基于VxWorks的嵌入式软件设计

Embedded Software Design Based on VxWorks

编者 王韬

审稿人 王永东

重庆大学通信工程学院

2010年10月

前 言

目前人类社会已经进入后PC 时代，嵌入式系统已经渗透到人类生产和生活的各个领域，可以说嵌入式产品无处不在。嵌入式实时操作系统是嵌入式系统的核心和灵魂，是嵌入式系统的开发平台和运行平台。WindRiver 公司生产的VxWorks 操作系统是商业化的嵌入式实时操作系统，以其成功应用于火星探测器而闻名于世。为适应国家对嵌入式人才的需要，重庆大学通信工程学院与重庆大学研究生院共建了教育部研究生教育创新试点工程－VxWorks 开放实验室，并面向重庆大学硕士研究生开设了嵌入式实时操作系统这门课程。为培养学生的实践动手能力和创新能力，特编写了本实验课程设计讲义。本讲义涵盖了多任务管理、通信、同步、互斥、中断处理、IO 设备驱动、定时器管理等几乎嵌入式系统软件可能涉及的所有方面。

本讲义是为重庆大学电类相关专业硕士研究生编写的一本内部讲义，是作者多年从事嵌入式技术研究开发和教学经验的总结，在编写过程中参考了国内外相关论文与手册，文中还提供了很多参考源程序，相信对读者会有很大帮助。由于作者水平所限，加之此书成稿时间太短，文中必有不少错误，希望读者批评指正。

重庆大学通信工程学院王韬　　2010年10月

目　录

1 嵌入式系统概述

1.1 嵌入式系统概述

1.1.1 嵌入式系统的定义和市场前景

嵌入式系统是指以应用为中心，以计算机技术为基础，软件硬件可剪裁，适应应用系统对功能可靠性、成本、体积、功耗严格要求的专用计算机系统。它主要由嵌入式微处理器、外围硬件设备、嵌入式操作系统以及用户应用软件等部分组成。用于实现对其它设备的控制、监视和管理等功能，它通常嵌入在主要设备中运行。

PC 机主要应用在办公室自动化领域，而嵌入式系统已经广泛渗透到人们的工作、生活中，从家用电器、手持通讯设备、信息终端、仪器仪表、汽车、航天航空、军事装备、制造工业、过程控制等。今天，嵌入式系统带来的工业年产值已超过1 万亿美元。美国著名未来学家尼葛洛庞帝99 年1 月访华时曾预言，4~5 年后嵌入式智能（电脑）工具将是PC 和因特网之后最伟大的发明。据统计，嵌入式处理器的数量占分散处理器的94%，而PC 机用的处理器只占6%。

汽车大王福特公司的高级经理曾称：“福特出售的‘计算能力’已超过了IBM！”用市场观点来看，PC已经从高速增长进入到平稳发展时期，其年增长率由上世纪90 年代中期的35%逐年下降，单纯由PC 机带领电子产业蒸蒸日上的时代已经成为历史，根据PC时代的概念，美国Business week 杂志提出了“后PC时代”概念。根据美国嵌入式系统专业杂志RTC 报道，21 世纪初的十年中，全球嵌入式系统市场需求量具有比PC市场大10 至100 倍的商机。1998 年在芝加哥举办的嵌入式系统会议上，与会专家一致认为，21 世纪嵌入式系统将无所不在，它将为人类生产带来革命性的发展，实现“PCs Everywhere”的生活梦想。

1.1.2 嵌入式系统的几个发展阶段

嵌入式系统的出现至今已经有30 多年的历史，近几年来，计算机、通信、消费电子的一体化趋势日益明显，嵌入式技术已成为一个研究热点。纵观嵌入式技术的发展过程，大致经历四个阶段。

第一阶段是以单芯片为核心的可编程控制器形式的系统，具有与监测、伺服、指示设备相配合的功能。这类系统大部分应用于一些专业性强的工业控制系统中，一般没有操作系统的支持，通过汇编语言编程对系统进行直接控制。这一阶段系统的主要特点是：系统结构和功能相对单一，处理效率较低，存储容量较小，几乎没有用户接口。由于这种嵌入式系统使用简单、价格低，以前在国内工业领域应用较为普遍，但是已经远不能适应高效的、需要大容量存储的现代工业控制和新兴信息家电等领域的需求。

第二阶段是以嵌入式CPU 为基础、以简单操作系统为核心的嵌入式系统。主要特点是：CPU 种类繁多，通用性比较弱；系统开销小，效率高；操作系统达到一定的兼容性和扩展性；应用软件较专业化，用户界面不够友好。

第三阶段是以嵌入式操作系统为标志的嵌入式系统。主要特点是：嵌入式操作系统能运行于各种不同类型的微处理器上，兼容性好；操作系统内核小、效率高，并且具有高度的模块化和扩展性；具备文件和目录管理、多任务、网络支持、图形窗口以及用户界面等功能；具有大量的应用程序接口API，开发应用程序较简单；嵌入式应用软件丰富。

第四阶段是以Internet 为标志的嵌入式系统。这是一个正在迅速发展的阶段。目前大多数嵌入式系统还孤立于Internet 之外，但随着Internet 的发展以及Internet 技术与信息家电、工业控制技术结合日益密切，嵌入式设备与Internet 的结合将代表嵌入式系统的未来。

嵌入式系统技术日益完善，32 位微处理器在该系统中占主导地位，嵌入式操作系统已经从简单走向成熟，它与网络、Internet 结合日益密切，因而，嵌入式系统应用将日益广泛。

11.3 嵌入式系统的技术特点

. *嵌入式系统是集软件、硬件于一体的高可靠性系统* 嵌入式系统麻雀虽小，五脏俱全，软件除操作系统外，还需有完成嵌入式系统功能的应用软件，硬件除了CPU 外，还需有外围电路支持，微处理器、微控制器、DSP 已构成嵌入式系统硬件的基础。

. *嵌入式系统是资源开销小的高性能价格比系统* 嵌入式系统的发展离不开应用，应用的共同要求是系统资源开销小，由于嵌入式系统技术日益完善，各种高性能嵌入式应用系统层出不穷，它已是资源开销小的高性能价格比的一类应用系统。为了满足系统资源开销小、高性能、高可靠性的要求，大多使用FlashMemory。
. *嵌入式系统是功能强大、使用灵活方便的系统* 嵌入式系统应用的广泛性，要求该系统通常是无键盘、无需编程的应用系统，使用它应如同使用家用电器一样方便。

1.1.4 嵌入式系统的发展趋势

. *低功耗嵌入式系统* 为满足高可靠性要求，低功耗的系统将应运而生。
. *Java 虚拟机与嵌入式Java* 开发嵌入式系统希望有一个方便的、跨平台的语言与工具，Java 正是用Java 虚拟机实现Java 程序独立于各机种的平台。经过努力，一个支持嵌入式系统开发的、足够小、足够快、又有足够确定性的嵌入式Java 程序包已经出现，Java 虚拟机与嵌入式Java 将成为开发嵌入式系统的有力工具。
. *嵌入式系统的多媒体化和网络化* 随着多媒体技术的发展，视频、音频信息的处理水平越来越高，为嵌入式系统的多媒体化创造了良好的条件，嵌入式系统的多媒体化将变成现实。它在网络环境中的应用已是不可抗拒的潮流，并将占领网络接入设备的主导地位。
. *嵌入式系统的智能化* 嵌入式系统与人工智能、模式识别技术的结合，将开发出各种更具人性化、智能化的嵌入式系统。

1.2 嵌入式系统的核心硬件

嵌入式系统的核心部件是各种类型的嵌入式处理器，据不完全统计，全世界嵌入式处理器的品种已有上千种之多。其中，我们最为熟悉的是8051 和68H结构的产品。实际上，几十年来，各种4、8、16 和32 位的处理器在嵌入式系统中都有广泛应用。嵌入式系统的处理器可以分为两大类：一类是采用通用计算机的CPU 为处理器，如X86 系列；另一类为微控制器和DSP，微控制器具有单片化、体积小、功耗低、可靠性高、芯片上的外设资源丰富等特点，成为嵌入式系统的主流器件。

当前，嵌入式系统处理器的发展趋势主要采用32位嵌入式CPU，其主流系列有ARM(包括Intel 公司的strong ARM 和XScale)、MIPS、PowerPC 和SH 四大系列。

嵌入式系统CPU 的另一类型为DSP。当前，DSP处理器的典型结构是单片化嵌入式DSP，如TI 公司的TMS320 系列；另一类是在通用CPU 或单片系统中增加DSP 协处理器，如Intel 公司的MCS-296 等。还有一种类型是选用嵌入式单片系统SOC(System On a Chip)。

其中，特别要指出，RISC 技术为计算机体系结构带来了一次重大的变革。简单的、固定长度的、单周期执行指令的RISC 计算系统，与传统、复杂、可变长度指令并行执行的CISC 计算机系统相比较，在相同的条件下，RISC 技术的速度快2~5 倍，具有巨大的性价比优势。RISC 技术推动着计算机体系结构从封闭的CISC 向开放的结构发展。因此，世界上各大CPU 芯片制造厂商争相开发生产RISC 芯片，目前的典型结构为ARM 系列、MIPS 和SH，32 位字长，最高时钟速率可达400MHz。多种嵌入式实时操作系统大都支持上述RISC 处理器。国际上有一种新的趋向，即可以购买IP 知识产权核模块，即现有的IC 电路模块的设计，在其基础上，可根据需求将多个IP 模块组合起来或经修改，形成自己的新的设计。由此可见，半导体芯片的设计现已不难了。其中，尤以ARM 的应用最为典型，各半导体厂商大多可生产ARM 的衍生产品。

1.3 嵌入式实时操作系统

嵌入式系统的概念出现在20 世纪70 年代。当时，大部分嵌入式系统的软件都是由汇编语言编纂而成，它只能应用于某种特定的微处理器和应用，当然，对于某些简单的应用是足够了.但是这时候的嵌入式系统的实时性并不高。实时的含义是指在规定的时限内能够传递正确的结果，迟到的结果就是错误。实时系统并非指“快速”系统，实时系统有限定的响应时间，从而使系统具有可预测性。实时系统又可以分为“硬实时系统”和“软实时系统”。二者的区别在于：前者如果在不满足响应时限、响应不及时或反应过早的情况下都会导致灾难性的后果（如航空航天系统）；而后者则在不满足响应时限时，系统性能退化，但并

不会导致灾难性的后果如（交换系统）。

实时操作系统RTOS(Real-time Operating System；简称RTOS)是嵌入式应用软件的基础和开发平台。目前大多数嵌入式应用还是载单片机上直接运行，没有RTOS，但仍要有一个主程序负责调度各个任务。RTOS是一段嵌入在目标程序中的程序，系统复位后首先执行，相当于用户的主程序，用户的其它应用程序都建立在RTOS上。不仅如此，RTOS还是一个标准的内核，将CPU时间、中断、I/O定时器等资源包装起来，留给用户一个标准的API（System call），系统调用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。实时操作系统的首要任务是调度一切可利用的资源完成实时控制任务，其次才着眼于提高计算机系统的使用效率，其重要特点是要通过任务调度来满足对于重要事件在规定的时间内做出正确的响应。

RTOS是针对不同地微处理器设计优化地高效率实时多任务内核，RTOS可以面对几十个系列的微处理器MPU、MCU、DSP、SOC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语言具有极大的可移植性。据专家预测，优秀的RTOS上跨处理器平台的程序移植只需修改1-4%内容。在RTOS基础上可以编写各种硬件驱动程序、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，促进行业知识产权交流。因此，RTOS又是一个软件开发平台。

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。

自1981 年发展了世界上第一个商业嵌入式实时操作系统（VRTX32）至今，嵌入式实时操作系统已有20多年的发展历程。20 世纪80 年代的产品还只支持16 位的微处理器，只有内核，以销售二进制代码为主，当时的产品如VRTX、PSOS 等，主要用于军事和电信设备。进入20 世纪90 年代，现代操作系统的设计思想，如微内核设计技术和模块化的设计思想已开始渗入其中，特别是互联网日渐风行，用户都要求有网络（即支持浏览器）和图形功能，并能方便使用大量现有的软件代码，希望支持标准的API，如Posix、Win32 等，并希望其开发环境与大家熟悉的Unix、Windows 一致，出现了几十种产品，代表性的有VxWorks、QNX、Nucleus和WinCE 等。

20 世纪90 年代后，嵌入式实时操作系统已在嵌入式系统中确立了主导地位，在技术上具有如下突出的特征：

(1) 因新的处理器越来越多，嵌入式实时操作系统的设计更易于移植，以便在短时间内支持多种微处理器。

(2) 开放源代码之风盛行，相当多的厂家在出售实时操作系统时，就附加了源代码并含生产版权。

(3) 电信设备、控制系统要求高的可靠性，迫使这些厂家下功夫提高性能，VxWorks 等对支持高可用性和热切换性都下了一番功夫。

(4) 嵌入式Linux在消费电子设备中得以广泛应用，Linux 得到了相当广泛厂商的支持和投资。RT-Linux 产品也取得了很大的进展。由32 位嵌入式CPU 与嵌入式RT-Linux 相结合，在家用电器、工业控制和军工装备方面将大有可为。

嵌入式实时操作系统典型产品如下：

VxWorks WindRiver 公司的高性能可扩展的实时操作系统，具有嵌入实时应用中最新一代的开发和执行环境，支持多种处理器和开发平台，并有多种开发工具，是目前世界上应用最广泛的产品。

PSOS ISI 公司研发的产品，该产品推出时间较早，因此比较成熟，可以支持多种处理器，曾是国际上应用最广泛的产品，主要应用领域是远程通信、航天、信息家电和工业控制。但该公司已被WindRiver公司兼并，并将推出VxWorks 与PSOS 合二为一的产品。

VRTX 是国际上最早推出的实时系统之一，比较成熟。其特点是内核紧凑，在模块化方面比原系统有重大的改善。

Nucleus 其特点是约95%的代码用C 语言编写，方便移植，同时，可提供web 支持、网络、图形、文件系统等模块。可全部提供源代码，用户只需通过DLL 动态连接便可进行任务级调试。

Lynx 具有Unix 兼容，符合Posix 标准。是专为要求快速确定相应的、复杂的实时应用设计的，它能为在任何一个Unix 平台上的应用提供相当高程度源级水平上的兼容。

. **Windows CE** 是微软公司嵌入式实时应用系统，支持众多的硬件平台，其最主要特点是拥有与桌上型Windows 家族一致的程序开发界面，因此，桌上型Windows 家族上开发的程序就能在WinCE上运行。但嵌入式操作系统追求高效、节省，WinCE 在这方面是笨拙的，它占用内存过大，应用程序庞大。

RT-Linux 是一种提供源代码、开放式自由软件，具有嵌入式操作系统的很多特色，突出的优势是适用多种CPU 和多种硬件平台，性能稳定，裁剪性好，开发和使用都很容易。它是发展未来嵌入式设备的绝佳资源。国内也有不少单位在RT-Linux 方面做了大量卓有成效的工作，具有广泛的应用前景。

此外，后PC 时代的众多产品，如手持设备等，并不需要强实时性，PalmOS、JavaOS 等应运而生。而Qssl 公司拥有的QNX是一种限于X86 平台的可提供集成化开发环境的实时操作系统。OS/9 在DVD 等产品中则有广泛应用。

2 vxWorks 简介

2.1 vxWorks 背景简介

我们知道，用户在开发具有日益复杂的 32 位嵌入式处理器的产品时，需要一个用来连接产品应用程序和底层硬件的操作系统。用户寻求的操作系统应该具有的最重要的特点包括：可靠性、高实时性、可裁剪性、可协同工作。VxWorks 操作系统是美国 WINDRIVER 公司于 1983 年设计开发出的一种嵌入式实时操作系统（RTOS），是 TornadoII 嵌入式开发环境的关键组成部分。良好的持续发展能力、高性能的内核以及友好的用户开发环境，在嵌入式实时操作系统领域占据了很大市场。首先，它十分灵活，具有多达 1800 个功能请达应用程序接口（API）；其次，它适用方面广，可以适用于从最简单的最复杂的产品设计；再次，它可靠性高，可以用于从防抱死刹车系统到星际探索的关键任务；最后，适用性强，可以用于所有的流行的 32 位 CPU 平台。vxWorks 以其成功地应用在火星探测器和爱国者导弹而闻名于世。

vxWorks就是众多嵌入式实时操作系统中的一种，它功能强大而且比较复杂，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。所以，仅仅依靠人工编程调试，很难发挥它的功能并设计出可靠、高效的嵌入式系统，必须要有与之相适应的开发工具。TornadoII 就是为开发VxWorks 应用系统提供的集成开发环境，TornadoII 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，可以按用户的需要裁剪配置VxWorks 内核；vxSim 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CrossWind 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。

2.2 vxWorks 的结构

VxWorks 嵌入式实时操作系统包括微内核wind、高级的网络支持、强有力的文件系统和I/O 管理、C++和其他标准支持等核心功能。这些核心功能还可以与WindRiver 公司的其他产品以及320 个WindRiver 公司的合作伙伴的产品联合使用。

VxWorks 操作系统包括了板级支持包BSP(Board Support Package)、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。而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 操作系统的基本构成部件主要有以下五个部分：

- 板级支持包BSP(Board Support Package)
- 微内核wind
- 网络系统
- 文件系统
- I/O 系统

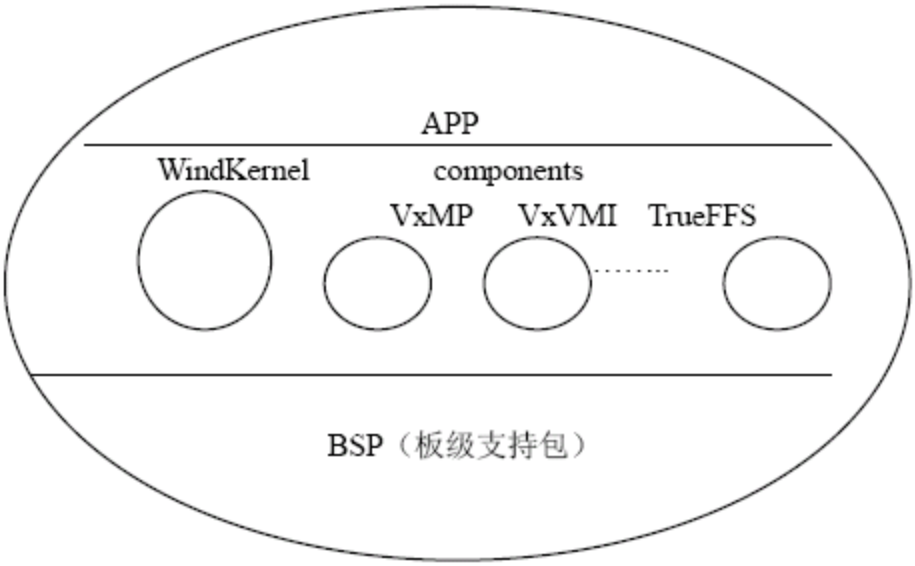


图 2－1 VxWorks 映象文件

2.2.1 高性能的微内核**WindKernel**

处于VxWorks 嵌入式实时操作系统核心的是高性能的微内核wind。这个微内核支持所有的实时特征：快速任务切换、中断支持、抢占式和时间片轮转调度等。微内核设计减少了系统开销，从而保证了对外部事件的快速、确定的反应。运行环境也提供了有效的任务间通信机制，允许独立的任务在实时系统中与其行动相协调。开发者在开发应用程序时可以使用多种方法：用于简单数据共享的共享内存、用于单CPU 的多任务间信息交换的消息队列和管道、套接口、用于网络通信的远程过程调用、用于处理异常事件的信号等。为了控制关键的系统资源，提供了三种信号灯：二进制、计数、有优先级继承特性的互斥信号灯。VxWorks 的核心，被称作Wind，包括多任务调度（采用优先级抢占方式），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。进程间通信机制使得这些任务的行为同步、协调。

Wind 使用中断驱动和优先级的方式。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。还有一些其它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 核提供信号量作为任务间同步和互斥的机制。在 wind 核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 核也提供了诸如消息队列、管道、套接字和信号等机制。

2.2.2 板级支持包 **BSP（Board Support Package；Hardware Layer Abstract）**

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）或其它启动机制。

2.2.3 综合的网络工具

VxWorks 是第一个支持工业标准TCP/IP 的实时操作系统。创新的传统伴随着VxWorks TCP/IP 协议栈，它支持最新的Berkeley 网络特性，包括：

- IP, IGMP, CIDR, TCP, UDP, ARP
- RIP v.1/v.2
- Standard Berkeley sockets and zbufs

- NFS client and server, ONC, RPC
- Point-to-Point Protocol
- BOOTP, DNS, DHCP, TFTP
- FTP, rlogin, telnet, rsh

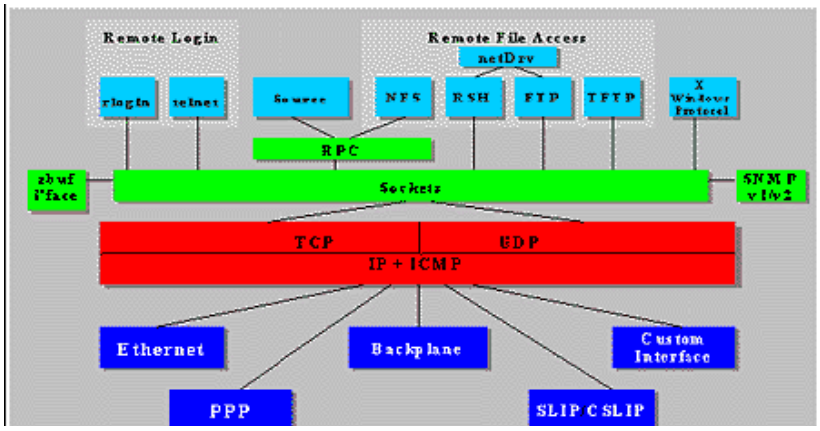


图 2－2 vxWorks 网络协议栈

WindRiver 也支持可选的WindNet 产品：SNMP v.1/v.2c, OSPF v.2, STREAMS。WindRiver 还通过提供工业级最广泛的网络开发环境来加强这些核心技术，这主要是通过WindLink for TornadoII 伙伴计划来实现的。高级的网络解决方案还包括：

- ATM, SMDS, frame relay, ISDN, SS7, X.25, V5 广域网网络协议
- IPX/SPX，AppleTalk，SNA 局域网网络协议
- 分布式网络管理的RMON，CMIP/GDMO，基于Web 网的解决方案
- CORBA 分布式计算机环境

2.2.4 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被灵活地在设备驱动程序上移植。

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。

VxWorks 支持四种文件系统：

- dosFs
- rt11Fs
- rawFs
- tapeFs

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。

2.2.5 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
- POSIX 标准的异步 I/O

VxWorks 包括以下驱动程序：

- 网络驱动
- 管道驱动
- RAM 盘驱动
- SCSI 驱动

- 键盘驱动
- 显示驱动
- 磁盘驱动
- 并口驱动

2.3 VxWorks 的主要特点

1、微内核(wind microkernel)的主要特点

- 高效的任务管理：

◇ 无限数目多任务，具有256 个优先级
◇ 具有优先级排队和时间片轮转调度
◇ 快速的、确定性的上下文切换

- 快速灵活的任务间通讯：

◇ 三种信号灯：二进制、计数、有优先级继承特性的互斥信号灯
◇ POSIX 管道、记数信号量、消息队列、信号和调度
◇ 控制套接口
◇ 共享内存(shared memory)

- 高度的可裁剪性

- 增量连接和部件加载

- 快速有效的中断和异常事件处理

- 优化的浮点支持

- 动态内存管理
- 系统时钟和计时工具

2、网络支持方面

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

3、快速灵活的I/O 和本地文件系统

- POSIX 异步I/O 和目录管理
- SCSI 支持

- 兼容MS-DOC 文件系统
- Raw disk 文件系统
- TrueFFS 闪存文件系统
- ISO 9660 CD-ROM 文件系统
- PCMCIA 支持

4、目标机开发特性

- 完全兼容ANSI C 和C++的异常处理和模板支持
- 兼容POSIX 1003.1, .1b 实时扩展
- 目标机shell 上的交互式C 解释器
- 符号调试和反汇编
- 强大的性能监视功能
- 扩展的内核、任务、系统信息工具
- 动态连接装载
- 超过1800 个实用例程库
- 灵活的启动方式，可以从ROM、本地磁盘或通过网络启动
- 高度可裁剪设计可以适用于广泛的应用
- 通过以太网、串行线、ICE 或ROM 仿真器的系统级调试

我们将在接下来的章节中介绍vxWorks的集成开发环境Tornado的使用和VxWorks实时操作系统内核的工作原理。

3 Tornado 简介

3.1 Tornado 集成开发环境概述

WRS(风河)公司的VxWorks是目前最先进的实时操作系统，Tornado是它的一体化开发环境。VxWorks第五版用C语言建立了占其98%的代码的common base code。这样向一个处理器移植VxWorks，完全是在C语言基础上进行，然后再根据情况用汇编语言对它加以优化。VxWorks的规模可伸缩能力也相当好，它提供了80多种可选的功能，供开发者用来配置不同规模的系统。最小规模是16K ROM和小于1K的RAM，已经可以满足所有小型嵌入式应用的要求。Tornado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 集成开发系统包含二个高度集成的部分：

- vxworks运行在目标机上的高性能、可裁剪的实时操作系统；
- Torado开发环境IDE；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成ROM仿真器等。

Tornado IDE的主要组成

- 集成的源代码编辑器
- 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。
- Winview，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3－9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：

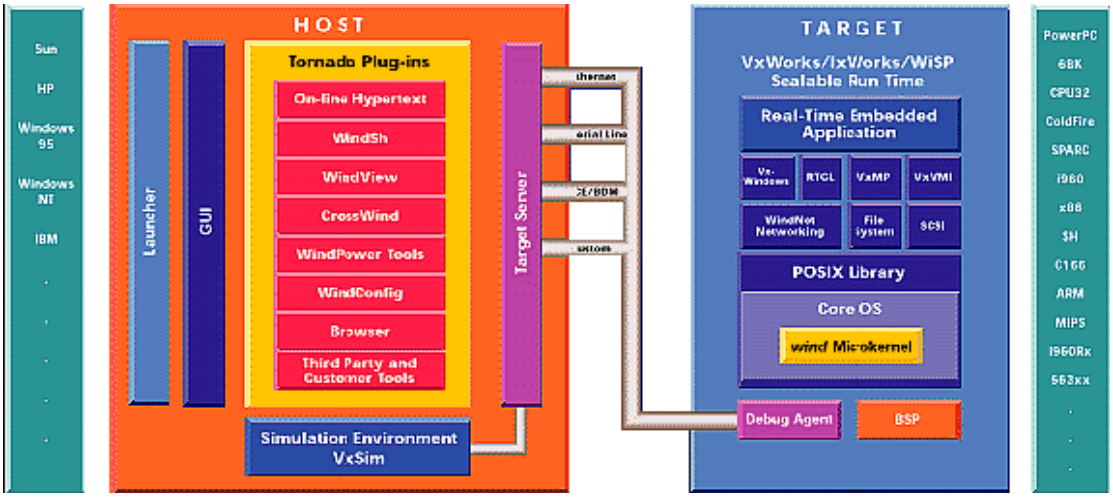


图 3－1 交叉开发环境

使用Tornado，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，可以位开发者省去通常的开发步骤：在十机上将应用积序与内核链接起来，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。

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、命令完成窗口和下捡式的历史记录窗口，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, 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TornadoII 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和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)

TornadoII 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。

TornadoII 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，这些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)

TornadoII 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管理功能。

▲ 目标机系统状态浏览器(Browser)

TornadoII 的目标机系统浏览器Browser 是TornadoII 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)

TornadoII的命令行执行工具WindSh是TornadoII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)

TornadoII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)

TornadoII 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)

TornadoII 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。

3.3 使用 Tornado 开发嵌入式软件的流程

1、开发BSP

1）假设tornado安装在E:\Tornadoppc\目录下，编写BSP源文件并将其存放在E:\Tornadoppc\target\config\wt860子目录下，按图所示编译连接生成bootrom.hex

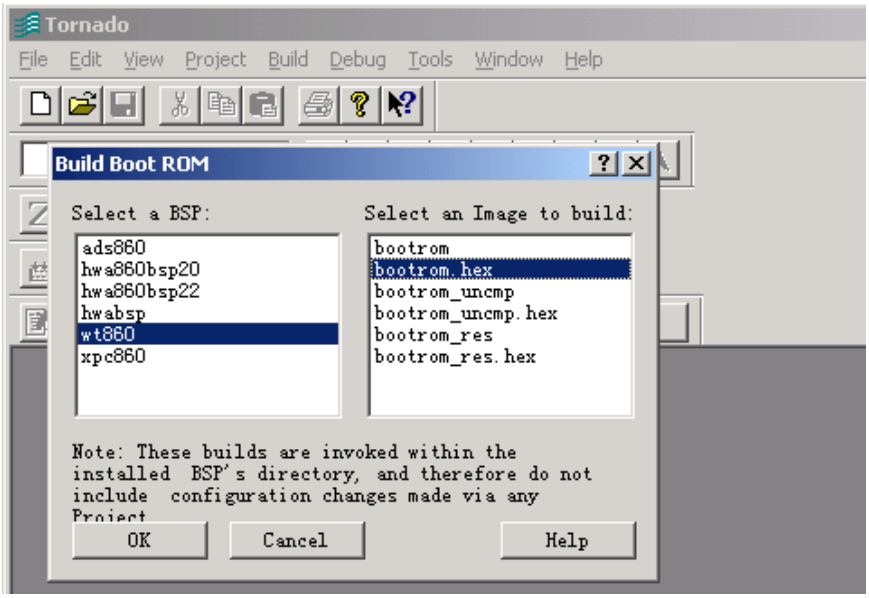


图3－2 编译连接生成bootrom.hex

2)通过编程器将生成bootrom.hex烧录到目标板上的flash中。

将仿真器的BDM仿真头与目标板相连，将仿真器通过并口与PC机相连，启动PC机上的flash Programmer软件，选择CPU和flash的型号，然后点击Reset Target，出现下图(Target Reset ASSertted)说明连接正确。然后点击program按钮出现如图3所示的画面，再点击Browser按钮选择刚刚生成的bootrom.hex文件，然后再点击Program按钮烧录bootrom.hex文件。

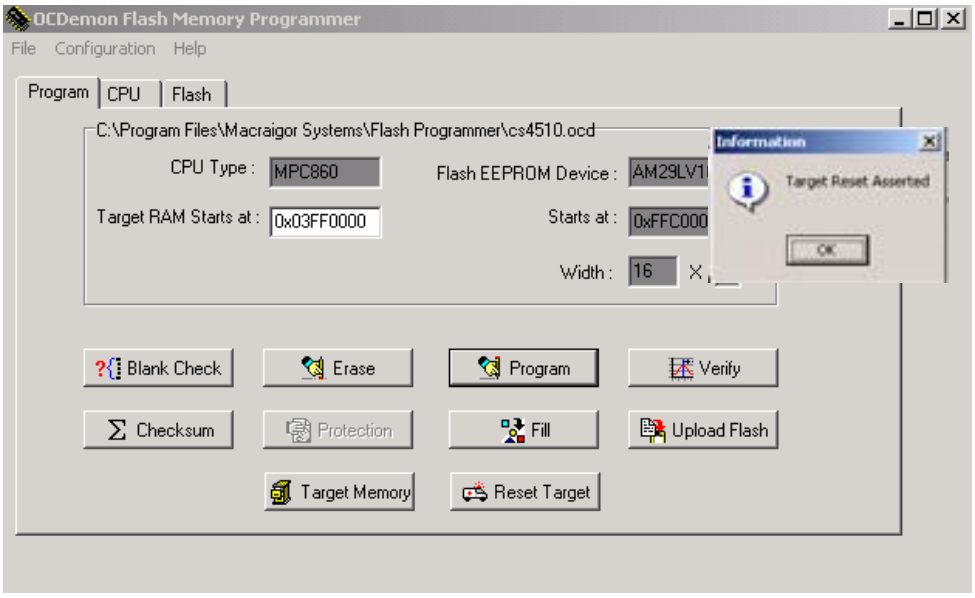


图3－3 配置flash program软件

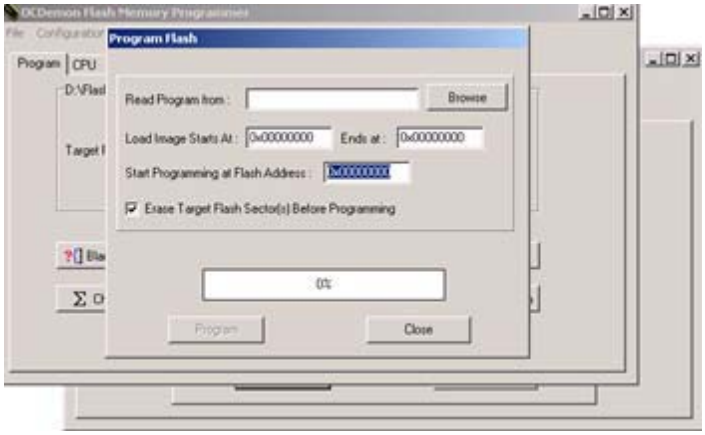


图3－4 烧录bootrom.hex文件

2.创建bootable 工程，生成vxworks，配置超级终端和ftp服务器并下载到目标机中。

1）点击Windows 任务栏上的“开始”按钮，选择 “程序”文件夹，然后选择“Tornado2”程序组，点击其中的Tornado 项，即可启动Tornado运行。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的对话框，如图所示：

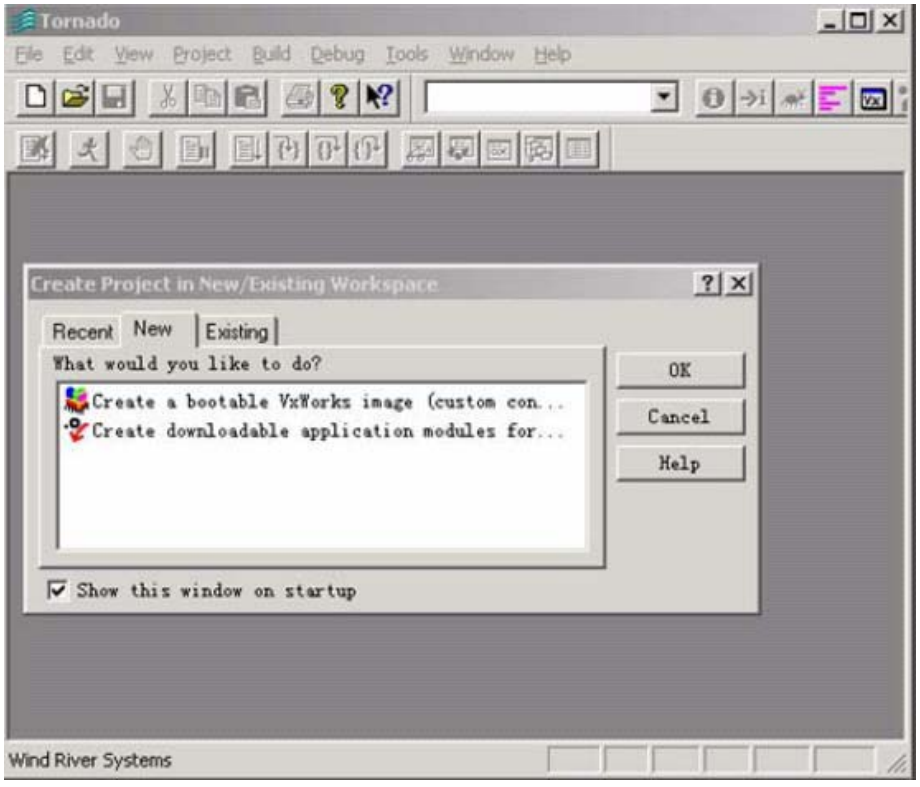


图3－5创建工程的对话框

2）选择“ “Create a bootable VxWorks image（custom con…”点击OK 继续。屏幕将出现如下的对话框：

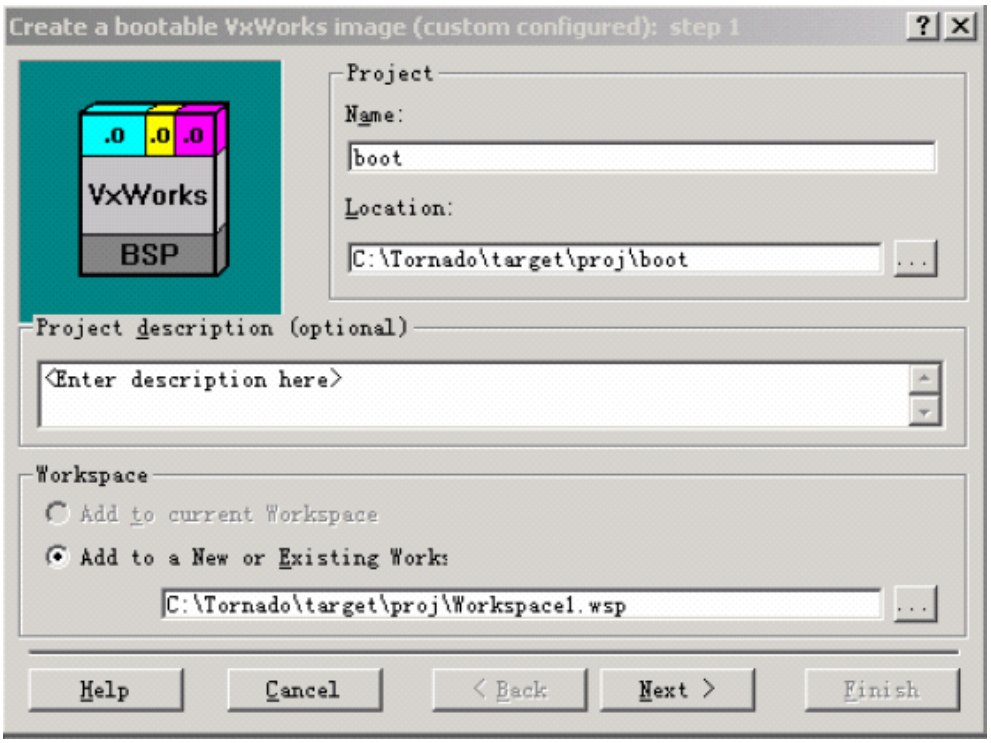


图3－6 工程属性对话框

在该对话框中，提示用户输入新工程的名字，工程文件存放的路径，工程描述信息，以及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。在本实验示例中，工程的名字是boot；工程的路径是C:\Tornado\target\proj\boot；工程的描述信息未输入；工作空间的路径及名称是c:\Tornado\target\proj\Workspace.wsp。

3）点击Next，继续下一步操作。在接下来的对话框，要输入实验用板的BSP 或选择基于已存在的工程来创建新的工程；选择hwabsp，操作界面如下：

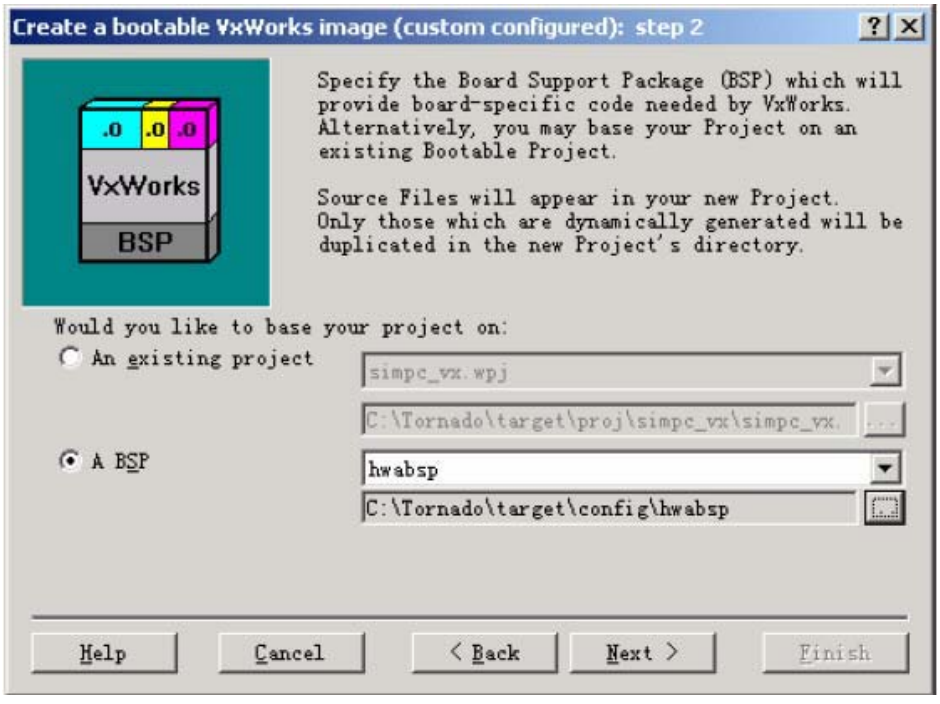


图3－7 BSP 选择对话框

4）点击Next；完成下一步，Toando 的引导程序出现最后的对话框要求对以上的输入信息进行确认。

操作界面如下：

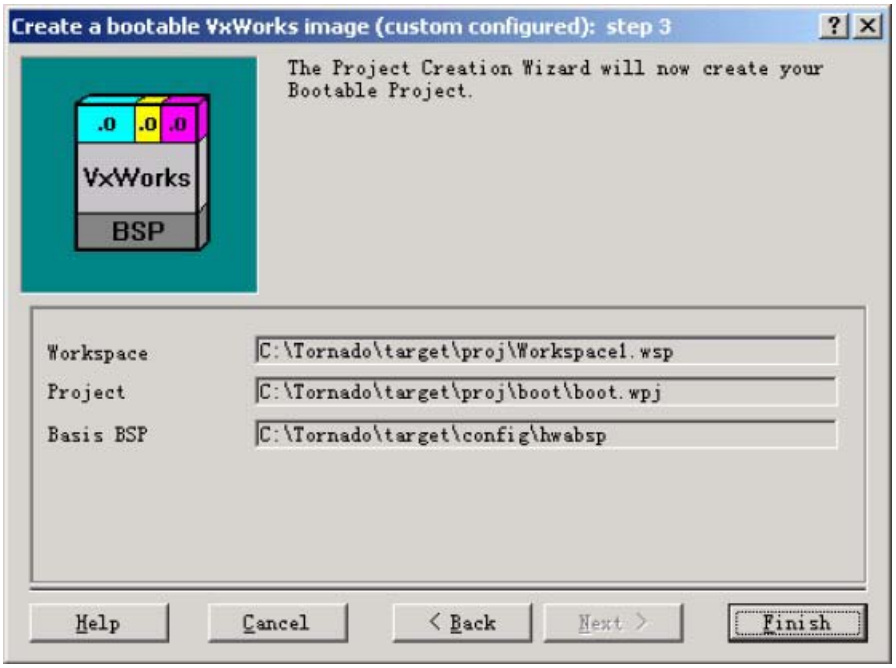


图3－8工程属性确认对话框

5） 点击Finish，继续下一步，这时将出现工作空间窗口。

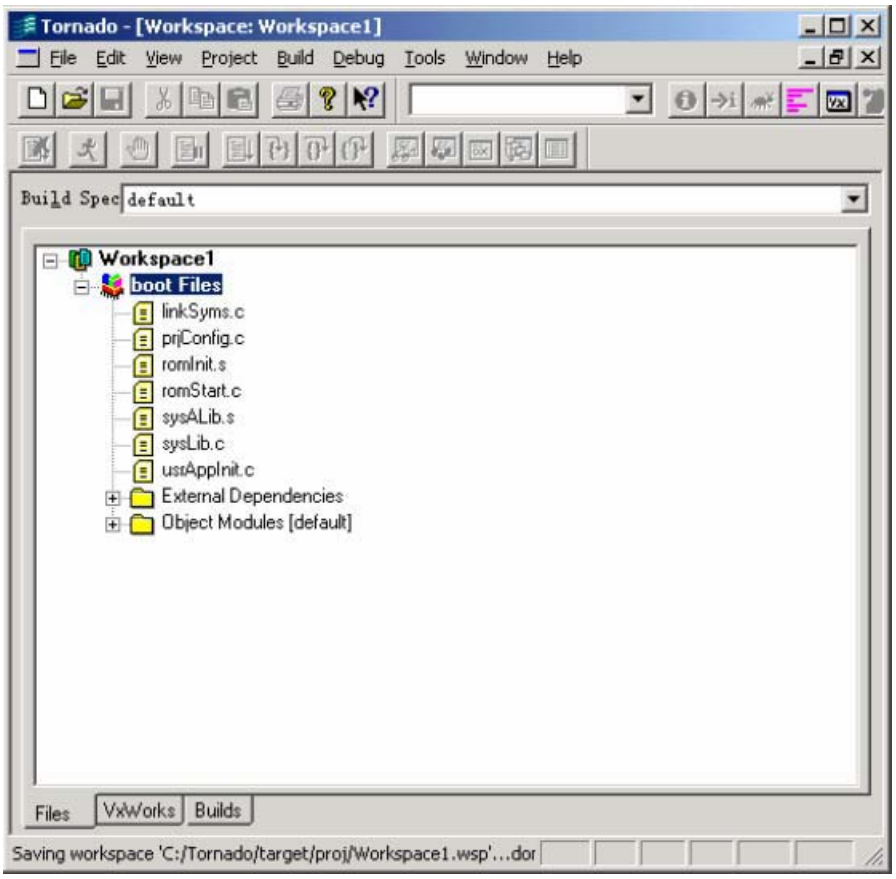


图3－9 工程创建完成后的界面

6）点击窗口左下方的Builds 图标，出现下图界面：

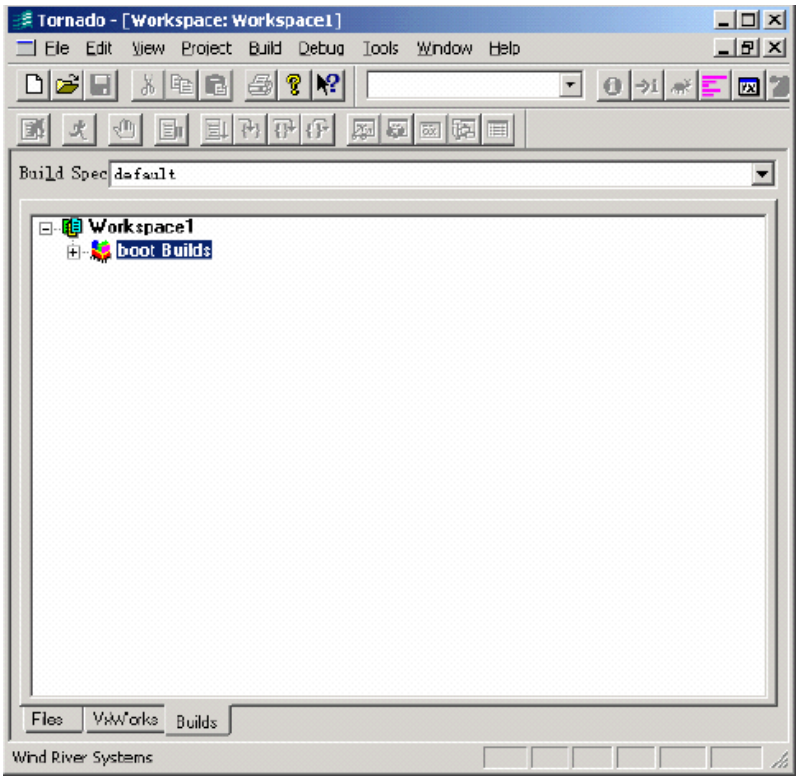


图3－10 点击Builds图标后的界面

7）点击“boot Builds”前的“＋”号，出现如下的画面：

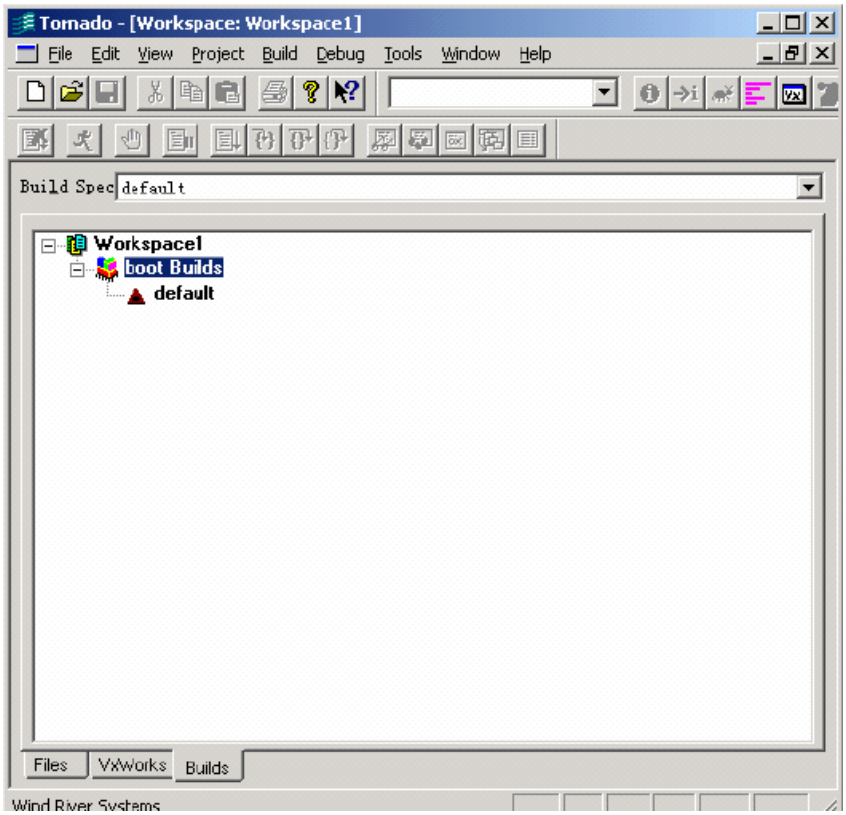


图3－11 设置default属性

8）双击default 图标，出现编译规则对话框。

图3－12 编译器及BSP属性对话框

9）点击“Rules”图标，在rule 的下拉列表中选择vxWorks 作为生成的镜像文件，操作界面如下图所示：

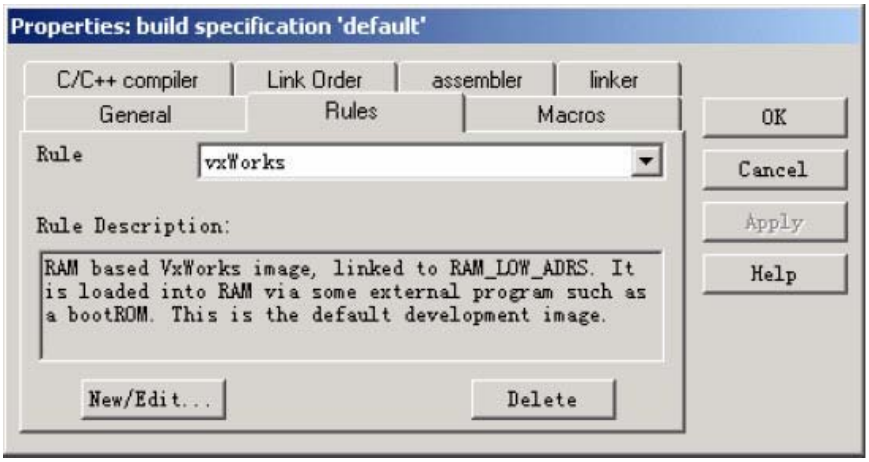


图3－13缺省目标映象文件选择属性框

10）在Tornado 的工作窗口中，点击右键，弹出上下文菜单，选择“build‘vxWorks’ ”，系统将编译连接生成vxWorks 映像文件。编译后的界面如下：

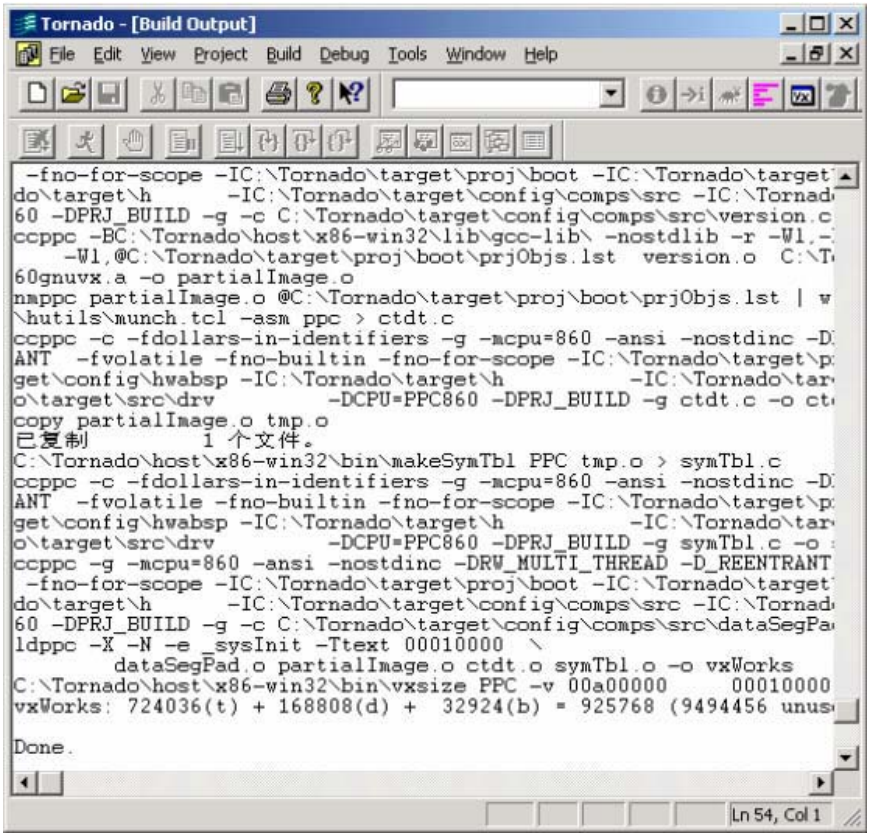


图3－14 编译过程显示的信息

编译生成的vxWorks 映像文件放在C:\Tornado\Target\proj\boot\default 目录下。

11）将宿主机和目标机连接起来，连接宿主机和目标机的串口线用交叉线，连接宿主机和目标机的网线也用交叉线。

12）启动超级终端

点击Windows 任务栏上的“开始”按钮，选择“程序”文件夹，然后选择“Tornado2”程序组，点击其中的VxWorks Com1 项，即可启动超级终端。启动后的界面如下：

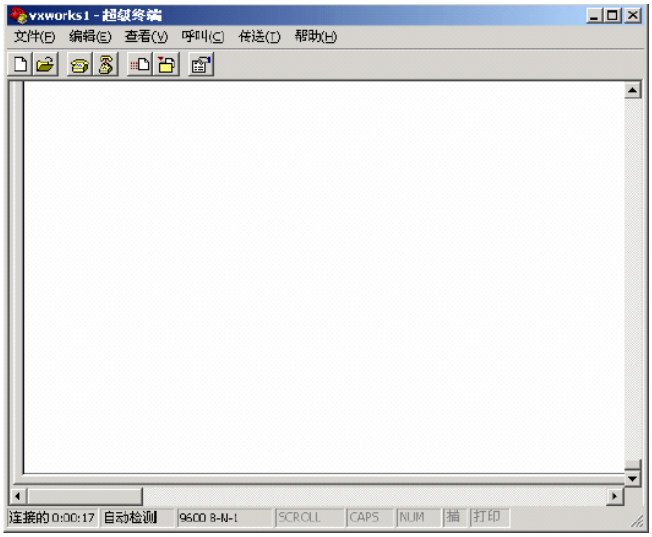


图3－15 启动超级终端

13）目标板加电，这是串口的出现的信息如下：

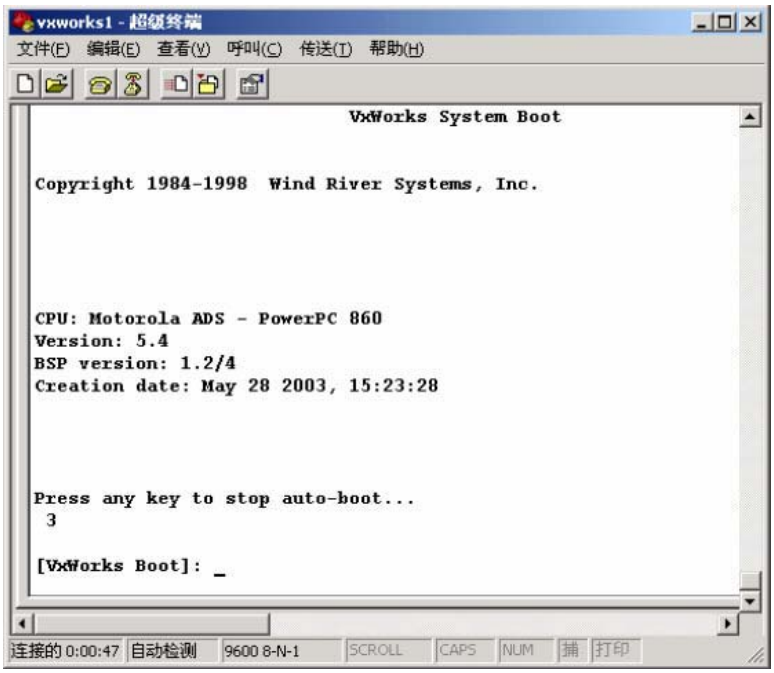


图3－16 启动目标板串口超级终端打印的信息

14）按任意键，进入VxWorks 启动参数的设置阶段，在【VxWorks Boot】：提示符后，敲入p 命令，可查看系统的默认设置。

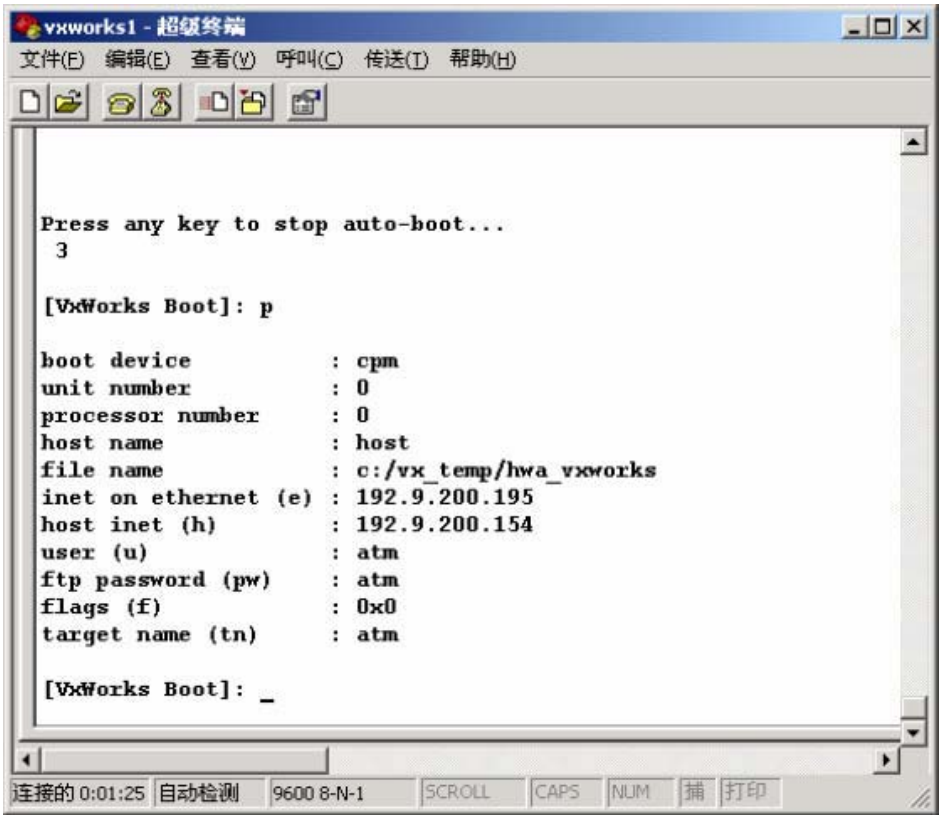


图3－17 目标板缺省的启动配置信息

15）为把vxWorks 映像文件从宿主机上下载到目标机上，要在[VxWorks Boot]：提示符后敲入c 命令，修改配置参数，修改后的参数配置如下图所示：

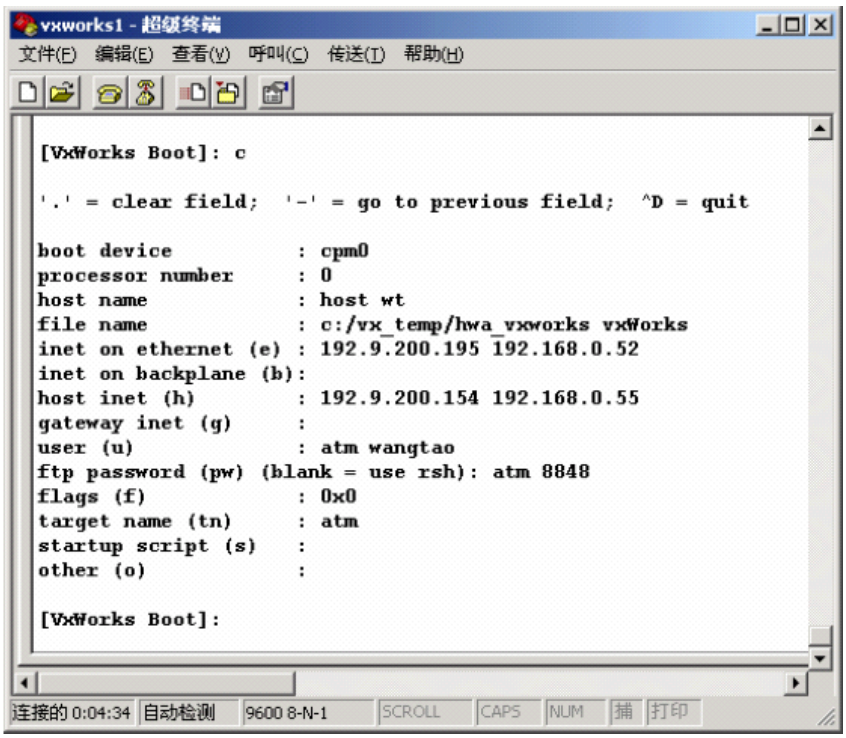


图3－18 修改目标板缺省的启动配置信息

其中

host name： 为开发程序的宿主机的机器名

file name： 为要下载到目标机的RAM 中的映像文件

inet on Ethernet（e）：为目标机要与主机通信使用的IP 地址，必须设置成与主机IP 地址在同一个网段。

Host inet（h）：为开发程序的宿主机的IP 地址

User （u）： 用于文件访问

ftp password（pw）：用于设置使用ftp 传输文件的设置用户的访问密码

16）启动ftp server，用于将映像文件传输到目标机

点击Windows 任务栏上的“开始”按钮，选择“程序”文件夹，然后选择“Tornado2”程序组，点击其中的FTP Server 项，即可启动FTP 服务器。启动后的界面如下：

图3－19 启动FTP Server

点击FTP Server 上Security 菜单，在下拉菜单中单击user/Rights Securitydialog 弹出下图的对话框：

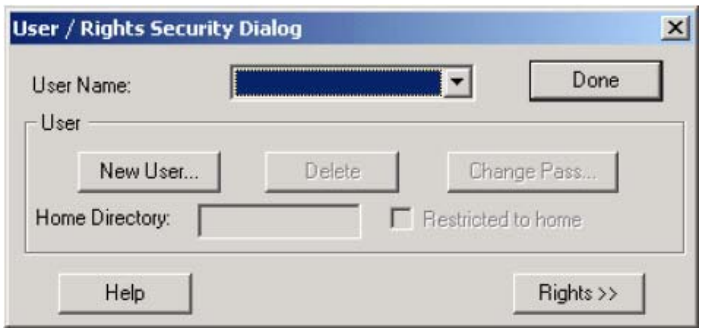


图3－20 设置User/Rights Security Dialog

单击New User…按钮，输入用户名wangtao（必须和超级终端设置完全一致），

图3－21 输入用户名

单击OK 按钮，弹出密码输入对话框，输入密码（必须和超级终端里的设置完全一致）。

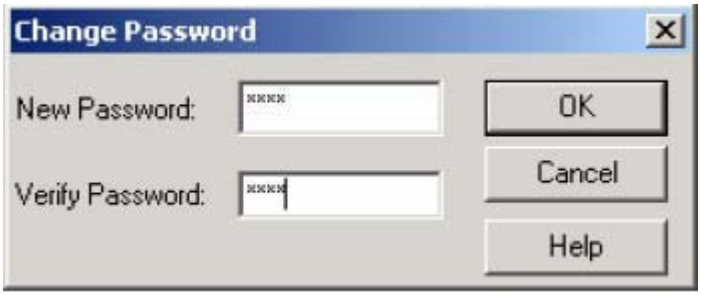


图3－22

单击OK，在Home Directory 输入框内输入vxWorks 映像文件在宿主机上存放的路径。

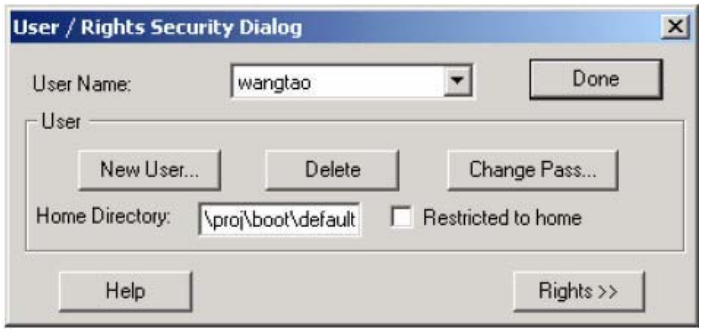


图3－23 输入用户密码

17）切换到超级终端，在【VxWorks Boot】：

命令提示符后敲入@命令执行下载操作。此时超级终端显示的信息如下图所示：

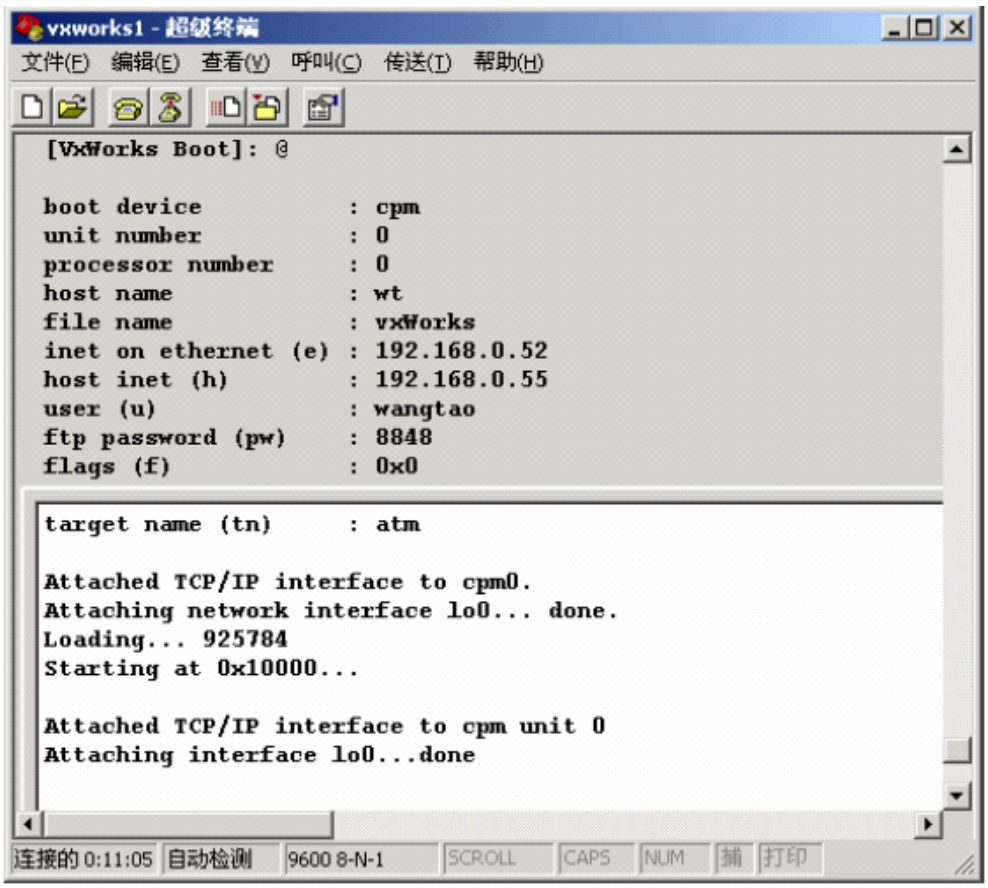


图 3－24 执行下载操作

3．创建downloadable 工程，添加并编译应用程序。

1） 单击File 下拉菜单，选择New Project…命令，弹出如下的对话框，选择Creat downloadbale applicaton modules for …

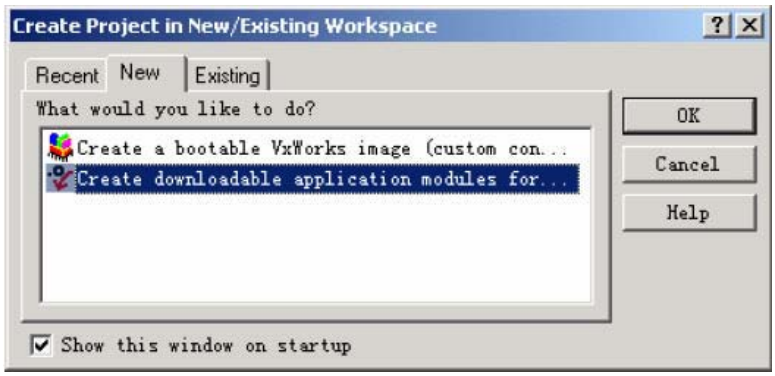


图3－25 创建Downloadable工程

2） 单击OK，弹出下图对话框，在对话框中输入工程的名字以及工程所在的路径和工作空间。

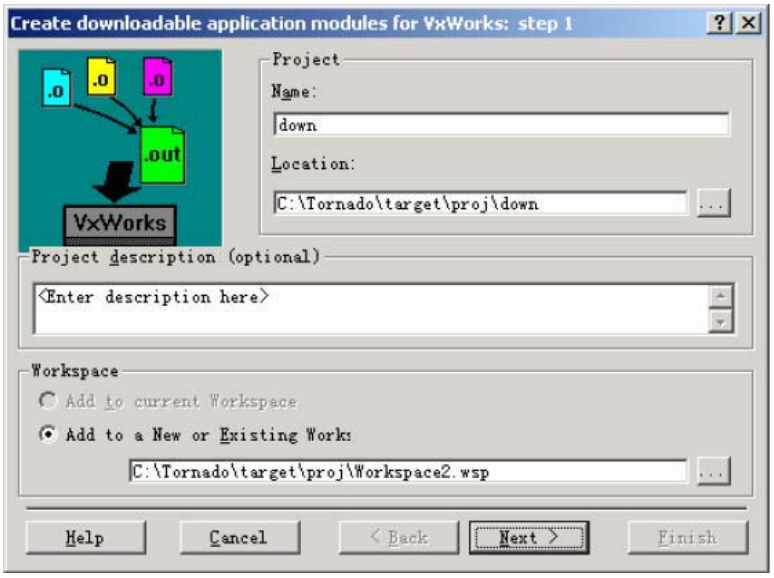


图3－26 设置Downloadable工程的属性

3） 单击Next 执行下一步操作，弹出如下画面，要求选择编译用的工具链。选择ppc860gnu。

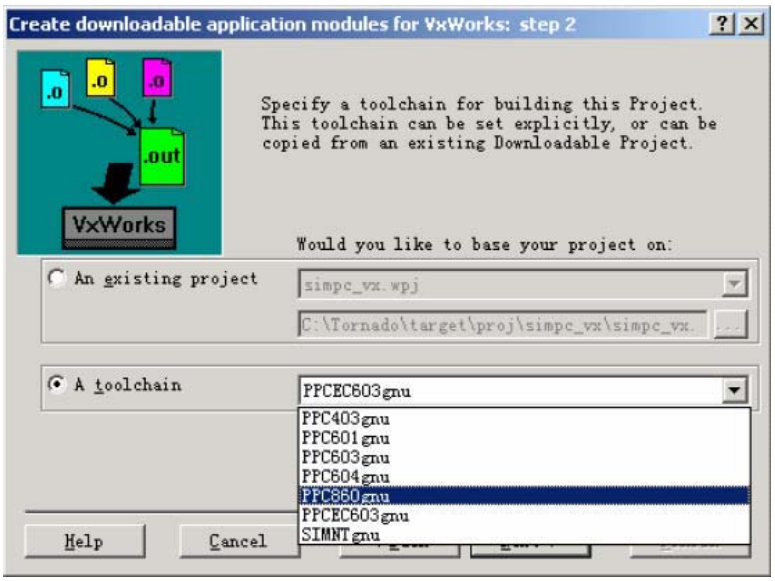


图3－27 选择编译器

4） 单击Next，弹出确认界面。

5） 单击Finish，完成工程的创建。

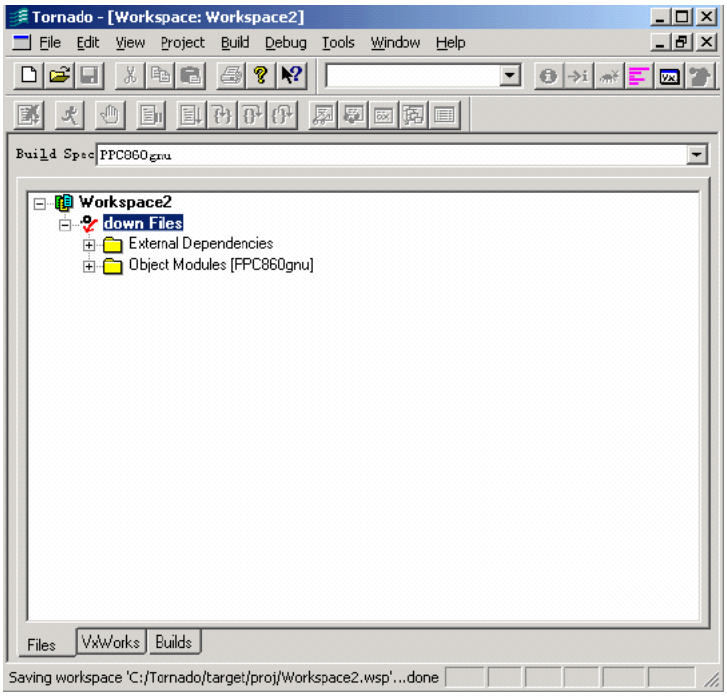


图3－28 工程创建完毕后的界面

6） 在活动窗口任意位置单击右键，弹出上下文菜单，选择其中的命令，加入要调试的源程序文件。单击右键弹出的上下文菜单如下图所示：

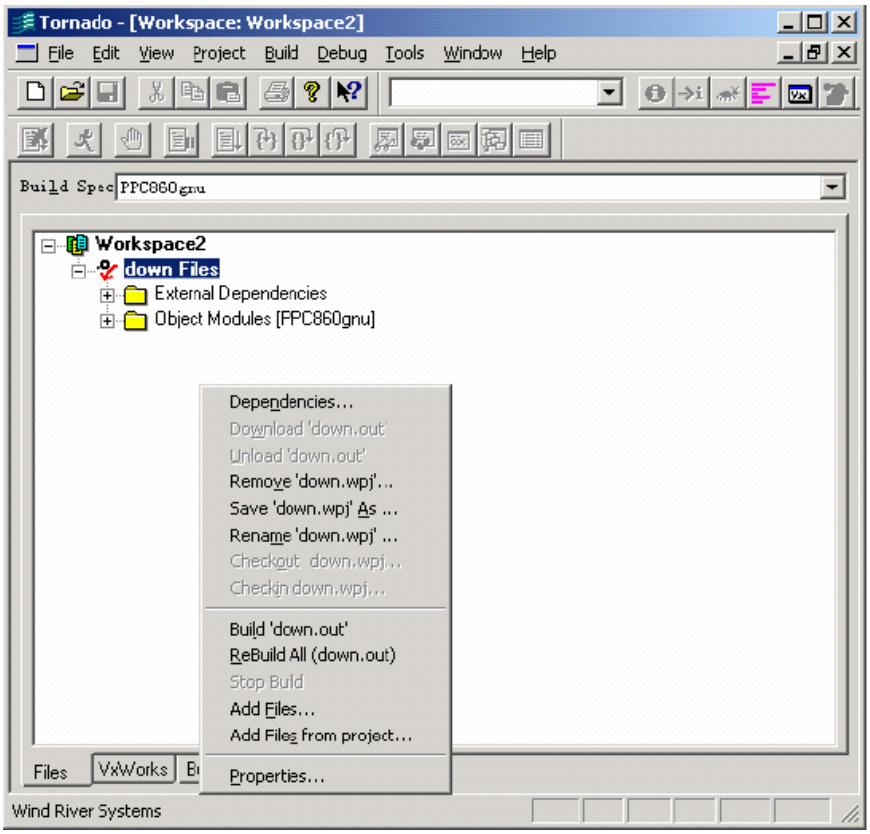


图3－29　工程的上下文菜单

7）选择Add File…命令弹出搜索源文件的对话框。选择合适的文件，单击Add 命令。

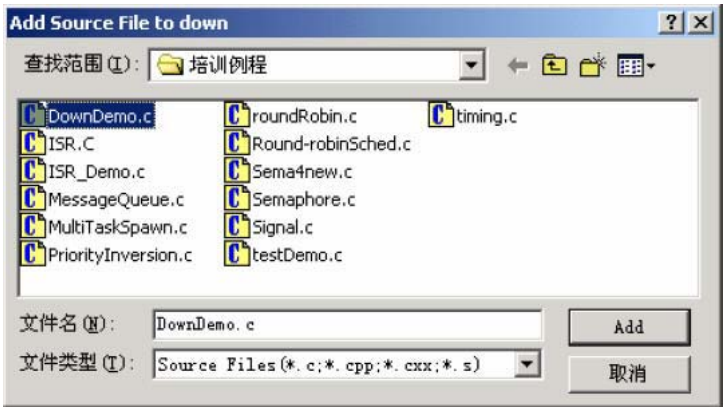


图3－3 0加入新的源文件

8）在加入源文件后，Tornado 的工作环境如下图所示：

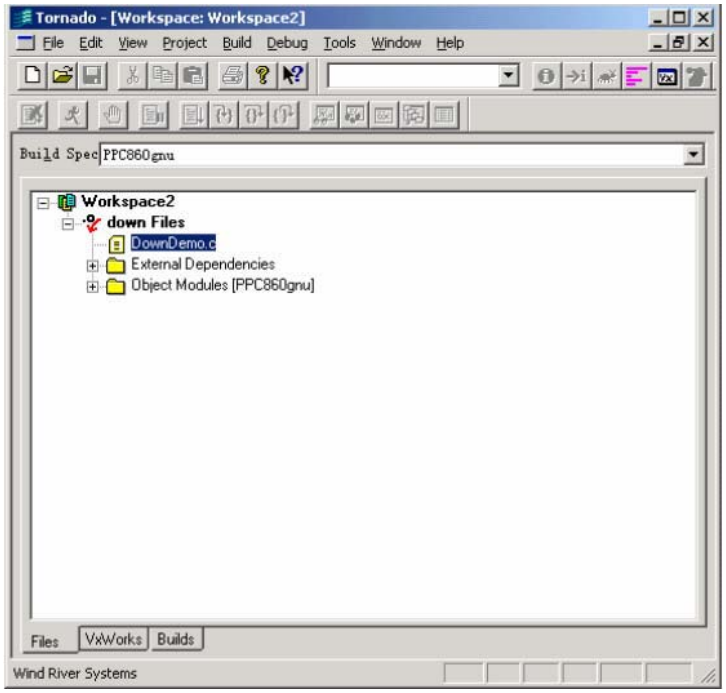


图3－31 加入源文件后的界面

9）鼠标右键点击新加入的DownDemo.c 文件弹出上下文菜单，选择Compile ‘DownDemo.c’命令，开始编译该新加入的源文件。

10）要下载该目标文件，须配置Target Server。

点击Tornado 主菜单中的Tools 下拉菜单选择Taget Server，接着选择Config…弹出配置Target Server 对话框。

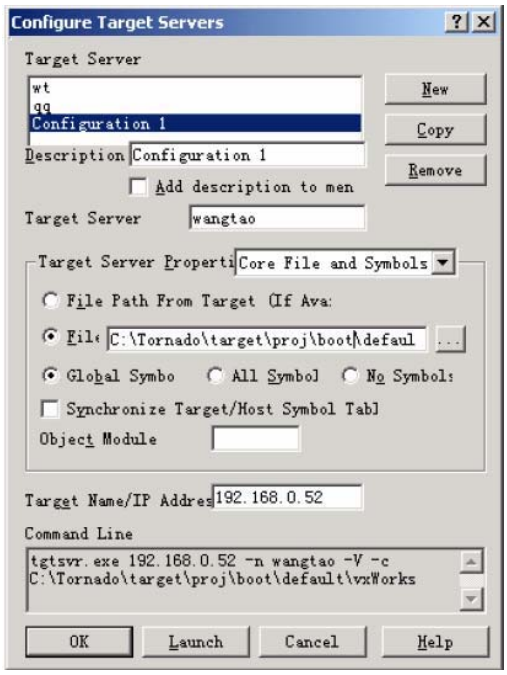


图3－32 设置Target Serve

单击New 命令新建一个Target Server，输入Target Server 名，在Target ServerProerti 右边的下拉列表中选择Core File and Symbols ，选中File，输入在boorable工程中生成的vxWorks 所在的路径及文件名。在Target Name/IP address 右边的方框中输入开发程序的宿主机的IP 地址。然后单击Launch 启动该Target Server。

11）然后在Tonado 的工作环境中，在生成的目标文件上单击鼠标右键弹出上下文菜单，选择“Download ‘DownDemo.o’ ”文件就可将该目标文件下载到目标机上。

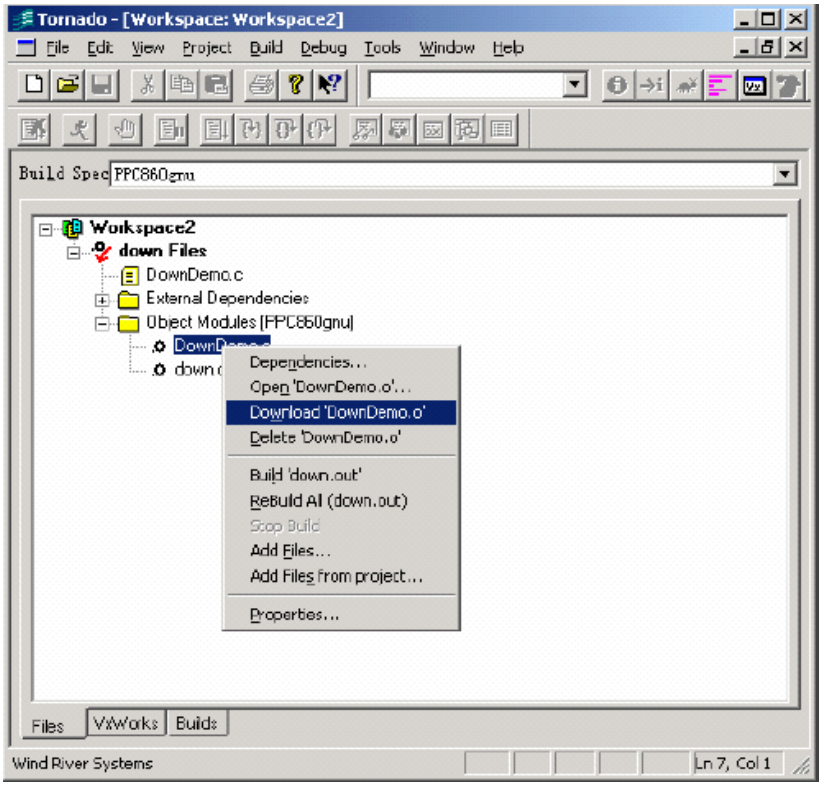


图3－33 下载目标文件到目标机中

12）在Tornado 开发环境中，单击工具条上的－>i 按钮，启动WindSh，启动后的界面如下：

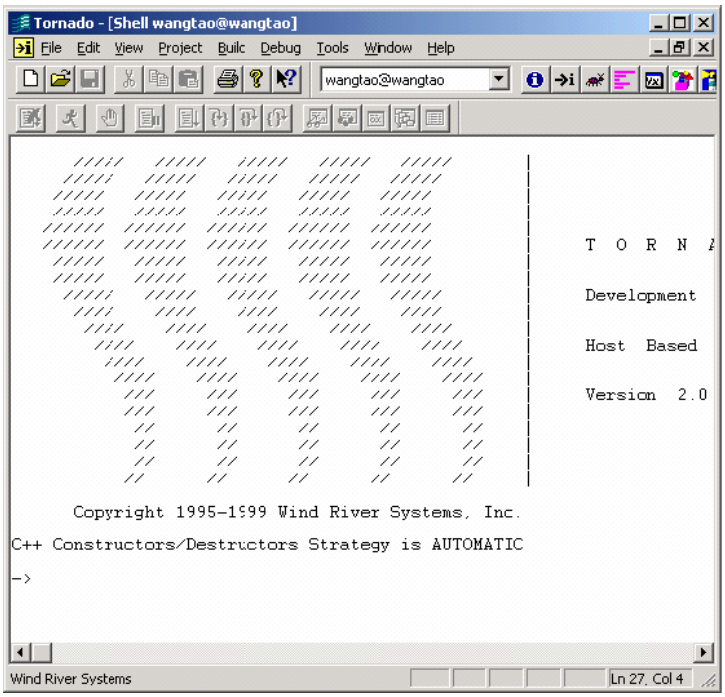


图3－34 启动WindSh

13）在WindSh 的－>提示符下，输入DownDemo.c 中的函数，可在超级终端中观察执行结果。

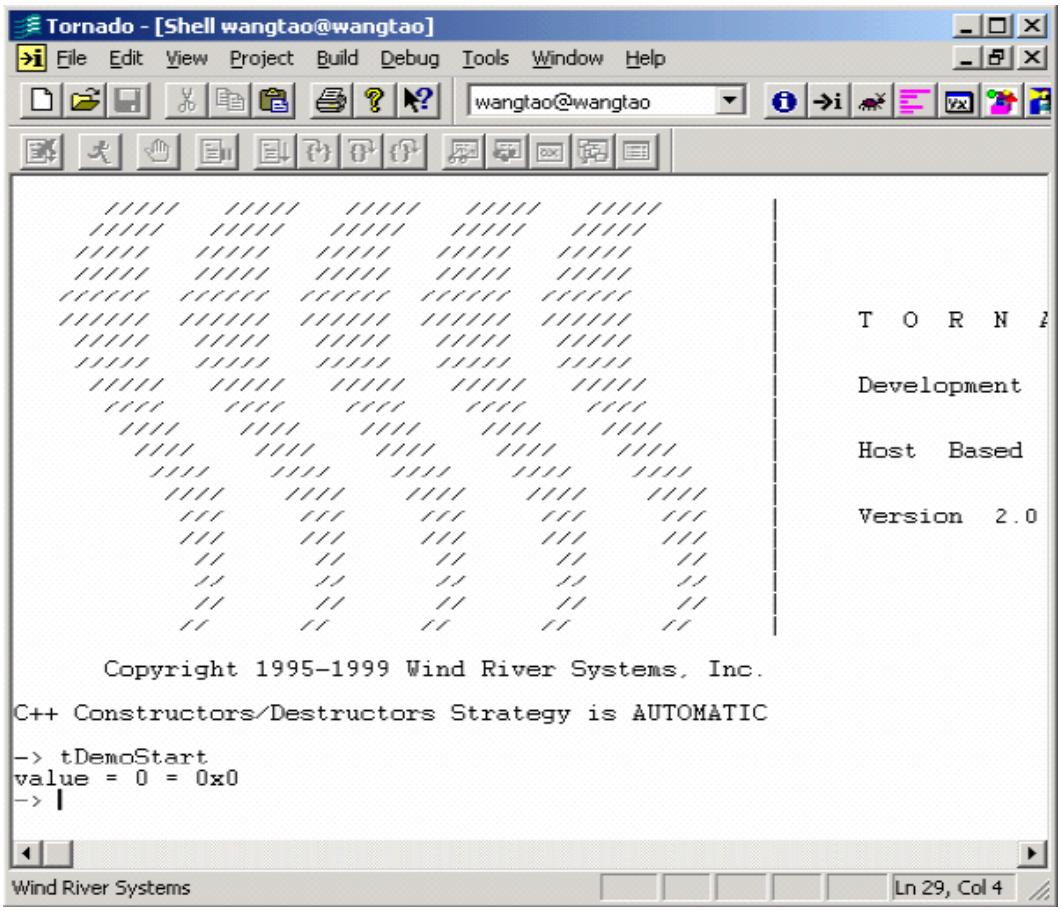


图3－35 执行源文件中的tDemoStart函数

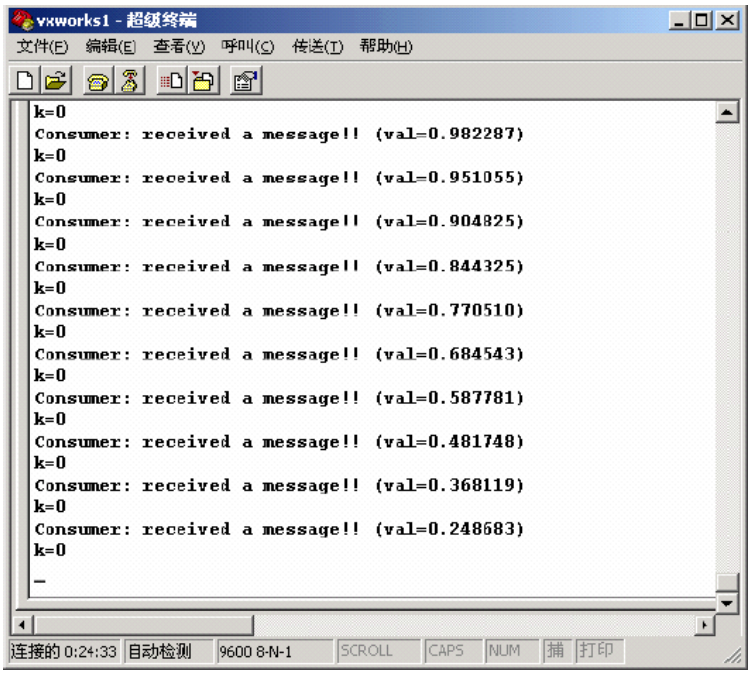


图3－36 超级终端打印的函数执行信息

4．将downloable工程中的文件添加到bootable工程中，生成最终的产品

1）将downloadable工程中的文件加入到bootable工程中，并修改usrAppInit（）函数，启动用户程序tDemoStart（）。重新编译连接生成新的vxworks按步骤2中的顺序重新下载到目标机中执行，观察程序运行结果。

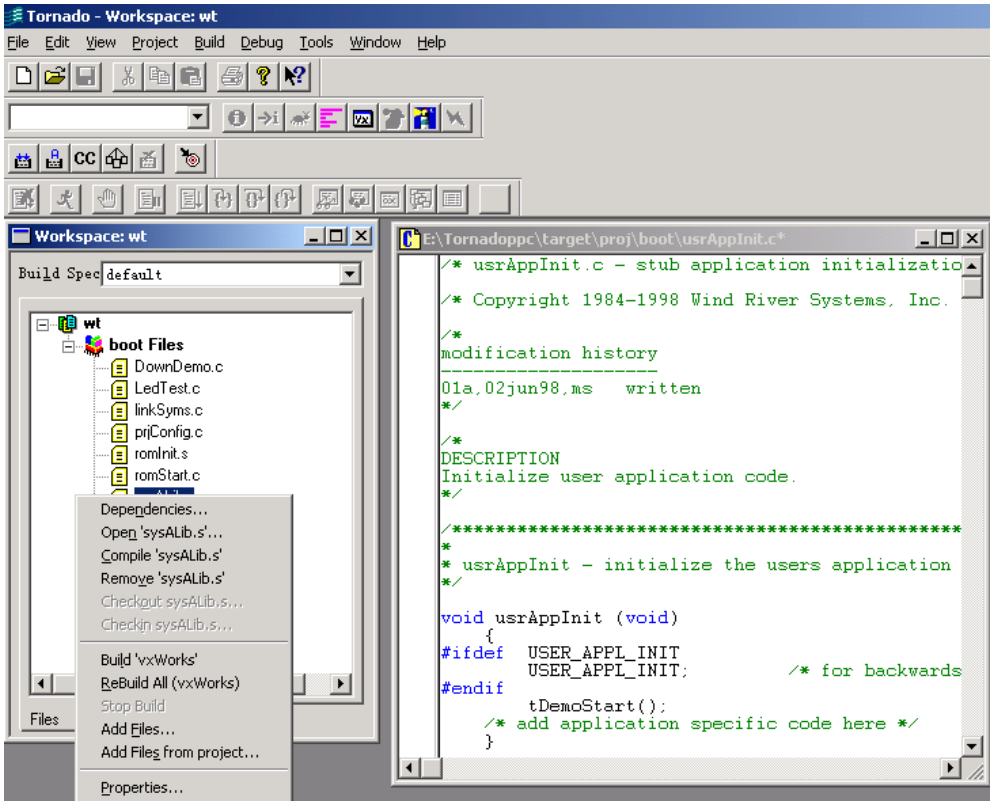


图3－37 重新编译链接含有应用程序的vxWorks映象文件

2）重新编译连接生成**vxWorks_romCompress.hex**，并烧录到flash中，烧录方法同步骤1所示。

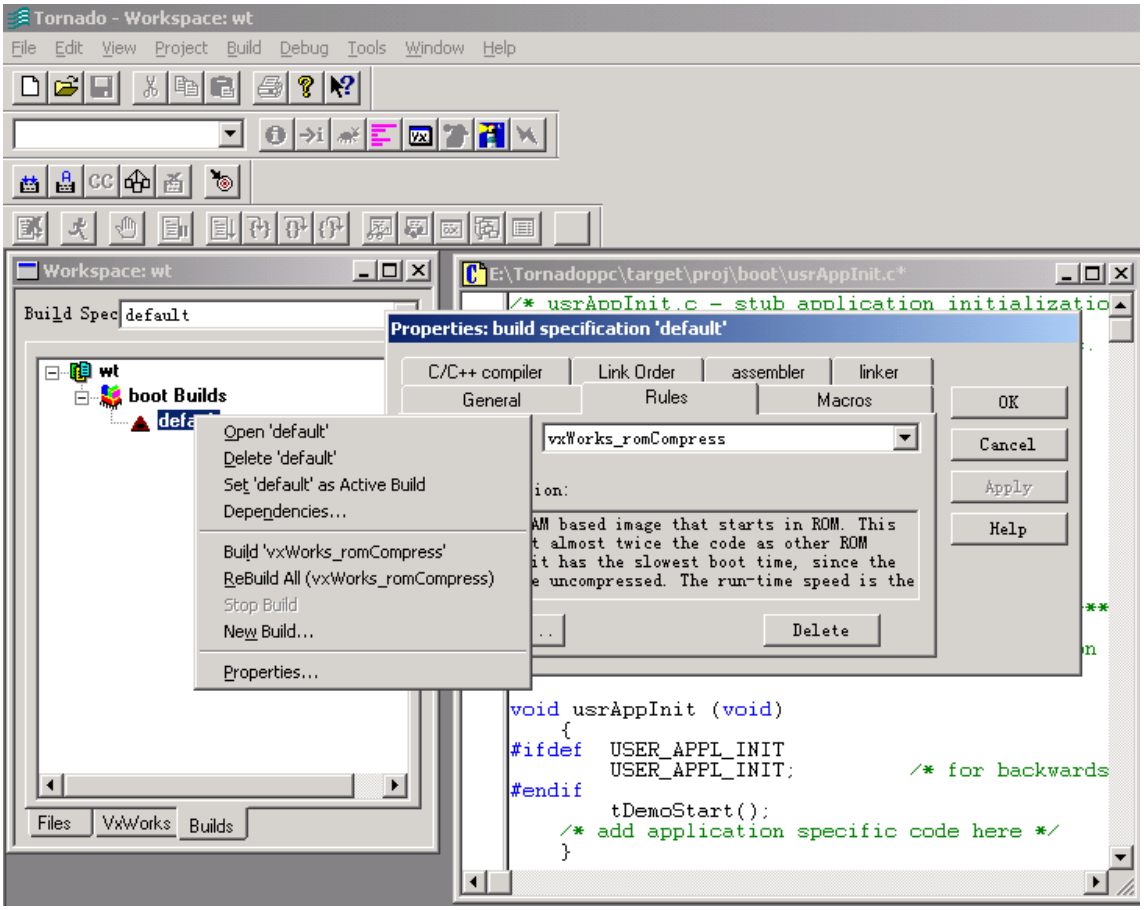


图 3－38 编译连接生成最终产品vxworks_romCompress.hex文件

以上即为一个完整的简单嵌入式软件的开发过程，但实际上在开发过程中可能会出现许多错误，这就牵涉到嵌入式软件的调试问题，BSP文件的调试要借助于硬件仿真器来进行，至于应用程序的调试可以借助于tornaod的调试和优化工具来进行。

4 任务管理

4.1 任务调度方式

多任务调度须采用一种调度算法来分配CPU给就绪态任务。Wind内核采用基于优先级的抢占式调度法作为它的缺省策略，同时它也提供了轮转调度法。基于优先级的抢占式调度，它具有很多优点。这种调度方法为每个任务指定不同的优先级。没有处于阻塞态或休眠态的最高优先级任务将一直运行下去。当更高优先级的任务由其他状态进入就绪态时，系统内核立即保存当前任务的上下文，切换到更高优先级的任务。

Wind 内核划分优先级为 256 级（0~255）。优先级 0 为最高优先级，优先级 255 为最低。当任务被创建时，系统根据给定值分配任务优先级。然而，优先级也可以是动态的，它们能在系统运行时被用户使用系统调用 taskPrioritySet()来加以改变，但不能在运行时被操作系统所改变。

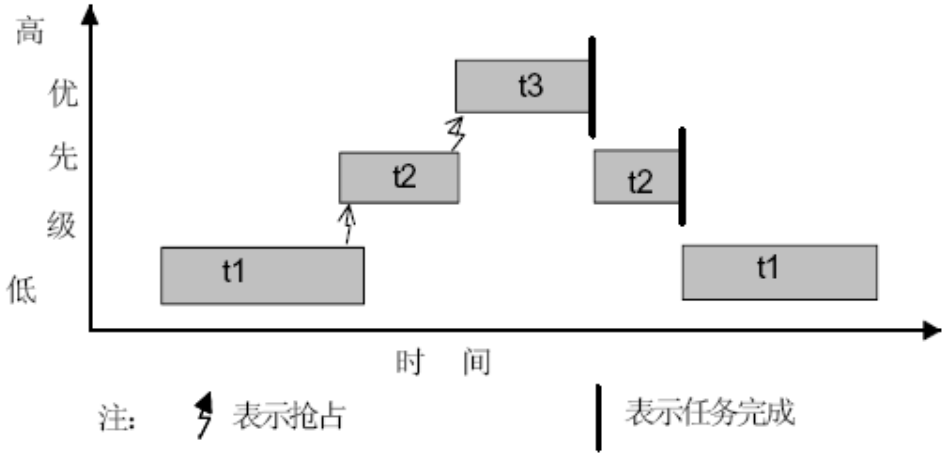


图 4－1 优先级抢占式调度

轮转调度法分配给处于就绪态的每个同优先级的任务一个相同的执行时间片。时间片的长度可由系统调用 KernelTimeSlice()通过输入参数值来指定。很明显，每个任务都有一运行时间计数器，任务运行时每一时间滴答加 1。一个任务用完时间片之后，就进行任务切换，停止执行当前运行的任务，将它放入队列尾部，对运行时间计数器置零，并开始执行就绪队列中的下一个任务。当运行任务被更高优先级的任务抢占时，此任务的运行时间计数器被保存，直到该任务下次运行时。

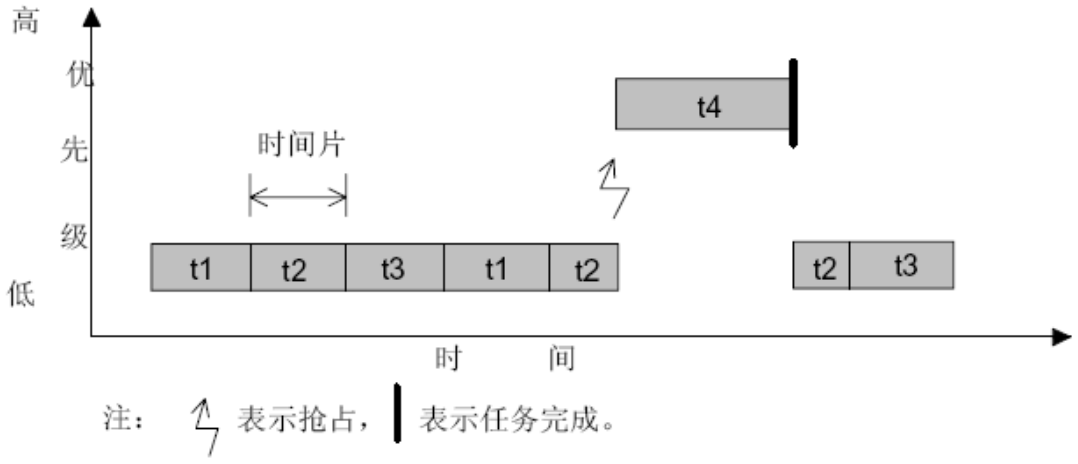


图 4－2 时间片轮转调度

4.2 任务管理相关函数

VxWorks内核的任务管理提供了动态创建、删除和控制任务的功能，具体实现通过如下一些系统调用：

taskSpawn() 创建（产生并激活）新任务
taskInit() 初始化一个新任务
taskActivate() 激活一个已初始化的任务
taskName() 由任务ID号得到任务名
taskNameToId()由任务名得到任务ID号
taskPriorityGet() 获得任务的优先级
taskIsSuspended() 检查任务是否被挂起
taskIsReady() 检查任务是否准备运行
taskTcb() 得到一个任务控制块的指针
taskDelete() 中止指定任务并释放内存（仅任务堆栈和控制块）
taskSafe() 保护被调用任务
taskSuspend() 挂起一个任务
taskResume() 恢复一个任务
taskRestart() 重启一个任务
taskDelay() 延迟一个任务

本次综合设计中，经常要用到 taskSpawn()函数，所以在此进行进一步详细阐述。作为设计中用到的最为基本的函数，应该对整个函数结构和几个主要的参量进行重点记忆和理解。

taskSpawn()函数原型为：taskSpawn(name，ptiority，options，stacksize，function，arg1…arg10)。其中，name即指创建的任务名字。ptiority指这个任务的优先级，优先级越高的就越容易先于其他任务执行。优先级从0－255共256个，数字越低，优先级越高。Stacksize为创建任务指定堆栈大小。options 选项可以指定为以下几个宏定义的或，**VX_FP_TASK** (0x0008)、**VX_PRIVATE_ENV** (0x0080)、**VX_UNBREAKABLE** (0x0002)、**VX_NO_STACK_FILL** (0x0100)。function是指任务的入口函数指针，就是说内核调度该任务时执行的函数。后面的10个参数，是指function可以携带10个入口参数。

4.3 任务创建程序示例

以下程序示例创建10个任务并打印各自的ID号。最先创建的任务先打印自己的ID号，如果在创建时优先级选项改为90－i，则最先创建的任务最后打印自己的ID号，这是由于当任务创建完成后处于就绪态，内核总是选择就绪队列中优先级最高的任务来执行。

```
#define ITERATIONS 10
void print(int i);
spawn_ten()/*Subroutine to perform the spawning*/
{
int i,taskId;
for(i=0;i<ITERATIONS;i++)
taskId=taskSpawn("tprint",90+i,0x100,2000,print,i,0,0,0,0,0,0,0,0,0,0);
}

void print(int i)
{
```

```
printf("Hello,I am task %d\n",i);
}
```

按照第三章讲述的使用 tornado 软件开发嵌入式软件的步骤 2－3，将源文件编译后下载到目标机中，在 windSh 下输入 spawnTenTask 函数，则得到如下运行结果：

```
Hello,I am task 0
Hello,I am task 1
Hello,I am task 2
Hello,I am task 3
Hello,I am task 4
Hello,I am task 5
Hello,I am task 6
Hello,I am task 7
Hello,I am task 8
Hello,I am task 9
```

如果创建任务的函数改写为：

```
taskId=taskSpawn("tprint",90+i,0x100,2000,print,i,0,0,0,0,0,0,0,0,0,0);
```

则得到如下的运行结果：

```
Hello,I am task 9
Hello,I am task 8
Hello,I am task 7
Hello,I am task 6
Hello,I am task 5
Hello,I am task 4
Hello,I am task 3
Hello,I am task 2
Hello,I am task 1
Hello,I am task 0
```

5 任务通信

5.1 任务间常见通信方式

VxWorks支持各种任务间通信机制，提供了多样的任务间通信方式，主要有如下几种：

- 共享内存，主要是数据的共享；
- 信号量，用于基本的互斥和任务同步；
- 消息队列和管道，单CPU的消息传送；
- Socket和远程过程调用，用于网络间任务消息传送；
- 信号，用于异常处理。

◇ 共享内存

任务间通信的最简单的方法是采用共享存储区，也即相关的各个任务分享属于它们的地址空间的同一内存区域。因为所有任务都存在于单一的线性地址空间，任务间共享数据。全局变量、线性队列、环形队列、链表、指针都可被运行在不同上下文的代码所指向。

◇ 消息队列

现实的实时应用由一系列互相独立又协同工作的任务组成。信号量为任务间同步和通信提供了高效方法。单处理器中任务间消息的传送采用消息消息队列。消息机制使用一个被各有关进程共享的消息队列，任务之间经由这个消息队列发送和接收消息。

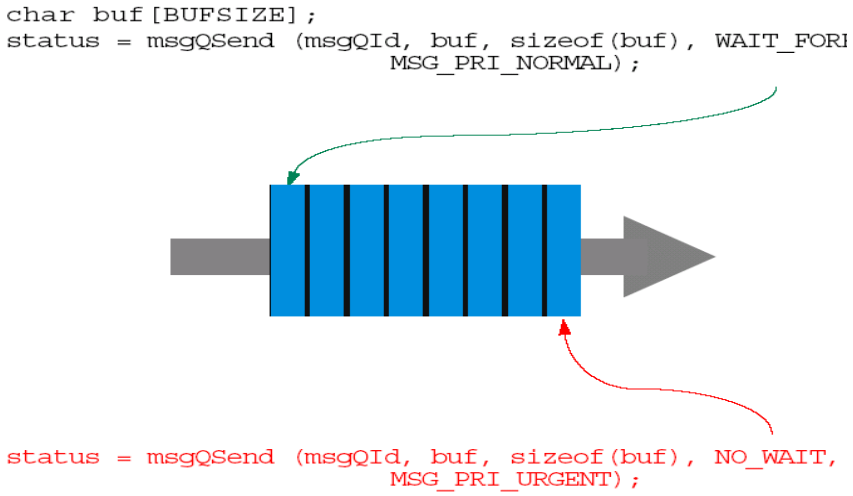


图 5－1 使用消息队列进行通信

◇ 管道

在vxworks中，管道是Driver pipeDrv所管理的虚拟I/O设备，因此我们可以使用标准的IO过程（open、read、write、ioctl）来操作它。

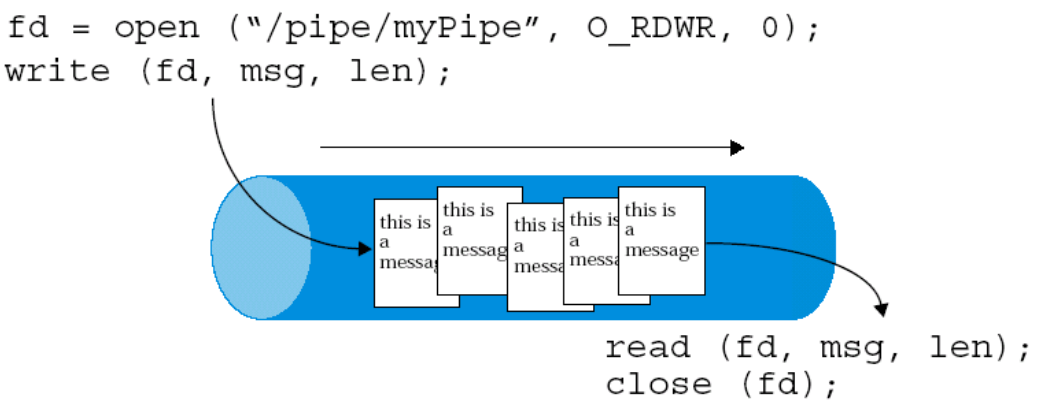


图5－2 使用管道进行任务间通信

5.2 消息队列

vxWors 提供与消息队列有关的的函数主要有：

msgQCreate() 创建（产生并激活）消息队列

msgQDelete() 删除消息队列

msgQSend() 向消息队列发送消息

msgQReceive() 从消息队列接收消息

*msgQCreate()*的函数原型为MSG_Q_ID msgCreate（int maxMsgs，int maxMsgLength，int options），其中参数maxMsgs定义了消息队列的最大消息数，maxMsgLength定义了消息最大长度，options为消息队列选项，为MSG_Q_FIFO(0x00)或者MSG_Q_PRIORITY(0x01)。消息队列成功创建后返回一个消息队列ID号。

用*msgQCreate()*创建好一个消息队列后，既可以调用*msgQSend()*向消息队列发送消息。*msgQSend()*的函数原型为:STATUS msgQSend(MSG_Q_ID msgQId,char *Buffer,UINT nBytes,int timeout,int priority)，其中参数msgQId为消息队列ID号，buffer为待发送的消息指针，nBytes为消息的长度，priority为消息的优先级，可以指定为MSG_PRI_NORMAL（0x00）表示将消息放在消息队列的尾部，或者指定为**MSG_PRI_URGENT** (0x01)将消息放在消息队列的首部。Timeout用于设置超时等待间隔，可以tick数，或者设置为**NO_WAIT** (0x00)不等待或者设置为**WAIT_FOREVER** (-1)永远等待。

调用msgQReceive()对发送来的消息进行接收，其函数原型如下。int msgQReceive(MSG_Q_ID msgQId, char *buffer,UINT maxNBytes,int timeout)其中参数msgQId为消息队列ID号，buffer为取出的消息存放地址指针，maxNBytes为buffer缓冲区的长度，timeout定义当消息队列为空时超时等待间隔，设置同msgQsend相同。

msgQDelete() 删除消息队列。其函数原型为STATUS msgQDelete(MSG_Q_ID msgQId)，msgQId为要删除的消息队列的ID号。

5.3 消息队列通信示例

以下给出使用消息队列实现任务间通信的一个实例。在该程序中，生产者任务通过消息队列向消费者任务传递8条消息。读者可以通过修改消息队列各个选项来体会消息队列的使用方法。

```
#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 */
struct msg {                              /* data structure for msg passing */
        int tid;                          /* task id */
```

```
        int value;                          /* msg value */
        };
    LOCAL MSG_Q_ID msgQId;                  /* message queue id */
    LOCAL int numMsg = 8;                   /* number of messages */
    LOCAL BOOL notDone;                     /* Flag to indicate the completion    of this
demo */
    LOCAL STATUS producerTask (); /* producer task */
    LOCAL STATUS consumerTask (); /* consumer task */

    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. */
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. */
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);
}
```

按照第三章讲述的使用 tornado 软件开发嵌入式软件的步骤 2－3，将源文件编译后下载到目标机中，在windSh 下输入 msgQDemo()函数，则得到如下运行结果：

```
producerTask started: task id = 0x4fce5c0
ProducerTask: tid = 0x4fce5c0, produced value = 10
ProducerTask: tid = 0x4fce5c0, produced value = 20
ProducerTask: tid = 0x4fce5c0, produced value = 30
ProducerTask: tid = 0x4fce5c0, produced value = 40
ProducerTask: tid = 0x4fce5c0, produced value = 50
ProducerTask: tid = 0x4fce5c0, produced value = 60
ProducerTask: tid = 0x4fce5c0, produced value = 70
ProducerTask: tid = 0x4fce5c0, produced value = 80

ConsumerTask: Started -  task id = 0x4fb56b8
ConsumerTask: Consuming msg of value 10 from tid = 0x4fce5c0
ConsumerTask: Consuming msg of value 20 from tid = 0x4fce5c0
ConsumerTask: Consuming msg of value 30 from tid = 0x4fce5c0
ConsumerTask: Consuming msg of value 40 from tid = 0x4fce5c0
ConsumerTask: Consuming msg of value 50 from tid = 0x4fce5c0
ConsumerTask: Consuming msg of value 60 from tid = 0x4fce5c0
ConsumerTask: Consuming msg of value 70 from tid = 0x4fce5c0
ConsumerTask: Consuming msg of value 80 from tid = 0x4fce5c0
```

5.4 管道

创建的管道正常是命名的I/O设备。任务能够使用标准的I/O程序来打开、读写管道，也可以调用ioctl来设置控制属性。象其他的I/O驱动那样，当从一个没有数据的管道读数据时，任务会阻塞。象对消息队列操作一样，ISR也能向管道写，但是不能从管道读。管道还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设备集合之一数据可用。Selcet函数也能够与其他的异步IO设备一起工作，包括socket和串行设备，因此使用select（）任务能够同时等待几个管道、socket和串行设备集合上的数据。管道也能实现任务间的客户机/服务器通信模型

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。如果试图向一个满的管道执行写操作，任务将阻塞，如果试图从一个空的管道执行读操作，任务也将阻塞等待消息的到达。创建管道的函数原型时是STATUS pipeDevCreate (name, nMessages,nBytes)，其中参数name是指创建管道设备的名字，比如 “/pipe/yourName”； nMessages 为管道中可存放的最大消息数，nBytes为每条消息

的最大长度。

5.5 管道通信示例

以下示例程序演示了如何利用管道实现客户机/服务器模型。

```
typedef struct
{
VOIDFUNCPTR routine;
int arg;
} MSG_REQUEST;

#define TASK_PRI                254         /* server priority */
#define TASK_STACK 20000    /* server stack space */
#define PIPE_NAME                   "/pipe/server"
#define NUM_MSGS                    10

int pipeFd;

LOCAL void pipeServer ();

/************************************************
* serverStart -- Initializes a server task to* execute functons at a low priority. Uses pipes as    the communication mechanism. *
* RETURNS: OK or ERROR on failure.
*/

STATUS serverStart (void)
    {
    /* Create the pipe device */

    if (pipeDevCreate (PIPE_NAME, NUM_MSGS,
            sizeof (MSG_REQUEST)) == ERROR)
        return (ERROR);

    /* Open the pipe */

    if ((pipeFd = open (PIPE_NAME, UPDATE, 0))
            == ERROR)
        return (ERROR);

    /* Spawn the server task */

    if (taskSpawn ("tServer", TASK_PRI, 0,
                    TASK_STACK, (FUNCPTR)pipeServer,
                    0,0,0,0,0,0,0,0,0,0) == ERROR)
```

```
        {
        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,
    int arg
    )
    {
    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 (void)
```

```
{
MSG_REQUEST msgRequest;
while (read (pipeFd, (char *)&msgRequest,
                        sizeof (MSG_REQUEST)) > 0)
    (*msgRequest.routine) (msgRequest.arg);
}
```

按第三章讲述的使用tornado软件开发嵌入式软件的步骤2－3，将上述程序经编译后下载到目标机中，在windsh下启动serverStart（）函数，然后执行serverSend（）函数，其中指定serverSend（）函数的两个入口参数为目标机中存在的函数以及该函数的入口参数，利如执行serverSend（reboot，0）函数，则tServer会响应该请求，重新启动目标板。

6 任务同步

6.1 任务间同步方式

实时操作系统中对于共享资源的保护与任务的同步协作，一般都提供了信号量机制。vxworks 提供的信号量分为三种，二值信号量，计数信号量，互斥信号量。二值信号量常用于各相互协作任务间的同步，计数信号量常用于管理多个共享资源的使用。互斥信号量常用于对单一共享资源的保护。

二进制同步信号量具有一定的抽象性，理解起来有点困难，这里举一个简单的例子进行说明。例如我们可以把公共汽车司机开车可以看成一个任务，乘客上车看成一个任务，司机具体什么时候开车要等待售票员的哨音，司机听到哨音就开车，听不到就继续等待，这里的哨音就相当于用于同步的二进制信号量。在VXWORKS中，也同样是运用了这个简单的规则。当优先及高的程序想要执行时，但是由于没有得到信号量，所以阻塞了，只有等待其他任务释放了信号量量之后高优先级的任务才能解除阻塞得以继续执行。这种一个任务的执行依赖于另一个任务执行的情况就是同步

6.2 二进制信号量

vxWorks提供了用于任务间同步的二进制信号量操作函数：semBCreate（）、semTake（）、semGive（）、semDelete（）函数。

semBCreate（int options，SEM_B_STATE initialState）：分配并初始化一个二进制信号量。其中参数options **可以设置为SEM_Q_PRIORITY** (0x1) 或者 **SEM_Q_FIFO** (0x0),用于指示阻塞在该信号量上的任务阻塞队列的类型。参数initialState可以设置为**SEM_FULL** (1) 或者**SEM_EMPTY** (0)，用于设置创建的二进制信号量的初始状态，一般设置为**SEM_EMPTY。当创建成功后返回一个信号量ID号。**

semTake(SEM_ID semId, in timeout)用于获取二进制信号量，其中第一个参数为要获取的信号量的 ID 号，第二个参数为当信号量初始状态为 **SEM_EMPTY，获取信号量的任务等待信号量的时间，可以设置为三种情况之一：ticks、或者 WAIT_FOREVER** (-1)或者 **NO_WAIT** (0)。

semGive(SEM_ID semId)函数用于释放二进制信号量，函数的参数 semId 为要释放的二进制信号领的 ID 号。

semDelete(SEM_ID semId)用于删除二进制信号量，参数 semId 为要删除的二进制信号领的 ID 号。

当使用二进制信号量用于任务间同步时，首先由执行执行创建二进制信号量。synSem = semBCreate(SEM_Q_FIFO(或SEM_PRIORITY) , SEM_EMPTY)，用于同步的二值信号量必须初始化为空。紧接着由同步触发任务（初始化任务）将所有的必须初始化工作做完后，释放信号量semGive (synSem)，唤醒后面的具体执行任务，具体执行任务在任务的开始执行处提取信号量semTake (synSem)。如下面程序框架所示。

```
synSem = semBCreate( SEM_Q_FIFO(或SEM_PRIORITY) , SEM_EMPTY);
taskSpawn ("tTaskA", TaskA_TASK_PRI, 0,TASK_STACK_SIZE, (FUNCPTR) taskA, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
taskSpawn ("tTaskB", TaskB_TASK_PRI, 0,TASK_STACK_SIZE, (FUNCPTR) taskB, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
taskA ( parameter )
{
FOREVER /*  任务为一个无限循环*/
{
```

```
相应的初始化和准备工作
. . . .
        semGive(synSem);/*信号量释放后，唤醒较高优先权的执行任务*/
    }
}
taskB (parameter)
{
FOREVER
  {
semTake (synSem);
. . . .
执行具体的操作
. . . .
}/*开始下一个循环时，由于得不到信号量，阻塞，让出CPU，任务A可以执行*/
}
```

下面用示意图说明信号量在释放和获取时，任务的执行情况。当某任务获取二进制信号量时，则检查该二进制新量是否可用，如果可用则任务继续执行，semTake（）返回OK，如果不可用，则任务阻塞在该信号量上，等待信号量变为有效，如果在等待的时间间隔内还没有获取到二进制信号量，则任务解除阻塞，semTake返回ERROR,如果在等待的时间间隔内信号量被释放了，则该任务解除阻塞，semTake返回OK。

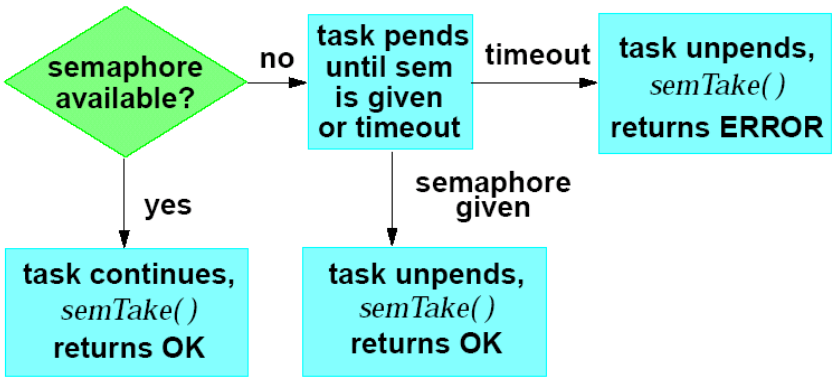


图6－1 获取二进制同步信号量

当任务释放二进制信号量时，如果有其他任务阻塞在该信号量上，则处于阻塞队列首部的那个任务解除阻塞变为就绪态，信号量继续保持不可用。如果没有任务阻塞在该信号量上，则信号量变为可用。

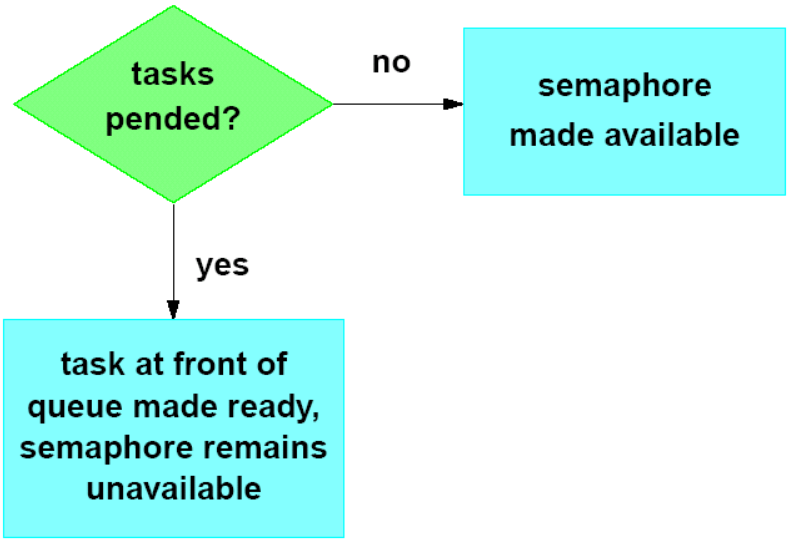


图6－2 释放二进制同步信号量

有些任务需要与外界交互（数据采集，信息接收等）这时可以在中断处理程序ISR中加入对二值同步信号量的释放语句（ISR中不允许加入获取二进制信号量的语句，否则会引起中断服务程序的阻塞，从而破坏系统的正常运行），由中断程序来唤醒相应的处理程序。这样，平时该任务处于阻塞状态，系统运行其它的任务，等到有了外部输入，ISR释放二进制信号量，从而任务解除阻塞被唤醒运行。 同步问题中往往会有多个任务等待在一个同步点上，当一个任务触发同步点时，所有的任务都处于就绪状态。这时使用semGive(synSem)，只能将其中一个任务唤醒进入就绪队列，而其他等待该信号量的任务仍在阻塞，将无法进入就绪队列。使用semFlush( )可以唤醒所有阻塞在该信号量上的任务，使其进入就绪队列，等待被执行。这也就是同步机制中的广播。

6.3 二进制信号量同步程序示例

下列程序使用了两个二进制信号量实现了任务A和任务B之间的双向同步。

```
#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.**************************************/
STATUS synchronizeDemo ()
{
notDone = TRUE;
/* semaphore semId1 is availble 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 ("syncronizeDemo: Error in deleting semId1 semaphore");
return (ERROR);
}
if (semDelete (semId2) == ERROR)
perror ("syncronizeDemo: 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.***************/
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'm done, taskB can now 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.****************/
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);
}
```

按第三章讲述的使用tornado软件开发嵌入式软件的步骤2－3，将上述程序经编译后下载到目标机中，在windsh下启动synchronizeDemo（）函数得到如下运行结果：taskA: Started first by taking the semId1 semaphore - 1 times

```
This is task    <tTaskA> : 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    <tTaskB> : 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    <tTaskA> : Event A now done
taskA: I'm done, taskB can now proceed; Releasing semId2 semaphore

taskB: Synchronized with taskA's release of semId2 - 2 times
This is task    <tTaskB> : 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    <tTaskA> : 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    <tTaskB> : Event B now done
taskB: I'm done, taskA can now proceed; Releasing semId1 semaphore
```

7 任务互斥

7.1 任务互斥

可以被一个以上任务使用的资源叫做共享资源。为了防止数据被破坏，每个任务在与共享资源打交道时，必须独占该资源，这叫做互斥。使用互斥信号量机制可以保护对共享资源的正确使用，但要注意防止优先级反转的出现，同时互斥信号量与共享内存相结合可以实现任务间的通信。当申请不到互斥信号量的任务即转入阻塞状态被放入等待信号量的队列中，让出对 CPU 的使用。若任务得到了互斥信号量即可对共享资源进行访问，这时系统仍然允许抢占。 下面给出一个简单的例子说明使用互斥信号量实现对共享资源进行保护的重要性。

当多个任务共享输入输出设备时，信号量特别有用。可以想象，如果允许两个任务同时给打印机送数据时会出现什么现象。打印机会打出相互交叉的两个任务的数据。例如任务 1 要打印“I am Task!”，而任务 2 要打印“I am Task2!”可能打印出来的结果是：“I Ia amm T Tasask k1!2!”

在这种情况下，使用互斥信号量并给信号量初始化为有效，规则很简单，要想使用打印机的任务，先要得到该资源的信号量。图 2.10 两个任务竞争得到排它性打印机使用权，图中信号量用一把钥匙表示，想使用打印机先要得到这把钥匙。

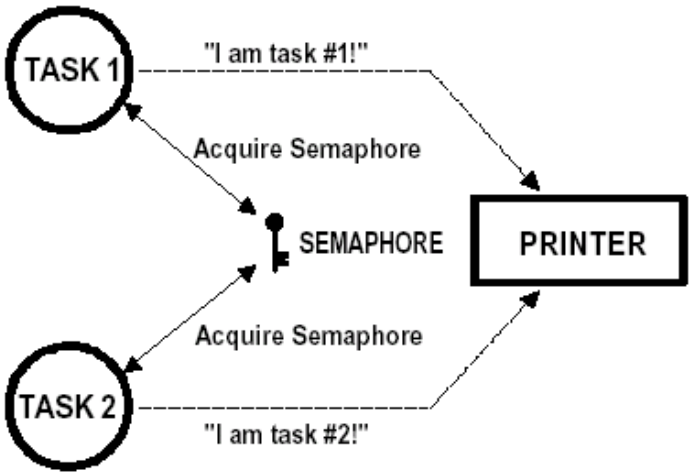


图 7－1 用获取信号量来得到打印机使用权

7.2 互斥信号量

vxWorks提供了用于任务间同步的二进制信号量操作函数：semMCreate（）、semTake（）、semGive（）、semDelete（）函数。

semMCreate（int options）：分配并初始化一个互斥信号量。其中参数 options **可以设置为 SEM_Q_PRIORITY** (0x1) 或者 **SEM_Q_FIFO** (0x0)（这两个选项为用于设置阻塞在互斥信号量的任务阻塞类型） 和 **SEM_DELETE_SAFE** (0x4) （这个选项表示删除安全）和 **SEM_INVERSION_SAFE** (0x8)（这个选项表示支持优先级翻转）的或。**当创建成功后返回一个信号量 ID 号。**

semTake（）、semGive（）、semDelete（）与用于同步的二进制信号量操作相同，这里就不再加以表述了。

这里就互斥信号量与二进制信号量异同加以说明。互斥信号量主要用于任务间对共享资源的互斥。与

二进制信号量提供的互斥功能不同，它更加严格的保护共享资源以及占用共享资源的任务。二值信号量没有优先权反转的保护；而且对于二值信号量，不同的任务都可以对其进行释放，在中断任务中也可释放二值信号量。甚至当任务持有二值信号量时，其它任务都可以删除它。但互斥信号量只能由申请该信号量的任务来释放它，决不允许在其他任务中释放它（由系统编译程序检查以实现该功能），也不允许在中断处理程序中提取与释放互斥信号量，更不允许任务锁住互斥信号量时，被其它任务删除（由参数 options 的值 SEM_DELETE_SAFE 来指定该项保护）；互斥信号量提供选择字参数 options,可以按优先权（SEM_Q_PRIORITY）与先入先出队列(SEM_Q_FIFO)两种方式排列等待对该互斥信号量进行上锁的任务，在选用优先权方式时，系统还提供优先权反转的保护。

下面用示意图说明任务使用互斥信号量的操作过程。当任务获取获取互斥信号量时，检查该信号量是否被占有，如果没有任务占有该信号量，则任务继续执行，semTake（）返回 OK，任务变为该信号量的占有者，如果发现本身已经占有了该信号量，任务继续执行，semTake（）返回 OK，count 加 1，如果发现信号量被其他任务占有，则任务阻塞直到互斥信号量被释放或者等待超时。

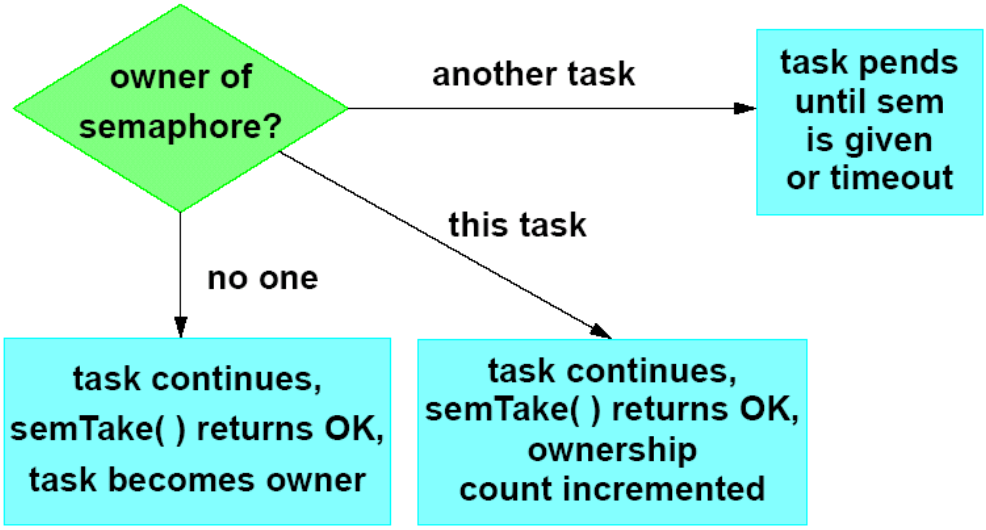


图 7－2 获取互斥信号量

当任务释放互斥信号量时，检查互斥信号量占有技术器 count，如果没有任务占有，semGive（）返回 ERROR，如果 count 大于 1，则 count 减 1，如果 count 等于 1，则判断是否有任务阻塞在该信号量上，如果有任务阻塞在该信号量上，处于阻塞队列首部的任务变为就绪态成为互斥信号量的占有者，如果没有任务阻塞在该信号量上，互斥信号量变为有效。

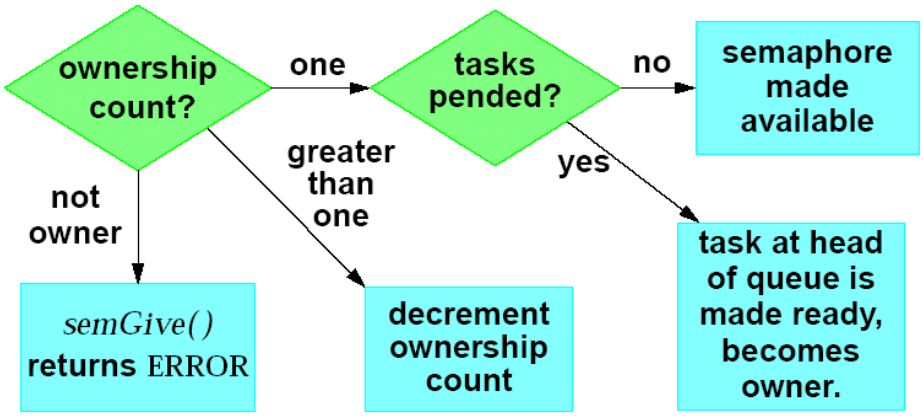


图 7－3 释放互斥信号量

在创建互斥信号量时，选择SEM_INVERSION_SAFE与SEM_Q_PRIORITY两选项的或，即可避免优先级反转。但此时系统在检查临界区的信号量上锁（semTake（））与解锁(semGive( ))时，比二值信号量实现互斥会有很大的开销，它花费了系统的开销来保护系统的稳定性。它的实现框架如下。

```
SEM_ID muxSem /*将其说明为全局变量，使所有使用它的任务都能访问到它*/
muxSem = semMCreate ( SEM_Q_PRORITY | SEM_INVERSION_SAFE );
```

taskSpawn ("tTaskA", TaskA_TASK_PRI, 0,TASK_STACK_SIZE, (FUNCPTR) taskA, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

taskSpawn ("tTaskB", TaskB_TASK_PRI, 0,TASK_STACK_SIZE, (FUNCPTR) taskB, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

```
taskA( )
{
semTake ( muxSem ,WAIT_FOREVER);
访问临界区的代码
semGive(muxSem); /*临界区上锁与解锁必须成对出现在一个任务中*/
非临界区代码
}
taskB( )
{
semTake ( muxSem ,WAIT_FOREVER);
访问临界区的代码
semGive(muxSem); /*临界区上锁与解锁必须成对出现在一个任务中*/
非临界区代码
}
```

7.3 互斥信号量用于任务间互斥操作程序示例

下面给出一个具体的程序，实现生产者任务和消费者任务通过访问由互斥信号量加以保护的共享内存来实现通信的一个例子。

程序举例：

```
#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*/
    };

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.********** */
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 demonstrationsimple */
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. ****************/
    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);
    /* relingiush the CPU so thatconsumerTask 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 filled to CONSUMED so that producerTask can put the next produced message in the
    global shared data structure. ******************************/
    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); /* relingiush 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****************** */
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******* */
LOCAL STATUS releaseProtectedSharedResource ()
{
```

```
if (semGive (mutexSemId) == ERROR)
{
perror ("protectSharedResource: Error in semTake");
return (ERROR);
}
else
return (OK);
}
```

按第三章讲述的使用tornado软件开发嵌入式软件的步骤2－3，将上述程序经编译后下载到目标机中，在windsh下启动mutexSemDemo() （） 函数得到如下运行结果：

```
0x4fb56b8 (tProducerTask): ProducerTask: tid = 0x4fb56b8, producing item = 1
0x4fce678 (tConsumerTask): ConsumerTask: Consuming item = 1 from tid = 0x4fb56b8

0x4fb56b8 (tProducerTask): ProducerTask: tid = 0x4fb56b8, producing item = 2
0x4fce678 (tConsumerTask): ConsumerTask: Consuming item = 2 from tid = 0x4fb56b8

0x4fb56b8 (tProducerTask): ProducerTask: tid = 0x4fb56b8, producing item = 3
0x4fce678 (tConsumerTask): ConsumerTask: Consuming item = 3 from tid = 0x4fb56b8

0x4fb56b8 (tProducerTask): ProducerTask: tid = 0x4fb56b8, producing item = 4
0x4fce678 (tConsumerTask): ConsumerTask: Consuming item = 4 from tid = 0x4fb56b8

0x4fb56b8 (tProducerTask): ProducerTask: tid = 0x4fb56b8, producing item = 5
0x4fce678 (tConsumerTask): ConsumerTask: Consuming item = 5 from tid = 0x4fb56b8
```

7.4 优先级翻转和优先级继承问题

VxWorks 通过优先级继承协议可以克服优先权的反转问题。优先权继承协议分为基本优先权继承协议，顶层优先权继承协议两种。基本优先权继承协议规定一个任务 T 获得了互斥信号量，同时阻塞了比其本身优先权高的多个任务后，该任务 T 在临界区放弃使用原先的优先权，将从被它阻塞的多个高优先权任务中，选出一个最高优先权任务，将该任务的优先权传给在临界区执行的低优先权任务 T，这样，低优先权任务 T 在临界区执行时，它的优先权被提高了，当其执行完临界区代码后，恢复原先进入临界区之前的优先权。基本优先权协议保证了高优先权任务被低优先权任务在临界区阻塞后不会发生优先权反转。VxWorks 系统实现了基本优先权协议，但基本优先权协议无法避免潜在的死锁。

使用实时内核,优先级反转问题是实时系统中出现得最多的问题。图 7－4 解释优先级反转是如何出现的。如图，任务 1 优先级高于任务 2，任务 2 优先级高于任务 3。任务 1 和任务 2 处于挂起状态，等待某一事件的发生，任务 3 正在运行如[图 7－4(1)]。此时，任务 3 要使用其共享资源。使用共享资源之前，首先必须得到该资源的信号量(Semaphore)。任务 3 得到了该信号量，并开始使用该共享资源[图 7－4（2）]。由于任务 1 优先级高,它等待的事件到来之后剥夺了任务 3 的 CPU 使用权[图 7－4(3)],任务 1 开始运行[图 7－4(4)]。运行过程中任务 1 也要使用那个任务 3 正在使用着的资源，由于该资源的信号量还被任务 3 占用着,任务 1 只能进入挂起状态,等待任务 3 释放该信号量[图 7－4(5)]。任务 3 得以继续运行[图 2.7(6)]。由于任务 2 的优先级高于任务 3，当任务 2 等待的事件发生后，任务 2 剥夺了任务 3 的 CPU 的使用权[图 7－4(7)]并开始运行。处理它该处理的事件[图 7－4(8)]，直到处理完之后将 CPU 控制权还给任 3[图 7－4(9)]。任务 3 接着运行[图 7－4(10)]，直到释放那个共享资源的信号量[图 7－4(11)]。直到此时,由于实时内核知道有个高优先级的任务在等待这个信号量,内核做任务切换,使任务 1 得到该信号量并接着运行[图

7－4(12)]。

在这种情况下,任务 1 优先级实际上降到了任务 3 的优先级水平。因为任务 1 要等,直等到任务 3 释放占有的那个共享资源。由于任务 2 剥夺任务 3 的 CPU 使用权,使任务 1 的状况更加恶化,任务 2 使任务 1 增加了额外的延迟时间。任务 1 和任务 2 的优先级发生了反转。

纠正的方法可以是,在任务 3 使用共享资源时,提升任务 3 的优先级。任务完成时予以恢复。任务 3 的优先级必须升至最高,高于允许使用该资源的任何任务。多任务内核应允许动态改变任务的优先级以避免发生优先级反转现象。然而改变任务的优先级是很花时间的。如果任务 3 并没有先被任务 1 剥夺 CPU 使用权,又被任务 2 抢走了 CPU 使用权,花很多时间在共享资源使用前提升任务 3 的优先级,然后又在资源使用后花时间恢复任务 3 的优先级,则无形中浪费了很多 CPU 时间。真正需要的是,为防止发生优先级反转,内核能自动变换任务的优先级,这叫做优先级继承(Priority inheritance)vxWorks 支持优先级继承。

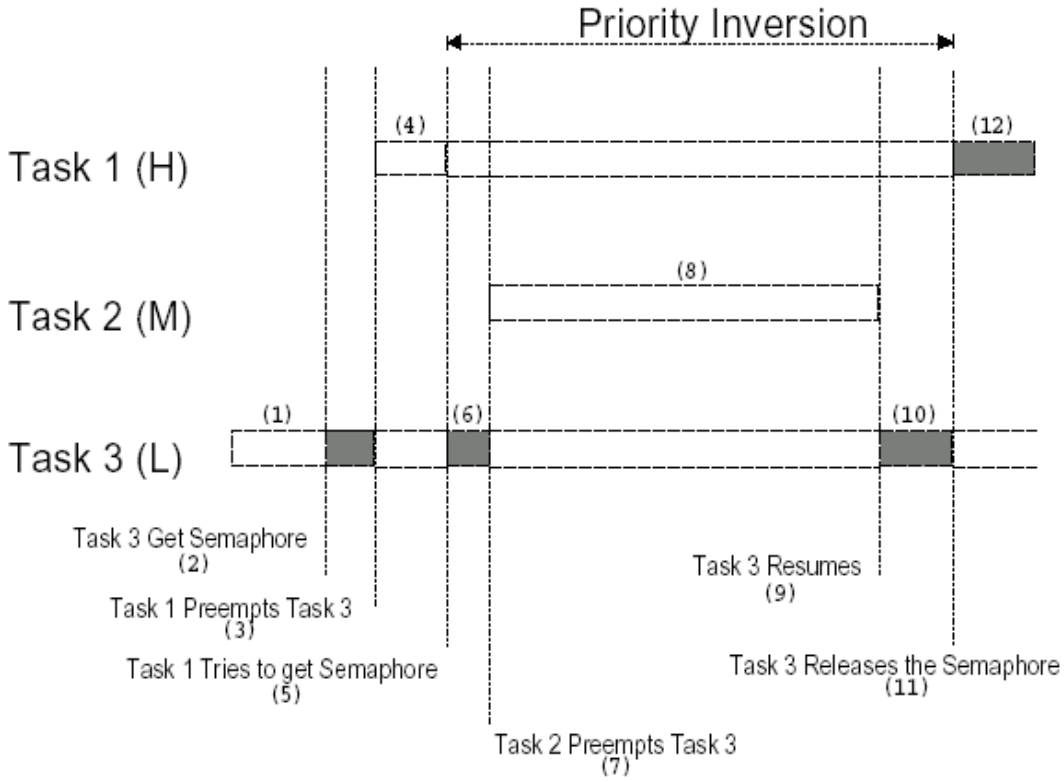


图 7－4 优先级反转问题

图 7－5 解释如果内核支持优先级继承的话,在上述例子中会是怎样一个过程。任务 3 在运行[图 7－5(1)],任务 3 申请信号量以获得共享资源使用权[图 7－5(2)],任务 3 得到并开始使用共享资源[图 7－5(3)]。后来 CPU 使用权被任务 1 剥夺[图 7－5(4)],任务 1 开始运行[图 7－5(5)],任务 1 申请共享资源信号量[图 7－5(6)]。此时,内核知道该信号量被任务 3 占用了,而任务 3 的优先级比任务 1 低,内核于是将任务 3 的优先级升至与任务 1 一样,,然而回到任务 3 继续运行,使用该共享资源[图 7－5(7)],直到任务 3 释放共享资源信号量[图 7－5(8)]。这时,内核恢复任务 3 本来的优先级并把信号量交给任务 1,任务 1 得以顺利运行。 [图 7－5(9)],任务 1 完成以后[图 7－5(10)]那些任务优先级在任务 1 与任务 3 之间的任务例如任务 2 才能得到 CPU 使用权,并开始运行 [图 7－5(11)]。注意,任务 2 在从[图 7－5(3)]到[图 7－5(10)]的任何一刻都有可能进入就绪态,并不影响任务 1、任务 3 的完成过程。在某种程度上,任务 2 和任务 3 之间也还是有不可避免的优先级反转。

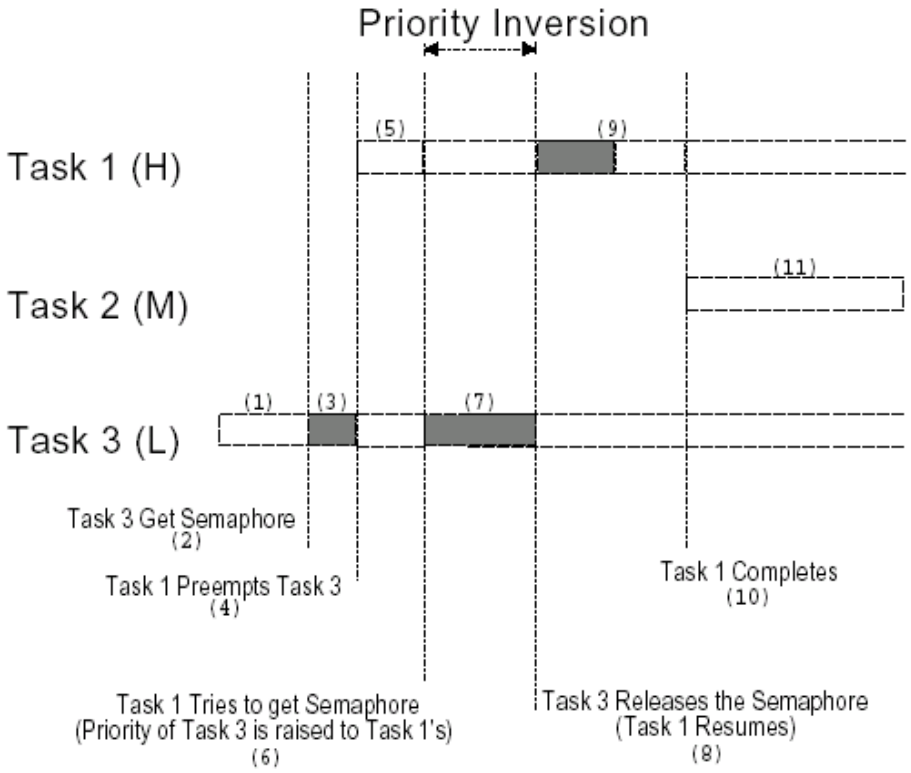


图 7－5 优先级继承

7.5 优先级翻转和继承示例源程序

下面这个程序演示了优先级翻转现象，如将程序中 semMutex=semMCreate(SEM_Q_PRIORITY)语句改为 semMutex=semMCreate(SEM_Q_PRIORITY| **SEM_INVERSION_SAFE**)则为优先级继承源程序。

```
/* function defined */
void taskLow(void);
void taskMiddle(void);
void taskHigh(void);

/* global varibles*/
#define ITER    1
#define HIGH 101
#define MIDDLE 102
#define LOW 103
#define LONG_TIME 3000000
SEM_ID semMutex;

void Inversion(void)/*function to create three tasks*/
{
    int i,taskIdLow,taskIdMiddle,taskIdHigh;
    printf("\n\n.............#RUNNING#.............\n\n");

    /* CREATE a mutex semaphore*/
```

```
        semMutex=semMCreate(SEM_Q_PRIORITY );

        /* Create three tasks ,each has a unique priority*/
        if(                                                                    taskIdLow=taskSpawn("task1",
LOW,0x100,20000,(FUNCPTR)taskLow,0,0,0,0,0,0,0,0,0,0)==ERROR)
            printf("\nTaskSpawn taskone failed\n");
        if(                                                                taskIdMiddle=taskSpawn("task2",
MIDDLE,0x100,20000,(FUNCPTR)taskMiddle,0,0,0,0,0,0,0,0,0,0)==ERROR)
            printf("\nTaskSpawn tasktwo failed\n");
        if(                                                                  taskIdHigh=taskSpawn("task3",
HIGH,0x100,20000,(FUNCPTR)taskHigh,0,0,0,0,0,0,0,0,0,0)==ERROR)
            printf("\nTaskSpawn taskthree failed\n");
    }

    void taskLow(void)
    {
        int i,j;
        for(i=0;i<ITER;i++)
        {
            semTake(semMutex,WAIT_FOREVER);/* WAIT indefinitely for the semaphore*/
            logMsg("taskLow locks the semaphore\n",0,0,0,0,0,0);

            for(j=0;j<10*LONG_TIME;j++);

            /* taskLow gives up the semaphore*/
            logMsg("tasklow unlocks the semaphore\n",0,0,0,0,0,0);
            semGive(semMutex);

        }
        logMsg("taskLow end!\n",0,0,0,0,0,0);
    }

    void taskMiddle(void)
    {
        int i;
        taskDelay(2);/* make taskLow runs first*/

        for(i=0;i<LONG_TIME*10;i++)
        {
            if ( (i%LONG_TIME)==0 )
                    logMsg("TaskMiddle is running\n",0,0,0,0,0,0);

        }
        logMsg("taskMiddle end!\n",0,0,0,0,0,0);
```

```
}

void taskHigh(void)
{
    int i,j;
    taskDelay(3);

    for(i=0;i<ITER;i++)
    {
        logMsg("taskHigh wants to get the semaphore\n",0,0,0,0,0,0);
        semTake(semMutex,WAIT_FOREVER);

        logMsg("taskHigh    LOCK the semaphore\n",0,0,0,0,0,0);
        for(j=0;j<LONG_TIME;j++);/*Allow time to context switch*/
        semGive(semMutex);
    }
    logMsg("taskHigh end!\n",0,0,0,0,0,0);
}
```

按第三章讲述的使用tornado软件开发嵌入式软件的步骤2－3，将上述程序经编译后下载到目标机中，在windsh下启动Inversion（）函数得到如下运行结果：

```
.............#RUNNING#.............

0x4fb56b8 (task1): taskLow locks the semaphore
0x4fae5f0 (task2): TaskMiddle is running
0x4fa7528 (task3): taskHigh wants to get the semaphore
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): taskMiddle end!
0x4fb56b8 (task1): tasklow unlocks the semaphore
0x4fa7528 (task3): taskHigh    LOCK the semaphore
0x4fa7528 (task3): taskHigh end!
0x4fb56b8 (task1): taskLow end!
```

将上述源程序中 semMutex=semMCreate(SEM_Q_PRIORITY)语句修改为 semMutex=semMCreate(SEM_Q_PRIORITY| **SEM_INVERSION_SAFE**),修改后的程序支持优先级继承，重新编译下载执行后得到的结果为：

```
.............#RUNNING#.............
```

0x4fb56b8 (task1): taskLow locks the semaphore
0x4fae5f0 (task2): TaskMiddle is running
0x4fa7528 (task3): taskHigh wants to get the semaphore
0x4fb56b8 (task1): tasklow unlocks the semaphore
0x4fa7528 (task3): taskHigh LOCK the semaphore
0x4fa7528 (task3): taskHigh end!
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): TaskMiddle is running
0x4fae5f0 (task2): taskMiddle end!
0x4fb56b8 (task1): taskLow end!

8 中断管理

8.1 中断简介

嵌入式实时系统（Real-Time System）是一个能够在指定或者确定的时间内对外部事件作出响应的系统，其重要的特性是实时响应性。嵌入式实时系统对外部事件的响应一般都是通过中断来处理的，其对中断的处理方式，直接影响到系统的实时性能。

实时多任务操作系统是嵌入式应用开发的基础平台。早期的嵌入式实时应用软件直接在处理器上运行，没有RTOS支持，现在的大多嵌入式应用开发都需要嵌入式操作系统的支持。实际上，此时的嵌入式操作系统相当于一个通用而复杂的主控程序，为嵌入式应用软件提供更强大的开发平台和运行环境。因为嵌入式系统已经将处理器、中断、定时器、I/O等资源包装起来，用一系列的API提供给用户，应用程序可以不关注底层硬件，直接借用操作系统提供的功能进行开发，此时的嵌入式操作系统可以视为一个虚拟机。

随着嵌入式实时系统的发展，为了方便对中断的处理，系统内核常接管中断的处理，比如提供一些系统调用接口来安装用户的中断，提供统一的中断处理接口等。根据系统内核的可抢占或者非抢占性，系统内核接管中断又有两种不同处理模式。

在非抢占式内核的中断处理模式中，当在中断处理过程中有高优先级任务就绪时，不会立即切换到高优先级的任务，必须等待中断处理完后返回到被中断的任务中，等待被中断的任务执行完后，再切换到高优先级任务。在抢占式内核的中断处理模式中，如果有高优先级任务就绪时，则立刻切换到高优先级的任务。

8.2 MPC860 中断控制器结构

首先要理解异常与中断在概念上的区别。异常是指与指令的执行出现错误而引起的系统中断操作，如总线错误，除0 错误等。而中断是指外部事件对正在执行的程序中断，它与当前正在执行的指令无关。在MPC860中所有的外部中断作为一个特殊的异常来处理。

图2 是MPC860 的中断控制器的框图。从图中可以看出MPC860 有两个中断控制器，SIU中断控制器和CPM中断控制器。

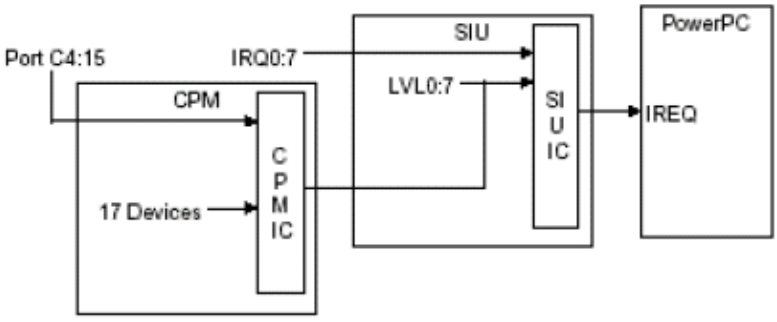


图8－1 MPC860中断控制器硬件结构

SIU中断控制器

SIU 接收从8 个外部管脚（IRQ0 到IRQ7）和8 个内部源（Level0 到Level7）总共16个能的中断源的中断输入，SIU有15 个中断源只可以设置一个向PowerPC的中断请求，SIU设置IREQ输入到PowerPC内核。为每个设备分配一个中断优先级，这样，当中断产生时，可以立即服务中断，用户可以分配多个设备为一个级别，但是，如果一个中断产生，SIU中断控制器必须查询设备以决定哪一个内部设备产生了中断。

CPM 中断控制器

CPM 有29 个中断源，CPM驱动SIU中断控制器的一个中断电平(LVL0～LVL7中的其中一个)，然后再驱动PowerPC 核心的IREQ。

它的中断源包括所有可以提供中断的内部设备和端口C 的12 个管脚，这些设备是4 个SCC、2 个SMC、SPI、I2C、PIP 和通用定时器，CPM 中断控制器CPIP允许屏蔽每个中断源，如果这些中断其中一个产生，CPIC 完成处理，然后以一个特定电平设置SIU 中断控制器。如果通过SIU 中断控制器完成中断处理，它设置IREQ 到PowerPC 核心，如果中断允许，程序控制转到相应的中断服务程序。

8.3 VxWorks 中断相关处理函数

VxWorks是美国WindRiver公司推出的一个运行在目标机上的高性能、可裁减的嵌入式实时操作系统。它为程序员提供了高效的实时多任务调度、中断管理、实时的系统资源、实时任务的通信同步机制、设备管理、文件系统管理以及丰富的网络协议功能。在各种CPU 平台上提供了统一的编程接口和一致的运行特性，使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和熟悉各种嵌入式微处理复杂的汇编指令。中断处理在实时系统中至关重要，外部事件总是通过中断告知系统它的发生并要求系统作快速的处理。为实现对中断尽可能快的响应，VxWorks 的中断服务历程运行在任何任务之外的上下文。中断的发生并不引起任务的切换。VxWorks 提供的中断处理函数主要有以下这些：

***intConnect*( )** 连接C函数与中断向量
***intContext*( )** 如果从中断级调用则返回真
***intCount*( )** 获取当前中断的嵌套深度
***intLevelSet*( )** 设置中断屏蔽级
***intLock*( )** 中断上锁
***intUnlock*( )** 中断解锁
***intVecBaseSet*( )** 设置中断向量基地址
***intVecBaseGet*( )** 获取中断向量基地址
***intVecSet*( )** 设置异常向量 ； MPC860不支持
intVecGet( ) 获取异常向量

在本次设计中主要会用到*intConnect*( )，所以需要同学们重点对这个函数进行理解和掌握。*intConnect*( )的原型为intConnect (VOIDFUNCPTR * vector, VOIDFUNCPTR routine, int parameter)，VOIDFUNCPTR * vector是指要联系的中断向量，VOIDFUNCPTR routine是指中断发生后要执行的函数， int parameter是指传递给中断处理函数的参数。

8.4 中断控制器初始化程序示例

8.4.1 设计 ppc860IntrInit（）例程

该例程被被 BSP 中的 sysHwInit()调用，实现中断控制器的初始化。ppc860IntrInit 完成的功能是将所有的中断与 0x500 系统的异常向量关联起来。它应包含以下语句：

defaultVec = (VOIDFUNCPTR) excVecGet ((FUNCPTR *) _EXC_OFF_INTR);

excIntConnect ((VOIDFUNCPTR) _EXC_OFF_INTR, ppc860IntrDeMux); / *安装异常向量为 0x500 的异常处理程序 ppc860IntrDeMux，将该函数的入口地址放到异常向量表 */

_func_intConnectRtn = ppc860IntConnect; /*定义执行 intConnect（）系统调用时执行的用户函数。

_func_intEnableRtn = ppc860IntEnable; /*定义执行 intEnable（）系统调用时执行的用户函数 */

_func_intDisableRtn = ppc860IntDisable; /*定义执行 intDisable（）系统调用时执行的用户函数 */

/*以上定义系统调用的钩子函数为用户定义的函数*/

for (vector = 0; vector < NUM_VEC_MAX; vector++)

intConnect (INUM_TO_IVEC(vector), defaultVec, 0); /*将所有的中断与外部异常向量关联起来。

8.4.2 设计 ppc860IntConnect（）函数

对于 MPC860 来说，中断向量表就是在内存中定义一个数组，来保存中断服务程序入口地址。数组定义如下：LOCAL INTR_HANDLER intrVecTable[NUM_VEC_MAX];

ppc860IntConnect 函数完成的功能是将 C 函数的地址和要传递的参数填充到中断向量表中，主要包含以下两个语句：

intrVecTable[IVEC_TO_INUM(vector)].vec = routine;

intrVecTable[IVEC_TO_INUM(vector)].arg = parameter;

8.4.3 设计 ppc860IntrDeMux（）函数

该函数的功能是：完成 SIU 中断控制器的中断去复用功能。也就是说当中断发生后，IREQ 对应于偏移量为 0x500 的异常处理。应包含以下语句：

regBase = vxImmrGet(); /*读中断向量寄存器*/

intVec = (* SIVEC(regBase)) >> 26; /*得到中断向量号*/

intMask = * SIMASK(regBase); /* 保存当前中断屏蔽寄存器 */

SIMASK(regBase) &= (0xffffffff << (32 - intVec));

/*给所有低于检测到的中断的优先级的中断上锁 */

intUnlock (_PPC_MSR_EE); /* 解锁中断 允许内核响应中断*/

intrVecTable[intVec].vec (intrVecTable[intVec].arg); /* 执行对应的中断处理程序*/

* SIMASK(regBase) = intMask; /* 恢复中断屏蔽 */

8.4.4 中断的允许与禁止

设计函数 ppc860IntEnable（intnum），使能进入 SIU 的 Level 或者 IRQ 外部中断请求管脚。应主要包含以下语句：

regBase = vxImmrGet();

* SIMASK(regBase) |= (1 << (31 - intNum)); /*允许中断号 intNum 对应的中断管脚提出中断申请 */

设计函数 ppc860IntDisable（intnum），该函数刚好与 ppc860IntEnable（intnum）相反，它禁止 Level 或者 IRQ 管脚向 SIU 发中断请求。应主要包括以下语句

regBase = vxImmrGet();

* SIMASK(regBase) &= ~(1 << (31 - intNum));

8.5 ISR 程序示例

作为一种实时机制，中断处理程序不能调用引起自己阻塞的函数，如调用 I/O 操作，以及获取信号量，或者进行浮点运算等。中断服务程序应尽可能地短，最简单的中断处理程序仅仅包含一个发送或释放信号

量的操作，把对中断的处理放到任务中去执行。中断也能与任务进行通信与同步操作，但中断应是发起者，而不是接受者，否则将会引起中断阻塞。有时出于对共享资源进行独占访问的需要，应采取关中断的方式实现任务与中断互斥访问，注意采取这种方式时要特别小心，应使关中断的时间尽可能短，以免影响整个嵌入式系统地实时性能。中断是引起任务重调度的原因之一，如低优先级的任务正在执行，此时中断到来，则该任务变为阻塞态，系统响应中断，如果中断处理程序使另一个较高优先级的任务就绪，则系统响应完中断后，内核发生调度，执行这个就绪的高优先级任务。因此在使用中断设计时应小心，避免发生优先级翻转。

下面给出一个简单的例子说明应用程序开发人员如何设计中断处理程序。在该应用中 S2 可为具体的某种特定应用的中断源，比如数据采集满了发中断。本例完成的功能是按下 S2 键产生一个中断，系统执行 C 语言的中断处理程序 TestIsr，打印“Interrupt is ok”信息。

```
void TestIsr(void)
{
*SIPEND(vxImmrGet()) |=0x00200000;
logMsg("Interrupt is OK \n",0,0,0,0,0,0);
}
void vx_main(void)
{
   logMsg("This is ISR test program\n",0,0,0,0,0,0);
   SIEL(vxImmrGet()) |=0x00200000;          /*设置中断触发方式为电平触发*/
   intConnect(IV_IRQ5, (VOIDFUNCPTR)TestIsr, 0);/*连接中断 IRQ5 和中断处理程序 TestIsr */
   intEnable(IV_IRQ5);     /*允许 IRQ5 中断请求  */
}
```

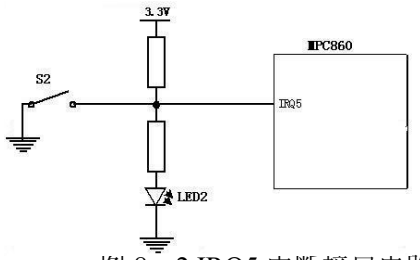


图 8－2 IRQ5 中断接口电路

9 定时器管理

9.1 定时器介绍

在嵌入式系统中，时钟管理是必不可少的。时钟管理主要提供以下功能：维护系统日历时钟；在任务等待消息、信号量或内存段时的超时处理；以一定的时间间隔或在特定的时间唤醒或发送告警到一个任务。处理任务调度中的时间片轮循。这些功能都依赖于周期性的定时中断，离开实时时钟或定时器硬件就无法工作。vxWorks操作系统提供了看门狗定时器和系统时钟，辅助时钟以及时戳时钟。由于时戳时钟在实现上和辅助时钟并没有不同，在本章里主要讨论看门狗定时器、系统时钟和辅助时钟。

9.2 看门狗定时器

VxWorks看门狗定时器作为系统时钟中断服务程序的一部分，允许C语言函数指定某一时间延迟。一般来说，被看门狗定时器激活的函数运行在系统时钟中断级。然而，如果内核不能立即运行该函数，函数被放入tExcTask工作队列中。在tExcTask工作队列中的任务运行在最高优先级0。

看门狗定时器调用函数：

wdCreate()	分配并初始化看门狗定时器
wdDelete()	中止并解除看门狗定时器
wdStart()	启动看门狗定时器
wdCancel()	取消当前正在计数的看门狗定时器

创建看门狗定时器的函数原型为WDOG_ID wdCreate (void)，当创建成功后后返回一个看门狗定时器ID号，删除看门狗定时器的函数原型是STATUS wdDelete(WDOG_ID wdId)删除由wdId号所标识的看门狗定时器。启动看门狗定时器的函数原型是STATUS wdStart(WDOG_ID wdId,int delay,FUNCPTR pRoutine,int parameter)，参数wdId是要启动的看门狗定时器的ID号，delay表示定时时间间隔，单位是ticks，pRoutine是指定时时间到后执行的定时器服务程序。Parameter是指定时器服务程序的入口参数。

9.3 看门狗定时器用于死限任务处理程序示例

实时系统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例程（deadline handlers）。看门狗可以用来启动这种时限事故处理例程。下面给出一个具体的例子。

协调任务向组织任务发送数据。组织任务接受来自于协调任务的数据，如果 5s 内（时限）没有数据发送，协调任务将复位。20s 后演示程序自动停止。

```
/* 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. */
    }
```

按第三章讲述的使用tornado软件开发嵌入式软件的步骤2－3，将上述程序经编译后下载到目标机中，在windsh下启动deadlineWdDemo ()函数得到如下运行结果：

```
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

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 = 0x3d0002
```

9.4 系统时钟

VxWorks需要一个专用的定时器用作系统时钟。VxWorks提供的系统时钟的驱动程序代码位于**../src/drv/timer/xxTimer.c**。系统时钟的初始化和启动是在tUserRoot任务中完成的。系统时钟由具有最高

优先级的中断服务程序usrClock（）提供。BSP开发人员必须提供以下例程用于支持系统时钟。

sysClkConnect() – 安装系统时钟

sysClkRateSet() – 设置系统时钟中断频率

sysClkRateGet() – 得到系统时钟频率

sysClkEnable() – 激活系统时钟.

sysClkDisable() – 禁止系统时钟

在本课程设计中，使用的是基于MPC860的嵌入式系统，系统定时器使用DEC异常来实现。在usrRoot中调用以下语句来实现系统定时器的初始化。

```
sysClkConnect ((FUNCPTR) usrClock, 0);  /* connect clock ISR */
sysClkRateSet (60);                  /* set system clock rate */
sysClkEnable ();               /* start it */
```

其中usrClock设计如下：

```
void usrClock ()
{  tickAnnounce ();    /* 向内核通知系统tick */
}
```

其他函数设计如下：

```
sysClkConnect（FUNCPTR    routine,int  arg）
{...
sysHwInit2(); 进一步对硬件进行初始化
excIntConnect ((VOIDFUNCPTR *) _EXC_OFF_DECR, (VOIDFUNCPTR) sysClkInt);连接异常处理程序
sysClkRoutine    = routine;
sysClkArg        = arg;
}
sysClkInt ()
{.....
if (sysClkRunning && (sysClkRoutine != NULL))
(*(FUNCPTR) sysClkRoutine) (sysClkArg);执行时钟中断服务例程
.....
}
void sysClkEnable (void)
{
sysClkRunning = TRUE;打开中断
vxDecSet (decCountVal);重新装入减法计数初值
}
sysClkDisable()
{
.....
sysClkRunning = FALSE;关闭时钟中断服务例程
.....
}
sysClkRateSet（int   ticksPerSecond  ）
{
sysClkTicksPerSecond = ticksPerSecond;
decCountVal = sysDecClkFrequency / ticksPerSecond; /*改变计数初值*/
```

```
}
```

9.5 辅助时钟

在使用 VxWorks 开发的嵌入式系统有时也需要辅助时钟。辅助时钟用于高速或低速轮询，Tornado 的 spy（）例程也需要。BSP 可以支持一个或多个辅助时钟，并系统时钟和辅助时钟不能共用一个硬件定时器。辅助定时器使用的 ppc860 的四个通用定时器中的 Timer2，860 的定时器具体结构请参见参考文献。

sysAuxClkConnect (fooPoll, param);
sysAuxClkRateSet (rate);
sysAuxClkEnable ();
其中　void fooPoll (int param)设计如下：

```
  {
  status = pDeviceAdr;    /* 轮询IO设备的状态*/
  ...
  }
```

其他函数设计如下：

```
sysAuxClkConnect (
FUNCPTR routine, /* routine called at each aux. clock interrupt */
int     arg      /* argument to auxiliary clock interrupt routine */
)
sysAuxClkRoutine   = routine;
sysAuxClkArg       = arg;
return (OK);
}
sysAuxClkEnable()
{
 tempDiv = SYS_CPU_FREQ / (sysAuxClkTicksPerSecond << 8);
*TRR2(CPM_MEM_BASE) = (UINT16) tempDiv;  /计数器计数值

(void) intConnect (IV_TIMER2, (VOIDFUNCPTR) sysAuxClkInt, NULL);
/* 连接硬件定时器与辅助时钟中断处理程序*/
*TGCR(CPM_MEM_BASE) |= TGCR_RST2; /* enable timer2 */
}
void sysAuxClkDisable (void)
{...
*CIMR(CPM_MEM_BASE) &= ~CISR_TIMER2;  /* disable interrupt */
*TGCR(CPM_MEM_BASE) |= TGCR_STP2; /* stop timer */
...}
void sysAuxClkInt (void)
{*TER2(CPM_MEM_BASE) |= TER_REF;      /* clear event register */
*CISR(CPM_MEM_BASE) = CISR_TIMER2;    /* clear in-service bit */
(*sysAuxClkRoutine) (sysAuxClkArg);  /* 调用安装的中断处理程序
}
int sysAuxClkRateGet (void)
{
```

```
return (sysAuxClkTicksPerSecond);
}
int sysAuxClkRateSet(int ticksPerSecond)
{
sysAuxClkTicksPerSecond = ticksPerSecond;
sysAuxClkDisable ();
sysAuxClkEnable ();
}
```

10　I/O 驱动设计

10.1 IO 驱动设计简介

为了提供应用程序的可移植性将应用程序从直接的操作设备的繁琐细节中解放出来操作系统应该为应用程序操作设备提供一个一致的接口。这个一致的接口就是由操作系统的 I/O 系统提供的。I/O 系统将应用程序的请求传递给设备专用的 I/O 函数. 这些设备专用的 I/O 函数就是由设备驱动程序提供的。为了对设备进行有效的管理操作系统通常还要具有一些内部的数据结构。对于 VxWorks 而言这些内部结构包括文件表、设备列表和驱动程序表。

驱动程序表中包含每个设备专用的 I/O 调用函数, 驱动程序表是 I/O 系统访问 driver 的入口点，是 I/O 系统中十分重要的数据结构。它的大小通常是固定的，包括相同的例程 creat( )和 open( )。它将 vxworks 提供的标准例程（如 open）和用户编写的设备驱动程序（myOpen）关联起来。可以使用 iosDrvShow( )来显示 Driver 表。

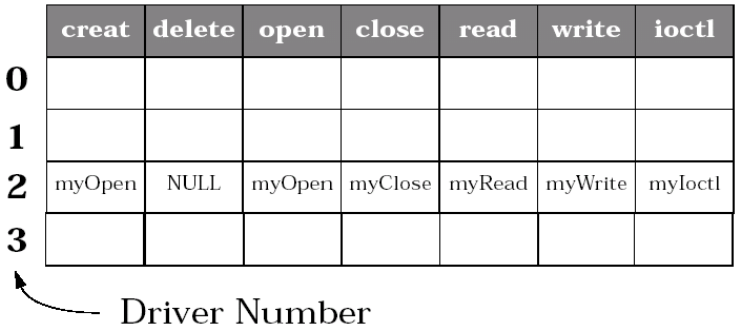


图10.1 驱动程序表

设备列表是由设备描述符组成的链表。每个设备有一个相应的设备描述符。描述符是由与驱动程序无关的部分和与驱动程序相关的部分组成。与驱动程序无关的部分是称为设备头的数据结构。它包含指向前后设备描述符(DEV_HDR)的指针设备名以及设备专用函数在系统驱动程序表中的, I/O索引等。与驱动程序有关的部分包含驱动程序专用的数据。I/O系统中open( )、creat( )和delete( )三个例程使用设备链表去匹配一个设备名。该链表是双向量标结构，在链表上的项目数是动态的。

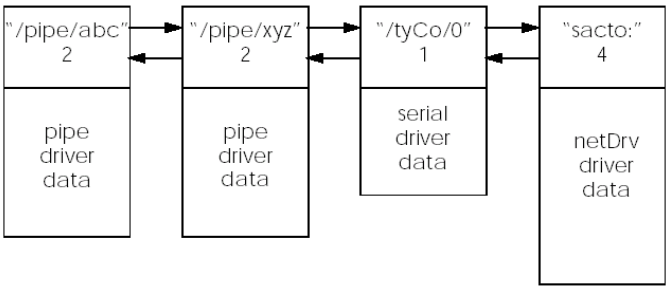


图 10.2 设备链表

文件表用于管理文件和设备是由文件描述符组成。对于每一个打开的文件或设备都有一个文件描述符。文件描述符是由设备专用函数在系统驱动程序表的索引和设备相I/O关的数据组成。应用程序调用或后系统返Open() Create() ,I/O回此设备的文件描述符索引fd。此后应用程序就可以使用fd对设备进行操作。I/O系统在设备或文件打开后使用文件描述表去表示一个驱动器。由open( )或creat()成功得到的文件描述符是该表的索引。它的大小是固定的并且可以通过configAll.h中的NUM_FILES参数配置。

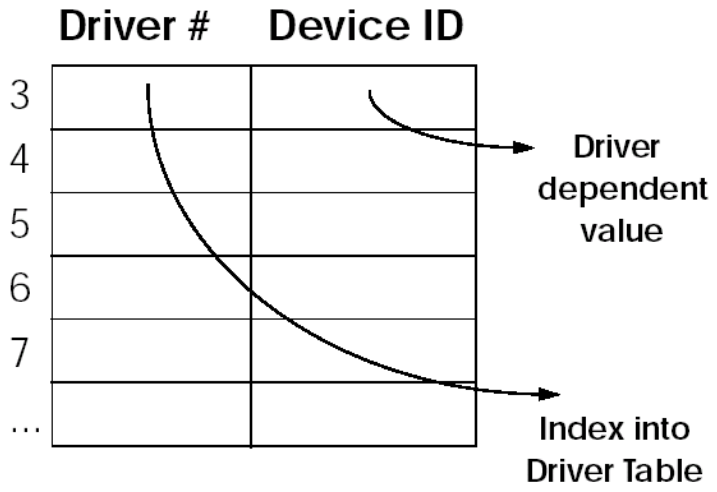


图 10.3 文件描述符表

10.2 字符型 IO 设备访问过程

VxWorks 为应用程序访问 IO 设备提供了统一的接口，对于应用程序开发人员来说只要使用 7 个标准的 IO 函数 create、delete、open、close、read、write、ioctl 等，不必熟悉底层的硬件具体实现细节，就可以实现对 IO 设备的操作，而对于 BSP 的程序开发人员来说，必须设计具体的 xxCreate、xxDelete、xxOpen、xxClose、xxRead、xxWrite、xxioCtl 来完成对 IO 设备的具体操作，并且实现对应的关系。因而 IO 设计也分为两个部分，第一个部分就是要实现设备驱动程序及设备的添加，这是在 BSP 阶段完成的。第二个部分就是在应用程序里打开 IO 设备，并对其进行读写操作。

在 usrRoot 函数中完成设备驱动程序的安装和 IO 设备的添加。在这一部分，用户要设计具体的函数来实现对设备的操作。主要包括以下语句：

（1）设计 xxDrv（）函数，它调用 iosDrvInstall（）来安装设备驱动程序，将用户设计的对 IO 具体操作的函数填写到设备驱动程序表中，并返回驱动程序号。如图 10.4 中（1）所示。

（2）设计 xxDevCreate（），它调用 isoDevAdd（）来添加设备到设备链表中。设备链表中的每个元素由三个部分组成，双向链表指针、设备名及设备的驱动程序号，设备依赖数据结构（与具体的设备有关）。如图 10.4 中（2）所示。

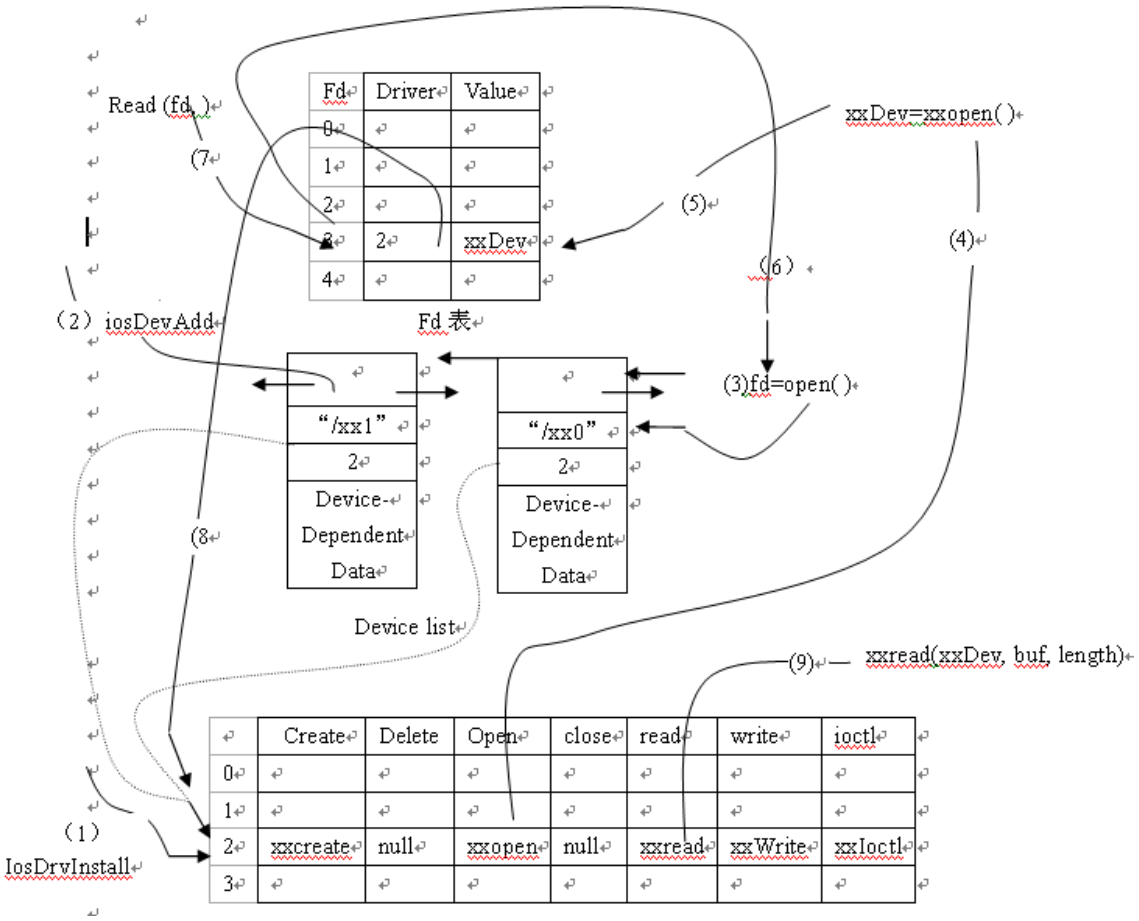


图 10.4 字符型 IO 设备访问过程

在图 10.4 中虚线框（5）中的任务要访问 IO 设备，其操作步骤如下：

（1）、如图 10.4 中（3）～（6）步，执行 open（）函数到设备链表中去查找匹配设备名，找到对应的 driver 号，执行 xxopen 函数，返回设备的 ID 号（device ID），然后将 driver 和 deviceID 号一并填写到 fd（文件描述符表）表中去，open 函数返回 Fd 号，以便 read，write、ioctl 函数操作使用。

（2）、如图 10.4 中（7）～（9）执行 read 函数，实现从刚才打开的设备中读取数据存储到缓冲区中。具体实现的过程如下：根据 fd 号，查找 fd 表（文件描述符表），得到相应的 driver 号，从而查找 driver 表可以得到 read 函数所对应的 xxread 函数，xxread 函数对 fd 表中 devID 所指定的设备进行操作。Write 函数的实现与 read 函数相似，这里就不再加以描述。

（3）、以上操作完成之后，使用 close 函数关闭刚才打开的设备。至此应用程序访问 IO 完成。

10.3 字符型 IO 设备设备驱动程序的编写

(1) xxDrv()

xxDrv()函数首先初始化设备驱动程序，然后调用 iosLib 库中的例程 iosDrvInstall()。它在设备驱动程序表中为设备分配一个条目并在其中填入此设备的 xxCreate(),xxDelete() 和 xxOpen(),xxClose(),xxRead(),xxWrite() xxIoctl()等函数的地址。然后返回此条目的索引见表即驱动程序号。对于不实现的功能在的 iosDrvInstall()参数表中相应的位置用 NULL 代替。

(2) xxDevCreate()

xxDevCreate()函数首先初始化设备然后调用 iosLib 库中的 iosDevAdd()例程。它的参数是设备描述符的指针设备名和前面 iosDrvInstall()例程返回的驱动程序号。它将设备描述符指针指向的设备描述符添加到设备列表中然后用设备名和驱动程序号来初始化化 DEV_HDR 数据结构.

(3)编写 xxOpen 和 xxCreate（）函数

此函数的形式是 xxOpen（指向 DEV_HDR 的指针、设备名、打开标志、方式）。方式位在字符设备中一般不使用。

xxOpen 函数的返回值有两种。当信息仅由设备维护时返回的是指向设备描述符的指针。另一种是返回驱动程序相关的

标识符，典型的是一个指向驱动程序的私有数据结构的指针。xxCreate（）函数情况类似。

(4)编写函数和函数 xxRead() xxWrite()

这两个函数分别具有如下形式:

int xxWrite(设备 ID, 缓冲区指针，字节数)

int xxRead(设备 ID，缓冲区指针，字节数)

5) xxIoctl()

此函数具有如下形式:

int xxIoctl(设备 ID,命令，参数)

参数可以是一个整数值一个接收数据或发送数据的缓,

冲区地址也可不使用。设备是由打开函数返回的。

(6) xxClose()

此函数关闭设备它具有如下形式:

STATUS xxClose(设备 ID)

10.4 字符型 IO 设备驱动程序示例

以下程序实现的功能是把内存中一段区域模拟字符型 IO 设备来操作。

```
typedef struct              /* MEM_DEV - memory device descriptor */
{
DEV_HDR devHdr;
int allowOffset1;
char *MemBase;

   /* Allow files to be opened with an offset. */
} MEM_DEV;

typedef struct              /* MEM_DEV - memory device descriptor */
{
MEM_DEV *pMemDev;

int allowOffset2;   /* Allow files to be opened with an offset. */
} MEM_DEV_PER_FD;

MEM_DEV_PER_FD pMemDevPerFd;

LOCAL int memDrvNum;        /* driver number of memory driver */

/* forward declarations */

LOCAL MEM_DEV_PER_FD *memOpen ();
```

```
LOCAL int memRead ();
LOCAL int memWrite ();
LOCAL int memClose ();
LOCAL STATUS memIoctl ();


/*memDrv - install a memory driver*/
STATUS memDrv (void)
    {
    if (memDrvNum > 0)
    return (OK);    /* driver already installed */
    memDrvNum = iosDrvInstall ((FUNCPTR) memOpen, (FUNCPTR) NULL,
                    (FUNCPTR) memOpen, memClose,
                    memRead, memWrite, memIoctl);
    return (memDrvNum == ERROR ? ERROR : OK);
    }
/* memDevCreate - create a memory devicey.*/
STATUS memDevCreate
    (
    char * name,        /* device name                  */
    char * base,        /* where to start in memory */
    int    length       /* number of bytes         */
    )
    {
    STATUS status;
    FAST MEM_DEV *pMemDv;
    if (memDrvNum < 1)
        {
        errnoSet (S_ioLib_NO_DRIVER);
        return (ERROR);
        }
    if ((pMemDv = (MEM_DEV *) calloc (1, sizeof (MEM_DEV))) == NULL)
    return (ERROR);
      pMemDv->MemBase=base;
    status = iosDevAdd ((DEV_HDR *) pMemDv, name, memDrvNum);
    if (status == ERROR)
    free ((char *) pMemDv);
    return (status);
    }

/* memOpen - open a memory file*/
LOCAL MEM_DEV_PER_FD    *memOpen
    (
    MEM_DEV *pMemDv, /* pointer to memory device descriptor */
```

```
        char *name,             /* name of file to open (a number) */
        int mode        /* access mode (O_RDONLY,O_WRONLY,O_RDWR) */
        )
        {
        MEM_DEV_PER_FD    *pMemDevPerFd = NULL;
        pMemDevPerFd = (MEM_DEV_PER_FD *) calloc(1,sizeof(MEM_DEV_PER_FD));
if (pMemDevPerFd != NULL)
        {
              pMemDevPerFd->pMemDev = (MEM_DEV *) pMemDv;
              pMemDevPerFd->allowOffset2 = 0;
        return pMemDevPerFd;
        }
  else
        return (MEM_DEV_PER_FD *) ERROR;
        }
/* memRead - read from a memory file*/
LOCAL int memRead
        (
      MEM_DEV_PER_FD *pfd,    /* file descriptor of file to read */
        char *buffer,   /* buffer to receive data */
        int maxbytes   /* max bytes to read in to buffer */
        )
        {
  bcopy (pfd->pMemDev->MemBase,buffer, maxbytes);
        return (maxbytes);
        }
/** memWrite - write to a memory file*/
LOCAL int memWrite
        (
        MEM_DEV_PER_FD *pfd,  /* file descriptor of file to write */
        char *buffer,   /* buffer to be written */
        int nbytes        /* number of bytes to write from buffer */
        )
        {
        bcopy (buffer,pfd->pMemDev->MemBase, nbytes);
    return (nbytes);
        }
/*memIoctl - do device specific control function*/

LOCAL STATUS memIoctl (pfd, function, arg)
        MEM_DEV_PER_FD *pfd;  /* descriptor to control */
        FAST int function;          /* function code */
        int    arg;                 /* some argument */
        {
```

```
    return (OK);
    }
/*memClose - close a memory file*/

LOCAL STATUS memClose (pfd)
    MEM_DEV_PER_FD *pfd;/* file descriptor of file to close */
    {
    free (pfd);
    return OK;
    }
```

将上述程序编译后下载到目标中，在 windsh 下执行以下命令。

```
-> iosDrvShow
drv    create     delete       open      close       read      write      ioctl
  1    421db4          0     421db4     421ddc     42b76c     42b69c     421e08
  2         0          0     424fd4          0     425004     425044     425130
  3    426e88     426f04     427170     426e34     427228     427254     426ffc
  4    415f40          0     415f40     416000     42b76c     42b69c     416074
  5    41626c     41642c     416288     416520     4165d0     416670     416710
value = 0 = 0x0
/*未安装内存字符型 I0 设备驱动程序之前，系统中存在的设备驱动程序。*/
-> memDrv
value = 0 = 0x0
/*安装内存字符行 I0 设备驱动程序*/
-> iosDrvShow
drv    create     delete       open      close       read      write      ioctl
  1    421db4          0     421db4     421ddc     42b76c     42b69c     421e08
  2         0          0     424fd4          0     425004     425044     425130
  3    426e88     426f04     427170     426e34     427228     427254     426ffc
  4    415f40          0     415f40     416000     42b76c     42b69c     416074
  5    41626c     41642c     416288     416520     4165d0     416670     416710
  6   4f4ef3c          0    4f4ef3c    4f4efe0    4f4ef84    4f4efac    4f4efd4
value = 0 = 0x0
/*再次检查系统中安装的设备驱动程序，发现内存字符型设备驱动程序安装成功*/
-> d 0x04f4ee60
04f4ee60:  0000 0000 0000 0000 8955 83e5 f83d f4ef   *........U...=...*
04f4ee70:  0004 047e c031 46eb d468 f4ef 6804 efac   *..~.1..Fh....h..*
04f4ee80:  04f4 8468 f4ef 6804 efe0 04f4 3c68 f4ef   *..h....h....h<..*
04f4ee90:  6a04 6800 ef3c 04f4 f3e8 4d25 83fb 1cc4   *.j.h<.....%M....*
/*查看目标机内存下信息*/
-> devs
drv name
  0 /null
  1 /tyCo/0
  3 host:
```

```
  4 /vio
  5 /tgtsvr
value = 0 = 0x0
/*查看目标中已经创建的设备*/
-> memDevCreate ("/memIoDevice",0x04f4ee60,10)
value = 0 = 0x0
/*添加内存字符型 IO 设备;把从内存地址 0x 0x04f4ee60 开始的 10 个字节作为字符型 IO 设备*/
-> devs
drv name
  0 /null
  1 /tyCo/0
  3 host:
  4 /vio
  5 /tgtsvr
  6 /memIoDevice
value = 0 = 0x0
/*再次查看目标中已经创建的设备，发现设备安装成功*/
-> iosFdShow
 fd name                 drv
  3 /tyCo/0                1
  4 /vio/1                 4
/*查看文件描述符表*/
-> fd=open("/memIoDevice",2,0)
_fd = 0x440e60: value = 5 = 0x5
/*打开已经创建的内存字符型 IO 设备*/
-> iosFdShow
 fd name                 drv
  3 /tyCo/0                1
  4 /vio/1                 4
  5 /memIoDevice           6
value = 0 = 0x0
/*再次查看文件描述符表，发现设备成功打开*/
-> yy="hello\n"
new symbol "yy" added to symbol table.
yy = 0x4f4ee28: value = 83160624 = 0x4f4ee30 = yy + 0x8
-> printf yy
hello
value = 6 = 0x6
-> write fd,yy,10
value = 10 = 0xa
/*将地址 yy 处存放的字符串写到内存字符型 IO 设备中去*/
-> d 0x04f4ee60
04f4ee60:  6568 6c6c 0a6f 0000 8955 83e5 f83d f4ef   *hello...U...=...*
04f4ee70:  0004 047e c031 46eb d468 f4ef 6804 efac   *..~.1..Fh....h..*
```

```
04f4ee80:  04f4 8468 f4ef 6804 efe0 04f4 3c68 f4ef   *..h....h....h<..*
value = 0 = 0x0
/*再次查看地址 04f4ee60，发现字符串已经成功地写到了内存字符型 IO 设备中*/
-> zz=calloc(10,1)
new symbol "zz" added to symbol table.
zz = 0x4f4ee20: value = 85125544 = 0x512e9a8
-> printf zz
value = 0 = 0x0
-> read fd,zz,10
value = 10 = 0xa
/*将内存字符型 IO 设备中的字符串读出来*/
-> printf zz
hello
value = 6 = 0x6
/*打印 zz，读出的字符串与写入的字符串完全相同，证明内存字符型 IO 设备访问成功*/
```

11 系统初始化流程

11.1 bootrom 介绍

Bootrom image 一般在嵌入式系统开发初期和调试时使用，它预先烧录在嵌入式目标板的程序存储器如EPROM 或者FLASH 中，当用户完成所有应用程序的开发和调试时，用户将bootrom 的代码和实时操作系统RTOS 和用户应用程序APP 编译连接生成最后的成品image，然后将其烧录到原来bootrom 所在的FLASH 中，并覆盖bootRom 代码。通常bootrom 包含最少的系统初始化,主要用于启动装载VxWorks image ， 有压缩和不压缩两种形式， 如bootrom 和boot_uncmp。与VxWorks image 的区别在于Bootrom调用bootConfig.c,而VxWorks调用usrConfig.c，本章就只叙述bootrom的执行顺序。。

11.2 bootrom 的执行逻辑

bootrom 中程序的函数的执行顺序是：文件romInit.s 中的romInit()---->文件bootInit.c 中的romStart()---->文件bootConfig.c 中的usrInit()----->sysHwInit()----->usrKernelInit()----->KernelInit(usrRoot,...) （其中/target/config/all/bootConfig.c 是Boot ROM 设置模块.用于通过网络加载VxWorksimage）---->usrRoot()---->bootCmdLoop(void)命令行选择,或autobooting----->bootLoad(pLine, &entry)加载模块到内存(网络,TFFS,TSFS...)----->netifAttach()---->go(entry)----->(entry)()从入口开始执行,不返回。

11.3 各个函数的主要作用

romInit()-----power up,disable interrupt,put boot type on the stack,clears caches.
romStart()-----load Image Segments into RAM.
usrInit()-----Interrupt lock out,save imformation about boot type,handle all the Initialization before the kernel is actually started,then starts the kernel execution to create an initial task usrRoot().This task completes the start up.
sysHwInit()-----Interrup locked,Initializes hardware,registers,activation kernel
KernelInit(usrRoot,...)-----

- Initializes and starts the kernel.
- Defines system memory partition.
- Activates a task tUsrRoot to complete initalization.
- Unlocks interrupts.
- Uses usrInit()stack.

usrRoot()

- 初始化内存分区表(memory partition library)
- 初始化系统系统时钟(system clock)
- 初始化输入输出系统(I/O system)----可选
- Create devices----可选
- 设置网络(Configure network)--------可选
- 激活WDB 目标通信(Activate WDB agent)---------可选
- 调用程序(Activate application)

11.4 bootrom 的创建

缺省的BootRom image 包含在tornado 提供的bsp 中，它被配置为支持网络开发环境。BootRom image 由最小的vxworks 配置和一个bootloader 机制组成。当发生以下情况时，你需要重新配置并建立一个新的BootRom 程序：

◇ 要工作的目标机不在网络上。

◇ 目标机没有target，每次启动不想在目标控制台上键入boot 参数时。

◇ 想使用一种替代的boot 过程，例如从目标服务文件系统（TSTF）启动时。

为支持目标板的开发环境的boot 参数，用户必须编辑**c:\tornado\target\config*bspname*\config.h**（**bsp** 的配置文件）。这个文件包括**DEFAULT_BOOT_LINE** 定义，该行包括以下一些参数：boot 设备、主机和目标机的IP 地址、要装入的vxworks image 的路径和文件名等等。

```
# define DEFAULT_BOOT_LINE \
"cpm(0,0)host:vxworks h=192.168.10.14 e=192.168.10.104 u=target pw=target "
```

在目标板启动后打印出来的信息如下图所示

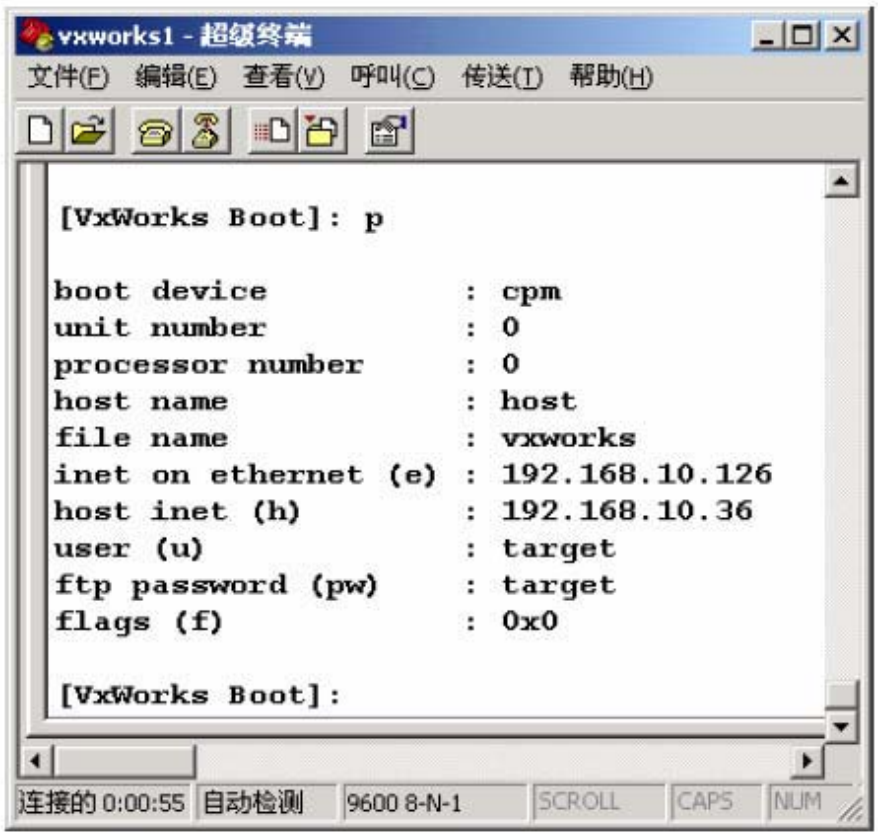


图11－1 超级终端打印出bootrom启动参数

为建立（Build）新的bootrom，从tornado 图形界面选择Build>Build Boot ROM。选择要建立的bootrom 的bsp，并在Build Boot ROM 对话框中选择要建立的bootrom 的类型（这里有三种类型的boot image 可供选择：bootrom，压缩的boot iamge；bootrom_uncmp ，未压缩的boot image；bootrom_res，驻留在ROM 中的boot image。），然后单击OK。

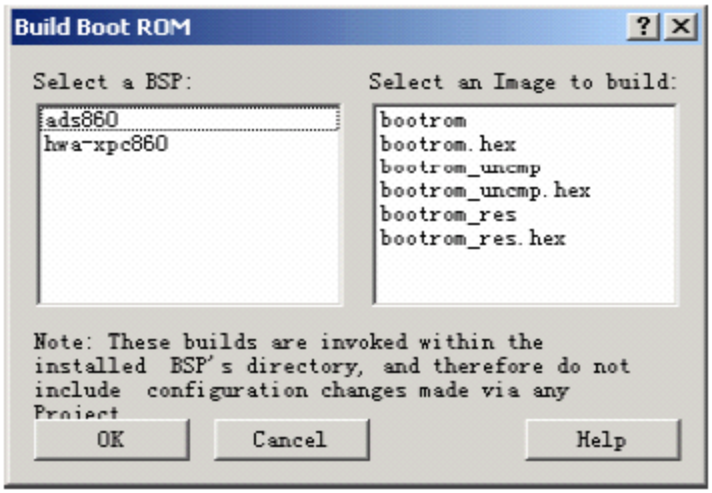


图11－2 编译连接生成bootrom

11.5 装载 vxWorks image 的过程

bootrom images 重定位VxWorks image，使用网络接口、串口、或SCSI disk 等等，boot参数存储在NVRAM，boot参数在boot时可以修改。对bootrom images来说，bootConfig.c 取代usrConfig.c：*usrRoot()* 将发起一个任务*tBoot*，它将重定位可装载的image（loadable image）。文件包含支持例程管理boot参数、image重定位，以及与用户交互。BSP开发者一般不需要修改bootConfig.c，如果目标板没有NVRAM，boot 参数在config.h被指定，并静态地链接到vxWorks。

任务*tBoot* 的入口点是***bootCmdLoop()*** ：

如果用户没有在规定的时间内（超时参数是**TIMEOUT**，它在**bootConfig.c**中定义。）按键，系统将根据NVRAM 中的参数调用***autoboot()***自动装载（autoboot）

对交互式会话，代码将交互式地循环，直到boot继续命令被发布。

重定位loadable image的例程是bootLoad（）：

◇ boot参数通过参数列表被传送。

◇ 为装载标识loadble vxworks image合适的设备接口。

◇ 在装载设备时执行必要的初始化

◇ 装载image。

当bootLoad（）返回处理器时，跳到被装载的image的入口处，sysInit（）。在romStart（）跳到usrInit（）后的执行顺序：

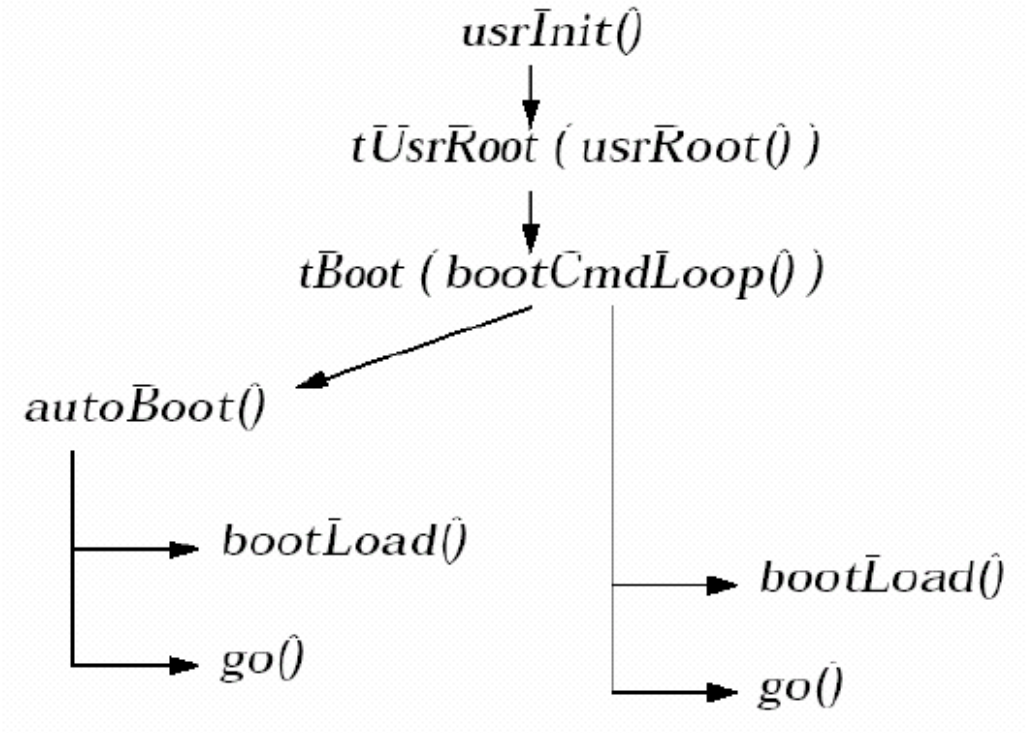


图11－3 bootrom 部分函数执行顺序

其中tUsrRoot的入口点是usrRoot（），tBoot的入口点是bootCmdLoop。

标准bootrom的bootstrap部分拷贝自身和它的数据到RAM中的RAM_LOW_ADRS处，并在冷启动时清零RAM中的其他地址，接着解压它主要的boot程序从ROM中到RAM中的RAM_HIGH+ADRS处。完成这些之后，跳到解压缩的boot

程序的入口地址处。这时RAM中的bootstrap部分就不需要了。

标准的bootrom在内存中的执行顺序如下图所示：

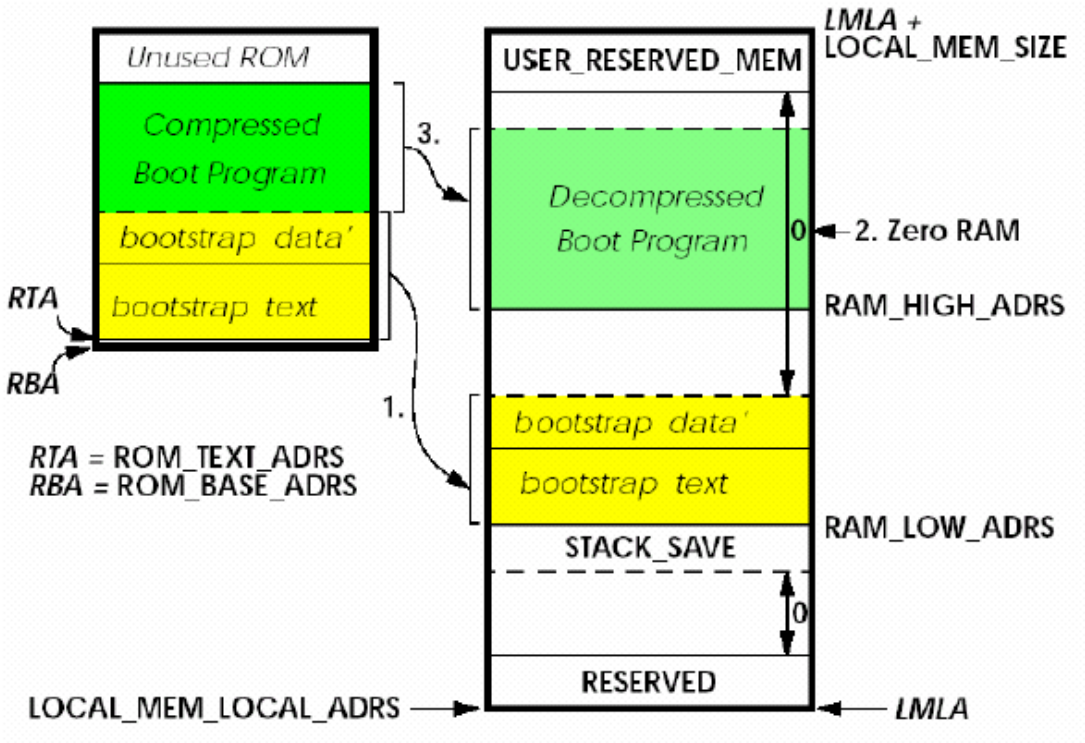


图11－4 标准bootrom在内存中的执行顺序

其中**LOCAL_MEM_LOCAL_ADRS**是RAM的起始地址；**RAM_LOW_ADRS** 是VxWorks的装入入口点。它也是VxWorks text段的起始地址。**FREE_RAM_ADRS** 标明VxWorks image的结束地址。 通常是系统内存池或target server内存池的起始地址。**RAM_HIGH_ADRS** 是 boot 程序的装入点。它也是boot程序的代码段的起始地址；除非boot image是**ROM-resident**类型的，在这种情况下，他是boot模块的数据段的起始地址。

Downloable vxworks image（名为vxworks的映象文件）的下载与执行

这个vxworks image不包含bootstrap代码，bootstrap代码从ROM拷贝自身到RAM。要求一个单独的boot程序：从本地存储设备或者从网络上获取image；装载到RAM中的RAM_LOW_ADRS地址处执行。标准的vxworks boot程序是一个专门的vxworks应用程序，有三种类型：**bootrom**, **bootrom_uncmp**, and**bootrom_res**。

装载之后的vxworks执行逻辑如下：

执行config/bspName/sysALib.s 文件中_sysInit （与_romInit相似，屏蔽中断，禁止cache，设置初始堆栈指针）接着调用：prjConfig.c中的usrInit( ) 接下来执行prjConfig.c中的*usrRoot( )* 接下来执行用户应用程序。如图所示：

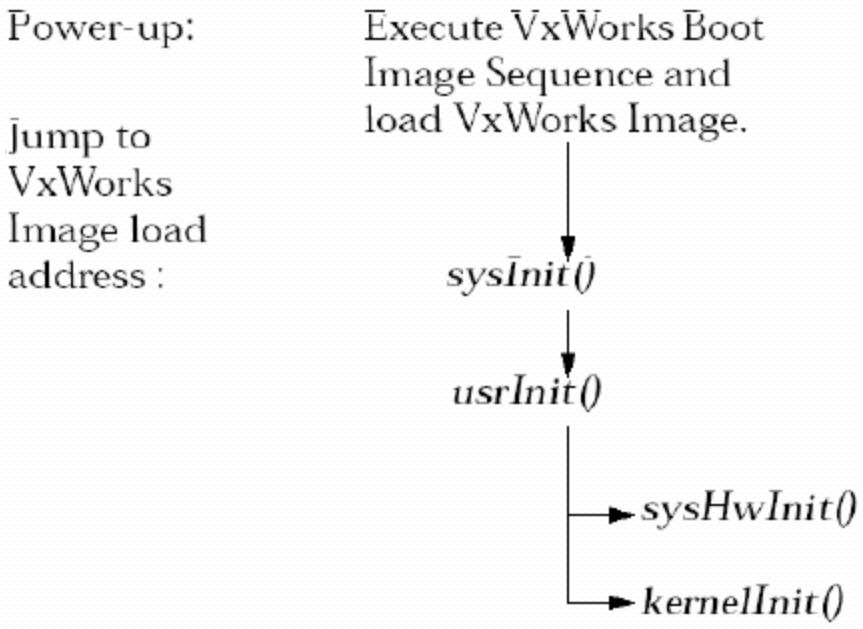


图11－5 vxWorks映象文件执行顺序

可装载的vxWorks image在内存中表示如下：

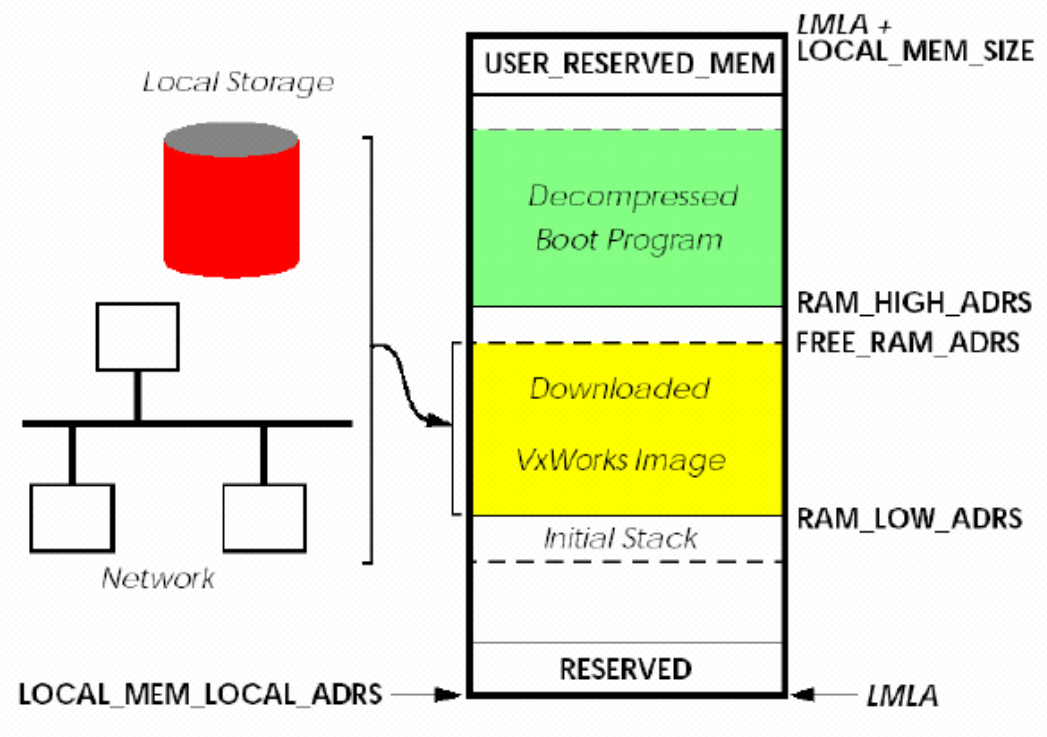


图11－6 vxWorks在内存中布局

图中RAM_LOW_ADRS和RAM_HIGH_ADRS在文件：wind/target/config/*bspName*/Makefile和wind/target/config/*bspName*/config.h中定义。

boot程序从NVRAM或者从用户输入获得boot参数，或者从DHCP或BOOTP服务器处或者使用缺省的boot参数。它接着从网络或本地存储设备如硬盘下载vxWorks。下载的vxworks装载在RAM中的RAM_LOW_ADRS，image

的顶部地址被称作**FREE_RAM_ADRS**。在下载完成之后，boot程序跳到位于**RAM_LOW_ADRS**处下载vxworks的入口s*ysInit()*处。从这点boot程序的代码和数据将被覆盖。

12 实验要求

12.1 实验目的

本课程设计教学所要达到的目的是：通过基本实验和综合实验训练，使学生理解嵌入式实时操作系统 VxWorks 的工作原理，掌握使用 Torando 软件开发嵌入式系统的过程。同时，由于设计中涉及到硬件微处理器的使用和 C 语言等知识，可以起到培养学生综合运用各种知识和技术解决问题的能力。

本实验课程的成绩主要以学生在实验过程中的实际表现为主，结合实验报告撰写情况予以评定，分为优良中差四个等级。

12.2 基本实验

本课程实验涵盖了嵌入式实时操作系统的几乎所有的软件设计部分，本实验课程涉及到的主要内容已在前面各章节中做了描述，内容如下：

1、多任务创建、管理与调度
2、任务间的通信机制
3、任务间的同步机制
4、任务间的互斥机制
5、IO 设备驱动程序
6、中断驱动程序
7、定时器驱动程序
8、嵌入式集成开发环境及各种工具使用(即嵌入式软件的交叉编译与交叉调试)

根据实验教学任务要求，安排 4 次基本实验，1 次综合实验。每次基本实验为 4 个学时，四次基本实验内容如下：

第一次实验内容为：

1、 嵌入式系统的开发流程：基于 ppc860(或者 s3c4510 实验板)和 VxWorks 的嵌入式系统软件开发流程，以 HelloWorld 程序设计与启动为例。
2、 调试多任务创建程序理解多任务调度算法(优先级调度和时间片轮转调度算法)。
3、 任务间通信：使用消息队列和管道机制实现两个任务间的通信，调试提供的参考程序，改写为客户机/服务器模型。

第二次实验内容为：

1、 多任务间互斥操作，理解互斥信号量的使用，以及优先级翻转现象及优先级继承算法。
2、 任务间同步机制：使用二进制信号量实现任务间的同步。调试提供的参考程序，改写为单向同步，考虑同步丢失的现象，将二进制信号量换为计数式信号量观察程序执行结果。

第三次实验内容为：

1、 中断服务程序及中断控制器初始例程设计：理解 ISR 的设计约束，以及中断控制器的初始化，调试提供的参考实验例程。
2、 定时器驱动程序设计：掌握看门狗定时器的使用方法(实现代码周期执行或死限任务处理)，理解辅助时钟定时器的工作原理，掌握定时器驱动程序设计方法及应用程序中使用定时器的方法。

第四次实验内容为：

1、 IO 设备驱动程序设计，使用标准的 IO 操作函数(如 open，ioctl 等)控制发光二极管的亮/灭。
2、 嵌入式系统交叉编译-makefile 文件的编写与使用。
3、 嵌入式系统的交叉调试-browser,windsh,debuger 工具使用。

12.3 综合实验

综合实验主要用于检查学生能否熟练使用嵌入式软硬件开发工具完成功能较为复杂的嵌入式软件设计；检查对他们对嵌入式实时操作系统工作原理的理解与掌握情况，对他们的学习能力，综合开发能力，创新能力做出评价，评定实验最终成绩。

综合实验内容主要包括以下两个方面：

第一个方面，检查基本实验掌握情况，重点检查以下两个基本实验

1、 嵌入式系统开发流程

2、 嵌入式系统交叉编译(makefile 文件编写)

第二个方面，从以下综合实验题目(不限于这几个题目，每年都可能随机增加新的综合实验题目)中任选一个题目，在短时间内完成该实验。综合实验题目主要有以下几个：

1、 编写程序，实现以下功能

系统包含三个任务，一个初始化任务(tInitTask)，一个生产者任务(tSendTask),一个消费者任务(tReceiveTask);

要求 tSendTaskt 与 ReceiveTask 之间使用消息队列传递数据；

要求 tInitTask 能设置 tSendTask 和 tReceiveTask 的优先级，

要求 tInitTask 能设置传递消息的数量和消息队列的大小。

2、 编写程序，实现以下功能

要求系统具有两个任务，两个任务之间使用二进制信号和计数式信号量实现同步

当在 windSh 下

执行 semTest ‘b’ 使用二进制信号量实现任务之间的同步

执行 semTest ‘c’ 使用计数式信号量实现任务之间的同步

3、 编写程序，实现以下功能

使用二进制信号量实现辅助时钟定时器服务程序与任务之间的同步；

当按下 R2 按钮(irq5 中断按钮)时，停止定时器工作。

4、 编写程序，实现以下功能

当执行 timerTestStart ‘w’时，使用看门狗定时器实现信息的周期打印

当执行 timerTestStart ‘a’时，使用辅助时钟实现信息的周期打印

当执行 timerTestStop ‘w’时，停止看门定时器

当执行 timerTestStop ‘a’时， 停止辅助时钟

5、 编写程序，实现以下功能

系统有两个任务 tTaskA,tTaskB；

当 tTaskA 执行时点亮发光二极管 LED3，

当 tTaskB 执行时点亮发光二极管 LED4；

当按下 R2,熄灭 LED3 与 LED4，并删除掉 tTaskA 与 tTaskB

6、 编写程序，实现以下功能

系统有两个任务，tClientTask 和 tServerTask；

它们之间使用消息队列实现任务间的通信，tClientTask 发送消息(函数名，如 add 和参数)，tServerTask 接收消息，在服务器任务上执行函数(如实现加法运算)，将执行的结果通过另一个消息队列返回给 tClientTask 任务，tClientTask 打印结果。

7、 编写程序，实现以下功能

编写辅助时钟驱动程序(1 分钟中断一次)，每隔一分钟点亮一次发光二极管，点量发光二极管要求用字符型 IO 设备驱动程序实现(如使用 ioctl 函数，或者 write 函数)。

8、 编写程序实现以下功能：

系统有两个任务，分别为 tTaskA 和 tTaskB：当 tTaskA 执行时，发光二极管 LED3 点亮 5 秒，当 tTaskB 执行时，发光二极管 LED4 点亮 5 秒；点亮发光二极管均使用设备驱动程序(如使用 ioctl 函数，或者 write 函数)，时间 5s 使用 taskDelay 函数实现，要求独占访问发光二极管(使用互斥信号量实现)。

9、 编写程序实现以下功能：

要求系统中有三个任务，tTaskA 和 tTaskB，以及 tTaskC，要求实现独立型同步或关联型同步；(可使用二进制信号量或者事件机制)

10、用二进制信号量和互斥信号量实现任务间的互斥操作，（并且实现优先级继承和反转）

12.4 实验设备

实验室需要每组一台电脑（安装有 Tornado 软件），实验板一套（hwa860 实验板一块、5V 直流电源一个、串口线一根、交叉网线一根），见附录 A。

12.5 实验步骤

具体的实验步骤请参考本书第 13 页 3.3 节(使用Tornado开发嵌入式软件的流程)。

12.6 实验报告要求

实验报告的撰写要符合重庆大学研究生院所规定的格式；实验报告书要求包含以下内容：

1、 实验进度安排
2、 实验目的
3、 实验内容
4、 实验步骤
5、 实验源程序
6、 实验结果
7、 实验分析

附录 A　实验板

A.1 概述

HWA-XPC860 板卡采用当今最流行的 32 位嵌入式 CPU－Motorola 公司的 MPC860 设计的。本板卡给用户提供一个基本的硬件环境，通过单独使用本板卡，用户可以更深入了解 32 位 CPU 的性能。为自己独立开发使用 MPC860 打下一个很好的基础。软件设计人员通过对本板卡的使用，从而对当今最流行的嵌入式实时操作系统 Vxworks 有一个更深入的了了解；硬件设计人员和地层程序编写人员可以通过的对本板卡的调试从而更加明了在 Tonado 的环境写如何编写 BSP 文件和编写驱动程序。从而摆脱 X86 的环境，减少程序移植的时间。

A.2 基本配置

本板的基本配置如下：

CPU:	XPC860
工作频率：	50MHz
Flash 大小：	4M 字节
内存大小：	32M 字节 SDRAM
串口：	两个 RS-232 串口 COM1 和 COM2
网口：	一个 10M 以太网接口（标配）;
	一个 10M/100M 自适应以太网接口（需用户指定）;
扩展接口：	4 个 PMC 接头构成一个外部扩展接口
电源接口：	一个+5VDC 电源接口；
调试接口：	一个 BDM 调试接口。

A.3 硬件结构

HWA-XPC860 板卡的硬件结构在本章节的各分节中分别描述。

A.3.1系统框图

HWA-XPC860 的系统结构如图附录-1 所示

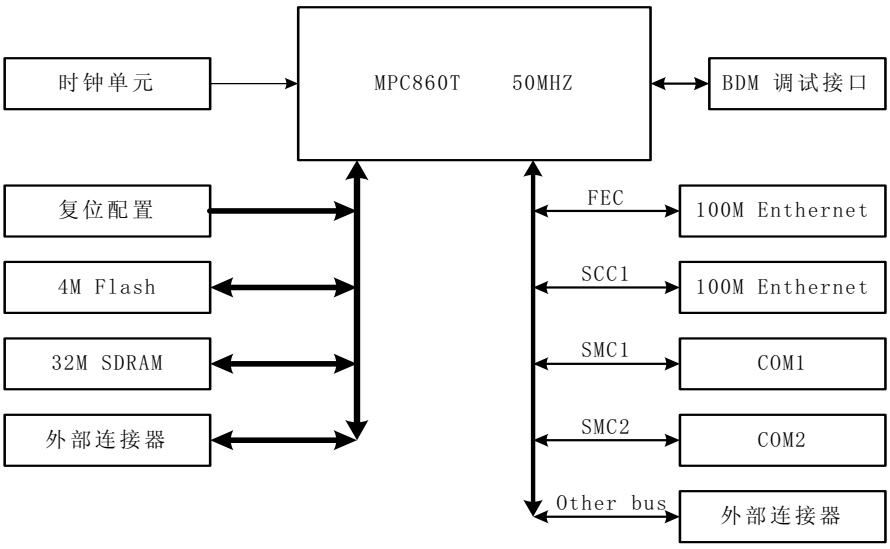


图 A-1　HWA-XPC860 板卡的系统结构框图

A.3.2内存分配

表附录-1　HWA-XPC860 板卡内存分配表

起始地址	结束地址	功能	片选信号
0x00000000	0x20000000	外部 SDRAM，内存空间	CS1
0xFF000000	0xFF010000	内部寄存器空间	无
0xFFC00000	0xFFFFFFFF	外部 Flash，程序存储空间	CS0

A.3.3系统时钟

本板卡的工作时钟为 50MHz，由外部一片钟振提供给 MPC860 芯片，输出频率 CLKOUT=50MHz；具体电路如图附录-2 所示

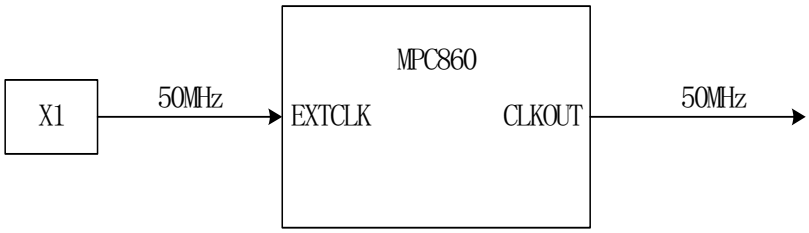


图 A-2　系统时钟单元框图

A.3.4HWA-XPC860板卡器件位置框图

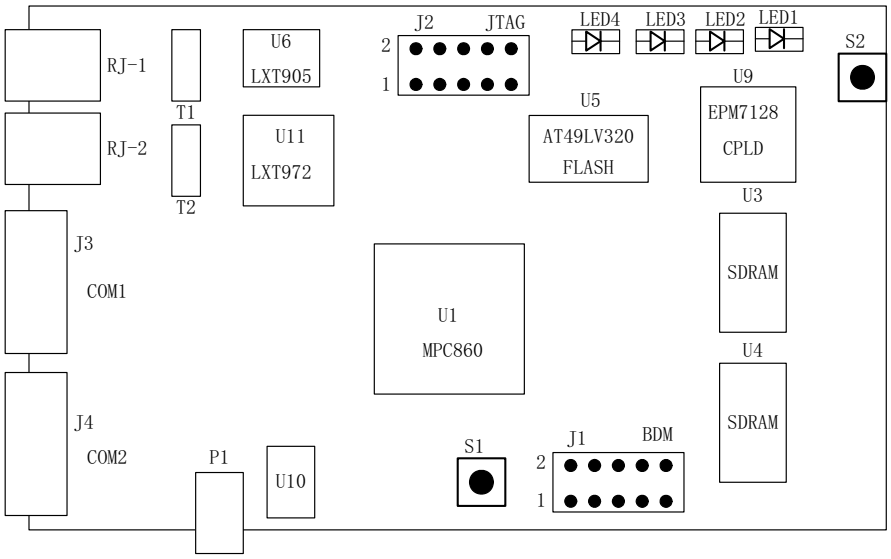


图 A-3　HWA-XPC860 板卡器件分布图

A.3.5硬件复位控制

S1 为复位控制按钮，按下 S1，硬件进入复位状态；松开 S1，硬件复位完成。S1 在板上的位置详见图 A-3。

A.3.6串口配置

HWA-XPC860 板卡提供两路串行接口，电气接口形式为 RS-232。本设计使用 SMC1 接口实现串口 COM1；SMC1 接口实现串口 COM2。

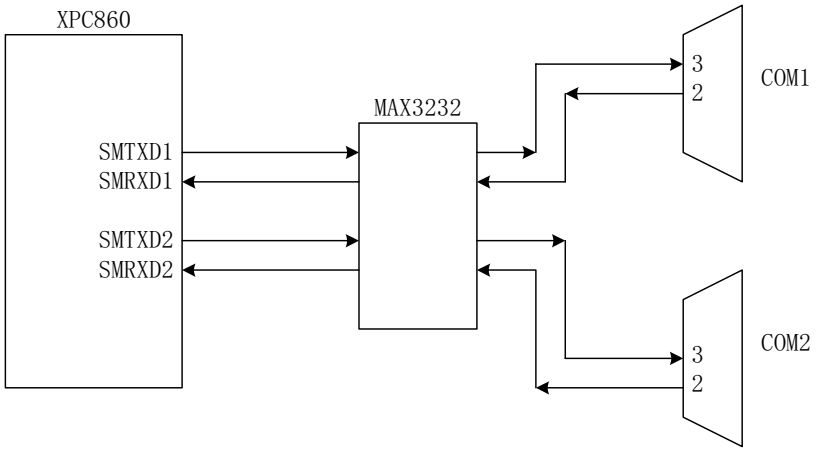


图 A-4　HWA-XPC860 串口逻辑图

当 HWA-XPC860 板的串口与 PC 机相连接时，请使用交叉串口线，交叉方法见图 A-5。

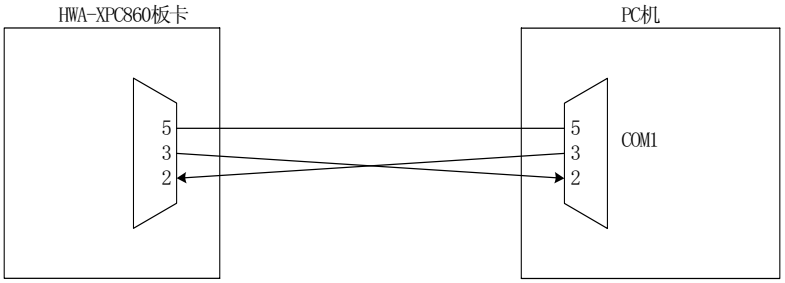


图 A-5　串口线交叉连接图

A.3.7网口配置

本板提供两路网络接口电路：10M 以太网和 10M/100M 自适应以太网接口。一般情况下，除非客户指定，本板不焊接 10M/100M 自适应以太网接口电路元件，即不提供 10M/100M 以太网功能。　10M 以太网接口由 MPC860 的 SCC1 实现，10M/100M 自适应以太网接口由 MPC860T 的 FEC 实现。接口接插件在板上分别表现为：

表 3-7　以太网接口说明

MPC860 接口	接口功能	接插件标识	接口样式
SCC1 接口	10M 接口	RJ-1	RJ-45 标准接插件
FEC 接口	10M/100M	RJ-2	RJ-45 标准接插件

A.3.8PMC扩展接口

HWA-XPC860 的对外接口由四个 PMC 接插件完成，各接插件的接口信号分别如表 3-8-1、表 3-8-2、表 3-8-3 和表 3-8-4 的说明。

表 3-8-1　J5 接口信号

针脚号	名称	说明	针脚号	名称	说明
1	D0	CPU 的 32 位数据信号线	2	D1	CPU 的 32 位数据信号线
3	D2		4	D3	
5	D4		6	D5	
7	D6		8	D7	

9	D8		10	D9	
11	D10		12	D11	
13	D12		14	D13	
15	D14		16	D15	
17	D16		18	D17	
19	D18		20	D19	
21	D20		22	D21	
23	D22		24	D23	
25	D24		26	D25	
27	D26		28	D27	
29	D28		30	D29	
31	D30		32	D31	
33	A30	CPU 的 32 位地址信号线	34	A31	CPU 的 32 位地址信号线
35	A28		36	A29	
37	A26		38	A27	
39	A24		40	A25	
41	A22		42	A23	
43	A20		44	A21	
45	A18		46	A19	
47	A16		48	A17	
49	A14		50	A15	
51	A12		52	A13	
53	A10		54	A11	
55	A8		56	A9	
57	A6		58	A7	
59	A4		60	A5	
61	A2		62	A3	
63	A0		64	A1	

表 3-8-2　J6 接口信号

针脚号	名称	说明	针脚号	名称	说明
1	IO0	与 CPLD 连接的 16 各 IO 信号。功能未定义。	2	IO1	与 CPLD 连接的 16 各 IO 信号。功能未定义。
3	IO2		4	IO3	
5	IO4		6	IO5	
7	IO6		8	IO7	
9	IO8		10	IO9	
11	IO10		12	IO11	
13	IO12		14	IO13	
15	IO14		16	IO15	
17	IRQ3	CPU 中断信号 IRQ5 没有接到 CPU 引脚上	18	IRQ4	CPU 中断信号 IRQ7 没有接到 CPU 引脚
19	IRQ5		20	IRQ6	

21	GND	电源地信号	22	IRQ7	
23	GND		24	NC	
25	PORESET	复位信号	26	HRESET	复位信号
27	SRESET	复位信号	28	GND	
29	NC		30	GND	
31	CS0	片选信号	32	CS3	片选信号
33	CS2		34	CS5	
35	CS4		36	CS7	
37	CS6		38	NC	
39	OE		40	R/W	
41	GND		42	GND	
43	BURST		44	BDIP	
45	TA		46	TEA	
47	TS		48	AS	
49	GND		50	BI	
51	GND		52	NC	
53	NC		54	NC	
55	WE0		56	WE1	
57	WE2		58	WE2	
59	NC		60	NC	
61	BS_A0		62	BS_A1	
63	BS_A2		64	BS_A3	

表 3-8-3　J7 接口信号

针脚号	名称	说明	针脚号	名称	说明
1	NC	这些信号未使用	2	TX+	10M 以太网接口信号。本信号为板上 10M 以太网变压器输出信号。
3	NC		4	TX-	
5	NC		6	RX+	
7	NC		8	RX-	
9	NC		10	NC	未使用
11	NC		12	NC	
13	NC		14	TX+	100M 以太网接口信号。本信号为板上 100M 以太网变压器输出信号。用户不用处理，直接接到 RJ-45 上就可以直接使用
15	NC		16	TX-	
17	NC		18	RX+	
19	NC		20	100-4	
21	NC		22	100-5	
23	NC		24	RX+	
25	NC		26	100-7	
27	NC		28	100-8	
29	NC		30	NC	这些信号未使用
31	NC		32	NC	

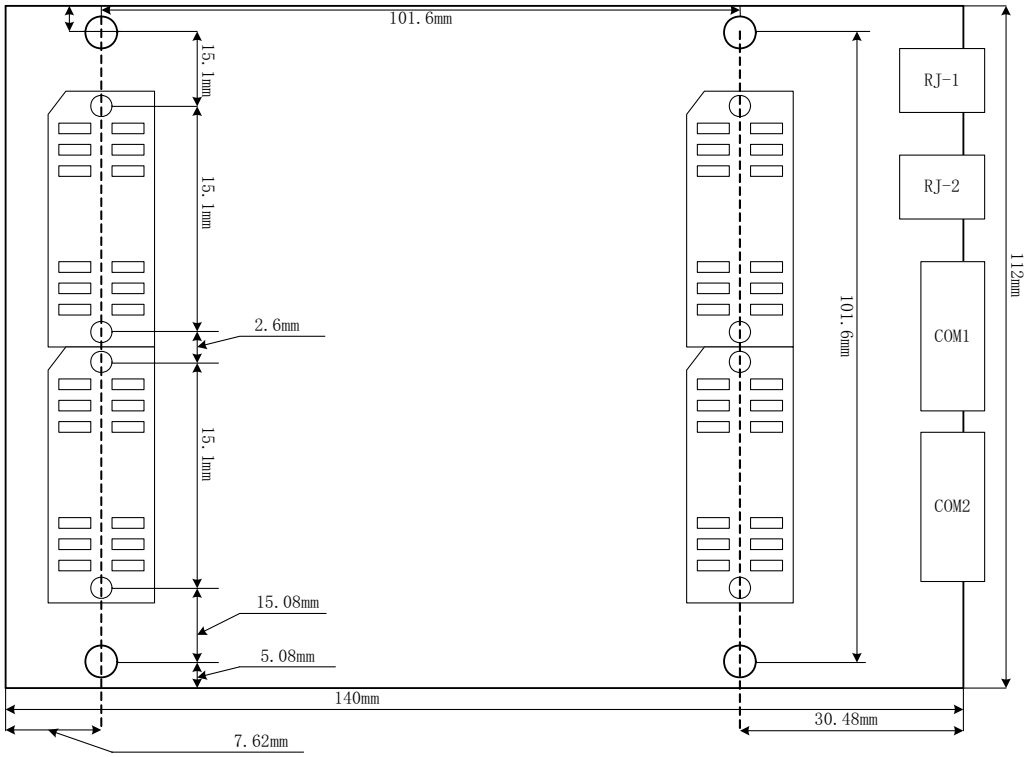
33	NC		34	NC	
35	NC		36	NC	
37	NC		38	NC	
39	NC		40	NC	
41	NC		42	NC	
43	NC		44	NC	
45	NC		46	NC	
47	NC		48	NC	
49	NC		50	NC	
51	NC		52	NC	
53	NC		54	NC	
55	NC		56	NC	
57	NC		58	NC	
59	NC		60	NC	
61	NC		62	NC	
63	NC		64	NC	

表 3-8-4　J8 接口信号

针脚号	名称	说明	针脚号	名称	说明
1	PA4	CPU 的 PA 接口信号信号。其中 PA4、PA6、PA14、PA15 用作 10M 以太网 PHY 控制信号。若不使用以太网接口可以用作一般 IO。	2	PA0	CPU 的 PA 接口信号
3	PA5		4	PA1	
5	PA6		6	PA2	
7	PA7		8	PA3	
9	PA8		10	PC4	CPU 的 PC 接口信号
11	PA9		12	PC5	
13	PA10		14	PC6	
15	PA11		16	PC7	
17	PA12		18	PC8	
19	PA13		20	PC9	
21	PA14		22	PC10	
23	PA15		24	PC11	
25	PB15	PB 端口信号	26	PC12	
27	PB14		28	PC13	
29	PB19		30	PC14	
31	PB18		32	PC15	
33	PB17		34	PB21	PB 端口信号
35	PB16		36	PB20	
37	PB22		38	PB24	
39	PB23		40	PB25	
41	PB26		42	IP_B7	IP_B 端口信号
43	PB27		44	IP_B6	

45	PB28		46	IP_B5	
47	PB29		48	IP_B4	
49	PB30		50	IP_B3	
51	PB31		52	IP_B2	
53	GND	电源地信号	54	GND	电源地信号
55	GND		56	GND	
57	NC	未定义	58	NC	未定义
59	NC		60	NC	
61	VCC	+5VDC 电源信号	62	VCC	+5VDC 电源信号
63	VCC		64	VCC	

A.3.9PMC接插件位置图



A.3.10BDM调试接口

CPU 的 BDM 调试接口标号为 J1。JI 接口信号的描述如下表

表 3-9　　BDM 接口信号列表

信号名称	引脚位置
VFLS0	1
RESET	2
GND	3
DSCK	4
GND	5
VFLS1	6
HRESET	7
DSDI	8

+3.3V	9
DSDO	10

A.3.11CPLD的JTAG接口信号

CPLD 的 JTAG 接口用来配置 U9－EPM7128AETC1001-10 的，JTAG 配置接口的标号为 J2。其信号接口的信号描述如下表所示。

表 3-10　BDM 接口信号列表

信号名称	引脚位置
TDI	1
GND	2
TDO	3
+3.3V	4
TMS	5
NC	6
NC	7
NC	8
TDK	9
GND	10

A.3.12中断测试

为了使用户更充分的了解了解 MPC860 的 CPU 性能，我们在板上设计了一个中断触发按钮 S2，同时使用指示灯 LED2 指示按钮的动作。指示灯的位置如图 A-3 所示。具体实现的硬件框图如图 A-11 所示。

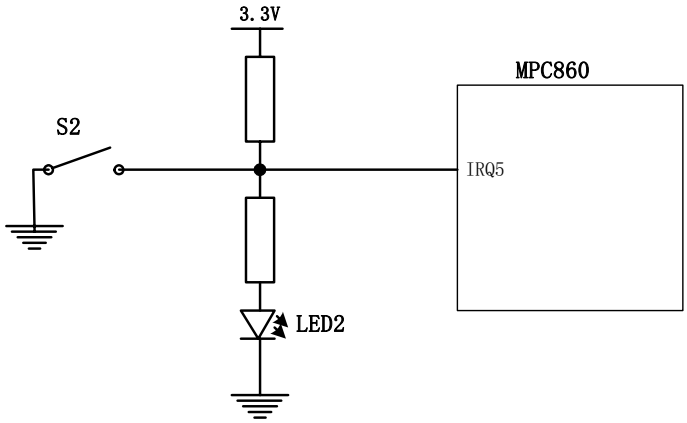


图 A-11　中断测试功能实现硬件逻辑示意图

当 S2 按下时，指示灯 LED2 灭，同时 CPU 的 IRQ5 产生一个中断。当放开 S2 时，指示灯 LED2 点亮。CPU 不产生中断。注意在本设计中为了防止 S2 按下的时间过长，编写程序时，请把中断 5 的触发方式设计为下降沿触发。

A.3.13IO功能测试

为了使用户对 MPC860 的 IO 信号功能有一个了解，我们提供了 IO 测试指示灯。PB22 和 PB23 信号为高电平时，指示灯 LED3 和 LED4 分别点亮；PB22 和 PB23 信号为低电平时，指示灯 LED3 和 LED4 分别灭；具体的硬件实现方法如图 A-12 所示。

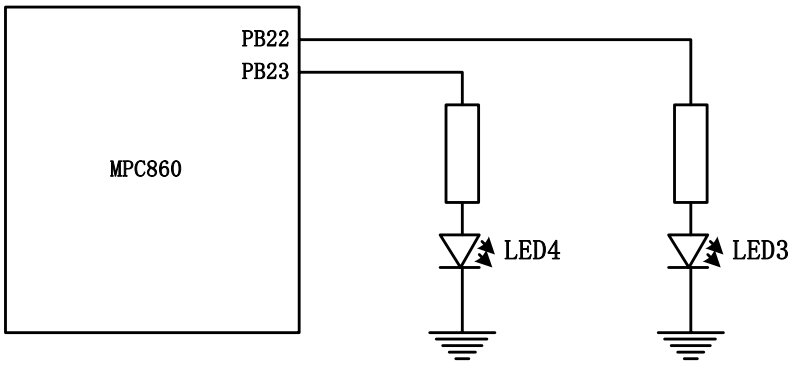


图 A-12　I/O 功能实现硬件逻辑示意图

A.3.14电源接口

HWA-XPC860 的电源接口为外部+5VDC,其具体的电源参数如下：

表 3-13 电源参数

项目	参数
输入电压	＋5VDC
电流	2A
接口形式	插孔

电源插孔的接线形式为：外壁为电源地，内芯为电源。具体示意如图 A-13 所示。

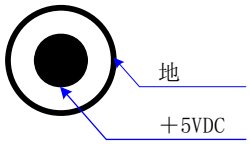


图 A-13　HWA-XPC860 板卡电源接口连接说明示意图