

嵌入式实时操作系统 VxWorks 及其开发环境 Tornado



Powered by xiaoguo's publishing studio
QQ:8204136



前 言

第 1 章 嵌入式实时系统软件设计	1
1.1 嵌入式实时系统.....	1
1.2 嵌入式实时系统软件开发设计.....	8
1.3 嵌入式实时操作系统.....	13
1.4 如何选择实时操作系统.....	19
1.5 本书的组织.....	21
第 2 章 VxWorks 操作系统与实时应用	23
2.1 实时应用的基本需求.....	23
2.2 VxWorks 简介.....	25
2.3 Tornado 集成开发环境简介.....	31
第 3 章 Tornado 使用初步	37
3.1 TornadoII 的新特征.....	37
3.2 TornadoII 安装.....	38
3.3 Tornado 简单教程.....	48
3.4 Tornado1.0.1 下的工程开发.....	67
第 4 章 VxWorks 任务与任务编程接口	85
4.1 VxWorks 任务.....	85
4.2 VxWorks 任务编程接口.....	98
4.3 POSIX 调度接口.....	105
第 5 章 任务间通信	111
5.1 VxWorks 任务间通信机制.....	111

5.2	共享数据结构.....	111
5.3	互斥	113
5.4	信号量	115
5.5	消息队列	152
5.6	管道	171
第 6 章	信号、中断处理与定时机制.....	177
6.1	信号 (Signals)	177
6.2	中断服务程序.....	180
6.3	看门狗	187
6.4	POSIX 时钟和计时器	196
6.5	POSIX 内存上锁接口	203
第 7 章	建立调试环境与实例分析.....	205
7.1	建立调试环境.....	205
7.2	实例分析	211
第 8 章	网络编程	219
8.1	VxWorks 网络组件	219
8.2	TCP/IP 协议	221
8.3	套接字基础	227
8.4	Socket 编程接口.....	232
8.5	Socket 的原始方式.....	246
第 9 章	客户 / 服务器编程.....	263
9.1	客户 / 服务器.....	263
9.2	客户端程序设计.....	264
9.3	服务器端程序设计	270
9.4	服务端程序结构.....	272
9.5	多协议 (TCP、UDP) 服务端.....	274
9.6	编程实例	274
第 10 章	VxWorks 操作系统配置	289
10.1	VxWorks 的目录与文件.....	289
10.2	VxWorks 的板级支持包 BSP.....	292
10.3	VxWorks 的配置文件与配置项.....	294
10.4	VxWorks 的初始化	298

10.5 可选的 VxWorks 配置.....	305
第 11 章 编程实战	311
11.1 程序执行时间.....	311
11.2 多任务.....	313
11.3 信号量.....	315
11.4 消息队列	318
11.5 轮转调度算法.....	321
11.6 基于优先级的抢占式调度	324
11.7 优先级转置.....	327
11.8 信号	332
11.9 中断服务程序.....	335
附录 参考文献	339

本章主要介绍嵌入式实时系统软件设计有关的基础知识,帮助读者建立嵌入式软件设计的基本概念。

主要包括嵌入式系统、实时系统的定义和特点;嵌入式实时软件开发的步骤;嵌入式实时操作系统的概念,选择准则;本书的章节组织等内容。

1.1 嵌入式实时系统

1.1.1 嵌入式系统

在计算机技术和信息技术高速发展的今天,计算机和计算机技术大量应用在我们的日常生活中。现代的计算机早已超出早期计算机的概念,广泛应用的嵌入式计算机(Embedded Computer)便是其中一种。嵌入式计算机或者叫嵌入式系统,源于20世纪60年代,是一种不被用户所觉察的专用计算机。

嵌入式系统,很难给其写出确切定义,多指深藏于工业系统、武器系统或一些机电仪表设备、消费电子类产品内部,完成一种或多种特定功能的计算机系统,是软硬件的紧密结合体。类似与BIOS的工作方式。具有软件代码小,高度自动化,响应速度快等特点。特别适合于要求实时的和多任务的应用体系。

这些专用计算机系统是以嵌入式计算机为技术核心,围绕应用系统的功能、可靠性、成本、体积、功耗等严格要求来开发设计的。它一般由嵌入式微处理器、外围硬件设备、嵌入式操作系统以及特定的应用程序等四个部分组成,用于实现对其他设备的控制(Control)、监视(Monitor)或管理(Management)等功能。



图 1.1 嵌入式计算机技术无处不在

其实，最早在 20 世纪 60 年代嵌入式计算机技术已用于国防系统中，是一种所谓的专用机。二十世纪 70~80 年代嵌入式微处理器逐渐应用于工业控制等领域。目前，从航天飞机到家用微波炉，嵌入式计算机系统广泛应用于工业、交通、能源、通信、科研、医疗卫生、国防以及日常生活（消费电子、CE）等领域，并发挥着极其重要的作用。

随着应用对智能控制需求的不断增长，同时也对嵌入式微处理器的运算速度、可扩充能力、系统可靠性、功耗和集成度等方面提出了更高的要求。

为了适应各方面的需求，嵌入式微处理器体系结构也经历了一个从 CISC 到 RISC 和 Compact RISC；位数从 4 位、8 位、16 位、32 位到 64 位；寻址空间从 64kB 到 16MB 甚至更大；处理速度从 0.1 MIPS 到 2000 MIPS；常用封装从 8 个引脚到 144 个引脚的过程。处理器的功耗也有了明显降低；集成度进一步提高。近闻，美国国家半导体公司又推出了一款高度集成的微处理器 SC1400（片上系统 System-On-A-Chip），它代表了目前嵌入式微处理器技术的最高集成度。

目前国外许多大处理器生产厂商（Motorola、Intel、AMD、日立、NEC 等）纷纷推出各种嵌入式微处理器。最具有代表性的是：Motorola 的 PowerPC 系列；Intel 的 StrongArm 系列和 National Semiconductor 的 x86 系列。其中最具影响力的当数 Motorola 的 PowerPC 系列，由于 PowerPC 系列微处理器种类繁多，而且性能优越，系统集成度高，扩展能力强，可以广泛应用于各类嵌入式系统中。因此，Motorola 已成为当今全球最大的嵌入式微处理器生产商，PowerPC 系列微处理器成为当今嵌入式系统应用的主流。

另一种嵌入式微处理器的热点就是越来越多的人在磁盘控制器、数码相机、手持电话、调制解调器等方面使用 DSP。采用 DSP 的好处是可以大大减少系统内 CPU 的数目，提高效率，并使编程简单化，但是毕竟 DSP 不能完全替代 CPU 的功能。目前已经有公司宣布推出

复合型的微处理器，如 Motorola 的 M.Core（一种新一代的 16/32 位微处理器），它将跨越 CPU 与 DSP。据悉，TI、Siemens 也在开发相似的产品。

1.1.2 实时系统

工业控制、舰船武器系统控制、航空航天等领域的多数嵌入式系统他们有一个共同的特性：对系统的响应时间有严格要求，这些系统也被称为实时系统。

虽然实时系统（Real-Time Systems）的发展已有四十多年的历史，但至今尚无一个能被人们广泛接受的定义。牛津计算机辞典对实时系统的定义为：“实时系统是指那些产生系统输出的时间对于系统是至关重要的系统。这通常是因为输入对应于物理世界的某些运动，同时输出也与一些运动相关。从输入到输出的滞后时间必须足够小到一个可以接受的时限（timeline）内。因此实时系统逻辑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”。

注意这里的时限指系统执行时间的限制，而系统功能的实现一般要通过软硬件的相互配合来完成，因此这些组成系统的软硬件的执行也就有了相应的时间限制。相对软件而言，这种时间限制体现到组成软件系统的任务的时间限制，也称之为时限（deadline，亦称死线）。

系统时限的大小和具体系统有关。例如，在一个导弹制导系统中，要求在几个毫秒内产生输出，而在一个计算机控制的汽车生产线中，系统的反应可以放松在一秒内。任务（将会在第四章中介绍）的时限是由软件划分所决定。

近半个世纪以来，随着计算机技术的发展，实时计算机系统在工业过程控制、航空航天、交通管理、作战指挥控制系统以及科学实验和日常生活等领域中得到迅猛发展。

这些实时系统可根据时限对其性能（或效益）影响程度的不同，分为软实时系统（soft real-time systems）和硬实时系统（hard real-time systems）。这是说如果任务在时限到来之前未能完成，前者只可能使系统性能降低（如图 1.2），而后者的后果无法预测且多是灾难性的（如图 1.3）。

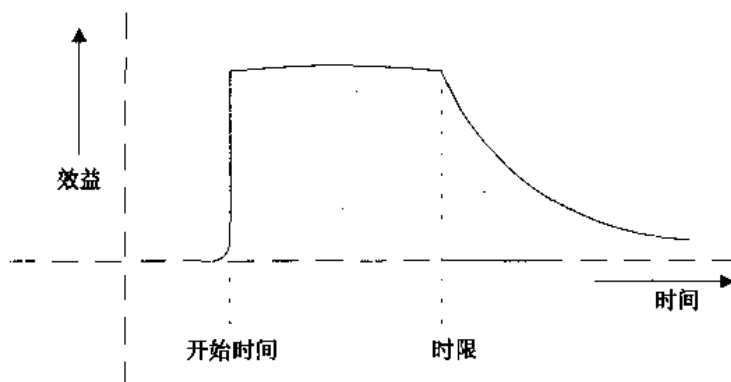


图 1.2 一个软时限事件示意图

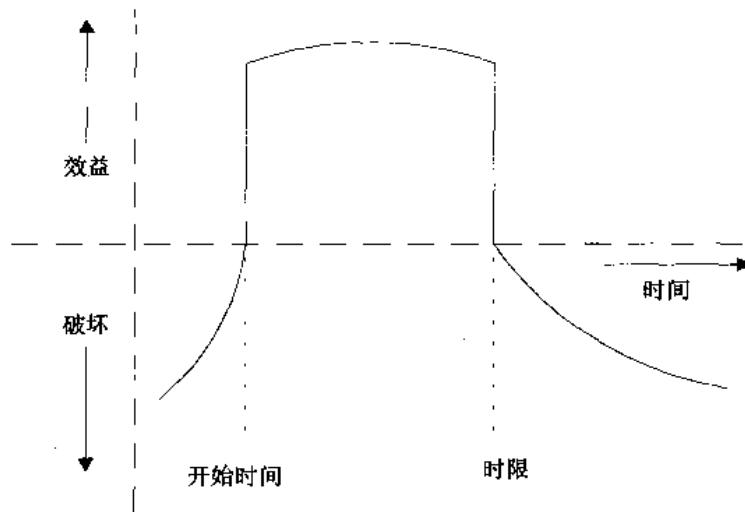


图 1.3 一个硬时限事件示意图

在一些较大实时系统中，并不是所有的计算时间都是硬实时的或关键的。一些事件没有时限，另一些可能仅仅只有软时限，软时限是指时限的错过不会损坏系统的完整性。

另外还可以根据应用领域的不同，将实时系统分为实时信息数据处理系统和实时控制系统。

实时信息数据处理系统和批处理系统的本质区别是，用户通过终端设备向系统提出信息处理请求，系统“实时”响应用户请求，完成处理后通过终端回答用户。这里的“实时”是相对用户反应而言，一般为软实时系统。这类系统的设计和一般计算机系统没有太大的区别，一般不需要用嵌入式系统实现。

实时控制系统，在这类系统中计算机通过特定的设备（器件）与被控对象联系，实时接收（采集）系统对象的信息。经处理后产生控制指令，实现对被控对象的控制。有时，人作为系统中的一个环节，从系统的显示或其他设备获得系统信息，并通过人机接口设备实现人工干预，改变系统的运行，（有的系统可能没有用户界面，完全是个黑匣子）。这类系统一般属于硬实时系统。

大部分硬实时系统是嵌入式系统（embedded system）。

1.1.3 嵌入式实时系统的特点

■ 嵌入式计算机系统特征

嵌入式计算机系统具有区别于通用计算机系统的一些特征，主要有：



➤ 专用的嵌入式 CPU

嵌入式 CPU 与通用型的最大不同就是嵌入式 CPU 大多工作在为特定用户群设计的系统中，它通常都具有低功耗、体积小、集成度高等特点，能够把通用 CPU 中许多由板卡完成的任务集成在芯片内部，从而有利于嵌入式系统设计趋于小型化，移动能力大大增强，与网络的耦合也越来越紧密。

➤ 专用性和算法的唯一性

它总是被设计成为完成某一特定任务，一旦设计完成就不再改变。嵌入式系统和具体应用有机地结合在一起，它的升级换代也和具体产品同步进行，因此嵌入式系统产品一旦进入市场，具有较长的生命周期。

➤ 多种技术的结合体

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

➤ 硬件与软件的互相依赖性

由于它的专用性决定了它的设计目标是单一的，硬件与软件的依赖性强，因而一般软件要进行共同设计 (Co-design)，以求达到共同完成预定的功能，并满足性能、成本和可靠性目标。嵌入式系统的硬件和软件都必须高效率地设计，量体裁衣、去除冗余，力争在同样的硅片面积上实现更高的性能，这样才能在具体应用对处理器的选择面前更具有竞争力。

➤ 系统对用户是透明的

用户在使用这种设备时只是按照预定的方式使用它，既不需要用户进行编程，也不需要用户知道设备内计算机系统的设计细节，用户也不能改变它。

➤ 嵌入式计算机系统大多数是实时控制系统

例如工业仪器、控制装置、数控系统、信息家电、军用设备和控制系统等。

➤ 系统配置专一，结构紧凑，坚固可靠，一般说来计算机资源（存储容量和速度）有限

这是由专用性、嵌入式（空间约束）以及适用环境所决定。

➤ 许多嵌入式计算机系统采用分布式系统实现

在各处理机之间存在通信链接，因为分布式系统更易于保证硬实时性要求、更便宜、更易于实现。



■ 嵌入式计算机系统软件特征

正由于嵌入式计算机系统具有上述特点，嵌入式计算机系统的软件则是更具有特色的软件。对于嵌入式实时系统来说，它具有如下特点：

➤ 响应时间快，并且有确定的硬实时性要求

一般说来，嵌入式系统软件对外部事件的反应是快速的、确定的、可重复实现或周期性的，不管系统当时的内部状态如何都是可以预测的。同时对于事件的处理往往要求在 deadline 到来之前完成，否则将可能引起系统的崩溃。

➤ 具有处理异步并发事件的能力

在实际环境中，嵌入式系统大多数是事件驱动的系统，而且处理的外部事件是多发的而且是并发的随机事件，也就是异步事件。嵌入式应用软件应能有效地处理这些并发事件。所以往往采用多进程（多任务）运行机制，以适应这种复杂的并发环境。还采用线程（thread）和轻进程（lightweight process），以获得更快的切换速度。嵌入式实时操作系统一般提供多任务或多处理机制来管理资源和任务切换。

➤ 具有快速启动、出错处理和自动复位功能

要求快速启动是对嵌入式实时系统的普遍要求，因而也不允许控制程序在运行前从磁盘上加载。所以，嵌入式系统程序大都放置在快速只读存储器中并可直接执行，因而程序是绝对定位、可再入的。并且应用程序应采用特殊的容错和出错处理措施，具有故障诊断和修复能力，在运行死机之前自动恢复先前的运行状态。

➤ 嵌入式系统软件的应用软件与操作系统之间的界线模糊，往往是一体化设计的程序

在通用计算机系统中，像操作系统等系统软件与应用软件之间的界线分明，应用软件是独立设计、独立运行的。但是，嵌入式系统中，操作系统与应用软件是一体化设计的，也就是说应用软件与操作系统是为特定的应用而设计的。嵌入式系统的配置不同，其操作系统和应用软件的配置也不同。所以嵌入式系统的设计往往是硬件与软件一起进行设计。

➤ 软件开发困难，要使用交叉开发环境

嵌入式系统的特点使得其软件受到时间和空间的严格限制，加上运行环境复杂，使得嵌入式系统软件的开发变得异常困难。为了设计一个满足功能、性能和 deadline 要求的代码并把它写进给定数量、位置的 ROM 中是困难的。它需要特别了解专门设计方法的人员，也需要在专门的开发平台上进行交叉开发，开发环境与运行环境不同。开发平台叫宿主系统，而嵌入式系统的运行系统叫目标系统。嵌入式系统的软件交叉开发环境对开发安全可靠、高性能和



复杂的嵌入式系统起着非常重要的作用。

1.1.4 嵌入方式

从 20 世纪 60 年代已经开始应用嵌入式系统，随着计算机技术的发展而迅速发展，嵌入方式有以下三种：

➤ 整机式嵌入

一个带有专用接口的计算机系统嵌入到一个控制系统中，成为控制系统的核心部分。一般这种计算机系统功能完整而强大，完成系统中的核心的关键的工作，也具有较完善的人机界面和外部设备。如指火控系统中多属于这一类。

➤ 部件式嵌入

以部件式嵌入到一个控制设备中，完成某一处理功能，与设备的其他硬件耦合更紧，功能更专一。如雷达的数字信号处理部件，一般选用专用 CPU 或 DSP。

➤ 芯片式嵌入

一个芯片是一个完整的专用计算机，具有完整的输入/输出接口，完成专一的功能。如显示处理机、微波炉控制器等。一般为专门设计的芯片，随着微电子技术的发展，芯片式嵌入应用将越来越广泛。

1.1.5 嵌入式系统的分类及应用

根据不同的分类标准，嵌入式系统有不同的分类方法，这里根据嵌入式系统的复杂程度，可以将嵌入式系统分为以下四类：

➤ 单个微处理器

这类系统可以在小型设备中（如温度传感器、烟雾和气体探测器及断路器）找到，是供应商根据设备的用途来设计的，受 Y2K 影响的可能性不大。

➤ 不带计时功能的微处理器装置

这类系统可在过程控制器、信号放大器、位置传感器及阀门传动器等中找到。这类设备也不太可能受到 Y2K 的影响。但是，如果它依赖于一个内部操作时钟，那么这个时钟可能受 Y2K 问题的影响。

➤ 带计时功能的组件

这类系统可见于开关装置、控制器、电话交换机、电梯、数据采集系统、医药监视系统、



诊断及实时控制系统等。它们是一个大系统的局部组件，由它们的传感器收集数据并传递给该系统。这种组体可同 PC 机一起操作，并可包括某种数据库（如事件数据库）。

➤ 分布式嵌入式系统

在许多嵌入式应用系统中，往往包含许多设备，这时分布式系统就是实现这种系统最方便、最现实的方法。因为：

- (1) 时间关键的任务放在不同 CPU 中可以更容易保证满足它的 deadline 要求。
- (2) 把微处理器放在设备级上更便于实现设备之间的接口。
- (3) 如果系统中包含从供应商购买的几个设备或系统，它们也包含有自己的 CPU，或者还包含有通信接口，通常不可能把系统的任务放到这些设备中，或者相反把设备的任务放到系统中。
- (4) 使用几个小 CPU 比使用一个大 CPU 更便宜。

所以，许多嵌入式系统用分布式系统实现，在分布的处理机之间用通信链路连接起来。通信链路可以是高速并行数据总线（紧耦合型）也可以是串行数据链路。

制造或过程控制中使用的计算机系统多属于这类系统。对于这类系统，计算机与仪器、机械及设备相连来控制这些装置的工作。这类系统包括自动仓储系统和自动发货系统。在这些系统中，计算机用于总体控制和监视，而不是对单个设备直接控制。过程控制系统可与业务系统连接（如根据销售额和库存量来决定定单或产品量）。在许多情况下，两个功能独立的子系统可在一个主系统操作下一同运行。如控制系统和安全系统，控制子系统控制处理过程以使系统中的不同设备能正确的操作和相互作用以生产产品；而安全子系统则用来降低那些会影响人身安全或危害环境的误操作风险。

1.2 嵌入式实时系统软件开发设计

1.2.1 嵌入式实时系统开发过程

嵌入式实时系统开发的过程一般如图 1.4 所示。

从图 1.4 中可看出，嵌入式实时系统的开发实际上是软硬件交叉并行设计的过程。但是一旦系统的体系结构设计完成，软硬件设计就可独立并行地进行了。等待两者的设计完成，再集成一体进行集成测试。

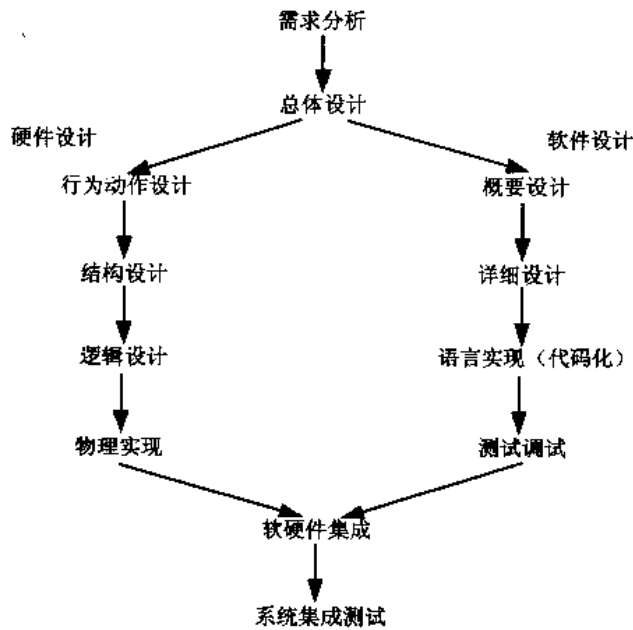


图 1.4 嵌入式系统的从顶向下的开发流程

1.2.2 嵌入式实时软件开发过程

嵌入式实时软件的开发过程如图 1.5 所示。

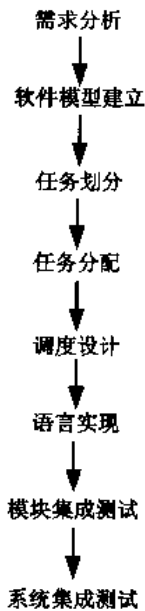


图 1.5 嵌入式实时软件开发流程



软件模型建立和任务划分相当于一般软件设计过程中的概要设计阶段。模型建立应能给系统建立一个并发模型。任务划分则是在这个并发模型基础上，按照一定的任务构造准则，把系统分解成功能合理和数目合适的任务集合。

任务分配是把任务按照一些规则和约束，放入到相应的计算机或处理机中。任务的执行顺序由任务调度来决定。

语言实现阶段就是用某一计算机语言实现任务划分阶段得到的每个模块。

在每个阶段的结束要进行相应的性能分析，也就是评估每个阶段的设计是否满足性能要求，达到一定的标准。它既是对本阶段成果的肯定，又是下一阶段得以顺利进行的保证。

一旦软件的各个部分已经实现，就应该进入测试阶段。软件的测试可分为单元测试和集成测试，前者应在编码阶段进行；后者是把已完成测试的软件单元组装成一个完整的系统进行测试。

1.2.3 嵌入式实时软件开发模型

由于嵌入式系统自身的特殊性，受限于嵌入式系统自身资源与空间的限制，嵌入式实时软件开发一般采用主机—目标机模型，如图 1.6 所示。



图 1.6 主机-目标机开发模型

嵌入式实时开发环境运行在开发主机上。开发主机可以是工作站、PC 机等，运行的操作系统多为 Unix、Microsoft Windows 等工具丰富界面友好的系统平台。目标机是我们开发的嵌入式硬件平台。主机和目标机通过串行口、以太网、仿真器或其他通信手段相联系。用户所有的开发工作在主机开发环境下完成，包括编码、编译、联接、下载和调试等工作。生成目标码由串行口或以太网口或其他通信途径下载到目标机，应用程序在目标机上执行，用户可以使用寄于主机的开发环境提供的调试工具调试运行在目标机上的应用程序。

这种开发环境除能够开发出处理器的全部功能以外，还应当是用户友好的。



1.2.4 嵌入式系统开发工具

目前市场上嵌入式系统的开发工具平台主要包括下面几类:

- ◆ 实时在线仿真系统 ICE (In-Circuit Emulator)。
- ◆ 高级语言编译器 (Compiler Tools)。
- ◆ 源程序模拟器 (Simulator)。
- ◆ 实时多任务操作系统。
- ◆ 集成开发环境。

■ 实时在线仿真系统 ICE

传统上,开发嵌入式系统的首选工具是仿真器。这是一块比较昂贵的设备,一般插于微处理器和它的总线之间的电路中,从而让开发者监视和控制所有输入和输出——微处理器的各种活动和行为。在装配方面,可能有一些困难,并且由于它们的侵入性,装上后可能造成性能不稳定。尽管这样,它们却能在总线级上给出一个系统正在发生什么的清晰的描绘,并排除了很多在硬件和软件接口最底层上的猜测工作。

实时在线仿真系统(ICE)仍是进行嵌入式应用系统调试最有效的开发工具。

ICE 首先可以通过实际执行,对应用程序进行原理性检验,排除以人的思维难以发现的设计逻辑错误。ICE 的另一个主要功能是在应用系统中仿真微控制器的实时执行,发现和排除由于硬件干扰等引起的异常执行行为。此外,高级的 ICE 带有完善的跟踪功能,可以将应用系统的实际状态变化、微控制器对状态变化的反应,以及应用系统对控制的响应等以一种录像的方式连续记录下来,以供分析,在分析中优化控制过程。很多机电系统难以建立一个精确有效的数学模型,或是建立模型需要大量人力,这时采用 ICE 的跟踪功能对系统进行记录和分析是一个快而有效的方法。

嵌入式应用的特点是和现实世界中的硬件系统有关,存在各种异变和事先未知的变化,这就给微控制器的指令执行带来了各种不确定性,这种不确定性只有通过 ICE 的实时在线仿真才能发现,特别是在分析可靠性时要在同样条件下多次仿真,以发现偶然出现的错误。

ICE 不仅是软件硬件除错工具,同时也是提高和优化系统性能指标的工具。高档 ICE 工具是可根据用户投资裁剪功能的系统,亦可根据需要选择配置各种档次的实时逻辑跟踪器(Trace)、实时映像存储器(Shadow RAM)及程序效率实时分析功能(PPA)。

■ 高级语言编译器 (Compiler Tools)

早期的嵌入式系统软件多是使用汇编语言编写的,并且仅能用于为其编写的微处理器上。当这些微处理器变得过时的时候,它们使用的软件也厄运同临,只能在新的处理器上重新写一遍才能运行。今天,许多这种早期的系统只不过成了人们模糊的记忆。当 C 语言出现



后，软件可以用一种高效的，稳定的和可移植的方式来编写。

C 语言作为一种通用的高级语言，大幅度提高了嵌入式系统工程师的工作效率，使之能够充分发挥出嵌入式处理器日益提高的性能，缩短产品进入市场时间。另外 C 语言便于移植和修改，使产品的升级和继承更迅速。更重要的是采用 C 语言编写的程序易于在不同的开发者之间进行交流，从而促进了嵌入式系统开发的产业化。软件的可复用性已经为人接受而且正在很好地发挥作用。

区别于一般计算机中的 C 语言编译器，嵌入式系统中的 C 语言编译器要专门进行优化，以提高编译效率。优秀的嵌入式系统 C 编译器编译的代码长度和执行时间仅比以汇编语言编写的同样功能程序长 5%~20%。编译质量的不同，是区别嵌入式 C 编译器工具的重要指标。而 C 编译器与汇编语言工具相比残余的 5%~20% 效率差别，完全可以由现代微控制器的高速度、大存储器空间以及产品提前进入市场的优势来弥补。

C/C++/EC++ 引入嵌入式系统，使得嵌入式开发和个人计算机、小型机等之间在开发上的差别正在逐渐消除，软件工程中的很多经验、方法乃至库函数可以移植到嵌入式系统。在嵌入式开发中采用高级语言，还使得硬件开发和软件开发可以分工，从事嵌入式软件开发不再必须精通系统硬件和相应用汇编语言指令集。

另一种高级语言，Java 的发展则具有戏剧性。Java 本来是为设备独立的嵌入式系统设计的，为了提高程序继承性的语言，但是目前基于 Java 的嵌入式开发工具代码生成长度要比嵌入式 C 编译工具差 10 倍以上。因此 EC++ 很可能将成为未来的主流工具。

■ 源程序模拟器 (Simulator)

源程序模拟器是在广泛使用的、人机接口完备的工作平台上，如小型机和 PC，通过软件手段模拟执行为某种嵌入式处理器内核编写的源程序测试工具。简单的模拟器可以通过指令解释方式逐条执行源程序，分配虚拟存储空间和外设，供程序员检查；高级的模拟器可以利用计算机的外部接口模拟出处理器的 I/O 电气信号。不同档次和功能模拟器工具价格差距巨大。

模拟器软件独立于处理器硬件，一般与编译器集成在同一个环境中，是一种有效的源程序检验和测试工具。但值得注意的是，模拟器毕竟是以一种处理器模拟另一种处理器的运行，在指令执行时间、中断响应、定时器等方面很可能与实际处理器有相当的差别。另外它无法和 ICE 一样，仿真嵌入式系统在实际应用系统中的实际执行情况。

■ 实时多任务操作系统 (RTOS)

实时多任务操作系统 (RTOS) 是嵌入式应用软件的基础和开发平台。

许多嵌入式系统软件开发还是基于处理器直接编写，根本就没有操作系统，只不过有一个控制环而已。这种现状在国内尤为突出。对很简单的嵌入式系统来说，这可能已经足够。



不过，随着嵌入式系统在复杂性上的增长，一个操作系统显得重要起来。因为否则的话，将使（控制）软件复杂度变得极不合理。可悲的是，现实中确实有一些复杂得另人生畏的嵌入式系统，而且它们之所以变得复杂，就因为它们的设计者坚持认为它们的系统不需要操作系统，从而系统软件和应用软件混在一起处理。

随着应用复杂度的增高，嵌入式系统要管理的资源越来越多，如存储器、外设、网络栈等。这些资源添加到一个仅用控制环来实现的简单嵌入式系统所带来的复杂程度可能足以唤起人们对一个操作系统的渴望。

有关是否使用 RTOS 的争论非常类似于是否使用高级语言的争论。正像高级语言一样，RTOS 使你可以更快地开发产品。它可能要求一些额外的开销，但是随着技术的进步，这种开销在变小。

RTOS 是针对不同处理器优化设计的高效率实时多任务内核，优秀商品化的 RTOS 可以面对几十个系列的嵌入式处理器 MPU、MCU、DSP 等提供类同的 API 接口，这是 RTOS 基于设备独立的应用程序开发基础。因此，基于 RTOS 上的 C 语言程序具有极大的可移植性。据专家测算，优秀 RTOS 上跨处理器平台的程序移植只需要修改 1%~5% 的内容。

在 RTOS 基础上可以编写出各种硬件驱动程序、专家库函数、行业库函数、产品库函数，和通用性的应用程序一起，可以作为产品销售，促进行业内的知识产权交流。因此 RTOS 又是一个软件开发平台。

■ 集成开发环境

由于嵌入式应用系统的软件开发受时间和空间的限制，一般通过交叉开发来实现，因而需要专门的集成开发环境，使用集成开发环境可以完成软件开发的整个过程。它应集成必要的编辑、编译、链接、下载、跟踪和调试，完善的还可能包含软件仿真器、版本管理等工具，并应具有友好的交互式用户操纵界面。

1.3 嵌入式实时操作系统

1.3.1 实时操作系统有关概念

■ 什么是实时操作系统

操作系统是计算机系统中的一个系统软件，是一些程序模块的集合。它们能以尽量有效合理的方式组织和管理计算机的软硬件资源，合理地组织计算机的工作流程，控制程序的执



行并向用户提供各种服务功能，使用户能够灵活、方便、有效地使用计算机，使整个计算机系统能高效地运行。

没有操作系统，任何应用软件都无法运行。操作系统实际上是一个计算机系统中硬、软件资源的总指挥部。

操作系统是软件技术含量最大、附加值最高的部分，是软件技术的核心，是应用软件的运行平台。

实时多任务操作系统（RTOS）是嵌入式应用软件的基础和开发平台。目前大多数嵌入式开发还是在处理器上直接进行，没有 RTOS，但仍要有一个主控程序负责调度各个任务。RTOS 可以简单地认为是功能强大的主控程序，它嵌入在目标代码中，系统复位后首先执行，它负责在硬件基础之上，为应用软件建立一个功能更为强大的运行环境，用户的其他应用程序都建立在 RTOS 之上，从这个意义上而言，操作系统的作用是为用户提供一台等价的扩展计算机，可以认为是一个虚拟机，它比底层硬件更容易编程。不仅如此，RTOS 还是一个标准的内核，将 CPU 时间、中断、I/O、定时器等资源都包装起来，留给用户一个标准的 API，并根据各个任务的优先级，合理地在不同任务之间分配 CPU 时间，从这个意义上而言，操作系统的作用是资源管理器。

实时操作系统不同于分时操作系统，它们有明显的区别。具体的说，对于分时操作系统，软件的执行在时间上的要求并不严格，时间上的错误，一般不会造成灾难性的后果。而对于实时操作系统，主要任务是对事件进行实时的处理，虽然事件可能在无法预知的时刻到达，但是软件上必须在事件发生时能够在严格的时限内作出响应（系统响应时间），即使是在尖峰负荷下，也应如此，系统时间响应的超时就意味着致命的失败。另外，实时操作系统的重要特点是具有系统的可确定性，即系统能对运行情况的最佳和最坏等情况做出精确的估计。

实时系统虽然没有统一的定义，但是所有的实时系统都要求满足时限。系统有同时发生或同时存在的进程，具有内在的并发性。可以分为硬实时系统和软实时系统，参考图 1.2、图 1.3。但是不存在硬实时操作系统或软实时操作系统！

一个实时操作系统仅能允许我们开发一个硬实时系统，但这样的 RTOS 并不能保证系统所有的时限都能得到满足！系统实现的满足还需要软件设计合理。

一个好的实时操作系统需要具备以下功能（必须的但是不充分的）：

- 多任务和可抢占的。
- 任务具有优先级。
- 操作系统具备支持可预测的任务同步机制。
- 支持多任务间的通信。
- 操作系统具备消除优先级转置的机制。
- 存储器优化管理（含 ROM 的管理）。
- 操作系统的（中断延迟、任务切换、驱动程序延迟等）行为是可知的和可预测的，



这是指在全负载的情形下，最坏反应时间可知。

- ◆ 实时时钟服务。
- ◆ 中断管理服务。

RTOS 最关键的部分是实时多任务内核，它的基本功能包括任务管理、定时器管理、存储器管理、资源管理、事件管理、系统管理、消息管理、队列管理、信号量管理等，这些管理功能是通过内核服务函数形式交给用户调用的，也就是 RTOS 的 API。

RTOS 的引入，解决了嵌入式软件开发标准化的难题。随着嵌入式系统中软件比重不断上升、应用程序越来越大，对开发人员、应用程序接口、程序档案的组织管理成为一个大的课题。引入 RTOS 相当于引入了一种新的管理模式，对于开发单位和开发人员都是一个提高。基于 RTOS 开发出的程序，具有较高的可移植性，实现 90% 以上的设备独立，一些成熟的通用程序可以作为专家库函数产品推向社会。

■ 实时操作系统中的重要概念

➤ 系统响应时间 (System response time)

系统发出处理要求到系统给出应答信号的时间。

➤ 任务切换时间 (Context-switching time)

任务之间切换而使用的时间。

➤ 中断延迟 (Interrupt latency)

是计算机接收到中断信号到操作系统作出响应，并完成切换转入中断服务程序的时间。

➤ 任务

实时操作系统中的任务 (Task) 等同于分时操作系统中的进程 (Process) 的概念。系统中的任务一般有四种状态：运行 (Executing)、就绪 (Ready)、挂起 (Suspended)、睡眠 (Dormant)。

- ◆ 运行：获得 CPU 控制权。
- ◆ 就绪：进入任务等待队列。通过调度转为运行状态。
- ◆ 挂起：任务发生阻塞，移出任务等待队列，等待系统实时事件的发生而唤醒。从而转为就绪或运行。
- ◆ 睡眠：任务完成或错误等原因被清除的任务。也可以认为是系统中不存在了的任务。单 CPU 系统中只能有一个任务在运行状态。各任务按级别（一般是优先级）通过时间片分别获得对 CPU 的访问权。



1.3.2 实时操作系统发展历史

实时操作系统 (RTOS) 的研究是从 20 世纪 60 年代开始的, 差不多是随嵌入式系统同时出现的。

从系统结构上看, RTOS 到现在已经历了如下三个阶段。

➤ 早期的实时操作系统

早期的实时操作系统, 还不能称为真正的 RTOS, 它只是小而简单的、带有一定专用性的软件, 功能较弱, 可以认为是一种实时监控程序。一般为用户提供对系统的初始化管理以及简单的实时时钟管理, 有的实时监控程序也引入了任务调度及简单的任务间协调等功能。这个时期, 实时应用较简单, 实时性要求也不高。应用程序、实时监控程序和硬件运行平台往往是紧密联系在一起。

这时期的开发工具也很简单, 主要用于创建和调试软件。各工程项目的运行软件通常以信手涂鸦的方式编出来。由于编译器经常有很多错误而且也缺乏像样的调试器, 这些软件差不多总是用汇编语言或宏语言来编写。

➤ 专用实时操作系统

随着应用的发展, 早期的 RTOS (实时监控程序) 已越来越显示出明显的不足了。有些实时系统的开发者为了满足实时应用的需要, 自己研制与特定硬件相匹配的实时操作系统。这类专用实时操作系统在国外称为 Real-Time Operating System Developed in House。

它们中的许多是用汇编语言实现的, 并且仅能用于为其编写的微处理器上。当这些微处理器变得过时的时候, 它们使用的 OS 也面临着相同的厄运, 只能在新的处理器上重新写一遍才能运行。它是在早期用户为满足自身开发需要而研制的, 它一般只能适用于特定的硬件环境, 且缺乏严格的评测, 移植性也不太好。属于这类实时操作系统的有 Intel 公司的 iRMX86 等。

➤ 通用实时操作系统

在各种专用 RTOS 中, 一些多任务的机制如基于优先级的调度、实时时钟管理、任务间的通信、同步互斥机构等基本上是相同的, 不同的只是面向各自的硬件环境与应用目标。实际上, 相同的多任务机制是能够共享的, 因而可以把这部分很好地组织起来, 形成一个通用的实时操作相同内核。

当 C 语言出现后, OS 可以用一种高效的, 稳定的和可移植的方式来编写。这种方式对使用和经营有直接的吸引力, 因为它承载着人们当微处理器废弃不用时能保护他们的软件投资的希望。用 C 来编写 OS 已经成了一种标准直至今日。总之, 软件的可复用性已经为人接受而且正在很好地发挥作用。



在 20 世纪 80 年代早期的 Wendon 操作系统是一个开发套件，人们可以通过选择一些组件来构建自己的 OS，——整个过程就像是从中餐菜单里订餐一样。比如，可以从库中的多个可行选项列表中精选出一种任务调度算法和内存管理方案。

这类实时操作系统大多采用软组件结构，以一个个软件“标准组件”构成通用的实时操作系统，一方面，在 RTOS 内核的最底层将不同的硬件特性屏蔽掉；另一方面，对不同的应用环境提供了标准的、可剪裁的系统服务软组件。这使得用户可根据不同的实时应用要求及硬件环境选择不同的软组件，也使得实时操作系统开发商在开发过程中减少了重复性工作。

许多用于嵌入式系统的商业操作系统在 20 世纪 80 年代获得了蓬勃发展。Wendon，这一原始的炖菜已经发展成为了商业操作系统这一现代炖肉。今天已经有许多的商业性操作系统可供选择，出现了许多互相竞争的产品，如 Integrated System 公司（已被 WindRiver 收购）的 pSOS、Intel 公司的 iRMX386、Ready System 公司（后与 Microtec Research 合并）的 VRTX32、WindRiver System 提供的 VxWorks 等。它们一般都提供了实时性较好的内核、多种任务通信机制、基于 TCP/IP 的网络组件、文件管理及 I/O 服务，提供了集编辑、编译、调试、仿真为一体的集成开发环境，支持用户使用 C、C++ 进行应用程序的开发，源代码级调试。

1.3.3 实时操作系统发展展望

实时操作系统经过多年的发展，先后从实模式进化到保护模式，从微内核技术进化到超微内核技术，在系统规模上也从单处理器的 RTOS 发展到支持多处理器的 RTOS 和网络 RTOS，在操作系统研究领域形成了一个重要分支。

今后，RTOS 研究方向主要集中在如下几个方面：

➤ RTOS 的标准化研究

如今国外的 RTOS 开发商有数十家，提供了上百个 RTOS，它们各具特色。但这也给应用开发者带来难题，首先是应用代码的重用性难，当选择不同的 RTOS 开发时，不能保护用户已有的软件投资，RTOS 的标准化研究越来越被重视。美国 IEEE 协会在 Unix 的基础上，制定了实时 UNIX 系统的标准 POSIX 1001.4 系列协议，但仍有许多工作还待完成。

➤ 多处理器结构 RTOS、分布式实时操作系统和实时网络的研究

实时应用的飞速发展，对 RTOS 的性能提出了更高的要求。单处理器的计算机系统已不能很好地满足某些复杂实时应用系统的需要，开发支持多处理器结构的 RTOS 已成为发展方向，这方面比较成功的系统有 pSOS 等。至于分布式 RTOS，国外一些 RTOS 厂家虽已推出部分产品，如 QNX、Chorus、Plan 9 等，但分布式实时操作系统的研究还未完全成熟，特别是在网络实时性和多处理器间任务调度算法上还需进一步研究。



➤ 集成的开放式实时系统开发环境的研究

RTOS 研究的另一个重要方向是集成开发环境的研究。开发实时应用系统，只有 RTOS 是不够的，需要集编辑、编译、调试、模拟仿真等功能为一体的集成开发环境的支持。开发环境的研究还包括网络上多主机间协作开发与调试应用技术的研究、RTOS 与环境的无缝连接技术等。

1.3.4 RTOS 的评价指标

RTOS 是操作系统研究的一个重要分支，它与一般商用多任务 OS 如 Unix、Windows 等有共同的一面，也有不同的一面。对于商用多任务 OS，其目的是方便用户管理计算机资源，追求系统资源最大利用率和公平对待所有的系统请求，一般称之为通用操作系统（GOS）；而 RTOS 追求的是实时性、可确定性、可靠性。评价一个实时操作系统一般可以从任务调度、内存管理、任务通信、内存开销、任务切换时间、最大中断禁止时间等几个方面来衡量。

➤ 任务调度机制

RTOS 的实时性和多任务能力在很大程度上取决于它的任务调度机制。从调度策略上来讲，分优先级调度策略和时间片轮转调度策略；从调度方式上来讲，分可抢占、不可抢占、选择可抢占调度方式；从时间片来看，分固定与可变时间片轮转。单纯从基于优先级的抢占式调度方式而言，又存在多种优先级计算算法。

➤ 内存管理

分实模式与保护模式，主要对 Intel x86 而言。

➤ 最小内存开销

RTOS 的设计过程中，最小内存开销是一个较重要的指标，这是因为实时系统，特别是包括消费类电子产品在内的嵌入式系统中，由于基于降低成本的考虑，其内存的配置一般都不大，而在这有限的空间内不仅要装载实时操作系统，还要装载用户程序。因此，在 RTOS 的设计中，其占用内存大小是一个很重要的指标，这是 RTOS 设计与其他操作系统设计的明显区别之一。

➤ 最大中断禁止时间

当 RTOS 运行在核态或执行某些系统调用的时候，是不会因为外部中断的到来而中断执行的。只有当 RTOS 重新回到用户状态时才响应外部中断请求，这一过程所需的最大时间就是最大中断禁止时间。



➤ 任务切换时间

当由于某种原因使一个任务退出运行时，RTOS 保存它的运行现场信息、插入相应队列、并依据一定的调度算法重新选择一个任务使之投入运行，这一过程所需时间称为任务切换时间。

上述几项中，最大中断禁止时间和任务切换时间是评价一个 RTOS 实时性最重要的两个技术指标。

1.4 如何选择实时操作系统

1.4.1 市场现状

在所有嵌入式系统开发商中，在其产品中仍有一半在使用自己写的实时操作系统，也就是称之为 in-house 的 RTOS。这可能是有多方面的原因：费用约束、产品特殊的技术要求或公司的政策等等。当然，也可能应用过于简单，仅需求很少的一点操作系统服务。在这种情况下，更简单的结构，比如一个主控程序可能就足够了。

另外的一半则是在使用商用嵌入式实时操作系统。目前市场上的嵌入式实时操作系统提供厂商超过 100 多家。其中比较出色的有：QNX、Lynx、Concurrent、VxWorks、Microware、Vrtx、pSOS、eCos 等。面对如此众多的商用产品，如何做出选择就成为一项很复杂的工作，需要依据一些准则。

1.4.2 选择准则

首先整理调查研究 RTOS 期间所搜集的信息，列一份可供选择的 RTOS 清单，作出选择就会容易一些。

➤ 是否支持目标硬件平台

在选择 RTOS 时，项目可能已经选定了微处理器。据此可以立即划掉不支持该 MCU 的 RTOS 从而得到较短的清单。如果选择的是无所不在的 68000 或者 x86 系列，则需要更多的准则来帮助你作出选择。

➤ 与其他开发工具能否相互关系

一个工程师选择实时操作系统时，如果不考虑其余与之相关的工具是不行的。微处理器、在线仿真器 (ICE)、编译器、汇编器、链接器、调试器以及模拟器，这些工具都这样或那样



地影响着操作系统。

有些在线仿真器供应商提供其 ICE 与实时操作系统接口的软件。因此需要考虑 ICE 是否能与所选的 RTOS 协同工作，这在调试那些最隐蔽的小错误 (bugs) 时是很有用的。然而，重要的是要了解 RTOS 对在线仿真器的操作对性能的影响。有时当 ICE 执行操作时增加了额外的开销，比如中断某行源代码在某个任务中的执行。

对给定微处理器家族上的某种操作系统来说，很可能 OS 供应商只支持所有可用编译工具 (包括编译器、汇编器和链接器) 的一个子集。应该确认供应商支持所用的编译工具。否则就可能增加额外的工作。

➤ 能否满足应用的关键要求

每一个应用开发都有差异，要求满足不同的性能指标，所选的 RTOS 必须满足这些特定要求。以下是选择时要考虑的几条重要指标。

- 操作系统本身的性能。包括：支持的优先级数、调度机制、多任务通信机制、保证应用功能的前提下内核最小可裁减大小、中断响应时间、上下文交换时间等。这些性能要以第三方测试为依据。
- 网络栈、设备和驱动程序支持、图形开发包和软件组件。在 1998 年 11 月的嵌入式系统会议上，Wind River Systems 的合伙创始人之一 Jerry Fiddler 描绘了将来 10 年嵌入式系统的图景：网络化的、无所不在的普通设备；到处都会有计算机，但计算机的外表不再是一成不变的。为了使美景成真，嵌入式系统应该通过各种标准加大开发需求的互操作性，开发者可能要依赖于他人开发的组件。假如你的应用需要通信协议、服务、库或者其他组件 (如 TCP/IP、HTTP、ftp、telnet、SNMP、CORBA 和图形)，先看看哪里可以获得它们。类似的，在设计中用到现成的板卡或 IC 时，要确定是否可以得到设备驱动程序。有些操作系统供应商提供这些特性或驱动程序的方式不同，可能作为操作系统的一部分，也可能作为可选配件。另外，这些服务也可以从第三方供应商获得。与供应商交涉时，要弄清楚你将要购买的 RTOS 里集成了哪些组件。
- 开发工具。包括编辑、编译、联接、调试工具，编程语言支持，是否支持动态下载，调试途径，网络、串口调试方式支持 (这非常重要)，操作系统内核配置工具等。
- 第三方工具支持程度。
- 标准兼容性。这将影响应用的可移植性。进而影响到开发周期和成本。另外还要考虑应用行业所要求的安全标准。有些 RTOS 供应商已经开始认证他们的产品。
- 价格、技术支持和声誉，这非常重要。



1.4.3 VxWorks 操作系统

VxWorks 是专门为实时嵌入式系统设计开发的操作系统软件,为程序员提供了高效的实时任务调度、中断管理、实时的系统资源以及实时的任务间通信。应用程序员可以将尽可能多的精力放在应用程序本身,而不必再去关心系统资源的管理。

VxWorks 从 1983 年设计成功以来,已经经过广泛的验证,已成功的应用在航空、航天、舰船、通信、医疗等关键领域。目前,VxWorks 得到了许多软硬件厂家的支持,这些第三方软硬件厂家提供丰富的 VxWorks 的扩展组件。因此,从应用软件角度而言,VxWorks 操作系统在各种 CPU 硬件平台上可以提供统一的接口和一致的运行特性,应用程序无需做过多的改动就可以运行在各种 CPU 上,为程序员提供了一致的开发、运行环境,减少了重复劳动。

嵌入式软件开发是一种比较复杂的劳动,操作系统性能再好,仅仅依靠人工编程调试,很难发挥它的功能,要设计出可靠、高效的嵌入式系统,必须要有与之相适应的开发工具。Tornado 就是为开发 VxWorks 应用系统提供的集成开发环境。Tornado 中包含的工程管理软件,可以将用户自己的代码与 VxWorks 的核心有效的组合起来;原型仿真器可以让程序员不用目标机的情况下,直接开发系统原型,作出系统评估;功能强大的调试器可以提供任务级和系统级的调试模式,可以进行多目标机的联调;优化分析工具可以帮助程序员从多种方式真正地观察、跟踪系统的运行,排除错误,优化性能。

1.5 本书的组织

正如前面指出的,嵌入式实时系统软件的设计核心是多任务设计、中断处理和网络编程,VxWorks 操作系统是目前在国内外风行的嵌入式强实时操作系统。本书将主要介绍 VxWorks 操作系统程序设计中多任务设计、中断处理和网络编程这三方面的内容。组织如下:

第一章简单介绍了嵌入式实时系统软件的基本知识。

第二章主要介绍 VxWorks 操作系统和 Tornado 开发环境的主要特征和能力。

第三章和第七章是一个 Tornado 集成开发环境的简单教程:软件安装,主机文件组织,project、shell、debug、browser 等工具的使用,主机目标机环境的建立,Tornado1.0 工程 makefile 文件的编写等内容。

第四、五、六、八、九章,系统讲解了 VxWorks 操作系统程序设计中多任务设计、任务间通信、中断处理和网络编程这几方面的内容,包括编程接口的使用、编程要注意的问题、代码实例等。



第十章介绍了 VxWorks 操作系统启动过程、几个主要文件介绍、操作系统配置选项含义、以及如何配置裁减操作系统。

第十一章是编程实验。

嵌入式实时应用是目前国内蓬勃发展的行业之一。由于其与一般计算机系统的差异，其开发系统与通用软件开发有着明显的不同，需要高实时性能的操作系统和开发环境。

WRS 是国际著名的嵌入式实时操作系统的供应商。其产品 VxWorks 操作系统自 1996 年登陆中国，短短几年来，逐渐进入了国内通信、国防、工业控制、医疗设备等嵌入式实时应用领域，在国内拥有了较多的用户。特别是最近两年，VxWorks 操作系统越来越多地占据了国内嵌入式实时应用市场。

这一章主要介绍实时应用的基本需求与 VxWorks 操作系统及其开发环境 Tornado 的特点与能力是如何满足这些要求的。

2.1 实时应用的基本需求

2.1.1 实时应用的基本特征

实时应用在航空航天、广播、运输、医学仪器、自动化生产和科学研究等领域得到了广泛应用。

应用范围包括：金融交易、飞行控制与仿真、武器控制、图形显示与图像处理、机械控制、医学成像、测试与研究、机器人技术、地震数据收集、交通与铁路系统。

除了面向设备和快速的反应能力以及对异步事件的可预测性之外，实时应用一般具备如表 2-1 所示特征。



表 2.1 实时应用的特征

特 征	描 述
专用资源	一个专用的实时应用系统使用一个或多个计算机来解决一个特定的问题或相关问题的集合。它将系统所有的资源用于该实时应用；不提供分时的服务。这种应用经常称为硬实时应用。
分布式处理	一个分布式实时系统通过分布式处理可以充分有效地利用计算机资源。
高 I/O 吞吐量	I/O 吞吐量是指计算机系统处理输入输出数据的速率。它依赖于 CPU、I/O 设备的速度、特性以及 I/O 总线的带宽。要求高 I/O 吞吐的实时应用需要大量数据的连续处理。实时应用常常需要同外部世界持续地进行大量的数据交换。
网络	越来越多的实时应用使用网络与其他系统通信和共享资源。
ROM	一些实时系统必须从 ROM 加载。
自满足	一个实时应用可能作为自满足系统的一部分存在，所谓自满足系统是指系统同时具备实时和分时的能力。例如一个应用可以实时地收集数据并且分析、管理或制订一个运行在自满足环境中的数据报告。这种应用常常称为软实时应用。
稳定性	实时系统一旦实现，必须要求是稳定的和可预测的。

许多实时系统（特别是控制系统）通过系统资源的可用性来优化资源使用。时间关键性应用要求系统在需要时提供充足的 CPU 处理时间、物理内存和 I/O 带宽。实时应用通过单个应用中多个代码路径的异步地执行，从而控制这些资源的使用。

广义上包括：

- 外部事件涉及的代码路径，如中断服务程序和 I/O 完成程序。
- 运行在单个处理器或物理上分布的多个处理器之上的多个分开的任务和子任务。
- 实时系统必须能够执行诸如上下文交换、任务间通信和有效的同步等操作。

2.1.2 开发阶段划分

嵌入式软件的开发过程如图 2.1 所示。

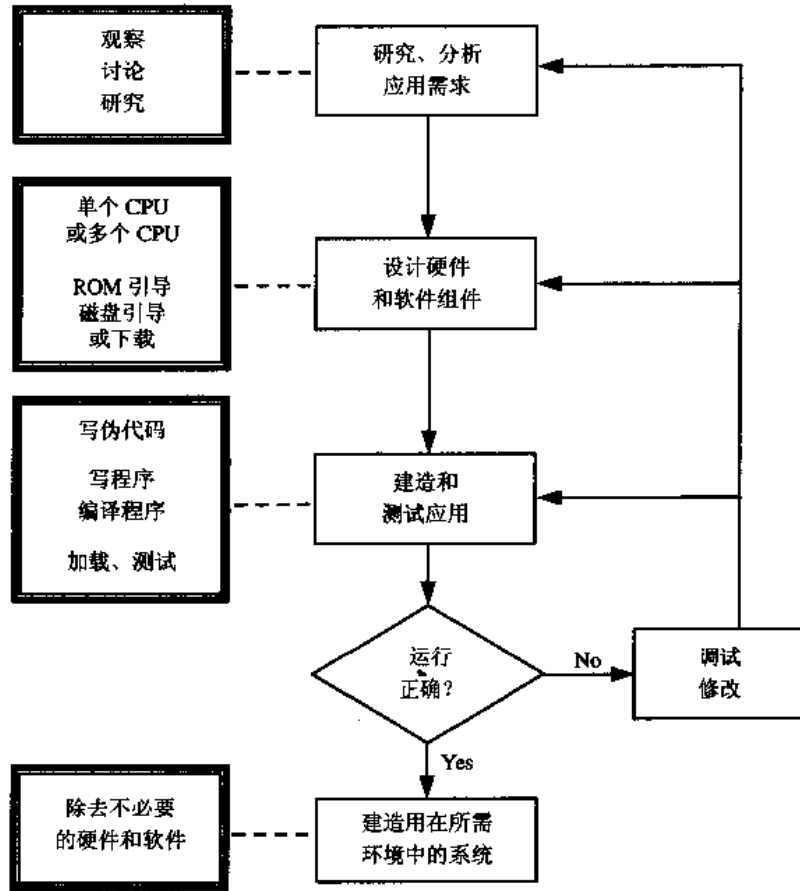


图 2.1 嵌入式软件的开发过程

2.2 VxWorks 简介

2.2.1 VxWorks 发展

这是从 VxWorks Faqs 得到的。1981 年, Jerry Fiddler 和 David Wilner 创建了 WRS。开始主要作为一个面向实时、嵌入式系统和其他感兴趣的东西的服务咨询商。Francis Coppola 是最早的顾客之一。

最初, 他们写了大量的程序, 其中的一大部分, 为嵌入式系统增加了非常优秀的特征, 包括一些在那个时代前所未闻的一些东西, 像 TCP/IP 网络协议和 NFS, 等等。

最初 VxWorks 是运行 VRTX、pSOS 以及早期的较慢的 WIND 内核等实时内核之上的软



件集合的名字。从 5.0 发行起，现在 VxWorks 不再运行于其他核之上，只运行于它自己的 WIND 内核之上，这个 WIND 内核由 John Fogelin 重写。

随着对 VxWorks 系统感兴趣的人越来越多，WRS 卖出了很多这个系统的拷贝，逐渐成长为一个成功的公司。

1995 年开发出的 Tornado，赢得了电子设计新闻该年度的“嵌入式开发软件创新奖”（Electronic Design News' "Embedded Development Software Innovation of the Year" award）。

2.2.2 VxWorks 在嵌入式开发中的角色

可能我们已经习惯了 Unix 和 Windows 这些用户界面友好、开发工具丰富的交互式编程开发系统，但是由于嵌入式、实时系统的时空局限性，他们不适合于实时应用开发。另一方面，传统的实时操作系统提供的用于开发的环境资源非常贫乏，比如图形用户接口、调试工具之类非实时组件。

WindRiver 的设计哲学是充分利用二者（VxWorks 和 UNIX，或 VxWorks 和 Windows）的优点，相互补充达到最优，而非要创建一个万能的单一的操作系统。VxWorks 处理紧急的实时事务，同时主机用于程序开发和非实时的事务。开发者可以根据应用需要恰当地裁减 VxWorks。开发时可以包含附加的功能（如网络功能）以便加速开发过程，在产品最终版本中，去掉附加的功能，以节省系统资源。

开发者可以使用基于主机上的集成开发环境 Tornado，来编辑、编译、联接和存储实时代码，但是实时代码的运行和调试都在 VxWorks 上进行。最终生成的目标映像可以脱离主机系统和网络，单独运行在 ROM 或磁盘上（软/硬/Flush 盘）。当然，主机系统和 VxWorks 也可以在一个混合应用中共同工作：通过网络连接，主机使用 VxWorks 系统作为实时服务器。

2.2.3 VxWorks 的特点

VxWorks 操作系统是现在所有独立于处理器的实时系统中最具特色的操作系统之一。

VxWorks 系统运行环境支持的 CPU 包括：Power PC、68K、CPU32、SPARC、i960、x86、Mips 等；同时支持 RISC、DSP 技术。支持多种硬件环境也是 VxWorks 得以流行的重要原因。VxWorks 的微内核 Wind 是一个具有较高性能的、标准的嵌入式实时操作系统内核，其主要特点包括：快速多任务切换、抢占式任务调度、任务间通信手段多样化等。该内核具有任务间切换时间短、中断延迟小、网络流量大等特点，与其他嵌入式实时操作系统相比具有一定的优势。

VxWorks 系统具有较好的可剪裁的能力，可剪裁的组件超过 80 个，用户可以根据自己



系统的功能目标通过交叉开发环境方便地进行配置。

VxWorks 支持应用程序的动态链接和动态下载,使开发者省去了每次调试都将应用程序与操作系统内核进行链接和下载的步骤,缩短了编辑/调试的周期。

VxWorks 具有较好的兼容性。VxWorks 良好的兼容性,使其在不同运行环境间可以方便的移植,从而使用户在开发和培训方面所做的工作得到保护,减少了开发周期和经费。

VxWorks 是最早兼容 POSIX1003.1b 标准的嵌入式实时操作系统之一,同时也是 POSIX 组织的主要会员。

VxWorks 的 TCP/IP 协议栈部分在保持与 BSD4.4 版本的 TCP/IP 兼容基础上,在实时性方面有较大提高。这使得基于 BSD4.4 UNIX Socket 的应用程序可以很方便地移植到 VxWorks 中去,并且网络的实时性得到提高。

VxWorks 还是第一个通过 Windows NT 测试的可以在 Window NT 平台进行开发和仿真的嵌入式实时操作系统。同时支持 ANSI C 标准,并通过 ISO9001 的认证。

2.2.4 VxWorks 操作系统组成

VxWorks 操作系统包括了进程管理、存储管理、设备管理、文件系统管理、网络协议及系统应用等几个部分。VxWorks 只占用了很小的存储空间,并可高度裁减,保证了系统能以较高的效率运行。VxWorks 体系结构框图如图 2.2 所示。

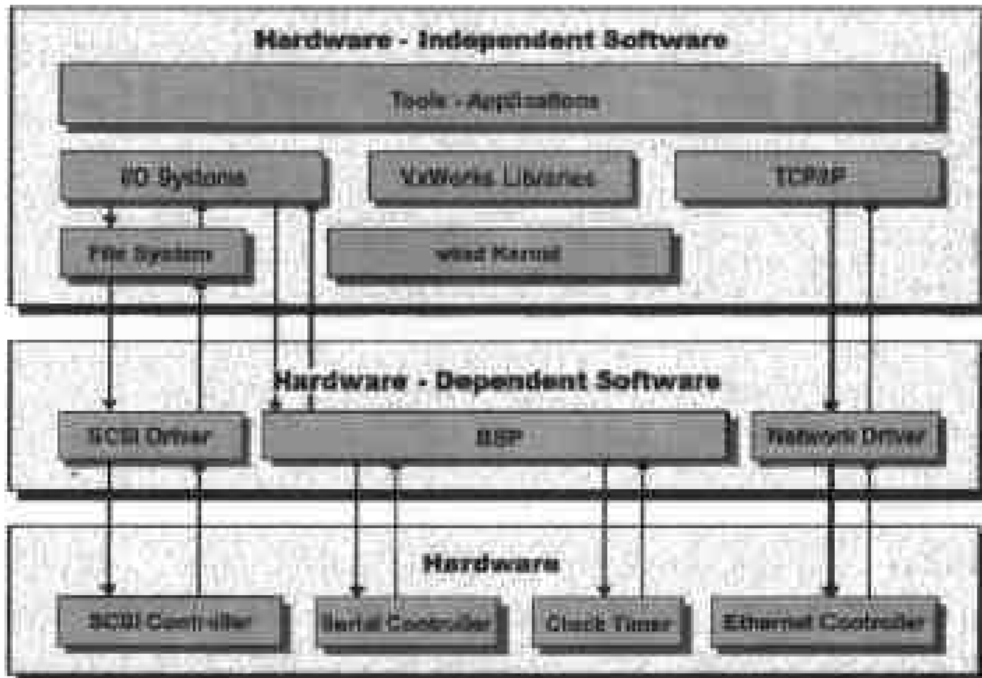


图 2.2 VxWorks 体系结构



VxWorks 由以下几个主要部分组成:

> 高性能的实时操作系统核心—Wind

VxWorks 的核心,一般称作 Wind,包括多任务调度(采用基于优先级的抢占方式),任务间的同步和进程间通信机制以及中断处理、看门狗和内存管理机制。一个多任务环境允许实时应用程序以一套独立任务的方式构筑,每个任务拥有独立的执行线程和它自己的一套系统资源。这些任务基于进程间通信机制同步、协调其行为。

Wind 使用中断驱动和基于优先级的调度方式。它缩短了上下文转换的时间开销和中断的时延。在 VxWorks 中,任何例程都可以被启动为一个单独的任务,拥有它自己的上下文和堆栈。还有一些其他的任务机制可以使任务挂起、继续、删除、延时或改变优先级。

Wind 提供信号量作为任务间同步和互斥的机制。在 Wind 中有几种类型的信号量,它们分别针对不同的应用需求:二进制信号量、计数信号量、互斥信号量和 POSIX 信号量。所有的这些信号量是快速和高效的,它们除了被应用在开发设计过程中外,还被广泛地应用在 VxWorks 高层应用系统中。对于进程间通信, Wind 也提供了诸如消息队列、管道、套接字和信号等机制。

> I/O 系统

VxWorks 提供了一个快速灵活的与 ANSI C 兼容的 I/O 系统,包括 UNIX 标准的缓冲 I/O 和 POSIX 标准的异步 I/O。VxWorks 包括以下驱动程序:

- ◆ 网络驱动:用于网络通信(以太网、共享内存),支持 3Com、ne2000、fei、DEC21x4x 等多种网卡。
- ◆ 管道驱动:用于任务间通信。
- ◆ RAM 盘驱动:用于常驻内存的文件。
- ◆ SCSI 驱动:用于 SCSI 硬盘、软盘、磁带、光驱。
- ◆ 键盘驱动:用于 x86 键盘(仅仅存在于 x86BSP)。
- ◆ 显示驱动:用于 x86VGA 文本显示(仅仅存在于 x86BSP)。
- ◆ 磁盘驱动:用于 IDE/ATA 硬盘、软盘(仅仅存在于 x86BSP)。
- ◆ 并口驱动:用于 PC 风格的目标机。

> 文件系统

VxWorks 提供的快速文件系统适合于实时系统应用。它包括几种支持使用块设备(如磁盘)的本地文件系统。这些设备都使用一个标准的接口,从而使得文件系统能够灵活地在设备驱动程序上移植。另外, VxWorks 也支持 SCSI 磁带设备的本地文件系统。

VxWorks I/O 体系结构甚至还支持在一个单独的 VxWorks 系统上同时并存几个不同的



文件系统，如 DosFs 用于软、硬盘，cdromFs 用于 CDROM 驱动器。

VxWorks 支持以下几种文件系统：

- ◆ dosFs：与 MS-DOS 兼容的文件系统。
- ◆ rt11Fs：一种与 RT11 操作系统兼容的文件系统。
- ◆ rawFs：raw disk file system。这种文件系统将整个盘作为一个文件，允许根据字节偏移读写磁盘的一部分，其优点是仅仅需要底层 I/O 操作，因而读写速度快，并且大小没有限制。
- ◆ tapeFs：SCSI 顺序文件系统。用于磁带设备，不使用标准的文件和目录结构，将整个磁带作为一个大文件来处理。
- ◆ TrueFFS：闪存文件系统。
- ◆ CdRomFs：VxWorks 提供的 cdromFs 文件系统，应用可读取任何按照 ISO 9660 文件系统标准格式化的 CD-ROM。一旦 cdRomFs 已经初始化，并且已经登录到一个 CD-ROM 块设备，应用就可调用标准 POSIX I/O 调用，访问 CD-ROM 设备上的数据。VxWorks 目前只支持 SCSI 接口的 CD-ROM，（我们已经开发了 IDE 接口的 CD-ROM 驱动程序）。

另一方面，普通数据文件，外部设备都统一作为文件处理。它们在用户面前有相同的语法定义，使用相同的保护机制。这样既简化了系统设计又便于用户使用。

➤ 板级支持包 BSP (Board Support Package)

板级支持包对各种板子的硬件功能提供了统一的软件接口，它包括硬件初始化、中断的产生和处理、硬件时钟和计时器管理、局域和总线内存地址映射、内存分配等等。每个板级支持包括一个 ROM 启动 (Boot ROM) 或其他启动机制。

➤ 网络设施

VxWorks 的网络结构如图 2.3 所示。它提供了对其他网络和 TCP/IP 网络系统的“透明”访问，包括与 BSD 套接字兼容的编程接口，远程过程调用 (RPC)，SNMP (可选项)，远程文件访问 (包括客户端和服务端的 NFS 机制以及使用 RSH、FTP 或 TFTP 的非 NFS 机制) 以及 BOOTP 和 ARP 代理。无论是松耦合的串行线路、标准的以太网连接还是紧耦合的利用共享内存的背板总线，所有的 VxWorks 网络机制都遵循标准的 Internet 协议。

➤ 先进的系列网络产品

VxWorks 内的 WindNet 是先进的、系列的网络产品，这些产品扩展了 VxWorks 的网络特性并增强了嵌入式处理器的网络特性。包括以下产品：

- ◆ BSD 4.4 TCP/IP。
- ◆ IP、IGMP、CIDR、TCP、UDP、ARP。



- ◆ RIP v.1/v.2。
- ◆ 标准 Berkeley 套接字, zbufs (zero-copy socket)。
- ◆ SLIP、CSLIP、PPP。
- ◆ BOOTP、DNS、DHCP、TFTP。
- ◆ NFS、ONC、RPC。
- ◆ FTP、rlogin、rsh、telnet。
- ◆ SNMP。
- ◆ 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)。

WindNet 第三方产品, 包括 OSI、SS7、ATM、Frame Relay、CORBA、ISDN、X.25、CMIP/GDMO、分布式网络管理等。

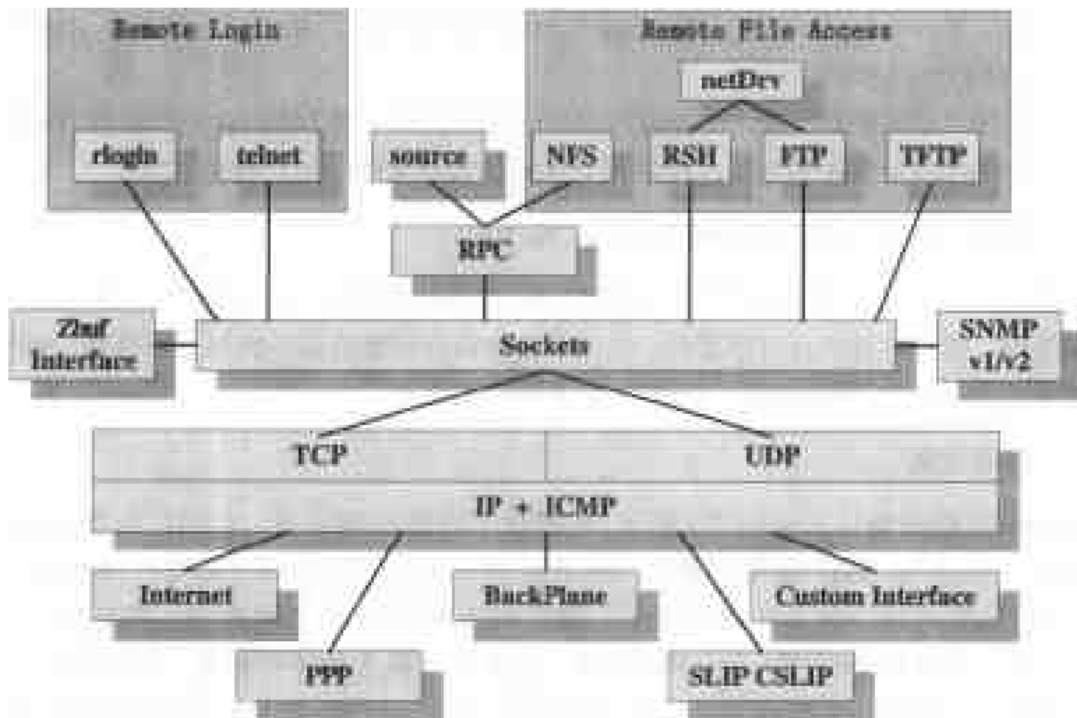


图 2.3 VxWorks 网络结构

➤ 虚拟内存 (即 VxVMI 选项) 与共享内存 (即 VxMP 选项)

VxVMI 为带有 MMU 的目标板提供了虚拟内存机制。VxMP 提供了共享信号量, 消息队列和在不同处理器之间的共享内存区域。



➤ 目标代理 (Target Agent)

目标代理遵循 WBD (Wind Debug) 协议, 允许目标机与主机上的 Tornado 开发工具相连。在目标代理的默认设置中, 如图 2.4 所示, 目标代理是以 VxWorks 的一个任务——tWdbTask 的形式运行的。

Tornado 目标服务器 (Target Server) 向目标代理发送调试请求。调试请求通常决定目标代理对系统中其他任务的控制和处理。默认状态下, 目标服务器与目标代理通过网络进行通信, 但是用户也可以改变通信方式。

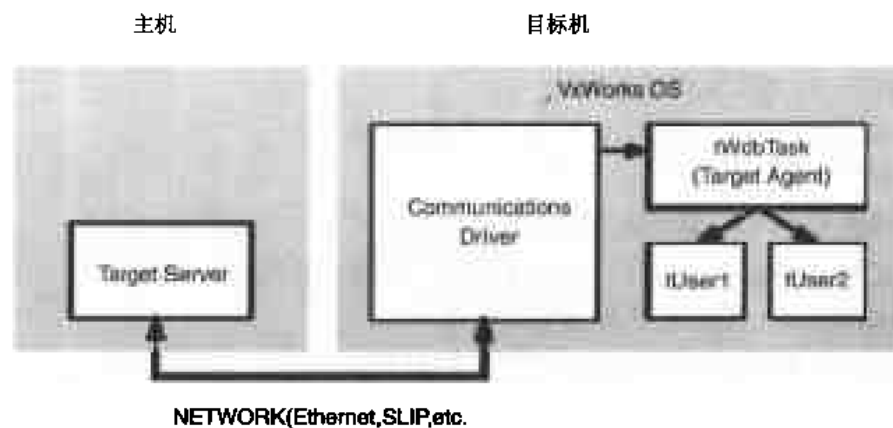


图 2.4 目标代理与目标服务器交互式工作示意

➤ 实用库

VxWorks 提供了一个实用例程的扩展集, 包括中断处理、看门狗计时器、消息登录、内存分配、字符扫描、线缓冲和环缓冲管理、链表管理和 ANSI C 标准。

➤ 基于目标机的工具

在 Tornado 开发系统中, 开发工具是驻留在主机上的。但是也可以根据需要将基于目标机的 Shell 和装载/卸载模块加入 VxWorks。

2.3 Tornado 集成开发环境简介

Tornado II 开发环境是嵌入式实时领域里最新一代的开发调试环境, 是实现嵌入式实时应用程序的完整的软件开发平台, 是交叉开发环境运行在主机上的部分, 是开发和调试 VxWorks 系统不可缺少的组成部分。Tornado 是集成了编辑器、编译器、调试器于一体的高度集成的窗口环境, 给嵌入式系统开发人员提供了一个不受目标机资源限制的超级开发和调



试环境。

Tornado II 开发系统包含三个高度集成的部分：

- VxWorks：运行在目标机上的高性能、可裁剪的实时操作系统；
- Tornado 开发环境：运行在宿主机上，包括一组强有力的交叉开发工具和实用程序，可对目标机上的应用进行跟踪和调试；
- 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。

2.3.1 Tornado IDE 的主要组成

Tornado IDE 的主要组成部分有：

- 集成的源代码编辑器（不过它不支持汉字输入）。
- 工程管理工具。
- 集成的 C 和 C++ 编译器和 make 工具。
- 浏览器（Browser），用于收集可视化的资源，监视目标系统。
- CrossWind，图形化的增强型调试器。
- WindSh，C 语言命令外壳，用于控制目标机。
- VxSim，集成的 VxWorks 目标机仿真器。
- WindView，集成的软件逻辑分析仪。
- 可配置的各种选项，可以改变归 Tornado GUI 的外观等。

Tornado 集成环境可以提供上述所有特征，而不受目标机资源的约束。这些工具通过共享寄于主机的动态联接的目标机系统的符号表运行在主机上。Tornado 主机工具与目标系统交互关系如图 2.5 所示。

主机工具与 VxWorks 系统间的通信由目标服务器和目标代理共同完成，如图 2.4、图 2.5。

使用 Tornado IDE，可以大大缩短嵌入式开发周期。Tornado 支持动态链接与加载，允许开发者可以分批将目标模块加载到目标系统上去。这种动态的链接和加载功能是 Tornado 系统的核心功能，可以使开发者省去通常的开发步骤：在主机上将应用程序与内核链接起来，然后将整个应用程序下载到目标系统上去。这样，编辑-测试-调试的周期就会大为缩短；而且，所有的模块都是可以共享的，主机上的应用程序模块也不需要重新链接，所以，加载目标模块到运行中的 VxWorks 目标系统中以达到调试和重新配置成为可能。

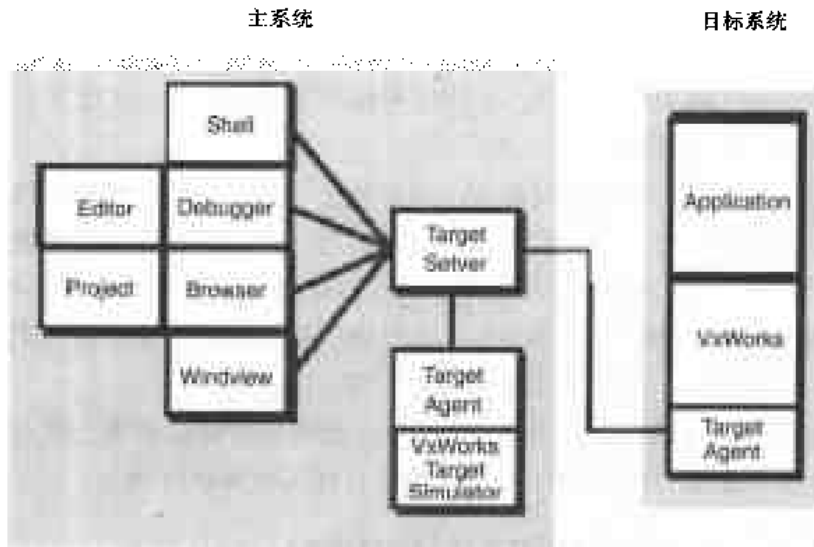


图 2.5 Tornado 主机工具与目标系统交互关系

Tornado II 嵌入式集成开发系统结构如图 2.6。

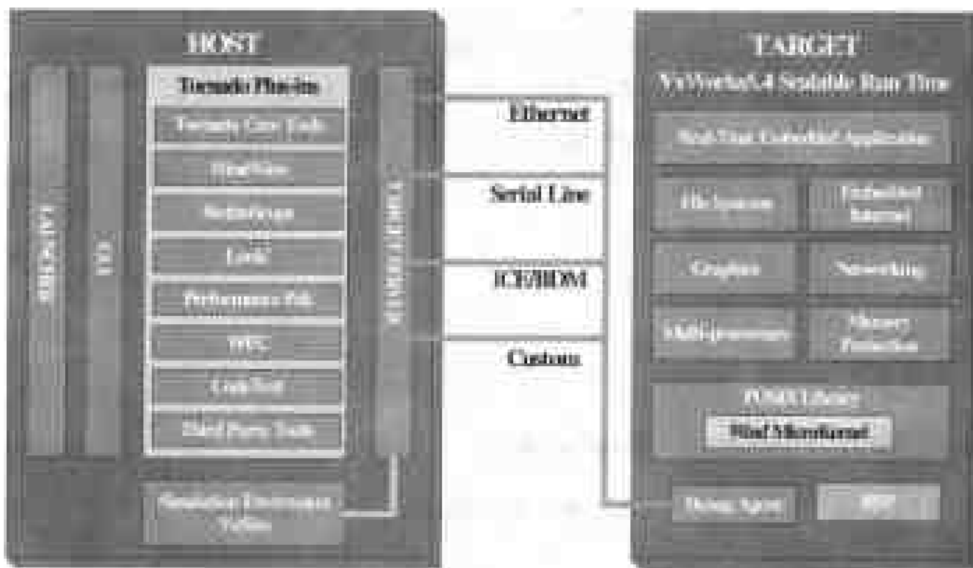


图 2.6 Torando II 嵌入式集成开发系统结构图

2.3.2 Tornado 核心工具

Tornado 软件工具包的核心工具是各个 Tornado 软件工具包都具有的开发工具，主要包



括以下几种:

➤ **图形化的交叉调试器 (Debugger) CrossWind/WDB**

这是一个远程的源代码集成调试器,支持任务级和系统级调试,支持混合源代码和汇编代码显示,支持多目标机同时调试。

这个高性能的调试器具有最新的提高生产率的图形化特征。加速器特征包括开发者可以成组地观察表达式的观察窗口;可以在调试器的图形用户界面中迅速改变变量、寄存器和局部变量的值;可以为不同组的元素设定根值数;通过信息规整和分类的方法有效地提供信息;还提供开发者熟悉的 GNU/GDB 调试器引擎,这种调试器引擎采用命令行方式、命令完成窗口和下拉式的历史记录窗口,因而具有很强的灵活性。

开发者可以在目标运行系统上产生和调试任务,也可以将调试器和已经运行的任务链接在一起,这些任务可以是源自应用程序也可以是来自任务级调试环境。

➤ **工程配置工具 (Project Facility/Configuration)**

这是一个强有力的图形化工具,提供了可以对 VxWorks 操作系统及其组件进行自动地配置。自动的依赖性分析、代码容量计算和自动裁剪 wizard 大大缩短了开发周期。

工程工具简化了 VxWorks 应用程序的组织、配置和建立工作;同时,还使工程管理和 VxWorks 配置的许多方面实现自动化;这种集成的图形化工程管理环境还增强了开发小组的专业技术:单独的组件可以各自独立开发,然后由小组的其他成员共享和重用。由于建立了与现在流行的源代码控制系统的联系,例如:ClearCase、SCCS、RCS、PVCS、MS Visual SourceSafe 等,所以允许小组中的各个成员可以平行工作而不互相干扰。

- ◆ Makefile 自动生成维护。
- ◆ 软件工程维护。
- ◆ 自动的依赖性分析。
- ◆ 代码容量计算。
- ◆ 自动裁剪。

➤ **集成仿真器 (Integrated Simulator)**

这种集成仿真器 VxSim 支持 CrossWind、WindView 和 Browser,提供与真实目标机一致的调试和仿真运行环境。

VxSim 仿真器作为核心工具包含在各个软件包中,因而允许开发者可以在没有 BSP、操作系统配置、目标机硬件的情况下,使用 Tornado 迅速开始开发工作。

作为核心工具包含在各个软件包中的 VxSim 都是限制版本,也就是说,它并不支持网络仿真;如果想获得全部功能的 VxSim,可以根据所买的软件包的条件从 WindPower 可选工具进行选择。



➤ 诊断分析工具 (WindView for the Integrated Simulator)

WindView 是一个图形化的动态诊断和分析工具，主要是向开发者提供目标机硬件上实际运行的应用程序的许多的详细情况。这种系统级的诊断分析工具可以与 VxSim 一起使用。

嵌入式系统开发者经常因为无法知道系统级的执行情况 and 软件的时间特性而感到失望，这种全功能版本的 WindView 提供了运行在集成仿真器上的 VxWorks 应用程序的详细的动态行为，图形化显示了任务、中断和系统对象相互作用的复杂关系。还可以选择用于监视目标硬件系统行为的 WindView。

➤ C/C++编译环境 (C/C++ Compilation Environment)

Tornado 提供交叉编译器、iostreams 类库和一些列的工具来支持 C 语言和 C++ 语言。交叉编译器进行了许多优化，允许开发者能够迅速产生高效而简洁的代码。

Diab C/C++ Compiler: 唯一获得 MOTOROLA 白金大奖的嵌入式编译器。

GNU C/C++ Compiler: 应用最广泛的编译器。

iostreams 类库支持 C++ 中的格式化的和类型安全的 I/O，也可以扩展到用户自定义数据类型，这是 C++ 应用程序开发的工业标准。

Tornado 提供对 C++ 全面的支持，包括：异常事件处理、标准模板库 (STL, Standard Template Library)、运行类型识别 (RTTI, Run-Time Type Identification)、支持静态构造器和析构器的加载器、C++ 调试器，保证了工具与开发环境紧密地结合在一起。

➤ 主机目标机连接配置器 (Launcher)

Tornado 的主机目标机连接配置器 Launcher 允许开发者轻松地设置和配置一定的开发环境，也提供对开发环境的管理和许多其他管理功能。

➤ 目标机系统状态浏览器 (Browser)

Tornado 的目标机系统浏览器 Browser 是 Tornado shell 的一个图形化组件，目标机系统状态浏览器 Browser 的主窗口提供目标系统的全面状态总结，也允许开发者监视独立的目标系统对象：任务、信号灯、消息队列、内存分区、定时器、模块、变量、堆栈等。这些显示根据开发者的选择进行周期性或条件性更新。

➤ 命令行执行工具 (WindSh)

Tornado 的命令行执行工具 WindSh 是 Tornado 所独有的功能强大的命令行解释器，可以直接解释执行 C 语句表达式、调用目标机上的 C 函数、访问系统符号表中登记的变量；还可以直接执行 TCL 语言。

WindSh 不仅可以解释几乎所有的 C 语言表达式，而且可以实现所有的调试功能。它主要有以下调试功能：下载软件模块；删除软件模块；创建并发起一个任务；删除任务；设置



断点；删除断点；运行、单步、继续执行程序；查看内存、寄存器、变量；修改内存、寄存器、变量；查看任务列表、内存使用情况、CPU 利用率；查看特定的对象（任务、信号量、消息队列、内存分区、类）；复位目标机等等。

➤ 多语言浏览器 (WindNavigator)

Tornado 的多语言浏览器 (WindNavigator) 提供源程序代码浏览，图形化显示函数调用关系，快速地进行代码定位，这样大大地缩短了评价 C/C++ 源代码的时间。

➤ 图形化核心配置工具 (WindConfig)

Tornado 的图形化核心配置工具 (WindConfig) 使用图形向导方式智能化的自动配置 VxWorks 内核及其组件参数。

➤ 增量加载器 (Incremental Loader)

Tornado 的增量加载器 (Incremental Loader) 可以动态的加载新增模块并在目标机与内核实现动态链接运行，不必重新下载内核及未改动的模块，加快开发速度。

工欲善其事，必先利其器。Tornado2.0 是与 VxWorks5.4 操作系统相配套的最新一代基于图形用户界面的集成开发环境。本章主要简单介绍 Tornado2.0 功能特点、软件安装，以及工程管理工具、目标仿真器、WindView 和调试器的使用。同时也将介绍 Tornado1.0 的一些使用经验。

3.1 TornadoII 的新特征

在我们使用 Tornado1.0 进行嵌入式实时应用开发的过程中，对之提供的丰富强大的开发工具如 shell、debug 等给开发带来的方便，以及其在较大项目开发如工程创建与管理等方面的一些不足已深有体会。

TornadoII 比 Tornado1.0 又有了较大的改进。主要体现在以下几方面：

➤ 与开发环境集成为一体的 VxWorks 目标仿真器 (VxSim)

在第一章中我们已经提到，嵌入式系统的开发通常是软硬件共同设计反复交叉进行的，所以软件的开发通常受到工程前期缺乏硬件环境的制约，因而软件开发需要滞后进行，影响了产品研制开发进度。VxWorks 目标仿真器的出现在一定程度上解决了这一问题：使得用户在工程开发初期，硬件环境尚不具备的情况下，可以利用 Tornado 进行与硬件无关模块的设计。

➤ 支持目标仿真器的集成的 WindView 逻辑分析仪

集成版的 WindView 可以提供在仿真器上运行的应用程序的动态可视的行为，分析手段更为直观。



➤ 工程管理工具

最新的工程工具可以以图形化方式来管理工程文件、配置 VxWorks 以及建造应用。

➤ 改进的基于 GUI 调试

新的基于 GUI 调试器更易于使用，可以更直接地访问调试信息。

●* 注意

集成的目标仿真器每个用户同时只能运行一次，并且没有网络支持，也不支持其他单独购买的产品，如 STREAMS、SNMP 以及 Wind Foundation Classes 等。单独购买的完全可裁减的仿真器 VxSim 支持这些产品，也支持一个用户同时发起多个事例，并且也支持网络。集成版的 WindView 仅仅支持 VxWorks 目标仿真器。支持所有目标平台的 WindView 需要另行购买。

3.2 Tornado 安装

3.2.1 术语与约定

下面是将要用的有关名词：

- 主机：运行 Tornado IDE 的计算机。
- 目标机：运行实时操作系统 VxWorks 和用 Tornado 开发的应用程序的 CPU 板。
- 目标机服务器 (target server)：一种服务，它运行在主机上，用来管理主机工具（诸如：Tornado shell、debugger 和 browser）与目标机系统自身之间的通信。每个目标机各自需要一个目标机服务器。
- Tornado 注册器 (registry)：一种 Tornado 服务，用于提供跟踪以及访问目标机服务器的服务。

一个子网上仅需要一个注册器，但是，注册器也可能运行在每台 Tornado 主机之上。

下面是一些约定：

默认的安装驱动器和根路径是：c:\tornado，当然用户在安装时可以根据需要选择合适的驱动器和安装路径。

在 GUI 中，连续的几个选择项，用 A>B>C 来表示。A、B 和 C 是菜单项、按钮或标签。



3.2.2 安装前的准备

简单来说，Tornado 安装涉及以下三步：

- (1) 确定 Tornado 安装 key；
- (2) 选择用户账号；
- (3) 运行 SETUP 安装程序，它将需要安装 key，以及一些其他的信息，如用户、工程等有关信息。

3.2.3 系统需求

在安装 Tornado 之前，最好先阅读《Tornado Release Notes》了解对主机操作系统要求如：版本、补丁等信息。

Tornado 能够运行在 PC 系列微机、Sun-4 和 HP9000 series 700 工作站上。

主机系统必须具备以下条件：

➤ UNIX

- ◆ 64 MB RAM (最好是 128 MB)；
- ◆ 典型安装需要至少 150 MB 磁盘剩余空间。
- ◆ 一个 CD-ROM 驱动器。
- ◆ 对于 Solaris 系统，最低需求是 SPARCstation 5，推荐使用 Ultra5，对于 HP-UX 系统，推荐使用 C100 以上的工作站。
- ◆ 用于阅读 HTML 文档的浏览器，推荐使用 Netscape 3.02 或以上版本。

➤ Windows

- ◆ 32 MB RAM (最好是 64 MB)
- ◆ 典型安装需要至少 300 MB 磁盘剩余空间。
- ◆ 一个 CD-ROM 驱动器。
- ◆ Intel 80486 以上微机，推荐使用 Intel Pentium 90 以上微机。
- ◆ 建议使用一块以太网接口卡，用来通过网络进行调试。
- ◆ 用于阅读 HTML 文档的浏览器，推荐使用 Netscape 3.02 或者 Microsoft IE 3.02 或以上版本。
- ◆ 主机系统必须安装 TCP/IP 协议，无论使用网络还是用串行口进行调试。



➤ 操作系统需求

Tornado 支持的操作系统如表 3.1。

表 3.1 Tornado 支持的操作系统

主 机	操 作 系 统	补 丁
PC	Windows NT4.0 Windows 95 Windows 98	Service pack 3 或以上版本
Sun-4	Solaris 2.5.1 Solaris 2.6 Solaris 2.7	必须是完全的升级 p105362-09 或以上版本
HP9000 series 700	HP-UX 10.2x	PHCO_15465 libc cumulative patch PHSS_13898 Xserver cumulative patch PHSS_14731 HP C++ core library components (A.10.36) PHSS_16585 HP aC++ runtime libraries (aCC A.01.15) PHKL_16750 SIG_IGN/SIGCLD,LVM,JFS,PCI/SCSI cumulative patch PHSS_15391 dld.sl(5) cumulative patch

3.2.4 安装步骤

本节以 Microsoft WindowsNT 4.0 为例介绍详细的安装步骤。

在安装之前首先检查主机操作系统是否满足上一节提出的要求。对 WindowsNT 4.0 而言，主要注意，TCP/IP 网络协议必须已经安装，因为无论 Tornado 使用串行线还是以太网线与目标机连接，Tornado 工具都必须使用 TCP/IP 协议来通信。因此，尽管主机系统可能不存在网络连接，都必须安装 TCP/IP 协议。Windows TCP/IP 协议包的安装方法参考相应的文档。

此外，还要允许 Windows 支持长文件名（Windows 安装默认支持）。

步骤 1: 启动 SETUP 程序

将 Tornado 光盘放入 CD-ROM 驱动器中，安装程序 SETUP 将自动运行，或者在资源管理器中光驱根目录下双击 SETUP.EXE。

屏幕出现欢迎对话框，如图 3.1。点击 Next 按钮继续。



图 3.1 Tornado 安装欢迎对话框

注意

SETUP GUI 带有上下文敏感的帮助，能够提供有关 SETUP 使用和安装过程的详细信息。使用方法是：点击标题栏 help 按钮，然后选择你想获得帮助的对象。

每个 SETUP 对话框都有一个 Cancel 按钮，点击它就可以退出 Tornado 安装程序。点击 Next 按钮，就可进入下一个对话框。如果需要修改前面曾经选择的信息，那么点击 Back 按钮（如果没有被使用其颜色为灰色），就可回到前一个对话框，做出重新选择。

步骤 2: 浏览 Readme 信息

当出现 README.TXT 窗口时，仔细阅读有关信息。点击 Next，继续安装过程。

步骤 3: 浏览 License Agreement

阅读 license agreement。如果你能够接受，点击 Accept 和 Next 继续安装，否则，点击 Cancel 退出安装过程。

步骤 4: 输入用户信息和安装 Key

在用户注册对话框中，输入用户名和公司名称，以及安装 key，如图 3.2。点击 Next，



继续安装过程。



图 3.2 Tornado 安装用户注册对话框

步骤 5: 选择安装类型

Tornado 既可以允许安装在本地主机，又可以安装到远程服务器上。如图 3.3。

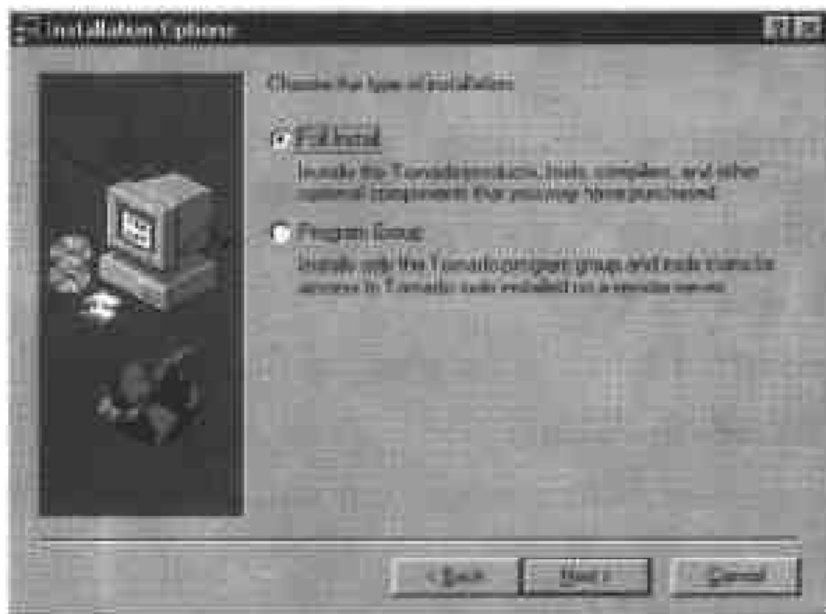


图 3.3 选择安装类型对话框



这个对话框中的第一个选项，将在本地主机安装完整的 Tornado 文件。第二个选项仅仅在本地主机的开始菜单中安装 Tornado 程序组，这样可以在本地主机运行远程服务器上的 Tornado。此外，第二个选项也允许访问安装在本地主机其他版本 Windows 下的 Tornado（对于一些双重引导系统），如果你的机器既有 Windows NT，也装有 Windows 98，在 Windows 98 中已经安装过 TornadoII，并且 Windows NT 也可访问 TornadoII 所在驱动器，那么你可以选择第二个选项，这样就可节省你的磁盘空间，也可避免重复劳动。

如果希望在本地主机上安装 Tornado，选择 Full Install，然后点击 Next，继续安装过程；如果希望访问安装在服务器上或者是安装在本地主机其他版本 Windows 下的 Tornado，选择 Program Group，然后点击 Next，继续安装过程。

然后出现选择安装目录窗口（Select Directory），如图 3.4。输入安装的驱动器和路径，或者点击 Browse 按钮，选择安装路径，然后点击 Next，继续安装过程。

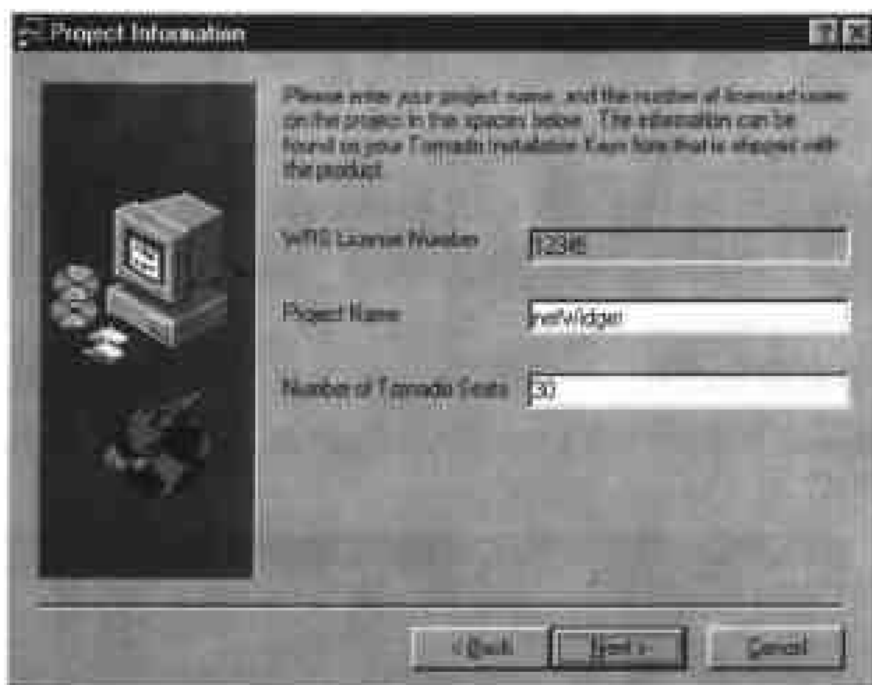


图 3.4 选择安装目录对话框

●* 注意

如果选择的是 Program Group，而非 Full Install，跳过 6、7、8 步，转到第 9 步：选择开始菜单程序组文件夹的名字。



步骤 6: 输入工程信息

在如图 3.4 所示的对话框中输入工程名称和用户的数目。license number 域将自动完成。然后单击 Next, 继续安装过程。

步骤 7: 指定安装目录

Tornado 及其有关软件需要安装在同一个目录下, 在 Select Directory 对话框中指定驱动器和安装的根目录。也可以使用 Browse 按钮来选择。安装所在驱动器应该有较充足的可用空间。然后单击 Next, 继续安装过程。

警告

不要将 Tornado 安装在子目录下, 例如 c:\Program Files\Tornado, 这样可能引起 make 和其他工具程序不能正常工作。

警告

不要将 Tornado 与以前安装的其他版本的 Tornado 安装在同一目录下, 否则两者都不能正常工作。

步骤 8: 浏览安装产品列表

在 Select Products 对话框中包含了所有可供安装的产品, 通过左边对应的复选框可以选择将要安装的产品, 如图 3.5。

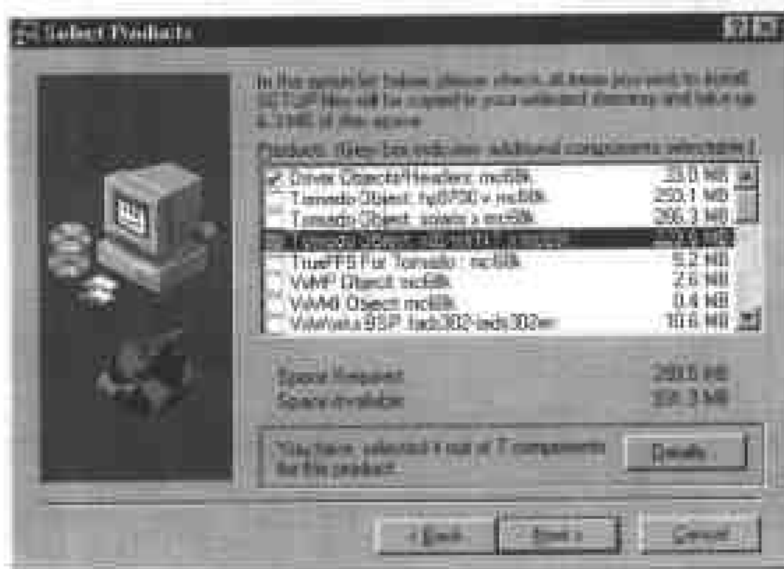


图 3.5 选择安装产品列表对话框



如果产品包含多个部分，点击 **Details** 按钮将显示每个部分的详细信息并能修改默认的选择。

Select Products 对话框显示安装需要的磁盘空间和可用磁盘空间大小等信息，如图 3.6。

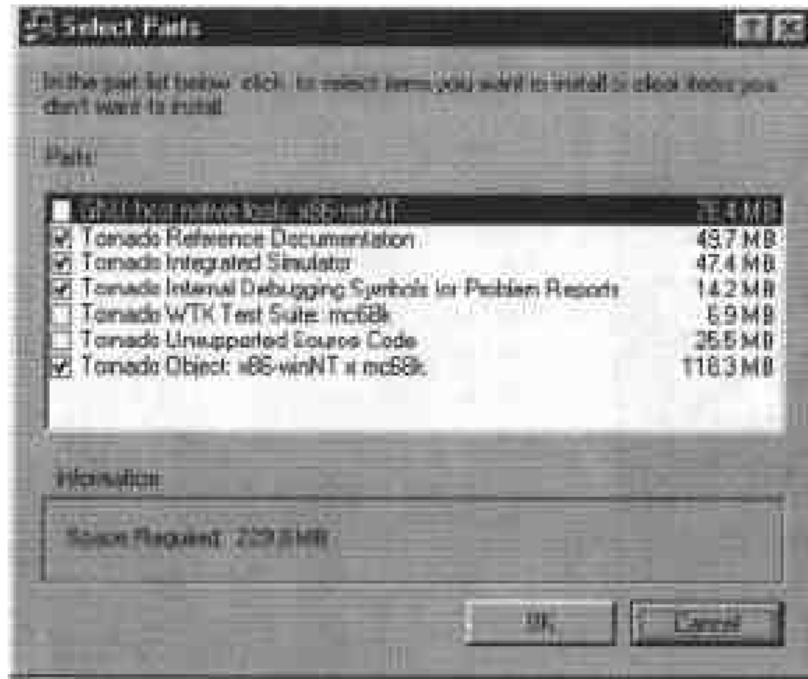


图 3.6 选择安装产品信息对话框

然后点击 **Next**，继续安装过程。

如果没有足够的磁盘空间用于安装，**SETUP** 将出现提示，可以重新选择安装驱动器或者忽略安装一些产品。

步骤 9: 选择开始菜单程序组文件夹的名字

在 **Select Folder** 对话框中为 **Tornado** 选择开始菜单程序组文件夹的名字。默认的名字是 **Tornado**。

然后点击 **Next**，继续安装过程。

步骤 10: 设置 Tornado 注册器 (Tornado registry)

Tornado registry 是一种 **Tornado** 服务，用于提供跟踪以及访问目标机服务器的服务。一个目标机服务器 (target server) 是一种服务，它运行在主机上，用来管理主机工具 (诸如: **Tornado shell**, **debugger** 和 **browser**) 与目标机系统自身之间的通信。每个目标机各自需要一个目标机服务器。**Tornado registry** 必须在目标机服务器被启动和 **Tornado** 工具与目标机通



信之前运行。Tornado 主机可以设置来访问一个本地或者远程主机上的 Tornado registry。在共享网络上仅需要一个 registry，而且如果只使用一个的网络 registry，它就允许从所有的 Tornado 主机访问所有的目标。

如果选择 No 和 continue 按钮，系统将提示释放系统空间。

如图 3.7，Registry 对话框提供以下选项：

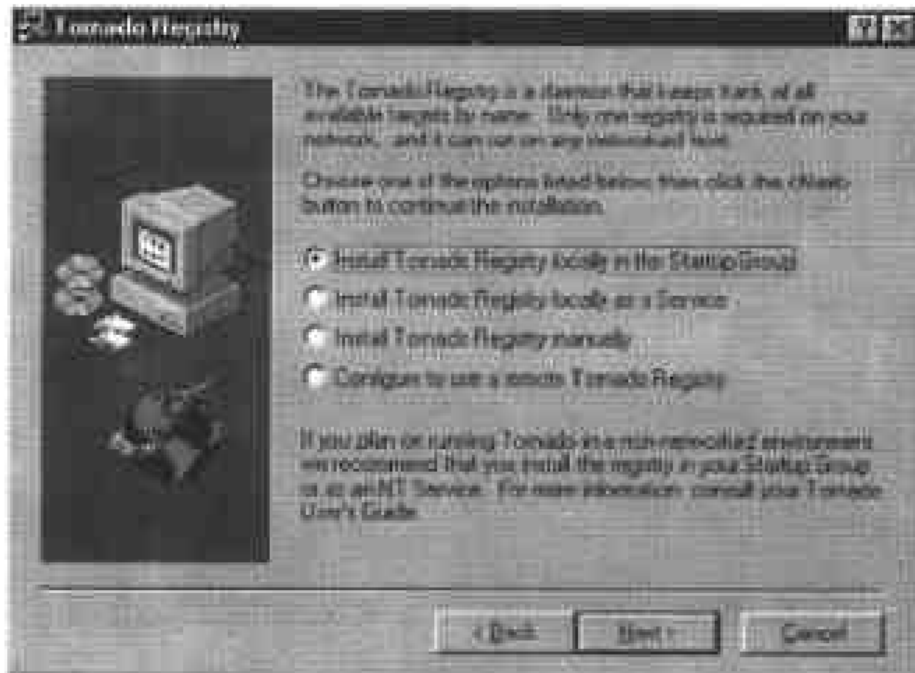


图 3.7 设置 Tornado 注册器对话框

- 1. Install Tornado Registry locally in the Startup Group**
将 Tornado Registry 安装在本地启动程序组中。
选择这个选项，Tornado registry 将在当前用户登录到本地主机时自动启动。
- 2. Install Tornado Registry locally as a Service**
将 Tornado Registry 作为一个服务安装。
选择这个选项，Tornado registry 将在本地主机启动时自动启动。该选项仅在 WindowsNT 下可用。
- 3. Install Tornado Registry manually**
手工安装 Tornado Registry。
选择这个选项，用户在运行 Tornado 之前，必须手工在本地主机启动 Tornado Registry。如果你不是经常使用 Tornado，那么选择这个选项对系统性能可能有所帮助。特别是在 Windows 98 中，Tornado1.0.1 的 Registry 没有提供退出命令，每次都需要强制结束，极为不便。



4. Configure to use a remote Tornado Registry

配置使用远程 Tornado Registry。

选择这个选项，将使用网络上其他主机的 Tornado Registry。将出现一个 Tornado Registry 对话框，要求输入所用 Tornado Registry 所在网络主机的名字。

然后点击 Next，继续安装过程。

步骤 11: 配置与 Tornado 1.0.1 工具兼容

如果希望在同一台开发主机上同时使用 Tornado 1.0.1 和 Tornado 2.0 的工具，在 Backward Compatibility 对话框中选择 yes，如图 3.8。熟悉 Tornado 1.0.1 的用户最好选择这个选项。

☛ 注意

如果在第 10 步时，选择了将 Tornado registry 作为一种服务安装，那么就不能在 Tornado 2.0 使用 Tornado 1.0.1 的工具，当然可以点击 Back 按钮，退回 Tornado Registry 对话框，改变原来的选择。

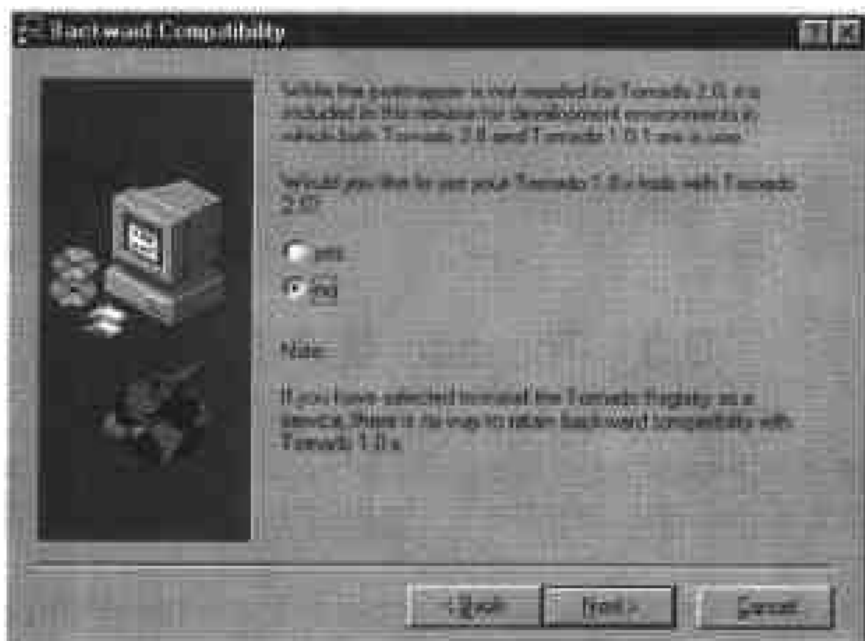


图 3.8 Backward Compatibility 对话框

然后点击 Next，继续安装过程。

SETUP 将根据前面的设置开始安装 Tornado。在安装过程中，将显示一些关于 Tornado 的信息，以及当前正在安装的文件的名字。然后 SETUP 升级 VxWorks 库，注册 Tornado DLL，



显示进度信息；整个安装过程可能需要几十分钟，由你的机器速度和选择安装的组件大小决定。

当 SETUP 完成 Tornado 的安装，最后将出现一个确认对话框，如图 3.9。点击 OK 按钮关闭该对话框，标志着整个 Tornado 的安装成功完成。



图 3.9 确认对话框

根据安装时选择的 Tornado registry 的启动方式的不同，可能需要重新启动计算机才能正确使用 Tornado。

3.3 Tornado 简单教程

这一节主要介绍 Tornado 2.0 使用方法和主要特点，与 Tornado IDE 集成在一起的 VxWorks 目标仿真器、WindView 的使用方法。它不需要任何目标硬件，也不需要主机系统特定的配置。

本章不是 VxWorks 编程教程，一些简单的程序主要是说明 Tornado 的使用方法。为了强化效果，按照本章的介绍，动手一步步地完成操作，可能会加快加深对 Tornado 使用的经验。

本节主要包括以下内容：

- 使用工程工具创建一个用于示例程序的工作空间和工程。
- 使用工程工具的 GUI 建造 (build) 程序。
- 将示例程序下载到运行在主机上的 VxWorks 目标仿真器 (VxSim)。



- 使用 Tornado shell 运行示例程序。
- 使用 browser 观察目标仿真器的内存使用情况。
- 使用软件逻辑分析仪 WindView 图形化地显示示例程序的执行流程，并说明任务优先级的问題。
- 使用 debugger 调试运行时应用的错误。

正常情况下，Tornado 允许多次修改程序，纠正其运行时的错误，重新建造、下载、重新运行。

本节假定读者具备基本的 C 语言多线程编程的知识，了解 Windows 系统的基本使用。

3.3.1 启动 Tornado

点击 Windows 任务栏上的“开始”按钮，选择“程序”文件夹，然后选择“Tornado”程序组（Tornado 默认安装其程序组名为“Tornado”），点击其中的 Tornado 项，即可启动 Tornado 运行。

如果 Tornado 没有连接到 Tornado registry，系统将提示启动一个 Tornado registry。可以点击 Tornado 程序组中的 Tornado registry 选项，手工启动 Tornado registry。

如果是第一次启动 Tornado，Tornado 主窗口出现的时候默认将弹出一个创建工程对话框，如图 3.10。



图 3.10 Tornado 主窗口与创建工程对话框



用户可以使用工具条作为浮动板或者将其放置在 Tornado 主窗口的两边、顶部或者是底部。

3.3.2 创建工程

如果在打开 Tornado 时在已有工作空间中创建新工程对话框（Create Project in New/Existing Workspace）没有出现，就需要点击：

File>New Project

将出现创建新工程对话框，如图 3.11。

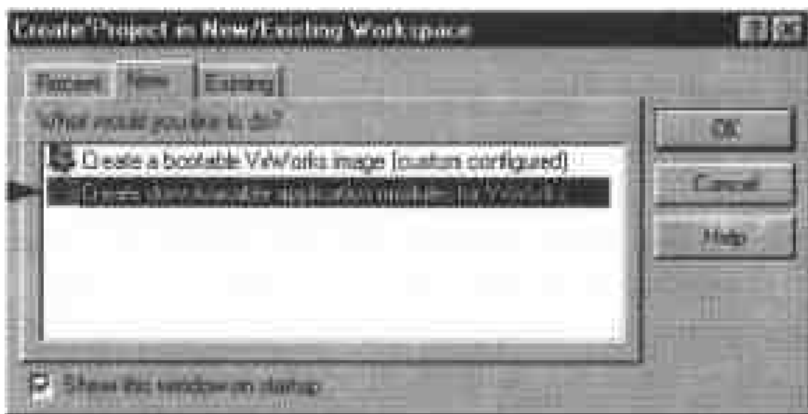


图 3.11 创建新工程对话框

然后选择“Create downloadable application Modules for VxWorks”；

点击 OK 继续。

屏幕将出现“Tornado application wizard”。该程序将引导用户一步步地创建一个新工程。

首先提示用户输入：

- 新工程的名字。
- 工程文件存放的路径。
- 工程有关的一些说明。
- 该工程所属工作空间的名字和位置，它包含了工作空间的有关信息。

一个工程由源代码文件，建造设置以及用于创建一个应用的二进制代码等部分组成。一个工作空间一般由一个或多个工程组成，为工程之间代码共享及应用间相互联系提供一种手段。一旦工作空间和工程已经创建，工作空间窗口将显示它所包含所有工程的有关信息。

现在，我们接受工作空间和工程这些默认设置，但是最好将它们放置到 Tornado 安装目录之外。（否则当升级 Tornado 时，这些工作空间和工程将可能遭到破坏）。



在下面显示的对话框中，工程名字是 `gizmo`；工程路径是 `d:\projects\gizmo`；工程描述是 `lightning gizmo`；工作空间的路径及名称是 `d:\projects\lightning.wsp`，如图 3.12。

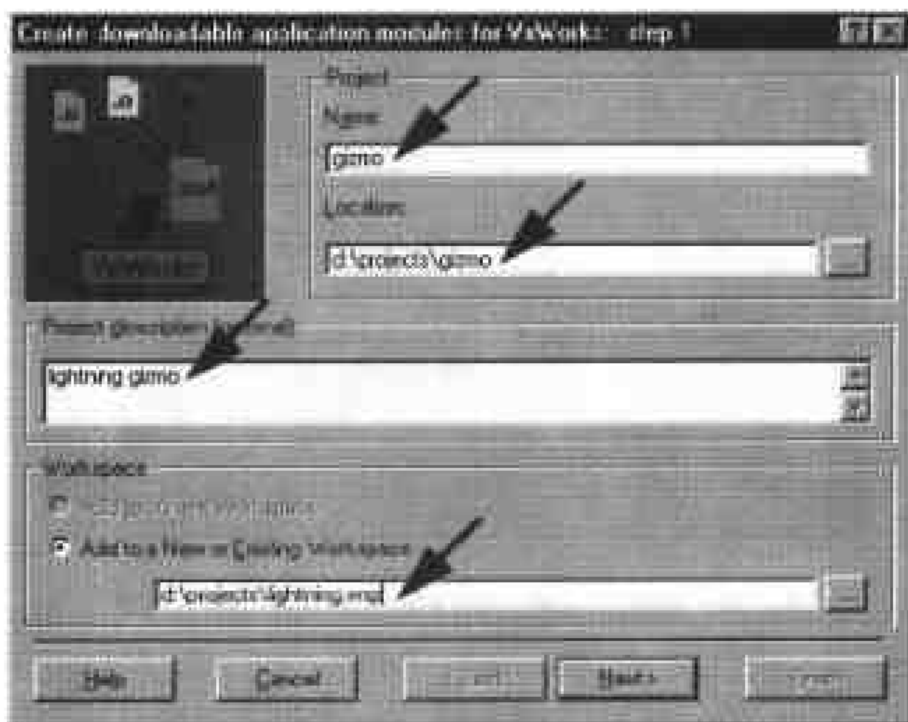


图 3.12 Tornado application wizard 第一步：工程定位

点击 `Next`，继续下一步。

在接下来出现的对话框中，需要我们选择建造该可下载应用使用的工具链（`toolchain`）。如图 3.13。一个工具链是一套用来建造针对特定目标机应用的交叉开发工具。Tornado 提供的工具链是基于 GNU 的预处理（`preprocessor`）、编译、汇编和链接器。针对目标仿真器，默认的工具链名字其形式是 `SIMhostOsgnu`，对于 NT 主机是 `SIMNTgnu`。

●* 注意

在 wizard 创建工程和工作空间之前，该工程和工作空间的根目录必须存在。在本例中，`d:\projects` 必须存在。

在下拉列表中选择针对目标仿真器的工具链和默认选项，如下图。

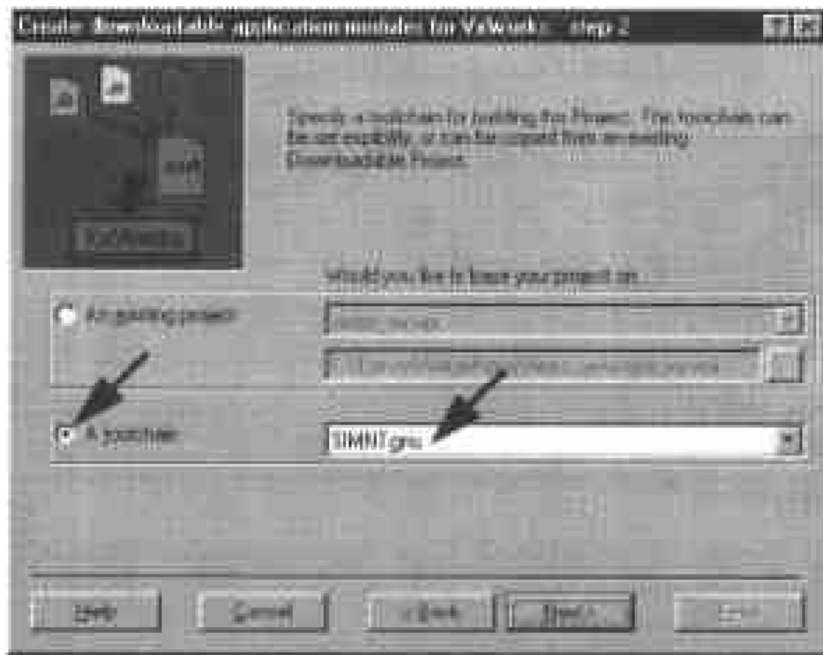


图 3.13 Tornado application wizard 第二步：选择工具链

点击 Next，继续下一步。

Wizard 最后的对话框将要求确认以上选择，如图 3.14。

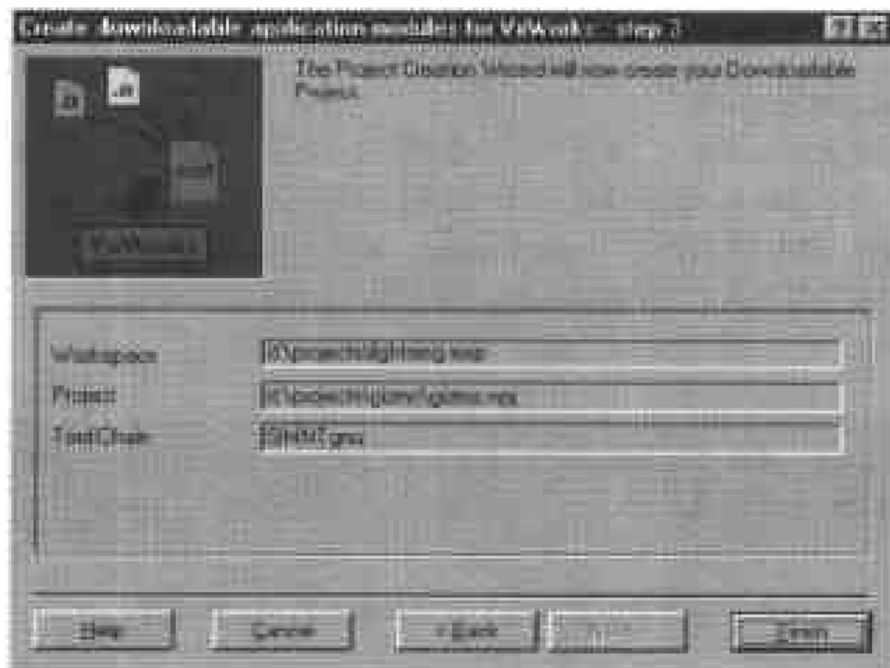


图 3.14 Tornado application wizard 第三步：确认



点击 **Finish**，继续下一步。

这时将出现工作空间窗口。工作空间窗口标题是该工作空间的名字。窗口自身包含了工程信息夹，如图 3.15。

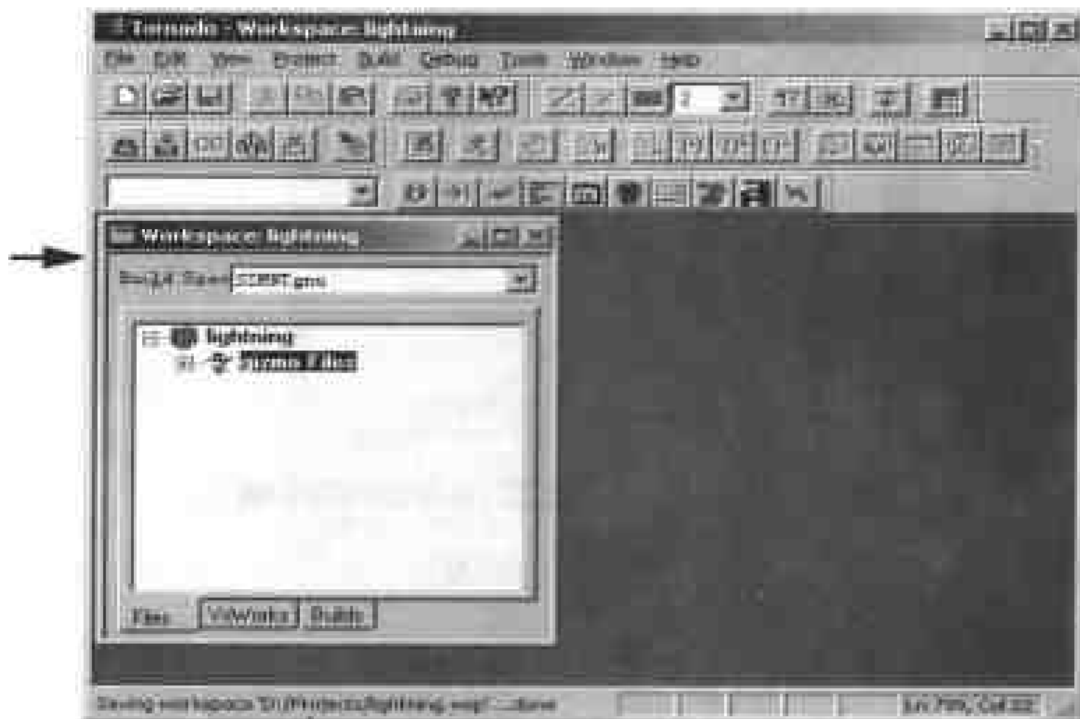


图 3.15 工作空间窗口

3.3.3 添加源文件

现在就可为工程添加源程序，本例中源程序的名字是 `cobble.c`。它是一个简单的多任务应用，模拟一个数据采集系统，数据来自于外部源（例如，当数据到达时，一个设备将生成中断）。第一个任务模拟一个中断服务程序生成新数据。第二个任务收集数据。第三个任务处理数据，完成计算并求出一个结果。第四个任务监视结果值，当结果超出安全范围时，向屏幕打印报警信息。

在向工程添加这个源程序之前，将它从 Tornado 安装目录复制到本地目录，也就是该工程所在目录 `d:\projects`。

示例源文件 `cobble.c` 位于 `c:\tornado\target\src\demo\start` 目录下（假定 Tornado 安装在 `c:\tornado` 目录下）。

当把 `c:\tornado\target\src\demo\start\cobble.c` 拷贝到 `d:\projects` 后，我们就可以把它添加到工程。在工作空间的 `File` tab 中，将鼠标定位到 `gizmo Files` 上，点击鼠标右键，将出现如



下图所示的上下文菜单，选择 Add Files，然后使用出现的文件浏览器选择 cobble.c 文件，如图 3.16。



图 3.16 工程上下文菜单

在 File tab 中打开对象模块文件夹 (Object Modules folder)，将显示源文件名以及由它们建造的目标文件，如图 3.17。

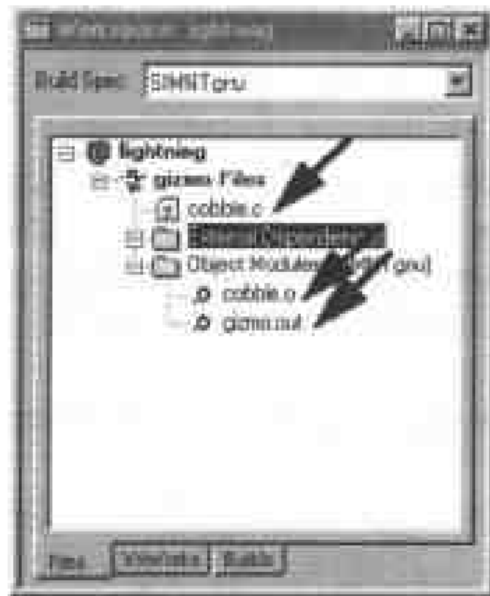


图 3.17 工作空间 File tab



当建造一个工程时，Tornado 工程工具能够自动生成一个目标文件 `projectName.out`，它包含了该可下载应用工程中的所有对象模块，用于同时下载所有目标模块。

3.3.4 建造工程

在用户创建工程之前，先预览默认的设置。在 `Workspace` 窗口中选择 `Builds` tab，打开 `gizmo Builds` 文件夹，双击默认创建名（例如 `SIMNTgnu`）。

当创建属性页出现时，用户可以预览默认的 `makefile` 规则和宏，以及用于建造的编译器、汇编器和链接器的选项设置。

当显示 C/C++ 编译器页时，我们将注意到默认设置时选择了包含调试信息的检查框。这种设置保证了当工程带调试信息编译时，优化选项关闭，如图 3.18。

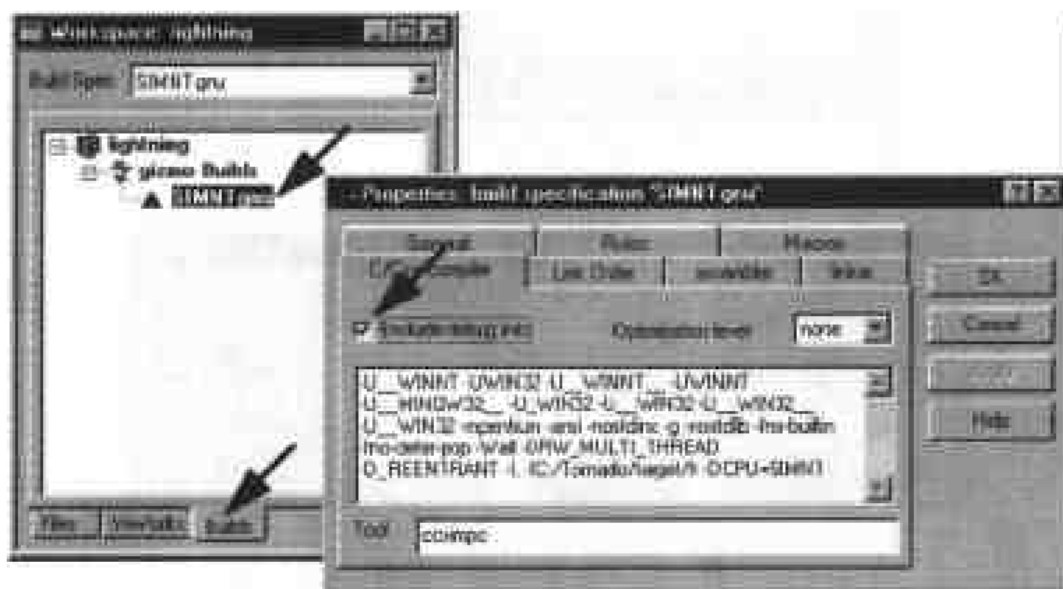


图 3.18 创建属性页

点击 `Cancel` 按钮关闭该属性页。

从上下文菜单中选择“Build 'gizmo.out'”，建造工程。选项“Build 'projectName.out'”将把工程中的所有模块建造成一个单一的、部分链接的模块，用于下载整个工程。如图 3.19。

Tornado 在建造工程之前，将弹出“Dependencies”对话框，提醒我们 `cobble.c` 的 `makefile dependencies` 没有生成，如图 3.20。

点击 `OK`，继续进行。

Tornado 将生成 `makefile dependencies` 文件并开始建造整个工程。

如果 Tornado 找到外部依赖文件，它们将自动添加到工作空间 `File tab` 中的 `External Dependencies` 文件中。



建造输出窗口将显示建造过程中产生的所有错误和警告。在本例中，编译器将发现一个多余的变量，如图 3.21。

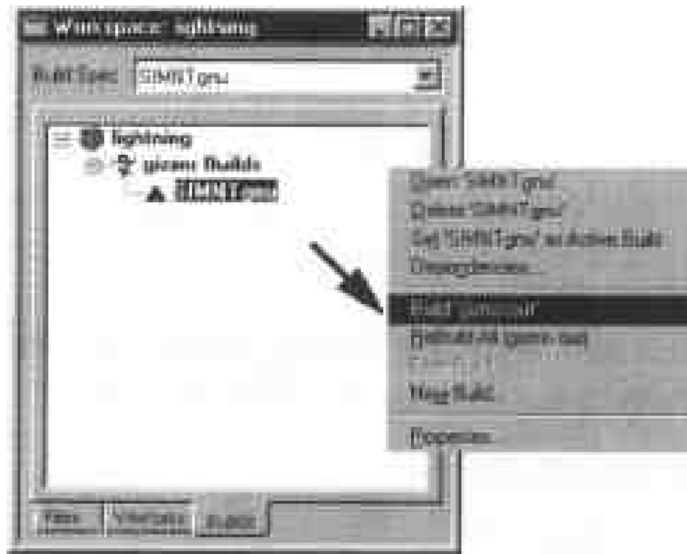


图 3.19 建造工程上下文菜单

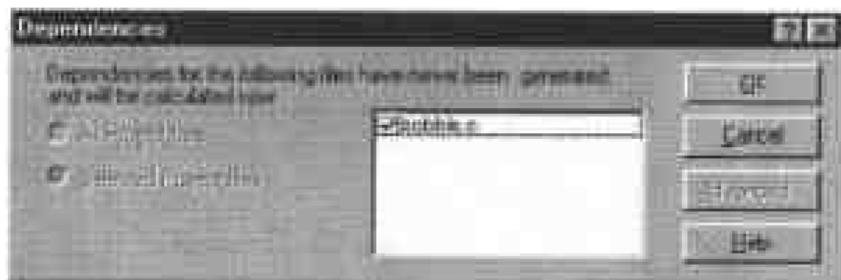


图 3.20 Dependencies 对话框

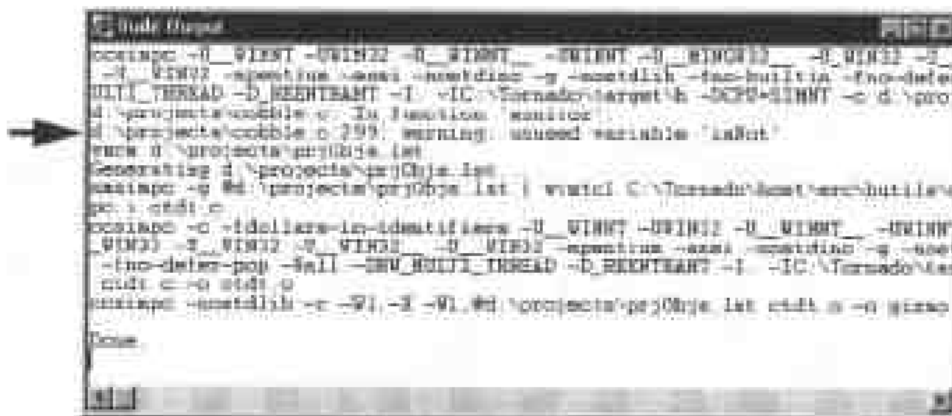


图 3.21 建造输出窗口



当双击一个错误或警告信息时，编辑器将打开对应的源文件，并且一个上下文指针将指向对应的行，如图 3.22。

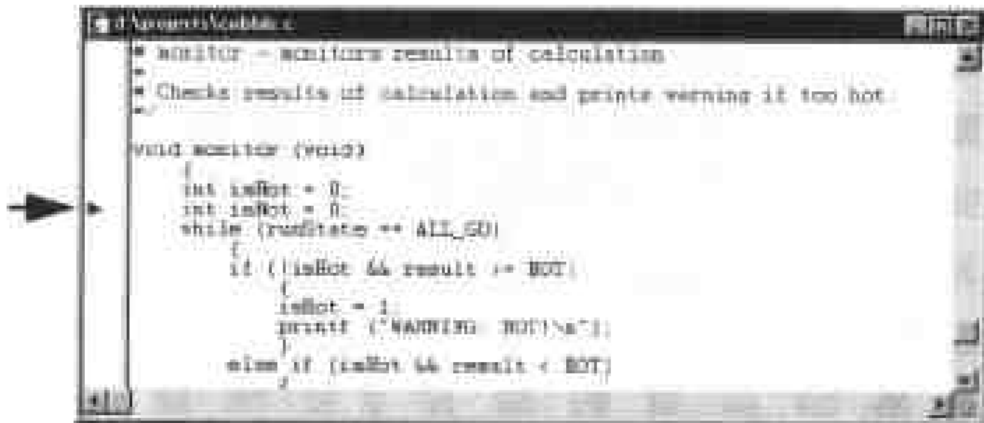


图 3.22 修改源文件上下文指针

我们删除该行，使用 **TRL+S** 或 **File>Save** 保存文件，重新建造工程。Tornado 编辑器提供了标准 Windows 文本操纵能力，遗憾的是不支持汉字输入编辑！

关闭建造输出窗口。

3.3.5 下载工程到目标仿真器

工作空间的 **File** tab 同样提供了目标文件下载功能。下载 **gizmo.out** 时系统同时提示我们启动目标仿真器。在工作空间的 **File** tab 中，选择工程文件夹，鼠标定位到 **gizmo.out** 文件上，点击鼠标右键，在上下文菜单中选择“**Download 'gizmo.out'**”选项，如图 3.23。

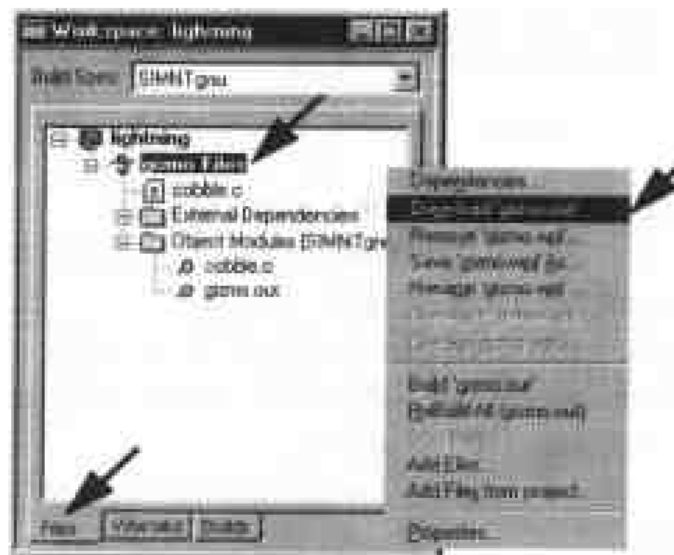


图 3.23 上下文菜单中的下载目标文件项



如图 3.24, Tornado 将提示我们启动目标仿真器。



图 3.24 提示启动目标仿真器对话框

点击 Yes, 继续进行。

将出现 VxSim-Lite Launch 对话框, 如图 3.25。

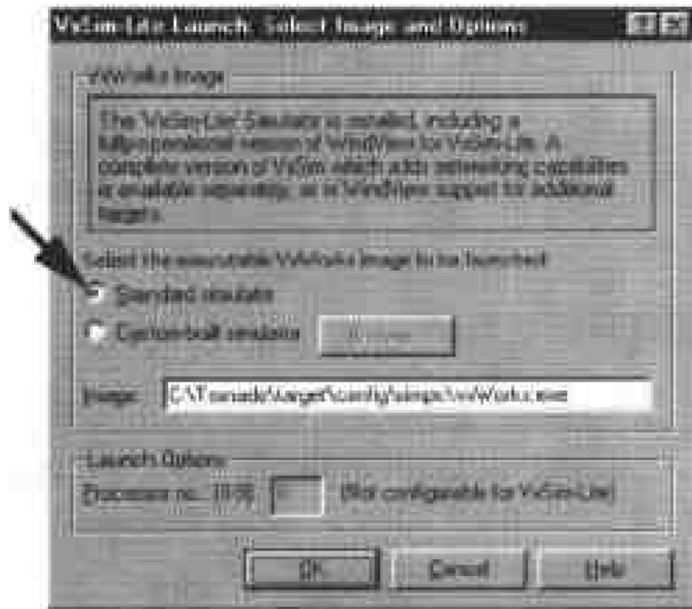


图 3.25 VxSim-Lite Launch 对话框

如上图选择 Standard Simulator, 点击 OK, 启动目标仿真器。

目标仿真器窗口打开, Tornado 提醒我们启动一个目标服务器。如图 3.26。点击 VxSim-Lite Launch 对话框中的 OK 按钮, 继续下一步。

正如前面所述, 目标服务器管理主机工具如调试器等与目标机间的所有通信。目标服务器的名字形式是 `targetName@hostName`。在本例中, 目标机的名字是 `vxsim` (集成的目标机仿真器默认名字), 开发主机的名字是 `badger`。

注意: 某一时刻, 只能有一个集成的目标仿真器运行。(关闭 VxWorks Simulator 窗口, 也就停止了仿真器的运行)。

如图 3.27, 目标服务器的名字将显示在 Tornado launch 工具栏中:

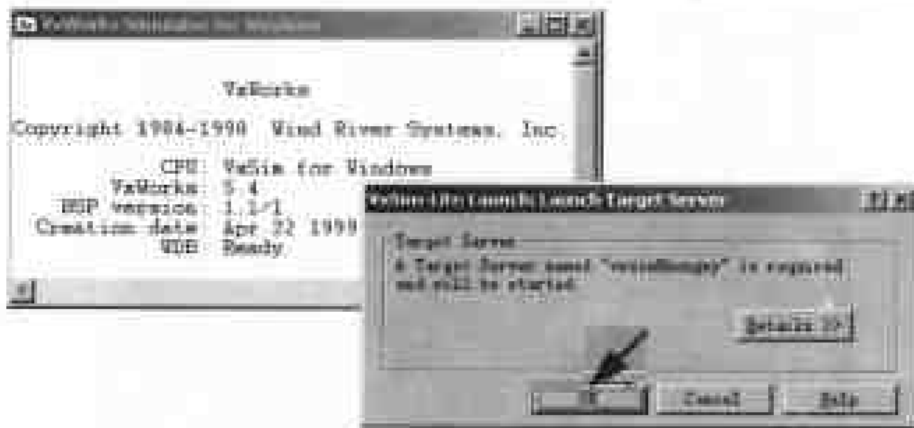


图 3.26 启动一个目标服务器



图 3.27 Tornado launch 工具栏及目标服务器的名字

这个工具栏包括启动 Tornado 工具的按钮，如 browser、shell 和 debugger。目标机启动后，这些按钮有效。

3.3.6 在 Tornado Shell 下运行应用

在运行应用之前，首先启动并配置 Tornado debugger 会有所帮助，这样 debugger 能够自动地对任何程序异常产生反应。

Tornado debugger (CrossWind) 同时支持图形和命令行两种调试接口。最常用的调试行为，例如设置断点，控制程序运行，都可以通过点击可视化图标接口直接完成。同时，程序列表和数据观察窗口对于应用的关键部分提供及时的可视内容。

配置调试器：从 Tornado 主窗口中选择 Tools>Options>Debugger。Options 对话框出现。如图 3.28。

在 Auto attach to tasks 下选择 Always appears。这种设置使得，当一个异常发生时，调试器就可以自动地和任务关联。点击 OK 继续进行。

使用在 Tornado launch 工具栏中的 debugger 按钮启动调试器。几秒后，在 Tornado 主窗口底部的状态行显示调试器正在运行，如图 3.29。

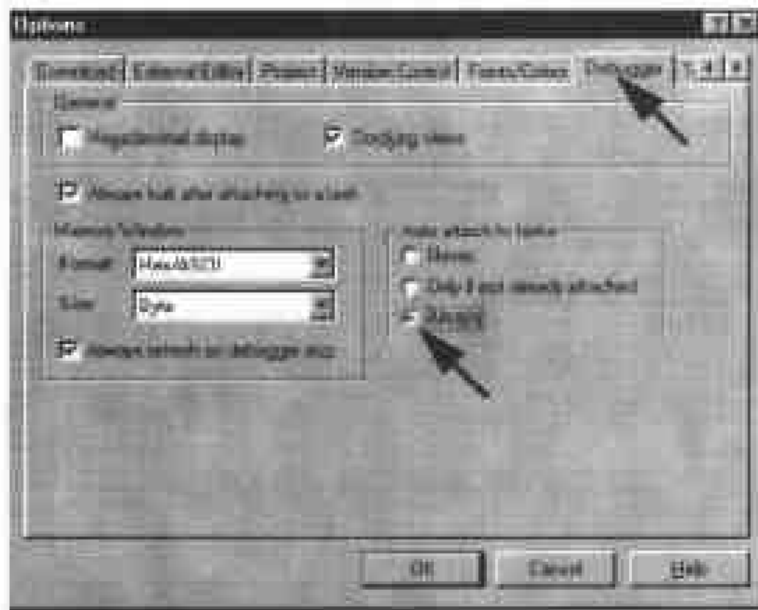


图 3.28 Tornado 工具配置对话框



图 3.29 Tornado 主窗口底部的状态行

Tornado shell (也就是 WindSh) 是一个 C 语言命令解释器。它允许在 shell 命令行中调用下载到目标机上的任何程序。Shell 自身同时提供了一套用于任务管理、访问系统信息、调试等命令。我们可以在 shell 命令行中运行程序。

点击 shell 按钮 ，启动 shell。

如图 3.30，当 shell 窗口打开后，在命令行中输入主程序的名字 progStart，然后输入回车，运行应用。

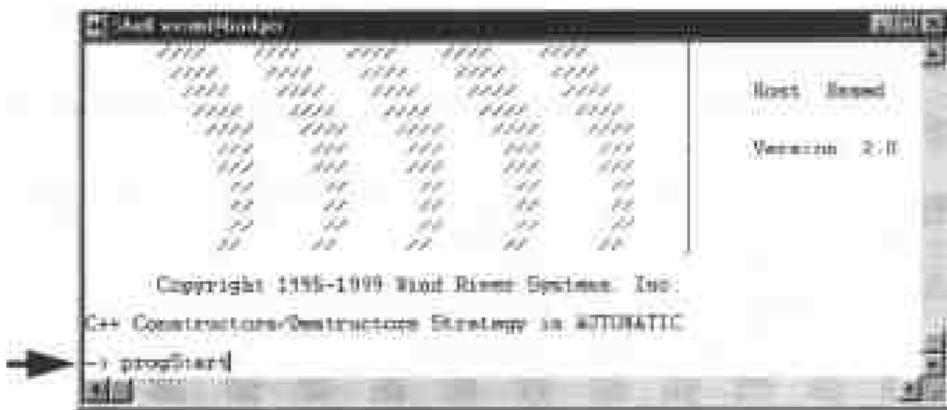




图 3.30 shell 窗口



3.3.7 检查目标内存消费

Tornado browser 是一个系统对象浏览器，是 Tornado shell 的一个图形化的伙伴助手。它可以提供显示工具，来监督目标系统状态，包括活动任务的概要、内存分配等等。

点击 Tornado launch 工具栏上的 browser 按钮  启动 browser。

当 browser 窗口出现后，从下拉菜单中选择内存使用 (Memory Usage)，点击周期刷新按钮 (periodic refresh) 。系统将周期地 (周期约为几秒) 刷新显示。

我们将看到 cobble.c 非常消耗内存，如图 3.31。



图 3.31 Browser 窗口

当关闭该窗口时，也就停止了 browser。

3.3.8 检查任务活动

WindView 是实时应用的逻辑分析仪，它是一个动态可视工具，可以提供上下文切换信息，以及导致发生这些交换的事件，还有与信号量、消息队列和看门狗计时器等对象有关的信息。

在 Tornado Launch 工具栏中点击 WindView 按钮，可以显示 WindView 控制窗口，同时将出现 WindView Collection Configuration 对话框，如图 3.32。从下拉列表中选择 Addition Instrumentation，然后点击 OK。

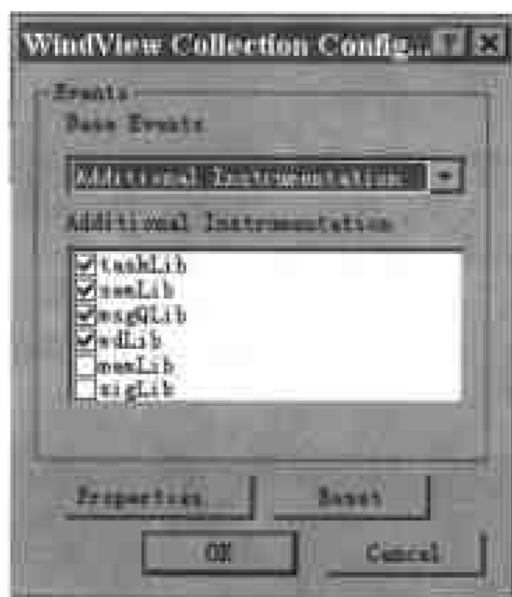




图 3.32 WindView Collection Configuration 对话框

如图 3.33, 在 WindView 控制窗口中点击 Go 按钮 , 开始数据收集。等待几秒钟后, 在 WindView 控制窗口中点击 Update 按钮 , 更新数据收集信息。

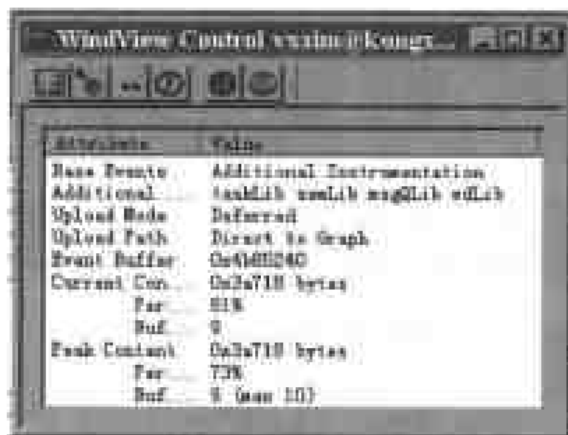



图 3.33 WindView 控制窗口

一旦收集到大约 50%的数据, 点击 Stop 按钮 , 结束数据收集。在这过程中, 可能会发现主机性能不太稳定, 这主要是由采样程序在仿真器上运行时产生的一些错误行为引起。

从目标仿真器向主机上载 WindView 数据之前, 在 Shell 命令行中输入 progStop, 停止采样程序运行。

然后点击 Upload 按钮上载数据。当上载数据完成, 将出现一个视图, 并有一个消息框



提示上传结束，点击 OK 继续。

当最大化视图窗口时，显示的数据如图 3.34 所示。

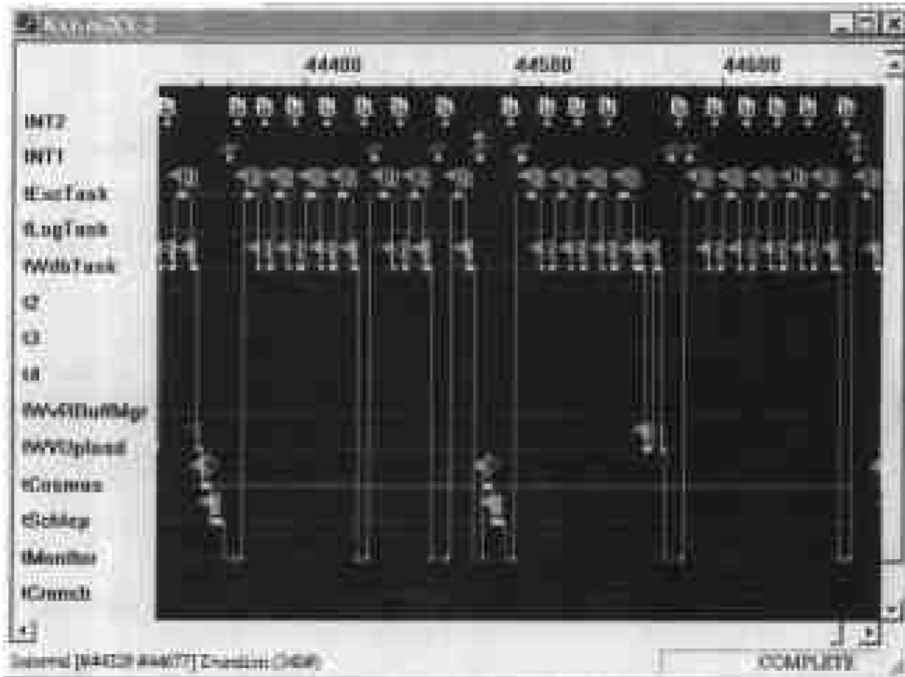





图 3.34 WindView 视图

使用 WindView 工具栏上的 zoom 按钮 ， 可以放大或缩小显示的范围；视图底部的滚动条可以显示数据的其余部分。（zoom 100% 按钮  一次可以显示所有的数据）。旗形图标指示信号量的释放和获取（give/take），水平线指示任务状态（执行、挂起、就绪等）。关于 WindView 图标的更多信息可以阅读帮助信息：

Help>WindView Help>Legend.

在视图中，我们可以注意到，处理数据和从链接链删除节点的任务 tCrunch 从没有运行过。

关闭视图窗口，在出现的提示保存数据的对话框（Save changes to hostName?）中点击 No。关闭 WindView 控制窗口。

3.3.9 固定任务优先级并找出错误

双击工作空间 File tab 中的 cobble.c 文件，在编辑器中打开源程序。找到 progStart() 函数，我们发现任务 tCrunch 分配到的优先级较低，仅为 240，低于任务 tMonitor 的优先级（230），这将导致前者从不会得到运行的机会。



因此，数据不会得到处理，节点也不会从链接链中删除！

编辑源文件，交换这两个任务间的优先级：

- (1) 改变调用 tCrunch 的 taskSpawn()函数的参数 240 为 230；
- (2) 改变调用 tMonitor 的 taskSpawn()函数的参数 230 为 240。

然后保存文件。

使用工作空间 File tab 中的上下文菜单，选择 ReBuild All (gizmo.out) 选项，重新建造工程。建造完成后关闭建造输出窗口。

然后使用上下文菜单中的 Download 'gizmo.out' 选项下载工程到目标仿真器。

注意

不要忘记下载新的 gizmo.out 到目标机！并且，如果已经停止了 debugger，点击 debugger 按钮 重新启动它。

在 Tornado shell 中输入命令 progStart 重新启动程序。

几秒钟后，目标集仿真器窗口将显示应用的输出，同时 shell 窗口显示发生了一个除法错误，如图 3.35。



图 3.35 程序运行状态信息

另外，debugger 自动打开编辑窗口，并在发生错误的 crunch()函数相应的行做出标记，如图 3.36。



图 3.36 debugger 自动打开编辑窗口并定位出错行

在处理该错误之前，我们需要停止程序运行，清除内存。

在 shell 窗口中输入 reboot 命令，然后回车。将出现一个提示目标连接丢失并且 debugger 停止的对话框，点击 OK 确认。

一旦目标仿真器重新启动，在 shell 窗口中使用 i 命令，可以看到正在运行的 VxWorks 系统任务，如图 3.37。

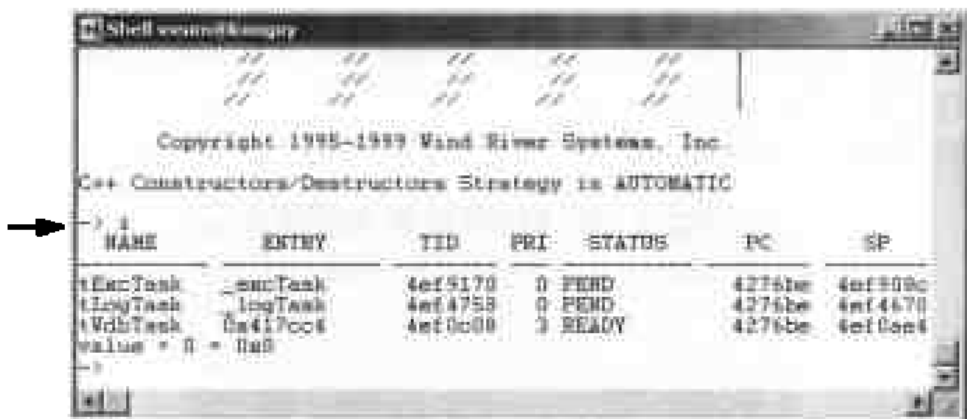


图 3.37 shell 窗口显示 VxWorks 系统任务

3.3.10 固定最后一个错误

找到 crunch()函数中产生除数为 0 的源代码，注释掉该行代码，然后保存文件。按照前几节讲述的方法，重新建造工程目标码 gizmo.out，下载并运行应用，并且启动调试器。

启动 WindView，点击 Go 按钮，开始数据收集，合适时刻停止数据收集，点击 Upload 按钮，上载收到的数据。上载完毕，将出现提示消息框。关闭消息框，收集到的数据将以视



图方式可见，最大化视图窗口，利用滚动条，观察所有数据。我们将会看到包括 tCrunch 在内的所有任务按照一种有序的方式运行，如图 3.38。

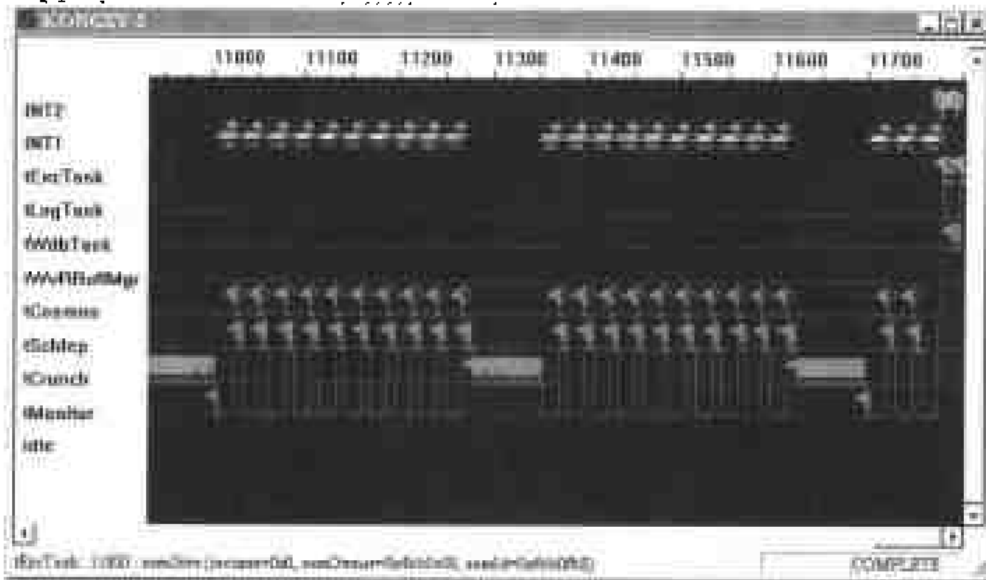


图 3.38 WindView 视图窗口

我们重新察看内存使用情况。首先启动，在下拉列表中选择 Memory Usage，点击 browser 周期刷新按钮。每隔几秒显示将被刷新。

现在我们将看到，内存的消耗将在一个常量的范围内浮动，如图 3.39。



图 3.39 Browser 窗口显示的内存消耗



而且，仿真器窗口不断有信息输出，显示出程序运行的非常忙碌，而且运行良好。这些输出信息并且告诉我们，程序当前是“hot”还是“OK”，如图 3.40。

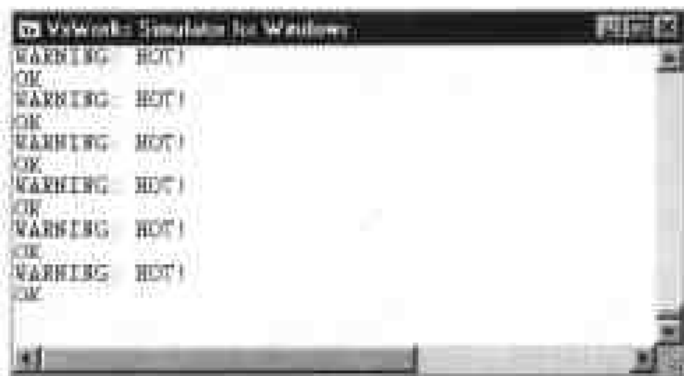


图 3.40 仿真器窗口输出

在 shell 命令行中输入 `progStop` 终止程序运行。

3.4 Tornado1.0.1 下的工程开发

前面我们已经提到，Tornado1.0.1 的开发能力也很强大，最大的不足在于缺乏工程管理能力：不能自动生成 `makefile` 文件。而编写 `makefile` 文件，需要编程者对 GNU Make 的用法有较深的理解，这给稍具规模的软件开发带来不便。这一节我们介绍在 Tornado1.0.1 开发环境下如何进行项目开发，主要包括如何分解工程、编写工程的 `makefile` 文件，GNU Make 的使用等方面的知识。

`make` 是所有想在使用 GNU 作为开发工具的系统（Unix、Linux 等）上编程的用户必须掌握的工具。如果仅仅写一些学习程序，可能不会用到 `make`；但是为任何稍具规模的工程项目而开发的程序，很难离开 `make`。

3.4.1 分解工程

首先介绍为什么要将 C 源代码分离成几个合理的独立文件，什么时候需要分，怎么才能分得合理。

■ 为何要分解工程

首先，将工程分解成多个文件的好处在哪里呢？

这样做看起来像把事情弄的复杂无比：又要头（`header`）文件，又要外部（`extern`）声明，而且如果需要查找一个变量，要在更多的文件里搜索。



但是有更有力的理由支持把一个工程分解成多个小块：

- 如果不进行分解，当改动一行代码，编译器需要全部重新编译所有代码，以生成一个新的可执行文件，这可能需要较长的时间（取决于代码的长度、编译器的性能和机器的速度等因素）。但是如果工程是分开在几个小文件里，当改动其中一个文件的时候，别的源文件的目标文件（object files）已经存在，所以不需要重新编译它们。所需要做的只是重新编译改动过的那个文件，然后重新链接所有的目标文件。在大型的工程中，这意味着从很长的（几分钟到几小时）重新编译缩短为几十秒的简单调整。
- 只要按照一定的规则，将一个工程分解成多个小文件可以更加容易的找到一段代码。很简单，根据代码的作用把代码分解到不同的文件里，并选择适当的名字（有意义的，能表达代码作用的，并且易记的）。当要看一段代码时，就可以准确的知道在那个文件中去寻找它。
- 从很多工程文件生成一个程序库（Library）比从一个单一的大目标文件生成要好的多。当然，实际上这是否真是一个优势则是由所用的系统来决定的。但是当使用 `cc/ld`（一个 GNU C 编译 / 链接器）把一个程序包链接到一个程序时，在链接的过程中，它会尝试不去链接没有使用到的部分。但它每次只能从程序包中把一个完整的目标文件排除在外。因此如果引用了一个程序包中某一个目标文件中任何一个符号，那么这个目标文件整个都会被链接进来。但是如果一个程序包经过非常充分的分解，那么经链接后，得到的可执行文件会比从一个大目标文件组成的程序包链接得到的文件小得多。
- 因为分解后的程序是经过模块化的，文件之间的共享部分被减到最少，那就有很多好处——可以很容易的追踪到 Bug，这些模块经常是可以用在其他的工程里的，同时其他人也可以更容易的理解的一段代码是干什么的等等。

■ 何时分解工程

很明显，把任何东西都分解是不合理的。像“世界，你们好”这样的简单程序根本就不能分，因为实在也没什么可分的。把用于测试用的小程序分解也是没什么意思的。但一般来说，当分解工程有助于分工开发、扩展和可读性的时候，都应该采取它。在大多数的情况下，这都是适用的。（所谓“世界，你们好”，既“hello world”，只是一个介绍一种编程语言时惯用的范例程序，它会在屏幕上显示一行“hello world”，是最简单的程序。）

如果需要开发一个相当大的工程，在开始前，应该考虑一下将如何实现它，并且生成几个文件（用适当的名字）来存放代码。当然，在工程开发的过程中，我们也可以建立新的文件，但如果这么做的话，说明我们可能改变了当初的想法，应该考虑是否需要整体结构也进行相应的调整。

对于中型的工程，当然也可以采用上述方法，但也可以直接编码，当程序代码多到难以管理的时候再把它们分解成不同的文件。但是就经验而言，开始时形成一个大概的方案，并



且尽量遵从它，或在开发过程中，随着程序的需要而修改，会使开发变得更加容易。

■ 怎样分解工程

如何分解工程，不同的人有不同的看法。这会涉及到有关编码风格的问题，大家从来就没有停止过在这个问题上的争论。在这里给出的仅作为一些建议：

- ◆ 不要用一个 header 文件指向多个源码文件（例外：程序包的 header 文件）。用一个 header 定义一个源码文件的方式会更有效，也更容易查寻。否则改变一个源文件的结构（并且它的 header 文件）就必须重新编译好几个文件。
- ◆ 如果可以的话，完全可以用超过一个的 header 文件来指向同一个源码文件。有时将不可公开调用的函数原型，类型定义等等，从它们的 C 源码文件中分离出来是非常有用的。使用一个 header 文件存放公开符号，用另一个存放私人符号，这意味着如果改变了这个源码文件的内部结构，就可以只是重新编译它，而不需要重新编译那些使用它的公开 header 文件的其他的源文件。
- ◆ 不要在多个 header 文件中重复定义信息。如果需要，在其中一个 header 文件里 #include 另一个，但是不要重复输入相同的 header 信息两次。这是因为如果以后改变了这个信息，就只需要把它改变一次，不用搜索并改变另外一个重复的信息。
- ◆ 在每一个源码文件里，#include 那些声明了源码文件中的符号的所有 header 文件。这样一来，在源码文件和 header 文件对某些函数做出的矛盾声明可以比较容易的被编译器发现。

■ 对于常见错误的解决

➤ 定义符 (Identifier) 在源码文件中的矛盾

在 C 里，变量和函数的默认状态是公用的。因此，任何 C 源码文件都可以引用存在于其他源码文件中的全局 (global) 函数和全局变量，即使这个文件没有那个变量或函数的声明或原型。因此必须保证在不同的两个文件里不能用同一个符号名称，否则会有链接错误或者在编译时会有警告。

一种避免这种错误的方法是在公用的符号前加上跟其所在源文件有关的前缀。比如：所有在 gfx.c 里的函数都加上前缀“gfx_”。如果在子程序中非常小心，使用有意义的函数名称，并且不是过分使用全局变量，当然这根本就不是问题。

要防止一个符号在它被定义的源文件以外被看到，可在它的定义前加上关键字“static” (VxWorks 中可以用“LOCAL”)。这对只在一个文件内部使用，其他文件都不会用到的简单函数是很有用的。这将有助于提高系统的稳定性和可靠性。



➤ 多次定义的符号

header 文件会被逐字的替换到源文件里#include 的位置的。因此，如果 header 文件被#include 到一个以上的源文件里，这个 header 文件中所有的定义就会出现在每一个有关的源码文件里。这会使它们里的符号被定义一次以上，从而出现链接错误（见上）。

解决方法：不要在 header 文件里定义变量。只需要在 header 文件里声明它们然后在适当的 C 源码文件（应该 #include 那个 header 文件的那个）里定义它们（一次）。

对于初学者来说，定义和声明是很容易混淆的。声明的作用是告诉编译器其所声明的符号应该存在，并且要有所指定的类型。但是，它并不会使编译器分配贮存空间。而定义的做法是要求编译器分配贮存空间。当做一个声明而不是做定义的时候，在声明前放一个关键字“extern”。

例如，假设有一个叫“counter”的变量，如果想让它成为公用的，我们在一个源码程序（只在一个里面）的开始定义它：“int counter”，再在相关的 header 文件里声明它：

```
extern int counter;
```

或

```
IMPORT int counter;
```

函数原型里隐含着 extern 和 IMPORT 的意思，所以不需顾虑这个问题。

➤ 重复定义，重复声明，矛盾类型

请考虑如果在一个 C 源码文件中#include 两个文件 a.h 和 b.h，而 a.h 又 #include 了 b.h 文件（原因是 b.h 文件定义了一些 a.h 需要的类型），会发生什么事呢？

这时该 C 源码文件 #include 了 b.h 两次。因此每一个在 b.h 中的#define 都发生了两次，每一个声明发生了两次等等。理论上，因为它们是完全一样的拷贝，所以应该没有什么问题，但在实际应用上，这是不符合 C 的语法的，可能在编译时出现错误，或至少是警告。

解决的方法是要确定每一个 header 文件在任一个源码文件中只被包含了一次。一般是用预处理器来达到这个目的的。当进入每一个 header 文件时，为这个 header 文件#define 一个宏指令。只有在这个宏指令没有被定义的前提下，我们才真正使用该 header 文件的主体。在实际应用上，只要简单的把下面一段码放在每一个 header 文件的开始部分：

```
#ifndef FILENAME_H  
#define FILENAME_H
```

然后把下面一行码放在最后：

```
#endif
```

用 header 文件的文件名（大写的）代替上面的 FILENAME_H，用底线代替文件名中



的点。有些人喜欢在 `#endif` 加上注释来提醒他们这个 `#endif` 指的是什么。例如：

```
#endif /* #ifndef FILENAME_H */
```

这只是各人的风格不同，无伤大雅。

只需要在那些有编译错误的 header 文件中加入这个技巧，但在所有的 header 文件中都加入也没什么损失，毕竟这是个好习惯。

■ 重新编译一个多文件工程

清楚的区别编译和链接是很重要的。编译器使用源码文件来产生某种形式的目标文件（object files）。在这个过程中，外部的符号参考并没有被解释或替换。然后我们使用链接器来链接这些目标文件和一些标准的程序包再加上指定的程序包，最后链接生成一个可执行程序文件。

在这个阶段，将解释一个目标文件中对别的文件中的符号的引用，并报告不能被解释的引用，一般是以错误信息的形式报告出来。

基本的步骤就应该是，把源码文件一个一个的编译成目标文件的格式，最后把所有的目标文件加上需要的程序包链接成一个可执行文件。

具体怎么做是由使用的编译器决定的。这里只给出 `cc`（GNU C 编译器）（对 x86 目标机而言是 `cc386`）的有关命令，有些可能对非 `cc` 编译器也适用。

`cc` 是一个多目标的工具。它在需要的时候调用其他的元件（预处理程序、编译器、组合程序、链接器）。具体的哪些元件被调用取决于输入文件的类型和传递给它的开关参数。

一般来说，如果只给它 C 源码文件，它将预处理、编译、组合所有的文件，然后把所得的目标文件链接成一个可执行文件（一般生成的文件被命名为 `xxx.o`），但是这样做会破坏很多把一个项目分解成多个文件所得到的好处。

如果给它一个 `-c` 开关，`cc` 只把给它的文件编译成目标文件，用源码文件的文件名命名但把其后缀由 `“.c”` 或 `“.cc”` 变成 `“.o”`。

如果给它的是一列目标文件，`cc` 会把它们链接成可执行文件，默认文件名是 `xxx.out`。可以改变默认名，用开关 `-o` 后跟指定的文件名。

因此，当改变了一个源码文件后，需要重新编译它：

```
cc -c filename.c
```

然后重新链接的项目：

```
cc -o exec_filename *.o
```

如果改变了一个 header 文件，需要重新编译所有 `#include` 过这个文件的源码文件，可以用



```
cc -c file1.c file2.c file3.c
```

然后像以前一样链接。

cc 工具的使用请参考 Tornado 在线帮助文档《GNU Toolkit User's Guide》。

当然这么做是很繁琐的，GNU Make 工具使这个步骤变得简单。

3.4.2 GNU Make

■ 基本 makefile 结构

GNU Make 的主要工作是读进一个文本文件：makefile。这个文件里主要是有关哪些文件（‘target’目的文件）是从哪些别的文件（‘dependencies’依靠文件）中产生的，用什么命令来进行这个产生过程。有了这些信息，make 会检查磁盘上的文件，如果目的文件的时间戳（该文件生成或被改动时的时间）至少比它的一个依靠文件旧的话，make 就执行相应的命令，以便更新目的文件（目的文件不一定是最后的可执行文件，它可以是任何一个文件）。

makefile 一般被叫做“makefile”或“Makefile”。当然可以在 make 的命令行指定别的文件名。如果不特别指定，它会寻找“makefile”或“Makefile”，因此使用这两个名字是最简单的。

一个 makefile 主要含有一系列的规则，如下：

```
...  
(tab)<command>  
(tab)<command>  
.
```

例如，考虑以下的 makefile：

```
#=== makefile 开始 ===  
myprog : foo.o bar.o  
cc oo.o bar.o -o myprog  
cc : foo.c foo.h bar.h  
cc -c foo.c -o foo.o  
bar.o : bar.c bar.h  
cc -c bar.c -o bar.o  
#=== makefile 结束 ===
```

这是一个非常基本的 makefile —— make 从最上面开始，把上面第一个目的，“myprog”，做为它的主要目标（一个它需要保证其总是最新的最终目标）。给出的规则说



明只要文件“myprog”比文件“foo.o”或“bar.o”中的任何一个旧，下一行的命令将会被执行。

但是，在检查文件 foo.o 和 bar.o 的时间戳之前，它会往下查找那些把 foo.o 或 bar.o 做为目标文件的规则。它找到的关于 foo.o 的规则，该文件的依靠文件是 foo.c、foo.h 和 bar.h。它从下面再找不到生成这些依靠文件的规则，它就开始检查磁盘上这些依靠文件的时间戳。如果这些文件中任何一个的时间戳比 foo.o 的新，命令

```
cc -o foo.o foo.c
```

将会执行，从而更新文件 foo.o。

接下来对文件 bar.o 做类似的检查，依靠文件在这里是文件 bar.c 和 bar.h。

现在，make 回到“myprog”的规则。如果刚才两个规则中的任何一个被执行，myprog 就需要重建（因为其中一个 .o 文件会比“myprog”新），因此链接命令将被执行。

到此，可以看出使用 make 工具来建立程序的好处——前面所说的所有繁琐的检查步骤都由 make 替我们做了：检查时间戳。

源码文件里一个简单改变都会造成那个文件被重新编译（因为 .o 文件依靠 .c 文件），进而可执行文件被重新链接（因为 .o 文件被改变了）。其实真正的收益是在当改变一个 header 文件的时候——不再需要记住那个源码文件依靠它，因为所有的资料都在 makefile 里。

make 会很轻松的替我们重新编译所有那些因依靠这个 header 文件而改变了的源码文件，如有需要，再进行重新链接。

当然，要确定在 makefile 中所写的规则是正确无误的，只列出那些在源码文件中被 #include 的 header 文件……

■ 编写 make 规则

最明显的（也是最简单的）编写规则的方法是一个一个的查看源码文件，把它们的目标文件做为目的，而 C 源码文件和被它 #include 的 header 文件做为依靠文件。但也要把其他被这些 header 文件 #include 的 header 文件也列为依靠文件，还有那些被包括的文件所包括的文件……，这显然是一个更为庞大的工程！需要容易些的方法。

事实上，在编译每一个源码文件的时候，它实在应该知道应该包括什么样的 header 文件。使用 cc 的时候，用 -M 开关，它会为每一个给它的 C 文件输出一个规则，把目标文件做为目的，而这个 C 文件和所有应该被 #include 的 header 文件将做为依靠文件。注意这个规则会加入所有 header 文件，包括被角括号 (<, >) 和双引号 ("") 所包围的文件。其实我们可以相当肯定系统 header 文件（比如 stdio.h, stdlib.h 等等）不会被更改，如果用 -MM 来代替 -M 传递给 cc，那些用角括号包围的 header 文件将不会被包括（这会节省一些编译时间）。

由 cc 输出的规则不会含有命令部分；可以写入命令或者什么也不写，而让 make 使用



它的隐含的规则（参考下面的内容）。

■ Makefile 变量

上面提到 makefiles 里主要包含一些规则。它们包含的其他的東西是变量定义。

makefile 里的变量就像一个环境变量 (environment variable)。事实上，环境变量在 make 过程中被解释成 make 的变量。这些变量是大小写敏感的，一般使用大写字母。它们可以从几乎任何地方被引用，也可以被用来做很多事情，比如：

- ◆ 储存一个文件名列表。在上面的例子里，生成可执行文件的规则包含一些目标文件名做为依靠。在这个规则的命令行里同样的那些文件被输送给 cc 做为命令参数。如果在这里使用一个变数来贮存所有的目标文件名，加入新的目标文件会变的简单而且较不易出错。
- ◆ 储存可执行文件名。如果开发的项目被用在一个非 cc 的系统里，或者如果想使用一个不同的编译器，必须将所有使用编译器的地方改成用新的编译器名。但是如果使用一个变量来代替编译器名，那么只需要改变一个地方，其他所有地方的命令名就都改变了。
- ◆ 储存编译器选项。假设想给所有的编译命令传递一组相同的选项（例如 -Wall -O -g）；如果把这组选项存入一个变量，那么就可以把这个变量放在所有呼叫编译器的地方。而当要改变选项的时候，只需在一个地方改变这个变量的内容。

要设定一个变量，只要在一行的开始写下这个变量的名字，后面跟一个 = 号，后面再跟要设定的这个变量的值。以后要引用这个变量，写一个 \$ 符号，后面是围在括号里的变量名。比如在下面，把前面的 makefile 利用变量重写一遍：

```
#=== makefile 开始 ===
OBJS = foo.o bar.o
CC = cc386
CFLAGS = -Wall -O -g
myprog : $(OBJS)
$(CC) $(OBJS) -o myprog
foo.o : foo.c foo.h bar.h
$(CC) $(CFLAGS) -c foo.c -o foo.o
bar.o : bar.c bar.h
$(CC) $(CFLAGS) -c bar.c -o bar.o
#=== makefile 结束 ===
```

还有一些设定好的内部变量，它们根据每一个规则内容定义。三个比较有用的变量是 \$@、\$< 和 \$^（这些变量不需要括号括住）。

- ◆ \$@ 扩展成当前规则的目的文件名。



- ◆ `$<` 扩展成依靠列表中的第一个依靠文件。
- ◆ `$^` 扩展成整个依靠的列表（除掉了里面所有重复的文件名）。

利用这些变量，我们可以把上面的 `makefile` 写成：

```
#=== makefile 开始 ===
OBJS = foo.o bar.o
CC = cc
CFLAGS = -Wall -O -g
myprog : $(OBJS)
$(CC) $^ -o $@
foo.o : foo.c foo.h bar.h
$(CC) $(CFLAGS) -c $< -o $@
bar.o : bar.c bar.h
$(CC) $(CFLAGS) -c $< -o $@
#=== makefile 结束 ===
```

还可以用变量做许多其他的事情，特别是当把它们和函数混合使用的时候。如果需要更进一步的了解，请参考 Tornado 在线帮助《GNU Make》手册。

■ 隐含规则（Implicit Rules）

请注意，在上面的例子里，几个产生 `.o` 文件的命令都是一样的。都是从 `.c` 文件和相关文件里产生 `.o` 文件，这是一个标准的步骤。其实 `make` 已经知道怎么做——它有一些叫隐含规则的内置的规则，这些规则告诉它当没有给出某些命令的时候，应该怎么办。

如果把生成 `foo.o` 和 `bar.o` 的命令从它们的规则中删除，`make` 将会查找它的隐含规则，然后会找到一个适当的命令。它的命令会使用一些变量，因此可以按照我们的想法来设定它：它使用变量 `CC` 做为编译器（象我们在前面的例子），并且传递变量 `CFLAGS`（给 C 编译器，C++ 编译器用 `CXXFLAGS`），`CPPFLAGS`（C 预处理选项），`TARGET_ARCH`（现在不用考虑这个），然后它加入选项“-c”，后面跟变量 `$<`（第一个依靠名），然后是旗标“-o”和变量 `$@`（目的文件名）。一个 C 编译的具体命令将会是：

```
$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c $< -o $@
```

当然可以按照我们的需要来定义这些变量。这就是为什么用 `cc` 的 `-M` 或 `-MM` 开关输出的代码可以直接用在一个 `makefile` 里。

■ 假像目的（Phony Targets）

假设项目最后需要产生两个可执行文件。我们的主要目标是产生两个可执行文件，但这两个文件是相互独立的——如果一个文件需要重建，并不影响另一个。可以使用“假像目的”来达到这种效果。一个假象目的跟一个正常的目的几乎是一样的，只是这个目的文件是不存



在的。因此，`make` 总是会假设它需要被生成，当把它的依赖文件更新后，就会执行它的规则里的命令行。

如果在 `makefile` 开始处输入：

```
all : exec1 exec2
```

其中 `exec1` 和 `exec2` 是做为目的的两个可执行文件。`make` 把这个“all”做为它的主要目的，每次执行时都会尝试把“all”更新。但既然这行规则里没有哪个命令来作用在一个叫“all”的实际文件（事实上 `all` 并不会在磁盘上实际产生），所以这个规则并不真的改变“all”的状态。可既然这个文件并不存在，所以 `make` 会尝试更新 `all` 规则，因此就检查它的依靠 `exec1`、`exec2` 是否需要更新，如果需要，就把它们更新，从而达到我们的目的。

假像目的也可以用来描述一组非预设的动作。例如，如果想把所有由 `make` 产生的文件删除，可以在 `makefile` 里设立这样一个规则：

```
veryclean :  
rm *.o  
rm myprog
```

前提是没有其他的规则依靠这个“`veryclean`”目的，它将永远不会被执行。但是，如果明确的使用命令

```
make veryclean
```

`make` 会把这个目的做为它的主要目标，执行那些 `rm` 命令。

如果磁盘上存在一个叫 `veryclean` 文件，会发生什么事？这时因为在这个规则里没有任何依靠文件，所以这个目的文件一定是最新的了（所有的依靠文件都已经是最新的了），所以即使用户明确命令 `make` 重新产生它，也不会有任何事情发生。解决方法是标明所有的假象目的（用 `.PHONY`），这就告诉 `make` 不用检查它们是否存在于磁盘上，也不用查找任何隐含规则，直接假设指定的目的需要被更新。在 `makefile` 里加入下面这行包含上面规则的规则：

```
.PHONY : veryclean
```

就可以了。注意，这是一个特殊的 `make` 规则，`make` 知道 `.PHONY` 是一个特殊目的，当然可以在它的依靠里加入想用的任何假象目的，而 `make` 知道它们都是假象目的。

■ 函数（Functions）

`makefile` 里的函数跟它的变量很相似——使用的时候，用一个 `$` 符号跟左括号，函数名，空格后跟一系列由逗号分隔的参数，最后用右括号结束。例如，在 GNU Make 里有一个



叫“wildcard”的函数，它有一个参数，功能是展开成一系列所有符合由其参数描述的文件名，文件间以空格间隔。可以像下面所示使用这个命令：

```
SOURCES = $(wildcard *.c)
```

这行会产生一个所有以“.c”结尾的文件的列表，然后存入变量 SOURCES 里。当然不需要一定要把结果存入一个变量。

另一个有用的函数是 patsubst (pattern substitute, 匹配替换的缩写) 函数。它需要 3 个参数：

- ◆ 第一个是一个需要匹配的式样。
- ◆ 第二个表示用什么来替换它。
- ◆ 第三个是一个需要被处理的由空格分隔的字列。

例如，处理那个经过上面定义后的变量

```
OBJS = $(patsubst %.c,%.o,$(SOURCES))
```

这行将处理所有在 SOURCES 字列中的字（一系列文件名），如果它的结尾是“c”，就用“.o”把“.c”取代。注意这里的 % 符号将匹配一个或多个字符，而它每次所匹配的字串叫做一个“柄”(stem)。在第二个参数里，% 被解读成用第一参数所匹配的那个柄。

■ 其他技巧

我们再解释几个技巧。

(1) 首先，在用于定义变量时，“:=”和“=”两个符号有区别。前者的作用是立即把定义中参考到的函数和变量都展开。如果使用“=”的话，函数和变量参考会留在那儿，就是说改变一个变量的值会导致其他变量的值也被改变。例如：

```
A = foo
```

```
B = $(A)
```

现在 B 是 \$(A)，而 \$(A) 是“foo”。

```
A = bar
```

现在 B 仍然是 \$(A)，但它的值已随着变成“bar”了。

```
B := $(A)
```

现在 B 的值是“bar”。

```
A = foo
```

B 的值仍然是“bar”。

make 会忽略在 # 符号后面内容直到那一行结束的所有文字。



(2) `ifneg...else...endif` 系统是 `makefile` 里让某一部分码有条件的失效 / 有效的工具。`ifeq` 使用两个参数, 如果它们相同, 它把直到 `else` (或者 `endif`, 如果没有 `else` 的话) 的一段码加进 `makefile` 里; 如果不同, 把 `else` 到 `endif` 间的一段码加入 `makefile` (如果有 `else`)。 `ifneq` 的用法刚好相反。

(3) “filter-out” 函数使用两个用空格分开的列表, 它把第二列表中所有的存在于第一列表中的项目删除。用它来处理列表, 把所有已经存在的项目都删除, 而只保留缺少的那些。

■ 一个比较有效的 `makefile`

利用我们现在所学的, 可以建立一个相当有效的 `makefile`。这个 `makefile` 可以完成大部分我们需要的依靠检查, 不用做太大的改变就可直接用在大多数的项目里。

首先我们需要一个基本的 `makefile` 来建我们的程序。我们可以让它搜索当前目录, 找到源码文件, 并且假设它们都是属于我们的项目的, 放进一个叫 `SOURCES` 的变量。这里如果也包含所有的 `*.cc` 文件, 也许会更保险, 因为源码文件可能是 C++ 码的。

```
SOURCES = $(wildcard *.c *.cc)
```

利用 `patsubst`, 可以由源码文件名产生目标文件名, 需要编译出这些目标文件。如果源码文件既有 `.c` 文件, 也有 `.cc` 文件, 则需要使用相嵌的 `patsubst` 函数呼叫:

```
OBJS = $(patsubst %.c,%.o,$(patsubst %.cc,%.o,$(SOURCES)))
```

最里面一层 `patsubst` 的呼叫会对 `.cc` 文件进行后缀替代, 产生的结果被外层的 `patsubst` 呼叫处理, 进行对 `.c` 文件后缀的替代。

现在我们可以设立一个规则来建可执行文件:

```
myprog : $(OBJS)
cc -o myprog $(OBJS)
```

进一步的规则不一定需要, `cc` 已经知道怎么去生成目标文件 (object files)。下面我们可以设定产生依靠信息的规则:

```
depends : $(SOURCES)
cc -M $(SOURCES) > depends
```

在这里如果有一个叫 “depends” 的文件不存在, 或任何一个源码文件比一个已存在的 `depends` 文件新, 那么一个 `depends` 文件会被生成。`depends` 文件将会含有由 `cc` 产生的关于源码文件的规则 (注意 `-M` 开关)。现在要让 `make` 把这些规则当做 `makefile` 文件的一部分。这里使用的技巧很像 C 语言中的 `#include` 系统——要求 `make` 把这个文件 `include` 到 `makefile` 里, 如下:



```
include depends
```

GNU Make 看到这个，检查“depends”目的是否更新了，如果没有，它用给它的命令重新产生 depends 文件。然后它会把这组（新）规则包含进来，继续处理最终目标“myprog”。当看到有关 myprog 的规则，它会检查所有的目标文件是否更新——利用 depends 文件里的规则，当然这些规则现在已经是更新过的了。

这个系统其实效率很低，因为每当一个源码文件被改动，所有的源码文件都要被预处理以产生一个新的“depends”文件。而且它也不是 100%的安全，这是因为当一个 header 文件被改动，依靠信息并不会被更新。但就基本工作来说，它也算相当有用的了。

■ 一个更好的 makefile

这是我设计的一个工程的 makefile，它用于生成一个基于 VxWorks 的窗口系统，该窗口系统目前支持标准 VGA 方式，提供 Win32 编程接口。该工程包含两级目录，根目录下包含以下子目录：

- ◆ 四个源程序目录：server、engine、drivers、demo。
- ◆ 一个输出文件目录 bin。
- ◆ 一个头文件目录 include。

每个源程序目录下包含一个 makefile。下面是根目录下的 makefile:

```
# Makefile - makefile skeleton for win-vga
#
# Copyright (C) 2000 Kongxiangying..
#
#

CPU          =      I80486
TOOL         =      gnu
ADDED_CFLAGS =      -O
#编译器和连接器
CC=cc386
LD=ld386

#工程根目录
TOPDIR = e:/microwin/win-vga
CFLAGS += -Wall -I$(TOPDIR)/include
#头文件目录
INCLDIR += $(TOPDIR)/include

export
#四个源程序子目录
```



```
SUBDIRS = server engine drivers demo

#四个源程序子目录最终目的文件
PROGRAMS = server/server engine/engine drivers/drivers demo/demo

#工程输出目的文件
EXEFILE= $(TOPDIR)/bin/jwin

.PHONY: all $(SUBDIRS)

all: $(SUBDIRS)
all: $(EXEFILE)

$(SUBDIRS):
    $(MAKE) -C $@

$(EXEFILE): $(PROGRAMS)
    $(LD) -o $(EXEFILE) -r $(PROGRAMS)

clean:
    rm -f $(PROGRAMS)
    rm -f `find . -type f -name '*.o' -print`
    rm -f core `find . -type f -name '*.a' -print`
    rm -f core `find . -type f -name '*.so' -print`
    rm -f core `find . -type f -name 'core' -print`

include $(WIND_BASE)/target/h/make/defs.bsp
include $(WIND_BASE)/target/h/make/make.$(CPU)$(TOOL)
include $(WIND_BASE)/target/h/make/defs.$(WIND_HOST_TYPE)
include $(WIND_BASE)/target/h/make/rules.bsp
```

每个源程序子目录下的 makefile 形式相同，这里只给出 dirver 目录的 makefile:

```
TOPDIR = e:/microwin/win-vga
CFLAGS += -I$(TOPDIR)/include/
INCLDIR += $(TOPDIR)/include/
CC=cc386
LD=ld386

SOURCES=$(wildcard *.c)
```



```

OBJS = $(patsubst %.c, %.o, $(SOURCES))

RESULT=drivers

$(RESULT) : $(OBJS) ; $(LD) -o $(RESULT) -r $(OBJS)

clear : rm *.o

#include $(WIND_BASE)/target/h/make/defs.bsp
#include $(WIND_BASE)/target/h/make/make.$(CPU)$(TOOL)
#include $(WIND_BASE)/target/h/make/defs.$(WIND_HOST_TYPE)
#include $(WIND_BASE)/target/h/make/rules.bsp

```

3.4.3 在开发环境中建造工程

利用前面介绍的知识，我们可以分解一个工程、编写相应的文件以及工程的 `makefile` 文件，然后在命令行方式下使用 `make` 命令生成最后的目标文件。Tornado1.0.1 允许我们将写好的 `makefile` 添加到集成开发环境中，像生成 VxWorks 那样生成工程的目标文件。下面介绍有关的步骤。

首先打开 Tornado，点击 **Project** 菜单，如图 3.41。



图 3.41 Project 菜单中的定制 (Customize) 菜单

然后在下拉菜单中选择 **Customize**，将弹出 **Customize Builds** 对话框，如图 3.42。

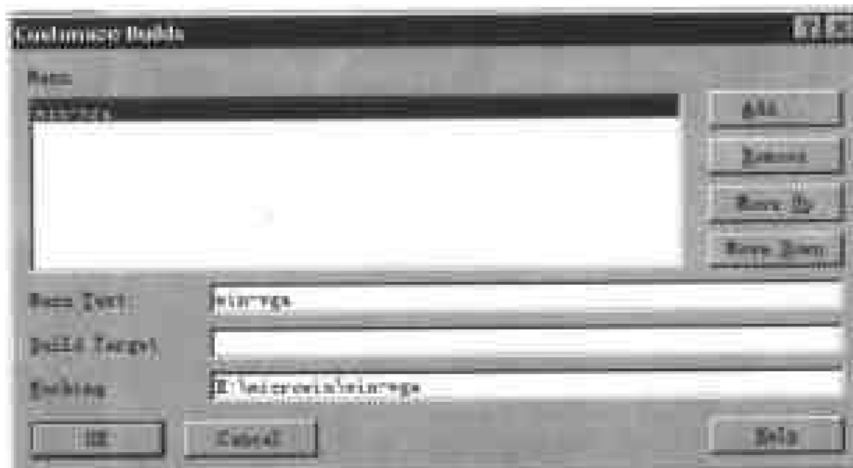


图 3.42 Customize Builds 对话框

然后，点击 Add 按钮，将弹出 Add New Build 对话框，如图 3.43。



图 3.43 Add New Build 对话框

点击 Browse 按钮，如图 3.44，在弹出的 Browse 对话框中选择工程的 makefile 文件所在路径，并选中该 makefile 文件。



图 3.44 选择 makefile 文件



点击打开按钮，Customize Builds 对话框将如下图 3.45 所示。



图 3.45 添加后的 Customize Builds 对话框

我们可以给它起个名字，将 Menu Text 文本框中的 makefile 改为 winvga，可以看到 Menu 中对应的 makefile 也改为 winvga（第二行），如图 3.46。

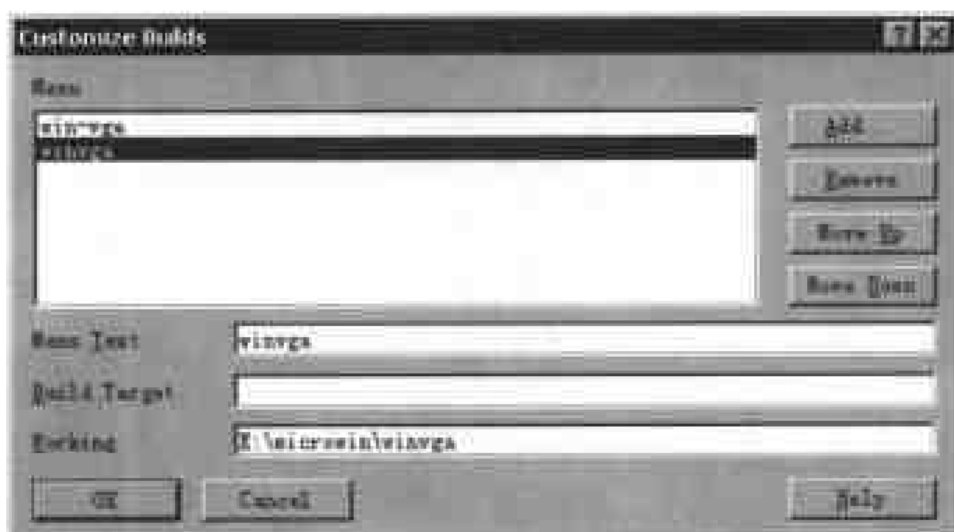


图 3.46 更改名后的 Customize Builds 对话框

点击 OK 按钮确认操作。我们再次打开 Project 菜单，将会发现多了 winvga 一项。选择该项，Tornado 将执行该 makefile 定义的操作，生成工程目标文件，如图 3.47。

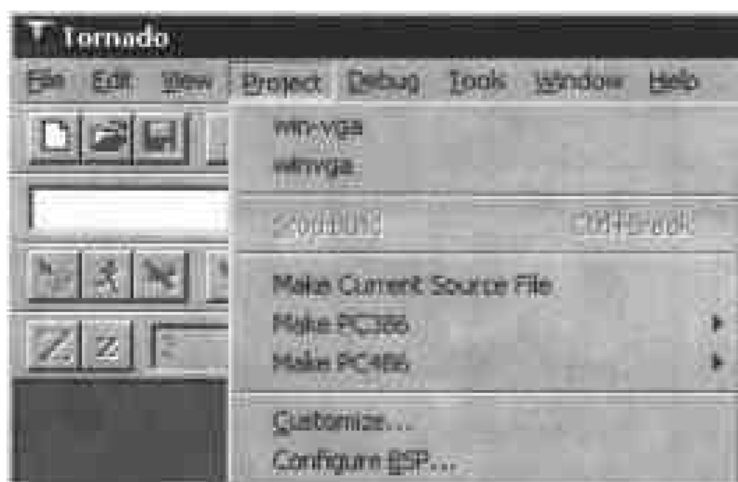


图 3.47 添加后的 Project 菜单

至此，我们简单了解 TornadoII 的使用。后面的章节介绍 VxWorks 编程的有关知识。

多任务内核、任务机制、任务间通信和中断处理机制，这些是 VxWorks 运行环境的核心。这一章和后面的几章将分别加以介绍。

多任务和任务间通信是现代实时操作系统的基石。一个多任务的环境允许将实时应用构造成一套独立的任务集合，每个任务拥有各自的执行线程和自己的系统资源集合，完成不同的功能。

POSIX 实时扩展标准 1003.1b 定义了一套实时操作系统内核功能的接口。为了实现可移植性，VxWorks Wind 内核包含 POSIX 接口和为 VxWorks 专门设计的接口。本章主要讨论 VxWorks 任务和任务编程接口。

4.1 VxWorks 任务

4.1.1 多任务

软件设计时通常将应用划分成独立的、相互作用的程序集合。对于每个程序，当其执行时，我们称之为任务。VxWorks 的任务可以直接或共享访问大多数系统资源，同时拥有足够的分离的上下文来维护各自的控制线程。这些任务共同合作来实现整个系统的功能。

多任务提供一种机制，使得应用可以控制响应（模拟）多重的、离散的现实世界中的事件。VxWorks 实时内核 Wind 提供基本的多任务环境。

在单 CPU 系统中，多任务构造出多个线程并发执行的假象。事实上，系统根据一个调度算法，将 Wind 内核插入到这些任务中执行。每个明显独立的程序称为一个任务。每个任务有自己的上下文（它的 CPU 环境、系统资源等）。任务由系统内核调度运行。在上下文交



换时，任务的上下文保存在任务控制块（TCB）中。一个任务的上下文包括以下内容：

- 任务的执行点、也就是任务的程序计数器。
- CPU 寄存器和浮点计数器。
- 动态变量和函数调用的堆栈。
- 标准输入输出和错误的 I/O 分配。
- 一个延时定时器。
- 一个时间片定时器。
- 内核控制结构。
- 信号处理器。
- 调试和性能监视值。

与 Windows 系统不同，VxWorks 操作系统的内存是线性的，所有代码执行在单一的公共的地址空间内，因而内存地址空间不属于任务上下文。每个任务各自的地址空间需要虚地址到物理地址的转换映射，这仅对使用可选产品 VxVMI 有效。

4.1.2 任务状态转换

任务状态反映任务当前在系统所处的情形。内核负责维护系统中所有任务的当前状态。一个任务从一个状态转变为另一个状态是应用调用内核调用的结果。

当创建时，任务处于挂起（suspended）状态，为使创建的任务进入就绪（ready）状态，必须要激活（Activation）该任务。激活阶段相当快，因此应用程序先创建任务，并在适当的时候将其激活。另一种方法是使用发起（spawning）原语，类似于 UNIX 中的 fork 调用。它使用一个单一的函数调用，创建并激活一个任务。任务可以在任何状态下被删除。

Wind 内核状态说明如表 4.1。对应的状态转换如图 4.1 所示，任务队列示如图 4.2。

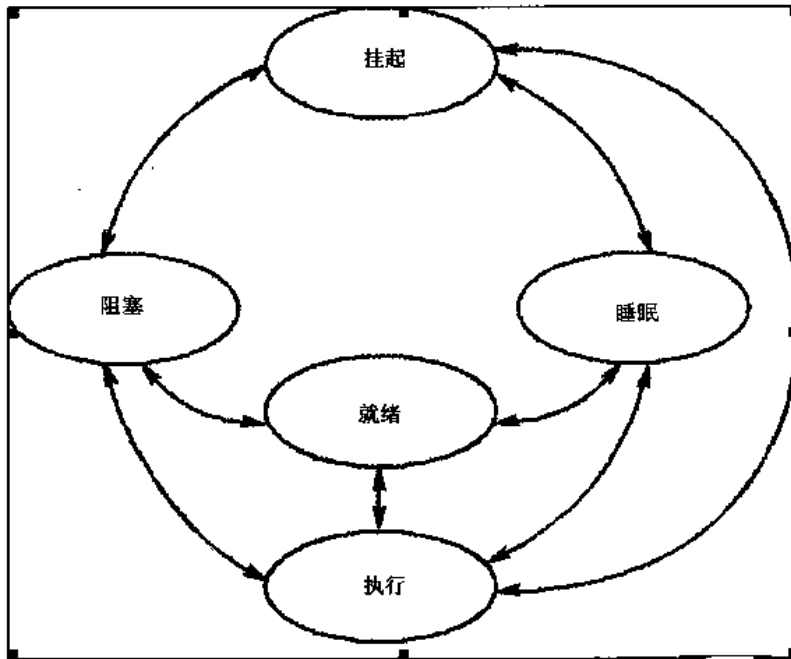
表 4.1 Wind 内核状态

状态符号	描述
就绪 (READY)	处于这种状态的任务除了等待 CPU 外，不需要等待其他资源
阻塞 (PEND)	由于一些资源不可用而阻塞的任务状态
睡眠 (DELAY)	处于睡眠的任务状态
挂起 (SUSPEND)	这种任务状态不能执行。主要用于调试。不会约束状态转换，仅仅约束任务的执行，因此，pended-suspended 任务仍然可以解锁，delayed-suspended 任务仍然可以唤醒
DELAY+S	既处于睡眠又处于挂起的任务状态



续表

状态符号	描述
PEND+S	既处于阻塞又处于挂起的任务状态
PEND+T	带有超时值处于阻塞的任务状态
PEND+S+T	带有超时值处于阻塞又处于挂起的任务状态
State+T	处于 state 带有一个继承优先级的任务状态



ready	→	pending	semTake()/msgQReceive()
ready	→	delayed	taskDelay()
ready	→	suspended	taskSuspend()
pending	→	ready	semGive()/msgQSend()
pending	→	suspended	taskSuspended()
delayed	→	ready	延时已到
delayed	→	suspended	taskSuspend()
suspended	→	ready	taskResume()/taskActivate()
suspended	→	pending	taskResume()
suspended	→	delayed	taskResume()

图 4.1 VxWorks 任务状态转换图

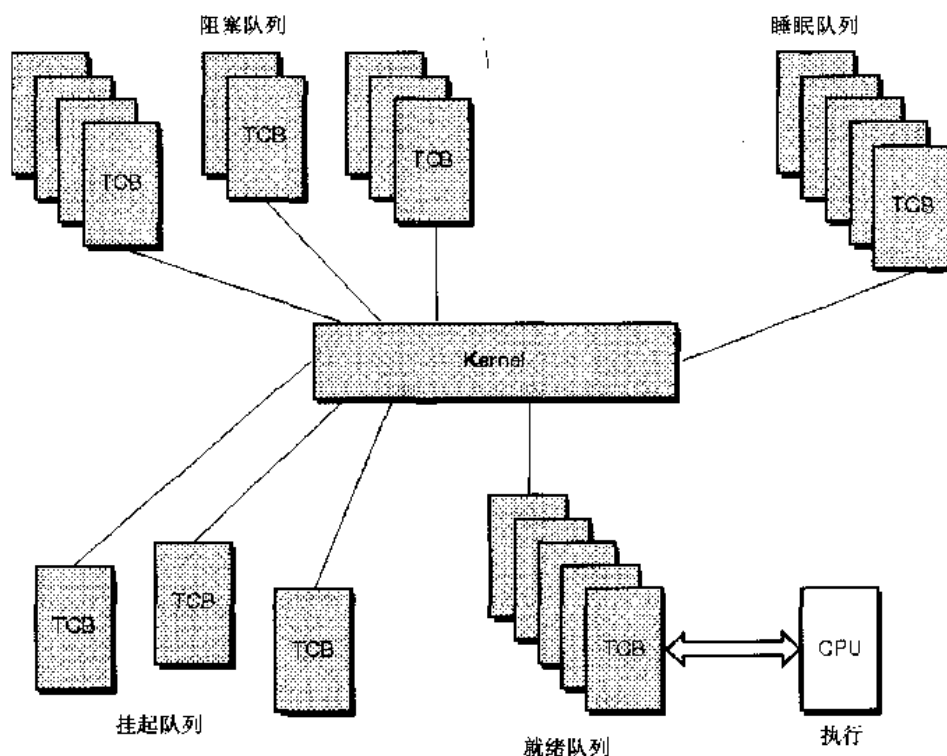


图 4.2 VxWorks 任务状态队列

4.1.3 WIND 任务调度

调度针对多任务系统而言，是指根据一定的约束规则，将 CPU 分配给符合条件的任务使用，这些约束规则就是所谓的一个调度算法。多任务系统使用多种算法来给处于就绪态（ready）的任务分配 CPU。Wind 内核默认采用基于优先级的抢占式调度（Priority-base preemptive scheduling）算法，同时，还可以选用轮转（round-robin）调度算法。控制任务调度的函数调用见表 4.2。

表 4.2 控制任务调度的函数调用

调 用	描 述
kernelTimeSlice()	控制轮转调度
taskPrioritySet()	改变任务的优先级
TaskLock()	禁止任务调度
taskUnlock()	允许任务调度



■ 基于优先级的抢占式任务调度

如果使用基于优先级的抢占式调度算法，系统中的每个任务都拥有一个优先级，任一时刻，内核将 CPU 分配给处于就绪态的优先级最高的任务运行。之所以说这种调度算法是抢占的，是因为如果系统内核一旦发现有一个优先级比当前正在运行的任务的优先级高的任务转变为就绪态，内核立即保存当前任务的上下文，当前任务状态变为阻塞，插入到相应队列，并且切换到这个高优先级任务的上下文执行。

在图 4.3 中，任务 Task1 被高优先级的任务 Task2 抢占，Task2 又被 Task3 抢占，当 Task3 运行结束，Task2 继续执行，当 Task2 运行结束，Task1 继续执行。

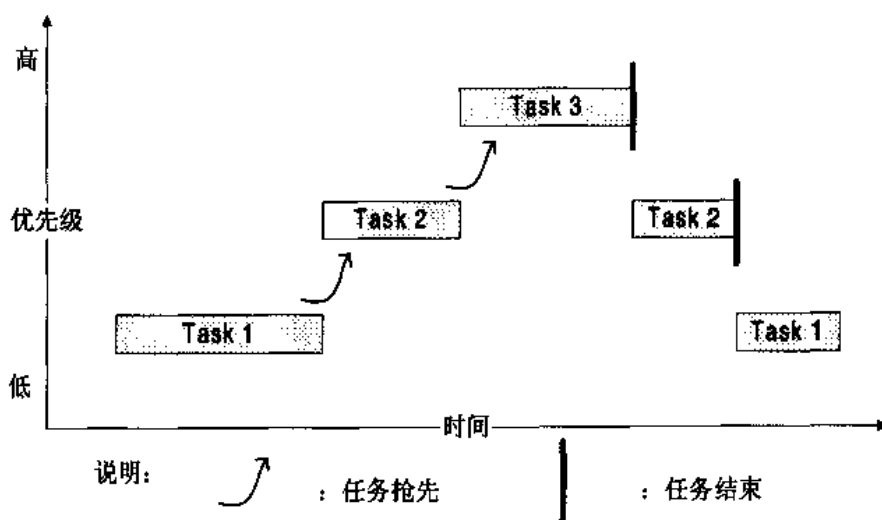


图 4.3 基于优先级的抢占式调度算法示意图

Wind 内核有 256 个优先级，编号 0~255，优先级 0 最高，255 最低。任务的优先级在创建时指定。然而 VxWorks 允许任务动态改变自己的优先级：当任务执行时，它可以调用 `taskPrioritySet()` 改变自己的优先级。这种允许动态改变任务优先级的能力允许应用遵循真实世界优先于（precedence）关系的变化。

■ 轮转调度

基于优先级的抢占式调度可以与轮转调度相结合。

轮转调度算法试图让优先级相同的、处于就绪态的任务公平地分享使用 CPU。如果不使用轮转调度，当系统中存在多个相同优先级的任务共享 CPU 时，第一个获得 CPU 的任务可以不被阻塞地独占 CPU，如果没有阻塞或其他情况出现，它从不会给其他相同优先级的



任务运行的机会。

轮转调度是使用时间片来实现这种相同优先级任务 CPU 公平分配的。一组任务中的每个任务执行一个预先确定的时间段，称为一个时间片；然后另一个任务执行相等的一个时间片，依次进行。这种分配是公正的，它保证一个优先级组中，在所有任务都得到一个时间片之前，不会有任务得到第二个时间片。这种调度算法特别适用于通用操作系统，如 Windows 操作系统。

在 VxWorks 系统中，可以调用函数 `kernelTimeSlice()` 来使用轮转调度，其参数是时间片的长度。时间片是每个任务在放弃 CPU 给另一个相同优先级任务之前，系统允许它运行的时间长度。

更准确地说，如果使用轮转调度算法，系统中的每个任务都有一个运行时间计数器。这个计数器随系统时钟增加而增加。当达到规定的时间片的值时，也就是说一个规定的时间片已经完成，这个计数器清零，调度器将这个任务放到相应任务优先级队列的尾部，将 CPU 交给队头的任务执行。新加入一个优先级组队列的任务将放到队列的尾部，计数器初始化为 0。

如果任务在它的时间片中被高优先级的任务抢占，调度器保存它的运行时间计数器，当它再一次符合执行条件的时候，调度器恢复运行时间计数器。

图 4.4 是采用轮转调度算法的一个例子。图中有三个相同优先级的任务：t1、t2、t3，t2 在执行时被高优先级的任务 t4 抢占，当 t4 完成后，t2 又恢复运行，时间片完成后然后是 t3。

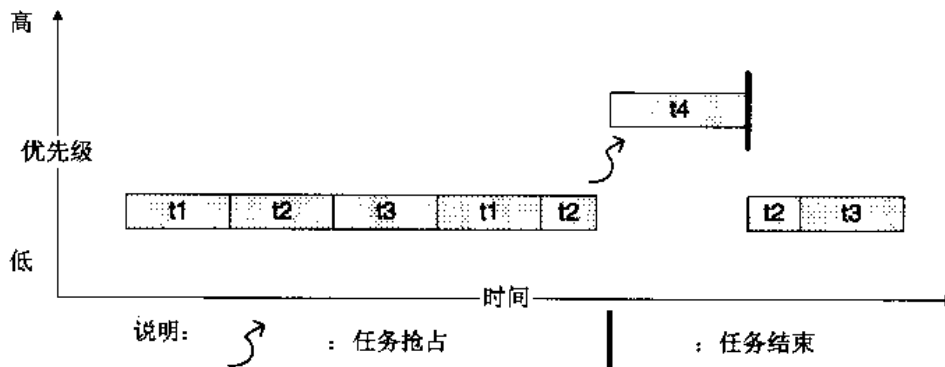


图 4.4 基于优先级的抢占式调度与轮转调度算法相结合调度示意图

■ 抢占上锁

实际应用中，并不是每次抢占都是合理的，非期待的抢占可能会导致系统出现意想不到的情况。因此，操作系统应该提供避免抢占的机制。



Wind 的调度器提供了允许或禁止调度的功能调用：

```
taskLock()
taskUnlock()
```

当一个任务调用 `taskLock()`，将禁止调度器抢占调度，那么该任务执行时，将不会发生基于优先级的抢占式调度。

然而如果这个任务在执行中被阻塞或挂起，调度器将选择具备资格的最高优先级任务运行。当这个禁止抢占的任务解除阻塞，再一次开始运行时，抢占再一次被禁止。调用 `taskUnlock()` 将恢复抢占调度。

注意，抢占上锁只能防止任务上下文交换，但不能禁止中断。

抢占上锁可用来实现互斥。需要指出的是，尽量使上锁的时间最小。

4.1.4 任务错误状态

按照惯例，C 库函数当产生错误时将一个全局整数 `errno` 设置某个合适的值，告诉系统发生了什么错误。这种惯例也是 ANSI C 标准的一部分。

■ `errno` 的分层定义

VxWorks 中的 `errno` 由两种不同的方式同时定义。在 ANSI C 中有一个潜在的命名为 `errno` 的全局变量，它可以在 Tornado 开发工具中显示；`errno` 同时作为一个宏也定义在 `errno.h` 中。对 VxWorks 来言，除一个函数外，其他所有的部分均可操作 `errno` 宏。`errno` 宏定义成对函数 `__errno()` 的一次调用：

```
#define errno    (__errno())
```

`__errno()` 函数返回全局变量 `errno` 的地址（正如你可能猜想的，这个函数不能调用使用自身的宏 `errno`）。这个实现带来一个有用的特征：由于 `__errno()` 是一个函数，用户可以在调试时在其中加入断点，用来监测一个特定的错误发生在哪儿。而且，C 程序可以按正常方式设置 `errno` 的值。

```
errno = someErrorNumber;
```

因此不要使用与 `errno` 名字相同的局部变量。

■ 任务各自的 `errno` 值

因为 VxWorks 将 `errno` 声明为一个全局变量，所以应用代码可以直接引用它。而且，对于 VxWorks 的多任务环境，由于每个任务都需要看到自己版本的 `errno`，`errno` 更是非常



有用。因而 VxWorks 把 `errno` 作为每个任务的上下文，在每次任务上下文交换时，`errno` 将被同时保存与恢复。

同样，中断服务程序也需要自己版本的 `errno`。这是由内核提供的作为中断一部分的中断处理程序进入和退出代码（见第六章）自动在中断堆栈中将 `errno` 保存和恢复来实现的。因此，不考虑 VxWorks 上下文，通过直接操纵全局变量 `errno`，可以访问出错代码。

■ 错误返回约定

几乎所有的 VxWorks 函数遵循一个约定：由函数实际返回的值来简单地指示函数操作成功与否。许多函数仅仅返回状态值 `OK` (`=0`) 或 `ERROR` (`= -1`)。有些函数正常时返回一个非负整数，例如 `open()` 返回一个文件描述符；出错时返回 `ERROR`，指明发生了一个错误。对于那些返回类型为指针的函数，通常用返回 `NULL` (`=0`) 来指明发生了错误。

在大多数情况下，函数在返回一个错误指示时，也将设置 `errno` 值来指明发生的是哪种特定错误。VxWorks 程序从不会清除全局变量 `errno`，因此它的值总是由最后发生的错误状态设置。如果一个 VxWorks 子程序在调用其他程序时得到一个错误，它通常返回自己的错误指示，而不去修改 `errno`。因此在底层程序设置的 `errno` 值作为一个错误类型的指示仍然有效。

例如，中断连结函数 `intConnect()`，它将一个硬件中断与某一用户程序相联系，调用 `malloc()` 来分配内存，在所分配的内存中建立中断处理程序，详细过程请参考第六章。如果系统由于没有足够内存，调用 `malloc()` 会失败，它设置 `errno` 来指明内存分配库 `memLib` 遇到这种没有足够内存（“insufficient memory”）用于分配的错误的。 `malloc()` 然后返回 `NULL` 来表示分配失败。 `intConnect()` 函数接受由 `malloc()` 返回的 `NULL`，返回它自己的错误指示 `ERROR`。然而，它没有改变 `errno`，`errno` 仍然是由 `malloc()` 设置的“insufficient memory”。

例如：

```
if ((pNew = malloc (CHUNK_SIZE)) == NULL)
return (ERROR);
```

Wind 推荐用户在自己的子程序中使用这种机制。同时设置检查 `errno` 也可作为调试的一种技术手段。如果 `errno` 的值在错误状态符号表（`error-status symbol table: statSymTb`）中对应一个字符串，调用函数 `printErrno()` 可以显示这一字符串。

■ 错误状态值分配

在 VxWorks 中，`errno` 值编码由四个字节组成。两个高字节表示产生错误的模块，指明产生错误的库（对应一个头文件）；低两个字节，表示错误号，指示在这个库中（头文件）特定的错误。所有 VxWorks 模块从 1~500 编码，`errno` 值为 0 的模块用来与源兼容。

头文件 `target/h/vwModNum.h` 定义了高 16 位对应的模块（头文件）。例如，`0xd0003`，



高 16 位是 0xd，对应的模块是 M_iosLib，对应的头文件应该是 target/h/iosLib.h。打开这个文件，就可找到低 16 位错误号 3 对应的意思：S_iosLib_INVALID_FILE_DESCRIPTOR。

另外，如果在配置操作系统时包含了 INCLUDE_STAT_SYM_TBL 选项，那么就可以在 shell 中用 printErrno() 打印错误信息。

应用程序可以使用所有其他的 errno 值，也就是说大于等于 128256 ($501 \ll 16$) 的所有正整数和所有负数。

4.1.5 任务异常处理

程序代码或数据的错误可能引起硬件异常状态，诸如：非法指令、总线或地址错、除数为 0 等等。VxWorks 异常处理包处理这些异常。默认的异常处理挂起 (suspends) 引起异常的任务，保存这个任务在异常点的状态，并将关于这个异常的描述送到 Tornado 开发工具中。我们可以根据这些信息分析产生异常的原因。同时内核和其他的任务不会被中断，系统继续执行。

VxWorks 同时允许任务使用信号功能激活自己的异常处理程序。如果一个任务已经提供了一个异常的信号处理程序，系统提供的上述的默认的异常处理程序不再执行。像发送硬件异常信号那样，信号也可用来发送软件异常信号。

4.1.6 共享代码和重入

VxWorks 提倡单个子程序的拷贝或子程序库被多个不同的任务调用。例如，许多任务可能要调用 printf()，但是系统中仅有一份拷贝。一个被多个任务调用的单个拷贝称为共享代码。VxWorks 动态链接功能很容易实现代码共享。代码共享也使得系统更为有效更容易维护，如图 4.5。

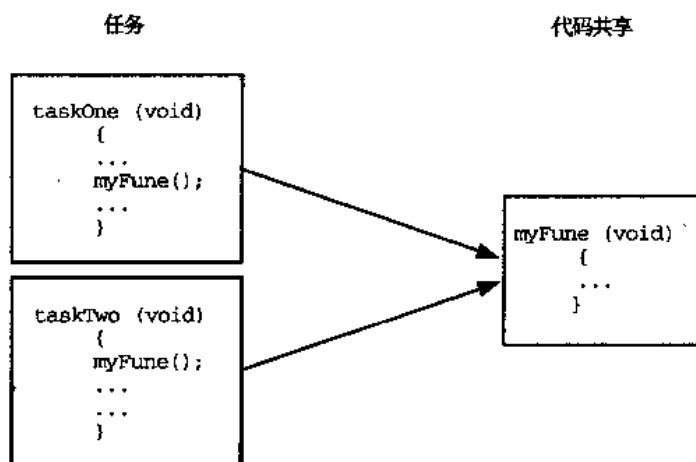


图 4.5 共享代码



共享代码必须是可重入的。一个子程序是可重入的，如果该程序的单个拷贝可以被多个任务同时调用而不会发生冲突。

这种冲突典型发生在在一个子程序修改了全局的或静态的变量，由于系统中仅有一份拷贝，该子程序指向这些变量的地址可能与其他任务的上下文重叠，也就是说其他的任务受到了影响，这种影响可能是用户所不期望的。

VxWorks 的很多子程序是可重入的。然而所有以 name_r()命名的子程序被认为是不可重入的。例如，由于 ldiv()有一个对应的程序 ldiv_r()，我们可以断定 ldiv()是不可重入的。

VxWorks 的 I/O 和驱动程序是可重入的，但是也要求应用小心设计。对于缓冲 (buffered) I/O，VxWorks 推荐使用文件指针。

由于 VxWorks 的文件描述符表是全局的，在驱动级，可能有来自不同的任务用流加载缓冲，这可能是你所希望的，也可能是你不希望的，这依赖于应用的本质。例如，由于包的头部指示每个包的目的地，因而包驱动程序能够混合来自不同任务的流。

大部分 VxWorks 程序使用下面的重入机制：

- 动态堆栈变量。
- 由信号量保护的全局或静态变量。
- 任务变量。

Wind 推荐使用这些技术来写可重入的代码。

■ 动态堆栈变量

在 VxWorks 应用中，许多子程序可能仅仅是纯代码，除了自己的动态堆栈变量外没有自己的数据。这些程序除了调用者以参数形式提供的数据之外，他们不需要其他数据就可工作。连结链库 (lstLib)，就是一个很好的例子。它的程序操作由调用者调用时提供 lists 和 nodes。这种子程序本质上是可重入的，多个任务可以同时调用它，相互之间不会干扰，这是因为由于每个任务只是在自己的堆栈内进行操作，如图 4.6 所示。

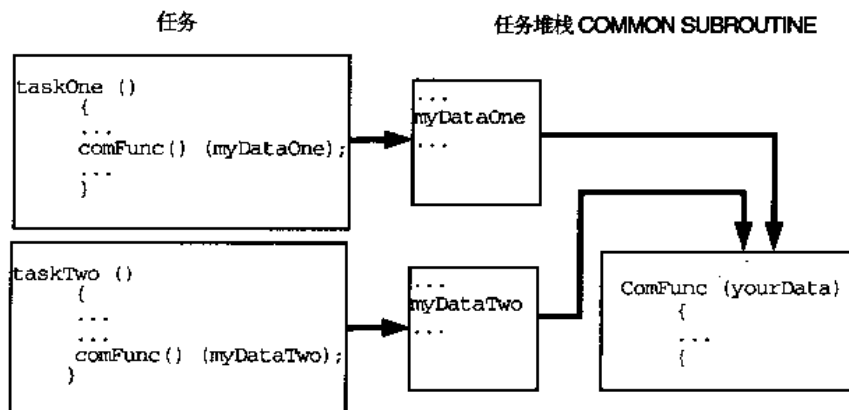


图 4.6 堆栈变量和共享代码



■ 受保护的全局或静态变量

VxWorks 的一些库封装对公共数据的访问。一个例子是内存分配库 `memLib`，它管理被多个任务使用的内存池。它声明并使用自己的静态数据变量跟踪内存池的分配。

由于这类库本质上不是可重入的，应用时需要小心对待。当多个任务同时调用这些库中的函数访问公共变量时可能会产生冲突。这类库必须借助于互斥机制，禁止任务同时访问临界区代码来实现可重入。通常使用的互斥机制是由 `semLib` 提供的信号量。

■ 任务变量

一些程序可能会被多个任务同时调用，这些任务可能要求全局变量和静态变量有不同的值。例如几个任务可能用相同的全局变量访问一块私有内存缓冲。为解决这种情况，VxWorks 提供所谓任务变量的机制 (task variables)，这种机制允许在任务上下文中增加 4 字节的变量，因此每次上下文交换时，改变量的值被保存。典型地，几个任务声明相同的变量 (4 个字节) 作为一个任务变量。每个任务可以把这个变量的内存地址作为自己的私有变量，如图 4.7。

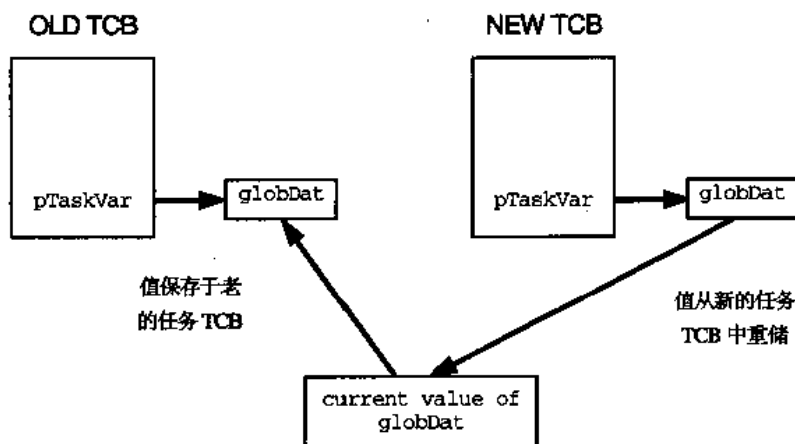


图 4.7 任务变量和上下文交换

这种机制由 `taskVarLib` 库中的函数 `taskVarAdd()`、`taskVarDelete()`、`taskVarSet()` 和 `taskVarGet()` 提供。

使用这种机制要非常节约。由于改变量必须作为任务上下文的一部分保存与恢复，每增加一个任务变量将会使该任务的上下文交换时间增加几微秒。如果应用需要的任务变量比较多，可以考虑使用下述技巧：搜集一个模块所有的任务变量，把它们存放在一个动态分配的数据结构中，使得所有对该数据结构的访问均可通过一个指针直接获得，这个指针可以作为使用这个模块的任务变量。



使用相同主程序的多个任务

VxWorks 可能使用相同的主程序作为不同的几个任务发起。每个发起的任务将使用自己的堆栈和上下文，分别传给该程序不同的参数。在这种情形下，就需要使用前面讲到的重入规则。

这种用法对于同一个程序需要以不同的参数并发执行是非常有用的。例如，一个监控指定类型装备的程序可能要发起多次，来监控装备中不同的部分。主程序的参数指定该任务要监控的部分，如图 4.8 所示。机械臂的多个连接可以使用相同的代码，任务调用 `joint()` 来操控连接。关节号 (`jointNum`) 用于指定机械臂要操纵的连接。

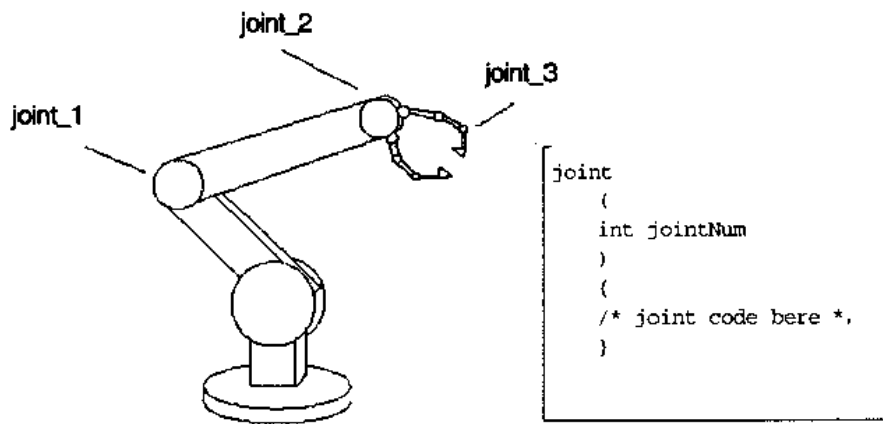


图 4.8 多个任务使用相同的代码

4.1.7 VxWorks 系统任务

VxWorks 包括以下系统任务：

➤ 根任务：tUsrRoot

根任务，tUsrRoot，是内核执行的第一个任务。其入口点是

```
installDir/target/config/all/usrConfig.c
```

文件中的 `usrRoot()` 初始化 VxWorks 系统的主要功能，发起诸如日志任务、异常处理任务、网络任务和 `tRlogind` 后台任务。正常情况下在所有初始化完成之后，根任务终止并被删除。用户可以向根任务自由的添加任何必需的初始化代码。



➤ **日志任务: tLogTask**

日志任务, tLogTask, 是 VxWorks 用来记录系统信息的任务。

➤ **异常处理任务: tExcTask**

异常处理任务 tExcTask, 提供 VxWorks 异常处理包, 完成在中断级不能执行的功能。必须具备系统最高的优先级。不需要挂起、删除、改变其任务的优先级。

➤ **网络任务: tNetTask**

tNetTask 后台处理 VxWorks 网络需要的任务级功能处理。

➤ **目标代理任务: tWdbTask**

目标代理任务, tWdbTask, 当目标代理设置为运行在任务模式时, 创建此任务。它处理来自 Tornado 目标服务的请求。

➤ **可选组件的任务**

如果定义了相关的配置常量, 将创建下面的 VxWorks 系统任务。

➤ **tShell**

如果在 VxWorks 配置中包含了 target shell, target shell 将作为任务发起。任何从 target shell 中调用而非发起的程序或任务, 均运行于 tShell 的上下文中。

➤ **tRlogind**

如果在 VxWorks 配置中包含了 target shell 和 rlogin 工具, 这个后台程序允许远程用户登录到 VxWorks。它接收来自于其他 VxWorks 或主机系统的远程登录请求, 发起 tRloginTask 和 tRlogoutTask 两个任务。这些任务随远程用户的退出 (logged on) 而退出。在远程会话中, shell 和其他任务的输入/输出重定向到远程用户。使用 VxWorks 为终端驱动程序 ptyDrv, 一个类 tty 的接口提供给远程用户。

➤ **tTelnetd**

如果在 VxWorks 配置中包含了 target shell 和 telnet 工具, 这个后台任务许远程用户使用 telnet 登录到 VxWorks。它接收来自于其他 VxWorks 或主机系统的远程登录请求, 发起输入任务 tTelnetInTask 和输出任务 tTelnetOutTask。这些任务随远程用户的退出 (logged on) 而退出。在远程会话中 shell 和其他任务的输入/输出重定向到远程用户。使用 VxWorks 为终端驱动程序 ptyDrv, 一个类 tty 的接口提供给远程用户。

➤ **tPortmapd**

如果在 VxWorks 配置中包含了 RPC 功能, 作为一个 RPC 服务器, 这个后台任务完成运行在同一台机器上的 RPC 服务中心注册功能。RPC 客户请求 tPortmapd 后台程序找出如



何联系不同的服务器。

4.2 VxWorks 任务编程接口

4.2.1 任务控制函数

VxWorks 提供了丰富的任务控制功能，这些函数调用包含在 taskLib 库中。这些功能包括：任务的创建、控制和获取任务信息。我们可以在 Tornado 集成开发环境提供的 Shell 工具中交互使用这些调用。

■ 任务的创建和激活

表 4.3 列出了用于任务创建的函数调用。

表 4.3 用于任务创建函数调用

函数调用	描述
taskSpawn()	创建并激活一个新任务
taskInit()	初始化一个新任务
taskActivate()	激活一个已初始化的任务

taskSpawn()的函数原型是：

```
int taskSpawn
(
    char    *name,          /* 新任务的任务名(stored at pStackBase) */
    int     priority,      /* 新任务的优先级 */
    int     options,       /* 任务选项字 */
    int     stackSize,     /* 堆栈大小 */
    FUNCPTR entryPt,      /* 新任务的入口函数 */
    int     arg1,          /* 以下是传给入口函数的 10 参数 */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
```



```

int    arg8,
int    arg9,
int    arg10
)

```

`taskSpawn()`调用共有 15 个参数，分别是新任务的名字、优先级、任务选项字、堆栈大小、任务入口函数以及作为启动参数传给入口程序的 10 个参数。该调用返回任务 ID 号。

调用示例：

```
Id= taskSpawn(name,priority,options,stacksize,enterFunc,arg1,...,arg10);
```

`TaskSpawn()`将创建新任务的上下文，包括分配堆栈和建立的含有特定参数的入口程序（一个普通的子程序）调用的任务环境。新的任务将在指定的函数入口处执行。

任务名要求便于记忆。在 Tornado Shell 中使用命令 `i`，显示的任务信息中，包含每个任务的名字。名字可以是任意长度、任意内容。

发起任务分配的唯一资源是定位于系统内存中的一个特定大小的堆栈。堆栈的大小应该是一个偶数。这块内存中包含了任务控制块（TCB）和任务名。剩余的内存是任务堆栈，每个字节用 `0xEE` 填充，这种初始化填充主要用于任务堆栈检查函数 `checkStack()`。

入口地址 `entryPt` 是任务入口函数的地址。一旦 C 环境建立，系统将调用这个程序，该程序参数由给定的十个参数获得，这十个参数（并且是仅能有 10 个）必须要传给发起函数。由于受到参数数目的限制，对于需要较多参数的程序，可以考虑将参数收集在一个结构中，把结构地址作为参数传给程序。

返回值：如果任务发起成功，返回任务 ID 号；如果没有充足内存或者任务创建失败，返回 `ERROR`。发起函数返回的 `ERRNOS` 可能的值有：

```

S_intLib_NOT_ISR_CALLABLE : 程序不能从一个中断服务程序中调用；
S_objLib_OBJ_ID_ERROR     : 不正确的任务 ID；
S_smObjLib_NOT_INITIALIZED : 在指定的分区中，没有足够的内存用于发起任务；
S_memLib_NOT_ENOUGH_MEMORY : 没有足够的内存用于发起任务；
S_memLib_BLOCK_ERROR      : 不能够对内存分区互斥访问。

```

`TaskSpawn()`包含了定位分配、初始化和激活等一些低级操作。初始化和激活功能由函数 `taskInit()`和 `taskActivate()`完成。

`taskInit()`的函数原型是：

```

STATUS taskInit
(
WIND_TCB *pTcb,          /* 新任务的 TCB 地址          */
char *name,             /* 新任务的名字              */
int priority,           /* 新任务的优先级            */
)

```



```
int      options,          /* 任务选项字          */
char     *pStackBase,     /* 新任务的堆栈基址    */
int      stackSize,       /* 任务需要的堆栈大小  */
FUNCPTR  entryPt,        /* 新任务的入口函数    */
int      arg1,            /* 以下是传给入口函数的 10 个参数 */
int      arg2,
int      arg3,
int      arg4,
int      arg5,
int      arg6,
int      arg7,
int      arg8,
int      arg9,
int      arg10
)
```

该函数初始化指定的内存区域作为任务的堆栈和控制块，而不是像 `taskSpawn()` 那样由系统自动分配。这个函数允许初始化一个静态的 `WIND_TCB` 变量，也允许应用为方便调试，把任务堆栈分配在指定的位置。

另外一个与 `taskSpawn()` 不同之处是，`taskInit()` 不一定要指定任务名，而允许发起一个未命名的任务，由系统自动分配任务名。

其余的参数与 `taskSpawn()` 相同。

返回值：成功时返回 `OK`；任务不能初始化则返回 `ERROR`。

可能返回的 `ERRNOS` 值有以下几个：

```
S_intLib_NOT_ISR_CALLABLE    : 程序不能从一个中断服务程序中调用；
S_objLib_OBJ_ID_ERROR        : 不正确的任务 ID。
```

`taskActivate()` 的函数原型是：

```
STATUS taskActivate
(
    int tid /* 要激活的任务 ID 号 */
)
```

函数 `taskActivate()` 激活由 `taskInit()` 创建的任务，其参数是 `taskInit()` 第一个参数任务控制块的地址 (`WIND_TCB`)，这可以通过强制类型转换得到：

```
tid = (int) pTcb;
```

仅仅当应用对分配定位和激活需要更多的控制时，才使用这两个函数。在大多数情况下，推荐直接使用 `taskSpawn()`。



■ 任务名和 ID 号

任务创建的时候，一般要指定一个名字，也就是任务名。VxWorks 将返回一个四字节的指向任务数据结构的任务 ID 号。大多数 VxWorks 程序使用任务的 ID 号定位某个任务。VxWorks 约定 ID 号 0 表示调用任务（指调用该函数的任务）。

VxWorks 保证新的任务名不会与现存的任何任务名冲突。而且，Tornado 开发工具的设计优点保证了任务名不会与全局可视变量和程序名冲突。为了避免冲突，VxWorks 使用约定的命名规则：

所有从目标机启动的任务以字母 t 开头命名，从主机启动的任务以字母 u 开头命名。

如果不想命名一个任务，VxWorks 自动为新任务分配一个独一无二的名字 tN，N 是一个随未命名任务递增的一个十进制整数。

表 4.4 列出了用于管理任务名称和 ID 号的函数。

表 4.4 用于管理任务名称和 ID 号的函数

函数调用	描 述
taskName()	由任务号得到任务名
taskNameToId()	由任务名得到任务 ID 号
taskIdSelf()	得到调用任务的 ID 号
TaskIdVerify()	证实一个特定任务的存在

■ 任务选项

taskSpawn 的第二个参数是选项参数，可以是表 4.5 中的几个选项之一或其组合。如果任务执行任何浮点操作，必须要求使用 VX_FP_TASK 选项。

任务发起后，调用表 4.6 中的函数可以检查或改变任务选项，当前仅有 VX_UNBREAKABLE 选项可以改变。

表 4.5 任务选项

名 称	值 (十进制)	描 述
VX_FP_TASK	0x8	运行时使用浮点运算协处理器
VX_NO_STACK_FILL	0x100	不要用 0xee 填充堆栈
VX_PRIVATE_ENV	0x80	在私有环境中执行
VX_UNBREAKABLE	0x2	禁止断点



表 4.6 取得或设置任务选项的函数

函数调用	描述
TaskOptionsGet()	取任务选项
taskOptionsSet()	设置任务选项

■ 任务信息

表 4.7 列出了获取任务信息的函数调用。由于任务的状态是动态的，除非任务正在挂起，否则获取的信息可能不是任务当前的状态。

表 4.7 获取任务信息的函数调用

函数调用	描述
taskIdListGet()	获取所有活动任务的 ID 号
taskInfoGet()	得到指定任务的信息
taskPriorityGet()	查看指定任务的优先级
taskRegsGet()	查看任务的寄存器
taskRegsSet()	设置任务的寄存器
taskIsSuspended()	查看指定任务是否处于 <code>suspended</code> 状态
taskIsReady()	查看指定任务是否就绪
taskTcb()	得到任务控制块指针

■ 任务删除和删除安全

任务可以动态地从系统中删除。VxWorks 提供了用于删除任务和保护任务避免被删除的函数调用，如表 4.8。

注意：确保不要在一个不合适的时刻删除任务，在任务被删除之前，该任务必须释放它所持有的所有资源。

表 4.8 用于删除任务和保护任务避免被删除的函数调用

调用	描述
exit()	终止调用任务的执行，释放所占用的内存（任务堆栈和任务控制块）
taskDelete()	终止一个指定的任务，释放所占用的内存（任务堆栈和任务控制块）



续表

调 用	描 述
taskSafe()	保护调用任务不被删除
taskUnsafe()	解除任务删除保护

如果在任务创建时，入口程序指定返回，任务将隐含调用 `exit()`。另外，任务可以在任何时候调用 `exit()` 杀死自身。一个任务可以调用 `taskDelete()` 删除其他任务。

当任务被删除时，删除不会通知给任何任务。VxWorks 允许调用 `taskSafe()` 和 `taskUnsafe()` 用来防止非期望的任务删除操作。`taskSafe()` 保护任务不会被其他任务删除。任务访问临界区时就需要这种保护。

注意，任务终止时，其执行时任务分配的内存不会被释放，如调用 `malloc()` 分配的内存，必须由任务自身编程释放。

例如，为了对某个数据结构互斥访问，任务可能取得一个信号量。当在临界区执行时，任务可能被其他任务删除，由于该任务不能完成临界区访问，该数据结构可能处于一种被破坏或者说不一致的状态。进一步说，由于该任务没有释放信号量，对其他任务而言，该临界区资源将不可用，本质上已经冻结。

为避免出现上述情形，应用可以调用 `taskSafe()` 保护取得这个信号量的任务。任何试图要删除被 `taskSafe()` 保护的的任务的任务将被阻塞。当完成临界区操作，这个受保护的的任务可以调用 `taskUnsafe()` 来解除保护，以允许其他任务删除自身，这时因调用 `taskDelete()` 而阻塞的任务将解除阻塞。

为了支持嵌套的删除保护，系统中维护一个计数器，它跟踪调用 `taskSafe()` 和 `taskUnsafe()` 的次数。仅当计数器为 0 时允许删除操作。保护操作仅对当前调用任务有效，一个任务不能对其他任务保护或解除保护。

下面的代码段说明如何使用 `taskSafe()` 和 `taskUnsafe()` 来保护一段临界取代码。

```
taskSafe ();
semTake (semId, WAIT_FOREVER); /* Block until semaphore available */
.
. 临界区操作代码
.
semGive (semId); /* Release semaphore */
taskUnsafe ();
```

删除安全性经常与互斥相联系。通常和一种特定的信号量“互斥信号量”一起使用。



■ 任务运行控制

表 4.9 中的函数用于控制任务的执行。

表 4.9 用于控制任务执行函数

函 数	描 述
taskSuspend()	挂起任务
taskResume()	恢复任务
taskRestart()	重新启动任务
taskDelay()	任务延时, 单位为 tick
nanosleep()	任务延时, 单位为纳秒

VxWorks 调试功能要求提供挂起和恢复任务的函数, 用于冻结任务的执行状态。由于某种原因正在执行的任务可能要求重新启动。重启函数 `taskRestart()` 将使用原有的参数重新创建一个任务, 当一个任务放弃请求时, Tornado shell 也使用这种机制来启动自身。

延时操作提供一个简单的任务睡眠机制。任务延时也常用于采用轮询机制的应用中。例如, 以下调用将任务延时空半秒。

```
taskDelay (sysClkRateGet ( ) / 2);
```

函数 `sysClkRateGet()` 返回系统时钟速率, 单位是 tick 数/每秒。

与 `taskDelay()` 函数相对应的 POSIX 函数是 `nanosleep()`, 后者可以直接规定延时的时间单位。两者仅仅延时单位不同, 其效果相同。`taskDelay()` 另一方面的效果是移动调用任务到相同优先级就绪队列的尾部。特别地, 我们可以通过调用 `taskDelay(0)`, 将 CPU 交给系统中其他相同优先级的任务。

```
taskDelay (NO_WAIT); /* allow other tasks of same priority to run */
```

延时为 0 的调用只能使用 `taskDelay()`, `nanosleep()` 认为这种调用是一个错误。

4.2.2 任务扩展函数

有时应用需要在任务创建、删除或上下文切换时增加相应的处理, 又不需要涉及修改内核。wind 提供了任务创建、交换、删除钩子函数, 可以满足这种需要。这些函数允许当任务创建、上下文交换发生或任务被删除时调用附加的函数。任务控制块 (TCB) 有一些空间, 应用可用来扩充任务上下文, 这些钩子函数如表 4.10。更为详细的信息参见 `taskHookLib`。



表 4.10 任务创建、交换、删除钩子函数

调 用	描 述
taskCreateHookAdd()	增加一个在每个任务创建时调用的函数
taskCreateHookDelete()	删除一个先前增加的任务创建时调用的函数
taskSwitchHookAdd()	增加一个在每次任务交换时调用的函数
taskSwitchHookDelete()	删除先前增加的在每次任务交换时调用的函数
taskDeleteHookAdd()	增加一个在每个任务删除时调用的函数
taskDeleteHookDelete()	删除先前增加的在每个任务删除时调用的函数

用户安装的交换钩子函数将在内核上下文中调用执行。因此，交换钩子函数不能调用所有的 VxWorks 系统调用。表 4.11 总结了交换钩子函数可以调用的函数。通常不涉及内核的函数均可被调用。

表 4.11 交换钩子函数可以调用的函数

库	函数调用
Blib	所有函数
FppArchLib	FppSave(), fppRestore()
IntLib	intContext(), intCount(), intVecSet(), intVecGet(), intLock(), intUnlock()
LstLib	除 lstFree()之外的所有程序
MathALib	如果使用 fppSave()/fppRestore(), 所有程序均可调用
RngLib	除 rngCreate() 和 roundlet()之外的所有程序
TaskLib	taskIdVerify(), taskIdDefault(), taskIsReady(), taskIsSuspended(), taskTcb()
VxLib	vxTas()

4.3 POSIX 调度接口

schedPxLib 库提供了 POSIX 1003.1b 调度函数，如表 4.12 所示。这些程序允许用户使用一个可移植的接口来得到或设置任务的优先级、得到任务的最大或最小优先级。如果使用了轮转调度算法，还可得到时间片的长度。



表 4.12 POSIX 调度接口

调 用	描 述
<code>sched_setparam()</code>	设置任务的优先级
<code>sched_getparam()</code>	获得指定任务的调度参数
<code>sched_setscheduler()</code>	设置任务的调度策略和参数
<code>sched_yield()</code>	放弃 CPU
<code>sched_getscheduler()</code>	获得当前调度策略
<code>sched_get_priority_max()</code>	获得最大优先级
<code>sched_get_priority_min()</code>	获得最小优先级
<code>sched_rr_get_interval()</code>	如果是轮转调度, 获得时间片大小

为了更好地理解如何使用这些函数, 我们先看看 POSIX 和 Wind 调度方法的有何不同。

4.3.1 POSIX 和 Wind 调度方法的差异

POSIX 和 Wind 调度方法的不同体现在以下几方面:

POSIX 调度基于进程, 而 Wind 调度则基于任务。任务和进程不同在于以下几方面。最明显的是, 任务可以直接内存访问, 而进程则不能。进程仅仅继承了父进程的特定特征, 而任务则操作在与父任务完全一样的环境中。

任务和进程的相似之处在于都可以被单独调度。

VxWorks 文档使用基于优先级的抢占式调度这个词, 而 POSIX 标准则使用 FIFO。这纯粹是由命名术语引起的差异。而这都是基于相同的优先级策略。

POSIX 调度算法应用在进程到进程基础之上的, 而 Wind 调度算法则应用于整个系统 (system-wide) 基础之上, 所有任务既可以使用轮转调度方案, 又可以使用基于优先级的抢占式调度方案。

POSIX 的优先级编号方案与 Wind 的方案相反。POSIX 中, 优先数越高, 优先级越高; Wind 方案中, 优先数越低, 优先级越高, 0 是最高的优先级。因此, 使用 POSIX 调度库 (schedPxLib) 的优先数将与使用 VxWorks 其他所有的成分不匹配。用户通过将默认的全局变量 `posixPriorityNumbering` 的设置改为 `FALSE`, 可以解决这种冲突。使用这种设置, `schedPxLib` 库将使用 Wind 编号方案 (小的编号对应高的优先级), 这将与 VxWorks 其他所有的成分相一致。

使用 POSIX 调度程序, 需要在配置 VxWorks 时, 包含 `INCLUDE_POSIX_SCHED` 宏定义, 系统将自动包含 POSIX 调度程序。



4.3.2 获得和设置 POSIX 任务优先级

获得和设置 POSIX 任务优先级由函数 `sched_setparam()` 和 `sched_getparam()` 分别完成。

函数原型分别为：

```
int sched_setparam
(
    pid_t          tid,          /* 任务 ID          */
    const struct sched_param * param /* 调度参数        */
);

int sched_getparam
(
    pid_t          tid,          /* 任务 ID          */
    struct sched_param * param /* 存储优先级的调度参数 */
);
```

两个函数均以任务和一个 `sched_param` 数据结构（在 `installDir/target/h/sched.h` 中定义）作为参数。

```
struct sched_param /* Scheduling parameter structure */
{
    int sched_priority; /* 调度优先级      */
};
```

以 0 作为任务 ID 号可以用来获得和设置调用任务的优先级。当调用 `sched_setparam()` 时, `sched_param` 数据结构中的成员 `sched_priority` 指定新的优先级。调用 `sched_getparam()` 将指定任务的当前优先级填到 `sched_priority` 中。

例 4.1 获得和设置 POSIX 任务优先级

```

/*****
 * 该程序将调用任务的优先级设置为 150，然后证实它的优先级。
 * 在 shell 下以任务的方式运行它：
 * -> sp priorityTest
 */

/* includes */
#include "vxWorks.h"
#include "sched.h"
```



```
/* defines */
#define PX_NEW_PRIORITY 150

STATUS priorityTest (void)
{
    struct sched_param myParam;

    /* 初始化参数 */
    myParam.sched_priority = PX_NEW_PRIORITY;
    if (sched_setparam (0, &myParam) == ERROR)
    {
        printf ("error setting priority\n");
        return (ERROR);
    }

    /* 获得任务的优先级，证实是否与设置的优先级值一致 */
    if (sched_getparam (0, &myParam) == ERROR)
    {
        printf ("error getting priority\n");
        return (ERROR);
    }
    if (myParam.sched_priority != PX_NEW_PRIORITY)
    {
        printf ("error - priorities do not match\n");
        return (ERROR);
    }
    else
        printf ("task priority = %d\n", myParam.sched_priority);
    return (OK);
}

/*****/
```

函数 `sched_setscheduler()` 用来设置一个 POSIX 进程的调度策略和优先级。在 VxWorks 内核中，`sched_setscheduler()` 仅仅控制任务的优先级，因为内核不允许任务有不同于其他任务的调度策略。如果指定的策略与当前整个系统的策略一致，那么 `sched_setscheduler()` 的设置等同于 `sched_setparam()`。如果不一致，`sched_setscheduler()` 将返回一个错误。

仅有的允许改变调度策略的方法是改变整个系统的调度策略。POSIX 没有提供这种函数调用。为改变整个系统的调度方法，可以使用 Wind 提供的函数调用 `kernelTimeSlice()`。



4.3.3 获得和显示当前的调度策略

POSIX 函数 `sched_getscheduler()` 返回当前的调度策略。

其函数原型为：

```
int sched_getscheduler
(
    pid_t tid /* task ID */
)
```

VxWorks 有两种有效的调度策略：基于优先级的抢占式调度（对应与 POSIX 的 `SCHED_FIFO`）和相同优先级的轮转调度（`SCHED_RR`）。

例 4.2 获得 POSIX 调度策略

```

/*****
* 该例子程序获得并显示当前的调度策略。
*/

/* includes */
#include "vxWorks.h"
#include "sched.h"

STATUS schedulerTest (void)
{
    int policy;
    if ((policy = sched_getscheduler (0)) == ERROR)
    {
        printf ("getting scheduler failed\n");
        return (ERROR);
    }
    /* 返回值应该是 SCHED_FIFO 或 SCHED_RR */
    if (policy == SCHED_FIFO)
        printf ("current scheduling policy is FIFO\n");
    else
        printf ("current scheduling policy is round robin\n");
    return (OK);
}
/*****/

```



4.3.4 获得调度参数: 优先级限制和时间片

函数 `sched_get_priority_max()` 和 `sched_get_priority_min()` 各自返回允许的最大和最小 POSIX 优先级值。如果采用轮转调度, 函数 `sched_rr_get_interval()` 返回当前时间片的长度, 其参数是一个 `timespec` 数据结构 (定义在 `time.h`) 的指针, 该函数将每个时间片的秒数和纳秒数填到这个数据结构合适的元素中。

例 4.3 返回当前 POSIX Round-Robin 调度时间片的长度

```
/* *****  
 * 该例子程序设置使用轮转调度, 然后获得并显示时间片长度  
 */  
  
/* includes */  
#include "vxWorks.h"  
#include "sched.h"  
  
STATUS rrgetintervalTest (void)  
{  
    struct timespec slice;  
  
    /* 设置使用轮转调度 */  
    kernelTimeSlice (30);  
    if (sched_rr_get_interval (0, &slice) == ERROR)  
    {  
        printf ("get-interval test failed\n");  
        return (ERROR);  
    }  
    printf ("time slice is %l seconds and %l nanoseconds\n",  
           slice.tv_sec, slice.tv_nsec);  
    return (OK);  
}  
/* *****/
```



任务间通信机制是多任务间相互同步和通信以协调各自活动的主要手段。VxWorks 提供的任务间通信手段按其速度由快到慢包括信号量、消息队列、管道到网络透明的套接字。

这一章介绍 VxWorks 提供任务间通信机制。

5.1 VxWorks 任务间通信机制

任务间通信功能用于协调多个独立任务间的活动。VxWorks 提供一套丰富的任务间通信机制，包括：

- ◆ 共享内存，数据的简单共享。
- ◆ 信号量，基本的互斥和同步。
- ◆ 消息队列和管道，同一 CPU 内多任务间消息传递。
- ◆ Sockets 和远程调用，任务间透明的网络通信。
- ◆ Signals，用于异常处理。

对于可选产品 VxMP，它提供运行在同一底板上多个 CPU 的任务间的通信机制。包括：信号量、共享消息队列、共享内存、共享名字数据库（shared name database）。

5.2 共享数据结构

任务间通信最直接最明显的方法是访问共享数据结构。由于所有 VxWorks 任务共存于单一的线性地址空间，在多个任务间共享数据结构是非常容易的。任一程序中定义的各种类型的全局变量，都可被所用任务直接访问，一个例子如图 5.1。

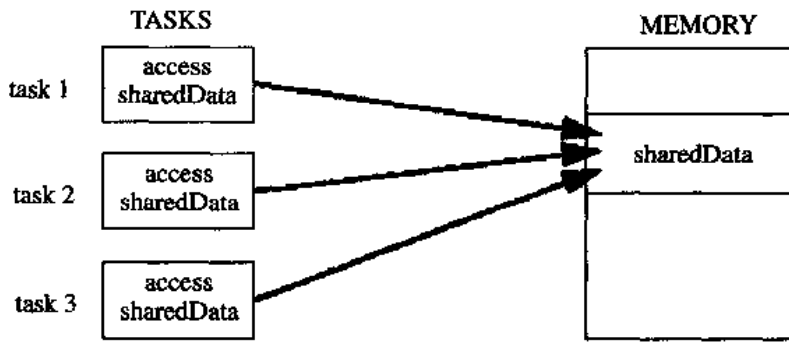


图 5.1 多个任务间共享数据结构

为方便编程，VxWorks 自身定义的几种数据类型：线性缓冲、环形缓冲、连接链等，这些类型可以被运行在不同上下文的代码引用。

5.2.1 连接链

连接链是一种双向连结的数据结构，定义在 `tornado\target\hlstLib.h` 中。VxWorks 提供的 `lstLib` 库用于设置和控制连接链。这些函数允许用户以多种方式操纵一个链，提供以下几种操作：

- ◆ 在链中任意位置插入或删除节点。
- ◆ 在链尾增加一个节点。
- ◆ 链接两个链。
- ◆ 从一个链中提取一个子链。
- ◆ 删除并返回链首节点。
- ◆ 计算链中节点数。

图 5.2 示意了一个连接链结构。

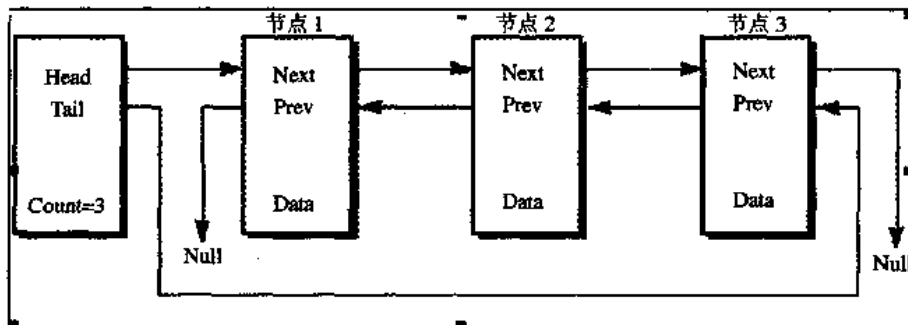


图 5.2 一个连接链



由图可以看出，每个节点包含了一个指向前一个节点的指针和一个指向后一个节点的指针，所以这个链是双向连结的。因而连结链可以在任意环节增长和收缩，所以连结链是一种功能非常强大的数据结构。

5.2.2 环形缓冲

环形缓冲是一种环形缓冲数据结构，定义在 `tornado\target\h\rngLib.h` 中。环形缓冲在用于任务和中断服务程序间传送字符时非常有用。环形缓冲大小固定，以先进先出方式工作，如图 5.3 所示。

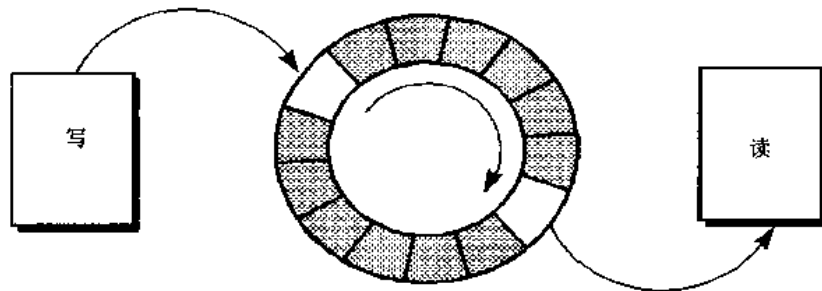


图 5.3 环形缓冲工作示意图

VxWorks `mgLib` 库提供环形缓冲管理函数，实现以下功能：

- 创建和删除环形缓冲。
- 从环形缓冲取得字符。
- 向环形缓冲添加字符。
- 以不同方式操纵环形缓冲。

环形缓冲使用时要考虑互斥问题。如果环形缓冲仅仅服务于一个读任务和一个写任务，那么不需要用信号量来控制对环形缓冲的访问：一个读任务仅仅影响读指针，一个写任务仅仅影响写指针。这两个任务不需要同步，这是因为环形缓冲允许批量地移入和移出。当读任务读取下一个节点时，该操作将使该节点变为空，使得节点对新数据可用。如果读任务不能跟上写任务的速度，缓冲区将会溢出，应用数据可能丢失。这种情形就需要较大的环形缓冲。

5.3 互 斥

当一个共享地址空间简单地用于交换数据时，为避免竞争，需要对该内存的访问上锁，



以保证访问互斥进行。实现资源互斥访问的方法很多，不同之处仅在于互斥的范围和程度。这些方法包括禁止中断、禁止抢占和使用信号量对资源上锁。

5.3.1 中断上锁和反应时间

互斥机制最强有力的方法是禁止中断。这种上锁保证了对 CPU 的独占访问：

```
funcA ()
{
    int lock = intLock();
    .
    .不能被中断的临界区
    .
    intUnlock (lock);
}
```

这种方法涉及到中断级互斥，也就是说，在互斥期间，即使外部事件产生，而引发的相应的中断，系统也不会切换到相应的中断服务程序 (ISR)。从而在上锁期间，它可能会造成系统对外部事件反应迟钝。这对于多于大多数实时系统而言，系统的实时性也就得不到保证，因而不适合作为一种通用的互斥方法。然而，当涉及到 ISR 需要互斥时，中断上锁又是必要的。但是在任何情形下，应该使中断上锁时间尽量短，这也是所有操作系统的基本要求。

5.3.2 抢占上锁和反应时间

另一种比中断上锁稍弱的互斥机制是禁止任务抢占。当不允许其他任务抢占当前任务的执行时，禁止抢占提供了一种较小限制性的互斥，在这种互斥存在的情形下，ISR 仍然能够执行。

```
funcA ()
{
    taskLock ();
    .
    .不能被中断的临界区
    .
    taskUnlock ();
}
```

不过，这方法仍可能造成系统的实时性得不到充分保证。这是因为在上锁的任务离开临



临界区解锁之前，处于就绪态的更高的优先级的任务仍不能够执行，尽管这个高优先级的任务没有涉及到临界区操作。当然，这种互斥使用非常简单。使用这种方法时，同样需要控制抢占禁止的时间尽可能地短。一种更好的机制是信号量。

5.4 信号量

VxWorks 信号量是提供任务间通信、同步和互斥的最优选择，它提供任务间的最快速通信。也是提供任务间同步和互斥的主要手段。

对于互斥，信号量可以上锁对共享资源的访问。并且比禁止中断或禁止抢占提供更精确的互斥粒度。

对于同步，信号量可以协调外部事件与任务的执行。

VxWorks 由三种类型的信号量，用于解决不同的问题：

- 二进制：最快的最常用的信号量，可用于同步或互斥。
- 互斥：为解决具有内在的互斥问题、优先级继承、删除安全和递归等情况而最优化的特殊的二进制信号量。
- 计数器：类似于二进制信号量，但是随信号量释放的次数改变而改变。适合于一个资源的多个实例需要保护的情形。

VxWorks 不仅提供主要为 VxWorks 设计的 Wind 信号量，同时为了使应用程序具有可移植性，也提供 POSIX 信号量。一个可候选的信号量库提供 POSIX 兼容的信号量接口。

上述信号量主要为单 CPU 系统设计，另一个可选产品 VxMP 提供用于多个处理器间的信号量。

5.4.1 信号量控制

为了简化程序设计，Wind 信号量提供一套单一的接口用于控制信号量，而不是为每种信号量定义一套完整的信号量控制函数。信号量的类型仅仅由创建函数确定，其他接口根据需要处理的信号量的类型自动完成相应的操作。表 5.1 列出了信号量控制函数。

`semBCreate()`、`semMCreate()`和 `semCCreate()`返回信号量 ID 号，为随后的信号量控制函数提供句柄。

信号量在创建的时候，队列类型就已确定。等待一个信号量的任务可以按优先级 (`SEM_Q_PRIORITY`) 或按先进先出 (`SEM_Q_FIFO`) 的顺序排队。



警告

`semDelete()` 调用将终止信号量，并且释放分配的与之相联系的内存。当删除信号量时必须要小心，特别是那些用于互斥的信号量。避免删除那些其他任务正在请求的信号量。不要删除一个信号量，除非是首先成功地取得了该信号量的任务才能删除它。

表 5.1 信号量控制函数

调 用	描 述
<code>semBCreate()</code>	分配并初始化一个二进制信号量
<code>semMCreate()</code>	分配并初始化一个互斥信号量
<code>semCCreate()</code>	分配并初始化一个计数器信号量
<code>semDelete()</code>	终止并释放一个信号量
<code>semTake()</code>	取一个信号量
<code>semGive()</code>	释放一个信号量
<code>semFlush()</code>	解锁所有等待该信号量的任务

5.4.2 二进制信号量

通用的二进制信号量能够满足两种类型的任务协调需要：互斥和同步。二进制信号量需要的系统开销最小，因而特别适于高性能的需求。下一节介绍的互斥信号量也是一种二进制信号量，但是它主要用于解决内在互斥的问题。二进制信号量适用于当不需要互斥信号量的高级特征情形下的互斥。二进制信号量可以看成是一个标记：对应的资源是可用（满）还是不可用（空）。当任务调用函数 `semTake()` 取一个信号量时，其结果依赖于在调用的时刻信号量是否可用，如图 5.4。

如果此时信号量可用，调用 `semTake()` 的结果使信号量变为不可用，任务继续执行；如果此时信号量不可用，调用 `semTake()` 的任务进入一个阻塞队列，进入等待该信号量变为可用的阻塞状态。

当任务调用 `semGive()` 释放一个二进制信号量，其结果也依赖于在调用时该信号量是否可用，如图 5.5。

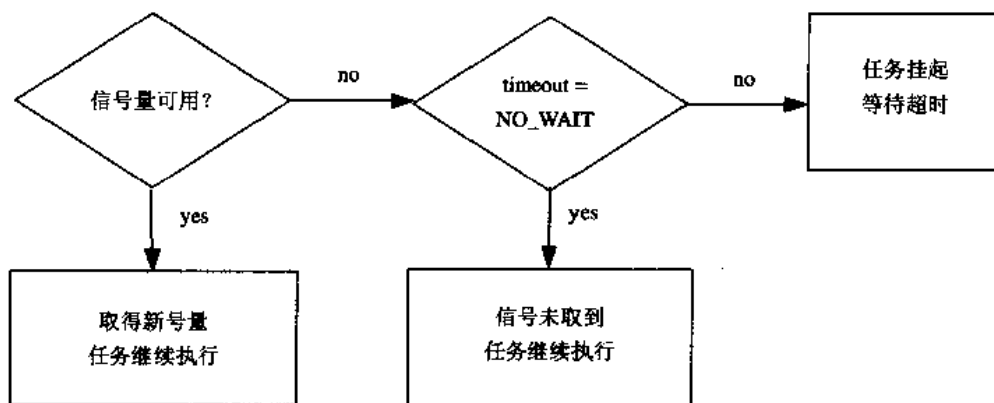


图 5.4 取一个信号量的流程

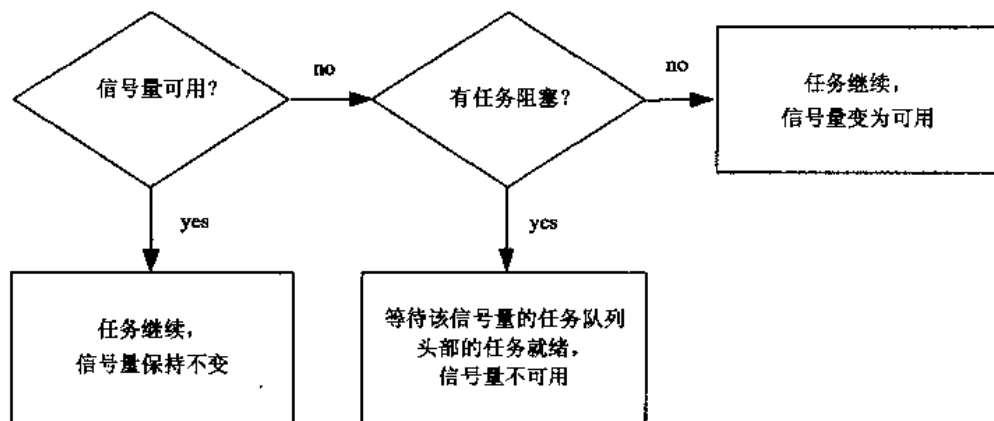


图 5.5 释放一个信号量的流程

如果此时信号量已经可用，释放信号量不产生任何影响。如果信号量不可用并且没有任务在等待它，那么信号量变为可用。如果信号量不可用并且有任务在等待它，那么阻塞在该信号量队列中的第一个任务解除阻塞，信号量仍不可用。

5.4.3 互斥

当两个或多个任务共享使用诸如同一块内存缓冲或同一个 I/O 设备之类的资源时，可能会发生竞争状态。例如 5.6 所示：五个任务共同使用一个缓冲（一个数据结构）来处理气象信息，任务 A、B、C 和 D 向缓冲写数据，任务 E 从缓冲读数据。

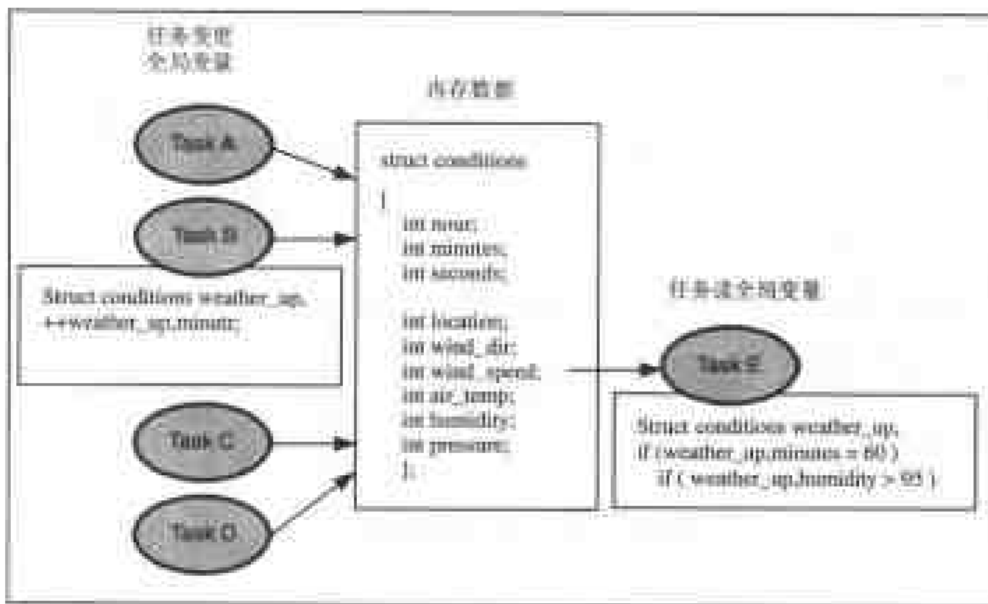


图 5.6 共享缓冲区中的数据结构数据被破坏的风险

如果这些任务运行在相同的优先级，而且系统使用轮转调度，我们考虑下述情况：当任务 B 在修改数据结构中的数据，仅仅修改了状态域，时间域还未修改，由于用完了时间片，系统发生上下文切换：由任务 B 切换到任务 E，如果任务 E 在上下文切换之后读该数据结构，读到的数据结构中的状态域是修改后的值，而时间域则还没有修改，也就是说状态和时间不一致，数据没有意义，系统中数据存在不一致性冲突。二进制信号量可以提供避免这种冲突的有效方法。

二进制信号量可以通过对共享资源上锁，实现高效的互斥访问。不像禁止中断或禁止抢占，二进制信号量将互斥仅仅限于对与之联系资源的访问。使用这种技术时，创建用于保护资源的二进制信号量，初始时信号量是可用的。

```

/* includes */
#include "vxWorks.h"
#include "semLib.h"
SEM_ID semMutex;
/* 创建一个二进制信号量，初始化为 full。
 * 阻塞在该信号量上的任务按优先级排队 Tasks
 */
semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
    
```

当任务需要访问这个资源，它必须首先要取得这个信号量。同时持有该信号量，所有其他想要访问这个资源的任务将被阻塞。当任务完成了对该资源的访问，它释放该信号量，允许其他的任务使用该资源。因此所有对一个需要互斥访问资源的操作由 `semTake()` 和



semGive()对括起:

```
semTake (semMutex, WAIT_FOREVER);
.
. 临界区, 某时刻仅能有一个任务来访问
.
semGive (semMutex);
```

5.4.4 同步

当用于任务同步时, 信号量可以作为任务等待的一个状态或事件。初始时信号量是不可用的。一个任务或中断处理程序释放该信号量来通知这个事件的发生。其他任务因调用 semTake() 等待该信号量而阻塞。等待的任务阻塞直到该事件发生和信号量释放。

注意信号量在用于互斥和同步情况下的不同的状态顺序:

- ◆ 用于互斥时, 信号量最初是满的、可用的, 每个任务首先是取, 然后放回。
- ◆ 用于同步时, 信号量最初是空的、不可用的, 一个任务首先是等待由其他任务释放的信号量。

在下例中: init() 程序创建一个二进制信号量, 将一个事件与一个中断处理程序相连接。然后发起一个任务处理这个事件。函数 task1() 一直运行到调用 semTake()。在这个点上, 它将阻塞直到一个事件发生引起 ISR (中断服务程序) 调用 semGive()。当 ISR 完成, task1() 将恢复执行, 处理这个事件。

这种将事件处理放在一个专门的任务中处理的方法具有一个优点: 仅有很少的处理放在中断级, 因而减少了中断反应事件。这种方法特别适用于实时应用。

例 5.1 使用信号量实现任务同步

```

/*****
* 该例子程序使用信号量实现任务同步
*/

/* includes */
#include "vxWorks.h"
#include "semLib.h"
#include "arch/arch/ivarch.h" /* 用特定体系结构的相应文件替代它 */

SEM_ID syncSem; /* ID of sync semaphore */

init (
    int someIntNum
```



```
    )
  {
    /* 连接中断处理程序 */
    intConnect(INUM_TO_IVEC (someIntNum), eventInterruptSvcRout, 0);

    /* 创建信号量 */
    syncSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);

    /* spawn task used for synchronization. */
    taskSpawn("sample", 100, 0, 20000, task1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
  }

task1 (void)
{
    ...
    semTake (syncSem, WAIT_FOREVER); /* wait for event to occur */
    printf ("task 1 got the semaphore\n");
    ... /* process event */
}

eventInterruptSvcRout (void)
{
    ...
    semGive (syncSem); /* let task 1 process event */
    ...
}
/*****/
```

VxWorks 还提供一种同步方式：广播同步，也就是说允许所有阻塞在同一个信号量上的任务集合全部自动解除阻塞。这种同步方式正确的应用行为常常需要一个任务集合，在这个集合中的任一任务有机会处理更多事件之前，整个集合任务处理一个事件。函数 `semFlush()` 可用于这类同步问题，它将解除阻塞在一个信号量上的所有任务。

5.4.5 互斥信号量

互斥信号量是一种特殊的二进制信号量，主要用于解决具有内在的互斥问题：优先级继承、删除安全和对资源的递归访问等情况。

互斥信号量的基本行为与二进制信号量一致。不同之处在于

- 它仅用于互斥。
- 仅能由取 (`semTake()`) 它的任务释放。



- ◆ 不能在 ISR 中释放 (semGive())。
- ◆ semFlush() 操作非法。

■ 优先级倒置

优先级倒置发生在一个高优先级的任务被迫等待一段不确定时间,等待一个低优先级任务完成。

考虑图 5.7 中的情况: t1、t2 和 t3 是优先级由高到低的三个任务。t3 占有了由二进制信号量保护的某种资源,进入临界区执行,当 t3 在临界区执行时,高优先级的任务 t1 就绪, t1 抢占 t3 执行(注意, t3 仍占有该信号量)。在 t1 执行过程中,需要进入相同的临界区执行,这时需要取得相同的信号量,这时由于资源竞争, t1 被阻塞。如果我们假定 t1 将会阻塞,直到 t3 正常地(不再被其他任务抢占)完成对该资源的访问,由于资源不能被抢占,这不会出现问题。然而这个低优先级的任务继续执行时可能被中等优先级的任务(如 t2)抢占, t3 再次阻塞,(仍然拥有信号量),这种状态可能持续下去,因而 t1 将在一段不确定的时间段内被阻塞。这种情况下,如果不采取措施,可能造成该实时系统的不可预测性。

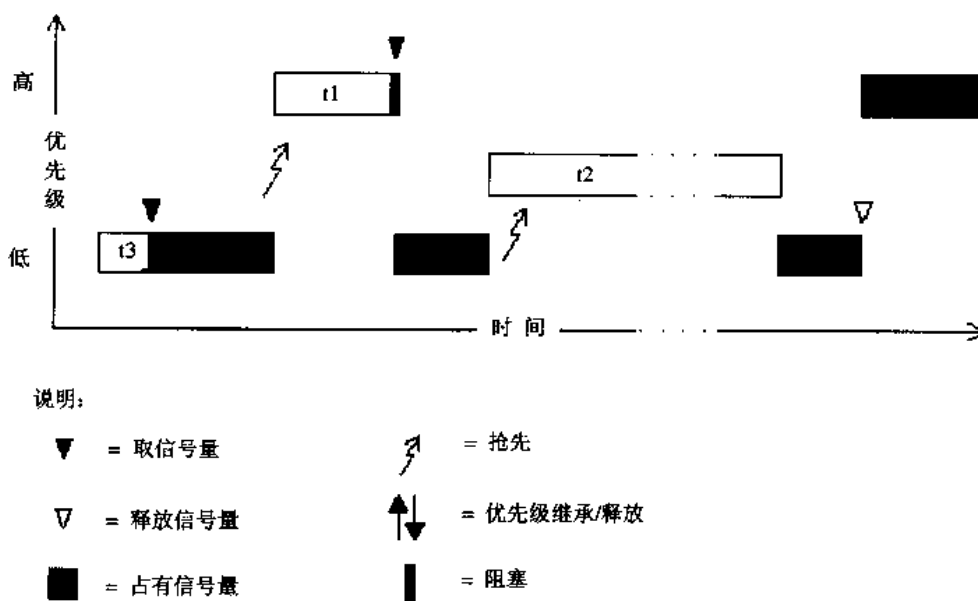


图 5.7 优先级转置

VxWorks 允许使用优先级继承算法,提供了解决这种问题的手段,保证了系统的可预测性。互斥信号量有个选项 SEM_INVERSION_SAFE,使用这个选项将使能优先级继承算法。优先级继承协议确保拥有资源的任务以阻塞在该资源上的所有任务中优先级最高的任务的优先级执行。一旦这个任务的优先级被抬高,它将以这个高优先级运行,直至它持有的所有互斥信号量全部释放,然后,该任务返回正常状态。因此,这个“继承的



高优先级”任务受到不会被任何中间优先级任务抢占的保护。这个选项必须与一个优先级队列 (SEM_Q_PRIORITY) 一齐使用。

图 5.8 中, 优先级继承解决了优先级倒置问题: 由于在 t1 阻塞在该信号量时, 把 t3 的优先级抬高到 t1 的优先级。这样保护 t3, 间接地保护了 t1 不被 t2 抢占。

下面的例子使用优先级继承算法创建一个互斥信号量。

```
semId = semCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE);
```

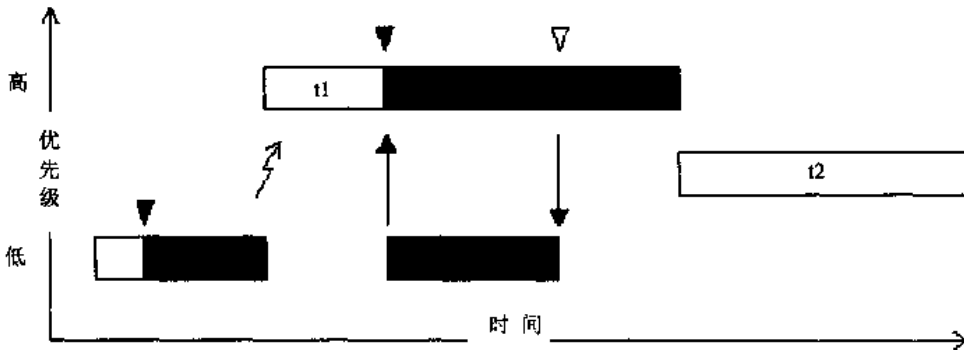


图 5.8 优先级继承

删除安全

另一个互斥问题涉及到任务删除。在一个受信号量保护的临界区, 经常需要保护在临界区执行的任务不会被意外地删除。删除一个在临界区执行的任务可能引起意想不到的后果, 造成保护资源的信号量不可用, 可能导致资源处于破坏状态。也就导致了其他要访问该资源的所有任务无法得到满足。

原语 `taskSafe()` 和 `taskUnsafe()` 提供了解决任务避免被意外删除的一种方法。同时, 互斥信号量提供了选项 `SEM_DELETE_SAFE`, 使用这个选项, 当每次调用 `semTake()` 时隐含地使能了 `taskSafe()`, 当每次调用 `semGive()` 时隐含地使能了 `taskUnsafe()`。在这种方式中, 得到信号量时, 任务能够得到不会被删除的保护。使用这个选项比原语 `taskSafe()` 和 `taskUnsafe()` 更有效, 并且使代码需要进入内核的部分更少。

```
semId = semMCreate (SEM_Q_FIFO | SEM_DELETE_SAFE);
```

递归资源访问

互斥信号量能够被递归地获取。这意味着信号量能够被一个拥有该信号量的任务在该信号量最终被释放之前多次获取。递归对于满足一些子程序即要求能够相互调用但是也要求互



斥访问一个资源非常有用。这种情形是可能的，由于系统需要跟踪哪一个任务当前拥有信号量。

在被释放之前，一个被递归获取 n 次的互斥信号量必须被释放与获取相同的 n 次。这由系统中的一个计数器来跟踪。每次 `semTake()` 调用计数器将加一，每次 `semGive()` 调用计数器将减一。

例 5.2 互斥信号量的递归使用

```
/* *****  
 * 在此例子中，funcA 调用 semTake (mySem, WAIT_FOREVER) 请求访问的一个资源，  
 * funcA 也需要调用 funcB；  
 * funcB 也将调用 semTake (mySem, WAIT_FOREVER) 请求访问的资源。  
 */  
  
/* includes */  
#include "vxWorks.h"  
#include "semLib.h"  
  
SEM_ID mySem;  
  
/* 创建一个互斥信号量 */  
init ()  
{  
    mySem = semMCreate (SEM_Q_PRIORITY);  
}  
  
funcA ()  
{  
    semTake (mySem, WAIT_FOREVER);  
    printf ("funcA: Got mutual-exclusion semaphore\n");  
    ...  
    funcB ();  
    ...  
    semGive (mySem);  
    printf ("funcA: Released mutual-exclusion semaphore\n");  
}  
  
funcB ()  
{  
    semTake (mySem, WAIT_FOREVER);  
    printf ("funcB: Got mutual-exclusion semaphore\n");  
    ...  
}
```



```

semGive (mySem);
printf ("funcB: Releases mutual-exclusion semaphore\n");
}
/*****/

```

5.4.6 计数器信号量

计数器信号量是实现任务同步和互斥的另一种手段。计数器信号量除了像二进制信号量那样工作外，它还保持对信号量释放次数的跟踪。信号量每次释放，计数器加一，每次获取，计数器减一。当计数器减到 0，试图获取该信号量的任务被阻塞。

正如二进制信号量，当信号量释放时，如果有任务阻塞在该信号量阻塞队列上，那么任务解除阻塞。然而，不像二进制信号量，如果信号量释放时，没有任务阻塞在该信号量阻塞队列上，那么计数器加一。

这意味着释放信号量两次，那么就有两次信号量获取操作不会阻塞。

表 5.2 示意了任务获取和释放一个初始值为 3 的计数器信号量的状态时间顺序。

表 5.2 计数器信号量实例

信号量调用	操作后的计数器值	调用导致的行为
semCCreate()	3	以计数器值为 3，信号量初始化
semTake()	2	信号量获取
semTake()	1	信号量获取
semTake()	0	信号量获取
semTake()	0	任务阻塞等待信号量可用
semGive()	0	信号量释放，阻塞的任务就绪
semGive()	1	信号量释放，没有阻塞的任务，计数器增加

计数器信号量适用于保护拥有多个拷贝的资源。例如如果系统中有 5 个相同的磁带驱动器，系统可以使用一个初始值为 5 的计数器信号量来协调这些磁带驱动器工作。如果应用程序需要使用 256 个入口的环形缓冲管理，可以使用初始值为 256 的计数器信号量来实现的。计数器信号量的初始值作为 semCCreate() 函数的一个参数告诉系统。

5.4.7 特定的信号量选项

Wind 标准信号量接口包括两个特定的选项，这对于后面将要所介绍的 POSIX 信号量兼



容接口是无效的。

➤ 超时

Wind 信号量提供在阻塞状态判断超时的能力。这由 `semTake()` 特定的参数来控制，这个参数是一定量的系统 ticks 数目，表示任务将要在阻塞态等待的时间。如果任务在规定的时间内成功地获得了信号量，`semTake()` 返回 OK。当规定的超时值已过，`semTake()` 未能成功取得新号量而返互 ERROR，系统设置 `errno` 的值。

一个以 `NO_WAIT (=0)` 作为参数的 `semTake()`调用，告诉系统，不要等待，如果调用时请求的信号量不可用，将直接返回，系统将 `errno` 设置为 `S_objLib_OBJ_UNAVAILABLE`。一个以正整数 `n` 作为参数的 `semTake()`调用，如果等待 `n` 个 tick 后，请求的信号量仍不可用，系统将 `errno` 设置为 `S_objLib_OBJ_TIMEOUT`，调用返回。一个以 `WAIT_FOREVER (=1)` 作为参数的 `semTake()` 调用将意味着无限期的等待，一直等到请求的信号量可用。

➤ 队列

Wind 信号量具备可以将阻塞在该信号量上的任务排队的能力。这些任务可以基于先进先出 (FIFO) 或优先级顺序两个标准中的任一排队，如图 5.9。

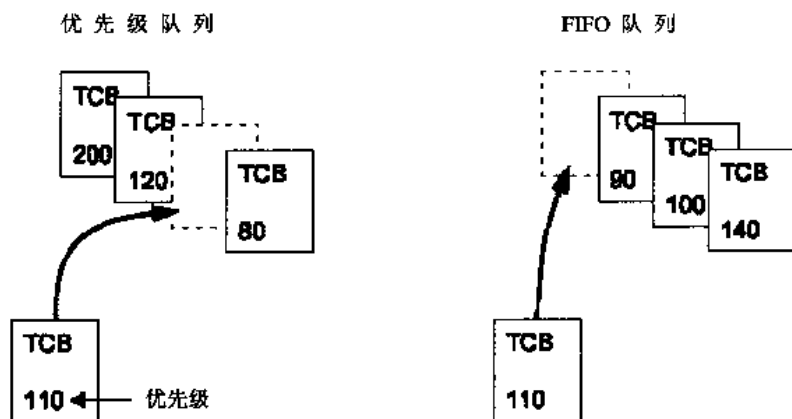


图 5.9 两种任务队列类型

优先级顺序更好地维持了系统按优先级构造的意图，但是也增加了用于 `semTake()` 阻塞任务按优先级排序排队的代价。FIFO 队列则不需要这种排序排队的花销，保证系统有一个恒定的时间性能。队列类型在信号量创建时确定，也就是调用 `semBCreate()`，`semMCreate()` 或 `semCCreate()` 时。使用优先级继承的信号量必须选择使用基于优先级的队列。



5.4.8 POSIX 信号量

■ POSIX 信号量函数接口

POSIX 定义了命名和未命名两种信号量。他们有相同的优先级，但是接口略微不同。POSIX 信号量库提供了创建、打开和摧毁命名和未命名两种信号量的函数接口。

semPxBLib 提供的 POSIX 信号量函数接口如表 5.3 所示。

表 5.3 POSIX 信号量函数调用

调 用	描 述
semPxBLibInit()	初始化 POSIX 信号量库 (non-POSIX)
sem_init()	初始化一个未命名 POSIX 信号量
sem_destroy()	摧毁未命名 POSIX 信号量
sem_open()	初始化打开一个命名 POSIX 信号量
sem_close()	关闭一个命名 POSIX 信号量
sem_unlink()	移出一个命名 POSIX 信号量
sem_wait()	上锁一个 POSIX 信号量
sem_trywait()	上锁一个 POSIX 信号量仅当它未上锁
sem_post()	解锁一个 POSIX 信号量
sem_getvalue()	得到一个 POSIX 信号量值

警告

sem_destroy() 调用终结一个未命名信号量，释放与之联系的所有内存。对于命名信号量，结合使用 sem_close() 和 sem_unlink() 有相同的效果。当删除信号量，特别是删除互斥信号量时应特别小心，避免删除一直被其他任务需要的信号量。不要删除一个信号量，除非该删除任务首先成功上锁该信号量。对于命名信号量，信号量的关闭仅能来自于打开它的任务。

使用命名信号量，可以在打开一个信号量时指定一个名字，其他的命名信号量函数将以这个名字作为参数，加以识别。

一些主机操作系统如 UNIX，要求信号量使用可以被其他进程共享的符号名字，这是因为在这些系统中，进程不能像操作系统那样正常地共享内存。在 VxWorks 中，则不需要命名信号量，因为所有对象都位于一个单一的地址空间，可以以内存定位的方式引用所有的共享对象。



POSIX 术语 wait (或 lock) 和 post (或 unlock) 各自对应于 VxWorks 术语 take 和 give。

如果在配置 VxWorks 时包含了 INCLUDE_POSIX_SEM 的定义, 初始化程序 semPxLibInit() 将由系统默认调用。函数 sem_open()、sem_unlink() 和 sem_close() 仅用于命名信号量的打开、关闭/摧毁; sem_init() 和 sem_destroy() 用于未命名信号量的初始化和摧毁。其他的上锁、解锁和得到信号量的值的函数可以用于两种类型的信号量。

■ POSIX 和 Wind 信号量的比较

POSIX 信号量是计数信号量, 也就说它跟踪被释放的次数。

除了 Wind 信号量提供一些附加的特征: 优先级继承、任务删除安全、单个任务多次获取一个信号量、互斥信号量的所有、信号量超时、排队选择等特征之外, Wind 信号量机制与 POSIX 说明中的相似。但是, Wind 信号量提供这些特征非常重要, 可以说 Wind 信号量更容易满足应用要求。

■ 使用未命名信号量

使用未命名信号量时, 一般是在一个任务中为这个信号量分配内存并初始化它。

一个未命名信号量对应一个数据结构 sem_t, 定义在 semaphore.h。信号量初始化函数 sem_init() 允许说明信号量初始值, 任何任务都可调用 sem_wait() (阻塞方式) 或 sem_trywait() (非阻塞方式) 上锁该信号量, 调用 sem_post() 解锁它。

正如前面提到的, 信号量可以用于同步和互斥。当信号量用于同步, 一般要初始化为 0, 等待要同步的任务阻塞在 sem_wait() 调用上。任务调用 sem_post() 解锁信号量, 来达到实现同步。如果仅有一个任务阻塞在该信号量上, 这个任务将就绪; 如果有多个任务阻塞在该信号量上, 那么优先级最高的任务就绪。

当信号量用于互斥时, 一般要初始化为一个大于 0 的值 (意味着资源可用)。因此, 第一个上锁该信号量的任务可以立即执行, 不会阻塞。后续的任务将会阻塞 (如果信号量的值初始化为 1)。

例 5.3 POSIX 未命名信号量

```

/*****
 * 该例子使用 POSIX 未命名信号量同步调用任务
 * 以及一个它发起的任务 (tSyncTask) 之间的活动
 * 在 shell 将该函数作为一个任务调用
 * -> sp unnameSem
 */

/* includes */
#include "vxWorks.h"

```



```
#include "semaphore.h"

/* forward declarations */
void syncTask (sem_t * pSem);

void unnameSem (void)
{
    sem_t * pSem;

    /* 为信号量申请内存*/
    pSem = (sem_t *) malloc (sizeof (sem_t));

    /* 初始化信号量, 初始状态为不可用 */
    if (sem_init (pSem, 0, 0) == -1)
    {
        printf ("unnameSem: sem_init failed\n");
        return;
    }

    /*发起同步任务*/
    printf ("unnameSem: spawning task...\n");
    taskSpawn ("tSyncTask", 90, 0, 2000, syncTask, pSem);

    /* 处理事务*/
    /* 解锁信号量 */
    printf ("unnameSem: posting semaphore - synchronizing action\n");
    if (sem_post (pSem) == -1)
    {
        printf ("unnameSem: posting semaphore failed\n");
        return;
    }

    /* 完成后, 摧毁信号量*/
    if (sem_destroy (pSem) == -1)
    {
        printf ("unnameSem: sem_destroy failed\n");
        return;
    }
}

void syncTask
(
```



```

sem_t * pSem
)
{
    /* 等待同步*/
    if (sem_wait (pSem) == -1)
    {
        printf ("syncTask: sem_wait failed \n");
        return;
    }
    else
        printf ("syncTask:sem locked; doing sync'ed action...\n");
    /* 处理事务 */
}

```

■ 使用命名信号量

sem_open() 函数既可以用来打开一个已经存在的命名信号量，作为可选，也可以创建一个新的信号量。以下是可用于组合的选项：

- ◆ O_CREAT 创建一个新信号量，如果它不存在的话（如果它存在，或者失败或者打开该信号量，由是否使用 O_EXCL 选项决定）。
- ◆ O_EXCL 打开新创建的信号量，如果已经存在则失败。

sem_open() 调用的结果依赖于设置的选项和访问的信号量是否已经存在。如表 5.4 所示。

表 5.4 调用 sem_open()可能的结果

标记设置	信号量存在	信号量不存在
None	信号量被打开	程序失败
O_CREAT	信号量被打开	信号量创建
O_CREAT 和 O_EXCL	程序失败	信号量创建

表中没有单独以 O_EXCL 作为参数的情形，这是因为以它单独作为参数没有意义。

一个 POSIX 命名信号量，一旦初始化，在明确摧毁之前，它将一直可用。任何时候，任务都可以明确表示一个信号量被摧毁，但是该信号量将一直存在于系统中，直到没有打开该信号量的任务为止。

如果在配置 VxWorks 时选择包含 INCLUDE_POSIX_SEM_SHOW，就可以在 Tornado shell 使用 show() 函数显示 POSIX 信号量的有关信息。注意，这不是一个 POSIX 程序，也不能在程序中使用，只能在 Tornado shell 中使用。



```
-> show semId  
value = 0 = 0x0
```

show() 函数输出送到标准输出设备, 显示的信息包含 POSIX 信号量 mySem 的有关信息:

```
Semaphore name :mySem  
sem_open() count :3  
Semaphore value :0  
No. of blocked tasks :2
```

对于一组合作使用同一个命名信号量的任务组, 其中一个任务应该首先创建并初始化该信号量, 调用以下语句:

```
sem_open(O_CREAT)
```

其他使用该信号量的任务然后以相同的名字作为参数调用 sem_open() (不要设置 O_CREAT) 打开该信号量。任何已经打开该信号量的任务调用 sem_wait() (阻塞) 或 sem_trywait() (非阻塞) 上锁该信号量, 或调用 sem_post() 解锁信号量。

为移出一个信号量, 所有使用它的任务必须首先调用 sem_close() 关闭它, 并且其中的一个任务必须调用 sem_unlink() 解开连接, 其作用是将该信号量的名字从名字表中移出。一旦这个名字已经从名字表中移出, 当前已经打开该信号量的任务仍然可以使用它, 但是新的任务就不能再打开该信号量。下一次试图打开该信号量的任务, 如果没有使用 O_CREAT 作为参数, 打开操作将失败。当最后一个任务关闭该信号量后, 该信号量将消失。

例 5.4 POSIX 命名信号量

```
/*  
* In this example, nameSem() creates a task for synchronization. The  
* new task, tSyncSemTask, blocks on the semaphore created in nameSem().  
* Once the synchronization takes place, both tasks close the semaphore,  
* and nameSem() unlinks it. To run this task from the shell, spawn  
* nameSem as a task:  
* -> sp nameSem, "myTest"  
*/  
  
/* includes */  
#include "vxWorks.h"  
#include "semaphore.h"  
#include "fcntl.h"  
/* forward declaration */
```



```
int syncSemTask (char * name);

int nameSem
(
    char * name
)
{
    sem_t * semId;
    /* create a named semaphore, initialize to 0*/
    printf ("nameSem: creating semaphore\n");
    if ((semId = sem_open (name, O_CREAT, 0, 0)) == (sem_t *) -1)
    {
        printf ("nameSem: sem_open failed\n");
        return;
    }
    printf ("nameSem: spawning sync task\n");
    taskSpawn ("tSyncSemTask", 90, 0, 2000, syncSemTask, name);
    /* do something useful to synchronize with syncSemTask */
    /* give semaphore */
    printf ("nameSem: posting semaphore - synchronizing action\n");
    if (sem_post (semId) == -1)
    {
        printf ("nameSem: sem_post failed\n");
        return;
    }
    /* all done */
    if (sem_close (semId) == -1)
    {
        printf ("nameSem: sem_close failed\n");
        return;
    }
    if (sem_unlink (name) == -1)
    {
        printf ("nameSem: sem_unlink failed\n");
        return;
    }
    printf ("nameSem: closed and unlinked semaphore\n");
}

int syncSemTask
(
    char * name
)
```



```
(  
    sem_t * semId;  
    /* open semaphore */  
    printf ("syncSemTask: opening semaphore\n");  
    if ((semId = sem_open (name, 0)) == (sem_t *) -1)  
    {  
        printf ("syncSemTask: sem_open failed\n");  
        return;  
    }  
    /* block waiting for synchronization from nameSem */  
    printf ("syncSemTask: attempting to take semaphore...\n");  
    if (sem_wait (semId) == -1)  
    {  
        printf ("syncSemTask: taking sem failed\n");  
        return;  
    }  
    printf ("syncSemTask: has semaphore, doing sync'ed action ...\n");  
    /* do something useful here */  
    if (sem_close (semId) == -1)  
    {  
        printf ("syncSemTask: sem_close failed\n");  
        return;  
    }  
}  
/*****/
```

5.4.9 信号量在多任务中的应用

本节给出几个例子，演示信号量在多任务编程中的应用。

■ 互斥信号量用于任务间同步

本例演示了任务间使用互斥信号量实现同步的方法。例子中的两个任务（生产者与消费者）使用互斥信号量实现对一个共享内存数据的互斥访问。

程序首先创建一个用于生产者与消费者实现同步的互斥信号量。生产者任务与消费者任务同时访问共享内存数据。为了避免这个共享数据结构受到破坏，需要使用互斥信号量。

程序然后发起（spawn）生产者任务，它生产信息并将信息写入共享数据结构中；发起消费者任务，它从共享数据结构读取消费信息并修改状态域为 CONSUMED。这样，生产者任务可以向共享数据结构写入下一条信息。一旦消费者任务已经消费了所用信息，将删除互



斥信号量。

例 5.5 互斥信号量用于任务间同步

```

/*****
/* mutexSemDemo.h - Header for the mutexSemDemo */

/* Copyright 1984-1997 Wind River Systems, Inc. */

/*
modification history
-----
01b, 06nov97, mm added copyright.
01a, 14jan94, ms written.
*/

#define CONSUMER_TASK_PRI      98 /* Priority of the consumerTask task*/
#define PRODUCER_TASK_PRI     99 /* Priority of the producerTask task*/
#define TASK_STACK_SIZE      5000 /* Stack size for spawned tasks */
#define PRODUCED 1           /* Flag to indicate produced status*/
#define CONSUMED 0          /* Flag to indicate consumed status*/
#define NUM_ITEMS 5         /* Number of items */
struct shMem                 /* Shared Memory data structure */
{
    int tid;                  /* task id */
    int count;                /* count number of item produced */
    int status;               /* 0 if consumed or 1 if produced*/
};

/* mutexSemDemo.c - Demonstrate the usage of the mutual exclusion semaphore
 *                   for intertask synchronization and obtaining exclusive
 *                   access to a data structure shared among multiple tasks.
 */

/* Copyright 1984-1997 Wind River Systems, Inc. */

/*
modification history
-----
01c, 06nov97, mm added copyright.
01b, 16sep97, ram included logLib.h, sysLib.h, stdio.h
        The arguments to logMsg are required arguments(6)

```



Since there were fewer than 6 arguments the remaining
have been filled up with zeros.

Ola, 14jan94, ms written.

*/

#include "vxWorks.h"

#include "semLib.h"

#include "taskLib.h"

#include "mutexSemDemo.h"

#include "logLib.h"

#include "sysLib.h"

#include "stdio.h"

```
LOCAL STATUS protectSharedResource (); /* protect shared data structure */
LOCAL STATUS releaseProtectedSharedResource (); /* release protected access */
LOCAL STATUS producerTask (); /* producer task */
LOCAL STATUS consumerTask (); /* consumer task */
LOCAL struct shMem shMemResource; /* shared memory structure */
LOCAL SEM_ID mutexSemId; /* mutual exclusion semaphore id*/
LOCAL BOOL notFinished; /* Flag that indicates the
                          * completion */
```

/*****

* mutexSemDemo - Demonstrate the usage of the mutual exclusion semaphore
* for intertask synchronization and obtaining exclusive
* access to a data structure shared among multiple tasks.

*

* DESCRIPTION

* Creates a mutual exclusion semaphore for intertask synchronization
* between the producerTask and the consumerTask. Both producerTask and
* consumerTask access and manipulate the global shared memory data
* structure simultaneously. To avoid corruption of the global shared
* memory data structure mutual exclusion semaphores are used.

*

* Spawns the producerTask that produces the message and puts the message
* in the global shared data structure. Spawns the consumerTask that
* consumes the message from the global shared data structure and
* updates the status field to CONSUMED so that producerTask can put
* the next produced message in the global shared data structure.
* After consumerTask has consumed all the messages, the mutual exclusion
* semaphore is deleted.

*



```
* RETURNS: OK or ERROR
*
* EXAMPLE
*
* -> sp mutexSemDemo
*
*/

STATUS mutexSemDemo()
{
    notFinished = TRUE; /* initialize the global flag */

    /* Create the mutual exclusion semaphore*/
    if ((mutexSemId = semMCreate (SEM_Q_PRIORITY | SEM_DELETE_SAFE
        | SEM_INVERSION_SAFE)) == NULL)
    {
        perror ("Error in creating mutual exclusion semaphore");
        return (ERROR);
    }

    /* Spwan the consumerTask task */
    if (taskSpawn ("tConsumerTask", CONSUMER_TASK_PRI, 0, TASK_STACK_SIZE,
        (FUNCPTR) consumerTask, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
        == ERROR)
    {
        perror ("consumerTask: Error in spawning demoTask");
        return (ERROR);
    }

    /* Spwan the producerTask task */
    if (taskSpawn ("tProducerTask", PRODUCER_TASK_PRI, 0, TASK_STACK_SIZE,
        (FUNCPTR) producerTask, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
        == ERROR)
    {
        perror ("producerTask: Error in spawning demoTask");
        return (ERROR);
    }

    /* Polling is not recommended. But used for making this demonstration
    * simple */
}
```



```
while (notFinished)
    taskDelay (sysClkRateGet ());

/* When done delete the mutual exclusion semaphore*/
if (semDelete (mutexSemId) == ERROR)
{
    perror ("Error in deleting mutual exclusion semaphore");
    return (ERROR);
}

return (OK);
}

/*****
 * producerTask - produce the message, and write the message to the global
 *                 shared data structure by obtaining exclusive access to
 *                 that structure which is shared with the consumerTask.
 *
 * RETURNS: OK or ERROR
 *
 */

STATUS producerTask ()
{
    int count = 0;
    int notDone = TRUE;

    while (notDone)
    {

        /* Produce NUM_ITEMS, write each of these items to the shared
         * global data structure.
         */

        if (count < NUM_ITEMS)
        {

            /* Obtain exclusive access to the global shared data structure */
            if (protectSharedResource() == ERROR)
                return (ERROR);

            /* Access and manipulate the global shared data structure */
            if (shMemResource.status == CONSUMED)
```



```

    {
        count++;
        shMemResource.tid = taskIdSelf ();
        shMemResource.count = count;
        shMemResource.status = PRODUCED;
    }
    /* Release exclusive access to the global shared data structure */
    if (releaseProtectedSharedResource () == ERROR)
        return (ERROR);

    logMsg ("ProducerTask: tid = %#x, producing item = %d\n",
            taskIdSelf (), count, 0, 0, 0, 0);
    taskDelay (sysClkRateGet()/6); /* relinquish the CPU so that
        * consumerTask can access the
        * global shared data structure.
        */
    }
    else
        notDone = FALSE;
    }

    return (OK);
}

/*****
 * consumerTask - consumes the message from the global shared data
 *                 structure and updates the status field to CONSUMED
 *                 so that producerTask can put the next produced message
 *                 in the global shared data structure.
 *
 * RETURNS: OK or ERROR
 *
 */

STATUS consumerTask ()
{
    int notDone = TRUE;

    /* Initialize to consumed status */
    if (protectSharedResource() == ERROR)
        return (ERROR);
    shMemResource.status = CONSUMED;

```



```

if (releaseProtectedSharedResource () == ERROR)
    return (ERROR);

while (notDone)
{
    taskDelay (sysClkRateGet()/6);    /* relinquish the CPU so that
        * producerTask can access the
        * global shared data structure.
        */

    /* Obtain exclusive access to the global shared data structure */
    if (protectSharedResource() == ERROR)
        return (ERROR);

    /* Access and manipulate the global shared data structure */
    if ((shMemResource.status == PRODUCED) && (shMemResource.count > 0))
    {
        logMsg ("ConsumerTask: Consuming item = %d from tid = %x\n\n",
            shMemResource.count, shMemResource.tid, 0, 0, 0, 0);
        shMemResource.status = CONSUMED;
    }
    if (shMemResource.count >= NUM_ITEMS)
        notDone = FALSE;

    /* Release exclusive access to the global shared data structure */
    if (releaseProtectedSharedResource () == ERROR)
        return (ERROR);

}

notFinished = FALSE;
return (OK);
}

/*****
* protectSharedResource - Protect access to the shared data structure with
*                         the mutual exclusion semaphore.
*
* RETURNS: OK or ERROR
*
*/

```



```

LOCAL STATUS protectSharedResource ()
{
    if (semTake (mutexSemId, WAIT_FOREVER) == ERROR)
    {
        perror ("protectSharedResource: Error in semTake");
        return (ERROR);
    }
    else
        return (OK);
}

/*****
 * releaseProtectedSharedResource - Release the protected access to the
 *                                 shared data structure using the mutual
 *                                 exclusion semaphore
 *
 * RETURNS: OK or ERROR
 *
 */
LOCAL STATUS releaseProtectedSharedResource ()
{
    if (semGive (mutexSemId) == ERROR)
    {
        perror ("protectSharedResource: Error in semTake");
        return (ERROR);
    }
    else
        return (OK);
}

/*****/

```

编译例子代码、下载目标代码到目标机。

在 VxWorks 目标机执行命令：

```

-> ld sp mutexSemDemo
task spawned: id = 5b7b40, name = u2
value = 5995328 = 0x5b7b40

```

输出信息如下：

```

semaphore taken 2 times
0x5b2b18 (tProducerTask): ProducerTask: tid = 0x5b2b18, producing item = 1

```



```

0x7fd030 (tConsumerTask): ConsumerTask: Consuming item = 1 from tid = 0x5b2b18

0x5b2b18 (tProducerTask): ProducerTask: tid = 0x5b2b18, producing item = 2
0x7fd030 (tConsumerTask): ConsumerTask: Consuming item = 2 from tid = 0x5b2b18

0x5b2b18 (tProducerTask): ProducerTask: tid = 0x5b2b18, producing item = 3
0x7fd030 (tConsumerTask): ConsumerTask: Consuming item = 3 from tid = 0x5b2b18

0x5b2b18 (tProducerTask): ProducerTask: tid = 0x5b2b18, producing item = 4
0x7fd030 (tConsumerTask): ConsumerTask: Consuming item = 4 from tid = 0x5b2b18

0x5b2b18 (tProducerTask): ProducerTask: tid = 0x5b2b18, producing item = 5
0x7fd030 (tConsumerTask): ConsumerTask: Consuming item = 5 from tid = 0x5b2b18
    
```

■ 二进制信号量用于任务间同步

本例演示任务间使用二进制信号量实现同步。

例子程序创建两个二进制信号量，用来实现两个任务（任务 A 和任务 B）间的同步。任务 A 需要操作事件 A。在事件 A 完成后，任务 B 需要操作另一个事件 B。在事件 B 完成后，任务 A 需要操作事件 A。这种操作将交替进行。

例 5.6 二进制信号量用于任务间同步

```

/*****
/* synchronizeDemo.c - Demonstrates intertask synchronization using binary
*      semaphores.
*/

/* Copyright 1984-1997 Wind River Systems, Inc. */

/*
modification history
-----
01c, 06nov97, mm added copyright.
01b, 16sep97, ram included stdio.h, sysLib.h
01a, 14feb94, ms  written.
*/

#include "vxWorks.h"
#include "taskLib.h"
#include "semLib.h"
    
```



```

#include "stdio.h"
#include "sysLib.h"

#define TASK_PRI          98  /* Priority of the spawned tasks */
#define TASK_STACK_SIZE  5000 /* stack size for spawned tasks */

LOCAL SEM_ID semId1;      /* semaphore id of binary semaphore 1 */
LOCAL SEM_ID semId2;      /* semaphore id of binary semaphore 2 */
LOCAL int numTimes = 3;   /* Number of iterations */
LOCAL BOOL notDone;      /* flag to indicate completion */
LOCAL STATUS taskA ();
LOCAL STATUS taskB ();

/*****
 * synchronizeDemo - Demonstrates intertask synchronization using binary
 *                   semaphores.
 *
 * DESCRIPTION
 * Creates two (semId1 and semId2) binary semaphores for intertask
 * synchronization between two tasks (taskA and taskB). taskA need to execute
 * an event (event A). On completion of event A, taskB needs to execute
 * another event (event B). On completion of the event B, taskA needs to
 * execute event A. This process needs to be done iteratively. synchronizeDemo
 * executes this process.
 *
 * RETURNS: OK or ERROR
 *
 * EXAMPLE
 *
 * -> sp synchronizeDemo
 *
 */

STATUS synchronizeDemo ()
{
    notDone = TRUE;

    /* semaphore semId1 is available after creation*/
    if ((semId1 = semBCreate (SEM_Q_PRIORITY, SEM_FULL)) == NULL)
    {
        perror ("synchronizeDemo: Error in creating semId1 semaphore");
        return (ERROR);
    }
}

```



```
/* semaphore semId2 is not available after creation*/
if ((semId2 = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY)) == NULL)
{
    perror ("synchronizeDemo: Error in creating semId2 semaphore");
    return (ERROR);
}

/* Spwan taskA*/
if (taskSpawn ("tTaskA", TASK_PRI, 0, TASK_STACK_SIZE, (FUNCPTR) taskA, 0,
              0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
{
    perror ("synchronizeDemo: Error in spawning taskA");
    return (ERROR);
}

/* Spwan taskB*/
if (taskSpawn ("tTaskB", TASK_PRI, 0, TASK_STACK_SIZE, (FUNCPTR) taskB, 0,
              0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
{
    perror ("synchronizeDemo: Error in spawning taskB");
    return (ERROR);
}

/* Polling is not recommended. But used for simple demonstration purpose */
while (notDone)
    taskDelay (sysClkRateGet()); /* wait here until done */
/* Delete the created semaphores */
if (semDelete (semId1) == ERROR)
{
    perror ("synchronizeDemo: Error in deleting semId1 semaphore");
    return (ERROR);
}
if (semDelete (semId2) == ERROR)
{
    perror ("synchronizeDemo: Error in deleting semId1 semaphore");
    return (ERROR);
}
printf ("\n\n synchronizeDemo now completed \n");

return (OK);
}
```



```
/******
 * taskA - executes event A first and wakes up taskB to excute event B next
 *         using binary semaphores for synchronization.
 *
 * RETURNS: OK or ERROR
 *
 */

LOCAL STATUS taskA ()
{
    int count;

    for (count = 0; count < numTimes; count++)
    {
        if (semTake (semId1, WAIT_FOREVER) == ERROR)
        {
            perror ("taskA: Error in semTake");
            return (ERROR);
        }
        printf ("taskA: Started first by taking the semId1 semaphore - %d times\n",
                (count + 1));
        printf("This is task <%s> : Event A now done\n", taskName (taskIdSelf()));
        printf("taskA: I'mdone, taskBcannot proceed; Releasing semId2 semaphore\n\n");
        if (semGive (semId2) == ERROR)
        {
            perror ("taskA: Error in semGive");
            return (ERROR);
        }
    }
    return (OK);
}

/******
 * taskB - executes event B first and wakes up taskA to excute event A next
 *         using binary semaphores for synchronization.
 *
 * RETURNS: OK or ERROR
 *
 */

LOCAL STATUS taskB()
{
```



```
int count;

for (count = 0; count < numTimes; count++)
{
    if (semTake (semId2, WAIT_FOREVER) == ERROR)
    {
        perror ("taskB: Error in semTake");
        return (ERROR);
    }
    printf ("taskB: Synchronized with taskA's release of semId2 - %d times\n",
           (count + 1 ));
    printf("This is task <%s> : Event B now done\n", taskName (taskIdSelf()));
    printf("taskB: I'm done , taskA can now proceed; Releasing semId1
semaphore\n\n\n");
    if (semGive (semId1) == ERROR)
    {
        perror ("taskB: Error in semGive");
        return (ERROR);
    }
}
notDone = FALSE;
return (OK);
}
/*****/
```

编译例子代码、下载目标代码到目标机。

在 VxWorks 目标机上执行命令：

```
-> ld synchronizeDemo
value = 0 = 0x0
```

输出信息如下：

```
taskA: Started first by taking the semId1 semaphore - 1 times
This is task : Event A now done
taskA: I'm done, taskB can now proceed; Releasing semId2 semaphore

taskB: Synchronized with taskA's release of semId2 - 1 times
This is task : Event B now done
taskB: I'm done, taskA can now proceed; Releasing semId1 semaphore
```



```

taskA: Started first by taking the semId1 semaphore - 2 times
This is task : Event A now done
taskA: I'm done, taskB can now proceed; Releasing semId2 semaphore

taskB: Synchronized with taskA's release of smes
This is task : Event B now done
taskB: I'm done, taskA can now proceed; Releasing semId1 semaphore

taskA: Started first by taking the semId1 semaphore - 3 times
This is task : Event A now done
taskA: I'm done, taskB can now proceed; Releasing semId2 semaphore

taskB: Synchronized with taskA's release of semId2 - 3 times
This is task : Event B now done
taskB: I'm done, taskA can now proceed; Releasing semId1 semaphore

synchronizeDemo now completed

```

■ 计数器信号量用于任务间同步

二进制信号量可以用于任务间的同步，但是如果事件发生的足够快，可能导致数据丢失。也就是说，如果事件产生的速度超过任务处理该事件的速度，就可能发生数据丢失。使用计数器信号量可以解决这一问题。

例子程序演示任务使用计数器信号量进行同步。当然，也可使用二进制信号量替代计数器信号量，以比较二者的异同。

例 5.7 计数器信号量用于任务间同步

```

/*****
/* countingSemDemo.c - Demonstrates task synchronization using counting
*      semaphores.
*/

/* Copyright 1984-1997 Wind River Systems, Inc. */

/*
modification history
-----
01c, 06nov97, mm added copyright.
01b, 16sep97, ram included files stdio.h, taskLib.h, usrLib.h, sysLib.h
      Used the semShow() function to display info about

```



```
the semaphore inplace of the show() function.
01a, 27mar94, ms cleaned up for VxDemo
*/

/*
 * DESCRIPTION
 * Counting semaphore example. Using binary semaphores for task
 * synchronization may, if the events can occur rapidly enough, cause
 * a loss of data, i.e. an event can be missed if the events can occur
 * faster then a task can process them. Using counting semaphores may
 * solve this problem. This program demonstrates task synchronization using
 * counting semaphores. User can also select to use a binary semaphore instead
 * of a counting semaphore in this demonstration, for comparision between the
 * two semaphores.
 *
 * RETURNS: OK or ERROR
 *
 * EXAMPLE
 *
 * -> sp countingSemDemo, typeOfSemaphore
 *
 * where typeOfSemaphore is the value of the type of semaphore to be used
 * for task synchronization in this demonstration. For using counting
 * semaphores use a value 'c' or 'C' for typeOfSemaphore parameter and for
 * using binary semaphores use a value 'b' or 'B' for typeOfSemaphore
 * parameter.
 *
 * example:
 * -> sp countingSemDemo, 'c'
 * -> sp countingSemDemo, 'b'
 *
 */

/* include files */

#include "vxWorks.h"
#include "wdLib.h"
#include "stdio.h"
#include "semLib.h"
#include "taskLib.h"
#include "usrLib.h"
#include "sysLib.h"
```



```

/* defines */

#define TASK_PRIORITY          101
#define TASK_STACK_SIZE       5000
#define TIME_BETWEEN_INTERRUPTS 1 /* 1 tick */
#define TASK_WORK_TIME        2 /* 2 ticks */
#define NUM_OF_GIVES          3

/* globals */

LOCAL SEM_ID semId = NULL; /* counting or binary semaphore ID */
LOCAL WDOG_ID wdId = NULL; /* watchdog ID */
LOCAL int syncTaskId = 0; /* tid of syncTask */
LOCAL int numToGive = NUM_OF_GIVES; /* Number of times semGive is called */

/* forward declaratiuon */
void syncISR(int); /* ISR to unblock syncTask */
void cleanUp (); /* cleanup routine */
void syncTask (); /* task that needs to be synchronized
                  * with external events */

/*****
 * countingSemDemo - demonstrates task synchronization using counting
 * semaphores. User can also select to use binary semaphore instead of
 * counting semaphore in this demonstration, for comparision between the two
 * semaphores.
 *
 * RETURNS: OK or ERROR
 *
 */

STATUS countingSemDemo
(
  char semType /* counting semaphore type 'c' or binary semaphore
               * type 'b'
               */
)
{
  switch (semType)
  {
    case 'c':
    case 'C':
      if ((semId = semCCreate (SEM_Q_PRIORITY, 0)) == NULL)

```



```
        {
            perror ("semCCreate");
            return (ERROR);
        }
        break;

    case 'b':
    case 'B':
        if ((semId = semBCreate(SEM_Q_PRIORITY, SEM_EMPTY)) == NULL)
        {
            perror ("semBCreate");
            return (ERROR);
        }
        break;

    default:
        printf ("Unknown semType -- must be 'c' or 'b'\n");
        return (ERROR);
    }

    if ((wdId = wdCreate()) == NULL)
    {
        perror ("wdCreate");
        cleanUp ();
        return (ERROR);
    }

    if ((syncTaskTid = taskSpawn ("tsyncTask", TASK_PRIORITY, 0, TASK_STACK_SIZE,
    (FUNCPTR) syncTask, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)) == ERROR)
    {
        perror ("taskSpawn");
        cleanUp();
        return (ERROR);
    }

    /* watchdog simulates hardware interrupts */
    if (wdStart (wdId, TIME_BETWEEN_INTERRUPTS, (FUNCPTR) syncISR, numToGive)
        == ERROR)
    {
        perror ("wdStart");
        cleanUp ();
        return (ERROR);
    }
}
```



```

    }

    /* arbitrary delay to allow program to complete before clean up */
    taskDelay (sysClkRateGet() + ((TASK_WORK_TIME + 2) * numToGive));

    cleanUp();
    return (OK);
}

/*****
 * syncTask - synchronizes with interrupts using counting or binary
 *             semaphores.
 */

void syncTask (void)
{
    int eventCount = 0;

    FOREVER
    {
        if (semTake (semId, WAIT_FOREVER) == ERROR)
        {
            perror ("syncTask semTake");
            return;
        }

        /* Do "work" */
        taskDelay (TASK_WORK_TIME);
        semShow (semId, 1);

        eventCount++;
        printf ("semaphore taken %d times\n", eventCount);
    }
}

/*****
 * syncISR - simulates a hardware device which generates interrupts very
 *           quickly and synchronizes with syncTask using semaphores.
 */
void syncISR

```



```

(
int times
)
{
semGive (semId);
times--;
if (times > 0)
    wdStart (wdId, TIME_BETWEEN_INTERRUPTS, (FUNCPTR) syncISR, times);
}

/*****
* cleanUP - deletes the syncTask, deletes the semaphore and the watchdog timer
*           previously created by countingSemDemo.
*/
void cleanUp ()
{
    if (syncTaskTid)
        taskDelete (syncTaskTid);
    if (semId)
        semDelete (semId);
    if (wdId)
        wdDelete (wdId);
}
/*****/

```

编译例子代码、下载目标代码到目标机。

使用计数器信号量，在 VxWorks 目标机上执行命令：

```

-> sp countingSemDemo, 'c'
task spawned: id = 5b7b40, name = u0
value = 5995328 = 0x5b7b40

```

输出信息如下：

```

Semaphore Id       : 0x7fd238
Semaphore Type     : COUNTING
Task Queuing       : PRIORITY
Pended Tasks       : 0
Count              : 2

```

```

semaphore taken 1 times

```



```
Semaphore Id      : 0x7fd238
Semaphore Type    : COUNTING
Task Queuing      : PRIORITY
Pended Tasks      : 0
Count             : 1
```

semaphore taken 2 times

```
Semaphore Id      : 0x7fd238
Semaphore Type    : COUNTING
Task Queuing      : PRIORITY
Pended Tasks      : 0
Count             : 0
```

semaphore taken 3 times

使用二进制信号量，在 VxWorks 目标机上执行命令：

```
-> sp countingSemDemo, 'b'
task spawned: id = 5b7b40, name = u1
value = 5995328 = 0x5b7b40
```

输出信息如下：

```
Semaphore Id      : 0x7fd238
Semaphore Type    : BINARY
Task Queuing      : PRIORITY
Pended Tasks      : 0
State             : FULL
```

semaphore taken 1 times

```
Semaphore Id      : 0x7fd238
Semaphore Type    : BINARY
Task Queuing      : PRIORITY
Pended Tasks      : 0
State             : EMPTY
```

semaphore taken 2 times



5.5 消息队列

5.5.1 什么是消息队列

现代实时应用通常构造成由一套独立的相互合作的任务集合。虽然，信号量提供高速的任务间同步和互斥机制，但是常常还需要一种较高级的允许合作任务之间相互通信的机制。VxWorks 中，单 CPU 任务间主要的通信机制是消息队列。作为一个可选产品，VxMP 提供处理器间任务通信的全局消息队列。

消息队列允许长度可变、数目可变的消息排队。任何任务或 ISR 可以发送消息到消息队列。任何任务可从消息队列接收消息。多个任务可向同一个消息队列发送消息或接收消息。两个任务间全双工地通信一般需要两个消息队列，每个提供一个流通方向，如图 5.10 所示。

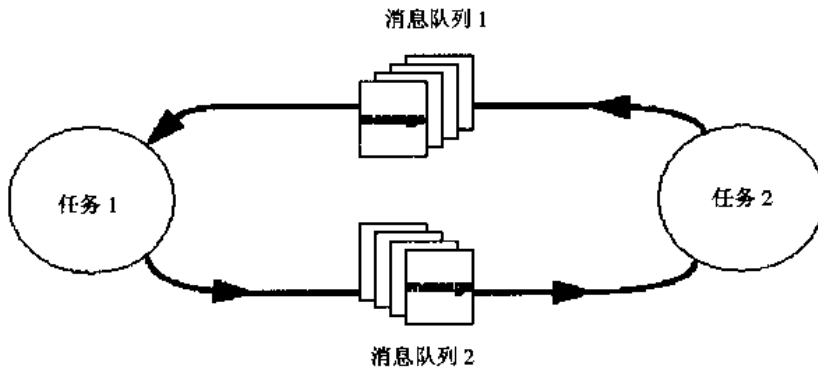


图 5.10 使用消息队列实现全双工通信

VxWorks 提供两个消息队列函数库，一个是 msgQLib，提供 Wind 消息队列，专门为 VxWorks 设计；另一个是 mqPxLib，提供与 POSIX 实时扩展标准（1003.1b）兼容。两种消息队列的异同稍后讨论。

5.5.2 Wind 消息队列

Wind 消息队列创建与删除函数如表 5.5 所示。这个库提供按 FIFO 排队的消息队列。但是有一个例外，Wind 消息队列有两个优先级，高优先级的消息将放在队列的头部。



表 5.5 Wind 消息队列控制函数

调 用	描 述
msgQCreate()	分配并初始化一个消息队列
msgQDelete()	终止并释放一个消息队列
msgQSend()	向一个消息队列发送消息
MsgQReceive()	从一个消息队列接收消息

消息队列由函数 `msgQCreate()` 创建，以它能够排队的最大的消息数目以及每个消息的最大字节长度作为参数。`msgQCreate()` 函数更具消息的数目和长度，预先分配足够的缓冲空间。

任务或 ISR 调用 `msgQSend()` 来向消息队列发送消息。此时如果没有任务在等待该队列中的消息，那么该消息进入消息队列的缓冲。如果有任务等待该队列的消息，那么这个消息立即提交给第一个等待的任务。任务调用 `msgQReceive()` 从消息队列接收消息。如果队列缓冲中已有可用的消息，那么第一个消息立即出队，并返回给调用者。如果没有消息可用，调用者将阻塞，进入等待该消息的任务队列排队。排队可以按两种顺序：任务优先级或 FIFO，这由队列创建时的参数决定。

➤ 超时

`msgQSend()` 和 `msgQReceive()` 两个函数都可有超时作为参数。当发送一个消息时，超时的含义是指当队列没有可用缓冲时，可以等待队列缓冲变为可用的 tick 长度。当接收一个消息时，超时的含义是指当队列没有消息立即可用时，可以等待队列消息变为可用的 tick 长度。和信号量一样，超时参数可以为 `NO_WAIT (=0)`，意味着立即返回，或 `WAIT_FOREVER (= -1)`，意味着程序永不超时。

➤ 紧急消息

函数 `msgQSend()` 使用一个参数来指定消息的优先级，正常 (`MSG_PRI_NORMAL`) 或紧急 (`MSG_PRI_URGENT`)。正常优先级消息追加到消息队列的尾部，紧急优先级任务添加到消息队列的首部。

下面给出一个例子，演示任务使用消息队列进行通信。演示程序创建一个消息队列，生产者任务和消费者任务使用该队列进行通信。它发起生产者任务，该任务向消息队列发送消息；发起消费者任务，从消息队列读取消息。一旦消费者任务“消费”了所有消息，该消息队列将被删除。



例 5.8 任务使用消息队列进行通信

```
/******  
/* msgQDemo.h - Header for the msgQDemo */  
  
/* Copyright 1984-1997 Wind River Systems, Inc. */  
  
/*  
modification history  
-----  
01b, 06nov97, mm added copyright.  
01a, 14dec93, ms written.  
*/  
  
#define CONSUMER_TASK_PRI    99 /* Priority of the consumer task */  
#define PRODUCER_TASK_PRI    98 /* Priority of the producer task */  
#define TASK_STACK_SIZE     5000 /* stack size for spawned tasks */  

```



```
#include "vxWorks.h"
#include "taskLib.h"
#include "msgQLib.h"
#include "msgQDemo.h"
#include "sysLib.h"
#include "stdio.h"

/* function prototypes */

LOCAL STATUS producerTask ();      /* producer task */
LOCAL STATUS consumerTask ();      /* consumer task */

/*****
 * msgQDemo - Demonstrates intertask communication using Message Queues
 *
 * DESCRIPTION
 * Creates a Message Queue for interTask communication between the
 * producerTask and the consumerTask. Spawns the producerTask that creates
 * messages and sends messages to the consumerTask using the message queue.
 * Spawns the consumerTask that reads messages from the message queue.
 * After consumerTask has consumed all the messages, the message queue is
 * deleted.
 *
 * RETURNS: OK or ERROR
 *
 * EXAMPLE
 *
 * -> sp msgQDemo
 */

STATUS msgQDemo()
{
    notDone = TRUE; /* initialize the global flag */

    /* Create the message queue*/
    if ((msgQId = msgQCreate (numMsg, sizeof (struct msg), MSG_Q_FIFO))
        == NULL)
    {
        perror ("Error in creating msgQ");
        return (ERROR);
    }
}
```



```
/* Spwan the producerTask task */
if (taskSpawn ("tProducerTask", PRODUCER_TASK_PRI, 0, TASK_STACK_SIZE,
              (FUNCPTR) producerTask, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
    == ERROR)
{
    perror ("producerTask: Error in spawning demoTask");
    return (ERROR);
}

/* Spwan the consumerTask task */
if (taskSpawn ("tConsumerTask", CONSUMER_TASK_PRI, 0, TASK_STACK_SIZE,
              (FUNCPTR) consumerTask, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
    == ERROR)
{
    perror ("consumerTask: Error in spawning demoTask");
    return (ERROR);
}

/* polling is not recommended. But used to make this demonstration simple*/
while (notDone)
    taskDelay (sysClkRateGet ());

if (msgQDelete (msgQId) == ERROR)
{
    perror ("Error in deleting msgQ");
    return (ERROR);
}

return (OK);
}

/*****
* producerTask - produces messages, and sends messages to the consumerTask
*                using the message queue.
*
* RETURNS: OK or ERROR
*
*/

STATUS producerTask (void)
{
    int count;
```



```

int value;
struct msg producedItem;      /* producer item - produced data */

printf ("producerTask started: task id = %#x \n", taskIdSelf ());

/* Produce numMsg number of messages and send these messages */

for (count = 1; count <= numMsg; count++)
{
    value = count * 10; /* produce a value */

    /* Fill in the data structure for message passing */
    producedItem.tid = taskIdSelf ();
    producedItem.value = value;

    /* Send Messages */
    if ((msgQSend (msgQId, (char *) &producedItem, sizeof (producedItem),
        WAIT_FOREVER, MSG_PRI_NORMAL)) == ERROR)
    {
        perror ("Error in sending the message");
        return (ERROR);
    }
    else
        printf ("ProducerTask: tid = %#x, produced value = %d \n",
            taskIdSelf (), value);
}

return (OK);
}

/*****
 * consumerTask - consumes all the messages from the message queue.
 *
 * RETURNS: OK or ERROR
 *
 */

STATUS consumerTask (void)
{
    int count;
    struct msg consumedItem;    /* consumer item - consumed data */

    printf ("\n\nConsumerTask: Started - task id = %#x\n", taskIdSelf());

```



```
/* consume numMsg number of messages */
for (count = 1; count <= numMsg; count++)
{
    /* Receive messages */
    if ((msgQReceive (msgQId, (char *) &consumedItem,
                    sizeof (consumedItem), WAIT_FOREVER)) == ERROR)
    {
        perror ("Error in receiving the message");
        return (ERROR);
    }
    else
        printf ("ConsumerTask: Consuming msg of value %d from tid = %#x\n",
                consumedItem.value, consumedItem.tid);
}

notDone = FALSE; /* set the global flag to FALSE to indicate completion*/
return (OK);
}

/*****/
```

在 VxWorks 目标机上执行命令:

```
-> ld sp msgQDemo
task spawned: id = 5ad730, name = u0
value = 5953328 = 0x5ad730
```

输出信息如下:

```
producerTask started: task id = 0x7fcea8
ProducerTask: tid = 0x7fcea8, produced value = 10
ProducerTask: tid = 0x7fcea8, produced value = 20
ProducerTask: tid = 0x7fcea8, produced value = 30
ProducerTask: tid = 0x7fcea8, produced value = 40
ProducerTask: tid = 0x7fcea8, produced value = 50
ProducerTask: tid = 0x7fcea8, produced value = 60
ProducerTask: tid = 0x7fcea8, produced value = 70
ProducerTask: tid = 0x7fcea8, produced value = 80

ConsumerTask: Started - task id = 0x7fcea8
ConsumerTask: Consuming msg of value 10 from tid = 0x7fcea8
ConsumerTask: Consuming msg of value 20 from tid = 0x7fcea8
```



```

ConsumerTask: Consuming msg of value 30 from tid = 0x7fcea8
ConsumerTask: Consuming msg of value 40 from tid = 0x7fcea8
ConsumerTask: Consuming msg of value 50 from tid = 0x7fcea8
ConsumerTask: Consuming msg of value 60 from tid = 0x7fcea8
ConsumerTask: Consuming msg of value 70 from tid = 0x7fcea8
ConsumerTask: Consuming msg of value 80 from tid = 0x7fcea8

```

5.5.3 POSIX 消息队列

■ POSIX 消息队列函数

POSIX 消息队列由 mqPxlLib 库提供，函数库如表 5.6。

表 5.6 POSIX 消息队列函数

调 用	描 述
mqPxlLibInit()	初始化 POSIX 消息队列库 (non-POSIX)
mq_open()	打开一个消息队列
mq_close()	关闭一个消息队列
mq_unlink()	移出一个消息队列
mq_send()	向一个消息队列发送消息
mq_receive()	从一个消息队列接收消息
mq_notify()	通知一个任务，有个消息在消息队列上等待
mq_setattr()	设置消息队列属性
mq_getattr()	得到消息队列属性

除了 POSIX 消息队列提供命名队列和消息具有一定范围的优先级之外，这些函数类似于 Wind 消息队列。

初始化函数 mqPxlLibInit()，使得 POSIX 消息队列函数可用，在使用其他消息队列函数之前，系统初始化代码必须调用这个函数。如果在配置 VxWorks 时包含了 INCLUDE_POSIX_MQ 定义，usrInit() 将自动调用 mqPxlLibInit()。在任务集合使用一个消息队列通信之前，其中之一必须以 O_CREAT 作为参数调用 mq_open() 创建这个消息队列。一旦消息队列已经创建，其他的任务可以根据其名字打开这个消息队列，然后可以收发消息。仅仅需要第一个打开这个队列的任务使用 O_CREAT 参数，后续的发送任务使用 O_WRONLY 参数打开该队列，接收任务使用 O_RDONLY 作为参数打开该队列，既收又发的任务使用 O_RDWR 作为打开函数的参数。



使用函数 `mq_send()`，可以将消息发送到消息队列。如果消息队列已满，这时，如果一个任务试图向该队列发送消息，这个任务将被阻塞，直到另外的任务从这个队列中读走一个消息，使得空间可用为止。为避免 `mq_send()` 阻塞，可以使用 `O_NONBLOCK` 作为参数打开消息队列。这样，如果队列满，`mq_send()` 将返回-1，并且设置 `errno` 为 `EAGAIN`，而任务不会阻塞，允许程序再试一次或采取其他合适的操作。

`mq_send()` 的一个参数用来指定优先级，优先级范围从 0（最低）到 31（最高）。

当任务调用 `mq_receive()` 从消息队列接收一个消息时，任务将接收到队列中当前优先级最高的消息。当多个消息有相同的优先级时，第一个放到队列中的消息先被取走（按照 FIFO 顺序）。

如果队列为空，任务将阻塞，直到有消息放入到队列。为避免任务阻塞在 `mq_receive()` 调用上，可以在打开队列时使用参数 `O_NONBLOCK`。这样，当任务试图从空队列取消息时，`mq_receive()` 返回-1，并且设置 `errno` 为 `EAGAIN`。

为关闭一个消息队列，可以调用函数 `mq_close()`。关闭一个队列不会摧毁它，仅仅保证调用任务不再使用它。为摧毁一个队列要调用 `mq_unlink()`。这个函数也不会立即摧毁队列，但是它阻止以后的任务打开该队列，并将该队列的名字从名字表中移出。那些当前已经打开该队列的任务仍然可以继续使用它。当最后关闭了这个队列，队列也就摧毁了。

例 5.9 任务使用 POSIX 消息队列进行通信

```
/* ***** */
/* In this example, the mqExInit() routine spawns two tasks that
 * communicate using the message queue.
 */
/* mqEx.h - message example header */
/* defines */
#define MQ_NAME "exampleMessageQueue"
/* forward declarations */
void receiveTask (void);
void sendTask (void);
/* testMQ.c - example using POSIX message queues */

/* includes */
#include "vxWorks.h"
#include "mqueue.h"
#include "fcntl.h"
#include "errno.h"
#include "mqEx.h"
/* defines */
#define HI_PRIO 31
#define MSG_SIZE 16.
```



```
int mqExInit (void)
{
    /* create two tasks */
    if (taskSpawn ("tRcvTask", 95, 0, 4000, receiveTask, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0) == ERROR)
    {
        printf ("taskSpawn of tRcvTask failed\n");
        return (ERROR);
    }
    if (taskSpawn ("tSndTask", 100, 0, 4000, sendTask, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0) == ERROR)
    {
        printf ("taskSpawn of tSendTask failed\n");
        return (ERROR);
    }
}

void receiveTask (void)
{
    mqd_t mqPId; /* msg queue descriptor */
    char msg[MSG_SIZE]; /* msg buffer */
    int prio; /* priority of message */
    /* open message queue using default attributes */
    if ((mqPId = mq_open (MQ_NAME, O_RDWR | O_CREAT, 0, NULL))
        == (mqd_t) -1)
    {
        printf ("receiveTask: mq_open failed\n");
        return;
    }
    /* try reading from queue */
    if (mq_receive (mqPId, msg, MSG_SIZE, &prio) == -1)
    {
        printf ("receiveTask: mq_receive failed\n");
        return;
    }
    else
    {
        printf ("receiveTask: Msg of priority %d received:\n\t\t%s\n",
            prio, msg);
    }
}
```



```
/* sendTask.c - mq sending example */
/* includes */
#include "vxWorks.h"
#include "mqqueue.h"
#include "fcntl.h"
#include "mqEx.h"
/* defines */
#define MSG "greetings"
#define HI_PRIO 30

void sendTask (void)
{
    mqd_t mqPXId; /* msg queue descriptor */
    /* open msg queue; should already exist with default attributes */
    if ((mqPXId = mq_open (MQ_NAME, O_RDWR, 0, NULL)) == (mqd_t) -1)
    {
        printf ("sendTask: mq_open failed\n");
        return;
    }
    /* try writing to queue */
    if (mq_send (mqPXId, MSG, sizeof (MSG), HI_PRIO) == -1)
    {
        printf ("sendTask: mq_send failed\n");
        return;
    }
    else
        printf ("sendTask: mq_send succeeded\n");
}
/*****/
```

■ 通知任务一个消息队列在等待

任务可以调用 `mq_notify()` 函数要求系统当有一个消息进入一个空的消息队列时通知它。这样可以避免任务阻塞或轮询等待一个消息。

函数 `mq_notify()` 以消息进入空队列时系统发给任务的信号 (signal) 作为参数。这种机制将 POSIX 数据传输 (data-carrying) 功能扩展到信号, 允许我们使用信号捎带队列的标示符 (identifier)。

`mq_notify()` 机制只对当前状态为空的消息队列有效, 当该队列有新消息可用时提醒任务。如果消息队列已有可用消息, 当有更多消息到达时, 通知不会发生。如果有其他任务因调用 `mq_receive()` 阻塞在该队列上, 也不会通知这个使用 `mq_notify()` 注册的任务。



也就是说,通知仅仅通知一个任务。每个队列任一时刻只能有一个注册等待通知的任务,一旦队列已经有一个任务等待通知,不会接受其他任务调用 `mq_notify()` 的注册,直到这个通知请求已经满足或者取消。

当队列向一个任务发送通知时,通知请求也就得到满足,队列和这个特定任务也就不存在这种特定的关系。也就是说,对每个 `mq_notify()` 请求,队列仅仅发送一次通知。如果这个任务需要继续接收通知信号,最好的方法是在接收这个通知的信号处理程序中调用 `mq_notify()`,这将重新安装通知请求。

用 `NULL` 代替通知信号调用 `mq_notify()` 函数,删除一个通知请求。注意,仅仅当前注册的任务才能删除它的通知信号。

例 5.10 通知任务一个消息队列在等待

```

/*****
/* In this example, a task uses mq_notify() to discover when a message
* is waiting for it on a previously empty queue.
*/

/* includes */
#include "vxWorks.h"
#include "signal.h"
#include "mqueue.h"
#include "fcntl.h"
#include "errno.h"
/* defines */
#define QNAM "PxQ1"
#define MSG_SIZE 64 /* limit on message sizes */
/* forward declarations */
static void exNotificationHandle (int, siginfo_t *, void *);
static void exMqRead (mqd_t);

/*****
* exMqNotify - example of how to use mq_notify()
*
* This routine illustrates the use of mq_notify() to request notification
* via signal of new messages in a queue. To simplify the example, a
* single task both sends and receives a message.
*/

int exMqNotify
(
    char * pMess /* text for message to self */

```



```
)
{

    struct mq_attr attr; /* queue attribute structure */
    struct sigevent sigNotify; /* to attach notification */
    struct sigaction mySigAction; /* to attach signal handler */
    mqd_t exMqId; /* id of message queue */
    /* Minor sanity check; avoid exceeding msg buffer */
    if (MSG_SIZE <= strlen (pMess))
    {
        printf ("exMqNotify: message too long\n");
        return (-1);
    }
    /* Install signal handler for the notify signal - fill in a
    * sigaction structure and pass it to sigaction(). Because the
    * handler needs the siginfo structure as an argument, the
    * SA_SIGINFO flag is set in sa_flags.
    */
    mySigAction.sa_sigaction = exNotificationHandle;
    mySigAction.sa_flags = SA_SIGINFO;
    sigemptyset (&mySigAction.sa_mask);
    if (sigaction (SIGUSR1, &mySigAction, NULL) == -1)
    {
        printf ("sigaction failed\n");
        return (-1);
    }
    /* Create a message queue - fill in a mq_attr structure with the
    * size and no. of messages required, and pass it to mq_open().
    */
    attr.mq_flags = O_NONBLOCK; /* make nonblocking */
    attr.mq_maxmsg = 2;
    attr.mq_msgsize = MSG_SIZE;
    if ( (exMqId = mq_open (QNAM, O_CREAT | O_RDWR, 0, &attr)) ==
        (mqd_t) - 1 )
    {
        printf ("mq_open failed\n");
        return (-1);
    }
    /* Set up notification: fill in a sigevent structure and pass it
    * to mq_notify(). The queue ID is passed as an argument to the
    * signal handler.
    */
    sigNotify.sigev_signo = SIGUSR1;
    sigNotify.sigev_notify = SIGEV_SIGNAL;
```



```
sigNotify.sigev_value.sival_int = (int) exMqId;
if (mq_notify (exMqId, &sigNotify) == -1)
{
    printf ("mq_notify failed\n");
    return (-1);
}
/* We just created the message queue, but it may not be empty;
* a higher-priority task may have placed a message there while
* we were requesting notification. mq_notify() does nothing if
* messages are already in the queue; therefore we try to
* retrieve any messages already in the queue.
*/
exMqRead (exMqId);
/* Now we know the queue is empty, so we will receive a signal
* the next time a message arrives.
*
* We send a message, which causes the notify handler to be
* invoked. It is a little silly to have the task that gets the
* notification be the one that puts the messages on the queue,
* but we do it here to simplify the example.
*
* A real application would do other work instead at this point.
*/
if (mq_send (exMqId, pMess, 1 + strlen (pMess), 0) == -1)
{
    printf ("mq_send failed\n");
    return (-1);
}
/* Cleanup */
if (mq_close (exMqId) == -1)
{
    printf ("mq_close failed\n");
    return (-1);
}
/* More cleanup */
if (mq_unlink (QNAM) == -1)
{
    printf ("mq_unlink failed\n");
    return (-1);
}
return (0);
}
```



```

/*****
* exNotificationHandle - handler to read in messages
*
* This routine is a signal handler; it reads in messages from a message
* queue.
*/
static void exNotificationHandle
(
    int sig, /* signal number */
    siginfo_t * pInfo, /* signal information */
    void * pSigContext /* unused (required by posix) */
)
{
    struct sigevent sigNotify;
    mqd_t exMqId;
    /* Get the ID of the message queue out of the siginfo structure. */
    exMqId = (mqd_t) pInfo->si_value.sival_int;
    /* Request notification again; it resets each time a notification
    * signal goes out.
    */
    sigNotify.sigev_signo = pInfo->si_signo;
    sigNotify.sigev_value = pInfo->si_value;
    sigNotify.sigev_notify = SIGEV_SIGNAL;
    if (mq_notify (exMqId, &sigNotify) == -1)
        {
            printf ("mq_notify failed\n");
            return;
        }

    /* Read in the messages */
    exMqRead (exMqId);
}

/*****
* exMqRead - read in messages
*
* This small utility routine receives and displays all messages
* currently in a POSIX message queue; assumes queue has O_NONBLOCK.
*/
static void exMqRead
(
    mqd_t exMqId
)

```



```

{
    char msg[MSG_SIZE];
    int prio;
    /* Read in the messages - uses a loop to read in the messages
    * because a notification is sent ONLY when a message is sent on
    * an EMPTY message queue. There could be multiple msgs if, for
    * example, a higher-priority task was sending them. Because the
    * message queue was opened with the O_NONBLOCK flag, eventually
    * this loop exits with errno set to EAGAIN (meaning we did an
    * mq_receive() on an empty message queue).
    */
    while (mq_receive (exMqId, msg, MSG_SIZE, &prio) != -1)
    {
        printf ("exMqRead: received message: %s\n", msg);
    }
    if (errno != EAGAIN)
    {
        printf ("mq_receive: errno = %d\n", errno);
    }
}
/*****/

```

消息队列属性

一个 POSIX 消息队列有以下属性：

- ◆ 一个可选的 O_NONBLOCK 标志。
- ◆ 消息队列允许的最大的消息数目。
- ◆ 每个消息的最大长度。
- ◆ 队列中当前消息的数目。

任务可以调用 mq_setattr() 和 mq_getattr() 函数分别来设置或获得队列的属性。

例 5.11 设置或获得队列的属性

```

/*****/
/* This example sets the O_NONBLOCK flag, and examines message queue
* attributes.
*/
/* includes */
#include "vxWorks.h"
#include "mqueue.h"
#include "fcntl.h"

```

```

#include "errno.h"
/* defines */
#define MSG_SIZE 16

int attrEx
(
    char * name
)
{
    mqd_t mqPId; /* mq descriptor */
    struct mq_attr attr; /* queue attribute structure */
    struct mq_attr oldAttr; /* old queue attributes */
    char buffer[MSG_SIZE];
    int prio;
    /* create read write queue that is blocking */
    attr.mq_flags = 0;
    attr.mq_maxmsg = 1;
    attr.mq_msgsize = 16;
    if ((mqPId = mq_open (name, O_CREAT | O_RDWR, 0, &attr))
        == (mqd_t) -1)
        return (ERROR);
    else
        printf ("mq_open with non-block succeeded\n");
    /* change attributes on queue - turn on non-blocking */
    attr.mq_flags = O_NONBLOCK;
    if (mq_setattr (mqPId, &attr, &oldAttr) == -1)
        return (ERROR);
    else
    {
        /* paranoia check - oldAttr should not include non-blocking. */
        if (oldAttr.mq_flags & O_NONBLOCK)
            return (ERROR);
        else
            printf ("mq_setattr turning on non-blocking succeeded\n");
    }
    /* try receiving - there are no messages but this shouldn't block */
    if (mq_receive (mqPId, buffer, MSG_SIZE, &prio) == -1)
    {
        if (errno != EAGAIN)
            return (ERROR);
        else
            printf ("mq_receive with non-blocking didn't block on empty
queue\n");
    }
    else
        return (ERROR);
}

```



```

/* use mq_getattr to verify success */
if (mq_getattr (mqPId, &oldAttr) == -1)
    return (ERROR);
else
{
    /* test that we got the values we think we should */
    if (!(oldAttr.mq_flags & O_NONBLOCK) || (oldAttr.mq_curmsgs != 0))
        return (ERROR);
    else
        printf ("queue attributes are:\n\tblocking is %s\n\t
            message size is: %d\n\t
            max messages in queue: %d\n\t
            no. of current msgs in queue: %d\n",
            oldAttr.mq_flags & O_NONBLOCK ? "on" : "off",
            oldAttr.mq_msgsize, oldAttr.mq_maxmsg,
            oldAttr.mq_curmsgs);
}

/* clean up - close and unlink mq */
if (mq_unlink (name) == -1)
    return (ERROR);
if (mq_close (mqPId) == -1)
    return (ERROR);
return (OK);
}
/*****/

```

5.5.4 POSIX 和 Wind 消息队列比较

POSIX 和 Wind 两种风格消息队列解决了同样的问题，但是也有较大的差别，总结在表 5.7 中。

表 5.7 POSIX 和 Wind 消息队列比较

特 征	Wind 消息	POSIX 消息队列
消息优先级	1	32
阻塞的任务队列	FIFO 或基于优先级	基于优先级
不带超时的接收	可选	不可用
任务通知	Not available	可选 (一个任务)
关闭/解锁语法	无	有



当然，POSIX 消息队列的另一个特征是可移植性：如果用户将已有的一个 1003.1b 兼容系统移植到 VxWorks，最好使用 POSIX 消息队列，这样可以减少代码的改动量，降低移植的费用。

5.5.5 显示消息队列的属性

VxWorks 提供的 show()命令可以用来显示消息队列的重要属性。例如，如果 mqPXiD 是一个 POSIX 消息队列，可以使用下述命令察看 mqPXiD 的属性：

```
-> show mqPXiD
value = 0 = 0x0
```

输出将发送到标准输入设备，将显示像下面类似的内容：

```
Message queue name : MyQueue
No. of messages in queue : 1
Maximum no. of messages : 16
Maximum message size : 16
```

可以与当 myMsgQId 是一个 Wind 消息队列时的输出相比较：

```
-> show myMsgQId
Message Queue Id : 0x3adaf0
Task Queuing : FIFO
Message Byte Len : 4
Messages Max : 30
Messages Queued : 14
Receivers Blocked : 0
Send timeouts : 0
Receive timeouts : 0
```

5.5.6 使用消息队列的服务器和客户

实时系统经常构造成客户/服务器任务模型。这种模型中，服务器任务接收来自客户任务的请求，执行一些服务，然后返回一个应答。这些请求和应答经常形成任务间的通信消息。VxWorks 中，消息队列或管道是实现这种模型的自然方式。

例如，客户-服务器通信如图 5.11 所示的方式实现。

每个服务任务创建一个消息队列，接收来自客户的请求。每个客户创建一个消息队列来



接收服务任务的应答。每个请求消息有一个域包含客户应答信息队列的 msgQId 。服务器任务轮询它的请求消息队列，读取请求消息，执行请求，将应答发送到该客户指定的应答消息队列。

使用管道也可以实现相同的功能。

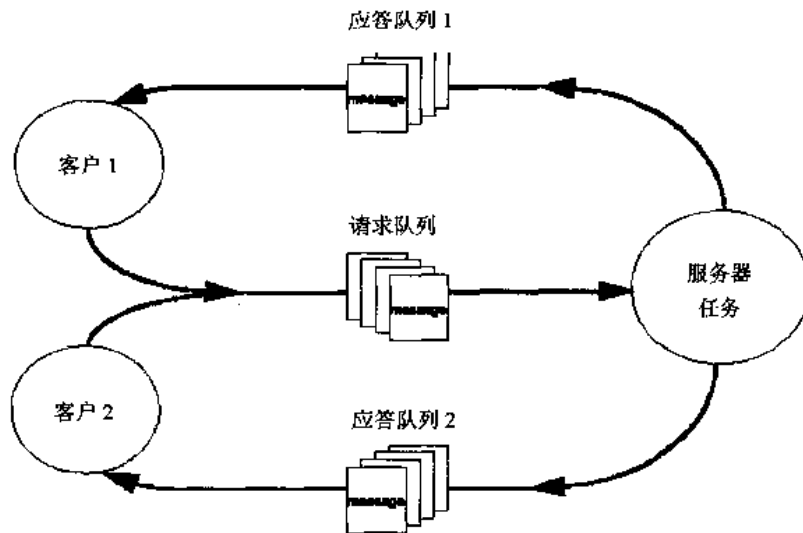


图 5.11 使用消息队列实现客户——服务器通信

5.6 管道

5.6.1 管道

管道使用 VxWorks I/O 系统，可提供能与消息队列互换的功能。管道是一种由 pipeDrv 驱动程序管理的虚拟 I/O 设备。函数 pipeDevCreate() 创建一个管道设备以及与该管道相联系的底层消息队列。调用指定了要创建管道的名字、能够排队的消息的最大数目、每个消息的最大长度等信息。

```
status = pipeDevCreate ("/pipe/name", max_msgs, max_length);
```

创建的管道正常是命名的 I/O 设备。任务能够使用标准 I/O 程序来打开、读、写管道，也可以调用 ioctl 函数设置控制属性。像其他的 I/O 驱动那样，当从一个没有数据可用的空的管道读数据时，任务会堵塞。像对消息队列的操作一样。ISRs 能够向管道写，但是不能从管道读，像 I/O 设备。管道能提供消息队列不能提供的一个重要特征，可以使用 select()



函数。它允许任务等待一个 I/O 设备集合之一的数据可用。select() 函数也能够与其他的异步 I/O 设备一起工作，包括网络套接字和串行设备。因此使用 select()，任务能够同时等待几个管道、套接字和串行设备集合上的数据。

管道也能实现任务间的客户/服务器通信的模式。

VxWorks 应用中的管道是一个先进/先出的缓冲，类似于 UNIX 的命名管道。这种虚拟设备类似于消息队列，却比消息队列使用简单。

一个任务可以创建一个管道，一旦管道创建，任务可以直接调用 read()和 write()语句对之进行读写访问。如果任务试图向一个已满的管道执行写操作，该任务将等待（挂起）。如果任务试图向一个空的管道执行读操作，任务也将等待直到有消息到达。在大多数情形下，我们可以使用管道来代替消息队列。

管道强调文件描述符，因而它提供消息队列所缺乏的这种能力，可以提供 select 和其他基本 I/O 操作。图 5.12 示意了一个任务等待来自于五个文件对象（具备文件描述符特征）的一个信息。

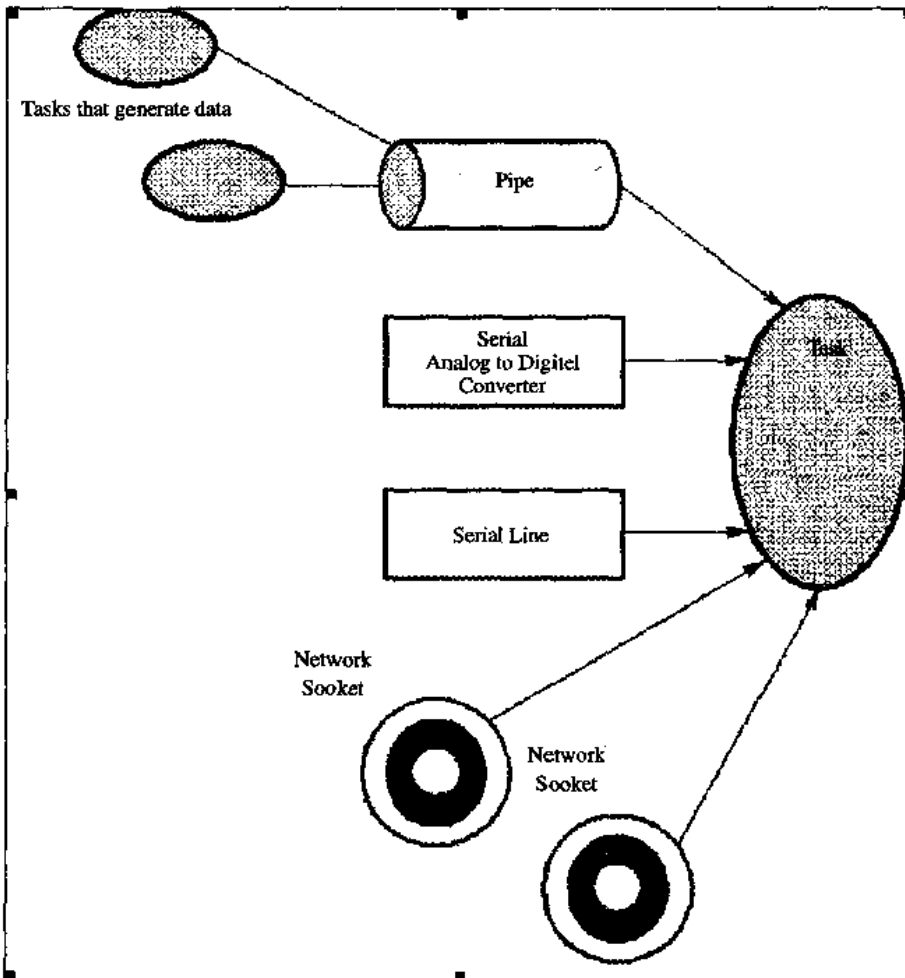


图 5.12 一个等待来自于五个 select 对象（包括一个管道）的任务



网络任务间通信方式将在第八章介绍。

5.6.2 管道用于任务间通信

下面给出一个例子，演示任务间使用管道进行通信。

演示程序中，函数 `ServerStart()` 初始化一个服务器任务，它以一个较低的优先级运行，使用管道作为通信机制。函数 `serverSend()` 向服务器任务发送请求，请求使用管道设备，以服务器任务的优先级执行一个功能。

例 5.12 管道用于任务间通信

```

/*****
/* pipeServer.c - Demonstrates intertask communication using pipes */

/* Copyright 1984-1997 Wind River Systems, Inc. */.

/*
modification history
-----
01c, 06nov97, mm added copyright.
01b, 19sep97, ram need to add INCLUDE_PIPE in configAll.h
        added include files stdio.h, ioLib.h, pipeDrv.h
01a, 11mar94, ms cleaned up for VxDemo
*/

/* DESCRIPTION
*
* Demonstrates intertask communication using pipes.
*
* serverStart initializes a server task to execute functions
* at a low priority and uses pipes as the communication
* mechanism.
*
* serverSend sends a request to the server to execute a
* function at the server's priority using pipe device.
* Usage : serverSend (<function>, <argument of the function>)
*
* EXAMPLE
* To run this demo from the Wind shell do as follows:

```



```
*
* -> serverStart
* value = 0 = 0x0
* -> serverSend (printf, "Hello World. \n")
* value = 0 = 0x0
*
* NOTE: A host function such as i() cannot be used as the function
*       argument to serverSend.
*       eg: -> serverSend(i, "tShell") will not work
*       However if you wish to do that then make sure to add
*       INCLUDE_CONFIGURATION_5_2 and INCLUDE_DEBUG in configAll.h
*/

/* includes */

#include "vxWorks.h"
#include "taskLib.h"
#include "stdio.h"
#include "ioLib.h"
#include "pipeDrv.h"

typedef struct
{
    VOIDFUNCPTR routine;
    int arg;
} MSG_REQUEST;          /* message structure */

#define TASK_PRI        254          /* tServers task's priority */
#define TASK_STACK_SIZE 5000        /* tServer task's stack size */
#define PIPE_NAME       "/pipe/server" /* name of the pipe device */
#define NUM_MSGS        10          /* max number of messages in the pipe */

LOCAL int pipeFd;          /* File descriptor for the pipe device */
LOCAL void pipeServer ();  /* server task */

/*****
* serverStart -- Initializes a server task to execute functions
*                at a low priority. Uses pipes as the communication
*                mechanism.
*
*****/
```



```

* RETURNS: OK or ERROR on failure.
*/

STATUS serverStart ()
{

if (pipeDevCreate (PIPE_NAME, NUM_MSGS, sizeof (MSG_REQUEST)) == ERROR)
{
    perror ("Error in creating pipe"); /* print error if pipe is already
                                        * created, but do not return */
}

/* Open the pipe */
if ((pipeFd = open (PIPE_NAME, UPDATE, 0)) == ERROR)
{
    perror ("Error in opening pipe device");
    return (ERROR);
}

/* Spawn the server task */
if (taskSpawn ("tServer", TASK_PRI, 0, TASK_STACK_SIZE,
              (FUNCPTR) pipeServer, 0, 0, 0, 0, 0, 0, 0, 0, 0)
    == ERROR)
{
    perror ("Error in spawning tServer task");
    close (pipeFd);
    return (ERROR);
}

return (OK);
}

/*****
* serverSend -- Sends a request to the server to execute a
*               function at the server's priority.
*
* RETURNS: OK or ERROR on failure.
*/

STATUS serverSend
(
    VOIDFUNCPTR routine,          /* name of the routine to execute */
    int arg                      /* argument of the routine */
)

```



```
)
{
MSG_REQUEST msgRequest;
int status;

/* Initialize the message structure */
msgRequest.routine = routine;
msgRequest.arg = arg;

/* Send the message and return the results */
status = write (pipeFd, (char *)&msgRequest, sizeof (MSG_REQUEST));

return ((status == sizeof (MSG_REQUEST)) ? OK : ERROR);
}

/*****
* pipeServer -- Server task which reads from a pipe and executes
* the function passed in the MSG_REQUEST data structure.
*
*/

LOCAL void pipeServer ()
{
MSG_REQUEST msgRequest;

while (read (pipeFd, (char *)&msgRequest, sizeof (MSG_REQUEST)) > 0)
(*msgRequest.routine) (msgRequest.arg);
}
/*****/
```

在 VxWorks 目标机上执行命令:

```
-> ld serverStart
value = 0 = 0x0
-> serverSend (printf, "Hello World. \n")
value = 0 = 0x0
```

输出信息如下:

```
Hello World.
```

中断通常是外部事件通知实时系统的主要机制，硬件中断处理也就成为影响实时系统性能的另一个关键因素。为获得尽可能的、最快的中断反应时间，VxWorks 的中断处理程序（ISRs）运行在他们特定的上下文中（独立于任何任务的上下文）。

6.1 信号（Signals）

VxWorks 支持软件信号功能。信号可以异步改变任务控制流。任何任务和 ISR 都可向指定任务发信号。获得信号的任务立即挂起当前的执行，在下次调度它运行时转而执行指定的信号处理程序。信号处理程序在信号接收任务的上下文中执行，使用该任务的堆栈。在任务阻塞时，信号处理程序仍可被唤醒。

与用于任务间通信相比，信号机制适合于错误和异常处理。通常，信号处理程序可以作为中断处理程序看待。任何可能导致调用程序阻塞的函数均不能在信号处理程序中调用。由于信号是异步的，很难预测当信号处理程序执行时，哪种资源是可用的。为了系统安全起见，信号处理程序仅能调用那些能在中断处理程序中安全使用的函数（见表 6.4），否则可能导致系统出现死锁。

wind 内核支持两种类型的信号接口：

- UNIX BSD 风格的信号。
- POSIX 兼容的信号。

POSIX 兼容的信号接口既包含了 POSIX 标准 1003.1 中定义的基本信号接口，也包含了 POSIX 1003.1b 扩展的队列信号接口。出于简单的考虑，编程时最好只使用其中一种接口，不要混合使用不同的接口形式，否则需要特别细心。



6.1.1 基本信号接口

SigLib 中定义了基本信号接口，如表 6.1。

表 6.1 基本信号函数调用 (BSD 和 POSIX 1003.1b)

POSIX 1003.1b 兼容调用	UNIX/BSD 兼容调用	说 明
signal()	signal()	指定信号的处理程序
kill()	Kill()	向任务发送一个信号
raise()	N/A	发信号给自身
sigaction()	sigvec()	检查或设置信号处理程序
sigsuspend()	Pause()	挂起任务直到信号提交
sigpending()	N/A	取回一组提交阻塞的信号
sigemptyset()		
sigfillset()		
sigaddset()	sigsetmask()	设置信号屏蔽
sigdelset()		
sigismember()		
sigprocmask()	sigsetmask()	设置阻塞信号的屏蔽
sigblock()	sigblock()	增加到一组阻塞的信号

调用这些函数之前，必须调用函数 sigInit() 初始化信号函数库。正常情况下，在中断允许之前，在程序 usrConfig.c 的 usrInit() 函数中调用。

信号在很多方面相似于硬件中断。基本信号接口提供 31 个不同的信号。调用 sigvec() 或 sigaction() 可为信号指定一个信号处理程序。这与调用 intConnect() 为中断指定一个中断处理程序(ISR)相似。可以调用 kill() 将信号发送给任务，这类似于中断发生。函数 sigsetmask() 和 sigblock() 或 sigprocmask() 可以用来像屏蔽中断那样屏蔽信号。

函数 sigvec() 和 kill() 的原型如下：

```
int sigvec
(
    int          sig,          /* 与处理程序相联系的信号 */
    const struct sigvec * pVec, /* 新的处理程序信息 */
    struct sigvec *   pOVec   /* 旧的处理程序信息 */
)
```



```
int kill
(
    int tid,          /* 接收信号的任务号 */
    int signo        /* 发送给任务的信号 */
)
```

信号的发生一般与硬件异常相联系，例如总线出错、非法指令和浮点数异常都可产生特定的异常。

6.1.2 POSIX 队列信号

`sigqueue()` 函数提供与 `kill()` 等价的功能：向任务发送信号。二者不尽相同。

`sigqueue()` 包括一个特定应用的值，作为信号的一部分发送。这个值可以提供给信号处理程序使用，类型是 `sigval`，定义在 `signal.h` 中。该值存放在信号处理程序的结构参数 `siginfo_t` 的 `si_value` 域中。POSIX `sigaction()` 函数扩展允许信号处理函数接收这个附加的参数。

`sigqueue()` 排队发送给任何任务的多个信号。相反，`kill()` 仅仅提交一个信号，尽管在信号处理程序运行之前，可能有多个信号已经到达。

`sigqueue()` 的函数原型如下：

```
int sigqueue
(
    int          tid,          /* 接收信号的任务号 */
    int          signo,       /* 发送给任务的信号 */
    const union sigval value /* 随信号发送的值 */
)
```

VxWorks 提供七个保留的信号留给应用程序使用。编号从 `SIGRTMIN` 连续递增（例如，第三个保留信号的编号为 `SIGRTMIN+2`）。这些保留是 POSIX 1003.1b 标准需要的，但是信号值却不是标准规定的值。

所有由 `sigqueue()` 提交的信号按照编号顺序排队，编号低的信号在编号高的前面。

POSIX 1003.1b 也引进了接收信号的手段。函数 `sigwaitinfo()` 不同于 `sigsuspend()` 和 `pause()`，它允许使用响应信号而不需要注册信号处理程序的机制。当信号可用时，`sigwaitinfo()` 返回信号的值，而不会调用信号处理程序，尽管可能已经注册了信号处理程序。

`sigwaitinfo()` 的函数原型如下：

```
int sigwaitinfo
(
```



```

const sigset_t * pSet, /*阻塞时的信号掩码*/
struct siginfo * pInfo /*返回值*/
)

```

函数 sigtimedwait() 与 sigwaitinfo() 相似，但是它可以设置超时选项，在规定的时间内等不到信号可以放弃等待。

sigtimedwait() 函数原型如下：

```

int sigtimedwait
(
const sigset_t * pSet, /*阻塞时的信号掩码 */
struct siginfo * pInfo, /*返回值 */
const struct timespec * pTimeout /* 等待的时间 */
)

```

6.1.3 信号配置

在配置 VxWorks 时，如果定义了 NCLUDE_SIGNALS，生成的系统将包含基本信号功能。在应用程序使用 POSIX 排队信号之前，必须调用函数 sigqueueInit()来初始化函数库。像基本信号初始化函数 sigInit()那样，这个函数正常情况下，一旦 sysInit()已经运行，由程序 usrConfig.c 中的 usrInit()函数调用。

为初始化排队信号功能，在配置 VxWorks 需要包含 INCLUDE_POSIX_SIGNALS 的定义。有了这个定义，sigqueueInit() 将自动调用。

调用 sigqueueInit(nQueues) 为函数 sigqueue()分配 nQueues 个缓冲，每个当前排队信号各需要一个缓冲。如果没有缓冲可用，sigqueue() 调用将失败。

表 6.2 POSIX 1003.1b 排队信号调用

函数调用	说 明
sigqueue()	发送一个排队信号
sigwaitinfo()	等待信号
sigtimedwait()	设置超时的信号等待

6.2 中断服务程序

硬件中断处理是实时系统设计的最重要、最关键的问题。由于中断通常对应着外部事件，



系统通过中断与外部事件交互。为了获得尽可能快的中断响应时间，VxWorks 的中断处理程序运行在特定的上下文中（在所有任务上下文之外）。因此，中断处理不会涉及任何任务上下文的交换。VxWorks 的库 `intLib` 和 `intArchLib` 提供中断函数，如表 6.3。

对于使用 MMU 的底板，可选产品 VxVMI 提供中断向量表写保护机制。

6.2.1 设置中断处理程序

应用程序可以使用 VxWorks 未用的硬件中断。VxWorks 提供函数 `intConnect()`，它允许将指定的 C 函数与任意中断相联系。

`intConnect()` 函数原型是：

```
STATUS intConnect
(
    VOIDFUNCPTR * vector, /* 要联系的中断向量 */
    VOIDFUNCPTR routine, /* 中断发生时要调用的函数 */
    int parameter /* 传递给中断处理函数的参数 */
)
```

该函数将指定的 C 函数 `routine` 与指定的中断向量 `vector` 相联系，函数的地址将存储在这个中断向量里。所以当中断发生时，系统将调用该函数，使用指定的参数 `parameter` 作为参数。中断处理程序在中断级以 `supervisor` 方式调用。将建立一个合适的 C 环境，保存必要的寄存器，建立堆栈。

中断处理函数可以是任何正常的 C 代码。但是它必须不能调用任何可能引起阻塞和执行 I/O 操作的操作系统函数。

该函数简单地调用 `intHandlerCreate()` 和 `intVecSet()`。处理程序的地址由 `intHandlerCreate()` 函数返回，实际上保存在中断向量中。

`intConnect()` 如果能够建造此中断处理程序，返回 OK，否则返回 ERROR。

表 6.3 中断函数

调 用	描 述
<code>intConnect()</code>	设置中断处理程序
<code>intContext()</code>	如果是在中断级调用，返回真
<code>intCount()</code>	得到当前的中断嵌套深度
<code>intLevelSet()</code>	设置处理器中断屏蔽级
<code>intLock()</code>	禁止中断
<code>intUnlock()</code>	重新允许中断



续表

调 用	描 述
intVecBaseSet()	设置向量基地址
intVecBaseGet()	得到向量基地址
intVecSet()	设置一个异常向量
intVecGet()	得到一个异常向量

事实上，中断向量不是直接地指向 intConnect() 指定的 C 函数。intConnect() 将创建一小段代码，这段代码用以保存必要寄存器、设置堆栈入口、包含将要传递的参数，或者在一个特殊的堆栈或者在当前任务的堆栈中调用这个连接函数。相反，当从该函数返回时，这段代码先恢复寄存器和堆栈，然后退出中断，如图 6.1。

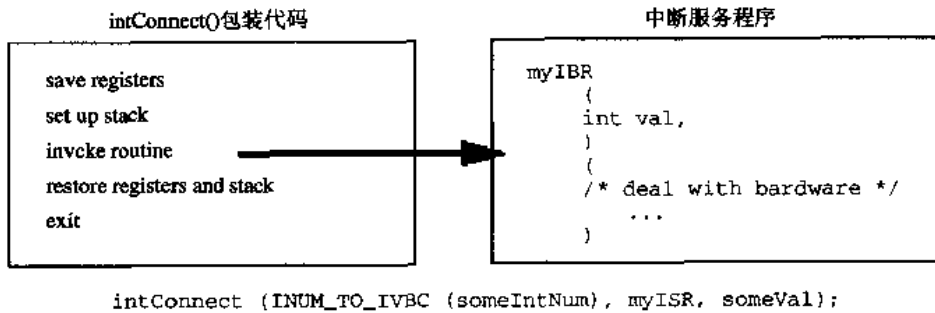


图 6.1 intConect()构造中断服务代码

中断处理过程如图 6.2 所示。

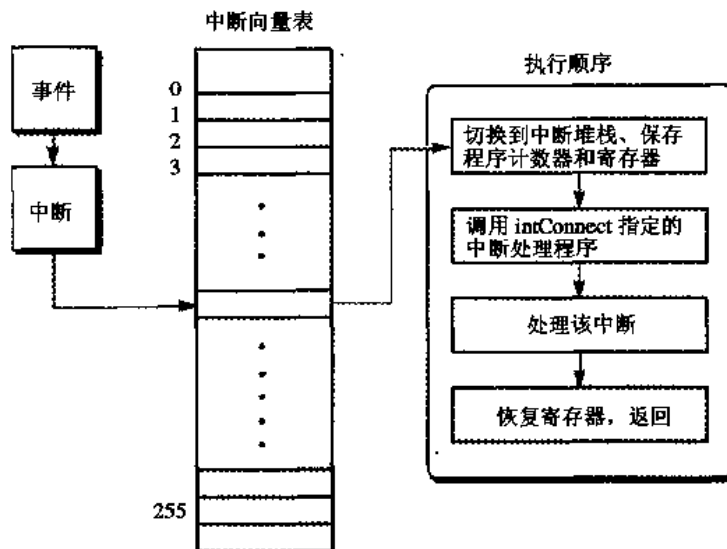


图 6.2 中断处理示意图



对于使用 VME 底板的目标版, BSP 提供两个标准的函数控制 VME 总线中断: `sysIntEnable()` 和 `sysIntDisable()`, 分别用于允许/禁止指定级别的中断。

对于使用 X86 体系的目标版, BSP 提供两个标准的函数控制 ISA 中断: `sysIntEnablePIC()` 和 `sysIntDisablePIC()`, 分别用于允许/禁止指定级别的中断。

6.2.2 中断堆栈

如果体系结构允许, 所有的 ISRs 使用相同的中断堆栈。堆栈的定位和初始化由系统在启动时根据指定的配置参数完成。堆栈必须足够大, 以保证能够处理系统最坏情形下的中断嵌套。

然而, 一些体系结构不允许使用一个特定的堆栈。在这种结构中, ISRs 使用中断任务的堆栈。对于这种结构的目标机, 应用必须创建有足够大堆栈空间, 以应付最坏情形下的中断嵌套和正常的任务嵌套调用的情形。

对于某种体系结构, 参考相应的 BSP 入口, 了解是否支持分离的中断堆栈。

在开发过程中, 可以调用 `checkStack()` 函数察看一个任务堆栈的使用情况或整个系统堆栈的使用情况。

6.2.3 ISR 的特殊限制

许多 VxWorks 函数可以在中断服务程序中使用, 但是有一些非常重要的限制。这些限制主要是因为中断服务程序不是在规则的任务上下文中运行。例如它没有任务控制块, 并且所有的中断处理程序共享一个堆栈。

由于这些原因, 中断服务程序必须遵循一个基本约束: 它必须不能调用可能引起调用阻塞的函数。例如: 在中断处理程序中不能试图获取一个信号量, 因为信号量可能不可用, 内核将该调用者切换到阻塞起态。然而, 中断服务程序可以释放一个信号量, 解除等待在该信号量上的任务。因为存储器函数 `malloc()` 和 `free()` 都需要获取一个信号量, 因而中断处理程序不能调用他们。例如中断服务程序不能调用任何创建和删除函数。

中断服务程序也不能通过 VxWorks 驱动执行 I/O 操作。尽管 I/O 系统不存在内在的约束, 多数设备驱动由于可能需要等待设备而引起调用者阻塞, 因此需要任务上下文交换。一个例外是 VxWorks 管道驱动, 它允许中断服务程序执行写操作。

VxWorks 支持记录功能, 任务可以向系统输出平台打印文本信息。这种机制为中断处理程序提供了特殊的支持, 中断处理程序可以调用 `logMsg()` 向系统输出信息。注意, `logMsg()` 必须带有 6 个整型参数。

`logMsg()` 的函数原型是:



```
int logMsg
(
    char * fmt,      /* 需要输出的格式化串 */
    int  arg1,      /* 需要格式化输出的六个参数的第一个 */
    int  arg2,
    int  arg3,
    int  arg4,
    int  arg5,
    int  arg6
)
```

更多的信息参考 `logLib` 库。

中断服务程序也不能调用使用浮点协处理器的函数。VxWorks 由 `intConnect()` 创建的中断驱动代码不能保存和恢复浮点寄存器。因此，中断服务程序不能包含浮点指令，如果一定需要的使用浮点指令，它必须明确调用 `fppArchLib` 库中的函数，保存和恢复浮点寄存器。

所有的 VxWorks 函数库，像连结链和环缓冲库，中断处理程序都可调用。正如前几章讨论的（任务错误状态 4.1.4），全局变量 `errno` 是作为中断进入和退出代码的一部分来保存和恢复的，这是由 `intConnect()` 函数来实现的。因此，`errno` 可以被其他代码访问和修改。

表 6.4 列出了中断服务程序可以调用的函数。

6.2.4 中断级异常

当任务引起一个异常时，例如非法指令或总线错，系统将该任务挂起，而系统的其余部分继续执行。而当中断引起一个异常时，则系统没有安全的资源用于处理异常。这是因为中断处理程序没有可以挂起的上下文。这时，VxWorks 首先将该异常的描述保存在低端内存，然后执行系统复位。每次引导时，VxWorks 引导 ROM 检查低端内存是否有异常描述出现，如果检查到，则将其显示在系统平台上，引导 ROM 的 `e` 命令重新显示该异常描述，参见《Tornado 用户手册: Setup and Startup》一节。

一个异常信息的例子如下：

```
workQPanic: Kernel work queue overflow.
```

这种异常通常是由中断处理程序以非常高频率的执行内核调用的结果。通常与中断信号的清除或其他相似的驱动问题有关。

图 6.3 给出了一个典型的异常处理过程。

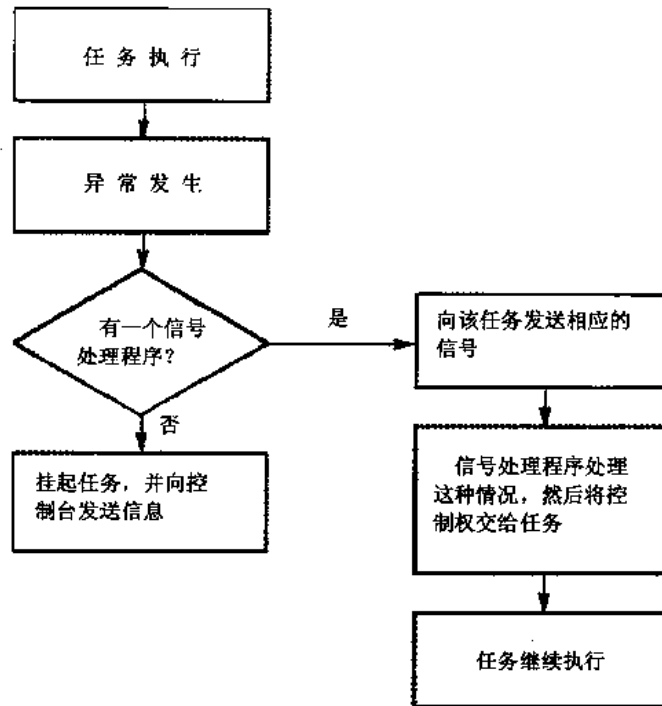


图 6.3 一个典型的异常处理过程

6.2.5 保留的最高中断级

在本章前面所述的 VxWorks 中断支持可以满足大多数应用需求。然而有时一些应用，比如对于重要运动控制和系统失败反应需要低级控制的应用而言，系统需要保留最高级别的中断，以确保对这些事件的零延时（zero-latency）反应。

为达到零延时反应，VxWorks 提供函数 `intLockLevelSet()`，用来设置整个系统的中断上锁级别。如果不指明级别，默认值是系统支持的最高中断级。默认值：MC680x0 是 7，SPARC 是 15，i960 是 31，i386/i486 是 1。

警告

一些硬件防止屏蔽某个中断级；请参考该硬件厂商的有关文档。例如，MC680x0 芯片，不允许屏蔽第 7 级中断。因为这种体系结构的第 7 级中断也是最高级别的中断级。VxWorks 使用它作为默认的上锁级，但是事实上等价于上锁第 6 级中断。这是因为硬件禁止上锁第 7 级中断。



表 6.4 中断服务程序可以调用的函数

库	函 数
lib	所有函数
errnoLib	errnoGet(), errnoSet()
fppArchLib	fppSave(), fppRestore()
intLib	intContext(), intCount(), intVecSet(), intVecGet()
intArchLib	intLock(), intUnlock()
logLib	logMsg()
intLib	除了 intFree()之外的所有函数
mathALib	所有函数, 如果使用 fppSave()/fppRestore()
msgQLib	msgQSend()
pipeDrv	write()
rngLib	除了 rngCreate()和 rngDelete()之外的所有函数
selectLib	selWakeup(), selWakeupAll()
semLib	除了互斥信号量 semGive(), semFlush()
sigLib	kill()
taskLib	taskSuspend(), taskResume(), taskPrioritySet(), taskPriorityGet(), taskIdVerify(), taskIdDefault(), taskIsReady(), taskIsSuspended(), taskTcb()
tickLib	tickAnnounce(), tickSet(), tickGet()
TyLib	tyIRd(), tyITx()
vxLib	vxTask(), vxMemProbe()
wdLib	wdStart(), wdCancel()

6.2.6 最高中断级 ISRs 的约束

与一个不能上锁的中断级（或者比由 intLockLevelSet() 设置的中断级高的中断级，或者由硬件设置的非屏蔽中断）相连接的中断服务程序有特定的约束：

- 仅有 intVecSet() 可以连结中断服务程序。
- 这个中断服务程序不能调用任何其正确操作与中断上锁有关的 VxWorks 操作系统功能调用。



6.2.7 中断与任务的通信

由于中断事件通常涉及到任务级代码，因此必须提供中断服务程序与一般任务的通信机制。VxWorks 支持运行在中断级运行的中断服务程序直接与一般任务进行通信。然而许多 VxWorks 功能不能在中断级代码中调用，包括对除了管道以外的设备的 I/O 操作。VxWorks 提供的中断服务程序与一般任务的通信机制有：

- 共享存储区和环缓冲，中断服务程序可以与任务共享变量、缓冲和环形缓冲。
- 信号量：中断服务程序能够释放信号量（不包括互斥信号量和 VxMP 共享信号量），任务能够等待该信号量。
- 消息队列，中断服务程序能够向消息队列发送消息，任务可以从消息队列接收消息（不包括 VxMP 使用的共享消息队列）。如果队列已满，消息则被丢弃。
- 管道，中断服务程序可以向管道写数据，任务可以从中读取。任务和中断服务程序能够向共享的管道写数据，但是如果管道已满，由于中断服务程序不允许阻塞，数据将被丢弃。中断服务程序不允许调用除 write() 之外的其他任何 I/O 函数调用。
- 信号灯，中断服务程序能够通过发信号来通知任务，触发相应的信号处理程序的异步调度。

6.3 看 门 狗

6.3.1 看门狗

VxWorks 提供一个看门狗定时器（Watchdog timer）机制，允许任何 C 函数与一个特定的时间延迟相联系。在 VxWorks 中，看门狗定时器作为系统时钟中断服务程序的一部分来维护，因此，通常与看门狗定时器相联系的函数以系统时钟中断级作为中断服务代码来执行。如果由于某种原因，如一个先前的中断或内核状态，不能立即执行某些函数，操作系统将该函数放在 tExcTask 工作队列。TExcTask 工作队列中的函数以 tExcTask（通常是 0）优先级运行。因此与看门狗定时器相联系的函数代码也有与中断服务程序相同的约束。

表 6.5 给出了 wdLib 提供的函数调用。

表 6.5 wdLib 提供的函数调用

调 用	描 述
WdCreate()	分配并初始化看门狗定时器



续表

调 用	描 述
WdDelete()	终止并释放看门狗定时器
wdStart()	启动看门狗定时器
wdCancel()	取消一个正在计时的看门狗定时器

看门狗定时器首先由 `wdCreate()` 创建，调用 `wdStart()` 启动计时器，一个以 tick 为单位的参数作为延时计时器的值，另一个参数是计时器结束时要调用的 C 程序。一旦指定的 tick 延时结束，该 C 程序立即被调用。在计时器计时结束之前的任意时刻，调用 `wdCancel()` 将取消该看门狗定时器。

例 6.1 看门狗定时器

```

/*****
 * This example creates a watchdog timer and sets it to go off in
 * 3 seconds.
 */

/* includes */
#include "vxWorks.h"
#include "logLib.h"
#include "wdLib.h"

/* defines */
#define SECONDS (3)
WDOG_ID myWatchDogId;

task (void)
{
    /* Create watchdog */
    if ((myWatchDogId = wdCreate( )) == NULL)
        return (ERROR);
    /* Set timer to go off in SECONDS - printing a message to stdout */
    if (wdStart (myWatchDogId, sysClkRateGet( ) * SECONDS, logMsg,
                "Watchdog timer just expired\n") == ERROR)
        return (ERROR);
    /* ... */
}
/*****

```

当需要延时一个任务的执行时，可以使用看门狗定时器。一旦一个任务调用 `wdStart()`



函数启动一个看门狗定时器，看门狗定时器进入延时态。一旦看门狗定时器计时结束，它将调用由 `wdStart()` 指定的一个中断服务程序。该中断服务程序在该任务的上下文之外执行。

该中断服务程序既可以完成某事件的必要处理，然后将控制交还给启动看门狗定时器的任务。也可以通过调用 `longjmp()` 函数，将控制转移到任务中的其他合适点执行，如图 6.4 所示。

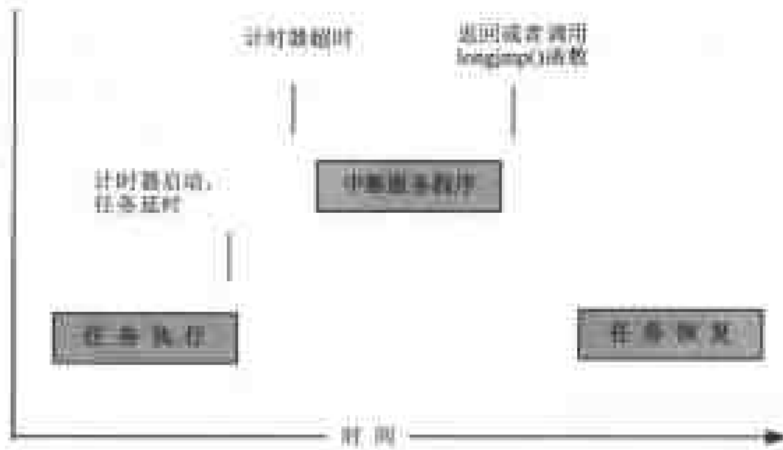


图 6.4 看门狗定时器的使用

6.3.2 利用看门狗处理任务时限

实时系统任务的一个重要特征是具有时限约束，任务执行一旦超出时限，系统可能导致灾难性的后果。因此执行超出时限的任务必须进行处理，以控制危害程度。通常启动一个时限事故处理任务（`deadline handlers`）。看门狗可以用来启动这种时限事故处理任务。下面给出一个例子。

协调任务（`CoordinatorTask`）发送数据给组织任务（`organizer Task`）。组织任务接收来自于协调任务的数据，如果 5 秒内（时限）没有数据发送，协调任务将复位。20 秒后，演示程序自动停止。

例 6.2 利用看门狗启动时限事故处理任务

```

/*****
* deadlineWdDemo.c - Demo for invoking deadline handlers using watchdog timers*/

/* Copyright 1992 Wind River Systems, Inc. */

```



```
/*
modification history
-----
01a,02mar94,ms  written
*/

/*
DESCRIPTION

This program demonstrates using watchdog timers to invoke deadline
handlers. CoordinatorTask sends data to the organizer. OrganizerTask
receives data from the coordinatorTask, and resets the coordinatorTask when
no data is sent by the coordinatorTask in the past five seconds
(deadline time). This demonstration program is automatically stopped after
twenty seconds.

EXAMPLE

To run this program from the VxWorks shell do as follows:

-> sp (deadlineWdDemo)

*/

/* includes */
#include "vxWorks.h"
#include "taskLib.h"
#include "msgQLib.h"
#include "wdLib.h"
#include "usrLib.h"
#include "sysLib.h"
#include "stdio.h"
#include "logLib.h"

/* defines */

#define FIVE_SEC      5
#define TWENTY_SEC   20
#define DEADLINE_TIME FIVE_SEC
#define NUM_MSGS     10
#define TASK_STACK_SIZE 20000
#define PRIORITY     101

/* globals */
LOCAL int countNum;
```



```

LOCAL int valueGot;
LOCAL BOOL working;
LOCAL BOOL notDone;
LOCAL WDOG_ID wdId;
LOCAL MSG_Q_ID msgQId;

/* function prototypes */
LOCAL void deadlineHandler (); /* deadline handler - watchdog handler
    * routine */
LOCAL STATUS coordinatorTask (); /* sends data to organizer */
LOCAL STATUS organizerTask (); /* receives data from the coordinator and
    * resets the coordinator when deadline
    * time elapses
    */
LOCAL void getDataFromDevice (); /* function that simulates data collection */
/*****
 * deadlineWdDemo - Demo for using watchdog timers to invoke deadline handlers.
 *
 */
STATUS deadlineWdDemo ()
{
    /* initialize the globals */
    notDone = TRUE;
    working = TRUE;
    countNum = 0;
    valueGot = 0;

    /* Create msgQ */
    if ((msgQId = msgQCreate (NUM_MSGS, sizeof (int), MSG_Q_FIFO)) == NULL)
    {
        perror ("Error in creating msgQ");
        return (ERROR);
    }

    /* Create watchdog timer */
    if ((wdId = wdCreate ()) == NULL)
    {
        perror ("cannot create watchdog");
        return (ERROR);
    }

    /* Spawn the organizerTask */
    if ((taskSpawn ("tOrgTask", PRIORITY, 0, TASK_STACK_SIZE,

```



```
        (FUNCPTR) organizerTask, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0)) == ERROR)
{
perror ("deadlinWdDemo: Spawning organizerTask failed");
return (ERROR);
}

/* Spawn the coordinatorTask */
if ((taskSpawn ("tCordTask", PRIORITY, 0, TASK_STACK_SIZE,
        (FUNCPTR) coordinatorTask, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0)) == ERROR)
{
perror ("deadlinWdDemo: Spawning coordinatorTask failed");
return (ERROR);
}

/* stop this demo after about 20 seconds*/
taskDelay (sysClkRateGet () * TWENTY_SEC);
printf ("\n\nStopping deadlineWdDemo\n");
notDone = FALSE;
if (msgQDelete (msgQId) == ERROR)
{
    perror ("Error in deleting msgQ");
    return (ERROR);
}
if (wdCancel (wdId) == ERROR)
{
    perror ("Error in cancelling watchdog timer");
    return (ERROR);
}
if (wdDelete (wdId) == ERROR)
{
    perror ("Error in deleting watchdog timer");
    return (ERROR);
}

return (ERROR);
}

/*****
* coordinatorTask - This task is assumed to collect data (countNum) from an
*                   external device and sends that data to the organizerTask.
*                   This task is constructed to miss the deadline (5 seconds)
*****/
```



```

*           for demonstration purpose. When this task misses deadline
*           (5 seconds), it gets reset by the deadlineHandler
*           (using watchdog timers).
*
* RETURNS: OK or ERROR
*
*/

STATUS coordinatorTask ()
{

    FOREVER
    {
        countNum ++;
        if (msgQSend (msgQId, (char *) &countNum, sizeof (int), NO_WAIT,
                     MSG_PRI_NORMAL)) == ERROR)
        {
            perror ("Error in sending the message");
            return (ERROR);
        }
        printf ("\n\ncoordinatorTask: Sent item = %d\n", countNum);
        printf ("coordinatorTask: idle for %d seconds\n", countNum);
        getDataFromDevice (); /* get data from the device */
        if (notDone == FALSE)
            break;
    }

    return (OK);
}

/*****
* organizerTask -  Receives data from the coordinatorTask, and resets the
*                  coordinatorTask when no data is sent by the
*                  coordinatorTask in the past five seconds (deadline time).
*
*
* RETURNS: OK or ERROR
*
*/

STATUS organizerTask ()
{

```



```
while (notDone)
{
    /* If coordinatorTask sends data within the deadline time (5 seconds),
    * the time elapsed by watchdog timer gets reset to the deadline time,
    * otherwise deadlineHandler is called to reset the coordinatorTask
    * when no data is sent by the coordinatorTask in the past five
    * seconds (deadline time).
    */
    if ((wdStart (wdId, sysClkRateGet ()* DEADLINE_TIME,
        (FUNCPTR) deadlineHandler, 0)) == ERROR)
    {
        perror ("Error in starting watchdog timer");
        return (ERROR);
    }
    if ((msgQReceive (msgQId, (char *) &valueGot, sizeof (int),
        WAIT_FOREVER)) == ERROR)
    {
        perror ("Error in receiving the message");
        return (ERROR);
    }
    else
        printf ("organizerTask: Received item = %d\n", valueGot);
}

return (OK);
}

/*****
* deadlineHandler - watchdog timer routine to reset the coordinatorTask
* when the deadline time expires
*
*/

LOCAL void deadlineHandler ()
{
    logMsg ("\n\nResetting the co-ordinator on elapse of the deadline
time\n", 0, 0, 0, 0, 0, 0);
    countNum = 0;
}

/*****
* getDataFromDevice - dummy function that delays for certain amount of time
```



```

*          to simulate the data collection for the purpose of
*          demonstration.
*/

LOCAL void getDataFromDevice ()
{
    taskDelay (sysClkRateGet () * countNum); /* delay this task for countNum
                                             * seconds. */
}
/*****

```

编译该程序，下载到目标机，运行 `deadlineWdDemo()` 函数。

在 VxWorkst 目标机上执行命令：

```

-> ld sp (deadlineWdDemo)
task spawned: id = 5d1728, name = u0
value = 6100776 = 0x5d1728

```

程序输出结果如下：

```

coordinatorTask: Sent item = 1
coordinatorTask: idle for 1 seconds
organizerTask: Received

coordinatorTask: Sent item = 2
coordinatorTask: idle for 2 seconds
organizerTask: Received item = 2

coordinatorTask: Sent item = 3
coordinatorTask: idle for 3 seconds
organizerTask: Received item = 3

coordinatorTask: Sent item = 4
coordinatorTask: idle for 4 seconds
organizerTask: Received item = 4

coordinatorTask: Sent item = 5
coordinatorTask: idle for 5 seconds
organizerTask: Received item = 5
interrupt:

```



Resetting the co-ordinator on elapse of the deadline time

```
coordinatorTask: Sent item = 1
coordinatorTask: idle for 1 seconds
organizerTask: Received item = 1
```

```
coordinatorTask: Sent item = 2
coordinatorTask: idle for 2 seconds
organizerTask: Received item = 2
```

```
coordinatorTask: Sent item = 3
coordinatorTask: idle for 3 seconds
organizerTask: Received item = 3
```

```
Stopping deadlineWdDemo
Error in receiving the message: errno = 0x3d0003
```

6.4 POSIX 时钟和计时器

6.4.1 POSIX 计时器

VxWorks 提供一个软件时钟（数据结构 `struct timespec`，定义在 `time.h`），用来以秒和纳秒两种单位纪录时间，这个软件时钟由系统时钟 `tick` 来修改。

VxWorks 同时提供一个 POSIX 1003.1b 标准时钟和计时器接口。

POSIX 标准支持识别多个虚拟时钟，但是整个系统仅需要一个实时时钟 `CLOCK_REALTIME`（定义在 `time.h` 中）。VxWorks 提供访问这个整个系统范围内实时时钟的函数，参见 `clockLib` 库。VxWorks 不支持虚拟时钟。

POSIX 计时器功能提供这种功能支持：任务可以在一段时间后通知自身。程序可以创建、设置和删除一个计时器。参见 `timerLib` 库。当计时器到达，将向该任务发送默认的信号（`SIGALRM`）。函数 `sigaction()` 将安装一个信号灯处理程序，该程序将在计时器结束时调用。

例 6.3 POSIX 计时器

```
/* ***** */
/* This example creates a new timer and stores it in timerid. */
```



```

*/

/* includes */
#include "vxWorks.h"
#include "time.h"

int createTimer (void)
{
    timer_t timerid;
    /* create timer */
    if (timer_create (CLOCK_REALTIME, NULL, &timerid) == ERROR)
    {
        printf ("create FAILED\n");
        return (ERROR);
    }
    return (OK);
}
/*****

```

另一个 POSIX 函数 `nanosleep()`，允许指定一个以秒和纳秒为单位的睡眠或延时时间，其功能与 Wind 核中的 `taskDelay()` 类似，后者以 tick 作为延时单位。二者只是延时单位不同，而不是精度不同，这两个延时函数有相同的精度，都由系统时钟频率决定。

6.4.2 利用 POSIX 计时器处理任务时限

POSIX 计时器也可用于处理任务时限。下面给出一个例子。

例子中，将创建一个与一个 POSIX 计时器相联系的信号处理程序，在 POSIX 计时器演示一段时间超时，该信号处理程序将启动执行。

例 6.4 利用 POSIX 计时器处理任务时限

```

/*****
/* posixTimerStartDemo.c - Demo for executing a signal handler connected to a
   POSIX TIMER upon expiration of delay specified in
   seconds
*/

/* Copyright 1992 Wind River Systems, Inc. */

/*
modification history

```



```
-----
02,24sep97,ram corrected the timer_create function call
11,15may95,ldt Added comments and include files. stdio.h, logLib.h added
    null parameters to logMsg calls.
01a,10mar94,ms written.
*/

/* includes */

#include "vxWorks.h"
#include "semLib.h"
#include "signal.h"
#include "sigLib.h"
#include "timers.h"
#include "taskLib.h"
#include "stdio.h"
#include "logLib.h"

/* defines */

#define TASK_PRI      254 /* Priority of spawned tasks */
#define TASK_STACK_SIZE 10000 /* stack size for spawned tasks */
#define DELAY        5 /* 5 seconds delay */
#define TIMER_RELATIVE 0 /* interval relative to the current time */

/* globals */

LOCAL SEM_ID semId; /* semaphore ID */

/* funtion prototypes */
LOCAL void demoTask ();
LOCAL STATUS posixTimerStart (int);
LOCAL void sigHandler (int, int, SIGCONTEXT *);

/*****
* posixTimerStartDemo - Demonstrates executing a signal handler connected to a
*                       POSIX TIMER upon expiration of delay specified in
*                       seconds.
*
* make sure that the following are INCLUDED in your configAll.h file:
* #define INCLUDE_POSIX_SIGNALS ** POSIX queued signals **
* #define INCLUDE_POSIX_TIMERS ** POSIX timers **
*
*****/
```



```

* EXAMPLE:
*
* To run this posixTimerStartDemo from the VxWorks shell do as follows:
* -> sp (posixTimerStartDemo)
*
*/

STATUS posixTimerStartDemo ()
{
    /* Spawn the demoTask */
    if (taskSpawn ("demoTask", TASK_PRI, 0, TASK_STACK_SIZE, (FUNCPTR) demoTask,
                  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
    {
        perror ("posixTimerStartDemo: Error in spawning demoTask");
        return (ERROR);
    }

    /* Spwan the posixTimerStart task */
    if (taskSpawn ("posixTimerStart", TASK_PRI, 0, TASK_STACK_SIZE,
                  (FUNCPTR) posixTimerStart, DELAY, 0, 0, 0, 0, 0, 0, 0,
                  0, 0) == ERROR)
    {
        perror ("posixTimerStartDemo: Error in spawning posixTimerStart task");
        return (ERROR);
    }

    /* set up semaphore for synchronization */
    if ((semId = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY)) == NULL)
    {
        perror ("posixTimerStartDemo: semBCreate failed");
        return (ERROR);
    }

    return (OK);
}

/*****
* demoTask - Demonstration task that Synchronize with SIGALRM signal
*           generated by a POSIX timer
*
*/

void demoTask ()

```



```
{
    printf (
        "demoTask: Waiting to be synchronized by posix timer SIGALRM signal\n");

    if (semTake (semId, WAIT_FOREVER) == ERROR)
    {
        perror ("intSync: semTake failed");
        return;
    }
    printf ("\ndemoTask: Synchronization done\n\n");

    /* Do the work that need to be synchronized */
    printf ("demoTask: RUNNING\n");
}

/*****
 * sigHandler - Signal handler gets executed when SIGALRM signal is generated
 *             by a POSIX timer.
 */

void sigHandler
{
    int sig, /* signal number */
    int code, /* additional code */
    SIGCONTEXT *sigContext /* context of task before signal */
}
{
    if (sig == SIGALRM)
    {
        logMsg ("\nSignal SIGALRM raised", 0, 0, 0, 0, 0, 0);
    }
    else
    {
        logMsg ("\nSignal %d received", sig, 0, 0, 0, 0, 0);
    }
    if (semGive (semId) == ERROR)
    {
        logMsg ("Error in giving semaphore", 0, 0, 0, 0, 0, 0);
    }
}
}
```



```
/******  
* posixTimerStart - This routine creates a POSIX timer. Signal handler  
*                   routine will be called to unblock demoTask from the  
*                   interrupt level upon the expiration of delay (in the  
*                   specified number of seconds) by the POSIX timer. Finally  
*                   created POSIX timer is deleted. For more information on  
*                   POSIX timers, please refer to the timerLib man pages.  
*  
* RETURNES OK or ERROR  
*/
```

```
STATUS posixTimerStart
```

```
{  
    int delay    /* delay count in seconds */  
}  
{  
    timer_t timerId;          /* id for the posix timer */  
    struct itimerspec timeToSet; /* time to be set */  
    struct timespec   timeValue; /* timer expiration value */  
    struct timespec   timeInterval; /* timer period */  
    struct sigaction  signalAction; /* signal action handler struct */  
  
    /* Intialize sigaction struct */  
    signalAction.sa_handler = (VOIDFUNCPTR) sigHandler;  
    signalAction.sa_mask   = 0;  
    signalAction.sa_flags  = 0;  
  
    /* Initialize timer expiration value */  
    timeValue.tv_sec      = delay;  
    timeValue.tv_nsec     = 0;  
  
    /* Initialize timer period */  
    timeInterval.tv_sec   = delay;  
    timeInterval.tv_nsec  = 0;  
  
    /* Set the time to be set value */  
    timeToSet.it_value    = timeValue;  
    timeToSet.it_interval = timeInterval;  
  
    /* Connect a signal handler routine to the SIGALRM signal */  
    if (sigaction (SIGALRM, &signalAction, NULL) == ERROR)  
    {  
        perror ("posixTimerStart: Error in executing sigaction");  
    }  
}
```



```
    return (ERROR);
}

printf ("posixTimerStart: Unblock the demoTask in %d seconds\n", delay);

/* Allocate a timer */
if ((timer_create (CLOCK_REALTIME, NULL,&timerId)) == ERROR)
{
    perror ("posixTimerStart: Error in allocating a timer");
    return (ERROR);
}

/* set the time until the next expiration and arm the timer (POSIX) */
if (timer_settime (timerId, TIMER_RELATIVE, &timeToSet, NULL) == ERROR)
{
    perror ("posixTimerStart: Error in setting time");
    return (ERROR);
}

/* suspend this task until delivery of a signal (POSIX)*/
sigsuspend (0);

/* delete the previously created timer */
if (timer_delete (timerId) == ERROR)
{
    perror ("posixTimerStart: Error in removing timer (POSIX)");
    return (ERROR);
}

return (OK);
}
/*****/
```

在运行演示程序之前，配置 VxWorks，包含了 POSIX 信号。

编译、下载演示程序目标文件到目标机。运行 `posixTimerStartDemo()` 函数。

在 VxWorks 目标机上执行命令：

```
-> ld sp (posixTimerStartDemo)
task spawned: id = 1fb54e8, name = u0
value = 33248488 = 0x1fb54e8
```

输出信息如下：



```
demoTask: Waiting to be synchronized by posix timer SIGALRM signal
posixTimerStart: Unblock the demoTask in 5 seconds
0x1fad70 (posixTimerStart):
Signal SIGALRM raised
demoTask: Synchronization done

demoTask: RUNNING
```

6.5 POSIX 内存上锁接口

许多操作系统实现内存分页和交换功能。这些技术通过将内存块数据拷贝到硬盘和从硬盘拷贝数据到内存，从而允许应用使用比物理内存大的多的存储空间，也就是虚拟内存。显然这些技术使得系统的执行难以预测，因而不适合于实时系统。

由于 wind 核是为实时系统设计，因而不能采用内存分页和交换技术。然而 POSIX 1003.1b 实时扩展标准包含实现分页和交换技术的操作系统。这些系统中，需要实时性能的应用可以利用 POSIX 锁页功能，也就是说，强制声明一块内存是不能被分页的和不能被交换到硬盘的，从而保证系统操作这块内存的确定性。Windows NT 也实现了这种锁页机制。

为了实现最大可能的可移植性，VxWorks 实现了 POSIX 锁页函数调用。

这些函数的实现、执行在 VxWorks 中并不存在困难，因为 VxWorks 中所有内存都是有效的（不会发生交换），总是“上锁”的。实现它们仅是为了其他 POSIX 兼容系统与 VxWorks 系统之间应用程序移植的方便。

POSIX 锁页函数在 mmanPxLib 库中，从名字可以看出，这些函数是 POSIX 内存管理（memory-management）函数的一部分。

由于 VxWorks 系统中的所有成分也都在内存中，所以表 6.6 中的所有函数调用均返回 OK (=0)，也不会对系统产生其他影响。

表 6.6 POSIX 内存管理函数

调 用	描 述
mlockall()	锁住一个任务使用的所有内存页
munlockall()	解锁一个任务使用的所有内存页
mlock()	锁住指定的内存页
unlock()	解锁指定的内存页



在使用工程配置工具配置 VxWorks 时，如果包含 INCLUDE_POSIX_MEM 选项，操作系统将会自动包含 MmanPxLib 库。

通过前几章介绍，我们基本了解了 VxWorks 多任务编程的知识，本章将介绍主机/目标机开发环境，并介绍一个多任务编程实例。

7.1 建立调试环境

Tornado 采用支持主机/目标机开发模式。本节以 x86 系列目标机为例介绍调试环境的建立。

7.1.1 配置文件 config.h

在 x86 系列的目标机上运行的 VxWorks 系统包括两部分：引导文件 `bootrom.sys` 和操作系统映像文件 `VxWorks`。

引导文件 `bootrom.sys` 的主要作用类似于 BIOS，一般它存放在一张软盘或目标机硬盘或目标机 FLASH 盘上，由 `Vxld` 或其他程序加载到内存。用于初始化目标机上的包括引导硬件在内的硬件，建立 VxWorks 运行的环境，从引导设备上加载 VxWorks 操作系统映像，并将 CPU 的控制权移交给操作系统。

`VxWorks` 是操作系统的映像文件。它是应用程序和目标代理程序（调试环境的目标机部分）运行的软件平台，一般存放在 `bootrom.sys` 所在软盘或目标机硬盘或目标机 FLASH 盘，或主机硬盘上，由 `bootrom.sys` 的引导程序加载到目标机内存。

`bootrom.sys` 和 `VxWorks` 这两个文件可以利用 Tornado 提供的工具，按照 BSP 配置文件 `config.h` 有关设置自动生成。特定 BSP 的配置文件 `config.h` 在该 BSP 目录下。如 `pc486` 的配置文件是 `Tornado\target\config\pc486\config.h`。



配置文件 config.h 主要定义了引导行、目标机操作系统包含的主要成分，如软驱、IDE 硬盘、SCSI 设备、网络等设备驱动，文件系统（DOSFS，TFFS，CDROMFS 等），调试方式，内存地址等及有关参数。

引导行（boot line）定义了引导设备、引导路径、操作系统文件名、主机/目标机 IP 地址、子网掩码、FTP 用户名和口令等参数。

对于 X86 平台目标机，引导设备可以是软盘（fd）、硬盘（ata 或 ide）、FLASH 盘（tffs）和网卡（elt、ene 等）等。一般采用通过以太网或通过 RS232 或 RS422 标准串口连结进行调试。

在 X86 平台上，首先确定调试手段，是通过网络还是串口调试？然后按下面的步骤建立调试环境：

- (1) 修改配置文件。
- (2) 生成 bootrom_uncmp 引导文件和 VxWorks 映像文件。
- (3) 制作启动软盘。
- (4) 配置主机环境。
- (5) 用启动软盘启动目标机。
- (6) 从主机搭接（attach）到目标机。

7.1.2 网络连接

使用以太网连结调试方式，通过网络使用 FTP 协议从主机下载 VxWorks 映像。在 X86 平台上，一般使用启动软盘来启动目标机。首先要确定目标机使用的网卡型号。VxWorks5.4 支持的网卡可以参考 config.h 文件，需要包含相应的 INCLUDE_XXX。常用的如表 7.1。

表 7.1 VxWorks5.4 支持的几种常用网卡

INCLUDE_XXX	网络接口卡	备注
INCLUDE_ENE	Eagle/Novell NE2000 接口卡	包括兼容卡，不支持即插即用方式
INCLUDE_ELT	3COM EtherLink III 接口卡	包括兼容卡，不支持即插即用方式
INCLUDE_EL_3C90X_END	3com fast etherLink XL PCI 网卡	3Com3c905 系列
INCLUDE_FEI	Intel Ether Express PRO100B PCI 接口卡	
INCLUDE_LN_97X_END	AMD 79C972 网卡	



➤ 修改 config.h

修改 config.h 最重要的一步是修改引导行。引导文件 bootrom.sys 根据引导行来确定引导设备、引导路径、操作系统文件，并且引导行存放在固定的位置。操作系统访问引导行来确定网络的配置，系统 reboot 时也将访问引导行。bootrom.sys 和 VxWorks 将它转换为一个特定的引导参数结构 BOOT_PARAMS。

在 config.h 中，引导行定义为有特定格式的字符串。BOOT_PARAMS 定义在 h\bootlib.h 中。解释如下：

```
typedef struct                                /* 引导参数结构 BOOT_PARAMS */
{
    char bootDev [BOOT_DEV_LEN];             /* 引导设备代码 */
    char hostName [BOOT_HOST_LEN];          /* 主机名 */
    char targetName [BOOT_HOST_LEN];       /* 目标机名 */
    char ead [BOOT_TARGET_ADDR_LEN];       /* 目标机以太网地址*/
    char bad [BOOT_TARGET_ADDR_LEN];       /* 底板以太网地址 */
    char had [BOOT_ADDR_LEN];              /* 主机以太网地址 */
    char gad [BOOT_ADDR_LEN];              /* 以太网网关 */
    char bootFile [BOOT_FILE_LEN];         /* 引导文件名 */
    char startupScript [BOOT_FILE_LEN];    /* 启动脚本文件名*/
    char usr [BOOT_USR_LEN];                /* 用户名 */
    char passwd [BOOT_PASSWORD_LEN];       /* 口令 */
    char other [BOOT_OTHER_LEN];           /* 留给应用程序使用 */
    int procNum;                             /* 处理器号*/
    int flags;                                /* 配置标示*/
    int unitNum;                              /* 网络设备编号 */
} BOOT_PARAMS;
```

以 ISA 3COM3c509b 网卡为例，网卡需要设置为非即插即用方式，中断号为 5，IO 端口为 0x300。主机 IP 地址为 222.1.5.169，目标机 IP 地址为 222.1.5.36。主机名为 VxHost，有一 FTP 用户 VxMe，口令为 isMe。

对应的引导行应为：

```
"elt(0,0)VxHost:\tornado\target\config\pc486\vxWorks h=222.1.5.169 e=222.1.5.36
u=VxMe pw=isMe tn=VxTarget"
```

解释如下：

```
elt          /* 启动设备名，为 ISA3COM3c509 网卡，ne2000 网卡应为 ene */
VxHost       /* 主机标志名，可以任意填写，不影响启动过程 */
\tornado\target\config\pc486\vxWorks /* 需要从主机加载的路径及映像文件名 */
```



```
h=222.1.5.169      /* 主机的 IP 地址 */
e=222.1.5.36      /* 目标机的 IP 地址 */
u= VxMe           /* 用户名, 主机的 Ftp 服务器必须有相应的同名用户 */
pw= isMe          /* 口令, 必须与主机的 Ftp 服务器相应的同名用户的口令相同*/
tn= VxTarget      /* 目标名, 可以任意设置, 不影响启动过程*/
```

将下面的一行

```
#undef INCLUDE_ELT      /* uninclue 3COM EtherLink III interface */
```

改为:

```
#define INCLUDE_ELT     /* include 3COM EtherLink III interface */
```

还要修改下面的定义

```
#define IO_ADRS_ELT0x240
#define INT_LVL_ELT0x0b
```

为:

```
#define IO_ADRS_ELT0x300      /* 网卡 I/O 地址 */
#define INT_LVL_ELT0x05      /* 网卡中断号 */
```

config.h 修改完毕, 下一步是生成目标文件 bootrom.sys 和 VxWorks。

➤ 生成目标文件 bootrom.sys 和 VxWorks

先生成 bootrom_uncmp。

在 Tornado 集成环境中执行菜单命令 Build-)> Build Boot ROM..., 在弹出的对话框中, 左边框中选择 BSP, 选中 pc486, 右边框中选择要生成的映像文件, 选择 bootrom_uncmp, 然后点击 OK 确认。Tornado 将生成 bootrom_uncmp。

再生成 VxWorks。

在 Tornado 集成环境中执行菜单命令 Build-)> Standard BSP Builds... (如果没有该项, 请执行菜单命令 Tools-)> Options..., 在弹出的对话框中, 选中 Project 页, 选择 Show Tornado 1.0.1 menu items, 点击 OK 确认)。与生成 bootrom_uncmp 类似, 在弹出的对话框中, BSP 选择 pc486, 映像文件选择 VxWorks。然后点击 OK 确认。Tornado 将生成 VxWorks。

注意, 必要时先进行 clean 操作。

➤ 制作启动软盘

拷贝 Tornado\target\config\pc486\bootrom_uncmp 至 Tornado\host\bin 下;

准备一张已格式化的空盘插入软驱;

在目录 Tornado\host\x86-win32\bin 下执行命令:



```
mkboot a: bootrom_uncmp
```

注意，有时，我们需要改变配置参数，重新生成引导文件，如果软盘已经是可引导的，那么只需更换软盘上的引导文件即可，方法是：

```
del a:. (需要键入'y'确认删除)
vxcopy bootrom_uncmp a:bootrom.sys
```

► 配置主机环境

主机操作系统 Win95 安装目录下有一文件 hosts.sam，向其中加入：

- ◆ 主机 IP 主机名
- ◆ 目标机 IP 目标机名

启动 Tornado 组件 FTP Server，在 WFTPD 窗口中选择菜单 Security 中的 User/right...，在其弹出窗口中选择 New User...，根据提示信息输入登录用户名和口令，并且要指定下载文件 VxWorks 所在根目录。可能还需要选取主菜单 Logging 中 Log options、Enable Logging、Gets、Logins、Commands、Warnings 选项。

► 用启动软盘启动目标机

将系统引导软盘插入目标机软驱，加电启动，目标机即通过 FTP 方式从主机下载 VxWorkst 系统。

在控制台上可以看到启动信息。如果需要修改，在等待用户配置时，按 c 键，进行相应修改。（注意：配置信息要与主机配置、Ftp 服务器配置一致），修改结束后，按 @ 键重新启动目标机。

► 从主机搭接 (attach) 到目标机

在 Tornado 集成环境中点取 Tools 菜单，选取 Target Server，选择 config...；

在 Configure Target Servers 窗口中先给目标服务器命名；

在配置目标服务器窗口中的“Target Servers Property”窗口中，选择 Back End；在“Available Back”窗口中选择 wdbrpc，在“Target IP/Address”窗口中输入目标机 IP（本例为 222.1.5.36）；

在配置目标服务器窗口中的“Target Servers Property”窗口中，选择 Core File and Symbols，选择 File 为 BSP 目标文件所在目录（本例为 PC486 目录）的 VxWorks，并选取为 All Symbols；

在配置目标服务器窗口中的“Target Servers Property”窗口中的其他各项可根据需要选择；

点击 Launch 按钮，连接主机和目标机，全部出现 succeeded 后即可进入应用程序调试；点击图形按钮中下拉框，选择和主机相连的目标机。即可建立主机目标机连接。



这时 Shell、Debugger 等按钮可用。

7.1.3 串口连接

串口连接一般从启动软盘下载 VxWorks 映像，其步骤如下。

➤ 修改通用配置文件

在 config.h 文件中加入以下宏定义：

```
#define INCLUDE_WDB
#define INCLUDE_WDB_TTY_TEST
#undef WDB_COMM_TYPE
#define WDB_COMM_TYPE WDB_COMM_SERIAL /*定义通信方式为串口联结*/
#define WDB_TTY_CHANNEL 1 /*通道号，目标机串口 2*/
#define WDB_TTY_BAUD 9600 /*串口速率，VxWorks 最高可设置至 38400*/
#define WDB_TTY_DEV_NAME "tyCo/1"
#define CONSOLE_TTY 0
```

引导行为

```
#define DEFAULT_BOOT_LINE \
"fd=0,0 (0,0) hostname: /fd0/vxWorks h=222.1.5.169 e=222.1.5.36 u=VxMe"
```

➤ 生成目标文件 bootrom.sys 和 VxWorks

同网络连接。

➤ 制作启动软盘

同网络连接。同时要把 Tornado\target\config\pc486\VxWorks 拷贝至软盘。

➤ 配置主机环境

不需要。

➤ 用启动软盘启动目标机

将系统制作盘插入目标机软驱，加电启动，目标机即由软盘下载 VxWorks 系统。

➤ 从主机搭接 (attach) 到目标机

在 Tornado 集成环境中点取 Tools 菜单，选取 Target Server，选择 config...:



在 Configure Target Servers 窗口中先给目标服务器命名；

在配置目标服务器窗口中的“Target Servers Property”窗口中选择 Back End，在“Available Back”窗口中选择 wdbserial，再在“Serial Port”窗口中选择主机与目标机连接所占用的串口号（COM1，COM2），再在“Speed (bps)”窗口中选择主机与目标机间串口速率；

在配置目标服务器窗口中的“Target Servers Property”窗口中选择 Core File and Symbols，选择 File 为 BSP 目标文件所在目录（本例为 PC486 目录）的 VxWorks，并选取为 All Symbols；

在配置目标服务器窗口中的“Target Servers Property”窗口中的其他各项可根据需要选择；

点击 Launch 按钮，连接主机和目标机，全部出现 succeeded 后即可进入应用程序调试；

点击图形按钮中下拉框，选择和主机相连的目标机。即可建立主机目标机连接。

这时 Shell、Debugger 等按钮可用。

7.2 实例分析

Tornado 采用支持主机/目标机开发模式。本节以 x86 系列目标机为例介绍调试环境的建立。

7.2.1 程序简介

该程序主要工作是重复检查 Wind 内核的不同功能。包括以下内容：

- 用于同步和互斥原语的信号量。
- 用于任务控制的函数 taskSuspend()/taskResume()。
- 任务间通信的消息队列。
- 处理任务超时的看门狗等等。

本例中使用到两个任务，一个高优先级任务和一个低优先级任务。高优先级的任务执行需要使用不可使用的资源。当高优先级的任务被堵塞时，低优先级的任务获得 CPU 使用权，开始执行，并使高优先级任务等待的资源可用。

这个程序似乎很简单，但是这个小程序执行时需要涉及到很多操作系统也是多数应用所使用的最基本功能：上下文切换、任务调度、阻塞队列、就绪队列以及定时器队列的管理。因此这个小程序基本上可以验证的 Wind 内核功能的正确性：同步、互斥、任务控制、任务间通信和定时器等。

因此这个“死循环”程序循环执行的状态，最能说明操作系统这些功能 real-life 性能的好坏。



有关函数的说明可以参考前面的章节。

图 7.1 是它的执行流程。图中虚线是 CPU 控制权流。

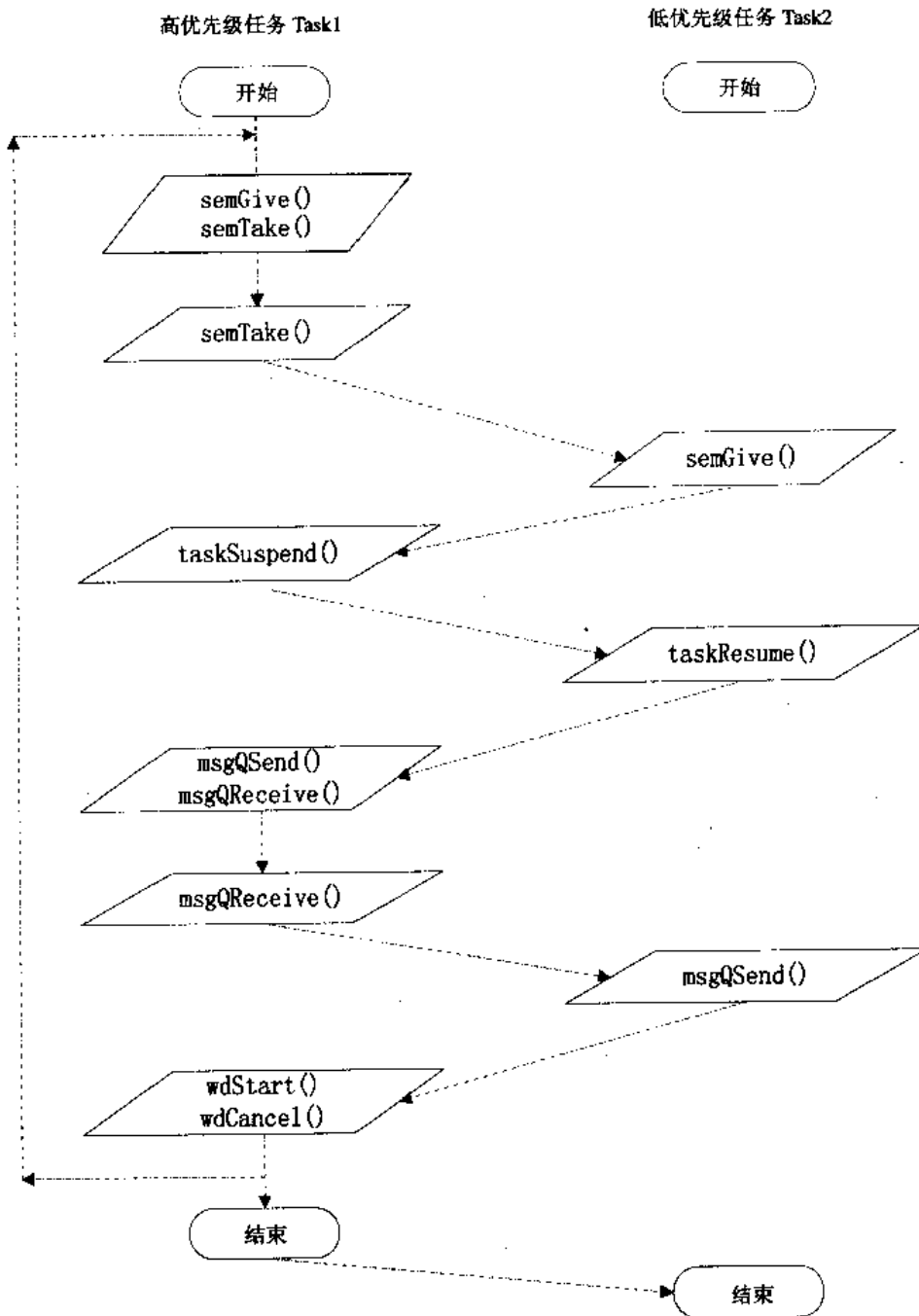


图 7.1 示例程序执行流程

我们可以利用 WindView 分析它的实际执行流程如图 7.2、图 7.3。

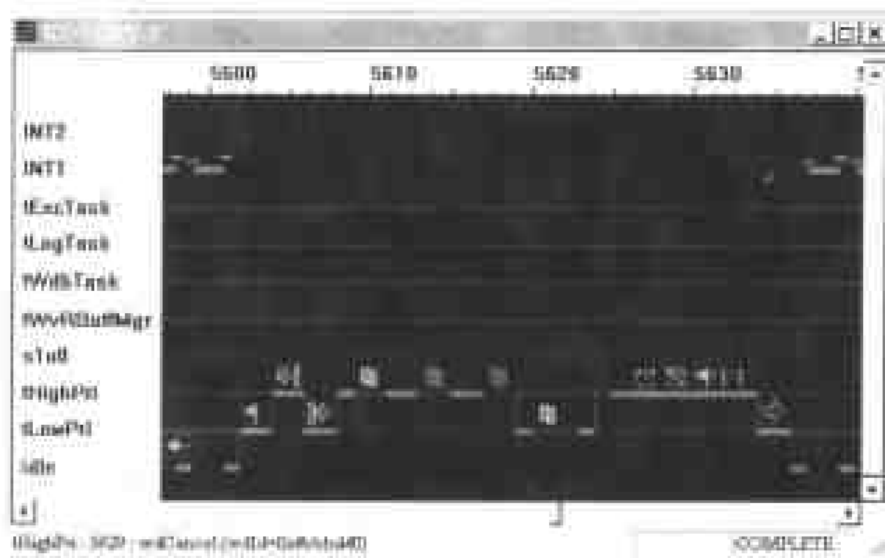


图 7.2 示例程序一个周期执行流程

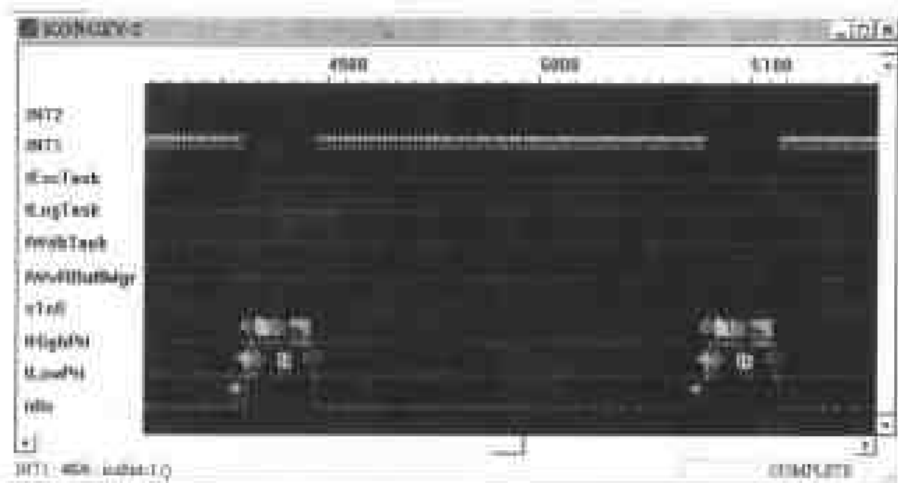


图 7.3 示例程序两个相邻周期执行流程

7.2.2 源代码

例 7.1 演示程序

```

/*****
#include "vxWorks.h"
#include "semLib.h"

```



```
#include "taskLib.h"
#include "msgQLib.h"
#include "wdLib.h"
#include "logLib.h"
#include "tickLib.h"
#include "sysLib.h"
#include "stdio.h"

/* defines */

#if FALSE
#define STATUS_INFO          /* define to allow printf() calls */
#endif

#define MAX_MSG              1          /* max number of messages in queue */
#define MSG_SIZE             sizeof (MY_MSG) /* size of message */
#define DELAY                100       /* 100 ticks */
#define HIGH_PRI             150       /* priority of high priority task */
#define LOW_PRI              200       /* priority of low priority task */

#define TASK_HIGHPRI_TEXT    "Hello from the 'high priority' task"
#define TASK_LOWPRI_TEXT    "Hello from the 'low priority' task"

/* typedefs */

typedef struct my_msg
{
    int  childLoopCount;      /* loop count in task sending msg */
    char * buffer;           /* message text */
} MY_MSG;

/* globals */

SEM_ID      semId;          /* semaphore ID */
MSG_Q_ID    msgQId;        /* message queue ID */
WDOG_ID     wdId;          /* watchdog ID */
int         highPriId;     /* task ID of high priority task */
int         lowPriId;      /* task ID of low priority task */
int         windDemoId;    /* task ID of windDemo task */
```



```

/* forward declarations */

LOCAL void taskHighPri (int iteration);
LOCAL void taskLowPri (int iteration);

/*****
 *
 *
 * windDemo - parent task to spawn children
 *
 * This task calls taskHighPri() and taskLowPri() to do the
 * actual operations of the test and suspends itself.
 * Task is resumed by the low priority task.
 *
 */

void windDemo
(
    int iteration                /* number of iterations of child code */
)
{
    int loopCount = 0;          /* number of times through windDemo */

#ifdef STATUS_INFO
    printf ("Entering windDemo\n");
#endif /* STATUS_INFO */

    if (iteration == 0)          /* set default to 10,000 */
        iteration = 10000;

    /* create objects used by the child tasks */

    msgQId    = msgQCreate (MAX_MSG, MSG_SIZE, MSG_Q_FIFO);
    semId     = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
    wdId = wdCreate ();

    windDemoId = taskIdSelf ();

    FOREVER
    {

        /* spawn child tasks to exercise kernel routines */

        highPriId = taskSpawn ("tHighPri", HIGH_PRI, VX_SUPERVISOR_MODE, 4000,

```



```
(FUNCPTR) taskHighPri, iteration,0,0,0,0,0,0,0,0,0);

lowPriId = taskSpawn ("tLowPri", LOW_PRI, VX_SUPERVISOR_MODE, 4000,
(FUNCPTR) taskLowPri, iteration,0,0,0,0,0,0,0,0,0);

taskSuspend (0);      /* to be waken up by taskLowPri */

#ifdef STATUS_INFO
    printf ("\nParent windDemo has just completed loop number %d\n",
            loopCount);
#endif /* STATUS_INFO */

    loopCount++;
}
}

/*****
*
* taskHighPri - high priority task
*
* This tasks exercises various kernel functions. It will block if the
* resource is not available and relingish the CPU to the next ready task.
*
*/

LOCAL void taskHighPri
(
    int iteration          /* number of iterations through loop */
)
{
    int ix;                /* loop counter */
    MY_MSG msg;            /* message to send */
    MY_MSG newMsg;         /* message to receive */

    for (ix = 0; ix < iteration; ix++)
    {

        /* take and give a semaphore - no context switch involved */

        semGive (semId);
        semTake (semId, 100);      /* semfake with timeout */
    }
}

```



```
/*
 * take semaphore - context switch will occur since semaphore
 * is unavailable
 */

semTake (semId, WAIT_FOREVER); /* semaphore not available */

taskSuspend (0);           /* suspend itself */

/* build message and send it */

msg.childLoopCount = ix;
msg.buffer = TASK_HIGHPRI_TEXT;

msgQSend (msgQId, (char *) &msg, MSG_SIZE, 0, MSG_PRI_NORMAL);

/*
 * read message that this task just sent and print it - no context
 * switch will occur since there is a message already in the queue
 */

msgQReceive (msgQId, (char *) &newMsg, MSG_SIZE, NO_WAIT);

#ifdef STATUS_INFO
printf ("%s\n Number of iterations is %d\n",
        newMsg.buffer, newMsg.childLoopCount);
#endif /* STATUS_INFO */

/*
 * block on message queue waiting for message from low priority task
 * context switch will occur since there is no message in the queue
 * when message is received, print it
 */

msgQReceive (msgQId, (char *) &newMsg, MSG_SIZE, WAIT_FOREVER);

#ifdef STATUS_INFO
printf ("%s\n Number of iterations by this task is: %d\n",
        newMsg.buffer, newMsg.childLoopCount);
#endif /* STATUS_INFO */

/* test watchdog timer */

wdStart (wdId, DELAY, (FUNCPTR) tickGet, 1);
```



```
        wdCancel (wdId);
    }
}

/*****
*
*
* taskLowPri - low priority task
*
* This task runs at a lower priority and is designed to make available
* the resouces that the high priority task is waiting for and subsequently
* unblock the high priority task.
*
*/

LOCAL void taskLowPri
(
    int iteration          /* number of times through loop */
)
{
    int ix;                /* loop counter */
    MY_MSG msg;           /* message to send */

    for (ix = 0; ix < iteration; ix++)
    {
        semGive (semId);   /* unblock tHighPri */

        taskResume (highPriId); /* unblock tHighPri */

        /* build message and send it */

        msg.childLoopCount = ix;
        msg.buffer = TASK_LOWPRI_TEXT;
        msgQSend (msgQId, (char *) &msg, MSG_SIZE, 0, MSG_PRI_NORMAL);
        taskDelay (60);
    }

    taskResume (windDemoId); /* wake up the windDemo task */
}
}
```

网络是 21 世纪的主旋律之一，网络编程是每个程序员应该掌握的基本技能。

网络是 VxWorks 系统之间以及与其他系统联系的主要途径。VxWorks 实现了与 BSD4.4 TCP/IP 兼容的网络协议栈，并且其实时性较之有很大提高，这使得基于 BSD4.4 UNIX Socket 的应用程序可以很方便地移植到 VxWorks 中去。VxWorks 也与 SUN 微系统公司的网络文件系统兼容。

VxWorks 网络在主机和目标机开发环境之间提供一种无缝连接。远程文件访问允许 VxWorks 任务通过网络访问其他系统上的文件。远程调用允许一个任务唤醒实际上运行在另一台机器上的过程。如果使用目标 shell，就可以在开发主机上使用 rlogin 和 telnet 访问该 shell。

本章介绍 VxWorks 网络组件、网络配置以及 VxWorks 网络编程知识，包括 TCP/IP 基础、Socket 编程接口、客户/服务器编程等内容。

8.1 VxWorks 网络组件

VxWorks 网络结构如图 2.3 所示。在最底层，VxWorks 通常使用以太网作为传输媒介。在长距离连接或使用公共底板（common backplane）共享内存的紧耦合环境中，VxWorks 也能使用串行线作为网络连接。在传输媒介的上一层，VxWorks 使用 TCP/IP 和 UDP/IP 协议，用于 VxWorks 进程与其他主机环境进程之间的传输数据。

在以太网协议之上，VxWorks 提供几种网络工具：



➤ **套接字 (Sockets)**

允许运行在 VxWorks 或其他主机环境下的任务相互通信。

➤ **远程调用 (Remote Procedure Calls)**

允许一个任务唤醒实际上运行在另一台机器上的过程, 调用任务和被调过程可以是运行在 VxWorks 或其他主机开发系统中。

➤ **远程文件访问**

允许 VxWorks 任务通过网络文件系统 (NFS)、远程 shell (RSH)、文件传输协议 (FTP)、TFTP 访问远程主机上的文件。

➤ **文件输出 (File Export)**

允许远程主机通过 NFS 客户端维护 VxWorks dos 文件系统。

➤ **远程命令执行**

允许 VxWorks 任务通过网络调用在主机开发环境上的命令。

VxWorks 支持以下几种物理连接:

➤ **以太网**

以太网是许多 VxWorks 网络操作的媒介。以太网已成为为许多销售商所支持的 10/100M bps 局域网标准。它是许多 VxWorks 应用理想的标准。

➤ **串行线接口协议 (SLIP and CSLIP)**

除了使用以太网之外, VxWorks 网络能够使用通过串行线连接的串行线接口协议 (SLIP) 或者使用压缩头的 SLIP 协议 (CSLIP) 与主机系统通信。使用 SLIP 或 CSLIP 作为网络接口驱动是机器间通过长距离电话线连接或 RS-232 串行线点对点连接使用 TCP/IP 软件的直接方法。

➤ **共享内存网络**

VxWorks 网络可以用于同一个底板的多个处理器间相互通信。在这种方式下, 数据的传递是通过共享内存进行的。这是通过共享网络驱动程序实现的, 这种“网络”驱动在共享内存之上完全实现上层网络协议。使得通过以太网实现的高层网络功能在这种共享内存网络之上也可以使用。



8.2 TCP/IP 协议

在原始以太网和底板传输机制之上，VxWorks 使用网间网协议族（通常称为 TCP/IP）在网络进程间通信。

8.2.1 TCP/IP 简介

TCP/IP 协议最初是为支持 ARPANET（一个美国政府资助的研究性网络）上计算机通信而设计的。ARPANET 提出了一些网络概念，如包交换和协议分层（一个协议使用另一个协议提供的服务）。ARPANET 于 1988 年隐退，但是它的继承者（NSF1 NET 和 Internet）却变得更为丰富，现在我们所熟知的万维网 World Wide Web 就是从 ARPANET 演变过来的。它自身支持 TCP/IP 协议。Unix TM 被广泛应用于 ARPANET，它的第一个网络版本是 4.3 BSD。

正是因为 BSD 这个编程接口非常流行，VxWorks 才选用它作为网络编程接口。VxWorks5.3.1 的网络实现是以 BSD4.3 为模型的，它支持 BSD sockets（及一些扩展）和所有的 TCP/IP 网络。并且有助于应用程序从其他平台的网络程序移植到 VxWorks 平台。VxWorks5.4 的网络实现则是基于 BSD4.4 的。

8.2.2 TCP/IP 协议层

TCP/IP 是 VxWorks 提供的网络间进程通信的主要机制。

TCP/IP 主要包括三个协议：Internet 协议、IP 协议和传输层协议。

Internet 协议，处于 TCP 和 UDP 之上的一组协议专门开发的应用程序。包括 telnet、文件传输协议（FTP）等。

IP 层也称网络层，包括 Internet 协议（IP）、网际控制报文协议（ICMP）和地址识别协议（ARP）。

IP 协议是 TCP/IP 协议族的基础。该协议被设计成互连分组交换通信网，已形成一个网际通信环境。它负责在源主机和目的地主机之间传输来自其较高层软件的称为数据报文的数据块，它在源和目的地之间提供非连结型传递服务。

网际控制报文协议（ICMP），实际上并不是 IP 层部分，但直接同 IP 层一起工作，报告网络上的某些出错情况。允许网际路由器传输差错信息或测试报文。

地址识别协议（ARP）实际上也不是 IP 层部分，它处于 IP 层和数据链路层之间，它是在 32 位 IP 地址和 48 位局域网地址之间执行翻译的协议。



传输层协议，包括传输控制协议。主要有 TCP 和 UDP 两种协议。

IP 协议是一个传输层的协议，其他协议可以用它来传输数据。传输控制协议 (TCP) 是一个可靠的端对端的协议，它用 IP 来传送和接收它自己的包。正如 IP 包有它自己的头一样，TCP 也有它自己的头。TCP 是一个面向连接的协议，两个网络应用程序通过一个虚连接相连，即使它们之间可能隔着很多子网、网关、路由器。TCP 可靠地传送和接收两应用程序间的数据，并保证数据不会丢失。当用 IP 来传输 TCP 包时，IP 包的数据段就是 TCP 包。每一个通信主机的 IP 层负责传送和接收 IP 包。

用户数据报协议 (UDP) 也用 IP 层来传输它的包，不像 TCP，UDP 不是一个可靠的协议，但它提供了一种数据报服务。

有多个协议可以使用 IP 层，接收 IP 包的时候必需知道该 IP 包中的数据是哪个上层协议的，因此 IP 包头中有一个字节包含着协议标识符。例如，当 TCP 请求 IP 层传输一个 IP 包时，IP 包的包头中用标识符指明该包包含一个 TCP 包，IP 接收层用该标识符决定由哪一协议来接收数据，这个例子中是 TCP 层，如图 8.1 所示。

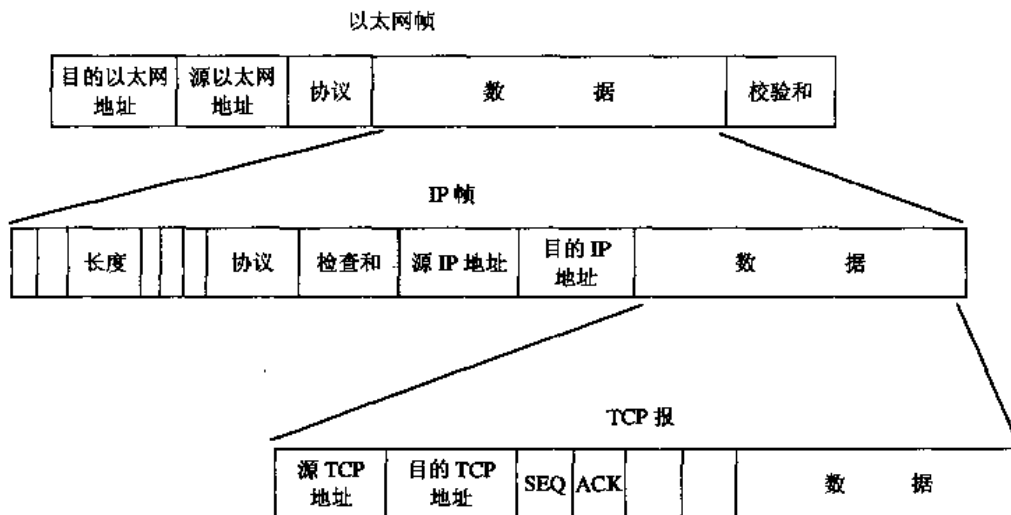


图 8.1 TCP/IP 协议层

当应用程序通过 TCP/IP 进行通信时，它们不仅要指定目标的 IP 地址，而且还要指定应用的端口地址。一个端口地址唯一地标识一个应用，标准的网络应用使用标准的端口地址，如 Web 服务使用 80 端口。这些已登记的端口地址可在有关资料中查到。

这一层的协议不仅仅是 TCP、UDP 和 IP。IP 协议层本身用很多种物理媒介将 IP 包从一个主机传到其他主机。这些媒介可以加入它们自己的协议头。以太网层就是一个例子，但 PPP 和 SLIP 不是这样。一个以太网允许很多主机同时连接到同一根物理电缆。传输中的每一个以太网帧可以被所有主机看见，因此每个以太网设备有唯一的地址。任何传送给该地址的以太网帧被有该地址的以太网设备接收，而其他主机则忽略该帧。这个唯一的地址内置



于每一以太网设备中，通常是在网卡出厂时就写在 SRAM 中了。以太网地址有 6 个字节长，如：08-00-2b-00-49-A4。一些以太网地址是保留给多点传送用的，送往这些地址的以太网帧将被网上所有的主机接收。

以太网帧可以携带很多种协议（作为数据），如 IP 包，并且也包括它们头中的协议标识符。这使得以太网层能正确地接收 IP 包并将它们传给 IP 层。

为了能通过像以太网这样的多连接协议传送 IP 包，IP 层必须找到每一 IP 主机的以太网地址。IP 地址仅仅是一个地址概念，以太网设备有它们自身的物理地址。从另一方面说，IP 地址是可以被网络管理员根据需要来分配和再分配的，而网络硬件只对含有它们自己的物理地址或多点传送地址的以太网帧作出响应。

操作系统用地址解析协议（ARP）来允许机器将 IP 地址转变成真正的硬件地址，如以太网地址。如果一个主机想知道某一 IP 地址对应的硬件地址，它就用一个多点传送地址将一个包含了该 IP 地址的 ARP 请求包发给网上所有节点，拥有该 IP 地址的目标主机则响应一个包含物理硬件地址的 ARP 应答。

ARP 不仅仅局限于以太网设备，它能够用来在其他一些物理媒介上解析 IP 地址，如 FDDI。那些不支持 ARP 的网络设备会被标记出来，VxWorks 将不会用 ARP。还有一个提供相反功能的反向地址解析协议（RARP），用来将物理网络地址转变为 IP 地址。这一协议常常被网关用来响应包含远程网络 IP 地址的 ARP 请求。

8.2.3 TCP/IP 与 ISO/OSI 对照

这里先假定读者对 ISO 的 OSI 七层模型已有了一定的了解，再对照前面介绍的 TCP/IP 模型。

表 8.1 TCP/IP 与 ISO/OSI 对照

层号	OSI	TCP/IP
7	应用层	应用层
6	表示层	无
5	会话层	无
4	传输层	传输层
3	网络层	互联网层
2	数据链路层	主机至网络
1	物理层	



ISO 的 OSI 对服务、接口和协议的概念区别十分明了,但它却没有真正的用户群。TCP/IP 模型对服务、接口和协议的概念区别不像 OSI 模型那样明晰,但很实用。TCP/IP 模型分为四层,对应于 OSI 七层模型。

在 TCP/IP 模型中,互联网层是基于无连接互联网络层的分组交换网络。在这一层中主机可以把报文(Packet)发往任何网络,报文独立地传向目标。互联网层定义了报文的格式和协议,这就是 IP 协议族(Internet Protocol)。互联网层的功能是将报文发送到目的地,主要的设计问题是报文路由和避免阻塞。互联网层上面是传输层,该层的主要功能和 OSI 模型的该层一样,主要使源和目的主机之间可以进行会话。该层定义了两个端到端的协议,一个是面向连接的传输控制协议 TCP,另一个是无连接的用户数据报协议 UDP。

TCP/IP 协议模型中没有会话层和表示层。传输层之上是应用层,它包含所有的高层协议,如远程虚拟终端协议 TELNET、文件传输协议 FTP、简单邮件传输协议 SMTP 等。

这些高层协议中常见的如 TELNET 协议,用来允许用户远程登录到另一台机器;FTP 协议用来传输文件;SMTP 协议用来传送 email,常见的服务器端程序有 netscape 等公司制作的程序,也有免费使用的 sendmail 程序;还有域名系统服务 DNS 协议,新闻组传送协议 NNTP,用于 WWW 的超文本传输协议 HTTP 等。

主机到网络这一层,在 TCP/IP 模型中没有详细定义,这里不做介绍。如需要学习更多的网络知识及 TCP/IP 的详细描述,请参考专门的书籍,这里不再深入探讨。

8.2.4 IP 地址

在 IP 网络中,每台机器都有一个 IP 地址和与之相联系的子网掩码。IP 地址是一个 32 位的数字,它唯一地标识这台机器,以一个以太网地址类开始,然后是一个网络标示和主机标示。如果没使用子网,地址掩码根据类号自动设置。

针对不同的网络配置有三类子网:

- A 类:用于由很多主机组成的网络中,支持较少的网络。
- B 类:用于由数目适中的主机组成的网络中,支持数目适中的网络。
- C 类:用于由数目较少的主机组成的网络中,支持数目较多的网络。

这三类网络由 IP 地址的高几位决定如表 8.2。

VxWorks 提供操纵以太网地址的函数。例如,将四点地址转换整数地址的函数,从地址中分别提取网络地址部分和主机地址部分的函数,由网络地址和主机地址生成以太网地址的函数等等,详细信息参见 inetLib 库。



表 8.2 以太网地址范围

类	高几位	默认子网掩码	地址范围
A	0	0xffff0000	0.0.0.0-126.255.255.255
保留			127.0.0.0-127.255.255.255
B	10	0xffff0000	128.0.0.0-191.255.255.255
C	110	0xfffff000	192.0.0.0-23.255.255.255

IP 地址由四个用点分开的数字表示，如 203.58.2.9。这个 IP 地址实际上分成两个部分：网络地址和主机地址，每部分的长度是可以变化的（有好几类 IP 地址）。以 15.32.0.9 为例，网络地址是 15.32，主机地址是 0.9。主机地址又进一步分为子网地址和主机地址。还是以 15.32.0.9 为例，子网地址是 15.32.0，主机地址是 15.32.0.9。这样的子划分可以允许某部门划分他们自己的子网络。例如，如果 15.32 是某个公司的网络地址，则 15.32.0 可能是子网 0，15.32.1 可能是子网 1。这些子网可以是分别建立的，可能租用电话线或用微波进行相互间通信。IP 地址由网络管理员分配，用 IP 子网可以很好地管理网络。IP 子网的管理员可以自由分配子网内的 IP 地址。

8.2.5 路由

同一子网内的主机之间可以直接发送 IP 包，而其他的 IP 包将被送到一个特定的主机：网关。网关（或路由器）是用来连接多个 IP 子网的，它们会转发从子网内来的 IP 包。例如，如果子网 90.0.0.0 和 91.0.0.0 之间通过一个网关相连，那么任何从子网 90.0.0.0 发往子网 91.0.0.0 的包必须由网关指引，网关可以帮这些包找到正确的路线。本地主机建立路由表用以 IP 包找到正确的机器。每一个目的 IP 都有一个条目在路由表中，用以告诉 VxWorks 将 IP 包送到哪一台主机。这些路由表是随网络的拓扑结构变化而动态变化的，如图 8.2 所示。

VxWorks 提供了路由表访问函数 `routeAdd()` 和 `routeDelete()`。

```
/* 通过 90.0.0.2 发送到网络 91.0.0.0 */
routeAdd ("91.0.0.0", "90.0.0.2");
/* 删除通过网关 90.0.0.2 到节点 91.0.0.51 的路由 */
routeDelete ("91.0.0.51", "90.0.0.2");
```

另一个和路由有关的函数 `routeNetAdd()`除了将目的地址作为网络对待外，其作用等价于 `routeAdd()`。



表 8.2 以太网地址范围

类	高几位	默认子网掩码	地址范围
A	0	0xffff0000	0.0.0.0-126.255.255.255
保留			127.0.0.0-127.255.255.255
B	10	0xffff0000	128.0.0.0-191.255.255.255
C	110	0xfffff000	192.0.0.0-23.255.255.255

IP 地址由四个用点分开的数字表示，如 203.58.2.9。这个 IP 地址实际上分成两个部分：网络地址和主机地址，每部分的长度是可以变化的（有好几类 IP 地址）。以 15.32.0.9 为例，网络地址是 15.32，主机地址是 0.9。主机地址又进一步分为子网地址和主机地址。还是以 15.32.0.9 为例，子网地址是 15.32.0，主机地址是 15.32.0.9。这样的子划分可以允许某部门划分他们自己的子网络。例如，如果 15.32 是某个公司的网络地址，则 15.32.0 可能是子网 0，15.32.1 可能是子网 1。这些子网可以是分别建立的，可能租用电话线或用微波进行相互间通信。IP 地址由网络管理员分配，用 IP 子网可以很好地管理网络。IP 子网的管理员可以自由分配子网内的 IP 地址。

8.2.5 路由

同一子网内的主机之间可以直接发送 IP 包，而其他的 IP 包将被送到一个特定的主机：网关。网关（或路由器）是用来连接多个 IP 子网的，它们会转发从子网内来的 IP 包。例如，如果子网 90.0.0.0 和 91.0.0.0 之间通过一个网关相连，那么任何从子网 90.0.0.0 发往子网 91.0.0.0 的包必须由网关指引，网关可以帮这些包找到正确的路线。本地主机建立路由表用以 IP 包找到正确的机器。每一个目的 IP 都有一个条目在路由表中，用以告诉 VxWorks 将 IP 包送到哪一台主机。这些路由表是随网络的拓扑结构变化而动态变化的，如图 8.2 所示。

VxWorks 提供了路由表访问函数 `routeAdd()` 和 `routeDelete()`。

```
/* 通过 90.0.0.2 发送到网络 91.0.0.0 */
routeAdd ("91.0.0.0", "90.0.0.2");
/* 删除通过网关 90.0.0.2 到节点 91.0.0.51 的路由 */
routeDelete ("91.0.0.51", "90.0.0.2");
```

另一个和路由有关的函数 `routeNetAdd()`除了将目的地址作为网络对待外，其作用等价于 `routeAdd()`。

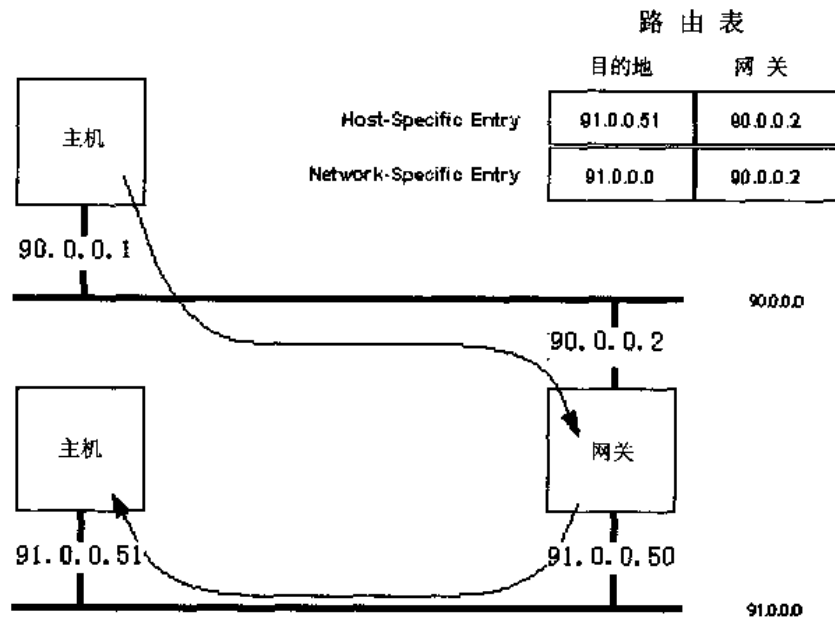


图 8.2 以太网路由

8.2.6 网络字节顺序

一个网络中可能由不同体系结构的 CPU 组成。这些不同体系结构的 CPU 使用的字节顺序不同：有的使用 big-endian 顺序，有的使用 little-endian 顺序。因此为了网络间能够进行数据交换，需要对这些不同的字节顺序进行处理。

VxWorks 中，字节顺序交换通常由网络自身进行处理。如套接字地址或共享信号量 ID 号等。网络字节顺序按 big-endian 编码。

表 8.3 列出了用于在主机与网络间转换长整型和短整型字节顺序的宏。

利用宏实现提高了转换调用效率，并且不受 CPU 体系机构的影响。请参考 h/netinet/in.h 文件。

表 8.3 用于转换网络地址的宏

宏	描 述
htonl	将一个主机长整型数据转换到网络字节顺序
htons	将一个主机短整型数据转换到网络字节顺序
ntohl	将一个网络长整型数据转换到主机字节顺序
ntohs	将一个网络短整型数据转换到主机字节顺序



为了避免宏扩展的副作用，不要直接对表达式使用这些宏。下面的语句作用的结果将使 pBuf 加 4（在 little-endian 体系结构的机器上）：

```
ntohl (*pBuf++); /* UNSAFE */
```

因此应当将增量运算与宏调用分开进行。下面的语句作用的结果仅使 pBuf 加 1，而与机器的体系结构字节顺序无关：

```
pBuf++;  
ntohl (*pBuf);
```

8.3 套接字基础

8.3.1 基本概念

网络编程有两种主要的编程接口，一种是 Berkeley UNIX (BSD UNIX) 的 socket 编程接口，另一种是 AT&T 的 TLI 接口（用于 UNIX SYSV）。VxWorks 实现了与 BSD4.4 TCP/IP 兼容的 socket 编程接口。

TCP/IP 参考模型是目前最为成熟的网络参考模型。它为今天网络技术的迅速发展立下汗马功劳。20 世纪 80 年代早期，远景规划局（Advanced Research Projects Agency, ARPA）资助加利福尼亚大学的伯克利分校的一个研究组，将 TCP/IP 第一次实现在 UNIX 系统上，这就是广为人知的套接字（socket）接口。目前很多操作系统都实现了套接字编程接口，这样套接字接口就被广泛使用，到现在已成为既成事实的标准。

网络通信的基石是套接字，一个套接字是通信的一端。在这一端上你可以找到与其对应的一个名字。一个正在被使用的套接字都有它的类型和与其相关的任务。

套接字可以根据通信性质分类：这种性质对于用户是可见的。应用程序一般仅在同一类的套接字间通信。不过只要底层的通信协议允许，不同类型的套接字间也照样可以通信。

VxWorks 用户目前可以使用两种套接字，即流套接字和数据报套接字。流套接字提供了双向的，有序的，无重复并且无记录边界的数据流服务。数据报套接字支持双向的数据流，但并不保证是可靠，有序，无重复的。也就是说，一个从数据报套接字接收信息的任务有可能发现信息重复了，或者和发出时的顺序不同。数据报套接字的一个重要特点是它保留了记录边界。对于这一特点，数据报套接字采用了与现在许多包交换网络（例如以太网）非常类似的模型。



8.3.2 socket 描述符

前面已经提到过，在 VxWorks 中，任务要对文件进行操作，一般使用 `open` 调用打开一个文件进行访问，每个任务都有一个文件描述符表，该表中存放打开的文件描述符。用户使用 `open` 等调用得到的文件描述符其实是文件描述符在该表中的索引号，该表项的内容是一个指向文件表的指针。应用程序只要使用该描述符就可以对指定文件进行操作。

同样，`socket` 接口增加了网络通信操作的抽象定义，与文件操作一样，每个打开的 `socket` 都对应一个整数，我们称它为 `socket` 描述符，该整数也是 `socket` 描述符在文件描述符表中的索引值。但 `socket` 描述符在描述符表中的表项并不指向文件表，而是指向一个与该 `socket` 有关的数据结构。

VxWorks 中新增加了一个 `socket` 调用，应用程序可以调用它来新建一个 `socket` 描述符，注意任务用 `open` 只能产生文件描述符，而不能产生 `socket` 描述符。`socket` 调用只能完成建立通信的部分工作，一旦建立了一个 `socket`，应用程序可以使用其他特定的调用来为它添加其他详细信息，以完成建立通信的过程。

8.3.3 客户机/服务器模型

网络编程中最常见的是客户/服务器模式。以该模式编程时，服务端有一个任务（或多个任务）在指定的端口等待客户来连接，服务程序等待客户的连接信息，一旦连接上之后，就可以按设计的数据交换方法和格式进行数据传输。客户端在需要的时刻发出向服务端的连接请求。下面讲述中所用到的调用，将在 8.4 节进行详细的阐述，这里为了便于理解，提到了这些调用及其大致的功能。

使用 `socket` 调用后，仅产生了一个可以使用的 `socket` 描述符，这时还不能进行通信，还要使用其他的调用，以使得 `socket` 所指的结构中使用的信息被填写完。

在使用 TCP 协议时，一般服务端任务先使用 `socket` 调用得到一个描述符，然后使用 `bind` 调用将一个名字与 `socket` 描述符连接起来，对于 Internet 域就是将 Internet 地址联编到 `socket`。

之后，服务端使用 `listen` 调用指出等待服务请求队列的长度。然后就可以使用 `accept` 调用等待客户端发起连接（一般是阻塞等待连接，后面章节会讲到非阻塞的方式），一旦有客户端发出连接，`accept` 返回客户的地址信息，并返回一个新的 `socket` 描述符，该描述符与原先的 `socket` 有相同的特性，这时服务端就可以使用这个新的 `socket` 进行读写操作了。

一般服务端可能在 `accept` 返回后创建一个新的任务进行与客户的通信，父任务则再到 `accept` 调用处等待另一个连接。



客户端任务一般先使用 `socket` 调用得到一个 `socket` 描述符，然后使用 `connect` 向指定的服务器上的指定端口发起连接，一旦连接成功返回，就说明已经建立了与服务器的连接，这时就可以通过 `socket` 描述符进行读写操作了。

使用无连接的 UDP 协议时，服务端任务创建一个 `socket`，接着调用 `recvfrom` 接收客户端的数据报，然后调用 `sendto` 将要返回客户端的消息发送给客户端任务。客户端也要先创建一个 `socket`，再使用 `sendto` 向服务端任务发出请求，使用 `recvfrom` 得到返回的消息。

这一请求/响应的过程可以简单的用图 8.3 表示。

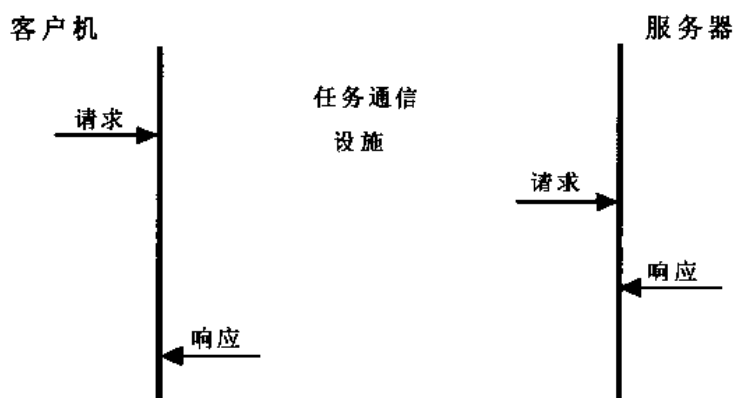


图 8.3 客户机/服务器模型

虽然基于连接的服务是设计客户机/服务器应用程序时的标准，但有些服务也是可以通过数据报套接字提供的。

8.3.4 带外数据

●* 注意

以下对于带外数据（也称为 TCP 紧急数据）的讨论，都是基于 BSD 模型而言的。用户和实现者必须注意，目前有两种互相矛盾的关于 RFC 793 的解释，也就是在这基础上，带外数据这一概念才被引入的。而且 BSD 对于带外数据的实现并没有符合 RFC 1122 定下的主机的要求。为了避免互操作时的问题，应用程序开发者最好不要使用带外数据，除非是与某一既成事实的服务互操作时所必需的。

流套接字的抽象中包括了带外数据这一概念，带外数据是相连的每一对流套接字间一个逻辑上独立的传输通道。带外数据是独立于普通数据传送给用户的，这一抽象要求带外数据



设备必须支持每一时刻至少一个带外数据消息被可靠地传送。这一消息可能包含至少一个字节；并且在任何时刻仅有一个带外数据信息等候发送。对于仅支持带内数据的通信协议来说（例如紧急数据是与普通数据在同一序列中发送的），系统通常把紧急数据从普通数据中分离出来单独存放。这就允许用户可以在顺序接收紧急数据和非顺序接收紧急数据之间作出选择（非顺序接收时可以省去缓存重叠数据的麻烦）。在这种情况下，用户也可以“偷看一眼”紧急数据。

某一个应用程序也可能喜欢线内处理紧急数据，即把其作为普通数据流的一部分。这可以靠设置套接字选项中的 `SO_OOBINLINE` 来实现。在这种情况下，应用程序可能希望确定未读数据中的哪一些是“紧急”的（“紧急”这一术语通常应用于线内带外数据）。应用程序可以使用 `SIOCATMARK socket()` 命令来确定在记号之前是否还有未读入的数据。应用程序可以使用这一记号与其对方进行重新同步。

8.3.5 广播

数据报套接字可以用来向许多系统支持的网络发送广播数据包。要实现这种功能，网络本身必须支持广播功能，因为系统软件并不提供对广播功能的任何模拟。广播信息将会给网络造成极重的负担，因为它们要求网络上的每台主机都为它们服务，所以发送广播数据包的能力被限制于那些用显式标记了允许广播的套接字中。广播通常是为了如下两个原因而使用的：

- ◆ 一个应用程序希望在本地网络中找到一个资源，而应用程序对该资源的地址又没有任何先前的知识。
- ◆ 一些重要的功能，例如路由要求把它们的信息发送给所有可以找到的邻机。

被广播信息的目的地址取决于这一信息将在何种网络上广播。Internet 域中支持一个速记地址用于广播—`INADDR_BROADCAST`。由于使用广播以前必须捆绑一个数据报套接字，所以所有收到的广播消息都带有发送者的地址和端口。

某些类型的网络支持多种广播的概念。例如 IEEE802.5 令牌环结构便支持链接层广播指示，它用来控制广播数据是否通过桥接器发送。

8.3.6 字节顺序

Intel 处理器的字节顺序是和 DEC VAX 处理器的字节顺序一致的。因此它与 68000 型处理器以及 Internet 的顺序是不同的，所以用户在使用时要特别小心以保证正确的顺序。

任何从 Sockets 函数对 IP 地址和端口号的引用和传送给 Sockets 函数的 IP 地址和端口号均是按照网络顺序组织的，这也包括了 `sockaddr_in` 结构这一数据类型中的 IP 地址域和端口域（但不包括 `sin_family` 域）。



考虑到一个应用程序通常用与“时间”服务对应的端口来和服务器连接，而服务器提供某种机制来通知用户使用另一端口。因此 `gethostname()` 函数返回的端口号已经是网络顺序了，可以直接用来组成一个地址，而不需要进行转换。然而如果用户输入一个数，而且指定使用这一端口号，应用程序则必须在使用它建立地址以前，把它从主机顺序转换成网络顺序（使用 `htons()` 函数）。相应地，如果应用程序希望显示包含于某一地址中的端口号（例如从 `getpeername()` 函数中返回的），这一端口号就必须在被显示前从网络顺序转换到主机顺序（使用 `ntohs()` 函数），参考表 8.3。

由于 Intel 处理器和 Internet 的字节顺序是不同的，上述的转换是无法避免的，应用程序的编写者应该使用作为 Sockets API 一部分的标准的转换函数，而不要使用自己的转换函数代码。因为将来的 Sockets 实现有可能在主机字节顺序与网络字节顺序相同的机器上运行。因此只有使用标准的转换函数的应用程序是可移植的。

8.3.7 套接字属性选项

Sockets 规范支持的套接字属性选项都列在对 `setsockopt()` 函数和 `getsockopt()` 函数的叙述中。任何一个 Sockets 实现必须能够识别所有这些属性选项，并且对每一个属性选项都返回合理的数值。每一个属性选项的默认值列在表 8.4 中。

表 8.4 套接字属性选项

选项	类型	含义	默认值	注意事项
SO_BROADCAST	BOOL	套接字被设置为可以	FALSE	发送广播数据
SO_DEBUG	BOOL	允许 Debug	FALSE	
SO_ERROR	int	得到并且清除错误状态	0	
SO_KEEPALIVE	BOOL	活跃信息正在被发送	FALSE	
SO_LINGER	struct	返回目前的 linger 信息		
SO_OOBINLINE	BOOL	带外数据正在普通数据流	FALSE	
SO_RCVBUF	int	接收缓冲区大小	决定于实现	
SO_REUSEADDR	BOOL	该套接字捆绑的地址	FALSE	
SO_SNDBUF	int	发送缓冲区大小	决定于实现	
TCP_NODELAY	BOOL	禁止采用 Nagle	决定于实现	



8.4 Socket 编程接口

这一节详细介绍了一些常用的系统调用，有些调用在上一节已经简单地提到过。在介绍中对于各种调用主要以 TCP/IP 为基础，对其他情况只稍微涉及，如果读者对网络的基本概念还不很熟悉，可以找网络的专门书籍进行了解。

8.4.1 socket 系统调用

调用 `socket` 用来建立一个通信的端点。该调用的声明格式如下：

```
int socket (  
    int domain,      /* 地址簇, 例如 AF_INET */  
    int type,        /* SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW */  
    int protocol     /* socket protocol (usually 0) */  
)
```

`socket` 调用建立一个通信端点并返回一个 `socket` 描述符。

在调用中参数 `domain` 用来指定发生通信的域，是用来选择要使用的协议族的参数。这些协议族在头文件 `<sys/socket.h>` 中定义。参考联机手册来编程。本书只讲述 TCP/IP 协议，所以我们只用到 `AF_INET`。

参数 `type` 用来指明通信的类型，现在使用的定义有如下几种：

- **SOCK_STREAM**：提供顺序、可靠和基于字节流的双向连接。支持带外数据（out-of-band）传输，对于带外数据，要求发送方在发送缓冲中的数据之前发送带外数据，接收方在处理缓冲之前处理带外数据。
- **SOCK_DGRAM**：支持数据报通信（无连接、不可靠、固定的消息最大长度）。
- **SOCK_SEQPACKET**：提供顺序的、可靠的、基于固定最大长度数据报的有传输路径的双向连接。一个接收者在每次 `recv` 系统调用时必须读入一个完整的报文（packet）。
- **SOCK_RAW**：提供原始网络协议访问。
- **SOCK_RDM**：提供一个可靠的数据报层，但不保证报文到达的顺序。

注意，以上的类型并不是对所有协议族都有效。

参数 `protocol` 用来指定在 `socket` 上使用的特定协议。一般来说，可能对一个通信域只有一个特定的协议支持特定的 `socket` 类型。例如，在 `AF_INET` 域中，类型为 `SOCK_STREAM` 的 `socket` 支持 TCP 协议，类型为 `SOCK_DGRAM` 的 `socket` 支持 UDP 协议。然而，可能在



给定的协议族中有多个协议存在，它们可以支持指定的 socket 类型，这时，就需要 protocol 参数来指定。该参数的指定与通信域有关。

socket 调用中，SOCK_STREAM 类型是全双工的字节流。一个流类型的 socket 必须连接以后才能传递数据。与另一个 socket 的连接由 connect 调用产生的。连接后就可以使用 send 和 recv 调用来传输数据。当一次会话关闭后，带外数据仍将被传送。

实现 SOCK_STREAM 的通信协议可以保证传输的数据不丢失和不重复。如果一个节点缓冲中的一块数据在一段时间内不能成功发送，那么就认为该连接已经断开，并且给传输数据的调用返回-1，设置错误代码 ETIMEOUT。可以使用协议选项强制周期性的传输一个特定信号，以使得 socket 连接保持活跃，这时如果超出一定时间 socket 连接没有响应，就产生错误。如果一个任务在一个断开连接的流上发送或接收数据，则产生一个 SIGPIPE 信号。

SOCK_SEQPACKET 类型与 SOCK_STREAM 基本相同，唯一的区别是 recv 调用只有在所需数据数量满足时才返回，其余到达的报文 (packet) 将被丢弃。

SOCK_DGRAM 和 SOCK_RAW 类型允许使用 send 调用发送数据报。一般可以使用 recvfrom 调用接收数据报。这些将在后续小节中进行介绍。该调用成功时返回 socket 描述符，否则返回-1。

8.4.2 connect 调用

调用 connect 的作用是在一个指定的 socket 上建立一个连接。

该调用声明的格式如下：

```
STATUS connect (
    int          s,           /* socket 描述符*/
    struct sockaddr * name,   /* 连接的套接字地址*/
    int          namelen     /* 名字的字节长度*/
)
```

第一个参数 s 是一个打开的 socket 描述符。如果该 socket 的类型是 SOCK_DGRAM，该调用指出与该 socket 绑定的节点，指定的地址是数据报发送到的地址。这时，connect 调用并不真正建立连接，仅仅是让系统知道写到该 socket 的数据报将发送到何地，而且只有该地址发来的数据才被该 socket 接收。

在使用 UDP 时，使用 connect 的好处是不必为每次发送和接收指定地址，可以使用 send 和 recv 调用。如果 socket 是 SOCK_STREAM 类型，该调用将试图与另一个指定名字的 socket 进行连接，一般来说指定的名字是由该通信域解释的一个地址。

通常一个流类型的 socket 只能使用一次 connect，而对于数据报类型的 socket 可以使用



多次 connect 调用来改变 socket 的绑定。数据报类型的 socket 可以通过绑定一个非法地址(如空地址)来取消绑定。

第二个参数是一个地址结构,指出了服务端的名字。该结构如下:

```
struct sockaddr {
    u_char    sa_len;           /* 总长度 */
    u_char    sa_family;       /* 地址簇 */
    char      sa_data[14];     /* 实际长度; 地址值 */
};
typedef struct sockaddr SOCKADDR;
```

第一个字段是整个结构的大小。第二个字段是地址族,指出通信域的地址表示格式,对于 internet 域的地址族为 AF_INET,一般常用的常量有:

- AF_LOCAL: 本地主机(类似管道)。
- AF_UNIX: 同 AF_LOCAL, 该常量的定义是为了以前兼容。
- AF_INET: Internet 地址,用于 TCP、UDP 等。
- AF_NS: XEROX 网络系统地址。
- AF_ISO: ISO 协议地址。
- AF_OSI: 同上。
- AF_IPX: Novell 网络地址。
- AF_DECnet: DEC 网络地址。
- AF_APPLETALK: Apple Talk 网络地址。
- AF_ROUTE: 内部路由协议地址。

第三个字段为实际地址。

对于 internet 域的地址(AF_INET)表示,我们常用的是 sockaddr_in 结构,它定义在头文件<netinet/in.h>中,该结构如下所示:

```
struct sockaddr_in {
    u_char    sin_len;
    u_char    sin_family;
    u_short   sin_port;
    struct    in_addr sin_addr;
    char      sin_zero[8];
};
```

该结构的前两个字段与 sockaddr 结构中的含义相同,第三个字段是远程要连接的服务端的端口。第四个字段是一个存放 IP 地址的结构,该结构如下:



```

struct in_addr {
    u_long s_addr;
};

```

对于 IPv4 的 IP 地址，刚好是四个字节，存放在一个无符号整型变量中。例如一个 IP 地址通常的写法是 162.27.8.98，在存放时按整型存放。在存放时，不要简单的将它按四个字节从高到低存在 int 类型的 s_addr 中，因为整型变量的字节存放是高位在前还是低位在前依赖于机器及系统的实现，为了程序的可移植性，最好使用下面的宏定义函数进行格式转换，以保证程序的可移植性。

```

htonl();
htons();
ntohl();
ntohs();

```

其中 htonl 函数将长整型的主机字节顺序的 hostlong 转换为网络字节顺序，htons 函数将 short int 类型的 hostshort 转换为网络字节顺序。其余两个函数刚好相反，可以从函数名看出各自的功能。也可以由函数 inet_addr 函数将字符串形式的以点分隔的 IP 地址转换为四字节地址，或由 gethostbyname、gethostbyaddr 函数得到 hostent 结构的地址，这些函数将在后面介绍。

connect 调用的第三个参数是指出地址的长度。当 connect 调用成功返回时，返回值为 0，否则返回-1。

8.4.3 close 调用

在网络程序中使用 close 调用的方法与一般文件基本相同，声明格式：

```

STATUS close (
    int fd /* socket 描述符*/
)

```

在使用 TCP 协议的程序中，当客户端或服务端使用完一个 socket 时，可以使用 close 来关闭该 socket。

close 调用的参数为 socket 描述符。在 TCP 协议下，发出该调用将断开连接，如果该描述符有多个引用（与打开文件相同），则将该 socket 的引用数减一，如果引用数减为 0，则释放该 socket。如果被关闭的 socket 具有保证可靠性的类型（如 SOCK_STREAM），那么核心将继续试图传送缓冲中的数据，直到超时为止。如果要求不再发送缓冲中的数据，那么可



以使用 `shutdown` 调用。

8.4.4 send 和 sendto 调用

系统调用 `send` 和 `sendto` 用来从一个 `socket` 发送一个消息。

该调用的声明格式如下：

```
int send (
    int s,                /* 一个用于标识已连接套接字的描述字 */
    char * buf,           /* 包含待发送数据的缓冲区 */
    int bufLen,           /* 缓冲区中数据的长度 */
    int flags              /* 调用执行方式 */
)

int sendto (
    int s,                /* 一个用于标识已连接套接字的描述字 */
    caddr_t buf,          /* 包含待发送数据的缓冲区 */
    int bufLen,           /* 缓冲区中数据的长度 */
    int flags,            /* 调用执行方式 */
    struct sockaddr * to, /* 指向目的套接字的地址 */
    int tolen              /* to 所指地址的长度 */
)
```

调用 `send` 和 `sendto` 用来发送一个消息到另一个 `socket`。

`send` 调用只用于处于连接状态的 `socket`，而 `sendto` 不需要 `socket` 处于连接状态，在 `sendto` 调用中要指出消息的目的地。参数 `s` 是一个 `socket` 描述符，参数 `msg` 指向用户空间的一片缓冲空间，该处存放要发送的消息，消息的长度由参数 `len` 指出。

在 `sendto` 调用中，参数 `to` 为 `sockaddr_in` 结构，用于指出在 `AF_INET` 域中目的地的地址，地址结构的大小由参数 `tolen` 指出。当要发送的消息长度太长而不能发送时，返回错误代码 `EMSGSIZE`，并且不发送该消息。当要传送的消息长度大于该 `socket` 的发送缓冲的剩余空间大小时，`send` 一般会被阻塞，除非 `socket` 被设置为非阻塞模式。

在 `socket` 被设置为非阻塞模式，且传送消息长度大于发送缓冲的剩余空间大小时，该调用（`send` 或 `sendto`）返回 `EAGAIN` 错误代码。

在这两个调用中，第四个参数 `flags` 可以包括一个或多个常量值（“或”运算），这些常量的定义如下：

```
#define MSG_OOB      0x1      /* process out-of-band data */
#define MSG_DONTROUTE 0x4     /* bypass routing, use direct interface */
```



```
#define MSG_WAITALL 0x40      /* wait for full request or error */
#define MSG_DONTWAIT 0x80    /* this message should be nonblocking */
```

这里解释一下这些常量的含义。`MSG_OOB`，在指定的 `socket` 上发送带外数据，该 `socket` 类型必须支持带外数据（如 `SOCK_STREAM`）。`MSG_DONTROUTE`，避开通常的路由表查询，直接发送报文到目标地址描述的接口。这个 `flag` 通常用于诊断程序或路由选择程序。`MSG_DONTWAIT`，允许非阻塞操作，如果对于不设置该标志则会发生阻塞的情况，在设置该标志后会返回 `EAGAIN` 错误代码。

8.4.5 `recv` 和 `recvfrom` 调用

系统调用 `recv` 和 `recvfrom` 用来从一个 `socket` 接收消息。

该调用的声明格式如下：

```
int recv (
    int s,                /* socket to receive data from */
    char * buf,           /* buffer to send data to */
    int bufLen,          /* length of buffer */
    int flags             /* flags to underlying protocols */
)

int recvfrom (
    int s,                /* socket to receive from */
    char * buf,           /* pointer to data buffer */
    int bufLen,          /* length of buffer */
    int flags,           /* flags to underlying protocols */
    struct sockaddr * from, /* where to copy sender's addr */
    int * pFromLen       /* value/result length of from */
)
```

这两个系统调用中，参数 `s` 为要接收数据的 `socket` 描述符。参数 `buf` 指向接收数据后存放的用户缓冲的地址。参数 `len` 指出用户缓冲的大小。

系统调用 `recvfrom` 来从一个 `socket` 接收消息，不管这个 `socket` 是否面向连接。如果参数 `from` 非空，并且该 `socket` 不是面向连接的，则 `recvfrom` 调用返回时，参数 `from` 中存放的是消息的源地址。参数 `fromlen` 也是一个返回值参数，调用前 `fromlen` 的初值是为参数 `from` 分配的缓存的大小，调用后指出 `from` 的实际大小。

调用 `recv` 一般仅用于一个已经连接的 `socket`，该调用等同于 `recvfrom` 中参数 `from` 为空指针 `NULL`。这两个调用成功时，返回消息的实际长度。如果一个消息太长，而缓冲不能完全放下，则剩余部分可能被丢弃（这要根据接收数据的 `socket` 的类型而定）。如果在指定 `socket`



没有消息到达,这两个调用会被阻塞等待消息。但如果 socket 被设置为非阻塞方式(参见 fcntl 调用),则调用返回-1。

一般这两个调用接收到数据就返回,而不等待到满足要求接收的数据数量(len)才返回。参数 flags 可以是以下常量中的一个或由下面常量中的多个进行“或”运算得到。

- ◆ MSG_OOB, 该标志请求接收带外数据。
- ◆ MSG_PEEK, 该标志使得接收操作返回接收队列头部的数据,并且不将数据从队列中移出,因此,随后的接收调用会返回相同的数据。
- ◆ MSG_WAITALL, 该标志请求操作被阻塞,直到所有请求的数据数量被满足为止。但是当接到一个 signal、发送错误或连接被断开时,设置该标志的接收调用不管数据是否满足也将返回。
- ◆ MSG_ERRQUEUE, 从错误队列中接收报文。这些定义与平台有关,前三个一般平台上都有定义。

这两个调用成功时,返回接收到的字符数,错误时返回-1。

8.4.6 shutdown 调用

该调用将关闭一个全双工连接的一端。

该调用声明的格式如下:

```
STATUS shutdown (  
    int s, /* socket to shut down */  
    int how /* 0 = receives disallowed 1 = sends disallowed 2 = sends and */  
           /* disallowed */  
)
```

shutdown 调用将导致 socket 描述符 s 所指的全双工 socket 连接被部分或全部关闭。如果参数 how 指定为 0,则被该调用关闭的 socket 将禁止接收数据。如果参数 how 为 1,则禁止继续发送数据。如果 how 为 2,则禁止在该 socket 上接收或发送数据。

该调用成功时返回 0,否则返回-1。

8.4.7 bind 调用

该调用将一个名字绑定到一个 socket 上。

该调用声明格式如下:

```
STATUS bind (  
    int s, /* socket descriptor */
```



```

struct sockaddr * name, /* name to be bound */
int          namelen /* length of name */
)

```

`bind` 调用为一个未命名的 `socket` 分配一个名字。换句话说，`bind` 调用是给套接字 `s` 赋予本地的地址 `my_addr`，这个 `my_addr` 地址的长度为 `addrlen`。

对于不同通信域的地址联编的规则是不同的，这里还是以 `internet` 域为主。如果想让任务侦听任何一个本地接口的报文，那么将地址结构 `sockaddr_in` 中的成员 `sin_addr` 设置为 `INADDR_ANY`。IP 协议的 `socket` 可以绑定到一个广播地址或组播地址。这里要注意，要侦听的端口 (`sin_port`) 的选择。

该调用成功时，返回值为 0，否则为 -1。

8.4.8 listen 调用

该调用在一个 `socket` 侦听连接，`listen` 调用声明的格式如下：

```

STATUS listen (
    int s, /* socket descriptor */
    int backlog /* number of connections to queue */
);

```

为了接受一个连接请求，首先由 `socket` 调用创建一个 `socket`，而 `listen` 调用指定接受连接的队列长度限制，然后由 `accept` 调用来接受连接请求。

该调用只能用在 `SOCK_STREAM` 或 `SOCK_SEQPACKET` 类型的 `socket` 上。参数 `backlog` 指定队列的最大长度，该队列用来放置等待处理的连接请求。如果服务端的等待队列已满，客户的连接请求将导致客户端收到一个 `ECONNREFUSED` 错误，或者当下层协议支持重新发送请求，该请求被忽略。该调用成功时返回 0，否则返回 -1。设置的错误代码除了 `EBADF` 和 `ENOTSOCK` 外，还可能是 `EOPNOTSUPP`，该错误代码表示指定的 `socket` 的类型不支持 `listen` 操作。

8.4.9 accept 调用

该调用在指定的 `socket` 上接受一个连接请求。

该调用的声明格式如下：

```

int accept
(

```



```
int          s,          /* socket descriptor */
struct sockaddr * addr, /* peer address */
int *       addrilen /* peer address length */
)
```

在 `accept` 中, 参数 `s` 是要接受连接请求的 `socket`, 这个 `socket` 是由 `socket` 调用创建, 由 `bind` 调用绑定到一个地址上, 再经过 `listen` 调用的 `socket`。该调用从请求队列中取出第一个未处理的连接请求, 然后创建一个新的与参数 `s` 所指的 `socket` 有相同属性的 `socket`, 并为这个新的 `socket` 分配一个新的 `socket` 描述符。如果队列中没有连接请求且 `socket` 没有被设置为非阻塞方式, 那么 `accept` 调用被阻塞, 直到有请求为止。如果 `socket` 被设置为非阻塞方式并且队列中没有等待处理的连接请求, `accept` 调用返回一个错误。已经接受连接请求的新 `socket` 不可以再接受其他的连接请求。原先的 `socket` 的参数 `s` 仍然保持打开状态。所以我们编程当中经常在 `accept` 后发起一个子任务与新的 `socket` 通信, 而原先的父任务返回 `accept` 处继续等待新的连接请求。

第二个参数 `addr` 是用来存放返回值的, 在使用 IP 地址时, 使用 `sockaddr_in` 结构, `accept` 返回时, 该结构中存放的是连接实体的地址。

第三个参数 `addrilen` 在调用前用来指出 `addr` 的大小, 在调用后返回精确的 `addr` 的长度。该调用被用于基于连接的 `socket` 类型, 即 `SOCK_STREAM`。

该调用失败时, 返回值为 -1, 否则返回一个非负整数, 该整数就是新的 `socket` 的描述符。

8.4.10 gethostbyname 等函数调用

VxWorks 提供一些有关获得网络主机登记信息的函数, 在 `hostlib` 库中。

- `hostTblInit()` 初始化网络主机表。
- `hostAdd()` 增加一个主机到主机表。
- `hostDelete()` 从主机表中删除一个主机。
- `hostGetByName()` 根据名字在主机表中查找主机。
- `hostGetByAddr()` 根据以太网地址在主机表中查找主机。
- `sethostname()` 设置机器的符号名。
- `gethostname()` 得到机器的符号名。

函数原型:

```
void hostTblInit (void)
STATUS hostAdd (char *hostName, char *hostAddr)
STATUS hostDelete (char *name, char *addr)
```



```
int hostGetByName (char *name)
STATUS hostGetByAddr (int addr, char *name)
int sethostname (char *name, int nameLen)
int gethostname (char *name, int nameLen)
```

函数调用 `hostGetByName` 对给定的主机名返回在主机表中的 IP 地址。函数调用 `hostGetByAddr` 则是根据 IP 地址在主机表查找相应的主机名，填到 `name` 参数中。这两个函数调用成功时，返回 `OK`，否则返回 `ERROR`。

下面介绍几个相关函数用来复制二进制数据。

```
void bzero(void *s, int n);
void bcopy (const void *src, void *dest, int n);
void *memcpy(void *dest, const void *src, size_t n);
```

函数 `bzero` 将参数 `s` 所指空间的前 `n` 个字节，写为 0。

函数 `bcopy` 将参数 `src` 所指空间的前 `n` 个字节复制到参数 `dest` 所指的地方。

函数 `memcpy` 将 `src` 所指的前 `n` 个字节复制到 `dest` 处。

8.4.11 inet_addr 等函数调用

在这一小节中，介绍一些有关 `internet` 地址操作的函数调用，主要介绍 `inet_addr`、`inet_network`、`inet_ntoa`、`inet_makeaddr`、`inet_makeaddr_b`、`inet_lnaof`、`inet_netof` 等函数调用。

这些函数调用的声明格式如下：

```
u_long inet_addr (char *inetString)
int inet_lnaof (int inetAddress)
void inet_makeaddr_b (int netAddr, int hostAddr, struct in_addr *pInetAddress)
struct in_addr inet_makeaddr (int netAddr, int hostAddr)
int inet_netof (struct in_addr inetAddress)
void inet_netof_string (char *inetString, char *netString)
u_long inet_network (char *inetString)
void inet_ntoa_b (struct in_addr inetAddress, char *pString)
char *inet_ntoa (struct in_addr inetAddress)
```

函数调用 `inet_ntoa` 调用将网络字节顺序的 `internet` 主机地址转换成标准的以 “.” 和数字构成的字符串。返回的字符串是一个静态分配的缓冲，会被后续的调用覆盖。

函数调用 `inet_addr` 将参数 `inetString` 所指的主机地址转换成网络字节顺序的二进制数据格式。



函数调用 `inet_network` 可以从地址 `inetString` 中取出主机字节顺序的网络地址。如果参数无效，返回 `ERROR`。

函数调用 `inet_makeaddr` 将参数 `netAddr` 所指的地址和参数 `hostAddr` 所指的在 `net` 网络中的本地地址经过运算组合，得到一个网络字节顺序的 `internet` 主机地址。参数 `netAddr` 和 `hostAddr` 都是按本地主机字节顺序参与运算的。

函数调用 `inet_lnaof` 以本地主机字节顺序返回本地主机在本子网中的地址。该计算根据参数 `inetAddress` 得到。例如：主机地址 `172.26.5.198`，属于一个 B 类地址，所以主机在 `172.26.0.0` 子网中的主机地址为 `0.0.5.198`。有关 `internet` 地址分类以及 `internet` 无类地址，请参考有关网络书籍。

函数调用 `inet_netof` 返回参数 `inetAddress` 所指的 `internet` 地址中的网络号（即上一例中的 `172.26.0.0`），该网络地址按本地主机字节顺序返回。这里还要强调一下的是，应存放的字节顺序是网络字节顺序还是主机字节顺序。最好使用 `htonl` 等系列调用转换字节顺序，以保证程序的可移植性。

8.4.12 getsockopt 和 setsockopt 调用

系统调用 `getsockopt` 和 `setsockopt` 用来读取或设置 `socket` 的任选项。

这两个调用的声明格式如下：

```
STATUS getsockopt (int s, int level, int optname, char *optval, int *optlen)
STATUS setsockopt (int s, int level, int optname, char *optval, int optlen)
```

参数说明：

- ◆ `s`：一个标识套接字的描述字。
- ◆ `level`：选项定义的层次。支持的层次仅有 `SOL_SOCKET` 和 `IPPROTO_TCP`。
- ◆ `optname`：需获取的套接字选项。
- ◆ `optval`：指针，指向存放所获得选项值的缓冲区。
- ◆ `optlen`：指针，指向 `optval` 缓冲区的长度值。

`getsockopt()` 函数用于获取任意类型、任意状态套接字的选项当前值，并把结果存入 `optval`。在不同协议层上存在选项，但往往是在最高的“套接字”层次上，设置选项影响套接字的操作，诸如操作的阻塞与否、包的选径方式、带外数据的传送等。

被选中选项的值放在 `optval` 缓冲区中。`optlen` 所指向的整形数在初始时包含缓冲区的长度，在调用返回时被置为实际值的长度。对 `SO_LINGER` 选项而言，相当于 `linger` 结构的大小，对其他选项来说，是一个整形数的大小。

如果未进行 `setsockopt()` 调用，则 `getsockopt()` 返回系统默认值。

`getsockopt()` 支持下列选项。其中“类型”栏指出了 `optval` 所指向的值。仅有 `TCP_NODELAY` 选项使用了 `IPPROTO_TCP` 层；其余选项均使用 `SOL_SOCKET` 层。



用一个未被支持的选项去调用 `getsockopt()` 将会返回一个错误。

返回值：若无错误发生，`getsockopt()` 返回 OK。否则的话，返回 ERROR。

`setsockopt()` 函数用于任意类型、任意状态套接字的设置选项值。尽管在不同协议层上存在选项，但本函数仅定义了最高的“套接字”层次上的选项。选项影响套接字的操作，诸如加急数据是否在普通数据流中接收，广播数据是否可以从套接字发送等等。

有两种套接字的选项：一种是布尔型选项，允许或禁止一种特性；另一种是整形或结构选项。

表 8.5 `getsockopt()` 支持选项

选 项	类 型	意 义
SO_ACCEPTCONN	BOOL	套接字正在用 <code>listen()</code> 监听
SO_BROADCAST	BOOL	套接字设置为传送广播信息
SO_DEBUG	BOOL	允许调试
SO_DONTLINGER	BOOL	若为真则 <code>SO_LINGER</code> 选项被禁止
SO_DONTROUTE	BOOL	禁止选路
SO_ERROR	int	获取错误状态并清除
SO_KEEPAIVE	BOOL	发送“保持活动”信息
SO_LINGER	struct linger FAR*	返回当前各 <code>linger</code> 选项
SO_OOBINLINE	BOOL	在普通数据流中接收带外数据
SO_RCVBUF	int	接收缓冲区大小
SO_REUSEADDR	BOOL	套接字能和一个已在使用中的地址捆绑
SO_SNDBUF	int	发送缓冲区大小
SO_TYPE	int	套接字类型（如 <code>SOCK_STREAM</code> ）
TCP_NODELAY	BOOL	禁止发送合并的 Nagle 算法

允许一个布尔型选项，则将 `optval` 指向非零整形数；禁止一个选项 `optval` 指向一个等于零的整形数。对于布尔型选项，`optlen` 应等于 `sizeof(int)`；对其他选项，`optval` 指向包含所需选项的整形数或结构，而 `optlen` 则为整形数或结构的长度。`SO_LINGER` 选项用于控制下述情况的行动：套接字上有排队的待发送数据，且 `closesocket()` 调用已执行。参见 `closesocket()` 函数中关于 `SO_LINGER` 选项对 `closesocket()` 语义的影响。应用程序通过创建一个 `linger` 结构来设置相应的操作特性：



```

struct linger {
    int l_onoff;
    int l_linger;
};
    
```

为了允许 SO_LINGER，应用程序应将 l_onoff 设为非零，将 l_linger 设为零或需要的超时值（以秒为单位），然后调用 setsockopt()。为了允许 SO_DONTLINGER（亦即禁止 SO_LINGER），l_onoff 应设为零，然后调用 setsockopt()。

默认条件下，一个套接字不能与一个已在使用中的本地地址捆绑（参见 bind()）。但有时需要“重用”地址。因为每一个连接都由本地地址和远端地址的组合唯一确定，所以只要远端地址不同，两个套接字与一个地址捆绑并无大碍。为了通知 VxWorks 套接字实现不要因为一个地址已被一个套接字使用就不让它与另一个套接字捆绑，应用程序可在 bind()调用前先设置 SO_REUSEADDR 选项。

请注意仅在 bind()调用时该选项才被解释；故此无需（但也无害）将一个不会共用地址的套接字设置该选项，或者在 bind()对这个或其他套接字无影响情况下设置或清除这一选项。

一个应用程序可以通过打开 SO_KEEPA L I V E 选项，使得 VxWorks 套接字实现在 TCP 连接情况下允许使用“保持活动”包。一个 VxWorks 套接字实现并不是必需支持“保持活动”，但是如果支持的话，具体的语义将与实现有关，应遵守 RFC1122 “Internet 主机要求一通信层”中第 4.2.3.6 节的规范。如果有关连接由于“保持活动”而失效，则进行中的任何对该套接字的调用都将以错误返回，后续的任何调用也将以错误返回。

表 8.6 setsockopt()支持选项

选 项	类 型	意 义
SO_BROADCAST	BOOL	允许套接字传送广播信息
SO_DEBUG	BOOL	记录调试信息
SO_DONTLINER	BOOL	不要因为数据未发送就阻塞关闭操作;设置本选项相当于将 SO_LINGER 的 l_onoff 元素置为零
SO_DONTROUTE	BOOL	禁止选径;直接传送
SO_KEEPA L I V E	BOOL	发送“保持活动”包
SO_LINGER	struct linger FAR*	如关闭时有未发送数据,则逗留
SO_OOBI N L I N E	BOOL	在常规数据流中接收带外数据
SO_RCVBU F	Int	为接收确定缓冲区大小



续表

选项	类型	意义
SO_REUSEADDR	BOOL	允许套接字和一个已在使用中的地址捆绑 (参见 bind())
SO_SNDBUF	int	指定发送缓冲区大小
TCP_NODELAY	BOOL	禁止发送合并的 Nagle 算法

TCP_NODELAY 选项禁止 Nagle 算法。Nagle 算法通过将未确认的数据存入缓冲区直到蓄足一个包一起发送的方法,来减少主机发送的零碎小数据包的数目。但对于某些应用来说,这种算法将降低系统性能。所以 TCP_NODELAY 可用来将此算法关闭。应用程序编写者只有在确切了解它的效果并确实需要的情况下,才设置 TCP_NODELAY 选项,因为设置后对网络性能有明显的负面影响。TCP_NODELAY 是唯一使用 IPPROTO_TCP 层的选项,其他所有选项都使用 SOL_SOCKET 层。

如果设置了 SO_DEBUG 选项, VxWorks 套接字供应商被鼓励 (但不是必需) 提供输出相应的调试信息。

setsockopt() 支持表 8.6 中的选项。其中“类型”表明 optval 所指数据的类型。

返回值: 若无错误发生, setsockopt() 返回 0。否则的话, 返回 SOCKET_ERROR 错误, 应用程序可通过 WSAGetLastError() 获取相应错误代码。

8.4.13 select 调用

select 调用及宏 FD_CLR、FD_ISSET、FD_SET、FD_ZERO 用于同步 I/O 复用。

该调用和宏定义的声明格式如下:

```
int select (
    int          width,          /* number of bits to examine from 0 */
    fd_set      *pReadFds,      /* read fds */
    fd_set      *pWriteFds,     /* write fds */
    fd_set      *pExceptFds,    /* exception fds (unsupported) */
    struct timeval *pTimeOut     /* max time to wait, NULL = forever */
)
FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ZERO(&fdset)
```

调用 select 用来等待一些描述符改变状态。有三个不相关的描述符集合被监测。列在



参数 `pReadFds` 集合中的描述符监测是否有字符可以从某个描述符读入，列在 `pWriteFds` 集合中的描述符监测是否某个描述符准备好了可以立即写入，列在参数 `pExceptFds` 集合中的描述符监测是否某个描述符有异常出现。当该调用退出时，集合被指向状态改变的描述符。

`FD_CLR`、`FD_SET`、`FD_ZERO` 这三个宏用来操作集合。`FD_ZERO` 用来清空一个集合。`FD_SET` 和 `FD_CLR` 用来从一个集合中增加或删除一个描述符。

在 `select` 调用中，参数 `width` 是三个集合中描述符的最大值。参数 `pTimeout` 是指出 `select` 返回的时间限制。当 `pTimeout` 为 0 时，`select` 调用立即返回。如果参数 `pTimeout` 为空 (`NULL`)，`select` 被阻塞。`select` 调用成功时，返回在描述符集合中包含的描述符值，如果返回 `OK`，表示在参数 `pTimeout` 时间里没有描述符改变状态。当发生错误时，返回 `EERROR` 错误发生后，集合和 `pTimeout` 变为没有定义，所以出错后的值是无效的。

当 `select` 调用用于 `socket` 的非阻塞 `connect` 时要注意，一般当一个 `socket` 描述符为既可读又可写的状态时，表示发生了错误，但也有可能是该连接在执行到 `select` 之前，已经变为可读。这时可以用 `getsockopt` 的 `SO_ERROR` 选项，得到该 `socket` 上的错误代码。如果调用正确并且错误代码为 0，表示连接正常建立了。否则当 `socket` 描述符变为可写时，表示连接正常建立。

该调用用于 I/O 多路复用的情况，比如在一个任务中有多个 `socket` 和终端都需要读入数据，而任务并不知道什么时候会有数据出现在哪个描述符上，这时就可以使用 `select`。

8.5 Socket 的原始方式

VxWorks 系统提供了 `socket` 编程原始方式的接口。在创建 `socket` 时选择类型为 `SOCK_RAW` 就能创建一个原始类型的 `socket`，在该 `socket` 上，程序员可以自己写 `icmp` 头、`tcp` 头等来发送原始报文。

8.5.1 TCP 报文头

在 C 语言中 TCP 头部使用 `tcphdr` 结构，一般描述如下：

```
/*
 * TCP 报文头
 * Per RFC 793, September, 1981.
 */
struct tcphdr {
    u_short    th_sport;    /* source port */
```



```

        u_short    th_dport;        /* destination port */
        tcp_seq    th_seq;          /* sequence number */
        tcp_seq    th_ack;          /* acknowledgement number */
#ifdef USE_BITFIELDS
#undef USE_BITFIELDS
#endif /* USE_BITFIELDS */
#ifdef USE_BITFIELDS
#if _BYTE_ORDER == _LITTLE_ENDIAN
        u_char     th_x2:4,          /* (unused) */
        th_off:4;    /* data offset */
#endif
#if _BYTE_ORDER == _BIG_ENDIAN
        u_char     th_off:4,         /* data offset */
        th_x2:4;    /* (unused) */
#endif
#else
        u_char     th_off_x2;        /* data offset (4msb) unused (4lsb) */
#endif
        u_char     th_flags;
#define TH_FIN    0x01
#define TH_SYN    0x02
#define TH_RST    0x04
#define TH_PUSH   0x08
#define TH_ACK    0x10
#define TH_URG    0x20
        u_short    th_win;          /* window */
        u_short    th_sum;          /* checksum */
        u_short    th_urp;          /* urgent pointer */
};

```

TCP 头部报文格式如图 8.4。

其中：

- ◆ 数据偏移用于标识数据段的开始。
- ◆ 保留段 6 位必须为 0。
- ◆ 标志包括紧急标志 (URG)、确认标志 (ACK)、入栈标志 (PUSH)、重置标志 (RST)、同步标志 (SYN) 和结束标志 (FIN)。
- ◆ 窗口指定接收方愿意接受的字节数。
- ◆ 校验和计算方式为将头与数据的所有 16 位字的补码求和，再求该补码和的补码。
- ◆ 选项长度是可变的。
- ◆ 填充区域随选项长度变化，用于确保长度为整字节的倍数。



源 端 口 (16 位)		目 的 端 口 (16 位)	
序 号 (32 位)			
确 认 号 (32 位)			
数据偏移 (4 位)	保留 (6 位)	标志 (6 位)	窗 口 (16 位)
校 验 和 (16 位)		紧 急 指 针 (16 位)	
选 项 (可变量)		填 充 位	

图 8.4 TCP 头部报文格式

8.5.2 UDP 报文头

UDP 头部结构比较简单，这里只列出它的结构描述：

```

/*
 * Udp 协议头;
 * Per RFC 768, September, 1981.
 */
struct udphdr {
    u_short    uh_sport;        /* source port */
    u_short    uh_dport;        /* destination port */
    short      uh_ulen;         /* udp length */
    u_short    uh_sum;         /* udp checksum */
};
    
```

其中，uh_sport 是发送方任务的源端口号，该成员可以设置为 0；uh_dport 是目的主机接收任务时用的目的端口号；uh_ulen 是 UDP 报文长度，该长度包括 udp 报文头和数据；uh_sum 是校验和，该值的计算方法是将 IP 报头、UDP 报头及数据求 16 位补码和，并对该补码和求 16 位补码。

8.5.3 IP 报文头

ip 头部结构如图 8.5 所示。



版本号 (4 位)	IP 头长度 (4 位)	服务类型 TOS (8 位)	数据包总长度 (16 位)	
标识 (16 位)			标志 (3 位)	分段偏移量 (13 位)
生存时间 TTL (8 位)	传输协议 (8 位)		头校验和 (16 位)	
源地址 (32 位)				
目标地址 (32 位)				
任选项和填充位 (可变量)				

图 8.5 IP 头部报文格式

其中:

- IP 头长度计算所用单位为 32 位字, 常用来计算数据开始偏移量。
- 数据包总长度用字节表示, 包括头的长度, 因此最大长度为 65535 字节。
- 生存时间表示数据被丢失前保存在网络上的时间, 以秒计算。
- 头校验和的算法是取所有 16 位字的 16 位和的补码。
- 选项长度是可变的。

填充区域随选项长度变化, 用于确保长度为整字节的倍数。

在 C 中 IP 头部使用 ip 结构, 一般描述如下:

```

/*
 * Structure of an internet header, naked of options.
 *
 * We declare ip_len and ip_off to be short, rather than u_short
 * pragmatically since otherwise unsigned comparisons can result
 * against negative integers quite easily, and fail in subtle ways.
 */
struct ip {
#ifdef _BYTE_ORDER == _LITTLE_ENDIAN
    u_int ip_hl:4,          /* header length */
          ip_v:4,          /* version */
#endif
#ifdef _BYTE_ORDER == _BIG_ENDIAN
    u_int ip_v:4,          /* version */
          ip_hl:4,        /* header length */
#endif
          ip_tos:8,        /* type of service */
          ip_len:16;       /* total length */
    u_short ip_id;        /* identification */

```



```

        short ip_off;                /* fragment offset field */
#define IP_DF 0x4000                /* dont fragment flag */
#define IP_MF 0x2000                /* more fragments flag */
#define IP_OFFMASK 0x1fff          /* mask for fragmenting bits */
        u_char ip_ttl;              /* time to live */
        u_char ip_p;                /* protocol */
        u_short ip_sum;             /* checksum */
        struct in_addr ip_src, ip_dst; /* source and dest address */
};

```

8.5.4 ICMP

ICMP 提供的差错报告和状态报告服务如下：

- ◆ 生存时间 (TTL) 超时。
- ◆ 参数不可理解。
- ◆ 报文不可达。
- ◆ 用于流控的源队列。
- ◆ Echo 和 Echo 应答。
- ◆ 重定向。
- ◆ 时间戳和时间戳应答。
- ◆ 信息请求和信息请求应答。
- ◆ 地址掩码请求和应答。

下面我们再来看看 ICMP 的报文格式。ICMP 报文是放在 IP 报文中的，其格式如图 8.6 所示。

IP 头部	
ICMP 类型 (8 位)	代码 (8 位)
校验和 (16 位)	
参数 (无参数时, 不使用该字段)	
信息 (可变长)	

图 8.6 ICMP 的报文格式

其中 ICMP 报文类型可以是：

- ◆ 目的地不可达报文 (Destination Unreachable Message)。
- 类型码值为 3，当代码字段分别为以下值时，表示不同含义：



- 0: 网络不可达。
 - 1: 主机不可达。
 - 2: 协议不可达。
 - 3: 端口不可达。
 - 4: 需要分段, 但设置了禁止分段标志。
 - 5: 源路径失败。
 - 6: 目的网络未知。
 - 7: 目的主机未知。
 - 8: 源主机隔离 (不再使用)。
 - 9: 管理性禁止目的网络。
 - 10: 管理性禁止目的主机。
 - 11: 网络无法提供指定的 TOS。
 - 12: 主机无法提供指定的 TOS。
 - 13: 管理性禁止通信。
 - 14: 违反主机优先级。
 - 15: 优先级停止使用。
- 报文格式如图 8.7 所示。

TYPE	CODE	Checksum
Unused		
IP 头+前 64 位原始数据报文		

图 8.7 目的地不可达报文的报文格式

➤ 超时报文 (Time Exceeded Message)

类型码值为 11, 当代码字段分别为以下值时, 表示不同含义:

- 0: 传送中 TTL 超时。
- 1: 分段重组超时。

➤ 源端抑制报文 (Source Quench Message)

类型码值为 4, 代码字段为 0。

➤ 重定向报文 (Redirect Message)

类型码值为 5, 当代码字段分别为以下值时, 表示不同含义:

- 0: 网络重定向。



- 1: 主机重定向。
 - 2: TOS 和网络重定向。
 - 3: TOS 和主机重定向。
- 报文格式如图 8.8 所示。

TYPE	CODE	Checksum
Gateway IP Address		
IP 头+前 64 位原始数据报文		

图 8.8 重定向报文报文格式

➤ 回应请求/应答报文

类型码值为 8，表示 Echo 报文。
 类型码值为 0，表示 Echo 应答报文。
 参数不可理解报文 (Parameter Problem Message)
 类型码值为 12。

➤ 信息请求/应答报文

类型码值为 15，表示信息请求报文。
 类型码值为 16，表示信息应答报文。
 报文格式如图 8.9。

TYPE	CODE	Checksum
Identifier		
Sequence No.		
Informationl		

图 8.9 信息请求/应答报文报文格式

➤ 时间戳/时间戳应答报文

类型码值为 13，表示时间戳请求报文。
 类型码值为 14，表示时间戳应答报文。
 时间戳报文格式如图 8.10。



TYPE	CODE	Checksum
ID	Sequence No.	
Original TimeStamp		
Receive TimeStamp		
Transmit TimeStamp		

图 8.10 时间戳报文格式

下面看一下 ICMP 报文头部结构 `icmphdr`。该结构一般定义如下：

```

/*
 * Structure of an icmp header.
 */
struct icmp {
    u_char    icmp_type;        /* type of message, see below */
    u_char    icmp_code;       /* type sub code */
    u_short   icmp_cksum;      /* ones complement cksum of struct */
    union {
        u_char ih_pptr;        /* ICMP_PARAMPROB */
        struct in_addr ih_gwaddr; /* ICMP_REDIRECT */
        struct ih_idseq ih_idseq;
        int ih_void;

        /* ICMP_UNREACH_NEEDFRAG -- Path MTU Discovery (RFC1191) */
        struct ih_pmtu {
            n_short ipm_void;
            n_short ipm_nextmtu;
        } ih_pmtu;
    } icmp_hun;
#define icmp_pptr    icmp_hun.ih_pptr
#define icmp_gwaddr  icmp_hun.ih_gwaddr
#define icmp_id      icmp_hun.ih_idseq.icd_id
#define icmp_seq     icmp_hun.ih_idseq.icd_seq
#define icmp_void    icmp_hun.ih_void
#define icmp_pnvoid  icmp_hun.ih_pmtu.ipm_void
#define icmp_nextmtu icmp_hun.ih_pmtu.ipm_nextmtu
    union {
        struct id_ts id_ts;
        struct id_ip id_ip;
        u_long    id_mask;
        char id_data[1];
    } icmp_dun;
#define icmp_otime  icmp_dun.id_ts.its_otime
#define icmp_rtime  icmp_dun.id_ts.its_rtime

```



```
#define icmp_ttime icmp_dun.id_ts.its_ttime
#define icmp_ip      icmp_dun.id_ip.idi_ip
#define icmp_mask    icmp_dun.id_mask
#define icmp_data    icmp_dun.id_data
};
```

8.5.5 一个例子

本节以 ping 程序作为例子，说明 SOCK_RAW 的用法。

Ping 使用 ICMP 协议的强制性 ECHO-REQUEST 报文来引发某个主机或网关的 ICMP 协议的 ECHO-RESPONSE 报文。通过检测返回的数据包的质量以及计算数据包所化的时间就可得出网上两个接点之间网络的通信质量。

在工作时，ping 向网上发送 ping 数据包，并计算收发数据包所用的时间，统计数据包的丢失情况。如果由重复的数据包被接收，它就将其计入收发时间中，这样它重复地执行发包、收包、计算的过程，直到发出的数据包达到指定的数目或者被中断。

例 8.1 ping 程序

```
/*
*****
*/
/* vxPing.c - send ICMP ECHO_REPLY packets to a particular network host */

/* Copyright 1984-1997 Wind River Systems, Inc. */

/*
modification history
-----
01i,14nov97,ns removed packetSize from the function xmtMsg
01h,14nov97,ns added bzero in constructIcmpMsg function
01g,14nov97,ns initialized msgNum to 0 , removed u_char pointer from
constructIcmpMsg function
01f,06nov97,mm added copyright.
01e,10Oct97,mm cast arg 1 of taskIdFigure
01d,10Oct97,mm definded transmitEchoMessage, xmtMsg and configureSocket
01c,10Oct97,mm included "netShow.h", "usrLib.h", "hostLib.h", <stdio.h>,
<string.h> and "sockLib.h"
01b,10Oct97,mm changed ip_hl to ip_v_hl
01a,18Feb94,ms extensively modified for VxDemo.
*/

/* INCLUDES */
#include "vxWorks.h"
#include "socket.h"
#include "sockLib.h"
```



```

#include "netinet/in_system.h"
#include "in.h"
#include "netinet/ip.h"
#include "netinet/ip_icmp.h"
#include "taskLib.h"
#include "netShow.h"
#include "usrLib.h"
#include "hostLib.h"
#include <stdio.h>
#include <string.h>
#include "inetLib.h"

#define TASK_PRI 100 /* Priority of spawned tasks */
#define TASK_STACK_SIZE 10000 /* stack size of the spawned tasks */
#define ICMP_HDRLEN sizeof (struct icmp)
#define MY_ICMP_ID 0x1234

/* globals */
LOCAL int msgNum = 0 ;
LOCAL struct sockaddr_in destAddr;
LOCAL int sockFd;
LOCAL char dstHost [20]; /* Name of the remote host to ping */
LOCAL int numReceived = 0 ;
LOCAL int origNumPackets;
LOCAL int numPkts;

/* forward declarations */
LOCAL STATUS startPing();
STATUS rcvPings ();
LOCAL void constructIcmpMsg (struct icmp *);
LOCAL void pingResponse ();
LOCAL u_short inCksm (u_short *);

STATUS transmitEchoMessage();
STATUS xmtMsg(struct icmp* msg);
STATUS configureSocket(char* nameofHost);

/* Needs to be compiled with -DUSE_BITFIELDS flag */

/*****
 * vxPing - send ICMP ECHO_REPLY packets to a particular network host to
 * elicit an ICMP ECHO_RESPONSE from that specified network host
 * and prints "<host> is alive" on the standard output if the host
 * responds, otherwise prints ICMP error messages.
 *
 * When using ping, make sure the host you want to ping is already
 * in the host table. If not, add the host to the host table using
 * hostAdd. If the host you want to ping is in a different network

```



```
*      than your VxWorks system, make sure route is established to that
*      host. If route is not already established, use routeAdd to add
*      the route.
*
* EXAMPLE:
*
*      To run ping, from the VxWorks shell do as follows:
*      -> vxPing ("nameOfTheHost", count)
*
*      where nameOfTheHost is the name of the host you want to ping and
*      count is the number of requests to send ( example: ping ("sevana", 5) )
*/
```

```
STATUS vxPing
(
  char *hostName, /* Name of the host to ping */
  int numPackets /* Number of requests to send */
)

{
  int num_packs;
  int status;

  /* get user's parameters */
  if (hostName == NULL)
  {
    printf ("Must enter a host name to Ping\n") ;
    return (ERROR);
  }

  if (numPackets > 1)
    num_packs = numPackets;
  else
    num_packs = 1;

  numPkts = num_packs;
  origNumPackets = num_packs;
  msgNum = 0;
  /* configure for network communications */
  status = configureSocket (hostName) ;

  if (status == ERROR)
  {
    printf ("error while trying to configure socket\n") ;
    return (ERROR);
  }
}
```



```

if (taskSpawn ("tStartPing", TASK_PRI, 0, TASK_STACK_SIZE,
              (FUNCPTR) startPing, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
{
    perror ("spawning startPing task failed");
    return (ERROR);
}

if (taskSpawn ("tRcvPing", TASK_PRI, 0, TASK_STACK_SIZE,
              (FUNCPTR) rcvPings, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
{
    perror ("spawning rcvPings task failed");
    return (ERROR);
}

return (OK);
}

/*****
* configureSocket - Configures socket for ICMP protocol
*
* RETURNS : OK or ERROR
*
*/

STATUS configureSocket
(
    char *nameOfHost
)
{
    printf ("Name of the host to ping is -> %s\n", nameOfHost);

    bzero ((char *) &destAddr, sizeof (destAddr));
    destAddr.sin_family = AF_INET ;
    if ((destAddr.sin_addr.s_addr = hostGetByName (nameOfHost)) == ERROR)
    {
        if ( (destAddr.sin_addr.s_addr = inet_addr (nameOfHost)) == ERROR)
        {
            perror ("bad host!");
            return (ERROR) ;
        }
    }

    strcpy (dstHost, nameOfHost);

    sockFd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (sockFd < 0)
    {

```



```
        perror ("socket failed in configure_socket");
        return (ERROR);
    }
else
    {
        return (OK) ;
    }
}

/*****
 * startPing - Initiates ping
 *
 * RETURNS : OK or ERROR
 *
 */

LOCAL STATUS startPing()
{
    while (1)
    {
        if (numPkts > 0)
        {
            numPkts--;
            if (transmitEchoMessage() == ERROR)
            {
                printf ("Error in transmitEchoMessage\n");
                return (ERROR);
            }
        }
        else
            break ;
        taskDelay(5) ;
    }
    pingResponse ();
    return (OK);
}

/*****
 * transmitEchoMessage - Fill ICMP header and transmit ECHO_REQUEST datagrams
 *
 * RETURNS : OK or ERROR
 *
 */

STATUS transmitEchoMessage()
```



```

    {
    struct icmp icmpMsg;

    constructIcmpMsg (&icmpMsg);
    if (xmtMsg (&icmpMsg) == ERROR)
        return (ERROR);
    return (OK);
    }

/*****
 * constructIcmpMsg - Fills ICMP header
 *
 */

LOCAL void constructIcmpMsg
(
    struct icmp *icmpMessage
)
{
    bzero ( (char *) icmpMessage, sizeof (struct icmp) );

    icmpMessage->icmp_type = ICMP_ECHO;
    icmpMessage->icmp_code = 0;
    icmpMessage->icmp_cksum = 0;
    icmpMessage->icmp_id = MY_ICMP_ID;
    icmpMessage->icmp_seq = msgNum++;

    icmpMessage->icmp_cksum = inCksm ((u_short *) icmpMessage);
}

/*****
 * xmtMsg - transmit ECHO_REQUEST datagrams
 *
 * RETURNS : OK or ERROR
 *
 */

STATUS xmtMsg
(
    struct icmp *msg
)
{
    int result;

    if ((result = sendto (sockFd, (caddr_t) msg, ICMP_HDRLLEN, 0,
        (struct sockaddr *) &destAddr, sizeof(destAddr))) < 0)

```



```
{
    perror ("sendto error");
    return (ERROR);
}
else
    if (result != sizeof (msg))
        printf ("wrote %s %d bytes, return = %d\n", dstHost,
            (int) ICMP_HDRLEN, result);
return (OK);
}

/*****
* rcvPings - receive ICMP ECHO_RESPONSE from the specified network host
*
*/

STATUS rcvPings()
{
    int      remAddrLen;
    struct sockaddr_in remAddr;
    int      ipHdrLen;
    struct ip      *ipPtr;
    struct icmp    *icmpPtr;
    u_char  rcvpack[4096] ;

    numReceived = 0 ;
    remAddrLen = sizeof(remAddr) ;

    while (1) /* loop until we break out */
    {
        if ((recvfrom (sockFd, rcvpack, sizeof (rcvpack), 0,
            (struct sockaddr *) &remAddr, &remAddrLen)) < 0)
        {
            perror ("recvfrom error");
            return (ERROR);
        }
        else
        {
            ipPtr = (struct ip *) rcvpack ;
            ipHdrLen = (LSB4 (ipPtr->ip_v_hl)) << 2;
            icmpPtr = (struct icmp *) (rcvpack + ipHdrLen);
            if ((icmpPtr->icmp_type == ICMP_ECHOREPLY)
                && (icmpPtr->icmp_id == MY_ICMP_ID))
            {
                numReceived++;
                if (numReceived == origNumPackets)

```



```

        return (OK);
    }
}

/*****
 * pingResponse - prints "<host> is alive" on the standard output if the host
 *                responds, otherwise prints ICMP error messages.
 *
 */

LOCAL void pingResponse()
{
    if (numReceived == origNumPackets)
        printf("\nhost %s is alive\n", dstHost);
    else
    {
        printf("\ntrouble response with %s: sent %d packet%c, received %d packet%c\n\n",
            dstHost, origNumPackets, ((origNumPackets != 1) ? 's' : ' '),
            numReceived, ((numReceived != 1) ? 's' : ' ') );
        icmpstatShow ();
        if (taskDelete (taskIdFigure ((int)"tRcvPing")) == ERROR)
        {
            perror ("Error in taskDelete");
            return;
        }
    }
}

/*****
 * inCksm - compute checksum
 *
 * RETURNS the calculated checksum (ones complement) of the icmp struct
 */

u_short inCksm
(
    register u_short *msg_ptr
)
{
    register long sum;
    register u_short answer;
    int index;

    sum = 0 ;

```



```
for (index = 0 ; index < 4 ; index++)
{
    sum = sum + *msg_ptr ;
    msg_ptr++ ;
}

sum = (sum >> 16) + (sum & 0xffff) ;
sum = sum + (sum >> 16) ;
answer = ~sum ;
return (answer) ;
}

int kbhit()
{
    int bytesRead=0;
    (void) ioctl (STD_IN, FIONREAD, (int) &bytesRead); /**判断有无键按下**/
    return bytesRead;
}

void flushkeybuff()
{
    (void) ioctl (STD_IN, FIOSETOPTIONS,
                 OPT_ECHO | OPT_CRMOD | OPT_TANDEM | OPT_7_BIT);
    /* flush standard input to get rid of any garbage;
     * E.g. the Heurikon HKV2F gets junk in USART if no terminal connected.
     */
    (void) ioctl (STD_IN, FIOFLUSH, 0 /**XXX*/);
}
/*****/
```

网络编程的主要模式是客户 / 服务器编程模式。这一章介绍这些模式，主要是介绍模式的框架和思路，读者应该结合上一章讲述中的各种调用，按照各种模式编制一些小的试验程序，以利于理解。本章最后给出编程实例。

9.1 客户 / 服务器

在客户 / 服务器模式中，我们将请求服务的一方称为客户，将提供某种服务的一方称为服务器。在 Internet 的应用中大多数应用都是比较单一的客户 / 服务器模式。例如 WWW 使用的 HTTP 协议，由提供服务的任务侦听 80 端口，该任务就是服务的提供者，而浏览器就是客户端。还有网络虚拟终端的 telnet 协议，服务端侦听 23 端口，客户端与服务端进行交互。这样的例子很多。

也有一些应用程序，本身请求服务的同时也提供一些服务，如果我们把功能拆分开来看，这种程序也是客户 / 服务器模式。例如，传送电子邮件的 SMTP 协议，接收/发送邮件的 Sendmail 程序，如果从接收邮件的功能来看，该程序就是一个邮件服务器，该程序侦听 25 端口，客户端可以通过客户端程序或使用一些特定的命令将邮件送到服务器端，这时 sendmail 就是一个服务端程序。Sendmail 向其他 SMTP 服务器转发邮件时，可以将它看做是一个客户端程序。

不管怎样，理解了客户 / 服务器模式后怎样运用，就看程序员了。关于 SMTP、FTP、HTTP、telnet 等应用层协议有很多书介绍，这里不作介绍，读者可以查看 RFC 文档或查阅其他书籍。

我们将网络编程模式定为客户 / 服务器模式后，程序就可分为客户端程序和服务端程



序。

在 TCP/IP 协议中，传输层协议包括面向连接的 TCP 和无连接数据报 UDP，这样我们又将编程分为四种，即使用 TCP 或 UDP 的服务端或客户端的不同组合。在前面几章，我们了解了多任务程序的概念及 TaskSpawn 等调用，这里我们又将程序分为多任务并发，单任务循环，单任务并发等模式。下面通过这几种分类我们来看看网络编程中的客户 / 服务器编程模式。

9.2 客户端程序设计

所谓客户端程序，我们主要用来指发出服务请求的程序。在客户端程序设计中要知道服务端的地址，服务所提供的端口号，服务使用的传输层协议：UDP 还是 TCP 协议，是否可以通过广播的形式找到服务端等。下面我们就来看看基本的设计思路。

9.2.1 基本概念

在客户 / 服务器模式中，客户端程序可以通过两种形式找到服务器：

- 通过指定服务器的 IP 地址和端口号来找到服务端。这种使用的比较多。
- 通过广播的形式来找服务端。由于广播的方式比较慢，占用网络带宽也比较大，不太常用。

通过指定服务器的 IP 地址和端口号比较方便和实用。指定服务器的方法也有几种：

- 第一种方法是直接在程序中指定固定的服务器 IP 地址和端口号，这样做的优点是客户端程序比较快，用户不要关心服务器的位置。缺点是服务器的地址改变要重新编译和分发客户端程序。这种缺点有时是不可容忍的。
- 第二种方法将服务器的 IP 地址和端口号通过参数的形式传递给程序，这种方法的优点是客户端程序比较灵活，对于服务器迁移位置不需要重新编译客户端程序，缺点是客户必须知道服务器的位置，每次执行时都要指定服务器位置。
- 第三种方法是在一个配置文件中存放服务器的位置。与这种方法类似的是通过环境变量传递服务器位置，但这就要求用户会配置自己的环境变量。

这里要说明一点，对于一些常用协议的常用端口号，在第二、第三种方法中不需要用户指定，可以由函数得到。当然，可以指定端口号的客户端程序给用户更大的灵活性。在很多实用的程序中都是结合这几种方法进行处理。

知道了服务端的位置，客户端任务就可以打开一个套接字，然后通过连接（对于 TCP 来说）服务器来发送或接收数据。或者直接向指定的服务器发送和接收数据（对于 UDP 来说）。



最简单的情况下，在一次应用结束时关闭套接字，这样客户端程序就结束了。一般情况下，客户端程序设计比服务端程序设计要简单一些。因为服务端程序要考虑并发的情况，即处理多个客户同时请求服务的情况。使用客户端并发程序设计时，效率要好得多。

9.2.2 相关问题

■ 主机名与 IP 的对应

在 Internet 中一个主机位置的识别，可以由 IP 地址确定，但由于 IP 地址难于记忆等原因，大多数情况下都是使用主机名（也称域名）来指定主机的位置，这样使用为服务主机迁移也带来了好处，一台服务主机迁移到其他地方更改了 IP 地址，我们只需将主机名和 IP 在对应的名字服务器上修改一下即可。

在比较早的网络中（如 ARPANET），主机数量不太多，可以直接在 hosts 中指定名字即可，但这样管理并不统一，经常会发生名字之间的冲突。随着 Internet 飞速发展，用这种方法管理主机名字已经不可能了，为了解决这种问题，发明了域名系统（DNS: Domain Name System），用来管理庞大的 Internet 上的主机域名。

DNS 是分级的、基于域的命名机制。为了实现这种机制由许多 Internet 上的名字服务器共同构成一个分布式数据库系统。该系统用来将主机名映射为 IP 地址。这里不讨论如何分级、域的构成、顶级域的概念，读者对这些应该很熟悉。有一个免费的 named 用来做名字服务器，这里也不讨论它的资源纪录。

我们主要来看看名字的解析过程。

前面介绍过，在程序中可以由 `gethostbyname` 函数得到一个指定的主机名的地址等信息。在该调用中，库函数首先调用一个解析器的库过程，如果该名字不是以“.”结束，该过程首先在本地的 hosts 中查找是否有该指定的名字，如果有该名字，则返回该名字在 hosts 中指定的 IP 地址。如果没有该名字或以“.”结束，则向名字服务器查询，如果服务器中有该映射则返回地址，否则名字服务器向它上级名字服务器查询。这里要注意一个主机名是否以“.”结束之间的区别。

例如：在 `gethostbyname` 里使用参数 `abc`，这时查询 hosts 中是否有关于 `abc` 主机名的说明，没有的话，假如这台主机处于 `yyy.org` 域中，则要查询是否有 `abc.yyy.org`，如果没有则得到 `host unknown`。

但如果一个主机名以“.”结束的话，例如 `abc.edu.cn` 则表示该主机处于 `edu.cn` 域，不会是 `abc.edu.cn.yyy.org`。这有点像绝对路径和相对路径的关系。这里介绍该过程是为了使读者有一个概念，一般编程中可直接使用 `gethostbyname` 来得到该映射关系。



■ 常用端口名与端口号的对应

根据 RFC 1700, 常用端口号是从 0 到 1023, 已注册端口号是从 1024 到 49151, 动态或私有的端口号从 49152 到 65535。最新的 IANA 的端口号分配可以从 <http://www.isi.edu/in-notes/iana/assignments/port-numbers> 获得。

在介绍 `getservent` 函数时, 我们已经看到一个 `services` 文件, 该文件纪录了服务的名称与它的公认端口号等参数。例如, 我们常见的服务协议 `telnet` (23)、`ftp` (21)、`SMTP` (25)、`http` (80) 等都有描述。我们可以使用 `getserventbyname` 等调用获得名称与端口号之间映射的关系。

上面讲述的是服务端的端口号, 我们可以将 IP 地址理解为网络层地址, 用来确定网络中主机的位置, 而端口号可以理解为传输层地址。IP 地址和端口号地址对构成了网络通信中唯一确定的一点。

那么客户端程序使用的端口号如何确定? 在一般情况下, 是通过系统为其分配一个新的未使用的端口来进行通信。

■ 字的存放结构

一般在编程时, 要注意存放地址、端口号的变量。这些变量在使用时, 一般是按网络字节顺序, 而一般的整型变量的存放顺序是按主机的字节顺序存放的, 可以使用前面介绍的 `htonl`、`htons`、`ntohl`、`ntohs` 等函数进行转换。

有的系统主机字节顺序与网络字节顺序一样, 在这样的系统中, 这些函数调用是一些空的宏定义。例如在一个套接字上接收一个整型数据, 接收方应该使用 `ntohl` 进行转换。

还要注意的, 如果接收方使用 `recv` 来得到这个整型数据, 一般都存放在一个字符类型的缓冲中, 这时不要将数据类型直接强制转换为整型来赋值给一个整型变量, 应该用 `bcopy` 将缓冲中的数据以二进制的形式复制到整型变量中, 再做一个字节顺序转换。否则就会出现一些程序员常见的问题: 为什么我按字节计算得到的数据是对的, 但赋值后总是不对。

前面介绍了许多调用, 读者要注意调用的参数和返回值, 如果参数或返回值与字节顺序有关, 本书中都有说明。

下面一个小例子, 可以看到一些有关字节顺序转换的调用, 注释处的地址 2222.1.2.211 换成其他主机名或 IP 地址, 这里也可以看到 `gethostbyname` 在使用主机名做参数时, 进行地址解析, 而用 IP 地址做参数时, 不进行查询。

例 9.1 字节顺序转换的调用演示

```

/*****/
#include <vxworks.h>
#include <netdb.h>

```



```

#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int testmain()
{
    struct hostent* host;
    struct in_addr inaddr;
    unsigned long lna;
    host = gethostbyname("2222.1.2.211"); //此处的地址改为其他真实的主机名或IP地址
    printf("%s %d %d\n",host->h_name,host->h_addrtype,host->h_length);
    bcopy(host->h_addr,&inaddr.s_addr, 4);
    printf("%s \n", inet_ntoa(inaddr));
    lna = htonl(inet_lnaof(inaddr));
    bcopy(&lna, &inaddr.s_addr, 4);
    printf("%s \n", inet_ntoa(inaddr));
}
/*****

```

这里一再强调字节顺序是为了程序有良好的可移植性和保证程序的正确性。

9.2.3 TCP 客户端编程

客户端编程分为面向连接的 TCP 和无连接的 UDP，这一小节介绍基于 TCP 的客户端的一般编程结构。

➤ 以 TCP 连接服务器

TCP 协议是一种保证传输可靠，保证顺序的传输层协议。在连接服务器之前，先要申请一个 SOCK_STREAM 类型的套接字，该套接字的通信域应指定为 PF_INET 类型，可以使用下面的例子来获得该套接字：

```

int sockfd;
struct protoent *proth;
...
proth = getprotobyname("tcp");
sockfd = socket(PF_INET, SOCK_STREAM, proth->p_proto);
...

```

这时，我们获得了一个指定通信域为 internet、使用 TCP 协议的套接字。下面应该将该



套接字与服务端连接，在连接前要知道服务端的地址（或主机名）和端口号（或服务名）。在前面这个例子的变量定义中再加入如下定义：

```
struct sockaddr_in sadd;
struct hostent *hostin;
int portnum;           //用来存放端口号，可以在程序中使用常量代替。
char hostname[80];     //程序得到主机名的方法由程序员来处理，这里只为叙述方便定义
```

上面变量定义中的 `portnum` 只是为了下面叙述方便定义的，程序员可以指定为常量，也可通过命令行参数得到。字符串 `hostname` 的定义也是为了叙述方便，怎样得到该字符串由程序员来自自己处理，该变量中存放主机名或 IP 地址。在前面例子的 `getprotobyname` 调用前加入以下代码：

```
bzero(&sadd, sizeof(sadd));
sadd.sin_family = AF_INET;
if (hostin = gethostbyname(hostname)) //这里是gethostbyname调用成功的情况。
bcopy(hostin->h_addr, (char *) &sadd.sin_addr, hostin->h_length);
else {
    //这里是host unknown的情况。
}
sadd.sin_port = htons(portnum);
```

然后在 `socket` 调用后加入 `connect` 连接调用：

```
if (connect(sockfd, (struct sockaddr *) &sadd, sizeof(sadd)) < 0) {
    printf("cannot connect\n");
    exit(1);
}
...
```

这样就完成了使用 TCP 协议的套接字连接过程，下面就可以使用这个套接字和服务端进行通信了。在这个例子中使用了指定端口号的方法，如果程序连接的服务端是一个公认的端口，那么可以将 `sadd.sin_port = htons(portnum);` 一行换成以下代码：（`struct servent *sevp;` 假设定义了该结构变量指针，并假设使用 ftp 服务）即可：

```
sevp = getservbyname("ftp", "tcp");
portnum = sevp->s_port;
```



➤ 使用 TCP 与服务器通信

因为 TCP 是面向连接的协议，在连接建立以后，可以通过 `send`、`recv` 等调用进行通信，就像管道一样。比如用 `send(sockfd, buf, sizeof(buf))` 来发送 `buf` 中的数据或用 `recv(socket, buf, sizeof(buf))` 来接收数据，接收的数据放在 `buf` 中。编程时，要注意调用是否会被阻塞，还有所使用的应用层协议。如果自己开发的应用，应该定义好协议，以利于客户端和服务端程序的编制。对于现有的协议编制客户端最好参考 RFC 文档，了解其通信机制。

➤ 关闭 TCP 连接

在客户端程序结束时，或使用完一个套接字可以应该使用 `close` 来关闭一个套接字。也可以对双工的套接字使用 `shutdown` 调用关闭写或读。

9.2.4 UDP 客户端编程

对于无连接 UDP 的客户端编程与基于 TCP 的程序在结构上稍有不同，这里介绍其基本的编程结构。与 TCP 编程一样，首先要得到服务端的地址及端口号，然后使用套接字调用得到一个套接字，这些操作与前一小节的方法相同。但在套接字调用中套接字类型应指定为 `SOCK_DGRAM`；在 `getprotobyname` 中参数改为 `udp`。如果使用 `getservbyname`，则协议参数改为 `udp`。然后可以使用 `connect`，也可以不使用 `connect` 调用。

➤ 使用 connect 的 UDP

在 UDP 程序中，对于 `SOCK_DGRAM` 类型的套接字，调用 `connect` 并不真正产生连接，而只是填写有关套接字的数据结构，使用 `connect` 调用的好处是，后边程序中不必每次指定地址，可以使用 `recv`、`send` 调用进行通信。

在使用 TCP 的程序中，一个套接字只能使用一次 `connect` 调用，对于已经连接的套接字不能再次使用 `connect` 进行连接，而使用 UDP 协议与此不同，在程序中，可以通过多次 `connect` 调用来多次改变套接字结构中的数据，使得程序既可以使用 `recv`、`send` 等不用每次指定服务端位置的调用，又可以在只使用一个套接字与多个服务端程序通信。

➤ 不使用 connect 的 UDP

不使用 `connect` 调用的 UDP 程序，在通信时，只能使用 `sendto` 和 `recvfrom` 调用，在这些调用中每次必须指出服务端的位置。使用 UDP 协议的程序与使用 TCP 的不同，使用 TCP 协议的程序一旦建立连接，该套接字就只能通过该连接通信。而使用 UDP 协议的程序，在一个套接字中可以与不同的服务端程序通信。



➤ 关闭一个使用的 UDP 的套接字

在程序使用完一个套接字后，与使用 TCP 的程序一样，关闭 (close) 或 shutdown 一个套接字。

9.3 服务器端程序设计

与客户端程序不同，服务端程序一直侦听某个端口，等待请求消息的到来。如果请求消息被认可，则返回应答消息。服务端程序也可分为基于 TCP 协议和 UDP 协议的两种方式。从结构上可划分为循环和并发，这一小节，不从结构上介绍，而是从使用不同协议上进行介绍。

9.3.1 基本概念

从概念上讲，服务端程序一般先创建一个套接字，然后绑定一个本地端口，并在该端口侦听请求消息，一旦有请求到达，就对该请求做出规定的响应。如此循环下去。一般情况下，服务端程序是一个在后台运行的进程，它等待某一事件的发生。

本节最初介绍的概念，只能一般性理解服务端程序的基本工作。如何对待一次服务还没完成时新的请求又到达，这就牵连到程序结构问题，是使用并发的模型还是循环的模型。如何设计应用层协议，应用层协议的设计要涉及底层使用 TCP 协议还是 UDP 协议。下面章节将分别介绍各种情况。

➤ 面向连接和无连接访问

在服务端程序设计中，对要实现的网络服务功能可以设计自己的应用层协议。在设计时，要考虑传输层使用的协议。在 TCP/IP 协议族中，TCP 协议是面向连接的，而 UDP 协议是无连接的。使用 TCP 协议时，我们称服务端为面向连接的服务端，而使用 UDP 时，称为无连接的服务端。使用的协议选择的恰当将简化应用层协议的设计。当然也可以根据需要进行传输层协议。

9.3.2 面向连接的服务端概念

总体上讲，面向连接的服务器编程要相对简单一些，这主要是因为，传输层使用 TCP 协议保证了传输的可靠性、顺序性。使得服务端程序不需要再考虑这些。服务端进程可以接受一个连接请求，并通过该连接进行通信。当建立连接后，TCP 协议保证通信的可靠性，其中包括丢失数据和报文校验错误时的重新传递机制，以及报文到达乱序时的重新排序。TCP



协议还可以告知客户端连接断开，这些都是使用面向连接的 TCP 协议带来的好处。

使用面向连接的服务端也有它的不足，服务端每接受一个连接请求，将会产生一个新的套接字，而在无连接的 UDP 中，一个套接字可以与多个主机通信。这样就带来了问题，因为每个套接字都要占用系统资源，这就又可能耗尽系统资源。比如，当客户端程序请求并建立连接后因为错误或其他原因停止运行，而 TCP 又不向空闲的连接发送数据，这时这些资源就被浪费了而不能回收（这些情况在现在的系统中有所改善），当系统长时间运行，而客户端经常发生这样的问题，就又可能耗尽系统的资源。

由于 TCP 协议要求连接，并且是一个点对点通信的模式，所以不能实现组播和广播的应用。

9.3.3 无连接的服务端概念

对于简单的应用服务，如 echo 和 daytime 等，使用 UDP 比较简洁，但要使用 UDP 作为传输层协议来实现比较复杂的应用时（比如有多种状态，在设计应用层协议时使用了状态机的情况），用 UDP 来实现就过于复杂。因为，UDP 协议不保证传输的可靠性和顺序。这样在服务端程序设计中，就要考虑报文的顺序、错误校验等问题。一般情况下，报文丢失由客户端程序负责重新传输。这样，客户端还要考虑多长时间认为是报文丢失。这给应用又带来了新的问题，因为在局域网中传输时间短，而广域网中传输时间长，为判断报文丢失带来了一定的复杂度。

当然，使用 UDP 的好处是不用考虑资源耗尽的问题。而且可以使用一个套接字负责与多个客户端通信，可以实现组播和广播机制。

9.3.4 服务端程序设计的两种基本模式

不管使用 TCP 还是 UDP 协议，在程序结构上都可以使用两种基本模式：循环模式和并发模式。

所谓循环模式，就是服务端进程在总体结构上是一个循环，一次处理一个请求。这样，有多个客户端请求时，请求放入队列，依次等待处理。这时，就产生一个时间问题，因为队列的长度是有限的，处理请求的时间过长会导致队列满而不能接受新的请求。例如 ftp 协议就不应该使用循环模式。循环模式使用单进程结构。

并发模式的服务端进程一般可以同时处理多个请求，结构上一般采用父进程接受请求，然后调用 fork 产生子进程，由子进程处理请求，这样一般是多进程的结构。并发模式的优点是可以同时处理多个请求，客户端等待时间短。在并发模式设计时，也可以采用单进程的结构，即使用 select 调用来获得异步 I/O。单进程并发的优点是服务端共享数据区，即对不同的客户不仅可共享正文，而且可以共享数据，使得客户端之间交换数据易于实现。



9.4 服务端程序结构

本节将介绍服务端编程中使用的各种程序结构。读者应结合以前章节介绍的内容来理解。其实，程序结构并不是死板的规定，这里只是总结的几种结构，如何制订程序的结构，关键还是程序设计者对网络编程的理解。

9.4.1 无连接服务端（UDP）的循环模式

对于无连接服务端的循环模式，从结构上讲，比较简单。但是，要使用这种模式实现比较复杂的应用层协议就有一定的复杂度。

第一步，使用套接字调用创建一个 `SOCK_DGRAM` 类型的套接字。

第二步，使用 `bind` 调用将端口和地址绑定到该套接字上。在 `bind` 地址和端口时，可以使用 `INADDR_ANY` 作为 `bind` 的地址。

第三步，用 `recvfrom` 得到请求消息。

第四步，使用 `sendto` 将要发送的消息发送回客户端，这里可以根据自己设计的应用协议处理。

第五步，处理完成后返回第三步，进入下一次循环。

9.4.2 无连接服务端（UDP）的并发模式

在无连接服务端编程步骤如下：

第一步，创建一个 `SOCK_DGRAM` 类型的套接字。

第二步，使用 `bind` 调用来指定本机地址和侦听端口。

第三步，调用 `recvfrom`，这时没有消息来时 `recvfrom` 被阻塞。

第四步，一旦 `recvfrom` 返回收到的消息，则调用 `taskSpawn` 产生一个新任务，新任务和客户端进行信息交互，而原任务返回第三步。

第四步，新任务一旦处理完客户的请求，则新任务结束并退出。

这时要考虑的一个问题是，处理客户请求的时间和调用 `taskSpawn` 产生新任务的花费之间的平衡。如果服务端与客户交换消息的时间要远大于或至少大于 `taskSpawn` 的时间，这时使用并发模式才是合算的。



9.4.3 面向连接服务端 (TCP) 的循环模式

这种模式的建立步骤如下:

第一步, 创建一个 `SOCK_STREAM` 类型的套接字。

第二步, 调用 `bind` 指定地址和端口。

第三步, 调用 `listen` 指定未处理连接请求的队列长度。

第四步, 调用 `accept` 等待连接请求 (在阻塞方式下), 一旦有连接请求, `accept` 返回一个新的套接字描述符。

第五步, 通过该描述符与客户端交互信息, 一旦交互完毕, 关闭新产生的套接字并返回第四步。

在这种面向连接的循环模式中也要考虑效率问题。当第五步交换信息的时间过长时, 会使得后续请求等待过长, 这时就要考虑使用并发模式。

9.4.4 面向连接服务端 (TCP) 的并发模式

在 9.4.3 中读者可以发现, 在第五步时, 完全可以产生一个新任务来处理与客户端的信息交互, 而原任务返回第四步调用 `accept` 来等待接收下一次连接请求。新任务在处理完与客户之间的交互则结束退出。这时考虑到效率的问题, 应该是处理与客户之间交互的时间要大于调用 `taskSpawn` 产生的花费。因为每次 `accept` 成功返回都会返回一个新的套接字描述符, 所以新任务之间不会有干扰。

9.4.5 单进程并发模式的 TCP 服务端设计

单进程并发是在一个进程中同时处理多个请求。单进程使用非阻塞连接, 使用套接字调用来编程。

在非阻塞连接使用 `select` 调用的 TCP 服务端程序设计中, 主要有这么几步:

第一步, 创建一个 `SOCK_STREAM` 类型的套接字, 并将该套接字加入 `select` 的监测是否有输入的集合中。

第二步, 使用 `bind` 调用, 设定本地地址和服务端口。

第三步, 使用 `listen` 调用, 指定队列长度。

第四步, 使用 `select` 等待某个套接字有数据到达。

第五步, 一旦有数据到达最早产生的套接字, 则调用 `accept` 来接受请求, 并产生新的套接字, 根据应用层协议处理新的套接字的信息, 并将新的套接字描述符加入 `select` 的监测输入集合, 返回第四步。如果是新的套接字有数据到达, 则按应用层协议处理。单进程并发的优点是: 不同的连接可以共享数据。对于多进程, 并发中各子进程要共享数据就要通过其他



的方法来实现（如进程间通信）。

9.5 多协议（TCP、UDP）服务端

目前有些应用支持 TCP 和 UDP 两种传输协议。这时，如果编写两个程序在系统上运行，一个进程处理通过 TCP 连接的请求，一个进程处理通过 UDP 的请求。这样，从各方面都有些浪费。但分离的服务程序有个好处就是，如果系统管理员要选择使用基于 TCP 的或者基于 UDP 的服务程序，这时系统管理员很容易控制。

对于一个服务使用两个进程来分别处理 TCP 和 UDP 请求的缺点是程序开发中的重复和进程运行时资源的浪费。因为在一个应用层协议中，处理的计算大部分相同，相同部分可以共用代码。

在多协议服务端程序中，第一步改为使用两次套接字调用，分别打开两个非阻塞的 SOCK_STREAM 和 SOCK_DGRAM 类型的套接字。在单进程结构的服务端中，第二步改为使用 select 来等待这两个套接字的数据到达。

9.6 编程实例

下面的例子使用客户 / 服务器通信模式。

9.6.1 使用 TCP 协议

服务器一方使用 TCP 套接字与客户端通信。在服务的主循环中，任务 tcpServerWorkTask 首先读来自客户端的请求，并向控制台输出客户信息，如果需要，将向客户端发送应答信息。

客户端通过控制台提示输入，建立请求报文，然后发往服务端，如果需要应答，则等待来自服务器端的回应。

为了简单起见，我们假定客户端和服务端代码执行在数据大小和排列方式相同的机器上。

例 9.2 TCP 客户 / 服务器通信模式示例

```

/*****/
/* tcpExample.h - header used by both TCP server and client examples */

```



```
/* defines */

#define SERVER_PORT_NUM      5001 /* server's port number for bind() */
#define SERVER_WORK_PRIORITY 100  /* priority of server's work task */
#define SERVER_STACK_SIZE    10000 /* stack size of server's work task */
#define SERVER_MAX_CONNECTIONS 4  /* max clients connected at a time */

#define REQUEST_MSG_SIZE     1024 /* max size of request message */

#define REPLY_MSG_SIZE       500  /* max size of reply message */

/* structure for requests from clients to server */

struct request
{
    int reply; /* TRUE = request reply from server */
    int msgLen; /* length of message text */
    char message[REQUEST_MSG_SIZE]; /* message buffer */
};

/*TCP 客户 / 服务器通信模式示例客户端代码*/

/* tcpClient.c - TCP client example */

/*
DESCRIPTION
This file contains the client-side of the VxWorks TCP example code.
The example code demonstrates the usage of several BSD 4.3-style
socket routine calls.
*/

/* includes */

#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "hostLib.h"
#include "ioLib.h"
#include "tcpExample.h"
```



```
/******  
*  
* tcpClient - send requests to server over a TCP socket  
*  
* This routine connects over a TCP socket to a server, and sends a  
* user-provided message to the server. Optionally, this routine  
* waits for the server's reply message.  
*  
* This routine may be invoked as follows:  
*   -> tcpClient "remoteSystem"  
*   Message to send:  
*   Hello out there  
*   Would you like a reply (Y or N):  
*   y  
*   value = 0 = 0x0  
*   -> MESSAGE FROM SERVER:  
  
*   Server received your message  
*  
* RETURNS: OK, or ERROR if the message could not be sent to the server.  
*/
```

STATUS tcpClient

```
(  
char *      serverName    /* name or IP address of server */  
)  
{  
struct request  myRequest; /* request to send to server */  
struct sockaddr_in serverAddr; /* server's socket address */  
char          replyBuf[REPLY_MSG_SIZE]; /* buffer for reply */  
char          reply;      /* if TRUE, expect reply back */  
  
int          sockAddrSize; /* size of socket address structure */  
int          sFd;         /* socket file descriptor */  
int          mlen;       /* length of message */  
  
/* create client's socket */  
  
if ((sFd = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)  
{  
perror ("socket");  
return (ERROR);  
}
```



```
/* bind not required - port number is dynamic */

/* build server socket address */

sockAddrSize = sizeof (struct sockaddr_in);

bzero ((char *) &serverAddr, sockAddrSize);
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons (SERVER_PORT_NUM);

if (((serverAddr.sin_addr.s_addr = inet_addr (serverName)) == ERROR) &&
    ((serverAddr.sin_addr.s_addr = hostGetByName (serverName)) == ERROR))
{
    perror ("unknown server name");
    close (sFd);
    return (ERROR);
}

/* connect to server */

if (connect (sFd, (struct sockaddr *) &serverAddr, sockAddrSize) == ERROR)
{
    perror ("connect");
    close (sFd);
    return (ERROR);
}

/* build request, prompting user for message */

printf ("Message to send: \n");
mlen = recv (STD_IN, myRequest.message, REQUEST_MSG_SIZE);
myRequest.msgLen = mlen;
myRequest.message[mlen - 1] = '\0';

printf ("Would you like a reply (Y or N): \n");
recv (STD_IN, &reply, 1);
switch (reply)
{
    case 'y':
    case 'Y': myRequest.reply = TRUE;
              break;
    default: myRequest.reply = FALSE;
}
```



```
        break;
    }

    /* send request to server */

    if (send (sFd, (char *) &myRequest, sizeof (myRequest)) == ERROR)
    {
        perror ("send");
        close (sFd);
        return (ERROR);
    }

    if (myRequest_reply) /* if expecting reply, recv and display it */
    {
        if (recv (sFd, replyBuf, REPLY_MSG_SIZE) < 0)
        {
            perror ("recv");
            close (sFd);
            return (ERROR);
        }

        printf ("MESSAGE FROM SERVER:\n%s\n", replyBuf);
    }

    close (sFd);
    return (OK);
}
```

/* TCP 客户 / 服务器通信模式示例服务器代码*/

/* tcpServer.c - TCP server example */

/*

DESCRIPTION

This file contains the server-side of the VxWorks TCP example code. The example code demonstrates the usage of several BSD 4.3-style socket routine calls.

***/**

/* includes */



```

#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "taskLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "ioLib.h"
#include "fioLib.h"
#include "tcpExample.h"

/* function declarations */

VOID tcpServerWorkTask (int sFd, char * address, u_short port);

/*****
 *
 * tcpServer - accept and process requests over a TCP socket
 *
 * This routine creates a TCP socket, and accepts connections over the socket
 * from clients. Each client connection is handled by spawning a separate
 * task to handle client requests.
 *
 * This routine may be invoked as follows:
 *   -> sp tcpServer
 *   task spawned: id = 0x3a6f1c, name = t1
 *   value = 3829532 = 0x3a6f1c
 *
 *   -> MESSAGE FROM CLIENT (Internet Address 90.0.0.10, port 1027):
 *   Hello out there
 *
 * RETURNS: Never, or ERROR if a resources could not be allocated.
 */

STATUS tcpServer (void)
{
    struct sockaddr_in serverAddr; /* server's socket address */
    struct sockaddr_in clientAddr; /* client's socket address */
    int sockAddrSize; /* size of socket address structure */
    int sFd; /* socket file descriptor */

    int newFd; /* socket descriptor from accept */
    int ix = 0; /* counter for work task names */
    char workName[16]; /* name of work task */

```



```
/* set up the local address */

sockAddrSize = sizeof (struct sockaddr_in);
bzero ((char *) &serverAddr, sockAddrSize);
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons (SERVER_PORT_NUM);
serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);

/* create a TCP-based socket */

if ((sFd = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
{
    perror ("socket");
    return (ERROR);
}

/* bind socket to local address */

if (bind (sFd, (struct sockaddr *) &serverAddr, sockAddrSize) == ERROR)
{
    perror ("bind");
    close (sFd);
    return (ERROR);
}

/* create queue for client connection requests */

if (listen (sFd, SERVER_MAX_CONNECTIONS) == ERROR)
{
    perror ("listen");

    close (sFd);
    return (ERROR);
}

/* accept new connect requests and spawn tasks to process them */

FOREVER
{
    if ((newFd = accept (sFd, (struct sockaddr *) &clientAddr,
        &sockAddrSize)) == ERROR)
    {
        perror ("accept");
        close (sFd);
    }
}
```



```

        return (ERROR);
    }

    sprintf (workName, "tTcpWork%d", ix++);
    if (taskSpawn(workName, SERVER_WORK_PRIORITY, 0, SERVER_STACK_SIZE,
        (FUNCPTR) tcpServerWorkTask, newFd,

            (int)      inet_ntoa      (clientAddr.sin_addr),      ntohs
(clientAddr.sin_port),
        0, 0, 0, 0, 0, 0, 0) == ERROR)
    {
        /* if taskSpawn fails, close fd and return to top of loop */

        perror ("taskSpawn");
        close (newFd);
    }
}

}

/*****
*
* tcpServerWorkTask - process client requests
*
* This routine recvs from the server's socket, and processes client
* requests. If the client requests a reply message, this routine
*
* will send a reply to the client.
*
* RETURNS: N/A.
*/

VOID tcpServerWorkTask
(
    int          sFd,          /* server's socket fd */
    char *       address,     /* client's socket address */
    u_short     port         /* client's socket port */
)
{
    struct request  clientRequest; /* request/message from client */
    int            nRead;         /* number of bytes recv */
    static char    replyMsg[] = "Server received your message";

    /* recv client request, display message */

```



```
while ((nRead = fioRead (sFd, (char *) &clientRequest,
    sizeof (clientRequest))) > 0)
{
    printf ("MESSAGE FROM CLIENT (Internet Address %s, port %d):\n%s\n",
        address, port, clientRequest.message);

    free (address);          /* free malloc from inet_ntoa() */

    if (clientRequest.reply)
        if (send (sFd, replyMsg, sizeof (replyMsg)) == ERROR)
            perror ("send");
}

if (nRead == ERROR)        /* error from recv() */
    perror ("recv");

close (sFd);              /* close server socket connection */
}
/*****/
```

9.6.2 使用 UDP 协议

正如 TCP 类似于电话通信，UDP 类似于发送邮件。下例使用 UDP 建立客户 / 服务器通信模式。服务器使用面向数据报的套接字与客户通信。

在主循环中，任务 `udpServer()` 读取来自客户端的请求，如果需要并显示来自客户的信息。客户端通过控制台提示输入建立请求报文，然后发送到服务器。

同上例，为了简单起见，我们假定客户端和服务器端代码执行在数据大小和排列方式相同的机器上。

例 9.3 UDP 客户 / 服务器通信模式示例

```
/*****/
/* udpExample.h - header used by both UDP server and client examples */

#define SERVER_PORT_NUM    5002  /* server's port number for bind() */
#define REQUEST_MSG_SIZE   1024  /* max size of request message */

/* structure used for client's request */
```



```
struct request
{
    int display;          /* TRUE = display message */
    char message[REQUEST_MSG_SIZE]; /* message buffer */
};

/* UDP 客户 / 服务器通信模式客户端示例代码 */
/* udpClient.c - UDP client example */

/*
DESCRIPTION
This file contains the client-side of the VxWorks UDP example code.
The example code demonstrates the usage of several BSD 4.3-style
socket routine calls.
*/

/* includes */

#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "hostLib.h"
#include "ioLib.h"
#include "udpExample.h"

/*****

*
* udpClient - send a message to a server over a UDP socket
*
* This routine sends a user-provided message to a server over a UDP socket.
* Optionally, this routine can request that the server display the message.
*
* This routine may be invoked as follows:
*   -> udpClient "remoteSystem"
*   Message to send:
*   Greetings from UDP client
*   Would you like server to display your message (Y or N):
*****/
```



```
*
 * y
 * value = 0 = 0x0
 *
 * RETURNS: OK, or ERROR if the message could not be sent to the server.
 */

STATUS udpClient
(
  char *      serverName /* name or IP address of server */
)
{
  struct request myRequest; /* request to send to server */
  struct sockaddr_in serverAddr; /* server's socket address */
  char display; /* if TRUE, server prints message */
  int sockAddrSize; /* size of socket address structure */
  int sFd; /* socket file descriptor */
  int mlen; /* length of message */

  /* create client's socket */

  if ((sFd = socket (AF_INET, SOCK_DGRAM, 0)) == ERROR)
  {
    perror ("socket");
    return (ERROR);
  }

  /* bind not required - port number is dynamic */

  /* build server socket address */

  sockAddrSize = sizeof (struct sockaddr_in);
  bzero ((char *) &serverAddr, sockAddrSize);
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_port = htons (SERVER_PORT_NUM);

  if (((serverAddr.sin_addr.s_addr = inet_addr (serverName)) == ERROR) &&
      ((serverAddr.sin_addr.s_addr = hostGetByName (serverName)) == ERROR))
  {
```



```
        perror ("unknown server name");
        close (sFd);
        return (ERROR);
    }

    /* build request, prompting user for message */

    printf ("Message to send: \n");
    mlen = read (STD_IN, myRequest.message, REQUEST_MSG_SIZE);
    myRequest.message[mlen - 1] = '\0';

    printf ("Would you like the server to display your message (Y or N): \n");
    read (STD_IN, &display, 1);
    switch (display)

        {
        case 'y':
        case 'Y': myRequest.display = TRUE;
            break;
        default: myRequest.display = FALSE;
            break;
        }

    /* send request to server */

    if (sendto (sFd, (caddr_t) &myRequest, sizeof (myRequest), 0,
        (struct sockaddr *) &serverAddr, sockAddrSize) == ERROR)
        {
        perror ("sendto");
        close (sFd);
        return (ERROR);
        }

    close (sFd);
    return (OK);
}

/* UDP 客户 / 服务器通信模式服务器端示例代码*/
/* udpServer.c - UDP server example */

/*
```



DESCRIPTION

This file contains the server-side of the VxWorks UDP example code. The example code demonstrates the usage of several BSD 4.3-style socket routine calls.

```
*/

/* includes */

#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "ioLib.h"
#include "fioLib.h"
#include "udpExample.h"

/*****

*
* udpServer - rcv from UDP socket and display client's message if requested
*
* Example of VxWorks UDP server:
*   -> sp udpServer
*   task spawned: id = 0x3a1f6c, name = t2
*   value = 3809132 = 0x3a1f6c
*   -> MESSAGE FROM CLIENT (Internet Address 90.0.0.11, port 1028):
*   Greetings from UDP client
*
* RETURNS: Never, or ERROR if a resources could not be allocated.
*/

STATUS udpServer (void)
{
    struct sockaddr_in serverAddr; /* server's socket address */

    struct sockaddr_in clientAddr; /* client's socket address */
    struct request     clientRequest; /* request/Message from client */
    int                sockAddrSize; /* size of socket address structure */
    int                sFd;          /* socket file descriptor */
    char               inetAddr[INET_ADDR_LEN];
```



```
/* buffer for client's inet addr */

/* set up the local address */

sockAddrSize = sizeof (struct sockaddr_in);

bzero ((char *) &serverAddr, sockAddrSize);
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons (SERVER_PORT_NUM);
serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);

/* create a UDP-based socket */

if ((sFd = socket (AF_INET, SOCK_DGRAM, 0)) == ERROR)
{
    perror ("socket");
    return (ERROR);
}

/* bind socket to local address */

if (bind (sFd, (struct sockaddr *) &serverAddr, sockAddrSize) == ERROR)
{
    perror ("bind");

    close (sFd);
    return (ERROR);
}

/* recv data from a socket and satisfy requests */

FOREVER
{
    if (recvfrom (sFd, (char *) &clientRequest, sizeof (clientRequest), 0,
        (struct sockaddr *) &clientAddr, &sockAddrSize) == ERROR)
    {
        perror ("recvfrom");
        close (sFd);
        return (ERROR);
    }
}
```



```
/* if client requested that message be displayed, print it */

if (clientRequest.display)
{
    /* convert inet address to dot notation */

    inet_ntoa_b (clientAddr.sin_addr, inetAddr);
    printf ("MESSAGE FROM CLIENT (Internet Address %s, port
%d):\n%s\n",
            inetAddr,      ntohs      (clientAddr.sin_port),
clientRequest.message);
    }
}

/*****/
```

WRS 公司提供的软件产品分为两部分：开发环境 Tornado 和实时操作系统 VxWorks，运行 Tornado 的一方称为宿主机，而运行 VxWorks 的一方称为目标机，二者根据实际应用可以采用网络或串行线连接。一般地，用户不能在自己的目标系统直接运行 WRS 公司提供的 VxWorks 映像，因为不同应用系统的目标机是有差别的，为此 VxWorks 要根据实际的目标环境重新配置与生成，这个过程是在宿主机上完成的。

10.1 VxWorks 的目录与文件

WRS 公司产品安装在一个独立的目录 \Tornado 下，形成自己的目录树，随着版本的不同，目录树也稍有变化，比如版本 5.4 与 5.3.1 根目录下是有差别的。一般地，宿主机工作文件目录在 \tornado\host 下，而目标系统的目录与文件在 \tornado\target 下，操作系统 VxWorks 的配置主要是通过对目标系统目录下相关配置文件进行编辑与修改来完成的。为了更好地了解它们，这里简要介绍一下。

10.1.1 配置目录 \tornado\target\config

该目录包含的文件是用来配置和生成一个专用 VxWorks 系统，它包括系统相关的模块和用户可修改的模块，这些文件被组织分配到几个子目录下：子目录 all 包含了所有 VxWorks 公共的执行模块（独立于系统的模块）；另一个子目录 bspname 包含了 VxWorks 对于指定目标机硬件的模块（系统相关的模块）。以下简要说明各子目录所包含的文件及其作用。



➤ config\all

该子目录含有系统配置模块，主要有以下几个文件组成：

- ◆ bootInit.c: 该程序是系统初始化模块，根据引导方式，将 ROM 程序拷贝到 RAM 中，如果是从 ROM 引导，完成系统所需工作区的初始化。
- ◆ configAll.h: 该文件包含所有目标系统公用的配置参数定义，用户可以根据目标系统的特殊要求，编辑或修改该文件的配置参数。
- ◆ usrConfig.c: 用户可定义的系统配置模块源程序，包含根任务、基本的系统初始化、网络初始化、时钟中断程序等。
- ◆ bootConfig.c: ROM 引导的 VxWorks 系统配置模块源程序。

对于上述两个源程序，可以通过编辑和修改配置文件进行裁剪。

➤ config\bspname

这个配置子目录包含有与一个特定目标机系统相关的 VxWorks 配置模块，它主要包括以下文件。

- ◆ Makefile: 为特定目标系统生成 ROM 引导与 VxWorks 系统映像，该文件定义了系统编译规则，生成引导映像的文件格式。
- ◆ sysALib.s: 它是用汇编语言编写的系统相关的模块，包含有系统基本的端口读写函数、CPU 检测函数和系统 GDT 的访问函数。
- ◆ sysLib.c: 该文件包含有系统相关的库函数，提供特定目标系统的可编程芯片驱动程序，如 i8250 UART 串行通信、i8253 定时器、i8259 中断控制器、PCI 总线驱动、网卡驱动等；该文件提供了内存映像数据结构，用户可以编辑该结构完成特殊设备的安装。
- ◆ sysSerial.c: 目标板上串行口初始化程序，主要是 COM1、COM2 设备的初始化。
- ◆ config.h: 硬件配置参数头文件。
- ◆ bspname.h: 目标板参数配置头文件，该文件详细描述的目标系统的所有资源配置，如端口地址、中断号等，对于 PC 版本，该文件名为 PC.H。
- ◆ target.txt: 目标板的参考信息。
- ◆ romInit.s: 用汇编语言编写的初始化代码源程序，它是 ROM 引导 VxWorks 和基于 ROM 版本的 VxWorks 的入口，该程序初始化系统寄存器，由实地址模式切换到平面保护模式。
- ◆ VxWorks 和 VxWorks.sym: 连接完成的二进制目标码 VxWorks 系统文件和根据配置文件建立的 VxWorks 的符号表文件。
- ◆ bootrom: 从 ROM 引导 VxWorks 的引导目标模块，当建立宿主机/目标机环境时，由宿主机通过串行线或网络将 VxWorks 系统映像加载到目标机。



10.1.2 目标系统头文件\tornado\target\h

该目录包含了 VxWorks 提供的所有头文件，在编制应用程序时必须包括它们中的一部分或全部。该头文件目录又包含了以下几个子目录，分别描述如下：

➤ h\arch

该子目录包含与硬件结构相关的头文件子目录 (i86)，当开发环境配置模拟器时，该子目录提供模拟目标系统的头文件目录 (simnt)。

➤ h\arpa

该子目录包含了使用 inetLib 库的头文件，主要是 ininternet 头文件。

➤ h\drv

该子目录包含了指定的硬件（主要是驱动程序）的头文件，如软驱、硬盘、串行口、计时器、并口、PCI 设备、SCSI、DMA、WDB、增强型网络接口等。

➤ h\make

该子目录包含的文件描述了针对每一 CPU 和工具箱的 makefiles 规则。

➤ h\net

该子目录包含了使用 VxWorks 网络时所包括的所有内部头文件；网络驱动程序必须要包括它们，应用模块可以不用。

➤ h\netinet

该子目录包含了特定的因特网头文件。

➤ h\private

该子目录包含了 VxWorks 私有代码的头文件。

➤ h\rpc

当应用系统使用远程过程调用函数库时，必须包括该子目录下包含的头文件。

➤ h\sys

该子目录包含有 POSIX 标准指定的头文件。

➤ h\types

该子目录包含了各种类型定义的头文件。



10.1.3 目标系统函数库子目录\tornado\target\lib

该目录包含了独立于机器的目标库和由 VxWorks 提供的模块。主要有以下内容:

➤ **objcputoolvx**

该目录包含了以单独文件形式的 VxWorks 目标模块, 它们适合于将模块动态下载到目标机。

➤ **objcputoolcf**

在该目录下, 对于使用可选的 Wind C++编译器而生成的 VxWorks 目标模块版本是可以改变的, 而其他目标模块是使用默认的 GNU C++编译器生成的。

➤ **libcputoolvx.a**

存档格式的库文件包含有生成 VxWorks 的目标模块。

10.2 VxWorks 的板级支持包 BSP

在上节中提到的目录 config/bspname 包含了在指定目标系统上运行 VxWorks 的硬件描述文件, 如板上串行口、并行口、键盘、显示等, 对于不同的目标系统, 用户可以通过修改这些文件, 完成 BSP 与 VxWorks 在异种平台的移植。该目录下主要文件有 Makefile、sysLib.c、sysALib.s、romInit.s、bspname.h、config.h。

10.2.1 系统库

sysLib.c 是一个与目标系统相关的库文件, 它提供了 VxWorks 的板级接口驱动程序和与应用代码集成的硬件相关处理。该文件涉及的功能是:

➤ **初始化**

- ◆ 初始化硬件成为已知状态, 如中断控制器 8259、串行口 8250、定时器 8253、打印机口、软驱、硬盘等。
- ◆ 初始化设备以便识别系统, 如 PCI 设备、SCSI 设备、PCMCIA 设备、用户专用设备等等。
- ◆ 设备驱动程序的初始化。



➤ 存储器/地址空间映像功能

- ◆ 获得板上存储器大小。
- ◆ 允许外部总线访问板上存储器。
- ◆ 映像局部和总线地址空间。
- ◆ 允许/禁止 cache 高速缓存存储器。
- ◆ 设置/获取非易失存储器内容。
- ◆ 定义板的存储器映像。
- ◆ 具有 MMU 的多处理器从虚拟到物理存储器映像。

➤ 总线中断功能

- ◆ 允许/禁止总线中断级。
- ◆ 产生总线中断。

➤ 时钟/计数器功能

- ◆ 允许/禁止定时器中断。
- ◆ 设置时间的周期率。

➤ 信箱/定位监听功能

- ◆ 允许信箱/定位监听中断。

一般地，不同的供应商提供的目标板是有差别的，sysLib 库不能支持每一种板，某些板 sysLib 功能是通过硬件开关、跳线或 PAL 器件，而不靠是通过软件对可编程寄存器访问实现的。

该系统库中子程序或函数模块是由在目录 config/all 下的配置模块 usrConfig.c 和 bootConfig.c 调用的。设备驱动程序使用某些存储器映像的子程序和总线函数。

10.2.2 虚拟存储器映像

对于支持 MMU 的目标板，sysPhysMemDesc 数据结构定义了虚拟到物理存储器映像。这个表在文件 sysLib.c 中定义，但是，某些 BSP 把它放在一个独立的文件 memDesc.c 中。该数据结构在系统中被定义为一个数组结构 PHYS_MEM_DESC。在这个数据结构的描述符中的两个入口不能覆盖；每一个入口必须是唯一的存储器空间。sysPhysMemDesc 数组反映了你的系统配置，并且你可以改变 MMU 存储器映像。例如，改变局部存储器的大小或者 VME 主存储器空间大小。当你访问没有映像的存储器时会发生一个总线错误。



10.2.3 串口设备驱动程序

文件 `ysSerial.c` 提供了指定板上串口初始化。串口 I/O 驱动程序在 `src/drv/sio` 目录下。`ttyDrv` 库使用此串口 I/O 驱动程序为 VxWorks 提供终端操作。

10.2.4 BSP 初始化模块

BSP 初始化模块有两个文件：

- ◆ `romInit.s` 是用汇编语言编写的初始化程序，是 VxWorks 从 ROM 引导的入口，它开始以实地址方式运行，初始化 CPU 寄存器后转保护模式。
- ◆ `sysALib.s` 也是用汇编语言编写的，含有初始化和系统相关的汇编级子程序，如端口字节、字、长字的读写，CPU 类型检测，系统描述符的访问等。

10.3 VxWorks 的配置文件与配置项

VxWorks 的配置文件分为配置头文件与配置模块文件，通过对它们进行编辑与修改，利用开发环境提供的工具进行配置与生成满足自己目标环境的操作系统，配置头文件有 `/target/config/all/configAll.h` 和 `/target/config/bspname/config.h`，配置模块文件主要有 `/target/config/all/usrConfig.c` 模块与 `/target/src/config` 目录下初始化模块。

10.3.1 配置头文件

在目录 `/target/config/all` 下的文件 `configAll.h` 称为全局配置头文件，而在 `/target/config/bspname` 目录下的文件 `config.h` 是与 BSP 确定的目标系统相关的头文件；你可以对这两个文件进行编辑，通过包括或去除在配置头文件中的定义来实施对 VxWorks 的配置。

■ 全局配置头文件 `configAll.h`

头文件 `configAll.h` 包含了 WRS 公司所提供的所有目标机的默认定义，这些定义也可以在头文件 `config.h` 中重新定义。该头文件提供了以下的选项和参数定义：

- ◆ 内核配置参数。
- ◆ I/O 系统配置参数。



- ◆ 网络文件系统(NFS)配置参数。
- ◆ 可选的软件模块配置。
- ◆ 可选的设备控制器配置。
- ◆ 高速存储器方式。
- ◆ 不同共享内存目标对象的最大数目。
- ◆ 设备控制器 I/O 地址、中断向量和中断级。
- ◆ 各种地址与常量。

■ 确定的 BSP 配置头文件 config.h

在目录 config/bspname 下, 有一个确定的 BSP 头文件 config.h。这个文件包含有确定目标机的参数与选项定义, 它们也可以在 configAll.h 文件中定义。如果某目标机不能以默认 I/O 地址 (在 configAll.h 文件中定义) 访问一个设备控制器, 则可以在 config.h 中重新定义。一般地, 不要修改 configAll.h 文件中的选项定义, 如果你的应用系统需要, 则在 config.h 文件中进行定义。

头文件 config.h 包括以下参数定义:

- ◆ 从 ROM 引导的默认的引导参数字符行。
- ◆ 系统时钟和校验出错的中断向量。
- ◆ 设备控制器 I/O 地址, 中断向量和中断级。
- ◆ 共享内存网络参数。
- ◆ 各种存储器地址和常量。

■ 配置选项

VxWorks 安装的选项和驱动程序可以根据目标系统进行裁剪或默认。它们由配置头文件的宏决定, 并且控制在文件 config/all/usrConfig.c 中的模块有条件地进行编译。

配置头文件 configAll.h 和 config.h 的发行版包括所有可用的软件选项和几个网络设备驱动程序。如果你使用工程工具, 则可以很方便地对它们进行选择; 对于手工方式你可以在配置文件中通过包括 (#define XXX) 与去除 (#undef XXX) 手段定义配置宏 (对于部分配置宏列表, 参见表 10.1)。例如, 要包含 ANSI C 声明库, 一定要定义 INCLUDE_ANSI_ASSERT; 要包含网络文件系统 (NFS) 工具, 一定要定义 INCLUDE_NFS。

在表 10.1 所示的宏中, 以 XXX 结尾的是无效的宏, 但代表一系列选项, 它可以用一个指定功能的后缀替代。例如, INCLUDE_CPLUS_XXX 指的是一系列宏, 它包括 INCLUDE_CPLUS_MIN 和 INCLUDE_CPLUS_BOOCH。



表 10.1 关键的 VxWorks 配置选项

宏 名	含 义
INCLUDE_ADA	支持 Ada 语言
INCLUDE_ANSI_XXX	各种 ANSI C 库选项
INCLUDE_BOOTP	支持 BOOTP
INCLUDE_CACHE_SUPPORT	支持高速存储器
INCLUDE_CPLUS	支持 Bundled C++
INCLUDE_CPLUS_XXX	各种 C++ 支持选项
INCLUDE_DEBUG	在目标机驻留反向兼容的命令解释时, 支持本地调试
INCLUDE_DEMO	使用简单的演示程序
INCLUDE_DOSFS	DOS 兼容的文件系统
INCLUDE_FLOATING_POINT	支持浮点 I/O
INCLUDE_FORMATTED_IO	格式化 I/O
INCLUDE_FTP_SERVER	FTP 服务支持
INCLUDE_HW_FP	硬件浮点支持
INCLUDE_INSTRUMENTATION	WindView 分析器
INCLUDE_IO_SYSTEM	I/O 系统包
INCLUDE_LOADER	驻留目标机的模块加载包
INCLUDE_LOGGING	登录工具
INCLUDE_MEM_MGR_FULL	所有内存管理器
INCLUDE_MIB2_XXX	各种 MIB-2 选项
INCLUDE_MMU_BASIC	Bundled MMU 支持
INCLUDE_MMU_FULL	Unbundled MMU 支持 (需要 VxVMI)
INCLUDE_MSG_Q	消息队列支持
INCLUDE_NETWORK	网络子系统代码
INCLUDE_NFS	网络文件系统 (NFS)
INCLUDE_NFS_SERVER	NFS 服务器
INCLUDE_PIPES	管道驱动程序
INCLUDE_POSIX_XXX	各种 POSIX 选项
INCLUDE_PROTECT_TEXT	正文段写保护 (需要 VxVMI)
INCLUDE_PROTECT_VEC_TABLE	向量表写保护(需要 VxVMI)



续表

宏 名	含 义
INCLUDE_PROXY_CLIENT	Proxy ARP 客户支持
INCLUDE_PROXY_SERVER	Proxy ARP 服务器支持
INCLUDE_RAWFS	原文件系统
INCLUDE_RLOGIN	远程注册与登录
INCLUDE_RPC	远程过程调用 (RPC)
INCLUDE_RT11FS	RT-11 文件系统
INCLUDE_SCSI	SCSI 支持
INCLUDE_SCSI2	SCSI-2 扩展
INCLUDE_SCSI_BOOT	允许从 SCSI 设备加载
INCLUDE_SECURITY	远程登录安全库
INCLUDE_SEM_BINARY	二进制信号量支持
INCLUDE_SEM_COUNTING	计数信号量支持
INCLUDE_SEM_MUTEX	互斥信号量支持
INCLUDE_SHELL	C 表达式解释器 (目标机 shell 下)
INCLUDE_SHOW_ROUTINES	各种系统目标显示工具
INCLUDE_SIGNALS	软件信号量工具
INCLUDE_SM_OBJ	共享内存对象支持 (需要 VxMP)
INCLUDE_SNMPD	SNMP 代理机构
INCLUDE_SPY	任务活动监视器
INCLUDE_STDIO	标准 I/O 包
INCLUDE_SW_FP	软件仿浮点包
INCLUDE_SYM_TBL	驻留目标机符号表支持
INCLUDE_TASK_HOOKS	内核调用支持
INCLUDE_TASK_VARS	任务可变支持
INCLUDE_TELNET	远程登录 (Remote login with telnet)
INCLUDE_TFTP_CLIENT	TFTP 客户端支持
INCLUDE_TFTP_SERVER	TFTP 服务器支持
INCLUDE_TIMEX	定时器运行函数 (Function execution timer)
INCLUDE_UNLOADER	驻留目标机模块卸载包
INCLUDE_WATCHDOGS	看门狗支持



续表

宏 名	含 义
INCLUDE_WDB	目标机调试机构
INCLUDE_WINDVIEW	WindView 命令服务器
INCLUDE_ZBUF SOCK	Zbuf 套接字接口

10.3.2 配置模块文件 usrConfig.c

为了满足你的开发需要，使用 VxWorks 的配置头文件配置你的 VxWorks 系统时，用户不要常去修改 WRS 公司提供的 `usrConfig.c`，或者在目录 `config/all` 下的其他文件。如果特殊情况需要修改，建议你将所有文件拷贝到另一个目录，并且在你的 `Makefile` 文件中增加一个 `CONFIG_ALL` 宏指向修改的文件。例如，将下面内容加到你的 `makefile` 文件。

```
# ../myAll contains a copy of all the ../all files
CONFIG_ALL = ../myAll
```

10.4 VxWorks 的初始化

对于一个确定目标系统的 VxWorks，它的初始化按一定的时间顺序进行，但对于不同版本的 VxWorks 的初始化序列也稍微不同。在本节描述了一个典型的应用配置中 VxWorks 的初始化序列，这些初始化步骤是以 VxWorks 运行为序。为了清晰，这个序列分为多个主要阶段或过程调用，关键过程以标题列出，并且以排列顺序描述。

10.4.1 VxWorks 入口 `sysInit`

VxWorks 系统开始运行的第一步是将 VxWorks 加载到主内存。在 VxWorks ROM 的控制下，通常从开发宿主机下载。下一步，引导 ROM 将控制权转到 VxWorks 的启动入口 `sysInit()`。这个入口由在 `makefile` 和 `config.h` 中的配置参数 `RAM_LOW_ADRS` 定义。VxWorks 内存规划随不同的目标体系结构而不同。

`sysInit()`是与系统相关的用汇编语言编写的模块，该过程在文件 `sysALib.s` 中。它首先关闭所有中断，禁止高速缓冲存储器，并且将处理器寄存器（包括 C 堆栈指针）初始化为默认值。同时它也禁止跟踪，清除所有挂起硬件中断，转而调用 `usrInit()`，该子程序是一段 C 程



序，在文件 `usrConfig.c` 中。对于某些目标机，`sysInit()` 也完成一些小部分系统相关的硬件初始化，以便能够执行 `usrInit()` 其余的初始化。它使用的初始堆栈指针占用的空间是在 VxWorks 系统映像的下部，但在中断向量区的上面。

10.4.2 初始程序 `usrInit()`

在文件 `usrConfig.c` 的程序 `usrInit()` 保存引导类型信息，在多任务内核启动前必须完成所有初始化，然后启动内核执行。它是 VxWorks 运行的第一个 C 代码，并且是在禁止所有硬件中断下调用的。

许多 VxWorks 服务不能由这个程序调用，因为它没有任务上下文（没有 TCB 和任务堆栈），这包括能引起可抢占调度的服务，如信号量，或使用该服务的函数，如 `printf()`。相反地，`usrInit()` 需要建立一初始任务 `usrRoot()`，并启动该任务。

`usrInit()` 中的初始化包括下述内容：

高速缓冲存储器初始化，`usrInit()` 的开始初始化高速缓冲存储器，设置高速缓冲存储器的模式并且置为安全状态。在 `usrInit()` 的最后，允许使用指令与数据高速缓冲。

系统 BSS 段初始化为 0，C 和 C++ 语言指定所有未初始化的变量值为 0。这些未初始化的变量被放在称为 BSS 的段中。该段在引导时实际上是不加载的，`usrInit()` 执行后第一个任务是将含有 BSS 的内存清为 0。在 ROM 引导 VxWorks 时清所有内存。

中断向量初始化，在允许中断和执行内核前必须建立异常中断向量。首先，程序 `intVecBaseSet()` 被调用建立向量表基地址；接着，`excVecInit()` 初始化所有异常向量为默认中断处理，以便能安全捕获和处理当程序出错或硬件中断引起异常情况。

将系统硬件初始化为静止状态，系统硬件初始化由调用子程序 `sysHwInit()` 完成，这主要包含复位和禁止硬件设备，在内核启动后允许产生中断。这很重要，直到 `usrRoot()` 任务完成系统初始化，VxWorks ISR 的中断服务程序没有连接到中断向量。但是，不要企图在 `sysHwInit()` 过程中连接中断向量，因为此时存储池仍未初始化。

10.4.3 初始化内核

`usrInit()` 程序以调用两个内核初始化函数宣告结束，这两个函数为：

- `usrKernelInit()` 程序（在文件 `usrKernel.c` 中）调用适当的初始化子程序。
- `kernelInit()` 程序初始化多任务环境，并且永远不返回，它要以下参数：
 - _ 被生成的作为根任务的应用程序，典型为 `usrRoot()`。
 - _ 堆栈的大小。
 - _ 开始使用的内存地址，也就是说，在 VxWorks 映像的主程序区、数据、BSS 之后。

在该区域之后的所有内存都加到系统存储池，由 `memPartLib` 管理。用于分配下载的动态模块、任务控制块等，都出自该区域。



- _ 由 sysMemTop() 指出的内存高端地址。
- _ 中断堆栈大小。
- _ 中断级。

kernelInit() 调用 intLockLevelSet(), 禁止轮循模式, 建立中断堆栈; 然后在存储池的高端创建根任务的堆栈和 TCB, 启动根任务 usrRoot(); usrInit() 运行结束后, 允许中断; 禁止所有中断源和清除挂起中断非常关键。

初始化存储池, VxWorks 包括一个在 memPartLib 模块中的内存分配工具, 它管理可用的存储器池。malloc() 函数允许从存储池中申请获得可变大小的内存块。在内部, VxWorks 使用 malloc() 动态分配内存, 尤其许多 VxWorks 服务在初始化时使用它分配数据结构。因此, 存储池初始化必须在其他任何 VxWorks 服务的初始化之前进行。

注意 Tornado 目标服务器使用和管理一部分存储器用来支持目标模块的下载和其他开发功能。

VxWorks 大量使用 malloc()。包括下载模块分配的空间, 任务启动堆栈的分配, 初始化分配的数据结构。同时鼓励你使用 malloc() 来分配任何你应用系统所需的内存。除非你必须为特殊用途保留一些绝对存储器区域外, 建议将所有未使用的内存都分配到 VxWorks 的存储池。

存储池在程序 kernelInit() 中初始化。kernelInit() 的参数指明了初始化存储池的开始和结束地址。VxWorks 用 usrInit() 的默认分配, 存储池开始地址立即被设置为引导系统的结束之后的部分, 并且包含了所有剩余可用的存储器。

可用存储器的范围由程序 sysMemTop() 决定, 这是一个系统相关的程序, 它定义可用存储器的大小。如果系统还有其他不连续的存储区域, 一般地, 你可以在 usrRoot() 任务之后调用 memAddToPool() 将它们加到系统存储池, 以便可用。

10.4.4 初始化任务 usrRoot()

当多任务内核开始执行时, 所有 VxWorks 多任务服务是可用的。在控制权转到任务 usrRoot() 后, 完成系统的初始化。主要初始化有:

- **系统时钟的初始化**
- **I/O 系统和设备驱动的初始化**
 - ◆ 控制台设备的创建
 - ◆ 标准输入输出的设置
 - ◆ 异常处理与登录的安装
 - ◆ 管道设备驱动的初始化
 - ◆ 标准 I/O 的初始化



- 文件系统设备和磁盘驱动安装的创建
- 浮点运算支持初始化
- 性能监视服务的初始化

➤ 可选服务的初始化

- WindView 的初始化。
- 目标调试机构的初始化。
- 用户提供的启动脚本的初始化。

参见文件 `config/all/usrConfig.c`，回顾 `usrRoot()` 完成的初始化序列，你可以根据自己的目标系统修改这些初始化。关于初始化的每一步含义和各种参数的意义在以下的章节中解释。

➤ 系统时钟的初始化

在任务 `usrRoot()` 中的第一个动作是初始化 VxWorks 时钟，通过调用 `sysClkConnect()` 将系统时钟中断向量与中断处理程序 `usrClock()` 建立连接。而后，系统时钟的频率（通常为 60Hz）由 `sysClkRateSet()` 设置。多数板允许时钟频率小于 30Hz（有些甚至小于 1Hz）或大于几千 Hz。但通常不希望使用高于 1000Hz 的时钟频率，因为这样会加重系统的额外开销。当时钟频率太快时，处理器要花费很多时间来处理中断，而应用程序因抢不到 CPU 得不到运行。

定时器驱动程序由 WRS 公司提供，包括一个 `sysHwInit2()` 调用，它是 `sysClkConnect()` 的一部分。BSP 使用 `sysHwInit2()` 完成对目标板的进一步初始化，而这些初始化在函数 `sysHwinit()` 是没有完成的。

➤ I/O 系统的初始化

如果在 `configAll.h` 中定义了 `INCLUDE_IO_SYSTEM`，那么通过调用 `iosInit()` 程序进行 VxWorks 的 I/O 系统初始化。调用参数指明了安装设备驱动的最大数目，在系统中能够同时打开的最大文件数目，在 VxWorks 的 I/O 系统中所希望的设备名称。

包含与不包含 `INCLUDE_IO_SYSTEM` 也影响是否创建控制台设备、标准输入、标准输出、标准错误的建立，更多信息参见下面两节。

➤ 控制台设备的创建

如果在配置中包括了 `INCLUDE_TTY_DEV` 板上串口设备驱动，就可以通过函数 `ttyDrv()` 调用初始化安装在系统的 I/O 中设备驱动程序。实际设备的建立与命名是通过设备创建程序完成，典型调用为 `ttyDevCreate()`，这个程序的调用参数包括设备名，一个串行 I/O 通道的描述器，输入输出缓冲区的大小。

宏 `NUM_TTY` 指定了 TTY 端口的数目（默认是 2），`CONSOLE_TTY` 指定了控制台的端口（默认为 0），`CONSOLE_BAUD_RATE` 指定了波特率（默认为 9600）。这些宏在文件 `configAll.h` 中定义，但是对于非标准端口的板可以在 `config.h` 文件中重写。



➤ 标准输入、输出、错误的设置

这些功能的分配是由操作控制台和调用 `ioGlobalStdSet()` 来建立的。这是 VxWorks 作为默认设备用来与应用开发系统进行通信，并将控制台作为一个交互式终端，通过设置选项 `OPT_TERMINAL`，调用 `ioctl()` 进行设置。

➤ 异常处理与登录的安装

VxWorks 的异常处理服务（由模块 `excLib` 提供）和登录服务（由 `logLib` 提供）的初始化早在根任务执行前进行的。它们检查根任务本身或者各种服务的初始化过程中的程序错误。

当 `INCLUDE_EXC_HANDLING` 和 `INCLUDE_EXC_TAS` 在配置文件中定义后，异常处理服务通过调用 `excInit()` 进行初始化。`excInit()` 函数启动异常支持任务 `excTask()`。这个初始化之后硬件异常造成的程序错误能被安全地捕获和报告，并且没有被初始化的硬件中断向量也能进行处理。VxWorks 通过发信号量来使用指定的异常处理任务，该信号量由 `sigInit()` 完成初始化，当然必须定义 `INCLUDE_SIGNALS` 选项。

当 `INCLUDE_LOGGING` 被定义后，登录服务由调用 `logInit()` 进行初始化。参数指定了登录信息写入的设备文件描述符，分配登录信息缓冲区的个数。登录初始化过程也包括启动一个登录任务 `logTask()`。

➤ 管道设备驱动的初始化

如果希望有一个命名的管道，那么在文件 `configAll.h` 中定义 `INCLUDE_PIPE` 选项，管道驱动程序 `pipeDrv()` 自动被调用，进而初始化管道服务；管道必须用 `pipeDevCreate()` 来创建，而后，任务就能够使用管道相互通信。

➤ 文件系统设备驱动的创建和设备驱动器程序的初始化

许多 VxWorks 配置至少包括一个磁盘或 RAM 盘，采用 `dosFs`、`rt11Fs` 或 `rawFs` 文件系统。首先磁盘驱动程序通过调用驱动初始化程序进行安装；其次设备创建程序定义一个设备，这个调用返回指向描述该设备的数据结构 `BLK_DEV`。

而后，新的设备能够通过调用系统设备初始化程序（`osFsDevInit()`、`rt11FsDevInit()`、`rawFsDevInit()`）进行初始化和命名，各种文件格式由 `INCLUDE_DOSFS`、`INCLUDE_RT11FS` 和 `INCLUDE_RAWFS` 定义（在设备初始化前，文件系统模块应先采用 `osFsDevInit()`、`rt11FsDevInit()`、`rawFsDevInit()` 进行初始化）。文件系统设备初始化参数与特殊的文件系统相关，典型地包括设备名、一个指向由设备创建程序建立的数据结构 `BLK_DEV`，还有一些特定的文件系统配置参数。

➤ 浮点支持的初始化

在文件 `configAll.h` 中定义 `NCLUDE_FLOATING_POINT` 后，浮点 I/O 支持由调用



floatInit()进行初始化, 当包含了 INCLUDE_HW_FP 时, 支持浮点的协处理器由调用 mathHardInit()进行初始化。

➤ 网络初始化

在使用网络前, 它必须使用 usrNetInit()进行初始化。当一个头文件中包含 INCLUDE_NET_INIT 时, usrRoot()将调用该函数(usrNetInit 的源程序在 src/config/usrNetwork.c 文件中), 以配置字符行为参数。配置字符行通常称为引导行, 它在 VxWorks 的 ROM 引导系统中指定。基于这个字符行, usrNetInit()执行以下网络初始化:

- 调用程序 netLibInit()初始化网络子系统。
- 连接和配置适当的网络驱动程序。
- 添加网关通道。
- 初始化远程文件存取驱动程序 netDrv, 并添加远程文件存取设备。
- 初始化远程登录工具。
- 可选择地初始化远过程调用服务 (RPC)。
- 可选择地初始化网络文件系统服务 (NFS)。

➤ 可选择产品与其他服务的初始化

共享存储器由可选产品 VxMP 提供, 在使用共享存储器的目标之前, 必须用程序 usrSmObjInit()(在文件 src/config/usrSmObj.c)对它们进行初始化, 如果包括了 INCLUDE_SM_OBJ, 则由 usrRoot()调用; 如果选项 INCLUDE_MMU_BASIC 被定义, 则提供基本的 MMU 支持。如果 INCLUDE_MMU_FULL 被定义, 可选配件 VxVMI 提供了程序保护。向量表保护和虚拟内存接口。MMU 由程序 usrMmuInit()初始化, 该程序在文件 src/config/usrMmuInit.c 中。如果宏 INCLUDE_PROTECT_TEXT 和 INCLUDE_PROTECT_VEC_TABLE 被定义, 则初始化程序保护和向量表保护。

Tornado 移植了 GNU C++ 编译器, 对于 GNU 编译器或可选的 CenterLine 编译器, 要初始化支持 C++, 由 INCLUDE_CPLUS 或 INCLUDE_CPLUS_MIN 定义。

要包括一个或更多类库, 定义适当的选项 INCLUDE_CPLUS_library。

➤ WindView 的初始化

内核分析器由可选的 WindView 提供, 当在文件 configAll.h 中定义 INCLUDE_INSTRUMENTATION 时, 由程序 usrRoot()初始化。

➤ 目标机调试机构的初始化

如果定义 INCLUDE_WDB, 在文件 src/config/usrWdb.c 中的 wdbConfig()被调用。该程序初始化通信接口, 然后运行调试机构。配置调试机构的信息和它的初始化序列, 参见 Tornado 用户指南: Getting Started。



➤ 启动脚本的执行

如果将驻留目标系统的命令解释程序配置到 VxWorks 中, `usrRoot()` 函数运行用户提供的启动脚本, 定义宏 `INCLUDE_STARTUP_SCRIPT`, 并且脚本的文件名在引导时指定, 以及启动脚本的参数 (参见用户指南: `Getting Started`); 如果没有参数, 则脚本不被执行。

10.4.5 初始化序列总结

典型配置的 VxWorks 整个实时系统初始化序列包括以下几方面。

(1) 系统初始化函数 `sysInit()`, 在文件 `sysALib.s` 中有:

- 关闭所有中断。
- 禁止高速缓冲存储器。
- 初始化系统的中断表。
- 初始化系统默认的表。
- 初始化 CPU 寄存器为已知默认值。
- 禁止跟踪。
- 清除所有未处理的中断。
- 用指定的引导类型调用 `usrInit()` 函数。

(2) `usrInit()`, 该函数在文件 `usrConfig.c` 中, 包括:

- 将 `bss` 未初始化的段清 0。
- 将引导? 类型保存到 `sysStartType`。
- 调用异常向量初始化函数 `excVecInit()`, 初始化所有系统和默认的中断向量。
- 调用系统的硬件设备初始化函数 `sysHwInit()`。
- 调用核心库初始化函数 `usrKernelInit()`。
- 调用内核初始化函数 `kernelInit()`。

(3) `usrKernelInit()`, 该函数在文件 `usrKernel.c` 中, 如果定义了相应的配置, 则调用以下的子程序:

- `classLibInit()`。
- `taskLibInit()`。
- `taskHookInit()`。
- `semBLibInit()`。
- `semMLibInit()`。
- `semCLibInit()`。
- `semOLibInit()`。
- `wdLibInit()`。



- ◆ msgQLibInit()。
- ◆ qlInit() for all system queues。
- ◆ workQInit()。

(4) kernelInit() 初始化并启动内核:

- ◆ 调用 intLockLevelSet()。
- ◆ 在存储池高端创建根任务和任务控制块 TCB。
- ◆ 调用任务初始化函数 taskInit()。
- ◆ 调用任务激活函数 taskActivate()。
- ◆ 转到函数 usrRoot()。

(5) usrRoot() 初始化 I/O 系统, 安装设备的驱动, 根据配置文件 configAll.h 和 config.h 中的配置, 创建指定的设备。

- ◆ 调用函数 sysClkConnect(), 建立系统时钟中断处理。
- ◆ 调用函数 sysClkRateSet(), 初始化系统时钟频率。
- ◆ 调用函数 iosInit(), 初始化 I/O 设备控制块。
 - _ 如果在配置中定义 INCLUDE_TTY_DEV 和 NUM_TTY, 则函数调用 ttyDrv(), 建立控制台标准输入、标准输出、标准错误接口: STD_IN, STD_OUT, STD_ERR。
 - _ 初始化异常处理, 调用函数 excInit(), logInit(), sigInit()。
 - _ 调用函数 pipeDrv() 初始化管道设备。
 - _ 标准 I/O 库初始化函数 stdioInit()。
 - _ 数学函数库初始化, 软件仿真 mathSoftInit() 或 硬件支持 mathHardInit()。
 - _ 调用函数 wdbConfig() 进行配置并初始化目标机调试器。
 - _ 如果配置了驻留目标机的命令解释, 则启动该脚本。

10.4.6 系统时钟程序: usrClock()

最后, 系统时钟中断服务程序 (ISR) usrClock() 连接到系统时钟中断, 它是由初始任务 usrRoot() 完成的。usrClock() 程序调用内核时间片程序 tickAnnounce() 完成 OS 的计时。你可以在该程序中增加指定的应用程序。

10.5 可选的 VxWorks 配置

了解 VxWorks 的配置文件与配置选项后, 当你设计一个应用产品的 VxWorks 映像时, 并不需要将所有选项的服务都集成到你的应用系统, 或许是目标环境所限, 你总想通过一种



或多种方法修改 VxWorks 配置，以便满足你的使用要求。

- 改变目标机调试机制的配置。
- 缩小 VxWorks 的大小。
- 从 ROM 中运行 VxWorks。

10.5.1 裁剪 VxWorks

在一个产品的配置中，为了减少系统对存储器的需求，缩短引导时间或者处于安全目的，通常希望去掉某些 VxWorks 服务。

可选的 VxWorks 服务通过在头文件 `configAll.h` 或 `config.h` 加注释或使用 `#undef` 进行省略。例如，登录服务可以用 `#undef INCLUDE_LOGGING` 进行省略，信号量服务可以用 `#undef INCLUDE_SIGNALS` 进行省略。

VxWorks 的结构很容易去掉你不需要的服务，然而，并非每一个 BSP 都采用这个方法。如果你希望应用系统最小化，只是在 VxWorks 配置文件中没有定义你所不需要的选项是不够的，应仔细检查你的 BSP 代码，去除你不要包括的服务。

■ 去除内核服务

下面在文件 `configAll.h` 中包含的定义是可选的，因为涉及的任何与内核相关的服务应用系统都自动包括：

- `INCLUDE_SEM_BINARY`。
- `INCLUDE_SEM_MUTEX`。
- `INCLUDE_SEM_COUNTING`。
- `INCLUDE_MSG_Q`。
- `INCLUDE_WATCHDOGS`。

这些配置常量出现在 VxWorks 默认配置中，以保证所有内核服务都能配置在系统内，即使应用系统没有涉及到这些服务。然而，你的目标是实现尽可能小的应用系统，应该去除这些常量，保证你不使用这些服务内核时就不包括这些服务。

这里有另外两个配置常量控制可选的 VxWorks 内核服务，`INCLUDE_TASK_HOOKS` 和 `INCLUDE_CONSTANT_RDY_Q`。如果一个应用既需要内核的标注也需要一个恒插入时间的基于优先权就绪队列，则在文件 `configAll.h` 定义这些选项。在系统中，一个有恒定插入时间的就绪队列是不管任务的数目，都允许以固定的系统开销进行上下文切换。否则，在系统中，最坏的性能是线性地与就绪队列的数目成比例下降。注意具有恒定插入时间的就绪队列使用 2K 字节的数据结构，有些系统没有足够内存来满足。在那种情况下，`INCLUDE_CONSTANT_RDY_Q` 的定义可以省略，而允许使用小的（小于确定的）就绪队列机制。



■ 去掉网络服务

在某些应用，去掉 VxWorks 网络服务是合适的。例如，基于 ROM 的系统或是一个独立配置，或许不需要网络服务。

要去掉网络服务，确信以下常量不被定义：

- ◆ INCLUDE_NETWORK。
- ◆ INCLUDE_NET_INIT。
- ◆ INCLUDE_NET_SYM_TBL。
- ◆ INCLUDE_NFS。
- ◆ INCLUDE_RPC。
- ◆ INCLUDE_RDB。

要去掉远程过程调用 (RPC)，不要定义 INCLUDE_RPC。

■ 从属选项

从属选项的代码在文件 target/src/config/usrDepend.c 中，所以，当一个特殊选项选中时，每一项必须包括进去。这能使你以最小的努力保证你的系统工作。尽管你能在文件 config.h 和 configAll.h 去掉不需要的特性，由于相关性，你应该意识到某些情况他们不能去掉。

例如，你不能在没有运行网络的情况下使用 telnet。因此，如果在文件 configAll.h 中选择了包含 telnet 的选项 INCLUDE_TELNET，但是没有选网络初始化选项 INCLUDE_NET_INIT，当你生成操作系统时，usrDepend.c 文件根据 INCLUDE_TELNET 选项，将为自动包含网络初始化选项 INCLUDE_NET_INIT，又由于网络初始化需要网络软件，文件 usrDepend.c 也将自动定义 INCLUDE_NETWORK。

10.5.2 从 ROM 中运行 VxWorks

你可以将 VxWorks 或一个 VxWorks 应用固化在 ROM 中，此时 ROM 引导的配置模块是文件 config/all/bootConfig.c，而不是 VxWorks 开发系统提供的默认配置文件 usrConfig.c。举一个基于 ROM 的 VxWorks 应用例子，看看 VxWorks 的 ROM 引导程序。

在 ROM 配置中，引导程序或 VxWorks 映像的程序与数据段首先拷贝到系统 RAM 中，在 RAM 中执行引导过程或 VxWorks。一旦系统存储器资源匮乏，它可能拷贝仅保存在 RAM 中的数据段，程序段保留在 ROM 中，并且从 ROM 中的地址运行，术语称为驻留 ROM。在 RAM 中将被程序段占用的内存对应用来说是可用的（对一个单独 VxWorks 系统可达到 300KB），注意基于 ROM 的 VxWorks 并不支持所有目标板。

基于 ROM 的程序的缺点是限制了数据的宽度和降低内存的访问时间，程序在 ROM 中运行比在 RAM 中运行慢得多。有时使用快速 EPROM 或重新配置去掉不必要的系统特性来



提高执行速度。

除了程序不拷贝到 RAM, ROM 引导 VxWorks 的驻留 ROM 版本和单独 VxWorks 系统是同一种常规版本。另一种是以压缩的驻留 ROM 的版本, 在 VxWorks 目标机配置子目录的 makefiles 文件中包括生成这些映像, 参见表 10.2。

表 10.2 生成的驻留 ROM 映像文件

文件名	描述
bootrom_res	驻留 ROM 的引导 ROM 映像
bootrom_res.hex	驻留 ROM 的引导 ROM 映像, 该映像是 Motorola 的 S 记录格式或 Intel 的 HEX 格式
VxWorks.res_rom	非压缩的驻留 ROM 的 VxWorks 独立版
VxWorks.res_rom.hex	非压缩的驻留 ROM 的 VxWorks 独立版, 该映像是 Motorola 的 S 记录格式或 Intel 的 HEX 格式
VxWorks.res_rom_nosym	非压缩或无符号表的驻留 ROM 的 VxWorks 映像

由于系统映像大小缘故, 对于驻留 ROM 的单独的 VxWorks 系统建议使用 512KB 的 EPROM, 如果与应用软件连接则需要更大空间。对于 ROM 引导的 ROM 驻留版本, 推荐使用 256KB 的 EPROM, 如果你使用非默认的 ROM 大小, 在目标机文件 Makefile 和 config.h 中修改 ROM_SIZE 的值。

一个新生成的目标系统 VxWorks.res_rom_nosym, 它创建并提供一个没有符号表的驻留 ROM 版本映像。这是使用 Tornado 开发环境生成的标准的 ROM 映像, 符号表留在宿主机上, 由于调试机制和 VxWorks 驻留 ROM, 在系统加电或复位后立即准备就绪。

只把 VxWorks 驻留 ROM 的数据段加载到 Makefile 文件定义的 RAM_LOW_ADRS 内存地址, 以减小碎片。以 ROM 引导驻留 ROM 的数据段被加载到 RAM_HIGH_ADRS, 所以加载 VxWorks 没有覆盖驻留 ROM 引导部分。对于一个小于 1MB 存储器的 CPU 板, 确保 RAM_HIGH_ADRS 要小于 LOCAL_MEM_SIZE, 留有足够的空间适应数据段要求。注意 RAM_HIGH_ADRS 在 Makefile 和 config.h 都要定义, 并且要一致。

图 10.1 表示 ROM 驻留引导和 VxWorks 映像的存储器划分, 框图低端部分表示 ROM 的划分, 上部表示 RAM 的划分。常量 LOCAL_MEM_LOCAL_ADRS 是 RAM 的开始地址。对于引导映像, 数据段拷贝到 RAM 中的 RAM_HIGH_ADRS 内存地址上面。对于 VxWorks 映像, 数据段拷贝到 RAM_LOW_ADRS 内存地址的上面。注意有两种映像程序段保留在 ROM 中。

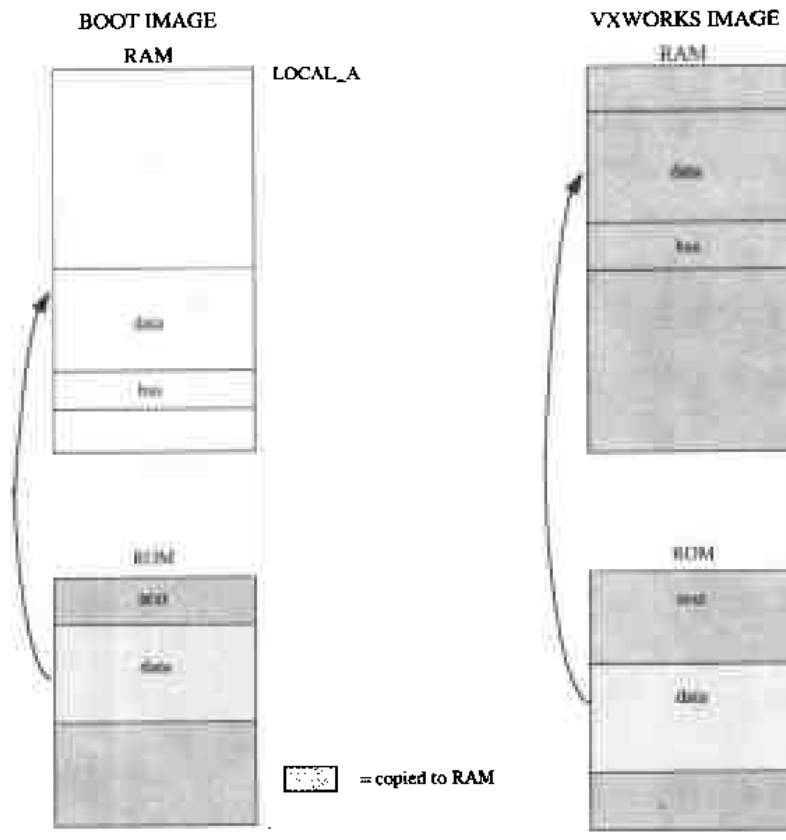


图 10.1 驻留 ROM 的内存划分

10.5.3 基于 ROM 的 VxWorks 初始化序列

系统初始化开始几步与基于 ROM 版本的 VxWorks 有些不同：多数目标机，执行两个函数 `romInit()` 和 `romStart()`，而不是通常 VxWorks 的入口 `sysInit()`。

■ ROM 入口点：`romInit()`

加电后，处理器开始执行 `romInit()`（在 `config/bspname/romInit.s` 定义）。程序 `romInit()` 禁止中断，将启动类型（冷/热）压栈，做硬件相关初始化（如清 caches 或允许 DRAM），且转到 `romStart()`。堆栈指针初始化为在数据部分的后面。

■ 拷贝 VxWorks 映像：`romStart()`

下一步，`romStart()` 程序（在文件 `config/all/bootInit.c` 中）加载 VxWorks 映像到 RAM 中，



如果选择 VxWorks 的 ROM 驻留版本, 数据段将从 ROM 中拷贝到 RAM 中, 且存储器清零。如果 VxWorks 不是在 ROM 驻留, 所有程序和代码段将被拷贝并且从 ROM 中解压到 RAM 中, 定位到在 Makefile 中定义的 RAM_HIGH_ADRS 内存地址。如果 VxWorks 既不是 ROM 驻留也不是压缩的, 则整个程序和数据段直接拷贝到 RAM, 并定位到在 Makefile 定义的 RAM_LOW_ADRS 内存地址。

■ 基于 ROM 的 VxWorks 的所有初始化

除了 romStart() 之外, 基于 ROM 的 VxWorks 初始化序列与通常一样, 从 usrInit() 调用继续运行。

下面总结了全部初始化序列。对于 romInit() 和 romStart() 之后详细步骤参见 10.4.5 节。

(1) romInit()

- ◆ 禁止中断。
- ◆ 保存启动类型 (冷/热)。
- ◆ 硬件相关初始化。
- ◆ 转到函数 romStart()。

(2) romStart()

- ◆ 从 ROM 中将数据段拷贝到 RAM, 将内存清 0。
- ◆ 从 ROM 中将代码段拷贝到 RAM, 如果是压缩的方式, 则解压缩。
- ◆ 根据启动类型调用函数 usrInit()。

(3) usrInit() 初始化程序

这一章我们介绍一些编程实战经验，结合前面介绍的内容，读者可以通过本章提供的程序了解 VxWorks 编程方法。

11.1 程序执行时间

实时系统应用程序必须优化执行的时间，编程者应该清楚知道自己编写的每段代码的执行时间。VxWorks 提供了一组调用：`timex()`和 `timexN()`可以测量应用程序的执行时间。对于执行时间非常短的程序，`timexN()`可以通过重复执行该程序来完成计时。

这一节，我们介绍使用 VxWorks `timex()`调用测量程序执行时间的方法。

`timex()`调用测量一个程序的单次执行时间，同时允许向该程序传递八个参数。当执行完成时，`timex()`调用显示程序的执行时间和测量误差。如果被测试程序执行太快，比系统时钟速率还快，测量误差大于 50%，测量就没有意义，这时会显示一个警告信息。对于这种情况，应该使用 `timexN()`测试该程序多次执行的时间。

`timex()`调用原型：

```
void timex(  
    FUNCPTR function_name,      /*待测量的函数名*/  
    int arg1,                   /* 传递给待测量函数的参数 */  
    ...  
    int arg8  
)
```



这里给出的小例子有两个子程序。第一个子程序 `timing()` 调用 `timex()` 来测量第二个子程序 `printit()` 的执行时间，没有参数传递给 `printit()`。`printit()`（调用 `taskIdSelf()`）重复 200 次显示其任务 id 号并且将变量 `i` 递增。

例 11.1 测量程序执行时间

```
/*
*****
#include "vxWorks.h" /* Always include this as the first thing in every program */
#include "timexLib.h"
#include "stdio.h"

#define ITERATIONS 200

int printit(void);

void timing()/* Function to perform the timing */
{
    FUNCPTR function_ptr = printit; /* a pointer to the function "printit" */
    timex(function_ptr,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL); /* Timing the "print"
function */
}

int printit(void) /* Function being timed */
{
    int i;
    for(i=0; i< ITERATIONS; i++) /* Printing the task id number and the increment variable
    "i" */
        printf("Hello, I am task %d and is i = %d\n",taskIdSelf(),i);
    return 0;
}
*****
*/
```

使用方法：

- (1) 编译代码。
- (2) 下载目标码到目标机。
- (3) 在 WindSh 中敲入 `timing`，运行代码。

注意，如果目标机没有显示输出，就需要重定向 I/O，否则将看不到输出结果。



我们可以通过改变 ITERATIONS 的值，程序的循环的次数 (300,400,500,600,700)，观察程序执行时间的变化；当减少循环的次数到 5 次，我们将得到警告信息，可以使用 `timexN()` 重写这个程序，测量 `printit` 的执行时间。

`timex()` 测量的是程序体的执行时间，不包含通常的子程序进入和退出代码执行时间。而且，也不包括参数传递和函数调用的时间。这是因为计时函数 `timex()` 自动减去这些时间。

更多的信息请参考《VxWorks Programmer's Manual and Reference Manual: (timexLib)》

11.2 多 任 务

现代实时系统基于多任务和任务间通信的概念，一个多任务环境允许实时应用构造成多个独立任务组成的集合，每个任务单独执行，拥有自己的一套资源。任务间通信功能允许这些任务间同步以协调他们的活动。

VxWorks 多任务内核 `wind` 使用中断驱动，基于优先级的任务调度机制，具有较快的上下文切换时间和较低的中断延迟。

这一节介绍利用 Vxworks 任务管理调用创建多任务和多任务间通信的例子。

多任务制造一种多个线程同时执行的假象。事实上，内核基于调度算法插入到这些任务的执行中。每个单独执行的程序称作一个“任务”。每个任务拥有自己的上下文，包括 CPU 环境和系统资源，这些是内核调度该任务运行时所必需的。

在上下文切换时，任务的上下文保存在任务控制块中 (TCB)。一个任务的上下文包括：

- ◆ 代码执行点，也就是任务的程序计数器。
- ◆ CPU 寄存器和浮点数寄存器（如果需要）。
- ◆ 动态变量和函数调用的堆栈。
- ◆ 标准输入输出和错误的 I/O 分配。
- ◆ 一个延时定时器。
- ◆ 一个时间片定时器。
- ◆ 内核控制结构。
- ◆ 信号处理器。
- ◆ 调试和性能监视值。

➤ 任务创建和激活

函数 `taskSpawn()` 创建一个新的任务上下文，包括为程序执行分配和建立任务环境，传递参数，新任务的入口由第五个参数指定。其他参数包括任务名、优先级、一个“options”字、堆栈大小以及传递给入口函数的 10 整形参数。返回值为任务 id 号。



➤ 语法

`id = taskSpawn(name,priority,options,stacksize,function, arg1,...arg10);`

➤ 例子

示例程序创建十个任务，每个任务打印输出自己的任务 id 号。

例 11.2 多任务示例

```
/*-----*/  
  
#define ITERATIONS 10  
  
void print(void);  
  
spawn_ten()/* Subroutine to perform the spawning */  
{  
int i, taskId;  
for(i=0; i < ITERATIONS; i++) /* Creates ten tasks */  
    taskId = taskSpawn("tprint",90,0x100,2000,print,0,0,0,0,0,0,0,0,0);  
}  
  
void print(void) /* Subroutine to be spawned */  
{  
printf("Hello, I am task %d\n",taskIdSelf()); /* Print task Id */  
}  
/*-----*/
```

使用方法：

- (1) 编译代码。
- (2) 下载目标码到目标机。
- (3) 在 WindSh 中敲入 `spawn_ten`，运行代码。

正如例子中看到的那样，在创建任务时最多允许传递 10 个参数给任务。例子中，在调用 `taskSpawn()` 时，传递给十个任务的唯一的一个参数皆不相同，每个任务输出传递给它的参数；请思考：当对每个任务分配不同的优先级时，输出结果将是怎样的顺序？解释原因。



11.3 信 号 量

信号量允许多个任务相互协调其活动。任务间最直接的通信方式是共享各式各样的数据结构。由于 VxWorks 中，所有任务存在于一个单一的线性地址空间，共享数据结构也就非常容易实现。全局的变量、线性缓冲、环形缓冲、连结链和指针都可以被运行在不同上下文的代码直接引用。

然而，共享地址空间简化数据交换的同时，需要保证这块内存的互斥访问。VxWorks 提供了许多实现共享临界区互斥访问的机制，信号量就是其中一种。

这一节的例子演示了 VxWorks 信号量的使用。

VxWorks 信号量是任务通信最优选择，同时也提供了最快的通信方式。信号量是实现任务间同步和互斥最直接的方式。

Wind 内核提供三种类型的信号量：用于解决不同类型的问题：

- ◆ 二进制信号量 (binary)：最快的、最普通的信号量，适合于实现同步，也适合于互斥。
- ◆ 互斥信号量 (mutual exclusion)：一种特殊的二进制信号量，适于解决具有内在互斥的问题：优先级继承、删除安全和递归。
- ◆ 计数信号量 (counting)：类似于二进制信号量，但是可以记录信号量释放的次数。适于保护对一类包含多个数目的资源的访问。

➤ 信号量控制

Wind 内核对上述三类信号量提供统一的调用接口。信号量的类型在创建时说明。

- ◆ `semBCreate(int options, SEM_B_STATE initialState)`：分配并初始化一个二进制信号量。
- ◆ `semMCreate(int options)`：分配并初始化一个互斥信号量。
- ◆ `semCCreate(int options, int initialCount)`：分配并初始化一个计数器信号量。
- ◆ `semDelete(SEM_ID semId)`：终止并释放一个信号量。
- ◆ `semTake(SEM_ID semId, int timeout)`：取一个信号量。
- ◆ `semGive(SEM_ID semId)`：释放一个信号量。
- ◆ `semFlush(SEM_ID semId)`：解锁所有等待该信号量的任务。

➤ 二进制信号量使用的例子

二进制信号量可以作为资源可用或不可用的标志。当任务取一个二进制信号量时要调用 `semTake()`，结果取决于调用时该二进制信号量是否可用。如果可用，信号量将变得不可用，而任务继续执行。如果信号量不可用，任务被挂起到任务阻塞队列，直到该信号量



可用。

当任务释放一个二进制信号量时要调用 `semGive()`，结果也要依赖于调用时刻该信号量是否可用。如果可用，本次释放信号量不起任何作用；如果信号量不可用，并且没有任务在等待该信号量，那么信号量变为可用；如果信号量不可用，并且有一个或多个任务在等待该信号量，那么阻塞队列中的第一个任务解除阻塞，而信号量仍不可用。

下面的例子中，两个任务（`taskOne` and `taskTwo`）竞争修改一个全局变量 `global` 的值，任务 `taskOne` 将 `global` 修改为 1，而任务 `taskTwo` 则将其值改为 0。

如果不是用信号量，`global` 的值将是随机的。

例 11.3 二进制信号量使用的例子

```
/* **** */
/* includes */
#include "vxWorks.h"
#include "taskLib.h"
#include "semLib.h"
#include "stdio.h"

/* function prototypes */
void taskOne(void);
void taskTwo(void);

/* globals */
#define ITER 10
SEM_ID semBinary;
int global = 0;

void binary(void)
{
    int taskIdOne, taskIdTwo;

    /* create semaphore with semaphore available and queue tasks on FIFO basis */
    semBinary = semBCreate(SEM_Q_FIFO, SEM_FULL);

    /* Note 1: lock the semaphore for scheduling purposes */
    semTake(semBinary, WAIT_FOREVER);
```



```

/* spawn the two tasks */
taskIdOne = taskSpawn("t1",90,0x100,2000,(FUNCPTR)LaskOne,0,0,0,0,0,0,0,0,0,0);
taskIdTwo = taskSpawn("t2",90,0x100,2000,(FUNCPTR)taskTwo,0,0,0,0,0,0,0,0,0,0);
}

void taskOne(void)
{
int i;
for (i=0; i < ITER; i++)
    {
semTake(semBinary,WAIT_FOREVER); /* wait indefinitely for semaphore */
printf("I am taskOne and global = %d.....\n", ++global);
semGive(semBinary); /* give up semaphore */
    }
}

void taskTwo(void)
{
int i;
semGive(semBinary); /* Note 2: give up semaphore(a scheduling fix) */
for (i=0; i < ITER; i++)
    {
semTake(semBinary,WAIT_FOREVER); /* wait indefinitely for semaphore */
printf("I am taskTwo and global = %d-----\n", --global);
semGive(semBinary); /* give up semaphore */
    }
}
/*****/

```

使用方法:

- (1) 编译代码。
- (2) 下载目标码到目标机。
- (3) 在 WindSh 中敲入 binary, 运行代码。

请思考:

(1) 在上面的例子中, 不使用注释 1 和注释 2 所在的两行代码, 重新编译、下载并运行, 输出结果将如何, 并解释原因。

(2) 在上面的例子中, 如果使用的是计数信号量, 作适当修改, 输出结果将如何, 并



解释原因。

11.4 消息队列

VxWorks 在单 CPU 中，多任务通信的主要机制是消息队列。消息队列允许以 FIFO 或基于优先级方式排队消息，消息的数目可变，消息的长度可变。任何任务都可以向消息队列发送消息，也可以从消息队列接收消息。多个任务允许从同一个消息队列收发消息。但是，两个任务间的双向通信通常需要两个消息队列，各自用于一个方向。

本节的例子主要演示 VxWorks 消息队列的使用。

VxWorks 消息队列的创建、删除、发送和接收调用如下：

- ◆ `msgQCreate(int maxMsgs, int maxMsgLength, int options)`: 分配并初始化一个消息队列。
- ◆ `msgQDelete(MSG_Q_ID msgQId)`: 终止并释放一个消息队列。

`msgQSend(MSG_Q_ID msgQId, char *Buffer, UINT nBytes, int timeout, int priority)`: 向一个消息队列发送一个消息。

`msgQReceive(MSG_Q_ID msgQId, char *Buffer, UINT nBytes, int timeout)` 从一个消息队列接收一个消息。

消息队列库允许消息按照先进先出方式排队，但是也有一个例外：存在两个优先级，优先级最高的消息排在队列的头部。

要创建一个队列可以调用 `msgQCreate()`。消息的最大数目以及一个消息的最大长度由其参数说明。

任务调用 `msgQSend()` 将消息发送到一个消息队列，如果没有任务在等待该队列的消息，那么这条消息增加到该队列的消息缓冲中，如果有任务在等待，那么该消息立即提供给第一个等待的任务。

任务如果需要从一个消息队列接收一条消息，它应该调用 `msgQReceive()`。如果该消息队列中已有消息可用，那么队列中的第一条消息立即出队，并提交给调用任务；如果没有消息可用，那么调用任务阻塞，并且加入到等待该消息的任务队列中。等待任务队列可以按两种方式排队：基于任务优先级或基于 FIFO 方式，由消息队列创建时指定。

- ◆ **Timeouts**: 函数 `msgQSend()` 和 `msgQReceive()` 都可以说明一个超时参数，规定任务等待的时间（tick 数）：发送消息任务等待队列空间可用，接收消息任务等待消息可用。
- ◆ **Urgent Messages**: 函数 `msgQSend()` 可以指定欲发送消息的优先级：正常 `MSG_PRI_NORMAL` 或紧急 `MSG_PRI_URGENT`。正常优先级的消息将加入到消息队列的尾部，而紧急优先级的消息将增加到消息队列的头部。



➤ 示例

例 11.4 VxWorks 消息队列的使用

```
/* **** */
/* includes */
#include "vxWorks.h"
#include "msgQLib.h"

/* function prototypes */
void taskOne(void);
void taskTwo(void);

/* defines */
#define MAX_MESSAGES 100
#define MAX_MESSAGE_LENGTH 50

/* globals */
MSG_Q_ID msgQueueId;

void message(void) /* function to create the message queue and two tasks */
{
    int taskIdOne, taskIdTwo;

    /* create message queue */
    if((msgQueueId= msgQCreate(MAX_MESSAGES, MAX_MESSAGE_LENGTH, MSG_Q_FIFO))
        == NULL)
        printf("msgQCreate in failed\n");

    /* spawn the two tasks that will use the message queue */
    if((taskIdOne = taskSpawn("t1", 90, 0x100, 2000, (FUNCPTR)taskOne, 0, 0, 0, 0, 0, 0,
        0, 0, 0)) == ERROR)
        printf("taskSpawn taskOne failed\n");
    if((taskIdTwo = taskSpawn("t2", 90, 0x100, 2000, (FUNCPTR)taskTwo, 0, 0, 0, 0, 0, 0,
        0, 0, 0)) == ERROR)
        printf("taskSpawn taskTwo failed\n");
}
```



```
void taskOne(void) /* task that writes to the message queue */
{
char message[] = "Received message from taskOne";

/* send message */
if((msgQSend(msgQueueId,message,MAX_MESSAGE_LENGTH, WAIT_FOREVER,
MSG_PRI_NORMAL)) == ERROR)
printf("msgQSend in taskOne failed\n");
}

void taskTwo(void) /* tasks that reads from the message queue */
{
char msgBuf[MAX_MESSAGE_LENGTH];

/* receive message */
if(msgQReceive(msgQueueId,msgBuf,MAX_MESSAGE_LENGTH, WAIT_FOREVER)
== ERROR)
printf("msgQReceive in taskTwo failed\n");
else
printf("%s\n",msgBuf);
msgQDelete(msgQueueId); /* delete message queue */
}
/*****/
```

使用方法:

- (1) 编译代码。
- (2) 下载目标码到目标机。
- (3) 在 WindSh 中敲入 message, 运行代码。

请思考:

- (1) 修改上面的例子, 使任务 taskTwo 能够向任务 taskOne 发送字符串消息 (“Received message from taskTwo”), 任务 taskOne 向控制台输出来至于 taskTwo 的消息。
- (2) 比较消息队列与全局变量用于任务间通信的优缺点?



11.5 轮转调度算法

任务调度就是基于某种规则约束，给一个任务集中的每个任务分配开始和结束时间。约束一般包括时间约束和资源约束。在一个时间共享（time-sharing）操作系统中，系统按照时间片依次轮流每个任务（或进程），因此，制造一种多个任务在单个处理器上同时执行的假象。

Wind 内核调度默认使用基于优先级抢占式调度，但是同时也允许使用轮转调度。

本节的例子说明 VxWorks 轮转调度机制的使用。

轮转调度算法目的是使相同优先级的所有就绪任务共享 CPU。如果不使用轮转调度，当多个相同优先级的任务需要共享处理器时，其中的一个任务因为永不阻塞，可能霸占处理器，造成其他同等优先级的任务得不到运行的机会。

时间片(time slicing)是轮转调度在多个相同优先级的任务间实现 CPU 公平分配的基础。每个任务执行一段确定的时间（一个时间片）；然后另一个任务执行同样大小的一段时间，如此循环下去。这种分配是相当公平的：在这个优先级相同的任务组中的其他就绪任务没得到一个时间片之前，不会出现一个任务得到第二个时间片的情形。

在 Wind 内核中，调用函数 `kernelTimeSlice()` 将允许轮转调度，时间片的大小由参数传给它，规定了每个任务一次允许占有 CPU 的最长时间。

函数原型：

`kernelTimeSlice(int ticks)`: 控制轮转调度，参数是时间片，以系统 tick 为单位。

► 使用轮转调度的例子：

在下面的例子中，三个优先级相同的任务向控制台输出他们的任务 id 号和任务名。如果不使用轮转调度，那么首先获得 CPU 控制权的任务将独占 CPU。直到它执行完成，其他的任务才能得到运行的机会。在本例中，由于任务 `taskOne` 是一个死循环任务，而它又将第一个获得 CPU，因而其他的两个任务将永远得不到运行的机会。

为了使其他两个任务能得到同等的 CPU 使用权，例子中调用 `kernelTimeSlice()` 允许系统使用轮转调度。调度的时间间隔：时间片等于 `TIMESLICE`。`TIMESLICE` 的值是 1 秒钟系统时钟的 ticks 数（在本例中，在 Intel x86 目标机上，默认的设置是每秒 60 个 ticks）。函数 `sysClkRateGet()` 返回每秒的时钟 ticks 数。

在设置了调度时间片后，程序发起三个任务。还要注意以下几点：保证 `sched` 任务的优先级要比其他发起的几个任务的优先级高。如果没有特别说明，Windsh 中直接运行的任务的优先级默认是 100。例子中的三个任务的优先级定为 101，也就保证了他们的优先级比 `sched` 任务的低。

另外，还要保证提供足够的时间用于上下文交换，在例子中，使用循环语句：



```
for (j=0; j < LONG_TIME; j++);
```

使用 `printf` 不是本例的理想选择，因为它可能被阻塞。这也就可能无法充分地说明轮转调度情形。理想的选择应该是使用 `logMsg()`，它不会引起阻塞，除非 `log` 消息队列已满。

例 11.5 轮转调度算法

```
/******  
/* includes */  
#include "vxWorks.h"  
#include "taskLib.h"  
#include "kernelLib.h"  
#include "sysLib.h"  
#include "logLib.h"  
  
/* function prototypes */  
void taskOne(void);  
void taskTwo(void);  
void taskThree(void);  
  
/* globals */  
#define ITER1 100  
#define ITER2 10  
#define PRIORITY 101  
#define TIMESLICE sysClkRateGet()  
#define LONG_TIME 1000000  
  
void sched(void) /* function to create the three tasks */  
{  
int taskIdOne, taskIdTwo, taskIdThree;  
  
if(kernelTimeSlice(TIMESLICE) == OK) /* turn round-robin on */  
    printf("\n\n\n\n\t\t\tTIMESLICE = %d seconds\n\n\n", TIMESLICE/60);  
  
/* spawn the three tasks */  
if((taskIdOne=  
taskSpawn("task1", PRIORITY, 0x100, 20000, (FUNCPTR)taskOne, 0, 0, 0, 0, 0, 0,  
0, 0, 0)) == ERROR)  
    printf("taskSpawn taskOne failed\n");
```



```
if((taskIdTwo=
taskSpawn("task2",PRIORITY,0x100,20000,(FUNCPTR)taskTwo,0,0,0,0,0,0,
0,0,0)) == ERROR)
    printf("taskSpawn taskTwo failed\n");
if((taskIdThree=
taskSpawn("task3",PRIORITY,0x100,20000,(FUNCPTR)taskThree,0,0,0,0,0,0,0,
0,0,0)) == ERROR)
    printf("taskSpawn taskThree failed\n");

}

void taskOne(void)
{
int i,j;
for (i=0; i < ITER1; i++)
    {
    for (j=0; j < ITER2; j++)
        logMsg("\n",0,0,0,0,0,0); /* log messages */
        for (j=0; j < LONG_TIME; j++); /* allow time for context switch */
    }
}

void taskTwo(void)
{
int i,j;
for (i=0; i < ITER1; i++)
    {
    for (j=0; j < ITER2; j++)
        logMsg("\n",0,0,0,0,0,0); /* log messages */
        for (j=0; j < LONG_TIME; j++); /* allow time for context switch */
    }
}

void taskThree(void)
{
int i,j;
for (i=0; i < ITER1; i++)
    {
```



```
for (j=0; j < ITER2; j++)
    logMsg("\n",0,0,0,0,0,0); /* log messages */
    for (j=0; j < LONG_TIME; j++); /* allow time for context switch */

}
}
/*****/
```

使用方法:

- (1) 编译代码。
- (2) 下载目标码到目标机。
- (3) 在 WindSh 中敲入 logFdSet 1, 这将定位 logMsg()输出到虚拟控制台。
- (4) 在 WindSh 中敲入 sched, 运行代码。

请思考:

- (1) 在上面的例子中, 问什么任务 taskOne、taskTwo 和 taskThree 必须比任务 sched 的优先级低?
- (2) 改变调度时间片的值, 重新编译、运行, 并观察结果。
- (3) 增加第四个任务, 其优先级为 80, 它和其他三个任务输出相同的信息, 重新编译、运行, 并观察结果。

11.6 基于优先级的抢占式调度

这一节的例子演示 VxWorks 基于优先级的抢占式调度。

使用基于优先级的抢占式调度, 每个任务有一个优先级, 任一时刻, 内核保证将 CPU 分配给处于就绪状态的优先级最高的任务执行。之所以说这种调度算法是抢占的, 是因为, 如果在某一时刻, 一个优先级比当前正在运行的任务的优先级高的任务变为就绪, 那么内核立即保存当前任务的上下文, 然后切换到这个最高优先级任务的上下文。

Wind 内核有 256 个优先级 (0-255), 优先数 0 对应着最高优先级, 优先数 255 对应着最低优先级。任务的优先级在其创建时指定, VxWorks 也允许任务在执行时调用 taskPrioritySet()改变自身的优先级。

► 例子: 基于优先级的抢占式调度。

函数 taskSpawn()用于创建并激活一个新任务。原型如下:

```
id = taskSpawn(name, priority, options, stacksize, function, arg1, ..., arg10)。
```

任务的优先级对于自身不是特别重要, 只是相对于其他任务而言才有意义。



任务在值形式时可以调用下面的函数改变自身的优先级：

`taskPrioritySet(int tid, int newPriority)`: 改变任务的优先级。

在下面的例子中，有三个优先级各不相同的任务。程序运行的结果将是拥有最高优先级的任务 `taskThree` 首先运行完成，紧跟着优先级次高的任务 `taskTwo` 运行，等它完成后，优先级最低的任务 `taskOne` 才获得运行。

例 11.6 基于优先级的抢占式调度

```

/*****
/* includes */
#include "vxWorks.h"
#include "taskLib.h"
#include "logLib.h"

/* function prototypes */
void taskOne(void);
void taskTwo(void);
void taskThree(void);

/* globals */
#define ITER1 100
#define ITER2 1
#define LONG_TIME 1000000
#define HIGH 100 /* high priority */
#define MID 101 /* medium priority */
#define LOW 102 /* low priority */

void sched(void) /* function to create the two tasks */
{
int taskIdOne, taskIdTwo, taskIdThree;

printf("\n\n\n\n\n");
/* spawn the three tasks */
if((taskIdOne=taskSpawn("task2",LOW,0x100,20000,(FUNCPTR)taskOne,0,0,0,0,0,0,0,
0,0,0)) == ERROR)
printf("taskSpawn taskOne failed\n");
if((taskIdTwo=taskSpawn("task2",MID,0x100,20000,(FUNCPTR)taskTwo,0,0,0,0,0,0,0,
0,0,0)) == ERROR)
printf("taskSpawn taskTwo failed\n");
if((taskIdThree=

```




使用方法:

- (1) 编译代码。
- (2) 下载目标码到目标机。
- (3) 在 WindSh 中敲入 logFdSet 1, 这将定位 logMsg()输出到虚拟控制台。
- (4) 在 WindSh 中敲入 sched, 运行代码。

请思考:

(1) 在上面的例子中, 如何修改程序, 使得任务的执行顺序变为: taskOne 先执行, 然后是 taskTwo, 最后是 taskThree。

(2) 修改程序, 使得任务 taskOne 具有最高的优先级, 并且它和任务 taskTwo 以相同的优先级运行, 观察输出结果。

11.7 优先级转置

优先级转置发生在高优先级任务不得不需要等待一段不确定的时间, 等待低优先级任务完成操作。例如, prioHigh、prioMedium 和 prioLow 三个任务的优先级由高到低排列, prioLow 请求一个与二进制信号量相联系的资源, 当 prioHigh 抢占 prioLow 并且需要竞争相同的资源, 它将被阻塞。如果 prioHigh 阻塞的时间不超过 prioLow 使用该资源的时间, 也不会存在什么问题, 因为资源不会被抢占。然而, 如果这个低优先级的任务不情愿地被中优先级的任务 prioMedium 抢占, 它不需要使用该资源, 这时 prioLow 没有释放该资源。这种情况可能进一步发生, 使得 prioHigh 被迫延长不确定的阻塞时间。为避免这种情形, 实时操作系统引进优先级转置机制。

本节的例子程序演示 VxWorks 优先级转置机制。

为了允许使用优先级转置机制, 当使用互斥信号量时, VxWorks 提供了一个附加的选项: SEM_INVERSION_SAFE, 它允许使用优先级转置算法。这个算法保证占有一个资源的任务, 在其运行时, 其优先级等于阻塞在该资源上的所有任务的优先级的最高值。当执行完成, 这个任务释放资源, 返回正常的优先级。因此, 这个继承了最高优先级的任务, 将不会被优先级比它的正常优先级高但又比继承的优先级低的任务抢占。需要指出的是, 这个选项必须与 SEM_Q_PRIORITY 一起使用:

```
semId = semMCreate(SEM_Q_PRIORITY | SEM_INVERSION_SAFE);
```

下面的例子说明了发生优先级转置的典型情况。描述如下:

- (1) prioLow 任务上锁信号量。
- (2) prioLow 任务被 prioMedium 任务抢占而阻塞, 后者运行较长的一段时间。
- (3) prioHigh 任务抢占 prioMedium 任务, 试图上锁以被 prioLow 上锁的信号量。



这种情形程序运行的输出如下描述:

```

Low priority task locks semaphore
Medium task running
High priority task trys to lock semaphore
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
-----Medium priority task exited
Low priority task unlocks semaphore
High priority task locks semaphore
High priority task unlocks semaphore
High priority task trys to lock semaphore
High priority task locks semaphore
High priority task unlocks semaphore
High priority task trys to lock semaphore
High priority task locks semaphore
High priority task unlocks semaphore
.....High priority task exited
Low priority task locks semaphore
Low priority task unlocks semaphore
Low priority task locks semaphore
Low priority task unlocks semaphore
.....Low priority task exited

```

由于任务 `prioLow` 和 `prioHigh` 都被阻塞, `prioMedium` 将一直运行直至结束(很长的一段时间)。这时, `prioHigh` 可能已经错过其时限。

程序代码如下:

例 11.7 优先级转置

```

/*****
/* includes */

```




```
int i,j;
for (i=0; i < ITER; i++)
    {
        semTake(semMutex,WAIT_FOREVER); /* wait indefinitely for semaphore */
        printf("Low priority task locks semaphore\n");
        for (j=0; j < LONG_TIME; j++);
        printf("Low priority task unlocks semaphore\n");
        semGive(semMutex); /* give up semaphore */
    }
printf(".....Low priority task exited\n");
}

void prioMedium(void)
{
int i;
taskDelay(20);/* allow time for task with the lowest priority to seize semaphore
*/
for (i=0; i < LONG_TIME*10; i++)
    {
        if ((i % LONG_TIME) == 0)
            printf("Medium task running\n");
    }
printf("-----Medium priority task
exited\n");
}

void prioHigh(void)
{
int i,j;
taskDelay(30);/* allow time for task with the lowest priority to seize semaphore
*/
for (i=0; i < ITER; i++)
    {
        printf("High priority task trys to lock semaphore\n");
        semTake(semMutex,WAIT_FOREVER); /* wait indefinitely for semaphore */
        printf("High priority task locks semaphore\n");
        for (j=0; j < LONG_TIME; j++);
        printf("High priority task unlocks semaphore\n");
        semGive(semMutex); /* give up semaphore */
    }
printf(".....High priority task exited\n");
```



```

}
/*****

```

使用方法:

- (1) 编译代码。
- (2) 下载目标码到目标机。
- (3) 在 WindSh 中敲入 inversion, 运行代码。

请思考, 在上面的例子中, 如何修改程序, 消除优先级转置, 能得到下面的输出结果:

```

Low priority task locks semaphore
Medium task running
High priority task trys to lock semaphore
Low priority task unlocks semaphore
High priority task locks semaphore
High priority task unlocks semaphore
High priority task trys to lock semaphore
High priority task locks semaphore
High priority task unlocks semaphore
High priority task trys to lock semaphore
High priority task locks semaphore
High priority task unlocks semaphore
.....High priority task exited
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
-----Medium priority task exited
Low priority task locks semaphore
Low priority task unlocks semaphore
Low priority task locks semaphore
Low priority task unlocks semaphore
.....Low priority task exited

```



11.8 信 号

信号是一种软件通知，用以通知处理事件的任务。当引起一个信号的事件发生时，信号产生（generated）。当处理事件的任务激活时，信号释放（delivered）。信号的生命期是从产生到释放之间的时间。一个已经产生还没有释放的信号是挂起的（pending）。信号的生命期可能较长。

VxWorks 支持软件信号功能。信号异步地改变任务的控制流。任何任务都可以向一个特定任务发送信号。被信号通知的任务立即挂起它当前的执行线程，在下次任务被调度运行的时刻，指定的信号处理程序将获得处理器。甚至尽管任务处于阻塞态，其信号处理程序仍可以被调用执行。信号处理程序是由用户提供，并与特定的信号相联系，用于执行当信号发生是必要的处理工作。信号适用于错误和异常处理，很少用于任务间通信。

Wind 内核提供了两种信号接口：BSD 4.3 和 POSIX 信号接口。POSIX 接口提供比 BSD 4.3 接口更强大的标准接口。应用程序仅能使用其中一个。

本节的例子程序演示 VxWorks 提供的 POSIX 信号调用的使用。

VxWorks 提供 31 种不同的信号。程序可以调用 kill() 产生一个信号，与中断和硬件异常类似；调用 sigaction() 将信号与指定的信号处理程序相连结。当信号处理程序运行时，其他信号被阻塞。通过调用 sigprocmask()，任务可以阻止一些信号的出现，如果当信号产生时被阻塞，它的信号处理程序将在信号解除阻塞时调用。

信号处理程序通常定义形式为：

```
void sigHandlerFunction(int signalNumber)
{
    ..... /* signal handler code */
    .....
    .....
}
```

这里，signalNumber 是调用该处理程序的信号编号。

任务可以调用函数 sigaction 安装信号处理程序：

```
int sigaction(int signo, const struct sigaction *pAct, struct sigaction *pOact)
```

数据结构 struct sigaction 包含处理程序的信息，sigaction 含数有三个参数：需要捕获的信号编号、指向新的处理数据结构的指针（类型为 of type struct sigaction），指向旧的处理数据结构的指针（类型为 of type struct sigaction）。如果程序不需要旧的处理数据结构的指针（*pOact），那么可以传递一个空指针 NULL。

当需要把一个信号发送给一个任务时，可以调用 kill(int, int) 函数，第一个参数是任务



id 号，第二个参数是欲发送信号。

➤ 例子

在下面的例子中，sigGenerator 函数产生 SIGINT 或 Ctrl-C 信号，发送给任务 sigCatcher。当 sigCatcher 接收到该信号，它停止正常的执行，转入信号处理程序（catchSIGINT）。

例 11.8 信号

```

/*****
/* includes */
#include "vxWorks.h"
#include "sigLib.h"
#include "taskLib.h"
#include "stdio.h"

/* function prototypes */
void catchSIGINT(int);
void sigCatcher(void);

/* globals */
#define NO_OPTIONS 0
#define ITER1 100
#define LONG_TIME 1000000
#define HIGHPRIORITY 100
#define LOWPRIORITY 101
int ownId;

void sigGenerator(void) /* task to generate the SIGINT signal */
{
int i, j, taskId;
STATUS taskAlive;

if((taskId
taskSpawn("signal",100,0x100,20000,(FUNCPTR)sigCatcher,0,0,0,0,0,0,
0,0,0)) == ERROR)
printf("taskSpawn sigCatcher failed\n");

ownId = taskIdSelf(); /* get sigGenerator's task id */

taskDelay(30); /* allow time to get sigCatcher to run */

for (i=0; i < ITER1; i++)
{

```



```
if ((taskAlive = taskIdVerify(taskId)) == OK)
{
    printf("++++++SIGINT sinal generated\n");
    kill(taskId, SIGINT); /* generate signal */
    /* lower sigGenerator priority to allow sigCatcher to run */
    taskPrioritySet(ownId, LOWPRIORITY);
}
else /* sigCatcher is dead */
    break;
}
printf("\n*****sigGenerator Exited*****\n");
}

void sigCatcher(void) /* task to handle the SIGINT signal */
{
    struct sigaction newAction;
    int i, j;

    newAction.sa_handler = catchSIGINT; /* set the new handler */
    sigemptyset(&newAction.sa_mask); /* no other signals blocked */
    newAction.sa_flags = NO_OPTIONS; /* no special options */

    if(sigaction(SIGINT, &newAction, NULL) == -1)
        printf("Could not install signal handler\n");

    for (i=0; i < ITER1; i++)
    {
        for (j=0; j < LONG_TIME; j++);
        printf("Normal processing in sigCatcher\n");
    }

    printf("\n++++++sigCatcher Exited+++++\n");
}

void catchSIGINT(int signal) /* signal handler code */
{
    printf("-----SIGINT signal caught\n");
    /* increase sigGenerator priority to allow sigGenerator to run */
    taskPrioritySet(ownId, HIGHPRIORITY);
}
/*****/
```

使用方法:

- (1) 编译代码。



- (2) 下载目标码到目标机。
- (3) 在 WindSh 中敲入 sigGenerator, 运行代码。

请思考:

(1) 修改上面的例子程序, 不提供与信号 sigCatcher 的处理函数, 程序的运行结果将如何?

(2) 修改上面的例子程序, 阻塞 SIGINT 信号(也就是说: 阻止“sigCatcher”接收 SIGINT 信号)。

11.9 中断服务程序

中断处理程序是实时系统的重要组成部分。系统通过中断机制了解外部世界, 并对外部事件做出响应。实时系统的反映取决于系统对中断的响应速度和中断处理程序的处理速度。为了获得最好的反应时间, 编程者必须清楚地了解 VxWorks 提供函数库的优点。

本节演示 VxWorks 中断处理程序的编写方法。

我们可能需要写一个中断服务程序 (ISR), 并利用 VxWorks 提供的 intConnect 调用将它与某个中断相连接。当中断发生时, Wind 内核将转到中断服务程序执行。从中断产生到中断服务程序执行中间的时间延迟, 就是所谓的中断延迟时间。因为在很短的时间内, 可能产生很多中断, 高优先级的中断将阻塞低优先级的中断, 因此, 必须尽量使中断服务程序的处理时间最短, 这也是编程者的责任。

与 VxWorks 中断管理相关的头文件是 intLib.h--它也是中断库的头文件; arch/archname/ivrachname.h (其中, archname 是目标机类型)。中断服务程序代码不像一般的任务那样运行在普通的任务上下文中。它没有任务控制块, 所有的 ISR 共享一个堆栈。由于存在这些不同, 能在 ISR 调用的程序必须满足特别的要求: 不能阻塞! 请参考第六章有关内容。

在编写中断处理程序时, 如果需要输出调试信息, 由于 printf() 可能引起阻塞, 所以应该调用 logMsg 或其他的由 logLib 库提供的函数调用。ISRs 也不能使用浮点数指令, 因为浮点数指令寄存器在进入中断处理程序时没有保存。如果一定要使用浮点数指令, 必须首先调用 fppALib 库中提供的调用保存浮点寄存器。然而, 浮点数操作比较费时, 应该尽量避免在 ISRs 中使用。

理想情况, 一个 ISR 仅仅调用 semGive 系统调用。这也就是说, ISR 的主要功能应该是发起一个任务来完成必要的处理。为提高 VxWorks 中断服务程序与任务的合作性能, 最好的机制是信号量。

在下面的例子中, 任务 interruptGenerator 产生一个硬中断 sysBusIntGen(INTERRUPT_NUM, INTERRUPT_LEVEL); 它被 interruptCatcher 捕获。其函数原型为:

```
STATUS sysBusIntGen
```



```
(
    int intLevel, /* bus interrupt level to generate */
    int vector /* interrupt vector to generate (0-255) */
)
```

返回值 OK, 如果 intLevel 超出板子规定生成中断的范围, 则返回 ERROR。

interruptCatcher 能够由安装的中断处理程序 interruptHandler 处理这个硬件中断。

InterruptCatcher 调用 intConnect() 连接到该硬中断。对应的调用是:

```
intConnect(INUM_TO_IVEC(INTERRUPT_LEVEL), (VOIDFUNCPTR) interruptHandler, i);
```

其中, INUM_TO_IVEC(INTERRUPT_LEVEL) 是一个宏, 它将一个硬件中断号转换成中断向量。

程序运行时, interruptCatcher 正常运行, 模拟正常的处理, 直到 interruptGenerator 产生一个硬中断。interruptCatcher 停止它的正常执行, 转到 interruptHandler 进行中断处理。一旦中断处理代码执行完, 控制转到 interruptCatcher, 这种活动将重复多次。

例 11.9 中断服务程序

```
/* **** */
/* includes */
#include "vxworks.h"
#include "intLib.h"
#include "taskLib.h"
#include "arch/mc68k/ivMc68k.h"
#include "logLib.h"

/* function prototypes */
void interruptHandler(int);
void interruptCatcher(void);

/* globals */
#define INTERRUPT_NUM 2
#define INTERRUPT_LEVEL 65
#define ITER1 40
#define LONG_TIME 1000000
#define PRIORITY 100
#define ONE_SECOND 100

void interruptGenerator(void) /* task to generate the SIGINT signal */
{
    int i, j, taskId, priority;
    STATUS taskAlive;
```



```

if((tasked = taskSpawn("interruptCatcher", PRIORITY, 0x100, 20000,
(FUNCPTR)interruptCatcher, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
== ERROR)
    logMsg("taskSpawn interruptCatcher failed\n", 0, 0, 0, 0, 0);

for (i=0; i < ITER1; i++)
    {
    taskDelay(ONE_SECOND); /* suspend interruptGenerator for one second */
    /* check to see if interruptCatcher task is alive! */
    if ((taskAlive = taskIdVerify(taskId)) == OK)
        {
        logMsg("++++++++++++++++++++Interrupt
generated\n", 0, 0, 0, 0, 0);
        /* generate hardware interrupt 2 */
        if((sysBusIntGen(INTERRUPT_NUM, INTERRUPT_LEVEL)) == ERROR)
            logMsg("Interrupt not generated\n", 0, 0, 0, 0, 0);

        }
    else /* interruptCatcher is dead */
        break;
    }
logMsg("\n*****interruptGenerator
Exited*****\n\n\n", 0, 0, 0, 0, 0);
}

void interruptCatcher(void) /* task to handle the interrupt */
{
int i, j;
STATUS connected;

/* connect the interrupt vector, INTERRUPT_LEVEL, to a specific interrupt
handler routine ,interruptHandler, and pass an argument, i */
if((connected = intConnect(INUM_TO_IVEC(INTERRUPT_LEVEL),
(VOIDFUNCPTR)interruptHandler, i)) == ERROR)
    logMsg("intConnect failed\n", 0, 0, 0, 0, 0);

for (i=0; i < ITER1; i++)
    {
    for (j=0; j < LONG_TIME; j++);
    logMsg("Normal processing in interruptCatcher\n", 0, 0, 0, 0, 0);
    }
logMsg("\n++++++++++++++++interruptCatcher
Exited++++++++++++\n", 0, 0, 0, 0, 0);
}

```



```
}

void interruptHandler(int arg) /* signal handler code */
{
int i;

logMsg("-----interrupt caught\n",0,0,0,0,0,0);
for (i=0; i < 5; i++)
    logMsg("interrupt processing\n",0,0,0,0,0,0);
}
/*****/
```

使用方法:

- (1) 编译代码。
- (2) 下载目标码到目标机。
- (3) 在 WindSh 中敲入 logFdSet 1, 这将定位 logMsg()输出到虚拟控制台。
- (4) 在 WindSh 中敲入 interruptGenerator, 运行代码。

请思考:

- (1) 为什么不能使用 printf()替代 logMsg()?
- (2) 修改上面的例子程序, 使其能够处理硬件中断 1、2 和 3, 这些中断按升序由 interruptGenerator 产生。interruptCatcher 使用三个中断处理程序分别处理每个中断。

附录



《VxWorks Programmer's Guide 5.4》 Wind River Systems, Inc
《Tornado User's Guide》 Wind River Systems, Inc