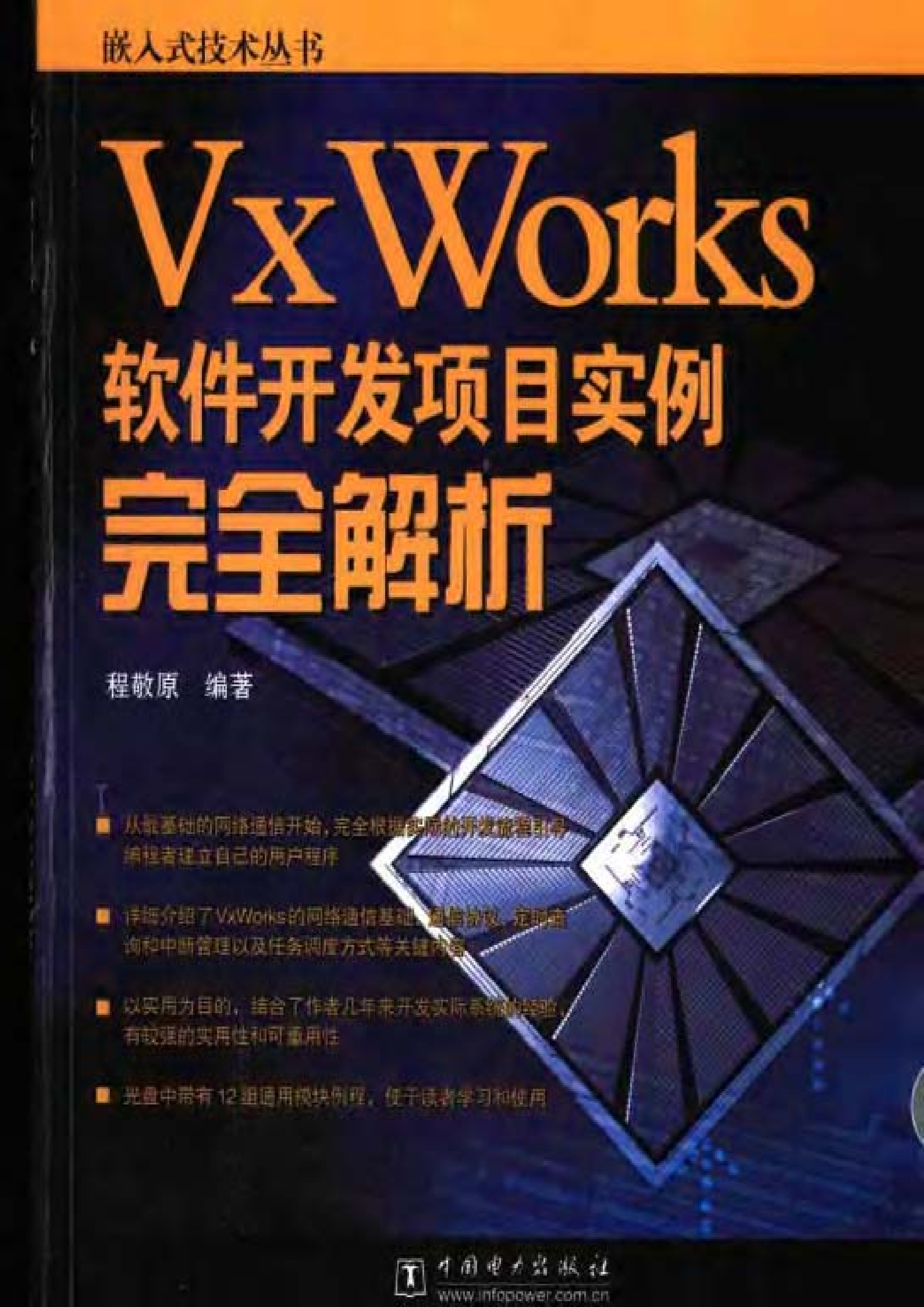
嵌入式技术丛书

VxWorks

软件开发项目实例完全解析

程敬原 编著



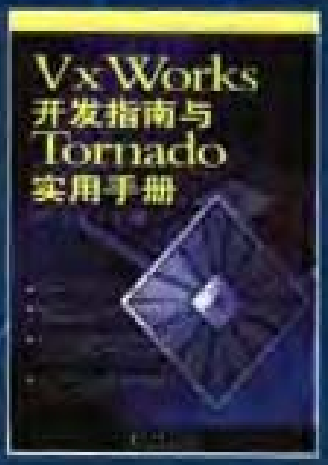
- 从最基础的网络通信开始，完全根据实际的开发流程引导编程者建立自己的用户程序
- 详细介绍了VxWorks的网络通信基础、通信协议、定时器、查询和中断管理以及任务调度方式等关键内容
- 以实用为目的，结合了作者几年来开发实际系统的经验，有较强的实用性和可重用性
- 光盘中带有12组通用模块例程，便于读者学习和使用

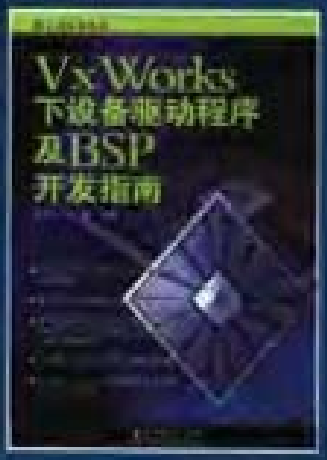
中国电力出版社
www.infopower.com.cn

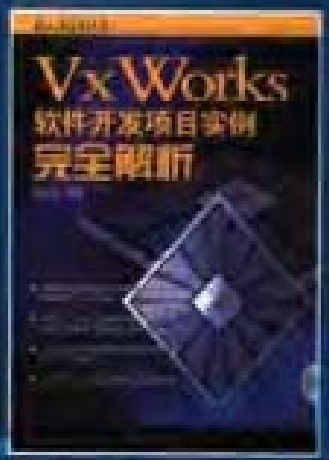
VxWorks 软件开发项目实例完全解析

嵌入式技术丛书

- C 语言嵌入式系统开发
- 嵌入式硬件设计
- 时间触发嵌入式系统设计模式：使用 8051 系列微控制器开发可靠应用
- 嵌入式 Linux
- 实时 UML —— 开发嵌入式系统高效对象
- 数字集成电路与嵌入式内核系统可测试性设计
- 嵌入式实时操作系统 VxWorks 及其开发环境 Tornado
- VxWorks 开发指南与 Tornado 实用手册
- VxWorks 下设备驱动程序及 BSP 开发指南
- 构建嵌入式 Linux 系统
- VxWorks 软件开发项目实例完全解析







ISBN 7-5083-3848-0

9 787508 338484 >

ISBN 7-5083-3848-0

定价：28.00元（附光盘）

嵌入式技术丛书

VxWorks
软件开发项目实例完全解析

程敬原　编著

中国电力出版社
www.infopower.com.cn

内容提要

本书从项目开发实践出发，力求实用，从最基础的网络通信开始，逐步展开对软硬件操作的讨论，由浅入深，并完全根据实际的软件开发流程划分章节，从而引导编程者从程序规划开始，逐步建立起一套自己的用户程序。在随书光盘中给出了12组VxWorks下的通用模块例程，以及其对应的Windows调试程序。所有程序均使用C或C++编程，方便入手，重用性强。

本书适合于初中级读者使用，特别适合于计算机相关专业在校大学生，以及从事VxWorks开发的科研设计人员使用。

图书在版编目（CIP）数据

VxWorks软件开发项目实例完全解析 / 程敬原编著. —北京：中国电力出版社，2005.9

ISBN 7-5083-3848-0

I. V... II.程... III. 实时操作系统，VxWorks—程序设计 IV. TP316.2

中国版本图书馆CIP数据核字（2005）第104133号

书　　名：VxWorks软件开发项目实例完全解析

出版发行：中国电力出版社

地　　址：北京市三里河路6号　　邮政编码：100044

电　　话：（010）68358031（总机）　　传　　真：（010）68316497，88383619

本书如有印装质量问题，我社负责退换

服务电话：（010）88515918（总机）　　传　　真：（010）88518169

E-mail：infopower@cepp.com.cn

印　　刷：北鑫印务有限公司

开本尺寸：185×233　　**印　　张**：13.25　　**字　　数**：298千字

书　　号：ISBN 7-5083-3848-0

版　　次：2005年10月北京第1版

印　　次：2005年10月第1次印刷

印　　数：0001—4000册

定　　价：28.00元

序

随着网络、通信、计算机技术的发展，嵌入式系统呈现出巨大的市场需求。基于嵌入式操作系统的开发模式成为嵌入式应用的主流开发模式。嵌入式操作系统作为嵌入式系统的基本配置与和核心技术，广泛应用于网络通信、移动计算、汽车舰船、信息安全、医疗设备以及武器装备等各个领域。目前已经涌现出各种类型的嵌入式操作系统，其中 Wind River 公司的 VxWorks 凭借其优良的性能和良好的开发环境，已成为目前市场占有率很高的产品之一。

本书根据作者几年来使用 VxWorks 开发高速数据采集系统的经验，围绕高速数据采集系统这一应用实例，介绍 VxWorks 的网络通信基础、通信协议、定时查询和中断管理以及任务调度方式等关键内容。作者以应用系统的实际开发流程为线索组织章节，内容由浅入深，读者容易入门，入手快。所选实例大多来自于作者几年来开发实际系统的积累，有较强的实用性和重用性。

本书可供从事嵌入式系统开发和应用的工程技术人员参考，也可作为本科高年级学生或研究生应用嵌入式操作系统进行课程设计的参考书。

中国科学院院士
中国科学技术大学教授、博导　陈国良

2005 年 6 月

前　言

随着计算机技术和信息技术的高速发展，计算机早已脱离了大型机和 PC 的概念，而嵌入式计算机由于其灵活多变、与硬件结合紧密、体积小、效率高等特点，得到了越来越广泛的应用。20 世纪 80 年代 Wind River 公司推出了 VxWorks，它是专门为实时嵌入式系统设计的 32 位操作系统。VxWorks 具有高性能内核和友好的用户开发界面，已被广泛应用于包括美国航天局火星探路者等的大量嵌入式设备上。

本书从项目开发实践出发，力求实用，从最基础的网络通信开始，逐步展开对软硬件操作的讨论，由浅入深，并完全根据实际的软件开发流程划分章节，从而引导编程者从程序规划开始，逐步建立起一套自己的用户程序。在随书光盘中给出了 12 组 VxWorks 下通用模块例程，以及其对应的 Windows 调试程序。所有程序均使用 C 或 C++编程，方便入手，重用性强。

全书共十章，主要内容如下：

第 1 章引导读者接近嵌入式实时操作系统 VxWorks，介绍了 VxWorks 的特点和编程规范，并列出了大量的参考资料来源。

第 2 章介绍了常用的带嵌入式小系统的采集板的软硬件设计框架，以及对应的人机控制界面 VC 程序的模块划分。

第 3 章主要介绍 VxWorks 下的网络编程。首先给出了一个用 VxWorks 作为服务器端、控制端作为客户端的例程。然后重点介绍了如何实现双缓冲队列网络通信，以减小网络通信死时间。最后介绍了软件对网络监控的方法。

第 4 章从网络通信建立通信协议的必要性入手，介绍了通信协议的基本形式，给出了命令的接收、执行和返回的基本模式。结合第 3 章的基础编程，给出网络通信的完整实现模块。

第 5 章主要介绍的是与硬件操作直接相关的方法，包括硬件轮询和中断响应。结合实践给出了一个基于 CPCI 背板的多中断联合管理程序的框架。

第 6 章介绍了软件的一体化设计方法。包括用户程序入口函数的提取，工作参数通过网络的动态配置，以及统一的任务命名和优先级管理方法。

第 7 章主要介绍将操作系统固化到板载 Flash 的方法。首先介绍了 Flash 的软件读写，然后介绍 Flash 中操作系统映像文件的动态更新方法，以及如何将用户程序和操作系统结合，最后介绍了将用户经常改动的参数存放到 Flash，实现系统下电记忆的功能。

第 8 章介绍了在 VC 下编写程序的函数接口，包括网络通信、对话框控件、目录、工具栏、状态框、绘图以及多线程创建和线程同步用信号灯。

第 9 章提出了一种脱离 Tornado，由用户程序和人机界面软件实现远程控制 VxWorks

下全局变量和函数的方法，简单便捷地实现了自启动 VxWorks 的底层控制。

第 10 章介绍了与操作系统直接相关的软件优化方法，包括基于用户保留内存的缓冲队列实现，BSP 中宏定义与用户程序的统一方法、软件大小和执行效率的综合考虑等。

感谢长期以来对我悉心指导的博导王研方教授、安琪教授和宋克柱副教授。感谢中国科大快电子学实验室（http://www.felab.ustc.edu.cn/）长期以来和我共同工作学习的杨俊峰、阮福明、陆增援、吴义华、曾翔等同学。感谢我的母亲蔡辉女士和我的祖父母孙麟、蔡世琼医师。

由于时间仓促，加之笔者水平有限，书中错误和不妥之处，在所难免，恳请读者批评指正。我的 E-mail 邮箱为：chengjy001@163.com，也欢迎共同讨论嵌入式相关的学术问题。

作　者

2005 年 5 月

目　录

第 1 章　选择嵌入式——VxWorks 入门

在目前的高速数据采集系统中，普遍采用的有个人计算机（PC）插卡和工控机箱插卡两种方式。PC 插卡虽然具有简洁方便的优点，但由于它处于 PC 中，在硬件上无法专用总线，在软件上也不满足强实时性的要求，且 PC 通常不能满足工业标准的抗振抗噪和 EMI 要求，因此越来越多的高精度高速数据采集采用了机箱插卡方式。应用在这些机箱插卡上的操作系统和应用程序属于嵌入式软件。嵌入式软件控制插卡上的嵌入式 CPU 和外围器件是完全针对本系统的数据采集来设计的，它具有强实时性、良好的多任务支持、小体积、大部分源代码公开并可裁减等优势。

目前，国内外嵌入式软件与高速数据采集相结合，正在成为新兴的经济增长点。但由于尚处于上升阶段，因而还没有一个比较规范的通用软件框架。本书试图通过构建多采集板系统的软件控制通用框架，并提供此框架下的一些通用模块，来引导嵌入式软件快速进入开发阶段，削减开发者的重复劳动，提高软硬件的工作效率。

读者通过阅读本书，将获得以下知识：

- 完整的多任务嵌入式软件框架
- 多板系统的实时中断响应和多任务管理模块
- 结合用户程序的自启动操作系统实现和在线更新模块
- 缓冲队列双工网络通信模块
- 嵌入式软件函数的远程调用模块

同时还可以获得上述各模块在 VxWorks 下的 C 语言实现。

应用该嵌入式软件框架，可以在软件代码级实现前合理规划软件结构，便于分工协作和联合调试。同时，由于绝大部分嵌入式操作系统采用标准 C 语言编写程序，因此软件模块可重用，且通用性强，大幅度缩减了数据采集插卡上的嵌入式软件开发和调试时间。经过统一规划中断响应处理机制和双缓冲队列网络通信方式，基于多任务机制、联合软件框架和全局信号的规范使用，保证了软件的实时性，并可尽量减小软件的响应时间，使其更加适用于高速数据采集系统。

1.1　VxWorks 简介

VxWorks 是专门为实时嵌入式系统设计开发的 32 位操作系统，自从 20 世纪 80 年代由 Wind River 公司推出以来，其高性能内核和友好的用户开发界面，VxWorks 并具备很多其他操作系统没有或达不到的以下七个的特点：

- 实时性强

- 支持多任务
- 体积小，可裁剪
- 支持多种 CPU
- 支持网络通信、串口通信
- 汇编＋标准 C 的编程模式，支持 C++，兼容 POSIX 标准
- 内核和定制任务可以分开编译、动态下载，支持用户自定义启动任务

下面将介绍嵌入式实时操作系统与常用的 Windows 系列操作系统的区别，以及这些区别给 VxWorks 带来的优势。

1.1.1 嵌入式系统和实时系统

嵌入式系统多指在工业系统或机电仪表设备内部为了完成特定功能而设计的计算机系统。它是软件和硬件的紧密结合体，具有软件体积小、自动化程度高、响应速度快等特点。这类系统围绕着功能、可靠性、成本、体积和功耗等要求进行设计，它们使用的操作系统和应用程序是为专用设备量身定做的，系统对这些应用程序的响应时间有严格的要求。

实时系统指的是输出时间对于系统正确运行具有至关重要作用的系统，从输入到输出的滞后时间必须足够小，且限制到一个可以接受的范围内。对这些系统来说，逻辑的正确性不仅依赖于计算结果的正确性，还取决于输出结果的时间。

实时系统根据时限对其性能的影响程度的不同，分为软实时和硬实时两种。对软实时系统来说，如果任务超过了时限范围才完成，会导致系统性能的降低；而对硬实时系统来说，就可能会造成无法预测的灾难性结果。

举例来说，人机界面控制系统一般都属于软实时系统，用户通过终端设备向系统提出请求，系统响应用户请求，完成处理后从终端回答用户。相对于用户反应时间来说，只要响应时间在秒量级，系统就是实时的。而数据采集系统一般属于硬实时系统，直接和硬件相联系，处理的一般是周期性任务或者硬件中断。作为数据采集的一个中间环节，如果响应时间超过了时限，必然会造成数据丢失或错误，因此任何任务的响应时间都绝对不能超过硬件的时限。将非实时操作系统或者软实时操作系统应用到硬实时的系统中，在短时间内可能不会产生错误，但由于其最长可能的任务响应时间可能不满足系统的要求，因此具有潜在的危险。

VxWorks 作为专门配合硬实时系统而制作的操作系统，其 wind 内核保证任务间切换时间被严格限制在毫秒量级。例如，在 68K 处理器上，切换时间仅需要 3.8μs，中断等待时间少于 3μs。只要经过一次运行参数的测试，以后任何时刻系统的状态都是可预测的。

而 Windows 系列操作系统是针对 PC 开发的，主要的输入设备是键盘和鼠标，输出设备为显示器。因此并不需要它的强实时性。当多个程序同时执行时，单个程序的响应速度往往会变慢，同时特定程序从开始执行到结束的时间不能被事先确定。相比之下，VxWorks 在要求强实时性的数据采集系统中的确占有优势。

1.1.2 多任务操作系统

嵌入式系统对其操作系统内核的要求除了强实时性，还需要具备多任务功能。由于嵌入式操作系统往往需要控制其所在电路板上的多个器件，同时还要通过串口或网络把一些信息传递给人机界面，因此，它必须具有同时处理多个事件的能力。对应于此需求，VxWorks引入了多任务机制。

每个明显单独运行的程序称为一个任务。VxWorks通过多个任务来控制和响应多重的、离散的现实世界中的事件。每个任务都可以直接或共享地访问大多数的系统资源和内存空间，并拥有自己的存放局部变量的栈和存放寄存器值、延时定时器、时间片定时器等的控制块。每个任务都拥有自己的任务名和任务ID，供内核调度时标志任务。VxWorks没有对任务个数做任何限制，只要内存足够，程序员就可以创建任意多的任务。

每一个任务都有一个任务优先级，从最高的0到最低的255。任务状态包括：正在执行（execute）、准备好（ready）、阻塞（pended）、延时（delayed）等，同时还包括这些独立状态的各种组合状态。已经准备好的任务将等待操作系统分配CPU。操作系统支持256个优先级，在默认情况下，0为最高优先级，255为最低优先级，高优先级的任务将优先获得CPU；优先级相同的任务可以采取抢占调度，即获得CPU的任务将一直执行直到任务结束或者被阻塞或延时；也可以采用轮转调度，这时将设置时间片，同优先级的ready任务将占用时间片长度的CPU，然后再回到ready状态，将CPU交给下个任务，再运行时间片的长度。

VxWorks是一个非常紧凑的系统，当启动完成后，内存中所有运行的就是3～4个系统任务，以及用户创建的任务和负责任务调度、中断响应的wind内核。这样简洁的方式使VxWorks不同于Windows系统，任务间交换信息的方式非常灵活。VxWorks提供共享内存、管道、信号量、消息队列等机制，用以实现任务间的通信与同步。对于Windows系统，单个程序内部线程之间的信息交换比较简单，但多个程序之间的信息交换就比较复杂了；而VxWorks系统中，除了优先级不同，所有任务对其他资源的使用都是平权的，丰富的通信机制和调度方式选择使VxWorks下的任务编制非常灵活，十分适合于在限定时间中处理多个事件的情况。

1.1.3 MB量级的操作系统

对于Windows系统，操作系统和应用软件的区别非常明显；而对于嵌入式操作系统，由于软件直接面对的就是硬件，因此应用软件和操作系统的分别往往不那么明显。

对于VxWorks，Wind River公司提供了封装后的内核初始化函数和很多器件的源码级驱动，这些代码和库文件统称板级支持包（Board Support Package，BSP）。这个包总共包括80多个组件，可以通过修改文件中宏的定义或在图形界面下选取两种方式进行选择和配置。此外，还可以通过对源代码进行直接修改来改动非常具体的配置，非常便于修改。

VxWorks系统是简洁高效的系统，一个同时支持网口、串口和多任务的VxWorks操作系统，文件大小只有500KB左右。将用户程序嵌入操作系统后，大小一般也不会超过2MB（在用户程序未申请MB量级静态数组的情况下），一片Flash就可以烧录全部的操作系统。

如果用户程序没有动态申请大片内存，也没有定义 MB 量级的局部变量，则操作系统运行时占用的内存一般不会超过 3MB。这远远小于 Windows 系统占用的硬盘空间和内存，大大节省了硬件资源。

表 1 所示为 VxWorks 和 Windows 系统的对照情况，其中 VxWorks 的开发完毕后模式只是 VxWorks 众多模式中的一种。本书中介绍的自启动系统制作就是基于这种模式的。

表 1　VxWorks 和 windows 操作系统对照表

	软　件	说　明	软　件	说　明
开发过程中	bootrom	由 Wind River 公司提供 BSP 编译，需要针对硬件修改，烧制到单板 ROM	BIOS	由主板生产方提供，固化到主板 ROM
	VxWorks	由 Wind River 公司提供 BSP 编译，需要针对硬件修改，通过串口或网络下载	Windows	由微软提供，安装到硬盘
	用户程序	用户编写，使用 Tornado 集成环境调试，通过串口或网络动态下载	用户开发的应用软件	使用 VC 等工具编写调试，从硬盘执行
开发完毕后	bootrom	固化到单板 ROM	BIOS	固化到主板 ROM
	VxWorks（+自启动用户程序）	烧入单板 FLASH	Windows（+自启动软件）	安装到硬盘

操作系统的制作在 BSP 基础上根据硬件略做改动即可，用户程序则需要全部由编程人员自己实现。

Windows 系统主要支持 Intel 和 AMD 为个人电脑而生产的通用型 CPU。而 VxWorks 系统支持的 CPU 则是各式各样的嵌入式 CPU。不同的 BSP 对应支持不同的 CPU，而用户程序可以在不同的 CPU 间通用。VxWorks 的支持对象包括 68k、PowerPC、CPU 32、i960、SPARC、SPARCLite、SH、ColdFire、R3000、R4000、C16X、ARM 和 MIPS 等。

它们共同的特点是功耗低、体积小、集成度高。这些嵌入式 CPU 还把很多板卡完成的任务集成在 CPU 内部，更加节省了系统的面积。例如，PowerPC860 集成了网口，PowerPC8240 集成了北桥。低功耗、小体积、功能全的 CPU，加上对 Flash 和内存的低要求，使 VxWorks 可以只花很小的代价就固化到电路板上。

同时，VxWorks 还支持 Intel 的 X86 和奔腾系列，在电路板还没有开发完毕之前，就可以在一台带网卡和显示器的 PC 机上开发调试相对与硬件关系不大的用户程序了。因此，可以做到应用软件的编写和电路板的设计同时进行，驱动程序的编写和电路板的调试同时进行，缩短了开发局期。

由于 VxWorks 使用 gcc 编译 C 语言，因此对于编程人员来说，不同的 CPU 只影响汇编部分，而 C 语言部分的程序则可以通用。操作系统和用户程序分开编译，使用固化到电路板 ROM 中的 Bootrom，就可以实现操作系统和用户程序的分别动态下载，方便调试。

1.1.4 对通信的良好支持

网络通信作为对系统灵活性、可配置性的基本要求，得到了 VxWorks 的良好支持。

在驱动方面，VxWorks 提供了很多网口的源码级驱动，如基于 PC 的 NE2000 网卡驱动和基于采集板的 Intel82559 网口芯片驱动等；在协议方面，支持到 TCP/IP 层；在软件接口方面，完全支持 BSD socket。

在以太网协议之上，VxWorks 还提供了更高一层的协议，包括：远程调用、远程文件访问、文件输出、远程命令执行等。

网络支持的协议汇总如下：

- BSD 4.4 TCP/IP
- IP、IGMP、CIDR、TCP、UDP、ARP
- RIP v.1/v.2
- 标准 Berkeley 套接口、zbufs(zero-copy socket)
- SLIP、CSLIP、PPP
- BOOTP、DNS、DHCP、TFTP
- NFS、ONC、RPC
- FTP、rlogin、rsh、telnet
- SNTP
- 具有 MIB 编译器的 WindNet SNMP v.1/v.2c（可选）
- WindNet OSPF v.2（可选）
- WindNet STREAMS SVR4（可选）

除了以太网，VxWorks 常用的通信手段还有串口，同样也提供了硬件驱动和 PPP 等通信协议，还可以将电路板的标准输入输出设置到串口，以利用超级终端在 PC 上对其进行监视和控制。

1.1.5 优秀的集成开发工具 Tornado

Wind River 公司提供了集成开发环境 Tornado，包含三个高度集成的部分：运行在宿主机和目标机上的强有力的交叉开发工具和实用程序；运行在目标机上的高性能、可裁剪的实时操作系统 VxWorks；连接宿主机和目标机的多种通信方式，如以太网、串口线、ICE 或 ROM 仿真器等。

Tornado 的独特之处在于其所有开发工具能够用于在应用开发的任意阶段以及任何档次的硬件资源上。而且，完整集的 Tornado 工具可以使开发人员完全不用考虑与目标连接的策略或目标机存储区大小。

Tornado 包括强大的开发和调试工具，尤其适用于面对大量问题的嵌入式开发人员。这些工具包括图形界面下的 C 和 C++远程源级调试器、host based shell、系统目标跟踪、内存使用分析和自动配置。另外，所有工具能够被同时运行，方便交互式开发。

所有程序编写测试完成后，可以将用户程序直接链接到操作系统中，实现自启动的用户程序。操作人员除了开关电路板电源外，不需要任何额外的操作。而是否启动完毕则可

以使用软件控制去板载 LED 以进行标志，对操作参数的修改可以通过网络，而对硬件的控制将由操作系统中内嵌的用户程序自动完成。如果用户程序支持的话，还可以从网络下载新的 VxWorks，再写到 Flash 中，以实现操作系统的在线更新。

1.2 使用 VxWorks 的预备知识

由于本书着重于实用，因此在后面的章节中，主要介绍的是编程的框架和详细的 C 语言模块。笔者假定读者已经掌握了标准 C 和 VxWorks 的相关基础知识，或者具有根据参考资料自学的能力，因而不会让这些内容占用大量篇幅。关于库函数和 VxWorks 内部通信机制的详细介绍可以从本节介绍的相关资料中获得。本章的最后两节将引导读者接触 VxWorks 的编译调试环境 Tornado，使读者可以初步搭建软件的调试装置，但同样不会在常规的软件使用方面做过多说明，读者可以参阅本节提到的资料、并通过软件的自带帮助来了解这些具体步骤。

本书中主要介绍 VxWorks5.4 嵌入式操作系统和与其配套的 Tornado2.0 开发环境，其他版本的 VxWorks 和 Tornado 可能与介绍中的略有不同。

在安装完 Tornado 之后，可以通过“Tornado 安装目录\docs\books.html”获得关于 Tornado 和 VxWorks 的在线帮助，主要包括下面的内容：

- BSP Reference
- Debugging with GDB
- GNU Make
- GNU Toolkit User's Guide
- PCMCIA for x86, Release Notes and Supplement
- Tornado API Guide
- Tornado API Reference
- Tornado Reference
- Tornado User's Guide (Windows Version)
- TrueFFS for Tornado Programmer's Guide
- VxWorks Network Programmer's Guide
- VxWorks Programmer's Guide
- VxWorks Reference Manual
- WindNet SNMPv1/v2c, Component Release Supplement
- WindNet STREAMS for Tornado, Component Release Supplement
- WindView User's Guide

在 Tornado 集成开发环境下选取 Help->Manuals Index 打开 Tornado2.0 Online Manuals，选取 Index 选项，可以直接查询函数和函数库。

上述的帮助文件都是英文的，下面推荐一些已出版的中文资料：

- 《嵌入式实时操作系统 VxWorks 及其开发环境 Tornado[M]》

中国电力出版社，孔祥营、柏桂枝著。该书主要介绍了嵌入式开发的基本概念、Tornado2.0 开发环境的使用和 VxWorks 操作系统程序设计核心技术，是 VxWorks 编程的入门级读物。

- 《VxWorks 开发指南与 Tornado 实用手册》

中国电力出版社，周启平、张杨、吴琼著。该书介绍了 VxWorks 操作系统内核技术、基于 VxWorks 操作系统的应用编程和 Tornado 开发环境的使用。

- 《VxWorks 开发人员指南丛书》

清华大学出版社，包括四本分册：《VxWorks 程序员指南》、《VxWorks 网络程序员指南》、《Tornado 用户指南》、《VxWorks BSP 开发人员指南》，王金刚等译。丛书主要由 Tornado 的在线帮助文档翻译整理而成，对于希望快速入门而英文阅读能力较低的编程人员有很大帮助。

有关标准 C 语言编程方面的基本知识可以参阅：

- 《C 程序设计》

清华大学出版社，谭浩强著。该书从最简单的 C 程序设计开始，逐步介绍了判选、循环控制、数组、函数、编译预处理、指针、结构体等 C 语言基本知识，在附录中可以方便地查询标准 C 的库函数（可能和 VxWorks 下的略有不同）和常用字符的 ASCII 码。

除此之外，网络是获得最新、最快信息的途径。下面给出一些常用网址：

- http://www.xs4all.nl/~borkhuis/vxworks/vxworks.html
  VxWorks / Tornado II FAQ（英文）。
- http://www.windriver.com/
  tornado 和 vxWorks 的开发公司主页（英文）。
- http://groups.google.com/groups?hl=zh-CN&lr=&ie=UTF-8&oe=UTF8&group=comp.os.vxworks
  goole 的 VxWorks 论坛（英文）。
- http://drew.nease.net/
  Drew 嵌入式系统开发［王东柱（Drew Wang）的个人网页］。
- http://bbs.edw.com.cn/list.asp?boardid=3
  电子产品世界——嵌入式设计论坛，此外在该网站的 http://download.eepw.com.cn/index.asp 上还可以下载到一些关于 VxWorks 的论文和英文文档。
- http://www.driverdevelop.com/forum/html_forum_14.html?1031540021
  驱动开发论坛——嵌入式系统开发。

善用搜索工具，可以解决很多开发中的问题。常用的搜索工具有：

- http://www.google.com
- http://www.yahoo.com
- http://www.baidu.com.cn
  前两个可以综合查询外文和中文，后一个在查询中文资料时可以配合使用。

1.3 VxWorks 编程规范

良好的编程规范有助于程序的模块化和清晰化。VxWorks 拥有自己的编程规范，参见《VxWorks Programmer's Guide》5.4, Edition 1, Appendices I. Coding Conventions。下面将根据笔者的经验，给出部分用户程序的编程规范。这些规范不仅仅适用于 VxWorks 下的用户程序编写，还部分适用于其他的 C 或 C++程序。

1.3.1 变量、宏定义、函数命名规范

VxWorks 下的命名应当遵循自明性，除了用作 for 循环变量的 i、j、k 等，尽量不要使用单个字母表示变量和函数，不要使用中文拼音命名。变量和函数默认以小写字母开头，用大写字母来分名称中的各个单词，如果名称太长，可以适当缩略单词中的元音字母。例如：

```
SEM_ID semFlash;                    /*Flash 保护信号灯*/
char flashIDCheck();                /*flash ID 号检查*/
char flashDataSet(int blockNum,char *pBuff,int buffLen);        /*将内存中从
pBuff 开始地址，长度为 buffLen 个字节的数据写入 flash 从 blockNum 开始的块*/
```

宏定义的所有字母大写，用下划线分开各个单词。宏定义主要分为三种类型：第一种用来表示选择和控制，通常在程序中使用#ifdef、#else 和#endif 来控制语句的编译，例如，

```
#define LOG_DEBUG_DETAILS                    /*控制是否打印调试的详细信息*/
```

第二种用来表示在程序中使用的常数，通常用于程序中的数据计算和条件判断，例如，

```
#define NET_MSG_MAX_SIZE        4096         /*网络发送数据打包的最大长度*/
```

第三种类似于函数，可以带参数，用来表示在不同位置使用的某种操作。由于宏定义在编译过程中就已经展开，因此这种调用不同于函数调用，不需要动态建立自己的堆栈，运行效率较高；但会增加程序编译后的二进制文件的长度。通常用于需要频繁调用而操作过程短的情况，例如，

```
#define FLASH2CPU_ADDR(flashAddr) (char*)((UINT)flashAddr+FLASH_BASE_ADDR)
/*Flash 地址到 CPU 地址的转换，将从 0 开始的 Flash 局部相对地址转换到以 FLASH_BASE_ADDR
开始的 CPU 绝对地址*/
```

应该给所有的全局变量和宏定义加上注释，VxWorks 下的注释使用“/*”和“*/”符号框注，可以跨行。默认设置不支持用“//”引导的单行注释。

在 VxWorks 开发环境 Tornado 下，可以正确打开并显示包含中文的文件，但无法进行中文的输入和保存。如果需要使用中文注释，应该采用第三方的代码编辑软件，例如 UltraEdit 等。推荐使用 Source Dynamics 公司出品的 Source Insight 作为代码编辑工具。此工具可以将多个文件建立成 project，用多种颜色和字体大小区分函数、变量、宏定义、关键字等，还可以在 project 中进行多个文件内容的搜索、覆盖等功能，十分适合多个程序文件的输入和综合管理，其操作界面参见图 1-1。

图 1-1 代码编辑工具 Source Insight

1.3.2 函数编程规范

函数主要包括函数说明、函数入口和参数定义、函数体三部分。函数说明应当包括完整的函数功能介绍、参数介绍、返回值介绍、调用此函数的其他函数列表，和被此函数调用的其他函数列表。函数入口即函数申明。函数体的书写中需要注意大括号的配对和对齐，并且注意语句的层次性，同一阶层上的语句对齐，这样可以保证函数的易读性。对于操作符优先级不明的情况，或者是在将其他操作系统下的 C 程序向 VxWorks 下移植时，适当地添加小括号对于保证表达式的正确性是非常重要的。

函数体最好分块，在相对独立的代码之间加上空行。视注释长度和注释对象的不同，函数体中间的注释，可以分别独立成行或加在代码后面。注释的总量大约占程序的 1/3。注释应当保持统一性，要么全部使用英文注释，要么全部使用中文，不要两者混合。

虽然编译器支持将多条语句放在一行中，但为了程序的可读性，除了 for(i=0;i<20;i++) 这样的循环控制语句，最好不要将多条语句放在一行中。同样，当单条语句太长时，应当将其断为多行，并注意对齐，例如，

```
if( (id1==FLASH_28F320C3_ID)
  && ((id2==FLASH_28F320C3_IDENT_T)||(id2==FLASH_28F320C3_IDENT_B)) )
    return(OK);
```

1.3.3 文件的编程规范

每个 C 文件应该包括四个部分：文件说明、文件包含、全局变量和函数申明、函数（由于文件较长，因此这里不给出文件实例，读者可以参阅后续章节中的具体模块实现）。

1. 文件说明

C 文件的开始应当有详细的文件说明，包括作者、版权说明、文件内容简介、外部函数列表、内部函数列表、被本文件调用的其他文件的函数列表、调用本文件中函数的其他文件的函数列表、对本文件的其他特殊说明以及修改记录（被本文件调用的其他文件的函数列表也可以采用.h 文件的格式包含，即将.c 文件的函数说明都放在.h 文件中，需要调用这些函数的文件只需要包含.h 头文件即可）。

2. 文件包含

使用#include 声明本文件需要包含的.h 文件，也可以直接包含.c 文件。.h 文件有系统库函数需要的.h 文件和用户自己定义的.h 文件两种。

在系统库函数需要的.h 文件中主要是库函数使用的宏定义、结构体定义和函数申明，在 Tornado 的在线帮助中可以查询特定库函数需要的.h 文件。例如信号灯的初始化函数 semCreate()，在 Tornado 中选择 Help->Manuals Index->Index，输入 semCreate，单击 Display（也可以双击对应选项）调出帮助界面，在 SEE ALSO 项下发现，semCreate()属于 semOLib 这个函数库，在此函数库的 INCLUDE FILES 项中就是使用 semCreate()函数的.c 文件必须包括的.h 文件。除此以外，所有 VxWorks 下的用户程序都需要包含 VxWorks 的基本库文件：

```
#include    "vxWorks.h"
```

这些.h 文件的位置在“Tornado 安装目录\target\h”下。

在用户自己定义的.h 文件中，主要是定义.c 文件中需要的宏。具体内容规范将在后面介绍。

3. 全局变量和函数申明

这里分类申明文件中需要使用的全局变量（包括单个变量、数组和结构体实例）、所有本文件中函数、本文件需要调用的外部全局变量和本文件需要调用的外部函数。

4. 函数

这部分是文件的主体，需要遵循上面的函数编程规范。注意将函数按逻辑关系排列，各个函数中间用空行隔开。

每个.h 文件分为文件说明和文件体两个部分。其中文件说明包括作者、版权说明、文件内容简介、对本文件的其他特殊说明以及修改记录。

文件体中主要是宏定义和结构体定义，应当分类分块地做详细说明。由于宏定义中包含的信息量大于 C 语句，因此应当尽量多注释。建议对每个宏定义、结构体中每个变量定义都写一句注释，并在每一类定义前加概括性的总注释。

1.3.4 经验性建议

1. 尽量避免在 C 语句中使用常数，而使用宏定义

使用宏定义方便修改，且使语句的自明性更强。在常数之间有关系的时候，使用宏定义保证程序修改时改动最小，从而避免了逐个修改常数时经常发生的遗漏情况。将与硬件相关的常数用宏定义代替，还可以加强用户程序的通用性。

2. 使用宏定义避免用户自己定义.h文件被嵌套调用

使用的宏即为.h 文件名的变化，在前面加上下划线，以便和其他宏定义区分，例如flash.h 的格式如下：

```
/*flash.h 文件说明*/
#ifndef _FLASH_H
#define _FLASH_H
/*flash.h 文件体*/
#endif  /*_FLASH_H*/
```

3. 适当加入注释

在宏定义控制语句编译的时候，如果控制的语句长度较长，最好在控制处加上注释，方便程序的阅读，例如，

```
#ifdef BOOT_FROM_FLASH
/*很长一段从 FLASH 启动 VxWorks 的代码*/
#else   /*ifndef BOOT_FROM_FLASH*/
/*很长一段用其他方式启动 VxWorks 的代码*/
#endif  /*BOOT_FROM_FLASH*/
```

4. 使用宏定义控制调试打印信息

合理使用宏定义，可以在保留代码的同时，在调试结束后快速去掉所有的打印信息，方法如下：

```
#define PRINT_DEBUG_MESSAGE                    /*定义打印调试信息*/
/*#undef PRINT_DEBUG_MESSAGE   */             /*不打印调试信息，调试完毕后去掉注释*/
#ifdef PRINT_DEBUG_MESSAGE
#define DBG_PRINTF(X)    printf(X);
#define DBG_LOGMSG(X,P1,P2,P3,P4,P5,P6) logMsg(X,P1,P2,P3,P4,P5,P6);
#else                                          /*ifndef PRINT_DEBUG_MESSAGE*/
#define DBG_PRINTF(X);
#define DBG_LOGMSG(X,P1,P2,P3,P4,P5,P6);
#endif                                         /* PRINT_DEBUG_MESSAGE*/
```

在函数中如果需要打印调试信息，则使用 DBG_PRINTF 和 DBG_LOGMSG 代替打印函数 printf()和 logMsg()。

5. 控制打印函数数量

printf()调用打印外设（对于一般的采集板来说通常是串口，因此速度大大低于采集系统的运行速度），大量使用 printf()函数会造成函数运行速度的下降，且不能在中断服务程序中使用。logMsg()在打印内容过多时，会采取丢弃打印信息的方法避免函数阻塞。两种情况都会造成不利结果，因此必须控制打印语句的使用。通常，对于一般速率上调用的函数，应该在运行的关键部分加上打印信息，而对于频繁调用的函数，最好不要在正常运行过程中打印，或者用判断语句控制打印数量；而出错情况则因为只运行一次就会导致系统

改变运行方式，因此必须打印，以保证能够迅速定位错误。

6. 合理打印，定位错误

尽量给所有的函数返回值和从硬件获取的数值加上判断，并给错误分支打印错误信息。打印语句最好能够包括所在函数的函数名信息，以便直接从打印信息中获得出错函数的位置，例如 flashInit()函数中，如果 flash 型号不对，可以打印信息如下：

```
logMsg("flashInit: flash ID check error,\nshould be 0x%08x 0x%08x",
FLASH_ID_1,FLASH_ID_2,0,0,0,0);
```

如果可能，尽量使用 logMsg()而不用 printf()，因为 logMsg()会由系统自动在打印信息前加上调用 logMsg()的任务名，以帮助定位错误。

在打印时，应当同时将对当前有影响或者有标志意义的变量打印出来，以携带尽量多的信息。

7. 注意全局变量

不要总是依赖全局变量在定义时赋给的初值，应当在函数中给每一个全局变量赋初值，并注意在哪些地方对这些全局变量进行了改变，否则使用全局变量很容易导致程序的错误。

对于 malloc()的内存，一定要用 free()配对。最好将 malloc()和 free()分别集中到一个函数内，而不要随意使用。

对于信号灯、socket 套接字等需要创建（semCreate()，socket()）的全局变量，一定要有对应的关闭函数（semDelete()，close()），以保证资源及时释放。

8. 注意数组的范围和“==”。

虽然是很基础的问题，但它们经常是意想不到的错误的原因。当全局变量被莫名其妙地改变，或者 if 语句总是只执行一个分支，就应当考虑是不是在给数组的 for 循环赋值时循环变量超出了数组范围中的一个，或者将表示判断的 if(A==B)写成了给 A 赋值并且判断恒定成立的 if(A=B)。

9. 注意左右移

由于不同操作系统在左右移操作时的补 0 或补 1 的方法不同（算术移位或者逻辑移位），可能会导致意想不到的结果，因此需要将移位操作结合位与进行。例如要将 UINT 数据左移 8 位，可以采用((value<<8)&0xFFFFFF00)来执行。由于位操作符的优先级都比较低，因此建议在所有位操作的两边都加上小括号，以保证执行顺序正确。

10. 检验入口参数

如果函数是只由用户程序自己调用，则编程者在将其入口参数赋值时，应该明确其是否有效（例如，是否是非法指针、操作长度是否超过允许范围等）。故对于纯粹的被用户程序调用的函数（通常是频繁调用的底层函数），检查入口参数的部分可以省略，以减小程序长度、提高运行速度。但如果考虑到用户在 shell 下直接输入函数名调用或者输入 sp 发起任务的情况[①]，则应该对于入口参数进行判断。特别是对于 sp，如果判断入口参数，将没有正

① 参见本书第 1.5 节中关于 shell 的描述，以及标出对应小节关于 taskSpawn()函数的描述。

确赋值的参数使用默认值，则既可以保证函数运行的安全性，又可以简化 shell 下的输入。

由于所有程序调用函数的参数都是可以确定的，因此相对安全性较高。而由手工输入的任何参数都是不确定的，因此必须经过判断。而手工输入对于 VxWorks 来说，就是 shell 下的输入；对于控制 VxWorks 的 Windows 程序来说，所有人机交互的部分都必须要特别注意。

1.3.5 版本控制

最简单的版本控制就是在程序所在文件夹外建立一个说明文件 his.txt，并在每次修改程序后在此文件中添加修改的日期和改动内容的说明，再使用 winzip 或 winrar 等压缩软件打包，压缩包的文件名定义为 XXX_年_月_日。建立一个 store 文件夹，用来存放压缩包。

应当保证压缩包中文件的可用性，即解压缩后不做任何修改就可以直接执行。对于需要配合电路板上 FPGA 或 DSP 版本才能运行的用户程序，最好能够同时备份 FPGA 源程序，或者至少是编译后下载到 FPGA 的逻辑文件。基于同样的原因，最好将 Windows 下的调试程序和控制端软件也同时备份，并保证每个压缩包内的文件能够配合使用。

良好的备份习惯可以减小由于软件修改而造成系统无法配合运作的风险，并保证在软硬件同时调试时，可时刻提供用于硬件调试的软件版本。

1.4 建立操作系统

可以使用两种方式编译出操作系统 VxWorks：命令行方式和 Tornado 下工程的方式，此两种方式都可以在 1.2 节找到。需要注意的是，这两种方式利用的文件不完全相同，本书中所有关于 VxWorks 建立的介绍都是针对命令行方式的，读者如果改用 Tornado 环境下建立工程来编译出 VxWorks，需要注意它们使用的某些文件是不同的。加载系统的详细步骤同样可以在网络上找到，这里不再赘述。

1.4.1 BSP 简介

命令行编译必须在 command 提示符下进行。在 Windows 系统下，选择“开始->运行”，对于 Win98，输入“command”，而对于 Win2000，则输入“cmd”，随后弹出 command 框，则可以输入需要的命令，如图 1-2 所示。

图 1-2 命令行方式的工作窗口

这种编译方式主要需要使用的文件为Wind River公司提供的板极支持包BSP，安装完毕后位于“Tornado安装目录\target\config”目录下，以CPU的类型分类命名。BSP是一系列文件的集合，这些文件按功能大致可分为三部分：

（1）硬件初始化文件、处理器初始化程序。

（2）操作系统初始化文件、各类头文件、驱动程序、操作系统内核初始化程序、创建多任务环境的程序。

（3）工具文件、各类 Make 文件、制作系统引导文件的工具。

BSP 的工作包括：

（1）硬件初始化，主要是 CPU 的初始化，为整个软件系统提供底层的硬件支持。

（2）为操作系统提供设备驱动程序和系统中断服务程序。

（3）初始化操作系统，为操作系统的正常运行做好准备。

1.4.2 修改 BSP

为了使通用的 BSP 适合用户的数据采集板，必须进行一些修改，主要内容如下。

1. 建立基本的制作环境

为了区分用户修改后的BSP和Wind River公司提供的原版BSP，首先创建用户自己的BSP目录。以目标板CPU的BSP为模板，在“Tornado安装目录\target\config”目录下创建用户BSP软件目录*bspname*，拷贝all目录和BSP模板下所有文件至当前目录*bspname*。

2. 修改 BSP 文件

这里就需要修改的关键点做一点说明，实际使用中编程者应当详细阅读 BSP 目录下的所有文件，并适当阅读“Tornado 安装目录\target\”目录下其他被调用的.c 和.h 文件，根据实际情况做出修改。

makefile 文件：

CPU	单板 CPU 类型
TARGET_DIR	BSP 目录名
VENDOR	单板制作公司名称
BOARD	单板名称
ROM_TEXT_ADRS boot	ROM 入口地址
ROM_SIZE	ROM 区大小
RAM_LOW_ADRS	加载 VxWorks 的地址
RAM_HIGH_ADRS	拷贝 boot ROM 映像到 RAM 中的目的地址
HEX_FLAGS	一般定义为-a $(ROM_TEXT_ADRS)
MACH_EXTRA	扩展文件，用户可加入自己的目标模块，在开发过程中设为空，应用过程中填入用户开发程序编译出的*.out

在 HEX_FLAGS 定义之后增加如下定义行：

```
CONFIG_ALL = \bspname\ALL
```

这样，系统编译 bootrom 时会自动使用用户 BSP 文件目录下 all 目录中的文件而非系

统的“Tornado 安装目录\target\config\all”目录下的文件。

bspname. h

用户应根据目标板硬件对时钟频率、内存空间分配值、串口数目等进行定义和设置。

config.h

config.h的关键部分为默认启动参数、CACHE和MMU的打开和关闭、目标板上内存地址和大小、ROM地址和大小。其中有关ROM和RAM的地址及大小必须与Makefile文件中的定义保持一致。

romInit.s

按BSP模板初始化流程，依目标板硬件对各寄存器值进行设置。

all/bootConfig.c和sysALib.s

一般不需用户修改，但也可根据需要修改。

sysSerial.c

根据用户所采用的串口及硬件连接做相应修改。

sysNet.c

根据用户所采用的网口及硬件连接做相应修改。

sysLib.c

一般只需修改网络设备初始化部分。

1.4.3 BSP 启动流程

使用经过修改后的 BSP 来编译启动引导程序 bootrom 和操作系统 VxWorks，命令分别为：

make bootrom.hex

make VxWorks

用编程器将 bootrom.hex 烧入电路板上的 ROM，上电后将引导系统启动，过程如下：

（1）目标板加电之后，程序指针指向reset 中断程序入口处，跳到ROM入口地址，执行初始化程序RomInit.s，设置机器状态字及其他硬件相关寄存器，然后禁止中断，初始化内存，并设置堆栈指针。

（2）跳到C程序bootInit.c的函数romStart()入口地址，根据堆栈中的参数决定是否清零内存RAM。根据不同的bootrom文件，把ROM中数据段和文本段拷贝到RAM（如果ROM代码是压缩的，还要解压）。

（3）程序跳到RAM入口地址（文件bootConfig.c中函数UsrInit()），使cache无效，并清零bss 段，初始化异常处理程序，进行板级硬件初始化sysHwInit()。

（4）启动多任务内核KernelInit()。

（5）初始化串口，创建console终端设备，最后根据单板设计选择不同方式加载操作系统映像文件，如网口、Flash等。

1.5 用户程序建立与调试

一般来说，在系统开发的最初阶段，操作系统映像是从网络加载的，需要使用到的工具包括串口监视工具超级终端，该软件由 Windows 自带。如果安装 Windows 98 时没有选择该组件，可以通过“控制面板→添加/删除软件→添加/删除系统组件”来选取。此外还需要用来下载文件的网络服务器。Tornado 自带有 FTP 服务器程序 wftpd32.exe，位于“Tornado 安装目录\host\x86-win32\bin\”目录下。通过在串口下修改启动参数和此程序中的参数一致，来启动 VxWorks。界面见图 1-3 和图 1-4。

图 1-3 VxWorks 加载的串口监视界面

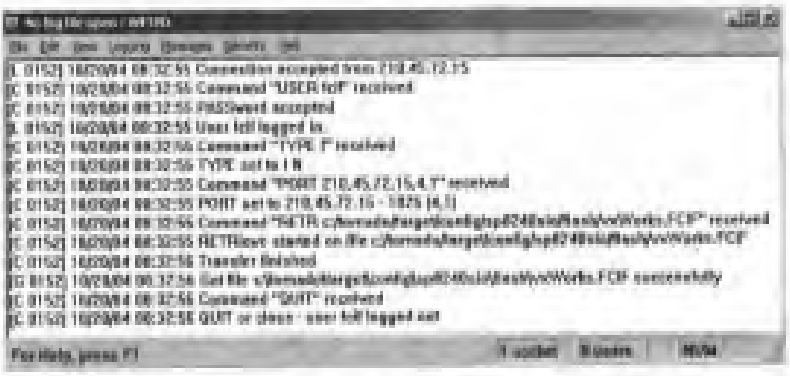


图 1-4 VxWorks 加载的 FTP 服务器界面

顺利加载完毕后，就可以通过 Tornado 的 Target Server 和 host based shell 来进行用户程序的开发和调试了。

在 Tornado 下选择“tools→Target Server→Configure”，弹出 Target Server 配置框，见图 1-5。这里不介绍具体的配置方式。

由一台 PC 控制多个目标板的情况在联合调试时经常出现，为此需要控制 Target Server，保证操作人员轻松区分各板的 Server。

首先修改 Windows 的网络标识文件 hosts，该文件位于“Win 2000 安装目录

\system32\drivers\etc”目录或“Windows 98 安装目录\system32\drivers”目录下，通过添加“IP 计算机名”的方式使 Tornado 用计算机名来代替计算机 IP（在这里，嵌入式操作系统所在的采集板也被认为是一台计算机）。例如，当有两块 IP 为 10.0.0.11 和 10.0.0.12 的板需要同时用 IP 为 10.0.0.1 的 PC 调试时，可以在 hosts 文件中添加：

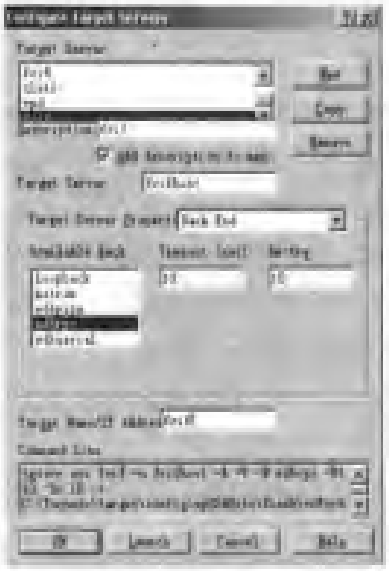


图 1-5　Target Server 配置对话框

```
10.0.0.1        bd1host      #local IP for board1 host
10.0.0.1        bd2host      #local IP for board2 host
10.0.0.11       bd1          #board1 IP
10.0.0.12       bd2          #board2 IP
```

接下来在 board1 的 Target Server 配置时，填写 Description、Target Server 和 Target Name/IP Address 为 board1、bd1host 和 bd1，对 board2 填写 board2、bd2host 和 bd2。则使用时开启 Target Server 的名称为 board1 和 board2，Tornado 工具条上供选择的 Target Server 分别为“bd1host@PC 计算机名”和“bd2host@PC 计算机名”，否则将因为 host 同为 10.0.0.1 而无法方便地区分两个 server。

启动 Target Server 后屏幕底部任务栏的右侧出现图标，双击可以弹出 Target Server Log Console 对话框，显示 Target Server 状况。正常情况见图 1-6。

图 1-6　Target Server Log Console 对话框界面

常见的错误有：

（1）网络连接或 IP 设置错误。如果网卡没有正确配置或者没有连接网线（如果采集板

和 PC 的网口直接连接，则使用对接网线；如果都接到公用 HUB，则使用普通网线。两种网线的排布方式有所不同），可以通过网口的显示灯看出。如果是 IP 错误，可以通过在 command 窗口下运行 ipconfig 察看本机 IP。

出现这种错误时，显示如下：

Wind River Systems Target Server: NT/Win95 version

Connecting to target agent...

且长期停止不动，表示一直在尝试连接 Target 而连接不上。

（2）Corefile 错误，即正在运行的 VxWorks 映像和 Target Server 中配置的"Target Server Properties→Core File and Symbols" 选项中定义的不同，显示类似下面的错误：

Warning: Target checksum: 0xef29 (computed from 0x10000 to 0x95ce8).

Host checksum: 0x2d0f (computed from 0x1930020 to 0x19b5d08).

Warning: Core file checksums do not match.

（3）Target Server 重复调用。当一个 Target Server 已经被启动，在任务栏显示图标后，再次调用此 server，则显示：

Target Server name (fcifhost@chengjy) already in use

Target Server will exit

正确配置并启动 Target Server 后，在 Tornado 工具条的下拉列表框中选择的相应 Target Server，然后点击工具条上的图标启动 host based shell，图 1-7 中给出了正确配置后启动完毕的界面图。

图 1-7　Tornado 工作界面

建立用户程序 project 和 download 到目标板的过程很简单，也不再细述了。

编程人员可以通过在 shell 下直接输入函数名，或者 "sp 函数名，函数参数 1，函数参数 2，..." 的方式启动函数，或者使用 debug->run 方式用调试模式启动用户程序。直接输

入函数名将使函数作为 shell 任务的一部分运行，可能会造成 shell 的阻塞；而用“sp 函数名”的方式相当于在程序中使用 taskSpawn 发起以“函数名”为入口的任务，不会造成 shell 的阻塞。

由于 shell 自带 C 语言解释器，因此单句的 C 语言可以直接在 shell 下执行。例如，在 shell 下输入语句再回车：

```
printf ("this is a test \n");
```

将显示如下：

```
-> printf("this is a test\n");
this is a test
value = 15 = 0xf
```

由于 VxWorks 操作系统是多任务操作系统，因此其调试概念和 Windows 下的软件调试不同，分为任务级调试和系统级调试。在进行任务级调试时，必须先进行 attach，并将调试对象设定为某个任务，这个任务的调试不会影响到其他任务的运行。如果先调试一个任务，调试到一个阶段后换调另一任务，则必须先 detach 此任务，再 attach 到另一任务。任务级调试无法“同时”调试多个任务，并且无法调试中断服务程序。系统级调试在 attach 时选择 attach system，此时可以调试中断，但由于中断服务程序中的断点会造成中断阻塞，所以一般情况下建议只是用任务级调试。

此外，VxWorks 系统可以在任意时刻进入和退出调试状态，而不是像 VisualC 那样将应用程序分为正常和调试两种方式运行，一旦程序运行就无法切换到另一状态。

除此之外，调试方法和 VisualC 等 Windows 系统编程环境基本相似，可以对任务设置断点，单步调试；也可以察看和修改全局变量和当前调试任务的局部变量，以及寄存器和内存。需要注意的是，普通的全局变量也可以在 shell 下直接查看，只需要输入变量名即可。对于全局数组，也可以在 shell 下调用函数 d()，直接查看一定长度的内存。而对于结构体，由于 shell 不辨识“.”，因此必须使用 debug 模式，选取工具栏上的 Watch 图标，在 Watch 栏中察看，见图 1-8。

图 1-8 通过 debug 模式下的 Watch 工具察看结构体

1.6 总结

本章旨在引导读者了解嵌入式实时操作系统 VxWorks。首先介绍了 VxWorks 的特点，然后给出了大量的参考资料来源。读者在进行后续章节的阅读之前，应当能凭借本章的内容和这些参考资料建立自己的用户程序调试平台。在着手编写程序之前，需要建立一个相对完整的编程规范，本章的第三节可以帮助程序员写出易读性强、模块化强、便于移植的程序，初次使用 C 语言编写大规模程序的人员，应当仔细考虑这些规范。接下来的两节中关于建立 VxWorks 和用户程序测试平台的说明只是抛砖引玉，具体的建立过程在多个参考资料中都有详细说明。

第 2 章　软件策划——模块化设计

2.1　模块化设计的目的

第 1 章给予读者关于 VxWorks 的初步认识，本章将介绍嵌入式采集板的软、硬件基本模式，并在此基础上给出软件的框架设计。

对于嵌入式采集板来说，可以在普遍意义上将硬件划分为嵌入式小系统和外围硬件两个部分。其中嵌入式小系统主要包括嵌入式 CPU、嵌入式软件运行需要的 ROM、RAM、固化嵌入式软件和存放硬件参数的 FLASH 或 NVRAM、串口和网口，主要负责对数据采集的配置和控制，以及数据的传输和存储。外围器件主要负责数据的采集，包括模数转换 ADC、数字信号处理 DSP、可编程逻辑器件 EPLD/FPGA、各种总线的接口芯片、各种通信途径的接口芯片等。常见的配置方式见图 2-1。

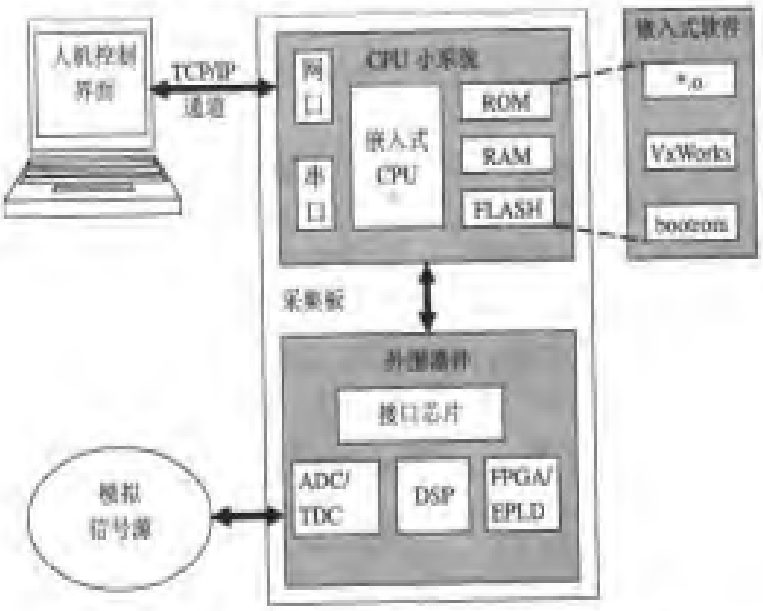


图 2-1　常见的带嵌入式控制的采集板系统

采集板的最终目标是实现模拟信号源产生信号的采集和存储。使用 ADC/TDC 将模拟信号转换为数字信号，通过 DSP 的快速处理，由 FPGA/EPLD 产生适当的时序，使数据能够通过接口芯片传到 CPU 小系统。嵌入式 CPU 可以将数据通过网口传送给人机控制界面，也可以直接将其写入硬盘或其他器件。

根据采集板硬件的模块性，可以将采集板的嵌入式软件也划分成多个部分，进行模块化设计。不同的模块分别处理软件中一个相对独立的部分，并且具有一定的通用性。

在软件代码级实现前合理规划软件结构，设计嵌入式软件框架，有助于分工协作和联合调试。同时由于绝大部分嵌入式操作系统下采用标准C语言编写程序，因此软件模块可重用、通用性强，大幅度缩减数据采集插卡上的嵌入式软件开发和调试时间。经过统一规划的中断响应处理机制和双缓冲队列网络通信方式，基于多任务机制，联合软件框架和全局信号的规范使用，保证了软件的强实时性，并尽量减小软件的响应死时间，使其更加适应于高速数据采集系统。

2.2 软件模块划分

从时间上来看，嵌入式软件可划分为Bootrom、VxWorks和用户程序三部分，见图2-2。

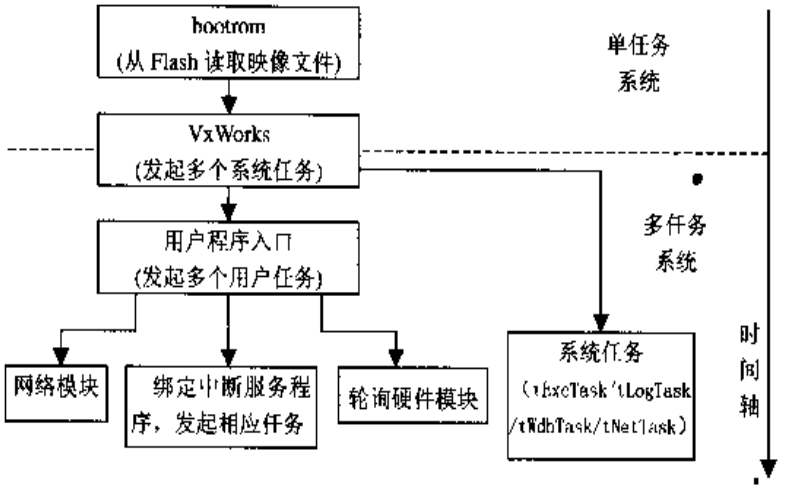


图2-2 嵌入式软件在时间上的分布

其中bootrom部分和VxWorks的前半部分仍然是单任务系统，操作系统VxWorks在结束前初始化多任务运行环境，并发起多个系统任务：

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tExcTask	excTask	2dfe638	0	PEND	8ebe0	2dfe548	0	0
tLogTask	logTask	2dfbcb0	0	PEND	8ebe0	2dfbbd0	0	0
tWdbTask	wdbCmdLoop	2d935e8	3	READY	6ddb0	2d934f0	0	0
tNetTask	netTask	2dbdc98	50	READY	6dc04	2dbdb20	0	0

用户程序使用单个函数作为入口，在此函数中发起多个任务，负责网络通信，中断服务和硬件的轮训。

从文件组织来看，嵌入式软件可分为两个部分：BSP和用户程序。BSP用于制作bootrom和VxWorks，由Wind River公司提供的BSP针对不同的CPU，提供网口和串口的底层驱动。而用户程序则负责网络在TCP/IP层之上的通信，针对不同采集板上的不同硬件的驱动和轮询，以及VxWorks映像在采集板上的固化和更新。

针对上面的特征，将嵌入式软件中可以通用的软件模块或者模块的框架分为下面几个部分：

- 网络通信和远程控制
- 硬件中断管理
- 硬件轮询
- 软件固化和更新
- 嵌入式软件用户定义函数的远程调用

其中，与硬件精密相关的中断管理和轮询囿于硬件的多样性和复杂性，只能给出框架设计，而其他三个部分则可以进行C语言的代码级实现。在不同系统间移植时只需要少量修改即可。

此外，人机控制界面也需要编写软件，且必须和嵌入式软件对应，将其也划分成几个模块：

- 网络通信
- 采集板参数控制
- 菜单、工具条和状态框
- 数据显示存储和多线程同步

本书中的第八章将提纲挈领地介绍如何用Microsoft Visual C++实现Windows操作系统下的人机界面软件，以及该软件的部分C++语言实现。

2.3 嵌入式软件各模块简介

为了满足采集板的CPU小系统和控制界面的通信、设计网络通信和远程控制模块，以保证控制界面能够灵活地控制采集板，实现参数配置和采集控制。要求采集板软件能够接收并缓存控制端的命令，实现命令的按顺序执行，保证不丢失命令，并且能够将执行结果返回。为此设计了基于缓冲队列的双工网络通信模式，结合VxWorks对多任务的良好支持，能够基本上做到无死时间的命令接收。通信双方需要能够互相理解对方传递过来的数据，因此需要统一的通信协议。此部分内容将在第三章和第四章中介绍。

硬件的管理主要分为中断和轮询两种方式。其中中断方式只在硬件完全准备好的前提下才会导致软件执行，比较节约系统资源，但是由于不能直接在中断服务程序中使用，可能造成阻塞的函数，因此受到一定的限制。轮询方式需要不停地检查硬件，且轮询速度必须大于硬件要求软件服务的速度，因此会浪费部分系统资源，但其优点是实现起来比较方便。工控机箱中多采集板的中断联合管理也是较大系统中可能会遇到的问题。此部分将在第五章中讨论。

当用户程序开发完毕后，通常需要将其和操作系统VxWorks结合在一起，实现用户程序的自启动，以保证采集板可以在上电启动后直接使用，而不需要人工操作。这样的VxWorks在制作完毕后可以采用软件方法烧录到采集板上的flash中，修改bootrom使其从flash加载映像文件。在开发过程中同样可以从flash启动VxWorks，加快系统启动速度，方便调试。此部分将在第七章中介绍。

VxWorks自启动后，所有的动态配置参数都需要从控制端输送，第八章中将介绍如何

用 Microsoft Visual C++开发 Windows 下的人机控制界面。

当所有开发都结束后，系统开始投入使用。通常此时的控制用 PC 机或工作站没有安装 VxWorks 的开发调试软件 Tornado，因此无法通过在 shell 中手动输入来调用用户程序中的函数。一旦使用中出现软件或硬件问题，需要现场定位问题时，就要求控制端能够灵活地远程调用 VxWorks 中的函数。此部分的实现将在第九章中介绍。

最后，在第十章中介绍了网络缓冲队列和网络监控算法针对 VxWorks 的优化方法，并对最后的软件大小和执行效率优化提出了一些建议。

2.4 总结

本章简要地介绍了常用的带嵌入式小系统的采集板的软、硬件设计框架，将嵌入式软件的用户程序划分成了几个模块，并将人机控制界面的 VC 程序也进行了模块划分。这些模块具有一定的通用性，具体的实现将在后续章节中介绍。

第 3 章　从通信入手——双缓冲网络通信

3.1　VxWorks 网络通信基础

网络通信是嵌入式采集板与人机界面通信方法中较简单也较常用的一种。许多嵌入式 CPU 自带网络接口，例如 Motorola 的 MPC860。在 VxWorks 的 BSP 完成了网口的底层驱动和 TCP/IP 层上的协议加载后，用户程序需要的就是在 TCP/IP 层之上进行通信。利用 VxWorks 对多任务和 BSD Socket 的良好支持，可以方便地实现网络通信。这不仅使嵌入式 CPU 与 PC 或工作站之间的网络通信成为可能，同时也丰富了系统的实时配置和调试方法，从而进一步扩展了 VxWorks 的功能和灵活性。

3.1.1　server 和 client 的概念

为了帮助理解 socket 编程中服务端（server）和客户端（client）的概念，首先来看一个生活中的例子。假定现在你能够通过提供某种服务、加工某些产品或者转发某些信息获得利益，那么首先需要做的是建立至少一个办事处，这个办事处必须具有相对固定的联系方式（如果不打算开皮包公司的话），保证需要联系你的客户总能够找到你。为了扩大声望，或许可以打些广告，之后就等着客户和你联系，当你接进客户的电话后，你派出一个联系员和他保持联系，以后客户和你这个办事处间的联系就由这个联系员全权负责了。他们互相打电话、发传真、写 e-mail，只要使用的是双方都能够明白的语言，那么客户就能够通过联系员对你的办事处发出要求，例如加工一批元件，办事处接着命令工厂加工出元件，加工完毕后又通过联系员把元件送出去，并把钱收回来。当然，如果加工过程中遇到了延误或者出现了不可挽回的事故，导致元件必须延期或者无法完成，联系员必须通知客户。完成这件业务之后，客户可以不再和你联系了，比如说他离开了地球，这样你可以告诉你的联系员不用再和他保持联系了。或者客户对你的产品非常满意，这样你们一直保持长期联系，并不停地互相打打电话，做点业务。

这个通信模式非常类似 server 和 client 的通信方式。可以将自己比作为服务端，通常就是嵌入式的采集板，它负责执行命令并返回命令执行的结果；而客户就是客户端，通常是人机控制界面的软件，它负责发布命令，监控命令的执行状况，并且可以随时不高兴就走人，并在它觉得高兴的任意时刻再回来和你联系。

在最终完成的系统中，嵌入式软件将不再需要任何手动操作，而不是像 Windows 下的人机界面，需要操作人员运行程序，并输入数据或点击按钮。因此，将采集板软件设置为 server 是非常合适的。它只需要在系统上电后自动启动 server，然后就进入等待状态了。所有其他的动作都可以等待人机界面的 client 和其建立连接后，由人机界面来控制了。

在网络通信中，首先由服务器端创建一个侦听用套接字（socket），然后与服务器的本地地址相绑定(bind)，接着进入侦听模式（listen）。这相当于开设的有固定联系方式的办事处。客户端开始时，同样也需要创建一个套接字，不同的是这个 socket 只用于通信，然后连接服务器端（connect）。这就是客户在用电话和你的办事处联系。唯一不同的是，这个客户不需要你去登广告，你们的联系地址都是事先商量好的。然后你接起了这个电话，并从来电显示上获得了客户的联系方式，于是你安排联系员用这个来电号码和客户联系。对应到网络通信就是：服务端用于侦听的套接字接受（accept）这个来自客户端的连接，然后建立新的通信用套接字并利用新建的通信套接字与客户端的进行通信。服务器端和客户端从各自的通信套接字传送和接收数据（send/recv），通信结束后再关闭相关的通信套接字（close）。你一直通过这个业务员和客户保持联系，直到所有的事情都办完。如果你的办事处只是为了这一个客户开创的，你可以连办事处（侦听用套接字）也关了，当然你也可以保留这个办公地址，保证随时可以接待客户。实现过程见图 3-1。

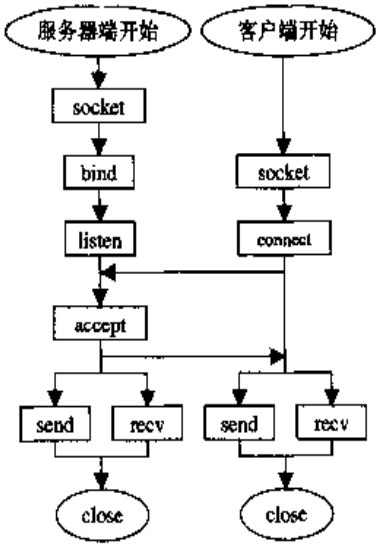


图 3-1　服务器端和客户端的建立

显然，不能因为正在执行客户的一个命令，而拒绝执行下一个命令。为此必须建立命令的缓冲机制，首先按顺序记录下客户的所有要求，再按顺序逐一执行。

通信双方必须使用双方都理解的语言，按照一定的格式和顺序进行通信，为此需要建立统一的通信协议。

在客户端非正常退出时，要求服务器端能够检测到客户端已经退出，并处理好善后工作，准备好迎接客户端的再一次连接。而客户端正常退出前，应该可以控制服务器端对应的通信用套接字的关闭。事实上，由于嵌入式采集板在开发完毕后，必须能够不经过任何手动操作，并随时接受人机界面的控制，所以本书中讨论的实现方法，是针对永远不关闭服务器端侦听套接字，并在客户端退出后，立刻关闭通信套接字，再次进入侦听状态的情况。

3.1.2　VxWorks 对 socket 通信的支持

VxWorks 完全支持 BSD socket，对应的函数库为 sockLib，必须包含的.h 文件为：types.h、mbuf.h、socket.h 和 socketvar.h，除此之外，关闭 socket 还需要使用 ioLib 中的 close()函数，必须包含的.h 文件为：ioLib.h。sockLib 中提供的库函数如表 3-1。

表 3-1　sockLib 中提供的库函数

函　数	说　明
socket()	创建套接字
bind()	将套接字与本地地址绑定

续表

函　数	说　明
listen()	开始侦听
connect()/connectWithTimeout()	连接远端的套接字
accept()	接收来自网络的连接请求并新建通信用套接字
send()/sendto()/sendmsg()	发送数据
recv()/recvfrom()/recvmsg()	接收数据
setsockopt()	设置套接字的控制属性
getsockopt()	获得套接字的控制属性
getsockname()	获得本地套接字对应的地址
getpeername()	获得远端套接字对应的地址
shutdown ()	关闭套接字的部分或全部连接能力

下面介绍在本书程序中涉及的网络函数，更详细的信息可以查阅函数库的在线帮助。

- 函数：int socket(int domain, int type, int protocol)

描述：建立一个套接字并返回一个套接字的描述符，此描述符被传递给其他的套接字程序，用来标志这个新建立的套接字。此描述符同时也是一个标准 IO 系统的描述符，可以使用 read()、write()、close()、ioctl()函数进行读写、关闭、属性控制操作。

参数：

domain，常用 AF_INET

type，　分为 SOCK_STREAM，SOCK_DGRAM 和 SOCK_RAW，分别对应基于连接（数据流）型，电报（UDP）型和最基础（raw）型的套接字，本书中使用的是 SOCK_STREAM

protocal，通常是 0

返回：创建成功返回套接字描述符，否则返回 ERROR（即 -1）。

- 函数：STATUS bind (int s, struct socket *name, int namelen)

描述：将网络地址（也被称为“名称”）绑定到一个特定的套接字，使其他套接字可以连接它。当套接字使用 socket()创建时，它只是属于某种网络地址的类型，而没有真正和网络地址绑定。

参数：

s，　套接字描述符。

*name，描述网络地址的结构体的指针。

namelen，描述网络地址的结构体的长度。

返回：成功绑定返回 OK（即 0）。如果套接字描述符是一个非法描述符，或者地址不

可用或已经被使用，或者该套接字已经被绑定，则返回 ERROR。

- 函数：STATUS listen(int s, int backlog)

描述：开始侦听，即允许对此套接字的连接。同时还规定了同一时间能够被缓冲在队列中的没有被接受的连接数。在允许连接后，实际上的连接是在 accept()时被建立的。

参数：

s,　　已经经过绑定的套接字描述符。

backlog,　允许缓冲的连接数目。

返回：开始成功返回 OK。否则如果套接字描述符为非法描述符或者无法侦听时返回 ERROR。

- 函数：STATUS connect(int s, struct sockaddr *name, int namelen)

描述：如果套接字的类型是 SOCK_STREAM，则此函数在套接字 s 和另一个地址为 name 所描述的套接字间建立虚拟回路。如果套接字类型为 SOCK_DGRAM，则此函数规定了用来发送数据的另一端。如果是 SOCK_RAW，则规定了另一端用来接受发送数据的基础套接字。

参数：

s,　　本地的套接字描述符。

*name,　规定了需要连接的另一方的网络地址的结构体指针。

namelen，结构体长度。

返回：如果正确连接返回 OK，失败则返回 ERROR。

- 函数：STATUS connectWithTimeout(int sock, struct sockaddr *adrs, int adrsLen, struct timeval *timeVal)

描述：如果*timeVal 为空指针 NULL，则此函数等同于 connect()，否则此函数尝试在*timeVal 指定的时间内建立连接。如果在规定时间没有建立连接，则函数结束，并报告超时错误。

参数：

sock,　　本地的套接字描述符。

*adrs,　　规定了需要连接的另一方的网络地址的结构体指针。

adrsLen,　　结构体长度。

*timeVal,　　指向规定连接时间的结构体的指针。

返回：如果正确连接返回 OK，失败或超时则返回 ERROR。

- 函数：int accept(int s, struct sockaddr *addr, int *addrlen)

描述：从侦听套接字接受一个连接，并且返回为此连接而新建的通信套接字的描述符。侦听套接字必须首先经过和本地地址的绑定 bind()，然后通过 listen()允许连接。accept()接受最先的连接，并按与侦听套接字 s 同样的属性创建新套接字。在建立连接前，此函数将阻塞调用它的任务，除非套接字 s 被特殊设定为非阻塞的。

函数同时将*addr 指向的结构体赋值，使其表示申请建立连接的另一端的网络地址。应当给*addrlen 赋初值，以标志结构体长度，函数返回时，此值被重新填写成*addr 指向结

构体中网络地址的长度。

参数：

s,　　侦听套接字描述符。

*addr,　指向另一端地址描述的结构体的指针。

*addrlen, 指向结构体长度的指针。

返回：通信用套接字的描述符，如果失败返回 ERROR。

- 函数：int send(int s, char *buf, int bufLen, int flags)

描述：将数据发送到已经建立的基于连接（数据流型）套接字。实际发送数据的最大长度受到 TCP 缓存大小的限制，参见 setsockopt()函数中关于 SO_SNDBUF 的讨论（TCP 的默认最大长度为 8192byte）。

参数：

s,　　将数据发送到的套接字。

*buf,　待发送数据的头指针。

bufLen,　发送数据的长度，单位 byte。

flags,　　通常为 0。

返回：实际发送的数据长度，如果发送失败返回 ERROR。

- 函数：int recv(int s, char *buf, int bufLen, int flags)

描述：从已经建立的基于连接（数据流型）的套接字中接收数据。实际接收数据的大小受到 TCP 缓存大小的限制，参见 setsockopt()函数中关于 SO_RCVBUF 的讨论（TCP 的默认最大长度为 8192byte）。

参数：

s,　　接收数据的套接字。

*buf,　接收到的数据存储位置的头指针。

bufLen,　希望接收数据的长度，单位 byte。

flags,　通常为 0。

返回：实际接收到的数据的长度，失败则返回 ERROR。

- 函数：STATUS close(int fd)

描述：关闭指定的文件，并释放文件描述符。实际调用的是器件的底层驱动。

参数：

fd，待关闭的文件描述符。

返回：低层驱动的返回值，如果文件描述符非法则返回 ERROR。

3.1.3 初始化实例

这里给出一组最简单的 VxWorks 作服务器端、PC 作客户端的配套程序。VxWorks 下的程序使用在 shell 下输入 netSendMsg（"send message"）的方式将字符数据发送到 PC 端，并将从 PC 端接收到的字符数据在标准输出设备上显示。其中标准输出 STD_OUT 可以在 BSP 中调用的 ioGlobalStdSet ()函数处找到，对于在 X86 系统上运行的 VxWorks 通常是显

示器，而对于其他系统，通常是串口。为了保证用户在 shell 下的误输入不会造成函数重入，使用了全局变量 flagNetInit 作为标志。

在光盘中还给出了配合此 VxWorks 用户程序使用的 PC 端软件的源代码和可执行文件（WinClt 目录），源代码的编译环境是 Microsoft Visual C++ 6.0。为了使其尽量简单，没有使用多线程，而是使用了 VC 提供的异步套接字 asynsock 库来编写。关于这一部分的详细讨论可以参看第八章。

首先在 shell 下输入 sp netInit 初始化服务器。然后运行 WinClt 程序，在 Server IP 和 port number 对话框中填上 VxWorks 下 server 的 IP 和端口号，点击“开始”连接 server。连接之后，在 message to send 对话框中可以填入需要发送到 VxWorks 的字符串，点击“送消息”将字符发送给 VxWorks；在 shell 下输入 netSend("hello from VxWorks"，则在 WinClt 程序的 message from VxWorks 对话框中显示同样的信息。通信结束后，在 message to send 对话框中写入 quit 并送消息，则 client 关闭，并准备好进行下一次连接；也可以直接关闭应用程序。两者都会造成 VxWorks 下 server 的退出。

执行后的界面见图 3-2。

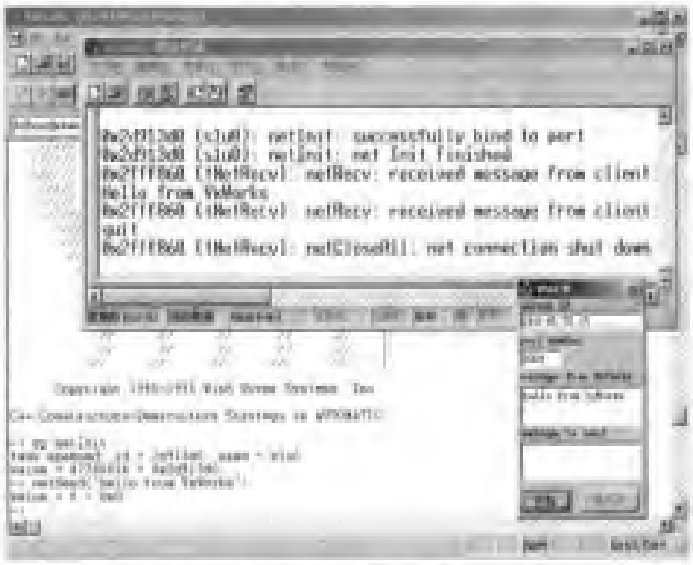


图 3-2　最基础的通信例程运行结果

VxWorks 下的程序见例 3-1[①]。

例 3-1　VxWorks 下最基础的通信例程(vxSvr.h 和 vxSvr.c)

```
/*****************************************************************************
 chengjy@felab, copyright 2002-2004
 vxSvr.h
 VxWork 下建立服务器端的需要常数定义。
```

① 本书中的所有例程都可以在光盘中同序号文件夹中找到源代码，对应的编辑工具 Source Insight 和编译工具 Tornado 的工程文件也放在同一目录下的 SI_Prj 和 Vx_Prj 中，由于工程文件中包含文件路径信息和采集板 CPU 型号信息，因此可能无法直接使用。如果 Tornado 安装在 C:\Tornado 目录下，并包含有 PPC603gnu，则将所有文件拷贝到\Tornado\target\proj\Vx_Dsgn 目录下，即可直接使用。否则请重新建立工程，并加入.c 和.h 文件。

```
 ************************************************************************/
#ifndef _VXSVR_H
#define _VXSVR_H

/*网络通信的端口*/
#define LOCAL_SERVER_PORT       2001        /*服务器端套接字的端口号*/
/*网络发送数据的最大长度*/
#define NET_MSG_MAX_SIZE        100         /*100byte*/
/*定义任务名*/
#define TNAME_RECV              "tNetRecv"  /*从网络接收数据的任务名*/
/*用户任务堆栈大小*/
#define USER_STACK_SIZE     2000
/*taskSpawn 使用的用户任务优先级*/
#define TPRI_CMDRECV 101    /*低于 shell 下 sp 任务的优先级，保证 shell 的优先响应
                    */
/*服务器初始化情况*/
#define NET_INIT_LISTENSKT      0x01
#define NET_INIT_COMMUSKT       0x02
#define NET_INIT_NULL           0x00

#endif  /*_VXSVR_H*/

/************************************************************************
 chengjy@felab, copyright 2002-2004
 vxSvr.c
 VxWorks 下建立服务器端的基本程序。
 函数:
        STATUS netInit();
        STATUS netRecv();
        STATUS netSend(char *pBuff);
        void netCloseAll();
 调用:   无
 被调用: shell 下手动调用
 说明:   编译下载完毕后，首先在 shell 下输入 sp netInit 初始化服务器端，然后通过输入
        netSend("message to be sent"),发送信息到 PC 端。通信完毕后输入 netCloseAll
        关闭所有 socket。
 ************************************************************************/
#include "VxWorks.h"
#include "taskLib.h"
#include "sockLib.h"
#include "inetLib.h"
#include "ioLib.h"
#include "logLib.h"
#include "string.h"
#include "stdio.h"
#include "netinet\\tcp.h"
```

```
#include "vxSvr.h"

int listenSkt;                          /*侦听 socket*/
int commuSkt;                           /*通信 socket*/
int flagNetInit = NET_INIT_NULL;        /*服务器初始化标志*/

STATUS netInit();
STATUS netRecv();
STATUS netSend(char *pBuff);
void netCloseAll();

/*************************************************************************
 STATUS netInit()
 函数说明:    网络服务器初始化，使用 flagNetInit 来标志初始化的进程。为了避免 accept
              阻塞任务时阻塞 shell，应当使用 sp netInit 的方法调用。
 参数:        无
 返回:        正确建立连接则返回 OK，否则返回 ERROR。
 调用:
              STATUS netRecv();
 被调用:      shell 下手动调用
 *************************************************************************/
STATUS netInit()
{
 struct  sockaddr_in  serverAddr;
 struct  sockaddr_in  clientAddr;
 int     sockAddrSize;
 int     i;
 char    optval = 1;

 if(flagNetInit==NET_INIT_COMMUSKT)
 {
     /*如果已经初始化了，不再重复进行*/
     logMsg("netInit: server has already been initialized\n",0,0,0,0,0,0);
     return(OK);
 }
 else if(flagNetInit== NET_INIT_NULL)
 {
     /*如果服务器侦听 socket 没有初始化，创建基于 TCP 的侦听 socket ，
     否则直接跳过侦听 socket 创立，进入准备接收连接的步骤*/
     if((listenSkt = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
     {
         logMsg("netInit: can not open listen socket\n",0,0,0,0,0,0);
        return(ERROR);
     }
```

```
        /*建立本地地址*/
        sockAddrSize = sizeof (struct sockaddr_in);
        bzero ((char *) &serverAddr, sockAddrSize);
        serverAddr.sin_family = AF_INET;
        serverAddr.sin_len = (u_char) sockAddrSize;
        serverAddr.sin_port = htons (LOCAL_SERVER_PORT);
        serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);

        /*绑定 socket 到本地地址*/
        if (bind (listenSkt, (struct sockaddr *) &serverAddr, sockAddrSize) ==
ERROR)
        {
            /*绑定出错，关闭侦听 socket*/
            logMsg("netInit:unable to bind to port %d\n",LOCAL_SERVER_PORT,0,0,
   0,0,0);
            close(listenSkt);
            return(ERROR);
        }

        /*绑定成功*/
        logMsg("netInit: successfully bind to port\n",LOCAL_SERVER_PORT,0,0,0,
   0,0);

        /*开始侦听*/
        if (listen (listenSkt, 1) == ERROR)
        {
            /*无法侦听，关闭侦听 socket*/
            logMsg("netInit: can not listen to listen socket\n",0,0,0,0,0,0);
            close (listenSkt);
            return(ERROR);
        }
        else
        {
            /*侦听 socket 建立完毕，设置全局标志*/
            flagNetInit = NET_INIT_LISTENSKT;
        }
    }

    /*接收外部连接，建立通信 socket*/
    commuSkt = accept(listenSkt, (struct sockaddr*)(&clientAddr),
&sockAddrSize);

    if(commuSkt==ERROR)
    {
        /*如果通信 socket 创建失败，则关闭侦听 socket*/
        logMsg("netInit: can not accept command socket\n",0,0,0,0,0,0);
```

```
        close (listenSkt);
        return(ERROR);
    }
    else
    {
        /*建立了正确的通信通道*/
        logMsg("netInit: net Init finished\n",0,0,0,0,0,0);

        /*所有的数据都必须立刻发送*/
        setsockopt (commuSkt, IPPROTO_TCP, TCP_NODELAY, &optval, sizeof
(optval));

        /*发起网络命令循环接收任务、命令处理和状态发送任务*/
        if(taskNameToId(TNAME_RECV)==ERROR)
        {
            taskSpawn(TNAME_RECV,TPRI_CMDRECV,0,USER_STACK_SIZE,
                    (FUNCPTR)netRecv,0,0,0,0,0,0,0,0,0,0);
        }
    }
    /*设置网络初始化完成*/
    flagNetInit = NET_INIT_COMMUSKT;
    return(OK);
    }

    /*************************************************************************
     STATUS netSend(char *pBuff)
     函数说明:    从已经建立的网络通道发送数据。
     参数:        pBuff，指向需要发送字符串首地址的指针。
     返回:        成功发送返回OK，否则返回ERROR。
     调用:        void netCloseAll();
     被调用:      shell下直接调用
     *************************************************************************/
    STATUS netSend(char *pBuff)
    {
     int buffLen;
     if((pBuff!=NULL)&&(flagNetInit==NET_INIT_COMMUSKT))
     {
         /*如果数据太长，则截断*/
         buffLen = strlen(pBuff);
         if(buffLen>=NET_MSG_MAX_SIZE)
         {
             buffLen = NET_MSG_MAX_SIZE;
             pBuff[buffLen-1]=0;
         }

         /*发送数据*/
```

```
    if(send(commuSkt,pBuff,buffLen,0) == ERROR)
    {
        /*发送错误*/
        logMsg("netSend: error in sending buff to client\n",0,0,0,0,0,0);
        netCloseAll();
        return(ERROR);
    }
    return(OK);
}
else
{
    /*根据网络初始化情况给出警告信息*/
    if(flagNetInit==NET_INIT_NULL)
        logMsg("netSend: server has not been initialized\n",0,0,0,0,0,0);
    else if(flagNetInit==NET_INIT_LISTENSKT)
    {
        logMsg("netSend: no connection created yet\n",0,0,0,0,0,0);
        logMsg("netSend: please connect to VxWorks server with PC
programm\n",0,0,0,0,0,0);
    }
    else
    {
        logMsg("netSend: you message is invalid\n",0,0,0,0,0,0);
    }
    return(ERROR);
}
}

/*************************************************************************
 STATUS netRecv()
 函数说明:    从已经建立的网络通道接收数据。
 参数:        无
 返回:        如果接收数据失败返回 ERROR，否则循环接收数据。
 调用:        void netCloseAll();
 被调用:      STATUS netInit();
 *************************************************************************/
STATUS netRecv()
{
    char pBuff[NET_MSG_MAX_SIZE+1];      /*接收数据的存储 buff*/
    int  recvLen;                        /*实际接收数据的长度*/

    /*如果网络正常，接受并显示数据*/
    while(flagNetInit==NET_INIT_COMMUSKT)
    {
        recvLen = recv(commuSkt,pBuff,NET_MSG_MAX_SIZE,0);
        if((recvLen!=ERROR) && (recvLen!=0))
```

```
            {
                pBuff[recvLen]=0;
                logMsg("netRecv:   received   message   from   client:\n%s\n",
(int)pBuff,0,0,0,0,0);
            }
            else
            {
                /*接收出错，退出*/
                netCloseAll();
                return(ERROR);
            }
        }

        /*网络通信通道被关闭，退出*/
        logMsg("netRecv: net connection is shut down, quit\n",0,0,0,0,0,0);
        return(OK);
    }


    /*************************************************************************
     void netCloseAll()
     函数说明:    关闭命令 socket，侦听 socket 以及和 socket 相关的任务
     参数:        无
     返回:        无
     调用:        无
     被调用:
                  STATUS netSend(char *pBuff);
                  STATUS netRecv();
                  用户在 shell 下调用
     *************************************************************************/
    void netCloseAll()
    {
        int taskId;

        /*删除网络接收任务，如果是被该任务调用则不删除*/
        taskId = taskNameToId("TNAME_RECV");
        if(taskId!=taskIdSelf() && taskId!=ERROR)
        {
            taskDelete(taskId);
        }

        /*根据网络初始化的不同状态作相应的退出工作*/
        if(flagNetInit==NET_INIT_COMMUSKT)
        {

            close(commuSkt);
```

```
        close(listenSkt);
    }
    else if(flagNetInit==NET_INIT_LISTENSKT)
    {
        close(listenSkt);
    }

    /*设置全局标志变量，表示网络恢复到未初始化的状态*/
    flagNetInit=NET_INIT_NULL;
    logMsg("netCloseAll: net connection shut down\n",0,0,0,0,0,0);
}
```

3.2 基于缓冲队列的多任务网络通信

在实际的采集板系统中，需要的并不是将获得的字符串打印，而是用固定格式的字符串表示特定信息，控制端和受控端都需要对其进行适当的反应。将网络上传递的含有命令性质的特性信息称为命令。为了保证信息的不丢失，使用缓冲队列将所有信息缓存，然后再逐个读出并执行，基本模式见图 3-3。

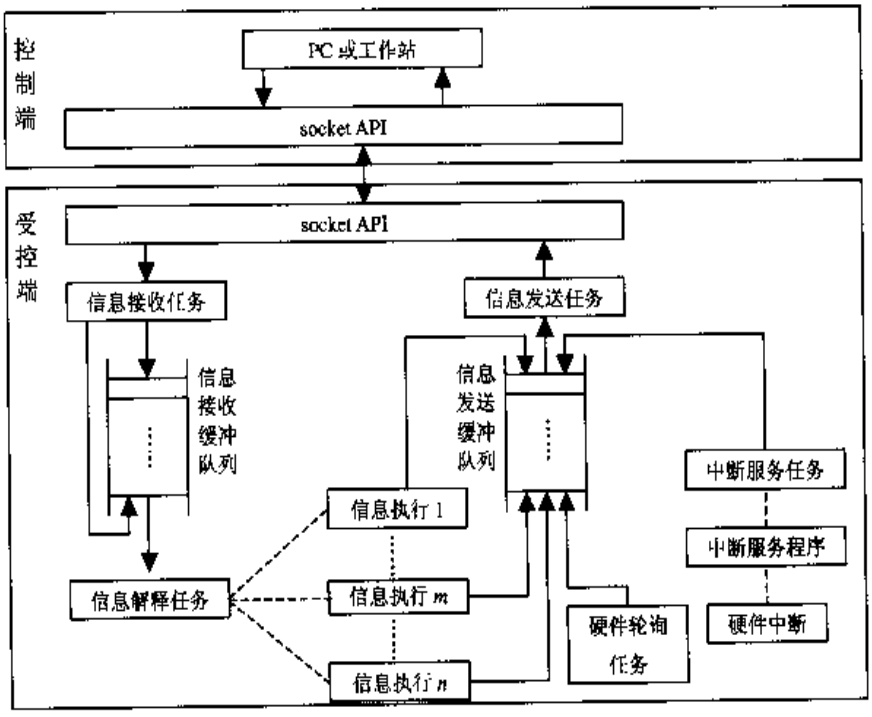


图 3-3 基于缓冲对列的双工通信模式

3.2.1 多任务系统

从图 3-3 可以看出，受控端（也就是运行 VxWorks 的采集板）使用了多个任务来实现网络通信，由控制端发起的通信过程如下。

（1）控制端：接收用户操作信息，将其转换为受控端可以理解的命令格式，通过 socket

接口下传到受控端，然后等待返回信息；

（2）受控端：信息接收任务从 socket 接口接收命令，将其添加到信息接收缓冲队列；信息解释任务从信息接收缓冲队列取出信息，根据具体情况执行特定的软硬件操作，操作结束后将执行结果添加到信息发送缓冲队列；信息发送任务从信息发送缓冲队列取出信息，通过 socket 接口将其返回给控制端。

（3）控制端：接收执行结果后返回给操作人员。

由受控端发起的通信过程如下。

（1）受控端：硬件产生中断，操作系统立即执行已经被绑定的中断服务程序，中断服务程序通知中断服务任务，后者将中断信息添加到信息发送缓冲队列；如果采取硬件轮询方式，也可以是轮询任务将信息添加到发送队列；信息发送任务将此信息上传给控制端。

（2）控制端：接收信息然后提供给操作人员，由操作人员选择下一步的操作方案。

信息接收缓冲队列的主要作用是缓存从控制端发下来的命令，保证所有的命令得到尽快地接收，避免由于接收速度慢造成控制端网络发送模块的阻塞，从而给控制端的软件带来不必要的麻烦。

信息发送缓冲队列则是将所有需要发送的命令缓存再发送，这样分离了网络发送和硬件的中断，在控制端软件接收模块阻塞而导致网络阻塞的情况下，不会影响中断处理的速度或者定时轮询的次数。

在受控端软件采用了双缓冲队列的情况下，控制端软件可以不再使用缓冲队列，而把所有的网络缓冲放在受控端。但为了软件的通用行和灵活性，应当在通信双方都建立双缓冲队列。

在多任务操作系统 VxWorks 下，任务由唯一的 ID 来标识，并且对应于某一个特定的任务名。系统支持 256 个优先级。在默认情况下，0 为最高优先级，255 为最低优先级，并且对同级的所有任务采取抢占式调度。建议用户在程序初始化时调用 kernelTimeSlice() 函数使系统进入轮转调度模式。

每个任务都有自己的状态，是下面中的一种或几种的组合状态：

excecute：	正在运行，占用 CPU。
ready：	就绪，但别的任务正在占用 CPU，排队等待占用 CPU 的任务释放 CPU。
pended：	阻塞，有条件不能被满足，等待条件被释放后进入 ready 状态。
delayed：	延时，为了保证任务不会死循环，使用 taskDelay(ticks)使任务进入延时状态，将在 ticks/sysClkRate()秒后退出，类似与 VC 下的 Sleep()。
suspended：	挂起，软件方式设定任务不参与排队等待 CPU，必须调用 resume()才会退出挂起状态。

中断服务程序经过事先绑定，一旦硬件产生中断，就会被优先调用，相当于最高优先级。但由于可能会造成阻塞的函数都不能在中断服务程序中调用，因此中断服务程序只进行最基本的清中断工作，而具体的功能则由中断服务任务执行。从整体上看，中断的响应优先级相当于中断服务任务的优先级。

对任务的操作使用 taskLib 中的库函数，需要包含的头文件为：taskLib.h。taskLib 中

的库函数见表 3-2。

表 3-2 VxWorks 下常用的任务处理函数

函 数	说 明
taskSpawn()	发起任务，即创建并运行任务
taskInit()	创建任务，并将其堆栈建立在指定位置
taskActivate()	激活任务，即使已经创建的任务开始运行
taskRestart()	重新开始运行任务，恢复堆栈到任务创建时的情况
exit()	退出任务（ANSI）
taskDelete()	删除任务，此函数不删除被设置了安全保护的任务
taskDeleteForce()	删除任务，包括被设置了安全保护的任务
taskSafe()	设置任务的安全保护，使其不能被 taskDelete()删除
taskUnsafe()	取消任务的安全保护
taskSuspend()	挂起一个任务
taskResume()	使任务退出挂起状态
taskPrioritySet()	设定任务的优先级
taskPriorityGet()	获得任务的优先级
taskLock()	锁定任务调度，保证当前正在执行的任务的执行，防止被系统的任务调度打断
taskUnLock()	重新允许任务调度
taskIdSelf()	获得任务本身的 ID
taskIdVerify()	判断 ID 对应的任务是否仍然存在
taskDelay()	延迟
taskTcb	获得任务的控制块（TCB，Task Control Block）

此外，经常使用的库还有 taskInfo，主要包括的是获得任务信息的库函数。需要包含的头文件为：taskLib.h。本书中涉及的函数为：taskNameToId()，即从任务的任务名获得对应的 ID 号。在任务延迟时还需要和系统直接相关的库 sysLib 中的函数 sysClkRateGet()和 sysClkRateSet()，需要包含的.h 文件为：sysLib.h。

下面介绍在本书程序中涉及的任务操作函数，更详细的信息可以查阅 Tornado 下的函数库在线帮助。

- 函数：int taskSpawn (char *name, int priority, int options, int stackSize, FUNCPTR

entryPt, int arg1, int arg2, int arg3, int arg4, int arg5, int arg6, int arg7, int arg8, int arg9, int arg10)

描述：发起以 entryPt 指向的函数为入口的任务，并且可以带 10 个参数作为 entryPt 指向函数的参数。可以使用一个函数发起多个任务，并同时运行。例如有函数 STATUS test(int param1, int param2)，则使用下面两条语句发起任务名分别为“tTest1”和“tTest2”，参数分别为“1，2”和“3，4”，任务优先级为 100，任务堆栈大小为 1000byte，test 作为函数入口的两个任务。

taskSpawn(“tTest1”,100,0,1000,(FUNCPTR)test,1,2,0,0,0,0,0,0,0,0);

taskSpawn(“tTest2”,100,0,1000,(FUNCPTR)test,3,4,0,0,0,0,0,0,0,0);

由于函数中的局部变量在发起任务时都存储在任务堆栈中，因此有较大局部数组的函数在发起任务时必须注意堆栈大小。

在 shell 下输入 sp，类似于 taskSpawn，sp 使用 taskSpawn 的默认值。格式为：

sp 函数名 函数参数 1，函数参数 2，…

shell 下输入 sp 携带的函数参数可以按照实际使用的参数个数，而不需要像程序中使用 taskSpawn()那样必须写满 10 个。如果输入的参数个数少于函数参数个数，则后面的参数值为 0。由 sp 发起的任务默认任务名，格式为“u[n]t[m]”，其中的 n 和 m 按顺序逐渐递增，保证任务名不重复。任务默认优先级 100，默认堆栈大小 20000byte，例如：

sp test,1,2

在 shell 下输入“i”，可以察看当前存活的任务的信息，如任务名、优先级和运行状态等。

参数：

*name: 指向发起任务的任务名字符串首地址的指针，存放在任务堆栈的启始（pStackBase）处

priority: 发起任务的优先级

options: 任务的可选项，包括：

VX_FP_TASK（0x0008，使任务支持浮点型计算，必须有嵌入式 CPU 的同时支持）

VX_PRIVATE_ENV［0x0080，支持私有环境变量（参见 envLib）］

VX_NO_STACK_FILL（0x0100，不在创建任务时使用 checkStack()填充堆栈）

VX_UNBREAKABLE（0x0002，不允许任务的断点调试）

stackSize: 堆栈加上任务名的总大小，单位为 byte

entryPt: 任务对应的函数入口指针

arg1～arg10: 函数参数

返回：任务 ID，如果分配堆栈时内存不足或创建失败返回 ERROR

- 函数：STATUS taskDelete (int tid)

描述：删除一个任务，同时释放堆栈和对应的 WIND_TCB 任务控制块的资源。在删除的同时，会调用 taskDeleteHookAdd()函数绑定的函数。

参数：

tid，待删除的任务 ID。

返回：OK，如果任务无法删除则返回 ERROR。

- 函数：STATUS taskDelay(int ticks)

描述：此函数使调用它的任务释放 CPU，并等待规定的时间。通常用于定时查询一些不使用中断的外部条件。

参数：

ticks，延迟的 tick 数。VxWorks 系统将时间分为时间片（tick）使用，默认 1s 为 60 个时间片。可以通过函数 sysClkRateGet()/sysClkRateSet()来获得/设置 1s 对应的时间片个数。也就是说，ticks 个时间片对应的时间是 ticks/sysClkRateGet()秒。

返回：OK；当被中断服务程序调用，或者调用其任务接收到没有被阻塞或者被忽略掉的信号（signal）时，返回 ERROR

- 函数：int sysClkRateGet()

描述：获得系统中 1s 对应的时间片个数。具体情况需要对应于相对的 BSP。在使用此函数前，请仔细查看 BSP 文件夹中的参考文档。

参数：无。

返回：系统中 1s 对应的时间片个数。

- 函数：STATUS sysClkRateSet(int ticksPerSecond)

描述：设置系统时钟每秒钟的中断数，通常被 usrConfig.c 中的 usrRoot()函数调用。此函数可能和 POSIX 标准的 clockLib 相关，请参看 clockLib。与 sysClkRateGet()类似，需要察看 BSP 的参考文档。提高每秒的 tick 数可以将系统时间划分得更细，但不能将其设置得太高，否则会导致系统将大量时间花费在任务调度上，降低系统效率。通常设置的 tick 数在几十到几千的范围内。

参数：

ticksPerSecond,　每秒的时间片数目

返回：OK，如果参数非法（例如将参数设为负数）或者无法设定时间片则返回 ERROR。

- 函数：int taskNameToId(char *name)

描述：通过任务名获得该任务的 ID。用这种方法获得任务的控制通常是低效的，因为函数必须搜索的任务列表的全部。

参数：

*name，指向任务名字符串的首地址的指针。

返回：任务 ID，当找不到使用该任务名的任务时返回 ERROR。

3.2.2 多任务的优先级划分

为了保证所有的信息被尽快地从套接字通道取出，在多个任务并行运行的时候，应当优先考虑信息接收任务。为了保证硬件轮询任务在时间上的准确性，如果使用的是 taskDelay()方式，可以适当提高其优先级，而如果使用的是看门狗，则没有这个必要（这两种定时方法将在第五章中详细介绍）。由于拥有缓冲队列，信息发送任务可以在别的任

务不执行的情况下再执行。一般情况下，用户程序使用的任务优先级不要小于 50。如果用户程序的优先级过高，超过了系统任务的优先级，可能会影响系统的运行。为了调试方便，一般由程序发起的任务优先级应当低于用 shell 下 sp 发起的任务的优先级，以保证 shell 的优先响应。故将用户优先级设定为低于 100。根据任务执行的先后次序划分任务优先级，见图 3-4。

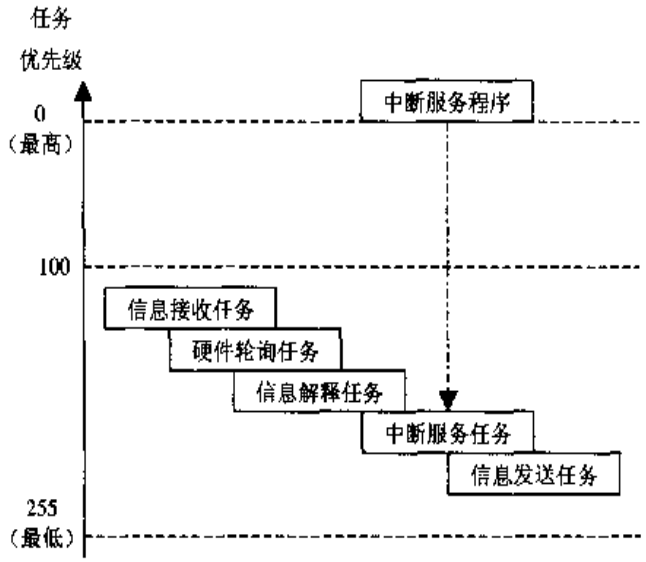


图 3-4　基于缓冲对列的双工通信任务优先级划分

任务间的同步使用信号灯。

VxWorks 的信号灯操作库为 semLib、semBLib、semCLib、semMLib，分别对应通用信号灯操作与二进制、计数型和互斥信号灯的操作。需要包括的.h 文件为 semLib.h。包含的函数见表 3-3。

表 3-3　信号灯操作函数

semGive()	释放信号灯
semTake()	获取信号灯
semDelete()	删除信号灯
semFlush()	使所有被该信号灯阻塞的任务脱离阻塞状态
semBCreate()	创建二进制型信号灯
semCCreate()	创建计数型信号灯
semMCreate()	创建互斥型信号灯

当一个任务的执行需要由另一个任务来触发时，就可以使用信号灯。例如任务 A 和任务 B，任务 A 一直等待某个条件被满足，而这个条件是否满足由任务 B 来确定。条件不满足时任务 A 必须不占用 CPU，而条件一旦满足，任务 A 立刻开始执行。此时任务 A 采用

下面的格式：

```
void funcA()
{
    while(semTake(semID,WAIT_FOREVER)==OK)
    {
        /*执行条件满足的需要进行的工作*/
    }
}
```

任务 B 采用下面的格式：

```
void funcB()
{
    /*执行条件是否满足的判断*/
    if(1/*条件满足*/)
        semGive(semID);
}
```

可以将信号灯看成一个变量，每个 semGive()使变量加 1，如果变量值已经达到上限，则不会改变（变量上限在信号灯初始化时确定）；每个 semTake()使变量减 1，如果变量值已经为 0，则 semTake()使调用它的任务阻塞，或者直接返回错误。

计数型信号灯的计数上限由 semCCreate()的参数决定，二进制型信号灯的计数上限为 1，而互斥型信号灯可以看成是某种特殊的二进制型信号灯。

中断服务程序和中断服务任务间的同步、缓冲队列的操作也需要使用信号灯。在本书中主要涉及的是二进制型和计数型信号灯，下面介绍在本书程序中涉及的信号灯函数，更详细的信息可以查阅函数库的在线帮助。

- 函数：SEM_ID semBCreate(int options, SEM_B_STATE initialState)

描述：初始化二进制信号灯。信号灯的初始状态由参数 initialState 决定，可以是 SEM_FULL（1）或者 SEM_EMPTY（0）。参数 options 决定被该信号灯阻塞的任务哪一个在信号灯被释放时优先执行，可以是基于优先级的［SEM_Q_PRIORITY（0x01），在所有被其阻塞的任务中，优先级最高的任务先执行］，也可以是基于先进先出的［SEM_Q_FIFO（0x00），在所有被其阻塞的任务中，最先被阻塞的任务先运行］。

参数：

options, 选择信号灯的排队机制。
initialState, 选择信号灯的初始状态。

返回：信号灯的 ID，如果创建失败则返回 NULL。

- 函数：SEM_ID semCCreate (int options, int initialCount)

描述：初始化计数型信号灯。初始计数等于参数 initialCount。信号灯的排队机制通过参数 options 控制，分为 SEM_Q_PRIORITY（0x01）和 SEM_Q_FIFO（0x00）。

参数：

options, 选择信号灯的排队机制。
initialCount, 信号灯的初始计数，必须大于或等于 0。

返回：信号灯的 ID，如果创建失败则返回 NULL。

- 函数：STATUS semGive (SEM_ID semId)

描述：释放信号灯。对于二进制信号灯，如果状态为 0，则变为 1；如果状态为 1，则保持不变。对于计数型信号灯，将计数加一。

参数：

semId，信号灯的 ID。

返回：OK，如果信号灯 ID 为非法 ID 则返回 ERROR。

- 函数：STATUS semTake (SEM_ID semId, int timeout)

描述：获取信号灯。获取成功将导致二进制信号灯的状态变为 0，计数型信号灯的计数减一。获取信号灯的时间由参数 timeout 决定，可以设置为不等待[NO_WAIT（0）]，即无论获取成功与否都立刻返回，因此不会造成调用其的任务阻塞；也可以设置为永远[WAIT_FOREVER（-1）]，即一直等待信号灯，在获取成功或者信号灯被删除后才会返回，这之前会阻塞调用其的任务；还可以设置成某一个正整数，单位为 tick，即在 timeout/sysClkRateGet()秒的时间内等待信号灯。如果信号灯获取由于超时而结束，则返回 ERROR。

注　此函数不能被中断服务程序调用。

参数：

semId，信号灯 ID。

timeout，获取信号灯的超时限制。

返回：如果获得了信号灯返回 OK，否则因为信号灯 ID 非法或者超时返回 ERROR。

- 函数：STATUS semDelete (SEM_ID semId)

描述：删除信号灯。任何被此信号灯阻塞的任务都将脱离阻塞状态，并返回 ERROR。注意删除信号灯（特别是互斥型信号灯）可能会造成麻烦。对应互斥型，必须在获得该信号灯后再删除。本书中采取的方法是：先删除被此信号灯阻塞的任务，再删除信号灯。

参数：

semId，信号灯 ID。

返回：OK，如果信号灯 ID 非法则返回 ERROR。

3.2.3 缓冲队列的实现

为了将接收到的和待传输的信息缓存，首先必须建立缓冲队列，队列中保存的是这些信息的头指针。而信息解释任务和信息发送任务将根据这些头指针指向的数据来进行实际的操作。由于保存的指针和实际的信息内容及长度无关，所以缓冲队列对于所有的信息来说是通用的。信息符合一定的规范，将自身的长度存储在信息内容的指定位置上，具体的信息格式将在第四章中讨论。

使用链表来动态实现队列中信息的动态添加和删除。每个包含一个信息首地址的单节称为链表的一个元素。链表需要实现元素向链表头和链表尾的添加，以及从链表头删除元素的功能。为了节省空间，是用单向链表，并用两个全局的元素指针来指向链表头和链表

尾，用一个全局变量表示链表的长度，以省略查找过程，直接获得链表头和链表尾的内容。

每个链表元素为一个结构体 struct cmdSingle，结构体内容包括两项：网络信息的头指针 pBuff，指向链表下一个元素的指针 pNext。全局变量为指向链表头的指针 queueHead，指向链表尾的指针 queueRear，以及链表长度 queueLen。

当链表元素为空时，链表结构见图 3-5。

图 3-5　空链表结构图

当链表元素为 1 时，链表头和链表尾都指向这唯一的元素，而该元素中指向下一元素的指针被赋值为 NULL，结构见图 3-6。

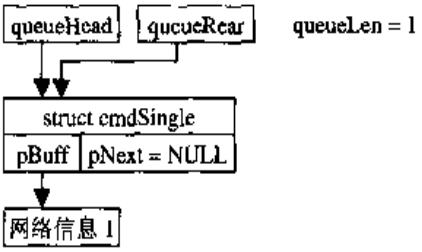


图 3-6　一个元素的链表结构图

当链表元素超过 1 时，链表头和链表尾指向的对象分别为链表的头尾，而最后一个元素的指向下一个元素的指针被赋值为 NULL，图 3-7 是一个有三个元素的链表结构图。

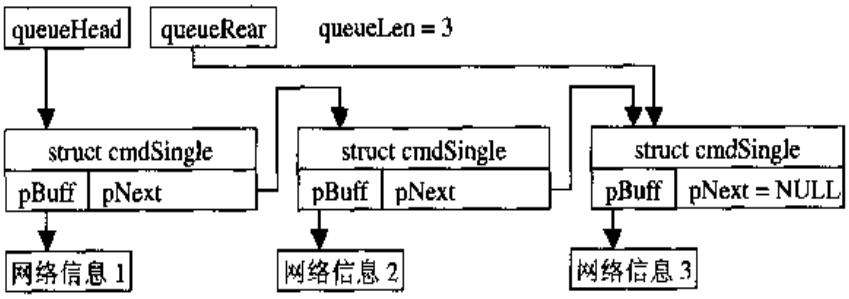


图 3-7　三个元素的链表结构图

从图 3-5～图 3-7 可以看出，如果需要添加一个元素到链表中，则首先需要创建这个元素。当添加元素到空链表时，需要将链表头指针和链表尾指针队指向这个队列。将该元素中指向下一元素的指针赋为 NULL，表示没有后继元素。最后修改链表长度为 1。

如果需要添加一个元素到链表头。需要将这个新元素中指向下一元素的指针赋为原来的链表头指针，再把链表头指针指向这个新的元素，并将链表长度加 1。如果添加元素到链表尾，则首先将这个元素中指向下一个元素的指针赋为 NULL，然后将链表尾指针指向的元素中指向下一个元素的指针从 NULL 改为指向这个新元素，再将链表尾指针也指向这

个新元素，最后将链表长度加1。

如果需要从队头删除一个元素，则先将链表头指针指向的元素中指向下一个元素的指针缓存，然后将这个元素的空间释放掉，再将链表头指针赋为刚才缓存的下一个元素指针。最后将链表长度减一。如果被删除的是最后一个元素，则删除完毕后链表头指针已经为空指针，还需要将链表尾指针也设为NULL。

需要删除全部元素时，只需要一直删除队头元素，直到链表长度等于0为止。

在实际使用时，网络通信需要接收和发送两个缓冲队列，因此在下面的程序中定义了两个队列，链表头指针、链表尾指针和链表长度都有两个，分别供两个链表使用。当需要使用多个链表时，只需要修改宏定义QUEUE_NUM控制链表数目即可。双缓冲队列的具体程序实现见例3-2。

例3-2 网络通信使用的缓冲队列(netQueue.h和netQueue.c)

```
/*****************************************************************************
 chengjy@felab, copyright 2002-2004
 netQueue.h
 网络缓冲队列常数定义
 *****************************************************************************/
#ifndef _NETQUEUE_H
#define _NETQUEUE_H

/*~~~~~~~~~~~~~~网络命令通道缓冲队列~~~~~~~~~~~~~~~~*/
/*队列的单节结构*/
struct cmdSingle
{
    struct cmdSingle *pNext;
    unsigned char *pBuff;
};

/*定义队列添加的优先级*/
#define QUEUE_PRI_HIGH      1       /*队头*/
#define QUEUE_PRI_LOW       0       /*队尾*/

/*定义队列的个数，对于双缓冲队列，定义为2*/
#define QUEUE_NUM           2       /*必须是大于0的整数*/

/*定义返回状态*/
#define STATUS_NORMAL           0x00
#define STATUS_ERROR            0x01
#define STATUS_WARNING          0x02
#define STATUS_INVALID          0x03
#define STATUS_UNAVAILABLE      0x04
#define STATUS_DISABLE          0x05
```

```
#endif  /*_NETQUEUE_H*/

/*************************************************************************
 chengjy@felab, copyright 2002-2004
 netQueue.c
 网络缓冲队列操作。队列的个数由宏定义 QUEUE_NUM 规定。
 函数:
        void queueInit();
        char queueAdd(int index, unsigned char* pBuff, int pri);
        char queueDelHead(int index);
        char queueDelAll();
 调用:      无
 被调用:    网络通信程序及用户程序的初始化部分
 *************************************************************************/
#include "vxWorks.h"
#include "taskLib.h"

#include "netQueue.h"

/*定义队头和对尾的指针*/
struct cmdSingle *queueHead[QUEUE_NUM],*queueRear[QUEUE_NUM];
/*定义队列长度*/
int queueLen[QUEUE_NUM];

void queueInit();
char queueAdd(int index, unsigned char* pBuff, int pri);
char queueDelHead(int index);
char queueDelAll();

/*************************************************************************
 void queueInit()
 函数说明:  队列初始化。
 参数:      无
 返回:      无
 调用:      无
 被调用:    网络的初始化模块。
 *************************************************************************/
void queueInit()
{
 int i;

 /*设置头尾指针为空，队列长度为 0*/
 for(i=0;i<QUEUE_NUM;i++)
```

```
{
    queueHead[i] = NULL;
    queueRear[i] = NULL;
    queueLen[i] = 0;
}
}

/*******************************************************************************
 char queueAdd(int index, unsigned char* pBuff, int pri)
 函数说明:    队列元素添加,根据 pri 决定添加到队头还是队尾。
 参数:
        index,  需要添加到的队列号，必须大于或等于 0 且小于 QUEUE_NUM。
        pBuff,  从网络接收数据或者待发送数据的首地址指针。
        pri,    添加的优先级。
 返回:        成功添加返回 STATUS_NORMAL，否则返回 STATUS_ERROR。
 调用:        无
 被调用:      网络的信息接收任务和其他需要发送信息的任务。
 *******************************************************************************/
char queueAdd(int index, unsigned char* pBuff, int pri)
{
    struct cmdSingle *queueHeadTmpt,*queueRearTmpt;

    /*判断参数是否合法*/
    if( (index<0) || (index>=QUEUE_NUM) )
    {
        logMsg("queueAdd: unable to local queue index %d\n", index,0,0,0,0,0);
        logMsg("queueAdd: index must be greater than or equal to 0 and smaller
                than %d\n",QUEUE_NUM,0,0,0,0,0);
    return(STATUS_ERROR);
    }

    taskLock();  /*禁止任务调度，保证当前任务执行过程中不会被打断*/

    /*如果队列中还没有元素，当前添加的元素同时位于对头和队尾*/
    if(queueLen[index]==0)
    {
        queueHead[index] = malloc(sizeof(struct cmdSingle));
        if(queueHead[index] == NULL)
        {
            /*内存分配失败*/
            logMsg("queueAdd: can't malloc enough memory\n",0,0,0,0,0,0);
            taskUnlock();
            return(STATUS_ERROR);
        }
```

```
        queueRear[index] = queueHead[index];
        queueRear[index]->pNext = NULL;
        queueRear[index]->pBuff = pBuff;
        queueLen[index]++;
    }
    else if(pri == QUEUE_PRI_HIGH)  /*高优先级，添加到队头*/
    {
        queueHeadTmpt = malloc(sizeof(struct cmdSingle));
        if(queueHeadTmpt == NULL)
        {
            /*内存分配失败*/
            logMsg("queueAdd: can't malloc enough memory\n",0,0,0,0,0,0);
            taskUnlock();
            return(STATUS_ERROR);
        }
        queueHeadTmpt->pNext = queueHead[index]->pNext;
        queueHeadTmpt->pBuff = pBuff;
        queueHead[index]->pNext = queueHeadTmpt;
        queueLen[index]++;
    }
    else if(pri == QUEUE_PRI_LOW)   /*低优先级，添加到队尾*/
    {
        queueRearTmpt = malloc(sizeof(struct cmdSingle));
        if(queueRearTmpt == NULL)
        {
            /*内存分配失败*/
            logMsg("queueAdd: can't malloc enough memory\n",0,0,0,0,0,0);
            taskUnlock();
            return(STATUS_ERROR);
        }
        queueRearTmpt->pNext = NULL;
        queueRearTmpt->pBuff = pBuff;
        queueRear[index]->pNext = queueRearTmpt;
        queueRear[index] = queueRearTmpt;
        queueLen[index]++;
    }
    else /*优先级参数非法*/
    {
        logMsg("queueAdd: priority be %d, should only be %d or %d\n",
                pri,QUEUE_PRI_HIGH,QUEUE_PRI_LOW,0,0,0);
    }

    taskUnlock();    /*允许任务重新调度*/
    return(STATUS_NORMAL);
```

```
}

/*******************************************************************************
 char queueDelHead(int index)
 函数说明:    队列队头元素删除。
 参数:
        index,  需要删除元素的队列号，必须大于或等于 0 且小于 QUEUE_NUM。
 返回:        STATUS_NORMAL。
 调用:        无
 被调用:      网络的信息解释任务
              信息发送任务
              char queueDelAll()
 *******************************************************************************/
char queueDelHead(int index)
{
    struct cmdSingle *queueHeadTmpt;

    /*判断参数是否合法*/
    if( (index<0) || (index>=QUEUE_NUM) )
    {
        logMsg("queueDelHead: unable to local queue index %d\n",
                index,0,0,0,0,0);
        logMsg("queueDelHead: index must be greater than or equal to 0
                and smaller than %d\n",QUEUE_NUM,0,0,0,0,0);
        return(STATUS_ERROR);
    }

    taskLock();  /*禁止任务调度，保证当前任务执行过程中不会被打断*/

    queueHeadTmpt = queueHead[index]->pNext;
    free(queueHead[index]);
    queueHead[index] = queueHeadTmpt;
    queueLen[index]--;

    if(queueLen[index]==0)  /*如果连最后一个元素也被删除，则队列尾指针需要设定为空*/
    {
        queueRear[index]=NULL;
    }

    taskUnlock();    /*允许任务调度*/

    return(STATUS_NORMAL);
}
```

```
/*****************************************************************************
 char queueDelAll()
 函数说明:    队列元素全部删除。
 参数:        无
 返回:        STATUS_NORMAL
 调用:
              char queueDelHead(int index)
 被调用:
              网络的退出模块
 *****************************************************************************/
char queueDelAll()
{
    char *pBegin;
    int i;

    /*将所有队列中的所有元素从队头开始全部删除*/
    for(i=0;i<QUEUE_NUM;i++)
 {
        while(queueLen[i]>0)
        {
            pBegin = queueHead[i]->pBuff;
            free(pBegin);
            queueDelHead(i);
        }
    }
    return(STATUS_NORMAL);
}
```

3.2.4 基于双缓冲队列网络通信的特点

使用双缓冲队列进行网络通信，可以给系统带来众多优势。为了理解这一点，首先需要了解通信的死时间。在通信过程中，如果信息到达时间相距太短，就有可能引起信息处理不及时或信息丢失。定义通信死时间为：在保证受控端正确接收前提下，由用户操作引起的两次信息发送可以间隔的最短时间。下面将讨论缓冲队列的使用对系统死时间的影响。

- 双方都没有缓冲队列 t(0,0)：

通信死时间为从控制端发出消息、受控端接收信息、分析信息、执行信息、信息返回到控制端接收信息并通知操作人员的全部时间。

- 有信息接收缓冲对列 t(0,1)：

通信死时间为接收端将信息添加到信息接收缓冲队列中所需的时间，一般来说受控端的嵌入式 CPU 频率低于控制端，因此死时间定为受控端添加消息的时间。

- 双方都有信息发送缓冲队列和信息接收缓冲队列 t(1,1)：

将信息发送时间间隔设为 t(0,1)，则可以保证正确接收。实际上由于信息接收任务对通信死时间内到达的第一条信息仍然可以接收，只是接收时间被推后到 t(0,1)时间结束后，所以在理论上将控制端的信息发送时间间隔设为 t(0,1)/2，系统仍然可以正常工作。但是考虑到网络传输时间的晃动，最好仍然设为 t(0,1)。

受控端时间间隔采用消息发送任务主动调用 taskDelay()方法实现，主控端则根据软件具体情况而定，例如对于 window 系统下 VC 编制的程序，可以使用 Sleep()实现。对于发送方来说，不必再考虑接收方死时间，信息产生后立即将其填入消息发送队列，并且在填入过程中仍然响应其他信息的产生，相当于在操作人员层面上实现了零通信死时间。

需要说明的是，受控端中断的发生也是随机的，因此在半双工通信中由中断产生的信息同样可能被丢失，双缓冲队列的使用使控制端可以正确接收所有的中断信息，原因同上。

使用缓冲队列实现了信息的流水线处理，降低了信息通信双方时间上的关联性，带来了操作人员层面的系统零死时间，同时也带来了其他的好处：

- 提高 CPU 利用率

零死时间使全双工成为可能， 理论上不存在因为等待返回而产生的 CPU 空闲且不响应时间，CPU 可以在任何时间被利用，客观上提高了 CPU 的利用率和软件的效率。

- 信息解随机

由于受控端的信息发出时间主要由操作人员决定，因此两条信息的时间间隔在很大程度上是随机的，但受控端执行特定信息的时间是基本固定的。缓冲队列将随机发出的信息排队，将信息的时间特性转换为其在队列中的位置特性，受控端软件不需要知道信息发出的时间，只需要知道队列中是否仍然有信息没有被执行，相当于解除了随机性。这使得控制端信息的批处理发送成为可能，例常性工作的操作时间可以大大缩短，因此更加适用于半自动控制。

- 紧急程度决定执行顺序

控制端可以通过设定信息的格式将信息紧急程度传递给受控端，紧急信息将被添加到信息接收队列队头，因此被优先执行；紧急信息的执行结果也会被添加的信息发送队列队头，被优先发送。同时中断信息也会被优先发送。这种根据紧急程度决定执行顺序的模式更符合逻辑，更适用于半自动控制。

3.3 网络通断检测

当通信的一方非正常退出时，通信的另一方也应当做出相应的反应。对于服务器端来说，应当关闭通信用套接字，并重新进入准备接收状态，等待客户端的再次连接。能够自动切换到等待状态的服务器端程序对于采集板来说很重要，因为最终程序应该能够保证上电后一直正常运行，其运行服从于但不依赖于控制端。采集板程序能够自动检查套接字的通断状态并调整本板程序的运行，也能够使控制端程序的运行调试更加方便。控制端程序可以随时强制退出（dos 程序的 Ctrl+C，Windows 程序的 Alt+Ctrl+Del->任务管理器->结束任务），而不需要在退出前关闭自己的套接字，且一旦采集板操作系统启动完毕，就可以

反复地退出再连接，不需要等待控制端重新启动。

同样，如果采集板作为客户端，控制端的程序一旦退出，客户端程序也应当立刻关闭自己的通信套接字。

如果通信的一端关闭了自己的套接字，或者非正常退出（断电退出除外），通信的另一端在调用接受或者发送函数的时候都会返回错误。

3.3.1 VxWorks 作为服务器端的网络检测程序

下面首先讨论 VxWorks 作为服务器端的情况。对于一般的采集板系统，会建立一个服务器端，一旦上电后就进入准备接收连接的状态。当连接建立后，信息接收任务会不断地试图从网络接收信息，而信息发送任务则根据实际情况，在某些时刻尝试发送信息。由于信息接收任务一直在监视着网络，因此服务器端可以清楚地获得另一端是否退出的情况。

在图 3-3 中的所有任务之外，再添加一个网络监控任务，并将其优先级设置在其他任务之上。此任务和信息接收任务、信息发送任务间使用二进制型信号灯通信，并利用全局变量表示网络的初始化状态。一旦信息接收任务或者信息发送任务发现通信另一端已经退出，就会释放信号灯，而网络监控任务则试图获取这个信号灯。正常通信时，由于无法获得信号灯，网络监控任务处于被阻塞的状态，不占用 CPU；而一旦网络出错，该任务获得了信号灯，并凭借优先级高获得优先执行的机会，它负责发起网络关闭任务，删除信息接收任务、信息处理任务、信息发送任务和此任务本身，删除网络相关的信号灯，删除网络缓冲队列，然后重新将初始化网络。

由于服务器的地址已经经过了绑定，因此再次初始化的时候不需要初始化服务器端的侦听套接字，只需要直接进入 accept 状态。这里除了侦听套接字，所有和网络相关的变量全部被重新初始化，保证程序的稳定运行。同时，其他和网络有联系的任务也可以一并删除并重新初始化。

使用网络通断监控任务统一地进行网络的关闭处理，使各任务的功能更加独立，程序更便于管理。

例 3-3 给出了服务器端网络检查程序的部分函数，为了节省篇幅，与例 3-1 中重复的网络初始化、退出部分使用注释的方式作了解释，用户实际使用的时候可以根据情况再添加。同时网络接收和网络发送部分也只写了和网络监控相关的部分。用户需要添加网络缓冲队列的处理和程序的重入判断。

例 3-3 服务器端的网络监控程序(vxSvr.h 和 vxSvr.c)

```
/*************************************************************************
 chengjy@felab, copyright 2002-2004
 vxSvr.h
 网络监控任务需要的常数，只定义相关常数，其他常数
 需要编程用户自己定义。
 *************************************************************************/
#ifndef _VXSVR_H
#define _VXSVR_H
```

```
/*服务器初始化情况*/
#define NET_INIT_LISTENSKT      0x01
#define NET_INIT_COMMUSKT       0x02
#define NET_INIT_NULL           0x00

/*定义网络关闭时是否要重新初始化*/
#define MODE_NET_DEFAULT        0    /*重新初始化*/
#define MODE_NET_REINIT         1    /*不再初始化*/

/*定义任务名*/
#define TNAME_NETINIT           "tNetInit"
#define TNAME_NETRECV           "tNetRecv"
#define TNAME_NETSEND           "tNetSend"
#define TNAME_NETCHECKLINK      "tNetCheckLink"
#define TNAME_NETCLOSEALL       "tNetCloseAll"

/*用户任务堆栈大小*/
#define USER_STACK_SIZE         2000

/*taskSpawn 使用的用户任务优先级*/
#define TPRI_NETCLOSEALL        101
#define TPRI_NETCHECKLINK       102
#define TPRI_NETRECV            103
#define TPRI_NETSEND            104
#define TPRI_NETINIT            105

#endif   /*_VXSVR_H*/

/*****************************************************************************
 chengjy@felab, copyright 2002-2004
 vxSvr.c
 VxWorks 下建立服务器端的程序框架，加上了网络监测的模块。
 函数:
        void netInit(int mode);
        void netCMDRecv();
        void netCMDSend();
        void netCloseAll(int mode);
        void netCheckLink();
 调用:      无
 被调用:    shell 下手动调用
 说明:      编译下载完毕后，在 shell 下输入 sp netInit 初始化服务器端即可。注意本文档
            仅仅是框架，用户还需要根据需要添加内容。为了简洁起见，函数都设定为无返回。
 *****************************************************************************/
```

```
#include "vxWorks.h"
#include "taskLib.h"
#include "sockLib.h"
#include "inetLib.h"
#include "ioLib.h"
#include "logLib.h"
#include "string.h"
#include "stdio.h"
#include "netinet\\tcp.h"
#include "semLib.h"

#include "vxSvr.h"

int flagNetInit = NET_INIT_NULL; /*服务器初始化标志,程序中省略了对应的判断和操作*/

int listenSkt;          /*侦听 socket*/
int commuSkt;           /*通信 socket*/
SEM_ID semCmdLink;      /*网络连接出错信号灯*/

void netInit(int mode);
void netCMDRecv();
void netCMDSend();
void netCloseAll(int mode);
void netCheckLink();

/*此处需要 extern 外部关于队列操作的函数和全局变量*/

/*************************************************************************
 void netInit(int mode);
 函数说明:    网络初始化程序
 参数:        mode,    分为 MODE_NET_DEFAULT 和 MODE_NET_REINIT 两种,
                      分别对应初始化侦听+通信套接字和只初始化通信套接字两种。
 返回:        无
 调用:
              void netCMDRecv();
              void netCMDSend();
              void netCheckLink();
 被调用:      用户程序初始化模块
              char netCloseAll(int mode);
 *************************************************************************/
void netInit(int mode)
{
   /*需要添加重入控制*/
```

```
    if(mode == MODE_NET_DEFAULT)
    {
        /*建立本地的侦听用套接字 listenSkt 并bind和listen*/
    }

    /*接收外部连接，建立通信套接字 commuSkt*/

    if(1/*改成已经正确建立通信套接字的判断*/)
    {
        /*初始化网络通信的缓冲对列*/

        /*初始化缓冲队列的信号灯*/

        /*发起网络命令循环接收任务、命令发送任务和网络监控任务*/
        taskDelete(taskNameToId(TNAME_NETCHECKLINK));
        taskSpawn(TNAME_NETCHECKLINK,TPRI_NETCHECKLINK,0,USER_STACK_SIZE,
                (FUNCPTR)netCheckLink,0,0,0,0,0,0,0,0,0,0);
        taskDelete(taskNameToId(TNAME_NETRECV));
        taskSpawn(TNAME_NETRECV,TPRI_NETRECV,0,
                USER_STACK_SIZE,(FUNCPTR)netCMDRecv,0,0,0,0,0,0,0,0,0,0);
        taskDelete(taskNameToId(TNAME_NETSEND));
        taskSpawn(TNAME_NETSEND,TPRI_NETSEND,0,
                USER_STACK_SIZE,(FUNCPTR)netCMDSend,0,0,0,0,0,0,0,0,0,0);
    }
}

/*************************************************************************
 void netCMDSend();
 函数说明:    循环从网络发送缓冲队列中取出数据并发送。
 参数:        无
 返回:        无
 调用:        外部的队列处理函数。
 被调用:      void netInit(int mode)
 *************************************************************************/
void netCMDSend()
{
    while(1)
    {
        /*从缓冲对列获取数据，此处使用信号灯同步，
        在没有数据要发送时任务被阻塞，不会占用CPU*/

        /*发送数据*/
        if(send(commuSkt,0,0,0/*改成其他三个参数*/) == ERROR)
        {
```

```
            /*发送错误*/
            logMsg("netSend: error in sending buff to client\n",0,0,0,0,0,0);
            semGive(semCmdLink);
            break;
        }
    }
}

/*************************************************************************
 void netCMDRecv()
 函数说明:    从已经建立的网络通道循环接收数据，并添加到接收缓冲队列等待处理。
 参数:        无
 返回:        无
 调用:        外部的队列处理函数。
 被调用:      STATUS netInit();
 *************************************************************************/
void netCMDRecv()
{
    int recvLen;        /*实际接收数据的长度*/
    while(1)
    {
        recvLen = recv(commuSkt,0,0,0/*改成其他三个参数*/);
        if(recvLen!=ERROR && recvLen!=0)
        {
            /*添加到队列中等待处理*/
        }
        else
        {
            /*接收出错*/
            logMsg("netCMDRecv: error in receive data from
                    client\n",0,0,0,0,0,0);
            semGive(semCmdLink);
            break;
        }
    }
}

/*************************************************************************
 void netCloseAll(int mode)
 函数说明: 关闭侦听 socket ，并删除命令缓冲队列以及与 socket 相关的任务，并根据模式的
           不同选择是直接关闭侦听套接字 (MODE_NET_DEFAULT) 还是重新初始化网络
           (MODE_NET_REINIT)。
 参数:     mode,选择关闭方式。
 返回:     无
```

```
调用:       void netInit(int mode);
被调用:     void netCheckLink();
            用户在 shell 下调用
*************************************************************************/
void netCloseAll(int mode)
{
    int taskId;

    /*删除网络相关任务*/
    taskId = taskNameToId(TNAME_NETRECV);
    if(taskId!=taskIdSelf() && taskId!=ERROR)
    {
        taskDelete(taskId);
    }
    taskId = taskNameToId(TNAME_NETSEND);
    if(taskId!=taskIdSelf() && taskId!=ERROR)
    {
        taskDelete(taskId);
    }
    taskId = taskNameToId(TNAME_NETCHECKLINK);
    if(taskId!=taskIdSelf() && taskId!=ERROR)
    {
        taskDelete(taskId);
    }

    /*调用外部函数删除缓冲队列*/
    /*删除缓冲队列的信号灯*/

    /*删除网络通断检测用信号灯*/
    semDelete(semCmdLink);

    /*根据网络初始化的不同状态做相应的退出工作*/
    if(mode==MODE_NET_DEFAULT)
{
        close(commuSkt);
        close(listenSkt);
    }
    else if(mode==MODE_NET_REINIT)
    {
        close(listenSkt);
        /*重新初始化网络*/
        taskDelete(taskNameToId(TNAME_NETINIT));
        taskSpawn(TNAME_NETINIT,TPRI_NETINIT,0,USER_STACK_SIZE,
                  (FUNCPTR)netInit,mode,0,0,0,0,0,0,0,0,0);
```

```
    }
}

/*************************************************************************
 void netCheckLink()
 函数说明:    监测网络状态，出错即关闭网络并重新初始化。
 参数:        无
 返回:        无
 调用:
              char netCloseAll(int mode);
 被调用:
              void netInit();
 *************************************************************************/
void netCheckLink()
{
    /*由于优先极高，因此先创建信号灯才会进行网络的接收和发送*/
    semCmdLink = semBCreate(SEM_Q_FIFO,SEM_EMPTY);

    /*等待 send()和 recv()出错释放信号灯*/
    semTake(semCmdLink,WAIT_FOREVER);

    /*获得信号灯，表示网络连接断开*/
    semDelete(semCmdLink);
    taskSpawn(TNAME_NETCLOSEALL,TPRI_NETCLOSEALL,0,USER_STACK_SIZE,
            (FUNCPTR)netCloseAll,MODE_NET_REINIT,0,0,0,0,0,0,0,0,0,0);
}
```

3.3.2 VxWorks 作为客户端的网络检测程序

当 VxWorks 需要和多台机器或者一台机器上的多个程序通信时，需要建立多个套接字通道。通常，我们在 VxWorks 下建立两个相对独立的网络连接通道，分别用来传输命令和数据。传输命令的通道上只传输控制信息和返回的信息，数据量小，通常是双向传输；而数据通道上则传输大量的数据，通常只由 VxWorks 到控制软件。VxWorks 下的命令通道一般使用服务器，以保证控制端的软件可以随时连接到 VxWorks 并获得对其的控制权。数据通道可以采用服务器端，也可以采用客户端，为了程序简洁起见，采用客户端实现，并利用已经建立的命令通道对其的连接和断开进行控制。

在数据通道的服务器端退出时，客户端也应当退出。为了程序的独立性，数据通道应当有自己的通断检测程序，而不是依赖与命令通道的通断检测。

由于数据通道不需要接收从人机界面发来的数据，且数据上传只在硬件产生数据，并且人机界面要求其上传输数据时才发生。因此客户端的通断检测与服务器端不同，和命令通道相比，除了要有一个统一管理网络通断的任务 netCheckLink()之外，还需要一个随时通过接收数据来监视网络的任务 netCheckLinkByRecv()。由于实际上数据通道上没有下传

的数据，因此这个任务一直被阻塞，不占用 CPU，只是在人机界面的服务器端强制退出时，recv()函数返回错误而进入运行状态。它和 netCheckLink 之间使用信号灯同步，一旦 recv()返回，就释放信号灯，通知 netCheckLink()关闭客户端。

客户端的建立和关闭与服务器端不同，客户端只需要建立一个套接字，并用这个套接字连接服务器端，连接成功后就用此套接字作为通信用套接字。关闭时也只需要关闭此套接字。关闭后的再次连接应当由命令通道控制，因此关闭后也不需要再次连接。

在连接的时候，可以采用 connectWithTimeout()函数定时连接，以保证连接时不会无限期阻塞任务。

例 3-4 给出了客户端网络检查程序的部分函数，为了节省篇幅，同样使用注释的方式说明被省略的部分，用户实际使用的时候可以根据情况再添加。

例 3-4　客户端的网络监控程序(vxClt.h 和 vxClt.c)

```
/*************************************************************************
 chengjy@felab, copyright 2002-2004
 vxClt.h
 客户端程序常数定义
 *************************************************************************/
#ifndef _VXCLT_H
#define _VXCLT_H

#define REMOTE_SERVER_IP            "210.45.72.100"
#define REMOTE_SERVER_PORT          7600

/*定义任务名*/
#define TNAME_NETCLTCHECKLINK       "tCltCheckLink"
#define TNAME_NETCLTCHKBYRECV       "tCltCheckByRecv"
#define TNAME_NETCLTCLOSEALL        "tCltCloseAll"

/*用户任务堆栈大小*/
#define USER_STACK_SIZE             2000

/*taskSpawn 使用的用户任务优先级*/
#define TPRI_NETCLTCLOSEALL         101
#define TPRI_NETCLTCHECKLINK        102
#define TPRI_NETCLTCHKBYRECV        103

#endif   /*_VXCLT_H*/

/*************************************************************************
 chengjy@felab, copyright 2002-2004
 vxClt.c
 VxWorks 下建立客户端的程序框架，包括网络监测模块。
```

```
函数:
            void netCltInit();
            void netCltSend();
            void netCltCloseAll();
            void netCltCheckLink();
            void netCltCheckByRecv();
调用:       无
被调用:     shell 下手动调用
说明:       注意本文档仅仅是框架，用户还需要根据需要添加内容。为了简洁起见，函数
            都设定为无返回。
*****************************************************************************/
#include "vxWorks.h"
#include "taskLib.h"
#include "sockLib.h"
#include "inetLib.h"
#include "ioLib.h"
#include "logLib.h"
#include "string.h"
#include "stdio.h"
#include "netinet\\tcp.h"
#include "semLib.h"

#include "vxClt.h"

int netCltSkt;                  /*客户端通信用套接字*/
SEM_ID semNetCltLink;           /*网络连接出错信号灯*/

void netCltInit();
void netCltSend();
void netCltCloseAll();
void netCltCheckLink();
void netCltCheckByRecv();

/*****************************************************************************
 void netCltInit();
 函数描述:   客户端初始化，建立到服务器的通信通道
 参数:       无
 返回:       无
 调用:
            void netCltCheckLink();
            void netCltCheckByRecv();
 被调用:
            命令通道的连接命令。
*****************************************************************************/
```

```
void netCltInit()
{
    int      mlen;
    int      i;
    struct   sockaddr_in  serverAddr;
    struct   timeval     connectTimeOut;
    int      sockAddrSize;
    char     remoteIP[20];
    UINT portNum;
    UINT maxPort;
    char optval = 1;

    /*新建socket*/
    if ((netCltSkt = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
    {
        logMsg("geNetDataInit: unable to create socket\n",0,0,0,0,0,0);
        return;
    }

    /*建立服务器端指定地址*/
    sockAddrSize = sizeof (struct sockaddr_in);
    bzero ((char *) &serverAddr, sockAddrSize);
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_len = (u_char) sockAddrSize;
    serverAddr.sin_addr.s_addr = inet_addr (REMOTE_SERVER_IP);
    serverAddr.sin_port = htons (REMOTE_SERVER_PORT);

    /*设置连接超时时间*/
    connectTimeOut.tv_sec   = 2;    /*2s*/
    connectTimeOut.tv_usec  = 0;    /*0ms*/

    /*连接服务器*/
    logMsg("geNetDataInit: connect %s\n port %d......\n",
           REMOTE_SERVER_IP,REMOTE_SERVER_PORT,0,0,0,0);
    if(connectWithTimeout(netCltSkt,(struct sockaddr *) &serverAddr,
                          sockAddrSize,&connectTimeOut) == ERROR)
    {
        logMsg("failed\n",0,0,0,0,0,0);
        close(netCltSkt);
        return;
    }
    else
    {
          logMsg("succeeded!\n",0,0,0,0,0,0);
```

```
            /*所有的数据都必须立刻发送*/
            setsockopt (netCltSkt, IPPROTO_TCP, TCP_NODELAY, &optval, sizeof
                        (optval));
    }

    if(taskNameToId(TNAME_NETCLTCHECKLINK)==ERROR)
    {
        taskSpawn(TNAME_NETCLTCHECKLINK,TPRI_NETCLTCHECKLINK,0,
            USER_STACK_SIZE,(FUNCPTR)netCltCheckLink,0,0,0,0,0,0,0,0,0,0);
    }
}

/*************************************************************************
 void netCltSend()
 函数说明:    发送数据到服务器端。
 参数:        无
 返回:        无
 调用:        无
 被调用:      需要发送数据的程序。
 *************************************************************************/
void netCltSend()
{
    if(send(netCltSkt,0,0,0/*替换成实际的参数*/)==ERROR)
    {
        semGive(semNetCltLink);
    }
}

/*************************************************************************
 void netCltCloseAll()
 函数描述:    客户端退出，结束 socket 通信。
 参数:        无
 返回:        无
 调用:        无
 被调用:      void netCltCheckLink();
 *************************************************************************/
void netCltCloseAll()
{
    int taskId;

    /*关闭客户端通信 socket*/
    close(netCltSkt);

    /*删除监控任务*/
```

```
    taskId = taskNameToId(TNAME_NETCLTCHECKLINK);
    if(taskId!= ERROR && taskId != taskIdSelf())
        taskDelete(taskId);

    logMsg("geNetDataCloseAll: socket closed\n",0,0,0,0,0,0);
}

/*************************************************************************
 void netCltCheckLink()
 函数说明:    检测网络是否正常，如果不正常，关闭客户端。
 参数:        无
 返回:        无
 调用:        void netCltCheckByRecv()
 被调用:      void netCltInit()
 *************************************************************************/
void netCltCheckLink()
{
    int taskId;

    /*创建信号灯，并发起通过接收数据监视网络的任务*/
    semNetCltLink= semBCreate(SEM_Q_FIFO,0);
    if(taskNameToId(TNAME_NETCLTCHKBYRECV)==ERROR)
    {
        taskSpawn(TNAME_NETCLTCHKBYRECV,TPRI_NETCLTCHKBYRECV,0,
        USER_STACK_SIZE, (FUNCPTR)netCltCheckByRecv,0,0,0,0,0,0,0,0,0,0);
    }

    /*如果获得了信号灯，表示网络已经断开*/
    if(semTake(semNetCltLink,WAIT_FOREVER)==OK)
    {
        taskId = taskNameToId(TNAME_NETCLTCHKBYRECV);
        if(taskId!= ERROR && taskId != taskIdSelf())
            taskDelete(taskId);
        semDelete(semNetCltLink);

        /*调用 netCltCloseAll 关闭客户端*/
        taskSpawn(TNAME_NETCLTCLOSEALL,TPRI_NETCLTCLOSEALL,0,
            USER_STACK_SIZE, (FUNCPTR)netCltCloseAll,0,0,0,0,0,0,0,0,0,0);
    }
}

/*************************************************************************
 void netCltCheckByRecv()
 函数说明:    通过 recv()监测网络，一旦不正常就释放信号灯通知 netCltCheckLink()。
```

```
参数:        无
返回:        无
调用:        无
被调用:      void netCltCheckLink()。
*************************************************************************/
void netCltCheckByRecv()
{
    char buff[2];
    int buffLen;
    while(1)
    {
        buffLen= recv(netCltSkt,buff,1,0);
        if(buffLen==0 || buffLen==ERROR)
        {
            semGive(semNetCltLink);
            break;
        }
    }
}
```

3.4 总结

本章主要介绍了 VxWorks 下的网络编程。首先介绍基于 server 与 client 的基本通信模式，给出了一个用 VxWorks 作为 server、控制端作为 client 的例程，以及与其配套的 PC 作为控制端的软件。接下来将大量篇幅放在双缓冲队列网络通信的介绍上，给出了双缓冲通信的基本运行模式、各任务的优先级设置，以及缓冲队列的具体实现，并讨论了双缓冲队列对减小软件通信死时间的贡献。最后讨论了软件对网络的监控，包括 VxWorks 下 server 和 client 的网络监控程序，并给出了 C 语言实现的框架。

第4章　与控制端交流——通信协议

网络连接建立以后，服务器端和客户端之间就可以互相发送信息了。如果使用 socket 通信，则用户不需要关心 TCP/IP 层以下的协议，编程使用的 recv()和 send()操作的信息流就是字符串。为了使通信双方可以互相理解对方传递过来的字符串，必须事先建立通信协议。规定特定位置上字符所表示的含义，以保证下传命令和返回信息能够得到正确的接收和解释。

4.1　通信协议格式

采集板软件和人机界面软件之间通常会建立两条通道：一条是用以控制的命令通道；另一条是采集板用来上传采集到数据的数据通道。由于数据通道的通信双方都明确地知道将要发送数据的大小、顺序和格式，因此不需要设定通信协议。而在命令通道上发送的是各种各样的控制信息、返回信息和报警信息，各条信息的发送时间、长度和格式均不同，因此必须有对应的通信协议。

网络中一次数据的发送和接收长度受到 TCP/IP 协议的限制，数据量较大时需要将一组数据分多次发送，一组数据中一次发送的数据称为一帧。为了使接收方能够确定一组数据的总量，在纯数据之前，加上 4byte 表示纯数据的总长度；为了使接收方能够区分信息的类别，再加上 2byte 的控制信息，称为命令号；此外，还需要 1byte 的长度来表示信息的紧急程度，默认值 0 表示普通，1 表示紧急；最后，为了程序将来的扩展，再加上 1byte 的备用长度。设定一次发送或者接收的帧长度最大为 4096byte。每一组数据的前 8byte 格式固定，byte7 之后的数据也被称为命令参数，是长度不固定、解释方式由命令号决定的纯数据。

通信协议中实际使用的基本帧格式见图 4-1。

当纯数据长度小于或等于 4088byte 时，只有首帧；当纯数据长度大于 4088 且小于等于 8184byte 时，数据分为首帧和尾帧两帧；当纯数据长度大于 8184byte 时，首帧和尾帧中间夹着若干长度固定为 4096byte 的中间帧。

举例来说，如果需要传送长度为 4byte、值为 0x12345678 的数据，命令号为 0x0100，非紧急，则只需要发送首帧，待发送 unsigned char 型数组的长度为 12，字节序号从 0~11 对应存储的内容按顺序为：0x00,0x00,0x00,0x0C,0x01,0x00,0x00,0x00,0x12,0x34,0x56,0x78。

在通信协议中应当给出一个详细的命令列表，由于命令分为两个 byte，因此可以利用前 1byte 作为命令的分类，后 1byte 作为每个分类的详细信息。

通信协议中还应当包括每个命令的详细解释，包括命令传递的方向、参数长度、参数详细含义、命令的返回情况。可以把每个命令看作是一个通过网络方式执行的函数，所有函数需要的说明都应该在通信协议中找到对应部分。

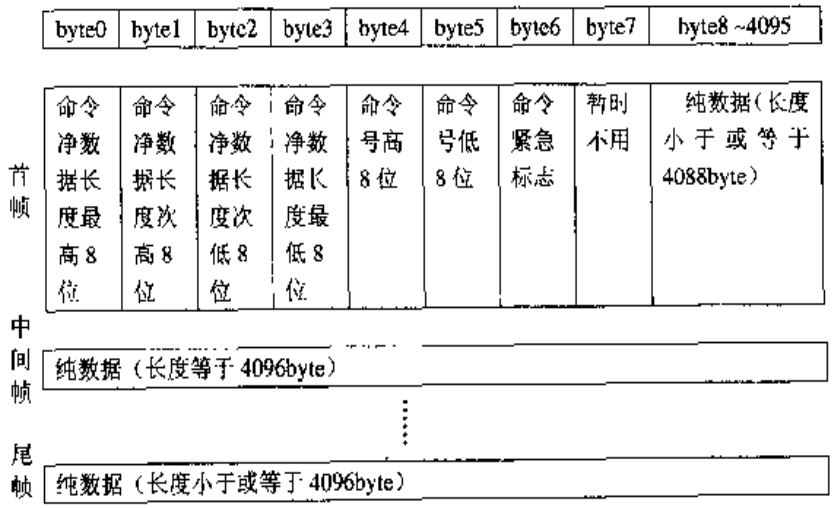


图 4-1　通信中数据的基本格式

一般来说，命令由人机控制界面发起，下传给采集板，采集板执行相应的操作后，返回操作的成功与否信息给人机界面。或者是人机界面需要查询采集板的状态，下传命令后采集板返回状态信息。还有一类比较特殊的情况，即采集板上发生了正常运转之外的特殊情况，例如有严重的硬件错误发生，此时必须立即报告人机界面。为了方便起见，也将其称为命令，并且将这种命令也归到通信协议的命令中，由于这种命令发出时间的完全随机，因此需要人机界面软件的特殊注意。

在通信协议的制订中，由于通信的双方是运行在不同 CPU 上的不同操作系统，双方面对的仅仅是网络，要使通信表达得准确，就需要注意下面的问题。

- 多字节参数的存储顺序

不同的操作系统的字节高低位可能不同，例如 int 型的 305419896(0x12345678)在不同的操作系统下，从地址低位到高位可能是 0x12/0x34/0x56/0x78 或者 0x78/0x56/0x34/0x12，且 1byte,2byte,4byte 混合存储的结构体的具体存储方式也随着操作系统的不同而不同。因此，最好不要期望通过在一个操作系统下定义结构体，直接用此结构体的指针作为网络发送函数 send()的参数来送数据，并在另一个操作系统下定义同样结构体，用结构体指针作为网络接收函数 recv()的参数来获得数据的方法来直接传递结构体。所有面对网络的数据都应该是 unsigned char 型的、unsigned short 或者 unsigned int 型的数据应当在通信协议中规定高低位对应的字节序号，并由两个操作系统各自在网络的两端进行数据的拆解和组装。这里讨论的通信协议采用了高位在前、低位在后的排列方式，这样以字符形式接收后，数据的高位存储在低地址。所有长度超过 1byte 的参数都采取这种拆分方式。

- 负数、浮点数、长整型的存储方法

对于带符号的数据，应当注意操作系统实际采用的反码补码方法。大部分的嵌入式操作系统的单个变量最长为 32bit，不支持浮点型和长整型（64bit）。因此，通信协议中涉及到这两种数据时需要注意。虽然嵌入式操作系统可以通过建立两个 32bit 数来模拟浮点和长整型的计算，但算法复杂且效率低，因此如果不是特殊需要，避免将浮点型或长整型数据以某种

格式传递给嵌入式操作系统，而将关于这些数据的计算放在人机界面软件中完成。

- 从命令下传到命令执行结束的时间

查询式命令必须有状态返回，而纯粹的命令可能不需要执行结果返回，但如果某条命令的执行需要很长的时间，则最好在命令执行结束时从网络返回信息，表示命令已经执行完毕。控制端界面借此对某些按钮进行使能/非使能控制，防止用户由于无法了解各条命令的执行情况，而发出重复的或者错误的命令。

- 采集板出错的紧急报告

通过给每一条命令都设置返回值，可以提供给控制端软件足够的信息，保证其在一条命令结束之前不发出下一条命令，但这并不能保证控制端不使用接收缓冲队列就正确完成所有的信息处理。采集板出错的紧急报告信息对于控制端完全是不可控制的随机事件，因此还是需要建立缓冲队列来解随机，再由从缓冲队列读出信息的线程负责将信息通知到等待返回或响应硬件错误的程序其他部分。由于紧急报告信息和绝大多数的命令循环方式不同，是直接的、随机的、从下到上发生的事件，因此需要特殊注意。

- 校验

可以看到，在上面的通信协议框架中没有任何的校验，也就是说，当信息出现丢失 byte、多出 byte、byte 错位、bit 出错等问题时，接受端仍然把它当成正确的信息来处理，由于 TCP/IP 网络有自己的纠错，因此如果系统没有特殊的误码率规定，直接用不带校验的通信协议即可，这样节省网络开支且可以省略纠错计算，但对于电信要求的连续运行系统则需要进行校验，可以利用首帧的 byte7 来存放校验码。

4.2 VxWorks 端的命令接收、处理和发送

在第三章中已经有所介绍，VxWorks 端将采用信息接收任务、信息解释任务和信息发送任务联合实现命令的接收、处理和发送，并给出了缓冲队列的代码级实现，本节将介绍如何具体地结合缓冲队列，编写这三个任务的代码。

首先看命令接收。所有的命令对于接收端都可以看成一个字符串，其长度可变，但可以通过前几个 byte 进行判断。命令接收任务并不区分各个任务，而是尽量快地将网络传来的所有字符接收，并按照前 4byte 规定的长度将其单次或多次接收的单个命令存放在内存中，并在接收缓冲队列中添加一个元素，用来存放此块内存的首地址。为了通知信息解释任务有命令需要执行，使用计数型信号灯进行同步。每当一个命令被完整地接收并填写到缓冲队列中后，就释放信号灯，信号灯的值对应于当前缓冲队列中还没有被执行的命令的个数。队列添加可以在队头（byte6 信息紧急标志等于 1）、也可以在队尾（byte6 信息紧急标志等于 0）。

由于嵌入式软件无法知道控制端发送命令的个数、频率和长度，因此不能依靠申明全局数组的方法来获得存储命令的空间，在本章的介绍中，命令接收函数使用 malloc 来创建存放命令的空间。这种方法的优势是程序简洁，且可以在内存足够大的情况下缓存尽量多的命令，但缺点是会造成内存碎片（第十章中会介绍针对 VxWorks 的用户内存控制，使用用户保留内存代替 malloc()获得内存的方法）。

命令执行任务首先获取信号灯，获取成功说明有需要执行的任务。从缓冲队列的队头取出一个元素，根据其中的内存首地址获得需要执行的命令的具体信息，包括命令号、命令长度和命令参数。注意到通信协议中的 byte6 信息紧急标志不需要在这里解析。由于命令接收任务已经将紧急命令添加在队头，而命令执行只从队头获得命令，因此紧急命令将被优先执行，多个紧急命令实行的是堆栈的后进先出调度，也就是说，控制端最后发出的紧急命令将最先被执行。而对于非紧急命令，命令接收任务将其添加在队尾，因此比紧急命令后执行，多个非紧急命令的调度使用的是 FIFO 的先进先出，即控制端先发出的非紧急命令先执行。非紧急命令一般用于常规控制，这种情况下命令按步执行；而紧急命令则用于非常规情况，例如撤消一个已经被发出但还没有被执行的命令。

每个命令号都应当拥有对应的函数，格式为 char t[命令号](unsigned char *pBuff)，例如命令号 0x0100 对应的函数为 char t0x0100(unsigned char *pBuff)。

命令执行任务根据命令号调用相应函数，并直接将自己的从队列中获得的内存首地址作为参数传递给这个函数，而由函数来进行具体操作。命令执行任务还负责从函数的返回值中了解是否成功地执行了操作，然后调用 free()释放由命令接收任务 malloc()申请的内存。

如果需要网络返回，这些函数需要 malloc 出适当内存，将需要发送的信息按照通信协议规定的格式填入内存，并将内存的首地址添加到发送缓冲队列，然后类似与信息接收任务和信息解释任务的同步方法，使用信号灯与信息发送任务同步。

由于硬件的中断或者轮询同样可能要发送信息到控制端，因此也会对信息发送队列进行处理。为了统一程序便于管理，使用唯一的函数进行信息到发送缓冲队列的添加，而命令执行任务和硬件中断都调用此函数来实现网络发送。

信息发送任务则循环获取信号灯，获取成功后从信息发送缓冲取出一个元素，发送结束后调用 free()释放对应的内存。

各任务和函数间的信号灯同步见图 4-2。

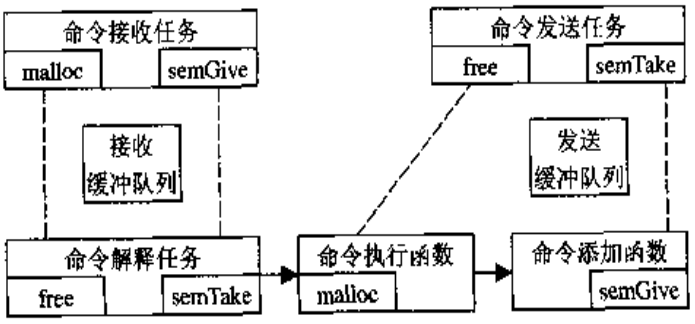


图 4-2　网络接收、处理和发送各任务间的信号灯同步

4.3　VxWorks 端命令通道通信实例

例 4-1 给出了 VxWorks 端的命令接收、处理和发送的具体函数，结合上一章的 Server 端和缓冲队列基本实现的程序，构成了一套完整的网络操作函数。如果需要服务器保持随

时准备好通信，则还需要加上网络监控的部分。为了节省篇幅，与前面相同的 netQueue.c 不再列出。缓冲队列的宏定义合并到采集板参数的宏定义文件 board.h 中。所有命令的具体操作函数集中放在了文件 netTask.c 中。如果用户需要添加新的命令，只需要在 netTask.c 和 netSvr.c 中仿照 t0x0200()等，添加对应函数和函数调用即可。

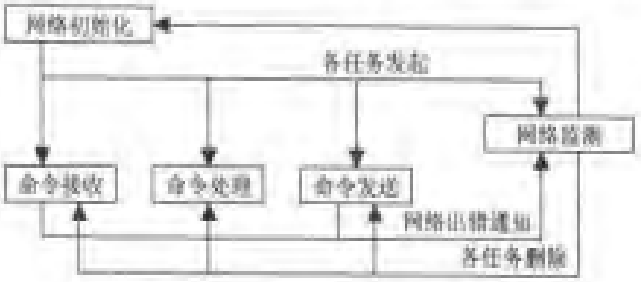


图 4-3 采集板命令通道软件的整体结构

软件的运行时的整体结构见图 4-3。

在光盘中还给出了配合其使用的 VC 程序 WinCltAll。

运行步骤如下：

（1）启动 VxWorks，在 shell 下运行 sp netInit，初始化服务器。

（2）运行 WinCltAll，在 IP 和 Port 端口填入 VxWorks 下服务器的 IP 和端口号，点击“连接”。

（3）如果连接成功，则 WinCltAll 程序下端两个“发送”按钮被激活，“连接”按钮被禁止，此时可以分别发送长帧或短帧到 VxWorks。长帧发送的参数被存储在硬盘的文件中，短帧发送的参数直接在界面里改写。对于长帧发送，如果填入的命令长度小于或等于 8，则将发送文件的全部内容，否则发送对应的长度。对于短帧发送，如果填入的命令长度小于 8 或者大于 12，将发送全部 4 个参数，否则根据命令长度对应发送。最右边的显示命令的返回情况。

（4）直接关闭 WinCltAll，VxWorks 将自动准备好迎接 WinCltAll 的下次连接。

程序的运行结果见图 4-4。

图 4-4 结合通信协议的网络程序运行结果

例 4-1 VxWorks 下包括通信协议支持和网络监测支持的完整网络通信程序(board.h, netQueue.c, netSvr.c 和 netTask.c)

```
/*************************************************************************
 chengjy@felab, copyright 2002-2004
 board.h
 采集板用户程序常数定义
 *************************************************************************/
#ifndef _BOARD_H
#define _BOARD_H

/*---------------网络命令通道缓冲队列----------------*/
/*队列的单节结构*/
struct cmdSingle
{
    struct cmdSingle *pNext;
    unsigned char *pBuff;
};

/*定义队列添加的优先级*/
#define QUEUE_PRI_HIGH          1    /*队头*/
#define QUEUE_PRI_LOW           0    /*队尾*/

/*定义队列的个数，对于双缓冲队列，定义为 2*/
#define QUEUE_NUM               2    /*必须是大于 0 的整数*/
#define QUEUE_INDEX_RECV        0
#define QUEUE_INDEX_SEND        1

/*定义返回状态*/
#define STATUS_NORMAL           0x00
#define STATUS_ERROR            0x01
#define STATUS_WARNING          0x02
#define STATUS_INVALID          0x03
#define STATUS_UNAVAILABLE      0x04
#define STATUS_DISABLE          0x05

/*本地的网络端口号*/
#define LOCAL_SERVER_PORT       2001     /*服务器端套接字的端口号*/

/*命令通道和数据通道的最大包长度*/
#define NET_MSG_MAX_SIZE        0x1000  /*4096bytes*/

/*服务器初始化情况*/
#define NET_INIT_LISTENSKT      0x01
#define NET_INIT_COMMUSKT       0x02
```

```
#define NET_INIT_NULL           0x00

/*定义网络关闭时是否要重新初始化*/
#define MODE_NET_DEFAULT        0    /*重新初始化*/
#define MODE_NET_REINIT         1        /*不再初始化*/

/*定义任务名*/
#define TNAME_NETRECV           "tNetRecv"
#define TNAME_NETEXPLAIN        "tNetExplain"
#define TNAME_NETSEND           "tNetSend"
#define TNAME_NETINIT           "tNetInit"
#define TNAME_NETCHECKLINK      "tNetCheckLink"
#define TNAME_NETCLOSEALL       "tNetCloseAll"

/*用户任务堆栈大小*/
#define USER_STACK_SIZE     2000

/*taskSpawn 使用的用户任务优先级*/
#define TPRI_NETRECV            114
#define TPRI_NETEXPLAIN         116
#define TPRI_NETSEND            114
#define TPRI_NETCLOSEALL        110
#define TPRI_NETINIT            112
#define TPRI_NETCHECKLINK       110

#define LOG_NETMSG_HEAD /*打印接收到的网络消息的前 8BYTE*/
#undef LOG_NETMSG_HEAD

#endif   /*_BOARD_H*/

/*****************************************************************************
 chengjy@felab, copyright 2002-2004
 netQueue.c
 参见第 3 章例 3-2
*****************************************************************************/

/*****************************************************************************
 chengjy@felab, copyright 2002-2004
 netSvr.c
 vxWorks 作为 server，双缓冲网络通信，带网络监控，同时支持通信协议
 函数:
     void netInit(int mode);
     void netCMDRecv();
     void netCMDExplain();
```

```
    void netCMDSend();
    void netCloseAll(int mode);
    void netCheckLink();
    char netCMDAdd(unsigned char *pBuff, int buffLen, int cmdNum,
                   unsigned char priority);
    char netRecvSize(unsigned char *pBuff, int len);
调用:
    netTask.c
        extern char t0x0200(unsigned char *pBuff);
        extern char t0x0400(unsigned char *pBuff);
        extern char t0x1600(unsigned char*pBuff);
    netQueue.c
        extern void queueInit();
        extern char queueAdd(int index, unsigned char* pBuff, int pri);
        extern char queueDelHead(int index);
        extern char queueDelAll();
被调用:
        用户程序初始化部分
        netTask.c
*****************************************************************************/

#include "vxWorks.h"
#include "taskLib.h"
#include "sockLib.h"
#include "ioLib.h"
#include "inetLib.h"
#include "logLib.h"
#include "string.h"
#include "fioLib.h"
#include "stdio.h"
#include "memLib.h"
#include "stdLib.h"
#include "semLib.h"
#include "netinet\\tcp.h"

#include "board.h"

/*通信协议支持的命令*/
extern char t0x0200(unsigned char *pBuff);      /*参数配置*/
extern char t0x0400(unsigned char *pBuff);      /*reset*/
extern char t0x1600(unsigned char*pBuff);       /*flash程序下载*/

/*缓冲队列操作*/
extern void queueInit();
```

```
extern char queueAdd(int index, unsigned char* pBuff, int pri);
extern char queueDelHead(int index);
extern char queueDelAll();
extern struct cmdSingle *queueHead[QUEUE_NUM];

int flagNetInit = NET_INIT_NULL; /*服务器初始化标志,程序中省略了对应的判断和操作*/
int listenSkt;                   /*命令通道侦听 socket*/
int commuSkt;                    /*命令通道通信 socket*/
SEM_ID semQueueRecv;             /*网络接收缓冲队列信号灯*/
SEM_ID semQueueSend;             /*网络发送缓冲队列信号灯*/
SEM_ID semCmdLink;               /*网络连接出错信号灯*/

void netInit(int mode);
void netCMDRecv();
void netCMDExplain();
void netCMDSend();
void netCloseAll(int mode);

void netCheckLink();

char netCMDAdd(unsigned char *pBuff, int buffLen, int cmdNum, unsigned char
priority);
char netRecvSize(unsigned char *pBuff, int len);

/*************************************************************************
 void netInit(int mode)
 函数说明:    网络初始化程序
 参数:        mode,   分为 MODE_NET_DEFAULT 和 MODE_NET_REINIT 两种,分别对应初始
                      化侦听+通信套接字和只初始化通信套接字两种。
 返回:        无
 调用:
              void netCMDRecv();
              void netCMDExplain();
              void netCMDSend();
              void netCheckLink();
          netQueue.c:
              extern void    queueInit();
 被调用:      用户程序初始化模块
          void netCloseAll(int mode);
 *************************************************************************/
void netInit(int mode)
{
    struct  sockaddr_in  serverAddr;
    struct  sockaddr_in  clientAddr;
```

```
        int     sockAddrSize;
        int     i;
        char    optval = 1;

        /*建立本地的侦听用套接字 listenSkt 并bind和listen*/
        if(flagNetInit==NET_INIT_COMMUSKT)
            return;
        if((mode == MODE_NET_DEFAULT) && (flagNetInit == NET_INIT_NULL))
        {
            if((listenSkt = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
            {
                logMsg("netInit: can not open listen socket\n",0,0,0,0,0,0);
                return;
            }
            sockAddrSize = sizeof (struct sockaddr_in);
            bzero ((char *) &serverAddr, sockAddrSize);
            serverAddr.sin_family = AF_INET;
            serverAddr.sin_len = (u_char) sockAddrSize;
            serverAddr.sin_port = htons (LOCAL_SERVER_PORT);
            serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);
            if (bind (listenSkt, (struct sockaddr *) &serverAddr, sockAddrSize)
== ERROR)
            {
                logMsg("netInit:unable to bind to port %d\n",LOCAL_SERVER_
                        PORT,0,0,0,0,0);
               close(listenSkt);
               return;
            }
            logMsg("netInit: successfully bind to port\n",LOCAL_SERVER_
                    PORT,0,0,0,0,0);
            if (listen (listenSkt, 1) == ERROR)
            {
                logMsg("netInit: can not listen to listen socket\n",0,0,0,0,0,0);
                close (listenSkt);
                return;
            }
            flagNetInit=NET_INIT_LISTENSKT;
        }

        /*接收外部连接，建立通信套接字 commuSkt*/
        if(flagNetInit==NET_INIT_LISTENSKT)
        {
            sockAddrSize = sizeof (struct sockaddr_in);
            commuSkt = accept(listenSkt, (struct sockaddr*)(&clientAddr),
```

```
    &sockAddrSize);
            if(commuSkt==ERROR)
            {
                logMsg("netInit: can not accept command socket\n",0,0,0,0,0,0);
                close (listenSkt);
                return;
            }
            setsockopt (commuSkt, IPPROTO_TCP, TCP_NODELAY, &optval, sizeof
(optval));
            /*初始化网络通信的缓冲对列*/
            queueInit();
            /*初始化缓冲队列的计数型信号灯*/
            semQueueSend = semCCreate(SEM_Q_FIFO,0);
            semQueueRecv = semCCreate(SEM_Q_FIFO,0);
            if((semQueueSend==NULL) ||(semQueueRecv==NULL))
            {
                semDelete(semQueueSend);
                semDelete(semQueueRecv);
            }
            /*发起网络命令循环接收任务、命令解释任务、命令发送任务和网络监控任务*/
            taskSpawn(TNAME_NETCHECKLINK,TPRI_NETCHECKLINK,0,
                USER_STACK_SIZE,(FUNCPTR)netCheckLink,0,0,0,0,0,0,0,0,0,0);
            taskSpawn(TNAME_NETRECV,TPRI_NETRECV,0,
                USER_STACK_SIZE, (FUNCPTR)netCMDRecv,0,0,0,0,0,0,0,0,0,0);
            taskSpawn(TNAME_NETEXPLAIN,TPRI_NETEXPLAIN,0,
                USER_STACK_SIZE, (FUNCPTR)netCMDExplain,0,0,0,0,0,0,0,0,0,0);
            taskSpawn(TNAME_NETSEND,TPRI_NETSEND,0,
                USER_STACK_SIZE, (FUNCPTR)netCMDSend,0,0,0,0,0,0,0,0,0,0);
            flagNetInit = NET_INIT_COMMUSKT;
            logMsg("netInit: netInit finished\n",0,0,0,0,0,0);
        }
    }

    /*************************************************************************
     void netCMDRecv()
     函数说明:    信息接收任务。循环从网络获得信息，打包后添加到缓冲队列，等待信息解释
            任务处理。
     参数:        无
     返回:        无
     调用:
                  char netRecvSize(unsigned char *pBuff, int len)
            netQueue.c
                  char queueAdd(int index, unsigned char* pBuff, int pri);
     被调用:
```

```
            void netInit()
 ***************************************************************************/
void netCMDRecv()
{
    unsigned char buff[8];
    unsigned char *pBuff;
    int lenAll;
    char pri;
    char state=STATUS_NORMAL;
    int i;

    /* 循环接收*/
    while(1)
    {
    if(netRecvSize(buff,8)!=STATUS_NORMAL)
    {
        logMsg("netCMDRecv: error in head Recv netRecvSize(buff,8)\n",
                0,0,0,0,0,0);
        break;
    }
    else
    {
#ifdef LOG_NETMSG_HEAD
        printf("netCMDRecv: print out head 8 bytes\n");
        for(i=0;i<8;i++)
            printf("0x%02x ",buff[i]);
        printf("\n");
#endif
        /*计算需要缓冲的总长度*/
        lenAll = ((buff[2]*0x1000000)&0xFF000000)+((buff[3]*0x10000)
                  &0xFF0000)+((buff[4]*0x100)&0xFF00)+buff[5];
                  /*单条命令的总长度*/
        pBuff = malloc(lenAll*sizeof(char));
        if(pBuff!=NULL)
        {
            memcpy(pBuff,buff,8*sizeof(char));
            if(lenAll>8)
            {
                state = netRecvSize(pBuff+8,lenAll-8);
                if(state!=STATUS_NORMAL)
                {
                    logMsg("netCMDRecv: error in rear part recv
                           netRecvSize(pBuff,%d)\n",lenAll-8,0,0,0,0,0);
                    break;
```

```
                }
            }
            /*正确接收了所有数据，添加到接收缓冲队列中*/
            if(state== STATUS_NORMAL)
            {
                queueAdd(QUEUE_INDEX_RECV, pBuff, buff[6]);
                semGive(semQueueRecv);
            }
        }
    }
}
logMsg("netCMDRecv: something wrong, now exit\n",0,0,0,0,0,0);
}

/*************************************************************************
 void netCMDExplain()
 函数说明:    主循环消息处理和状态发送任务。循环获取信号灯获取成功后，根据命令号执
              行相应任务。
 参数:        无
 返回:        无
 调用:
        netQueue.c
            char queueDelHead();
        netTask:
            STATUS t0xXXXX(char *pBuff);
 被调用:
            void netInit()
 *************************************************************************/
void netCMDExplain()
{
    unsigned char *pBegin;
    int CMDNum;
    char    state;
    while(semTake(semQueueRecv,WAIT_FOREVER)== OK)
    {
        /*从队头取出信息*/
        pBegin = queueHead[QUEUE_INDEX_RECV]->pBuff;
        CMDNum = pBegin[0]*0x100+pBegin[1];
        /*这里只是示意性地做了几个命令的解释，用户应该根据自己的通信协议再添加内容*/
        switch(CMDNum)
        {
            case 0x0200:
                state = t0x0200(pBegin);
                break;
```

```
            case 0x0400:
                state = t0x0400(pBegin);
                break;
            case 0x1600:
                state = t0x1600(pBegin);
                break;
            default:
                logMsg("netCMDExplain: command 0x%04x is not supported\n",
                       CMDNum,0,0,0,0,0);
                break;
        }
        if(state == STATUS_NORMAL)
            logMsg("netCMDExplain: result of command 0x%04x is *OK*\n",
                   CMDNum,0,0,0,0,0);
        else
            logMsg("netCMDExplain: result of command 0x%04x is *ERROR*\n",
                   CMDNum,0,0,0,0,0);

        free(pBegin);
        queueDelHead(QUEUE_INDEX_RECV);
    }
    logMsg("netCMDExplain: error in taking semQueue\n",0,0,0,0,0,0);
}

/*************************************************************************
void netCMDSend()
函数说明:    循环从网络发送缓冲队列中取出数据并发送。
参数:        无
返回:        无
调用:
        netQueue.c
            char queueDelHead(int index);
被调用:
        netTask.c
            char t0xXXXX(char *pBuff);
*************************************************************************/
void netCMDSend()
{
    unsigned char *pBegin;
    int lenAll,remainLen,realLen;
    while(semTake(semQueueSend,WAIT_FOREVER)!=ERROR)
    {
        /*从队头取出信息*/
        pBegin = queueHead[QUEUE_INDEX_SEND]->pBuff;
```

```
            lenAll = ((pBegin[2]*0x1000000)&0xFF000000)+((pBegin[3]*0x10000)
&0xFF0000)+((pBegin[4]*0x100)&0xFF00)+pBegin[5]; /*单条命令的总长度*/

            /*循环发送，直到发送完毕或者出错为止*/
            remainLen = lenAll;
            while(remainLen>0)
            {
                if(remainLen>NET_MSG_MAX_SIZE)
                {
                    realLen = send(commuSkt,pBegin+lenAll-remainLen,
                                   NET_MSG_MAX_SIZE,0);
                }
                else
                {
                    realLen= send(commuSkt,pBegin+lenAll-remainLen,remainLen,0);
                }
                if(realLen== ERROR)
                {
                    logMsg("netCMDSend: unable to send command\n",0,0,0,0,0,0);
                    semGive(semCmdLink);
                    break;
                }
                remainLen = remainLen-realLen;
            }

            /*结束发送，释放空间*/
            free(pBegin);
            queueDelHead(QUEUE_INDEX_SEND);

            if(remainLen!=0)        /*发送出错，中途跳出的情况*/
                break;
        }
        logMsg("netCMDSend: something wrong, now exit\n",0,0,0,0,0,0);
    }

    /*************************************************************************
     void netCloseAll(int mode)
     函数说明：  关闭侦听 socket ，并删除命令缓冲队列以及与 socket 相关的任务，并根据模式的
                 不同选择是直接关闭侦听套接字 (MODE_NET_DEFAULT) 还是重新初始化网络
                 (MODE_NET_REINIT)。
     参数：      mode,   选择关闭方式。
     返回：      无
     调用：      void netInit(int mode);
     被调用：    void netCheckLink();
```

```
                用户在 shell 下调用
 ************************************************************************/
void netCloseAll(int mode)
{
    int taskId;

    /*删除网络相关任务*/
    taskId = taskNameToId(TNAME_NETRECV);
    if(taskId!=taskIdSelf() && taskId!=ERROR)
    {
        taskDelete(taskId);
    }
    taskId = taskNameToId(TNAME_NETSEND);
    if(taskId!=taskIdSelf() && taskId!=ERROR)
    {
        taskDelete(taskId);
    }
    taskId = taskNameToId(TNAME_NETEXPLAIN);
    if(taskId!=taskIdSelf() && taskId!=ERROR)
    {
        taskDelete(taskId);
    }
    taskId = taskNameToId(TNAME_NETCHECKLINK);
    if(taskId!=taskIdSelf() && taskId!=ERROR)
    {
        taskDelete(taskId);
    }

    /*删除缓冲队列*/
    queueDelAll();
    /*删除缓冲队列的信号灯*/
    semDelete(semQueueRecv);
    semDelete(semQueueSend);

    /*删除网络通断检测用信号灯*/
    semDelete(semCmdLink);

    /*根据网络初始化的不同状态作相应的退出工作*/
    if(mode==MODE_NET_DEFAULT)
    {
        close(commuSkt);
        close(listenSkt);
        flagNetInit = NET_INIT_NULL;
    }
```

```
    else if(mode==MODE_NET_REINIT)
    {
        close(commuSkt);
        flagNetInit = NET_INIT_LISTENSKT;
        /*重新初始化网络*/
        logMsg("netCloseAll: reInitialize net server\n",0,0,0,0,0,0);
        taskDelete(taskNameToId(TNAME_NETINIT));
        taskSpawn(TNAME_NETINIT,TPRI_NETINIT,0,USER_STACK_SIZE,
                (FUNCPTR)netInit,mode,0,0,0,0,0,0,0,0,0,0);
    }
}


/*************************************************************************
 void netCheckLink()
 函数说明:    监测网络状态，出错即关闭网络并重新初始化。
 参数:        无
 返回:        无
 调用:
              void netCloseAll(int mode)
 被调用:
              void netInit();
 *************************************************************************/
void netCheckLink()
{
    /*由于优先极高，因此先创建信号灯才会进行网络的接收和发送*/
    semCmdLink = semBCreate(SEM_Q_FIFO,SEM_EMPTY);

    /*等待 send()和 recv()出错释放信号灯*/
    semTake(semCmdLink,WAIT_FOREVER);

    /*获得信号灯，表示网络连接断开*/
    semDelete(semCmdLink);
    taskSpawn(TNAME_NETCLOSEALL,TPRI_NETCLOSEALL,0,USER_STACK_SIZE,
            (FUNCPTR)netCloseAll,MODE_NET_REINIT,0,0,0,0,0,0,0,0,0,0);
}

/*************************************************************************
 char netRecvSize(unsigned char *pBuff, int len)
 函数说明:    底层函数，从网络接收数据。
 参数:        pBuff,   接收数据存储的首地址。
              len,     需要接收的总长度。
 返回:        正确执行返回 STATUS_NORMAL，否则返回错误。
 调用:        无
```

```
    被调用:     void netCMDRecv();
    ************************************************************************/
   char netRecvSize(unsigned char *pBuff, int len)
   {
       int recvLen,remainLen;
       if((len<=0) || (flagNetInit!=NET_INIT_COMMUSKT))
           return(STATUS_INVALID);
       else
           remainLen = len;
       while(remainLen>0)
       {
           recvLen = recv(commuSkt,pBuff+len-remainLen, remainLen,0);
           if( (recvLen==ERROR) || (recvLen ==0) )
           {
               semGive(semCmdLink);
               return(STATUS_ERROR);
           }
           remainLen = remainLen-recvLen;
       }
       return(STATUS_NORMAL);
   }

   /************************************************************************
    char netCMDAdd(unsigned char *pBuff, int buffLen, int cmdNum, unsigned char
priority)
    函数说明:   添加内容到发送缓冲队列。
    参数:       pBuff,      需要经过修改后添加的数组首地址。
                buffLen,    数组长度，单位为byte。
                cmdNum,     命令号
                priority,   优先级
    返回:       添加成功返回STATUS_NORMAL，否则返回错误。
    调用:
           netQueue.c
               char queueAdd(int index, unsigned char* pBuff, int pri)
    被调用:
       netTask.c
           char t0xXXXX(char *pBuff);
    ************************************************************************/
   char netCMDAdd(unsigned char *pBuff, int buffLen, int cmdNum, unsigned char
priority)
   {
       pBuff[0] = (cmdNum&0xFF00)>>8;
       pBuff[1] = cmdNum&0x00FF;
       pBuff[2] = ((buffLen>>24)&0xFF);
```

```
    pBuff[3] = ((buffLen>>16)&0xFF);
    pBuff[4] = ((buffLen>>8)&0xFF);
    pBuff[5] = (buffLen&0xFF);
    pBuff[6] = priority;
    pBuff[7] = 0;

    queueAdd(QUEUE_INDEX_SEND, pBuff, priority);
    semGive(semQueueSend);
    return(STATUS_NORMAL);
}

/*****************************************************************************
 chengjy@felab, copyright 2002-2004
 netTask.c
 总的任务解析。
 所有函数格式统一:
    char t0xXXXX(char *pBuff)
 所有函数都被调用:
    void netCMDExplain();
    所有的函数的返回值都为 char 型，分为 STATUS_NORMAL(成功)和失败(STATUS_ERROR
等)。具体任务内容参见通信协议。本文件仅仅给出软件框架，用户可以根据自己的通信协议添加或
者删除函数，并修改具体的内容。
 调用:
    netSvr.c
        char netCMDAdd(unsigned char *pBuff, int buffLen, int cmdNum,
                       unsigned char priority);
 被调用:
      netSvr.c
        void netCMDExplain();
 *****************************************************************************/
#include "vxWorks.h"
#include "logLib.h"
#include "taskLib.h"
#include "stdio.h"
#include "memLib.h"
#include "stdLib.h"
#include "sysLib.h"
#include "bootLib.h"

#include "board.h"

/*netSvr.c*/
extern char netCMDAdd(unsigned char *pBuff, int buffLen, int cmdNum, unsigned
char priority);
```

```
/*可接受的命令号*/
char t0x0200(unsigned char *pBuff);/*参数配置*/
char t0x0400(unsigned char *pBuff);/*reset*/
char t0x1600(unsigned char*pBuff); /*flash 程序下载*/

/*************************************************************************
char t0x0200(unsigned char *pBuff)
函数描述：获得控制界面发来的配置信息，配置全局变量。
网络返回：执行结束后发送 9byte 帧。
调用：
    netSvr.c
        char netCMDAdd(unsigned char *pBuff, int buffLen, int cmdNum,
                       unsigned char priority);
*************************************************************************/
char t0x0200(unsigned char *pBuff)
{
    char state = STATUS_NORMAL;
    unsigned char *pSendBuff;
    int i, len;
    logMsg("t0x0200: beging...\n",0,0,0,0,0,0);

    /*解析参数并配置参数，根据情况设定 state 参数*/

    /*测试用代码，打印 0x0200 的所有参数*/
    len = pBuff[2]*0x01000000+pBuff[3]*0x00010000+pBuff[4]*0x00000100
          +pBuff[5]-8;
    if(len!=0)
    {
        logMsg("t0x0200: print out %d params for 0x0200\n",len,0,0,0,0,0);
        for(i=0;i<len;i++)
        {
            printf("%c",pBuff[i+8]);
            if(i%60==59)
                printf("\n");
        }
        printf("\n");
    }
    else
    {
        logMsg("t0x0200: 0x0200 has no params\n",0,0,0,0,0,0);
    }
    /*网络返回*/
    pSendBuff = malloc(9*sizeof(char));
```

```
    if(pSendBuff!=NULL)
    {
        pSendBuff[8] = state;
        netCMDAdd(pSendBuff,9,0x0200,QUEUE_PRI_LOW);
    }
    else
    {
        logMsg("t0x0200: unable to malloc net return buff\n",0,0,0,0,0,0);
        state = STATUS_ERROR;
    }

    logMsg("t0x0200: ended\n",0,0,0,0,0,0);
    return(state);
}

/*****************************************************************************
char t0x0400(unsigned char *pBuff)
函数描述：单板重启动。
网络返回：无
调用：    无
*****************************************************************************/
char t0x0400(unsigned char *pBuff)
{
logMsg("t0x0400: beging...\n",0,0,0,0,0,0);
reboot(BOOT_NORMAL);    /*之后的语句不会被执行*/
logMsg("t0x0400: ended\n",0,0,0,0,0,0);
return(STATUS_NORMAL);
}

/*****************************************************************************
char t0x1600(unsigned char*pBuff)
函数描述：将 vxWorks 文件（对于单板来说就时 VxWorks.bin）下载到 Flash，准备下次
        启动时使用。
网络返回：配置后发送发送 9byte 帧。
调用：    外部的 Flash 操作函数。
*****************************************************************************/
char t0x1600(unsigned char*pBuff)
{
    char state;
    unsigned char *pSendBuff;
    int len;

    logMsg("t0x1600: begin...\n",0,0,0,0,0,0);
```

```
        /*len表示VxWorks.bin的总长度*/
        len = pBuff[2]*0x01000000+pBuff[3]*0x00010000+pBuff[4]*0x00000100
              +pBuff[5]-8;
        /*在vxWorks.bin前添加4byte表示总长度信息，一起写入flash*/
        pBuff[4]=(unsigned char)((len>>24)&0xFF);
        pBuff[5]=(unsigned char)((len>>16)&0xFF);
        pBuff[6]=(unsigned char)((len>>8)&0xFF);
        pBuff[7]=(unsigned char)(len&0xFF);

        /*网络返回*/
        pSendBuff = malloc(9*sizeof(char));
        if(pSendBuff!=NULL)
        {
            pSendBuff[8] = state;
            netCMDAdd(pSendBuff,9,0x1600,QUEUE_PRI_LOW);
        }
        else
        {
            logMsg("t0x1600: unable to malloc net return buff\n",0,0,0,0,0,0);
            state = STATUS_ERROR;
        }
        logMsg("t0x1600: ended\n",0,0,0,0,0,0);
        return(state);
    }
```

4.4 总结

本章首先从网络通信建立通信协议的必要性入手，介绍了通信协议的基本形式，并提出了修订用户协议时的几点建议，然后给出了命令的接收、执行和返回的基本模式，介绍了各任务间的信号灯同步方式。最后给出了网络通信的完整例程。在阅读完前一章和本章之后，用户能够建立自己的网络通信，并使用光盘中给出的配套VC程序进行程序调试。

第 5 章　与硬件打交道——定时查询和中断管理

采集板上的嵌入式操作系统向上需要面对控制端软件，向下需要面对采集板上的外围硬件。和控制软件的通信在前面的两章中已经做了介绍，而硬件管理除了最基本的端口操作，还包括轮询和中断管理，本章将介绍的就是这方面的内容。

5.1　硬件的定时查询

如果软件需要从硬件处获得信息，可以采用两种方式：软件定时轮询和硬件中断。前者由软件发起，对软硬件的要求低，硬件只需要将某个信号置高置低，或者将数据填入寄存器，等待软件查询；后者由硬件发起，对软硬件要求高，除了需要达到轮询方式的硬件要求之外，还要求硬件在数据准备好时发起中断，并且提供清中断机制。

软件查询的要求低，相应的性能也低。定时查询的频率必须高于硬件提供数据的频率，才能保证所有的数据都得到响应。由此必然会导致某些次的查询是无效查询，降低了软件的执行效率，特别是在数据的提供比较不规律的情况下，为了保证最近的两次数据提供都得到响应，必然会造成很多的无效查询。而硬件中断方式则不存在这个问题。由于中断是由硬件发起的，软件只是被动地响应和处理中断，因此数据的提供和处理能够做到一一对应，提高了软件效率。但硬件中断方式也有自己的缺点：中断服务程序中不能调用会导致阻塞的函数，且函数的格式必须固定。因此每次中断处理最好分为两部分：中断服务程序和中断服务任务，这样两者之间又存在任务同步的问题，故软件比较复杂。编程者需要根据实际情况选择使用哪种方式来从硬件获得数据。

首先介绍一种最简单的定时查询方法：while(1)+taskDelay()。

用比较高的优先级发起一个任务，此任务的结构为：

```
void dailyTask()
{
    while(!flagQuit)
    {
        /*查询硬件*/
        taskDelay(1 /*改换成每次轮询需要间隔的时间*/);
    }
}
```

其中 flagQuit 是标志任务是否退出的全局变量，默认为 0。如果不再需要查询，可以将其置为 1。也可以通过直接 taskDelete()的方法将轮询任务删除。

此方法的软件最为简单，能够实现不停地查询硬件，但是缺点也随之而来：定时非常不准确。首先，它的定时精度不可能超过一个时间片；其次，如果 taskDelay 在 delay 了足够长的时间后，有其他任务的优先级高于或等于该轮询任务的优先级，则任务虽然脱离 delayed 状态，却不一定能够进入执行状态，每个循环的执行时间是不确定的；最后，每次循环的时间不仅包括 taskDelay 的时间，还包括查询硬件需要的时间，而后者是非常难以计算的。这些因素都将导致循环时间不能精确控制。

使用看门狗 watchdog 定时，可以解决定时不准的问题。VxWorks 提供了看门狗库函数，包含在 wdLib 中，需要包含的头文件为 wdLib.h。

看门狗可以被任意任务创建和调用，用以在一定时间的延迟后运行一个函数。此函数的运行基于系统时钟的中断，而不是任务调度，能够保证严格的定时。

使用看门狗之前，需要先用 wdCreatc()创建看门狗，然后用 wdStart()来定义需要延迟的时间和此时间后需要调用的函数，以及此函数需要使用的参数。在延迟的期间内，如果 wdCancel()被调用了，则不会再调用原定调用的函数。无论调用 wdStart()的任务是在运行，还是被挂起、被删除，都不影响一定时间后函数的调用。需要注意的是，由于看门狗调用函数是中断级的调用，因此被调用的函数中不能使用会导致任务阻塞的库函数，例如不能调用 semTake()以及其他涉及 I/O 系统的函数，其他中断服务程序需要遵循的规则也必须被看门狗所调用的函数遵循。

wdLib 中包含的库函数见表 5-1。

表 5-1　看门狗操作函数

wdCreate()	创建看门狗
wdDelete()	删除看门狗
wdStart()	启动看门狗，在一定时间后调用函数
wdCancel()	取消已经启动的看门狗

下面介绍在本书程序中涉及的看门狗操作函数，更详细的信息可以查阅函数库的在线帮助。

- 函数：WDOG_ID wdCreate()

描述：通过在内存中创建一个 WDOG 结构体的方式创建一个看门狗计时器。

参数：无

返回：成功创建返回看门狗 ID，否则返回 NULL

- 函数：STATUS wdDelete(WDOG_ID wdId)

描述：通过释放 WDOG 结构体来删除看门狗。如果此看门狗已经被调用了 wdStart()，则在一定延迟后调用函数的功能被取消。此函数必须和 wdCreate()配对使用。

参数：

wdId　　看门狗 Id

返回：OK，如果无法删除返回 ERROR

- 函数：STATUS wdStart(WDOG_ID wdId, int delay, FUNCPTR pRoutine, int parameter)

描述：此函数将一个看门狗加到系统 tick 的队列中。当额定个数的时间片过去后，系统从中断级调用函数。此函数可以在中断中调用。为了改变延迟时间或者改变被调用的函数，可以在 wdStart 之后、延迟时间到来之前，重新调用 wdStart。只有最近一次调用的 wdStart 才起实际作用。如果用户程序需要多个看门狗调用程序，则必须创建同样多的看门狗，每个 ID 对应一个函数。看门狗计数器在 wdStart 后，只会调用一次函数，如果需要周期性地执行某个函数，则此函数本身必须调用 wdStart。

参数：

wdId	看门狗 ID
delay	延迟计数，单位为 tick，即延迟时间为 delay/sysClkRateGet()秒
pRoutine	被调用的函数入口
parameter	被调用的函数调用时的参数

返回：OK，如果不成功返回 ERROR

- 函数：STATUS wdCancel(WDOG_ID wdId)

描述：通过将一个正在运行的看门狗计数器的 delay 置为零来撤消其对函数的调用。此函数可以在中断服务程序中使用。

参数：

wdId	看门狗 ID

返回：OK，如果失败返回 ERROR

看门狗可以用于对超时操作的报警，也可以用于删除超时的任务。假设 func1()是一个需要调用运行时间较长函数 funcOfLongTime()的函数，则下面的语句将在 funcOfLongTime()执行时间超过 10s 的情况下打印出错信息，如果 10s 内 funcOfLongTime()返回，则不打印任何信息。

```
WDOG_ID wdogId;
void func1()
{
    wdogId = wdCreate();
    wdStart(wdogId, sysClkRateGet()*10, (FUNCPTR)logMsg,
            "funcOfLongTime has not returned after 10 seconds\n");
    funcOfLongTime();
    wdCancel(wdogId);
}
```

注意到上面的 wdStart 函数的参数使用了 logMsg 而没有使用 printf，其原因就是 printf 会造成 IO 阻塞，不能被中断级的函数调用。

如果还希望将因为调用 funcOfLongTime()而超时的任务 tFunc1（func1 作为入口函数）删除，则书写语句如下：

```
WDOG_ID wdogId;
```

```
void func2(int taskId)
{
    taskDelete(taskId);
}
void func1()
{
    wdogId = wdCreate();
    wdStart(wdogId, sysClkRateGet()*10, (FUNCPTR)func2, taskIdSelf());
        funcOfLongTime();
        wdCancel(wdogId);
}
```

对于需要定时循环调用的函数，采用下面的结构。

```
    WDOG_ID wdDailyFunc;
void dailyFunc(int param)
{
    wdStart(wdDailyFunc, sysClkRateGet()/*更改成需要 delay 的时间片个数*/,
            (FUNCPTR)dailyFunc,param);
    /*需要定时执行的内容*/
}
void initDailyFunc()
{
    wdDailyFunc = wdCreate();
    wdStart(wdDailyFunc, sysClkRateGet()/*更改成需要 delay 的时间片个数*/,
            (FUNCPTR)dailyFunc,0/*更改成需要的参数*/);
}
```

运行时先调用 initDailyFunc()，此函数发起一次看门狗，导致在 1s 后 dailyFunc()被执行，而 dailyFunc 在入口处又重新调用了 wdStart()，导致 1s 后 dailyFunc()再次被执行，重复以往，保证每隔 1s dailyFunc 被调用一次，从而实现函数的定时执行。

需要注意的是，用这种方法去进行硬件的定时查询，必须保证一个前提：定时查询的内容中不能包含可能造成函数阻塞的成分，且必须保证每次硬件查询的时间都小于函数被执行的间隔时间，否则将造成函数被反复调用，随着时间的推移，将有越来越多的函数被同时运行，导致系统资源的耗竭。

5.2 硬件的中断响应

当嵌入式 CPU 的特定管脚被接到外围硬件上，外围硬件便可以通过将特定脚拉低来产生中断。一般来说，CPU 的每个中断管脚对应一个中断号，将这个中断号和中断服务程序绑定在一起，就可以在中断产生时自动调用中断服务程序。

采用这种方式可以保证每次硬件的数据准备好时，软件都可以及时地获得信息。

VxWorks 下中断处理的函数库为 intArchLib，需要包含的头文件为 iv.h 和 intLib.h。此库函数中的绝大多数函数和嵌入式 CPU 的种类有关，在此仅介绍其中的一部分，见表 5-2。

表 5-2 中断操作函数

intConnect()	将 C 函数绑定到硬件中断
intEnable()	允许对应的中断位（MIPS, PowerPC, ARM）
intDisable()	禁止对应的中断位（MIPS, PowerPC, ARM）

下面介绍在本书程序中涉及的中断操作函数，更详细的信息可以查阅函数库的在线帮助。

- 函数：STATUS intConnect(VOIDFUNCPTR *vector, VOIDFUNCPTR routine, int parameter)

描述：将一个特定的 C 程序绑定到中断。通常函数入口 routine 被存放在 vector 处，所以函数可以带参数。当有硬件中断发生时，函数自动被调用。系统为此建立适当的 C 环境、保存寄存器和建立堆栈。

该函数可以是普通的 C 语言函数，但不能包括可能引起任务阻塞的语句。通常，中断矢量可以从中断号获得，使用方法为 INUM_TO_IVEC(intNumber)。

参数：

vector　　中断矢量
routing　　中断服务程序的函数入口指针
parameter　　中断服务程序的参数

返回：OK，如果无法建立中断处理则返回 ERROR

- 函数：int intEnable(int level)

描述：允许对应的中断位

参数：

level　　中断位　(0x00~0xff00)

返回：OK，失败返回 ERROR

- 函数：int intDisable(int level)

描述：禁止对应的中断位。

参数：

level　　中断位

返回：OK，失败返回 ERROR

一般来说，中断服务程序不需要带参数，所有需要使用的参数都可以用全局变量动态改变。但当多个中断使用一个中断服务程序时，每次中断服务程序被调用时，需要知道是哪个中断导致程序被调用，此时可以使用中断号作为中断服务程序的参数。

按照 intConnect()的格式，中断服务程序只能带一个参数，如果中断服务程序有多个参数，那么硬件中断被调用时只有第一个参数被赋值，后面的参数都为 0。例如：

```
void intFunc(int param1, short param2, char param3)
{
    logMsg("intFunc: params = %d, %d, %d\n",param1,param2,param3,0,0,0);
}
```

如果 intConnect(INUM_TO_IVEC(intNum),(FUNCPTR)intFunc,3)；则打印信息为：

```
interrupt: intFunc: params =3, 0, 0
```

对于确实需要带多个参数的情况，可以将多个参数定义为结构体，而函数参数为结构体指针，如下所示：

```
/*参数结构体定义*/
struct intFuncParam
{
    int     paramInt;
    short   paramShort;
    char        paramChar;
};
/*中断服务程序*/
void intFunc(void* pParam)
{
    struct intFuncParam *pStructParam;
    pStructParam = (struct intFuncParam*)pParam;
    logMsg("intFunc: params = %d, %d, %d\n", pParam->paramInt,
           pParam->paramShort, pParam->paramChar,0,0,0);
}
/*中断服务程序参数的申明*/
struct intFuncParam funcParam;
/*参数赋值和中断绑定*/
funcParam.paramInt = 1;
funcParam.paramShort = 2;
funcParam.paramChar = 3;
intConnect( INUM_TO_IVEC(intNum),(FUNCPTR)intFunc,
           ((void *)(&funcParam.paramInt)) );
```

如此则中断发生时打印出：

```
interrupt: intFunc: params =1, 2, 3
```

注意到传递给中断服务程序的是指针而非实际值，因此必须注意指针的合法性，当中断发生时结构体必须还存在，否则结构体指针成为非法指针，不能使用。一般来说，将参数申明为全局变量可以保证指针合法，而将参数作为某个任务的局部变量则是十分危险的。

由于中断服务程序中不能调用可能导致阻塞的任务，因此中断服务程序的函数本身受

到很大的局限（watchdog 机制也可以看作是某种程度上的中断），对于必须紧接着中断而进行的网络或者 I/O 操作，可以通过将中断服务程序拆分来完成。将原来的中断服务程序拆分成中断服务程序和中断服务任务两部分：新的中断服务程序仅仅执行最基本的中断处理，例如禁止中断、判断中断类型等；绝大多数的任务处理，特别是会造成阻塞的任务，应该在中断服务任务中执行。中断服务程序和中断服务任务使用信号灯同步。模式如下：

```
void intFunc(int param)          中断服务程序*/
{
    intDisable(intNum);          /*禁止中断*/
    /*清中断*/
    semGive(semInt);             /*释放同步信号灯*/
}
void intHandle()                 /*中断服务任务*/
{
    while(semTake(semInt, WAIT_FOREVER)!=ERROR)    /*获取同步信号灯*/
    {
        /*获取成功，表示有中断发生，执行大部分的工作*/
        intEnable(intNum);       /*重新允许中断*/
    }
    intDisble(intNum);           /*出错，禁止中断*/
    intClose();                  /*结束，不再响应中断*/
    logMsg("intHandle: error in semTake, now exit\n",0,0,0,0,0,0);
}
void intInit()                   /*中断初始化*/
{
    semInt = semCCreate(SEM_Q_FIFO,0);
    intConnect(INUM_TO_IVEC(intNum), (FUNCPTR)intFunc, param);
    taskSpawn("tIntHandle",100,0,2000,(FUNCPTR)intHandle,0,0,0,0,0,0,0,0,
              0,0,0);
    intEnable(intNum);
}
void intClose()                  /*中断结束*/
{
    int taskId;
    intDisable(intNum);
    taskId = taskNameToId("tIntHandle");
    if(taskId!=ERROR && taskId!= taskIdSelf())
    {
        taskDelete(taskId);
    }
    semDelete(semInt);
}
```

除信号灯之外，中断服务程序和中断服务任务之间还可以采用的通信机制包括：共享内存、消息队列和管道等。

可以将多个中断绑定到同一个函数，这些中断中任意一个被拉低，都会导致此函数的调用。通常用于多个中断的处理任务大体一致而中断源不同的情况。为了使函数能够区分硬件上发生的到底是哪个中断，可以采用中断号作为函数参数的方法。如下：

```
void intFunc(int intNum)
{
    logMsg("intNum = %d\n", intNum,0,0,0,0,0);
    /*根据 intNum 判断中断源，并执行响应操作*/
}
void intInit()
{
    intConnect(INUM_TO_IVEC(1), (FUNCPTR)intFunc, 1);
    intConnect(INUM_TO_IVEC(2), (FUNCPTR)intFunc, 2);
}
```

这样当中断 1 被拉低时，打印出信息：

```
interrupt: intNum = 1
```

中断 2 被拉低时，打印出信息：

```
interrupt: intNum = 2
```

5.3 多采集板系统中断管理实例

在多采集板的大型系统中，通常采用工业机箱给多块板统一供电，使用零槽控制器对采集板插卡进行统一管理。本节将简要介绍一种采用 CPCI 机箱的多采集板系统中零槽控制器的中断管理方法。

5.3.1 CPCI 机箱 CPX2408

CompactPCI 是一种基于 PCI 总线、服务于工业和电信领域的计算机标准。它在电气特性上和编程接口上等同于台式机的 PCI 总线，但采用了不同的物理板形。CPCI 总线工控机之所以被业界所青睐，是因为其既具有 PCI 总线的高性能又具有欧洲卡结构的高可靠性，是符合国际标准的真正工业型计算机，适合在可靠性要求较高的工业和军事设备上应用。CPX2408 是 IEEE1101.10 兼容的 8 槽 CompactPCI 背板机箱。其中的零槽被固定为系统槽，管理整个 CPCI 机箱，包括对各插板进行配置、中断管理、热插拔控制等。下面要介绍的软件是基于的零槽控制器是 Motorola 公司的 Mcp750，它使用高性能的 PowerPC750 作为处理器，采用 PowerPlus 结构，优化的低功耗设计，功能强大，具有很高的可靠性。

5.3.2 VxWorks 下的 PCI 配置和操作

零槽的软件分为操作系统和用户程序两部分。其中操作系统即由 Wind River 提供的板

级支持包编译生成的 bootrom 和 VxWorks，上电后操作系统自动对 PCI 总线进行扫描和空间配置。而对各器件来说，除 PCI 首部寄存器之外的其他用户定制寄存器、申请的内存空间操作以及中断响应，都必须由用户程序负责。整个 CPCI 机箱中的 PCI 空间拓扑结构见图 5-1。

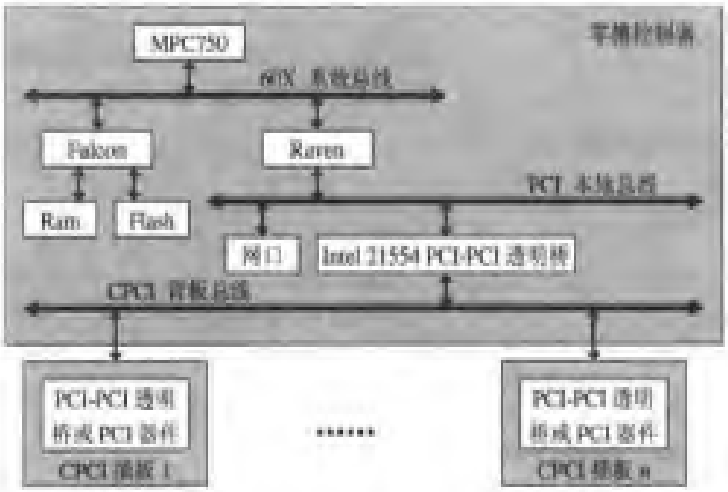


图 5-1 CPCI 机箱中的 PCI 空间拓扑结构

VxWorks 默认支持 PCI 2.0 版本的器件及透明桥地址空间配置，将非透明桥也看作器件进行配置。使用 pciAutoConfigLib.c 和 pciConfigShow.c 进行自动配置和配置结果显示。配置过程如下：

（1）sysHwInit()调用 sysRavenInit()实现系统总线到本地 PCI 桥 raven 的驱动。

（2）sysHwInit()中调用 sysPciAutoConfig()。sysPciAutoConfig()将 PCI 总空间的基地址和大小以及其他参数分别赋值给指定的变量，并挂接用户定义函数到四个变量——sysParams.includeRtn、sysParams.intAssignRtn、sysParams.bridgePreConfigInit、sysParams.bridgePostConfigInit，用来指定自动配置的器件、中断挂接、桥片预配置和桥片后配置。最后调用以这些变量为参数的库函数 pciAutoConfig()。

（3）pciAutoConfig()逐个检测 PCI 器件和桥片的基地址寄存器（Base Address Registers，简称 BAR），根据写入全 F 再读出的方法确定 PCI 器件申请空间的大小，在相应的地址总空间内申请空间，并将申请到空间的首地址写入 BAR。透明桥的 BAR 在其下级总线上的器件全部配置完毕后再配置。

一般来说，如果使用的 PCI 器件符合 PCI 规范，程序就会自动配置包括 mem/IO 空间和中断在内的首部寄存器。用户需要修改 BSP 目录下的 sysBusPci.c 和 sysBusPci.h 两个文件，改变空间范围和器件对应的中断号。需要注意的是，默认的 sysPciAutoConfig()不支持 64 位地址的 PCI 器件。

用户程序需要对指定器件的指定 PCI 首部寄存器进行读写操作，需要用到的主要函数见表 5-3。

在 Tornado 的在线帮助中没有关于这些函数的说明，用户可以参阅“Tornado 安装目录

\target\src\drv\pci”目录下的 pciAutoConfigLib.c 和 pciConfigShow.c 等文件来理解这些函数的使用，对应的.h 文件存放在“Tornado 安装目录\target\h\drv\pci”目录下。

表 5-3 PCI 器件操作函数

pciFindDevice()	查询特定 device ID 和 vendor ID 的 PCI 器件
pciConfigInByte() pciConfigInWord() pciConfigInLong()	从特定的 PCI 器件的特定首部寄存器获得 1/2/4B 长度的数据
pciConfigOutByte() pciConfigOutWord() pciConfigOutLong()	向特定的 PCI 器件的特定首部寄存器填写 1/2/4B 长度的数据
pciDeviceShow()	显示特定总线上的所有 PCI 器件的信息
pciHeaderShow()	显示特定 PCI 器件的部分首部寄存器信息

- 函数：STATUS pciFindDevice (int vendorId, int deviceId, int index, int * pBusNo, int * pDeviceNo, int * pFuncNo);

描述：查询 vendorId、deviceId 和 index 分别符合要求的 PCI 器件，如果找到，将其所对应的 bus 号、device 号和 func 号分别填入给定的内存地址。

参数：

vendorId　PCI 器件的 vendorId，长度 2B，对应的 PCI 首部寄存器的偏移地址为 0x00

deviceId　PCI 器件的 deviceId，长度 2B，对应的 PCI 首局寄存器的偏移地址为 0x02

index　PCI 器件的标号。如果在 PCI 总线上有一个以上的 vendorId 和 deviceId 都相同的器件，则这些器件的标号按照总线号和器件位置号排列：总线号越小，器件标号越小；当器件在同一条总线上时，器件位置号越小，器件标号越小。最小标号为 0。当只有一个器件时，标号即为 0

pBusNo　用来填写器件所在总线号的地址

pDeviceNo　用来填写器件位置号的地址

pFuncNo　用来填写器件使用的 func 号的地址

返回：OK，如果找不到符合条件的 PCI 器件则返回 ERROR

- 函数：STATUS pciConfigInByte(int busNo, int deviceNo, int funcNo, int address, UINT8 * pData);

STATUS pciConfigInWord (int busNo, int deviceNo, int funcNo, int address, UINT16 * pData);

STATUS pciConfigInLong (int busNo, int deviceNo, int funcNo, int address, UINT32 * pData);

描述：从由总线号、器件位置号和 func 号指定的 PCI 器件的偏移量为 address 的首部

寄存器中读取长度为 1/2/4B 的值。

参数：

busNo	指定器件的总线号
deviceNo	指定器件的位置号
funcNo	指定器件的 func 号
address	首部寄存器的偏移量
pData	读取到的数据的存放地址

返回：OK，如果器件不存在则返回 ERROR

- 函数：STATUS pciConfigOutByte (int busNo, int deviceNo, int funcNo, int address, UINT8 data);

  STATUS pciConfigOutWord (int busNo, int deviceNo, int funcNo, int address, UINT16 data);

  STATUS pciConfigOutLong (int busNo, int deviceNo, int funcNo, int address, UINT32 data);

描述：向由总线号、器件位置号和 func 号指定的 PCI 器件的偏移量为 address 的首部寄存器中填写长度为 1/2/4B 的值。

参数：

busNo	指定器件的总线号
deviceNo	指定器件的位置号
funcNo	指定器件的 func 号
address	首部寄存器的偏移量
pData	读取到的数据的存放地址

返回：OK，如果器件不存在则返回 ERROR

- 函数：STATUS pciDeviceShow (int busNo);

描述：显示某条总线上所有的器件的信息。

参数：

busNo	总线号

返回：OK，如果没有这条总线则返回 ERROR

- 函数：STATUS pciHeaderShow (int busNo, int deviceNo, int funcNo);

描述：打印由总线号、器件位置号和 func 号指定的 PCI 器件的部分首地址寄存器值。

参数：

busNo	指定器件的总线号
deviceNo	指定器件的位置号
funcNo	指定器件的 func 号

返回：OK，失败返回 ERROR

为了方便用户查阅，在表 5-4 中给出了标准的 PCI 首部寄存器布局。

表 5-4 PCI 首部寄存器布局(Version2.2)

bit31 16	15 0	偏移地址
Device ID	Vendor ID	0x00
Status	Command	0x04
Class Code	Revision ID	0x08
BIST / Header Type / LatencyTimer / Cache Line Size		0x0C
Base Address Register 0		0x10
Base Address Register 1		0x14
Base Address Register 2		0x18
Base Address Register 3		0x1C
Base Address Register 4		0x20
Base Address Register 5		0x24
Cardbus CIS Pointer		0x28
Subsystem ID	Subsystem Vendor ID	0x2C
Expansion ROM Base Address		0x30
Reserved	Capabilities Pointer	0x34
Reserved		0x38
Max_Lat / Min_Gnt / Interrupt Pin / Interrupt Line		0x3C

5.3.3 CPCI 背板中断的多任务管理

由于只有零槽能够获得所有插板的 PCI 信息，因此如果各插板之间需要通过背板传递信息，就必须由零槽通知它们需要传输数据的目的地址。传输完毕之后可以通过背板中断通知零槽，并由零槽进行必要的操作，以实现下一次的数据传输。这些功能的实现借助于：存储各板的 memory 首地址及中断等信息的全局结构体变量、分别绑定到 4 个背板中断的中断服务程序，以及对应到 7 块插板的 7 个中断服务任务。其中中断服务程序和中断服务任务的入口函数分别都只有一个，而不同的参数对应不同中断和不同插板。

1. 初始化

初始化需要进行的工作有全局变量初始化、中断绑定和多个任务的发起。

用来定义全局变量的结构体如下：

```
struct pciBoardIndex
{
    int index;          /*板号*/
    char    intNum;     /*中断号*/
    int intTimes;       /*已经产生的中断次数*/
    UINT    pcimem0;    /*CPCI 器件申请的 memory0 的基地址*/
    UINT    pcimem1;    /*CPCI 器件申请的 memory1 的基地址*/
    SEM_ID  semPci;     /*同步用信号灯*/
};
```

```
struct pciIntIndex
{
    char enable;    /*该参数的每  bit 对应是否有该 index 的插板被分配到该中断*/
}
```

使用此结构体申请全局变量，相当于建立了中断号、板号和槽号（与 PCI 操作函数的参数 deviceNo 有对应关系）之间的对照表，供中断服务程序和中断服务任务使用。

```
struct boardStruct pciBoard[7];    /*7 块插板分别对应的中断信息*/
```

中断服务程序对应的函数为：void cPciIntFunc(int intNum);

中断服务任务对应的函数为：void cPciIntHandle(int slotNum);

用户程序首先通过读取各板的 PCI 首部寄存器来获得各槽上的插板信息，给 pciBoard[slotNum]的 index、各 pcimem[i]和 intNum 赋初值。如果该槽上没有插板，则返回的寄存器值为全 F，对应这样的槽，可以将这些值都赋为 0。初始化所有的 pciBoard[slotNum].intTimes 为 0，对于已经有插板的槽，发起对应的中断服务任务：

```
sprintf(buff, "intHdl%d",slotNum);
    taskSpawn (buff, TASK_PRIORITY, 0, TASK_STACKSIZE, (FUNCPTR)
              cPciIntHandle, slotNum, 0, 0, 0, 0, 0, 0, 0, 0, 0);
```

创建此中断服务任务使用的信号灯 pciBoard[slotNum].semBoard。

将四个中断绑定到中断服务程序（这里的 intNum 从 18 开始，是因为在 CPX2408 机箱中，背板的四个 PCI 中断 INTA～INTD 分别对应到零槽 CPU 的中断 18～21）：

```
for(intNum=18;i<22;i++)
{
    intConnect((INUM_TO_IVEC(intNum)),f9656IntFunc, intNum);
```

2. 中断服务程序和中断服务任务

多板的中断服务程序和中断服务任务的通用处理流程见图 5-2。

中断服务程序 cPciIntFunc(intNum)被调用，说明 CPU 接收到中断号为 intNum 的中断。此程序首先关中断 intNum，然后进行接到中断后最基本的工作，例如清中断标志等。而中断服务程序的首要工作是：通过查询所有挂到此中断的插板的指定寄存器（判断条件为 pciBoard[slotNum].intNum = = intNum && 指定寄存器= =指定值），确定此中断是由哪块插板（slotNum）产生的。然后释放对应的信号灯（semGive(pciBoard[slotNum]. semBoard)），使能中断 intNum。

中断服务任务 cPciIntHandle(slotNum)循环获取信号灯（semTake(pciBoard[slotNum]. semBoard，WAIT_FOREVER)），在没有获得被中断服务程序释放的信号灯之前，任务处于阻塞状态，不占用 CPU。获取成功说明该插板产生了中断，此任务将 pciBoard[slotNum]. intTimes 加一，接下来再进行相应的处理，例如数据的处理和贮存，然后才允许此插板再次传输数据。这样此板才有再次产生中断的可能。而由于此中断已经在中断服务程序中使能了，因此只要数据传输完毕，此中断对应的中断服务程序就会被系统调用。而在此之前，中断服务程序仍然能够响应与此板挂在同一中断上的另一块插板所产生的中断。由于在

CPCI 机箱中，中断与插板是一对多的关系，因此只有采用中断服务程序负责同一中断，而中断服务任务负责单一插板的工作模式，才能保证所有的中断都能够得到及时响应。如果中断比较频繁，且系统工作时间较长，则在中断服务任务的一次循环结束以前还需要进行中断次数的过界判断，以保证不会出现 pciBoard[slotNum]增加到 0xFFFFFFFF 后再加一而发生意外变化的情况。

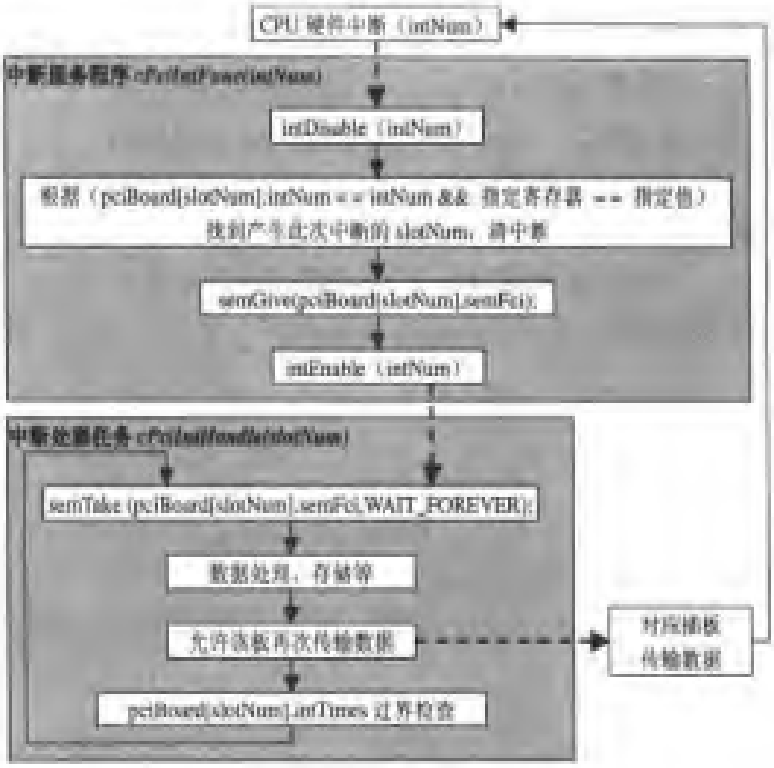


图 5-2　多中断的处理流程

这种中断处理的方法能够在很大程度上简化软件，中断服务程序和中断服务任务都只需要写一个函数，就可以分别响应最多 4 种中断，并分别处理插在全部 7 个插槽或部分插槽上多块采集卡的数据。它充分利用了 VxWorks 的强实时性和对多任务的良好支持。程序结构广泛适用于各种背板总线上的各插板间数据传输管理，且便于移植到其他嵌入式操作系统。

例 5-1 是多采集板中断控制软件的框架，用户需要根据实际板卡的不同做出相应修改。

例 5-1　多采集板中断控制软件框架（pciMultiInt.h 和 pciMultiInt.c）

```
/*****************************************************************************
 chengjy@felab, copyright 2002-2004
 cpciMultiInt.h
 零槽控制器的多板中断处理需要的常数定义
 *****************************************************************************/
#ifndef _CPCIMULTIINT_H
#define _CPCIMULTIINT_H
```

```
/*~~~~~~~~~~~~~~~~~~~~地址转换~~~~~~~~~~~~~~~~~~~~~~~~*/
/*基地址定义，从bsp继承：mcpx750.h*/
#define PCI_MEM_BASE_MODE1      0x81000000
#define PCI_MEM_BASE_MODE2      0xc0000000
#define PCI_MSTR_MEMIO_LOCAL    PCI_MEM_BASE_MODE2 /*pci mem
                                                   -- cpu view base address*/
#define PCI_MSTR_IO_LOCAL       0x80000000  /*pci io
                                                   -- cpu view base address*/
#define PCI_SLV_MEM_BUS         0x80000000  /*mem
                                                   -- pci view base address*/

/*PCI总线上的地址到CPU地址的转换*/
#define PCIMEM2CPU(x)       ((UINT*)((UINT)(x)+PCI_MSTR_MEMIO_LOCAL))
#define PCIIO2CPU(x)        ((UINT*)((UINT)(x)+PCI_MSTR_IO_LOCAL))

/*内存地址从CPU LOCAL BUS到PCI地址的转换*/
#define CPU2PCIMEM(x)       ((UINT*)((UINT)(x)-PCI_MSTR_MEMIO_LOCAL))
#define CPU2PCIIO(x)        ((UINT*)((UINT)(x)-PCI_MSTR_IO_LOCAL))

/*CPU mem 地址到PCI地址的转换*/
#define CPUMEM2PCI(x)       ((UINT*)((UINT)(x)+PCI_SLV_MEM_BUS))

/**********************FCI 9656相关*****************************/
/*FCI板9656的vendor 和device ID号*/
#define PCI_F9656_VENDOR_ID     0x10B5
#define PCI_F9656_DEVICE_ID     0x9656

/*fci 板9656 的PCI mem 操作*/
#define F9656MEM2CPU0(x,index)  ((UINT*)(pciBoard[index].pcimem0+(x)))
 /*mem0区*/
#define F9656MEM2CPU1(x,index)  ((UINT*)(pciBoard[index].pcimem1+(x)))
 /*mem1区*/

/*9656 字序转换*/
#define F9656_BYTE_SWAP(x)  \
     (((((UINT)(x))<<24)&0xFF000000)+((((UINT)(x))<<8)&0x00FF0000) \
     +((((UINT)(x))>>8)&0x0000FF00)+((((UINT)(x))>>24)&0x000000FF))

/*定义单板状态*/
#define STATUS_NORMAL           0x00
#define STATUS_ERROR            0x01
#define STATUS_WARNING          0x02
#define STATUS_INVALID          0x03
```

```
#define STATUS_UNAVAILABLE  0x04
#define STATUS_DISABLE      0x05

/*用户任务的堆栈大小*/
#define USER_STACK_SIZE     2000

/*定义任务名*/
#define TNAME_F9656INTHDL   "tF9656intHdl"
#define TPRI_F96561NTHDL    120

/*~~~~~~~~~~~~~~~~~~~~~~CPCI 插卡的基本信息~~~~~~~~~~~~~~~~~~~~~~*/
struct pciBoardIndex
{
int  index;                 /*板号*/
char intNum;                /*中断号*/
int  intTimes;              /*已经产生的中断次数*/
UINT    pcimem0;            /*CPCI 器件申请的 memory0 的基地址*/
UINT    pcimem1;            /*CPCI 器件申请的 memory1 的基地址*/
 SEM_ID  semPci;            /*同步用信号灯*/
};

struct pciIntIndex
{
 char enable;
};

#define MAX_BOARD_SUPPORT   7       /*最多支持的插板数*/
#define MAX_INT_SUPPORT     4       /*CPCI 插板最多支持 4 个中断*/

/* CompactPCI Bus INTA# level */
#define CPCIINT2CPU(x)   ((char)(x)+0x10) /*从获取的中断号转换到 CPU 对应的中断号*/
#define CPCI_INTA_LEVEL_CPU CPCIINT2CPU(8)      /*CPU 看到的 CPCI 的中断 A */
#define INT_INDEX2NUM(x)        ((x)+CPCI_INTA_LEVEL_CPU)
#define INT_NUM2INDEX(x)        ((x)-CPCI_INTA_LEVEL_CPU)

#endif  /*_CPCIMULTIINT_H*/

/*************************************************************************
 chengjy@felab, copyright 2002-2004
 cpciMultiInt.c
 函数:
     void f9656Init();
     void f9656IntFunc(int intIndex);
     void f9656IntHandle(int boardNum);
```

```
    void f9656ShutDown();
 调用:        无
 被调用:      硬件响应底层函数，被命令通道调用
 说明:        本文件与硬件关联紧密，仅为中断响应的框架，需要根据实际情况添加中断响
        应和处理代码才能使用
 ******************************************************************************/
#include "vxWorks.h"
#include "logLib.h"
#include "arch/ppc/ivPpc.h"
#include "intLib.h"
#include "taskLib.h"
#include "semLib.h"
#include "string.h"

#include "cpciMultiInt.h"

#if 0       /*不打印 9656 中断信息*/
#define F9656LOG(a,b,c,d,e,f,g);          /*带 6 个参数的 logMsg*/
#define F9656PRINT1(x)      ;             /*没有参数的 printf*/
#define F9656PRINT2(x,y)        ;         /*带一个参数的 printf*/
#else
#define F9656LOG(a,b,c,d,e,f,g) ;  /*logMsg((a),(b),(c),(d),(e),(f),(g))*/
#define F9656PRINT1(x)             printf(x)
#define F9656PRINT2(x,y)           printf((x),(y))
#endif

struct pciBoardIndex    pciBoard[MAX_BOARD_SUPPORT];
struct pciIntIndex      pciInt[MAX_INT_SUPPORT];
int pciBoardNum;        /*实际使用的 pci 板的数目，从 0 开始*/

void f9656Init();
void f9656IntFunc(int intIndex);
void f9656IntHandle(int boardNum);
void f9656ShutDown();

/*****************************************************************************
 void f9656Init()
 函数描述:    9656 初始化，创建信号灯，发起服务任务，计算基地址，挂接中断。
 参数:        无
 返回:        无
 调用:        void f9656IntFunc(int intIndex);
              void f9656IntHandle(int boardNum);
 被调用:      用户程序初始化部分
              命令通道的命令
```

```
*****************************************************************************/
void f9656Init()
{
    int busNo,deviceNo,funcNo;
    int intIndex;
    int intNum;
    int i,j;
    STATUS state;

    char taskName[30];

    /*需要避免重入的控制*/

    /*初始化每块插板信息*/
    for(i=0;i<MAX_BOARD_SUPPORT;i++)
    {
        /*首先全部设置为0*/
        pciBoard[i].pcimem0     = 0;
        pciBoard[i].pcimem1     = 0;
        pciBoard[i].intNum      = 0xFF;
        pciBoard[i].intTimes    = 0;
    }

    /*初始化*/
    for(pciBoardNum=0;pciBoardNum<MAX_BOARD_SUPPORT;pciBoardNum++)
    {
        /*寻找9656*/
        if(pciFindDevice(PCI_F9656_VENDOR_ID,PCI_F9656_DEVICE_ID,
           pciBoardNum,&busNo,&deviceNo,&funcNo)!=OK)
     {
         break;
     }
     logMsg("f9656Init: pci device, vendor 0x%x,device 0x%x, index 0x%x\n",
            PCI_F9656_VENDOR_ID,PCI_F9656_DEVICE_ID,pciBoardNum,0,0,0);

     /*mem基地址*/
     pciConfigInLong(busNo,deviceNo,funcNo,0x10,
                &(pciBoard[pciBoardNum].pcimem0));
     pciBoard[pciBoardNum].pcimem0 =
                (UINT)PCIMEM2CPU(pciBoard[pciBoardNum].pcimem0&(~0xf));
     pciConfigInLong(busNo,deviceNo,funcNo,0x18,
                &(pciBoard[pciBoardNum].pcimem1));
     pciBoard[pciBoardNum].pcimem1 =
                (UINT)PCIMEM2CPU(pciBoard[pciBoardNum].pcimem1&(~0xf));
```

```
    /*中断号*/
    pciConfigInByte(busNo,deviceNo,funcNo,0x3C,&(pciBoard[pciBoardNum].
    intNum));
    pciBoard[pciBoardNum].intNum = CPCIINT2CPU(pciBoard[pciBoardNum].
    intNum);

    /*板号，可以通过9656的首部寄存器或者memory 1某个地址的值不同来区分，
    这里为了简便直接赋值为pciBoardNum+1*/
    pciBoard[pciBoardNum].index = pciBoardNum+1;

    logMsg("f9656Init: memBase: 0x%8x,0x%8x,intNum 0x%x\n",
           pciBoard[pciBoardNum].pcimem0,pciBoard[pciBoardNum].pcimem1,
           pciBoard[pciBoardNum].intNum,0,0,0);
}

/*如果没有找到任何CPCI器件，退出*/
if(pciBoardNum==0)
{
    logMsg("f9656Init: no cpci board found, now exit\n",0,0,0,0,0,0);
    return;
}

/*没有找到CPCI器件的中断index，对应的enable全部为0*/
for(i=0;i<MAX_INT_SUPPORT;i++)
{
    pciInt[i].enable = 0;
}
for(i=0;i<pciBoardNum;i++)
{
    /*某个中断相应位上的1表示对应有插板使用此中断*/
    intIndex = INT_NUM2INDEX(pciBoard[pciBoardNum].intNum);
    pciInt[intIndex].enable = (pciInt[intIndex].enable | (0x01<<i));
}

/*逐个使能中断*/
for(intIndex=0;intIndex<MAX_INT_SUPPORT;intIndex++)
{
    if(pciInt[intIndex].enable!=0)
    {
        intNum = INT_INDEX2NUM(intIndex);
        F9656LOG("f9656Init: int number: 0x%x connected\n",intNum,0,0,0,0,0);
        state = intConnect((INUM_TO_IVEC(intNum)),f9656IntFunc,intIndex);
        if(state!=OK)
```

```
            logMsg("f9656Init: error in intConnect\n",0,0,0,0,0,0);
        intEnable(intNum);
    }
}

/*逐个初始化插板*/
for(i=0;i<pciBoardNum;i++)
{
    /*创建信号灯*/
    pciBoard[i].semPci = semCCreate(SEM_Q_FIFO,SEM_EMPTY);

    if(pciBoard[i].semPci == NULL)
    {
        logMsg("f9656Init: unable to create semF9656\n",0,0,0,0,0,0);
        exit(-1);
    }

    /*发起相应的处理函数*/
    sprintf(taskName,TNAME_F9656INTHDL,"%d",i);
    taskSpawn(taskName,TPRI_F9656INTHDL,0,USER_STACK_SIZE,
              (FUNCPTR)f9656IntHandle,i,0,0,0,0,0,0,0,0,0);
}
}

/*************************************************************************
 void f9656IntFunc(int intIndex)
 函数描述:  中断服务程序,挂接到9656中断,中断时自动被调用。具体工作由f9656IntHandle()
            完成,使用各板的semPci 同步。此函数中的清中断机制直接针对硬件,如果使用9656
            之外的其他PCI桥片,请参阅芯片的datasheet.
 参数:      index,产生的中断的index,取值范围从0~3.
 返回:      无
 调用:      无
 被调用:    void f9656Init()
 *************************************************************************/
void f9656IntFunc(int intIndex)
{
    int  intNum;
    UINT regValue;
    int  pciIndex;

    intNum = INT_INDEX2NUM(intIndex);
    /*屏蔽中断*/
    intDisable(intNum);
```

```
    F9656LOG("f9656IntFunc: interrupt received\n",0,0,0,0,0,0);

    /*获取中断的板号*/
    for(pciIndex=0;pciIndex<MAX_BOARD_SUPPORT;pciIndex++)
{
        if((pciInt[intIndex].enable&(0x01<<pciIndex))!=0)
        {
            /*逐个判断可能产生此中断的CPCI 插板是否已经产生了中断*/
            regValue = *F9656MEM2CPU0(0x68,pciIndex);
            if((regValue&0x00002000)!=0)
            {
                /*清单板的中断标志*/
                *F9656MEM2CPU0(0xA8,pciIndex) = 0x18000000;
                /*释放信号灯，通知 f9656IntHandle 开始处理*/
                semGive(pciBoard[pciIndex].semPci);
            }
        }
    }

    /*允许响应其他使用此中断号的插板的中断*/
    intEnable(intNum);
}

/*************************************************************************
 void f9656IntHandle(int boardNum)
 函数描述:    中断后的实际任务处理函数，中断服务程序并不执行大量工作，只是通知此函数
 参数:        boardNum,产生中断的板的CPCI 号，取值范围 0~7
 返回:        无
 调用:        void f9656ShutDown()
 被调用:      void f9656Init()
 *************************************************************************/
void f9656IntHandle(int boardNum)
{
    while(1)
    {
        if(semTake(pciBoard[boardNum].semPci,WAIT_FOREVER)!=OK)
        {
            logMsg("f9656IntHandle: error in semTake %d\n",boardNum,
                    0,0,0,0,0);
            break;
        }

        /*成功获取信号灯，说明该板产生了一次中断*/
        pciBoard[boardNum].intTimes++;
```

```
        /*防止过界*/
        if(pciBoard[boardNum].intTimes==0x1000000)
            pciBoard[boardNum].intTimes = 0;

        /*允许 9656 再次传送数据到零槽*/
    }
    /*出错退出*/
    logMsg("f9656IntHandle: something wrong, now exit\n",0,0,0,0,0,0);
    f9656ShutDown();
    }

/*************************************************************************
 void f9656ShutDown()
 函数描述:    关闭 9656 的操作，主要是解决一些在 shell 下重复使用 F9656Init()的信号
        灯没有删除的问题，在这里删除信号灯以及中断后的处理函数
 参数:        无
 返回:        无
 调用:        无
 被调用:      void f9656IntHandle(int boardNum)
              用户在 shell 下调用
              命令通道的某个命令
 *************************************************************************/
void f9656ShutDown()
{
    int i;
    int intIndex;
    char taskName[30];

    /*需要添加避免重入的判断*/

    /*按中断 index 0～3 关闭中断*/
    for(intIndex=0;intIndex<4;intIndex++)
    {
        intDisable(INT_INDEX2NUM(intIndex));
    }
    /*按插板 index 0～7 结束任务*/
    for(i=0;i<pciBoardNum;i++)
    {
        /*删除中断服务任务*/
        sprintf(taskName,TNAME_F9656INTHDL,"%d",i);
        /*如果是被 f9656IntHandle()调用，自己会退出，否则必须删除任务*/
        if(taskNameToId(taskName)!=taskIdSelf())
        {
```

```
                taskDelete(taskNameToId(taskName));
                F9656LOG("f9656ShutDown:%d deleted\n",i,0,0,0,0,0);
            }

            /*删除信号灯*/
            semDelete(pciBoard[i].semPci);
        }
        logMsg("f9656ShutDown:9656 shut down now\n",0,0,0,0,0,0);
    }
```

5.4 总结

本章主要介绍的是与硬件操作直接相关的一些函数。由于硬件操作归根到底就是通过CPU对固定的地址的读写产生外部硬件需要的时序，而具体操作必须由硬件决定，因此介绍的主要是方法而非详细内容。首先介绍软件如何产生定时轮询，分为taskDelay和watchdog两种方式，并比较了两者的优缺点；然后介绍了软件的中断响应机制，及如何划分必须要调用阻塞函数的中断服务程序；最后给出了一个基于CPCI背板的多中断联合管理程序的框架。

第6章　一体化设计——多任务控制

在向上与控制软件交流的网络通信和向下与硬件交流的中断轮询的基本程序完成之后，再回过头来整理一下程序，将分开的模块一体化，并协调各部分之间的关系，以保证整个系统的联合运行。

6.1　任务优先级划分

在前面关于网络通信的部分已经介绍了如何通过设置各任务优先级的高低来保证命令接收的优先执行。当所有的底层函数已经完成时，用户程序的编写者应当对每一时刻正在运行的各任务有一个全局的了解。

可以按照下面的步骤列出任务，帮助理解和划分任务优先级。

（1）找出全部可能被发起的任务，并按照各自所在的.c文件分组。

（2）给所有的任务写出对应的任务名，格式统一采用#define TNAME_XXX “tXXX”。

（3）将这些任务分类。

（4）给各个任务规定其优先级，格式统一采用#define TPRI_XXX　　[优先级]。

一般可以将任务分为3类：初始化使用的任务xxxInit()、结束时使用的任务xxxClose()以及那些贯穿整个正常运行过程的其他函数。通常，初始化任务会发起正常运行的函数，并在发起后结束。即正常情况下，它们只在系统初始化的时候被运行，也可以由控制端命令其执行。结束任务删除所有相关的任务，通常在某个部分出错时被运行，或者是控制端命令其执行。而正常运行的函数则普遍采用 while(1)形式，以保证其不会退出，但通常会调用使其自身阻塞的函数，只在特定条件下解除阻塞，进行一次循环，然后再次阻塞。

一般来说，将初始化任务和结束任务的优先级划分得较高，以保证其优先运行。中断服务任务的优先级也可以设置得较高，否则中断服务程序的优先响应相当于没有效果。

任务发起一律使用taskSpawn，采用格式：

```
taskSpawn(TNAME_XXX,TPRI_XXX,0,USER_STACK_SIZE,(FUNCPTR)funcXXX,
      param1,param2,param3,param4, param5, param6, param7, param8, param9,
      param10);
```

6.2　全局变量

对待全局变量必须十分谨慎，应当尽量少地使用全局变量，并且注意其不会被两个同时运行的任务调用，除非有同步用的信号灯。

用户程序中的全局变量可分为二类：

（1）库函数需要使用的标志 ID。

包括看门狗 ID、信号灯 ID、网络通信的套接字等。这些变量一般只由库函数进行处理。有初始化的库函数，就一定要有对应的关闭的库函数。在操作系统运行过程中，如果没有关闭这些 ID 变量就直接下载新编译的程序，不仅会造成 ID 变量的归零，而且会导致资源无法释放。例如，如果套接字没有关闭，则再次 bind()时就会因为端口已经被占用而无法成功，而从前的 ID 又已经丢失，只有通过重新启动复位硬件来解决。

（2）需要灵活配置的参数。

主要是和硬件紧密相关的参数，例如采集板采集数据的时间长度等。这些参数通常采用宏定义的方式给定初值，在初始化的时候赋值，运行过程中可以随时改变。一般来说，当系统成型之后，通过网络的命令通道传输这些参数到采集板，以实现对采集板参数的灵活控制。为了管理方便，通常将这些参数定义在一起，构成一个结构体，并使用一个函数统一赋给初值。改变这些参数的值只在系统初始化、命令通道控制或者用户调试时，其他函数将这些值作为参考，而不改变它们。

（3）根据运行情况改变的变量。

这些变量用来标志系统的运行情况，只由相关的用户程序改变，并作为其他程序运行的参考。在命令通道和 shell 下都只是通过读取来探查系统运行情况，而不应该改变它们的值。例如中断计数变量，每次中断后加一，只由中断服务任务改变。用户一般是在 shell 下输入此变量名，查看中断发生的情况。

为了避免程序的重复调用和硬件的重复初始化，通常使用一个全局变量来标志软硬件的初始情况。用这个全局变量的各 bit 标志各个硬件是否已经初始化。例如：

```
int boardInit = BOARD_UNINIT;
```

声明 boardInit 用来标志采集板，这里也可以直接声明成：

```
int boardInit;
```

由于 VxWorks 默认将所有没有定义初值的全局变量填为 0，因此 boardInit 的初值为零。定义各标志位如下：

```
#define BOARD_UNINIT            0x00
#define BOARD_F9656_INITED      0x01
#define BOARD_NET_INITED        0x02
#define BOARD_GENET_INITED      0x04
#define BOARD_FLASH_INITED      0x08
#define BOARD_SOFTPRAM_INITED   0x80
#define BOARD_INITED            0x9F
```

在硬件初始化时，采用下面的语句来避免重复初始化：

```
void XXXInit()
{
        if(!(boardInit&BOARD_XXX _INITED))
    {
```

```
            /*初始化具体代码*/
            logMsg("XXXInit: initialization finished\n",0,0,0,0,0,0);
            boardInit = boardInit | BOARD_ XXX _INITED;
        }
        else
    {
        logMsg("XXXInit: nothing is done because XXX has already been
initialized\n",0,0,0,0,0,0);-
    }
}
```

在硬件关闭时，采用下面的语句来避免被重复关闭：

```
void XXXClose()
{
    if(boardInit&BOARD_ XXX _INITED)
    {
        /*关闭硬件具体代码*/
        boardInit = boardInit & (~(BOARD_XXX_INITED));
        logMsg("XXXClose: closure finished\n",0,0,0,0,0,0);
    }
    else
    {
        logMsg("XXXClose: nothing is done because XXX has already been
closed\n",0,0,0,0,0,0);
    }
}
```

在某些需要硬件初始化之后才能执行的函数中添加如下控制：

```
char XXXDoSth()
{
    if(!(boardInit&BOARD_S5933_INITED))
    {
        logMsg("XXXDoSth: XXX unInited, quit\n",0,0,0,0,0,0);
        return(STATUS_WARNING);
    }
    /*真正需要执行的硬件操作*/
}
```

6.3 用户程序入口和灵活配置参数的初始化

由于网络通信已经建立，因此普通的流程操作都可以使用网络控制。为了方便操作，并且使初始化部分更加明确和可控，将所有的用户程序统一到一个入口。将这个进行最初初始化的函数称为用户程序入口函数，格式为 void usr[*boardName*]Root()。

此函数只需要进行最基本的初始化，包括灵活配置参数的初始化和网络通信模块的初始化，一般不涉及其他硬件的初始化，它们的初始化和关闭使用网络命令控制。规划后的用户程序结构见图 6-1。

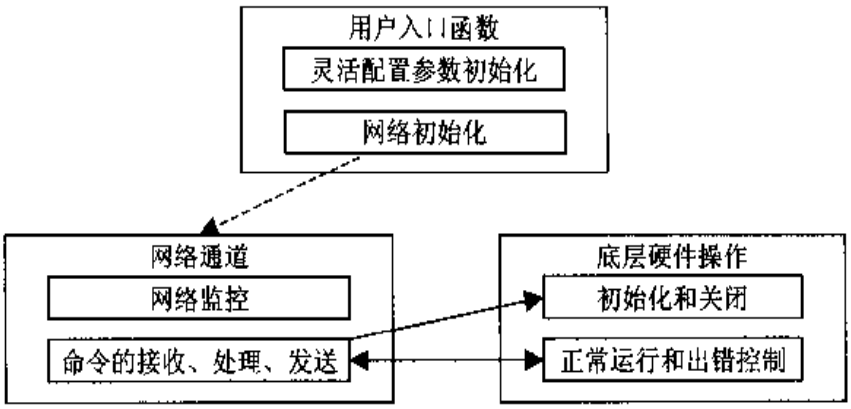


图 6-1　用户程序的结构

对于网络初始化，可以用下面的语句执行。由于网络拥有自己的初始化重入控制，不必担心其是否被重复调用。

```
taskSpawn(TNAME_NETINIT,TPRI_NETINIT,0,USER_STACK_SIZE,
          (FUNCPTR)netInit,0,0,0,0,0,0,0,0,0,0);
```

对于灵活配置参数，通常将其作为一个结构体统一管理，下面给出一个例子，其中需要灵活配置的只有一个参数 dmaAutoflag，用户可以根据自己的采集板实际情况改变下面的结构体。

```
/*~~~~~~~~~~~~~~~~~~~~~~板子工作环境~~~~~~~~~~~~~~~~~~~~~~*/
struct BoardWorkEnv
{
    int     boardInit;          /*boardInit 单个位用来表示初始化状态*/
    char    autoBootFlag;       /*标志采集板是完成阶段还是调试阶段*/
    char    dmaAutoflag;        /*定义采集板的 dma 是由硬件控制还是零槽软件控制*/
};
```

除了 boardInit 使用来表示硬件是否初始化的变量之外，其他的变量应该有自己的默认值，采用下面的方式来定义：

```
#define TB_SOURCE_OCM       0   /*TB 从工作站通过命令单独发出*/
#define TB_SOURCE_SLOT0     1   /*TB 从零槽连续发出*/
#define DMA_AUTO_DISABLE    0
#define DMA_AUTO_ENABLE     1
#define DFT_DMA_AUTO        DMA_AUTO_ENABLE
#define USR_DMA_AUTO        DMA_AUTO_DISABLE
```

使用一个函数 usrDefaultParamLoad ()统一给灵活配置参数赋初值，分为两种模式：第

一种是软件最后成型的模式，它采用 DFT_XXX 的定义，是操作系统和用户程序结合后的默认参数；第二种是用户调试模式，采用 USR_XXX 的定义，主要是因为用户在 shell 下初始化硬件，必须使用灵活配置参数，而调用此函数给它们赋初值。调用方由 autoBootFlag 来判断，由于默认的 autoBootFlag 为 0，因此程序编译下载后如果直接调用硬件初始化函数，则 autoBootFlag = = 0，标志为用户调用；而如果是用户程序入口函数调用，则在调用前修改 autoBootFlag=1，表示使用 DFT_XXX 参数。

此外，还需要一个能够统一显示灵活配置参数的函数 usrParamShow()，用来列出这些参数的配置情况。该函数通常在改变这些参数后调用，以保证参数配置的正确性。

对于网络控制的情况，通常在用户程序启动后硬件初始化之前，控制端还会使用某条命令对这些灵活配置参数的全部或者部分统一重新配置。这种由采集板软件提供初值，再由控制端改变采集板工作参数的方法，大大增加了软件的灵活性。

为了标志各板的不同，采用另一个结构体显示某些不可由用户程序来改变的参数。

```
/*~~~~~~~~~~~~~~~~~~~~~~板子标志结构~~~~~~~~~~~~~~~~~~~~~~*/
struct  BoardIndex
{
char    swVersion;          /*软件版本号*/
char    localIP[16];        /*本地 IP*/
int     localPortCMD;       /*和控制端的通信端口*/
char    swDiscri[100];      /*采集软件描述*/
};
```

这部分也列入 usrParamInit()和 usrParamShow()的操作。

例 6-1 给出了这种一个用户入口函数工作模式下的初始化部分的软件框架。

例 6-1　用户程序入口函数及其相关部分软件框架（board.h 部分和 usrSlot0Root.c）

```
/*************************************************************************
 chengjy@felab, copyright 2002-2004
 board.h (部分)
 用户程序入口函数需要的常数定义部分
 *************************************************************************/
#ifndef _BOARD_H
#define _BOARD_H

/*bootline 在内存中的位置，与 BSP 相同*/
#define LOCAL_MEM_LOCAL_ADRS    0x0
#define BOOT_LINE_OFFSET     0x4200
#define BOOT_LINE_ADRS          \
                  ((char*) (LOCAL_MEM_LOCAL_ADRS+BOOT_LINE_OFFSET))

/*~~~~~~~~~~~~~~~~~~~~~~板子标志结构~~~~~~~~~~~~~~~~~~~~~~*/
struct  BoardIndex
{
```

```
char    swVersion;          /*软件版本号*/
char swDiscri[100];         /*软件描述*/
char localIP[16];           /*本地 IP*/
char remoteIP[16];          /*控制端 IP*/
int  localPort;             /*通信端口号*/
};

#define LOCAL_PORT_CMD          2001        /*命令通道端口号*/

/*~~~~~~~~~~~~~~~~~~~~~~板子工作环境~~~~~~~~~~~~~~~~~~~~~~*/
struct BoardWorkEnv
{
int     boardinit;      /*boardInit 单个位用来表示初始化状态*/
char    autoBootFlag;   /*标志采集板是完成阶段还是调试阶段*/
char    dmaAutoflag;    /*定义采集板的 dma 是由硬件自动控制还是零槽软件控制*/
};

#define BOARD_UNINIT                0x00
#define BOARD_F9656_INITED          0x01
#define BOARD_NET_INITED            0x02
#define BOARD_GENET_INITED          0x04
#define BOARD_FLASH_INITED          0x08
#define BOARD_SOFTPRAM_INITED       0x80
#define BOARD_INITED                0x9F

#define DMA_AUTO_DISABLE        0       /*软件控制*/
#define DMA_AUTO_ENABLE         1       /*硬件控制*/
#define DFT_DMA_AUTO            DMA_AUTO_ENABLE
#define USR_DMA_AUTO            DMA_AUTO_DISABLE

/*定义任务名*/
#define TNAME_NETINIT           "tNetInit"
/*用户任务堆栈大小*/
#define USER_STACK_SIZE         2000
/*taskSpawn 使用的用户任务优先级*/
#define TPRI_NETINIT            112

/*定义单板状态*/
#define STATUS_NORMAL           0x00
#define STATUS_ERROR            0x01
#define STATUS_WARNING          0x02
#define STATUS_INVALID          0x03
#define STATUS_UNAVAILABLE      0x04
#define STATUS_DISABLE          0x05
```

```
#endif   /*_BOARD_H*/

/*****************************************************************************
 chengjy@felab, copyright 2002-2004
 usrXXXRoot.c
 采集板的统  入口程序
 函数:
     void usrXXXRoot();
     void usrDefaultParamLoad();
     void usrParamShow();
 调用:
     netSvr.c
     extern void netInit();
 被调用:
     BSP usrConfig.c
     void usrRoot(char* pMemPoolStart, unsigned memPoolSize);
     netTask.c
 *****************************************************************************/
#include "vxWorks.h"
#include "taskLib.h"
#include "logLib.h"
#include "sysLib.h"
#include "stdLib.h"
#include "stdio.h"
#include "string.h"

#include "board.h"

extern void netInit();

void usrXXXRoot();
void usrDefaultParamLoad();
void usrParamShow();

struct BoardIndex   boardIndex;
struct BoardWorkEnv boardWorkEnv;

/*****************************************************************************
 void usrXXXRoot()
 函数说明:  用户程序入口函数，给灵活配置参数全局变量赋初值，并发起网络初始化任务，等待
            控制端连接
 参数:        无
 返回:        无
```

```
调用:
        void usrDefaultParamLoad();
        void usrParamShow();
    netSvr.c
        extern void netInit();
被调用:
        BSP usrConfig.c
        void usrRoot(char* pMemPoolStart, unsigned memPoolSize);
*************************************************************************/
void usrXXXRoot()
{
   usrDefaultParamLoad();
   usrParamShow();
   boardWorkEnv.autoBootFlag = 1;
   taskSpawn(TNAME_NETINIT,TPRI_NETINIT,0,USER_STACK_SIZE,
            (FUNCPTR)netInit,0,0,0,0,0,0,0,0,0,0);
}

/*************************************************************************
 void usrDefaultParamLoad()
 函数描述:    全局变量参数初始化
 参数:        无
 返回:        无
 调用:        无
 被调用:
        void usrXXXRoot()
 *************************************************************************/
void usrDefaultParamLoad()
{
    BOOT_PARAMS bootParam;

    /*初始化软件版本号和说明*/
    boardIndex.swVersion    = 1;
    sprintf(boardIndex.swDiscri,"chengjy: tornado2.0,vxWorks5.4, copyright
2002-2004\n");

    /*从bootline获得IP*/
    bootStringToStruct(BOOT_LINE_ADRS,&bootParam);
    /*填写从bootline中获得的IP*/
    boardIndex.localIP[0]       = '\0';
    strcpy(boardIndex.localIP, bootParam.ead);
    boardIndex.remoteIP[0] = '\0';
    strcpy(boardIndex.remoteIP, bootParam.had);
    /*通信端口号*/
```

```
    boardIndex.localPort = LOCAL_PORT_CMD;

    /*硬件工作环境初始化*/
    if(boardWorkEnv.autoBootFlag!=0)
        boardWorkEnv.dmaAutoflag = USR_DMA_AUTO;
     else
        boardWorkEnv.dmaAutoflag = DFT_DMA_AUTO;

    /*软件参数初始化完毕*/
    boardWorkEnv.boardInit = boardWorkEnv.boardInit | BOARD_SOFTPRAM_INITED;
}

/*************************************************************************
 void usrParamShow()
 函数描述:    打印本板的信息
 参数:        无
 返回:        无
 调用:        无
 被调用:
              void usrXXXRoot()
              网络命令控制改变全局变量的命令对应函数
 *************************************************************************/
void usrParamShow()
{
    /*如果本板还没有初始化，则直接退出*/
    if(!(boardWorkEnv.boardInit & BOARD_SOFTPRAM_INITED))
    {
        printf("usrParamShow: warning: board is not initialized yet\n");
        return;
    }

    /*打印本板信息*/
    printf("usrParamShow:\n");
    printf("**************board XXX ************\n");
    printf(" software version: %d\n",boardIndex.swVersion);
    printf(" local IP:%s, local port %d\n",boardIndex.localIP,boardIndex.
localPort);
    printf(" remote IP should be:%s\n",boardIndex.remoteIP);
    printf(" %s\n",boardIndex.swDiscri);

    if(boardWorkEnv.boardInit & BOARD_NET_INITED)
        printf(" *net is initialized\n");
    else
        printf(" net is not intialized\n");
```

```
    if(boardWorkEnv.boardInit & BOARD_F9656_INITED)
        printf(" *9656 is initialized\n");
    else
        printf(" 9656 is not initialzed\n");
    if(boardWorkEnv.boardInit & BOARD_GENET_INITED)
        printf(" *geNet is initialized\n");
    else
        printf(" geNet is not intialized\n");
    if(boardWorkEnv.boardInit & BOARD_FLASH_INITED)
        printf(" *flash is initialized\n");
    else
        printf(" flash is not intialized\n");
    printf("\n");
    printf(" dma: ");
    if(boardWorkEnv.dmaAutoflag == DMA_AUTO_DISABLE)
        printf("not auto, controled by software\n");
    else
        printf("auto, controlled by hardware\n");
    printf("\n");
}
```

6.4 总结

本章介绍了如何提取出用户程序的入口函数，以及入口函数需要进行的操作。提出了用全局结构体的方法来控制可变参数，并实现网络配置，以提高软件的灵活性。还介绍了统一的任务名、任务优先级和参数管理方法。

第7章 设计完成——自启动的用户程序

在VxWorks操作系统的调试完成之后，可以将其烧录到采集板的板载ROM中，并改变Bootrom的默认启动方式，使其从Flash加载VxWorks映像，提高启动速度。在用户程序完成调试并归一到唯一的用户程序入口函数之后，可以设定VxWorks在启动的最后阶段发起用户程序，在将其烧录到Flash之后，就可以实现VxWorks加用户程序的上电自启动系统，完全脱离shell而用控制端软件全权控制。此外，还可以将某些需要下电保存的参数（例如启动行）保存在Flash里，使软件具有记忆功能。

7.1 Flash操作

Flash是一种电可擦除的下电仍然保存其内容的器件，由于它同时具备可以软件控制读写和掉电数据不丢失两种特性，常常被用来保存操作系统VxWorks的映像文件。对于Flash来说，软件操作归结起来就是向某个特征地址填写特定的操作字，然后通过回读特定地址并判断返回值，获得操作结果或者操作是否成功的标志。

在Flash操作时，需要注意Flash地址到CPU地址的映射关系，根据Flash种类和Flash地址线的接法不同，Flash地址和CPU地址可能不是一一对应的。

7.1.1 Intel28F320C3系列Flash

不同Flash芯片的具体操作可能不同，但操作方式大体一致，本章中介绍的Flash操作是基于intel28F320C3进行的。此系列的Flash还有28F800C3、28F160C3和28F640C3等，分别对应的容量为8Mbit、16Mbit、32Mbit和64Mbit。对于VxWorks操作系统，一般大小在1Mbyte左右，如果再加上用户程序的大小，可以考虑16Mbit的intel28F160C3。如果希望留出空间为将来的扩展做准备，推荐使用intel28F320C3。

它是一款16bit位宽的芯片，也就是说，当其地址线上的地址加一时，将给出16bit数据，这和CPU的地址加1对应8bit数据不同。采用图7-1所示的硬件接法，实现CPU地址与flash地址的一一对应，即CPU地址加4，flash给出32bit数据。这样使用了4片flash，总容量达到了32Mbyte。

对于不需要这么大空间的情况，可以只焊接第一片flash，此时CPU地址加8，flash给出16bit数据。根据地址线的选通情况可以看出，CPU地址应该是8的整数倍，本章中讨论的就是这种情况。为此，定义宏操作来进行Flash地址到CPU地址的转换：

```
#define FLASH2CPU_ADDR(flashAddr)        \
                ((FLASH_VALUE*)((UINT)(flashAddr)*8+FLASH_BASE_ADDR))
```

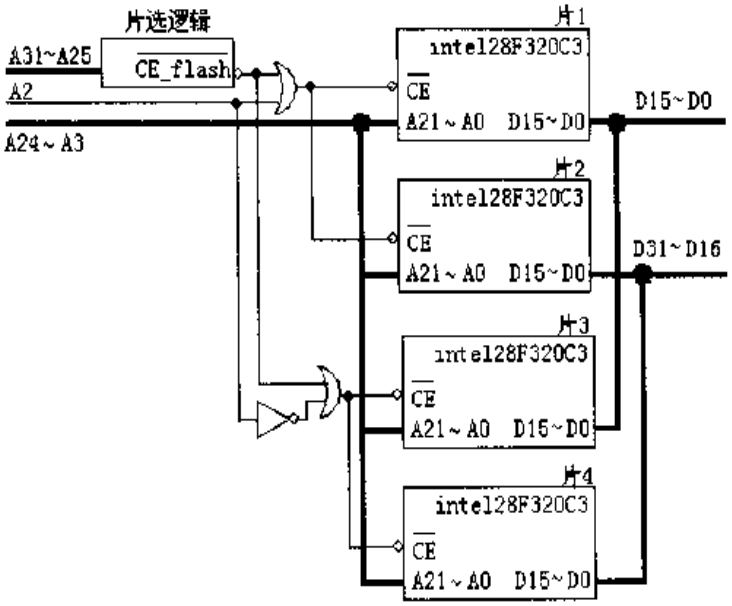


图 7-1 CPU 地址与 flash 数据完全对应的硬件接法

其中 FLASH_VALUE 表示数据有效宽度为 16bit，定义为：

```
#define FLASH_VALUE                    unsigned short
```

FLASH_BASE_ADDR 为 flash 基地址，即 flash 零地址对应到的 CPU 地址。

intel28F320C3 的空间被分成多个块（block），这些块又被划分成两个部分。第一部分从 Flash 的起始地址到 Flash 地址 0x00008000，包括 8 个大小为 8Kbyte 的块，被称为参数块（parameter block），可以用来存放经常更新的少量数据。第二部分即剩下的所有部分，分为 63 个大小为 64Kbyte 的块，被称为主块(main block)。由于两部分的大小不同，操作起来需要分别处理，且所有参数块的大小相当于一个主块，在本章介绍的函数中将统一使用主块，而不使用 8 个参数块，只将其看作一个主块进行地址转换。

对 Flash 的主要操作包括：读/写 Flash 内容，锁/解锁块，擦除块和读 Flash 标志等。所有的操作都可以归结到总线的读写两种基本操作。

读操作分为：读数据（read array），读标志(read configuration)，读状态(read status)和读信息(read query)。读数据即表示读出的是存储在 Flash 对应地址中的实际数据；读状态表示读出的是 Flash 的标志 ID，可以用来判断使用的是否是软件对应支持的 Flash；读状态一般用于判断对 Flash 进行某种操作后是否成功，凭借读出内容的某一 bit 判断执行结果；读信息则可以获得该 Flash 的具体信息，例如块的大小、电气规定值等特殊信息。上电后 Flash 默认处于读数据状态。

写操作主要用于对 Flash 的某个块进行解锁/擦除/烧录。为了保证用软件烧录的成功，一般在烧录前，先要对 Flash 进行解锁和擦除操作。这些写操作必须配合读状态操作进行。

程序中需要使用到的操作方法见表 7-1，其中地址和数据的具体关系请参阅光盘中给出的 Flash 说明书（28f320.pdf）和后面的例程 7-1。

表 7-1 基本的 Flash 操作

命 令	第一个操作周期			第二个操作周期		
	操 作	地 址	数 据	操 作	地 址	数 据
Read array	写	任意	0xFF	/	/	/
Read Configuration	写	任意	0x90	读	标志地址	标志数据
Read Status Register	写	任意	0x70	读	任意	状态数据
Programm	写	任意	0x40	写	目的地址	数据
Block Erase/Confirm	写	任意	0x20	写	块首地址	0xD0
Unlock Block	写	任意	0x60	写	块首地址	0xD0

进行操作后的状态见表 7-2。

表 7-2 Flash 操作及各状态转化对照表

当前状态	读数据	写 0xFF	写 0x90	写 0x70	写 0x40	写 0x20	写 0x60	写 0xD0
Read Array	flash 存储数据	Read Array	Read Config.	Read Status	Prog. setup	Erase setup	Lock setup	Read Array
Read Status	状态	Read Array	Read Config.	Read Status	Prog. setup	Erase setup	Lock setup	Read Array
Read Config.	标志 ID	Read Array	Read Config.	Read Status	Prog. setup	Erase setup	Lock setup	Read Array
Prog. setup	状态	Program						
Prog. (not done)	状态	Program (not done)						
Prog. (done)	状态	Read Array	Read Config.	Read Status	Prog. setup	Erase setup	Lock setup	Read Array
Erase setup	状态	出错	出错	出错	出错	出错	出错	Erase (not done)
Erase (not done)	状态	Erase (not done)						
Erase (done)	状态	Read Array	Read Config.	Read Status	Prog. setup	Erase setup	Lock setup	Read Array
Lock setup	状态	出错	出错	出错	出错	出错	出错	解锁完毕

擦除和烧录的流程比较复杂，见图 7-2。

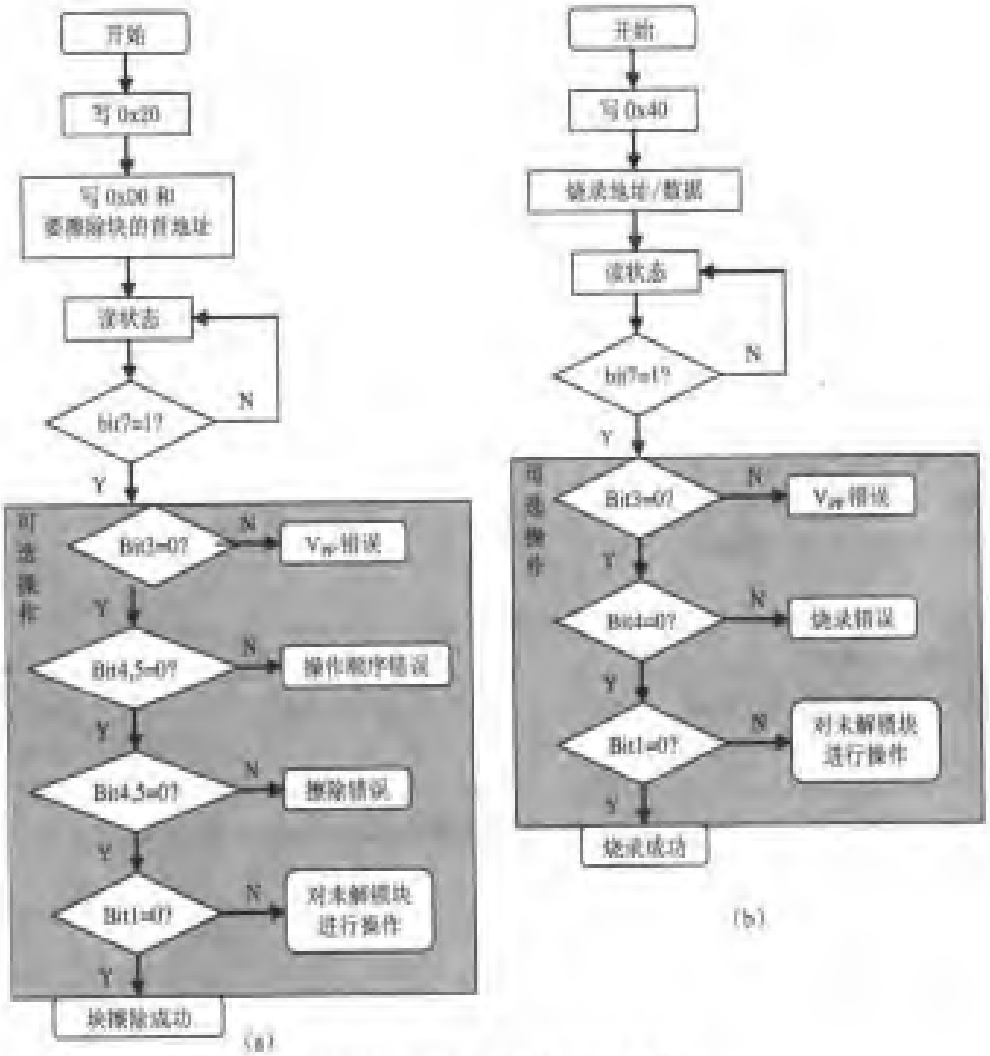


图 7-2 Flash 擦除和烧录流程图

(a) 擦除流程图；(b) 烧录流程图

7.1.2 Flash 基本操作例程

例 7-1 给出了 Flash 基本操作的函数，包括 Flash ID 检查、解锁、擦除和数据的烧录和读写。使用了信号灯进行保护，避免不同函数同时对 Flash 的进行操作。

例 7-1 flash intel28F320C3 的基本操作（board.h（部分）和 flash.c）

```
/*****************************************************************************
 chengjy@felab, copyright 2002-2004
 board.h
 flash 操作常数定义
 *****************************************************************************/
#ifndef _BOARD_H
#define _BOARD_H
```

```
/*~~~~~~~~~~~~~~~~~~~~~~板子工作环境~~~~~~~~~~~~~~~~~~~~~~*/
struct BoardWorkEnv
{
 int boardInit;                /*boardInit 单个位用来表示初始化状态*/
};

#define BOARD_FLASH_INITED      0x08

/*定义单板状态*/
#define STATUS_NORMAL           0x00
#define STATUS_ERROR            0x01
#define STATUS_WARNING          0x02
#define STATUS_INVALID          0x03
#define STATUS_UNAVAILABLE      0x04
#define STATUS_DISABLE          0x05

/*~~~~~~~~~~~~~~~~~~~~FLASH 相关参数~~~~~~~~~~~~~~~~~~~~~~*/
/*FLASH 地址空间*/
#define FLASH_BASE_ADDR         ((UINT)0xFF000000)  /*FLASH 起始地址*/
#define FLASH_DATA_BLOCK_NUM    63                  /*实际包含 63 块*/
#define FLASH_DATA_BLOCK_SIZE_W ((UINT)0x00008000)  /*单片容量,单位 WORD*/
#define FLASH_DATA_CPU    1 /*vxWorks 存放的 block,main block0 和 8 个 para block
                            空出*/
#define FLASH_DATA_BOOTLINE     30              /*bootline 存放的 block*/

/*FLASH 操作参数*/
#define FLASH_VALUE         unsigned short      /*FLASH 位宽 16bit*/

/*FLASH 状态转换参数*/
#define FLASH_CC_RESERVED               ((FLASH_VALUE)0x0000)
#define FLASH_CC_READ_ARRAY             ((FLASH_VALUE)0x00FF)
#define FLASH_CC_PROGRAM_SETUP          ((FLASH_VALUE)0x0040)
#define FLASH_CC_ALTERPROG_SETUP        ((FLASH_VALUE)0x0010)
#define FLASH_CC ERASE_SETUP            ((FLASH_VALUE)0x0020)
#define FLASH_CC_ERASE_CONFIRM          ((FLASH_VALUE)0x00D0)
#define FLASH_CC_RESUME                 ((FLASH_VALUE)0x00D0)
#define FLASH_CC_SUSPEND                ((FLASH_VALUE)0x00B0)
#define FLASH_CC_READ_STATUS            ((FLASH_VALUE)0x0070)
#define FLASH_CC_CLEAR_STATUS           ((FLASH_VALUE)0x0050)
#define FLASH_CC_READ_IDENTIFY          ((FLASH_VALUE)0x0090)
#define FLASH_CC_READ_QUERY             ((FLASH_VALUE)0x0098)
#define FLASH_CC_CONFIG_SETUP           ((FLASH_VALUE)0x0060)
#define FLASH_CC_CONFIG_CONFIRM         ((FLASH_VALUE)0x00D0)
```

```
/*状态寄存器检测*/
#define FLASH_CC_SET_CHECK1     ((FLASH_VALUE)0x0080)   /*program 结束*/
#define FLASH_CC_SET_CHECK2     ((FLASH_VALUE)0x001A)   /*program 可能出错的
                                位置*/
#define FLASH_CC_ERASE_CHECK1   ((FLASH_VALUE)0x0080)   /*erase 结束*/
#define FLASH_CC_ERASE_CHECK2   ((FLASH_VALUE)0x0010)   /*erase 可能出错的位
                                置*/

/*FLASH 芯片类型*/
#define FLASH_28F320C3_ID       ((FLASH_VALUE)0x0089)
#define FLASH_28F320C3_IDENT_T  ((FLASH_VALUE)0x88C4)
#define FLASH_28F320C3_IDENT_B  ((FLASH_VALUE)0x88C5)

/*flash 地址转换到 cpu 地址*/
#define FLASH2CPU_ADDR(flashAddr)                \
        ((FLASH_VALUE*)((UINT)(flashAddr)*8+FLASH_BASE_ADDR))

/*flash 操作最长时间*/
#define FLASH_ERASE_BLOCK_TIME 5000         /*最多 50000 次*/

#endif  /*_BOARD_H*/

/*****************************************************************************
 chengjy@felab, copyright 2002-2004
 flash.c
 flash 操作部分。提供 FLASH 的写入和读出，暂时不支持锁定。
 函数:
        void flashInit();
        char flashIDCheck();
        char flashDataSet(int dataNum,char *pBuff,int buffLen);
        char flashDataGet(int dataNum,char *pBuff,int buffLen);
        char flashDataErase(int dataNum, int eraseLen);
        char flashUnLock(int dataNum, int unLockLen);
        void flashShutDown();
 调用:      无
 被调用:
        netTask.c
            STATUS t0x1600(char*pBuff);
        BSP bootConfig.c
            LOCAL char autoboot (int timeout);
 说明:  现有的配置模式是基于如下地址处理方式的:对单片 flash,地址增加 1,可以获取到
        16bit 数据,而这个地址的提供必须是 cpu 地址加 8,相当于 cpu 地址加 8,用 unsigned
    short 型可以获取 2byte 数据。使用在 board.h 中定义的 FLASH2CPU_ADDR(flashAddr)
    宏来转换地址。
```

```
 ************************************************************************/
#include "vxworks.h"
#include "semLib.h"
#include "taskLib.h"
#include "logLib.h"
#include "stdio.h"
#include "string.h"

#include "board.h"

void flashInit();
char flashIDCheck();
char flashDataSet(int dataNum,char *pBuff,int buffLen);
char flashDataGet(int dataNum,char *pBuff,int buffLen);
char flashDataErase(int dataNum, int eraseLen);
char flashUnLock(int dataNum, int unLockLen);
void flashShutDown();

extern struct BoardWorkEnv  boardWorkEnv;

SEM_ID semFlash;    /*flash 保护信号灯，在对 flash 操作前获取，操作结束释放*/

/************************************************************************
 void flashInit()
 函数描述:    Flash 初始化。使用 boardWorkEnv.boardInit 和 BOARD_FLASH_INITED 控制
        重新初始化。首先检查 flash 的 ID，然后创建信号灯。
 参数:        无
 返回:        无
 调用:
              char flashIDCheck()
 被调用:
         netTask.c
             STATUS t0x1600(char*pBuff);
 ************************************************************************/
void flashInit()
{
    /*如果已经初始化了，不再重复进行*/
    if(!(boardWorkEnv.boardInit&BOARD_FLASH_INITED))
    {
        /*创建信号灯*/
        semFlash = semBCreate(SEM_Q_FIFO,SEM_EMPTY);
        if(semFlash == NULL)
            logMsg("flashInit: unable to create semFlash\n",0,0,0,0,0,0);
            return;
```

```
        }

        /*检查芯片类型*/
        if(flashIDCheck()!=STATUS_NORMAL)
        {
            logMsg("flashInit: only 28F320C3 is acceptable\n",0,0,0,0,0,0);
            semDelete(semFlash);
            return;
        }

        /*设置为 read array 模式*/
        *FLASH2CPU_ADDR(0) = FLASH_CC_READ_ARRAY;

        /*释放 flash 控制权*/
        semGive(semFlash);

        boardWorkEnv.boardInit = boardWorkEnv.boardInit | BOARD_FLASH_INITED;
    }
}

/*************************************************************************
 char flashDataSet(int dataNum,char *pBuff,int buffLen);
 函数描述:    将从 pBuff 开始的 buffLen 长度的数据写入 Flash 中 dataNum 对应的地址。使
         用 datatNum 对应 FLASH 的块号。如果 buffLen 不是 4 的整数倍。则将最后的数据
         填在高位上。在写之前必须保证对应的块已经经过了解锁和擦除。
 参数:        dataNum,    flash 的块序号，取值范围从 1~63。
              pBuff,      需要写到 flash 的数据的内存首地址。
              buffLen,    需要写到 flash 的数据的长度，单位 byte。
 返回:        如果成功写入返回 STATUS_NORMAL，否则返回对应的出错标志。
 调用:        无
 被调用:
         netTask.c
             STATUS t0x1600(char*pBuff);
 *************************************************************************/
char flashDataSet(int dataNum,char *pBuff,int buffLen)
{
    char stateReturn=STATUS_NORMAL;
    int iBuff,iBuffEnd;

    FLASH_VALUE *addr;
    FLASH_VALUE value;


    /*获取 flash 控制权*/
```

```
if(semTake(semFlash,100)==ERROR)
{
    printf("flashDataSet: unable to get semFlash\n");
    return(STATUS_ERROR);
}

/*计算是否过界,如果过界取到最后一个字节*/
iBuffEnd = (FLASH_DATA_BLOCK_NUM-dataNum)*
        FLASH_DATA_BLOCK_SIZE_W*2;
if(buffLen> iBuffEnd)
{
    printf("flashDataSet: no enough place for write\n");
    stateReturn = STATUS_WARNING;
}
else
    iBuffEnd = buffLen;

if(stateReturn == STATUS_NORMAL)
{
    iBuffEnd = ((unsigned int)(buffLen/2))*2;   /*填充长度取整*/

    printf("flashDataSet: addr 0x       ");

    /*首次的地址和数据*/
    addr = FLASH2CPU_ADDR(dataNum*FLASH_DATA_BLOCK_SIZE_W);
    value = pBuff[0]*0x0100+pBuff[1];
    iBuff=0;
    while(iBuff<buffLen)
    {
        /*设置为 program 格式*/
        *addr = FLASH_CC_PROGRAM_SETUP;
        *addr = value;

        /*计算下一次的地址和数据*/
            iBuff=iBuff+2;
        value = pBuff[iBuff]*0x0100+pBuff[iBuff+1];
        addr = addr+4;

        if(((UINT)addr&0xFFFF)==0)
            printf("\b\b\b\b\b\b\b\b%8x",(UINT)addr);

        /*检查状态*/
        *addr = FLASH_CC_READ_STATUS;
        while(!((*addr)&FLASH_CC_SET_CHECK1))
```

```
            {
                *addr = FLASH_CC_READ_STATUS;
            }
            if(((*addr)&FLASH_CC_SET_CHECK2)!=0x0000)
            {
                printf("\nflashDataSet: error in set set addr %x value %x in
                        time\n",(int)addr,value);
                stateReturn = STATUS_ERROR;   /*出错，停止配置，跳出*/
                break;
            }
        }

        printf("\n");
        if(stateReturn==STATUS_NORMAL)
            printf("flashDataSet: finished\n");
    }
    else
    {
        printf("\nflashDataSet: no enough place for file,quit\n");
        stateReturn = STATUS_ERROR;
    }

    /*配置为 array read*/
    *FLASH2CPU_ADDR(0) = FLASH_CC_READ_ARRAY;

    /*释放 FLASH 控制权*/
    semGive(semFlash);
    return(stateReturn);
}

/*************************************************************************
 char flashDataGet(int dataNum,char *pBuff,int buffLen)
 函数描述:    从 Flash 中 dataNum 对应地址开始读取 buffLen 个字节到 pBuff。
 参数:        dataNum,    flash 的块序号，取值范围从 1~63
              pBuff,  从 flash 读取到的数据存储的内存首地址。
              buffLen,    从 flash 读取的数据的长度，单位 byte。
 返回:        如果成功读取返回 STATUS_NORMAL，否则返回对应的出错标志。
 调用:        无
 被调用:
         BSP bootConfig.c
             LOCAL char autoboot (int timeout)
 *************************************************************************/
char flashDataGet(int dataNum,char *pBuff,int buffLen)
{
```

```
char stateReturn=STATUS_NORMAL;
int iBuff,iBuffEnd;
FLASH_VALUE * addr;
FLASH_VALUE value;

/*获取 flash 控制权*/
if(semTake(semFlash,100)==ERROR)
{
   printf("flashDataGet: unable to get semFlash\n");
   return(STATUS_ERROR);
}

/*计算是否过界,如果过界取到最后一个字节*/
iBuffEnd = (FLASH_DATA_BLOCK_NUM-dataNum)*
           FLASH_DATA_BLOCK_SIZE_W*2;

if(buffLen> iBuffEnd)
{
   printf("flashDataGet: no enough place for read\n");
   stateReturn = STATUS_WARNING;
}
else
   iBuffEnd = buffLen;

if(stateReturn!= STATUS_ERROR)
{
   /*读数据*/
   for(iBuff=0;iBuff<iBuffEnd;iBuff=iBuff+2)
   {
      addr =
      FLASH2CPU_ADDR(dataNum*FLASH_DATA_BLOCK_SIZE_W+iBuff/2);

      /*设置为 read array, 然后读数据*/
      *(addr) = FLASH_CC_READ_ARRAY;
      value = *addr;    /*因为是直接赋值,读取时也直接读取,保证字序一致*/

      if(iBuffEnd-iBuff>=2)/* 普通情况*/
      {
         pBuff[iBuff]   = (char)((value>>8) &0xFF);
         pBuff[iBuff+1]= (char)(value&0xFF);
      }
      else  /*不是整块情况,取高位*/
      {
         pBuff[iBuff]   = (char)((value>>8) &0xFF);
```

```
            }
        }
        /*设置为 read array 状态*/
        *FLASH2CPU_ADDR(0) = FLASH_CC_READ_ARRAY;
    }

    /*释放 flash 控制权*/
    semGive(semFlash);

    return (stateReturn);
}

/*****************************************************************************
 char flashDataErase(int dataNum,int eraseLen)
 函数描述:    将从 dataNum 开始的地址擦除，如果 eraseLen 不是块长度的整数倍，仍然将
        最后一块整体擦除。
 参数:        dataNum,    flash 的块序号，取值范围从 1~63。
              eraseLen,   需要擦除的总长度和将要写 flash 的长度对应相等。
 返回:        如果成功擦除返回 STATUS_NORMAL，否则返回对应的出错标志。
 调用:
              char flashUnLock(int dataNum, int unLockLen)
 被调用:
        netTask.c
              STATUS t0x1600(char*pBuff);

*****************************************************************************/
char flashDataErase(int dataNum, int eraseLen)
{
    FLASH_VALUE *addr;
    FLASH_VALUE value;
    int iBlock,lenBlock,endBlock;
    int eraseTime = 0;
    char stateReturn = STATUS_NORMAL;
    int eraseDelay;

    /*因为解锁函数内部调用了 semTake 和 semGive,
      因此必须放在 flashDataErase 函数调用 semTake 前*/
    if(flashUnLock(dataNum,eraseLen)!=STATUS_NORMAL)
    {
        printf("unable to unlock block from %d for %d bytes\n",
        dataNum,eraseLen);
        return(STATUS_ERROR);
        }
```

```
    /*获取 flash 控制权*/
    if(semTake(semFlash,100)==ERROR)
    {
        printf("flashDataErase: unable to get semFlash\n");
        return(STATUS_ERROR);
    }

    /*计算需要擦除的块数量*/
    lenBlock = (eraseLen-1)/(FLASH_DATA_BLOCK_SIZE_W*2)+1;

    endBlock = dataNum+lenBlock;
    if(endBlock>FLASH_DATA_BLOCK_NUM)    /*超过范围，擦除到最后一块为止*/
    {
        printf("flashDataErase: no more place for erase %d blocks\n",lenBlock);
        endBlock = FLASH_DATA_BLOCK_NUM;
        stateReturn = STATUS_WARNING;
    }

    printf("flashDataErase: block number 0x  ");

    /*逐块擦除*/
    for(iBlock=dataNum;iBlock<endBlock;iBlock++)
    {
        printf("\b\b%2x",iBlock);
        addr =
FLASH2CPU_ADDR((FLASH_VALUE*)((UINT)iBlock*FLASH_DATA_BLOCK_SIZE_W));
        /*开始擦除*/
        *addr = FLASH_CC_ERASE_SETUP;
        *addr = FLASH_CC_ERASE_CONFIRM;

        eraseTime = 0;
        /*读取状态*/
        while(1)
        {
            /*不断检查是否擦除完毕*/
            *addr = FLASH_CC_READ_STATUS;
            value = *addr;
            if((value & FLASH_CC_ERASE_CHECK1) == FLASH_CC_ERASE_CHECK1)
                break;     /*擦除完毕*/
            else
            {
                eraseTime++;
                for(eraseDelay=0;eraseDelay<10000;eraseDelay++);
                if(eraseTime ==FLASH_ERASE_BLOCK_TIME)    /*操作超时*/
```

```
                {
                    printf("\nflashDataErase: can't erase block %d in time\n",
                            iBlock);
                    stateReturn = STATUS_ERROR;
                    break;
                }
            }
        }
        if(stateReturn==STATUS_NORMAL)
        {
        /*检查是否正确擦除*/
            if((value & FLASH_CC_ERASE_CHECK2) != 0x0000)    /*擦除出错*/
            {
                printf("\nflashDataErase: error in erase block %d, value =%x\n",
                        iBlock,value);
            *   stateReturn = STATUS_ERROR;
            }
            /*正确擦除，回到 read array 状态*/
            *FLASH2CPU_ADDR(0) = FLASH_CC_READ_ARRAY;
        }
        else
            break;
    }
    printf("\n");
    printf("flashDataErase: erase finished\n");
    /*无论是否正确擦除，都回到 read array 状态*/
    *FLASH2CPU_ADDR(0) = FLASH_CC_READ_ARRAY;

    /*释放 flash 控制权*/
semGive(semFlash);

    /*全部擦除完毕*/
    return(stateReturn);
}

/*****************************************************************************
 char flashUnLock(int dataNum, int unLockLen)
 函数描述:    将从 dataNum 开始的地址解锁，如果 eraseLen 不是块长度的整数倍，仍然将
        最后一块解锁。
 参数:        dataNum,    flash 的块序号，取值范围从 1~63。
              unLockLen,  需要解锁的总长度，和将要写 flash 的长度对应相等。
 返回:        如果成功解锁返回 STATUS_NORMAL，否则返回对应的出错标志。
 调用:        无
 被调用:
```

```
            char flashDataErase(int dataNum, int eraseLen)
 ***************************************************************************/
char flashUnLock(int dataNum, int unLockLen)
{
    FLASH_VALUE *addr;
    int iBlock,lenBlock,endBlock;
    char stateReturn = STATUS_NORMAL;

    /*获取 flash 控制权*/
        if(semTake(semFlash,100)==ERROR)
    {
        printf("flashUnLock: unable to get semFlash\n");
        return(STATUS_ERROR);
    }

    /*计算需要解锁的块数量*/
    lenBlock = (unLockLen-1)/(FLASH_DATA_BLOCK_SIZE_W*2)+1;
    endBlock = dataNum+lenBlock;
    if(endBlock>FLASH_DATA_BLOCK_NUM)  /*超过范围，解锁到最后一块为止*/
    {
        printf("flashUnLock: no more place for unlock %d blocks\n",lenBlock);
        endBlock = FLASH_DATA_BLOCK_NUM;
        stateReturn = STATUS_WARNING;
    }

    /*逐块解锁*/
    for(iBlock=dataNum;iBlock<endBlock;iBlock++)
    {
        addr =
FLASH2CPU_ADDR((FLASH_VALUE*)((UINT)iBlock*FLASH_DATA_BLOCK_SIZE_W));
        /*开始解锁*/
        *addr = FLASH_CC_CONFIG_SETUP;
        *addr = FLASH_CC_CONFIG_CONFIRM;
        /*恢复到 read array 状态*/
        *addr = FLASH_CC_READ_ARRAY;
    }

    /*释放信号灯*/
    semGive(semFlash);
    return(stateReturn);
}

/***************************************************************************
 char flashIDCheck()
```

```
函数描述:    检查 flash 芯片类型。
参数:        无
返回:        如果是软件要求的 flash，返回 STATUS_NORMAL，否则返回 STATUS_ERROR。
调用:        无
被调用:
             void flashInit()
*****************************************************************************/
char flashIDCheck()
{
    FLASH_VALUE id1,id2;

    /*设置为 read identify 模式*/
    *FLASH2CPU_ADDR(0) = FLASH_CC_READ_IDENTIFY;

    /*读芯片代号*/
    id1  = *FLASH2CPU_ADDR(0);
    id2  = *FLASH2CPU_ADDR(1);

    /*根据芯片是否是 28F320B3 做出返回值*/
    if( (id1==FLASH_28F320C3_ID) &&
       ((id2==FLASH_28F320C3_IDENT_T)||(id2==FLASH_28F320C3_IDENT_B)) )
    return(STATUS_NORMAL);
    else
        return(STATUS_ERROR);
}

/*****************************************************************************
 void flashShutDown()
 函数描述:    关闭 FLASH 的操作，主要是解决一些在 SHELL 下重复使用 flashInit()的信号
              灯没有删除的问题，在这里删除信号灯，并将 flash 重新置到 read array 状态。
 参数:        无
 返回:        无
 调用:        无
 被调用:
        netTask.c
            STATUS t0x1600(char*pBuff);
 *****************************************************************************/
void flashShutDown()
{
    if(boardWorkEnv.boardInit&BOARD_FLASH_INITED)
    {
        /*恢复成 read array 状态*/
        *FLASH2CPU_ADDR(0) = FLASH_CC_READ_ARRAY;
        /*删除 flash 信号灯*/
```

```
            semDelete(semFlash);
            boardWorkEnv.boardInit=boardWorkEnv.boardInit&
            (~(BOARD_FLASH_INITED));
        }
    }
```

7.2 从 Flash 启动操作系统 VxWorks

为了从 Flash 启动映像文件，首先需要将映像文件烧录到板载 Flash，在修改 bootConfig.c，让 bootrom 从 Flash 读取 VxWorks，为了使 bootrom 能够直截了当地获取 VxWorks 映像文件的长度，在映像文件前添加 4 个 byte，用来表示其总长度，并一起烧进 Flash。

烧录到 Flash 的文件采用从 ELF 格式的 vxWorks 制作的二进制格式文件 vxWorks.bin，将“Tornado 安装目录\host\x86-win32\bin”目录下的 torvars.bat 文件拷贝到 bsp 目录下，然后在命令行下输入：

```
torvars
make VxWorks
    elfToBin <VxWorks> vxWorks.bin
```

这样可以制作 VxWorks.bin。由于将 vxWorks.bin 烧录入 Flash 并从 Flash 启动之后，target server 需要将对应的 VxWorks 作为 corefile 才能正常运行，而 vxWorks 直接放在 BSP 目录下，容易在修改 BSP 时被直接覆盖。因此，建议在 BSP 目录下建立子目录 flash，用来存放烧入 Flash 的 vxWorks.bin 对应的 vxWorks。还可以将 vxWorks 改名为 vxWorks.[板名]来区分多块采集板公用一个 BSP 时的各个 vxWorks，例如 vxWorks.FCIF。在此目录下建立 elf.bat，内容如下：

```
rem Command line build environments
set WIND_HOST_TYPE=x86-win32
set WIND_BASE=C:\Tornado
set PATH=%WIND_BASE%\host\%WIND_HOST_TYPE%\bin;%PATH%
elfToBin <vxWorks.FCIF> vxWorks.bin.FCIF
elfToBin <vxWorks.FCIFAUTO> vxWorks.bin.FCIFAUTO
```

前面的 4 行是 torvars 的内容，目的是建立 make 需要的环境变量，如果用户使用的 Tornado 操作系统或者安装目录于上面定义的不同，则应该参照 Tornado 安装后附带的 torvars.bat 文件做相应修改。后两行是分别将名称为 vxWorks.FCIF 和 vxWorks.FCIFAUTO 的 ELF 格式 vxWorks 制作成 vxWorks.bin，还可以将光盘中例 4-1 的 WinCltAll 的可执行程序 winCltAll.exe 拷贝到此目录下，专门用来将.bin 文件下载到 Flash。

用户程序部分需要做的工作是：通过网络配合 winCltAll 程序获得 vxWorks.bin 文件，并将其写入 Flash。为此，定义命令号 0x1600 作为烧录 Flash 的命令号，winCltAll 使用此命令将.bin 文件作为参数传递给用户程序，用户程序根据整个命令的长度推断出.bin 文件

的长度，将长度和文件一并写入 Flash。

需要修改的文件包括 netTask.c 和 netSvr.c。首先在 netTask.c 中添加 t0x1600 的申明和函数体。

```
char t0x1600(unsigned char*pBuff); /*Flash 程序下载*/
/*************************************************************************
char t0x1600(unsigned char*pBuff)
```

函数描述：将 vxWorks 文件（对于单板来说就时 vxWorks.bin）下载到 Flash，准备下次启动时使用。

网络返回：配置后发送发送 9byte 帧。

调用：　外部的 Flash 操作函数。

```
*************************************************************************/
char t0x1600(unsigned char*pBuff)
{
    char state;
    unsigned char *pSendBuff;
    int len;

    logMsg("t0x1600: begin...\n",0,0,0,0,0,0);

    /*len 表示 vxWorks.bin 的总长度*/
    len = pBuff[2]*0x01000000+pBuff[3]*0x00010000+pBuff[4]*0x00000100
        +pBuff[5]-8;
    /*在 vxWorks.bin 前添加 4byte 表示总长度信息，一起写入 Flash*/
    pBuff[4]=(unsigned char)((len>>24)&0xFF);
    pBuff[5]=(unsigned char)((len>>16)&0xFF);
    pBuff[6]=(unsigned char)((len>>8)&0xFF);
    pBuff[7]=(unsigned char)(len&0xFF);
    flashInit();
    if(!(boardWorkEnv.boardInit&BOARD_FLASH_INITED))
    {
        state = STATUS_ERROR;
    }
    else
    {
        state = flashDataErase(FLASH_DATA_CPU,len+4);
        if(state==STATUS_NORMAL)
            state = flashDataSet(FLASH_DATA_CPU,pBuff+6,len+4);
    }
    flashShutDown();

    /*网络返回*/
    pSendBuff = malloc(9*sizeof(char));
```

```
    if(pSendBuff!=NULL)
    {
        pSendBuff[8] = state;
        netCMDAdd(pSendBuff,9,0x1600,QUEUE_PRI_LOW);
    }
    else
    {
        logMsg("t0x1600: unable to malloc net return buff\n",0,0,0,0,0,0);
        state = STATUS_ERROR;
    }
    logMsg("t0x1600: ended\n",0,0,0,0,0,0);
    return(state);
}
```

然后在netSvr.c中添加外部函数的申明，并修改netCMDExplain()，使其响应命令号0x1600。

```
extern STATUS t0x1600(char*pBuff);      /*flash程序下载*/
/************************netCMDExplain()部分************************/
case 0x1600:
    state = t0x1600(pBegin);
    break;
```

完成之后，将用户程序编译下载运行，即可以用winCltAll.exe下载软件了，运行情况见图7-3。完整的用户程序参见光盘中的例7-2，这里就不再列出了。

图7-3 使用用户程序将vxWorks映像下载到板载Flash

BSP部分只需要修改Bootrom对应部分，因此修改bootConfig.c文件和config.h。由于bootConfig.c属于各BSP的公用目录，为了保证此BSP的改动不影响其他的，将“Tornado安装目录\target\config\all”目录下的所有文件拷贝到当前BSP目录下，并在makefile中添加：

```
#BOOTINIT       =./bootInit.c
#BOOTCONFIG     =./bootConfig.c
#USRCONFIG      =./usrConfig.c
```

将 bootConfig.c 等文件的路径设定为当前的 BSP 目录，成为当前 BSP 的专用文件。然后在 config.h 中添加控制从 flash 启动的宏定义。

```
#define BOOT_FROM_FLASH          /*系统从 FLASH 启动*/
```

将 flash.c 和 board.h 拷贝到 BSP 目录下，在 bootConfig.c 中直接包含这两个文件，且由于 Flash 的重复初始化控制变量被定义在 usrXXXRoot.c 中，而此文件没有被包含，因此在 bootConfig.c 中声明同样的全局变量，实际只使用了该结构体中的变量 boardInit。

```
#ifdef  BOOT_FROM_FLASH
#include "board.h"
#include "flash.c"
struct BoardWorkEnvboardWorkEnv;
#endif
```

最后修改函数 LOCAL char autoboot(int timeout)。首先添加局部变量声明：

```
#ifdef  BOOT_FROM_FLASH
    char tmpt[4];
    int vxWorksLen;
#endif
```

然后在语句 if (bootLoad (BOOT_LINE_ADRS, &entry) == OK)前添加从 Flash 读取 vxWorks.bin 文件的部分。

```
#ifdef BOOT_FROM_FLASH
if(flashInit()==STATUS_NORMAL) /*初始化 FLASH*/
{
    /*首先读取 vxWorks 的长度，存储在 FLASH_DATA_CPU 开始的头 4 个字节*/
    if(flashDataGet(FLASH_DATA_CPU,tmpt,4)!=STATUS_NORMAL)
        printf("error in getting vxWorks length\n");
    else
    {
        vxWorksLen = tmpt[0]*0x01000000+tmpt[1]*0x00010000
                   +tmpt[2]*0x00000100+tmpt[3];
        /*从 flash 中的第 FLASH_DATA_CPU 块开始读取 vxWorks 本身长度再加上头 4 个字
         节长度的字节*/
        printf("\nloading vxWorks for %d bytes from flash\n",vxWorksLen);
        if(flashDataGet(FLASH_DATA_CPU,(char *)(((UINT)entry)-4),
            vxWorksLen+4)
            !=STATUS_NORMAL)
            printf("error in getting vxWorks.bin from flash\n");
        else
        {
```

```
                /*开始运行 vxWorks*/
                printf("now running...\n");
                go(entry);
            }
        }
    }
    #endif
```

使用修改后的 BSP 重新制作 Bootrom，如果在打印出“Press any key to stop auto-boot...”开始启动倒计时之后不按下任何按键，则 bootrom 从 Flash 获取 vxWorks.bin 文件，将其加载到内存后启动。按任意键仍然可以选择从网络启动。

由于烧录 Flash 需要时间，因此在调试 VxWorks 期间，推荐仍然从网络启动，一旦 VxWorks 调试完毕，经过一次烧录，就可以从 Flash 启动，速度超过网络启动，且不再需要在 PC 端运行服务器程序 wftpd32.exe，只要保证烧录到 Flash 的.bin 文件是由 target server 中指定的 corefile 制作出来的，就仍然可以正常使用 target server。

实际上，可以将 VxWorks 进行的工作尽量简化，除了必要的串口网口和 PCI 总线初始化等，将更多的工作放在用户程序里进行，不仅可以增强 VxWorks 的通用性，而且可以提前 VxWorks 的完工日期，尽早地将其烧录入 Flash，提高工作效率。

7.3 结合用户程序的自启动系统

在用户程序调试完毕后，将其与 VxWorks 结合，制作成带用户程序的自启动操作系统，并用上一节介绍的方法烧录到 Flash，就可以实现不需要任何人工干预的自启动系统，脱离开发环境 Tornado，完全由控制端的软件控制。

- 基于静态链接的自启动 VxWorks 制作

用户程序在开发过程中使用单独的工程编译，编译结束生成一批和.c 文件同名的.o 文件，这些文件通过 target server 动态下载，这样提供便捷的调试环境，但需要主机环境的支持。当开发结束后，需要给操作人员提供最方便的启动方式，因此必须将用户程序也编译链接到 vxWorks.bin 中，在 vxWorks 启动后就直接启动用户程序。链接方式包括静态链接和动态链接两种。静态链接将提供的所有函数都链接到 vxWorks 中，而动态链接只链接被调用的函数，用户开发过程中用于测试而在最终系统中并不运行的函数将不被链接。

通过使用 nm[CPU 类型]可以查看包括 vxWorks 在内的 HEX 类型文件中含有的函数声明和全局变量，在命令行下输入：

```
nmppc --numeric-sort vxWorks >symTab.txt
```

可以将 vxWork 中含有的函数和全局变量写到文件 symTab.txt 中，可以分别查看静、动态链接产生的 vxWorks 对用户程序中的函数包含情况。这里使用命令 nmppc 是因为笔者使用的 CPU 属于 PowerPC 系列。

静态链接有两种方式：.c 的链接和.o 的链接。

（1）.c 的链接方式：先在 BSP 目录下建立用户程序文件夹 usrAPI，将所有用户源程序

拷贝到此目录下，然后在 usrConfig.c 的 usrRoot()函数结尾处调用用户程序，需要在 usrConfig.c 程序开始添加声明：#include "用户程序名.c"。

(2).o 的链接方式：同样建立目录 usrAPI，将在工程中编译生成的.o 文件拷贝到 usrAPI，在 makefile 中添加 LIB_EXTRA 的定义来将.o 包括到 vxWorks 中。例如用户程序生成的.o 有：a.o 和 b.o，则 makefile 中加入：LIB_EXTRA = usrAPI/a.o usrAPI/b.o，同样修改 usrRoot()函数，并在 usrConfig.c 程序开始添加外部函数声明：extern 函数类型 函数名（参数……）

为了避免和没有结合用户程序的 VxWorks 混淆，使用 dos 命令 rename，将 vxWorks 改名为 vxWorks.[板号 AUTO]，并拷贝到 flash 目录下，然后运行制作好的批处理 elf.bat，生成的 vxWorks.bin.[板号 AUTO]已经包含自启动的用户程序了。

- 基于动态链接的自启动 vxWorks 制作

动态链接使用.a 文件。由于动态链接方式也是最后使用的方式，因此这种方法的介绍将完全基于例 7-2 中的文件。

同样制作 usrAPI 目录并拷贝.o 文件，然后在此目录下建立批处理文件 makeA.bat。

首先是设定环境变量：

```
set WIND_HOST_TYPE=x86-win32
set WIND_BASE=C:\Tornado
set PATH=%WIND_BASE%\host\%WIND_HOST_TYPE%\bin;%PATH%
```

然后使用 ar[CPU 类型]将所有.o 加入到一个档案库文件 usrAPI.a 中。使用 arppc 同样是因为笔者采用的 CPU 属于 PowerPC 类型。

```
arppc -crusv usrAPI.a flash.o netQueue.o netSvr.o netTask.o usrXXXRoot.o
```

运行该批处理文件将在 usrAPI 目录下生成档案库文件 usrAPI.a。

在 config.h 中定义包含用户程序：

```
/*定义用户程序接口，如果定义了此宏，usrRoot()结束前会发起用户初始程序，
需要makefile支持，定义LIB_EXTRA      = usrAPI/usrAPI.a*/
#define INCLUDE_USRAPI_BOOT
```

在 usrConfig.c 中添加外部函数声明：

```
#ifdef INCLUDE_USRAPI_BOOT
extern void usrXXXRoot();
#endif
```

在 VxWorks 结束前，即函数 void usrRoot(char *pMemPoolStart,unsigned memPoolSize)结束前，添加发起用户程序的代码：

```
taskSpawn("usrAPIRoot",20,0,2000,(FUNCPTR)usrXXXRoot,0,0,0,0,0,0,0,0,0,0);
```

最后在 makefile 中加入 LIB_EXTRA = usrAPI/usrAPI.a，重新制作 VxWorks 和对应的.bin 文件即可。

制作.a 文件还有另一种可选方法，即直接在 Tornado 下选择编译后的输出文件为.a 文件而非.out 文件。在 Tornado 的 Workspace 窗口下，选择 build，点右键调出 properties 对话

框，选择 Rules，然后在 Rule 下拉框下选择 archive，点击 OK 确定选择。然后点右键，选择 build'xxx.a'即可，见图 7-4。

图 7-4 在 Tornado 环境下编译.a 文件

- 不同链接方法的对比

基于.c 的链接方法将所有用户编写的源程序保留在 BSP 文件夹中，移植时不利于程序的版权保密。而在不同 BSP 之间用户程序的移植更是不方便，虽然这种方法不需要对 makefile 进行任何改动，修改的只是.c 文件，比较容易理解，但仍然不推荐使用。

基于.o 的链接方法保留了全部的的测试函数接口，并且移植时提供编译过的.o 文件也比提供源代码.c 文件利于保密。用户源程序和 BSP 源程序分开，文档结构整齐，特别是对源程序比较庞杂的时候，使用库文件.o 或者.a 更加有利于文件管理。

基于.a 的链接方法只保留必要的函数接口，自动优化了 vxWorks 的文件大小，因此将对单板的 Flash 大小要求降到最低。这种方法同时使用户程序在不同 BSP 间的移植变得非常简单。对于一个新的 BSP，所需要的就是把 usrAPI 目录和 flash 目录整个拷贝过去，用基于这个 BSP 类型的编译方法制作.o 文件覆盖 usrAPI 目录下的.o，拷贝原 BSP 中 makefile 中 LIB_EXTRA = usrAPI/usrAPI.a 到新 BSP 目录下的 makefile 中，使用原 BSP 中 usrConfig.c 和 bootConfig.c 中的 autoboot()函数和 usrRoot()函数结尾覆盖新 BSP 中相应的部分，最后运行 makeA.bat 制作 usrAPI.a，再制作 vxWorks，用 elf.bat 制作 vxWork.bin，就将用户程序完全移植了。全部移植工作只是一些文件和代码的复制以及批处理文件的运行，十分方便。如果需要将新的.o 加入档案库文件，也只需要拷贝此文件到 usrAPI 目录，在 makeA.bat 中的 arppc 的各个.o 文件后加上这个新的.o，然后再运行一次 makeA.bat 就可以了。

唯一需要注意的是，如果在生成了 bootrom 和 vxWorks 之后，希望通过更新.o 来更新 vxWorks 中包含的用户程序，不仅需要重新运行 makeA.bat 更新 usrAPI.a，还需要在命令行下输入 make clean，将 vxWorks 和 BSP 目录下（不含子目录，因此用户程序生成的.o 程序不受影响）所有的*.o、*.rpo、vxWorks*、bootrom*、stdt.c、symTbl.c 和 depend.[*bspName*]

删除，然后运行 makeA.bat。因为编译时会自动检查 bootrom 和 vxWorks 的存在以及原 BSP 目录下文件有没有改变，如果没有改变，则认为没有重新编译的必要。这样运行 makeA.bat 不会更新 bootrom 和 vxWorks，而使用 make clean 之后，系统检测不到 bootrom 和 vxWorks 的存在，就会重新编译了。而如果使用前面介绍的方法，每次生成 vxWorks 都被重命名并移动到 flash 目录下，则不会出现这个问题。

7.4 用户参数的下电保存

最经常和最普通的需要改动的用户参数是 VxWorks 的启动行，也就是 config.h 中的 DEFAULT_BOOT_LINE。该启动行中保存着 bootrom 引导 VxWorks 需要的参数，如 IP、用户名等。在 bootrom 启动到打印“Press any key to stop auto-boot...”时，按下回车键，可以查看和手动修改启动参数。输入 p 将解析并打印启动行，输入 c 可以逐项修改启动参数。但这些参数只在当前的启动时有效，一旦系统下电，再次上电时使用的仍然是 config.h 中默认启动行的参数。如果需要修改参数，则需要重新制作和烧制 bootrom，比较麻烦。

有了 Flash 的操作后，就可以将修改后的启动行保存到 Flash，上电首先检查 Flash，如果保存有有效的启动行，则使用 Flash 中的参数，否则才使用 config.h 中定义的参数。系统启动行被保存在内存的固定位置，在 configAll.h 中有下面的定义：

```
#define LOCAL_MEM_LOCAL_ADRS    0x0
#define BOOT_LINE_OFFSET        0x4200
#define BOOT_LINE_ADRS          ((char*) (LOCAL_MEM_LOCAL_ADRS +   \
                                BOOT_LINE_OFFSET))
#define BOOT_LINE_SIZE          300
```

启动行被保存在内存的 BOOT_LINE_ADRS 处，长度为 BOOT_LINE_SIZE 个 byte，系统启动使用的就是这些参数。修改 bootConfig.c，在用户更新内存中的启动行后，将内存中的启动行保存到 Flash。首先在 config.h 中添加控制宏定义：

```
#define  BOOTLINE_IN_FLASH      /*bootline 存放在 flash 中*/
```

系统将启动行装载到内存使用的函数为 bootConfig.c 中的 void usrBootLineInit (int startType)，修改此函数，使其在将 DEFAULT_BOOT_LINE 加载到内存之后，再尝试加载 Flash 中的启动行。

```
    void usrBootLineInit (int startType)
    {
#ifdef BOOTLINE_IN_FLASH
        char bootLine[BOOT_LINE_SIZE];
#endif
        if (startType & BOOT_CLEAR)
        {
            if ( *BOOT_LINE_ADRS == EOS )
            {
```

```
                strcpy (BOOT_LINE_ADRS, DEFAULT_BOOT_LINE);
                /*从 flash 中读取默认的启动行*/
#ifdef BOOTLINE_IN_FLASH
                flashInit();
                flashDataGet(FLASH_DATA_BOOTLINE,bootLine, BOOT_LINE_SIZE);
                if((strstr(bootLine,"h=")!=NULL) && (strstr(bootLine,"e=")
                    !=NULL))
                {
                    strcpy(BOOT_LINE_ADRS,bootLine);
                }
#endif
            }
        }
    }
```

这里也调用了 Flash 的初始化函数 flashInit()，由于有 boardWorkEnv.boardInit 控制，不会重复初始化，因此不必担心 LOCAL char autoboot(int timeout)函数中的 flashInit()和这里的 flashInit()初始化两次硬件造成错误。

用户手动修改内存中启动行的部分包含在函数 LOCAL void bootCmdLoop (void)中。首先在其中添加局部变量申明：

```
#ifdef BOOTLINE_IN_FLASH
    char saveCmd[20];
#endif
```

然后修改 switch (*(pLine++))的选项"case 'c':"如下：

```
    case 'c':       /* change boot params */
    bootParamsPrompt (BOOT_LINE_ADRS);  /*系统函数，按照用户的输入更改
                                          内存中的启动行*/
#ifdef BOOTLINE_IN_FLASH
    printf("save change to flash? Y/N\n");
    scanf("%s",&(saveCmd[0]));
    if(strchr(&(saveCmd[0]),'Y')!=NULL ||strchr(&(saveCmd[0]),'y')!=NULL)
    {
        flashDataErase(FLASH_DATA_BOOTLINE, BOOT_LINE_SIZE);
        flashDataSet(FLASH_DATA_BOOTLINE,BOOT_LINE_ADRS, BOOT_LINE_SIZE);
    }
#endif
    break;
```

这样在用户修改完内存中的启动行之后，console 会打印出提示信息，指示用户输入 Y 或 N 选择是否将修改后的参数保存到 Flash。

除了启动行之外，还可以保存其他的用户参数到 Flash，例如用户程序中的全局变量 boardWorkEnv，或者其他涉及到硬件的配置参数。无论 Flash 中的实际存储方式如何，经过一次写和一次读，从 Flash 中获得的数据字序一定和写入 Flash 的数据的字序相同，因此

可以直接把结构体变量的指针转化成 char 型指针，按照字符串的结构存储和读取结构体。首先在 board.h 中定义存放参数的 block。

```
#define FLASH_DATA_USRPARAM      32
```

然后在用户程序中使用下面的语句存储和获取结构体：

```
flashDataSet(FLASH_DATA_USRPARAM,((char*)(&boardWorkEnv.boardInit)),
            sizeof(struct BoardWorkEnv);   /*存储结构体*/
flashDataGet(FLASH_DATA_USRPARM, ((char*)(&boardWorkEnv.boardInit)),
            sizeof(struct BoardWorkEnv);   /*获取结构体*/
```

需要注意的是 boardWorkEnv.boardInit 是一个随着硬件初始化而改变的标志性参数，在获取结构体时不应该覆盖它的值，可以采用一个局部变量先保存它的值，在获取结构体后在用这个局部变量恢复。

7.5 总结

本章结合 Flash 芯片 intel28F320C3 介绍了 Flash 的软件操作方法，给出了对 intel28F320C3 进行基本操作的例程。结合此例程，通过添加用户程序的一个命令通道，并修改 Bootrom，实现从 Flash 加载 VxWorks 映像，以及 flash 中 VxWorks 映像的在线更新。接下来介绍了如何将用户程序和操作系统结合，实现脱离 Tornado 独立运行的自启动操作系统。最后介绍了如何将启动行等用户经常改动的参数存放到 Flash，实现系统的下电记忆功能。

第 8 章　人机界面——控制端软件设计

在结束了采集板软件的设计后，所有控制工作将交由人机界面的软件完成。本章将简略介绍在 Microsoft Visual C++环境下制作 Windows 应用程序实现人机界面的方法。

8.1　区分 VC6.0 与 VxWorks5.4 的编程方式

VC 下的编程方式和前面已经介绍的 VxWorks 下的编程方式有较大不同。

第一，VC 下使用的面向对象的编程，把相关的同一类函数和变量都集中到一个类中，通常尽量避免一个类直接调用另一个类的变量，除了 public 类型的函数和变量，调用时都会受到一定的限制。同时，除了类的实例申明，最好不要再有别的全局变量。而 VxWorks 下如果不选择 C++支持（选择 C++支持会增加 VxWorks 的长度），则不能使用类，所有的全局变量和函数都可以自由的互相调用。关于 VC 和类概念的资料在书店和网上都很多，这里就不详细介绍了。

第二，需要注意的是 Windows 下的内存映射方式和 VxWorks 下的不同。VxWorks5.4 采用的是最简单的映射方式，无论是局部的还是全局的数组，都不可能出现相邻元素地址不连续的情况。即如果有数组 char buff[4096]，则（&buff[i+1]-&buff[i]）恒等于 1，其中 i 为从 0～4094 间的任意数。

而在 Windows 下类的数组，可能出现地址不连续的现象，例如假设类

```
class CusrClass: public CAsyncSocket
{   /*类定义*/
   public:
   CusrClass ();
   virtual ~ CusrClass ();
   public:
       char buff[4096];
}
CusrClass usrClass;     /*类生成的实例*/
```

则（&usrClass.buff[i+1]-&usrClass.buff[i]）不一定等于 1，具体情况视程序编译和运行而不同。使用 malloc()申请内存可以避免这种情况。例如：

```
class CusrClass: public CAsyncSocket
{   /*类定义*/
   public:
       CusrClass ();
```

```
        virtual ~ CusrClass ();
    public:
        char *buff;
}
CusrClass:: CusrClass ()        /*构造函数*/
{
    buff = malloc(4096);
}
CusrClass:: ~ CusrClass ()      /*析构函数*/
{
    free(buff);
}
CusrClass usrClass;             /*类生成的实例*/
```

如果 malloc 申请成功，则（&usrClass.buff[i+1]-&usrClass.buff[i]）恒等于 1。

第三，VC 下较容易实现的是控件的回调函数，响应的是用户对界面的点击、键盘事件等；而 VxWorks 则是多任务系统，响应的是网络命令和硬件中断。如果在 VC 下要同时进行多种操作，可以使用多线程。

第四，VC 的命名规则不同于 VxWorks，最主要的区别就是 VC 下的变量和函数基本上是以大写字母开头的，而 VxWorks 下则默认用小写字母开头。

VxWorks 用户程序编写人员应该能够认识到不同操作系统下程序工作原理的不同，并在编程时加以注意。

8.2 以太网网络

为了建立人机界面和采集板的联系，使用基于 TCP/IP 的网络。在第三章中已经介绍了 VxWorks 下的网络程序。Windows 下的网络程序可以参看光盘中例 3-1、例 4-1 和例 9-1 中的 VC 程序。它们使用的都是 VC 提供的异步套接字类：CAsyncSocket。VC 还提供了类似 VxWorks 下 socket 操作的另一套库函数，它们各有优劣，下面将分别介绍。由于 VC 提供的函数众多，很多操作都包含一套向前兼容纯 C 编程的函数和另一套用类方法集合到一起的函数，它们的详细用法在 Visual Studio6.0 的帮助 MSDN Library 中都可以查到（在 VC 下按 F1 快捷键调用 MSDN），除了特殊需要注意的函数，下文中一般只给出函数名和简略介绍。

8.2.1 类似于 VxWorks 下的 socket 库函数

Windows 提供一套伯克利架构（Berkeley-style）的套接字编程接口，使用方法和 VxWorks 下的套接字类似，它们的使用方法可以参看第三章中关于网络编程的相关描述，以及 VC6.0 的 MSDN 帮助。在调用 socket 创建套接字之前，必须要先使用 WSAStartup()初始化 Windows socket 库，并在程序退出时调用对应的 WSACleanup()。通过在 VC 下设置 Project →Settings→Link，将 Object/library modules 对话框下填入“Ws2_32.lib Kernel32.lib”

来提供编译和运行时需要使用的库。此外，还需要在.h 文件中用#include 包含 Winsock2.h 文件。

- 函数：int WSAStartup（WORD wVersionRequested, LPWSADATA lpWSAData）

描述：初始化 Windows socket 动态链接库 Ws2_32.dll 的使用。

参数：

wVersionRequested, 用户程序可以使用的 Windows Socket 最高版本，高 8bit 表示版本号小数点前的部分，低 8bit 表示小数点后的部分。

LpWSAData, 指向一个 WDADATA 结构体，包含 Windows Socket 具体应用的参数。

返回：如果成功，返回 0。否则返回下面宏中的一个：

WSASYSNOTREADY, 底层网络系统还没有初始化好。

WSAVERNOTSUPPORTED, 现有的 Windows Sockets 不支持要求的版本。

WSAEINPROGRESS, 存在被阻塞的 Windows Socket 1.1 操作。

WSAEPROCLIM, 达到 Windows Sockets 支持的操作个数极限。

WSAEFAULT, lpWSAData 是非法指针。

- 函数：int WSACleanup();

描述：结束动态链接库 Ws2_32.dll 的使用。

参数：无

返回：如果成功，返回 0。否则返回 SOCKET_ERROR，并可以用 WSAGetLastError() 获得具体的错误值。如果退出时存在被阻塞的操作，则必须先调用 WSACancelBlockingCall() 再调用此函数。

初始化部分程序可以参照下面部分：

```
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD( 2, 2 );
err = WSAStartup( wVersionRequested, &wsaData );
if (err != 0)
{
    ;/*不支持 socket 网络通信*/
}
else if( LOBYTE( wsaData.wVersion ) != 2 ||HIBYTE( wsaData.wVersion ) != 2 )
{
    /*不支持版本 2.0*/
    WSACleanup( );
}
else
{
    /*库加载成功，调用 socket()等函数初始化网络通信*/
}
```

在编程时需要注意，套接字操作出错时返回的是 SOCKET_ERROR 而非 VxWorks 下的 ERROR。

这种套接字编程接口简单易用，尤其是对于有 VxWorks 下的网络程序编写经验的人员，可以很快上手。相对于下面介绍的 CAsyncSocket 类，用户使用 send()和 recv()的长度限制较小，在大数据量的情况下比较方便。它的缺点是需要初始化和结束库的使用。在多套接字时不方便用一个类生成多个实例的方法来实现，而最好是在一个类中包括多个套接字，此类的每个函数都使用一个参数来确定具体要操作的套接字序号。

如果需要实现类似于 VxWorks 下的多任务“同时”进行命令的接收和发送，则需要使用多线程。一般在界面的回调函数中发送命令，同时使用一个专门的命令接收解释线程处理所有的从网络接收到的信息。

8.2.2 CAsyncSocket 类

使用 CAsyncSocket 可以“同时”进行接收和发送，而不需要使用多线程，但发送和接收的过程中受到 TCP/IP 的缓冲大小的限制，使长帧的发送和接收变得较为复杂。因此 CAsyncSocket 适用于界面控制的小量数据发送和接收，每个控件的回掉函数中只进行 Kbyte 量级或更少的单次网络接收或发送。

为了使软件支持 CAsyncSocket，在 VC 建立工程的第二步，需要选中对话框“Windows Sockets”。在程序中申明一个继承 CAsyncSocket 的类，即可以使用所有这些异步套接字的操作函数，人机界面程序中需要使用的主要函数见表 8-1。

表 8-1 Windows CAsyncSocket 类的成员函数

基本操作	Create()	创建套接字
	Bind()	将套接字与本地地址绑定
	Listen()	开始侦听
	Connect()	连接远端的套接字
	Accept()	接收来自网络的连接请求并新建通信用套接字
	Send()	发送数据
	Recv()	接收数据
	Close()	删除套接字
	AsyncSelect()	控制异步套接字的响应特性
操作响应	OnConnect()	连接结束（成功或者失败）即被调用
	OnAccept()	接收到来自网络的连接请求即被调用
	OnReceive()	接收到数据即被调用
	OnSend()	使用 AsyncSelect()设置 FD_WRITE 时即被调用
	OnClose()	网络通信出错时即被调用

其中，基本操作直接使用程序调用，而操作响应是在符合某一条件时自动被调用。所有的 OnXXX()函数都是可以修改的，用户可以在其中添加自己的代码。在 MFC 下使用 ClassWizard 就可以向类中添加操作响应的函数。在 VxWorks 下，判断对套接字的操作结果是否成功依靠的是操作函数的返回值，而对于 CasyncSocket 类来说，通过操作对应的响应回调函数的入口参数来决定。以 OnConnect()举例，在用户程序调用 Connect()并执行结束后就会被调用，可以通过它的参数来判断连接是否成功，并根据情况对界面做一些控制。例 9-1 中的 Windows 下 client 连接 VxWorks 下 server，通过一个按钮的回调函数和 Connect()的响应函数来完成，如下，其中 cltSkt 是用 CusrSkt 在 CwinCltAllDlg 类中申明的一个实例。

```
void CWinCltAllDlg::OnconnectServer()
{
    CString buff;
    int serverport;
    BOOL flag;

    flag = cltSkt.Create(0,SOCK_STREAM,FD_CONNECT);
    if(flag==0)
    {
        MessageBox("无法创建客户端通信用套接字");
        return;
    }
    /*从对话框获得 IP 和端口号*/
    GetDlgItem(IDC_server_port)->GetWindowText(buff);
    serverport = atoi((LPCTSTR)buff);
    GetDlgItem(IDC_serverIP)->GetWindowText(buff);
    cltSkt.Connect(buff,serverport);
}

void CusrSkt::OnConnect(int nErrorCode)
{
    BOOL nodelayflag = TRUE;
    if(nErrorCode == 0)
    {
        SetSockOpt(TCP_NODELAY, &nodelayflag, sizeof (BOOL));
        AsyncSelect(FD_READ|FD_CLOSE);
        MessageBox(NULL," 连接成功","连接成功",MB_OK);
        /*设置用户界面相关按钮的使能和非使能*/
        EnableWindow(GetDlgItem(theApp.m_pMainWnd->m_hWnd,
                                ID_connectServer),FALSE);
        EnableWindow(GetDlgItem(theApp.m_pMainWnd->m_hWnd,
                                ID_cmdSendLong),TRUE);
        EnableWindow(GetDlgItem(theApp.m_pMainWnd->m_hWnd,
                                ID_cmdSendShort),TRUE);
```

```
            EnableWindow(GetDlgItem(theApp.m_pMainWnd->m_hWnd,
                                    ID_cmdSendVariDebug),TRUE);
            EnableWindow(GetDlgItem(theApp.m_pMainWnd->m_hWnd,
                                    ID_cmdSendFuncDebug),TRUE);
        }
        else
        {
            Close();
            MessageBox(NULL,"无法连接，请检查网络和 IP 与端口号的设置",
                       "onConnect",MB_OK);
            EnableWindow(GetDlgItem(theApp.m_pMainWnd->m_hWnd,
                                    ID_connectServer),TRUE);
        }
        CAsyncSocket::OnConnect(nErrorCode);
    }
```

除了 OnXXX()回调函数外，CAsyncSocket 的网络发送也和 VxWorks 下的有很大不同，其他函数的操作方法比较简单，可以参看例 3-1、例 4-1 和例 9-1 中的 VC 源程序。

- 函数：CAsyncSocket::AsyncSelect(long lEvent);

描述：如果执行正确，则返回非零值，否则返回 0。此函数选择哪个或者哪些回调函数可以被响应，并自动将套接字设为非阻塞型。

参数：lEvent, 使用或的方法将下列的一项或多项设置为有效。

FD_READ: 当执行从套接字的读操作时，自动回调 OnReceive()。

FD_WRITE: 通知套接字数据已经准备好，可以发送，自动回调 OnSend()。

FD_ACCEPT: 当接收到远端的连接请求，自动回调 OnAccept()。

FD_CONNECT: 当试图连接远端并被接收后，自动回调 OnConnect()。

FD_CLOSE: 当网络通信出错时，自动调用 OnClose()。

返回：如果成功，返回 0。否则返回 SOCKET_ERROR，并可以用 WSAGetLastError()

如果需要发送数据，必须使用 AsyncSelect()，遵循下面的步骤：

- 界面回调函数或其他要求发送数据，调用 AsyncSelect(FD_WRITE|FC_CLOSE)
- 响应函数 OnSend()自动被调用，调用操作函数 Send()发送数据
- 如果数据量太大，继续调用 AsyncSelect(FD_WRITE)使 OnSend()再次被自动调用，直到所有数据发送结束
- 设置套接字准备接受数据 AsyncSelect(FD_READ|FC_CLOSE)

对于异步套接字，无法在一个函数中通过连续调用 Send()来一次完成大量数据的发送，必须连续使用 OnSend()函数，则发送长度等变量必须成为发送函数之外的变量，不能像 VxWorks 下做成一个函数的局部变量。两种方式的流程对比见图 8-1 和图 8-2。

对比两图可以很明显地看出，CAsyncSocket 类在一次发送大量数据的时候，复杂度远远高于 VxWorks 下的套接字（前面介绍的那种方法和 VxWorks 下的操作类似）。但它的优势是可以自动检测网络的通断，并在网络出错时自动调用 OnClose()通知用户。选择哪种套接字编程，需要根据具体情况而定。

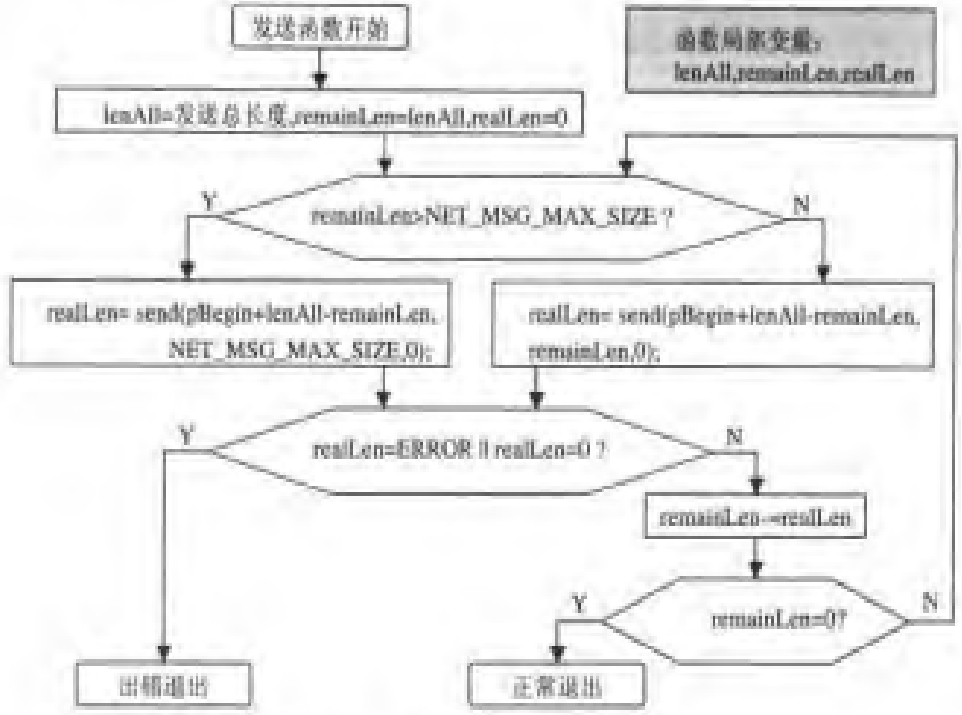


图 8-1　VxWorks 下的网络发送流程图

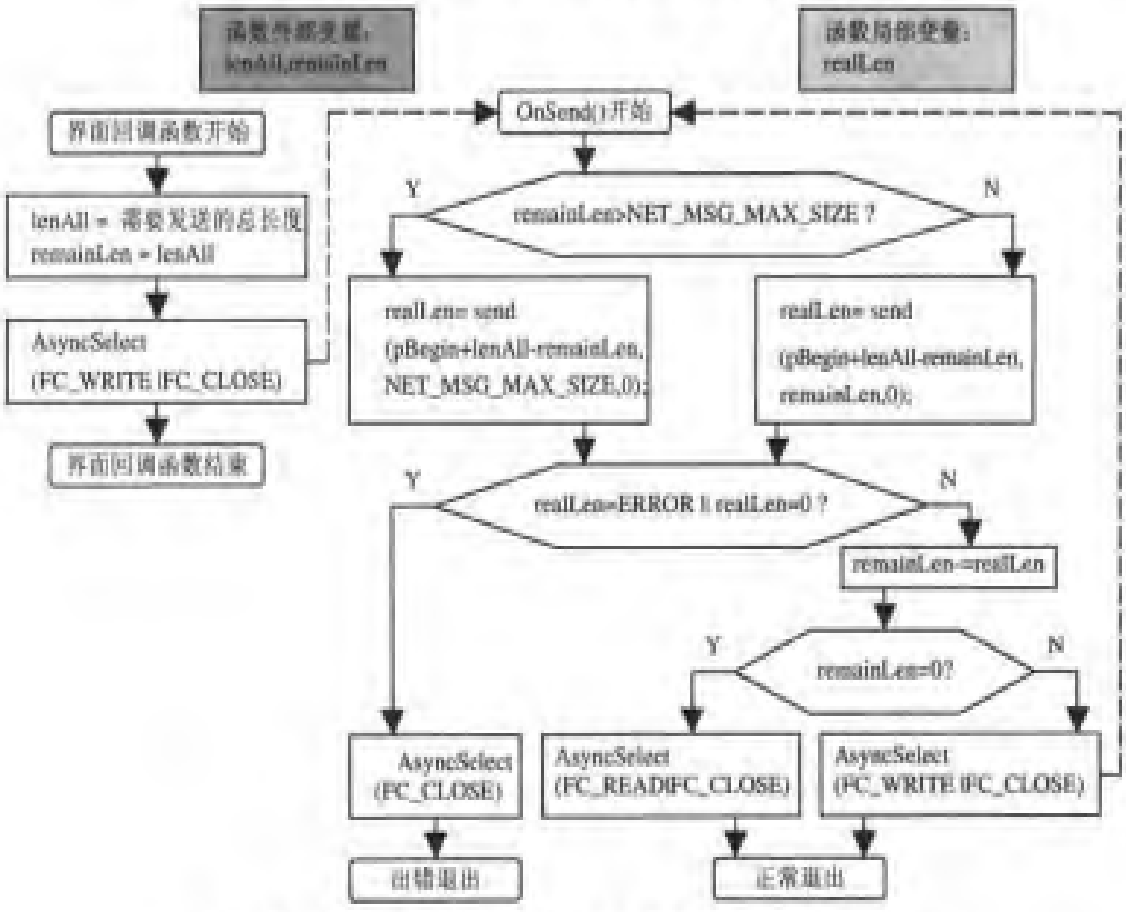


图 8-2　Windows 下 CAsyncSocket 类的网络发送流程图

8.3 参数控制

一般来说，可以通过两种方式改变程序运行时的参数：从文件获取或者动态输入。在这之前，需要规划一下需要使用的参数，可以建立一个结构体来统一管理。将对参数操作的函数和这个结构体放在一起，建立用户参数类。

从文件获取的方式比较简单，通过读取文件并将其值赋给参数即可。对于习惯用纯 C 编程的人，最基本的文件操作函数见表 8-2。

表 8-2 Stream I/O 函数（文件读写）

fopen()	打开流形式(stream)文件
fread()	从文件读取数据
fwrite()	向文件写入数据
fclose()	关闭文件

需要注意的是，如果文件采取二进制格式存储，即文件中含有 0x00 等特殊值，则打开文件时的 mode 参数必须包含'r'，否则认为 0x00 为文件结束符，文件读取在 0x00 处结束。

参数控制更多的是利用向 Dialog Box 上添加控件来实现。VC 下添加控件的界面见图 8-3。使用鼠标左键可以从右边的 Controls 框上将控件直接拖到 Dialog Box 上。Controls 框的显示控制在 Tools→Customize→Toolbars 下。

图 8-3 VC 下向 Dialog 上添加控件

每一个控件对应独立且唯一的 ID 号（静态文字 Static Text 除外），该 ID 使用一个宏定义将控件对应到一个数值，定义在 Resource.h 中。VC 的 Dialog Box 采用的所见即所得的策略，运行中的对话框大小和控件排列与这里编辑的完全一样。可以通过直接拖动来排列各控件。也可以改变控件的大小。选用图 8-4 左下角的 Dialog 条上的各按钮可以方便地调整控件的大小和对其排列方式，其显示控制同样在 Tools→Customize→Toolbars 下。在控

键上点击右键可以调出控键的属性，主要包括 ID 号的宏定义、控件上显示的字符、控件的初始属性（是否可见、是否使能等），选择 Styles 和 Extended Styles 还可以进行更多的初始值控制。

图 8-4 编辑控键的属性

在本书的例子中主要使用了三种控件：按钮（Button）、编辑框（Edit Box）和圆形选择钮（Radio Button）。涉及到的操作函数见表 8-3。具体的使用方式参见程序。

表 8-3 控件 Button、Edit Box 和 Raio 操作函数

GetDlgItem()	获得特定 Dialog Box 上的指定控件的句柄
EnableWindow()	设置控件使能/非使能（即响应/不响应鼠标和键盘）
IsWindowEnabled()	判断控件是否使能
GetWindowText()	将控件的内容拷贝到字符串（可用于 Edit Box）
SetWindowText()	将字符串设置到控件内容中（同上）
CheckDlgButton()	控制选中/非选中 Radio Button

可以选择将通过动态输入改变的全局参数存储到文件中，作为下次启动时的默认参数。一般在 Dialog Box 初始化时根据默认参数控制控件的选中和使能，并填写编辑框的内容。在 MFC ClassWizard 的 Messages 中选中 WM_INITDIALOG 建立回调函数 OnInitDialog()，并在此函数中添加相关控制，以保证对话框出现在用户面前时所有的可选择可编辑控件都有一个默认值。

可以使用专门的一个 Dialog Box 进行参数配置。在 VC Project 的 Resource 下选择添加新的 Dialog (Insert Dialog)，然后在此 Dialog 上添加控件，用来控制参数。建立对应的类 CXXXDlg，编写控件的回调函数，当需要弹出对话框时，使用下面的语句：

```
CXXXDlg xxxDlg;
xxxDlg.DoModal();
```

8.4 菜单、工具条和状态框

在 VC 新建 Project 时选择 MFC AppWizard（exe），并在 Step1 时选择 Single Document 或者 Multiple documents，则生成的 exe 文件包含菜单（Menu）和工具条（Toolbar），它们可以在程序运行时动态生成或改变，也可以直接在 VC 下使用所见即所得的方式编辑，见

图 8-5 和图 8-6。

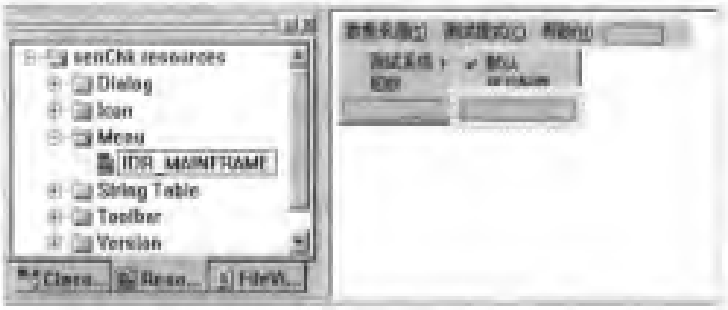


图 8-5 VC 下编辑菜单

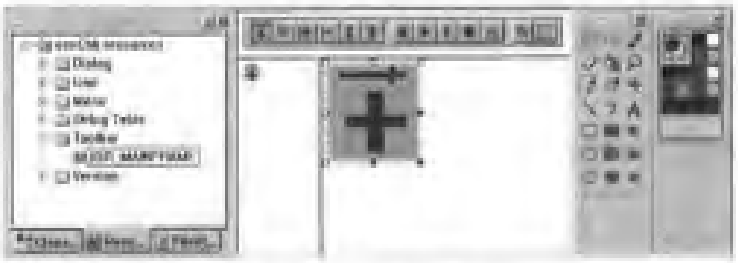


图 8-6 VC 下编辑工具条

对于菜单，可以设置为选中/非选中，使能/非使能。选中/非选中可以直接使用函数 CheckMenuItem()，参数 1 为菜单选项的 ID，参数 2 为选中（MF_CHECKED）/非选中（MF_UNCHECKED）：

```
CMenu *pMenu = GetMainWnd()->GetMenu();
pMenu->CheckMenuItem(ID_MENU_XXX, MF_UNCHECKED);
```

获取菜单选项是否被选中的函数为 GetMenuItemInfo()。

使能/非使能的设置需要在 MFC ClassWizard 的 Messages 中选中 UPDATE_COMMAND_UI，建立回调函数 OnUpdateXXX()，并使用全局变量作为控制参数：

```
void CMainFrame::OnUpdateXXX(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(flagXXX);
}
```

建议为所有需要设置使能/非使能的菜单和工具条选项建立一个结构体，以方便管理。

对于包含下级选单的菜单选项，没有对应的 ID 号，因此无法建立 UPDATE_COMMAND_UI 对应的回调函数，直接使用 EnableMenuItem()来设置菜单的使能/非使能。非使能后，将不能使用下级选单。

使用 SetPaneText()设置状态框的信息：

```
void CXXXApp::refreshStatusBar()
{
```

```
    CStatusBar *pStatusbar = &(((CMainFrame*)(m_pMainWnd))->m_wndStatusBar);
    pStatusbar->SetPaneText(0,"info of statusbar",TRUE);
}
```

设置状态框信息可以用来为程序的操作者提供实时帮助，对控件和菜单、工具条等的操作进行说明，并提供程序运行当前状态的相关信息。

建议为状态框信息建立一个字符数组进行缓存，以保存状态框信息。当进行某项操作时需要状态框提供帮助信息，而在操作结束后状态框恢复到操作前的显示，则需要将信息进行缓存。

可以将工具条的按钮设置为按下/非按下状态。默认的初始状态为非按下，使用SetCheck()改变按钮的状态，下面的代码实现按一下按钮则状态在按下/非按下状态间切换一次的功能，其中 flagEnableXXX 代表了按钮的使能状态，而 flagCheckXXX 则代表了按钮是否按下。

```
void CMainFrame::OnUpdateXXX (CCmdUI* pCmdUI)
{
    pCmdUI->Enable(flagEnableXXX);
    if(flagEnableXXX == TRUE)
    {
        if(flagCheckXXX == TRUE)
            pCmdUI->SetCheck(1);
        else
            pCmdUI->SetCheck(0);
    }
}
```

8.5 数据显示、存储和回放

对于数据采集系统的控制端软件，数据的显示和存储是必须的，同时还应当提供存储下来的数据的回放功能。图 8-7 是笔者为中国海洋石油服务公司开发的电缆测试系统的控制端软件界面，采用网络控制嵌入式采集板，接受采集板传输来的数据，并提供实时显示和数据分析。除了上面介绍过的菜单、工具条和状态框的使用之外，还涉及到数据的绘图和存储。

VC 下的 CScrollView 提供自带滚动条的绘图类，编程者需要在新建 Project 时的 Step 6 将 CXXXView 的 Base Class 选为 CScrollView，当需要绘图区域面积大于当前窗口的绘图区域，就会自动出现滚动条，供用户选择察看区域。使用 SetScrollSizes()设置绘图区域大小、滚动区域大小和每次滚动的长度。

当窗口需要重画时(例如用户拖动了窗口)，程序会自动调用 OnDraw()，因此在 OnDraw()函数内画图即可，程序的其他部分调用 InvalidateRect()也会使 OnDraw()被自动调用。

图 8-7　使用 VC 开发的一款控制端软件的界面

```
void CXXXApp::refreshClientRect()
{
    CRect clientRect;
    GetMainWnd()->GetClientRect(&clientRect);
    GetMainWnd()->InvalidateRect(clientRect,TRUE);
}
```

绘图函数参见 MSDN 的 CDC Class Members，其中常用的基本函数见表 8-4。

表 8-4　绘图常用函数

SelectStockObject	选刷子
Rectangle	画矩形
SelectObject	选画笔
MoveTo	将画笔移动至某处
LineTo	从画笔的当前位置连线到某处
SetBkColor	设置写字的背景色
SetTextColor	设置写字的颜色
TextOut	写字

需要注意的是，系统拥有自己的默认画笔，因此如果编程者需要使用自定义的画笔，必须首先保存原有的画笔，并在使用完之后进行恢复。下面的例子选择灰色的线宽为 3 的画笔，从点(0,0)画线到点(200,100)：

```
CPen usrPen (PS_DASHDOTDOT,3,RGB(192,192,192));
```

```
CPen *pOldPen = pDC->SelectObject(&usrPen);
pDC->MoveTo(0,0);
pDC->LineTo(200,100);
pDC->SelectObject(pOldPen);
```

应用这些函数就可完成简单的绘图任务，还可以通过添加回调函数的方式响应鼠标和键盘的事件。对于鼠标事件，通过对比鼠标点击点（鼠标事件回调函数参数 CPoint point）、滚动区域原点（GetDeviceScrollPosition()函数的返回值）和绘图区域原点（0,0），即可获得鼠标点击点对应的数据位置，从而显示具体的数据分析值。

为了优先保证数据的上传，通常将数据的接收和绘图存储分为两个线程，并开辟两片内存区域，分别供两个线程使用，线程之间的同步使用信号灯。基本流程如图 8-8 所示。

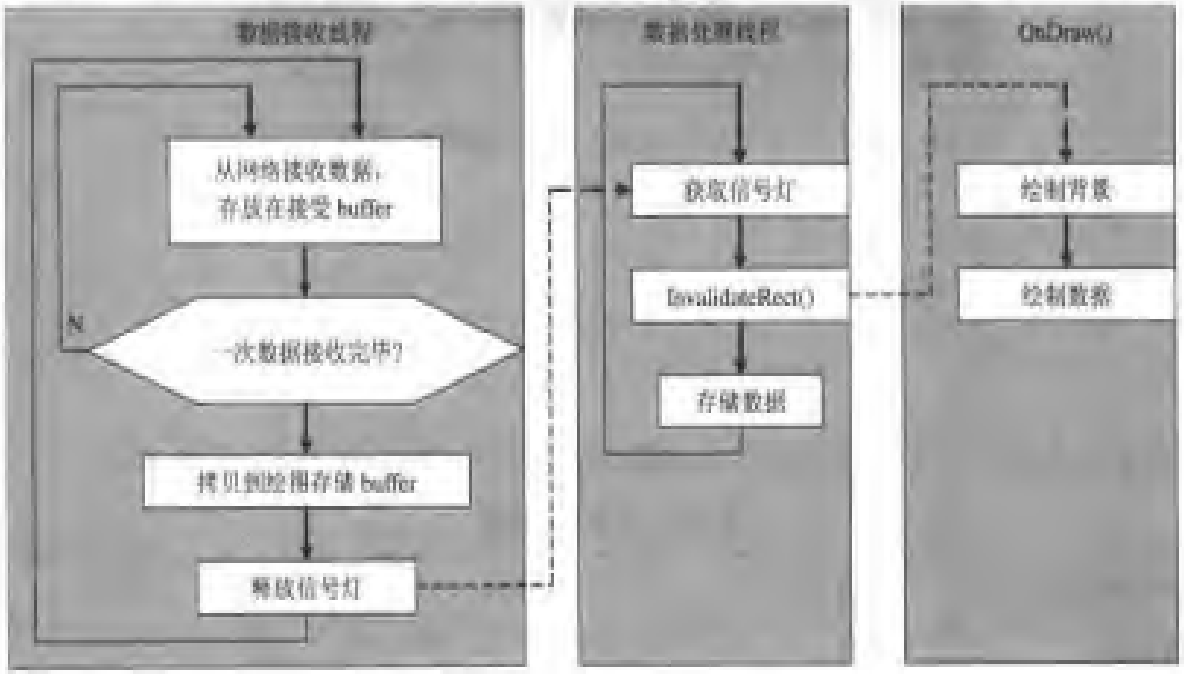


图 8-8　数据的接收和存储流程

为了使习惯于 VxWorks 下编程的人员可以尽快地上手使用线程和信号灯，这里介绍的线程操作和信号灯操作都是最基本的，使用方法尽量贴近 VxWorks 下编程习惯的，更多基于类方法的实现请参看 VC 的帮助。

线程必须依据某个函数来发起，这类似于 VxWorks 下的 taskSpawn，不同的是此函数的基本格式必须固定：

UINT threadName(LPVOID param);

当需要传递多个参数作为线程的参数时，可以使用结构体，并将 param 填写成指向结构体的指针。发起线程的函数为 AfxBeginThread()。

- 函数：CWinThread* AfxBeginThread
  (AFX_THREADPROC pfnThreadProc, LPVOID pParam,
  int nPriority = THREAD_PRIORITY_NORMAL, UINT nStackSize = 0,

DWORD dwCreateFlags = 0, LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL)

返回：新创建线程的指针。

参数：

pfnThreadProc, 用以发起线程的函数入口，不能为 NULL。

pParam, 函数的参数。

nPriority, 被发起线程的优先级。

nStackSize, 被发起线程的堆栈大小。如果填写为 0，则默认为发起此线程的线程堆栈大小。

dwCreateFlags, 控制发起 thread。可以是下面两个值中的一个：

CREATE_SUSPENDED，发起后挂起，挂起数为 1，直到在别处调用了 ResumeThread()才会运行。

0, 发起后立刻运行。

lpSecurityAttrs, 指向一个 SECURITY_ATTRIBUTES 结构体的指针，如果是 NULL，则使用发起此线程的线程的安全限制参数。

如果需要使线程退出，必须在线程内部调用 AfxEndThread()，或者在线程内部 return。不能像 VxWorks 下那样从一个任务调用 taskDelete 直接删除另一个任务，因此如果需要从一个函数命令由另一个函数作为入口的线程退出，两函数间必须有某种方式的通信通道。例如采用全局变量判断或者信号灯。

当线程中调用了会导致阻塞的函数[例如网络接收的 recv()函数]时，可能会因为阻塞在此条语句而使线程无法从内部退出。因此，在多线程共同运作时，需要注意线程退出的顺序，可以用全局变量来标志线程的运行位置。

设置线程优先级可以使用 SetThreadPriority()。

VC 下的信号灯操作函数见表 8-5。

表 8-5 信号灯操作函数

CreateSemaphore()	创建信号灯
CloseHandle()	删除信号灯
ReleaseSemaphore()	释放信号灯
WaitForSingleObject()	获取信号灯

- 函数：HANDLE CreateSemaphore（LPSECURITY_ATTRIBUTES lpSemaphore Attributes, LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName）

返回：如果成功则返回新创建信号灯的句柄。如果同名信号灯已经存在，则返回此信号灯的句柄。如果创建失败，则返回 NULL。

参数：

lpSemaphoreAttributes, 指向一个 SECURITY_ATTRIBUTES 结构体的指针，用来

确定信号灯的继承属性，如果 lpSemaphoreAttributes 为 NULL，则信号灯不能被继承。

lInitialCount, 信号灯初始计数，必须大于等于零且小于等于 lMaximumCount。

lMaximumCount, 信号灯最大计数，必须大于零。

lpName, 指向信号灯名称字符串的首地址指针，如果为 NULL，则新建的信号灯没有名称。

- 函数：BOOL CloseHandle (HANDLE hObject)

返回：如果成功则返回非零值，否则返回零。

参数：

hObject, 需要结束并释放的句柄，包括信号灯句柄。

- 函数： BOOL ReleaseSemaphore
  (HANDLE hSemaphore, LONG lReleaseCount, LPLONG lpPreviousCout)

返回：如果成功则返回非零值，否则返回零。

参数：

hSemaphore, 需要释放的信号灯句柄，即 CreateSemaphore 的返回值。

lReleaseCount, 信号灯计数的增加量，必须大于 0。如果指定的值导致信号灯计数超过创建信号灯时规定的最大计数，则信号灯计数不会改变，且函数返回值为 FALSE。

lpPreviousCount, 用来存储信号灯操作前计数的变量的指针，如果不需要获得操作前的信号灯计数，则可以设置为 NULL。

- 函数：DWORD WaitForSingleObject (HANDLE hHandle, DWORD dwMilliseconds)

返回：如果在超时前获取成功则返回 WAIT_OBJECT_0 ，否则如果超时则返回 WAIT_TIMEOUT。如果获取失败返回 WAIT_FAILED

参数：

hHandle, 需要获取的信号灯句柄。

dwMilliseconds, 等待时间，单位为 ms。函数返回的条件为此时间已经过去，或者成功地获得了信号灯。如果此参数为 0，则立即返回；如果此参数定义为 INFINITE，则永远不会因为超时而返回。

8.6 总结

本章简要介绍了一些在 VC 下编写程序的函数接口，用来编写采集板 VxWorks 的调试和控制软件，包括网络通信、Dialog Box 控件、目录、工具栏、状态框、绘图以及多线程创建和线程同步用信号灯的操作接口。由于 VC 编程不是本书的重点，因此大部分的函数说明都只是介绍性的，编程者可以参阅 MSDN，以获得更加详细的函数说明。为了保证 VxWorks 程序的正确和可控性，无论最终的控制软件是否是 Windows 程序，都应当建立 Windows 下的调试软件，并随着 VxWorks 用户程序的逐步开发而不断更新。

第 9 章　随心所欲——嵌入式函数和全局变量的远端调用

当嵌入式软件架构完毕、制成自启动系统之后，就可以完全采用控制端软件控制采集板了。在系统交付使用后，可能还会暴露潜在的问题。此时系统处于工作现场而非研发实验室，因此不可能期望控制端软件安装的机器上同时安装 VxWorks 开发环境 Tornado，所有对采集板软件的控制都必须通过控制软件完成。为此，在采集板的控制端软件中保留一个命令号作为调试通道，VxWorks 下的函数调用、变量显示、内存修改都通过此通道进行。开发人员可以通过 e-mail 指导操作人员进行错误定位。

9.1　整理用户程序的全局变量和函数接口

首先看一下 VxWorks 的开发调试环境 Tornado。Tornado 的组件 shell 具有 C 语言解释器，用户可以通过直接在 shell 下输入函数名的方法调用函数，也可以通过输入变量名的方法查看和改变全局变量的值。在 PC 的 shell 下输入操作方式，而具体操作由采集板上的嵌入式软件执行。为了达到这一点，PC 端首先需要运行 target server，以建立 PC 端和采集板软件的联系；其次，PC 端必须知道或者以某种方式通知 VxWorks 使用由 PC 端指定的函数和变量的指针。通过将 target server 的 corefile 设置成与采集板运行的 VxWorks 映像文件一致，PC 端获得了函数名和函数指针的对应关系。

如果要在脱离 target server 和 host based shell 并且没有 VxWorks corefile 的情况下，调用 VxWorks 下的函数和变量，就需要建立一套类似的系统。可以借助控制端和采集板的命令通道作为网络通信的途径。而函数和变量的指针则必须由用户程序提供。

首先考虑全局变量。无论是直接定义的全局变量、数组、结构体，还是只供库函数使用的 ID，对其的操作归根结底都可以看作是对内存的操作。VxWorks 是 32 位操作系统，支持的变量类型包括 1byte、2byte 和 4byte。

每个变量所包含的信息有：

- 变量指针：4byte
- 变量长度：1/2/4byte
- 变量说明：定义最大为 255byte

定义结构体描述这些内容：

```
struct TaskVariInfo
{
```

```
    unsigned short  variNum;            /*变量的分类序号*/
    UINT            variPointer;        /*变量的指针*/
    char            variLen;            /*变量长度*/
    char            variName[255];      /*变量名说明*/
};
```

对于数组，由于每个元素的长度特性等相同，且地址随着元素的序号线性递增，因此只给出数组中的第一个元素的信息，后续元素的信息和修改使用函数，参见关于函数接口的描述。而对于结构体，各元素的信息不尽相同，采取将结构体拆开，作为多个变量的形式。

再考虑函数，每个函数包含的信息有：

- 函数入口指针：4byte
- 参数：最多支持 10 个参数，长度可以为 1/2/4byte
- 函数返回长度：如果 void 型则长度为 0，否则长度为 1/2/4byte
- 函数说明：定义为最大 255byte

定义结构体描述这些内容：

```
struct TaskFuncInfo
{
 unsigned short     funcNum;            /*函数号*/
 FUNCPTR        funcPointer;            /*函数入口*/
 unsigned char  funcReturnSize;         /*函数返回值的长度*/
 char               funcName[255];      /*函数说明*/
};
```

用上面的两个结构体声明全局数组，存储采集板变量和函数信息。为了保证随时添加变量和函数的新条目，定义支持的变量和函数的最大条数，并用此条数作为数组最后一个结构体元素的 variNum 或者 funcNum，如下：

```
#define VARI_END_NUM   0xFFFF
struct TaskVariInfo  variInfo[] =
{
{0x0000,((UINT)&semFlash), sizeof(SEM_ID),"SEM_ID semFlash"},
{0x0100,((UINT)&(boardWorkEnv.boardInit)),sizeof(int),
                                    "boardWorkEnv.(int)boardInit"},
{0x0101, ((UINT)&(boardWorkEnv.autoBootFlag)),sizeof(char),
                                    "boardWorkEnv.(char)autoBootFlag"},
{0x0102, ((UINT)&(boardWorkEnv.dmaAutoFlag)),sizeof(char),
                                    "boardWorkEnv.(char)dmaAutoFlag"},
/*此处还可以继续添加其他全局变量参数*/
{VARI_END_NUM,0,0,"0"}
};
```

对于全局变量操作的所有查询方式都是基于序号的，如果查询到序号等于VARI_END_NUM，则表示结构体已经结束，查询序号无效。对于函数信息的处理也类似。除了可以将用户程序的函数列入结构体数组中，还可以添加需要使用的库函数，格式与用户函数完全一致，但必须注意要包括相应的.h文件。

```
#include "usrLib.h"
#define FUNC_END_NUM    0xFFFF
struct TaskFuncInfo funcInfo[]=
{
/*用户函数*/
{0x0300,(FUNCPTR)usrDefaultParamLoad,0,"void usrDefaultParamLoad()"},
{0x0301,(FUNCPTR)usrParamShow,0,"void usrParamShow()"},
/*库函数*/
{0xF100,(FUNCPTR)d,0,"void d(void*adrs, int nuints, int width)"},
{0xF200,(FUNCPTR)taskDelete,sizeof(STATUS),"STATUS taskDelete(int tid)"},
/*此处还可以添加其他函数的接口信息*/
{FUNC_END_NUM,NULL,0,"NULL"}
};
```

实际定义这些全局结构体数组时，应当尽量使结构体的各个元素对齐，保证程序可读性。

在整理用户程序中的全局变量和函数时应当注意下面几点：

- 在序号的编排上，应该尽量把一种类型的变量（函数）的序号编成一组，以方便管理和查阅。
- 如果将类似信号灯 ID 等由库函数操作的全局变量列入了全局变量结构体数组，则应当将操作它的基本库函数包括到函数结构体数组中，否则这种 ID 型全局变量因为无法操作而变得没有意义。
- 命令通道的用户函数可以列入函数结构体数组，但使用时一定要小心，因为使用不当可能造成网络通信通道的崩溃，导致整个用户程序出现致命错误。
- 对于结构体数组等较复杂变量，只将数组第一项的结构体各元素列入全局变量列表，可以根据结构体各元素的长度以及结构体的总长度，推算出数组第二项的结构体中各变量的指针。
- 保留一定量的测试函数，并适当加入库函数，以便加大程序的灵活性，方便定位错误。

9.2 控制全局变量和函数的通信协议

要使控制端能够远程操作这些列入了结构体数组的全局变量和函数，必须给它们定义命令号，制定适当的通信协议。使用命令号 0xFF00 和 0xFF01 分别作为变量和函数的调试通道。

9.2.1 全局变量的通信协议

对于变量来说，需要进行的操作分为读写两种。此外，为了提供参考信息，还需要帮助命令。用0xFF00命令的byte8表示命令种类，分为读、写和帮助。

帮助命令不需要其他任何参数，帧长度固定为9，采集板接收到帮助命令后，将全局变量结构体数组的内容按一定格式打印出来，同时还给出全局变量的值，见图9-1。

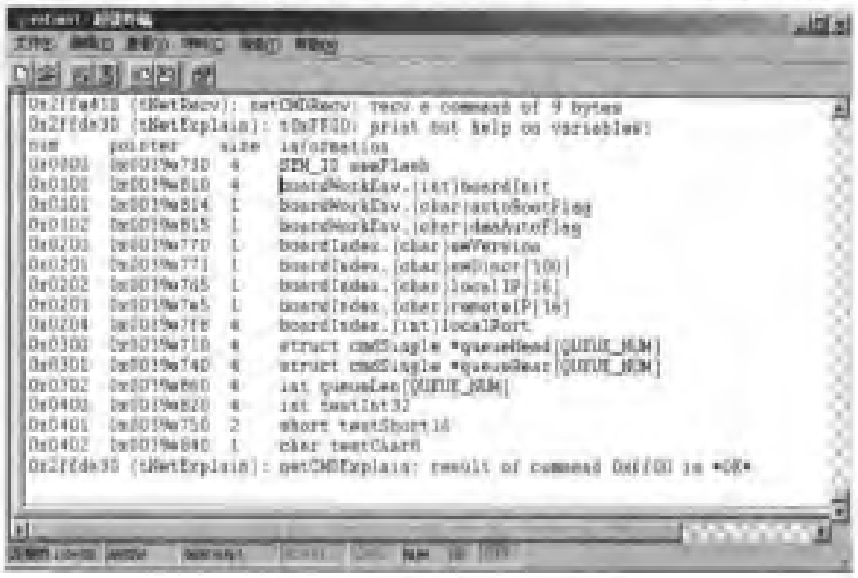


图9-1 远程控制的全局变量帮助信息打印

读、写命令都需要提供操作对象的序号，变量和序号的对应关系可以通过帮助命令查询。由于包括用户程序在内的所有嵌入式软件都是固化到采集板的，因此每次启动不会改变全局变量和变量序号之间的关系。使用命令的byte9和byte10表示变量序号，高8位在前，低8位在后。

读操作不需要更多的参数，但为了和写操作统一，给出1byte的数据长度，即从变量序号对应的首地址读取的字节数，故全局变量读命令长度固定为12byte。采集板接收命令后，根据变量序号从变量结构体数组中找到对应的元素，并利用该结构体中的变量指针和变量长度信息获取变量的值，并通过将变量值打印的方式给出执行结果。

写操作需要提供给全局变量赋的新值，且不同变量的长度不同，需要的值的长度也就不同。定义命令的byte11为变量长度，必须和帮助命令给出的对应参数的长度一致，否则采集板将会显示错误信息，而不会给变量赋值。byte12~byte15根据byte11而定，如果变量长度为1，则byte12有效；如果变量长度为2，则byte12和byte13有效，byte12为变量的高8位；如果变量长度为4，则byte12~byte15全部有效，byte12是变量的最高8位。因此写操作命令的长度为（12+变量长度）。

全局变量读写操作的实际执行见图9-2。

全局变量操作的命令格式见表9-1。

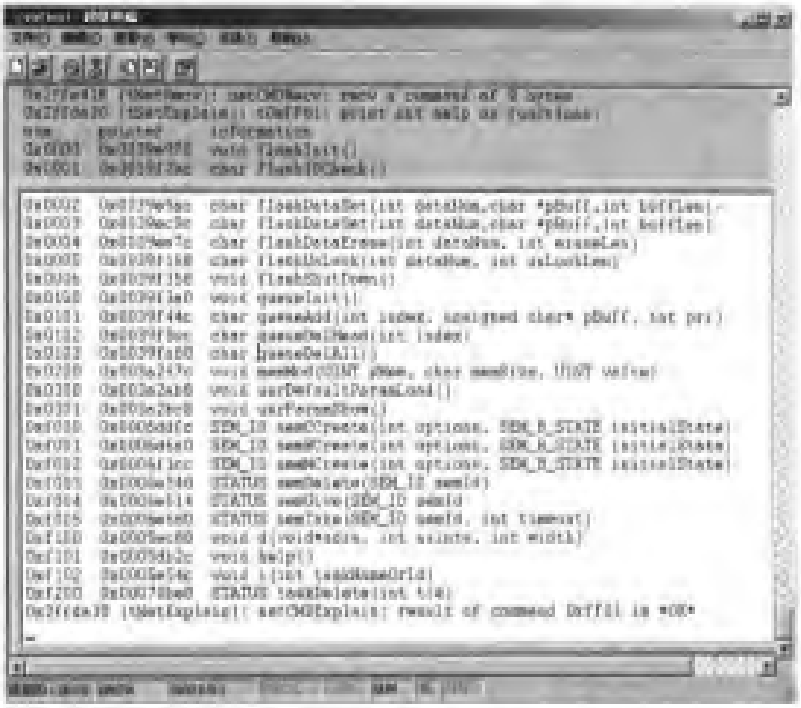


图 9-2 远程控制的全局变量读写

表 9-1 全局变量操作的命令格式

	帮助	读	写(char)	写(short)	写(int)
byte0	0xFF	0xFF	0xFF	0xFF	0xFF
byte1	0x00	0x00	0x00	0x00	0x00
byte2	0x00	0x00	0x00	0x00	0x00
byte3	0x00	0x00	0x00	0x00	0x00
byte4	0x00	0x00	0x00	0x00	0x00
byte5	0x09	0x0C	0x0D	0x0E	0x10
byte6	0x00	0x00	0x00	0x00	0x00
byte7	0x00	0x00	0x00	0x00	0x00
byte8	0xFF	0x00	0x01	0x01	0x01
byte9	无	变量序号 高 8 位	变量序号 高 8 位	变量序号 高 8 位	变量序号 高 8 位
byte10	无	变量序号 低 8 位	变量序号 低 8 位	变量序号 低 8 位	变量序号 低 8 位
byte11	无	1/2/4	1	2	4

续表

	帮助	读	写(char)	写(short)	写(int)
byte12	无	无	变量值	变量值 高8位	变量值 最高8位
byte13	无	无	无	变量值 低8位	变量值 次高8位
byte14	无	无	无	无	变量值 次低8位
Byte15	无	无	无	无	变量值最低8位

9.2.2 函数的通信协议

对于函数来说，操作包括使用 taskSpawn 发起和直接调用，加上帮助总共三种操作方式，用 byte8 表示，分别为 0x00，0x01 和 0xFF。

帮助操作不需要任何其他参数，故命令长度固定为 9byte。采集板接收到帮助命令后，将函数结构体数组的内容按一定格式打印出来，见图 9-3。

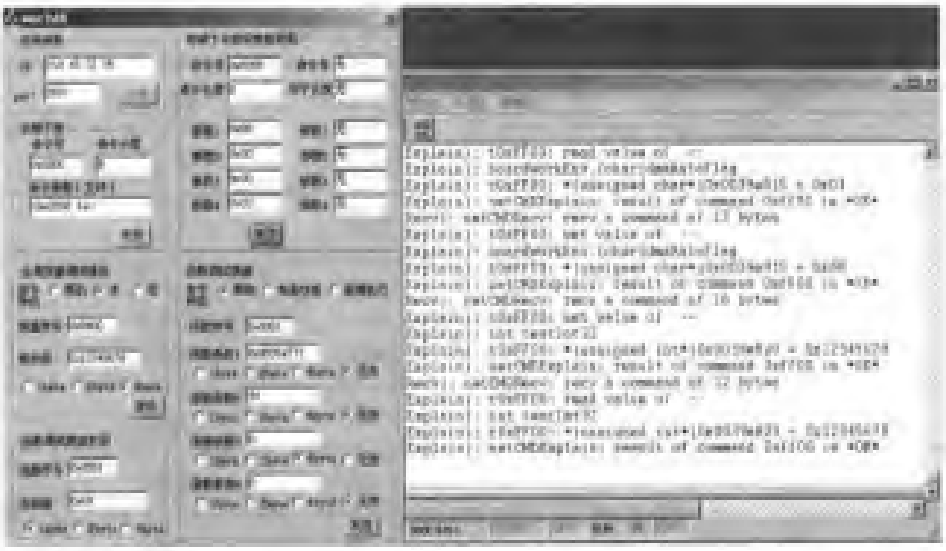


图 9-3 全程控制的函数帮助信息打印

无论是哪种方式调用函数，都需要给出函数的序号和对应的参数。函数序号用 byte9 和 byte10 表示，分别为 0xFF 和 0x01。函数参数使用类似于全局变量命令的格式给出。不同的是，函数可能会有一个以上的长度各不相同的参数，且不同函数的参数数量不同，因此命令 0xFF01 的长度无法统一确定。从 byte11 开始，每个参数都使用 1byte 参数长度+对应长度参数的格式，按顺序排列。参数的高位在前，低位在后。如果希望以直接执行的方式调用函数{0x0200,(FUNCPTR)memMod,0, "void memMod(UINT pMem, char memSize, UINT value)"}，参数分别为 0x12345678、1 和 0x00000011，则下传命令各字节按顺序为：

（命令号，命令长度）　0xFF,0x01,0x00,0x00,0x00,0x17,0x00,0x00,
（函数执行方式，序号）　0x01,0x02,0x00,
（函数参数）　0x04,0x12,0x34,0x56,0x78,0x01,0x01,0x04,0x00,0x00,0x00,0x11

最多支持的函数参数为 10 个，这也是 taskSpawn 发起任务的入口函数可以携带参数的上限。对于绝大多数的函数来说，10 个参数完全可以满足要求。

如果函数采用直接调用的方式，还可以得到函数的返回值，且返回值分为 1/2/4byte 和没有返回(void 型函数)4 种情况。采用下面的格式将返回值回传给控制端：前 8byte 使用通用通信协议的格式，0xFF01 作为命令号。byte8 和 byte9 为函数序号，byte10 为返回长度，如果是 void 型函数此 byte 赋为 0，否则对应为 1/2/4，byte11~byte14 根据 byte10 而定，如果变量长度为 1，则 byte11 有效；如果变量长度为 2，则 byte11 和 byte12 有效，byte11 为变量的高 8 位；如果变量长度为 4，则 byte11~byte14 全部有效，byte11 是变量的最高 8 位。

图 9-4 给出了远程控制函数执行的运行情况。

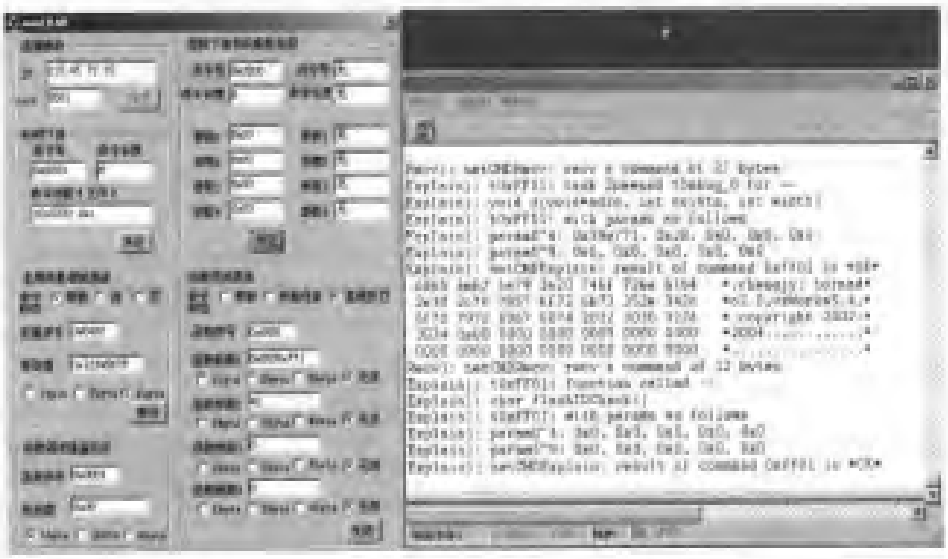


图 9-4　远程控制的函数的直接执行和任务发起

9.3　控制端调用和受控端响应实例

例 9-1 给出了 VxWorks 受控端对全局变量和函数操作的相应程序。如果用户程序需要支持更多的变量或者函数，则需要修改 netVariFunc.h 和 netTask.c，使用 extern 在前者中添加外部变量和外部函数的说明，如果是系统的库函数还需要使用#include 添加库函数需要包括的.h。然后在 netTask.c 开始处的对应结构体数组中添加一项说明。

随书光盘中还给出了 VC6.0 下的控制端软件。操作方法参见上面的图 9-1~图 9-4。

由于这两个命令通道的执行涉及到用户程序运行的底层，且通过函数 memMod 的调用可以将任意地址的内容修改成任意值，因此它具有强大的调试能力，但如果使用不当也会造成破坏性后果。因此调试通道只应当由系统设计者在紧急情况下使用，或者在设计者无法现场操作的情况下由设计者指导用户使用。在实际提供给使用者的采集板软件中，此调

试通道的调用应当使用密码保护，或者是使用某种方式的快捷键才能调出。

除了例 9-1 中提到的程序外，将命令号 0xFF00 和 0xFF01 加入用户程序还需要修改主循环程序 netSvr.c，在文件开头添加两个命令号对应函数的外部声明：

```
extern char t0xFF00(unsigned char *pBuff);      /*全局变量调试通道*/
extern char t0xFF01(unsigned char *pBuff);      /*函数调试通道*/
```

然后在命令执行任务对应的函数的 switch 语句中添加：

```
case 0xFF00:
    state = t0xFF00(pBegin);
    break;
    case 0xFF01:
    tate = t0xFF01(pBegin);
    break;
```

以保证其对这两个命令的对应。光盘中给出了包括 netSvr.c 在内的响应全局变量和函数操作的全部用户程序。

例 9-1　全局变量和函数远程操作例程[netVariFunc.h、netTask.c（部分）和 test.c]

```
/*************************************************************************
 chengjy@felab, copyright 2002-2004
 netVariFunc.h
 采集板参数和函数的网络控制需要的参数定义
 *************************************************************************/
#ifndef _NETVARIFUNC_H
#define _NETVARIFUNC_H

#include "board.h"

/*变量信息结构体*/
struct TaskVariInfo
{
    unsigned short  variNum;            /*变量的分类序号*/
    UINT            variPointer;        /*变量的指针*/
    char            variLen;            /*变量长度*/
    char            variName[255];      /*变量名说明*/
};

/*函数信息结构体*/
struct TaskFuncInfo
{
    unsigned short      funcNum;        /*函数号*/
    FUNCPTR         funcPointer;        /*函数入口*/
    unsigned char   funcReturnSize;     /*函数返回值的长度*/
    char                funcName[255];  /*函数说明*/
};
```

```
/*最多支持的函数个数，同时也作为结束符*/
#define VARI_END_NUM    ((unsigned short)0xFFFF)

/*最多支持的函数个数，同时也作为结束符*/
#define FUNC_END_NUM    ((unsigned short)0xFFFF)

/*同一个函数最多可以发起的任务个数*/
#define FUNC_SPAWN_MAX          10

/*发起任务使用的基本参数*/
#define TNAME_DEBUG_FUNC        "tDebug"
#define TPRI_DEBUG_FUNC         130
#define STACKSIZE_DEBUG_FUNC    2000

/*外部的全局变量*/
extern SEM_ID semFlash;
extern struct BoardIndex    boardIndex;
extern struct BoardWorkEnv  boardWorkEnv;
extern struct cmdSingle *queueHead[QUEUE_NUM],*queueRear[QUEUE_NUM];
extern int queueLen[QUEUE_NUM];

/*外部的函数*/
extern void flashInit();
extern char flashIDCheck();
extern char flashDataSet(int dataNum,char *pBuff,int buffLen);
extern char flashDataGet(int dataNum,char *pBuff,int buffLen);
extern char flashDataErase(int dataNum, int eraseLen);
extern char flashUnLock(int dataNum, int unLockLen);
extern void flashShutDown();
extern void queueInit();
extern char queueAdd(int index, unsigned char* pBuff, int pri);
extern char queueDelHead(int index);
extern char queueDelAll();
extern void memMod(UINT pMem, char memSize, UINT value);
extern void usrDefaultParamLoad();
extern void usrParamShow();

/*全局变量命令类型*/
#define CMD_VARI_READ       0x00
#define CMD_VARI_WRITE      0x01
#define CMD_VARI_HELP       0xFF

/*函数命令类型*/
#define CMD_FUNC_SPAWN      0x00
#define CMD_FUNC_DO         0x01
#define CMD_FUNC_HELP       0xFF
```

```
#endif  /*_NETVARIFUNC_H*/

/*************************************************************************
 chengjy@felab, copyright 2002-2004
 netTask.c
 这里只列出了和调试通道相关的部分，完成的程序请参见光盘。
 *************************************************************************/
#include "vxWorks.h"
#include "logLib.h"
#include "taskLib.h"
#include "stdio.h"
#include "memLib.h"
#include "stdLib.h"
#include "sysLib.h"
#include "bootLib.h"
#include "usrLib.h"

#include "board.h"
#include "netVariFunc.h"

int testInt32;
short testShort16;
char testChar8;

/*全局变量相关信息*/
struct TaskVariInfo  variInfo[] =
{
{0x0000,((UINT)&semFlash), sizeof(SEM_ID),"SEM_ID semFlash"},
{0x0100,((UINT)&(boardWorkEnv.boardInit)),sizeof(int),
                                "boardWorkEnv.(int)boardInit"},
{0x0101,((UINT)&(boardWorkEnv.autoBootFlag)),sizeof(char),
                                "boardWorkEnv.(char)autoBootFlag"},
{0x0102,((UINT)&(boardWorkEnv.dmaAutoFlag)),sizeof(char),
                                "boardWorkEnv.(char)dmaAutoFlag"},
{0x0200,((UINT)&(boardIndex.swVersion)),sizeof(char),
                                "boardIndex.(char)swVersion"},
{0x0201,((UINT)&(boardIndex.swDiscri)),
 sizeof(char),"boardIndex.(char)swDiscr[100]"},
{0x0202,((UINT)&(boardIndex.localIP[0])),
 sizeof(char),"boardIndex.(char)localIP[16]"},
{0x0203,((UINT)&(boardIndex.remoteIP[0])),sizeof(char),
 "boardIndex.(char)remoteIP[16]"},
{0x0204,((UINT)&(boardIndex.localPort)),
 sizeof(int),"boardIndex.(int)localPort"},
{0x0300,((UINT)&(queueHead[0])),sizeof(struct cmdSingle *),
```

```
                                    "struct cmdSingle *queueHead[QUEUE_NUM]"},
    {0x0301,((UINT)&(queueRear[0])),sizeof(struct cmdSingle *),
                                    "struct cmdSingle *queueRear[QUEUE_NUM]"},
    {0x0302,((UINT)&(queueLen[0])),sizeof(int),"int queueLen[QUEUE_NUM]"},
    {0x0400,((UINT)&testInt32),     sizeof(int),"int testInt32"},
    {0x0401,((UINT)&testShort16),sizeof(short),"short testShort16"},
    {0x0402,((UINT)&testChar8),     sizeof(char),"char testChar8"},
    /*此处还可以继续填加其他全局变量参数*/
    {VARI_END_NUM,0,0,"0"}
    };

    /*函数接口相关信息*/
    struct TaskFuncInfo funcInfo[]=
    {
    /*用户函数*/
    {0x0000,(FUNCPTR)flashInit,0,"void flashInit()"},
    {0x0001,(FUNCPTR)flashIDCheck,sizeof(char),"char flashIDCheck()"},
    {0x0002,(FUNCPTR)flashDataSet,sizeof(char),
                    "char flashDataSet(int dataNum,char *pBuff,int buffLen)"},
    {0x0003,(FUNCPTR)flashDataGet,sizeof(char),
                "char flashDataGet(int dataNum,char *pBuff,int buffLen)"},
    {0x0004,(FUNCPTR)flashDataErase,sizeof(char),
                    "char flashDataErase(int dataNum, int eraseLen)"},
    {0x0005,(FUNCPTR)flashUnLock,sizeof(char),
                    "char flashUnLock(int dataNum, int unLockLen)"},
    {0x0006,(FUNCPTR)flashShutDown,0,"void flashShutDown()"},
    {0x0100,(FUNCPTR)queueInit,0,   "void queueInit()"},
    {0x0101,(FUNCPTR)queueAdd,sizeof(char),
                    "char queueAdd(int index, unsigned char* pBuff, int pri)"},
    {0x0102,(FUNCPTR)queueDelHead,sizeof(char),
                    "char queueDelHead(int index)"},
    {0x0103,(FUNCPTR)queueDelAll,sizeof(char),"char queueDelAll()"},
    {0x0200,(FUNCPTR)memMod,0, "void memMod(UINT pMem, char memSize, UINT
value)"},
    {0x0300,(FUNCPTR)usrDefaultParamLoad,0,"void usrDefaultParamLoad()"},
    {0x0301,(FUNCPTR)usrParamShow,0,    "void usrParamShow()"},/*库函数*/
    {0xF000,(FUNCPTR)semCCreate,sizeof(SEM_ID),
                    "SEM_ID semCCreate(int options, SEM_B_STATE initialState)"},
    {0xF001,(FUNCPTR)semBCreate,sizeof(SEM_ID),
                    "SEM_ID semBCreate(int options, SEM_B_STATE initialState)"},
    {0xF002,(FUNCPTR)semMCreate,sizeof(SEM_ID),
                    "SEM_ID semMCreate(int options, SEM_B_STATE initialState)"},
    {0xF003,(FUNCPTR)semDelete,sizeof(STATUS),
                    "STATUS semDelete(SEM_ID semId)"},
    {0xF004,(FUNCPTR)semGive,sizeof(STATUS),"STATUS semGive(SEM_ID semId")},
    {0xF005,(FUNCPTR)semTake,sizeof(STATUS),
```

```
                "STATUS semTake(SEM_ID semId, int timeout)"},
{0xF100,(FUNCPTR)d,0,"void d(void*adrs, int nuints, int width)"},
{0xF101,(FUNCPTR)help,0,"void help()"},
{0xF102,(FUNCPTR)i,0,"void i(int taskNameOrId)"},
{0xF200,(FUNCPTR)taskDelete,sizeof(STATUS),"STATUS taskDelete(int tid)"},
/*此处还可以添加其他函数的接口信息*/
{FUNC_END_NUM,NULL,0,"NULL"}
};

char t0xFF00(unsigned char *pBuff);/*全局变量调试通道*/
char t0xFF01(unsigned char *pBuff);/*函数调试通道*/

/*************************************************************************
char t0xFF00(unsigned char*pBuff)
函数描述:    对VxWorks下的全局变量进行读写操作，具体格式参见通信协议。
网络返回:    无
调用:        无
*************************************************************************/
char t0xFF00(unsigned char *pBuff)
{
    int i;
    unsigned short variNum;
    unsigned char *pVari;

    if(pBuff[8]==CMD_VARI_HELP)    /*帮助信息*/
    {
        /*打印关于全局变量的帮助信息*/
        logMsg("t0xFF00: print out help on variables:\n",0,0,0,0,0,0);
        printf("num    pointer    size  information\n");
        i=0;
        while(variInfo[i].variNum!= VARI_END_NUM)
        {
            printf("0x%04x  0x%08x %2d    %s\n",variInfo[i].variNum,
               variInfo[i].variPointer,variInfo[i].variLen,variInfo[i].
               variName);
            i++;
        }
    }
    else if(pBuff[8] == CMD_VARI_READ) /*读取变量*/
    {
        /*查找全局变量值*/
        i=0;
        variNum = pBuff[9]*0x100+pBuff[10];
        while(variInfo[i].variNum!=VARI_END_NUM)
        {
            if(variInfo[i].variNum==variNum)
```

```
            break;
        else
            i++;
    }
    if(variInfo[i].variNum!=VARI_END_NUM)
    {
        /*存在此序号对应的全局变量，根据长度打印内容*/
        logMsg("t0xFF00: read value of  -- \n",0,0,0,0,0,0);
        logMsg("%s\n",(int)(variInfo[i].variName),0,0,0,0,0);

        if(variInfo[i].variLen==1)
        {
            logMsg("t0xFF00: *(unsigned char*)0x%08x = 0x%02x\n",
                variInfo[i].variPointer,*((unsigned
                char*)(variInfo[i].variPointer)),0,0,0,0);
        }
        else if(variInfo[i].variLen ==2)
        {
            logMsg("t0xFF00: *(unsigned short*)0x%08x = 0x%04x\n",
            variInfo[i].variPointer, *((unsigned short*)(variInfo[i].
            variPointer)),0,0,0,0);
        }
        else if(variInfo[i].variLen==4)
        {
            logMsg("t0xFF00: *(unsigned int*)0x%08x = 0x%04x\n",
            variInfo[i].variPointer, *((unsigned int*)(variInfo[i].
            variPointer)),0,0,0,0);
        }
    }
    else
    {
        /*没有找到对应的全局变量*/
        logMsg("t0xFF00: no variable with Num as 0x%04x defined\n",
            variNum,0,0,0,0,0);
        logMsg("t0xFF00: please send command with 0xFF00 as command Num
            and 0x%02x as first parameter\n",CMD_VARI_HELP,0,0,0,0,0);
        return(STATUS_WARNING);
    }
    }
    else if(pBuff[8] == CMD_VARI_WRITE)
    {
        /*查找全局变量值*/
        i=0;
        variNum = pBuff[9]*0x100+pBuff[10];
        while(variInfo[i].variNum!=VARI_END_NUM)
        {
```

```
            if(variInfo[i].variNum==variNum)
                break;
            else
                i++;
        }
        if(variInfo[i].variNum!=VARI_END_NUM)  /*修改变量*/
        {
            /*存在此变量序号对应的变量，根据其长度修改内存值*/
            logMsg("t0xFF00: set value of  -- \n",0,0,0,0,0,0);
            logMsg("%s\n",(int)(variInfo[i].variName),0,0,0,0,0);
            pVari = (unsigned char*)(variInfo[i].variPointer);
            if((variInfo[i].variLen==1)&&(pBuff[11]==1))
            {
                pVari[0] = pBuff[12];
                logMsg("t0xFF00: *(unsigned char*)0x%08x = 0x%02x\n",
                variInfo[i].variPointer, *((unsigned char*)(variInfo[i].
                variPointer)),0,0,0,0);
            }
            else if((variInfo[i].variLen ==2)&&(pBuff[11]==2))
            {
                pVari[0] = pBuff[12];
                pVari[1] = pBuff[13];
                logMsg("t0xFF00: *(unsigned short*)0x%08x = 0x%04x\n",
                variInfo[i].variPointer, *((unsigned short*)(variInfo[i].
                variPointer)),0,0,0,0);
            }
            else if((variInfo[i].variLen==4)&&(pBuff[11]==4))
            {
                pVari[0] = pBuff[12];
                pVari[1] = pBuff[13];
                pVari[2] = pBuff[14];
                pVari[3] = pBuff[15];
                logMsg("t0xFF00: *(unsigned int*)0x%08x = 0x%08x\n",
                variInfo[i].variPointer, *((unsigned int*)(variInfo[i].
                variPointer)),0,0,0,0);
            }
        else
        {
            logMsg("t0xFF00: write variable Num 0x%04x, variable length is %d,
              should be %d\n",variNum,variInfo[i].variLen,pBuff[11],0,0,0);
        }
    }
    else
    {
        /*没有找到对应的全局变量*/
        logMsg("t0xFF00: no variable with Num as 0x%04x defined\n",
```

```
                variNum,0,0,0,0,0);
        logMsg("t0xFF00: please send command with 0xFF00 as command Num
                and 0x%02x as first parameter\n",CMD_VARI_HELP,0,0,0,0,0);
        return(STATUS_WARNING);
    }
}
else
{
    /*全局变量的命令方式不正确*/
    logMsg("t0xFF00: no variable mode as 0x%02x\n",pBuff[8],0,0,0,0,0);
    logMsg("t0xFF00: please set first parameter of 0xFF00 as follows\n",
           0,0,0,0,0,0);
    logMsg("0x%02x for help, 0x%02x for read, 0x%02x for write\n",
           CMD_VARI_HELP,CMD_VARI_READ,CMD_VARI_WRITE,0,0,0);
        return(STATUS_WARNING);
    }
    return(STATUS_NORMAL);
}

/*************************************************************************
char t0xFF01(unsigned char*pBuff)
函数描述:    对VxWorks下的函数调用，具体格式参见通信协议。
网络返回:    执行完毕返回12byte 帧，具体格式参见通信协议。
调用:
    netTask.c:
           char netCMDAdd(unsigned char *pBuff, int buffLen, int cmdNum,
                          unsigned char priority)
*************************************************************************/
char t0xFF01(unsigned char *pBuff)
{
    unsigned short funcNum;
    int paramNum;
    UINT param32[10];
    int i,paramI,nameI;
    char funcSpawnName[30];
    unsigned char *pSendBuff;
    unsigned char      returnVal8;
    unsigned short returnVal16;
    unsigned int       returnVal32;

    if(pBuff[8]==CMD_FUNC_HELP)          /*帮助信息*/
    {
        /*打印关于函数的帮助信息*/
        logMsg("t0xFF01: print out help on functions:\n",0,0,0,0,0,0);
        printf("num    pointer    information\n");
        i=0;
```

```
        while(funcInfo[i].funcNum!=FUNC_END_NUM)
        {
            printf("0x%04x  0x%08x  %s\n",
                    funcInfo[i].funcNum,funcInfo[i].funcPointer,
                    funcInfo[i].funcName,0,0,0);
            i++;
        }
    }
    else if( (pBuff[8]==CMD_FUNC_DO) || (pBuff[8]==CMD_FUNC_SPAWN))
    {
        /*寻找函数 num 对应的函数入口*/
        funcNum = pBuff[9]*0x100+pBuff[10];
        i=0;
        while(funcInfo[i].funcNum!=FUNC_END_NUM)
        {
            if(funcInfo[i].funcNum==funcNum)
                break;
            else
                i++;
        }
        if(funcInfo[i].funcNum!=FUNC_END_NUM)
        {
            /*找到了对应的函数，逐个初始化函数使用的参数*/
            for(paramI=0;paramI<10;paramI++)   /*首先全部初始化为 0*/
            {
                param32[paramI]=0;
            }
            paramNum = 0;
            paramI = 11;
            while(paramNum<10)
            {
            if(pBuff[paramI]==1)   /*pBuff[paramI]对应当前参数的 byte 数*/
            {
                param32[paramNum] = (UINT)pBuff[paramI+1];
                paramNum++;
                paramI = paramI+2;
            }
            else if(pBuff[paramI]==2)
            {
                param32[paramNum] =
                        (UINT)(pBuff[paramI+1]*0x100+pBuff[paramI+2]);
                paramNum++;
                paramI = paramI+3;
            }
            else if(pBuff[paramI]==4)
            {
```

```
            param32[paramNum] = (UINT)(pBuff[paramI+1]*0x1000000+
                pBuff[paramI+2]*0x10000+pBuff[paramI+3]*0x100+pBuff[p
            aramI+4]);
            paramNum++;
            paramI = paramI+5;
        }
        else
            break;
    }
    /*根据命令方式决定是发起任务还是直接执行*/
    if(pBuff[8]==CMD_FUNC_SPAWN)
    {
        /*寻找不在使用的任务名*/
        nameI=0;
        while(nameI<FUNC_SPAWN_MAX)
        {
            sprintf(funcSpawnName,"%s0x%04x_%d",TNAME_DEBUG_FUNC,
                    funcInfo[i].funcNum,nameI);
            if(taskNameToId(funcSpawnName)==ERROR)
                break;
            else
                nameI++;
        }
        if(nameI==FUNC_SPAWN_MAX)  /*任务发起个数已达上限*/
        {
            logMsg("t0xFF01: the tasks spawned from funcNum 0x%04x has
        exceeded max--%d\n",funcInfo[i].funcNum,FUNC_SPAWN_MAX,0,0,0,0);
            return(STATUS_WARNING);
        }
        else
        {
            /*此序号还没有对应的任务，可以发起*/
            sprintf(funcSpawnName,"%s_%d",TNAME_DEBUG_FUNC,nameI);
            logMsg("t0xFF01: task Spawned %s for --\n",
                    ((int)funcSpawnName),0,0,0,0,0);
            logMsg("%s\n",((int)(funcInfo[i].funcName)),0,0,0,0,0);
            logMsg("t0xFF01: with params as follows\n",0,0,0,0,0,0);
            logMsg("param0~4: 0x%x, 0x%x, 0x%x, 0x%x, 0x%x\n",
            param32[0],param32[1],param32[2],param32[3],param32[4],0);
            logMsg("param5~9: 0x%x, 0x%x, 0x%x, 0x%x, 0x%x\n",
            param32[5],param32[6],param32[7],param32[8],param32[9],0);
            taskSpawn(funcSpawnName,TPRI_DEBUG_FUNC,0,
             STACKSIZE_DEBUG_FUNC,funcInfo[i].funcPointer,
            param32[0],param32[1],param32[2],param32[3],param32[4],
            param32[5],param32[6],param32[7],param32[8],param32[9]);
        }
```

```
}
else if(pBuff[8]==CMD_FUNC_DO)
{
    /*准备好网络返回*/
    pSendBuff = malloc((10+funcInfo[i].funcReturnSize)*sizeof(char));
    if(pSendBuff!=NULL)
    {
        pSendBuff[8]    = pBuff[9];
        pSendBuff[9]    = pBuff[10];
        pSendBuff[10] = funcInfo[i].funcReturnSize;
    }
    else
    {
        logMsg("t0xFF01: unable to malloc net return buff\n",
               0,0,0,0,0,0);
        return(STATUS_WARNING);
    }

    logMsg("t0xFF01: function called --\n", ((int)funcSpawnName),
           0,0,0,0,0);
    logMsg("%s\n",((int)(funcInfo[i].funcName)),0,0,0,0,0);
    logMsg("t0xFF01: with params as follows\n",0,0,0,0,0,0);
    logMsg("param0~4: 0x%x, 0x%x, 0x%x, 0x%x, 0x%x\n",
      param32[0],param32[1],param32[2],param32[3],param32[4],0);
    logMsg("param5~9: 0x%x, 0x%x, 0x%x, 0x%x, 0x%x\n",
      param32[5],param32[6],param32[7],param32[8],param32[9],0);

    /*根据提供的参数个数执行函数*/
    if(funcInfo[i].funcReturnSize == 0)
    {
        ((funcInfo[i].funcPointer)(param32[0],param32[1],param32[2],
                    param32[3],param32[4],param32[5],param32[6],
                    param32[7],param32[8],param32[9]));
    }
    else if(funcInfo[i].funcReturnSize == 1)
    {
        returnVal8 = ((funcInfo[i].funcPointer)
                   (param32[0],param32[1],
                    param32[2],param32[3],param32[4],param32[5],
                    param32[6],param32[7],param32[8],param32[9]));
        pSendBuff[11]   = returnVal8;
    }
    else if(funcInfo[i].funcReturnSize == 2)
    {
        returnVal16 = ((funcInfo[i].funcPointer)
                   (param32[0],param32[1],
```

```
                        param32[2],param32[3],param32[4],param32[5],
                        param32[6],param32[7],param32[8],param32[9]));
                        pSendBuff[11]   = ((returnVal16>>8)&0xFF);
                        pSendBuff[12]   = (returnVal16&0xFF);
                }
                else if(funcInfo[i].funcReturnSize == 4)
                {
                    returnVal32 = ((funcInfo[i].funcPointer)
                           (param32[0],param32[1],
                            param32[2],param32[3],param32[4],param32[5],
                            param32[6],param32[7],param32[8],param32[9]));

                    pSendBuff[11] = ((returnVal32>>24)&0xFF);
                    pSendBuff[12] = ((returnVal32>>16)&0xFF);
                    pSendBuff[13] = ((returnVal32>>8)&0xFF);
                    pSendBuff[14] = (returnVal32&0xFF);
                }

                /*网络返回*/
                if(pSendBuff!=NULL)
                {
                    netCMDAdd(pSendBuff,(10+funcInfo[i].funcReturnSize)
                           *sizeof(char),0xFF01,QUEUE_PRI_LOW);
                }
            }
        }
        else
        {
            /*没有找到对应的函数*/
            logMsg("t0xFF00: no function with Num as 0x%04x defined\n",
                funcNum,0,0,0,0,0);
            logMsg("t0xFF00: please send command with 0xFF01 as command Num
                and 0x%02x as first parameter\n",CMD_FUNC_HELP,0,0,0,0,0);
            return(STATUS_WARNING);
        }
    }
    else
    {
        /*函数的命令方式不正确*/
        logMsg("t0xFF01: no function mode as 0x%02x\n",pBuff[8],0,0,0,0,0);
        logMsg("t0xFF01: please set first parameter of 0xFF01 as follows\n",
                0,0,0,0,0,0);
        logMsg("0x%02x for help, 0x%02x for immediately do,
                0x%02x for task spawn\n",
                CMD_FUNC_HELP,CMD_FUNC_DO,CMD_FUNC_SPAWN,0,0,0);
        return(STATUS_WARNING);
    }
```

```
    return(STATUS_NORMAL);
}
/*************************************************************************
 chengjy@felab, copyright 2002-2004
 test.c
```

此文件保存用户用户测试的底层函数。一般来说，它们被用户在 shell 下调用，如果被 netVariFunc.h 声明并在调试通道中使用，则在将用户程序和 vxWorks 整合时，自动被优化不占用 vxWorks 映像文件的空间。

函数：

```
        void memMod(UINT pMem, char memSize, UINT value);
 *************************************************************************/
#include "vxWorks.h"

void memMod(UINT pMem, char memSize, UINT value);

/*************************************************************************
 void memMod(UINT pMem, char memSize, UINT value)
 函数说明:    直接修改内存
 参数:        pMem,       需要修改的内存地址。
              memSize,    需要修改的内存的长度，可取值 1,2,4。
              value,      修改内存的值，如果 memSize 取值 1，则使用最高 byte，否则对
                          应取最高 2byte 和全部的 byte。
 返回:        无
 *************************************************************************/
void memMod(UINT pMem, char memSize, UINT value)
{
 if(memSize==1)
    *((unsigned char*)pMem) = ((unsigned char)((value>>24)&0xFF));
 else if(memSize==2)
    *((unsigned short*)pMem) = ((unsigned short)((value>>16)&0xFFFF));
 else if(memSize==4)
    *((UINT*)pMem) = value;
}
```

9.4 总结

为了使系统交付使用后再出现问题时便于定位错误，本章提出了一种远程控制 VxWorks 下全局变量和函数的方法。该方法完全借助于控制端和受控端之间的网络命令通道，完全脱离 VxWorks 开发环境 Tornado，简单便捷地实现了自启动 VxWorks 的底层控制。首先介绍了将嵌入式用户程序的变量和函数分类的方法，然后给出了控制变量读写和函数执行的通信协议，最后给出了 VxWorks 下的 C 语言实现例程。结合光盘中给出的 VC 程序，即可实现嵌入式软件的远程控制。

第 10 章　让我们做得更好——针对 VxWorks 的算法优化

当阅读到这一章时，相信你已经能够较为完整地构建自己的嵌入式软件了。没错，凭借前九章的基础，所有需要实现的通用功能都能被实现了。但是，我们还需要百尺竿头更进一步，本章将为你介绍更好的网络监控和缓冲队列实现。与前面介绍的方法不同，它们的实现更加依赖于 VxWorks 的支持，因此如果你希望将它们移植到别的嵌入式操作系统，最好先搞清那个操作系统下的实现方式。

10.1　查获非法关机的网络监控程序

首先回忆一下第三章中介绍的网络监控程序，它主要是利用了网络发送 send()和接收 recv()的返回值，如果返回 ERROR，或者 recv()返回的实际接收长度为 0，则认为当前的套接字通道出错，从而通过释放信号灯的方式通知网络监控任务，以关闭当前套接字，并（只对 server 端）恢复到准备接受的状态。

这种检测使用于通信一方使用软件关闭套接字，或者软件方式强制退出程序（例如 Windows 下的任务管理器、Solaris 和 Dos 下的 Ctrl+C）。但如果遇到因为非法关机而破坏套接字的特殊情况，则通信的另一方的 recv()不会收到任何错误返回，且由于 send()不是随时被执行，因此无法判断出网络出错。

使用 ping()可以检测出对方的网口是否被初始化，但双方必须同时支持 ping 操作。Windows 操作系统默认支持 ping，直接在命令行下输入 ping [remoteIP]，就可以察看对方，前提是对方必须支持 ping 操作且没有将其屏蔽。

VxWorks 也支持 ping 操作，但必须要先初始化库。VxWorks 下关于 ping 操作的函数有 ping()和 pingLibInit()，需要包含的库文件为 pingLib.h。

- 函数：STATUS pingLibInit()

描述：初始化 ping()需要使用的资源。必须在调用 ping()函数前调用此函数。如果 BSP 的 config.h 中定义了宏 INCLUDE_PING，则此初始化函数被 BSP 自动调用。

参数：无

返回：OK，初始化失败则返回 ERROR。

- 函数：STATUS ping(char* host, int numPackets,ulong_t options)

描述：此函数通过发送 ICMP 包并等待回应来测试网络远端，可以程序调用，也可以从 shell 下调用如下：

```
->ping "210.45.72.143",1,0
```

可以通过修改下面的全局变量改变 ping()的参数。

```
_pingTxLen:          ICMP 包长，默认 64
_pingTxInterval:     每个包之间的时间间隔，默认 1S
_pingTxTmo:          收包的超时限制，默认 5S
```

参数：

char *host	对方的 IP 地址字符串首地址
int numPackets	该参数规定了从远端返回的 ICMP 包个数。如果 numPackets 为 1，则此任务值等待第一个回应包，然后就打印远端是否可以联系到的信息。如果 numPacket 不等于 1，则当接到回应包时还会打印时间和顺序信息。如果 numPacket 等于 0，则此函数一直运行。
ulong_t options	可以使用 0，还可以通过“或”的方式使用下面的一个或多个选项：

- PING_OPT_SILENT：此选项常用于程序调用 ping()的情况，压缩输出格式。
- PING_OPT_DONTROUTE：此选项用于不路由本地网络上的包。

返回：OK，如果无法从远端获得回应则返回 ERROR

操作系统造成的非法关机恢复一般需要分钟量级的时间，因此可以将 ping()的查询间隔设置为秒量级，例如 20s 检查一次。过分频繁的 ping 操作会导致网络上运行过多的非命令通道包，从而降低网络效率。由于不要求强定时，且需要避免 ping()重入，软件定时使用 while(1)+taskDelay 的方法。

在网络的命令通道建立后，发起 netCheckLink()函数监控网络。对于服务器端，只是等待 send()和 recv()出错释放的信号灯；对于客户端，还需要发起通过 recv()监控网络的函数 netCltCheckByRecv()。现在再给 netCheckLink()添加一个任务，发起 netCheckByPing()。此函数初始化 ping 库，并每隔一段时间 ping 一次控制端，一旦 ping 返回错误，则说明对方非法关机，此时释放网络监控信号灯。

recv()、send()和 ping()返回错误都说明网络通讯出错，它们都会释放信号灯，而只要其中任意一个释放信号灯，netCheckLink()就会进行网络关闭的工作。

例 10-1 中给出了使用 ping()监控网络的 netSvr.c 中的部分关键代码。在使用之前还必须在 board.h 中添加宏定义：

```
#define NET_CHECKBYPING_INTEVAL    20  /*每隔 20s 使用 ping()检查一次网络*/
#define TNAME_NETCHKBYPING      "tNetChkByPing"
#define TPRI_NETCHKBYPING  118
```

注意到例 10-1 中使用的远端 IP 是 boardIndex.remoteIP，此 IP 在 usrDefaultParamLoad()中被赋值为启动行中的 host IP。而启动行是被存储到 Flash 的，因此完全可以使用串口在启动时任意改变，保证代码的灵活。

添加了 ping 监控的采集板全部代码参见本书随附的光盘。

例 10-1 使用 ping()监控网络的程序(netSvr.c(部分))

```
/*************************************************************************
chengjy@felab, copyright 2002-2004
netSvr.c (部分)
使用 ping()监控网络需要添加和改变的函数，需要配合 board.h。这里只给出关键代码，全部代码参见本书光盘。
*************************************************************************/
#include "pingLib.h"
extern struct BoardIndex    boardIndex;
void netCheckByPing();

/*************************************************************************
 void netCheckLink()
 函数说明:    监测网络状态，出错即关闭网络并重新初始化
 参数:        无
 返回:        无
 调用:
              void netCloseAll(int mode)
              void netCheckByPing()
 被调用:
              void netInit();
 *************************************************************************/
void netCheckLink()
{
    /*由于优先极高，因此先创建信号灯才会进行网络的接收和发送*/
    semCmdLink = semBCreate(SEM_Q_FIFO,SEM_EMPTY);

    /*发起用 ping 监控网络的任务*/
    taskSpawn(TNAME_NETCHKBYPING,TPRI_NETCHKBYPING,0,USER_STACK_SIZE,
            (FUNCPTR) netCheckByPing,MODE_NET_REINIT,0,0,0,0,0,0,0,0,0);

    /*等待 send()和 recv()出错释放信号灯*/
    semTake(semCmdLink,WAIT_FOREVER);

    /*获得信号灯，表示网络连接断开*/
    semDelete(semCmdLink);
    taskSpawn(TNAME_NETCLOSEALL,TPRI_NETCLOSEALL,0,USER_STACK_SIZE,
```

```
            (FUNCPTR)netCloseAll,MODE_NET_REINIT,0,0,0,0,0,0,0,0,0);
}

/*************************************************************************
 void netCheckByPing()
 函数说明:    使用 ping 函数监控网络，一旦出错则释放信号灯，通知关闭网络
 参数:        无
 返回:        无
 调用:        无
 被调用:
    void netCheckLink()
 *************************************************************************/
void netCheckByPing()
{
    if(pingLibInit()!=ERROR)
    {
        while(1)
        {
            if(ping(boardIndex.remoteIP,1,PING_OPT_SILENT)!=ERROR)
            {
                taskDelay(NET_CHECKBYPING_INTEVAL*sysClkRateGet());
            }
            else
            {
                semGive(semCmdLink);
                break;
            }
        }
    }
    else
    {
        logMsg("netCheckByPing: unable to init pingLib\n",0,0,0,0,0,0);
    }
}
```

由于 ping 通常是由操作系统负责处理，可以脱离用户程序，因此程序比较简洁，但不利于控制。也可以对通信双方定义专门的网络检测包，由通信的一方 A 发出，另一方 B 接到此包后立刻返回，A 接到返回后再次发送包给 B。通信双方都可以通过衡量从发出包到接收包之间的时间来判断网络是否出现异常。对于 VxWorks，只要使用 watchdog 就可以实现。

10.2 脱离 malloc 的缓冲队列

第 3 章中介绍的缓冲队列的实现采用了 malloc()动态分配内存来存储每个命令并动态建立链表的方法。这样做的优势是：程序结构简单，编程者不必考虑内存分配的具体方法，建立的链表长度可以很长，只受到 malloc 成功失败的限制。但它会带来另一个潜在的问题：造成大量的内存碎片。由于网络命令的长短不一，且短命令居多，因此每个命令的命令长度和实际占用的空间比往往较小，没有合理地利用内存，且长时间运行容易造成系统的效率下降。

如果使用全局数组作为数据缓冲，则会大大增加编译后用户程序的长度。一般来说，为了动态地更新 VxWorks 到板载 Flash，一次命令的缓冲至少要有 1MB 以上，如果全部申请静态数组，则结合了用户程序的 VxWorks.bin 文件大小将增大 1MB，这反过来又要求更大的静态数组来缓存，最终无法通过申请全局数组的方法解决 VxWorks 动态更新，除非采取一边接收文件一边烧录的方法。但这种方法需要发送多个命令才能烧录完毕，且这些命令之间必须定义一定的协调方法，才能保证联合执行结果的正确。同时，为了多缓存一些数据，就需要多申请静态空间，而由于网络数据的不确定性，只能按照最多的原则来申请，增加了额外的 VxWorks 大小，也就是要求更大的硬件 Flash 空间，最终增加了硬件成本。

为了解决这个问题，用户程序可以利用 VxWorks 的用户保留内存，并根据网络缓冲队列的特点管理这部分内存。

10.2.1 VxWorks 操作系统下的内存分配

在 VxWorks 的 BSP 中，使用宏定义规定了内存各区间的使用方法。与 VxWorks 运行时内存分配的宏定义见图 10-1。

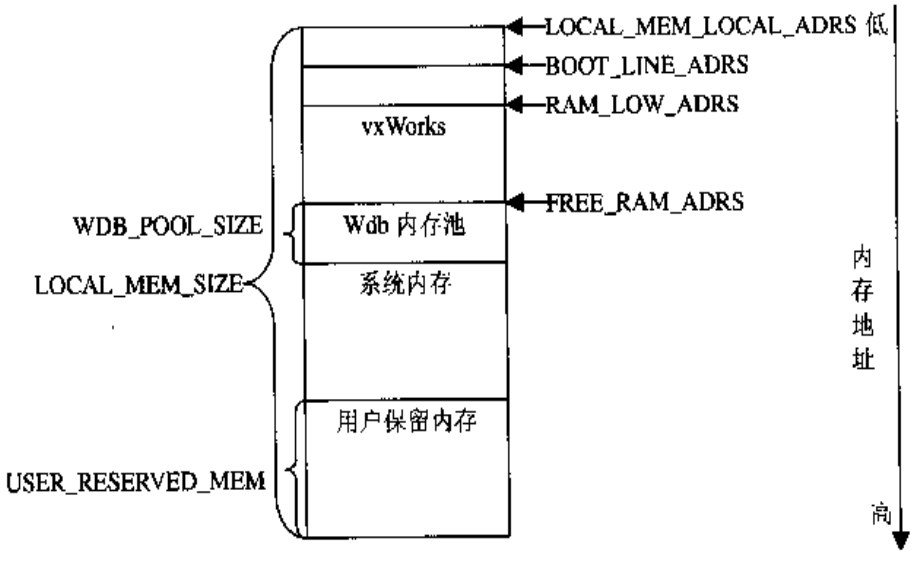


图 10-1 VxWorks 内存分配

其中：

LOCAL_MEM_LOCAL_ADRS	内存首地址
BOOT_LINE_ADRS	启动行在内存中的启始位置
RAM_LOW_ADRS	VxWorks 在内存中的启始位置
FREE_RAM_ADRS	VxWorks 在内存中的结束位置
WDB_POOL_SIZE	操作系统用户调试的内存池大小
USR_RESERVED_MEM	用户保留内存大小
LOCAL_MEM_SIZE	内存总空间

此部分与软件定义有关，不同的 BSP 可能使用不同的方式规划内存，需要仔细查看的文件包括 BSP 下定义采集板常数的.h 文件、config.h 文件和 sysLib.c 中的由结构体 PHYS_MEM_DESC 申明的空间分配全局数组。此部分更需要硬件的支持，尤其是用户保留内存。如果内存硬件没有达到 LOCAL_MEM_SIZE 规定的内存大小，首先受到影响的就是用户保留内存，因此在用户程序使用这些从 BSP 获得的宏之前，必须要对自己采集板的软硬件有详细的了解。

10.2.2 用户内存的规划和标示

由于命令接收缓存和发送缓存的结构可以完全一样，因此这里只考虑一个队列的情况。多个队列的扩展可以使用数组很容易地完成，两个队列的程序可以参见下面的例 10-2。

缓冲队列分为两个部分，一部分是存放网络命令的缓存，即第三章中程序 malloc 出的空间；另一部分是记录此缓存的首地址等信息的标志，即第三章中程序使用结构体建立的链表。为了不使用 malloc()，将缓存放在用户内存中，将链表修改成结构体的全局数组。为了标志结构体全局数组中头和尾对应的结构体，使用两个全局变量来标志有效命令的起始位置和有效命令的个数。

首先考虑缓存在用户内存中的分配。缓存的网络命令根据其紧急程度分为两类：紧急的命令采用后进先出策略，保证后添加的命令先发送；普通的命令采用先进先出策略，使命令按照添加的顺序逐个执行。为了保证这一点，命令在用户保留内存中的排列方式见图 10-2，命令的执行顺序为先紧急后普通，序号从小到大。

由于紧急命令只从一个位置进出，因此可以保证最后一条待执行的紧急命令的首地址保持定值；而普通命令由于进出位置不同，如果不移动命令在内存中存储的位置，整体必然会逐渐向添加方向移动。因此，将紧急和普通命令分别放在划分给这条缓冲队列的用户保留内存的两端，也就是说，初始化后的第一条紧急命令的首地址将为划分内存的首地址，而第一普通命令的结束地址将为划分内存的结束地址。随着命令的执行，普通命令将逐渐上移，如图 10-2 所示。为了尽量减少内存移动，只有当内存 A 的空间小于需要添加的命令的大小时，才将所有当前缓存在用户内存中的普通命令进行平移，使第一条普通命令的结束地址等于划分内存的结束地址。

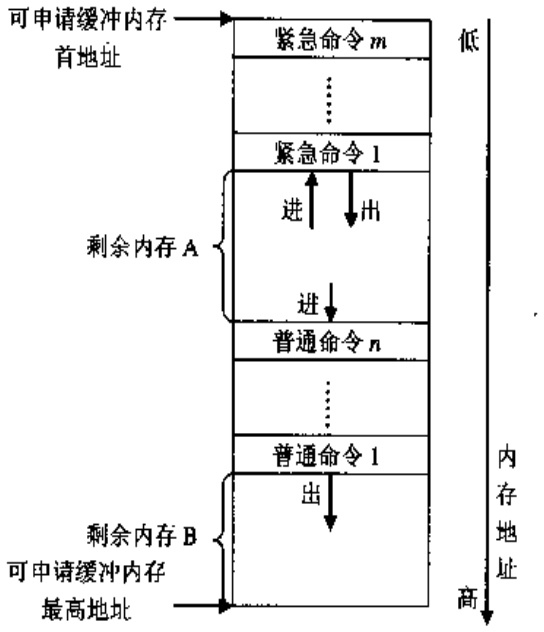


图 10-2　命令在用户保留内存中的排列方式

各命令在内存的存储中没有间隔，使用结构体来标志命令在内存中的存储。

```
struct cmdSingle
{
    UINT    pBegin;                 /*命令在内存中的存放地址*/
    int     len;                    /*命令长度*/
    unsigned char cmdIsHead;        /*命令紧急类型，1 为紧急，0 为普通*/
};
```

规定最大可以缓存的命令条数：

```
#define QUEUE_CMDNUM_MAX   20       /*每个缓冲队列最多缓存命令的条数*/
```

则可以定义全局结构体数组，用来存储每条命令在用户保留内存中的信息。

```
struct cmdSingle queue[QUEUE_CMDNUM_MAX];
```

而这些结构体的是否有效则用两个全局变量来标志：

```
int queueLen;                       /*队列长度*/
int queueHead;                      /*队列的头位置*/
```

各个变量在内存中的关系见图 10-3。

由于网络的命令解释任务和命令发送任务总是从队列头向外取数据，因此随着程序的运行，queueHead 将逐渐增加。采取环形方式处理有效结构体在数组中的排列。如果 queueLen 小于最大支持的长度 QUEUE_CMDNUM_MAX，则当 queueHead 为 0、下一个添加的命令为紧急命令时，将 queueHead 设置为 QUEUE_CMDNUM_MAX-1；当 queueHead+queueLen 等于 QUEUE_CMDNUM_MAX、下一个添加的命令为普通命令时，将 queueHead 设置为 0。故各元素按执行先后顺序排列在结构体数组中，有两种情况，见

图 10-4。

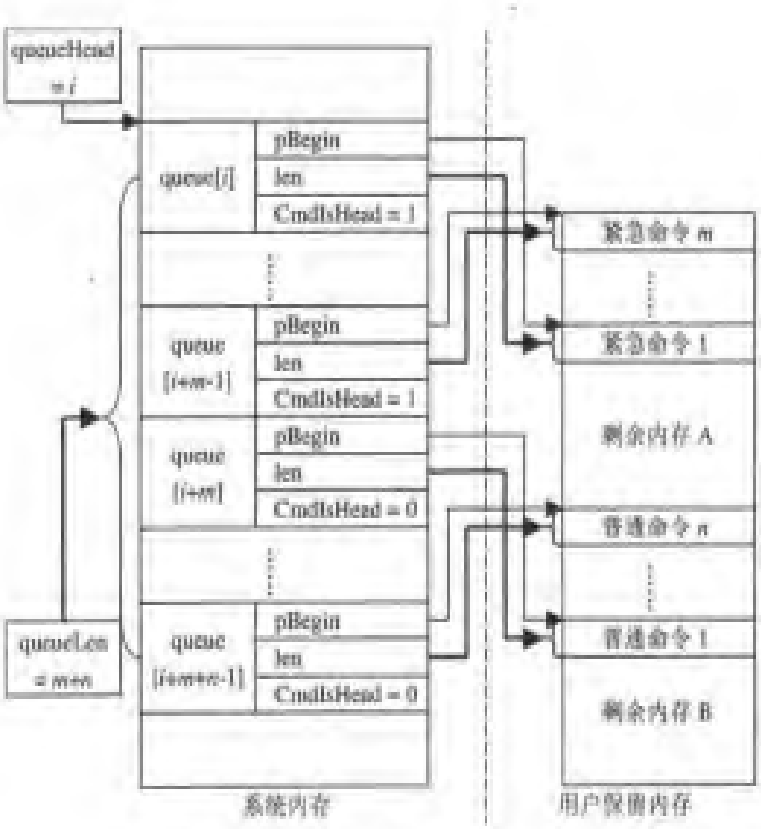


图 10-3　缓冲队列各变量的关系

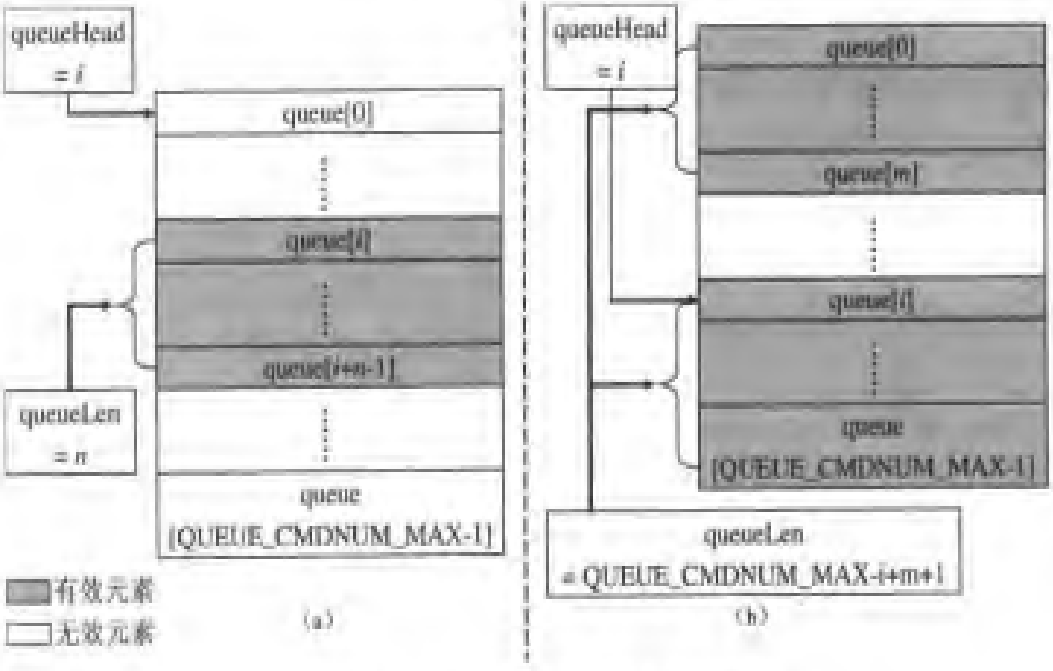


图 10-4　结构体数组中有效元素的排列

(a) 情况 1；(b) 情况 2

10.2.3 缓冲队列的基本操作和网络缓冲队列的使用

对于基于用户保留内存的网络缓冲队列来说，需要进行的操作有以下几项，后面将简略介绍它们的实现方法：

- 根据紧急程度从用户保留内存申请空间
- 释放用户保留程序的空间
- 将修改后的空间长度和首地址根据紧急程度添加到队列的头或尾
- 获得队列头指向的用户内存首地址
- 删除队列头
- 初始化队列
- 删除全部队列

1. 用户保留内存空间申请

当队列长度还没有达到最大长度限制 QUEUE_CMDNUM_MAX 时，可以添加新命令到队列，此时可以申请用户保留空间。根据 queueHead 和 queueLen 以及 queue[]中的各项内容，推算出图 10-2 中剩余内存 A 和剩余内存 B 的基地址和大小。如果申请的空间小于剩余内存 A 的大小，则直接在剩余内存 A 中划分，并返回首地址；如果申请的空间小于总的剩余空间，则先移动已经申请的内存，将所有的已申请普通命令向高地址移动，将剩余内存 B 的空间合并到剩余内存 A，即每个普通命令移动的偏移量为剩余内存 B 的大小，然后再从合并后的剩余内存 A 中划分新空间并返回。如果申请空间过大，超过所有剩余内存大小的总和，则直接返回空指针 NULL。新划分的空间必须紧挨着已经分配的空间，即如果申请紧急命令缓存空间，则返回的指针为剩余内存 A 的首地址；如果申请普通命令缓存空间，则返回的指针为（剩余内存 A 结束地址-申请空间大小）。

这里需要注意三个问题：

（1）缓冲内存、结构体数组以及标志结构体的 queueHead 和 queueLen 互相关联，因此每个操作应当避免和其他操作同时进行，且为了维护申请内存的连续，操作间必须有一定的顺序，为此需要使用信号灯保护。详细的信号灯保护方法参见后面关于队列使用的描述。

（2）在移动内存时，应当注意同时改变结构体数组中相应元素的指针 pBegin，以保证结构体和内存的对应关系仍然成立。

（3）在移动内存时，应当使用 memmove()，而不能使用 memcpy()，因为后者无法保证移动前后空间有重叠时拷贝的正确性。

2. 释放已申请的用户保留空间

参见“删除队列头”部分。

3. 将修改后的空间长度和首地址根据紧急程度添加到队列的头或尾

当队列长度还没有达到最大长度限制 QUEUE_CMDNUM_MAX 时，可以添加新的命令。如果需要添加的是紧急命令，则将 queueHead 向前移动 1，将 queueHead 指向的结构体各内容赋值，然后将 queueLen 加 1；如果添加的是非紧急命令，则将 queueHead+queueLen 指向的结构体内容赋值，再将 queueLen 加 1 即可。注意，这里使用了环形策略，即如果从

0 向前移 1，则变为 QUEUE_CMDNUM_MAX-1；如果从 QUEUE_CMDNUM_MAX-1 向后移，则变为 0。

4. 获得队列头指向的用户内存首地址

如果 queueLen 大于 0，则将返回 queue[queueHead].pBegin；否则返回空指针 NULL。

5. 删除队列头

如果 queueLen 大于 0，则将 queueHead 向后移 1，并将 queueLen 减 1。此时原 queueHead 指向的结构体中 pBegin 所指向的用户内存直接被释放。如果是紧急命令，则释放的空间并入剩余内存 A，普通命令释放的空间并入剩余内存 B。由于各个命令在用户保留内存中是紧密排列的，且只从队头删除元素，因此剩下的各个命令在用户保留内存中仍然可以保持紧密排列，并保持图 10-2 中的格式。

6. 初始化队列

将 queueLen、queueHead 都初始化为 0，并创建互斥型保护信号灯。

7. 删除全部队列

将 queueLen、queueHead 直接设置为 0，即释放了全部的用户保留内存空间。然后删除保护信号灯。

为了使外部函数在使用队列时不直接操作保护信号灯，因此再添加两个与队列相关的操作：获取和释放信号灯。同时，将队列也作为采集板初始化标志 boardWorkEnv.boardInit 的一项内容，以标志队列是否初始化。

网络使用两个缓冲队列，因此上面提到的变量和内存空间都需要两组。变量可以直接使用一维数组和二维数组解决，内存空间则是在用户保留内存中划分两块空间。除去初始化和删除两种操作，运行过程中需要使用的操作主要分为两组：申请空间，填写新数据并将其添加到队列；从对列取出队头指向的数据，执行完毕后删除队头。进行前一种操作的包括命令接收任务 netCMDRecv()和各种需要网络返回的函数（例如 t0x0200()等），进行后一种操作的包括命令解释任务 netCMDExplain()和命令发送任务 netCMDSend()。这两种操作间使用信号灯同步，同时还需要在操作前获得队列保护信号灯、在操作后释放队列保护信号灯。

以命令接收任务和命令解释任务为例，它们执行流程见图 10-5。

在上面的执行过程中，一旦任务出错退出，就会释放网络监控信号灯，通知网络监控任务关闭连接，删除缓冲队列和同步信号灯，并重新初始化网络。

需要网络发送命令的各任务和命令发送任务间使用缓冲队列传递信息的方法也类似。

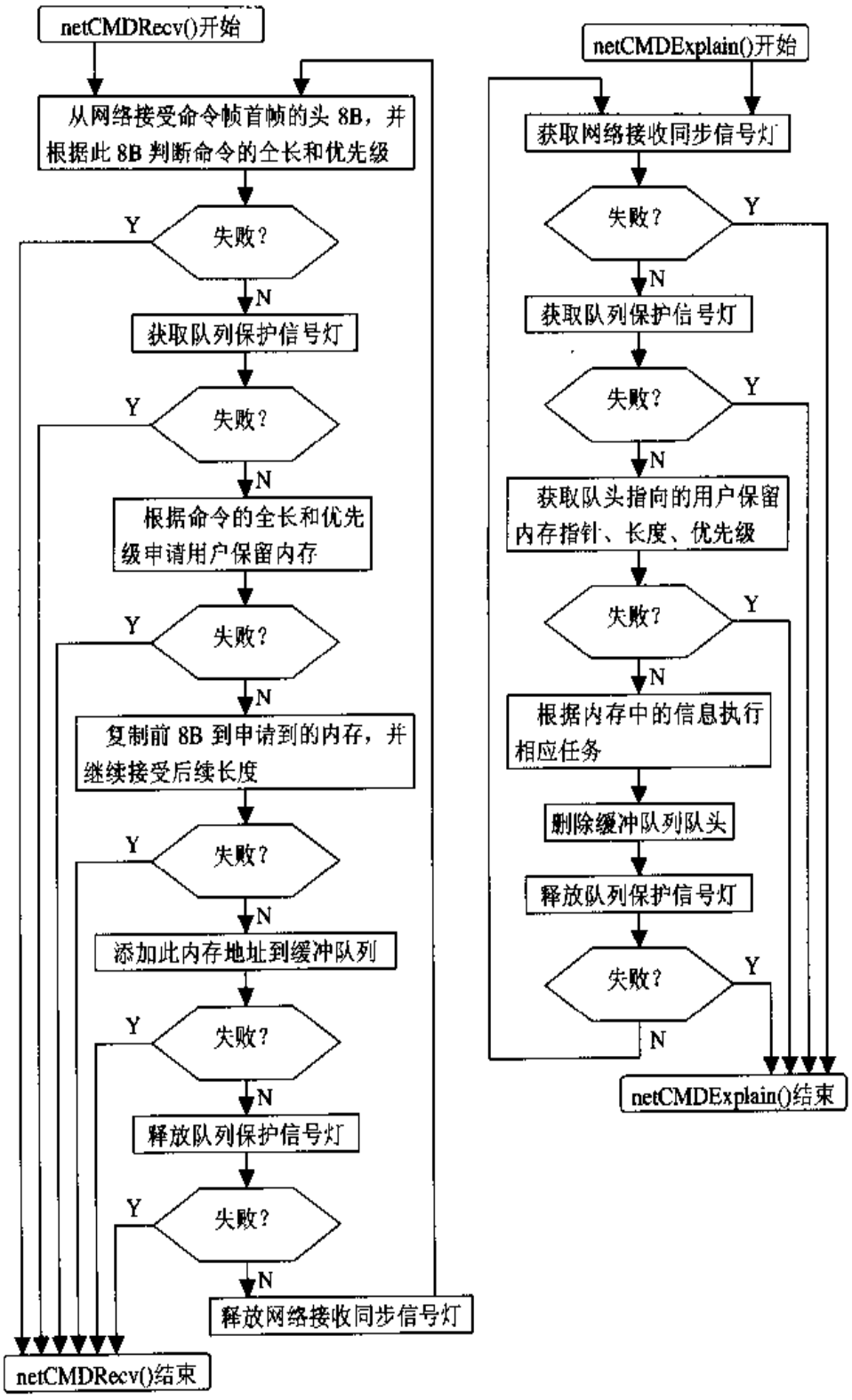


图 10-5　使用用户保留内存实现缓冲队列的命令接收任务和命令解释任务流程图

10.2.4　双缓冲队列实现实例

例 10-2 中给出了网络使用的双缓冲队列的软件实现。这里只列出 board.h 中的部分常

数定义和缓冲队列实现文件 netQueue.c，netSvr.c 和 netTask.c 中使用缓冲队列的地方可以参考图 10-5 中的流程进行修改。同时，netVariFunc.h 中的全局变量和函数入口，以及 netTask.c 头部的全局变量和函数的结构体数组也要作相应修改，修改方法已经在第 9 章做过介绍，这里也不再列出程序了。本书所附光盘中给出了经过修改的所有程序。

例 10-2　网络使用双缓冲队列例程[board.h（部分）和 netQueue.c]

```
/*************************************************************************
 chengjy@felab, copyright 2002-2004
 board.h
 采集板用户程序常数定义（部分，只给出双缓冲队列相关的宏定义）
 *************************************************************************/
#ifndef _BOARD_H
#define _BOARD_H

/*bootline 在内存中的位置，与 BSP 相同*/
#define LOCAL_MEM_LOCAL_ADRS    0x0
#define BOOT_LINE_OFFSET        0x4200
#define BOOT_LINE_ADRS          \
        ((char*) (LOCAL_MEM_LOCAL_ADRS+BOOT_LINE_OFFSET))

#define BOARD_UNINIT            0x00
#define BOARD_F9656_INITED      0x01
#define BOARD_NET_INITED        0x02
#define BOARD_GENET_INITED      0x04
#define BOARD_FLASH_INITED      0x08
#define BOARD_NETQUEUE_INITED   0x10
#define BOARD_SOFTPRAM_INITED   0x80
#define BOARD_INITED            0x9F

/*~~~~~~~~~~~~~~网络命令通道缓冲队列~~~~~~~~~~~~~~~~*/
/*每个命令的标志结构体*/
struct cmdSingle
{
 UINT pBegin;                /*命令在内存中的存放地址*/
 int len;                    /*命令长度*/
 unsigned char cmdIsHead;    /*命令紧急类型，1 为紧急，0 为普通*/
};

/*定义队列添加的优先级*/
#define QUEUE_PRI_HIGH      1    /*队头*/
#define QUEUE_PRI_LOW       0    /*队尾*/

/*定义队列的个数，对于双缓冲队列，定义为 2*/
```

```
#define QUEUE_NUM           2    /*必须是大于 0 的整数*/
#define QUEUE_INDEX_RECV    0
#define QUEUE_INDEX_SEND    1

/*与 BSP 定义相同，用以计算用户缓冲存放的基地址*/
#define LOCAL_MEM_LOCAL_ADRS        0x0
#define LOCAL_MEM_SIZE              0x04000000  /* 64 M */
#define USER_RESERVED_MEM           0x01000000  /* 16M */
#define USER_QUEUE_BASE_ADRS        \
(LOCAL_MEM_LOCAL_ADRS+LOCAL_MEM_SIZE-USER_RESERVED_MEM)
#define QUEUE_RECV_SIZEALL          0x00A00000  /*10M 接收总缓存*/
#define QUEUE_SEND_SIZEALL          0x00100000  /*1M 发送总缓存*/

/*命令缓存的空间分布*/
#define QUEUE_RECV_BASE_ADRS        USER_QUEUE_BASE_ADRS
#define QUEUE_RECV_END_ADRS         \
            (QUEUE_RECV_BASE_ADRS+QUEUE_RECV_SIZEALL)
#define QUEUE_SEND_BASE_ADRS        QUEUE_RECV_END_ADRS
#define QUEUE_SEND_END_ADRS         \
                (QUEUE_SEND_BASE_ADRS+QUEUE_SEND_SIZEALL)
#define QUEUE_CMDNUM_MAX            20   /*每个缓冲队列最多缓存命令的条数*/

#endif   /*_BOARD_H*/

/*************************************************************************
 chengjy@felab, copyright 2002-2004
 netQueue.c
 网络缓冲队列操作。队列的个数由宏定义 QUEUE_NUM 规定。
 函数:
        void queueInitVx();
        char queueAddVx(int index, UINT pBuff, unsigned char pri, int len);
        UINT queueGetHeadPointVx(int index);
        char queueDelHeadVx(int index);
        void queueCloseVx();
 调用:   无
 被调用: 网络通信程序及用户程序的初始化部分
 *************************************************************************/
#include "vxWorks.h"
#include "taskLib.h"

#include "board.h"

extern struct BoardWorkEnv boardWorkEnv;
```

```
int queueLen[QUEUE_NUM];                                    /*队列长度*/
int queueHead[QUEUE_NUM];                                   /*队列的头位置*/

struct cmdSingle queue[QUEUE_NUM][QUEUE_CMDNUM_MAX];        /*队列内存标志*/
SEM_ID  semQueueBuff[QUEUE_NUM];                            /*队列保护信号灯*/

void queueInitVx();
char queueAddVx(int index, UINT pBuff, unsigned char pri, int len);
UINT queueGetHeadPointVx(int index);
char queueDelHeadVx(int index);
void queueCloseVx();

UINT queueMallocVx(int index, unsigned char cmdIsHead, int size);
STATUS queueSemTakeVx(int index, int timeout);
STATUS queueSemGiveVx(int index);

/*****************************************************************************
 void queueInit()
 函数说明:    队列初始化
 参数:        无
 返回:        无
 调用:        无
 被调用:
        netSvr.c
            void netInit(int mode)
 *****************************************************************************/
void queueInitVx()
{
    int i;
    /*如果已经初始化了，不再重复进行*/
    if(!(boardWorkEnv.boardInit&BOARD_NETQUEUE_INITED))
    {
        /*初始化全局变量*/
        queueLen[QUEUE_INDEX_RECV]      = 0;
        queueLen[QUEUE_INDEX_SEND]      = 0;
        queueHead[QUEUE_INDEX_RECV]     = 0;
        queueHead[QUEUE_INDEX_SEND]     = 0;

        /*创建保护信号灯*/
        semQueueBuff[QUEUE_INDEX_RECV] = semBCreate(SEM_Q_PRIORITY,
                                                    SEM_EMPTY);
        if(semQueueBuff[QUEUE_INDEX_RECV]==NULL)
        {
            logMsg("queueInitVx: unable to semBCreate for net queue recv\n",
```

```
                0,0,0,0,0,0);
            return;
        }
        else
        {
            semQueueBuff[QUEUE_INDEX_SEND] = semBCreate(SEM_Q_PRIORITY,
                                                        SEM_EMPTY);
            if(semQueueBuff[QUEUE_INDEX_SEND]==NULL)
            {
                logMsg("queueInitVx: unable to semBCreate for net queue send\n",
                        0,0,0,0,0,0);
                semDelete(semQueueBuff[QUEUE_INDEX_RECV]);
                return;
            }
        }
        boardWorkEnv.boardInit = boardWorkEnv.boardInit | BOARD_NETQUEUE_
                                 INITED;
    }
}

/*****************************************************************************
 char queueAddVx(int index, UINT pBuff, unsigned char pri, int len)
 函数说明:    队列元素添加,根据 pri 决定添加到队头还是队尾
 参数:
        index   需要添加到的队列号，必须大于或等于 0 且小于 QUEUE_NUM
        pBuff   从网络接收数据或者待发送数据的首地址指针
        pri     添加的优先级
        len     添加数据的总长度，单位 B
 返回:       成功添加返回 STATUS_NORMAL，否则返回 STATUS_WARNING。
 调用:       无
 被调用:
        netSvr.c
        void netCMDRecv();
        char netCMDAdd(unsigned char *pBuff, int buffLen, int cmdNum,
                       unsigned char priority);
 *****************************************************************************/
char queueAddVx(int index, UINT pBuff, unsigned char pri, int len)
{
    struct cmdSingle *pCmd;
    char state = STATUS_WARNING;
    if(boardWorkEnv.boardInit&BOARD_NETQUEUE_INITED)
    {
        /*检查是否可以添加数据*/
        if(queueLen[index]<QUEUE_CMDNUM_MAX)
```

```
        {
            if(pri==QUEUE_PRI_HIGH)
            {
                if(queueLen[index]!=0)
                {
                    if(queueHead[index]==0)
                        queueHead[index] = QUEUE_CMDNUM_MAX-1;
                    else
                        queueHead[index]--;
                    }
                pCmd = (struct cmdSingle*)(&(queue[index][queueHead[index]]
                        .pBegin));
            }
            else
            {
                pCmd = (struct cmdSingle*)(&(queue[index]
            [(queueHead[index]+queueLen[index])%QUEUE_CMDNUM_MAX].pBegin));
            }
            queueLen[index]++;
            pCmd->pBegin     = pBuff;
            pCmd->len        = len;
            pCmd->cmdIsHead  = pri;
            state = STATUS_NORMAL;
        }
    }
    return(state);
}

/*****************************************************************************
 UINT queueGetHeadPointVx(int index)
 函数说明:    获取队列头的数据头指针。
 参数:
        index          选择缓冲队列，发送还是接收
 返回:        如果队列已经被初始化，且不为空，则返回头指针；否则返回 NULL
 调用:        无
 被调用:
        netSvr.c
        void netCMDExplain();
        void netCMDSend();
 *****************************************************************************/
UINT queueGetHeadPointVx(int index)
{
    if(boardWorkEnv.boardInit&BOARD_NETQUEUE_INITED)
    {
```

```
        if(queueLen[index]>0)
        {
            return(queue[index][queueHead[index]].pBegin);
        }
    }
    else
        return(NULL);
}

/*****************************************************************************
 char queueDelHeadVx(int index)
 函数说明:    删除队列头
 参数:
        index        选择缓冲队列，发送还是接收
 返回:        如果队列已经被初始化，且不为空，则删除后返回 STATUS_NORMAL；否则返回
        STATUS_WARNING
 调用:        无
 被调用:
        netSvr.c
        void netCMDExplain();
        void netCMDSend();
 *****************************************************************************/
char queueDelHeadVx(int index)
{
    char state = STATUS_WARNING;
    if(boardWorkEnv.boardInit&BOARD_NETQUEUE_INITED)
    {
        /*如果还有命令可以被删除*/
        if(queueLen[index]>0)
        {
            queueHead[index] = (queueHead[index]+1)%QUEUE_CMDNUM_MAX;
            queueLen[index]--;
            state = STATUS_NORMAL;
        }
    }
    return(state);
}

/*****************************************************************************
 void queueCloseVx()
 函数说明:    删除队列中的全部内容，并删除信号灯
 参数:        无
 返回:        无
 调用:        无
```

```
被调用:
            netSvr.c
            void netCloseAll(int mode)
*********************************************************************/
void queueCloseVx()
{
    /*如果已经被初始化，才能够关闭*/
    if(boardWorkEnv.boardInit&BOARD_NETQUEUE_INITED)
    {
        queueLen[QUEUE_INDEX_RECV]      = 0;
        queueLen[QUEUE_INDEX_SEND]      = 0;
        queueHead[QUEUE_INDEX_RECV]     = 0;
        queueHead[QUEUE_INDEX_SEND]     = 0;

        /*删除保护信号灯*/
        semDelete(semQueueBuff[QUEUE_INDEX_RECV]);
        semDelete(semQueueBuff[QUEUE_INDEX_SEND]);

        /*关闭结束*/
        boardWorkEnv.boardInit = boardWorkEnv.boardInit &
                        (~((UINT)(BOARD_NETQUEUE_INITED)));
    }
}

/*********************************************************************
 UINT queueMallocVx(int index, unsigned char cmdIsHead, int size)
 函数说明:    在用户保留的RAM空间中申请一块内存
 参数:
        index       选择缓冲队列，发送还是接收
        cmdIsHead   选择开辟的内存准备存储到队头还是队尾
        size        需要开辟的空间，单位B
 返回:       申请成功返回申请到的内存首地址；否则返回NULL
 调用:       无
 被调用:
        netSvr.c
        void netCMDRecv();
    netTask.c
        需要网络返回的命令
 *********************************************************************/
UINT queueMallocVx(int index, unsigned char cmdIsHead, int size)
{
    int i;
    int queueLenThis,queueHeadThis;
    UINT pHeadBegin,pHeadEnd,pRearBegin,pRearEnd;
```

```
int remainLen,shiftLen;
struct cmdSingle *pCmd;

if(boardWorkEnv.boardInit&BOARD_NETQUEUE_INITED)
{
    queueLenThis = queueLen[index];
    /*检查队列中是存储的命令数是否已经到达极限*/
    if(queueLenThis<QUEUE_CMDNUM_MAX)
    {
        queueHeadThis = queueHead[index];

        /*寻找已经占用空间的首尾地址*/
        if(index == QUEUE_INDEX_RECV)
        {
            pHeadBegin  = QUEUE_RECV_BASE_ADRS;
            pHeadEnd    = QUEUE_RECV_BASE_ADRS;
            pRearBegin  = QUEUE_RECV_END_ADRS;
            pRearEnd    = QUEUE_RECV_END_ADRS;
        }
        else
        {
            pHeadBegin  = QUEUE_SEND_BASE_ADRS;
            pHeadEnd    = QUEUE_SEND_BASE_ADRS;
            pRearBegin  = QUEUE_SEND_END_ADRS;
            pRearEnd    = QUEUE_SEND_END_ADRS;
        }
        i=0;
        while(i<queueLenThis)
        {
            pCmd = (struct cmdSingle*)(&(queue[index]
                   [(queueHead[index]+i)%QUEUE_CMDNUM_MAX].pBegin));
            if(pCmd->cmdIsHead!=0) /*添加到对头的命令*/
            {
                /*因为根据队列的使用方法，pHeadBegin 为固定值，
                  所以不需要为 pHeadBegin 赋值*/
                pHeadEnd = pCmd->pBegin;
            }
            else        /*添加到队尾的命令*/
            {
                if(pHeadBegin==QUEUE_SEND_BASE_ADRS)
                {
                    pRearBegin = pCmd->pBegin+pCmd->len;
                }
                pRearEnd = pCmd->pBegin;
```

```
        }
        i++;
    }

    /*计算剩余的总空间*/
    if(index == QUEUE_INDEX_RECV)
        remainLen = QUEUE_RECV_SIZEALL - (pHeadBegin-pHeadEnd)
                    - (pRearBegin-pRearEnd);
    else
        remainLen = QUEUE_SEND_SIZEALL - (pHeadBegin-pHeadEnd)
                    - (pRearBegin-pRearEnd);

    if(remainLen<size) /*申请空间超过剩余空间*/
    {
        return(NULL);
    }
    else
    {
        if(pRearEnd-pHeadBegin<size)
        /*中间的空间不足，需要移动原有的内存*/
        {
            /*计算需要移动的相对位置*/
            if(index == QUEUE_INDEX_RECV)
                shiftLen = pRearBegin - QUEUE_RECV_END_ADRS;
            else
                shiftLen = pRearBegin - QUEUE_SEND_END_ADRS;
            if(shiftLen!=0)
            {
                i=0;
                while(i<queueLenThis)
                {
                    pCmd = (struct cmdSingle*)(&(queue[index]
                [(queueHead[index]+i)%QUEUE_CMDNUM_MAX].pBegin));
                    if(pCmd->cmdIsHead==0)
                    {
                    /*添加到队头的命令，队尾命令不需要处理*/
                        memmove(pCmd->pBegin+shiftLen, pCmd->pBegin,
                                pCmd->len);
                        pCmd->pBegin = pCmd->pBegin - pRearBegin;
                    }
                    i++;
                }
                pRearBegin = pRearBegin+shiftLen;
                pRearEnd = pRearEnd+shiftLen;
```

```
                }
            }
            if(cmdIsHead!=0)    /*申请队头空间*/
                return(pHeadBegin+size);
            else
                return(pRearEnd-size);
        }
    }
    else        /*队列长度已经达到极限*/
    {
        return(NULL);
    }
}
else            /*队列尚未初始化，直接返回*/
{
    return(NULL);
}
}

/*****************************************************************************
 STATUS queueSemTakeVx(int index, int timeout)
 函数说明:    获取缓冲队列内存保护信号灯
 参数:
        index       选择缓冲队列，发送还是接收
        timeout     超时时间，定义方法等同 semTake
 返回:       获取成功返回 OK，否则返回 ERROR
 调用:       无
 被调用:
        netSvr.c
            void netCMDRecv()
            void netCMDExplain()
            void netCMDSend()
        netTask.c
            需要网络返回的命令
 *****************************************************************************/
STATUS queueSemTakeVx(int index, int timeout)
{
    if(boardWorkEnv.boardInit&BOARD_NETQUEUE_INITED)
    {
        /*如果已经初始化，则获取信号灯*/
        return(semTake(semQueueBuff[index],timeout));
    }
    else
    {
```

```
            return(ERROR);
        }
    }

    /*************************************************************************
     STATUS queueSemGiveVx(int index)
     函数说明:    释放缓冲队列内存保护信号灯
     参数:
            index    选择缓冲队列，发送还是接收
     返回:        释放成功返回 OK，否则返回 ERROR
     调用:        无
     被调用:
            netSvr.c
                void netInit(int mode)
                void netCMDRecv()
                void netCMDExplain()
                void netCMDSend()
        netTask.c
            需要网络返回的命令
     *************************************************************************/
    STATUS queueSemGiveVx(int index)
    {
        if(boardWorkEnv.boardInit&BOARD_NETQUEUE_INITED)
        {
            /*如果已经初始化，则释放信号灯*/
            return(semGive(semQueueBuff[index]));
        }
        else
        {
            return(ERROR);
        }
    }
```

10.3 取舍权衡

算法优化完毕之后，还需要根据程序的具体情况作一些调整。

10.3.1 BSP 中宏定义与用户程序的统一

在本章的程序中，用户程序使用了 BSP 中定义的宏定义。不仅如此，有些和硬件直接相关的程序可能会放在 BSP 中，而用户程序需要根据硬件初始化的情况来决定程序的运行。最简单的保持统一的方法就是在用户程序的.h 文件中直接加上和 BSP 中一样的宏定义，为了保证将代码直接拷贝到 BSP 时不破坏原有定义，使用#ifndef 作为定义条件，例如：

```
#ifndef LOCAL_MEM_LOCAL_ADRS
#define  LOCAL_MEM_LOCAL_ADRS   0x0
#endif
```

如果采用第七章中介绍的基于档案库的用户程序结合方法，由于用户程序的宏定义已经在编译成.a 文件时使用了，不会再带入到 BSP 中，则不需要#ifndef，直接#define 就可以了。

将宏定义直接带入用户程序的缺点是：在将用户程序移植到新的 BSP 时，可能会由于新 BSP 中宏定义与用户程序.h 的宏定义不一致，而导致运行时出现致命错误。

可以提醒编程者的一个方法是：在 BSP 中申请和宏定义一样的全局变量，在用户程序中原来使用 BSP 宏的地方使用这些全局变量。例如：

BSP usrConfig.c:

```
UINT localMemLocalAdrs = LOCAL_MEM_LOCAL_ADRS;
```

用户程序：

```
extern UINT localMemLocalAdrs;
```

这样，对于 BSP 中有申请全局变量的 VxWorks，用户程序就可以直接使用；而新移植的 BSP 中没有这个全局变量，因此在下载时就会报错，见图 10-6。提醒编程者，在 BSP 中添加变量后再使用。本书光盘中的例 10-2 中定义了 localMemLocalAdrs，但使用/**/将其注释了。读者可以去掉注释符，然后重新编译下载程序，以察看报错提醒。

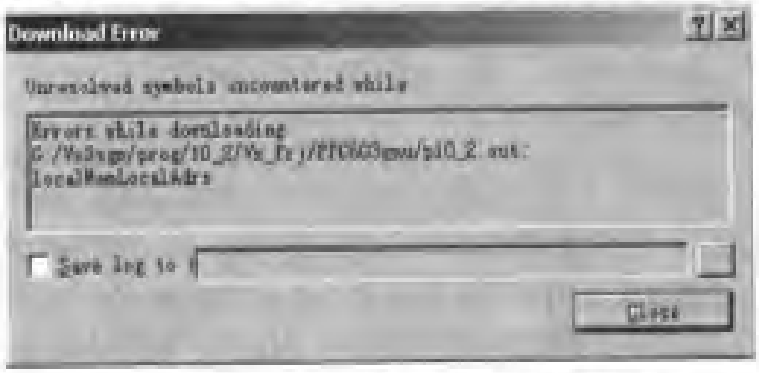


图 10-6　全局变量方法使用 BSP 宏时的报错提醒

需要注意的是，如果没有函数调用这些申明为外部的变量，下载时就不会报错。

此外，不要把全局变量申明放到 bootConfig.c 等这种 Bootrom 使用的文件中，因为 Bootrom 所在的内存区域会在 VxWorks 加载到内存后作为系统内存或者用户保留内存使用，这些文件中的全局变量申明不会带到 VxWorks 和用户程序中。

10.3.2　软件大小和执行效率

所有全局变量都将在 VxWorks 映像中占有同样大的空间，因此申明过多的全局变量会导致 VxWorks 映像变大。局部变量在运行时动态申请，因此相对于全局变量，使用局部变量可以减小映像的大小，但同时由于其申请空间时需要进行内存操作，因此会占用运行时

问。

对于需要多次取用的数组元素，可以使用局部变量的方法来加快运行速度。例如，如果 pBuff[16]的值在某个函数内多次作为参数使用，且在使用期间不发生改变，就可以在函数开始时用 char temp = pBuff[16]来存储其值。每次使用 temp 只需要一个读操作，而每次使用 pBuff[16]时还需要计算内存地址，因此速度较慢。

同样，对于某些和硬件相关的变量，变量值的获得需要进行硬件操作。如果需要多次取用，且这些次取用的过程与变量相关硬件不发生变化，也为其申请一个全局变量，以避免每次取值时由于操作硬件而耽误的时间。例如第五章中提到的分配给 PCI 器件的内存地址和中断号，它们的值由 BSP 和硬件连线决定，用户程序中取用只是作为参考值，因此只需要在初始化时从 PCI 首地址读出其值，并将其赋给全局变量，后续程序中都只使用全局变量。使用这种方法，编程者必须事先考虑好全局变量和硬件参数的一致性，否则一旦硬件值发生改变，而全局变量没有相应的改动机制，将发生严重的错误。

理论上来说，可以通过将所有的全局变量都存储到用户保留内存中的方法来减小软件大小，但实际上，将全局变量变为用户保留内存中指定地址的固定长度数据，丢失了软件的灵活性，且一个全局变量最大也只是能节省 4B 空间。因此只将长度较大的数组类数据存放在用户内存中。在用户程序的调试过程中，仍然使用全局数组；在程序调试完毕后，用宏定义代替全局数组的首地址指针。例如，对于全局数组：

```
char globalBuff[100];
```

在调试结束后，在.h 文件中添加宏定义：

```
#define SHIFT_GLOBALBUFF    0x00B00000
#define USR_BUFF_BASE_ADRS \
        (LOCAL_MEM_LOCAL_ADRS+LOCAL_MEM_SIZE-USER_RESERVED_MEM)
#define globalBuff      ((char*)( USR_BUFF_BASE_ADRS+ SHIFT_GLOBALBUFF))
```

分别定义 globalBuff 在用户保留内存中存储的偏移地址、用户保留内存的基地址和 globalBuff 指针，然后删除原来的全局变量定义。由于 VxWorks5.4 使用的内存映射方式是直接映射，数组各元素按顺序连续排列在内存中，因此用户程序中调用 globalBuff 的地方不需要做任何修改，就可以继续使用。

10.4 总结

本章介绍了与操作系统直接相关的软件优化方法，包括基于 ping()的网络监控和基于用户保留内存的缓冲队列实现，它们的运行直接依赖于 VxWorks5.4 的某些特性。最后介绍了 BSP 中宏定义与用户程序的统一方法、软件大小和执行效率综合考虑，它们也和 VxWorks 系统的特性有关，针对性很强，能够帮助编程者建立更高效的 VxWorks 用户程序。