟时间为 hr 小时(0~55)，mn 分钟(0~59)，sec 秒(0~59)，mils 毫秒(0~999)。“mils”要求是 RTC 节拍的整数倍。定义该函数的任务是任务延迟。

返回 RH：函数 OSTimedlyHMSM()返回给 OS 一个错误代码，如下：

(i) 如果四个参数都有效，并且是在延迟成功后恢复任务执行，则返回 OS_NO_ERR。

(ii) 如果四个参数分别大于 55，59，59 和 999，则分别返回 OS_TIME_INVALID_HOURS，OS_TIME_INVALID_MINUTES,OS_TIME_INVALID_SECONDS 和 OS_TIME_INVALID_MILLI。

(iii) 如果传递的所有参数都为 0，则返回 OS_TIME_ZERO_DLY。

传递 PH 的任务参数:hr、mn、sec 和 ms 是以小时、分钟、秒和毫秒来计算的任务延迟时间，即任务在恢复执行前的延迟。

OSTimeDlyHMSM(0,0,0,1000)函数的使用范例参见示例 10-12 中步骤 8 的任务代码。该函数使任务延迟 1000 毫秒。如果每 1 毫秒产生一个系统 RTC 节拍，那么这样一次延时至少是 1000 毫秒(如果节拍数定义为 9000000 毫秒，那么 OSTimeDlyResume 就不可能在要求恢复执行时，进行恢复任务的操作。这是因为节拍数必须小于 65535)。

10.2.4 函数相关的存储器分配

任务中用到的 MUCOS 存储器函数如表 10-4 所示。

表 10-4 用于查询、创建、获取和放置的 RTOS 存储函数

函 数 原 型	返回和传递的参数	何时调用 OS 函数
OSMem*OSMemCreate(void*memAddr, MEMTYPEnumBlocks,MEMTYPEblockSize, unsigned byte *memErr)	RI 和 PI	创建和初始化存储分区时调用该函数(参见示例 2-12)
Void*OSMemGet(OS_MEM*memCBPointer, unsigned byte *memErr)	RJ 和 PJ	获取分配给存储块的存储控制块的指针时调用该函数，如果没有块，则返回 NULL
Unsigned byte OSMemQuery(OS_MEM*mem CBPointer,OS_MEM_DATA * memData)	RK 和 PK	获取存储控制块和 OS_MemData 数据结构的指针时调用该函数
Unsigned byte OSMemPut (OS_MEM * mem CBPointer, void *memBlock)	RL 和 PL	从存储控制块指针中返回指向存储分区中的存储块的指针时调用该函数(参见示例 2-12)

(1) 按照存储器地址开始创建存储块

函数 OSMem *OSMemCreate(void *memAddr, MEMTYPE numBlocks, MEMTYPE blockSize, unsigned byte *memErr)是一个 OS 函数，该函数从存储块中的某个地址开始将存储区域进行划分。将存储区域进行划分有助于 OS 进行资源分配。

返回 RI：函数*OSMemCreate()为创建的存储器分区返回指向控制块的指针。如果没有已经创建的分区，则该函数返回 NULL 指针。

传递 PI 的任务参数：MEMTYPE 是根据存储器定义的数据类型，一般为 16 位或者 32 位的 CPU 存储器地址。例如，在 68HC11 和 8051 中是 16 位。(i)*memAddr 是指向存储器的开始地址的指针。(ii)numBlock 是存储器必须被划分的个数块(必须是 2 块或者更多)。(iii)blockSize 是划分完之后每个块的大小。(iv)*memErr 是指向保存错误代码的地址指针。在*memErr 地址上，下面这些全局错误变量从 true 变为 fasle：如果创建成功，OS_NO_ERR = true，如果至少两个块没有作为参数被传递，则 OS_MEM_INVALID_BLKS = true。(v)如果划分的存储块不可用，则 OS_MEM_INVALID_PART=true。(vi)如果块的大小比指针变量还小，则 OS_MEM_INVALID_SIZE = true。

示例 10-14 给出了 4 个存储块的创建过程，每个块的大小为 1k。存储器开始地址是 0x8000。

示例 10-14

```
1. /* Definition in Pre-Processor for a 16-bit unsigned number, MEMTYPE to define
number of blocks which can be between 0 and 65535 and to define the number of
bytes that store at a block. Maximum number of bytes at a block can be 65535.
*/
typedef unsigned short MEMTYPE;
3. /* Codes for main function or for a task function */
.
.
4. /* Codes for creating the blocks of memory*/
memAddr = 0x8000;
numBlocks =4;
blockSize = 1024; /* Each block is of 1KB memory */
*OSMemCreate (*memAddr, numBlocks, blockSize, *memErr);
.
.
5. /* Other Codes for the function. */
.
.
} /* End of the function */
```

(2) 按照存储器地址获取存储块

函数 void *OSMemGet(OS_MEM *memCBPointer, unsigned byte *memErr)从先前创建的存储器分区中检索某个存储块。

返回 RJ：函数 OSMemGet()为存储器分区返回指向存储控制块的指针。如果没有块存在，则返回 NULL。

传递 PJ 的任务参数：(i)作为参数传递指向存储器分区控制块的指针。(ii)函数 OSMemGet()传递错误代码指针*memErro，从而返回如下参数值：(i)如果从存储块中成功返回到存储器分区，则返回 OS_NO_ERR。(ii)如果由于存储器分区满，导致存储块不能被放到存储器分区中，

则返回 OS_MEM_FULL。

示例 10-15 介绍了如何获得指向先前创建的存储块的指针。

示例 10-15

```
1. to 5. /* Codes as per Example 10.14 */
6. /* Codes for retrieving the pointer to memory block in a partition created
by step 5 in example 10.14 */
memPointer = 0xA000;
memErr = OS_MEM_NO_FREE_BLKS;
*OSMemGet (*memPointer, *memErr);
.
.
5. /* Other Codes for the function. */
.
.
} /* End of the function */
```

(3) 查询存储块

函数 unsigned byte OSMemQuery(OS_MEM *memCBPointer, OS_MEMDATA *memData)用于查询并返回错误代码和指针给存储器分区。如果在*OS_MEMDATA 存在存储器地址*memPointer，则 OS_NO_ERROR 为 1，否则返回 0。

返回 RK：函数 OSMemQuery()返回错误代码，该代码是一个无符号字节。如果查询成功，则错误代码 OS_NO_ERR 为 1，否则为 0。

传递 PK 的任务参数：函数 OSMemQuery () 传递(a)先前创建的存储块的指针 memPointer 和(b)数据结构的指针 OS_MEM_DATA。由于指针是作为引用传递，因而返回的关于存储器分区的信息中有存储控制块指针。

(4) 将存储块放置到存储器分区中

函数 unsigned byte OSMemPut(OS_MEM *memCBPointer, void *memBlock)返回*memBlock 指向的存储块，该存储控制块的指针为*memCBPointer。

返回 RL：函数 OSMemPut()返回错误代码，情况如下：(i)如果存储块返回到存储器分区，则返回 OS_NO_ERR。(ii)如果由于存储器分区满，导致存储块不能被放置到存储器分区中，则返回 OS_MEM_FULL。

传递 PL 的任务指针：(i)函数 OSMemPut( )将存储控制块的指针*memCBPointer 传递给存储器分区，即放置存储块的地方。(ii)存储块的指针*memBlock 被放到存储器分区中。

10.2.5 信号量相关函数

表 10-5 中的每一行给出了用于任务的 MUCOS 信号量函数。所以，当该 OS 创建的信号量作为资源获取键被使用时，信号量的值应该从 1 开始，1 是指资源可用，而 0 是指资源不可用。MUCOS 为使用相同的信号量函数提供一个事件信号标志，或者计数信号量(参见 8.3.2 节)。在此情况下，该值应该从 0 开始。

表 10-5 用于任务间通信的 RTOS 信号量函数

函 数 原 型	返回和传递的参数	何时调用 OS 函数
OS_Event OSSemCreate (unsigned short semVal)	RM 和 PM	创建和初始化信号量时调用该函数
void OSSemPend(OS_Event *eventPointer, unsigned short timeout, unsigned byte *SemErrPointer)	RN 和 PN	检查信号量是否处于待定状态(0 或者>0)时调用该函数。如果处于待定状态(= 0)，就挂起任务直到>0(释放)。如果>0，则将信号量的值加 1 并运行等待代码。挂起待定任务，并继续释放信号量。在特定数量的定时器节拍(RTC 中断)之后按 timeOut 释放。信号量值加 1，可以使其再次进入待定状态，等待其他任务使用
unsigned short OSSemAccept (OS_EVENT *eventPointer)	RO 和 PO	检查信号量的值是否为 0 时调用该函数，如果是，则检索到并将其减 1。当在没有必要挂起任务的情况下使用时，如果该值已经不是 0，就将其减到 0
unsigned byte OSSemPost (OS_EVENT *eventPointer)	RP 和 PP	如果 SemVal 是 0 或更大，则将其加 1。加 1 使得信号量再次脱离等待任务的待定状态。如果任务处于阻塞状态，正在等待 SemVal 信号量获取大于 0 的值，然后使这些任务可以随时在内核的引导下开始运行。内核获得正要运行和准备就绪的任务的优先级，然后首先运行优先级最高的一个任务
Unsigned byte OSSemQuery (OS_EVENT *eventPointer, OS_SEM_DATA *SemData)	RQ 和 PQ	该函数用于获取信号量信息

第 1 列的 unsigned byte 是指无符号整数。第 2 列请参考文中相应的解释段落。

(1) 为 IPC 创建的信号量

函数 OS_Event OSSemCreate(unsigned short semVal)用于创建 OS 的 ECB(Event for an IPC Control Block，IPC 控制块事件)，创建时使用参数 semVal 返回指向 ECB 的指针。信号量创建和初始化时使用的参数值为 semVal。

返回 RM：函数 OSSemCreate()为分配给信号量的 ECB 返回指针*eventPointer。如果没有 ECB 可用，则返回 NULL。

传递 PM 的任务参数：传递的 semVal 在 0~65535 之间。作为 IPC 的事件标志，semFlag 必须传递 0，作为资源获取键，SemKey 则必须传递 1。作为 IPC 的计数信号量，SemCount 必须是 0 或者必须传递一个计数值。

为进一步了解 OSSemCreate 的用法，请参见示例 10-16、示例 10-17 和示例 10-18 的内容。

(2) 等待用于信号量释放的 IPC

函数 void OSSemPend(OS_Event *eventPointer, unsigned short timeout, unsigned byte *SemErrPointer)使任务等待，直到发生某个信号量释放的事件：SemFlag、SemKey 或者 SemCount。后者是发生在*eventPointer 所指向的 ECB 内。SemFlag、SemKey 或者 SemCount 变到大于 0，就是标志处于等待状态的任务得到释放的事件。这时任务就绪，可以运行(如果没有其他优先级更高的任务等待运行，则运行这些任务)。任务在时间超过预定义的超时时间 timeOut 后，也会就绪等待运行。SemFlag、SemKey 或者 SemCount 减 1，如果它们变成 0，就让信号量再次转为待定状态，这时其他任务必须等待它的释放。

返回 RN：当信号量处于待定状态时，函数 OSSemPend()挂起任务直到信号量大于 0(释放)，并在取消阻塞(恢复执行)该任务后将 semVal 减 1。该函数返回如下参数值：(i)如果信号量查找成功(SemVal>0)，则返回 OS_NO_ERR。(ii)如果信号量在预先定义的超时限制时间内没有释放(没有变成大于 0)则返回 OS_TIMEOUT。(iii)如果该函数被 ISR 调用，则返回 OS_ERR_PEND_ISR。(iv)如果 *eventPointer 不是指向信号量的指针，则返回 OS_ERR_EVENT_TYPE。

传递 PN 的任务参数：(i)OS_Event *eventPointer 作为指针传递给和信号量关联的 ECB，这些信号量可能是 SemFlag、SemKey 或者 SemCount。(ii)传递超时时间 timeOut 的定时器节拍数。任务在延时超过 timeOut 后，即使信号量没有被释放，仍然恢复执行。这样做就避免了无限制的等待。如果不使用这个规定，则这个参数必须传递 0 值。(iii)传递*err，持有错误代码的指针。

为进一步了解 OSSemPend 的使用，请参考示例 10-16、示例 10-17 和示例 10-18 的内容。

(3) 在信号量释放后检查 IPC 的可用性

函数 unsigned short OSSemAccept(OS_Event *eventPointer)检查 ECB 中信号量的值是否大于 0。检索结果将是一个无符号 16 位的值，然后这个值会减 1。

返回 RO：如果 semVal 大于 0，函数 OSSemAccept()将其减 1，并返回预先被减少的值，以一个无符号的 16 位数表示。如果 semVal 是 0，并且信号量不是处于等待何时被送出(释放)的待定状态，则返回 0。

传递 PO 的任务参数：OS_Event *eventPointer 传递和信号量 semVal 关联的 ECB 的指针。

该函数的使用在示例 10-18 中的步骤 25。

(4) 在信号量释放后发送 IPC

函数 unsigned byte OSSemPost(OS_Event *eventPointer)让另外一个正在等待的任务在函数执行后不再等待，并发送用于信号量释放事件的 IPC，该信号量可能是 SemFlag、SemKey 或者 SemCount(示例 10-16)。后者处在*eventPointer 所指的 ECB 中。SemFlag、SemKey 或者 SemCount 加 1，如果它们变得比 0 大，则它是一个释放处于等待状态的任务的事件。现在任务准备就绪，可以运行(如果没有优先级更高的任务等待运行，它们就会运行)。SemFlag、SemKey 或者 SemCount 加 1，如果变成大于 0，那么信号量可能再次进入待定状态，并且其他任务必须等待它的释放。

返回 RP：如果 semVal 是 0 或者大于 0，函数 OSSemPost ( )则将其加 1，并返回以下三种错误代码中的一种。(i)如果信号量给出成功信号(SemVal 大于 0 或等于 0)，则返回 OS_NO_ERR。(ii)如果*eventPointer 指针没有指向信号量，则返回 OS_ERR_EVENT_TYPE。(iii)如果 semVal 溢出(已经是 65535，不能再加 1)，则返回 OS_SEM_OVF。

传递 PP 的任务参数；OS_Event *eventPointer 作为指针传递给和信号量关联的 ECB。

想进一步了解 OSSemPost 的用法，请参考示例 10-16、示例 10-17 和示例 10-18 的内容。

示例 10-16

OSSemPost 和 OSSemPend 作为事件信号标志的使用如下。在使用 OSSemCreate 创建信号量时，将事件标志的初始值 SemFlag 置 0。任务首先执行 OSSemPost，将 SemFlag 值增加到 1，然后发信号通知事件。当 SemFlag 变成 1 被释放(未得到)时，正在发送信号量 1 的等待任务(执行 OSSemPend 的任务)可以开始运行(在没有其他优先级更高的任务等待运行时，该任务就会运行)。信号量 SemFlag 从 OSSemPen 函数返回时减到 0(再次进入待定状态或者未得到状态)。此时任务的等待代码开始运行。

回顾示例 5-1 中从网络读取字节的内容。其中给出了步骤 a、b 和 c 如何在等待和发送 IPC 中使用信号量标志来实现同步。

(1) 对于步骤 a，任务改成 Task_CharCheck，该任务检查端口 A 的字符，如果端口不可用就等待。

(2) 对于步骤 b，任务改成 Task_Read_Port_A，该任务当端口 A 可用时，读取字符。

(3) 对于步骤 c，任务改成 Task_Decrypt_Port_A，该任务对消息进行译码。

创建 3 个任务和 3 个任务实现同步的代码如下：

```
1. /* Codes as per Example 10.7 Step 1 except last comment line*/
.
.
2. /* Preprocessor definitions for maximum number of inter process events to
let the MUCOS allocate memory for the Event Control Blocks */
#define OS_MAX_EVENTS 8/* Let maximum IPC events be 8 */
#define OS_SEM_EN 1/* Enable inclusion of semaphore functions in applications
using MUCOS */
/* End of preprocessor commands */
3. /*Codes as per Example 10.7 Step 2 */
.
.
4. /* Prototype definitions for three tasks, stacks and priorities. */
static void Task_CharCheck (void *taskPointer);
static void Task_Read_Port_A (void *taskPointer);
static void Task_Decrypt_Port_A (void *taskPointer);
static OS_STK Task_CharCheckStack [Task_CharCheckStackSize];
static OS_STK Task_Read_Port_AStack [Task_Read_Port_AStackSize];
static OS_STK Task_Decrypt_Port_AStack [Task_Decrypt_Port_AStackSize];
#define Task_CharCheckStackSize 100 /* Define task 1 stack */
#define Task_Read_Port_AStackSize 100 /* Define task 2 stack */
#define Task_Decrypt_Port_AStackSize 100 /* Define task 3 stack */
#define Task_CharCheckPriority 6 /* Define task 1 priority */
#define Task_Read_Port_APriority 8 /* Define task 2 priority */
#define Task_Decrypt_Port_APriority 10 /* Define task 3 priority */
5. /* Prototype definitions for the semaphores */
OS_EVENT *SemFlag1; /* Needed when using Semaphore as flag for inter-process
communication between port check and port read tasks. Port read has to wait for
check O.K.*/
OS_EVENT *SemFlag2; /* Needed when using Semaphore as flag for inter-process
```

```
communication between port read and port read decipher task. Port decrypting
has to wait for port read */
OS_EVENT *SemKey1; /* Needed when using Semaphore as resource key as in Example
7.17*/
OS_EVENT *SemCount; /* Needed when using Semaphore as Counting as in Example
7.18*/
6. /* Codes as per Example 10.7 Step 3 to 5 */
7. /* Create Semaphores and Start MUCOS RTOS to let us RTOS control and run the
created tasks */
SemFlag1 = OSSemCreate (0); /*Declare initial value of semaphore = 0 for using
it as an event flag*/
SemFlag2 = OSSemCreate (0); /*Declare initial value of semaphore = 0 for using
it as an event flag*/
OSStart ( );
/* Infinite while-loop is there in each task. So there is no return from the
RTOS function OSStart
( ). */
} /* End of while loop*/
}/ *** End of the Main function ***/
/* Codes as per Example 10.7 Step 7 and 8 */
8. /* Create three tasks as per Step 2 by defining three task identities,
Task_CharCheck,
Task_Read_Port_A and Task_Decrypt_Port_A and the stack sizes and other TCB
parameters.
*/
OSTaskCreate (Task_CharCheck, void (*) 0, (void *) & Task_CharCheckStack
[Task_CharCheckStackSize], Task_CharCheckPriority);
OSTaskCreate (Task_Read_Port_A, void (*) 0, (void *) & Task_Read_Port_AStack
[Task_Read_Port_AStackSize], Task_Read_Port_APriority);
OSTaskCreate (Task_Decrypt_Port_A, void (*) 0, (void *) &
Task_Decrypt_Port_AStack
[Task_Decrypt_Port_AStackSize], Task_Decrypt_Port_APriority);
9. while (1) { /* Infinite loop of FirstTask */
.
.
10. /* Suspend, with no resumption later, the First task as it must run once
only for initiation of
timer ticks and for creating the tasks that the scheduler controls by preemption.
*/
11. OSTaskSuspend (FirstTask_Priority); /*Suspend First Task and control of the
RTOS
passes to other tasks waiting execution*/
12. } /* End of while loop */
13. } /* End of FirstTask Codes */
/
**************************************************************************/
14. /* The codes for the Task_CharCheck */
static void Task_CharCheck (void *taskPointer) {
15. /* Initial assignments of the variables and pre-infinite loop statements
that execute once
only. Also refer to Example 4.5*/
.
```

```
.
16. /* Start an infinite while-loop. */
while (1) {
17. /*Codes for Task_CharCheck */
.
.
18. /* Let the characters reach Port A at 300 characters per second (every 3.3
ms only). Wait for
3 ms. This will also let the other task of lower priority execute Port A read
task, and then allow
another task to decrypt Port A message. */
OSTimeDly (3);
19. /* Release semaphore to a task waiting for the read at Port A */
OSSemPost (SemFlag1);
20. }; /* End of while loop*/
21. }/ * End of the Task_CharCheck function */
/
*************************************************************************/
22. /* The codes for the Task_Read_Port_A */
static void Task_Read_Port_A (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that
execute once only*/
.
.
.
23. /* Start an infinite while-loop. */
while (1) {
24. / *Wait for SemFlag1 =1 by OSSemPost function of character availability check
task */
OSSemPend (SemFlag1, 0, &SemErrPointer);
25. /* Codes for reading from Port A and storing message at a queue or buffer*/
.
.
26. /* Release semaphore to a task waiting for the decrypting*/
OSSemPost (SemFlag2);
OSTimeDlyResume (Task_CharCheckPriority); /* Resume the delayed character check
task */
27. }; /* End of while loop*/
28. }/ * End of the Task_Read_Port_A function */
/
*************************************************************************/
29 /* Start of Task_Decrypt_Port_A codes */
static void Task_Decrypt_Port_A (void *taskPointer) {
30. /* Initial assignments of the variables and pre-infinite loop statements
that execute once only*/
.
.
.
31. /* Start an infinite while-loop. */
while (1) {
OSSemPend (SemFlag2, 0, &SemErrPointer); / *Wait for SemFlag2 =1 by OSSemPost
function for a
```

```
character read at Port A */
32'. /* Codes for Task_Decrypt_Port_A*/
.
.
33. }; /* End of while loop*/
34. }/ * End of the Task_Decrypt_Port_A function */
/
*****************************************************************************/
```

示例 10-17

OSSemPost 和 OSSemPend 作为资源获取键的使用方法如下：设置资源键可用标志 SemKey 的初始值为 1。任务必须首先执行 OSSemPend，该函数将 SemKey 值减到 0，然后运行任务的临界段的代码。同样的任务必须当其代码在临界段中执行完后执行 OSSemPost，然后将资源键可用的信号发给其他任务。SemKey 变成 1，并在从 OSSemPost 函数返回时释放(未得到)。如果没有其他优先级更高的任务等待运行，则另外一个和先前任务共享数据的任务就会在进入共享数据区时执行 OSSemPend 函数，并在从共享数据区离开时执行 OSSemPost 函数。在任务部分将 SemKey 置 0 和 1，就可以使任务在其特定的运行状态中获取资源。

回顾示例 4-1 步骤 b 和步骤 c 中从网络读取消息进而编码的内容。如果没有消息，如何进行编码呢？对于这个问题可以回头看看示例 10-16。现在这个例子将给出步骤 b 和步骤 c 如何使用信号量键来实现同步，这里的信号量键是用于键等待和键发送 IPC 的，以及步骤 a 和步骤 b 如何使用信号量标志来实现同步。

```
1. /* Codes as per Example 10.16 Step 1 to 5 */
.
.
2. /* Prototype definition for the semaphore used as resource key for
inter-process communica-tion between port read and port message encrypt read
tasks. */
OS_EVENT *SemKey1; /* Needed when using Semaphore as resource key*/
/* Codes as per Example 10.16 Steps 6 to 21. However, must create the semaphore
SemKey1before calling OSStart ( ) in main*/
SemKey1= OSSemCreate (1); /*Declare initial value of semaphore = 1 for using
it as a resource
acquiring key*/
/
*****************************************************************************/
4. /* After the end of codes for Task_CharCheck, the codes for the task_Read_Port_A
rede-fined to show a use of the key*/
static void Task_Read_Port_A (void *taskPointer) {
5. /* Initial assignments of the variables and pre-infinite loop statements that
execute once
only*/
.
.
.
6. while (1) { /* Start an infinite while-loop. */
7. OSSemPend (SemFlag1, 0, &SemErrPointer); / *Wait for SemFlag1 =1 by OSSemPost
func-tion
of character availability check task */
```

```
/*Acquire resource as SemKey presently > 0 and decrement it and not allow any
other task to use this key*/
8. OSSemPend (SemKey1, 0, &SemErrPointer);
9. /* Codes for reading from Port A and storing at a queue or buffer*/
.
.
10. /* Release the key to a task waiting for the decrypting*/
OSSemPost (SemKey1);
11. /* To exit the infinite loop at a task that has been assigned a higher priority
and to let the lower priority task run call OS delay function for wait of 1 ms
(one OS timer tick. This is the method to let the other task of lower priority
execute Port A decrypt. */
OSTimeDly (1);
}; /* End of while loop*/
12. }/ * End of the Task_Read_Port_A function */
/
*************************************************************************/
13. /* Start of Task_Decrypt_Port_A codes */
static void Task_Decrypt_Port_A (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that
execute once only*/
.
.
14. while (1) { /* Start the infinite loop */
15. / Acquiring the resource as SemKey1 > 0 and decrement it to not to let port
read task use this key*/
OSSemPend (SemKey1, 0, *SemErrPointer);
16. /* Codes for Task_Decrypt_Port_A or deciphering the message read at Port
A*/
.
.
17. /* Release the key to a task waiting for the port read*/
OSSemPost (SemKey1);
OSTimeDlyResume (Task_Read_Port_APriority); /* Resume the delayed task */
18. }; /* End of while loop*/
19. } / * End of the Task_Decrypt_Port_A function */
/
*************************************************************************/
```

示例 10-18

为了解释计数信号量的使用，我们在 11.1 节将给出一个巧克力自动售卖机的案例研究。

计数信号量的使用对打印机缓冲区，或者有界缓冲区问题(生产者-消费者问题)的编程很有帮助(参见 8.1.2(vi))。在计数信号量中 OSSemPost 增加计数值和 OSSemPend 减少计数值的使用方法如下。回顾示例 5-1a，我们首先按照如下步骤修改这个例子：

(1) 对于步骤 a，任务改成 Task_CharCheck，该任务检查端口 A 的字符，如果端口不可用就等待。

(2) 对于步骤 b，任务改成 Task_Read_Port_A，当端口 A 可用时，读取字符。让 Task_ReadPostA 从网络端口 A 读取字符流。当任务逐个读取字符时，它将读到的字符放进有界缓冲区——(这是生产任务，缓冲区被类似打印机缓冲区的限制界定着)。

(3) 对于步骤 c，任务改成 Task_Decrypt_Port_A，它进行消息译码(这是消费任务。它类似于从打印缓冲区中打印字符)。

(4) 对于步骤 d，任务改成 Task_Encrypt_PortB。任务为另外一个端口对被译码的消息重新进行编码。

(5) 对于步骤 e，任务改成 Task_SendPortB。任务为另外一个端口对被译码的消息重新进行编码。

这个示例显示了步骤 b 和步骤 c 如何使用计数信号量来实现同步，以及步骤 a~e 如何实现同步。步骤 a 和 b 以及步骤 d 和 e 使用任务间 IPC 的两个信号量标志实现同步。

(1) 保证有一个计数器 SemCount，用它计算任务发送信号量的次数。将 SemCount 的初始值置 0。任务的开始部分必须首先执行 OSSemPost，它将 SemCount 加到 1。每次该任务运行，SemCount 就加 1。每次任务对另外一个任务中的字符进行译码，SemCount 就减 1。

(2) 当 SemCount 达到了特定的预设值时，信号量事件标志 SemCountLimitFlag 复位，并且计数值重新置 0。如果有一个延迟函数的 OS 调用，那么用于译码的优先级较低的任务开始运行并获取键。信号量资源键 SemKey 变得对读任务不可用，进一步的读操作会停止，直到译码任务释放键，并执行 OSSemPend 来减小 SemCount 的值，从而使到达界限的任务再次运行。

```
1. /* Codes as per Example 10.17 Steps 1 and 2*/
.
.
2. /* Prototype definitions for five tasks*/
static void Task_CharCheck (void *taskPointer);
static void Task_Read_Port_A (void *taskPointer);
static void Task_Decrypt_Port_A (void *taskPointer);
static void Task_Encrypt_PortB (void *taskPointer);
static void Task_SendPortB (void *taskPointer);
3. /* Definitions for five task stacks */
static OS_STK Task_CharCheckStack [Task_CharCheckStackSize];
static OS_STK Task_Read_Port_AStack [Task_Read_Port_AStackSize];
static OS_STK Task_Decrypt_Port_AStack [Task_Decrypt_Port_AStackSize];
static OS_STK Task_ EncryptPortBStack [Task_ EncryptPortBStackSize];
static OS_STK Task_SendPortBStack [Task_SendPortBStackSize];
4. /* Definitions for five task stack size */
#define Task_CharCheckStackSize 100 /* Define task 1 stack size*/
#define Task_Read_Port_AStackSize 100 /* Define task 2 stack size*/
#define Task_Decrypt_Port_AStackSize 100 /* Define task 3 stack size*/
#define Task_EncryptPortBStackSize 100 /* Define task 4 stack size*/
#define Task_SendPortBStackSize 100 /* Define task 5 stack size*/
5. /* Definitions for five task priorities. */
#define Task_CharCheckPriority 6 /* Define task 1 priority */
#define Task_ ReadPortAPriority 7 /* Define task 2 priority */
#define Task_DecryptPortAPriority 8 /* Define task 3 priority */
#define Task_EncryptPortBPriority 9 /* Define task 4 priority */
#define Task_SendPortBPriority 10 /* Define task 5 priority */
6. /* Prototype definitions for the semaphores */
OS_EVENT *SemFlag1; /* Needed when using semaphore as the flag for inter-process
commu-nication between port check and port read tasks. Port A read task has to
wait for check O.K.*/ OS_EVENT *SemFlag2; /* Needed when using semaphore as the
flag for inter-process commu-nication between encrypting Port B task and sending
```

```
task for Port B. */
OS_EVENT *SemCountLimitFlag; /* Needed when using semaphore as the flag for
limiting the semaphore count value in the inter-process communication between
port read and port read decipher task. Port reading has to wait for port read
*/
OS_EVENT *SemKey; /* Needed when using semaphore as resource key */
OS_EVENT *SemCount; /* Needed when using semaphore as counting as in Example
7.18*/
8. /* Codes as per Example 10.7 Step 3 to 8. However, the semaphores are to be
created and initialised as under */
SemFlag1 = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using
it as an event flag*/
SemFlag2 = OSSemCreate (0); /* Declare initial value of semaphore = 0 for using
it as an event flag*/
SemCountLimitFlag = OSSemCreate (0); /* Declare initial value of semaphore =
0 for an event flag*/
SemKey = OSSemCreate (1); /* Declare initial value of semaphore = 1 for using
it as a resource key*/
/* Declare initial value of semaphore count = 0 for using as a counter that gives
the number of times atask, which sends into a buffer that stores a character
stream, ran minus the number of times the task which used the character from
the stream ran from the buffer */
SemCount = OSSemCreate (0);
.
9. /* Create five tasks as per Step 3, defined by three task identities,
Task_CharCheck, Task Read_Port_A, Task_Decrypt_Port_A, Task_EncryptPortB and
Task_SendPortB and the stacksizes and other TCB parameters. */
OSTaskCreate (Task CharCheck, void (*) 0, (void *) & Task_CharCheckStack
[Task_CharCheckStackSize], Task CharCheckPriority);
OSTaskCreate (Task_Read Port_A, void (*) 0, (void *) & Task_Read_Port_AStack
[Task_Read_Port_AStackSize], Task ReadPortAPriority);
OSTaskCreate (Task_Decrypt_Port_A, void (*) 0, (void *) &
Task_Decrypt_Port_AStack
[Task_Decrypt_Port_AStackSize], Task_DecryptPortAPriority);
OSTaskCreate (Task_EncryptPortB, void (*) 0, (void *) & EncryptPortBStack [Task_
EncryptPortBStackSize], Task_EncryptPortBPriority);
OSTaskCreate (Task_SendPortB, void (*) 0, (void *) & Task_SendPortBStack
[Task_SendPortBStackSize], Task_SendPortBPriority);
10. /* Codes same as at Steps 9 to 21 in Example 10.16 */
.
.
11. /* The codes for the Task_Read_Port_A redefined to use the key, flag and
counter*/
static void Task_Read_Port_A (void *taskPointer) {12. /* Initial assignments
of the variables and pre-infinite loop statements that execute once only*/
unsigned short * countLimit = 80; /* Declare the buffer-size for the characters
countLimit = 80 */
.
.
13. while (1) { /* Start an infinite while-loop. */
14. / *Wait for SemFlag1 1 by OSSemPost function of character availability check
task */ OSSemPend (SemFlag1, 0, &SemErrPointer);
```

```
15. /* Take the key to not let port decipher the task that needs SemKey run */
OSSemPend (SemKey, 0, &SemErrPointer);
16. if (*SemCount > = countLimit) {
OSSemPost (CountLimitFlag); /* Post the CountLimitFlag */
/* To exit the infinite loop of this assigned higher priority task to let the
lower priority task run call the OS delay function for a wait of 1 ms (one OS
timer tick). This is the method to let the other task of lower priority execute
Port A's message deciphering task */
OSSemPost (SemKey); /* Release the SemKey to let the next cycle of this loop
start */ OSTimeDly (1);
OSSemPend (SemKey, 0, &SemErrPointer); /* Take the SemKey */
};/* End of codes for the action on reaching the limit of putting characters
into the buffer */
17. /* Codes for reading from Port A and storing a character at a queue or buffer*/
.
.
18. OSSemPost (SemCount); /*Let the counting semaphore value increase because
one character has been put into the buffer holding the character stream*/
19. OSSemPost (SemKey); /* Release the SemKey to let next cycle of this loop
start */
20. }; /* End of while loop*/
21. } / * End of the Task_Read_Port_A function */
/
*****************************************************************************/
22. /* Start of Task_Decrypt_Port_A codes */
static void Task_Decrypt_Port_A (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that
execute only once*/
.
.
23. while (1) { /* Start the infinite loop */
24. /* Take the key to not letting the Task_Read_Port_A run before at least one
cycle of this while loop*/
OSSemPend (SemKey, 0, &SemErrPointer);
25. /* Decrease SemCountLimitFlag if. >0. Accept the SemCountLimitFlag if
available (>0). There is no need to suspend task if not available. The decrypting
task has to run whether the countLimit is exceeded or not. The only condition
is that it must run if exceeded. Hence the OSSemPend is not used here. */
OSSemAccept (SemCountLimitFlag);
26. /* Wait for SemCount to become > 0 and decrease the Semaphore Count */
OSSemPend (SemCount, 0, &SemErrPointer);
27. /* Codes for Task_Decrypt_Port_A or deciphering the message read at Port
A*/
.
.
28. /* Release the key to let a task, which was waiting for the port read run*/
OSSemPost (SemKey);
OSTimeDlyResume (Task_ ReadPortAPriority); /* Resume the task Port A Read
29. }; /* End of while loop*/
30. }/ * End of the Task_Decrypt_Port_A function */
/
*****************************************************************************/
```

```
31. /* The codes for the Task_EncryptPortB */
static void Task_EncryptPortB (void *taskPointer) {
32. /* Initial assignments of the variables and pre-infinite loop statements
that execute only
once */
.
.
.
33. while (1) { /* Start an infinite while-loop. */
34. /* Codes for encrypting again the deciphered characters of Port A */
.
.
35. /* Release the key to a task waiting for the decrypting*/
OSSemPost (SemFlag2);
36. /* To exit the infinite loop at the assigned higher priority task to let
the lower priority task run and call the OS delay function for wait of 1 ms (one
OS timer tick). This is the method to let the other task of lower priority execute
Port B sending the characters. */
OSTimeDly (1);
}; /* End of while loop*/
37. }/ * End of the Task_EncryptPortB function */
/
*********************************************************************/
38. /* Start of Task_SendPortB codes */
static void Task_SendPortB (void *taskPointer) {
/* Variable initial assignments and pre-infinite loop statements that execute
only once */
.
.
39. while (1) { /* Start the infinite loop */
40. /* Take the event flag SemFlag2 > 0 and decrement it */
OSSemPend (SemFlag2, 0, &SemErrPointer);
41. /* Codes for Task_SendPortB the message encrypted for Port B*/
.
.
OSTimeDlyResume (Task_EncryptPortBPriority); /* Resume Delayed task Encrypt
Port B */
42. }; /* End of while loop*/
43. }/ * End of the Task_SendPortB function */
/
*********************************************************************/
```

(5) 检索信号量的错误信息

函数 unsigned byte OSSemQuery(OS_EVENT *eventPointer, OS_SEM_DATA *SemData)将信号量的数据值放到指针 SemData 中，并从 ECB 中检查信号量错误信息参数 OS_NO_ERR 和 OS_ERR_EVENT_TYPE。

返回 RQ：函数 OSSemQuery()返回错误代码：(i)如果查询成功，则返回 OS_NO_ERR。(ii)如果*eventPointer 指针没有指向信号量，则返回 OS_ERR_EVENT_TYPE。

传递 PQ 的任务参数：函数 OSSemQuery( )传递：(i)先前在*eventPointer 创建的信号量指针。(ii)为创建的信号量传递在*SemData 的数据结构指针。

10.2.6 邮箱相关函数

在示例 10-18 中已经看到，信号量 SemCount 的值加 1，就开始从执行端口 A 读取操作的任务到另一个译码任务的通信。在对字符进行译码时，SemCount 减小，并传回给任务用于读取操作。这就产生了两个任务共享有界缓冲区的情形。SemCount 是一个 16 位的消息，在这些任务间的 IPC 中，发送和使用(产生放进缓冲区并从缓冲区中流出)字符。我们可以使用计数信号量，如一个 16 位的值，比如在 MUCOS 中就是使用这样的 IPC 函数作为信号量(参见信号量 SemVal 的使用，它将字符从小键盘传递给其他的等待任务)。

但是，假设消息不是 16 位，而比许多字节或者字更长，则不使用 16 位消息的信号量 IPC，而使用邮箱 IPC 传送指向这些信息的指针。参见图 8-6(a)，图中给出了邮箱的多种类型。在 MUCOS 中，邮箱类型是每个邮箱的一个消息指针。

保证有一个指向邮箱的指针*mboxMsg，和另外一个指向消息事件的指针*MsgPointer(用于检索消息自身)。用于任务的 MUCOS 邮箱 IPC 函数如表 10-6 所示。

表 10-6 用于任务间通信的邮箱 RTOS 邮箱函数

服务和系统时钟函数原型	返回和传递的参数	何时调用 OS 函数
OS_Event *OSMboxCreate(void *mboxMsg)	M1 和 M1	为邮箱消息的 ECB 创建和初始化邮箱消息指针时调用该函数
void *OSMboxAccept (OS_EVENT *mboxMsg)	M2 和 M2	检查在 *MsgPointer 的邮箱消息 a 是否在 *mboxMsg 可用时调用该函数。不像 OSMboxPend 函数，如果消息不可用，它就不阻塞(挂起)任务。如果可用，它就返回指针并且 *mboxMsg 再次指向 NULL
void *OSMboxPend (OS_Event *mboxMsg, unsigned short timeOut, unsigned byte *MboxErr)	M3 和 M3	检查待定(可用)邮箱消息的消息指针是否被读取，并且邮箱是否已清零时调用该函数，*mboxMsg 再次指向 NULL。如果消息不可用(*mboxMsg 指向 NULL)，它就等待，然后挂起任务，并阻止进一步运行(直到系统定时器发出 timeOut 节拍)。如果任务处于待定状态，那么就依据可用性恢复执行。如果超时，任务也会恢复执行
unsigned byte OSMboxPost (OS_EVENT *mboxMsg, void *MsgPointer)	M4 和 M4	通过发送地址指针给 mboxMsg 来发送处于地址 MsgPointer 上的任务消息。如果优先级较高的任务也将发送消息，那么就将上下文切换到这个任务或者任意其他任务。如果邮箱满，那么消息就没有地方放置，并会给出错误消息
unsigned byte OSMbox Query(OS_EVENT *mbox Msg,OS_MBOX_DATA *mboxData)	M5 和 M5	获得数据结构中的邮箱错误信息 OS_NO_ERR 和 OS_ERR_EVENT_TYPE 时调用该函数

第 2 列请参考文中使用邮箱进行任务间通信的相应段落。

(1) 为 IPC 创建邮箱

函数 OS_Event *OSMboxCreate(void *mboxMsg)在 RTOS 中创建一个 ECB，并返回一个指针，指向带有*mboxMsg 指针的 ECB。

传递 M1 的任务参数：*mboxMsg 是消息邮箱指针。对于 IPC，是通过发送消息指针*mboxMsg 来传送消息。

返回 M1：函数 OSMboxCreate( )返回指针给 MUCOS 中的 ECB。

示例 10-19 中步骤 8 介绍了如何使用 OSMboxCreate 函数。

(2) 检查邮箱收到消息后 IPC 的可用性

函数 void *OSMboxAccept(OS_EVENT *mboxMsg)检查 ECB 中 mboxMsg(事件指针)所指的邮箱消息。如果消息可用(*mboxMsg 不是指向 NULL)，则指向消息(*MsgPointer)的指针在返回时进行检索。返回后，邮箱被清空。随后*MboxMsg 指针将指向 NULL(它与 OSMboxPend 函数的区别在于，如果消息不可用，OSMboxPend 则挂起任务并等待*mboxMsg 变为非 NULL)。

传递 M2 的任务参数：OS_Event *mboxMsg 作为指针传递给和邮箱相关的 ECB。

返回 M2：函数 OSMboxAccept()检查*mboxMsg 指向的消息，并返回消息指针*MsgPointer。如果消息指针不可用(*mboxMsg 指向 NULL)，则返回 NULL 指针。

如果邮箱中有任何可用的错误字符串，并且没有特别等待和阻塞任务，那么如何使用 OSMboxAccept 来检索这些错误字符串，这个问题在示例 10-19 的步骤 40 进行了介绍。

(3) 等待邮箱中消息用到的 IPC 的可用性

函数 void *OSMboxPend(OS_Event *mboxMsg, unsigned short timeOut, unsigned byte *MboxErr)检查 ECB 事件指针*mboxMsg 指向的邮箱消息。如果消息可用(*mboxMsg 不是指向 NULL)，则消息的指针在返回进行检索，否则就等待，直到消息可用或者超时，而不管消息可用和超时哪一个先发生。如果 timeOut 参数值是 0，那就意味着要无限制等待，直到消息可用为止。

传递 M3 的任务参数：(i)OS_Event *mboxMsg 作为指针传递给和邮箱消息相关的 ECB。(ii)传递参数 timeOut。在延迟等于(timeOut-1)个 RTOS 定时器的计数输入(节拍)时恢复执行阻塞任务。(iii)传递*MboxErr，它是持有错误代码的指针。

返回 M3：函数 OSMboxPend 检查并等待*mboxMsg 指向的消息，然后返回*MsgPointer 指针。返回后，邮箱被清空。随后*mboxMsg 将指向 NULL。当消息不可用时，该函数就挂起任务，并且只要*mboxMsg 不是 NULL，就阻塞任务。如果消息不可用(mboxMsg 指向 NULL)，函数就返回 NULL 指针。返回情形如下：(i)如果邮箱消息检索成功，就返回 OS_NO_ERR；(ii)如果在节拍定义的超时限制 timeout 大于 0 时，邮箱消息检索没有成功，则返回 OS_TIMEOUT。(iii)如果函数调用来自 ISR，则返回 OS_ERR_PEND_ISR。(iv)如果 mboxMsg 不是指向邮箱*MsgPointer 消息，则返回 OS_ERR_EVENT_TYPE。

示例 10-19 的步骤 40 介绍了如何使用 OSMboxPend 来检索错误字符串，如果使用 Task_ErrSR 检索到任何邮箱中可用的错误字符串，那么如何检索在 Task_OutPortB 的读字符串，并特地等待和阻塞任务？

(4) 通过邮箱发送消息获得 IPC

函数 unsigned byte OSMboxPost(OS_EVENT *mboxMsg, void *MsgPointer)发送 ECB 事件指针*mboxMsg 指向的邮箱消息。这个被发送的消息就是*MsgPointer。

传递 M4 的任务参数：(i)OS_Event *mboxMsg 作为指针传递给和消息相关的 ECB。该消息被送到邮箱地址*mboxMsg。(ii)*MsgPointer 被传递给 ECB。

返回 M4：函数 OSMboxPost() 发送消息，然后返回如下 3 种错误代码之一：(i)如果有效消息发送成功，则返回 OS_NO_ERR。(ii)如果*MsgPointer 没有指向*mboxMsg 所指的消息，则返回 OS_ERR_EVENT_TYPE。(iii)如果在 mboxMsg 的邮箱已经有未被接受或是被返回的消息，则返回 OS_MBOX_FULL。

示例 10-19 的步骤 40 给出了 OSMboxPost，通过发送消息给等待任务以获取消息。

(5) 为邮箱寻找邮箱数据并检索错误信息

函数 unsigned byte OSMboxQuery(OS_EVENT *mboxMsg, OS_MBOX_DATA *mboxData) 检查邮箱数据，并将其放到 mboxData 中。该函数也为 ECB 寻找错误信息参数 OS_NO_ERR_EVENT_TYPE。

传递 M5 的任务参数：函数 OSMboxQuery()传递(i)先前在*mboxMsg 创建的邮箱消息的指针和(ii)在 mboxData 的数据结构的指针。

返回 M5：函数 OSMboxQuery()在将邮箱信息存储到 mboxData 之后，返回错误代码的情形如下：(i)如果邮箱消息查询成功，则返回 OS_NO_ERR。(ii)如果 MsgPointer 不是指向 mboxMsg 处邮箱中的消息，则返回 OS_ERR_EVENT_TYPE。

示例 10-19

11.1 节将给出邮箱消息的一个使用范例，关于巧克力自动售卖机的案例研究。邮箱使用方法如下：

(1) 任务为 Task_Read_Port_A，在收到消息字符串(读取端口 A 接收到的一组字符)之后，使用 OSMboxPost 发送 IPC 到另外一个正在等待(被阻塞)的任务，以获取该消息。一种示范情形是，从移动电话中端口 A 的小键盘接收一个被呼叫 party 的电话号码，任务在确定号码中没有无效字符时，等待拨号并发送号码。等待任务是 Task_OutPortB 和 Task_SendPortB。后者在等待结束后发送字符串到端口 B。

(2) 不仅 Task_Read_Port_A，还有另外一个任务 Task_Err，在检测到无效字符或者字符串长度超过限制时，会发送错误消息给 Task_OutPortB。OSMboxPend 函数的应用程序等待消息和等待错误消息字符串(参见任务 Task_SendPortB 中通过 OSMboxPend 进行等待的内容，其中的任务是在检测到错误字符串的情况下执行服务例程)。例如，在移动电话中，Task_OutPortB 可以按照如下方式使用：当没有错误消息时，和网络服务建立连接，然后使用 Task_SendPortB 拨号并发送被叫号码。当有错误消息时，Task_OutPortB 直接发送消息给另外一个任务 Task_ErrSR。这个任务显示错误消息字符串，提醒用户在使用移动电话时通过小键盘重拨号码。以上操作的具体步骤如下：

步骤 a：任务 Task_CharCheck 检查端口 A 字符的可用性(例如，如果键盘被按下一个键，该任务就寻找相关字符并将其发送给端口 A)。如果字符不可用，则等待。这里，由于 IPC 仅仅是针对事件标志，所以使用如同示例 10-16 中的信号量 SemFlag1 就足够了。

步骤 b：任务 Task_Read_Port_A 等待 SemFlag1，并执行代码，将字符累加到数组中以获得字符串 str。如果在超时之前没有其他键按下，则 OSMboxPost 发送指向 str 的消息指针。

步骤 c：任务 Task_Err 检查在端口 A 中读到的每个字符，如果字符无效，则发送字符串 errStr 到邮箱中。例如，在步骤 b 中读电话号码时，得到的字符不是数字。这里，由于 IPC 消息只是一个 8 位的字符，没有必要将邮箱作为 IPC 使用，因而在步骤 b 和步骤 c 之间使用信号量值 SemVal 就足够了。

步骤 d：任务 Task_OutPortB 等待邮箱中的 str 和 errstr。在步骤 b 和步骤 c 之间以及步骤 c 和步骤 d 之间可以将邮箱作为 IPC 使用。

步骤 e：如果没有错误，任务 Task_SendPortB 就发送端口 B 的消息。

步骤 f：任务 Task_ErrSR 执行有错误时的服务例程。

该示例介绍了步骤 a 和步骤 b 如何通过 IPC SemFlag1 实现同步，步骤 b 和步骤 c 的任务如何使用 IPC SemVal 实现同步，以及步骤 b~d 和步骤 c、d 如何使用 MUCOS 的邮箱函数实现同步。

```
1 /* Define Boolean variable as per Example 6.1, define a NULL pointer to point
in case mailbox is empty and codes as per Example 10.17 Steps 1*/
typedef unsigned char int8bit;
#define int8bit boolean
#define false 0
#define true 1
/* Define a NULL pointer; */
#define NULL (void*) 0x0000;
.
.
2. /* Preprocessor definitions for maximum number of inter-process events to
let the MUCOS allocate memory for the Event Control Blocks */
#define OS_MAX_EVENTS 12/* Let maximum IPC events be 12 */
#define OS_SEM_EN 1/* Enable inclusion of semaphore functions in application.
*/
#define OS_MBOX_EN 1/* Enable inclusion of mailbox functions in application.
*/
/* End of preprocessor commands */
3. /* Prototype definitions for six tasks for steps a to e above. */
static void Task_CharCheck (void *taskPointer);
static void Task_Read_Port_A (void *taskPointer);
static void Task_Err (void *taskPointer);
static void Task_OutPortB (void *taskPointer);
static void Task_SendPortB (void *taskPointer);
static void Task_ErrSR (*taskPointer);
/* Definitions for six task stacks */
static OS_STK Task_CharCheckStack [Task_CharCheckStackSize];
static OS_STK Task_Read_Port_AStack [Task_Read_Port_AStackSize];
static OS_STK Task_ErrStack [Task_ErrStackSize];
static OS_STK Task_OutPortBStack [Task_OutPortBStackSize];
static OS_STK Task_SendPortBStack [Task_SendPortBStackSize];
static OS_STK Task_ErrSRStack [Task_ErrSRStackSize];
/* Definitions for six task stack size */
#define Task_CharCheckStackSize 100 /* Define task 1 stack size*/
#define Task_Read_Port_AStackSize 100 /* Define task 2 stack size*/
#define Task_ErrStackSize 100 /* Define task 3 stack size*/
#define Task_OutPortBStackSize 100 /* Define task 4 stack size*/
#define Task_SendPortBStackSize 100 /* Define task 5 stack size*/
#define Task_ErrSRStackSize 100 /* Define task 3 stack size*/
4. /* Definitions for six task priorities. */
#define Task_CharCheckPriority 5 /* Define task 1 priority */
#define Task_ ReadPortAPriority 6 /* Define task 2 priority */
#define Task_ErrPriority 7 /* Define task 3 priority */
```

```
#define Task_OutPortBPriority 8 /* Define task 4 priority */
#define Task_SendPortBPriority 9 /* Define task 5 priority */
#define Task_ErrSRPriority 10 /* Define task 6 priority */
5. /* Prototype definitions for the semaphores */
OS_EVENT *SemFlag1; /* Needed when using the semaphore as a flag for inter-process
com-munication between port status check task, Task_CharCheck and port read task,
Task_Read_Port_A Port A read task has to wait for semaphore till port A interrupts.
*/
OS_EVENT *SemFlag2; /* Needed when using semaphore as a flag for inter-process
communi-cation between Task_OutPortB task and sending task Task_SendPortB. */
OS_EVENT *SemCountLimitFlag; /* A flag to define the limits of the semaphore
count */
OS_EVENT *SemKey; /* Needed when using the semaphore as a resource key by
Task_Read_Port_A and Task_Err */
OS_EVENT *SemVal; /* Needed when using the semaphore for passing the 16-bit
message between steps b and c. */
OS_EVENT *SemCount; /* Needed when using the semaphore for passing the counts
as the number of characters read as a 16-bit message between tasks at the steps
b and d. */
6. /* Prototype definitions for the mailboxes */
OS_EVENT *MboxStrMsg; /* Needed when using the mailbox message between steps
b and d and Steps d and e */
OS_EVENT *MboxErrStrMsg; /* Needed when using the mailbox message between steps
c and d */
7. /* Codes as per codes in Example 4.5 except the main function codes. These
are for reading from Port A and storing a character at a queue or buffer */
.
.
8. /* Codes as per Example 10.7 Step 3 to 8. However, before the 'OSStart ( );
', the semaphore and mailbox must be created and initialised as under. */
SemFlag1 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using
it as an event flag*/
SemFlag2 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using
it as an event flag*/
SemCountLimitFlag = OSSemCreate (0) /* Declare initial value of semaphore = 0
as an event flag*/
SemKey = OSSemCreate (1) /* Declare initial value of semaphore = 1 for using
it as a resource key*/
SemVal= OSSemCreate (0) /* Declare initial value of semaphore character be =
'0' */
/ * Declare initial count as 0 as a counter that gives the number of times a
task, which sends into a buffer that stores a character stream, ran minus the
number of times the task, which used the character from the stream, ran from
the buffer */
SemCount = OSSemCreate (0);
/* Create Mailboxes for the tasks. */
MboxStrMsg = OSMboxCreate (NULL); /* Needed when using mailbox message between
steps b and d to pass a string message pointer*/
MboxErrStrMsg = OSMboxCreate (NULL); /* Needed when using mailbox message between
steps b and c to pass a string message pointer*/
9.
.
```

```
.
10. /* Create six tasks as per Step 3 defining six task identities, Task_CharCheck,
Task_Read_Port_A, Task_Err, Task_OutPortB and Task_SendPortB and the stack
sizes, other TCB parameters. */
OSTaskCreate (Task_CharCheck, void (*) 0, (void *) & Task_CharCheckStack
[Task_CharCheckStackSize], Task_CharCheckPriority);
OSTaskCreate (Task_Read_Port_A, void (*) 0, (void *) & Task_Read_Port_AStack
[Task_Read_Port_AStackSize], Task_ReadPortAPriority);
OSTaskCreate (Task_Err, void (*) 0, (void *) & Task_ErrStack [Task_ErrStackSize],
Task_ErrPriority);
OSTaskCreate (Task_OutPortB, void (*) 0, (void *) & Task_OutPortBStack
[Task_OutPortBStackSize], Task_OutPortBPriority);
OSTaskCreate (Task_SendPortB, void (*) 0, (void *) & Task_SendPortBStack
[Task_SendPortBStackSize], Task_SendPortBPriority);
OSTaskCreate (Task_ErrSR, void (*) 0, (void *) & Task_ErrSRStack [Task_ErrSRS
tackSize], Task_ErrSRPriority);
11. /* Codes same as those in Steps 9 to 17 in Example 10.16 */
.
.
12. /* Refer to Example 4.5 Wait till an interrupt occurs and sets STAF*/
while (STAF != 1) { };
/* Execute interrupt service routine for port A*/
13. if (STAF == 1) {OSSemPost (SemFlag1);} /* Post semaphore to a task waiting
for the read at Port A
*/
enable_PortA_Intr ( ); /* Prepare for another interrupt from port A*/
OSTimeDly (3); /* Let a low priority execute, as the next character at Port A
will take time. */
}; /* End of while loop*/
}/ * End of the Task_CharCheck function */
/
*************************************************************************/
14. /* The codes for the Task_Read_Port_A redefined to use the key, flag and
16-bit value and mailbox*/
static void Task_Read_Port_A (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that
execute only once*/ unsigned char *portAdata;
static void portA_ISR_Input (*portAdata);
unsigned char [ ] portAinputStr; /* Let port A input string be an array to hold
the data from port A*/
unsigned char * inputMaxSize = 16; /* Let Maximum size of the string be 16
characters. */
/* Start an infinite while-loop and Wait for SemFlag1 1 by OSSemPost function
of character availability check task */
while (1) {
OSSemPend (SemFlag1, 0, &SemErrPointer);
15. /* Take the key to not let the Task_Err run and block this task. Remember
that Task_Err also needs SemKey for running state */
OSSemPend (SemKey, 0, &SemErrPointer);
16. /* Actions on maximum size exceeding the string buffer Size */
if (*SemCount > = inputMaxSize) {
OSSemPost (SemCountLimitFlag); /* Post the SemCountLimitFlag */
```

```
OSSemPost (SemKey); /* Release the SemKey to let next cycle of this loop start */
/* To exit the infinite loop of this higher priority assigned task to let the
lower priority task error detect run by a call to the OS delay function that
forces a wait of 1 ms (one OS timer tick) or until delay resume function executes.
This is the method to let the other task of lower priority execute Port A message
error-detecting task */
OSTimeDly (1);
OSSemPend (SemKey, 0, &SemErrPointer); /* Take the SemKey */
SemCount =0; /* Reset the Semaphore counter to 0. */
};/* End of Codes for the action on reaching the limit of putting the characters
into the buffer */
18. /* Execute Port A interrupt service routine.*/
PortA_ISR_Input (&portAdata); /* Remember as soon as Port A is read, STAF will
reset itself to reflect next interrupt status. */
/* If an ASCII code for start of text is found then initialize SemCount - 0.
*/
If (portAdata = = 0x02) {SemCount = 0;};
19. /* Write the array element that returned as Port A data into the port A input
string */ PortAinputStr [SemCount] = portAdata;
20. /*Let the counting semaphore count increase after one character has been
put into the string holding the character stream*/
OSSemPost (SemCount);
21. /*Let the character pass as an IPC to Task_Err, not as a mailbox message
but as a semaphore value. Note that we are not using Mailbox as an IPC of a character
that has a message of 8-bits only. */
*SemVal = portAdata;
OSSemPost (SemVal);
22. OSTimeDly (1); /* Let a low priority Task_Err run in order to check the error
if any in the read byte */
23. /* Prepare for another interrupt from port A*/
enable_PortA_Intr ( );
OSTimeDlyResume (Task_CharCheckPriority);/* Resume the task that was delayed
to start this task*/
24. /* When the character is equal to End of Text, ASCII code at Port A (for
example, the Enter key pressed) is found or last String character is received,
send the message pointer String to the waiting mail box at Task_OutPortB and
make initial SemCount = 0 again for next string*/
if (SemVal = = 03 || SemCount = = inputMaxSize - 1) {OSMboxPost (MboxStrMsg,
portAinputStr); SemCount = 0};
25. OSSemPost (SemKey); /* Release the SemKey to let next cycle of this while
loop start */ }; /* End of while loop*/
26. }/ * End of the Task_ReadPortA function */
/
*************************************************************************
27. /* Start of Task_Err codes */
static void Task_Err (void *taskPointer) { /* Initial assignments of the
variables and pre-infinite loop statements that execute once only*/
/*Declare an error string for using when at the Step d an error invalid-character
is found at the mailbox.unsigned char [ ] ErrStr1 = "Invalid Character Found";
/* Declare an error string message for task at Step d when the limit exceeds.
*/
unsigned char [ ] ErrStr2 = "Characters in the message exceeded the Limit declared
```

```
at the task for read";
boolean invalid = false; /* Declare invalid variable 'false' and will be assigned
'true' when character read is found invalid. * /
28. while (1) { /* Start the infinite loop */
29. /* Take the key to not to let port read task read at least one cycle of the
hile loop*/
OSSemPend (SemKey, 0, &SemErrPointer);
30. /* Post Limit of Message Exceeded Message to task at step d. */
If (SemCountLimitFlag = = 1) {OSMboxPost (MboxErrStrMsg, ErrStr2);};
31. /* Codes for Checking any invalid character in SemVal*/
OSSemAccept (SemVal);
.
.
32. /* Post Mailbox message to task at step d if an invalid character is detected
*/
if (SemCountLimitFlag == 0 && invalid = = true) {OSMboxPost (MboxErrStrMsg,
ErrStrl);};
33. /* Decrease SemCountLimitFlag (if >0) by accepting the SemCountLimitFlag
semaphore. Task does not suspend even if semaphore not available (not > 0). This
task has to run whether  ountLimit is exceeded or not. The only condition is
that it must run if exceeded. Hence the OSSemPend is not used here. */
OSSemAccept (SemCountLimitFlag);
34. /* Release the key to let a task, which was waiting for the port read run*/
OSSemPost (SemKey);
OSTimeDlyResume (Task_ ReadPortAPriority); /* Resume the task Port A Read
}; /* End of while loop*/
35. } / * End of the Task_Err function */
/
*************************************************************************/
36. /* The codes for the Task_OutPortB */
static void Task_OutPortB (void *taskPointer) {
37. /* Initial assignments of the variables and pre-infinite loop statements
hat execute once only*/
unsigned char [ ] message; /* Declare error free message string pointer*/
unsigned char [ ] errMessage; /* Declare error message pointer. */
38. while (1) { /* Start an infinite while-loop. */
39. /* Wait for Mailbox Message available (not NULL) */
message = OSMboxPend (MboxStrMsg, 0, & Mboxerr);
40. /* Check for Mailbox Error Message available (not NULL) */
errMessage = OSMboxAccept (MboxErrStrMsg);
if (errMessage ! = NULL) {OSMboxPost (MboxErrStrMsg, errMessage);}
else {
/*Codes for again sending the Port B string of characters to task for transmission;
he message is tested to see that it has no invalid character or that it never
exceeds the limits of its size..*/
OSMboxPost (MboxStrMsg, message);
/* Release the flag to a task waiting for sending the output at port B */
OSSemPost (SemFlag2);
};
41. OSTimeDlyResume (Task_ErrPriority); /* Let delayed higher priority task err
resume. */
/* To exit the infinite loop at higher priority assigned task to let the lower
```

```
riority task run, call the OS delay function for wait of 1 ms (one OS timer tick.
This is the method  to let the other task of lower priority execute Port B sending
the characters. */
OSTimeDly (1);
}; /* End of while loop*/
42. }/ * End of the Task_OutPortB function */
/
*****************************************************************************/
43. /* Start of Task_SendPortB codes */
static void Task_SendPortB (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that
xecute only once */
.
.
44. while (1) { /* Start the infinite loop */
45. /* Take the event flag SemFlag2 > 0 and decrement it */
OSSemPend (SemFlag2, 0, &SemErrPointer);
46. /* Wait for error free message from Port B. If available, retrieve it. */
OSMboxPend (MboxStrMsg, 0, & MboxErr);
47. /* Codes for sending the valid message to a memory buffer where it is saved
r to a network for transmission */
.
.
48. OSTimeDlyResume (Task_OutPortBPriority); /* Resume Delayed Task_OutPortB
*/
}; /* End of while loop*/
49. }/ * End of the Task_SendPortB function */
/
*****************************************************************************/
50. /* Start of Task_ErrSR codes */
static void Task_ErrSR (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that
xecute only once */
unsigned char [ ] errMessage; /* Declare error message pointer. */
51. while (1) { /* Start an infinite while-loop. */
/* Take the event flag SemFlag2 > 0 and decrement it */
OSSemPend (SemFlag2, 0, &SemErrPointer);
52. /* Check for Mailbox Error Message available (not NULL) */
errMessage = OSMboxAccept (MboxErrStrMsg);
53. /* Codes for the action on error message. */
if (strcmp (errMessage, "Invalid Character Found") == 0) {
/* Codes for the actions that need to be taken when invalid characters are found.
For example, codes for displaying "Invalid Number Dialed. Dial Again" on an LCD. */
.
.
};
if (strcmp (errMessage, " Characters in the message exceeded the limit declared
at the task for read. ")
==0) {
/* Codes for actions needed on limit exceeded. Codes for displaying on an LCD
"Message too long to
Accept. Dial again". */
```

```
.
.
}
OSTimeDlyResume (Task_ ReadPortAPriority); /* Resume the task Port A Read
54. }; /* End of while loop*/
55. } / * End of the Task_ErrSR function */
/
*****************************************************************************/
```

10.2.7 队列相关函数

在示例 10-18 中，我们已经看到信号量如何发送 16 位 IPC 消息。在示例 10-19 中我们可以使用邮箱传送较长的消息的指针。而使用队列，可以将来自任务的消息指针组成数组。任务可以把消息指针送进队列，如果是队列则放到后面，如果是堆栈则放到前面。因而，任务可以在传送优先级消息时插入一个特定的消息，用于删除先进先出(FIFO)模式或者后进先出(LIFO)模式(提示：就将元素插入队列中可用的模式而言，IPC 队列不同于数据结构队列(5.4.2 节))。

参见图 8-5，MUCOS 支持指针数组队列。首先有一指针**Qtop 指向消息指针队列的头，还有两个指针*QfrontPointer 和*QbackPointer，分别用于发送和检索消息的指针。用于任务 IPC 的 MUCOS 队列函数如表 10-7 所示。MUCOS 允许多达 256 个消息指针进入队列(也就是说，队列长度可以是 256)。MUCOS 不像信号量、邮箱或队列那样分别规定堆栈 ECB，但是可能会有另外的规定。函数 OSPostFront 启用消息指针数组(参见 5.6 节)上的堆栈操作。该函数启用插入操作，以便等待任务可以对消息指针进行 LIFO 检索，然后对优先级消息进行检索。

表 10-7 任务间通信的队列函数

系统时钟函数和服务的原型定义	返回和传递的参数	何时调用 OS 函数
OS_Event OSQCreate(void **QTop, unsigned byte qSize)	R 和 R	OS 创建队列 ECB 时调用该函数。该函数为 QTop 处的队列创建并初始化一个指针数组。队列可以达到最大长度 qSize。QTop 应该指向顶端(数组的第 0 个元素)。ECB 指在 QmsgPointer 处
unsigned byte *OSQFlush	Void 和 S	参见文中的内容
void *OSQPend(OS_Event *QMsgPointer, unsigned short timeout, unsigned byte *Qerr)	T 和 T	删除队列中所有已经被发送的消息时调用该函数。该函数检查队列在 QMsgPointer 处是否有一个待定消息(ECB 中队列的头指针没有指向 NULL)。然后函数返回 ECB 中所有队列头指针和队列尾指针之间的消息指针。在返回时，错误代码和 QMsgPointer 将指向 NULL

(续表)

系统时钟函数和服务的原型定义	返回和传递的参数	何时调用 OS 函数
unsigned byte OSQPost (OS_EVENT *QMsgPointer, void *QMsg)	U 和 U	该函数将消息指针 QMsg 发送给队列后面的 QMsgPointer。该消息是 ECB 中队列尾指针的内容
unsigned byte OSQFront (OS_EVENT *QMsg Pointer, void *QMsg)	V 和 V	该函数发送 QMsg 指针给队列中的 QMsgPointer。该指针指向 ECB 中队列的头指针，QMsg 的指针存储了压入堆栈的其他消息指针*
unsigned byte OSQQuery (OS_EVENT *QMsg Pointer, OS_Q_DATA *QData)	X 和 X	获取队列消息信息和错误信息时调用该函数

第 2 列参见文中相应的解释。

*根据消息优先级选择使用 OSQPostFront 或者 OSQPost 函数。如果消息有较高的优先级，就在前面发送该消息；否则按惯例放进队列。我们可以使用 post-front 和 post 函数建立一个队列，队列中存储消息指针数组，并按优先级进行排序。所以该队列被称作优先级排序队列。OSQPostFront 和后面 OSQPend 的使用间接启用了消息指针检索的 LIFO 模式。

(1) 为 IPC 创建队列

OS_Event QMsgPointer = OSQCreate (void **QTop, unsigned byte qSize)用于为 QTop 创建一个 OS 的 ECB，队列是 QMsgPointer 所指的一个指针数组。该数组长度的最大值可以声明为 256(第 0~256 个元素)(参见 5.5.1 节)。最初这个创建的指针数组指向 NULL。

传递 R 的任务参数：**QTop 作为 void 数组的指针传递。“qSize”是这个数组的长度(队列可以在读取之前插入，消息指针的个数在 0~255 之间)。

返回 R：函数 OSQCreate ( )返回指向分配给队列的 ECB 的指针。它最初是 void 数组。如果没有 ECB 可用，则返回 NULL。

示例 10-20 解释了 OSQCreate 函数的使用。

(2) 等待队列中的 IPC 消息

函数 void *OSQPend(OS_Event *QMsgPointer, unsigned short timeout, unsigned byte *Querry)检查队列是否在 ECB QMsgPointer 指向的位置有待定的消息(QMsgPointer 不指向 NULL)。在 QMsgPointer 定义的队列的 ECB 中，消息指针指向队列头。如果没有消息待定(直到消息被接收，或者等待周期在 RTOS 定时器的超时节拍之后结束，这里的等待周期是通过参数 timeOut 传递的)。ECB 中来自指针的队列将在返回消息指针之后，随即加 1 指向下一条消息。

返回 void：该函数返回一个队列 ECB 指针。返回情形如下：(i)如果队列消息检索成功，则返回 OS_NO_ERR。(ii)如果队列在 timeOut 定义的节拍期间没有获得消息，则返回 OS_TIMEOUT。(iii)如果该函数调用来自 ISR，则返回 OS_ERR_PEND_ISR。(iv)如果 *QMsgPointer 没有指向队列消息，则返回 OS_ERR_EVENT_TYPE。

传递 S 的任务参数：(i)OS_Event *QbackPointer 作为指针传递给和队列相关的 ECB。(ii)

它传递 16 位无符号整数参数 timeOut。该函数在延迟等于(timeOut-1)个 RTOS 系统时钟的计数输入(节拍)时，恢复执行任务。(iii)传递*err，持有错误代码的指针。

示例 10-20 解释了 OSQPend 函数的使用。

(3) 清空队列并删除所有消息指针

函数 unsigned byte *OSQFlush (OS_EVENT *QMsgPointer) 检查队列是否在 QMsgPointer 指向的位置有待定的消息(ECB 中队列头指针不指向 NULL)。该函数必须返回 ECB 中所有队列头指针和尾指针之间的消息指针。它返回一个错误代码和 ECB 中的 QMsgPointer。它们当前指向 NULL。

传递 T 的任务参数：OS_Event *QMsgPointer 作为指针传递给和队列相关的 ECB。

返回 T：函数 OSQFlush( )返回如下错误代码中的一个：(i)如果消息队列刷新成功，则返回 OS_NO_ERR。(ii)如果指针没有指向消息队列，则返回 OS_ERR_EVENT_TYPE。

(4) 发送消息指针到队列

函数 unsigned byte OSQPost(OS_EVENT *QMsgPointer, void *QMsg)发送消息*QMsg 的指针。消息指针*QMsg 在 ECB 中为*QMsgPointer 存储队列尾指针。

传递 U 的任务参数：OS_Event *QMsgPointer 作为指针传递给和队列尾相关的 ECB。消息指针*QMsg 作为消息被传递。

返回 U：函数 OSQPost( )如下 3 个错误代码中的一个：(i)如果队列发送信号成功，则返回 OS_NO_ERR。(ii)如果*QbackPointer 不是指向队列，则返回 OS_ERR_EVENT_TYPE。(iii)如果队列消息不能发送(如果 QSize 已经是 255，则不能再长)，则返回 OS_Q_FULL。

示例 10-20 解释了 OSQPost 函数的使用。

(5) 发送消息指针并将其插到队列头

函数 unsigned byte OSQPostFront (OS_EVENT *QMsgPointer, void *QMsg)将 QMsg 指针到发送到队列中的 QMsgPointer，但是，它在 ECB 中队列的头指针处；在这里 QMsg 的指针正在进行存储操作，同时将其他消息指针压回队列。

传递 V 的任务参数：OS_EVENT *QMsgPointer 作为指针传递给和队列相关的 ECB。第二个参数是消息 QMsg 的地址，它是队列的头地址。

返回 V：函数 OSQPostFront( )返回如下错误代码中的一个：(i)如果队列中的消息放置成功，则返回 OS_NO_ERR。(ii)如果指针*QbackPointer 不是指向消息队列，则返回 OS_ERR_EVENT_TYPE。(iii)如果 qSize 声明为 n，并且队列有 n 个消息正在等待读取，则返回 OS_Q_FULL。

示例 10-20 解释了 OSQPostFront 函数的用法。

(6) 查找队列 ECB 的消息和错误信息的查询操作

函数 unsigned byte OSQQuery (OS_EVENT *QMsgPointer, OS_Q_DATA *QData)检查队列数据，并将其放到 QData 所指位置。该函数也会从 ECB 中 QMsgPointer 所指位置查找错误信息参数，OS_NO_ERR 和 OS_ERR_EVENT_TYPE。

传递 X 的任务参数：函数 OSQQuery 传递(i)*QMsgPointer ECB 所指位置的队列指针和(ii)*QData 所指位置数据结构的指针。

返回 X：函数 OSQQuery 返回错误代码情形如下：(i)如果查询成功，则返回 OS_NO_ERR。(ii)如果*QMsgPointer 不指向队列消息，则返回 OS_ERR_EVENT_TYPE。

示例 10-20

任务 Task_ReadPortA 接收字符，并将它们放到队列 QMsg 中。该任务使用 OSQPost 向另外一个正在等待(被阻塞)这些字符串的任务发送一个 IPC。其优点之一是，只要消息就绪，就可以被其他的任务使用，不需要等待整个消息字符串传送完毕，如示例 10-10 中所示。邮箱中只允许有一条消息或者一个消息指针。队列中允许有任意数目的消息，直到队列被塞满。队列满的意思是指，消息总长度达到了为队列定义的最大数组长度。它的另外一个优点是，端口 A 没有必要是正在发送字符或字节的端口，但可以是 NIC(网络接口卡)或者任何其他发送字、帧、或者消息段的输入设备，这些字、帧、或者消息段需要按顺序发送给其他等待的任务。另外，Task_Err 发送的任何错误消息也可以作为优先级消息送进相同的队列，这样代码就得到了简化。相对于邮箱或者信号量的重要区别是，为队列声明 ECB 时也声明指针数组，而在邮箱或者队列中，只声明了 ECB。

不仅 Task_ReadPortA 任务，而且 Task_Err 任务，都会在检测到无效字符，或者字符长度超过限制时，发送错误消息给 Task_MessagePortA。OSQPostFront 的用途是作为优先级消息发送错误信息。OSQPend 用于等待消息或者错误消息字符串的 IPC。以上这些操作步骤如下：

步骤 a：任务 Task_CharCheck 检查端口 A 字符的可用性(例如，如果在端口 A 用于发送字符的键按下，则该任务被激活)。如果字符不可用，则等待。这里，由于 IPC 仅仅是针对事件标志，所以使用如同示例 10-16 中的信号量 SemFlag1 就足够了。

步骤 b：任务 Task_Read_Port_A 等待 SemFlag1，并执行代码，将字符累加到数组中以获得字符串 str 的代码。

步骤 c：任务 Task_Err 检查在端口 A 中读到的每个字符或消息，如果字符无效，则发送字符串 errStr 到一般的消息队列中。例如，在读取电话号码的情况下，步骤 b 中读到的字符或消息不是数字。这里，由于 IPC 消息只是一个 8 位的字符，使用 IPC 邮箱是没有必要的，因而在步骤 b 和步骤 c 之间使用信号量值 SemVal 就足够了。

步骤 d：任务 Task_MessagePortA 等待指针数组形式的字符或者消息。队列用作步骤 b 和步骤 c 之间的 IPC，以及步骤 c 和步骤 d 之间的 IPC。

步骤 e：任务 Task_ErrLogins 等待队列中作为优先级消息的头指针发送的错误信息。

步骤 f：任务 Task_ServiceMessage 根据 Task_MessagePortA 处检索到的消息等待服务。

这个示例显示了步骤 a 和 b 如何通过 IPC SemFlag1 实现同步，步骤 b 和 c 的任务如何使用 IPC SemVal 实现同步，以及步骤 b~d 和步骤 c、d 如何使用 MUCOS 的邮箱函数实现同步。

```
1. /* Codes are the same as in Step 1 Example 10.19, except that the statements
are shown in bold for the mailbox. The mailbox-related statements are replaced
by the queue-related messages. */
.
.
2. /* Preprocessor definitions for maximum number of inter-process events to
let the MUCOS allocate memory for the Event Control Blocks */
#define OS_MAX_EVENTS 12/* Let maximum IPC events be 12 */
#define OS_SEM_EN 1/* Enable inclusion of semaphore functions in applications
using MUCOS */
#define OS_Q_EN 1/* Enable inclusion of queue functions in applications using
MUCOS */
/* End of preprocessor commands */
```

```
3. /* Prototype definitions for six tasks for steps a to e above. */
static void Task_CharCheck (void *taskPointer);
static void Task_ReadPortA (void *taskPointer);
static void Task_Err (void *taskPointer);
static void Task_MessagePortA (void *taskPointer);
static void Task_ServiceMessage (void *taskPointer);
static void Task_ErrLogins (*taskPointer);
4. /* Definitions for six task stacks */
static OS_STK Task_CharCheckStack [Task_CharCheckStackSize];
static OS_STK Task_ReadPortAStack [Task_ReadPortAStackSize];
static OS_STK Task_ErrStack [Task_ErrStackSize];
static OS_STK Task_MessagePortAStack [Task_MessagePortAStackSize];
static OS_STK Task_ServiceMessageStack [Task_ServiceMessageStackSize];
static OS_STK Task_ErrLoginsStack [Task_ErrLoginsStackSize];
5. /* Definitions for six task stack size */
#define Task_CharCheckStackSize 100 /* Define task 1 stack size*/
#define Task_ReadPortAStackSize 100 /* Define task 2 stack size*/
#define Task_ErrStackSize 100 /* Define task 3 stack size*/
#define Task_MessagePortAStackSize 100 /* Define task 4 stack size*/
#define Task_ErrLoginsStackSize 100 /* Define task 6 stack size*/
#define Task_ServiceMessageStackSize 100 /* Define task 5 stack size*/
6. /* Definitions for six task priorities. */
#define Task_CharCheckPriority 5 /* Define task 1 priority */
#define Task_ ReadPortAPriority 6 /* Define task 2 priority */
#define Task_ErrPriority 7 /* Define task 3 priority */
#define Task_MessagePortAPriority 8 /* Define task 4 priority */
#define Task_ServiceMessagePriority 9 /* Define task 5 priority */
#define Task_ErrLoginsPriority 10 /* Define task 6 priority */
7. /* Prototype definitions for the semaphores */
OS_EVENT *SemFlag1; /* Needed when using the semaphore as a flag for inter-process
com-munication between port status check task, Task_CharCheck and port read task,
Task_ReadPortA Port A read task has to wait for semaphore till port A
interrupts.*/
OS_EVENT *SemFlag2; /* Needed when using the semaphore as flag for inter-process
communication between Task_MessagePortA task and sending task
Task_ServiceMessage. */
OS_EVENT *SemCountLimitFlag; /* Needed when using the semaphore as flag for
reaching the limits of semaphore count in the inter-process communication between
port read and port read decipher task. Port reading has to wait for port read
*/
OS_EVENT *SemKey; /* Needed when using the semaphore as resource key by
Task_ReadPortA and Task_Err */
OS_EVENT *SemVal; /* Needed when using the semaphore for passing the 16-bit
message between steps b and c. */
OS_EVENT *SemCount; /* Needed when using the semaphore for passing the counts
as the number of messages read as a 16-bit message between tasks at the steps
b and d. */
8. /* Prototype for queue ECBs and for message-pointers array at a queue. */
OS_EVENT *QMsgPointer; /* Needed when using mailbox message between steps b and
d and steps d and e */
void *QMsgPointer [QMessagesSize]; /* Let the maximum number of message-pointers
at the queue be QMessagesSize. */
```

```
OS_EVENT *QErrMsgPointer; /* Needed when using a mailbox message between steps
c and d */
void *QErrMsgPointer [QErrMessagesSize]; /* Let the maximum number of error
message-pointers at the queue be QErrMessagesSize. */
9. /* Define both queues array sizes. */
#define QMessagesSize = 64; /* Define size of message-pointer queue when full
*/
#define QErrMessagesSize = 16; /* Define size of error message-pointer queue
when full */
10. /* Codes as per codes in Example 4.5 except the main function codes. These
are for reading from Port A and storing a character or message at a queue or
buffer. Alternatively, modify the code for reading from a port at NIC or any
other device or peripheral. */
.
.
11. /* Codes as per Example 10.7, steps 3 to 8. However, before the 'OSStart
( ); ', the semaphore and mailbox must be created and initialised as under: */
SemFlag1 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using
it as an event flag*/
SemFlag2 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using
it as an event flag*/
SemCountLimitFlag = OSSemCreate (0) /* Declare initial value of semaphore = 0
as an event flag*/
SemKey = OSSemCreate (1) /* Declare initial value of semaphore = 1 for using
it as a resource key*/
SemVal= OSSemCreate (0) /* Declare initial value of semaphore character or
message be = '0' */
12. / * Declare initial count as 0 as a counter that gives the number of times
a task, which sends into a buffer that stores a character or message stream,
ran minus the number of times the task which used the character or message from
the stream ran from the buffer */
SemCount = OSSemCreate (0);
13/* Create Two queues for the tasks, one general purpose queue and another for
error logins only after selecting error messages from the general queue. */
/* Define a top of the message pointer array. QMsgPointer points to top of the
Messages to start with.
*/
QMsgPointer = OSQCreate (&QMsg [0], QMessagesSize);
/* Define a top of the message pointer array. QMsgPointer points to top of the
Messages to start with.
*/
QErrMsgPointer = OSQCreate (&QErrMsg [0], QErrMessagesSize); /* Needed when
using mailbox message between steps b and c to pass a string message pointer*/
14.
.
.
15. /* Create six Tasks as per Step 3 defining by six task identities,
Task_CharCheck, Task_ReadPortA,Task_Err, Task_MessagePortA and Task_ServiceMessage
and the stack sizes, other TCB parameters.
*/
OSTaskCreate (Task_CharCheck, void (*) 0, (void *) & Task_CharCheckStack
[Task_CharCheckStackSize], Task_CharCheckPriority);
```

```
OSTaskCreate (Task_ReadPortA, void (*) 0, (void *) & Task_ReadPortAStack
[Task_ReadPortAStackSize], Task_ReadPortAPriority);
OSTaskCreate (Task_Err, void (*) 0, (void *) & Task_ErrStack [Task_ErrStackSize],
Task_ErrPriority);
OSTaskCreate (Task_MessagePortA, void (*) 0, (void *) & Task_MessagePortAStack
[Task_MessagesPortAStackSize], Task_MessagePortAPriority);
OSTaskCreate (Task_ServiceMessage, void (*) 0, (void *) &
Task_ServiceMessageStack
[Task_ServiceMessageStackSize], Task_ServiceMessagePriority);
OSTaskCreate (Task_ErrLogins, void (*) 0, (void *) & Task_ErrLoginsStack
[Task_ErrLoginsStackSize], Task_ErrLoginsPriority);
11. /* Codes same as at Steps 9 to 17 in Example 10.16 */
.
.
16. /* Refer to Example 4.5 Wait till an interrupt occurs and sets STAF*/
while (STAF != 1) { };
/* Execute interrupt service routine for port A*/
17. if (STAF == 1) {OSSemPost (SemFlag1);} /* Post semaphore to a task waiting
for the a read at Port A */
enable_PortA_Intr ( ); /* Prepare for another interrupt from port A*/
OSTimeDly (3); /* Let a low priority task execute, as next character or message
at the Port A will take
time. */
}; /* End of while loop*/
}/ * End of the Task_CharCheck function */
/
**********************************************************************/
18. /* The codes for the Task_ReadPortA redefined to use the key, flag and 16-bit
value and mailbox*/
static void Task_ReadPortA (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that
execute only once */
unsigned char *portAdata;
static void PortA_ISR_Input (*portAdata);
19. /* Start an infinite while-loop and Wait for SemFlag1 1 by OSSemPost function
of character or message availability check task */
while (1) {
OSSemPend (SemFlag1, 0, &SemErrPointer);
/* Take the key to not let the error task detect and block the task that also
needs SemKey for running state */
OSSemPend (SemKey, 0, &SemErrPointer);
20. /* Actions on maximum size exceeding the string buffer size */
if (*SemCount > = QMessagesSize) {
OSSemPost (SemCountLimitFlag); /* Post the SemCountLimitFlag */
OSSemPost (SemKey); /* Release the SemKey to let the next cycle of this loop
start */
/* To exit the infinite loop of this higher priority assigned task and to let
the lower priority task error detect run by a call to the OS delay function that
forces a wait of 1 ms (one OS timer tick) or until delay resume function executes.
This is the method to let the other task of lower priority execute the Port A
message error-detecting task */
OSTimeDly (1);
```

```
OSSemPend (SemKey, 0, &SemErrPointer); /* Take the SemKey */
SemCount =0; /* Reset the Semaphore counter to 0. */
};/* End of Codes for the action on reaching the queue array limit of the putting
the message pointers into the buffer */
/* Execute Port A interrupt service routine. */
PortA_ISR_Input (&portAdata); /* Remember that as soon as Port A is read STAF
will reset itself to reflect next interrupt status. */
21. /* Write the array element that returned as Port A data into the port A input
string */ OSQPost (QMsgPointer, &portAdata);
/*Let counting semaphore value increase after one character or message has been
put into the String, holding the character or message stream*/
OSSemPost (SemCount);
22. /*Let character or message pass as an IPC to Task_Err not as a mailbox message
but as a semaphore value. Note that we are not using mailbox as an IPC of a character
or message of 8 bits only. */
*SemVal = portAdata;
OSSemPost (SemVal);
23. OSTimeDly (1); /* Let a low priority Task_Err run in order to check the error,
if any, in the read byte */
243. /* Prepare for another interrupt from port A*/
enable_PortA_Intr ( );
OSTimeDlyResume (Task_CharCheckPriority);/* Resume the task that was delayed
to start
this task*/
25. OSSemPost (SemKey); /* Release the SemKey to let the next cycle of this while
loop start */
}; /* End of while loop*/
26. }/ * End of the Task_ReadPortA function */
/
*****************************************************************************/
27. /* Start of Task_Err codes */
static void Task_Err (void *taskPointer) { /* Initial assignments of the
variables and pre-infinite loop statements that execute only once */
/*Declare an error string for Step d error invalid message data found to the
mailbox. */
unsigned char [ ] ErrStr1 = "Invalid Message Data Found";
/* Declare an error string message for task at Step d when the limit exceeds.
*/
unsigned char [ ] ErrStr2 = "Array Size exceeded the Limit. Queue Full";
boolean invalid = false; /* Declare invalid variable 'false' and will be assigned
'true' when character or message read is found invalid. * /
28. while (1) { /* Start the infinite loop */
29. /* Take the key to not let Task_ReadPortA wait at least one cycle of the
current while loop*/
OSSemPend (SemKey, 0, &SemErrPointer);
30. /* Post message Limit of Message Exceeded to task at step d. */
If (SemCountLimitFlag = = 1) {OSQPostFront (QMsgPointer, ErrStr2);};
31. /* Codes for checking any invalid character or message in SemVal*/
OSSemAccept (SemVal);
.
.
32. /* Post as priority message queue message to task at step d if an invalid
```

```
message data is detected */
if (SemCountLimitFlag == 0 && invalid = = true) {OSQPostFront (QMsgPointer,
ErrStr1);};
33. /* Decrement and accept the SemCountLimitFlag if available (>0). There is
no requirement of task suspension. Next task must run whether countLimit is
exceeded or not. The only condition is that it must run if exceeded. Hence the
OSSemPend is not used here. */
OSSemAccept (SemCountLimitFlag);
34. /* Release the key to let a task, which was waiting for the port read, run*/
OSSemPost (SemKey);
OSTimeDlyResume (Task_ ReadPortAPriority); /* Resume the task Port A Read
}; /* End of while loop*/
35. } / * End of the Task_Err function */
/
*****************************************************************************/
36. /* The codes for the Task_MessagePortA */
static void Task_MessagePortA (void *taskPointer) {
37. /* Initial assignments of the variables and pre-infinite loop statements
that execute only once */ void * message;
38. while (1) { /* Start an infinite while-loop. */
39. /* Wait for Queue Message Pointer available (not NULL) */
&message = OSQPend (QMsgPointer, 0, &QErrPointer);
40. /* Find if the message has invalid character or the messages found as Error
Messages. Check for Mailbox Error Message available (not NULL) */
if (strcmp ((char *) message, "Invalid Message Data Found ")==0) {OSQPost
(QErrMsgPointer, mes-sage);};
if (strcmp ((char *) message , "Array Size exceeded the Limit. Queue Full") ==
0){OSQPost(QErrMsgPointer, message);};
/* To exit the infinite loop at the task that has been assigned a higher priority
and to let the lower priority task run, call the OS delay function for wait of
1 ms (one OS timer tick). This is the method to let the other task of lower priority
execute. */ OSTimeDly (1);
/* Release the flag to a task waiting for the servicing the messages received
at port A after filtering the error messages*/
OSQPost (QMsgPointer, message);
OSSemPost (SemFlag2);
};
41. OSTimeDlyResume (Task_ErrPriority); /* Let delayed higher priority task err
resume. */
}; /* End of while loop*/
42. }/ * End of the Task_MessagePortA function */
/
*****************************************************************************/
43. /* Start of Task_ServiceMessage codes */
static void Task_ServiceMessage (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that
execute only once */
void *message;
.
.
44. while (1) { /* Start the infinite loop */
45. /* Wait for valid message for queue from Port A. Take the event flag SemFlag2
```

```
> 0 and decrement it */
OSSemPend (SemFlag2, 0, &SemErrPointer);
46. /* Get the message. */
&message = OSQPend (QMsgPointer, 0, &QErrPointer);
47. /* Codes for servicing as per the valid message to a memory buffer for saving
or to a network
or to dial and transmit */
.
.
48. OSTimeDlyResume (Task_MessagePortAPriority); /* Resume Delayed
Task_OutPortB */ }; /* End of while loop*/
49. }/ * End of the Task_ServiceMessage function */
/
*************************************************************************/
50. /* Start of Task_ErrLogins codes */
static void Task_ErrLogins (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that
execute once only*/ void *errorLogged; /* Declare error message pointer. */
51. while (1) { /* Start an infinite while-loop. */
/* Take the event flag SemFlag2 > 0 and decrement it */
OSSemPend (SemFlag2, 0, &SemErrPointer);
52. /* Check for Mailbox Error Message available (not NULL) */
&errorLogged = OSQPend (QErrMsgPointer, 0, &QErrPointer);
53. /* Codes for the action as per the error logged in */
54. if (errorLogged = = " Invalid Message Data Found") {
/* Codes for actions needed on invalid character or message found. For example,
codes for displaying "Invalid Number Dialed. Dial Again" on an LCD. */
.
.
};
if (errorLogged = = " Array Size exceeded the Limit. Queue Full ") {
/* Codes for actions needed on limit exceeded. Codes for displaying on an LCD
"Message too long to Accept. Dial again". */
.
.
}
OSTimeDlyResume (Task_ ReadPortAPriority); /* Resume the task Port A Read
54. }; /* End of while loop*/
55. } / * End of the Task_ErrLogins function */
/
*************************************************************************/
```

10.3 VxWorks

对于复杂的嵌入式系统，有一种流行的 RTOS，即来自 WindRiver 的 VxWorks (http://www.wrs.com/)。VxWorks 是一种高性能、可扩展的 RTOS，它在目标处理器上执行。VxWorks RTOS 的设计具有层次性(参见 9.10 节)。

VxWorks 的通信选项非常多(参见 8.3 节)。它提供了以下服务：

(i) 多任务环境

(ii) 进程间通信(IPC)

(iii) 任务和 ISR 中使用(a)事件标志的同步实现，(b)使用资源键的互斥访问(mutex)和(c)使用 3 类信号量的计数机制

(iv) 使用 POSIX 标准信号量的同步实现和其他 IPC

(v) 任务和中断服务函数(ISR)中独立的上下文(有独立 TCB 和 ISR 的各个任务之间有公用堆栈)

(vi) Watchdog 定时器

(vii) 使用管道的虚拟 I/O 设备

(viii) 虚拟存储器管理函数

VxWorks 为每个任务保存如下信息：

(i) OS 的控制信息，包括优先级、堆栈大小、状态和选项

(ii) 任务的 CPU 上下文，包括 PC、SP、CPU 注册信息和任务变量

另外，VxWorks IO 系统也提供了大部分 POSIX 1003.1b 标准接口。

VxWorks 还提供：

(a) 作为 I/O 虚拟设备用于进程间通信的管道驱动程序。

(b) 网络透明套接字。

(c) 共享存储器和 Ethernet 的网络驱动程序。

(d) 存储驻留文件的 RAM 磁盘驱动程序。

(e) 计算机系统的 SCSI、键盘、VGA 显示器、磁盘和并口，以及 HDD、软盘、磁带、键盘和显示器的驱动程序。

VxWorks I/O 系统也包括 POSIX 标准异步 I/O 和 UNIX 标准缓冲 I/O。它还为仿真(VxSim)(12.4.1 节)、软件逻辑分析器(WindView)、VxWorks 和 TCP/IP 网络系统之间的网络设施提供服务。对于许多其他设置，可以参考该产品附带提供的《VxWorks 编程指南》和《VxWorks 网络编程指南》。

10.3.1 基本特性

在复杂嵌入式系统的设计中，VxWorks 的重要特性总结如下：

(1) VxWorks 是可扩展 OS(只有一些必要的 OS 函数才会成为应用程序代码的一部分，因此就减少了它对存储器的需求)。运行时可配置特征给出了 VxWorks 中比较高的性能。任务服务、进程间通信等必需的函数必须在配置文件中预先定义，这些配置文件包含在用户代码中。由于并非所有函数都在调度程序中，因而会出现抢先延时减少。

(2) RTOS 层次包括定时器、信号、TCP/IP 套接字、排队函数库、NFS、RPC、Berkeley 端口和套接字(8.3.6 节)、管道(8.3.7 节)、Unix 兼容加载器、语言解释器、shell、用在 Unix 中的调试工具和连接加载器(这些和系统任务类似。当运行 ISR 时，调度程序运行这些服务。)

(3) 对于多任务，如 MUCOS，VxWorks 使用抢占式调度程序(9.6.4 节)。VxWorks 是一个带有 256 个优先级的抢占式调度程序，优先级从 0~255，它们在任务创建时指定。VxWorks 在时间片循环模式下也是可配置的(9.6.3 节)。当 CPU 访问变到较高优先级时，任务上下文迅速保存。相同优先级的不同任务在循环模式下执行。一组循环执行的任务中的每个任务都运行指定数量的系统节拍，并且在超时后变成任务组的最后一个任务。

(4) 在 VxWorks 中，我们既可以使用抢占式优先级调度，也可以使用时间片调度。它不像

POSIX 函数，在 POSIX 函数中可以使用 VxWorks 在时间片模式下调度特定的任务，而在抢占式调度模式下，它调度其他任务(提示：抢占式优先级调度和 POSIX FIFO 调度是相同的)。

(5) 参见示例 10-19 和示例 10-20。在 Task_CharCheck 中检查中断服务例程的中断标志，中断服务例程在 Task_ReadPortA 中被调用。VxWorks RTOS 独立调度 ISR，并有特定函数用于中断处理(参见 10.3.3(iii))。

(6) VxWorks 有系统级函数。这些函数由于 RTOS 初始化和启动、RTC 节拍(中断)初始化和 ISR 函数、连接中断向量的 ISR 和屏蔽函数。回顾 8.2 节。对于临界段，VxWorks 有中断禁用和启用函数，分别在进入和退出临界段时执行(由于基于信号量的待定消息不应该在 ISR 中被调用，所以信号量函数作为事件标志和计数，不应该被使用)。

(7) 如果某个任务正在等待来自另一个任务的消息，并通过使用任务删除函数删除消息，那么 RTOS 就禁止删除操作。

(8) VxWorks 有任务服务函数(表 10-9)。VxWorks 单独的任务创建(初始化)操作没有使任务进入活动任务的列表(5.7.3 节)。活动任务的意思是处于就绪、运行和等待(阻塞或待定)3 种状态之一的任务。因而，VxWorks 不仅有任务创建、运行、等待(阻塞或者待定，直到超时或者资源可用)、挂起(禁止任务执行)和恢复指向函数，而且有用于任务生成(在激活后接着创建)的函数。VxWorks 也包含任务待定连同挂起函数和带有超时限制的待定连同挂起的函数。VxWorks 还包含有状态和被继承的优先级的任务。10.3.4(5)描述了这些函数。

(9) VxWorks 有任务延迟函数和任务延迟连同挂起函数(参见 10.3.4 节)。

(10) 为了改善 RTOS 的性能，VxWorks 在存储器中对所有任务提供了一个共享地址。这有助于通过指针进行快速访问。没有必要给管道分配一块单独的存储空间。当然，由于可能有非法访问，所以这样处理带有一定风险。

(11) VxWorks 有共享存储器分配函数和有界环形缓冲区分配函数，用于在任务和 ISR 之间共享存储器和缓冲区。

(12) VxWorks 有进程间通信(IPC)函数，它们比 MUCOS 函数更加复杂。回顾前面的知识我们知道，MUCOS 有用于事件标志、资源获取键和计数信号量的信号量函数。前面我们还讲过信号量 SemKey 的使用方法，在 SemKey 中有 OSSemPend 和 OSSemPost 函数。在示例 10-17~示例 10-20 中，SemKey 被各种任务作为资源获取键使用。VxWorks 分别为 3 类信号量提供服务(POSIX IPC 是附加的)。

(13) 在临界段互斥访问，VxWorks 有一些特殊属性。当使用 VxWorks 时，我们对资源键使用互斥信号量。作为互斥使用的一类信号量有如下特殊属性：(a)一个任务可以被保护，不被任何其他任务删除。因而，在使用互斥信号量函数时，不可能发生无保护的删除。互斥创建可以使用包含删除保护的方法来完成。(b)只有正在通过互斥信号量获取资源键的任务可以释放(给出或者发送)键，没有其他任务可以释放它。这样就提供了互斥访问。(c)如果任务使用互斥获取键，可以防止优先级反转。高优先级任务的优先级指定现在可以继承，以至于在被中等优先级任务抢占的情况下，高优先级任务不会被阻塞(8.2.3 节)。这样就防止了优先级反转。

(14) 互斥锁可用于任务和中断，禁止中断或禁止抢占(任务切换)，以及使用互斥信号量的资源锁定。

(15) 回顾 8.2.2(iv)中的图 8-4(a)，了解用于锁定资源的 P 和 V 互斥信号量。VxWorks 也提供了 P 和 V 信号量函数。

(16) VxWorks 中的互斥锁和解锁函数不会导致优先级反转问题(8.1.2(v))。优先级首先是继

承，然后返回到初始位置。

(17) 不像 MUCOS，VxWorks 没有单独的邮箱函数，这使其区分于消息队列。VxWorks 消息可以排队。它支持发送可变长度的消息到队列中。MUCOS 队列函数只允许消息是一个指针，并且队列是消息指针的数组。MUCOS 消息可以插入，以至于它们的消息具有优先级时，采用 FIFO 先进先出)方法或者 LIFO(如果消息指针在前面发送，而不是在后面发送)方法检索。

(18) 除了队列之外，VxWorks 通过管道提供 IPC(8.3.5 节)，ISR 或者任务可以通过调用函数 write( )写进管道。任务可以通过使用 open( )和 read( )函数从管道中读取。VxWorks 管道是一个带有虚拟 I/O 设备函数的队列(参见 10.3.4(12))。

(19) VxWorks 的调度程序设计特征和 POSIX 1003.1b 是兼容的。POSIX 库可以包含到 VxWorks 中。

(20) 各种 RTOS 函数采用的定时方法和表 7-6 中给出的类似。

VxWorks 和核心库函数在头文件“vxWorks.h”和“kernelLib.h”中。系统和任务库函数在头文件“taskLib.h”和“sysLib.h”中。用于记录日志的库函数是“logLib.h”。

下面将描述这些重要的 VxWorks 函数。

10.3.2 系统库头文件中的任务管理库

每个任务划分为 8 种情形(状态)。其中的 4 种情形在 MUCOS 中是可用的。

(a) 挂起(创建之后的空闲状态或者执行被继承的情形)(参见示例 10-8 的步骤 12 中 FirstTask 任务的 OSTaskSuspend 函数，以及示例 10-16~示例 10-20 中的 FirstTask 代码)。

(b) 就绪(等待运行和在被调度程序调度时的 CPU 访问，但通过 IPC 传送的消息)。

(c) 待定(由于任务等待来自 IPC 或者资源的消息时被阻塞；只有 CPU 可以进一步处理)(这被称作阻塞情形。参见示例 10-16~示例 10-20 中的 OSSemPend、OSMboxPend 和 OSQPend 函数)。

(d) 延迟(休眠特定的时间间隔)(参见示例 10-16~示例 10-20 中 OSTimeDly 的使用)。

(e) 延迟＋挂起(如果任务在延迟期间没有被抢占调度，就被延迟，然后挂起)。

(f) 待定＋挂起(如果阻塞状态没有改变，就转为待定状态，然后挂起)。

(g) 待定＋延迟(在延迟一定时间间隔后转为待定状态，然后被抢占调度)。

(h) 待定＋挂起＋延迟(在延迟一定时间间隔后转为待定状态，然后被挂起)。

如果包含 sysLib.h 文件，那么任务库就会包含进来。库函数在表 10-8 和表 10-10 中给出。表 10-8 列出了任务状态转换的函数。表 10-9 给出了任务创建、命名和控制服务函数。

(1) 通过任务 TaskSpawn 函数创建和激活任务

用于任务创建和激活的函数是 taskSpawn( )。原型是 unsigned int taskId = taskSpawn(name, priority, options, stacksize, main, arg0, arg1,……, arg8, arg9)。分配存储器给堆栈和 TCB。taskID 所标识的任务将被分配大小为 stacksize 的堆栈，同时 arg0~arg9 被传递给堆栈。该函数也会被分配一个 TCB 指针，它指向任务执行的 main 函数的入口点。

回顾前面的知识，在 MUCOS 中，我们通过任务的优先级参数访问任务，比如 OSTimeDlyResume(Task_CharCheckPriority)。而在 VxWorks 中，我们通过任务的 ID，taskID 访问任务(参见下面小节中函数的 taskID 参数)。taskID 是一个 32 位的正数。另外，如果 taskID = 0 的任务被访问，VxWorks 就会认为调用任务正在被访问。

新任务 taskID 在生成时获得任务名、优先级、选项和堆栈大小等参数。如果使用 NULL

指针作为任务名的参数，那么按惯例，任务名的形式应该为 tN(字符 t 后接数字 N)。

- 函数“unsigned int taskIdSelf( )”返回调用任务的标识符。
- 按“unsigned int [] listTasks = taskIdListGet()”形式调用函数时，函数会返回数组所需的任务列表 ListTasks。
- 函数 taskIDVerify(taskId)核实任务 taskId 是否存在。

默认每个任务的最高优先级为 100。用户任务的优先级在 101~255 之间。最低优先级是指优先级值为 255(100 以下的优先级数用于系统级和调度程序级任务)。任务可以有其他的调用函数，例如，在 taskLib 中有 3 个这样的函数。调用函数可以使用函数“taskPriorityGet(tasked, &priority)”查找优先级。任务的优先级可以动态改变。函数“taskPriorityPut(tasked, newPriority)”将重新为 taskId 所标识的任务中的 newPriority 指定优先级(提示：在 POSIX 中优先级编号方案和 VxWorks 是相反的：数字越小，POSIX 优先级越低)。

可以在生成时定义的选项如下：

- VX_PRIVATE_ENV。其含义是任务必须在私有环境中执行，因为任务不是公共的。
- VX_NO_STACK_FILL。其含义是没有堆栈填满了 16 进制数 0xEE。
- VX_FP_TASK。其含义是任务执行时必须带有浮点处理器。
- VX_UNBREAKABLE。其含义是在任务执行阶段禁用断点(断点在调试阶段很有帮助)。选项名可以分别使用 16 进制数 0x80、0x100、0x8 和 0x2 代替。

多个选项作为参数形式给出，例如 VX_PRIVATE_ENV，VX_NO_STACK_FILL 在任务生成时都要选择。和优先级任务类似，选项也可以使用 taskOptionSet( )和 taskOptionGet( )函数重新指定(由于符号&在 C 语言中指的是接连变量的地址，所有在多重选项之间使用|符号)。

应该使用 stacksize 声明足够的堆栈空间。VxWorks 会将堆栈的一部分转换为缓冲区来保护堆栈不会溢出，这样可能导致不可预料的系统行为。为了正常启动，任务堆栈开始时填充字节 0xEE(当使用仿真器 VxSim 作为 VxWorks 应用程序时，VxSim 将附加的 8000 字节与 stacksize 相加。所以，用于仿真中断的堆栈变为可用)。库函数“unsigned int checkStack (tasked)”返回堆栈使用情况，它首先通过计算末端为 0xEE 的字节数目来查找未使用的堆栈区域，然后从 stackSize 中减掉该值。

参数 main 是主例程的地址。MUCOS，以及许多 RTOS 中，main( )是第一个被 RTOS 调用的函数。参见示例 10-7。main 函数用于创建第一次执行的任务 FirstTask。当 FirstTask 在后面执行时，它初始化系统定时器。然后该任务创建(也可以是激活)应用程序任务，并挂起自己，等待 OS 调度并运行创建的应用程序任务。在 VxWorks 中，可能使用模拟的 main ( )函数，即 schedule( )。

当使用 VxWorks 时，不像 Unix 操作系统或者 MUCOS( )，所有任务作为对等体生成(就是说，每个任务是独立的，并且没有任务调用或者生成其他任务)。这意味着它和示例 10-7 中的 FirstTask 类似：启动任务第一次不需要生成。启动任务生成子任务，然后挂起自己，防止被调度。在父任务中的子任务，只有当父任务在并发处理子任务的服务任务时才生成。

arg0~arg9 这 10 个参数可以送进主例程。这些参数给出了启动参数。

需要注意的一个要点是，MUCOS OSTaskCreate 创建任务，同时也激活它(将它放到即将被调度的任务列表中)。这些函数在 VxWorks 中是独立的。taskInit( )和 taskActivate( )也可以代替

taskSpawn 函数单独使用。但是，除非在应用程序中，创建操作需要更大的控制权，然后是激活操作，我们都应该使用 taskSpawn 函数。

示例 10-23 步骤 4 将解释 taskSpawn 函数的用法。

(2) 任务挂起和恢复执行的函数

函数 taskSuspend (taskId)禁止 taskId 所标识的任务执行。函数 taskResume(taskId)恢复执行 taskId 所标识的任务。函数 taskRestart(taskId)首先结束任务，然后使用它初始指定的参数进行任务生成。该函数使用在这种特定的情形。优先级可以在结束任务和生成任务之间重新指定，这时初始的优先级将得到恢复。同样，启动参数也必须得到恢复。

(3) 删除和删除保护

函数 taskDelete (taskeId)不仅永久禁止了任务的执行，而且取消了任务堆栈和 TCB 的存储块分配，从而删除任务，释放存储器。被删除的是由参数 taskId 所标识的任务。函数"exit (code)"删除任务自身，但将代码存储在一个 TCB 区域 exitcode 中。调试器可以使用这些代码检查 TCB。

如果任务有资源键，它就不应该被删除，这是因为键可能永远不会被释放。保护措施可用于调用任务，它是通过在进入临界段之前调用函数 taskSafe ( )和在临界段的结尾调用函数 taskUnsafe( )来实现的。如果使用互斥信号量，则可以在创建的时候选择选项 SEM_DELETE_SAFT(参见 10.3.4 节)。

为什么要使用任务删除或者退出函数呢？这是因为可能会出现多次系统资源被回收并重复使用；对于特定的应用程序，存储器资源可能不足。TCB 和堆栈是惟一可以自动回收的资源。内核没有针对由其他任务生成的任务的节约措施。每个任务都应该自己执行如下的代码：

- 存储器释放。
- 确保等待任务得到了期望的 IPC。
- 关闭之前打开的文件。
- 当父任务执行 exit ( )函数时删除子任务。

(4) 延迟任务，让优先级较低的任务得到访问权

函数"int sysClkRateGet ( )"返回频率(系统每秒产生的节拍和 RTC 中断数)。所以若要延迟 0.25 秒，就使用函数 taskDelay (sysClkRateGet( )/4)。回顾 OSTimeDly 和后面 MUCOS 中用于恢复任务执行的 OSTimeDly(参见示例 10-16~示例 10-20)，它让优先级较低的任务运行。VxWorks taskDelay(NO_Wait)允许相同优先级或优先级更低的其他任务运行(因为超时时间是 0)。在该函数之后运行的其他优先级任务中，不需要恢复执行任何被延迟的任务。函数 nanosleep(1000000)延迟任务 1000 微秒，它是 POSIX 函数，其整型参数定义了延迟(休眠)的纳秒数。

表 10-8 任务状态转换函数

函　数	当前状态	下一状态	在调用之前的函数调用或前一状态
taskResume ()	挂起	延迟或待定	taskInit( )(任务必须已经初始化为空闲状态)
TaskResume ()或 taskActivate ()	挂起	就绪	TaskInit ( )
taskSuspend ( )	延迟	挂起	TaskSpawn ()或 askActivate ()
taskSuspend () 超时后	延迟	就绪	挂起

(续表)

函　数	当前状态	下一状态	在调用之前的函数调用或前一状态
taskSuspend ( ) 等待资源后	待定	挂起	就绪
semGive ( )或 msgQSend ( )	待定	就绪	挂起
semTake ()或 msgQReceive ()	就绪	待定	延迟或挂起
taskDelay ( )	就绪	延迟	待定或挂起
taskSuspend ()	就绪	挂起	延迟或待定
taskInit ( )	未知	挂起	
exit()[1]	挂起	结束	
taskDelete ( )[2]	挂起	删除	

1. 结束任务。
2. 结束任务并释放存储器。

表 10-9　任务创建、命名和控制函数

函　数	描　述
taskSafe ( )	保护调用函数不被删除
task Unsafe ( )	允许之前受保护的任务被删除
taskDelete ( )	删除任务
taskRestart ( )	由于之前任务返回错误，而重启(重新创建)正在运行的任务
taskActivate ( )	如果之前已经初始化，则激活任务
taskSpawn ( )	创建并激活
taskName ( )	返回和作为参数传递的 taskID 相关的任务名
taskNameTold ( )	返回和作为参数传递的 taskID 相关的 taskId
taskID Verify ( )	核实参数 taskID 所标识的任务是否可用
taskIDSelf( )	返回任务的 ID
taskIDListGet ( )	返回就绪任务 ID 的数组
taskInfoGet ( )	返回信息(任务参数)
taskRegsGet ( )	返回任务的注册信息
taskRegsSet ( )	设置任务的注册信息
TaskOptionsSet ( )	设置任务选项
taskOptionsGet ( )	返回先前定义的任务选项
taskIsSuspended ( )	检查任务是否是挂起状态
taskIsReady ( )	检查任务是否是就绪状态
taskTcb ( )	返回指向任务控制块的指针
taskPriority( )	返回任务优先级
taskLock( )	禁止其他任务执行，然后重新调度。[2]当任务正在运行时不会有优先级抢占
taskUnlock( )	允许其他任务执行，然后重新调度。[3]当任务正在运行时不会有优先级抢占

(续表)

函　数	描　述
taskPrioritySet( )	设置 0~255 的优先级
kernelTimeSlice(int numTicks)	在启用任务的循环运行之后定义每个任务的时间片。如果 numTicks = 50，则在 50 个 RTC 中断之后定义时间片循环周期

1. 表明给开始阶段定义的堆栈和控制块分配存储器。

2. 表明当任务正在运行时，调度程序不能阻塞该任务，即使需要调度优先级较高的任务。

3. 表明当任务正在运行，并且有优先级较高的任务需要调度时，调度程序可以阻塞该任务。

10.3.3 VxWorks 系统函数和系统任务

调度程序执行的第一个任务是 UsrRoot，其入口点是文件 install/Dir/target/config/all/usr/Config.C 的 usrRoot( )函数。它生成 VxWorks 工具和以下任务。在所有初始化完成之后中止根任务。任何根任务都可以被初始化或者中止。tLogTask 这组函数记录系统消息，不包括当前任务上下文 I/O。端口监控程序(一大组函数)支持任务级网络函数。异常处理函数是 tExcTask 组，它具有最高的优先级，不会被挂起、删除或者被指定较低的优先级。系统使用它报告在运行调度程序和任务时产生的异常情况。

具有特定目标的一组主要函数是 tWdbTask 系列函数。用户创建它们，为来自 Tornado 目标服务器的请求提供服务，这就是目标代理任务。

1. 系统时钟(RTC)和 Watchdog 定时器相关函数

VxWorks sysLib 是系统库函数，定义在头文件 kernelLib.h 中。以下是它提供的一些重要函数：

(1) 函数 sysClkDisable( )禁用系统时钟中断，sysClkEnable( )启用系统时钟中断。

(2) 函数 sysClkRate(TICK numTicks)设置每秒钟产生的节拍数。因而它会定义每秒钟产生的 RTC 系统中断数。函数 sysClkRateSet(1000)表示设置每 1/1000 秒(1 毫秒)产生一个 RTC 节拍。该函数应该在 main( )函数中，或者 FirstTask 启动的时候，或者作为启动函数被调用。vxAbsTicks 是一个 32 位的全局变量。变量 lower 在每个节拍和 vxAbsTicks 之后加 1。变量在每 2^{32} 个节拍后加 1。“TICK”使用 typedef 定义如下：

```
typedef struct{
unsigned long lower;
unsigned long upper;
}TICK
TICK vxAbsTicks;
/*Function"unsigned long tickGet( ) "returns vx.AbsTicks.lower。*/
```

(3) 函数 sysClkConnect( )连接一个 C 函数到系统时钟中断。

(4) 函数 sysAuxClkDisable( )禁用系统时钟中断，sysAuxClkEnable( )启用系统时钟中断。

(5) 函数 sysAuxClkRateSet(numTicks)每秒钟设置一个节拍作为辅助时钟。因此它定义了每秒钟产生的 RTC 中断的数量。函数 sysAuxClkRateSet( )返回系统每秒钟产生的辅助节拍(系统 RTC 中断)。

(6) 函数 sysAuxClkConnect( )连接一个 C 函数到系统，辅助 RTC 中断。

(7) 函数“WDOG_ID wdCreate( )”创建一个 Watchdog 定时器。语句“wdtID = wdCreate( );”将创建一个监督定时器 wdtID。有一个函数 STATUS wdStart(wdtID, delayNumTicks, wdtRoutine, wdtParameter)，创建的定时器启动时调用该函数。应该作为该函数的参数进行传递的参数如下：(i)wdtID，定义 Watchdog 定时器的标识符。(ii)delayNumTicks，在系统的 RTC 中断数等于 delayNumTicks 时，定时器产生中断信号。(iii)wdtRoutine，在每次中断时被调用的函数(不是任务或 ISR)。(iv)wdtParameter，传给 wdtRoutine 的参数。已经启动的 Watchdog 定时器 wdtID，在调用 STATUS wdCancel(wdtID)时中止；在调用 STATUS wdtDelete(wdtID)时释放存储器。

2. 为循环时间片调度定义时间片循环周期

函数 kernelTimeSlice(int numTicks)控制循环调度，打开时间片循环和关闭抢占式优先级调度。假设每毫秒有一个系统 RTC 节拍，那么 kernelTimeSlice(50)就会设置时间片循环周期为 50 秒。这是在为另外一个同等优先级的任务放弃 CPU 控制权之前，每个任务允许运行的时间。

我们使用代码#define TIMESLICE sysClkRateGet( )和 sysClkRateSet(1000)。当使用“sysClkRateSet(1000); if kernelTimeSlice(TIMESLICE) TIMESLICE = TIMESLICE 60;”语句时，时间片循环周期是 1000/60 毫秒。如果我们不设置时钟率，那么 numTicks 会设为每秒 60 个，并且每个任务时间片旋转周期将为 1 秒。

3. 中断处理函数

表 10-10 给出了中断服务相关函数。参见 4.1.2 节。内部硬件中断设备(中断源和中断源组)为每个设备自动生成一个中断向量地址 ISR_VECTRADDR。异常定义在用户软件中，所以对于异常，ISR_VECTRADDR 不会自动生成。这就必须通过 intVectSet( )函数定义。函数 intConnect( )连接 ISR_VECTADDR 到一个 C 函数来处理这个 ISR。设备驱动程序按如下方式使用该函数：按“int lock = intLock( );”这样使用的锁定函数将禁用中断，并返回一个整数 lock。在解锁函数中将使用这个整数作为参数来启用中断。解锁函数以“int Unlock(lock);”方式使用时将启用中断。

表 10-10 中断服务函数

函　数	描　述	函　数	描　述
intLock( )	禁用中断[1]	intUnlock( )	启用中断[2]
intVectSet( )	[3]设置中断向量	intCount( )	嵌套中断的计数值
intVecGet( )	获取中断向量	intVecBaseSet( )	设置中断向量的基地址
intVecBaseGet( )	获取中断向量基地址	intLevelSet( )	设置处理器的中断屏蔽级别
intContext( )	当调用函数是ISR时返回true	intConnect( )	连接一个C函数到中断向量

1 和 2 处的含义在文中有解释，参见 8.2 节。它可以在临界段区域作为最后一个选项来使用，这是因为它用于增加所有资源的中断延迟周期。

3. 仅仅针对“Expection”，对于硬件内部设备中断，中断向量是固定的，不能进行设置。

VxWorks 对于不同于任务设计的中断设计有如下规定：

(1) ISR 有最高优先级，可以抢夺任何正在运行的任务的控制权。由于内部设备事件(例如来自片上的定时器)和异常(用户定义的特定错误情形下的软件中断)产生 ISR 时会出现这种情况。

(2) ISR 禁止任务的执行，直到返回。

(3) ISR 不像任务那样执行，它没有正规的任务上下文，而只有一个特殊的“中断上下文”。

(4) 每个任务都有自己的 TCB(Task Control Block，任务控制块)，包括它自己的堆栈，除非如此，否则其他系统或者处理器的特殊体系结构是不允许的，而所有的 ISR 使用相同的中断堆栈。在如此特殊的体系结构中，代替中断服务支持函数，VxWorks 任务可以像示例 10-19 和示例 10-20 中一样用作 MUCOS 任务使用。CPUs 80x86 和 R6000 是允许使用这种特殊体系结构的例子，并且它使用任务堆栈来处理堆栈。

(5) ISR 在取信号量或者其他 IPC 时不应该等待。(ISR 不能使用 SemTake 函数)。ISR 不应该调用“melloc( )”用于存储器分配，因为这些函数使用了信号量。ISR 不应该使用互斥信号量。ISR 可以使用计数信号量。

(6) ISR 应该在存储器中写入请求数据(例如示例 10-19 中 STAF 的例子)，或者发送 IPC，或者非阻塞地写到消息队列中(8.3 节)，从而其代码变得简短，并且其大部分函数，以及不重要的和花费时间长的函数都在任务中执行。

(7) ISR 不应该使用浮点函数，因为浮点函数执行时会花费较长时间。可以将这些函数传递给在后面采用中断方式运行的任务。

4. 信号和中断处理函数

函数“void sigHandler (int sigNum);”为 sigNum 所标识的信号声明信号服务例程。信号服务例程按这样的方式注册信号：signal(sigNum, sigISR)。传递的参数是 sigNum(作为信号的标识符)和信号服务例程名 sigISR。函数 sigHandler 传递 sigNum 和附加的代码。sigHandler 代码和 sigHandler 相关联。指针*pSigCtx 和信号上下文相关联。信号上下文保存 PC、SP、注册信息等，类似一个 ISR 上下文。sigHandler 返回时恢复被保存的上下文信息。

令 sigISR 是一个服务信号中断的 C 函数，它的地址是 ISR_ADDR，信号使用 sigNum 来标识。函数“intConnect(I_NUM_TO_IVEC(sigNum), sigISR, sigArg)”将为 sigNum 所标识的信号连接信号中断服务例程 sigISR。ISR_ADDR I_NUM_TO_IVEC(sigNum)是一个使用参数 sigNum 从中断向量中查找程序计数器 PC 值的函数，并使用该 PC 值作为 ISR_ADDR。参数 sigNum 传递给 C 函数使用。

sigISR 可以调用如下函数：

(1) 调用“taskRestart( )”重启任务，同时生成 sigNum。重启过程将指定创建时的初始上下文。这时就恢复任务的初始 PC、SP、参数和选项。

(2) 调用“exit( )”中止任务，同时生成 sigNum。

(3) 调用 longjump( )。这会导致从某个指定的存储单元开始执行。这个位置是调用函数 setjump( )函数时保存的位置。

10.3.4 进程(任务)间通信函数

表 10-11 给出了进程间通信函数的列表及其描述。回顾 8.3.1 节、8.3.5 节和 8.3.6 节。该表没有给出用于信号、套接字和 RPC 的函数。信号使用 ISR 进行服务，它们单独在 10.3.3(4)中给出。

表 10-11 进程间通信函数

函 数	描 述
semBCreate( )	创建二进制信号量。[1]
semMCreate( )	创建互斥信号量。[1]
semCCreate( )	创建计数信号量。[1]
semDelete( )	删除信号量
semTake( )	获取信号量
semGive( )	释放信号量
semFlush( )	恢复执行所有等待的阻塞任务
msgQCreate( )	为消息分配并初始化队列
msgQDelete( )	删除消息队列，释放存储器
msgQSend( )	放进消息发送队列
msgQReceive( )	接收消息放进队列。[2]
pipeDevCreate( )	创建管道设备。[3]
select( )	任务等待几种消息，来自管道的、套接字的和串行 I/O 的消息

1. #意思是指定选项 SEM_Q_PRIORITY 作为排序依据，如果有许多等待任务，则按照这种顺序获取信号量。指定 SEM_Q_FIFO 用于定义获取信号量的 FIFO 模式。

2. !是指如果没有可用消息以及消息被任务读取时调用的任务块。根据选项参数，插入队列的消息可以以优先级作为排序参数进行排序，或者使用 FIFO 模式读取。

3. @意思是 "statement status = pipeDevCreate("/pipe/name" , max_msgs, max_length)"将以最大管道长度 max_length 字节中的最大 max_msgs 消息数量，创建指定名称的管道。如果消息不可用并且管道为空，并且任务试图读取管道时，任务就会发生阻塞。

VxWorks 提供了 3 种类型的信号量：二进制(标志)、十六进制和计数。互斥信号量也关心优先级反转问题。二进制信号量不禁用中断，它仅允许使用相关联的资源。任务必须取得信号量才能运行。通常，消息数量必须放进队列，在它可以运行之前作为等待任务。VxWorks 队列函数在库 msgQLib 中，用户在使用之前需要包含该库。对于两个任务之间的全双工通信，必须创建两个队列，每一个任务一个。mqPxLib 函数和 POSIX 1003.1b 是兼容的。表中的信号有以下含义。3 类 VxWorks 信号量和消息队列的详细描述在下文中给出。

(1) 为 IPC 创建的二进制信号量

函数 "SEM_ID semBCreate(options, initalState)" 创建一个由 SEM_ID 所指的 ECB。调用该函数时必须传递下面提出的两个选项之一：

参数传递：(1)option(s)：(i)可以被选择的一个选项是 SEM_Q_PRIORITY。(ii)另外一个选项是 SEM_Q_FIFO。假设在某个瞬间，多个任务处于阻塞状态，正在等待(待定)一个二进制信号量用于发送。等待任务可以采取两种方式取得信号量：(a)优先级比其他等待任务更高的任务首先取得信号量。这在使用 SEM_Q_PRIORITY 选项时是可行的。(b)在所有等待的任务中间，首先被阻塞并进入等待状态的任务首先取得信号量。这在使用 SEM_Q_FIFO 选项时是可行的。二进制信号量的初始状态通过参数 initialState 传递。如果使用二进制信号量作为事件标志，它就是 0。对于初始状态，有两个选项可以选择：SEM_FULL，选择时，所创建信号量的初始状态开始指定为 taken(已得到)；SEM_EMPTY，选择时，所创建信号量的初始状态开始指定为 not taken(未得到)(回顾 MUCOS 中信号量 SemKey 和 SemFlag 的使用)。当创建二进制信号量时，在 VxWorks 中主要使用 SEM_EMPTY，这是因为，在 VxWorks 中互斥信号量的规定比 MUCOS 中的 SemKey 更好。(2)initialState，在创建任务时定义其初始状态。

返回参数：函数 semBCreate( )返回指针*SEM_ID。如果分配给二进制信号量的 ECB 有错误，则返回 NULL。如果没有 ECB 可用，也返回 NULL。

示例 10-21 解释了 semBCreate 的用法。示例中假设 Task_CharCheck 检查端口 A 的字节可用性，接着，另外一个任务在等到来自 Task_CharCheck 的可用信号量时，读取这些字节。

示例 10-21

```
1. /* Include the VxWorks header file as well as semaphore functions from a library. */
2. # include "vxWorks.h"
3. # include "semLib.h"
4. # include "taskLib.h"
5. 2. /* Task parameters declarations */
6. .
7. .
8. 3. /* Declare a binary semaphore to be used as flag. */
SEM_ID semBCharCheckFlagID;
9. /* Create the binary semaphore and pass the options chosen selected to it.
*/
semBCharCheckFlagID = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY);
.
.
10. /* Task creation codes */
11. /* Codes for Task_CharCheck*/
.
.
12. /* At the end, make the binary semaphore full (taken) and available like
SemKey set as 1 in MUCOS. Refer to Section 10.3.4 for an understanding of the
semGive function. */
13. semGive (semBCharCheckFlagID);
14. /* Other remaining codes for the task. */
15. .
16. .
17. /* End of Task CharCheck Codes */
```

(2) 等待 IPC 用于释放二进制或者其他类型的信号量，或者在释放后检查 IPC 的可用性[C]函数“STATUS semTake(semId, timeOut)”让任务等待，直到释放发送二进制或者其他类型

信号量的事件。直到 SemId 被任务发送或者超时才停止等待，不管哪种情况发生，都会中止等待过程。semTake(semBCharCheckFlagID, WAIT_FOREVER)是该函数的一个使用范例。WAIT_FOREVER 是指 timeout = -1，等待时间没有限制。semTake 类似 MUCOS 中的 OSSemPend 函数。

参数传递：(1)semID 是挂起任务等待的信号量。(2)timeOut 是超时期限。可供选择的一个选项是 WAIT_FOREVER，如果 SemID 的发送必须等待，则选择这一项。另一个是 NO_WAIT。回顾 MUCOS 中的 OSSemAccept 函数。无论何时，该任务被调度程序调度或者该函数被调用，都要取得 SemID 所标识的信号量。第三个选项是 NO_Wait，它仅用于检查信号量的可用性。如果取得的信号量可用，执行该函数，否则就运行其他后续代码(这和 MUCOS 中的 OSSemAccept( )函数是等价的)。例如，如果出现错误消息，就没有必要等待，特别是对于信号量，只需要检查。

返回参数：函数 semTake( )返回 STATUS。当成功获取 semID 时，返回 STATUS = OK，否则出现错误时，返回“ERROR”。信号量 Id 在超时的情况下也会变成无效。在 semTake( )函数取消阻塞任务时，它再次变成可用(空或者未得到)。

想了解 semTake 的用法，请参考示例 8-22。

(3) 在二进制、互斥或者计数信号量 释放(发送)之后发送 IPC

函数“STATUS semGive(semId)”让任务发送和释放二进制或者其他类型的信号量。释放信号量后，任务就可以取消阻塞。至于取消阻塞哪个任务，这依赖于创建信号量时定义的选项。取消阻塞可以依据 SEM_Q_FIFO 或者 SEM_Q_PRIORTIY 选项。它的一种使用范例是示例 10-21 步骤 13 中的 semGive(semBCharCheckFlagID)。

参数传递：semID，该任务或者其他任务有一个等待该参数的过程。

返回参数：函数 semGive( )返回 STATUS。当成功获取 semID 时，它返回 STATUS = OK，否则出现 semID 无效的错误时，返回“ERROR”。

若要了解 semGive( )的用法，请参见示例 10-22。

(4) 在下一次发送之前多次获取信号量，直到信号量不可用

函数 STATUS semFlush(semFlagID)用于刷新。该函数在下一次发送之前多次获取信号量，直到信号量不可用。它使任何等待的任务不再等待，不仅取消阻塞调用任务，而且取消阻塞所有等待获取信号量 semFlagID 的其他任务。

参数传递：semFlagID 在和信号量关联的 ECB 中作为 SEM_ID 指针传递。

返回参数：函数 semFlush( )返回 STATUS，并置信号量状态为 SEM_EMPTY。它在信号量刷新成功的情况下返回 STATUS = OK，并将 semFlagID 状态置为 SEM_EMPTY。另外，它在由于超时、信号量不可用或者信号量标识符无效出现错误的情况下返回“ERROR”。执行 semFlush( )函数之后，所有等待该 semFlagID 的任务都取消阻塞(semFlagID 先前状态为 SEM_FULL)。

(5) 为 IPC 创建互斥信号量

当存在和其他任务共享数据结构的临界区域时，就需要使用互斥信号量(例如，发送任务和接收任务之间缓冲区中的字节)。也可能出现两个任务之间共享硬件设备或者文件。

a. 使用二进制信号量用于互斥操作的使用范例如下：这里，semMKeyId 不是作为特定信号量用于获取键，而是作为信号量密钥 SemKey，如示例 10-17~示例 10-20 中用在 MUCOS 中的信号量键。

```
1. SEM_ID semMKeyID;
2. semMKeyID = semBCreate(SEM_Q_PRIORITY, SEM_FULL);
```

b. 函数“SEM_ID semMCreate (options)”创建一个由 SEM_ID 指针所指的 ECB。10.3.4(2)解释了 semTake 函数，10.3.4(3)解释了 semGive 函数。假定进入任务的临界区域时执行 semTake(semMReadPortAKey)，离开临界区域时执行 semGive(semMReadPortAKey)函数，这里使用互斥信号量 semMReadPortAKey。示例 10-22 给出了该函数传递两个选项的使用范例，在示例的步骤 8 中，通过使用 semMCreate 函数传递这两个选项。这里，我们通过选择其中一个选项，可以防止出现优先级反转的情形(8.2.3 节)。另一个选项用于选择 SEM_Q_FIFO 或者 SEM_Q_PRIORITY。但是，当选择 SEM_INVERSION_SAFE 时，必须选择 SEM_Q_PRIORITY(在 10.3.2(1)中给出了在通过单个参数传递多个选项的情况下，使用|符号而不使用&符号的原因)。

c. 下面是另一个传递 3 个选项的使用范例，其中使用了 semMCreate 函数。这里，我们在整个过程中防止了发生优先级反转的情形，并保护任务不被另外一个任务，直到信号量在任务的临界区域末尾被清空(未得到)。选择的两个选项如下：

```
1. SEM_ID semMReadPortAKey;
2. semMReadPortAKey = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE
|SEM_DELETE_SAFE);
```

传递参数：(i)在 SEM_Q_PRIORITY 和 SEM_Q_FIFO 之间选项的使用等同于二进制信号量，在先前的的 10.3.4(1)中有描述。(ii)SEM_INVERSION_SAFE 使得使用互斥的临界区域时不会出现优先级反转的情形(参见 8.2.3 节)，这表明所创建的信号量的初始状态为 taken。(iii)回顾 SEM_DELETE_SAFE 的使用，当在任务的临界区域使用时，可以防止任务被删除。

返回参数：函数 semMCreate( )返回指针*SEM_ID，指向分配给互斥信号量的 ECB。如果出现没有可用 ECB 的错误，则返回 NULL。

想进一步了解 semMCreate 的使用，请参考示例 10-22 的内容。

示例 10-22

```
1. /* Include the VxWorks header file as well as the task and semaphore functions
from a library. */
2. # include "vxWorks.h"
3. # include "semLib.h"
4. # include "taskLib.h"
5. /* Declare a semaphore key to be used as mutex. */
6. SEM_ID semMReadPortAKey;
7. /* Create the mutex and pass the options chosen selected to it. */
8. semMReadPortAKey = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE);
9. /* This makes the mutex semaphore full (taken) and available like SemKey set
as 1 in MUCOS. */
10. semGive (semMReadPortAKey);
/* Other codes for task creation */
.
.
11. /* End of Task Creation Codes */
12. /* Task Codes */
.
. .
```

```
/* Entering Critical region. *
13. semTake (semMReadPortAKey, WAIT_FOREVER);
.
. .
14. /* Leaving the critical region. */
15. semGive (semMReadPortAKey);
/*************************************************************************/
```

若要使用互斥量，必须考虑如下情况：

a. 使用互斥量用于资源保护的临界区域不应该太长，而且应该尽可能的短。

b. 如果一些任务使用 taskDelete( )函数，则使用选项 SEM_DELETE_SAFE。

c. 如果可能出现一个优先级反转情形，则使用选项 SEM_INVERSION_SAFT。

d. 在临界区域的开始部分使用 semTake 函数，在结尾部分使用 semGive 函数，临界区域里有一些共享资源。semTake 可以循环使用，但是 semTake 执行的总次数应该和 semGive 执行的总次数相同。

e. 不使用 semGive( )进行互斥发送。

f. 不使用 semFlush( )(在使用互斥信号量时使用它是非法的)。semFlush 函数请参见 10.3.4(4)部分的内容。

(6) 创建用于 IPC 的计数信号量

VxWorks 计数信号量(参见 8.1.2(5))类似于 POSIX 信号量(参见 8.2.2(6))。它们都在发送(提供)信号量时加 1，在取得(等到)信号量时减 1。在信号量被取得之前允许发送至多 256 次。只有在信号量提供的次数等于其获取的次数时，信号量的状态值才会等于计数信号量的初始值。计数信号量有助于解决有界缓冲区问题、环形缓冲区问题和消费者-生产者问题(参见 8.2.2(6))。我们已经在示例 10-17~示例 10-20 看到了这一点。如果初始计数值等于 0，则等待信号量的任务被阻塞。

函数 SEM_ID semCCreate (options, unsigned byte initialCount)用于创建一个 SEM_ID 指针所指的 ECB。在调用函数时必须传递两个选项中的一个。例如："semCCharCheckFlagID = semCCreate (SEM_Q_PRIORITY, SEM_EMPTY);"。

```
1. SEM_ID semCID;
2. SEM_ID = semCCreate (SEM_Q_PRIORITY, 0); /*初始计数值为 0。*/
```

传递参数：(i)可供选择的选项一个是 SEM_Q_PRIORITY，另一个是 SEM_Q_FIFO。两个选项都可以作为初始状态被选择：initialCount 应该要么传递 0 值，要么是一个固定值。这取决于信号量对于已经阻塞的任务是(a)减 1 计数，还是(b)加 1 计数(回顾 MUCOS 中信号量 semCount 的使用。(ii)在示例 10-17~示例 10-20 中 semCount 值已经被初始化为 0)。

返回参数：函数 semCCreate( )返回指针*SEM_ID。在分配给计数信号量的 ECB 发生错误时，返回 NULL。如果没有可用 ECB，也返回 NULL。

回顾 MUCOS 示例 10-18 中步骤 11~21。示例 10-23 介绍了如何使用 VxWorks semCCreate 函数，以及用于任务生成和信号量的其他 VxWorks 函数的用法。

示例 10-23

```
1. /* Same as Steps 1 and 2 of Examples 10.21 and 10.22. */
2. /* Declare and Create Semaphores function, its identifying variables. */
```

```
/* Declare SemFlag1ID as the argument that passes to the task whenever called.
Declare SemMKeyID and SemCCountID as the mutex and counting semaphores. */
SEM_ID SemFlag1ID, SemMKeyID, SemCCountID;
3 /* Create Semaphore flag and declare unblocking of the tasks priority wise.
Declare initially semaphore flag unavailability. */
SemFlag1ID = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY);
4. / * Create Semaphore mutex and declare unblocking of the tasks priority wise.
Initially sema-phore mutex is available by default. */
SemMKeyID = semMCreate (SEM_Q_PRIORITY); /* SEM_Q_PRIORITY |
SEM_DELETE_SAFE two options can also be used. However that prolongs the execution
time.
We are not using safe option as taskDelete ( ) function is not used anywhere
in the codes. */
5. /* Create Semaphore for counting and declare unblocking of the tasks priority
wise. Initially sema-phore mutex is available by default. */
unsigned byte initialCount = 0;
SemCCountID = semCCreate (SEM_Q_FIFO, initialCount);
unsigned short COUNT_LIMIT = 80; /* Declare limiting Count = 80 */
6. /* Declare and Create Semaphores task function, its variables and parameters.
*/
void Task_ReadPortA (SEM_ID SemFlag1ID);
int readTaskID = ERROR; /* Let initial ID till spawned be none */
int Task_ReadPortAPriority = 105; /* Let priority be 105 */
int Task_ReadPortAOptions = 0; /* Let there be no option. It is the only task
that waits for the SemFlag1ID from the port. */
int Task_ReadPortAStackSize = 4096; /* Let stack size be 4 KB memory */
4. /* Create and initiate a task for reading at Port A. Task name starts with
't'. The task calling-function is Task_ReadPortA */
readTaskID = taskSpawn ("tTask_ReadPortA", Task_ReadPortAPriority, Task_Read
PortAOptions, Task_ReadPortAStackSize, void (* Task_ReadPortA) (SEM_ID
SemFlag1ID), SemMKeyID,
SemCCountID, &initialCount, COUNT_LIMIT, 0, 0, 0, 0, 0, 0); /* Pass SemFlag1ID
as the argument of task function and pass other arguments SemMKeyID and
SemCCountID as arg0 and arg1. Remaining eight arguments are 0s. */
.
.
/* Other Codes */
.
.
5. /* The codes for the Task_ReadPortA redefined to use the key, flag and counter*/
static void Task_ReadPortA (SEM_ID SemFlag1ID) {
6. /* Initial assignments of the variables and pre-infinite loop statements that
execute only once */
; /* Declare the buffer-size for the characters countLimit = 80 */
.
.
7. while (1) { /* Start an infinite while-loop. We can also use FOREVER in place
of while (1). */
8. / *Wait for SemFlag1ID state change to SEM_FULL by semGive function of
character availability
check task */
semTake (SemFlag1ID, WAIT_FOREVER);
```

```
9. /* Take the key so that another task, port decipher does not unblock. That
task needs SemMKeyID to unblock and run */
semTake (SemMKeyID, WAIT_FOREVER); /* SemMKeyID is now not available and the
critical region starts */
10. if (initialCount > = COUNT_LIMIT) {
.
.
};/* End of Codes for the action on reaching the limit of putting the characters
into the buffer */
11. /* Codes for reading from Port A and storing a character at a queue or buffer*/
.
.
12. semGive (SemCCountID); initialCount ++; /*Let counting semaphore value
increase because one character has been put into the buffer holding the character
stream. initialCount incremented because of the need to compare later with the
COUNT_LIMIT*/
13. semGive (SemMKeyID); /* Critical region ends. Release the mutex SemMKeyID
to let next
cycle of this loop start */
14. }; /* End of while loop*/
15. } / * End of the Task_ReadPortA function */
/
*****************************************************************************/
```

(7) 使用 POSIX 信号量

POSIX 信号量函数也可以用于 VxWorks 计数信号量。函数“semPxLibInit( )”初始化 VxWorks 库，以允许使用 POSIX 信号量函数。函数 sem_open( ), sem_close( )和 sem_unlink( )初始化、关闭和删除已命名的信号量。函数 sem_post( )和 sem_wait( )对信号量进行解锁和加锁。sem_trywait( )在信号量没有锁定的情况下锁定信号量。这些函数的动作是提供、获取和接受 VxWords 计数信号量或者 MUCOS 的 OSSemPost、OSSemPend 和 OSSemAccept 函数。函数 sem_getvalue( )检索 POSIX 信号量的值。POSIX 信号量函数 sem_init( )和 sem_destory( )初始化和销毁未命名的信号量。销毁的意思是释放与该信号量 ECB 相关的存储器。这样做的效果和首先关闭信号量，然后通过 sem_close 和 sem_unlink 断开链接是相同的。记住，在使用这些函数之前，没有类似 VxWorks 互斥的删除保护可用。VxWorks 信号量具有如下附加特性：(i)防止优先级反转和任务删除的保护选项。(ii)单个任务可以多次循环取得信号量。(iii)可以定义互斥所有权。(iv)两个选项，FIFO 和用于多任务信号量等待的任务优先级。

(8) 创建用于 IPC 的消息队列

函数“MSG_Q_ID msgQCreate (int maxNumMsg, int maxMsgLength, int qOptions)”用于创建指针 MSG_Q_ID 所指的 ECB。在调用函数时必须传递下面提出的两个选项。对用于保存消息的缓冲区的分配存储器是根据 maxNumMsg 和 maxMsgLength 进行的。

需要注意的一点是，在 MUCOS 中消息指针被传递给消息指针数组。另外，设定可以提供(发送)的计数信号量时没有规定界限，它固定为 256。所以，必须使用 COUNT_LIMIT 变量，如示例 10-23，并将它和 initialCount 变量比较，这个变量在每次调用“semGive (SemCCountID)”函数时加 1。另外，在消息队列中，消息的最大数目为 $2^{31}-1$，每个消息的最大字节数为 $2^{31}-1$。在 VxWorks 中，这些变量不声明为无符号整数，而声明为整数，这样做的好处是，简化了在为这些变量传递负值时发生错误的处理。

传递参数：(i)对于该函数，maxNumMsg 传递可以发送到队列的最大消息数目。(ii)maxMsgLength 传递允许作为消息发送的最大字节数。(iii)当消息送进一个 FIFO 模式的队列时，可以选择的一个选项是 MSG_PRI_NORMAL。当消息送进带有该选项的队列时，消息以 LIFO 的方式接收。发送类似错误登陆的紧急消息时需要选择该选项。最后一个消息最先被读取。(iv)可以选择的另外一个选项是 MSG_Q_PRIORITY。其他选项是 MSG_Q_FIFO。假定在某一时刻，几个任务处于阻塞状态，并且正在等待来自相同队列的消息用于发送。等待任务可以采用两种方式获得消息。(a)优先级比其他等待任务更高的任务首先从队列中取得消息。通过选择 MSG_Q_PRIORITY 选项可以做到这一点。(b)在所有等待任务中，先阻塞并进入等待状态的任务，优先从队列中取得消息。通过选择 MSG_Q_FIFO 选项可以做到这一点。

返回参数：函数 msgQCreate( )返回指针*MSG_Q_ID。如果分配给消息队列的 ECB 出现错误，则返回 NULL。如果没有可用 ECB 块，则返回 NULL。

考虑应用程序中任务的客户机-服务器体系结构。服务器任务可以通过从公共请求队列中接收信息来接收客户端任务的请求。该函数创建请求队列。服务器可以通过单独的回复队列单独向每个客户端发送回复信息。该函数也创建回复队列。

示例 10-24 解释了 MsgQCreate 的使用。假定 Task_ReadPortA 从端口 A 读字节，并将其发送到另外一个任务，该任务在等到队列消息可用后，从队列中接收消息。

示例 10-24

```
1. /* Include the header files in Example 10.22 Step 1 as well as queue functions
from a library. */
2. # include "msgQLib.h"
3. /* Declare message queue identity and message data type or structure. */
4. MSG_Q_ID portAInputID;
5. unsigned byte portAdata;
6. void * message; /* Pointer for the message buffer */
7. /* Create the message queue identity and pass the parameters and options chosen
selected to it and let the maximum number of messages be 80 and message be of
1 byte each. */
8. PortAInputID = msgQCreate (80,1, MSG_Q_FIFO | MSG_PRI_NORMAL)
9. /* Task Creation Codes as in Example 10.23. */
10. .
11. .
12. /* Start Codes for Task_ReadPortA. */
13. void Task_ReadPortA {
14. .
15. while ( ) {
16. semTake (SemFlag1ID, WAIT_FOREVER);
17. /* Take the key to not let port-decipher task unblock and run. Because that
task also needs SemMKeyID for running. */
18. semTake (SemMKeyID, WAIT_FOREVER); /* SemMKeyID is now not available and
the critical region starts */
19. .
20. .
21. /* At the end, the send the byte, which is read at Port A. It is sent as
a message to queue, portAInputID. Refer to Section 10.3.4.(10) for an
understanding of the msgQSend function. */
```

```
22. *message = portAData;
23. msgQSend (portAInputID, &message, 1, NO_WAIT, MSG_PRI_NORMAL);
24. /* Other remaining codes for the task. */
25. .
26. .
27. }
28. }/* End of Task_ReadPortA Codes */
```

(9) 为 IPC 等待二进制或者其他类型信号量的释放或者在释放后检查 IPC 的可用性

函数“int msgQReceive (msgQId, &buffer, maxBytes, timeOut)”让任务等待，直到发送消息。任务一直等待，直到 MsgQSend 函数发送任务中的消息，或者出现超时，不论哪种情况发生，等待过程都会结束。

传递参数：(i)无论何时调度程序调度这个任务或者该函数被调用，都会等待 msgOld 指针所指的消息。(ii)buffer 是存储器接收消息的缓冲区的地址。(iii)maxBytes 是使用 msgQReceive 函数接收到的每条消息中，可接收的最大字节数。(iv)timeOut 是超时时间。如果必须等待使用 msgQSend 函数发送消息，则可以选择 WAIT_FOREVER 选项。另一个选项是 NO_WAIT。回顾 MUCOS 中的 OSQFlush 函数。NO_WAIT 选项仅用于检查消息的可用性。如果消息可用，就将其放到缓冲区中，否则任务运行其他后续代码。如果出现错误消息，比如，没有特别等待消息，就只需要检查。

返回参数：函数 msgQReceive( )返回一个表示检索到的字节数的整数，并将其放到缓冲区中。

示例 10-25 解释了 msgQReceive 函数的用法。

示例 10-25

```
1. to 9. /* Codes as per Steps 1 to 9 in Example 10.24 */
10. /* The codes for the Task_MessagePortA */
static void Task_MessagePortA (void *taskPointer) {
11. /* Initial assignments of the variables and pre-infinite loop statements
that execute once only*/
int maxBytes = 80;
12. while (1) { /* Start an infinite while-loop. */
13. /* Wait for a Queue Message sending or availability. */
msgQReceive (msgQId, message, maxBytes, WAIT_FOREVER); /* WAIT_FOREVER means
timeout = -1 and the period is thus infinity. */
14. /* Other remaining Codes */
.
.
15. }; /* End of while loop*/
16. }/ * End of the Task_MessagePortA function */
/
**********************************************************************/
```

(10) 在消息送进队列后发送 IPC

函数“STATUS msgQSend (msgQId, &buffer, numBytes, timeOut, msgPriority)”用于将任务送进队列。

传递参数：(i)队列使用 msgOLD 进行标识。(ii)息发送到指定地址的缓冲区。送进缓冲区的字节数为 numBuffer。(iii)timeOut 是队列满的情况下发送消息的最长等待时间。(iv)msgPriority

被指定为 MSG_PRI_NORMAL 或者 MSG_PRI_URGENT，这取决于消息插入后，使用 FIFO 模式还是 LIFO 模式检索。

返回参数：函数 msgQSend( )返回 STATUS。如果成功取得 msgOLD，返回 STATUS = OK，如果出现 msgOLD 无效的错误，则返回“ERROR”。

示例 10-24 的步骤 23 介绍了该函数的用法。

(11) 使用 POSIX 队列

使用 POSIX 队列时要注意以下几点：

1. 函数“mqPxLibInit( )”初始化 VxWorks 库，以允许使用 POSIX 信号量队列。

2. 函数 mq_open ( )，mq_close ( )和 mq_unlink ( ) 初始化、关闭和删除命名队列。

3. 函数 mq_setattr ( ) 设置 POSIX 队列的属性。

4. 函数 mq_send ( )和 mq_receive ( )对队列解锁和加锁。

5. 函数 mq_notify ( )给当前消息可用的单个等待任务发送信号。该信号专门用于单个任务，它应该已经注册为接收该信号的任务(已经注册的意思是随后可以获得 mq_notify 的通知信号)。这个规定对于服务器任务特别有用。服务器任务通过信号处理函数(如 ISR)接收来自客户端任务的通知。

6. 函数 mq_getattr ( ) 检索 POSIX 队列的属性。

7. POSIX 队列函数 mq_unlink ( )不会立即销毁队列，但是会防止其他任务使用该队列。只有在最后一个任务关闭队列时，队列才被销毁。销毁的意思是释放和队列 ECB 相关的存储器。

VxWorks 有如下附加特性：(i)可以使用超时选项。(ii)两个选项，FIFO 和多个任务等待队列的任务优先级。POSIX 队列有如下附加特性：(i)单个等待任务可用时任务发出通知。(ii)可以有 32 个消息优先级，而在 VxWorks 中只有一个优先级 URGENT。

(12) 为 IPC 创建管道设备和读写通道

在 VxWorks 中的管道是一个通过管道驱动(类似设备驱动)pipedrv 管理的队列(这和 UNIX 中的命名管道相似)。管道也在一组任务之间实现客户机-服务器的体系结构。

函数 pipeDevCreate(“/pipe/pipeName”, maxMsgs, maxMsgBytes)创建名为 pipeName 的管道设备，可以发送最大数目为 maxMsgs 个的消息。每个消息最大长度可以为 maxMsgBytes。该设备在创建时进入设备列表。devs( )函数使用分配给每个设备，包括管道设备的设备号检索设备列表。

考虑创建一个名为 pipeUserInfo 的管道示例。假定最大消息数目为 4：用户名、密码、电话号码和电子邮箱 ID。其中每个消息的最大长度可以是 32 字节。全局变量 fd 是用作文件描述符的一个整数，文件描述符用于标识 I/O 系统中众多设备中的一个。示例 10-26 解释了创建、写操作和读操作的代码。

示例 10-26

```
1. # include "fioLib.h"/* Include the IO library functions. */
pipeDrv ( ); /* Install a pipe driver. */
2. /* Declare file descriptor. */
int fd;
3. /* Mode refers to the permission in an NFS (Network File Server). Mode is
reset as 0 for unrestricted permission. */
int mode;
```

```
4. /* Create pipe named as pipeUserInfo for 4 messages, each 32 bytes maximum.
*/
pipeDevCreate ("/pipe/pipeUserInfo", 4, 32); mode = 0x0;
Messages can be written into a pipe by the function by first opening a pipe device
and then writing into that. The function for opening is open ("/pipe/pipeUserInfo",
rdwrFlag, mode).We define flag = O_RDWR, which both permits read and write. Flag
O_RDONLY permits the read only option and flag O_WRONLY permits the write only
option. Remember that after opening a pipe, when we finish using it, we must
use the function 'STATUS close ( ) '. Writing to a pipe is analogous to writing
to an I/O device. To write, the coding is as following:
5. /* Open read-write device using, a pipe named as pipeUserInfo with mode =
0 for unrestricted permission. */
fd = open ("/pipe/pipeUserInfo", O_RDWR, 0);
6. /* Write a message, info of lBytes. */
unsigned char [ ] info; Let the message be a string of characters. */
int lBytes;
lBytes =12;
write (fd, info, lBytes);
The message can be read from an open pipe by the function 'int read (fd, &buffer,
lBytes) '
Reading from a pipe is analogous to reading from an I/O device as per file
descriptor. To read, the coding is as following:
7. int numRead; /* An integer to indicate the number of bytes successfully read.*
/
lBytes =12; /* Let the Message bytes to read = 12*/
numRead = read (fd, info, lBytes);
```

(13) 从 IPC 中选出表示设备号的位

文件描述符可用于管道、套接字、串行设备或者其他类型的设备。设定文件描述符 fd = n；一个位数组中的第 n 位对应于 fd = n。还有“C”结构 struct fd_set，该结构包含有一组文件描述符和一组函数。

如果使用该结构中的函数 FD_SET(n, &fdSet)，则在执行时，将第 n 位置为 set。函数 FD_SET(m, &fdSet) 将第 m 位置为 set。FD_CLR(n, &fdSet)将第 n 位置为 clear。FD_ZERO(&fdSet)将数组的所有位置 0。如果数组中的第 n 位是 set，则 FD_ISSET(n, &fdSet) 返回 true，如果是 reset，则返回 false。

现在，我们分析一下任务如何寻找和选择某一时刻活动设备的数量。任务要么发现管道 pipeUserInfo 是有效的，要么发现 pipeServerReply 是有效的。用于选择的函数是“int select (numBitWidth, pointerReadFds, pointerWriteFds, pointerExceptFds, pointerTimeOut)”。传递的参数如下：numBitWidth 为两个指针 pointerReadFds 和 pointerWritedFds 的位数组中要检验的位数。检查是按照某种数据结构中的值进行，这里的数据结构在设置无限等待时，存储 NULL，在设置超时限制时，存储一个表示超时时间的值。选择清除和未就绪设备对应的所有位。该函数返回活动设备的数量。如果出现错误，则返回 ERROR。

本章小结

本章所学内容总结如下：

- 在复杂多任务嵌入式系统中必须使用测试稳定和调试合格的 RTOS。两个重要的 RTOS 是 MUCOS 和 VxWorks。
- 使用 MUCOS 任务创建、删除、挂起和恢复执行的函数进行任务控制和调度。
- 在 MUCOS 中有初始化系统定时器的函数。通过第一个任务启动多任务系统，并在后面将它永远挂起，这在本章中作为多任务系统编程的一项技术给出。
- MUCOS 采用类似的方式处理调度任务和 ISR。
- MUCOS 中有延迟和延迟恢复函数。这些函数在使低优先级任务运行时非常有用。
- MUCOS 有进程间通信(IPC)函数，信号量、邮箱、和队列。MUCOS 的简单特性在于同样的信号量函数可以用于二进制信号量、事件标志、资源键和计数。
- MUCOS 有邮箱函数，它的一个简单特性是每个邮箱有一个消息指针。这样的结果是，使用同样的指针可以访问任意数量的消息或者字节。
- MUCOS 有队列函数。队列从发送任务接收一组消息指针。消息指针插入队列后，可以使用 FIFO(先进先出)模式，也可以使用 LIFO(后进先出)模式在队列中检索消息指针。到底使用哪种模式取决于使用发送还是使用发送前函数。这有助于注意到队列中优先级较高的消息。
- VxWorks 不是使用一个创建函数，而是使用 3 个函数，任务创建、任务激活和任务生成(创建并激活)。
- VxWorks 采用不同方式处理调度任务和 ISR 函数。它为 ISR 分配所有任务中最高的优先级。
- VxWorks 具有信号服务例程。信号服务例程是一个 C 函数。它在出现中断或异常时执行。连接函数将函数和中断向量连接起来。
- VxWorks 具有称作信号的 IPC。它用于异常处理或者中断处理。
- 异常是指软中断。在硬中断的情况下，信号设置相当于标志设置。
- VxWorks 提供循环时间片调度和抢占式调度。
- VxWorks 提供了两种方式用于取消阻塞待定任务。一种是按照任务优先级，另一种是在接收(已得到)IPC 时使用 FIFO 模式。
- VxWorks 有 3 个不同的信号量函数用于 IPC 中的事件标志、资源键和计数信号量。VxWorks 也支持 POSIX 信号量。
- VxWorks 不仅对消息指针排队，还提供了消息的排队。队列可以如同在 MUCOS 中一样，按 LIFO 模式用于优先级消息。
- VxWorks 也支持使用管道和 POSIX 队列。
- VxWorks 管道是可以像文件一样打开和关闭的队列。管理类似虚拟的 I/O 设备，按 FIFO 方式存储消息。

关键词及其定义

- 测试稳定和调试合格的 RTOS：在许多情况下通过充分的测试和调试的 RTOS。
- 复杂多任务嵌入式系统：有多种特性、多个任务需要的系统，并且在这个系统中对任务的时限也有规定。
- MUCOS：Jean J.Labrosse 设计的一种 RTOS μC/OS-II。
- VxWorks：来源于 Wind River®系统的一种 RTOS。
- 任务创建：任务被分配一个 TCB(任务控制块)和标识符。MUCOS 中的创建也包括创建的初始化和调度。
- 任务删除：任务不再有其 TCB，直到再次创建之前，任务一直被忽略。
- TCB：任务控制块，它带有任务参数，任务切换保存被切换任务的参数，在任务重新切换回来时，使得任务可以从其离开的地方继续执行。任务是独立的进程。
- 任务挂起：任务不能继续运行其代码。
- 任务恢复执行：被延迟或者挂起的任务，在恢复执行时可以再次被调度。
- 系统定时器：可以按预设时间间隔设置中断的 RTC。系统周期性地产生中断信号，并且时间周期性更新。RTOS 获得 CPU 的控制权，检查是否需要抢占。任务优先级为系统定时器函数、延迟函数和延迟恢复函数提供优先级。
- 任务延迟：让任务等待一个由系统节拍数定义的最短时间，该时间作为参数传递给任务延迟函数。
- 任务生成：任务创建和激活。
- 信号：在通过执行 ISR 运行一个需要紧急注意的事件时，向 RTOS 发出的类似标志的一种提示，用于特定情形下的事件处理。
- 异常处理：收到信号后执行处理函数。错误也通过使用异常处理函数来处理。
- 事件标志：一种标志，在事件发生时置位，在响应时间时复位。
- 资源键：一种信号量，在临界区域代码开始执行时复位，在执行完成时置位。
- 计数信号量：一种信号量，在任务或者任务的一段提供 IPC 时加 1。当等待任务取消阻塞并开始运行时减 1。
- POSIX 信号量：遵循 IEEE POSIX 标准的信号量函数。
- 消息队列：存放任务发送的消息的队列，并且可以被另外一个任务检索。
- POSIX 队列：遵循 POSIX 标准的 IPC 队列函数。
- 管道：一种队列，一个任务从中获取消息，而其他任务将消息放入其中。VxWorks 管道是一种 I/O 设备函数操作的队列。在管道中放置和获取消息如同在文件中放置和获取消息。
- 文件设备：文件读、文件写和文件关闭函数操作的存储块，如同磁盘中的文件操作。

问题回顾

(1) 测试稳定和调试合格，通用而且可信的 RTOS 具有哪些优点(提示：嵌入式软件开发必须快速、高效。进行设备驱动程序、存储器和设备管理器、网络任务、异常处理、测试向量、API 等开发的小组需要综合编码技巧)?

(2) 如何区分邮箱消息和队列消息？可以使用消息队列作为计数信号量吗？

(3) 解释 ECB(事件控制块)的含义。

实践练习

注意：

练习 5~12 属于 MicroC/OS-II 的相关内容，练习 14~23 属于 VxWorks 的相关内容。

(4) 通过互联网(例如 www.eet.com)查找最新的 RTOS 产品。

(5) 分类并列出和处理器相关的源文件，以及与处理器无关的源文件。

(6) 设计一个表，给出 MUCOS 的特性。

(7) MUCOS 只有一类信号量用作资源键、标志、计数信号量和互斥量。这种简化有何优点？

(8) 如何使用函数 void OSTimeSet (unsigned int counts)设置系统时钟。

(9) 什么时候使用 OS_ENTER_CRITICAL ( ) 和 OS_EXIT_CRITICAL ( )?

(10) 如何设置抢占式调度任务的优先级和参数 OS_LOWEST_PRIO 与 OS_MAX_TASKS?

(11) 首先创建启动任务，同时创建所有需要的任务，初始化系统时钟，然后挂起任务。为什么要使用这种策略？

(12) 分别列出 10 个信号量、邮箱和消息队列的应用程序示例。

(13) VxWorks 内核包括 POSIX 标准接口和 VxWorks 专用接口。说明用于信号量和队列的专用接口有什么优点。

(14) 列出任务状态转换函数。

(15) 如何初始化循环时间片调度？给出 10 个循环调度的例子。

(16) 如何初始化抢占式调度并指定调度任务的优先级，给出 10 个抢占式调度的例子？

(17) 如何使用信号量，以及如何使用函数“void sigHandler (int sigNum)”，“signal (sigNum, sigISR)”和“intConnect (I_NUM_TO_IVEC (sigNum), sigISR, sigArg)”？给出 10 个使用这些函数的例子。

(18) 如何创建计数信号量？

(19) OS 支持所有 ISR 共享一个堆栈。需要加以何种限制来支持该特性？

(20) 如何使用 RTOS 函数创建、删除、打开、关闭、读、写，以及作为设备的 I/O 控制？给出一个管道从网络设备传送 I/O 流的例子。

(21) 解释用于 I/O 设备和文件的文件描述符。

(22) 在抢占式调度程序中，如何执行一个优先级较低的任务？给出 4 个编码示例。

(23) 如何生成任务？为什么除非存在存储器限制，否则不能删除任务(提示：参见表 9-11)?

(24) 写出用于定时器、信号量和队列的 POSIX 函数的编码。

(25) 如何使用套接字来创建客户机/服务器体系结构？

第 11 章　RTOS 编程案例研究

本章前所学内容

以下是前面章节中所涉及到的有关嵌入式系统的几个重要方面：

(1) 嵌入式系统中的硬件单元、处理器、存储器、总线、接口电路、端口设备、实时时钟驱动软件定时器、设备驱动程序和中断处理机制(参见第 1~4 章)。

(2) 高级语言编程原理、程序模型、软件工程方法、RTOS 和 IPC 信号量的使用、邮箱、队列和管道，它们用于同步多任务，以及使用中断禁用/启用机制或二进制数、互斥、PV 操作、POSIX 和计数信号量来解决与进程间变量共享相关的问题(参见第 8~10 章)。

(3) 27 个使用示例，涵盖了嵌入式系统两个最常用的 RTOS 的系统、任务和 IPC 函数。这两个 RTOS 是源自 Jean J. Labrosse 的 MUCOS(μC/OS-Ⅱ)和源自 WindRiver 系统的 VxWorks(参见第 10 章)。

本章将学内容

本章给出的每个研究案例都解释并详细说明了在以下 4 个领域中 RTOS 编程工具的使用。

(1) 消费电子系统。

(2) 通信和网络系统。

(3) 汽车电子系统中的控制系统。

(4) 安全的 Soc 系统。

本章研究案例的示例代码将解释：

a. 11.1 节给出了使用 MUCOS 作为 RTOS 而设计的巧克力自动售卖机系统(ACVS)的详细代码。它介绍了(i)一项有趣的 RTOS 应用；(ii)相同类型的信号量作为事件标志、资源键和计数器的使用；(iii)系统 RTC 和邮箱的使用。

b. 11.2 节将介绍在网络上发送 TCP/IP 栈(字节流)的代码和代码设计。这些字节流通过 TCP/IP 协议在多个层之间传递。最终由网络驱动器将这些字节流发送到 TCP/IP 网络上。该案例研究给出 VxWorks RTOS 的各种功能的应用。我们将学习 VxWorks 二进制和互斥信号量类型、消息队列和管道的应用。可以预先定义发送到队列和管道的消息的字节数，而不是 MUCOS 中的消息指针。VxWorks RTOS 中的这些特性使得在嵌入式系统中应用 RTOS 变得非常方便。

c. 11.3 节给出了汽车行驶控制(automotive cruise control，ACC)系统的代码和代码设计。首先应该了解用于汽车电子设备的嵌入式系统、控制系统中的自适应算法，以及包括标准 RTOS 和 OSEK-OS 在内的一些软硬件的重要标准。在理解了它和 VxWorks、MUCOS 的区别之后，我们还应该了解，如何使用集成了 OSEK 特性的 VxWorks 进行代码设计。

d. 11.4 节给出了智能卡的代码和代码设计。选择这个案例是为了解释一些必要的特殊嵌入式硬件和特殊的 RTOS 函数。

11.1 使用 MUCOS RTOS 对巧克力自动售卖机编码

嵌入式系统是消费电子领域使用最多的系统之一。我们在这里给出了 MUCOS RTOS 的一个案例研究。现在假设需要设计一个巧克力自动售卖机。

11.1.1 案例定义、多任务及其函数

我们首先要了解系统的需求。该机器有一个用于插入硬币的插槽。以下是案例的定义。

图 11-1(a)给出了“巧克力自动售卖机系统”(ACVS)的基本结构。图 11-1(b)给出了 ACVS 上的各个端口。系统需求如下：

(1) 需要一个插槽，小孩(购买者)可以从这里投入硬币购买巧克力。巧克力的价格是 Rs. 8。硬币可以是三种面值中的一种：Rs.1、Rs.2 和 Rs.5。无论何时投入硬币，机器的系统都可以将其转到适当的端口，Port_1、Port_2 和 Port_5。

(2) 该机器应该有一个 LCD 显示装置作为“用户界面”。将界面端口记为 Port_Display。它可以显示 3 行消息字符串。

(3) 它应该有一个小碗，购买者可以通过一个投放巧克力的端口从这里取得巧克力。将该端口记为 Port_Deliver。Port_Deliver 连到巧克力通道，机器的主人可以使用巧克力填满这个通道，不论通道中有多少个巧克力剩余空间。购买者也可以分别在零钱不够的情况下通过端口 Port_Refund，以及在零钱过多的情况下通过端口 Port_ExcessRefund 从这个小碗中收取全部退款或者多余退款。所有端口 Port_Deliver、Port_Refund 和 Port_ExcessRefund 都和 Port_Collect 通信。Port_Collect 是一个连到小碗的公共机器接口。

(4) RTOS 必须调度从开始到结束期间的购买过程(任务)。

(5) 它也应该可以实现，不论在以下哪种情况下，系统 ROM、闪存或 EPROM 中代码的重新编写和重新移植：(i)提高巧克力价格；(ii)改变消息行；(iii)改变机器特性。

选择 MUCOS RTOS 用于 ACVS 的嵌入式软件开发。以下是多个任务的详细内容：

(1) 3 个端口，Port_1、Port_2、Port_3 的所有 24 个位都处于复位状态(0)，或者在加电时将其复位。

(2) 除了连接机器的子系统，每个端口都向 ACVS 提供数字输入位，并接收系统输出位。

(3) 机器的子系统提供了帮助设施，它在出现两个或者两个以上相同类型的硬币时提供帮助。因而，在每个端口上，不同点数硬币的总和最大可以设为 8。

(4) 信号量 SemAmtCount 的使用方法是，在没有超时之前，小孩可以插入 8 个 Rs. 1 的硬币得到 Port_1 的 8 个点，或者插入一个 Rs. 1 得到 Port_1 点，插入一个 Rs. 2 得到 Port_2 点，插入一个 Rs. 5 得到 Port_5 点。小孩可以选择多种组合方法，机器可以通过这些组合，在收取硬币和投放巧克力之前得到巧克力的费用。

(5) ACVS 任务 Task_ReadPorts 完成以下工作：(i)从上面给出的每个端口读取字节(8 位)。(ii)如果 SemAmtCount 是依据巧克力的费用反映硬币的状态，则发送一个标志给 Task_Collect。Task_Collect 初始化将硬币收到收集部件的动作。(iii)Task_ReadPorts 在读完之后复位，保持端口以及所有的 24 个位处于就绪状态，为机器的下一个周期作好准备。(iv)在以下情况执行步骤

10 中描述的其他动作：(a)在给定的超时时间内，硬币没有将费用或者多余费用累加；(b)依据步骤 11 进行其他操作，在切换到其他任务之前，根据端口状态显示消息。该任务通过 3 个邮箱指针发送消息。

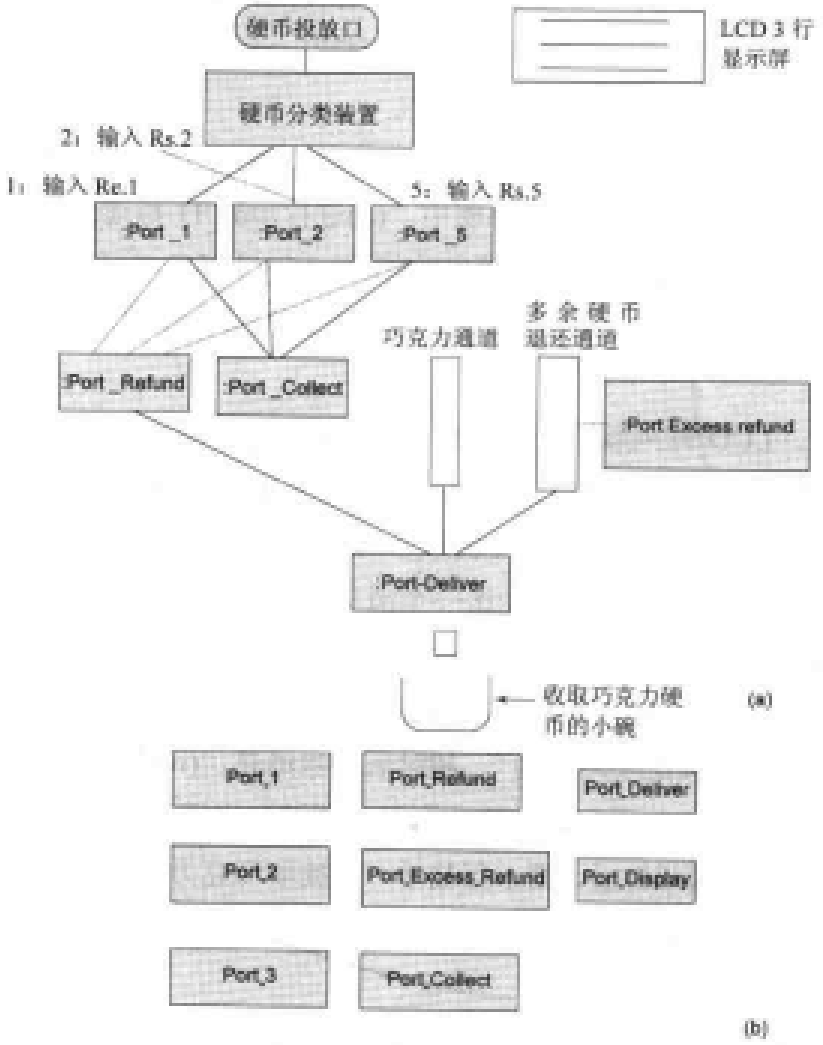


图 11-1　(a)巧克力自动售卖机的基本结构(ACVS)；(b)ACVS 的端口

(6) ACVS 电路设计是，当有一个硬币可用并且验证正确时，端口第 0 位，即位 0 置位(=1)。当有两个硬币可用时，端口位 1 也置位，当有 3 个硬币可用时，端口位 2 也置位，依此类推。有 8 个位作为端口的 8 个点。端口中 1 的数量决定了端口中硬币的数量。

(7) 如果端口 Port_Collect 接收到来自任务 Task_Collect 的一个指示(一个标志格式的信号，置位状态)，那么 3 个端口上用于 24 点的所有 24 个位使用电机设备释放硬币。硬币收集到一个收集部件内(要回收这些硬币，机器的主人可以在某个方便的时间使用钥匙打开该部件提取现金。机器主人也会在这个时候，在 Port_ExcessRefund 端口填满 8 个点的部件，保证小孩插入了多余的硬币时可以得到退款)。

(8) ACVS 任务 Task_Collect 做以下工作：(i)指导 Port_Collect 进行操作，该单元从端口 Port_1、Port_2 个 Port_5 收集所有可用的硬币。(ii)收集完毕之后，通过 Port_Deliver 发送一个 IPC 到另外一个任务 task_Deliver。该端口在置位时送出巧克力，然后复位等待下一个周期。(iii)进行步骤 11 中描述的其他用于显示的动作。

(9) 当端口 Port_Refund 接收到来自任务 Task_Refund 的一个指示(一个标志格式的信号，置位状态)，它指导每个端口上所有 8 个点，使用电机设备释放硬币。如果发现 3 个端口的硬币总额少于需要的费用，则将全部硬币退回到小碗中。当端口 Port_ExcessRefund 收到来自任务 Task_ExcessRefund 的指示时，它指导另外一个端口上的 8 个点，使用电动设备释放多余的硬币。如果发现 3 个端口的硬币总额多于需要的费用，则将多余的硬币退回到小碗中(要回收这些硬币，小孩可以查看小碗收取退款)。

(10) ACVS 任务 Task_Refund 做以下工作：(i)指导 Port_Refund 进行操作，该单元在硬币总额不够时将来自 3 个端口的硬币送回到小碗中。(ii)进行步骤 11 描述的其他动作。发送一个请求邮箱的显示消息，Task_Display 等待邮箱中的邮件。ACVS 任务 Task_ExcessRefind 操作如下：(i)指示单元中的多余退款操作，退出多余金额的硬币。(ii)进行步骤 11 中描述的其他动作，并向邮箱发送消息。

(11) 该步骤中，LCD 阵列端口获得来自 Task_Display 的消息。按照机器的状态，或者向任务等待邮箱发送邮件的时间日期进行显示。显示的消息如下：(i)当机器复位，或者周期开始时，在第 1 行显示欢迎消息“欢迎购买甜美难忘的巧克力”。第 2 行显示“请投币”。第 3 行右角显示“时间和日期”，它的值由每秒发送一个消息的任务 Task_TimeDateDisplay 决定。(ii)收到来自 Task_Collect 的邮件后，在第 1 行显示消息“请等待”。第 2 行显示“稍后可获得甜美的巧克力”。超时后清除消息。(iii)收到来自 Task_Deliver 的邮件后，第 1 行显示“请提取巧克力”。第 2 行显示“投入硬币继续购买巧克力”。超时后清除消息。(iv)收到来自 Task_Refund 的时间标志后，第 1 行消息显示“对不起!”。第 2 行显示“请取出退款”。超时后清除消息。(v)收到来自 Task_ExcessRefund 的邮件后，第 1 行显示“请提取巧克力和现金”。第 2 行显示“请勿忘记收取零钱”。超时后清除消息。(vi)超时后，或者机器为下一个周期复位时，按步骤(i)重复显示动作。

(12) OSSemPost 和 OSSemPend 信号量函数是同步的，所以任务必须等待代码的执行，直到在指定的超时时间内得到需要的令牌，令牌由 SemAmtCount 指出。Task_Deliver 向 AVCS 中投放巧克力的端口 Port_Deliver 发送信号，假设它仅仅在收取了特定硬币组合时才投放巧克力，从而保证收到的金额等于或者大于巧克力的价格。

由此可以看出，这里的多个任务需要使用 IPC 来实现互相同步。这里还需要使用 RTOS，以及二进制信号量、资源键信号量、计数信号量和邮箱。

11.1.2 创建任务、函数和 IPC

现在，设计者需要按这样的方式创建一个表：分别在 1、2、3 列的第一行写上任务名称、优先级和期望的动作。待定 IPC 和发送 IPC 应该列在第 4 列和第 5 列。机器或者其他系统的输入和输出放到第 6 列和第 7 列。图 11-2 给出了 ACVS 的多个任务以及它们的同步模型。表 11-1 给出了模型设计表。任务同步模型如下，这些明显来自表中的内容，它们同时也在图 11-2 中给出。

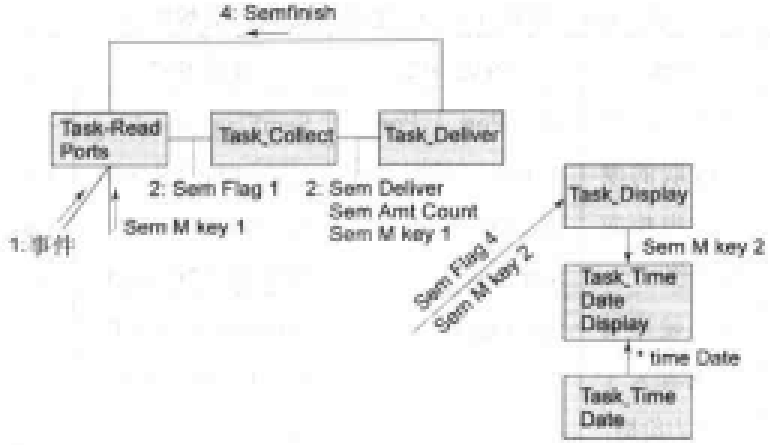


图 11-2　ACVS 中的任务及其同步模型

(1) Task_TimeDateDisplay 在每 1000 个系统时钟节拍后超时，复位为 1ms。超时后，更新指针*timeDate 所指的时间和日期值。并将值发送到邮箱*timeDate 中。Task_Display 将其显示在第 3 行，即 LCD 阵列的右下角。

(2) Task_ReadPorts 只有在信号量 SemFinish 被 Task_Deliver、Task_Refund 或者 Task_ExcessRefund 复位并发送的时候，才开始其动作。该任务发送 SemMKey1，访问临界资源 SemAmtCount。然后任务等待来自 Port_1、Port_2 或者 Port_5 的时间信号。通过发送信号给信号量 SemAmtCount，开始计算金额。机器输入的是 Port_1、Port_2 和 Port_5 上的硬币。在收到等于或者大于巧克力价格(= 8)的硬币总额时，发送事件标志 SemFlag 给 Task_Collect。如果收取的金额在超时时间内没有达到足够数量，则发送 SemFlag3 给 Task_ExcessRefund。同时，将发送 SemFlag4 给 Task_Display。最后，该任务发送请求邮箱的消息指针等待邮件。

(3) Task_Collect 等待获取 SemFlag1 和 SemAmtCount。释放 SemDeliver，使 Task_Deliver 对端口 Port_Deliver 进行写操作，从而投放巧克力。Task_Collect 也在 Task_ExcessRefund 得到 SemFlag3 时释放 SemDeliver。这是因为，小孩已经付了足够的金额，所以他或者她必须得到巧克力。在完成投放，并接受 SemAmtCount 将其置 0 的时候，Task_Deliver 释放 SemFinish。

(4) Task_Refund 等待获取 SemFlag2，然后刷新为 0、SemAmtCount 和 SemMKey1。在完成退款，并接受了 SemAmtCount 将其置 0 的时候，释放 Sem_Finish。最后发送 SemFinish 给 Task_ReadPorts。

(5) Task_ExcessRefund 等待获取 SemFlag3，接受 SemAmtCount 将其减 8，然后发送 SemFlag1，释放 SemMKey1。

(6) Task_Display 等待获取 SemFlag4。在向 Port_Display 的字节流发送字节之前，获取互斥量 SemMKey2，并在发送后释放互斥量。该任务显示消息指针*Collect、*delivered、*refund 和 *ExcessRefund 所指的邮箱消息。

表 11-1 示例 11-1 中使用的任务、函数和 IPC

任务函数	优先级	动作	待定 IPC	发送 IPC	ACVS 输入	ACVS 输出
Task_ Read Ports	5	等待硬币,并按收取的硬币作出相应的动作	来自 Port_1、Port_2 和 Port_5 的时间标志; SemFinish, SemMKey1	SemFlag1,SemFlag 2,SemFlag3,SemFlag4,SemMKey1 ，消息指针*Collect	在 Port_1、Port_2 和 Port_5 的硬币	–
Task_ Collect	7	等待硬币等于或者大于巧克力价格，直到超时并作出相应动作	SemFlag1	SemMKey1,SemDeliver，消息指针 *wait	–	在 Port_ Collect 的硬币
Task_ Deliver	8	–	SemDeliver,SemAmtCount,SemMKey1	SemFinish,SemAmtCount,SemMKey1 ,消息指针 *delivered	来自发送巧克力通道的巧克力	投放巧克力到小碗中
Task_ Refund	9	等待退款事件，并退回所有金额	SemFlag2,SemMKey1,SemAmtCount	SemFinish,SemAmtCount，SemMKey1，消息指针 *Refund	在 Port_1、Port_2 和 Port_5 的硬币	刷回 Port_ Exit 的硬币
Task_ Excess Refund	11	退回剩余金额	SemFlag3,SemAmtCount,SemMKey1	SemFlag1,SemMKey1,SemAmtCount ,消息指针*Excess Refund	在 Port_ ExcessRefund 的硬币	(a) 在 Port_ Collect 的硬币 (b) 来自 Port_ Excess Refund 的硬币
Task_ Display	13	等待消息邮件	SemFlag4,SemMKey2，消息指针 *delivered、 *refund、 *ExcessRefund、 *timeDate	SemMKey2	第 1、2 和 3 行的字符串以及时间	显示到 LCD 阵列第 1、2 和 3 行的字符串
Task_ Time Date Display	14	通过计算系统时钟节拍数来更新时间和日期	在每 1000ms (1000 个系统时钟中断)后,得到一个来自 RTOS 的超时消息	消息指针 *timeDate	来自系统节拍的中断	时间日期的字符串

11.1.3 编码步骤示例

示例 11-1

```
1 /* Define Boolean variable as per example 4.5, define a NULL pointer to point
in case mailbox is empty.
Codes are as per Example 8.17 Step 1*/
typedef unsigned char int8bit;
#define int8bit boolean
#define false 0
#define true 1
/* Define a NULL pointer; */
#define NULL (void*) 0x0000;
/* Preprocessor commands define OS tasks service and timing functions as enabled
and their constants; similar to Example 10.7. */
#define OS_MAX_TASKS 10
#define OS_LOWEST_PRIO 20 /* Let lowest priority task in the OS be 15. */
#define OS_TASK_CREATE_EN 1 /* Enable inclusion of OSTaskCreate ( ) function
*/
#define OS_TASK_DEL_EN 1 /* Enable inclusion of OSTaskDel ( ) function */
#define OS_TASK_SUSPEND_EN 1/* Enable inclusion of OSTaskSuspend ( ) function
*/
#define OS_TASK_RESUME_EN 1/* Enable inclusion of OSTaskResume ( ) function */
.
.
/* Specify all child prototype of the first task function that is called by the
main function and is to be scheduled by MUCOS at the start. In step 11, we will
be creating all other tasks within the first task. */
/ * Remember: Static means permanent memory allocation */
static void FirstTask (void *taskPointer);
static OS_STK FirstTaskStack [FirstTask_StackSize];
/* Define public variables of the task service and timing functions */
#define OS_TASK_IDLE_STK_SIZE 100 /* Let memory allocation be for an idle state
task stack size be 100*/
#define OS_TICKS_PER_SEC 1000 / * Let the number of ticks be 1000 per second.
An RTCSWT will interrupt and thus tick every 1 ms to update counts. */
#define FirstTask_Priority 4 /* Define first task in main priority */
#define FirstTask_StackSize 100 /* Define first task in main stack size */
.
2. /* Preprocessor definitions for maximum number of inter-process events to
let the MUCOS allocate memory for the Event Control Blocks */
#define OS_MAX_EVENTS 24/* Let maximum IPC events be 24 */
#define OS_SEM_EN 1/* Enable inclusion of semaphore functions. */
#define OS_MBOX_EN 1/* Enable inclusion of mailbox functions for the mailing
of the mes-sage pointers to Task_Display. */
#define OS_Q_EN 1/* Enable inclusion of queue functions for sending the string
pointers to LCD matrix Port_Display * /
/* End of preprocessor commands */
3. /* Prototype definitions for seven tasks for steps 1 to 12 above. */
static void Task_ReadPorts (void *taskPointer);
static void Task_ExcessRefund (void *taskPointer);
static void Task_Deliver (void *taskPointer);
```

```
static void Task_Refund (void *taskPointer);
static void Task_Collect (*taskPointer);
static void Task_Display (void *taskPointer);
static void Task_TimeDateDisplay (void *taskPointer);
/* Definitions for seven task stacks. */
static OS_STK Task_ReadPortsStack [Task_ReadPortsStackSize];
static OS_STK Task_ExcessRefundStack [Task_ExcessRefundStackSize];
static OS_STK Task_DeliverStack [Task_DeliverStackSize];
static OS_STK Task_RefundStack [Task_RefundStackSize];
static OS_STK Task_CollectStack [Task_CollectStackSize];
static OS_STK Task_DisplayStack [Task_DisplayStackSize];
static OS_STK Task_TimeDateDisplayStack [Task_TimeDateDisplayStackSize];
/* Definitions for seven task-stack sizes. */
#define Task_ReadPortsStackSize 100 /* Define task 2 stack size*/
#define Task_ExcessRefundStackSize 100 /* Define task 3 stack size*/
#define Task_DeliverStackSize 100 /* Define task 4 stack size*/
#define Task_RefundStackSize 100 /* Define task 5 stack size*/
#define Task_CollectStackSize 100 /* Define task 3 stack size*/
#define Task_DisplayStackSize 100 /* Define task 1 stack size*/
#define Task_TimeDateDisplayStackSize 100 /* Define task 1 stack size*/
4. /* Definitions for seven task-priorities. */
#define Task_ReadPortsPriority 5 /* Define task 1 priority */
#define Task_ExcessRefundPriority 11 /* Define task 5 priority */
#define Task_DeliverPriority 8 /* Define task 3 priority */
#define Task_RefundPriority 9 /* Define task 4 priority */
#define Task_CollectPriority 7 /* Define task 2 priority */
#define Task_DisplayPriority 13 /* Define task 6 priority */
#define Task_TimeDateDisplayPriority 14 /* Define task 7 priority */
5. /* Prototype definitions for the semaphores. */
OS_EVENT *SemFlag1; /* On interrupt signals from Port_1 to Port_5, Task_ReadPorts
starts running. This flag needed when using semaphore for inter-process
communication between tasks reading Port_1, Port_2 and Port_5 and port,
Port_Collect. Also posted by Task_ExcessRefund to deliver the chocolate also
while refunding the excess. */
OS_EVENT *SemFlag2; /* Needed when using semaphore as flag for inter-process
communi-cation between Task_ReadPorts and amount sending task, Task_Refund. */
OS_EVENT *SemFlag3; /* Needed when using semaphore as flag for inter-process
communi-cation between Task_ReadPorts and Task_ExcessRefund. */
OS_EVENT *SemFlag4; /* Needed when using semaphore as flag for inter-process
communi-cation between Task_ReadPorts and displaying task Task_Display. */
OS_EVENT *SemAmtCount; /* Needed when using semaphore for passing the counts
as the number of characters read as a 16-bit message between Task_ReadPorts and
other tasks, collecting or excess refunding the amount. */
OS_EVENT *SemChocoCostVal; /* Needed when using semaphore as cost value for
choco-late. When SemAmtCount reaches cost value or exceeds this, the
inter-process communication
from Task_ReadPorts initiates. */
OS_EVENT *SemMKey1; /* Needed when using semaphore as resource key by
Task_ReadPorts and Task_ExcessRefund */
OS_EVENT *SemMKey2; /* Needed when using semaphore as resource key by
Task_ReadPorts and Task_ExcessRefund */
OS_EVENT *SemVal; /* Needed when using semaphore for passing the 16-bit message
```

```
be-tween steps b and c. */
6. /* Prototype definitions for the mailboxes and queues. */
OS_EVENT *MboxStr1Msg; /* Needed when using mailbox message between steps 1 and
12.
*/
OS_EVENT *MboxStr2Msg; /* Needed when using mailbox message between steps 1 and
12.
*/
7. /* Codes as per codes in Example 4.5 (except the main function codes). These
are for reading from Port_1, 2 and 5. */
OS_EVENT *MboxStr3Msg; /* Needed when using mailbox message between steps 1 and
12.
*/
OS_EVENT *MboxTimeDateStrMsg; /* Needed when using mailbox message from
Task_TimeDateDisplay */
OS_EVENT *QMsgPointer; /* Needed when using mailbox message between steps 1 and
12.
void *QMsgPointer [QMessagesSize]; /* Let the maximum number of message-pointers
at the queue be QMessagesSize. */
OS_EVENT *QErrMsgPointer; /* Needed when using mailbox message between steps
1 and
12 */
void *QErrMsgPointer [QErrMessagesSize]; /* Let the maximum number of error
message-pointers at the queue be QErrMessagesSize. */
9. /* Define both queues array sizes. */
#define QMessagesSize = 64; /* Define size of message-pointer queue when full
*/
#define QErrMessagesSize = 16; /* Define size of error message-pointer queue
when full */
SemFlag1 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using
it as an event flag*/
SemFlag2 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using
it as an event flag*/
SemFlag3 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using
it as an event flag*/
SemFlag4 = OSSemCreate (0) /* Declare initial value of semaphore = 0 for using
it as an event flag*/
SemChocoCostVal = OSSemCreate (8) /* Declare initial value of semaphore = 8 as
an event flag*/
SemChocoCostVal = OSSemCreate (8) /* Declare initial value of semaphore = 8 as
an event flag*/
SemAmtCount = OSSemCreate (0) /* Declare initial value of semaphore = 0 on reset
of the machine. */
SemMKey1 = OSSemCreate (1) /* Declare initial value of semaphore = 1 for using
it as a resource key for SemAmtCount using sections in the tasks. */
SemMKey2 = OSSemCreate (1) /* Declare initial value of semaphore = 1 for using
it as a Display resource key*/
SemVal= OSSemCreate (0) /* Declare initial value of semaphore character = '0'
*/
/* Declare initial counts as 0 as a counter that shows the number of times a
task, which sends into a buffer that stores a character stream, ran minus the
number of times the task which used the character from the stream ran from the
```

```
buffer */
SemAmtCount = OSSemCreate (0);
/* Create Mailboxes for the tasks. */
/* The following are required for three lines of LCD matrix at Port_Display when
using mailbox message from the tasks to the Task_Display. */
MboxStr1Msg = OSMboxCreate (NULL);
MboxStr2Msg = OSMboxCreate (NULL);
MboxStr3Msg = OSMboxCreate (NULL);
MboxTimeDateStrMsg = OSMboxCreate (NULL); /* For message from
Task_TimeDateDisplay to Task_Display. */
9. /* Any other OS Events for the IPCs. */
.
.
10. /* The codes similar to the codes in Example 4.5, except the main function
codes. These are for reading from Port A and storing a character. Here, we have
three ports, Port_1, Port_2 and Port_5 for Rs. 1, 2 and 5 denomination coins.
These are basically device driver codes for port_1, port_2 and port_5 and three
status flags for resetting to the beginning. */
STAF _1 = 0;
STAF _2 = 0;
STAF _3 = 0;
.
.
11. /* Start of the codes of the application from Main. Note: Code steps are
similar to Steps 9 to
17 in Example 10.16 */
.
.
void main (void) {
12. /* Initiate MUCOS RTOS to let us use the OS kernel functions */
OSInit ( );
13. /* Create first task, FirstTask that must execute once before any other.
Task creates by defining its identity as FirstTask, stack size and other TCB
parameters. */
OSTaskCreate (FirstTask, void (*) 0, (void *) &FirstTaskStack FirstTask_StackSize],
FirstTask_Priority);
/* Create other main tasks and inter-process communication variables if these
must also ex-ecute at least once after the FirstTask. */
15. /* Start MUCOS RTOS to let us RTOS control and run the created tasks */
OSStart ( );
/* Infinite while-loop exits in each task. So there is no return from the RTOS
function OSStart ( ). RTOS takes the control forever. */
16. }/ *** End of the Main function ***/
/* The codes of the application first task that main created. */
17. static void FirstTask (void *taskPointer){
18. /* Start Timer Ticks for using timer ticks later. */
OSTickInit ( ); /* Function for initiating RTCSWT that starts ticks at the
configured time in the MUCOS configuration preprocessor commands in Step 1 */.
19. /* Create seven Tasks defining by seven task identities, Task_TimeDateDisplay,
Task_Display, Task_ReadPorts, Task_ExcessRefund, Task_Deliver and Task_Refund
and the stack sizes, other TCB parameters. */
OSTaskCreate (Task_Display, void (*) 0, (void *) & Task_DisplayStack
```

```
[Task_DisplayStackSize], Task_DisplayPriority);
OSTaskCreate (Task_TimeDateDisplay, void (*) 0, (void *) &
Task_TimeDateDisplayStack [Task_TimeDateDisplayStackSize],
Task_TimeDateDisplayPriority);
OSTaskCreate (Task_ReadPorts, void (*) 0, (void *) & Task_ReadPortsStack
[Task_ReadPortsStackSize], Task_ReadPortsPriority);
OSTaskCreate (Task_ExcessRefund, void (*) 0, (void *) & Task_ExcessRefundStack
[Task_ExcessRefundStackSize], Task_ExcessRefundPriority);
OSTaskCreate (Task_Deliver, void (*) 0, (void *) & Task_DeliverStack
[Task_DeliverStackSize], Task_DeliverPriority);
OSTaskCreate (Task_Refund, void (*) 0, (void *) & Task_RefundStack
[Task_RefundStackSize], Task_RefundPriority);
OSTaskCreate (Task_Collect, void (*) 0, (void *) & Task_CollectStack
[Task_CollectStackSize], Task_CollectPriority);
20. while (1) { /* Start of the while loop*/
21. /* Suspend with no resumption later the First task, as it must run once only
for initiation of timer ticks and for creating the tasks that the scheduler
controls by preemption. */
OSTaskSuspend (FirstTask_Priority); /*Suspend First Task and control of the RTOS
passes forever to other tasks, waiting their execution*/
22. } /* End of while loop */
23. } /* End of FirstTask Codes */
/
*****************************************************************************/
/*** RTOS schedules the task functions below and never returns the control to
the first task.
***/
24. /* The codes for the Task_ReadPorts redefined to use the key, flag and 16-bit
value and mailbox*/
static void Task_ReadPorts (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that
execute once only.
The variable, i used for counting the 1's at the 8 points of a port and j is
to wait for one more cycle after first if SemAmtCount is less than the
SemChocoCostVal. */
static unsigned byte i, j;
static unsigned char *port_1data, *port_2data, *port_5data;
static void port_1_ISR_Input (*port_1data); /* Declare function to call to
retrieve port_1data.
*/
static void port_2_ISR_Input (*port_2data); /* Declare function to call to
retrieve port_2data.
*/
static void port_5_ISR_Input (*port_5data); /* Declare function to call to
retrieve port_5data.
*/
25. while (1) { /* Start an infinite while-loop */
/* Take the event flag SemFinish > 0 and if so decrement it Wait forever till
0. Take mutex for critical section. Variable SemAmtCount is used by other tasks
also. */
OSSemPend (SemFinish, 0, & SemErrPointer);
OSSemPend (SemMKey1, 0, & SemErrPointer);
```

```
26. /* Post Two mails to waiting Task_Display through two Message pointers for
first line and second line of LCD matrix at Port_Display. */
OSMboxPost (MboxStr1Msg, "Welcome to sweat memorable chocolate"); OSMboxPost
(MboxStr2Msg, " 'Insert amount, please");
/* Set initial amount counts to SemAmtCount. */
SemAmtCount = 0;
OSSemPost (SemFlag4); /* Release flag for Task_Display. */
27. /* Device driver Codes for Port_1, Port_2 and Port_5 and three status flags
will set. */
/* Let us give time for the child to insert the coins in 10000 ms (10 s) and
mechanical subsystem distribute the child coins to the Port_1, Port_2 and Port_5.
*/
/* Wait for STAF_1 and STAF_2 and STAF_5 setting to 1 within a time limit and
wait for SemAmtCount > = SemChocoCostVal*/
j = 0;
while (j < 2 | | (STAF_1 != 1   | STAF_2 != 1 | | STAF_5 != 1) && SemAmtCount
< SemChocoCostVal) {/* Wait for 10 s in first cycle and 10 s in second cycle.
*/
OSTimeDly (10000); *port_1data = 0; *port_2data = 0; *port_2data = 0;
/* Refer to Example 4.5 Wait till an interrupt occurs and sets the STAF flags
at Rs. 1 and Rs. 2 and Rs. 5 ports. */
28. /* Execute Port_1 interrupt service routine and find the amount received
at the Port_1. */
if (STAF_1 = = 1){Port_1_ISR_Input (&port_1data);}; /* Remember as soon as Port_1
is read STAF_1 will reset itself to reflect next interrupt status. */
/* Count the number of points having Rs. 1 coin. */
for (i = 0; i < 8; i++) {
If (*port_1data & 0x01) {SemAmtCount ++; *port_1data >> SemChocoCostVal; i++;};
STAF_1 = 0;}; /* Reset again Port_1 for next interrupt. */
29. /* Execute Port_2 interrupt service routine and find the amount received
at the Port_2. */
if (STAF_2 = = 1){Port_2_ISR_Input (&port_2data);}; /* Remember as soon as Port_2
is read STAF_2 will reset itself to reflect next interrupt status. */
/* Count the number of points having Rs. 2 coin. */
for (i = 0; i < 8; i++) {
If (*port_2data & 0x01) {SemAmtCount = SemAmtCount +2; port_2data >>
SemChocoCostVal;
i++;};
STAF_2 = 0;}; /* Reset again Port_2 for next interrupt. */
30. /* Execute Port_5 interrupt service routine and find the amount received
at the Port_5. */
if (STAF_5 = = 1){Port_5_ISR_Input (&port_5data);}; /* Remember as soon as Port_5
is read STAF_5 will reset itself to reflect next interrupt status. */
/* Count the number of points having Rs. 5 coin. */
for (i = 0; i < 8; i++) {
If (*port_5data & 0x01) {SemAmtCount = SemAmtCount +5; port_5data >>
SemChocoCostVal; i++;};
STAF_5 = 0; };/* Reset again Port_5 for next interrupt. */
j++;
};
31. /* Posts the semaphore flags for the actions as per the amount */
if (SemAmtCount = = SemChocoCostVal) {OSSemPost (SemFlag1);}; /* Post semaphore
```

```
to a task waiting Task_Collect. */
if (SemAmtCount < SemChocoCostVal) {OSSemPost (SemFlag2);}; /* Post semaphore
to a task waiting Task_Refund. */
if (SemAmtCount > SemChocoCostVal) {OSSemPost (SemFlag3);}; /* Post semaphore
to a task waiting Task_ExcessRefund. */
OSSemPost (SemAmtCount); /* Post counting semaphore for three waiting tasks.
*/
OSSemPost (SemMKey1); /* Release the resource key used for above critical section.
*/
enable_Port_1_Intr ( ); /* Prepare for another interrupt from port_1. */
enable_Port_2_Intr ( ); /* Prepare for another interrupt from port_2. */
enable_Port_5_Intr ( ); /* Prepare for another interrupt from port_5. */
OSTimeDly (5000); /* Let a low priority task execute as the next coin insertions
will take time.
The time 5 s is expected for the mechanical subsystems to deliver or refund or
refund excess. */
32. }; /* End of while loop*/
}/ * End of the Task_ReadPorts function */
/
*****************************************************************************/
33. /* Start of Task_Collect codes */
static void Task_Collect (void *taskPointer) {
34. /* Initial Assignments of the variables and pre-infinite loop statements
that execute once only*/
.
.
35. while (1) { /* Start an infinite while-loop /
/* Take the SemFlag1, SemMKey1 and accept SemAmtCount. */
OSSemPend (SemFlag1, 0, & SemErrPointer);
36. /* Post two mails to waiting Task_Display through two Message pointers for
fist line and second line of LCD matrix at Port_Display. */
OSMboxPost (MboxStr1Msg, " Wait for a moment "); OSMboxPost (MboxStr2Msg, "
Collect a nice Chocolate soon ");
37. /* Codes for device drivers for Port_Collect to collect the coins from Port_1,
Port_2 and Port_5. */
.
.
}
OSTimeDly (3000); /* Let task Deliver lower in priority deliver in 3 s, expected
time for mechanical subsystem to deliver. */
38. }; /* End of while loop*/
39. } / * End of the Task_Collect function */
/
*****************************************************************************/
40. /* The codes for the Task_Deliver */
static void Task_Deliver (void *taskPointer) {
41. /* The initial assignments of the variables and pre-infinite loop statements
that execute once only*/
.
.
42. while (1) { /* Start an infinite while-loop. */
43. /* Wait for flag SemDeliver from Task_Collect */
```

```
OSSemPend (SemDeliver, 0, & SemErrPointer);
44. /* Codes for device driver for Port_Deliver for delivering a chocolate into
a bowl. */
.
.
45. /* Post two mails to waiting Task_Display through two message pointers for
fist line and second line of LCD matrix at Port_Display. */
OSMboxPost (MboxStr1Msg, "Collect the nice chocolate"); OSMboxPost (MboxStr2Msg,
"Insert coins for more");
OSSemPend (SemMKey1, 0, & SemErrPointer);
OSSemAccept (SemAmtCount, 0, & SemErrPointer);
46. /* Reset SemAmtCount. Exit critical section*/
SemAmtCount =0; OSSemPost (SemMKey1);
47. /* Let delayed higher priority task err resume. */
OSTimeDlyResume (Task_CollectPriority);
OSTimeDlyResume (Task_ReadPortsPriority)
48./* Post semaphore to flag that the chocolate delivery is over. */
OSSemPost (SemFinish);
}; /* End of while loop*/
49. }/ * End of the Task_Deliver function */
/
*****************************************************************************/
50. /* Start of Task_Refund codes */
static void Task_Refund (void *taskPointer) {
/* Initial assignments of the variables and pre-infinite loop statements that
execute once only*/
.
.
51. while (1) { /* Start the infinite loop */
OSSemPend (SemFlag2, 0, & SemErrPointer);
52. /* Code for the device driver to let Port_Exit release the coins as refund
as within two cycles
of timeouts, if child fails to insert the coins of the required amount. */
.
.
53. /* Post two mails to waiting Task_Display through two message pointers for
fist line and second line of LCD matrix at Port_Display. */
OSMboxPost (MboxStr1Msg, "Collect the nice chocolate"); OSMboxPost (MboxStr2Msg,
"Insert coins for more");
OSSemPend (SemMKey1, 0, & SemErrPointer);
OSSemAccept (SemAmtCount, 0, & SemErrPointer);
54. /* Reset SemAmtCount. Exit critical section*/
SemAmtCount =0; OSSemPost (SemMKey1);
OSSemPost (SemFinish); /* Return to Task_ReadPorts. */
55. /* Let delayed higher priority task err resume. */
OSTimeDlyResume (Task_ReadPortsPriority) /* Resume Delayed Task_ReadPorts. */
}; /* End of while loop*/
56. } / * End of the Task_Refund function */
/
*****************************************************************************/
57. /* Start of Task_ExcessRefund codes */
static void Task_ExcessRefund (void *taskPointer) {
```

```
58. /* Initial assignments of the variables assignments and pre-infinite loop
statements that execute once only*/
.
.
59. while (1) { /* Start the infinite loop */
OSSemPend (SemFlag3, 0, & SemErrPointer);
OSSemPend (SemMKey1, 0, & SemErrPointer);
OSSemAccept (SemAmtCount, 0, & SemErrPointer);
60. /* Reset SemAmtCount. Exit critical section*/
SemAmtCount =SemAmtCount -8; OSSemPost (SemMKey1);
61. /* Code for the device driver to let Port_ExcessRefund release the coins
from a channel as per the SemAmtCount value now. AVCS thus refund the coins within
two cycles of timeouts, if child fails to insert the coins of the required amount.
*/
.
.
62. /* Post two mails to waiting Task_Display through two message-pointers for
the fist line and second line of LCD matrix at Port_Display. */
OSMboxPost (MboxStr1Msg, "Collect the nice chocolate"); OSMboxPost (MboxStr2Msg,
" 'Insert coins for more");
63. /* Also deliver the chocolate now after refunding the excess amount. */
OSSemPost (SemFlag1); /* Run Task_Collect, after Task_Deliver run. */
}; /* End of while loop*/
64. }/ * End of the Task_ExcessRefund function */
/
**************************************************************************/
65. /* The codes for the Task_Display */
static void Task_CharCheck (void *taskPointer) {
66. /* Declare string variables for the three lines and other initial assignments
and pre-infinite loop statements that execute once only. */
unsigned char Str1 [ ];
unsigned char Str2 [ ];
unsigned char Str3 [ ];
unsigned char Str3Right [ ]; /* A variable to display at the right corner of
line 3 of LCD Matrix
*/
.
.
67. /* Start an infinite while-loop. */
while (1) {
OSSemPend (SemFlag4, 0, & SemErrPointer); /* Wait for the flag for Display */
OSSemPend (SemMKey2, 0, & SemErrPointer); /* Take access to Display resources.
*/
/* Wait for Messages for line 1, line 2 and line 3. */
Str1= OSMboxAccept (MboxStr1Msg, 0, & MboxErrPointer);
Str2 = OSMboxAccept (MboxStr1Msg, 0, & MboxErrPointer);
Str3 = OSMboxAccept (MboxStr3Msg, 0, & MboxErrPointer);
Str3Right = OSMboxAccept (MboxTimeDateStrMsg, 0, & mboxErrPointer);
68. /* Device driver Codes for sending the four strings to a byte stream from
line 0 character first to last character line through Port_Display. For using
the queues for byte stream, refer to thecodes in Example 8.20. */
```

```
  .
  OSSemPost (SemMKey2);
  69. }/ * End of While loop
  }; /* End codes for the Task_Display */
  /
  ************************************************************************/
  70. /* The codes for the Task_TimeDateDisplay */
  static void Task TimeDateDisplay (void *taskPointer) {
71. /* Initial assignments of the variables and pre-infinite loop statements
that execute once only. */
  unsigned char *timeDate;
  .
  .
  72. /* Start an infinite while-loop. */
  while (1) {
73. /* Codes for creating a message for time and date after each 1000th interrupt
from the system RTC tick. */
  .
  .
  74. /* Post time and date to Task_Display.*/
  OSMboxPost (MboxTimeDateStrMsg, timeDate);
  75. }/* End of Codes for Task_TimeDateDisplay */
/
************************************************************************/
```

11.2 使用 RTOS VxWorks 将应用层字节流发送到 TCP/IP 网络上

嵌入式系统是当前在“数据通信和网络”领域使用最广的系统之一。RTOS 被广泛地应用在嵌入式系统当中。本节描述一个在这些应用领域中使用 VxWorks RTOS 的研究案例。解释下面的案例时，我们假定读者已经熟悉网络基础知识，尤其是 TCP/IP 和其他一些重要的协议(可以参考由 Tata McGraw-Hill 公司出版的，本书作者所著的 *Internet and Web Technologies* 一书。它描述了位(bit-wise)格式的 TCP 段、UDP 数据报、IP 包，以及 SLIP 和 Ethernet 帧)。

通信可以发生在同一系统内部，或者通过网络设备连接的远程系统之间。通信也可以从点到点(peer-to-peer)，或者在客户端和服务器之间进行。图 8-8 给出了两个套接字使用堆栈通信的函数。网络上，套接字 A 通常是客户端套接字，套接字 B 通常是服务器套接字。每个网络套接字使用一个参数对，即 IP 地址和端口号来标识。不论是什么应用程序，使用什么操作系统，两个应用程序之间的套接字间通信都是这样。

VxWorks 提供了网络用的 API。它有一个库 sockLib，用于套接字编程(参见 8.3.5 节)。由于我们的目标是通过对一个案例的研究，学习多任务 RTOS 中 IPC 的使用，所有这里没有使用这些 API 和 sockLib 创建 TCP 栈(在使用这些 API 和 sockLib 时，可以参考《VxWorks 网络程

序员指南》。该指南源自 Wind River 系统(www.wrs.com))。

假设在一个嵌入式系统的案例中，RTOS 任务和来自应用程序的 TCP 栈通信。应用程序的例子有 HTTP 或者 FTP。它们分别同 Web 服务器通信，或者传送文件。TCP/IP 栈是一种堆栈，它包含在网络上使用 TCP/IP 协议套件的通信帧。应用层字节流在后续各层被套上不同的格式，从而得到最终能在网络上使用的堆栈。字节流的格式依据的是中间层的协议规范。TCP 栈一般有许多帧，网络驱动器可以向其中写入字节信息。示例 11-2 和 11.2.3 节介绍了写入堆栈的 RTOS 应用。这里我们使用 10.3 节中的 VxWorks 函数描述当前的案例。在另外一个节点上(网络上的端点)，接收堆栈并检索应用程序输入字节的任务的优先级显然是逆序排列。它的编码作为练习留给读者。

在这个例子中，使用多任务的优势会变得相当明显。首先，应用程序可以不用连续输出字符串。所以在这期间，其他层的任务可以同时并发处理。当对 I/O 字节流使用全局变量时，信号量如何用作互斥量来保护共享数据？信号量如何用作事件标志，来提供在各种优先级的任务之间实现同步的有效途径？这个例子表明，使用 VxWorks 或者其他任何经过测试的 RTOS，可以简化任务调度的编码，并且 RTOS 中 IPC 函数当前也处于就绪可用的状态。

11.2.1　案例定义、多任务及其函数

图 11-3(a)给出了一个应用程序的子系统，该应用程序的内容是传输 TCP/IP 栈。图 11-3(b)给出了 TCP/IP 栈传输期间的任务调度序列。

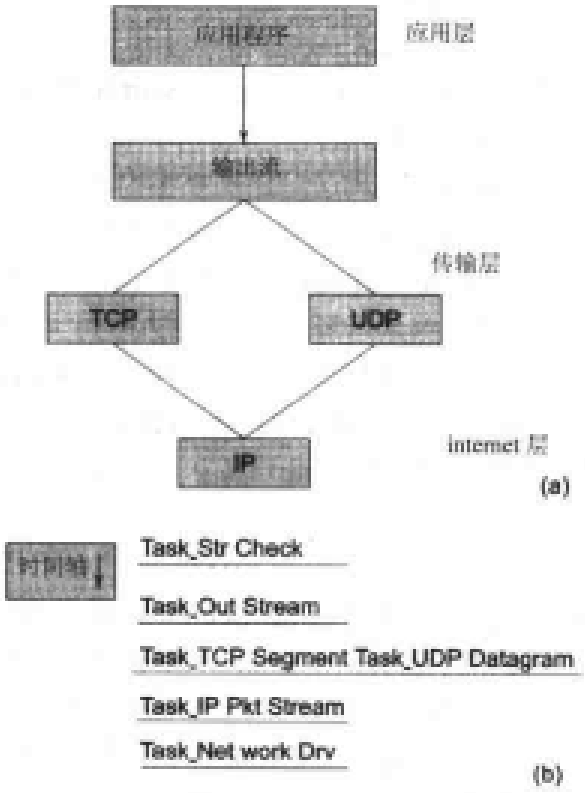


图 11-3　(a)传输 TCP/IP 栈的应用子系统；(b)TCP/IP 栈传输期间的任务及其调度序列

图 11-4 详细说明了各个任务之间同步的模型。在 TCP/IP 网络中，顶层是 “应用” 层。

步骤 A：任务 Task_StrCheck 检查应用程序的输出中字符串的可用性。该任务是在向网络上发送数据时优先级最高的任务。如果检查通过状态标志信号量 STAF 显示了字符串可用，则该任务提供(发送)信号量给等待任务 Task_OutStream。字节流的最大长度记为 maxSizeStream。

步骤 B：等待任务在得到信号时，取消阻塞。然后读取字符串，并将字节流送到消息队列。在来自应用程序的字符串到达缓冲队列 OutStream 之前，任务得到的信号量记为 SemFlag2。Task_StrCheck 提供的信号量和 Task_OutStream 得到的信号量记为 SemFlag1。第 2 个任务使用消息队列，OutStream 将来自应用程序的字符串发送给下一个等待任务。OutStream 为 QStreamInputID 所标识的消息队列提供字节信息。

步骤 C：在 TCP/IP 网络中，应用层的下一层是传输层。它等同于 OSI 模型中的传输层。该层任务的优先级仅次于上一层。除了 STAF 之外，应用程序任务还发送两个其他的信号量标志。一个标志的用途是取消阻塞等待任务 Task_TCPSegment。如果该层的协议是 TCP，则该任务得到信号量后取消阻塞。另外一个标志的用途是取消阻塞另外一个等待任务 Task_UDPDatagram。当这一层的协议是 TCP 时，它取消阻塞。

步骤 D：必须在传输层的字节流前面插入适当的头信息，使字节流的格式变成 TCP 段，或者 UDP 数据报，到底是哪种格式取决于取消阻塞(转为运行状态)的任务是 Task_TCPSegment 还是 Task_UDPDatagram。任务将每 256 字节(即 blkSize = 256 字节)的块放进队列。这样做是必要的，因为如果将字节放进队列，则 RTOS 在任务间调用上下文切换的时间将增加到总开销当中。因此总的执行时间就会增加，中断延时也会增加(参见 4.6 节)。

步骤 E：同块比较起来，TCP 段，或者 UDP 数据报都太长。在后面代码的模拟运行期间，块大小必须优化。较大块会让另外一个任务一直等待，直到做好发送整块的准备。而另一方面，较小块则会增加任务切换的开销和中断延时。该任务有一个临界段。头字节在队列前面插入临界段，并保证此时没有其他任何任务会向队列插入字节。互斥量 SemKMey1 通过资源锁定保护临界段。TCP 任务和 UDP 任务得到的信号量分别记为 SemTCPFlag 和 SemUDPFlag。令任务向 QStreamInputID 标识的消息队列提供块。

步骤 F：传输层的下一层是“internet”层(这里是一个小写的 i，而非大写。表示的含义的网络间(inter-networking)，而不是 Internet)。它等同于 OSI 模型中的网络层。任务 Task_IPPktStream 首先等待，当来自 Task_TCPSegment 或者 Task_UDPDatagram 的块可用时取消阻塞。每个报文最大长度为 2^{16} 字节。该层任务的优先级仅次于上一层。Task_IPPktStream 构成 IP 报文，通过网络驱动器在网络上传输。该任务向套接字字节流 SocketStream 中插入(写入)IP 报头，从而得到网络套接字。在从报文的上层堆栈中接收到数据块之后，它向流中插入头信息。该任务准备好向网络驱动器提交的报文时，发送信号量标志。最后向 QPktInputID 所标识的消息队列发送报文。

步骤 G：internet 层的下一层是“网络接口”层。它等同于 OSI 模型中的数据链路层。网络驱动任务 Task_NetworkDrv 等待报文就绪标志 SemPktFlag。当任务取消阻塞时，读取 SocketStream，并将帧 frame 写入管道 pipeNetStream。管道存储并传输帧头 frameHeader、SocketStream 数据或者它的分段数据，以及末尾字节 trailBytes 中的字节。后者通常用作错误控制函数或者帧结束(中止)函数。该任务向 pipeNetStream 标识的管道逐个提供帧。当没有其他可用于传输的字节时，则将该任务提供的，以及后面应用程序得到的信号量记为 SemFinishFlag。当一块字节已经就绪时，该任务提供信号量 SemFlag2。它让下一层的任务取消阻塞 outStream，并初始化报文的格式，而不管下一次 RTOS 何时调度它。

SLIP、PPP、Ethernet 和令牌环是网络接口层协议的例子。frameHeader 和 trailBytes 的格式取决于网络驱动器使用的协议。某些协议不会写入任何 frameHeader，例如 SLIP。而某些协议则不会写入 trailBytes。

Task_NetworkDrv 打开配置文件。配置文件是嵌入式系统中的一个虚拟文件设备(虚拟文件设备是指文件不在物理磁盘中，而只在内存中，它和访问磁盘文件一样可以打开、读写和关闭函数)。该文件的作用是指定配置信息。在 VxWorks 中，函数 creat ( )(提示：这个函数名中没有 create 末尾的“e”)和 remove ( )用于创建和打开一个文件设备。VxWorks 的打开和读写函数在示例 8-26 中给出了。对文件操作的 select ( )和 close ( )函数，以及对管道操作的 select ( )和 close ( )函数是类似的(参见 10.3.4(12))。

文件中的配置信息可以如此：在串连的情况下，定义下面这些参数：

(i) 连接协议(SLIP 或者 PPP)

(ii) 主机

(iii) 端口

(iv) 波特率(Baud rate)

(v) 每个字符的位数，例如 8

(vi) 中止位的位数，例如 1

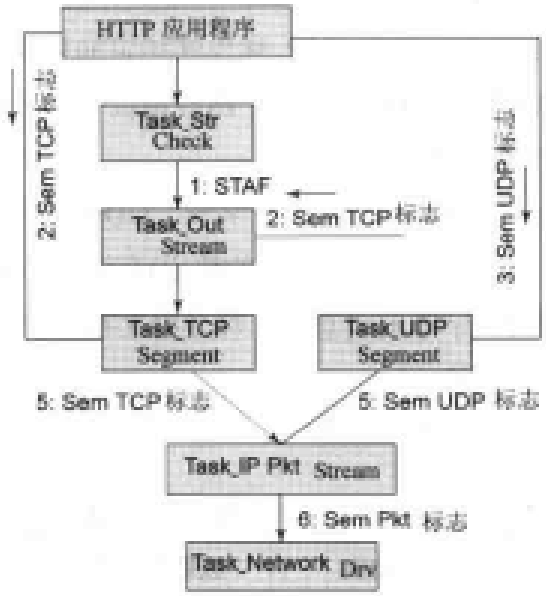


图 11-4　任务同步模型

在使用以太网卡作为网络驱动器的情况下，配置文件可以按以下方式描述配置信息。第一行的格式可以以 net 开头。它表示该行用于指定网卡地址。然后接下来的 3 个词是“Ethernet 3COM 0xXYZ”。它表示网络是以太网，网卡是 3COM 公司生产的，系统中的卡地址是 16 进制数 XYZ。下面一行的格式可以以 IP 开头。它表示这一行的下一个字应该定义 IP 地址。地址是连接网络的节点。如果在同一网络中存在其他的连接，则可能有另外一行用来分配另外的 IP 地址(IP 地址由 4 个字节构成。对于到所有节点都是广播连接的情况，IP 地址是 0xFFFFFFFF。它也可以用传统的方式写作 255.255.255.255)。

11.2.2 创建任务、函数和 IPC

设计代码的程序员首先应该准备一个列表，内容包括任务函数、任务优先级、任务作用的层、任务的动作、每个任务或者代码段取消阻塞之前等待(获取)的 IPC、让其他等待的代码段，或者取消阻塞的任务而提供(发送)的 IPC，以及任务作为输出内容而发送的输出流。然后开始编码。表 11-2 列出了这些内容。以下几点是表中注解。

(1) 优先级必须在 100 以上，并按层的顺序排列。

(2) 设置一项约定。IPC 以初始字符“Sem”开头是指二进制信号量，以“SemM”开头是指互斥量。标志一个事件的 IPC 有“Flag”这 4 个字符。资源锁定的 IPC 有“Key”这 3 个字符。消息队列标识符以字符“Q”开头。管道标识符以“pipe”这 4 个字符开头。

(3) 应用程序任务提供 SemTCPFlag 或者 SemUDPFlag。具体提供哪一个依据传输控制层上使用的协议而定。IPC SemFinishFlag 将由应用程序或者任意其他任务获取。它标志网络驱动器将消息字符串放进管道的任务完成。该标志是一种应答通知。另外一个 IPCSemFlag2 将由 OutStream 获取作为应答通知，应答的内容是发送到驱动器的字符流已经成功放进管道，从而新的字符流可以送进输入端。

(4) TCP 报头不同于 UDP 报头。前者比较便于在网络上通过适当的参数交换，建立连接、网络流控制和管理，以及连接终端。因而 TCP 称作连接定向协议。后者是发送给接收器的一个简单的数据报消息。因而 UDP 是无连接协议。两种情况下任务指定的优先级相等。指定协议的任何标志可以由应用程序提供(发送)。

(5) 数据报是独立的消息流，最大长度为 2^{16} 字节。UDP 仅传送源和目标端口号、流长度以及“internet”层的校验和。套接字字节流(该层的输出)有源和目标端口号，以及 IP 地址。一个套接字的报文是最大长度为 2^{16} 字节的消息流。每个报文都有嵌入的 IP 报头。套接字则通过报头中的 IP 地址和端口号进行识别。

(6) 网络驱动器依据网络驱动配置形成帧。

表 11-2 示例 11-2 中使用的任务、函数和 IPC

任务函数	优先级	TCP/IP®中的层	动作	获取的 IPC	发送的 IPC	输出字节流
Task_Str Check	120	应用层	获得来自应用程序的字符串	–	SemFlag1	没有
Task_Out Stream	121	应用层	读取字符串，并将字节放进输出流	SemFlag1，SemFlag2	QStream InputID	Out Stream
Task_TCP Segment	122	传输控制层(传输层)	将 TCP 报头插入字符流	SemTCP-Flag，SemM-Key1，QStreamInputID	QStreamInputID，SemMKey1	Out Stream

(续表)

任务函数	优先级	TCP/IP@中的层	动作	获取的 IPC	发送的 IPC	输出字节流
Task_UDP DataGram	122	传输控制层(传输层)	将 UDP 报头插入作为数据报发送的字符流	SemUDP-Flag, SemMKey1, QStreamInputID	QStreamInputID, SemMKey1	Out Stream
Task_IPPkt Stream	123	internet(网络层)	形成 2^{16} 字节的报文	SemMKey1, QStreamInputID	SemMKey1, SemPktFlag, QPktInputID	Socket Stream
Task_Net workDrv	124	网络接口层(数据链路层)	以帧的方式发送报文	SemPktFlag, QPktInputID, SemMKey1	SemFinishFlag, SemFlag2	pipeNet Stream

@ 括号中是OSI模型中相应层的名称。

11.2.3 编码步骤示例

示例 11-2

```
1. # include "vxWorks.h" /* Include VxWorks functions. */
# include "semLib.h" /* Include semaphore functions library. */
# include "taskLib.h" /* Include multitasking functions library. */
# include "msgQLib.h" /* Include message queue functions library. */
# include "fioLib.h" /* Include file-device input-output functions library. */
# include "sysLib.c" /* Include system library for system functions. */
pipeDrv ( ); /* Install a pipe driver. */
# include "netDrvConfig.txt" /* Include network driver configuration file for
frame formatting protocol (SLIP, PPP, Ethernet) description, card
description/make, address at the system, IP addresss of the nodes that drive
the card for transmitting or receiving from the network. */
# include "prctlHandlers.c" /* Include file for the codes for handling and actions
as per the protocols used for driving streams to the network. */
2. sysClkRateSet (1000); /* Set system clock rate 1000 ticks per second. */
3. /* Initialise the socket parameters and other network parameters initial
values*/
/* SourcePort means source port number of the application used. DestnPort means
destination port
number. Define a variable string, Str. */
unsigned short SourcePort; unsigned short DestnPort; unsigned char [ ] Str;
unsigned short SourcePort = ;
unsigned short DestnPort = ;
unsigned short SourceIPAddr = ;
unsigned short DestnIPAddr = ;
4. /* Declare data types of Output Byte Streams for arguments in the tasks. */
unsigned char [ ] applStr, OutStream, socketStream, pipeNetStream;
```

```
5. /* Declare data types of the maximum sizes of streams from and to the tasks.
Declare data type of block size, blkSize. It is the number of bytes that must
be available first before sending an IPC to a buffering stream. It avoids repeated
switching from one task to the next after each byte. */
unsigned int blkSize, strSize, strSize, maxSizeOutStream, maxSizeSocketStream,
maxSizepipeNetStream;
6. /* Allocate Default Values to various sizes*/
maxSizeOutStream = 1024 * 1024; /* Let application put 1 MB on the net. */
unsigned int strSize = 1; /* Let default string size from an application be 1
byte. */
blkSize = 256; /* Let default block be 256 bytes. */
maxSizeSocketStream = 64 * 1024; /* Let Socket put 64 KB packet on the net. */
maxSizepipeNetStream = 16 * maxSizeSocketStream; /* Let network driver put 16
packets = 1 MB maximum number of bytes. */
7. /* Declare all Table 11.2 Task function prototypes. */
void Task_StrCheck (SemID SemFlag1); /*Task check for the string Availability.
*/
void Task_OutStream (SEM_ID SemFlag1, SEM_ID SemFlag2, MSG_Q_ID
QStreamInputID);
void Task_TCPSegment (SEM_ID SemTCPFlag, SEM_ID SemMKey1, MSG_Q_ID
QStreamInputID);
void Task_UDPDatagram (SEM_ID SemUDPFlag, SEM_ID SemMKey1, MSG_Q_ID
QStreamInputID, OutStream, SourcePort, DestnPort, maxSizeOutStream, blkSize);
void Task_IPPktStream (SEM_ID SemMKey1, SEM_ID SemPktFlag, MSG_Q_ID
QPktInputID);
void Task_NetworkDrv (SEM_ID SemMKey1, SEM_ID SemPktFlag, SEM_ID SemFinishFlag,
SEM_ID SemFlag2, MSG_Q_ID QPktInputID socketStream, pipeNetStream, blkSize,
maxSizeSocketStream, maxSizepipeNetStream, MSG_FRAME aFrame);
8. /* Declare all Table 11.2 Task IDs, Priorities, Options and Stacksize. Let
initial ID till spawned be none. No options and stacksize = 4096 for each of
six tasks. */
int Task_StrCheckID = ERROR; int Task_StrCheckPriority = 120; int Task_Str
CheckOptions = 0; int Task_StrCheckStackSize = 4096;
int Task_OutStreamID = ERROR; int Task_OutStreamPriority = 121; int Task_Out
StreamOptions = 0; int Task_OutStreamStackSize = 4096;
int Task_TCPSegmentID = ERROR; int Task_TCPSegmentPriority = 122; int
Task_TCPSegmentOptions= 0; int Task_TCPSegmentStackSize = 4096;
int Task_UDPDatagramID = ERROR; int Task_UDPDatagramPriority = 122; int
Task_UDPDatagramOptions = 0; int Task_UDPDatagramStackSize = 4096;
int Task_IPPktStreamID = ERROR; int Task_IPPktStreamPriority = 123; int
Task_IPPktStreamOptions = 0; int Task_IPPktStreamStackSize = 4096;
int Task_NetworkDrvID = ERROR; int Task_NetworkDrvPriority = 124; int
Task_NetworkDrvOptions = 0; int Task_NetworkDrvStackSize = 4096;
9. /* Create and Initiate (Spawn) all the six tasks of Table 11.2. */
Task_StrCheckID = taskSpawn (" tTask_StrCheck", Task_StrCheckPriority,
Task_StrCheckOptions,
Task_StrCheckStackSize, void (*Task_StrCheck) (SEM_ID STAF, SEM_ID SemFlag1),
0, 0, 0,0, 0, 0, 0, 0, 0, 0);
Task_OutStreamID = taskSpawn (" tTask_OutStream", Task_OutStreamPriority,
Task_OutStreamOptions,
Task_OutStreamStackSize, void (*Task_OutStream) (SEM_ID SemFlag1, MSG_Q_ID
QStreamInputID, applStr, OutStream, maxSizeOutStream, blkSize), 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0);
Task_TCPSegmentID = taskSpawn (" tTask_TCPSegment", Task_TCPSegmentPriority,
Task_TCPSegmentOptions, Task_TCPSegmentStackSize, void (*Task_TCPSegment)
(SEM_ID SemTCPFlag, SEM_ID SemMKey1, MSG_Q_ID QStreamInputID, unsigned char [ ]
OutStream, int maxSizeOutStream, int blkSize, unsigned char [16] txtcpState,
unsigned char [16]txtcpOpPdFormat) unsigned short SourcePort, unsigned DestnPort,
unsigned int SequenNum, unsigned int AckNum, unsigned char *TCPHdrLen, unsigned
*TCPHdrFlags, unsigned short *TCPChecksum16, unsigned short window , unsigned
short *UrgPtr, un-signed char [optPdLen] extras)
Task_UDPDatagramID = taskSpawn (" tTask_UDPDatagram", Task_UDPDatagramPriority,
Task_UDPDatagramOptions, Task_UDPDatagramStackSize, void (*Task_UDPDatagram)
(SEM_ID SemUDPFlag, SEM_ID SemMKey1, MSG_Q_ID QStreamInputID, unsigned char [ ]
OutStream, int maxSizeOutStream, int blkSize), SourcePort, DestnPort, 0, 0, 0,
0, 0, 0, 0, 0);
Task_IPPktStreamID = taskSpawn ("tTask_IPPktStream", Task_IPPktStreamPriority,
Task_IPPktStreamOptions, Task_IPPktStreamStackSize, void (*Task_IPPktStream)
(SEM_ID SemPktFlag, MSG_Q_ID QPktInputID, unsigned char [ ] OutStream, int
maxSizeOutStream,blkSize, unsigned char [ ] SocketStream, maxSizeSocketStream),
0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
Task_NetworkDrvID = taskSpawn ("tTask_NetworkDrv", Task_NetworkDrvPriority,
Task_NetworkDrvOptions, Task_NetworkDrvStackSize, void (* Task_NetworkDrv)
(SEM_ID SemMKey1, SEM_ID SemPktFlag, SEM_ID SemFinishFlag, SEM_ID SemFlag2,
MSG_Q_ID QPktInputID socketStream, unsigned char [ ] pipeNetStream, int blkSize,
int maxSizeSocketStream, int maxSizepipeNetStream, unsigned char[ ] frameHeader,
unsigned char[ ] trailBytes), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
10. /* Declare IDs and create the binary semaphore flags, keys and message queues.
*/
SEM_ID SemTCPFlag, SemUDPFlag; /* Declared at the application */
SemTCPFlag = semBCreate (SEM_Q_FIFO, SEM_EMPTY); /* Declared at the application
*/
SemUDPFlag = semBCreate (SEM_Q_FIFO, SEM_EMPTY); /* Declared at the application
*/
/* Note: The application posts wither SemUDPFlag or SemTCPFlag to let one of
the two tasks run. [Task_TCPSegment or Task_UDPDatagram. */
SEM_ID SemFlag1, SemFlag2, SemPktFlag, SemFinishFlag, SemMKey1; /* Declared for
the six tasks listed in Table 11.2. */
11. /* Create the binary semaphores, message queue for a stream of bytes from
an application and pass the options selected to it. */
SemFlag1 = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY); / *Task higher in priority
takes it first. */
SemFlag2 = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY); / *Task higher in priority
takes it first. */
SemPktFlag = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY); /*Task higher in priority
takes it first.*/
SemMKey1 = semBCreate (SEM_Q_FIFO, SEM_EMPTY); / *Taken in FIFO */
SemFinishFlag = semBCreate (SEM_Q_FIFO, SEM_EMPTY); / *Taken in FIFO */
MSG_Q_ID QStreamInputID;
void char [ ] msgStream; /* Pointer for the message buffer */
12. /* Create the message queue identity and pass the parameters and chosen
options to it. Let maximum number of messages be 256 KB and the message be of
1 byte each. An IP packet has a maximum 2^16bytes. Assume that a TCP segment stream
for transmitting has maximum of 256 KB. Let 64 bytes be additionally assigned
```

```
for the headers. Header bytes add at the lower layers (refer column 3 in Table
11.2). */
QStreamInputID = msgQCreate (maxSizeOutStream, strSize, MSG_Q_FIFO | MSG_PRI_NORMAL);
QPktInputID = msgQCreate (maxSizeSocketStream, strSize, MSG_Q_FIFO | MSG_PRI_NORMAL);
13. /* Steps 1 to 3 as per Example 8.26 for creating a pipe. */
.
.
14. /* Create pipe named as pipeNetStream for including overheads and frame
messages, each if 1 byte.
Overheads mean header bytes as well as trailing bytes. */
STATUS pipestatus;
pipestatus = pipeDevCreate ("/pipe/pipeNetStream", maxSizepipeNetStream, 1);
mode = 0 = 0x0;
15. /* Other declarations that are needed. */
.
.
16. /* Declare common functions needed in the networking tasks. */
/* Declare the functions to get 32-bit, 16-bit and 8-bit lengths from a stream
or string. */
unsigned int getLength32 (unsigned char [ ] Str) {
/* Codes for finding as an unsigned integer the length of a string or stream
up to 2^32 bytes * /
.
.
};
unsigned short getLength16 (unsigned char [ ] Str) {
/* Codes for finding, as an unsigned short, the length of a string or stream
up to 2^16 bytes. * /
.
.
};
unsigned byte getLength8 (unsigned char [ ] Str) {
/* Codes for finding, as an unsigned byte, the length of a string up to 256 bytes.
* /
.
.
};
17. /* Declare Codes for adding extra padding in shorter size message or string
or stream to fill 0s so that string size now equals block size, blkSize. */
unsigned char [ ] StrAddPadding (unsigned char [ ] Str, unsigned int blkSize) {
.
.
};
18. /* Exemplary codes for finding a function to get 16-bit checksum from a byte
stream or string. */ unsigned short checksum16 (unsigned char [ ] Str) {
/* Codes for finding, as an unsigned integer, the length of a byte stream or
string* /
/* StrPtr is pointer to the string or stream. For the while loop, 'i' is a 16-bit
integer. Number of carries generated = numCarry. */
static unsigned short *strPtr, i, numCarry;
static unsigned int sum = 0;
unsigned short num = (unsigned short) getLength32 (unsigned char [ ] Str); /*
```

```
num is 16-bit number for total number of bytes in the string. We are typecasting
the length data type to 16-bit */
/* A 16-bit Checksum method is as follows: Sum the 16-bit words and first count
the carries, which generate on successive additions. Then add these carries to
get the 16-bit checksum.
However, we are having the byte stream. So method is as follows: */
strPtr = (unsigned short) Str; /* Type cast address to 16-bit value pointer.
*/
i = num/2; /* Let us split the calculation into two parts because we are adding
the bytes instead of 16-bit words to get the checksum*/
while (i - -) {
sum += *strPtr ++; /* add the byte into the sum and then pointer to next byte.
*/
if (i & 1){sum += *(unsigned char *) strPtr;}; /* Sum the odd byte from next
address. */
while ((numCarry = (unsigned short) (sum >>16)) ! = 0) {sum = (sum & 0xFFFF)
+ numCarry);}
return ((unsigned short) sum); /* Type cast sum to 16- bit. */
};
/* Reader to write 32 bit checksum codes for function checksum32 by himself (or
herself). */
.
.
/* Declare a Function for Cycle Redundancy Check */
unsigned int CRC32 (unsigned int crc, unsigned char aByte) {
/* Codes for finding 32-bit cycle redundancy check bits as an unsigned integer
for the given message frame. * /
.
.
};
19. /* On a network, while we are sending the bytes a protocol uses the different
data types, such as unsigned int, unsigned short, unsigned char, etc. Hence,
the definitions of the following six functions to get the lower byte and higher
byte from a 16-bit short, data16 and to get four bytes byte0, byte1, byte2 and
byte3 from a 32-bit int, data32 is essential. */
unsigned char LByte (unsigned short data16) { … }; unsigned char HByte (unsigned
short data16) { … };
unsigned char byte0 (unsigned int data32) { … }; unsigned char byte1 (unsigned
int data32) { … };
unsigned char byte2 (unsigned int data32) { … }; unsigned char byte3 (unsigned
int data32) { … };
20. to 27./* Code for other declaration steps specific to the various networks
*/
.
.
/* End of the codes for creation of the tasks, semaphores, message queue, pipe
tasks, and variables and all needed function declarations */
/
**********************************************************************/
28. /* Start of Codes for Task_StrCheck*/
void Task_StrCheck (SEM_ID STAF, SEM_ID SemFlag1) {
```

```
.
29.
while (1) { /* Start of while loop. */
/* When character output is generated by the application, the semaphore is given.
*/
SemTake (STAF, WAIT_FOREVER); /* Wait for Status flag from Application. */
30. /* Codes for the task. *
.
.
semGive (SemFlag1);
semGive (SemMKey1); /* Release the mutex if any taken before. */
taskDelay (10); /* Delay 10 ms to let lower priority task run. */
}; /* End of while loop. */
} /* End of Task_StrCheck. */
31. /* Start of Codes for Task_OutStream. */
void Task_OutStream (SEM_ID SemFlag1, MSG_Q_ID QStreamInputID, applStr,
OutStream, maxSizeOutStream, blkSize) {
32. /* Initial assignments of the variables and pre-infinite loop statements
that execute once only*/ int numBytes;
.
.
33. while (1) { /* Start an infinite while-loop. We can also use FOREVER in place
of while (1). */
34. / *Wait for SemFlag1 state change to SEM_FULL by semGive function for string
availability check task */
semTake (SemFlag1, WAIT_FOREVER);
35. /* Take the key to not let any application other than the present task put
the message into the queue port decipher task that needs SemMKeyID run */
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available and the
critical region
starts */
36. /* Codes for Encrypting applStr if any or illegal character or message check,
if any */
.
.
37. /* At the end, send the byte input at OutStream as message to queue,
QStreamInputID. Refer to Section 10.3.4.(10) for understanding the msgQSend
function. NO_Wait if string is more than the block size 256 bytes. Otherwise
string is too short to deserve sending on the stream without padding it with
extra 0s. Call a function to add padding bytes. */
numBytes = getLength32 (applStr);
if (numBytes < blkSize) {StrAddPadding (applStr, blkSize); numBytes = blkSize};
/* Now wait if any of the previous bytes of applStr have not been put into the
socket streams.
Task_IPPktStream does that. It posts SemFlag2 on successfully sending the applStr
into the stream with two transport and internet layer headers. */
semTake (SemFlag2, WAIT_FOREVER);
msgQSend (QStreamInputID, applStr, numBytes, NO_WAIT, MSG_PRI_NORMAL);};
semGive (SemMKey1); /* Critical Region ends here. */}
38. /* Resume the delayed task, Task_StrCheck as message has been put into the
message queue. */ taskDelay (20)
taskResume (Task_StrCheckID);
```

```
. /* Other remaining codes for the task. */
.
.
};
39. } /* End of the codes for Task_OutStream. */
40. /* Start of Codes for Task_TCPSegment. */
void Task_TCPSegment (SEM_ID SemTCPFlag, SEM_ID SemMKey1, SEM_ID SemFlag2,
MSG_Q_ID QStreamInputID, OutStream, maxSizeOutStream, blkSize, unsigned char
[16] txtcpState, unsigned char [16] txtcpOpPdFormat) {
41. /* Initial assignments of the variables and pre-infinite loop statements
that execute once only*/
static int numBytes; /* Number of bytes successfully read. Equals error on timeout
or message queue ID not identified. */
static int timeout = 20; /* Let after 20 system clock-ticks 20 ms*/
unsigned char [16] txtcpState;
unsigned char [16] txtcpOpPdFormat;
unsigned char [ ] OptionAndPds (unsigned char [16] txtcpState, unsigned char
[16] txtcpOpPdFormat);
unsigned char [optPdLen] extras;
unsigned char [ ] Str, unsigned int SequenNum, unsigned int AckNum, unsigned
char *
TCPHdrLen, unsigned *TCPHdrFlags, unsigned short window, unsigned short*TCPChecksum16,
unsigned short *UrgPtr, unsigned char [optPdLen] extras);
static unsigned char [ ] header;
unsigned char [ ] TCPHeader (unsigned short SourcePort, unsigned short DestnPort,
unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat, unsigned char
[ ] Str, un-signed int SequenNum, unsigned int AckNum, unsigned char * TCPHdrLen,
unsigned *TCPHdrFlags, unsigned short window, unsigned short *TCPChecksum16,
unsigned short *UrgPtr, unsigned char [optPdLen] extras;
.
.
42. while (1) { /* Start task infinite loop. */
/* Take mutex so that till header is inserted into the stream, it is not released
to Task_OutStream. */
semTake (SemTCPFlag, WAIT_FOREVER);
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available. Wait for
entering critical region*/
43. /* Receive the message sent by Task_OutStream */
numBytes = msgQReceive (QStreamInputID, OutStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
44. /* Code for defining the txtcpState as per the connection status. */
.
.
/* Code for defining the SequenNum as per txtcpState */
.
.
/* Code for defining the AckNum as per txtcpState */
.
/* Code for defining the window as per txtcpState */
.
.
/* Code for defining the txtcpOpPdFormat as per txtcpState */
```

```
.
.
45. /* Codes for finding the additional integers as options and padding to be
put as the header. */
extras = OptionAndPds (txtcpState, txtcpOpPdFormat);
46. /* Codes for retrieving the TCP header bytes */
header = TCPHeader (SourcePort, DestnPort, txtcpState, txtcpOpPdFormat, OutStream,
SequenNum, AckNum, * TCPHdrLen, *TCPHdrFlags, window, *TCPChecksum16, *UrgPtr,
extras);
47. /* Send header into the front of the queue */
msgQSend (QStreamInputID, header, getLength32 (header), NO_WAIT, MSG_PRI_URGENT);
48. /* Send data to the back of the queue */
msgQSend (QStreamInputID, applStr, numBytes, NO_WAIT, MSG_PRI_NORMAL);
}; /* End of the message handling codes. */
semGive (SemMKey1); /* Critical region ends here. */}
semGive (SemTCPFlag); /* SemTCPFlag. */
taskDelay (20);
taskResume (Task_OutStream);
}; / * End of while loop. */
49. } /* End of codes for Task_TCPSegment */
50. /* Start of codes for Task_UDPDatagram */
void Task_UDPDatagram (SEM_ID SemUDPFlag, SEM_ID SemMKey1, SEM_ID SemFlag2,
MSG_Q_ID QStreamInputID, OutStream, maxSizeOutStream, blkSize) {
51. /* Initial assignments of the variables and pre-infinite loop statements
that execute once only*/ static int numBytes; /* Number of bytes successfully
read. Equals error on timeout or if message queue ID not identified. */
static int timeout = 20; /* Let after 20 system clock-ticks 20 ms*/
static unsigned char [8] header;
unsigned char [8] UDPHeader (unsigned short SourcePort, unsigned short DestnPort,
unsigned char [ ] OutStream);
.
.
52. while (1) { /* Start task infinite loop. */
/* Take mutex so that till header is inserted into the stream; it is not released
to Task_OutStream. */
semTake (SemUDPFlag, WAIT_FOREVER);
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available. Wait for
entering the critical region*/
53. /* Receive the message sent by Task_OutStream */
numBytes = msgQReceive (QStreamInputID, OutStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
54. /* Codes for retrieving the UDP header bytes */
header = UDPHeader (SourcePort, DestnPort, OutStream); 55. /* Send Header to
the front of the queue
*/
msgQSend (QStreamInputID, header, 8, NO_WAIT, MSG_PRI_URGENT);
56. /* Send Data to the back of the queue */
msgQSend (QStreamInputID, applStr, numBytes, NO_WAIT, MSG_PRI_NORMAL);
}; /* End of the message handling codes. */
/* Critical Region ends here. */
57. semGive (SemMKey1);
semGive (SemUDPFlag); /* SemUDPFlag release for use in new application string,
```

```
applStr. */
58. /* Let lower priority task, Task_IPPktStream start Higher priority one resume.
*/
taskDelay (20);
taskResume (Task_OutStream);
}; / * End of while loop. */
59. } /* End of codes for Task_UDPDatagram */
60. /* Start of Codes for Task_IPPktStream */
void Task_IPPktStream (SEM_ID SemFlag2, SEM_ID SemPktFlag, MSG_Q_ID QPktInputID,
int maxSizeOutStream, int blkSize, unsigned short SourceAddr, unsigned short
DestnAddr, unsigned short IPverHdrPrioSer, unsigned char [16] txipState,
unsigned char [16] txipOpPdFormat, unsigned char *timeToLive, unsigned char
*PrctlField) {
61. /* Initial assignments of the variables and pre-infinite loop statements
that execute once only*/
static int numBytes; /* Number of bytes successfully read. Equals error on timeout
or if message queue ID not identified. */
static int timeout = 20; /* Let after 20 system clock-ticks 20 ms*/
static unsigned char [8] header; unsigned char [ ] Str;
unsigned char [ ] IPHeaderSelPkt (unsigned short SourceAddr, unsigned short
DestnAddr, unsigned short IPverHdrPrioSer, unsigned short *IPPktLen, char *
IPVerHdrLen, unsigned char * IPHdrLen, unsigned char *IPHdrFlags, unsigned short
*IPHdrFrag, unsigned short *IPChecksum16, unsigned short UniqueID, unsigned
char *timeToLive, unsigned char *PrctlField, unsigned char [ ] Str, unsigned
char [ ] OutStream, char [optPdLen] IPextras, unsigned char [16] txipState,
unsigned char [16] txipOpPdFormat);
62. while (1) { /* Start task infinite loop. */
/* Take mutex so that till header is inserted into the stream; it is not released
to Task_OutStream.*/
semTake (SemMKey1, WAIT_FOREVER); /* SemMKeyID is now not available. Wait for
entering critical region*/
63. /* Receive the message sent by Task_OutStream */
numBytes = msgQReceive (QStreamInputID, OutStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
64. /* Codes for retrieving the IP Packet header bytes */
header = IPHeaderSelPkt (SourceAddr, DestnAddr, IPverHdrPrioSer, *IPPktLen, *
IPVerHdrLen, * IPHdrLen, *IPHdrFlags, *IPHdrFrag, *IPChecksum16, UniqueID,
*timeToLive, *PrctlField, Str, OutStream, IPextras, txipState, txipOpPdFormat);
65. /* Send Header into the front of the queue */
msgQSend (QPktInputID, header, * IPHdrLen, NO_WAIT, MSG_PRI_URGENT);};
66. /* Send data to the back of the queue */
numBytes = *IPPktLen - * IPHdrLen;
msgQSend (QPktInputID, Str, numBytes, NO_WAIT, MSG_PRI_NORMAL);};
/* Further send header and Str bytes into the queue if more IP packets are to
be sent */
.
.
}; /* End of the message handling codes. */
/* Critical Region ends here. */}
67. semGive (SemMKey1);
semGive (SemPktFlag); /* SemPktFlag release for use in Network Driver Task. */
68. /* Let lower priority task, Task_IPPktStream start Higher priority one resume.
```

```
*/
taskDelay (20);
taskResume (Task_UDPDatagramID);
taskResume (Task_TCPSegmentID);
}; / * End of while loop. */
69. } /* End of codes for Task_IPPktStream. */
70. /* Start of Codes for Task_NetworkDrv. */
void Task_NetworkDrv (SEM_ID SemMKey1, SEM_ID SemPktFlag, SEM_ID SemFinishFlag,
SEM_ID SemFlag2, MSG_Q_ID QPktInputID socketStream, unsigned char [ ] pipeNetStream,
int blkSize, int maxSizeSocketStream, int maxSizepipeNetStream, unsigned char
[ ] frameHeader, un-signed char [ ] trailBytes) {
71. /* Declare data type for specifying the headers. Let header length of frame
be length; type of network driver be netDrvType; and fragment offset at the stream
be frameOffset, which specifies the index in the byte array from which a frame
starts in case the stream is sent in fragments. */
unsigned char [ ] frame, frameHeader, trailBytes, fragment, trailBytes;
unsigned short fragOffset; unsigned short *FRlength, * HdrLen, * endLen,
*FrameHdrFlags, *FrameHdrFragOffset, *PrctlField;
unsigned int *FrameCRC32;
unsigned char [ ] Str, unsigned char [ ] SocketStream, unsigned char [ ] FRextras,
unsigned char [16]
txFRState, unsigned char [16] txFROpPdFormat, unsigned char [12] netDrvType
72. /* Declare Network Driver Function. */
void NetHdrFrTr (unsigned char [ ] frame, unsigned char [ ] frameHeader, unsigned
char [ ] trailBytes, unsigned short fragOffset, unsigned char [ ] fragment,
unsigned char [ ] trailBytes, unsigned short *FRlength, unsigned short * HdrLen,
unsigned short * endLen, unsigned short *FrameHdrFlags, unsigned short
*FrameHdrFragOffset unsigned short *FrameCRC32, unsigned short *PrctlField,
un-signed char [ ] Str, unsigned char [ ] SocketStream, unsigned char [ ] FRextras,
unsigned char [16] txFRState, unsigned char [16 ] txFROpPdFormat, unsigned char
[12] netDrvType);
72. /* Integers for number of bytes successfully read, message size and file
device, respectively. */ int numBytes, lBytes, fd;
73. /* Let a pointer netDrvConfig define a pointer to a configuration file for
a network driver. */ int fd; /* Define an integer for a file device. */
fd = open (netDrvConfig.txt, O_RDWR, 0); /* Refer Step 5 Example 8.26. */
74. /* Code for reading netDrvType, protocol for the link (SLIP or PPP), data
link layer format (say Ethernet), host IP, port, baudrate, card specifications,
etc. Use lBytes and function, read as follows. */ lBytes =12; /* Let the first
message bytes read = 12*/
numBytes = read (fd, Str, lBytes);
.
.
close (fd);
75. /* Open the pipe for the Network Stream. Refer Step 5 Example 8.26. */
fd = open ("/pipe/pipeNetStream", O_RDWR, 0);
76. while (1) {;
/* Task reads SocketStream on unblocking and writes the frame, frame into a pipe,
pipeNetStream. The pipe stores and transmits the bytes at the frame header,
frameHeader, and frame fragment from the SocketStream in case frame is of smaller
size than the SocketStream and at the end trailing bytes, railBytes. The latter
are usually for error control functions or frame terminal (end) functions. Let
```

```
the task give the frame one after another to a pipe identified by pipeNetStream.
This lets the next layer task unblock outStream and initiate the formation of
packets whenever the RTOS schedules it. */
semTake (SemPktFlag, WAIT_FOREVER);
semTake (SemMKey1, WAIT_FOREVER);
77. /* Receive the message sent by Task_IPPktStream */
numBytes = msgQReceive (QStreamInputID, SocketStream, maxSizeOutStream, 20);
if (numBytes != ERROR) {
78. /* Codes for retrieving the frame bytes */
NetHdrFrTr (frame, frameHeader, trailBytes, fragOffset, fragment, trailBytes,
*FRlength, * HdrLen, *endLen, *FrameHdrFlags, *FrameHdrFragOffset, *FrameCRC32,
*PrctlField, Str, SocketStream, FRextras, txFRState, txFROpPdFormat, netDrvType);
/* Write a message, info of lBytes. */
unsigned char [ ] info; /* Let the message be a string of characters. */
int lBytes;
lBytes =12;
write (fd, frame, *FRlength);
79. /* Further write into the pipe if more frames are to be sent */
.
.
; };
semGive (SemMKey1);
semGive (SemFinishFlag); /* Post the semaphore for next waiting task at the
application layer.
*/
semGive (SemFlag2); /* Post the semaphore for waiting task for sending an Out
Stream. */
taskResume (Task_IPPktStreamID); /* Resume the previously delayed higher
priority task. */
}; /* End of While-loop */
80. } /* End of codes for Task_NetworkDrv */
/* Start of codes for Task_NetworkDrv. */
/ *
*****************************************************************************/
```

文件 prcltHandlers.c 中有用于 UDP、TCP、IP 和其他网络协议的代码，这些网络协议在形成 IP 报文和帧字节时，包含并处理头函数，选择字节片段。代码设计如下：

```
1. /* Declare a Function for returning a UDP header string. DatagramLen means
length of UDPdatagram, which transmits to next layer. Str means the stream or
string from the application layer to be sent with UDP protocol. */
unsigned char [8] UDPHeader (unsigned short SourcePort, unsigned short DestnPort,
un-signed char [ ] Str) {
2. /* Codes for returning UDP Header String. Remember that UDP protocol transmits
the inte-gers with big endian. Refer to paragraph (1) in the instructions for
a hardware designer in Section 2.1. Big endian means the most significant bytes
transmit first. * /
unsigned char [8] UDPHStr;
unsigned short DatagramLen = getLength16 (Str) + 8; /* Add 8-byte header length
also. */
unsigned short UDPChecksum16 = checksum16 (Str);
UDPHStr [0] = HByte (SourcePort); UDPHStr [1] = LByte (SourcePort);
```

```
UDPHStr [2] = HByte (DestnPort); UDPHStr [3] = LByte (DestnPort);
UDPHStr [4] = HByte (DatagramLen); UDPHStr [5] =LByte (DatagramLen);
UDPHStr [6] = HByte (UDPChecksum16); UDPHStr [7] = LByte (UDPChecksum16);
return (UDPHStr);
};
/* ---------------------------------------------------------------------
----------/*
```

3. /* Declarations and codes for the function, TCPHeader for returning a TCP header string. *Str* means the stream from the application layer at a node, *node1* end to be sent with TCP protocol to other end, *node2*. */

```
/* Define the byte position (index) of the present OutStream bytes from the
application. Initial value =0*/
unsigned int SequenNum = 0;
/* Define the total number of bytes already received at an input stream from
node2 to this node1. Input stream is the one that was sent as TCP stacks and
received at the node1 from the receiver node2. This is to let an out stream
simultaneously convey to the other end an acknowledgement along with a new
sequence of bytes. OutStream and input stream synchronize and there is controlled
flow of bytes between the two ends, node1 and node2. Initial value = 0. */
unsigned int AckNum =0;
/* Declare 4-bit and 4- bit unused, reserved for future expansion. TCP headers
vary between 5 to 15
unsigned integers of 32-bit each. */
unsigned char *TCPHdrLen;
/* Let 16-character string txtcpState specify the state of transmission of the
current TCP segment and its action required for the controlled transmission.
Action is to be as desired. It may be to establish a connection, for termination
or management or flow control. Refer to TCP protocol in any standard text
for definitions. */
unsigned char [16] txtcpState;
/* Declare TCPHdrFlags for 6 bits for flags and 2 bits unused and reserved for
future modifications in protocol. Bit 0 is FIN, bit 1 is SYN, bit 2 is RST, bit
3 is PUSH, bit 4 is ACK and bit 5 is URG. */
unsigned char *TCPHdrFlags;
/* Let another 16-character string txtcpOpPdFormat specify format, which gives
the number and mean-ing of optionally added integers and padded integers. For
example, an optional integer may be to specify an alternative window of 32 bits
in place of 16 given in the 16-bit window field at fourth integer in the TCP
header. Another option integer may specify the TCP maxSizeOutStream. */
unsigned char [16] txtcpOpPdFormat;
```

4. /* Start of the codes for returning a *short*. It specifies the 4-bit TCP Header length field as well as six-flag fields. These are as per transmission state and transmitting TCP options and padding format, *txtcpState* and *txtcpOpPdFormat*, respectively */

```
unsigned short TCPHdlenFlagBits (unsigned char [16] txtcpState, unsigned char
[16] txtcpOpPdFormat, &TCPHdrFlags, &TCPHdrLen) {
Boolean bit 15, bit 14, bit 13, bit 12, bit 11, bit 10, bit 9, bit 8;
```

5. /* Codes to have four bits, bit 15, bit 14, bit 13, bit 12 in the returned integer as per the TCP header size, which specifies the total number of unsigned integers in the TCP header. The total is 5 plus the unsigned integers for options and padding. These are as per txtcpState and txtcpOpPdFormat used by TCP segment that is transmitting. The bits, bit 11, bit 10, bit 9, bit 8 are reserved. */

```
/* Bits 0 to 5 are as per six flags. Bits 6 to 11 are reserved. Bits 0 to 7 are
at unsigned character pointer TCPHdrFlags. */
*TCPHdrFlags = *TCPFlags (txtcpState); /* Using the function find bits 0 to 7, */
.
.
6. /* Code to find TCPHdrLen from bit 15, bit 14, bit 13, bit 12. */
.
.
  }; /* End of the codes for returning an integer that specifies TCP Header length
  and flag fields.
*/
7. /* Declare a function TCPFlags to return TCPHdrFlags as per txtcpState. Start
of the codes for finding the byte to represent the TCP flag fields. */
unsigned char *TCPFlags (unsigned char txtcpState) {
Boolean FIN, SYN, RST, PUSH, ACK, URG, bit 6, bit 7;
/* Codes to create as per txtcpState in which TCP segment is transmitting. */
.
.
  }; /* End of the codes for returning the pointer for a byte for the TCP flags
  field. */
8. /* Declare other TCP Header fields. */
unsigned short window; /* node1 specifies by window as an advertisement to node2
how many more bytes at node1 can be buffered in its buffer beyond the ones already
buffered and acknowledged by node1 to node2. Buffering is an intermediate state
in which bytes are still to be sent to an upper application layer by a TCP stack
receiving entity. Node 2 if finds window = 0 or too small a number, then it should
not send any thing to this node1. This 16-bit field then is a request to the
other end node not to flood data bytes on the network unnecessarily. It is
then indirectly a request to wait. */
unsigned short *TCPChecksum16; /* Define 16-bit Checksum. */
/* Define a 16-bit Urgent pointer. */
unsigned short Urgent;
/ *A 16-bit value that will indicate an urgency to node2 whenever the URG flag
is set. It indicates the first byte of the start of the urgent data from node1.
It is a request to the node2 that it should consider the pointed bytes first
in place of attending to the bytes in the buffer and the bytes after this segment
header. The buffered data and beginning data may be ignored and bytes beginning
from UrgPtr are requested to be given urgency. Given or not depends on node2
program task at TCP segment that sends the bytes to its application layer. */
unsigned short *urgPtr = Urgent;
unsigned short *TCPURGENT (unsigned char txtcpState, &TCPHdrFlags, *urgPtr) {
Boolean URG;
URG = getFlag (&TCPHdrFlags) {
/ * Codes to extract the URG bit. It is bit 5 of byte at address of TCPHdrFlags.
.
  }
/* .If URG is true, then set the urgent field and return a 16-bit short. Codes
to return a 16-bit
short as urgent pointer as per txtcpState in which TCP segment is transmitting.
*/
if (URG) {
.
```

```
. };
};
unsigned char [ ] extras; unsigned char OptPdLen;
extras = OptionAndPds ( txtcpState, txtcpOpPdFormat);
optPdLen = getLength8 (extras);
*TCPHdrLen = (optPdLen + 20) /4 ;
unsigned char [ ] OptionAndPds (unsigned char [16] txtcpState, unsigned char
[16] txtcpOpPdFormat) {
9. /* Codes for returning an array of bytes for the 32-bit integers for the options
and padding. These are
as per the State and thus TCP header length parameters. */
.
.
  };
10. /* Codes for returning TCP Header String. Remember that TCP protocol transmits
the integers with big endian. Refer to Section 2.1.2. (i). It means that the most
significant byte should transmit first. [Refer to Section 2.1.2. (i) also]. * /
/* Note: Instead of using many arguments, a typedef could have been used for
the header data struc-ture.However, this will consume more memory. */
unsigned char [ ] TCPHeader (unsigned short SourcePort, unsigned short DestnPort,
unsigned char [16] txtcpState, unsigned char [16] txtcpOpPdFormat, unsigned char
[ ] Str, unsigned int SequenNum, unsigned int AckNum, unsigned char * TCPHdrLen,
unsigned *TCPHdrFlags, unsigned short win-dow, unsigned short *TCPChecksum16,
unsigned short *UrgPtr, unsigned char [optPdLen] extras) {int i; unsigned char
[ ] TCPHStr; unsigned short lenflag; unsigned char optPdLen;
unsigned short TCPChecksum16 = checksum16 (Str); unsigned short *urg; unsigned
char [ ] extras;*urg = *TCPURGENT (unsigned char txtcpState, &TCPHdrFlags,
*urgPtr)TCPHStr [0] = HByte (SourcePort); TCPHStr [1] = LByte (SourcePort);
TCPHStr [2] = HByte(DestnPort); TCPHStr [3] = LByte (DestnPort); TCPHStr [4]
= Byte0 (SequenNum); TCPHStr [5] = Byte1(SequenNum); TCPHStr [6] = Byte2
(SequenNum); TCPHStr [7] = Byte3 (SequenNum);
TCPHStr [8] = Byte0 (AckNum); TCPHStr [9] = Byte1 (AckNum); TCPHStr [10] = Byte2
(AckNum);TCPHStr [11] = Byte3 (AckNum);
*lenflag = * TCPHdlenFlagBits (unsigned char [16] txtcpState, unsigned char [16]
txtcpOpPdFormat&TCPHdrFlags, &TCPHdrLen);
TCPHStr [12] = HByte (lenflag); TCPHStr [13] = LByte (lenflag);
TCPHStr [14] = HByte (window); TCPHStr [15] = Byte3 (window);
TCPHStr [16] = HByte (TCPChecksum16); TCPHStr [17] = LByte (TCPChecksum16);
TCPHStr [18] = *Urg++); TCPHStr [19] = *(unsigned char *) urg);
extras = OptionAndPds (unsigned char [16] txtcpState, unsigned char [16]
txtcpOpPdFormat);
optPdLen = getLength8 ( extras);
*TCPHdrLen = (optPdLen + 20) /4;
while (optPdLen > = ++i) {TCPHStr [i] = extras [i - 20];}; /* Fill the options
and padding bytes. */
unsigned short TCPSegLen = getLength16 (Str) + (optPdLen + 20); /* Add byte of
header length also.
*/
return (TCPHStr);
};
/* ------------------------------------------------------------------/*
/* Declarations and codes for the function, IPHeaderSelPkt for returning an IP
```

header string and the packet data in socketStream from OutStream. *Str* means the stream from transmission control layer at a node, *node1*, end to be sent with header as per IP protocol to other end, *node2*, through the network driver, switches, bridges and routers. */

11. /* First let us declare IPVerHdrLen for 4 bits (bit 7, bit 6, bit 5 and bit 4) for the version number. Presently, IP version 4 is mostly used. So these bits are 0100. Another four bits (bit 3, bit 2, bit 1 and bit 0) are the numbers of unsigned integers in the header. Header length includes 5 unsigned standard integers and 0 to 10 integers for options and padding. The latter integers are used for controlling the path and flow through the routers. Some of these integers may also be used for adding network security. Options may be recording of the route of packet, time stamp on the packet, source router IP address to used before routing through a common source, any flexible source option, security option, etc. Details can be found in the classic work of *D. E. Comer and D. Stevens, Internetworking with TCP/IP Vol.1, Principles, Protocols and Architecture from Prentice Hall, NJ, 1995*. Usually the options and padding are not present. Then the four lower bits are 0101. */

unsigned char * IPVerHdrLen;

/* Header Length as per four upper bits in IPVerHdrLen. */

unsigned char ***IPHdrLen;**

12. /* Let us assume that the options and padding are as per the format specified by a 16-characters string, txipOpPdFormat. It represents the number and meanings of optionally added integers and padded integers. */

unsigned char [16] ***txipOpPdFormat;***

13. /* Let a 16-character string txipState specify the protocol of the current IP segment and its action required for the controlled transmission. Protocol can be ICMP, IGMP, OSPF, EGP and BGP. Refer to this author's book *Internet and Web Technologies from Tata McGraw-Hill* for definitions and any standard text for details of these protocols. */

unsigned char [16] ***txipState;***

14. /* Start of the codes for returning a *short*. It specifies 4-bit version field plus 4-bit IP Header length field in * IPVerHdrLen followed by 3-bit specifying precedence of the TP packet on the network plus 5 bits for the type of service. Let the latter 8 bits be at the address pointed by IPPrioServ. Precedence bits, '000' means usual precedence on the Internet. '111' means highest precedence, the one needed for streaming the audio and video on the net service bits for quality of service to be provided in terms of security, speed, delayed or cost to be charged from the sender. The 16-bit integer returned by function, IPVerHdrPrioServBits is thus as per version and IP packet transmission state and transmitting IP options and padding format, *txipState* and *txipOpPdFormat*, respectively. */

unsigned char * IPPrioServ;

unsigned short **IPVerHdrPrioServBits** (unsigned char [16] *txipState*, unsigned char [16] *txipOpPdFormat*, &IPPrioServ, &IPHdrLen) {

15. /* Returned integer bit 15, bit 14, bit 13, bit 12, bit 11, bit 10, bit 9 and bit 8 are as per 8 bits at IPHdrLen. Three precedence bits, bit 7, bit 6 and bit 5 in the returned integer are as per the IP precedence and service bits bit 4, bit 3, bit 2, bit 1 and bit 0 are as per QOS (quality of service specified in *txipState*. */

Boolean bit15, bit14, bit13, bit12, bit11, bit10, bit9, bit8, bit7, bit6, bit5, bit4, bit3, bit2, bit1, bit0;

/* The codes to find character at IPPrioServ from *txipState*. */

```
.
.
16. /* Code to find IPHdrLen from txipOpPdFormat */
.
.
 }; /* End of the codes for returning the integer that specifies IP version,
 Header length, precedence and service. */
17. /* Two 16-bit short integers for packet length and packet identification
by receiver. */
unsigned short IPPktLen; /* Specify the length of IP Packet */
unsigned short UniqueID; /* Specify the UniqueID of the present IP Packet This
is put by the router or the transmitter, node1. It uniquely identification for
the packet routed. This will help the receiver to reassemble the fragments of
this IP Packet at node2 for upward transmission to transport and application
layer there. Note: A fragment may be lost on the net in between routers due to
some error popping in. This helps in recovering the lost fragment. */
18/* A stream from OutStream may be bigger than 6536 bytes minus the header bytes
in the IP packet. Therefore, it is to be fragmented. Fragmentation is also
necessitated when the network driver and other units in-between do not permit
even an IP packet of 65536 bytes and need shorter frames. Function, IPFlagFragBits
Codes is for returning 3 flag bits and 13 fragment-offset bits. The offset
specifies fragment number. Besides precedence and QOS, txipState specifies what
3 flag bits and 13 fragment-field bits should be defined by this function. */
unsigned char *IPHdrFlags; unsigned short *IPHdrFrag;
/* Function for finding a stream of bytes actually been put with the packet.
*/
void selectPktData (unsigned char [ ] Str, unsigned char OutStream, unsigned
char [16] txipState, unsigned char [16] txipOpPdFormat, unsigned char [ ] Str)
{
.
.
};
unsigned short IPFlagFragBits (unsigned char [16] txipState, unsigned char [16]
txipOpPdFormat,
&IPHdrFlags, &IPHdrFrag), char [ ] OutStream) {
Boolean bit15, bit14, bit13, bit12, bit11, bit10, bit9, bit8, bit7, bit6, bit5,
bit4, bit3, bit2, bit1, bit0;
19. /* Codes to have three bits, bit 15, bit 14 and bit 13 in the returned integer
as per the flags. A flag is mfb. mfb = 0 means that the receiver should wait
as there are more fragments to follow this fragment. Mfb = 0 for last fragment.
It refers to the IP header size, which specifies the total number of unsigned
integers in the IP header. The total is 5 plus the unsigned integers for options
and padding. These are as per txipState and txipOpPdFormat used by IP segment
that is transmitting. The bits, bit 11, bit 10, bit 9, bit 8 are reserved. */
/* Bits, bit 12 down to bit 0 are as fragment offset bits and as per already
transmitted bytes from network driver task at node1. */
/* Code to find character IPHdrFlags from bit 15, bit 14, and bit 13. The bits
are as per txipState.
.
```

```
.
20. /* Code to find 16-bit short integer IPHdrFrag from bit 13 down to bit 0.
These bits are as per txipState specification, OutStreamSize and the bits already
transmitted by the network driver. IPHdrFrag = i means OutStream byte numbering
(8)i is being sent with this packet. */
.
.
}; /* End of the codes for returning an integer that specifies IP Header flags
and fragment-offset bits. */
21. /* Declare other header field, 16-bit Checksum, 8-bit time to live and 8-bit,
protocol filed, source nodel IP address and destination node 2 IP address. */
unsigned short *IPChecksum16;
unsigned char *timeToLive; /* Note: It decrements at each router on the way to
node2. */
unsigned char *PrctlField; / Note: PrctlField = 17 for UDP, = 6 for TCP, = 1
for ICMP.
unsigned short SourceIPAddr;
unsigned short DestnIPAddr;
unsigned char [ ] extras; unsigned char OptPdLen;
IPextras = OptionAndPds (txipState, txipOpPdFormat);
optPdLen = getLength8 (IPextras);
*IPHdrLen = (optPdLen + 20) /4;
unsigned char [ ] IPOptionAndPds (unsigned char [16] txipState, unsigned char
[16] txipOpPdFormat)
{
22. /* Codes for returning an array of bytes for the 32-bit integers for the
options and padding. These are as per the State and thus IP header length
parameters.
.
.
};
23. /* Codes for returning IP Header String. Remember that IP protocol transmits
the integers with big endian. * /
/* Note: Instead of using many arguments, a typedef could have been used for
the header data structure. However, this will consume more memory. */
unsigned char [ ] IPHeaderSelPkt (unsigned short SourceAddr, unsigned short
DestnAddr,unsigned short IPverHdrPrioSer, unsigned short *IPPktLen, char *
IPVerHdrLen, unsigned char * IPHdrLen, unsigned char *IPHdrFlags, unsigned short
*IPHdrFrag, unsigned short *IPChecksum16, unsigned short UniqueID, unsigned
char *timeToLive, unsigned char *PrctlField, unsigned char [ ] Str, unsigned
char OutStream, char [optPdLen] IPextras, un-signed char [16] txipState,
unsigned char [16] txipOpPdFormat) { int i; unsigned char [ ] IPHStr; unsigned
short lenflag; unsigned char optPdLen; unsigned short new;
24. /* Code for finding a stream of bytes actually been put with the packet.
*/
new = IPVerHdrPrioServBits (txipState, txipOpPdFormat, &IPPrioServ, IPHdrLen);
IPHStr [0] = HByte (new);
IPHStr [1] = LByte (new);
```

```
25. /* Codes for estimating assigning UniqueID to the packet. * /
.
.
IPHStr [4] = HByte (UniqueID);
IPHStr [5] = LByte (UniqueID);
26. /* Codes for assigning Time to live and Protocol field from txipState. */
.
.
IPHStr [8] = *timeToLive; IPHStr [9] = *PrctlField;
IPHStr [12] = byte0 (SourceAddr); IPHStr [13] = byte1 (SourceAddr);
IPHStr [14] = byte2 (SourceAddr); IPHStr [15] = byte3 (SourceAddr);
IPHStr [16] = byte0 (DestnAddr); IPHStr [17] = = byte1 (DestnAddr);
IPHStr [18] = = byte0 (DestnAddr); IPHStr [19] = = byte0 (DestnAddr);
IPextras = IPOptionAndPds (txipState, txipOpPdFormat); /* Find Options and
Padding. */
optPdLen = getLength8 (IPextras);
*IPHdrLen = (optPdLen + 20) /4;
while (optPdLen > = ++i) {IPHStr [i] = IPextras [i - 20];}; /* Fill the options
and padding bytes.*/
27. /* Codes for selecting the socket stream data to be sent and then estimating
IPPktLen / selectPktData (Str, OutStream, txipState, txipOpPdFormat, IPextras);
unsigned short *IPPktLen = getLength16 (Str) + (optPdLen + 20); /* Add byte of
header length also. */
IPHStr [2] = HByte (IPPktLen); IPHStr [3] = LByte (IPPktLen);
new = IPFlagFragBits (txipState, txipOpPdFormat, &IPHdrFlags, &IPHdrFrag, Str)
IPHStr [6] = HByte (new); IPHStr [7] = LByte (new);
unsigned short IPChecksum16 = checksum16 (IPHStr); /* Find checksum of IP header
part only. */
IPHStr [10] = HByte (IPChecksum16); IPHStr [11] = LByte (IPChecksum16);
return (IPHStr);
};
/* Declarations and codes for the function, NetHdrFrTr for returning a Frame
header string, frameHeader and the fragment data in frame [ ] array from
SocketStream. Str means the stream from internet layer at a node, node1, end
to be sent with header as per data link protocol and card protocol to other end,
node2, through the pipe having byte streams from the network driver. */
void NetHdrFrTr (unsigned char [ ] frame, unsigned char [ ] frameHeader, unsigned
char [ ] trailBytes, unsigned short fragOffset, unsigned char [ ] fragment,
unsigned har [ ] trailBytes, unsigned short *FRlength, unsigned short * HdrLen,
unsigned short * endLen, unsigned short *FrameHdrFlags, unsigned short
*FrameHdrFragOffset unsigned short *FrameCRC32, unsigned short *PrctlField,
unsigned char [ ] Str, unsigned char [ ] SocketStream, unsigned char [ ] FRextras,
unsigned char [16] txFRState, unsigned char [16 ] txFROpPdFormat, unsigned
char [12] netDrvType) {
unsigned char [ ] frame, frameHeader, trailBytes, fragment, trailBytes; unsigned
short fragOffset; unsigned short *FRlength, * HdrLen, * endLen, *FrameHdrFlags,
*FrameHdrFragOffset *FrameCRC32;
unsigned int CRC32 (unsigned int crc, unsigned char aByte);
```

```
/* Codes as per netDrvType, transmitting frame state txFRState, transmitting
frame option and padding format, txFROpPdFormat create an array of characters
for the frameHeader, for the fragment and for trailBytes. Each is of length,
HdrLen, fraglen, endLen, FRlength. Other fields calculated are short integer
for fragOffset and unsigned integer for frame CRC bits, FRCRC32, etc. */
.
.
}
/*****************************************************************/
```

11.3 汽车自适应巡航控制系统的嵌入式系统

嵌入式系统是当前汽车电子领域应用最广泛的系统之一。从 1990 年起，就已经在汽车的离散独立组件或者集成中心服务系统的下述功能中使用。

(1) 任务中使用的 RTC 和 watchdog 定时器(参见 3.2 节)。

(2) 用于所有电子、电机和机械系统的实时控制(参见 9.4 节)。

(3) 维持恒定速度的自适应巡航控制(Adaptive Cruise Control，ACC)。现在可能增加了一项特性，即在高速公路上出现多汽车流的情况下，维持车队稳定(string stability)的功能(车队稳定是指维持车间距离不变)。

(4) 数据采集系统(Data Acquisition System，DAS)如下：

a. 当前时间和日期的显示、更新，故障状况和正常工作时间的记录。更新也可以周期性地通过来自广播发射台的信号来实现时间同步。

b. 外部温度。

c. 内部温度。

d. 里程计和 ACC 上覆盖的距离总和。

e. ACC 和速度计上以千米/小时为单位的道路速度和警告速度。

f. 以 r.p.m 为单位的引擎速度(转/分钟)。

g. 冷却剂温度，温度超过 115℃和维持在 80℃~90℃之间的时间。

h. 显示面板和汽车内部的光照度。

i. 油量(空、R1、R2、R3、1/4、3/8、1/2、5/8、3/4、7/8 或者满)。

j. 油压表，启动引擎速度超过 5000r.p.m 时的油压提醒和超过 15000r.p.m 时的报警系统。

k. 当前耦合的齿轮信息。

l. 前方汽车的距离。

(5) 面板切换和显示控制。

(6) 发出报警信号的端口。这是一个显示面板和可以发出嘟嘟声的发声系统。通过统计图表的显示，以及启动发声报警和对诊断分析的适当记录来发出警告信号(统计图表是图像数据文件中的一个预记录图片。它显示在 LCD 阵列上。统计图表依据相应的消息显示图片)。

(7) 启动故障时间统计和技工快速失效诊断分析结果的诊断计算。

(8) 多媒体接口。

(9) 控制区域网络接口。

(10) 串行通信接口(SCI)、传输器和接收器。

行驶控制是一个负责控制主动轮油门，并以预定的恒定速度引导汽车行驶的系统。行驶控制也可以在高速公路上出现多汽车流，或者护送贵宾的情况下保持车队稳定(参见 Chi-Ying Liang 和 Huei Peng 的论文 *Optimal Adaptive Cruise Control with Guaranteed String Stability*，摘自 *Vehicle System Dynamics* 杂志，31，pp. 313-330，1999)。通常，驾驶员在驾驶期间，使用加速器踏板控制车速。但是导航控制系统解除了驾驶员的这项工作，并且在道路条件比较适宜(没有水和冰，或者没有强风和大雾)时，ACC 可以代替手动控制。当交通不太拥挤时，汽车以比较高的速度行驶。在需要的时候，驾驶员也可以恢复控制。

自适应控制是指一种调整控制输入的当前状态的算法，该算法的方程中使用的不是一组恒定不变的数学参数，而是可以动态调整的。需要不断调整的参数示例有比例常量、积分常量和微分常量。ACC 系统很早就用于航空电子仪器和防御战斗机的导航。在汽车领域的应用最近才兴起(参见 http://www.ee.surrey.ac.uk/Personal/R.Young/java/html/cruise.html)。

图 11-5 介绍了自适应控制算法如何进行调整和操作。算法计算控制信号的输出值(对于控制系统算法的细节，读者可以参考控制设计的标准文档，即 John F.Dorsey 编著的 *Continuous and Discrete Control Systems*，McGraw-Hill 国际版，2002，Madan Gopal 编著的 *Digital Control and State Variable Method*，Tata McGraw-Hill 出版社，新德里，1997，以及 Walter J.Grantham 和 Thomas L.Vincent 合著的 *Modern Control Systems–Analysis and Design*，John Wiley & Sons 出版社，纽约，1993)。

自适应控制的主要组件

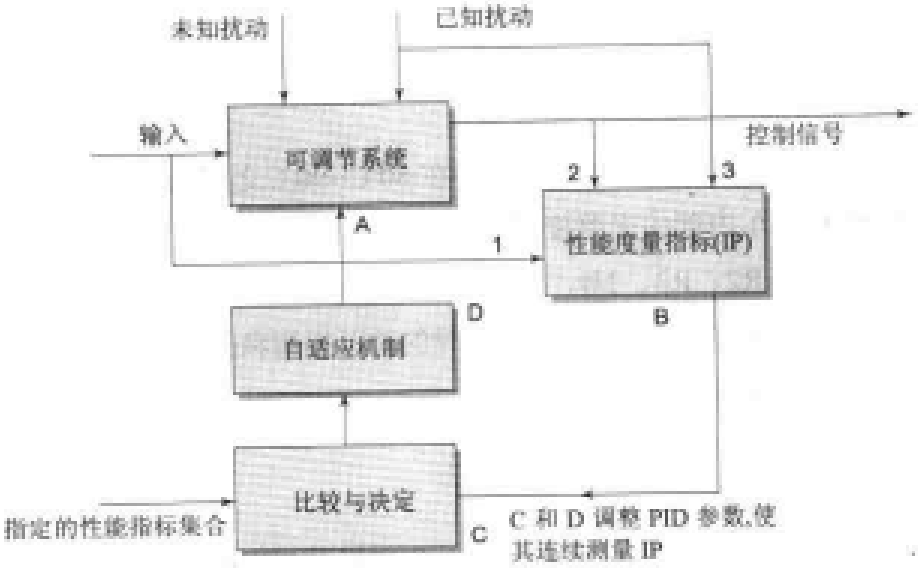


图 11-5 调整操作的自适应控制算法模型

图 11-6 给出了 ACC 嵌入式系统的结构图。常见的 ACC 系统执行下列行为。

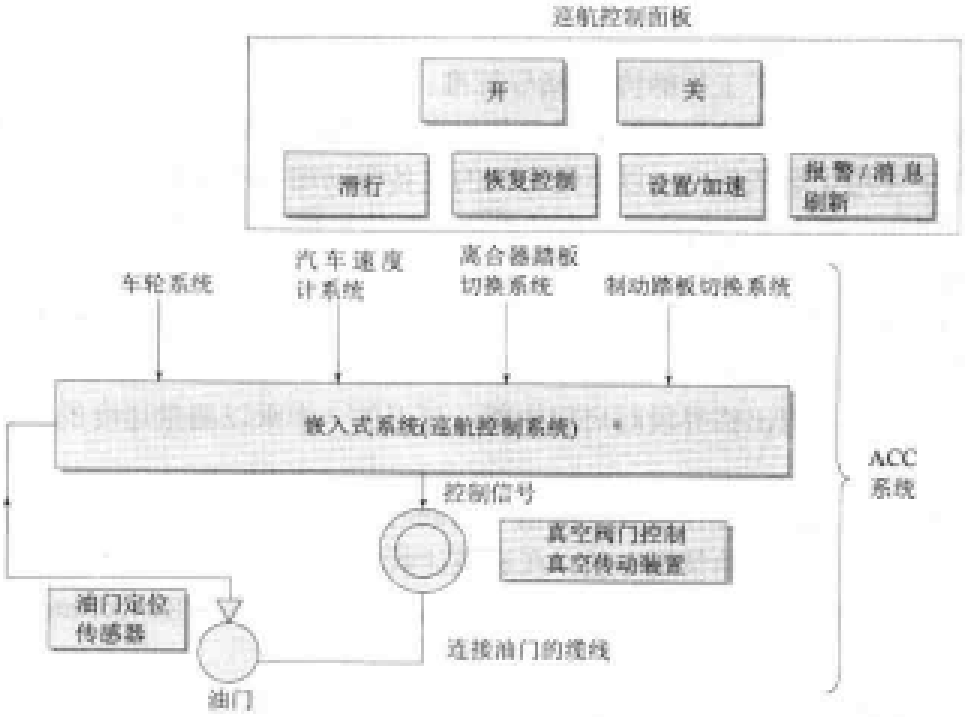


图 11-6　ACC 嵌入式系统的结构图

(1) 从 DAS 单元内的速度计得到道路速度。

(2) 从 DAS 单元内的引擎速度得到加速度。

(3) 得到用于刹车操作的刹车开关输入。

(4) 向踏板系统发送输出信息，用于紧急刹车以及 ACC 行驶控制下的驾驶员不干涉行驶。

(5) 运行自适应算法，计算并发送控制信号给真空管传动装置中的步进电动机。开启真空管的小孔控制油门。这个阀门是电气阀门。建立真空时通过风箱提供所需动力。也可以直接使用油门上带有螺纹驱动附件的 A.D.C 或者步进电动机，代替真空传动装置和风箱。

(6) 通过步进电动机定位传感器得到油门定位。

(7) 控制前置面板，该面板有以下部件：(a)用于“开”、“关”、“自动驾驶”、“恢复控制”、“设置/加速”开关和显示。驾驶员通过分别按下“开”或者“关”来激活和关闭 ACC 系统。通过分别按下“自动驾驶”或者“恢复控制”来移交和恢复 ACC 系统的控制权。驾驶员使用“设置/加速”开关设置行驶速度。开关依据 ACC 激活时的状态闪烁绿光或者红光。(b)报警和消息闪烁单元发出适当的报警信号和消息闪烁统计图表。

(8) 从前方车队的车轮上的雷达或者 UVHF(Ultra Very High Frequency，超高频率)附件中得到前方汽车的距离。步进电动机调整该附件，使雷达发射器的视线和前方汽车保持在一条直线上。雷达系统保证了车队稳定，并在紧急情况发出警告。

选择 ACC 的案例研究，是为了证明 RTOS 在某些应用程序中是必需的。我们使用 RTOS 的一些功能特性，来得到汽车电子系统中可靠的控制系统。这里给出的 ACC 系统示例，包括系统维持车队稳定的算法和端口。汽车电子设备中的硬件系统必须提供功能上的安全。以下是当前一些重要的硬件标准和准则：

a. TTP(Time Triggered Protocol，定时触发协议)

b. CAN(Control Area Network，控制域网络)(参见 3.3.2 节)

c. MOST(Media Oriented System Transport，媒体定向系统传输)

d. 为 EMC(Electromagetic Magnetic Control，电磁控制)和功能安全准则制定的 IEE(Institute of Electrical Enginerrs，电子工程师协会)指导标准。

当前，基于嵌入式系统的汽车控制单元使用微处理器、微控制器、DSP 或者 ASIP。我们假设在该案例中使用了这些单元，以下是它们的设备驱动端口和函数。

(1) Port_Align：步进电动机端口。电动机按中断信号顺时针或者逆时针运转。该电动机还会调整雷达或者 UVHF 传输设备和前方汽车在一条直线上。

(2) Port_Ranging：将时间差 timeDiff 读入端口设备。然后服务例程开始执行，雷达发射信号，传感器接收来自前方汽车的反射信号。该程序刚开始禁用中断。端口设备电路测量两个时刻之间的延迟。程序在退出临界段时启用中断。延迟的一半乘以测量速度的任务提供的速度，得到前方汽车的距离。

(3) Port_Speed：端口设备在接收中断信号的同时，设置一个 N_rotation 递减计数器，并注明该时刻的时间。程序刚开始禁用中断。在递减计数溢出后，它再次注明时间，并得到时间差 deltaT。程序在退出临界段时启用中断。Port_Speed 收到来自 Port_RangeRate 的信号时，在速度计上显示当前速度 speedNow。

(4) Port_RangeRate：它向端口传输信号 rangNow 和 speedNow，发送给车队中的其他汽车。同时发送信号给 Port_Speed，在速度计上显示 speedNow。

(5) Port_Brake：端口设备在收到中断信号时应用刹车和紧急刹车操作。该服务例程刚开始禁用中断，在退出临界段时启用中断。它也应用刹车操作并将刹车信息以信号的方式发送给车队中的其他汽车。

图 11-7 给出了 ACC 系统的动作周期和任务调度模型。注意图中周期开始和结束的标记。

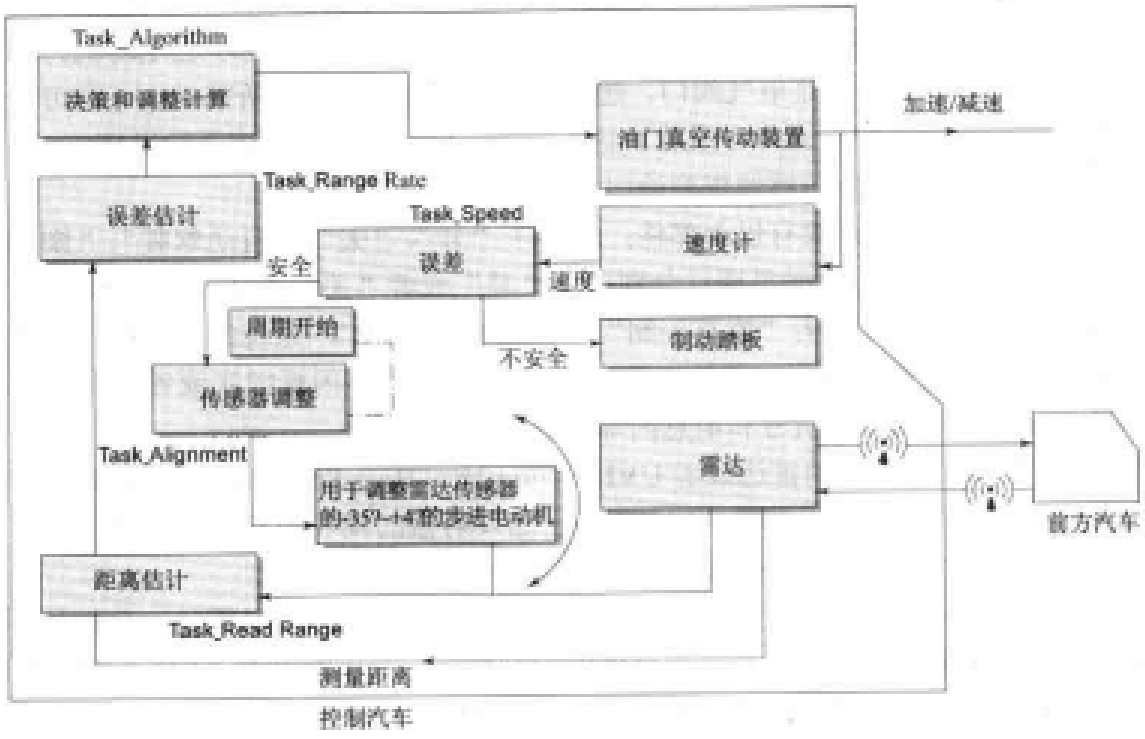


图 11-7 ACC 系统的动作周期和任务调度模型

(1) 周期从任务 Task_Alignment 开始。它向步进电动机端口 Port_Align 发送信号。步进电动机按照信号的指示，顺时针或者逆时针运转。

(2) 任务 Task_ReadRange 测量前方汽车距离。该任务在进入临界段时禁用中断。显然需要实时测量距离。Port_Ranging 同时得到 timeDiff。

(3) Task_Speed 得到端口 Port_Speed 上读到的结果。端口发送输入内容 deltaT，它是第 1 转到第 N 转的时间间隔。

(4) Task_Range_Rate 发送 rangeNow 给 Task_Alignment，让步进电动机将传感器调整到和前方汽车在一条直线上。该任务在维持车队稳定时，从 Task_ReadRange 的输出中计算最后的误差，从 Task_Speed 的输出中计算维持车速的误差。它输出两个误差值，在控制系统自适应算法中使用。通过 Port_RangeRate 将误差传给其他汽车。发送当前速度 speedNow 给 Port_Speed，这个速度和时间差 deltaT 成反比。比例 N_rotation/deltaT 在应用过滤函数后，给出 speedNow。Port_Speed 连接到 DAS 的速度计系统，该系统在应用适当的过滤函数后显示 speedNow。Port_RangeRate 也向车队中的其他汽车传输 speedNow。它计算距离和速率误差，通过 Port_RangeRate 传输 rangeNow 和 speedNow。

(5) Task_Algorithm 运行主自适应算法。它按以下方式输入。从 Task_RangeRate 获得输入信息，输出目标是 Port_Throttle。Port_Throttle 连接真空传动步进电动机。整个周期在此结束。一段延迟后，新的周期再次从 Task_Alignment 开始。它通过 Port_RangeRate 接收其他汽车的距离和速率误差消息，并读取该车和车队中其他汽车的 Port_Brake 的状态。

Task_Algorithm 为 ACC 的动作生成输出内容。输出端口为带有油门的端口生成控制信号。端口 Port_Throttle 上的信号按以下方式计算：可调整的算法 A，得到速度“speed”和前面对象的距离“range”，以及未知和已知扰动 P_Unkown 和 P_Known 的输入。它调节发送给 Port_Throttle 的输出信号。算法 B 计算性能指标 IP。算法 C 将该 IP 同一组给定的 IP 值进行比较。算法 D 依据自适应机制，调整 C 的输出。C 发送新参数，A 将调整这些参数。

表 11-3 列出了具有车队稳定算法的 ACC 中需要的任务、函数和 IPC。我们可以从发送的 IPC 和获取的 IPC 列中注意到，信号量用作定义任务运行顺序，以及在 Task_Algorithm 中循环运行 ACC 和车队稳定控制算法的顺序。

表 11-3 示例 11-3 中使用的 BCC 1[1] 任务函数和 IPC 的列表

BCC 1 任务函数	BCC 1 优先级	动　作	待定 IPC	发送的 IPC	输　入	输　出
Task_ Alignment	101	向 Port_Align 发送信号	SemReset	SemAlign	deltaStep，step	step
Task_ReadRange	103	禁用中断，向 Port_ Ranging 发送信号。端口激活雷达扫描，记录激活时间，得到察觉反射雷达信号的时间，计算时间差 timeDiff。启用中断	SemAlign	SemRange	–	timeDiff
Task_ peed	105	向 Port_Speed 发送信号，启动定时器和计数器，然后等待车轮旋转 10 次的计数值。输出 deltaT	SemRange	SemSpeed	–	deltaT

(续表)

BCC 1任务函数	BCC 1优先级	动　作	待定 IPC	发送的 IPC	输　入	输　出
Task_angeRate	107	通过 timeDiff 计算出 rangeNow。从存储和比较的结果中得到预定的前方汽车距离和车队长度。得到预定行驶速度，cruise speed；将它同当前速度 speedNow 比较	SemSpeed	Sem-Reset, SemACC	avgTire Circum, timeDiff, deltaT, string Range, Cruise Speed, N_rotation	Range Error, speed Error, range Now, speed Now
Task_ Algorithm	109	(i)得到速度和距离的误差，执行自适应控制算法。(ii)通过 Port_ RangeRate 得到其他汽车的误差。(iii)得到其他汽车 Port_Brake 的状态。(iv)得到当前油门定位。(v)向 Port_ Throttle 发送输出数据 throttleAdjust。(vi)在需要紧急刹车的情况下向 Port_Brake 发送信号。(vii)Port_Brake 也向其他汽车传输必要的动作信息	SemACC	SemReset	RangeError, speedError, 所有 Port_ RangeRate 值和 Port_Brake 状态, VehicleID	throttle adjust, emergency

1. 每个优先级的任务只有一个，并且只激活一次的基本任务，称作 BCC 1(Basic Conformance Class 1)。

10.1 节到 10.3 节中描述的 RTOS 对于该系统和其他必要的功能特性来说是不够的。汽车系统中的嵌入式系统在操作系统上需要一些超过 MUCOS 或者 VxWorks 以及 MS DOS 和 UNIX 的特殊性能。所需的特殊 OS 性能如下所示：

(1) 语言可以与应用程序相关，没有必要是 C 或者 C++，数据类型也应该与具体的应用程序相关，但不与具体的 RTOS 相关。例如，在 VxWorks 中 STATUS 是 RTOS 相关的，这是不允许的，因为它可能成为一个错误源，所以不可靠。

(2) 它的 OS、每种方法、类和运行时库应该是可伸缩的。这优化了对存储器的需求。

(3) 任务可以分成 4 类。这向程序员提供了一个明显的任务划分，使得他们可以确定系统中哪些模块使用哪类任务。

a. 每个优先级的任务只有一个，并且只激活一次的基本任务称作 BCC 1(Basic Conformance Class 1)。例如示例 9-17 中主函数创建的 FirstTask。

b. 每个优先级的任务只有一个，并且只激活一次的扩展任务称作 ECC 1(Extended Conformance Class 1)。例如示例 9-17 中 FirstTask 创建的任务。

c. 每个优先级的任务有多个，并且运行期间可激活多次的基本任务称作 BCC 2。

d. 每个优先级的任务有多个，并且运行期间可激活多次的扩展任务称作 ECC 2。

(4) OS 可以以截然不同的方式调度 ISR 和任务(VxWork 调度程序也是这样做的)。

(5) 中断系统在服务例程开始时被禁用，在返回时启用。这使得任务可以在实时环境中运行。

(6) 任务可以实时调度。

(7) 任务可以由 3 组对象组成，事件(信号量)、资源(语句和函数)和设备。前面描述了一些端口设备。报警器就是一个例子。它显示统计图表、消息和闪烁消息，并发出嘟嘟声和蜂鸣声。

(8) 不允许创建和删除定时器、任务或者信号量对象。运行时故障可能导致定时器或者信号量未经允许就被删除。这是潜在的问题根源，因而不可靠。

(9) 消息队列通过任务发送的 IPC 不允许作为等待任务，因为它可能无限等待所需要的全部消息。RTOS 队列类型、无限等待和消息超时可能是潜在的故障源，因而不可靠。信号量作为资源键或者计数器时可能会有类似的风险，所以不使用它们。

(10) 进入临界段之前和执行服务例程的时候，必须禁用所有中断，只在返回的时候才启用(参见 6.8 节)。

汽车电子设备中的软件也必须是标准的。当前一些重要的软件标准和准则是 AMI-C(汽车多媒体接口协作)(http://www.amic.org)，MISRA-C(针对汽车系统中 C 语言软件准则的发动机行业可靠性协会标准)(www.misra.org.uk)和用于 RTOS、通信和网络管理的 OSEK/VDX(参见站点 http://www.osek-vxd.org 和 Joseph Lemieux 编著的 *Programming in the OSEK/VXD* 一书)。

OSEK 定义了 3 项标准：

(1) 操作系统方面的 OSEK-OS，具有较高的可靠性。因为基于 OS 之上的 OSEK，具备了上文提到的 10 点特性。

(2) 网络管理方面的 OSEK-NM 体系结构。OS 中的任务分成 4 类，而 NM 进一步将体系结构分成了两类。(i)网络消息的直接传送和互换(ii)同时在两个节点之间的间接传送和互换。

(3) 针对相同CPU控制单元任务之间和不同CPU控制单元任务之间的IPC 的OSEK-COM 体系结构。在不同单元任务之间，存在数据链路层和物理层。不同的 CPU 物理层通过 CAN 总线结构连接。

从这里我们已经看到，OSEK-OS 标准比 VxWorks 或者 MUCOS 更可靠。为了说明 ACC 的应用，先看一看 VxWorks 的应用以及如何在示例(示例 11-3)代码应用以下内容：

(1) 使用 BCC 1 类型的任务，如示例 11-2 中 VxWorks 的应用。定义每个优先级不同的任务，并且它们在代码中只激活一次。

(2) 不使用消息队列。

(3) 不使用任务的中间创建和删除。

(4) 只有当任务不在运行时创建和删除时，才使用信号量作为事件标志。

示例 11-3

```
1. # include "vxWorks.h" /* Include VxWorks functions. */
# include "semLib.h" /* Include semaphore functions Library. */
# include "taskLib.h" /* Include multitasking functions Library. */
# include "sysLib.c" /* Include system library for system functions. */
#include "sigLib.h / * Initialise kernel component for signal functions. */
2. /* Set system clock rate 10000 ticks per second. Every 100 is per tick*/
sysClkRateSet (10000);
3. /* Declare and Initialise Global parameters. */
unsigned byte VehicleID; /* Declare this car ID. */
```

```
static numCars = …; /* Numbers of cars in the string that should move with cruise
speed. */
static unsigned byte N_rotation = ....; /* Initialise number of rotations needed
when finding the speed.*/
static int avgTireCircum = … ; /* Initialise average tire Circumference in mm. */
4. /* Initialise the string Range = Separation to be maintained between the two
cars in the string in mm. Initialise cruise speed. */
/* All distances are in mm, speeds in km/hr and time in nanosecond unless otherwise
specified. */
static int stringRange = …; /* In unit of mm */
static int CruiseSpeed = …; /* In unit of km/hr */
static float unitChange = 3600000.0; /* Km in one mm divided by hours in one
nanoseconds. */
float permittedSpeedError = … ; / * In units of mm/ns error in speed permitted.
It prevents oscillations in control system. */
static boolean alignment = false; /* Declare alignment = false to initiate radar
transmitter alignment. */
5. /* Other Variables. */
static byte *step; /* Stepper motor step angle in degrees. */
static byte * deltaStep = 0; /* Stepper motor step angle change in degrees. */
/* Time difference between emitted signal and reflected signal from front-end
car. rangeNow is present range in mm. It is timeDiff multiplied by speedNow after
a filter function application. */
static unsigned long *timeDiff; static int *rangeNow;
static unsigned long *deltaT; /* Time interval for N rotation in ns. */
6. /* Declare pointers for variables Range error, range now, speed error, speed
now. */
int rangeError, speedError, rangeNow = 0, speedNow =0; /* Speeds are in km/hr
and range in mm. */
7. /* Declare arrays of size number of cars, numCars. These many cars are running
as a string. Declare brakeStatus, RangeErrors, SpeedErrors, Ranges, Speeds for
all the cars. */
boolean brakeStatus [numCars];
int RangeErrors [numCars], Ranges [numCars], SpeedErrors [numCars], Speeds
[numCars]; boolean emergency [numCars]; /* Declare variable for emergency
message sent to Port_Brake of N^th car. */
int *throttleAdjsut; /* Declare variable for throttle adjusting parameter */
8. /* Other Variables. */
.
.
9. /* Declare all Table 11.3 Task function prototypes. */
void Task_Alignment (SemID SemReset, SemID SemAlign, byte *step, byte
*deltaStep); /
*Task for aligning stepper motor in front-end car view. */
void Task_ReadRange (SEM_ID SemAlign, SEM_ID SemRange, unsigned long *timeDiff); /
*Task for receiving the timeDiff using the radar for calculating rangeNow. */
void Task_Speed (SEM_ID SemRange, SEM_ID SemSpeed, unsigned long *deltaT);
/*Task for receiving the deltaT using the wheel counter and timer for calculating
speedNow. */
void Task_Range_Rate (SEM_ID SemSpeed, SEM_ID SemReset, SEM_ID SemACC, int
avgTireCircum, unsigned byte N_rotation, int CruiseSpeed, int stringRange,
unsigned long *
```

```
time-Diff, unsigned long *deltaT, int * range-Error, int *speedError, int
*range-Now, int
*speed-Now); /*Task for calculating rangeNow, speedNow, rangeError, speedError.
*/
void Task_Algorithm (SEM_ID SemACC, SEM_ID SemReset, boolean brakeStatus
[numCars],
int RangeErrors [numCars], int Ranges [numCars], int SpeedErrors [numCars], int
Speeds
[numCars],
boolean emergency [numCars], unsigned byte VehicleID); /* Declare array for
emergency message sent to Port_Brake of N^th car. */
int *throttleAdjsut; /* Declare variable for throttle adjusting parameter */
10. /* Declare all Table 11.3 Task IDs, Priorities, Options and Stacksize. Let
initial ID, till spawned be none. No options and Stacksize = 4096 for each of
six tasks. */
int Task_AlignID = ERROR; int Task_AlignPriority = 101; int Task_AlignOptions
= 0; int Task_AlignStackSize = 4096;
int Task_ReadRangeID = ERROR; int Task_ReadRangePriority = 103; int Task_Read
RangeOptions = 0; int Task_ReadRangeStackSize = 4096;
int Task_SpeedID = ERROR; int Task_SpeedPriority = 105; int Task_SpeedOptions
= 0; int Task_SpeedStackSize = 4096;
int Task_RangeRateID = ERROR; int Task_RangeRatePriority = 107; int Task_Range
RateOptions = 0; int Task_RangeRateStackSize = 4096;
int Task_AlgorithmID = ERROR; int Task_AlgorithmPriority = 109; int Task_Algorithm
Options = 0; int Task_AlgorithmStackSize = 4096;
10. /* Create and Initiate (Spawn) all the six tasks of Table 11.2. */
Task_AlignID = taskSpawn ("tTask_Align", Task_AlignPriority, Task_AlignOptions,
Task_AlignStackSize, void (*Task_Alignment) (SemID SemReset, SemID SemAlign,
byte *step, byte *deltaStep), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
Task_ReadRangeID = taskSpawn ( tTask_ReadRange, Task_ReadRangePriority,
Task_ReadRangeOptions, Task_ReadRangeStackSize, void (*Task_ReadRange)
(SEM_ID SemAlign, SEM_ID SemRange, unsigned long *timeDiff), 0, 0, 0, 0, 0, 0,
0, 0, 0, 0);
Task_SpeedID = taskSpawn ("tTask_Speed",Task_SpeedPriority,Task_SpeedOptions,
Task_SpeedStackSize, void (*Task_Speed) (SEM_ID SemRange, SEM_ID SemSpeed,
un-signed long *deltaT), 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
Task_RangeRateID = taskSpawn ("tTask_Range_Rate", Task_RangeRatePriority,
Task_RangeRateOptions, Task_RangeRateStackSize, void (*Task_Range_Rate)
(SEM_ID SemSpeed, SEM_ID SemReset, SEM_ID SemACC, int avgTireCircum, unsigned
byte N_rotation, int CruiseSpeed, int stringRange, unsigned long * time-Diff,
unsigned long *deltaT, int * range-Error, int *speedError, int *rangeNow, int
*speedNow), 0, 0, 0, 0, 0, 0, 0, 0,0, 0);
Task_AlgorithmID = taskSpawn ("tTask_Algorithm", Task_AlgorithmPriority,
Task_AlgorithmOptions, Task_AlgorithmStackSize, void (*Task_Algorithm)
(SEM_ID SemACC, SEM_ID SemReset, boolean brakeStatus [numCars], int RangeErrors
[numCars], int Ranges [numCars], int SpeedErrors [numCars], int Speeds [numCars],
boolean emergency [numCars], unsigned byte VehicleID), 0, 0, 0, 0, 0, 0, 0, 0,
0, 0);
12. /* Declare IDs and create the binary semaphore event flags. */
SEM_ID SemAlign, SemRange, SemSpeed, SemACC, SemReset; /* Declared for Table
11.3 five tasks. */
13. /* Create the binary semaphores taken in FIFO and as empty to start with. */
```

```
SemAlign = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
SemRange = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
SemSpeed = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
SemACC = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
SemReset = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
14. /* Declare function for starting a RTC timer at the Port_Ranging. */
void RTCtimer_Port_Ranging_Start ( ) {
.
.
};
15. /* Declare a function for starting a RTC timer at the Port_Speed. */
void RTCtimer_Port_Speed_Start ( ) {
.
.
};
16. /* Declare a function to read 64-bit time from an RTC. */
unsigned long timer_gettime (&RTC) {
.
.
};
17. /* Define a macro for calculating time between instance when control bit
= true and Status flag = true. */
boolean * CB; /* Control bit */
boolean * SF; / * Status flag. */
*CB =0; /* Control bit = 0; */
*SF = 0; /* Status flag = 0; */
unsigned short * RTC; /* Pointer to a real-time clock. */
#define unsigned long calculate_TimeInterval (unsigned short * RTC, boolean *
CB, boolean *SF) (unsigned long timeInstance0, timerInstance1;
while (*CB != 1 && *SF = = 0) { }; /* Wait for read instruction to timer. */
timeInstance0 = timer_gettime (&RTC); /* Find initial time at start in the timer
*/
while (*SF ! = 1) { }; /* Wait for sensor status flag to be true */
timeInstance1 = timer_gettime (&RTC); /* Find initial time at start in the timer
*/
*SF =0; *CB =0;
return (timeInstance1 - timrInstance0);
} /* End of Macro to calculating time interval between two instances specified
by CB and SF becoming true. */
18. /* Define a macro for calculating time interval deltaT between instance when
control run bit = true and instance of N-th count input. */
boolean CR; /* Control Run bit. */
boolean countInput; / *Status flag. */
*CR =0; /* Control run bit = 0; */
*countInput = 0; /* countInput initial value. */
#define unsigned long DelT (unsigned short &RTC, boolean *CR, N_rotation, boolean *
countInput)(
unsigned long timeInstance = 0; byte N = 0;
for (byte N =0; N < N_rotation; N++) {
while (*CR != 1 && *countInput != 0) { }; /* Wait for count input true. */
timeInstance += timer_gettime (&RTC); /* Find initial time at start in the timer
*/
```

```
*countInput =0; /*Reset countInput. It will set on start of next rotation. */
}; *countInput =0;
return (timeInstance);
) /* End of Macro to calculate time interval for N count inputs. */
19. /* Declare Macro for sending a byte for step angle setting to Port_Align.
*/
unsigned short * Port_Align; /* Declare a pointer for Port_Align. */
# define Out_Alignment (&Port_Align, Step) / * Codes in Assembly Language for
stepper motor routine. * /
.
.
); /* End of Macro to sending byte for step angle to Port_Align. */
20. /* Declare Macro to find timeDiff from Port_Ranging. */
unsigned short RTC_Port_Ranging =…; /* Declare Address of RTC at Port_Ranging.
* /
RTCtimer_Port_Ranging_Start ( );
#define RANGE (unsigned short * Port_Ranging, unsigned long * timeDiff) (
unsigned short * RTC_Port_Ranging = …..; /* Declare address of RTC of Port_Ranging.
*/
intLock ( ); /*disable interrupts. */
/ * Codes in Assembly Language for Ranging routine for Start Radar transmission
by making control bit CB = 1* /
*CB = 1;
*timeDiff = calculate_TimeInterval (&RTC_Port_Ranging, &CB, &SF);
intUnlock ( ); /* Enable Interrupts. */
) /* End of Macro to find time interval for reflected radar signals. */
21. /* Declare Macro to find deltaT from Port_Speed. */
unsigned short RTC_Port_Speed =…; /* Declare Address of RTC at Port_Speed.* /
RTCtimer_Port_Speed_Start ( );
#define SPEED (unsigned short * Port_Speed, unsigned long * deltaT) (
unsigned short * RTC Port Ranging = …..; /* Declare address of RTC of
Port_Ranging.*/
intLock ( ); /*disable interrupts. */,
/ * Codes in Assembly Language for Speed routine for Start counting the tire
rotation count inputs by making control bit CR = 1* /
*CR = 1;
*deltaT = DelT (&RTC_Port_Speed, &CR, N_rotation, &countInput);
intUnlock ( ); /* Enable Interrupts. */
) /* End of Macro to find time interval for N count-inputs. */
23.
# define float filter_speed (float calculatedSpeed, int *speedNow, float
permittedSpeedError) (/* Codes for filtering the calculated speed. If the
calculated value in mm/ns is within plus minus limit of permittedSpeedError,
in mm/ns do not change it; else modify it with new value. */
float a;
a = (float) (speedNow/ unitChange);
if (calculatedSpeed > a + permittedSpeedError | | calculatedSpeed < a -
permittedSpeedError) {return(calculatedSpeed);} else return (a);
) /* End of macro for filtering the speed calculated to prevent vibrations and
oscillation in controlling vehicle. * /
24.
#define RangeRate (unsigned short * Port_RangeRate, int *speedNow, int *rangeNow,
```

```
int *speedError,
int *rangeError, unsigned byte VehicleID) {
/ *Assembly codes for sending to Port_RangeRate and transmitting the range and
rate parameters and vehicleID. Port_RangeRate sends also *speedNow for display
on speedometer at Port_Speed. /
.
.
); /* End of Macro to transmit range and rate parameters through Port_RangeRate.
*/
25 to 27. /* Code for Other Declaration Steps specific to the various functions
*/
.
.
/* End of the codes for creation of the tasks, semaphores, message queue, pipe
tasks, and variables and all needed function declarations */
/
*****************************************************************/
28. /* Start of Codes for Task_Alignment. */
void Task_Alignment (SemID SemReset, SemID SemAlign, byte *step, byte *deltaStep)
{
.
29.
while (1) { /* Start of while loop. */
/* When cycle starts the semaphore is given. Wait for it. */
semTake (SemReset, WAIT_FOREVER); /* Wait for Cycle Reset Event flag. */
30. /* Codes for sending the step to the address of Port_Ranging. *
*step = *step + *deltaStep;
Out_Alignment (&Port_Align, *Step);
semGive (SemAlign);
}; /* End of while loop. */
} /* End of Task_Alignment. */
/
*****************************************************************/
31. /* Start of codes for Task_ReadRange. */
void Task_ReadRange (SEM_ID SemAlign, SEM_ID SemRange, unsigned long *timeDiff)
{
32. /* Initial assignments of the variables and pre-infinite loop statements
that execute once only*/
static unsigned short Port_Ranging = ....; /* Declare pointer to Port_Ranging.
*/
.
.
33. while (1) { /* Start an infinite while-loop. We can also use FOREVER in place
of while (1). */
34. / *Wait for SemAlign state change to SEM_FULL by semGive function. */
semTake (SemAlign, WAIT_FOREVER);
/*Send signal to Port_Ranging. Port activates a radar flashing, records
activation time, gets time of sensing the reflected radar signal and finds time
difference, timeDiff. Enable interrupts. */
```

```
RANGE (& Port_Ranging, &timeDiff);
);
semGive (SemRange);
};
35. } /* End of the codes for Task_ReadRange. */
/
**************************************************************************/
36. /* Start of codes for Task_Speed. */
void Task_Speed (SEM_ID SemRange, SEM_ID SemSpeed, unsigned long *deltaT) {
37. /* Initial assignments of the variables and pre-infinite loop statements
that execute once only*/
static unsigned short Port_Speed = …; /* Declare pointer to Port_Speed. */
.
.
38. while (1) { /* Start task infinite loop. */
semTake (SemRange, WAIT_FOREVER);
/* Codes for receiving the deltaT using the wheel counter and timer for later
on calculating speedNow. */
SPEED (& Port_Speed, &deltaT);
semGive (SemSpeed);
}; / * End of while loop. */
39. } /* End of codes for Task_Speed */
/
**************************************************************************/
/* Start of codes for Task_Range_Rate. */
void Task_Range_Rate (SEM_ID SemSpeed, SEM_ID SemReset, SEM_ID SemACC, int
avgTireCircum, unsigned byte N_rotation, int CruiseSpeed, int stringRange,
unsigned long * time-Diff, unsigned long *deltaT, int * range-Error, int
*speedError, int *range-Now, int *speed-Now) {
static unsigned short Port_RangeRate = …; /* Declare pointer to Port_Ranging.
*/
.
.
40. while (1) { /* Start task infinite loop . */
semTake (SemRange, WAIT_FOREVER);
41. /* .Codes for calculating rangeNow, speedNow, rangeError, speedError. */
*speedNow = (int) (unitChange * filter_speed ((float) (avgTireCircum *
N_rotation) / (float)(*deltaT), int *speedNow, float permittedSpeedError)));
*rangeNow = (*speedNow/unitChange) * (*timeDiff)/2.0; /* Divide by 2 because
reflected signal travels twice the distance in mm/ns. */
*speedError = cruiseSpeed - *SpeedNow;
RangeRate (& Port_RangeRate, *speedNow, *rangeNow, *speedError, *rangeError,
VehicleID); / *Send the parameters for transmission to other vehicles and
Port_Speed for displays. */
if (alignment != true) {semGive (SemReset);
42. /* Code for loop of tasks of priorities 101, 103, 105 in which the values
of rangeNow are calculated at different step values by changing deltaStep and
finally at that instance alignment is declared as true for rangeNow is minimum.
```

```
Front-end vehicle is now in line of sight. */
.
.
} else semGive (SemACC); /* After alignment is perfect the control algorithm
is sent the event flag. */
}; / * End of while loop. */
43. } /* End of codes for Task_Range_Rate. */
/
*****************************************************************************/
/* Start of Codes for Task_Algorithm. */
void Task_Algorithm (SEM_ID SemACC, SEM_ID SemReset, boolean brakeStatus
[numCars],int RangeErrors [numCars], int Ranges [numCars], int SpeedErrors
[numCars], int Speeds [numCars],boolean emergency [numCars], unsigned byte
VehicleID) {
.
.
44. while (1) { /* Start task infinite loop . */
semTake (SemACC, WAIT_FOREVER);
45. /* Assembly codes for getting errors of other vehicles through Port_RangeRate
and other vehicles Port_Brake statuses through Port_Brake. */
.
.
46. /* Assembly codes to read throttle position from Port_Throttle. */
.
.
47. /* Codes for cruise speed adaptive algorithm and code for string stability
maintaining adaptive algorithm to generate appropriate throttleAdjsut signal
to Port_Throttle. */
.
.
48. /* Codes for Port_Brake action, if emergency = true. Port_Brake transmits
the action needed to other vehicles also. */
if (emergency [numCars] = =1) {
.
} else semGive (SemACC); /* After alignment is perfect the control algorithm
is sent the event flag. */
49. }; / * End of while loop. */
50. } /* End of codes for Task_Algorithm. */
/
*****************************************************************************/
```

11.4 智能卡中的嵌入式系统

智能卡是现在使用嵌入式系统最多的领域之一。它用作银行信用卡、ATM 卡、电子钱包卡、身份识别卡、医疗卡(记录病史和诊断信息)和许多新兴领域中用到的卡片(可以访问 http://www.sguthery@tiac.net 站点，从中得到一些与卡有关的信息)。

图 1-11 给出了一个示例卡的硬件，并且 1.6.6 节列出常见卡片中的单元部件。当智能卡用于金融和银行相关事务时，其安全性是最重要的(可以访问 http://www.home.hkstar.com/~alanchan/papers/smartCardSecurity/he 和 http://www.research. ibm.com/secure_systems/scard.htm 获得有关卡的安全需求规范的详细信息)。

11.4.1 嵌入式硬件

智能卡是一种塑性卡，尺寸遵循 85.60mm×53.98×0.80mm 的 ISO 标准。它也是一种嵌入式的 SoC(System-On-Chip，片上系统)(ISO 标准是指卡式主机接触件的 ISO7816(1–4)标准和弱接触卡的 ISO14443(A 或 B)标准)。

半导体芯片的尺寸一般只有几毫米，并且封装在介质层之间。芯片的超小尺寸可以保护卡片不被弯曲。在考虑软件和操作系统之前，我们先了解一下各个硬件单元。下面是智能卡硬件的几个基本特征：

(1) 使用的微控制器可以是 MC68HC11D0 或者 PIC16C84，或者是智能卡处理器 Philips Smart XA 和 ASIP Processor。MC68HC11D0 有 8KB 的内置 RAM 和 32KB 的 EPROM，以及 2/3 的金属丝保护存储器。大部分智能卡使用 8 位 CPU。关于智能卡的最新介绍中给出了一个 32 位的 RISC CPU。智能卡应该有一些特殊的功能，例如安全锁，这种锁用于保护存储器的一个特定区域。微控制器的一个保护位可以保护 1KB 或者更多的数据，防止它们被任何外部程序访问和修改，或者受保护存储区以外的指令访问和修改。一旦在微控制器内的可屏蔽 ROM 中设置了保护位，那么这一部分存储区的指令或数据就只能被该区域内部的指令访问，而不允许外部指令或者这一区域以外的指令访问它们。CPU 可以通过阻塞总线上数据位的写周期，来禁止对物理存储器上受保护区域的指令和数据进行访问，这个过程处在智能卡初始化的某个阶段之后和卡发行给用户之前。另外一种保护方法如下：CPU 可以使用物理地址访问存储器，这不同于程序中使用的逻辑地址。可能有类似 I2C 或者扩展 I2C 的总线体系结构，采用这种结构的智能卡可以自由访问存储器(参见 3.6 节的图 3-16)。ROM 和 RAM 的访问总线是分离的，受保护存储器的访问采用了 2-Bus 总线协议。为了更有效地保护存储器和 IO 系统，也可能有 3-Bus 总线协议。

(2) 智能卡中使用了标准 ROM。ROM 通常是 8KB 和 64KB，它们分别适用于卡的普通和高级加密功能。只有在安全检测通过之后，ROM 总线的全部或者部分才会起作用。处理器保护了一部分存储器的访问。ROM 中存储了如下数据：

i. 生产密钥，每个卡的惟一密钥，在生成过程中植入。

ii. 私有密钥，当芯片在印刷电路板上测试之后植入。在测试阶段使用物理地址。该密钥保留了生产密钥，并且密钥植入时保留了卡的特征。该密钥植入后，RTOS 和应用程序就只使用逻辑地址。

iii. RTOS 代码

iv. 应用程序代码

v. 防止对两个 PIN 进行修改和对操作系统及应用程序指令进行访问的应用锁(utilisation lock)。该锁在智能卡投入使用后被存储进卡中。

(3) EEPROM 或者闪存是可扩展的。这意味着只有存储器的必需部分才会解锁投入使用，使用时需要特殊的操作。授权者使用它的必需部分；应用程序使用它的其他部分。利用存储在其中的访问条件，存储器得到保护。它存储了如下数据：

i. PIN(Personal Identification Number，个人身份验证号)，它通过授权者(例如一家银行)分配和写入，可能仅仅通过私有密钥和生产密钥进行写操作。在以后的事务处理中，它将用来验证卡的用户身份。用户之前可以得到这个密钥。另外，用户会得到一个可以修改的密码，并且通过这个密码可以打开 PIN 密钥。

ii. 授权者(如银行)使用的开启 PIN。卡上的系统会在开启之前通过这个密钥验证授权者身份。用户数据开启，提供给授权者使用，并且授权者通过主机对卡上的信息进行存储。

iii. 不同层次数据文件的访问条件。

iv. 卡的用户数据，例如用户名、银行和分行身份验证号，以及账号或者健康保险的详细资料。

v. 应用程序生成的发行数据。例如使用电子钱包时，其中保存着先前事务和当前结算的详细资料。又如使用医疗卡时，卡上的医疗病史和诊断细节，以及先前的保险索赔和未裁决的保险索赔记录。

vi. 应用程序的非易失性数据。

vii. 卡的过期使用或者误操作，以及用户帐号关闭请求后主机发送的无效锁。它锁定控制者的数据文件或者个别文件，或者同时锁定两者。

(4) 在通过运行操作系统和应用程序对卡进行操作时，RAM 存储操作过程中的临时变量和堆栈。

(5) 芯片电源通过电荷泵电路的运行来供应。这个电荷泵从主机的电信号中抽取电荷，然后向卡的芯片、存储器和 I/O 系统提供电压，类似于鼠标在计算机中的作用。电信号可以从天线或者时钟发生器的管脚获得。在一个典型的 0.18-μm 工艺的的智能卡中，电压阈值是 1.6V~5.5V，对于一个 0.35-μm 工艺的的智能卡，电源阈值是 2.7V~5.5V。

(6) 芯片的 I/O 系统通过异步串行的 UART(图 3-2a)与主机进行交互，速度是 9.6k、106k 或 115.2k 波特/秒。芯片要么通过金质接触点，要么通过两端的天线线圈和卡的主机系统互连(读和写)。后者提供了弱接触的互连。基于接触交互的 I/O 管脚，通常用于 RST 信号(主机的 Reset Signal)和时钟信号(来自主机)。弱接触交互是通过天线中的信号辐射实现的。卡和主机通过卡上的调制解调器以及上机上的调制解调器实现互连。调制器以 10%的振幅指标调制 13.66MHz 的载波。1/16 频率的副载波通过 BPSK(Binary Phase Shifted Keying，二进制相移键控)调制负载。

11.4.2 嵌入式软件

智能卡是当今安全 SoC 系统领域使用得最广泛的系统之一。智能卡是一个安全 SoC 的例子。这种卡需要加密软件。第 10 章描述的 RTOS 对于智能卡的需求和其他基本特性来说是不

够的。卡中的嵌入式系统需要在操作系统中有一些超过 MS DOS 或者 UNIX 系统特性的特殊性能。需要的特殊性能如下：

(1) 受保护环境。意思是它应该存储在 ROM 的受保护区域。

(2) 严格的运行时环境。

(3) 它的 OS、每种方法、类和运行库应该是可扩展的。

(4) 生成的代码大小应该是优化的(参见 5.2 节)。系统的存储器需求不应该超过 64KB。

(5) 数据类型的限制使用：多维数组，64 位长整数和浮点，以及非常有限的错误句柄、异常、信号、串行化、调试和描述的使用。

(6) 用于数据的 3 层文件系统。作为主文件的一个文件存储所有文件头。文件头包括文件状态、访问条件和文件锁。第 2 个文件是专用文件，保存一个文件的分组信息和组的中继元文件的文件头。第 3 个文件是保存文件头及其文件数据的元文件。

(7) 有一个固定长度的文件管理，或者借助每个文件所携带的预定义偏移量的可变长度的文件管理。

(8) 它应该针对网络、套接字、连接、数据报、字符输入输出流、安全管理、数字认证、基于密钥的对称和非对称加密，以及使用 DES/3、DES、RSA 和 SHA 1 的数字签名进行分类。(对于术语的解释，请参考本书作者所编著的 *Internet and Web Technologies* 一书，由 Tata McGraw-Hill 出版社于 2002 年出版)。

Java Card™，嵌入式 Java 或者 J2ME(Java 2 Micro Edition，Java 2 微型版)提供了一个智能卡的方案(参见 5.9 节和访问相关网站)。JVM 有内置的线程调度。因而当使用 Java 时，不需要独立的多任务 OS，这是因为所有的 Java 字节码运行在 JVM 环境中。Java 对这两个方面提供的支持特性：(i)使用 java.lang.SecurityManager 类的安全管理；(ii)加密要求(java.security*包)(参考 Joseph L. Weber 编著的 *Using Java™ 2 Platform* 一书，Prentice Hall of India 出版社，2002 年 5 月重印)。Java 提供了对连接、数据报、I/O 流和网络套接字的支持。Java mix 是一项新技术，这项技术中，卡的本地应用程序运行在 C 或者 C++的环境中，可下载的应用程序运行在 Java 或者 Java Card™ 环境中。系统中同时有 OS 和 JVM。

考虑一个弱接触智能卡的例子。该卡不具有磁性(早期的卡使用磁条作为非易失性存储器。而现在使用 EEPROM 保存非易失性应用数据)。在这个例子中，假设 SmartOS 是一个理想的 OS，作为智能卡的 RTOS。记住，使用类似的 OS 函数名称是为了便于理解，它等同与 MUCOS，但实际的 SmartOS 肯定和 MUCOS 不同。其文件结构不同，有两个函数：函数 unsigned char [ ] SmartOSEncrypt (unsigned char *applStr, EnType type)依据一定的加密方法加密，然后返回加密字符串，加密方法是所选的 EnType ="RSA"or"DES"算法。函数 unsigned char [ ] SmartOSDecrypt (unsigned char *Str, DeType type)依据一定的解密方法解密，然后返回解密字符串，解密方法是所选的 DeType = "RSA" or "DES" 算法。检查访问条件后执行 SmartOSEncrypt 和 SmartOSDecrypt，这里的访问条件来自存储密钥、PIN 和密码的数据文件。

步骤 1：使用卡连接的情况下，在插入卡时，从主机接收载波频率的辐射或者时钟信号。接通电源，给调制解调器、处理器、存储器和 Port_IO(卡的 UART 端口)设备供电。

步骤 2：在 ResetTask 复位时执行引导任务的代码。按示例 11-1 代码中对 FirstTask 任务类似的编码方法进行编码。代码从 main 函数开始执行，然后 main 函数创建并初始化该任务，启动 SmartOS。这里是 ResetTask，它最先执行。

表 11-4 给出了这个案例研究中用于卡 OS 的任务。任务同步模型如下，它们同时也在图

11-8 中给出。

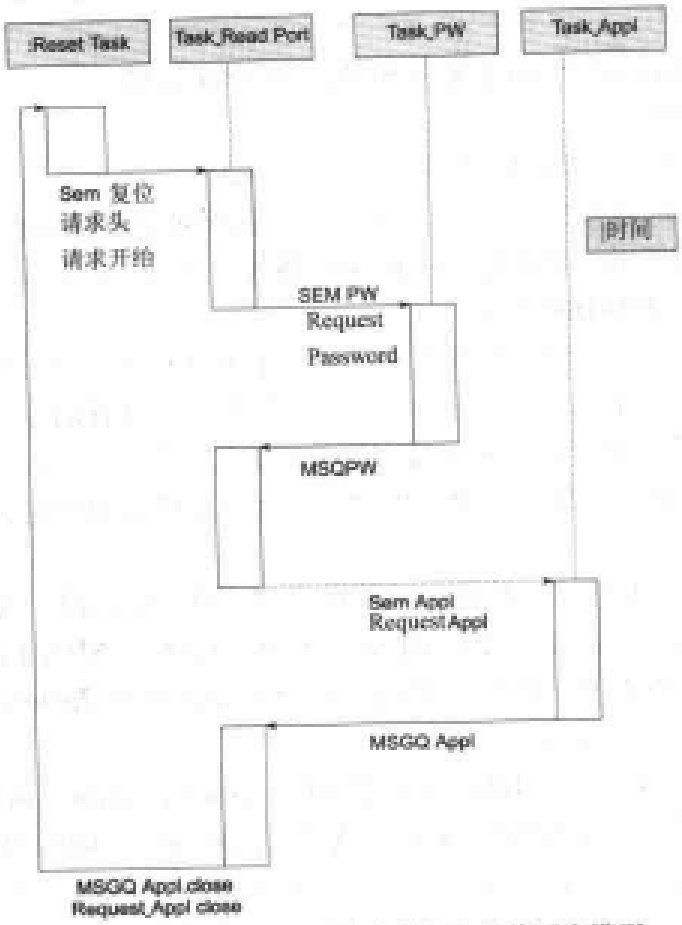


图 11-8 智能卡软件中的任务及其同步模型

表 11-4 示例 11-4 中使用的任务、函数和 IPC

任务函数	优先级	动作	待定 IPC	发送的 IPC	字符串、系统或主机输入	字符串、系统或主机输出
Reset Task	1	初始化系统定时器节拍，创建任务，发送初始消息并挂起自己	没有	SemReset, MsgQStart	SmartOS 调用主函数	请求头文件，request Start
Task___Read Port	2	等待 ResetTask 挂起，发送队列消息并接收消息，启动应用程序并得到关闭应用程序的关闭许可	SemReset,来自 MsgQstart, MsgQPW MsgQappl, MsgQAppl Close 的消息	SemPW, SemAppl	函数 SmartOSEncrypt, SmartOSDecrypt, ApplStr,Str, 关闭许可	请求密码, request Appl, request Appl Close
Task_PW	3	如果 SemPW = 1,在主机验证时发送密码请求	SemPW	MsgQPW	请求密码	–
Task_ppl	8	如果 SemPW = 1, 运行应用程序	SemAppl	MsgQAppl	–	–

(1) ResetTask 任务，执行类似示例 11-1 中的 FirstTask。它在执行完下列操作后永久挂起：(a)为系统节拍初始化定时器，节拍时间设置为 1 ms。(b)创建 3 个任务，Task_ReadPort、Task_PW 和 Task_Appl，下文中有进一步描述。(c)对于等待任务 Task_ReadPort，它发送请求头字符串 requestHeader 到消息队列 MsgQstart 中。前者指明银行分配给用户的 PIN。(d)为了标识主机，发送另一个字符串 requestStart，在 I/O 端口 Port_IO 请求主机 PIN。(e)发送信号量标志 SemReset，并挂起自己，系统控制权不会返回给它，直到出现另一次复位。

(2) 参见示例 4-5。宏函数 ReceiveStr (&Str)使用函数 void portIO_ISR_Input (*portIOdata) 返回输入字符串 Str。“portIO_ISR_Input”在后续的调用中从端口逐个接收字符。同样，函数 void Port_OutStr (unsigned char [ ] *applStr)使用宏函数 SendStr (&ApplStr)来发送输出字符串。portIO_ISR_Output 向端口发送字符串。

(3) Task_ReadPort 仅在任务 ResetTask 将信号量 SemReset 复位或者发送时，才开始执行。(a)Task_ReadPort 获取来自队列 MsgQStart 的消息，并得到待定队列消息 requestHeader 和 requestStart。任务对这两个字符串加密，并发送到 Port_IO，该端口通过调制解调器将加密字符串传输给主机。它通过调制解调器接收主机消息 hostStr。hostStr 指明主机的 PIN，该 PIN 用作卡的银行授权 PIN。(b)如果当前正在运行的任务验证了 hostPIN，则发送信号量标志 SemPW 给等待任务 Task_PW。然后等待来自队列 MsgQPW 的消息，并在解密了端口输入数据字符串后接收 userPW。(c)如果存储在 EEPROM 的文件中的用户密码验证正确，则发送信号量 SemAppl。它在末尾向队列 MsgQApplClose 发送关闭请求消息，向端口 Port_IO 发送消息 requestApplClose，并接收加密字符串 Closure Permitted。任务在解密时删除。

(4) Task_PW 加密后，在获取待定 SemPW 的同时，将发送字符串 requestPW。当得到 SemPW 时，它将 requestPW 发送到 MsgQPW 中。为了在主机上验证用户身份，Task_ReadPort 通过 IO 端口 Port_IO 将它发给主机。

(5) Task_Appl 在得到信号量 SemAppl 时开始运行。操作可能有：(i)修改用户密码；(ii)打印用户银行账号的短小语句；(iii)从主机投出需要的现金；(iv)请求接受带有现金的信封；(v)请求打印本次事务记录；(vi)请求传送给其他人。该任务通过 Task_ReadPort 交互，交互过程中通过队列 MsgQAppl 发送消息。

示例 11-4 给出这种智能卡的编码过程。

示例 11-4

```
1. /* Preprocessor definitions for maximum number of inter-process events to
let the SmartOS allocate memory for the Event Control Blocks */
#define SmartOS_MAX_EVENTS 24/* Let maximum IPC events be 24 */
#define SmartOS_SEM_EN 1/* Enable inclusion of semaphore functions in application.
*/
#define SmartOS_Q_EN 1/* Enable inclusion of queue functions for sending the
string pointers to Task_ReadPort * /
#define SmartOS_Task_Del_En = 0 /* Disable task deletion by SmartOS at the start.*/
/* End of preprocessor commands for enabling IPC functions of the SmartOS*/
2. /* Specify all user prototype of the reset task function that is called by
the main function and is to be scheduled by SmartOS first at the start. In step
11, we will be creating all other tasks within the reset task. Remember: Static
means permanent memory allocation. */
static void ResetTask (void *taskPointer);
```

```
static SmartOS_STK_ResetTaskStack [ResetTask_StackSize];
3. /* Define public variable of the task service and timing functions */
#define SmartOS_TASK_IDLE_STK_SIZE 100 /* Let memory allocation for an idle state
task stack size be 100*/
#define SmartOS_TICKS_PER_SEC 1000 / * Let the number of ticks be 1000 per second.
An RTCSWT will interrupt and thus tick every 1 ms to update counts. */
#define ResetTask_Priority 1 /* Define reset task in main priority */
#define ResetTask_StackSize 100 /* Define reset task in main stack size */
STAF _In = 0; /*Define flag for signaling modem interrupt for receiving a
character. */
STAF_Out = 0; /*Define flag for signal from a modem interrupt after sending a
character. */
/*-------------------------------------------------------------------------*/
*/3. /* Prototype definitions for three tasks for the car application codes after
reset. */
static void Task_ReadPort (void *taskPointer);
static void Task_PW (void *taskPointer);
static void Task_Appl (void *taskPointer);
4. /* Definitions for three task stacks. */
static SmartOS_STK Task_ReadPortStack [Task_ReadPortStackSize];
static SmartOS_STK Task_PWStack [Task_PWStackSize];
static SmartOS_STK Task_ApplStack [Task_ApplStackSize];
5. /* Definitions for three task stack size. */
#define Task_ReadPortStackSize 100 /* Define task 2 stack size*/
#define Task_PWStackSize 100 /* Define task 3 stack size*/
#define Task_ApplStackSize 100 /* Define task 4 stack size*/
6. /* Definitions for three tasks priorities. */
#define Task_ReadPortPriority 2 /* Define task 2 priority */
#define Task_PWPriority 3 /* Define task 3 priority */
6. /* Prototype definitions for the semaphores. */
SmartOS_EVENT *SemReset; /* First task that resets the card posts it. */
SmartOS_EVENT *SemPW; /* Task_PW posts it to send request for getting user
password through the host. */
SmartOS_EVENT *SemAppl; /* Needed when using Semaphore as flag for inter-process
com-munication between Task_ReadPort and Task_PW. */
7. /* Prototype definitions for the queues. */
SmartOS_EVENT *MsgQStart; /* Needed for IPC between ResetTask and Task_ReadPort.*/
void *MsgQStart [QStartMessagesSize]; /* Let the maximum number of
message-pointers at the queue be QStartMessagesSize. */
SmartOS_EVENT *MsgQPW; /* Needed for IPC between Task_PW and Task_ReadPort. */
void *MsgQPW [QPWMessagesSize]; /* Let the maximum number of message-pointers
at the queue be QPWMessagesSize. */
SmartOS_EVENT *MsgQAppl; /* Needed for IPC between Task_Appl and Task_ReadPort.*/
void *MsgQAppl [QApplMessagesSize]; /* Let the maximum number of
message-pointers at the queue be QApplMessagesSize. */
SmartOS_EVENT *MsgQApplClose; /* Needed for IPC between Task_Appl and
Task_ReadPort. */
void *MsgQApplClose [QApplCloseMessagesSize]; /* Let the maximum number of
message-pointers at the queue be QApplCloseMessagesSize. */
8. /* Define both queues array sizes. Assume maximum 16 strings can be sent in
a queue. */
#define QStartMessagesSize 16; /* Define size of start message-pointer queue
```

```
when full */
#define QPWMessagesSize 16; /* Define size of password message-pointer queue
when full */
#define QApplMessagesSize 16; /* Define size of application message-pointer
queue when full */
#define QApplCloseMessagesSize 16; /* Define size of application message-pointer
queue when full */
9. /* Define Semaphore initial values, 0 when using as an event flag and 1 when
resource key. */
SemReset = SmartOSSemCreate (0); /* Declare initial value of semaphore = 0 for
using it as an event flag from ResetTask. */
SemPW = SmartOSSemCreate (0); /* Declare initial value of semaphore = 0 for using
it as an event flag from Task_ReadPort */
SemAppl = SmartOSSemCreate (0); /* Declare initial value of semaphore = 0 for
using it as an event flag from Task_ReadPort */
/* Define a top of the message pointer array. QMsgPointer points to top of the
Messages to start with. */
MsgQStart = SmartOSQCreate (&QStart [0], QStartMessagesSize);
MsgQPW = SmartOSQCreate (&QPW [0], QPWMessagesSize);
MsgQAppl = SmartOSQCreate (&QAppl [0], QApplMessagesSize);
MsgQApplClose = SmartOSQCreate (&QApplClose [0], QApplCloseMessagesSize);
.
.
10. /* Any other SmartOS Events for the IPCs. */
.
.
11. /* Code similar to Example 4.5, except the main function codes. These were
for reading from Port A and storing a character. Here, we have Port_IO and one
status flag, STAF_IO. It is reset at the beginning. */
/* Prototype Declarations for modem Port_IO input and output strings. */
char [ ] Str; /* Port_IO input string to hold the data from the host through
the demodulator circuit of modem. */
char [ ] ApplStr; /* Port_IO string, which the modem transfers to host after
modulation. */
unsigned char *portIndata; unsigned char *portOutdata;
void portIO_ISR_Input (*portIndata); /* Prototype declaration for receiving an
input character. */
void portIO_ISR_Output (*portOutdata); /* Prototype declaration for sending an
output character. */
/* Start of Port_IO Input Interrupt Service Routine */
void portIO_ISR_Input (*portIOdata) {
disable_PortIO_InIntr ( ); /* Function for disabling another interrupt from port
IO input. */
/* Insert Code for reading Port I/O bits portIOdata = &Str; */
/* Start of Port_IO Output Interrupt Service Routine */
void portIO_ISR_Output (*portIOdata) {
disable_PortIO_OutIntr ( ); /* Function for disabling another interrupt from
port I/O output */
/* Define a macro for sending a String */
unsigned byte i;
#define SendStr (&ApplStr) (
portIOdata = &ApplStr; i = 0; STAF_Out = 1;
```

```
while (STAF_Out = =1 && ApplStr [i] ! = NULL) {
portIOdata = ApplStr [i]; / *Pick a character from the queue message. */
portIO_ISR_Output (&portIOdata); /*Send it to the Port_IO for the modem output.*/
i++; /* Be Ready for next Character */
STAF_Out =0; /*Modem interrupt when one character sent by setting STAF_Out again.*/
}
ApplStr = ; /* Clear the Queue message for the new one*/
&ApplStr = ApplStr [0];
} /* End of the macro function SendStr. */
/* Define a macro function for string comparison. Note: 'C' function strcmp is
available at a C library. In order to optimize codes, we are not using strcmp
library function, but our own macro here. */
#define boolean strcmp (ApplStr, Str) {
.
.
}
/*-------------------------------------------------------------------*/
/* Define a macro for receiving a string */
#define ReceiveStr (&Str) {
while (STAF_In != 1) { }; i=0;
/* Execute interrupt service routines for each character received at modem
Port_IO */
while (STAF_In == 1 && Str [i] != NULL) {
portIO_ISR_Input (&portIOdata);};
STAF_In =0; /* Remember as soon as Port A is read STAF will reset itself to reflect
next interrupt status. */
Str [i] = portIOdata;/* Write port I/O input array element from the returned
data*/
i++;}
} /* End of the macro function ReceiveStr. */
12. /* Start of the codes of the application from Main. Note: Code steps are
similar to Steps 9 to 17 in Example 8.16 */
void main (void) {
/* Initiate SmartOS RTOS to let us use the OS kernel functions */
SmartOSInit ( );
13. /* Create Reset task, ResetTask that must execute once before any other.
Task creates by defining its Identity as ResetTask, stack size and other TCB
parameters. */
SmartOSTaskCreate (ResetTask, void (*) 0, (void *) &ResetTaskStack
[ResetTask_StackSize], ResetTask_Priority);
14. /* Create other main tasks and inter-process communication variables if these
must also execute at least once after the ResetTask. */
15. /* Start SmartOS RTOS to let us RTOS control and run the created tasks */
SmartOSStart ( );
/* Infinite while-loop exits in each task. So never there is return from the
RTOS function SmartOSStart ( ). RTOS takes the control forever. */
16. } / *** End of the Main function ***/
/*-------------------------------------------------------------------*/
*/
/* The codes of the application reset task that main created. */
17. static void ResetTask (void *taskPointer){
18. /* Start Timer Ticks for using timer ticks later. */
```

```
SmartOSTickInit ( ); /* Function for initiating RTCSWT that starts ticks at the
configured time in the SmartOS configuration preprocessor commands in Step 1 */.
19. /* Create three tasks defined by three task identities, Task_ReadPort,
Task_PW, Task_Appl and the stack sizes, other TCB parameters. */
SmartOSTaskCreate (Task_ReadPort, void (*) 0, (void *) & Task_ReadPortStack
[Task_ReadPortStackSize], Task_ReadPortPriority);
SmartOSTaskCreate (Task_PW, void (*) 0, (void *) & Task_PWStack
[Task_PWStackSize], Task_PWPriority);
SmartOSTaskCreate (Task_Appl, void (*) 0, (void *) & Task_ApplStack
[Task_ApplStackSize], Task_ApplPriority);
/* Declare requestHeader */
unsigned char [ ] requestHeader;
unsigned char [ ] requestStart;
20. while (1) { /* Start of the while loop*/
/* Code for retrieving two strings requestHeader for user PIN and request for
host PIN string from the protected file structure. */
.
.
/* Write the array elements after encryption. */
ApplStr = SmartOSEncrypt (requestHeader, DES); /* Using an RTOS function encrypt
requestHeader and post it in message queue. */
SmartOSQPost (MsgQStart, ApplStr);
*ApplStr = SmartOSEncrypt (requestStart, DES); /* Using an RTOS function encrypt
requestStart and post it in message queue. */
SmartOSQPost (MsgQStart, ApplStr);
SmartOSSemPost (SemReset); /* Post Semaphore event flag. */
21. /* Suspend with no resumption later, the Reset task, as it must run once
only for initiation of timer ticks and for creating the tasks that the scheduler
controls by preemption. */
SmartOSTaskSuspend (ResetTask_Priority); /*Suspend Reset Task and control of
the RTOS passes to other tasks of waiting execution*/
22. } /* End of while loop */
23. } /* End of ResetTask Codes */
/*************************************************************************/
24. static void Task_ReadPort (void *taskPointer) {
while (1) {
25. /* Wait for IPC from ResetTask. */
SmartOSSemPend (SemReset, 0, & SemErrPointer);
26. /* Wait for a message for requestHeader from queue MsgQStart. */
&QStart = SmartOSQPend (MsgQStart, 0, & QErrPointer);
SendStr (&QStart); /* Send it to modem Port_IO. Not after sending the message
in string QStart becomes null, . */
27. /* Wait for a message for requestStart from queue MsgQStart. */
&QStart = SmartOSQPend (MsgQStart, 0, & QErrPointer);
SendStr (&QStart); /* Send it to modem Port_IO. */
/* Receive and decipher a string from the modem Port IO. */
ReceiveStr (&Str);
ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input
string from the host */
28. /* Code for saving the Host PIN and if verified, then application commands
from the host is also saved. The savings are at the protected file structure.*/
```

```
.
SmartOSSemPost (SemPW); /* Post event flag for requesting a password at MsgQPW.*/
SmartOSTimeDly (100); /* Delay for 100 ms so that let lower priority task Task_PW
run. */
29. /* Wait for a message for requestPassword from queue MsgQPW. If available,
send request and wait
for the password. */
&QPW = SmartOSQPend (MsgQPW, 0, & QErrPointer);
SendStr (&QPW); /* Send password request to modem Port_IO. */
ReceiveStr (&Str); /* Receive a String from the modem Port IO. */
ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input
string for password from the host */
30. /* Code for verifying the deciphered user password at the protected memory
or file data. If verified, then application commands from the host by posing
event flag SemAppl. */
.
.
SmartOSSemPost (SemAppl);
SmartOSTimeDly (100); /* Delay for 100 ms so that let lower priority task Task_PW
run. */
29. /* Wait for a message for requestAppl from queue MsgQAppl. If available,
send request and wait for the application command and user data. */
&QAppl = SmartOSQPend (MsgQAppl, 0, & QErrPointer);
SendStr (&QAppl); /* Send password request to modem Port_IO. */
ReceiveStr (&Str); /* Receive a String from the modem Port IO. */
ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input
string for application command and user data from the host */
30. /* Code for using the user data and executing the application command received.*/
.
.
31. /* Using an RTOS function, encrypt request closing request and post it in
a message queue.The closing request is from a message queue MsgQApplClose. Then
retrieve it from queue in QApplClose after encryption. */
.
.
&QApplClose = SmartOSEncrypt (requestApplClose, DES);
Send (*QApplClose);
ReceiveStr (&Str);
ApplStr = SmartOSDecrypt (Str, DES); /* Using an RTOS function, decrypt the input
string for password from the host. */
/* Compare deciphered string with message Closure Permitted. If found equal,
then closure permitted, then delete this Task and other low priority tasks. */
If (strcmp (ApplStr, Closure Permitted) {
SmartOS_Task_DelEn = 1 /* Enable task deletion by SmartOS. */
OSTaskDel (Task_ReadPortPriority); OSTaskDel (Task_PWPriority); OSTaskDel
(Task_ApplPriority);};
} /* End of While loop */
/* End of Task_ReadPorts Codes. */
/
/*************************************************************************/
40. static void Task_PW (void *taskPointer) {
while (1) {
```

```
41. /* Wait for IPC from Task_ReadPort. */
SmartOSSemPend (SemPW, 0, & SemErrPointer);
42. /* Code for retrieving one string from the protected file structure. It is
for requesting the password from the user at the host of the card. */
.
.
43. /* Write the array elements after encryption. */
ApplStr = SmartOSEncrypt (requestPassword, DES);
SmartOSQPost (MsgQPW, ApplStr);
SmartOSTimeDlyResume (Task_ReadPortPriority); /* Resume Delayed task
Task_ReadPort. */
44. } /* End of While loop */
45. /* End of Task_PW Codes. */
/
/*************************************************************************/
46. static void Task_Appl (void *taskPointer) {
while (1) {
SmartOSSemPend (SemAppl, 0, & SemErrPointer); /* Wait for IPC from Task_ReadPort.
*/
47. /* Code for retrieving one string for requesting the application commands
requestAppl from the protected file structure. It is for requesting the password
from the user at the host of the card. */
.
.
48. /* Write the array elements after encryption. */
ApplStr = SmartOSEncrypt (requestAppl, DES);
SmartOSQPost (MsgQStart, ApplStr);
49. /* Resume Delayed task Task_ReadPort. */
SmartOSTimeDlyResume (Task_ReadPortPriority);
} /* End of While loop */
50. }/* End of Task_Appl Codes. */
/*************************************************************************/
```

本章小结

- MUCOS RTOS 函数用在巧克力自动售卖机的嵌入式系统中。在第 1 个任务初始化后，通过 7 个任务控制各种机器功能：(a)系统中用户硬币有 3 种面值 Rs.1、Rs.2 和 Rs.5。(b)收取盒子中的硬币。(c)缺少零钱的情况下，退出全部硬币。(d)投入的硬币总额有多余时，从通道中退出多余的硬币。(e)通过端口投放巧克力。(f)依据机器状态显示消息。(g)显示时间和日期。它显示和 MUCOS 相同类型的信号量如何用作事件标志、资源键和计数器。它还介绍了如何使用 MUCOS 和系统 RTC。
- 对于生成 TCP/IP 栈后的网卡驱动，使用 VxWorks RTOS 嵌入式系统代码。示例代码给出了发送字节流到网络上的代码设计方法。首先字节流通过 TCP/IP 协议在多个层之间传递。网络驱动器最后将字节流送到 TCP/IP 网络上。示例给出了 VxWorks 如何调度 6 个任务。(a)一个任务检查应用程序插入的应用程序字符串。(b)一个任务创建传输控制层的应用程序字节流。(c)两个任务在传输层，使用 TCP 和 UDP 两种协议中的任意一种插入报头域。(d)一个任务通过分割 TCP 段字节流，创建网络传输的 IP 报文。(e)一

个任务驱动网卡，将字节流放入管道(一种虚拟设备)。示例还给出了任务如何处理报头域中的多个数据类型。该案例研究对 VxWorks RTOS 中的系统 RTC 应用程序和各种函数进行详细的说明。可以使用 VxWorks 为嵌入式网络系统制定开发方案。VxWorks 提供了使用 IPC、二进制和互斥信号量类型、消息队列和管道的好处。我们在任务中预先定义了发送到队列和管道中的消息的字节数。为优化存储器，这里没有使用 C 中的 typedef。

- 嵌入式系统应用在许多汽车电子领域，这里我们总结了几个要点。本章描述了现代汽车中一项功能特性，汽车行驶控制系统。它定义了自适应控制系统。由于从人身安全的角度考虑，系统的可靠性要求较高，所以用于汽车的 OS 中需要一些特殊性能。OSEK-OS 是汽车电子控制的标准 RTOS。为了在后面可以使用 OSEK 开发汽车应用的嵌入式系统，文中首先解释了它和 VxWorks 和 MUCOS 的差别。然后给出了汽车巡航系统的案例研究。在示例编码时，使用保留并集成了 OSEK-OS 特性之后的 VxWorks 进行代码设计。
- 本章还给出了一个智能卡的案例研究，描述了案例中需要的特殊嵌入式硬件和特殊 RTOS 函数。智能卡的案例研究表明，对于嵌入式系统代码的开发，RTOS MUCOS 或者 VxWorks 是不够的。我们使用一个理想的 RTOS——SmartOS 进行代码设计，SmartOS 具有 MUCOS 的特性，以及嵌入式系统中必需的加密特性和文件安全、访问条件和限制访问权限。执行系统启动的初始化任务执行之后，SmartOS 继续调度 3 个任务。(a) 一个任务从卡数据文件中读取应用程序字符串，在加密后，作为消息发送给 UART。它接收来自主机的 UART 的加密字符串。该示例也介绍了如何通过同样的任务处理多个函数，以减少存储器的需求。这是智能卡要求的一项特性。在大部分情况下只有 8KB 的存储空间，极少的情况下才有 64KB。请求密码的任务通过该任务的 IPC 和应用程序进行交互。最后，在得到主机许可后，删除任务。(b) 一个任务发送来自用户的密码请求，并和主机交互。(c)一个任务用于获取执行期望的应用程序例程，以及得到来自主机的用户数据的命令。

关键词及其定义

- 应用层：由一些附加的域组成，这些域是在将消息放到网络上，以便其他端点的应用程序可以了解请求和服务之前附加的。
- 自适应算法：一种参数匹配的调整算法，限制控制系统中的变化扰动。
- 自适应控制：使用自适应算法生成输出控制信号的控制系统。
- Big Endian：首先将整数的高位字节放到网络上或者存储器中。
- 电荷泵：二极管和真空管的组合，可以释放和存储电荷，连接到调整电路后向系统提供电源。
- 校验和(checksum)：表示 8 位、16 位或者 32 位数字相加时产生的进位数的和。
- CRC(Cyclic Redundancy Check，循环冗余校验)：它是一个通过计算得到的 32 位或者 16 位整数，如果在传输过程中出现错误，则这个错误可以通过将它同接收消息的 CRC 进行比较来检测到。它的计算时间开销比较大，但是使用效果比校验和好。

- 连接定向协议：在这种协议下，网间通信时首先建立连接，然后消息在一种流控制机制下流动。在完成合适的网间通信后，中止连接。传输控制协议 TCP 就是该协议的一个例子。
- 无连接协议：在这种协议下，网间通信在没有首先建立连接、没有流控制机制、没有网间通信的连接中止的情况下发生。通常在广播模式下可以见到该协议。用户数据表协议 UDP 是该协议的一个例子。
- 加密软件：对消息加密和解密或者设置字节流的软件。它使用一种算法加密，使用另一种算法解密。
- 数据采集系统：从多个端口和通道采集数据。
- 数据报：和前面的字节流无关的字节流。UDP 数据报的最大长度为 2^{16} 字节。
- DES：数据加密标准。
- DES3：DES 的一个版本。
- 电机设备：使用电子信号操作机械系统的设备。
- 生产密钥：在生产卡的时候嵌入到 ROM 的密钥，使得卡具有惟一的标识。
- “internet”层：通过 TCP/IP 协议互联的层。它由一些域组成，这些域是在通过路由器将消息放到网络上之前附加的，以便路由器可以依据层中的源和目标 IP 地址发送消息。该层的域用于网络流控制机制。例如 TCP/IP 报文中的 IP 报头域。
- 无效锁：如果将该锁置于卡的应用程序数据文件中，则卡就会无效，不能继续使用。
- Java Card：智能卡应用程序的 Java 语言格式。
- JVM(Java Virtual Machine，Java 虚拟机)：执行 Java 类的管理代码，这些类被 Java 编译器作为字节码编译。代码在计算机系统中 JVM 的帮助下运行。
- LCD 阵列：一组液晶显示屏，由行和列组成，类似一个阵列。使用适当的控制器将字符或者图像显示在阵列上。
- 逻辑地址：RTOS 或者应用程序中指令和数据字节使用的存储器地址。
- 网络驱动器：向网络发送消息和从网络接收消息的物理设备卡，用于不同层之间字节流的物理连接。驱动器可以在依据其他的 Ethernet 协议放置头字节和尾字节之后，使用 SLIP 或者 PPP 协议驱动网络。每个驱动器都有惟一的地址。
- 用于汽车系统的 OS：具有很高的可靠性，并提供了不会导致死锁、优先级反转或者不可预测的延迟等特性的操作系统。
- OSEK-OS：用于汽车嵌入式系统软件的操作系统。
- 私有密钥：测试智能卡电路后植入卡中的密钥。卡自身的受保护存储区域，以及任务实际运行期间物理和逻辑地址之间的转换方案，都是卡私有的。在插入该密钥后，RTOS 和应用程序仅使用逻辑地址，处理器在两类地址转换期间使用该密钥。
- PIN：个人身份验证号。银行或者主控服务分配该 PIN。分配单位也有自己的 PIN，即主 PIN。
- 保护位：ROM 中的一位，处理器使用该位以防止受保护部分的指令和数据被放到总线上。处理器从外部阻塞对这些受保护地址的写周期。
- 雷达(无线电探测和测距)：使用 1 米以下的无线电波，通过测量发射信号和接收信号之间的延迟，进行短距离对象的测距。

- RSA：使用质数的一种算法。RSA 分别代表该算法的 3 个发明者姓名的首字母，他们是 Ron Rivest、Adi Shamir、Leonard Adleman。
- SHA(Security Hash Algorithm)：基于散列函数的安全散列算法。
- SLIP(Serial Line Interface Protocol)：串行线接口协议。
- 机动车辆的车队稳定：在高速公路上或者执行 VIP 的护航任务时，保持车队中的多个汽车之间距离的恒定。
- TCP 报头：依据 TCP 协议在传输控制层置入报文的头信息。
- 油门：控制引擎前进或者后退加速的阀门。
- 传输控制层：将 TCP 头域，或者 UDP 头域放到 TCP/IP 网络上的网络层。
- UDP：传输控制层发送 TCP/IP 报文的协议。
- UDP 报头：包含 4 个域的报头，这 4 个域包括源和目标端口地址，长度和校验和信息。
- 解锁 PIN：用于解锁卡存储器的特定部分，以便使用该部分的主控 PIN。例如，在解锁(通过修改访问条件域而允许访问)存储器中的密码文件后允许用户修改密码。

问题回顾

(1) 为什么设计人员在设计代码之前必须了解系统需求？什么是在使用 RTOS 函数之前必需的任务列表和同步模型？

(2) 可扩展闪存的含义是什么？

(3) 为什么智能卡应用程序流行使用 Java 语言？

实践练习

注意：

问题和练习 4~7、8~13、14~19 和 20~25 分别属于第 11.1 节、11.2 节、11.3 节和 11.4 节的内容。

(4) 第一个任务的 while 循环只包含任务挂起函数。请解释原因。

(5) 解释读端口的任务如何同端口设备驱动器同步。

(6) 为什么当可用计数值对于计算从端口收取的硬币金额可选时，选择使用 SemAmtCount 信号量？

(7) Task_ Display 的作用是什么？如何为 LCD 阵列上的多行消息编写代码？

(8) 请解释在 ACVS 任务中 SemMKey 的用法。任务中的 SemMKey 和互斥量用法有何区别？

(9) 请解释如何有效使用邮箱 IPC 消息。为什么不使用消息队列代替多个邮箱？

(10) 在示例 11-2 中按 CDFG 对任务建模。

(11) 解释所使用的每个信号量的作用。信号量如何使用，以及示例 11-1 中的代码在对系统进行了以下修改时，如何作相应的修改？使用随机号码生成器 C 函数的 ACVS，平均插入 8 个硬币就免费投放一个巧克力，并退回得到幸运巧克力的所有硬币。

(12) VxWorks 中优先级分配区间同 MUCOS 的优先级分配区间有何不同？

(13) 请解释如何在传输层，使用信号量指导 UDP 代替 TCP 的使用。

(14) 请解释语句 fd = open (netDrvConfig.txt, O_RDWR, 0);的用法。

(15) 如何处理长度大于报文和 MTU 允许长度的字节流？
(16) 请解释 ACC 中的任务列表。
(17) 汽车的 OSEK 标准将系统任务分成 4 类。这样做的好处是什么？
(18) OSEK 区分处理 ISR 和任务。这样做的好处是什么？
(19) 为什么 OSEK 不允许任务发送消息队列？
(20) 请解释优先级分配。
(21) 列出示例 11-3 代码中使用的宏，以及每种宏的用法。
(22) 弱接触智能卡硬件如何获得电源供应？
(23) 为什么使用带有存储器保护位的处理器很重要？
(24) 使用生产密钥、私有密钥、应用锁和 PIN 进行加密的好处有哪些？
(25) 用表格列出智能卡 OS 需要的特性。

第 12 章　嵌入式系统中的软硬件协同设计

本章前所学内容

回顾第 1 章，我们定义了一个嵌入式系统，在这个系统中，硬件部分嵌入了作为系统重要组件的软件部分。“嵌入式系统”一般具有以下 3 个主要组件：

- 硬件
- 主要应用软件(这些应用软件可以同时执行一系列任务或者多个任务)
- RTOS

我们已经学习了有关嵌入式系统的 3 个主要组件：

(1) 嵌入式系统硬件由处理器、存储设备、I/O 设备和基本硬件单元——包括电源、时钟电路和复位电路。

(2) I/O 设备由访问外设和其他片上或者片下单元的 I/O 端口组成。物理设备有 UART、调制解调器、收发器、定时器计数器、小键盘、键盘、LED 显示单元、LCD 显示单元、DAC 和 ADC 以及脉冲拨号电路。

(3) 软件定时器驱动的实时时钟。

(4) 虚拟设备。

(5) 嵌入式系统中的设备驱动程序和中断处理机制。

(6) 在进行高速运算，组织处理器、存储设备和 I/O 设备时，需要使用处理器指令进行功耗管理。

(7) 在进行系统性能优化时，选择合适的处理器和存储设备。

(8) 连接存储器和 I/O 设备的系统总线的接口技术，以及通过启用可以直接访问系统存储器的 I/O 单元，来改善系统性能的 DMA 控制器。

(9) 高级语言编程原理、程序模型、软件工程方法、RTOS 和 IPC 的使用，以及 RTOS MUCOS (μC/OS-II)和 VxWorks 函数的使用范例。

(10) 研究以下 4 个领域中的 MUCOS 和 VxWorks RTOS 编程工具的案例：(i)消费类电子系统。(ii)通信和网络系统。(iii)自动电子系统中的控制系统。(iv)安全 SoC 系统。

因此，我们已经学习了嵌入式系统的体系结构、编程、软件设计和开发过程。

有两种方式用于嵌入式系统设计：(1)在进行系统设计时，软件生命周期结束，并且将该软件集成到硬件的过程的生命周期开始。(2)当协同设计一个时间紧急的复杂系统时，可以两个周期同时进行(SoC 中的协同设计与之不同)。当系统实现时，最终的设计会给出符合预定目标的

嵌入式系统以及最终产品。所以，(a)软硬件设计以及两者集成于系统的理解和(b)软硬件协同设计是嵌入式系统设计的重要方面。

我们可以看看 Jean-Louis Berlet 在论文 *Exploring Hardware/ Software Codesign with Vertex-II Pro FPGA*(Xcell Journal, pp. 24~29, Summer issue, 2002)中的阐述。当被问及成功完成设计所需的专业知识时，Berlet 的回答是："软件设计人员必须了解硬件设计的种类和硬件设计小组可能遇到的问题类型。还必须了解硬件的可行性及其性能。同样，硬件小组的所有成员必须对软件以及应用程序如何操作有充分了解。双方成员还必须正确地理解对方的语言和意愿。"

普林斯顿大学嵌入式系统开发小组的 Wayne Wolf、Burak Ozer 和 Tiehan Lu，最近在 *Smart Cameras as Embedded System IEEE Computer* 中报导有新的研究发现。这篇论文论证了在进行复杂嵌入式系统开发时，硬件设计过程中对软件的正确选择，以及软件设计过程中对硬件的可行性和性能的了解，是尤为重要的。

本章的目标就是让您深刻理解这些问题。

本章将学内容

为了实现以上所提的目标，必须学习如下内容：

(1) 系统项目管理。

(2) 系统软硬件协同设计周期的行动计划。

(3) 目标系统电路或者仿真器的使用。

(4) 内置电路仿真器(In-Circuit Emulators，ICE)的使用。

(5) 模拟器的使用，在源代码设计工具中基本模块的使用，以及原型开发工具的使用。

(6) IDE (Integrated Development Environment，集成开发环境)的使用。

(7) 使用简单 LED、示波器、分析仪和位率测量仪进行的硬件测试。

(8) ROM 中系统监控代码的调试。

在嵌入式系统设计中，有一些特定的问题需要开发小组解决。这些问题包括，在系统开发过程中独立的软硬件设计和软硬件协同设计问题，OS/RTOS 中选择合适的处理器、存储器、设备和总线以及移植的问题。我们会在后面的学习中了解这些问题。

12.1 嵌入式系统项目管理

回顾 7.8 节和表 7-9。其中解释了使用 Pressman 著名的 4 P，包括人(People)、产品(Product)、过程(Process)和项目(Project)，来进行软件项目管理。嵌入式系统项目管理与此类似。本节涉及到的人员是指软件开发团队、硬件开发团队和系统集成工程师。表 7-9 列出了 4P，但是嵌入式系统项目管理的含义还包括这 4P 的组织问题。表 12-1 给出了在嵌入式系统项目的开发过程中如何组织上面提到的 4P。

表 12-1 嵌入式系统项目管理的 4 要素

要　素	任　务	需要避免的问题
人员 高级管理人员	同表 7-9 中的定义	同表 7-9 中的定义
项目技术管理员或者团队领导	(i)选择软硬件开发过程中使用的软硬件语言、工具和软件开发过程的生命周期模型；(ii)协调和重组可用的软硬件规范、详细设计、组件以及现有的进程；(iii)-(v)在表 7-9 中有描述	对实现人员的感受缺乏理解和不协调的开发
实现人员	使用建模、源代码设计、测试、模拟、调试和产品验证工具来实现软硬件的开发过程和集成过程	不遵循已达成共识的设计方案，在实现人员之间缺乏协调性
嵌入式系统的客户	指定产品及其质量需求，并同高级管理人员协商费用	干预开发过程，在达成一致意见之后私自更改产品说明书
终端用户	在建议的范围内使用产品	不按照说明书使用产品。例如，在结冰的路上使用 ACC 控制系统驾驶汽车(11.3 节)。
产品(嵌入式系统)	同表 7-9 中的定义	缺少正确的产品说明书
过程	使用分层模型(表 9-1)，在应用程序和硬件之间将系统分层，并将过程调整到和表 7-9 中的类似	错误划分和适应错误模型
项目	嵌入式系统项目管理目标是，创建基于表 7-9 中所列出的标准的成功产品	不正确的计划、错误的工作、预算，对成功与否不够关注，让开发人员忙于一些与项目无关的活动

12.2 系统开发过程中嵌入式系统设计和协同设计问题

12.2.1 嵌入式系统开发过程的目标

在嵌入式系统开发过程的最后阶段需要达到的目标是，生成一个完全通过测试和验证的系统。

12.2.2 行动计划

回顾软件开发过程生命周期模型。参见 7.2 节和表 7-1。其中也给出了嵌入式系统开发过程的行动计划。即使是简单的小规模嵌入式系统也需要有一个详细的计划。

定义行动计划是设计任务系统的第一步。图 12-1 给出了系统开发阶段设计系统的行动计划。

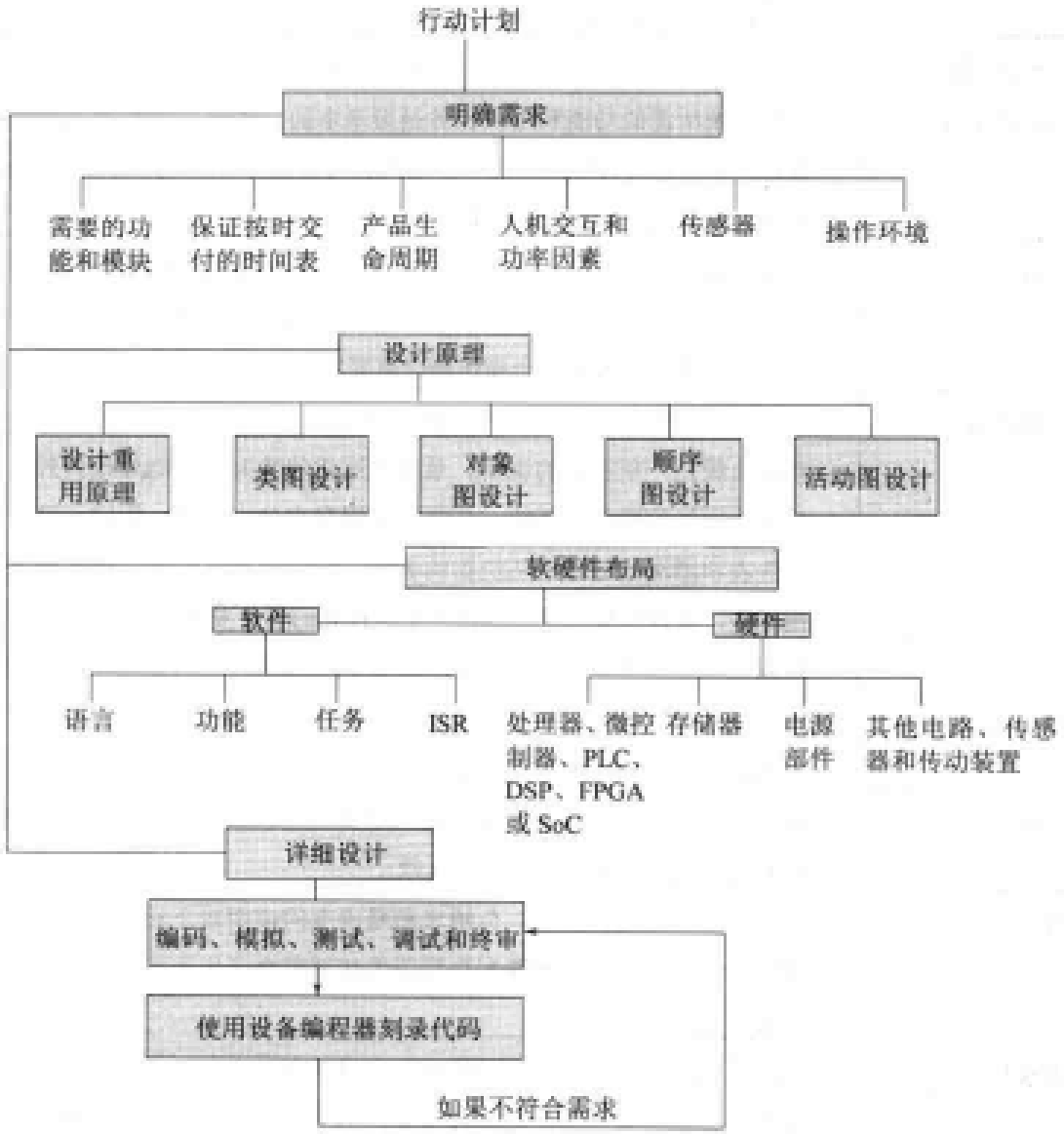


图 12-1　系统开发过程中设计系统的行动计划

考虑一个为自动洗衣机开发小型嵌入式系统的计划(“衣进衣出”的嵌入式系统)。我们带着这样的观点来看这个例子：通过它很容易了解任何嵌入式系统开发过程中都可能需要的行动计划。对于用户来说，这只是一个简单的“衣进衣出”系统。而对于嵌入式系统设计者来说，它不仅仅是一个软件。对于一般的系统设计者，它是“字节进字节出”系统，而不是衣进衣出的机器。设计者必须依据一个非常完备的行动计划进行系统设计。下面就是系统开发过程所需要的一些步骤。

12.2.3　完整的规范和系统需求

第一步是完全了解所需系统的规范。表 12-2 的第 1 列给出的规范必须全部由开发团队提供。

表 12-2 系统规范

系统规范	解释
产品功能和任务	了解系统所需的功能和用户任务是最基本的。考虑一个机器人的嵌入式系统设计。希望它具有哪些功能？移动所有部件(腰、肩、肘、手和手指)需要的自由度？它应该完成哪些任务？是否需要智能？
交付时间表	交付时间表非常重要。合理的交付时间表通过适当调整线性序列开发过程，强制结合使用面向对象方法、第 4 代工具和可用的硬件设计，或者使用通用处理器，都可以形成一个高速的快速开发模型
产品生命周期	产品的生命周期。如果产品生命周期较短，则开发团队需要频繁更改设计
系统负载	系统负载是一项重要的说明。系统在过载的情况下可能导致失效。比如一个自动洗衣机的嵌入式系统。负载按衣服多少分小、中、满缸。这些区别将会在洗衣和漂洗的过程中从机器的容量、时间和加水深度上体现出来。另一个例子是，处理图像、视频剪辑和实时视频的硬件和处理器会有不同的负载
人机交互	规范中需要回答下面这些问题。什么是人机交互？这体现在小键盘输入和显示屏输出的规划中。显示屏显示什么内容以及如何规定显示方法？这个问题的答案体现在接口电路和显示屏的编程上。例如，电视机遥控器的小键盘
操作环境	操作温度、湿度和环境参数说明也是必需的。系统有可能会在高山上失效，或者可能在高温真空的情况下失效
传感器	传感器的灵敏度、精度、分辨率和精确度的说明对于设计是否符合需求很重要。视频会议图像传感器和视频图像传感器会产生不同像素的分辨率、格式和速率的系统输入
功耗需求和环境	系统负载越大，需要的功耗就越大。设计有源系统时需要有关智能省电的软件设计和硬件设计的功耗管理方案。对于连续电源供应和间断电源供应条件下的系统操作应有不同的说明。连续电源供应情况下的设计比较简单，因为它不需要存储空间频繁地保存系统状态
系统开销	系统的最大可承受开销必须指明，以确定项目对于开发团队和团队所要做的工作量来说是否可接受

注意：

以下系统规范必须在设计过程开始之前准备完毕。(i)产品功能和任务；(ii)交付时间表；(iii)产品生命周期；(iv)系统负载；(v)人机交互；(vi)操作环境；(vii)传感器；(viii)功耗需求和环境；(ix)系统开销。

12.2.4 原理设计

第二步是系统的原理设计开发。我们将要讨论的问题有：系统开发过程的模型是什么？关于这个问题可以回顾一下 7.10 节的内容。可以使用 UML 来开发原理设计模型。原理设计可以使用 UML“用例图”、“对象图”、“顺序图”、“状态图”、“类图”和“活动图”。原理设计对于

应用软件和硬件结构布局的开发很有帮助。

注意：

UML“类图”、“用例图”、“对象图”、“顺序图”、“状态图”有助于原理设计的开发，以及获取应用软件和硬件的结构布局。

12.2.5 软硬件布局设计

第三步是系统软硬件结构布局的开发。有两种方式：

(1) 紧跟集成之后的独立设计方式：在设计系统时，软件生命周期结束，同时将该软件集成到硬件的过程的生命周期开始。

(2) 并行协同设计方式：当协同设计一个时间紧急的复杂系统时，两个周期同时进行。

当开发软件布局时，需要回答这样的问题：所需软件模型是什么？以下是自动洗衣机的3个示范模型：

(i) 获得用户输入信息，使用LED输出提供与机器进行人机交互的软件模块。实现人机交互功能的示例，如：(a)在启动的时候显示LED的当前默认用户设置。人机交互可以是智能的(默认设置可以依据用户的喜好。比如冬天喜欢穿羊毛制的衣服，而在夏天喜欢穿棉纱制的衣服)。(b)一个按键可以循环设置3种可能的衣服负载(如果不是自动识别负载)。LED及时显示所选内容。当用户设置了期望的负载时，停止按键。如果处理器发现在响应端口没有继续按键，就将输入内容装载到EEPROM存储器中。(c)另外一个按键可以循环设置衣服类型。它可以是羊毛、聚脂和棉纱3种类型。模块也将存储这些用户指令。

(ii) 如果有一个用户指令要启动，而该指令可在按下按键，或者收到一个远程切换信号甚至是一个Internet消息(在Internet兼容系统中)之后发出信号，则必须有另外一个软件模块初始化该循环过程，并调度每个循环周期。这个软件模块初始化洗涤、漂洗、脱水周期。(a)这些过程必须在确保衣服已经放进洗衣机内，并且关闭了手拉门之后进行。(b)电源输出必须符合要求。(c)连接发动机的电源和开关必须符合要求而且可用。(d)如果刚开始需要初始化洗衣周期，那么在EEPROM中必须设有机器状态的重启功能。(e)如果机器状态根据机器先前的运行周期中发出的一些中断来指示剩余时间，则它应该可以自己决定选择哪些动作继续。

(iii) 另一个模块是启动洗衣周期。功能示范如：(a)打开进水口。(b)进水口必须在水位传感器阵列显示水位超过了给定衣服所需的水位时关闭。进水口也应该在预定时间内关闭，实现对水位传感器的输入监控。(c)以慢速或者中速模式启动发送机。

软硬件实现工具的说明

开发整体布局和详细设计之前最重要的问题是：开发过程的要素(软硬件说明)是什么？这个问题可以通过实现表12-2提出的说明所需的要素来回答。例如，对于自动洗衣机的硬件需求的要素是：(a)发动机汽缸。(b)水位传感器阵列。(c)发送机和电子电路的电源。(d)通用嵌入式处理器或微控制器。例如，内部带有足够ROM、EEPROM和RAM的8位微控制器、定时器、中断处理器、外设串行或端口控制器。(e)按人体特征设计的按键阵列和LED。输入键用于接收用户指令。LED指示机器状态以及完成和剩余时间。(f)接口电路。(g)220V电源以及进水口和出水口阀门。

我们可以参考不同的系统硬件单元(见表1-5)，也可以参考描述硬件要素的1.2节和1.3节。

注意：

有两种设计方式。第一种是独立设计紧跟在系统集成之后。这种方式是指，在设计系统时，软件生命周期结束，同时将该软件集成到硬件的过程的生命周期开始。另一种方式是在复杂嵌入式系统的设计中需要采用的并行协同设计。即当协同设计一个时间紧急的复杂系统时，两个周期可以同时进行。有许多软硬件设计工具，可以帮助我们完成简单的工作，很容易实现系统的设计。

12.2.6 详细设计

第四步是目标系统和代码的详细设计，涉及到处理器和存储器的选择(参见2.2节和2.4节)。这一步还需要决定在软件和硬件上需要实现的功能。12.7.1节将描述设计问题，并讨论如何选择合适的开发平台。软件和硬件布局有助于后面的详细设计，包括软件代码的实现和得到目标系统的电路的实现。

12.2.7 实现工具

可以参考表12-3，其中列出了系统开发过程中用到的硬件工具。

表12-3 详细设计所用的硬件工具

硬件工具	应用
仿真器	用于仿真目标系统的电路，与特定的目标系统和处理器无关，可以在大多数目标系统的开发阶段中使用，这些目标系统将来和特定的微控制器芯片结合。仿真器为在单一系统上(而不是多目标系统上)开发不同应用程序提供了很大的灵活性和便利性。它可以独立工作，也可通过串行线连接到PC上工作
内置电路仿真器	一种仿真器电路，用于仿真目标处理器电路，但必须使用扁平电缆以串行的方式连接到PC或者微控制器上。它可以在开发阶段仿真不同型号的微控制器
逻辑分析仪	通过多输入线路(如24或48线)从总线、端口等收集大量总线事务(大约128或者更多)的电源工具。它将这些信息显示在监视器(显示屏)上，用于调试实时触发条件。这有助于在指令执行时顺序查找信号
设备编程器[1]	设备的编程系统，可以是PROM EPROM芯片，或者微控制器中的单元，或者、PLA、 GAL、PLC。将计算机和该电路连接起来，使用计算机上的软件工具，对插入插槽(在设备编程器上)的设备向每个地址传输字节的过程进行编程。设备程序需要输出定位器和输出记录程序。这些输出必须反映最终的设计结果，或者附带压缩记录的启动程序，处理器在嵌入式系统开始执行之前进行解压缩

1. 详细内容参见附录G.2。

注意：

用于硬件设计和系统集成的硬件工具有仿真器和内置电路仿真器。软件工具有模拟器、编辑器、编译器、汇编器、源代码设计工具、性能评测器(用于查看每个函数或者指令集消耗的时间)、存储示波器、代码执行查看器、存储器和代码覆盖示波器。

12.2.8　测试

为了在初始阶段便于进行测试，我们将这个问题分成几个小部分来讨论。明确定义每个阶段的输入和输出，加以标识，并画出数据流图(data flow graph，DFG)(参见 6.1 节)。

回顾 7.7 节和 7.8 节。其中的一项测试技术是调用中断服务例程。使用 assert 宏是另外一项重要的测试技术。例如，给出这样一条指令，“assert(pPointer !=NULL);”。当 pPointer 变成 NULL 时，程序就中止。将该代码插到一段程序中，该程序的内容是检查条件或参数实际变成 true 还是 false。如果为 false，则程序中止。我们可以在应用程序中不同的关键地方使用 assert 宏。

当把所有的模块合到一起时，中间的每个阶段以及最后阶段都必须进行测试和调试。在测试和调试时，应该认真遵循这样的规则，“不能肯定是正确的，那么它就是错误的”。为每个阶段编写详细文档也是必要的。

可以参考表 1-6，了解用于汇编语言编程、高级语言编程、RTOS、调试的软件工具，以及软件需求规范确定的系统集成工具。

12.3　嵌入式系统开发阶段中的设计周期

图 12-2 给出了(a)嵌入式系统开发过程中的设计周期，以及(b)实现阶段的编辑-测试-调试周期。在开发阶段也有编辑-测试-调试周期。原来选择的处理器部分仍然不变，而应用软件代码必须通过多次运行和测试进行完善。处理器的开销相对较小，但开发最终目标系统的开销就相对较高，而且比硬件电路的设计需要更多的时间帧。

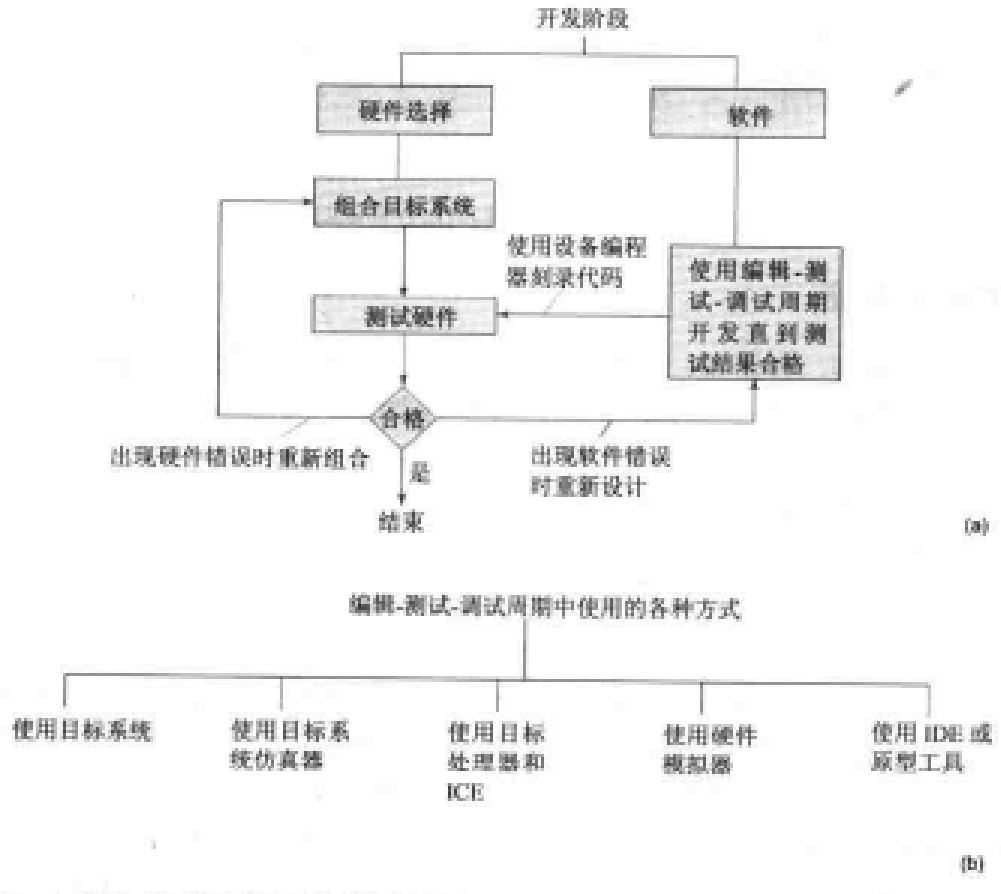


图 12-2　(a)嵌入式系统开发过程的设计周期；(b)开发过程的实现阶段中的编辑-测试-调试周期

开发者使用以下 4 种方式来完成编辑-测试-调试周期。

(1) 使用目标系统。

(2) 使用中间工具 ICE(内置电路仿真器)(参见 12.2.2 节)，仅在目标系统上使用处理器。

(3) 使用不带硬件的模拟器(参见 12.5.2 节)。

(4) 开发者使用原型工具或 IDE(参见 12.5.3 节和 12.5.4 节)。

12.4 目标系统或其仿真器和内置电路仿真器(ICE)

图 12-3 中的(a)和(b)分别给出了简单和复杂的目标系统示例。

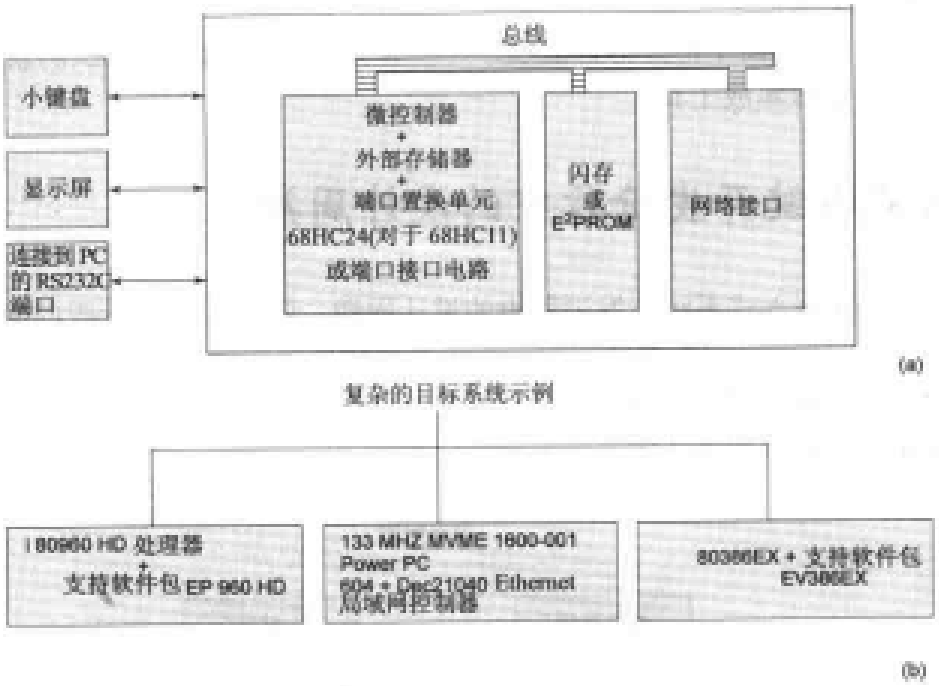


图 12-3 (a)简单的目标系统示例；(b)复杂的目标系统示例

12.4.1 使用目标系统

目标系统包含处理器、存储器、外围设备和接口。一些目标系统有 8MB 或者 16MB 的闪存和 64MB 的 SDRAM(参见 2.3 节，了解存储器的类型)。目标系统还可能有 RS232 以及 10/100-Base 以太网连接或者 USB 端口(参见 3.3.3 节，了解 USB)。

目标系统和最终系统不同。它和计算机结合起来可以很好地工作，如同一个独立的系统。在开发阶段可能需要重复将代码下载到其中。目标系统或者它的副本只在后面作为嵌入式系统工作。

考虑一个开发过程中的目标系统。必须在目标系统的开发阶段编写应用软件代码，比如路由器。这些代码必须嵌入到存储器、闪存、EEPROM 或者 EPROM。它们必须使用模拟器和调试工具反复编写和修改，直到编辑-测试-调试周期中最终的测试表明，它可以按照规范中的要求进行工作，才进行嵌入操作。此后，设计者可以简单地将其复制到最终系统或者产品中。另外，最终的系统可以使用 ROM 代替目标系统中的闪存、EEPROM 或者 EPROM。

带有 i80960HD Intel 处理器的主板就是一个目标系统的示例。支持软件包源自 Intel Cyclone

EP960HD，它有一个 8KB 的数据缓存和 16KB 的指令缓存。另一个支持软件包来自 Intel 公司的 Intel 386 Ex 处理器，即 Intel EV 386Ex。

WindRiver 目标系统主板支持 133MHz MVME 1600-001 PowerPC 604。它有 16KB 的指令缓存，16KB 的数据缓存和 256KB 的 L2 缓存，一个片上 PowerPC 时间工具和浮点单元。它有 DEC 21040 PCI Ethernet LAN 控制器。通过 16K 缓冲区的套接字发送 TCP 报文(在套接字中最多可以排 1600 个 TCP 报文)。它所带的 VxWorks RTOS 支持大约 1MB/s 的 TCP 吞吐量。

我们再考虑一个复杂目标系统的示例 VxWorks 5.4。它通过可伸缩的 RTOS 支持、Internet 协议支持、POSIX 库支持、文件系统和图形支持提供了运行时支持。它包含一个调试代理。而且对特殊处理器或者微控制器有支持软件包。目标系统通过带有 ICE(参见 12.4.2 节)的目标服务器工具、Ethernet 或者从主机引出的串行线并行连接主机上的仿真器。图 12-3(a)和(b)分别给出了简单和复杂目标系统的两个示例。

12.4.2 仿真器和 ICE

我们可以放弃为获得实际的嵌入式系统而复制的目标系统，而采用一个保持与特定目标系统和处理器无关的独立单元吗？答案是肯定的。就是使用仿真器或者 ICE。它为在单一系统上(而不是多目标系统)开发不同的应用程序提供了很大的灵活性和便利性。图 12-4(a)和(b)分别给出了一个仿真器和一个 ICE。

ICE 和仿真器有何区别呢？仿真器使用由微控制器和处理器自身组成的电路。仿真器可以模拟具有扩展存储器的目标系统，并在编辑-测试-调试周期具有代码下载能力。ICE 使用另外一个带卡的电路，这个卡通过插座和目标处理器(或电路)相连。

仿真器有许多子单元，表 12-4 将其列出并进行了解释。

表 12-4 仿真器和 ICE 子单元

仿真器子单元[1]	动　作
接口电路	下载 ROM 镜像到 EPROM 中，并将 RAM 字节从 PC 下载到仿真器中。它使用 PC 的串口(COM RS232C)(图 12-4(a)和(b))。该电路有助于将来自 PC 的大型应用程序代码嵌入到程序存储器部分。代码可以在 PC 机上使用高级语言开发。例如，应用程序代码开发的设计者发现，编写庞大的应用程序比起使用仿真器上的 20 个按键输入机器码要容易得多
插座	插入通用处理器、DSP、嵌入式处理器或者微控制器的多管脚凹凸插座，它通过电缆(通常使用扁平电缆)和连接器与 ICE 连接(参见图 12-4(b)右角插座)
外部存储器	附加的 RAM 和 EPROM 或者 EEPROM，保证足够大多数目标系统以及它们的应用程序使用
仿真器板上的显示单元	单线 8(或者 12)字符显示器。逐个显示存储器地址中的内容。它还显示程序执行到每一步时寄存器的内容
20 键的小键盘	用户使用它可以直接在现场将数据和代码输入到存储器地址中。这些代码必须是机器码
寄存器	在系统测试阶段，用于单步，以及全速测试运行的附加系统寄存器
连接器	将该仿真器插入接口电路和其他一些典型的系统设备和外围设备。用于目标系统显示模块的连接器就是一个例子。还有一个例子就是用于 PC 接口电路的连接器

(续表)

仿真器子单元[1]	动　作
目标系统键盘	键盘用户输入板，等同于目标系统期望的键盘
目标系统驱动电路	例如，用于网络、发动机、电磁阀、反应堆或者打印机的驱动硬件
监控代码	它们位于仿真器 EPROM 或者 EEROM 中

1. Orion 仪器公司生产的一种仿真器，USA 嵌入了逻辑类分析仪设施(参见 12.6.3 节)。Intel 提供了仿真器以及对它的不同处理器和微控制器的支持软件包。

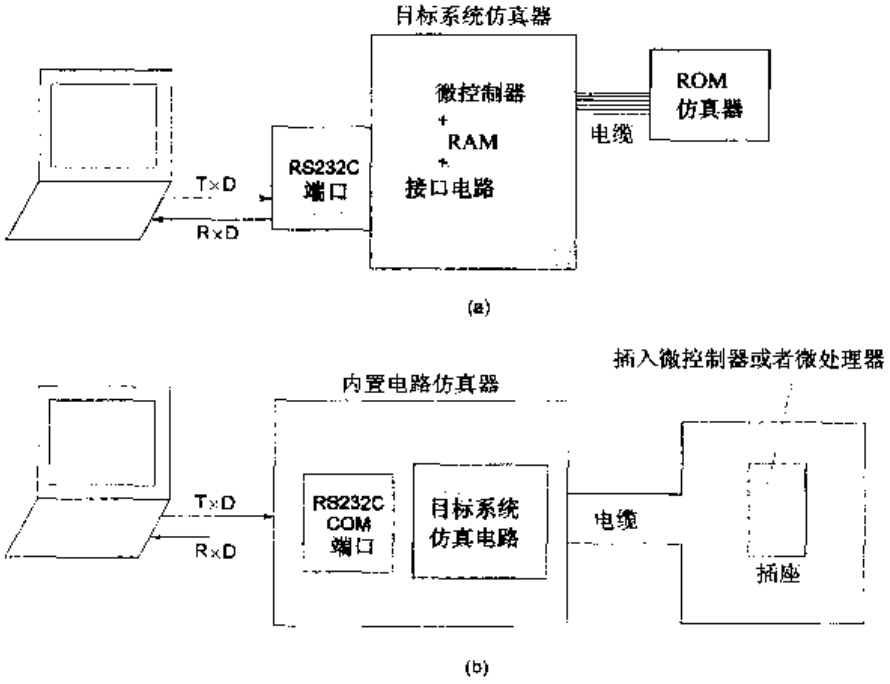


图 12-4　(a)仿真器；(b)ICE

ICE(内置电路仿真器)由以下部分组成：(i)带有扁平电缆的仿真器容器，它延伸到目标系统的处理器或微控制器插座(参见图 12-4(b))。后面会将处理器 IC 插到该插座上。然后，就可以测试目标系统(记住该系统对最终的嵌入式系统是一对一的复制)。为了避免由于电缆过长而产生的耦合电容的影响，使用的电缆必须尽可能短。该容器模仿了目标系统的微控制器或者处理器。(ii)该容器串联到计算机的 COM RS232C 端口。通过这个端口，该容器从计算机获得下载代码。用于仿真器的计算机程序对寄存器和存储单元进行完全监控。该容器在它的基本电路和扁平电缆跳线之间存在一些卡。这些卡的相互替换使得将 ICE 用于其他型号的处理器成为可能。

嵌入 ICE 的处理器自身有哪些特点？ARM7 和 9 的处理器(附录 B.1)的一个特性是都有 ICE 子单元。这有助于调试其硬件。

ICE 或者仿真器在开发阶段结束后就不再使用。通过复制使用 ICE 开发的代码构成电路。该电路在和目标处理器建立连接之后，由使用的处理器、必要的存储器芯片、按键和显示单元或者其他外设组成。它工作起来应该和我们在开发阶段后期，使用仿真器或者 ICE 完成的工作同样精确和完善。仿真器对于完成最终目标系统之前进行的系统开发非常有用。

Motorola 提供了 M68HC11EVM 和 M68HCEVB 作为基于 68HC11 微控制器的目标系统的仿真器。这些仿真器具有以下外部连接。

当使用 ICE 或者仿真器时，实现阶段所需要的软件是编辑器、汇编器、反汇编器、模拟器等(参见 12.5 节)。PC 仅用于下载代码到仿真器以及传回仿真器中不同地址上的代码和数据。即使非常熟练的设计者也需要使用 PC 来节省机器级编程时间，这个时间对于复杂应用程序来说可能会相当长。对于不同型号的微控制器，我们拥有附加的插座连接器：例如，用于仿真 68HC11 的 48 管脚型号和 52 管脚型号。

“visionICE I”是一种具有网络性能的 ICE。它结合了 10/100Mbps Ethernet 的连接性。这使得 ICE 在局域网上也可以使用。这样做的好处是可以进行远程调试。它可以连接到目标系统的串口。

ROM 仿真器(图 12-4(a))仅仅仿真了 ROM。目标系统通过 ROM 插座连接，同时连接到计算机。在编辑-测试-调试阶段，需要周期性地将代码下载到目标系统的闪存或者 EEPROM 中。而 ROM 仿真器消除了这样做的必要。

12.4.3 用于将最终代码下载到 ROM 中的设备编程器

假定从系统设计阶段直到目标系统设计阶段已经结束。要了解后者的存储器映射，请参考图 2-9。最终的系统必须植入非易失性存储器。定位器的输出是带有启动程序、系统程序(压缩或者未压缩)、初始数据和掩膜 RAM 数据的最终设计结果。参见 2.5 节。记住，在下载之前应该检查已下载代码在处理器从特定的处理器复位地址复位时是否工作。

参考表 12-3 了解设备编程器的定义。烧写(burning)是植入代码的过程。这里的代码是指那些依据 ROM 镜像(定位器输出)下载的代码。附录 G.2 解释了设备编程器的工作过程。烧制的二进制镜像必须反应自引导程序，并且某些时候还可以在嵌入式系统开始执行之前，解压缩系统程序(通过引导程序开始运行系统)。烧写是指在实验室使用设备编程器将代码烧写到一个已擦除的 EPROM、EEPROM 或者 PROM 中(提示：IDE 集成了设备编程器)。

另一种方案是，将应用程序代码和表格形式的数据发送给特定的生产商。这样就可以获得所需的 ROM 型号。ROM 型号尤其在需要批量生产时是必需的。然后集成目标系统的处理器、带 RAM 的存储器和其他硬件，就得到了最终产品。

12.5 嵌入式系统开发中的软件工具

12.5.1 代码生成工具(汇编器、编译器、加载器和链接器)

表 1-6 列出并解释了包括仿真器在内的 13 个软件模块和工具。高级语言是与机器无关的。其中可能会出现类似 X=X+23，或者 X=2*Y+V*Z+19 这样的表达式。每一种语言都需要相应的编译器。使用“Basic”这样的解释器是不能执行程序的。在为目标系统获得机器码时需要使用一定的工具。C 提供了高级语言工具(参见 5.1 节)。设计者使用编辑器编写 C 程序及助记符。结合 PC 上的鼠标键盘就可以进行程序代码的输入工作。

(1) 解释器(interpreter)将表达式逐个(逐行)翻译成可执行的机器码。

(2) 编译器(compiler)使用完整的表达式集合。它还包括来自库函数的表达式；也就是标准的库例程。解释器有助于代码的联机执行，而编译器有助于为了在后面获得可执行的机器码而进行脱机编程。C 程序可以使用解释器，也可以使用编译器。

(3) 汇编语言程序中有和机器相关的助记符。例如 SBC A, 0x0B。这是一条指令，用十六

进制数 0x0B 减去处理器中 A 寄存器的值，同时向前进位。汇编助记符是针对特定的处理器和微控制器的。它根据指令集中所提供的指令执行操作。汇编助记符需要解释器将其翻译成可以在特定处理设备上执行的机器码。

(4) 反汇编器将目标代码翻译成助记符形式的汇编语言。这有助于理解之前生成的目标代码。

(5) 汇编器是将汇编助记符翻译成二进制操作码和指令的程序，这里说的二进制操作码和指令也就是称为目标文件的可执行文件。它还创建一个可以打印出来的列表文件。该列表文件包括地址、源代码(汇编语言的助记符)和十六进制的目标代码。目标文件还包含在汇编语言程序的实际执行过程中需要调整的地址。加载器是在操作码和操作数加载到计算机存储器之前，帮助任务重新分配地址的程序。

(6) 链接器链接所需的目标代码文件和库代码文件。该操作在加载器重新分配地址，并将代码放到存储器的物理地址中，以及程序在计算机上运行之前进行。加载器在主机上实现的功能和它在目标系统上与设备编程器结合工作时的功能类似。

(7) 交叉汇编器在将一个微控制器或者处理器上使用的目标代码转换为另外一个微控制器或者处理器上使用的其他代码时非常有用。交叉编译器使得我们可以使用用于系统开发的 PC 上的处理器。它随后提供目标代码。在最终完成的系统中需要使用这些代码，而这个系统以后将使用另外的处理器或者微控制器。

12.5.2 模拟器

飞行员在驾驶飞行器或者战斗机之前，先使用飞行模拟器进行训练。同样，软件也可以模拟类似仿真器、外围设备、网络和 PC 上的输入输出设备等硬件单元。模拟器具有对特定目标系统的无关性。它用在系统的应用软件开发阶段，这里所说的系统将来会使用特殊的处理器或者处理设备芯片。模拟器实质上就是一种软件，用于模拟嵌入式系统电路的所有功能，这里的电路包括任何附加存储器、外设和总线。模拟器和实际的目标系统一样，使用交叉编译器、链接器和定位器。

模拟器在使用可能嵌入了 IP 或者 RISC 的 ASIC 的情况下会失效(ASIC 生产商和 RISC 处理器可以永久提供一种可选的调试工具。例如，ARM7 和 ARM9 处理器(参见附录 B.1)中的 ICE)。模拟器没有解决时序问题和硬件相关问题。回顾 6.3.9 节。模拟器可能不会显示共享数据的错误，因为它们仅仅是由某些特定情形下的中断引起的。例如，4 个寄存器中的一个长字被部分加载时就会产生中断。

模拟器在最终的目标系统完成之前对于系统的开发是很有帮助的，过程中可能仅仅使用 PC 作为开发工具。模拟器完全可用于嵌入式系统所使用的不同处理器和处理设备，而且，系统设计者和(或者)开发者在设计室开发应用软件和硬件时，不需要用于模拟器的代码。图 12-5 显示了使用模拟器时详细的设计开发过程。模拟器大都运行在 Windows 环境中。模拟器一般具有以下特性：

(1) 定义了用于目标系统的处理器或者处理设备系列及其不同型号。

(2) 监控执行过程进行到每一步时，源代码部分的详细信息，使用标签和符号参数表示。

(3) 提供执行过程进行到每一步时，已定义目标系统中 RAM 和端口(模拟)状态的详细信息。

(4) 提供已定义系统中外设(模拟，假设即将连接)状态的详细信息。

(5) 提供执行过程进行到每一步或者每一个模块时，寄存器的详细信息。它还监视系统响应并确定吞吐量。

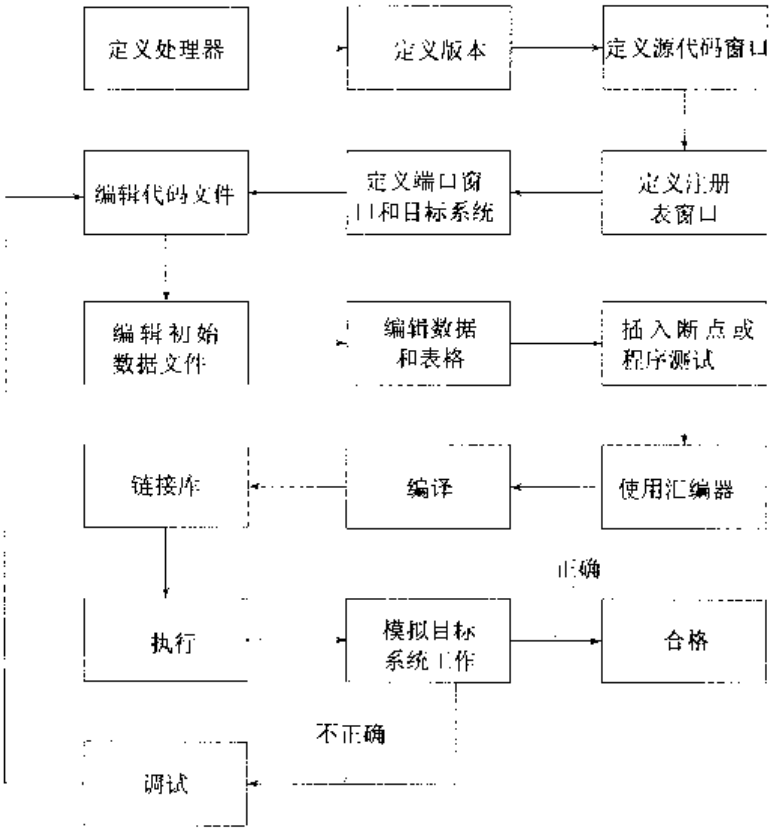


图 12-5　使用模拟器时详细的设计开发过程

(6) 显示屏上的窗口提供了以下内容：

(i) 定义的微控制器系统的堆栈、端口(模拟)的详细状态信息。

(ii) 执行过程中程序流程的跟踪信息。跟踪程序计数器和处理器寄存器的输出内容。它是汇编语言程序调试的重要工具。应用软件的跟踪是指函数逐步执行时所选变量的输出。跟踪示波器在程序执行时，显示 X 轴上的时间和 Y 轴上所选变量的变化关系(跟踪示波器是一个 Tornado®工具模块，用以跟踪随 X 轴时间的变化，模块和任务的变化。也可以生成一个动作列表)。

(7) 帮助显示屏上的窗口提供当前命令的具体功能。

(8) 当从键盘输入，或者从菜单中选择模拟器命令时，监控这些命令的详细信息。

(9) 结合了用于 C 语言表达式和汇编语言助记符(表达式)的汇编器、反汇编器、用户定义的击键或者鼠标选择的宏和解释器。用户定义的击键宏非常有用。比如说，定义击键为 1，用于提供端口 n 上的一个特定的输入字节和特定的 RAM 地址字节。

(10) 支持条件(可达 8、16 或者 32 个条件)和非条件断点。它具有指令执行无数次之后中止代码的功能。断点和跟踪都是重要的测试和调试工具。

(11) 易于实现内部外围设备和延迟的同步。

(12) 使用抢占任务的 RTOS 调度程序。

(13) 模拟来自中断、定时器、端口和外围设备的输入。从而可以使用它们来测试代码。

(14) 提供网络驱动器和设备驱动器支持。

(15) 回顾 9.6.6 节。考虑多任务循环调度。如何确定切换上下文之前的时间片？如何确定超时？使用模拟器，可以通过模拟热处理(Simulated Annealing)方法调度各种任务。它模拟了不同的固定调度，然后逐渐增加每个任务(定时器延迟设置)的调度，并且只要没有任务超过时限，就继续增加调度。

12.5.3 节描述了一个功能强大的模拟工具 VxSim。另外一个是 pSOSim，它模拟了 RTOS pSOSystem pSOS+â 内核。pSOSim 对工具的选择是无限制的，它吸收了表 12-5 所给出的特性。

表 12-5 模拟器 pSOSim 的特性

支持的特性	活　动
应用程序开发工具	支持 UML 和“RougeWave”。它提供了较短的设计周期
本地开发环境	支持多种本地开发环境和调试。开发环境可以是 MS Visual C++或者 GNU 工具
RTOS 的 API	模拟给定硬件的 RTOS 的许多 API
调试功能	强大的调试功能，使得查错成为一件非常容易的事情
设备模拟	模拟设备和设备驱动器的行为
网络模拟	网络模拟功能使其成为一个虚拟的测试平台，它允许建模复杂的多节点网络系统。如路由器或者网关。网络应用程序可以模拟内部子网或者真实的网络。当模拟网络时，它为不同的网络标准协议生成堆栈，甚至包括 IP 多播和 IP 广播
用户接口	例如模拟一系列的盒状接口

12.5.3 嵌入式系统的原型开发、测试和调试工具

表 1-6 描述了其他的嵌入式系统开发工具，它们并不是调度程序的一部分。但是在使用 RTOS 调度程序时这些工具非常有用。一个重要的概念是代替目标系统硬件使用的原型开发工具。例如 WindRiver®的 Tornado Prototyper，它用在带有一系列工具的集成交叉开发环境中。Tornado Prototyper 中的一系列原型工具也实现了 RTOS VxWorks(参见 10.3 节)。这些工具通过浏览器进行模拟、编译和调试。浏览器在开发阶段总结最终的目标嵌入式系统的完成状态。表 12-6 给出了来自 WindRiver 的一系列原型工具的特性。

表 12-6 Wind River 原型工具

工　具	特　性
ScopeProfile	该工具动态执行性能评测器以类似示波器波形的形式，让我们看到 CPU 在哪些地方耗费了时间，从而了解性能瓶颈所在。它显示了处理器在任务或者 ISR 中的每个任务上消耗的处理器时间
存储示波器(MemScope)	存储器的使用是嵌入式系统中的一个关键问题。是否浪费存储空间？是否出现存储器泄漏错误？存储器泄漏是指指针增加到了无符号区域，导致任务和堆栈溢出，或者在数组的最后一位后面写入内容。存储示波器给出了存储块的使用情况。它检测由于系统调用或者其他移植模块导致的存储器泄漏

(续表)

工　具	特　性
软件示波器(StethoScope)	如同帮助医生诊断用的听诊器，它动态跟踪任何程序变量的改变，跟踪参数的改变，使我们了解正在执行的多线程(任务)执行序列的情况。它还记录整个过程的历史信息
跟踪示波器(TraceScope)	该工具帮助跟踪 X 轴的时间变化和动作列表中各项的变化信息(表 9-11 中给出了期望时间刻度的动作列表)。跟踪示波器使我们在任务切换期间可以查找到 RTOS 调度程序的行为，并注明各个 RTOS 动作发生的时间
代码测试存储、跟踪和覆盖示波器	这些工具通过动态存储器分配的分析、被控流视图跟踪和各种实字(real word)情形下代码覆盖来帮助测试代码。代码覆盖的研究有助于删除特定应用程序中不必要的额外代码和函数。它提供了可伸缩的系统
VxSim	强大的模拟器工具，提供开发和调试代码的虚拟目标。它有助于避免在嵌入式系统的实际目标板上重复加载代码。使用 VxSim 模拟应用程序在开发阶段早期非常有益，这是因为 RTOS 任务调度在目标系统实现之前可以完全模拟出来
VxWorks 网络堆栈	这是另一个可以加快代码开发进程的强大工具。VxWorks RTOS 为发送到 Internet 上用来测试高性能交换设备的数据准备堆栈。堆栈的属性取决于所选的本协议。协议如下：RFC-1323、CIDR、IP 多播、IP、UDP、TCP、DNS 客户端、DHCP 服务器、SMTP 服务器、RIPv1 支持、RIPv2 支持、ARP、Proxy ARP、BOOTP、RLOGIN 客户机和服务器(用于 Telnet)。这样嵌入式系统的套接字就可以连接到多协议 LAN、ATM 网络、SONET 或无线访问和智能网络

12.5.4 集成开发环境

我们可以有一个集成开发环境(Integrated Development Environment，IDE)，它由具有编辑器、编译器、汇编器等的模拟器，仿真器、逻辑分析仪和 EPROM/EEPROM 应用程序代码刻录机组成。IDE 必须具有如下特性：

(1) 可以方便地定义处理器系列及其型号。IDE 包含源代码设计工具(参见 5.11 节)，该工具集成了编辑器、C 和嵌入式 C++编译器、汇编器、链接器、定位器、逻辑分析仪等工具，从而方便用户使用。

(2) 有用户可定义的编译器，用以支持新型号或者新类型的处理器。提供多用户环境。

(3) 设计过程分成几个子部分。给每个设计者分配既相互独立又有联系的任务。

(4) 在 PC 机上模拟仿真器、外设和 I/O 设备等硬件单元。支持条件和无条件断点。

(5) 可以单步调试。容易实现内部外设的同步。

(6) 在显示屏上提供窗口界面。在这些窗口中显示继续执行时源代码部分的详细信息，用标签和符号参数、寄存器表示，以及继续执行时外围设备的状态、RAM 和端口的状态、堆栈和程序流状态的详细信息。

(7) 使用仿真器和逻辑分析仪验证目标系统的性能。仿真器已植入开发系统，并和特定目

标系统无关，逻辑分析仪可用于分析多达 256 或 512 个地址总线和数据总线上的事务。

WindRiver®系统的 Tornado Tool 3 就是一个 IDE 工具，它使用了 VxWorks RTOS(参见 10.3 节)。Tornado Tool 体系结构的特性是“动态链接并递增加载对象模块”到目标系统。该工具支持的目标处理器系列有 PowerPC、Intel、Motorola、Pentiums、MIPS 和 ARM/Strong ARM。它有助于原型开发和测试原型应用程序。它有一个含 GNU C/C++编译器的文本编辑器。可以在 3 个级别上进行调试工作，源代码级、任务级(调度、IPC 和中断研究)和域级。该工具还包括 VxSim、软件示波器和跟踪示波器。图 12-6(a)和(b)分别给出了一个简单和复杂的 IDE。

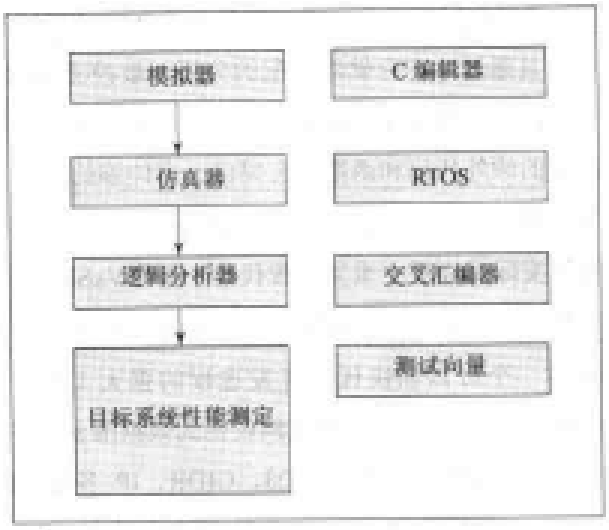


图 12-6 (a)简单 IDE；(b)复杂 IDE

用于 RTOS pSOSystemä 的 IDE 是 pRISM+。它支持的目标处理器系列有 PowerPC，Motorola 68K，Intel80x86，Pentiums ，MIPS、ARM 和 Mitsubishi M32/R。该工具有包含 pRISM+编辑器、SNIFF+(参见 5.8 节)的源代码设计工具，有交叉编译器、源级调试器、对象浏览器、逐个查看事件和运行时行为的监视器。对象浏览器浏览应用程序行为的超时。它使用图形显示 pSOS RTOS 任务、队列、信号量和 IPC 对象，实时分析(Real-Time Analysis，RTA)套件描述代码覆盖，并定位运行时错误。该工具还优化了存储器的使用。

注意：

代码生成工具用于创建和编译代码。使用模拟器和许多最新的工具，如存储示波器、软件示波器，以及存储器和代码覆盖示波器，进行代码测试。

12.5.5 存储器、处理器敏感程序和设备驱动程序

源代码设计的标准工具、仿真器、模拟器和原型系统，对存储器、处理器敏感程序和设备驱动程序没有给出优化方案。表 12-7 给出了存储器、处理器敏感程序和设备驱动程序的示例。

表 12-7 硬件敏感程序

程 序	示 例
处理器敏感程序	回顾 2.1 节。处理器有不同类型的结构单元。它有存储器映射的 IO 或者 IO 映射的 IO。IO 指令是处理器敏感指令。处理器可能仅有定点 ALU。当需要浮点操作时，在定点处理器上的处理和在带有浮点操作的处理器上的处理不同。处理器可能不提供 SIMD 和 VLIW 指令(参见 6.3 节)。需要这些指令的模块的编程对于不同的处理器也是有区别的。汇编语言有时可以帮助优化使用处理器的特殊功能和指令。某些处理器提供编译器或者优化编译器子单元的功能，避免使用汇编语言进行编程
存储器敏感程序	(i)存储器敏感程序的例子有视频处理和实时视频处理。不包括可接收的失帧数量每秒可以处理多少帧？这取决于在使用指定的实时编程模型和算法时，任务可用的存储器和延迟。(ii)回顾 2.5.4 节。IO 设备寄存器、缓冲区、控制寄存器的存储器地址和用于中断源或者源组的向量地址在微控制器中是预先指定的。RTOS 不可能把这些地址分配给任务。存储器敏感程序也必须通过熟练的汇编语言编程来优化存储器的使用
设备驱动程序	不仅 I/O 设备寄存器、缓冲区、控制寄存器的地址，中断源或者源组的向量地址是预先指定的，而且控制寄存器位和状态寄存器位使得设备的强大功能特性得以体现。使用每个位进行编程，有助于程序员挖掘设备的所有特性。使用汇编语言编写的设备驱动程序可能会变得非常有用。这些驱动程序提供了对 I/O 端口、寄存器、堆栈和 RAM 的直接控制，而且很容易编程(参见第 4 章的示例)。参见 4.1.4 节和 9.12 节，有许多 Linux 端口、总线和物理媒介附件可用的开放源码的驱动程序。Linux 的设备驱动程序允许像使用进程那样使用设备寄存器注销和调度的每个模块。由于这些也都是开放源码，所有程序员可以直接移植它们

注意：

对于处理器敏感程序、存储器敏感程序，以及将中断服务例程植入系统，需要适当的接口函数。对于设备敏感程序也需要适当的驱动程序。

12.5.6 动态链接库

回顾前面所学的内容，像 square root ( )这类标准库函数可以节省程序员的编码时间，并且嵌入式系统中还有一些新的库函数，专门用于 C 或者 C++编译器。如 delay( )、wait( )或 sleep( )。再回顾示例 5-1。我们在预处理器命令中包含了 VxWorks 函数头文件“vxWorks.h”，信号量函数库头文件“semLib.h”，多任务管理函数库头文件“taskLib.h”，消息队列函数库头文件

“msgQLib.h”，文件设备 I/O 函数库头文件“fioLib.h”，用于系统函数(比如用作实时软件定时器的系统定时器)的系统库头文件“sysLib.c”，和协议处理函数库头文件“prctlHandlers.c”。

编译器在编译的时候，通常会链接源头文件和库文件中的库函数。因而，嵌入式系统的 ROM 镜像中都含有所需的就绪可用的库函数。这表现了简单的静态可执行文件的优点。

回顾 6.1.2 节。来自一个确定节点的 CDFG 图形无关路径可能很少执行。编译过程中进行静态链接时，存储块被执行频率很低的库函数占有。如果可以动态链接这些函数，那么 RAM 存储空间的使用就可以通过运行时函数 melloc( )和 free( )得到优化。这些函数可以被动态调用来链接程序，并在使用之后释放存储空间。链接程序可以通过系统的 I/O 端口，或者其他主机或服务器部分读取。

假定有一个分布式嵌入式系统，通过网络连在一起。库函数可能只是在某些特定的控制条件下需要。这种情况下，我们可以使用进程间通信、RPC(参见 8.3.7 节)调用必需的函数，并将它们和系统程序动态链接。

虽然使用动态链接库时可能出现存储器错误，但是这样做节省了存储空间(在测试不正确的情况下才会出现这里所说的存储器错误)。这是因为库中的函数可以加载或卸载，而且只在运行时才被调用。然而，DLL 和非系统必需的函数比较起来，应该更加简短，并且对需要的函数的划分更细致。

DLL 有助于在系统配置发生变化时，重新配置系统。

注意：

如果可以从某个源(端口或网络)连接库函数，并且这些函数可以节省存储空间，就考虑使用动态链接库(DLL)。

12.6 示波器和逻辑分析仪在系统硬件测试中的使用

在使用了模拟器、ICE 以及 ROM 中的调试代码之后，我们在调试的最后阶段使用故障检测硬件诊断工具，该工具可以记录随时间变化的状态信息和随其他状态变化的状态信息。

12.6.1 逻辑探测器或者 LED 测试

要记住，数字信号仅仅是两个电压范围之间的离散信号。在 CMOS 逻辑中，“1”是指电压处在 V_{DD}~$0.66V_{DD}$ 的范围内，“0”是指电压处在 $0.33V_{DD}$~V_{DD} 的范围内，一般情况下 V_{DD} 是指接地线，它的电压是 5V。而模拟信号就不同。逻辑探测器是最简单的硬件测试设备。它是一个类似 LED 设备的手持笔，当探头接触到测试点(端口或者硬件管脚)并且位是“1”时，LED 发绿光，如果位是“0”时，则发红光。如果测试点在“1”位，逻辑探测器的 LED 就会快速闪烁，如果在“0”位，则不发光。该设备另外一端通过导线连到地线上。

当研究端口上的长延迟效应(>1 秒)时，逻辑探测器便成为一个重要的工具。用于延迟，然后使用逻辑探测器发送端口处结果的小程序，将测试 OS 定时器的节拍。

12.6.2 示波器

最后，代码下载硬件需要在完成编辑测试和调试周期之后，使用模拟器或 IDE 进行测试。示波器是带有显示屏的一种显示装置，它显示两个信号的电压随时间变化的情况。它还可以显

示模拟信号和数字信号随时间变化的情况。

使用示波器时必须具有 DC(直接耦合)输入。另外一个输入终端总是保持接地电势。术语 DC 不应该和直流电相混淆。示波器有两种输入选择，DC 和 AC。DC 是指将输入直接耦合到显示器的输入放大器。AC 是指在输入给电容器后再输给放大器(垂直面放大器)。当使用 AC 方式输入时，信号形状可能会失真。只有需要在窗体中查看信号时，才使用示波器，当平均超时(给定信号的可选组件)时，信号放大倍数为 0(这里的平均是指简单的平均，而不是均方根的平均)。如果使用 AC 方式查看总线信号，就可能显示错误的过冲和下冲。

运行的时钟会在示波器上显示其状态。在相连的两个上升沿之间的水平间隔给出时钟的周期。例如，使用 12MHz 晶振的 8051，每 0.825 毫秒就会有一个时钟状态。在两个示波器的输入放大器上同时检查时钟状态和 ALE(地址锁存使能)，可以测试处理器的活动。另一个示波器用于检查实时时钟例程和脉宽输出例程。使用示波器很容易对实时软件进行测试和调试。串口上的输出信号或者并口测试的输出位会提供重要信息。示波器也可用于测试延迟时间例程。在 3 个寄存器中设置 3 个延迟参数，并注明端口位在每次运行时的改变所花费的时间。我们根据这 3 个时间来估计用于请求延迟的寄存器中的实际设置。然后在该延迟参数设置下运行，并使用示波器和微调进行测试，以得到准确的延迟设置。

使用示波器的好处是可以作为噪声检测工具和电压表。还有一个用处是检测一个时钟周期内“0”和“1”两个状态之间的突变。这可以用于调试总线故障。存储示波器(storage scope)是另外一种型号的示波器，可以存储时间信号对。后面我们将根据这些数据来分析存储活动。

12.6.3 逻辑分析仪

逻辑分析仪可以很方便地调试小型嵌入式系统。这是一个比示波器的功能更强大的工具。示波器仅仅查看和检查两个线路。而逻辑分析仪是一个强大的软件工具，可以检查载有地址数据、控制位和时钟的多个线路。分析仪只能识别离散的电压状态，“1”和“0”。

分析仪有多个输入线路(24 或者 48)，可以同步连续地收集、存储和跟踪多路信号。连接来自系统和 IO 总线、端口和外围设备的线路，同时收集多个总线事务(大约 128 个或者更多)的持续时间。稍后，它可以使用该工具在计算机显示器(屏)上显示每个线路的每个事务，也可以打印显示内容。每个输入线路的相位差也可以提供重要的信息。逻辑分析仪还可以调试实时触发条件。在执行指令时，帮助顺序查找总线信号和端口信号状态。逻辑分析仪的变种也可以在需要时提供模拟测量。

另外，逻辑分析仪可以在水平轴上显示状态，而不是所显状态模式下的时间。当显示逻辑状态时，特定的线路可以在另一个线路上显示“0”和“1”的变化关系。

逻辑分析仪也可以通过连续和重复运行系统来记录间断产生的某些 Bug。逻辑分析仪对由于 bug 而导致的程序中止束手无策，它不能显示处理器寄存器和存储器的内容。如果处理器使用高速缓存，那么只检查总线根本没用。正如在模拟器上那样，我们也不能修改存储器的内容和输入参数。这些改变的影响是不可见的。

12.6.4 位率测量仪

位率测量仪是一种测量设备，它在预先选择的时间区间查找“1”和“0”的数量。如何测量吞吐量(即网络上每秒传输的字节数)？重复发送 0xA55A(二进制表示为 1010010101011010)。1 秒钟内接收到的“1”的数量除以 4 就是以字节每秒(Byte/s)为单位的吞吐量。同样可以查找

另种位模式，并找到在给定时间内所期望的位数。估计测试消息中的位“1”和位“0”，然后使用位率测量仪检查它是否和消息匹配。

12.6.5 用于 ROM 调试的系统监控代码

ICE 中的下载代码可以在自举电路上运行“Power On Self Test”(POST)程序。ROM 也可以有调试监控器。当嵌入式系统耦合到计算机的 RS232C COM 端口或者网络端口时，使用 gdb，它是一种 GNU 调试器，而且是一个可下载的免费软件。它为系统提供了调试监控代码，这些代码随同定位器二进制镜像被下载。

目标板生产商也提供调试监控代码和主机使用的调试工具。下面是使用这些装置时，调试动作的命令。(i)重置程序，或者启动(重启)程序；(ii)对系统存储器地址的读写操作；(iii)对系统处理器的寄存器的读写操作；(iv)当前或者指定地址的单步执行。

12.7 嵌入式系统设计中的问题

12.7.1 选择合适的平台

嵌入式系统的系统设计还涉及到选择合适的平台。平台由许多单元组成。表 12-8 列出了这些单元及其对应的章节，设计者在选择单元以最终得到合适的平台和开发工具时，可以参考这些内容。

表 12-8 为最终得到合适的平台和合适的开发工具，可以选择的单元列表

可选的单元	描述其详细内容的章节
处理器	[1]1.2 和 2.2 节
ASSP	1.2.6 节
多处理器	1.2.7 节
片上系统	1.7 节
存储器	2.4 节
系统的其他硬件单元	1.3 节
总线	3.3 节、3.4 节和附录 F.2
软件语言	[2]5.1 节、5.8 节和 5.9 节
RTOS(实时编程操作系统)	9.4 节、9.11 节、10.2 节和 10.3 节
代码生成工具	汇编器、编译器、加载器、链接器、仿真器、模拟器和调试器等调试和系统集成工具(1.4.7 节和 12.5 节)
最终将软件嵌入到二进制镜像的工具	附录 E.2

1. 处理器可以插入编译器作为子单元。例如，Philips TriMedia TM-1300 有一个优化 C/C++编译器的子单元(TriMedia 在这里指的是数据、声音和图像处理功能)。

2. 设计者或者开发者为什么使用 C++或者 Java 语言呢？参考 5.8 节和 5.9 节来回答这个问题。设计者为什么应该使用 C 语言？参考 5.1 节的内容。

12.7.2　嵌入式系统处理器的选择

(1) 无处理器的系统

回顾 1.2.1 节。可以选择另一种部件来代替微处理器、微控制器或者 DSP 吗？图 12-7(a)给出了使用 PLC 代替处理器的情形。我们可以将 PLC 用于自动"衣进衣出"这类系统(参见 12.2.2 节)。PLC 由可编程控制门、PAL、GAL、PLD 和 CPLD 构成。

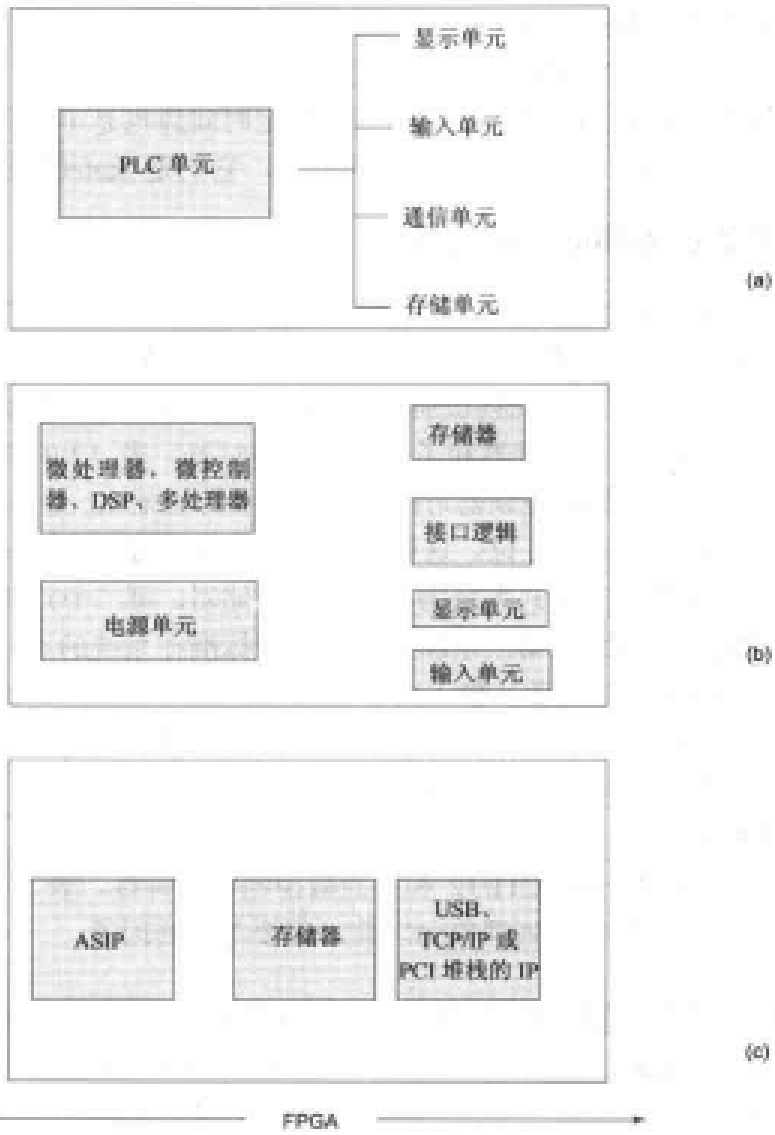


图 12-7　(a)使用 PLC 代替处理器；(b)使用微处理器、微控制器或者 DSP；(c)使用嵌入到 FPGA 中的 IP 来代替 ALU 进行函数处理

PLC 的操作速度非常慢，而且计算能力也很差。但是它具有非常强大的多输入输出的接口功能，具有系统相关的可编程能力。它在应用程序中的使用很简单，设计实现也相当迅速。巧克力自动售卖机就是 PLC 的另一个应用示例。

(2) 带有微处理器、微控制器或者 DSP 的系统

1.2 节给出了用于嵌入式系统的处理器的详细描述。图 12-7(b)给出了微处理器、微控制器或者 DSP 的使用。

(3) 在 VLSI 或者 FPGA 中带有 ASIP 的系统

图 12-7(c)给出了使用嵌入到 VLSI 或者 FPGA 中的 IP 代替 ALU 进行函数处理的情形。

设计中的动作序列可以是：使用 IP，利用 VHDL 等工具进行综合处理，然后将结果嵌入到 FPGA 中。该 FPGA 实现最后的功能，如果使用 ALU 来实现，设计者将会花费大量的时间进行编码。

我们在 VLSI 上使用授权的 IP 来实现复杂的操作。每个 IP 可以使用 VHDL 或者 Verilog 在门级综合。VHDL(VLSI High Level Description Language，VLSI 高级描述语言)和 Verilog 是模拟和综合门级设计的语言。VHDL 实现了并发和同步问题，以及结构分层策略。Verilog 除了具有这些特性外，还使用 C 函数。所以，异常处理和定时问题也是可编程的。有两种语言用于编程和实现 FSM、状态迁移、并发、同步和行为分层，它们是 StateCharts 和 SpecCharts。

12.7.3 需要考虑的因素和必需的特性

回顾 1.2 节和表 1-6。应该选用何种处理器或者微控制器？当使用 32 位系统、16KB+片上存储器，以及需要高速缓存、存储器管理单元、SIMD、MIMD 或者 DSP 指令时，就使用微处理器或者 DSP。例如，视频游戏、视频识别和图像过滤系统就需要 DSP。

微控制器的优势是提供片上存储器和子系统，如定时器。表 12-9 可以帮助我们选择微控制器。在合理开销范围内使用的微控制器可能不支持这些片上需求。这样就要决定哪一个 ALU 才会满足需求，8 位、16 位还是 32 位。这样就可以决定哪个微控制器及其型号具有所需要的功能特性。第一个判断标准是嵌入式系统需要的片上存储器。第二个判断标准是每个系统需要的片上定时器和串行通信子系统。第三个和第四个判断标准在系统的设计中可要可不要。第五个判断标准是对输入捕捉(中断或者从输入内容中提取时间)和输出比较(输出和定时器内容等于比较寄存器时的中断)是必要的。第四个判断标准关于 PWM 或者 ADC 的片上可用性。8 位、16 位和 32 位微控制器的最新型号可以从 *giants* 的网站中找到，如 Intel、Motorola、Phillips 和 Microchip(用于 8 位系统)。

参考附录 C，了解 8051、68HC11/12 和其他微控制器的特性。表 12-9 列出了系统设计者在选择微处理器或者微控制器作为处理单元之前，需要考虑的因素。

表 12-9 选择系统的微处理器、微控制器或者 DSP 处理单元时需要考虑的因素和必需的特性

片上特性的因素	是否必需或者需要哪些	在所选芯片上是否可用
8 位、16 位或者 32 位 ALU	8/16/32	8/16/32
高速缓存、存储器管理单元或者 DSP 计算	是或否	是或否
高速率的密集计算	是或否	是或否
总共可达 64KB 以上的外部和内部存储器	是或否	是或否
内部 RAM	256B/512B	256B/512B
内部 ROM/EPROM/EEPROM	4KB/8KB/16KB	4KB/8KB/16KB
闪存	16KB/64KB/1MB	16KB/64KB/1MB
定时器 1、2、3	1/2/3	1/2/3
Watchdog 定时器	是或否	是或否
串行同步通信全双工或者半双工	全/半	全/半

(续表)

片上特性的因素	是否必需或者需要哪些	在所选芯片上是否可用
串行 UART	是或否	是或否
输入捕捉和输出比较	是或否	是或否
PWM	是或否	是或否
单(S)或者多通道(Mc)ADC 有(W)或者没有(WO)可编程电压基准(单(S)或者双(D)基准)	S/Mc W/WO V_{ref}S/D	S/Mc W/WO V_{ref}S/D
DMA 控制器	是或否	是或否
功耗	低或普通	低或普通

12.7.4 软硬件权衡

在硬件和软件之间需要权衡。硬件(微控制器)中的某些子系统、实时时钟、脉宽调制器、定时器和串行通信也可以通过软件实现。串行通信实时时钟和有定时器功能的微控制器的开销，比带有外部存储器和软件实现的微控制器的开销可能更大。硬件实现提供了处理速度的优势。硬件实现不仅可以加快操作速度，而且会增加功耗。硬件实现提供了如下优势：(i)减少了程序存储器；(ii)减少了芯片数量，但增加了开销；(iii)简化了设备驱动程序的编码；(iv)嵌入内部的代码，比外部的 ROM 上的代码更安全。

而软件实现提供了以下优势：(i)当有新型号的硬件可用时，软件易于更改；(ii)复杂操作的编程能力；(iii)更快的开发速度；(iv)模块性和可移植性；(v)标准软件设计、建模和 RTOS 工具的使用；(vi)使用高速微处理器，复杂函数的操作速度更快；(vii)简单系统的开销较少。

注意：

硬件(微控制器)中的某些子系统可以通过软件实现，使系统开销最少，从而优化系统性能。

12.7.5 性能建模

(1) 系统性能指标

最终开发出来的嵌入式系统的性能是衡量开发成功与否的尺度。在开发过程的每个生命周期，系统的性能测试如下：在测试之后，每个所需的功能必须显示其特征和所需的并达成共识的规范一致。

系统性能指标可以定义为，在使用最少量的存储资源、功耗和设备，最少的设计工作量，以及每个资源优化使用(例如，高 CPU 负载)的情况下，满足所需功能和规范的能力。

最好的嵌入式软硬件是那些可以在不同的性能度量之间获得平衡的软硬件。

(2) 多处理器系统性能

回顾 6.3 节，多处理器系统性能的衡量方式有：(i)任务中程序的优化分区，或者在不同处理器之间的一组指令。(ii)指令和数据在可用的处理器时间和资源内的优化调度。如果所剩的空闲时间比可用时间多，则性能开销就大。首先得到性能矩阵(参见 6.3.6 节)来计算整体的性能开销。

(3) MIPs 和 MFLOPs 作为性能指标

性能设计的衡量标准是系统花费多少时间执行期望的系统函数。处理器时钟频率和 MIPs(百万条指令/秒)和 MFLOPs(百万条浮点指令/秒)经常用作期望系统性能的设计特征。但是，它有时也不能进行正确的衡量。处理器性能设计的衡量是 Dhrystone/s(参见附录 B 了解细节)。

(4) 性能衡量：缓冲区需求、IO 性能和带宽需求

缓冲区有助于提高系统性能。存储器或者 I/O 缓冲区的需求有时可能会成为一种限制。IO 性能通过吞吐量和缓冲区的使用来衡量。客户机/服务器系统中较大的带宽需求也可能成为一种限制。

(5) 实时程序的性能

回顾 9.7 节。3 个性能尺度描述如下：(i)中断延时总和占函数执行时间的比例；(ii)CPU 负载；(iii)相对平均执行时间最长的案例执行时间。

数据通信和多媒体通信有不同的性能指标。位丢失需要重新发送位，帧或者包的丢失则是不允许的。然而，在视频和多媒体系统中，在可接受的范围内，失帧是允许的。

任务的调度时间可以使用适当的示波器或分析仪，或者使用指令计数，使用模拟器中的指令执行时间评测器来衡量。

合适的实时编程模型、划分任务和调度算法通过以下 3 种标准来反映：

(1) 系统吞吐量。系统开发过程中后面生命周期相对于前面生命周期的性能比较，或者后来系统相对之前系统的性能比较。相对性能＝吞吐量的相对增长。

(2) 每个任务的延时或者响应时间(参见 4.6 节)。吞吐量和延时两者无关。有时延时大，但是吞吐量却很小。

(3) 在某些案例中，延迟脓包(delay zitter)可以代替响应时间作为衡量性能的标准。数据帧、包或者视频帧的刷新延迟(延时)可以不同。这种差异是随机的(符合统计学上的高斯分布)，将它们称作延迟脓包。它会降低系统性能。图像脓包是不允许存在的，但是可接受阈值内的延迟刷新是允许的。

12.7.6 性能加速器

有几种方式可以提高性能(参见附录 B.3)。这些方式的示例如下：

(1) 例如，使用循环展开(循环程序段转换成直线程序流)将 CDFG 转换成 DFG，和使用查找表代替控制条件测试来决定程序流路径。

(2) 重用数组、存储器和适当的变量选择、适当的存储器分配和释放策略。

(3) 只要可以，就尽量使用堆栈代替队列，使用队列代替列表作为数据结构。

(4) 首先计算最慢周期，并检查加速的可能性。

(5) 更多以字节形式，而不是以多字节字的形式从 ROM 中读取字的代码。

12.7.7 嵌入式系统中 OS 的移植问题

当在嵌入式平台中使用操作系统时，可能会出现以下的可移植性问题。表 12-10 给出了平台相关问题，以及每个问题中对合适的 OS-硬件接口函数的需求。

表 12-10　平台相关问题和对合适的 OS-硬件接口函数的需求

平 台 相 关	对合适的 OS-硬件接口函数的需求
I/O 指令	端口指令数据类型在不同的平台上可能不同，如下所示： (i)unsigned char* (PowerPC, M68HC11/12, M68K, S390), (ii)unsigned int (ARM) (iii)unsigned long (Itanium, Alfa, SPARC) (iv)unsigned short (80x86)
中断服务例程	中断向量的定义互不相同。OS 在不同平台上支持不同的中断向量
数据类型	OS 对于数据类型应该有适当的 API。Linux 也可能需要在<asm/types.h>中声明所有数据类型，在<linux/types.h>中包含如下类型： (i)unsigned byte (也指 8 位字符) (ii)unsigned word (指无符号 16 位数，也写作 unsigned short) (iii)unsigned int (指无符号 32 位数) (iv)unsigned long (指无符号 64 位数)
接口专用数据类型	例如，网络接口卡支持 32 位无符号整数，它们都是高字节先传(big endian)格式
字节顺序	字节顺序可能依赖于处理器。可以是低字节先传(little endian)格式和高字节先传(big endian)格式。大部分高级平台支持高字节先传格式。一些处理器两种格式都支持(如 ARM)
数据对齐	(i)存储在一个地址上的两个或者 3 个字节，但是处理器以每次访问 4 个字节的方式访问该地址 (ii)“C”源文件中同样的数据结构在不同的平台上的显示方式也不同(“C”在 16 位处理器上取 16 位整数，在 32 位处理器上取 32 位整数)。编译器必须使用 OS 硬件接口函数，强制数据对齐
链表[1]	OS 可保存不同数据结构的列表，提供双链表和循环链表的标准实现。平台相关设备管理器和驱动程序必须包含对这些结构的支持
存储页大小	PAGE_SIZE 在 Linux 中是 4KB。处理器还可以支持其他页面大小
时间间隔	Linux 系统时钟每 10 毫秒滴答一次。超时函数需要在将 OS 移植到平台上时通过检查

1. 循环链表是指列表的最后一个元素不是指向 NULL 指针，而是指向列表的第一个元素。双链表是指每个元素有两个指针，一个指向下一个元素，一个指向前一个元素。

注意：

在将 RTOS 代码移植到系统中时，必须注意 I/O 指令、中断服务例程、数据类型、接口专用数据类型、字节顺序、数据对齐、链表、存储页大小和时间间隔的移植，因为它们都与具体平台相关。这些移植工作需要用到 OS 硬件接口函数。

本章小结

- 在硬件设计过程中选择合适的硬件，和软件设计过程中对硬件的可行性及性能的深入了解，对于复杂嵌入式系统的开发都非常重要。
- 系统项目管理使用软件和硬件的团队成员、人、过程、产品和项目来组织。
- 对于系统软硬件协同设计，需要制定各设计周期中的行动计划。系统需求和规范要对不同参数有详细说明。如下这些系统规范必须在设计过程开始之前准备好：(i)产品功能和任务；(ii)交付时间表；(iii)产品生命周期；(iv)系统负载；(v)人机交互(例如使用键盘和显示器)；(vi)操作环境(例如温度和湿度)；(vii)传感器；(viii)功耗需求和环境；(ix)系统开销。
- 随后进行原理设计，得到应用软件和硬件的结构和布局。UML“类图”、“用例图”、“对象图”、“顺序图”、“状态图”有助于原理设计、系统结构和布局的开发。
- 一种设计方式是在系统集成之后独立设计。这种方式的含义是：在设计系统时，软件生命周期结束，同时将该软件集成到硬件中的过程的生命周期开始。另一种设计方式是在复杂嵌入式系统的设计中采用并行协同设计。即当协同设计一个时间紧急的复杂系统时，两个周期同时进行。
- 有许多软硬件工具可以帮助我们简单地实现系统设计。用于硬件设计和系统集成的硬件工具有仿真器和内置电路仿真器。这些工具有模拟器、编辑器、编译器、汇编器、源代码设计工具、性能评测器(用于查看每个函数或者指令集消耗的时间)、存储示波器、类似听诊器的代码执行查看器、存储器和代码覆盖示波器、仿真器、ICE、示波器、逻辑探测器、逻辑分析仪、EPROM/EEPROM 应用程序代码刻录机。
- 通过使用目标系统电路或其仿真器、内置电路仿真器(ICE)、将最终代码下载到 ROM 中的设备编程器、模拟器、源代码设计工具中的基本模块，来完成系统的实现和集成。
- 使用原型开发工具和 IDE，可以在投入较少的工作量时，开发出完全模拟、通过测试和调试的复杂嵌入式系统。
- 可以使用简单的 LED 和逻辑探测器进行硬件测试。它是基于 LED 的探测器，显示逻辑状态“1”或“0”。可以用于测量端口状态，是测试事件和秒级响应时间的重要工具。还可以用于系统时钟的微调。
- 逻辑分析仪有助于从总线和端口，通过其多条输入线路收集信号，检查记录并分析许多总线事务(可达 128 或者更多)。它在显示器(屏)上显示这些记录来调试实时触发条件。它还有助于在指令执行时，顺序查找信号。
- 通过测量延迟时间和时限的超过率，了解实时编程、调度模型和算法的性能。
- 有几种衡量系统性能的方式。可以是对需求和协商过的规范的符合程度、功耗、吞吐量、IO 吞吐量、任务响应时间、时限的超过率、偶发任务的响应、存储缓冲区、带宽需求和存储器优化。
- 性能指标给出了相对于需求规范或参数的期望性能。
- 使用性能加速器改善系统性能。加速的意思是通过各种适当的方式使用同一系统，如减少一组代码的执行时间，减少任务的延时，增加吞吐量，最小化功耗和存储器使用，减少时限的超过率。还有一些方式是循环展开，查找表，重用数组和存储器，适当的

变量选择，适当的存储器分配和释放策略，只要可以，就尽量使用堆栈代替队列，使用队列代替列表作为数据结构。首先必须计算最慢周期，并检查加速该周期的可能性。

- 必须选择合适的处理器、存储器、设备和总线，使用 OS/RTOS 移植处理器敏感、存储器敏感和设备敏感指令。字节顺序和数据对齐必须依据平台来选择。

关键词及其定义

- 行动计划：开发过程的行动计划。
- 目标系统：在开发阶段使用的目标嵌入式系统，最终产品的软硬件通过该系统完成。
- ICE：内置电路仿真器，用于仿真目标系统，在一端连接处理器，另一端连接 PC。
- In-Circuit-Emulator：参见 ICE。
- 仿真器：用于仿真目标系统的一种电路。
- 汇编器：对使用助记符编辑的代码进行汇编。
- 高字节先传(Big Endian)格式：首先取得数据的最高字节的字节顺序。
- 低字节先传(Little Endian)格式：首先取得数据的最低字节的字节顺序。
- 位率测量仪：测量吞吐量的设备，以位/秒为单位表示，计算特定的时间间隔内 1 和 0 的数量。
- 刻录：使用压缩或者非压缩格式，将代码和数据的 ROM 镜像植入 EPROM、EEPROM、闪存、微控制器或者其他类似设备中的一种行为。
- 循环链：一种列表的数据结构，最后一个元素指向第一个元素，而不是像常见列表那样指向 NULL 指针。
- 双链表：一种列表的数据结构，列表中的每个元素既指向后一个元素，又指向前一个元素，而不是像常见列表那样只指向后一个元素或者 NULL 指针。
- 协同设计：在设计过程中，软件设计团队完全了解硬件的性能和特征，或者完全了解软件 CDFG 和需要实现的功能(在 SoCs 中，协同设计有不同的含义，指的是作为单个 VLSI 单元设计的软件实现电路、处理器电路和设备)。
- 交叉汇编器：一种汇编器，编写用于主机模拟或其他目的的汇编代码，然后生成用于目标处理器的汇编代码。
- 数据对齐：数据对齐的意思是位置对齐，比如，(i)存储在一个地址上的两个或者 3 个字节，但是处理器以每次访问 4 个字节的方式访问该地址；(ii)“C”源文件中同样的数据结构在不同的平台上显示方式也不同。
- 调试工具：调试嵌入式系统的硬件和软件功能的工具。
- 延迟脉包(Delay Zitter)：延迟脉包是指获取或者连续的数据集合到达时，延迟的随机差异。
- 设备编程器：用于烧制代码的设备(参见附录 G.2)。
- 反汇编器：从机器码中得到较高级别的代码，这些机器码是以前汇编得到的。
- DLL：一种库，包含了在编译时不连接，而在后面的运行过程中需要时才连接的函数。当从其他系统或者外部源中获取函数时，这样做可以节省存储空间。
- 编辑-测试-调试周期：在实现阶段，对测试时所报告的错误，对代码进行编辑、测试和调试的周期。

- 嵌入式系统项目管理：组织人员、过程、产品和项目。嵌入式系统开发项目中人员的含义是一个包括软件开发人员、硬件开发人员和系统集成工程师的团队。
- 人机交互：用户通过键盘、显示器和图形用户接口等工具和机器进行的交互。
- I/O 指令：处理机的读、写、字节操作和其他在端口上使用设备的指令。
- 平台相关：函数、ISR、设备驱动程序、OS 函数、数据类型或者数据结构的使用，取决于系统中的处理器、存储器或者设备。
- 集成开发环境：参见 IDE。
- IDE：一个集成工具，包括带编辑器的模拟器、编译器、汇编器、源代码设计工具、性能评测器(用于查看每个函数或指令集合执行所耗费的时间)、存储示波器、类似听诊器的代码执行查看器、存储器和代码覆盖示波器、仿真器、逻辑分析仪和 EPROM/EEPROM 应用程序代码刻录机。
- 网络堆栈：堆栈的属性取决于所选协议。协议可以是：RFC-1323、CIDR、IP 多播、IP、UDP、TCP、DNS 客户端、DHCP 服务器、SMTP 服务器、RIPv1 支持、RIPv2 支持、ARP、Proxy ARP、BOOTP、RLOGIN 客户端和服务器(用于 Telnet)。然后，嵌入式系统的套接字就可以连接到多协议 LAN、ATM 网络、SONET 或无线访问和智能网络。
- 逻辑探测器：基于 LED 的探测器，显示逻辑状态“1”或“0”。可以用于测量端口状态，是测试事件和秒级响应时间的重要工具。还可以用于系统时钟的微调。
- 逻辑分析仪：一种通过其多个输入线路(24 或者 48)收集端口和总线事务信号，检查记录并分析总线事务(可达 128 个或者更多)的强大工具。它将这些内容显示在显示器(屏)上，从而可以对触发条件进行实时调试。它还有助于在执行指令时，帮助顺序查找信号。
- 解释器：解释器将代码逐表达式(逐行)地翻译成可执行的机器码。
- 延时：事件之后的激活所花费时间，或者在下一个事件开始之前执行某些代码所花费的时间。
- 性能指标：测量期望的性能相对于需求规范的指标。
- 性能加速器：使用相同的系统，不同的方式，改善一组代码的执行时间，以减少延时、增加吞吐量、最小化功耗或存储器使用。性能加速器也可以是和主处理器接口的硬件单元。
- 性能尺度：使用不同尺度测量性能的指标。
- 页面：以 KB 为单位计量的存储单元，可以是从开始地址算起的一个块，其中的存储地址可以使用开始地址加偏移量来引用。
- 页面大小：存储管理器所管理的页面的大小。
- 原型设计工具：协同设计一个嵌入式系统的原型开发工具。
- PLC：执行顺序逻辑控制函数的可编程单元。
- 合适的平台：合适的硬件平台，带有合适的软件，使得可以通过最小的工作量和开销得到最好的性能。
- ROM 仿真器：在实现和测试阶段使用，以仿真目标系统中 ROM 的闪存或者 EEPROM。
- 模拟器：PC 或者主机的模拟工具，模拟所有硬件和软件功能，对于软件开发团队的测试和调试工作有很大帮助。
- RougeWave：类似 UML 的建模、设计和测试工具。

- 生命周期：软硬件开发过程中，研究系统需求、规范、原理设计、详细设计、实现和测试的各个周期。
- 示波器：带有显示屏的监视器，显示两个信号电压随时间变化的情况。还可以显示模拟信号和数字信号随时间变化的情况。
- 存储示波器：也称作数字存储示波器。在存储器中存储对应时间的信号。存储的内容可以在后面查看。可以预先设置用于记录的时间窗口。这有助于标记和测量延迟和传输时间，以及相对时间差。
- 软硬件权衡：制定合适的计划，使性能开销最少，通过软硬件子单元和软件模块实现函数和代码集合(例如,VLIWs)。
- 系统开销：软硬件开销。包括开发团队和管理工作的所有开销。
- 系统集成：嵌入式软件集成到硬件中，性能得到优化的合格产品。
- 吞吐量：单位时间内执行的进程或特定函数的数量。对于 IO 系统，它是指单位时间内输出或者读取的字节数量。
- VHDL 和 Verilog：设计和综合 VLSI 实现的语言，可以是系统的全部实现或者部分实现。

问题回顾

(1) 嵌入式系统开发过程中的总体目标是什么？它和软件开发过程有何不同？

(2) 您认为嵌入式系统独立设计是什么含义？这里的独立设计后面紧跟系统集成和嵌入式系统的并发软硬件协同设计。为每种设计策略给出 5 个示例。

(3) 设计嵌入式系统时需要遵循的行动计划是什么？

(4) 为什么设备驱动程序是存储器敏感和处理器敏感程序？

(5) 系统设计阶段选择处理器要考虑哪些因素？

(6) 什么是目标系统？目标系统和最终实现的嵌入式系统有什么区别？

(7) 如何理解目标系统的应用软件？

(8) 什么是仿真器？仿真器有哪些不同的组件？使用 ICE 的好处有哪些？

(9) 模拟器在开发阶段的用途是什么？

(10) RTOS 中就绪可用的网络堆栈如何帮助进行较快的无错设计？

(11) 中断例程的调用如何帮助测试设计结果？

(12) 什么是汇编语言程序？什么是助记符？什么是机器码？给出SPI和SCI驱动程序的代码示例。

(13) 什么是汇编器？如何使用汇编器？写出汇编系统中 ACVS 输入端口的示例代码(参见案例研究示例 11-1)。

(14) 什么是编译器、链接器、加载器和解释器？

(15) 什么是反汇编器？

(16) 什么是交叉汇编器？

(17) 什么是集成开发环境？

(18) 逻辑分析仪的时间模式是什么？逻辑分析器的状态模式又是什么？

(19) 如何理解逻辑分析仪？在开发阶段，逻辑分析仪的用途是什么？

(20) LED 电路也是一种功能强大的分析工具。为什么？

(21) 示波器的用途是什么？

(22) 如何使用位率测量仪测量一个实时系统的吞吐量？

(23) 数据如何对齐？给出以高字节先传格式存储 32 位整数以对齐输入流字节的例子。

(24) 如何解决接口专用数据类型的问题？

(25) 嵌入式系统中，什么时候使用 DLL？

(26) 为什么在嵌入式系统开发过程中，选择合适的平台很重要？

(27) 解释软硬件权衡。软件实现代替硬件实现的好处是什么？

实践练习

(28) 在嵌入式系统开发项目中的人员(people)是指谁？在示例 11-1 到示例 11-4 的系统中，如何选择他们？当开发实时视频处理系统时，团队如何变化？

(29) 列出开发一个 ACC 的规范(示例 11-3)。

(30) 列出数码相机的如下系统规范：(1)产品功能和任务；(2)交付时间表；(3)产品生命周期；(4)系统负载；(5)人机交互；(6)操作环境；(7)传感器；(8)功耗需求和环境；(9)系统开销。照相机应该可以存储 4 分钟视频或者 500 张静态图片。该系统应该包含 USB 端口、图像视频软件、单插槽定时器标准和 10 秒延迟模式。包括多种分辨率 1024 x 768、640 x 480、320 x 240 和 160 x 120 像素。

(31) 对于智能边界路由器，回答练习(30)中的各个问题。边界路由器有 10/100Mb/s 带宽、用于 LAN 的 Ethernet 接口、连接服务器的 Gbps Ethernet 接口、WAN 和帧延迟的 Internet 接口、ATM 和 SONET/SDH 上的数据包。

(32) 开发 ACVS 的软件和硬件布局(案例研究示例 11-1)。

(33) 参考案例研究示例 11-4。开发一个智能卡的软件和硬件布局，该卡存储一个人的所有医疗记录和历史。

(34) 解释如下硬件工具的用途：仿真器、内置电路仿真器和设备编程器。什么时候使用设备编程器？(参见附录 F.2)

(35) Motorola 的哪个模拟器可用于 68HC11？

(36) Motorola 的哪个仿真器可用于 68HC11？Intel 的哪个仿真器可用于 Intel 80x96 系列？使用 Web 搜索工具搜索可用于 Intel 80x86，Motorola 68K 和 TI ARM 系列处理器的 ICE。

(37) 列出可用于各种微处理器、微控制器和 DSP 的仿真器系统？

(38) 举例说明下列各项的作用：应用程序开发工具、本地开发环境、RTOS 的 API、调试功能设备模拟、网络模拟和用户接口。

(39) 举例说明下列软件工具的用法：性能评测示波器、存储示波器、程序流跟踪示波器、存储器分配使用示波器、代码作用域示波器。

(40) 硬件实现代替软件实现的优缺点是什么？

(41) 利用在 68HC16 和 683xx 中的 TPU 示例，解释软硬件权衡(参见附录 C)。

(42) 为什么当使用最少存储资源、功耗和设备，最少的设计工作量和每项资源的优化利用时，系统性能指标定义为符合需求功能和规范的能力？

(43) 练习(31)中定义的基于微处理器的嵌入式系统边界路由器的性能尺度是什么？

(44) 缓冲区如何帮助提高系统性能?

(45) I/O 为什么和平台无关? 定义一个 I/O 系统的吞吐量。

(46) 最小中断延时什么时候作为嵌入式系统性能尺度?

(47) 可以通过 3 种途径设计一个 SoC: 使用门阵列、使用标准单元以及使用 IP 和基本组件布局。列出使用每种途径设计的嵌入式系统的示例。

(48) 在嵌入式系统中使用 FPSLIC(Filed Programmable System Logic IC)的好处是什么?

(49) 什么情况下使用类似 RougeWave 这样的工具?

(50) 列出带有通用 RTOS 的原型开发工具。

附录 A　CISC 和 RISC 处理器体系结构和指令集示例

A.1　CISC 和 RISC 处理器的指令及其处理

当使用一种处理器的汇编语言进行编码时，理解指令和数据的格式以及寻址模式是很重要的。下面是对它们的详细解释。

A.1.1　指令和数据的格式

指令使用称作操作码(opcode)的位来指定操作。指令也指定了操作数(operand)。操作数规定一个操作使用的位。考虑一个运算指令 ADD z, x, y。规定的操作是加法(addition)。操作数的位来自 x 和 y。对这些位进行操作后，得到的结果位指向 z。处理器从指令中指定的寄存器或者指定的存储单元中得到 x 和 y。

(1) 考虑一个指令示例，ADD r4, r4, r6。它指定了操作码位为 ADD，指定了目标操作数寄存器为 r4，源操作数寄存器为 r4 和 r6。

(2) 在一个指令中，x 和 y 都可以是立即数(不是来自源寄存器或存储器)，可以使用下面的指令格式来说明这样的操作。考虑指令 ADD r4, r4, 15，它指定操作码位为 ADD，目标操作数寄存器为 r4，一个源操作数寄存器为 r4，一个立即数为 15。它将 r4 中的数和 15 相加，结果放到 r4。ADD 区分该指令的操作码位和上一个指令的操作码位。这个操作使 ID 对 CU 指令进行解码，并控制它的执行(参见 2.1 节)。15 前面的#标志是指该数以立即数的方式得到。

(3) 另一个指令示例 ADD r3, r1, [M1]可以由特殊的处理器支持。它指定操作数位为 ADD，目标操作数寄存器为 r3，一个源操作数为 r1，另一个源操作数为存储器地址 M1。这条指令将 r4 中的数和从地址 M1 得到的数相加，然后将结果放到 r3。ADD 区分该指令的操作码位和上一个指令的操作码位。这里方括号的意思是其中包含的是一个存储器地址。

(4) 这样一个指令示例 ADD r1, (r7), (r8)只有某些处理器才支持。它指定操作码位为 ADD，目标操作数寄存器为 r1，一个源操作数来自寄存器 r7 所指的存储器地址，另一个操作数来自 r8 所指的存储器地址。该指令将来自两个指针(所指的存储器地址)的数相加，结果放到 r4。ADD 区分该指令的操作码位和上一个指令的操作码位。括号的意思是其中包含的是一个存储器地址指针寄存器。

注意：

处理器的指令格式由操作码位和其后指定位的操作数组成。这种格式规定了可变长度或者固定长度操作码位。格式可以首先提供源操作数，然后再提供目标操作数，也可以反过来。指令格式必须依据一定的寻址模式和指令指定的地址数目。

如果在 68HC11 的指令中定义一个字的存储器地址操作数，那么高字节和低字节就分别在低位和高位存储器地址中。而在 Intel 处理器 80x86 中，一个字的高字节和低字节分别在高位和低位存储器地址中。如果在 ARM 处理器中定义指令中的一个字，则格式在复位后的开始几个时钟周期是可配置的。

A.1.2 寻址模式

(1) 前面给出的指令示例都是用于 3 地址机器的。在某些处理器中，没有必要在指令中指明所有 3 个操作数。所有 3 个、2 个或者 1 个操作数(源或目标)都可以给处理器以暗示，该处理器就分别称作 0、1 或者 2 地址机器(处理器)。这不仅提供了比较短的指令长度，而且提供了简化 CU 的好处，因而处理指令时的操作速度较快。它可能不会使程序长度减短，因为所需的指令数量在短地址机器中可能会增加。

(2) 某些处理器可能规定指令的格式为每条指令长度都相同，而其他一些处理器则可能规定指令长度可变。

(3) 回顾 1.2.5 节和 6.3 节。VLIW 和 SIMD 指令需要 3 个以上的操作数，允许指令指定最多 p 个地址的处理器(称作 p 地址机器)。

寻址模式定义为指令运行时访问操作数的一种方式。处理器以其不同类型的指令和指令操作数，支持各种寻址模式。(i)处理器可以为一个操作数支持一种寻址模式，为其他的操作数支持另外一种寻址模式；(ii)处理器电路可以支持运算和逻辑指令的一系列寻址模式，也可以支持数据传输指令的一系列不同的寻址模式(非正交指令集)；(iii)处理器也可以在其指令中支持几种寻址模式(有关 RISC 的内容请参考 2.1 节)。假设一个处理器有 8 个通用寄存器 r0~r7，并假设它是一个 3 地址机器。下面通过指令示范解释这种处理器中操作数的寻址模式。

(1) 考虑指令示例 ADD r4, r4, r6。3 个操作数的寻址模式都是寄存器寻址。

(2) 考虑指令示例 ADD r4, r4, #15。第 2 个源操作数的寻址模式是立即寻址。

(3) 考虑指令示例 ADD r3, r1, [M1]。第 2 个源操作数的寻址模式是绝对(也称扩展)寻址。指令中 16 位绝对存储器地址可以使用较少的几个位来指定。这些位和称作页地址的隐含地址相加。比如，假设绝对地址为 0xF0B1，指令指定 M1 的短地址(称作直接地址)为 0xB1。0xF000 是隐含的页地址，页地址在之前的指令中指明(不一定是前一个)。

(4) 考虑 ADD r1, (r7), (r2)。第 1 个和第 2 个源操作数的寻址模式是间接(也称变址或基址)寻址模式。

(5) 考虑 ADD r1, (r6),(r7, r2)。第 2 个操作数的寻址模式是基址变址寻址模式。r7 用作基址寄存器，r2 用作变址寄存器。第 1 个操作数称作使用基址寻址模式或者变址寻址模式得到的操作数，是基址还是变址取决于 r6 指定为基址还是变址。

(6) 考虑某种处理器支持的指令 ADD r1, (r7), (r2, #10)。寻址模式是：第 1 个源操作数是变址寻址模式，第 2 个源操作数是变址相对寻址模式。目标寄存器是 r1。加法是在 r7 所指存储器地址中的数和 r2+10 所指的存储器地址中的数之间进行。10 前面的#标志说明其取值为 10。这里的 10 就是指定相对地址的偏移，它由指令指定。该偏移不是无符号整数或者字节，而是一个两次求补得到的数(有符号数)。因此，存储器地址可以是加上 r2 或者减 r2 得到的值所指的地址。这种模式也称作基址偏移寻址模式。

(7) 考虑指令 Load r7, (r2, #10)。这种处理器支持的寻址模式是：对于目标操作数，是寄存器寻址模式，对于源操作数，是基址寻址模式。目标寄存器是 r7。操作数来自 r2+10 所指的

存储器地址。但是在 Load 操作之后，r2 保留其初始值。这种模式称作自动变址寻址模式。

(8) 考虑某种处理器支持的指令 ADD r1, (r7, r2, #10), (r2)。其寻址模式是：第 1 个源操作数的寻找模式为基址变址相对寻址，第 2 个源操作数的寻址模式是变址寻址。

(9) 考虑指令 ADD r1, (r7)+, (r2)。源操作数的寻址模式是自动变址递增寄存器寻址模式。加法操作在 r7 和 r2 所指的存储器地址中的数之间进行，加法操作之后，r7 中的数加 1，然后指向下一个存储器地址，在后续指令中使用。

(10) 考虑指令 LOAD r1, (r7), #10。第 1 操作数的寻址模式是寄存器寻址。Load 操作是指从 r7 所指的存储器地址中取值。但在该操作之后，r7 必须加 10，然后 r7 中的数指向下一个存储器地址，在后续指令中使用。这种模式称作后自动变址寻址模式。

(11) 如果操作数的寄存器刚好是堆栈指针，则寻址模式是堆栈寻址(参见 2.5.2(A))。将所指地址的字节压栈时，寄存器自动减 1，将栈中字节弹出放到指定地址时，寄存器自动加 1。

(12) 回顾 2.5 节。带有段地址寄存器的 CPU 规定了 3 类寻址模式——栈间、栈内和栈覆盖寻址模式。

A.1.3 指令集

每种型号的处理器都有不同的指令集。这是因为，为任何指令集指定的指令取数据和执行所设计的结构单元(参见图 2-1)电路不同。汇编语言编程器或者编译器使用指令集中的指令实现高级语言的命令。这些指令链接高级语言软件和用于实现的处理器结构单元。指令集一般由如下指令子集组成：

(1) 数据传送指令：有 2 个操作数。数据从源操作数传送到目的操作数。操作数可以来自寄存器或者存储器(或端口)地址。这些指令用于寄存器-寄存器、存储器-寄存器、寄存器-存储器、寄存器-端口、端口-寄存器、寄存器-堆栈(称压栈操作)或者堆栈-寄存器(称出栈操作)之间的数据传送(端口和设备一般使用相同的读写指令)。

(2) 位传送或控制指令：有 1 个操作数。操作数的位在其内部传送。示例如下：指令中字节的低四位传送(交换)到高四位。低字节可以和高字节互传或交换。位控制指令可以是，指令中的位可以置位和复位，位可以左移或者右移，位可以左循环移位或者右循环移位。移位或循环移位可以使用进位标志来存储或者参与。被控制和交换的位可以在寄存器中，或者在存储器(或端口和设备)地址中。

(3) 运算指令和逻辑指令：有 3 个操作数。每个操作有两个源操作数，结果存到目的操作数中，也可以存到标志中(类似 NOT 这样的逻辑指令只需要一个源操作数)。这些指令使用 ALU 和 FLPU(如果存在)。在 p 地址机器上的 VLIW 处理中可以有 p 个操作数。也可以有一个比较两操作数的指令。使用 ALU 在一个数和另一个数之间进行理想的减法操作之后，只有标志位受影响。也可能存在两个操作数之间的校验指令。两个数之间进行理想的逻辑与(AND)之后，也只有标志位受到影响。

(4) 程序流控制指令：这些指令改变正常的程序流。正常的程序流控制通过程序计数器或者指令指针递增到下一条指令来实现。程序流控制组指令都有一次递增操作，然后才紧接着改变(为什么是一个两步的过程？这是因为在子程序调用时，增加的值必须保存到堆栈中)。指令可以实现循环、子程序调用或者切换到其他任务。它也可以是暂停、中止、等待、复位或中断指令。

A.1.4 CISC 和 RISC 体系结构

微处理器可以借鉴 CISC(Complex Instruction Set Computer，复杂指令集计算机)的特性。CISC 指令集的指令有许多寻址模式。CISC 中的寄存器和变址寄存器(也有 80x86 中的基址寄存器和段地址寄存器)提供了数据传送、位序控制和逻辑运算的多种寻址模式(参见附录 A.1.2)。

微控制器也可以借鉴 RISC(Reduced Instruction Set Computer，精简指令集计算机)的特性。远程通信、视频和图形处理等应用领域中的微处理器大都采用 RISC 指令集。表 A-1 比较了 CISC 和 RISC 的一些特性。

表 A-1 CISC 和 RISC 的特性

CISC	RISC
提供了许多寻址模式(参见附录 A.1.2)。很多寻址模式使得汇编语言程序和处理各种数据结构的机器级指令更加容易。对于高级语言的编译器设计实现也变得更加容易。堆栈也是处理器之外的存储器地址空间。但是，在何种开销下 CISC 特性才可用？(a)复杂指令集，因为它有多种指令格式，而且操作数有许多可变寻址模式。(b)可变长度的指令。(c)执行不同指令的时钟周期的数量可变。(d)复杂 CU(指令译码、控制和排序)(参见 2.1 节)	提供很少的寻址模式。它有一个较小的指令集，用于存储、压栈、加载、出栈、存入存储器、从存储器中取值等操作。大部分指令和运算指令仅仅通过对堆栈或者寄存器的操作来实现。堆栈也是处理器内部提供的寄存器组。如果堆栈像寄存器一样也在微处理器内，那么访问速度就会很快。窗口可以存储不同子程序中变量和堆栈的值。这样的结果是，进程之间的切换将会变得很迅速(参见 4.6 节和 8.1 节)
有一个带控制存储器[1]的微编程单元，利用较少的硬件实现大规模的指令集，有些个别的指令有独立的实现电路	有一个不带控制存储器的有线编程单元来实现小规模的指令集，还有一个单独的硬件来实现每个指令
需要简单的编译器设计	需要复杂的编译器设计
由于基于微编程控制存储器的实现和比 RISC 更加频繁的外部存储器访问的需求，所以，CISC 特性提供了精确和密集的慢速计算	由于硬线的实现方法和比 CISC 更低的外部存储器访问频率，所以，RISC 特性提供了精确密集的快速计算。外部存储器访问频率较少是因为 RISC 机器有许多可用的寄存器组、寄存器堆和内部堆栈寄存器(多组寄存器存储了压栈和加载的值，以及中间结果。这大大减少了处理器期间对外部存储器的引用)

1. 指令(在指令译码器中接收到的指令)包括一组微指令和超微指令。微指令是一组控制序列，用于实现一般指令。超微指令是控制序列的子集，微指令在需要时调用。各种微指令和超微指令(控制序列)的重叠操作组成了各种一般指令。这样做可以简化电路。每组控制序列存储到控制存储器中，并使用公共电路来实现。CISC 的微编程单元有控制存储器和电路。RISC 的每个指令有独立的实现电路。

RISC 指令集中的指令，对于所有的位控制和运算逻辑指令逻辑组的操作数都只采用寄存器寻址模式。RISC 体系结构，由于在 ALU 和 FLPU 操作中只有寄存器寻址模式，因而是加载—存储(Load-Store)体系结构。在流水线体系结构(参见 2.1 节)中，如果只有寄存器寻址模式，

则指令的实现会比较快。RISC 为数据传送指令提供的寻址模式有：绝对寻址(用于加载和存储)、寄存器寻址、堆栈寻址(用于压栈和出栈)或者立即寻址。

每种处理器的指令集是惟一的。附录 A.2 给出了嵌入式计算系统中，最近流行的处理器 ARM7 的指令集的简要说明。

微处理器也可以同时结合 RISC 和 CISC 的特性(这种处理器的例子有 Intel 80960，ARM7 和 ARM9)。如果处理器有 RISC 处理核心，并且有内置的兼容单元，可以首先将 CISC 指令编译成 RISC 格式，然后使用处理器的 RISC 核心实现(支持基于复杂寻址模式的指令集，但是许多指令的内部实现类似使用 RISC 的实现方式(不带微编程单元))。

同时结合两类特性的处理器有这样几种类型的指令集。它们是附加寻址模式，包括用于数据传送指令的间接(变址)寻址模式、自动变址寻址模式和变址相对寻址模式。第二个操作数也可以使用运算和逻辑指令的立即寻址模式得到。

以上特性提供了在功能性上使用 CISC 的好处，提供了在快速编程实现和精简代码长度上使用 RISC 的好处。快速实现依赖于寄存器字对执行单元的立即可用性。由于大部分指令使用寄存器作为操作数，指令中作为操作数的寄存器不使用位来指定，因而精简了代码长度(指令中作为操作数的存储器地址和偏移量使用 8、16 或 32 位的位来指定)。

A.2 指令集示例——ARM7

ARM7 指令集是按照 Thumb®指令集定义的。

(1) 数据传送指令——下面所给的是寄存器—存储器之间数据传送的指令。存储器地址是按照用作变址、相对变址或者快速自动变址寻址模式的寄存器的内容得到的。

a. 寄存器加载一个字(LDR)。

b. 寄存器字存储一个字(STR)。

c. 设置存储器地址到寄存器中。地址是 12 位(寄存器中另一种 16 位地址设置方法是在运算操作中使用任意寄存器或者 r15)。

d. 寄存器加载一个字节(LDRB)。

e. 寄存器字节存储(STRB)。

f. 寄存器半字存储(STRH)(在 ARM 中一个字是 32 位)。

g. 寄存器加载半字时保持不变或者加上符号(LDRH 或者 LDRSH)。

以下是寄存器之间的字传送指令:

a. 移动(MOV)。

b. 求反后移动(MVR)。

加载、移动或者存储指令可以按条件来实现。例如，MOVLT r3, #10。如果前面的比较指令表明第 1 个源操作数小于第 2 个源操作数，则立即操作数 10 传送到 r3。条件有 LT(有符号数小于)、GT(有符号数大于)、LE(有符号数小于或等于)、EQ(等于)、NE(不等于)、VS(溢出)、VC(未溢出)、GE(有符号数大于或等于)、HI(无符号数高于)、LS(无符号数低于)、PL(正数，非负数)、MI(负)、CC(进位复位)、CS(进位置位)。

(2) 位传送或控制指令

a. 寄存器位逻辑左移位(LSL)。

b. 寄存器位逻辑左算术移位(ASL)。

c. 寄存器位逻辑右移(LSR)。

d. 寄存器位逻辑右算术移位(ASR)。

e. 寄存器位循环右移(ROR)。

f. 寄存器位带进位的循环右移，也称作循环扩展(RRX)。

(3) 运算和逻辑指令

以下是用于算术操作的指令：每条指令使用 3 个来自寄存器的操作数。但是，在加法和减法操作中，一个源操作数可以使用立即寻址。

a. 两个字无进位相加，结果在第 3 个操作数中(ADD)。

b. 两个字带进位相加，结果在第 3 个操作数中(ADC)。

c. 两个字无进位相减，结果在第 3 个操作数中(SUB)(进位用作借位)。

d. 两个字带进位相减，结果在第 3 个操作数中(SBC)。

e. 两个字无进位反转(第 2 个和第 1 个源操作数反转)相减，结果在第 3 个操作数中(RSB)(进位用作借位)。

f. 两个字带进位反转相减，结果在第 3 个操作数中(RSC)。

g. 两个不同的寄存器相乘，结果在目标寄存器中(MUL)。

h. 两个源寄存器相乘，结果和第 3 个源寄存器相加，然后把新的结果放到目标寄存器(MLA)(有 4 个操作数寄存器)。

以下是用于逻辑操作的指令：

a. 两个字按位或(OR)，结果在第 3 个操作数中(ORR)。

b. 两个字按位与(AND)，结果在第 3 个操作数中(AND)。

c. 两个字按位异或(OR)，结果在第 3 个操作数中(EOR)。

d. 清位(BIC)(一个源操作数用于位，第 2 个源操作数用于屏蔽，结果在第 3 个操作数中)。

运算和逻辑指令可以按条件来实现。例如 SUBGE r1, r3, r5。如果在前面出现 GE 条件(在比较两个有符号数时 N 和 V 状态位相等)，则 r3 中的操作数被减去 r5 中的操作数。条件的类型和前面所提到的一样，这些条件都是比较或验证的结果。

以下是比较和验证操作的指令。结果放到 CPSR 中，它存储 4 个条件位 N、V、C 和 Z。

a. 两个字按位检验(TST)。

b. 两个字按位求反检验(TEQ)。

c. 比较两个字，结果放到 CPSR 条件位(CMP)。

d. 比较两个负字，结果放到 CPSR 条件位(CMN)。

(4) 程序流控制指令：以下是用于分支操作的指令。分支指令也可以按条件实现。转到与 PC 字 r15 (B)的相对值为“B #1A8”的地址的分支，意思是 PC 加 1A8，以改变程序流程。“BGE #100”意思是如果在和 0 比较时结果为 GE，则 PC 加 100。对于处理器状态标志的不同条件，指令是相似的。

A.3 ARM 处理器的汇编语言程序示例

示例 A-1

回顾 3 个数的加法问题，x，y 和 z(分别等于 127，29 和 40)3 个数相加，结果存放到存储器地址 a 中(a = x + y + z)。使用前面的指令集，得到的汇编语言程序如下：

(1) BEGIN: MOV r2, #0x007F ;将 127 送到处理器寄存器 r2 中。

(2) MOV r3, #0x001D ; 将 29 送到处理器寄存器 r3 中。

(3) MOV r4, #0x0028 ; 将 40 送到处理器寄存器 r4 中。

(4) MOV r1, #x000 ; 将 0 送到处理器寄存器 r1 中。

(5) ADD r1, r1, r4 ; 寄存器 r4 的数和 r1 相加，结果放到 r1。

(6) ADC r1, r1, r3 ; 寄存器 r3 的数和 r1 相加，结果连同加法的进位(如果有)放到 r1。

(7) ADC r1, r2 ; 寄存器 r2 的数和 r1 相加，结果连同加法的进位(如果有)放到 A。

(8) ADR r5, 0x800; 将存储器地址 0x800 放到 r5 中。

(9) STR [r5], r1;将 r1 的内容存放到 r5 所指的地址中。

关键词及其定义

- 绝对寻址模式(Absolute Addressing Mode)：在指令中定义所有地址位的寻址模式。
- 自动索引：指令执行之后，索引寄存器中的内容自动改变。
- 指令集：处理器中可执行指令的限定集合。
- 基址：数据结构中第一个元素开始的地址。
- CISC(Complex Instruction Set Computer)：复杂指令集计算机(处理器)，在这种指令集中，有大量用来进行算术、逻辑和其他操作的寻址模式。
- 直接地址：在指令中可直接使用的地址。通常是存储器中一个页的地址。
- 指令格式：表达一条指令的格式。
- 程序流指令：在程序执行时，程序计数器和指令指针的改变与正常情况不同的指令。
- RSCI(Reduced Instruction Set Computer)：精简指令集计算机(处理器)，在这种指令集中，寻址模式非常少——如 load、store、push 和 pop，大多数的算术、逻辑和其他指令都是零地址指令。
- 零地址指令：地址对处理器而言是隐含的指令，通常是处理器内部的堆栈，或与程序模块相关的寄存器集合。
- 带 CISC 功能的 RISC：RISC 实现的处理器，而用户的编程方式与 CISC 相似。
- Thumb®指令集：每一条指令都是在 32 位处理器上运行的 16 位指令。这样减少了代码的密度，因此使用 16 位指令集时，系统开销只有 8 或 16 位，却能够达到 32 位的性能。这类指令通常用于 ARM 处理器中。

附录 B　嵌入式系统高性能处理器

回顾 1.2.2 节和 2.1 节。高计算性能应用中的复杂嵌入式系统需要资源、功耗、高速缓存和存储器的优化使用。下面是衡量处理器性能的标准：

(1) (a)高 MIPS，(b)高 MFLOPS 和(c)基于 Dhrystone 测试程序的高 MIPS。

(2) 处理器中优化的编译器单元性能。

上面这些性能尺度是最新设计的处理器提供的。高性能处理器结合了功能和资源、功耗、高速缓存和存储器的优化使用。它还使得在每秒 10 亿次操作的嵌入式系统中，高性能处理器 IC 和核心的使用随着 VLSI 技术的高速发展而成为可能。

注意：

测试程序 Dhrystone 是 Reinhold P. Weicher 于 1984 年开发的。它测量处理器处理整数和字符串(字符)数据的性能，使用可用于 C、Pascal 和 Java 的测试程序。它测试 CPU，而不是 IO 或者 OS 调用的性能。每秒的 Dhrystone 就是用于测量程序每秒可运行多少次的尺度。1 MIPS = 1757 Dhrystone/s(为什么？因为执行 1 MIPS 时，VAX11/780 可运行 1757 次 Dhrystone 测试程序(访问站点 http://www.webopedia.com/TERM/D/Dhrystone.html))。

有一个国际性的 EDN 嵌入式测试协会(EDN Embedded Benchmark Consortium，EEMBC)(EDN 是出版国际杂志 EDN 的组织，该杂志致力于向客户提供嵌入式系统方面的信息。访问站点 http://www.e-insite.net/edmag/)。EEMBC 在嵌入式系统的 5 个不同的的应用领域提出了 5 个测试程序集：(a)远程通信；(b)消费电子产品；(c)自动工业电子产品；(d)办公自动化。这些测试程序集也用来测量和比较嵌入式系统处理器的性能。

B.1　ARM 处理器示例

站点 http://www.arm.com 上有关于 ARM 的详细资料。以下是 ARM 处理器的几个显著特点：

(1) ARMv4T(4 Thumb 版)微体系结构对于 ARM7、ARM9、ARM10 和 ARM11 是通用的。Thumb 是一个工业标准。它是 16 位的指令集，可以使用 8/16 位的系统开销得到 32 位的性能。这节省了高达 35%的存储空间，等价于 32 位的代码，保持了 32 位系统的所有优点(比如对完全的 32 位地址空间的访问)。在 Thumb 和一般 ARM 状态之间的移动开销为零。两种状态在例程上是兼容的。代码设计者对于性能和代码长度的优化有完全的控制权。

(2) v4T 增强的性能表现在 ARMv5TE 体系结构(1999)中。它带有 ARM DSP 扩展指令集，将指令集在音频 DSP 应用方面的速度提高了 70%(某些应用需要微控制器数据处理功能，以及单处理器代替微处理器系统的 DSP 功能)。

(3) v5TE 的增强性能表现在 ARMv5TEJ 体系结构(2000)中。它结合了用于 Java 的 Jazelle Java 执行加速器技术。这提供了比基于软件的 Java 虚拟机(Java Virtual Machine，JVM)高许多的性能(Java 执行性能的 8 倍)。在功耗上比非 Java 加速核心减少了 80%。这种功能由平台开发者实现。Java 代码和 OS 应用程序可以运行在一个处理器上。

(4) v5TEJ 的增强性能表现在 ARM11 微体系结构中使用的 ARMv6 体系结构(首先 2002 年实现)中。它有一个扩展 SIMD(参见 6.3 节)，对包括视频和音频 CODEC 的应用进行了优化。SIMD 执行性能提高了 4 倍。

(5) ARM9E 和 ARM10 系列使用向量浮点(Vector Floating Point，VFP)ARM 协处理器，增加了完整的浮点操作数。VFP 也提供了在使用如 MatLab®这样的工具时，对 SoC 设计的快速开发。其应用领域有图像处理(缩放)、2D 和 3D 转换、字体生成和数字滤波。

(6) ARM 使用智能能源管理(Intelligent Energy Manager，IEM)技术，实现了优化平衡处理器负载和能源消耗的高级算法，使系统响应达到最大化。IEM 同操作系统和移动 OS 一起工作。运行在移动电话上的应用程序，动态调节所要求的 CPU 性能级别。它使用了标准程序设计模型。

(7) ARM 处理器使用 AHB(AMBA Advanced High Performance bus，AMBA 改进的高性能总线)。AMBA 为片上互连确定了开发源代码规范，作为一个框架，为 SoC 设计和 IP 库的开发提供服务。所有新的 ARM 核都支持 AHB。它提供了高性能和完全同步的底板。ARM926EJ-S 和 ARM10 系列的所有成员中的多层 AHB 体现了明显的改进。它减少了延时，增加了可用于多控制系统的带宽。

(8) ARM 代码向前兼容其高级版本。例如，ARM7 代码向前兼容 ARM9、ARM9E 和 ARM10 处理器，以及 Intel XScale 微体系结构。

(9) ARM 的调试和跟踪工具，以全速的内核速度，快速调试实时软件，跟踪指令执行和相关程序数据。

(10) 对于引导 EDA(Electronic Design Automation，电子设计自动化)环境(如 CADENCE EDA 环境)的开发工具和模拟模型的选择范围很广，对 SoC 设计的调试支持很好。

表 B-1 给出了高性能 ARM 系列处理器的特性和及其比较。

表 B-1 ARM 系列处理器

特　性	ARM7™ Thumb®系列	ARM9™ Thumb®系列	ARM11
系列成员示例	(a)ARM7TDMI®(整数内核) (b)ARM7TDMI-S™(ARM7TDMI 的同步版本) (c)ARM7EJ-S™(DSP 和 Jazelle 技术的内核)和 ARM720T™(高速缓存处理器宏单元[2]，8K 高速缓存内核，带有支持操作系统[1]存储器管理单元(MMU))	(a)ARM920T(双 16k 高速缓存，带有支持多操作系统[1]的 MMU) (b)ARM922T(用于应用程序的双 8k 高速缓存支持多操作系统[1]) (c)ARM940T™(用于运行 RTOS 的嵌入式控制应用程序的双 4k 高速缓存[2])	带有 ARMv6 指令集体系结构的系列，包括用于代码密度的 Thumb®扩展，用于加速的 Jazelle™ 技术，ARM DSP 扩展和 SIMD 媒体处理扩展。带有支持操作系统和掌上 OS 的 MMU

(续表)

特　性	ARM7TM Thumb®系列	ARM9TM Thumb®系列	ARM11
带有ARM®和 Thumb®指令集的内核	32 位 RISC 内核	32 位 RISC 处理器内核超标量 5 级整数流水线。8 入口写缓冲器。它防止在外部存储器进行写操作时阻塞处理器	32 位 RISC 处理器内核，带有 8 级整数流水线、静态和动态分支预测、独立的 load-store 和算术流水线，从而使指令吞吐量最大化
应用领域	价格和功耗敏感的个人消费领域，如个人音频设备(MP3、WMA、AAC 播放机)，入门级移动电话、两路寻呼机、静态数码相机、PDA	机顶盒、住宅网关、游戏控制台、MP3、MPEG4 摄像头视频电话、便携式通话机、PDA、下一代手持产品、数字消费商品、图像商品、桌面打印机、静态图片照相机、数码视频照相机、自动遥测装置和信息维持系统	充电电池和高密度嵌入式应用。面向下一代无线和家电应用的嵌入式 SoC。致力于嵌入式应用、高级操作系统和诸如音频和视频 CODEC 的多媒体的需求，包括 2.5G 和 3.0G 的移动电话听筒设备、PDA 和多媒体无线设备、诸如图像和数码相机的家庭应用，包括 IP 通话和宽带调制器的基础设施
性能	在通常 0.13μm 工艺下使用 Dhrystone 2.1 测试为 130 MIPS	达到 1.1MIPS/MHz。在通常 0.13μm 工艺下为 300 MIPS Dhrystone2.1	把 Dhrystone MIPS 400~1200 作为性能范围目标
代码密度	高代码密度(与 16 位微控制器相比)	高代码密度	高代码密度
硅芯片尺寸	小芯片，可移植到 0.25μm、0.18μm 和 0.13μm 各型号	在 ARM940T 中芯片面积为 4.2mm^2。可移植到最新的 0.18μm、0.15μm 和 0.13μm 硅片工艺。在 0.18μm 的 ARM940T 上频率为 185MHz	0.13μm 制造工艺在最坏的情况下可以提供 350~500+MHz 的频率，在下一代制造工艺中则可能超过 1GHz

(续表)

特　性	ARM7TM Thumb®系列	ARM9TM Thumb®系列	ARM11
存储器耦合(参见6.3节)	非紧耦合	非紧耦合	
功耗性能	超低功耗	超低功耗。在0.18μm的普通硅片制造工艺下，940T功耗为0.8mW/MHz。最坏的情况是：1.62V，125C，慢速耗损硅。典型情况为：1.8V，25C，额定硅	优化的功耗效率，乱序完成的单流出操作，使用尽量少的门数量，在0.13μm的铸造工艺上，功耗少于0.4mW/MHz
总线接口	AHB	单32位AMBA总线接口	无

1. 含义是支持Window CE、Palm OS、Symbian OS、Linux和其他OS/RTOS。掌上OS支持ARM920T和ARM922T处理器。ARM940T有存储器保护单元(MPU)和包含VxWorks的实时操作系统的支持范围。

2. 集成指令和数据高速缓存。

3. 关于ARM体系结构的具体资料，请参考体系结构指令集和程序设计模型，如ARMv5TE体系结构、ARMv5TEJ体系结构和ARM11中的ARMv6。关于ARM微体系结构的具体资料，请参考诸如ARM9™系列内核和ARM10系列内核的体系结构的实现。如ARM926EJ-S™内核和ARM1020E™内核就是基于这些早期微体系结构的CPU产品。

B.2　高性能处理器示例

Intel XScale和StrongARM SA-110、Motorola Power PC 860、IBM PowerPC 750X、TI OPMAP和MIPS R5000是其他的32位和32/64位高性能处理器的示例。它们也用在许多嵌入式系统的应用中。表B-2给出了高性能IBM PowerPC 750X的特性 (http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699300 58A78D.htm)。

表B-2　IBM PowerPC 750™系列处理器

特　性	IBM PowerPC 750™ Thumb®系列
系列成员实例	(a)IBM PowerPC 750™ PID 8t；(b)IBM PowerPC 750™ PID 8p
核心	32位RISC内核、64位数据总线、32位地址总线、带有铜电平的CMOS 7S处理器技术、动态电源管理、集成温度管理的辅助单元、高速L2缓存、60x总线接口、支持最多233MHz的SRAM
高速缓存	1MB 两路联合L2高速缓存控制器，用于1MB 、L2、128字节的段(当L2为256kB时，段为64字节)

(续表)

特　性	IBM PowerPC 750™ Thumb®系列
应用领域举例	入口级移动设备
性能	在使用 1MB L2 高速缓存的 500MHz 操作下，性能为 33.9 SPECint95 和 14.6 SPECfp95
ALU 和 FLPU	一个周期的硬连线乘和除定点单元，带有优化的单处理乘/加的 64 位浮点单元
硅芯片尺寸	小芯片，可移植到 0.26μm 和 5 金属级版本(PID 8t)以及 0.20μm 和 6 金属级版本(PID 8p)
功耗性能	对于 PID 8t 是 2.5V~2.75V 核心和 3.3V 的 IO，对于 PID 8p 是 2.0V~2.1V 的核心和 1.8V、2.5V、3.3V 的 IO。在 PID 8p 频率为 500Hz 的情况下，功耗为 0.6W
总线接口	奇偶校验、快速复位、功能强大诊断测试接口，使用 CCP(Common Chip Processor，通用芯片处理器)和 IEEE 1149.4 JTAG 接口

一些处理器致力于提供一些特殊的性能。如最近的 X10 系列网络处理器，它提供了用于 IPv6 的 10Gbps 的端口性能(宽带 Internet)。

B.3 加速器

加速器的作用是加速代码执行。它可以是 ASIC、IP 核或 FPGA。加速器可以包含总线接口单元、DMA、读写单元、寄存器和加速器内核。加速器使用程序设计模型来加速，不像处理器那样带有针对特定任务的指令。处理器控制寄存器通过总线和加速器连接和交互，因而处理器可以控制加速器，以获得更高的性能。

例如，来自 Nazonin Communication 的最新 JA108 就是一个 Java 加速器，它将 Java 代码的运行速度提高了 15~60 倍。另外一个例子是视频加速器，其作用是加速视频处理器任务的执行。

关键词及其定义

- 加速器：ASIC、IP 核或 FPGA，它可以加速代码执行，也可以包含总线接口单元、DMA、读写单元，以及带有其内核的寄存器。
- Java 加速器：获得比 JVM 速度更快的 Java 代码执行速度的加速器。
- JVM：可以在指定的系统上运行编译过的 Java 程序字节码的机器代码。
- 视频加速器：用于视频输出的加速器。
- AMBA：为片上互连确定开放源代码规范，作为框架为 SoC 设计和 IP 库开发提供服务。
- AHB：ARM 处理器使用的高性能 AMBA 版本。
- ARM：高性能精简代码密度处理器系列 ARM7、ARM9、ARM10 和 ARM11，作为芯片或者 ASIC 和 SoC 中的内核，使用在嵌入式系统中。
- Dhrystone：测试程序，测量处理整数和字符串(字符)的处理器的性能。使用的测试程序可用于 C、Pascal 或 Java 等环境。它测试的是 CPU 的性能，而不是 IO 或者 OS 调用的性能。1MIPS=1757Dhrystone/s。

- EDA：强大的电子设计自动化工具。
- EEMBC：EDN 嵌入式测试协会(EDN Embedded Benchmark Consortium)。
- 性能测试：衡量系统性能的度量方式。
- 微体系结构：当处理器体系结构特指体系结构的指令集和程序设计模型时，微体系结构就特指这些体系结构的实现。处理器可以使用 RISC 微体系结构实现 CISC 体系结构。

附录 C　嵌入式系统 8/16/32 位微处理器及其体系结构概述

回顾 1.2.3 节，对于许多应用领域，如电气自动化和计算机网络等嵌入式系统都包括以下芯片或核心功能部件：

(1) 具备必需的 8 位、16 位或 32 位 MIPS 的处理器。

(2) 设备、RAM 和 ROM、定时器、中断控制器、PWM、ADC、串行和并行 I/O。

微控制器中提供了这些功能部件。微控制器应该具备微计算机的能力，使得系统可以在带有一个或者几个 IC 的情况下工作。

C.1　Intel、Motorola 和 PIC 系列微控制器的体系结构概述

回顾微控制器中的功能电路，如图 1-2 所示。图中给出了指定微控制器系列的特定型号中的应用单元。表 2-2 对 Intel 8051 和 Motorola 68HC11 微控制器的处理器的性能进行了比较。表 C-1 对 Motorola 68HC12A4 和 PIC 系列 16F84 微控制器的处理器性能进行了比较。记住，在同一系列的不同成员中，相同的外设有不同的形式。

表 C-1　微控制器中的处理器特定特征

性　能	Motorola 68HC12A4	PIC 16F84
以毫秒为单位的处理器指令周期	0.125	0.2
以位为单位的内部总线宽度	16	14
CISC 或 RISC 体系结构	CISC	CISC
带复位值的程序计数器位	22[(0xFFFE)]	13
处理器定义的带复位值的堆栈指针位	16	13
超标量体系结构	无	无
片上 RAM 和/或寄存器堆字节	1024	68 x 8b + 64 x 8b EEPROM
指令缓存	无	无
数据缓存	无	无
内部程序存储器 EPROM/EEPROM	4KB	1K x 14b
以字节为单位的外部程序存储器容量(如果是独立的)	非独立	-
数据存储器容量	4MB + 128 KB	–
外部中断	24	1

(续表)

性　能	Motorola 68HC12A4	PIC 16F84
位操作指令	有	有
浮点处理器	无	无
中断控制器	有	无
DMA 控制器通道	无	无
片上 MMU	无	无

Motorola 微控制器芯片 68HC11 和 68HC12 具有兼容性。不仅汇编代码具有兼容性，而且 68HC11 的可执行文件可以在 68HC12 上运行，68HC11 的汇编代码可以在 68HC12 上运行。68HC12 有较大的存储器地址空间，提供了 4MB+128KB 地址空间的芯片存储器映射，而在 68HC11 中为 64KB。68HC12 以较高的时钟速度进行操作(内部时钟为 8MHz， HC11 中为 2MHz)，带有较大的 RAM(1024B)和 EEPROM(4KB，HC11 中为 2KB)，更多端口和 8 通道定时器(8 个输入捕捉/输出比较寄存器)。68HC12 有 16 位的脉冲累加器，68HC11 中为 8 位。它有一种变异的变址寻址方法，能使汇编程序代码更短。一部分地址可以编码成指令字节，这有助于在程序中使用较少的代码和硬件。这些芯片的基本框架是相同的。使用 C 编译器时，代码比 HC11 缩短了 30%。另外，它有更多的 8 位端口可供使用——总共 12 个(包括 1 个 ADC，3 个用于 24 键唤醒，1 个用于中断，1 个用于串行 IO)，而在 68HC11 中总共只有 5 个。更多的定时器有助于减少对机器人引擎控制的基于实时时钟的软件定时器的需求。68HC11 中的 24 个唤醒线，它对于使用小键盘输入非常有益。68HC12 中的端口比 HC11 中更多，提供了在机器人应用领域的其他优势。机器人有 4 个或者更多的自由度，需要 4 个或者更多的用于手腕、颈部、肩膀、腰部和手掌活动的引擎。更多的端口或者端口的多路复用也是很有必要的。

PICF84 微控制器没有外部地址和数据总线。它适合小型单芯片系统，例如小键盘和显示控制系统(关于 PIC 微控制器的细节，请参考 Tim Wilmshurst 编著的 *An Introduction to the Design of Small Scale Embedded Systems—— with examples from PIC, 8051, and 68HC05/08 Microcontrollers* 一书)。

表 C-2 选择了几种流行的微控制器，对它们常用的性能进行了比较。最近的微控制器推出了一些更高级的设备，例如集成了串行 EEPROM、CAN 总线接口、I^2C 总线接口、I/O 扩展器、USB1.1 或 2.0 接口以及 PCI-X 接口。

表 C-2　微控制器的具体特性

性　能	Intel 8051 和 Intel 8751	Intel 80196	Motorola M68HC12A4	Motorola M68HC11E2	PIC 16C76
8 管脚的输入输出端口	4	4	12	5	2+6 位
定时器	2	2	8	1	2
串行输入输出 SYN. 和 ASYN.	有	有	两个 SPI 和 SCI	SPI 和 SCI	SSP+USART

(续表)

性　能	Intel 8051 和 Intel 8751	Intel 80196	Motorola M68HC12A4	Motorola M68HC11E2	PIC 16C76
事件或信号的实时检测(捕捉和比较事件发生的时间)	无	有	有	有	有
对 DAC 的脉宽调制	无	有	有(16)	有(8)	有
模数转换(位)	无	有	有(8)8 通道	有(8)4 通道	有(8)
调制解调	无	无	无	无	无
数字信号处理指令	无	无	无	无	无
非线性控制器指令	无	无	无	无	无
断电模式	无	有	有	有	无
空闲模式	无	有	有	有	无
Watchdog 定时器	无	有	有	有	无

1. 在 8052 型号中有 3 个定时器。

2. 68HC11 系列中有主定时器。其他的是辅助定时器。

3. 在高级型号的 8051 中也存在实时时间检测、ADC、PWM、Watchdog 定时器和输入捕捉设施。最新的型号是 80251。

4. IC 型号的数据手册必须是对微控制器系列型号经过商议的准确说明。这是因为生产商可能会不断修订或增加某些特性，以及提供其他的系列型号。

5. PIC 16F876 是 PICF76 的闪存型号。

6. 68HC12 中存在 24 键唤醒线，和 68HC11 相比，它的 3 个辅助端口有中断功能。

C.2　Motorola 系列 CISC 和 RISC 的新一代微控制器

表 C-3 给出了 Motorola 系列微控制器的比较。

表 C-3　最近的 Motorola 系列高性能微控制器的特性

特　性	M68HC16 系列	M683XX 系列	MCORE MMC2001
CPU 特性	16 位增强 68HC11	32 位 M68000 操作码兼容性	32 位 load/storeRISC 体系结构，超低功耗
4 级指令流水线、带有 16 入口专用备选寄存器堆的快速中断支持、快速上下文切换矢量和自动矢量中断支持、复位单元	无	无	有

(续表)

特　性	M68HC16 系列	M683XX 系列	MCORE MMC2001
增加了虚拟闪存、向量寄存器、循环模式、寻址模式和指令	无	可选的 M68010 和 20 个增加项	无，每个指令都是 16 位，并且代码密度最佳[3]
表查询和内插(TABL)指令	无	有	无
数据＋程序存储器	1MB + 1MB	程序 16MB	无[2]
DSP 函数	有	无	无[2]
通用定时器——两个 16 位自激计数器，带有可编程的预标量，16 个通道，每个通道都和一个 I/O 管脚相关、一个 16 位捕捉寄存器、一个 16 位比较寄存器和一个 16 位大于或等于比较器	有，9 级预标量，16 位 FRC[1]	有	无
可配置定时器模块	有	有	时间定时器、周期中断定时器的时间
带有微程序控制单元、控制存储器和 20 多个不同定时函数的定时器处理单元	TPU 或 TPU2	有，TPU/TPU2	无
CAN	有，CAN 版本 2.0 A/B	有	无
多通道通信接口	1 SPI + 2 SCI	有	无
排队串行模块	有	有	无
系统积分器	12 个可选的可编程芯片	有	无
闪存、EEPROM 或者闪存 EEPROM	闪存 EEPROM	有	256KB 可屏蔽 ROM[2]
SRAM	1 到 1k 字节的块可编程性	有	32KB[2]
动态总线尺寸恢复	8 位或者 16 位	8 位或者 16 位	无[2]
高级语言支持	有	有	无[2]
时钟频率	16MHz、20MHz、25MHz	16MHz、20MHz、25MHz	对于 31MIPS 性能为 33MHz[2]
操作电压	5V 或者 2.7V~3.6V	5V 或者 2.7V~3.6V	1.8V 和 IO 3.6V[2]
串行输入输出 SYN.和 ASYN.	有	有	2 UART[2]

(续表)

特　性	M68HC16 系列	M683XX 系列	MCORE MMC2001
事件和信号的实时检测(捕捉和比较事件发生的时间)	有	有	无
对 DAC ADC 的脉宽调制	有 8 位或 10 位可编程，带有备用电压基准和可编程样本	无	6 通道[5] 无
排队 ADC	有，16 通道	有	无
调制解调	无	无	无
非线性控制器指令	无	无	无
断电模式	有	也是低功耗阻塞(LPSTOP)指令	
空闲模式	有	有	有
Watchdog 定时器	有	有	有

1. 是指 16 位自激计数器。
2. 第一个 MCORE 系列成员 MMC 2001 的数据。
3. 例如 ARM Thumb 指令集。

关键词及其定义

- 高级语言支持：具有处理器结构的支持单元，它方便了使用 C 或者其他高级语言进行的程序设计，使它们通过内部编译，像机器码一样运行。
- 排队 ADC：对以 FIFO 方式放入队列的通道执行 ADC 操作的 ADC。
- 调制和解调单元：调制解调器。
- 自激计数器(FRC)：一直运行从不停止的计数器，不可以复位。它有许多定时器方面的应用。它在处理器时钟频率预标量之后获得输入(如果预标量系数配置为 64，则 FRC 输入对于 8MHz 的处理器时钟是 0.125 毫秒(1/8MHz))。
- 定时器处理单元(TPU)：定时器有大量中断，相应的 ISR 和函数。例如定时器中断可以是输入捕捉、输出比较、实时时钟、计数器、定时器溢出和软件定时器。通过单独处理 ISR，TPU 对创建非常有帮助。
- 输入捕捉：当有输入事件时捕捉 FRC 读，并为相应 ISR 的调用产生中断。
- 输出比较：将 FRC 读取的值和比较寄存器中的预定值进行比较。它用作类似报警的函数。如果两个值相等，则可能产生中断，并启动输出事件。
- Watchdog 定时器：预先设置的定时器。其溢出表明处理过程在某个地方阻塞了，因而处理器需要复位并重启。

附录 D　嵌入式数字信号处理器

回顾 1.2.5 节，对于视频录像机和移动电话等应用领域的嵌入式系统，应该具备以下处理器性能，以得到资源、功耗和存储器的优化使用：

(1) 高 MIPS 和/或高 MFLOPS

(2) 高 MAC

数字信号处理器(DSP)提供了这些性能。数字信号处理器应该具备发声和视频处理能力，嵌入式系统中的 DSP-IC 和 DSP 内核则应该具有实时发声和视频处理能力。

D.1　数字信号处理器的体系结构

可以考虑 TMSC64x™ DSP 系列的 DSP 示例，以深入理解 DSP 的体系结构。

表 D-1 给出了 TMSC64x™ DSP 系列的主要结构单元及其功能。图 D-1 通过处理器结构示意图介绍了 25 个结构单元的内部连接情况。表 D-2 列出了处理器 TMS320C64x64™ VelociTI ™ VLIW 扩展体系结构中的辅助结构单元和功能。

表 D-1　DSP 内核中处理器的结构单元及功能

内核结构单元	功　能
基本单元	MDR、内部总线、数据总线、地址总线、控制总线、总线接口单元、指令取寄存器、指令译码器、控制单元、指令缓存、数据缓存、多级流水线处理、为了获得高于 1 指令/时钟周期的处理器速度的多流水线超标量处理以及同表 2-1 相似的程序计数器
指令分派寄存器	将指令分派给适当的处理单元
控制寄存器	同处理器的控制单元相关的控制寄存器
仿真单元	实现仿真
寄存器堆 A	处理数据通道 1 中的指令时使用的片上寄存器组。它们命名为 A0…A15 和 A16…A31。回顾一下寄存器堆的概念，它是指和 ALU 或 FLPU 等单元相关联的寄存器集合
寄存器堆 B	处理数据通道 2 中的指令时使用的片上寄存器组。它们命名为 B0…B15 和 B16…B31
预取单元	CPU 在每个周期取出 8 条 32 位的类似于 RISC 的指令
处理单元	包含两个乘法器和 6 个算术单元，高度互不相关。还包含编译器、汇编优化器以及执行资源
算术逻辑子单元	根据当前 IR 中的指令，执行算术或逻辑指令的子单元
辅助逻辑子单元	减法操作时使用的子单元(在加法之前查找 2 的补码，然后相加，从而实现减法)

(续表)

内核结构单元	功　能
乘法器子单元	实现乘法操作
浮点处理子单元	C67x™ 中的这个子单元是为了便于浮点处理，而从 ALU 中分离出来的(参见附录 D.3)
汇编优化器	优化汇编代码
C 编译器	提供高效的编译能力

在许多功能单元中可以利用具有频繁使用的指令的正交指令。

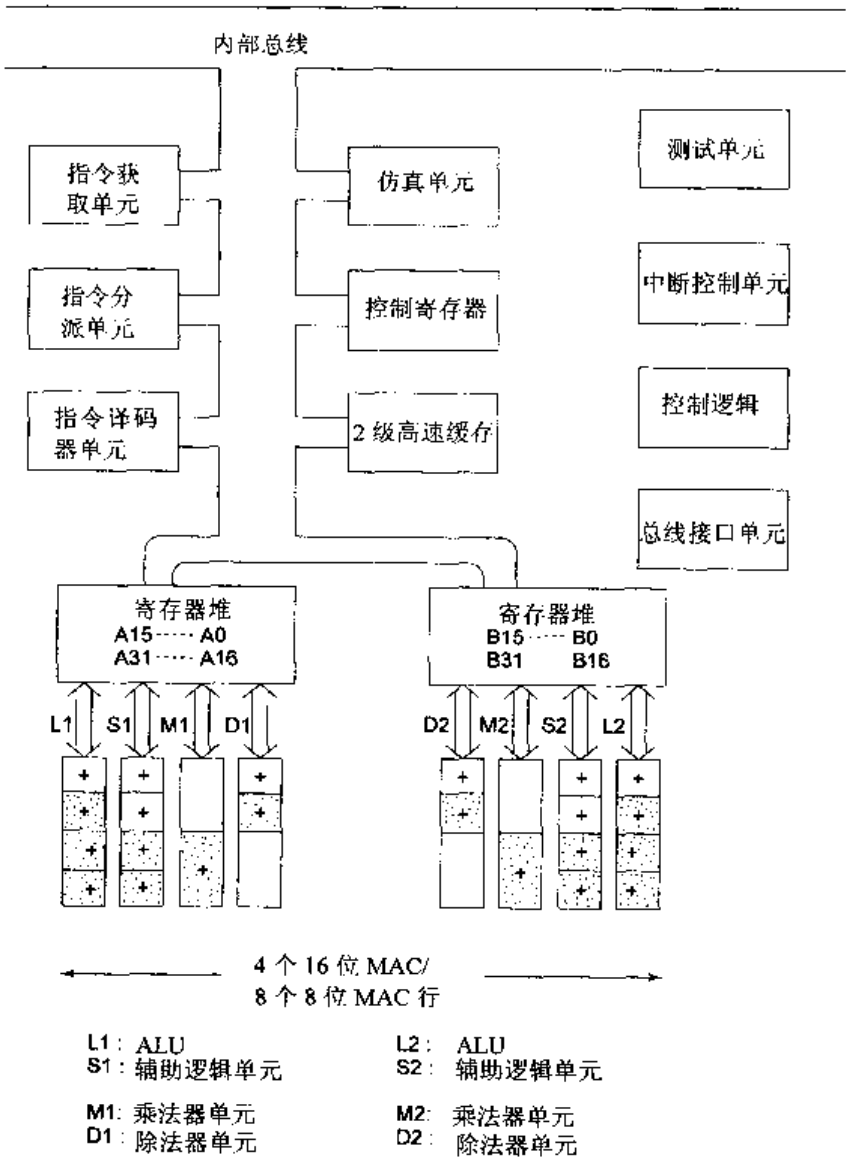


图 D-1　TMS320C64x 中的内核和特殊结构

表 D-2 TMS320C64x64™ VelociTI ™ VLIW 扩展体系结构中处理器的辅助功能单元及其功能

内核结构单元	功 能
压缩数据处理	8 位或 16 位数据作为 32 位数据压缩和处理
并行执行 MAC 单元	4 线 16 位 MAC/8 线 8 位 MAC(参见 2.1 节和附录 D.2)
特殊指令	宽带设施和图像处理 VLIW
2 级缓存	提高每个取周期的性能
指令压缩单元	VLIW 并行执行时的指令压缩

D.2 DSP 处理器和传统处理器的比较

正弦信号是最简单的传输模拟信号。失真正弦信号以周期 T 重复，它是一系列基频 f(=1/T)正弦信号和整数倍基频的正弦信号的叠加。数字滤波是分离所选组件的过程。噪声导致的失真信号可以通过使用数字滤波消除噪声的方法，重新得到正确的信号。数字滤波不仅可以检测到电话中传送的音调，而且可以消除信号中的回波。当对来自传感器的超声波扫描信号进行滤波时，滤波结果只显示诊断者感兴趣的区域。数字滤波是 DSP 所作的一项重要的基本处理操作。

任何模拟信号必须通过采样转换成数字信号。每间隔时间 T 进行一次信号采样操作。采样得到的信号可以通过乘以一个常数进行缩放。每次采样，x_n 可以进行编码得到信号 y_n。考虑如下方程式：

$$y_n = ax_n + \sum(b_{-j} \cdot y_{-j}) + \sum(a_{-k} \cdot x_{-k})，\text{其中 j 从 1 到 m，k 从 1 到 n} \quad \text{D.1}$$

$$y_n = \sum(a_i \cdot x_{n-i}) + \sum(b_l \cdot y_{n-l})，\text{其中 i 从 0 到 N，l 从 1 到 M} \quad \text{D.2}$$

方程式 D.1 可以按这样理解。假定 y 是第 10 个输出分量。第一项是第 10 个输入乘以一个常数的结果。第 2 项是倒数第 2、倒数第 3，直到倒数第 m+1 个乘积的总和。这里的乘积是指输出值“y”和系数“b”相乘的结果。第 3 项是最后 1 个、倒数第 2、倒数第 3，直到倒数第 n+1 个乘积的总和。这里的乘积是指输入值“x”和系数“a”相乘的结果。等式 D.2 可以按同样的方式来理解。

方程式 D.2 可以采用 DSP 操作中使用的标准形式(称为差分方程)来书写，如下所示：

$$y_n = \sum a_i \cdot x \cdot (n - i) + \sum b_l \cdot y \cdot (n - l) \quad \text{D.3}$$

此处方程式 D.2 中的下标使用系数代替了，它的意思是第 p 个系数是一个简单的 p。

该项的一个特例是 $y_n = (a_i \cdot x_{n-i})$，参见 2.1 节，它用在 FIR(Finite Impulse Response Filter，有限脉冲响应滤波器)中。FIR 滤波器将系数 b_l 设为 0，其含义是，不需要本次采样之前的输出。信号相位是线性的。另一个滤波器叫做 IIR(Infinite Impulse Response，无限脉冲响应)滤波器，它在信号相位是非线性时使用。

DSP 中的 MAC(Multiply and Accumulate，乘和累加)单元提供了两个操作数的快速乘法操作，以及单个地址结果的累加。它可以快速计算出 Sum = Sum + [$a_i \cdot x \cdot (n - i)$]的结果，这是方程式 D.3 中第 2 项或者第 3 项中的表达式。DSP 也可以有双 MAC 单元，同时计算方程式 D.2

或者 D.3 中第 1 个和第 2 个求和项。

所以 DSP 进行多项求和运算应该比传统的 GPP(通用处理器)更快，而且可以处理 VLIW(参见 6.3 节)。它利用 MAC 的数量，而不是传统处理器中的 MIPS 来度量 DSP 的性能。

D.3 定点运算和浮点运算的比较

假设系数为 0.35967，变量为 1.4824931。

(a) 定点运算

首先，35967 和 14824931 可以在定点运算处理中相乘，然后中间结果在最后一步被 10^{12} 除，从而得到最终结果。

(b) 浮点运算

0.35967 和 1.4824931 首先使用浮点格式表示进行乘法操作，然后按位保存结果。在浮点运算处理期间，处理器遵循多级处理方式。

(c) 定点运算中的精度丢失

定点运算必须考虑可能发生的溢出。溢出是指结果超出处理器寄存器的容量时，将溢出内容作为一个字存储的操作。在必要的情况下，两个相乘的十进制数可以先除以一个适当的数，保证中间结果不会超过寄存器中可存储的最大数值。在 32 位和 64 位处理器中，对于 32 位的整数操作，这个最大数值是 2147483648($=2^{31}$)，对于 64 位的长整数操作，最大数值是 9223372036854775808($=2^{63}$)。

下面通过一个简单的例子来理解精度丢失问题。假设要使用一个 8 位的处理器来实现 0.29 和 0.15 的乘法操作，寄存器中最大可以存储 8 位数的结果。因而存储的最大和最小 8 位整数分别是 127 和-128。假设处理器只有定点算术单元，29 和 15 相乘时，就会产生溢出。这是因为寄存器中不能存储中间结果(=435)。所以，我们选择一个除数 4，中间结果和它的乘积才是最终的结果。所以， 0.29/2 = 0.145 和 0.15/2 = 0.075 选择从定点运算开始。现在，处理器先舍去每个数的最后一位，然后进行 14 和 7 的乘法操作。中间结果是 98。而实际的中间结果应该是 108.5(0.29x0.15/4 = 0.435/4 = 0.010875)。

在一个处理器中，定点运算比浮点运算速度更快。这是因为在浮点运算中需要较多的步骤。具有 4800 MIPS 性能的处理器可以获得 1350 MFLOPS 的性能(记住，MIPS 的含义是百万条指令每秒，MFLOPS 的含义是百万条浮点操作每秒)。

如果处理结果在可接受的精度内，则在 DSP 操作期间使用 64 位定点操作时，需要舍去精度。而另外一方面，如果处理结果在可接受的精度内，则在 DSP 操作期间使用 32 位定点操作时，需要舍去较多的精度。带 32 位定点操作的 DSP 的处理速度，比带 64 位定点操作的大约快 2 倍。对于同一个存储在计算机中的 32 位数，定点操作比浮点操作大约快 4 倍。

偶尔也会需要精度，这就可能需要浮点操作。例如，在 IIR 滤波器中，系数 b_1 的精度就很重要。

D.4 嵌入式系统的 DSP

Texas 仪器股份有限公司(Texas Instruments Inc)拥有名为 TMS320C2000™、

TMS320C5000™ 和 TMS320C6000™ 的 DSP 平台。它们实现了所有设备之间的代码完全兼容。因而，该平台允许在以后的系统升级过程中，重用现有代码(访问站点 http://www.dspvillage.ti.com，了解更多详细信息)。

D.4.1 TMS320C2000™ 平台

TMS320C2000 系列平台由高精度控制的 32 位 DSP 控制器组成，性能参数可达 152 MIPS。它集成了许多外设，并提供了一种独特的片上外设组合方式。其应用示例如下：(a)工业应用；(b)自动控制应用；(c)光纤网络；(d)手持电源工具；(e)智能传感器。它比早期的 8 位或 16 位微控制器，具有更大的灵活性，控制着优化的外设和功能强大的处理器。

D.4.2 TMS320C5000™ 平台

设备中的电源效率、高性能系统和可选的外设，对系统设计者在移动 Internet 系统和无限通信系统方面的应用有一定的限制。而 DSP 平台为移动媒体和通信产品的应用提供了性能优化的嵌入式系统，这些应用可以是数字音乐播放器、GPS(Geographical Positioning System，全球定位系统)接收器、移动医疗设备、可视电话、调制解调器、3G 手机和移动成像。

以下是在开发应用程序过程中需要用到的一些工具：

(1) IDE 是 Code Composer Studio，评估模块(EVM)，以及免费的评估工具。

(2) DSK 启动工具包。

(3) XDS560 仿真器。

(4) 开发客户端电话时，客户端开发者使用的工具包。

(5) 芯片支持库。包括外设的驱动程序。

(6) DSP 和图像库。可以免费下载的，与平台相关的库(称作 TI 基础软件)。使用这些库函数可以缩短系统的开发时间。其中包括一些高级模块和优化的 DSP 函数。

(7) Code Composer Studio IDE 是 DSP 指定的代码设计器。它提供了类似于 MS Visual C++ 的开发环境。IDE 环境由以下部分组成：(a)多级(C 和 DSP 汇编)调试器；(b)C 和 DSP 汇编指定编辑器；(c)探测点；(d)文件 IO；(e)综合数据可视化显示；(f)基于 C 的 GEL 脚本语言。

(8) C5000C 是在 TMS320C5000™ 平台上使用的编译器、汇编器和链接器。编译器有 3 个作用：(a)通用 C 代码的性能接近手动编写的汇编代码。(b)C 运行时环境的编程接口。实现了汇编语言中关键的 DSP 算法，使性能得到极大的优化。(c)易于使用 C 语言编写的高性能应用程序。

TMS320C5000™ 平台已经出现了 TMS320C54x™ DSP、TMS320C54x™ (DSP + RISC)和 TMS320C55x™ DSP 几代产品。

D.4.3 TMS320C6000™ 平台

TMS320C6000™ 平台使用新技术设备，具有类似 RISC 编码的性能和优化的效率。其性能远远高于 5000 系列，每 MHz 操作的功耗很低。这些设备带有 2 级缓存。存储器、外设和协处理器一起提供了在宽带服务、高性能声音图像处理方面的应用。

TMS320C6000™ 平台已经出现了定点操作的 TMS320C62x™ DSP、TMS320C64x™ DSP 和浮点操作 TMS320C67x™ DSP 这几代产品。它们互相兼容。

以下是开发应用程序时需要用到的一些工具：

(1) IDE 是 Code Composer Studio，评估模块(EVM)，以及免费的评估工具。

(2) 网络视频开发工具包。

(3) 图像开发工具包。

(4) 网络开发工具包。

(5) DSK 启动工具包。通过并口、16 位数据转换器 TLC320AD535、电源管理设备和 PC 相连。

(6) 用于仿真和调试的 JTAG 控制器。

(7) XD560 仿真器。

(8) 芯片支持库。包括外围设备、可免费下载的平台相关的库和 TI 基础软件。它们有高级优化 DSP 函数。

(9) DSP 和图像库。可免费下载的、与平台相关的库(称作 TI 基础软件)。使用这些函数可以缩短系统的开发时间。包括一些高级模块和优化的 DSP 函数。

(10) Code Composer Studio IDE 是 DSP 指定的代码设计器。它提供了类似于 MS Visual C++ 的开发环境。IDE 环境由以下部分组成：(a)多级(C 和 DSP 汇编)调试器；(b)C 和 DSP 汇编指定编辑器；(c)探测点；(d)文件 IO；(e)综合数据可视化显示；(f)基于 C 的 GEL 脚本语言。

(11) 高性能 C6000C 引擎，编译器借助了一定的体系结构，维持其最高性能。它使用可免费下载的优化工具，平衡了代码长度，从而加快了高性能应用程序的设计开发。

(12) 参考框架。

D.4.4 DSP 的 TMS320C24x 和 C28X 代产品

TMSC24x 给出了用于数字控制应用领域的 16 位数据定点 DSP 内核。它提供 SCI、SPI、CAN、A/D、事件管理器、watchdog 定时器和片上闪存。C24x 内核拥有 20~41 MIPS 的计算带宽，可以运行大量复杂的实时控制算法。它具备以下性能：(a)无传感器速度控制；(b)随机 PWM；(c)矫正功率系数；(d)同其他 C2000 系列设备的代码兼容；(e)代码效率很高。

TMSC28x 内核给出了用于数字控制应用领域的高性能 DSP 内核。它提供了 32 位数据定点 DSP 和 SCI、SPI、CAN、12 位 A/D、McBSP、Watchdog 定时器、片上闪存。C28x 内核拥有最多 400 MIPS 的计算带宽，可以运行大量复杂的实时控制算法。它具备以下性能：(a)无传感器速度控制；(b)随机 PWM；(c)矫正功率系数；(d)和其他 C2000 系列设备的代码兼容；(e)代码效率很高(McBSP (high-speed communication Multi-channel Buffered Serial Port) 的含义是高速通信多通道缓冲串口。访问站点 http://www.ti.com/sc/docs/psheets/ abstract/apps/spra638a.htm)。

TMS320C24x 有 15 个设备，其中一些选项和通道的 ADC 数量、CAN 模块、串口以及闪存相关的选项。表 D-3 列出了这些设备的特性。嵌入式系统的设计者可以依据系统需求，准确地选择其中一项。

表 D-3 TMS320C2000™ 平台和 TMS320C24x™ DSP 及其示例 TMS320FC2402A 的特性

C2000™ 系列平台的特性	C24x™ 系列 DSP 的特性	TMS320C24x™ 代 DSP 中具有 15 个设备的示例 TMS320FC2402A
时钟率	20MHz~40MHz	40MHz
DSP 内核处理器	定点 DSP	定点 DSP
以字为单位的 RAM	544 或 1k	1k
以字为单位的 ROM	0、4k、6k、8k、16k 或 32k	0k
以字为单位的闪存	0、4k 或 16k	8k
ADC 10 位通道	2、5、8 或 16	8
以纳秒为单位的 ADC 转换时间	6100、900、500、425 或 375	500
CAN 模块	有或者无	无
PWM 通道	7、8 或 12	8
定时器	2、3 或 4	2
串口	1 或 2	1
启动加载器	无或者 ROM 或者闪存	ROM
20~400 MIPS 之间的性能	20 或 40 MIPS	40 MIPS
外部存储器接口	有或无	无

D.4.5 TMS320C54x 和 TMS320C55x 代 DSP

TMS320C54x 代产品有 17 个以上的代码兼容设备，提供了很多性能选项和外设选项，低功耗操作及创新的体系结构和指令集。它为系统设计者提供了在低开销系统中获得高性能、低功耗的有效方法。C5470 和 C5471 系统级 DSP 集成了 DSP、RISC、已有操作系统和对新一代设计开发的完全支持。

TMS320C55x 代产品提供了有效功率最大的 DSP，因而可以应用在移动 Internet 设备和高速无线通信等方面。使用高级的功率管理技术，获得超低功耗性能。在外围设备、存储器和核心功能单元变到非活动状态时，其功耗自动降低。C55x DSP 的内核是 OMAP 5910 处理器(内核是指芯片上集成了其他设备的一块区域)。

当 OMAP 在一块芯片中集成了使用 TI 升级型号 ARM925 的 C55x DSP 核心时，可以得到低功耗的高性能优化组合。这是一种独特的体系结构。嵌入式系统 DSP 和 ARM 开发者结合 ARM 的命令和控制功能，获得了 DSP 的低功耗实时信号处理能力。

表 D-4 给出了 TMS320C5000™ 平台中具有 20 个以上设备的特性，其中每个设备都包含处理器 OMAP5910。嵌入式系统设计者可以根据系统需求从中选择一个。

表 D-4 TMS320C5000™ 平台和 TMS320C54x™ DSP 及其示例 TMS320LC54V90 的特性

C5000™ 系列平台的特性	C54x™ 系列 DSP 的特性	TMS320C54x™ 代 DSP 中具有 17 个设备的示例 TMS320LC54V90
时钟率	40MHz~60MHz	117MHz/58.98MHz
DSP 核心处理器	OMAP5910	OMAP5910
周期时间	8.33ns~25ns	8.5ns/17ns
每个单元 MHz 操作的功耗	0.33mA/MHz	0.33mA/MHz
以 MIPS 为单位的性能	最多 900 MIPS	117.96/58.98 MIPS
存储器接口	高级多总线体系结构，3 个独立的 16 位数据存储器总线和 1 个程序存储器总线	高级多总线体系结构，3 个独立的 16 位数据存储器总线和 1 个程序存储器总线
以字为单位的存储器	64K 数据存储器、1M~8M 程序存储器、5k~40k 的 RAM、2k~128k 的 ROM	64K 数据存储器和 8M 程序存储器，40k 的 RAM 和 128k 的 ROM
定时器(16 位)	0 或 1 或 2	2
通信 I/F	HPI(主机端口接口)，HPI 8/16	HPI 8/16
外部 DMA 通道支持	0 或 6	6 个通道
串口标准串口	0~2	0
串口总和	0~2	2
TDM 串口	0~1	0
缓冲串口	0~3(选项 McBSP[2])	2
其他串口	UART、DAA[1]	1 UART + 1 DAA
启动加载器	可选	0

1. DAA(Direct Access Arrangement)的含义是直接访问排列 (http://e-insite.net/edmag/)。DAA 串口具有模拟输入和输出功能，最多包含 1 个主 CODEC 和 7 个辅 CODEC。CODEC 是指一个编码解码单元，它分别在输出和输入时，通过 ADC 和其他操作对数字信号编码，通过 DAC 和其他操作对模拟信号解码。例如，视频 CODEC 单元完成这样的工作：(i)编码。进行降低噪声的预处理，在估计了图片移动的速率之后控制传送的速率。它压缩音频信号并进行同步操作，最终将比特流送到输出部件和流网络。(ii)解码。接收比特流，解压分离音频信号和视频信号，并通过预处理消除噪声。

2. 参见 D.4.4 节。

D.4.6 TMS320C62x、64x 和 C67x 代 DSP

TMS320C62x 代产品通过使用一些新技术展示了定点 DSP，同时，这也促使启用新的装备，并激励了已有技术的最终实现。这些产品提供了多通道多功能应用。其应用示例有：(a)无线基点；(b)数字用户环路(xDSL)系统；(c)远程访问服务器(RAS)；(d)高级图像学/生物统计学；(e)工业扫描器和安全系统；(f)多通道电话。

TMS320C64x 代产品也带有定点 DSP。这代产品的设备运行性能可以达到 600MHz~4800

MIPS。另外，增加的新指令在一些关键应用领域中提高了产品的性能。新的应用领域例如：(a)数字通信基础设施；(b)视频和图像处理。

TMS320C67x 代产品带有浮点 DSP。它在开销敏感的应用领域中取得了创新。应用示例有：(a)声音和语音识别；(b)高端图形和图像学；(c)工业自动化。

表 D-5 列出了 TMS320C67x™ 代产品中具有 14 个以上设备的特性。嵌入式系统设计者可以根据系统需求准确地从中选择一个。

表 D-5 TMS320C67x™ DSP 及其示例 TMS320C6711-100 的特性

C6000™ 系列平台的特性	C67x™ 系列 DSP 的特性	TMS320C67x™ 代 DSP 中具有 17 个设备的示例 TMS320C6711-100
DSP 内核处理器	浮点处理器	浮点处理器
周期时间	4.4 ns~10 ns	10 ns
1200~4800 MIPS 之间的性能	最多 4800 MIPS	1200 MIPS
600~1350 MFLOPS(67x 系列)	最多 1350 MIPS	600 MFLOPS
DMA 通道	4 DMA 或者 16 EDMA	16 EDMA
以字为单位的存储器	512k 程序位和 512k 数据位，或者 32 位 L1P 程序缓存和 32 位或 64 位 L1D 数据缓存，在 6711 和更高版本中是 512k 位 L2 高速缓存	32k 位 L1P 程序缓存和 32L1D 数据缓存和 512k 位 L2 高速缓存
主机端口或者扩展总线或者 PCI	没有，或者 16 位 HPI	16 位 HPI
通用 IO	0 或 1 或 1(16 管脚)	0
McBSP	2	2
MsASP	0 或 2	0
定时器(32 位)	2	2
外部存储器接口	32 位或 PYP 16 位或 GDP 32 位	32 位
I^2C	0 或 2	2
内核电源电压	1.2V~1.8V	1.8V
IO 电源电压	3.3V	3.3V

D.4.7 RISC 环境下的 OMAP5910 嵌入式处理器 DSP

OMAP5910 双内核是 RISC 环境下最新的 DSP，集成了 TMS320C、TMS320C55x 和 ARM925，包含带 USB1.1 的 192kB RAM。另外有 1 个主机端和客户端，以及 MMC/SP 卡接口。

D.4.8 基于 SoC 的解决方案 Texas DSP TMS320DM310

最近公布的数字媒体应用方面 RISC 环境下的 DSP 创新产品是基于 Soc 的 Texas DSP TMS320DM310。它集成了 TMC320C55x 和 ARM925，包含带 USB1.1 的 192kB RAM。另外有 1 个主机端和客户端，以及 MMC/SP 卡接口。TMS320DM310 的特性如表 D-6 所示。

表 D-6　TMS320DM310 的特性

特　　性	用于 Soc 的 TMS320DM310 DSP 数字媒体的特性
DSP 内核处理器	TMS320C55x
RISC 内核	带高速缓存的 ARM 9
协处理器	对 640 x 480 VGA MPEG2/MPEG4 视频信号实时解码和对 352 x 288 CIF MPEG1、MPEG 2 和 MPEG4 实时编码的图像协处理器
时钟	125 MHz DSP 核心和 160 MHz ARM RISC
性能 MIPS	最多 2125 MIPS，可编程
直接 IO 的 USB 主机	有
应用	6M 像素静止数码相机的 1s 图像捕捉、数字成像和数字录音、MPEG$ 和 JPG 移动设备、web-pad

关键词及其定义

- MAC 单元：在 DSP 操作中对$[ax_n + \sum(b_j . y_j)]$这类表达式进行快速计算的单元。
- Code Composer Studio：用于 TI DSP 特殊代码设计的 IDE，提供了类似于 MS Visual C++ 的开发环境。它由以下部分组成：多级(C 和 DSP 汇编)调试器、编译器、汇编优化器、优化性能和效率的类 RISC 汇编代码和类 RISC 调度、探测点、文件 IO 函数、综合数据可视化显示和基于 C 的 GEL 脚本语言。
- CODEC：编码解码单元，它分别在输出和输入时，通过 ADC 和其他操作的对数字信号编码，以及通过 DAC 和其他操作对模拟信号解码。它用在音频信号和视频信号处理方面。
- 代码兼容性：产品系列中各代产品之间代码的可重用性。
- 向前代码兼容性：产品系列中各升级版本中代码的可重用性。
- 向后代码兼容性：产品系列中各早期版本中代码的可重用性。
- DAA：直接访问排列，例如，典型的 DAA 串行输入输出端口，直接使用最多 1 个主 CODEC 和 7 个辅 CODEC 传送模拟输入输出信号。
- McBSP：高速通信多通道缓冲串口。
- DCT：在许多 DSP 函数中使用的离散余弦变换函数，例如 MPEG2/MPEG4 压缩。
- 数字滤波：使用 DSP 函数的信号滤波器。
- 回波：在一段延迟后接收到的信号，它会叠加到初始信号上。例如，我们在墙内或者山里会同时听到原声和回音。
- 回波消除：消除回波的处理过程。
- 定点运算：使用处理器寄存器或存储器对有符号或无符号整数进行的运算。
- 浮点运算：使用处理器寄存器或存储器的运算，其中存储的十进制数和小数都是标准的浮点格式。
- 噪声消除：消除随机引入无关信号的处理过程。
- OMAP 5910 处理器：一种 TI 处理器，采用了高性能低功耗 DSP 芯片中独特的体系结构。
- 实时视频处理：视频信号的处理，比如在一个时间帧内处理所有或者大部分输入帧，以使每个处理帧之间保持一定相位差，同时在它们之间保持一定的时间间隔。

附录E 嵌入式系统应用的新型处理器

回顾1.2.5节和1.2.6节。某些应用领域的复杂嵌入式系统，如带流图像和无线Internet的移动电话，其处理器应该具备以下性能，以保证资源、功耗和存储器的优化使用：

(1) 高MIPS和高MFPS。

(2) 较高的DSP性能。

(3) 较高的IO收发器带宽。

最新设计的媒体处理器具备了这些性能。该媒体处理器结合了视频处理器和IO处理器的功能。媒体处理器IC和嵌入式系统中核心的使用成为该领域的一项创新。而移动处理器，如2003年3月提出的Intel“centrino”是另一项创新。这使得带802.11 LAN无线数据链接的移动嵌入式系统的设计将会变得相当简单。

嵌入式系统的媒体处理器

视频处理器应该具备以下处理器功能：

(1) VLIW(参见6.3节)、定点和浮点运算。

(2) 离散余弦变换(DCT)处理单元。

(3) 数字转换器(数字转换是指对来自照相机或者耳机接口信号中的模拟信号进行编码，并转换成数字信号(比特流)输出)。

(4) 对图形、二维文本(固定维的文本)、MPEG和移动JPG(来自移动系统的JPG帧)的库函数的处理。

(5) 图像上色和色调调整、图像旋转、图像缩放、阴影增强、检测图像边缘和锐化图像。

(6) 视频信号编码，降低噪声的预处理，在估计图片移动速率之后控制传送速率，压缩同步音频信号，最后将比特流传送到流网络(streaming network)上。

(7) 视频信号解码，接收比特流，解压缩和分离音频信号和视频信号，并通过预处理消除噪声。

(8) 降低噪声和消除回波。

流网络定义为客户端通过服务器从网络上连续搜索报文或数据帧的网络，如同在实时音频和视频通信中的数据传送一样。流网络、智能化数码相机和电子娱乐产品都需要使用媒体处理器。

以下是可利用C语言编程的实时媒体处理器的应用示例：

(1) 用于网络广播(通过 Web 接收信号)的流网络。

(2) 音频流和视频流 Web 服务器以及 Internet 连接设备。

(3) 视频会议。

(4) 电子消费产品——像 DVD 这样的娱乐产品。

(5) 汽车通信电子产品。

(6) 轻便无线电话和传真。

(7) 带有 Internet 和视频图像访问功能的移动电话。

(8) X 线断层摄影、心动描记法和心血管方面的医疗电子设备，以及 X 射线显示存储器。

(9) 安全监视和观测系统。

(10) 语音处理。

(11) 降低噪声和消除回波。

使用 DSP 或 GPP 接收的图像帧首先需要实时地存储到存储器中，然后进行解压、解密和处理工作。

照相机的图像帧格式可以是水平像素乘以垂直像素，1024x768，640x480 或者 160x120。在视频会议中以 10~15 帧/秒的速度传输视频信号时，帧以两种标准速率，25 帧/秒和 30 帧/秒之一到达。帧的格式可以是为视频信号定义的两种标准格式中的一种。

同样，当用 DSP 或 GPP 传送图像时，图像帧需要首先缓冲到存储器中，然后进行解压、解密和处理工作，接着将它实时发送。实时视频处理器的处理能力比 DSP 处理器或者 GPP 处理器(general purpose processor，通用处理器)要快许多。由于很少情况需要多帧存储和处理前的缓冲，所以该过程也只需要较小的程序和数据存储器。

多媒体处理器提供了移动电话和宽带 Internet 的无缝结合。它促进了基于声音的 Web 访问、语音识别、文字声音转换、VoIP(IP 语音)和基于语音的移动 Net 标准 XML 版本的出现和发展。

Philips 半导体公司最近推出了 C 可编程实时多媒体处理器 Trimedia 1x00 系列产品(3 种多媒体是指数据、音频和视频)。它们具有视频音频处理能力，以及流网络处理功能，使得实时多媒体数据流的应用变得非常容易。Tri-Media TM-1300 多媒体处理器是该系列的成员。请访问站点 http://www.semiconductors.philips.com/trimedia/了解更多有关 TriMedia 系列产品的详细信息。关于 Philips TriMedia H.261 和 H.263 视频 CODEC 库的信息，请访问站点 http://www.4i2i/com/h_263_philips_trimedia.htm(提示：(a)TriMedia 是完全由 TriMedia 技术股份有限公司所有的一项媒体处理技术。TriMedia 是 TriMedia 技术股份有限公司的商标。(b)CODEC 是指对进入和流出信号的编码和解码)。

Philips 半导体公司最近还推出了 Nexperia PNX 1300 系列和 PNX 1500 系列。图 E-1 是 PNX 1300 多媒体处理器的示意图。表 E-1 列出了 PNX 1300 和早期的 TM-1300 处理器的特性。

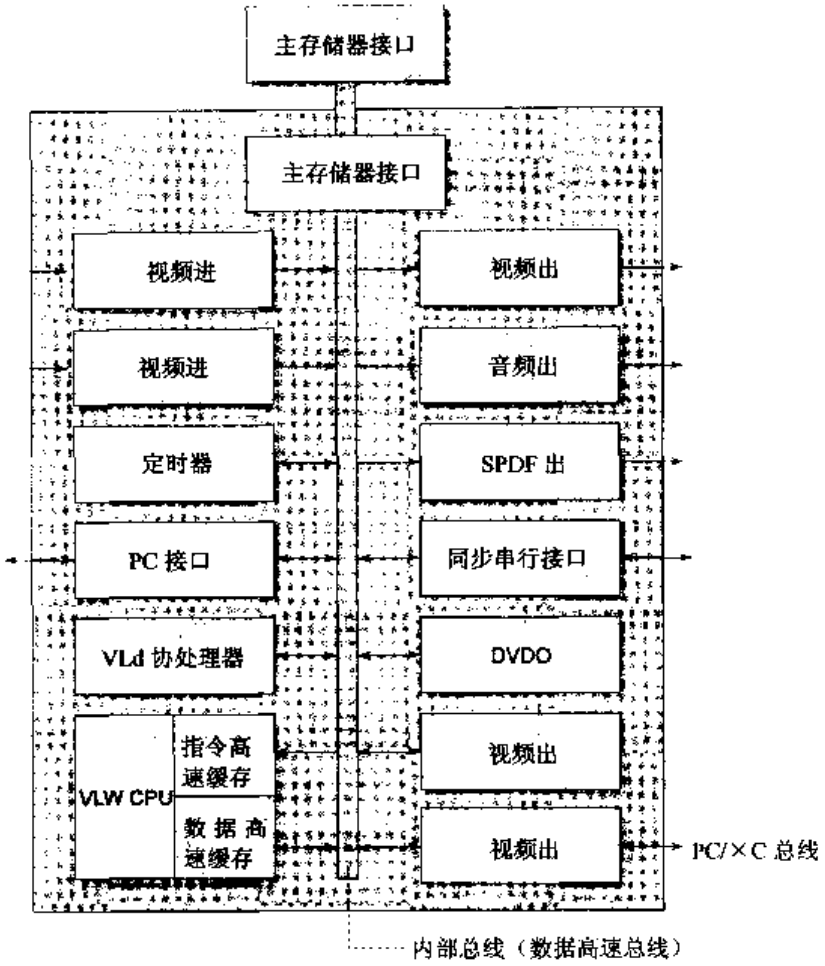


图 E-1　Nexperia PNX1300 系列媒体处理器的结构图

表 E-1　Nexperia PNX1300 系列和早期的 TM-1300 特性比较

特　性	TM-1300 处理器	Nexperia PNX1300 系列
时钟频率	100MHz	对于某些多媒体应用可达 200MHz 的处理功率
存储器接口	比 PNX-1300 慢	比 TM-1300 系列快
管脚兼容性	和 PNX 1300 兼容	管脚与 TM-1300 前期产品 100%兼容
结构单元	32 位高速公共数据内部总线到 SDRAM 的主存储器接口、视频输入单元、视频输出单元、音频输入单元、音频输出单元、定时器、I^2C 接口、同步串行接口、VLD 协处理器、DVDO、带指令和数据高速缓存的 VLIW CPU、专用图像协处理器和 PCI-X IO 接口	
C/C++可编程性	内置优化代码生成编译器单元	内置优化代码生成编译器单元
G 操作每秒	~G/s。比 PNX 1300 慢了许多	7G

(续表)

特　性	TM-1300 处理器	Nexperia PNX1300 系列
多媒体处理	视频、音频、图形和通信数据流处理	比以前的视频、音频、图形和通信数据流处理更快
限制功耗的多媒体设计中的操作电压系列	可选低电压媒体处理器	低电压操作的选项

关键词及其定义

- 多媒体处理器：多媒体处理器是一种高性能的新型处理器，使用在实时视频操作、音频处理和数据流化等方面。其结构单元包括 VLIW(参见 6.3 节)定点和浮点运算单元，离散余弦变换(DCT)处理单元，数字转换单元，处理图形、二维文本(固定维的文本)、MPEG 和移动 JPG(来自移动系统的 JPG 帧)的库函数的单元，图像上色和色调调整、图像旋转、图像缩放、阴影增强、检测图像边缘和锐化图像单元，以及视频信号编码和解码单元。
- 相对 GPP 和 DSP 的多媒体处理器：它是一种新型的多媒体处理器，就系统性能、存储器缓冲需求和帧存储需求而言，比同时用于实时视频操作、音频处理和数据流化的处理器的几个 DSP 的性能更加优良。传统处理器也提供定点和浮点运算操作(参见 2.1 节)，而且速度很快。但是它们的 ALU 只能处理加法、乘法、减法和除法这类指令。当处理视频、音频或者流网络的数据时，它们的性能会变得很差。
- 实时视频处理：视频信号的处理，包括加密、解密、压缩、解压、编码、解码和其他的操作，这些操作都是为了保证输入速率和流出速率的同步。假设在视频输入端的输入速率是 30 帧/秒，则在没有建立内部缓冲区和输出端没有延迟脓包的情况下，每秒钟应该有 30 帧信号得到处理。
- 多媒体处理器时钟频率：指处理器的时钟频率，它控制每秒钟的最大可能的操作数量。对于 TM-1300，这个频率是 100MHz，对于 Nexperia PNX 1300 系列是 200MHz(在较高时钟频率下的操作，同时也会有较高的功耗)。
- 移动处理器：能够对无线短距离(100m) 802.11 LAN 和移动技术方面的应用进行高级处理的处理器。
- 多媒体处理器总线：32 位高性能总线，同主板的 SDRAM 相连。它也用于取指令操作和代码缓存。高性能总线是指一种取数据的速率比时钟速度快几倍的总线。
- PNX 多媒体处理器的结构单元：这些单元有主存储器接口、同 SDRAM 相连的 32 位高速公共数据内部总线、视频输入单元、视频输出单元、音频输入单元、音频输出单元、定时器、I^2C 接口、同步串行接口、VLD 协处理器、DVDO、带指令和数据高速缓存的 VLIW CPU、专用图像协处理器和 PCI-X IO 接口。
- 在嵌入式系统中使用多媒体处理器：多媒体处理器日渐代替了 DSP，应用在复杂嵌入式系统中，用在低功耗的网络流，以及存储器性能优化的实时系统上。它们尤其适合

于网络广播通过 Web 接收信号)、音频流和视频流 Web 服务器、Internet 连接设备、视频会议、DVD 等消费电子产品-娱乐产品、汽车通信电子产品、轻便无绳电话和传真、移动电话、在 X 线断层摄影、心动描记法和心血管方面的医疗电子设备以及 X 射线显示存储器、安全监视和观察系统，以及声音处理。媒体处理器嵌入式系统的应用仍然在继续增多。

- 网络广播：将信息发布(广播)到许多注册过的 Web 地址。
- 流网络：网络中的客户端通过服务器从网络上连续搜索报文或数据帧的网络，如同在实时音频和视频通信中的数据传送一样。
- 音频流：像流水一样通过网络实时接收到的音频消息。
- 视频流：像流水一样通过网络(一般是宽带 Internet)实时接收到的视频消息。

附录 F　串行和并行总线

F.1　新的串行总线标准(USB 2.0，IEEE 1394)

3.3 节描述了串行通信总线，“I^2C”、“CAN”、“USB”和高级串行总线。表 F-1 列出了包括新的总线标准 USB 2.0 和 IEEE 1394 的串行总线的一些显著特性。

表 F-1　连接设备的串行总线

特　性	详 细 信 息
按比特率递增的顺序排列的串行总线	UART (512 Baud/s)，1 线 CAN (33 kb/s)，工业 I^2C(100 kb/s)，SM I^2C 总线(100 kb/s)，SPI(100 kb/s)，容错 CAN(110 kb/s)，串口(230 kb/s)，MicroWire (300 kb/s)，I^2C(400 kb/s，2 米)，高速 CAN (1 Mb/s)，IEEE 1284(2.4 Mb/s)，高速 I^2C(3.4 Mb/s，0.5 米)，USB 1.1(低速通道 1.5 Mb/s 和 3 米，高速通道 12 Mb/s 和 25 米)，SCSI 并行(40 Mb/s)，快速 SCSI(8M~80 Mb/s)，Ultra SCSI-3(8M~160 Mb/s)，FireWire/IEEE 1394(400 Mb/s，72 米)，高速 USB 2.0(480 Mb/s，25 米)
UART	简单，非即插即用，不安全，低速，没有主从模式
USB 2.0	速度接近 IEEE 1394，方便 UHF 传送，即插即用，总线或外部电源供应，和串口、IrDA 和 Ethernet 有接口，主从模式
IEEE 1394	主机最多可连接 63 个设备，4.5 米中继电缆，最多 16 个中继段，传输速率恒定，确保固定带宽，两个隔离的数据总线对，闸门、电源对可以包含在单个电线对中，IEEE 1394 专用连接器、DVD IO 和 MPEG2 的总线标准，数据吞吐量较高、数据速率标准 100 Mb/s、200 Mb/s、400 Mb/s ，数据速率标准 3.2 Gb/s，即插即用，设备自动重配置，主从模式，允许热插拔
CAN	安全，比 USB 和 I^2C 慢，非即插即用，复杂寻址，主从模式
I^2C	简单即插即用，效能成本合算，速度仅高于 3.4 Mb/s

F.2　新的并行总线标准(Compact PCI、PCI-X)

3.4 节描述了主机和计算机系统并行通信接口 ISA、PIC、PCI-X 和高级总线。表 F-2 列出了包括新的总线标准 PCI-X 和 Compact PCI(cPCI)的并行 PIC 总线的显著特性。

表 F-2 并行 PCI 总线

标 准	特 性
PCI 2.2	64 位总线，66 MHz 选项，32 位 33 MHz 吞吐量=133 Mb/s，完全组件级别，连接器(50 个信号的 94 管脚连接器)和卡板说明书，多路复用 AD0： 31 总线，支持双地址 64 位。信号上反映的无限制总线和信号中继用于获取最终数值(参见 3.4 节)
PCI-X(扩展 PCI)	对现有 PCI 卡的向后兼容，改善了高带宽设备(光纤通道和处理器，它们是机群和千兆 Ethernet 中的一部分)中 PCI 的速度，从原来的 133 Mb/s 到现在的 1 GB 之多
Compact PCI(cPCI)	只使用 Euro(VME)卡 3U/6U 格式的 PCI 说明书，这种格式带 2 毫米连接器，并且它的总线使用 8、16、32 或 64 位进行传输。最大 264 Mb/s 的吞吐量，6U 卡包含用户定义 I/O 的辅助管脚，支持热插(热交换)，背面(不同的连接器上)支持两个独立总线，支持 Ethernet，无限带宽，支持星型结构(基于开关结构的系统)
PXI	对装置、附加信号和底盘需求额外增强的 cPCI，等同于 VXI 加上 VME(与 VXI 不同，PXI 不要求卡被封装在金属外壳中)

关键词及其定义

- USB 2.0：新的 480 Mbps USB 总线标准，适用于主机和即插即用设备之间 UHF 数据传输，距离可达 25 米，可以和串口、IrDA 和 Ethernet 相连接。
- IEEE 1394 总线：用于 DVD IO 和 MPEG2 的新总线标准，具有很高的数据吞吐量。详细信息请参考表 F-1 中的文本内容。
- 即插即用：可以直接和总线连接，不执行设备初始化和配置代码就可以使用的设备。
- 设备自动重配置：总线上的主控制器可以自动检测到有新设备连接或拔出。它在检测到新的设备连接时，自动进行设备初始化和配置代码。
- 热插：在正在运行的系统上连接设备，不需要重启或重新配置系统。
- 主从模式：有一个控制其他设备的主控制器，其他设备称为从设备(主设备以外的设备)。

附录G　嵌入式系统中的设备

该附录提供了嵌入式系统中使用的存储器、RAM/ROM/闪存设备、设备编程器、SPI、SCI、UART、定时器、PWM 和 Watchdog 定时器的详细介绍。

G.1　各种形式的 ROM 设备

表 G-1 列出了嵌入式系统中各种形式的 ROM，并对它们的特性进行了比较。擦除整个 ROM 或者闪存中的一个存储扇区，分别是指将所有存储单元或者一个扇区的存储单元置 1。编程的过程就是使用设备编程器将 1 置为 0(参见附录 G.2)。该表解释了在何处编写 ROM 程序、如何擦除、如何刻录(写入文件称作刻录)以及每种类型的 ROM 可以写入的次数。

表 G-1　各种形式的 ROM 及其特性比较

ROM 类型	在何处编写程序	如何对位编程 (1 变为 0)	如何擦除位 (0 变为 1)	一次可擦除的内容	可编程次数
屏蔽 ROM	系统 ROM 的屏蔽生产商	适当屏蔽硅片上的电路	不可擦除	不可擦除	0
EPROM	系统设计者或者系统生产厂家所用的设备编程器[1]	使用实验室中的设备编程器对位编程	使用紫外线照射它的石英窗口来进行擦除[2]	一次擦除所有内容	100 次以上
PROM (OTPROM)	系统设计者或者系统生产厂家，仅编写一次	使用实验室中的设备编程器对位编程	由于它只能编程一次，所以没有擦除操作	不可擦除	仅 1 次
EEPROM	首先在使用设备编程器设计的系统上，然后在嵌入式程序的内置电路上	在系统内置电路上或者设备编程器上，通过两次写操作逐字节编程——首先将 8 位全都置 1，然后按所需字节置 0	在系统的内置电路上或者设备编程电路上，通过对所有位置 1 进行擦除	一次 1 个字节	1 百万次以上
闪存	首先在使用设备编程器设计的系统上，然后在嵌入式程序的内置电路上	如果它的某个扇区以前被擦除了，那么就使用一次写入所需字节的方式在系统内置电路上或者设备编程器上编程	在闪存的内置电路上或者设备编程电路上，通过向所选扇区写入 1 来进行擦除操作	闪存中一个扇区的字节	10K 以上(在最新的闪存上)

1. 设备编程器使用适易的软件工具，该软件包括一个可以具体到对某个设备编程的程序。

2. 紫外线照射功率是~12mW/cm^2，照射位置在芯片的石英窗口上 2.5cm 处，照射时间大约为 1000s。

(3)ROM/EPROM/EEPROM 在编程后可以在微控制器系列的不同型号上验证(验证的意思是读取一个已编程的 EPROM 或者 EEPROM，检查应该编程的位是否已编程)。

G.2 ROM 设备编程器

考虑一个带有 512kB 可编程存储器的设备。它有 8 个信号位(D_0~D_7)和 19[= $\log_2$(512* 1024)]个地址 A_0~A_{18}。总共有 524, 288(=512*1024)个数组单元，每个数组有 8 个单元。每个单元都有一个用于输出的 D 位，在未编程状态是逻辑 1。对设备编程(定位需要的字节)是指在相应的每个数组单元地址上用 0 代替 1。保存的字节在编程后通过嵌入式系统的定位程序来生成(编程后，设备存储器部分将根据软件开发、测试和调试周期结束之后的需求保留最后的字节)。

ROM 设备编程器是对微控制器，或者设备中的 PROM、EPROM 芯片或单元进行编程的系统。对插入插槽(在设备编程器电路上)的设备编程时，将计算机和该电路连接起来，使用计算机上的软件工具向每个地址传送字节。设备程序需要在输入时查找输出记录。该输出必须反映最终的设计，只有设备程序可以将最终的输入放入 ROM。对于正在编程的设备，输入到设备编程器的记录格式有 3 种，如下所示。

G.2.1 二进制映像

映射的二进制位(二进制映像)是指按照从开始地址到结束地址的顺序发送的字节。

G.2.2 Motorola S-Record 格式

Motorola S- Record 格式是存储定位程序文件的一项工业标准，在它之前是使用设备编程器或者屏蔽 ROM 设计程序(它被称为 S- Record 是因为这种格式的每一行首字符都是 S)。例如：第 1 个字符是 S，第 2 个字符是 2(指明记录类型)，第 3 个和第 4 个是 14(指明在这一行有 20 个字节)，剩下的 40 个字符(半字节)分为地址(3 字节)、数据(16 字节)和校验和(1 字节)。表 G-2 给出了一个作为定位程序输出和设备编程器输入的 S-Record 格式。留给读者一个练习，填充表 G-2 中的第 6 行记录，其中 Addr 是 0x000037。

表 G-2 Motorola S- Record 格式示例

行号[1]	第 1 个字符	第 2 个字符[2]	N[3]的第 3 个和第 4 个字符，	地址 Addr[4]	从 Addr 开始的存储在 ROM 中的 N_d[5] 字节(N_d 的最大值可以是十进制数 253)	校验和[6]
0	S	2	1 0	000000	aa bb cc dd ee ff xx yy zz bb cc dd	cs0
1	S	2	0 C	00000C	cc aa cc dd ee ff xx yy	cs1
2	S	2	1 2	000014	dd bb cc dd ee ff xx yy zz bb cc dd aa xx	cs2
3	S	2	0 5	000022	0A	cs3

(续表)

行号 1	第 1 个字符	第 2 个字符[2]	N3 的第 3 个和第 4 个字符，	地址 Addr[4]	从 Addr 开始的存储在 ROM 中的 Nd5 字节(Nd 的最大值可以是十进制数 253)	校验和[6]
4	S	2	0 8	000023	dd bb cc dd	cs4
5	S	2	1 4	000027	dd bb cc dd ee ff xx yy zz bb cc dd aa ff 01 c0	cs5

1. 行号不在记录中。

2. 是指该行中数据记录的可用性。数据中的字节按顺序刻录到 ROM 中。

3. N = 10 意思是该行有 16 个十六进制字节，包括行尾用作地址的 3 个字节和用于校验和的 1 个字节。该行指定存储的数据字节的数量 N_d = 12(十进制数)。0C 意思是 N = 12，N_d = 8(十进制数)

4. 占 3 个字节的开始地址，000000 意思是接下来的 12 个字节存储在地址 0x0000~0x000B 之间。所以下一行的开始地址 Addr = 0x000C。

5. 该行是将要刻录到 ROM 的字节，编号为 N_d。该列的每个字符都代表一个半字节。

6. cs0、cs1，……分别是 0，1，……行中每个字节的校验和。

G.2.3 Intel Hex-File 格式

Intel Hex File 格式是存储定位程序文件的另外一项工业标准，在它之前是使用设备编程器或者屏蔽 ROM 设计程序。例如：第 1 个字符为“:”(冒号)，第 2 个和第 3 个字符用于指定该行的数据个数(假设在 N_d = 16 的情况下，个数为十六进制数 10)(地址字节、校验和字节和数据类型字节除外，只计算该行实际的数据字节，它们将被刻录 ROM)。第 4~7 个地址(2 个字节)，第 6 个和第 7 个分别是 0 和 0，用来指明数据是 ROM 中的，剩下的 32 个字符是数据内容(16 个字节)，2 个字符用作校验和(1 个字节)。表 G-3 列出了 Intel hex-file 格式示例，其中的数据分别和表 G-2 中对应的 Motorola S -record 格式的数据相同，它们作为定位程序的输出和设备编程器的输入。留给读者一个练习，请填充表 G-3 记录的第 6 行，其中 Addr 是 0x0037。

表 G-3 Intel Hex- File 格式示例

行号[1]	第 1 个字符	C_d 的第 2 个字符和第 3 个字符[2]	地址 Addr[3]	第 6 个和第 7 个字符[4]	从 Addr 开始的存储在 ROM 中的 N_d[5] 字节(N_d 的最大值可以是十进制数 253)	校验和[6]
0	:	0 C	0000	0 0	aa bb cc dd ee ff xx yy zz bb cc dd	cs0
1	:	0 8	000C	0 0	cc aa cc dd ee ff xx yy	cs1
2	:	0 E	0014	0 0	dd bb cc dd ee ff xx yy zz bb cc dd aa xx	cs2
3	:	0 1	0022	0 0	0A	cs3
4	:	0 4	0023	0 0	dd bb cc dd	cs4
5	:	1 0	0027	0 0	dd bb cc dd ee ff xx yy zz bb cc dd aa ff 01 c0	cs5

1. 行号不在记录中。

2. 该行指定的用于存储的数据字节数，C_d等于十进制数 12。0C 是指 C_d=12。

3. N = 10 意思是该行有 16 个十六进制字节，包括行尾用于地址的 3 个字节和用于校验和的 1 个字节。该行指定用于存储的数据字节数 N_d = 12(十进制数)。0C 意思是 N = 12，N_d = 8(十进制数)

4. 2 个字节的开始地址，0000 意思是接下来的 12 个字节存储在地址 0x0000~0x000B 之间。所以下一行的开始地址 Addr = 0x000C。

5. 将要刻录进 ROM 字节处在该行，并且编号为 N_d。该列的每个字符都表示一个半字节。

6. cs0，cs1，……分别是 0，1，……行中每个字节的校验和。

G.2.4 设备编程器的编程方法

当对设备编程器电路在 12V 高电压的情况下应用持续几毫秒的选通脉冲时，512kB 的设备数组单元(在 A_0~A_{18}信号位上定义的地址)按照 D_0~D_7上的每个 0 位存储“0”值。对其存储器单元编程的设备编程器，使用软件工具、计算机和设备编程器电路，按顺序执行以下 8 个步骤：(i)必要时应用 A0~A18 作为所选地址的数组单元的输入。(ii)应用 D_0~D_7表示输入时的地址。(iii)应用高电压，使得在毫秒级的持续时间内可以完成编程操作。(iv)应用持续时间足够长的编程脉冲，熔化数组中的期望连接，将“1”转化成“0”。(v)切断高电压。(vi)应用比前一个地址高的下一个地址。(vii)重复以上步骤(ii)~(iv)，当前情况下，在新地址上写入(转换)D_0~D_7位的逻辑状态值。(viii)继续操作，直到最后一个需要编程的数组单元地址编程完毕。

写入设备的操作是将逻辑 1 转换成逻辑 0，它是通过应用高电压和持续时间短的编程脉冲熔断连接完成操作。

图 G-1 给出了将定位程序生成的 S 记录和 hex 记录烧进 EPROM 或者 EEPROM 中的方法。首先使用紫外线擦除处理设备(例如微控制器)的 EPROM 部分。擦除操作使每个地址上的所有 8 个位都置“1”。无论何时为系统的另外一个版本而改变应用软件时，擦除设备都能使存储器重用。在计算机上执行的软件对 EPROM 编程，同时在 EPROM 和计算机的 RS232C 串口之间的接口电路的帮助下，验证刻录到 EPROM 的字节。EPROM 接口电路通过 TxD 线串行接收来自计算机中的字节，然后连同字节的地址，烧进处理设备的 EPROM。代码的刻录过程是这样的，当该电路的地址和数据可用时，它接通~V_p伏的高电压(编程电压 V_p 在一些设备中是 +12.75V，在 68HC11 中是 19V)，然后应用编程脉冲，并持续一段时间。在每个编程脉冲到来之后，在地址递增的同时，顺序地在每个地址上对该电路编程。

EPROM 接口电路通过 TxD 线串行地从计算机上接收其他字节，然后再次发送该字节，将其刻录到处理设备的适当地址上。后续地址的字节使用计算机，通过接口和 RxD 线，以验证模式接收。一些处理设备对它们的 EPROM 有自动编程模式，可以自动从 IC 中复制代码和数据。

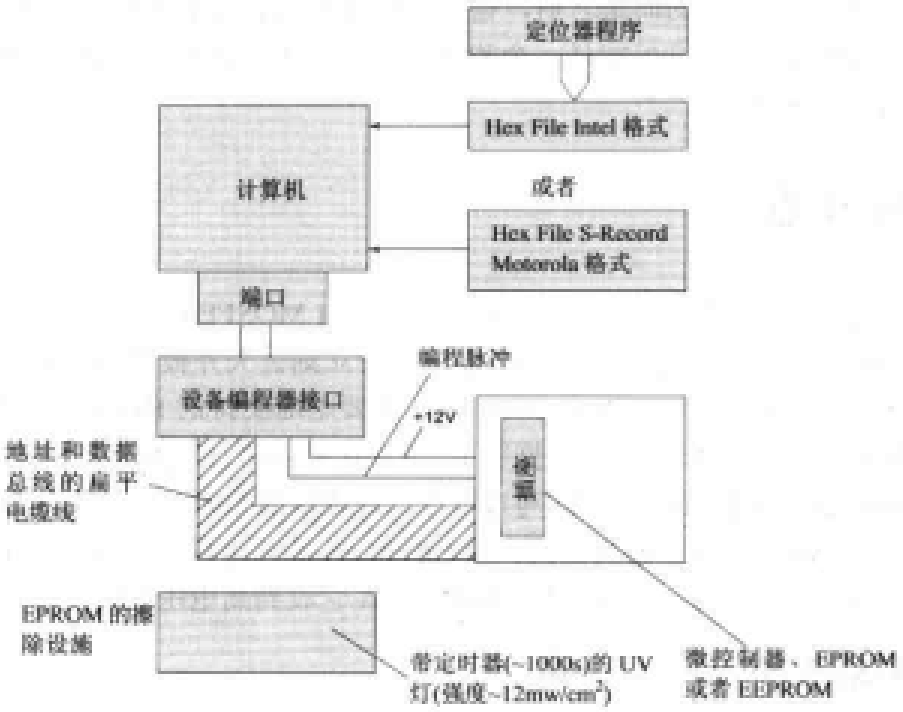


图 G-1 使用设备编程器的应用软件代码、数据和表的刻录

设备编程器根据处理和存储设备进行工作。设备编程器对 68HC11 的内部 EEPROM 编程的工作原理如下。68HC11 有一个控制寄存器 CONFIG，用于系统配置控制。它保留内部 EEPROM 等地址上的数据位。无论何时在 68HC11 内部使用编程单元，它都称作 EEPROM 寄存器。在地址 0x003B 有另一个 EEPROM 寄存器。保留该寄存器，对 CONFIG 和 68HC11 中的 EEPROM 地址(68HC11 中的片上 EEPROM 地址是从 0xB600~0B7FF)进行编程。68HC11 中刻录 EEPROM 时需要 19V 的电压。如果 CONFIG.0 位(EPROM)是“0”，则不可能通过指令进行任何擦除和写入(刻录)操作。想要刻录代码，必须首先将 CONFIG 的第 0 位置为“1”，然后加上 EEPROM 编程电压。如果第 1 位也置“1”，则 EEPROM 地址和它们的数据就会在 68HC11 内编程单元的帮助下被锁存。如果寄存器 EEPROM.3 和.4 位变成 00(分别是 0 和 0)，则在 EEPROM 地址上发生的是整体擦除(整体擦除会涉及到所有的 EEPROM)。如果写入 01，则发生字节擦除(字节擦除的意思是仅仅擦除一个 EEPROM 地址)。如果写入 10，则擦除一行 16 个字节。如果 EEPROM 中的第 2 个位置“1”，则表示仅启用擦除功能。这里的擦除是指 EEPROM 地址上的所有位置“1”。所有 3 种模式下总共的擦除时间是 10ms。如果擦除功能被禁用(这是由于第 2 位为“0”，但是由于第 0 位为“1”导致了编程电压为启动状态)，则通过执行适当地址的写指令进行字节的刻录。

计算机的 RS232C UART 端口分别通过一根线接收器和一根线驱动器，以 1200 波特的速度发送和接收数据，它分别连接到 68HC11 的 RxD 和 TxD 两个管脚。VDD、VRH、IRQ、XIRQ 管脚是+5V。VSS 和 VRL 处在“0”电平。复位电路和 8 MHz 的石英电路按普通方式连接。68HC11 在自引导模式下一次配置，它的 EEPROM 地址上的位可以使用计算机上的软件进行编程，当然这里的计算机是开发系统的一部分。当使用带 68HC11 的 8 MHz 石英电路时，外部计算机在自引导模式下以 1200 波特的速度传送。另外，首先传送所有位都是“1”(0xFF)的字节。然后计算机向 68HC11 传送 256 个字节。这些字节加载到 0x0000~0x00FF 内部 RAM 地址之间。当

68HC11 从 0xFFFE 读取了它的向量之后，自动在传输尾端设置其程序计数器。所以，它开始执行称作自引导程序的程序，这可能是一个测试程序，包括向 CONFIG 或 EPROM 寄存器和向 EEPROM 地址的写入操作。

G.3 RAM 设备

RAM 是由存储单元组成的，而这些存储单元又是由 MOSFET 组成的。对于存储器中的每个位都有一个存储单元。但是，这些数据在断电之后会立刻丢失并被释放。有两种形式的 ROM，分别是 SRAM(静态 RAM)和 DRAM(动态 RAM)。需要电池支持的 RAM 称作 NVRAM(非易失性 RAM)。RAM 可以加载大量程序和数据字节。RAM 可以进行无数次写操作。那么，如何区分向 EEPROM 的写过程和向 NVRAM 的写过程呢？答案是向 NVRAM 的写过程类似向 RAM 的写过程。区别在于 NVRAM 是非易失性的。向 EEPROM 的写过程，首先写入字节 0xFF(字节擦除)，然后再写入所需要的字节。另外，EEPROM 仅在它收到高电压的时候才进行写入操作。

G.3.1 静态和动态 RAM

在静态 RAM 中，每个存储单元总共有 4 个 MOSFET。静态的意思是一个处理器周期每次只写存储单元中的一位，该位的值保持不变，直到在某个处理器周期对它进行了修改，或者电源断开。SRAM 的优点是只要电源接通，对它的写过程都是静态的。静态存储器不需要任何刷新电路。但是它有两点不足。首先，对于特定级别的 MOSFET 通道长度，它的每个芯片上的存储密度要小大约 4 倍。这显然是由于每个存储单元上只有 4 个 MOSFET。其次，CMOS 中的操作速度，同基于相同通道长度的电路的 n-MOSFET 相比要小。

存储单元在动态 RAM(DRAM)中也可以由 n 通道的 MOSFET 门组成。当通道在逻辑状态为“1”时不传导数据，它还有门和通道电容，电流会通过电容泄漏，没有额外的电路来维持恒定电流。DRAM 的每个存储单元应该在~4ms 或者更短的时间内再次被读写，从而可以在它的任何一个存储单元中保留一个位。这样做有两个好处。一个是 n-MOS 操作比 CMOS 操作更迅速。另一个是它的 MOSFET 的数量比 SRAM 少 4 倍，存储器密度得到了很大提高。关于刷新过程还有一些疑问，这些在下面的内容中进行解释。DRAM 刷新控制器 IC 在一段编程时给定的时间内刷新每个存储单元，执行奇偶校验，并在 DRAM 字节校验失效的情况下向处理器发出奇偶错误中断。该芯片在 DRAM 中重复读写，不妨碍处理器总线的其他活动。考虑一个 RAM IC 的示例 MCM32257A-20。它是一个 256kB 的静态 RAM。有 18 个地址输入信号位 A_0~A_{17}。5110012 动态 RAM 多路复用 10 个低位和 10 个高位(D_0~D_9 和 D_{10}~D_{19} 多路复位)，并且每个存储单元必须在 4ms 内刷新。数据总线上某些特定的位的作用是访问奇偶校验位。动态刷新控制器使用这些位检查是否有位丢失。

嵌入式系统在许多应用电路上得到了很好的使用，它的存储器需求不像计算机那么大。而且，当使用低于~0.3μm 通道长度存储单元的 HCMOS 时，SRAM 速度非常快。最近，0.07μm/0.09μm 工艺的 DRAM 已经设计成功，并且会在短期内投入使用。

G.3.2 EDO RAM

你可能会问，会有 0 等待状态的 RAM 吗？当然有，0 等待状态 RAM 在高速处理器中是

非常必要的。0 等待状态是指在处理器请求位和在总线上布置位之间，没有等待状态(一个时钟周期有一个以上的状态。在上升沿和下降沿，在高电平和低电平，都有一些总线活动。状态定义为两个连续活动之间的最短时间)。有一种型号的 DRAM 称作 EDO RAM(Extended Data Out DRAM，扩展数据外部 DRAM)。当处理器读取第一个位并放到高速缓存的同时，DRAM 内部计算出下一个将读取的位，从而在第一个位传送到高速缓存后，第二个位就可以立刻变成就绪。但是，当处理器速度增加到 100 MHz 以上时，EDO RAM 的操作速度就不再继续提升。

G.3.3 SDRAM

EDO RAM 的改进型号称为 SDRAM(同步 DRAM)。它的每个存储单元以行和列的方式组织。它不是在读第一个位时计算下一个位，而是在读前一个行和列时计算下一行。读完前一行和列时，下一行就立刻就绪。在这样的存储单元组织方式下，速度就应该乘以列数。但是，当处理器速度增加到 1 GHz 以上时，EDO RAM 的操作速度就不再继续提升。

G.3.4 RDRAM

RDRAM 是 Rambus DRAM 的缩写形式。Rambus 是一家开发公司的名称。当前，使用 SDRAM 的成本很低，所以甚至当处理器工作在 1GHz 以上时，芯片组可能仍然使用 SDRAM。这是因为，它由内部高速缓存预先提供指令和数据。芯片组也可能提供两个配置选项，SDRAM 和 RDRAM(它的总线技术使用 16 位代替了 32 位或 64 位，以改善总线带宽)。32 位或 64 位的总线会降低总线带宽，总线带宽是根据从 SDRAM 中访问位的速率来确定的。RDRAM 上总线带宽得到改善(不论总线带宽是否比 16 位更窄)，是由于在 Rambus 的情况下提供了 4 个连续的取操作的补偿，而在 SDRAM 的情况下只提供了 1 个取操作。

G.3.5 参数化的分布式 RAM

系统中 RAM 的切片可以静态或者动态地分布和分配到不同地单元和子单元。一个典型系统中的分布式 RAM 可以保持 8k~256k 位。一个典型系统中可以有 1k~16k 个切片。分布式 RAM 中的行和列的大小可以伸缩和配置。例如，分布式 RAM 用作视频或者显示子系统 RAM(保存屏幕显示的像素的缓冲 RAM)时，可以静态分配，并在显示处理器和系统处理器之间共享。用作 IO 缓冲的分布式 RAM 可以在 IO 处理器和系统处理器之间共享，并且它可以被动态分配去获取各种长度的比特流或者字节流。适当伸缩可以将 RAM 分布到各个进程和 IO 设备。另外，一旦使用高速总线，它就可以定位到和系统、IO 或者其他逻辑单元很近的地方。

G.3.6 参数化的块 RAM

存储器块可以附加到嵌入式系统中特定的硬件块中(例如 MAC 操作中 16x32 乘法器单元的存储器块。MAC 是 DSP 乘法和累加操作中的术语)。一个典型的系统可以有 4~168 个块，一个块可以保存 4kB~32kB 的数据。在某些系统中，块大小也是可配置的。

G.4 微控制器中的并口

许多端口可以在存储器映射 IO 处理器中用作存储地址的提供者。内部带有处理器的芯片

上可以存在多个并口。表 G-4 给出了 3 个微控制器系列中带有片上并口的处理器的特性。

表 G-4 微控制器中带有片上并口地处理器

特 性	Intel 8051 和 Intel 8751	Motorola M68HC11E2	Intel 80196
支持外部 I/O 端口的存储器映射	有	有	有
单芯片模式[1]中的内部 I/O 端口	P0，P1，P2 和 P3	PA，PB，PC，PD 和 PE	P0，P2，P3 和 P4
扩充模式[1]中的片上端口	没有	没有	P1
端口驱动能力	漏极开路(在扩展模式下是拟方向的)	1mA 输入，10mA 输出	P0 输入端口，最大 1mA，P1-P2 拟方向，P3-P4 漏极开路(在扩展模式下是拟方向的)
握手端口	没有	带有两个独立的握手信号 STRA 和 STRB 的 PC	没有

1. 扩充模式是指处理器使用外部端口管脚作为总线。总线通向外部存储器，用作外部中断、串行 IO、定时器时间和多通道模拟输入。

G.5 串行通信设备

一个系统(或者一对系统)可以拥有主从串行设备。它非常适合系统中处理器间和处理器内的通信。主处理器、设备或者系统，同步或者异步地控制向其他几个不同的处理器、设备或者系统的输出，它们称作从处理器、设备或者系统。如果从方有一个地址，则主方可以选择将输出信号发送给从方。主方可以选择接收来自它在某个时刻选择的任何从方的输入信号。从处理器、设备或者系统控制和接收来自主处理器、设备或者系统的输入信号。这里的从方有明确的地址，它是由主方选择的。通常我们考虑两种情况，一种是同步主从通信，另一种是异步主从通信。

G.5.1 Motorola 68HC11 中的 SPI 和 SCI

Motorola 68HC11 有单独用于同步和异步通信的硬件设备，分别是 SPI(Serial Peripheral Interface，串行外围接口)和 SCI(Serial Communication Interface，串行通信接口)。

(a) SPI 提供了全双工同步通信。它的串行输入和输出都是以时钟位中的可编程速率进行的。在串行输出和输入数据的一位间隔内出现-ve 边沿和+ve 边沿的情形，在 SPI 中也是可编程的。从主方到从方的开路漏极或者标志杆输出，和通过软硬件选择主设备或者从设备也是可编程的。如果选择从管脚连接“1”，则 SPI 作为主方，如果选择从管脚连接“0”，则 SPI 作为从方。

(b) SCI 提供了全双工 UART 异步通信。波特率同上面的相同(不能单独编程)，通信模式是全双工。波特率可以有 32 种选择，用 3 个速率位和 2 个预标量位表示。接收器唤醒特性是可

编程的。如果 RWU(SCC2 的第 1 个位)置位，接收器唤醒可用，如果 RWU 复位，接收唤醒器禁用。如果 RWU 置位，则从方的接收器不会被后续的帧中断。T8 和 R8 提供了 11 位格式的处理器间(主从)UART 通信。

记住，系统中的主方和从方没有必要一定是相同处理器系列。而且，从方也可以通过执行适当的驱动软件转换成主方。

G.5.2 微控制器中的串行通信设备

表 G-5 给出了所选微控制器中，带有片上串行设备的处理器的特性。

表 G-5 微控制器汇总带有片上串口(设备)的处理器

特　性	Intel 8051 和 Intel 8751	Motorola M68HC11E2	Intel 80196
同步串口(半双工或者全双工)	半双工	全双工	半双工
异步 UART 端口(半双工或者全双工)	全双工	全双工	全双工
每字节 10 和 11 位的可编程能力	有	有	有
异步和 UART 串口的独立无复用的端口管脚	没有	有(独立的 4 个管脚)	没有
通过软件及其硬件定义的同步串口主方和从方	软件	硬件和软件	软件
通过 p 位软件编程定义的 UART 串口主方和从方	有	有	有
同步串口寄存器	SCON、SBUF 和 TL-TH 0-1	SPCR 、 SPSR 和 SPDR	SPCON SPSTAT BAUD_RATE 和 SBUF
UART 串口寄存器	SCON、SBUF 和 TL-TH 0-1(在 8052 中是定时器 2)	BAUD 、 SCC1 、 SCC2 、 SCSR 、 SCIRDR 和 SCITDR	SPCON SPSTAT BAUD_RATE 和 SBUF
使用内部定时器或者使用独立可编程波特率生成器	定时器	独立	独立和定时器

Intel 80960 和 Power PC 604 没有内部串口。

一个系统中的 8051 系列处理器带有片上通用硬件设备 USART(Universal Synchronous and Asynchronous Receiver and Transmitter，通用同步异步收发器)，它称作 SI(Serial Interface，串行接口)，其特性如下。SCON 是用于 SI 的串行控制和串行状态的特殊功能寄存器，作用是设置通信模式。SFR、SBUF 是串行缓冲器，用于同步和 UART 通信模式下指令的读写。在串行输

出数据的一位间隔内出现-ve 边沿和+ve 边沿的情形，是不可编程的。8051 SI 中的串行接收器是双缓冲。当 SBUF 仍在等待被读的同时，一个立即数寄存器保存了收到的最后一帧数据位。如果 SBUF 被读，则立即数寄存器将其值传给 SBUF。越界错误只有在 SBUF 没有读取时才发生，直到立即数寄存器得到了最后一帧的新值。SI 按以下两种模式中的一种进行操作。

(i) 半双工同步操作模式，称作模式 0。当 8051 中 12 MHz 石英连到处理器时，时钟位是 1 μ s 的时间间隔。

(ii) 全双工异步串行通信，称作模式 1 或者模式 2 或者模式 3。在模式 1 和模式 3 中，在称作 PCON 的 SFR 上使用 SMOD 位时，波特率仅对两种速率可编程。它们是 8051 中 1/64 或者 1/32 的振荡频率。T8 和 R8 提供了 11 位格式处理器间(主从)通信。

Intel 80196 带有一个称作 SI 的片上通用硬件设备。它类似一个 USART，其特性如下。以 BAUD_RATE 加载字节的可编程速率被注册两次。在串行输出数据的一位间隔内出现-ve 边沿和+ve 边沿的情形是不可编程的。8096 SI 中的串行接收器是双缓冲。SI 按以下两种模式中的一种进行操作。

(1) 半双工同步串行通信。位率依据的是定时器 T2 的时钟或者波特率寄存器。一个位的最小持续时间是 1.33ms(寄存器应该从来不对所有 0 位编程)。

(2) USART 电路(一种异步串行通信接口)的全双工 UART 模式。波特率对每个定时器 T2 时钟或者 BAUD_RATE 寄存器都是可编程的(寄存器应该不对所有 0 位编程)。它有一个 PEN 位(SPCON 的第 3 个位)，这使得 UART 11 位格式中的 P 位可以用作奇校验位，或者偶校验位，或者处理器间通信的 T8 和 R8。

G.6 微处理器中的定时器

8051 系列中，timer0 和 timer1 两个定时器是主定时器。它们的定时器模式是递减计数。8052 系列有一个辅助寄存器 timer2。在 8051 的最新型号中，还有 Watchdog 定时器 timer3。对它们编程的寄存器是地址 0x80~0xFF 之间的 8 位特殊功能寄存器(SFR)。

68HC11 有以下定时器：

(1) 主定时器 TCNT

Motorola 68HC11 中用于各种定时功能的定时器有控制、比较、捕捉、标志和中断屏蔽寄存器的地址、名称和位域。

a. TCNT 是 FRC 模式下的 16 位定时器。它的值在任何情况下都是只读寄存器 TCNT(H)和 TCNT(LO)。所有其他功能，如多输出比较、多输入捕捉和实时时钟节拍，都是使用 TCNT 实现的。

b. FRC 使用两个位的 FRC 计数输入预标量 PR1-PR0。如果是默认情况，则微控制器的复位位处于复位状态，如果是用户手动复位，则这些位都是 00，预标量系数 p 为 1。对于 8 MHz 的石英，T = 0.5μs。如果 PR1-PR0 位是 01、10 和 11，则 p 分别等于 4、8 和 16。因此向 FRC 的输入是按 PR1-PR0 位来确定的，而且是在连续时间间隔 p 之后。T = dT = 0.5μs、2μs、4μs 或者 8μs。PR1 和 PR0 是定时器中断屏蔽寄存器 TMSK2 的最低两位(第 1 位和第 0 位)。它必须在处理器复位后 64 个时钟周期内写入。

c. FRC 计数值 x 溢出。当 PR1-PR0 是 00、01、10 或 11 时，分别在(65,536)*0.5μs，(65,536)*2μs，

(65,536)×4μs 或者(65,536)×8μs 之后发生溢出。溢出时，标志(位)TOF 置位。置位的 TOF 表明有溢出，于是必须在下一次溢出之前，使用中断服务例程代码将其置位。如果初始级别 I 位是复位状态，就会执行这些代码。

d. 输入捕捉寄存器瞬间捕捉 x 的值。68HC11 中，捕捉发生在 16 位输入捕捉寄存器 TIC1、TIC2、TIC3 之中的一个。有 3 个输入管脚用于定义捕捉事件。当 3 个 PORTA 位，PA2、PA1 和 PA0 之一到达边沿时，捕捉就会发生， 3 个 PORTA 位分别是来自外部源、设备或者电路的输入。3 个 PA 位之一，是+ve 边沿还是-ve 边沿输入导致了 TIC 寄存器中 x 的捕捉，取决于定时控制寄存器 2 TCTL2 上的控制位。它是一个只写寄存器。它的 6 个位 TCTL.5、TCTL.4、TCTL.3、TCTL.2、TCTL.1、TCTL.0 分别是 edge1B、edge1A、edge2B、edge2A、edge3B、edge3A，对应 3 个 PA 位上的输入。如果这些控制位，edgeA 和 edgeB 是 00，则禁用输入捕捉，如果是 01，则允许在输入时对+ve 边沿进行输入捕捉，如果是 10，则允许对-ve 边沿进行输入捕捉，如果是 11，则允许对+ve 和-ve 边沿进行输入捕捉。用户软件可以通过读取输入捕捉寄存器来捕捉边沿出现时的事件，同时增加一个 16 位的值，并存到输出比较寄存器，在一段预定的延迟之后触发事件或者中断服务例程。

e. x 在每次递增时，同 5 个 16 位输出比较寄存器进行比较。这 5 个寄存器分别是 TOC1、TOC2、TOC3、TOC4 和 TOC5。每个比较寄存器都和一个 OCR 等价。TOC 寄存器的每个字节都有一个独立的地址。

(2) 恒定时间间隔内产生中断的实时时钟定时器

该定时器有一个称作脉冲累计控制寄存器(Pulse Accumulator Control Register，PACTL)的寄存器和两个最低位 RT1-RT0(第 1 个和第 0 个)。PACTL 是只写寄存器。如果 RT1-RT0 对的值是 00，则可能在 2^{13} 个 E 时钟脉冲后发生一次中断。如果 E 时钟脉冲频率是 2 MHz，T 是 0.5 μ s，则在每个 4.096ms 之后发生一次来自实时时钟的中断。如果 RT1-RT0 对值是 01，则可能在 2^{14} 个 E 时钟脉冲之后，也就是 8.192ms 之后发生一次中断。如果 RT1-RT0 对值是 10，则可能在 2^{15} 个 E 时钟脉冲之后，也就是 16.384ms 之后发生一次中断。如果 RT1-RT0 对值是 11，则可能在 2^{16} 个 E 时钟脉冲之后，也就是 32.768ms 之后发生一次中断。实时时钟是一个自激计数器。RT1-RT0 位控制它的预标量系数。

来自实时时钟的中断通过微控制器的 CC 寄存器中的 I 位来启用或禁用。来自实时时钟的中断也可以局部地被定时器中断屏蔽寄存器 2，TMASK2 的第 6 位 RTI 屏蔽。该位置位时，不屏蔽，复位时局部屏蔽实时中断。如果 RTI 和 I 位允许实时中断请求，则微控制器读取中断服务例程的低字节和高字节，地址从 0xFFF0(高字节)~0xFFF1(低字节)，这是实时中断的向量地址。中断服务例程必须清零(置 0)RTIF，它是实时中断的中断标志。RTIF 是定时器中断标志寄存器 2 TFLG2 中的一位。为了在从相应的服务例程返回之前，和在下次实时时钟中断发生之前启用中断，它必须清零。

(3) 脉冲累加计数器(PACT)

定时器实质上是一个计数器，它必须在恒定时间间隔内赋给输入内容。但是，计数器可以在变化时间间隔内赋给输入。PACTL 是一个 8 位脉冲累加控制寄存器。对于软件，它是一个只写寄存器。使用 PACNT 有助于计算自外部电路到达 PAI 输入的脉冲。服务例程指令读取 FRC

显示的当前时间。PACNT 加载值 q1。如果它是 q1，则同溢出时间比较，即 PAI 输入上个(256-q1)输入脉冲所花费的时间。

(4) Watchdog 定时器

有两个寄存器，CONFIG 和 COPRST。它们用于对 Watchdog 定时器的中断进行编程。CONFIG 是系统配置控制寄存器，其中有一个 NOCOP 位。它通过写地址 0x003F 进行配置操作。NOCOP 是 CONFIG 的第 2 个位。如果该位是 0，则启用 COP 功能(COP 是 computer(68HC11) operating properly 的缩写形式)。COP 功能提供了对用户程序执行时间的监督。当用户程序在例程中花费的时间比用户软件计划或者期望的时间更长时，用户规定在所要求的时间间隔内进行存储操作；在计算机复位控制寄存器 COPRST 上，首先是 0x55，然后是 0xAA。保持监督是指只要 Watchdog 定时器溢出(超时)，程序计数器就分别按照来自地址 0xFFFA 和 0xFFFB 的低字节和高字节对 16 个位进行复位。如果这 16 个位和地址 0xFFFE 及 0xFFFF 中的位相同，则微控制器在加电复位时执行指令。选项寄存器 OPTION 的第 0 位和第 1 位，在地址 0.0039 上是 CR1 和 CR0 位。如果 NOCOP 复位(置 0)，CR1-CR0　0-0，则 Watchdog 定时器在每 2^{16} 个脉冲后发生一次超时。由于对于 2MHz 的处理器 E 时钟，T = 0.5μs，所以每 16.384ms(2^{16}*0.5μs)将发生一次 WDT 超时，除非用户软件在超时之前定期存储，在计算机复位控制寄存器 COPRST 中首先是 0x55，然后是 0xAA(如果 CR1-CR0 = 0-1，则在 2^{15} 个脉冲之后，如果 CR1-CR0 = 1-0，则在 2^{14} 个脉冲之后，如果 CR1-CR0 = 1-1，则在 2^{13} 个脉冲之后)。

G.7　各种处理器系列中的中断源及其控制

表 G-6 总结了各种系列处理器中内部设备的内部可屏蔽中断，还给出了这些中断的优先级和分类。表 G-7 总结了嵌入式系统中各种系列处理器的中断控制器的系统特性，还给出了来自内部不可屏蔽源、外部可屏蔽和不可屏蔽源的中断的优先级和分类。

表 G-6　内部设备中断

处　理　器	可屏蔽中断的内部设备
8051/52 和 8751	{TI, RI} (0), TF1 (1), TF0 (3), SI (4), TF2 (5), [某些版本中有 WDTI (7)]
M68HC11E2	SCI {TI, TCI, OR, RDR, ILI, FEI, NI} (1), SPI (2), PAI (3), PAOV (4), TOF (5), OC5I (6) to OC1I (10), IC3I (11) to IC1I (13), RTI (14)
Intel 80196KC	T1OVF (0), ADConv. over (1), HSI_Received data (2), HSO_Data-Outs (3), HSI.0I (4), {SWT0 to SWT3} (5), SI_INT (6), TI (8), RI (9), HSI_FIFO 4th entry (10), T2CAP (11), T2OVF (12), HSI_FIFO full (14), WDTI (16)

P_{hw} 是小括号中的数。大括号内是相同的中断源组。68HC11 中的中断源可在复位后前 64 个时钟周期内分配最高优先级。

表 G-7　内部中断具体特性

性　　能	8051/52	Motorola M68HC11E2	Intel 80196KC	Intel 80960CA	80x86
从 ISR 转到另外一个优先级较高的可屏蔽中断源	有	没有	有	有	有
在初始化时可以声明为不可屏蔽的特殊可屏蔽中断源	没有	有	没有	没有	没有
向量优先级的所有中断源	有	有	有	有	有
所有不可屏蔽中断源的初级屏蔽位	有(IE.7)	有(CCR.4)	有(PSW.9)	有	有(标志 IF)
系列基本型号中的中断源数量	10	26	28	248	242
基本型号中中断源组的向量地址的数量	8	21	18	248(Int 8~255)	242
源的优先级别 p_{sw} 的软件范围	0-1	没有	没有	1[3]-31	没有
不同于程序计数器上下文保存	没有	有，CPU 寄存器	上下文切换[2]	上下文切换[4]	EFLAGS
对设备中断服务使用 DMA 通道	没有	没有	是	4 个	没有
不可屏蔽外部中断管脚	没有	XIRQ(NM[1])(16)	NMI(15)	NMI(31)	NMI n_{type}=2[%]
可屏蔽外部中断管脚	INT0(6)，INT1(2){8052 中有 EXF2}(5)	{IRQ，STRA} INTR	EXINT1(15)	EXINT 1(13)	XINT0-XINT7[5]
不可屏蔽内部中断	没有	CME(19)，NOCOP(18)，非法操作码(17)，SWI 指令(0)	自陷指令，非实现操作码	-	n_{type} = 0，2、4、6 和 8&

表注的含义和解释请参考本节内容。在 80x86 中，$p_{hw} = 256 - n_{type}$。

在上面两个表中，p_{hw} 是小括号中较大的一个值，代表定向到相应的 ISR_VECTADDR 并读取 ISR 的较高优先级。大括号内是中断源组。表 G-7 中表注的含义如下。对于 68HC11，NM[1]

是指仪在复位后的前 64 个时钟周期内可以声明为不可屏蔽。对于 80196KC，PTS 是指有一个外围事务服务器提供了类 DMA 的特性。表注 5 的意思是 80960 有 3 个带专用向量的管脚。5 个管脚对编码中断输入是可配置的。对于 80960 表注 3 是指 p_{sw} 为 0，并且不定义中断优先级。分配的优先级是从 1~31。对于 80x86，&是指被 0 除，如果 TF 标志置位，每条指令后会出现调试中断，n_{type}=0、2、4、6 和 8 分别表示溢出、数组越界检查、非法操作码、设备不可用和双失效。n_{type} 是一个 8 位中断向量类型(级别)。有一个仿真 Int 3 的指令 type3。上下文切换 2 是指寄存器窗口的切换导致了对新 ISR 的快速上下文引用(完全上下文切换，堆栈帧指针、程序计数器和局部寄存器组)。上下文切换 4 是指对局部寄存器的引用，和对新 ISR 的快速上下文引用时，堆栈帧的重新分配和释放。显然，寄存器组的重新分配和快速上下文切换减少了中断延时。

G.8 80x86 处理器的中断

IBM PC 使用 80x86 系列处理器。在调用 ISR 时，EFLAGS(扩展 32 标志位寄存器)和 32 位地址将压入堆栈(参见 2.2 节)。

该类处理器对于所有的中断，不论是软中断还是硬中断，都有一个分类。n_{type} 表示 ISR_VECTADDR 的一个 8 位中断向量号。n_{type} 越高，p_{hw} 就越低。ISR_VECTADDR 等于 0x00000004 乘以 n_{type}。它们有来自 244 个编程时可用的中断源的 242 个 ISR_VECTADDR。

(a) 有一个 2 字节指令 Int n_{type}，所以每个硬件中断源都有一个仿真该指令的软件指令。

(b) 有一个 1 字节指令 type3，仿真 Int 3 指令。

(c) 有一个 1 字节指令 INTO，仿真 Int 4 两字节指令和溢出异常。

(d) 当 NMI 外部管脚中断发生时，它仿真 2 字节指令 Int 2。

(e) n = 255~32，是指两字节指令 Int n 产生的软中断。n_{type} = 31~18、15 和 9 不可用。

(f) n_{type} 为 0、1、4~8 时分别表示中断、被零除、溢出、数组越界检查、非法操作码、设备不可用和双失效。

(g) n_{type} 为 10 是指异常，它使用 Jump、Call、IRET 或者 INT 指令产生一个非法调用。

(h) n_{type} 为 11、12、13、14、16 和 17 时分别是指中断、不可用段访问、堆栈失效、受保护存储器访问、页失效、浮点错误和未对齐的存储器访问。也可以用软件指令仿真这些中断。

(i) n_{type} 为 16、24 和 28 时表示定时器中断、键盘中断和 IBM PC ROM BIOS(Basic Input Output System，基本输入输出系统)中的实时时钟节拍，“如果访问不可用段”，“堆栈失效”、“受保护存储器访问”、“页失效”、“浮点错误”和“未对齐的存储器访问”。也可以用软件指令仿真这些中断。

G.9 68HC11 中的中断

G.9.1 中断服务

当中断出现时，(i)68HC11 首先执行当前指令；(ii)如果 I 位不是“0”，则它不响应任何可屏蔽中断，继续运行现有的指令集合。否则，处理器(a)将 CPU 寄存器压入存储器堆栈，并(b)

在内部暂时屏蔽 I 位“0”，直到从 ISR 中返回。这禁止了对任何后续的较低或者较高优先级的中断的响应(有一个特例)；(iii)在 21 个向量地址中读取相应的 ISR_VECTADDR。处理器在程序的每条指令末尾进行该读取操作。有 27 个中断源可以中断该程序。

G.9.2 中断源

在 68HC11 中有 27 个中断源，包括 21 个向量地址。当服务转给中断时，处理器从这些地址中读取一个 ISR_VECTADDR。中断控制器系统预先分配了 20 个优先级别，p_{hw} 从 19~0，$p_{hw}=0$ 表示优先级最低，$p_{hw}=19$ 表示优先级最高。这 27 个中断源有以下特点。

来自两个中断源的中断是不可屏蔽的，并且每个都有其向量地址。这两个中断源是非法操作码自陷()和 SWI(最低优先级)。它们没有标志。

在 68HC11 中有一个特殊类型的不可屏蔽中断，它来自管脚 XIRQ，带有一个向量地址。要使它不可屏蔽，必须在初始化后的 64 个时钟周期内启用它的不可屏蔽特性。CCR.6 上的 X 位应该置“0”，使得该中断源在后面(在加电复位后或者置位断电后)成为一个不可屏蔽中断源。

关键词及其定义

- 异步串行通信：数据字节或者帧在串行通信中不保持一致的相位差。
- 同步通信：数据字节或者帧在串行通信中保持一致的相位差。
- 波特率(Band Rate)：UART 通信期间，接收线路上串行位的速率。
- 块擦除(Bulk Erase)：存储设备中所有位都置“1”。
- 设备编程：烧进存储器、微控制器、PLA、PAL、CPLD 或者其他设备的位编程。
- 设备编程器：通过烧制对设备编程的系统或者单元。
- 刻录(Burning-in)：将设备中的某些位从“1”变到“0”的过程。
- 分布式 RAM：系统中的 RAM 切片可以静态或者动态地分布和分配到不同单元或者子单元。
- DRAM：动态 RAM，它必须不断使用称作 DRAM 刷新控制器的设备进行刷新。一次编程后，它通过扫描整个 DRAM，自动重复读写同一个位集合。
- 擦除时间(Erase Time)：设备擦除花费的时间。
- 闪存：一种存储器，进行擦除操作时，它的一个扇区或者一组扇区会同时被擦除。
- 主方(Master)：异步或者同步控制不同处理器、设备或者系统的输出的处理器、设备或者系统，这些被控处理器、设备或者系统称作从方。如果从方有一个明显的地址，则主方可以选择将输出信号发送给从方。主方可以选择接收来自它瞬间选择的任何从方的输入信号。
- 从方(Slave)：控制和接收来自主处理器、设备或者系统的输入信号的处理器、设备或者系统。从方有明确地址，并且是由主方选择的。
- 片上并口：片上的端口，它一次接收或发送 8 位或者 16 位数据。
- 片上串口：片上的端口，它一次串行接收或发送 1 位数据，速率单位是 kbps(在 UART 中是波特率)。
- 脉冲累加计数器(Pulse Accumulator Counter)：计算选定的时间间隔内输入脉冲的数量的计数器。当用作一个重复可加载的定时器时，它所起的作用是一个脉宽调制器。

附录H　嵌入式系统体系结构、编程和设计中的重要内容

以下是嵌入式系统的体系结构、编程和设计中需要了解的一些重要内容。在这里列出它们的目的是为了帮助教师制定教学大纲。

H.1　推荐使用的教学大纲

第 I 部分

单元 1：嵌入式系统介绍：定义和分类，嵌入式系统中的处理器和硬件单元概述，嵌入系统的软件，嵌入式系统示例，片上嵌入式系统(SoC)和 VLSI 设计电路的使用。

单元 2：处理器和存储器的组织结构：处理器中的结构单元，存储器设备，处理器和存储器的选择，存储器映射和分配，不同数据结构的存储块，DMA，存储器接口、处理器和 IO 设备的接口，以及粘合逻辑的使用。

单元 3：设备网络的设备和总线：I/O 设备，设备 I/O 类型和示例，来自串行设备的同步、准同步和异步通信，内部串行通信设备的示例，UART 和 HDLC，并口设备，设备/端口中复杂的接口特性，定时器和计数设备，“I^2C”、“USB”、“CAN”和高级 I/O 串行高速总线，ISA、PCI、PCI-X、cPCI 和高级总线。

单元 4：设备驱动程序和中断服务机制：设备驱动程序是使用中断和中断服务例程的设备服务，Linux 用于设备驱动程序和网络函数的内部组件，系统中的写物理设备驱动 ISR，虚拟设备的原理，设备驱动程序示例，中断服务(处理)例程，硬件和软件相关中断源，软件错误相关的硬件中断，软件指令相关中断源，上下文切换，时限，延时优先级。

单元 5：C 和 C++的编程原理和嵌入式编程：使用汇编语言(ALP)和高级语言的编程，C 程序要素，宏和函数，使用指针，NULL 指针，使用函数调用，主函数中按一定的循环顺序调用多个函数，函数指针，函数队列和中断服务例程队列指针，数据类型，使用队列实现网络协议，函数和中断的排队，使用 FIFO(先进先出)队列进行网络的流控制，数据结构，数组，堆栈，队列，树，链表和有序链表，使用活动设备驱动程序(软件定时器)的列表，使用就绪列表中的任务列表，C++嵌入式编程原理，面向对象编程，C++嵌入式编程，C 程序编译器，交叉编译器，存储器代码的优化。

单元 6：单处理器和多处理器系统软件开发过程中的程序建模原理：程序分析中数据流图的使用，程序分析中使用控制数据流图的使用，有限状态机器模型，Petri Net 模型，Petri Net 的特例 FSM，实时编程中 Petri 的使用，多处理器系统的建模：多处理器系统中的问题，同步数据流图(SDFG)模型，齐次同步数据流图(SDFG)模型，Acrylic 优先膨胀图(APEG)模型，定时 Petri Net 和扩展判定/过渡 Net 模型，多线程图(MTG)系统模型，图形的应用和多处理器系统的 Petri Net。

第 II 部分

单元 1：嵌入式软件开发过程中的软件设计实践：软件算法复杂性，开发过程生命周期(瀑布)模型，软件开发过程中线性顺序模型(瀑布模型或生命周期模型)的使用，RAD(Rapid Development Phase，快速开发阶段)模型，增量模型，并发模型，基于构件(面向对象)软件开发过程模型的使用，基于第 4 代工具的软件开发过程模型的使用，基于面向对象和基于第 4 代工具的开发方法，软件分析、设计和实现阶段，测试，验证和确认，调试，实时程序开发问题，软件项目管理，项目规格，软件维护，UML 的使用。

单元 2：进程间通信和同步：进程、任务和线程的定义，函数、ISR 和任务之间特征上的明显区别，共享数据问题，信号量的使用，优先级反转问题和死锁情形，使用信号、信号量标志或互斥作为资源键的进程通信，消息队列，邮箱，管道，虚拟(逻辑)套接字，远程过程调用(Remote Procedure Call，RPC)。

单元 3：操作系统：操作系统服务，目标，结构，内核，进程管理，存储管理，设备管理，文件系统的组织和实现，I/O 子系统，网络 OS，移动 OS，RTOS，使用 RTOS 的多任务调度管理，RTOS 中的中断例程处理。

单元 4：实时操作系统：RTOS 任务调度模型，任务调度和作为性能尺度的延时和时限的处理，协作循环调度，时间片循环调度(曲率单调协作调度)，使用抢占式调度模型策略，使用抢占式调度程序的临界段服务，固定(静态)实时任务调度，优先权分配，实时程序的性能尺度，偶发任务模型，IEEE 标准 POSIX 1003.1b 函数，抢占式调度程序的基本动作，多进程同步策略，用于设备驱动程序和嵌入式系统的嵌入式 Linux 内部组件，OS 中的安全问题。

单元 5：实时操作系统编程工具和应用示例：对测试合格和调试通过的 RTOS 的需求，应用示例，MicroC/OS-II 或者 VxWorks 或者任何其他流行的 RTOS 的研究，RTOS 系统级函数，任务服务函数，时间延迟函数，存储器分配相关函数，信号量相关函数，邮箱相关函数，队列相关函数，对 RTOS 编程的案例研究，理解案例定义，多任务及其函数，创建任务、函数和 IPC 的列表，编码步骤示例。

单元 6：嵌入式系统的软硬件协同设计和集成：嵌入式系统项目管理，开发过程中嵌入式系统设计和协同设计问题，嵌入式系统开发过程目标，行动计划，完整的说明书和系统需求，实现工具，测试，嵌入式系统的开发阶段中的设计周期，目标系统，仿真器，ICE，下载最终代码到 ROM 中的设备编程器的使用，代码生成工具(汇编器、编译器、加载器和链接器)的使用，模拟器，原型开发示例，嵌入式系统的测试和调试工具，集成开发环境(Integrated Development Environment，IDE)。

存储器和处理器敏感程序和设备驱动程序，系统设计中动态链接库的使用问题，选择合适的平台，嵌入式系统处理器的选择，需要考虑的因素和必要特性，软硬件权衡，性能建模，嵌入式平台中 OS 的移植问题。

在 ROM 中调试时，监视器、探测器、LED 测试、示波器、逻辑分析仪、位率测量仪、系统监控代码的使用。

H.2 CDAC 嵌入式系统课程教学大纲涉及的内容

CDAC(高级计算开发中心)已经设计了嵌入式系统的通用课程。下面列出了 CDAC 教学大纲模块所涉及的内容。与内容相关的章节在括号中给出。

编程原理模块：编程原理，数据结构介绍，算法复杂性，数组，堆栈，队列，树，C 语言编程回顾(第 5 章)。

面向对象编程原理模块：C/C++嵌入式编程，嵌入式系统编程(第 5 章)。

软件设计方法模块：软件开发阶段，软件生命周期模型，设计，实现和测试，软件项目管理，软件维护，统一建模语言(Unified Modeling Language，UML)实时编程问题(第 7 章)。

实时操作系统模块：OS 服务、目标和结构，进程管理，存储管理，文件系统组织和实现，I/O 子系统，网络操作系统，实时/嵌入式操作系统，实时任务模型和性能尺度，调度和资源管理，进程间通信，RTOS 环境中的中断例程，OS 安全问题，嵌入式 Linux 内部组件和移动 OS(第 8 章和第 9 章)。

嵌入式系统编程模块：使用 RTOS 的编程(第 10 章和第 11 章)。

嵌入式系统设计模块：嵌入式系统中的项目管理，嵌入式系统设计问题，协同设计问题，集成开发环境(IDE)，代码生成工具(汇编器、编译器、加载器、链接器)，仿真器，模拟器和调试器，存储器和处理器敏感程序，设备驱动程序，DLL，选择合适的平台，性能建模，嵌入式平台中 OS 的移植问题(第 12 章)。

8/16/32 位微控制器和接口模块：微控制器体系结构概述，8 位微控制器及它们的体系结构(附录 C)。

设备模块：定时器，UART，SPI，PWM，WDT，ADC 和闪存(附录 G)。

新的总线标准(Compact PCI、PCI-X、USB 2.0 等)模块：(第 3 章和附录 F)。

数字信号处理模块：数字信号处理器的体系结构，DSP 处理器和传统处理器的比较，定点运算和浮点运算的比较，嵌入式系统中的 DSP(附录 D)。

参 考 文 献

下面列出了本书中参考的书籍、网站和杂志论文，以便于读者进一步深入学习嵌入式系统的相关知识，它们覆盖了本书的所有章节。

A. 参考书目

(1) Al Williams，*Embedded Internet Design*，McGraw Hill，July 2002。

(2) Alessandro Rubini，*Linux Device Drivers*，O'Reilly，USA，June 1999。

(3) Arnold S. Berger，*Embedded Systems Design- An Introduction to Processes，Tools and Techniques*，CMP Books，Nov. 2001。

(4) B.Demuth and D.Eisenreich，*Designing Embedded Internet Devices*，Butterworth Heinemann，July 2002。

(5) Barry Kauler，*Flow Design for Embedded Systems–A simple unified Object Oriented Methodology*，CMP Books，Feb.1999

(6) Bob Zeidman，*Designing with FPGAs and CPLDs*，CMP Books，Sept.2002。

(7) Bruce Powel Douglass，*Real-Time UML – Developing Efficient Objects for the Embedded Systems*，Addison Wesley Object Technology Series，1998。

(8) Craig Hollabaugh，*Embedded Linux Hardware and Software*，Addison Wesley March 2002

(9) D.Lewis，*Fundamentals of Embedded Software：Where C and Assembly meet*，Prentice Hall，Feb.2002。

(10) Daniel Tabak，*Advanced Microprocessors*，McGraw-Hill，USA 1995。

(11) David E. Simon，*An Embedded Software Primer*，Addison Wesley Longman，Inc.，USA，(Pearson Education Asia) Singapore，USA 1999(India Reprint 2000)。

(12) Dreamtech Software Team，*Programming for Embedded Systems-Cracking the Code*，Hungry Minds，April 2002 Chapter。

(13) Ed Sutter，*Embedded System Firmware Demystified(with CD)*，CMP Books，Feb.2002。

(14) Eric Giguere，*Java 2 Micro Edition-The ultimate Guide to Programming Handheld and Embedded Devices*，John Wiley，USA，Canada 2000。

(15) F. .M. Cady，*Microcontrollers and Microcomputers—Principles of Software and Hardware Engineering*，Oxford University Press，New York，1997。

(16) F. Balarin，M. Cliodo，A. Jurecska，H. Hsieh，A. L. Lavagno，C. Paasserone，A. E. Sangiovanni-Vincentelli，E. Sentovich，K. Suzuki，and B. Tabbara，*Hardware-Software Co-Design of Embedded Systems*：A Polis Approach Norwell，MA，Kluwer Academic Publishers，June 1997。

(17) F. M. Cady，*Software and Hardware Engineering—Motorola M68HC11*，Oxford University Press，1997。

(18) Filip Thoen and Francky Catthoor，*Modeling，Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems*，Kluwer Academic Publishers 2000。

(19) Frank Vahid and Tony Givargis，*Embedded System-A unified Hardware/Software*

Introduction, John Wiley and Sons, Inc. 2002。

(20) Franz J. Rammig (Ed.), *Distributed and Parallel Embedded Systems*, Kluwer Academic Publishers, Netherlands, 1999。

(21) Fred Halsall, *Data Communication, Computer Networks and Open Systems*, 4th Edition, Pearson Education, 1996 (Fourth Indian Reprint, 2001)。

(22) G. D. Greenfield, *The 68HC11 Microcontroller*, Saunders College Publishing, 1991。

(23) G. F. Franklin, J.D. Powell and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 3rd Ed., Addison Wesley, Reading, MA, USA, 1994。

(24) Gajski, Daniel D., Frank Vahid, Sanjiv Narayan and Jie Gong, *Specification and Design of Embedded Systems*, Englewood Cliffs, NJ, Prentice Hall, 1994。

(25) Gary Nutt, *Operating Systems-A Modern Perspective*, Addison Wesley Longman, Inc., USA, 2000(Pearson Education Asia Singapore, India Reprint 2000)。

(26) George Pajari, *Unix Device Drivers*, Pearson Education, Indian Reprint, 2002。

(27) J. B. Peatman, *Design with Microcontrollers and Microcomputers*, McGraw-Hill, 1988。

(28) J. W. Stewart, *The 8051 Microcontroller—Hardware, Software and Interfacing*, Prentice Hall, 1993。

(29) Jack G. Ganssle, *Art of Programming Embedded Systems Academic USA*, 1992。

(30) Jack G. Ganssle, *Art of Programming Embedded Systems*, Butter-worth Heinemann, Newton, Mass., USA, 1999。

(31) Jack W. Crenshaw, *Math Toolkit for Real-Time Programming*, CMP Books, Aug. 2000。

(32) Jane W.S. Liu, *Real Time Systems*, Pearson Education, 2000(First Indian Reprint 2001)。

(33) Jean J Labrosse, *Embedded Systems Building Blocks*, 2nd Edition, CMP Books, Dec. 1999。

(34) Jean J. Labrosse, *MicroC/OS-II The Real Time Kernel*, R&D Books, an Imprint of Miller Freeman, Inc. Lawrence, KS 66046, USA, 1999.(Also 2nd Edition in 2002 from CMP Books)。

(35) Jeremy Bentham, *TCP/IP Lean Web Servers for Embedded Systems*, CMP Books, USA 2000.(Also 2nd Edition, 2002)。

(36) Jim Ledin, *Simulation Engineering-Build Better Embedded Systems faster*, CMP Books, Aug. 2001。

(37) John A. Stankovic, Marco Spuri, Krithi Ramamritham and Giorgio C Buttazzo, *Deadline Scheduling for Real-Time Systems–EDF and Related Algorithms*, Kluwer Academic Publishers, Netherlands, Oct. 1998。

(38) John Forrest Brown, *Embedded System Programming in C and Assembly*, Van Nostrand, Reinhold, New York, USA, 1996。

(39) John Hyde, *USB Design by Example*, John Wiley & Sons, Inc., New York, 1999。

(40) John Uffenbeck, *The 80x86 Family*, 3rd Ed., Pearson Education India, 2002。

(41) Joseph L. Weber, *Using JavaTM 2 Platform*, Que Corporation, Reprint by Prentice Hall of India, New Delhi, May 2000。

(42) Joseph Lemieux, *Programming in the OSEK/VDX Environment*, CMP Books, Oct. 2001。

(43) K. J. Hintz and Daniel Tabak, *Microcontrollers—Architecture, Implementation and*

Programming, McGraw-Hill, 1992。

(44) Kirk Zurell, *C Programming for Embedded Systems*, CMP Books, Feb. 2002。

(45) Luis Miguel Silveira, Srinivas Devadas, Ricardo A. Reis, *VLSI: Systems on a Chip*, Kluwer Academic Publishers, Dec. 1999。

(46) M. Ali Mazidi and J.G. Mazidi, *The 8051 Microcontroller and Embedded Systems*, Pearson Education, 2000, First Indian Reprint, 2002。

(47) M. Tim Jones, *TCP/IP Applications Layer Protocols for Embedded Systems*, Charles River Media, June 2002。

(48) M.C. Calcutt, F.J.Cowan, and G.H.Parchizadeh, *8051 Microcontrollers—Hardware, Software and Applications*, Arnold(and also by John Wiley), 1998。

(49) M.Costanzo, *Programmable Logic Controllers—The Industrial Computers*, Arnold(and also John Wiley)1997。

(50) Macii, Benini and Poncino, *Modern Design Technologies for Low Energy Embedded Systems*, Kluwer Academic Publishers, March 2002。

(51) Michael Barr, *Programming Embedded Systems in C and C++*, O'Reilly, USA Aug.1999 Reprinted Shroff Pubs. India Reprint August, 1999。

(52) Michael J. Pont, *Embedded C*, Addison Wesley, April 2002。

(53) Miro Samek, *Practical StateCharts in C/C++ - Quantum Programming for Embedded Systems*, CMP Books, July, 2002。

(54) Myke Predko, *Programming and Customizing the 8051 Microcontroller*, McGraw-Hill, 1999, Third Reprint, Tata McGraw-Hill, 2002。

(55) Niall Murphy, *Front Panel–Designing Software for Embedded User Interface*, CMP Books, June 1998。

(56) Peter Marwedel, and Gerl Gossens, *Code Generation for Embedded Processors*, Kluwer Academic Publishers, June, 1995。

(57) Peter Spasov, *Microcontroller Technology-The 68HC11*, 2nd Edition, Prentice Hall, Englewood Cliffs, NJ, 1996。

(58) Phillip A. Laplante, *Real-Time Systems Design and Analysis* – An Engineer's Handbook, 2nd Edition, IEE Press, USA, 1997(Prentice Hall of India, Third Indian Reprint, April, 2002)。

(59) Rainer Laeupers, *Code Optimization Techniques for Embedded Processors: Methods, Algorithms and Tools*, Kluwer Academic Publishers, Oct. 2000。

(60) Raj Kamal, *Internet and Web Technologies*, Tata McGraw-Hill, 2002。

(61) Raj Kamal, *The Concepts and Features of Microcontrollers(68HC11, 8051 and 8096) –Includes Programmable Logic Controllers*, S. Chand & Co.(Originally Wheeler Pubs.), New Delhi, 2000。

(62) Randall S. Janka, *Specification and Design Methodology for Real-Time Embedded Systems*, CMP Books, Nov. 2001。

(63) Rick Grehan, Robert Moote and Ingo Cyliax, *Real-Time Programming–A guide to 32-bit Embedded Development*, Addison Wesley, 1998。

(64) Rogers S. Pressman, *Software Engineering*, 20th Edition, McGraw-Hill, 2001。

(65) S. L. Pfleeger, *Software Engineering Theory and Practices*, Pearson Education, USA Singapore, India Reprint 2001。

(66) Scott Rixner, *Stream Processor Architecture* Kluwer Academic Publishers, Nov. 2001。

(67) Silberschatz and P.B.Galvin, *Operating Systems*, Addison Wesley, Reading, MA, USA, 1996。

(68) Sommerville, *Software Engineering*, Addison Wesley, Reading, MA, USA, 2000。

(69) Steve B. Farber, *ARM System-on-Chip Architecture*, 2nd Edition, Addison Wesley & Benjamin Cummings, 2002。

(70) Steve Heath, *Embedded System Design: Real World Design*, Butter-worth Heinemann, Newton, Mass. USA, May 2002。

(71) Steve White, *Digital Signal Processing*, Thomson Learning–Delmar, 2000(First Indian Reprint, Vikas Publishing House, 2002)。

(72) Stuart R. Ball, *Debugging Embedded Microprocessor Systems*, Butter-worth Heinemann, Newton, Mass. USA, 1998。

(73) Stuart R. Ball, *Embedded Microprocessor Systems: Real World Design*, Butter-worth Heinemann, Newton, Mass. USA, 1996.(2nd Edition, May 2002)。

(74) Sundrajan Sriram, and Survra S. Bhattacharya, *Embedded Multiprocessors-Scheduling and Synchronization*, Marcel Dekker, Inc., NewYork, USA 2000。

(75) Thomas D Burd and Robert W Brodersen, *Energy Efficient Microprocessor Design* Kluwer Academic Publishers, Oct. 2001。

(76) Tim Wilmshurst, *An Introduction to the Design of Small Scale Embedded Systems–with examples from PIC, 8051, and 68HC05/08 Microcontrollers*, Palgrave, Great Britain, 2001。

(77) Todd D. Morton, *Embedded Microcontrollers*, Prentice Hall, New Jersey USA 2001。

(78) Walter J.Grantham and Thomas L. Vincent, *Modern Control Systems-Analysis and Design*, John Wiley, 1993。

(79) Wayne Wolf, *Computers as Components-Principles of Embedded Computing System Design*, Academic Press(A Harcourt Science and Technology Company), USA, 2001。

(80) Wayne Wolf, *Modern VLSI: System on Chip Design* Pearson, Jan. 2002。

(81) William A. Shay, *Understanding Data Communications and Networks*, 2nd Edition, Thomson Learning–Brooks/Cole, 1999(First Indian Reprint, Vikas Publishing House, 2001)。

B. 网站参考

(1) http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569930058 A78D (PowerPC 750 微处理器的参考资料)。

(2) http://www.4i2i.com/h_263_philips_trimedia.htm (Philips Trimedia 处理器的参考资料)

(3) http://www.arm.com (ARM 处理器的参考资料)

(4) http://www.ami-c.org

(5) http://www.cs.ucr.edu/esd (加利福尼亚大学的计算机科学嵌入式系统设计网站)
(6) http://www.dspvillage.ti.com (得州仪器公司 DSP 处理器的参考资料)
(7) http://www.e-insite.net/edmag/ (订阅流行的嵌入式系统设计杂志)
(8) http://www.EETAsia.com (订阅流行的嵌入式系统设计杂志)
(9) http://www.ee.surrey.ac.uk/Personal/R.Young/java/html/cruise.html (11.3 节的参考资料)
(10) http://www.eembc.org (嵌入式系统性能测试参考资料)
(11) http://www.eet.com/embedsub (订阅流行的嵌入式系统设计杂志)
(12) http:// www.embedded-computing.com/eletter (订阅流行的嵌入式计算系统设计杂志)
(13) http://www.eg3.com (有关文章和教程)
(14) http://www.goembedded.com (流行嵌入式系统的新站点)
(15) http://www.home.hkstar.com/~alanchan/papers/smartCardSecurity/(11.4 节的参考资料)
(16) http://www.i2Chip.com (ASIP 芯片的参考资料)
(17) http://www.instantweb.com/~foldoc/contents.html
(18) http:// www.java.sun.com/ products/javacard (11.4 节的参考资料)
(19) http://www.linuxdoc.org (9.11 节的参考资料)
(20) http://www.mentorg.com/seamless
(21) http://www.misra.org.uk (11.3 节的参考资料)
(22) http://www.osek-vdx.org (11.3 节的参考资料)
(23) http://www.research.ibm.com/securesystems/scard.htm (11.4 节的参考资料)
(24) http://www.semiconductors.philips.com/trimedia/(附录 E.1 的参考资料)
(25) http:// www.sguthery@tiac.net
(26) http://www.ti.com/sc/docs/asic/modules/arm7.htm and arm9.htm (附录 D.4 的参考资料)
(27) http://www.ti.com/sc/docs/psheets/abstract/apps/spra638a.htm (附录 D.4 的参考资料)
(28) http://www.vsi.org
(29) http://www.webopedia.com/TERM/N/operating_system.htm (9.13 节的参考资料)
(30) http://www.wrs.com (10.3 节的参考资料)

C. 杂志论文参考

(1) Chi-Ying Liang and Huei Peng，*"Optimal Adaptive Cruise Control with Guaranteed String Stability"* Journal Vehicle System Dynamics，31，pp. 313–330，1999。
(2) Jean-Louis Brelet，*"Exploring Hardware/ Software Co-design with Vertex-II Pro FPGAs"* Xcell Journal，pp. 24–29，Summer issue，2002。
(3) Lars-Berno，Fredriksson and Kvaser，*"CAN for Critical Embedded Automotive Networks"* IEEE Micro，22(4)，28–35，2002。
(4) Tadao Murata，*"Petri Nets，Properties，Analysis and Applications"*，Proc. IEEE，77(4)，541–580)，1989。
(5) Wayne Wolf，Burak Ozer and Tiehan Lu，*"Smart Cameras as Embedded Systems"*，IEEE computer，22(5)，48–55，2002。

(5) http://www.cs.ucr.edu/esd (加利福尼亚大学的计算机科学嵌入式系统设计网站)
(6) http://www.dspvillage.ti.com (得州仪器公司 DSP 处理器的参考资料)
(7) http://www.e-insite.net/edmag/ (订阅流行的嵌入式系统设计杂志)
(8) http://www.EETAsia.com (订阅流行的嵌入式系统设计杂志)
(9) http://www.ee.surrey.ac.uk/Personal/R.Young/java/html/cruise.html (11.3 节的参考资料)
(10) http://www.eembc.org (嵌入式系统性能测试参考资料)
(11) http://www.eet.com/embedsub (订阅流行的嵌入式系统设计杂志)
(12) http:// www.embedded-computing.com/eletter (订阅流行的嵌入式计算系统设计杂志)
(13) http://www.eg3.com (有关文章和教程)
(14) http://www.goembedded.com (流行嵌入式系统的新站点)
(15) http://www.home.hkstar.com/~alanchan/papers/smartCardSecurity/(11.4 节的参考资料)
(16) http://www.i2Chip.com (ASIP 芯片的参考资料)
(17) http://www.instantweb.com/~foldoc/contents.html
(18) http:// www.java.sun.com/ products/javacard (11.4 节的参考资料)
(19) http://www.linuxdoc.org (9.11 节的参考资料)
(20) http://www.mentorg.com/seamless
(21) http://www.misra.org.uk (11.3 节的参考资料)
(22) http://www.osek-vdx.org (11.3 节的参考资料)
(23) http://www.research.ibm.com/securesystems/scard.htm (11.4 节的参考资料)
(24) http://www.semiconductors.philips.com/trimedia/(附录 E.1 的参考资料)
(25) http:// www.sguthery@tiac.net
(26) http://www.ti.com/sc/docs/asic/modules/arm7.htm and arm9.htm (附录 D.4 的参考资料)
(27) http://www.ti.com/sc/docs/psheets/abstract/apps/spra638a.htm (附录 D.4 的参考资料)
(28) http://www.vsi.org
(29) http://www.webopedia.com/TERM/N/operating_system.htm (9.13 节的参考资料)
(30) http://www.wrs.com (10.3 节的参考资料)

C. 杂志论文参考

(1) Chi-Ying Liang and Huei Peng，*"Optimal Adaptive Cruise Control with Guaranteed String Stability"* Journal Vehicle System Dynamics，31，pp. 313–330，1999。
(2) Jean-Louis Brelet，*"Exploring Hardware/ Software Co-design with Vertex-II Pro FPGAs"* Xcell Journal，pp. 24–29，Summer issue，2002。
(3) Lars-Berno，Fredriksson and Kvaser，*"CAN for Critical Embedded Automotive Networks"* IEEE Micro，22(4)，28–35，2002。
(4) Tadao Murata，*"Petri Nets，Properties，Analysis and Applications"*，Proc. IEEE，77(4)，541–580)，1989。
(5) Wayne Wolf，Burak Ozer and Tiehan Lu，*"Smart Cameras as Embedded Systems"*，IEEE computer，22(5)，48–55，2002。